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).

Introduction

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(Criteria) of a Good Algorithm

1. **Input**: Zero or more quantities are externally supplied.
2. **Output:** At least one quantity is produced.
3. **Definiteness**: Each instruction is clear and unambiguous of an algorithm.
4. **Finiteness:** If we trace out the instructions of an algorithm for all cases, then the algorithm should terminate after a finite number of steps.
5. **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.

<u>Pseudo code for expressing algorithms</u>

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;
                  }
            }
}
```

<u>Performance Analysis:</u>

- 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. <u>Space Complexity:</u>

- 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.

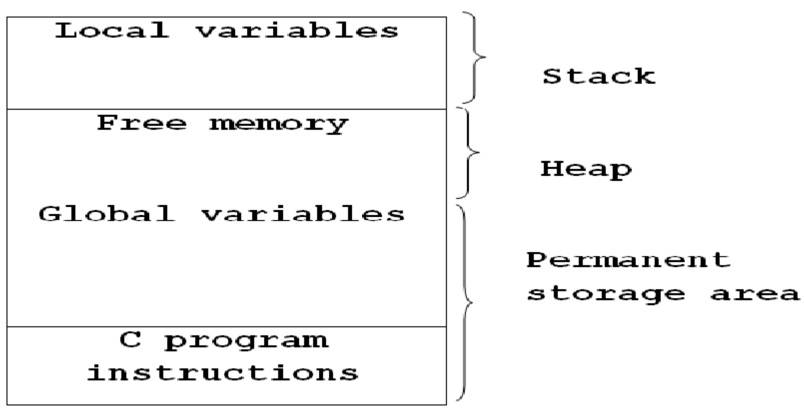
- Space taken by the variables and constants varies from language to language and platform to platform.

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.

<u>Summary of space complexity:</u>

- ➢ 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 compile 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)=**C +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 sum(a,n)

```
{
    s:=0;
   for i:=1 to n do
         s:=s+a[i];
     return s;
}
```

Recall: **Address of the first element of the array will be passed.**

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

Ex-2:

Algorithm RSum(a,n)

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

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

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

II. Time Complexity:

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

- 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 an 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.***

<u>Methods to compute the step count:</u>

1) **Introduce global variable count into programs with initial value zero.**

- Statements to increment count by the appropriate amount are introduced into the program.
- The value of the count by the time program terminates is the number steps taken by the program.

2) **Tabular method**

- Determine the total number of steps contributed by each statement per execution × frequency
- Add up the contribution of all statements

Method-I: Introduce variable count into programs

Ex-1:- **Iterative sum of n numbers**

```
Algorithm sum(a,  n)
{
    s:=0;
            count:=count+1;  // for assignment statement
             for i:=1 to n do
             {      count:=count+1; // For for
                    s:=s+a[i];
                    count:=count+1; // for assignment statement
             }
                 count:=count+1; // for last time of for
              return s;
        count:=count+1; // for return
   }
```

2n + 3 steps

Note: *Step count tells us how the run time for a program changes with changes in the instance characteristics.*

Ex-2:- Addition of two m×n matrices

Algorithm Add(a,b,c,,m,n)

```
{
  for i:=1 to m do
  {
        for j:=1 to n do
        {
            c[i,j]:=a[i,j]+b[i,j];

        }
  }
}
```

➔ **2mn + 2m+1 steps**

Ex-3:- Recursive sum of n numbers

```
Algorithm RSum(a,n)
{
    count:=count+1; // for the if conditional
    if(n ≤ 0) then
    {
            return 0;
            count:=count+1; // for the  return
    }
    else
    {
            return RSum(a,n-1)+a[n];
            count:=count+1;   // For the addition, function invocation and return
    }
}
```

- When analyzing a recursive program for its step count, we often obtain a recursive formula for the step count.
- We obtain the following recursive formula for above (RSum) algorithm.

$$t_{RSum}(n) = \begin{cases} \mathbf{2} & \textbf{If } \mathbf{n=0} \\ \mathbf{2+ t_{RSum}(n-1)} & \textbf{If } \mathbf{n>0} \end{cases}$$

- One way of solving such recursive formula is by repeated substitutions for each occurrence of the function t_{RSum} on the right side until all such occurrences disappear:

$$\begin{aligned} t_{RSum}(n) &= 2+t_{RSum}(n-1) \\ &= 2+2+t_{RSum}(n-2) \\ &= 2(2)+t_{RSum}(n-2) \\ &\quad \vdots \\ &\quad \vdots \\ &= n(2)+t_{RSum}(n-n) \\ &= 2n+t_{RSum}(0) \\ &= 2n+2 \end{aligned}$$

The step count for Rsum is 2n+2

Method-II: Tabular method(steps for execution (s/e))

Ex-1:- Iterative sum of n numbers

Statement	s/e	frequency	Total steps

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

Total			2n+3

Ex-2:- Addition of two m×n matrices

Statement	s/e	frequency	Total steps

Algorithm Add(a,b,c,m, n)
{
for i:=1 to m do
for j:=1 to n do
c[i,j]:=a[i,j]+b[i,j] ;
}
	0
	0
	1
	1
	1
	0
		--
		--
		m+1
		m(n+1)
		mn
		--
			0
			0
			m+1
			mn+m
			mn
			0

Total			2mn+2m+1

Ex-3:- Recursive sum of n numbers.

Statement	s/e	Frequency n=0 n>0	Total steps n=0 n>0

Algorithm RSum(a,n)	**0**	**--**	**--**	**0**	**0**
{	**0**	**--**	**--**	**0**	**0**
if(n ≤ 0) then	**1**	**1**	**1**	**1**	**1**
return 0;	**1**	**1**	**0**	**1**	**0**
else return					
Rsum(a,n-1)+a[n] ;	**1+x**	**0**	**1**	**0**	**1+x**
}	**0**	**--**	**--**	**0**	**0**
Total				**2**	**2+x**

$x=t_{RSum}(n-1)$

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.

Definition: [Big "oh'']

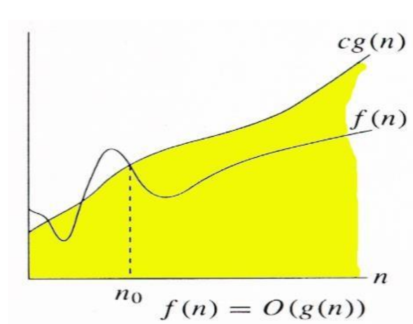
– f(*n*) = O(g(*n*))

– (read as "*f* of *n* is big oh of *g* of *n*") iff there exist positive constants 'c' and n_0 such that

f(*n*) ≤ **cg(*n*)** for all n, n ≥ n_0.

Running Time

- ➢ The definition states that the function f(n) is **at most** c times the function g(n) except when n is smaller than n_0.
- ➢ *In other words, f*(*n*) grows slower than or same rate as g(n).
- ➢ When providing an upper –bound function g for f, we normally use a single term in n.



Examples:

– *f*(*n*) = 3n+2

- 3n + 2 <= 4n, for all n >= 2, ∴3n + 2 = O (*n*)

– *f*(*n*) = $10n^2$+4n+2

  - $10n^2+4n+2 <= 11n^2$, for all $n >= 5$, $\therefore$ $10n^2+4n+2 = O(n^2)$
- $f(n)=6*2^n+n^2=O(2^n)$ /* $6*2^n+n^2 \le 7*2^n$ for $n \ge 4$ */

➢ It also possible to write $10n^2+4n+2 = O(n^3)$ since $10n^2+4n+2 <=7n^3$ for $n>=2$

➢ Although n^3 is an upper bound for $10n^2+4n+2$, it is not a tight upper bound; we can find a smaller function (n^2) that satisfies big oh relation.

2. <u>Omega (Ω) notation:</u>

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

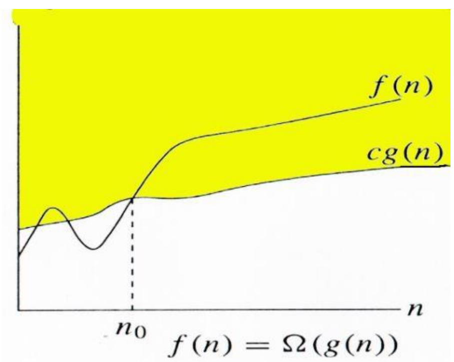
Definition: [Omega]

- ***f*(*n*) = Ω(*g*(*n*))**
- (read as "*f* of *n* is omega of *g* of *n*")
- iff there exist positive constants c and n_0 such that
- $f(n) \ge cg(n)$ for all n, $n \ge n_0$.

Running Time

➢ The definition states that the function f(n) is **at least** c times the function g(n) except when n is smaller than n_0.

➢ *In other words* , *f(n)* grows faster than or same rate as" g(n).



Examples

- $f(n) = 3n+2$
  - $3n + 2 >= 3n$, for all $n >= 1$, $\therefore 3n + 2 = \Omega(n)$
- $f(n) = 10n^2+4n+2$
  - $10n^2+4n+2 >= n^2$, for all $n >= 1$, $\therefore$ $10n^2+4n+2 = \Omega(n^2)$

➢ It also possible to write $10n^2+4n+2 = \Omega(n)$ since $10n^2+4n+2 >=n$ for $n>=0$

➢ Although n is a lower bound for $10n^2+4n+2$, it is not a tight lower bound; we can find a larger function (n^2) that satisfies omega relation.

➢ But, we can not write $10n^2+4n+2 = \Omega(n^3)$, since it does not satisfy the omega relation for sufficiently large input.

3. <u>Theta (Θ) notation:</u>

– The Theta notation describes **a tight bound** on the **asymptotic growth rate** of the function **f**.

– **Definition:** [Theta]

– ***f*(*n*) = Θ(*g*(*n*))**

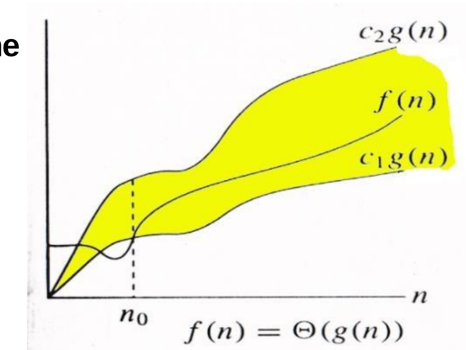
– (read as "*f* of *n* is theta of *g* of *n*")]

– iff there exist positive constants c_1, c_2, and n_0 such that

- $c_1g(n) \le f(n) \le c_2g(n)$ for all n, $n \ge n_0$.

- The definition states that **the function f(n) lies between c1 times the function g(n) and c2 times the function g(n)** except when n is smaller than n_0.
- In other **words, f(n) grows same rate as" g(n).**

Examples:-



- $f(n) = 3n+2$
    - $3n <= 3n + 2 <= 4n$, for all $n >= 2$, $\therefore$ $3n + 2 = \Theta(n)$
- $f(n) = 10n^2+4n+2$
    - $n^2 <= 10n^2+4n+2 <= 11n^2$, for all $n >= 5$, $\therefore$ $10n^2+4n+2 = \Theta(n^2)$

- But, we can not write either $10n^2+4n+2= \Theta(n)$ or $10n^2+4n+2= \Theta(n^3)$, since neither of these will satisfy the theta relation.

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).
- Let f(n) and g(n) are the functions that map positive real numbers. We can say that the function f(n) is o(g(n)) if for any real positive constant c, there exists an integer constant $n0 \leq 1$ such that $f(n) > 0$.

Mathematical Relation of Little o notation

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Definition: [Little "oh''']

- f(*n*) = o(g(*n*)) (read as "*f* of *n* is little oh of *g* of *n*") iff **f(*n*) <cg(*n*)** for all non negative values of 'n',where $n \geq n_0$.
-

- The definition states that the function f(n) is less than c times the function g(n) except when n is smaller than n_0.
- *In other words, f*(*n*) grows slower than" g(n).

Examples

- $f(n) = 3n+2=o(n^2)$
    - Since
- *However,* $3n+2 \neq o(n)$

Big-Oh, Theta, Omega and Little-oh:

Tips :

- Think of **O(g(n))** as "less than or equal to" g(n)
  - **Upper bound:** "grows slower than or same rate as" g(n)
- Think of **Ω(g(n))** as "greater than or equal to" g(n)
  - **Lower bound:** "grows faster than or same rate as" g(n)
- Think of **Θ(g(n))** as "equal to" g(n)
  - **"Tight" bound: same growth rate**
- Think of **o(g(n))** as "less than to" g(n)
  - Strict Upper bound: "grows slower than " g(n)
  - (*True for large N*)

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**
n2	**Growth is quadratic**
n3	**Growth is cubic**
2n	**Growth is exponential**
n!	**Growth is factorial**

$$1 < \log n < n < n\log n < n^2 < n^3 < 2^n < n!$$

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}

Figure 1.7 Function values

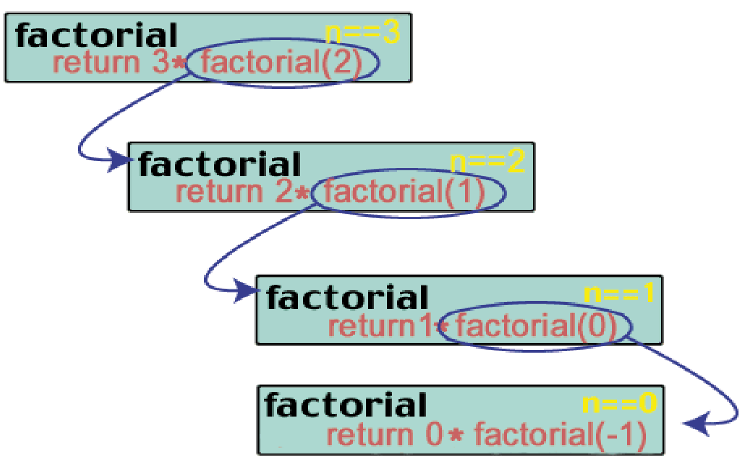
Recursion:

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.
- 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.



Tyeps of Recursions:

Recursion is a process in which a method calls itself to solve a problem. In Java, recursion can be broadly classified into two main types:

1. Direct Recursion:

- **Description:** When a method calls itself directly, it is known as direct recursion. This is the most common form of recursion.

Example: Factorial of a number.

```
public class DirectRecursionExample
{
    // Method to calculate factorial of a number
    public static int factorial(int n)
      {
        if (n == 0) {  // Base condition to stop recursion
            return 1;
        }
        return n * factorial(n - 1);  // Recursive call
    }
    public static void main(String[] args)
      {
        int number = 5;
        int result = factorial(number);
        System.out.println("Factorial of " + number + " is: " + result);
    }
}
```

2. Indirect Recursion:

- **Description:** In indirect recursion, a method calls another method, which eventually leads back to the first method.

Example: Two methods calling each other.

```
public class IndirectRecursionExample
{
    // Method A
    public static void methodA(int n)
  {
        if (n > 0)
        {
            System.out.println("In method A: " + n);
            methodB(n - 1);  // Calls methodB
        }
```

```
    }
    // Method B
    public static void methodB(int n) {
        if (n > 0) {
            System.out.println("In method B: " + n);
            methodA(n - 1);  // Calls methodA, creating indirect recursion
        }
    }
    public static void main(String[] args) {
        methodA(5);  // Start recursion
    }
}
```

3. Tail Recursion

- **Description:** Tail recursion is a special case of recursion where the recursive call is the last statement in the method. It is optimized by the compiler or runtime, leading to better performance.

Example: Tail-recursive factorial.

```
public class TailRecursionExample
{
    // Tail-recursive method to calculate factorial
    public static int tailFactorial(int n, int result)
    {
        if (n == 0)
        {
            return result;
        }
        return tailFactorial(n - 1, n * result);  // Tail-recursive call
    }

    public static void main(String[] args)
    {
        int number = 5;
        int result = tailFactorial(number, 1);  // Start with result as 1
        System.out.println("Factorial of " + number + " is: " + result);
    }
}
```

4. Non-Tail Recursion:

- **Description:** In non-tail recursion, the recursive call is not the last operation performed. The method needs to perform additional operations after the recursive call.

Example: Fibonacci series.

```
public class NonTailRecursionExample
{
    // Method to calculate nth Fibonacci number
    public static int fibonacci(int n)
   {
        if (n <= 1)
          {
            return n;
        }
        return fibonacci(n - 1) + fibonacci(n - 2);  // Non-tail recursive call
    }

    public static void main(String[] args) {
        int number = 5;
        int result = fibonacci(number);
        System.out.println("Fibonacci of " + number + " is: " + result);
    }
}
```

5. **Mutual Recursion:**

- **Description:** This is a form of indirect recursion where two or more methods call each other in a mutually recursive way.

Example: Odd and even number checker using mutual recursion.

```
public class MutualRecursionExample
{
    // Method to check if a number is even
    public static boolean isEven(int n)
   {
        if (n == 0)
       {
            return true;  // Base case
        }
        return isOdd(n - 1);  // Mutual recursion
    }

    // Method to check if a number is odd
    public static boolean isOdd(int n)
     {
        if (n == 0)
```

```
  {
     return false;  // Base case
   }
   return isEven(n - 1);  // Mutual recursion
}

public static void main(String[] args)
  {
   int number = 5;
   if (isEven(number))
   {
      System.out.println(number + " is even.");
   } else {
      System.out.println(number + " is odd.");
   }
}
```

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, ...

<u>Fibonacci Series Using Recursion in Java :</u> 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) +" ");
            }
    }
}
```

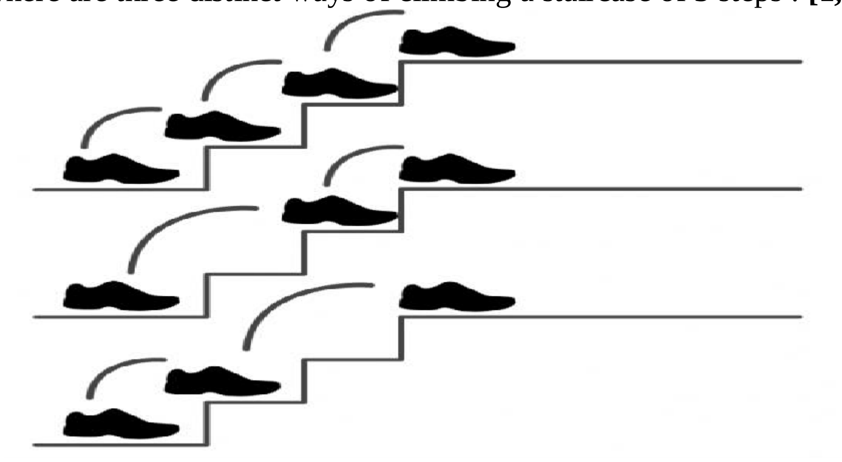
Output: Fibonacci Series of 10 numbers: 0 1 1 2 3 5 8 13 21 34

2. <u>Climbing Stairs:</u>

- 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 3 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)

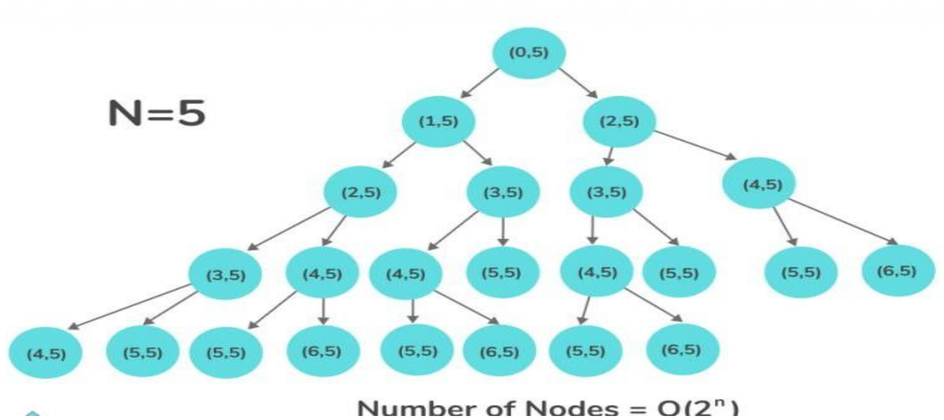
<u>Method-1: Brute Force (Recursive) Approach</u>

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.

<u>The recursive function would be:</u>

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)**.

<u>**Java program for method-1**</u>**: Stairs_M1.java**

```
import java.util.*;

class Stairs_M1 {
	// A simple recursive program to find
	// n'th fibonacci number

	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));
	}
}
```

3. Reverse String

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

- First, remove the first character from the string and append that character at the end of the string.
- Repeat the above step until the input string becomes empty.

Suppose, the input string **KMIT** is to be reversed. We will remove the first character from the string and append it to a variable **reverse sting**.

Let's implement the functionality in a Java program and reverse the string using recursion.

- In the following example, we have created a method named **reverseString().**
- It parses the string that we want to reverse. Inside the method, first, we have checked that the string is empty or not. If the string is empty, it returns the same string and also prints **String is empty**.
- We have used the following two methods of the String class:

substring(): It returns the new string that is a substring of the specified string. It parses a parameter that specifies the starting index (beginning) of the substring.

charAt(): It returns a character at the specified index. The index lies between 0 too length()-1.

Java program for string reverse using recursion: **ReverseString.java**

```
import java.util.*;
public class ReverseString
{

	public static void main(String[] args)
	{

	 Scanner sc =new Scanner(System.in);
	System.out.println("enter the string");

				String myStr = sc.next();

			//create Method and pass and input parameter string

			String reversed = reverseString(myStr);
			System.out.println("The reversed string is: " + reversed);

	}
```

```
    //Method take string parameter and check string is empty or not

    public static String reverseString(String myStr)
    {
        if (myStr.isEmpty())
        {
         System.out.println("String in now Empty");
         return myStr;
        }

        //Calling Function Recursively

System.out.println("String to be passed in Recursive Function:   +myStr.substring(1));

return reverseString(myStr.substring(1)) + myStr.charAt(0);
    }

 }
```

Output:

enter the string
kmit
String to be passed in Recursive Function: mit
String to be passed in Recursive Function: it
String to be passed in Recursive Function: t
String to be passed in Recursive Function:
String in now Empty
The reversed string is: timk

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;
  }
}
```

Example_1:
Input=
Enter a number
13
Output=
Happy Number
Example_2:
Input=
Enter a number
36
Output=
Not a Happy number

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 divisions 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));

        }
     }
```

Input:

Enter first number

63

Enter second number

21

Output:

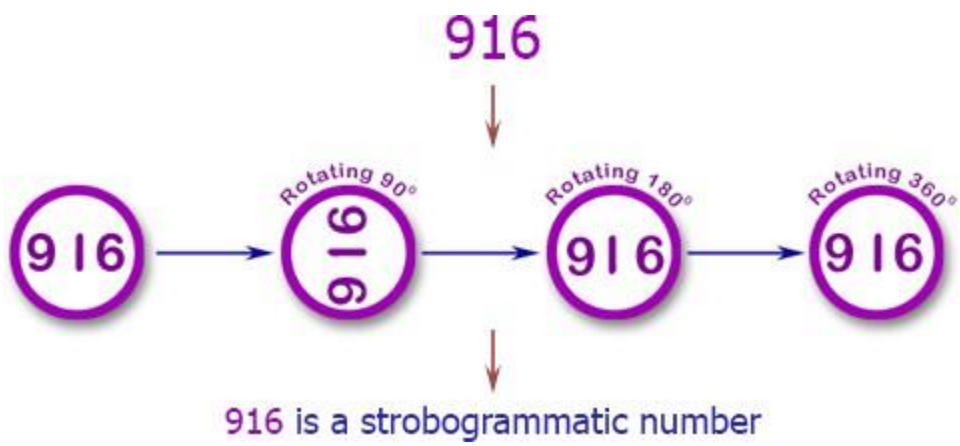
21

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, 609, 808, 906, 111, 619, 818, 916, 181, 689, 888, 986
n = 4: 1001, 6009, 8008, 9006, 1111, 6119, 8118, 9116, 1691, 6699, 8698, 9696, 1881, 6889, 8888, 9886, 1961, 6969, 8968, 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], become 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. It should be noted that when the n layer is added.

[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.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Strobogrammatic_Number_II
{
   public static void main(String[] args)
         {
       Strobogrammatic_Number_II out = new Strobogrammatic_Number_II();
       Solution s = out.new Solution();
         Scanner sc=new Scanner(System,in);
         int n= sc.nextInt();
       System.out.println(s.findStrobogrammatic(n));
   }
  class Solution
 {
   // not char[], because List can direct return as result
       List<String> singleDigitList = new ArrayList<>(Arrays.asList("0", "1", "8"));
         // except '0', a special case
     char[][] digitPair = { {'1', '1'}, {'8', '8'}, {'6', '9'}, {'9', '6'} };
       public List<String> findStrobogrammatic(int n)
         {
```

```
            return dfs(n, n);
        }

        public List<String> dfs(int k, int n) {
            if (k <= 0) {
                return new ArrayList<String>(Arrays.asList(""));
            }
            if (k == 1) {
                return singleDigitList;
            }
            List<String> subList = dfs(k - 2, n);
            List<String> result = new ArrayList<>();
            for (String str : subList) {
                if (k != n) { // @note: cannot start with 0
                    result.add("0" + str + "0");
                }
                for (char[] aDigitPair : digitPair)
                {
                    result.add(aDigitPair[0] + str + aDigitPair[1]);
                }
            }
            return result;
        }
    }
}
```

<u>Iutput:</u>

2

<u>Output:</u>

[11, 88, 69, 96]

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 $n_1,n_2,\ldots,n_k$, 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 and c are known constants, and ***n is a power of b** (i.e **$n=b^k$**)*

Applications:

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

Master Theorem for Divide and Conquer Recurrences

- All divide and conquer algorithms (also discussed in detail in the Divide and Conquer chapter) divide the problem into sub-problems, each of which is part of the original problem, and then perform some additional work to compute the final answer.
- As an example, a merge sort algorithm [for details, refer to Sorting chapter] operates on two sub-problems, each of which is half the size of the original, and then performs O(n) additional work for merging.
- This gives the running time equation:

  $T(n) = 2T(n/2) + O(n)$
- The following theorem can be used to determine the running time of the divide and conquer algorithms.
- For a given program (algorithm), first, we try to find the recurrence relation for the problem.
- If the recurrence is of the below form then we can directly give the answer without fully solving it.
- If the recurrence is of the form , $\mathbf{T(n) = aT\ (n/b) + \Theta(n^k \log^p n)}$ where $a \geq 1$, $b > 1, k \geq 0$ and p is a real number, then:

Total Three Cases:

1) If $a > b^k$, then $T(n) = \Theta\left(n^{\log_b^a}\right)$
2) If $a = b^k$
   - a. If $p > -1$, then $T(n) = \Theta\left(n^{\log_b^a} \log^{p+1} n\right)$
   - b. If $p = -1$, then $T(n) = \Theta\left(n^{\log_b^a} \log\log n\right)$
   - c. If $p < -1$, then $T(n) = \Theta\left(n^{\log_b^a}\right)$
3) If $a < b^k$
   - a. If $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$
   - b. If $p < 0$, then $T(n) = O(n^k)$

Divide and Conquer Master Theorem: Problems & Solutions

For each of the following recurrences, give an expression for the runtime T(n) if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

Problem-1 $T(n) = 3T(n/2) + n^2$
Solution: $T(n) = 3T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 3.a)

Problem-2 $T(n) = 4T(n/2) + n^2$
Solution: $T(n) = 4T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2 logn)$ (Master Theorem Case 2.a)

Problem-3 $T(n) = T(n/2) + n^2$
Solution: $T(n) = T(n/2) + n^2 \Rightarrow \Theta(n^2)$ (Master Theorem Case 3.a)

Problem-4 $T(n) = 2^n T(n/2) + n^n$
Solution: $T(n) = 2^n T(n/2) + n^n \Rightarrow$ Does not apply (a is not constant)

Problem-5 $T(n) = 16T(n/4) + n$
Solution: $T(n) = 16T(n/4) + n \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-6 $T(n) = 2T(n/2) + nlogn$
Solution: $T(n) = 2T(n/2) + nlogn \Rightarrow T(n) = \Theta(nlog^2 n)$ (Master Theorem Case 2.a)

Problem-7 $T(n) = 2T(n/2) + n/logn$
Solution: $T(n) = 2T(n/2) + n/logn \Rightarrow T(n) = \Theta(nloglogn)$ (Master Theorem Case 2. b)

Problem-8 $T(n) = 2T(n/4) + n^{051}$
Solution: $T(n) = 2T(n/4) + n^{051} \Rightarrow T(n) = \Theta(n^{0.51})$ (Master Theorem Case 3.b)

Problem-9 $T(n) = 0.5T(n/2) + 1/n$
Solution: $T(n) = 0.5T(n/2) + 1/n \Rightarrow$ Does not apply ($a < 1$)

Problem-10 $T(n) = 6T(n/3) + n^2 logn$
Solution: $T(n) = 6T(n/3) + n^2 logn \Rightarrow T(n) = \Theta(n^2 logn)$ (Master Theorem Case 3.a)

Problem-11 $T(n) = 64T(n/8) - n^2 logn$
Solution: $T(n) = 64T(n/8) - n^2 logn \Rightarrow$ Does not apply (function is not positive)

Problem-12 $T(n) = 7T(n/3) + n^2$
Solution: $T(n) = 7T(n/3) + n^2 \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 3.as)

Problem-13 $T(n) = 4T(n/2) + logn$
Solution: $T(n) = 4T(n/2) + logn \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-14 $T(n) = 16T(n/4) + n!$
Solution: $T(n) = 16T(n/4) + n! \Rightarrow T(n) = \Theta(n!)$ (Master Theorem Case 3.a)

Problem-15 $T(n) = \sqrt{2}T(n/2) + logn$
Solution: $T(n) = \sqrt{2}T(n/2) + logn \Rightarrow T(n) = \Theta(\sqrt{n})$ (Master Theorem Case 1)

Problem-16 $T(n) = 3T(n/2) + n$
Solution: $T(n) = 3T(n/2) + n \Rightarrow T(n) = \Theta(n^{log3})$ (Master Theorem Case 1)

Problem-17 $T(n) = 3T(n/3) + \sqrt{n}$
Solution: $T(n) = 3T(n/3) + \sqrt{n} \Rightarrow T(n) = \Theta(n)$ (Master Theorem Case 1)

Problem-18 $T(n) = 4T(n/2) + cn$
Solution: $T(n) = 4T(n/2) + cn \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-19 $T(n) = 3T(n/4) + nlogn$
Solution: $T(n) = 3T(n/4) + nlogn \Rightarrow T(n) = \Theta(nlogn)$ (Master Theorem Case 3.a)

Problem-20 $T(n) = 3T(n/3) + n/2$
Solution: $T(n) = 3T(n/3) + n/2 \Rightarrow T(n) = \Theta(nlogn)$ (Master Theorem Case 2.a)

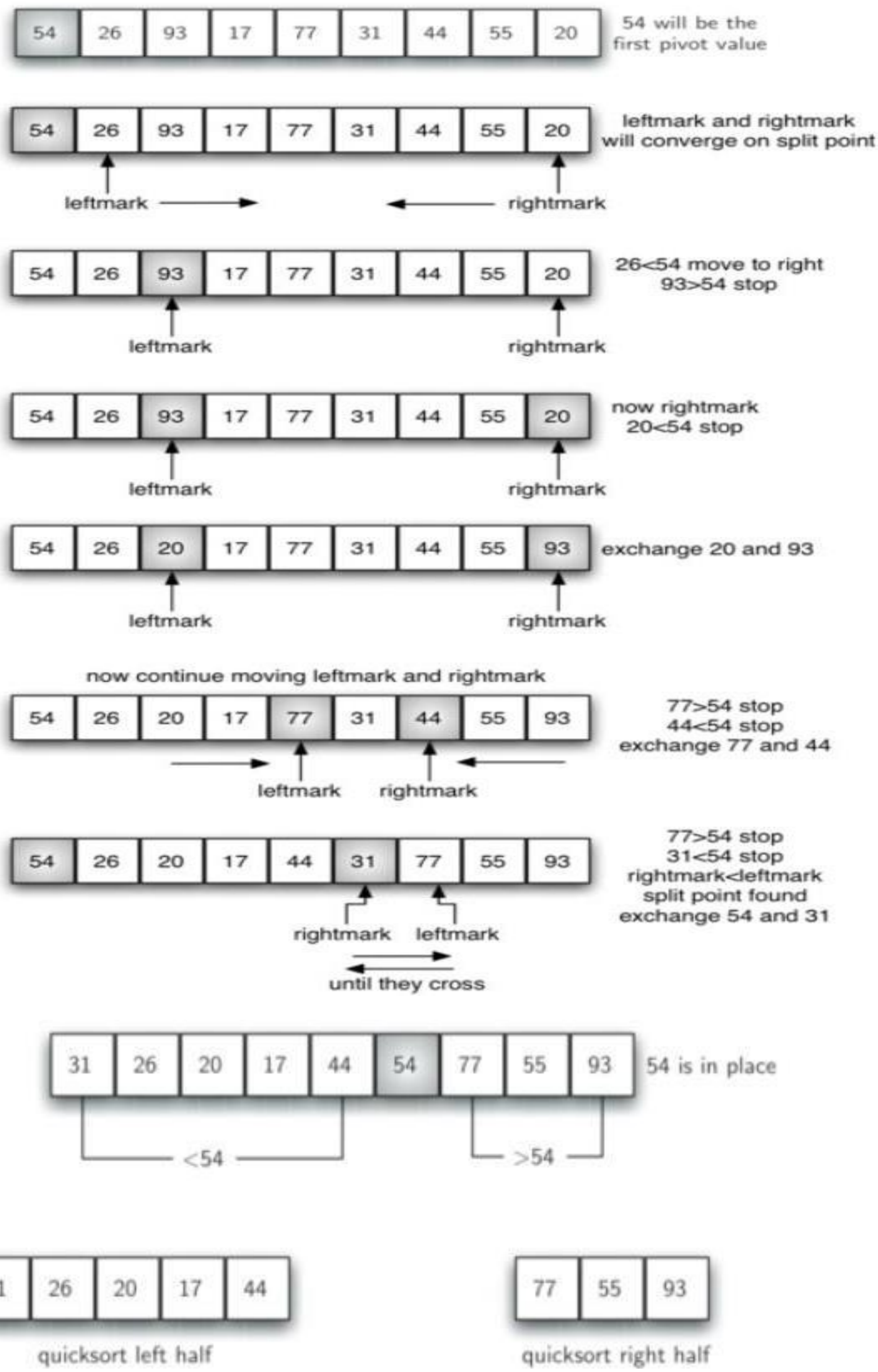
1. Quick sort.

- 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.

<u>Example:</u>


54 26 93 17 77 31 44 55 20
54 will be the first pivot value
leftmark and rightmark will converge on split point
leftmark
rightmark
26<54 move to right
93>54 stop
now rightmark
20<54 stop
exchange 20 and 93
now continue moving leftmark and rightmark
77>54 stop
44<54 stop
exchange 77 and 44
77>54 stop
31<54 stop
rightmark<leftmark
split point found
exchange 54 and 31
until they cross
31 26 20 17 44 54 77 55 93
54 is in place
<54
>54
quicksort left half
quicksort right half

Pseudo Code for Quick Sort (Recursion):

```
Algorithm QuickSort(p, q)
// Sorts the elements a[p], ..., a[q] which reside in the global
// array a[1 : n] into ascending order; a[n + 1] is considered to
// be defined and must be ≥ all the elements in a[1 : n].
{
    if (p < q) then  // If there are more than one element
    {
        // divide P into two subproblems.
            j := Partition(a, p, q + 1);
                // j is the position of the partitioning element.
        // Solve the subproblems.
            QuickSort(p, j − 1);
            QuickSort(j + 1, q);
        // There is no need for combining solutions.
    }
}
```

```
Algorithm Partition(a, m, p)
// Within a[m], a[m + 1], ..., a[p − 1] the elements are
// rearranged in such a manner that if initially t = a[m],
// then after completion a[q] = t for some q between m
// and p − 1, a[k] ≤ t for m ≤ k < q, and a[k] ≥ t
// for q < k < p. q is returned. Set a[p] = ∞.
{
    v := a[m]; i := m; j := p;
    repeat
    {
        repeat
            i := i + 1;
        until (a[i] ≥ v);

        repeat
            j := j − 1;
        until (a[j] ≤ v);

        if (i < j) then Interchange(a, i, j);

    } until (i ≥ j);

    a[m] := a[j]; a[j] := v; return j;
}
```

```
Algorithm Interchange(a, i, j)
// Exchange a[i] with a[j].
{
    p := a[i];
    a[i] := a[j]; a[j] := p;
}
```

Pseudo Code for Quick Sort (Iterative):

```
Algorithm QuickSort2(p, q)
// Sorts the elements in a[p : q].
{
    // stack is a stack of size 2 log(n).
    repeat
    {
        while (p < q) do
        {
            j := Partition(a, p, q + 1);
            if ((j - p) < (q - j)) then
            {
                Add(j + 1); // Add j + 1 to stack.
                Add(q); q := j - 1; // Add q to stack
            }
            else
            {
                Add(p); // Add p to stack.
                Add(j - 1); p := j + 1; // Add j - 1 to stack
            }
        } // Sort the smaller subfile.
        if stack is empty then return;
        Delete(q); Delete(p); // Delete q and p from stack.
    } until (false);
}
```

Pseudo Code for Randomized Quick Sort Algorithm:

```
Algorithm RQuickSort(p, q)
// Sorts the elements a[p], ..., a[q] which reside in the global
// array a[1 : n] into ascending order. a[n + 1] is considered to
// be defined and must be ≥ all the elements in a[1 : n].
{
    if (p < q) then
    {
        if ((q - p) > 5) then
            Interchange(a, Random() mod (q - p + 1) + p, p);
        j := Partition(a, p, q + 1);
            // j is the position of the partitioning element.
        RQuickSort(p, j - 1);
        RQuickSort(j + 1, q);
    }
}
```

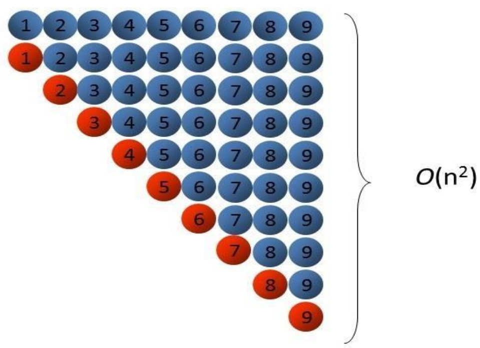
Time complexity analysis:

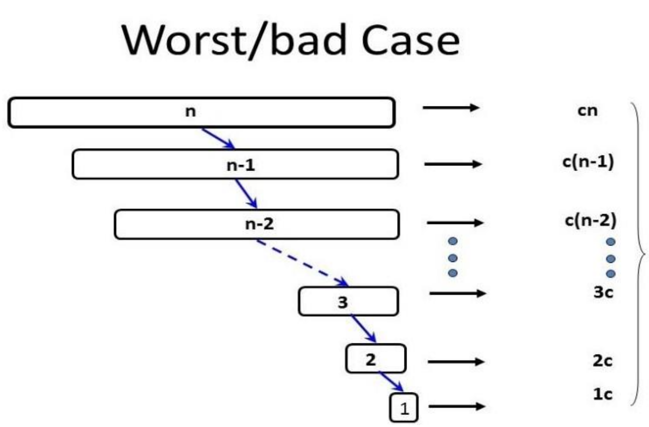
- The time required to sort n elements using quicksort involves 3 components.
    - Time required for partitioning the array, which is *roughly proportional to n.*
    - Time required for sorting lower subarray.
    - Time required for sorting upper subarray.
- Assume that there are k elements in the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

$$T(n) = \begin{cases} c_1 & n=1, c1 \text{ is a constant} \\ T(n) = 2T(n/2) + c\,n & n>1 \end{cases}$$

A worst/bad case

It occurs if the list is already in sorted order





contd...

- 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,

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

$$\begin{aligned}
&= T(n-1) + c_2 n \\
&= T(n-2) + c_2(n-1) + c_2 n \\
&= T(n-3) + c_2(n-2) + c_2(n-1) + c_2 n \\
&\quad \ldots \\
&\quad \ldots \\
&= n(n+1)/2 = (n^2+n)/2 = O(n^2)
\end{aligned}$$

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 QuickSort
{
   //selects last element as pivot, pi using which array is partitioned.
   int partition(int intArray[], int low, int high)
  {
      int pi = intArray[high];
      int i = (low-1); // smaller element index
      for (int j=low; j<high; j++)
     {
         // check if current element is less than or equal to pi
         if (intArray[j] <= pi)
        {
            i++;
            // swap intArray[i] and intArray[j]
            int temp = intArray[i];
            intArray[i] = intArray[j];
            intArray[j] = temp;
         }
      }
```

```
        // swap intArray[i+1] and intArray[high] (or pi)
        int temp = intArray[i+1];
        intArray[i+1] = intArray[high];
        intArray[high] = temp;

        return i+1;
    }
    //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(intArray, low, high);

            // sort each partition recursively
            quick_sort(intArray, low, pi-1);
            quick_sort(intArray, pi+1, high);
        }
    }
}
 class Quicksort_Recursive
{
    public static void main(String args[])
 {
            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 QuickSort object
```

```
        QuickSort obj = new QuickSort();

        obj.quick_sort(a, 0, n-1);

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

Output:

```
enter array size
5
enter the elements of array
10
9
5
78
14
Original Array: [10, 9, 5, 78, 14]
Sorted Array: [5, 9, 10, 14, 78]
```

Iterative Quicksort

In iterative quicksort, we use the auxiliary stack to place intermediate parameters instead of using recursion and sort partitions.

Java program implements iterative quicksort: **Quicksort_iterative.java**

```
import java.util.*;

class Quicksort_iterative
{
    //partitions the array around pivot=> last element
    static int partition(int numArray[], int low, int high)
{
        int pivot = numArray[high];
        // smaller element index
        int i = (low - 1);
        for (int j = low; j <= high - 1; j++) {
            // check if current element is less than or equal to pivot
            if (numArray[j] <= pivot) {
                i++;
                // swap the elements
                int temp = numArray[i];
                numArray[i] = numArray[j];
                numArray[j] = temp;
```

```
            }
        }
        // swap numArray[i+1] and numArray[high] (or pivot)
        int temp = numArray[i + 1];
        numArray[i + 1] = numArray[high];
        numArray[high] = temp;
        return i + 1;
    }
//sort the array using quickSort
    static void quickSort(int numArray[], int low, int high)
    {
        //auxillary stack
        int[] intStack = new int[high - low + 1];

        // top of stack initialized to -1
        int top = -1;

        // push initial values of low and high to stack
        intStack[++top] = low;
        intStack[++top] = high;
          // Keep popping from stack while is not empty
        while (top >= 0) {
            // Pop h and l
            high = intStack[top--];
            low = intStack[top--];
              // Set pivot element at its correct position
            // in sorted array
            int pivot = partition(numArray, low, high);

            // If there are elements on left side of pivot,
            // then push left side to stack
            if (pivot - 1 > low) {
                intStack[++top] = low;
                intStack[++top] = pivot - 1;
            }
              // If there are elements on right side of pivot,
            // then push right side to stack
            if (pivot + 1 < high) {
                intStack[++top] = pivot + 1;
                intStack[++top] = high;
            }
```

```
        }
    }
 public static void main(String args[])
    {

       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();
            }


        System.out.println("Original Array:" + Arrays.toString(a));
        // call quickSort routine to sort the array
        quickSort(a, 0, n - 1);
        //print the sorted array
        System.out.println("\nSorted Array:" + Arrays.toString(a));
    }
}
```

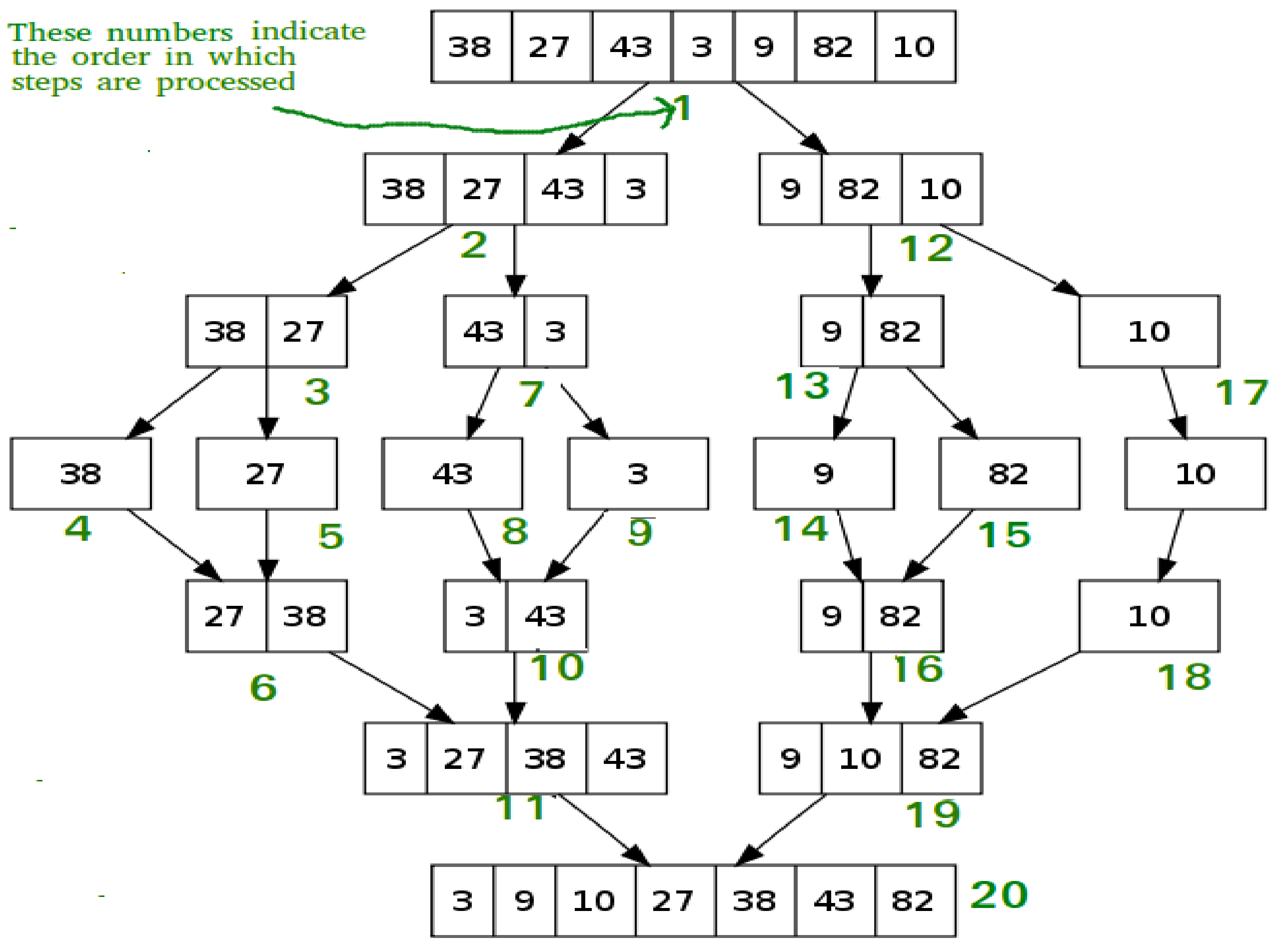
Output:

```
enter array size
5
enter the elements of array
12
32
41
56
20
Original Array:[12, 32, 41, 56, 20]
Sorted Array:[12, 20, 32, 41, 56]
```

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)
// a[low : high] is a global array to be sorted.
// 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
    {
        // Divide P into subproblems.
            // Find where to split the set.
                mid := ⌊(low + high)/2⌋;
        // Solve the subproblems.
            MergeSort(low, mid);
            MergeSort(mid + 1, high);
        // Combine the solutions.
            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 an auxiliary 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];
}
```

Calculating Time complexity Using Substitute Method:

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

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

When n is a power of 2, $n = 2^k$, we can solve this equation by successive substitutions:

$$\begin{aligned} T(n) &= 2(2T(n/4) + cn/2) + cn \\ &= 4T(n/4) + 2cn \\ &= 4(2T(n/8) + cn/4) + 2cn \\ &\vdots \\ &= 2^k T(1) + kcn \\ &= an + cn \log n \end{aligned}$$

It is easy to see that if $2^k < n \leq 2^{k+1}$, then $T(n) \leq T(2^{k+1})$. Therefore

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

Time complexities: Most of the work done in combining the solutions.

- **Best case** : ***o*(n log(n)) time**
- **Average case** :*o*(n log(n)) time
- **Worst case** : *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;

    public class Mergesort_recursive
     {
       public static void main(String a[])
         {

           int temp;
           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 \n");
            for(int i = 0; i < list.length; i++)
            System.out.println( list[i]+"  ");
            mergeSort(list,0, list.length-1);
            System.out.print("List after sorting \n");
            for(int i = 0; i < list.length; i++)
             {
                    System.out.print(list[i]+"  ");
             }
           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);
                }
           private static void merge(int list[], int low, int middle, int high)
```

```
{
  int IstList_end= middle;

 int IIndList_start = middle + 1;

 int l=low;
  while ((l <= IstList_end) && (IIndList_start <= high))
          {

                    if (list[low] < list[IIndList_start])
                   {

        low++;

      } else {

          int temp = list[IIndList_start];

          for (int j = IIndList_start-1; j >= low; j--) {

          list[j+1] = list[j];
        }
     list[low] = temp;
     low++;
     IstList_end++;
     IIndList_start++;
     }
   }
  }
}
```

Output:

Enter aray size
5
Enter numbers
9
8
10
7
4
List before sorting
9 8 10 7 4
List after sorting
4 7 8 9 10

Merge sort using iteration in java: **Mergesort_iterative.java**

```
import java.lang.Math.*;
import java.util.*;

class Mergesort_iterative
 {
	/* Iterative mergesort function to sor t arr[0...n-1] */
	static void mergeSort(int arr[], int n)
	{
		int curr_size;

		// For picking starting index of
		// left subarray to be merged
		int left_start;
		// Merge subarrays in bottom up
		// manner. First merge subarrays
		// of size 1 to create sorted
		// subarrays of size 2, then merge
		// subarrays of size 2 to create
		// sorted subarrays of size 4, and
		// so on.
		for (curr_size = 1; curr_size <= n-1;
					curr_size = 2*curr_size)
		{

			// Pick starting point of different
			// subarrays of current size
			for (left_start = 0; left_start < n-1;
						left_start += 2*curr_size)
			{
				// Find ending point of left
				// subarray. mid+1 is starting
				// point of right
				int mid = Math.min(left_start + curr_size - 1, n-1);

				int right_end = Math.min(left_start
							+ 2*curr_size - 1, n-1);

				// Merge Subarrays arr[left_start...mid]
				// & arr[mid+1...right_end]
				merge(arr, left_start, mid, right_end);
			}
		}
	}

	/* Function to merge the two haves 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 n1 = m - l + 1;
        int n2 = r - m;
        /* create temp arrays */
        int L[] = new int[n1];
        int R[] = new int[n2];
        /* Copy data to temp arrays L[]
        and R[] */
        for (i = 0; i < n1; i++)
                L[i] = arr[l + i];
        for (j = 0; j < n2; j++)
                R[j] = arr[m + 1+ j];
        /* Merge the temp arrays back into
        arr[l..r]*/
        i = 0;
        j = 0;
        k = l;
        while (i < n1 && j < n2)
        {
                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 < n1)
        {
                arr[k] = L[i];
                i++;
                k++;
        }
    /* Copy the remaining elements of    R[], if there are any */
        while (j < n2)
        {
                arr[k] = R[j];
                j++;
                k++;
```

```
                }
            }
                /* Function to print an array */
            static void printArray(int A[], int size)
            {
                int i;
                for (i=0; i < size; i++)
                    System.out.printf("%d ", A[i]);
                System.out.printf("\n");
            }

            /* Driver program to test above functions */
public static void main(String[] args)
    {
    Scanner sc=new Scanner (System.in);
     System.out.println("Enter array size");
     int n = sc.nextInt();
      int [] arr=new int[n];
     System.out.println("Enter numbers ");
     for (int i=0 ; i<n; i++)
        {
        int  number = sc.nextInt();
        arr[i]=number;
        }
            System.out.printf("Given array is \n");
                printArray(arr, n);
                mergeSort(arr, n);
                System.out.printf("\nSorted array is \n");
                printArray(arr, n);
            }
    }
```

Input:

Enter array size

5

Enter numbers

12 13 10 9 8

Output:

Given array is

12 13 10 9 8

Sorted array is

8 9 10 12 13

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

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 subslice'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 Majority
{
   public static void main (String args[])
   {
      Majority s = new  Majority();
      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();
       }
      System.out.println(s.getMajorityElement(a,n));
   }
   static int  getMajorityElement (int X[], int n)
   {
      return getMajority(X, 0, n-1) ;
   }

static int getMajority (int X[], int l, int r)
```

```
    {
       if (l==r)
          return X[l];

       int mid = (r-l)/2 + l;
             int leftMajority = getMajority (X, l, mid);
             int rightMajority = getMajority (X, mid + 1, r);

      if(leftMajority==rightMajority)
       return leftMajority;

       int leftCount = countFrequency(X, l, r, leftMajority);
       int rightCount = countFrequency(X, l, r, rightMajority);

       if(leftCount > (r-l+1)/2)
                return leftMajority;
       else if( rightCount > (r-l+1)/2)
          return rightMajority;
          else
            return -1;
    }
    static int countFrequency (int X[], int l, int r, int majority)
    {
       int count =0;
       for (int i = l; i <= r; i = i + 1)
       {
          if (X[i] == majority)
             count = count + 1;
       }
       return count;
    }
 }
```

Example-1:

Input=

7

2 2 1 1 1 2 2

Output=

2

Example-2:

Input=

6

1 2 2 1 3 1

Output= -1

4. Calculate pow(x,n):

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

For example,

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

Appraoch-1: with O(n) Time complexity (Not D&C just Naïve Approach)

Procedure: 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:

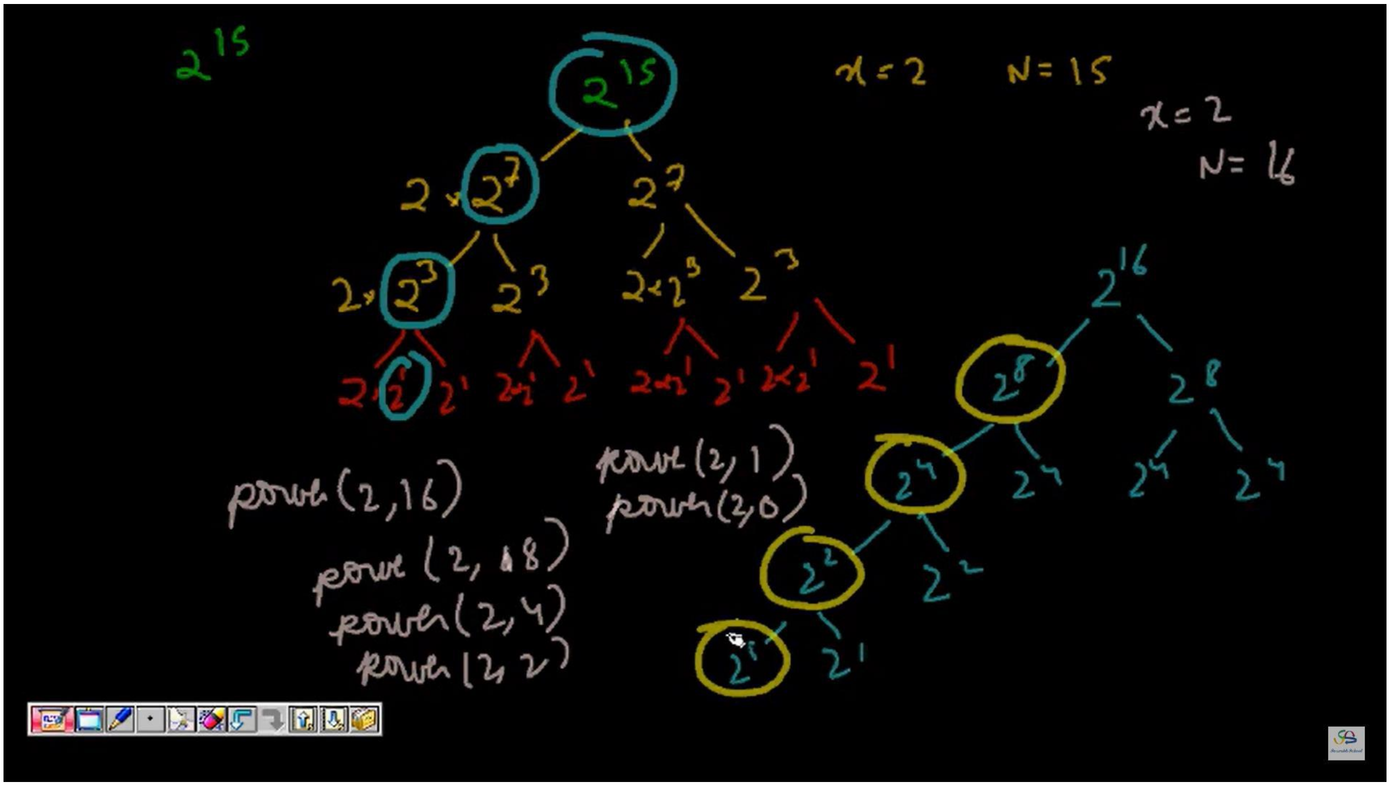
```
pow(-2, 10) = 1024
```

<u>Approach-2:</u> Using Divide and Conquer with O(nlogn) Time Complexity

We can recursively define the problem as:

Case: 1 power(x, n) = power(x, n / 2) × power(x, n / 2); // otherwise, n is even
Case: 2 power(x, n) = x × power(x, n / 2) × power(x, n / 2); // if n is odd*/

Pictorial Representation: if x=2 and n=15 & If x=2 and N=16



<u>Computing Time Complexity:</u>

Problem: Compute a^n, where $n \in \mathbb{N}$.

Naive algorithm: $\Theta(n)$.

Divide-and-conquer algorithm:

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even;} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{if } n \text{ is odd.} \end{cases}$$

$$T(n) = T(n/2) + \Theta(1) \implies T(n) = \Theta(\lg n) .$$

<u>Java program for calculate Power(x,n):</u> **Calulatepower_Recursion.java**

```
import java.util.*;

class Calulatepower_Recursion
{
    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( power(x, n));
    }
}
```

Example-1:
input=
enter x
2
enter n
3
output=
8.0

Example-2:
input =

enter x
-2
enter n
-3
output =
-0.125

Example-3:
input =
enter x
-5
enter n
9
output =
-1953125.0