CP312: Algorithm Design & Analysis

Assignment #4

CP312, WLU, 2022

Instructor: Masoomeh Rudafshani

Due: Friday, March 4th, 2022

Declan Hollingworth
190765210
holl5210@mylaurier.ca

Nishant Tewari
190684430
tewa4430@mylaurier.ca

Q1. In the topic of Greedy Algorithms, we proved the optimality of the greedy algorithm for the coin change problem when the US. coin denomination was used. Why does that proof not work for the U.S. post office denomination?

A great example of this problem was discussed in class for the change total of $1.40 (140¢)

U.S Coin Denominations: 1, 5, 10, 25, 100
Solving using U.S. Denomination:
140¢ = 100¢ + 25¢ + 10¢ + 5¢ (Optimal)

When using the cashier's algorithm with U.S. denominations, the algorithm is effective because there is no single denomination that, when doubled, is greater than that of the next greatest denomination.

U.S. Post Office Denominations: 1, 10, 21, 34, 70, 100, 350, 1225, 1500
Solving using U.S. Postage Denomination:
140¢ = 100¢ + 34¢ + 1¢ + 1¢ + 1¢ + 1¢ + 1¢ + 1¢ (Not Optimal)
The optimal solution would be…
140¢ = 70¢ + 70¢

When using the cashier's algorithm with other denominations such as the U.S Postage, we may run into issues with certain change totals since some of the denominations, when doubled, are greater than the denomination of the next greatest value

We can represent this problem with some arbitrary equation. Let d[i] represent some denomination i in an ascending set of d denominations. Cashiers algorithm will not hold true if the following inequality is true.

2*d[i] > d[i+1]

In summary, the cashiers algorithm is only optimal when there is no denomination that is greater, when doubled, than the next highest denomination

Q2. In the topic of greedy algorithms, we solved the following problem: Scheduling to minimize lateness. Prove that this problem has the optimal substructure property.
Note: We talked about proving optimal substructure properties when talking about dynamic programming. You can use the technique discussed in dynamic programming here.

We can use an algorithm for the scheduling to minimize lateness problems. Algorithm will contain the following:

- List for the times of the jobs
- List for the deadlines of the jobs
- Sort the jobs (increasing order)
- Set time to 0 to start at the job with the least time for completion
- Iterate through the jobs and set t equal to "start time (Si) + completion time (Fi) "

As we iterate through the jobs, we can see that each substructure is optimized to ensure minimum lateness is scheduled to each job. Therefore, the entire algorithm ensures the use of an optimal solution as the substructures are optimized as well.

Q3. Can we solve the above problem (scheduling to minimize lateness) using dynamic programming. If yes, how? If not, why?

To know when to use Dynamic Programming, the problem must include optimal substructure or overlapping subproblems. Therefore, question 2 can be solved using dynamic programming because of the inclusion of both subproblems.

Optimal Substructure - Question 2 has been proven to have an optimal solution as we iterate through the jobs, we can see that each substructure is optimized to ensure minimum lateness is scheduled to each job.

Overlapping Subproblems - Question 2 has an overlapping subproblem where the completion of the next job will be dependent on the completion of the jobs before.

Q4. Write the pseudocode for the first programming question described below and prove the optimality of the problem. You need to develop the pseudocode, although the most part of the points goes to proving the optimality of your solution.

In order to generate the maximum sum of values, you need to subtract the smallest values in the list. To do so, you can sort the list then use the required negative operations on the smallest values to minimize how much you subtract. This method takes O(n log n) time. One alternative solution to this problem would be comparing values in the list to one another to fund the smallest value, however the runtime for that method would have an exponential runtime.

```
Def maxValueSeq(values, p, n)
    values.sort() # O(n logn)
    sum =  0
    maxSeq = []

    for item in values: # O(n)
        # Negative signs
        if(n > 0):
            sum -= item # Subtract smallest values
            maxSeq.append("-")
            n -= 1
            maxSeq.append(item)
        # Positive Signs
        else:
            sum += item
            maxSeq.append("+")
            p -= 1
            maxSeq.append(item)

    return sum, maxSeq
```

Q5. Design and develop the pseudo-code for the problem described in the second programming question below. Write the pseudo code using memoized technique, bottom-up technique and analyze the time complexity of your algorithm in both bottom-up and memoized versions.

```
def packageOptimize(values, sum)
    values.reverse()
    for item in values: # O(n)
        if(sum % item == 0):
            print(sum // item)
            print(sum // item, "packages of size", item)
            exit
```