Automata, Computability, and Formal Languages
Fall 2023, Solutions to Test 3

1-2. A perfect grading is when half of the students get Bs, one quarter As, and one quarter Cs. Let L be the language of all the sequences of letters A, B, and C that correspond to perfect grading. For example, $BABC \in L$ but $ABC \notin L$. Prove that this language is not context-free.

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 $uv^i xy^i z \in L$.

Let us take the word $w = A^p B^{2p} C^p \in L$, in which first we have A repeated p times, then B repeated $2p$ times, and then C repeated p times. The length of this word is $p + 2p + p = 4p > p$, 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 three different digits, i.e., A's, B's, and C's – otherwise, it will have to contain all the B's between A's and C'ss – there are $2p$ of these symbols, and also at least one of A's and one of C's, to the total of more than p. So, we have the following possible cases:

- vxy is in A's;
- vxy is between A's and B's;
- vxy is in B's;
- vxy is between B's and C's;
- vxy is in C's.

In the first case, v and y contain only A's. So, when we go from $uvxyz$ to $uvvxyyz$, we add A's, but we do not add B's or C's; thus, the desired balance between numbers of A's, B's, and C's 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 A's and B's. So, when we go from $uvxyz$ to $uvvxyyz$, we add A's and B's, but we do not add any C's; thus, the desired balance between numbers of A's, B's, and C's 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.

3. The following Turing machine deletes a binary number:

- start, – → moving, R (here, – means blank)
- moving, 0 → R
- moving, 1 → R
- moving, – → deleting, L
- deleting, 0 → –, L
- deleting, 1 → –, L
- deleting, – → halt

Trace it on the example of the word 10. Explain how each step will be represented if we interpret the Turing machine as a finite automaton with two stacks.

Solution:

Moment 1:

$\underline{-}$	0	1	–	...

start

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

–
0
1

Moment 2:

–	$\underline{0}$	1	–	...

moving

Here, the stacks have the following form:

–

0
1

Moment 3:

–	0	$\underline{1}$	–	...

moving

Here, the stacks have the following form:

0
–

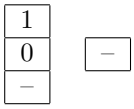
1

Moment 4:

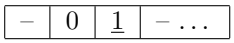
–	0	1	$\underline{-}$	...

moving

Here, the stacks have the following form:

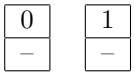


Moment 5:



deleting

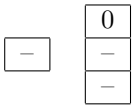
Here, the stacks have the following form:



Moment 6:

deleting

Here, the stacks have the following form:



Moment 7:

deleting

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

Moment 8:

halt

Here, the stacks are the same as in Moment 7.

4. Arithmetic operations on Turing machines:

   a Design a Turing machine that subtracts 2 from a binary number.

   b Trace your Turing machine, step-by-step, on the example of the number 3.

   c Why in Turing machines (and in most actual computers) the representation of a binary number starts with the least significant digit?

Solution: The idea is to skip the first bit and then to use the algorithm for subtracting 1 from a binary number. Here are the rules:

start, – → skip, R
skip, 1 → move, R
skip, 0 → move, R
move, 0 → 1, R
move, 1 → 0, L, back
back, 0 → L
back, 1 → L
back, – → halt.

Here is the tracing:

<u>–</u>	1	1	–	–	–	–	...	start
–	<u>1</u>	1	–	–	–	–	...	skip
–	1	<u>1</u>	–	–	–	–	...	move
–	<u>1</u>	0	–	–	–	–	...	back
<u>–</u>	1	0	–	–	–	–	...	back
<u>–</u>	1	0	–	–	–	–	...	halt

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.

The same representation is used for Turing machines, to make them closer to how actual computers work and thus, make them more realistic.

5. The following finite automaton describes binary strings that start with 1:
   - the starting state s;
   - the final state f meaning that the first symbol was 1; and
   - the error state e meaning that the first symbol was not 1.

   Transitions are as follows:
   - from the state s, symbol 1 leads to the state f and symbol 0 leads to the state e;
   - from the state f, each symbol leads back to f;
   - from the state e, each symbol leads back to e.

   Use the general algorithm to transform this finite automaton into a Turing machine. Show, step-by-step, how your Turing machine will accept the string 10.

Solution: This Turing machine will have the following rules:

- start, $- \rightarrow$ s, R
- s, $1 \rightarrow$ f, R
- s, $0 \rightarrow$ e, R
- f, $0 \rightarrow$ f, R
- f, $1 \rightarrow$ f, R
- e, $0 \rightarrow$ e, R
- e, $1 \rightarrow$ e, R
- s, $- \rightarrow$ reject
- e, $- \rightarrow$ reject
- f, $- \rightarrow$ accept

Tracing:

$\underline{-}$	1	0	$-$	$-$	...	start
$-$	$\underline{1}$	0	$-$	$-$	...	s
$-$	1	$\underline{0}$	$-$	$-$	...	f
$-$	1	0	$\underline{-}$	$-$	...	f
$-$	1	0	$\underline{-}$	$-$	...	accept

6. Give the formal definition of a feasible algorithm, and an explanation of what practically feasible means. Give two examples different from what we had in class:

- an example of a computation time which is formally feasible, but not practically feasible, and
- an example of a computation time which is practically feasible but not formally feasible.

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 called practically feasible if for every input of reasonable length, it finished its computations in reasonable time.

Time complexity $t^w_A(n) = 10^{1923}$ 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 $\exp(10^{-1923} \cdot n)$ is an exponential function and thus, grows faster than an polynomial, but even for largest realistic lengths n – e.g., for $n = 10^{22}$ – the resulting value is smaller than 3 and is, thus, perfectly practically feasible.

7. What is P? What is NP? What does it means for a problem to be NP-hard? NP-complete? Give brief definitions. Give an example of an NP-complete problem: explain what is the input, what is the desired output. 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.

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.

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