Automata, Spring 2022, Solutions to Final Exam

Problem 1. *Finite automata and regular languages.*

Problem 1a. Design a finite automaton for recognizing words that contain the letter a. Assume that the input strings contain only symbols a and b. The easiest is to have two states:

- the starting state s indicating that we have not yet read any a's; and
- the state f indicating that we have already read a symbol a.

You just need to describe transitions between these states, and which states are final. Show, step-by-step, how your automaton will accept the word $abab$.

Solution.

- from s, a leads to f, and b leads back to s;
- from f, both a and b lead back to f.

The final state is f. Derivation of $abab$ is as follows:

$$\begin{array}{ccccccccc} | & a & | & b & | & a & | & b & | \\ s & & f & & f & & f & & f \end{array}$$

Problem 1b. Explain why in most computers binary numbers are represented starting with the lowest possible digit.

Solution. In most actual computers, the representation of a number starts with the least significant digit, since all arithmetic operations like addition, subtraction, or multiplication start with the least significant digit. So, if we store the number the way we write numbers, most significant digits first, computers will have to waste time going through all the digits until they come up with the least significant digit and start the actual computations. To speed up computations, representations therefore start with the least significant digits.

Problem 1c. On the example of the above automaton, show how the word *abab* can be represented as *xyz* in accordance with the pumping lemma.

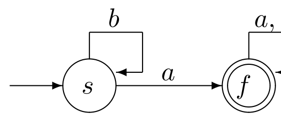
Solution. Here, the first repeating state is f:

$$\begin{array}{ccccccccc} | & a & | & b & | & a & | & b & | \\ s & & \underline{f} & & \underline{f} & & f & & f \end{array}$$

So, $x = a$, $y = b$, and $z = ab$.

Problem 1d. Use a general algorithm to describe a regular expression corresponding to the finite automaton from the Problem 1a. (If you are running out of time, it is Ok not to finish, just eliminate the first state.)

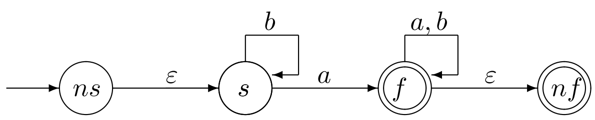
Solution. We start with the described automaton:



According to the general algorithm, first we add a new start state ns and a few final state f, and we add jumps:

- from the new start state ns to the old start state, and
- from each old final state to the new final state nf.

As a result, we get the following automaton.



Then, we need to eliminate the two intermediate states s and f one by one. Let us start with eliminating the state s. In general, we have a formula

$$R'_{i,j} = R_{i,j} \cup (R_{i,k} R^*_{k,k} R_{k,j}),$$

where k is the state that we are eliminating. In this case, we have $k = s$, so

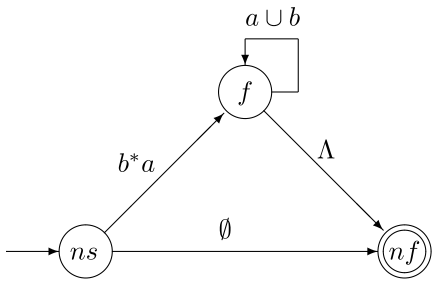
$$R'_{i,j} = R_{i,j} \cup (R_{i,s} R^*_{s,s} R_{s,j}).$$

So, we have the following:

$$R'_{ns,f} = R_{ns,f} \cup (R_{ns,s} R^*_{s,s} R_{s,f}) = \emptyset \cup (\Lambda b^* a) = b^* a;$$

$$R'_{ns,nf} = R_{ns,nf} \cup (R_{ns,s}R^*_{s,s}R_{s,nf}) = \emptyset \cup (\Lambda b^* \emptyset) = \emptyset \cup \emptyset = \emptyset;$$
$$R'_{f,f} = R_{f,f} \cup (R_{f,s}R^*_{s,s}R_{s,f}) = a \cup b \cup (\emptyset \ldots) = a \cup b;$$
$$R'_{f,nf} = R_{f,nf} \cup (R_{f,s}R^*_{s,s}R_{s,nf}) = \Lambda \cup (\emptyset \ldots) = \Lambda \cup \emptyset = \Lambda.$$

Thus, the 3-state a-automaton takes the following form:



The final expression is the corresponding expression for $R'_{ns,nf}$:

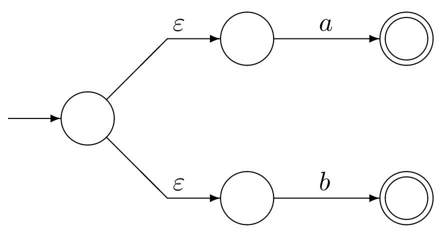
$$R'_{ns,nf} = R_{ns,nf} \cup (R_{ns,f}R^*_{f,f}R_{f,nf}) =$$
$$\emptyset \cup (b^*a(a \cup b)^*\Lambda) =$$
$$b^*a(a \cup b)^*.$$

The last expression is a regular expression corresponding to the original automaton.

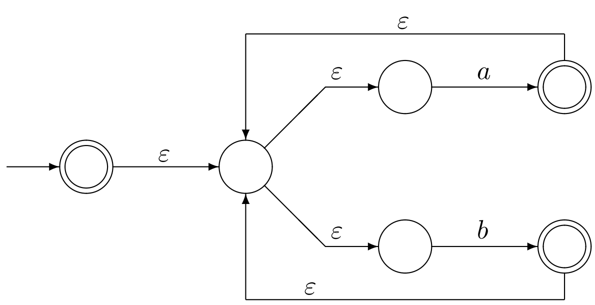
Problem 1e-f. The resulting language can be described by a regular expression $(a \cup b)^* ab^*$. Use a general algorithm to transform this regular expression into a finite automaton: first a non-deterministic one, then a deterministic one.

Solution. We start with the standard non-deterministic automata for recognizing the words a and b:

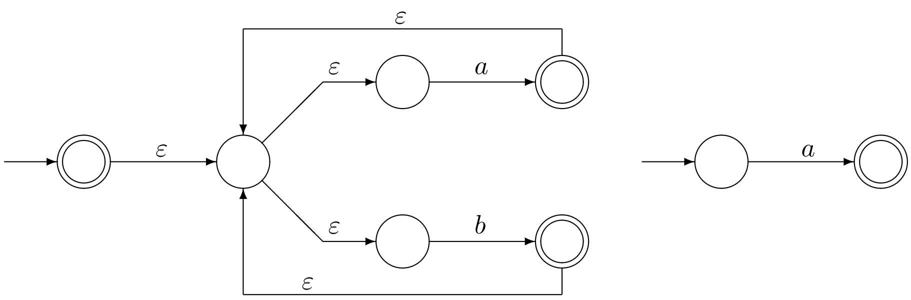
Then, we use the general algorithm for the union to design a non-deterministic automaton for recognizing the language $a \cup b$:



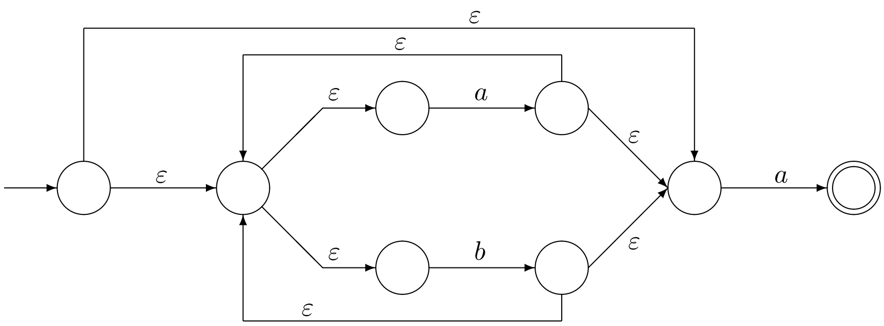
Now, we apply a standard algorithm for the Kleene star, and we get the following non-deterministic automaton for $(a \cup b)^*$:



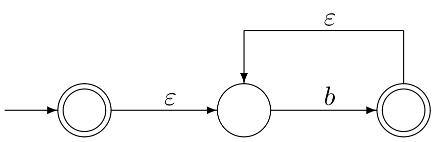
Now, we also take a standard automaton for the language a, and use the algorithm for concatenation for combine them:



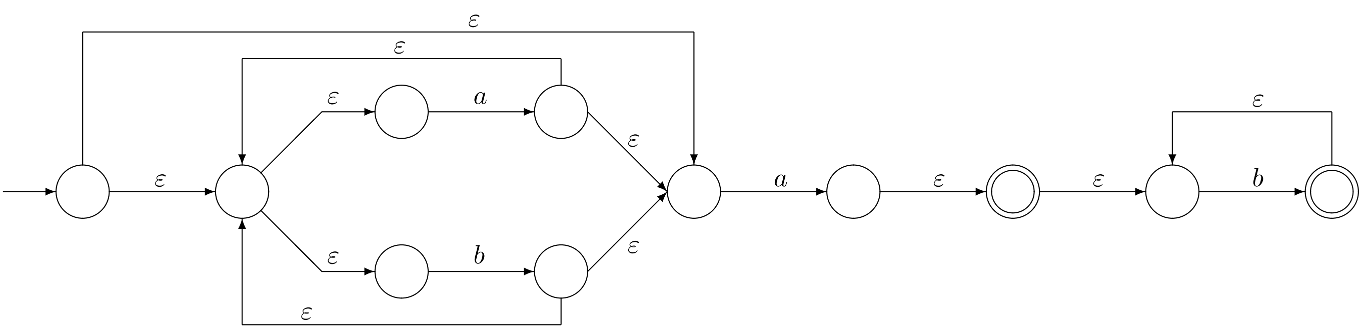
Then, we get:



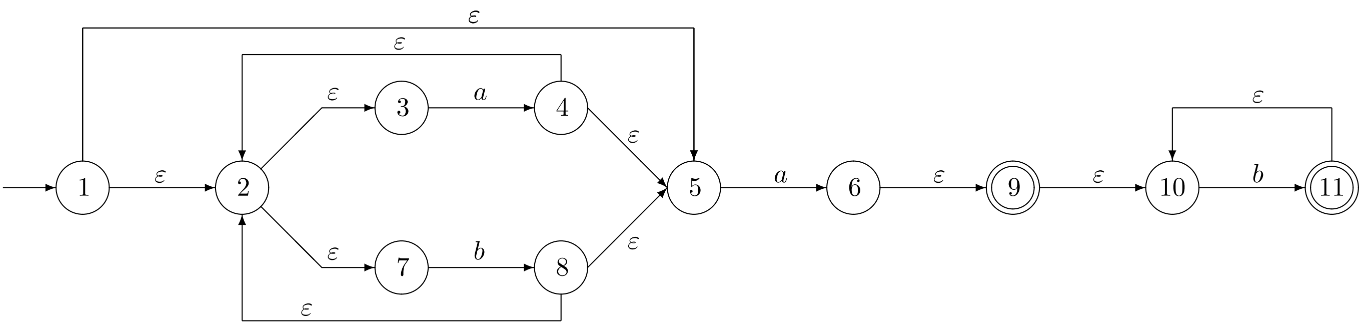
Then, we use the algorithm for Kleene star to get an automaton for b^*:



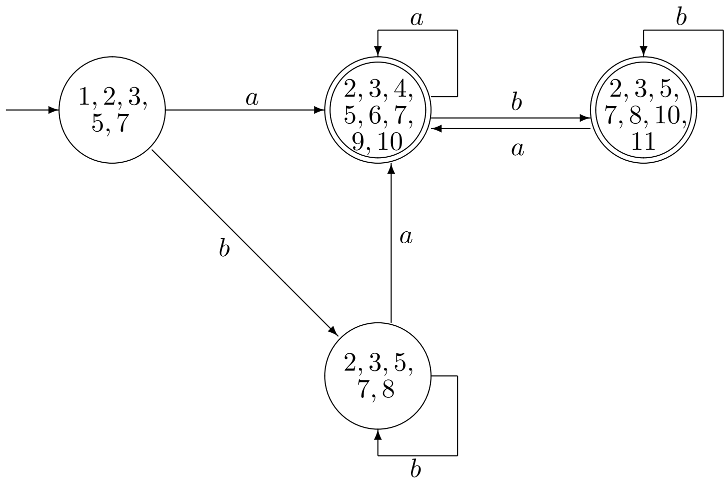
Then, finally, we design an automaton for concatenation



To get a deterministic finite automation, first, we enumerate the states:



Then, we get the following deterministic automaton:



Problem 2. *Beyond finite automata: pushdown automata and context-free grammars*

Problem 2a. Some researchers believe that in the ideal class, there should same number of As and Cs, and twice as many Bs. For example, a sequence ABCB is ideal in this sense, as well as CABCBBAB. Prove that the set of all such "ideal" sequences is not regular and therefore, cannot be recognized by a finite automaton.

Solution We will prove it by contradiction. Let us assume that the language L of all fair sequences is regular, and let us show that this assumption leads to a contradiction.

Since this language is regular, according to the Pumping Lemma, there exists an integer p such that every word from L whose length $\text{len}(w)$ is at least p can be represented as a concatenation $w = xyz$, where:

- y is non-empty;
- the length $\text{len}(xy)$ does not exceed p, and
- for every natural number i, the word $xy^iz \stackrel{\text{def}}{=} xy\ldots yz$, in which y is repeated i times, also belongs to the language L.

Let us take the word

$$w = \text{A}^p\text{B}^{2p}\text{C}^p = \text{A}\ldots\text{AB}\ldots\text{BC}\ldots\text{C},$$

in which first A is repeated p times, then B is repeated $2p$ times, then C is repeated p times. The length of this word is $p + 2p + p = 4p > p$. So, by pumping lemma, this word can be represented as $w = xyz$ with $\text{len}(xy) \le p$. This word starts with xy, and the length of xy is smaller than or equal to p. Thus, xy is among the first p symbols of the word w – and these symbols are all As. So, the word y only has As.

Thus, when we go from the word $w = xyz$ to the word $xyyz$, we add As, and we do not add any Bs or Cs. So, in the word $xyyz$, there are more As than Cs. Thus, the word $xyyz$ cannot be in the language L, since by definition L only contains words which have exactly as many As as Cs.

On the other hand, by Pumping Lemma, the word $xyyz$ must be in the language L. So, we proved two opposite statements:

- that this word *is not* in L and
- that this word *is* in L.

This is a contradiction.

The only assumption that led to this contradiction is that L is a regular language. Thus, this assumption is false, so L is not regular.

Problem 2b. Use a general algorithm to transform the finite automaton from the Problem 1a into a context-free grammar (CFG). Show, step-by-step, how this CFG will generate the word *abab*.

Solution. According to the general algorithm, the corresponding grammar should have two variables S and F, and the following rules:

$$S \to aF, \quad S \to bS, \quad F \to aF, \quad F \to bF, \quad F \to \varepsilon.$$

The word *abab* is accepted by the finite automaton as follows:

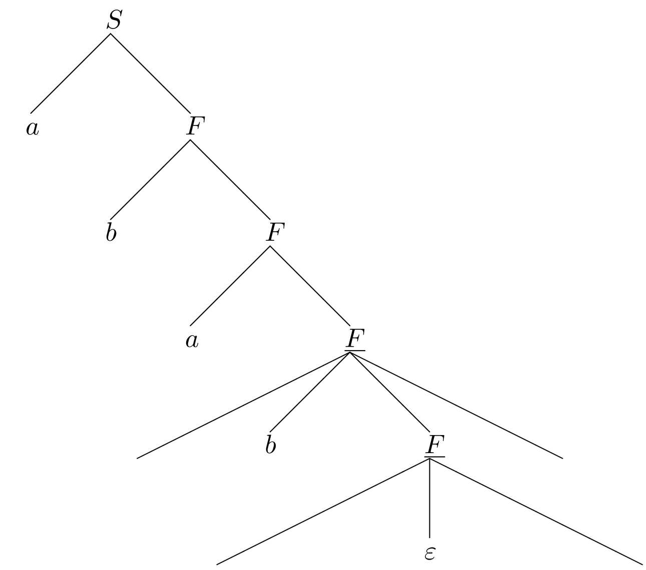
$$\begin{array}{ccccccccc} | & a & | & b & | & a & | & b & | \\ s & & f & & f & & f & & f \end{array}$$

Thus, its derivation takes the following form:

$$\underline{S} \to a\underline{F} \to ab\underline{F} \to aba\underline{F} \to abab\underline{F} \to abab.$$

Problem 2c. For the context-free grammar from the Problem 2b, show how the word *abab* can be represented as $uvxyz$ in accordance with the pumping lemma.

Solution. In terms of a tree, the derivation of the word *abab* can be represented as follows:



Here, $u = aba$, $v = b$, and $x = y = z = \varepsilon$.

Problem 2d. Use a general algorithm to translate the CFG from 2b into Chomsky normal form.

Solution.

Preliminary step. We add a new starting variable S_0 and a rule $S_0 \to S$:

$$S \to aF, \quad S \to bS, \quad F \to aF, \quad F \to bF, \quad F \to \varepsilon, \quad S_0 \to S.$$

Step 0. We need to eliminate the rule $F \to \varepsilon$. This means adding the rules $S \to a$, $F \to a$, and $F \to b$:

$$S \to aF, \quad S \to bS, \quad F \to aF, \quad F \to bF, \quad S_0 \to S, \quad S \to a, \quad F \to a, \quad F \to b.$$

Step 1. We eliminate the rule $S_0 \to S$ by adding the rules $S_0 \to aF$, $S_0 \to bS$, and $S_0 \to a$:

$$S \to aF, \quad S \to bS, \quad F \to aF, \quad F \to bS, \quad S \to a, \quad F \to a, \quad F \to b,$$

$$S_0 \to aF, \quad S_0 \to bS, \quad S_0 \to a.$$

Step 2. We introduce two new variables V_a and V_b, replace a and b in length-2 right-hand sides with V_a and V_b, and add rules $V_a \to a$ and $V_p \to p$:

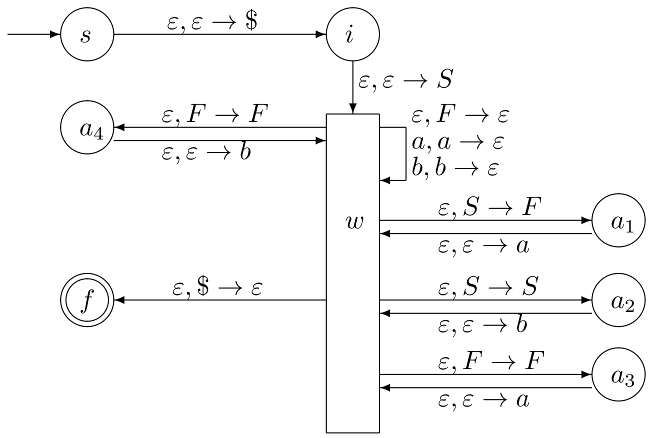
$$S \to V_aF, \quad S \to V_bS, \quad F \to V_aF, \quad F \to V_bS, \quad S \to a, \quad F \to a, \quad F \to b,$$

$$S_0 \to V_aF, \quad S_0 \to V_bS, \quad S_0 \to a, \quad V_a \to a, \quad V_b \to b.$$

This is already Chomsky normal form.

Problem 2e. Use a general algorithm to translate the CFG from 2b into an appropriate push-down automaton. Explain, step-by-step, how this automaton will accept the word *abab*.

Solution.



The word *abab* is derived as follows:

$$\underline{S} \to a\underline{F} \to ab\underline{F} \to aba\underline{F} \to abab\underline{F} \to abab.$$

So, we have the following acceptance by the pushdown automaton:

					a			b			a			b		
s	i	w	a_1	w	w	a_4	w	w	a_1	w	w	a_4	w	w	w	f
	$\$$	S	F	a	F	F	b	F	F	a	F	F	b	F	$\$$	
		$\$$	$\$$	F	$\$$	$\$$	F	$\$$	$\$$	S	$\$$	$\$$	F	$\$$		
				$\$$			$\$$			$\$$			$\$$			

Problem 2f. Use the general stack-based algorithms to show:

- how the compiler will transform a Java expression $2 + 2/7$ into inverse Polish (postfix) notation, and
- how it will compute the value of this expression.

Solution. Let us show, step by step, how the above expression is transformed into the postfix form:

$$\begin{array}{ccccccc}
2 & + & 2 & / & 7 & & \\
\hline
2 & & 2 & & 7 & / & + \\
\hline
 & + & + & / & / & + & \\
 & & & + & + & &
\end{array}$$

Let us now show how this expression will be computed:

$$\begin{array}{ccccc}
2 & 2 & 7 & / & + \\
\hline
2 & 2 & 7 & 0 & 2 \\
 & 2 & 2 & 2 & \\
 & & 2 & &
\end{array}$$

Problem 3. *Beyond pushdown automata: Turing machines*

Problem 3a. Prove that the language of ideal sequences as described in Problem 2a is not *context-free* and therefore, cannot be recognized by a pushdown automaton.

Solution: Proof by contradiction. Let us assume that this language is context-free. Then, by the pumping lemma for context-free grammars, there exists an integer p such that every word w from this language whose length is at least p can be represented as $w = uvxyz$, where $\text{len}(vy) > 0$, $\text{len}(vxy) \le p$, and for every i, we have $uv^ixy^iz \in L$.

Let us take the word $w = \text{A}^p\text{B}^{2p}\text{C}^p \in L$. The length of this word is $3p > p$, so this word can be represented as $w = uvxyz$.

Where is vxy? Since the length of this part does not exceed p, this word cannot contain As, Bs, and Cs – otherwise, it will have to contain all the B symbols – there are p of these symbols – and also at least one A and at least one C, so vxy will have a length at least $p + 2$, but its length is $\le p$. So, we have the following possible cases:

- vxy is in As;
- vxy is between As and Bs;
- vxy is in Bs;
- vxy is between Bs and Cs; or
- vxy is in Cs.

In the first case, v and y contain only As. So, when we go from $uvxyz$ to $uvvxyyz$, we add As, but we do not add Bs; thus, the desired balance between numbers of As and Bs is disrupted, and so $uvvxyyz \notin L$ – while by pumping lemma, we should have $uvvxyyz \in L$. Thus, this case is impossible.

In the second case, v and y contain only As and Bs. So, when we go from $uvxyz$ to $uvvxyyz$, we add As and Bs, but we do not add any Cs; thus, the desired balance between numbers of As, Bs, and Cs is disrupted, and so $uvvxyyz \notin L$ – while by pumping lemma, we should have $uvvxyyz \in L$. Thus, this case is impossible.

Similarly, we can prove that the other cases are also not possible. So, none of the cases is possible, which means that our assumption that the language L is context-free is wrong.

Problem 3b-c. Use a general algorithm to design a Turing machine that accepts exactly all sequences accepted by a finite automaton from Problem 1a. Show, step-by-step, how this Turing machine will accept the word *abab*. Describe, for each step, how the state of the tape can be represented in terms of states of two stacks.

Solution: This Turing machine will have the following rules:

- start, $-\to s$, R
- s, $a \to f$, R
- s, $b \to s$, R
- f, $a \to f$, R
- f, $b \to f$, R
- s, $-\to$ reject
- f, $-\to$ accept

Tracing:

Moment 1:

$\underline{-}$	a	b	a	b	$-$	$\dots$	start

Here, the left stack is empty, the right stack has the following form:

$-$
a
b
a
b

Moment 2:

$-$	$\underline{a}$	b	a	b	$-$	$\dots$	s

Here, the stacks have the following form:

	a
$-$	b
	a
	b

Moment 3:

$$\begin{array}{|c|c|c|c|c|c|c} \hline - & a & \underline{b} & a & b & - & \dots \\ \hline \end{array} \quad f$$

Here, the stacks have the following form:

$$\begin{array}{|c|} \hline a \\ \hline - \\ \hline \end{array} \quad \begin{array}{|c|} \hline b \\ \hline a \\ \hline b \\ \hline \end{array}$$

Moment 4:

$$\begin{array}{|c|c|c|c|c|c|c} \hline - & a & b & \underline{a} & b & - & \dots \\ \hline \end{array} \quad f$$

Here, the stacks have the following form:

$$\begin{array}{|c|} \hline b \\ \hline a \\ \hline - \\ \hline \end{array} \quad \begin{array}{|c|} \hline a \\ \hline b \\ \hline \end{array}$$

Moment 5:

$$\begin{array}{|c|c|c|c|c|c|c} \hline - & a & b & a & \underline{b} & - & \dots \\ \hline \end{array} \quad f$$

Here, the stacks have the following form:

$$\begin{array}{|c|} \hline a \\ \hline b \\ \hline a \\ \hline - \\ \hline \end{array} \quad \begin{array}{|c|} \hline b \\ \hline \end{array}$$

Moment 6:

$$\begin{array}{|c|c|c|c|c|c|c} \hline - & a & b & a & b & \underline{-} & \dots \\ \hline \end{array} \quad f$$

Here, the stacks have the following form:

$$\begin{array}{|c|} \hline b \\ \hline a \\ \hline b \\ \hline a \\ \hline - \\ \hline \end{array} \quad \begin{array}{|c|} \hline - \\ \hline \end{array}$$

Moment 7:

$$\begin{array}{|c|c|c|c|c|c|c} \hline - & a & b & a & b & \underline{-} & \dots \\ \hline \end{array} \quad \text{accept}$$

The machine halts, the stack remain the same.

Problem 3d-e. Design Turing machines for computing $a - 4$ in unary and in binary codes. Trace both Turing machines for $a = 5$.

Solution: unary case. The rules are:

$$
\begin{gathered}
\text{start}, - \to \text{R}, \text{moving} \\
\text{moving}, 1 \to \text{R} \\
\text{moving}, - \to \text{L}, \text{erasing1} \\
\text{erasing1}, 1 \to -, \text{L}, \text{erasing2} \\
\text{erasing2}, 1 \to -, \text{L}, \text{erasing3} \\
\text{erasing3}, 1 \to -, \text{L}, \text{erasing4} \\
\text{erasing4}, 1 \to -, \text{L}, \text{back} \\
\text{back}, 1 \to \text{L} \\
\text{back}, - \to \text{halt}
\end{gathered}
$$

Tracing:

$\underline{-}$	1	1	1	1	1	−	...	start
−	$\underline{1}$	1	1	1	1	−	...	moving
−	1	$\underline{1}$	1	1	1	−	...	moving
−	1	1	$\underline{1}$	1	1	−	...	moving
−	1	1	1	$\underline{1}$	1	−	...	moving
−	1	1	1	1	$\underline{1}$	−	...	moving
−	1	1	1	1	1	$\underline{-}$	...	moving
−	1	1	1	1	$\underline{1}$	−	...	erasing1
−	1	1	1	$\underline{1}$	−	−	...	erasing2
−	1	1	$\underline{1}$	−	−	−	...	erasing3
−	1	$\underline{1}$	−	−	−	−	...	erasing4
−	$\underline{1}$	−	-	−	−	−	...	back
$\underline{-}$	1	−	-	−	−	−	...	back
$\underline{-}$	1	−	-	−	−	−	...	halt

Solution: binary case. The rules are:

$$
\begin{gathered}
\text{start}, - \to \text{R}, \text{skip1} \\
\text{skip1}, 0 \to \text{R}, \text{skip2} \\
\text{skip1}, 1 \to \text{R}, \text{skip2} \\
\text{skip2}, 0 \to \text{R}, \text{moving} \\
\text{skip2}, 1 \to \text{R}, \text{moving} \\
\text{moving}, 0 \to 1, \text{R}
\end{gathered}
$$

moving, $1 \to 0$, L, back
back, $0 \to$ L
back, $1 \to$ L
back, $- \to$ halt

Here is tracing:

$\underline{-}$	1	0	1	$-$	$-$	$-$	...	start
$-$	$\underline{1}$	0	1	$-$	$-$	$-$	...	skip1
$-$	1	$\underline{0}$	1	$-$	$-$	$-$	...	skip2
$-$	1	0	$\underline{1}$	$-$	$-$	$-$	...	moving
$-$	1	$\underline{0}$	0	$-$	$-$	$-$	...	back
$-$	$\underline{1}$	0	0	$-$	$-$	$-$	...	back
$\underline{-}$	1	0	0	$-$	$-$	$-$	...	back
$\underline{-}$	1	0	0	$-$	$-$	$-$	...	halt

Problem 4. *Beyond Turing machines: computability*

Problem 4a. Formulate Church-Turing thesis. Is it a mathematical theorem? Is it a statement about the physical world?

Solution: Church-Turing thesis states that any function that can be computed on any physical device can also be computed by a Turing machine (or, equivalently, by a Java program).

Whether this statement is true or not depends on the properties of the physical world. Thus, this statement is not a mathematical theorem, it is a statement about the physical world.

Problem 4b. Prove that the halting problem is not algorithmically solvable.

Solution: The halting problem is the problem of checking whether a given program p halts on given data d. We can prove that it is not possible to have an algorithm haltChecker(p,d) that always solves this program by contradiction. Indeed, suppose that such an algorithm – i.e., such a Java program – exists. Then, we can build the following auxiliary Java program:

```
public static int aux(String x)
{if(haltChecker(x,x))
  (while(true) x= x;}
else{return 0;}}
```

If aux halts on aux, then haltChecker(aux,aux) is true, so the program aux goes into an infinite loop – and never halts. On the other hand, if aux does not halt on aux, then haltChecker(aux,aux) is false, so the program aux returns 0 – and thus, halts. In both cases, we get a contradiction, which proves that haltChecker is not possible.

Problem 4c. Not all algorithms are feasible, but, unfortunately, we do not have a perfect definition is feasibility. Give a current formal definition of feasibility, explain what is means to be practically feasible, and give two examples:

- an example of an algorithm's running time which is feasible according to the current definition but not practically feasible, and
- an example of an algorithm's running time which is practically feasible but not feasible according to the current definition.

Solution: An algorithm A is called feasible if its running time $t_A(x)$ on each input x is bounded by some polynomial $P(\text{len}(x))$ of the length $\text{len}(x)$ of the input: $t_A(x) \le P(\text{len}(x))$. In other words, the algorithm is feasible if for each length n, the worst-case complexity $t^w_A(n) = \max\{t_A(x) : \text{len}(x) = n\}$ is bounded by a polynomial: $t^w_A(n) \le P(n)$.

An algorithm is practically feasible if for all inputs of reasonable length, it finishes its computations in reasonable time.

Time complexity $t^w_A(n) = 10^{200}$ is a constant – thus a polynomial, so from the viewpoint of the formal definition, it is feasible. However, this number is larger than the number of particles in the Universe, so it is clearly not practically feasible.

On the other hand, the function $\exp(10^{-22}\cdot n)$ is an exponential function and thus, grows faster than an polynomial, but even for largest realistic lengths n – e.g., for $n = 10^{18}$ – the resulting value is smaller than 3 and is, thus, perfectly practically feasible.

Problem 4d. Briefly describe what is P, what is NP, what is NP-hard, and what is NP-complete. Is P equal to NP?

Solution: P is the class of all the problems that can be solved in polynomial time.

NP is the class of all the problems for which, once we have a candidate for a solution, we can check, in polynomial time, whether it is indeed a solution.

A problem is called NP-hard if every problem from the class NP can be reduced to this problem.

A problem is called NP-complete if it is NP-hard and itself belongs to the class NP.

At present, no one knows whether P is equal to NP. Most computer scientists believe that these two classes are different.

Problem 4e. Give an example of an NP-complete problem: what is given, and what we want to find.

Solution. An example of an NP-complete problem is propositional satisfiability:

- given: a propositional formula, i.e., any expression obtained from Boolean variables by using "and", "or", and "not",
- find: the values of the Boolean variables that makes this formula true.

Problem 4f. Give definitions of a recursive (decidable) language and of a recursively enumerable (Turing-recognizable) language.

Solution. A language L is called *decidable* if there exists an algorithm (or, equivalently, a Turing machine) that:

- given a word,
- returns "yes" or "no" depending on whether this word belongs to this language or not.

A language is called *semi-decidable*, *Turing-recognizable*, or *recursively enumerable* if there exists a Turing machine that:

- given a word w,
- returns "yes" if and only if the word w belongs to the language L.