Saleh Alkhalifa CS 5800
11/8/2022 Module 8

Homework #8

Problem 1 Maximum Difference (10 pts)

Given an array of numbers $x_1,\ldots,x_n$ we are interested in finding
$D=\max(x_j - x_i)$ where $1 \le i \le j \le n$

Describe an efficient algorithm that calculates *D*. In addition to describing the algorithm, explain the efficiency of your algorithm clearly.

- There are a number of different methods you can use here to calculate D
- Let us look over one method in which we take advantage of the ability to save and remember some of our information in the process, thus relying not only on time complexity but space complexity as well
- Let us assume we have an array A, and assume the array is ordered in an ascending fashion. We can use the following algorithm to calculate the maximum difference.
- Algorithm:
    - First, we instantiate an integer variable which we can call A_{max} representing the maximum difference found thus far.
    - Normally what can do here is iterate and recalculate the value in each interval. Instead, we can iterate of the array via a loop, and rely on the stored value.
    - At each iteration, we will compare A[i] - A[j] with the current value
    - As we iterate over the values, we can now update D as needed
- When it comes to time complexity, since we iterate over the array in a linear fashion, and compare values in a linear fashion, its safe to assume that the algorithm exhibits a linear nature giving us O(n) time complexity. Note that order is assumed here.

Problem 2 Minimum Number of Coins (10 pts)

Given is a list of *K* distinct coin denominations *($V_1,\ldots,V_k$)* and the total sum *S>0*. Find the minimum number of coins whose sum is equal to *S* (we can use as many coins of one type as we want), or report that it's not possible to select coins in such a way that they sum up to *S*. Justify your explanation

- Given the list of K distinct coin denominations, let us use a dynamic programming method to determine the minimum number of coins with a sum equal to S.
- This question is an example of the Knapsack problem discussed in lecture
- Since we need to keep the values within the K distinct denominations, we will need to account for that as well.
- Given the dynamic nature of the problem, we can dissect our problem into subproblems.
- Ultimately, for each coin j, we will then determine the minimum for $i-V_j$ describing the previous value
- Eventually, if this value is less than the minimum number of coins we found, then we can update the sum S

- Algorithm:
    - Instantiate a variable sum which we can set to 0
    - Looking at sum+1, giving us a value of 1. Given the k distinct denominations, we check to see if any of the coins are less than or equal to this value.
    - If so, we can note the coin and update the sum accordingly.
    - Iteratively, we can continue this process for a sum of 2 and checking again. If we have coin values of 1 and 5 for example, then an optimal solution would now be two coins of value 1. Or, if we had coins of values 1 and 2, the optimal solution would be one coin of value 2.
    - We can continue this dynamic programming process over and over by taking advantage of the knowledge of the previous step.
    - Finally, we return the coins required to make up S.

Homework #8

Problem 3 Consecutive sums (5 + 5 = 10 pts)

Let (a_1,...a_n) be a sequence of distinct numbers some of which may be negative. For 1≤i≤j≤n, consider the sum $S_{ij}=a_i+....+a_j$

a) What is the running time of a brute force algorithm to calculate *max S_{ij}*?

- A brute force algorithm in this case would iterate across all possibilities and determine the final sum as represented by $S_{ij} = a_i + \cdots + a_j$
- Ultimately, this algorithm would comprise two variables i, and j, representing the ends of the array of numbers mentioned above
- This algorithm would also comprise two loops, the first iterating across these two variables and another to calculate the sums
- Ultimately, this would allow for the maximum value of $\max(S_{ij})$ to be calculated.
- Because of this, we will see that the running time or time complexity will be $O(n^3)$

b) Give an efficient algorithm to find the above maximum. In addition to giving the algorithm, describe the efficiency of your algorithm clearly.

- Given the dynamic nature of the answer to a question like this, we can use this to break down the answer into sub problems.
    - Let us take for example a given array, which as described may contain negative values.
    - First we instantiate a variable which we can call max to keep track of the maximum value
    - We can then create a summed array in which the values represent the sums of the previous two elements. For example, given an array A = {-1, 3, 5}, we would yield A_{sum}= {-1, 2, 7} since -1 + 3 gives us 2, and 2+5 gives us 7.
    - We can now begin to iterate over A_{sum} starting from index position 1
    - From there we can calculate the maximum. If the maximum is found to be greater, we can update max variable
- When it comes to time complexity, there are two main items we need to consider that went into this algorithm. First, we created A_{sum} which can be represented by O(n) given the linear nature of the calculation. Second, we compared values to determine the maximum value which is also O(n). Therefore, the total run time for this algorithm is likely to be O(n) given the linear nature of it.

References:

[1] https://northeastern.instructure.com/courses/117409/pages/module-8/module_item_id=7833155

[2] https://en.wikipedia.org/wiki/Knapsack_problem

[3] Introduction to Algorithms, Cormen, Third Edition. (CLRS)