Ammar Rashed 214200715 Assignment 3

Question 1:

The base case is that the left/right most leaf nodes (at depth k) can only be reached through the previous left/right most leaf node (at depth k-1).

We start from the top of the mountain, and then add its value to the two children and update their values. Then in the next level each node can be visited from two parents, we pick the one that has the bigger value and update the current node with the new value and also update its parent to be that one that it was visited through. We keep doing this until we reach the bottom of the mountain, and then each leaf node has the maximum accumulated value and it also knows about its parent, and its parent knows about its parent and so on until the root node (i.e. the top of the mountain).

Apparently the complexity of this algorithm is the proportional to the depth of the mountain, so if it is of depth k then it is O(K2) (Quadratic), but also it can be represented in terms of number of nodes, then it would be O(n) (linear), it is just a matter of the variable we use in the Big-O notation.

Question 2:

This is identical to the weighted job scheduling problem.

First the conferences are sorted by their ending times. Then we put their number of participants with the same order in an array.

So if their order is c2: 300, c1: 200, c3: 400, the array will look like: [300, 200, 400].

We loop over the array with two pointers, i and j, where i starts from 1 (second index) till the end of the array, and j starts from 0 till i. We then check if the conference at index i is conflicting with the conference at index j, if so, then we increment j, otherwise, we choose the maximum between array[i] (which indicates the maximum number of participants if conference at index i is attended) and the summation of conference at index i’s participants and the maximum number of participants if conference at index j is attended. After that we simply pick the index in the array holding the maximum number of participants and get the combination of the conferences from the parallel combinations array.

The complexity of this algorithm is apparently quadratic O(n2), where n represents the number of conferences, since we have two nested for loops.

Question 3:

Integer partitioning problem is identical to coin change problem so I used the same solution.

We initiate a matrix whose columns represent integers from 0 up to n, and rows represent integers from 1 to n-1. That means at a row i, and a column j, the matrix holds how many combinations of integers summing up to j are possible if we have integers 1, 2, 3, … , i.

The base case is that if we have only integer 1, there is only 1 way to form any bigger integer, and if our target is 0, then we assume that we need 1 of any integer i at row i, even though it doesn’t have much meaning, but it is needed as a base case.

If the integer at row i is bigger than j, then matrix[i][j] inherits the value from matrix[i-1][j], which means that this integer at row i is actually bigger than integer j, so we can’t use it because the summation is going to exceed the limit which is j. Otherwise, matrix[i][j] equals the number of combinations if we had only {1, 2, … i-1} integers to form j plus if we had {1, 2, …i} to form j-i. That is, matrix[i][j] = matrix[i-1][j] + matrix[i][j-i]. In python we may use matrix[i][j- (i+1)], since the first row represents the integer 1, has the index 0.

In order to keep track of the combinations, instead of filling the matrix with integers indicating the number of combinations, I used Comb, which is a data structure that has a parameter (int value, list combination). So when it inherits a value from a cell in the matrix, it also inherits the combinations at this cell.

The complexity of this algorithm obviously is quadratic, O(n2), where n is the target integer we want to find the number of combinations to form it using integers smaller than it.