Assignment 8

Problem 1: Maximum Difference

Given an array of numbers $x_1,...,x_n$ we are interested in finding

$D = \max\ (x_j - x_i)$ where $1 \leq i \leq j \leq n$

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

Answer:

A two-step strategy can be used to find the largest difference between two elements in an array. First, we locate the smallest element in the array, and then we find the largest element in the array after that. The difference between these two aspects will provide us with the greatest difference.

Here's the algorithm:

1. Initialize two variables, min_element, and max_diff, to keep track of the minimum element encountered so far and the maximum difference found.
2. Iterate through the array from left to right:
    - If the current element is less than min_element, update min_element to the current element.
    - If the current element and min_element difference is greater than max_diff, update max_diff to reflect the difference.
3. Return max_diff.

Here is the code for the algorithm

```
def find_max_difference(arr):
    n = len(arr)
    if n < 2:
        return 0

    min_element = arr[0]
    max_diff = arr[1] - arr[0]

    for i in range(1, n):
```

```
        if arr[i] - min_element > max_diff:
            max_diff = arr[i] - min_element
        if arr[i] < min_element:
            min_element = arr[i]

    return max_diff
```

The complexity of this algorithm is O(n), where n is the size of the input array. The algorithm performs a single pass through the array, comparing each element with the current minimum element and updating the maximum difference if necessary. Therefore, the time complexity is linear, making it an efficient solution for the problem.

In cases where the array has fewer than two elements (n < 2), the function returns 0 as there are not enough elements to compute the difference.

Problem 2: Minimum Number of Coins

Given is a list of *K* distinct coin denominations *(V1,...,Vk)* 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

Answer:

This problem can be solved efficiently using dynamic programming. Here's the algorithm:

1. Initialize an array, min_coins, of size S+1 with infinity (or a large value) except for min_coins[0], which is set to 0. The min_coins[i] will represent the minimum number of coins needed to make the sum i.
2. Iterate through the denominations Vi from 1 to K:
    - For each denomination Vi, iterate through the sums from Vi to S:
        - Update min_coins[sum] as min(min_coins[sum], min_coins[sum - Vi] + 1).
3. After completing the iterations, if min_coins[S] is still infinity, it means that it's not possible to select coins to sum up to S. Otherwise, min_coins[S] will give the minimum number of coins needed.

Let's break down the algorithm and explain its efficiency:

The algorithm uses dynamic programming to build a bottom-up solution by iteratively computing the minimum number of coins needed for each sum from 0 to S. It does this by considering each denomination and updating the minimum number of coins needed for each sum accordingly.

The time complexity of this algorithm is O(K*S) because we have nested loops iterating through the denominations (K) and the sums (S). Since both K and S can vary, the overall time complexity is proportional to K multiplied by S.

The space complexity of this algorithm is O(S), as we only need to maintain the min_coins array of size S+1 to store the minimum number of coins needed for each sum. The space requirement is linear with respect to the value of S.

Overall, this dynamic programming approach provides an efficient solution to the problem by leveraging the overlapping subproblem property and can handle larger values of S and larger sets of coin denominations.

Here's the code for finding the minimum number of coins needed to make a sum of S:

```
def find_min_coins(coins, S):
    min_coins = [float('inf')] * (S + 1)
    min_coins[0] = 0

    for coin in coins:
        for i in range(coin, S + 1):
            min_coins[i] = min(min_coins[i], min_coins[i - coin] + 1)

    if min_coins[S] == float('inf'):
        return -1  # Not possible to make the sum S
    else:
        return min_coins[S]
```

You can use this function by passing a list of coin denominations and the desired sum as arguments. The function will return the minimum number of coins needed to make the sum S. If it's not possible to make the sum, it will return -1.

Here's an example usage of the function:

```
coins = [1, 2, 5]
S = 11

min_num_coins = find_min_coins(coins, S)
print(min_num_coins)  # Output: 3
```

In this example, we have coins with denominations [1, 2, 5], and we want to make a sum of 11. The function returns 3, indicating that we need a minimum of 3 coins (two coins with denomination 5 and one coin with denomination 1) to make the sum 11.

Problem 3: Consecutive sums

Let $(a_1,...a_n)$ be a sequence of distinct numbers some of which maybe negative. For $1 \leq i \leq j \leq n$, 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}?

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

Answer:

a) The brute force algorithm to calculate the maximum sum Sij would involve trying all possible combinations of indices i and j and computing the corresponding sums. The total number of sums to compute is given by the sum of integers from 1 to n, which can be represented as (n*(n+1))/2. This is because there are n possibilities for i and n-i choices for j (where i ranges from 1 to n).

Therefore, the running time of the brute force algorithm would be O(n^2) because we need to compute O(n^2) sums in the worst case scenario.

b) To find the maximum sum Sij efficiently, we can use a dynamic programming approach. Here's the algorithm:

1. Initialize two variables, max_sum and current_sum, to keep track of the maximum sum encountered so far and the current sum of the subsequence.
2. Iterate through the sequence from left to right:
    - Add the current element to the current_sum.
    - If the current_sum is greater than max_sum, update max_sum to current_sum.
    - If the current_sum becomes negative, reset it to 0 (start a new sequence).
3. Return max_sum.

The efficiency of this algorithm lies in the fact that it uses a single pass through the sequence, updating the current sum and the maximum sum as it goes. The algorithm

does not require nested loops or trying all possible combinations, resulting in a linear time complexity.

The time complexity of this algorithm is O(n) because we iterate through the sequence once, performing constant time operations at each step. Therefore, the algorithm is efficient as it solves the problem in linear time.

Note: If all the numbers in the sequence are negative, the algorithm will return 0, indicating an empty subsequence.

here is the code for the algorithm

```
def find_max_sum(sequence):
    n = len(sequence)
    max_sum = float('-inf')
    current_sum = 0

    for i in range(n):
        current_sum += sequence[i]

        if current_sum > max_sum:
            max_sum = current_sum

        if current_sum < 0:
            current_sum = 0

    return max_sum
```

You can use this function by passing the sequence of numbers as an argument. The function will return the maximum sum Sij.

Here's an example usage of the function:

```
sequence = [1, -2, 3, 5, -3, 2]
max_sum = find_max_sum(sequence)
print(max_sum)  # Output: 8
```

In this example, the sequence is [1, -2, 3, 5, -3, 2]. The function returns 8, which is the maximum sum Sij that can be obtained from any subsequence of the given sequence.