Practice Session Notes: Theory of Algorithms

Viktória Nemkin (nemkin@cs.bme.hu)

May 25, 2024

These, are my practice session notes, they are based on my personal notes and on the Hungarian course's solutions, found here under the heading **"A gyakorlatokon használt feladatsorok"**:
http://www.cs.bme.hu/~csima/algel2020/index_2020_tavasz_vegleges.html → In the case of hashings, this guide uses open hashings with quadratic probing by starting from the left, while we start that one to the right. Check Session 13, Exercise 9 for details!

This is a work in progress, I am updating these whenever I have time.

Mistakes are possible, if you spot any or if you have any questions about these notes, please contact me at nemkin@cs.bme.hu or in MS Teams.

Version log: https://github.com/nemkin/thalg-notes/commits/main - So you can check what was updated and when. (Or send me a PR if you find any errors? :))

Contents

1 February 16th (Session 1): O, Ω, Θ, pattern matching **6**
1.1 Session 1, Exercise 1 . . . 6
1.2 Session 1, Exercise 2 . . . 7
1.3 Session 1, Exercise 3 . . . 9
1.4 Session 1, Exercise 4 . . . 11
1.5 Session 1, Exercise 5 . . . 13
1.6 Session 1, Exercise 6 . . . 14
1.7 Session 1, Exercise 7 . . . 15
1.8 Session 1, Exercise 8 . . . 18
1.9 Session 1, Exercise 9 . . . 19
1.10 Session 1, Exercise 10 . . . 20
1.11 Session 1, Exercise 11 . . . 21
1.12 Session 1, Exercise 12 . . . 22

2 February 23rd (Session 2): Finite Automata **23**
2.1 Session 2, Exercise 1 . . . 23
2.2 Session 2, Exercise 2 . . . 24
2.3 Session 2, Exercise 3 . . . 26
2.4 Session 2, Exercise 4 . . . 27
2.5 Session 2, Exercise 5 . . . 29
2.6 Session 2, Exercise 6 . . . 30
2.7 Session 2, Exercise 7 . . . 36
2.8 Session 2, Exercise 8 . . . 39
2.9 Session 2, Exercise 9 . . . 40
2.10 Session 2, Exercise 10 . . . 41
2.11 Session 2, Exercise 11 . . . 45

2.12 Session 2, Exercise 12 . . . 46
2.13 Session 2, Exercise 13 . . . 47

3 March 2nd (Session 3): Regular expressions, Context-free languages 48
3.1 Session 3, Exercise 1 . . . 48
3.2 Session 3, Exercise 2 . . . 49
3.3 Session 3, Exercise 3 . . . 50
3.3.1 Even number of 0's . . . 50
3.3.2 Odd number of 0's . . . 50
3.3.3 A perfect square number of 0's . . . 51
3.3.4 A power of 2 . . . 51
3.4 Session 3, Exercise 4 . . . 52
3.5 Session 3, Exercise 5 . . . 53
3.5.1 Words of odd lengths . . . 53
3.5.2 Words of even length that start and end with 1 . . . 53
3.5.3 Words of odd length that start and end with 1 . . . 53
3.5.4 Words containing at least three 0's . . . 53
3.5.5 Words containing an even number of 0's . . . 54
3.5.6 Words of odd lengths starting with 0 and words of even length starting with 1 . . . 54
3.5.7 Words of odd length containing subword 00 . . . 54
3.6 Session 3, Exercise 6 . . . 55
3.7 Session 3, Exercise 7 . . . 56
3.8 Session 3, Exercise 8 . . . 57
3.9 Session 3, Exercise 9 . . . 58
3.9.1 Contains 011 as a substring . . . 58
3.9.2 Starts with 1, ends with 0 . . . 58
3.9.3 Is of even length . . . 58
3.10 Session 3, Exercise 10 . . . 59
3.11 Session 3, Exercise 11 . . . 60
3.12 Session 3, Exercise 12 . . . 61

4 March 9th (Session 4): Context-free grammars, Pushdown automata 62
4.1 Sessions 4 and 5, Exercise 1 . . . 62
4.2 Sessions 4 and 5, Exercise 2 . . . 63
4.3 Sessions 4 and 5, Exercise 3 . . . 65
4.4 Sessions 4 and 5, Exercise 4 . . . 67
4.5 Sessions 4 and 5, Exercise 5 . . . 68
4.6 Sessions 4 and 5, Exercise 6 . . . 70
4.7 Sessions 4 and 5, Exercise 7 . . . 71
4.8 Sessions 4 and 5, Exercise 8 . . . 72
4.9 Sessions 4 and 5, Exercise 9 . . . 73
4.10 Sessions 4 and 5, Exercise 10 . . . 74
4.11 Sessions 4 and 5, Exercise 11 . . . 75
4.12 Sessions 4 and 5, Exercise 12 . . . 76

5 March 23th (Session 5): Turing-machines 77

5.1 Session 5, Exercise 1 . . . 77
5.2 Session 5, Exercise 2 . . . 78
5.3 Session 5, Exercise 3 . . . 79
5.4 Session 5, Exercise 4 . . . 81
5.5 Session 5, Exercise 5 . . . 82
5.6 Session 5, Exercise 6 . . . 84
5.7 Session 5, Exercise 7 . . . 85
5.8 Session 5, Exercise 8 . . . 87
5.9 Session 5, Exercise 9 . . . 88

6 March 30th (Session 6): Turing-machines 2 **89**
6.1 Session 6, Exercise 1 . . . 89
6.2 Session 6, Exercise 2 . . . 90
6.3 Session 6, Exercise 3 . . . 91
6.4 Session 6, Exercise 4 . . . 92
6.5 Session 6, Exercise 5 . . . 93
6.6 Session 6, Exercise 6 . . . 94
6.7 Session 6, Exercise 7 . . . 95
6.8 Session 6, Exercise 8 . . . 96

7 April 6th (Session 7): P, NP **97**
7.1 Session 7, Exercise 1 . . . 97
7.2 Session 7, Exercise 2 . . . 99
7.3 Session 7, Exercise 3 . . . 100
7.4 Session 7, Exercise 4 . . . 101
7.5 Session 7, Exercise 5 . . . 103
7.6 Session 7, Exercise 6 . . . 104
7.7 Session 7, Exercise 7 . . . 105
7.8 Session 7, Exercise 8 . . . 106
7.9 Session 7, Exercise 9 . . . 107

8 April 13th (Session 8): NP-completeness **108**
8.1 What is a Karp-reduction? . . . 108
8.2 Session 8, Exercise 1 . . . 111
8.3 Session 8, Exercise 2 . . . 114
8.4 Session 8, Exercise 3 . . . 115
8.5 Session 8, Exercise 4 . . . 116
8.6 Session 8, Exercise 5 . . . 119
8.7 Session 8, Exercise 6 . . . 120
8.8 Session 8, Exercise 7 . . . 122
8.9 Session 8, Exercise 8 . . . 124
8.10 Session 8, Exercise 9 . . . 125
8.11 Session 8, Exercise 10 . . . 126
8.12 Session 8, Exercise 11 . . . 127
8.13 Session 8, Exercise 12 . . . 128

9 April 27th (Session 9): NP-completeness, Integer Programming, Linear Programming **129**
9.1 Session 9, Exercise 1 . . . 129
9.2 Session 9, Exercise 2 . . . 133
9.3 Session 9, Exercise 3 . . . 134
9.4 Session 9, Exercise 4 . . . 135
9.5 Session 9, Exercise 5 . . . 136
9.6 Session 9, Exercise 6 . . . 137
9.7 Session 9, Exercise 7 . . . 138
9.8 Session 9, Exercise 8 . . . 139
9.9 Session 9, Exercise 9 . . . 140
9.10 Session 9, Exercise 10 . . . 141

10 May 5th (Session 10): Binpacking FF/FFD, Dynamic Programming **142**
10.1 Session 10, Exercise 1 . . . 142
10.2 Session 10, Exercise 2 . . . 147
10.3 Session 10, Exercise 3 . . . 149
10.4 Session 10, Exercise 4 . . . 150
10.5 Session 10, Exercise 5 . . . 152
10.6 Session 10, Exercise 6 . . . 158
10.7 Session 10, Exercise 7 . . . 161
10.8 Session 10, Exercise 7 . . . 163
10.9 Session 10, Exercise 8 . . . 165
10.10 Session 10, Exercise 9 . . . 166
10.11 Session 10, Exercise 10 . . . 167
10.12 Session 10, Exercise 11 . . . 169

11 May 11th (Session 11): Sorting, Binary Search Trees **170**
11.1 Session 11, Exercise 1 . . . 170
11.2 Session 11, Exercise 2 . . . 172
11.3 Session 11, Exercise 3 . . . 173
11.4 Session 11, Exercise 4 . . . 174
11.5 Session 11, Exercise 5 . . . 175
11.6 Session 11, Exercise 6 . . . 176
11.7 Session 11, Exercise 7 . . . 178
11.8 Session 11, Exercise 8 . . . 179
11.9 Session 11, Exercise 9 . . . 180
11.10 Session 11, Exercise 10 . . . 181
11.11 Session 11, Exercise 11 . . . 182
11.12 Session 11, Exercise 12 . . . 193
11.13 Session 11, Exercise 13 . . . 194
11.14 Session 11, Exercise 14 . . . 195
11.15 Session 11, Exercise 15 . . . 196

12 May 18th (Session 12): Binary Search Trees, 2-3-trees, Hash **197**
12.1 Solutions written by Prof. Attila Sali, on Moodle . . . 197
12.2 Session 12, Exercise 9 . . . 198

13 Exam: 2019. 01. 06. **200**
13.1 Exams: 2019. 01. 06., Exercise 1 . . . 200

14 Exam: 2019. 06. 17. **204**
14.4 Exams: 2019. 06. 17., Exercise 4 . . . 204

15 Exam: 2020. 01. 23. **209**
15.3 Exam: 2020. 01. 23., Exercise 3 . . . 209

16 Exam: 2022. 05. 30. **211**
16.1 Exam: 2022. 05. 30., Exercise 1 . . . 211
16.2 Exam: 2022. 05. 30., Exercise 2 . . . 212
16.3 Exam: 2022. 05. 30., Exercise 3 . . . 213
16.4 Exam: 2022. 05. 30., Exercise 4 . . . 221
16.5 Exam: 2022. 05. 30., Exercise 5 . . . 222
16.6 Exam: 2022. 05. 30., Exercise 6 . . . 224
16.7 Exam: 2022. 05. 30., Exercise 7 . . . 225

17 Exam: 2022. 06. 13. **226**
17.1 Exam: 2022. 06. 13., Exercise 1 . . . 226
17.2 Exam: 2022. 06. 13., Exercise 2 . . . 227
17.3 Exam: 2022. 06. 13., Exercise 3 . . . 228
17.4 Exam: 2022. 06. 13., Exercise 4 . . . 229
17.5 Exam: 2022. 06. 13., Exercise 5 . . . 231
17.6 Exam: 2022. 06. 13., Exercise 6 . . . 232

18 Exam: 2022. 06. 20. **233**
18.1 Exam: 2022. 06. 20., Exercise 1 . . . 233
18.2 Exam: 2022. 06. 20., Exercise 2 . . . 234
18.3 Exam: 2022. 06. 20., Exercise 3 . . . 235
18.4 Exam: 2022. 06. 20., Exercise 4 . . . 237
18.5 Exam: 2022. 06. 20., Exercise 5 . . . 238
18.6 Exam: 2022. 06. 20., Exercise 6 . . . 239
18.7 Exam: 2022. 06. 20., Exercise 7 . . . 241

1 February 16th (Session 1): O, Ω, Θ, pattern matching

1.1 Session 1, Exercise 1

Exercise

There are two algorithms for the same problem, A and B. Let the functions describing their worst case running times be f_A and f_B. It is known that $f_A(n) = O(f_B(n))$. Does it follow that...

a.) A is faster on every input than B?

b.) except for finitely many inputs A is faster than B?

c.) for large enough inputs A is faster than B?

Solution

The answer is **no**, in all three cases.

A few good counterexamples:

- $f_A(n) = 2n$ and $f_B(n) = n$. Even though $f_A(n) \in O(f_B(n))$, actually $f_A(n) > f_B(n)$, so A is worst case slower than B.
- Any functions where $f_a(n) = f_b(n) = f(n)$, since $f(n) \in O(f(n))$.
- For just exercise *a.*), $f_A(n) = n$ and $f_B(n) = \frac{n^2}{4}$, since $n \in O(\frac{n^2}{4})$, however for $n = 1, 2, 3$ $f_A(n) > f_B(n)$.

1.2 Session 1, Exercise 2

Exercise

Which of the following functions are $O(n^2)$ and which are $\Omega(n^2)$?

a.) $f_1(n) = 11n^2 + 100000$

b.) $f_2(n) = 8n^2 \log_2(n)$

c.) $f_3(n) = 1.5n + 3\sqrt{n}$

Solution

Definitions:

- $f(n) \in O(g(n))$ if there exists $c > 0$ and $n_0 \in \mathbb{Z}^+$ so that $f(n) \leq cg(n)$ for $\forall n \geq n_0$.
- $f(n) \in \Omega(g(n))$ if there exists $c > 0$ and $n_0 \in \mathbb{Z}^+$ so that $f(n) \geq cg(n)$ for $\forall n \geq n_0$.

To prove that something is $\in O(g(n))$ or $\in \Omega(g(n))$, give one c and n_0 and show that the above requirement is holds true.

To prove that something is not $\in O(g(n))$ or $\in \Omega(g(n))$, prove that no possible c and n_0 combination would make the above requirement hold true.

Fist look at the given function, and figure out whether you think the definition is true or not and then based on that choose the appropriate proof.

a)

$f(n) = 11n^2 + 100000$.

Gut feeling: The most significant component here is n^2, so we believe it is asymptotically similar to n^2, so both $f(n) \in O(n^2)$ and $f(n) \in \Omega(n^2)$.

First let's show that $f(n) \in O(n^2)$:

Start from the function and give **upper** bounds until we reach something in the form of const $* n^2$.

$f(n) = 11n^2 + 10^5 \leq 12n^2$ if $n^2 \geq 10^5$, or $n \geq 10^{5/2}$, where we can be generous and just say $n \geq 1000$, since it is not necessary to find the smallest possible n_0, only one sufficient.

We have arrived at: $f(n) \leq 12n^2$ if $n \geq 1000$.

Then by choosing $n_0 = 1000$ and $c = 12$, this turns into $f(n) \leq cn^2$ if $n \geq n_0$, which by definition means that $f(n) \in O(n^2)$.

Now let's show that $f(n) \in \Omega(n^2)$:

Start from the function again, however now we give **lower** bounds until we reach something in the form of const $* n^2$.

$f(n) = 11n^2 + 10^5 \geq 11n^2$, since $10^5 \geq 0$.

We have arrived at: $f(n) \geq 11n^2$. Now $c = 11$, and there was no requirement for n, so we can choose $n_0 = 1$, and then this turns into $f(n) \geq cn^2$ if $n \geq n_0$, which by definition means that $f(n) \in \Omega(n^2)$.

Note:

- Since we have shown that $f(n) \in O(n^2)$ and also $f(n) \in \Omega(n^2)$, the combination of the two is that $f(n) \in \Theta(n^2)$, so $11n^2 + 100000$ and n^2 are asymptotically similar.

b)

$f(n) = 8n^2 \log_2(n)$

Gut feeling: This is going to be asymptotically bigger than n^2, since it is also multiplied with $\log_2(n)$. So we believe $f(n) \in \Omega(n^2)$, however $f(n) \notin O(n^2)$.

Let's start with showing that $8n^2 \log_2(n) \notin O(n^2)$:

In this case we want to show that there is no possible choice of c and n_0, which is done using indirect proof and arriving at a contradiction.

Let's indirectly state that $8n^2 \log_2(n) \in O(n^2)$, which is equivalent to saying:

There exists some $c > 0$ and $n_0 \in \mathbb{Z}^+$ constants, so that

$8n^2 \log_2(n) \leq cn^2$ for $\forall n \geq n_0$.

Let's divide the left and the righthand side with n^2:

$8 \log_2(n) \leq c$ for $\forall n \geq n_0$.

Let's move the 8 to the right:

$\log_2(n) \leq \frac{c}{8}$ for $\forall n \geq n_0$.

And this is a contradiction, since $\frac{c}{8}$ is a constant, while $\log_2(n)$ can be arbitrarily large for a given $n \geq n_0$, and we cannot give an upper bound to something arbitrarily large with a constant.

This means that the indirect statement was wrong so that $8n^2 \log_2(n) \notin O(n^2)$.

Now let's show that $f(n) \in \Omega(n^2)$:

Start from the function again and give **lower** bounds until we reach something in the form of const $* n^2$.

$f(n) = 8n^2 \log_2(n) \geq 8n^2$ when $n \geq 2$, since $\log_2(n) \geq 1$ when $n \geq 2$.

We have arrived at $f(n) \geq 8n^2$ when $n \geq 2$, so we can choose $c = 8$ and $n_0 = 2$, which gives $f(n) \geq cn^2$ when $n \geq n_0$, which is the definition of $f(n) \in \Omega(n^2)$.

c)

$f(n) = 1.5n + 3\sqrt{n}$

Gut feeling: This is going to be asymptotically smaller than n^2, since the most significant component is only n. So we believe $f(n) \in O(n^2)$, however $f(n) \notin \Omega(n^2)$.

Let's show that $f(n) \in O(n^2)$:

Start from the function and give **upper** bounds until we reach something in the form of const $* n^2$. (Actually, in this case, an even stonger statement will be made.)

$f(n) = 1.5n + 3\sqrt{n} \leq 4.5n$, since $\sqrt{n} \leq n$ (for $n \geq 1$, which is a given).

We have arrived at $f(n) \leq 4.5n$, which means that $c = 4.5$ and since there is no lower bound on n, we can choose $n_0 = 1$, to turn it into $f(n) \leq cn$ for $\forall n \geq n_0$, which means by definition that $f(n) \in O(n)$.

Now since we know that $O(n) \subset O(n^2)$, this also proves that $f(n) \in O(n^2)$.

Let's continue with showing that $1.5n + 3\sqrt{n} \notin \Omega(n^2)$:

In this case we want to show that there is no possible choice of c and n_0, which is done using indirect proof and arriving at a contradiction.

Let's indirectly state that $1.5n + 3\sqrt{n} \in \Omega(n^2)$, which is equivalent to saying:

There exists some $c > 0$ and $n_0 \in \mathbb{Z}^+$ constants, so that

$1.5n + 3\sqrt{n} \geq cn^2$ for $\forall n \geq n_0$.

How we handle this situation is by giving something that is larger than $1.5n + 3\sqrt{n}$ and show that even this larger thing still cannot upper bound cn^2, so the smaller, original function can not either:

Borrowing from the previous proof, we know that $4.5n \geq 1.5n + 3\sqrt{n}$:

$4.5n \geq 1.5n + 3\sqrt{n} \geq cn^2$ for $\forall n \geq n_0$.

Now let's just look at the two sides:

$4.5n \geq cn^2$ for $\forall n \geq n_0$.

Let's divide both sides by n:

$4.5 \geq cn$ for $\forall n \geq n_0$.

And move the constant to the left:

$\frac{4.5}{c} \geq n$ for $\forall n \geq n_0$.

And here is the contradiction: this can not be true, since n is arbitrarily large, while it should be upper bound by a constant: $\frac{4.5}{c}$. This shows that $4.5n$ can not be an upper bound for cn^2, so an even smaller function, $1.5n + 3\sqrt{n}$ definitely can't be either for any given c and n_0 pair.

1.3 Session 1, Exercise 3

Exercise

For which integers $a, b > 1$ do the following hold true?

a.) $n^a = O(n^b)$

b.) $2^{an} = O(2^{bn})$

c.) $\log_a(n) = O(\log_b(n))$

Solution

a)

$n^a = O(n^b)$ holds true if there exists $c > 0$ and $n_0 \in \mathbb{Z}^+$, such that $n^a \leq cn^b$, for $\forall n \geq n_0$.

Then by moving n^b to the left, we get:

$\frac{n^a}{n^b} \leq c$, for $\forall n \geq n_0$.

Or

$n^{a-b} \leq c$, for $\forall n \geq n_0$.

Now, depending on the exponent n^x can behave very differently as n grows:

- For $x > 0$ n^x grows arbitrarily large, and cannot be upper bound with a constant.
- For $x = 0$, $n^0 = 1$ is a constant and can be upper bound.
- For $x < 0$, $n^x = n^{-|x|} = \frac{1}{n^{|x|}}$, where while $n^{|x|}$ grows arbitrarily large $\frac{1}{n^{|x|}} \to \frac{1}{\infty} = 0$. So for a large enough n, we can upper bound this with a constant. (For example $x^{-2} \leq \frac{1}{4}$ for $\forall n \geq 2$.)

To sum up, if $x = a - b \leq 0$, or simply when $a \leq b$, then $n^a = O(n^b)$ holds true, otherwise not.

b)

$2^{an} = O(2^{bn})$ holds true if there exists $c > 0$ and $n_0 \in \mathbb{Z}^+$, such that $2^{an} \leq c2^{bn}$, for $\forall n \geq n_0$.

Again, let's move 2^{bn} to the left:

$\frac{2^{an}}{2^{bn}} \leq c$, for $\forall n \geq n_0$.

Or

$2^{(a-b)n} \leq c$, for $\forall n \geq n_0$.

Again, depending on the exponent's multiplier 2^{xn} can behave very differently as n grows:

- If $x > 0$, then 2^{xn} grows to infinity, or can be arbitrarily large.
- If $x = 0$, then $2^{0n} = 1$, which can be upper bound with a constant.
- If $x < 0$, then $2^{xn} = 2^{-|x|n} = \frac{1}{2^{|x|n}}$, where while $2^{|x|n}$ grows arbitrarily large $\frac{1}{2^{|x|n}} \to \frac{1}{\infty} = 0$. So for a large enough n, we can upper bound this with a constant. (For example $2^{-n} \leq 1$ for $\forall n \geq 1$.)

c)

$\log_a(n) = O(\log_b(n))$ holds true if there exists $c > 0$ and $n_0 \in \mathbb{Z}^+$, such that $\log_a(n) \leq c \log_b(n)$, for $\forall n \geq n_0$.

Looking at this we already know that since a is the base of a logarithm, it must be positive, $a > 0$ and $a \neq 1$ (the base of a logarithm cannot be 1). Similarly $b > 0$ and $b \neq 1$.

Let's move $log_b(n)$ to the left:

$\frac{\log_a(n)}{\log_b(n)} \leq c$, for $\forall n \geq n_0$.

Here, we can use the change of base rule for logarithms:

$\log_a(n) = \frac{\log_b(n)}{\log_b(a)}$.

Let's substitute this into the previous formula:

$\frac{\frac{\log_b(n)}{\log_b(a)}}{\log_b(n)} \leq c$, for $\forall n \geq n_0$.

Now, since both the numerator and denominator contains $\log_b(n)$, we can reduce with it, to arrive at:

$\frac{\frac{1}{\log_b(a)}}{1} \leq c$, for $\forall n \geq n_0$.

And just leave the outer denominator:

$\frac{1}{\log_b(a)} \leq c$, for $\forall n \geq n_0$.

Now if the original statements are true ($a > 0$, $a \neq 1$, $b > 0$, $b \neq 1$), that means that the number $\frac{1}{\log_b(a)}$ exists and is a constant (since a and b are also constants), which can be upper estimated with a positive c, any c works.

1.4 Session 1, Exercise 4

Exercise

Put the following functions in order of non-decreasing order of magnitude, i.e., if f_i is immediately followed by f_j, then $f_i(n) = O(f_j(n))$ holds.

a.) $f_1(n) = 8n^3$

b.) $f_2(n) = 5\sqrt{n} + 1000n$

c.) $f_3(n) = 2^{(\log_2 n)^2}$

d.) $f_4(n) = 1514n^2 \log_2 n$

Solution

Gut feeling: Firstly, we just look at the functions and what we see if we squint is:

a.) $f_1(n) \approx n^3$

b.) $f_2(n) \approx n$

c.) $f_3(n)$ well this one is complicated.

d.) $f_4(n) \approx n^2 \log_2 n$

Let's work on $f_3(n)$ a bit:

$f_3(n) = 2^{(\log_2 n)^2} = 2^{(\log_2 n)(\log_2 n)} = (2^{(\log_2 n)})^{(\log_2 n)} = n^{\log_2 n}$.

Now if we put this back in:

a.) $f_1(n) \approx n^3$

b.) $f_2(n) \approx n$

c.) $f_3(n) = n^{\log_2 n}$

d.) $f_4(n) \approx n^2 \log_2 n$

The order should be something along the lines of $n << n^2 \log_2 n << n^3 << n^{\log_2 n}$, since multiplying n^2 by $log_2 n$ results in asymptotically a bit bigger than n^2, but not enough to reach n^3 and $n^{log_2 n}$ has a non-constant exponent, while n^3's exponent is constant, so between the two, $n^{log_2 n}$ is asymptotically bigger.

Plugging back the function's names, we need to prove that $f_2(n) << f_4(n) << f_1(n) << f_3(n)$. This can be done by proving these three things:

- $f_2(n) \in O(f_4(n))$
- $f_4(n) \in O(f_1(n))$
- $f_1(n) \in O(f_3(n))$

And the transitivity of O means that all relations hold true.

Let's begin with $f_2(n) \in O(f_4(n))$:

$5\sqrt{n} + 1000n \in O(1514n^2 \log_2 n)$:

We need to find c and n_0 so that $5\sqrt{n} + 1000n \leq c1514n^2 \log_2 n$ for $\forall n \geq n_0$.

Let's work both sides individually:

$5\sqrt{n} + 1000n \leq 1005n$, since $\sqrt{n} \leq n$ when $n \geq 1$.

$c1514n^2 \log_2 n \geq c1514n^2$, since $\log_2 n \geq 1$ when $n \geq 2$.

If we put them together:

$5\sqrt{n} + 1000n \leq 1005n \leq c1514n^2 \leq c1514n^2 \log_2 n$ when $n \geq 2$.

Just the middle part:

$1005n \leq c1514n^2$ when $n \geq 2$.

Divide by n and move the constants around:

$\frac{1005}{c1514} \leq n$ when $n \geq 2$.

We can just set $c = 1$, which means that $\frac{1005}{1514} \leq n$ must hold true, and it does, because we already have $n \geq 2$ as a requisite. So $n_0 = 2$ and $c = 1$ works.

Let's continue with $f_4(n) \in O(f_1(n))$:

$1514n^2 \log_2 n \in O(n^3)$, since the righthand side is already pretty clean we can just work on the lefthand side.

$1514n^2 \log_2 n \leq 1514n^3$ since $\log_2 n \leq n$, for any n. Then we can choose $c = 1514$ and $n_0 = 1$.

Finally let's show that $f_1(n) \in O(f_3(n))$:

$n^3 \in O(n^{\log_2 n})$

$n^3 \leq cn^{\log_2 n}$.

Let's just set $c = 1$, the functions are easier to handle and work on finding an appropriate n_0.

When $n \geq 1$ for a given n this holds true if $3 \leq \log_2 n$ which is true when $2^3 \leq n$, or $8 \leq n$. So $c = 1$ and $n_0 = 8$ works.

1.5 Session 1, Exercise 5

Exercise

Give O bounds for the following functions:

a.) $(n^2 + 8)(n + 1)$

b.) $(n \log n + n^2)(n^3 + 2)$

c.) $(n! + 2^n)(n^3 + \log(n^2 + 1))$

d.) $(2^n + n^2)(n^3 + 3^n)$

Solution

a)

$(n^2+8)(n+1) = n^3+n^2+8n+1 \leq n^3+n^3+8n^3+n^3 = 11n^3$, when $n \geq 1$. So $c = 11$ and $n_0 = 1$ works for $\in O(n^3)$.

b)

$(n \log n + n^2)(n^3 + 2) \leq (n^2 + n^2)(n^3 + 2)$ since $\log n \leq n$ (and every component is positive here).

$(n^2 + n^2)(n^3 + 2) \leq (n^2 + n^2)(n^3 + 2n^3)$ since $1 \leq n^3$, when $1 \leq n$ (and every component is positive here).

$(n^2 + n^2)(n^3 + 2n^3) = 2n^2\ 3n^3 = 6n^5$. So $c = 6$ and $n_0 = 1$ works for $\in O(n^5)$.

c)

$(n! + 2^n)(n^3 + \log(n^2 + 1)) \leq (n! + 2^n)(n^3 + n^2 + 1)$, because $log(x) \leq x$.

$(n! + 2^n)(n^3 + n^2 + 1) \leq (n! + 2^n)(n^3 + n^3 + n^3) = (n! + 2^n)\ 3n^3$.

For $n!$ and 2^n, we can argue that $n! = 1 * 2 * 3 * ... * n$, while $2^n = 2 * 2 * 2 * ... * 2$, both containing n factors. While $1 < 2$, the other factors of $n!$ are greater than or equal to 2. So eventually, for a large enough n, $2^n \leq n!$ must hold true.

n	2^n	$n!$
1	2	1
2	4	2
3	8	6
4	16	24

We can see that for $n \geq 4$ $n! \geq 2^n$.

$(n! + 2^n)\ 3n^3 \leq 2n!3n^3 = 6n^3n!$, for $n \geq 4$. So $c = 6$ and $n_0 = 4$ works for $O(n^3n!)$.

d)

$(2^n + n^2)(n^3 + 3^n)$

We know that the exponential function grows faster than the polinomial.

$(2^n + n^2)(n^3 + 3^n) \leq (2^n + 2^n)(n^3 + 3^n)$ since $2^n \geq n^2$ when $n \geq 4$.

$(2^n + 2^n)(n^3 + 3^n) \leq (2^n + 2^n)(3^n + 3^n)$ since $3^n \geq n^3$ when $n \geq 4$.

$(2^n + 2^n)(3^n + 3^n) = 2 * 2^n 3 * 3^n = 6 * 6^n$, so $c = 6$ and $n_0 = 4$ works for $O(6^n)$.

1.6 Session 1, Exercise 6

Exercise

Let $f_1(n) = 1.5n!$ and $f_2(n) = 200(n-1)!$. Which of the following is true?

a.) $f_1 = O(f_2)$

b.) $f_2 = O(f_1)$

c.) $f_1 = \Omega(f_2)$

d.) $f_2 = \Omega(f_1)$

Solution

$n! = n(n-1)!$, so we can already see that $f_1(n) \approx nf_2(n)$.
Let's prove that $f_2 \in O(f_1)$:
$f_2(n) = 200(n-1)! \leq 200n(n-1)! = 200n! = \frac{200}{1.5}1.5n! = \frac{200}{1.5}f_1(n)$. So $c = \frac{200}{1.5}$ and $n_0 = 1$ works.
Let's also prove that $f_1 \notin O(f_2)$, $1.5n! \notin O(200(n-1)!)$.
Let's assume indirectly that $1.5n! \in O(200(n-1)!)$.
That means that there exists $c > 0$ and $n_0 \in \mathbb{Z}^+$ such that
$1.5n! \leq c200(n-1)!$ for $\forall n \geq n_0$.
Let's use $n! = n(n-1)!$:
$1.5n(n-1)! \leq c200(n-1)!$ for $\forall n \geq n_0$.
Let's divide by $(n-1)!$:
$1.5n \leq c200$ for $\forall n \geq n_0$.
And move the constants to the right:
$n \leq c\frac{200}{1.5}$ for $\forall n \geq n_0$.
This is a contradiction, since we would want to upper bound n with a constant, however n can be arbitrarily large.
Thus the indirect statement is false, the original statement is true, so $f_1 \notin O(f_2)$.
Let's look at the questions now:

a.) $f_1 = O(f_2)$, we proved this to be false.

b.) $f_2 = O(f_1)$, we proved this to be true.

c.) $f_1 = \Omega(f_2)$, this is true, because $f_2 = O(f_1)$ implies that $f_1 = \Omega(f_2)$.

d.) $f_2 = \Omega(f_1)$, this is false, because this would imply $f_1 = O(f_2)$, which is false.

Note:

- It is very important to not only prove that $f_2 \in O(f_1)$, but also separately prove that $f_1 \notin O(f_2)$, since both $f_2 \in O(f_1)$ and $f_1 \in O(f_2)$ could be true, for example in the case of $f_1 = f_2$.
- $f \in O(g)$ implies that $g \in \Omega(f)$, since both definitions require us to come up with a c and an n_0 and if we used the same n_0s and $c_2 = \frac{1}{c_1}$ or the reciprocal of the constants we can quickly arrive at $g \in \Omega(f)$ from $f \in O(g)$.

1.7 Session 1, Exercise 7

Exercise

Let $L(n)$ denote the worst case running time of an algorithm on an input of length n. What can be said about the order of magnitude of function $L(n)$ if $L(1) = 2$ holds and for $n > 1$ we have:

a.) $L(n) = L(n-1) + 3$

b.) $L(n) = L(n-1) + 5$

c.) $L(n) = L(n-1) + 3n$

d.) $L(n) = 2L(n-1) + 3$

e.) $L(n) = L(\lceil \frac{n}{2} \rceil) + 3$

f.) $L(n) = L(\lceil \frac{n}{2} \rceil) + n^k$

g.) $L(n) = 2L(\lceil \frac{n}{2} \rceil) + 3$

h.) $L(n) = 4L(\lceil \frac{n}{2} \rceil) + 3$

(For exercises e)-h), we can assume that $n = 2^m$.)

What happens if instead of equality we have $\leq$ or $\geq$?

Solution

Let's start from the end: What happens if instead of equality we have $\leq$ or $\geq$?

When there is $\leq$, it means that we only have an upper estimate on $L(n)$, which means that we can only talk about O, however we can say nothing about Ω.

When there is $\geq$, it means that we only have a lower estimate on $L(n)$, which means that we can only talk about Ω, however we can say nothing about O.

a)

$L(n) = L(n-1) + 3 = L(n-2) + 3 + 3 = L(n-3) + 3 * 3 = ... = L(n-i) + i * 3.$

The base case is when $n - i = 1$, or $i = n - 1$.

$L(n-i) + i * 3 = L(n-(n-1)) + (n-1) * 3 = L(1) + 3(n-1) = 3n - 3 + 2 = 3n - 1.$

So $L(n) = 3n - 1$ for $n > 1$.

$3n - 1 \in \Theta(n)$ (proof is left to the reader).

b)

$L(n) = L(n-1) + 5 = L(n-2) + 5 + 5 = L(n-3) + 3 * 5 = ... = L(n-i) + i * 5.$

The base case is when $n - i = 1$, or $i = n - 1$.

$L(n-i) + i * 5 = L(n-(n-1)) + (n-1) * 5 = L(1) + 5(n-1) = 5n - 5 + 2 = 5n - 3.$

So $L(n) = 5n - 3$ for $n > 1$.

$5n - 3 \in \Theta(n)$ (proof is left to the reader).

c)

$L(n) = L(n-1) + 3n = L(n-2) + 3n + 3(n-1) = L(n-3) + 3 * (n + (n-1) + (n-2)) = ... = L(n-i) + 3 * (n + (n-1) + (n-2) + ... + (n-(i-1))).$

The base case is when $n - i = 1$, or $i = n - 1$.

$L(n-i)+3*(n+(n-1)+(n-2)+...+(n-(i-1))) = L(n-(n-1))+3*(n+(n-1)+(n-2)+...+(n-(n-1-1))) = L(1)+3*(n+(n-1)+(n-2)+...+2) = 2+3*(n+(n-1)+(n-2)+...+2) = 3*(n+(n-1)+(n-2)+...+2+1)-1 = \frac{n(n+1)}{2}-1.$

We can use the Gauss Summation: $n + (n-1) + (n-2) + ... + 2 + 1 = \frac{n(n+1)}{2}$

So $L(n) = \frac{n(n+1)}{2} - 1$ for $n > 1$.

$\frac{n(n+1)}{2} - 1 \in \Theta(n^2)$ (proof is left to the reader).

This is one of those cases when calculating the exact value of $L(n)$ is a bit tedious, however an upper estimate is much simple, which allows us to reason about O! (But not Ω, since that would require a lower estimate.)

$L(n) = L(n-1) + 3n = L(n-2) + 3n + 3(n-1) \leq L(n-2) + 3n + 3n = L(n-3) + 3n + 3n + 3(n-2) \leq L(n-3) + 3 * (3n) \leq L(n-i) + i * (3n)$.

The base case is when $n - i = 1$, or $i = n - 1$

$L(n-i) + i * (3n) \leq L(n-(n-1)) + (n-1) * (3n) = L(1) + (n-1) * (3n) = 3n * (n-1) + 2 = 3n^2 - 3n + 2$.

$L(n) \leq 3n^2 - 3n + 2$

Now, we can only use this to estimate O, since that would continue the upper estimation:

$L(n) \leq 3n^2 - 3n + 2 \leq 3n^2 + 2 \leq 3n^2 + 2n^2 = 5n^2$, so $c = 5$ and $n_0 = 1$ works for $O(n^2)$.

d)

$L(n) = 2L(n-1) + 3 = 2^2 L(n-2) + 2 * 3 + 3 = 2^3 L(n-3) + 2^2 * 3 + 2 * 3 + 3 = ... = 2^i L(n-i) + 3 * (2^{i-1} + ... + 2^0)$.

For $2^{i-1} + ... + 2^0$ we can use the formula for the sum of the geometric progression: $2^{i-1} + ... + 2^0 = \frac{2^i - 1}{2-1} = 2^i - 1$.

$2^i L(n-i) + 3 * (2^{i-1} + ... + 2^0) = 2^i * L(n-i) + 3 * 2^i - 3$.

The base case is when $n - i = 1$, or $i = n - 1$

$2^i * L(n-i) + 3 * 2^i - 3 = 2^{n-1} * L(n-(n-1)) + 3 * 2^{n-1} - 3 = 2^{n-1} * L(1) + 3 * 2^{n-1} - 3 = 2^{n-1} * 2 + 3 * 2^{n-1} - 3 = 5 * 2^{n-1} - 3 = \frac{5}{2} 2^n - 3$.

So $L(n) = \frac{5}{2} 2^n - 3$ for $n > 1$.

$\frac{5}{2} 2^n - 3 \in \Theta(2^n)$ (proof is left to the reader).

e)

We can assume that $n = 2^m$.

$L(2^m) = L(2^{m-1}) + 3 = L(2^{m-2}) + 3 + 3 = L(2^{m-3}) + 3 + 3 + 3 = ... = L(2^{m-i}) + 3 * i$.

The base case is when $2^{m-i} = 1$, or $m - i = 0$, or $i = m$.

$L(2^{m-i}) + 3 * i = ... = L(2^0) + 3 * m = L(1) + 3 * m = 3m + 2$.

$L(2^m) = 3m + 2$.

Now using that $n = 2^m$, we also know that $m = \log_2(n)$.

$L(n) = 3 \log_2(n) + 2$.

$3 \log_2(n) + 2 \in \Theta(\log_2(n))$ (proof is left to the reader).

f)

We can assume that $n = 2^m$. (This is going to be really tedious, but I will show a simpler solution after the complicated one. Don't worry, you won't have to do anything like this on the exam.)

$L(2^m) = L(2^{m-1}) + 2^{mk} = L(2^{m-2}) + 2^{(m-1)k} + 2^{mk} = L(2^{m-3}) + 2^{(m-2)k} + 2^{(m-1)k} + 2^{mk} = ... = L(2^{m-i}) + 2^{(m-(i-1))k} + ... + 2^{mk}$.

The base case is when $2^{m-i} = 1$, or $m - i = 0$, or $i = m$.

$L(2^{m-i}) + 2^{(m-(i-1))k} + ... + 2^{mk} = L(2^{m-m}) + 2^{(m-(m-1))k} + ... + 2^{mk} = L(1) + 2^{1k} + ... + 2^{mk} = 2 + 2^{1k} + ... + 2^{mk}$.

For $2^{1k} + ... + 2^{mk} = (2^k)^m + ... + (2^k)^1$, we can use the formula for the sum of the geometric progression again:

$(2^k)^m + ... + (2^k)^1 = (2^k)^m + ... + (2^k)^1 + (2^k)^0 - (2^k)^0$

$(2^k)^m + ... + (2^k)^1 + (2^k)^0 = \frac{(2^k)^{m+1} - 1}{2^k - 1}$

$(2^k)^m + ... + (2^k)^1 + (2^k)^0 - (2^k)^0 = \frac{(2^k)^{m+1} - 1}{2^k - 1} - (2^k)^0 = \frac{(2^k)^{m+1} - 1}{2^k - 1} - 1$.

So putting this back we arrive at:

$L(2^m) = 2 + \frac{(2^k)^{m+1} - 1}{2^k - 1} - 1 = \frac{(2^k)^{m+1} - 1}{2^k - 1} + 1$.

Now using that $n = 2^m$, we also know that $m = \log_2(n)$.

$L(n) = \frac{(2^k)^{\log_2(n)+1} - 1}{2^k - 1} + 1 = \frac{(2^{\log_2(n)+1})^k - 1}{2^k - 1} + 1 = \frac{(2n)^k - 1}{2^k - 1} + 1$. If we assume that k is a constant then $\frac{(2n)^k - 1}{2^k - 1} + 1 \in \Theta(n^k)$.

This was REALLY tedious. However, if we only want to reason about O, then we can use upper bounds to arrive at an O much easier:

$L(2^m) = L(2^{m-1}) + 2^{mk} \leq L(2^{m-2}) + 2^{mk} + 2^{mk} \leq L(2^{m-3}) + 2^{mk} + 2^{mk} + 2^{mk} \leq ... \leq L(2^{m-i}) + i * 2^{mk}$.

The base case is when $2^{m-i} = 1$, or $m - i = 0$, or $i = m$.

$L(2^{m-i}) + i * 2^{mk} \leq L(2^{m-m}) + m * 2^{mk} = L(1) + m * 2^{mk} = m * 2^{mk} + 2.$

$L(2^m) \leq m * 2^{mk} + 2.$

Now using that $n = 2^m$, we also know that $m = \log_2(n)$.

$L(n) \leq \log_2(n) * 2^{\log_2(n)k} + 2 = \log_2(n) * n^k + 2.$

$L(n) \leq \log_2(n) * n^k + 2$, and then $\log_2(n) * n^k + 2 \in O(\log_2(n) * n^k)$.

Yes, we did not arrive at the exact solution of $O(n^k)$, but $O(\log_2(n) * n^k)$. Our upper estimates in this case happened to break the boundary of the $O(n^k)$ set, since we upper estimated $\log_2(n)$ number of times with n^k. Sometimes it is better to arrive at a bit worse estimate quicker, than to have a more precise one, but suffer a lot of math for it.

1.8 Session 1, Exercise 8

Exercise

Using first Simple search, then Quick search determine how many times pattern $M = ABABC$ does occur in text $S = ABBABACABCBAC$. How many comparisons are made in each case?

Solution

Simple search: Looking for pattern ABABC.

# Comparisons	A	B	B	A	B	A	C	A	B	C	B	A	C
3	A	B	A										
1		A											
1			A										
4				A	B	A	B						
1					A								
2						A	B						
1							A						
3								A	B	A			
1									A				

Total: 17

Simple search tries all possible shifts = all possible positions for the pattern.

For a specific position it compares the pattern to the text above from left to right and moves to the next shift on the first non-matching character found.

If only the first occurrence is sought, the algorithm would stop at the first full match. When all occurrences are needed, the algorithm continues even after a full match. In this case, no match is found.

Jump function for pattern **ABABC:**

X	U[X]
A	3
B	2
C	1

The jump function is calculated as so:

1. Take the alphabet: (A,B,C) in this case.
2. If the given character is not in the pattern put length of pattern + 1, or 5+1=6. (This didn't happen in our case.)
3. If the given character count **backwards**, the index of the first occurrence. E.g. C is the last character of the pattern, so U[C]=1, B is the second-last character so U[B]=2, and A is the third-last character so U[A]=3.

Quick search: Looking for pattern ABABC.

# Comparisons	A	B	B	A	B	A	C	A	B	C	B	A	C	
3	A	B	A											
4				A	B	A	B							U[A]=3
2						A	B							U[B]=2
3								A	B	A				U[B]=2
1									A					U[C]=1

Total: 13

The Quick search uses the jump function to skip some positions that will definitely not be a match.

It looks at the upcoming character in the text (shown in orange) and checks its value of the jump function: it will shift the current position by that much.

For example in the first jump, the upcoming character in the text is A. U[A]=3, which means that the first shift that will result in a match on the upcoming character will be with shift A. We know that shifts of 1 and of 2 will not result in a match, since the jump function tells us that A is on the third position in the pattern when counting from backwards.

This is shown with light green background.

(Technically we could skip the comparison on the green position, but we don't, so make sure to count that as well!)

In the second jump, the last B would be positioned under the upcoming B character in the text, however we fail before we reach that comparison.

1.9 Session 1, Exercise 9

Exercise

Let both the pattern and the text consist of only 0's, let the length of the pattern be m, while that of the text be n. How many comparisons are done...

a.) by Simple search if only the first occurrence of the pattern is sought?

b.) by Simple search if all occurrences of the pattern are needed?

c.) by Quick search if only the first occurrence of the pattern is sought?

d.) by Quick search if all occurrences of the pattern are needed?

Solution

a.) m, since the first position is immediately a match.

b.) All positions match. There are $n-m+1$ positions, the pattern will be checked on all characters, so $m(n-m+1)$.

c.) m, since the first position is immediately a match.

d.) All positions match. There are $n-m+1$ positions, the pattern will be checked on all characters, so $m(n-m+1)$.

Note:

- In this case there is no difference between what Simple search and what Quick search does. Quick search is able to skip **some** definitely non-matching positions based on a heuristic. In this case, all positions match, there is nothing to skip.

1.10 Session 1, Exercise 10

Exercise

Give text S of length n such that for a pattern consisting of $m > 2$ 0's Simple search uses $O(n)$ comparisons on S, independently of m.

Solution

If the first character doesn't match every position will fail after 1 comparison. There will be $n-m+1$ positions, which is $O(n)$, because $n - m + 1 \leq n + 1 \leq n + n \leq 2n$, constants $c = 2$, $n_0 = 1$ work.

So let's make a text in which no character is a 0, e.g. all characters are 1's, so the first 0 of the pattern will never match.

1.11 Session 1, Exercise 11

Note: this is a hard exercise, using probability theory as well, it is not included in the exam!

Exercise

Prove that the expected running time of Simple search is $O(n)$, when both the text and pattern are random 0 - 1 sequences (the bits are independent of each other and probabilities of 0 and 1 are both $\frac{1}{2}$). What happens if only the pattern is random?

Solution

The pattern is denoted by M and its length is $m = |M|$, the text is denoted by S and its length is $n = |S|$.

Let's denote with the random variable t_i the number of comparisons made by the Simple search algorithm for a pattern position with a shift of i number of characters.

Then, in total the number of comparisons made is $\sum_{i=0}^{n-m} t_i$, so the expected number of comparisons is $E(\sum_{i=0}^{n-m} t_i)$.

$E(\sum_{i=0}^{n-m} t_i) = \sum_{i=0}^{n-m} E(t_i)$, due to the Linearity of Expectation. It is important to remember, that this holds true even when the variables are correlated, like in our case!

Now, the only thing left to find is $E(t_i)$.

For a given position k in the pattern, the probability of it matching the current position in the text, or $P(S[k + i] = M[k])$ is $\frac{1}{2}$, since both the pattern and the text is random, and they match for $S[k + i] = M[k] = 0$ and $S[k + i] = M[k] = 1$ while not match for $S[k + i] = 0, M[k] = 1$ and $S[k + i] = 1, M[k] = 0$, all four of these happen with $\frac{1}{4}$ probability, and two of these are the desired.

Now, the comparisons are made up until the point one of them fails, and we care about the number of them. This would be a geometric distribution, if the number of possible positions would be infinite. While this is not true, since the pattern and the text are both finite, since we only care about an upper bound, we can over-estimate the expectation value with the geometric distribution's expectation value.

$E(t_i) \leq \sum_{j=1}^{\infty} j2^{-j} = 2.$

Then we plug this back in:

$E(\sum_{i=0}^{n-m} t_i) = \sum_{i=0}^{n-m} E(t_i) \leq \sum_{i=0}^{n-m} 2 = 2(n - m + 1) \in O(n).$

When only the pattern is random, the only thing changing here is how we calculate $P(S[k + i] = M[k])$. If the text's character is a 0, or $S[k + i] = 0$ then the probability of a random $M[k]$ matching it is $\frac{1}{2}$, when $M[k] = 0$. Similarly, if the text's character is a 1, or $S[k + i] = 1$ then the probability of a random $M[k]$ matching it is $\frac{1}{2}$, when $M[k] = 1$. So the probability is the same and the same result holds here as well.

1.12 Session 1, Exercise 12

Note: this is a hard exercise, it is not included in the exam!

Exercise

Algorithm A solves the problem of pattern matching for 0 - 1 sequences, in case of pattern of m bits and text of n bits it uses $T(n, m)$ steps to give all occurrences of the pattern (in increasing order). How can this be used to find all occurrences of a length m pattern in a length n text over an arbitrary alphabet Σ using $O((n+T(n,m))log_2|\Sigma|)$ time?

Solution

Let's just say that $O((n+T(n,m))log_2|\Sigma|)$ is suspiciously specific. Especially the $log_2|\Sigma|$ part indicates that we should encode the alphabet in binary form, then a length of an original character in binary will be $log_2|\Sigma|$ in this new alphabet.

However, an issue with this approach will be, that only whole-character shifts should be allowed. We can not allow the algorithm to shift the pattern by half a binary-encoded character's length and find a match there. There is another issue, where this would also result in $T(log_2|\Sigma|n, log_2|\Sigma|m)$ runtime for A and we have no idea about the inner workings of T to somehow estimate this using $T(n, m)$.

Both of these issues will be solved, if we instead create $k = \lceil log_2|\Sigma| \rceil$ number of different pattern matching tasks, and the ith task will contain the original task's characters replaced by their ith bits.

Now if we let the algorithm find all the occurrences, if it finds let's say an occurrence with shift a in all of the k tasks, that means that all of the bits of the characters match, so the original pattern matches the original text with shift a as well!

To keep track of the results of the k tasks, we create an array Z of length n, initialized with 0's at the beginning. If the algorithm on the ith task finds an occurrence with shift a, it increments $Z[a]$ with 1. Then, at the end when all algorithms finished we read Z and if the value at position a is k that means that all of the k tasks found that as a match, so the original pattern matches with shift a as well.

Finally, the number of steps required to run A on k tasks with the same length as the original string but in binary is $kT(n, m)$, while initializing and incrementing the Z array (of size n) is at most nk, so in total we are at $O((n+T(n,m))k)$ steps, which is $O((n + T(n, m))log_2|\Sigma|)$.

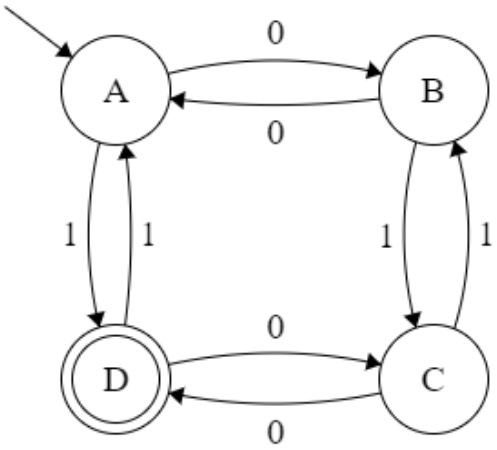
2 February 23rd (Session 2): Finite Automata

2.1 Session 2, Exercise 1

Exercise

Let $\Sigma = \{0, 1\}$. Give a deterministic finite automaton that accepts the words that contain an even number of zeros and an odd number of ones.

Solution



Proof:

Let's look at what the states mean:

- A: State A represents words that contain an even number of 0's and 1's.
- B: State B represents words that contain an odd number of 0's and an even number of 1's.
- C: State C represents words that contain an odd number of 0's and an odd number of 1's.
- D: State D represents words that contain an even number of 0's and an odd number of 1's.

Let's look at the starting state and the accepting and rejecting states:

- The starting state is the state that should represent the empty string. The empty string contains zero 0's and zero 1's, and zero is even, which is represented by A, so the starting state is A.
- The only accepting state is D, since we want to accept words that contain an even number of zeros and an odd number of ones, which are represented by D.
- A,B and C are rejecting states, since they represent words that are not in the desired language.

Let's look at the transitions:

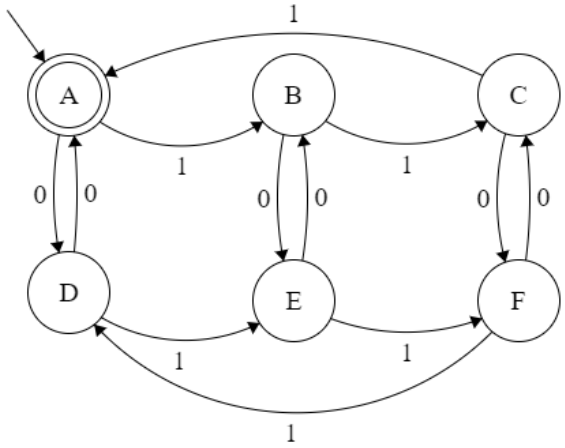
- Transitions triggered by a 0 input are $A \to B$, $B \to A$, $C \to D$, $D \to C$. In these cases the parity of the 1's doesn't change, while the parity of the 0's is inverted. If we look back on what the states represent we can verify in all 4 cases that this is the case.
- Transitions triggered by a 1 input are $A \to D$, $D \to A$, $B \to C$, $C \to D$. In these cases the parity of the 0's doesn't change, while the parity of the 1's is inverted. If we look back on what the states represent we can verify in all 4 cases that this is the case.

2.2 Session 2, Exercise 2

Exercise

Let $\Sigma = \{0, 1\}$. Give a deterministic finite automaton that accepts the words that contain an even number of zeroes, while the number of ones is divisible by three.

Solution



Proof:

Let's look at what the states mean:

- A: State A represents words that contain an even number of 0's and the number of 1's is in the form $3k$ (divisible by three).
- B: State B represents words that contain an even number of 0's and the number of 1's is in the form $3k + 1$.
- C: State C represents words that contain an even number of 0's and the number of 1's is in the form $3k + 2$.
- D: State D represents words that contain an odd number of 0's and the number of 1's is in the form $3k$ (divisible by three).
- E: State E represents words that contain an odd number of 0's and the number of 1's is in the form $3k + 1$.
- F: State F represents words that contain an odd number of 0's and the number of 1's is in the form $3k + 2$.

Let's look at the starting state and the accepting and rejecting states:

- The starting state is the state that should represent the empty string. The empty string contains zero 0's and zero 1's, and zero is even and also divisible by three, which is represented by A, so the starting state is A.
- The only accepting state is A, since we want to accept words that contain an even number of zeros and the number of ones is divisible by three in them, which are represented by A.
- The other states are rejecting states, since they represent words that are not in the desired language.

Let's look at the transitions:

- Transitions triggered by a 0 input are $A \to D$, $D \to A$, $B \to E$, $E \to B$, $C \to F$, $F \to C$. In these cases the parity of the 1's doesn't change, while the parity of the 0's is inverted. If we look back on what the states represent we can verify in all 6 cases that this is the case.
- Transitions triggered by a 1 input:
  - $A \to B$ and $D \to E$ move from states that have seen $3k$ 1's to states that have sen $3k+1$ 1's (the remainder goes from 0 to 1).

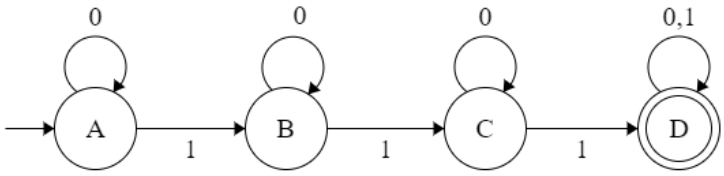
- $B \to C$ and $E \to F$ move from states that have seen $3k + 1$ 1's to states that have sen $3k + 2$ 1's (the remainder goes from 1 to 2).
- $C \to A$ and $F \to D$ move from states that have seen $3k+2$ 1's to states that have sen $3k$ 1's (the remainder goes from 2 to 0).

2.3 Session 2, Exercise 3

Exercise

Let $\Sigma = \{0, 1\}$. Give a deterministic finite automaton that accepts the words that contain at least three 1's.

Solution



Proof:
Let's look at what the states mean:

- A: State A represents words that contain no 1's.
- B: State B represents words that contain exactly one 1.
- C: State C represents words that contain exactly two 1's.
- D: State D represents words that contain at least three 1's.

Let's look at the starting state and the accepting and rejecting states:

- The starting state is the state that should represent the empty string. The empty string contains no 1's, which is represented by A, so the starting state is A.
- The only accepting state is D, since we want to accept words that contain at least three 1's, which is represented by state D.
- A,B and C are rejecting states, since they represent words that contain less than three 1's, which are not in the language.

Let's look at the transitions:

- Transitions $A \xrightarrow{1} B$, $B \xrightarrow{1} C$, $C \xrightarrow{1} D$: the amount of 1's in the word is incremented by one, which means we should move one step closer to state D.
- Transitions $A \xrightarrow{0} A$, $B \xrightarrow{0} B$, $C \xrightarrow{0} C$: Reading in a 0 does not change the current state. It does not move us closer to state D, however it does not ruin any progress already made, since the 1's don't have to be consecutive. So we just stay in the current state, discarding the incoming 0's.
- $D \xrightarrow{0,1} D$: when we have already seen at least three 1's, seeing characters (0's or 1's) will not give us additional benefits, we can already accept the word as is, so we just stay in state D until we reach the end of the input.

2.4 Session 2, Exercise 4

Exercise

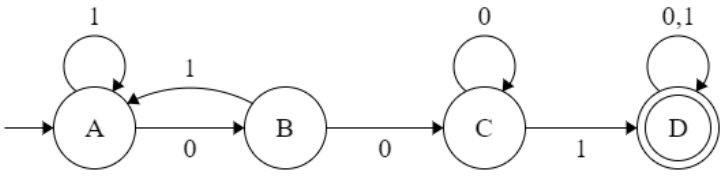
Let $\Sigma = \{0, 1\}$. Give a deterministic finite automaton that accepts the words that do not contain the subword 001.

Solution

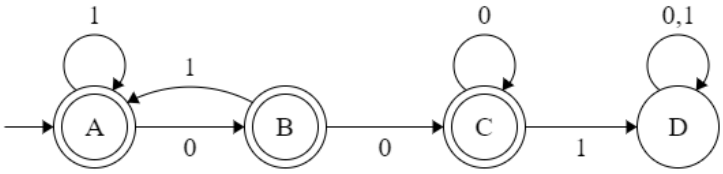
When dealing with **deterministic** finite automatons, a useful trick to keep in mind: sometimes it is easier to give a DFA for the complementer of the language, and then a DFA for the original language can be quickly created.

IMPORTANT: This trick only works for **deterministic finite automatons**!!!

Step 1: Create DFA for the complementer of the language:



Step 2: Invert the accept/reject status of the states:



Proof:

Let's look at what the states mean:

- A: State A represents words that end in something that is not a prefix of 001: they either are the empty string or they end in a 1; and do not contain 001 itself.
- B: State B represents words that end in 0.
- C: State C represents words that end in 00.
- D: State D represents words that contain 001.

Let's look at the starting state and the accepting and rejecting states:

- The starting state is the state that should represent the empty string. The empty string contains no prefix of 001, so it is represented by A.
- In the original DFA words that contain 001 were accepted, which is represented by state D, while states A,B and C were rejecting.
- To get a DFA for the complementer of the language, we simply need to accepts words that we have rejected before and reject words that we have accepted before. This is done by making states A,B and C accept, while making state D reject.

Let's look at the transitions:

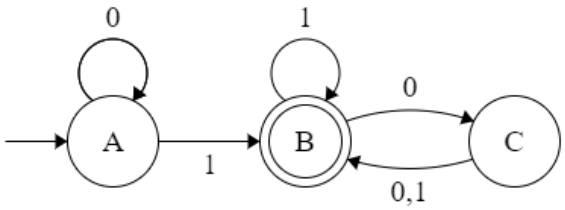
- In state A, when we read in a 0 we can move to state B, since now the current ending is 0. However when we read 1's, we are not getting closer to finding a 001 substring (we need the 0's first), so we just discard them.

- In state B, we have the ending 0. When we read another 0 in, this results in having the ending 00, so we can move forward to state C, one step closer to finding the substring 001. However if we read in a 1, this ruins our progress, since the current ending is now 01, but we needed a 00. We can't even stay in state B, since we would need a 0 ending for that, but the last character has been a 1. We have to move all the way back to state A.
- In state C the current ending is 00. If we read in a 1, that means we just found a 001 substring, we can move to state D! And if we read a 0 in, while we can't move to state D, we can stay in C, since the current ending is 000, or discarding the oldest 0: 00, which means we can stay in state C.
- In state D we have already found the substring, we just read in the remainder of the input, discard it and accept when done.

2.5 Session 2, Exercise 5

Exercise

Which words are accepted by this automaton? ($\Sigma = \{0, 1\}$).



Solution

I want to prove that the states mean the following in the given automaton:

- State A: words that don't contain a 1.
- State B: words that contain a 1 and end in an even number of 0's (including zero number of 0's).
- State C: words that contain a 1 and end in an odd number of 0's.

We can look at the defined transitions to prove that this is indeed correct:

- In state A as long as we only read 0's in, we stay in state A. If we read a single 1 in, we move away and we can never come back. So only words that contain no 1's can end up in state A.
- In state B and C the word already contains a 1, since we left A.
- If we read in a 1 in either B or C we always "reset" to B back: since when the input ends in a 1, that means that it ends in an even (zero) number of 0's. This is done by the transitions $B \xrightarrow{1} B$ and $C \xrightarrow{1} B$.
- If we read a 0 in state B the parity of the ending zeroes changes from even to odd, and vice versa for state C. These are done by the transitions $B \xrightarrow{0} C$ and $C \xrightarrow{0} B$.

Furthermore the starting state is correct:

- The starting state is state A, since the empty string contains no 1's, which is represented by state A.

Now let's look at the accepting/rejecting states:

- State B accepts, which means that the language of the automaton is **"words that contain a** 1 **and end in an even number of** 0**'s (including zero number of** 0**'s)"**.
- The other two states A and C reject (which means words that don't contain a 1 and words that contain a 1 but end in an odd number of 0's are rejected).

2.6 Session 2, Exercise 6

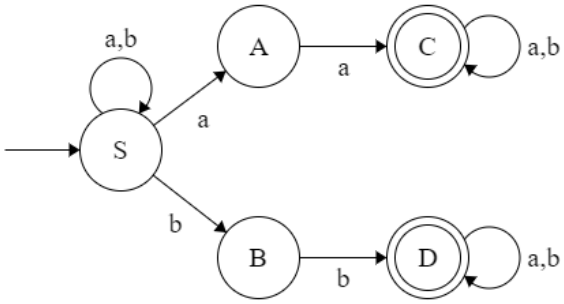
Exercise

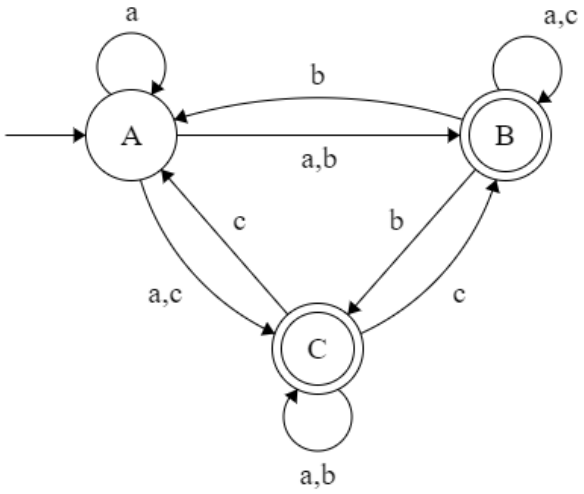
Perform the following for both nondeterministic finite automata.

a.) Give the computation tree of word *baabab*.

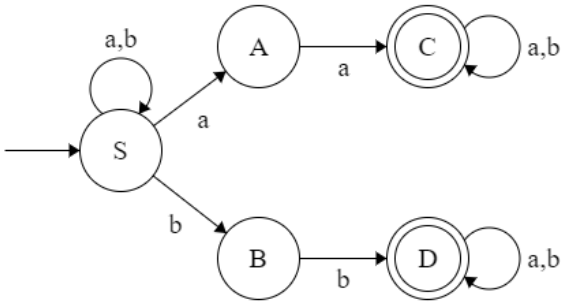
b.) Create an equivalent DFA by the procedure studied in class.

c.) Which languages are recognized by these automata?

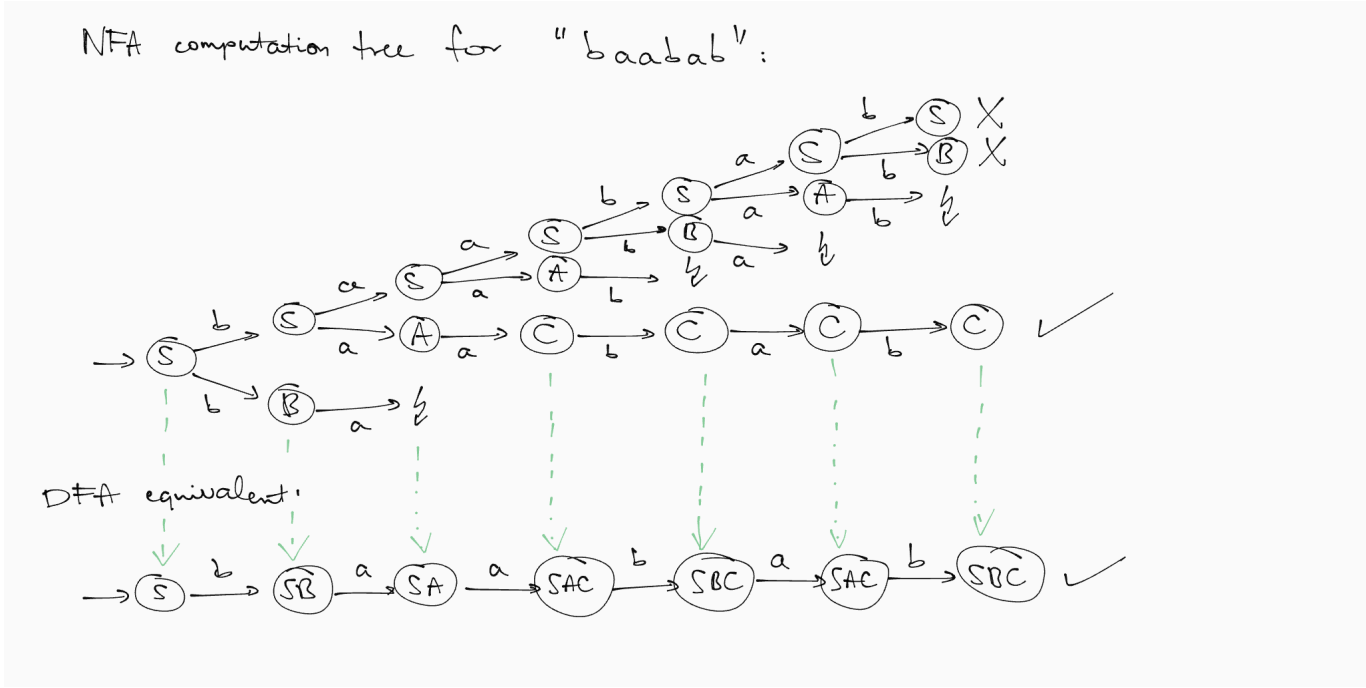




Solution



Give the computation tree of word *baabab*:

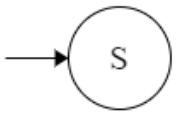


The word *baabab* is accepted, since there exists a branch that resulted in an accept state (C).

The DFA equivalent computation is also shown in the drawing. We basically follow all branches in "parallel", using meta states. We will construct the DFA, that will do a computation like this one:

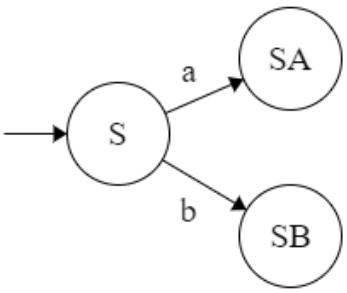
Step 1

We start with the same starting state as in the NFA: S.



Step 2

Then, we check where can we move for an a input from S in the NFA: $S \xrightarrow{a} \{S, A\}$, so we add the SA state for an a transition, and similarly for $S \xrightarrow{b} \{S, B\}$, so we add the SB state for a b transition.



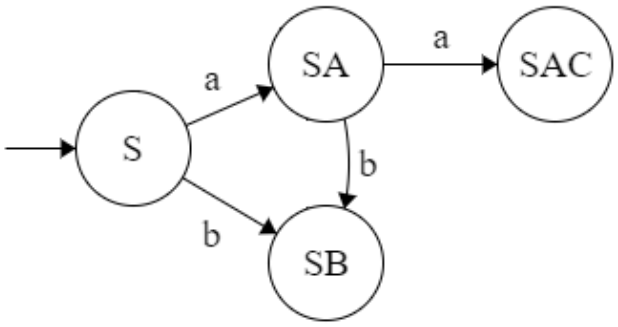
Check back on the DFA equivalent on the drawing above! You will notice the state SB as the second step.

Step 3

And we continue the same thing for the new states:

Where does SA move for an input character of a? The transitions from S and from A for input a are: $S \xrightarrow{a} \{S, A\}$ and $A \xrightarrow{a} \{C\}$, so together they move to state SAC.

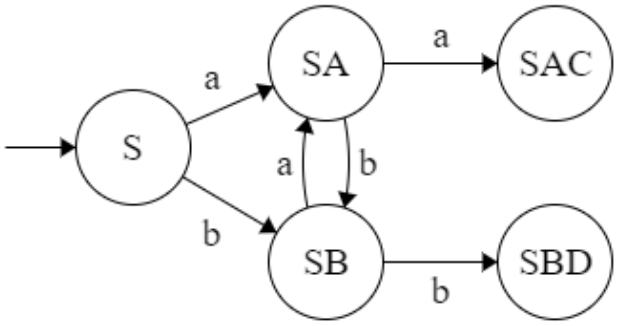
Where does SA move for an input character of b? The transitions from S and from A for input b are: $S \xrightarrow{b} \{S, B\}$ and $A \xrightarrow{b} \{\}$, so together they move to state SB.



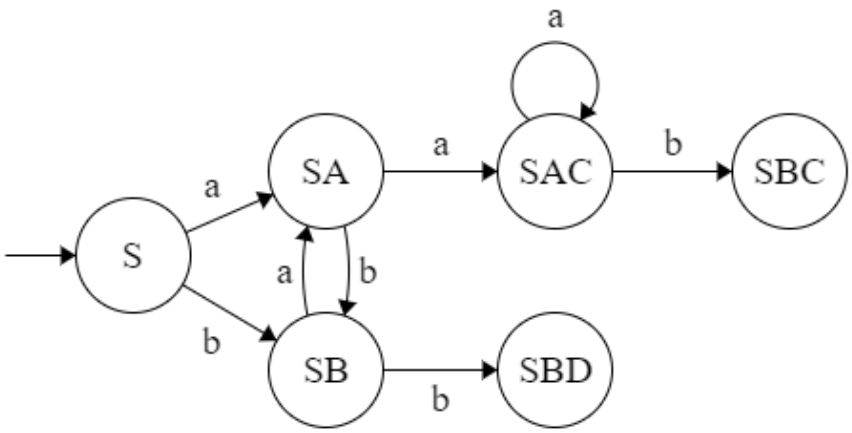
The process is the same: find a state that does not have all transitions defined (for a and for b input as well), check where their basis states transition in the NFA for the given input and add it as a transition. If the resulting state does not exist, add the state.

When no new states are added and all states transitions are fully defined the process is done.

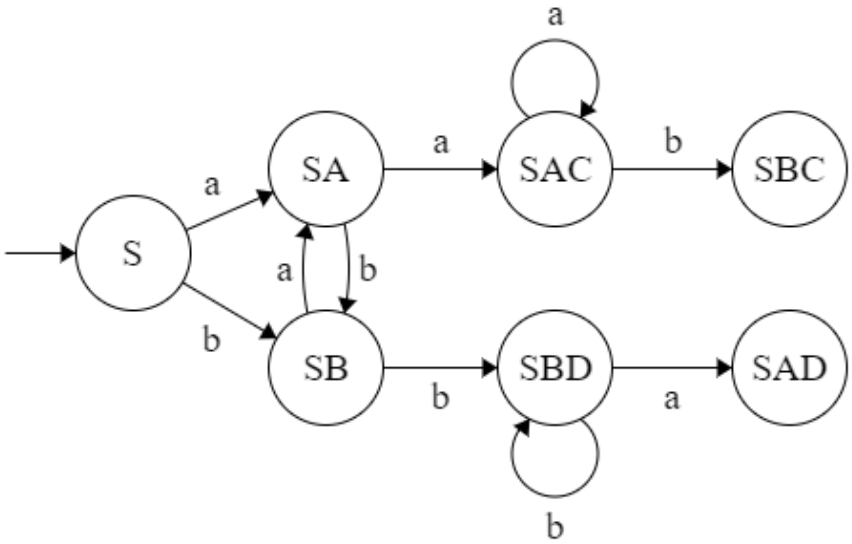
Step 4



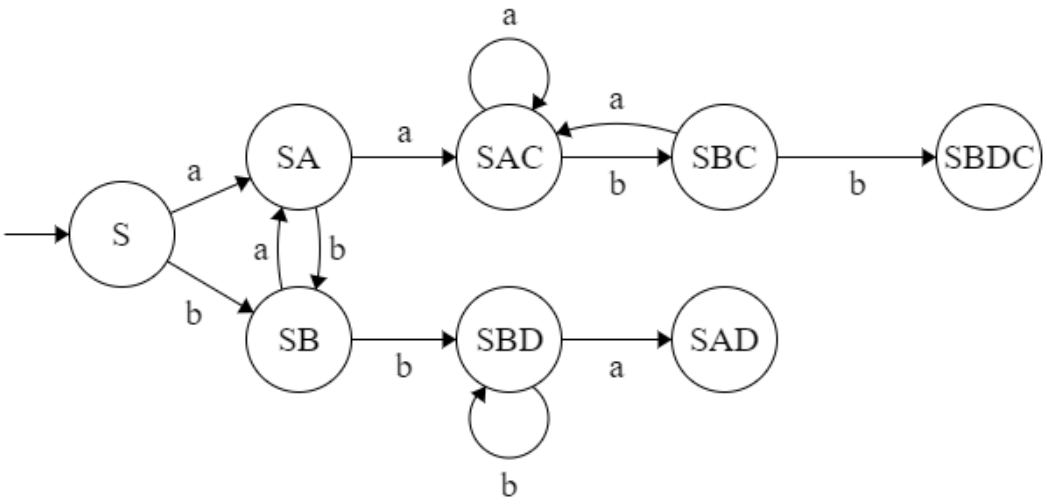
Step 5



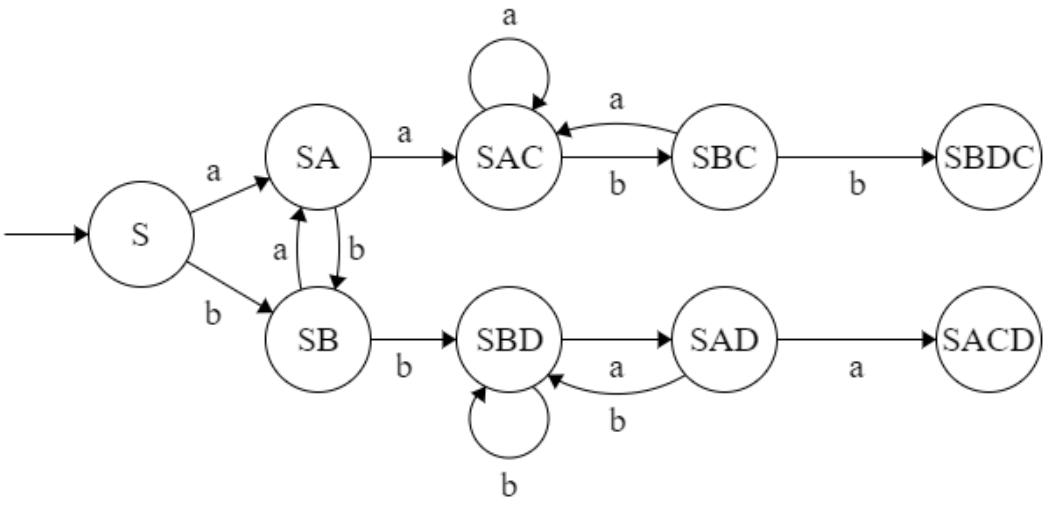
Step 6



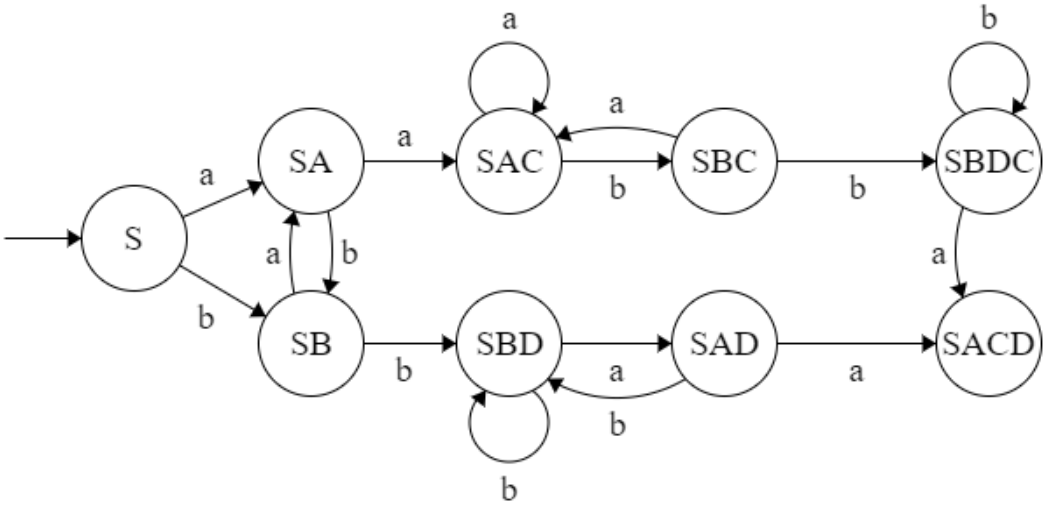
Step 7



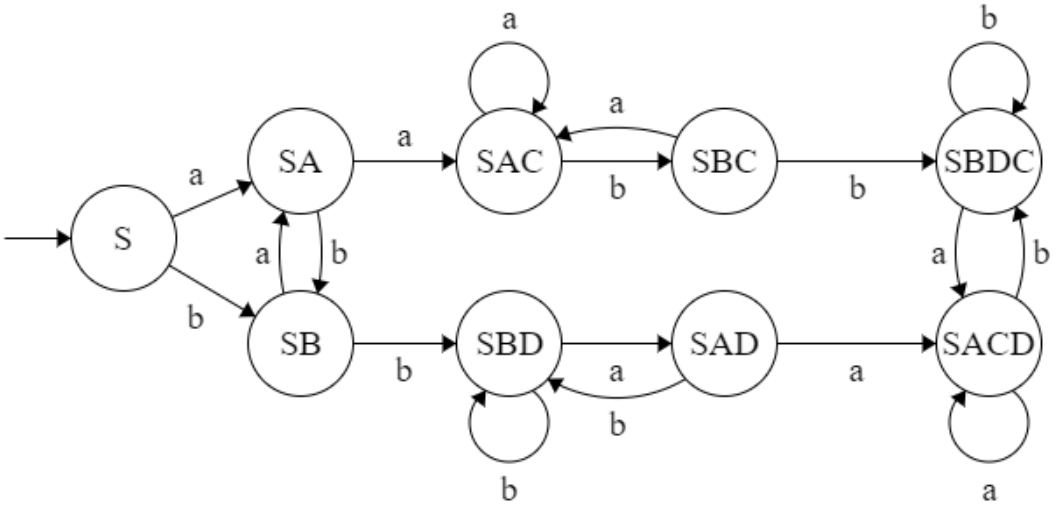
Step 8



Step 9

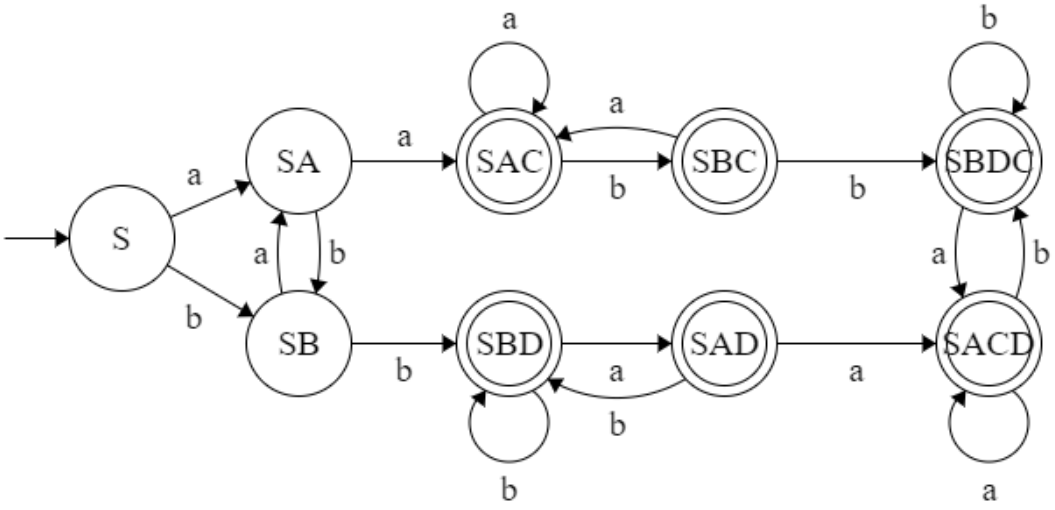


Step 10



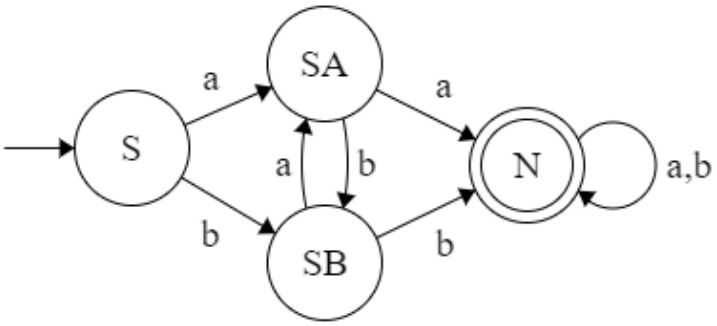
Step 11

Finally, define the accepts states: Any state that contains an original accept state (in this case C or D) will be an accepting state:

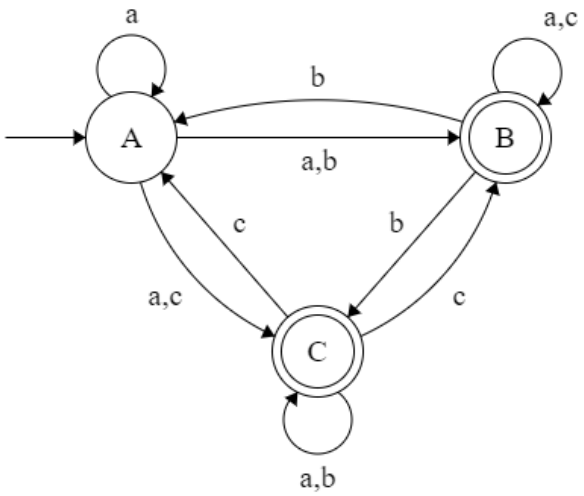


Note

By the way, the automaton can be simplified, like this:



Since the moment we reach any of the accepts states, we will never leave them.



TODO

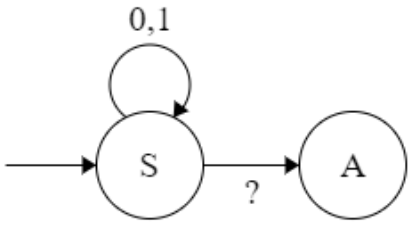
2.7 Session 2, Exercise 7

Exercise

Give a nondeterministic finite automaton that accepts those words that have 10100 as subword.

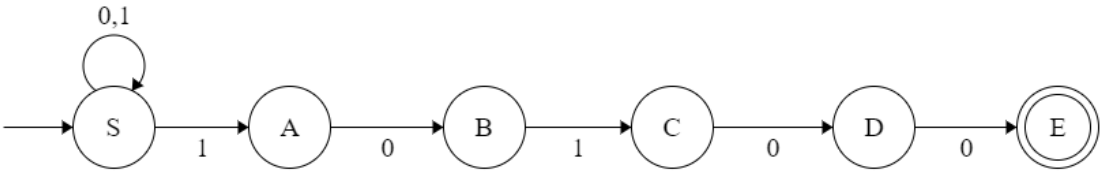
Solution

The key to solving this exercise using a nondetermnistic automaton is to create a delayed start on the starting state by introducing a 0,1 loop on it:

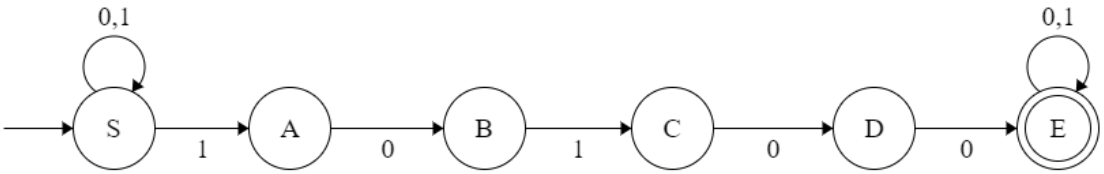


This will "eat up" some prefix of the input word before allowing the computation to proceed to state A. Whatever we put in place of the ? on the leaving transition will be a nondeterministic choice for the automaton.

The next step is to add the success path to the automaton which contains the string we want to have as a subword: 10100.



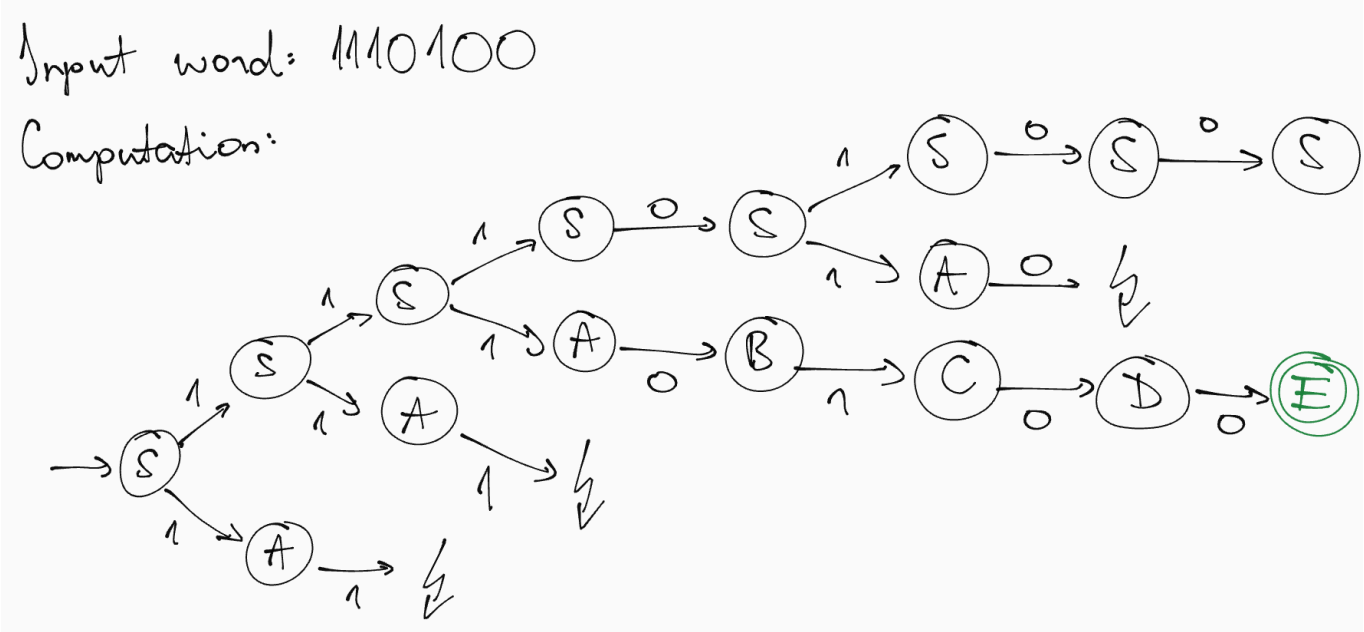
Then finally, let's allow "eating up" any remaining suffix of the word as well:



Some observations:

- Firstly, this is a nondeterministic finite automaton: Notice how in state S, for an input character of 1 we can either remain in S, or move to state A.
- There are also transitions missing: it is an incomplete finite automaton as well. For example, in state A, for an input character of 1, the machine halts (and halting due to a missing transition means rejection, **regardless of the current state's accept/reject status**.
- Notice how visually similar this automaton is to the regular expression for the same language: (0+1)*10100(0+1)*.

Let's look at an example computation on the word 1110100, which is in the language of the automaton.



- See, how it tried to leave state S and move to state A many times in different places of the input string? Many times it failed, when the timing was incorrect. Maybe it was too early, when the first two 1's, which are not part of the matching subword were given to it, or it was too late, when the pattern was already gone.
- There was already a lazy case, the top branch, which just remained in state S forever, which also could not succeed.
- However there only needs to be a single successful branch and it only needs to time its move to state A correctly once: on the third branch it successfully finished in state E, which means that the word is accepted, correctly!
- Notice how it is only possible to reach state E, when the input word contains 10100. We need a 1 to move from S to A, then we need a consecutive 0 to move from A to B, then we need a consecutive 1 to move from B to C, then we need a consecutive 0 to move from C to D, then we need another consecutive 0, to move from D to E.
- When the timing is just right and we catch the beginning of the pattern we can "sail smoothly" towards E.

When you are on the exam, you will need to give some sort of proof that the automaton you wrote up does what the exercise is asking you to do. For this exercise this is how a proof like this might look like:

Proof:

1. Let's look at the states of the automaton: What the different states mean and why their transitions are correct.

- State S is the starting state. Here, we non-deterministically wait to start our computation, using the 0,1 loop. Since the first character of the pattern we seek is a 1, on an input character of 1, the automaton can decide to move to state A.
- State A represents the information "already read the first character of the pattern". Here, we allow a transition to state B if the second character of the pattern comes, which is a 0. However, the transition for an input of 1 is missing, which means that the computation (on that branch) will halt. This is correct, because the pattern's characters must be consecutive.
- States B, C and D work similaly: they represent the next character recognized from the pattern we seek and we only allow a transition for the upcoming character from the pattern to move forward, towards E.
- Finally, state E represents "pattern found". This means that we can accept the word, furthermore, any further input characters are allowed, since the pattern can be anywhere in the word.

2. Let's look at the accepted and rejected words of the automaton:

Any word that contains the pattern 10100 will be accepted, because the automaton on the (single) accepting branch of the computation first non-deterministically reads the prefix of the word before the pattern, then transitions from S, to E using the consecutive characters of the pattern, then finally in state E further characters can be read (the remaining suffix of the word) and the word will be accepted.

Any word that does not contain the pattern 10100 can not reach state E, because each step towards E requires the next character of the pattern to be present consecutively in the word. There is correct timing to leave S, no computational branch can end up in E, so the word will be rejected.

(End of proof.)

(Due to time limits on the exam, it is okay to not use full sentences like I did above, abbreviate things, etc.)

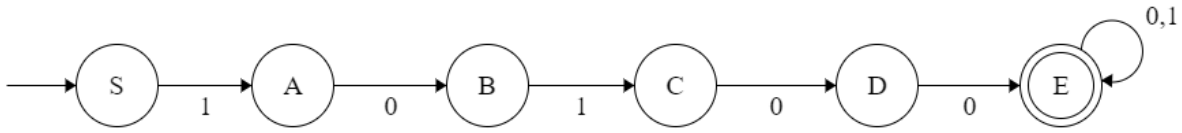
It is important when doing these proofs, that you do not use a specific input word as an example, but generalize to any possible words, like the proof above.

Deterministic solution

This exercise allows us to truly appreciate nondeterministic automata, since it made it really easy for us to come up with a design for a given subword.

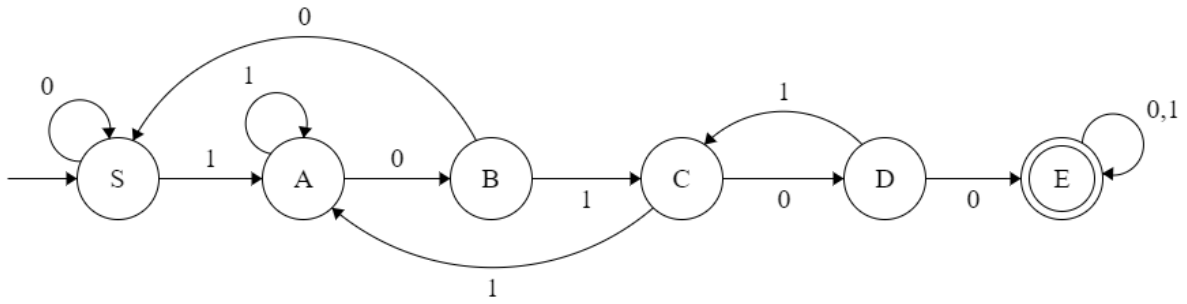
However, the task is also possible with a deterministic automaton (it must be, since we can convert any NFA into a DFA, but there is an even easier method to obtain a DFA, than the general conversion algorithm). This is not part of the task, but nice to see how it works as well.

Step 1: Get rid of loop on the starting state, we need to be deterministic now.



Step 2: Put in the missing transitions, states S-D all miss 1 of them.

The main idea here is that the missing transitions are failures in recognizing the next character of the pattern at the current position. How big of a failure it is depends on what the pattern we have found so far is and how much of it can be salvaged when we add the incorrect character at the end.



- When we are in state S, the pattern we have found so far is nothing. Until we read 0's we remain in state S, since the first character of the pattern is a 1.
- When we are in state A, the pattern we have found so far is "1". If we read another 1 now, we now have "11". The pattern is "10100", so that second 1 character can still turn out to be the beginning of the pattern, so we need to remember that we have a "1" and thus stay in state A.
- When we are in state B, the pattern we have found so far is "10", if we read a 0 in now, that makes it a "100". Unfortunately, there is no salvaging this: not "100", not "00" and not even a single "0" is useful for us, neither of them are the prefixes of the pattern "10100". We need to scratch everything and go back to state S.
- When we are in state C, the pattern we have found so far is "101". If we read a 1 now, that makes it "1011". The possible suffixes to remember here are "011", "11" and "1". In general we always need to keep the longest one that is still a prefix of the pattern: in this case, that is "1", which is represented by state A, so we move back there.
- When we are in state D, the pattern we have found so far is "1010". If we read a 1 now, that makes it "10101". This is great news, because we can actually just forget the first two characters and we can still keep the remaining "101", which is the first three characters of the pattern! Not much is lost, "101" is represented by state C, so we can move there.

2.8 Session 2, Exercise 8

Exercise

Prove that the language that consists o those words that have two 1's such that the number of 0's between them is divisible by four, is regular. (There could be several 1's between the two chosen 1's, besides the $4k$ 0's.)

Solution

2.9 Session 2, Exercise 9

Exercise

Design a finite automaton that accepts positive rational numbers written in decimal form. (Σ contains the decimal point and digits $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$.) The number to be accepted is either an integer without decimal point (e.g. 123), or it contains a decimal point. In this latter case numbers without integer part or fractional part must also be accepted, but at most one of these parts may be missing. (For example, 123.456, 123. and .456 are all accepted, but a single decimal point is not.) It is also requested that a number cannot begin with dummy 0's, however 0.456 is OK.)

Solution

2.10 Session 2, Exercise 10

Exercise

Let $\Sigma = \{0, 1\}$. The sequences are considered as binary numbers. Give a finite automaton that accepts exactly those words that represent numbers divisible by three in binary form. Take into consideration that a number does not begin with 0, except for number zero itself, and that the input number is read beginning with the most significant digit.

Solution

Any automaton that we design will work by reading the input binary string from left to right. We will want to keep track of the remainder of the current binary number after each 0 or 1 we read. The question is how do we update this remainder when the next input character comes? First, let's do a simpler task, just keeping track of the number itself and updating it as we go.

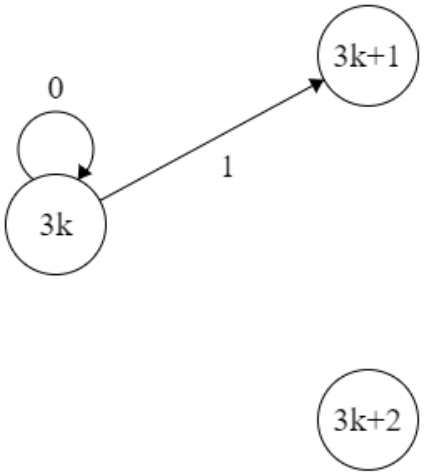
- For example, let's say that so far we have read the the "101" binary string on the input. That is a 5 in decimal form.
- Let's say the next character is also a 1, so now the current string is "1011", or in decimal form 11.
- This was achieved by shifting the string "101" to the left and adding a "1" as the least significant character.
- A left shift in binary corresponds to multiplication by 2 in decimal, and then if the next character is a 1, we just need to add 1 to the decimal value as well. So in our example, $5 * 2 + 1 = 11$.
- In general, if we read a binary number from left to right, to calculate its decimal value, we simply multiply the current decimal value by 2, and if the bit we read was a 1, we add a 1 and continue to the next bit.

If we don't care about the entire number, just its remainder when divided by 3, we can do the same calculation, but modulo 3.

If the current number is divisible by 3, or in the form $3k$, and the next binary character is a 0, then to update we do $3k * 2 + 0 = 6k$, which means that the updated number will still be divisible by 3.

If the next binary character is a 1, then to update we do $3k * 2 + 1 = 6k + 1 = 3 * (2k) + 1$, which means that the updated number has a remainder of 1 when divided by 3.

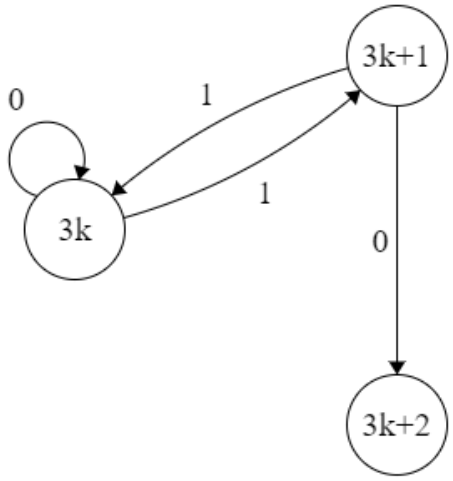
We can represent these statements by the following two transitions from state $3k$:



If the current number has a remainder of 1, when divided by 3, or in the form of $3k+1$, and the next binary character is a 0, then to update we do $(3k + 1) * 2 + 0 = 6k + 2 = 3 * (2k) + 2$, which means that the updated number has a remainder of 2, when divided by 3.

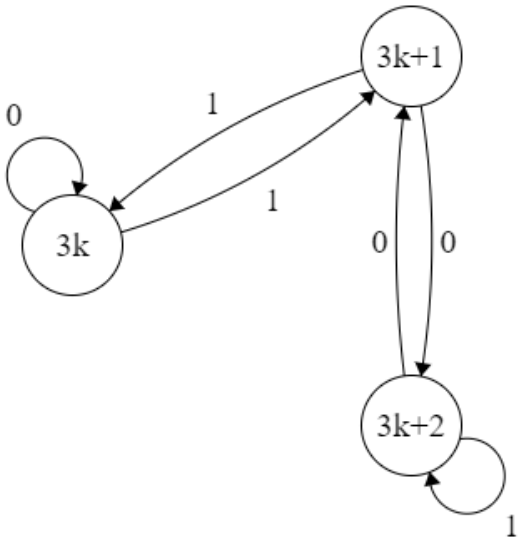
If the next binary character is a 1, then to update we do $(3k + 1) * 2 + 1 = 6k + 3 = 3 * (2k + 1)$, which means that the updated number is divisible by 3.

We can represent these statements by the additional two transitions from state $3k+1$:



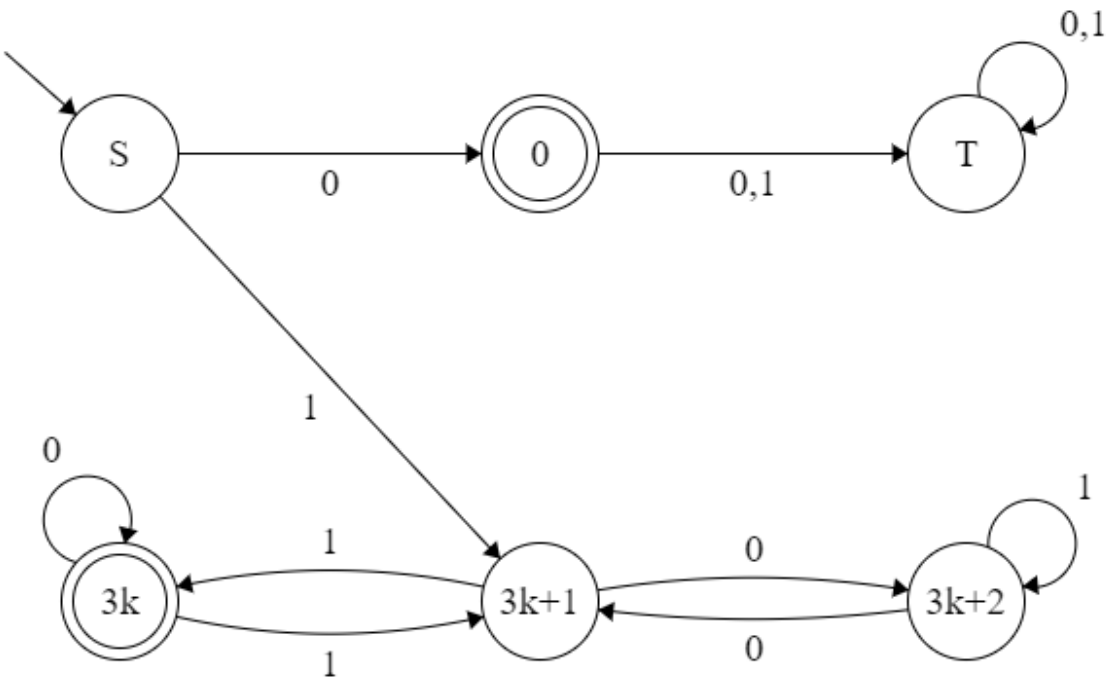
Finally, if the current number has a remainder of 2, when divided by 3, or in the for of $3k+2$, and the next binary character is a 0, then to update we do $(3k+2)*2+0 = 6k+4 = 3*(2k+1)+1$, which means that the updated number has a remainder of 1, when divided by 3.

If the next binary character is a 1, then to update we do $(3k+2)*2+1 = 6k+5 = 3*(2k+1)+2$, which means that the updated number has a remainder of 2, when divided by 3.



If you want to try this automaton, come up with any decimal number, turn it into binary form and calculate its remainder by 3, by applying the transitions above using its binary form as input. The starting state should be $3k$, since when we have not read anything in yet, so we want to start from a state that is equivalent to the number 0, which is divisible by 3.

I however, did not mark a starting state yet on this image, because there is one additional statement to keep in mind: "Take into consideration that a number does not begin with 0, except for number zero itself.". This means that we do not consider any string longer than 1 characters starting with a 0 a number, much less a number that is divisible by 3, so we need to reject these words.



Now, if this was an exam, here is how to prove that this accepts the language: (Basically everything I just said, but in a shortened form.)

Proof:

1. Let's explain what the different states mean and that their transitions are correct:

- S is the starting state. If we read a 0 in, we move to a state dedicated to the number 0. If we read a 1 in, we move to the state that represents binary numbers, that give a remainder of 1 when divided by 3, which is true for the (binary) number 1.
- The only word for state 0 is 0 itself, this is divisible by 3 and is accepted. If we read anything else after a 0 that is an incorrectly formatted number and moves to a trap state T which rejects it.
- The states $3k$, $3k+1$ and $3k+2$ represent the 3 remainder classes of division by 3. When a new character is read on the input, we update the remainder class with multiplying by 2 (binary shift) and adding the 0 or 1 bit we just read. We can see that the transitions are correct, since:
  - $3k * 2 + 0 = 3 * (2k)$
  - $3k * 2 + 1 = 3 * (2k) + 1$
  - $(3k+1) * 2 + 0 = 3 * (2k) + 2$
  - $(3k+1) * 2 + 1 = 3 * (2k+1)$
  - $(3k+2) * 2 + 0 = 3 * (2k+1) + 1$
  - $(3k+2) * 2 + 1 = 3 * (2k+1) + 2$

2. Let's explain that the correct words are accepted and the words not in the language are rejected:

The words in the language are any numbers that are divisible by 3, which is either the number 0 or anything that is in the form $3k$, these states are accepting.

A word could be outside of the language due to being malformed (starting with 0, but not being 0 itself), which will be redirected to a trap T; or due to not being divisible by 3, in which case it will land in either $3k+1$ or $3k+2$ and will be rejected there. Finally, the empty string is rejected because it's not a number, which is the only word that will end up in state S.

(End of proof.)

Notes:

- I ended up making a deterministic automaton here, however the task would allow a non-deterministic one as well. We could get rid of the enitre T state and define no transitions outwards from state 0. If there is input left to be read in state 0 the automaton halts due to a missing transition, which rejects the word regardless of the accept/reject status of the current state!

2.11 Session 2, Exercise 11

Exercise

Let language L_k consist of those word over alphabet $\Sigma = \{a, b\}$ that have character b on the k^{th} position counting rom backwards. (For example $bbaa \in L_3 \cap L_4$.)

a.) Prove that there exists a nondeterministic automaton of $k + 1$ states recognizing language L_k for all $k \geq 1$.

b.) Prove that every deterministic automaton recognizing L_k has at least 2^k states.

Solution

2.12 Session 2, Exercise 12

Exercise

Prove that every NFA can be transformed so that it recognizes the same language, however it has a unique accept state.

Solution

2.13 Session 2, Exercise 13

Exercise

The language L^R is obtained from language L so that every word in L is reversed, that is the characters of the word are written in reverse order. Prove that L is regular $\Leftrightarrow$ L^R is regular.

Solution

3 March 2nd (Session 3): Regular expressions, Context-free languages

3.1 Session 3, Exercise 1

Exercise

Let $\Sigma = \{a, b\}$ and let language L consist of words that contain the same number of a's and b's. Is L regular?

Solution

Gut feeling (This is not yet a proof!)

Not regular.

This language is similar to $a^n b^n$ (studied in the lecture). The main issue with it will be similar: we would need to remember the difference of the number of a's and b's we have read in so far and only accept the word if the difference is 0 after reading in the entire word.

For every possible difference, we will need a separate state, however the difference can be arbitrarily large, while we can only have a finite number of states using Finite Automata, so it won't be possible to construct such a machine.

Proof

We will do proof by contradiction:

- Let's assume that L is regular.
- Then, that means that there exists a Deterministic Finite Automata, that accepts the language L.
- Let's take one such automata, and name it M.
- Let's count the number of states in M and name this number n.
- Now let's list exactly $n + 1$ specifically chosen words from the L language: ab, $aabb$, a^3b^3, $\ldots$, $a^{n+1}b^{n+1}$.
- Then imagine feeding these $n + 1$ words into M. For all of them, let M read in the a letters and then stop and take note of which state the word is at the moment, halfway-through the operation.
- After reading in a, aa, a^3, $\ldots$, a^{n+1}, since these are $n + 1$ cases, while M only has n states, we can use the Pigeonhole Principle and say, that there exists at least two different strings a^i and a^j $(i \neq j)$, for which M arrived at the same state after feeding it these inputs. Let's name this state S.
- Since $a^i b^i$ is in language L, it must be accepted by M. This means that when we continue from state S and feed in the b's of the word, the machine must arrive in an accepting state. So there exists a path from state S to an accepting state that is traversed by the input b^i.
- However, M also arrives in state S when it reads a^j. We just noted, that if from S it reads b^i it will arrive in an accept state. If we put these two together, it means that M accepts the word $a^i b^j$, where $i \neq j$, which is **not** in L, since it doesn't have the same number of a's and b's.
- We stated in the beginning that M is a machine whose language is L, however we just found a word that is not in L, but accepted by M, so this is a contradiction.

Notes:

- This is symmetric, we could also prove that M accepts $a^i b^j$, for $i \neq j$ which is also a contradiction, since that word is also not in L.
- To put it shortly: the machine cannot distinguish a^i and a^j $(i \neq j)$ and since $a^i b^i$ and $a^j b^j$ are accepted, so are $a^i b^j$ and $a^j b^i$, which are not in L, which is a contradiction.
- Note, that this proof is exactly the same as the proof for language $a^n b^n$ studied in the lecture. This is due to the fact, that these languages are similar, for both of them the issue is keeping track of the number of a's to (eventually or simultaneously) compare them to the number of b's.
- It is **not true**, that "the proof works because $a^n b^n$ is a subset of L". For example, $a^n b^n$ is also a subset of Σ^*, which is regular!

3.2 Session 3, Exercise 2

Exercise

Let $\Sigma = \{(,)\}$. Prove that the language of properly matched parentheses sequences is not regular.

Solution

Quite similar to 3.1. The $n + 1$ words from L, the language of properly matched parentheses to be used are (), (()), $(^3)^3$, ..., $(^{(n+1)})^{(n+1)}$, so just substitute $a = ($ and $b =)$.

3.3 Session 3, Exercise 3

Exercise

Is the language regular, that consists of sequences of 0's of a length that is...

a.) an even number?

b.) an odd number?

c.) a perfect square?

d.) a power of 2?

Solution

3.3.1 Even number of 0's

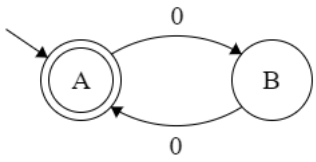
Regular.

Proof 1: The regular expression $(00)^*$ matches them.

Proof that this regular expression matches the language:

- The * operator allows any number of repeats, even 0.
- Inside the * operator we have two 0's, which can be repeated any number of times to match any even number of 0's.
- The empty string, also known as zero number of 0's contains an even number of 0's, so it is part of the language. $(00)^*$ matches the empty string, which is correct.

Proof 2: The following DFA accepts the language:



Proof that this automaton accepts the language:

- Words that end up in state A are the words that contain an even number of 0's, while words that end up in state B contain an odd number of 0's.
- The empty string is correctly accepted.
- From state A, reading another 0 moves to state B, so after reading an even number of 0's, if we read one more, now we have an odd number of 0's.
- And similarly for state B.

3.3.2 Odd number of 0's

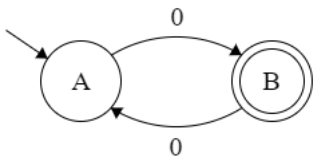
Regular.

Proof 1: The regular expression $0(00)^*$ matches them.

Proof that this regular expression matches the language:

- We have just seen that $(00)^*$ matches an even number of 0's.
- Adding the 0 at the front will then match an odd number of 0's.

Proof 2: The following DFA accepts the language:



Proof that this automaton accepts the language:

- This is the same automaton as in the previous exercise, but now the accept state is B, to accept an odd number of 0's.

3.3.3 A perfect square number of 0's

Gut feeling (This is not yet proof!)

Not regular.

The issue here is going to be, that the length of the accepted words get further and further away from each other as the length of the words increases. We always need to keep track of how far away we are from the next accepted word, how many more 0's we need to finally accept. We would need separate states for each of these "x number of 0's before we can accept", however x could be arbitrarily large and we only have a finite number of states.

Proof

Quite similar to 3.1, the $n+1$ words to use here are of the length of the first $n+1$ square numbers: $0 = 0^{1^2}$, $0000 = 0^{2^2}$, $0^9 = 0^{3^2}$, $0^{16} = 0^{4^2}$, 0^{5^2}, 0^{6^2}, $\ldots$, $0^{(n+1)^2}$.

Then, the finishing of 3.1 is a bit different:

When we find that for an $i \neq j$, both 0^{i^2} and 0^{j^2} end up in the same state S, the reasoning is a bit different. Without loss of generality, we can assume that $i < j$. If we were to continue 0^{i^2} from state S with $2i+1$ more 0's, then the whole input would be $0^{i^2+2i+1} = 0^{(i+1)^2}$, which means that we should accept this word, so from state S, for $2i+1$ number of 0's we must reach an accept state.

However, this also means that when we continue 0^{j^2} (which remember, also arrives in S) with $2i+1$ 0's, so the word is 0^{j^2+2i+1}, we will also arrive at the same accept state.

However j^2+2i+1 is not a square number. It is between two consecutive square numbers j^2 and $(j+1)^2 = j^2+2j+1$, but not equal to either of them:

- $j^2 < j^2 + 2i + 1$, since $0 < i$.
- $j^2 + 2i + 1 < j^2 + 2j + 1$, since $i < j$.

Thus, we found a word accepted by M, however not in L, which is a contradiction.

3.3.4 A power of 2

Gut feeling (This is not yet proof!)

Not regular, the situation is even worse than for square numbers, since powers of 2 are even further spaced apart as the exponent continues to grow.

Proof

The $n+1$ words to be used are 0^{2^1}, 0^{2^2}, 0^{2^3}, $\ldots$, $0^{2^{n+1}}$.

Then similarly to the previous proof, when for $i \neq j$, 0^{2^i} and 0^{2^j} end up in the same state S, if we continue by 2^i more 0's, we should accept, since $0^{2 \cdot 2^i}$ is also a power of two number of 0's, however this means $0^{(2^i+2^i)}$ will also be accepted, but $2^i + 2^j$ is not a power of 2, since $i \neq j$.

3.4 Session 3, Exercise 4

Exercise

Let $\Sigma = \{0, 1\}$. Determine the languages of the following regular expressions.

a.) $(0 + 1)^*011(0 + 1)^*$

b.) $1(0 + 1)^*0$

c.) $((0 + 1)(0 + 1))^*$

Solution

$(0+1)^*$ is a regular expression that accepts any string from Σ^*, since $0+1$ accepts either a 0 or a 1, and the * operator allows any number of repeats, including zero.

Thus, $(0 + 1)^*011(0 + 1)^*$ is a regular expression that accepts strings that can begin anyhow (including the empty string), then contain the word 011, then end anyhow, including the empty string. Or, simply put, it accepts all words that contain the string 011.

For $1(0 + 1)^*0$, the string must begin with a 1 and must end with a 0, while anything, including the empty string can be in between. So this regular expression accepts words that begin with a 1 and end with a 0.

For $((0 + 1)(0 + 1))^*$, the inner regular expression $(0 + 1)(0 + 1)$ accepts any strings with a length of two. Using the * operator on this allows this to repeat any number of times, allowing for any even length to be accepted, including the length of 0, which is the empty string.

3.5 Session 3, Exercise 5

Exercise

Give regular expressions for the languages over alphabet $\{0, 1\}$ that consist of the following words.

(a) Words of odd lengths.

(b) Words of even length that start and end with 1.

(c) Words containing at least three 0's.

(d) Words containing an even number of 0's.

(e) Words of odd lengths starting with 0 and words of even length starting with 1.

(f) Words of odd length containing subword 00.

Solution

3.5.1 Words of odd lengths

From the previous exercise we know, that $((0 + 1)(0 + 1))^*$ accepts the words of even lengths. If we add a $(0 + 1)$ at the beginning it will add one more character to the lengths, making them odd: $(0 + 1)((0 + 1)(0 + 1))^*$.

3.5.2 Words of even length that start and end with 1

To start and end with 1's, the regular expression will be 1<something here>1. For <something here>, we need to add a regular expression, that together with the two other 1's will allow for an even number of characters. So without the two 1's, we need an even number of characters, for which we know the regular expression: $((0 + 1)(0 + 1))^*$. Putting these together we arrive at $1((0 + 1)(0 + 1))^*1$. Since $((0 + 1)(0 + 1))^*$ accepts the empty string, the final regular expression will also accept 11, which is the shortest possible string in the language.

3.5.3 Words of odd length that start and end with 1

This was not in the exercise, however I would like to illustrate a point here.

Let's follow the same pattern of thought, as in the previous exercise:

- To start and end with 1's, the regular expression will be 1<something here>1.
- For <something here>, we need to add a regular expression, that together with the two other 1's will allow for an odd number of characters.
- So without the two 1's, we need an odd number of characters, for which we know the regular expression: $(0 + 1)((0 + 1)(0 + 1))^*$.
- Putting these together we arrive at $1(0 + 1)((0 + 1)(0 + 1))^*1$.
- We have made a mistake...

What is the shortest word in this language? It's 1, which is not accepted by the regular expression above. The issue is that we did not think about the fact, that both starting and ending in a 1 can also literally mean the same 1 character, nothing more.

In general, it is always important when building regular expressions from multiple parts, to check for short strings in the language, to see if our expression works for the simplest cases as well.

In this example, to fix the regular expression above, we simply append the missing case at the end: $1(0+1)((0+1)(0+1))^*1 + 1$. This accepts words of length 1, 3, and so on, which is what we've wanted.

3.5.4 Words containing at least three 0's

Anything can go between the 0's, so simply $(0 + 1)^*0(0 + 1)^*0(0 + 1)^*0(0 + 1)^*$. The three 0's between the $(0 + 1)^*$'s enforce that the string must contain at least three 0's.

3.5.5 Words containing an even number of 0's

Let's start by words containing exactly two 0's: $1^*01^*01^*$. Then, to allow an even number of 0's, we can use the * operator on this: $(1^*01^*01^*)^*$. However, this regular expression has one problem: if the number of 0's is zero, we are unable to match anything other than the empty string. However, we would like to match 1^*, so we can fix this, by either simply appending it at the end: $(1^*01^*01^*)^* + 1^*$, or to make it shorter, simply moving the first one outside of the outer * operator: $1^*(01^*01^*)^*$.

3.5.6 Words of odd lengths starting with 0 and words of even length starting with 1

Let's make two regular expressions and combine them with a + at the end.

Words of odd lengths starting with 0: $0((0+1)(0+1))^*$. 0 matching literal 0 at the beginning, then $((0+1)(0+1))^*$, matching the remaining even number of any characters, which in total result in an odd number of characters.

Words of even length starting with 1: $1(0+1)((0+1)(0+1))^*$. Since the word must start with a 1, this no longer can be an empty string. The shortest possible words are 10 and 11, which are correctly matched, and can be followed by an even number of characters.

Finally, combinging the two: $0((0+1)(0+1))^*+1(0+1)((0+1)(0+1))^*$, or to make it shorter: $(0+1(0+1))((0+1)(0+1))^*$.

3.5.7 Words of odd length containing subword 00

00 is even length, so it either starts with an odd length string, then 00, then an even length string or vice versa: $((0+1)(0+1))^*(0+1)00((0+1)(0+1))^* + ((0+1)(0+1))^*00(0+1)((0+1)(0+1))^*$ and we can shorten this a little like this: $((0+1)(0+1))^*((0+1)00+00(0+1))((0+1)(0+1))^*$. Again, paying attention to the shortest words in the language, which are of length three, this one can match those as well.

3.6 Session 3, Exercise 6

Exercise

Give regular expressions that are shorter than the following ones but give the same languages.

a.) $(0 + \varepsilon)^*$

b.) $((0 + \varepsilon)(0 + \varepsilon))^*$

c.) $(0 + 1)^*01(0 + 1)^* + 1^*0^*$

Solutions

$(0 + \varepsilon)^*$ can match at every recurrence either a 0 or an empty string, repeated any number of times. We can either not match any 0's, then this results in the empty string, or we can match any number of 0's as well (with any number of empty strings in-between them, which do not add anything to the result), so this is the same as 0^*.

For $((0 + \varepsilon)(0 + \varepsilon))^*$, the inner expression can match the empty string, 0, or 00. Repeated any number of times allows for matching any number of 0's, including the empty string as well. So this is also the same as 0^*.

$(0 + 1)^*01(0 + 1)^* + 1^*0^*$ is an interesting one: The word either contains a 0, followed by a 1 (in the first part of the sum), or... it does not: 1^*0^* matches words where the 1's can only be before the 0's, not after them, while $(0 + 1)^*01(0 + 1)^* + 1^*0^*$ matches words where there is a 0 that is followed by a 1. All possible 0/1 strings fit into either of these categories, so these two together match anything from $(0 + 1)^*$.

3.7 Session 3, Exercise 7

Exercise

Give a regular expression whose language consists of all words over $\{0, 1\}$ without the subword 110.

Solution

Unfortunately there is no method to take the complementer of a language when dealing with regular expressions (as opposed to DFA's).

The main thinking behind this is "where can we stop before we end up with a 110"?

- If we start with any number of 0's, we are in the clear: 0^*
- If we add a 1, we are getting closer to the fire, however we can quickly correct it by adding at least one 0 right after: $0^* + 0^*(10^*0)$.
- We can repeat this step as many times as we want to: $0^* + 0^*(10^*0)^*$, since that mandatory 0 between the 1's will make sure no two 1's follow each other.
- However, if we add two 1's, right after each other, we can not follow by a 0, or that would result in a 110. We can either end the string there, or we can still follow with more 1's, however no more 0's are allowed at that point. So we can either end the string here with as many 0's or 1's as we want: $0^* + 0^*1(0^*01)^*(0^* + 1^*)$

There is a more elegant form available, equivalent to the previous form:

$(0 + 10)^*1^*$.

The main idea behind this, is that we can repeat 0's, or 1's followed immediately by a 0 as many times and in whichever order we like. Then, we can one time decide to put down two consecutive 1's, at which point we can no longer put down 0's.

3.8 Session 3, Exercise 8

Exercise

What language is generated by the following grammar?

$$S \to A|B$$
$$A \to 0A1|01$$
$$B \to 1B0|10$$

Solution

We make a decision at S, whether to continue with variable A, or B.

A generates with the following producion rule: $A \to 0A1|01$, which will put the same number of 0's at the beginning as the number of 1's at the end, using the "one to the left and one to the right" method: every single activation of the $A \to 0A1$ puts one 0 to the left and one 1 to the right, making sure their numbers remain equal as we generate the string.

The empty string is not generated, because there is no production rule that would allow A to terminate in a ε. This is the same as 0^n1^n, where $n > 0$: $L_A = \{0^n1^n | n > 0\}$.

With similar logic, $L_B = \{1^n0^n | n > 0\}$.

And finally, then $L = L_A \cup L_B$.

3.9 Session 3, Exercise 9

Exercise

Give CF-grammars for the following regular languages from Exercise 4:

Let $\Sigma = \{0, 1\}$.

a.) Contains 011 as a substring: $(0 + 1)^*011(0 + 1)^*$

b.) Starts with 1, ends with 0: $1(0 + 1)^*0$

c.) Is of even length: $((0 + 1)(0 + 1))^*$

Solutions

3.9.1 Contains 011 as a substring

Let's make a variable for producting $(0 + 1)^*$:

$$A \to 0A|1A|\varepsilon$$

A generates any string from left-to right, using the first rule if the next character is a 0 and the second rule if the next character is a 1. When we reach the end of the string, with no more characters left, the third rule is applied and the production is completed.

Using A, we can now do

$$S \to A011A$$
$$A \to 0A|1A|\varepsilon$$

where S creates the required 011 substring and puts an A at the beginning and at the end A to generate any optional characters.

3.9.2 Starts with 1, ends with 0

Using the same A, and the same logic as before:

$$S \to 1A0$$
$$A \to 0A|1A|\varepsilon$$

3.9.3 Is of even length

Change up the wording a little: We generate a string that consists of two parts, that are of equal length.

Any time we need to generate something of equal length to something (or any numerical relationship between their lengths) the only way it will work, is if we generate them by the "one (some) to the left and one (some) to the right" method.

Right now, any characters are allowed, the only thing that matters is that they are of the same length, so we can do:

$$S \to 0S0|0S1|1S0|1S1|\varepsilon$$

Or some might prefer this form, may be a bit more cleaner:

$$S \to TST|\varepsilon$$
$$T \to 0|1$$

3.10 Session 3, Exercise 10

Exercise

Give CF-grammar for the language of properly matched parentheses sequences.

Solution

Again, apply the "one to the left, one to the right" method, and generate the matching pairs of parentheses at the same time!

Starting out with something like this, which is not yet correct:

$$S \to (S)|\varepsilon$$

However, this only generates $((((\ldots))))$, while we can also add properly matched parentheses next to each other:

$$S \to SS|(S)|\varepsilon$$

Proof that this is correct:

To generate a string of properly matched parentheses, we start by counting the number of the outermost parentheses pairs, and use the first rule to generate an equal number of S variables. Then, we use the second rule once on all of them, to put down the outer parentheses. Then, we repeat for the inner strings, which must be also properly matched parentheses. When there is no more inner string, we use the third rule to terminate the generation.

No improper strings can be generated, since the second rule guarantees that only matched pairs are generated in that step, while the inner string will also be a properly matched parentheses sequence, since it is generated from S as well. The first rule is correct also, since properly matched parentheses can be concatenated to arrive at another properly matched parentheses string.

3.11 Session 3, Exercise 11

Exercise

Determine the languages generated by the following grammars.

a.)

$$T \to TT|aTb|bTa|a|\varepsilon$$

b.)

$$R \to TaT$$
$$T \to TT|aTb|bTa|a|\varepsilon$$

Solution

a.)

$$T \to TT|aTb|bTa|a|\varepsilon$$

We can quickly see, that for any word generated by this grammar, the number of a's can not be less than the number of b's in it.

Intuitively we have a gut feeling, that due to the first rule, the order in which these characters appear might be completely arbitrary and actually all words like this can be generated.

We will show that this is true:

Proof

Statement: Any word for which the number of a's is not less than the number of b's can be generated using the production rules above.

We will be using mathematical induction for the length of the generated strings.

For the base case, of length either 1 or 0, the word can either be ε (the empty string) or a, both of which can be generated (in one step, using either the fourth or the fifth rule).

Inductive step: We will prove that if the statement is true for any word of length less than n, then it is also true for any word of lengfth n.

Let's take a word of length n: $w = x_1x_2, \ldots, x_n$.

Let's take the smallest i, for which in the word $w_{1..i}$ there is exactly as many a's as b's.

If there is no such i, but we are in the language, the only possible way is that all prefixes contain strictly more a's, then b's, specifically for $i = 1$ as well, which means that $w_1 = a$. We can generate this letter by using the first and the fourth production rules as such $T \to TT \to aT$, where T would have to generate the word $w_{2..n}$, which is of length $n - 1$, which can be generated via T according to the induction hypothesis.

If there is such i, then $x_i \neq x_1$, since i is the first time the number of a's is equal to the number of b's.

Then, we can use the first, then either the second or the third production rule to generate the characters x_1 and x_i, such as $T \to TT \to x_1Tx_iT$. Where the remainder two parts of the word $w_{2..i-1}$ and $w_{i+1..n}$ are also in the language and can be generated using T, since their lenght is less than n, according to the induction hypothesis.

b.)

$$R \to TaT$$
$$T \to TT|aTb|bTa|a|\varepsilon$$

T is the same as in a.), and due to R, now the number of a's must be strictly greater than the number of b's.

3.12 Session 3, Exercise 12

Exercises

Determine the language generated by this grammar.

$$
\begin{aligned}
R &\to XRX|S \\
S &\to aTb|bTa \\
T &\to XTX|X|\varepsilon \\
X &\to a|b
\end{aligned}
$$

Solution

Let's get rid of X first:

$$
\begin{aligned}
R &\to aRa|aRb|bRa|bRb|S \\
S &\to aTb|bTa \\
T &\to aTa|aTb|bTa|bTb|a|b|\varepsilon
\end{aligned}
$$

Then S:

$$
\begin{aligned}
R &\to aRa|aRb|bRa|bRb|aTb|bTa \\
T &\to aTa|aTb|bTa|bTb|a|b|\varepsilon
\end{aligned}
$$

- R generates a string to its left and to its right, of the same length using the "one to the left, one to the right" method.
- Importantly, when it changes into T, only two transitions are allowed: where the generated characters are different.
- Then, T continues generating a string to its left and to its right, of the same length using the "one to the left, one to the right" method.
- Then it terminates in either a, b, or ε.

This is the opposite of a palindrome generator: due to that transitioning from R to T, there must be at least one position where the palindromness of the string is broken. Other positions can be either matching, or non-matching, but there will be at least one, where the characters don't match.

Proof:

Any non-palindrom can be generated with this: since it is a non-palindrom, there is a position where the mirrored position contains the wrong character. Use the first 4 rules until we reach that position, then use the 5th or the 6th rule to move to T, then continue using rules 7 to 10, until we are left with a single character (for a string of odd length), then use rule 11 or 12, or no characters (for a string of even length), then use the last rule.

Any palindrom can not be generated, since there is no way to transition R into a T (no position breaks the palindromness), so the generation can never terminate.

4 March 9th (Session 4): Context-free grammars, Pushdown automata

4.1 Sessions 4 and 5, Exercise 1

Exercise

Let the grammar be

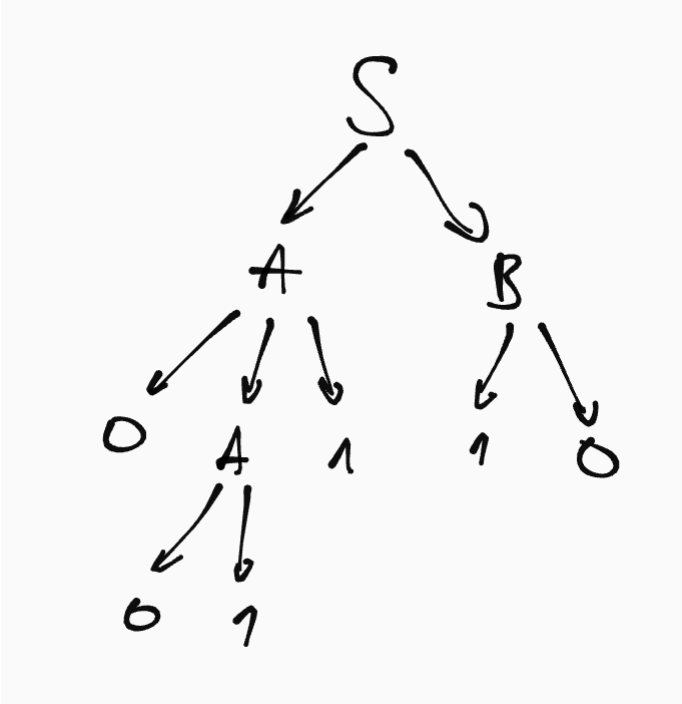
$$
\begin{aligned}
S &\to AB \\
A &\to 0A1|01 \\
B &\to 1B0|10
\end{aligned}
$$

(a) Give a parse tree for word 001110.

(b) Determine the language generated by this grammar.

Solution

a)



b)

A generates $L_A = \{0^k 1^k | k \geq 1\}$, B generates $L_B = \{1^j 0^j | j \geq 1\}$, and S concatenates the two, so $L_S = \{0^k 1^{k+j} 0^j | j, k \geq 1\}$.

4.2 Sessions 4 and 5, Exercise 2

Exercise

Consider

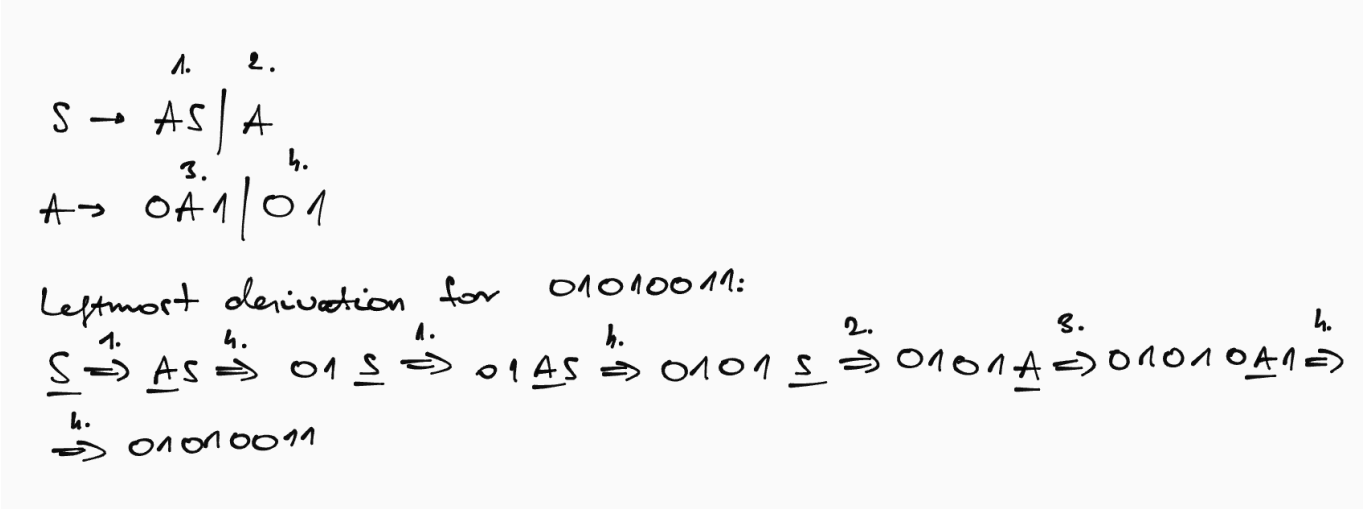
$$S \to AS|A$$
$$A \to 0A1|01$$

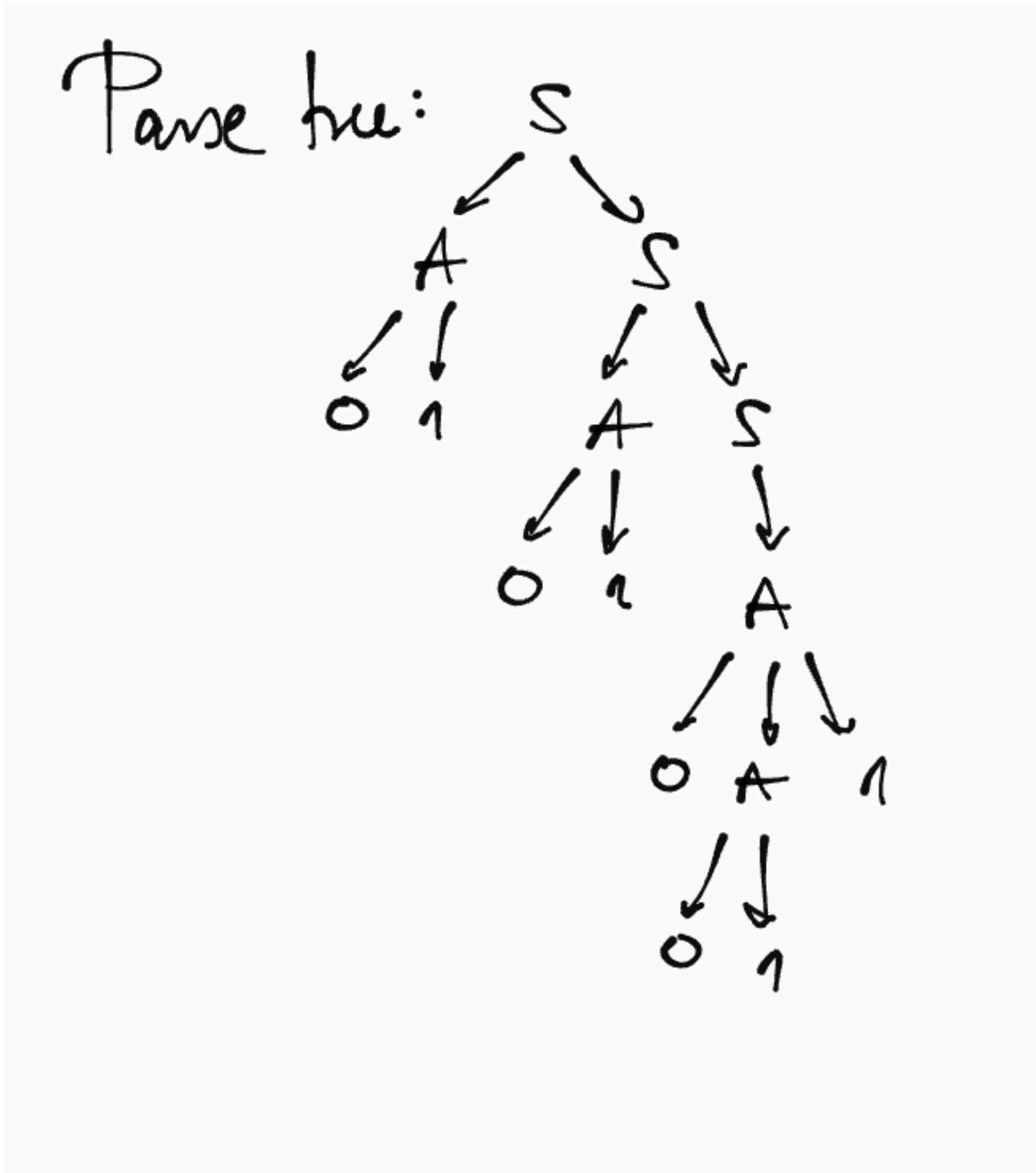
(a) Give a parse tree and a leftmost derivation for word 01010011.

(b) Determine the language generated by this grammar.

Solution

a)





b)

From A we can derive the following set of words: $L_A = \{0^n 1^n | n \geq 1\}$, and S allows for conatenating A's, so $L_S = \{0^{n_1} 1^{n_1} 0^{n_2} 1^{n_2} \dots 0^{n_k} 1^{n_k} | k \geq 1, n_1, n_2 \dots, n_k \geq 1\}$.

4.3 Sessions 4 and 5, Exercise 3

Exercise

Give CF-grammars for the following languages. Are these unambiguous?

(a) $L = \{a^n b^{n+1} | n \geq 0\}$

(b) $L = \{a^n b^{2n} | n \geq 0\}$

(c) Palindromes over $\{a, b\}$

(d) $L = \{a^i b^j c^k | (i = j \text{ or } i = k) \text{ and } i, j, k \geq 0\}$

Solution

(Multiple solutions can be correct here, we give one example for each.)

a)

$$\begin{aligned} S \rightarrow & Tb \\ T \rightarrow & aTb|\varepsilon \end{aligned}$$

$L_T = \{a^n b^n | n \geq 1\}$, and S just adds one more b at the end. This language is unambiguous, since for **any** word in the grammar there is only a single possible (leftmost) derivation: we must always first use the $S \rightarrow Tb$ production rule, then the number of a's and b's determines the number of times the $T \rightarrow aTb$ production rule is used, finally the $T \rightarrow \varepsilon$ must end the derivation.

Another good solution is $S \rightarrow aSb|b$, which does exactly the same thing with less rules.

b)

$$S \rightarrow aSbb|\varepsilon$$

Since we now need twice as many b's as a's. This grammar is also unambiguous, since there is only one possible (leftmost) derivation for any word in the language: we use the $S \rightarrow aSbb$ as many times as there are a's in the word, and then finally use $S \rightarrow \varepsilon$.

c)

$$S \rightarrow aSa|bSb|a|b|\varepsilon$$

The matching characters in the palindromes are generated with rules $S \rightarrow aSa$ and $S \rightarrow bSb$, then if the palindrome is of odd length, the middle character is generated with rules $S \rightarrow a$ and $S \rightarrow b$, while if the palindrome is of even length, then no middle character is needed, so $S \rightarrow \varepsilon$ is used.

This grammar is unambiguous, since for any given palindrome, there is only one possible leftmost derivation for it: we read the input word from left-to-middle and when we see a character a we must use production rule $S \rightarrow aSa$, when we see a b we must use $S \rightarrow bSb$, then we take care of the middle character as we said before.

d)

L is the union of $L_1 = \{a^i b^i c^k | i, k \geq 0\}$ and $L_2 = \{a^i b^j c^i | i, j \geq 0\}$. So we can create a grammar by creating two independent grammars for L_1 and L_2 and combining them:

For L_1:

$$\begin{aligned} X \rightarrow & TC \\ T \rightarrow & aTb|\varepsilon \\ C \rightarrow & cC|\varepsilon \end{aligned}$$

For L_2:

$$\begin{aligned} Y \rightarrow & aYc|B \\ B \rightarrow & bB|\varepsilon \end{aligned}$$

For $L = L_1 \cup L_2$:

For L_1:

$$
\begin{aligned}
S &\to X|Y \\
X &\to TC \\
T &\to aTb|\varepsilon \\
C &\to cC|\varepsilon \\
Y &\to aYc|B \\
B &\to bB|\varepsilon
\end{aligned}
$$

This grammar is ambiguous, since the words of $a^i b^i c^i$ can be derived from both variable X and variable Y as well, so they have at least two leftmost derivations or parse trees.

4.4 Sessions 4 and 5, Exercise 4

Exercise

Are the following grammars unambiguous? Are the languages generated by them unambiguous?

a)

$$S \rightarrow aSa|bSb|aa|bb|a|b$$

b)

$$\begin{aligned} S &\rightarrow TT|U \\ T &\rightarrow 0T|T0|\# \\ U &\rightarrow 0U00|\# \end{aligned}$$

Solution

a)

The grammar is unambiguous, it generates non-empty string palindromes, for any given palindrome, there is only one possible leftmost derivation for it: we read the input word from left-to-middle and when we see a character a we must use production rule $S \rightarrow aSa$, when we see a b we must use $S \rightarrow bSb$, then we take care of the middle character as we said before. If the palindrome is of odd length, the middle character is generated with rules $S \rightarrow a$ and $S \rightarrow b$, while if the palindrome is of even length, we stop at the last two remaining characters, which are either aa or bb and use production rule $S \rightarrow aa$ or $S \rightarrow bb$ to generate them.

Since there exists an unambiguous grammar for this language the language itself is unambiguous.

b)

The language of this gammar:

$L_T = \{0^i\#0^j|i,j \geq 0\}$

$L_{TT} = \{0^i\#0^j0^k\#0^l|i,j,k,l \geq 0\} = \{0^i\#0^m\#0^l|i,m,l \geq 0\}$

$L_U = \{0^n\#0^{2n}|n \geq 0\}$

$L_S = \{0^i\#0^m\#0^l|i,m,l \geq 0\} \cup \{0^n\#0^{2n}|n \geq 0\}$.

For any given word in the language, if the word contains 1 #'s, then we must use production rule $S \rightarrow U$ at the beginning, while if the word contains 2 #'s, then we must use production rule $S \rightarrow TT$ at the beginning, so so far this choice is determined. However, there will be an issue in L_{TT}, since the middle zeroes can be generated by either the first or the second T variable, so there will be multiple leftmost derivations for some words in the language (E.g. write two leftmost derivations for the word 0#0#0!). So this grammar is ambiguous.

However, the language is unambigous, and to show this we show a different grammar for the same language that is unambigous.

$$\begin{aligned} S &\rightarrow N\#N\#N|U \\ N &\rightarrow 0N|\varepsilon \\ U &\rightarrow 0U00|\# \end{aligned}$$

In this grammar for any given word in the language:

If the word contains 1 #'s, then we must use production rule $S \rightarrow U$ at the beginning, then we must repeat $U \rightarrow 0U00$ as many times as there are 0's before the #, finally the $U \rightarrow \#$ rule must be used to end the leftmost derivation, there is no other way to generate these types of words.

If the word contains 2 #'s, then we must use production rule $S \rightarrow N\#N\#N$ at the beginning, then for each individual variable N we must use the rule $N \rightarrow 0N$ as many times as there are 0's in that section, and finally use $N \rightarrow \varepsilon$ at the end.

4.5 Sessions 4 and 5, Exercise 5

Exercise

Let the alphabet be $\Sigma = \{0, 1\}$, states of the pushdown automaton be $Q = \{A, B, C\}$, where A is the start state, C is the only accept state, let Z be the start symbol of the stack. The transition function is the following:

1. $\delta(A, 0, \varepsilon) = \{(A, a)\}$
2. $\delta(A, 1, \varepsilon) = \{(A, b)\}$
3. $\delta(A, \varepsilon, \varepsilon) = \{(B, \varepsilon)\}$
4. $\delta(B, 0, a) = \{(B, \varepsilon)\}$
5. $\delta(B, 1, b) = \{(B, \varepsilon)\}$
6. $\delta(B, \varepsilon, Z) = \{(C, \varepsilon)\}$

a) Give the possible computations of the automaton on word 010.

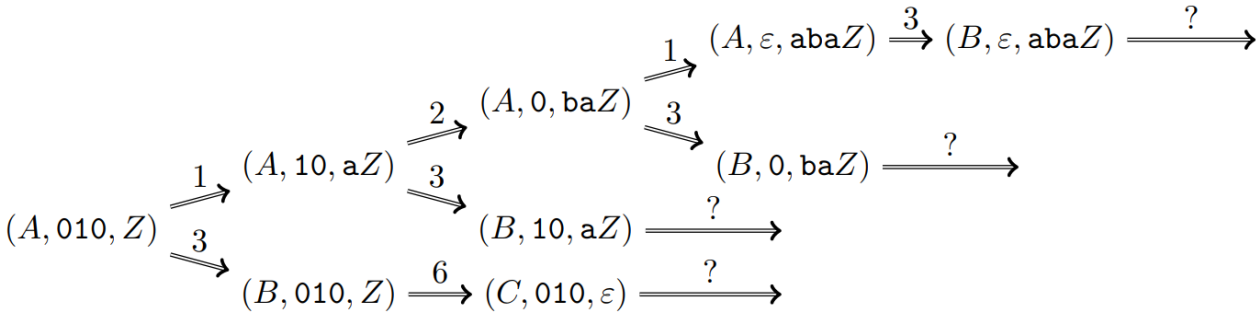
b) Does it accept word 0110?

c) What is the language recognized by the automaton?

d) Give a CF-grammar for this language.

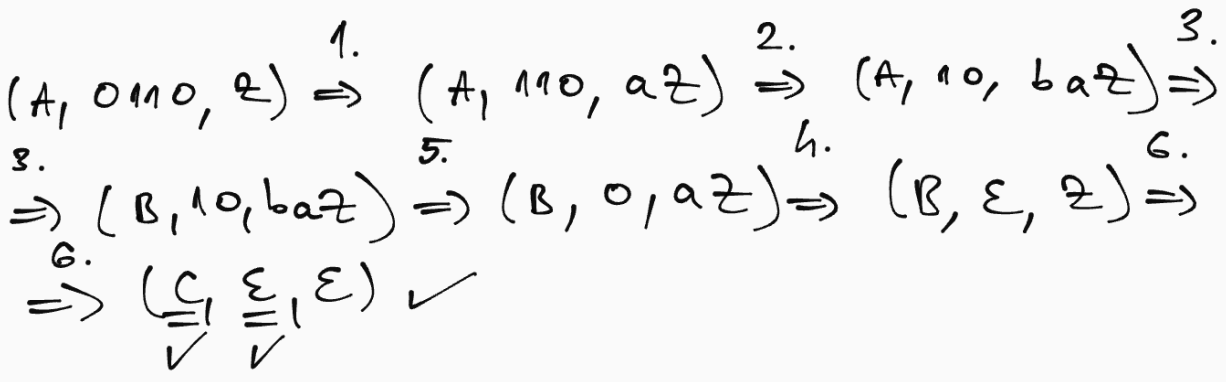
Solution

a)



b)

Yes, for example an accepting comutation (branch) is the following:



c)

The language recognized is palindromes of odd length. These are accepted, since an accepting computation for these types of words will be: for the first half of the word transitions 1. and 2. will push the word ($0 = a$, $1 = b$) into the stack, then transition 3. will move to variable B, where the second (mirrored) half of the word will be compared to the first half on the stack using transitions 4. and 5.. Since a stack is last-in-first-out, the comparisons will happen in the

correct order. The stack can be emptied out this way and that allows for transition 6. to occur on the stack bottom symbol and move to the acceting C state.

Words not in this language are rejected, since:

- Only words of even length have a chance to be accepted, since the stack must be emptied out to move to the only accepting state and each character we put on the stack must have a pair that we use to remove it from the stack.
- Transition 3. must occur exactly at the middle of the word, for the same reason as stated above: if the word's length is $2n$, n characters must be put on the stack, then move to state B, where n characters must be removed from the stack. If the transition occurs too late, there will be remaining characters on the stack and we won't be able to move to state C, while if the transition occurs too early, the stack might be emptied out, however there will be remaining characters on he input so we could move to state C, however we won't accept since there are remaining characters on the input.
- If the word is not a palindrome, there must be at least one position where the character on it's mirrored position is wrong, either a paired with a b or b paired with an a. This means, that when reading the stack back in state B, when we reach this position, we will have a wrong pairing: 0 on the input with a character b on the stack, or 1 on the input with a character a on the stack, for which no transition is defined from B, thus the machine will stop and the stack will not be empty, so we cannot move to state C and B is a rejecting state (plus, there will also be remaining input as well).

d)

$$S \rightarrow aSa|bSb|\varepsilon$$

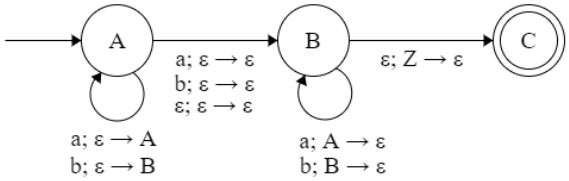
We have shown in previous exercises that this is indeed the grammar for palindromes of even length.

4.6 Sessions 4 and 5, Exercise 6

Exercise

Construct a pushdown automaton for the language of palindromes.

Solution



Proof:

States meaning:

- State A is used to read the first half of the word, and store it on the stack.
- State B is used to read the second half of the word and compae it to the first half of the word on the stack.
- State C can only be reached by emptying the stack out, so only palindromes can reach it.

Transitions:

- State A's loop transitions will store the corresponding characters on the stack.
- Transition from A to B takes care of the middle character, in case of an odd length palindrome, or is an epsilon transition in case of an even length palindrome.
- State B's loop transitions will compare the second half of the word with the first half (mirrored, since a stack is a LIFO), and will only remove characters from the input and the stack if they match. If there is a character on the stack that doesn't match the PDA will halt in state B, which will reject.
- If the stack is emptied out we move to state C. In this case, if we read the entire input we can be sure that the word's first half matches the second half, and we will accept. If there are charaters remaining on the input, it means that not the entire word, only the prefix of the word was a palindrome, in which case the PDA correctly rejects, since there is still input remaining.

Accept / reject states:

The only accept state is state C which can only be reached with no input remaining if the first half of the input is the mirror of the second half of the input. If a word is not a palindrome there won't be any accepting branch in the computation, all branches will end up in either A (store the entire word), or B: start comparing too late, stack is not empty to move to C, or C but with input remaining, since we started comparing too early.

4.7 Sessions 4 and 5, Exercise 7

Exercise

Create pushdown automata for the following languages.

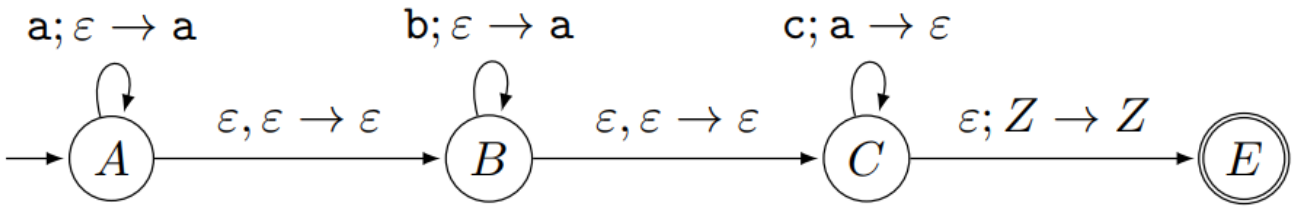
a) $L_a = \{a^i b^j c^k | i, j, k \geq 0 \text{ and } i + j = k\}$

b) $L_a = \{a^i b^j c^k | i, j, k \geq 0 \text{ and } j + k = i\}$

c) $L_a = \{a^i b^j c^k | i, j, k \geq 0 \text{ and } i + k = j\}$

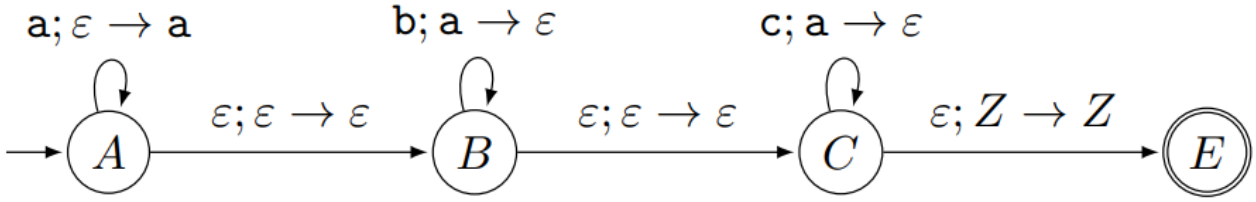
Solution

a)



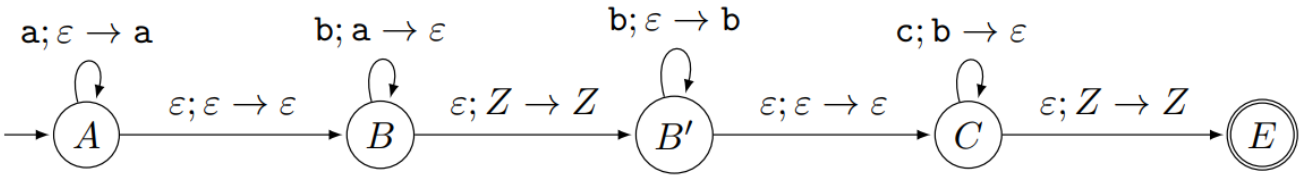
TODO Proof

b)



TODO Proof

c)



TODO Proof

Note: c) is the tricky one, since we need the number of a's plus the number of c's to be equal to the number of b's, but the b's come in-between the a's and the c's. In this case we split the processing of B's into two parts: in the first part we compare the number of b's to the a's that came before, while in the second part we store the number of b's to be compared with the number of the upcoming c's.

In this case we used non-determinism heavily, since the transition between B and B' must happen at the correct time for the calculation to work: one computational branch will time it correctly and that one can become an accepting branch (if everything else is correct with the word).

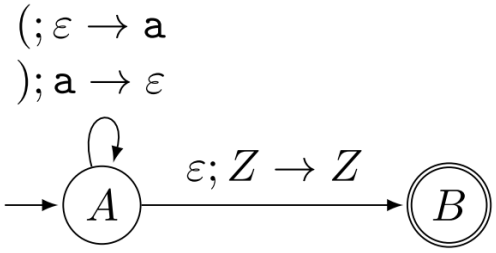
4.8 Sessions 4 and 5, Exercise 8

Exercise

Create a pushdown automaton for the language of proper parenthesisations.

Solution

Idea: correct parenthesisation can be checked by counting from left to right: start with 0 and add +1 for every (encountered and −1 for every) encountered. The parenthesisation is correct if the sum never drops below 0 during the calculation (too many) in that case at that point) and at the end it equals to 0. A simple PDA can be constructed to do exactly this calculation.



TODO Proof

4.9 Sessions 4 and 5, Exercise 9

Exercise

Construct pushdown automata for the following languages.

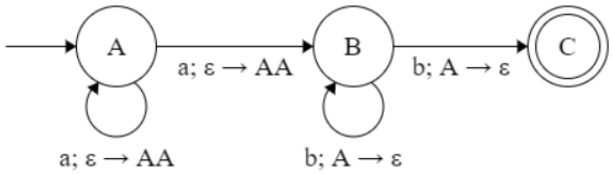
a) $L_a = \{a^n b^m | 2n = m \geq 1\}$

b) $L_b = \{a^n b^m | 2n \geq m \geq n \geq 1\}$

Solution

a)

The first one is simpler, since we need to check for twice as many b's as a's. We can count the number of a's in the stack with TWO tokens instead of one, so then the number of tokens in the stack must be equal to the number of b's.



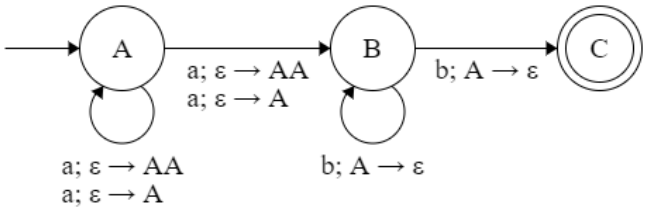
To enforce that the loops run at least once, due to the requirement that $n, m \geq 1$ ($n \geq 0.5$ is the same as $n \geq 1$, since n is an integer), we used non-epsilone transitions between the states, and copied the loop's transition to the existing transitions from the states as well.

TODO Proof

b)

The second one is more complex, since the number of b's is now anywhere between $2n$ and n! How do we enforce $2n \geq m \geq n$? We just did it with $2n = m$, also if it were only $n = m$, we could replace the $a; \varepsilon \to AA$ transitions with $a; \varepsilon \to A$ transitions. But how do we check for in-between these two numbers?

We will rely on non-determinism again:



We use both $a; \varepsilon \to A$ and $a; \varepsilon \to AA$. The PDA will nondeterministically use either the first or the second transition and push either 1 or 2 A's on the stack. So the number of A's on the stack will be anywhere between $2n$ and n, and there will exist a computational branch (if the word is in the language) where their number will be equal to m and the word will be accepted.

4.10 Sessions 4 and 5, Exercise 10

Exercise

TODO

Solution

TODO

4.11 Sessions 4 and 5, Exercise 11

Exercise

TODO

Solution

TODO

4.12 Sessions 4 and 5, Exercise 12

Exercise

TODO

Solution

TODO

5 March 23th (Session 5): Turing-machines

5.1 Session 5, Exercise 1

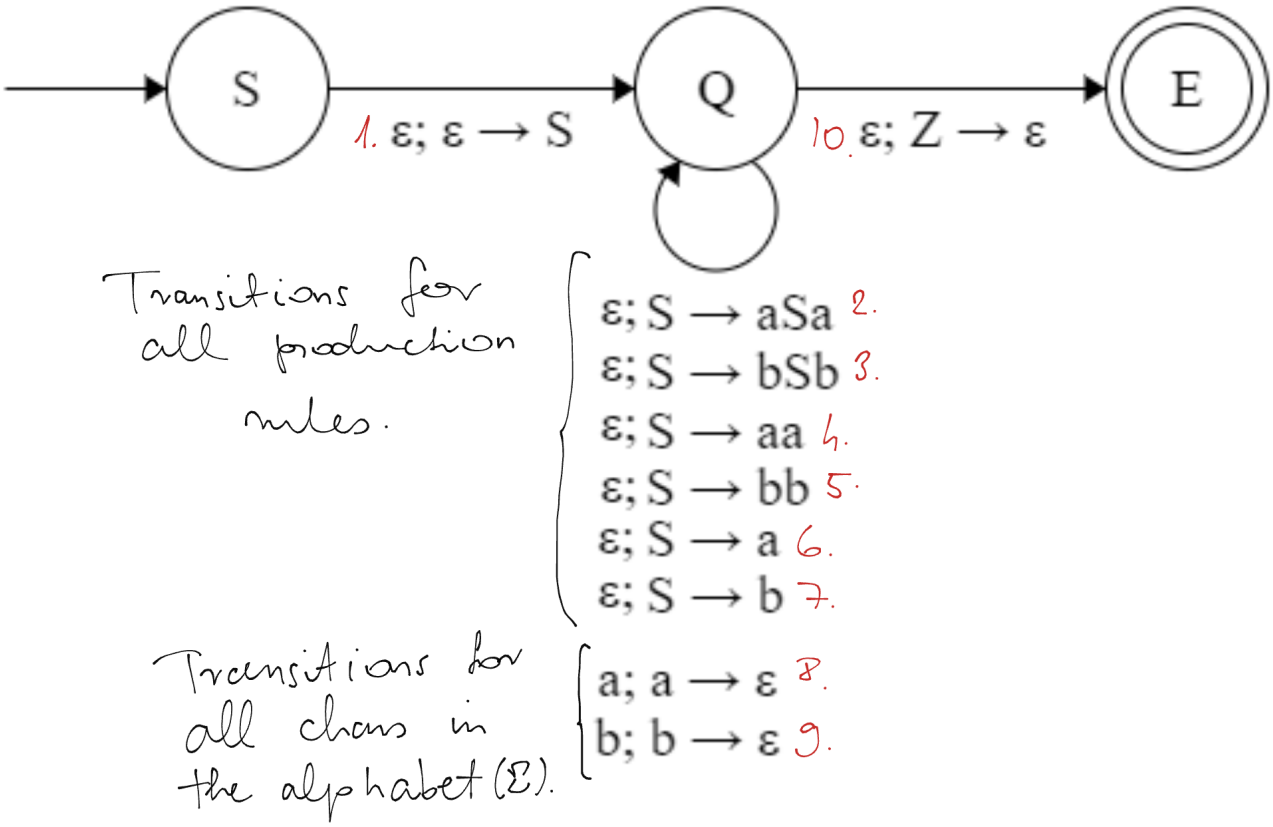
Exercise

Create a PDA from the grammar $S \to aSa|bSb|aa|bb|a|b$ and give an accepting computation for the word *ababa* (if such one exists).

Solution

a)

Using the schema for turning a CF-grammar into a PDA:



- Transition 1: First, we put the starting variable S on top of the stack.
- Transitions 2-7: For all production rules in the grammar we add transitions, which will take care of doing the (leftmost) derivation inside the stack, while not reading from the input.
- Transitions 8-9: For all letters in the alphabet we add production rules, which will take care of comparing the input with the derived word in the stack.
- Transition 10: Finally we check if the stack is empty and only move to the accept state E if it is.

b)

Accepting computation for *ababa*:

$(S, ababa, Z) \xrightarrow{1.} (Q, ababa, SZ) \xrightarrow{2.} (Q, ababa, aSaZ) \xrightarrow{8.} (Q, baba, SaZ) \xrightarrow{3.} (Q, baba, bSbaZ) \xrightarrow{9.} (Q, aba, SbaZ) \xrightarrow{6.} (Q, aba, abaZ) \xrightarrow{8.} (Q, ba, baZ) \xrightarrow{9.} (Q, a, aZ) \xrightarrow{8.} (Q, \varepsilon, Z) \xrightarrow{10.} (E, \varepsilon, \varepsilon)$

5.2 Session 5, Exercise 2

Exercise

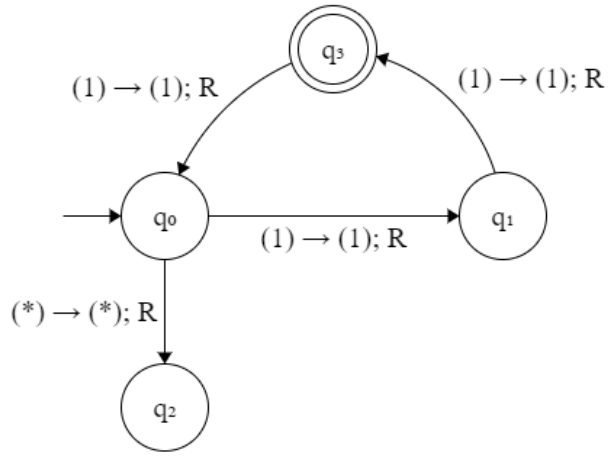
Let the transition function of the 1-tape Turing-machine be

$$\delta(q_0, 1) = (q_1, 1, R)$$
$$\delta(q_0, *) = (q_2, *, R)$$
$$\delta(q_1, 1) = (q_3, 1, R)$$
$$\delta(q_3, 1) = (q_0, 1, R)$$

start state is q_0, accept state is q_3. What is the language recognized by this machine?

Solution

It's easier to see what's happening if we draw the machine:

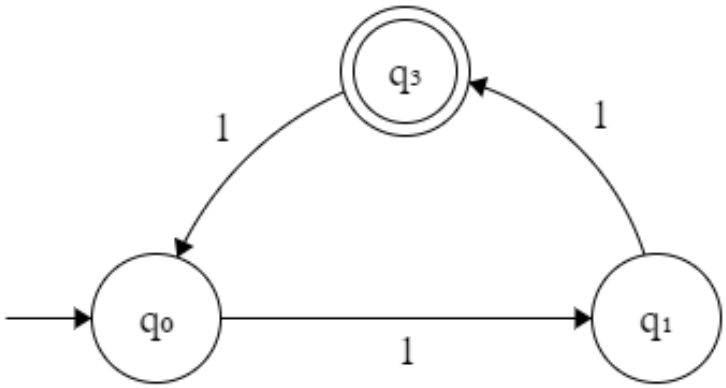


It the q_0, q_1, q_3 cycle reads the input word on the tape from left to right and counts the number of 1's modulo 3. In q_0 the remainder is 0, in q_1 the remainder is 1 and in q_3 the remainder is 2.

When the input is ε (the empty string) the $q_0 \xrightarrow{(*)\to(*);\ R} q_2$ transition will move the machine to q_2 and it will halt there, since there is no transitions defined, thus reject the empty string input. (This transition could have been left out, since then the machine would remain in q_0 and reject the empty string input similarly.)

Since the accept state is q_3, words containing $3k + 2$ number of 1's will be accepted, where $k \geq 0$.

Notice how the above Turing-machine does exactly the same thing as this Finite Automaton:



We have seen similar modulo counter automatons on the first practice session.

5.3 Session 5, Exercise 3

Exercise

The following table contains the transition function of a 2-tape Turing machine, where $*$ is the blank symbol on the tapes, and q_0 is the start state.

state	tape 1	tape 2	tape 1		tape 2		new state
q_0	0	$*$	0	S	X	R	q_1
	1	$*$	1	S	X	R	q_1
	$*$	$*$	$*$	S	$*$	S	q_5
q_1	0	$*$	0	R	0	R	q_1
	1	$*$	1	R	1	R	q_1
	$*$	$*$	$*$	S	$*$	L	q_2
q_2	$*$	0	$*$	S	0	L	q_2
	$*$	1	$*$	S	1	L	q_2
	$*$	X	$*$	L	X	R	q_3
q_3	0	0	0	S	0	R	q_4
	1	1	1	S	1	R	q_4
q_4	0	0	0	L	0	S	q_3
	0	1	0	L	1	S	q_3
	1	0	1	L	0	S	q_3
	1	1	1	L	1	S	q_3
	0	$*$	0	S	$*$	S	q_5
	1	$*$	1	S	$*$	S	q_5

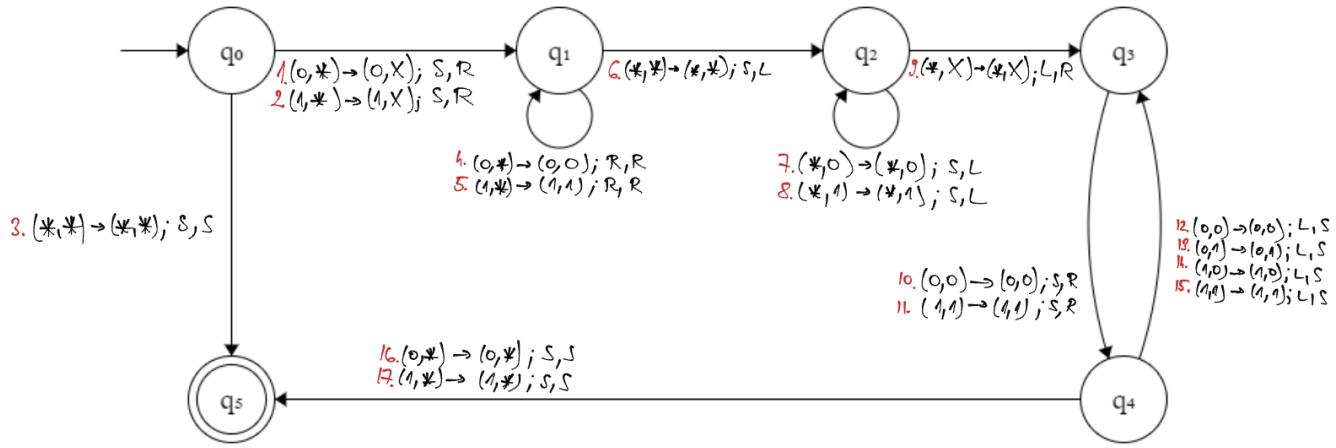
a) What is the content of tape 2 when the machine moves to state q_2?

b) What is the language $L(M)$ if the only accept state is q_5?

c) At most how many steps are done by the machine on an input of length n before it stops?

Solution

To see a little bit easier what this TM does, I converted it line-by-line to the below form:



What this does:

- Transition 3: This one handles the empty string input and immediately moves to the accept state q_5. So the empty string is accepted. So the empty string is accepted in 1 step. For any other input, the following happens:

- Transitions 1-2: They put down an X character at the beginning of the second tape. This X will be used later to make sure that the head does not fall of the tape (moving from the first position to the left would cause the head to fall off, this X is there so we can detect it and make sure no transition is defined that moves the 2nd head to the left when it reads an X). This is 1 step.
- Transitions 4-5: They copy the first tape's contents to the second tape. This is n steps if the input is of length n.
- Transition 6: When the on the first tape the head is at the end of the input word (sees the $* =$ empty cell character we move to state q_2 and we position the second head on the last character of the copied input. (While the first head remains on the $* =$ empty cell after the input.) This is 1 step.

a) When we move to state q_2 the contents of tape 2 will be the character X at the beginning, then the input word copied afterwards.

- Transitions 7-8: The first head stays on the same $*$ cell, while the second had moves to the left until it finds the character X (beginning of the second tape). This is n steps if the input is of length n.
- Transition 9: The first head is positioned at the end of the input word, while the second head is positioned at the beginning of the (copied) input word. This is 1 step.
- Transitions 10-15: Together these make $2n$ steps for an input word of length n, see explanation below:
  - Transitions 10-11: These compare the two strings (the input word twice, on the first tape from right to left and on the second tape from left to right). However, the first head is not moved immediately to the left. This is because the first head could fall off the first tape, since there is no protective X there. We cannot recklessly move to the left.
  - Transitions 12-15: Instead we lag behind the second head and make sure that the second head reads either a 0 or a 1 (and the first head can read 0 or 1 as well), which means that the word has not yet ended! (The second head would read $*$ here when the word ends.) This means that it is **safe** to move the first head to the left, since it is not yet at the beginning of the tape, so we move.
- Transitions 16-17: Finally, when the second head reads the empty cell, it means that the word has ended (and has been successfully compared), so we can move to q_5. We make sure that we **DO NOT** move the first head to the left in this step, since it would fall off. Since the first head could be on a character 0 or a 1 we need to define 2 transitions to cover all possibilities. We move to q_5 here. This is 1 step.

Together the number of steps for a successfull computation has been (for a non-empty input):

- Transitions 1-2: 1 step.
- Transitions 4-5: n steps.
- Transition 6: 1 step.
- Transitions 7-8: n steps.
- Transition 9: 1 step.
- Transitions 10-15: $2n$ steps.
- Transitions 16-17: 1 step.

At most $4n+4$, but keep in mind that for a rejecting computation the number of steps is smaller, depending on where it halts in transitions 10-15. (At least $2n+3$ steps, since copying and moving the second head back to the first position will be done regardless of rejecting / accepting.)

5.4 Session 5, Exercise 4

Exercise

Let language L consist of words over $\{0, 1\}$ whose middle character is 0. Prove that L is CF.

Solution

We prove that L is CF by giving a CF-grammar. (We could also give a PDA.)

$$S \rightarrow 0S0|0S1|1S0|1S1|0$$

This grammar is similar to the odd-palindrome one, however we allow all possible combinations of letters. But the middle one must be a 0.

Proof: this CF-grammar generates L.

Any word in the language has to be of odd length. We look at the first and last characters and use one of the four transitions from $S \rightarrow 0S0|0S1|1S0|1S1$ to generate them, then we remove those from the word and keep repeating until 1 character is left. This character is the middle one, since we removed an equal number of characters before and after it (1-1 in each step), so it has to be a 0, which can be generated using production rule $S \rightarrow 0$.

Any word outside of this language is either:

- of even length, in which case there is no possilbe way for this grammar to generate it, since rules $S \rightarrow 0S0|0S1|1S0|1S1$ generate an even number of characters, while the $S \rightarrow 0$ rule ends the derivation and can be used only once, which results in an odd lenght word. This includes the ε (empty string), which cannot be generated, since there is no $S \rightarrow \varepsilon$ rule.
- of odd length, but with the 1 middle character. Using the previous explanation, since there is no $S \rightarrow 1$ rule, there is no way to put a 1 in the middle, so these words can't be generated either.

We have shown that this grammar generates **the correct words** and **only the correct words = does not generate words outside of the language**.

5.5 Session 5, Exercise 5

Exercise

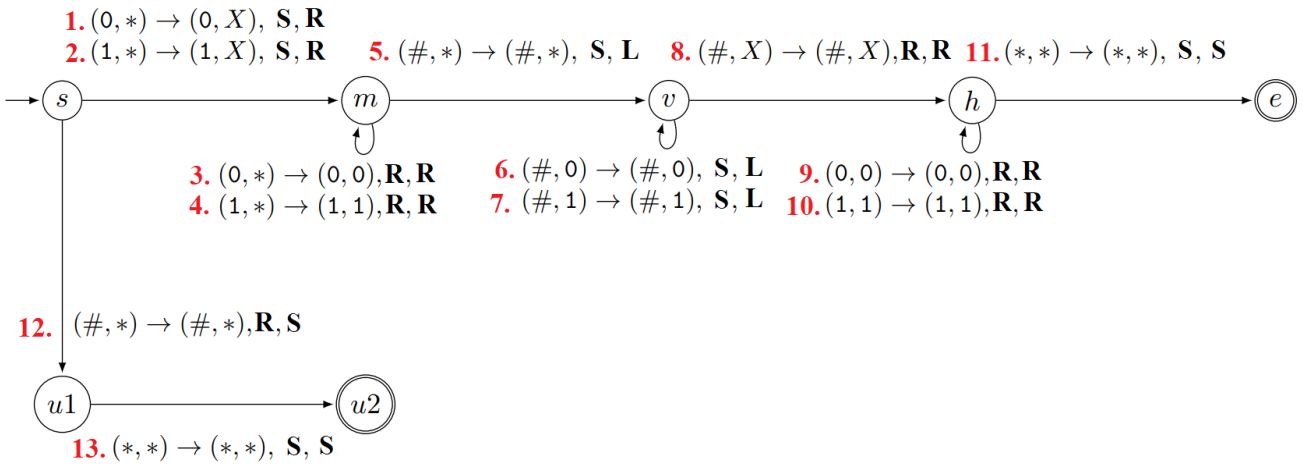
Give a Turing-machine for the language

$$L = \{w\#w | w \in \{0,1\}^*\}.$$

Give an upper bound for the running time.

Solution

This is the Turing-machine:



Proof: This machine's language is L.

States and transitions:

- State s: The starting state, the empty string will remain here.
- Transitions 1 and 2: will put an X at the beginning of the second tape, for protection against failling off.
- State m: Used for copying the input word up until the first # to the second tape.
- Transitions 3-4: Copy a 0 or a 1 to the second tape, move both heads to the next position.
- Transition 5: When we encounter the first # we stop copying and move to state v, while positioning the second head back to the last copied character. If there is no # in the word the computation halts in state m and rejects the input correctly.
- State v: Used for moving the second head back to the beginning of the second tape.
- Transitions 6-7: The first head will stay on the character #, while the second head moves to the left while it sees a 0 or a 1.
- Transition 8: When the second head sees the X at the beginning of the second tape it reached the beginning of the tape. The heads are positioned, so the first head is at the beginning of the (copied) input, while the second head is at the first character after the # in the input.
- State h: Used to compare the first part of the input (before the #) to the second part of the input (after the #).
- Transitions 9-10: We make sure that all characters match in the fist and the second part of the inputa word, so that it is in $w\#w$ form.
- Transition 11: Only allows moving to the accepting e state if the the two heads reach the empty cells at the same time = if the two parts of the input word match. If the word contains another #, or if the two parts don't match, then this transition won't be able to fire and we won't be able to move to state e.
- State e: Only words in the language can reach this state to accept. No further computation needed.

- Transition 12: There is a special word, #, which the generic computation above cannot handle, since it has no 0 or 1 characters, however it should be accepted, since w is allowed to be the empty string. We handle this case specially, we detect that on the input we see the # character, and
- Transition 13: Detects that there is no further characters in the input, so the input word is # exactly.
- States $u1$ and $u2$: Used for handling this special case, $u2$ will accept the # only.

Accepting / rejecting states, accepted / rejected words:

- If the word starts with # character it is moved to $u1$. It can only be accepted if the word is # exactly, no further characters can come. # is moved to $u2$ and accepted, while the others remain in $u1$ and get rejected there.
- If the word starts with some 0 or 1 characters, those get copied to the second tape.
- If the word contains no # character then it stops in state m and gets rejected.
- If the word contains a # character, the characters coming after should match what characters were before. So any word in the form $w\#w$ is accepted, however any $w\#h$, where $w \neq h$ will stop in state h and will be rejected. This includes the words that contain more #'s (in h) and words that contain only 0's and 1's in h, but not match w, since the only possible way to activate transition 11 is to have matching characters before the # and after it. State e accepts these words.

Upper bound for the running time: the longest running time will be an accepted word that reachers state e. If the word is $w\#w$, of lenght $n = 2k + 1$, so that w is of length k, then:

- Transitions 1-2: 1 step, to put down an X.
- Transitions 3-4: k steps, to copy w to the second tape.
- Transition 5: 1 step to detect the # and position.
- Transitions 6-7: k steps to move the second head to the first position on the second tape.
- Transition 8: 1 step to detect the beginning of the second tape and position the heads.
- Transitions 9-10: k steps to compare the two parts.
- Transition 11: 1 step to see that they end at the same time.

$1 + k + 1 + k + 1 + k + 1 = 3k + 4$ steps, where $n = 2k + 1$, so $k = \frac{n}{2} - 1$, so $3k + 4 = 3(\frac{n}{2} - 1) + 4 = \frac{3n}{2} + 1 \in O(n)$. For accepted words this is $\Theta(n)$, for rejected words not necessarily, for example the input word $\#0^k$, for any large k will be rejected after 1 step, in $u1$.

5.6 Session 5, Exercise 6

Exercise

Let L_r be an arbitrary regular language and let L_c be an arbitrary CF language.

1. Show an example when $L_r \cap L_c$ is not regular.
2. Prove that $L_r \cap L_c$ is always context-free.
3. Show an example when L_1 and L_2 are both context-free but $L_1 \cap L_2$ is not.

Solution

a)

For example if we take $L_r = \Sigma^*$, which is regular. Then, we take a known non-regular, but CF language, $L_c = \{a^n b^n | n \geq 0\}$. Ther instersection is L_c itself, which is not regular, but CF.

b) If L_r is regular, there exists a DFA that accepts it, let's call this M_r. Then, since L_c is CF, there exists a PDA that accepts it, let's call this M_c. To show that $L_r \cap L_c$ is CF we construct a PDA from M_r and M_c that accepts it.

The main idea is to take all the possible pairs of states, where one state comes from M_r and the other from M_c. For a given input character, we define the transition function, so that "it keeps track of what's happening in both M_r and M_c at the same time", for each statepair (q_r, q_c), by checking what M_r would do for the given input character in state q_r and what would q_c do, and moving to that statepair (or, since the PDA can have a set of possible states it moves into, the set of all statepairs).

We keep track of what's happening in M_c's stack in the stack we have (this is why we cannot do this for two CF-languages, we would need to keep track of two stacks).

The starting statepair is going to be the statepair which contains the starting states from their respective machines.

The accepting statepairs will be the ones for which both states accept in their respective machines, since we need $L_r \cap L_c$.

The PDA constructed in this way will accept $L_r \cap L_c$, which means that the language is CF.

c)

These languages are both CF:

$L_1 = \{a^n b^n c^k | n, k \geq 0\}$ (Number of a's and b's is equal.)

$L_2 = \{a^i b^j c^j | i, j \geq 0\}$ (Number of b's and c's is equal.)

(See 4.3 for a CF-grammar for L_1, and based on that L_2 can be constructed in a similar manner.)

$L_1 \cap L_2 = \{a^n b^n c^n\}$ (Number of a's and b's and c's is equal.)

Which is known to be non-CF. (The idea behind this is that we would need to use the stack to keep track of the number of a's, but we would throw them out when comparing them with the number of b's and there will be nothing left to compare to when the c's come. The formal proof is more complicated and outside of the scope of this class.)

5.7 Session 5, Exercise 7

Exercise

Sketch a Turing machine for the following languages. It is not neccessary to give the transition function precisely, it is enough to describe (in details) the idea of the working of the TM.

a) $L_1 = \{a^n b^m | 2n = m \geq 1\}$

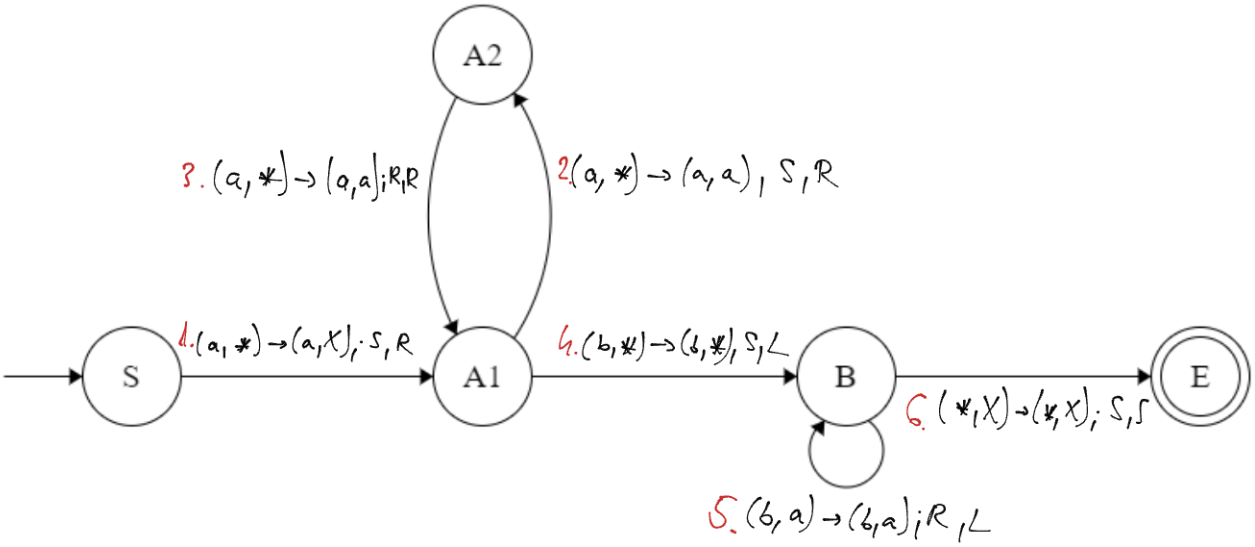
b) $L_2 = \{a^i b^j c^k | i, j, k \geq 1, i + j = k\}$

c) $L_3 = \{a^i b^j c^k | i, j, k \geq 1, i \cdot j = k\}$

Solution

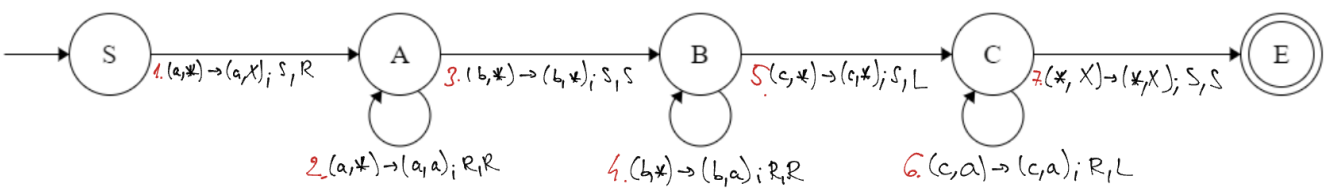
I'm giving the TM's precisely, but you could just describe them. (Basically what the itemized list contains below, without referencing the specific index of the transitions.)

a) Similar to 4.9 a), but now we need to give a TM, instead of a PDA:



- Transition 1: We mark the beginning of the second tape with an X, so we don't fall off later.
- Transitions 2-3: For every a character on the first tape we copy it twice to the second tape. Pay attention: in transition 2 the first head doesn't move, but in transition 3 it does.
- Transition 4: When the a's are done and the b's are coming we position the second head to the end of the copied a's.
- Transition 5: We compare the number of a's and b's on the second and first tape. They have to be equal, which means that the original input had to have half as many a's as b's.
- Transition 6: If the first head reaches the end of the input at the same time the second head reaches the beginning of the second tape (marked with the X), it means that the number of a's and b's is correct and the word can be accepted.

b) Same as 4.7, but now we need to give a TM, instead of a PDA:



- Transition 1: We mark the beginning of the second tape with an X, so we don't fall off later.
- Transition 2: Copy the a's to the second tape.
- Transition 3: Detect that the b's are coming.
- Transition 4: Copy the b's to the second tape, copy them as a's since we only care about their collective numbers and this spares us one extra transition (similar to transition 6, but for a b).
- Transition 5: Detect that the c's are coming.
- Transition 6: Compare the number of a's and b's = the number of a's on the second tape to the number of c's.
- Transition 7: If we run out of a's and c's at the same time it means that there were an equal number of them, so we can accept the input.

Important: Transition 1,3 and 5 enforce that $i, j, k \geq 1$.

c)

The idea is to: Use 3 tapes, mark the beginning of the second and third tape. Copy the a's to the first tape. Then step on the first a on the second tape and run through the first tape copying all of the b's to the third tape. Now step to the left on the second tape, and do the same for the next a, and do it until there are a's left. When you reach the X mark on the second tape, the third tape will contain $i \cdot j$ b's. Now compare the number of b's on the third tape to the remaining c's on the first tape.

It is important to keep in mind that for an odd number of a's the head on the first tape will be at the last b (next to the c), while for an even number of a's the head on the first thape will be at the first b (not next to the c). Handle the latter case by moving the first head to the right until it sees a c.

5.8 Session 5, Exercise 8

Exercise

Solution

5.9 Session 5, Exercise 9

Exercise

Solution

6 March 30th (Session 6): Turing-machines 2

6.1 Session 6, Exercise 1

Same as 5.3.

6.2 Session 6, Exercise 2

Same as 5.5.

6.3 Session 6, Exercise 3

Exercise

Sketch a 1-tape Turing machine for the language $\{0^{2^n} | n \geq 0\}$.

Solution

We have to implement a TM that can divide by 2 on a single tape.

We will be moving back and forth on the single tape, so we need to make sure we do not fall off at the beginning of the tape. We can achieve this by two ways:

- By replacing the first 0 with a special character, for example Z, which indicates both that that is a Z, but also indicates the beginning of the tape. The empty string is not accepted, since $0^{2^0} = 0^1$, so that can immediately go to a special trap state to be rejected.
- By just replacing the first zero with an X and moving the head to the first empty cell, writing a 0 it, then moving back on the X. Handle the empty string similarly as previously.

I will be using the first method, since it will allow for an easier success check later.

We then move the head to the left and at every second step we replace the 0 with a character R, to mark it as removed. This is done by cycling between two states, the first transition just stepping over a 0, the second replacing it with the R. If there is no 0 to be replaced, the division fails (the current number of 0's is odd), which means that we should reject).

When we reach the empty cell we move our head left until we reach Z.

In the upcoming runs we have to do the same thing as previously, however now we need to ignore R's already placed, which can be done by adding two loops on the two states mentioned that just skip over any R characters on the tapes.

Every single run of our subprocess results in dividing the number of 0's by two, which if the number was a power of 2 should end up with a single zero: the specially marked Z character at the beginning of the tape.

When we move to the right from the Z cell and we find an empty cell, that means that we are done and can accept the input.

6.4 Session 6, Exercise 4

Exercise

Sketch a Turing-machine for the language $\{ww|w \in \{0,1\}^*\}$.

Solution

This is similar to 5.5, however we do not have the # to mark the middle of the two words.

This issue can be fixed, in the following way, making a similar 2-tape TM:

- We treat the empty string separately and accept it.
- We mark the beginning of the second tape with an X.
- We copy the **entire** input to the second tape.
- We step backwards on the second tape, but only step backwards on the first tape every second step on the second tape.
- When the second head reaches the X the first head will be in the middle. (And won't fall off the tape, since the empty string is treated for separately, and for every other input, the first head moves less than the second head.)
- Now we can move to the right with both heads and compare the first half of the input (on the second tape) with the second half of the input (on the second tape).
- When the first head reaches the first empty cell it means that all comparisons were successfull and we can move to a separate accept state.

6.5 Session 6, Exercise 5

Exercise

Let $\Sigma = \{0, 1, +\}$. Sketch a Turing machine that on an input of form $x+y$ where x, $y \in \{0,1\}^*$ are nonempty bitstrings stops in finite time and when it stops on its 5th tape the sum of binary numbers x and y stands. Give an upper bound on its running time.

Solution

- Mark the beginning of tapes 2,3,4 tapes with an X.
- We first copy the input up until the + character to the second tape. If no + is found the input is rejected.
- We then copy the second part of the input after the + character until the first emtpy cell. If there is another + found, we reject.
- Now step with heads 2 and 3 1 step backwards, to stand on the least significant bit of both x and y.
- We do the method of summing two numbers we learnt in primary school. We store the current carry bit in our current state: C_0 and C_1. There are 8 (12) possibilities:
  - If the current state is C_0:
    - If both heads see a 0: We write a 0 on the 4th tape and stay in C_0.
    - If one head sees a 0 or an empty cell and the other a 1: We write a 1 on the 4th tape and stay in state C_0.
    - If both heads see a 1: We write a 0 on the 4th tape and move to state C_1.
    - If both heads see an empty cell: Computation is done here, move to the copying stage.
  - If the current state is C_1:
    - If both heads see a 0: We write a 1 on the 4th tape and move to C_0.
    - If one head sees a 0 or an empty cell and the other a 1: We write a 0 on the 4th tape and stay in state C_1.
    - If both heads see a 1: We write a 1 on the 4th tape and stay in state C_1.
    - If both heads see an empty cell: Since we still have a carry bit, we write that down on the 4th tape, then computation is done here, move to the copying stage.
- And if the current head saw a 0 or a 1 we move it to the left for both tape 2 and 3, while the head on tape 4 moves to the right.
- After finishing with the computation we will have the required sum on the 4th tape, however the least significant bit will be on the first place. We need to reverse it.
- We can done this by copying from the current (last) position on the 4th tape to the 5th, by moving the 4th head to the left and the 5th head to the right, step-by-step.

This computation is done in $O(n)$, since copying to tape 2 and 3 is done in $O(n)$, then summing is done in $O(n)$ and finally reversal is also done in $O(n)$. (The resulting sum's length will not exceed the sum of the input x and y number's length in binary form.)

6.6 Session 6, Exercise 6

Exercise

Solve the previous problem for multiplication in place of addition.

Solution

(We implement the algorithm we have learned in primary school for multiplication.)

6.7 Session 6, Exercise 7

Exercise

Sketch a Turing machine that recognizes the complement of the diagonal language. Does it stop on every input?

Solution

L_d, the diagonal language consists of Turing-machine codes that do **not** accept their own codes.

The complement of the diagonal language, $\overline{L_d}$ consits of $0, 1$ bit words, where the words are either:

- The word does not code a Turing-machine (given a selected specific coding method).
- The word does code a Turing-machine, however it accepts its own code as an input.

We first check whether the given input is a Turing-machine code. If it is not, we halt and accept it.

If it is, then we will simulate the Turing-machine based on its code. This is done, by copying the input word to a second tape, which will act as the input tape for the simulated TM. The third tape will keep track of the current state of the simulated TM. In every step, the code of the TM (on the first tape) will tell us what the next step must be, given the contents of the 2nd and the 3rd tapes. We execute this step (change the 2nd simulated input tape accordingly and move the head accordingly on it) and change the simulated current state tape as well.

If the simulated TM halted, we halt as well and we check whether it halted in an accepting state: if it did, we accept, if it did not we reject.

Since it easy to construct a TM, that will end up in an infinite loop for any given input, including its own code, this means that in this case the above simulation will never halt either. This is okay, since the TM that never halts rejected its input and the simulator never halts, so it also rejects the input correctly (TMs that reject their own code are not in $\overline{L_d}$. Thus this TM will not halt for every possible input.

The details of the construction of such TM are out of scope for this class, if you are interested you can look up "Universal Turing machine" for further details.

Side note: we assumed that the input Turing machine code codes a TM that has a single tape. Since there exists a single tape TM that accepts the same language for any multitape TM's, this assumption is acceptable in this case, we could define the TM coding method in a way that we must contruct an equivalent single-tape TM in order to encode any TM.

6.8 Session 6, Exercise 8

Exercise

Solution

7 April 6th (Session 7): P, NP

7.1 Session 7, Exercise 1

Exercise

Let language L consist of undirected graphs that do not contain cycles. Prove that $L \in P$.

Solution

At first glance, we can use the definition of the P language class:

If there exists a...

- **deterministic** Turing-machine
- that accepts the language L
- that **stops on all** possible inputs
- and given a specific input finishes its computation in **polynomial time** relative to the input's size

... then we say that $L \in P$ ("L is in P").

Note:

- When a Turing-machine accepts the language L and stops on all possible inputs, we say that it decides L.
- Versus when a Turing-machine accepts the language L and stops on all **accepted** inputs, we say that it recognizes L. We say that this type of TM can reject a word by never stopping computation on it. However this type of rejection cannot be detected, since we can never know whether the TM will finish eventually or if it is indeed in an infinite computation.

However constructing a Turing-machine can be cumbersome, so we are going to apply the Church-Turing thesis: the TM described above exists if and only if a polynomial-time bound algorithm exists for the problem.

So according to the Church-Turing thesis we can prove that $L \in P$, by describing the algorithm that solves the problem and showing that it's runtime complexity is indeed polynomial relative to the input's size.

Specifically: $L =$ undirected graphs that do not contain cycles.

Three types of inputs are possible:

- A description of an undirected graph that does not contain a cycle. $\rightarrow$ These must be accepted.
- A description of an undirected graph that contains a cycle. $\rightarrow$ These must be rejected.
- An input that does not represent an undirected graph. $\rightarrow$ These must be rejected.

The task did not specify, so we can select the input format: let's say that we want to represent graph G via its adjacency matrix: for n vertices that is an $n \times n$ square matrix containing 0's and 1's, and in row i column j there is a 1 if $\{i, j\}$ is an edge of the graph, and a 0 if it's not an edge. We can set the alphabet to be binary: $\Sigma = \{0, 1\}$.

The algorithm is as follows:

Step 1: (In later exercises we will neglect this step.)

Check the input word's format:

- If the number of the input characters is not a square number, then it does not represent an adjacency matrix of a graph. Reject this type of input.
- If the input is a square matrix, however it is not symmetric (the undirected graph's square matrix must be symmetric), it means that it is a directed graph. Reject this type of input as well.

Step 2:

Check whether the input undirected graph contains a cycle.

This can be accomplished by running a BFS = Breadth First Search (or a DFS = Depth First Search). These algorithms will output a spanning tree of the graph.

- If they also find non-tree edges = cross-edges, that means there is a cycle in the graph. The cycle is given by: the cross-edge plus the paths from the two vertices of the cross-edge up to the first common ancestor (the root vertex in the furthest case). → Reject these types of inputs.
- If these algorithms find only tree-edges, then the graph is a forest (tree, if connected) and has no cycles as a result. → Accept these types of inputs.

Time complexity analysis:

- We know that if the input graph contains n vertices, then the size of the input is $n \times n$.
- We know from the subject **Introduction to the Theory of Computing 1/2** that the BFS algorithm runs in $O(n^2)$ ($O(|V| + |E|)$ in general, but in our case the edges are given in an adjacency matrix format).
- Additionally, the format checking step can also be completed in linear time: counting the characters on the input, then checking $\frac{n^2}{2}$ pairs of cells if they contain the same value.
- This means that our algorithm is linear, so for a size m input, it will run in $O(m)$.

Note:

- If we were to do this on a Turing-machine, while the time complexity will still be polynomial, but it won't be linear. This is due to the fact that the TM is unable to "index the adjacency matrix" in constant time. It needs $O(n^2)$ time to seek a specific location. We say that the polynomial time complexity class is robust, meaning that it does not depend on the "architecture" of the machine we use to implement the algorithm on.

7.2 Session 7, Exercise 2

Exercise

Let L consist of triplets (G, s, t) such that G is a directed graph, s and t are two vertices of G and there exists a path from s to t. Prove hat $L \in P$.

Solution

The details of the proof are similar to that of 7.1.

The main difference is that we are looking for a path between s and t, not a cycle. To find a path, we can use a BFS started from vertex s. If this BFS finds t on a path from s then we accept, otherwise reject.

The less important difference is that the input can now be directed, so we don't check for adjacency matrix symmetry during the first step of the algorithm.

7.3 Session 7, Exercise 3

Exercise

Let language L consist of pairs (n, m) where n and m are positive integers written in binary and the two numbers are coprime (their greatest common divisor is 1). Is it true that $L \in P$?

Solution

The Euclidean algorithm we know from the subject **Introduction to the Theory of Computing 1/2** can find the greatest common divisor of its input n and m integers in $O(min(log_2(n), log_2(m)))$ time.

The input size, if the input is given in binary is $O(log_2(n) + log_2(m))$. This means that the algorithm runs in linear (sublinear) time.

The details of the proof can be done based on 7.1.

7.4 Session 7, Exercise 4

Exercise

Prove that the following two languages belong to NP. Can you prove for any of them that it is in P? Can you prove for any of them that it is in coNP?

a.) Language of undirected graphs that contain a cycle of length at most 100.

b.) Language consisting of pairs (G, k), where G is an undirected graph that has a cycle of length at most k.

Solution

The definition of a language belonging to NP is that there exists a **nondeterministic** Turing-machine for that can **decide** L (accept L and stop on all possible inputs) in polynomial time.

- Polynomial time for a nondeterministic TM means that the length of the longest computational branch is upper bound by a polynomial of the input size.

To prove that a language is in L we could give such a TM as described above, however it would be cumbersome. Instead, we will apply the **Witness Theorem**.

The Witness Theorem in short says that if we can find a polynomial time "verifier" algorithm for a problem, that can check whether a word is in the language, then the language is in NP.

The connection to the nondeterministic TM is as follows:

- If we can check a verifying witness in polynomial time, then we can construct a nondeterministic TM, to first "nondeterministically" generate all possible witnesses (in parallel computational branches), then use the polynomial algorithm (=deterministic TM) to check it. This in total will run in polynomial time as well.
- If there exists a nondeterministic TM that can decide L in polynomial time, then a good witness for this problem is the description of how to find the accepting branch in the computational tree for any given input. (For example, in every branching step, the index of the chosen branch.) If we are given a witness, we can escape the multiple computational branches and simply navigate into the accepting state, which turns our computation deterministic.

In practice the witness theorem means that a problem is in NP, if we are given a possible solution, we can at least verify its correctness efficiently. The following Youtube video explains this nicely: P vs. NP and the Computational Complexity Zoo by hackerdashery.

How to construct a proof using the Witness Theorem:

We need to give the following items:

- Witness:
  - It must exist for all accepted words.
  - It must NOT exist for any of the rejected words.
  - Length must be polynomial (relative to the corresponding input size).
- Witness checking algorithm:
  - It must be able to verify the witness for a given input.
  - Runtime must be polynomial (relative to the input + witness size together).

For task a):

- Witness:
  - A description of a cycle in the graph: a list of nodes, in the same order as they are in the cycle: $\{v_1, v_2, \ldots, v_m\}$.
  - If the graph is accepted, then there exists a cycle in it (and the witness checking algorithm will be able to verify, that it is indeed a cycle in the graph).
  - If the graph is rejected, then there exists no cycle in it: no witness exists for these inputs (and the witness checking algorithm will figure out if we are trying to fool it by giving it a faulty witness - something that is not actually a cycle!).

  - Since the maximum length of the cycle is 100, and a single vertex's index can be described using $O(\log_2 n)$ bits, where n is the total number of vertexes, then the witness' length is $100 * O(\log_2 n)$, or simply $O(\log_2 n)$. The input's length is $O(n^2)$, so if we take $k = n^2$, then $\sqrt{k} = n$ and $\log_2 \sqrt{k} = \log_2 n$. So the witness' length relative to the input size is $O(\log_2 \sqrt{k})$, better than polynomial.

- Witness checking algorithm:
  - Count that the number of vertices does not exceed 100, and check that all of them exist in the graph (their index is not too big). This is a constant step, since the moment we counted the 101th vertex, we can reject the witness.
  - Look up the following cells in the adjacency matrix: $\{v_1, v_2\}$, $\{v_2, v_3\}$, $\{v_3, v_4\}$, ..., $\{v_{m-1}, v_m\}$ and finally $\{v_m, v_1\}$. (Don't forget the last edge closing the cycle!)
  - If all of these are edges of the graph (contain a 1 in every cell), then this is a cycle.
  - This step can be done in $O(m)$, where m is the number of vertices in the cycle, which can not exceed 100. This is a constant step.
  - The witness checking algorithm runs in constant time!

For task b):

Similar to task a), except that the cycle's length is replaced by k instead of 100. Since k can be upper bound by n, the corresponding space (witness size) and time (witness checking algorithm runtime) complexity calculations will still result in a polynomial Big-O.

Are these in P?

Both of these languages are in fact in P. To show this, we give a polynomial time algorithm that can find the shortest cycle in a graph. Then, if we check the size of this cycle and it is within limits (either 100 or k), then we can accept, and if it is outside of the limits, since this is the shortest possible cycle in the graph we reject.

We can use the BFS algorithm to find cycles in the graph. When the BFS is started from a given vertex, it will find all cycles that contain that specific vertex. To find all possible cycles we start a BFS from all vertexes one-by-one. When a non-tree edge is found, we can qucikly calculate the corresponding cycle's length, by tracing back to the first common parent. Then we keep track of the currently found minimum length cycle.

The runtime of this algorithm:

- The BFS will be executed n times, once for all vertices in the graph.
- Then during a single BFS, when a non-tree edge is found, we trace our steps back. A slightly large upper limit for the number of possible non-tree edges is n^2, and the number of steps required to trace back is also n^2, since in both cases we will touch all edges at most once.
- In total this is still $O(n^5)$, for an input of size $O(n^2)$ which is a very generous upper estimation, but still polynomial.

Are these in coNP?

$coNP$ means that the complementer of the language is in NP. Or, that an efficient verifier = witness / witness checking algorithm can be found for the 'NO' answer.

Since we know that $P \subseteq coNP$, and we have just shown that these languages are in P this also proves that these languages are also in $coNP$.

7.5 Session 7, Exercise 5

Exercise

Prove that

a) Language $MAXCLIQUE = \{(G, k) : \text{undirected graph G has a clique of size k}\}$ is in NP.

b) Language $5CLIQUE = \{G : \text{undirected graph G has a clique of size 5}\}$ is

- in NP,
- in coNP,
- in P

Solution

a) Using the Witness Theorem:

- Witness: a description of a k-clique in the graph.
  - If the graph is in the language, the witness exists.
  - If the graph is not in the language, the witness doesn't exist.
  - The size of a description of a clique in the graph will be a list of vertex indexes that constitute the clique. This is of size $O(n \log_2 n)$ (at most n vertex indexes in binary form), which is polyinomial relative to the $O(n^2)$ input size.
- Witness checking algorithm:
  - Check that the witness consits exactly k vertices, all of them exist in the graph (their index is not too big), and check for all $\binom{n}{2}$ pairs of vertices in the witness that the edge exists between them.
  - This in total is $O(n + n^2) = O(n^2)$ time, which is polynomial to the input + witness size.

b)

- in NP: Same as above, for $k = 5$.
- in P: Yes, because we can check all $\binom{n}{5} = O(n^5)$ possible 5 vertices if they form a clique.
- in coNP: Yes, because it is in P also.

7.6 Session 7, Exercise 6

The 3 theorems used in this exercise are all studied in **Introduction to the Theory of Computing 1/2**.

Exercise

Prove that the following languages belong to coNP.

a) Language of those bipartite graphs that have complete (perfect) matching.

b) Language of those garphs that have complete (perfect) matching.

c) Language of planar graphs.

d) Language of graphs that whichever way their edges are colored using 2 colors, there is a monochromatic triangle.

Solution

a) Using the Witness Theorem (witness for no answer).

According to Hall's marriage theorem there exists a complete matching in a bipartite graph **if and only if** there is no set of X vertices coming from one side of the bipartite graph for which $|X| > |N(X)|$. (The number of their neighbours is less than their number.)

- Witness: A set of X vertices, breaking Hall's condition.
  - Since Hall's theorem states that the existence of such a set is equivalent to the bipartite graph having no complete matching, this means that the witness exists for all rejected words, but no accepted words.
  - The size of the witness is at most $O(n \log_2 n)$.
- Witness checking algorithm:
  - Check that all vertices exist, are on the same side of the bipartite graph and count their neigbhours.
  - It can be shown that this runs in polynomial time.

b) Using the Witness Theorem: Similar to the previous one, using Tutte's Theorem.

Tutte's Theorem: A (generic) graph has a perfect matching if and only if for every subset U its vertices, removing that U from the graph: $G \setminus U$ falls apart to having at most $|U|$ odd components.

Then, the witness is such an U, etc.

c) Using the Witness Theorem: Similar to the previous one, using Kuratowski's Theorem.

Kruskal's Theorem: A graph is planar if and only if it does not contain a subgraph that is a subdivision of K_5 or $K_{3,3}$.

Then, the witness is such a subgraph, etc.

7.7 Session 7, Exercise 7

Exercise

Solution

7.8 Session 7, Exercise 8

Exercise

Solution

7.9 Session 7, Exercise 9

Exercise

Solution

8 April 13th (Session 8): NP-completeness

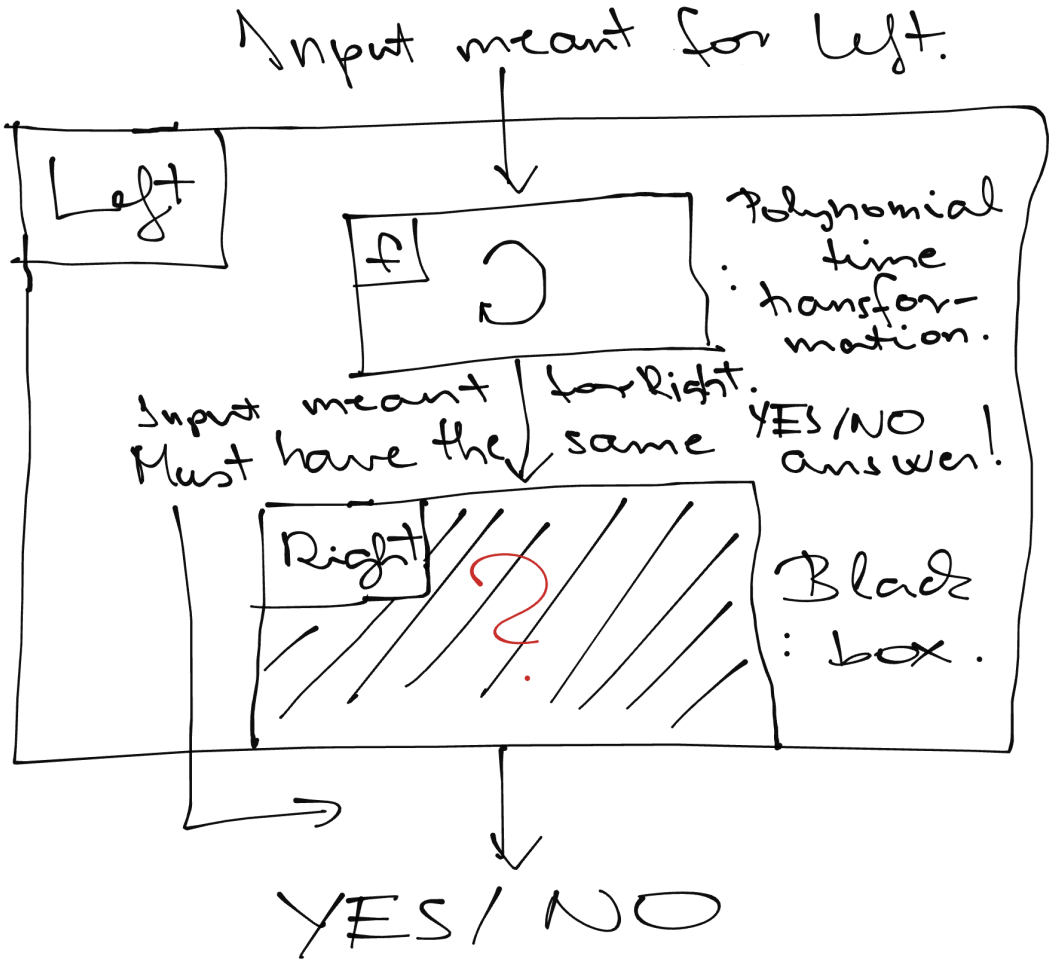
8.1 What is a Karp-reduction?

When we say that a Karp-reduction $Left \prec Right$ exists, it means all of these things:

(Where $Left$ and $Right$ are both some decision problems. Traditionally we would use letters, like $X \prec Y$, or the name of the actual problems, but I replaced them with $Left$, meaning left side and $Right$ right side, it makes the following stuff more readable.)

- $Left$ can be solved using $Right$.
- The hardness of $Left$ cannot be more, than the hardness of $Right$, since $Left$ can be solved using $Right$.
- The way we can solve $Left$ with using $Right$ is very specific:
  - We take the input meant to be given to $Left$.
  - We run in through a **polynomial time transformation**, which we note by function f.
  - Then we give the transformed input to $Right$.
  - Then whatever $Right$ outputs, we must respond with the same thing. (This can be tricky to achieve!)
- The polynomial-time transformation is usually noted by f. It must be created in a way, so that the YES / NO answers are the same: for any input $Left$ answers with a YES, the transformed input must result in a YES answer from $Right$ as well, and same for the answer NO.

You can think about it like this:



Notice, how $Right$ is attached to the bottom of $Left$, this is meant to show, that we cannot modify that answer, we must return it immediately. (By the way, other polynomial-time reductions exist, which allow you to run the $Right$ algorithm multiple times, or modify the output, but we do not study these.)

It can be difficult to remember the order of these (which one is used to solve which one?), for that I suggest remembering the phrase: "When there is nothing (to solve) $Left$, one must do the (algorithm to solve the) $Right$ thing!"

The reason why they invented polynomial-time reductions, like the Karp-reduction is because groups of researchers have been struggling for a very long time to come up with efficient solutions for a number of very important problems (these are the NP-complete problems we are studying). They have started to notice, that these problems did not differ from one another by that much, in fact, if someone came up with a fast solution to one of these problems, the others could create a solution for theirs by running a fast tranformation of their input and then using the solution for the other. Essentially, we cannot (yet? or not at all?) solve these problems, but we are prepared for a solution to one of them pop up, since we have done all the work to solve everything else from that. We are just waiting for that one solution...

This video talks about this, I highly recommend checking it out: P vs. NP and the Computational Complexity Zoo from hackerdashery.

Whenever you are given a task to give a Karp-reduction between two problems, I see a lot of confusion on what the task is asking for. I would like to use an example and answer common questions on it: let's say you need to give a Karp-reduction $s-t-HAMPATH \prec HAM$.

- Which one is being used to solve which one?

  $\rightarrow$ The left is solved using the right one, so in this case $s-t-HAMPATH$ is solved using an imaginary preexisting solution to HAM.

- Did someone solve HAM but struggles to solve $s-t-HAMPATH$? These are almost the same, we should just look at the code and fix it up!

  $\rightarrow$ We do not know anything about the solution to HAM, we cannot look at the code. This is because nobody actually solved HAM yet. We can imagine if someone solved HAM, then we could probably modify the code and solve $s-t-HAMPATH$ too with it, but without knowing what the solution is, all we can do is treat it as a blackbox.

- Why can't we change the result? For example, what if we wanted to return YES any time $Right$ says NO and vice versa? Why can't we run the algorithm for $Right$ more than once, for differently transformed inputs and collect and make sense of the result?

  $\rightarrow$ This is specific to this type of polynomial-time reduction. Other's exist for which the rules are different, but we don't study those. I've talked about this on the practice sessions: limitations like this usually exist, because they make it easier to give proofs. The less stuff can happen, the easier it is to argue. Then, in practice we usually use a less-rule-limited version of stuff, with proofs that the less-rule-limited version can't actually do more than the rule-limited version.

- How do we create the Karp-reduction?

  $\rightarrow$ You give the f polynomial time transformation:

  - First, you explain what the inputs and outputs are for HAM and $s-t-HAMPATH$.
    - HAM's input is an undirected graph, for which it checks whether it contains a Hamiltonian cycle. (If it does, it returns YES, if it does not, it returns NO.)
    - $s-t-HAMPATH$'s input is an undirected graph, and two vertices of that graph, one is noted by s, the other is noted by t. It checks if there exists a Hamiltonian path between vertices s and t in the graph (returns YES if exists, returns NO, if does not).
    - (Hamiltonian *something* means that the *something* contains all vertices of the graph.)
  - Then, you explain what the input transformation is.
    - It is very important here, that it is very likely not just a copy and paste of whatever input matches and then fill in the rest!
    - So in this case, we will have to modify the G graph, since just giving it straight to a solver that's trying to find a Hamiltonian cycle in it, while we only need a Hamiltonian path is not going to work.
    - If the graph has a Hamiltonian cycle, then it also has a Hamiltonian path, however that path might not be so that s and t are neighbours on it (so the $s-t-$Hamiltonian path does not exist, that part of the path does not contain all vertices), and it could be that the graph does not have a Hamiltonian cycle, but it does have the $s-t$-Hamiltonian path (a single $s-t$ edge is missing to complete the cycle).
    - In this case, the transformation is the following: add one more vertex to the graph, note it with v, and connect it to the vertices s and t.
  - Then, you explain why any input for $s-t-Hampath$ that is a YES answer, if we transform the input in the specified way, the answer will be YES for HAM as well.

    * If there was an $s-t$-Hamiltonian path in G, then the edges of the Hamiltonian cycle in the transformed G' graph will be the edges of the $s-t$-Hamiltonian path + the $s-v$ and $v-t$ edges we added.
    * This contains all vertices from G, plus the one additional vertex v as well, making it a Hamiltonian cycle.
  - Then, you explain the same for he NO answers (usually this will be an indirect proof):
    * If there is no $s-t$-Hamiltonian path in G, then there cannot be a Hamiltonian cycle in G'. Why? Let's argue using indirect proof: let's indirectly assume that there is no $s-t$-Hamiltonian path in G, but there still exists a Hamiltonian cycle in G'. Then, if we remove the vertex v from that cycle, we will be left with a path that contains all other vertices of the G', so all original vertices of G, plus the endpoints of that path will be s and t, since v was only connected to those two, so they must also be neighbours on the Hamiltonian cycle. This is a contradiction, since we assumed no $s-t$-Hamiltonian path exists, so this proves our original statement.
  - Finally, you explain why the transformation can be done in polynomial time (don't forget this step, if the transformation is too slow, then we can't use it in a fast solution for $s-t-HAMPATH$).
    * Graphs are given by their adjacency matrices.
    * To add one more vertex to the graph, we append one row and one column to this matrix (copy the entire thing into a matrix that has one extra column and row).
    * We add 1's to this row at the indexes corresponding to s and t. If the adjacency matrix is of size n^2, then this can be done in $O(n^2)$ time, which is linear relative to n^2.

I also like to imagine Karp-reduction with the following fictive code:

```cpp
#include <vector>
using namespace std;
using adj = vector<vector<int>>; // Declare type for adjacency matrix.

// Karp came up with this idea...
bool s_t_hampath_solver(adj G, int s, int t) {
    vector<vector<int>> G_prime = f(G, s, t);
    return ham_solver(G_prime);
}

// ..and asked you to implement this part:
vector<vector<int>> f(adj G, int s, int t) { // Totally polynomial!
    int size = G.size();
    vector<vector<int>> G_prime(size+1, size+1);
    for(int i=0; i<size; ++i) {
        for(int j=0; j<size; ++j) {
            G_prime[i][j] = G[i][j];
        }
    }
    int v = size; // Indexing starts with 0.
    G_prime[v][s] = 1;
    G_prime[v][t] = 1;
    G_prime[s][v] = 1;
    G_prime[s][t] = 1;
    return G_prime;
}

bool ham_solver(adj G) {
    // TODO
}
```

8.2 Session 8, Exercise 1

Exercise

Prove that $2-COLOR \prec 3-COLOR$ and that $3-COLOR \prec 100-COLOR$. What do these Karp-reductions say about the complexities of the three problems?

Solution

The $n-COLOR$ problem, for any n is the following: For a given (undirected) G graph, can the graph's vertices be correctly colored (=no edge-connected vertices get the same color) using n colors?

Let's do $2-COLOR \prec 3-COLOR$ first.

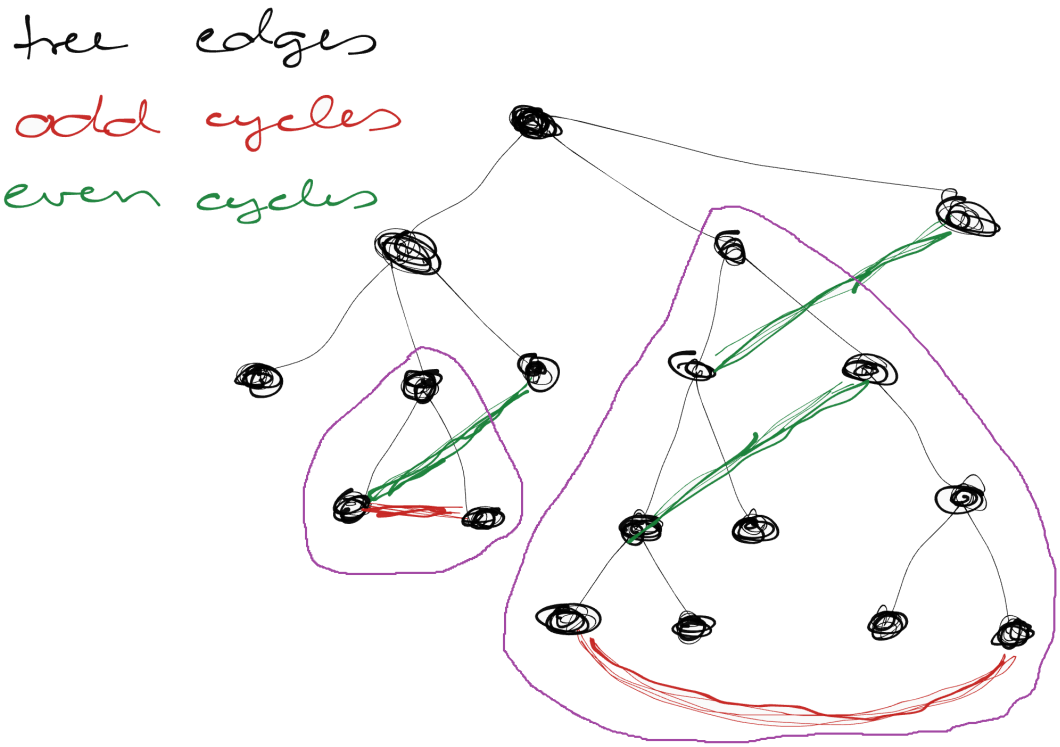
First solution (existence of $NP \prec NP$-hard)

This solution is pecific to the fact that the right side is NP-hard and the left side is NP, and that you only need to prove the existence of the Karp-reduction.

$2-COLOR$ is in P. To show that something is in P we give a polynomial algorithm that can solve the problem.

The set of graphs that are 2 colorable is called bipartite graphs, since these are the ones where the vertices can be grouped to set A and set B so that the edges are only running between sets. x To check if a graph is bipartite, we can run the BFS (Breadth-first search) algorithm. (You have learned this in Introduction to the Theory of Computing):

- The algorithm returns a spanning tree (or forest, if the graph is not connected) of the graph, where the vertices are organized into layers of the tree (in the case of forest do the following for all trees = all connected parts of the graph separately).
- If we look at all of the edges from the graph (not just the edges that are part of the spanning tree), they can only run between consecutive layers, or inside the layers.
- If there was an edge, that say went from layer i, to not layer i or $i+1$, but $i+2$, $i+3$, etc, then the child in that edge should have been expanded on layer $i+1$, since BFS expands all unexplored connected vertices of a vertex on the next layer!
- A necessary and sufficient condition for a graph being bipartite is that it does not contain any odd cycles.
- How would an odd cycle look in this graph?



- The edges of the tree are presented in black, while the other edges are either green (running between two layers), or red (running inside a layer).
- The problem edges are the red ones: if we look at their endpoints and trace them back up to their first common ancestor, we got ourselves an odd cycle. I have circled these with purple.
- The green edges will cause even cycles, which are bipartite (check this yourself!), so they are fine.
- So we have found that if the resulting tree has any red edge = any edge running inside a layer, then its not bipartite.
- BFS can be run in polynomial time and then we can check the endpoints of all of the edges, which layer they belong to in polynomial time as well.

So $2-COLOR$ is in P. However, starting from $3-COLOR$, up to any $n-COLOR$ where $3 \leq n$, the problems are NP-complete. We have seen the NP-completeness of $3-COLOR$ on the lectures. When we need to prove that a Karp-reduction exists (as opposed to GIVE a Karp-reduction), it is not always necessary to actually give it. So the quickest way to prove that this Karp-reduction exists is the following:

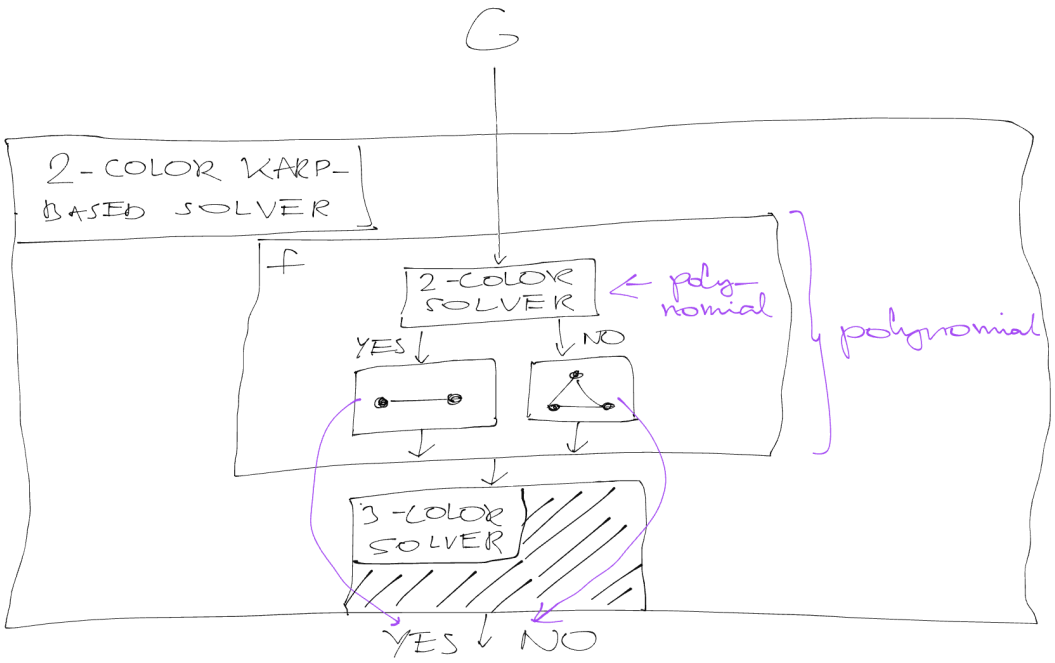
- $3-COLOR \in NP$-complete, which means it is both true that $3-COLOR \in NP$, and $3-COLOR \in NP$-hard.
- Also $2-COLOR \in P \subseteq NP$.
- The definition of NP-hard is that a Karp-reduction exists from all languages that are in NP.
- So $2-COLOR \in NP$ and $3-COLOR \in NP$-hard means the $2-COLOR \prec 3-COLOR$ Karp reduction exists by definition.

Second solution, specific to the fact that the left side is in P.

We borrow the fact that $2-COLOR$ is in P, from the first solution, then we give a Karp-reduction using the fact that it is in P:

- 2-COLOR is in P, so there exists a polynomial time algorithm to solve it.
- Let's run this algorithm as the first step of our Karp-reduction! This is allowed, because we can run any polynomial transformation algorithm in a Karp-reduction.

- Now we know what the solution is, but the problem is that we can't directly return that. We must go through 3-COLOR, because the output from $3-COLOR$ is what we must use.
- Well, since we already know the solution, we can just make up two small graphs, one for which $3-COLOR$ answers YES and another for which it answers NO. Then we can use those instead of our YES / NO answers to make $3-COLOR$ output exactly what we need. For example a single edge, for the YES answer and a cycle of 3 vertices for the NO answer is good.
- Let's see this on the same Karp diagram:



Third solution, now specific to the coloring problem. :)

- Let's add one more vertex to the graph and connect that to every other vertex!
- This one vertex has to get a unique color, since it is connected to everything. We just wasted one of the colors from $3-COLOR$.
- Now $3-COLOR$ must give up one of its colors for the new vertex and try to color everything else using 2 colors which is just what we wanted.
- This transformation is polynomial, its adding a new row and column to the adjacency matrix and filling them with 1's, except for the bottom right corner, which is zero.
- If there is a 2-coloring in G, then there will be a 3-coloring in G', since the one extra vertex can get the third color.
- If there is no 2-coloring in G, then there cant be any 3-coloring of G', since if there were a correct 3-coloring in G', then the new vertex must get a unique color, since it is connected to everything, so the other vertices can only get 2 other colors, so this would be a correct 2-coloring of G, which is a contradiction.

The solution to $3-COLOR \prec 100-COLOR$ is similar to the previous ones:

- First solution: $3-COLOR$ and 100-COLOR are both NP-complete, so they are both in NP and in NP-hard. Use the NP fact for $3-COLOR$ and the NP-hard fact for $100-COLOR$, then it's the same argument as in the previous part.
- Second solution: We can't do this, since $3-COLOR$ is not in P. :(
- Third solution: We need to waste 97 colors, this can be done by adding a fully connected graph of 97 vertices to G, which is traditionally denoted by K_{97} and also connecting all of K_{97}'s vertices to all of G's vertices. (So full bipartite graph between these two.) Then, the same argument works as previously.

8.3 Session 8, Exercise 2

Exercise

Prove that if $X \prec Y$, then $\overline{X} \prec \overline{Y}$.

Solution

- If $X \prec Y$, then a polynomial time computabe f function exists, that will convert all inputs of X into inputs of Y, such that if the answer is YES for a particular input word w in X, then the answer will be YES for $f(w)$ if checked by Y and if the answer is no for w in X, then the answer is NO for $f(w)$ in Y.
  - This, but with more math: $\exists f : \Sigma^* \to \Sigma^*$, so that f is polynomial time computable and $w \in X \Leftrightarrow f(w) \in Y$.
- $\overline{X}$ is the complementer of the X language, so for anything that X returns a YES it will return a NO and for anything that returns a NO in X it will return a YES. Similarly for $\overline{Y}$ and Y.
- This means that the **same f function** can be used for the $\overline{X} \prec \overline{Y}$ Karp-reduction, since it transforms inputs so the original-transformed pair returns YES or NO at the same time. The answer will be inverted in both $\overline{X}$ and $\overline{Y}$, so they still result in the same answer anyways!

8.4 Session 8, Exercise 3

Exercise

Let language L consist of simple graphs G, such that a proper coloring of its vertices uses at least 4 colors. Show that the $PRIME \prec L$ Karp-reduction exists.

Solution

- What does it mean that a proper coloring of a graph's vertices uses at least 4 colors? $\rightarrow$ It means that the graph is not 3-colorable, or not in $3-COLOR$.
- So $L = \overline{3-COLOR}$.
- $PRIME$ is the language of primes. The complementer of this language is $COMPOSITE$, so $PRIME = \overline{COMPOSITE}$.
- So the question is actuall, does the $\overline{COMPOSITE} \prec \overline{3-COLOR}$ Karp-reduction exist?
- In Session 8, Exercise 2 we have shown that a Karp-reduction exists between two languages if and only if the Karp-reduction exists between the complementer languages as well.
- So $\overline{COMPOSITE} \prec \overline{3-COLOR}$ exists if and only if $COMPOSITE \prec 3-COLOR$.
- Now we can prove that $COMPOSITE \in NP$, using the witness theorem: a goot witness for a composite number is a real divisor (e.g. a divisor other than the number itself, or the number 1). The witness checking algorithm is division. The witness size is polynomial (a smaller number, so at most as many bits as the larger one), and the witness checking algorithm runs in polynomial (the division algorithm we have learned in primary school for example).
- We know that $3-COLOR$ is NP-complete, which means it's both in NP and NP-hard. Let's use the fact that it's NP-hard.
- The situation is the same as in Session 8, Exercise 1, the left side is in NP, the right side is NP-hard, the definition of NP-hard states that a Karp-reduction must exist from all languages in NP onto the NP-hard language, so this one should too.

Side note:

- You may have noticed, that we have slightly rushed over the fact, that a number that is not prime may not necessarily be composite either, for example the number 0 and the number 1 are neither.
- This is not that important, but for the sake of completeness, we can define the encoding of the numbers so that 0 and 1 get no codes.
- For example, if we used a binary alphabet $\Sigma = \{0, 1\}$, we could say that the code 0 represents the number 2, code 1 represents number 3, and so on.

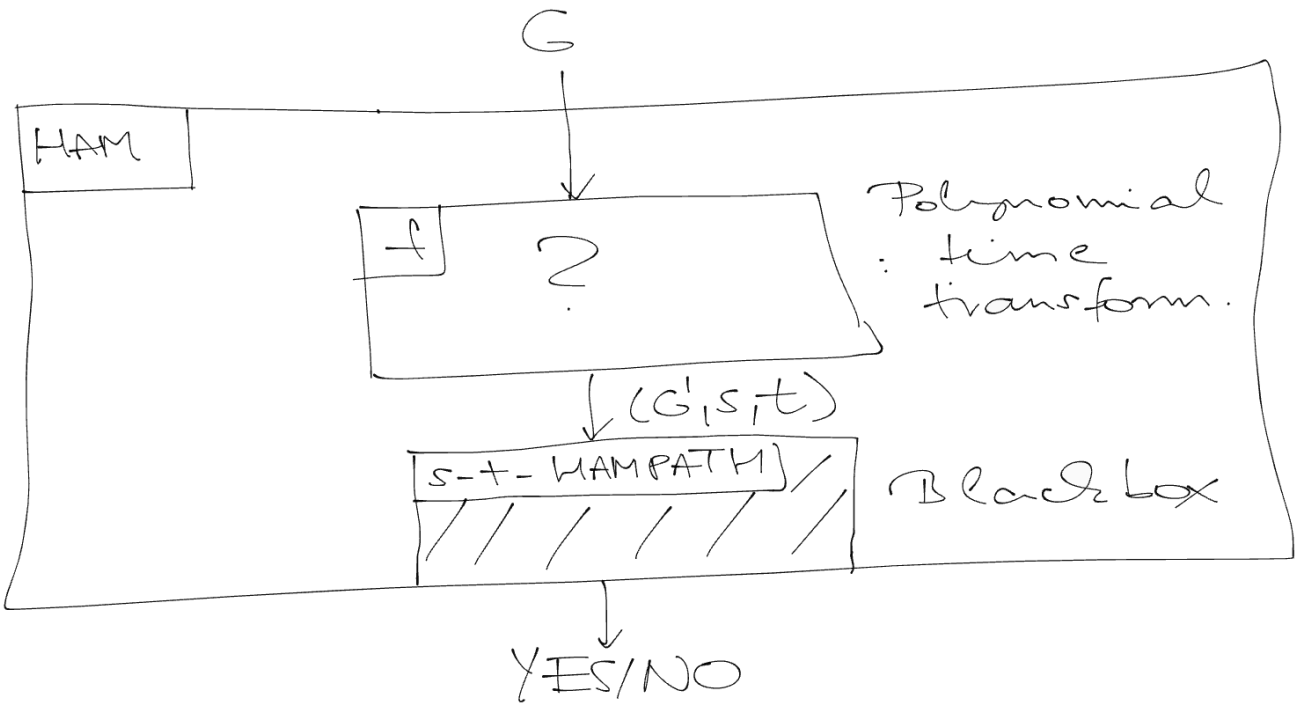
8.5 Session 8, Exercise 4

Exercise

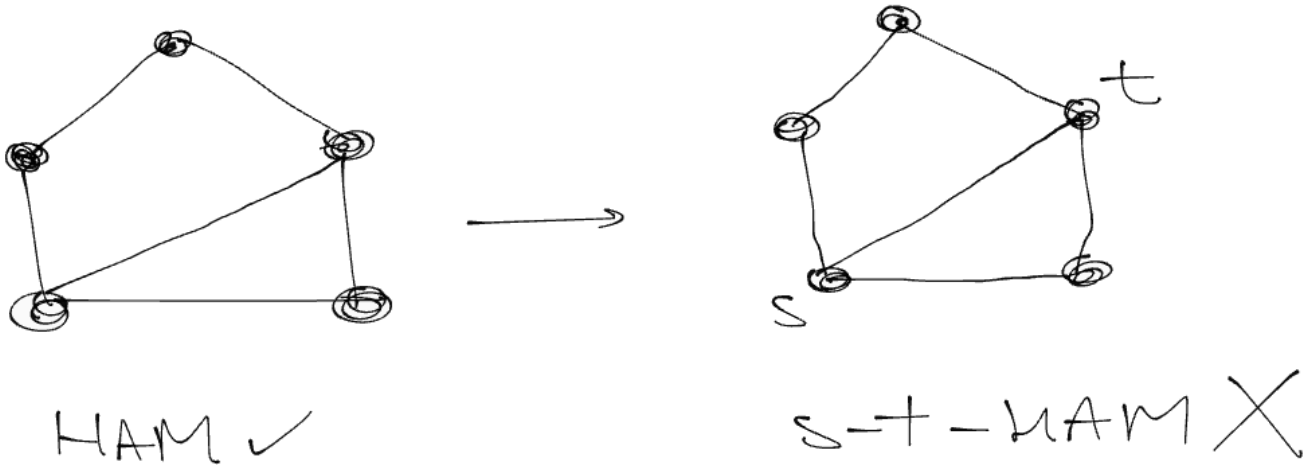
Give a $HAM \prec s-t-HAMPATH$ Karp-reduction.

Solution

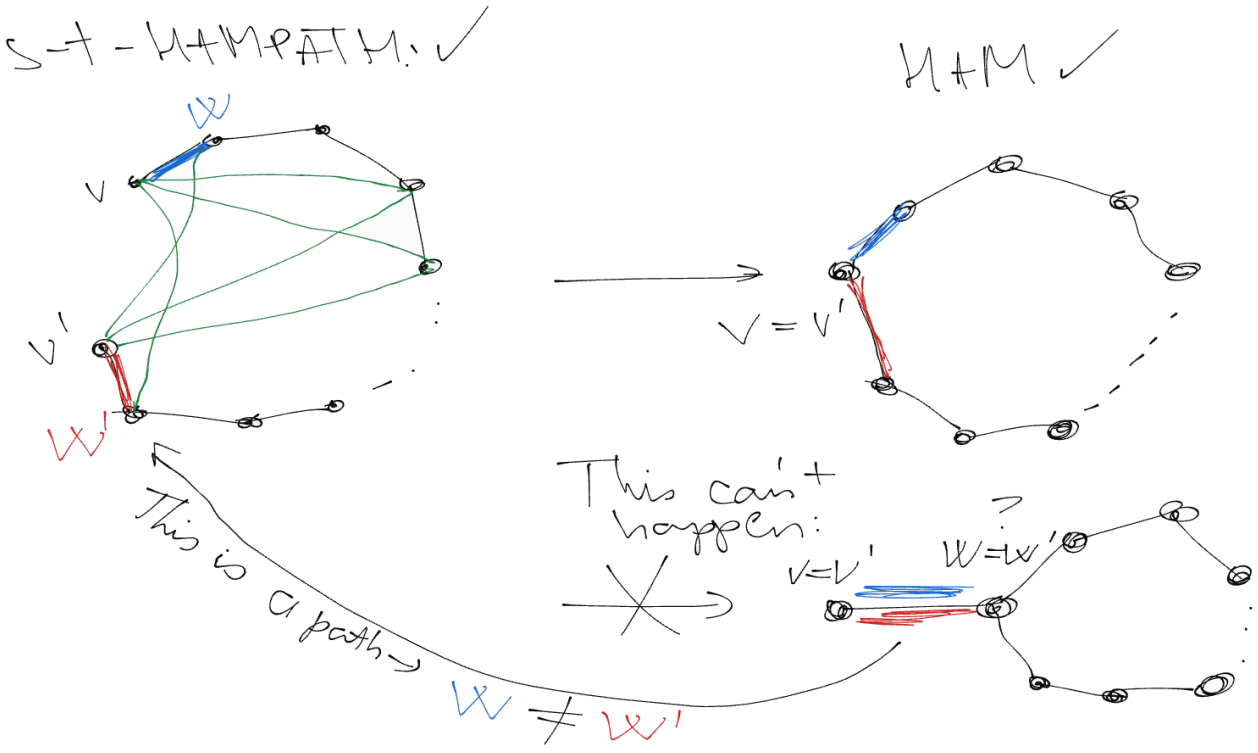
So we need to solve HAM, using $s-t-HAPATH$, e.g.:



- The HAM language gets a G graph as an input and we need to transform this G graph in such a way, that the G' transformed graph contains an $s-t-HAMPATH$ **exactly** when the G contained a Hamiltonian-cycle.
- The first idea that could come to mind is to just leave G as is, and choose two neighbouring vertices in it, assign s and t to those and give these to $s-t-HAMPATH$.
- It is true, that if we find a Hamiltonian-path between these two vertices, then also using the fact that there is an edge between them, that edge completes the $s-t-HAMPATH$ into a Hamiltonian-cycle.
- However, if we do not find a Hamiltonian-path between the two vertices, then the Hamiltonian-cycle could still exist in the original graph, we might have just made an unlucky choice for s and t. Consider the following example (the original G graph is on the left, and we choose the vertices of the edge that is not part of the Hamiltonian-cycle):

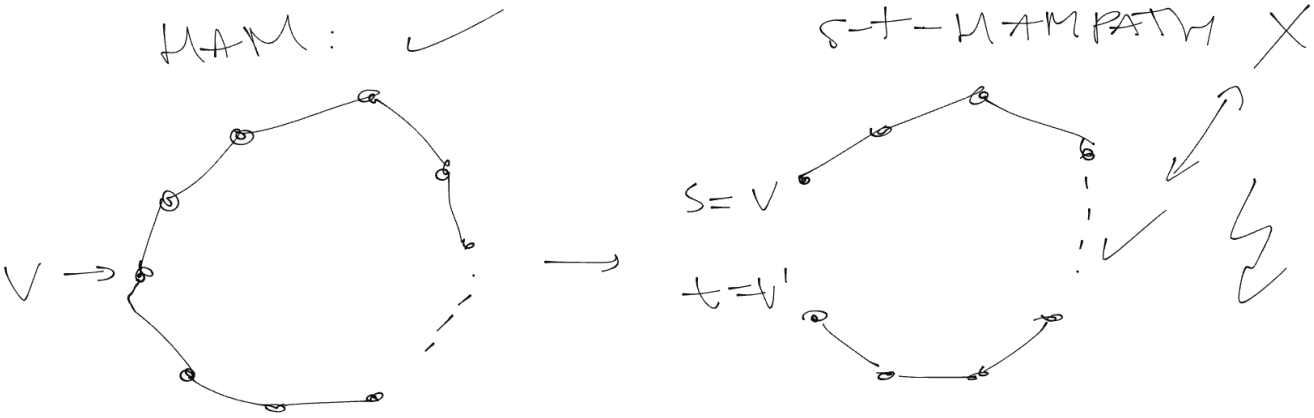


- The left graph has a Hamiltonian-cycle, but the right graph doesn't have an $s-t-$Hamiltonian-path.
- The issue here is that we tried to use different vertices and the edge between those vertices may or may not be part of the Hamiltonian-cycle.
- To fix, we are going to choose the same vertex twice! Well, we can't really do that, since that would contradict the definition of a path (it has to end in different vertices), but we are going to fix this by simply creating a copy of one of the vertices.
- So the transformation is as follows:
  - Select one of the vertices of G, let's call this v.
  - Duplicate it (add edges to the same neighbours as v has) and call it v'.
  - This resulting graph is G', and $s = v$, $t = v'$.
  - Give this to $s-t-HAMPATH$ and see what it returns.
- If $s-t-HAMPATH$ returned a YES, this means that there is a Hamiltonian path between v and v'. What does this mean in the original graph?



- Then we can reconstruct a Hamiltonian-cycle in the original graph, as we can see on the image above.

- It could be an issue if the blue and the red edge were the same, because we copied v to v', so we might have accidentally used the same edge, however that cannot happen, because that would require w and w' to be the same vertex too and they cannot be, since a path cannot contain duplicate vertices.
- If $s-t-HAMPATH$ returned a NO, then this means that there cannot be a Hamiltonian-cycle in the original graph. This can be show using proof by contradiction.
- Let's indirectly state that $s-t-HAMPATH$ returns a NO, but there is a Hamiltonian-cycle in the original graph. This Hamiltonian cycle contains the v vertex we duplicated, since it contains all vertices of the graph.
- When we duplicated the v to v', the Hamiltonian-cycle was broken up into a path that contains all vertices and starts in v and ends in v'. Which means this is a $v-v'$-Hamiltonian-path, or an $s-t$-Hamiltonian path, so $s-t-HAMPATH$ could not have returned a NO answer. See below:



- This is a contradiction, so we have shown that the original statement was true, if $s-t-HAMPATH$ returned a NO, then this means that HAM will return a NO too, for G', s, t.

8.6 Session 8, Exercise 5

Exercise

It is known that $L_1 \prec L_2$ and the complement of L_2 can be Karp-reduced to language $PARTITION$. Prove that $L_1 \in coNP$.

Solution

- As we have seen in Session 8, Exercise 2, $L_1 \prec L_2$ means $\overline{L_1} \prec \overline{L_2}$ is also true.
- We know that Karp-reduction is transitive, so $\overline{L_1} \prec \overline{L_2} \prec PARTITION$, so this also means $\overline{L_1} \prec PARTITION$.
- $PARTITION$ is $NP - complete$, which means it is both in NP and in $NP - hard$.
- It might be tempting to use the fact that $PARTITION$ is $NP - hard$, so all languages in NP must have a Karp-reduction onto it, however this does not necessarily mean that if a language has a Karp-reduction onto it, then it must be in NP. (Well, it does, but we need to show it, and here is how:)
- We will use the fact that $PARTITION$ is in NP! This means that there exists a witness, that is polynomial in size and a witness checking algorithm that runs in polynomial for every single word that is in the language and none that is outside of the language.
- Let's take an input word w for $\overline{L_1}$ and let's use the f polynomial time transformation function on it that comes from the $\overline{L_1} \prec PARTITION$ Karp-reduction!
- If $w \in \overline{L1}$ if and only if $f(w) \in PARTITION$. Since f can be computed in polynomial time, it also means that the size of $f(w)$ is polynomial relative to w.
- We know that $f(w) \in PARTITION$ if and only if a witness exists for $f(w)$, that is polynomial in size relative to $f(w)$, which is polynomial in size relative to w. A polynomial's polynomial is also a polynomial, so the witness' size is polynomial relative to w too. The witness checking algorithm runs in polynomial time, for similar reasons, relative to the size of w also.
- This means that a witness for $w \in \overline{L1}$ is going to be the same witness that is for $f(w) \in PARTITION$ and the witness checking algorithm is exactly the same.
- This shows that $\overline{L1} \in NP$ (using the Witness Theorem).
- And the definition of $coNP$ is: the complementer of the languages in NP. So $\overline{\overline{L1}} \in coNP \rightarrow L_1 \in coNP$.

8.7 Session 8, Exercise 6

Exercise

Prove that the following Karp-reductions exist:

a) $SUBSETSUM \prec HAM$

b) $CONNECTED \prec 3 - SAT$

c) $CONNECTED \prec BIPARTITE$

where $CONNECTED$ is the language of connected graphs, while $BIPARTITE$ is the language of bipartite graphs.

Solution

Quick rundown here of what is what:

- $SUBSETSUM$ is NP-complete, we learned this
- HAM is NP-complete, we learned this.
- $CONNECTED$ is in P, because we run a BFS on a graph from any vertex to see if it's connected, if it reaches all vertices, then it is. BFS is a polynomial time algorithm.
- $3 - SAT$ is NP-complete, we learned this.
- $BIPARTITE$, we have shown in Session 8, Exercise 1 that it is also in P.

a)

- Is an NP-complete problem Karp-reducible to another NP-complete problem? Yes.
- NP-complete means it is both in NP and NP-hard.
- Let's use the NP fact for the left side, the NP-hard fact for the right side.
- Is a language that is in NP Karp-reducible to another that is NP-hard? Yes. This is the definition of NP-hardness, all languages in NP can be Karp-reduced onto it.

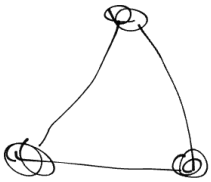
b)

- Is a language in P Karp-reducible to a language that is NP-complete?
- P is a subset of NP, so the question is exactly the same as in a): left side is in NP, right side is NP-hard, so the Karp-reduction must exist. (See Session 8, Exercise 1, second solution for a different approach, for when you actually have to give the Karp-reduction.)

c)

- Is a language in P Karp-reducible to another language in P?
- Yes, a language in P is basically Karp-reducible to anything, for similar reasons as in Session 8, Exercise 1, second solution:
- The Karp-reduction transformation function allows us to run any polynomial algorithm. Since the language on the left side is in P, we can start by solving it in polynomial time.
- Then When we know the solution, we are faced with one more difficulty: we have to go through the solver algorithm for the language on the right side. We can't directly output the YES or NO answer.
- So we will come up with a specific input that we know will make the solver say YES and another that will make the solver say NO.
- To make the $BIPARTITE$ solver say yes, we input this graph:

- To make the $BIPARTITE$ solver say no, we input this graph:

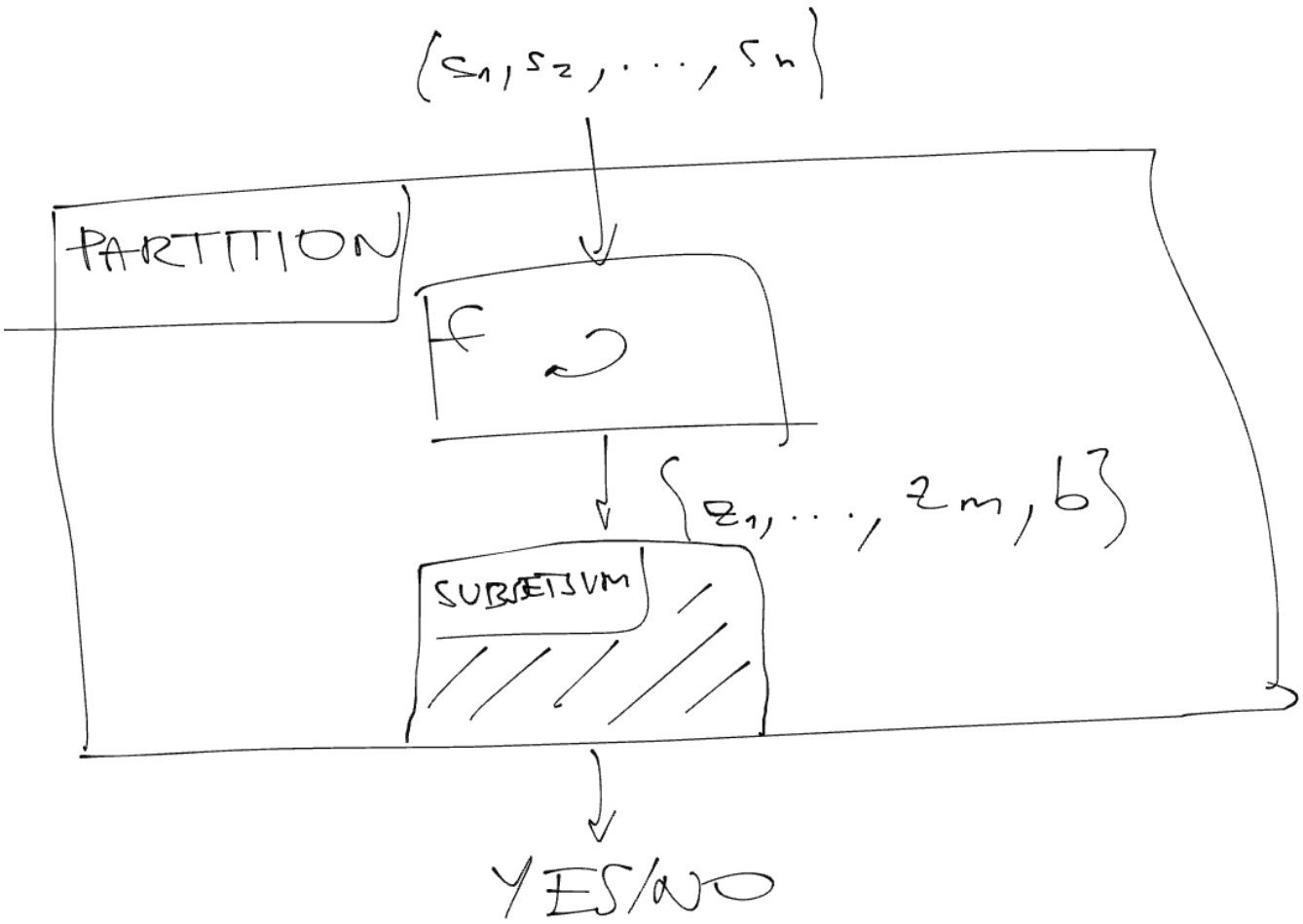


8.8 Session 8, Exercise 7

Exercise

Give a Karp-reduction from $PARTITION$ to $SUBSETSUM$.

Solution



- So we need to solve $PARTITION$ uing $SUBSETSUM$.
- $PARTITION$ tells us whether or not we separtate our numbers into two sets for which their sum is equal.
- $SUBSETSUM$ tells us whether we can find a subset of numbers, for which their sum is a specific number.
- In $PARTITION$ we can quickly figure out what the sum of the two sets should be, by just summing up all the input numbers and dividing them by 2.
- If the sum is an odd number, then the answer is immediately no. There is no way to divide those numbers equally, since if the sum of the two equal parts would be some number $a = a$, then the total sum should be $2a$, which is even.
- So if the sum is odd, we need to make $SUBSETSUM$ return a NO answer, this can be done by creating an artificial input: $\{2, 2\}$ and $b = 1$.
- There is no way to select a subset from $\{2, 2\}$, for which the sum of those numbers would be equal to 1, so this definitely results in a NO answer.
- In the even case, when we divide by 2 we get what the sum of one of the subsets should be, let's call this a.
- We can give the original input numbers to $SUBSETSUM$ and this a number, which will tell us if it can find a subset for which the sum is exactly this a. Since the total sum was $2a$, this also means that the remaining numbers will also add up to a as well, making this a partition.

- If the original input is even, but cannot be partitioned, then there is no way the $SUBSETSUM$ will find a subset of these numbers for which, their sum is a. If we indirectly assume that it did, then that would mean that the remaining numbers will also add up to a, so this is a partition, which is a contradiction.
- To calculate the total sum of these n numbers and divide by 2, we have used up polynomial time, so the transformation function is polynomial as well.

8.9 Session 8, Exercise 8

Exercise

Give a Karp-reduction from $SUBSETSUM$ to $PARTITION$.

Solution

- Now we must do the opposite as in Session 8, Exercise 7: we must solve $SUBSETSUM$, using $PARTITION$.
- Let's say that the input is $\{s_1, s_2, \ldots, s_n\}$ and b. We need to find a subset of this, that sums up to b.
- Let's calculate the total sum again! This will be S.
- Let's add two additional items to the set: $S + b$ and $2S - b$ and give this to $PARTITION$: $\{s_1, s_2, \ldots, s_n, S + b, S - b\}$
- What will $PARTITION$ do with these? It cannot put $S + b$ and $2S - b$ in the same part, since their sum is $S + b + 2S - b$, which is $3S$, so even adding everything else to the other part, that would still become only S, which is too small.
- Instead $S + b$ and $2S - b$ will be in separate parts. For the remaining elements in $\{s_1, s_2, \ldots, s_n\}$, $PARTITION$ will try to arrange them in the two sets, so that their sum is equal.
- The total sum of everything in $\{s_1, s_2, \ldots, s_n, S + b, S - b\}$ is $S + S + b + 2S - b = 4S$, so $PARTITION$ will divide these into $2S$ sized sums. Whatever goes next to $2S - b$, has to sum up to b then. The remaining parts then will sum up to $S - b$, which added to $S + b$ will result in a sum of $2S$ also.
- This means that the items that went next to $2S - b$ are the solution to the $SUBSETSUM$, if it exists, and if it doesn't then there is no way we can arrange them in this way.
- We used $O(n)$ additions in our transformation algorithm only, so this was done in polynomial time too.

8.10 Session 8, Exercise 9

Exercise

Prove that if $coNP \neq NP$, then $MAXCLIQUE \notin P$.

Solution

- Let's indirectly state that $MAXCLIQUE \in P$, while $coNP \neq NP$!
- $MAXCLIQUE$ is an NP-complete language, which means that it's NP-hard.
- This means that all languages in NP can be Karp-reduced onto it.
- $MAXCLIQUE \in P$ means that we have found a polynomial solution to it.
- This means that using all the existing Karp-reductions, we can insert the 'missing piece', the polynomial $MAXCLIQUE$ solver into them, to achieve completely polynomial solvers for all languages in NP.
- This means that $NP \subseteq P$.
- $P \subseteq NP$ already, so this leads to $P = NP$.
- Now since any language in NP can be solved (decided) in polynomial time, this also means that the complementer language (which only inverts the YES / NO answer) can be solved the same way as well.
- The complementer languages of the languages in NP comprise the $coNP$ set, which now means that all languages in $coNP$ have polynomial solutions too.
- This means that $coNP \subseteq P$.
- $P \subseteq coNP$ also, so now we achieved $NP = P = coNP$. (And we are rich too! :))
- Actually the last steps were not needed for $coNP$, but I wanted to show that too.
- We could have stopped at $P = NP$, which means that since $MAXCLIQUE \in NP$, it is now also in P, which is a contradiction to the original indirect assumption.
- So we have proven the original statement to be true.

8.11 Session 8, Exercise 10

Exercise

Prove that if language L is NP-complete, and $L \in NP \cap coNP$, then $NP = coNP$.

Solution

- If L is NP-complete, then it means that it is NP-hard, which means that all languages in NP can be Karp-reduced onto it.
- So $\forall L^{'} \in NP \Rightarrow L^{'} \prec L$.
- Since L is also in $coNP$, this means that a polynomial witness exists for the NO answer of that language.
- The $L^{'} \prec L$ gives us a polynomial transformation function, for which the witness can be the witness for the NO answer of $L^{'}$ as well, similarly to how we reasoned in Session 8, Exercise 5.
- Since we have a witness for the NO answer for $L^{'}$, it means that $L^{'} \in coNP$.
- So putting everything together $\forall L^{'} \in NP \Rightarrow L^{'} \prec L \Rightarrow L^{'} \in coNP$. Everything that is in NP is also in $coNP$, so $NP \subseteq coNP$.
- Using $\forall L^{'} \in NP \Rightarrow L^{'} \prec L$ again, we know (from Session 8, Exercise 2, that the complementer languages also have the same Karp-reduction, so $\overline{L^{'}} \prec \overline{L}$ is also true.
- If we iterate over the NP set with $L^{'}$, that means that $\overline{L^{'}}$ iterates over the entire $coNP$ set (the definition of $coNP$ is the complementers of all languages in NP).
- This means that we have just proven that $\forall \overline{L^{'}} \in coNP$, $\overline{L^{'}} \prec \overline{L}$ is true.
- If $L \in NP \cap coNP$, this also means that $\overline{L} \in NP \cap coNP$, since it was in NP, so its complementer is in $coNP$ and it was in $coNP$, so its complementer is also in NP.
- We will use $\overline{L} \in NP$, which gives us a witness for the YES answer for $\overline{L^{'}}$ as well, using the Karp-reduction, which means that $\overline{L^{'}} \in NP$.
- We have just proven that $\forall \overline{L^{'}} \in coNP \Rightarrow \overline{L^{'}} \in NP$, so all languages in $coNP$ are also in NP, so $coNP \subseteq NP$.
- Putting $NP \subseteq coNP$ from previously and the newly achieved $coNP \subseteq NP$ together, we get $NP = coNP$.

8.12 Session 8, Exercise 11

Exercise

Let us assume that we have a procedure that decides about any boolean formula whether it belongs to SAT or not in polynomial time. How can this procedure be used to find a substitution of variables for a given formula $\Phi(x_1, x_2, \ldots, x_n)$ in polynomial time that makes the formula true?

Solution

- This exercise is important, because this is where we convert solvers of decision problems, to solvers that actually tell you the solution (which is usually what we need, not just a yes or no answer).
- The first step is to run the SAT solver on the original formula and see if it can be satisfied.
- If not, then we are sad, but done.
- If it can be, then we take x_1 and assign it, for example a $False$ value and give it to the SAT solver again.
- If the answer is still yes, that means that we can keep $x_1 = False$ and continue to substitute other variables.
- If the answer is no, this means that while the original formula was satisfiable, that cannot happen if $x_1 = False$. This immediately means that x_1 must be $True$! And this is where the exponential boom does not happen, because we do not have to try $x_1 = True$, we know it is satisfiable!
- Now we know the value of x_1 for which the formula is still satisfiable, and we continue to do the same thing for the rest of the variables.
- Since we ran the SAT solver for all variables only once, so n times in total, this results in a polynomial solution that finds the correct variable assignment.

8.13 Session 8, Exercise 12

Exercise

Let us assume that we have a procedure that decides about any n-vertex graph whether it has a Hamiltonian cycle. How can this procedure be used to find a Hamiltonian cycle of a given graph G in polynomial time?

Solution

- Similar to Session 8, Exercise 11.
- We iterate over the edges of the graph. For each edge, we remove it from the graph and ask the solver if the graph still has a Hamiltonian-cycle.
- If the answer is yes, then we don't need that edge, so we forget about it. :)
- If the answer is no, then we put the edge back and never touch it again.
- Repeat this for all edges.
- The edges remaining after all of this will be exactly one of the Hamiltonian-cycles of the graph.
- We ran the HAM solver as many times as there were edges, exactly once, so this is a polynomial time algorithm.

9 April 27th (Session 9): NP-completeness, Integer Programming, Linear Programming

9.1 Session 9, Exercise 1

Exercise

Prove that the following languages are NP-complete.

a) Language of those graphs that can be properly colored using 3 colors, such that each color occurs on the same number of vertices.

b) Language of those quadruples (G, a, b, k), where G is an undirected graph, $a, b \in V(G)$ and $k > 0$ integer and there exists a path between vertices a and b of G of length at least k.

c) The language consisting of such triples (G, a, b) that G is an undirected graph $a, b > 0$ are integers and G an induced subgraph isomorphic to complete bipartite graph $K_{a,b}$.

d) The language consisting of such undirected graphs G that have a cycle C such that every vertex $v \notin C$ is connected by an edge to a vertex of C.

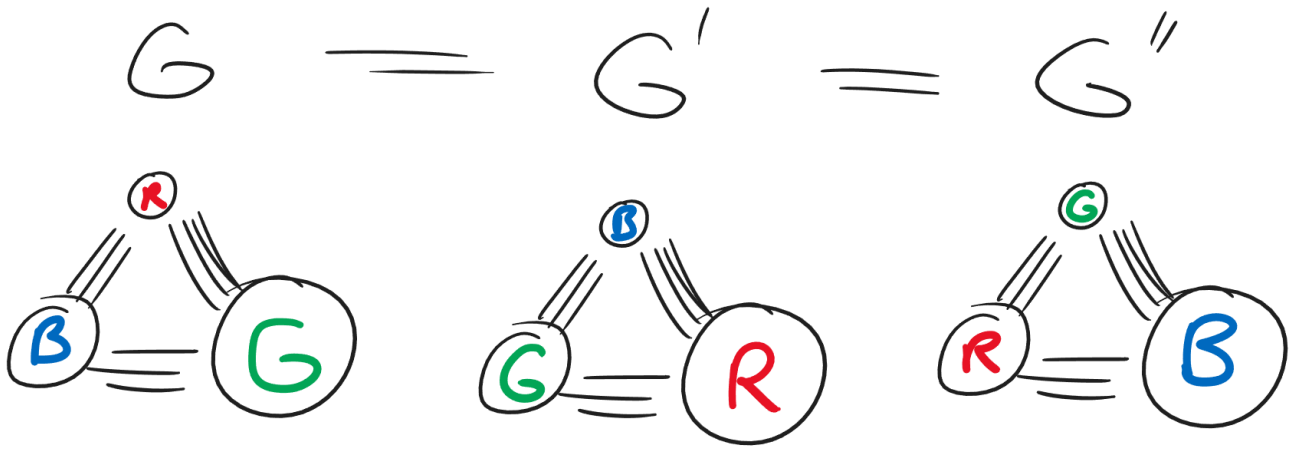
Solution

- To prove that a language is NP-complete we must prove that it is both in NP and NP-hard.
- To prove that a language is in NP we use the Witness Theorem.
- To prove that a lanugage us NP-hard we find an already known NP-hard / NP-complete language and Karp-reduce this one onto the language in question, so we solve the well known NP-hard problem using the hypothetical solver for problem in question. E.g. $WellKnown \prec Question$.

Task a):

- We first start with the Witness Theorem:
- The Witness will be the description of a 3-coloring where each color occurs the same number of times. For v vertices we can define an array $A[1:n]$ of numbers $1, 2$ and 3, where $A[i] = c$ means that the ith vertex has color c.
- The size of the Witness is polynomial relative to the input size, since the witness size is $2bits \cdot n = O(n)$ (we can represent the $1, 2, 3$ numbers using 2 bits), while the input size is $O(n^2)$ (the adjacency matrix of the graph).
- To check if this witness is a correct solution we check:
  - The size of the array is the same as the number of vertices on the inputs.
  - Only numbers $1, 2$ and 3 are present.
  - We count the occurrences of the numbers and check that they are the same.
  - We iterate through all edges of the graph and check that their ends don't have the same color, as that would be an improper coloring.
- All these checks are iterating the vertices once or the edges once, so in total the runtime complexity is $O(n^2)$.
- Then we try to find a well-known NP-complete problem to Karp-reduce. You can find a table of all NP-complete problems we have studied in exercise Exam: 2022. 05. 30., Exercise 3.
- It is a good idea to start with a similar problem and if it is not working out, then move on to others.
- In this case, $3-COLOR$ is a good choice to start with, since that is the most similar to our problem.
- So we are trying to give a $3-COLOR \prec L_a$ Karp-reduction: transform the input of $3-COLOR$ so that we can give it to the L_a solver and it will correctly return YES and NO.
- The solver of L_a does almost what we need: but it additionally checks that the colors have to be assigned to the same number of vertices.

- We somehow want to skip this check, or make sure that it always succeeds.
- In the general G graph input of $3-COLOR$, it is not necessarily true that the number of the 3 colors will be equal, so we somehow want to transform G to make sure that they are equal.
- We can do this by creating 3 copies of the G graph.
- If a proper 3-coloring exists on G, then by creating 3 copies, we allow the solver of L_a to rotate the colors around the other 2 copies of the G graph, so in total it would assign one copy of each vertex red, one copy of each vertex blue and one copy of each vertex green, resulting in $n = n = n$ number of colors.



- If no propert 3 coloring exists, on the original G graph, then there won't be any proper 3 coloring on this new graph, since it contains G three times.
- So with this modification, we are able to satisfy the one additional rule the L_a solver is checking for and this makes sure that L_a returns YES for the $f(G)$ graph exactly when for the $3-COLOR$ G is a YES instance and NO when it's a NO instance.
- We can create a copy of G 2 times in $O(n^2)$ time, by making the adjacency matrix 3 times 3 larger and copying the original one into it 3 times.
- So f also runs in polynomial time, which we need to check for a Karp-reduction.

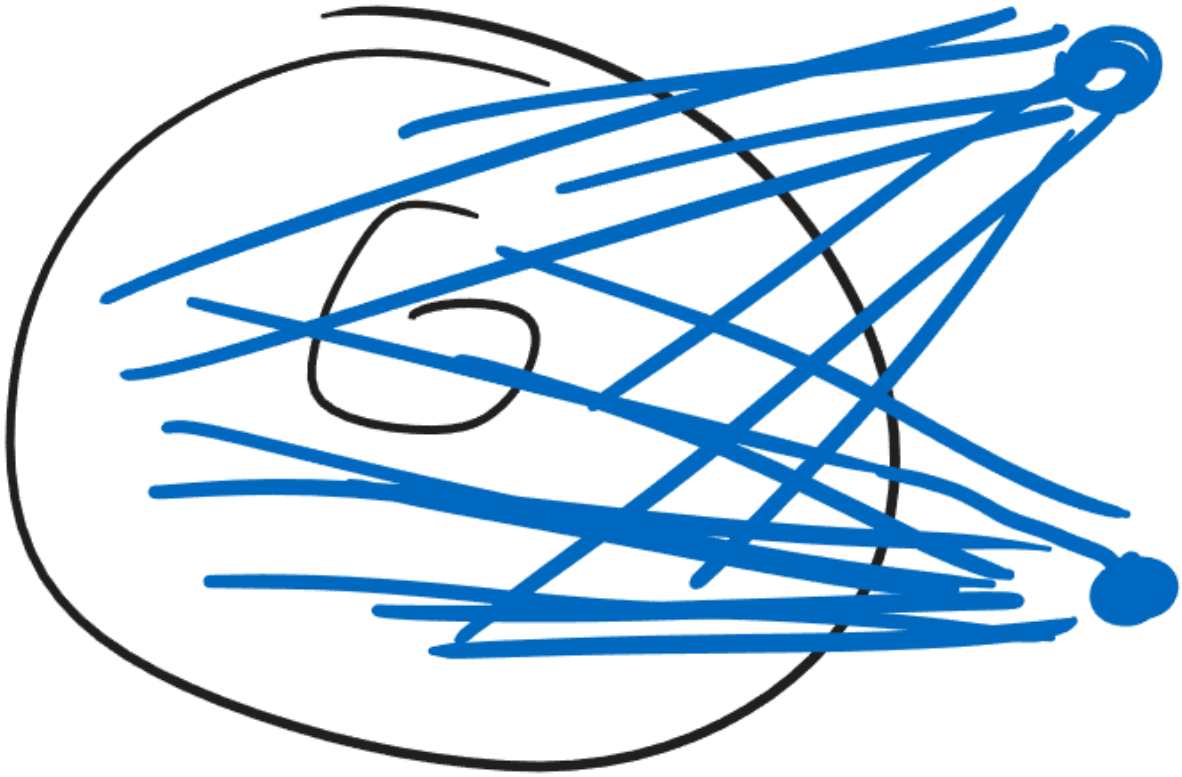
Task b):

- Witness Theorem: The witness is the description of the undirected path in question, given by the list of vertices along it. Since a path has to have unique vertices along it, at most there will be n vertices on the path and a single vertex can be given by it's index in binary format: $O(\log n)$ bits. So in total $O(n \log n)$ size, while the input is $O(n^2 + \log_2(a) + \log_2(b) + \log_2(k))$, so it is polynomial.
- Witness Checking Algorithm: Chek if this is indeed at least k vertices (and they are all indeed part of G) and that for all v_i and v_{i+1} vertex on the path, the edge exists between them in G. Also $v_1 = a$ and the last $v_m = b$.
- We can Karp-reduce the $s-t-HAMPATH$ problem onto this. By assigning $k = n$, we require the path to be full length, of n vertices, which is only possible if it is a Hamiltonian-path. Then, $a = s, b = t$ and the graph is the same. This transformation is trivially done in polynomial time.

Task c):

- Witness Theorem: The witness will be the list of vertices on side A and side B of the induced bipartite graph in question. This is similarly given in at most $O(\log n)$ bits, which is similarly polynomial as previously.
- Witness Checking Algorithm: Check all pairs of edges between vertices of A and B, since we need a fully connected subgraph. Also check that there are no edges running inside A and inside B either, since the $K_{a,b}$ graph is induced, meaning that all edges present in the original graph must be part of it, so if $K_{a,b}$ is bipartite, it means that there are no edges running in A and in B either. This can be done in $O(n^2)$ time, so this is also polynomial. Check also that $|A| = a$ and $|B| = b$.

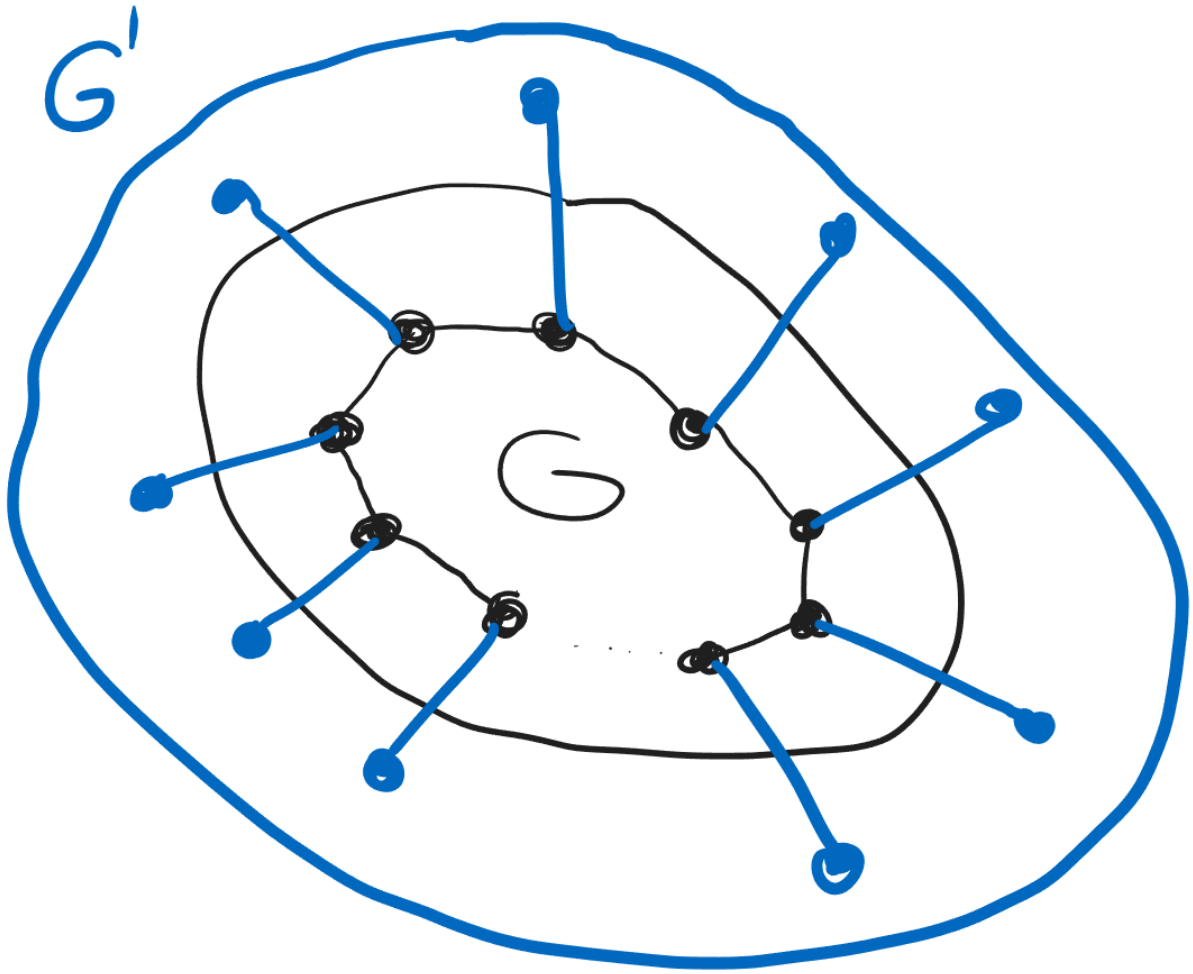
- Karp-reduction from well-known NP-complete problem: we can use the fact that A and B must be independent sets and Karp-reduce from $MAXINDEP$: $MAXINDEP \prec L_c$.
- We add 2 additional vertices to the G graph and connect both to all original vertices of G, but not to each other.
- Then, $MAXINDEP$'s input is (G, k), where k is the size of the independent set we are looking for. We can transform this into $(G', k, 2)$.
- If the L_c solver finds such a $K_{k,2}$ bipartite graph, then it was either the 2 vertices we just added, connected to a size k independent set, or it might even find some induced subgraph completely in G that meets the criteria. Either way, that will contain a size k independent set for $|A|$, which is the correct solution to the original problem. If no such set exists, then L_c won't find anything.



Task d):

- Witness Theorem: The witness will be the description of this cycle in question. Given by the vertices of the cycle for example, in the order which they are encountered if we walked along the cycle. Similarly polynomial, size is $O(\log n)$ bits.
- Witness Checking Algorithm: Check that this is indeed a cycle, all edges exist between v_i and v_{i+1} neighbours, also v_n and v_1. Check that the remaining vertices of the graph are all connected to one of the vertices of this cycle. This can be done by checking all edges at most, in $O(n^2)$ time. This is polynomial as well.
- Karp-reduction from well-known NP-complete problem: Let's use HAM, since that also has a cycle, the HAM cycle. We want our solver to find a HAM cycle. However the solver allows these dangling vertices as well. We want to modify our original input graph, such that all original vertices must be present in the cycle the solver finds. We can do this by adding n extra vertices to graph G, the ith connected to the original ith vertex of G.
- The blue vertices can never be part of the cycle found, since they only have a single edge. So they all will be dangling. This means that their neighbour must be part of the cycle, making all original vertices a must for the cycle that the L_d solver finds. This means that the L_d solver will find a HAM-cycle if it exist, and won't find anything if it doesn't.

- Adding n vertices to G and n corresponding edges can also be done in polynomial time.



9.2 Session 9, Exercise 2

Exercise

Let us assume that $P \neq NP$ and $L_1 \in P$. Is it possible that

a) L_1 can be Karp-reduced to an NP-complete language L_2?

b) an NP-complete language L_2 can be Karp-reduced to L_1?

c) $L_1 \in NP$?

Solution

- Task a): Yes, almost all Karp-reductions exist from all polynomial languages.
- This is because during the polynomial transformation, we can actually solve the question, if it is in P and then give two mock data, one that is a YES-instance and one that is a NO-instane to the target solver.
- The only reason why this might not be possible if the target solver is the empty language (everything is a NO instance) or the complete Σ^* (everything is a YES instance), in which case we won't be able to come up with one of the mock inputs needed.
- Task b): No, because if $L_2 \prec L_1 \in P$ exists, then that means that $L_2 \in P$ as well (polynomial the Karp-reduction tranformation function + the polynomial solver for L_1 becomes a solver for L_2).
- This means that we have just solved an NP-complete problem in polynomial time, which mans that all problems in NP can be solved in polynomial time (since all Karp-reduction from all problems in NP exist). We know that $P \subseteq NP$, so now it becomes $P = NP$, but we assumed that $P \neq NP$, which is a contradiction.
- Task c): Yes it is always true, since $L_1 \in P \subseteq NP$.

9.3 Session 9, Exercise 3

Exercise

Is the following problem in P or is it NP-complete? We are given integer numbers $a1, \ldots, a_n$ and question is whether this set of numbers can be partitioned into three parts so that sum of the numbers in each part is the same.

Solution

- This smells like an NP-complete problem, because it looks similar to $PARTITION$.
- First prove that it is in NP using the Witness Theorem:
- The witness will be a description of such a partition, which is given by a list of n numbers between 1 and 3 the ith representing which partition the a_i belongs to. This is polynomial, $O(n)$ size.
- Witness checking will be summing the partitions separately and comparing that they indeed sum to the same number and also that no index number left out (by giving a number other than $1, 2, 3$ on the input or giving less than n numbers on the input). This can be done in polynomial time.
- Now prove that it is NP-hard by Karp-reducing the well-known $PARTITION$ NP-complete problem onto it.
- We can take the sum of all a_i's from the input of $PARTITION$ and divide by 2. Add this c value to the list of a_i's and give this list to the solver for L.
- The solver must put c into a a separate group, since if it tried to put additional numbers next to it, the group would become too big, and the remaining numbers won't be able to balance it out.
- So if there is a $3 - way$ partition of these numbers to an equal sum, it can only be done by c being a separate group, meaning that the other two groups will be a $2 - way$ partition of the original input.
- The transformation here is polynomial as well, we sum n nubmers and do some additional arithmetic, done in $O(n)$.

9.4 Session 9, Exercise 4

Exercise

There are n people working at an organization, where there are c committees. We want to schedule committee meetings. Two of the committees may have a meeting on the same day if they have no common members. Let positive integer k be given and the lists of members of each committee. We want to determine whether the meetings of all committees can be scheduled in a period of k days. Either give a polynomial time algorithm that solves this problem or prove that it is NP-complete.

Solution

- It is NP-complete. Scheduling problems are usually related to coloring. In this case, we will solve the $COLOR$ problem using a solver for L.
- The Witness Theorem can be used to prove that it is in NP, give a witness, a vertifyer algorithm and prove that the witness is polynomial in size and he verifyer runs in polynomial time.
- The input of $COLOR$ is graph G and the m number of colors we want to use to color it. We will translate each vertex of the graph G into a committee, so v_i vertex to c_i comittee.
- Then, where there is an edge running between two vertices, that will correspond to a person, who is present in both comittees. We assign p_i person to e_i, edge, add p_i to the comitees corresponding to both ends of the edge.
- Then, we will say that we want to schedule meetings in $k = m$ days, each day corresponding to a single color of the original graph.
- If two committees can be scheduled to meet at the same day, that means they don't have common members, so there is no edge running between their vertices.
- So a proper scheduling onto k days corresponds to a proper coloring of the graph using k colors.
- If no proper scheduling is possible onto k days, then no k coloring is possible, either. If a coloring was possible, that would correspond to a scheduling as well.
- It can be shown that this transformation is also polynomial.

9.5 Session 9, Exercise 5

Exercise

Consider the problem that what is the maximum weight of a path in a edge weighted undirected graph. Give the language that belongs to this problem and show that it is NP-complete.

Solution

- It is NP-complete.
- Use Witness Theorem to prove that this is in NP.
- Then, we will Karp-reduce the HAMPATH problem. We assign weights of 1 to all edges in the graph, then we will try to find a path of weight $n-1$. This is only possible if that is a Hamiltonian-path.
- We can do this transformation in polynomial time.

9.6 Session 9, Exercise 6

Exercise

Prove that there exist Karp-reductions from the following languages to s-t-HamPath.

- HamPath: Language of graphs that have Hamiltonian path.
- HAM: Language of graphs that have Hamiltonian cycle.

Solution

- We can use the fact that $s-t-HAMPATH$ is NP-complete and so is HAM and $HAMPATH$.
- This means that all of them are in NP and NP-hard as well.
- Let's use from HAM and HAMPATH that they are in NP nd for $s-t-HAMPATH$ that it is NP-hard, so all Karp-redutions from all languages in NP must exist onto it, namely HAM and HAMPATH too as well.

If we want to give a Karp-reduction, not just prove its existence:

- To solve HAMPATH using s-t-HAMPATH, we can add two more vertices and connect them to all of the other vertices of G but not to each other. Assign these vertices to s and t and give this G' graph to the $s-t-HAMPATH$ solver. If $s-t-HAMPATH$ finds an $s-t-HAMPATH$, then the original graph containd some Hamiltonain path in it, to which we connected s and t If it finds none, then none can exist, otherwise it would result in an $s-t-HAMPATH$.
- To solve HAM using $s-t-HAMPATH$, we can't exactly do the same thing as above, since we need to make sure that the cycle closes, so whatever the neighbours are of s and t in the path, those two are connected to each other as well. You can find the solution in Session 8, Exercise 4.

9.7 Session 9, Exercise 7

Exercise

The input of each problem below is an undirected graph $G(V, E)$ and a subset $S \subseteq V$ of its vertices. Determine which problem is in P and which is NP-complete.

a) Question: does there exist a spanning tree of G such that every element of S is a leaf of the tree?

b) Question: does there exist a spanning tree of G whose leaves are exactly the vertices in S?

c) Question: does there exist a spanning tree of G whose leaves are contained in S?

Solution

- Task a) is in P, since we can just remove the vertices of S from G, find a spanning tree using BFS, then connect back everything form S into one of their neighbours (that are not in S). If someone in S only has a neighbour that is also in S, then this is not possible.
- Task b) and c) are NP-complete. They are in NP (Witness Theorem) and we can Karp-reduce the $s - t - HAMPATH$ problem onto them, by assigning $S = \{s, t\}$, so these two must be the leaves of the spanning tree. A spanning tree with two leaves is a path, so this will result in an $s - t - HAMPATH$ for b), and for c), here is no tree which only has one or zero leaves, so this is going to result in the same task, nothing can be left out of S.

9.8 Session 9, Exercise 8

Exercise

A big firm moves to a new headquarters into a building of F floors. Each floor has the same amount of office space. Each department has a given office space requirement. We want to determine whether the move can be done so that no department is cut, that is each department must have all its offices on the same floor (however, more than one departments can have offices on the same floor). What is the language that belongs to this problem? Is it in P or is it NP-complete?

Solution

9.9 Session 9, Exercise 9

Exercise

Solution

9.10 Session 9, Exercise 10

Exercise

Solution

10 May 5th (Session 10): Binpacking FF/FFD, Dynamic Programming

10.1 Session 10, Exercise 1

Exercise

Let the weights in problem $BINPACKING$ be $s_1 = 0.4; s_2 = 0.7; s_3 = 0.1; s_4 = 0.5$. Perform in this case

a) algorithm FF

b) algorithm FFD

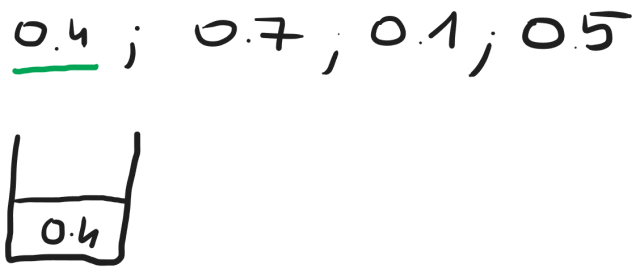
- How many bins are used by an optimal packing?
- What if the last weight changed to $s_4' = 0.6$?

Solution

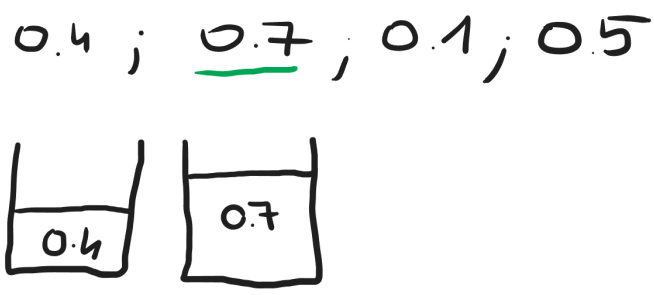
- In all binpacking exercises the bins are of size 1.0, while weights can be between $0 < s_i \leq 1$.

FF

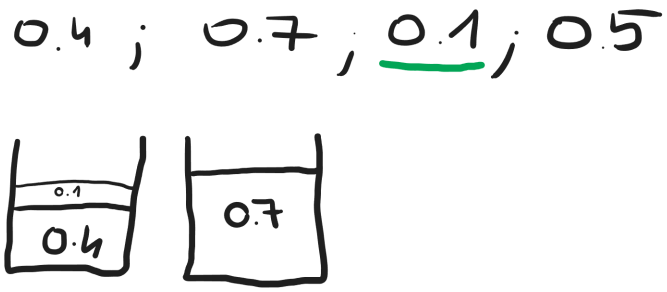
- FF, or first fit will iterate over the given weights, in the order which they are given and will try to fit it to the first available bin. If there is no bin, into which it would fit, then it opens a new bin.
- For the first weight of 0.4, FF opens a new bin.



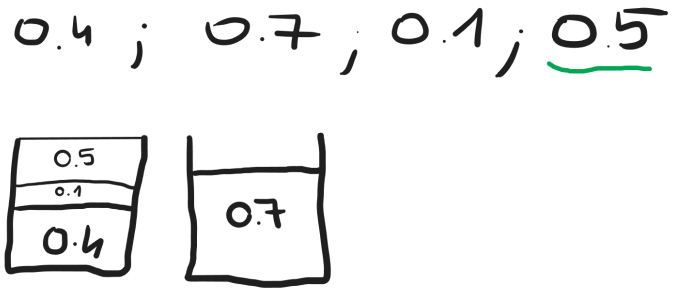
- For 0.7 we must open a new bin, since $0.4 + 0.7 = 1.1$, so they won't fit into a single bin.



- 0.1 fits immediately into the first bin, next to the 0.4, so the algorithm will put it there and will not try further bins.

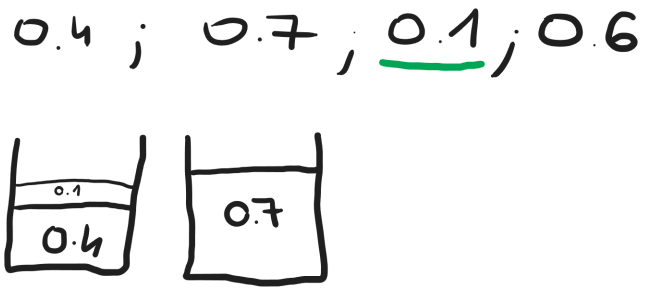


- 0.5 fits next to $0.1 + 0.4$, since $0.1 + 0.4 + 0.5 = 1.0$, so they fill the first bin exactly!

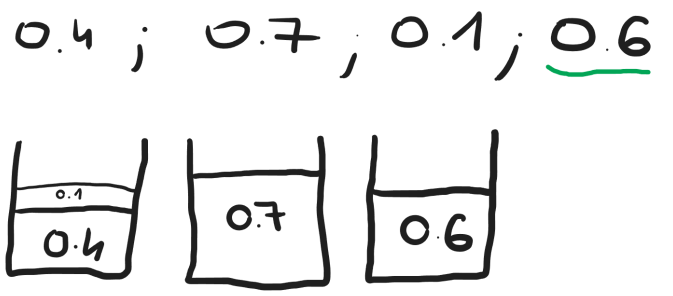


FF with $s'_4 = 0.6$

- The first three weights are placed the same way as previously, since nothing changed here.

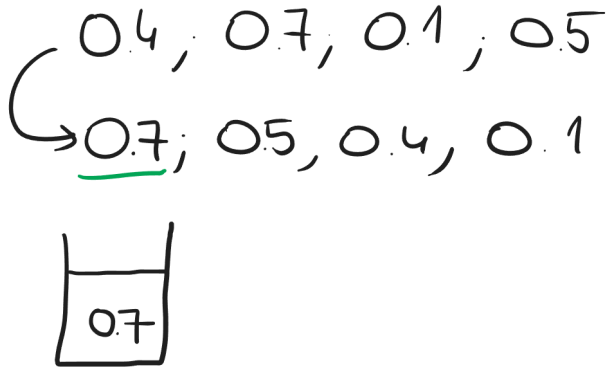


- 0.6 will not fit next to $0.1 + 0.4$, since $0.1 + 0.4 + 0.6 = 1.1$, neither next to 0.7, since $0.7 + 0.6 = 1.3$ so we need to open a new bin for it!

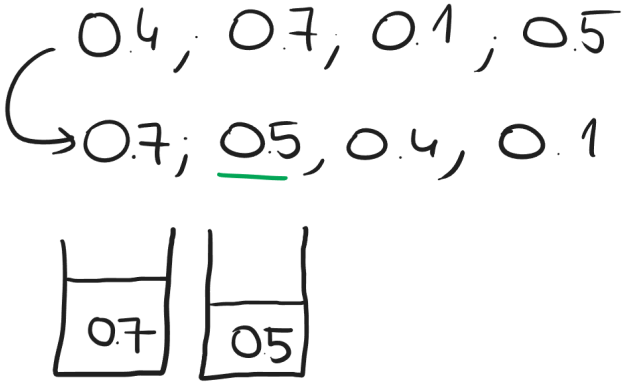


FFD

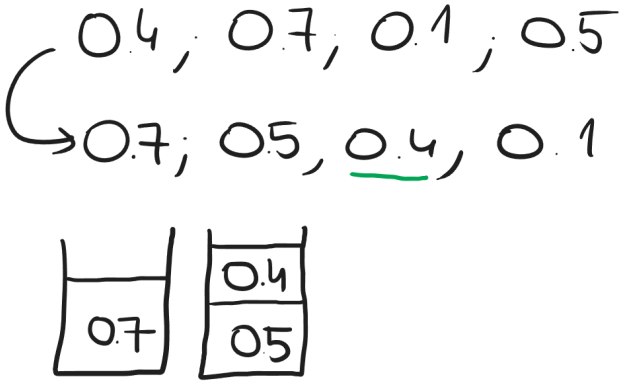
- FFD, or first fit decreasing will first sort the weights in decreasing order, then iterate over them in this order and will try to fit the current item to the first available bin. If there is no bin, into which it would fit, then it opens a new bin.
- For the first weight of 0.7, FFD opens a new bin.



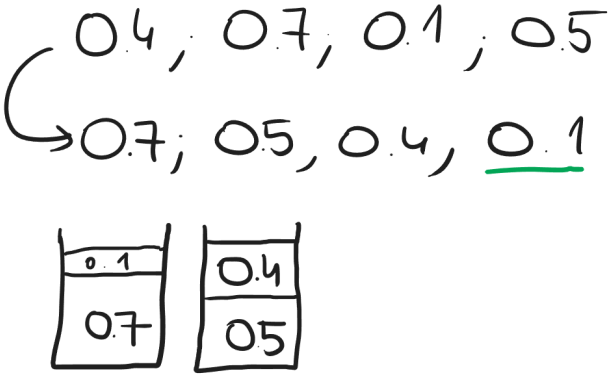
- For 0.5 we must open a new bin, since $0.5 + 0.7 = 1.2$, so they won't fit into a single bin.



- 0.4 won't fit next to 0.7, since $0.7 + 0.4 = 1.1$, however it fits next to 0.5, since $0.5 + 0.4 = 0.9$, so we put it there.

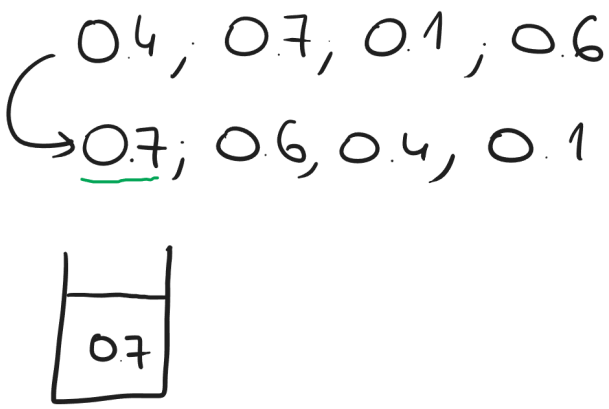


- Finally, 0.1 fits immediately next to 0.7, since $0.7 + 0.1 = 0.8$, so the algorithm stops here!

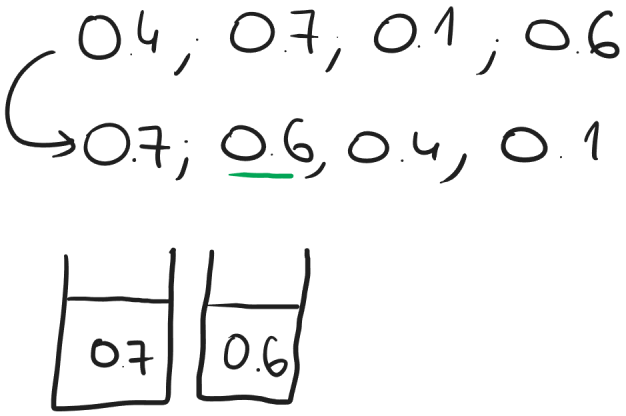


FFD with $s'_4 = 0.6$

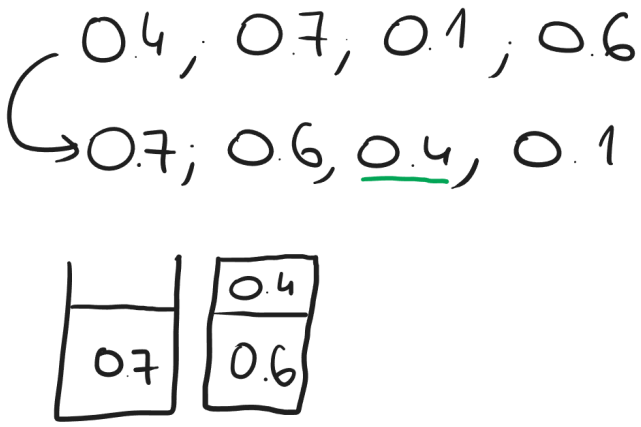
- *FFD* sorts the weights again and opens a new bin for the first weight of 0.7.



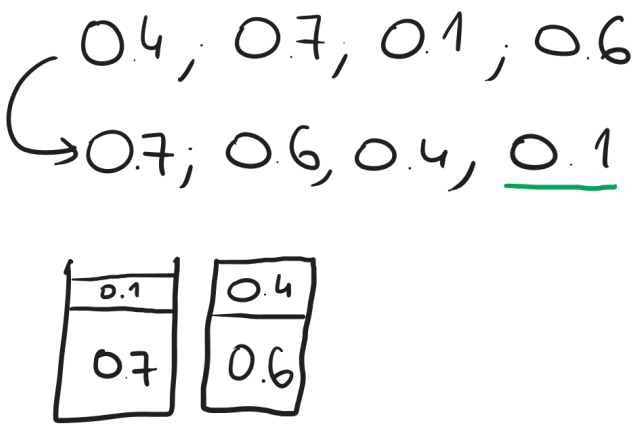
- For 0.6 we must open a new bin, since $0.6 + 0.7 = 1.3$, so they won't fit into a single bin.



- 0.4 won't fit next to 0.7, since $0.7 + 0.4 = 1.1$, however it fits next to 0.6, since $0.6 + 0.4 = 1.0$, so we put it there.



- Finally, 0.1 fits next to 0.7, since $0.7 + 0.1 = 0.8$, so we put it into the first bin and the algorithm stops here!



The optimal solutions

- For weights 0.4, 0.7, 0.1 and 0.5, the optimal solution uses 2 bins, for example the first bin $0.1 + 0.4 + 0.5 = 1.0$ and the second bin 0.7. We know this is an optimal solution, because every single bin, except (maybe) the last one is filled fully, so there is no possible way to compress this further! Both FF and FFD finds an optimal solution, which uses two bins.
- For weights 0.4, 0.7, 0.1 and 0.6, the optimal solution uses 2 bins, for example the first bin $0.4 + 0.6 = 1.0$ and the second bin $0.7 + 0.1 = 0.8$. We know this is an optimal solution, because every single bin, except (maybe) the last one is filled fully, so there is no possible way to compress this further! In this case FF used 3 bins, while FFD found an optimal solution using 2 bins.
- In the second example, we can see that FFD is an algorithm that might be more successful in compressing the weights into a smaller amount of bins, since it first tries to fit the bigger items, and then comes back to fill in the leftover space with smaller items. Compare this to FF, that might put the smaller items together into a single bin, and then it must open new bins for every single bigger item.
- We could ask what would be the benefit of using FF over FFD? One benefit of FF is that it does not need to know in advance all of the weights that will be placed in the bins, since it does not need to sort them before starting the algorithm. This means, that FF is an online algorithm, so it can be used in a setting where we do not know the upcoming sizes.
- A good real-life example of this is cloud computing, where requests are sent in to rent a chunk of processing power (weight) for a period of time, which must be allocated to physical processors (bins). Since we do not know in advance, what requests will be sent in next, we cannot use FFD to optimize (minimize) the number of used physical processors.

10.2 Session 10, Exercise 2

Exercise

In the problem $BINPACKING$, let's have the folowing weights:

- $6k$ number of objects of weight 0.51
- $6k$ number of objects of weight 0.27
- $6k$ number of objects of weight 0.26
- $12k$ number of objects of weight 0.23

How are these packed by FFD? Is that optimal?

Solution

What will FFD do?

- FFD will pack the weights in decreasing order.
- It will open up new bins for every single item weighing 0.51, in total $6k$ bins.
- Then, it will be able to put a single weight of 0.27 next to the first 0.51 weight, since $0.27 + 0.51 = 0.78$. However, it won't be able to put another weight of 0.27 next to the first $0.27 + 0.51$, since $2 \cdot 0.27 + 0.51 = 1.05$. So it will place a single 0.27 next to each 0.51. After placing all of them, we will have $6k$ bins, each containing $0.51 + 0.27 = 0.78$.
- Since a weight of 0.26 won't fit next to the $0.51 + 0.27 = 0.78$, as it would become $0.51 + 0.27 + 0.26 = 1.04$, so these will be placed in new bins. 3 of them will fit into a single one, since $3 \cdot 0.26 = 0.78$, and 4 would be too much $4 \cdot 0.26 = 1.04$. So it will open up, in total $\frac{6k}{3} = 2k$ bins, each containing 3 weights of 0.26.
- Finally, the 0.23 will not fit into the type $0.51 + 0.27 = 0.78$ bin, neither into the type $3 \cdot 0.26 = 0.78$ bin, since in both cases the total weight would become 1.01, so these will also be placed into new bins.
- 4 of the 0.23 can be placed into a single bin, since $4 \cdot 0.23 = 0.92$, but 5 would be too much. So the algorithm will open $\frac{12k}{4} = 3k$ new bins, each of which will contain 4 weights of 0.23.
- So the result of FFD will be:
  - $6k$ bins containing $0.51 + 0.27 = 0.78$.
  - $2k$ bins containing $3 \cdot 0.26 = 0.78$.
  - $3k$ bins containing $4 \cdot 0.23 = 0.92$.
- Using $11k$ bins in total.

What is an optimal solution?

- When we are trying to figure out the optimal solution, we must aim to make as many completely full bins as possible.
- In this case, a type of bin that works well is $0.51 + 0.26 + 0.23 = 1.0$, since this is completely full. We can make $6k$ of these bins, using up all 0.51 and 0.26 weights, and half of the 0.23 weights.
- The remaining $6k$ number of 0.27 and $6k$ number of 0.23 weights can be placed into a type of bin with $2 \cdot 0.27 + 2 \cdot 0.23 = 1.0$, using up $\frac{6k}{2} = 3k$ bins in total.
- Since all of the bins we have are filled completely full, it is easy to see that this is an optimal solution, there is no way to compress this further.
- We can see, that the optimal solution is:
  - $6k$ bins containing $0.51 + 0.26 + 0.23 = 1.0$.
  - $3k$ bins containing $2 \cdot 0.27 + 2 \cdot 0.23 = 1.0$.

- The optimal solution uses $9k$ bins in total.

Notes:

- We can see that FFD used $\frac{11}{9}$ as many as the optimal solution.
- In general, it can be proven, that FFD will use at most $\frac{11}{9}$ times as the optimal, plus $\frac{6}{9}$ bins, which we denote by $FFD \leq \frac{11}{9}OPT + \frac{6}{9}$.

10.3 Session 10, Exercise 3

Exercise

Consider the special case of $BINPACKING$ where each weight is either $\frac{1}{2}$ or 1. Prove that this version is in P.

Solution

- The algorithms we have seen for $BINPACKING$, FF and FFD are not going to give an optimal solution in all cases, they can only approximate the optimal solution. For example, FFD will use at most $\frac{11}{9}OTP + \frac{6}{9}$ bins.
- Now we need to come up with an algorithm that will **always** give us the optimal solution, **in polynomial time**, for the special case of having only 0.5 and 1.0 weights, in order to prove that this problem is in P.
- The optimal solution is the following: place all weights of 1.0 into separate bins. They will fill their own bin fully.
- Next: pair up the 0.5 weights and place these pairs into separate bins. $2 \cdot 0.5 = 1.0$, so these will fill theirs bins fully as well.
- Finally, if the number of 0.5 weights is odd, there will be a single weight of 0.5 left, which we will place into a separate bin.
- We can see that this will give us an optimal result, since all of the bins, except (maybe) the last one are completely fully filled, so there is no way to compress this solution further.
- This was indeed a polynomial solution, since we just counted the number of 1.0 weights and the 0.5 weights and opened bins according to these numbers.
- By the way, this is exactly what FFD will do if we run it on an input like this, and FFD is also polynomial (sorting can be done in $O(n \log(n))$, so we have just proven, that in the special case of having only 0.5 and 1.0 weights, FFD will always give us the optimal solution.

10.4 Session 10, Exercise 4

Exercise

Consider the special case of $BINPACKING$ where each weight is larger than $\frac{1}{3}$. Is this special case in P or is it NP-complete?

Solution

Hint 1

- It is very important in this exercise, that the weight is **strictly** larger than $\frac{1}{3}$, it cannot be equal to it.
- This means that we know for a fact that in every single bin, we can but at most 2 weights. If we put 3 weights into a bin, all of them being larger than $\frac{1}{3}$, they would sum up to larger than 1.0, so they wouldn't fit.

Hint 2

- We can represent the weights using a graph. The ith vertex of the graph corresponds to the s_i weight, while we say that there is an edge between vertices i and j, if the weights s_i and s_j can be placed into a single bin, meaning $s_i + s_j \leq 1.0$.

Solution

- To find an optimal placement using this graph, we must find the maximum matching in it. Each $\{i, j\}$ edge in this matching corresponds to a bin that contains the weights s_i and s_j, since they have an edge running between them, they will fit into the same bin. The vertices that are not part of this matching will get a separate bin for the corresponding weight.
- If for n weights, the maximum matching contains k edges, then that will result in k bins that have two weights in them, while the remaining $n - 2k$ weights will be placed in separate bins. So in total, it will correspond to using $n - k$ bins.
- We can prove that the maximum matching corresponds to the optimal binpacking:
  - Let's take a set of weights $s_1, \ldots, s_n$, and a corresponding graph. Let's say that the maximum matching number is m, while the optimal binpacking uses $n-k$ bins, where k is the number of bins that have 2 items in them.
  - We can see that $k \leq m$, since if we take an optimal binpacking, of these weights and look at the bins that have two items in them, each of these will correspond to an independent edge in the graph. (Since the two items fit into a bin, they have an edge running between them and each item can only be placed to a single bin, so these edges are independent, they don't have common vertices = weights.) This means that we have found an independent set of k edges, so the maximum matching is at least k.
  - Similarly, we can see that $m \leq k$, since if we take a maximum matching, we can iterate over the edges in this matching and put the weights corresponding the edges' vertices into one bin per edge. They will fit, since an edge can only run between vertices for which the corresponding weights' sum is ≤ 1.0. Finally, we will put the remaining weights into separate bins, resulting in $n - m$ bins in total (m bins containing 2 weights and $n - 2m$ bins for the rest of the weights). This means that if the optimal binpacking can only be less than or equal to this, so if the optimal uses $n - k$ bins, then $n - k \leq n - m$, since this is a correct binpacking. Resulting in $k \geq m$.
  - Since we have just seen that both $m \leq k$ and $k \leq m$ is true, this means that $m = k$, so the optimal binpacking corresponds to a maximum matching.
- So to solve this specific $BINPACKING$ problem, we first generate the corresponding graph. This can be done in $O(n^2)$ time, since we check all pairs of weight, whether they could fit in the same bin or not and add an edge when they fit into the adjacency matrix of the graph.
- Then, we can find a maximum matching in this graph using the Hungarian method in polynomial time.
- Finally we iterate the edges of this matching and place the corresponding vertices into one bin per edge, and the remaining weights into separate bins. This can be done in $O(n^2 + n)$ time, so all of this is still polynomial.

- We just described a polynomial time algorithm that finds the optimal solution to the $BINPACKING$ problem, in the special case of all weights being strictly greater than $\frac{1}{3}$, which means that this special case of the $BINPACKING$ problem is in P.
- Be aware: The general case of the $BINPACKING$ problem is still NP-complete! So the fact that the weights are all bigger than $\frac{1}{3}$ made the problem a lot more easier!

10.5 Session 10, Exercise 5

Exercise

We are moving on the squares of an $n \times n$ chessboard from the lower left corner to the upper right corner so that in one step we can move to the neighboring square to the right or up. However, there are some squares given that are "forbidden", that is we cannot step on them. Give a dynamic programming procedure that determines how many different ways can we get to the upper right corner.

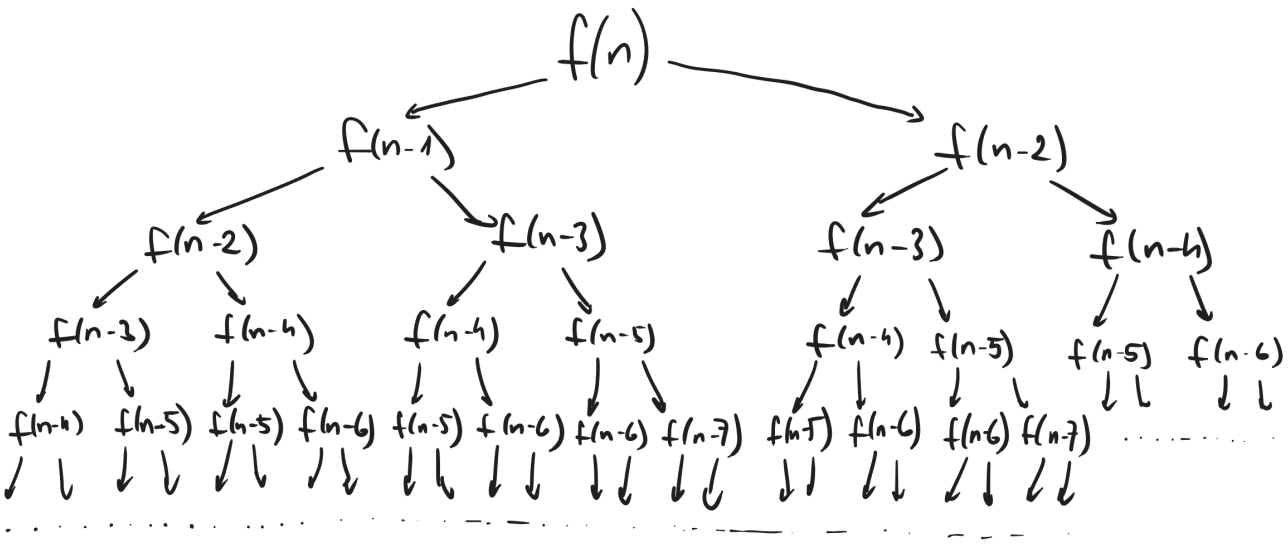
Solution

What is Dynamic Programming?

- Dynamic programming is just smart recursion. I like to give the example of calculating Fibonacci-numbers to explain what this means.
- Fibonacci-numbers are defined recursively. The first and second Fibonacci-numbers are 1 and 1. If we denote the ith Fibonacci-number with $F[i]$, then $F[1] = 1, F[2] = 1$. After that, each Fibonacci-number is the sum of the previous two: $F[i] = F[i-1] + F[i-2]$, for $i \geq 2$.
- If we wrote a program to calculate these numbers, we could write a recursive function like so:

```
int f(int index) {
  if(index == 1 || index == 2) {
    return 1;
  }
  return f(index-1) + f(index-2);
}
```

- Let's take a look at the function calls for $f(n)$:



- We calculate the $(n-2)$th Fibonacci number two times, the $(n-3)$th 3 times, the $(n-4)$th 5 times, the $(n-5)$th 7 times (1 of which is not plotted), and so on... Quite wasteful isn't it?
- Bonus points if you noticed, that the number of times we recalculate the $(n-i)$th Fibonacci number is actually the ith Fibonacci number. Double bonus points if you can prove this. :)
- Wouldn't it be better if we just remembered our calculations and whenever we needed them again, we could just pull the information out of memory? Let's create an array, where we store the Fibonacci numbers we have already calculated!

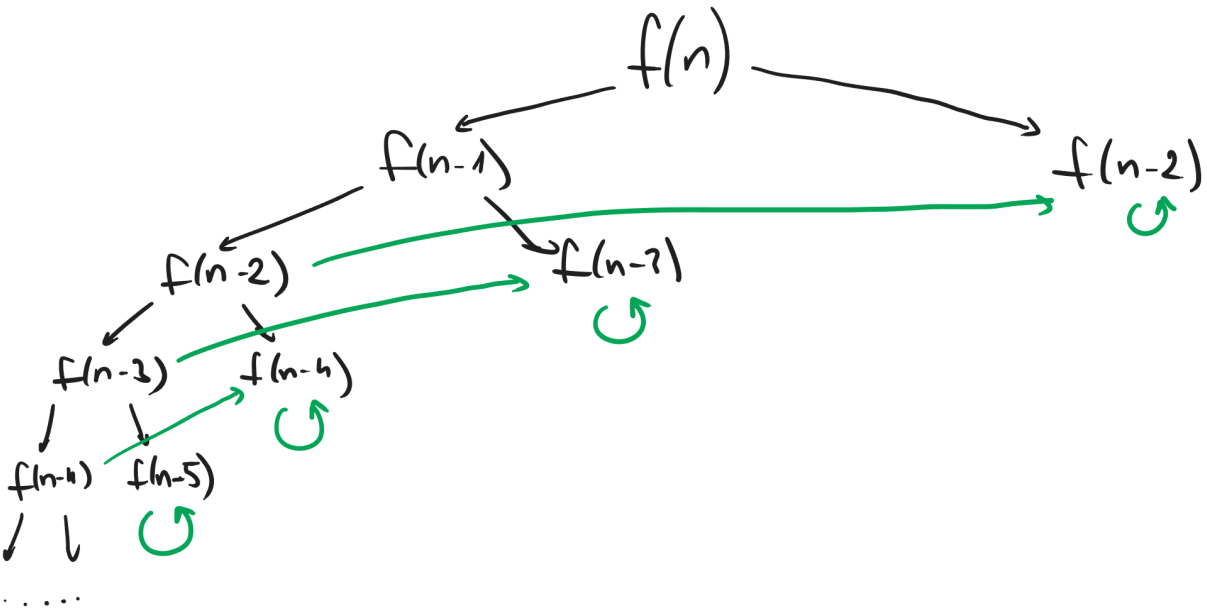
```
#include <vector>

```

```
int N = 100;
// Initializes vector of size N of 0's.
// (N+1 so indexing is 1-based for convenience.)
vector<int> T = new vector<int>(N+1);

// Base cases.
T[1] = 1;
T[2] = 1;

int f(int index) {
  if (T[index]) { // If T[index] is already calculated, we return it.
    return T[index];
  }
  else { // Otherwise T[index] = 0, so not yet calculated: calculate it, store it, then return it.
    T[index] = f(index-1) + f(index-2);
    return T[index];
  }
}
```

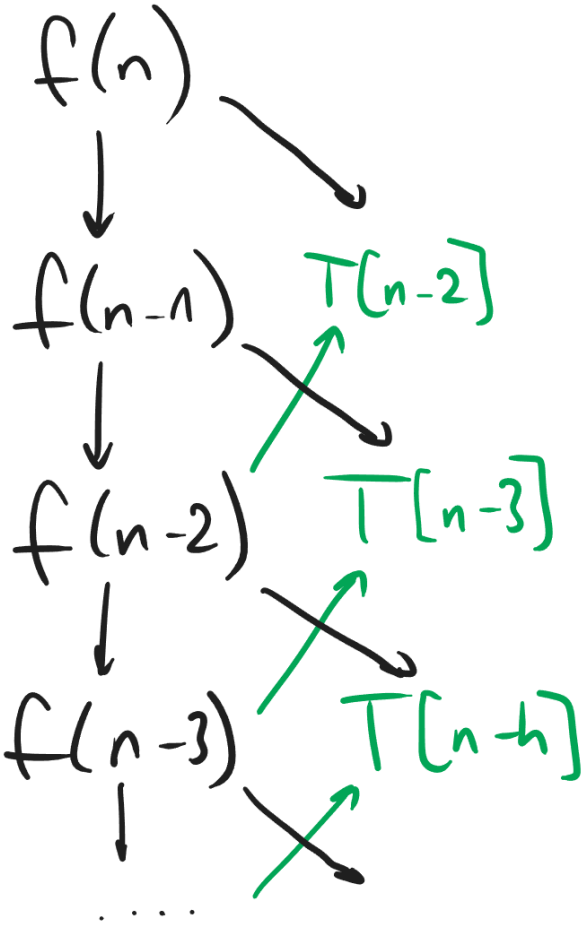
- To simplify this a bit:

```
#include <vector>

int N = 100;
vector<int> T = new vector<int>(N+1);

T[1] = 1;
T[2] = 1;

int f(int index) {
  if (!T[index]) {
    T[index] = f(index-1) + f(index-2);
  }
  return T[i];
}
```

- If we look back on the branching of the calculation of $f(n)$, we just cut off all right-side branches:



- So essetially, our recursion turned into a single-branching, insted of a double-branching:



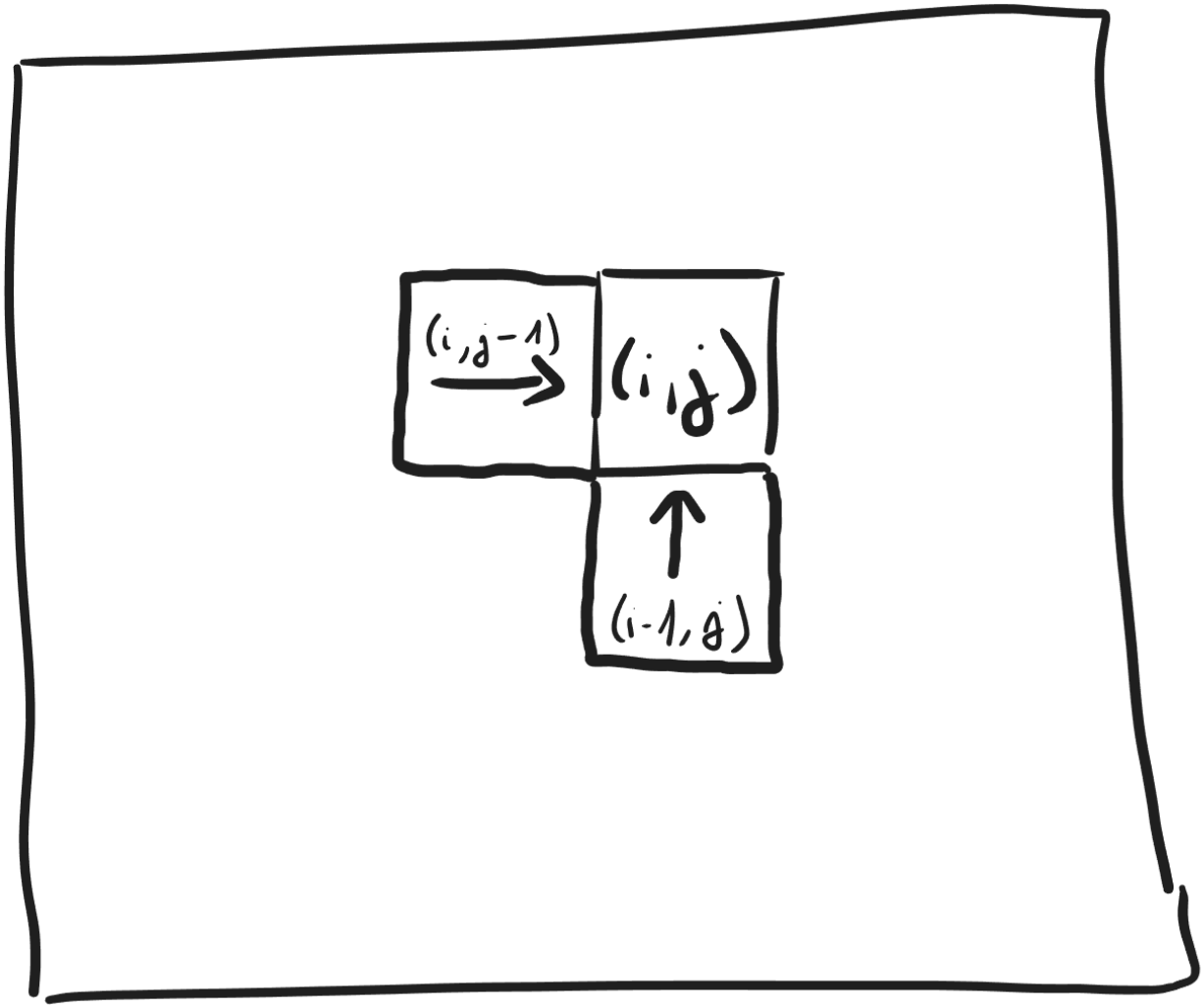
- What we have done here is called **memoization** (and the missing **r** is not a typo).
- For example in Python, we can add memoization to any function without writing additional code, by using a decorator, called @cache.
- While memoization allowed us to cut the $O(2^n)$ runtime into $O(n)$, so this runs fast, there is still one problem that remains: we are still abusing and wasting the stack.
- Every time a function is called, a block of memory is reserved on the stack, for things like storing the internal variables, the return value and the instruction pointer to return to. The stack size is around $1-2$ MB at most. Some programming languages, like Python limit recursion depth to around 1000, while other languages like C++ let you exhaust it at your own will, resulting in a stack overflow.
- This means that using this approach, we won't be able to calculate Fibonacci-numbers with an index greater than 1000.
- However, instead of calculating top-down, starting from the nth number, back to 1 using recursion, we could instead calculate bottom-up, by filling out the T array starting from the index 1 all the way to n, like this:

```
#include <vector>

int N = 100;
vector<int> T = new vector<int>(N+1);

T[1] = 1;
T[2] = 1;

for(int index = 3; index <= n; ++index) {
    T[index] = T[index-1] + T[index-2]; // The previous indexes have already been calculated.
}

// Result is in T[100].
```

- We no longer need a recursive function, since we intentionally fill out the T array in an order, where the values needed to calculate the current value are already available.
- So Dynamic Programming = Recursion + Memoization (runtime optimization) + a Bottom-up approach (memory usage optimization).
- The difficulty with dynamic programming lies in two things:
  - Recognizing that the problem in front of us is solveable using dynamic programming.
  - Figuring out the recursive formula for the solution.
- In the following part, I'll explain how I usually solve dynamic programming tasks (in a competitive programming setting) using something for lack of a better phrase I called a **key statement**. When I'm reading an exercise, I usually notice this key statement from the description first, and that's when I realize it's a DP problem.

Figuring out the solution

- Let's go back to the current exercise about the chess board:
- Every time we solve a dynamic programming problem, the first step is to figure out the **key statement** for this problem, which will reveal the recursive formula for the problem.
- In general, we derive the key statement from the question of the exercise. The question here is "How can we reach the upper right corner?".
- Then, we make an observation on the different ways this can happen. Here, we can come to this square from the left hand side, or the lower square.
- So in this case, they **key statement** is: "To reach the upper right corner, we can come from the left hand side square or the lower square."
- This is the hardest part of a dynamic programming exercise, getting this statement correctly. If you get this statement correct, everything else will come naturally. In general, the statement will be an **if this, then that, else that** type of sentence, that will "partition" the solution to the problem into a few distinct cases.
- To formalize the statement above a bit, let's introduce a notation for the board squares. Let $B[i, j]$ be the square in the ith row and jth column.
- So the key statement using this notation: **To reach** $B[n, n]$**, we could come from** $B[n, n-1]$ **or** $B[n-1, n]$**.**
- So if we stored he number of ways in which we can reach a square in a 2D array, T, so the square in the ith row and jth column is in $T[i, j]$, we can turn the key statement into the following recursive formula:
- $T[n, n] = T[n, n-1] + T[n-1, n]$. The number of ways to reach $T[n, n]$ is $T[n, n-1]$ plus $T[n-1, n]$, since we either could come from the square $B[n, n-1]$ or $B[n-1, n]$.
- Or for any i, j square, $T[i, j] = T[i, j-1] + T[i-1, j]$.

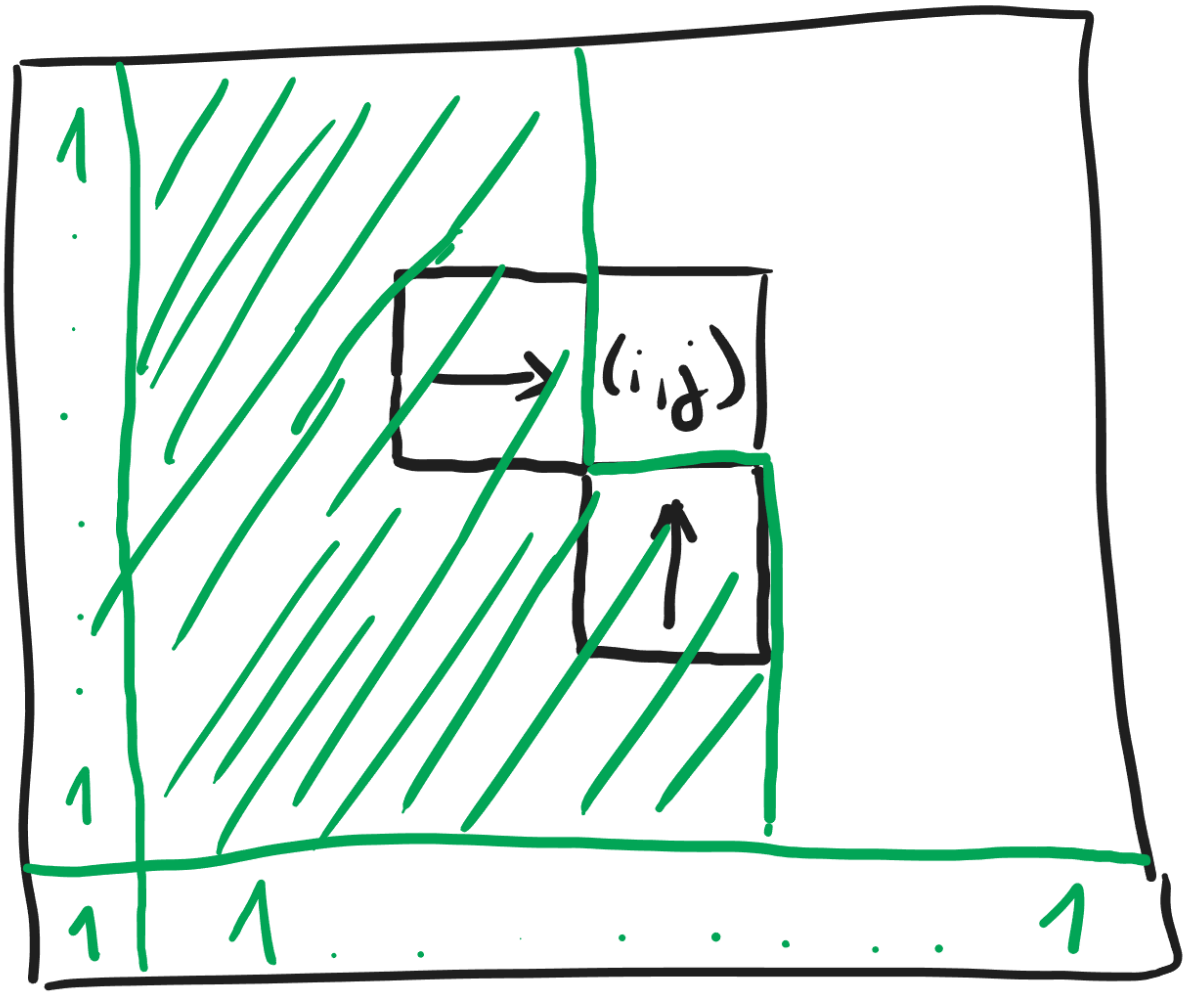


- This does not yet take into account un-steppable cells. If $T[i, j]$ is unsteppable, we would like to zero out its value.
- Let's modify the previously introduced 2D board $B[i, j]$, and say that the value of $B[i, j]$ will be 1 if that square is steppable and 0 if not.
- Now we can simply multiply the result with $B[i, j]$, which will make the cell value zero if it's not steppable.
- $T[i, j] = B[i, j] \cdot (T[i, j - 1] + T[i - 1, j])$.
- What's left for us to do is:
  - The above recursive formula relies on previous square values. This needs to start somewhere, so we need to give the base cases for the formula.
  - We must give the order in which T is filled out, so every time a cell is filled, we already have the needed previous values computed.
  - We must say where the solution is in T. While in this exercise, the solution will be in $T[n, n]$, in other exercises it could be a maximum of all values in T, or other possibilities. See Exam: 2022. 05. 30., Exercise 7 for an example like this.
  - Say what is the runtime of the resulting algorithm. Most exercises will ask us to give a specific $O(\dots)$ to fulfill, so we need to show, that our algorithm fulfills this criteria.

I like to summarize all of this into 6 steps, and this is what I would write down in the exam:

- **Step 1**: Define the T DP table. What does $T[i, j]$ mean?
  - T is a 2 dimensional table, of size $(n \times n)$.
  - $T[i, j]$ contains the number of ways the square in the ith row and jth column is reachable from the bottom left square.

- **Step 2**: Give the generic recursive formula. Explain why it is correct.
  - $T[i,j] = B[i,j] \cdot (T[i,j-1] + T[i-1,j])$, for $i,j > 1$.
  - To step into the (i,j) square, we could come from $(i,j-1)$ or $(i-1,j)$, so we add up the ways we could reach these cells.
  - Then we multiply by $B[i,j]$, which is 0 if $B[i,j]$ is not steppable, and 1 if it is steppable. This means that non-steppable squares can be reached in 0 ways, as intended.
  - The formula only works if we are not in the first row or column, since in these cases, there is no left hand side or lower square. This is handled by the base cases instead.
- **Step 3**: Give the base cases for the recursive formula (how to start filling out the array):
  - $T[j,1] = 1$ and $T[1,j] = 1$ for all $1 \leq j \leq n$.
  - Explanation: $T[1,1] = 1$, since we start there, then this value is copied in the first row and column, since there is a single path to reach the squares here.
- **Step 4**: The order in which T must be filled.
  - We can fill out T by iterating the columns first, from 1 to n, and inside each column iterate the rows from 1 to n.
  - This way the left and lower cells will already be computed for the current cell to use their value:



- **Step 5**: Where is the solution in T?
  - In $T[n,n]$, since that is how many ways we can reach the upper right square.
- **Step 6**: What is the runtime of the algorithm?
  - We fill out an n^2 sized table. In every step, we do an $O(1)$ calculation: we check on 2 other values of the table, add them, multiply by $B[i,j]$. So in total, we do this in $O(n^2)$.

10.6 Session 10, Exercise 6

Exercise

Some rungs of a ladder of f rungs are so rotten that if someone steps on them, then they break. Fortunately, it known which rungs are so bad. We can step over three rungs at most in one step. Give a dynamic programming algorithm that determines

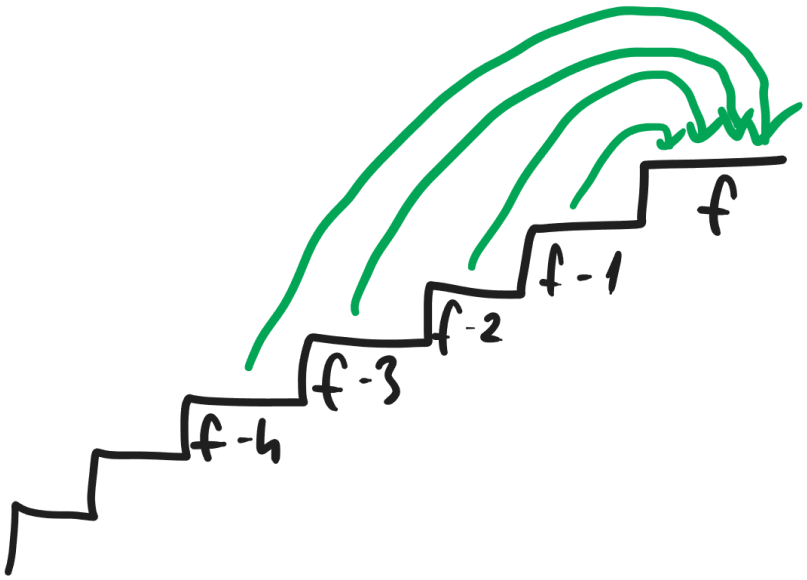
a) whether we can climb up on the top of the ladder from the bottom,

b) how many ways can we reach the top rung?

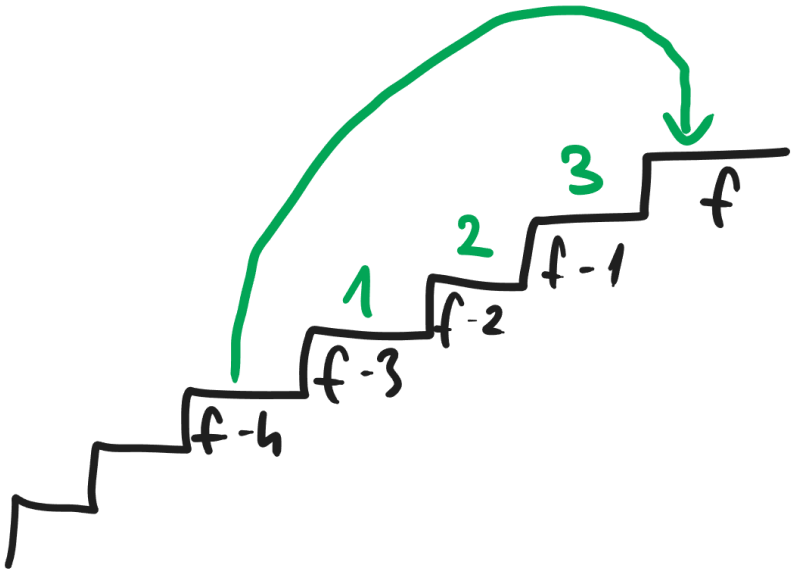
(It is assumed that one can step on the top rung.) What is the running time of your algorithm?

Solution

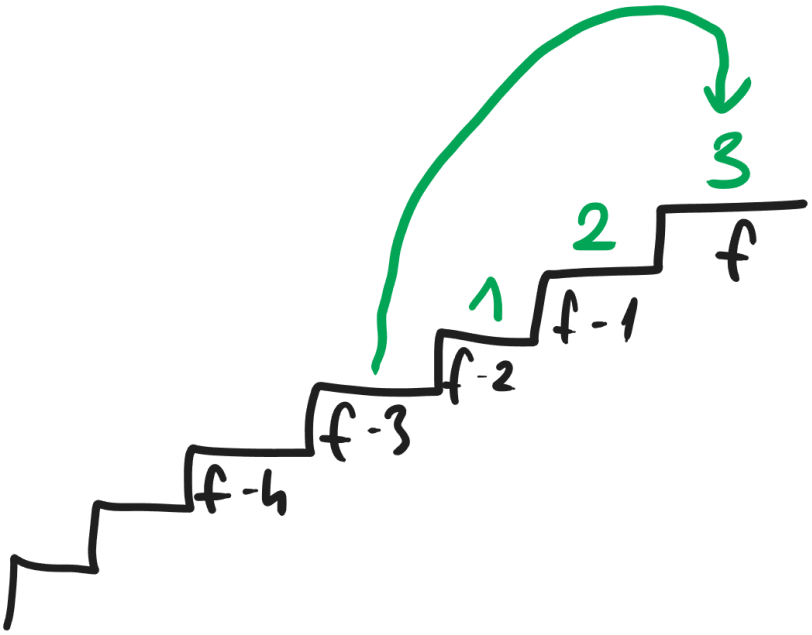
- The exercise is asking if we can reach the fth step. To reach the fth step we could come from the $(f-1)$, $(f-2)$, $(f-3)$ or $(f-4)$th steps, since we can step over 3 steps.



- Side note: I would argue that stepping over 3 steps looks like this:



- But many of you have argued that it looks like this instead:



- I think it depends on how you interpret the word 'over', if you interpret it to mean 'stepping above the steps without touching them', then it means the first drawing, while if you interpret it as 'stepping forwards/upwards that many steps', then it means the second drawing. In either case, it doesn't really matter here and if you're not sure during the exam, always ask the instructors present for clarification.

Back to the exercise:

- The exercise is asking if we can reach the fth step. To reach the fth step we could come from the $(f-1)$, $(f-2)$, $(f-3)$ or $(f-4)$th steps. This is our **key statement**.
- If we store if we can reach the ith step from the bottom (let's say that's the 0th step) in a 1D array A indexed from 0 to f ($\rightarrow$ **Step 1**: Define A, what $A[i]$ means?), then $A[f] = A[f-1] \vee A[f-2] \vee A[f-3] \vee A[f-4]$, where $\vee$ is the logical or symbol.
- Or, in general for any ith step: $A[i] = A[i-1] \vee A[i-2] \vee A[i-3] \vee A[i-4]$.

- And to deal with broken steps, let's say that $B[i]$ is $True$ if the ith step is steppable and $False$ if not. Now we can say that $A[i]$ must be steppable **and** we must be able to step on any of the previous steps:
- $A[i] = B[i] \wedge (A[i-1] \vee A[i-2] \vee A[i-3] \vee A[i-4])$ ($\rightarrow$ **Step 2**: Generic recursive formula.), where $\wedge$ is the logical and symbol.
- The base case is $A[0] = True$, since we are standing on the floor at the bottom.
- This recursive formula breaks down the moment $i < 4$, we will start to index A with negative numbers, whenever we indexed with a negative number we should remove that $A[i]$ from the or.
- Let's just say that $A[i] = False$, for $i < 0$, because I'm lazy and $A[0] = True$. ($\rightarrow$ **Step 3**: Base cases.)
- We can fill out the A from index 0 to f, so the solutions for the previous indexes are already available for us when calculating the current value. ($\rightarrow$ **Step 4**: Order in which A is filled.)
- The solution is in $A[f]$, it is $True$ if we can reach the fth step and $False$ if not. ($\rightarrow$ **Step 5**: Where is the solution?)
- The runtime complexity: we fill out an array of size $f + 1$, and in each step we reach back to 4 other values plus a value in B and do some logical operators between them. So each step can be done in constant time, so the runtime complexity is $O(n)$ in total. ($\rightarrow$ **Step 6**: Runtime complexity.)

And the second task:

- The backbone of the algorithm is going to be the same, we only need to slightly tweak our calculations: switch from boolean logic to integers.
- Usually to go from boolean logic to integers: switch and to multiplication, switch or to addition, and switch $False$ to 0 and $True$ to 1.
- So change this:

$$A[i] = B[i] \wedge (A[i-1] \vee A[i-2] \vee A[i-3] \vee A[i-4])$$

- Into this:

$$A[i] = B[i] \cdot (A[i-1] + A[i-2] + A[i-3] + A[i-4])$$

- So now $B[i]$ should be 1 if the ith step is steppable and 0 if not, so there will be 0 ways to reach unsteppable steps as intended.
- And $A[i]$ will contain how many ways the ith step is reachable.
- $A[i] = 0$ for $i < 0$ and $A[0] = 1$.
- The rest is the same.

10.7 Session 10, Exercise 7

Exercise

An $(n \times n)$ table contains integer numbers. We want to go from the lower left corner to the upper right corner so that in one step we can move to the neighboring entry to the right or up. The numbers visited during the tour must come in increasing order. Give an $O(n^2)$ running time algorithm that determines

a) how many such tours exist,

b) what is the largest weight of such a tour if the weight of a tour is the sum of the entries visited.

Solution

The first task:

- This is very similar to Session 10, Exercise 5, with the only twist being that sometimes you cannot step from the left or lower square, since the numbers are not in the correct order, but all squares themselves are steppable.
- Let's just say that the integers in the table are in the $A[i, j]$ 2D table.
- So the rule to calculate $T[i, j]$ becomes:

$$T[i, j] = (A[i, j-1] < A[i, j]) \cdot T[i, j-1] + (A[i-1, j] < A[i, j]) \cdot T[i-1, j]$$

- And the $<$ operator shall return 0 instead of $False$ and 1 instead of $True$, for convenience.
- So when we multiply with $(A[i, j-1] < A[i, j])$ we are essentially saying that we should only add $T[i, j-1]$ to the result if the numbers $A[i, j-1]$ and $A[i, j]$ are in increasing order, so we are allowed to step in this order. Similarly for the other direction.
- The base cases, order to fill, where is the solution, runtime are all the same as in Session 10, Exercise 5.

The second task:

- Instead of number of ways, now we are looking for maximum weight (maximum sum of visited numbers).
- So now the meaning of $A[i, j]$ is the maximum path weight possible to reach that cell.
- First of all, this means that the plus here should be a maximization instead:

$$T[i, j] = (A[i, j-1] < A[i, j]) \cdot T[i, j-1] + (A[i-1, j] < A[i, j]) \cdot T[i-1, j]$$

$$T[i, j] = \max((A[i, j-1] < A[i, j]) \cdot T[i, j-1], (A[i-1, j] < A[i, j]) \cdot T[i-1, j])$$

- And assuming that $T[i, j-1]$ and $T[i-1, j]$ already contains the maximum path weight to those cells, we must add the $A[i, j]$ weight of the cell we are stepping onto now:

$$T[i, j] = \max((A[i, j-1] < A[i, j]) \cdot (T[i, j-1] + A[i, j]), (A[i-1, j] < A[i, j]) \cdot (T[i-1, j] + A[i, j]))$$

- Be careful! It is very tempting to do this instead:

$$T[i, j] = \max((A[i, j-1] < A[i, j]) \cdot T[i, j-1], (A[i-1, j] < A[i, j]) \cdot T[i-1, j]) + \cancel{A[i, j]}$$

- But this is problematic. If neither $A[i, j-1] < A[i, j]$ nor $A[i-1, j] < A[i, j]$ is true, we are supposed to get a 0 from this equation, which happens correctly in the previous equation but not in this latter one. So use the previous one.
- And now the base cases will be $T[1, 1] = A[1, 1]$, and then the first row and column, for $j > 1$ (basically just ignore the part of the *max* from the generic recursive formula that indexes into non-exsistent cells):

$$\begin{aligned} T[1, j] =& (A[1, j-1] < A[1, j]) \cdot (T[1, j-1] + A[1, j]) \\ T[j, 1] =& (A[j-1, 1] < A[j, 1]) \cdot (T[j-1, 1] + A[j, 1]) \end{aligned}$$

- Everything else (order to fill, where is the solution, runtime) is the same as previously.

10.8 Session 10, Exercise 7

Exercise

An $(n \times n)$ table contains integer numbers. We want to go from the lower left corner to the upper right corner so that in one step we can move to the neighboring entry to the right or up. The numbers visited during the tour must come in increasing order. Give an $O(n^2)$ running time algorithm that determines

a) how many such tours exist,

b) what is the largest weight of such a tour if the weight of a tour is the sum of the entries visited.

Solution

The first task:

- This is very similar to Session 10, Exercise 5, with the only twist being that sometimes you cannot step from the left or lower square, since the numbers are not in the correct order, but all squares themselves are steppable.
- Let's just say that the integers in the table are in the $A[i, j]$ 2D table.
- So the rule to calculate $T[i, j]$ becomes:

$$T[i, j] = (A[i, j-1] < A[i, j]) \cdot T[i, j-1] + (A[i-1, j] < A[i, j]) \cdot T[i-1, j]$$

- And the $<$ operator shall return 0 instead of $False$ and 1 instead of $True$, for convenience.
- So when we multiply with $(A[i, j-1] < A[i, j])$ we are essentially saying that we should only add $T[i, j-1]$ to the result if the numbers $A[i, j-1]$ and $A[i, j]$ are in increasing order, so we are allowed to step in this order. Similarly for the other direction.
- The base cases, order to fill, where is the solution, runtime are all the same as in Session 10, Exercise 5.

The second task:

- Instead of number of ways, now we are looking for maximum weight (maximum sum of visited numbers).
- So now the meaning of $A[i, j]$ is the maximum path weight possible to reach that cell.
- First of all, this means that the plus here should be a maximization instead:

$$T[i, j] = (A[i, j-1] < A[i, j]) \cdot T[i, j-1] + (A[i-1, j] < A[i, j]) \cdot T[i-1, j]$$

$$T[i, j] = \max((A[i, j-1] < A[i, j]) \cdot T[i, j-1], (A[i-1, j] < A[i, j]) \cdot T[i-1, j])$$

- And assuming that $T[i, j-1]$ and $T[i-1, j]$ already contains the maximum path weight to those cells, we must add the $A[i, j]$ weight of the cell we are stepping onto now:

$$T[i, j] = \max((A[i, j-1] < A[i, j]) \cdot (T[i, j-1] + A[i, j]), (A[i-1, j] < A[i, j]) \cdot (T[i-1, j] + A[i, j]))$$

- Be careful! It is very tempting to do this instead:

$$T[i, j] = \max((A[i, j-1] < A[i, j]) \cdot T[i, j-1], (A[i-1, j] < A[i, j]) \cdot T[i-1, j]) + \cancel{A[i, j]}$$

- But this is problematic. If neither $A[i, j-1] < A[i, j]$ nor $A[i-1, j] < A[i, j]$ is true, we are supposed to get a 0 from this equation, which happens correctly in the previous equation but not in this latter one. So use the previous one.
- And now the base cases will be $T[1, 1] = A[1, 1]$, and then the first row and column, for $j > 1$ (basically just ignore the part of the *max* from the generic recursive formula that indexes into non-exsistent cells):

$$\begin{aligned} T[1, j] =& (A[1, j-1] < A[1, j]) \cdot (T[1, j-1] + A[1, j]) \\ T[j, 1] =& (A[j-1, 1] < A[j, 1]) \cdot (T[j-1, 1] + A[j, 1]) \end{aligned}$$

- Everything else (order to fill, where is the solution, runtime) is the same as previously.

10.9 Session 10, Exercise 8

Exercise

Let $s_1, s_2, \ldots s_n$ and $t_1, t_2, \ldots t_m$ be two words over alphabet $\{0, 1\}$. Entry $A[i, j]$ of an $n \times m$ array A should contain the largest number k such that the last k characters of words $s_1 s_2 \ldots s_i$ and $t_1 t_2 \ldots t_j$ agree (so basically we go from s_i and t_j backwards and count how many characters are the same in the two words and write that number into $A[i, j]$). Give an algorithm that fills array A up in $O(nm)$ steps.

Solution

- The 2D table is kind of already defined for us here ($\rightarrow$ **Step 1**: What is the DP table?), but how do we step in it?
- When we are at $A[i, j]$ at first we need to check whether $s_i \stackrel{?}{=} t_j$, if no we write 0, if yes we counted 1 so far,
- and then we would keep going with $s_{i-1} \stackrel{?}{=} t_{j-1}$ and $s_{i-2} \stackrel{?}{=} t_{j-2}$ and so on, however there is no need to do this, since this calculation was already done when we did $A[i-1, j-1]$.
- So we can just use the value stored there $+1$, for the additional $s_i = t_j$.
- The **key statement** here would be something like: **Either the last characters s_i and t_j are the same, or they are not.** If they are the same, then the value is $A[i, j] = A[i-1, j-1] + 1$, if not, then $A[i, j] = 0$.
- So the full equation is:

$$A[i, j] = (s_i \stackrel{?}{=} t_j)(A[i-1, j-1] + 1)$$

- Again, just say that $\stackrel{?}{=}$ returns 0 if the equality is not true, and 1 if it is true.
- ($\rightarrow$ **Step 2**: Generic recursive formula.)
- Now,the base case will be wherever we index out of the table with the generic formula, so the first row and the first column of A.
- Since in the first row, $A[1, j]$, we can only have a single character from s, s_1, we are only checking whether that is equal to t_j, we cannot go backwards in s, this means that $A[1, j] = (s_1 \stackrel{?}{=} t_j)$, $1 \leq j \leq m$.
- Similarly $A[i, 1] = (s_i \stackrel{?}{=} t_1)$, $1 \leq i \leq n$.
- ($\rightarrow$ **Step 3**: Base cases.)
- We fill out the array from the first row to the nth and in each row from the first column to the mth, so the previous values are always present when we need to use them.
- ($\rightarrow$ **Step 4**: Order to fill DP table.)
- The solution in this case is the entire A table, since that is what the exercise asked for. ($\rightarrow$ **Step 5**: Where is the solution?)
- We fill out the $(n \times m)$ table in nm steps, each step looking up a single other value of the table and comparing two characters from the input words, which is $O(1)$ runtime, so the total runtime is indeed $O(nm)$. ($\rightarrow$ **Step 6**: Runtime complexity.)

10.10 Session 10, Exercise 9

Exercise

We want to find the longest common consecutive subsequence of two character sequences $a_1a_2 \dots a_n$ and $b_1b_2 \dots b_m$. That is, we need to find pair of indices $1 \leq i \leq n$ and $1 \leq j \leq m$ such that $a_{i+1} = b_{j+1}, a_{i+2} = b_{j+2}, \dots, a_{i+t} = b_{j+t}$ holds for the largest possible t. Give an $O(mn)$ running time algorithm solving this problem.

Solution

- Notice how this is the same problem as in Session 10, Exercise 8, the only difference is that we are looking for the index pair, for which the maximum length of common subsequence is achieved.
- So at first, we do the same algorithm and calculate the A matrix for the input words a and b.
- Then, we find the cell with the maximum value, for example $A[i, j] = k$. However, this k means that starting from a_i and b_j, **backwards** up to k steps the strings are equal. And the exercise is asking for the starting indexes, not the ending. So the answer is not i and j, but $i - (k - 1)$ and $j - (k - 1)$.
- As the saying goes, "There are two hard things in computer science: cache invalidation, naming things, and off-by-one errors." To quickly chek that our indexing is correct, I like to do the following:
- The starting index is $i - (k - 1)$, the end is i and this is correct if there are k characters between these two.
- I remember the following formula: Between numbers a and b, (counting a and b too), there are exactly $b - a + 1$ numbers.
- So between $i-(k-1) = i-k+1$ and i (counting both) there are exactly $i-(i-k+1)+1 = i-i+k-1+1 = k$ numbers, so this is correct.

10.11 Session 10, Exercise 10

Exercise

We are given two $0-1$ sequences, one of length n and one of length m. The sequences are $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_m$. Based on them an array T is filled up as follows:

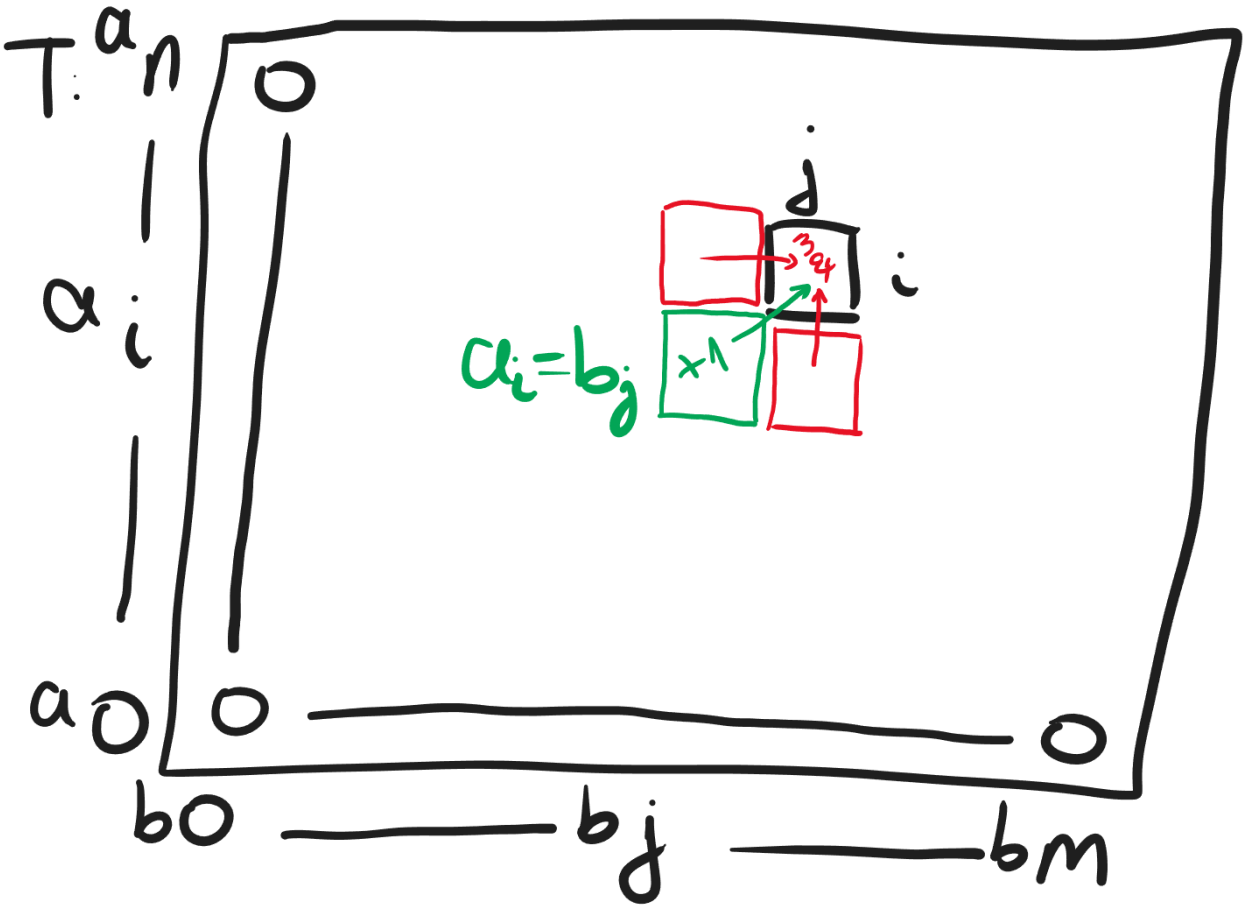
- If $0 \leq i \leq n$, then $T[i, 0] = 0$.
- If $0 \leq j \leq m$, then $T[0, j] = 0$.
- If $1 \leq i \leq n$ and $1 \leq j \leq m$, then

$$T[i,j] = \begin{cases} T[i-1, j-1] + 1 & \text{if } a_i = b_j \\ max(T[i, j-1], T[i-1, j]) & \text{if } a_i \neq b_j \end{cases}$$

What is the meaning of $T[i, j]$? What property of the two sequences is given by value $T[n, m]$?

Solution

- Let's visualise this!



- So we zero out all borders of the array, which is an alternative to having to give base cases: when we would be indexing out of the array, there is a border of zeroes that protects us and just returns a neutral element instead.
- Then, in the formula, if $a_i = b_j$, that is very similar to the previous exercise, Session 10, Exercise 9: we count the matching characters with the $+1$ and add how many characters matched previously, with $T[i-1, j-1]$.

- However, there is an alternative case: if the characters don't match, we take $T[i, j-1]$ or $T[i-1, j]$, whichever is bigger. This means that we are allowed to skip over non-matching characters from either side: the matching sequence does not have to be from **consecutive** characters, but could be any **non-consecutive** subsequence from a and b.
- So $T[n, m]$ contains the length of the longest, not necessarily consecutive subsequences that are the same from a and b. And $T[i, j]$ does the same, but for he prefix $a_1 \dots a_i$ and $b_1 \dots b_j$.

10.12 Session 10, Exercise 11

Exercise

A directed graph G of n vertices and e edges is given by its adjacency list. Give an $O(n + e)$ running time algorithm that determines for each vertex v the number of those paths that:

a) go from a fixed source vertex to v,

b) go from v to a fixed end vertex.

Solution

TODO

11 May 11th (Session 11): Sorting, Binary Search Trees

11.1 Session 11, Exercise 1

Exercise

Consider the version of problem $SUBSETSUM$ where $a_1, a_2, \ldots, a_n$ and b are integers, such that $0 < a_i < n^2$ for all $1 < leqi \leq n$. The question is that does there exist a subset $I \subseteq \{1, \ldots n\}$, such that $\sum_{i \in I} a_i = b$. Prove that this version of the problem is in P.

Solution

- So if all weights are limited to n^2 (where n is the number of weights), then the problem should be easier than the original. Why is that? Why is the n^2 limit helpful to us?
- First of all, if a single number is limited to n^2, then the total sum is limited to at most n^3.
- If you remember from the $KNAPSACK$ solver (which is very similar to $BINPACKING$), it was a DP algorithm that created a table of size n (corresponding to the number of items) times b (corresponding to the weight limit there). But b is was not polynomial relative to the input size, which is why this algorithm was exponential (the input size of the number b is $O(log_2(b))$).
- This problem is now solved, by b being limited by n^3, so the DP table size is limited to $O(n \cdot n^3)$. (If the input b is larger than n^3, we return "NOT POSSIBLE", without making a table.)
- So this tells us that we should create a solver, similar to Knapsack, that in this case will run in polynomial time.
- The **key statement** for this DP solver is the following: **To reach the sum b, we either use the number a_n or not use it.**
- Since we are looking for a recursive formula, it makes sense to reduce to the smaller size problem of $\{a_1, \ldots, a_{n-1}\}$ and either b or $b - a_n$, depending on whether we wanted to use a_n or not. So usually the key statement will look at and remove the last number from the input, not the first.
- If we use a_n, this means that we need to get the sum of $b - a_n$ using the values $\{a_1, \ldots, a_{n-1}\}$, while if we don't use a_n, we need to get the sum b using the values $\{a_1, \ldots, a_{n-1}\}$.
- We need to keep track of both the current sum and the index i of the last a_i number, so we define $T[i, c]$ to mean: can we achieve the sum c using values $\{a_1, \ldots a_i\}$? ($\rightarrow$ **Step 1**: Define the DP table. What does $T[i, c]$ mean?)
- The recursive formula is the following:

$$T[i, c] = T[i-1, c-a_i] \vee T[i-1, c]$$

- If either sum is possible, then $T[i, c]$ is also possible. $\vee$ means the boolean or operator. ($\rightarrow$ **Step 2**: Generic recursive formula.)
- The base cases is where we index out of the table T. This can happen if $i = 1$, and also there are a lot of negative indexes in the second dimension.
- Let's just say, that if in $T[i, j]$, the second index is negative then we know that whatever sum we are trying to reach is not possible like that, so it should be a $False$ value. So define $T[i, j] = False$ if $j < 0$.
- Then for $T[1.j]$ we are trying to get a sum of exactly j, using only a_1. This is only possible if $j = a_1$. So $T[1, a_1] = True$, and false for any other j. ($\rightarrow$ **Step 3**: Base cases.)
- The order in which we must fill out T: We want to go row-by-row starting from smaller to larger indexes, since the recursive formula is using the previous row's values. Then inside a row, we go from smaller to larger index columns. There is no need to fill T further than the bth column, since we don't need those values for the sum of b. ($\rightarrow$ **Step 4**: Filling order.)
- The solution is in $T[n, b]$, meaning if the sum of b is possible using any subset of all of the values.

- We fill this table in $O(n \cdot n^3)$ time, since the size of the table is $n \cdot n^3$ and we do $O(1)$ operations for each cell. ($\rightarrow$ **Step 6**: Runtime complexity.)
- This is a polynomial algorithm for this specific case of the $BINPACKING$ problem, so this proves that this special case is in P. (The general case is still NP-complete though!)

11.2 Session 11, Exercise 2

Exercise

Rational numbers are stored in array $A[1:n]$. Determine in $O(n \log n)$ steps

a) those elements that occur more than once,

b) the most frequent values (that is those numbers that no other number occurs more times than them).

Solution

- $O(n \log n)$ means we are allowed to sort, which is useful to detect duplicate elements in an array.
- So let's sort the array using for example merge sort, which runs in $O(n \log n)$ time.
- Then, the duplicate values will be next to each other. We can just iterate the array in $O(n)$ time and count the number of duplicates as we go.
- For the first task, whenever we counted at least 2 from a number, we print it out immediately.
- For the second task, we keep track of the maximum count seen and in the first loop we don't print anything. Then we do this loop one more time, now we know the count we are looking for and whenever we reach this count with a specific number, we print that number out.
- In total we did $O(n \log n + 2n) = O(n \log n)$ steps, as requested by the task.

11.3 Session 11, Exercise 3

Exercise

IT is known for an array $A[1:n]$ that $A[1] \neq A[n]$. Give an algorithm using $O(\log n)$ comparisons that finds an index i such that $A[i] \neq A[i+1]$.

Solution

- $O(\log n)$ means we can't even read the entire array once through! What we can do however is binary search.
- The items are not in a sorted order, however the property we are looking for does not require that.
- Let's start with two poiners, s and e (meaning start and end). In the beginning $s = 1$ and $e = n$. We know that $A[1] \neq A[n]$, so $A[s] \neq A[e]$.
- Next, we will move one of the pointers closer to the other, while this property will still hold true!
- This is done by indexing in the middle: $m = \lfloor \frac{s+e}{2} \rfloor$. Let's compare $A[m]$ to $A[s]$ and $A[e]$. $A[m]$ cannot be equal to both of those, since they are not equal to each other!
- So either $A[m] \neq A[s]$, in which case let $e = m$. Or $A[m] \neq A[e]$, in which case let $s = m$.
- This means that the property $A[s] \neq A[e]$ is still true, while the distance between s and e halved.
- We keep repeating this process, until s and e are right next to each other, when $i = s$ will be the solution.
- This is a general algorithm design technique / formal verification technique, called a loop invariant. A loop invariant is a property that is true before starting a loop and in the loop the operations we do keep the loop invariant true. So at the end, the property is still true, while the loop did something useful to us (in this case moved the pointers right next to each other).

11.4 Session 11, Exercise 4

Exercise

An array of pairwise distinct integers is called *bitonic*, if it is increasing up to an index and then it is decreasing, or vise versa, it starts as decreasing and turns into increasing at a point. For example the arrays $(1; 3; 7; 21; 12; 9; 5)$, $(9; 7; 5; 4; 6; 8)$ and $(1; 2; 3; 4; 5)$ are bitonic. Give an $O(n)$-time algorithm that sorts a bitonic array.

Solution

- We have two sorted parts of an array, and we need to combine them into one big sorted array. We can use the merging step of the merge sort algorithm to do this in $O(n)$ time!
- We check which variant of bitonic array we have, let's assume it's first increasing then decreasing.
- Put one pointer at the first and one at the last element of the array.
- Check which elements poitned by one of our pointers is smaller. Put that in a final array and move the pointer closer to the middle.
- The pointers should not run over to the other side of the bitonic array, since that would mean the other side had a smaller element and that pointer should have moved first.
- We will end up with a sorted array.
- If it is the other variant of a bitonic array we can either do a linear search to find the position where the decreasing part turns into an increasing one and place the two pointers there, moving them outwards or we can just sort in decreasing order first, then reverse at the end in $O(n)$ time.

11.5 Session 11, Exercise 5

Exercise

Array $A[1:n]$ contains integer numbers. Let b be a given integer. We want to find indices $i, j \in \{1, \ldots, n\}$, such that $A[i] + A[j] = b$. How can this be managed in $O(n \log n)$ running time?

Solution

- The naive algorithm would check all pairs in $O(n^2)$ time using nested loops, which is too much. However:
- $O(n \log n)$ = we can sort.
- Let's sort the array using merge sort for example, which runs in $O(n \log n)$ time.
- Then, instead of checking all $O(n^2)$ pairs, we are only going to iterate in a single loop, using index i. The inner loop is going to be replaced by binary search:
- since we know that $A[i] + A[j] = b$, we know exactly that we are looking for the value $A[j] = b - A[i]$.
- So just binary search $b - A[i]$ in the entire array (just make sure $i \neq j$),
- or just binary search in $A[i+1:n]$, since we don't need to check $A[1:i-1]$, as any pair that was there was already checked when the i pointer was there in the previous steps.
- (We would be checking all pairs twice if we searched in this part as well.)

11.6 Session 11, Exercise 6

Exercise

Give an $O(n)$ running time algorithm that sorts n integers that are from the range

a) $\{1, \ldots, 3n\}$

b) $\{1, \ldots, n^3 - 1\}$

Solution

- Let's say that the incoming integers are in array $A[1:n]$.
- To sort in $O(n)$ time we need to use bucket/radix sorting. **This is only available if we know the range of the incoming integers.** So in this task we can do this.
- In general, if we don't have an upper limit on the values in an array, bucket sorting is not possible (the first step is to define an array of size equal to the upper limit).
- For the first task, we can use bucket sort and define the array $B[1:3n]$ and initialize all buckets in it (all cells of the array) to 0.
- Then, we iterate over $A[1:n]$, for a_i we place that in the a_ith bucket, by incrementing the $B[a_i]$ value by 1.
- In the end, we iterate over $B[1:3n]$ and for every non-zero value, at index i we print the number i $B[i]$ times. This results in a sorted list of the input numbers.
- For the second task we can no longer simply initialize $B[1:n^3-1]$, since that would require $O(n^3)$ time, which is not allowed.
- This is where radix sort comes in: Let's instead imagine that the incoming numbers are in a base-n number system, so a_i becomes 3 base-n digits: $a_{i,2}a_{i,1}a_{i,0}$, or: $a_i = a_{i,2} \cdot n^2 + a_{i,1} \cdot n + a_{i,0}$.
- This is convenient, because in a base-n number system, on 3 digits, the largest possble number is exactly $n^3 - 1 = (n-1) \cdot n^2 + (n-1) \cdot n + (n-1)$.
- Then, radix sort is going to do exactly 3 bucket sorts (corresponding to the 3 digits or 3 parts ot the numbers).
- The first bucket sort will order the numbers based on the last digit, or $a_{i,0}$ for all i's.
- Instead of just counting how many times a specific $a_{i,0}$ value appears, we now need to keep track of the other two digits.
- So we initialize the array $B[1:n]$, but now we add stacks in every $B[i]$ cell.
- We iterate the a_i's, then push a_i into the stack at $B[a_{i,0}]$.
- Finally, we read out everything in $B[1:n]$, iterating over the indexes from 1 to n, and for $B[i]$, the stack must be read out and reversed.
- The reversion part is important, because we want to keep the original order in which the a_i's came in, when they are added to the same stack. A stack is a LIFO (last in first out), so we must reverse the order of the numbers as they come out.
- Since in total we have n numbers, the reversion still will be performed in $O(n)$ time.
- We could also read $B[1:n]$ from the last to the first index, without reversing at individual steps: this will result in a decreasing order, and we just reverse the entire result at the end.
- Doing bucket sort like this means that bucket sort is a stable sort. So elements that are equal (in the current run) will return in the same order they went into the algorithm.
- We repeate the same procedure using the middle and finally the first digit.
- When we sort based on the first digit: if there is an equality among the first digits, due to the stability of the sorting algorithm the order in which they came in will be the result. The order in which they came in was the sorted order according to the second digit. And when there is an equality among the second digits as well, then the order in which they came in coming into the second bucket sort will matter: so the increasing order according to the last digit, which was performed by the first bucket sort we did.

- This results in exactly what we want: sorted order, where the less significant digits decide the order, when the previous digits are equal.
- This is why we use a **stable sort** inner algorithm, bucket sort and do the sorting in **reverse order, from least significant to most significant** digits.
- Each bucket sort was performed in $O(n)$ time and there were 3 of them, which is still linear, so we are done!

11.7 Session 11, Exercise 7

Exercise

The value of one element of the originally increasing sequence $a_1, a_2, \ldots, a_n$ has changed, but we do not know which one. How can be the elements sorted again in increasing order in $O(n)$ running time?

Solution

- Simplest solution is to find the element that is not in correct order, using linear search. (It could be that the change will still result in an increasing order, so we might not find one at all and just exit.)
- After we found the element, we insert it into its correct postion (a single step of insertion sort). This is done in $O(n)$, for example by swapping the found element with its neighbours until we reach the correct spot.

11.8 Session 11, Exercise 8

Exercise

Graph $G = (V, E)$ that does not have parallel edges is given by its adjacency list, where the neighbours of each vertex are listed in arbitrary orders. Give a $O(|V| + |E|)$ running time algorithm that creates an adjacency list where neighbours of vertices are listed in increasing order.

Solution

Let's denote the adjacency list with $A[1 : n]$ ($n = |V|$), where $A[i]$ is an array of size d_i (the degree of vertex i), and $A[i, j]$ contains the jth successor of v_i, but in an arbitrary, unsorted order.

By successor here, we mean a neighbouring vertex, to which the edge points in the direction from v_i to the neighbour.

What won't work:

- We could try to iterate over $A[1 : n]$ and for every $A[i]$ index we use a comparison-based sort on the $A[i]$ array.
- We will run n comparison based-sorts, for the ith vertex running in $O(d_i \log d_i)$ time.
- In total $\sum_{i=1}^{n} O(d_i \log d_i) = O(\sum_{i=1}^{n} d_i \log d_i)$ time was used.
- We can upper-estimate this sum by taking the maximal $\log d_i$ value for all indexes, let's call denote his with $\log d_{max}$.
- $\sum_{i=1}^{n} d_i \log d_i \leq \sum_{i=1}^{n} d_i \log (d_{max}) = \log (d_{max}) \sum_{i=1}^{n} d_i = \log (d_{max}) \cdot |E| \leq \log (|V|) \cdot |E|$.
- This is not going to work unfortunately, we need $O(|V| + |E|)$ and $O(\log (|V|) \cdot |E|)$ is too much. :(
- Even if we tried to run bucket sort on $A[i]$, a single bucket sort will run on d_i elements and $|V|$ buckets in $O(d_i + |V|)$ time, resulting in total $O(|E| + |V|^2)$ runtime (summing up for all vertices).

Solution 1:

- Transform the $A[i]$ format into an edge list format, e.g. $E[k]$ contains the kth edge of the graph, represented by vertices (i, j).
- We can do this in $O(|V| + |E|)$ time, since we must iterate $A[1 : n]$, and for $A[i]$ we read through d_i vertices, so in total we will read in the sum of all degrees, or $|E|$ edges. For each of these we create an item in the resulting E array.
- Then we can run a radix sort on these, (e.g. if the edges are (i, j), sort by j first into $|V|$ buckets, then by i into $|V|$ buckets again).
- This will be the desired order, we only need to get back to the original format.
- We can do this by emptying out the lists in $A[1 : n]$ first, then iterating over the sorted E edge list and for edge (i, j) we add j back to $A[i]$.
- This is also done in $O(|E|)$ time and emptying out the lists is $O(|V| + |E|)$.

Solution 2:

- We already know an algorithm that can reverse the edges of a directed graph.
- This is done by creating a new empty adjacency list A' and filling it by iterating over the original $A[1 : n]$ and for the vertex $A[i, j] = u$ (e.g. there is an edge pointing from i to u) we put i into the list of $A'[u]$.
- We also know that this will result in ordered edge lists, since the i's that we inserted into the edge lists of A' came from the outer index of the A table, which we iterated over in increasing order.
- This is also done on $O(|V| + |E|)$ runtime, similarly to the previous solution.
- So if we do this edge reversal twice, we will get back the original graph, but now in the desired sorted order.

11.9 Session 11, Exercise 9

Exercise

Prove that there exists no comparison based sorting algorithm that compares each element at most 2022 times on every possible input.

Solution

- So in total for n elements it would do at most $\frac{2022n}{2} = 1011n$ comparisons. (The division by 2 is there because each pair is counted from both individual elements of the pair.)
- We have studied that all comparison based sorting algorithms run in $\Omega(n \log n)$ time.
- However $1011n \notin \Omega(n \log n)$, since no constants c, n_0 exist that $c \cdot n \log n \leq 1011n$ for all $n \geq n_0$,
- as that would mean $\log n \leq \frac{1011}{c}$, or $n \leq 2^{\frac{1011}{c}}$ for all $n \geq n_0$, so n would have to be upper bound by a constant, but it can be arbitrarily large, which is a contradiction.

11.10 Session 11, Exercise 10

Exercise

Array A contains positive numbers, not necessarily in sorted order. Give an algorithm that chooses k elements of A so that the sum of the chosen elements is not larger than k^3. If there is no such k, then the algorithm should note that. The running time must be $O(n \log n)$.

Solution

- The value of k is not an input, but also something we have to find during the algorithm. It can be anywhere between 1 and n (the size of the array).
- It is not required to choose consecutive numbers from A, so their placement is irrelevant.
- Since we are allowed $O(n \log n)$ time, we can sort the array, using merge sort for example, which runs in $O(n \log n)$ time.
- Now, if we wanted to choose k elements, so that their sum is less than or equal to k^3, it is always better to choose the k smallest elements, than any other. Choosing any other would only increase the sum, which would be worse for us.
- So we are going to iterate the sorted array A, starting from $i = 1$ to $i = n$. We will keep track of the current sum, starting with $sum = 0$ and in step i we add $sum = sum + a_i$.
- After doing this, we check whether $sum \leq i^3$ is true.
- If this is true, we can return the current i as $k = i$.
- Even if this is not true for a specific i, it can still become true later, as we add more elements to the sum, since i^3 can increase more than the additional elements added.
- If we reach $i = n$ and we had no success, we return NOT POSSIBLE. This is correct, since we tested all $k = 1$ to $k = n$ choices and for each we tested the k smallest elements in the array. If it is not possible using these, then it is definitely not possible using larger elements instead.

11.11 Session 11, Exercise 11

Exercise

Vertices of a binary tree are labeled by integers between 0 and 9 (inclusive). The *InOrder* walk lists the labels as $9, 3, 1, 0, 4, 2, 7, 6, 8, 5$, while the PostOrder lists them as $9, 1, 4, 0, 3, x, 7, 5, y, 2$. What could x and y be?

Solution

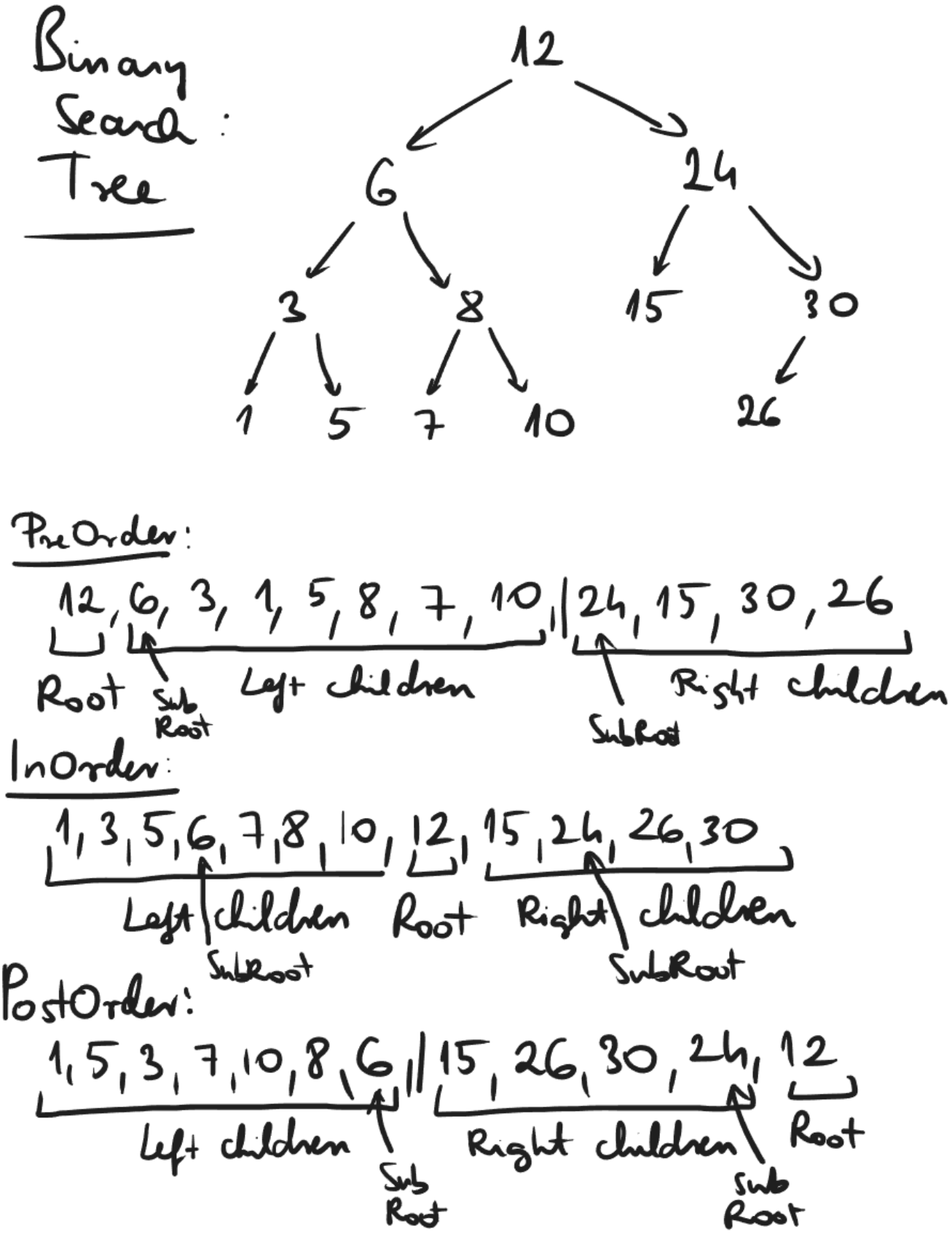
- Note here: **the labels are not the keys** based on which the binary tree is structured! We can see this since the *InOrder* walk is supposed to return the keys in increasing order and the labels it returned are not sorted!
- The labels are just there to hide the actual keys from you in this exercise.
- First let's talk about the *PreOrder*, *InOrder* and *PostOrder*, walks of the binary search tree:
- Let's imagine that the following struct represent a node in the tree:

```
struct {
    int key;
    node* left;
    node* right;
} node;
```

- Now, every single walk will explore the left-side of any node first and the right-side later. The difference comes from **when the current node itself is explored**.

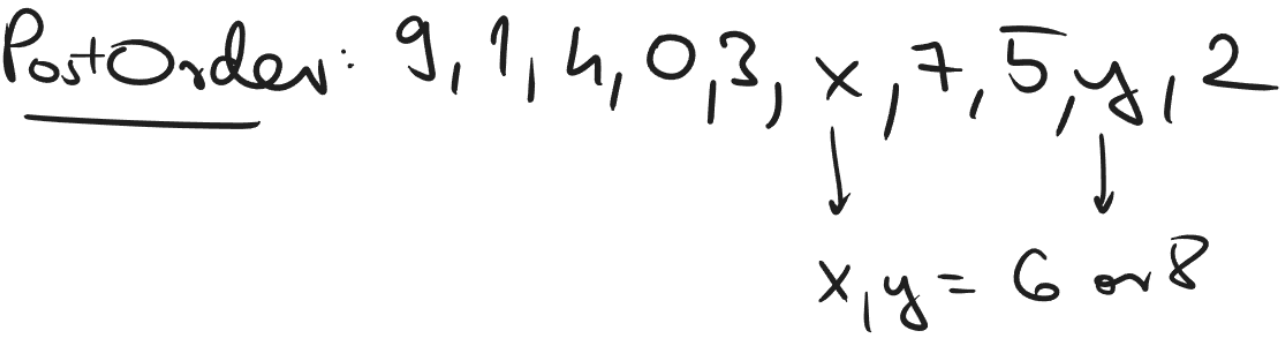
```
void preOrder(node current) {
  // Explores current node before moving on to children.
  cout << current.key;
  preOrder(current->left);
  preOrder(current->right);
}

void inOrder(node current) {
  inOrder(current->left);
  // Explores current node after finishing left but before starting right child.
  cout << current.key;
  inOrder(current->right);
}

void postOrder(node current) {
  postOrder(current->left);
  postOrder(current->right);
  // Explores current node after all children were explored.
  cout << current.key;
}
```

- Let's run these algorithms on an example:

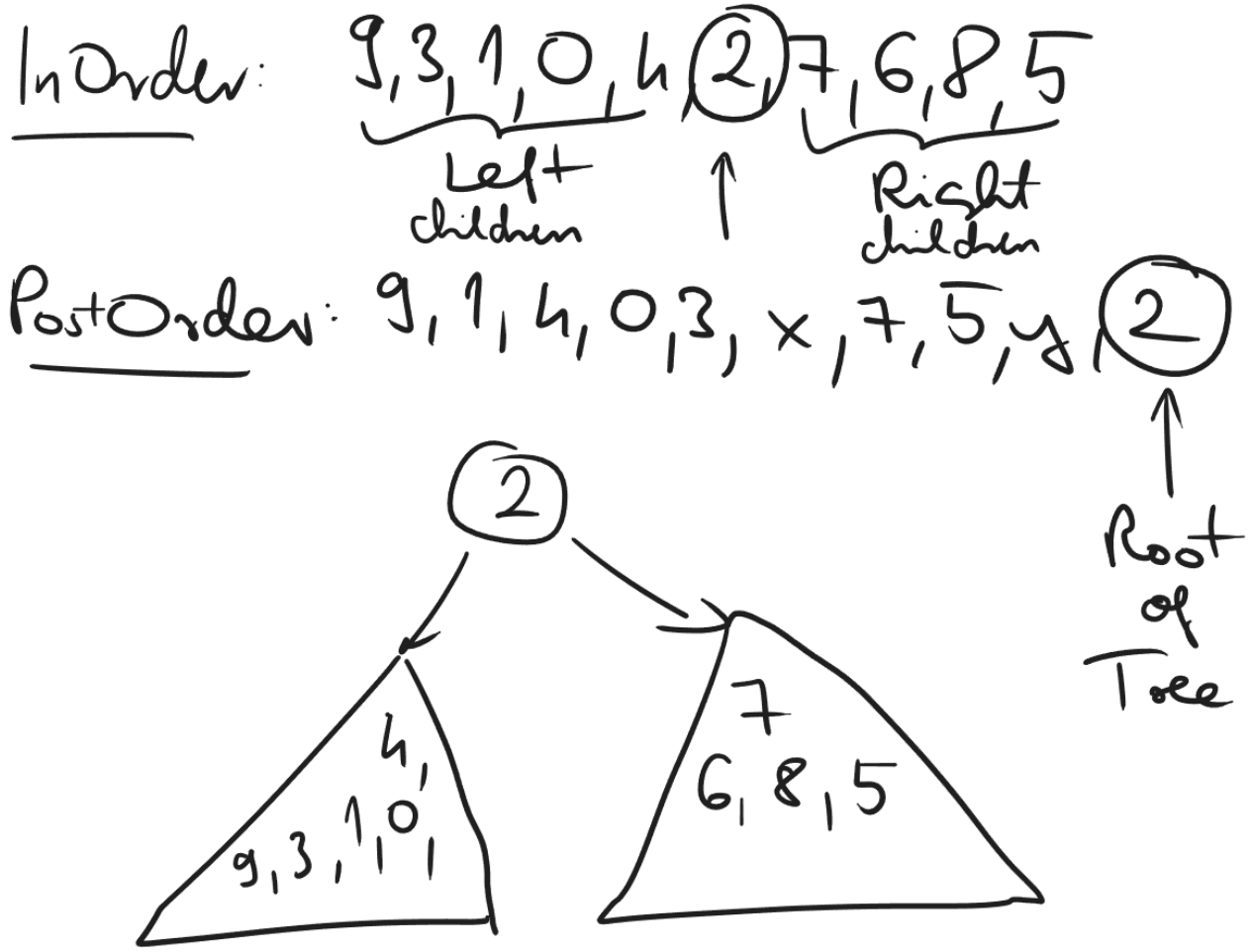


- Interestingly, if we are given the InOrder walk result and either a PostOrder or a PreOrder result, we will be able to reconstruct the original tree.
- In this example, the InOrder result is redundant, since we know the correct increasing order of the number, but in the case of the original exercise the labels hide this information.
- So let's say if we know the InOrder and PostOrder results:

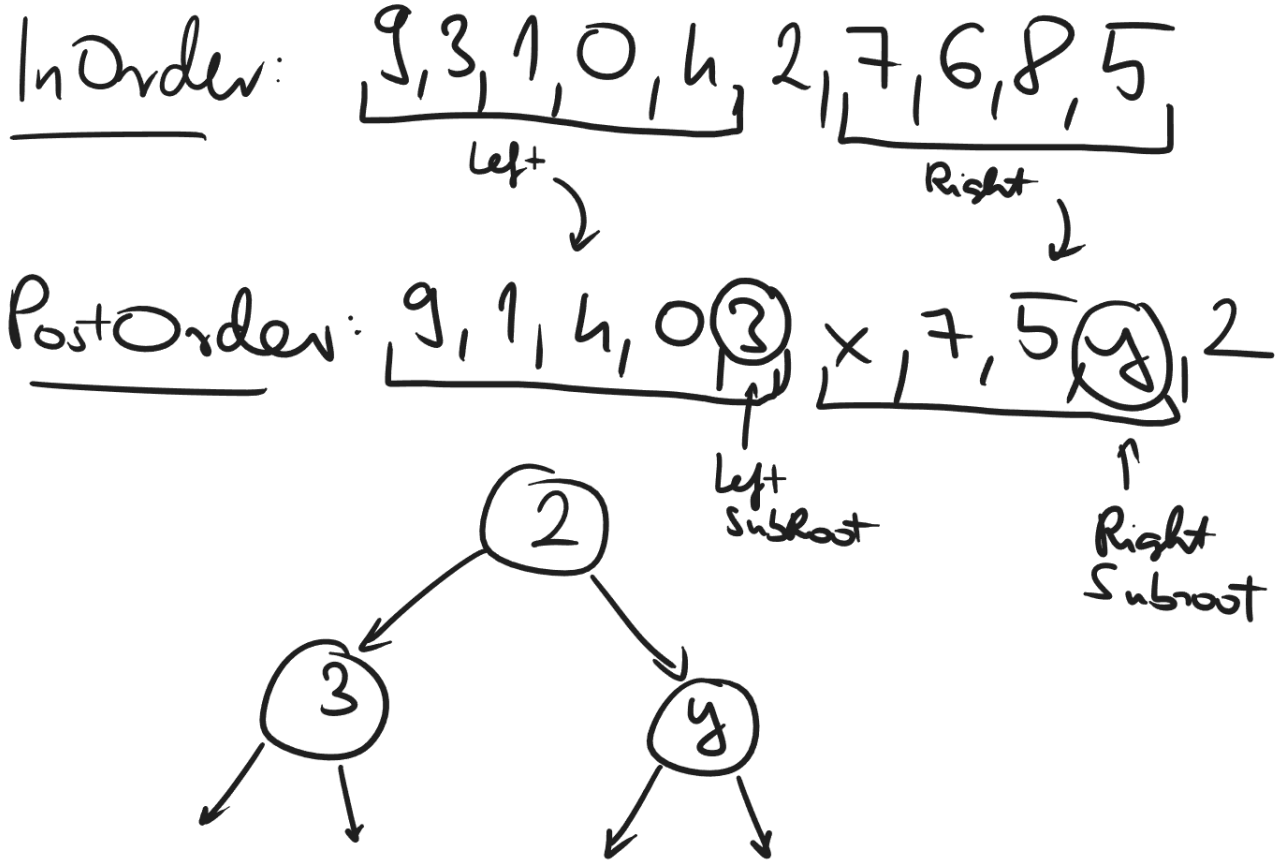
- At first, we look at the PostOrder result. We know for a fact, that the Root of the entire tree will be the very last element in this result.
- So looking at the PostOrder numbers, we know that the root is 12.
- Now we look over to the InOrder list: here the Root is going to be between its left children and right children.
- From this information we know that the left children of 12 are $1, 3, 5, 6, 7, 8, 10$ and the right children are $15, 24, 26, 30$.
- Now if we look back at the PostOrder results, we finally know that the separator between the left and right children is actually between 6 and 15.
- We can now continue the same thing recursively for the two subtrees.
- The root of the left subtree is 6, since that is the last element of that subtree in the PostOrder result.
- If we look at the InOrder result for that subtree, we know that 6 is between its left and right children, so the left children of 6 must be $1, 3, 5$ and the right children must be $7, 8, 10$.
- You can try to finish this example at home, but I will do this algorithm for the original exercise.



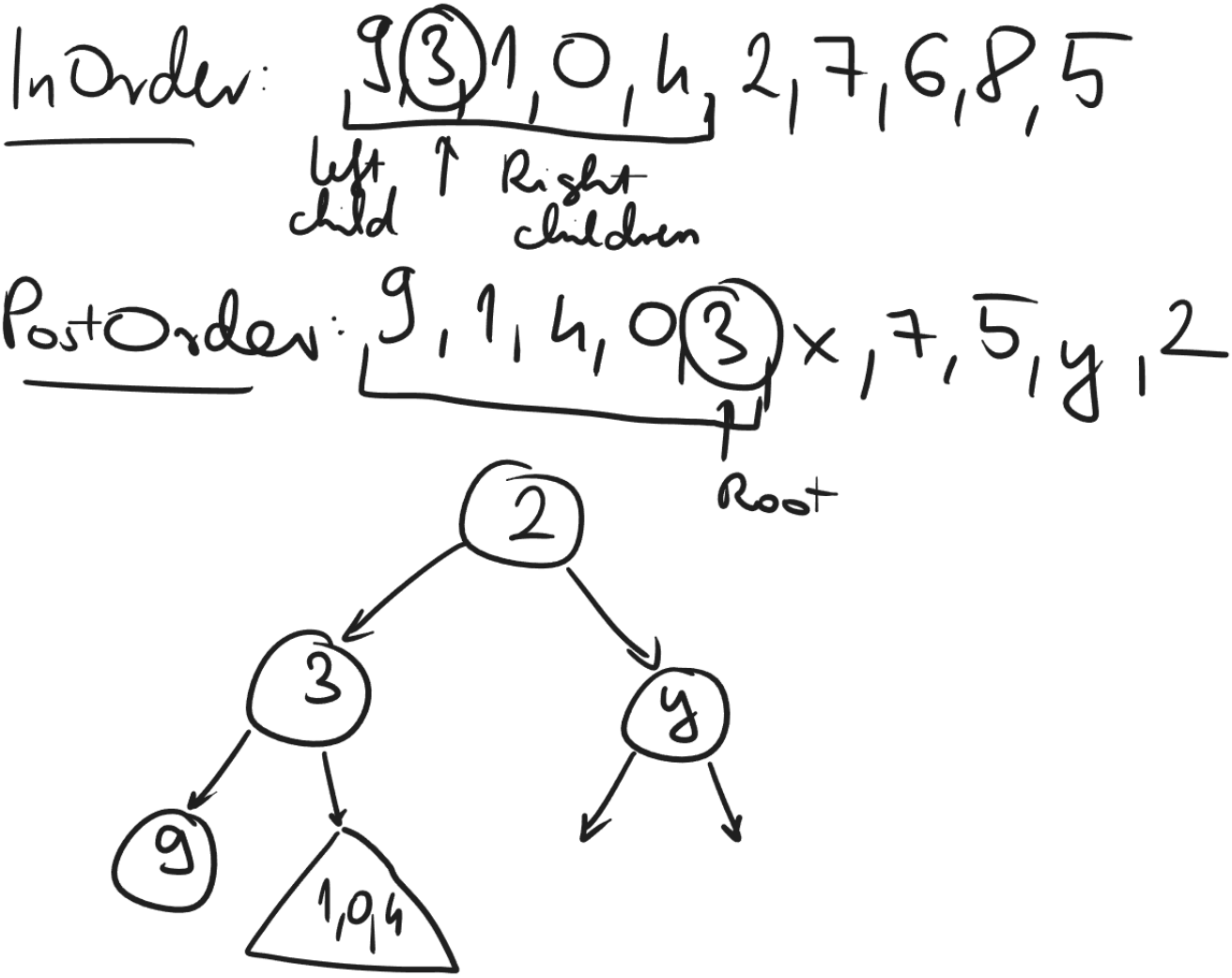
- We see that the two missing numbers from the PostOrder walk are 6 and 8 so x and y will be these two in some order.



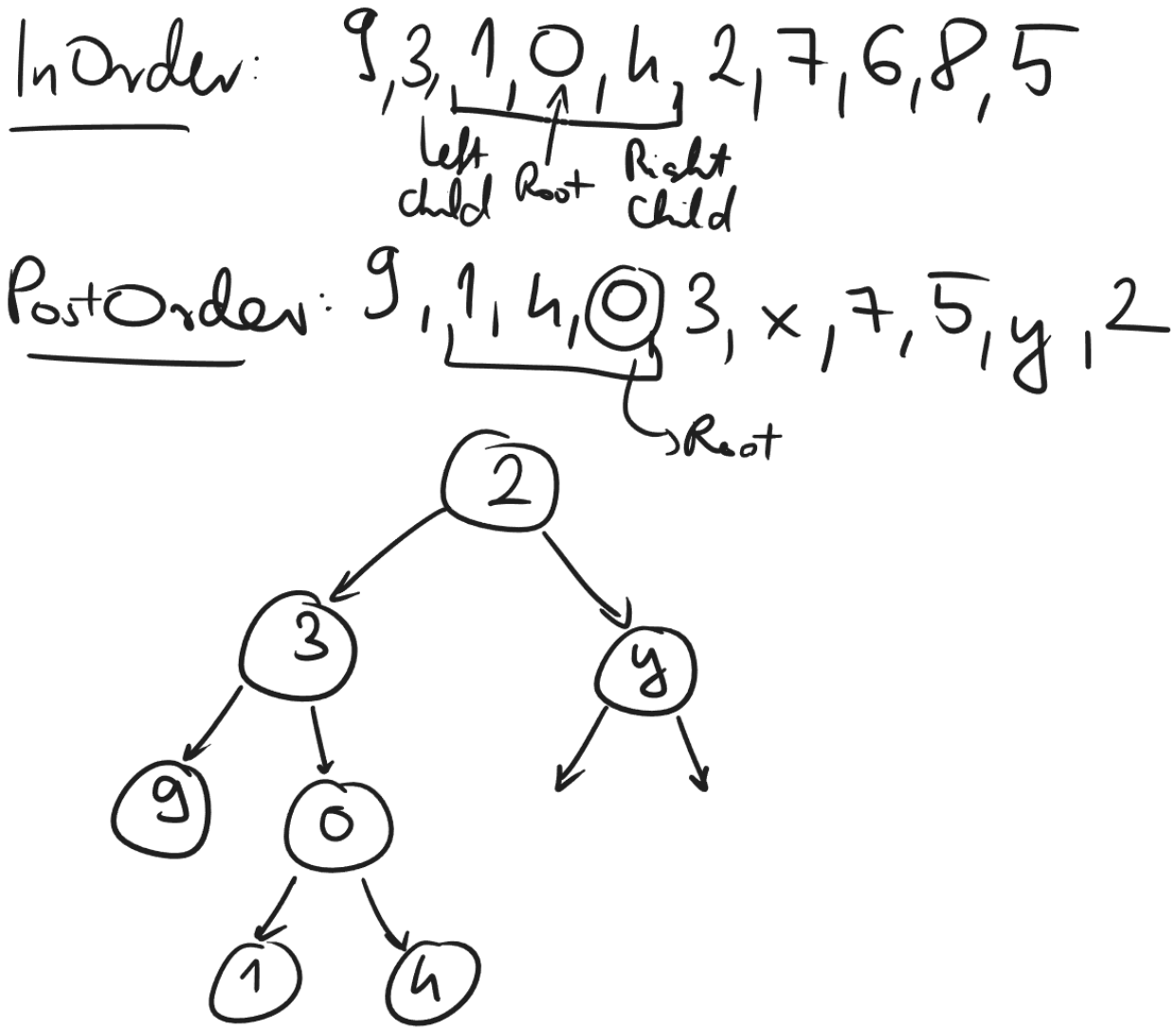
- Since 2 is the last element of the PostOrder walk, that is the root of the tree.
- Looking at the InOrder result, we see which elements are on its left and which are on its right.



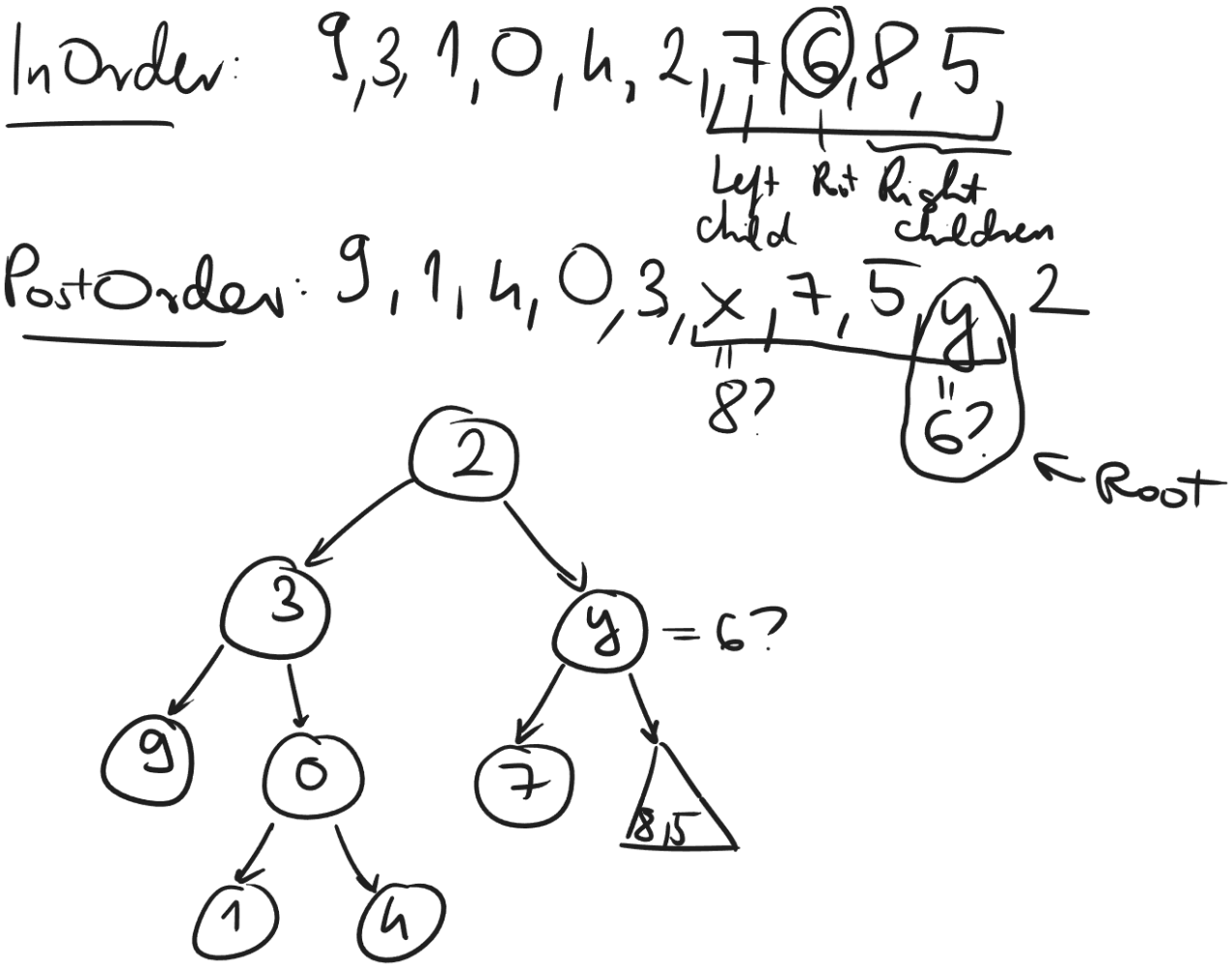
- If we go back to the PostOrder result and look for the same elements, we see that the subroot of the tree on the left is 3, since that is the last element of that sublist. Similarly, y is the root of the right subtree, since that is the last element of that sublist. Both x and y must be on the righthand side of the root.



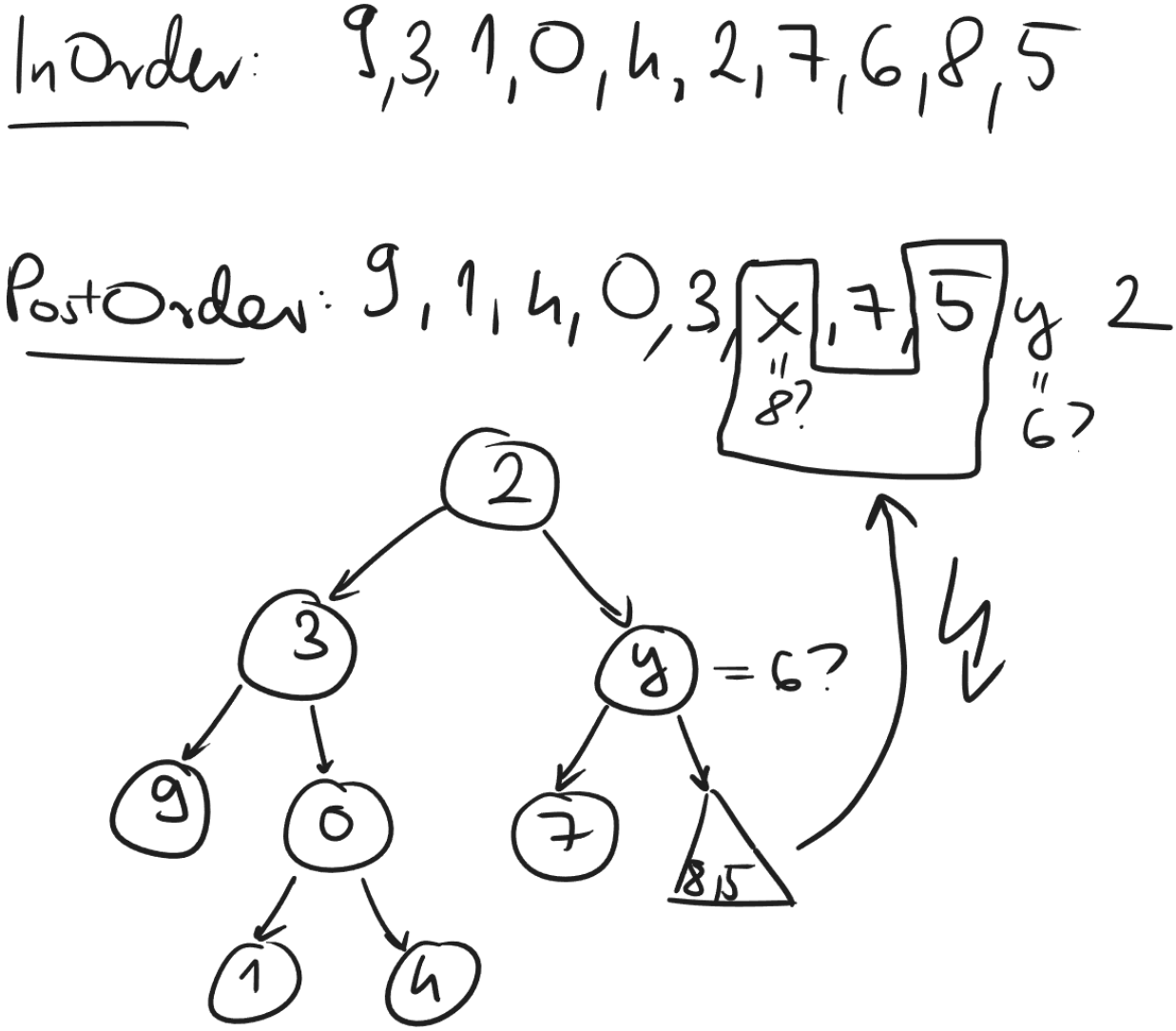
- If we continue on the left side, we see that the root is 3, so looking at the InOrder list, we see that the left child is just the number 9, while the right side contains numbers $1, 0, 4$.



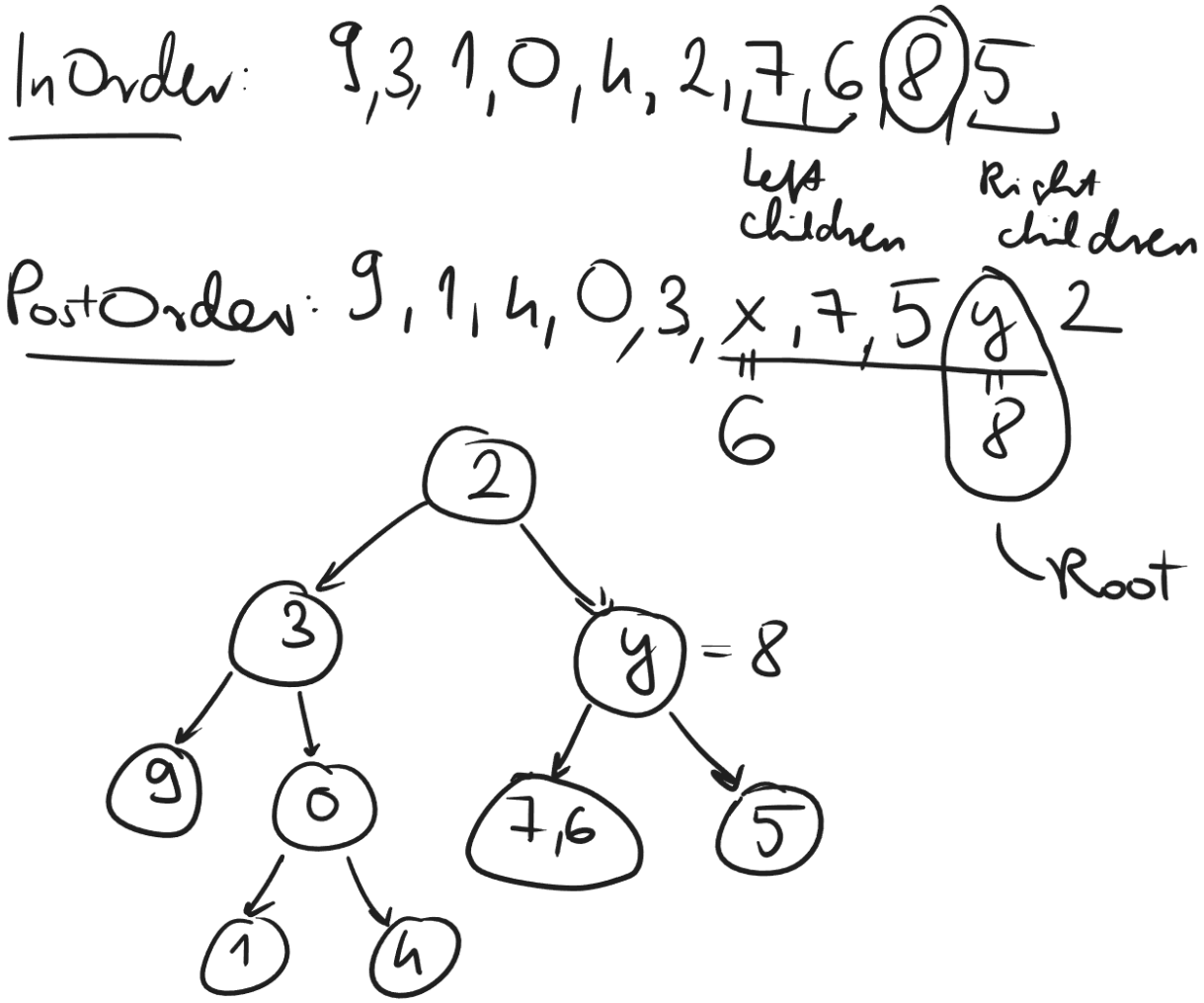
- Looking at the subtree $1, 0, 4$ in the PostOrder walk, we see that the last element is 0, so that will be the root of that subtree.
- Then looking back at the InOrderWalk, we see that the left child must be 1 (on the left of the root), while the right child must be 4 (on the right of the root).
- We have reconstructed the left side of the root of the original tree.
- Now we need to think about the right side. y could either be 6 or 8.
- Let's try $y = 6$ and $x = 8$.



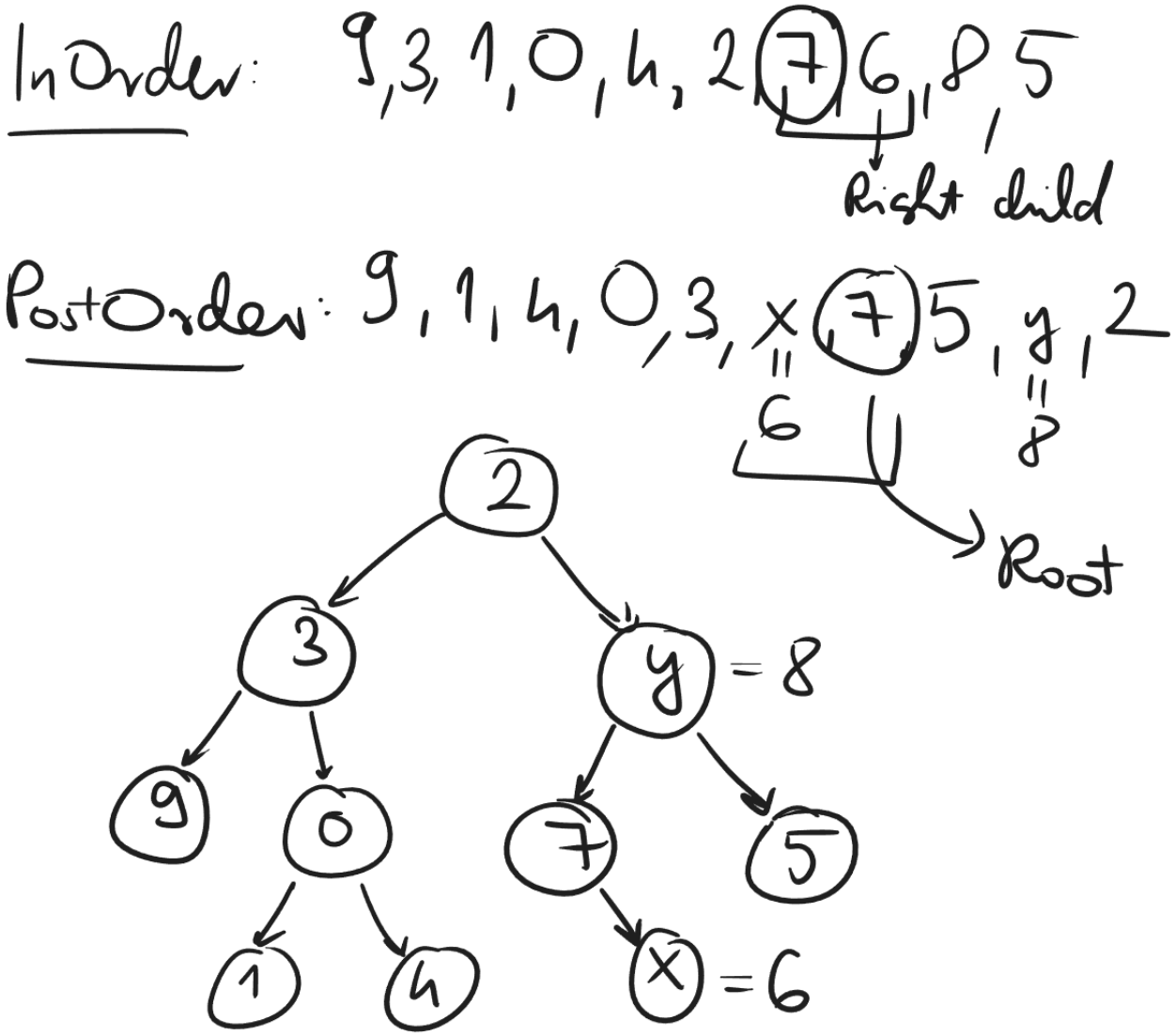
- Then we can step further, the 7 must be the left child of $y = 6$ and $x = 8$ and 5 must be on the right.



- However, this will result in a problem: $x = 8$ and 5 are a subtree, but in the PostOrder walk they appear not consecutively. Number 7, which is their sibling subtree comes between them. This is impossible, subtrees should always appear in a consecutive order in all three of the walks! So this assignment cannot be correct!
- The only other way is $y = 8$ and $x = 6$, so let's go back and try doing that:



- Now we see that the left children of $y = 8$ are 7 and 6, while the right child is 5.



- This works well, since 7 is the root of the $7, 6$ subtree, we see this from the PostOrder result. Finally, 6 is a right child of 7, we see this from the InOrder walk.
- So $x = 6$, $y = 8$ and the corresponding tree is the one above.

11.12 Session 11, Exercise 12

Exercise

Determine those binary trees whose vertices are listed by *PreOrder* walk exactly in opposite order as it is done by *PostOrder* walk.

Solution

- Since *PreOrder* does $node, left(node), right(node)$ and *PostOrder* does $left(node), right(node), node$, if there was an element in both the left and the right subtree, the order of those would not change relative to each other, so it could not result in a fully reversed order globally.
- So the only way we can get a completely reversed order is if every node has at most one child.
- That is called a linked list. For a linked list, these two walks indeed return exactly the opposite order ($node, child(node)$, vs $child(node), node$).
- So the linked list (the degenerative case of a binary seach tree) is the type of binary tree for which the result of *PostOrder* and *PreOrder* is exactly the opposite order.

11.13 Session 11, Exercise 13

Exercise

n points are given on the plane whose coordinates are $(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)$, respectively. We want to find a point $P = (x, y)$ on the plane so that the sum $\sum_{i=1}^{n}(|a_i - x| + |b_i - y|)$ is minimal. Give an $O(n \log n)$ running time algorithm that determines one such point P.

Solution

- The coordinates x and y are independent of each other. To minimize $\sum_{i=1}^{n}(|a_i - x| + |b_i - y|)$ is the same as minimizing $\sum_{i=1}^{n}(|a_i - x|)$ for x and $\sum_{i=1}^{n}(|b_i - y|)$ for y separately.
- To minimize $\sum_{i=1}^{n}(|a_i - x|)$ for an x.
- Let's imagine x on a $1D$ coordinate axis.
- If x is smaller than the smallest a_i, by moving x to the right we only decrease the sum.
- As long as there are less a_i's to the left of it than to the right of x, the sum will keep decreasing as we move x to the right. (The distance will change by the same amount for each individual point, however the ones on the left of x will increase, while the ones on the right will decrease.)
- However, if more a_i's are behind x then it is no longer optimal to move x to the right anymore.
- If there are an odd number of a_i's, then x has to be equal to the middle one (the median).
- If there are an even number of a_i's, then x can be anywhere between the two middle ones. For example, median is defined by the arithmetic average of the two middle a_i's, so in this case we can still chose the median of the a_i's as x.
- To find the median, we must sort the a_i's, using for example merge sort, which runs in $O(n \log n)$ time. Then we select the middle one, or the two middle ones and take their averages for x.
- We do the exact same thing for y.

11.14 Session 11, Exercise 14

Exercise

Give an efficient algorithm that finds the smallest and the largest element among n numbers.

Solution

- The array is not necessarily sorted.
- Efficiency in this case means finding an algorithm for the smallest possible number of comparisons.

Firs try:

- We can use linear search, in $O(n)$ time. (This is better than sorting.)
- We keep track of the minimum and maximum we have seen so far, and for every current index, if that value is smaller than the minimum or greater than the maximum so far, we update those values.
- In the end we'll have the minimum and maximum in the array.
- This will result in $2 \cdot (n-1) = 2n-2$ comparisons, since the first element is not compared to anything, but after that each element will be compared both to the current minimum and maximum.
- Actually, we can even do one less comparisons: select the first two values of the array, compare then with each other and take the smaller as current minimum and the bigger as current maximum. This is 1 comparison. Then compare the rest of the $n-2$ elements as previously: $2(n-2)+1 = 2n-3$.

Better solution:

- Let's pair up the elements of the arry with their next neighbours: $A[1]$ pairs with $A[2]$, $A[3]$ pairs wth $A[4]$ and so on, $A[n]$ might not having a pair if n is odd.
- Compare the pairs between each other. This uses $\lfloor \frac{n}{2} \rfloor$ comparisons.
- Now from each pair, there will be a smaller and a bigger number. Put these into two separate arrays. If n is odd, compare the last 3 numbers with each other using 3 comparions to find the minimum and maximum between the 3.
- Run the previous minimum finding algorithm on the array containing the smaller numbers: the array has $\lfloor \frac{n}{2} \rfloor$ numbers, so the algorithm will run in $O(\lfloor \frac{n}{2} \rfloor - 1)$. Same for the other one, but find the maximum there.
- In total we have done $\lfloor \frac{n}{2} \rfloor + 3 + 2(\lfloor \frac{n}{2} \rfloor - 1) \approx 1.5n$ comparisons.

11.15 Session 11, Exercise 15

Exercise

Give an efficient algorithm that finds the smallest and the second smallest element among n numbers.

Solution

- The smallest can be found in $n - 1$ steps.
- Imagine this process as a tournament: whenever two numbers are compared, the smallest one "wins" and moves forward.
- The minimum is the one that wins the tournament. The second smallest can only lose to the minimum, so we can trace back who the minimum was compared to and check only those numbers for being the second smallest possible.
- We know that the tournament can be done in $O(\log_2(n))$ rounds, so the second smallest can be between $\log_2(n)$ numbers (that the final winner played with).
- So we find the minimum of these $\log_2(n)$ numbers in $\log_2(n) - 1$ comparisons for the second smallest.
- In total we have done $n - 1 + \log_2(n) - 1 = n + \log_2(n) - 2$ comparisons.

12 May 18th (Session 12): Binary Search Trees, 2-3-trees, Hash

12.1 Solutions written by Prof. Attila Sali, on Moodle

Solutions to May 18th (Session 12): Binary Search Trees, 2-3-trees, Hash: To Moodle.

12.2 Session 12, Exercise 9

Exercise

Hash function $h(x) = x \pmod{M}$ is used in open addressing hash to insert keys $4, 5, 14, 15, 16, 26, 3$ in this order into an initially empty table of size $M = 11$. Show the resulting table if

a) linear probe is used

b) quadratic probe is used

c) double hashing is used with second hash function $h'(x) = 7x \pmod{(M-1)}$ is used

for collision resolution. How many collisions occure in each case?

Solution

All types of open hashings try in the original $h(x)$ position first. When a collision happens, they try the following offsets:

- Linear probing starts to the left, and tries the offsets $-1, -2, -3, \ldots$ etc.
- Quadratic probing starts to the **right** (!), and tries the offsets $+1, -1, +4, -4, +9, -9 \ldots$ etc (square numbers).
- Double hasing uses a secondary hash function, h'(x) and tries the offsets $-1 \cdot h'(x), -2 \cdot h'(x), -3 \cdot h'(x), \ldots$ etc.

Whenever you run out at the end of the array, you come back at the other. (Or, you can imagine the indexing having a mod tablesize applied to it.)

The offsets are not cumulative, all of them are applied to the original h(x).

	Linear probing (starts to the left)											Hash 1	Hash 2 (for c))
Indexes	0	1	2	3	4	5	6	7	8	9	10		
(0 collisions)					**4**							h(4) = 4	
(0 collisions)					4	**5**						h(5) = 5	
(0 collisions)				**14**	4	5						h(14) = 3	
Probing order:			3	2	1								
(2 collisions)			**15**	14	4	5						h(15) = 4	
Probing order:		5	4	3	2	1							
(4 collisions)		**16**	15	14	4	5						h(16) = 5	
Probing order:	5	4	3	2	1								
(4 collisions)	**26**	16	15	14	4	5						h(26) = 4	
Probing order:	4	3	2	1							5		
(4 collisions)	26	16	15	14	4	5					**3**	h(3) = 3	

Quadratic probing (starts to the right)

Indexes	0	1	2	3	4	5	6	7	8	9	10	
(0 collisions)					**4**							h(4) = 4
(0 collisions)					4	**5**						h(5) = 5
(0 collisions)				**14**	4	5						h(14) = 3
Probing order:				3	1	2			5			
(3 collisions)				14	4	5			**15**			h(15) = 4
Probing order:						1	2					
(1 collision)				14	4	5	**16**		15			h(16) = 5
Probing order:	5			3	1	2			4			
(4 collisions)	**26**			14	4	5	16		15			h(26) = 4
Probing order:			3	1	2							
(2 collision)	26		**3**	14	4	5	16		15			h(3) = 3

Double hashing (starts to the left)

Indexes	0	1	2	3	4	5	6	7	8	9	10		
(0 collisions)					**4**							h(4) = 4	
(0 collisions)					4	**5**						h(5) = 5	
(0 collisions)				**14**	4	5						h(14) = 3	
Probing order:					1						2		
(1 collision)				14	4	5					**15**	h(15) = 4	h'(15) = 7*15 (mod 10) = 105 (mod 10) = 5
Probing order:		3		2		1							
(2 collisions)		**16**		14	4	5					15	h(16) = 5	h'(16) = 7* 16 (mod 10) = 112 (mod 10) = 2
Probing order:			2		1								
(1 collision)		16	**26**	14	4	5					15	h(26) = 4	h'(26) = 7*26 (mod 10) = 182 (mod 10) = 2
Probing order:	4	3	2	1									
(3 collisions)	**3**	16	26	14	4	5					15	h(3) = 3	h'(3) = 7*3 (mod 10) = 21 (mod 10) = 1

13 Exam: 2019. 01. 06.

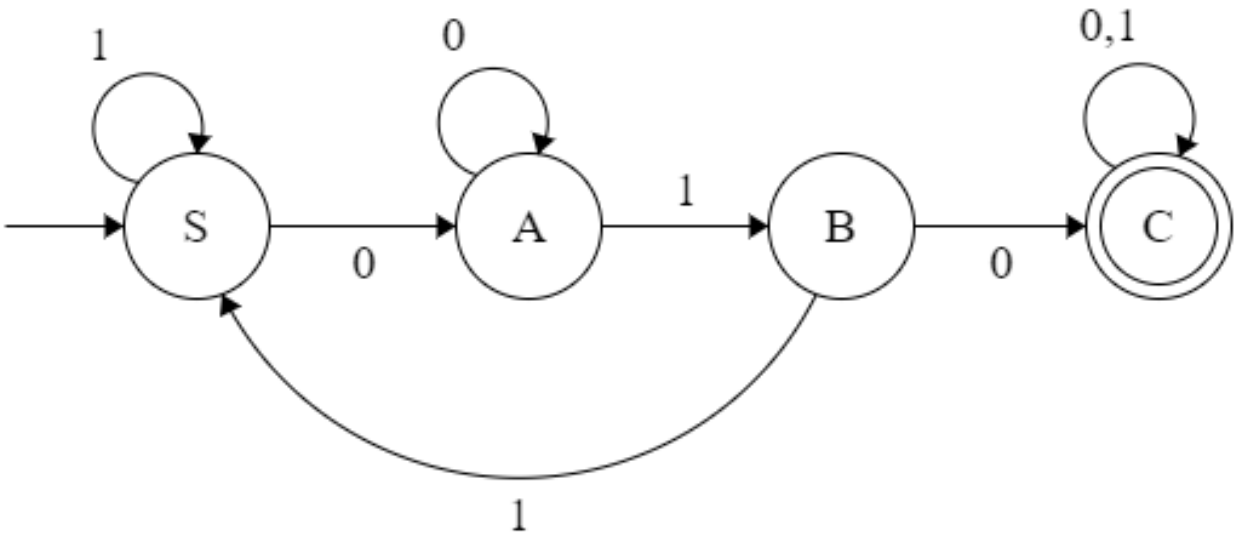
13.1 Exams: 2019. 01. 06., Exercise 1

Exercise

Construct a deterministic finite automaton that accepts thos words from $(0 + 1)*$ that contain subword 010 exactly once. (For example 0100110 is an element of the language, however 010010 and 01010 do not belong to it.)

Solution

The first subtask to do is to construct a DFA that accepts words that contain the 010 subword (without the other requirements):



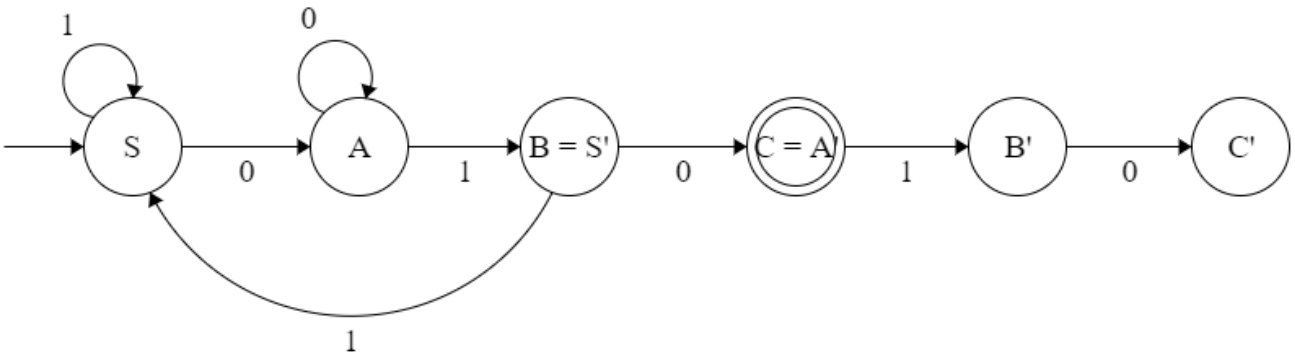
All of the states represent how long of a prefix we have received so far: S for no useful prefix, A for the "0" being the last character we have read and possibly continuing with "10" after, B for the 10 word being the last two characters we have read and possibly continuing with "0" and finally C means we have read the entire word.

Whenever we receive one more matching character, we move upwards, while if we receive the other character we need to fail back some steps. In state S, if we get 1's we just stay in S, that is not useful, we need a 0 to start. In state A, if we receive 0's, we can stay in 0, since the last 0 character is useful to us. In state B, if we receive a 1, that ruins everything, we know that that means the last characters on the input were 011, which has no useful parts to us, we fail back to S.

Notice additionally, that this automaton reaches the state C the **first time** it finds the 010 on the input.

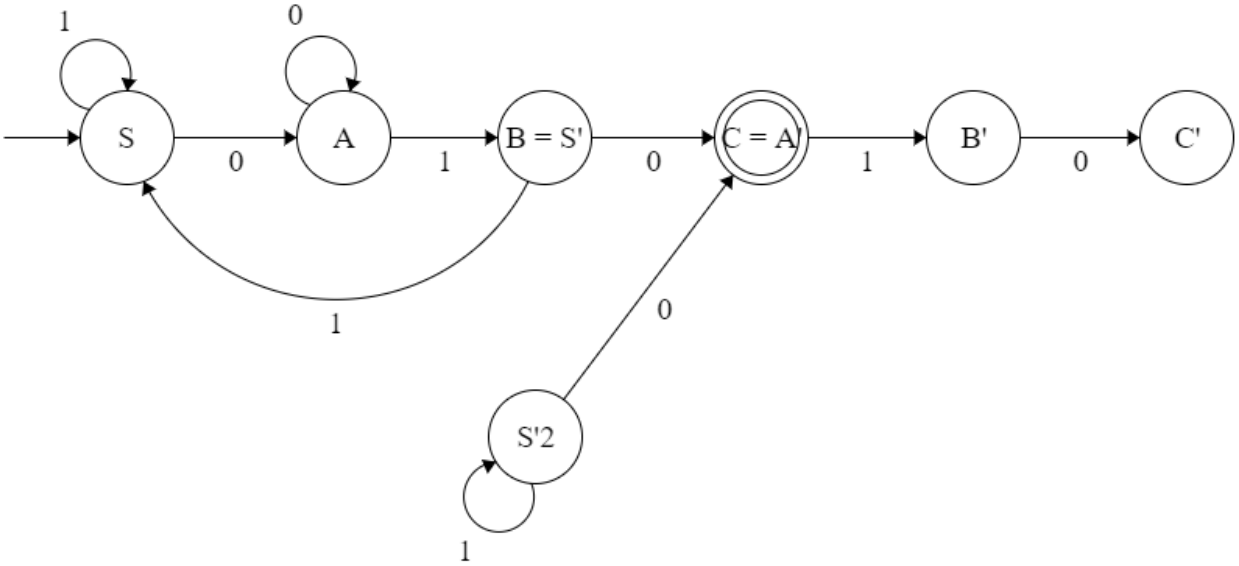
This means that if we start trying to find another 010 there, in a similar manner, we can detect a second 010. We cannot start in C, however, C should already mean the 0 was received, so we kind of copy and paste this DFA onto itself, the next copy having $S' = BandA' = C$, like so:

(And remove the looping on the original C.)



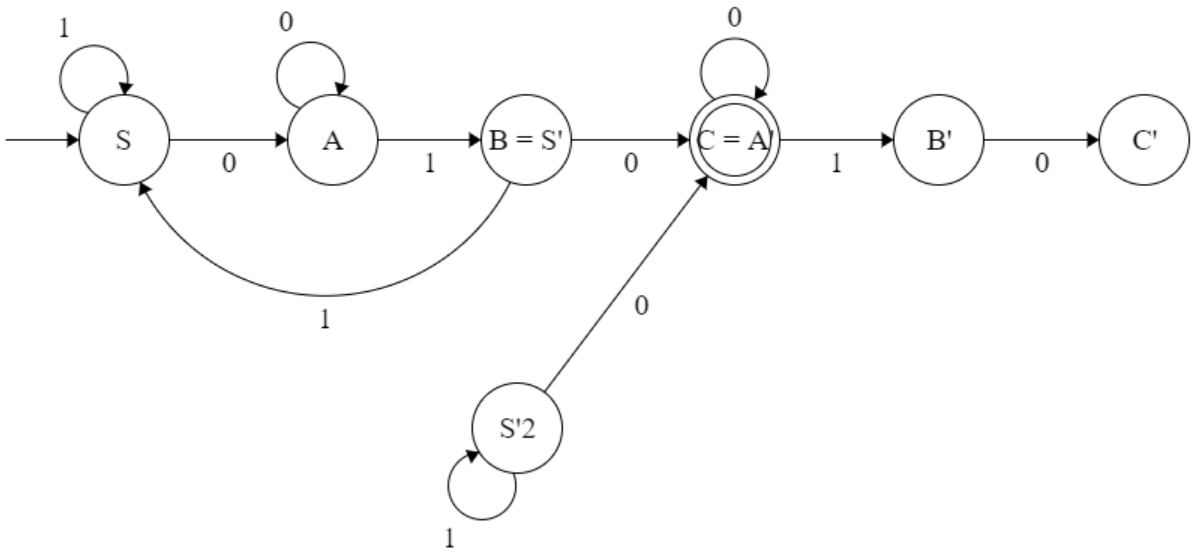
I have added the forward pointing transitions, and now we need to be really careful about the backward pointing ones:

There is a loop of 1's in state S, this is waiting for the computation to start. We cannot add this to $B = S'$, because there is already a 1 exiting from it. However, we need a similar function in the second part of the automata, something that we can go back to when nothing matches, and we can wait there until we are reading 1's. So for this, we introduce a new state:

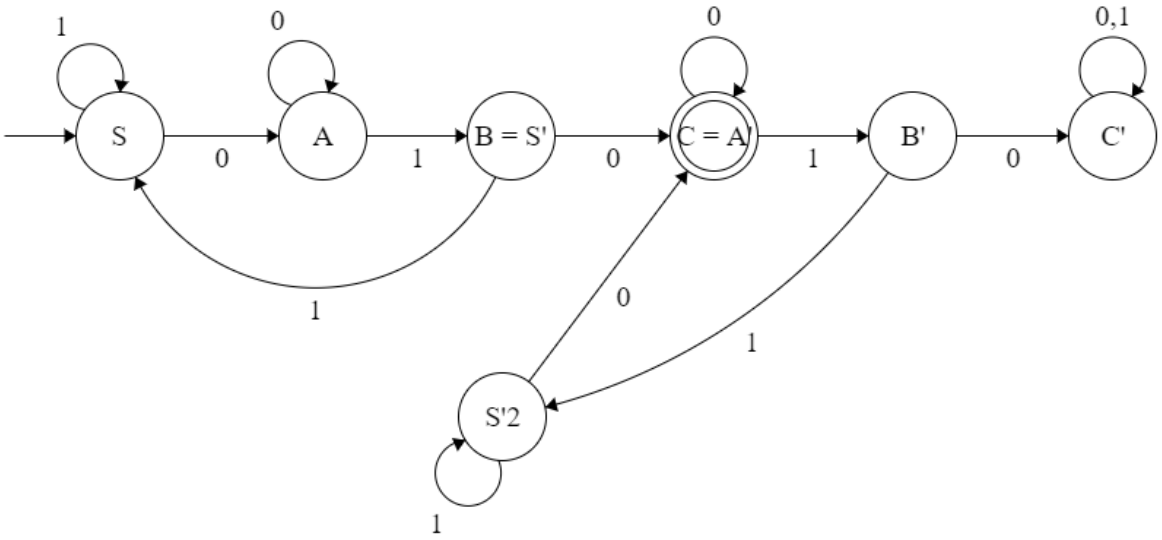


Whenever in the later parts we would move back to S', instead we will use S'_2 so that will correctly loop with the 1's instead of throwing us back at the very beginning.

Now I can add the 0 loop to $C = A'$:



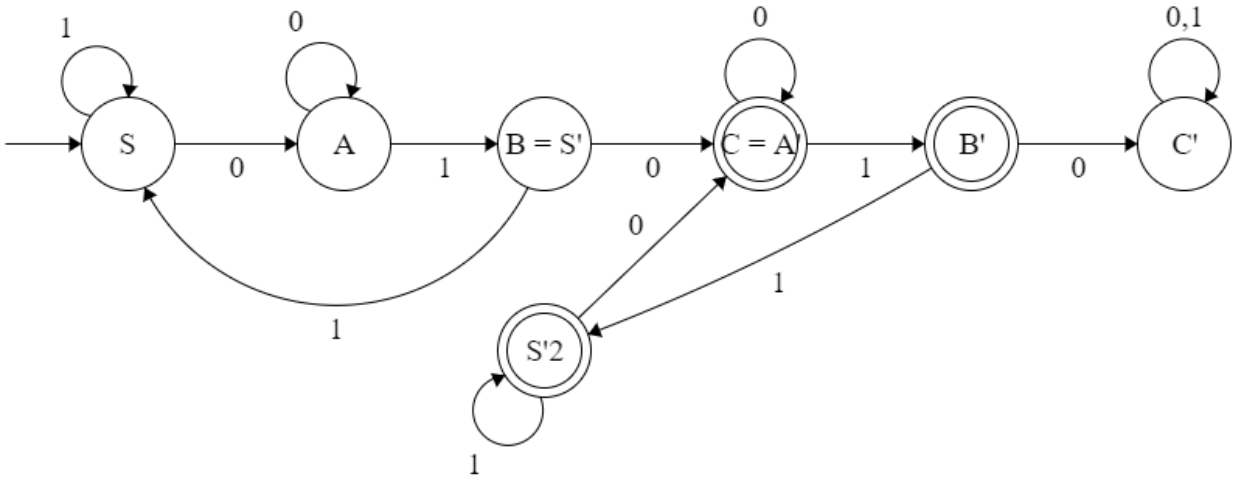
Then in B', if we read a 1 we move back to S, and in this case we are going to use our S'_2 state, so it cannot escape back to S, which we cannot allow:



And finally I added the $0, 1$ loop to C' too, since there we can wait forever.

The new accepting states will be anything new that is not C', since C' means we have found the second 010 in our input, for which we should fail.

So the final DFA is:



So essentially the second 010 recognizer here is S_2', A', B', C', and we arrive into it by connecting the last state of the first recognizer, C, to the A' state, since we already have the first 0 when we arrive.

14 Exam: 2019. 06. 17.

14.4 Exams: 2019. 06. 17., Exercise 4

Exercise

The Faculty of Electrical Engineering and Informatics works on a grandiose research project that has subprojects $Q = \{Q_1, \ldots, Q_k\}$. It is known for each researcher R_i of the faculty, which subset $S_i \subseteq Q$ of subprojects can she/he work on. A subproject can be successfully executed if at least 3 researchers work on it. Each researcher belongs to one of the 10 departments of the faculty and it is necessary that each department take part in the big project. However, we would like to complete the project with the minimum number of participating researchers. Which researchers should be involved to have at least three people working on each subproject, there must be at least one person from each department and we want the number of involved researchers to be minimal. Write this problem as an Integer Programming problem. *(You do not have to solve the Integer Programming problem you obtained.)*

Solution

- This task is about transforming the above text into an Integer Programming problem format.
- The Integer Programming format is the following:
  - You define your variables first and explain what they mean: $\{x_1, \ldots, x_n\}$.
  - Then you list the constraints that the values of the variables must satisfy. Each constraint is a linear combination of the variables (so some variables, multiplied by some constant numbers and added together), which must be less than or equal to some constant number. It is important that you follow this format, since in real life we would be feeding this into a computer, which only understands this format.
  - So for example, if you have k number of constraints, then they will be:

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + \ldots a_{1,n}x_n &\leq b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \ldots a_{2,n}x_n &\leq b_2 \\ &\vdots \\ a_{k,1}x_1 + a_{k,2}x_2 + \ldots a_{k,n}x_n &\leq b_k \end{aligned}$$

  - The $a_{1,1} \ldots a_{k,n}$ and $b_1, \ldots b_k$ parameters are constants here.
  - Then, you give a target function, which should be maximized:

$$\text{maximize } c_1x_1 + c_2x_2 + \ldots c_nx_n$$

First, let's summarize the long text of the exercise in an easier to read format:

The input data is the following:

- There are k subprojects $Q = \{Q_1, \ldots, Q_k\}$.
- There are n researchers $R = \{R_1, \ldots, R_n\}$.
- There are 10 departments $D = \{D_1, \ldots, D_{10}\}$ (every researcher belongs to exactly one department, $R_i \in D_l$).
- The researcher R_i can take up on the subprojects $S_i \subseteq Q$.

And the following constrains and optimizations are required:

- Every subproject must have at least 3 researchers working on it.
- Every department must have at least 1 researcher that is working on the big project.
- Minimize the number of participating researchers.

The question is: Which researchers should be involved?

- The first step is figuring out what the variables should represent. When we translated the problem and gave it to the IP solver, the solver will report back an assignment for the variables. Based on this assignment, we should be able to answer the question, so we should be defining our variables, so that we can answer the question from them.
- A good choice would be to create a variable for every single researcher: variable x_i represents the researcher R_i, so we have n variables in total. We want to know whether that researcher is involved in the project or not, so these must be **binary variables**.
- In order to create binary variables, we have to make sure that the solver can only give the values 0 or 1 to them. This is done by using constraints:
- $\forall i, 1 \leq i \leq n$, we need constraints that will make sure that $0 \leq x_i \leq 1$. But we must stick to the less-than format as above, and the $0 \leq x_i$ part of the constraint is not in this format.
- Whenever we need to implement a greater-than constraint, we multiply it by -1, which turns the $\geq$ sign into a $\leq$. So let's multiply both sides of $0 \leq x_i$ by -1, to get $-x_i \leq 0$!

So far, our IP problem looks like this:

The variables are:

$x_1, x_2, \ldots, x_n$, where x_i represents the ith researcher.

- $x_i = 1$ means the researcher is working on the big project.
- $x_i = 0$ means they are not working.

The constraints are:

- All variables are binary:

$$x_i \leq 1 \qquad (1 \leq i \leq n)$$
$$-x_i \leq 0 \qquad (1 \leq i \leq n)$$

- Then we start adding constraints for every requirement listed.

Every subproject must have at least 3 researchers working on it.

- We go through each subproject in Q, and check which researchers are working on it. Researcher R_i is working on the subproject Q_j if Q_j is in S_i.
- How do we count the number of researchers working on Q_j? Well, add up all of the variables which represent researchers that can work on that subproject (remember: if Q_j is in S_i, it means that the ith researcher can work on the jth subproject):

$$\sum_{\forall i,\ Q_j \in S_i} x_i$$

- We need at least 3 workers on each project, so the constraint should be:

$$\sum_{\forall i,\ Q_j \in S_i} x_i \geq 3$$

- Again, we multiply by -1 to turn this into a less-than format as required:

$$\sum_{\forall i,\ Q_j \in S_i} -x_i \leq -3$$

- Finally, add this to our IP problem:

The variables are:

$x_1, x_2, \ldots, x_n$, where x_i represents the ith researcher.

- $x_i = 1$ means the researcher is working on the big project.
- $x_i = 0$ means they are not working.

The constraints are:

- All variables are binary:

$$\begin{aligned} x_i &\leq 1 && (1 \leq i \leq n) \\ -x_i &\leq 0 && (1 \leq i \leq n) \end{aligned}$$

- All subprojects must have at least 3 people working on them:

$$\sum_{\forall i,\ Q_j \in S_i} -x_i \leq -3 \qquad (1 \leq j \leq k)$$

Every department must have at least 1 researcher that is working on the big project.

- This is a very similar requirement, now instead of researchers on a subproject, we are talking about researchers in a department, and the count must be at least 1, instead of 3.
- Researcher i works in department l, if $R_i \in D_l$.
- The count of all researchers that are working on the project from the lth department:

$$\sum_{\forall i,\ R_i \in D_l} x_i$$

- And it must be at least 1:

$$\sum_{\forall i,\ R_i \in D_l} x_i \geq 1$$

- Flip it around as usual:

$$\sum_{\forall i,\ R_i \in D_l} -x_i \leq -1$$

- And add this to the IP problem as well.

The variables are:

$x_1, x_2, \ldots, x_n$, where x_i represents the ith researcher.

- $x_i = 1$ means the researcher is working on the big project.
- $x_i = 0$ means they are not working.

The constraints are:

- All variables are binary:

$$\begin{aligned} x_i &\leq 1 && (1 \leq i \leq n) \\ -x_i &\leq 0 && (1 \leq i \leq n) \end{aligned}$$

- All subprojects must have at least 3 people working on them:

$$\sum_{\forall i,\ Q_j \in S_i} -x_i \leq -3 \qquad (1 \leq j \leq k)$$

- All departments must have at least 1 person working on the big project in them:

$$\sum_{\forall i,\ R_i \in D_l} -x_i \leq -1 \qquad (1 \leq l \leq 10)$$

Finally, the optimization part:

Minimize the number of participating researchers.

- We must minimize the total number of researchers working.
- The total number of researchers working is just the sum of all variables:

$$\sum_{i=1}^{n} x_i$$

- So the target function should be:

$$\text{minimize} \sum_{i=1}^{n} x_i$$

- However, we can only use maximization as a target function. To turn minimization into maximization, again, we multiply by -1, so the minimal value of the positive sum turns into the maximal value of a negative sum:

$$\text{maximize} \sum_{i=1}^{n} -x_i$$

- Add this to our problem too:

The variables are:

$x_1, x_2, \ldots, x_n$, where x_i represents the ith researcher.

- $x_i = 1$ means the researcher is working on the big project.
- $x_i = 0$ means they are not working.

The constraints are:

- All variables are binary:

$$\begin{aligned} x_i &\leq 1 && (1 \leq i \leq n) \\ -x_i &\leq 0 && (1 \leq i \leq n) \end{aligned}$$

- All subprojects must have at least 3 people working on them:

$$\sum_{\forall i,\ Q_j \in S_i} -x_i \leq -3 \qquad (1 \leq j \leq k)$$

- All departments must have at least 1 person working on the big project in them:

$$\sum_{\forall i,\ R_i \in D_l} -x_i \leq -1 \qquad (1 \leq l \leq 10)$$

The target function is:

- Minimize the number of researchers working:

$$\text{maximize} \sum_{i=1}^{n} -x_i$$

Takeaways from this exercise

- Variables are defined based on what the question asks. Here the question asked which researcher should work on the project, which can be represented by binary variables assigned to each researcher.
- Constraints are defined based on the explicit constraints from the exercise (3 researcher per subproject, 1 researcher per department) and the implicit constraints that come out, for example to set the domain of the variables (binary).
- The target function represents the optimization task of the exercise. Here, we wanted to minimize the number of researchers working on the project, so this became the target function.
- To encode greater-than type of constraints, you multiply them by -1, which flips the $\geq$ into a $\leq$.
- To encode a minimization target function, you multiply it by -1, to turn it into a maximization target function.
- If you need to set an interval of possible values for a variable, e.g. $x_i \in [a, b]$, then the constraint $a \leq x_i \leq b$ is turned into $-x_i \leq -a$ and $x_i \leq b$.

15 Exam: 2020. 01. 23.

15.3 Exam: 2020. 01. 23., Exercise 3

Exercise

Let $\mathcal{A}, \mathcal{B}, \mathcal{C}$, be decision problems. Prove that if there exists a Karp-reduction $\mathcal{A} \prec \mathcal{B}$ of $O(n^2)$ time and Karp-reduction $\mathcal{B} \prec \mathcal{C}$ of $O(n^5)$ time, then there exists also a Karp-reduction $\mathcal{A} \prec \mathcal{C}$ of $O(n^10)$ time.

Solution

- The first thing we can realize is that we can run the first transformation algorithm from the first Karp-reduction and the second algorithm from the second on the output of the previous one to transform the input from $\mathcal{A}$ to $\mathcal{C}$.
- The only question is what is the runtime of this transformation?
- I will show you how I first tried to solve this, why I couldn't finish it that way and how I fixed it.

First try, the bad one

- If we have a Karp-reduction $\mathcal{A} \prec \mathcal{B}$ that runs in $O(n^2)$ time, it also means that whatever the output it gives, for an input of size n, it will also be of **size** $O(n^2)$, since you cannot output more than how many steps you are allowed to make.
- So there exists some function, let's call this $m(n)$ with which we can give an upper estimation on the size of the output of the first Karp-reduction.
- Due to that $O(n^2)$ limit, we know that $m(n) = c_0 n^2$ if $n > n_0$, for some c_0 and n_0 constants.
- Then, similarly due to the $\mathcal{B} \prec \mathcal{C}$ Karp-reduction, there exists some function, let's call it $k(m)$, which gives an upper bound on the estimation of the size of.
- Due to that $O(m^5)$ limit, we know that $k(m) = c_1 m^5$ if $m > m_0$, for some c_1 and m_0 constants.
- So we can put these two together and say that $k(m(n)) = c_1(c_0 n^2)^5 = c_1 c_0^5 n^1 0$ is $n > n_0$ and $m(n) > m_0$.
- This would be almost good, but there is a catch: due to $m(n) > m_0$, we would need a lower limit on $m(n)$, and we can only use a lower limit of n to achieve that, which we cannot really do, $m(n)$ can be arbitrarily small, we only know an upper limit of it.
- The whole issue comes from the n_0 and m_0 constants, so let's get rid of them!

Round 2

- If we have a Karp-reduction $\mathcal{A} \prec \mathcal{B}$ that runs in $O(n^2)$ time, it also means that whatever the output it gives, for an input of size n, it will also be of **size** $O(n^2)$, since you cannot output more than how many steps you are allowed to make.
- So there exists some function, let's call this $m(n)$ with which we can give an upper estimation on the size of the output of the first Karp-reduction.
- Due to that $O(n^2)$ limit, we know that $m(n) = c_0 n^2$ if $n > n_0$, for some c_0 and n_0 constants.
- Also, if $n < n_0$, a.k.a. 'the size of the input is limited by n_0', then the number of possible inputs is finite, since the alphabet is also finite. For a finite number of inputs, the finite number of outputs has some maximum size, let's call that S_n.
- Let's add this S_n number to $m(n)$! $m(n) = c_0 n^2 + S_n$. This is going to be an upper estimation for both $> n_0$ and $<= n_0$ input sizes, since either the left or the right hand side is going to be bigger than the resulting output size.
- We got rid of n_0! :)
- Do the same thing for $k(m)$, we get $k(m) = c_1 m^5 + S_m$.

- Then substitute in: $k(m) = c_1(c_0 n^2 + S_n)^5 + S_m = c_1 c_0^5 n^{10} + c_1 S_n^5 + S_m$.
- Then, just upper estimate again by multiplying the constant parts with n^10 as well: $k(m) = c_1 c_0^5 n^{10} + c_1 S_n^5 + S_m \leq c_1 c_0^5 n^{10} + c_1 S_n n^{10} + S_m n^{10} = (c_1 c_0^5 + c_1 S_n + S_m) n^{10}$. Choose $c = (c_1 c_0^5 + c_1 S_n + S_m)$ and $n_0 = 1$ and this proves that $k(n) \in O(n^{10})$.

16 Exam: 2022. 05. 30.

16.1 Exam: 2022. 05. 30., Exercise 1

Exercise

It is known that $f(n)$ is a non-negative monotone increasing function that satisfies $f(n) \in O(n^2)$. Does that imply $f(n^2 f(n) + 3f(n) + 5) \in O(n^4)$?

Solution

It does **not** imply!

- What does it mean that $f(n) \in O(n^2)$ implies $f(n^2 f(n) + 3f(n) + 5) \in O(n^4)$? It means that in all cases where $f(n) \in O(n^2)$ is true, $f(n^2 f(n) + 3f(n) + 5) \in O(n^4)$ must also be true. If we can find one example, where $f(n) \in O(n^2)$ is true, however $f(n^2 f(n) + 3f(n) + 5) \in O(n^4)$ is not, that will prove that the implication is false. This is called a counterexample.
- Be careful here: In general giving 2-3 examples is **not** a proving a statement (What about the infinite other examples where the statement could fail?). However, when the original statement is about "all things", for example "all apples are red" we can disprove this statement, by showing a single "green apple" (the counterexample). All apples cannot be red, since here is a green apple in my hand!
- Implication is exactly like this: a statement about all things, for example "If there are clouds in the sky, does it imply that it will rain today?". This is the same as saying "For every day, when there were clouds in the sky, it rained.". If you can show a single day, when there were clouds in the sky, but it did not rain, that breaks the implication. This day is the counterexample, that disproves the original statement.
- So we can give a counterexample here, for which even though $f(n) \in O(n^2)$ is true, $f(n^2 f(n) + 3f(n) + 5) \in O(n^4)$ is not. Let's use the simplest counterexample $f(n) = n^2 \in O(n^2)$, and it is also monotone increasing and non-negative (for $n \geq 1$)!
- $f(n^2 f(n) + 3f(n) + 5) = (n^2 f(n) + 3f(n) + 5)^2 = (n^4 + 3n^2 + 5)^2$
- To show that this is not $O(n^4)$, we can under-estimate the function and if even this under-estimate is not $O(n^4)$, then the original cannot be either:
- $(n^4 + 3n^2 + 5)^2 \geq (n^4)^2 \geq n^8$
- $n^8 \notin$, since there do not exist $c > 0$ and $n_0 \in N^+$ constants, for which $n^8 \leq cn^4$ for all $n > n_0$, since that would require $n \leq \sqrt[4]{c}$ to be true, for arbitrarily large n's.

Notes:

- We cannot say that either, that $f(n) \in O(n^2)$ implies that $f(n^2 f(n) + 3f(n) + 5) \in O(n^4)$ is not true!
- For example, take $f(n) = \sqrt{n}$, which is $\in O(n^2)$, monotone increasing and non-negative for $n \geq 1$.
- $f(n^2 f(n) + 3f(n) + 5) = \sqrt{n^2 f(n) + 3f(n) + 5} = \sqrt{\sqrt{n}^3 + 3\sqrt{n} + 5} \leq \sqrt{n}^3 + 3\sqrt{n} + 5 \leq \sqrt{n}^3 + 3\sqrt{n}^3 + 5\sqrt{n}^3 = 9\sqrt{n}^3 = 9n^{1.5} \in O(n^2)$, using $c = 9$ and $n_0 = 1$.
- So all we can say is that based on the information that $f(n) \in O(n^2)$, we cannot determine whether $f(n^2 f(n) + 3f(n) + 5) \in O(n^4)$ is true or false. It does not imply its trueness or faleness either.

16.2 Exam: 2022. 05. 30., Exercise 2

Exercise

Give a context-free grammar for the language

$$L_2 = \{a^{2n}b^n c^m d^{3m} \mid n \geq 1; m \geq 0\}$$

over alphabet $\{a, b, c, d\}$.

Solution

- We can see that the $a^{2n}b^n$ part and the $c^m d^{3m}$ are independent of each other, so we can introduce two separate variables to generate these.
- So we start by $S \to AB$.
- $A \to aaAb|aab$ will generate $a^{2n}b^n$, using the two to the left, one to the right method and $A \to aab$ makes sure that $n \geq 1$.
- $B \to cBddd|\varepsilon$ will generate $c^m d^{3m}$ using the similar method and $B \to \varepsilon$ specifically makes sure that $m \geq 0$.

So the final CF-grammar is:

$$S \to AB$$
$$A \to aaAb|aab$$
$$B \to cBddd|\varepsilon$$

Notes:

- Whenever you have to generate something in the form of $a^i b^j$, where i and j are in some relationship with each other, e.g. $i = j$, $i = 2j$, $i < j$, and so on, the only viable way is to use the "one (some) to the right one (some) to the left" method, since this makes sure that in evey generation step, the required relationship always remains true.
- If we tried to do this using 4 variables instead of 2, we will inevitably fail. For example solutions such as this are incorrect:

$$S \to ABCD$$
$$A \to aA|a$$
$$B \to bB|b$$
$$C \to cC|\varepsilon$$
$$D \to dD|\varepsilon$$

- These rules can generate a's, b's, c's and d's completely independently from each other, so we will never be able to force the requirements $(\#a) = 2(\#b)$ and $3(\#c) = (\#d)$. This means we generate a bunch of false positives (words that are not in the language, but get generated by this grammar).
- A solution such as this is still incorrect:

$$S \to ABCD$$
$$A \to aaA|aa$$
$$B \to bB|b$$
$$C \to cC|\varepsilon$$
$$D \to dddD|\varepsilon$$

- Since this generates the language $a^{2i}b^j c^k d^{3l}$, but it cannot make sure that $i = j$ and $k = l$, so we will still have a lot of false positives.

16.3 Exam: 2022. 05. 30., Exercise 3

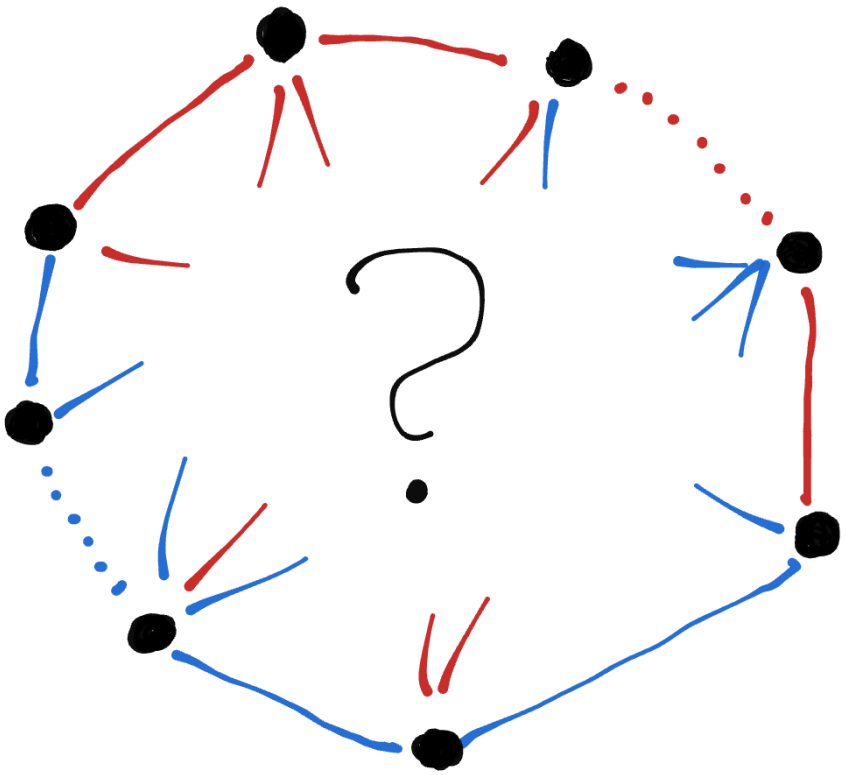
Exercise

Consider the following decision problem. We are given an undirected connected graph G of even number of vertices, whose edges are colored red or blue. We want to decide whether there exists a Hamiltonian cycle in G that consists of a red path and a blue path ofthe same length, that is, going along the Hamiltonian cycle, the red edges form a path and the blue edges form a path and those two paths have the same number of vertices. Prove that this problem is NP-complete.

Solution

I belive this exercise caused the most problems and misunderstandings on the exam, so let's break down and understand what we are being asked for here.

- We are given decision problem, let's name its language $HAMRB$, for Hamiltonian-cycle, red-blue version.
- The input is a graph here, which already has its edges colored in for us in red and blue colors. The coloring is given, we cannot modify this. (Many students did try to color the edges.)
- We are asked the question, whether there is a Hamiltonian-cycle (a cycle that contains all vertices of the graph) in this graph, which has a "red edged half" and a "blue edged half", like so:



- It is easy to see this structure in the graph now, because the vertices are laid out or us in a nice way, in the correct order, however when you are given a graph, it is difficult (we'll prove soon that it's NP-complete :)) to find this special cycle.
- In order to show that this problem is NP-complete, we must show that it is both in NP and NP-hard.

Step 1: Showing that HAMRB $\in$ NP

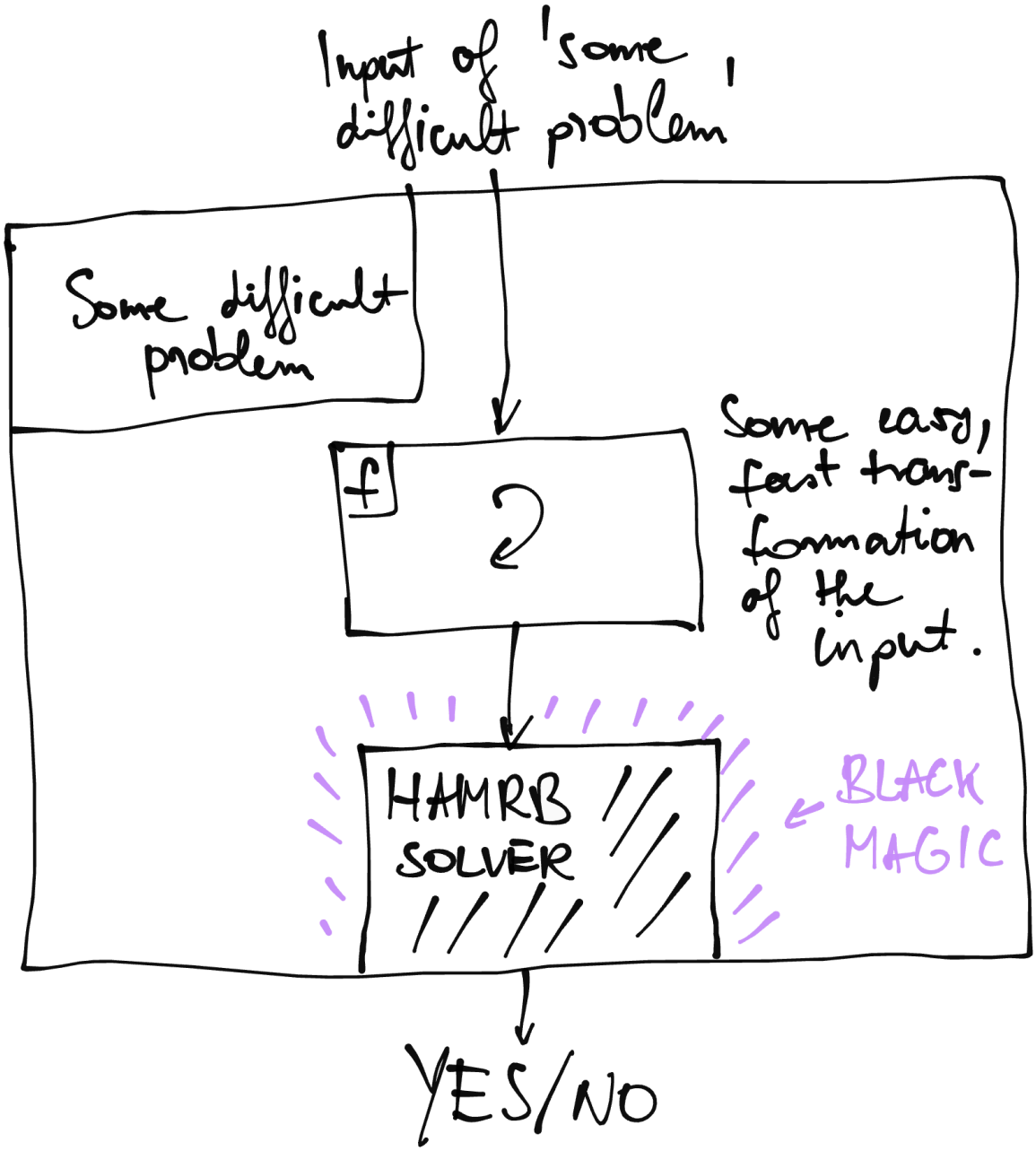
- To show that a language is in NP, we use the Witness Theorem.
- To use the Witness Theorem, we have to give the following things for $HAMRB$:

  - A witness:
  - A witness must exist for every input in $HAMRB$, every colored edged graph, which has this type of red-blue Hamiltonian-cycle in it.
  - A witness like this cannot exist for any input that is not in $HAMRB$, so if it does not have type of red-blue Hamiltonian-cycle in it.
  - The size of the witness must be polynomial in relation to the input size.
  - A witness checking algorithm:
  - Must be able to verify, that the witness is correct. We do not trust the person that gives us the witness, but based on the witness we can check whether the input is in $HAMRB$ or not.
  - Usually the witness is basically the "solution" of the problem, think of how solving a Sudoku-puzzle is difficult, but if someone fills out a Sudoku table for us, it is easy to verify if the solution is correct. In this case the witness is the filled-out Sudoku-table, while the input is the initial table.
  - This must run in polynomial time relative to the input and witness size.

- In this case, the witness (the solution to the problem) is going to be the list of all of the vertices of the graph, in the order in which they are present in the cycle. E.g. $\{v_1, v_2, \dots v_n\}$. And the starting vertex should be the first red edge, so it's easier to check.
- Given this ordering, we can verify that this is indeed a correct solution, since we can check the following:
  - All vertices of the graph are present.
  - All cycle edges exist, so there is an edge between vertices v_1 and v_2, between v_2 and v_3, between v_3 and v_4, and so on, v_{n-1} and v_n and finally v_n and v_1.
  - The first half of the checked edges above are red, the second half of the checked edges above are blue.
- The witness' size is $O(n \log_2 n)$, since there are n vertices and one of them can be represented by its index, so a number in a binary format, that requires $log_2 n$ bits. This is polynomial relative to the $O(n^2)$ input size (adjacency matrix with coloring information).
- The witness checking algorithm did a for loop on all vertices 3 times, which makes it polynomial.

Step 2: Showing that HAMRB is NP-hard

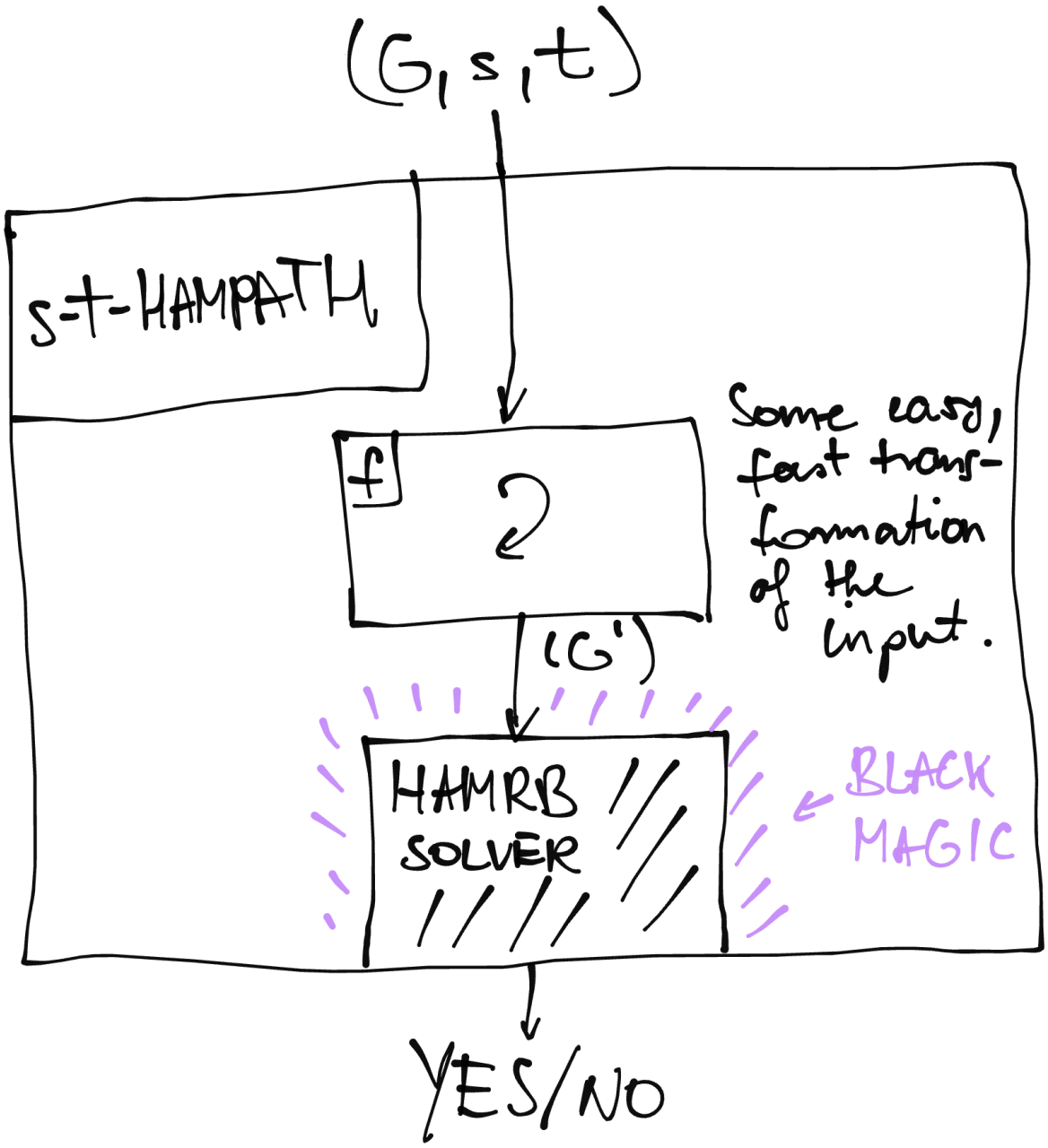
- In order to sho that $HAMRB$ is NP-hard, we have to Karp-reduce another NP-hard (NP-complete) language onto it. Please read Section 8.1, What is a Karp-reduction? to understand what a Karp-reduction is and continue with the next item after you're done.
- What's happening now is this: in order to show that $HAMRB$ is a difficult problem to solve, we are going to show, that if we had a solution to $HAMRB$, then we could use that algorithm to solve another well-known difficult problem. So the solution to $HAMRB$ is also a 'big achievement' to figure out.
- The hardest part here is figuring out which well-known difficult problem could be solved with the $HAMRB$ solver.



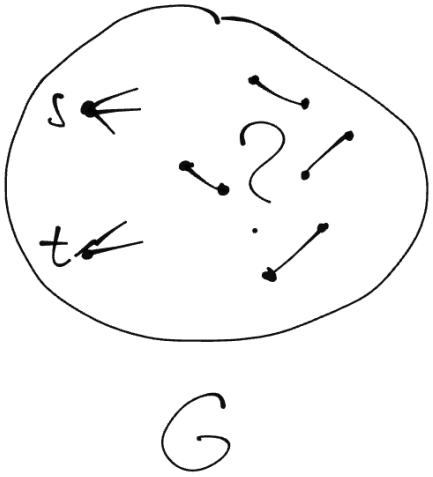
- Luckily, we have only studied a few of these problems, namely (I'm giving the decision version of the problems, which can all be turned into optimization problems):

SAT(Φ)	Given the Φ boolean formula, can it be satisfied, e.g. is there an assigment of the variables in it for which the formula evaluates to true?
3-SAT(Φ), 4-SAT(Φ), ..., n-SAT(Φ), where $n \geq 3$	Given a (Φ) formula of Boolean-variables, which is specifically in conjunctive normal form, where a term consists of exactly n variables, can this formula be satisfied? Be careful, 2-SAT is in P!
MAXINDEP(G, k)	Is there a set of independent vertices of size (at least) k in G?
MAXCLIQUE(G, k)	Is there clique of size (at least) k in G?
3-COLOR(G), 4-COLOR(G), ..., n-COLOR(G), where $n \geq 3$	Can G's vertices be properly colored using 3 colors? (So no two vertices of the same color have an edge running between them.) Be careful, 2-COLOR is in P!
HAM(G)	Does graph G contain a Hamiltonian cycle (a cycle that contains all vertices of G)?
HAMPATH(G)	Does graph G contain a Hamiltonian path (a path that contains all vertices of G)?
s-t-HAMPATH(G, s, t)	Does graph G contain a Hamiltonian path that starts specifically in vertex s and ends in vertex t?
TSP(G, n)	Given a complete weighted graph G (distances of cities), is there a Hamiltonian cycle with a weight of at most n?
SUBSETSUM($a_1, a_2, \ldots a_n, t$)	Is there a subset of the integers $a_1, \ldots a_n$ that sum to exactly t?
PARTITION($a_1, a_2, \ldots a_n$)	Can you partition the integers $a_1, \ldots a_n$ into exactly two sets, such that their sum is equal?
IP(A, b, c^T, k)	Integer programming: Is there an x integer vector for which $k \leq c^T x$, while $Ax \leq b$? Be careful, LP (linear programming) is in P!
BINPACKING($s_1, s_2, \ldots s_n, k$)	Can we place all of the s_i items of size $0 \leq s_i \leq 1$ into at most k number of size 1 bins?
KNAPSACK($s_1, s_2, \ldots s_n, v_1, v_2, \ldots v_n, S, V$)	Can we select a subset of the items i, so that their size s_i sums to at most S, while their value v_i sums to at least V?

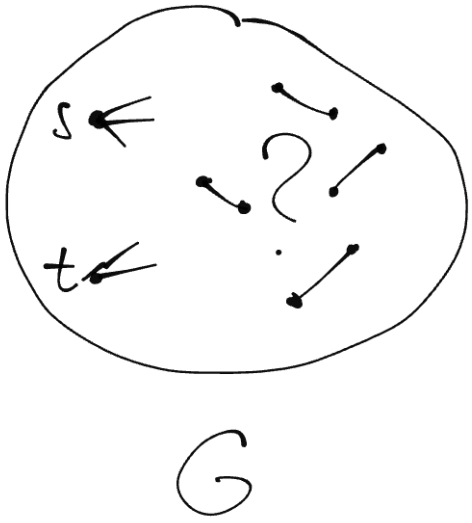
- The task here is to select one from the above and try to transform its input into something that we can give to $HAMRB$ and whenever $HAMRB$ returns yes, that means that the answer to the original problem is also yes, and same for no.
- It is a good idea to start looking around the problems that are similar to the one being asked at first, and when we have no success, then try to move onto different ones.
- In this case HAM, HAMPATH and s-t-HAMPATH are the ones that are similar.
- Many students have tried to use HAM here, but I believe that one would be more tricky to figure out (remember, if one of them can be Karp-reducted, then all Karp-reductions exist, all NP-complete languages can be Karp-reduced to all others, however some of them are harder to figure out (or can be done indirectly)!).
- I believe the best choice here is s-t-HAMPATH, because the s and t vertices are fixed, which will make it easier for us to define a coloring of the transformed graph's edges (you'll see why)!
- So the Karp-reduction we are doing is $s - t - HAMPATH \prec HAMRB$:

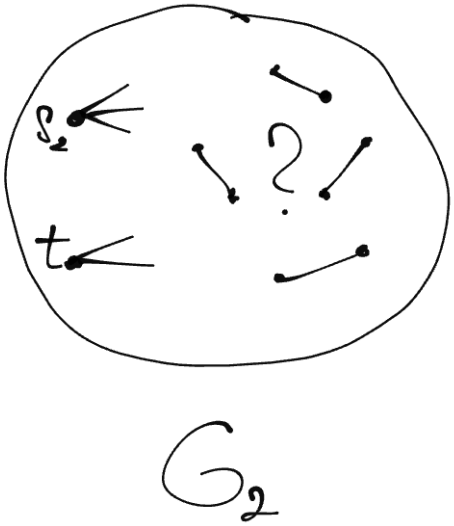


- Now we need to figure out what f is.
- The biggest problem I saw here in the exam papers is that many students thought that we know where the s-t-HAMPATH (or in the case of $HAM \prec HAMRB$, where the Hamiltonian-cycle) is. This is an incorrect assumption, we do **not** know where the s-t-HAMPATH is in G! We need to figure out if it exists or not, using the "blackbox" solution to $HAMRB$. We will transform the input (G, s, t) into a G' graph, which has a red-blue-Hamiltonian cycle exactly when G has an s-t-HAMPATH.
- How do we do this:
- While we do not know where the full s-t-HAMPATH is going, we do know the first and last vertex on it, s and t.
- We begin with the input we receive for s-t-HAMPATH: (G, s, t):

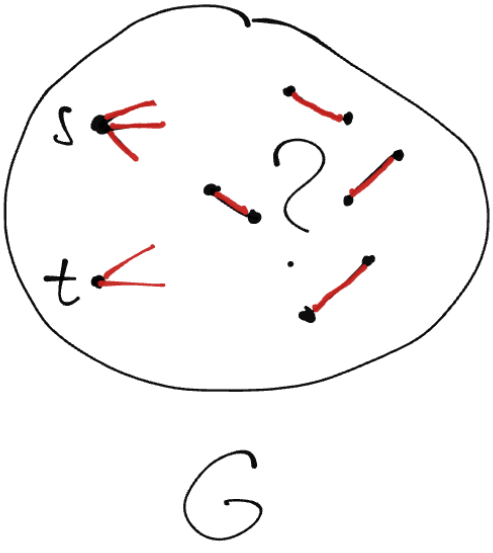


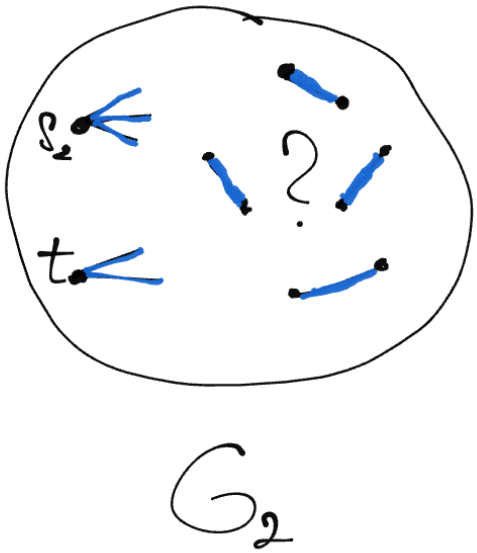
- We are going to create a copy of G, like so:



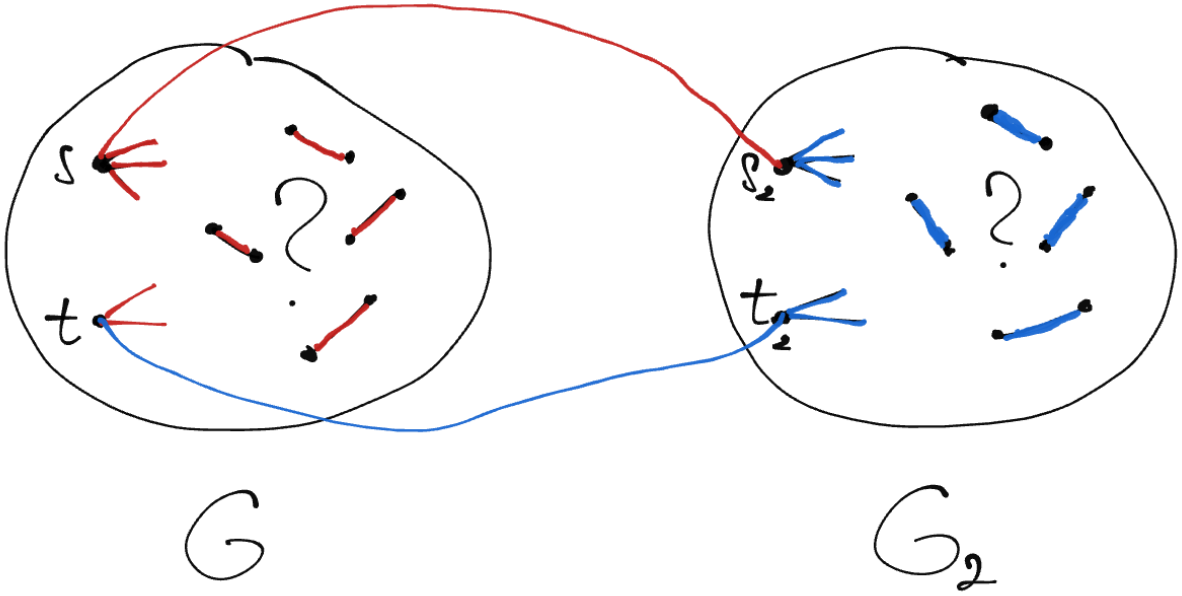


- Now we will color all of the edges in G red and all of the edges in G_2 blue. **Notice here how we don't need to know where the HAMPATH is!**

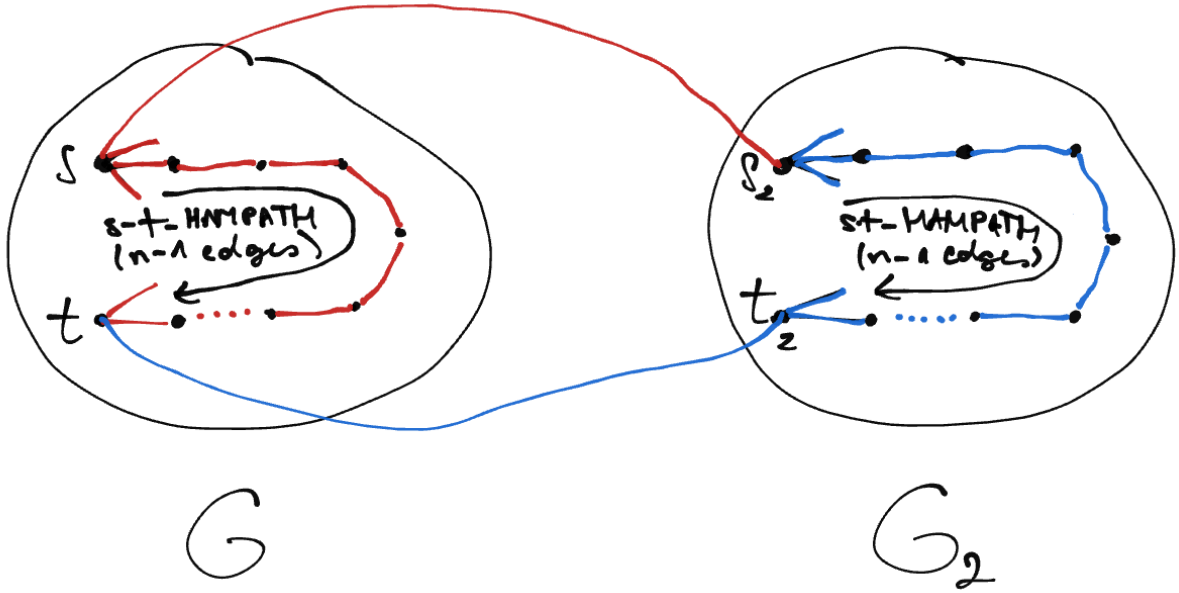




- Finally, we will connect s and s_2 with a red edge, while we will connect the t and t_2 with a blue edge:



- Now. Let's see what we have created here.
- If G had an s-t-HAMPATH, what did that turn into?

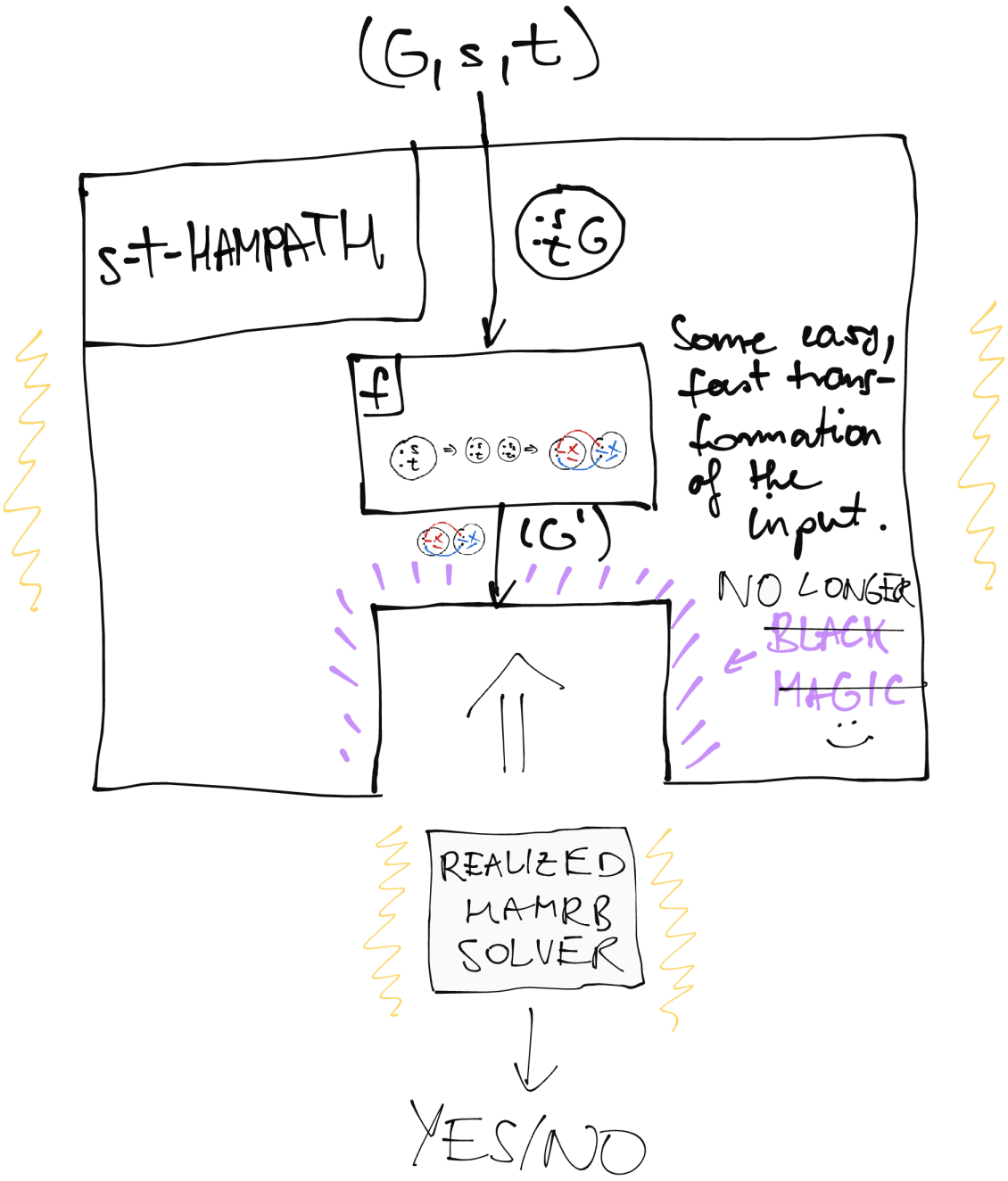


- The s-t-HAMPATH was copied to G_2 as well, G's s-t-HAMPATH was colored red, while G_2's s-t-HAMPATH was colored blue.
- A Hamiltonian-cycle has emerged in this $G \cup G_2$ graph: Follow along the $(n-1)$ red edges, then use $s - s_2$, then follow along the $(n-1)$ blue edges, finally use the edge $t_2 - t$.
- This is exactly the type of red-blue cycle that HAMRB says yes to! So because the original G graph had an s-t-HAMPATH in it, this transformation turned that into a red/blue cycle that HAMRB says yes to (the number of the red and blue edges is the same and also the number of the total vertices in the transformed graph is $2n$.)
- Also, we can see that if there is no s-t-HAMPATH in the original G graph, then there is no way for HAMRB to find a red/blue HAM cycle in $G \cap G_2$, since the only possible way to connect between G and G_2 is to use the edges $s - s_2$ and $t - t_2$, so the only way for HAMRB to find a HAM cycle in $G \cap G_2$ at all, is to have G's vertices lined up between s and t on a graph, and similarly for G_2 between s_2 and t_2, which is the s-t-HAMPATH we are looking for.

- Finally, the transformation function must also be polynomial runtime. Here we duplicated G, and added two more edges, which can be done in polynomial time, so we are done with the proof.

Notes:

- After we have completed this proof, if someone comes up with a polynomial solver for HAMRB, what can we do with it?
- We can insert that into our "Karp-machine", to the missing place, which will turn it into a complete polynomial solver for s-t-HAMPATH no additional steps required!



16.4 Exam: 2022. 05. 30., Exercise 4

Exercise

Let language A be contained in coNP, while language B be NP-complete. Does that imply

a) there exists Karp-reduction $\overline{A} \prec B$?

b) there exists Karp-reduction $A \prec \overline{B}$?

($\overline{X}$ denotes the complement of language X.)

Solution

- If A is in $coNP$, then $\overline{A}$ is in NP. This is the definition of the coNP set, it contains the complementers of the languages in NP.
- B is NP-complete, which means it is in NP and also NP-hard. The definition of NP-hard is that there exists Karp-reductions from all languages in NP onto B.
- So this means the $\overline{A} \prec B$ Karp-reduction must exist, since $\overline{A}$ is in NP.
- If an $X \prec Y$ Karp-reduction exists, so does $\overline{X} \prec \overline{Y}$ too, since we can use the same f transformation function to transform their inputs. See Session 8, Exercise 3 for a similar exercise.
- Since $\overline{A} \prec B$ exists, as we have just shown, then so does $\overline{\overline{A}} \prec \overline{B}$, where $\overline{\overline{A}} = A$, so $A \prec \overline{B}$ exists too.

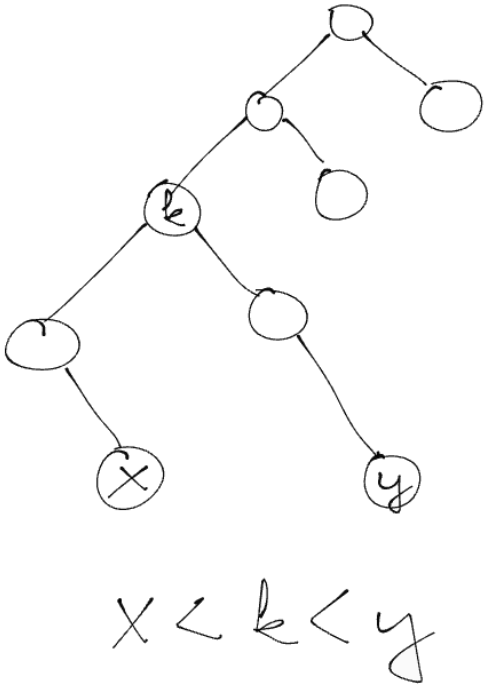
16.5 Exam: 2022. 05. 30., Exercise 5

Exercise

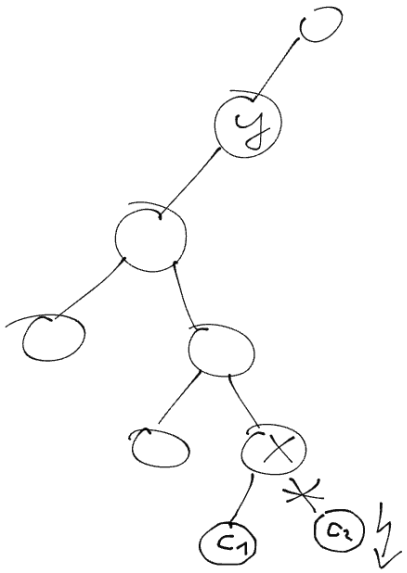
Pairwise distinct keys are stored in a binary search tree. It is known that node x of the tree has two children. How many children the node y can have, whose key is the successor of the key of x in the ordering of the keys?

Solution

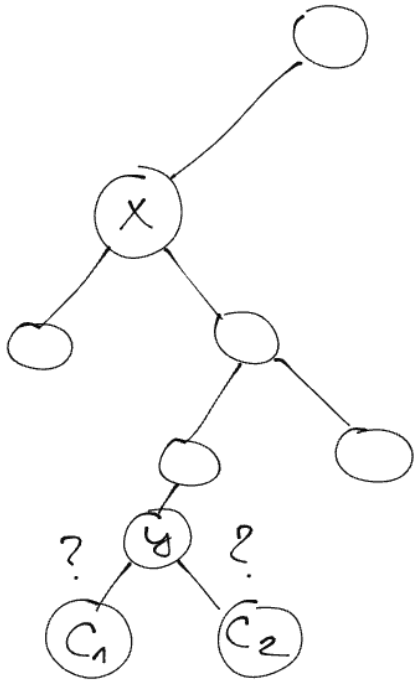
- Successor means that $x < y$ and x is immediately followed by y, with nothing between the two.
- Since x and y immediately follow each other, one of them has to be the ancestor of the other one. If they are not ancestor of each other, then if we take their first common ancestor, e.g. k, x and y will be on different sides of k. Which would mean that k comes between x and y in the order, so they couldn't be successors.



- Either x is an ancestor to y or y is an ancestor to x.
- If y is the ancestor of x, then:



- Then x cannot have a right child c_2, since that would come between x and y: $x < c_2 < y$. But the task says that x has two children, so this case cannot happen.
- This means that x is the ancestor of y:



- y must be on the righthand side of x, since $x < y$, however it must be the leftmost child in that substree, since nothing can come between x and y.
- y cannot have a lefthand side child c_1, since that would come between x and y, $x < c_1 < y$, which would make x and y not be successors.
- y can have 0 children or 1 child (the right child), but not 2 children.

16.6 Exam: 2022. 05. 30., Exercise 6

Exercise

We are given integer numbers $a_1, a_2, \ldots, a_n$. Give an algorithm, that uses $O(n^2 \log n)$ comparisons, which decides whether there exist 4 pairwise distinct numbers $a, b, c, d \in \{a_1, a_2, \ldots, a_n\}$ that satisfy $a - b = c - d$.

Solution

- As a zeroeth step we remove all duplicate numbers. This can be done, by storing the original index of all numbers, sorting, then copying the array to a new one by skipping over the duplicates, then ordering them according to their stored original indexes. All done in $O(n \log n)$ time. This step is needed for later, I will explain where when it comes up. If there exists at least 2 pairs of duplicates, then their difference (both 0) satisfies the condition, we return them (detect them during duplicate removal) and stop. Otherwise, there is at most one duplicate pair / three copies of a number, which we all removed here.
- First we calculate the difference for all pairs of a_i and a_j: $a_i - a_j$.
- This is stored in an n^2 sized array, filled up in $O(n^2)$ time. We store the additional information of which i and j indexes the items came from.
- In this new array we need to find 2 numbers that are the same, and they came from 4 completely different i and j indexes. To find equal numbers in an array, we can sort it first:
- Then, we stort this array using a merge sort (a comparison based sort that runs in $O(n \log n)$ comparisons for an $O(n)$ sized array). For a size $O(n^2)$ array, that is $O(n^2 \log n^2)$, where $\log n^2 = 2 \log n$, so this is actually $O(n^2 \log n)$ comparisons.
- Now the equal numbers are arranged next to each other. We can read this sorted array from left to right:
- For every item, we check ahead to see if there is an equal number next to it. However, it can happen that one of the original indexes in the upcoming number is the same as one of the original indexes in the current number. This means that they overlap with each other and we can't choose them. We need to keep looking forward until we see the same number and checking if their original indexes are different to the original number we are checking right now.
- Luckily, we need to look ahead to at most 2 places. This is because we removed the duplicates in the zeroeth step, so either one or the other original index matches, in both cases only one number exists, that can satisfy that condition.
- This is lucky, because to look the array over, we need to look at n^2 items, so we cannot allow an inner loop of another n^2 steps. This makes the runtime of this step $O(n^2)$.
- Finally, if we find a unique pair of duplicates, we return them.

16.7 Exam: 2022. 05. 30., Exercise 7

Exercise

Sequence $b_1, b_2, \ldots, b_n$ consists of nonnegative integers. We want to determine the largest sum of monotone increasing subsequences. For example, if the sequence is $2, 1, 10, 6, 3, 8, 4, 9$, then the largest sum is 25 given by the subsequence $2, 6, 8, 9$. Give an $O(n^2)$ algorithm that solves this problem. (It is assumed hat arithmetic operations with arbitrary sized numbers can be done in 1 step.)

Solution

- This is a dynamic programming problem.
- We need to decompile it into subproblems, which we can conveniently put back together to solve the next subproblem.
- The first subproblem we could come up with, is maybe what is the largest sum we can achieve in the subarray $b_1, \ldots b_i$.
- However we run into a problem here: How do we move forward and update this sum when the next b_{i+1} number comes? Since we do not know exactly at which element the subsequence ended, only that it was somewhere in $[1, i]$, we cannot check whether it is an increasing subsequence or not.
- This is why the subproblem we define must contain not only what sum we could achieve so far, but exactly what the last element of the subsequence was that achieved this sum.
- So we define the following 1D array: $T[i]$ is the largest sum of subsequence we can achieve, for which the subsequence's last element is b_i.
- The update here is as follows $T[i] = \max_j\{T[j] + b_i \mid j < i \text{ and } b_j < b_i\}$: so we take all b_j's that were before b_i and where adding b_i to that subsequence would still be an increasing subsequence we add b_i to $T[j]$ (the current sum) and then take whatever the maximum of these possibilities is.
- The initial step is $T[1] = b_1$, since that is the only thing we can achieve with a sequence of 1 element. Also let's define $T[0] = 0$, and $b_0 = -\infty$, this is so any time the b_i is not larger than anything in $b_1, \ldots, b_{i-1}$, it can still start its own subsequence.
  - Here you could also define the max function to return 0 for an empty set and then add b_i outside of the function.
- The T table must be filled in the order from left-to-right, so all the necessary cells are already filled when we reach the current cell.
- Finally, the solution is the maximum over T, since a maximum subsequence is allowed to end anywhere.
- The size of the T table is n, for every cell filling, we take at most $O(n)$ steps, and taking the maximum can be done in linear time, so the runtime is $O(n^2 + n) = O(n^2)$.

17 Exam: 2022. 06. 13.

17.1 Exam: 2022. 06. 13., Exercise 1

Exercise

Prove, that $\sqrt{n}^{\sqrt{n}} \in O(2^n)$.

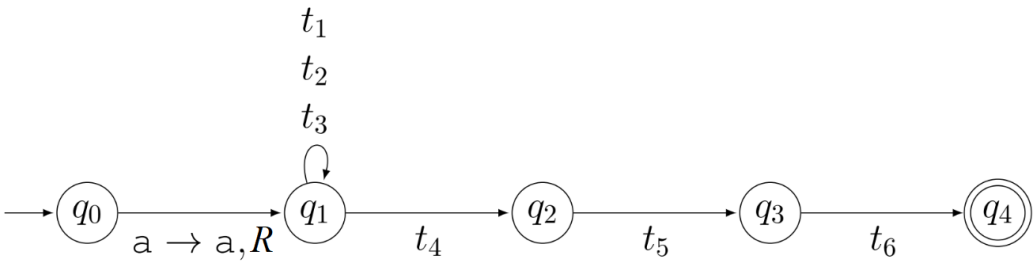
Solution

- The first step is to make the $\sqrt{n}^{\sqrt{n}}$ and the 2^n comparable. The problem currently is, that the 2^n has a constant for a base, while the $\sqrt{n}^{\sqrt{n}}$ has a non-constant for a base.
- We can apply the logarithmic identity $a = 2^{\log_2(a)}$ to $\sqrt{n}$, to receive $\sqrt{n} = 2^{\log_2 \sqrt{n}}$.
- So $\sqrt{n}^{\sqrt{n}} = (2^{\log_2 \sqrt{n}})^{\sqrt{n}} = 2^{\sqrt{n} \log_2 \sqrt{n}}$.
- Now we need to find $c > 0$ and $n_0 \in \mathbb{N}$, such that $2^{\sqrt{n} \log_2 \sqrt{n}} \leq c \cdot 2^n$, $\forall n \geq n_0$.
- Since $\log_2$ is a **monotone increasing** function, we can apply that to both sides of the above inequality, so that the inequality still holds true.
- $\log_2(2^{\sqrt{n} \log_2 \sqrt{n}}) \leq \log_2(c \cdot 2^n)$.
- $\sqrt{n} \log_2 \sqrt{n} \leq \log_2(c) + \log_2(2^n)$.
- $\sqrt{n} \log_2 \sqrt{n} \leq \log_2(c) + n$.
- The c part is not going to be helpful to us, since it is no longer multiplying the n, just adding a constant value to it, so we just want to get rid of it at this point. Assign $c = 1$, so $\log_2(c) = 0$.
- $\sqrt{n} \log_2 \sqrt{n} \leq n$.
- Divide by $\sqrt{n}$.
- $\log_2 \sqrt{n} \leq \sqrt{n}$.
- We know this is true, since we have studied in the lectures that $\log_2(x) \in O(x)$, with $c = 1$ and $n_0 = 1$. In this case we substitute $x = \sqrt{n}$.
- So $n_0 = 1$, and the previously set $c = 1$ are a good choice.

17.2 Exam: 2022. 06. 13., Exercise 2

Exercise

We would like to create a single tape, **deterministic** Turing-machine, that recognizes the language $L = a(a+b+c)*bc$. On the following diagram, we can see the Turing-machine's structure, with missing transitions, labeled t_1, t_2, t_3, t_4, t_5. Give the transitions, so that the Turing-machine recognizes L and explain how the machine works!(t_1, t_2 and t_3 are parallel edges in the TM.)



Solution

- The $q_0 \to q_1$ transition makes sure that the first a character from $L = a(a+b+c)^*bc$ is present.
- The t_1, t_2, t_3 loop is important, because that's the only way we can deal with the $*$ operator on $(a+b+c)^*$. The loop can execute any number of times (including zero), similarly the star operator means the inner part can be repeated any number of times (including zero).
- Inside $(a+b+c)^*$, we have $a+b+c$, which means either a, or b or c character is present.
- The loop is allowed to have 3 transitons, so we can use 1 transition to deal with one of the three characters.
- So $t_1 = a \to a, R$, $t_2 = b \to b, R$, $t_3 = c \to c, R$. It is important to move the head to the right, otherwise we would be stuck in the same place, reading in the same character over and over again, resulting in an infinite loop, which would mean the TM rejects the input word and we don't want that.
- Finally, we need to make sure that the word ends in bc, since $L = a(a+b+c)^*bc$.
- This part many people wrote $t_4 = b \to b, R$. Unfortunately, this is incorrect, because it collides with the t_2 transition. Both t_2 and t_4 can be executed when the head sees a b character on the tape, resulting in nondeterministic behaviour. The task asked to give a deterministic Turing-machine, so this is not good.
- Instead, we are going to allow the t_2 and the t_3 transition to read the b and c at the end of the word in. While we cannot guarantee at this point that the word actually ended in b and c, the t_1, t_2 and t_3 transitions will move the head past whatever characters are at the end, until we see a $*$ on the tape.
- We are going to use t_4 to detect this $*$, so $t_4 = * \to *, L$. We are moving to the left here, because we want to go back and make sure the word ends in bc.
- Now we will encounted the ending c character first, so $t_5 = c \to c, L$.
- Finally we make sure the previous character is b, so $t_6 = b \to b, L$.

17.3 Exam: 2022. 06. 13., Exercise 3

Exercise

Prove, that if $MAXCLIQUE \in P$ is true, then $SAT \in P$ is also true!

Solution

- Both $MAXCLIQUE$ and SAT are NP-complete languages.
- Many people tried to prove that $MAXCLIQUE \in P$ is true. This is not what the task asks (and it would be quite unfair to ask this from you, since **nobody in the world can prove or disprove this**). The task is asking to **assume** that $MAXCLIQUE \in P$ is true and **using this assumption** prove that $SAT \in P$ is also true.
- It is highly unlikely that you can prove that $MAXCLIQUE \in P$ or $SAT \in P$ without any assumption. Researchers have been trying to prove this true or false for years, with no success. If someone could prove this (without an assumption), that would lead to $P = NP$, which is one of the Millenium Prize Problems.
- The reason why it would lead to $P = NP$ is exactly what we are going to use now: The fact that if $MAXCLIQUE \in P$, then everything in NP (including SAT) will be in P, meaning $NP \subseteq P$. $P \subseteq NP$ is true already, leading to $P = NP$.
- Now let's show specifically for SAT, that if $MAXCLIQUE \in P$, then SAT, another NP-complete language is also in P!
- SAT is NP-complete, which means that it is both in NP and NP-hard. Let's remember the fact that SAT is in NP.
- $MAXCLIQUE$ is also NP-complete, so also in NP and NP-hard. Let's remember the fact that $MAXCLIQUE$ is NP-hard.
- The definition of NP-hard means that all Karp-reductions from all languages in NP must exist. Including the Karp-reduction **from** SAT **to** $MAXCLIQUE$, denoted by $SAT \prec MAXCLIQUE$. (While the Karp-reduction does exist the other way around as well, this is the order that will be useful to us in a second!)
- The fact that a Karp-reductin exists **from** SAT **to** $MAXCLIQUE$ means, that SAT can be solved by applying a polynomial time transformation to its input, then feeding that to whatever solver we have for $MAXCLIQUE$.
- Since we have assumed that $MAXCLIQUE$ is in P, this means that we do have a polynomial time solver for $MAXCLIQUE$.
- So putting these together: SAT can be solved by applying a polynomial time transformation to its input, then feeding that to the polynomial time solver we have for $MAXCLIQUE$.
- This means, that we have just solved SAT in polynomial time!
- This means that SAT is in P!
  - But **only if we make the assumption that** $MAXCLIQUE$ **is in** P!

17.4 Exam: 2022. 06. 13., Exercise 4

Exercise

We are given a sum of positive integers $a_1 + a_2 + \ldots + a_n$. We can change any addition symbol into a multiplication symbol, if they don't already have a neighbouring multiplication symbol. (So any number can be a part of at most one multiplication.) Give a $O(n)$ runtime algorithm, that returns the maximum possible value of the expression above! For example, from the sum $1 + 4 + 3 + 2 + 3 + 4 + 2$, the maximum possible value is $29 = 1 + 4 \cdot 3 + 2 + 3 \cdot 4 + 2$.

Solution

- The naive algorithm here would be to check all possible addition and multiplication sign placements, check that that is a valid placement (no neighbouring multiplication signs), calculate the value of that assignment and remember the maximum we have seen at all times. There are $(n-1)$ places where a multiplication or addition sign can be, so without accounting for the other checks and remembering stuff, this is already looking out to be 2^{n-1} possible assignments, which is exponential.
- The question is: How do we make this not exponential?
- I've received many heuristic solutions to this problem. A heuristic solution is something that does not account for all 2^{n-1} possible assignments, but only checks a subset of these. For example:
  - Let's try changing the first + sign into a $\cdot$. Check what the current value is: $a_1 \cdot a_2$.
  - Then let's move forward and try to change the second + sign into a $\cdot$ (so now the first + must stay a +): $a_1 + a_2 \cdot a_3$.
  - Let's check which one is bigger: $a_1 \cdot a_2 + a_3$ or $a_1 + a_2 \cdot a_3$?
  - Move forward with whichever is bigger ... $\rightarrow$ ERROR: HEURISTIC DETECTED!
- The problem in this solution is that we are making **locally optimal choices** that remove possible solutions from the available 2^{n-1} sized solution pool that should not be removed.
- Just because based on the first 3 values in the array a, we think it is better to choose the $a_1 + a_2 \cdot a_3$ assignment over the $a_1 \cdot a_2 + a_3$ assignment, we cannot make this assumption, since choosing $a_1 + a_2 \cdot a_3$ also means $a_3 \cdot a_4$ cannot be choosen, so we just thrown out possible solutions!

Whenever we are faced with an exponentially large solution pool, it should pop into our heads, that this could be a task that could be solvable using **dynamic programming**. Dynamic programming is exactly the algorithmic design technique, that can make something exponential into something fast! So let's try that!

- I like to figure out the dynamic programming solution to a problem, by working backwards, instead of forwards.
- This is going to help, because DP is essentially smart recursion, and whenever we are trying to figure out a recursive formula, we need to work backwards.
- In this case, I will look at the very last + sign, instead of the first one and make the following **key statement**: **This sign (in the optimal solution) is either going to be a + or a $\cdot$!**
- If the sign is a +, then the globally optimal solution is whatever the optimal solution is to the problem $a_1 + \cdots + a_{n-1}$, PLUS the a_n value. Let's denote this by

$$OPT(a_1 + \cdots + a_n) = OPT(a_1 + \cdots + a_{n-1}) + a_n$$

  .
- If the sign is a $\cdot$, then the previous sign **canot be a** $\cdot$ as well, since that would be two neighbouring $\cdot$'s.
- So in this latter case, the globally optimal solution is whatever the optimal solution is to the problem $a_1 + \cdots + a_{n-2}$, then a + and a $\cdot$ sign: $+a_{n-1} \cdot a_n$. Or

$$OPT(a_1 + \cdots + a_n) = OPT(a_1 + \cdots a_{n-2}) + a_{n-1} \cdot a_n$$

  .

- So going back to our **key statement**: **The last sign in the optimal solution is either going to be a $+$ or a $\cdot$!** resulted in:

$$OPT(a_1 + \cdots + a_n) \stackrel{?}{=} \begin{cases} OPT(a_1 + \cdots + a_{n-1}) + a_n \\ OPT(a_1 + \cdots a_{n-2}) + a_{n-1} \cdot a_n \end{cases}$$

- The question is, how do we know which one is **the** optimal solution between these two? It's simple: **the one that maximizes the value**.
- So

$$OPT(a_1 + \cdots + a_n) = \max(\ OPT(a_1 + \cdots + a_{n-1}) + a_n,\ OPT(a_1 + \cdots a_{n-2}) + a_{n-1} \cdot a_n\ )$$

.

- If we stored the optimal solutions in an array, called T, where $T[i] = OPT(a_1 + \cdots + a_i)$ (so the optimal solution up to the number a_i), then the above turns into:

$$T[n] = \max(\ T[n-1] + a_n,\ T[n-2]) + a_{n-1} \cdot a_n\ )$$

- And there you have it: the recursive formula for a DP problem.

When you have the recursive formula, you only need to fill in the gaps of everything else, this task is going to be easy once you have the formula on hand. It involves 6 steps:

- **Step 1**: Define the T DP table. What does $T[i]$ mean?
  - Let T be a 1 dimensional array, of length n. $T[i]$ means the maximum value achieveable to the subproblem $a_1 + \cdots + a_i$, while not allowing two neighbouring $\cdot$'s in the subproblem either.
- **Step 2**: Give the recursive formula and explain why it is true.
  - $$T[i] = \max(\ T[i-1] + a_i,\ T[i-2]) + a_{i-1} \cdot a_i\ )$$
  - Since the last sign can either be a $+$ or a $\cdot$.
  - In case of a $+$, we add a_i to whatever is the best assignment to $a_1 + \ldots a_{i-1}$.
  - In case of a $\cdot$, the previous sign must be a $+$, so we add $+a_{i-1} \cdot a_i$ to whatever is the best assignment to $a_1 + \ldots a_{i-2}$.
  - Here we needed the solutions to the previous two values. So this formula only makes sense for $i > 3$, since for example for $i = 2$, $T[i-2] = T[0]$ does not exist. Hence why the next step is:
- **Step 3**: Give the base case for the recursive formula (how do we start filling out the array).
  - $T[1] = a_1$ (no signs for $i = 1$).
  - $T[2] = \max(a_1 + a_2, a_1 \cdot a_2)$.
  - Side note: Instead of $T[2]$, you could also say that $T[0] = 0$, and then the generic formula would have a $T[0]$ value available, which would result in exactly the same value as above.
- **Step 4**: The order in which T must be filled.
  - If you look at the generic formula, to calculate $T[i]$, we need to have already calculated $T[i-1]$ and $T[i-2]$. So we must fill T from $i = 1$ to $i = n$.
  - While this sounds trivial, for more complicated, higher dimensional DP tables, this is also a crucial question. (Along which dimension we step first?)
- **Step 5**: Where is the solution in T?
  - In this exercise, the solution is in $T[n]$, which is $OPT(a_1 + \cdots + a_n)$.
  - In other cases, the solution could be for example the maximum in the entire T table. This was the case for example in Exam: 2022. 05. 30., Exercise 7.
- **Step 6**: What is the runtime of the algorithm?
  - We fill T, a size n array, so so far $n \cdot$ something. In one step, we check back on two previous values and maximize over those, so one step is constant-time. Then, the solution is reading one cell's value, so this is a $O(n)$ algorithm.

17.5 Exam: 2022. 06. 13., Exercise 5

Exercise

We are given the array $A[1:n]$, of length n, which contains **unique** integers. We want to decide, whether or not we can select 100 different numbers from the array, such that the difference of any pairs will be at most 2022. Give a $O(n \log n)$ runtime algorithm to solve this problem!

Solution

- The first thought here is: $O(n \log n)$ runtime allowed $\rightarrow$ we can sort! And that will help!
- Using e.g. merge sort, which runs in $O(n \log n)$ time we sort the array.
- It will help, because after sorting, values that are closer to each other in difference, will be places physically closer to each other.
- This means that after sorting, to find 100 numbers for which all pairs of numbers differ by at most 2022, we can look at a consecutive subarray of length exactly 100 and check if the first and the last element of this subarray meet this requirement.
- If they do, then all pairs inside the subarray also do, their difference can be equal or smaller.
- If there is no solution like this, then there is no possible solution either way. If a solution of size 100 exists, then a solution which is 100 consecutive numbers in the sorted array must also exist. Any gaps in a non-consecutive solution contain numbers that also meet the criteria.
- So in the sorted array we go from $k = 1$ to $k = n - 99$ and check if $|A_{\text{sorted}}[k] - A_{\text{sorted}}[k + 99]| \leq 2022$. If we find such a k, then numbers $\{A_{\text{sorted}}[k], \cdots, A_{\text{sorted}}[k + 99]\}$ are the solution.
- If we find no such k, then no solution exists.

17.6 Exam: 2022. 06. 13., Exercise 6

Exercise

Insert the numbers $0, 1, 15, 10, 66, 43, 55, 53$ in this order into a size $M = 11$ hash table using quadratic probing!

Solution

	0	1	2	3	4	5	6	7	8	9	10	
h(0) = 0	**0**											The hash of 0 is 0, which is empty, so we insert it there.
h(1) = 1	0	**1**										Similarly.
h(15) = 4	0	1			**15**							Similarly.
h(10) = 10	0	1			15						**10**	Similarly.
	1	2			4			5			3	<- Which index is tried at which step.
h(66) = 0	0	1			15			**66**			10	First collision. Apply +1, -1, +4, -4, +9, -9, etc shifts from the original index., modulo 11
												So we try 0+1=1, 0-1 = -1+11 = 10, 0+4 = 4, 0-4 = -4+11 = 7, which is finally empty.
	2									3	1	<- Which index is tried at which step.
h(43) = 10	0	1			15			66		43	10	Collision again, try 10+1=11-11 = 0, 10-1=9, which is empty.
	1	2	7		4			5		6	3	<- Which index is tried at which step.
h(55) = 0	0	1	**55**		15			66		43	10	Collision, try 0+1=1, 0-1 = -1+11 = 10, 0+4 = 4, 0-4 = -4+11 = 7, 0+9=9, 0-9 = -9+11 = 2, which is finally empty.
												Note, how this starts out the same as it was for 66, since their original hash is the same.
									3	1	2	<- Which index is tried at which step.
h(53) = 9	0	1	55		15			66	**53**	43	10	Collision, try 9+1=10, 9-1=8, which is empty.

18 Exam: 2022. 06. 20.

18.1 Exam: 2022. 06. 20., Exercise 1

Exercise

For language A and B over alphabet $\Sigma = \{0, 1\}$ let $A \bigodot B$ denote the language, which consists of those words that are obtained by concatenating a word $w \in A$ with $w' \in B$ that **has the same length** as w, in all possible ways. That is, $A \bigodot B = \{ww' \mid w \in A, w' \in B, |w| = |w'|\}$. Give regular languages A and B, for which $A \bigodot B$ is not regular.

Solution

- This exercise is basically asking if you remember the $a^n b^n$ language, or in this case $0^n 1^n$.
- $A = 0^n$ is regular, because for example I can give the regular expression 0^* for it.
- $B = 1^n$ is also regular, because I can give the regular expression 1^* for it.
- Since only words that are of the same length will be concatenated from A and B this means that only the same number of 0's will be concatenated to the same number of 1's.
- So the $A \bigodot B$ language is in fact $0^n 1^n$, and we have studied this in the lectures that it is not regular (we have seen $a^n b^n$, and we can just substitute $a = 0$ and $b = 1$).
- It is enough to reference that we have seen this proof in the lectures. If you gave a slightly different language and you also explain why the same proof works in that case, then it is also accepted.
- If your $A \bigodot B$ language is very different, then you will need to prove why it is not regular, using the same indirect technique, but probably different $k + 1$ words. See Session 3, Exercise 1 for the proof.
- If you said something along the lines of "we can't recognize this language without a stack", then that is good intuition, but not really a proof. :(Make sure to always reference the $a^n b^n$ language, say that we have proven this in the lectures and then show why yourlanguage is the same (e.g. $0 = a, 1 = b$ in this case).

18.2 Exam: 2022. 06. 20., Exercise 2

Exercise

Give a context-free grammar for the following language: $\{a^{2k+3l}b^kc^l \mid 0 \leq k, l\}$.

Solution

- **KEY IDEA: Any time you need to generate a language where some count of letters should be the same as some other count of letters, you have to generate those letters from the same variable, using the one to the left - one to the right method**.
- In this case, $a^{2k+3l}b^kc^l = a^{3l}a^{2k}b^kc^l$, so there needs to be a variable for "$a^{3k} \dots c^k$" and another variable for "$a^{2k}b^k$".

I have seen the following **wrong** solution so many times:

$$\begin{aligned} S &\to AB \\ A &\to aaaSaa \mid \varepsilon \\ B &\to bBc \mid \varepsilon \end{aligned}$$

- You cannot guarantee that the count of aaa's will be the same as the count of b's (and similarly for aa and c). Since the $A \to aaaSaa$ rule can be executed a different number of times as the $B \to bBc$ rule.
- For example I can generate the following word: $S \to AB \to aaaSaaB \to aaaaaaSaaaaB \to aaaaaaSaaaabBc \to aaaaaaaaaabBc \to aaaaaaaaaabc = a^{6+4}b^1c^1$. The counts are disjoint from each other, because we used different variables!

Instead, you will need the following two variables (and we will talk about the starting variable and how to put them together in a moment):

$$\begin{aligned} A &\to aaaAc \mid \varepsilon \\ B &\to aaBc \mid \varepsilon \end{aligned}$$

- The amount of times the $A \to aaaAc$ rule is executed will correspond to the number l of the input word: if we want $l = 2$ for example, aka. a^6c^2, then the rule is executed 2 times.
- Similarly the amount of times the $B \to aaAb$ rule is executed will correspond to the number k of the input word: if we want $k = 4$ for example, aka. a^8b^4, then the rule is executed 4 times.
- Now, the only problem is that the "$a^{2k}b^k$" part needs to come between the a's and the c's of the "$a^{3k} \dots c^k$" part of the word. So we cannot make the starting variable $S \to AB$, because that would result in $a^{3k}c^ka^{2l}b^l$, which is not the correct order.
- Instead, this can be achieved by changing the $A \to \varepsilon$ rule into an $A \to B$ rule, like this:

$$\begin{aligned} A &\to aaaAc \mid B \\ B &\to aaBc \mid \varepsilon \end{aligned}$$

- Now the starting variable is A, and when the $A \to aaaAc$ rule generated he external a's and c's, our exit rule for the A variable is to change into a B and start generating the inner a's and b's.
- So to achieve the word $a^{3l}a^{2k}b^kc^l$, we will execute the rule $A \to aaaAc$ exactly l times, then we execute the rule $A \to B$, then we execute the rule $B \to aaBc$ exactly k times, then we execute the rule $B \to \varepsilon$ and we are done.

18.3 Exam: 2022. 06. 20., Exercise 3

Exercise

The input of decision problem $NFA - EMPTY$ is a description of a non-deterministic finite automaton M, and the question is to decide whether the language of M is empty. Prove that $NFA - EMPTY \in coNP$.

Solution

- Let's start by the definition of $coNP$: the languages of $coNP$ are the complementers of languages in NP.
- So to prove that $NFA - EMPTY \in coNP$, we must prove that $\overline{NFA - EMPTY} \in NP$.
- To prove that a language is in NP, we use the Witness Theorem.
- In this case, the language $\overline{NFA - EMPTY}$:
  - Answers YES, if it is given an NFA description on its input that corresponds to an NFA, which has at least one accepted word in its language (the complementer to empty language = no accepted words, is having at least one accepted word).
  - Answers NO if it is given something that is not an NFA description, or given an NFA description which doesn't have an accepted word (it's language is empty).
- We need a witness for the YES answer, so given an NFA description we need to 'point out' an accepted word in it.

First a longer approach, that I would like to explain anyways, to help your understanding of the topic and appreciate the quicker answer:

- It is important that the witness size is polynomial, so any accepted word won't do. We can think of an NFA description in the following way: a list of its n states, a list of its $k = |\Sigma|$ sized alphabet, and a description of it's transitions: from any state, to any state, by any letter in the alphabet: $O(n^2k)$.
- So the input size is $O(n + k + n^2k) = O(n^2k)$.
- We need an input word that is in this size limit. Luckily any input word that has more than n characters in it will correspond to a path in the NFA that contains at least one loop (since it will have to return to one of the states to fit into the NFA), so we can cut this loop out and keep the non-looped path, corresponding to an input word that has less than n characters in it.
- The witness checking algorithm would be writing out the branching computational tree of the input word and checking if it has an accepted state at the end.
- The witness checking algorithm has to run in polynomial time. If you just naively execute the algorithm we have studied in the class, that will be $O(n^n)$, since in any state, it can branch out to n other branches (for any character we can move to any other state, since it is a nondeterministic automaton), and there will be n steps done.
- However, this branching would contain many duplicate states, since after reading in some characters from the input word, it only matters which of the n states we can end up in, not the many possible paths we could have taken.
- So if we modify the algorithm and just write down after each step, which of the states we could be in, removing duplicates in every case, then the runtime will be just $O(n^3)$: the input word is at most n characters and for each character we can have a list of at most n current states, so we can move to at most n^2 states in each step, so n^3 steps in total, but in each step we remove the duplicates from n^2, reducing the current number of states to just n, for the next step.
- So if you write down 'a witness is an accepted word', you would also have to explain all of the above, which is quite a lot, would not recommend.

And then a quicker, simpler approach:

- Just point out a path from the start state to any accepting state. That will correspond to an accepting computation (for the input word corresponding to the edges of the path). The important distinction here is that in the witness itself, we also give the information of which computational branch is the accepting one.
- The length of the witness is $O(n)$, since it's a path in an n-state automaton.
- And the witness checking is just seeing if that path indeed exists in the finite automaton. We don't have to worry about non-determinism in this case.
- Actually, finding a path from a given state (vertex) to some other state (vertex) in an NFA (directed graph) can be done using BFS, so actually this problem is in P. :)

18.4 Exam: 2022. 06. 20., Exercise 4

Exercise

The array $A[1:n]$ contains the daily prices of a given share in the way that $A[i]$ is the price on day i (there is only one price on a day). If we buy the share on the ith day and sell on the jth day ($j > i$), then we realize a profit of $A[j] - A[i]$. Give a dynamic programming algorithm that runs in $O(n)$ time and determines the maximum realizable profit by buying the share on a single day and selling the share on a single day. (For example if $A[1:n] = [5, 11, 2, 6, 4, 10, 1, 8]$, then the maximal profit is 8, realized by buying on the third day and selling on the sixth day.)

Solution

This is not technically a dynamic programming solution, but one that runs in $O(n)$ time and is the simplest (and I have seen this from most papers):

- If we sold on the ith day, then we must have bought the shares on one of the days between 1 and $(i-1)$.
- The best day to buy is the one with the lowest price.
- So we will make a loop with pointer i, from $i = 2$ to $i = n$, which will represent the current selling date.
- In each step we keep track of the minimum price we have seen before: on $i = 2$, $min = A[1]$, and for every other i, min is updated by $min = min(min, A[i-1])$. So min will always contain the minimum of $A[1:(i-1)]$.
- Then we check the current best profit $A[i] - min$, and keep track of the maximum we have seen in another max variable, which we print out at the end.

18.5 Exam: 2022. 06. 20., Exercise 5

Exercise

Array A contains positive numbers, not necessarily in sorted order. Give an algorithm that chooses k elements of A, so that the sum of the chosen elements is not larger than k^3. If there is no such k, then the algorithm should note that. The running time must be $O(n \log n)$.

Solution

- First thing to notice: $O(n \log n)$ means we can sort! It will be useful to us in this case, because:
- We can take any k elements, not necessarily consecutive ones, so their current placement doesn't matter.
- And if we took k elements, it is always better to choose the smallest k elements, since we have an upper limit on their sum.
- So let's sort the array and iterate over it with an i iterator: in each step we keep track of the current sum. For $i = 1$, $sum = A[1]$, and when we move i, we can update the current sum by $sum = sum + A[i]$.
- Then, in each step, we check if $sum \leq i^3$. If for a specific i this is true, then $k = i$ is found and we print the $A[1 : i]$ numbers out.
- If for a specific step the sum is too large, we can continue with i, since as i^3 grows it can be able to upper-estimate the sum again!
- If we have went through the entire array and have not found a solution, we can print out 'NOT POSSIBLE'. This is because if it is not possible for the smallest k elements to sum to $\leq k^3$, then by choosing other, larger elements, we just make it worse. So it is definitely not possible in other ways either.

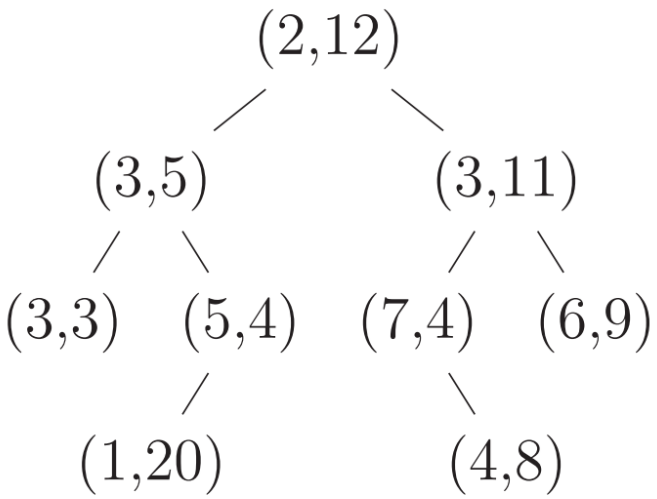
18.6 Exam: 2022. 06. 20., Exercise 6

Exercise

Define an ordering of pairs of positive integers as follows. $(a, b) < (c, d)$ holds if and only if

- $a \cdot b < c \cdot d$ or
- $a \cdot b < c \cdot d$ and $\min(a, b) < \min(c, d)$

(For example, $(3, 5) < (2, 12)$, because $15 < 24$ and $(1, 20) < (5, 4)$, because both products are 20, but $1 < 4$.)
The figure below is a binary search tree with using ordering.

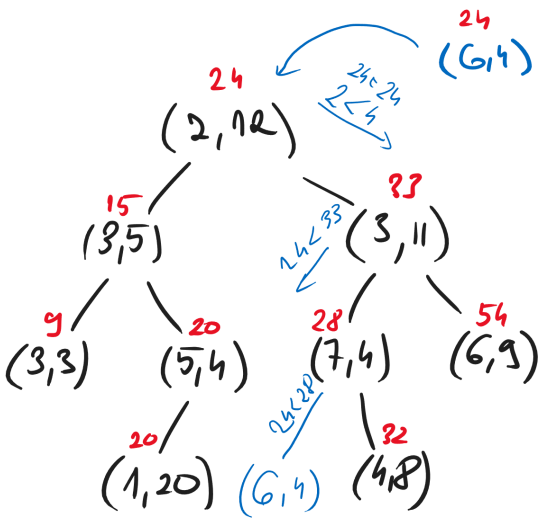


a) Insert the element $(6, 4)$ into the tree.

b) Delete the pair $(2, 12)$ from the tree obtained in part a).

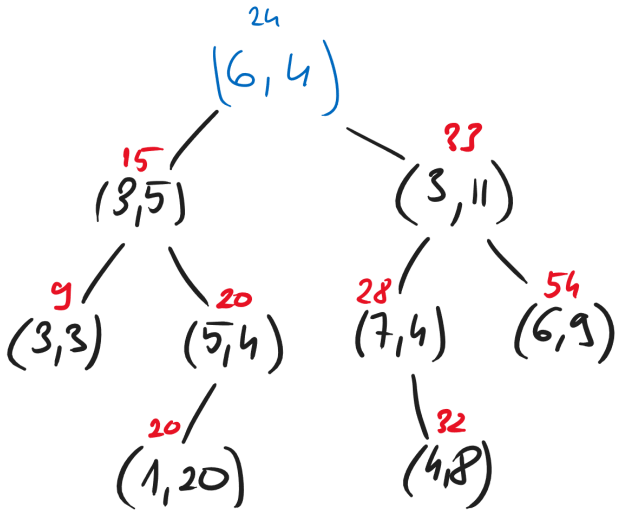
Solution

Insertion:



Deletion:

- The minimum element of the right subtree gets put in place of $(2, 12)$, which is the $(6, 4)$ we just inserted.
- You can find the minimum element of the subtree by always navigating to the left, until you reach a leaf.



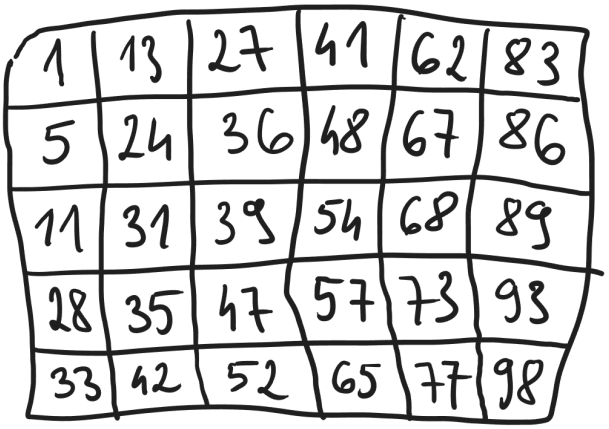
18.7 Exam: 2022. 06. 20., Exercise 7

Exercise

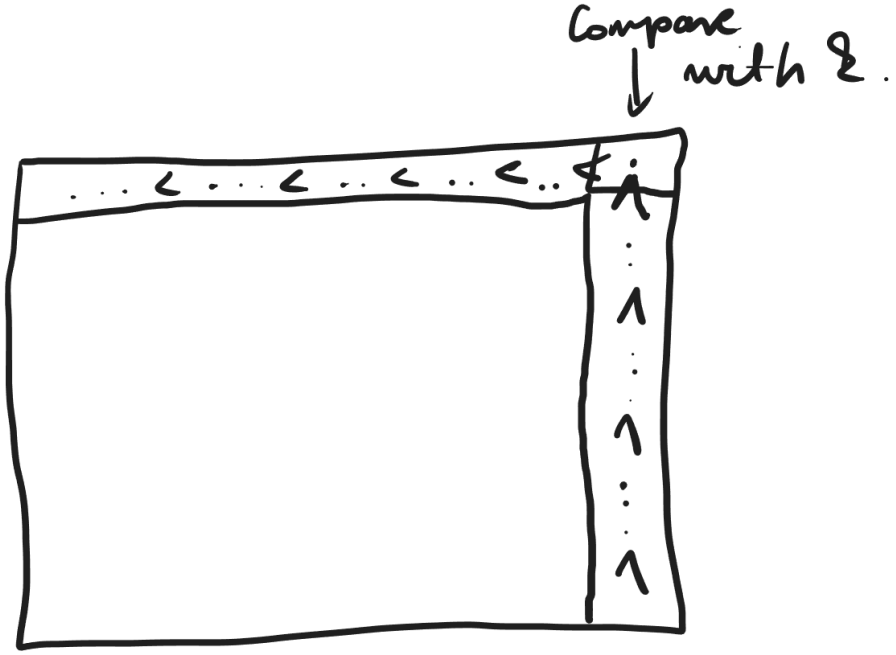
Pairwise distinct integer numbers are stored in an $(n \times n)$ table in such a way, that in each row the numbers are increasing from left to right and in each column they are increasing from top to bottom. Give an algorithm using $O(n)$ comparisons that decides whether a given integer k is contained in the table. (The comparison of x and y may have three different results: $x = y, x < yorx > y$.)

Solution

- First thing to note: the array is not necessarily fully sorted! It is only partially (inside each row and inside each column). For example, this is a valid input, you can see that it is not fully sorted:



- First, we will check the upper right corner:



- There are 3 possibilities:
  - This can be $= k$, in this case we found k and return.
  - This can be $< k$, in this case we know that not only is k not in this cell, but also not in the entire first row either, since everything in the first row is to the left of this cell, so they are even smaller than it.

  - this can be $> k$, in this case we know that not only is k not in this cell, but also not in the entire last column either, since everything in the last column is below this cell, so they are even bigger than it.

- This means, that we can either throw away the first row, or the last column from our table.
- Then we do the same thing recursively, for the remaining table, until we find k, or until we reach the bottom left corner. That can either be k, or not, and if it's not k, then k is not in the table at all.
- Since the table has in the beginning n rows and n columns, and in each step we get rid of 1 row or 1 column, in total we need to get rid of $2n$ things to finish.
- To implement getting rid of a row, or column, the best thing is to just keep two variables at hand, one for storing 'the first row of the remaining table' and one for storing the 'last column of the remaining table'. We don't have to actually remove these from our table, since that would be too many operations.
- This means that the algorithm will run in $O(2n) = O(n)$ time.