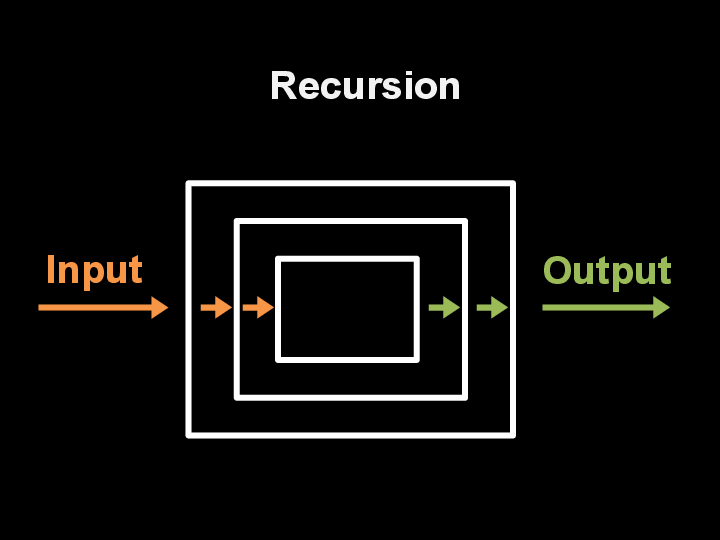
Recursion Notes [Java]

**Recursive algorithm**

It is an algorithm which calls itself with smaller (or simpler) input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller input. More generally if a problem can be solved utilizing solutions to smaller versions of the same problem, and the smaller versions reduce to easily solvable cases, then one can use a recursive algorithm to solve that problem. 

**Principal of Mathematical Induction**

Recursive programming is directly related to mathematical induction, a technique for proving facts about natural numbers. Proving that a statement involving an integer n is true for infinitely many values of n by mathematical induction involves the following two steps:

* The base case: prove the statement true for some specific value or values of n (usually 0 or 1).
* The induction step: assume that the statement to be true for all positive integers less than n, then use that fact to prove it true for n.

Such a proof suffices to show that the statement is true for infinitely many values of n: we can start at the base case, and use our proof to establish that the statement is true for each larger value of n, one by one.

Factorial

Your first recursive program. The "Hello, World" for recursion is the factorial function, which is defined for positive integers n by the equation

n!=n×(n−1)×(n−2)×…×2×1

The quantity n! is easy to compute with a for loop, but an even easier method is to use the following recursive function:

public static long factorial(int n) {

if (n == 1) return 1;

return n \* factorial(n-1);

}

We can trace this computation in precisely the same way that we trace any sequence of function calls.

factorial(5)

factorial(4)

factorial(3)

factorial(2)

factorial(1)

return 1

return 2\*1 = 2

return 3\*2 = 6

return 4\*6 = 24

return 5\*24 = 120

For factorial(), the base case is n = 1. The reduction step is the central part of a recursive function. The value of n decreases by 1 for each call, so the sequence of input values converges to the base case.

**Recursive strategies**

Head Recursion

public int factorial(int n) {

if (n == 0) {

return 1;

} else {

return n \* factorial(n - 1);

}

}

As you can see in above example, above function is calling itself with updated argument until termination condition is met. Let’s try above example with an initial value = 3. Note cs = Call Stack.

cs3: n = 0 ----> 1 returns 1

cs2: n = 1 ----> 1 \* factorial(0) returns 1 \* 1

cs1: n = 2 ----> 2 \* factorial(1) returns 2 \* 1 \* 1

cs0: n = 3 ----> 3 \* factorial(2) returns 3 \* 2 \* 1 \* 1

As you can see above, each recursive call freezes its local variables and makes and recursive call for updated n until n becomes 0. This method is called Head Recursion. You make a recursive call first then do the calculation once the call is back.

Tail Recursion

public int factorial(int n, int accum) {

if (n == 0) {

return accum;

} else {

return factorial(n - 1, accum \* n);

}

}

As you can see in above code, we are passing around variable accum which does the necessary computation first, then makes a recursive call. This means no need to preserve the stack frame from the previous iterative step, you can dump it and re-use the same stack frame with updated arguments. For this reason tail recursion does not cause stack overflow.

For n = 3

n = 3 factorial( 3 - 1, 1 \* 3 )

n = 2 factorial( 2 - 1, 1 \* 3 \* 2 )

n = 1 factorial( 1 - 1, 1 \* 3 \* 2 \* 1 )

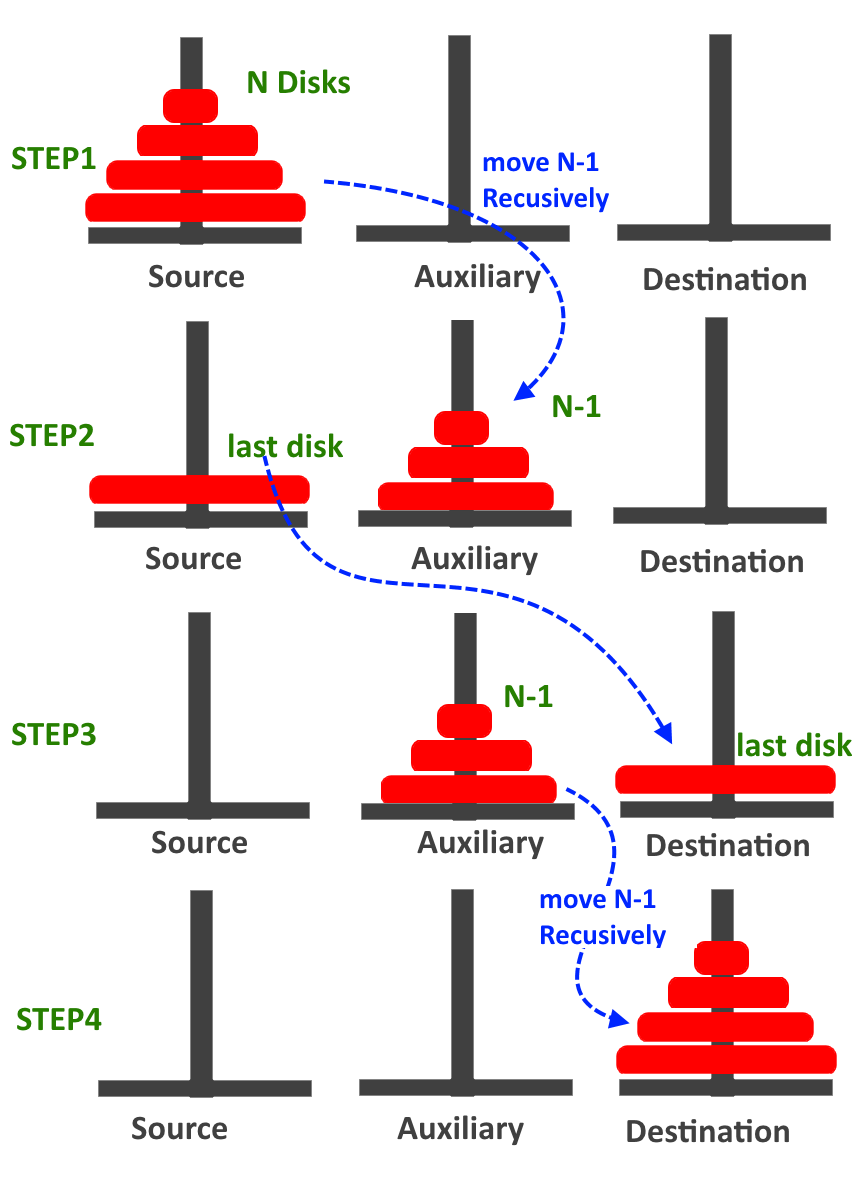
Unfortunately in Java, tail recursion is not supported, therefore above call will pile the unnecessary call stack therefore could have a potential danger of stack overflow.

Tower of Hanoi

In the towers of Hanoi problem, we have three poles and n discs that fit onto the poles. The discs differ in size and are initially stacked on one of the poles, in order from largest (disc n) at the bottom to smallest (disc 1) at the top. The task is to move all n discs to another pole, while obeying the following rules:

* Move only one disc at a time.
* Never place a larger disc on a smaller one.

Recursion provides just the plan that we need: First we move the top n−1 discs to an empty pole, then we move the largest disc to the other empty pole, then complete the job by moving the n−1 discs onto the largest disc. TowersOfHanoi.java is a direct implementation of this strategy.



public class MainClass {

public static void main(String[] args) {

int nDisks = 3;

doTowers(nDisks, 'A', 'B', 'C');

}

public static void doTowers(int topN, char from, char inter, char to) {

if (topN == 1) {

System.out.println("Disk 1 from " + from + " to " + to);

} else {

doTowers(topN - 1, from, to, inter);

System.out.println("Disk " + topN + " from " + from + " to " + to);

doTowers(topN - 1, inter, from, to);

}

}

}

Output:

Disk 1 from A to C

Disk 2 from A to B

Disk 1 from C to B

Disk 3 from A to C

Disk 1 from B to A

Disk 2 from B to C

Disk 1 from A to C

## GCD of two positive integers

The greatest common divisor (gcd) of two positive integers is the largest integer that divides evenly into both of them. For example, the gcd(102, 68) = 34. We can efficiently compute the gcd using the following property, which holds for positive integers p and q: If p > q, the gcd of p and q is the same as the gcd of q and p % q.

gcd(1440, 408)

gcd(408, 216)

gcd(216, 192)

gcd(192, 24)

gcd(24, 0)

return 24

return 24

return 24

return 24

return 24

This is Euclid's algorithm

public static int gcd(int p, int q) {

if (q == 0)

return p;

else

return gcd(q, p % q);

}

# Sorting using recursion

## Merge Sort

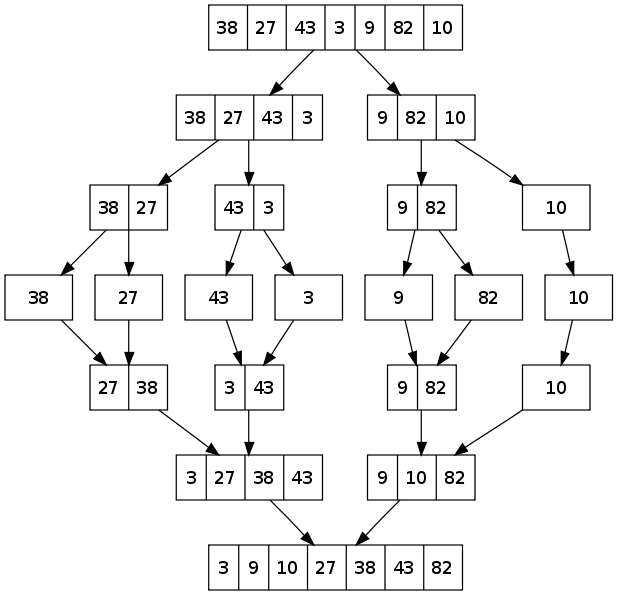
Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being Ο(n log n), it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

#### Algorithm

Step 1 − if it is only one element in the list it is already sorted, return.   
Step 2 − divide the list recursively into two halves until it can no more be divided.  
Step 3 − merge the smaller lists into new list in sorted order.

Note: This strategy is called Divide and conquer



class Sorting{

public static void MergeSort(int[] arr){

if(arr.length<=1)

return ;

int mid = arr.length/2,i=0,j=0;

int[] left = new int[mid];

int[] right = new int[arr.length-mid];

for(i=0;i<left.length;i++){

left[i]=arr[i];

}

for(;j<right.length;j++){

right[j]=arr[mid+j];

}

MergeSort(left);

MergeSort(right);

merge(left,right,arr);

}

public static void merge(int[] arr1,int[] arr2,int[] arr3){

int i=0,j=0,k=0;

for(;i<arr1.length && j<arr2.length;k++){

if(arr1[i]<arr2[j])

arr3[k]=arr1[i++];

else

arr3[k]=arr2[j++];

}

while(i<arr1.length)

arr3[k++]=arr1[i++];

while(j<arr2.length)

arr3[k++]=arr2[j++];

}

public static void main(String[] args) {

int arr[] = { 5,4,3,2,1 };

MergeSort(arr);

for(int i:arr){

System.out.print(i+" ");

}

}

}

Output:

1 2 3 4 5

## Quick Sort

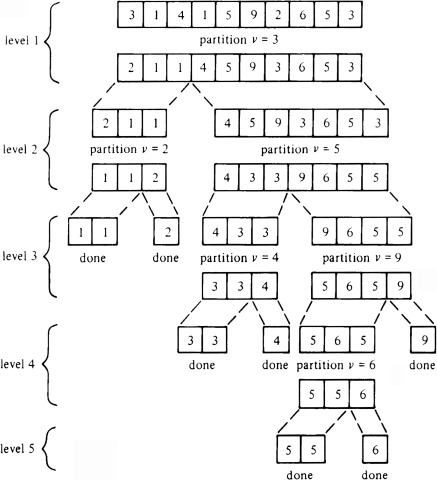
Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

* Always pick first element as pivot.
* Always pick last element as pivot (implemented below)
* Pick a random element as pivot.
* Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time. This algorithm is quite efficient for large-sized data sets as its average and worst case complexity are of Ο(nlogn), where n is the number of items.

#### Algorithm

Step 1 − Choose the highest index value has pivot  
Step 2 − Take two variables to point left and right of the list excluding pivot  
Step 3 − left points to the low index  
Step 4 − right points to the high  
Step 5 − while value at left is less than pivot move right  
Step 6 − while value at right is greater than pivot move left  
Step 7 − if both step 5 and step 6 does not match swap left and right  
Step 8 − if left ≥ right, the point where they met is new pivot



class Sorting{

public static void sort(int[] input,int beg,int end){

if(beg>=end)

return;

int pivot = partion1(input,beg,end);

sort(input,beg,pivot-1);

sort(input,pivot+1,end);

}

public static int partion1(int[] array,int beg,int end){

int x = array[beg];//pivot point

int i= end+1;

int j= end;

int temp=0;

for(;j>beg;j--){

if(array[j]>=x){

i--;

temp = array[i];

array[i] = array[j];

array[j] = temp;

}

}

i--;

temp = array[i];

array[i] = array[beg];

array[beg] = temp;

return i;

}

public static void main(String[] args) {

int arr[] = { 5,4,3,2,1 };

sort(arr,0,arr.length-1);

for(int i=0;i<arr.length;i++){

System.out.print(arr[i]+ " ");

}

}

}

Output:

1 2 3 4 5

### Why Quick Sort is preferred over MergeSort for sorting Arrays

Quick Sort in its general form is an in-place sort (i.e. it doesn’t require any extra storage) whereas merge sort requires O(N) extra storage, N denoting the array size which may be quite expensive. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm. Comparing average complexity we find that both type of sorts have O(NlogN) average complexity but the constants differ. For arrays, merge sort loses due to the use of extra O(N) storage space. Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of O(nLogn). The worst case is possible in randomized version also, but worst case doesn’t occur for a particular pattern (like sorted array) and randomized Quick Sort works well in practice.