Shubham Gupta **Theoretical Assignment: 2**
180749 **Part: 2**

Problem 4

We will use the merge sort algorithm to tackle the problem. A modification in the merge step will ensure that we increment the counts when the element comes from the right because the halves are sorted.

- **Pseudo-code**

```
Merge (A[], B[], C[], l, mid, r){
        a ← r - l + 1; b ← l; c ← mid + 1; d ← 0; sC ← 0;
        R[a] = 0;
        While (b <= mid and c <= r)    {
                If (A[B[b]] >= A[B[c]])    {sC++; R[d++] = R[c++];}
                Else    {C[B[b]] ← C[B[b]] + sC; R[d++] = R[b++];}
        }
        While (b <= mid)    {C[B[b]] ← C[B[b]] + r - mid; R[d++] = R[b++];}
        While (c <= r)    {R[d++] = R[c++];}
        Copy array R to B;
}
Merge_sort (A[], B[], C[], l, r){
        If (l = r) return 0;
        Else{
                mid ← (l + r) / 2;
                Merge_sort (A[], B[], C[], l, mid);
                Merge_sort (A[], B[], C[], mid + 1, r);
                Merge (A[], B[], C[], l, mid, r);
        }
}
```

- **Analysis of Time Complexity**

Two Merge_sort run in 2*T(n/2)
Merge runs in O(n)

If n = 1, T(n) = c for some constant c
If n > 1,

$T(n) = c*n + 2*T(n/2)$
$= O(n*\log(n))$ *Using Master Theorem

- ## Problem 5
  - ### Data Structure and Algorithm
    We will use Graphs and do a DFS (Depth First Search) traversal to solve this problem. We will obtain a DFS tree after the DFS traversal. If there is a back edge in the DFS tree, then there has to be a cycle in our graph. We can keep an array that stores all the visited nodes, and while traversing, if we find a back edge to any of the visited node, we have a cycle and will return True.

    The data structures used is Graphs, while the algorithm is DFS.

  - ### Analysis of Time Complexity
    GIVEN: An undirected graph G = (V, E), where V is the set of vertices and E is the set of edges.

    We have to look at 2 cases when V > E and V <= E.

    CASE1: V > E

    We go through the algorithm mentioned above, which could have two outcomes. The graph will have a cycle and return True, or it will return False. In either case, the time complexity will be O(V + E), which will be equivalent to O(V) since V > E.

    CASE2: V <= E

    For this case, there must be a cycle in the graph. We will see two subcases for case2.
    A premonition must set up for this derivation about a tree. A tree has n nodes and n-1 edges, and this can be proven easily using induction.
    - SUBCASE1: We have a connected graph G = (V, E)
      We will follow the algorithm mentioned above, which will give us a DFS tree (T) with all Vs. T has to have V-1 edges. We also know that V <= E, therefore there will be at least one edge in G which will not be in T. All these left edges will give cycles in G.
    - SUBCASE2: We have an unconnected graph G = (V, E)
      Let G have 2 connected components $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. As we know that V <= E, this implies that either $V_1 <= E_1$, or $V_2 <= E_2$ or both of them simultaneously. For these cases to be true, either G_1, or G_2, or both simultaneously will have a cycle.

    Both these subcases run in O(V), hence independent of the number of edges.

- ## Problem 6

  The problem will use BFS on an undirected graph to find the shortest path for this question. At max, a knight can move eight places from its current position. We move to each of this position; if the required path is not reached, we push all these paths in a queue. While pushing, we keep incrementing the count of jumps by 1. We stop when we get to the required position.

  For all this, we need a data structure to store x and y coordinates with distance. We will also need a function to check if any point lies on the chessboard or not. And finally, a procedure to implement our queue algorithm.

  - ### Pseudo-code

```
\\ A structure to store x coordinate, y coordinate and distance.
struct CELL{}

\\ Boolean function to check if any point lies on the chessboard or not
Inside (x, y, Bound){
        If((x >= 1 and x <= Bound) and (y >= 1 and y <= Bound))    { return True; }
        Else    { return False; }
}

\\ Main function
Min_steps (init, final, Bound) {
        X[8] ← all feasible x coordinates; Y[8] ← all feasible y coordinates;
        board[Bound + 1][Bound + 1] = False; \\ no cell is visitd
        Q; // queue
        Q.push(initial positon of the knight);
        While (Q is not empty) {
                a ← Q.front; Q.pop;
                If (current position == final position)    {return distance of a;}
                For ( 0 to 8 ) {
                        x = current x + X[i];
                        y = current y + Y[i];
                        If ( Inside (x,y, Bound) and not yet visited) {
                                board[x][y] = True;
                                Q.push(current position);
                        }
                }
        }
}
```

  - ### Analysis of time complexity

    Worst-case time complexity will be $O(n^2)$, as you might visit all the position on

boards before reaching the destination.
Therefore, the time complexity is $O(n^2)$.

- **Analysis of space complexity**
  We have to maintain a 2D array of size N*N. Therefore, the space complexity will be $O(n^2)$.