Types of Recursive Methods

- Types of Recursive Methods
- Direct and Indirect Recursive Methods
- Nested and Non-Nested Recursive Methods
- Tail and Non-Tail Recursive Methods
- Linear and Tree Recursive Methods
- Excessive Recursion

Types of Recursive Methods

- **A recursive method is characterized based on:**
  - **Whether the method calls itself or not (direct or indirect recursion).**
  - **Whether the recursion is nested or not.**
  - **Whether there are pending operations at each recursive call (tail-recursive or not).**
  - **The shape of the calling pattern -- whether pending operations are also recursive (linear or tree-recursive).**
  - **Whether the method is excessively recursive or not.**

Direct and Indirect Recursive Methods

- A method is *directly* recursive if it contains an explicit call to itself.

```
long factorial (int x) {
   if (x == 0)
      return 1;
   else
      return x * factorial (x – 1);
}
```

- A method x is *indirectly* recursive if it contains a call to another method which in turn calls x. They are also known as *mutually recursive* methods:

```
public static boolean isEven(int n) {
   if (n==0)
      return true;
   else
      return(isOdd(n-1));
}

public static boolean isOdd(int n) {
  return (! isEven(n));
}
```

Direct and Indirect Recursive Methods

- **Another example of mutually recursive methods:**

$$\sin(x) = \sin\left(\frac{x}{3}\right)\frac{\left(3 - \tan^2\left(\frac{x}{3}\right)\right)}{\left(1 + \tan^2\left(\frac{x}{3}\right)\right)}$$

$$\sin(x) \approx x - \frac{x^3}{x} \qquad \text{for small values of x}$$

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

$$\cos(x) = \sqrt{1 - \sin^2 x}$$

Direct and Indirect Recursive Methods

```
public static double sin(double x){
   if(x < 0.0000001)
      return x - (x*x*x)/6;
   else{
      double y = tan(x/3);
      return sin(x/3)*((3 - y*y)/(1 + y*y));
   }
}

public static double tan(double x){
   return sin(x)/cos(x);
}

public static double cos(double x){
   double y = sin(x);
   return Math.sqrt(1 - y*y);
}
```

Nested and Non-Nested Recursive Methods

- Nested recursion occurs when a method is not only defined in terms of itself; but it is also used as one of the parameters:
- Example: The Ackerman function

$$A(n, m) = \begin{cases} m + 1 & \text{if } n = 0 \\ A(n - 1, 1) & \text{if } n > 0, m = 0 \\ A(n - 1, A(n, m - 1)) & \text{otherwise} \end{cases}$$

```
public static long Ackmn(long n, long m){
    if (n == 0)
          return m + 1;
    else if (n > 0 && m == 0)
           return Ackmn(n – 1, 1);
     else
          return Ackmn(n – 1, Ackmn(n, m – 1));
}
```

- The Ackermann function grows faster than a multiple exponential function.

Tail and Non-Tail Recursive Methods

- A method is tail recursive if in each of its recursive cases it executes one recursive call and if there are no pending operations after that call.
- Example 1:

```
public static void f1(int n){
   System.out.print(n + "  ");
   if(n > 0)
      f1(n - 1);
}
```

- Example 2:

```
public static void f3(int n){
   if(n > 6){
      System.out.print(2*n + "  ");
      f3(n – 2);
   } else if(n > 0){
      System.out.print(n + "  ");
      f3(n – 1);
   }
}
```

Tail and Non-Tail Recursive Methods

- Example of non-tail recursive methods:
- Example 1:

```
public static void f4(int n){
   if (n > 0)
      f4(n - 1);
   System.out.print(n + "  ");
}
```

  – After each recursive call there is a pending System.out.print(n + " ") operation.

- Example 2:

```
long factorial(int x) {
    if (x == 0)
      return 1;
    else
      return x * factorial(x – 1);
}
```

  – After each recursive call there is a pending * operation.

Converting tail-recursive method to iterative

- **It is easy to convert a tail recursive method into an iterative one:**

Tail recursive method

```
public static void f1(int n) {
   System.out.print(n + "  ");
   if (n > 0)
      f1(n - 1);
}
```

Corresponding iterative method

```
public static void f1(int n) {
   for( int k = n; k >= 0; k--)
      System.out.print(k + "  ");
}
```

```
public static void f3 (int n) {
  if (n > 6) {
    System.out.print(2*n + "  ");
    f3(n – 2);
  } else if (n > 0) {
    System.out.print(n + "  ");
    f3 (n – 1);
  }
}
```

```
public static void f3 (int n) {
  while (n > 0) {
    if (n > 6) {
      System.out.print(2*n + "  ");
      n = n – 2;
    } else if (n > 0) {
      System.out.print(n + "  ");
      n = n – 1;
    }
  }
}
```

Why tail recursion?

- It is desirable to have tail-recursive methods, because:

  a. The amount of information that gets stored during computation is independent of the number of recursive calls.

  b. Some compilers can produce optimized code that replaces tail recursion by iteration (saving the overhead of the recursive calls).

  c. Tail recursion is important in languages like Prolog and Functional languages like Clean, Haskell, Miranda, and SML that do not have explicit loop constructs (loops are simulated by recursion).

Converting non-tail to tail recursive method

- A non-tail recursive method can often be converted to a tail-recursive method by means of an "auxiliary" parameter. This parameter is used to form the result.

- The idea is to attempt to incorporate the pending operation into the auxiliary parameter in such a way that the recursive call no longer has a pending operation.

- The technique is usually used in conjunction with an "auxiliary" method. This is simply to keep the syntax clean and to hide the fact that auxiliary parameters are needed.

Converting non-tail to tail recursive method

- Example 1: Converting non-tail recursive factorial to tail-recursive factorial

```
long factorial (int n) {
   if (n == 0)
      return 1;
   else
      return n * factorial (n – 1);
}
```

- We introduce an auxiliary parameter *result* and initialize it to 1. The parameter *result* keeps track of the partial computation of n! :

```
public long tailRecursiveFact (int n) {
     return factAux(n, 1);
}
private long factAux (int n, int result) {
   if (n == 0)
       return result;
   else
       return factAux(n-1, n * result);
}
```

Converting non-tail to tail recursive method

- Example 2: Converting non-tail recursive fib to tail-recursive fib
- The fibonacci sequence is:

  0 1 1 2 3 5 8 13 21 . . .

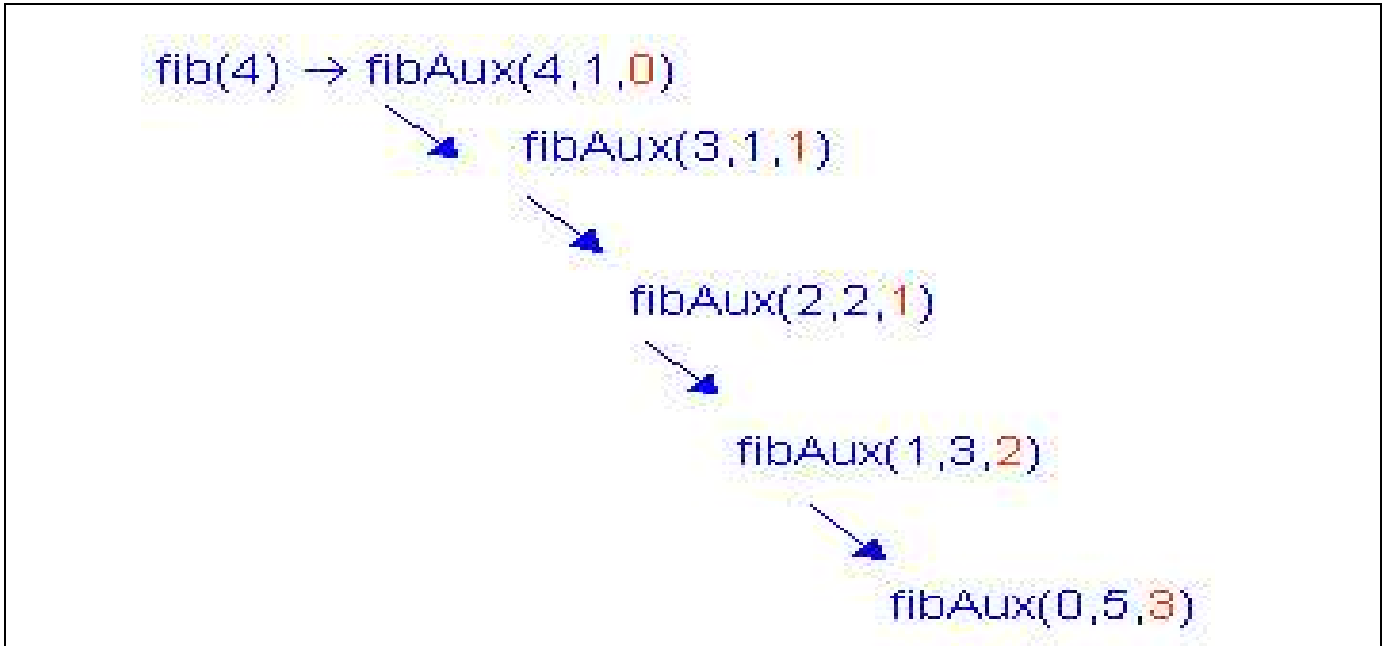
- Each term except the first two is a sum of the previous two terms.

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

- Because there are two recursive calls, a tail-recursive fibonacci method can be implemented by using two auxiliary parameters for accumulating results:

Converting non-tail to tail recursive method

```
int fib (int n) {
   return fibAux(n, 1, 0);
}
int fibAux (int n, int next, int result) {
  if (n == 0)
     return result;
  else
     return fibAux(n – 1, next + result, next);
}
```



Linear and Tree Recursive Methods

- Another way to characterize recursive methods is by the way in which the recursion grows. The two basic ways are "linear" and "tree."
- A recursive method is said to be *linearly* recursive when no pending operation involves another recursive call to the method.
- For example, the factorial method is linearly recursive. The pending operation is simply multiplication by a variable, it does not involve another call to factorial.

```
long factorial (int n) {
   if (n == 0)
      return 1;
   else
      return n * factorial (n – 1);
}
```

Linear and Tree Recursive Methods

- A recursive method is said to be *tree* recursive when the pending operation involves another recursive call.

- The Fibonacci method fib provides a classic example of tree recursion.

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

Excessive Recursion

- A recursive method is excessively recursive if it repeats computations for some parameter values.
- Example: The call fib(6) results in two repetitions of f(4). This in turn results in repetitions of fib(3), fib(2), fib(1) and fib(0):