CS 430 – Fall 2024
Introduction To Algorithms
Homework #5

**1a)**

```
function MaxSubarraySum(arr, n):
    max_sum = 0
    for i from 1 to n:
        current_sum = 0
        for j from i to n:
            current_sum += arr[j]
            if current_sum > max_sum:
                max_sum = current_sum
                max_i = i
                max_j = j
    return max_sum, max_i, max_j
```

This brute-force approach has a time complexity of $O(n^2)$.

**1b)**

```
function MaxSubarraySumOptimized(arr, n):
    max_sum = arr[1]
    current_sum = arr[1]
    max_i = 1
    max_j = 1
    temp = 1
    for j from 2 to n:
        if current_sum < 0:
            current_sum = arr[j]
            temp = j
        else:
            current_sum += arr[j]
        if current_sum > max_sum:
            max_sum = current_sum
            max_i = temp
            max_j = j

    return max_sum, max_i, max_j
```

This algorithm has a time complexity of just O(n).

**2)**

Initialize pointers i = m and j = n, starting from the bottom-right cell of the c table.
Backtrack through the table:

If X[i] = Y[j]:
    Add X[i] (or Y[j]) to the LCS
    Move diagonally up and left to (i − 1, j − 1)
If X[i] ≠ Y[j]:
    if c[i − 1][j] ≥ c[i][j − 1]
        Move to (i − 1, j)
        Otherwise, move to (i, j − 1)

Stop when i or j = 0, which means we've backtracked completely.

**3)**

Recursive Relation:

- For each coin S[i], we can either include it in our solution or exclude it:
    - If we include S[i]: We look at ways to make change for (n − S[i])
    - If we exclude S[i]: We consider solutions without using the i-th coin, which is represented by a previous state of the array dp.
- This gives us the recursive formula: dp[n] = dp[n] + dp[n − S[i]]

Base Case:

- dp[0] = 1: There is only one way to make change for zero cents, which is to use no coins.

Algorithm:

- Use a 1D DP array, where dp[n] will store the number of ways to make change for amount n.
- For each coin S[i], iterate over all values from S[i] up to N, updating the ways to make each amount by adding the ways to make the amount minus the coin's value.

Pseudocode:

```
function countWaysToMakeChange(S, m, N):
    dp = array of size N+1, initialized to 0
    dp[0] = 1  # Base case

    for i from 0 to m-1:            # For each coin in S
        for j from S[i] to N:    # Update dp for amounts from S[i] to N
            dp[j] += dp[j - S[i]]
    return dp[N]
```

Explanation of the Algorithm:

- Outer loop(coins): For each coin in S, update the number of ways to make change for each amount from the coin's value up to N.
- Inner loop(amounts): For each amount j from S[i] to N, increment dp[j] by dp[j − S[i]], which represents the ways to make j by adding the current coin S[i] to all ways of making j − S[i].

**4a)**

Let $L_1 = 3$, $L_2 = 2$, and $L_3 = 1$

Tape length X should be at least $L_1 + L_2 + L_3 = 6$ for all programs to fit.

Possible Permutations and Calculations:

Each program in the order has a retrieval time that is the sum of lengths of all preceding programs, plus its own length.

Permutation 1: $(L_1, L_2, L_3) = (3, 2, 1)$

Retrieval time for $L_1 = 3$

Retrieval time for $L_2 = 3 + 2 = 5$

Retrieval time for $L_3 = 3 + 2 + 1 = 6$

Total Retrieval Time = 3 + 5 + 6 = 14

MRT = 14/3 = 4.67

Permutation 2: $(L_1, L_2, L_3) = (3, 1, 2)$

Retrieval time for $L_1 = 3$

Retrieval time for $L_2 = 3 + 1 = 4$

Retrieval time for $L_3 = 3 + 1 + 2 = 6$

Total Retrieval Time = 3 + 4 + 6 = 13

MRT = 13/3 = 4.33

Permutation 3: $(L_1, L_2, L_3) = (2, 3, 1)$

Retrieval time for $L_1 = 2$

Retrieval time for $L_2 = 2 + 3 = 5$

Retrieval time for $L_3 = 2 + 3 + 1 = 6$

Total Retrieval Time = 2 + 5 + 6 = 13

MRT = 13/3 = 4.33

Permutation 4: $(L_1, L_2, L_3) = (2, 1, 3)$

Retrieval time for $L_1 = 2$

Retrieval time for $L_2 = 2 + 1 = 3$

Retrieval time for $L_3 = 2 + 1 + 3 = 6$

Total Retrieval Time = 2 + 3 + 6 = 11
MRT = 11/3 = 3.67

Permutation 5: $(L_1, L_2, L_3) = (1, 2, 3)$
Retrieval time for $L_1 = 1$
Retrieval time for $L_2 = 1 + 2 = 3$
Retrieval time for $L_3 = 1 + 2 + 3 = 6$
Total Retrieval Time = 1 + 3 + 6 = 10
MRT = 10/3 = 3.33

Permutation 6: $(L_1, L_2, L_3) = (1, 3, 2)$
Retrieval time for $L_1 = 1$
Retrieval time for $L_2 = 1 + 3 = 4$
Retrieval time for $L_3 = 1 + 4 + 2 = 6$
Total Retrieval Time = 1 + 4 + 6 = 11
MRT = 11/3 = 3.67

Optimal Ordering and Strategy:
The optimal ordering of programs to minimize MRT is $(L_1, L_2, L_3) = (1, 2, 3)$ with an MRT of 3.33.

Greedy Choice Strategy:
The optimal ordering places programs in increasing order of length. This suggests a greedy strategy of storing programs on the tape in ascending order of their lengths to minimize MRT.

**4b)** The optimal substructure property holds if an optimal solution to a problem can be constructed from optimal solutions to its subproblems.
Proof:

- Suppose we have an optimal order of programs that minimizes the MRT for n programs.
- For any subset of $k < n$ programs arranged optimally, removing the last program in the optimal arrangement leaves the previous arrangement, which is optimal for that subset.

This confirms that the problem has optimal substructure.

**4c)** The greedy choice property holds if a locally optimal choice at each step leads to a globally optimal solution. In this problem, the greedy choice is to place programs with shorter lengths earlier on the tape.
Proof :

- Assume we sort programs in ascending order of length and place them in that order on the tape.

- For two programs $p_i$ and $p_j$ where $L_i < L_j$, placing $p_i$ before $p_j$ minimizes the time to reach $p_i$ and the cumulative time to reach both $p_i$ and $p_j$ (compared to placing $p_j$ first).
- By placing shorter programs first, we minimize the contribution of each program's length to the MRT, as each program's length impacts the retrieval times of all subsequent programs.

Sorting the programs by length in ascending order leads to the optimal solution for minimizing the MRT. This verifies the greedy choice property.

**5a)** To maximize the height of the Huffman tree, we need to create an unbalanced tree by making the frequency distribution as uneven as possible. This will force the less frequent characters to be placed deeper in the tree, increasing the height.

Frequency Assignment for Maximum Height:

a = 64
b = 32
c = 16
d = 8
e = 4
f = 2
g = 1
h = 1

This distribution sum: 64 + 32 + 16 + 8 + 4 + 2 + 1 + 1 = 128

Height of the Tree:
Since each frequency is roughly half of the previous frequency, each character will be positioned one level deeper in the Huffman tree.

Character a will be at depth 0
Character b will be at depth 1
Character c will be at depth 2
Character d will be at depth 3
Character e will be at depth 4
Character f will be at depth 5
Characters g and h will be at depth 6

The maximum height of the tree is therefore 6.

Length of the Encoded File:
The length of the encoded file is the sum of the frequency of each character multiplied by its depth in the tree:
Length = (64×0) + (32×1) + (16×2) + (8×3) + (4×4) + (2×5) + (1×6) + (1×6)
Length = 0 + 32 + 32 + 24 + 16 + 10 + 6 + 6 = 126 bits

**5b)** To minimize the height of the Huffman tree, we need to balance the tree as much as possible, which requires assigning frequencies that are as close to equal as possible.

Frequency Assignment for Minimum Height:
Assign each character a, b, c, d, e, f, g, h a frequency of 16.

Height of the Tree:
With each character having the same frequency, Huffman's algorithm will create a perfectly balanced binary tree:
All characters will be at the same depth, at 3 and the minimum height of the tree is therefore 3.
Length of the Encoded File.

Length of the encoded file:
Length = $16 \times 8 \times 3 = 384$ bits

**6)**

```
function min_cars_for_locations(M, requests):
    # M is the number of locations
    # requests is a list of tuples representing pick-up location L at time T and drop-off at
    location L_prime at time T_prime

    events_by_location = create_empty_dict_of_lists(M)

    for each (L, T, L_prime, T_prime) in requests:
        events_by_location[L].append((T, -1))
        events_by_location[L_prime].append((T_prime, +1))

    min_cars_needed = create_empty_dict(M)

    for each location in 1 to M:
        events = sort_by_time_then_change(events_by_location[location])
        current_cars = 0            # Tracks car count during simulation
        min_cars_at_loc = 0         # Tracks minimum cars required during simulation

        for each (time, change) in events:
            current_cars += change
            if current_cars < min_cars_at_loc:
                min_cars_at_loc = current_cars
```

```
            min_cars_needed[location] = abs(min_cars_at_loc)

    total_cars_needed = sum(min_cars_needed[location] for location in 1 to M)
    return min_cars_needed, total_cars_needed
```