Problem Set 5 Solutions

Problem 5-1. Computing Logs.

You bring an L-foot log of wood to your favorite sawmill. You want the log cut at k specific places, $L_1, L_2, \ldots, L_k$ feet from the left end, where $L_1 < L_2 < \ldots < L_k < L$. The sawmill charges x dollars to cut an x-foot log once, in any place you want. Give an efficient algorithm to minimize the total cost.

Solution:

Notation: Set $L_0 = 0$ and $L_{k+1} = L$. Let c_{ij} be the minimum cost of completely cutting the log with the left endpoint at L_i and the right endpoint at L_j.

Then we can write the recurrence for computing $c_{i,j}$ as follows:

$$c_{i,j} = \begin{cases} (L_j - L_i) + \min_{i<m<j}(c_{i,m} + c_{m,j}) & j > i+1 \\ 0 & \text{otherwise} \end{cases}$$

The minimum-cost is returned in $c_{0,k+1}$. There are $O(k^2)$ subproblems and it costs $O(k)$ to compute each subproblem. Thus the total time to compute the minimum cost is $O(k^3)$. In order to compute the optimal order of cutting, we store along with $c_{i,j}$ a value of m corresponding to the minimum cost. Then L_m is the place at which we should cut the log.

Problem 5-2. Making Change

Consider the problem of making change for n cents using the least number of coins. Ben Bitdiddle, a cashier at the local Star Market, claims that a greedy algorithm is correct: "It's best to first use as many quarters as possible, because using smaller denominations would require more coins for the same amount of money. Then use as many dimes as possible, by the same reasoning. The same goes for nickels. Then use pennies to finish the job."

(a) Ben's manager questions his reasoning. Is this a valid argument that the greedy algorithm is optimal? Explain why or why not.

Solution:

Ben's reasoning is bogus. First note that nothing in his argument depends upon the particular denominations (quarters, dimes, nickels, pennies). While a greedy algorithm *does* work for US coin denominations, Ben's argument does not constitute a proof of optimality in general. Specifically, it does not include a proof of the greedy-choice property, which in fact does not hold for certain other sets of denominations (see part **(c)**).

(b) Suppose that the available coins are in the denominations $c^0, c^1, \ldots, c^k$ for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm yields an optimal solution.

Solution:

When the coin denominations form a geometric series, the following algorithm produces a multiset A on the set $\left\{``c^0", ``c^1", \ldots, ``c^k"\right\}$ such that $\sum_{j=0}^{k} \mu_A(``c^j")c^j = n$. Here $\mu_A(``c^j")$ means the number of "c^j" in the multiset A.

GREEDY-GEOMETRIC-CHANGE(n)

$A \leftarrow \emptyset$
for $j \leftarrow k$ **downto** 0
    do while $n \geq c^j$
        do $A \leftarrow A + \{``c^j"\}$
            $n \leftarrow n - c^j$
return A

It is clear that this algorithm makes change for n cents correctly (since $c^0 = 1$). We will now show by induction on n that this algorithm yields the optimal solution.

Let j be such that $c^j \leq n < c^{j+1}$ (if $j > k$ then set $j = k$). The algorithm returns $A = \{``c^j"\} + A'$ where, by the inductive hypothesis, A' is an optimal solution for $n - c^j$. Let B be an optimal solution for n. For the greedy-choice property we show that B must contain a "c^j". Suppose B does not contain a "c^j". Then B contains only elements "c^i" with $i < j$, so

$$\sum_{i=0}^{j-1} \mu_B(``c^i")c^i = n \geq c^j. \tag{1}$$

Now assume that $\mu_B(``c^i") < c$ for each $0 \leq i < j$. Then

$$\sum_{i=0}^{j-1} \mu_B(``c^i")c^i \leq (c-1)\sum_{i=0}^{j-1} c^i = (c-1)\frac{c^j - 1}{c-1} = c^j - 1.$$

But this contradicts (1), so there must be some i in the range $0 \leq i \leq j-1$ for which $\mu_B(``c^i") \geq c$. Then we can replace the c "c^i"s with a single "c^{i+1}" which contradicts the optimality of B. Thus B must contain a "c^j".

To prove the optimal-substructure property, let j be defined as before ($c^j \leq n < c^{j+1}$) and let S be an optimal solution. We need to show that $S' = S - \{``c^j"\}$ is an optimal solution for $n - c^j$. If it were not, then it would consist of more coins than an optimal solution $S''$. Then $S'' \cup \{``c^j"\}$ would be an optimal solution for n, and would consist of fewer coins than S, which is a contradiction, since S was assumed to be optimal. From the greedy-choice and the optimal-substructure properties it follows that GREEDY-GEOMETRIC-CHANGE must produce an optimal solution.

(c) Give a set of coin denominations for which the greedy algorithm does not always yield an optimal solution. (Your set must include pennies, so that exact change can always be made.)

Solution:

Consider a set of denominations containing only quarters, dimes, and pennies. To optimally make change for 30 cents, one would use 3 dimes. However, a greedy strategy would use one quarter and 5 pennies. There are many similar examples.

(d) Give an $O(nk)$ dynamic programming algorithm which works for any set of k different coin denominations. You may assume that the set includes pennies.

Solution:

Suppose we have k different coins with denominations $d_1, \ldots, d_k$. Let $c(i)$ denote the minimum number of coins needed to make change for i cents. We let $c(i) = \infty$ for $i < 0$, and we let $c(0) = 0$. Then for $i > 0$ we can compute $c(i)$ by

$$c(i) = \min_{1 \le j \le k} \left(c(i - d_j)\right) + 1 \ . \tag{2}$$

By applying Equation 2 iteratively, $c(n)$ can be computed in $O(kn)$ steps (because $O(n)$ minimums have to be computed and each takes $O(k)$ time). To actually generate the set of coins, for each $c(i)$, we record the value of j which minimized Equation 2, then, once the table is computed, we trace back through these values.

Problem 5-3. Greedy Graphs

In this problem, assume that graphs are given by an adjacency-list representation.

(a) An **independent set** of a graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices such that for any two vertices $u, v \in V'$, $(u, v) \notin E$ (i.e., there is no edge between any two vertices in an independent set). A **maximal independent set** is an independent set V' such that for all vertices $v \in V - V'$, the set $V' \cup \{v\}$ is not independent (i.e., every vertex not in V' is adjacent to some vertex in V'). Give an $O(|V| + |E|)$ algorithm for finding a maximal independent set for any graph $G = (V, E)$.

Solution:

Assume that V' is initially empty. Start from any vertex v in V. Add v to V' and remove v along with all of its neighbors from V. Repeat this process until V is empty. The resulting set V' is independent and is maximal since any vertex in $V - V'$ is a neighbor of some vertex in V'. This constitutes the basic correctness argument. The algorithm runs in $\Theta(|V| + |E|)$ time because each vertex and each edge is examined only once.

Notice that there may be many different independent sets of different sizes. We are not trying to find an independent set of maximum size. Nobody knows how to solve this problem in polynomial time. (Problems of this type are called **NP**-complete.)

(b) A **k-coloring** of an undirected graph $G = (V, E)$ is a function

$$\chi : V \to \{1, 2, 3, \ldots k\}$$

such that $\chi(u) \neq \chi(v)$ for every edge $(u, v) \in E$. In other words, each vertex is given a color (a number in $\{1, 2, 3, \ldots, k\}$), and adjacent vertices have different colors. Give an $O(|V| + |E|)$ algorithm that finds a $(d+1)$-coloring of a graph $G = (V, E)$, where d is the largest degree of the vertices of G.

Solution:

Pick an un-colored vertex v. v has at most d neighbors, some of them may already be colored, others may not. At most d different colors must be avoided to color v. Since there are $d+1$ colors to choose from, v can be colored. Repeat until the entire graph has been colored.

This algorithm runs in time $\Theta(|V| + |E|)$ proportional to the sum of the degrees of all vertices, which is $2|E|$. Intuitively this follows from the fact that each edge has to be examined at most twice.

Correctness can be proved by induction on $n = |V|$. When $n = 1$, a single vertex is colored correctly. Suppose that the algorithm finds a correct $(d+1)$-coloring of any graph with $n-1$ vertices, each with a degree at most d. We show that it follows that the algorithm also finds a correct solution for a graph of size n and the same degree restriction. Consider such a graph. The algorithm picks one vertex after another. Consider the set of $n-1$ vertices that are picked first. By the inductive hypothesis, they are colored correctly. Consider the last vertex v. The algorithm is able to color v with one of the $d+1$ colors since v's degree is at most d. Thus, the entire graph of size n is colored correctly.

Problem 5-4. The knapsack problem

In this problem, we explore variants on the famous “knapsack” problem. We have a knapsack of a given size M, and n objects with integer sizes $m_1, m_2, \ldots, m_n$. The general idea of the problem is to find a subset of objects that exactly fit into the knapsack, where by “fit,” we mean that the sum of the sizes of objects in the subset equals the size of the knapsack (i.e., the knapsack is one-dimensional).

(a) Give a recursive algorithm to determine whether there exists a subset of the n items that fits exactly into a knapsack of size M. How many distinct subproblems actually occur during the execution of your algorithm?

Solution:

The key insight is to note that the problem can be parameterized so that it exhibits the optimal substructure property. Specifically, let $Knap(i, j)$ deonte the knapsack problem where the knapsack has size j and objects $1 \ldots i$ can be placed in the knapsack. Let $Knap(i, j) = 1$ if there is a subset of the i objects which exactly fills the knapsack, and $Knap(i, j) = 0$ otherwise. Then

$$Knap(i, j) = Knap(i-1, j-m_i) \text{ or } Knap(i-1, j).$$

This formula says that in order to fill the knapsack, object i either is or is not used. If it is, then a subset of objects $\{1 \ldots i-1\}$ which uses the remaining $j - m_i$ space in the knapsack must be found. If it is not, then a subset of the remaining $i-1$ objects which uses the remaining j space in the knapsack must be found.

An obvious recursive algorithm is the divide and conquer algorithm described by the formula $Knap(i,j) = Knap(i-1, j-m_i)$ or $Knap(i-1,j)$. The base cases are as follows:

1. $Knap(i,0) = 1$ since one can always fill a knapsack of size zero by putting nothing in it.
2. $Knap(0,j) = 0$ for $j \neq 0$ because it is impossible to fill up non-zero space without using any objects.
3. $Knap(i,j) = 0$ for $j < 0$ because if $j < 0$ the knapsack is already overfull. There is no way that adding objects (or failing to add objects) can rectify this.

This algorithm has exponential running time since two subsubproblems are created to solve each subproblem. However, there are only $O(nM)$ distinct subproblems: $Knap(i,j)$ for $0 \le i \le n$ and $0 \le j \le M$.

(b) Give an $O(nM)$-time dynamic programming algorithm to solve the problem.

Solution:

The recursive formula given above can also be used as the basis for a dynamic programming solution to the Knapsack problem. This algorithm tabulates results in an $(n+1) \times (M+1)$ matrix K where $K[i,j] = Knap(i,j)$ $i \in \{0 \ldots n\}$, $j \in \{0 \ldots M\}$ when the algorithm terminates. The following code fills in this matrix in a bottom up fashion:

```
KNAPSACK(n, M)
for i ← 0 to n
      do K[i, 0] ← 1
for i ← 1 to M
      do K[0, i] ← 0
for i ← 1 to n
      do for j ← 1 to M
               do if K[i − 1, j] = 1
                     then K[i, j] ← 1
                  if (j − m_i ≥ 0) and (K[i − 1, j − m_i] = 1)
                     then K[i, j] ← 1
```

The algorithms running time is dominated by the nested for loops, which take $O(nM)$ time. $O(nM)$ space is used to store the array K.

(c) Improve your algorithm to use $O(M)$ space.

Solution:

The ith row of K depends only upon the $i-1$th row of K. Thus, there is no reason to store anything but the row we are presently computing and the previous row. This reduces the space $O(M)$.

(d) Improve your algorithm to print out the subset if it exists. Maintain the $O(M)$ space bound if you can.

Solution:

The obvious algorithm for printing out the solution uses $O(nM)$ space. In this solution an additional bit of information is stored in each matrix entry $K[i, j]$. It indicates whether the given subproblem was solved with or without object i (its value is meaningless if the subproblem has no solution, i.e. if $K[i, j] = 0$). Using these flags, it is easy to reconstruct the solution starting at $K[n, M]$ *if we keep the entire table around*. Unfortunately, this solution brings us back to $O(nM)$ space, which we would like to avoid if possible.

In order to use only $O(M)$ space, we must be more clever. The following is in fact a variant of the solution to part (c), but it is easier to explain it directly rather as a modification of that algorithm. We need an array $A[1 \ldots M]$ which will represent the knapsack. Originally every element of the array is initialized to -1. When the algorithm terminates, $A[i] = -1$ if and only if $Knap(n, i) = 0$. Furthermore, if $A[i] = j \neq -1$ then $A[i - m_j] \neq -1$. In other words $A[i] = j \neq -1$ tells us that object j is used in the solution to $Knap(n, i)$. We can then look in $A[i - m_j]$ to find an object which solves $Knap(n, i - m_j)$ and so on, until we reach $A[0]$. At this point we will have discovered all of the objects used to solve $Knap(n, i)$.

The code below considers the objects one at a time (the outer loop), and updates A to reflect the fact that the object under consideration can be placed into the knapsack (the inner loop).

KNAPSACK'(n, M)
1 **for** $i \leftarrow 1$ **to** M
2   **do** $A[i] \leftarrow -1$
3 **for** $i \leftarrow 1$ **to** n
4   **do for** $j \leftarrow M$ **downto** 1
5   **do if** $(j - m_i \geq 0)$ and $(A[j - m_i] \neq -1)$
6   **then** $A[j] \leftarrow i$

A loop invariant on the for loop starting on line 3 is that $A[l] = -1$ if $Knap(i, l) = 0$ and otherwise both $l - m_{A[l]} >= 0$ and $A[l - m_{A[l]}] \neq -1$. This implies correctness of the algorithm by the above description of how to reconstruct a solution to $Knap(n, i)$ given $A[i]$.