The methods in java gets stored as stacks in the memory, one on top of another

When the main method calls the other functions, it gets stored above main, and when that function returns a value, that stack which stored the function gets cleared, when main method is done it also gets cleared.

Methods and Functions:

Methods called through objects of class

Functions called directly

Time Complexity: Relation between Input Size and Running Time (Operations)

Time complexity directly proportional input size

Best case : Minimum

Average Case: average

Worst Case: wont exceed this time

3 ways to do time complexity

Best Case : omega(1)

Avg: theta(n+1/2)

Worst: O(n) //Time Complexity for competitive coding

SPACE COMPLEXITY

1 SPACE COMPLEXITY: normally if you have some variables in your code those will be the space complexity, even if you increase the input size of a variable it wont affect the space complexity of the program.

2 Space complexity

Arrays: In case of arrays if you increase the number of elements to be stored in the array, it will increase the space complexity

Array initialization

Int [] arr = new int [10]; // new: used for making a new space inside memory

Memory address gets stored in hexadecimal

Default initialization:

Objects: null

Int:0

Float: 0.0

String:””

Boolean: false

Array: depends of type of array

2D Array :

Int [][] arr = new int[rows][columns];

We use. CompareTo() method to check equality of two strings s1 and s2

Eg: s1.CompareTo(s2);

It returns 0 if s1 = s2

+ve if s1> s2 (a is lowest, z is highest, if equal goes to next character )

-ve if s1<s2

When we initialize a variable

String s = “Alan Saji”;

s gets stored in stack

Alan Saji gets stored in heap

<< left shift: shift to left with zeros adding on places 3 << 1 =? 010 << 1 = 100

>> right shift: shift to right with zeros adding on places

Bit manipulation

Get:

Set: set to 1 if its 0 or keep it 1

Clear: set to 0

Update: if 1 then 0 , if 0 then 1

GET BIT OPERATION

Get the 3rd bit (position 2) of a number n. (n=0101) {position counting starts from right to left starting with zero}

* 0 1(3rdbit) 0 1

Process:

Bit Mask: 1<<i (we have to do this bit mask manipulation ( bit mask is an extra number we have to apply in the process) (here i is the position)

Operation: AND (we have to do ‘and’ operation with the original n and the bit mask )

Execution:

* 1<<2 = 0001 << 2 = 0100
* 0100 & 0101(original n)
* 0100 => this implies the 3rd bit was 1 because the result is non-zero, if it was 0000 then the 3rd bit would be zero (we got the 3rd bit by GET operation)

SET BIT OPERATION

Set the 2nd bit (position 1) of a number n. n = 0101

Process:

Bit mask: 1<<i

Operation: OR

Execution:

* 0001 << 1 = 0010
* 0010 OR 0101 => 0111 (hence we set the 2nd bit to 1) the number 5 has changed to 7

CLEAR BIT OPERATION

Clear the 3rd bit of a number n , n = 0101

PROCESS:

Bit mask = 1 << i

Operation : AND with NOT( of bitmask)

Execution:

* 1 << 2 = 0001 << 2 = 0100
* !(0100) => ~(0100) => 1011
* 1011 AND 0101 => 0001 ( 0001 is one in decimal

UPDATE BIT OPERATION:

Update the 2nd bit position(1) of a number n to 1 , n = 0101

For 0:

Bitmask : 1<<i

Operation : AND with NOT (of bitmask)

For 1:

Bitmask: 1<<i

Operation : OR

SORTING

BUBBLE SORT, SELECTION SORT, INSERTION SORT

Bubble sort is where the elements get bubbled up in one side while we traverse the array

package alan;

import java.util.Arrays;

public class BubbleSort {

// the highest or the lowest number gets bubbled up on one side of the array

static void bubblesort(int[] arr){

//outer loop goes till the end of array

for (int i = 0; i < arr.length-1; i++) {

//inner loop goes till the point where its not already sorted

for (int j = 0; j < arr.length -i - 1; j++) {

//ascending

//here te smaller one goes to right side and larger one gets bubbled on left

if(arr[j] > arr[j+1]){

int temp = arr[j];

arr[j] = arr[j+1];

arr[j+1] = temp;

}

}

}

System.out.println(Arrays.toString(arr));

}

public static void main(String[] args) {

int [] arr = {7,8,4,6,1,2,7,6,6,8,3,34,554,5,6,65,5,57,7,5,746,7,65,7,567,56,7,567,56,7,567,43,64,63};

bubblesort(arr);

}

}

Time complexity :

O(n^2)

SELECTION SORT

Only one swap in one iteration

package alan;  
  
import java.util.Arrays;  
  
public class SelectionSort {  
 public static void main(String[] args) {  
 int [] arr = {1,2,4,6,2,5,7,9,3,2};  
  
 //selection sort  
 //first loop goes till end ,  
 for (int i = 0; i < arr.length; i++) {  
 int smallest = i;  
 //second loop starts from the next element and checks for smallest  
 for (int j = i+1; j <arr.length ; j++) {  
 if(arr[smallest] > arr[j]){  
 smallest = j;  
 }  
 }  
 //swapping (if smallest was not found, no change will happen)  
 //here we are swapping smallest to the beginning  
 int temp = arr[smallest];  
 arr[smallest] = arr[i];  
 arr[i] = temp;  
  
 }  
 System.*out*.println(Arrays.*toString*(arr));  
  
 }  
}  
  
// time complexity is n^2

INSERTION SORT

package alan;  
  
import java.util.Arrays;  
  
public class InsertionSort {  
 public static void main(String[] args) {  
 int [] arr = {1,4,5,2,4,6,2,1,6,8,9};  
  
 //insertion sort: here we consider a sorted part and an unsorted part  
 //then we slowly add elements in the sorted part and making a complete sorted array  
  
 for (int i = 1; i < arr.length; i++) {  
  
 int current = arr[i];  
 int j = i - 1;  
 while(j>=0 && current < arr[j]){  
 arr[j+ 1] = arr[j];  
 j--;  
 }  
 //placement  
 arr[j+1] = current;  
  
 }  
 System.*out*.println(Arrays.*toString*(arr));  
  
 }  
}  
//time complexity n^2

RECURSION

Outer function , Inner function

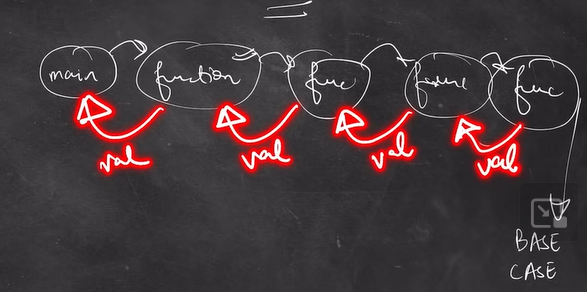
Outer function assumes the inner function to do certain works

Outer function do some “steps”

Inner does rest of the work

Then the inner function return an output and gives to the outer function

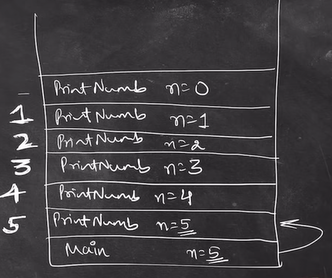
In recursion the function keeps on calling itself till some base case is not satisfied



CODE:

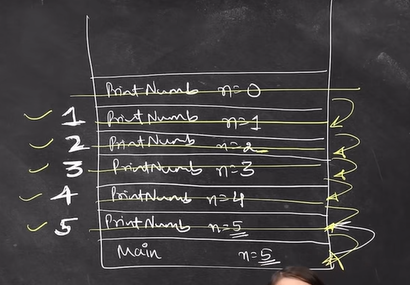
package alan;  
  
public class recursion {  
  
  
 static void printNum(int n){  
 //base case  
 if(n==0){  
 return;  
 }  
 //print statement  
 System.*out*.print(n+" ");  
 *printNum*(n-1);  
 }  
  
 public static void main(String[] args) {  
 // print the numbers from 5 to 1 using recursion  
 int n = 5;  
 *printNum*(n);  
  
  
 }  
}

the recursive functions get stored in form of stack in the memory



After return the functions go back to the functions calls from which it were called

And a cascading effect happens with the remaining functions



For every n used in the recursion calls a different memory space is allotted

Print sum of first n natural numbers

package alan;  
  
public class recur3 {  
 static int *sum* = 0;  
 public static void print(int n){  
  
 if (n == 0) {  
 return;  
 }  
 *sum* = *sum* + n;  
 *print*(n-1);  
  
 }  
 public static void anotherMethod(int i , int n , int sum){  
 if(i == n){  
 sum += i;  
 System.*out*.println(sum);  
 return;  
 }  
  
 sum+=i;  
 *anotherMethod*(i+1,n,sum );  
 }  
  
  
 public static void main(String[] args) {  
 // print sum of first n natural numbers  
  
 int n = 10;  
 *print*(n);  
 *anotherMethod*(0, 10, 0);  
 System.*out*.println(*sum*);  
  
  
 }  
}