Algorithms and Complexity. Exercise session 7

Undecidability

Undecidability 1 Is there an explicit program P so that for a given y it is decidable whether P terminates on input y?

Solution to Undecidability 1
Yes, there are plenty of such programs. Take for example the simple program $P(y) = \textbf{return}$. It is easy to see that P terminates on all input. □

Undecidability 2 Is there an explicit program P so that for a given y it is undecidable whether P terminates on input y?

Solution to Undecidability 2
Yes. Let P be an interpreter, and the input y be a program x followed by input y' to that program. This means that $P(y)$ behaves in the same way as the program x on input y, and it is undecidable whether a program x terminates on a specific input y. □

Undecidability 3 If the program x terminates on empty input, we denote by $f(x)$ the number of steps before terminating. Otherwise, we set $f(x) = 0$. Define now $MR(y)$, the maximum running time of all programs whose binary encoding is less than y.

$$MR(y) = \max_{x \leq y} f(x),$$

Is MR computable?

Solution to Undecidability 3
No, MR is not computable. We know it's undecidable whether a program terminates on the empty input (empty string). Therefore reduce this problem to MR. Note that $MR(y)$ is the maximum number of steps that each program of "size" at most y need before terminating, if it starts with an empty input.

StopBlank(y) =
  $s \leftarrow MR(y)$
  if $s = 0$ **then return false**
  else
    simulate y on empty input for s steps (or until y terminates)
    if y terminates **then return true**
    else return false

If y doesn't terminate within s steps, we know that it will never terminate. Since StoppBlank is not computable, MR can not be computable either. □

Undecidability 4 Show that the function MR in the previous exercise grows faster than any recursive function. In particular, show that for every recursive function g there is a y such that $MR(y) > g(y)$.

Solution to Undecidability 4
Suppose the opposite, namely that there is a recursive function g such that $g(y) \geq MR(y)$ for all y. Then we could replace the first instruction $s \leftarrow MR(y)$ in the solution to the previous exercise with the instruction $s \leftarrow g(y)$ and the program will still work. In addition, the program will be computable, but we know it can not be so. Therefore, we have a contradiction and our assumption is false. □

Undecidability 5 Suppose you are given a Turing machine M, an input X (which is on the tape from the beginning) and an integer constant K. Is the following problem decidable or not?

Does M terminate on input X using at most K cells of the tape (a used cell may written and read several times)?

Solution to Undecidability 5
The problem is decidable. Since the Turing machine must remain within the K cells on the tape, there are only a finite number of configurations of the Turing machine can be in. If the machine has m states, the number of configurations is $m \cdot K \cdot 3^K$ (the number of states times the number of possible positions of the head for reading and writing times the number of possible tape configurations). Simulate a Turing machine in $m \cdot K \cdot 3^K + 1$ steps and check all the time that it does not move beyond the K cells on the tape. If the Turing machine stops within this time, we answer *yes*. If the Turing machine does not stop after this time, it must returned to a configuration it has previously been in, therefore it is inside an infinite loop and we can safely say *no*. □

Construct knapsack Knapsack problem is a well-known NP-complete problem. The input is a set P of objects with weight w_i and the value v_i, the knapsack size S and a target K. The question is to determine if it is possible to pick a subset of objects of value at least K, such that their total weight does not exceed S. All numbers in input are non negative integers.

Now assume that we have an algorithm A that solves the above decision problem. Construct an algorithm that uses A to solve the constructive knapsack problem, ie. given the same input as A it answers the knapsack problem and secondly, it tells you exactly which objects to pack into the knapsack. The algorithm may call A $O(|P|)$ times, so it can not take more than polynomial time.

Then construct and analyze a Turing reduction of the constructive knapsack problem to the usual knapsack problem (which is a decision problem).

Solution to Construct knapsack

```
Knapsack(P,S,K)=
  if not A(P,S,K) then return "No solution"
  foreach p in P do
    if A(P-p,S,K) then P:=P-{p}
  return P
```

The algorithm calls A at most $|P|+1$ times. The returned amount of P is a solution that meets the following two criteria.

- *P contains no unnecessary elements* since the for loop runs over all elements in P, and in each iteration we check if an element p is redundant; if this is the case, we remove it from P.
- *P contains a solution* because this is an invariant of the loop. If you enter the loop, this is satisfied, and it holds for each iteration in the loop.

$\square$
