CS 141, Spring 2019 Homework 5

Problem 1. (25 points)

In the United States, coins are minted with denominations of 1, 5, 10, 25, and 50 cents. Now consider a country whose coins are minted with denominations of $\{d_1, \ldots, d_k\}$ units. They seek an algorithm that will enable them to make change of n units using the minimum number of coins.

1. The greedy algorithm for making change repeatedly uses the biggest coin smaller than the amount to be changed until it is zero. Provide a greedy algorithm for making change of n units using US denominations. Prove its correctness and analyze its time complexity.

2. Show that the greedy algorithm does not always give the minimum number of coins in a country whose denominations are $\{1, 6, 10\}$.

3. Give dynamic programming algorithm that correctly determines the minimum number of coins needed to make change of n units using denominations $\{d_1, \ldots, d_k\}$. Analyze its running time.

Answer: Here is the greedy algorithm.

Inputs: number of units to make change for n

Outputs: number of half dollars, quarter, dimes, nickels, and pennies to use $(c_{50}, c_{25}, c_{10}, c_5, c_1)$.

```
Algorithm MakeChange($n$)
    $c_{50} = n \div 50$
    $n = n \bmod 50$
    $c_{25} = n \div 25$
    $n = n \bmod 25$
    $c_{10} = n \div 10$
    $n = n \bmod 10$
    $c_5 = n \div 5$
    $n = n \bmod 5$
    $c_1 = n$
    return $(c_{50}, c_{25}, c_{10}, c_5, c_1)$
```

Because the algorithm always performs 10 calculations, its worst-case running time is $O(1)$.

Proof of Optimality: Assume that the best non-greedy solution for a given instance of the problem is $(b_{50}, b_{25}, b_{10}, b_5, b_1)$, where $n = 50b_{50} + 25b_{25} + 10b_{10} + 5b_5 + b_1$. We show that the greedy solution is as good as or better than the best solution. The greedy solution is $(c_{50}, c_{25}, c_{10}, c_5, c_1)$. We want to show that $c_{50} + c_{25} + c_{10} + c_5 + c_1 \leq b_{50} + b_{25} + b_{10} + b_5 + b_1$.

Since the best solution is not greedy at some point there will be fewer coins of some denomination in the best solution vs. the greedy solution. We will show that any combination of coins with lower denominations which make up for the difference could be replaced with fewer coins. Therefore, the best solution must be equivalent to the greedy solution.

If $b_{50} < c_{50}$ then $25b_{25} + 10b_{10} + 5b_5 + b_1 \geq 50$. To satisfy the given inequality these are all the possibilities.

1. if $b_{25} \geq 2$, replace with 1 half-dollar

2. if $b_{25} = 1$ we must also have either 2 dimes and 1 nickel, 1 dime and 3 nickels, etc., any of these combinations can be replaced with 1 half-dollar therefore using fewer coins

3. if $b_{25} = 0$ we must also have either 5 dimes, 4 dimes and 2 nickels, etc., any of these combinations can be replaced with 1 half-dollar

If $b_{50} = c_{50}$ and $b_{25} < c_{25}$ then $10b_{10} + 5b_5 + b_1 \geq 25$. These are the possibilities.

1. if $b_{10} \geq 3$, replace with 1 quarter and 1 nickel
2. if $b_{10} = 2$ we must also have either 1 nickels or 5 pennies, all of which can be replaced with 1 quarter
3. if $b_{10} = 1$ we must also have either 3 nickels, 2 nickels and 5 pennies, etc., any of these combinations can be replaced with 1 quarter
4. if $b_{10} = 0$ we must also have either 5 nickels, 5 nickels and 5 pennies, etc., any of these combinations can be replaced with 1 quarter

The entire proof would continue through the case if $b_{50} = c_{50}, b_{25} = c_{25}, b_{10} = c_{10}$, and $b_5 < c_5$.

2) We can show that the greedy algorithm doesn't work for all possible denominations by giving a counter-example. If $n = 12$ and $(d_1, d_2, d_3) = (1, 6, 10)$, then the greedy algorithm would return $(c_{10}, c_6, c_1) = (1, 0, 2)$. However, the optimal solution is $(c_{10}, c_6, c_1) = (0, 2, 0)$.

3) Given a list of k coin values, $(d_1, d_2, \ldots, d_k)$, and a number n, we want to find the integers $(c_{d_1}, c_{d_2}, \ldots, c_{d_k})$ such that $n = \sum_{i=1}^{k} d_i c_{d_i}$ and that $\sum_{i=1}^{k} c_{d_i}$ is minimal.

Our subproblems consist of the optimal change set for 1 through n. To keep track of the optimal solution for each subproblem we will use an array called *sumc* which is indexed by subproblem. (i.e. $sumc[i]$ contains the least number of coins needed to make change for i). $coin[i]$ designates which coin denomination was last used when making change for i units.

$sumc[d_1] = 1$, $sumc[d_2] = 1$, $\ldots$, $sumc[d_k] = 1$

$sumc[i] = \min_{1 \leq j \leq k} sumc[i - d_j] + 1$

Inputs: denominations $(d_1, d_2, \ldots, d_k)$, units n

Outputs: the count of each denomination $(c_{d_1}, c_{d_2}, \ldots, c_{d_k})$.

```
Algorithm MakeChange($n, (d_1, d_2, \ldots, d_k)$)
    for $i \leftarrow 1$ to $n$ do
        $sumc[i] \leftarrow \infty$
    for $j \leftarrow 1$ to $k$ do
        $sumc[d_j] \leftarrow 1; coin[d_j] \leftarrow j$
    // calculate $sumc[i]$ for $1 \leq i \leq n$
    for $i \leftarrow 1$ to $n$ do
        for $j \leftarrow 1$ to $k$ do
            $temp \leftarrow sumc[i - d_j] + 1$
            if $temp < sumc[i]$ then
                $sumc[i] \leftarrow temp$; $coin[i] \leftarrow j$
    // determine if it is possible to make change
    if $sumc[n] = 1$ then return impossible
    else // generate answer
        for $j \leftarrow 1$ to $sumc[n]$ do
            $c_{d_j} = 0$ // initialization
        // traverse through coins used to make best change
        $total \leftarrow n$
        while $total > 0$ do
            $cd_{coin[total]} \leftarrow cd_{coin[total]} + 1$
            $total \leftarrow total - d_{coin[total]}$
        return $(c_{d_1}, c_{d_2}, \ldots, c_{d_k})$
```

The running time of the above algorithm is $O(nk)$. Note that this algorithm is pseudo-polynomial.

Problem 2. (25 points)

Given an array $A = \{a_1, a_2, \ldots, a_n\}$ of integers, we say that a subsequence $\{a_{i_1}, a_{i_2}, \ldots, a_{i_k}\}$ is *(monotonically) increasing* if for every $i_s < i_t$, we have $a_{i_s} < a_{i_t}$. Given an array A of size n, we want to compute the length of the longest increasing subsequence (LIS) in A. For instance, if $A = \{9, 5, 2, 8, 7, 3, 1, 6, 4\}$ the length of the LIS is 3, because $(2, 3, 4)$ (or $(2, 3, 6)$) are LIS of A. Give a $O(n^2)$ dynamic programming algorithm for this problem. Analyze the time- and space-complexity of your solution.

Answer: Define $L(i)$ be the length of the LIS for a prefix $\{a_1, \ldots, a_i\}$ of A such that a_i is the last element in LIS; then $L(i)$ can be recursively written as:

$$L(i) = \begin{cases} 1 & \text{if } i = 1 \\ 1 + \max_{1 \leq j < i}\{L(j) : a_j < a_i\} & \text{otherwise} \end{cases}$$

where we assume that the max of an empty set would return zero.

Time complexity is $O(n^2)$ because it takes linear time to fill each entry of the array (it is possible to decrease the total complexity to $O(n \log n)$, but it is a little more complicated). Space complexity is $O(n)$.

Problem 3. (25 points)

You have a set of n jobs $(a_1, a_2, \ldots, a_n)$ to process on a machine. Each job j has a processing time t_j, a profit p_j and a deadline d_j. The machine can process only one job at a time, and job

j must run uninterruptedly for t_j consecutive units of time. If job j is completed by its deadline d_j, you receive a profit p_j, otherwise a profit of 0. You can assume that all parameters are integers between 1 and n, that $d_j \geq t_j$ and that the jobs are sorted in increasing order of deadline. Give a dynamic programming algorithm to the problem of determining the schedule that gives the maximum amount of profit. Analyze the time- and space-complexity of your solution.

Answer: Recall that the jobs are sorted by increasing deadline. We define $P[i,j]$ to be the maximum profit we can make using jobs $1,\ldots,i$ within time j.

$$P[i,j] = \begin{cases} 0 & \text{if } i = 0 \\ \max(P[i-1,j], P[i-1,j-t_i]+p_i) & \text{if } i > 0 \text{ and } t_i \leq j \leq d_i \\ P[i-1,j] & \text{otherwise} \end{cases}$$

We need to compute j up to n^2 because all processing times $t_j \leq n$. In the worst case we could have all $t_j = n$ which would require at most n^2 time. Therefore, the table P is $n^2 \times n$ thus its size is $O(n^3)$. Filling in each entry in P requires constant time, thus the time complexity is $O(n^3)$.

Problem 4. (25 points)

We are given a list of n items with sizes $s_1, s_2, \ldots, s_n$. A *sequential bin packing* of these items is an assignment of items to bins, such that in each bins the items are consecutive. (That is, each bin has items $s_i, s_{i+1}, \ldots, s_j$ for some indices $i < j$.) Bins have unbounded capacities. The *load* of a bin is the sum of the elements in it. Give an algorithm that determines a sequential packing of n items into k bins for which the maximum load of a bin is minimized. Analyze the time- complexity and space-complexity.

Answer: Let's define $S[i,j]$ be the minimum cost of a bin packing for item $s_1, s_2, \ldots, s_i$ into j bins (where the cost of a bin packing is defined as max load of the bins). We have

$$S[i,j] = \begin{cases} s_1 & i = 1, j = 1 \\ \sum_{l=1}^{i} s_l & i > 1, j = 1 \\ \min_{1 \leq h \leq i-1} \max(S[h, j-1], \sum_{l=h+1}^{i} s_l) & i > 1, j > 1 \end{cases}$$

The S table is $n \times k$. Each entry can be computed in $O(n)$ time if the prefix sums of the s_i's are available. Let's define $A[i] = \sum_{j=1}^{i} s_j$. Array A can be computed in $O(n)$. Then we can compute S as follows

$$S[i,j] = \begin{cases} s_1 & i = 1, j = 1 \\ A[i] & i > 1, j = 1 \\ \min_{1 \leq h \leq i-1} \max(S[h, j-1], A[i] - A[h]) & i > 1, j > 1 \end{cases}$$

Therefore the time required to compute S is $O(kn^2)$. Space is $O(nk)$.