1. Linear Search.

```
package linearsearch;
public class Linearsearch {
public static int linearsearch(int arr[],int key){
    int n=arr.length;
    for(int i=0;i<n;i++){
      if(arr[i]==key);
      return i;
    }
    return -1;
  }
  public static void main(String[] args) {
    int arr[]={10,20,30,40,50,60};
    int key=40;
    int result=linearsearch(arr,key);
    if(result==-1)
      System.out.println("Element Not Found");

    else
      System.out.println("Element Found at Index:"+result);
    }
}
```

Time Complexity:

Linear Search Algorithm:

- for→ 0 to n-1;
- If(index==key)
- return index;
- end loop

In this algorithm loop is running 0 to n-1.

Worst case: O(n). [Key element at last index]

Best Case: O(1). [Key element at first index]

Average Case: O(n). [Key element in any index without first and last index]

2. Binary Search

```
package binareysearch;
public class BinareySearch {
int binearySearch(int arr[],int l, int h,int key){
 while(l<=h){
      int mid=(l+h)/2;
      if(arr[mid]==key)
        return mid;
      if(arr[mid]>key)
          l=mid-1;
      else
          l=mid;
      }
    return -1;
  }
  public static void main(String[] args) {
     BinareySearch obj = new BinareySearch();
    int arr[]={10,20,30,40,50,60};
    int n=arr.length;
    int key=60;
    int result=obj.binearySearch(arr,0,n,key);
    if(result==-1)
      System.out.println("Element not Found");
```

```
        else
            System.out.println("Element Found at " + "index " + result);
    }
}
```

Time Complexity:

In this algorithm total time is divided by 2 recursively.
Lets, total time is n.

$$T(n)=\frac{n}{2}+\frac{n}{4}+\frac{n}{8}+\cdots+1$$

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

$$So, \frac{n}{2^x}=1$$

x=n log n

Time complexity of Binary search is n log n.

3. Bubble Sort

```
package bubblesort;
import java.util.Arrays;

public class Bubblesort {

public static void main(String[] args) {

    int[]numbers={11,10,21,13,45,6,7,8,15,1};
    System.out.println(Arrays.toString(numbers));
      bubblesort(numbers);
   }
  public static void bubblesort(int arr[]){
    int n=arr.length;
    for(int i=0;i<n-1;i++)
        for(int j=0;j<n-i-1;j++){
            if(arr[j]>arr[j+1]){
                int temp=arr[j];
                arr[j]=arr[j+1];
                arr[j+1]=temp;
```

```
            }
          }
        }
      }
```

Time complexity:
This algorithm has nested loop. In nested loop inner loop run for “n” time while outer loop run one time. So in best case its time complexity is O(1)=O(n).Because array is already sorted. But in worst case its time complexity is O(n^2). Because outer loop run “n” times. So it is O(n*n).

4. Insertion Sort

```
package insertationsort;

public class Insertationsort {

// A funcation to sort array using insertation sort.

    void sort(int arr[]){
      int n=arr.length;
      for(int i=1;i<n;i++){
      int key=arr[i];
      int j=i-1;
      while(j>=0 && arr[j]>key){
        arr[j+1]=arr[j];
        j=j-1;
      }
      arr[j+1]=key;
      }
    }
    static void printArray(int arr[])
    {
      int n=arr.length;
      for(int i=0;i<n;i++)
        System.out.println(arr[i]+" ");
      System.out.println();
    }
    // Driver method

    public static void main(String[] args)
    {
      int arr[]={12,11,13,5,6};
      Insertationsort obj=new Insertationsort();
```

```
        obj.sort(arr);
        printArray(arr);
    }
    }
```

Time complexity:

This algorithm has nested loop. In nested loop inner loop run for “n” time while outer loop run one time. So in best case its time complexity is O(1)=O(n).Because array is already sorted. But in worst case its time complexity is O(n^2). Because outer loop run “n” times. So it is O(n*n).

5. Fibonacci Number

```
package fabonnacci;
public class Fabonnacci {

  static int fabonacci(int n){
     int f[]=new int[n+2];
   int i;
     f[0]=0;
     f[1]=1;
     for( i=2;i<=n;i++){

        f[i]=f[i-1]+f[i-2];
     }
     return f[n];
  }
   public static void main(String[] args) {
      int n=10;
      System.out.println(fabonacci(n));
   }
}
```

Time complexity:

T(n) =T(n-1) +T(n-2)

or,T(n-1)≈ $T(n-2)$

So, T(n)=2T(n-2) +c

or,T(n)=2{2T(n-2) +c}+c

or, T(n)= 4T(n-4)+3c

or, T(n)= n^2 (n-2.K)+ c

So we can say time complexity is n^2 in worst case.

6. **Last Digit of Large Fibonacci Number**

```
package fabonnacci;

public class Fabonnacci {

static int fabonacci(int n){

    int f[]=new int[n+2];
    int i;
    f[0]=0;
    f[1]=1;

    for( i=2;i<=n;i++){
        f[i]=f[i-1]+f[i-2];

    }
    return f[n];
  }
  public static void main(String[] args) {
    int n=10;
    fabonacci(n);
    System.out.println("Last Digit of Large Fabonacci Number:"+fabonacci(n)%10);
  }

}
```

Time complexity:

T(n) =T(n-1) +T(n-2)
or,T(n-1)≈ $T(n-2)$
So, T(n)=2T(n-2) +c
or,T(n)=2{2T(n-2) +c}+c
or, T(n)= 4T(n-4)+3c
or, T(n)= n^2 (n-2.K)+ c

So we can say time complexity is n^2 in worst case.

7. **Greatest common divisor**

```
package gcd_example;
```

```
import java.util.Scanner;

public class GCD_Example {

public static void main(String[] args) {

    //Enter two integer
    Scanner in=new Scanner(System.in);
    System.out.println("Enter Value of a:");
    int a=in.nextInt();
    System.out.println("Enter Value of b:");
    int b=in.nextInt();
    System.out.println("GCD of two numbers is " + a + " and " +b +":"+findGCD(a,b));
  }
  private static int findGCD(int a,int b){
    if(b==0)
      return a;
    return findGCD(b,a%b);
  }

}
```

Time complexity:

In this code there no loop. Every line takes a constant time.

C1+ C2+C3+….+ Cn

or, C(1+2+3+…+n)

So, in best case it takes O(1) and in worst case it takes O(n) time.