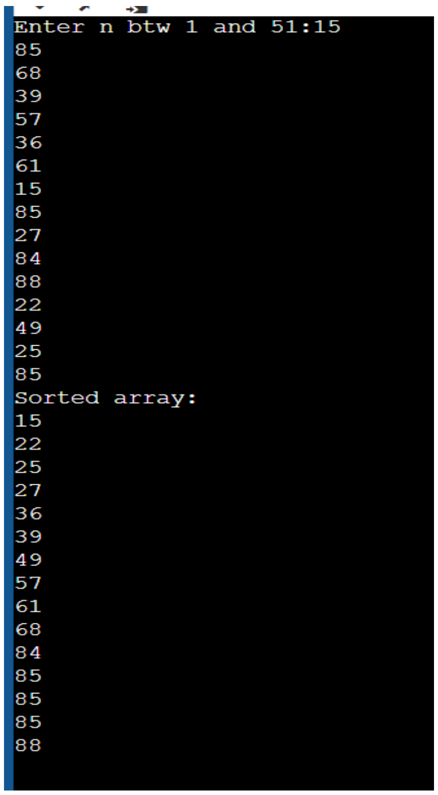
FCE
Homework 2
Sorting and Searching

1. Merge sort algorithm: sentinel value is fixed as the maximum value of an integer using limits header file.

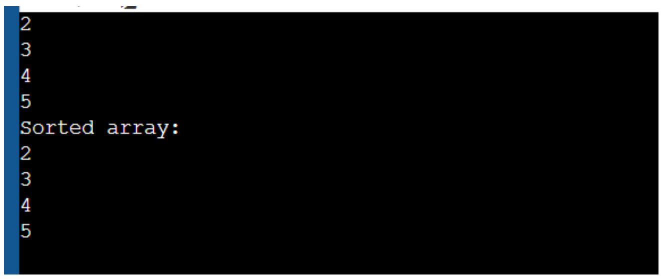
a. The user is asked to input the value between 1 and 51
b. The array is generated randomly in the range of numbers 0 to 100 using cstdlib header file, and rand function.
c. Mergesort function is sorted recursively and merge function is called with an array a, updated p, q, and r values. Sentinels are used in both sub-arrays to reduce the number of operations.
d. the picture below shows the array before and after sorting with n=15



```
Enter n btw 1 and 51:15
85
68
39
57
36
61
15
85
27
84
88
22
49
25
85
Sorted array:
15
22
25
27
36
39
49
57
61
68
84
85
85
85
88
```

e. The merge function is altered by adding one statement that checks if the array is already sorted then it skips the merge process and gives the output. To check that it doesn't go into the merge process, a print statement is added to check if it prints then it follows the whole merge function and if it doesn't print anything then it comes out of that function. This is tested by passing a sorted array. If the mid-index value of array a is less than the adjacent value then the array is sorted.

```
f.          if (a[q]<=a[q+1])
              1. return;
g.          cout<<"value of q:"<<q<<endl;
```



```
2
3
4
5
Sorted array:
2
3
4
5
```

2. merge sort without sentinels: algorithm of merge sort without sentinels is given in the picture below.

1. Two new sub-arrays are created with sizes n1 and n2, extra space is not added to store sentinels
2. The value of k runs from k=p to r; first, while loop runs till i reaches n1 and j reaches n2 checking for the lesser values in both sub-arrays and editing those values in the original array
3. i and j are incremented until n1 and n2 then k is incremented after each if-else condition.
4. The remaining values of the array(if any left in the sub-arrays) are added to the original array by the second and third while loop.

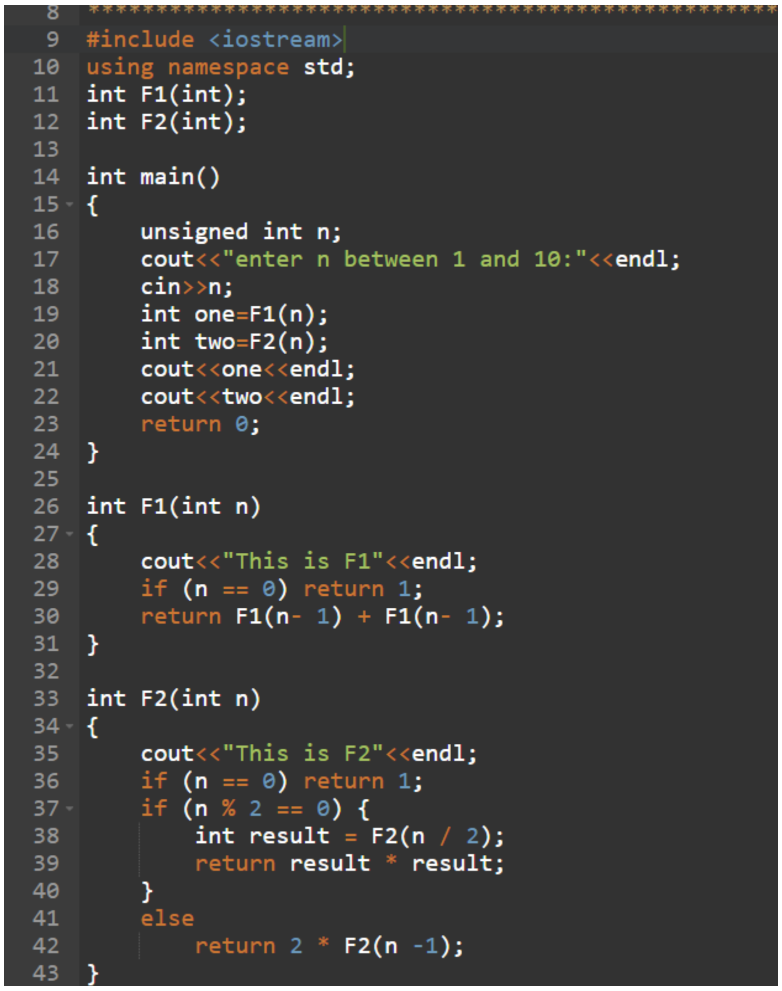
```
2) Merge-sort (A, p, r)
   1.   if  p<r
            q = ⌊(p+(r-p))/2⌋      → To avoid overflow for larger values
            Merge-sort (A, p, q)
            Merge-sort (A, q+1, r)
            Merge (A, p, q, r)
```

```
   Merge (A, p, q, r)

1.  n1 = p-q+1
2.  n2 = r-q
3.  int  L[n1]  and R[n2]  be  2 new arrays
4.  for i=1 to n1
5.       L[i] = A [p+i-1]
6.  for j=1 to n2
7.       R[j] = A [q+j]
8.  int  i=j=1
9.  int  k=p
10.      while  i<n1  and  j<n2
11.          if  L[i]  ≤  R[j]
12.              A[K]  = L[i]
13.              i → i+1
14.          else
15.              A[K] = R[j]
16.              j → j+1
             K → K+1
17. while  i<n1                      → Remaining elements in array (if any)
18.      A [K] = L[i]
19.          i → i+1 , K → K+1
20. while  j<n2
21.      A[K] = R[j]
22.          j → j+1 , K → K+1
```

#q=p+(r-p)/2

3. Two functions F1 and F2 are given as-

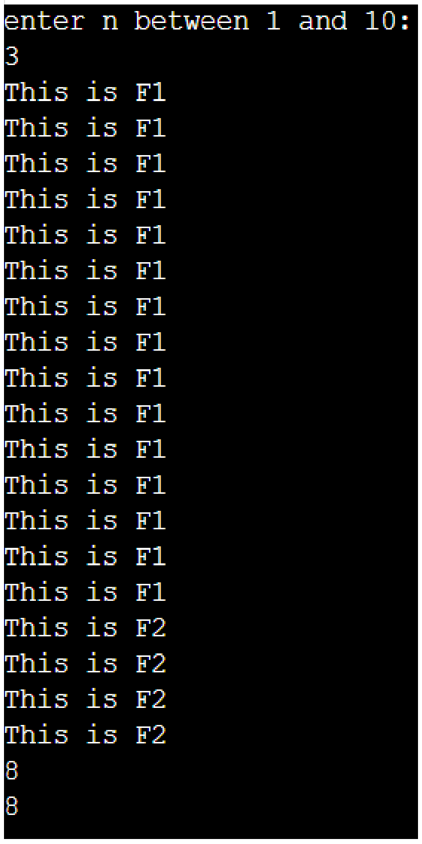


```cpp
#include <iostream>
using namespace std;
int F1(int);
int F2(int);

int main()
{
    unsigned int n;
    cout<<"enter n between 1 and 10:"<<endl;
    cin>>n;
    int one=F1(n);
    int two=F2(n);
    cout<<one<<endl;
    cout<<two<<endl;
    return 0;
}

int F1(int n)
{
    cout<<"This is F1"<<endl;
    if (n == 0) return 1;
    return F1(n- 1) + F1(n- 1);
}

int F2(int n)
{
    cout<<"This is F2"<<endl;
    if (n == 0) return 1;
    if (n % 2 == 0) {
        int result = F2(n / 2);
        return result * result;
    }
    else
        return 2 * F2(n -1);
}
```



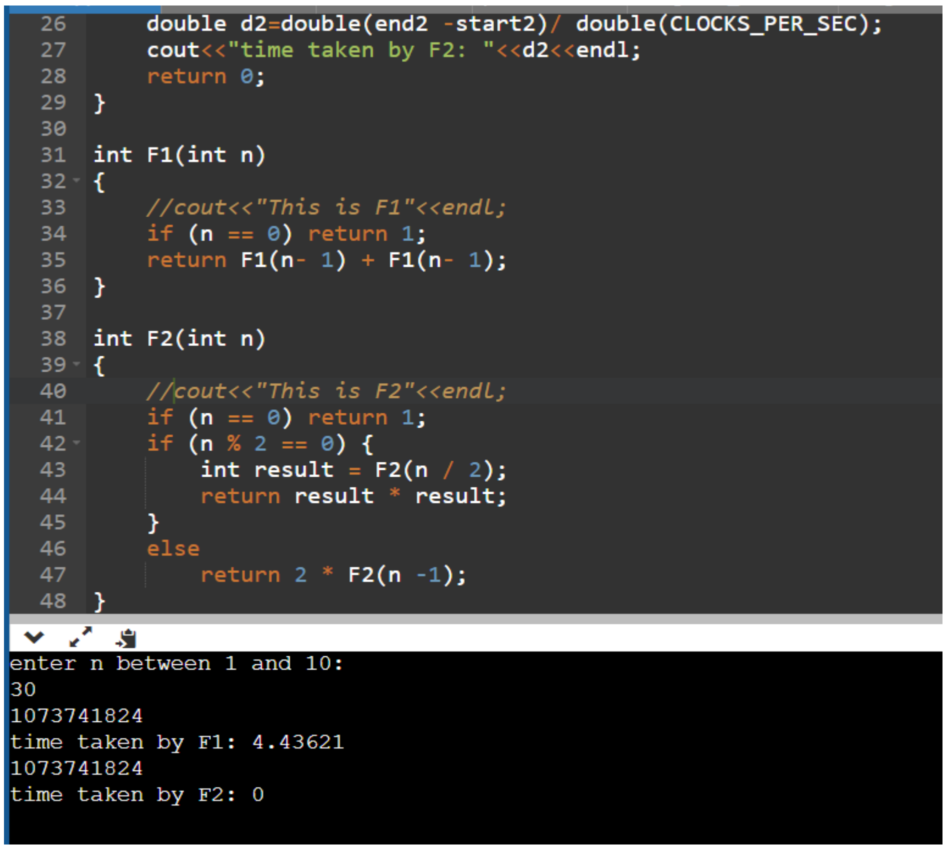
```
enter n between 1 and 10:
3
This is F1
This is F1
This is F1
This is F1
This is F1
This is F1
This is F1
This is F1
This is F1
This is F1
This is F1
This is F1
This is F1
This is F1
This is F1
This is F2
This is F2
This is F2
This is F2
8
8
```

a. **Differences**: The F1 function performs recursively until it has reached n=0 which returns the value as 1 by reducing the value of n to n-1 each time and adding the result at the end. F2 function has added a condition to reach the result which is to check if n is even or odd. If n is even then it can reduce the value by half and call the function F2 with this new value of n. Then multiplying the result with itself and when n is odd, n is reduced to n-1 and multiplied by 2 will give the same output as F1.

   Similarities: Both F1 and F2 functions evaluate the exponent of 2- (2^n) given n as input between 1 and 10. The picture above shows n=3, the exponent of 2^3 =8 as the output value of both functions stored in the variables "one" and "two" respectively. Both functions are recursive in nature. Whenever the function is called, it prints

“This is <function name>” for us to understand how many times a function is being called.

b. It is evident that F2 is faster than F1. F2 is being called a lesser number of times than F1. F2 is different from F1 in the sense of the if-else condition. F2 checks if the number is an even number or an odd number and evaluates accordingly. When n=30, the function F1 kept on running for a large amount of time for around 4.5 ms while F2 kept running for a negligible amount of time.



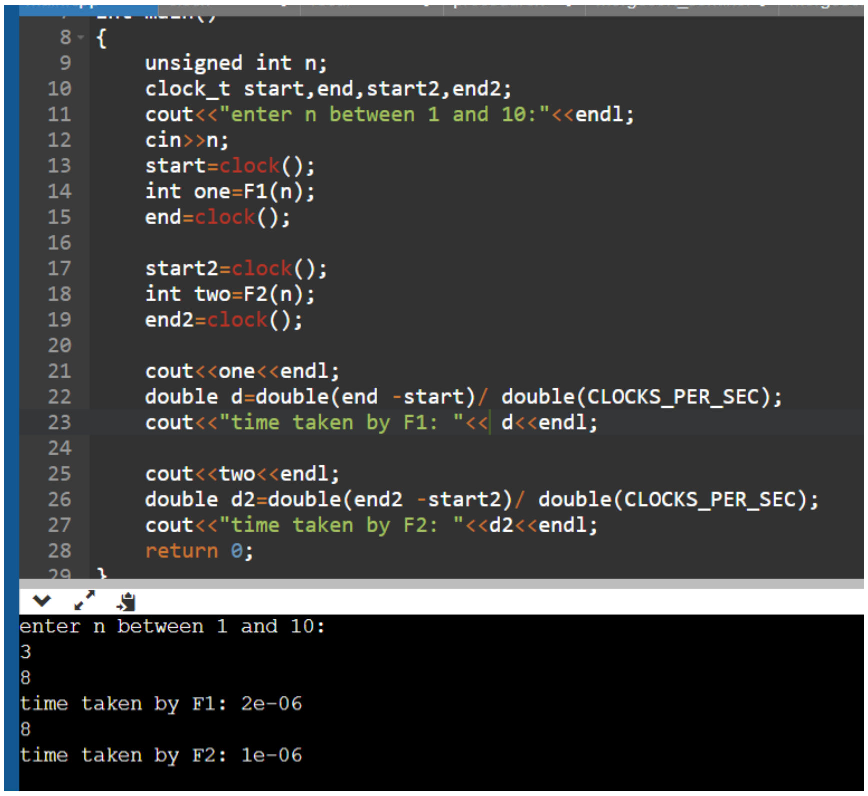
```
    double d2=double(end2 -start2)/ double(CLOCKS_PER_SEC);
    cout<<"time taken by F2: "<<d2<<endl;
    return 0;
}

int F1(int n)
{
    //cout<<"This is F1"<<endl;
    if (n == 0) return 1;
    return F1(n- 1) + F1(n- 1);
}

int F2(int n)
{
    //cout<<"This is F2"<<endl;
    if (n == 0) return 1;
    if (n % 2 == 0) {
        int result = F2(n / 2);
        return result * result;
    }
    else
        return 2 * F2(n -1);
}
```

```
enter n between 1 and 10:
30
1073741824
time taken by F1: 4.43621
1073741824
time taken by F2: 0
```

c. Running time growth: F1 takes exponent time of approximately (2^{n+1}) -1 with time complexity, $O(2^n)$ and $\Theta(2^n)$ and F2 takes logarithmic time of O(nlogn). For example when n=3; the following images show the running time growth of both the functions, therefore, F2 is faster than F1 → O(nlogn) < $O(2^n)$



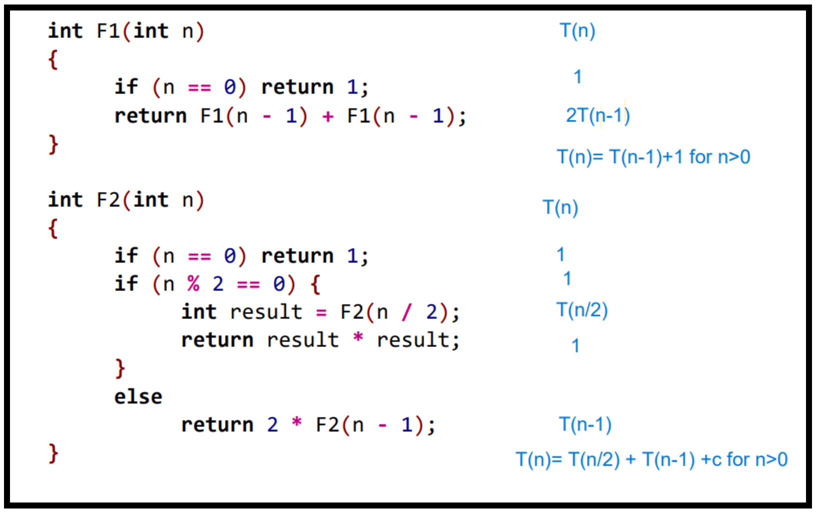
```
{
    unsigned int n;
    clock_t start,end,start2,end2;
    cout<<"enter n between 1 and 10:"<<endl;
    cin>>n;
    start=clock();
    int one=F1(n);
    end=clock();

    start2=clock();
    int two=F2(n);
    end2=clock();

    cout<<one<<endl;
    double d=double(end -start)/ double(CLOCKS_PER_SEC);
    cout<<"time taken by F1: "<< d<<endl;

    cout<<two<<endl;
    double d2=double(end2 -start2)/ double(CLOCKS_PER_SEC);
    cout<<"time taken by F2: "<<d2<<endl;
    return 0;
}
```

```
enter n between 1 and 10:
3
8
time taken by F1: 2e-06
8
time taken by F2: 1e-06
```



```
int F1(int n)                                  T(n)
{
    if (n == 0) return 1;                       1
    return F1(n - 1) + F1(n - 1);              2T(n-1)
}
                                               T(n)= T(n-1)+1 for n>0

int F2(int n)                                  T(n)
{
    if (n == 0) return 1;                      1
    if (n % 2 == 0) {                          1
        int result = F2(n / 2);                T(n/2)
        return result * result;                1
    }
    else
        return 2 * F2(n - 1);                  T(n-1)
}                                           T(n)= T(n/2) + T(n-1) +c for n>0
```

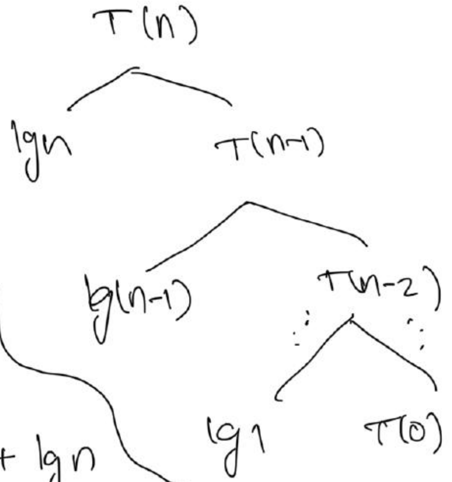
$\boxed{F2}$

$\lg n + \lg(n-1) + \dots + \lg 2 + \lg 1$

$= \lg [\, n \times n-1 \times \dots \times 2 \times 1 \,]$

$= \lg n! \rightarrow O(n \lg n)$

$$T(n) = \begin{cases} 1 & n=0 \\ T(n-1) + \lg n & n>0 \end{cases}$$



$T(n) = T(n-1) + \lg n \quad - \textcircled{1}$

$T(n) = [\, \overset{\downarrow}{T}(n-2) + \lg(n-1) \,] + \lg n$

$T(n) = T(n-2) + \lg(n-1) + \lg n \quad - \textcircled{2}$

$T(n) = [\, \overset{\downarrow}{T}(n-3) + \lg(n-2) \,] + \lg(n-1) + \lg n \quad - \textcircled{3}$

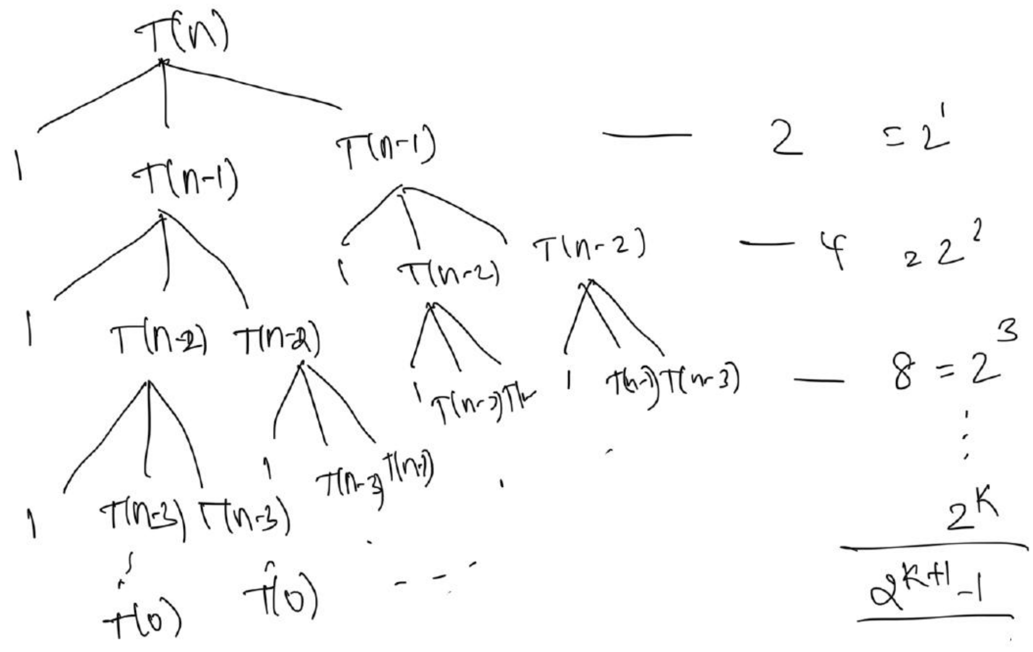
$\vdots$

$T(n) = T(n-K) + \lg 1 + \lg 2 + \dots \lg(n-1) + \lg n$

Assume $n-K=0 \ \exists\ n=k$

$T(n) = T(0) + \lg n! = 1 + \lg n! = \underline{O(n \lg n)}$

$\boxed{A}$ $T(n) = \begin{cases} 1 & n=0 \\ 2T(n-1)+1 & n>0 \end{cases}$



$= 2^{K+1} - 1 = 2^{n+1} - 1 \rightarrow \Theta(2^n)$

Recurrence Eqn-

$T(n) = 2T(n-1) + 1 \quad -\textcircled{1}$

$T(n) = 2^1 [2T(n-2) + 1] + 1$

$\vdots$

$T(n) = 2^K T(n-K) + 2^{K-1} + 2^{K-2} + \cdots + 2 + 1$

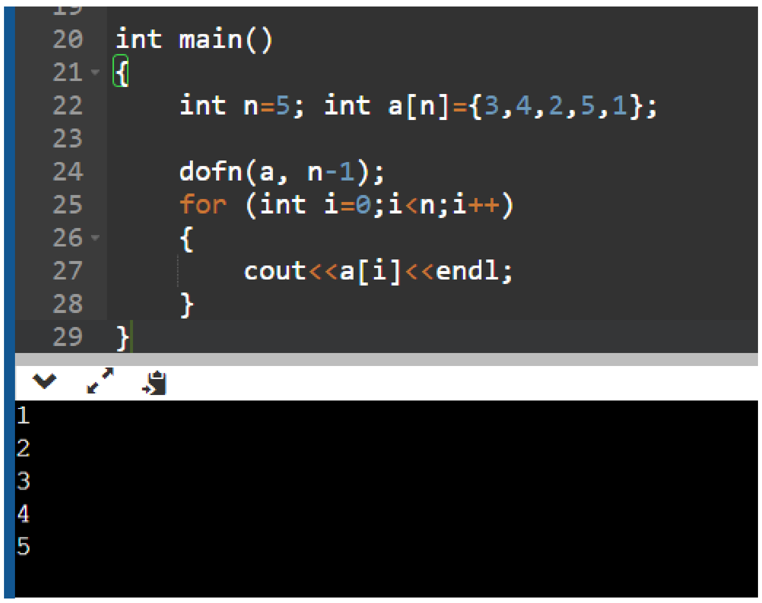
when $n-k=0 \Rightarrow n=k,\ T(n) = 2^n T(0) + 2^k - 1$

$$= 2^n + 2^n - 1$$

$$= 2^{n+1} - 1 \rightarrow \left\{ \begin{matrix} O(2^n) \\ \Theta(2^n) \end{matrix} \right\}$$

4. A. purpose of the code: to sort the array using bubble sort algorithm.

It starts with taking index 1 as i and j is the last index of the array of size n. For the first loop of i=1, j loops from n to i+1, checking all the values in the array from j to j-1, decrementing further, and if the value of a[j] is less than the value of a[j-1] then it swaps these values. If the value of a[j] is not lesser than a[j-1] then the loop continues. When j reaches i, it comes out of the loop and i is incremented. The result is given in ascending order. The if condition helps the sorting function to check the lesser values in the array from the last element to (i+1)th element.



```
int main()
{
    int n=5; int a[n]={3,4,2,5,1};

    dofn(a, n-1);
    for (int i=0;i<n;i++)
    {
        cout<<a[i]<<endl;
    }
}
```

```
1
2
3
4
5
```

B. worst-case running time formula is given in the following image- $O(n^2)$

4) Procedure X (A)

```
1.  for i = 1 to n-1
2.      for j = n downto i+1
            if A[j] < A[j-1]
                exchange A[j] with A[j-1]
```

Cost	times
c_1	n
c_2	$\sum_{j=2}^{n} t_j$
c_3	n
c_4	n
$T(n)$	$3n + \sum_{j=2}^{n} t_j$

Worst-case running time-

Total cost, $T(n) = 3n + \sum_{j=2}^{n} t_j$

(t_j) is j for worst-case scenario,

$$T(n) = 3n + \frac{n(n+1)}{2} - 1$$

$$= 3n + \frac{n^2+n}{2} + c$$

$$= \frac{6n + n^2 + n}{2} + C \quad [c \text{ is constant}]$$

$$\therefore T(n) = \frac{n^2}{2} + \frac{7n}{2} + C \quad (\text{quadratic fn of } n)$$

order of complexity $\rightarrow O(n^2)$

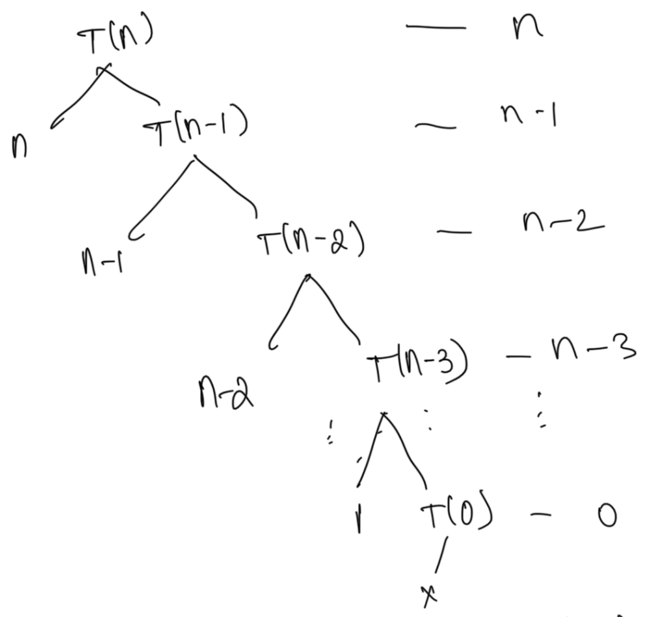
5. Recursive insertion sort algorithm:

insertion-sort(A, n)

1. If n<=1 return; //base case
2. insertion-sort(A, n-1)
3. Insert A[n] in A[n-1] // last element at it correct index in the sorted array

The recursive function of this algorithm sets the base case as n less than or equal to 1 and the recursive call to the function is made by n-1 elements

Running time growth: recurrence equation is derived in the picture below with time complexity as O(n^2)



$$\Rightarrow n + n-1 + n-2 + n-3 + \dots + 0 = \frac{n(n+1)}{2}$$

$$T(n) = \frac{n(n+1)}{2} = \Theta(n^2)$$

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + n & n > 0 \end{cases}$$

Insertion sort (A , n) — T(n)

```
1.  if n<=1 return                  — 1
2.  Insertion sort (A, n-1)         — T(n-1)
3.  while (j>0 ~~&&~~ A[j] > key)   — n
```

$\therefore T(n) = T(n-1) + n \qquad n > 0$

$T(n) = [T(n-2) + n-1] + n$

$T(n) = [T(n-3) + n-2] + n-1 + n$

$\vdots$

$T(n) = T(n-k) + (n-k + \dots + n-1+n)$

when $n = k$

$T(n) = T(0) + (1 + \dots + n)$

$T(n) = n + \frac{n(n+1)}{2}$

$T(n) = n + \frac{n^2+n}{2}$

$T(n) = O(n^2)$