Practice Session Notes: Theory of Algorithms

Viktória Nemkin (nemkin@cs.bme.hu)

June 9, 2022

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 O, Ω, Θ, pattern matching **5**

1.1 Session 1, Exercise 01 . . . 5
1.2 Session 1, Exercise 02 . . . 6
1.3 Session 1, Exercise 03 . . . 8
1.4 Session 1, Exercise 04 . . . 10
1.5 Session 1, Exercise 05 . . . 12
1.6 Session 1, Exercise 06 . . . 13
1.7 Session 1, Exercise 07 . . . 14
1.8 Session 1, Exercise 08 . . . 17
1.9 Session 1, Exercise 09 . . . 18
1.10 Session 1, Exercise 10 . . . 19
1.11 Session 1, Exercise 11 . . . 20
1.12 Session 1, Exercise 12 . . . 21

2 Finite Automata **22**

2.1 Session 2, Exercise 01 . . . 22
2.2 Session 2, Exercise 02 . . . 23
2.3 Session 2, Exercise 03 . . . 25
2.4 Session 2, Exercise 04 . . . 26
2.5 Session 2, Exercise 05 . . . 28
2.6 Session 2, Exercise 06 . . . 29
2.7 Session 2, Exercise 7 . . . 35
2.8 Session 2, Exercise 08 . . . 38
2.9 Session 2, Exercise 09 . . . 39
2.10 Session 2, Exercise 10 . . . 40
2.11 Session 2, Exercise 11 . . . 44

2.12 Session 2, Exercise 12 . . . 45
2.13 Session 2, Exercise 13 . . . 46

3 Regular expressions, context-free languages 47

3.1 Session 3, Exercise 1 . . . 47
3.2 Session 3, Exercise 2 . . . 48
3.3 Session 3, Exercise 3 . . . 49
3.3.1 Even number of 0's . . . 49
3.3.2 Odd number of 0's . . . 49
3.3.3 A perfect square number of 0's . . . 50
3.3.4 A power of 2 . . . 50
3.4 Session 3, Exercise 4 . . . 51
3.5 Session 3, Exercise 5 . . . 52
3.5.1 Words of odd lengths . . . 52
3.5.2 Words of even length that start and end with 1 . . . 52
3.5.3 Words of odd length that start and end with 1 . . . 52
3.5.4 Words containing at least three 0's . . . 52
3.5.5 Words containing an even number of 0's . . . 53
3.5.6 Words of odd lengths starting with 0 and words of even length starting with 1 . . . 53
3.5.7 Words of odd length containing subword 00 . . . 53
3.6 Session 3, Exercise 6 . . . 54
3.7 Session 3, Exercise 7 . . . 55
3.8 Session 3, Exercise 8 . . . 56
3.9 Session 3, Exercise 9 . . . 57
3.9.1 Contains 011 as a substring . . . 57
3.9.2 Starts with 1, ends with 0 . . . 57
3.9.3 Is of even length . . . 57
3.10 Session 3, Exercise 10 . . . 58
3.11 Session 3, Exercise 11 . . . 59
3.12 Session 3, Exercise 12 . . . 60

4 Context-free grammars, pushdown automata 61

4.1 Sessions 4 and 5, Exercise 1 . . . 61
4.2 Sessions 4 and 5, Exercise 2 . . . 62
4.3 Sessions 4 and 5, Exercise 3 . . . 64
4.4 Sessions 4 and 5, Exercise 4 . . . 66
4.5 Sessions 4 and 5, Exercise 5 . . . 67
4.6 Sessions 4 and 5, Exercise 6 . . . 69
4.7 Sessions 4 and 5, Exercise 7 . . . 70
4.8 Sessions 4 and 5, Exercise 8 . . . 71
4.9 Sessions 4 and 5, Exercise 9 . . . 72
4.10 Sessions 4 and 5, Exercise 10 . . . 73
4.11 Sessions 4 and 5, Exercise 11 . . . 74
4.12 Sessions 4 and 5, Exercise 12 . . . 75

5 Turing-machines 76

5.1 Session 5, Exercise 1 . . . 76
5.2 Session 5, Exercise 2 . . . 77
5.3 Session 5, Exercise 3 . . . 78
5.4 Session 5, Exercise 4 . . . 80
5.5 Session 5, Exercise 5 . . . 81
5.6 Session 5, Exercise 6 . . . 83
5.7 Session 5, Exercise 7 . . . 84
5.8 Session 5, Exercise 8 . . . 86
5.9 Session 5, Exercise 9 . . . 87

6 Turing-machines 2 88

6.1 Session 6, Exercise 1 . . . 88
6.2 Session 6, Exercise 2 . . . 89
6.3 Session 6, Exercise 3 . . . 90
6.4 Session 6, Exercise 4 . . . 91
6.5 Session 6, Exercise 5 . . . 92
6.6 Session 6, Exercise 6 . . . 93
6.7 Session 6, Exercise 7 . . . 94
6.8 Session 6, Exercise 8 . . . 95

7 P, NP 96

7.1 Session 7, Exercise 1 . . . 96
7.2 Session 7, Exercise 2 . . . 98
7.3 Session 7, Exercise 3 . . . 99
7.4 Session 7, Exercise 4 . . . 100
7.5 Session 7, Exercise 5 . . . 102
7.6 Session 7, Exercise 6 . . . 103
7.7 Session 7, Exercise 7 . . . 104
7.8 Session 7, Exercise 8 . . . 105
7.9 Session 7, Exercise 9 . . . 106

8 NP-completeness 107

8.1 What is a Karp-reduction? . . . 107
8.2 Session 8, Exercise 1 . . . 110
8.3 Session 8, Exercise 2 . . . 113
8.4 Session 8, Exercise 3 . . . 114
8.5 Session 8, Exercise 4 . . . 115
8.6 Session 8, Exercise 5 . . . 118
8.7 Session 8, Exercise 6 . . . 119
8.8 Session 8, Exercise 7 . . . 121
8.9 Session 8, Exercise 8 . . . 123
8.10 Session 8, Exercise 9 . . . 124
8.11 Session 8, Exercise 10 . . . 125
8.12 Session 8, Exercise 11 . . . 126
8.13 Session 8, Exercise 12 . . . 127

9 **128**

10 **128**

11 **128**

12 **128**

13 Binary search trees, 2-3-trees, hash **128**

13.1 Session 13, Exercise 9 128

14 Exam: 2022. 05. 30. **131**

14.1 Exam: 2022. 05. 30., Exercise 1 131

14.2 Exam: 2022. 05. 30., Exercise 2 132

14.3 Exam: 2022. 05. 30., Exercise 3 133

14.4 Exam: 2022. 05. 30., Exercise 4 134

14.5 Exam: 2022. 05. 30., Exercise 5 135

14.6 Exam: 2022. 05. 30., Exercise 6 136

14.7 Exam: 2022. 05. 30., Exercise 7 137

15 Exam: Miscellaneous **138**

15.1 Exams, Exercise 1 138

15.2 Exams, Exercise 2 140

1 O, Ω, Θ, pattern matching

1.1 Session 1, Exercise 01

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 02

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

$8n^2 \log_2(n) \le cn^2$ for $\forall n \ge n_0$.

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

$8 \log_2(n) \le c$ for $\forall n \ge n_0$.

Let's move the 8 to the right:

$\log_2(n) \le \frac{c}{8}$ for $\forall n \ge n_0$.

And this is a contradiction, since $\frac{c}{8}$ is a constant, while $\log_2(n)$ can be arbitrarily large for a given $n \ge n_0$, 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 $* n^2$.

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

We have arrived at $f(n) \ge 8n^2$ when $n \ge 2$, so we can choose $c = 8$ and $n_0 = 2$, which gives $f(n) \ge cn^2$ when $n \ge n_0$, 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.)

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

We have arrived at $f(n) \le 4.5n$, 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 $f(n) \le cn$ for $\forall n \ge n_0$, 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

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

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 $4.5n \ge 1.5n + 3\sqrt{n}$:

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

Now let's just look at the two sides:

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

Let's divide both sides by n:

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

And move the constant to the left:

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

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 03

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 $\frac{1}{n^{|x|}} \rightarrow \frac{1}{\infty} = 0$. 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 $\frac{1}{2^{|x|n}} \rightarrow \frac{1}{\infty} = 0$. 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 04

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 05

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 $n \geq 4$.

$(2^n+2^n)(3^n+3^n) = 2*2^n3*3^n = 6*6^n$, so $c = 6$ and $n_0 = 4$ works for $O(6^n)$.

1.6 Session 1, Exercise 06

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 07

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)

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

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

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

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 $\log_2(n) * n^k + 2 \in O(\log_2(n) * n^k)$.

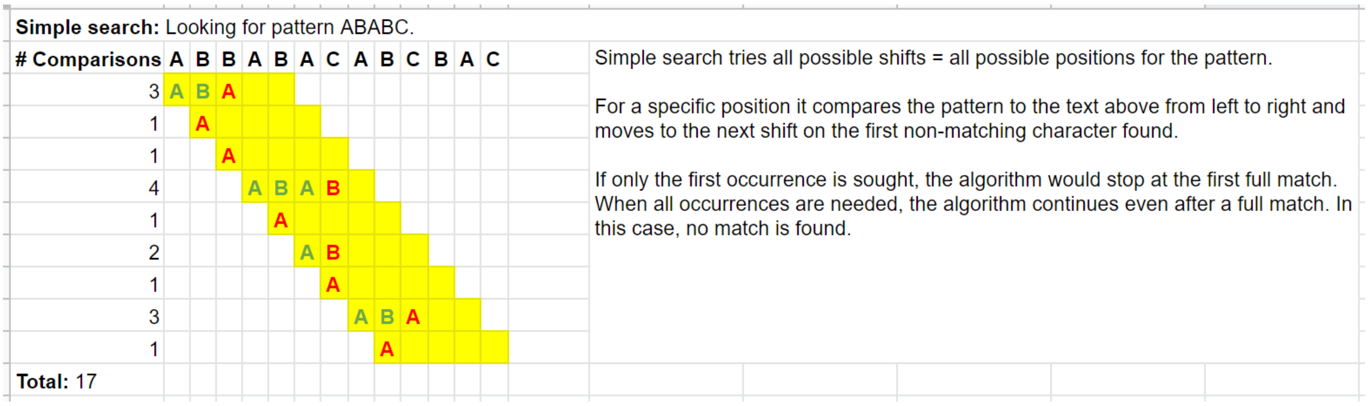
Yes, we did not arrive at the exact solution of $O(n^k)$, but $O(\log_2(n) * n^k)$. Our upper estimates in this case happened to break the boundary of the $O(n^k)$ set, since we upper estimated $\log_2(n)$ number of times with n^k. Sometimes it is better to arrive at a bit worse estimate quicker, than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

1.8 Session 1, Exercise 08

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



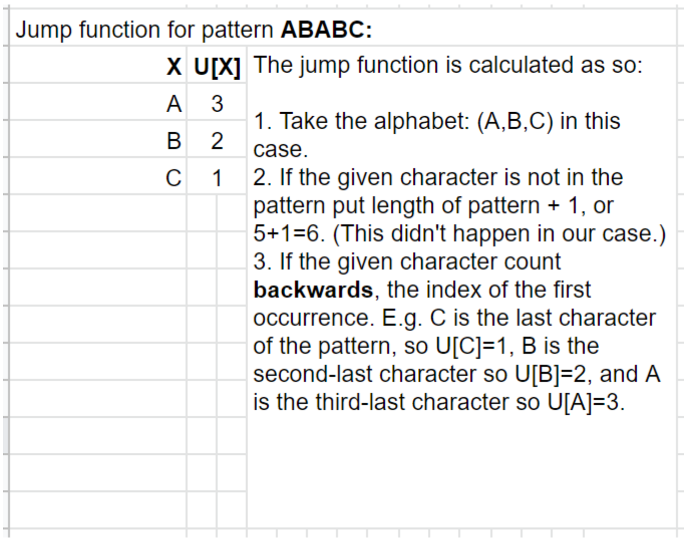
Simple search: Looking for pattern ABABC.

# 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.

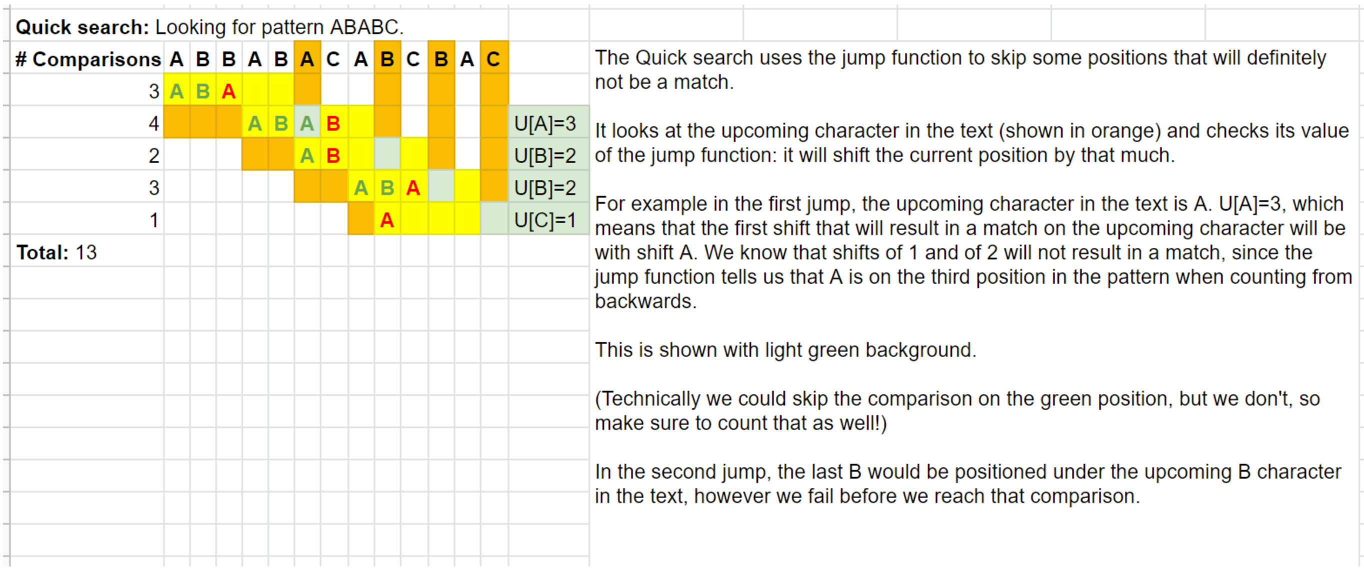


Jump function for pattern **ABABC:**

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 09

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 $\sum\limits_{i=0}^{n-m} t_i$, so the expected number of comparisons is $E(\sum\limits_{i=0}^{n-m} t_i)$.

$E(\sum\limits_{i=0}^{n-m} t_i) = \sum\limits_{i=0}^{n-m} E(t_i)$, 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.

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

Then we plug this back in:

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

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 $O((n+T(n,m))log_2|\Sigma|)$.

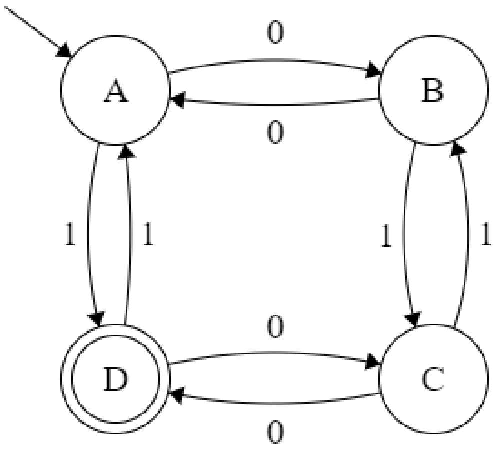
2 Finite Automata

2.1 Session 2, Exercise 01

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:

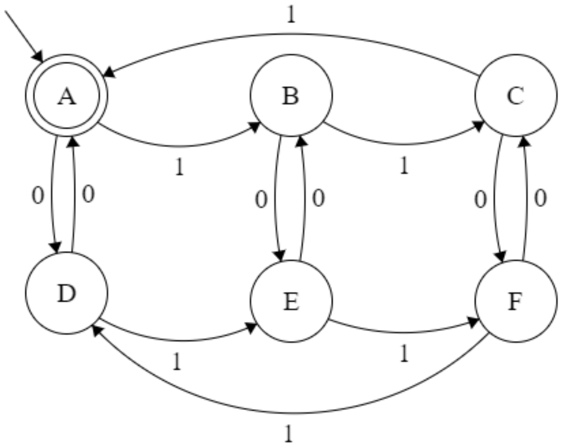
- Transitions triggered by a 0 input are $A \to B$, $B \to A$, $C \to D$, $D \to C$. In these cases the parity of the 1's doesn't change, while the parity of the 0's is inverted. If we look back on what the states represent we can verify in all 4 cases that this is the case.
- Transitions triggered by 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 02

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).

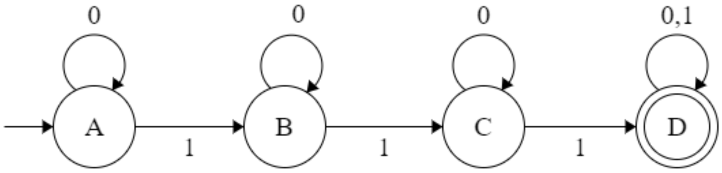
- $B \to C$ and $E \to F$ move from states that have seen $3k + 1$ 1's to states that have sen $3k + 2$ 1's (the remainder goes from 1 to 2).
- $C \to A$ and $F \to D$ move from states that have seen $3k+2$ 1's to states that have sen $3k$ 1's (the remainder goes from 2 to 0).

2.3 Session 2, Exercise 03

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 04

Exercise

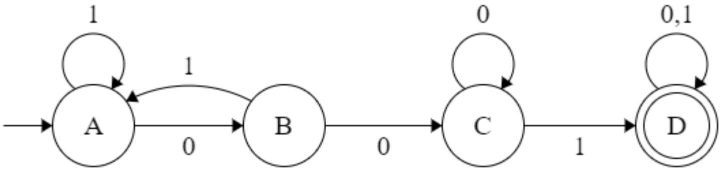
Let $\Sigma = \{0, 1\}$. Give a deterministic finite automaton that accepts the words that do not contain the subword 001.

Solution

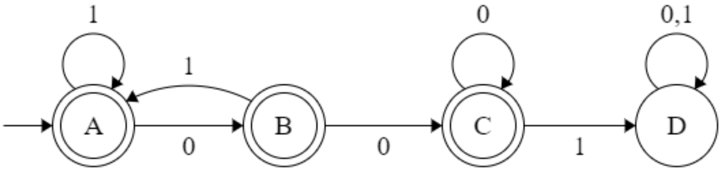
When dealing with **deterministic** finite automatons, a useful trick to keep in mind: sometimes it is easier to give a DFA for the complementer of the language, and then a DFA for the original language can be quickly created.

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.

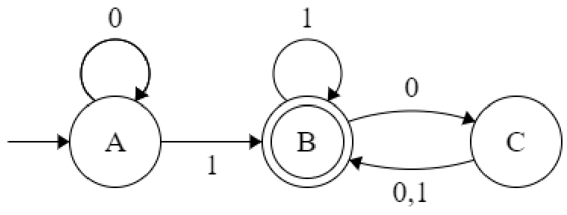
- In state C the current ending is 00. If we read in a 1, that means we just found a 001 substring, we can move to state D! And if we read a 0 in, while we can't move to state D, we can stay in C, since the current ending is 000, or discarding the oldest 0: 00, which means we can stay in state C.

- 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 05

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 06

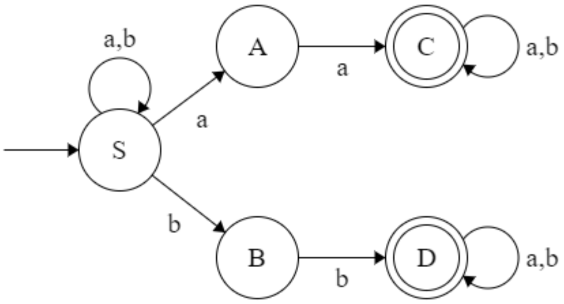
Exercise

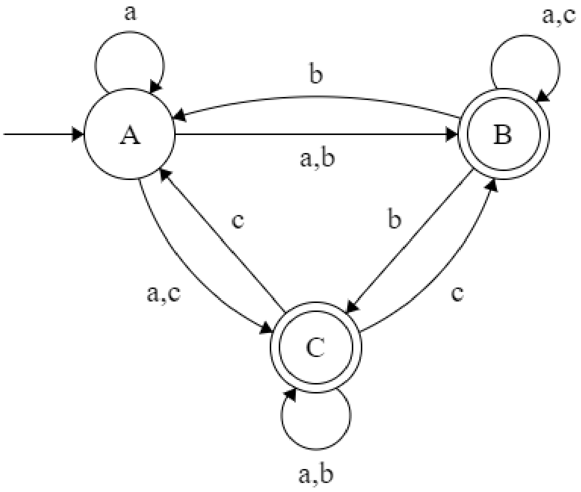
Perform the following for both nondeterministic finite automata.

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

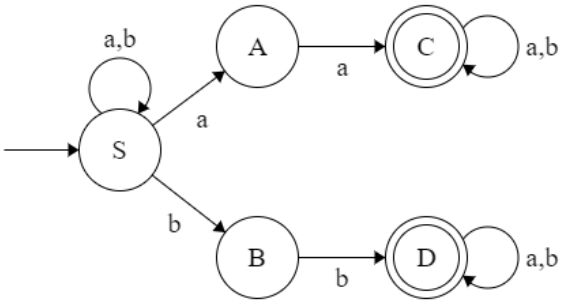
b.) Create an equivalent DFA by the procedure studied in class.

c.) Which languages are recognized by these automata?

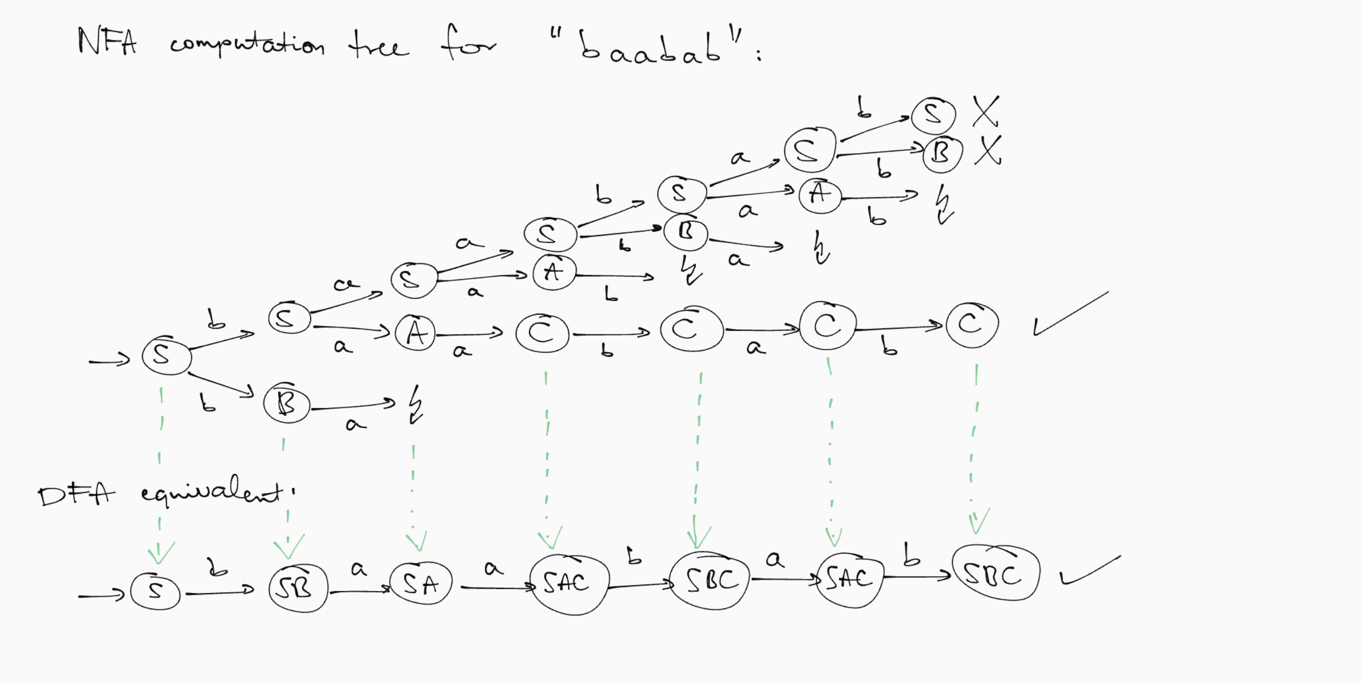




Solution



Give the computation tree of word *baabab*:

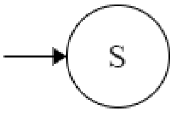


The word *baabab* is accepted, since there exists a branch that resulted in an accept state (C).

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:

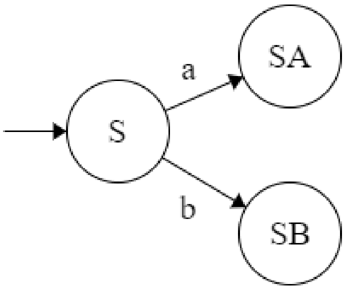
Step 1

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.



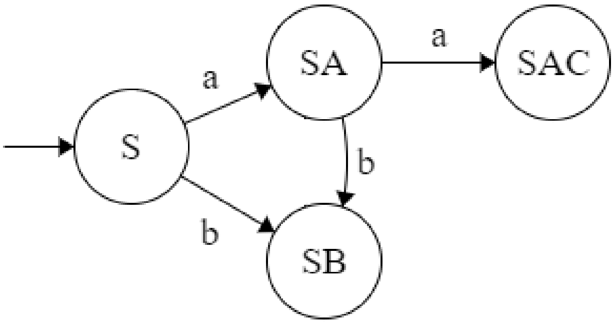
Check back on the DFA equivalent on the drawing above! You will notice the state SB as the second step.

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.

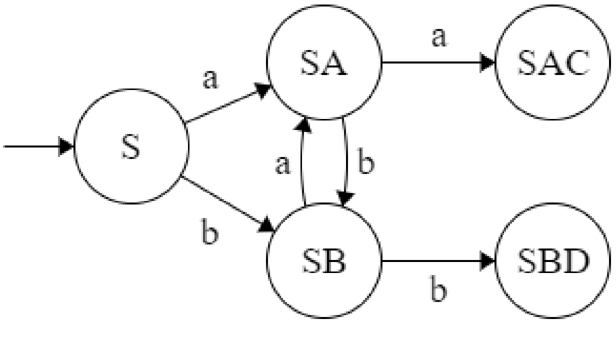
Where does SA move for an input character of b? The transitions from S and from A for input b are: $S \xrightarrow{b} \{S, B\}$ and $A \xrightarrow{b} \{\}$, so together they move to state SB.



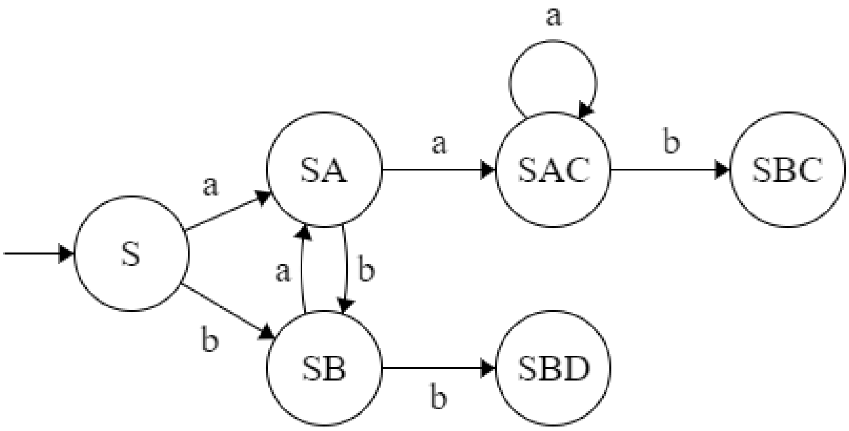
The process is the same: find a state that does not have all transitions defined (for a and for b input as well), check where their basis states transition in the NFA for the given input and add it as a transition. If the resulting state does not exist, add the state.

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

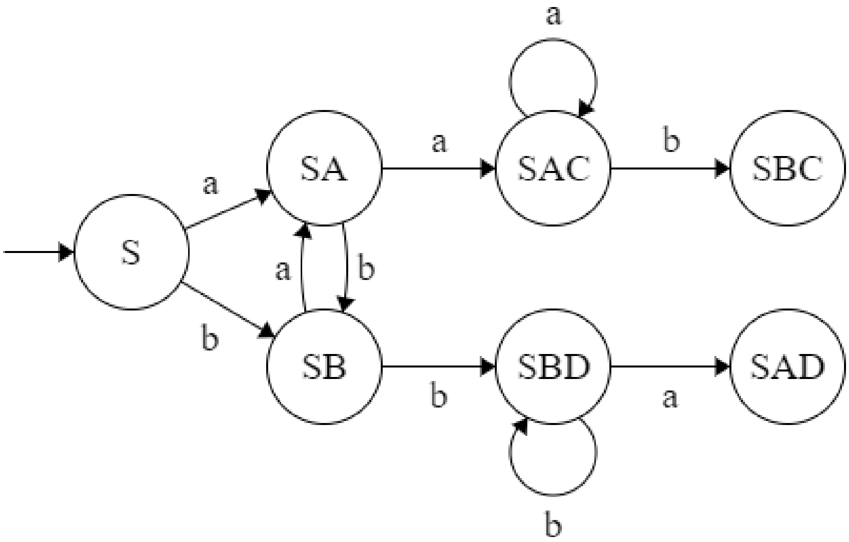
Step 4



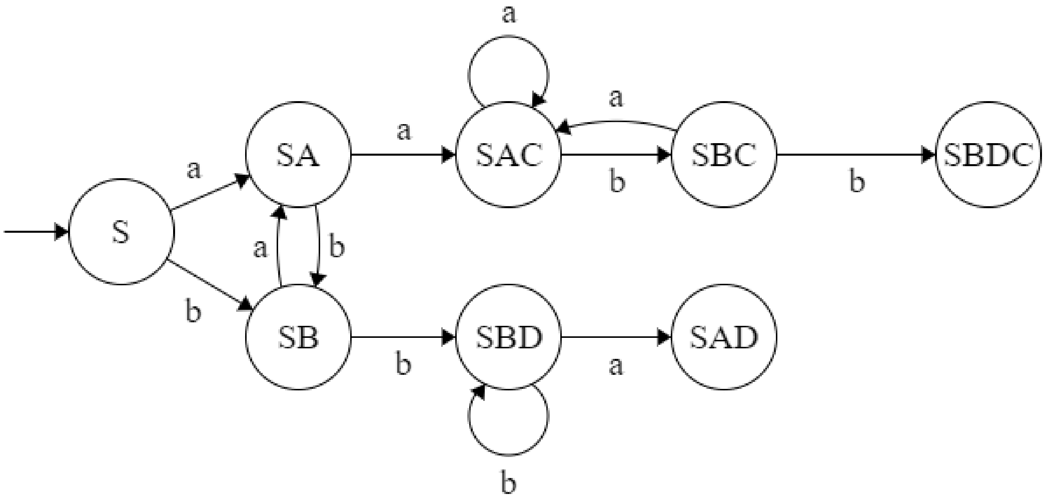
Step 5



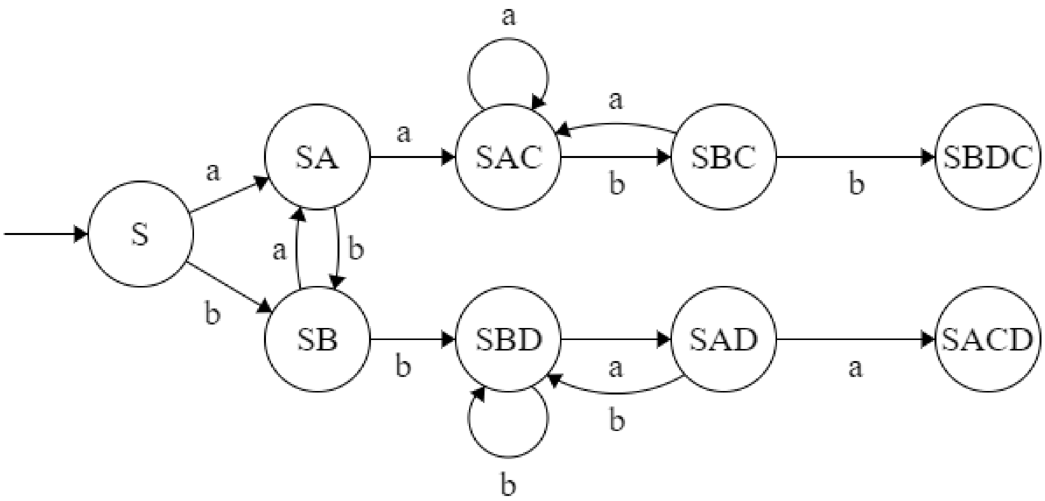
Step 6



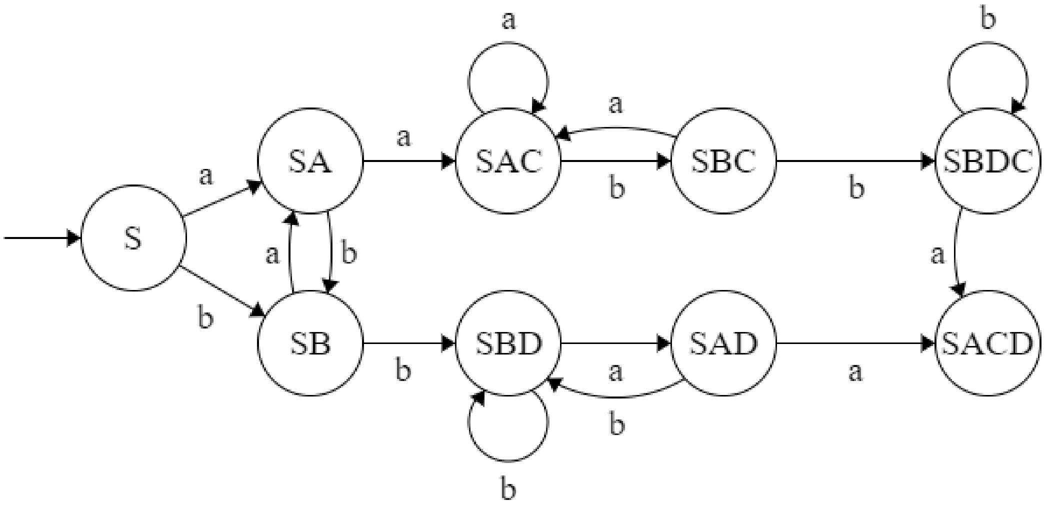
Step 7



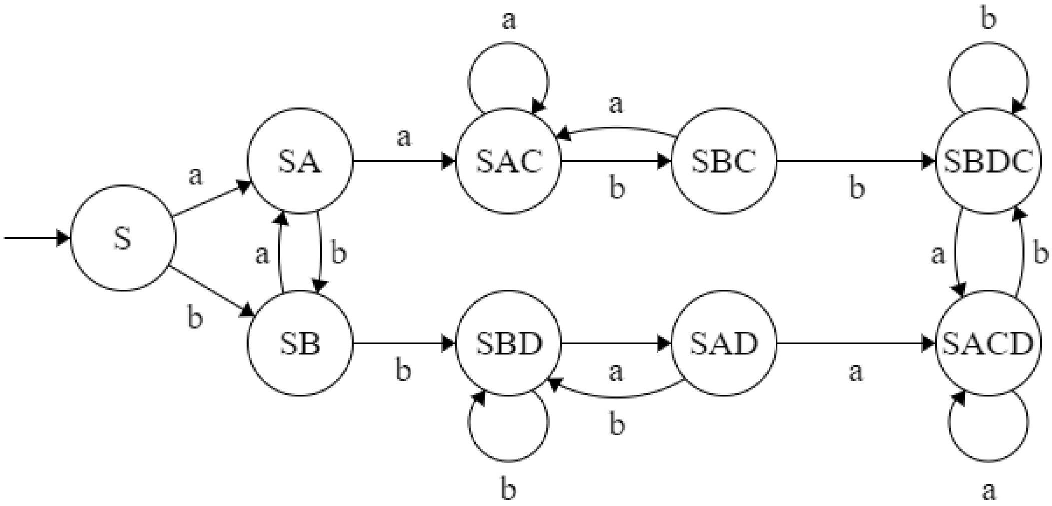
Step 8



Step 9

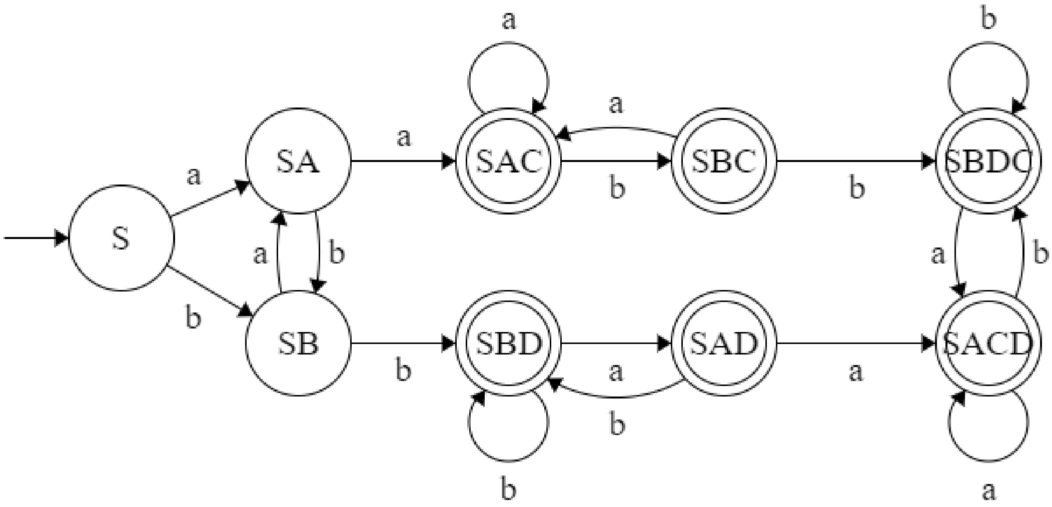


Step 10



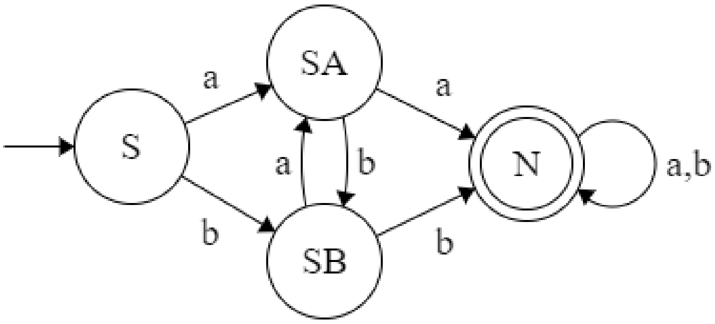
Step 11

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

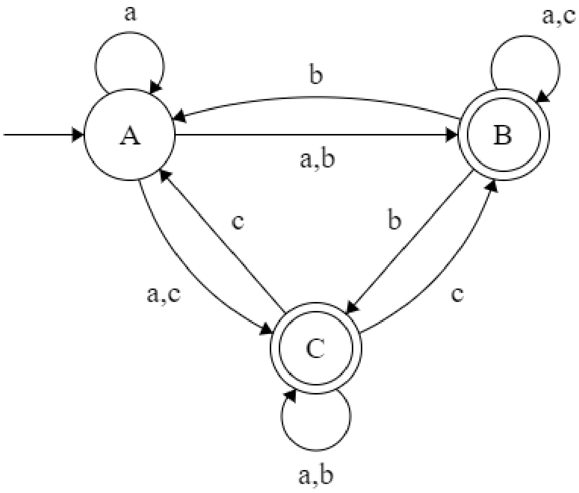


Note

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

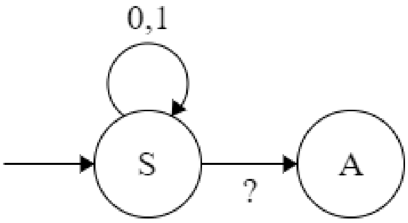
2.7 Session 2, Exercise 7

Exercise

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

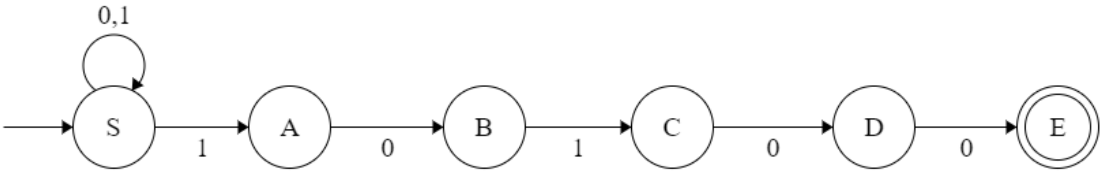
Solution

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:

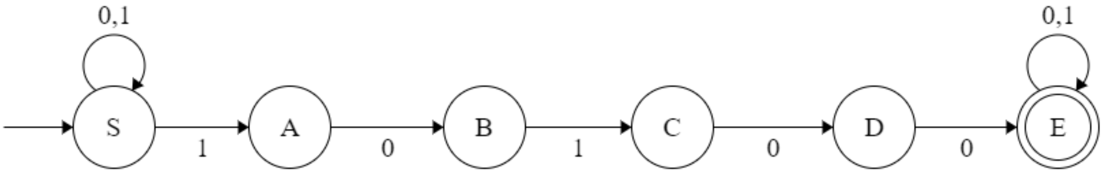


This will "eat up" some prefix of the input word before allowing the computation to proceed to state A. Whatever we put in place of the ? on the leaving transition will be a nondeterministic choice for the automaton.

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



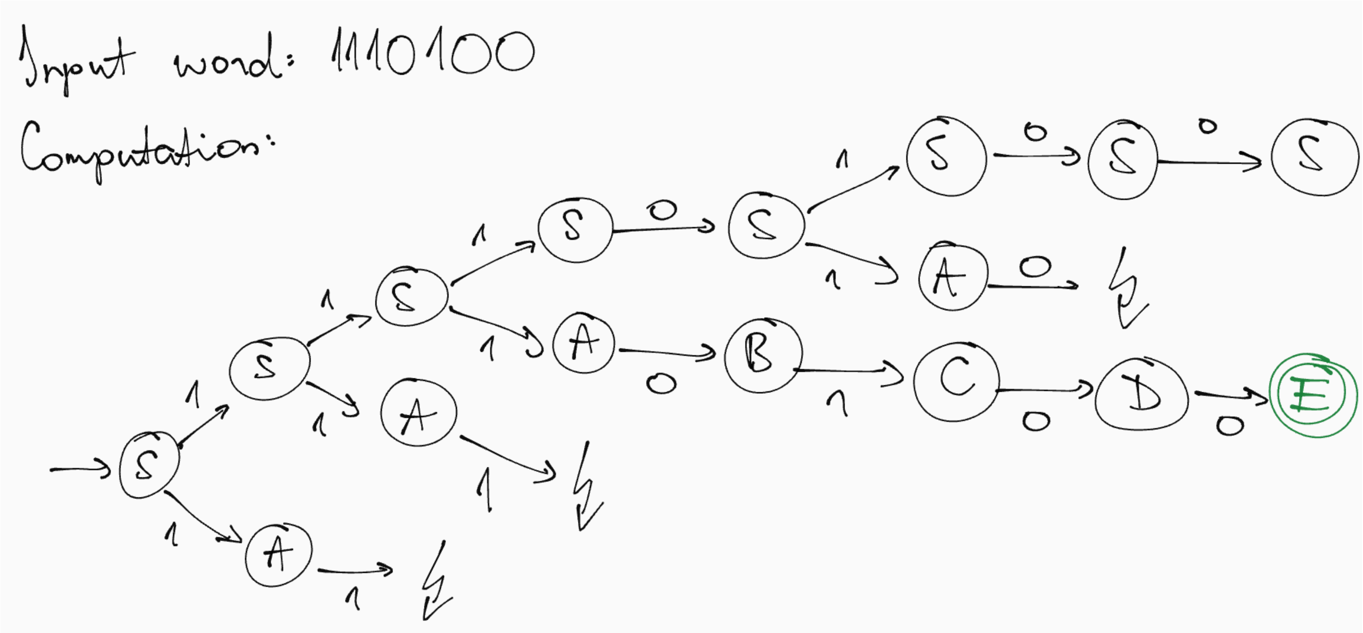
Then finally, let's allow "eating up" any remaining suffix of the word as well:



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.)

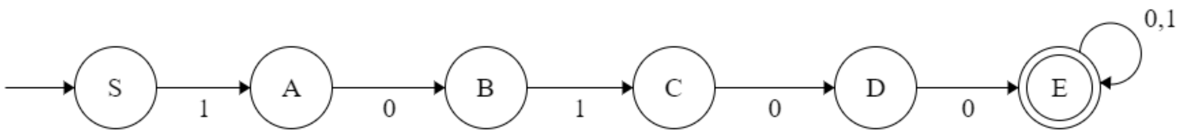
It is important when doing these proofs, that you do not use a specific input word as an example, but generalize to any possible words, like the proof above.

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.

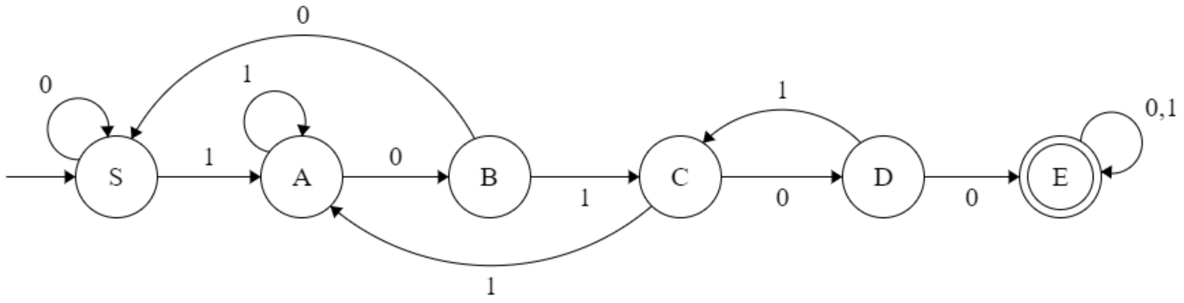
However, the task is also possible with a deterministic automaton (it must be, since we can convert any NFA into a DFA, but there is an even easier method to obtain a DFA, than the general conversion algorithm). This is not part of the task, but nice to see how it works as well.

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 08

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 09

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.

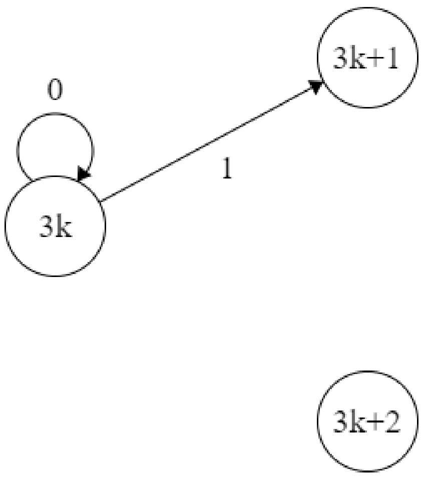
- For example, let's say that so far we have read the the "101" binary string on the input. That is a 5 in decimal form.
- Let's say the next character is also a 1, so now the current string is "1011", or in decimal form 11.
- This was achieved by shifting the string "101" to the left and adding a "1" 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.

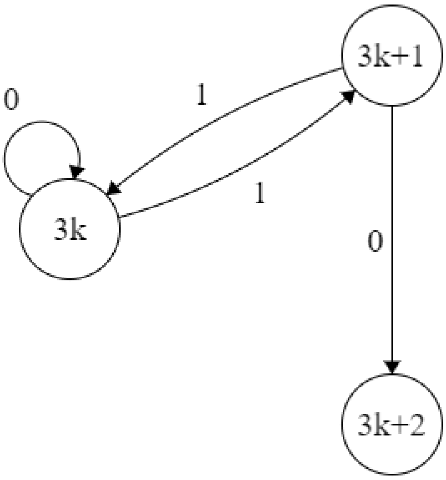
We can represent these statements by the following two transitions from state $3k$:



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.

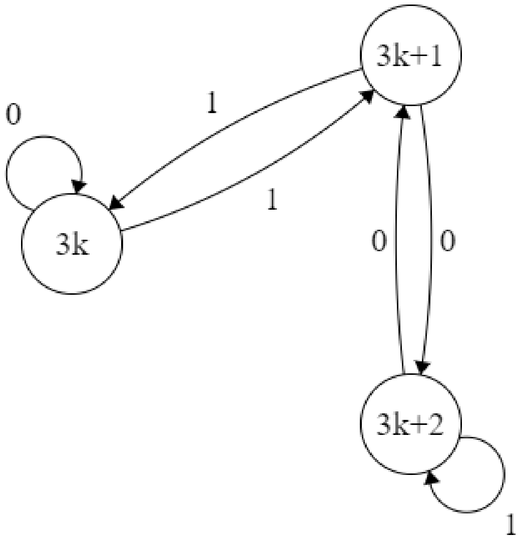
If the next binary character is a 1, then to update we do $(3k + 1) * 2 + 1 = 6k + 3 = 3 * (2k + 1)$, which means that the updated number is divisible by 3.

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



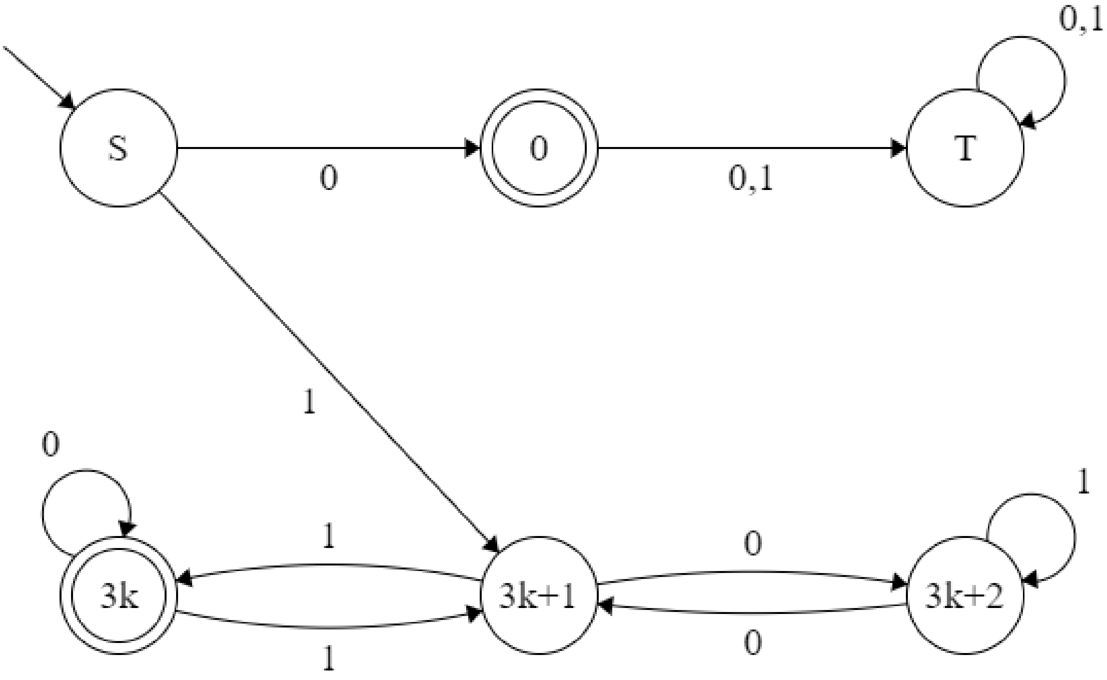
Finally, if the current number has a remainder of 2, when divided by 3, or in the for of $3k + 2$, and the next binary character is a 0, then to update we do $(3k + 2) * 2 + 0 = 6k + 4 = 3 * (2k + 1) + 1$, which means that the updated number has a remainder of 1, when divided by 3.

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 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

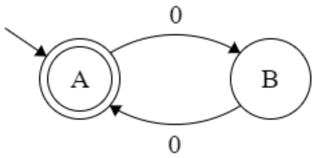
Regular.

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

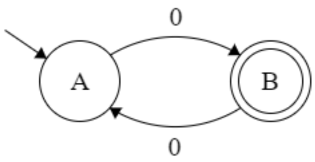
Regular.

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?

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

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

$$\begin{aligned} S &\to A011A \\ A &\to 0A|1A|\varepsilon \end{aligned}$$

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:

$$\begin{aligned} S &\to 1A0 \\ A &\to 0A|1A|\varepsilon \end{aligned}$$

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:

$$\begin{aligned} S &\to TST|\varepsilon \\ T &\to 0|1 \end{aligned}$$

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:

$$S \rightarrow (S)|\varepsilon$$

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

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

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 Context-free grammars, pushdown automata

4.1 Sessions 4 and 5, Exercise 1

Exercise

Let the grammar be

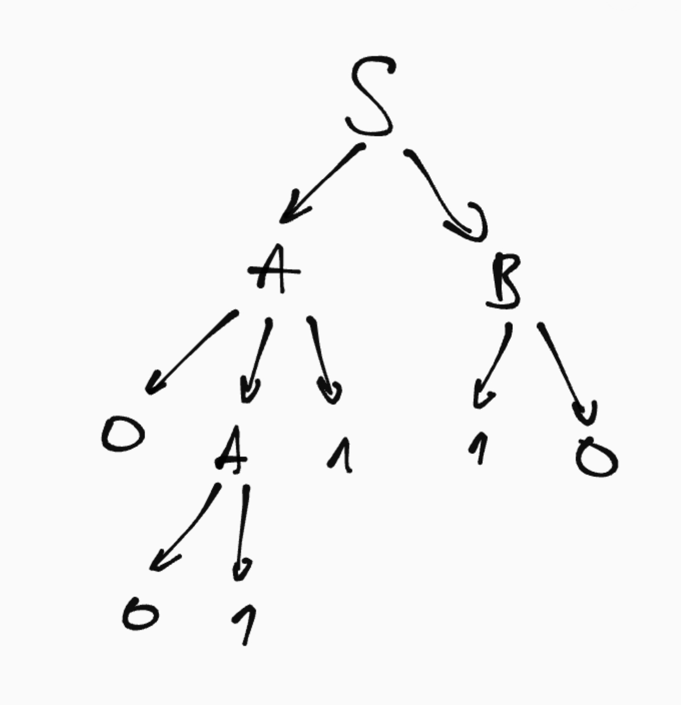
$$
\begin{aligned}
S \to & AB \\
A \to & 0A1|01 \\
B \to & 1B0|10
\end{aligned}
$$

(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

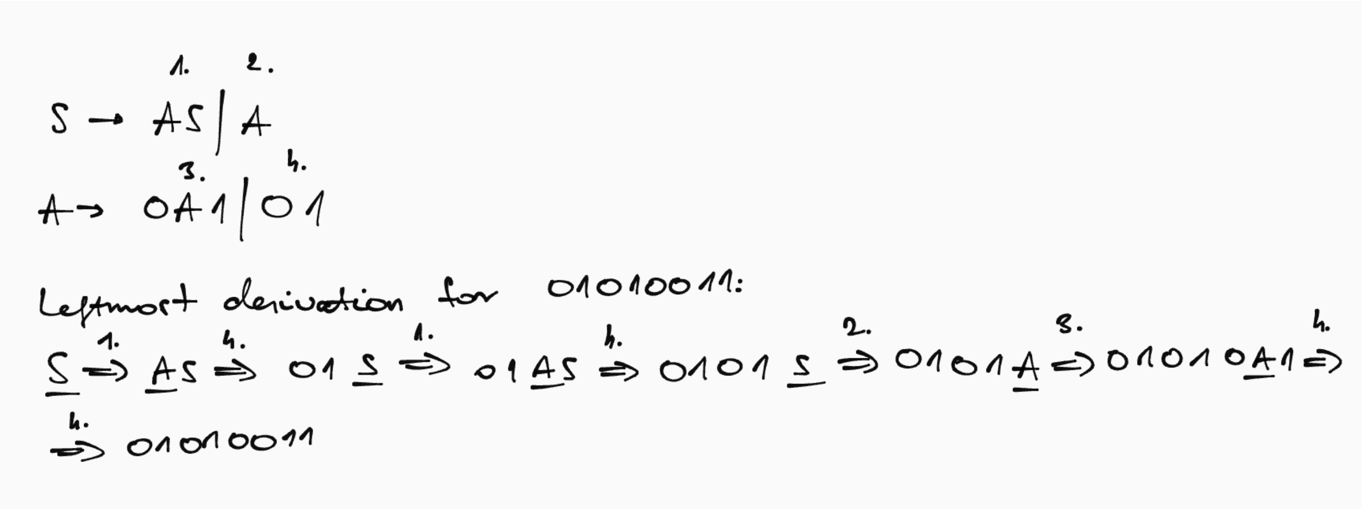
$$\begin{aligned} S \to& AS|A \\ A \to& 0A1|01 \end{aligned}$$

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

(b) Determine the language generated by this grammar.

Solution

a)



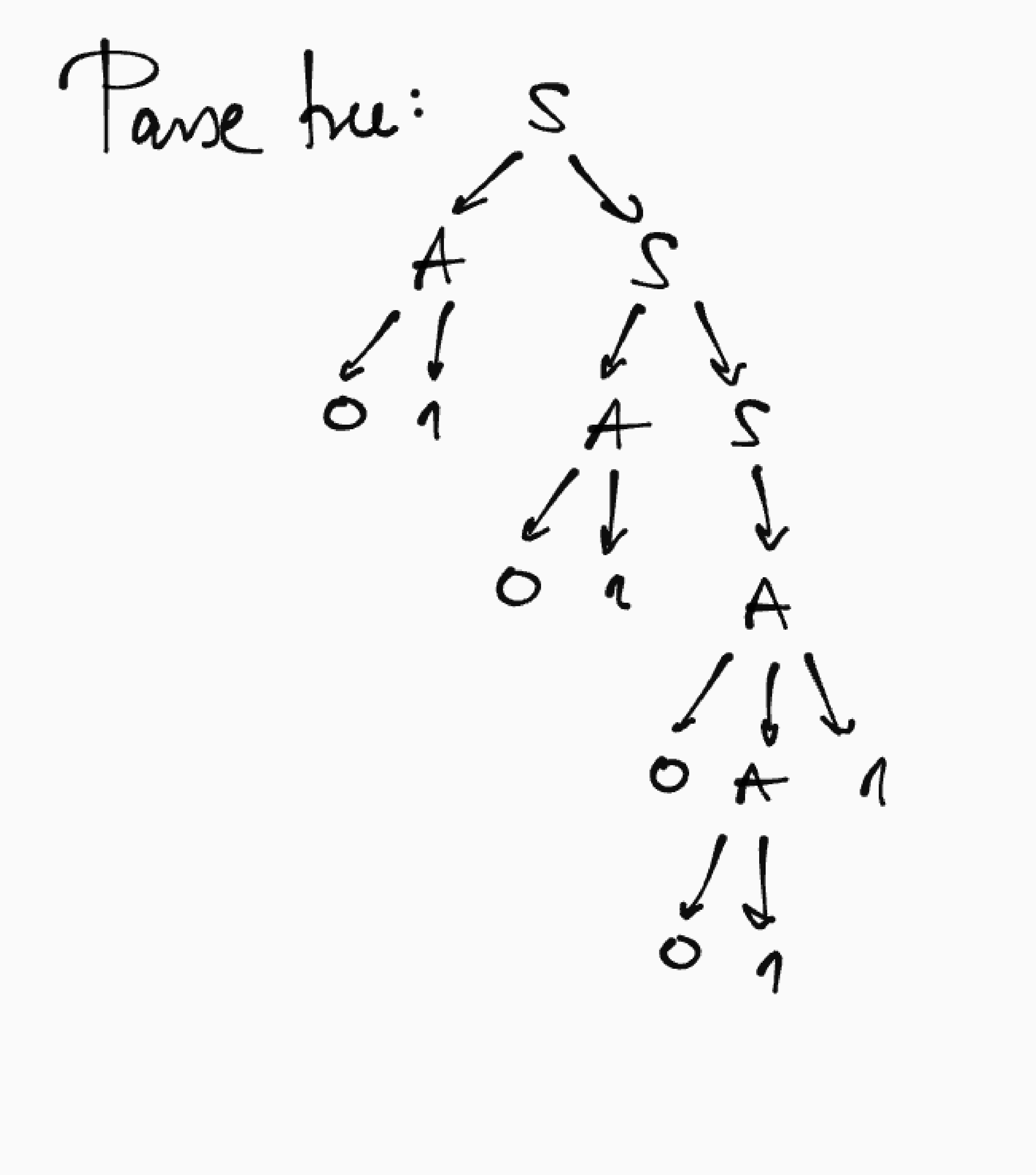
$S \to \overset{1.}{AS} \mid \overset{2.}{A}$

$A \to \overset{3.}{0A1} \mid \overset{4.}{01}$

Leftmost derivation for 01010011:

$\underline{S} \overset{1.}{\Rightarrow} \underline{A}S \overset{4.}{\Rightarrow} 01\underline{S} \overset{1.}{\Rightarrow} 01\underline{A}S \overset{4.}{\Rightarrow} 0101\underline{S} \overset{2.}{\Rightarrow} 0101\underline{A} \overset{3.}{\Rightarrow} 01010\underline{A}1 \Rightarrow$

$\overset{4.}{\Rightarrow} 01010011$



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 $L_S = \{0^{n_1} 1^{n_1} 0^{n_2} 1^{n_2} \ldots 0^{n_k} 1^{n_k} | k \geq 1, n_1, n_2 \ldots, n_k \geq 1\}$.

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:

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

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)

$$\begin{tikzcd}
 & & & (A, \varepsilon, \mathtt{aba}Z) \arrow[r, "3"] & (B, \varepsilon, \mathtt{aba}Z) \arrow[r, "?"] & {} \\
 & & (A, \mathtt{0}, \mathtt{ba}Z) \arrow[ur, "1"] \arrow[dr, "3"] & & & \\
 & (A, \mathtt{10}, \mathtt{a}Z) \arrow[ur, "2"] \arrow[dr, "3"] & & (B, \mathtt{0}, \mathtt{ba}Z) \arrow[r, "?"] & {} & \\
(A, \mathtt{010}, Z) \arrow[ur, "1"] \arrow[dr, "3"] & & (B, \mathtt{10}, \mathtt{a}Z) \arrow[r, "?"] & {} & & \\
 & (B, \mathtt{010}, Z) \arrow[r, "6"] & (C, \mathtt{010}, \varepsilon) \arrow[r, "?"] & {} & &
\end{tikzcd}$$

b)

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

$(A, 0110, Z) \overset{1.}{\Rightarrow} (A, 110, aZ) \overset{2.}{\Rightarrow} (A, 10, baZ) \overset{3.}{\Rightarrow}$

$\overset{3.}{\Rightarrow} (B, 10, baZ) \overset{5.}{\Rightarrow} (B, 0, aZ) \overset{4.}{\Rightarrow} (B, \varepsilon, Z) \overset{6.}{\Rightarrow}$

$\overset{6.}{\Rightarrow} (\underset{✓}{\underline{C}}, \underset{✓}{\underline{\varepsilon}}, \varepsilon)$ ✓

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)

$$S \to aSa|bSb|\varepsilon$$

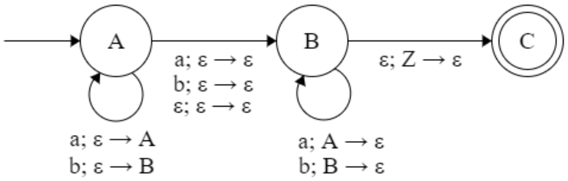
We have shown in previous exercises that this is indeed the grammar for palindromes of even length.

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.

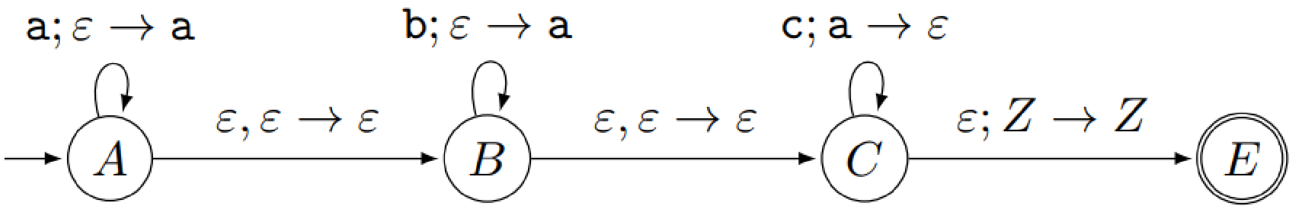
a) $L_a = \{a^i b^j c^k | i, j, k \geq 0 \text{ and } i + j = k\}$

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

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

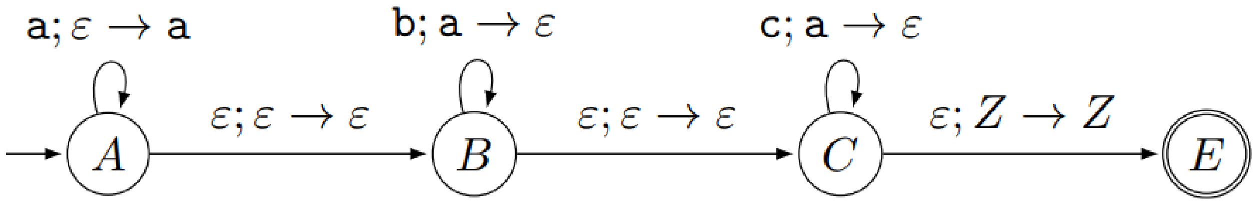
Solution

a)



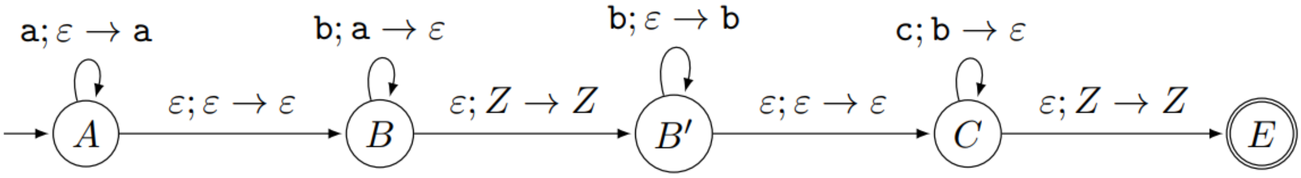
TODO Proof

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).

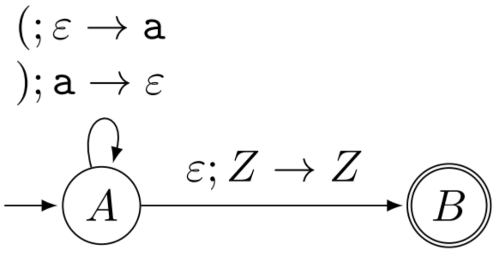
4.8 Sessions 4 and 5, Exercise 8

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.

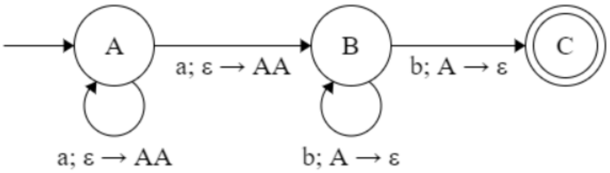
a) $L_a = \{a^n b^m | 2n = m \geq 1\}$

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.



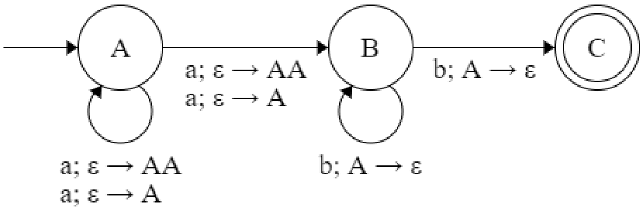
To enforce that the loops run at least once, due to the requirement that $n, m \geq 1$ ($n \geq 0.5$ is the same as $n \geq 1$, since n is an integer), we used non-epsilone transitions between the states, and copied the loop's transition to the existing transitions from the states as well.

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 Turing-machines

5.1 Session 5, Exercise 1

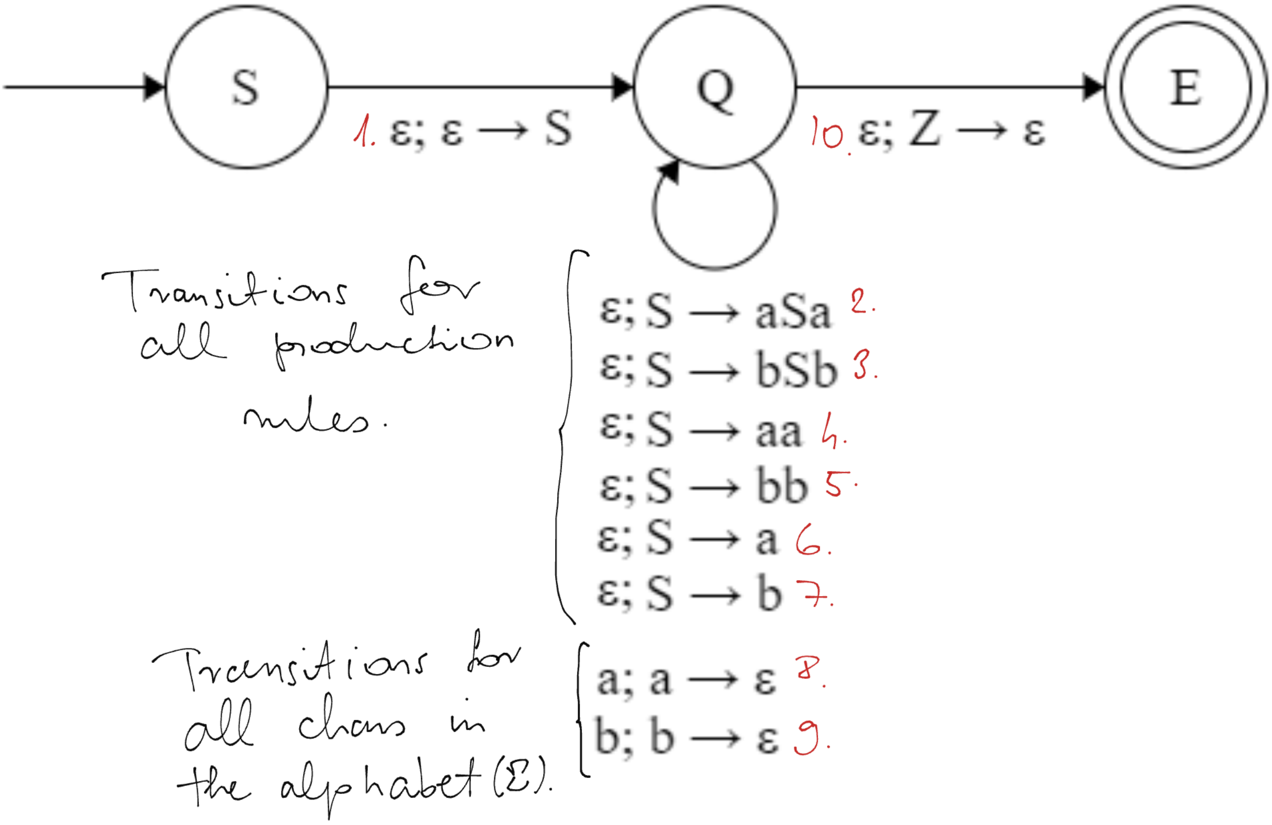
Exercise

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

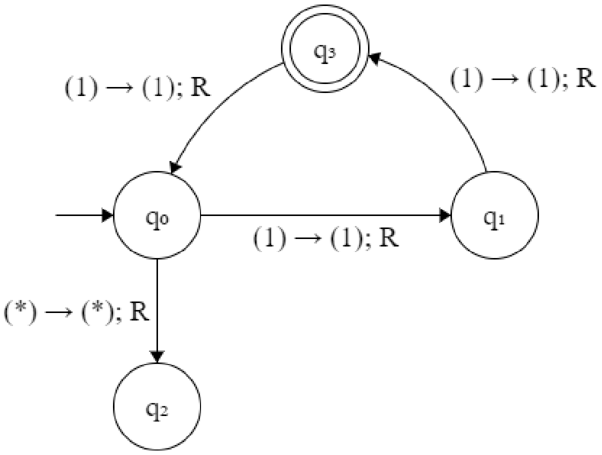
Let the transition function of the 1-tape Turing-machine be

$$\begin{aligned}
\delta(q_0, 1) &= (q_1, 1, R) \\
\delta(q_0, *) &= (q_2, *, R) \\
\delta(q_1, 1) &= (q_3, 1, R) \\
\delta(q_3, 1) &= (q_0, 1, R)
\end{aligned}$$

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:

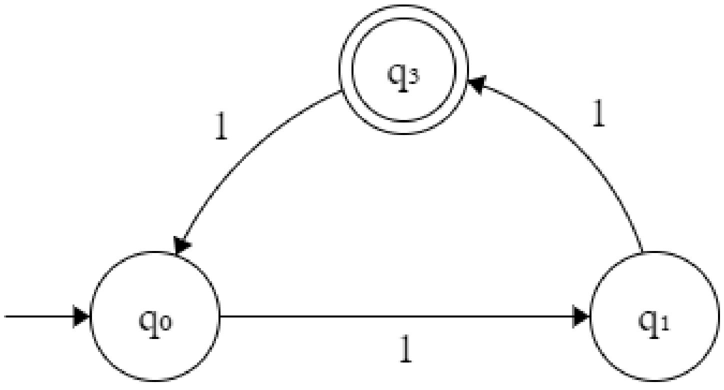


It the q_0, q_1, q_3 cycle reads the input word on the tape from left to right and counts the number of 1's modulo 3. In q_0 the remainder is 0, in q_1 the remainder is 1 and in q_3 the remainder is 2.

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

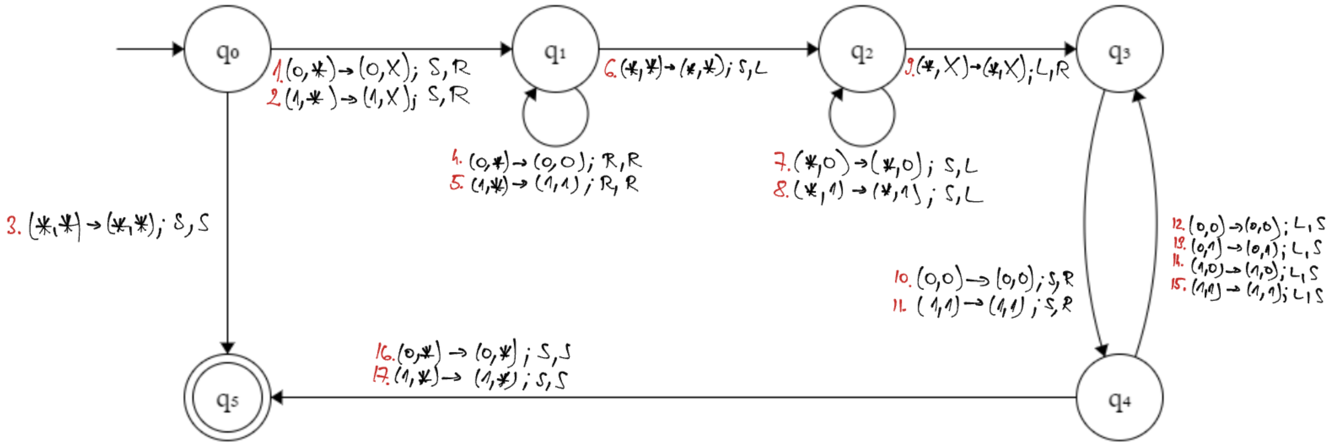
a) What is the content of tape 2 when the machine moves to state q_2?

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

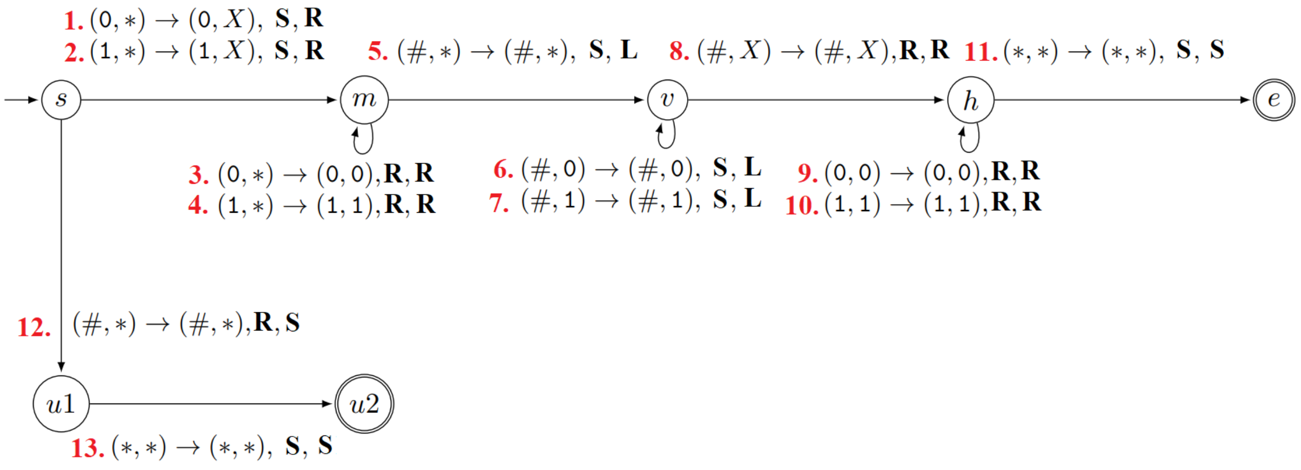
Give a Turing-machine for the language

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

Give an upper bound for the running time.

Solution

This is the Turing-machine:



1. $(0,*) \to (0,X)$, S, R
2. $(1,*) \to (1,X)$, S, R
5. $(\#,*) \to (\#,*)$, S, L
8. $(\#,X) \to (\#,X)$, R, R
11. $(*,*) \to (*,*)$, S, S
s
m
v
h
e
3. $(0,*) \to (0,0)$, R, R
4. $(1,*) \to (1,1)$, R, R
6. $(\#,0) \to (\#,0)$, S, L
7. $(\#,1) \to (\#,1)$, S, L
9. $(0,0) \to (0,0)$, R, R
10. $(1,1) \to (1,1)$, R, R
12. $(\#,*) \to (\#,*)$, R, S
u1
u2
13. $(*,*) \to (*,*)$, S, S

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

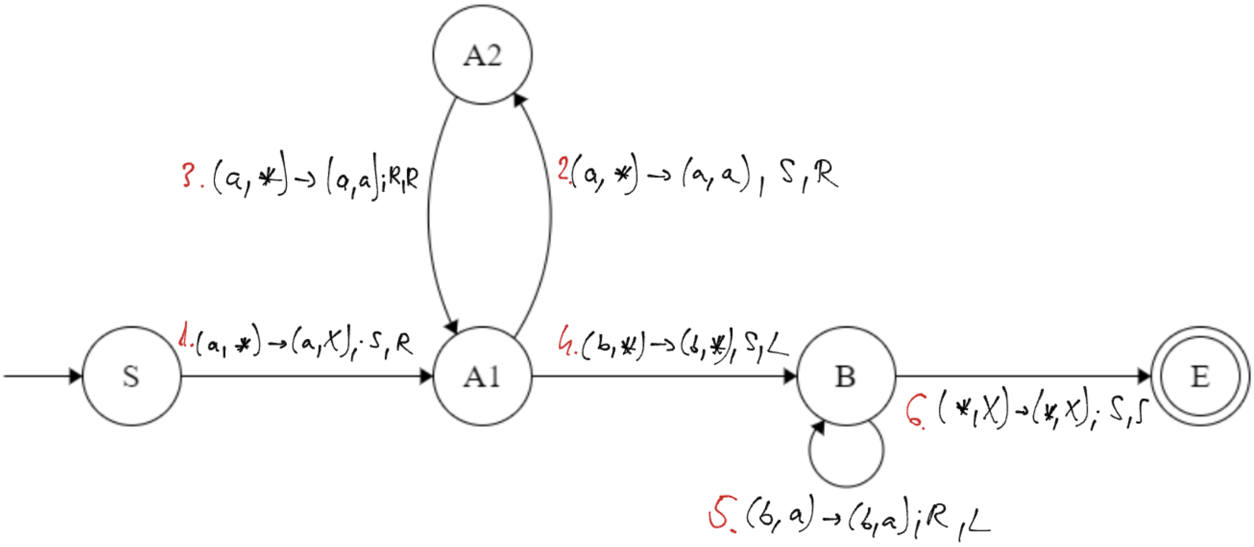
b) $L_2 = \{a^i b^j c^k | i, j, k \geq 1, i + j = k\}$

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

Solution

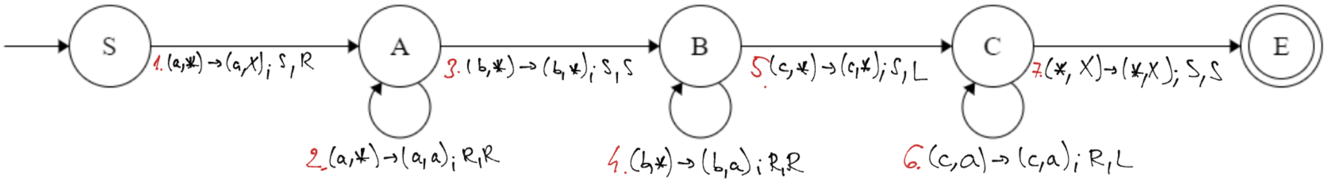
I'm giving the TM's precisely, but you could just describe them. (Basically what the itemized list contains below, without referencing the specific index of the transitions.)

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 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 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. → These must be accepted.
- A description of an undirected graph that contains a cycle. → These must be rejected.
- An input that does not represent an undirected graph. → 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 $MAXCLIQUE = \{(G, k)$: undirected graph G has a clique of size k$\}$ is in NP.

b) Language $5CLIQUE = \{G$: 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 NP-completeness

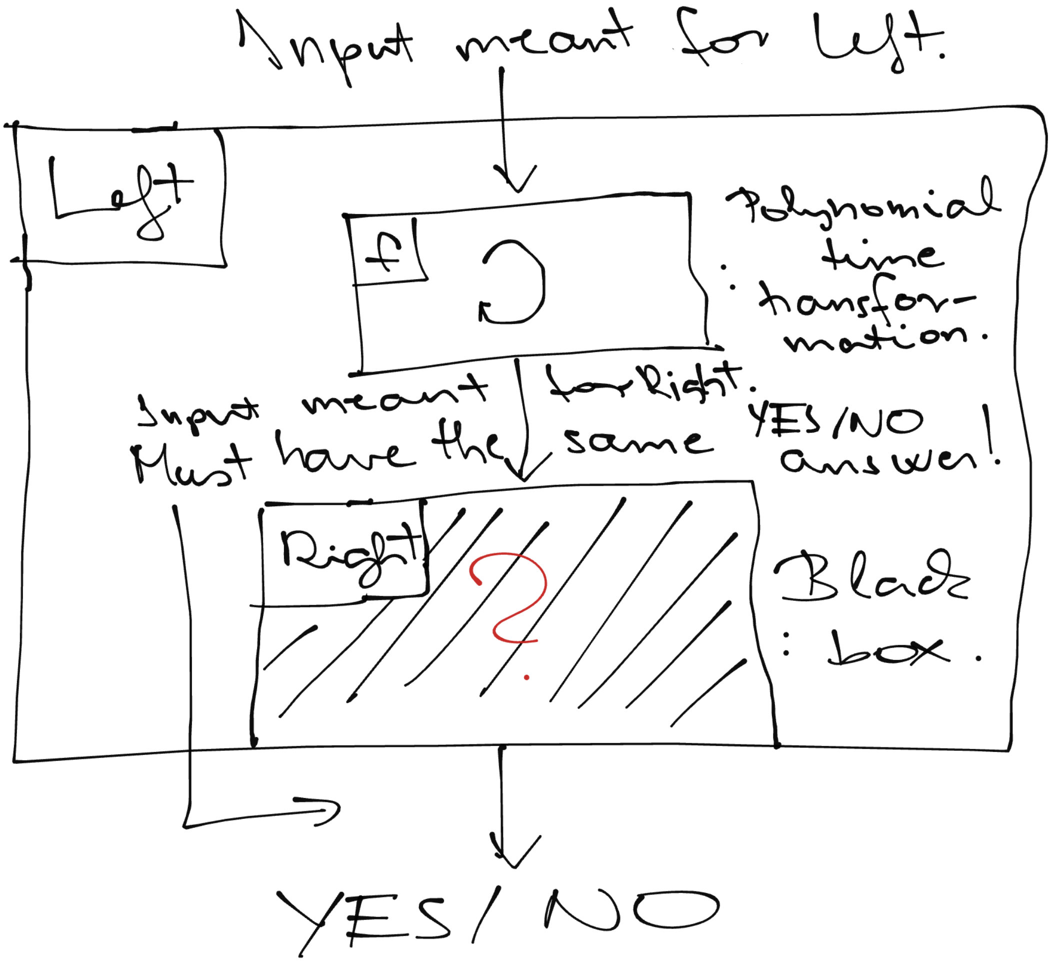
8.1 What is a Karp-reduction?

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?

  → 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!

  → 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?

  → 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?

  → 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.

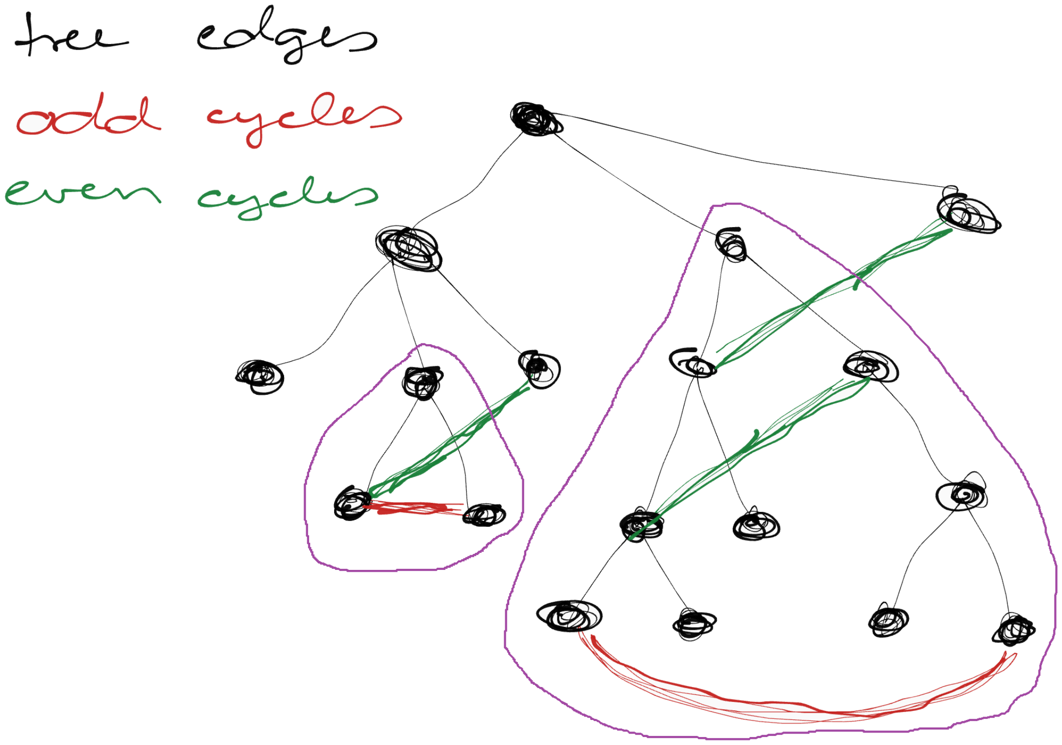
First solution (existence of $NP \prec NP$-hard)

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:

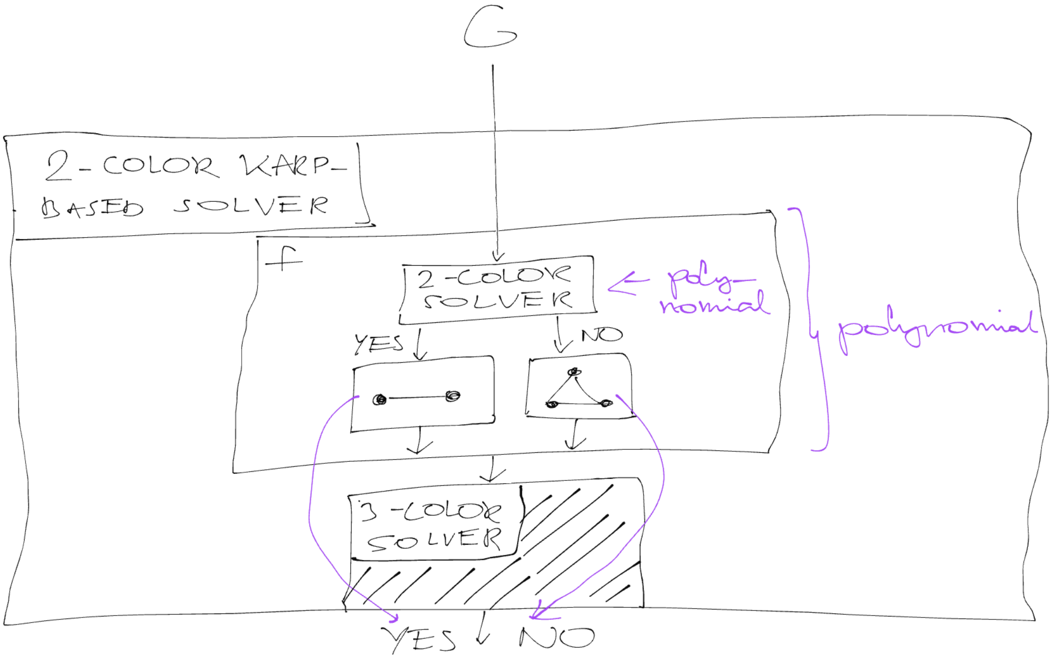
- $3-COLOR \in NP$-complete, which means it is both true that $3-COLOR \in NP$, and $3-COLOR \in NP$-hard.
- Also $2-COLOR \in P \subseteq NP$.
- The definition of NP-hard is that a Karp-reduction exists from all languages that are in NP.
- So $2-COLOR \in NP$ and $3-COLOR \in NP$-hard means the $2-COLOR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

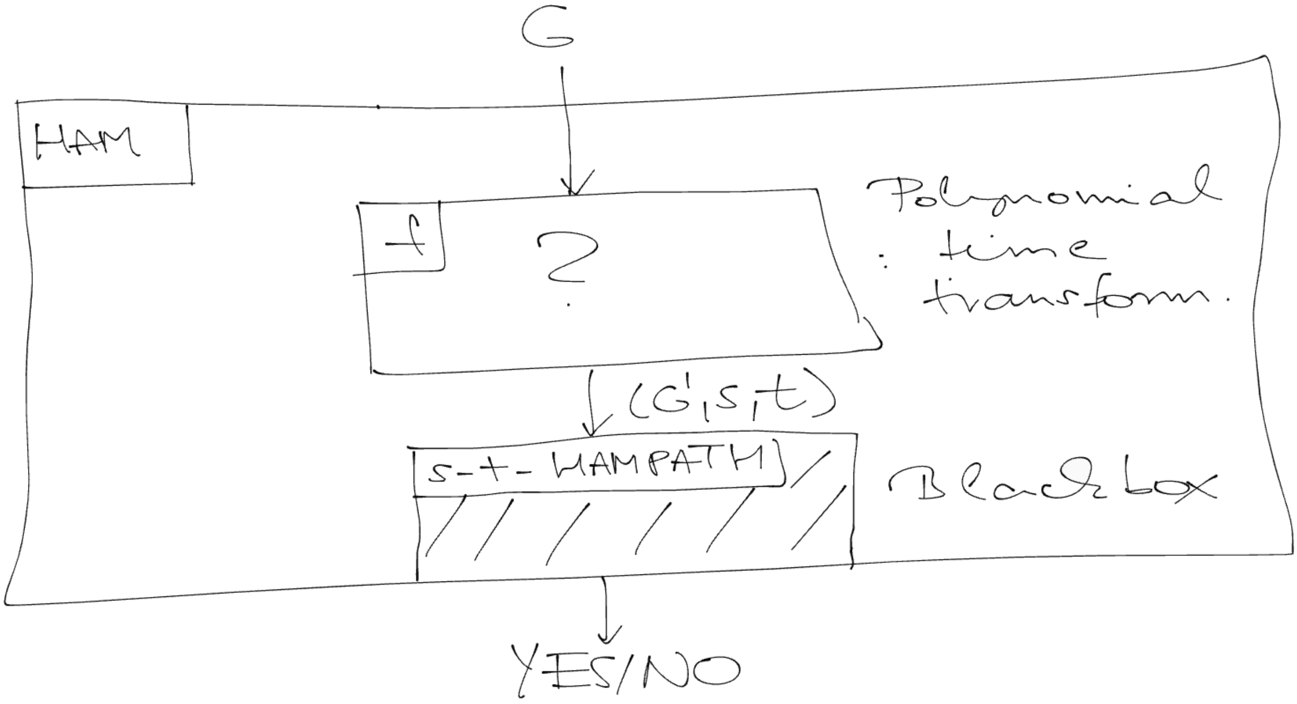
8.5 Session 8, Exercise 4

Exercise

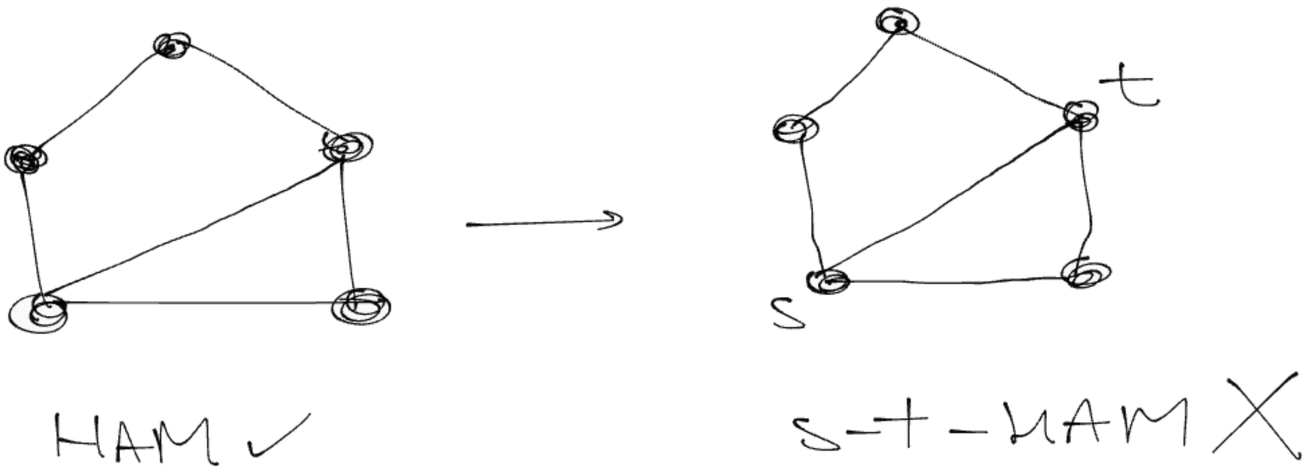
Give a $HAM \prec s - t - HAMPATH$ Karp-reduction.

Solution

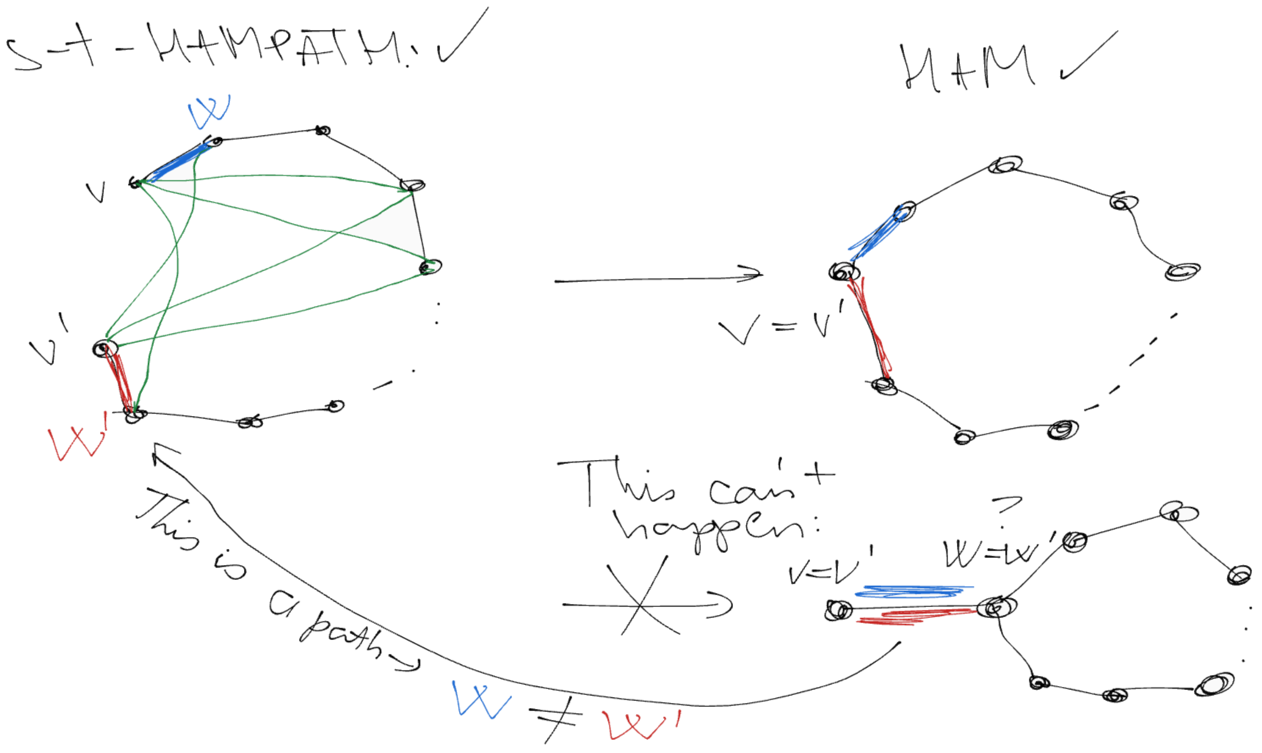
So we need to solve HAM, using $s - t - HAPATH$, e.g.:



- 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):

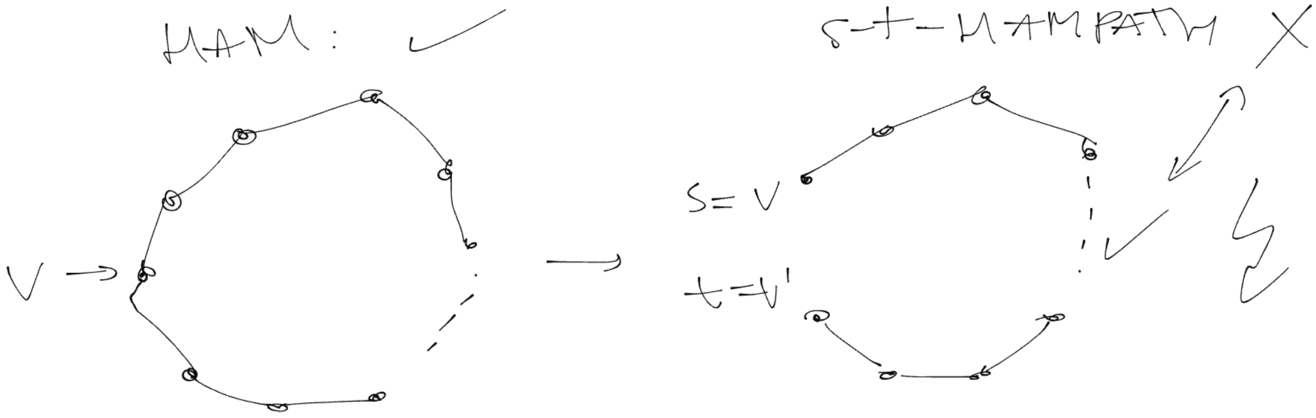


- The left graph has a Hamiltonian-cycle, but the right graph doesn't have an $s-t-$Hamiltonian-path.
- The issue here is that we tried to use different vertices and the edge between those vertices may or may not be part of the Hamiltonian-cycle.
- To fix, we are going to choose the same vertex twice! 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 $\overline{\overline{L1}} \in coNP \to L_1 \in coNP$.

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)

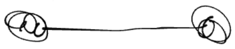
- Is an NP-complete problem Karp-reducible to another NP-complete problem? Yes.
- NP-complete means it is both in NP and NP-hard.
- Let's use the NP fact for the left side, the NP-hard fact for the right side.
- Is a language that is in NP Karp-reducible to another that is NP-hard? Yes. This is the definition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

b)

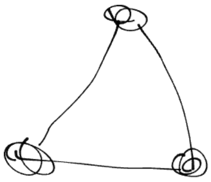
- Is a language in P Karp-reducible to a language that is NP-complete?
- P is a subset of NP, so the question is exactly the same as in a): left side is in NP, right side is NP-hard, so the Karp-reduction must exist. (See Session 8, Exercise 1, second solution for a different approach, for when you actually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

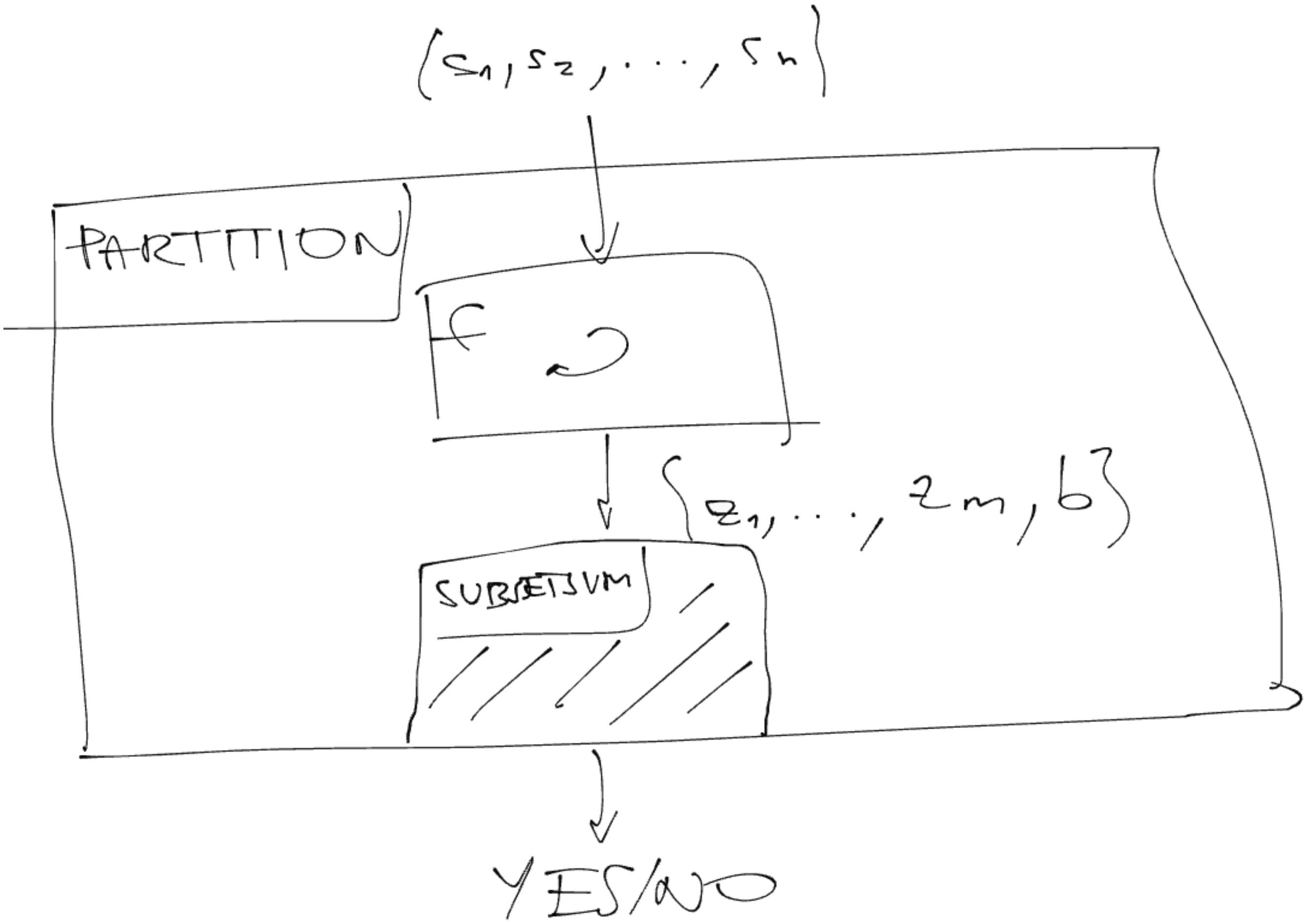


8.8 Session 8, Exercise 7

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

10

11

12

13 Binary search trees, 2-3-trees, hash

13.1 Session 13, 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?

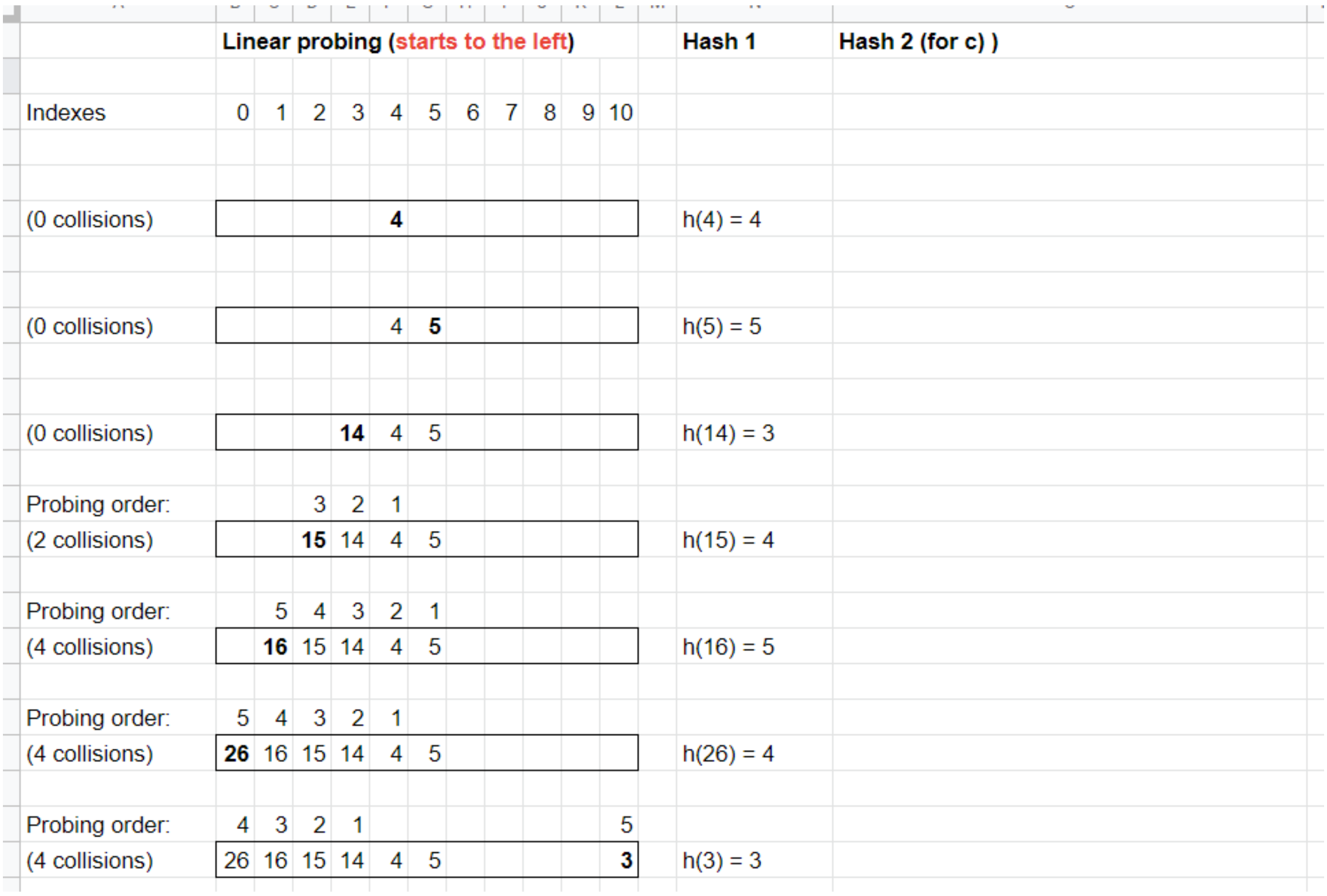
Solution

All types of open hashings try in the original $h(x)$ position first. When a collision happens, they try the following offsets:

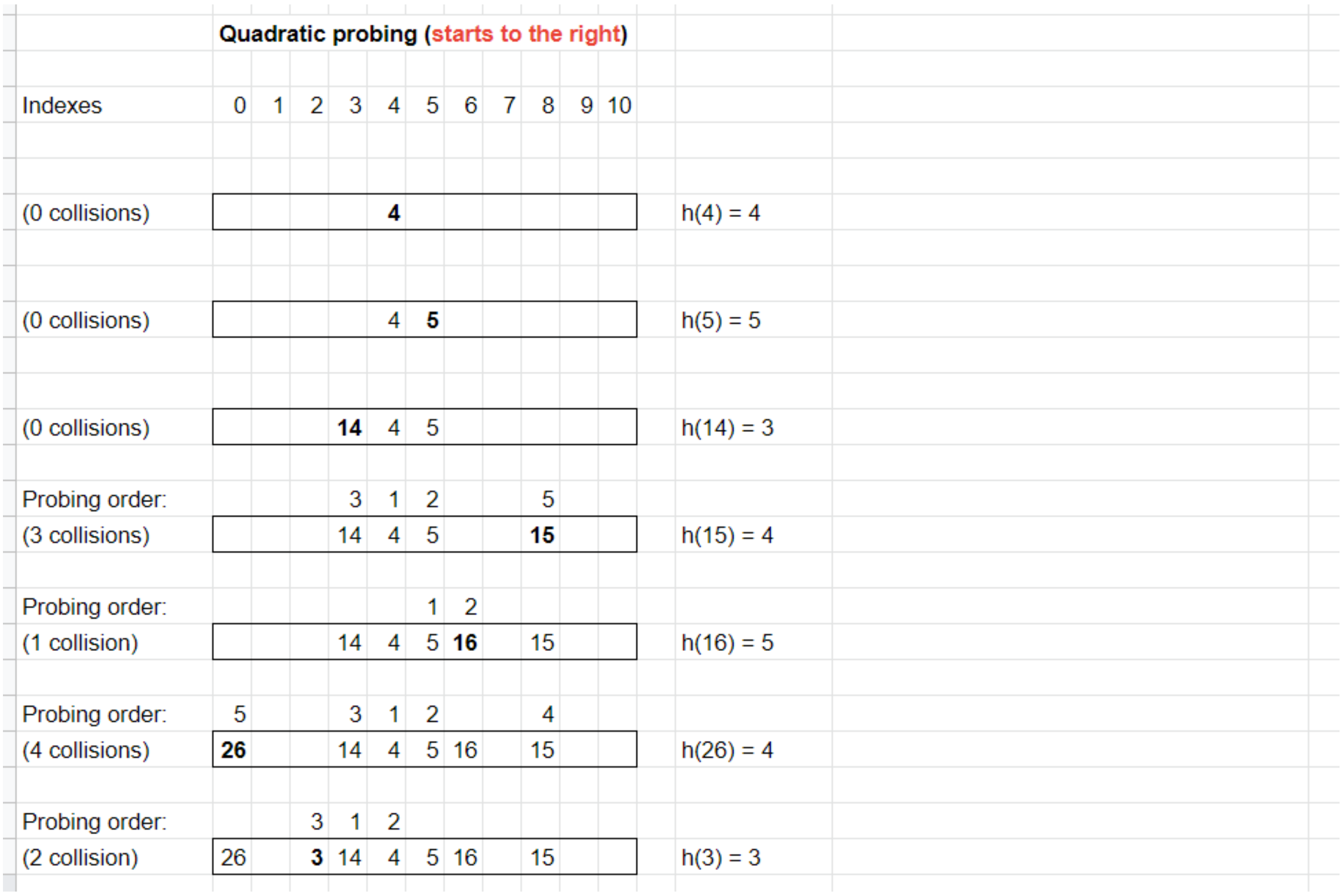
- Linear probing starts to the left, and tries the offsets $-1, -2, -3, \ldots$ etc.
- Quadratic probing starts to the **right** (!), and tries the offsets $+1, -1, +4, -4, +9, -9\ldots$ etc (square numbers).
- Double hasing uses a secondary hash function, h'(x) and tries the offsets $-1 \cdot h'(x), -2 \cdot h'(x), -3 \cdot h'(x), \ldots$ 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.)

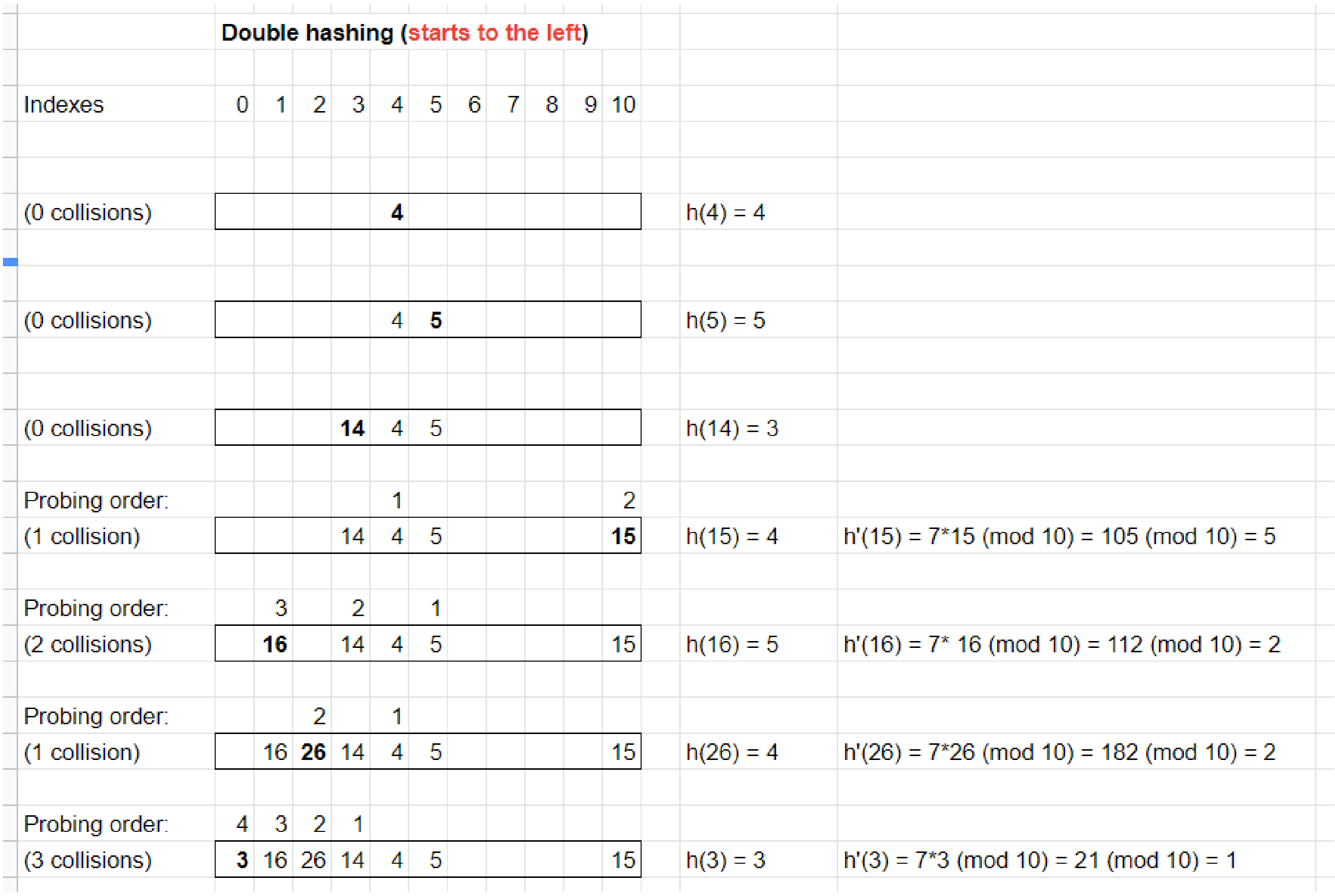
The offsets are not cumulative, all of them are applied to the original h(x).



	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

14 Exam: 2022. 05. 30.

14.1 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 $f(n^2f(n) + 3f(n) + 5) \in O(n^4)$?

Solution

It does **not** imply!

- What does it mean that $f(n) \in O(n^2)$ implies $f(n^2f(n) + 3f(n) + 5) \in O(n^4)$? It means that in all cases where $f(n) \in O(n^2)$ is true, $f(n^2f(n) + 3f(n) + 5) \in O(n^4)$ must also be true. If we can find one example, where $f(n) \in O(n^2)$ is true, however $f(n^2f(n)+3f(n)+5) \in O(n^4)$ 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, $f(n^2f(n)+3f(n)+5) \in O(n^4)$ 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 $n \geq 1$)!
- $f(n^2f(n) + 3f(n) + 5) = (n^2f(n) + 3f(n) + 5)^2 = (n^4 + 3n^2 + 5)^2$
- 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 $f(n^2f(n) + 3f(n) + 5) \in O(n^4)$ is not true!
- For example, take $f(n) = \sqrt{n}$, which is $\in O(n^2)$, monotone increasing and non-negative for $n \geq 1$.
- $f(n^2f(n)+3f(n)+5) = \sqrt{n^2f(n) + 3f(n) + 5} = \sqrt{\sqrt{n}^3 + 3\sqrt{n} + 5} \leq \sqrt{n}^3 + 3\sqrt{n} + 5 \leq \sqrt{n}^3 + 3\sqrt{n}^3 + 5\sqrt{n}^3 = 9\sqrt{n}^3 = 9n^{1.5} \in O(n^2)$, 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 $f(n^2f(n) + 3f(n) + 5) \in O(n^4)$ is true or false. It does not imply its trueness or faleness either.

14.2 Exam: 2022. 05. 30., Exercise 2

Exercise

Give a context-free grammar for the language

$$L_2 = \{a^{2n}b^nc^md^{3m} \mid n \geq 1; m \geq 0\}$$

over alphabet $\{a, b, c, d\}$.

Solution

14.3 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

14.4 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

14.5 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

14.6 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

14.7 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

15 Exam: Miscellaneous

15.1 Exams, Exercise 1

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 $O(n^1 0)$ 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: $k(m) = c_1(c_0n^2 + S_n)^5 + S_m = c_1c_0^5n^{10} + c_1S_n^5 + S_m$.
- Then, just upper estimate again by multiplying the constant parts with n^10 as well: $k(m) = c_1c_0^5n^{10}+c_1S_n^5+S_m \leq c_1c_0^5n^{10} + c_1S_nn^{10} + S_mn^{10} = (c_1c_0^5 + c_1S_n + S_m)n^{10}$. Choose $c = (c_1c_0^5 + c_1S_n + S_m)$ and $n_0 = 1$ and this proves that $k(n) \in O(n^{10})$.

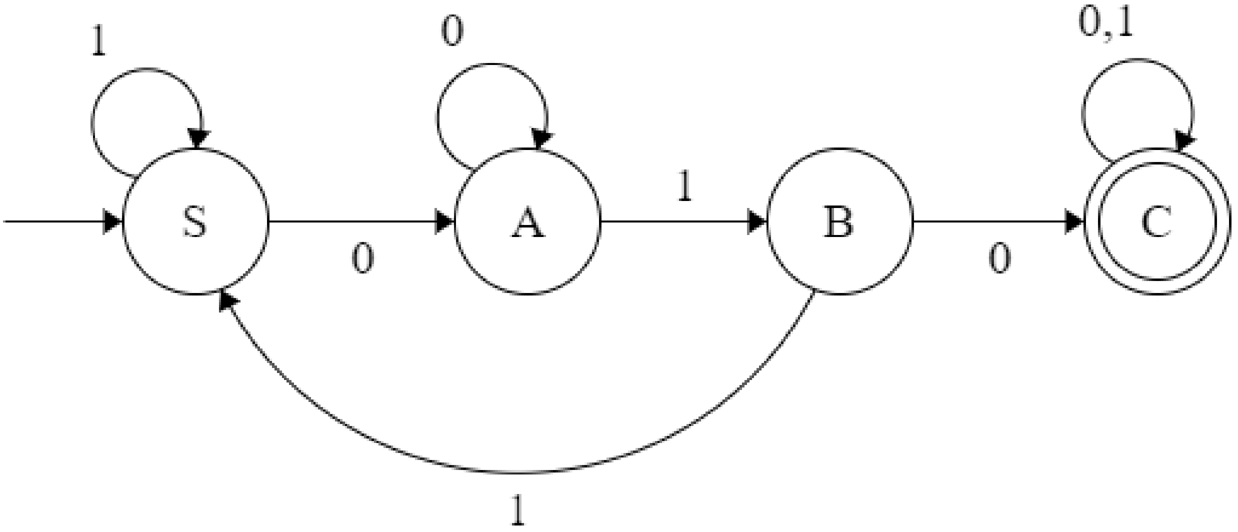
15.2 Exams, Exercise 2

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):



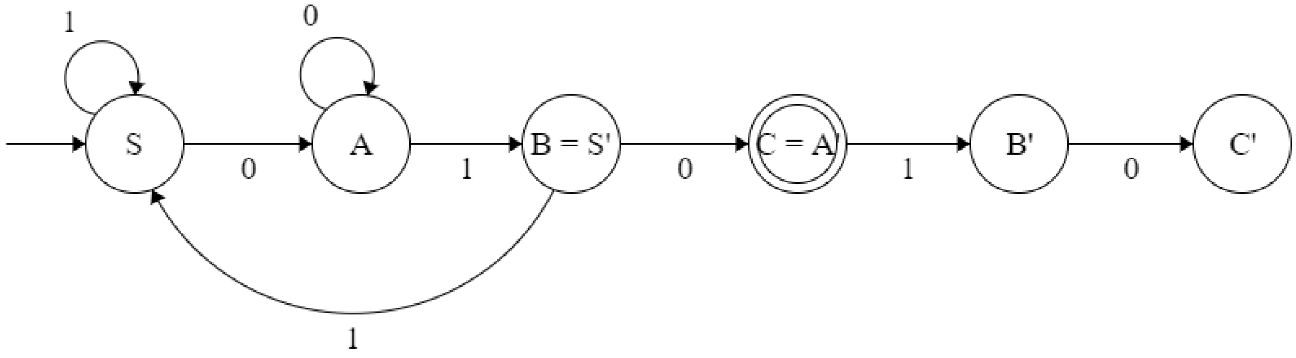
All of the states represent how long of a prefix we have received so far: S for no useful prefix, A for the "0" being the last character we have read and possibly continuing with "10" after, B for the 10 word being the last two characters we have read and possibly continuing with "0" and finally C means 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.

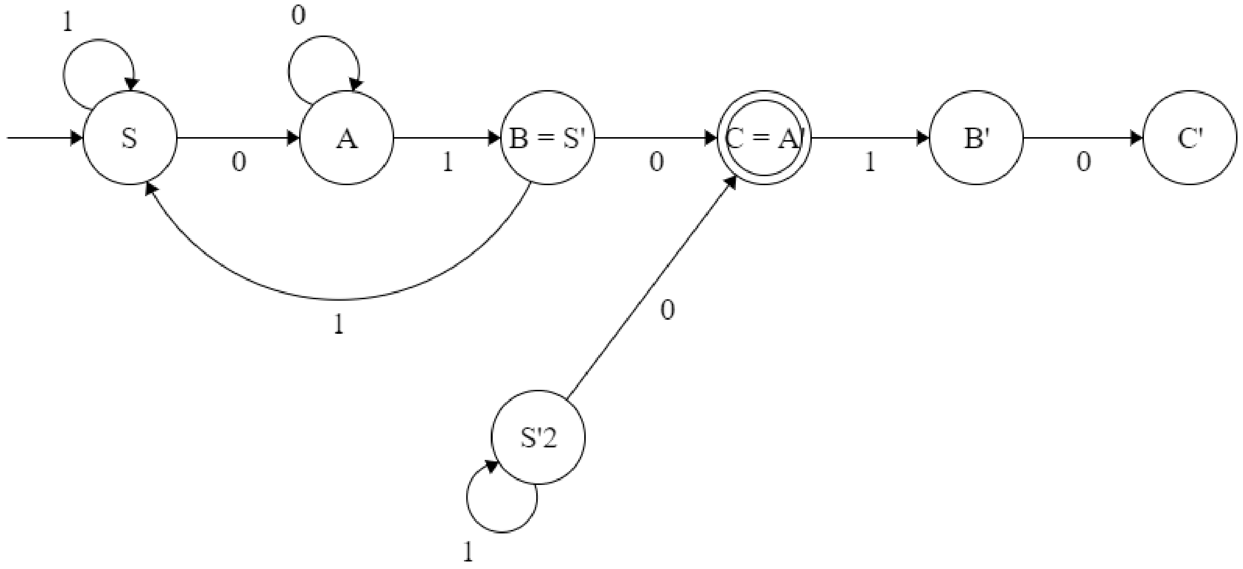
This means that if we start trying to find another 010 there, in a similar manner, we can detect a second 010. We cannot start in C, however, C should already mean the 0 was received, so we kind of copy and paste this DFA onto itself, the next copy having $S' = BandA' = C$, like so:

(And remove the looping on the original C.)



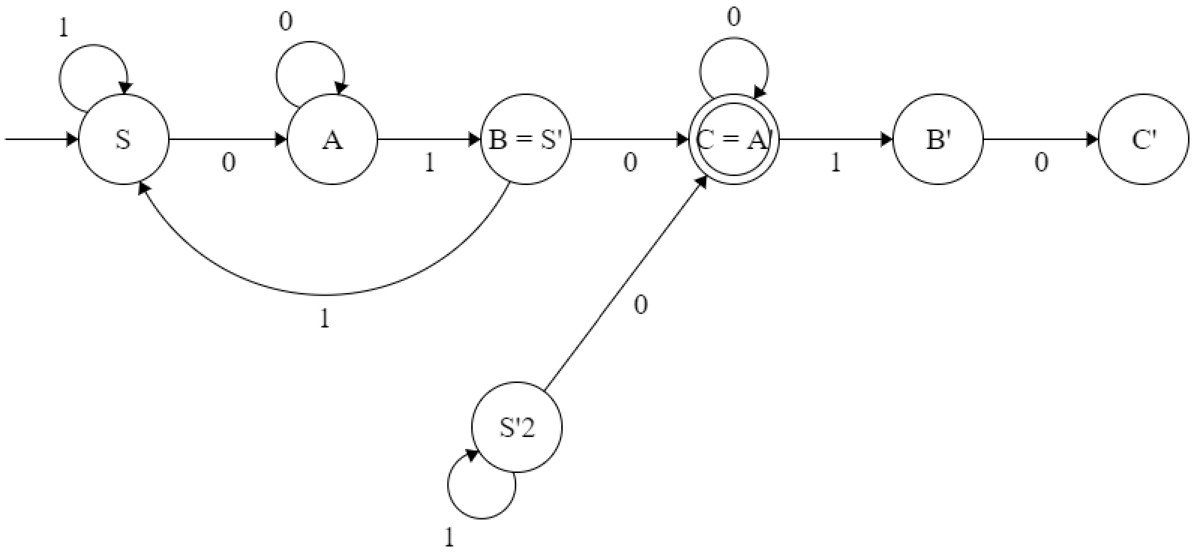
I have added the forward pointing transitions, and now we need to be really careful about the backward pointing ones:

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:

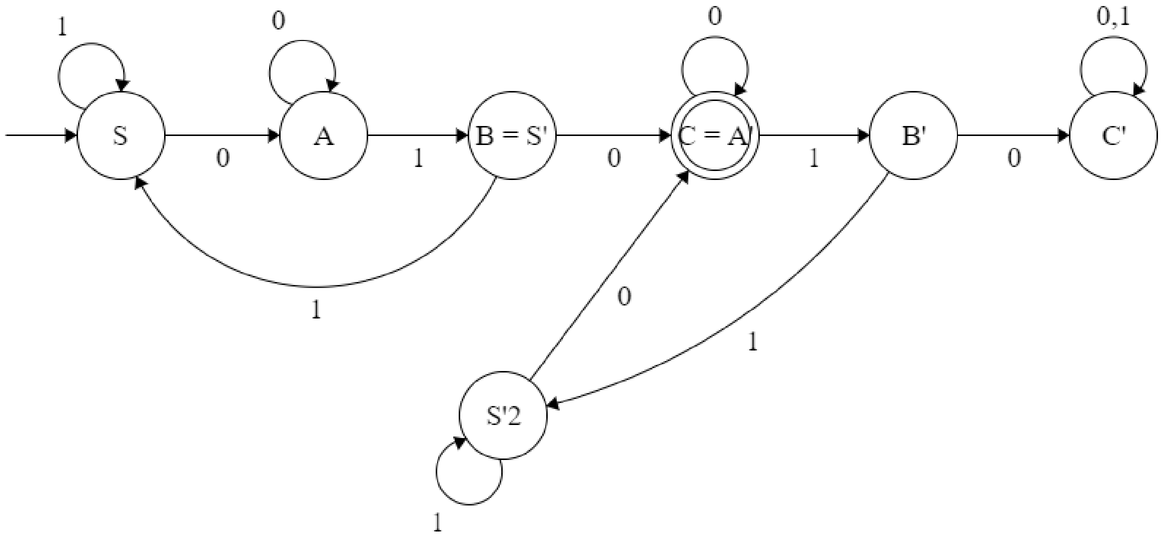


Whenever in the later parts we would move back to S', instead we will use S'_2 so that will correctly loop with the 1's instead of throwing us back at the very beginning.

Now I can add the 0 loop to $C = A'$:



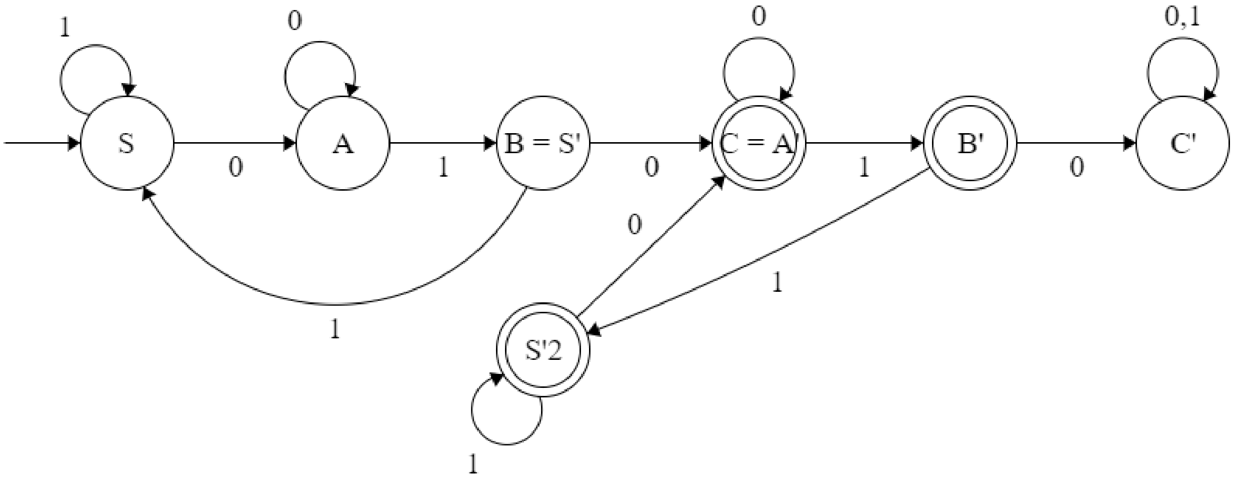
Then in B', if we read a 1 we move back to S, and in this case we are going to use our S_2' state, so it cannot escape back to S, which we cannot allow:



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.