COMP3121 ASSIGNMNET 3

Jingjie Jin

Z5085901

Question 1:

(a).

[6, 5, 10, 100, 4, 5]. This is a counterexample that if we pick every other item from the max value(100), we will get possible maximum sum is 5+100+5=110, whereas the real maximum total sum is 6+100+5 = 111.

(b).

[6, 5, 10, 100, 4, 5]. Odd positioned sum is 6+10+4=20. Even positioned sum is 5+100+5=110. Max(odd, even) = 110. However, the real maximum total sum is 6+100+5=111.

(c). Algorithm: **O(n)**

1. Loop for all elements in A[i]
2. Maintain two sums **incl** and **excl**.
   incl = Maximum sum including the previous item.
   excl = Maximum sum excluding the previous item
3. Maximum sum excluding the current item = **max(incl, excl)**
   Maximum sum including the current item = **excl + current item**

Function: **findMaxSum(A):**

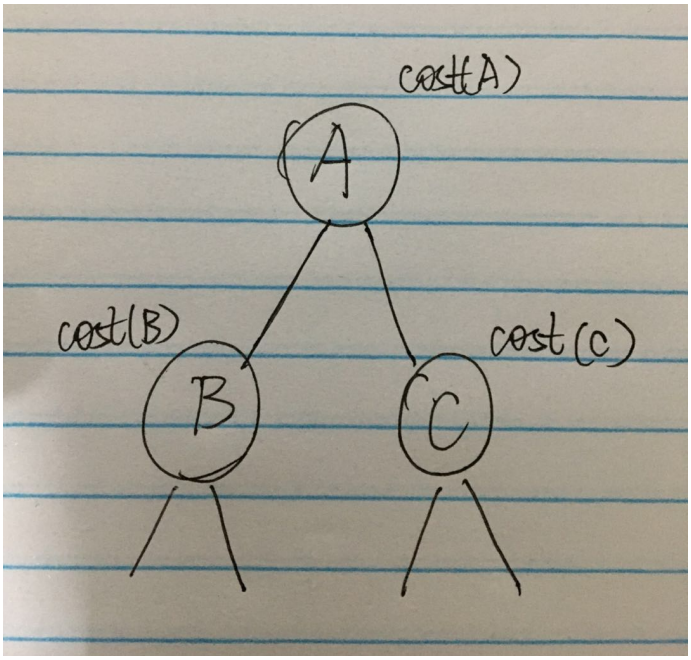
```
incl = A[1], excl = 0, temp = 0.
For i=2 to n:
    temp = max(incl, excl)
    incl = excl + A[i]
    excl = temp
Return max(incl, excl)
```

Question 2:

This problem can be solved by dynamic programming. Suppose the company hierarchy is organised as a tree and the CEO is the root. Let each supervisor be a node and the subordinates of them be their children. Let the individual cost of an employee be the value of the node.

Assume that the subproblem is the minimum of either, the total cost of including employee i, or the total cost of excluding employee i but including his/her subordinates due to the constraint.

For example:



As the graph above shows,

1. If A is invited, its children can be invited or not. The minimum total cost of this case is: **Inv(A) = cost(A) + min(Inv(B), NotInv(B)) + min(Inv(C), NotInv(C))**
2. If A is not invited, its children must be invited. The minimum total cost of this case is: **NotInv(A) = Inv(B) + Inv(C)**

Start the above algorithm from the leaves and recur upward until the root. After comparing Inv(root) with NotInv(root), we choose the smaller one as our optimal solution which represents the minimum total cost.

Question 3:

Suppose we have two sequences of Roman letters A and B, n is length of A and m is length of B. To find the number of different occurrences of A in B, we need to find all possibilities that all elements in A occurring in B as the same order of A.

Strategy:

1. If m = 0, return 1.
2. Else if n = 0, return 0.
3. Else if last characters of A and B do not match, remove last character of S and recur for remaining, then value is same as the value without last character in A. return subsequenceCount(A, B, n-1, m).
4. Else (last characters match), the result is sum of two counts.
   a) subsequenceCount(A, B, n-1, m)
   b) subsequenceCount(A, B, n-1, m-1)

Since there are overlapping subproblems in above, we can solve this problem by dynamic programming. Create a 2D array dp[n+1][m+1]. dp[i][j] denotes the number of distinct sequence of subsequence A[1...i] and subsequence B[1...j]. Thus, dp[n][m] will return the solution.

Function: **findOccurrenceCount(A, B, n, m)**

Initialize first row with all 0s: dp[0][j] = 0

Initialize first column with all 1s: dp[i][0] = 1

```
For i=1 to n:
    For j=1 to m:
        If ( A[i-1] != B[j-1] ):
            dp[i][j] = dp[i-1][j]
        else:
            dp[i][j] = dp[i-1][j] + dp[i-1][j-1]
End For
Return dp[i][j]
```

Question 4:

(a).

This problem can be solved by dynamic programming, assuming the subproblem is that the maximum sum of cell (r, c) is represented by the larger one between the maximum sum of cell (r-1, c) and cell (r, c-1), adding the value of A[r][c].

sum[r][c] = max(sum[r-1][c], sum[r][c-1]) + A[r][c]

Function: **maximumScore(A[n][n])**

```
sum[1][1] = A[1][1]
for r=2 to n:      sum[r][0] = sum[r-1][0] + A[r][0]
for c=2 to n:      sum[0][c] = sum[0][c-1] + A[0][c]
for r=2 to n:
    for c=2 to n:
        sum[r][c] = max(sum[r-1][c], sum[r][c-1]) + A[r][c]
return sum[r][c]
```

We start the algorithm from top-left and recur to the bottom-right by moving immediately below or right at each step. Thus, the algorithm takes time $O(n^2)$.

(b).

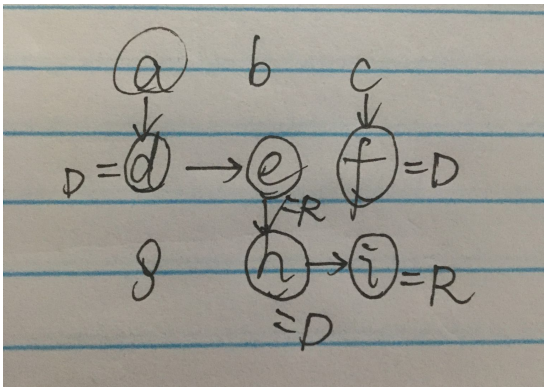
The algorithm can be modified slightly by marking the origin of current cell in order to output the path of the maximum score.

Create a 2D array **path[r][c]** representing the direction of each cell.

If path[r][c] == D, that means there is a path from A[r-1][c] to A[r][c].

Similarly, if path[r][c] == R, that means there is a path from A[r][c-1] to A[r][c].

For example:



Function: **markPath(A[n][n])**

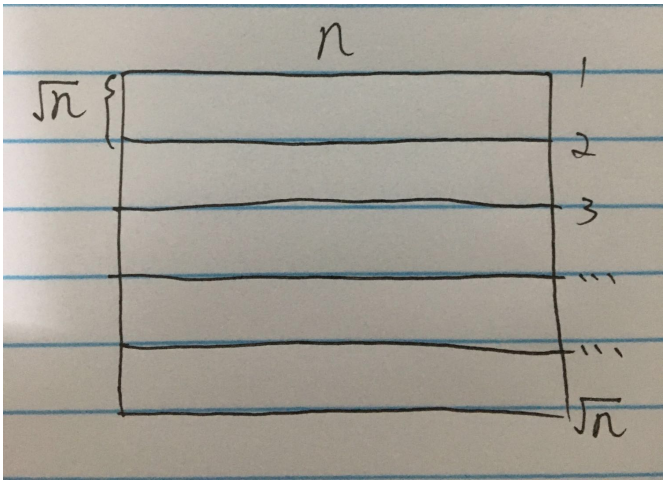
```
sum[1][1] = A[1][1]
for r=2 to n:
    sum[r][0] = sum[r-1][0] + A[r][0];
    path[r][0] = D
for c=2 to n:
    sum[0][c] = sum[0][c-1] + A[0][c]
    path[0][c] = R
for r=2 to n:
    for c=2 to n:
        if sum[r-1][c] > sum[r][c-1]:
            path[r][c] = D
        else:
            path[r][c] = R
        sum[r][c] = max(sum[r-1][c], sum[r][c-1]) + A[r][c]

Create a list: result=[]
While r>1 or c>1:
    if path[r][c] = D:
        result.add(D)
        GOTO path[r-1][c]
    else:
        result.add(R):
        GOTO path[r][c-1]

outputPath = result.reverse()
return sum[r][c], outputPath
```

After marking the direction of each cell, we check the path from the bottom-right cell to the top-left cell. Use a list result to store the path backwards. And then, reverse the list result and return. We will obtain a valid path.

Time complexity: $O(n^2) + O(n) = O(n^2)$

(c).

For the constraint on the memory, we take $\sqrt{n}$ many lines with each line in the cells of $\sqrt{n}$, so the total memory is $\sqrt{n} * \sqrt{n} * n$ = n^2 as the graph below:



Algorithm:

We start running the standard algorithm as (b). Obviously, all what we need remember is only the previous row to go to the next (The path from the same row has been known, which is not needed to be memorized). Thus, we keep running the algorithm until we reach the line i (supposed), and we only need to memorize the piece of line between i-1 and i which accounts for $n\sqrt{n}$ memory of the whole grid. For each cell in this line i, we can backtrack and **mark the previous cell** where the optimal path comes.

For the next line i+1, we erase the memory we memorized before and start to memory this piece. Because of the checkpoints we marked for the previous cells, we do not need to worry about the path information.

Eventually, we end up at bottom-right cell. Also, we can find the optimal path by backtracking the previous cell as the method in (b).

Question 5:

This problem can be regarded as finding a unique minimum cut in distinct vertices u and v such that u is in the same side of the cut as the source s and v is in the same side of the cut as the sink t, in a network flow graph G.

Firstly, we classify all vertices into three types:

1. **Upstream**, that is, vertex u such that for all minimum s-t cuts (A, B), we have $u \in A$, u lies on the source side of every minimum cut.
2. **Downstream**, that is, vertex v such that for all minimum s-t cuts (A, B), we have $v \in B$, v lies on the sink side of every minimum cut.
3. **Central**, that is, vertex x that neither upstream nor downstream, there is a minimum cut (A, B) such that $x \in A$, and at least one minimum cut (A', B') such that $x \in B$.

Consider the cut (A^*, B^*) found by using BFS on the residual graph G_f at the end of a maximum flow algorithm. We claim that a vertex i is upstream if and only if $i \in A^*$. Clearly, if i is upstream then it must belong to A^*, otherwise, it lies on the sink side of the minimum cut (A^*, B^*). On the contrary, if $i \in A^*$ is not upstream, then there would be a minimum cut A', B' with $i \in B'$.

Now since $i \in A^*$, there is a path P in G_f from s to i. Since $i \in B'$, this path must have an edge (i, j) with $i \in A'$ and $j \in B'$. But this is a contradiction, because no edge in the residual graph can go from the source side to the sink side of any minimum cut.

Symmetrically, the following argument can be proved. Let B_* denote the vertices that can reach t in G_f, and let $A_*=V-B_*$. Then (A_*, B_*) is a minimum cut, and a node j is downstream if and only if $j \in B_*$.

Thus, our algorithm is to compute a maximum flow f, build G_f, and use BFS to find the sets A^* and B_*. These are the upstream and downstream vertices respectively. the remaining vertices are central. Obviously, this algorithm runs in polynomial time.

Conclusion:

If there exist any central vertices, this means that there is not a unique minimum cut by the definition of "Central".

If there are no central vertices, then the partition of the vertices into upstream and downstream defines a cut (A, B). Suppose there are some minimum cut (A', B') other than (A, B), Then either there is a vertex $i \in A-A'$, contracting the definition of "Upstream", or there is a vertex $i \in B-B'$, contracting the definition of "Downstream". Thus (A, B) is the unique minimum cut which is our optimal result.

Question 6:

(a).

Observe that this problem can be proved by greedy method which is optimal. Sort the rides in non-increasing order of D[i] and go on in this order.

Firstly, define the total payment until ride k as: $f(k) = D_k + \sum_{i=1}^{k} C_i$, so the overall total payment is the maximum among all f(k).

Suppose there are two adjacent deposit i and i+1 in our order such that **D[i]<D[i+1]**.

Then modify the order as we expect: **1, 2, … i-1, i+1, i, i+2, …, n.**

Let g(k) be the total payment for ride k under this order.

Now, let **S=C[1]+C[2]+…+C[i-1]**, then we have that:

f(i+1) = S + C[i] + C[i+1] + D[i+1]

g(i) = S + C[i+1] + C[i] + D[i]

g(i+1) = S + C[i+1] + D[i+1]

Since D[i]<D[i+1], we have g(i)<f(i+1), and also, g(i+1)<f(i+1).

Hence, after swapping i and i+1 to get a modified sorted order, we can get a strictly smaller sum of total payment that will **never increase max[f(k)]**.

Therefore, a strictly smaller total payment can delay the bottleneck of the rides going. So if and only if we go on under this order, we can complete all the rides.

(b).

Firstly, sort the rides in non-increasing order of deposit D[i] which runs in time O(nlogn) using merge-sort.

Next, create a 2D array **park[T][n]** with row i (1 to T) denotes the money has been used and column j denotes the rides in non-increasing order of deposit. park[i][j] represents the number of rides have been played.

Initially, let park[i][j] = 0 for all i from 1 to T, all j from 1 to n.

Then, the subproblem is that park[i][j] will decide its following values such that:

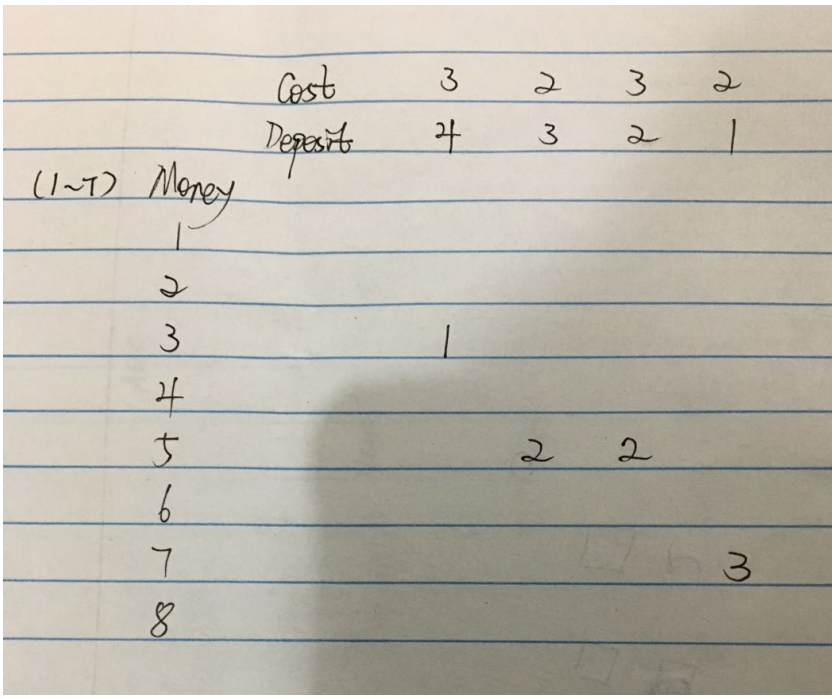
Case1: If play ride j+1, then park[i+c][j+1] = park[i][j] + 1

(c is the cost of ride j C[j])

Case2: If not play ride j+1, then park[i][j+1] = park[i][j]

Also, before assigning the value into the array park[T][n], we need to determine whether the total payment exceeds T:
Case1: If i+C[j+1]+D[j+1] <= T, that means ride j is available to be played, we can add 1 to park[i][j] at park[i+c][j+1].
Case2: If i+C[j+1]+D[j+1] > T, that means ride j cannot be choose, we remain the number at this column with the same row.

PS: The typical example is shown in graph below:



This algorithm will terminate after scanning the last ride n. We can determine the maximum number by using **max(park[i][n]) i∈T** which could be found near the bottom of the column.
Because we have n different rides and T initial money, this algorithm runs in time O(nT).
In total, plus the sort time O(nlogn), we can determine the maximum number of different rides that can go on in time **O(nlogn + nT)**.

(c).
If there are too many rides in this park such that n is an exponential number, this

algorithm may lead to pseudo-polynomial time.
But in our daily life, n and T can always be some "reasonable" number, so in this case, the algorithm can run in polynomial time.

(d).

END