1.Solution:

In this algorithm, the first one has to find the minimum element in the array and take the difference between one element from the array and the minimum element. And on each traversal, keep track of the difference.

Pseudocode:

```
Findmaxdiff():
1:int max_diff, min_element
#To store the maximum difference and the minimum element
2:min_element = arr[0] #Initializing min_element with the first element
3:max_diff = arr[1] - arr[0]
# Initializing max_element with the difference of first and second element, since in question it is given that arr[j]-arr[i] should be max and j > i.
4:for i=1 to n: #Here n is the size of the array.
5:    if arr[i]-min_element > max_diff:
#If the difference is greater than max_diff, update max_diff
6:            max_diff = arr[i] - min_element
7:    if arr[i] < min_element:
#If the current element is smaller than the min_element, update min_element
8:            min_element = arr[i]
9:return    max_diff
```

Time Complexity:

Since the loop in it traverses the array only once, hence the run time complexity will be equal to the size of the array, which is O(n).

2. Solution:

Let the maximum clique found, denoted as C_m, as a lower bound of $\omega(G)$. Let C denote the current clique, and $P = \Lambda(C)$ denote the candidate set from which a vertex will be selected for C to grow next. Suppose there is a function U(C, P) which returns an upper bound of $\omega(C \cup P)$, i.e., the size of the maximum clique that can be found by selecting any additional vertex in P to grow from C. If $U(C, P) > |C_m|$, it continues to grow C by selecting one vertex from P. Otherwise, it stops searching from the current C.

Pseudocode:

MaxClique (G)

1: $C_m \leftarrow \emptyset$, sort V in some specific ordering o

initializes C_m as $\emptyset$ and sorts vertices in V in some specific ordering

2: **for** i = 1 to n according to o **do**

3: $C \leftarrow \{v_i\}$, $P \leftarrow \{v_i+1, \ldots, v_n\} \cap \Gamma(v_i)$

booktracking search from v_i considers only vertex set $\{v_i, v_i+1, \ldots, v_n\}$

4: sort P in some ordering o' # vertices in P are also sorted in some ordering

5: for $v_j \in P$ according to the sorting order o' do Clique(G, C, P, v_j)

branching from vi iteratively adds a candidate vertex v_j from P to C by invoking the procedure Clique(G, C, P, v_j), updating C_m or pruning fruitless branches

6: **end for**

7: **return** C_m #After processing all vertices, the resulting C_m by processing all vertices in V is the maximum clique of G

8: **Procedure** Clique(G, C, P, v_j)

9: $C \leftarrow C \cup \{vj\}$, $P \leftarrow P \cap \Gamma(vj)$# updates C and P by adding v_j to C and condensing P through setting $P \leftarrow P \cap \Gamma(v_j)$

10: **if** $P = \emptyset$ then

11: **if** $|C| > |C_m|$ then $C_m \leftarrow C$

if $P = \emptyset$, a maximal clique is found, and C_m will be updated if $|C| > |C_m|$

12: **else if** $U(C, P) > |C_m|$ **then**

13: sort P in some ordering o'

14: **for** $v_k \in P$ according to the sorting order o"**do** Clique(G, C, P, v_k)

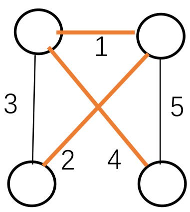
compares the lower bound $|C_m|$ with the upper bound U(C, P). If $U(C, P) > |C_m|$, it branches from vertices in P recursively

15: **end if**

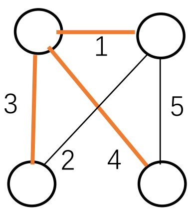
3. Solution:

To see that the MST is unique, observe that since the graph is connected and all edge weights are distinct, then there is a unique light edge crossing every cut. Assuming there are two MSTs, one is called T, and another one is called T'. For any e in T, if we delete e from T, then T becomes unconnected, and we'll have a cut(S, V - S). Edge e is the light edge through cut(S, V - S). If there's an edge x is in T' and through a cut(S, V - S), then x is also a lightweight. Since the light edge is unique, so e and x are the same, e is also in T'. Since we choose e at random, of all edges in T, also in T'. As a result, the MST is unique.

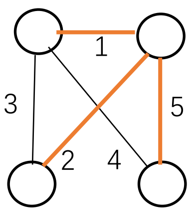
In order to see that the second-best minimum spanning tree doesn't need to be unique, here we have a weighted, undirected graph with a unique MST of weight 7 and two second-best minimum spanning trees of weight 8, as follows:



Minimum spanning tree



Second-best minimum spanning tree



second-best minimum spanning tree

4. Solution:

Since we need a polynomial algorithm, thus we can use the Breadth-first search (BFS) algorithm to find the shortest path.

Pseudocode:

Given a weighted directed graph G = (V, E) and node s ∈ V (vertex s (the start vertex))

```
BFS(s)
Mark all vertices as unvisited
Initialize search tree T to be empty
Mark vertex s as visited
set Q to be the empty queue
enqueue(s)              # enqueue: Adds an element to the end of the list
while Q is nonempty do
    u = dequeue(Q) # dequeue: Removes an element from the front of the list
    for each vertex v ∈ Adj(u)
        if v is not visited then
            add edge (u, v) to T
            Mark v as visited and enqueue(v)
return
```

Time Complexity:

Since the sum of the degrees in a graph is O(m), this gives an overall running time of O(m + n).