QUICK SORT :

package day18;

import java.util.\*;

public class task1 {

public static void quickSort(int arr[],int start , int end)

{

if(start<end)

{

int parind=*partition*(arr,start,end);

*quickSort*(arr, start, parind - 1);

*quickSort*(arr, parind + 1, end);

}

}

public static int partition(int arr[],int start , int end)

{

int pivot = arr[end];

int i = start - 1;

for (int j = start; j < end; j++) {

if (arr[j] < pivot) {

i++;

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

int temp = arr[i + 1];

arr[i + 1] = arr[end];

arr[end] = temp;

return i + 1;

}

public static void main(String args[]) {

Scanner s = new Scanner(System.***in***);

int arr[] = Arrays.*stream*(s.nextLine().split(" ")).mapToInt(Integer::*parseInt*).toArray();

int n = arr.length;

*quickSort*(arr,0,n-1);

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

}

}

Insertion sort :

package day18;

import java.util.\*;

public class task2 {

public static void insertionSort(int arr[], int n) {

for (int i = 1; i < n; i++) {

int key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) {

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

j = j - 1;

}

arr[j + 1] = key;

}

System.***out***.println("Sorted array: " + Arrays.*toString*(arr));

}

public static void main(String args[])

{

Scanner s = new Scanner(System.***in***);

int arr[] = Arrays.*stream*(s.nextLine().split(" ")).mapToInt(Integer ::*parseInt*).toArray();

*insertionSort*(arr,arr.length);

}

}

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Algor name | Best case | Avg case | worst case | space | stable | Use case |
| Bubble sort | 0(n) | 0(n^2) | 0(n^2) | O(1) | yes | Small data set and sorted |
| Selection sort | O(n^2) | O(n^2) | O(n^2) | O(1) | no | Swap are smaller |
| Insertion sort | O(n) | O(n^2) | O(n^2) | O(1) | yes | Small data set and nearly sorted data |
| Merge sort | O(n log n) | O(n log n) | O(n log n) | O(n) | yes | Large data set and linked list |
| Quick sort | O(n log n) | O(n log n) | O(n^2) | O(log n) | no | General purpose sorting and efficient in happens |
| Heap sort | O(n log n) | O(n log n) | O(n log n) | O(1) | no | Priority queue and real time application |

Bubble , selection 🡪 are simple but inefficient for large data set .

Insertion sort 🡪best for small and nearly sorted array .

Merger sort 🡪 it is stable DC algorithm and good for linked list .

Quick sort 🡪 its past on avg but cant be trusted when pivot are bad .

Heap sort 🡪 ensures o(n log n) always but it is not stable .

**RECURSION :**

package day18;

import java.util.Scanner;

public class task4{

public static int fact(int n) {

if (n == 0 || n == 1) {

return 1;

}

return n \* *fact*(n - 1);

}

public static void main(String args[])

{

Scanner s = new Scanner(System.***in***);

int n = s.nextInt();

System.***out***.println(*fact*(n));

}

}

5

120

Recursion is a technique where a function call it self to solve a problem in a smaller sub problem until the base condition is met .

Recursion is more general expensive and directly using the stack .

Due to this over head of function called memory consumption is high .

Recursion becomes expensive and also type consuming

**Key component of data structure .**

Base case

Recursive case .

package day18;

import java.util.Scanner;

public class task3 {

public static void fact(int n)

{

int fact =1;

for(int i=n;i>0;i++)

{

fact = fact \*i;

System.***out***.println("loop");

}

System.***out***.println(fact);

}

public static void main(String args[])

{

Scanner s = new Scanner(System.***in***);

int n = s.nextInt();

*fact*(n);

}

}

The above code is not solid case the code balance to stop the code saving the execution cycle.

-------------------------------------------------------------------------------------------------------------------------------------

package day18;

import java.util.Scanner;

public class task4{

public static int fact(int n) {

if (n == 0 || n == 1) {

return 1;

}

return *fact*(n \* *fact*(n - 1));

}

public static void main(String args[])

{

Scanner s = new Scanner(System.***in***);

int n = s.nextInt();

System.***out***.println(*fact*(n));

}

}

at java/day18.task4.fact(task4.java:10)

at java/day18.task4.fact(task4.java:10)

at java/day18.task4.fact(task4.java:10)

at java/day18.task4.fact(task4.java:10)

the above code met the error and end up in continuously recursion call passing stack overflow .

thus at scenario where recursion can be replaced by the iteration it is recommended to follow iteration .

**each time function is called recursion the following step happend:**1.the functional parameter and local var is store on the call stack .

2.the return add is execution address after the function is return to store

3.the function execution a make all it self again .

4.when the function return it call from the stack and resume the execution at the same return address.

5.this tack overhead throws with the depth of recursion .

|  |  |
| --- | --- |
| **RECURSIVE** | **ITERATIVE** |
| Uses the call stack which is limited | Uses the explicit |
| Can leave to stack overflow error if recursion depth is to high | No function call overhead |
| Function calls extra data that is parameter , return address , local variable . | More memory efficient has it avoid storing it address and function variable in stack frames. |
| Use when back track is needed . ex: tree | When the performance is critical use the iteration . |
|  |  |

package day18;

public class task5 {

public static long pow(long base , long expon)

{

if(expon==0)

{

return 1;

}

long temp = *pow*(base,expon/2);

if(expon%2==0)

{

System.***out***.println(temp + "\*" + temp);

return temp\*temp;}

else {

System.***out***.println(base + "\*" + temp + "\*" + temp);

return base \* temp \* temp;}

}

public static void main(String args[])

{

int base=3 , expon=13;

System.***out***.println(*pow*(base,expon)); }}

Modulo :

(a \* b ) % s 🡪( (a % s) \* (b % s))%s;