Algorithms Lab 7

The problem set is due in one week. The goal of the labs is to give you practice problems to work on. More important that getting the right answers is the process of coming up with the solution. Discussing and bouncing ideas with peers in the classroom can be useful, if done in the right way. If your exam grade was lower than your lab average, you should work alone and use the TAs, not your peers, for questions. If you end up discussing problems with your peers, list their names.

In lab

1. Complete all questions in the rod-cutting handout.

   Also: 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.

Homework

1. Write code, in a language of your choice, to implement the rod cutting problem. You need to implement TWO functions:

   - One that corresponds to solving the problem recursively without DP. You could name it `maxrev_recursive`.
   - One that corresponds to solving the problem with DP (it can be recurive or not, your choice). You could name it `maxrev_DP`.
   - Each function should take i (the length of the rod) as variable and should retun the optimal revenue of cutting a rod of length i.
   - The array that holds the prices can be a global.
   - The length of the rod n has to be read from the command line. If you use C/C++, use `argc, argv`. Use something similar if you use python, but make sure that the user does not need to edit your code to change n.
   - Initialize the and price array with random values and (assume that prices integers), but come up with a reasonable scheme where prices are generally not decreasing (e.g.: a rod of length 10 should get a price that's the same or better than a rod of length 9).

- Your code shoud time the two functions functions and show the timings. For example when I run Ryan's code, I expect to see something like this:

```
g++ ryan.cpp -o ryan
./ryan 100
you entered n=100
running rev_dynprogr  with n=100: maxrev= 128, total time = ...
running rev_recursive with n=100: maxrev= 128, total time =
```

- Test both functions on increasing values of n, until you notice a significant difference in running time between the two methods.
- Run your code on the price array in the handout, and print how long the two functions take, and what maximum revenue you get.
- What to turn in: email me the .cpp file or the python file. When you hand in the lab, attach a hard copy of your code, and the results of running your code (1) on the array in the handout; and (2) on a random array of size n large enough so that teh DP solution is significantly faster.
- This problem is individual, you cannot have a partner. But you can discuss it with anyone in the class, as long as you follow the honor code.

2. 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.

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.