Algorithms: Assignment #8

Tanvi Magdum
magdum.t@northeastern.edu

March 4, 2023

Answers

1 Problem 1 Solution :

An efficient algorithm to solve this can be -

1. Take 2 variables min and max_diff and initialize both to the first element of array.
2. Initialize a boolean variable to false to keep track of whether there is max_diff greater than zero.
3. Iterate through the array from second element onwards.
   (a) If the current element is smaller than min, update min with that element.
   (b) If the difference between current element and min is greater than max_diff, then update the max_diff and set flag to true.
4. If flag is false then set max_diff to 0 as in this case, all elements of the array are same.
5. Return the max_diff.

The goal of this algorithm is to find the maximum difference between any 2 elements in the array. The approach used here is to keep the track of minimum element and update the maximum difference whenever a larger difference is found. It iterates through the array and updates these variables accordingly. At the end it returns the maximum difference.

```
Code :

arr : input array
min : minimum of all the elements of array
max_diff : maximum different in two elements of an array
flag : boolean to maintain the track whether the max_diff is 0.

    static int calcMaxDiff(int[] arr) {
        int min = arr[0];
        int max_diff = arr[0];
        int n = arr.length;
        boolean flag = false;

        for (int i = 1; i < n; i++) {
            if (arr[i] < min) {
                min = arr[i];
            }
            if ((arr[i] - min) > max_diff) {
                max_diff = arr[i] - min;
                flag = true;
            }
        }

        if (!flag) {
            max_diff = 0;
        }
        return max_diff;
    }
```

The time complexity of this algorithm is $O(n)$ because it iterates through the array only once. The space complexity is $O(1)$ because it uses only 3 variables to store the minimum and maximum values and a flag to keep a track whether there is a difference of zero. Hence, algorithm is both time and space efficient.

2 Problem 2 Solution :

We have to find the number of coins required to get a sum of S, when we are provided with a list of coin denominations. We can use dynamic programming for this. We can create an array of size S+1, where the ith element of the array represents the minimum number of coins required to get sum i. Let first element of array be 0, since we require 0 coins to get sum 0. For each denomination, we iterate through the array from coin value to S and update the array like - we take the minimum of the current value, and the value at index of array minus the coin value plus 1. It represents the minimum number of coins required to get the current sum using the current coin denomination. We will repeat this process for each coin denomination. Finally, the value at the Sth index of array represents the minimum number of coins required to get sum S. If this value is still equal to infinity, which we initialized, then it is not possible to get S using given denominations.

Algorithm

1. Initialize an array of size S+1 with all the elements set to infinity, except the first element which is set to 0. This array will store the minimum number of coins required to get sum i, where i is the index of array "min_coins".
2. For each denomination in the list, iterate through the array from the value of denomination to S.
   (a) Take a variable coin to store current coin denomination.
   (b) For i from coin to S, update the min_coins[i] as- min_coins[i] = min(min_coins[i], min_coins[i-coin] + 1)
   (c) min_coins[i-coin] + 1 is the minimum number of coins required to get (i-coin) using the current coin denomination plus the current coin.
3. The min_coins[S] now represents the minimum number of coins required to get sum S using given denominations.
4. If the value of min_coins[S] is still infinity, then it is not possible to get sum S using given denominations.

For example, we have a list of denominations {1,5,10} and target sum S is 12. We will follow the steps below:

1. Initialize the array as : min_coins = {0,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf,inf}
2. For the first coin denomination 1, iterate through the array from 1 to 12.
   (a) min_coins[1] = min(min_coins[1], min_coins[0] + 1) = min(inf, 0+1) = 1
   (b) min_coins[2] = min(min_coins[2], min_coins[1] + 1) = min(inf, 1+1) = 2
   (c) min_coins[3] = min(min_coins[3], min_coins[2] + 1) = min(inf, 2+1) = 3
   (d) ......
   (e) min_coins[12] = min(min_coins[12], min_coins[11] + 1) = min(inf, 11+1) = 12

3. For the second coin denomination 5, iterate through the array from 5 to 12.
    (a) min_coins[5] = min(min_coins[5], min_coins[0] + 1) = min(5, 0+1) = 1
    (b) min_coins[6] = min(min_coins[6], min_coins[1] + 1) = min(6, 1+1) = 2
    (c) min_coins[7] = min(min_coins[7], min_coins[2] + 1) = min(7, 2+1) = 3
    (d) min_coins[8] = min(min_coins[8], min_coins[3] + 1) = min(7, 3+1) = 4
    (e) min_coins[9] = min(min_coins[9], min_coins[4] + 1) = min(7, 4+1) = 5
    (f) min_coins[10] = min(min_coins[10], min_coins[5] + 1) = min(7, 1+1) = 2
    (g) min_coins[11] = min(min_coins[11], min_coins[6] + 1) = min(7, 2+1) = 3
    (h) min_coins[12] = min(min_coins[12], min_coins[7] + 1) = min(12, 3+1) = 4
4. For the third coin denomination 10, iterate through the array from 10 to 12.
    (a) min_coins[10] = min(min_coins[10], min_coins[0] + 1) = min(2, 0+1) = 1
    (b) min_coins[11] = min(min_coins[11], min_coins[1] + 1) = min(3, 1+1) = 2
    (c) min_coins[12] = min(min_coins[12], min_coins[2] + 1) = min(4, 2+1) = 3
5. The final value of min_coins[12] is 3, which means we can get the sum 12 using the coin denominations [1, 5, 10] with a minimum of 3 coins.

The outer loop runs n times, where n is number of coin denominations and the inner loop runs (S-coin) times for each coin denomination. Hence, the time complexity of this algorithm is $O(ns)$. The space complexity is also $O(s)$ as we just need to store an array of size (S+1). Hence, the algorithm is both time and space efficient.

3 Problem 3 Solution :

3.1 Problem 3.1 Solution :

A brute force algorithm to calculate the maximum sum S_{ij} would consist of calculating the sum of each possible subsequence and selecting the maximum sum among them. Suppose that we have a sequence of numbers - {3,7,-2,1,5}. We need to find a subsequence with highest sum of elements. In this case the such subsequence is actually the whole sequence and the sum is 14.

To find the subsequence with largest sum, we have to consider all the possible subsequences and calculate their sums. Starting with {3}, then {3,7}, then {3,7,-2} and so on we have to calculate sum for each sequence. In our case, there would be total 15 subsequences. To calculate sum, for {3} it is 3, {3,7} it is 10, {3,7,-2} it is 8 and so on. We will keep track of largest sum we have so far and update it whenever we find a larger sum. The maximum sum would be for {3,7,-2,1,5} which is 14.

Algorithm -

1. Initialize two variables max_sum to the first element of array.
2. Iterate through the whole sequence using i from 1 to n.
    (a) Iterate through the subsequence using j from i to n.
    (b) Take a variable current_sum, assign it to 0.
        i. Iterate between i to j.
        ii. Compute the sum of elements between i to j and store it into current_sum.
    (c) If current_sum is greater than max_sum, then update max_sum to current_sum.
3. Return max_sum.

This brute force algorithm would work, but it would get slow for large sequences. If we have a sequence of n elements, then there are n*(n+1)/2 possible subsequences. For instance, if we have a sequence of length 100, there would be total 5050 subsequences to consider. Calculation of sum of each subsequence takes at most $O(n)$ time, so the total time complexity of brute force algorithm will be $O(n^3)$, as there are total $O(n^2)$ possible pairs (i,j) and each pair requires at most $O(n)$ to calculate the sum. The total time complexity of the brute force algorithm can be calculated as follows:

sum over all pairs (i,j) of Sij

= sum over all i from 1 to n of sum over all j from i to n of S_{ij}.

= sum over all i from 1 to n of sum over all j from i to n of (j-i+1).

We consider all possible pairs (i,j) such that i <= j. For a sequence of length n there are n possible values of i as i = 1,2,3...,n and for each value of i, there are n-i+1 possible values of j as j = i, i+1,..n. Hence, total number of pairs (i,j) that we have to consider is sum over all possible values of i of j, which is

sum over all i from 1 to n of sum over all j

= sum over all i from 1 to n of n - sum over all i from 1 to n of i + n

$= n^2 - (n * (n+1)/2) + n$

$= (n^2 + n)/2$

$= n(n+1)/2$

Therefore, the total number of pairs (i, j) is $O(n^2)$.

The range [i,j] is a subsequence of original range and it can be at max n as we have seen in the example. So, in worst cases the the time complexity of computing sum can be as high as $O(n)$. Therefore, the time complexity of the brute force algorithm is $O(n^3)$. The algorithm takes $O(n^2)$ time to generate all pairs (i,j) and $O(n)$ time to compute the sum of each pair.

3.2 Problem 3.2 Solution :

The maximum sum subsequence problem can be solved efficiently using the following algorithm -

1. Initialize max_sum to the minimum possible integer value.
2. Initialize current_sum to 0.
3. Loop through all elements of sequence.
   (a) Add the current element to current_sum.
   (b) Update max_sum if current_sum is greater than max_sum.
   (c) If current_sum becomes negative, reset it to 0. (This means that we have found a subsequence that has negative sum, so we can discard it and start over from next element.)
4. Return max_sum

This algorithm is efficient because, it goes through the sequence just once and keeps track of maximum subsequence sum so far and the current subsequence sum. The maximum sum is updated whenever the current sum exceeds maximum sum. This means that we don't have to consider all possible subsequence sums. We just need to consider those which are relevant to current subsequence sum. This makes this particular algorithm faster than the brute force algorithm, especially for large sequences.

```
Psuedocode :

max_sum: maximum subsequence sum so far
current_sum: current subsequence sum
```

```
Max_Sum_Array(arr):
max_sum = Integer.MIN_VALUE
current_sum = 0
for i=0 to arr.length:
    current_sum += arr[i]
    if current_sum > max_sum:
        max_sum = current_sum
    if current_sum < 0:
        current_sum = 0
return max_sum
```

To calculate the time complexity, we count the number of operations performed by algorithm. We are only performing constant number of operations for each element of this sequence. So, the total number of operations is proportional to the length of the sequence. The time complexity of this algorithm is $O(n)$, where n is the length of the sequence as we are looping through the sequence just once. This means that the algorithm is optimal.