CSE 431/531 Homework Assignment 2

Due in class on Thursday, Feb 15.

January 31, 2007

There are totally 7 problems, 10 points each. You should do them all. We will grade only 5 problems chosen at my discretion. If it so happens that you don't do one of the problems we don't grade, then no points will be deducted.

To "disprove" a statement, you must find a counter example to show that the statement is wrong. In general, your answers should be short but concise. (This will come with experience.)

Example 1 (Sample Problem). We want to make change for n cents using the least number of coins. The coins are of denominations $1 = c^0, c^1 \dots, c^k$, for some integers $c > 1$, and $k \geq 0$.

Devise an efficient algorithm to solve this problem.

Sample Solution. **When you are asked to devise an algorithm, please conform to the following format:** (a) describe the idea, (b) write the pseudo code, (c) prove its correctness, (d) analyze its running time.

(a) **Idea**: our algorithm is a greedy one. We start by taking as many coins of denomination c^k as possible, then as many of c^{k-1} as possible, and so on. Since there's a coin of denomination 1 (coin c^0), this process is guaranteed to finish.

(b) **Pseudo code**: every solution S to this problem is of the form

$$S = (f_0, f_1, \dots, f_k)$$

where the f_i are all natural numbers, f_i is the number of coins of denomination c^i we took, and

$$f_0 c^0 + f_1 c^1 + \dots f_k c^k = n.$$

COIN-CHANGING(n, c, k)

for $j \leftarrow k$ **downto** 1 **do**
    $f_k \leftarrow \lfloor n/c^k \rfloor$
    $n \leftarrow n - f_k c^k$
end for

(c) **Proof of correctness**: We will use the "first type" of induction. Basically, our greedy choice is to set $f_k = \lfloor n/c^k \rfloor$, and then solve the subproblem with coins of denominations $c^0, \dots, c^{k-1}$, and the new number of cents $n' = n - f_k c^k$. The cost of a solution S is $g(S) = f_0 + \dots + f_k$. We'd like to find an S minimizing $g(S)$.

Lemma 1. *There exists an optimal solution* $O = (f_0, f_1, \dots, f_k)$ *for which* $f_k = \lfloor n/c^k \rfloor$.

Proof. If $n < c^k$, $f_k = 0 = \lfloor n/c^k \rfloor$ for any feasible solution. Thus, we can assume $n \geq c^k$.

Let $O' = (f'_0, f'_1, \ldots, f'_k)$ be any optimal solution. If $f'_k = \lfloor n/c^k \rfloor$, then we are done. Suppose $f'_k \leq \lfloor n/c^k \rfloor - 1$. We have

$$f'_0 c^0 + f'_1 c^1 + \cdots + f'_{k-1} c^{k-1} = n - f'_k c^k \geq n - \left(\lfloor n/c^k \rfloor - 1 \right) c^k \geq c^k. \tag{1}$$

Now, if $f'_j \leq c - 1$, $\forall j = 0, \ldots, k-1$, then

$$f'_0 c^0 + f'_1 c^1 + \cdots + f'_{k-1} c^{k-1} \leq (c-1)(c^0 + \cdots + c^{k-1}) = (c-1)\frac{c^k - 1}{c - 1} = c^k - 1,$$

contradicting (1).

Hence, there must be some $j \in \{0, \ldots, k-1\}$ for which $f'_j \geq c$. However, if we reduce f'_j by c, and increase f'_{j+1} by 1, then we get another feasible solution where the total number of coins is $(c-1)$ less, contradicting the fact that O' is optimal. □

Lemma 2. *Let $O = (f_0, \ldots, f_k)$ be an optimal solution for which $f_k = \lfloor n/c^k \rfloor$, then $O' = (f_0, \ldots, f_{k-1})$ is an optimal solution to the problem with the number of cents $n' = n - c^k \lfloor n/c^k \rfloor$ and coin denominations $c^0, \ldots, c^{k-1}$.*

Proof. If there is a better solution $O'' = (f''_0, \ldots, f''_{k-1})$ for the subproblem, i.e.

$$\begin{aligned} f''_0 + \cdots + f''_{k-1} &< f_0 + \cdots + f_{k-1} \\ f''_0 c^0 + \ldots f''_{k-1} c^{k-1} &= n'. \end{aligned}$$

Then, obviously $(f''_0, f''_1, \ldots, f''_{k-1}, f_k)$ is a better solution for the original problem than O, contradiction! □

Theorem 1. *Algorithm* COIN-CHANGING *returns an optimal solution S.*

Proof. We show by induction on k that $g(S) = g(O)$, where O is an optimal solution.

The base case when $k = 0$ is trivial.

Consider $k > 0$. Let $S = (f_0, \ldots, f_k)$, and O be any optimal solution $O = (f'_0, \ldots, f'_k)$ for which $f'_k = f_k = \lfloor n/c^k \rfloor$. Such an optimal solution exists due to Lemma 1.

By induction hypothesis, $(f_0, \ldots, f_{k-1})$ is an optimal solution to the sub-problem. By Lemma 2, $(f'_0, \ldots, f'_{k-1})$ is an optimal solution to the sub-problem also. Thus,

$$f'_0 + \cdots + f'_{k-1} = f_0 + \cdots + f_{k-1},$$

which implies $g(S) = g(O)$ as desired. □

(d) **Running Time:** There is a loop of k iterations. The time in each iteration is dominated by computing $\lfloor n/c^k \rfloor$. We do not discuss numerical algorithms in this course, hence I will not analyze precisely the running time of this algorithm. (Refer to Knuth's TACP for numerical computation algorithms.) Let's just say $f(n, k)$ is the time it takes to compute $\lfloor n/c^k \rfloor$, then our algorithm runs in time $\Theta(k f(n, k))$.

□

Problem 1. Our CSE department has one supercomputer and (infinitely) many identical PCs. There are n distinct jobs $J_1, \ldots, J_n$, which can be performed completely independently of one another. Each job consists of 2 stages: first it needs to be *pre-processed* on the supercomputer, and then it needs to be *finished* on a PC. Job J_i needs p_i seconds on the computer, followed by f_i seconds on a PC. Since there are many PCs, the finishing of the jobs can be performed fully in parallel. The problem is, however, the supercomputer can only process one job at a time.

You are asked to design a scheduling of jobs on the supercomputer. The *completion time* of a schedule is the time at which all jobs will have finished processing on the PCs.

Assuming the transition time between the supercomputer and a PC is negligible. Give a polynomial time algorithm that finds a schedule minimizing the completion time.

Problem 2. There are n jobs $J_1, \ldots, J_n$ to be processed on a single machine. Only one job can be processed at a time. The starting time is 0. Job i requires t_i seconds to be processed. For any schedule, the completion time C_i of job i is the time at which the job is completely processed. Each job i also has a "weight" $w_i > 0$.

Devise an efficient algorithm to find a schedule (an ordering of jobs) which minimizes the weighted sum $\sum_{i=1}^{n} w_i C_i$.

(For example, suppose there are two jobs, $t_1 = 1, w_1 = 10, t_2 = 3, w_2 = 2$. Then, doing job 1 first would yeild a weighted completion time of $10 \cdot 1 + 2 \cdot 4 = 18$, while doing the job 2 first would give $10 \cdot 4 + 2 \cdot 3 = 46$.

Problem 3 (Textbook, Problem 19, Chapter 4). A group of network designers at the communications company CluNet find themselves facing the following problem. They have a connected undirected graph $G = (V, E)$, in which the nodes represent sites that want to communicate. Each edge e is a communication link, with a given available bandwidth b_e.

For each pair of nodes $u, v \in V$, they want to select a single u-v path P on which this pair will communicate. The *bottleneck rate* $b(P)$ of this path P is the minimum bandwidth of any edge it contains; that is, $b(P) = \min_{e \in P} b_e$. The *best achievable bottleneck rate* for the pair u, v in G is simply the maximum, over all u-v paths P in G, of the value $b(P)$.

It's getting to be very complicated to keep track of a path for each pair of nodes, and so one of the network designers makes a bold suggestion: May be one can find a spanning tree T of G so that for *every* pair of nodes u, v, the unique u-v path in the tree actually attains the best achievable bottleneck rate for u, v in G. (In other words, even if you could choose any u-v path in the whole graph, you couldn't do better than the u-v path in T.)

This idea is roundly heckled in the offices of CluNet for a few days, and there's a natural reason for the skepticism: each pair of nodes might want a very different-looking path to maximize its bottleneck rate; why should there be a single tree that simultaneously makes everybody happy? But after some failed attempts to rule out the idea, people begin to suspect it could be possible.

Show that such a tree exists, and give an efficient algorithm to find one. That is, given an algorithm constructing a spanning tree T in which, for each $u, v \in V$, the bottleneck rate of the u-v path in T equal to the best achievable bottleneck rate for the pair u, v in G.

Problem 4. You are given n closed intervals $I_1, \ldots, I_n$ on the real line, where $I_j = [s_j, f_j]$, and n real numbers t_j, $1 \leq j \leq n$.

Devise an efficient algorithm to decide if there is a way to assign to each number t a distinct interval I such that $t \in I$. In other words, we want to know if there exists a one-to-one pairing of the t_j and the I_j, so that each number belongs to its corresponding interval. If possible, you should make your running time $O(n^2)$.

Problem 5. For any graph G and any minimum spanning tree T of G, is there a valid execution of Kruskal's algorithm on G that produces T as output? Give a proof or a counter example.

Problem 6. Consider the minimum spanning tree problem on an undirected graph $G = (V, E)$ with a cost $c_e \geq 0$ on each edge, where the costs are not necessarily distinct. When the costs are not distinct, there can in general be many distinct minimum-cost solutions.

Suppose we are given a spanning tree T with the guarantee that for every edge $e \in T$, e belongs to *some* minimum-cost spanning tree in G.

Can we conclude that T itself must be a minimum spanning tree in G? Give a proof or a counterexample with explanation.

Problem 7. Given a connected graph $G = (V, E)$. Let $n = |V|$. Each edge in G is already colored with either RED or BLUE. Devise an efficient (i.e. polynomial-time) algorithm which, given an integer k, $1 \leq k \leq n - 1$, **either** (a) returns a spanning tree with k BLUE edges and $n - 1 - k$ RED edges, **or** (b) reports correctly that no such tree exists.