Anuj Kumar - 214297998

Assignment 03

1.Maximum Segment Sum, Leveled Up

(a) The subproblems are the maximum segment sums of sub-arrays from 1 to k, where k is any integer less than n. This problem doesn't require any specific data structure. Simply keeping a few variables will be enough as shown in the pseudocode below.

A variable 'best' is used to keep track of the best solution (maximum segment sum) in a sub-array, A[1..k], a variable 'curr' is used to keep track of the current solution, which keeps the sum of the sub-array ending in k having the maximum possible positive sum, and a third variable, 'alt_curr' which keeps track of the alternate sum which doesn't negate any numbers in the subarray.

(b) The recursive relation can be explained as follows; Let **MSS** denote a function which computes the maximum segment sum in a given subarray, starting from the first element until the kth element, MSS(A[1..k]). Let this function return three values, 1) the best value, which is the maximum segment sum in A[1..k] and 2) current value which is the maximum segment sum ending in k, and 3) alternate current value which is the maximum segment sum, ending in k, without negating any values. Then the new current value & alternate current value in MSS(A[1..k]) are updated as shown below. If the current value is greater than the best value, then the best value is also updated.

```
MSS(A[1..k]) {
    // base case
    If k == 1:
        best = abs(A[1]), curr = abs(A[1]), alt_curr = A[1])

    else {
        best, curr, alt_curr = MSS(A[1..k-1])
        If (A[k] + curr) > abs(A[k]) + alt_curr):
            curr = A[k] + curr
            alt_curr = alt_curr + A[k]
        else:
            curr = abs(A[k]) + alt_curr
            alt_curr = max(0, alt_curr + A[k])

        If (curr > best) then best = curr
    }

    return (best, curr, alt_curr)
}
```

The above is essentially a pseudocode for top-down solution. (Recursive relation appears in top-down approach only).

(c) Yes. The solution MSS(A[1..k]) builds on top of the solutions of a smaller subarray, MSS(A[1..k-1]). The proof is by induction. The base case is obvious. For an array of size 1, if the value is positive, then it is the best value, otherwise the best value is the negation of it. If we assume MSS(A[1..k]) returns the best value and the current value and alternate current value, then it is easy to check that the best value, current value and alternate current value assigned for MSS(A[1..k+1]) are correct. There are two cases.

1) A[k+1] is non-negative. Then this value can be simply added to the current value as well as the alternate current value.
2) A[k+1] is negative. Then negating this value may give a larger sum than the current value. Alternate value keeps the sum of the sequence without negation. So if the sum of the negated value and the alternate value is greater than the sum of the current value and the negative value, then it is better to negate A[k+1].

Then the current value is checked against the best value so far. If it is greater than the best value, the best value is updated to be the current value. In both the cases, the correct values of current value, alternate value and best values are updated. Hence by the principle of mathematical induction, this method gives the correct solution.

(d) As explained above, the base case is when the size of the array is 1. Simply return the absolute value of the element.

(e) We find the maximum segment sum from left to right (from the first element to the last).

(f) The time complexity is $O(n)$. This is because, as can be seen from the pseudocode given below, we perform only one linear scan across the whole array A. Each of the computation & comparisons are constant operations.

(g) The space complexity of our bottom-up solution given below is $O(1)$ as we use only 3 variables to keep track of our solution.

Anuj Kumar - 214297998

(h) Pseudococe

```
MSS (A):
Input: A[1..n]
Output: best
    // initialization; abs: absolute value
    best = abs(A[1]), curr = abs(A[1], alt_curr = A[1]

    for i = 2 to n {
        If (A[i] + curr) > (abs(A[i] + alt_curr) {
            curr = A[i] + curr
            alt_curr = alt_curr + A[i]
        }
        else {
            curr = alt_curr + abs(A[i])
            alt_curr = max(0, alt_curr + A[i])
        }

        if (curr > best) then best = curr
    }
    return (best)
```

2.Coin Collection, Leveled Up

(a) The subproblems are the solutions to how many more coins can be taken considering that the first k coins have already been taken. As the maximum number of coins is n, then we can keep (memoize) the solutions in an nxn matrix.

(b) The recursive solution is explained as follows: Player 1 has the option to take 1 or more coins upto 2S. Consider each of the subproblems wherein, Player 1 takes k coins, and Player 2 will pick from the rest of the coins in such a way to maximize their value. So player 1 chooses the k (k <= 2S) such that the maximum value Player 2 can obtain is minimized. Let **CC**(start, s) denote the value obtained by a player choosing coins starting from index ‘start’ and the value of S is s. Note that the same function CC can be used to compute the maximum value obtained by either Player 1 or Player 2 as both the playing conditions are identical. Then player 2 tries to minimize the gain of player 2 by iteratively considering all the possible moves of player 2 (and the maximum value of player 2 is calculated recursively).

(c) Yes. the maximum value of player 1 is calculated by considering all possibilities of choosing coins such that the maximum value obtained by player 2, who chooses from the rest of the coins, is minimized. So the assumption is that if the optimal solutions to player 2 are found, then player 1 can consider all possible value of k such that k<=2s, the maximum value obtained by player 2 is minimized. Hence if the base cases are correct, then the overall solution will be correct by the principle of mathematical induction.

(d) The bases case is that if the number of coins left is less than or equal to 2s, then choose all those coins, since that can maximize the value for a player. Then the returned value is the sum of values of all those coins selected.

(e) The data structure used is a matrix of size nxn. The order in which it will be filled is from the bottom right position.

(f) The time complexity is $O(n^3)$. This is because we have to fill in a matrix of nxn and each call to the recursive function takes at most $O(n)$ iterations.

(g) The space complexity is $O(n^2)$, since we have to fill in a matrix of dimension nxn.

(h) Pseudocode

Input: values of n coins in an array A[1..n]
Output: maximum value for player 1

Anuj Kumar - 214297998

Initialization: A matrix M[n][n] initialized to -inf

```
function max_value(start, s) {
    coins_left = n - start + 1
    if coins_left <= 2*s then return sum(A[start..n])
    else {
        if M[start][s] is not -inf then return M[start][s] // already memoized
        else {
            min_opp_score = -inf
            for (k=1 to 2*s) {
                // recurrence
                min_opp_score  =  min(min_opp_score,  max_value(start+k,
max(s,k)
            }
            M[start][s] = sum(A[start..n] - min_opp_score
        }
        Return M[start][s]

    }

max_value(1, 1) // This will find the maximum value for player 1
```