Math 363: Theory of Algorithms
Instructor: Sreekar M. Shastry
Solutions to the Final Examination
2012-04-26-Thu 0900-1200

- There are 7 problems. Each problem is worth 5 points. The maximum score is 35 points.

(1) Consider the following problems.

3-SAT
Independent Set, Set Packing
Vertex Cover, Set Cover
3-D matching, Graph 3-Coloring
Hamilton Cycle, Hamilton Path, Traveling Salesman
Subset Sum, Knapsack

(a) Give *precise* definitions for as many of these problems as you can.
(b) Prove two (and no more than two!) polynomial reductions among these problems.

Solution. In chapter 8 of KT.

(2) At a summer sports camp, for each sport offered, there should exist one counselor who has the skills to guide the students in that sport. The camp has received job applications from m potential counselors, each of whom is skilled in at least one of the sports offered. Moreover, for each of the n sports there is some applicant qualified in that sport.

Consider the following algorithmic problem: For a given number $k \leq m$, is it possible to hire at most k of the counselors and have at least one qualified in each of the n sports?

Show that this problem is NP-complete.

Solution. Let us recall the Set Cover problem: Given a set U of n elements, a collection $S_1, S_2, \ldots, S_m$ of subsets of U, and a number k, does there exist a collection of at most k of these sets whose union is equal to all of U?

It is shown on page 472 of KT that this problem is NP-complete.

To solve the problem at hand, it suffices to show that an arbitrary instance of Set Cover can be reduced to an instance of our sports camp problem. Given an instance of Set Cover, regard U as a set of sports, and the set S_i as the set of sports that the ith applicant is skilled in. Thus a black box for the sports camp problem solves Set Cover, as required.

(3) The Fibonacci numbers are defined by the following recurrence:

$$F_0 := 0, F_1 := 1, \text{ and } F_i := F_{i-1} + F_{i-2} \text{ for } i \geq 2.$$

We may directly convert this mathematical definition into a recursive function to compute the Fibonacci numbers.

```
1: procedure FIB(n)
2:     if n = 0 or n = 1 then
3:         return n
4:     end if
5:     if n ≥ 2 then
6:         return FIB(n − 1)+FIB(n − 2)
7:     end if
8: end procedure
```

It is a basic fact which you make take for granted that if we write $\varphi := (1 + \sqrt{5})/2$ and $\overline{\varphi} := (1 - \sqrt{5})/2$ then $F_i = (\varphi^i - \overline{\varphi}^i)/\sqrt{5}$.

(a) Show that with the above definition, FIB(n) has running time $O(\text{FIB}(n))$. Conclude that it requires exponential time.

(b) Using memoization, give a linear time algorithm to compute FIB(n).

Solution. (a) Let f_n be the number of calls to $\textsc{Fib}(\cdot)$ made in the course of the computation of $\textsc{Fib}(n)$. By this we mean the following: we initialize $f_n := 0$ and as we trace through the execution of $\textsc{Fib}(n)$, we increment f_n by one each time we make a call to $\textsc{Fib}(i)$ for some $i \leq n$.

We claim that $f_n = 2F_n + 1$. The proof proceeds by induction. For $n = 0, 1$ we have that $f_n = 2 \cdot 1 - 1 = 1$. Now suppose that $f_n = 2F_n + 1$. We compute

$$\begin{aligned} f_{n+1} &= f_n + f_{n+1} + 1 && (+1 \text{ is the initial call to } \textsc{Fib}(n+1)) \\ &= (2F_n - 1) + (2F_{n-1} - 1) + 1 && (\text{by induction}) \\ &= 2F_{n+1} + 1 \end{aligned}$$

as required.

The fact that this is exponential follows from the formula $F_i = (\varphi^i - \overline{\varphi}^i)/\sqrt{5}$.

(b) The following algorithm runs in time $O(n)$.

```
allocate an array memo[0 : N], where N is the maximum allowed input
initialization: memo[0] := 0, memo[1] := 1, memo[k] := 0 for k ≥ 2.
procedure F(n)
    if n = 0 or n = 1 or memo[n] ≠ 0 then
        return memo[n]
    end if
    memo[n] := F(n − 1)+F(n − 2)
    return memo[n]
end procedure
```

(4) There are two types of wrestlers, the Good Guys and the Bad Guys. Between any pair of wrestlers, they may or may not be a rivalry. Suppose that we have n professional wrestlers and we have a list of m pairs of wrestlers for which there are rivalries. Give an $O(n+m)$-time algorithm that determines whether it is possible to designate some of the wrestlers as Good Guys and the remainder as Bad Guys such that each rivalry is between a Good Guy and a Bad Guy. If it is possible to perform such a designation, your algorithm should produce it.

Solution. Examples of wrestlers are Macho Man Randy Savage, The Undertaker, The Iron Sheikh, and Hulk Hogan. There is a pair (Hulk Hogan, The Iron Sheikh) indicating a rivalry between the two wrestlers.

Let G be a graph with the wrestlers as nodes and the rivalries as edges. Perform as many breadth first traversal so as to visit all vertices. Choose a root node arbitrarily and declare that it corresponds to a Good Guy. Declare to be Good Guys all wrestlers whose distance from the root is an even integer. The Bad Guys are those at an odd distance.

Now, we must verify by looping over the edges that each edge goes between a Good Guy and a Bad Guy. If this is not the case, then we are guaranteed the existence of the following configuration of edges, where the bottom row consists nodes at distance d from the root and the top row at distance $d+1$ for some positive integer d (it is possible that $x = y$).

This picture implies that it is not possible to consistently assign Good Guys and Bad Guys compatibly with rivalries and we may return "Not possible." Another way to say the same thing is that Good Guys is the color blue and Bad Guys is the color red. This configuration implies that it is impossible to color both x, y with one color and both r, s with the other color.

If no edge goes between a Good Guy and a Bad Guy (i.e. the above loop went through without hitting an offending edge) we return the sets of Good Guys and Bad Guys that we constructed earlier.

The BFS takes $O(n+m)$, the assignment of Good and Bad takes $O(n)$, and the verification loop over edges takes $O(m)$. Thus the complexity is $O(m+n)$, as required.

(5) Consider the problem of making change for n cents using the fewest number of coins, where we are given a set of possible coin denominations.

(a) Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies (of denominations 25, 10, 5, and 1 cents respectively). Prove that your algorithm yields an optimal solution.

(b) Give an example of a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that a solution exists for every n.

Solution. We first investigate the structure of optimal solutions.

Let $\text{opt}(n)$ be the optimal solution for making change worth n cents, and suppose it uses k coins to solve the problem. Suppose that this solution uses a coin worth c. We claim that $\text{opt}(n)$ contains within it $\text{opt}(n-c)$, and that the latter uses $k-1$ coins to optimally make change for $n-c$ cents. To prove the claim, clearly there are at most $k-1$ coins in $\text{opt}(n-c)$ and if there were fewer than $k-1$, we could use this solution together with the coin of denomination c to produce a solution of $\text{opt}(n)$ which uses fewer than k coins.

(a) A greedy algorithm to make change using quarters, dimes, nickels, and pennies is as follows. For a positive real number x, write $[x]$ for the greatest integer smaller than x. We are given n and we must make change for it.

Use $q := [n/25]$ quarters. Put $n_q := n \mod 25$ (by this we mean the positive integer which is less than 25 which is congruent to $n_q \mod 25$).

Then use $d := [n_q/10]$ dimes. Put $n_d := n_q \mod 10$.

Then use $k := [n_d/5]$ nickels. Put $n_k := n_d \mod 5$.

Then use $p := n_k$ pennies.

Another, less efficient way to prescribe the same algorithm is as follows. We want to make change for n cents. If $n = 0$ then put $\text{opt}(0) := \varnothing$. If $n > 0$ then determine the largest coin whose value c is $\leq n$. Give one such coin and then recursively solve the subproblem of making change for $n-c$ cents.

To prove the correctness of this algorithm, it suffices by what we have already said to prove the following property, which we will refer to as P_n: There exists an optimal solution for making change for n cents contains a coin worth c cents where c is the largest coin value such that $c \leq n$. Suppose we are given an optimal solution. If it already contains a coin of denomination c, we are done. So suppose it does not contain such a coin; we will transform it into a solution on fewer coins, thus verifying P_n.

There are four cases.

(i) $1 \leq n < 5 \Rightarrow$ the solution consists of only pennies so P_n is verified.

(ii) $5 \leq n < 10 \Rightarrow$ the solution consists only of pennies since it has no nickel. Replace five pennies by a nickel to give a solution with fewer coins.

(iii) $10 \leq n < 25 \Rightarrow c = 10$. By assumption, there is no dime in the solution, so it contains only nickels and pennies. Some subset of nickels and pennies must add up to 10 cents (if there are two nickels we are done; if there is one nickel, use it and five pennies; if there are no nickels, use ten pennies). Thus we may replace this subset of nickels and pennies by a dime to give a solution with fewer coins.

(iv) $25 \leq n \Rightarrow c = 25$. By supposition, the optimal solution contains no quarter, only nickels, dimes, and pennies. If it contains three dimes, we can replace that configuration by one quarter and one nickel for a better solution. If it contains at most two dimes, then some subset of the dimes, nickels, and pennies adds up to 25 cents, and we can replace that subset by a quarter for a better solution.

Regarding the running time, the first description of the algorithm which uses the greatest integer function and modular arithmetic runs in $O(1)$ because we only have to do two computations for each of the four denominations $1, 5, 10, 25$. The recursive version of the algorithm runs in $O(k)$ times where k is the number of coins used in an optimal solution; since $k \leq n$ we obtain a running time of $O(n)$.

(b) Suppose that $n = 30$ and we only have denominations of $1, 10, 25$ i.e. pennies, dimes, and quarters but no nickels. The greedy solution gives one quarter and five pennies for six coins total. The optimal (non-greedy) solution consists of three dimes.

(6) Let us recall the Subset Sum problem. Given a set of natural numbers $\{w_1, w_2, \ldots, w_n\}$ and a target number W, is there a subset of $\{w_i\}$ that adds up precisely to W?

It is shown in chapter 8 of the text that this is an NP-complete problem.

On the other hand, it is shown in chapter 6 (using dynamic programming) that there is an algorithm with running time $O(nW)$ that solves this problem

Why does this not imply that P=NP?

Solution. This is explained on page 491 of KT.

(7) Given a sequence $A := \{a_1, a_2, \ldots, a_n\}$ of distinct real numbers, find the size of the largest subset such that for every $i < j$, $a_i < a_j$. In other words, find the length of the longest increasing subsequence of A. Your algorithm should have $O(n^2)$ or better running time. (Hint: use dynamic programming.)

Solution. Let $\text{opt}(k)$ be the length of the longest increasing subsequence of A subject to the constraint that the sequence ends with a_k. Since the longest increasing subsequence of A must end on some element of A, we can find its length by finding the maximum value of opt. It remains to actually compute the function $\text{opt}(\cdot)$.

Put

$$S_k := \{i : i < k \text{ and } a_i < a_k\}.$$

If $S_k = \varnothing$ then all elements that come before a_k are greater than it and thus $\text{opt}(k) = 1$ in this case. Otherwise, $S_k \neq \varnothing$ and we have the recursion

$$\text{opt}(k) = \max(\text{opt}(j) : j \in S_k) + 1.$$

In words, the longest subsequence sequence ending with a_k consists of the element a_k appended to the longest subsequence ending with some a_j where $j < k$ and $a_j < a_k$. The algorithm goes as follows.

```
1: procedure LongestIncreasingSubsequence(A)
2:     Let n be the length of A
3:     Let opt be an array of length n, initially all zero
4:     for k = 0, ..., n do
5:         max := 0
6:         for j = 0, ..., k do
7:             if A[k] > A[j] then // this restricts us to S_k
8:                 if opt[j] > max then
9:                     max := opt[j] // find max(opt(j) : j ∈ S_k)
10:                end if
11:            end if
12:        end for
13:        opt[k] := max + 1
14:    end for
15:    return max{opt[i] : i = 0, ..., n}
16: end procedure
```