Syllabus

Introduction: Algorithm, Performance Analysis-Space Complexity, Time complexity, Asymptotic Notations- Big oh notation, Omega notation, Theta notation and Little oh notation.

Recursion: Introduction, Fibonacci sequence, Climbing Stairs, Reverse String, Happy Number, Greatest Common Divisor, Strobogrammatic Number II.

Divide and Conquer: General method, Quick sort, Merge sort, Applications: Majority Element, Calculate pow(x,n).

What is an algorithm?

An algorithm is a procedure used for solving a problem or performing a computation. Algorithms act as an exact list of instructions that conduct specified actions step by step in either hardware- or software-based routines.

Algorithms are widely used throughout all areas of IT. In mathematics and computer science, an algorithm usually refers to a small procedure that solves a recurrent problem. Algorithms are also used as specifications for performing data processing and play a major role in automated systems.

An algorithm could be used for sorting sets of numbers or for more complicated tasks, like recommending user content on social media. Algorithms typically start with initial input and instructions that describe a specific computation. When the computation is executed, the process produces an output.

Qualities of a Good Algorithm

Input: Zero or more quantities are externally supplied.

Output: At least one quantity is produced.

Definiteness: Each instruction is clear and unambiguous of an algorithm.

Finiteness: If we trace out the instructions of an algorithm for all cases, then the algorithm should terminate after a finite number of steps.

Effectiveness: Instruction is basic enough to be carried out.

How do algorithms work?

Algorithms can be expressed as natural languages, programming languages, pseudocode, flowcharts and control tables. Natural language expressions are rare, as they are more ambiguous. Programming languages are normally used for expressing algorithms executed by a computer.

Algorithms use an initial input along with a set of instructions. The input is the initial data needed to make decisions and can be represented in the form of numbers or words. The input data gets put through a set of instructions, or computations, which can include arithmetic and decision-making processes. The output is the last step in an algorithm and is normally expressed as more data.

What are different types of algorithms?

There are several types of algorithms, all designed to accomplish different tasks. For example, algorithms perform the following:

- **Search engine algorithm.** This algorithm takes strings of keywords and operators as input, searches its associated database for relevant webpages and returns results.
- **Encryption algorithm.** This computing algorithm transforms data according to specified actions to protect it. A symmetric key algorithm, such as the Data_Encryption_Standard, for example, uses the same key to encrypt and decrypt data. As long as the algorithm is sufficiently sophisticated, no one lacking the key can decrypt the data.
- **Greedy algorithm.** This algorithm solves optimization problems by finding the locally optimal solution, hoping it is the optimal solution at the global level. However, it does not guarantee the most optimal solution.
- **Recursive algorithm.** This algorithm calls itself repeatedly until it solves a problem. Recursive algorithms call themselves with a smaller value every time a recursive function is invoked.
- **Backtracking algorithm.** This algorithm finds a solution to a given problem in incremental approaches and solves it one piece at a time.
- **Divide-and-conquer algorithm.** This common algorithm is divided into two parts. One part divides a problem into smaller sub problems. The second part solves these problems and then combines them together to produce a solution.
- **Dynamic programming algorithm.** This algorithm solves problems by dividing them into subproblems. The results are then stored to be applied for future corresponding problems.

- **Brute-force algorithm.** This algorithm iterates all possible solutions to a problem blindly, searching for one or more solutions to a function.
- **Sorting algorithm.** Sorting algorithms are used to rearrange data structure based on a comparison operator, which is used to decide a new order for data.
- **Hashing algorithm.** This algorithm takes data and converts it into a uniform message with a hashing
- **Randomized algorithm.** This algorithm reduces running times and time-based complexities. It uses random elements as part of its logic.

Pseudo code for expressing algorithms

We present the algorithms using pseudo code that looks like C and Pascal code.

1. Comments begin with // and continue until end of the line.
2. Blocks are indicated with braces: **{ }.**
   i. A compound statement.
   ii. Body of a function.
3. i). The data types of variables are not explicitly declared.
   ii). The types will be clear from the context.
   iii). Whether a variable is global or local to a function will also be clear from the context.
   iv). We assume simple data types such as integer, float, char, boolean, and so on.
   v). Compound data types can be formed with **records**.

```
node = record
        {
                datatype_1    data_1;
                        :
                datatype_n    data_n;
                node          *link
        }
```

data items of a record can be accessed with and period(.)

4. Assignment statement.

   < variable > **:=** < expression >

5. Boolean values are **true** and **false.** Logical operators are **and**, **or** and **not** and the relational operators are **<, ≤, =, ≠, ≥** and >.
6. Elements of arrays are accessed using [and]. For example the (i,j)th element of the array A is denoted as A[i,j].
7. The following looping statements are used: for, while, and repeat until.

The general form of a **while** loop:

```
while( condition ) do
{
        statement_1;
                :
```

```
        statement_n;
    }
```

The general form of a **for** loop:

```
    for variable := value1 to value2 step step do
    {
        statement_1;
            :
        statement_n;
    }
```

- Here value1, value2, and step are arithmetic expressions.
- The clause "**step** *step"* is optional and taken as +1 if it does not occur.
- *step* could be either positive or negative.

Ex: 1: for i:= 1 to 10 step 2 do // increment by 2, 5 iterations
2: for i:= 1 to 10 do // increment by 1, 10 iterations

while loop as follows:

```
variable:=value1;
incr:=step;
while( ( variable – value2)*step ≤ 0 ) do
{
    <statement 1>
        :
      <statement n>
    variable :=variable+incr;
}
```

- The general form of a **repeat-until** loop:

```
repeat
        <statement 1>
            :
        <statement n>
    until ( condition )
```

- The statements are executed as long as condition is false.

Ex:

```
number:=10 ; sum:=0;
repeat
        sum := sum + number;
        number := number - 1;
until   number = 0;
```

8. A conditional statement has the following

forms:

if< *condition* > **then** < *statement* >

if< *condition* > **then** < *statement* 1> else

< *statement* 2>

9. Input and output are done using the instructions **read** and **write.**

10. Procedure or function starts with the word **Algorithm.**

General form :

Algorithm *Name*(<parameter list>)

{

body

}

where *Name* is the name of the procedure.

- Simple variables to functions are passed by value.
- Arrays and records(structure or object) are passed by reference.

Ex-1: Algorithm that finds and returns the maximum of n given numbers.

```
Algorithm max(a,n)
// a is an array of size n
{
    Result:=a[1];
    for i:=2 to n do
       if a[i]>Result  then Result:=a[i];

  return Result;
}
```

Ex-2:-Write an algorithm to sort an array of n integers using bubble sort.

```
Algorithm sort(a,n)
// a is an array of size n
{
        for i:=1 to n-1 do
        {
            for j:=1 to n-1-i do
            {
                if( a[j] >a[j+1] ) then
                    t=:a[j];
                    a[j]:=a[j+1];
                    a[j]:=t;
            }
```

```
        }
}
```

Performance Analysis:

- There are many things upon which the performance will depend.
  - Does the program efficiently use primary and Secondary storage?
  - Is the program's running Time acceptable for the task?
  - Does it do what we want it to do?
  - Does it work correctly according to the specifications of the task?
  - Does the program contain documentation that shows how to use it and how it works?
  - Is the program's code readable?

I. Space Complexity:

- The space required by an algorithm is called a space complexity
- The space required by an algorithm has the following components

1. Instruction space. 2. Data space 3. Environmental stack space

1. Instruction space:

- Instruction space is the space needed to store the compiled version of the program instructions.
- The amount of instruction space that is needed depends on the compiler used to compile the program into machine code.

2. Data space:

Data space is the space needed to store all constant and variable values.

Data space has two components.

i. Space needed by constants
(ex; 1 and 2 in max of n num algorithm) and simple variables (such as i, j, n etc).

ii. Space needed by a dynamically allocated objects such as arrays and class instances.

- Space taken by the variables and constants varies from language to language and platform to platform.

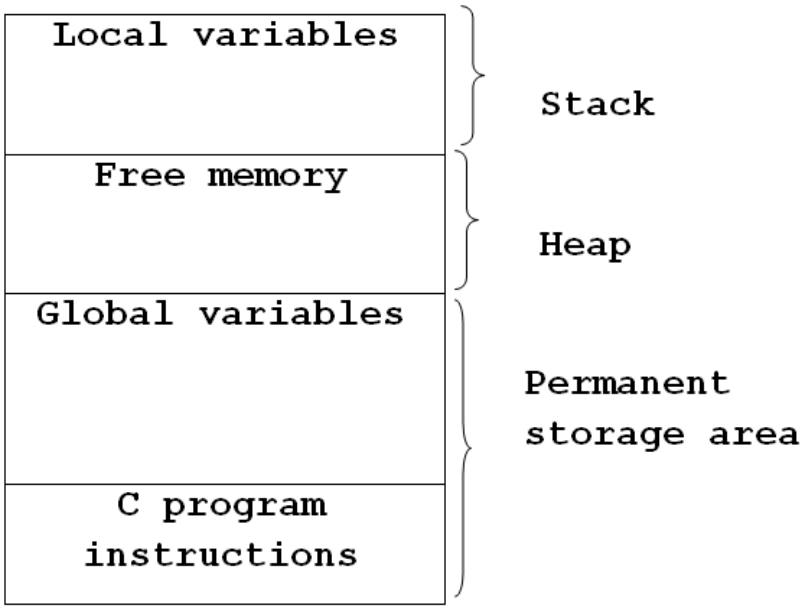
3. Environmental stack space

- The environmental stack is used to save information needed to resume execution of partially completed functions.
- Each time a function is invoked the following data are saved on the environment stack.

i. The return address.
ii. The values of all local variables and formal parameters in the function being invoked (necessary for recursive functions only).

Memory allocation process

- The following figure shows the storage of a program in memory.



Ex: Recursive algorithm

Algorithm rfactorial(n)

```
// n is an integer
 {
        fact=1;
        if(n=1 or n=0) return fact;
        else
                fact=n*rfactorial(n-1);
        return fact;
 }
```

Each time the recursive function rfactorial is invoked, the current values of n and fact and the program location to return to on completion are saved in the environment stack.

Summary of space complexity:

- The space needed by a program depends on several factors.
- We cannot make an accurate analysis of the space requirements of a program unless we know the computer or compiler that will be used.
  - However, we can determine the components that depend on the characteristics of the problem instance

  (e.x., the number of inputs and outputs or magnitude of the numbers) *to be solved.*

  Ex:-1 Space requirements of a program that sorts n elements can be expressed as a function of n.

  Ex:-2 Space requirements of a program that adds two m×n matrices can be expressed as a function of m, n.

- The size of the instruction space is independent of the problem instance to be solved.
- The contribution of the constants and simple variables to the data space is also independent of the problem instance to be solved.
- Most of the dynamically allocated memory (ex., arrays, class instances etc) depends on problem instance to be solved.
- The environmental stack space is generally independent of the problem instance unless recursive functions are in use.

Therefore, we can divide the total space needed by Program into two parts:

i) Fixed Space Requirements (*C*)

Independent of the characteristics of the problem instance (I)

- instruction space
- space for simple variables and constants.

ii) Variable Space Requirements ($S_P(I)$)

depend on the characteristics of the problem instance (I)

Number of inputs and outputs associated with I

recursive stack space (formal parameters, local variables, return address).

Therefore, the space requirement of any problem P can be written as

$S(p)=\mathbf{C}+\mathbf{S_p}$ **(Instance characteristics)**

Note:

- When analyzing the space complexity of an algorithm, we concentrate only on estimating

  S_p (Instance characteristics).
- We do not concentrate on estimating fixed part C.
- We need to identify the instance characteristics of the problem to measure S_P

Ex-1:

```
Algorithm abc(a, b, c)
{
        return a+b+b*c+(a+b-c)/(a+b)+4.0;
}
```

- Problem instance characterized by the specific values of a, b and c.
- If we assume one word (4 bytes) is adequate to store the values of each a, b, and c , then the space needed by abc is independent of the instance characteristics.
  Therefore, S_{abc} (instance characteristics) =0.

Ex-2:

```
Algorithm sum(a,n)
{
        s:=0;
        for i:=1 to n do
              s:=s+a[i];
        return s;
}
```

- Problem instance characterized by n.
- The amount of space needed does not depend on the value of n.
  Therefore, $S_{sum}(n)=0$

Ex-3:

```
Algorithm RSum(a,n)
{
        if (n ≤ 0) then return 0;
        else return RSum(a, n-1)+a[n];
}
```

Type	Name	Number of bytes
formal parameter: int	a	2
formal parameter: int	n	2
return address (Used internally)		2
Total per one recursive call		6

Total no. of recursive calls n, therefore $S_{RSum}(n)=6(n+1)$

Ex-4:

```
int addSequence (int n){
   int sum = 0;
   for (int i = 0; i < n; i++){
      sum += pairSum(i, i+1);
   }
   return sum;
}

int pairSum(int x, int y){
   return x + y;
}
```

There will be roughly O(n) calls to pairSum. However, those calls do not exist simultaneously on the call stack, so you only need O(1) space.

1) Need to store 1 billion peoples age? What would be the space complexity?

2) Need to store 2 lakh employee Ids? What would be the space complexity?

II. Time Complexity:

Time Complexity is a notation/analysis that is used to determine how the number of steps in an algorithm increase with the increase in input size.

$$T(P)=C+T_P(I)$$

- The time, $T(P)$, taken by a program, P, is the sum of its compile time C and its run (or execution) time, $T_P(I)$.
- The compile time does not depend on the instance characteristics.
- We will concentrate on estimating run time $T_p(I)$.
- If we know the characteristics of the compiler to be used, we can determine the No. of additions, subtractions, multiplications, divisions, compares, and so on.

Then we can obtain an expression for $T_p(n)$ Of the form

$$T_P(n)=c_aADD(n)+c_sSUB(n)+c_mMUL(n)+c_dDIV(n)+\ldots\ldots$$

Where,

- n denotes the instance characteristics.
- c_a, c_s, c_m, c_d and so on denote the time needed for an addition, subtraction, multiplication, division and so on.
- ADD, SUB, MUL, DIV, and so on are functions whose values are the no. of additions, subtractions, multiplications, divisions, and so on.

➢ Obtaining such an exact formula is an impossible task, since time needed for an addition, subtraction, and so on, depends on numbers being added, subtracted, and so on.

➢ The value of **Tp(n)** for any given n can be obtained only experimentally.

➢ Even with the experimental approach, one could face difficulties.

➢ In a multiuser system the execution time of a program **p** depends on the number of other programs running on the computer at the time program **p** is running.

➢ As there were some problems in determining the execution time using earlier methods, we will go one step further and count only the number of ***program steps.***

1) Imagine a room of 100 students in which you gave your pen to one person. You have to find that pen without knowing to whom you gave it.
What is the time complexity?

Is it?
$O(n^2)$ or O(n) or O(nlogn)

The $O(n^2)$ searches if only one student knows which student has the pen.
The O(n) if one student had the pen and only, they knew it.
The O(log n) search if all the students knew, but would only tell me if I guessed the right side.

2) Counting frequencies of array elements
Is it?
$O(n^2)$ or O(nlogn) or O(n)

The time complexity to find the frequency of array elements depends on the approach used:

1. Brute Force Method: Compare each element with every other element to count occurrences.
   - Time complexity: O(n^2) where n is the number of elements.
2. Hash Table Method: Use a hash table to store elements as keys and their frequencies as values.
   - Time complexity: O(n) for inserting elements into the hash table and O(n) for iterating through the hash table to get frequencies.
   - Total time complexity: O(n)
3. Sorting Method: Sort the array and then iterate through the sorted array to count consecutive occurrences.
   - Time complexity: O(n log n) for sorting and O(n) for iterating through the sorted array.
   - Total time complexity: O(n log n)

Note: n is the number of elements in the array.

The most efficient approach is the Hash Table Method with a time complexity of O(n).

3) Given an array of elements search for an element
Is it?
O(1) or O(logn) or O(n)

The time complexity to search for an element in an array depends on the search algorithm used. Here are some common search algorithms and their time complexities:

1. Linear Search: O(n)
   - Iterate through each element in the array until a match is found.
2. Binary Search: O(log n)

- Require a sorted array. Divide the search interval in half repeatedly until a match is found.

3. Hash Table Search: O(1) average, O(n) worst-case
   - Store elements in a hash table. Look up the element by its key (index).

Note:
- n is the number of elements in the array.
- log n is the logarithm of n, usually base 2.

If the array is:
- Small, linear search might be sufficient.
- Large and sorted, binary search is more efficient.
- Large and unsorted, consider using a hash table or sorting the array first.

Keep in mind that these complexities assume a single search operation. If you need to search for multiple elements, the overall time complexity may change.

Program step

-Program step is *loosely* defined as a syntactically or semantically meaningful program *segment* whose execution time is independent of the *instance characteristics.*

Example:

```
result = a + b + b * c + (a + b - c) / (a + b) + 4.0;
sum = a + b + c;
```

- The number of steps assigned to any ***program statement*** depends on the kind of statement.
- **Comments** are counted as **zero** number of steps.
- An **assignment** statement which does not involve any calls to other functions counted as **one** step.
- For loops, such as the **for**, **while**, and **repeat-until**, we consider the step counts only for the **control part** of the statement.
- The control parts for **for** and **while** statements have the following forms:
  - for i:= <expr1> to <expr2> do
  - while (<expr>) do
- Each execution of the control part of a while statement is one, unless <expr> is a function of instance characteristics.
- The step count for each execution of the control part of a for statement is one, unless <expr1> and <expr2> are functions of the instance characteristics.

- The step count for each execution of the condition of conditional statements is one, unless condition is a function of instance characteristics.
- If any statement (assignment statement, control part, condition etc.) involves function calls, then the step count is equal to the number of steps assignable to the function plus one.

Best, Worst, Average Cases

- ✓ Not all inputs of a given size take the same number of program steps.
- ✓ Sequential search for *K* in an array of *n* integers:
  - Begin at first element in array and look at each element in turn until *K* is found.

1. **Best-Case Step count:-** Minimum number of steps executed by the algorithm for the given parameters.
2. **Worst-Case Step count:-** Maximum number of steps executed by the algorithm for the given parameters.
3. **Average-Case Step count:-** Average number of steps executed by an algorithm.

Asymptotic Notations:

Asymptotic notation describes the behavior of functions for the large inputs.

1. Big Oh(*O*) notation:

The big oh notation describes an upper bound on the asymptotic growth rate of the function f.

Big-O notation represents the upper bound of the running time of an algorithm. Therefore, it gives the worst-case complexity of an algorithm.

It is the most widely used notation for Asymptotic analysis.

It specifies the upper bound of a function.

The maximum time required by an algorithm or the worst-case time complexity.

It returns the highest possible output value(big-O) for a given input.

Big-Oh(Worst Case) It is defined as the condition that allows an algorithm to complete statement execution in the longest amount of time possible.

2. *Omega (∧)* notation:

The omega notation describes a lower bound on the asymptotic growth rate of the function f.

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

The execution time serves as a lower bound on the algorithm's time complexity.

It is defined as the condition that allows an algorithm to complete statement execution in the shortest amount of time.

3. *Theta (∪)* notation:

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.

Theta (Average Case)
You add the running times for each possible input combination and take the average in the average case.

4. Little Oh(*O)* notation:

Little o notation is used to describe an upper bound that cannot be tight. In other words, loose upper bound of f(n).

Constant Time Algorithms – O(1)

```
int n = 1000;
System.out.println("Output: " + n);
```

Clearly, it doesn't matter what n is. It takes a constant amount of time to run. It's not dependent on the size of n.

```
int n = 1000;
System.out.println("Output: " + n);
System.out.println("Next output: " + n);
System.out.println("One more output " + n);
```

The above example is also constant time.
Even if it takes 3 times as long to run, it doesn't depend on the size of the input, n. We denote constant time algorithms as follows: O(1). Note that O(2), O(3) or even O(1000) would mean the same.

We don't care about exactly how long it takes to run, only that it takes constant time.

Logarithmic Time Algorithms – O(log n)

Constant time algorithms are the quickest. Logarithmic time is the next quickest.

One common example of a logarithmic time algorithm is the binary search algorithm.

What is important here is that the running time grows in proportion to the logarithm of the input (in this case, log to the base 2):

```
for (int i = 1; i < n; i = i * 2){
    System.out.println("Output: " + i);
}
```

If n is 8, the output will be the following:

Output: 1
Output: 2
Output: 4

Linear Time Algorithms – O(n)

After logarithmic time algorithms, we get the next fastest: linear time algorithms.

It grows directly proportional to the size of its inputs.

```
for (int i = 0; i < n; i++) {
    System.out.println("Output: " + i);
}
```

N Log N Time Algorithms – O(n log n)

The running time grows in proportion to n log n of the input:

```
for (int i = 1; i < n; i++){
    for(int j = 1; j < n; j = j * 2) {
        System.out.println("Output: " + i + " and " + j);
    }
}
```

If n is 8, then this algorithm will run 8 * log(8) = 8 * 3 = 24 times.

Polynomial Time Algorithms – $O(n^p)$

These algorithms are even slower than n log n algorithms.

The term polynomial is a general term which contains quadratic (n^2), cubic (n^3), quartic (n^4), etc. functions.

What's important to know is that $O(n^2)$ is faster than $O(n^3)$ which is faster than $O(n^4)$, etc.

```
for (int i = 1; i <= n; i++) {
    for(int j = 1; j <= n; j++) {
        System.out.println("Output: " + i + " and " + j);
    }
}
```

This algorithm will run $8^2 = 64$ times. Note, if we were to nest another for loop, this would become an $O(n^3)$ algorithm.

Exponential Time Algorithms – ***O(k^n)***

These algorithms grow in proportion to some factor exponentiated by the input size.

For example, $O(2^n)$ **algorithms double with every additional input.** So, if $n = 2$, these algorithms will run four times; if $n = 3$, they will run eight times (kind of like the opposite of logarithmic time algorithms).

$O(3^n)$ **algorithms triple with every additional input,** $O(k^n)$ **algorithms will get k times bigger with every additional input.**

```
for (int i = 1; i <= Math.pow(2, n); i++){
    System.out.println("Output: " + i);
}
```

If n is 8 it will run 2^8 = 256 times.

Factorial Time Algorithms – ***O(n!)***

```
for (int i = 1; i <= factorial(n); i++){
    System.out.println("Output: " + i);
}
```

where factorial(n) simply calculates n!. If n is 8, this algorithm will run 8! = 40320 times.

Functions ordered by growth rate:

Function	Name
1	Growth is constant

logn	**Growth is logarithmic**
n	**Growth is linear**
nlogn	**Growth is n-log-n**
n^2	**Growth is quadratic**
n^3	**Growth is cubic**
2^n	**Growth is exponential**
n!	**Growth is factorial**

- **$< \log n < n < n\log n < n^2 < n^3 < 2^n < n!$**

- **To get a feel for how the various functions grow with n, you are advised to study the following figs:**

Instance characteristic n							
Time	Name	1	2	4	8	16	32
1	Constant	1	1	1	1	1	1
$\log n$	Logarithmic	0	1	2	3	4	5
n	Linear	1	2	4	8	16	32
$n \log n$	Log linear	0	2	8	24	64	160
n^2	Quadratic	1	4	16	64	256	1024
n^3	Cubic	1	8	64	512	4096	32768
2^n	Exponential	2	4	16	256	65536	4294967296
$n!$	Factorial	1	2	24	40326	20922789888000	26313×10^{33}

Recursion:

Introduction:

- Recursion is a powerful algorithmic technique in which a function calls itself (either directly or indirectly) on a smaller problem of the same type in order to simplify the problem to a solvable state.
- Every recursive function must have at least two cases: the recursive case and the base case.
- The base case is a small problem that we know how to solve and is the case that causes the recursion to end.
- The recursive case is the more general case of the problem we're trying to solve.

As an example, with the factorial function $n!$, the recursive case is $n! = n*(n - 1)!$ and the base case is $n = 1$ when $n = = 0$ or $n = = 1$.

- Recursive techniques can often present simple and elegant solutions to problems.
- However, they are not always the most efficient. Recursive functions often use a good deal of memory and stack space during their operation.
- The stack space is the memory set aside for a program to use to keep track of all of the functions and their local states currently in the middle of execution.
- Because they are easy to implement but relatively inefficient, recursive solutions are often best used in cases where development time is a significant concern.
- There are many different kinds of recursion, such as linear, tail, binary, nested, and mutual. All of these will be examined.

What is recursion?

- Sometimes a problem is too difficult or too complex to solve because it is too big.
- If the problem can be broken down into smaller versions of itself, we may be able to find a way to solve one of these smaller versions and then be able to build up to a solution to the entire problem.
- This is the idea behind recursion; recursive algorithms break down a problem into smaller pieces which you either already know the answer to, or can solve by applying the same algorithm to each piece, and then combining the results.
- Stated more concisely, a recursive definition is defined in terms of itself.
- Recursion is a computer programming technique involving the use of a procedure, subroutine, function, or algorithm that calls itself in a step having a termination condition so that successive repetitions are processed up to the critical step where the condition is

met at which time the rest of each repetition is processed from the last one called to the first.

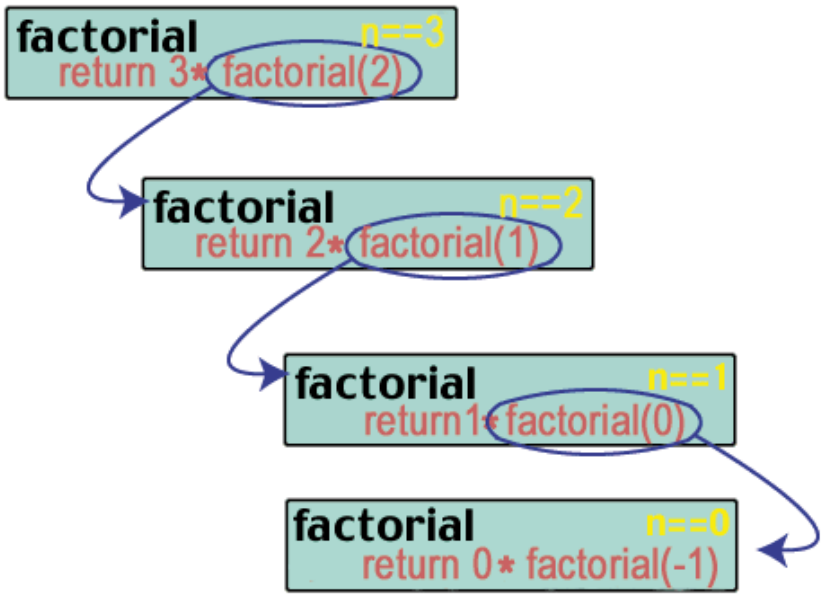
Example:

let's try writing our factorial function **int factorial(int n).**

We want to code in the $n! = n*(n - 1)!$ functionality.

```
int factorial(int n)
{
        // Handling base case
         if (n == 0 || n == 1)
                return 1;
        return n * factorial(n-1);
}
```

Wasn't that easy? Lets test it to make sure it works. We call factorial on a value of 3, factorial(3). Figure shows the scenario when we don't have the base condition.



Applications:

1. **Fibonacci Series**
2. **Climbing Stairs**
3. **Reverse String**
4. **Happy Number**

5. **GCD(Greatest common Divisor)**
6. **Strobogrammatic number II:**

1. Fibonacci Series:

- A **Fibonacci Series** in Java is a series of numbers in which the next number is the sum of the previous two numbers.
- The first two numbers of the Fibonacci series are 0 and 1. The Fibonacci numbers are significantly used in the computational run-time study of an algorithm to determine the greatest common divisor of two integers.
- In arithmetic, the Wythoff array is an infinite matrix of numbers resulting from the Fibonacci sequence.

The Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Fibonacci Series Using Recursion in Java : FibonacciCalc.java

```
public class FibonacciCalc
{
	public static int fibonacciRecursion(int n)
	{
		if(n == 0)
		{
			return 0;
		}
		if(n == 1 || n == 2)
		{
			return 1;
		}
		return fibonacciRecursion(n-2) + fibonacciRecursion(n-1);
	}
	public static void main(String args[])
	{
		int maxNumber = 10;
		System.out.print("Fibonacci Series of "+maxNumber+" numbers: ");
		for(int i = 0; i < maxNumber; i++)
		{
			System.out.print(fibonacciRecursion(i) +" ");
		}
```

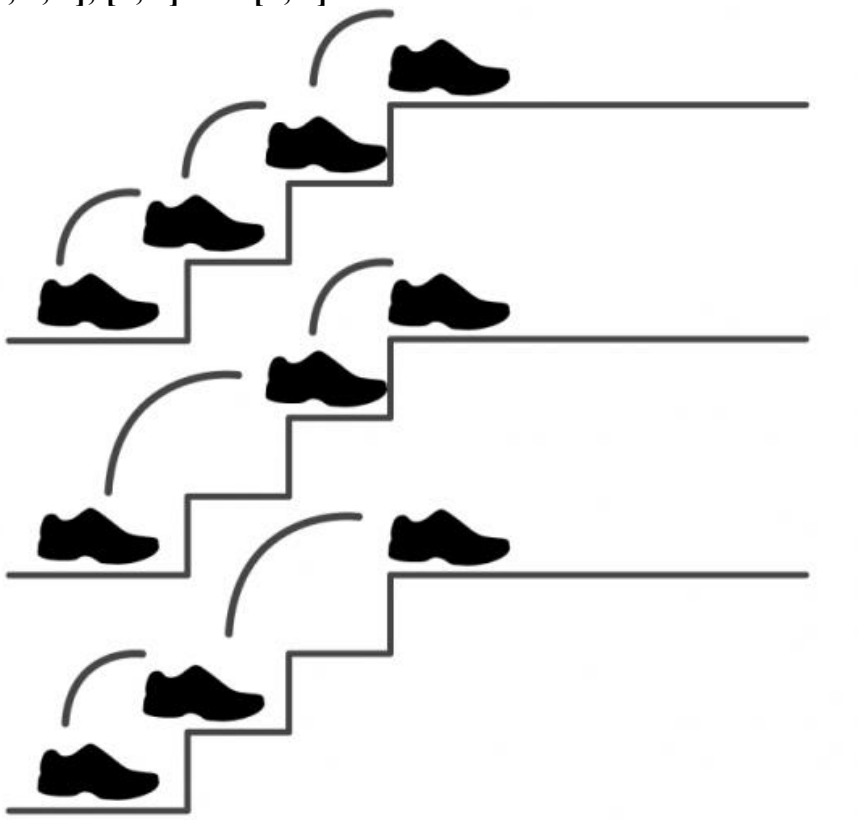
```
        }
}
```

2. Climbing Stairs:

Given a staircase of **N** steps and you can either climb 1 or 2 steps at a given time. The task is to return the count of distinct ways to climb to the top.
Note: The order of the steps taken matters.
Examples:
Input: N = 3

Output: 3

Explanation:
There are three distinct ways of climbing a staircase of 3 steps :
[1, 1, 1], [2, 1] and [1, 2].



Input: N = 2
Output: 2
Explanation:
There are two distinct ways of climbing a staircase of 2 steps :
[1, 1] and [2].

Input: n = 4

Output: 5
(1, 1, 1, 1), (1, 1, 2), (2, 1, 1), (1, 2, 1), (2, 2)

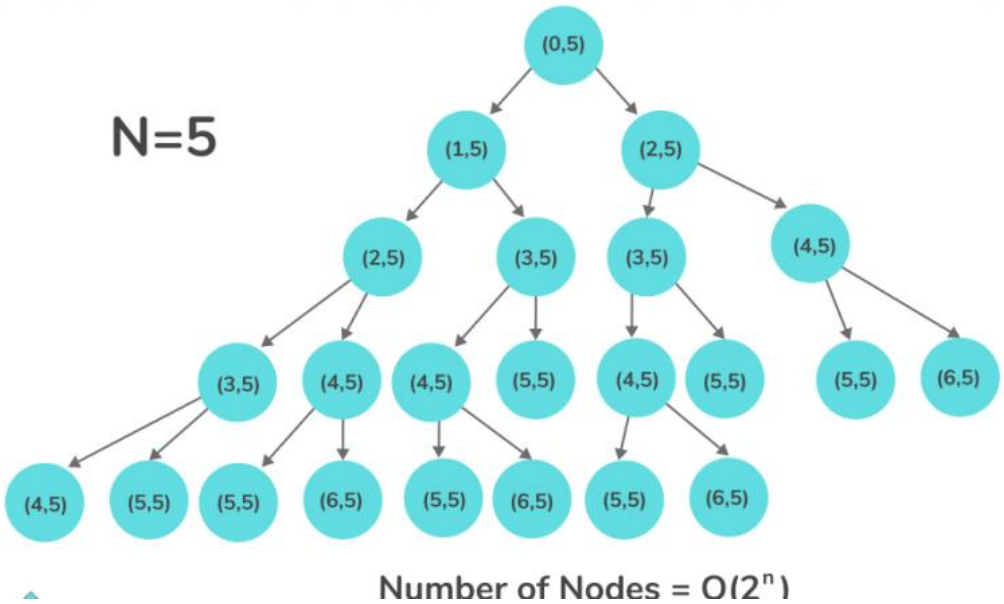
Method-1: Brute Force (Recursive) Approach

The approach is to consider all possible combination steps i.e. 1 and 2, at every step. To reach the **Nth** stair, one can jump from either **(N – 1)th** or from **(N – 2)th** stair. Hence, for each step, total ways would be the summation of **(N – 1)th stair + (N – 2)**th stair.

The recursive function would be:

ClimbStairs(N) = ClimbStairs(N – 1) + ClimbStairs(N – 2).

If we observe carefully, the expression is nothing but the **Fibonacci Sequence**.
SampleRecursive Tree for **N = 5**:



Algorithm:

- If **N < 2,** return **1**. This is the termination condition for the function.
- Else, find the summation of **ClimbStairs(N – 1) + ClimbStairs(N – 2)**.

Java program for method-1: Stairs_M1.java

```
import java.util.*;

class Stairs_M1 {
	static int ClimbStairs(int n)
	{
		if (n <= 2)
			return n;
		return  ClimbStairs(n-1) + ClimbStairs(n-2);
	}

	// Returns number of ways to reach s'th stair
	static int countWays(int s)
	{
		return ClimbStairs(s);
	}

	public static void main(String args[])
	{
		Scanner sc =new Scanner(System.in);
		System.out.println("enter no.of stairs");
		int s = sc.nextInt();
		System.out.println("Number of ways = " + countWays(s));
	}
}
```

Generalization of the Problem:

How to count the number of ways if the person can climb up to m stairs for a given value m. For example, if m is 4, **the person can climb 1 stair or 2 stairs or 3 stairs or 4 stairs at a time**.

Approach: For the generalization of above approach the following recursive relation can be used.
ways (n, m) = ways(n-1, m) + ways(n-2, m) + ... ways(n-m, m)

in this approach to reach **n^{th} stair**, try climbing all **possible number of stairs lesser than equal to n** from present stair.

Java program for generalization method: Stairs_M2.java

```
import java.util.*;

class Stairs_M2
{
        // Recursive function to count ways to reach nth stair
    public static int countWays(int n, int m)
        {
        // Base case: exactly at stair 0
        if (n == 0) {
            return 1;
        }
        // Base case: below stair 0
        if (n < 0) {
            return 0;
        }

        int totalWays = 0;
        // Try every possible jump from 1 to m
        for (int i = 1; i <= m; i++) {
            totalWays += countWays(n - i, m);
        }
        return totalWays;
    }

        public static void main(String args[])
        {
                Scanner sc =new Scanner(System.in);
                System.out.println("enter total no.of stairs");
                int s = sc.nextInt();
                System.out.println("enter possible stairs ");
                int m = sc.nextInt();
                System.out.println("Number of ways = "   + countWays(s, m));
        }
```

}

3. Reverse String

The recursive function performs the following steps to reverse a string:

Input a line of text.

Pass each word to reverseString function to reverse it.

In reverseString function,

Swap left most and right most characters.

Increment left index and decrement right index.

Repeat swapping by recursive call till the left>=right

```
import java.util.*;

// Using Recursion
class ReverseString
{
	public void reverseString(char[] s, int left, int right) {
		if (left >= right) return;
		char tmp = s[left];
		s[left++] = s[right];
		s[right--] = tmp;
		reverseString(s, left, right);
	}

	// hello world genesis kmit ngit
	public static void main(String args[])
	{
		Scanner sc=new Scanner(System.in);
		String[] s=sc.nextLine().split(" ");
		for(int i = 0; i < s.length; i++)
		{
			char [] data = s[i].toCharArray();
			new ReverseString().reverseString(data, 0, data.length - 1);
			System.out.println(data);
```

```
            }
        }
}
```

4. Happy number:

- A happy number is a number which eventually reaches 1 when replaced by the sum of the square of each digit.
- Whereas if during this process any number gets repeated, the cycle will run infinitely and such numbers are called unhappy numbers.

Example-1: 13 is a happy number because,

1^2 + 3^2 = 10 and,

1^2 + 0^2 = 1

On the other hand, 36 is an unhappy number.

Example-2:

28=2^2+8^2=4+64=68
68=6^2+8^2=36+64=100
100=1^2+0^2+0^2=1 +0+0=1
Hence,28 is a happy number.

Example-3 :

12 =1^2+2^2=1+4=5
Hence, 12 is not a happy number.

Example-4 : 7

$7^2 = 49$
$4^2 + 9^2 = 97$
$9^2 + 7^2 = 130$
$1^2 + 3^2 + 0^2 = 10$
$1^2 + 0^2 = 1$

Java program for Happy number using recursion: Happynumber.java

```
import java.util.Scanner;
 class Happynumber
```

```
{
 public static void main(String[] args)
 {
  Scanner in = new Scanner(System.in);
   System.out.println("Enter a number");
  int num = in.nextInt();
   if(isHappy(num))
    System.out.println("Happy Number");
  else
    System.out.println("Not a Happy number");
 }

 private static boolean isHappy(int num){
  if(num == 1)
    return true;
  if(num == 4)
    return false;
   //recall the function with sum value
  return isHappy(sumOfDigits(num));
 }

 //Function to return sum of square of digits
 private static int sumOfDigits(int num){
  int sum = 0;
  while(num>0){
    sum += Math.pow(num%10, 2);
    num = num/10;
  }
return sum;
 }
}
```

5. GCD (Greatest Common Divisor):

- The GCD of two numbers A and B is the largest positive common divisor that divide both the integers (A and B).

  For example – Let's take two numbers **63** and **21.**

  General Procedure:

- Factors of 63 – **3, 7, 9, 21 and 63**
- Factors of 21 – **3, 7, 21**

- The common divisiors of both the numbers are **3, 7, 21.** Out of which the greatest common divisior is **21.**
- So the GCD (63,21) is **21.**

Euclidean Algorithm:

Pseudo Code of the Algorithm-

Step 1: **Let a, b be the two numbers**

Step 2: **a mod b = R**

Step 3: **Let a = b and b = R**

Step 4: **Repeat Steps 2 and 3 until a mod b is greater than 0**

Step 5: **GCD = b**

Step 6: Finish

Example:

- Use the Euclidean Algorithm to find GCD of more than two numbers. Since, GCD is associative, the following operation is valid- GCD(a,b,c) == GCD(GCD(a,b), c)
- Calculate the GCD of the first two numbers, then find GCD of the result and the next number. Example- GCD(203,91,77) == GCD(GCD(203,91),77) == GCD(7, 77) == 7

java program for Find GCD of Two Numbers using Recursion: GCD.java

```
import java.util.*;

public class GCD
{
    public static int calculateGCD(int a, int b)
     {
       //If both the number are equal
       if (a == b)
       {
          return a;
       }

       /* If a is equal to zero then return b */
       else if (a == 0)
       {
          return b;
```

```
        }
    /* If b is equal to zero then return a */
    else if (b == 0)
    {
        return a;
      }
    else if (a > b)
    {
        //Recursive call
        return calculateGCD(a % b, b);
    } else {
        //Recursive call
        return calculateGCD(a, b % a);
    }
  }
   public static void main(String[] args)
   {
     Scanner in = new Scanner(System.in);
     System.out.println("Enter first number");
     int a= in.nextInt();
     System.out.println("Enter second number");
     int b= in.nextInt();
     System.out.println(calculateGCD(a, b));

  }
}
```

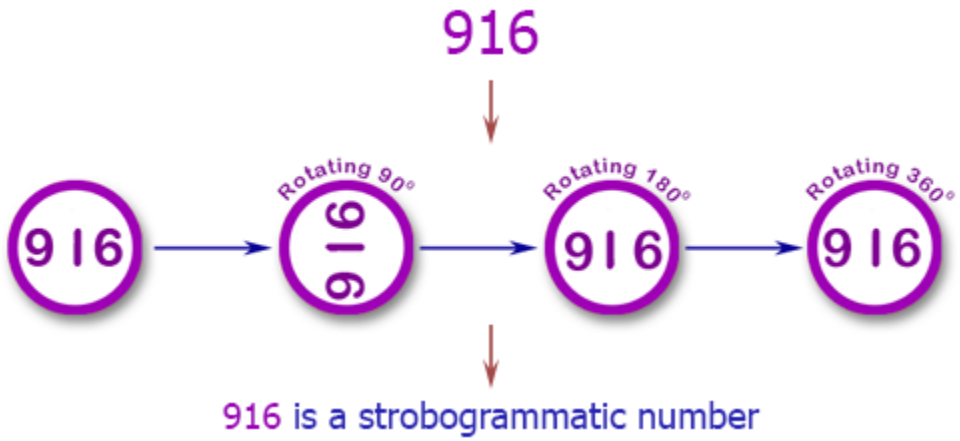
6. Strobogrammatic number II:

- A strobogrammatic number is a number whose numeral is rotationally symmetric, so that it appears the same when rotated 180 degrees.
- In other words, the numeral looks the same right-side up and upside down (e.g., 69, 96, 1001).
- A strobogrammatic prime is a strobogrammatic number that is also a prime number, i.e., a number that is only divisible by one and itself (e.g., 11).
- It is a type of ambigram, words and numbers that retain their meaning when viewed from a different perspective, such as palindromes."

The first few strobogrammatic numbers are :
0, 1, 8, 11, 69, 88, 96, 101, 111, 181, 609, 619, 689, 808, 818, 888, 906, 916, 986, 1001,

1111, 1691, 1881, 1961, 6009, 6119, 6699, 6889, 6969, 8008, 8118, 8698, 8888, 8968, 9006, 9116, 9696, 9886, 9966, ...

Pictorial Presentation:



Algorithm

Let us first enumerate the cases where n is 0,1,2,3,4:

```
n = 0: none
n = 1: 0, 1, 8
n = 2: 11, 69, 88, 96
n = 3: 101, 111, 181, 609, 619, 689, 808, 818, 888, 906, 916, 986
n = 4: 1001, 1111, 1691, 1881, 1961, 6009, 6119, 6699, 6889, 6969, 8008, 8118, 8698,
8888, 8968, 9006, 9116, 9696, 9886, 9966
```

Observations:

Look at n=0 and n=2, you can find that the latter is based on the former, and the left and right sides of each number are increased by [1 1], [6 9], [8 8], [9 6]

Look at n=1 and n=3, it's more obvious, increase [1 1] around 0 become 101, increase around 0 [6 9] becomes 609, increase around 0 [8 8] becomes 808, increases [9 6] to the left and right of 0 becomes 906, and then adds the four sets of numbers to the left and right sides of 1 and 8 respectively

In fact, it starts from the m=0 layer and adds layer by layer.

[0 0] cannot be added to the left and right sides, because 0 cannot appear at the beginning of two or more digits. In the process of recursive in the middle, it is necessary to add 0 to the left and right sides of the number.

Java program for Strobogrammatic Number IIusing recursion:

Strobogrammatic_Number_II.java

```
import java.util.*;

class Solution
{
	static char[][] digitPair = { {'1', '1'}, {'8', '8'}, {'6', '9'}, {'9', '6'} };
	public static List<String> findStrobogrammatic(int n) {
		return helper(n, n);
	}

	static List<String> helper(int n, int m) {
		if (n == 0) return new ArrayList<String>(Arrays.asList(""));
		if (n == 1) return new ArrayList<String>(Arrays.asList("0", "1", "8"));

		List<String> list = helper(n - 2, m);

		List<String> res = new ArrayList<String>();

		System.out.println("list.size " + list.size());
		for (int i = 0; i < list.size(); i++)
		{
			String str = list.get(i);
			System.out.println("n " + n + " m " + m + " str " + str);

			if (n != m) res.add("0" + str + "0");

			for (char[] aDigitPair : digitPair)
			{
				res.add(aDigitPair[0] + str + aDigitPair[1]);
			}
		}
		return res;
	}
	public static void main(String args[])
	{
```

```
        Scanner sc=new Scanner(System.in);
        int N=sc.nextInt();
        List<String> list=findStrobogrammatic(N);
        Collections.sort(list);
        System.out.println(list);
    }
}
```

Divide and conquer:

General method:

- D&C Technique splits n inputs into k subsets , $1< k \leq n$, generating k sub problems.
- These sub problems will be solved and then combined by using a separate method to get solution to the whole problem.
- If the sub problems are large, then the D&C Technique will be reapplied.
- Often sub problems getting from the D&C Technique are of the same type as the original problem.
- The reapplication of the D&C Technique is naturally expressed by a recursive algorithm.
- Now smaller and smaller problems of the same kind are generated until subproblems that are small enough to solve without splitting further.

Control Abstraction / General Method for Divide and Conquer Technique:

```
Algorithm DAndC(p)
{
    if Small(p) then return s(p);
    else
    {
        Divide p into smaller problems p1,p2,…….,pk, k≥1;
        Apply D&C to each of these subproblems;
        return Combine(DAndC(p1), DAndC(p2),……,DAndC(pk));
    }
}
```

- If the size of p is n and the sizes of the k subproblems are $n_1,n_2,….,n_k$, then the computing time of D&C is described by the recurrence relation

$$T(n)=\begin{cases} g(n) & n \text{ small} \\ T(n_1)+T(n_2)+\ldots\ldots+T(n_k)+f(n) & \text{otherwise} \end{cases}$$

- Where T(n) is the time for D&C on any input of size n and g(n) is the time to compute the answer directly for small inputs.
- The function f(n) is the time for dividing p and combining the solutions of subproblems.
- The Time Complexity of many D&C algorithms is given by recurrences of the form

$$T(n)=\begin{cases} n & \text{small} \\ aT(n/b)+f(n) & \text{Otherwise} \end{cases}$$

- Where a , b are known constants, and ***n is a power of b*** *(i.e* $n=b^k$ *)*

Master Theorem

The **Master Theorem** is a tool in computer science that provides a quick way to determine the time complexity of divide-and-conquer algorithms. These algorithms break a problem into smaller subproblems, solve each recursively, and then combine the results. The Master Theorem gives a formula to analyze the overall time complexity based on the recurrence relation that describes the algorithm.

Standard Form of Recurrence Relation

A divide-and-conquer algorithm typically follows a recurrence relation of the form:

$T(n)=aT(n/b)+O(n^d)$

Where:

- T(n) is the time complexity for a problem of size n.
- a is the number of subproblems the problem is divided into.
- n/b is the size of each subproblem (i.e., each subproblem is a fraction of the original problem).
- $O(n^d)$ represents the cost of dividing the problem and combining the results (non-recursive work done outside the recursive calls).

The Master Theorem helps to determine T(n) by comparing how fast the sub problems grow (the number of sub problems and their size) with the cost of dividing and combining the problem. It provides three cases, depending on the relative values of a, b, and d.

The Three Cases of the Master Theorem

1. **Case 1: $a>b^d$**
   - When the number of sub problems grows faster than the cost of combining the results.
   - In this case, the recursion dominates, and the time complexity is determined by the recursive term.

   $T(n)=O(n^{\log_b a})$

   Example: $T(n)=3T(n/2)+O(n)$

   Here, a=3, b=2, and d=1.

   Since $a=3>2^1=2$, this is Case 1, and the complexity is $O(n^{\log 2^3}) \approx O(n1.585)$.

2. **Case 2: $a=b^d$**

   - When the number of subproblems and the cost of combining the results grow at the same rate.
   - In this case, both the recursion and the combining cost contribute equally, and the complexity is dominated by a logarithmic factor.

   $T(n)=O(n^d \log n)$

   Example: $T(n)=2T(n/2)+O(n)$
   Here, a=2, b=2, and d=1. Since $a=b^d$ (i.e., 2 = 2^1), this is Case 2, and the complexity is O(nlogn).

Case 3: $a<b^d$

- When the cost of combining the results grows faster than the number of subproblems.
- In this case, the cost of dividing and combining dominates the time complexity.

T(n)=O(nd)T(n) = O(n^d)T(n)=O(nd)

Example: $T(n)=T(n/2)+O(n^2)$
Here, a=1, b=2, and d=2. Since $a=1<2^2=4$, this is Case 3, and the complexity is $O(n^2)$.

Summary of Master Theorem Cases

1. **Case 1:** If $a>b^d$, then $T(n)=O(n^{\log_b a})$.
2. **Case 2:** If $a=b^d$, then $T(n)=O(n^d \log n)$.
3. **Case 3:** If $a<b^d$, then $T(n)=O(n^d)$.

Applicability of the Master Theorem

- The Master Theorem applies only to recurrences of the form $T(n)=aT(n/b)+O(n^d)$.
- The values of a, b, and d must be constants.
- The size of the subproblem n/b must reduce by a constant factor in each recursive call.

Example Application

- **Merge Sort**: The recurrence is $T(n)=2T(n/2)+O(n)$, which fits into Case 2, yielding a time complexity of O(nlogn).
- **Binary Search**: The recurrence is $T(n)=T(n/2)+O(1)$, which fits into Case 3, yielding a time complexity of O(logn).

Applications:

1. **Quick sort.**
2. **Merge sort.**
3. **Majority Element.**
4. **Calculate pow(x,n).**

1. Quick 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 quick Sort that pick pivot in different ways.
    - ->Always pick first element as pivot.
    - ->Always pick last element as pivot
    - ->Pick a random element as pivot.
    - ->Pick median as pivot.
- The key process in quick Sort 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.

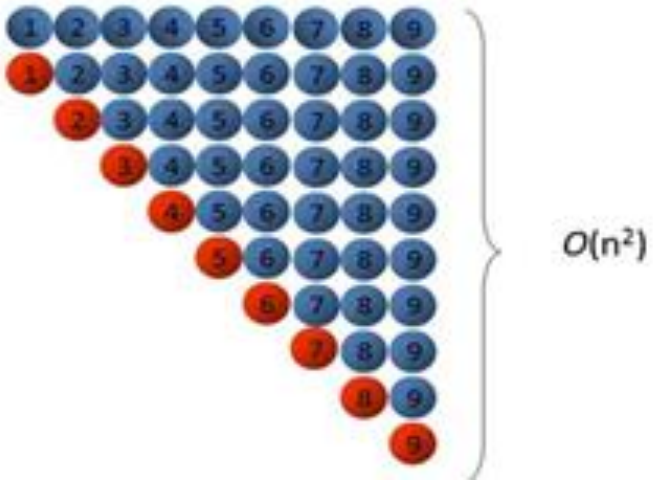
- **Divide**:
    - Pick any element as the pivot, e.g, the last element
    - Partition the remaining elements into
        - FirstPart, which contains all elements < pivot
        - SecondPart, which contains all elements > pivot
- **Recursively sort** First Part and Second Part.
- **Combine**: no work is necessary since sorting is done in place.

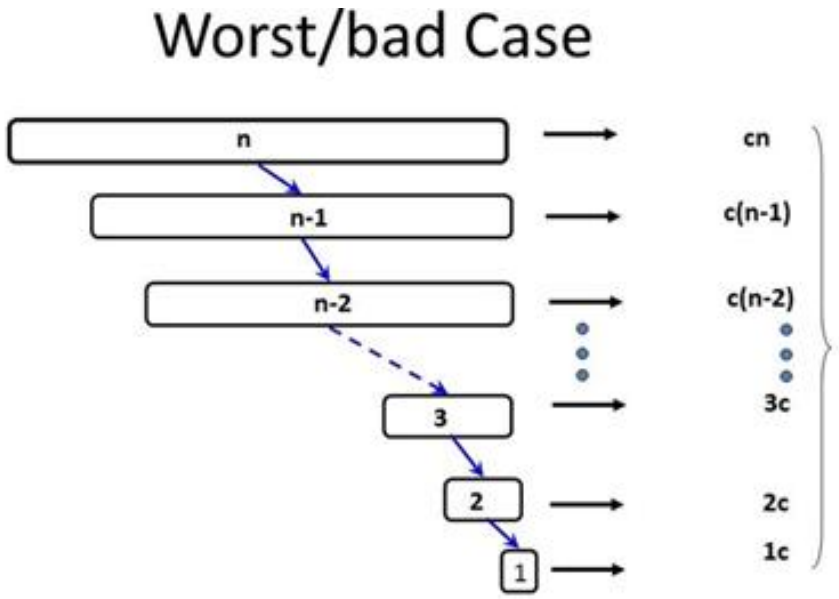
Time complexity analysis:

- The time required to sort n elements using quicksort involves 3 components.
  - Time required for partitioning the array, which is *roughly proportional to n*.
  - Time required for sorting lower subarray.
  - Time required for sorting upper subarray.
- Assume that there are k elements in the lower subarray.
- Therefore,

A worst/bad case

It occurs if the list is already in sorted order





- In the worst case, the array is always partitioned into two subarrays in which one of them is always empty. Thus , for the worst case analysis,

$$T(n) = \begin{cases} T(n-1) + c_2 n & n>1,\ c_2 \text{ is a constant} \end{cases}$$

$$= T(n-1) + c_2 n$$
$$= T(n-2) + c_2 (n-1) + c_2 n$$
$$= T(n-3) + c_2 (n-2) + c_2 (n-1) + c_2 n$$
$$\ldots\ldots$$
$$\ldots\ldots$$
$$= n(n+1)/2 = (n^2+n)/2 = O(n^2)$$

Time complexties:

- Most of the work done in partitioning
- Best case takes ***O*(n log(n))** time
- Average case takes ***O*(n log(n))** time
- Worst case takes ***O*(n^2)** time

Quick sort using recursion in java: Quicksort_Recursive.java

```
import java.util.*;
class QuickSortR
{
	public static void swap (int[] arr, int i, int j)
	{
		if(i != j)
		{
			int temp = arr[i];
			arr[i] = arr[j];
			arr[j] = temp;
		}
	}
```

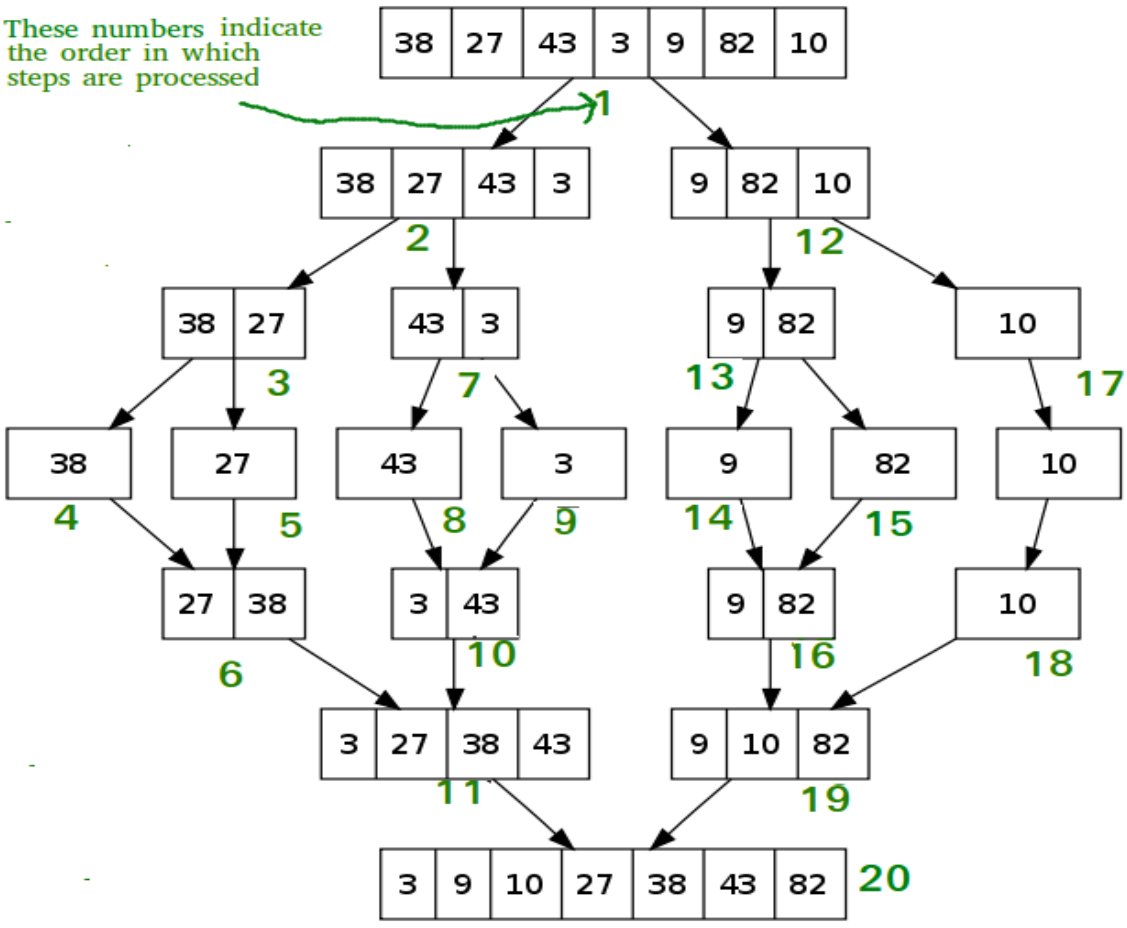
```
// selects last element as pivot, pi using which array is partitioned.
static int partition_high(int arr[], int low, int high)
{
        int pivot = arr[high];

        // smaller element index
        int i = low;
        for (int j = low; j < high; j++)
        {
                // check if current element is lesser than pivot
                if (arr[j] < pivot) {
                        swap(arr, i, j);
                        i++;
                }
        }
        // swap arr[i] and arr[high] (or pivot)
        swap(arr, i, high);
        return i;
}

// selects first element as pivot, pi using which array is partitioned.
static int partition_low(int arr[], int low, int high)
{
        // First element as pivot
         int pivot = arr[low];
        int k = high;

        for (int i = high; i > low; i--) {
                if (arr[i] > pivot)
                {
                        swap(arr, i, k);
                        k--;
                }
        }
        swap(arr, low, k);

        // As we got pivot element index is high, now pivot element is at its sorted
position
        // return pivot element index (high)
        return k;
}
```

```
        // routine to sort the array partitions recursively
        void quick_sort(int intArray[], int low, int high)
        {
                if (low < high)
                {
                        // partition the array around pi=>partitioning index and return pi
                        int pi = partition_low(intArray, low, high);

                        // sort each partition recursively
                        quick_sort(intArray, low, pi-1);
                        quick_sort(intArray, pi+1, high);
                }
        }
        public static void main(String args[])
        {
                // 7
                // 5 2 8 3 1 6 4

                // 6
                // 24 9 29 14 19 27
                Scanner sc=new Scanner(System.in);
                System.out.println("enter array size");

                int n = sc.nextInt();
                int a[]=new int[n];

                System.out.println("enter the elements of array ");
                for(int i=0;i<n;i++)
                {
                        a[i] =sc.nextInt();
                }
                // print the original array
                System.out.println ("Original Array: " + Arrays.toString(a));

                // call quick_sort routine using QuickSortR object
                QuickSortR obj = new QuickSortR();
                obj.quick_sort(a, 0, n-1);

                // print the sorted array
                System.out.println("Sorted Array: " + Arrays.toString(a));
        }
}
```

2. Merge Sort:

- The Merge sort algorithm can be used to sort a collection of objects.
- Merge sort is also called as divide and conquer algorithm**.**
- Base Case, solve the problem directly if it is small enough (only one element).
- Divide the problem into two or more similar and smaller subproblems.
- Recursively solve the subproblems.
- Combine solutions to the subproblems

Merge Sort Example



Pseudo code:

Algorithm MergeSort (low,high)

// sorts the elements a[low],…,a[high] which are in the global array

```
//a[1:n] into ascending order (increasing order).

// Small(p) is true if there is only one element to sort. In this case the list is already
sorted.

{
        if  (low < high) then  //  if there are more than one element
        {
                Mid=(low+high)/2;
                MergeSort(low,mid); // recursion
                MergeSort(mid+1, high);
                Merge(low, mid, high);
        }
}
```

Algorithm Merge(low, mid, high)

```
// a[low:high] is a global array containing two sorted subsets in a[low:mid]
// and in a[mid+1:high]. The goal is to merge these two sets into a single
// set  residing in a  [low:high]. b[ ] is a temporary global array.
{
        h:=low; i:=low; j:=mid+1;
                while( h ≤ mid ) and ( j ≤ high ) do
                {
                        if( a[h] ≤ a[j] ) then
                        {
                                b[i]:=a[h]; h:=h+1;
                        }
                        else
                        {
                                b[i]:=a[j]; j:=j+1;
                        }
                        i:=i+1;
                }

                if( h > mid ) then
                        for k:=j to high do
                        {
                                b[i] := a[k]; i:= i+1;
                        }
                else
                        for k:=h to mid do
```

```
                {
                        b[i] := a[k]; i:= i+1;
                }
            for k:= low  to high do  a[k]:=b[k];
}
```

Merge-Sort Time Complexity

If the time for the merging operation is proportional to n, then the computing time for merge sort is described by the recurrence relation

$$T(n) = \begin{cases} c_1 & n=1, c_1 \text{ is a constant} \\ 2T(n/2) + c_2 n & n>1, c_2 \text{ is a constant} \end{cases}$$

Assume $n=2^k$, then

$$\begin{aligned} T(n) &= 2T(n/2) + c_2 n \\ &= 2(2T(n/4)+c2n/2)+cn \\ &= 4T(n/4)+2c_2 n \\ &\ldots\ldots \\ &\ldots\ldots \\ &= 2^k\, T(1)+ kc_2 n \\ &= c_1 n+c_2 n\log n = = O(n\log n) \end{aligned}$$

Time complexities:

- Most of the work done in combining the solutions.
- Best case takes *o*(n log(n)) time
- Average case takes *o*(n log(n)) time
- Worst case takes *o*(n log(n)) time

Applications for merge sort in real time:

- Merge sort is useful for sorting linked lists
- Inversion count problem
- Used in external sorting- in tape drivers
- e commerce applications

Merge sort using recursion in java:

Mergesort_recursive.java

```
import java.util.Scanner;
import java.util.*;

public class Mergesort_recursive
{
	public static void main(String a[])
	{
		Scanner sc=new Scanner (System.in);
		System.out.println("Enter array size");
		int n = sc.nextInt();

		int [] list=new int[n];
		System.out.println("Enter numbers ");

		for (int i=0 ; i<n; i++)
		{
			int  number = sc.nextInt();
			list[i]=number;
		}
		System.out.println("List before  sorting: " + Arrays.toString(list));
		mergeSort(list,0, list.length-1);
		System.out.println("List after  sorting: " + Arrays.toString(list));
	}


	public static void mergeSort(int list[],int low, int high)
	{
		if (low >= high)
			return;

		int middle = (low + high) / 2;
		mergeSort(list, low, middle);
		mergeSort(list, middle + 1, high);
		merge(list, low,middle,high);
	}

	/* Function to merge the two halfs arr[l..m] and arr[m+1..r] of array arr[] */
	static void merge(int arr[], int l, int m, int r)
	{
		int i, j, k;
		int left = m - l + 1;
		int right = r - m;

		/* create temp arrays */
```

```
        int L[] = new int[left];
        int R[] = new int[right];

        /* Copy data to temp arrays L[] and R[] */
        for (i = 0; i < left; i++)
                L[i] = arr[l + i];
        for (j = 0; j < right; j++)
                R[j] = arr[m + 1+ j];

        /* Merge the temp arrays back into arr[l..r]*/
        i = 0;
        j = 0;
        k = l;
        while (i < left && j < right)
        {
                if (L[i] <= R[j])
                {
                        arr[k] = L[i];
                        i++;
                }
                else
                {
                        arr[k] = R[j];
                        j++;
                }
                k++;
        }

        /* Copy the remaining elements of L[], if there are any */
        while (i < left)
        {
                arr[k] = L[i];
                i++;
                k++;
        }

        /* Copy the remaining elements of R[], if there are any */
        while (j < right)
        {
                arr[k] = R[j];
                j++;
                k++;
        }

    }
}
```

3. Majority Element:

Given an array 'nums' of size n, return the majority element.

The majority element is the element that appears more than ⌊n / 2⌋ times. You may assume that the majority element always exists in the array.

Example 1:
Input: nums = [3,2,3]
Output: 3

Example 2:
Input: nums = [2,2,1,1,1,2,2]
Output: 2

constraints:

- n == nums.length
- $1 <= n <= 5 * 10^4$
- $-10^9 <= nums[i] <= 10^9$

Intuition

If we know the majority element in the left and right halves of an array, we can determine which is the global majority element in linear time.

Algorithm

- Here, we apply a classical divide & conquer approach that recurses on the left and right halves of an array until an answer can be trivially achieved for a length-1 array.
- Note that because actually passing copies of subarrays costs time and space, we instead pass lo and hi indices that describe the relevant slice of the overall array.
- In this case, the majority element for a length-1 slice is trivially its only element, so the recursion stops there. If the current slice is longer than length-1, we must combine the answers for the slice's left and right halves.
- If they agree on the majority element, then the majority element for the overall slice is obviously the same. If they disagree, only one of them can be "right", so we need to count the occurrences of the left and right majority elements to determine which sub slice's answer is globally correct.

- The overall answer for the array is thus the majority element between indices 0 and n.

Complexity Analysis

Time complexity: O(nlogn)

- Each recursive call to majority_element_rec performs two recursive calls on subslices of size n/2 and two linear scans of length n.
- Therefore, the time complexity of the divide & conquer approach can be represented by the following recurrence relation:

$T(n)=2T(n/2)+2n$

By the master theorem, the recurrence satisfies case 2, so the complexity can be analyzed as such:

$T(n) = \Theta(n^{\log_b a} \log n)$

$= \Theta(n^{\log_2 2} \log n)$

$= \Theta(n \log n)$

Space complexity : O(logn)

- Although the divide & conquer does not explicitly allocate any additional memory, it uses a non-constant amount of additional memory in stack frames due to recursion.
- Because the algorithm "cuts" the array in half at each level of recursion, it follows that there can only be O(logn) "cuts" before the base case of 1 is reached.
- It follows from this fact that the resulting recursion tree is balanced, and therefore all paths from the root to a leaf are of length O(logn).
- Because the recursion tree is traversed in a depth-first manner, the space complexity is therefore equivalent to the length of the longest path, which is, of course, O(logn).

Java program for Majority using Recursion: Majority.java

```
import java.util.*;
class MajorityElement
{
        private int countInRange(int[] nums, int num, int lo, int hi)
```

```
{
        int count = 0;
        for (int i = lo; i <= hi; i++)
        {
                if (nums[i] == num)
                {
                        count++;
                }
        }
        return count;
}

private int majorityElementRec(int[] nums, int lo, int hi)
{
        // base case; the only element in an array of size 1 is the majority element.
        if (lo == hi)
        {
                return nums[lo];
        }

        // recurse on left and right halves of this slice.
        int mid = (hi+lo)/2;
        int left = majorityElementRec(nums, lo, mid);
        int right = majorityElementRec(nums, mid+1, hi);

        // if the two halves agree on the majority element, return it.
        if (left == right)
        {
                return left;
        }
        // otherwise, count each element and return the "winner".
        int leftCount = countInRange(nums, left, lo, hi);
        int rightCount = countInRange(nums, right, lo, hi);
        return leftCount > rightCount ? left : right;
}

public int majorityElement(int[] nums)
{
        return majorityElementRec(nums, 0, nums.length-1);
}

public static void main(String[] args)
{
        Scanner sc=new Scanner(System.in);
        int n=sc.nextInt();
        int arr[] = new int[n];
```

```
            for(int i=0;i<n;i++)
                arr[i]=sc.nextInt();
            int majority_ele = new MajorityElement().majorityElement(arr);
            System.out.println(majority_ele);
        }
}
```

4. Calculate pow(x,n):

implement power function using Iterative and Recursive

Given two integers, x and n, where x and n are +ve and –ve numbers, efficiently compute the power function pow(x, n).
For example,

pow(-2, 10) = 1024	pow(-3, 4) = 81
pow(5, 0) = 1	pow(-2, -3) = -0.125

i. Iterative Solution: Calulatepower_iterative.java

A simple solution to calculate pow(x, n) would multiply x exactly n times. We can do that by using a simple for loop.

```
class Calulatepower_iterative
{
   // Naive iterative solution to calculate `pow(x, n)`
   public static long power(int x, int n)
   {
      // initialize result by 1
      long pow = 1L;
       // multiply `x` exactly `n` times
      for (int i = 0; i < n; i++) {
         pow = pow * x;
      }
       return pow;
   }


    public static void main(String[] args)
   {
      int x = -2;
      int n = 10;
       System.out.println("pow(" + x + ", " + n + ") = " + power(x, n));
```

```
    }
}
```

Output:

```
pow(-2, 10) = 1024
```

ii. Using Divide and Conquer: Calulatepower_Recursion1.java

```
/* We can recursively define the problem as:

power(x, n) = power(x, n / 2) × power(x, n / 2);     // otherwise, n is even
power(x, n) = x × power(x, n / 2) × power(x, n / 2);    // if n is odd

*/
```

```
import java.util.*;
class Calulatepower_Recursion1
{
        static double power(double x, int y)
        {
                double temp;
                if( y == 0)
                        return 1;
                temp = power(x, y/2);

                if (y%2 == 0)
                        return temp*temp;
                else
                {
                        if(y > 0)
                                return x * temp * temp;
                        else
                                return (temp * temp) / x;
                }
        }

        public static void main(String[] args)
        {
                Scanner sc=new Scanner(System.in);
```

```
                System.out.println("enter x");
                double x = sc.nextDouble();
                System.out.println("enter n");
                int n = sc.nextInt();
                System.out.println("pow(" + x + ", " + n + ") = " + power(x, n));
        }

}
```