Problem # 1

a) Before visiting any vertex, we have initialized the distance of source vertex G has zero and the rest of the vertices as infinity. Then, we visit the source vertex first as it hasn't been visited yet and has the least distance (from the source vertex). Since G is connected to E and F, we update the distances of E and F (from the source vertex). Then, we add G to the set of vertices already visited so that we don't visit G again.

Next, we pick vertex E to visit as it has the least distance. This time, we update the distance of B, C and D. Although vertex E is connected to vertex F, we do not update the distance of F as 9+8>11 and we only update the distance if it is lesser. We then add vertex E to the set of vertices already visited.

We continue for all other vertices until the set of vertices visited contains all vertices and the final distances are the shortest distances from the vertices to the source vertex G.

Node visited	G	A	B	C	D	E	F
Start	0	Infinity	Infinity	Infinity	Infinity	Infinity	infinity
G	0	Infinity	Infinity	Infinity	Infinity	9	11
E	0	Infinity	9+7=16	9+5=14	9+15=24	9	11
F	0	Infinity	16	14	11+6=17	9	11
C	0	Infinity	16	14	17	9	11
B	0	16+7=23	16	14	17	9	11
D	0	17+5=22	16	14	17	9	11
A	0	22	16	14	17	9	11

b) When visiting node E, we encounter a problem as the distance from the source vertex G to the vertex F becomes 9+(-8) = 1 which is wrong. And in the end, this wrong distance is shown to be the correct one and the program thinks that it should take the path: G->E->F to reach F when it should go directly from G to F.

Dijkstra's algorithm is designed to work correctly when all edge weights are non-negative as it is an algorithm of greedy nature. When the weights are non-negative, the algorithm makes sure that when the shortest path is found, it never changes (from G to F). But when the edge weights are negative, they affect the distance values by decreasing the distances iteratively and produce incorrect results (distance from G to F becoming 1 by replacing 11).

Node visited	G	A	B	C	D	E	F
Start	0	Infinity	Infinity	Infinity	Infinity	Infinity	infinity
G	0	Infinity	Infinity	Infinity	Infinity	9	11
E	0	Infinity	9+7=16	9+5=14	9+15=24	9	9-8=1

F	0	Infinity	16	14	11+6=17	9	1
C	0	Infinity	16	14	17	9	1
B	0	16+7=23	16	14	17	9	1
D	0	17+5=22	16	14	17	9	1
A	0	22	16	14	17	9	11

As more edges become negative, the distances from the source vertex G to the vertex updates which causes Dijkstra's algorithm to output the wrong paths and distances. Above, you can see how changing EF from +8 to -8 is altering the distance of F from the source vertex G wrongly.

c) Before the start of each iteration of the main loop in Dijkstra's algorithm, the set of vertices with the shortest distance from the source vertex form the subspace for the set of vertices that are visited so far. The distances assigned to all vertices represent the shortest distance from the source vertex to the set of those vertices.

Initialization stage: At the start of the algorithm, the set of vertices visited is empty and the distances for all vertices are set to infinity except for the source vertex whose distance is set to zero and the distance from the source vertex to itself will always be zero. The loop invariant holds trivially at the start.

Maintenance stage: During each iteration of the main loop, we choose to vertex with the least distance that has not been visited and we count the distance from that vertex to its outgoing edges and we update the tentative distance of adjacent vertices if a shorter path is found. This ensures that that the set of vertices which have been visited grows iteratively and the distances remain accurate.

Termination: The main loop terminates when all the vertices have been visited and when the distance from each vertex to the source vertex is accurately known.

Proof:

During the initialization stage, all distances are set to infinity while the distance from the source vertex to itself (G) is initialized to zero.

Then, vertex E is picked as it has the least distance directly from vertex G and it is added to the set of vertices visited. This is done for all other vertices until the final vertex A is also visited.

Once all the vertices are visited, the loop terminates and the distance from each to the source vertex G is known to be accurate.

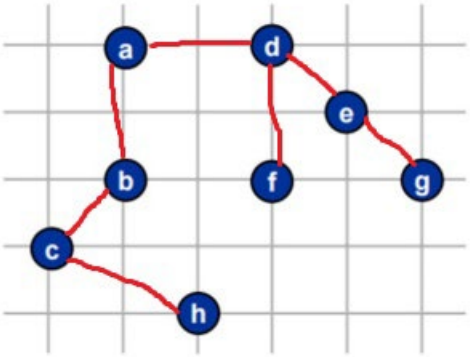
Problem # 2

a) The Kruskal's Algorithm sorts all the edges based on their weights hence it takes O(ElogV) time due to adding all vertices in an efficient binary manner into the minimum spanning tree.

The Prim's Algorithm starts with an arbitrary vertex and iteratively grows the minimum spanning tree by adding the smallest edge connecting the current tree to a vertex that is not yet present in the tree.
A priority queue is used usually which is implemented with a binary hence the time taken is O(E+VlogV).

In a dense graph, the number of edges approach the maximum number of edges which are V(V-1)/2. Hence, E~V^2 approximately. For Kruskal's algorithm, if we replace E with V^2, we get O(V^2 * logV). For Prim's algorithm, if we replace E with V^2, we get O(V^2 + VlogV)=O(V^2). From this, we can see that Prim's algorithm has a better time complexity and hence it will determine and output the minimum spanning tree more quickly due to it taking an arbitrary vertex and constructing the minimum spanning tree.

b)



The minimum-weight spanning tree has been shown above.
The algorithm used is Prim's algorithm.

First, we take a vertex randomly so let's take b. From b, we go to the vertex with the least distance which is sqrt (2) (edge weight) so we visit c.
After c, we have three options, go to a from b (distance is 2), go to f from b (distance is 2) or go to h from c (distance is sqrt (5)). We will choose to go to a from b as it has the least distance.
From a, we will go to d with a distance of 2.
Then we will go to e from d with a distance of sqrt (2).
Next, we will visit g from e with a distance of sqrt (2) again.

After that, we will visit f from d as it has a distance of 2.
Finally, we will visit the last vertex h from c with a distance of sqrt (5).

Total distance = sqrt (5) + sqrt (2) + 2 + 2 + 2 + sqrt (2) + sqrt (2)
Total distance = 12.5 (correct to one decimal place)

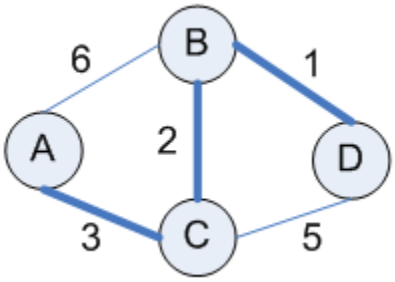
Iteration number	Node added	From vertex	Distance
1	b	First node	0
2	c	b	Sqrt (2)
3	a	b	2
4	d	a	2
5	e	d	Sqrt (2)
6	g	e	Sqrt (2)
7	f	d	2
6	h	c	Sqrt (5)

c) dist(h,c) = dist(h,b) = dist(h,f) = sqrt (5)
dist(b,f) = dist(d,f) = 2
dist(d,e) = dist(f,e) = sqrt (2)

Now, we can see that we can visit vertex h from either vertex c, b or f and the distance will be the same (which is sqrt 5). Meaning these three paths are three possible combinations that could be present in the minimum-weight spanning trees.
In a similar manner, there are two ways to visit vertex f (from vertex b or d) and two ways to visit vertex e (from vertex d or f).
Hence, if we combine all these ways and find the total combination aka the total number of minimum-weight spanning trees so it will become 3*2*2 = 12.

d) By definition, a minimum-weight spanning tree of a graph is the spanning tree with the smallest sum possible sum of edge weights (least distance). We will use a proof by contradiction to solve this question:
First, we will assume that G has more than one minimum-weight spanning tree (which is a contradiction to the initial knowledge given in the question).
Then, we could say that T1 and T2 are two distinct minimum-weight spanning trees of G.
Since T1 and T2 are distinct, there must be an edge e which is present in T1 but not in T2.
If we add that edge e to T2, a cycle is formed in T2 hence it is a connected graph and no longer a tree (by the definition of a spanning tree).
As T2 does not remain a tree after adding an edge, we can say that a second tree cannot exist.

Therefore, there cannot be two distinct minimum-weight spanning trees hence the proof is complete.

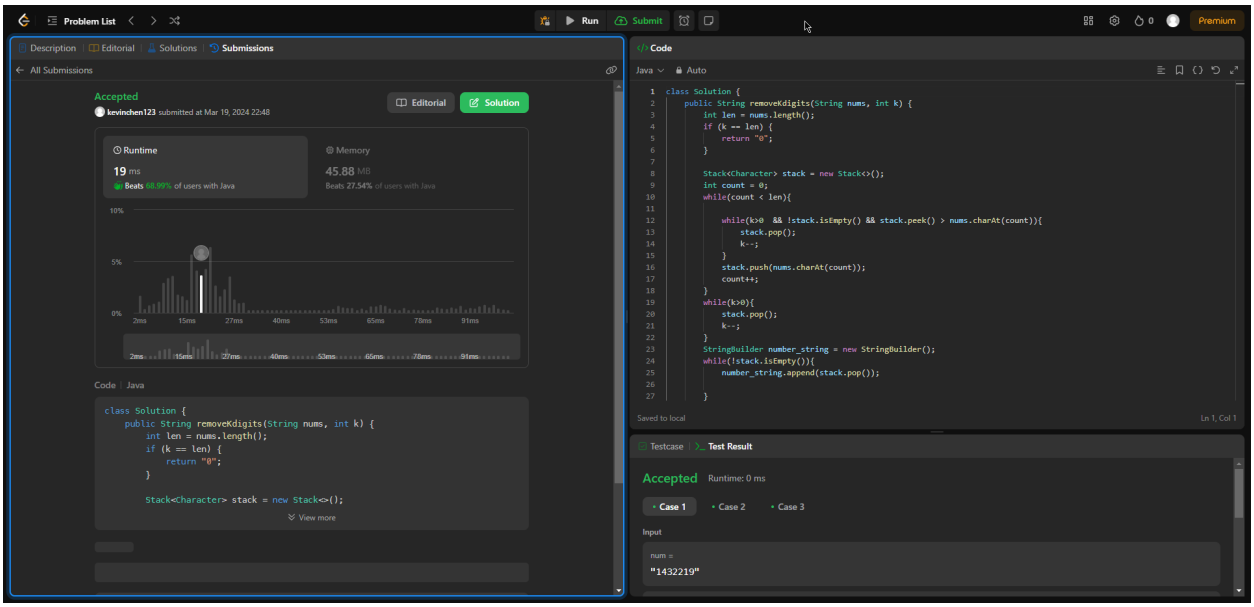


In the above figure, if we ignore the current edge weights, we could see that T1 and T2 could not exist together.
If T1 is made using edges AC, BC and BD and T2 is made using AB, BC, BD. If we try adding edge AC to T2 (since it is in T1 but not T2), T2 will become a connected graph and hence not a tree.

Problem # 3

a)

i) Problem Number: 402
Problem Title: Remove K Digits
Difficulty Level: Medium
Screenshot:

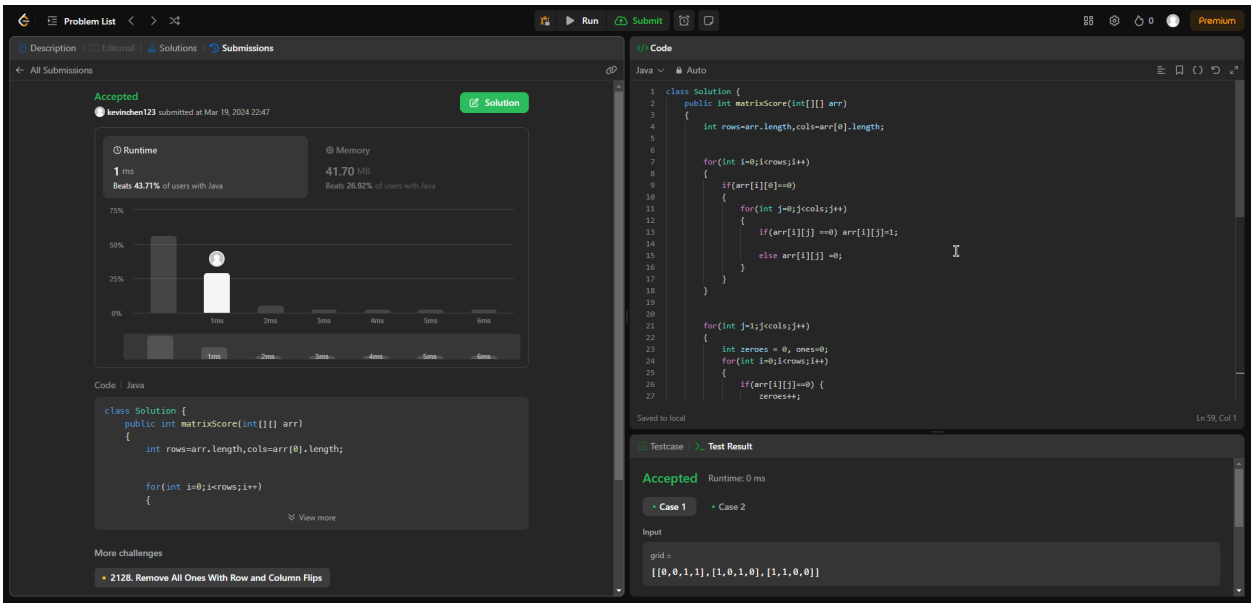


Code:

```
class Solution {
    public String removeKdigits(String nums, int k) {
        int len = nums.length();
        if (k == len) {
            return "0";
        }

        Stack<Character> stack = new Stack<>();
        int count = 0;
        while(count < len){

            while(k>0  && !stack.isEmpty() && stack.peek() > nums.charAt(count)){
                stack.pop();
                k--;
            }
            stack.push(nums.charAt(count));
            count++;
        }
        while(k>0){
            stack.pop();
```

```
            k--;
        }
        StringBuilder number_string = new StringBuilder();
        while(!stack.isEmpty()){
            number_string.append(stack.pop());

        }
        number_string.reverse();

        while(number_string.length() > 1 && number_string.charAt(0) =='0'){
            number_string.deleteCharAt(0);

        }
        return number_string.toString();

    }
}
```

j) Problem Number: 861
Problem Title: Score After Flipping Matrix
Difficulty Level: Medium
Screenshot:



Code:

```
class Solution {
    public int matrixScore(int[][] arr)
    {
        int rows=arr.length,cols=arr[0].length;
```

```
for(int i=0;i<rows;i++)
{
    if(arr[i][0]==0)
    {
        for(int j=0;j<cols;j++)
        {
            if(arr[i][j] ==0) arr[i][j]=1;

            else arr[i][j] =0;
        }
    }
}


for(int j=1;j<cols;j++)
{
    int zeroes = 0, ones=0;
    for(int i=0;i<rows;i++)
    {
        if(arr[i][j]==0) {
            zeroes++;
        } else {
            ones++;
        }
    }

    if(zeroes>ones)
    {
        for(int i=0;i<rows;i++)
        {
            if(arr[i][j]==0) arr[i][j]=1;
            else arr[i][j]=0;
        }
    }
}

int sum=0,k=0;

for(int i=0;i<rows;i++)
{
    k=1;
    for(int j=cols-1;j>=0;j--)
    {
        sum += arr[i][j] * k;
```

```
            k*=2;
        }
    }

    return sum;
  }
}
```

b) For the problem “Remove K Digits”, I faced several problems. Firstly, I struggled with trying to remove the leading zeroes without affecting the number’s value. I also did not understand initially how I could efficiently remove the specified number of digits (in the parameter) and also maintain the smallest resulting number at the same time.

Eventually, I brainstormed the idea of using the stack data structure to push and pop digits while iterating. The LIFO (last-in first-out) property of the stack helped me immensely in keeping track of all the digits in the initial number string.

As for my learning, I learnt how to effectively implement edge cases properly and I also learnt how to leverage the LIFO property of the stack to push and pop data from the stack in a very specific manner that is helpful in many cases. Also, brainstorming helped me in understanding how to find alternative solutions which are more efficient.