Ans 1(a):

Approach:

- Sort last k elements in decreasing order; It will take O(k log(k)) time complexity.
- As last k elements were smaller than all n-k coming before, it will make the whole array mountain like (i.e., first increasing, then decreasing).
- Then using binary search we will find the index, where peak (say p) of the mountain is coming in O(log(n))
- After finding the peak index (p), given **Val** we can apply binary search for it in the two regions that are from (0 -> p) and (p+1 -> n-1) and return true if present in either.
- Above query step will take O(log(p+1) + log(n-p+1)) <= O(log(n));
- So overall Time complexity will be **O(k log(k) + q log(n))** for q queries.

Pseudo-code:

```
Sort(A,n-k,n-1); // sorts array on last k indexes
For finding peak index:
int lo=0,hi=n-1;
int mid= (lo+hi)/2;
int p=n-1;
while(lo<hi){
        if(A[mid]>=A[mid-1] && A[mid]>=A[mid+1]){
                p=mid;
                break;
        }
        else if(A[mid]>=A[mid-1]){
                lo=mid+1;
        }
        else{
                hi=mid-1;
        }
        mid= (lo+hi)/2;
}
return p;

For query:
bool present=false;
If(Val>=A[0] && Val<=A[p]) present |= binary_search_inc(A,0,p,Val);
If(Val>=A[n-1] && Val<=A[p]) present |= binary_search_dec(A,p,n-1,Val);
return present;

(binary_search_inc -> binary search on increasing array)
(binary_search_dec -> binary search on decreasing array)
```

Ans1(b):

Time complexity analysis done above in approach section and got it as **O(k log(k) + q log(n)) .**
Proof of correctness:

- As last k elements were randomly arranged, sorting them decreasingly makes the array fully mountain likes as last k elements were all less than rest of elements.
- Finding peak (p) using binary search works, because firstly there will be unique maxima of the array lying somewhere between first and last element.
  As the array becomes mountain like so if our A[mid] is our peak then it will be greater than both of it neighbors (distinctness is given in question).
  Else if A[mid] is greater than A[mid-1] and not A[mid+1], it is on left side of the mountain array and maxima(peak) is ahead of it, so we just increased lo;
  Else if A[mid] is greater than A[mid+1] and not A[mid-1], it is on right side of the mountain array and maxima(peak) is behind of it, so we just decreased hi;
  And since we can neglect a portion(nearly half) of search space by either increasing lo or decreasing hi, Binary search works.
- Given a query, if element is present, it will be either between start and peak of mountain like array or between peak to end of array.
- Each part of mountain array is sorted so as binary search works correctly for sorted array, our algorithm works fine.

Ans2(a):

Proving right to left:

- As for all k ∈ {0, 1, . . . , n − 1} there exists a vertex v with Outdegree(v) = k, because there are only n vertices, for each k in above range there will be **exactly one** vertice.
- Now consider the vertex with Outdegree(v) = n-1, clearly, it will have edges going out to every other vertex. And every other vertex can't have an edge from them to this vertex, as there is atmost one edge between any pair of vertices.
- Now continuing the same argument as above for the vertex with Outdegree(v) = n-2, clearly it will have n-2 edges going out to every other vertex except the one with Outdegree(v) = n-1, because of reason stated above.
- Now continuing like this upto Outdegree(v)=1, we can observe that there is an edge from vertex a to vertex b if **Outdegree(a) > Outdegree (b).**
- Now given any two vertex a and b, there will always be an edge as:
  If (**Outdegree(a) > Outdegree (b)**) edge from a to b
  Else if(**Outdegree(a) < Outdegree (b)**) edge from b to a
  By the reasoning stated just above
  (Note: **Outdegree(a) != Outdegree (b)** ever because we have exactly one vertex of each outdegree)
  So we are done with first requirement of Perfect Complete Graph
- Now if there is an edge from a to b and an edge from b to c,
  => **Outdegree(a) > Outdegree (b)** and **Outdegree(b) > Outdegree (c)**
  => **Outdegree(a) > Outdegree (c)** => there is an edge from a to c
  Hence second requirement of Perfect Complete Graph is also fulfilled.

Proving left to right:

- First of all, observe that if we remove a vertex and all of its connected edges from a Perfect Complete Graph, it will still remain a Perfect Complete Graph.
- Now, we will prove the argument by induction on size of a Perfect Complete Graph, for n=1 its trivially true as Outdegee(v)=0.
- Let the argument is true for Perfect Complete Graph of size n,
  -> i.e. there is at most one edge, and for all k ∈ {0, 1, . . . , n − 1}, there exist a vertex v in the graph, such that Outdegree(v) = k.
  We need to prove it is also then true for n+1.
- Given a Perfect Complete Graph of size n+1,
  For a while, remove a vertex with the highest Outdegree (say r), so it will now become a Perfect Complete Graph of size n. And the above properties hold.
- Now it suffices to show that if we restore the removed vertex (r) and edges, it has Outdegree(r) = n.
- Suppose Outdegree(r) !=n , now because this is Perfect Complete Graph there is an edge for any pair of distinct vertices, hence there is an edge with vertex v which had Outdegree(v) = n-1 in the vertex removed version of the graph.

-> If edge is from v to r then Outdegree(v)=n in non-removed version of the graph and as we can't have edge simultaneously from r to v, Outdegree(r) < n = Outdegree(v), contradiction to vertex r had highest Outdegree !!
-> If edge is from r to v, then as v has edges going out to all other vertices(except r), r will also have edge going out to all other vertices, which implies Outdegree(r)=n, again contradiction !!

- Hence Outdegree(r)=n, and by induction, we already had vertices with Outdegree $\in \{0, 1, \ldots, n-1\}$. And there is exactly one edge with every vertex so at most one edge condition is also satisfied.

Hence Both side proved !

Ans2(b):

Approach:

- Let adjacency matrix be A, then first check for all 0 <= i < j <= n , A[i][j] + A[j][i] = 1, which suffices that there is exactly one edge between every pair of distinct vertices
- Now take a vector of size n, say OUT, in which we will update whether there is a vertex of Outdgree = i at i-th index (0 <= i <= n-1).
- Now take row-sum of each row in A and update OUT[row_sum(A[i])]++
- Now in the vector OUT check if every entry is >0, if yes then it is Perfect Complete Graph, else not (this works as characterization given in part (a) holds)

Pseudo-code:

```
bool isPerfect_Complete_Graph ( vector<vector<int>> A ){
        for(int i=0;i<n;i++){
                for(int j=i+1;j<n;j++){
                        if(A[i][j]+A[j][i] != 1) return false;
                }
        }
        vector<int> OUT(n,0);
        for(int i=0;i<n;i++){
                int x = row_sum(A[i]); // row_sum gives sum of the vector entries
                OUT[x]++;
        }
        for(int i=0;i<n;i++) if(OUT[i]==0) return false;
        return true;
}
```

Time Complexity Analysis:

- First two nested loops will take O(n^2) time complexity.
- Then vector initialization in O(n)
- row_sum will take O(n), hence that loop will take O(n^2) time complexity
- Last loop again O(n), so overall O(n^2+n+n^2+n)= O(2n^2 +2n) = **O(n^2) time complexity**.

Ans3(a):

$S(A) = \sum_{i=1}^{n-1} |a_{i+1} - a_i| \geq |a_n - a_{n-1} + a_{n-1} - a_{n-2} + \ldots + a_2 - a_1| = |a_n - a_1|$ (By Triangle inequality)

And this equality holds when all the terms ins summation have same sign,

i.e. either $a_{i+1} - a_i \geq 0$ or $a_{i+1} - a_i \leq 0$ for all i= 1,2,...,n-1

So S(A(Π)) is minimum when A(Π) have either $a_{i+1} - a_i \geq 0$ or $a_{i+1} - a_i \leq 0$ for all i= 1,2,...,n-1

⇒ **Good permutations or A(Π_0) are increasing and decreasing sorted permutations of A.**

Ans3(b):

Approach:

- So we need to calculate minimum cost required to transform the given array A to a sorted array A (lets do for increasingly sorted, other will follow similarly)
- First make a copy of array and sort it. Call it array B.
- Now, as cost for swapping the elements at index i and j is **max(a_i , a_j).** So greedily we will obviously try to move bigger elements as less as possible.
- Go to first element in A, note it and then find its index no. in B. now go to this index in A and repeat the process until it becomes a loop (i.e. first element comes again). Now sorting numbers in this loop sequence is a sub-problem and nothing do with others numbers.
- Minimum cost for sorting this loop sequence would be **Sum of all elements except the smallest one.** (This comes as a result that we have to swap each wrong positioned element atleast once and least possible times for bigger elements, so in each swapping we just put the biggest element at its correct place and after all elements(in loop sequence) are placed, the minimum element will automatically be at correct position. It's an simple group theoretic result)
- Now, after loop sequence repeat the process from next unvisited element in A and add their minimum costs to the final answer. (Obvoiusly, element placed at correct place will have 0 as min. cost)
- Apply the same method while taking B as decreasingly sorted, and return the minimum of the two costs.

Pseudo-code:

```
Min_cost(A){ // this calculates the required minimum cost
        B=A;//copy of array A
        sort_inc(B); //sort in inc. order

        n=A.size();
        visited(n,0); // initially all postions unvisited
        Cost1=0;
        for( int i=0 ;i<n ; i++){
                if(!visited[i]){
                        Cost1+= cost_calc(A,B,i,visited);
                }
```

```
        }

        C=A;//copy of array A
        sort_dec(C); //sort in dec. order

        visited2(n,0); // initially all postions unvisited
        Cost2=0;
        for( int i=0 ;i<n ; i++){
                if(!visited2[i]){
                        Cost2+= cost_calc2(A,C,i,visited2);
                }
        }

        return min(Cost1,Cost2);
}
cost_calc(A,B,i,visited){
        mini= ∞ ;
        curr=i;
        Cost=0;
        while(visited[curr]!=1){
                visited[curr]=1;
                Cost+=A[curr];
                mini=min(mini,A[curr]);
                p = binary_search_inc_index(B,A[i]);
                curr=p;
        }
        return Cost-mini;
}
cost_calc2(A,C,i,visited){
        mini= ∞ ;
        curr=i;
        Cost=0;
        while(visited[curr]!=1){
                visited[curr]=1;
                Cost+=A[curr];
                mini=min(mini,A[curr]);
                p = binary_search_dec_index(C,A[i]);
                curr=p;
        }
        return Cost-mini;
}

// binary_search_inc_index-> returns index in increasing sorted array using binary search
// binary_search_dec_index-> returns index in decreasing sorted array using binary search
```

Time Complexity Analysis:

- Copying takes O(n) and sorting takes O(n log(n)) time complexity.
- Each of **cost_calc** function works in O(k log(n)) for loop of size k, as while loop will run for each element in loop and binary_search will take O(log(n)) each time.
- So, the Min_cost function,for loop will on total take summation of O(k log(n)) on every loop of elements, hence O(n log(n)) time complexity for it.
- Hence, overall time complexity is O(n) + O(n log(n)) = **O(n log(n))**

Ans4(a):

Approach:

- Run a BFS and store the edges in path from s to t in a set (say L). And the no. of edges in shortest path of s to t is let say k.
- Now, for each (u, v) ∈ E, set M[u, v] = k;
- Now for each edge (u, v) ∈ L, run BFS(with not considering (u, v) as an edge) and calculate the dist(s,t), update M[u,v]= dist(s,t).

Pseudo-code:

```
Solve_Batman(G,M){
        k=BFS(G,s,t,NULL,NULL)      //return shortest path length
        L=BFS_L(G,s,t) //return edges in shortest path

        For each (u,v) in E
        {
                M[u,v]=k;
        }

        For each (u,v) in L
        {
                len=BFS(G,s,t,u,v)
                M[u,v]=len;
        }
        return M;
}

BFS(G, s, t, a , b){ // running BFS with not considering edge (a,b)
        distance(u)= ∞; //initially for all u in V
        CreateEmptyQueue(Q);
        distance(s)=0;
        Enqueue(s,Q);
        While(Not IsEmptyQueue(Q)){
                v= Dequeue(Q);
                For each neighbor w of v {
                        if(distance(w)== ∞  && ! (w==a && v== b ) && ! (w==b && v== a ) ){
                                distance(w)=distance(v)+1;
                                Enqueue(w,Q);
                        }
                }
        }
        return distance(t);
}
```

```
BFS_L(G, s, t){
        parent(u)= -1 //initially for all u in V
        distance(u)= ∞; //initially for all u in V
        CreateEmptyQueue(Q);
        distance(s)=0;
        Enqueue(s,Q);
        While(Not IsEmptyQueue(Q)){
                v= Dequeue(Q);
                For each neighbor w of v {
                        if(distance(w)== ∞){
                                parent(w)=v;
                                distance(w)=distance(v)+1;
                                Enqueue(w,Q);
                        }
                }
        }
        L <- new adjacency list
        curr=t
        while(parent(curr) != -1){
                Add_edge( L , (parent(curr),curr) ); //adds edge in the adjacency list
                curr=parent(curr);
        }
        return L;
}
```

Ans4(b):

Time Comlexity Analysis:

- **BFS()** ⇒ initialization of distance array in O(|V|), while loop will run for each element and for loop in total will run atmost 2*E times so O(|V|+|E|) for it. So overall BFS() takes O(|V|+|E|)+O(|V|)= **O(|V|+|E|)** for each of its function call.
- **BFS_L()** ⇒ apart from executions same as BFS(), it has extra while loop which will run for at most |V| times as path between any two vertex can't be more than that. So overall O(|V|+|E|)+O(|V|)= **O(|V|+|E|)** time complexity for BFS_L().
- Now in the main function **Solve_Batman()**, first BFS() & BFS_L() calls will take **O(|V|+|E|)** time. Then for loop on E will take **O(|E|)** time.
- In next for loop, BFS() will take **O(|V|+|E|) in each iteration,** So overall in each iteration of this for loop it will take O(|V|) + O(|V|+|E|) = **O(|V|+|E|)** time.
  ⇒ Now, this loop will run for less than |V| times because shortest path between two vertex has to be less than equal to |V|-1 obviously.
  Therefore, time complexity for this loop will be **O(|V| . (|V| + |E|))**.

- So overall time complexity of the Algorithm will be:
  O(|V|+|E|) + O(|E|) + O(|V| . (|V| + |E|)) = **O(|V| . (|V| + |E|))**

Ans4(c):

Proof of Correctness:

- If edges except those are in a shortest path (of s & t) are destroyed, the shortest distance(of s & t) will not change. Hence after knowing a shortest path L, we can set all other entries M[u,v] to shortest distance. (And for simplicity of algorithm set entries corresponding to edges in L, also to shortest distance, which we will update later)
- **BFS()** is general bfs algorithm with just small change of not considering given edge (a,b). So it correctly gives required shortest distance.
- **BFS_L()** is again an extension of general bfs algorithm, which uses parent array to store the parent element from which an edge made path to the given argument(element in bracket).

  While loop at the end, backtracks the vertices which were used to reach t from s using parent array, by adding edges between parent and child vertices to a new adjacency list.
- Now, the second for loop in **Solve_Batman()** using **BFS()** gets the length of shortest path if given edge is destroyed and it works fine as explained above. Then just update the corresponding entry in M.
- In overall, we've updated M for each $(u, v) \in E$, hence our Algorithm is fully correct.

Ans5(a):

Approach:

- Let maximum of distance from vertex s to any other vertex u ∈ V, equal to max_d,
- Now take p= min(max_d, 2k)
- Now take any vertex t from V_p, it will make the eqn **min(dist(u, s), dist(u, t)) ≤ k** for every vertex u ∈ V

Pseudo-code:

```
distance(u)= ∞; // initially for all u in V; stores distance from s)
T_finder (G,s,distance,n){
        CreateEmptyQueue(Q);
        distance(s)=0;
        Enqueue(s,Q);
        While(Not IsEmptyQueue(Q)){
                v= Dequeue(Q);
                For each neighbor w of v {
                        if(distance(w)== ∞){
                                distance(w)=distance(v)+1;
                                Enqueue(w,Q);
                        }
                }
        }
        max_d=0;
        For u in V {
                max_d= max(max_d,distance(u));
        }
        k=n/3;
        p = min(max_d, 2*k);
        For t in V {
                if(distance(t)==p) return t;
        }
}
```

Ans5(b):

Proof of Correctness:

- The condition $\forall\ i \geq 0 : u \in V_i , v \in V_{i+1} \Rightarrow (u, v) \in E$, implies there is an undirected edge between any two vertices of consecutive sets (levels) V_i and V_{i+1}, hence also for V_{i-1} and V_i.
- This implies there will always be a path of two edges between any two vertices from the sets V_i and V_{i+2}. And this will be their distance(shortest path length) otherwise if it had

been 0 or 1 then the vertice from V_{i+2} had been in V_i or V_{i+1} or V_{i-1} which isn't possible by defintion of V_i.

- Continuing similarly we can say between any two vertices from sets V_i and V_j , it will have a distance equal to |i-j|.
- If p = 2k,
  take any u ∈ V, suppose dist(u,s)>k

  ⇒ u ∈ V_i for some i s.t. *k < i <= max_d <= 3k-1* (as maximum 3k-1 edges can come between 3k vertices)
  ⇒ -k < i-2k <= k-1 < k
  ⇒ |i-2k|<k ⇒ dist(i,p)<k

  Else, (when max_d<2k, p= max_d)
  take any u ∈ V, suppose dist(u,s)>k

  ⇒ u ∈ V_i for some i such that *k < i <= max_d*
  ⇒ k -max_d < i- max_d <= 0
  ⇒ | i - max_d | < | max_d - k | < | 2k - k | = k
  ⇒ dist(i,p) < k

  ⇒ Hence any vertex t from V_p (p= min(max_d,2k)) satisfies condition **min(dist(u, s), dist(u, t)) ≤ k** for every vertex u ∈ V