1) FACTORIAL
------------

```
import java.util.*;
class test
{
public static void main(String[]args)
{
Scanner in = new Scanner(System.in);
System.out.println("Enter a number");
int n = in.nextInt();
if(n<0)
System.out.println("Number should not be negative");
int fact = fact_fun(n);
System.out.println("Factorial of "+n+" is "+fact);
}

public static int fact_fun(int num)
{
int fact = 1;
for(int i=1;i<=num;i++)
{
fact=fact*i;
}
return fact;

}
}



public static int fact_fun(int num)
{
int fact;
if(num==1 || num==0)
return 1;
else
return num*fact_fun(num-1);

}
```

2) PRIME NUUMBERS
-----------------

```
booolean isPrime=true;
for(int i=2;i<=Math.sqrt(n);i++)
{
if(n%i==0)
{
isPrime=false;
break;
}
}
```

```
if(isPrime==true)
System.out.println("Prime number");
else
System.out.println("Not a Prime number");
```

2) ARMSTRONG NUUMBERS

```
int num=100;
int m=0;
double d=0,sum=0;
m=num;
while(num>0)
{
d=num%10;
num=num/10;
sum=sum+Math.pow(d,3);
}
if(sum==m)
System.out.println("Armstrong number");
else
System.out.println("Not Armstrong number");
```

3) PALINDROME

```
String str = "Hello World";
String rev;
for(int i=str.length()-1;i>=0;i--)
{
rev=rev+str.charAt(i);
}
if(rev.equalsIgnoreCase(str))
System.out.println("Palindrome");
else
System.out.println("Not a palindrome");
```

4) COMPARE TO

```
public static void main(String args[])
   {
      String s1, s2;
      Scanner in = new Scanner(System.in);

      System.out.println("Enter the first string");
      s1 = in.nextLine();

      System.out.println("Enter the second string");
      s2 = in.nextLine();

      if ( s1.compareTo(s2) > 0 )
```

```
         System.out.println("First string is greater than second.");
      else if ( s1.compareTo(s2) < 0 )
         System.out.println("First string is smaller than second.");
      else
         System.out.println("Both strings are equal.");
   }
```

5) LINEAR SEARCH

Complexity of linear search is O(n).

```
boolean found=true;
int pos=0;
Scanner in = new Scanner(System.in);
System.out.println("Enter the size");
int n = in.nextInt();
int ar[] = new int [n];
for(int i=0;i<n;i++)
{
System.out.println("Enter a number");
ar[i] = in.nextInt();
}

System.out.println("Enter a number to search");
int num = in.nextInt();

for(int i=0;i<n;i++)
{
if(ar[i]==num)
{
found=true;
pos=i;
break;
}
}
if(found==true)
System.out.println("Number found at pos"+pos);
else
System.out.println("Number not found");
   }
```

6) BINARY SEARCH

It should be applied on sorted array. Complexity is log(n).

```
boolean found=false;
int low=0;
int high = ar.length-1;
//input n;
while(low<=high)
{
```

```
mid=(low+high)/2;
if(ar[mid]==n)
{
found=true;
pos=i;
break;
}
else if (n<ar[mid])
high=mid-1;
else if(n>ar[mid])
low=mid+1;
}
```

Recursive

```
public int bin_search(int ar[],int key,int low,int high)
{
if(low<high)
{
int mid=(low+high)/2;
if(key<ar[mid])
return bin_search(ar,key,low,mid-1);
else if(key>ar[mid])
return bin_search(ar,key,mid+1,high);
else if(key==ar[mid])
return mid;
}
else
return -1;
}
```

DATETIME

```
 day = date.get(Calendar.DAY_OF_MONTH);
      month = date.get(Calendar.MONTH);
      year = date.get(Calendar.YEAR);
```

GARBAGE COLLECTION

```
import java.util.*;

class GarbageCollection
{
   public static void main(String s[]) throws Exception
   {
      Runtime rs =  Runtime.getRuntime();
      System.out.println("Free memory in JVM before Garbage Collection = "+rs.freeMemory());
      rs.gc();
      System.out.println("Free memory in JVM after Garbage Collection = "+rs.freeMemory());
   }
}
```

RANDOM NUMBERS

```
import java.util.*;

class RandomNumbers {
  public static void main(String[] args) {
    int c;
    Random t = new Random();

    // random integers in [0, 100]

    for (c = 1; c <= 10; c++) {
      System.out.println(t.nextInt(100));
    }
  }
}
```

REVERSE

```
while( n != 0 )
      {
          reverse = reverse * 10;
          reverse = reverse + n%10;
          n = n/10;
      }
```

FIBONNACI SERIES

```
int f1=0,f2=0,f3=1;
for(int i=1;i<=num;i++)
{
System.out.print(f3);
f1=f2;
f2=f3;
f3=f1+f2;
}
```

BUBBLE SORT

The heaviest bubble settles at the bottom. Bubble sort has worst-case and average complexity both (n2), where n is the number of items being sorted.

```
 for (c = 0; c < ( n - 1 ); c++) {
      for (d = 0; d < n - c - 1; d++) {
        if (array[d] > array[d+1]) /* For descending order use < */
        {
          swap       = array[d];
          array[d]   = array[d+1];
          array[d+1] = swap;
        }
      }
    }
```

SELECTION SORT

It searches the min element in the array and swaps it. Complexity is O(n2).

```
for(int i=0;i<n;i++)
{
for(int j=i+1;j<n;j++)
{
pos=i;
min=ar[i];
if(ar[j]<min)
{
min=ar[j];
pos=j;
}
}
ar[pos]=ar[i];
ar[i]=min;
}
```

INSERTION SORT

Complexity is O(n2). It is faster than bubble sort.

```
int pos,temp;
for(int i=1;i<n;i++)
{
pos=i;
temp=ar[i];
while(pos>0 && ar[pos-1]>temp)
{
ar[pos]=ar[pos-1]
pos--;
}
ar[pos]=temp;
}
```

STACK

A Stack is an abstract data type or collection where in Push,the addition of data elements to the collection, and Pop, the removal of data elements from the collection, are the major operations performed on the collection. The Push and Pop operations are performed only at one end of the Stack which is referred to as the 'top of the stack'. In other words,a Stack can be simply defined as Last In First Out (LIFO) data structure,i.e.,the last element added at the top of the stack(In) should be the first element to be removed(Out) from the stack.

```
class stack
{
int top;
int ar[];

stack()
{
top=-1;
ar=new int [10];
}

void pushItem(int num)
{
if(top==ar.length-1)
SOPln("Stack is full");
else
{
ar[++top]=num;
SOPln("Pushed Item : +num");
}
}

void PopItem()
{
if(top==-1)
SOPln("Stack is empty");
else
SOPln("Popped Item : +ar[top--]");
}

void top()
{
if(top==-1)
SOPln("Stack is empty");
else
SOPln("Top of stack is "+top+" and element is :"+ar[top]);
}
}
```

QUEUE

A queue is a kind of abstract data type or collection in which the entities in the collection are kept in order and the only operations on the collection are the addition of entities to the rear terminal position, called as enqueue, and removal of entities from the front terminal position, called as dequeue. The queue is called as First-In-First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be

the first one to be removed. This is equivalent to the requirement that once a new element is added, all elements that were added before have to be removed before the new element can be removed. Often a peek or front operation is also entered, returning the value of the front element without dequeuing it. A queue is an example of a linear data structure, or more abstractly a sequential collection.

```
class queue
{
int ar[];
int front,rear;

queue(int size)
{
ar=new int [size];
front=-1;
rear=-1;
}

boolean isEmpty()
{
return front==-1 && rear==-1;
}

boolean isFull()
{
return front==0&&rear=ar.length-1;
}

void push(int ele)
{
if (isEmpty())
{
front=0;
rear=0;
ar[front]=ele;
}
else if (isFull())
SoPln("Queue is full");
else
ar[++rear]=ele;
}

int Pop()
{
if(isEmpty())
SOPln("Queue is empty");
return -1;
else if (front==0 && rear==0)
{
int ele = ar[front];
front=-1;
rear=-1;
return ele;
}
else
int ele = ar[front];
for(int i=0;i<rear-1;i++)
{
ar[i]=ar[i+1];
}
rear--;
return ele;
}
}
}
```

1.Requirement gathering and analysis
2.Design
3.Implementation or coding
4.Testing
5.Deployment
6.Maintenance

unit testing, integration testing, system testing, acceptance testing