CS 141 homework5

Celyna Su

TOTAL POINTS

79.5 / 100

QUESTION 1

1 Problem 1 25 / 25

- ✓ **- 0 pts** Correct
- **- 5 pts** No greedy algorithm for part 1
- **- 4 pts** No proof of correctness for part 1
- **- 1 pts** No/Incorrect time complexity for part 1
- **- 3 pts** Incorrect/Incomplete greedy algorithm for part 1
- **- 2 pts** Incorrect proof of correctness for part 1
- **- 5 pts** No solution for part 2
- **- 3 pts** Incorrect solution for part 2
- **- 10 pts** No solution for part 3
- **- 10 pts** No solution for part 1
- **- 2 pts** No/Incorrect time complexity for part 3
- **- 5 pts** Incorrect/Incomplete algorithm for part 3
- **- 2 pts** There is a more efficient algorithm.

QUESTION 2

2 Problem 2 20 / 25

- ✓ **+ 15 pts** Explains O(n^2) (or faster) dynamic programming algorithm.
- ✓ **+ 5 pts** Correctly analyzes O(n^2) time complexity of algorithm.
- **+ 5 pts** Correctly analyzes space complexity of given algorithm.
- **+ 12 pts** Insufficiently explains dynamic programming algorithm.
- **+ 4 pts** Insufficiently analyzes time complexity of algorithm.
- **+ 4 pts** Insufficiently analyzes space complexity of given algorithm.
- **+ 5 pts** Incorrect algorithm
- **+ 0 pts** Mission solution

QUESTION 3

3 Problem 3 20 / 25

- **- 0 pts** Correct
- ✓ **- 2 pts** No/Incorrect time complexity
- **- 25 pts** No solution
- **- 10 pts** Incorrect relation
- **- 8 pts** Explanation provided but no recurrence relation provided.
- **- 5 pts** Unclear recurrence relation
- ✓ **- 3 pts** Missing time component
- **- 2 pts** No/Incorrect space complexity
- **- 3 pts** Condition is not right

QUESTION 4

4 Problem 4 14.5 / 25

- **- 0 pts** Correct
- **- 20 pts** Incorrect Algorithm
- **- 25 pts** No attempt
- ✓ **- 8 pts** Recursion formula not shown / Incorrect
- **- 2.5 pts** Space complexity incorrect / not shown
- ✓ **- 2.5 pts** Not shown / Incorrect Time complexity
- **- 7 pts** One case not handled

CS 141, Spring 2019 Homework 5

Posted: May 9th, 2019 Due: May 16th, 2019

Name: Celyna Su

Student ID #: 862037643

- You are expected to work on this assignment on your own
- Use pseudocode, Python-like or English to describe your algorithms. Absolutely no C++/C/Java
- When designing an algorithm, you are allowed to use any algorithm or data structure we explained in class, without giving its details, unless the question specifically requires that you give such details
- Always remember to analyze the time complexity of your algorithms
- Homework has to be submitted electronically on Gradescope by the deadline. No late assignments will be accepted

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:
(Credit to UCSD for guiding me to the answer)

1. This greedy algorithm takes in the **number of units to generate change for** n and outputs the number of fifty-cents (denoted c_{50}), quarters (denoted c_{25}), dimes (denoted c_{10}), nickels (denoted c_5), and pennies (denoted c_1) to use.
   1. $c_{50} = n \div 50$
   2. change $= n$ % 50
   3. $c_{25} = n \div 25$
   4. change $= n$ % 25
   5. $c_{10} = n \div 10$
   6. change $= n$ % 10
   7. $c_5 = n \div 5$
   8. change $= n$ % 5
   9. $c_1 = n \div 1$
   10. change $= n$ % 1
   11. return ($c_{50}\, c_{25}\, c_{10}\, c_5\, c_1$

   Runtime: $\Theta(1)$ because the above algorithm always performs 10 calculations.

   Proof of Optimality: Assume that the ideal non-greedy solution, denoted by a_{50}, a_{25}, a_{10}, a_5, c_1 where $n = 50a_{50}+25a_{25}+10a_{10}+5a_5+1a_1$ WTS that the greedy solution is either equal to, or more optimal than the best solution, or $50c_{50}+25c_{25}+10c_{10}+5c_5+1c_1 \leq 50a_{50} + 25a_{25} + 10a_{10} + 5a_5 + 1a_1$.
   Because the best solution is not greedy at some point, there will be a varying amount of coins (i.e there will be fewer coins of some denomination in the best solution compared to the greedy solution. However, any combo of coins with lower denominations which

make up for the difference can be replaced with fewer coins. This means that the best solution must be equal to the greedy solution.
If $a_{50} < c_{50}$ then $25a_{25} + 10a_{10} + 5a_5 + 1a_1 \geq 50$, which means a_{25} can be ≥ 2. If that is the case, then 2 quarters could be replaced with a half-dollar (so there are less coins used).
1. If $a_{25} \geq 2$, replace with 1 half-dollar
2. If $a_{25} = 1$, we can use a half dollar to use fewer coins rather than using combinations of either 1 dime and 3 nickels, 5 nickels, etc.
3. If $a_{25} = 0$ we can also use a half dollar to use fewer coins rather than using 5 dimes, 4 times and 2 nickels, etc.

If $a_{50} = c_{50}$ and $a_{25} < c_{25}$ then $10a_{10} + 5a_5 + 1a_1 \geq 25$.
1. If $a_{10} \geq 3$, replace with 1 quarter and 1 nickel
2. If $a_{10} = 2$ we can replace with 1 quarter rather than having either 1 nickel or 5 pennies.
3. If $a_{10} = 1$ we can replace with 1 quarter rather than having 3 nickels, 2 nickels and 5 pennis, etc. 4. If $a_{10} = 0$ we can replace with 1 quarter rather than having 5 nickels, 5 nickels and 5 pennies, etc.

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

2. We can prove that the greedy algorithm doesn't work for all possible denominations because for 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. We want to find the integers $(c_{d_1}, c_{d_2}, ..., c_{d_k})$ s.t $n = \sum_{i=1}^{k} d_i c_{d_i} = 1$ and that $\sum_{i=1}^{k} c_{d_i}$ is minimal.

   In this algorithm, we will use an array called $sum[]$ which will contain the least number of coins needed to make change for the index. $coin[]$ shows which coin denomination was last used when making change for the indexed amount of units (i).

   1. // initialize variables
   2. for $i = 1$ to n
   3. $sum[i] = \infty$
   4. for $j = 1$ to k
   5. $sum[d_j] = 1$; $coin[d_j] = j$
   6. for $i = 1$ to n

7. for $j = 1$ to k
8. $temp = sum[i - d_j] + 1$
9. if $temp < sum[i]$
10. $sum[i] = temp; coin[i] = j$
11. if $sum[n] = \infty$ return "failed; not possible"
12. else
13. for $j = 1$ to $sum[n]$
14. $c_{d_j} = 0$ //init
15. traverse through coins used so you can make change optimally
16. total = n
17. while $total > 0$
18. $c_{d_{coin[total]}} = c_{d_{coin[total]}} + 1$
19. total = total - $d_{coin[total]}$
20. return $(c_{d_1}, c_{d_2}, ..., c_{d_k}$

The runtime is $O(nk)$. The base case is $coin[0] = 0$. For each n we need $O(k)$ tests, therefore we need, in the worst case, $O(kn)$ tests overall.

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:
$A[0..n-1]$ is the input array.
$length[0...n-1]$ is the dynamic programming array that stores the length of the LIS ending at each index

1.
declare length array, initialize first index as 2. length[0] = 1 3.
calculate the length array
4. for $i = 1$ to n
5. length[i] = 1
6. for $(j = 0)$ to i
7.
check if A[j] is worth considering or not for i
8. if A[i] ¡ A[j] and length[i] ¡ length[j]+1
9. length[i] = length[j] + 1
10. finally, this length array has LIS ending at each index

The above algorithm is $O(n^2)$ runtime. Although we may be able to solve it in $O(n \log n)$ time, but we can't use it here because we want to find LIS ending at each array element and that will give us LIS for the entire array.

Problem 3. (25 points)

You have a set of n jobs 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, 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:
Input: $n, s_1, ..., s_n, f_1, ..., f_n, v_1, ..., v_n$
1. Sort jobs by finish times so $f_1 \leq f_2 \leq ... \leq f_n$
2. Compute $p(1), p(2), ..., p(n)$
3. $OPT[0] = 0$
4. for j = 1 to n
5. $OPT[j] = max(v_j + OPT[p(j)], OPT[j-1]$
6. Output $OPT[n]$

The main loop is running n number of times which means $O(n)$ computing time, and sorting takes $O(\log n)$ of computation time. Therefore, the overall runtime is $O(n \log n)$. The space complexity $= O(n)$ because we are storing the number of repetitive recursive calls in the array denoted OPT.

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: 1. If n is $\leq$ number of bins k:
2. assign 1 item to each bin until you are out of items
3. Else if $n \geq 2k$
4. assign 2 bins each and as soon as $n - k$ bins are remaining, start assigning 1 item to each bin
5. Else: assign $\frac{n}{k}$ items to each bin

The time-complexity is $O(kn)$ because you are putting n items into k number of bins. The space-complexity is $O(k)$ because you are packing into k number of bins.