Algorithms Lab 7

The in-lab problems are to be solved during the lab time. Work with your team, but write your solutions individually. Your answers will not be graded, however you need to work through these problems; please staple the answers to the in-class exercises to the assignment that you hand in.

The homework problem set is due in one week. Work with your team, but write your solutions individually. List the people with whom you discussed the problems.

In lab

1. Rod cutting: Describe how to augment your solution for the rod cutting problem so that it computes the actual cuts corresponding to the optimal revenue, instead of just the revenue.

   For example, let's say that the optimal revenue for a rod of length 12 was achieved by cutting the rod at $\{5, 5, 2\}$. Show how to compute these cuts.

2. The longest-common subsequence (LCS) problem (separate handout).

Homework

1. Rod cutting: Someone suggets the following strategy as an optimization to our dynamic programming solution: when you determine the next cut, instead of trying all possibilities, go with a piece that maximizes the revenue per length. This is called a *greedy* strategy, because it greedily chooses the option that looks locally best, without exploring the choices "deeply".

   Naturally, greedy strategies always work in the sense that they always compute something —- the question for a particular greedy strategy is whether it always computes the correct (optimal) solution. In this case we say that the greedy strategy is correct.

   If this particuar greedy strategy for rod cutting is correct, then we could solve the problem in $O(n)$ time plus an initial sort (instead of $O(n^2)$ with dynamic programming).

   Prove that this strategy is not correct by showing a counterexample.

2. (**0-1 knapsack problem:**) The knapsack problem is the following: You are given n items, numbered 1 to n, and two arrays $w[1..n]$ and $v[1..n]$ that specify how much each item is worth and how much it weighs: item i is worth $v[i]$ and has weight $w[i]$ pounds.

Note that we index the arrays from 1 to n just to make it simpler. You are also given a knapsack of capacity W pounds. The goal is to fill the knapsack with items so that the total value of the items in the backpack is maximal. Put differently, you want to find a subset of items such that their total weight is smaller than W (i.e. they fit into the backpack) and their total value is maximized, over all such subsets.

For simplicity, we'll start by *only* computing the maximal value that we can put in the backpack. Once we know how to do this, we'll show that the solution can be extended to compute not only the value, but the actual set of items that achieve this value.

Let's try to come up with a solution for this problem.

(a) Imagine that one of your (algorithms-savy) friends claims that the following greedy strategy always works: pick the items with the biggest value-per-pound, and recurse. Prove him wrong by showing a counter-example —that is, show that there exist an instance of the knapsack problem where using the greedy strategy will not give the optimal solution.

(b) You have yet another friend who claims that the following greedy strategy always works: Select the items in order of their values; that is, select with largest value first, and so on. Prove him wrong.

(c) So...you can't realy on your friends, you have to solve it yourself; great! Let's do it! If you think about it a little bit, you see that the problem is recursive and it has optimal sub-structure: once you pick an item to put in the backpack, you need to fill the remaining capacity with the remaining items, optimally. That is, you have to find the largest value you can put in the remaining capacity with the remaining items. Do you start seeing the recursive formulation?

Often the hardest part is coming up with a notation. Let us denote:

optknapsack(k, w): the maximal value obtainable when filling a knapsack of capacity w using items among items 1 through k.

To solve our problem we need to compute *optknapsack*(n, W).

Come up with the recursive formulation for *optknapsack*(k, w). *Hint: The idea is to consider each item, one at a time. Let's take item k: either it's part of the optimal solution, or not. We need to compute both options, and chose the best one.*

(d) Show the recursive algorithm for computing *optknapsack*(k, w).

(e) Analysis: Let $T(n, W)$ be the running time of *optknapsack*(k, w).
Write a recurrence relation for $T(n, W)$.

(f) Argue that $T(n, W)$ is exponential.

(g) Describe a dynamic programming solution and analyze its running time.

(h) How can we modify the dynamic programming algorithm for 0-1 Knapsack from simply computing the best value, to actually computing the actual set that gives this value?

3. You have been hired to design algorithms for optimizing system performance. Your input is an array $J[1..n]$ where $J[i]$ or J_i represents the running time of job i; jobs do not have specific start and end times, but they can be started at any time (this is a different scenario than in interval scheduling). The running times are integers.

   Generally speaking, your task is to find an optimal load balance of these tasks over two processors.

   Design an algorithm for determining whether there is a subset S in J such that the running time of the elements in S sum up precisely to the same amount as the sum of the elements not in S; more formally, $\sum_{J_i \in S} J_i = \sum_{J_i \in J-S} J_i$. The algorithm should run in time $O(n \cdot N)$, where N is the sum of the running times of the n jobs.