Recursion

Recursive function

- A recursive function is a function that calls itself, directly or indirectly.
- A recursive function consists of two parts:

  Termination Condition and Body (which include recursive expansion).

  1. Termination Condition: A recursive function always contains one or more terminating condition. A condition in which recursive function is processing as a simple case and do not call itself.
  2. Body (including recursive expansion): The main logic of the recursive function contained in the body of the function. It also contains the recursion expansion statement that in turn calls the function itself.

Recursive function

All recursive functions works in 2 phases –

Winding phase: It begins when the recursive function is called for the first time, and each recursive call continues the winding phase.

In this phase function keeps on calling itself and no return statements are executed.

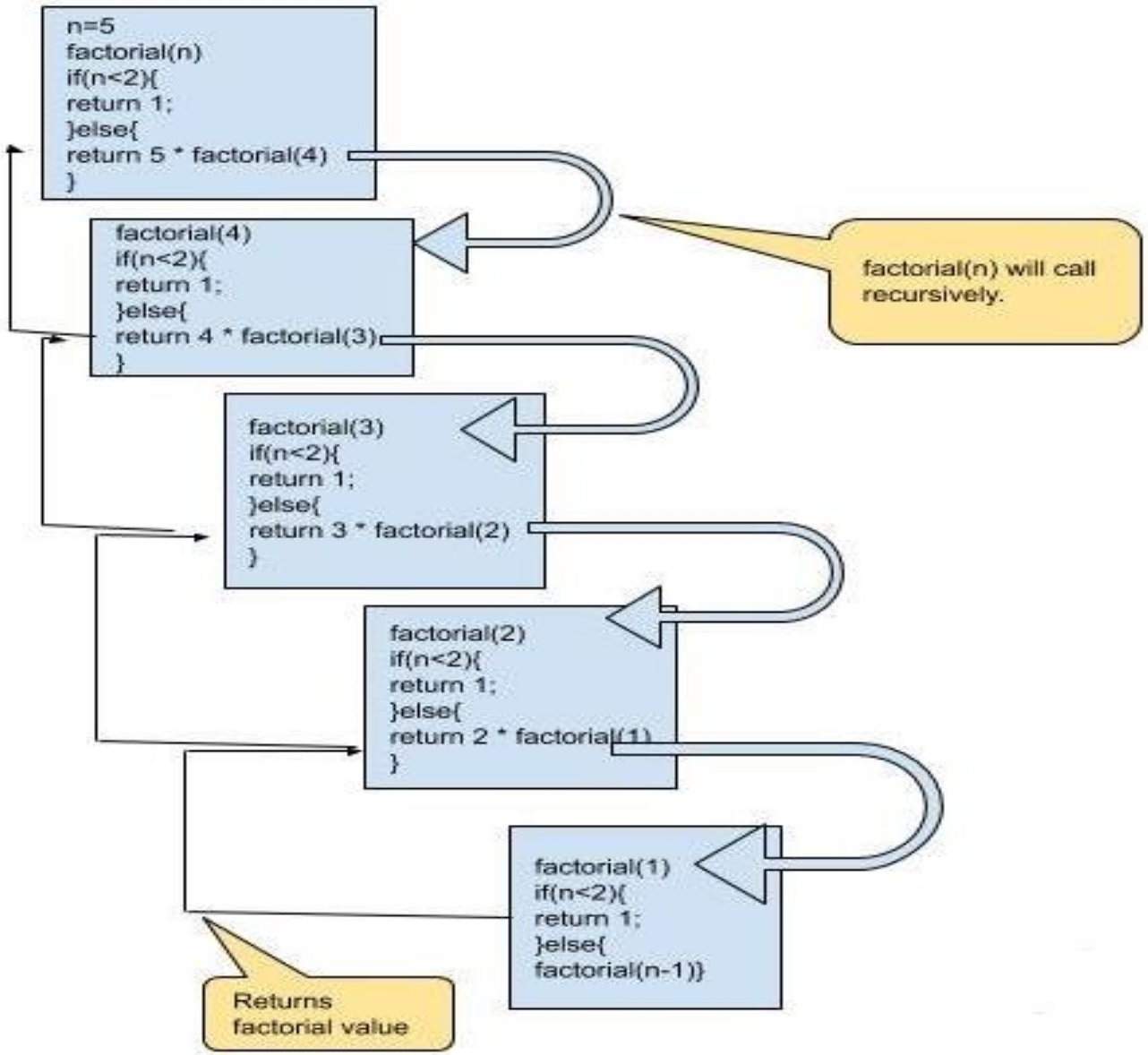
This phase terminates when the terminating condition becomes true in a call

Recursive function

Unwinding phase: It begins when the termination condition become true and function calls starts returning in reverse order till the first instance of function returns.

In this phase the control returns through each instance of function.

In some algorithms we need to perform some work while returning from recursive calls, so we write that particular code just after the recursive call




```
main()
{
   sysout(fact(5));
}

public static int fact(int n)
{
  if(n<2) return 1;


  else return n * fact(n-1);
}
```

```
main()
{
  fun(1);
}
```

```
void fun(int n)
{
 if(n>5)
       return;
 sysout(n);  //forward recursive
 fun(n+1);

 return;
}
```

forward recursive statement: The statements which gets executed during winding phase.

---

backward recursive statement: The statements which gets executed during unwinding phase.

```
void fun(int n)
{
 if(n>5)
  return;

 fun(n+1);
 sysout(n);  //backward recursive
 return;
}
```

Properties of recursive algorithm

1) A recursive algorithm must have a termination condition.
2) A recursive algorithm must change its state, and move towards the termination condition. Without termination condition, the recursive function may run forever and will finally consume all the stack memory
3) A recursive algorithm must call itself.

Note: The speed of a recursive program is slower because of stack overheads. If the same task can be done using an iterative solution (loops), then we should prefer an iterative solution (loops) in place of recursion to avoid stack overhead.

Why Recursion?

- Recursive code is generally shorter and easier to write than iterative code.
- Recursion is most useful for tasks that can be defined in terms of similar subtasks.
- For example, sort, search, and traversal problems often have simple recursive solutions.

Important points

- Recursive algorithms have two types of cases, recursive cases and base cases.
- Every recursive function case must terminate at a base case.
- Generally, iterative solutions are more efficient than recursive solutions [due to the overhead of function calls].
- A recursive algorithm can be implemented without recursive function calls using a stack, but it’s usually more trouble.
- Some problems are best suited for recursive solutions while others are not.

Example Algorithms of Recursion

- Fibonacci Series, Factorial Finding
- Merge Sort, Quick Sort
- Binary Search
- Tree Traversals and many Tree Problems: InOrder, PreOrder PostOrder
- Graph Traversals: DFS [Depth First Search] and BFS [Breadth First Search]
- Divide and Conquer Algorithms

Some more examples

```
P s v main()
{
   int arr[]= {10,20,30,40,50,60,70,80,90,100};
   display(arr,0);
}

static void display(int a[],int i)
 {
   if(i>9)
     return;
   display(a,i+1);
   sysout(a[i]);
 }
```

Some more examples

```
P s v main()
{
   int arr[]= {10,20,30,40,50,60,70,80,90,100};
   sysout(“Sum = “+ sum(arr,0));
}

static int sum(int []a,int i)
 {
    if(i>9)
        return 0;
   sysout(a[i]);
   return a[i] + sum(a,i+1);
 }
```

Some more examples

```
public static int cnt(int n)
{
  if(n/10 == 0) return 1;

  return 1 + cnt(n/10);
}
```

```
public static int sum_of_digit(int n)
{
  if(n/10 == 0) return n;

  return n%10 + sum_of_digit(n/10);
}
```

```
public static void displayR(int n)
{
  if(n/10==0)
  {
    sysout(n);
    return;
  }
  sysout(n%10);
  displayR(n/10);
}
```

```
public static int power(int a, int p)
{
 if(p==0) return 1;

 return a * power(a,p-1);
}
```

LinkedList and recursive function

```
static void disp(listNode temp)
{
   if(temp==null)  return;

   disp(temp.getNext());
   sysout(temp.getData());
}
```

Function Call:

```
disp(ll.getHead());
```

```
public static listNode reverse(listNode head)
{
  listNode temp;
  if(head.getNext()==null) return head;

  temp=reverse( head.getNext());
  head.getNext().setNext(head);
  head.setNext(null);
  return temp;
}
```

Function Call:

```
 list.setHead(reverse(list.getHead());
 sysout(list);
```

LinkedList and recursive function

```
int length(listNode *p)
{
   if(p == null) return 0;

   return 1 + length(p.getNext());
}
```

Function Call:

```
sysout(ll.length(ll.getHead()));
```

```
int sum_nodes(listNode *p)
{
   if(p == null) return 0;

   return p.getData() + sum_nodes(p.getNext());
}
```

Function Call:

```
sysout(ll.sum_nodes(ll.getHead());
```