CS41 Lab 8: Dynamic Programming

Thursday, March 19 2015

The goal of this week's lab is to get hands-on experience with dynamic programming. For the first part of the lab, you'll be experimenting with several different solution implementations for the Steel Rod problem. I provided three to start:

- `brute-force.cpp`: straight brute-force implementation. Iterate over all possible sets of cuts, compute how much revenue is gained, and keep track of the maximum revenue
- `cutrod.cpp`: brute-force-ish implementation which uses the naive recursive calls from Wednesday's class: to compute how much revenue can be obtained by cutting at k-feet, make a recursive call on the $(n-k)$-foot rod and add in $P[k]$.
- `cutrod2.cpp`: another recursive brute-force-ish implementation which uses the recursive formulation with two recursive calls: to compute how much revenue can be obtained by cutting at k-feet, make recursive calls on both the k-foot rod and the $(n-k)$-foot rod.

Each solution uses ICPC-style semantics for I/O: input comes from standard-in, and output goes to standard-out. However, it might be easier to read in input from a file. To do this, compile your program and run e.g.

```
./bf < inputs/ten.in
```

1. First, open up `brute-force.cpp`.
   (a) Look over the code to make sure you understand what it does.
   (b) Then, compile the code. e.g. with
   ```
   g++ -o bf brute-force.cpp
   ```
   (c) Run the code on a couple of input sets. Sample usage:
   ```
   ./bf < inputs/twenty.in
   ```
   (d) Now, time this implementation using the UNIX `time` command:
   ```
   time ./bf < inputs/twenty.in
   ```
   How long does it take to run on a twenty-foot steel rod? ten feet? Try to find an input size which makes the brute-force algorithm take roughly one minute.
2. Repeat the steps for `cutrod.cpp` and `cutrod2.cpp`. Which implementation runs fastest? How large should the inputs be if the program is to run one minute?
3. Now, open up `dp.cpp` and implement a dynamic programming solution to the Steel Rod Problem.
   (a) compile and run your program on some of the same inputs as you ran the brute-force program(s). Does your program give the same output? (If not, your program is buggy).

(b) Time your program on different input sizes, as in the previous problem. How large should the input be so your program runs roughly one minute?

(c) If you have time in this part, modify your program so it outputs both the maximum possible revenue and which cuts are needed to obtain that revenue.

4. For the remainder of class, work on one of the following dynamic programming problems. Focus on the first two steps of the dynamic programming process; don't worry about constructing pseudocode. **Hint:** Focus on the **choice** you might make to construct an optimal solution. For example, with the Steel Rod Problem, our choice was where to make the left-most cut.

   (a) **Harry's Hoagie Hut**. Harry is back, and in an effort to increase profit, he has gotten rid of his one-price-fits-all business plan. This time, there are n hoagies $\{1, \ldots, n\}$, with start times $\{s_i\}$, finish times $\{f_i\}$. Furthermore, each hoagie has a profit p_i. As always, Harry has no help and can only make one hoagie at a time.

   Design a polynomial time algorithm that takes the start time, finish time, and profit for n hoagies and returns the maximum profit Harry can make.

   (b) **Minimum Cost Paths**. Let $G = (V, E)$ be a directed graph, with edge costs $\{c_e : e \in E\}$. In this problem, edge costs can be negative. Design a polynomial time algorithm that takes $G, \{c_e\}$ and special vertices $s, t \in V$ and returns the minimum cost $s \rightsquigarrow v$ path, where the cost of a path P is $\sum_{e \in P} c_e$. Assume there are no negative cost cycles.

   Note: when edge costs are all positive, you can just use Dijkstra's algorithm. Unlike Prim's Algorithm for minimum spanning trees, Dijkstra's algorithm doesn't extend to negative edges.