Chapter: Recursion

The topics

1. Recursion

1.1. What is Recursion?

Recursion is a technique that allows us to break down a problem into one or more sub-problems that are similar in form to the original problem.
Recursion in breaking down of an algorithm in multiple modules of similar type and when any function uses this technique within its body, the function is called recursive function.

1.2. Example: The Factorial of a number

Recall that factorial, which is written n!, has the following definition:
n! = 1 * 2 * 3 * .... * (n-2) * (n-1) * n
We can use this definition to write:

```
int fact(int n) {
        int i;
        int result;

        result = 1;
        for (i = 1; i <= n; i++) {
           result = result * i;
        }
        return result;
    }
```

We can write a function that uses recursion as follows:

```
int fact(int n) {
        if (n = = 0)
            return 1;
        return n * fact(n - 1);
    }
```

Comparing the two versions:

- The iterative version has two local variables; the recursive version has none.
- The iterative version has three statements; the recursive version has one.
- The iterative version must save the solution in an intermediate variable before it can be returned; the recursive version calculates and returns its result as a single expression.

Recursion simplifies the fact function! It does so by making the computer do more work, so that you can do less work.

1.3. Definition of recursion

Recursion is a powerful technique of writing a complicated algorithm in an easy way. According to this technique, a problem is defined in terms of itself. To implement recursion technique in programming, a function should be capable of calling itself. The function that calls itself again and again from inside the function body is called recursive function. In recursive function, the calling function and the called function both are same.

2. Recursive Function

2.1 The recursive functions can be characterized into the following categories

1. Depending upon whether the function calls itself or not, the recursion will be a **direct or indirect recursion**.
2. Depending upon whether there are pending operations at each recursive call, the recursion will be a **tail-recursion or non-tail recursion**.
3. Depending upon the shape of the calling pattern, the recursion will be **linear or tree-recursion**.

3. Working Principle Of Recursion

3.1 Every recursive process consists of two parts

- A smallest, **base case** that is processed without recursion; and
- A general method, a **recursive case** that reduces a particular case to one or more of the smaller cases, thereby making progress toward eventually reducing the problem all the way to the base case. (Here in recursive case, updation of the variable i.e parameter takes place)

3.2 Calculating Factorials using recursion

```
int factorial(int n)
{
        if (n = = 0 || n==1) // base case
                return 1;
        else
                return n * factorial(n - 1); //recursive case
}
```

Working :

```
factorial(5)
= 5 * factorial(4)
= 5 * (4 * factorial(3))
= 5 * (4 * (3 * factorial(2)))
= 5 * (4 * (3 * (2 * factorial(1))))
```

Recursive call to next function

```
= 5 * (4 * (3 * (2 * (1))))
= 5 * (4 * (3 * (2 * 1)))
= 5 * (4 * (3 * 2))
= 5 * (4 * 6)
= 5 * 24
= 120 //final output
```

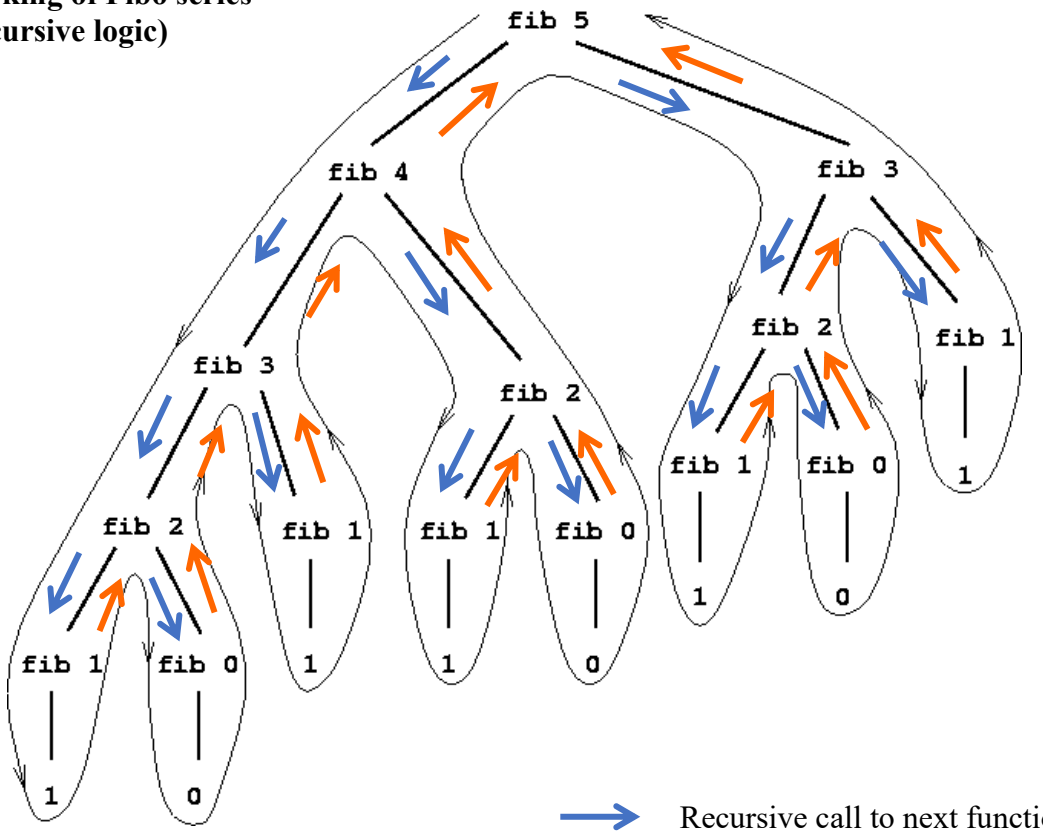
Backtracking to previous function

3.3 Calculating the Fibonacci series using recursion

```
int fib(int n) // generating fibo. no of term-n
{
        if (n == 0)
            return 0;
        else if (n == 1)
            return 1;
        else
            return  fib(n - 1) + fib(n - 2);
}

void display( )
{
        for( int i=0;i<=5;i++)
        {
            System.out.print(fib(i)+ " ");
        }
}
```

Working of Fibo series (Recursive logic)



3.4 Designing Recursive Algorithms

- Find the key step. Begin by asking yourself, “How can this problem be divided into parts?” or “How will the key step in the middle be done?”
- Find a stopping rule. This stopping rule is usually the small, special case that is trivial or easy to handle without recursion.
- Outline your algorithm. Combine the stopping rule and the key step, using an **if** statement to select between them.
- Check termination. Verify that the recursion will always terminate. Be sure that your algorithm correctly handles extreme cases.
- Draw a recursion tree. The height of the tree is closely related to the amount of memory that the program will require, and the total size of the tree reflects the number of times the key step will be done.

4. Recursion Vs Iteration

4.1 Advantages of recursion

- The use of recursion makes the code more compact and elegant.
- It simplifies the logic and hence makes the program easier to understand.
- It works well for computations, which are larger in depth.

4.2 Disadvantages of recursion

1. Recursion makes a program slow because of many function calls involved in it.
2. It involved much more memory consumption in compare to iteration.
3. If the stopping rule is not correctly implemented, then the function may enter into an infinite loop.

4.3 Difference between recursion and iteration

1. Recursive function – is a function that is partially defined by itself, whereas Iterative functions are loop-based imperative repetitions of a process
2. Recursion uses a selection structure, whereas Iteration uses a repetition structure
3. An infinite loop occurs with iteration if the loop-continuation test never becomes false, whereas infinite recursion occurs if the recursion step does not reduce the problem in a manner that converges on the base case.
4. Iteration terminates when the loop-continuation condition fails, whereas recursion terminates when a base case is recognised
5. When using recursion, multiple activation records are created on the stack for each call. When using iteration, everything is done in one activation record.
6. Recursion is usually slower than iteration due to the overhead of maintaining the stack, whereas iteration does not use the stack, so it's faster than recursion
7. Recursion uses more memory than iteration
8. Infinite recursion can crash the system, whereas infinite looping uses CPU cycles repeatedly
9. Recursion makes code smaller, and iteration makes code longer

5. Types of Recursive Function

5.1 The recursive functions can be characterised into the following categories

4. Depending upon whether the function calls itself or not, the recursion will be a direct or indirect recursion.
5. Depending on whether there are pending operations at each recursive call, the recursion will be a tail-recursion or non-tail recursion.
6. Depending upon the shape of the calling pattern, the recursion will be linear or tree recursion.

5.2. Direct Recursion

A function is directly recursive if it contains an explicit call to itself. For example, the function

```
int F1(int x)
{
        if (x <= 0)
                return x;
        else
                return F1(x - 1);
}
```

includes a call to itself, so it's directly recursive. The recursive call will occur for positive values of x.

5.3. Indirect Recursion

A function F1 is indirectly recursive if it contains a call to another function F2 which ultimately calls F1. The following pair of functions is indirectly recursive. Since they call each other, they are also known as mutually recursive functions.

```
int F1(int x)
{
        if (x <= 0)
                return x;
        return F2(x);
}

int F2(int y)
{
        return F1(y - 1);
}
```

5.4. Tail Recursion

A recursive function is said to be **tail-recursive if there are no pending operations** to be performed on return from a recursive call.
Tail-recursive functions are often said to "return the value of the last recursive call as the value of the function." Tail recursion is very desirable because the amount of information which must be stored during the computation is independent of the number of recursive calls. Some modern computing systems will actually compute tail-recursive functions using an iterative process.

The factorial function fact is usually written in **a non-tail-recursive** manner:

```
int fact (int n)
{
        if (n = = 0|| n = = 1)
                return 1;
        return n * fact(n - 1);
}
```

```
Fact(5) = 5*Fact(4)
Fact(4) = 4*Fact(3)
Fact(3) = 3*Fact(2)
Fact(2) = 2*Fact(1)
Fact(1) = 1
```

```
1
1*2=2
1*2*3=6
1*2*3*4=24
1*2*3*4*5=120
```

Notice that there is a "pending operation," namely multiplication, to be performed on return from each recursive call. Whenever there is a pending operation, the function is non-tail-recursive. Information about each pending operation must be stored, so the amount of information is not independent of the number of calls.

The factorial function can be written in a tail-recursive way:

```
int fact_aux(int n, int result)
{
        if (n = = 1)
                return result;
        return fact_aux(n - 1, n * result);
}
int fact(n)
{
         return fact_aux(n, 1);
}
```

Fact(4,1) = Fact(3,4)
Fact(3,4) = Fact(2,12)
Fact(2,12) = Fact(1,24)
Fact(1,24) = 24

The "auxiliary" function fact_aux is used to keep the syntax of fact(n) the same as before. The recursive function is really fact_aux, not fact. Note that fact_aux has no pending operations on return from recursive calls. The value computed by the recursive call is simply returned with no modification. The amount of information which must be stored is constant (the value of n and the value of result), independent of the number of recursive calls.

5.5. Linear and Tree Recursion

Another way to characterise recursive functions is by the way in which the recursion grows. The two basic ways are "linear" and "tree."

A recursive function is said to be linearly recursive when no pending operation involves another recursive call to the function.
For example, the "infamous" fact function is linearly recursive. The pending operation is simply multiplication by a scalar; it does not involve another call to fact.

A recursive function is said to be tree recursive (or non-linearly recursive) when the pending operation involves more than one recursive call to the function simultaneously.
The Fibonacci function **fib()** provides a classic example of tree recursion. The Fibonacci numbers can be defined by the rule:

fib(n) = 0 if n is 0,
= 1 if n is 1,
= fib(n-1) + fib(n-2) otherwise

For example, the first six Fibonacci numbers are :

Fib(0) = 0
Fib(1) = 1
Fib(2) = Fib(1) + Fib(0) = 1+0 = 1
Fib(3) = Fib(2) + Fib(1) = 1+1 = 2
Fib(4) = Fib(3) + Fib(2) = 2+1 = 3
Fib(5) = Fib(4) + Fib(3) = 3+2 = 5

This leads to the following implementation:

```
int fib(int n)
{
        if (n == 0)
                return 0;
        else if (n == 1)
                return 1;
        else
                return  fib(n - 1) + fib(n - 2);
}
```

Notice that the pending operation for the recursive call is another call to fib. Therefore, fib is tree-recursive.

6. Examples to show the working principle of Recursion

Example 1

Let us observe the following code segment and find the output with a proper dry run/working for the value of n=6.

```
int find( int n )
{
    If (n == 0)
            return 0;
    else if (n == 1)
            return 1;
    else
            return (n + find(n-1));
}
```

Solution:

find(6) --- 6 + find(5)
find(5) --- 5 + find(4)
find(4) --- 4 + find(3)
find(3) --- 3 + find(2)
find(2) --- 2 + find(1)
find(1) --- return 1 (Base case matched)

1+2+3+4+5+6 = 21

Example 2

Let us find the output of the following code segment with proper dry run/working for the value of n=31 and f=2. Try out n=13 and f=2; n=10 and f=2

```
int find( int n, int f)
{
    if (n == f)
            return 1;
    else if (n == 1 || n%f == 0)
            return 0;
    else
            return (find(n, f+1));
}
```

Solution:

find(31,2) → find(31,3) → find(31,4) → find(31,5) → find(31,6) → ……
…….. → find(31,30) → find(31,31) → 1

The above code segment is returning 1 for prime number else returning 0.

7. Some example programs on recursive functions

Question 1

An integer number is said to be special number, if the sum of factorials of all the digits is equal to the number itself. E.g. 145=1!+4!+5!
Create a class **Special** with following class description:

Class name : Special

Data members/instance variables

int n	:	an integer variable to hold the number

Member functions:-

Special(int)	:	parameterized constructor to initialize n
int fact(int)	:	return the factorial of a number using recursive technique only
int extract(void)	:	extract each digit from the number n and return the factorial of that digit
void display(void)	:	check and print whether n is a special number or not.

Solution :

```
class Special
{
   int n;

   public Special(int i)
   {
      n=Math.abs(i);
   }

   public int fact(int m) //used recursion
   {
      if(m==0||m==1)
      return 1;
      else
      return m*fact(m-1);
   }

   public int extract()
   {
      if(p==0)
      return 0;
      else
      {  int x=fact(p%10);
         p=p/10;
         return x+extract();
      }
   }

   void display()
   {  int m=n;
      int sum=extract(m);
      System.out.println(sum);
      if(sum==n)
      System.out.println("Special number");
      else
      System.out.println("Not a Special number");
      p=n;
      int sum2=extract();
      System.out.println(sum2);
   }
}
```

Alternate logic [Using recursive technique]

```
public int extract(int m) //extraction of digits using recursion
   {
      if(m==0)//base case
      return 0;
      else
      {  int x = fact(m%10); //each digit extracted passed to factorial()
         return x + extract(m/10);//recursive case
      }
   }
```

Question 2

Declare a **class Recur** with following class description:

Data members :-

int n	to store n^{th} term
int x	to store the value of x
int sum	to store the sum of the series : $1 + x^2 + x^3 + x^4 + \ldots$ up to n^{th} term.

Member functions :-

Recur(int, int)	constructor to initialize n and x
int sum(int x, int n)	it will add each term with sum using recursion technique and return the value.
int power(int x, int n)	it will calculate the value of x^n using recursion technique and return the value.
void result( )	it will display the result.

Solution :

```
class Recur
{
        //Data members
        int n;          // to store nth term
        int x;          // to store the value of x
        int sum;        // to store the sum of the series up to n term.

        //Member functions
        public Recur( int i, int j)          // constructor to initialize n and x
        {
                n=i;
                x=j;
        }

        public int sum(int x, int n)         // it will add each term with sum using recursion
        {                                    // technique and return the value.
                if(n==0)
                        return 1;
                else
                        return power(x, n) + sum(x,n-1);
        }

        public int power(int x, int n)       // it will calculate the value of xn using recursion
        {                                    // technique and return the value.
                if(n==0)
                        return 1;
                else
                        return x*power(x, n-1);
        }

        public void result( )       // it will display the result.
        {
                sum=1;
                sum=sum(x, n);
                System.out.println(sum);
        }
}
```

Question 3.

Class **Convert** has been defined to express digits as an integer in words. The details of the class are given below:

Class name Convert

Data members/Instance variables:-

int n	-	Integer whose digits are to be expressed in words
String s	-	Stores the word format of the number.

Member functions:-

Convert( )	default constructor
void readNum( )	to read the number from the user and store it in n.
void extDigit(int)	to extract the digits of n using a recursive technique.
void numToWord(int)	to convert the digit to word and store it in s.
void display( )	to display the number in figure and words.

Solution:

```
import java.util.*;
class Convert
{
   int n, m;
   String s;
   public Convert()
   {
      n=0;
      s="";
   }
   void readNum()throws IOException
   {
      Scanner sc=new Scanner(System.in);
      System.out.println("Enter a number");
      n=sc.nextInt();
      extDigit(n);
   }
   public void extDigit(int t)     // with recursive technique
   {
      int r;
      if(t==0)
        return;
      else
      {
         r=t%10;
         extDigit(t/10);
         numToWord(r);
      }
   }
   public void extDigit1(int t) // without recursive technique
   {
      m=n;
      int r=0;
      while(m>0)
      {
         r=r*10+m%10;
         m=m/10;
      }
      m=r;
      r=0;
      while(m>0)
      {
         r=m%10;
         m=m/10;
         numToWord(r);
      }
```

```
    }
public void numToWord(int t)
    {
        String d="";
        switch(t)
        {
            case 0: d="Zero";       break;
            case 1: d="One";        break;
            case 2: d="Two";        break;
            case 3: d="Three";      break;
            case 4: d="Four";       break;
            case 5: d="Five";       break;
            case 6: d="Six";        break;
            case 7: d="Seven";      break;
            case 8: d="Eight";      break;
            case 9: d="Nine";       break;
        }
        s=s+" "+d;
    }
    public void display()
    {
        System.out.println("No. in figure: "+n);
        System.out.println("No. in words: "+s);
    }
    public static void main()
    {
        Convert t1=new Convert();
        t1.readNum();
        t1.display();
    }
}
```

8. Find the output of the given code snippets:

a.
```
int dispF(int x, int y) // x=14, y=10
    {
        if( x>=y)
        {
            x = x – y;
            return dispF(x, y);
        }
        else
            return x;
    }
```

b.
```
void example(int n) // n=11
    {
        if( n% 2 == 0)
            return 1;
        else
            return function( n/2 )* 10 + n%2;
    }
```

c.
```
void series(int n) // if n=5
    {
        if(n<=0)
            return 1;
        else
        {
            sum+=(n+1)*(n+2);
            series(n-2);
        }
```

```
}
```

9. ***Problems on Recursion***

Question 1. Declare a **class Recur** with following class description:

Data members :-

int n	to store n^{th} term
int x	to store the value of x
int sum	to store the sum of the series : $1 - x^2 + x^3 - x^4 + \ldots$ up to n^{th} term.

Member functions :-

Recur(int, int)	constructor to initialize n and x
int sum(int x, int n, int k)	it will add each term with **sum using recursion technique** and return the value.
int power(int x, int n)	it will calculate the value of $\mathbf{x^n}$ **using recursion technique** and return the value.
void result( )	it will display the result.

Question 2. Declare a class **Convert** to express **digits of an integer in words**. The details of the class are given below:

Class name	**:**	**Convert**
Data members :-		
int N	-	to store the number N
Member functions :-		
Convert( )		default constructor
void acceptNo( )		to accept the number from the user
int extractDigit(int)		it will extract the digits of N from front end using recursive technique.
void numToWord( )		it will call extractDigit( ) and changed( ) both.
void changed(int)		it will display the digit in word

Question 3. Any number can be reduced to one digit number by using the following rule: input a number and then multiply each digit of that number. This process is to be repeated until the product is reduced to one digit. The number of times digits need to be multiplied to reach one digit is called the persistence of the number.

For example $86 \rightarrow 48 \rightarrow 32 \rightarrow 6$ (persistence 3)

$341 \rightarrow 12 \rightarrow 2$ (persistence 2)

Declare a class **Persist** with following class description:

Class name	**:**	**Persist**
Data members :-		
int N	-	to store the number N
int count	-	to store the persistence value
Member functions :-		
Persist( )	-	default constructor
void acceptNo( )	-	to accept the number from the user
int product(int)	-	it will **product the digits of a number** until the product is reduced to an one digit number and returns the product value **using recursion technique**.
void pers(int)	-	it will display the persistence value of that number.

Question 4. A prime number is a number that is divisible by 1 and that number. Twin prime numbers are the pair of 2 prime numbers whose difference is 2, e.g (3,5), (5,7), (7,9) etc. Declare a class named TwinPrime with two member functions **int primeCheck(int)** and void **TwinPrimeDisp(int,int)**. Using above class WAP to print all the twin prime numbers between a range given by the user.

Apply recursive technique in primeCheck( ) function.

Question 5. $nCr = \dfrac{n!}{(n-r)! \times r!}$, where n and r are unknown numbers entered by the user.

Declare a class **Recur** with following class description:

Data members :-	int n, r
	float c
Member functions :-	
Recur(int, int)	constructor to initialize n and r
int factorial(int)	it will calculate the **factorial of a number using recursion technique**.
void result()	it will compute the value of c using the above formula and display the result.