Problem 1

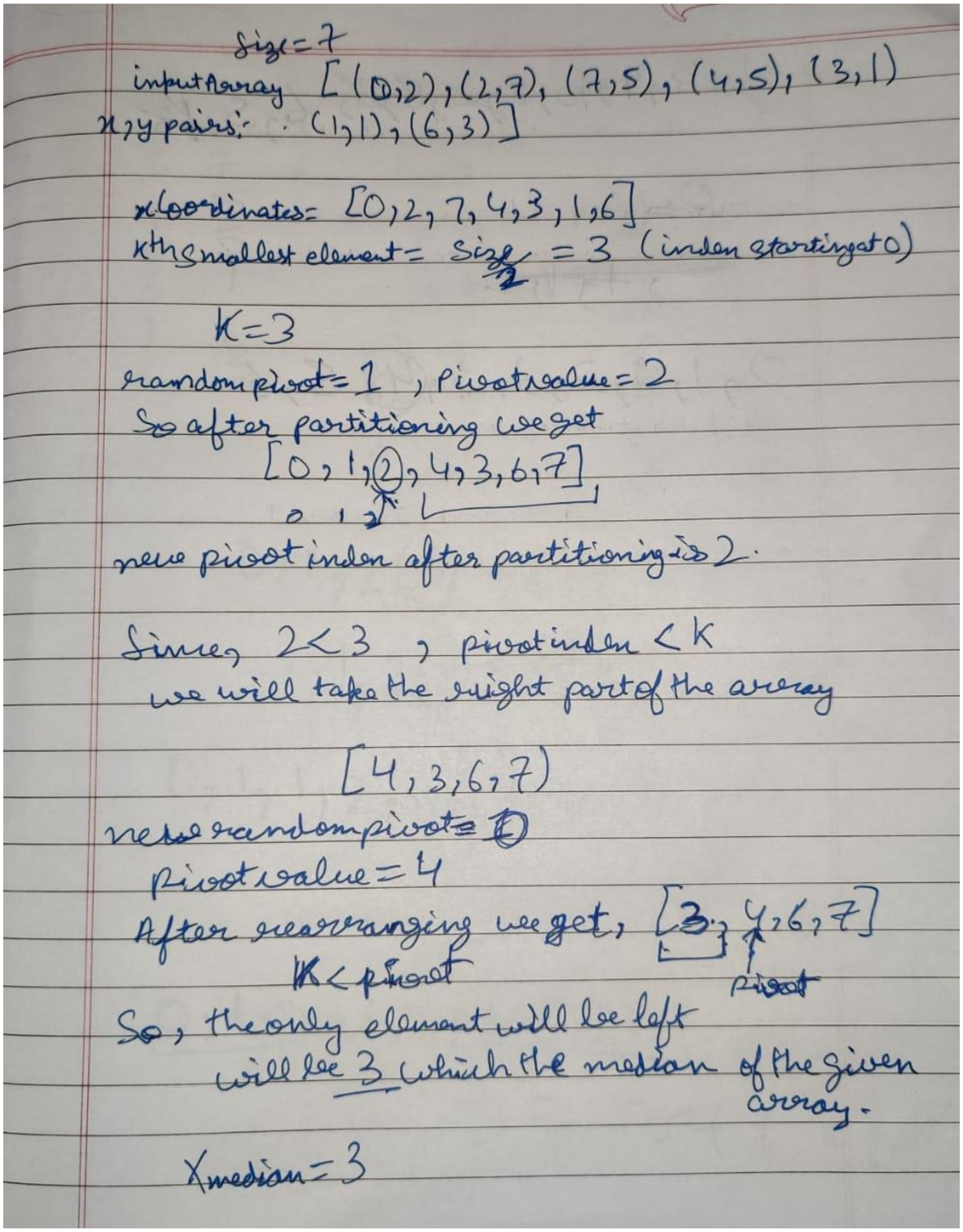
Description:

The problem given was to find the ideal location where we can open the new donut store. The ideal location was described by the location, which was closest to all the police stations, for which the coordinates were provided. The ideal location could be found by taking the median of the X coordinates and the Y coordinates of the provided police station coordinates. The mean was calculates using the select algorithm and which provided us with the ideal coordinates for the store. The task was to calculate the total distance of the police stations from the ideal point which is equal to the sum of the absolute difference between the x and y coordinates of the ideal points with all the police stations.

Pseudo Code:

```
distance(coordinate a, coordinate b):
    return absolute value of(a.x-b.x) + absolute value of (a.y-b.y)
getStoreLocation(xCoordinates[], yCoordinates[]):
    size= length of array
    xCoord= selectMedian(xCoordinates, size/2, 0, size-1)
    yCoord= selectMedian(yCoordinates, size/2,0,size-1)
    return Coordinate(xCoord, yCoord)
swap(input[], a, b):
    temp=input[a]
    input[a]=input[b]
    input[b]=temp
partition(input[], start, end, k):
    pivot= input[ k ]
    swap(input,k,end)
    storeIndex= start
    for i from start till end:
        if input[i] <pivot:
            swap(input,storeIndex,i)
            storeIndex++
    swap(input, storeIndex, end)
    return storeIndex
selectMedian(input[], k, start, end):
    if start==end:
        return input[start]
    pivot = randomInRange(start, end)
    pivot=partition(input, start, end, pivot)
    if k==pivotIndex:
        return input[k]
    else if k<pivotIndex:
        return selectMedian(input, k, start, pivotIndex-1)
    else:
        return selectMedian(input, k, pivotIndex+1, end)
randomInRange(min, max):
    return Math.random()*(max-min+1)+min
findTotalDistance(input[] ,storeLocation ):
    sum=0
    for i till length of input:
        sum+= distance(input[i], storeLocation)
    return sum
Main:
    Input with scanner
    Size= number of coordinates in the file
    Input[] = array of type Coordinate a static class that stores both x and y
    xCoords= new array of length size
    yCoords= new array of length size
    storeLocation= getStoreLocation(xCoords, yCoords)
    print(findTotalDistance(input, storeLocation)
```

Sketch:



size=7

inputArray [(0,2),(2,7),(7,5),(4,5),(3,1)

x,y pairs: (1,1),(6,3)]

xcoordinates= [0,2,7,4,3,1,6]

kth smallest element = $\frac{size}{2}$ = 3 (index starting at 0)

K=3

random pivot=1 , Pivot value=2

So after partitioning we get

[0,1,2,4,3,6,7]

0 1 2

new pivot index after partitioning is 2.

Since, 2<3 , pivot index < K

we will take the right part of the array

[4,3,6,7)

new random pivot= 1

Pivot value=4

After rearranging we get, [3, 4,6,7]

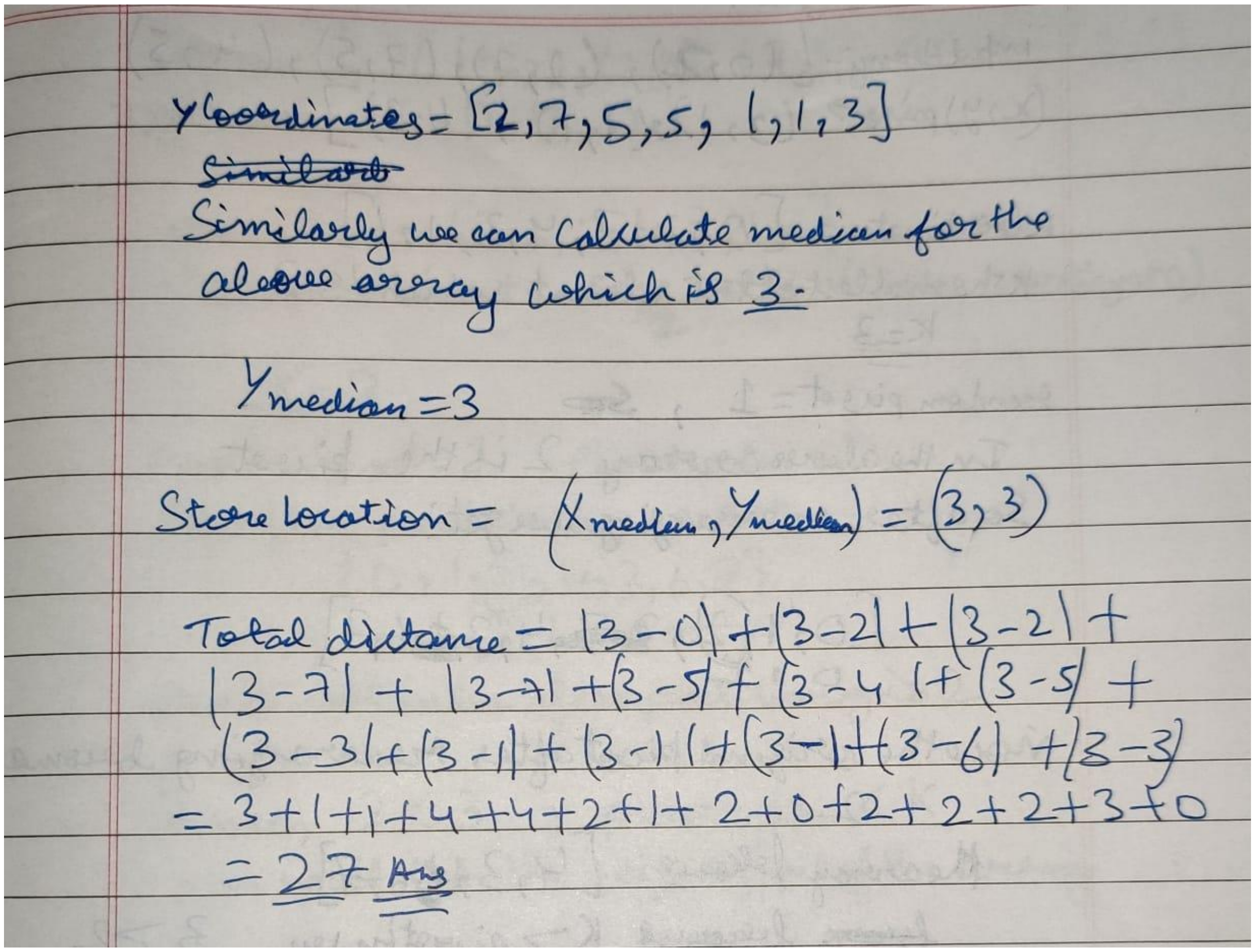
K< pivot

pivot

So, the only element will be left

will be 3 which the median of the given array.

Xmedian=3



Y coordinates = [2, 7, 5, 5, 1, 1, 3]

~~Similarly~~

Similarly we can calculate median for the above array which is 3.

$Y_{median} = 3$

Store location = $(X_{median}, Y_{median}) = (3, 3)$

$$\text{Total distance} = |3-0| + |3-2| + |3-2| + |3-7| + |3-7| + |3-5| + |3-4| + |3-5| + |3-3| + |3-1| + |3-1| + |3-1| + |3-6| + |3-3|$$

$$= 3+1+1+4+4+2+1+2+0+2+2+2+3+0$$

$$= \underline{27}\ \underline{\text{Ans}}$$

Running Time estimate:

- Initially for taking the input from the file is O(n)
- For calculating the location of the store, first calculate the median of all the x values and median of y values which gives us the location of the store.
- Median calculation is using select algorithm which is O(n)
- So, getting the store location was 2*O(n)
- Calculating the total distance from the store location to all the police stations is again O(n)

So the total complexity is the sum of all the complexities which is simply: O(n) + 2*O(n) + O(n) so which is 4*O(n) and 4 is just a constant.

Final complexity: O(n)

Problem 2

Pseudo-Code:

```
longestSubsequence(int[] arr)
    maxLen =0
    for i=1 to length of arr:
        k=arr[i]
        count=1
        for j=i+1 to length of arr:
            if(arr[j] > k)
                k=arr[j]
                count++
        maxLen=max(len,count)
    return maxLen
```

Running Time Estimate:

The running time can be estimated as follows:

- The first loop is iterating over all the values from index 1 till the length which is O(n).
- Inside that loop, we have another loop that iterates from the current index of i , till the length and adds all values if they are greater than the previous value k.

The total complexity for the algorithm comes out to be O(n*n)

Counter Example:

The provided algorithm will not work for the following example:
[5, 10, 15, 6, 7, 8]

If we consider the above examples and if we apply the algorithm, we get the following permutations of the longest sequence:

[5,10,15] len=3

[10,15] len =2

[15] len =1

[6,7,8] len =3

[7,8] len =2

[8] len =1

According to the algorithm the solution would just be either [5,10,15] or [6,7,8] but it missed the actual solution which is [5,6,7,8] of length=4

Problem 3

Table for the execution times:

Time is in milliseconds

N	incrSubseqRecursive()	incrSubseqDP()
10	28.2	15.4
20	56.8	9.4
30	94.1	21.8
100	131558.6	161.7
200	Timeout	499.9
400	Timeout	1517.7
500	Timeout	2162.2
1000	Timeout	3738.9
10000	Timeout	136986
100000	Timeout	13179642

The table given above has the execution times for 10 random inputs of given size N and it clearly shows that incrSubseqDP is exponentially faster than incrSubseqRecursive(). Recursive is a brute force approach and has an overall complexity of O(2^n) whereas DP has a complexity of O(n*n) which is clearly faster than 2^n. For values greater than 100 the program fails as the complexity reaches to 2^200 which is exponentially more and the program times out after that but the DP program still runs has very less runtime.