Solutions to Test 2

Problem 1. Why do we need to study decidable and recursively enumerable (r.e.) sets?

Solution. Until we started studying decidable and r.e. sets, we have dealt with algorithms and programs, objects with which we are very familiar and for which we have good intuition.

To solve some problems, however, this intuition is not well suited, so we need to reformulate the problem in precise terms – i.e., in terms of mathematics. We have already done this for for-loops and for while-loops: to analyze what can be computed by using the corresponding loops. It is reasonable to expect that similar reformulation may help in other cases as well. Let us think of it in the most general terms: how can we reformulate computational concepts in mathematical terms?

In modern mathematics, the most basic notion is the notion of a *set*. All other notions are formulated in terms of sets. Sets is what all the students study in high school – as well as natural set operations such as union, intersection, and complement. Sets is what computer science students study again in a Discrete Math (or Discrete Structures) course. So, a natural idea is to reformulate the notions related to computability in terms of sets – so that, in addition to computational intuition, we will be able to use our intuition about sets, about their unions, intersections, and complements.

Problem 2. Is the intersection of two r.e. sets always r.e.? If yes, prove it, if no, provide a counterexample.

Solution. We will show that intersection $A \cap B$ of r.e. sets A and B is always r.e.

The fact that the set A is r.e. means that there exists an algorithm a that eventually prints all the elements of the set A. Similarly, the fact that the set B is r.e. means that there exists an algorithm b that eventually prints all the elements of the set B. Let us describe a new algorithm that prints all the elements that belong to both sets – i.e., all the elements of the intersection. This algorithm is as follows:

- first, we run algorithm a for 1 hour;
- then, we run algorithm b for 1 hour;
- then, we compare the two lists printed-so-far by these two algorithms, and print all common elements of these two lists;
- then, we resume the run of the algorithm a and run it for 1 more hour;
- then, we resume the run of the algorithm b and run it for 1 more hour;
- then, we compare the two lists printed-so-far by these two algorithms, and print all common elements of these two lists that have not been printed before,
- etc.

Clearly, all printed elements belong to both sets. Let us show that every natural number n that belongs to both sets will thus be printed. Indeed, since this number belongs to the set A, it will be printed by the algorithm a at some time k. Similarly, since the number n belongs to the set A, it will be printed by the algorithm a at some time k.

Here, the k-th hour of the algorithm a is performed by the new algorithm in its $(2k-1)$-st hour, and the ℓ-th hour of the algorithm b is performed by the new algorithm in its (2ℓ)-th hour. So, by the time $\max(2k-1, 2\ell)$, the number n will be generated by both algorithms, and thus, will be printed by the new algorithm.

The statement is proven.

Problem 3. Is the set $(A \cup B) - C$, where $X - Y$ is $\{x : x \text{ is in } X \text{ and } x \text{ is not in } Y\}$, and A, B, and C are r.e., always r.e.? If yes, prove it, if no, provide a counterexample.

Solution. The only example of a non-r.e. set that we know is the complement $-H = N - H$, where N is the set of all natural numbers, and H is the halting set. This set is not r.e., since H is, and if $-H$ was r.e., we would be able to conclude that H is decidable, but we know that this set is not decidable.

So, if we take, e.g., $A = B = N$ and $C = H$, then all these three sets are r.e., but the set $(A \cup B) - C = N - H$ is not r.e. This example shows that the set $(A \cup B) - C$ is not always r.e.

Problem 4. Prove that it is not possible, given a program that always halts, to check whether this program always computes $5n + 6$.

Solution. We will prove that if such a checker exists, then we can construct a zero-checker – and we already know that zero-checkers are not possible. Indeed, let us assume that we have an algorithm $checker(p)$ that, given a program p that always halts, checked whether $\forall n\,(p(n) = 5n + 6)$. Suppose that we have a program q that always halts and we want to check whether this program q always returns 0. To check this, we form the following auxiliary program that always returns $q(n) + 5n + 6$:

```
public static int aux(int n)
  {return q(n) + 5 * n + 6;}
```

The value $q(n) + 5n + 6$ is always equal to $5n + 6$ if and only if the value $q(n)$ is always equal to 0.

Thus, the algorithm $checker(q(n)+5n+6)$ that applies *checker* to the above auxiliary program is a zero-checker. However, we have proven that zero-checkers do not exist. This contradiction shows that our assumption – that the desired checkers are possible – leads to a contradiction. Thus, such checkers are not possible. The result is proven.

Problem 5. Design a Turing machine that computes $n + 4$ in binary code. Trace this machine on the example of $n = 101_2$.

Solution. When we add $4_{10} = 100_2$ to a binary number, the two bits of 100_2 are 0s, so the last two bit of the sum does not change, but for other bits, we have the same algorithm as for computing $n + 1$. Here is the resulting algorithm:

- we skip the last two bits,
- after that, if we see 1, we replace 1 with 0;
- if we see 0 or blank, we replace them with 1 and start going back.

Here are the corresponding Turing machine rules:

- start, $- \to$ R, skip1st
- skip1st, $1 \to$ R, skip2nd
- skip1st, $0 \to$ R, skip2nd
- skip2nd, $1 \to$ R, moving
- skip2nd, $0 \to$ R, moving
- moving, $1 \to 0$, R
- moving, $0 \to 1$, L, back
- moving, $- \to 1$, L, back
- back, $0 \to$ L
- back, $1 \to$ L
- back, $- \to$ halt

Here is a tracing on the example of 101 + 100:

$\underline{-}$	1	0	1	−	−	...	start
−	$\underline{1}$	0	1	−	−	...	skip1st
−	1	$\underline{0}$	1	−	−	...	skip2nd
−	1	0	$\underline{1}$	−	−	...	moving
−	1	0	0	$\underline{-}$	−	...	moving
−	1	0	$\underline{0}$	1	−	...	back
−	1	$\underline{0}$	0	1	−	...	back
−	$\underline{1}$	0	0	1	−	...	back
$\underline{-}$	1	0	0	1	−	...	back
$\underline{-}$	1	0	0	1	−	...	halt

Problem 6. Use a general algorithm for a Turing machine that represents composition to transform your design from Problem 5 into a Turing machine for computing $f(f(n)) = n + 8$.

Solution.

- start, – $\to$ R, skip1st_1
- skip1st_1, 1 $\to$ R, skip2nd_1
- skip1st_1, 0 $\to$ R, skip2nd_1
- skip2nd_1, 1 $\to$ R, moving_1
- skip2nd_1, 0 $\to$ R, moving_1
- moving_1, 1 $\to$ 0, R
- moving_1, 0 $\to$ 1, L, back_1
- moving_1, – $\to$ 1, L, back_1
- back_1, 0 $\to$ L
- back_1, 1 $\to$ L
- back_1, – $\to$ start_2
- start_2, – $\to$ R, skip1st_2
- skip1st_2, 1 $\to$ R, skip2nd_2
- skip1st_2, 0 $\to$ R, skip2nd_2
- skip2nd_2, 1 $\to$ R, moving_2
- skip2nd_2, 0 $\to$ R, moving_2
- moving_2, 1 $\to$ 0, R
- moving_2, 0 $\to$ 1, L, back_2
- moving_2, – $\to$ 1, L, back_2
- back_2, 0 $\to$ L
- back_2, 1 $\to$ L
- back_2, – $\to$ halt

Problem 7. Give a formal definition of feasibility. Give two examples:

- an example when an algorithm is feasible in the sense of the formal definition but not practically feasible, and
- an example when an algorithm is practically feasible, but not feasible according to the formal definition.

These examples must be different from the examples that we had in class.

Solution. An algorithm A is feasible if there exists a polynomial $P(n)$ such that for each input x of size $\text{len}(x) = n$, the computation time $t_A(x)$ is smaller than or equal to $P(n)$:

$$t_A(x) \leq P(\text{len}(x)).$$

Examples:

- an example when an algorithm is formally feasible, but not practically feasible: $t_A^w(n) = 10^{300}$;
- an example when an algorithm is practically feasible but not formally feasible: $t_A^w(n) = \exp(10^{-100} \cdot n)$.

Problem 8. What is P? NP? NP-hard? NP-complete? Brief definitions are OK. What do we gain and what do we lose when we prove that a problem is NP-complete? Explain one negative consequence (what we cannot do) and one positive one (what we can do).

Solution.

- P is the class of all the problems that can be solved in polynomial (= feasible) time.

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

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

- A problem is called NP-hard if every problem from the class NP can be reduced to this problem. *Comment:* the difference from NP-completeness is that an NP-hard problem may not be from the class NP.

What do we gain and what do we lose when we prove that a problem is NP-complete? A positive consequence is that if we have a good algorithm for solving some cases of the problem, then we automatically get good algorithms for all other problems from the class NP – and many good algorithms have been obtained this way. A negative consequence is that, unless it turns out that P = NP, we cannot have a feasible algorithm for solving all particular cases of this problem.

Problem 9. What is propositional satisfiability? Give an example. Explain why this problem is important in software testing.

Solution. Propositional satisfiability:

- *given:* a propositional formula, i.e., any expression obtained from Boolean variables by using "and" (&& in Java), "or" (|| in Java), and "not" (! in Java) – e.g., $!(a \mid\mid !b)$ && $(!a \mid\mid b)$;
- *find:* the values of the Boolean variables that make the given formula true.

Why is this problem interesting? Because when we test a program with branching, we need to make sure that we have tested both branches. For this purpose, we need to find the values of the variables for which the corresponding condition is true. This is exactly what propositional satisfiability is about.

Problem 10. Step-by-step, apply the general algorithm to translate the following formula into DNF and CNF: $0.5 \cdot a \geq b$.

Solution. Let us describe the truth values of this formula F for all possible combinations of values a and b.

a	b	F	$\neg F$
0	0	1	0
0	1	0	1
1	0	1	0
1	1	0	1

The DNF form describes that the formula is true if we are in one the rows for which $F = 1$. So, the DNF form is as follows:

$$(\neg a \,\&\, \neg b) \vee (a \,\&\, \neg b).$$

To get the CNF form, we first need to write down the DNF form for the negation $\neg F$:

$$(\neg a \,\&\, b) \vee (a \,\&\, b).$$

The CNF form is the negation of the DNF form for $\neg F$, obtained by using de Morgan laws:

$$(a \vee \neg b) \,\&\, (\neg a \vee \neg b).$$