COSC 302: Analysis of Algorithms — Spring 2018
Prof. Darren Strash
Colgate University

Problem Set 8 — Greedy Algorithms and Dynamic Programming I
Due by 4:30pm Friday, April 6, 2018 as a single pdf via Moodle (either generated via LaTeX, or concatenated photos of your work). Late assignments are not accepted.

This is an *individual* assignment: collaboration (such as discussing problems and brainstorming ideas for solving them) on this assignment is highly encouraged, but the work you submit must be your own. Give information only as a tutor would: ask questions so that your classmate is able to figure out the answer for themselves. It is unacceptable to share any artifacts, such as code and/or write-ups for this assignment. If you work with someone in close collaboration, you must mention your collaborator on your assignment.

Suggested practice problems, from CLRS: 15.1-4, 15.1-5; 15.3-3; 15.3-5; 16.2-1; 16.2-4; 16.2-5

1. **(Worth 10 points)** Problem 16-1 from CLRS.

   (a) Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.

   Solution: In order from highest denomination (quarters) to lowest (pennies), we divide n by the denomination (and take the floor) to determine how many coins of that type to use, then repeat with the next denomination on the remainder.

   Proof. By contradiction. Let O be a minimal set of coins for n cents in change. We prove that our algorithm produces the same set of coins (since the configuration is unique). Suppose our algorithm doesn't compute an optimal solution. Then let d be the highest denomination, such that our solution differs from O. Since we picked the maximum number of coins to use of denomination d, that means the optimal solution has fewer coins of this denomination. Note that d is equal to a sum of two or more lower denomination coins. Therefore, in order for O to contain fewer coins of denomination d, it must contain 2 or more coins of a lower denomination. Therefore, we could remove these 2 (or more) coins and add one of denomination d, and have a new set of coins of size at most $|O| - 2 + 1 = |O| - 1$ therefore, O was not optimal: a contradiction. Therefore, our greedy algorithm is optimal. $\square$

   (b) Suppose that the available coins are in the denominations that are powers of c, i.e., the denominations are c^0, c^1, $\ldots$, c^k for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution.

   Solution:

   Proof. Our argument for (a) applies here. Firstly, note that for any possible n cents, there is a solution of at most n coins of denomination c^0. Furthermore, each coin denomination c^i can be achieved with c coins denomination c^{i-1}. Thus, the same argument in (a) directly translates to this problem. $\square$

(c) Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of n.

Solution: Coins with denominations: $\{14, 13, 3, 1\}$.

For $n = 16$ cents in change, the greedy algorithm will use 3 coins: 14, 1, 1. However, the optimal solution is to use 2 coins: 13, 3.

(d) Give an $O(nk)$-time algorithm that makes change for any set of k different coin denominations, assuming that one of the coins is a penny.

Solution: Let $C[1..k]$ be an array of coin denominations $1 = C[1] < C[2] < \cdots < C[k]$. We give a dynamic programming algorithm with running time $\Theta(kn)$. We first describe a subproblems for this problem. Let the i, j subproblem be the problem of making change for i cents with coin denominations $C[1..j]$. Then the optimal number of coins for n cents in change is the nk subproblem. Let $OPT_{i,j}$ be the minimum number of coins for the ij subproblem. Then we have several cases. If j is 1 then only pennies are allowed and we have that the number of coins is equal to i, the amount of change. If i is 1 then we can only make change with 1 penny, and the solution is 1. Furthermore, either the j-th coin is in the optimal solution or it isn't. If it is, then we count 1 for the j-th coin and consider making the rest of the change with coins $C[1..j]$. If the j-th coin isn't used then we consider the subproblem without the j-th coin. (This is similar to matching the last character or not in the LCS problem). Finally, if i is negative, we aren't able to make change, so we return ∞ to signify an invalid subproblem. We have the following recurrence:

$$OPT_{i,j} = \begin{cases} \infty & \text{if } i < 0, \\ i & \text{if } j = 1, \\ 1 & \text{if } i = 1, \\ \min\{OPT_{i,j-1}, OPT_{i-C_j,j} + 1\} & i > 1, j > 1. \end{cases}$$

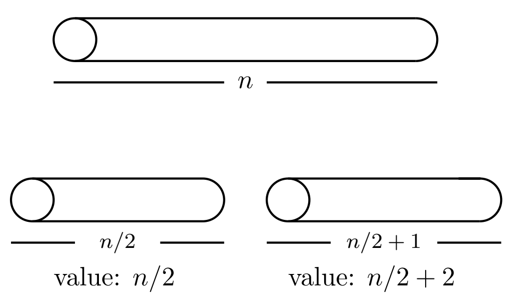
Algorithm 1 Compute fewest coins for n cents.

```
proc Make-Change(n, C[1..k])
OPT[1..n][1..k] = {∞}
for i ← 1 to n do
    OPT[i][1] ← i        ▷ Base case: using only pennies requires i coins for i cents in change.
for j ← 1 to k do
    OPT[1][j] ← 1        ▷ Base case: making change for 1 cent always uses exactly 1 penny.
for i ← 2 to n do                                        ▷ Row by row (amount of change)
    for j ← 2 to k do                                    ▷ Column by column (number of coins)
        OPT[i][j] = OPT[i][j − 1]
        if i ≥ C[j] and OPT[i][j] > OPT[i − C[j]][j] + 1 then
            OPT[i][j] ← OPT[i − C[j]][j] + 1

▷ We now recover a solution by asking, for each entry, whether the j-th coin was used or not.

coins ← []
i ← n
j ← k
while j > 0 and i > 0 do
    if OPT[i][j] = OPT[i][j − 1] then                    ▷ Don't use j-th coin
        j ← j − 1
    else                                                 ▷ Use j-th coin
        i ← i − C[j]
        coins.append(C[j])
return coins
```

2. Problem 15.1-2 from CLRS.

   Solution: Given a rod of total length n, Suppose there are two cut lengths to choose from:

   - Cut 1: $n/2 + 1$, with profit $n/2 + 2$ (density > 1), and
   - Cut 2: $n/2$, with provide $n/2$ (density $= 1$).



   Then the greedy strategy will select Cut 1, which excludes any further cuts, giving total profit $n/2 + 2$. However, we could perform 2 cuts of length $n/2$, with total profit n. Therefore, this greedy strategy does not produce an optimal solution.

3. Problem 15.1-3 from CLRS.

Solution:

Now, in addition to length-profit pairs l_1, p_1 and l_2, p_2, through l_k, p_k, we have a cut cost of c. The subproblems are the same, however we now have a new recurrence, which includes the cost of making a cut.

$$R_j = \begin{cases} -\infty & \text{if } j < 0, \\ 0 & \text{if } j = 0, \\ \max_{1 \le i \le k}\{R_{j-l_i} + p_i - c\} & j > 0. \end{cases}$$

Algorithm 2 Maximize profit for cutting a rod of length n

```
proc CutsToMaximizeProfit(n, L[1..k], P[1..k], c)
R[0..n] = 0
for j ← 1 to n do
    max_profit = 0
    for i ← 1 to k do
        if P[i] ≤ j and max_profit < R[j − L[i]] + P[i] − c then
            max_profit = R[j − L[i]] + P[i] − c
    R[j] = max_profit
return R

▷ We now recover the cuts by asking, for each rod length, which rightmost cut (if any) is made.

cuts ← []
current_profit = R[n]
j ← n
while current_profit > 0 do
    for i ← 1 to k do
        if R[j] = R[j − L[i]] + P[i] − c then
            cuts = cuts.append(i)
            j ← j − L[i]
            current_profit = R[j]
            break
return cuts
```