Midterm 2 Take-home

Joe Puccio

November 19, 2014

Collaborators: Rahul Ramkumar, Max Daum, Matthew Leming, Fred Landis, Nathan Weatherly, Devin Beauchamp, and Cameron Wall.

1.

a) Input: n, quantity to make change from (in cents). Output: array of how many of each denomination was used (in descending order), e.g. $(4, 1, 1, 1)$ means 4 quarters, 1 dime, 1 nickel, and 1 penny. Invariant: at the beginning of each subtraction of a given denomination, c_k, the amount remaining at that point, call it r, is $< c_{k-1}$ (when, of course, $k \neq 0$). The greedy algorithm is: begin with your total amount, select your highest denomination (quarters; 25), and iteratively subtract until the remaining amount is less than this highest denomination (< 25), then continue the process with the remaining amount and the second highest denomination (dimes; 10) until the remaining amount is less than this second highest denomination (< 10), and continue identically for the remaining two denominations.

This greedy algorithm yields an optimal solution if and only if the problem exhibits optimal substructure and the greedy choice property. Let's first show optimal substructure (via proof by contradiction): assume set O (of the form $\{c_4, c_4, c_3..\}$, where c is the set of denominations) to be the optimal solution for making change out of n cents, and say our algorithm first subtracts some coin with denomination c_d, and has cent value c_v, we want to show that $O - \{c_d\}$ is an optimal solution to the subproblem $n - c_v$. Well, let's pretend that there is a solution superior to $O - \{c_d\}$ for changing $n - c_v$ cents, called S. Well, that would mean that S used strictly fewer coins than $O - \{c_d\}$, which would imply that $S \cup c_d$ would use strictly fewer coins than O to change n cents. But that violates our assumptions that O was optimal for n cents, and therefore this superior set, S, for arbitrary sub-problem $(n - c_v)$ must not exist. Therefore $O - \{c_d\}$ is optimal for $n - c_v$, and our problem exhibits optimal substructure.

Now we must show that our problem exhibits the greedy choice property: (let's break things down into cases) if our first subtraction is a penny, then that means that the input must have been at most 4 cents, and the optimal solution couldn't do any better either, because using a higher denomination isn't valid. If our first subtraction is a nickel, then the input must have been between 5 and 9 cents. We know that the optimal solution wouldn't use 5 or more pennies, as 5 of them could be replaced with a nickel, and the remaining 4 would be treated by the previous case (note: we can refer to previous cases, because they are in fact subproblems and we have shown that optimal subproblems yield optimal solutions because we have shown optimal substructure). If our first subtraction is a dime, then the input must have been between 10 and 24 cents. We know that the optimal solution wouldn't use more than 2 nickels, because these can be replaced by a dime and similarly for more than 5 pennies because these could be replaced by a nickel. Again, the remaining cents after this subtraction are handled by previously mentioned subproblems (which again, are valid due to optimal substructure). Lastly, if our first subtraction were a quarter, then the amount to change is at least 25 cents. The optimal choice certainly couldn't use more than 3 dimes (could be replaced with a quarter), 2 nickels (could be replaced by a dime), or 5 pennies (could be replaced by a nickel). And the remaining change is again, handled by previous cases. Thus, we have shown both optimal substructure and the greedy choice property, so we may conclude that our greedy algorithm is optimal. □

b) Using the denominations $\{8, 6, 1\}$, choose $n = 12$ and find that the greedy algorithm yields $|\{8, 1, 1, 1, 1\}| = 5$ while the optimal is $|\{6, 6\}| = 2$. We notice that this problem does not satisfy the greedy choice property, because the locally optimal choice (subtracting 8 from 12), is not consistent with the globally optimal solution (waiting until 6 is considered, and subtracting 6 twice).

c) Input: n, quantity to make change from (in cents), int array c containing the denominations, int array *solns* initialized to all -1's. Output: an array containing the optimal number of each denomination that yields the input n. Invariant: for any $n \geq 0$, with coin denominations $c[1...k] \leq n$, we have that the optimal way to represent n is exactly $1 + min(minCoins(\{n - c[0], n - c[1], n - c[2], ..., n - c[k]\}))$, that is, the minimum of the minimum number of coins that may be used to represent each of the subproblems created by subtracting each denomination. This code reveals optimal substructure because its use of recursion demonstrates that it's solving each problem optimally by optimally solving it's subproblems (that is, upon each evaluation of a subproblem of size $g < n$, it recursively examines each of the possible ways of solving this subproblem, g, by retroactively choosing the best solution to each of g's subproblems). The number of subproblems is kn, where n is the original amount to make change out of and k is the number of denominations.

```
findMinCoins(int n, int[] c){
        int[] solns = new int[n]; // filled with -1s
        return recursFind(n,c,solns);
}

recursFind(int n; int[] c, int[] solns){

        if(c.contains(n)) return 1; //we know we can produce with one coin
        if(n<0) return -1; //change could not be made by this attempt

        if(solns[n]<0){//haven't solved this subproblem yet
                int minCoinsForN= -1 //need to initialize
                for(int i=1; i<=c.length; i++){//using 1 indexing
                        int potentialMin = recursFind(n-c[i],c,solns);
                        if(minCoinsForN==-1)
                                minCoinsForN = potentialMin;
                        if(potentialMin<minCoinsForN && potentialMin != -1)
                                minCoinsForN = potentialMin;
                        if (minCoinsForN != -1)
                                solns[n]=1+minCoinsForN;
                }
        }
        return solns[n] //here's the dynamic
        //programming bit: we don't calculate
        //subproblems that we've already computed
}
```

2.

a) Answer is given.

b) We cannot solve this problem via a greedy algorithm, which we can see by examining the following situation: first, define a variable $r_i = p_i - f_i$, where r_i denotes the revenue that can be obtained from dock i.

Now, imagine the case where all docks are equally spaced ($d_i = d_{i+1}\ \forall i$) and D, the distance you can travel on a full tank of gas, is exactly the distance between two docks. And imagine the following setup: $d_1 = -7$, $d_2 = -8$, $d_3 = -5$, $d_4 = 0$. A greedy algorithm would examine all docks reachable in D (that is, d_1, d_2), and choose the best option locally (d_1), without considering the consequences of the choice, and subsequently choose d_3, resulting in an overall revenue of -12. However, it's evident that a dynamic programming algorithm would notice that the optimal path is $d_2 \rightarrow d_4$, which yields overall revenue of -8. Now let's consider the dynamic programming algorithm which examines the consequences of choices recursively, and follows the path that maximizes revenue.

We assume all $d_i \leq D$, as otherwise we could not reach a dock even on a full tank of gas. Because g_i is assumed to be constant for all n, and because we have to fill up our tank at dock n regardless of our gas usage, we don't need to concern ourselves with which docks to stop at due to gas concerns. Additionally, because we have sufficient product ($K = n$), we don't need to concern ourselves with arriving at a dock without having any product to sell. Therefore, we only need to concern ourselves with the price of product p_i and the fee of docking f_i for each dock. We can again define a variable $r_i = p_i - f_i$, denoting the revenue possible at each dock. We can approach this problem with dynamic programming recursively as so: at each dock, d_i, examine all docks within reach of distance D (the distance that could be traveled on a full tank of gas; again, where gas is purchased does not matter, so we can always assume that our tank is full), and for each of those docks, d_k, determine the revenue obtained at d_i as well as (plus) the revenue attainable assuming the raft moves to d_k (calculated recursively, of course, by repeating the same process but on d_k instead of d_i), and choose the maximum of these options produced from each d_k. Also, notice that when a given $p_i - f_i < 0$, then we should always choose not to stop at this dock to sell, unless necessary for gas, as we would incur only a cost; this algorithm does take those cases into account. This is solving the original problem optimally ($d_i = d_0$), by recursively solving its subproblems optimally (all d_k's).

The algorithm thus far can be summarized as

$$optimalPath(d_i) = (p_i - f_i) + max(optimalPath(\{d_{i+1}, d_{i+2}, ..., d_{j-1}, d_j\}))$$

where d_j is the farthest reachable dock from d_i with a full tank of gas.

However, because many subproblems overlap for parent problems, we can cache the solutions to these subproblems once they're calculated so that redundant calculations aren't made (the essence of dynamic programming). Because the resulting number of subproblems we have is n, our run time is of order n, that is $O(n)$.

c) The algorithm for this problem follows almost trivially from part b. To repeat the assumptions and setup: we assume all $d_i \leq D$, as otherwise we could not reach a dock even on a full tank of gas. Because g_i is assumed to be constant for all n, and because we have to fill up our tank at dock n regardless of our gas usage, we don't need to concern ourselves with which docks to stop at due to gas concerns. However, now we are not guaranteed sufficient product to sell at each dock. If $K \geq n$, then our solution from part b can be used. If not, our algorithm better take into account the revenue factor ($r_i = p_i - f_i$) as well as the number of products available to sell at any given subproblem. Thus, our dynamic programming solution is: for each dock, d_i, examine all docks within distance D of d_i, and for each of those docks, d_k, determine the revenue that could be obtained at d_k and by docks reachable from d_k (recursively), for each of the following two cases: assuming product had been sold at d_i, and assuming product had not been sold at d_i. This will ensure that we are only adding on the revenue for a given d_k when we still have a product to sell at d_k. Note that now we need not specially consider the case where a given $p_i - f_i < 0$, because now at every i we consider both the case where we had sold at i and where we had not sold at i.

The algorithm at this point can be summarized as

$$optimalPath(d_i, n) = \begin{cases} max((p_i - f_i) + optimalPath(\{d_{i+1}, ..., d_j\}, n-1), optimalPath(\{d_{i+1}, ..., d_j\}, n)) & \text{if } n > 0 \\ \text{Greedy solution from part a, minimizing stops on remaining part } d_i \text{ to } d_n & \text{if } n = 0 \end{cases}$$

again, where d_j is the farthest reachable dock from d_i with a full tank of gas.

Now, we note that we also must take into account the fact that in the cases where we have no product left to sell, our objective is to simply make as few stops as possible as to reduce docking fees, f_i, incurred when purchasing gas in order to make the remaining distance to dock d_n. So really, once we reach cases of $n = 0$, we should follow the algorithm specified in part a when $P - F < 0$. Again, because many of these subproblems overlap, we can cache a solution after calculating a subproblem the first time for subsequent use. We can have as many as K problems per dock, which means a runtime of $O(Kn)$.

(d) As usual, we assume all $d_i \leq D$, as otherwise we could not reach a dock even on a full tank of gas. We solve the problem with dynamic programming as so: at each dock d_i, we have two choices, one of which will be the optimal solution for dock d_i, we choose the solution that will result in spending the least on gas, with each option evaluated recursively. The first option is to buy a full tank of gas at the current dock and then go to the next dock within D that has optimal gas price, and at which point we'll repeat the process. The second option is to buy just enough gas at d_i to get to the next dock, at which point we'll repeat the process. Of course, as this problem has overlapping subproblems, we may cache the solution to a subproblem for later use. This algorithm can be summarized with the following recurrence (where "nextDock" indicates the dock immediately after "currentDock", "gasPerDistance" indicates the constant value n/DT, where DT is the total distance from dock d_0 to d_n, and the rest of the variables follow from their name):

```
max_profits(current_dock, gas_in_boat) =
max(
max_profits(next_dock, total_capacity - (next_dock.location - current_dock.location)*
gas_per_distance) - (total_capacity - gas_in_boat)*current_dock.gas_price,

max_profits(next_dock, 0) - ((next_dock.location - current_dock.location)*
gas_per_distance - gas_in_boat)*current_dock.gas_price
)
```

We find that the runtime for this algorithm is $O(n^2)$ where n is the number of docks.

(e) Again, we assume that all $d_i \leq D$, as otherwise we could not reach a dock even with a full tank of gas. Again, we solve this problem with dynamic programming. We simply (well, not really *simply*) must combine the solutions for each of our previous problems where only certain parameters vary. At every dock, d_i, we must determine if up to this point we have not used all of the product, that is, we still have product to sell, and if we do have product to sell we must decide whether to sell at this dock or hold of from selling (as we did in part c). We must additionally, factor in whether we want to purchase gas at that given dock and how much to purchase (two options, as we did in part d), or abstain from purchasing gas entirely (essentially, just stopping at the dock to sell product). And lastly, we must consider the case where we never stopped at dock d_i at all. Essentially, as we have done in the previous dynamic programming problems, we make every combination of these decisions, at every dock d_i, and retroactively (that is, after calculating each result), choose the optimal decision. Again, because there are overlapping subproblems, we may cache solutions so that we don't have to perform redundant calculations. We can use a recurrence to solve this. The initial call will start at the first dock, d_0, and work its way through each dock calculating each possible action and the consequences of that action, and choosing the optimal option for that dock based on its consequences. This can be summarized in the following algorithm, which does what was described above (note that the

variable notation is identical to part d, and the newly introduced variable "otherNextDock" which is defined essentially as a set of values: all d_j such that d_j is within D of "currentDock"):

```
if n>0
max_profits(current_dock, n, sell, gas_in_boat) =
sell*(current_dock.price) - current_dock.fee +
max(

max_profits(next_dock, n, 0, total_capacity -
(next_dock.location - current_dock.location)* gas_per_distance)
- (total_capacity - gas_in_boat)*current_dock.gas_price,

max_profits(next_dock, n, 0, 0) -
((next_dock.location - current_dock.location)*
gas_per_distance - gas_in_boat)*current_dock.gas_price,

max_profits(next_dock, n, 0, gas_in_boat -
(next_dock.location - current_dock.location)
* gas_per_distance),

max_profits(next_dock, n-1, 1, total_capacity
- (next_dock.location - current_dock.location)* gas_per_distance)
 - (total_capacity - gas_in_boat)*current_dock.gas_price,

max_profits(next_dock, n-1, 1,  (other_next_dock.location -
next_dock.location)*gas_per_distance) - ((other_next_dock.location -
current_dock.location)*gas_per_distance - gas_in_boat)*current_dock.gas_price,

max_profits(next_dock, n-1, 1, gas_in_boat -
(next_dock.location - current_dock.location)* gas_per_distance),

)
else
max_profits(current_dock, 0, 0, gas_in_boat) =
- current_dock.fee +
max(
max_profits(next_dock, 0, 0, total_capacity -
(next_dock.location - current_dock.location)* gas_per_distance)
- (total_capacity - gas_in_boat)*current_dock.gas_price,

max_profits(next_dock, 0, 0, (next_dock.location - next_dock.location)
*gas_per_distance) - ((other_next_dock.location - current_dock.location)
*gas_per_distance - gas_in_boat)*current_dock.gas_price,

max_profits(next_dock, 0, 0, gas_in_boat -
(next_dock.location - current_dock.location)
* gas_per_distance)
)
```

We find that the runtime for this algorithm is $O(Kn^2)$, where n is the number of docks and K is the amount of product.