Recursion :-

Illustrative Examples :

Def: Recursion is a technique by which a method makes one or more calls to itself during excution, or by which a dataStructure relies upon smaller instances of the very same type of DataStructure in it's representation. (Tree DataStructure)

In Computing, Recursion provides an elegant and powerful alternative for performing repetitive tasks.

In fact, a few programing languages do not explicitly support looping Constructs and instead rely on Recursion Directly to express repetition.

now we will introduce you to some examples on Recursion.

1 factorial funation :-

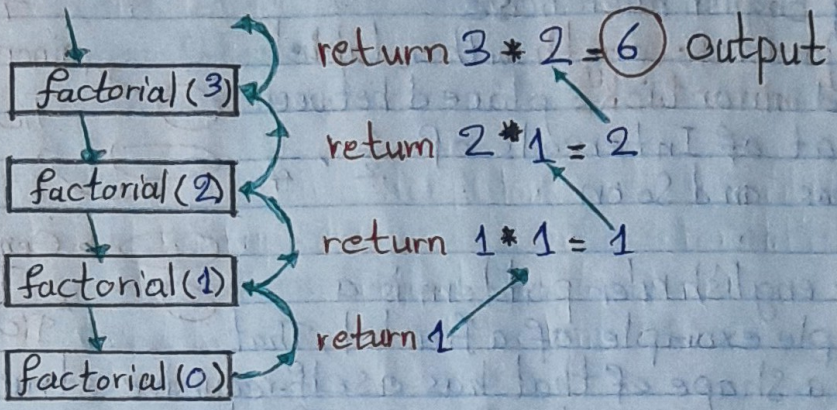
Code:

```
public static int factorial(int n)
    throws IllegalArgumentException {
  if (n<0) throw new IllegalArgumentException();
  else if (n==0) return 1; // Base Case
  else return n * factorial(n-1); // Recursive case
}
```

Handling non-positive entries

we will show you the Track of this execution :

The following figure is indicate to something Called Recursion trace &



A Recursion trace Closely mirrors a programing languages ~~exeution~~ execution of the Recursion.

In java, each time a method (recursive or Otherwise) is called, a structure Known as an activation record or activation frame is created to Store information about the progress of That invocation of The method.

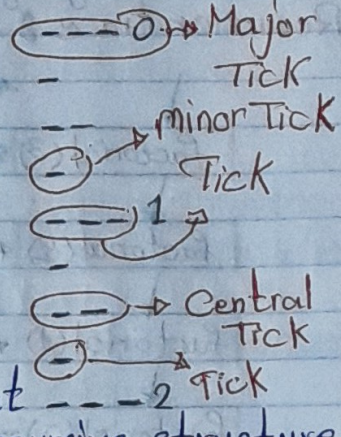
★ Recursion.
★ Thinking:

when the execution of a method leads ta nested method call, The execution of the former Call is Suspended and its frame stores the place in the Source code at which the flow of control should Continue upon return of The nested Call.

The key point is to have a Separate frame for each active call.

now let's go on amazing example on recursion.

2 Drawing an English Ruler :-



any English ruler has a major tick length with a label and minor ticks placed between or at of Intervals of $\frac{1}{2}$, $\frac{1}{4}$ inches and So on

The english ruler pattern is a Simple example of a fractal, That is, a shape of that has a self-recursive structure at various Levels of magnification.

In general an interval with a central tick length $L \geq 1$ is composed of:

- An interval with a central tick length L-1.
- A single tick of length L.
- An interval with a central tick length L-1.

we will design 4 methods:

1. drawRuler(int nInches, int majorLength)
2. drawInterval(int centralLength)
3. drawLine(int tickLength, int tickLabel)
4. drawLine(int tickLength)

Let's implement our multiRecursive methods:

Sample input: 1 3

Code:

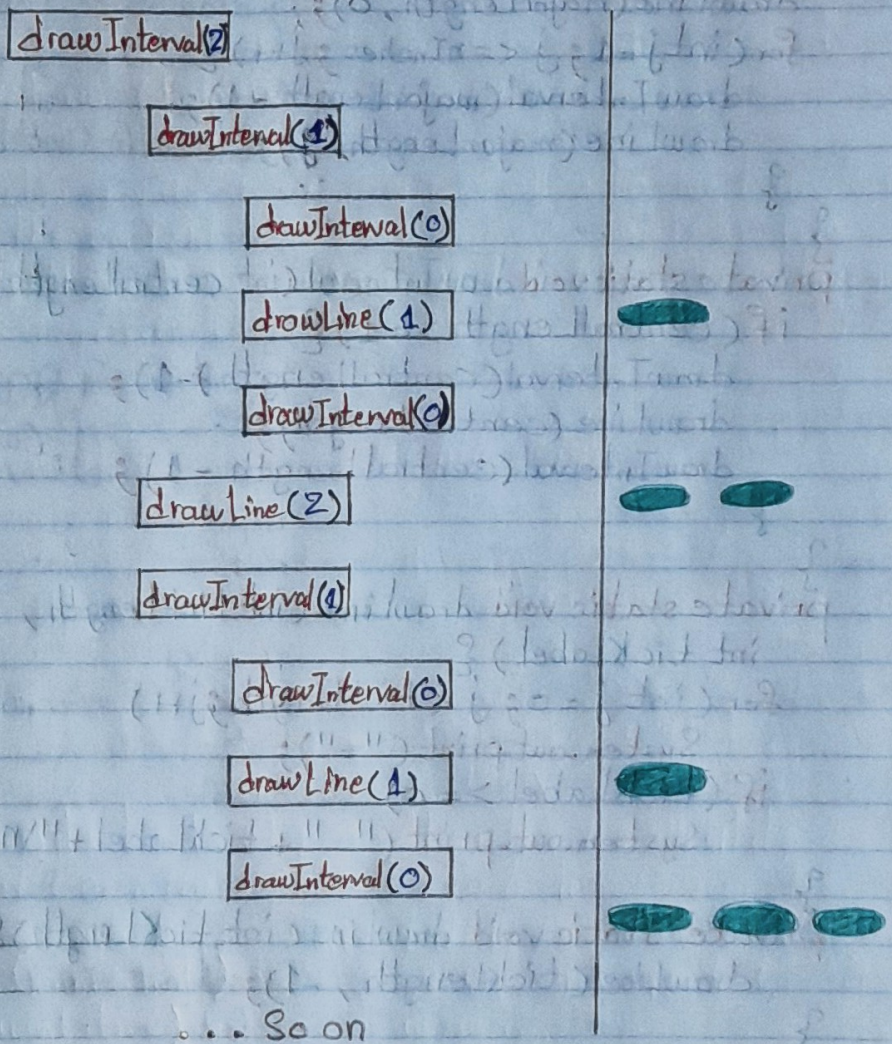
```
public static void drawRuler(int nInches, int majorLength){
    drawLine(majorLength, 0); → draw first Tick
    for(int j=1; j <= nInches; j++) {
        drawInterval(majorLength - 1);
        drawLine(majorLength, j); → draw Tick with label
    }
}
private static void drawInterval(int centralLength) {
    if( centralLength >= 1) {
        drawInterval(centralLength-1);
        drawLine(centralLength);
        drawInterval(centralLength - 1);
    }
}
private static void drawLine(int tickLength,
    int tickLabel) {
    for(int j = 0; j < tickLength; j++)
        System.out.print("-");
    if (tickLabel >= 0)
        System.out.print(" " + tickLabel + "\n");
}
private static void drawLine(int tickLength){
    drawLine(tickLength, -1); → draw Tick without label
}
```

} Recursive calls

outputs:

```
-
-
-
-
---1
```

Let's explain what happend in the RecursiveTrace 8



and all the previous steps is ~~repete~~ repeated.

now we can use recursion in Searching Algorithms like Binary Search.

3 Binary Search :-

Codes-

```
public static boolean binarySearch(int[] data,
    int target, int low, int high){
  if (low > high) return false;
  else {
    int mid = (low + high) /2;
    if (target == data[mid]) return true;
    else if (target < data[mid])
      return BinarySearch(data, target, low, mid-1);
    else
      return BinarySearch(data, target, mid+1, heigh);
  }
}
```

for ~~this~~ a That approach of search you should Sort your data and it's depend on 3 steps:

- Target = data[mid] BaseCase True
- Target > data[mid] RecursiveCase search to high
- Target < data[mid] RecursiveCase search to low

The Steps of That Algorithm is Like The following Sequence:

$$\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \dots \text{ Soon } = \log_2 n$$

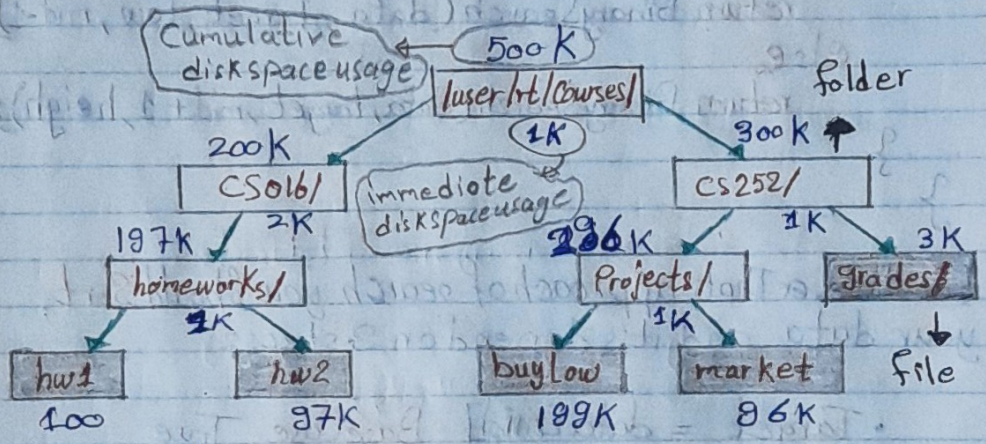
Then The Time Complexity is $O(\log n)$. It's Better Than The LinearSearch $O(n)$.

now we have The last important example on ~~Re~~ Recursion:

4 File Systems :-

Modern operating Systems define file System & directories (Folders) in a Recursive way.

Namely, a filesystem Consists of a top level directory, and the contents of this directory Consists of files and other directories, which in turn can Contain files and other directories, and so on ...



Our goal is to develop an Algorithm To calculate The Total Capacity of the current directory or The disk usage of the current directory by computing The (immediate) disk space usage and sum it to get (Cumulative) disk space usage.

```
Algorithm: DiskUsage(path)
Input: path of the directory   Output: Total Disk space
{total = size(path)
 if path represents a directory then
    for (each child entry stored within dir.Path do
        total += DiskUsage(child)
 return total }
```

for implementing the algorithm we should use File Class because it has 4 important methods:

1. newFile(pathString) or new File(parentFile, childString)
   Constructing a new file or a new child.
2. file.length() measured in bytes.
   return the size of the file.
3. file.isDirectory()
   return true if file instance represents a directory.
4. file.list()
   return an array of entries of the given directory.

Code:

```
public static Long diskUsage(File root){
    Long total = root.size();
    if(root.isDirectory()){
        for(String childname : root.List()){
            File child = new File(root, childname);
            total += diskUsage(child);
        }
    }
    System.out.println(total += "\t" + root);
    return total;
}
```

Recursive Trace:

/user/rt/Courses/Cs016/homeworks/hw1
/user/rt/Courses/Cs016/homeworks/hw2
/user/rt/Courses/Cs016/homeworks
/user/rt/Courses/Cs016
/user/rt/courses/cs252/Projects/baylaw
/user/rt/courses/cs252/Projects/market and so on....

Analyzing Recursive Algorithms:

with
~~write~~ a recursive algorithm, we will account for each operation that is performed based upon the particular activation of the method that manages the flow of control it is executed.

Calculating Time Complexity:

stated another way, for(each invocation) of the method we only account for the number of operations that are performed within the body of that activation

we can then account for the overall number of operations that are executed as part of the recursive algorithm by taking the sum overall activations of the number of operations that take place during each individual ~~invo~~ activation.

1. Computing factorial n :-

$n * (n-1) * (n-2) * \cdots * 3 * 2 * 1 *$ factorial(0)

(n steps; factorial(0): one step)

So time complexity = $c(n+1) = O(n)$

(c: worst case, 1: best case)

2. Drawing an English Ruler :-

we consider here the fundamental question of how many total lines of output are generated by an initial call to drawInterval(c) where c denotes the center length.

drawInterval(0) Do nothing and for both upInterval and DownInterval are making 2^0 output Then The Time Complexity Should be Zero.

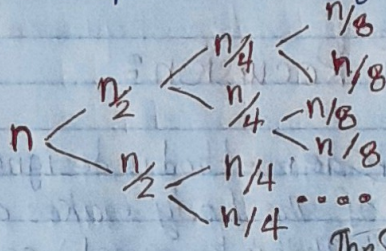
$$2^0 + ? = 0 \Rightarrow 2^0 - 1 = 1 - 1 = 0$$

Then the formal Time complexity $= O(2^n + 1) = O(2^n)$
worstCase ↳ BestCase

3 Binary Search :-

we here Should count The elements we are Searching in The search Space. Let's say, we have n elements

Then



high - low + 1 = n

.... So on. until $\frac{n}{2^r} < 1$
The Smallest integer in The Search Space

if we calculated it 8

1step halfstep

$$n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \dots\dots = \log_2(n)$$

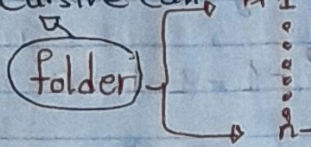
Then the formal Time Complexity $= O(\log n + 1) = O(\log n)$
worstCase ↳ bestcase

4 Computing Disk Space Usage :-

~~Hear~~ here we focus on how many Times The recursive call will be executed.

Let's say we have a file System with n ~~entries~~ intries:

1 recursive call, ☆1



So we have n-1 recursive calls for n-1 files

Then The Time Complexity = O(n-1+1) = O(n)

now let me say Something else. you may notice That we always That every call will be in the worst case ~~is ate~~ That Technique is called amortization.

Further Examples of Recursion:

def: If a recursive ~~call, is de~~ method is designed So That each invocation of The body makes at most one new recursive call, This is know as Linear Recursion.

ex:

```
/** Sum of Array elements */
public static int linearSum (int[] data, int n){
   if (n==0) return 0;
   else return linearSum (data, n-1)+ data[n-1];
}//O(n)
```

↳ one Recursive call for each step.

ex:

```
/** Reversing Elements */
public static void reverseArray(int[] data, int Low,
    int high) {
  if ( low < high){ int temp =data[low];
     data[low] = data[high]; data[high]=temp;
     reverseArray (data, low+1, high-1);}}//O(n)
```

ex:

```
/** Computing power(n, i) -> n^i */
public static double power(double x, int n){
    if (n==0) return 0;
    else return x * power(x, n-1);
} //O(n) -> x^n = x · x^(n-1)
```

ex:

```
/** Computing power fast */
public static double power(double x, int n) {
    if (n == 0) return 1;
    else {
        double partial = power(x, n/2);
        double result = partial * partial;
    if (n % 2 == 1) result *= x;
    return result;
    }
} //O(logn) -> x^n = (x^(n/2))^2 if even(n) | x^n = (x^(n/2))^2 · x if odd(n)
```

def: when a method makes two recursive calls, we say that it is uses Binary Recursion.

ex:

```
/** Sum of elements an array by Divide it */
public static int binarySum(int[] data, int low,
    int high) {
    if (low > high) return 0;
    else if (low == high) return data[low];
    else {
        int mid = (low + high) / 2;   // Two Recursive calls in step.
        return binarySum(data, low, mid) +
               binarySum(data, mid+1, high);
    }
} //O(n): Time. Complexity - O(logn): Space Complexity
```

def: we define Multiple Recursion as a process in which a method may make more than two Recursive calls

as an example for it diskUsage(Path) method.
for any folder there are K recursive calls if it has K inbries.

ex: as more complex example we can develope a method To Solve puzzleGame and we need to assign a unique digit (0,1,2,3,4,5,6,7,8,9) to each letter in the equation, in order to make the equation True.

we should follow those steps:

1. Recursively generating the Sequences of K-1 elements.
2. Appending to each such Sequence an element not already Contained in it.

you can think of it as Taking all possible subsets and test them if they satisfying the Solution.

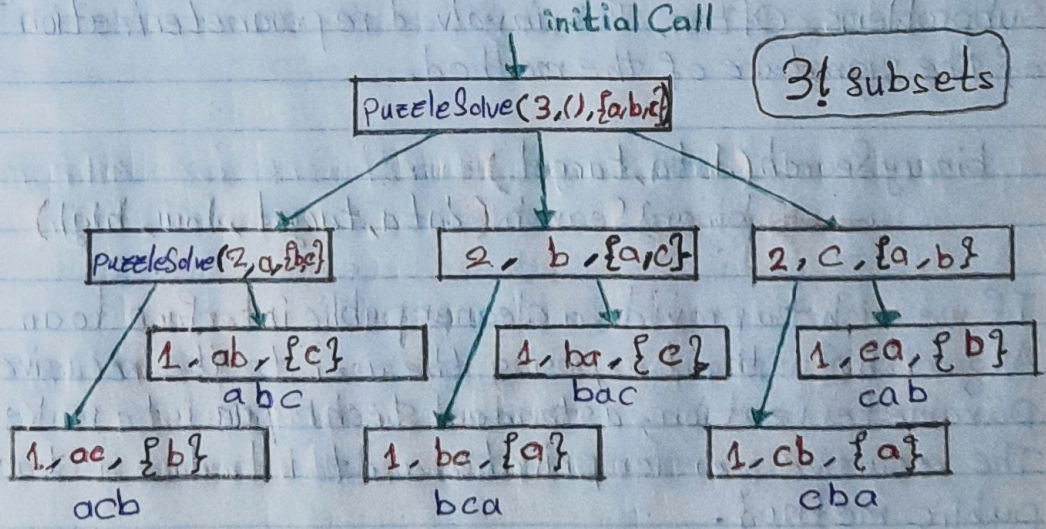
Let's say we have K elements in ordered Subsets of U = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} and each position in the sequence corresponds to a given letter.

now let's design our Algorithm for th summation Puzzles.

Algorithm: puzzleSolve (K, S, U)

Inputs: An integer K, sequence S, and set U.
outputs: An enumeration of all K-length extensions to S using elements in U without repeatation.

```
for each e in U do
    Add e to the end of S
    Remove e from U
    if K == 1 then
        Test whether S is a configuration that solves the
        puzzle
        if S solves the puzzle then
            add S to output
    else
        puzzleSolve (K-1, S, U)
    Remove e from the end of S
    Add e back to U
```



Designing Recursive Algorithms :

you may notice that all recursive algorithms based on 2 cases BaseCase and Recursive case.

The (Base cases) are designed to terminate The flow of program at that moment.

The (Recursive Cases) are designed to reach the base case at some point The recursion trace.

Thinking in Recursive Problems

To design a recursive algorithm for a given problem, it is useful to think of The different ways we might define subproblems that have The same general structure as the original problem

A successfull design sometimes requires that we redefine The Original Problem to facilitate similar looking subproblems. Often, This involved reparameterization of the signature of the method.

ex:

binarySearch(data, target)
→ binarySearch(data, target, low, high)

If we wish to provide a cleaner public interface to an Algorithm without exposing the user to the recursive parameterization, a standard technique is to make The recursive version private, and to introduce a cleaner public method.

Recursion Run Amok:

Although recursion is a very powerful tool, it can easily be misused in various ways.

ex: Let's say we want to develope a program that state if the array contains distinct numbers or not. Show the following Syntax:

Code:
```
/** Recursive unchecked elements */
public static boolean unique3(int[] data, int low,
    int high) {
  if (low >= high) return true;
  else if (!unique3(data, low, high - 1)) return false;
  else if (!unique3(data, low + 1, high)) return false;
  else return (data[low] != data[high]);
}
```

The previous method didn't checkedon the first and last elements of the array.

ex: anothe misuese for the recursion is to use more than needed Recursive calls to do something as getting fibonacci nth number. Show the following Syntax:

codesex:
```
public static long fibonacciBad(int n) {
  if (n <= 1) return n;
  else return fibonacciBad(n-2) + fibonacciBad(n-1);
}
```
2 Recursive calls in the step

Let's improve our fibonacci Algorithm. Show the following Syntax:

Code:

```
public static long[] fibonacciGood(int n) {
    if (n <= 1) {
        Long[] answer = {n, 0};
        return answer;
    } else {
        Long[] temp = fibonacciGood(n-1);
        Long[] answer = {temp[0] + temp[1], temp[0]};
        return answer;
    }
}
```

one Recursive call in Step

ex: another Comonon way it to make infinite Recursion. Show the following Syntax:

Code:

```
public static int fibonacci (int n) {
    return fibonacci (n); // Fn = Fn
}
```

notes: A programer Should ensure that each recursive call is in Some way progressing toward a base case.

Java Virtual Machien Machine Provides or throws StackOverflowError. The precise value of this limit depends upon The Java installation, but a typical value might allow upward 1000 simultaneous calls.

note: we can reconfigure The JVM To store Greater Space for nested calls by setting the Xss runtime option.

Eliminating Tail Recursion 8

def: A recursive function is said to be Tail Recursive if the recursive call is the last thing done by the function. There is no need to keep record of the previous state.

ex:

```
public static void fun(int n){
    if(n==0) return ;
    else system.out.println(n);
    return fun(n-1); // The last thing in the recursive
}// Tail Recursion is a Linear Recursion // case.
```

Tail Recursion

def: A recursive function is said to be NON-Tail Recursive if the Recursive call is not the last thing done by the function. After returning back, there is something left to evaluate.

ex:

```
public static int fun(int n){
    if(n==1) return 0;
    else {return 1 + fun(n/2); // The last thing is
}                              // Addition Operation
```

eliminating

NoN-Tail Recursion

we have benefits of eliminate Tail Recursions

1. avoiding StackOverflowError and reducing the ~~Spee~~ Space complexity of the algorithm (Recursive Algorithm).
2. Maintaining our problem as we want.