Q2: Let A[1::n] and B[1::n] be two arrays of distinct integers sorted in increasing order. Give an efficient algorithm (O(log n)) to find the median of the 2n elements in both A and B. Derive the time complexity. Implement the algorithm in any programming language.

ALGORITHM

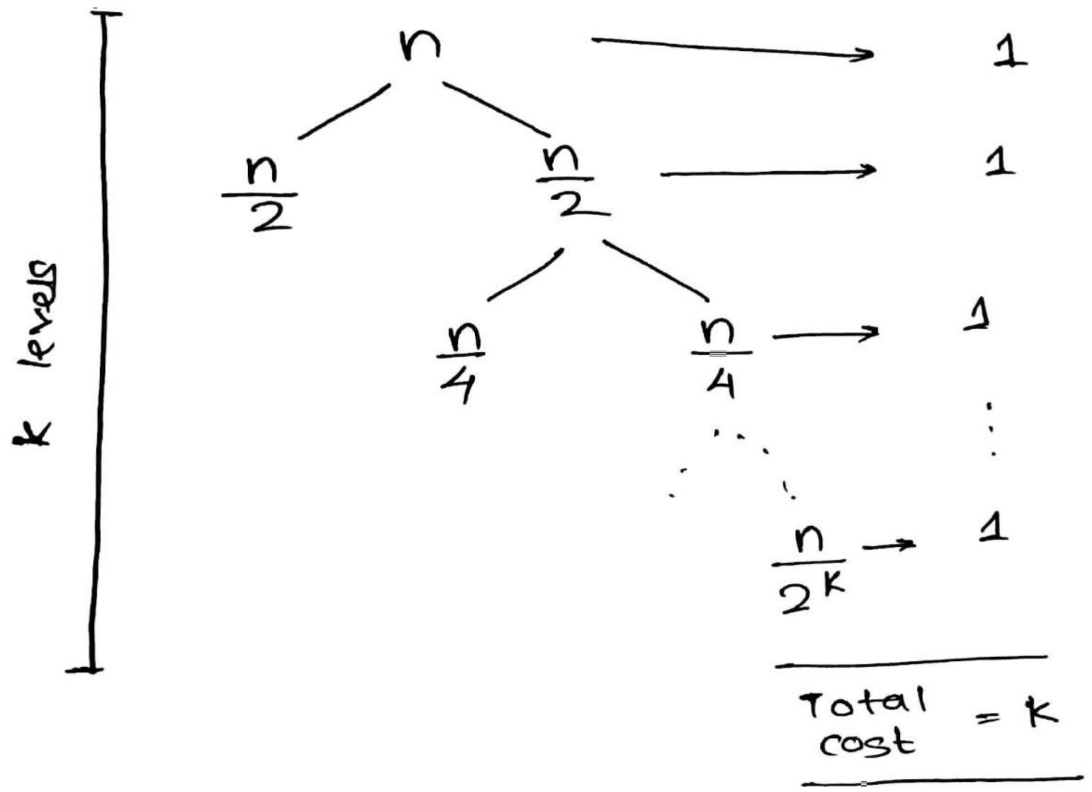
Input:	Two sorted arrays A [1…. n] and B [1….m]
Output:	Median of the n + m elements, median
Assumption:	The two arrays must be sorted in increasing order. Let n ≤ m, i.e. A is shorter.

Steps:

1. Start
2. Binary Search on shorter array. Set startA = 0 and endA = n
3. Partition both array at some point.
   $partitionA = \frac{(startA+endA)}{2}$
   $partitionB = \frac{(n+m-1)}{2} - partitionA$
   The partition must be such that number of elements in combined array of A and B must be equal on both sides.
4. A_1 is the maximum element on left partition of A and A_2 is the minimum element on the right partition of A. B_1 is the maximum element on the left partition of B and B_2 is the minimum element on the right partition of B.
   Note: If the left side of any partition contain no element, i.e. partitioning point is along the edge then we assume the element to be −INF and if same situation arises on the right side then take it to be +INF.
5. If $A_1 \leq B_2$ and $B_1 \leq A_2$, then partition is correct, i.e. all elements to left of partitioning point of A is smaller than all the elements to right of partitioning point of B and all elements to the left of the partitioning point of B is less than all the elements to the right of partitioning point of A.
6. A_1,A_2,B_1,B_2 are on the border of the partition. If partitioning is correct, Median must be among these elements.
   $if)$
   $\quad median = avg[max(A_1, B_1), min(A_2, B_2)]$
   $otherwise,$
   $\quad median = max\ (A_1, B_1)$
   [This is because left partitioning contains one extra element than the right which must be the median. GOTO Step 8.]
7. If the partition is not correct i.e. condition in Step 5 does not hold. Then we need to find whether partitioning point in A need to move left or right.
   If $A_1 > B_2$ i.e. all elements to the left of A is not less than all elements to right of B. Then $partitionA$ needs to move left. Hence, we search for median in left side of current partitioning point of A. Thus, the array reduces to$endA = partitionA - 1$; otherwise, partition has to shift right. We search for partitioning point in the right side of current partition of A. Thus, the array reduces to$startA = partitionA + 1$. GOTO Step 3.
8. PRINT median
9. Stop

Analysis:

At every iteration we look for proper partitioning in any one side of the array.
Thus, the recursion is$T(n) = T\left(\frac{n}{2}\right) + 1$; where $T\left(\frac{n}{2}\right)$ assumes each time, partitioning occurs at the middle and 1 refers to the cost of finding partition after sub-dividing is constant. Thus, we have,



Now, $\frac{n}{2^k} = 1$ implying $k = log_2 n$. Therefore, $T(n) = log_2 n = O(logn)$
Now we have assumed $n \leq m$, but if we remove this restriction, we always perform binary search on $min\ (m, n)$. Thus, running time becomes, $T(n) = O\left(log\left(min(m, n)\right)\right)$

PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
#include<bits/stdc++.h>

int main()
{
	int A[500],B[500],sizeA,sizeB,i,A1,A2,B1,B2;
	float median;

	printf("Enter size of array A: ");
	scanf("%d",&sizeA);

	if(sizeA <= 0 || sizeA > 500)
	{
		printf("Invalid Size. Enter a positive integer from 1 to 500.");
		exit(0);
	}

	printf("Enter elements of array A in ascending order: \n");
	for(i=0;i<sizeA;i++)
	{
		printf("Enter %d-th element: ",i+1);
		scanf("%d",&A[i]);
	}

	for(i=0;i<sizeA-1;i++)
	{
		if(A[i] > A[i+1])
		{
			printf("The input array should be in ascending order.");
```

```
                exit(0);
            }
    }

    printf("Enter size of array B  in ascending order: ");
    scanf("%d",&sizeB);

    if(sizeB <= 0 || sizeB > 500)
    {
            printf("Invalid Size. Enter a positive integer from 1 to 500.");
            exit(0);
    }

    for(i=0;i<sizeB;i++)
    {
            printf("Enter %d-th element: ",i+1);
            scanf("%d",&B[i]);
    }

    for(i=0;i<sizeB-1;i++)
    {
            if(B[i] > B[i+1])
            {
                printf("The input array should be in ascending order.");
                exit(0);
            }
    }

    int startA = 0;
    int endA = sizeA;

    int partitionA = (startA + endA)/2;
    int partitionB = ((sizeA + sizeB + 1)/2) - partitionA;

    if(partitionA-1 < 0)
            A1 = INT_MIN;
    else
            A1 = A[partitionA-1];

    if(partitionA >= sizeA)
            A2 = INT_MAX;
    else
            A2 = A[partitionA];

    if(partitionB-1 < 0)
            B1 = INT_MIN;
    else
            B1 = B[partitionB-1];

    if(partitionB >= sizeB)
            B2 = INT_MAX;
    else
            B2 = B[partitionB];

    while(true)
    {
            printf("start=%d            end=%d partitionA=%d A1=%d A2=%d partitionB=%d   B1=%d
    B2=%d\n",startA,endA,partitionA,A1,A2,partitionB,B1,B2);
            if(A1<=B2 && B1<=A2)
            {
```

```
        if(((sizeA + sizeB)%2) != 0)
        {
            median = (A1 >= B1)? A1 : B1;
        }
        else
        {
            median = (float)(((A1 >= B1)? A1 : B1) + ((A2 <= B2)? A2 : B2))/2;
        }
        break;
    }
    else if(A1 > B2)
    {
        endA = partitionA-1;

        partitionA = (startA + endA)/2;
        partitionB = ((sizeA + sizeB + 1)/2) - partitionA;

        if(partitionA-1 < 0)
            A1 = INT_MIN;
        else
            A1 = A[partitionA-1];

        if(partitionA >= sizeA)
            A2 = INT_MAX;
        else
            A2 = A[partitionA];

        if(partitionB-1 < 0)
            B1 = INT_MIN;
        else
            B1 = B[partitionB-1];

        if(partitionB >= sizeB)
            B2 = INT_MAX;
        else
            B2 = B[partitionB];
    }
    else
    {
        startA = partitionA+1;

        partitionA = (startA + endA)/2;
        partitionB = ((sizeA + sizeB + 1)/2) - partitionA;

        if(partitionA-1 < 0)
            A1 = INT_MIN;
        else
            A1 = A[partitionA-1];

        if(partitionA >= sizeA)
            A2 = INT_MAX;
        else
            A2 = A[partitionA];

        if(partitionB-1 < 0)
            B1 = INT_MIN;
        else
            B1 = B[partitionB-1];

        if(partitionB >= sizeB)
```

```
                                        B2 = INT_MAX;
                                else
                                        B2 = B[partitionB];
                        }
                }

                printf("Median = %0.2f",median);

                return 0;
}
```

OUTPUT

```
SET 1:
Enter size of array A: 3
Enter elements of array A in ascending order:
Enter 1-th element: 1
Enter 2-th element: 2
Enter 3-th element: 3
Enter size of array B: 5
Enter elements of array A in ascending order:
Enter 1-th element: 2
Enter 2-th element: 3
Enter 3-th element: 4
Enter 4-th element: 5
Enter 5-th element: 6
start=0 end=3 partitionA=1      A1=1    A2=2    partitionB=3    B1=4    B2=5
start=2 end=3 partitionA=2      A1=2    A2=3    partitionB=2    B1=3    B2=4
Median = 3.00

SET 2:
Enter size of array A: 5
Enter elements of array A in ascending order:
Enter 1-th element: 3
Enter 2-th element: 8
Enter 3-th element: 9
Enter 4-th element: 15
Enter 5-th element: 18
Enter size of array B: 4
Enter elements of array B in ascending order:
Enter 1-th element: 11
Enter 2-th element: 19
Enter 3-th element: 21
Enter 4-th element: 25
start=0 end=5 partitionA=2      A1=8    A2=9    partitionB=3    B1=21   B2=25
start=3 end=5 partitionA=4      A1=15   A2=18   partitionB=1    B1=11   B2=19
Median = 15.00
```

Q3: Let A[1::n] be an array of n integers and x an integer. Derive a divide and conquer algorithm to find the frequency of x in A, i.e the number of times x appears in A. What is the time complexity of your algorithm? Implement the algorithm in any programming language.

ALGORITHM

Input:	A sorted array A [1…. n] and an element x, whose frequency is to be calculated
Output:	Frequency of x, count
Assumption:	The two arrays must be sorted in increasing order.

Steps:

1. Start
2. Set count = 0
3. Perform Binary Search on A to find x. If x is found, Set count = 1 and ‘mid’ holds the position where x is found.
4. Check the left side of mid for x. For every occurrence of x, increment count by 1. Repeat this Step until an element not equal to x is found.
5. Check the right side of mid for x. For every occurrence of x, increment count by 1. Repeat this Step until an element not equal to x is found.
6. Print count
7. Stop

Analysis:

T(n) = Time Complexity of Binary Search + Number of Iterations on Left of Partition + Number of Iterations on Right of Partition

Frequency Count, count = Number of iterations on Left of Partition + Number of iterations on Right of Partition

Therefore, T(n) = O(log n) + count = O(log n + count)

PROGRAM

```
#include<stdio.h>
#include<stdlib.h>

int main()
{
        int A[500],sizeA,i,start,end,mid,number,count=0,position;
        float median;

        printf("Enter size of array A: ");
        scanf("%d",&sizeA);

        if(sizeA <= 0 || sizeA > 500)
        {
                printf("Invalid Size. Enter a positive integer from 1 to 500.");
                exit(0);
        }

        printf("Enter elements of array A in ascending order: \n");
        for(i=0;i<sizeA;i++)
        {
                printf("Enter %d-th element: ",i+1);
                scanf("%d",&A[i]);
        }

        for(i=0;i<sizeA-1;i++)
        {
                if(A[i] > A[i+1])
                {
                        printf("The input array should be in ascending order.");
                exit(0);
            }
        }

        printf("Enter the element to be searched: ");
        scanf("%d",&number);

        start = 0;
        end = sizeA-1;

        while(start <= end)
        {
            mid=(start+end)/2;

            if(A[mid] == number)
            {
                count = 1;
                break;
            }
            else if(number < A[mid])
                end = mid - 1;
            else
                start = mid + 1;
        }

        position = mid;
        while(A[position-1] == number)
        {
            count++;
            position--;
        }

        position = mid;
        while(A[position+1] == number)
        {
            count++;
            position++;
        }

        printf("Frequency of %d is: %d",number,count);

        return 0;
}
```

OUTPUT

```
SET 1:
Enter size of array A: 559
Invalid Size. Enter a positive integer from 1 to 500.

SET 2:
Enter size of array A: 10
Enter elements of array A in ascending order:
Enter 1-th element: 5
Enter 2-th element: 2
Enter 3-th element: 2
Enter 4-th element: 2
Enter 5-th element: 1
Enter 6-th element: 1
Enter 7-th element: 8
Enter 8-th element: 9
Enter 9-th element: 24
Enter 10-th element: 890
The input array should be in ascending order.

SET 3:
Enter size of array A: 20
Enter elements of array A in ascending order:
Enter 1-th element: 1
Enter 2-th element: 1
Enter 3-th element: 1
Enter 4-th element: 1
Enter 5-th element: 1
Enter 6-th element: 1
Enter 7-th element: 3
Enter 8-th element: 3
Enter 9-th element: 3
Enter 10-th element: 3
Enter 11-th element: 3
Enter 12-th element: 3
Enter 13-th element: 6
Enter 14-th element: 6
Enter 15-th element: 6
Enter 16-th element: 6
Enter 17-th element: 32
Enter 18-th element: 32
Enter 19-th element: 32
Enter 20-th element: 56
Enter the element to be searched: 3
Frequency of 3 is: 6
```

Q4: Derive the time complexity of Select algorithm if the group size is 9. Implement the RandomizedSelect. Count the number of elements in each recursion and number of total recursion in different runs of the RandomizedSelect varying the number of elements n and value of k. Plot graph to see how it matches with theoretical complexity bounds.

PROGRAM

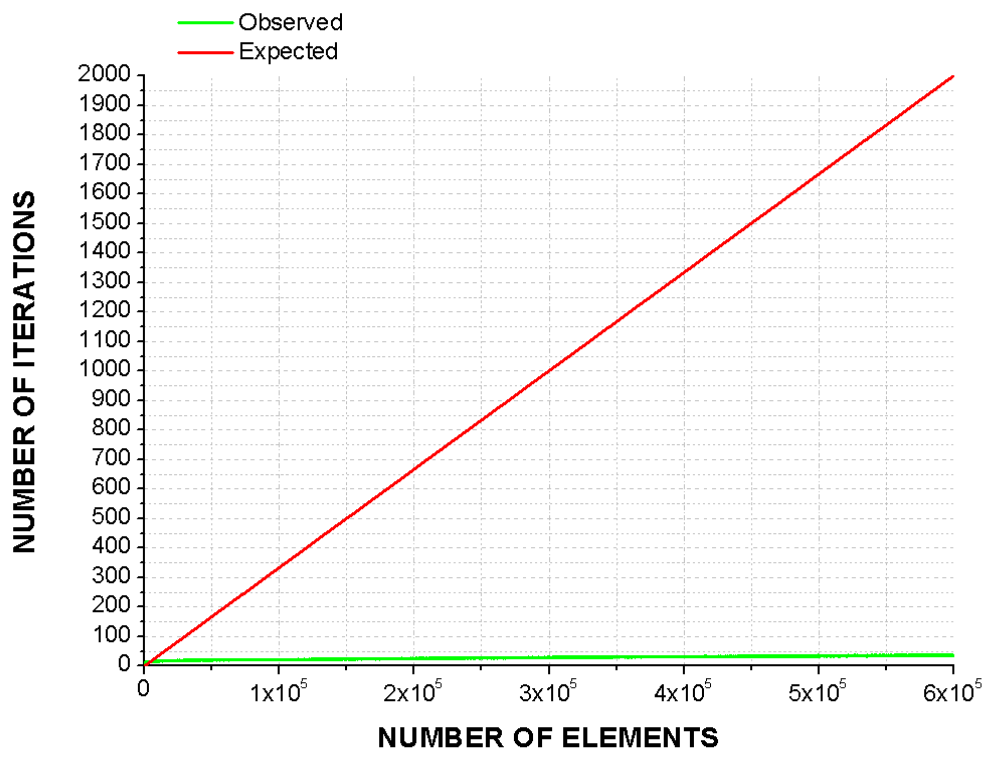
```
#include <stdio.h>
#include <stdlib.h>

int count;

int partition(int arr[], int low, int high)
{
	int pivot,i,j,temp;
	pivot = arr[high];
    i = (low - 1);

    for (j = low; j <= high-1; j++)
    {
        if (arr[j] < pivot)
        {
            i++;
```

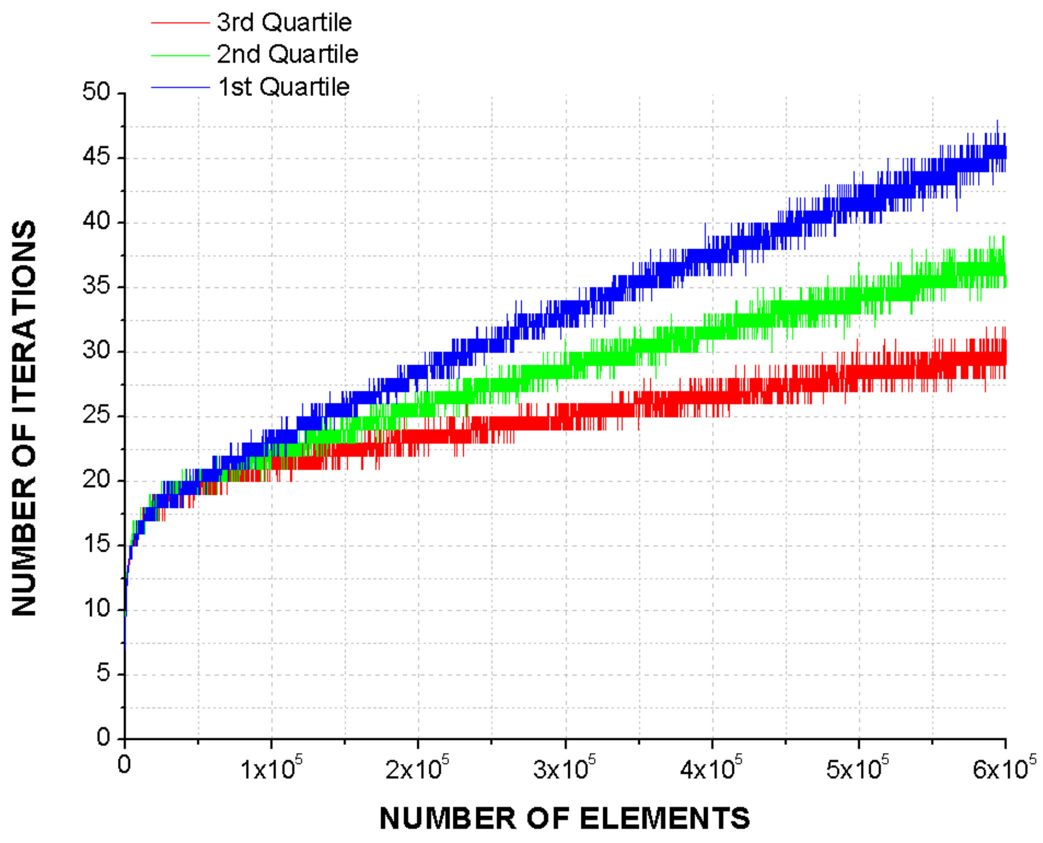
```
                                temp = arr[j];
                                arr[j] = arr[i];
                                arr[i] = temp;
        }
    }
            temp = arr[high];
            arr[high] = arr[i+1];
            arr[i+1] = temp;

    return (i+1);
}

int random_partition(int arr[], int low, int high)
{
            int n,i,temp;
            n = high - low + 1;
            i = low + (rand()%n);
            temp = arr[high];
            arr[high] = arr[i];
            arr[i] = temp;

            return partition(arr, low, high);
}

int random_select(int arr[], int low, int high, int k)
{
            int p,i;
            count = count + 1;
            if (low == high)
    {
            return arr[low];
    }

            p = random_partition(arr, low, high);
            i = p - low + 1;

            if(k == (i))
            {
                        return arr[p];
            }
            else if(k<i)
            {
        return random_select(arr, low, p - 1, k);
    }
    else
    {
        return random_select(arr, p+1, high, k-(i));
    }
}

int main()
{
            int *A,*count_1,*count_2,*count_3,*count_4,quartile3,quartile1,quartile2,sizeA,i,median,k,j;

            FILE *fptr;

            count_1 = (int*) malloc(1000*sizeof(int));
            count_2 = (int*) malloc(1000*sizeof(int));
            count_3 = (int*) malloc(1000*sizeof(int));
```

```
	fptr = fopen("varying__kth_element_1.txt", "w");
	if(fptr == NULL)
	{
		printf("Error!");
	   exit(1);
	}

	for(sizeA=100;sizeA<=600000;sizeA+=100)
	{
		for(j=0;j<1000;j++)
		{
			A = (int*) malloc(sizeA*sizeof(int));
			srand(rand());
			for(i=0;i<sizeA;i++)
			{
				A[i] = rand();
			}

			k = (sizeA+1)/4;
			count = 0;
			quartile1 = random_select(A,0,sizeA-1,k);
			count_1[j]=count;
			printf("\nIteration: %d-%d			1st		Quartile:		%d,		k:
%d",sizeA,(j+1),quartile1,k);

			k = (sizeA+1)/2;
			count = 0;
			quartile2 = random_select(A,0,sizeA-1,k);
			count_2[j]=count;
			printf("		2nd Quartile: %d, k: %d",quartile2,k);

			k = (3*(sizeA+1))/4;
			count = 0;
			quartile3 = random_select(A,0,sizeA-1,k);
			count_3[j]=count;
			printf("		3rd Quartile: %d, k: %d",quartile3,k);
		}

		count=0;
		for(i=0;i<1000;i++)
		{
			count=count+count_1[i];
		}
		count=count/1000;
		fprintf(fptr,"%d %d",sizeA,count);

		count=0;
		for(i=0;i<1000;i++)
		{
			count=count+count_2[i];
		}
		count=count/1000;
		fprintf(fptr,"	%d",count);

		count=0;
		for(i=0;i<1000;i++)
		{
			count=count+count_3[i];
		}
		count=count/1000;
```

```
                fprintf(fptr,"      %d\n",count);
        }

        fclose(fptr);

        return 0;
}
```

GRAPH PLOT


Observed
Expected
1x10^5
2x10^5
3x10^5
4x10^5
5x10^5
6x10^5
NUMBER OF ITERATIONS
NUMBER OF ELEMENTS


3rd Quartile
2nd Quartile
1st Quartile
1x10^5
2x10^5
3x10^5
4x10^5
5x10^5
6x10^5
NUMBER OF ITERATIONS
NUMBER OF ELEMENTS