COMP3121 Assignment 3

Ziyi Cui z5097491

1.(a).

Example array:
A = [5, 10, 9, 1]
The sum of A[0] and A[2] is 14 > the sum of A[1] and A[3] is 11
The maximum total sum of values of the items is 14, because we cannot pick the adjacent item. However, the largest value in A is 10 at A[1].

1.(b).

A = [50,1,1,50]
The maximum total sum of values of the items is 100.
We pick A[0] and A[3], which is a combination of odd and even numbered elements

1.(c).

DP-solution:

```
If(N == 0):
        Return
sum_table[N]
i = 0
while( i < N ):
        if(i == 0) :
                sum_table[i] = A[i]
        if(i == 1):
                sum_table[i] = (A[0] ≥ A[1]) ? A[0] :A[1]
        else:
                sum_table[i] = sum_table[i-2]
                if(A[i] ≥ 0 and sum_table[i] ≥ 0):
                        sum_table[i] += A[i]
                if(A[i] ≥ 0 and sum_table[i] < 0):
                        sum_table[i] = A[i]
                if(A[i] < 0 and sum_table[i] < 0):
                        sum_table[i] = (sum_table[i] ≥ A[i]) ? sum_table[i] : A[i]
        i += 1
```

we use sum_table[i] store the potential maximum sum of the A[0..i] with no adjacent items above in time O(n). And the solution is the largest value in the sum_table.
Right now we can go through the sum_table to find the actual maximum sum in the table, which runs in linear time.

And the maximum sum in the table is the solution.
This solution runs in linear time.

2.

We use tree-like structure to represent this hierarchical structure, each node contains the cost of retreat the employee.

We first solve the sub-problem that find the optimal cost for levels of the tree.

If the tree only has one level (level == 0), then we don't need to retreat anyone and the total cost is 0. Level 0 cost is zero (stored in level[0])

If the number of the levels > 1, then we have to retreat some employees because if x's immediate supervisor (parent in the tree) is not attending the retreat, then x must attend the retreat.

If level == 1, then we compare the sum of the cost of retreat all children in the current level with the cost of retreat their parents in level 0. Pick the smaller cost between the 2 costs and store the cost in level[1]. (If 2 the costs are same, then pick the first one).

If level > 1 (assume the current level is i), then we compare the sum of retreat all children in the current level and optimal cost in level[i-1] with the cost of their parents in the level i – 1 (Note that we do not count the cost of node at level i-1 who has no children) and optimal cost in level[i-2]. Pick the smaller cost between the 2 costs and store the cost in level[i]. (If 2 the costs are same, then pick the first one).

In this method, we always find the optimal solution for next level by using the optimal solution from the previous levels. And, we will find the final optimal solution at the last level.

3.

2D-DP solution:

For all $1 \le i \le n$ and all $1 \le j \le m$ let c[i,j] be the number of different occurrences of A in B, B = $<b_1, \dots, b_i>$ and A = $<a_1, \dots, a_j>$

$$c[i,j] = \begin{cases} 0 & \text{If } i = 0 \\ 1 & \text{If } j = 0 \\ 1 & \text{If } j = 0 \text{ and } i = 0 \\ c[i-1, j-1] + c[i-1, j] & \text{If } i, j > 0 \text{ and } a_j = b_i \\ c[i-1, j] & \text{If } i, j > 0 \text{ and } a_j \neq b_i \end{cases}$$

For example:
1.A = ba, B = baba

	i	0	1	2	3	4
j			b	a	b	a
0		1	1	1	1	1
1	b	0	1	1	2	2
2	a	0	0	1	1	3

So the final solution is store in the c[4,2], which is 3

2.A = aa, B = aaaa

	i	0	1	2	3	4
j			a	a	a	a
0		1	1	1	1	1
1	a	0	1	2	3	4
2	a	0	0	1	3	6

The solution is store in the c[4,2], which is 6
In this method we always solve the sub-problem, and store the optimal solution in c[i,j].
Explain:
If we want to find optimal solution for c[i,j], we assume all the values we get before is optimal.
If $a_j = b_i$, then we check how many matches we get before for the current sub-sequence at c[i,j-1], and then we check how many new matches we can get by combining the current letter with previous sub-sequence without the current letter at c[i-1,j-1]. So the total matches in c[i,j] is c[i-1,j-1] + c[i,j-1]
If $a_j \neq b_i$, we just simple check how many matches we get before for the current sub-sequence at c[i,j-1]. c[i,j] = c[i,j-1].
And the solution for this question is In c[n,m].
Therefore, we can get optimal solution in time O(n^2).

4.(a)

2D-DP solution:

For all 1 ≤ i ≤ n and all 1 ≤ j ≤ n let C[i,j] be the maximum score we can get at the point A[i,j]. (A[i,j] == A[i][j])

$$C[i,j] = \begin{cases} 0, & if\ i,j = 0 \\ C[i-1,j] + A[i,j], & if\ i,j > 0\ and\ C[i-1,j] \geq C[i,j-1] \\ C[i,j-1] + A[i,j], & if\ i,j > 0\ and\ C[i-1,j] < C[i,j-1] \end{cases}$$

Explain:
In this method we always solve the sub-problem, and store the optimal solution in c[i,j].
If we want to find optimal solution for c[i,j], we assume all the values we get before is optimal.
If i,j = 0, we first padding the table C with 0.
Now we are at the point A[i,j], we know all the the optimal solution in c[1..i-1, 1..j-1].
There are only 2 points can direct go to the point A[i,j] (A[i-1,j], A[i,j-1]).
If we want to get the maximum score at A[i,j] we have to come from the point max(c[i-1,j], c[i,j-1]).
Therefore, we can guarantee we get the maximum score for each point in the time O(n^2)
The solution for this question stored in c[n,n].

For example:
A:

	1	2	3	4
1	7	7	5	9
2	4	3	10	20
3	11	12	13	15
4	1	2	4	1

$\longrightarrow$

C:

	0	1	2	3	4
0	0	0	0	0	0
1	0	7	14	19	28
2	0	11	17	29	49
3	0	22	34	47	64
4	0	23	36	51	65

4.(b)

In part(a), we only store the maximum score for each point in table C. Right now, we store a tuple (maximum score, last move) rather than a simple score for each point in table C. And we set the last move to null for the starting point because there are no last move for the starting point.
If last move is right then we set the last move to R (from A[i-1,j] to A[i,j])
Else we set the last move to D (from A[i,j-1] to A[i,j])

For example:
A:

	1	2	3	4
1	7	7	5	9
2	4	3	10	20
3	11	12	13	15
4	1	2	4	1

$\longrightarrow$

C:

	0	1	2	3	4
0	0	0	0	0	0
1	0	$[7, null]$	$[14, R]$	$[19, R]$	$[28, R]$
2	0	$[11, D]$	$[17, D]$	$[29, D]$	$[49, R]$
3	0	$[22, D]$	$[34, R]$	$[47, R]$	$[64, D]$
4	0	$[23, D]$	$[36, D]$	$[51, D]$	$[65, D]$

After we get table C above, we can back track the route from the end point c[n,n] to the starting point.
If the last move in c[i,j] is D, we push D into path_array and go check the next point c[i,j-1].
If the last move in c[i,j] is R ,we push R into path_array and go check the next point c[i-1,j] .
If the last move is null, then we stop.

We'll get path_array[D,D,R,D,R,R].
To get the real path, we need to reverse the path_array .
The final path is RRDRDD.

The time spend on generating Table C is O(n^2) and finding the final path is O(n).
Therefore, the time complexity is O(n^2).

4.(c)

We need slightly change the algorithm in part(a) and part(b), We firstly change the tuple to (maximum scrouce, path).

We can only use O(n$\sqrt{n}$) memory to store the information, so we use a circular buffer to do this question. The table has $\sqrt{n}$ rows and n column to maintain the useful data. If i,j > 1, then we only need the one left and one up point to get the maximum score and path for C[i,j]. Therefore, we can rewrite the memory that we already used and replace it with the new point.

Basic recursion:

```
If i,j = 1, then
        C[i, j] = A[i, j]; path = []
Else If i = 1, then
        C[i, j] = C[i, j-1] + A[i, j]; path += R
Else
        r = i mod floor(√n)
        c = (i-1) mod floor(√n)
        if j = 1, then
                C[r, j] = C[c, j] + A[i, j]; path += D;
        Else:
                If C[c, j] ≥ C[r, j-1], then
                        C[r, j] = C[c, j] + A[i, j]; path += R
                Else
                        C[r, j] = C[r, j-1] + A[i, j]; path += D
```

we finish the whole process after we add the point A[n,n] and its path.
And the score and path stored in the last C[r,j] we got.

For example:

A:

	1	2	3	4
1	7	7	5	9
2	4	3	10	20
3	11	12	13	15
4	1	2	4	1

→ C:

	1	2	3	4
1	$[7,[]]$	$[14,R]$	$[19,RR]$	$[28,RRR]$
2	$[11,D]$	$[17,RD]$	$[29,RRD]$	$[49,RRDR]$

→ C:

	1	2	3	4
1	$[22,DD]$	$[34,DDR]$	$[47,DDRR]$	$[64,RRDRD]$
2	$[11,D]$	$[17,RD]$	$[29,RRD]$	$[49,RRDR]$

→ C:

	1	2	3	4
1	$[22,DD]$	$[34,DDR]$	$[47,DDRR]$	$[64,RRDRD]$
2	$[23,DDD]$	$[36,DDRD]$	$[51,DDRRD]$	$[65,RRDRDD]$

The last C[r,c] we get is C[2,4].

This algorithm runs in time O(n^2) and only takes O(n$\sqrt{n}$) additional memory.

5.

Use a super source S to connect s and u with 2 infinite capacities edges edge(S,s) and edge(S,u) because we want to keep u and s in the same side. Also, use a super sink T to connect t and v with 2 infinite capacities edges edge(T,t) and edge(T,v).

Theorem: The maximal amount of flow in a flow network is equal to the capacity of the cut of minimal capacity.

Apply the Edmonds-Karp algorithm to find the maximum flow between S and T, then we can get the minimum cut among all cuts for s and u in the same side of the cut, and t,v in another side.

The super source and super sink will not affect the maximum flow of original network flow graph G, because all edges between them and other vertices has infinite capacities.

The time complexity of Edmonds-Karp algorithm is O($|V||E|^2$), which runs in polynomial time

6.(a)

We use cost[i] to represent the cost of the total cost from the take 1st ride to the i^{th} ride, where 1 ≤ i ≤ n. (cost[i] = sum(C[1..i]))
If she wants to take the next ride i+1, then the money she has must satisfy:

$$Cost[i+1] + D[i+1] \leq T$$

We know that T can satisfy the inequality above for some subset of all the rides in some order.
If we swap ride i and ride i + 1 when D[i+1] > D[i], then
We will have the inequality:

$$Cost[i+1] + D[i] < Cost[i+1] + D[i+1] \leq T$$

$$Cost[i+1] + D[i] < T$$

Which means, she still can ride of all the rides in this order in this subset.
We keep swapping ride i and ride i + 1 until there are no 2 rides can satisfy D[i+1] > D[i]. Finally we will get D[1..n] in a non-increasing order.
Therefore, she can go on the same subset S of rides in a non-increasing order of deposit D[i]

6.(b)

We mergesort the rides follow the rules above to get a list of rides in non-increasing on D. O(nlogn)