Design and Analysis of Algorithms, Fall 2014
Exercise III: Solutions

III-1 Show that the subset-sum problem is solvable in polynomial time if the target value t is expressed in unary.

Let $S_i, \ldots, S_n$ be the sequence of positive integers. We want to find out if there is a (multi-)subset of S that sums up to $t > 0$. For all $0 \leq i \leq n$ and $0 \leq p \leq t$, we define $L[i,p]$ to be true if there is a subset of $S_1, \ldots, S_i$ that sums up p, and false otherwise. Then $L[n,t]$ is the solution to the problem. Observe that $L[i,0]$ is true for all, i and $L[0,p]$ is false for all $p > 0$. For $i \geq 1$ and $p \geq 1$, there is a subset of $S_1, \ldots, S_i$ summing up to p if and only if there is a subset of $S_1, \ldots, S_{i-1}$ summing to either p or $p - S_i$. Thus, we get

$$L[i,p] = \begin{cases} \text{true} & \text{if } p = 0 \\ \text{false} & \text{if } p > 0 \text{ and } i = 0 \\ L[i-1,p] \vee L[i-1,p-S_i] & \text{otherwise.} \end{cases}$$

The term $L[n,t]$ is easily evaluated in $O(nt)$ time using dynamic programming. Let b be the size of the input in bits. If t is expressed in binary, as is common, then $b = O(\log_2 t)$ and the running time is $O(2^b n)$. However, if we express t in unary, then $b = O(t)$ and the running time is $O(nb)$, which is polynomial in the input b.

III-2 (**CLRS 34.1-5**) Show that if an algorithm makes at most a constant number of calls to polynomial-time subroutines and performs an additional amount of work that also takes polynomial time, then it runs in polynomial time. Also show that a polynomial number of calls to polynomial-time subroutines may result in an exponential-time algorithm.

Let k be the number of subroutines. The algorithm starts out with an input data of size n. Each subroutine takes (some of) the available data as an input, performs some steps, then returns some amount of data as an output. Every time a subroutine returns, the output accumulates the amount of data the algorithm has access to. Any or all of this data may then be given as an input to the next subroutine. Since each subroutine runs in polynomial time, the output must also have a size polynomial in the size of the input. Let d be an upper bound on the degree of the polynomials. Then there is a function $p(n) = n^d + c$, where c is a constant, such that $p(n)$ is an upper bound for the size of the output of any subroutine when given an input of size n.

Let $n_0 = n$ and $n_i = n_{i-1} + p(n_{i-1})$ for all $1 \leq i \leq k$. We show by induction that n_i is an upper bound for the amount of data available to the algorithm after the ith subroutine call. The base case is trivial. Assume the claim holds for $i-1$ and let n' be the exact amount of data available before the ith call. Then we have $n_i \leq n' + p(n')$, since the ith call accumulates the amount of the data by at most $p(n')$. Since p is increasing, by assumption we have $n' \leq n_{i-1}$ and $p(n') \leq p(n_{i-1})$, from which the claim follows.

We use induction again to show that each n_i is polynomial in n. The base case is again trivial. Assume n_{i-1} is polynomial in n. Since the composition of two polynomials is also polynomial, we have that $p(n_{i-1})$ is polynomial in n. Since also the sum of two polynomials is polynomial, we have that $n_{i-1} + p(n_{i-1}) = n_i$ is polynomial in n. Therefore n_k, which is an upper bound for the amount of data after the final subroutine, is also polynomial in n, and the time must also be polynomial.

For the second part, observe that if we have a subroutine whose output is always twice the size of its input, and we call this subroutine n times, starting with input of size 1 and always feeding the previous output back into the subroutine, the final output will have size 2^n. This means that the algorithm will take exponential time.

III-3 (**CLRS 34.5-8**) In the half 3-CNF satisfiability problem, we are given a 3-CNF formula ϕ with n variables and m clauses, where m is even. We wish to determine whether there exists a truth assignment to the variables of ϕ such that exactly half the clauses evaluate to 0 and exactly half the clauses evaluate to 1. Prove that the half 3-CNF satisfiability problem is NP-complete. (You may assume that the 3-CNF formula has *at most* 3 literals per clause, not necessarily exactly 3.)

First observe that given a 3-CNF formula and an assignment, it is easy to check in polynomial time if the assignment satisfies exactly half of the clauses. Therefore the half 3-CNF satisfiability is in NP.

We show NP-completeness by reduction from 3-CNF satisfiability. Let φ be a 3-CNF-SAT formula with n variables and m clauses. We construct a 3-CNF-SAT formula ψ such that exactly half of the clauses in ψ can be satisfied if and only if φ can be satisfied. Suppose that y_i and z_i for $i = 1, \ldots, m+1$ as well as p are variables that do not appear in φ. We add to ψ by all the clauses of φ, $m(m+1)+1$ distinct clauses of form $\{q, \neg q, p\}$, where q can be any variable (we call these *type 1 clauses*) and clause $\{y_i, z_j, p\}$ for all $i, j \in \{1, \ldots, m+1\}$ (we call these *type 2 clauses*). Constructing ψ clearly takes polynomial time.

We observe that all type 1 clauses are always satisfied. Since there are total of $2(m+1)^2$ clauses, we have to satisfy precisely m other clauses to satisfy half of the clauses of ψ. If we tried to satisfy any type 2 clause $\{y_i, z_j, p\}$, we would also satisfy all type 2 clauses with variable y_i or all type 2 clauses with variable z_j. This means that we would satisfy at least $m+1$ additional clauses, that is, total of at least $(m+1)^2+1$ clauses. Thus the only way to satisfy exactly $(m+1)^2$ clauses in ψ is to satisfy all the clauses of φ. This implies that exactly half of the clauses in ψ can be satisfied if and only if φ can be satisfied.

Thus, given a polynomial-time algorithm for the half 3-CNF-SAT problem, we could solve 3-CNF-SAT in polynomial time. Since we know 3-CNF-SAT to be NP-complete, it follows that the half 3-CNF-SAT is NP-complete as well.

III-4 (**CLRS 34.4-6**) Suppose someone gives you a polynomial-time algorithm to decide formula satisfiability. Describe how to use this algorithm to find satisfying assignments in polynomial time.

Let φ be the input SAT formula that is satisfiable and contains n variables. Let $\varphi|_{x_i=0}$ and $\varphi|_{x_i=1}$ be the simplified SAT formulas obtained by replacing variable x_i by values 0 and 1 respectively and eliminating constants 0 and 1 by partially evaluating the formula. These can be computed in polynomial time. The results are SAT formulas containing $n-1$ variables. For example, if $\varphi = ((x_1 \to x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$ then

$$\varphi|_{x_2=0} = ((x_1 \to 0) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg 0 = (\neg x_1 \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \text{ and}$$
$$\varphi|_{x_2=1} = ((x_1 \to 1) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg 1 = 0.$$

Clearly φ is satisfiable by assignment containing $x_i = c$ if and only if $\varphi|_{x_i=c}$ is satisfiable. The algorithm now takes any variable x_i and asks the oracle whether $\varphi|_{x_i=0}$ is satisfiable. If the answer is yes, then the algorithm sets $x_i = 0$ and recursively repeats the procedure with $\varphi|_{x_i=0}$. Otherwise $\varphi|_{x_i=1}$ must be satisfiable, so the algorithm sets $x_i = 1$ and recursively repeats the procedure with $\varphi|_{x_i=1}$. Once all variables have been assigned a value, this assignment satisfies the original SAT formula. The algorithm takes n polynomial time steps and thus works in polynomial time.

Instead of reducing the formula, one may alternatively augment it with conjunctions, producing formulas of form $\varphi \wedge x_i$ and $\varphi \wedge \neg x_i$. Again, one consults the oracle and recurses on a satisfiable formula until for each variable either the variable or its negation has been added, yielding a satisfying assignment.

III-5 (**CLRS 34.5-6**) Show that the hamiltonian-path problem is NP-complete. (You may assume that you know that HAM-CYCLE is NP-complete.)

Again, observe that given a sequence of vertices it is easy to check in polynomial time if the sequence is a hamiltonian path, and thus the problem is in NP.

We reduce from the hamiltonian cycle problem. Let $G = (V, E)$ be a graph. The reduction transforms graph G into G' as follows. We pick an arbitrary vertex $v \in V$ and add a new vertex v' that is connected to all the neighbors of v. We also add new vertices u and u' so that u is adjacent to v and u' is adjacent to v'. This reduction clearly takes a polynomial time.

To complete the proof, we have to show that G has a hamiltonian cycle if and only if G' has a hamiltonian path. Now if there is a hamiltonian cycle $(v, v_2, \ldots, v_n, v)$ in G, then $(u, v, v_2, \ldots, v_n, v', u')$ is a hamiltonian path in G'. On the other hand, if there is a hamiltonian path in G', its endpoints have to be u and u', because these have only on neighbor and thus cannot be in a middle of the path. Thus, the path has form $(u, v, v_2, \ldots, v_n, v', u')$ and we have that $(v, v_2, \ldots, v_n, v)$ is a hamiltonian cycle in G.