Henry Daniels-Koch
November 4 2016

Algorithms: Lab 7

1). I implemented two algorithms for rod cutting. One algorithm solves the problem recursively with dynamic programming. Since the algorithm recursively calls itself on all elements, it has at exponential runtime of at least $T(n) = \Omega(2^n)$. I also implemented the rod cutting algorithm with dynamic programming, which stores maximal revenues for already computed rod sizes. Since the rod is of length n, this algorithm run in linear $T(n) = O(n)$ time. Both functions read the length of the rod from the command line and are generally initialized with random values such that the price of each piece is greater than all prices of smaller pieces. I ran my code on various rod lengths and compared the runtimes shown in the following table:

We observe that dynamic programming significantly decreases runtime by many factors of magnitude and confirm our predicted exponential and linear runtime relative comparisons. We also ran both algorithms on the price array in the handout and obtain a maximum revenue of 38.

2). If we were to implement a greedy strategy to solve the rod cutting problem, we would not always get the optimal solution. This is evident in the following counterexample:

A greedy solution picks the piece with the highest revenue per length which would be a length of 3 with rev/length 51. It would then take a piece of length 1 for a total greedy revenue of 153 + 40 = 193. The optimal solution is to pick two pieces of length 2 for an optimal revenue of 100 + 100 = 200. Thus, the greedy solution provides a revenue less than the optimal revenue.

3). To find the optimal load balance on two processors, we split the load in half such that a subset of the load can run for the same amount of time on one processors as the rest of the set can run on the other processor. Therefore, all we need to do is find one subset that runs for half of total runtime. Therefore, we first find the total runtime (N) by summing over all the individual runtimes. To implement our algorithm with dynamic programming, we need to initialize a two dimensional table of n rows for each job and N/2 columns for all the possible runtimes. Both of these actions occur before our function. Inside of our function, we recursively call our function twice: once that includes the job we are at in our sum and once without the job. This way, we can check every possible sum. We call our function recursively starting from the left and on smaller and smaller indices until we hit a base case and adding to an updated sum accordingly. Our four base cases are as follows: the sum is half of the total, which is what we are looking for, the index is out of bounds, the sum is greater than half of the total in which case the subset of jobs cannot possible be half of the sum, and the case where we have already computed whether a half sum is possible with the current jobs. Lastly, after our recursive calls of our function, we store whether either list of jobs had a sum that was half of the total as a boolean value and then return that boolean value. Since this table stores up to $n * N/2$ values, this algorithm runs in $O(n * N)$ time. The pseudocode is follows:

initialize a table T[i][sum] with ints of -1
total = sum of all elements

//runtime is table of runtimes

```
bool optSplit(runtime, table, i, sum)
    if(()sumWith == 0.5*total), or (sumWithout == 0.5*total)) : return true
    if(i < 0) : return false
    if(sum > 0.5*total): return false
    if(T[i][sum] != -1) : return T[i][sum]
    bool with = optSplit(runtime, table, i-1, sum + runtime[i])
    bool without = optSplit(runtime, table, i-1, sum)
    T[i][sum] = (with or without) //boolean value
    return (with or without)
```