

①

9

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CS3401 ALGORITHMS

UNIT I INTRODUCTION

Algorithm analysis: Time and space complexity - Asymptotic Notations and its properties Best case, Worst case and average case analysis - Recurrence relation: substitution method - Lower bounds - **searching:** linear search, binary search and Interpolation Search, **Pattern search:** The naïve string-matching algorithm - Rabin-Karp algorithm - Knuth-Morris-Pratt algorithm. **Sorting:** Insertion sort - heap sort

UNIT II GRAPH ALGORITHMS

Graph algorithms: Representations of graphs - Graph traversal: DFS - BFS - applications - Connectivity, strong connectivity, bi-connectivity - Minimum spanning tree: Kruskal's and Prim's algorithm- Shortest path: Bellman-Ford algorithm - Dijkstra's algorithm - Floyd-Warshall algorithm Network flow: Flow networks - Ford-Fulkerson method - Matching: Maximum bipartite matching

UNIT III ALGORITHM DESIGN TECHNIQUES

Divide and Conquer methodology: Finding maximum and minimum - Merge sort - Quick sort **Dynamic programming:** Elements of dynamic programming — Matrix-chain multiplication - Multi stage graph — Optimal Binary Search Trees. **Greedy Technique:** Elements of the greedy strategy - Activity-selection problem — Optimal Merge pattern — Huffman Trees.

UNIT IV STATE SPACE SEARCH ALGORITHMS

Backtracking: n-Queens problem - Hamiltonian Circuit Problem - Subset Sum Problem - Graph colouring problem **Branch and Bound:** Solving 15-Puzzle problem - Assignment problem - Knapsack Problem - Travelling Salesman Problem

UNIT V NP-COMPLETE AND APPROXIMATION ALGORITHM

Tractable and intractable problems: Polynomial time algorithms - Venn diagram representation - NP-algorithms - NP-hardness and NP-completeness - Bin Packing problem - Problem reduction: TSP - 3-CNF problem. **Approximation Algorithms:** TSP - **Randomized Algorithms:** concept and application - primality testing - randomized quick sort - Finding k^{th} smallest number

TOTAL: 45 PERIODS

UNIT-1

INTRODUCTION.

③

CS3401 ALGORITHMS

UNIT I INTRODUCTION

Algorithm analysis: Time and space complexity - Asymptotic Notations and its properties Best case, Worst case and average case analysis - Recurrence relation: substitution method - Lower bounds - **searching:** linear search, binary search and Interpolation Search, **Pattern search:** The naïve string-matching algorithm - Rabin-Karp algorithm - Knuth-Morris-Pratt algorithm. **Sorting:** Insertion sort - heap sort

Algorithm:

→ An algorithm is a finite set of instructions that accomplishes a particular task.

→ It is a step-by-step procedure for solving a problem in a finite amount of time.

Algorithm - Definition:

An algorithm is a sequence of unambiguous instructions for solving a problem.
i.e., for obtaining a required output for any legitimate input in a finite amount of time.

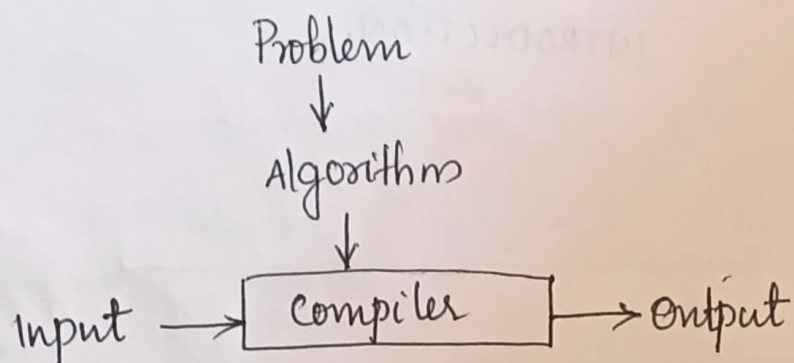


Fig: The notation of the algorithm

* Characteristics of Algorithm:

An algorithm is a method or process to solve a problem satisfying the following characteristics.

1) Input:

- Valid inputs are clearly specified.
- Zero or more quantities are externally supplied.

2) Output:

- The output is the data resulting from the computation.
- An algorithm should have one or more well-defined outputs.

3) Definiteness:

- Specifying the sequence of operations for turning input into output.

properties of algo - Non-ambiguity, Range of input, multiplicity, speed, finiteness

4) Finiteness :

→ The algorithm terminates after a finite number of steps.

5) Effectiveness :

→ Each instruction is sufficiently simple and basic.

→ It should not contain any unnecessary and redundant steps.

1.1 ALGORITHM ANALYSIS :

The term "analysis of algorithms" is defined as an investigation of an algorithm's efficiency with respect to two resources.

i) Running Time (Efficiency)

ii) Memory Space.

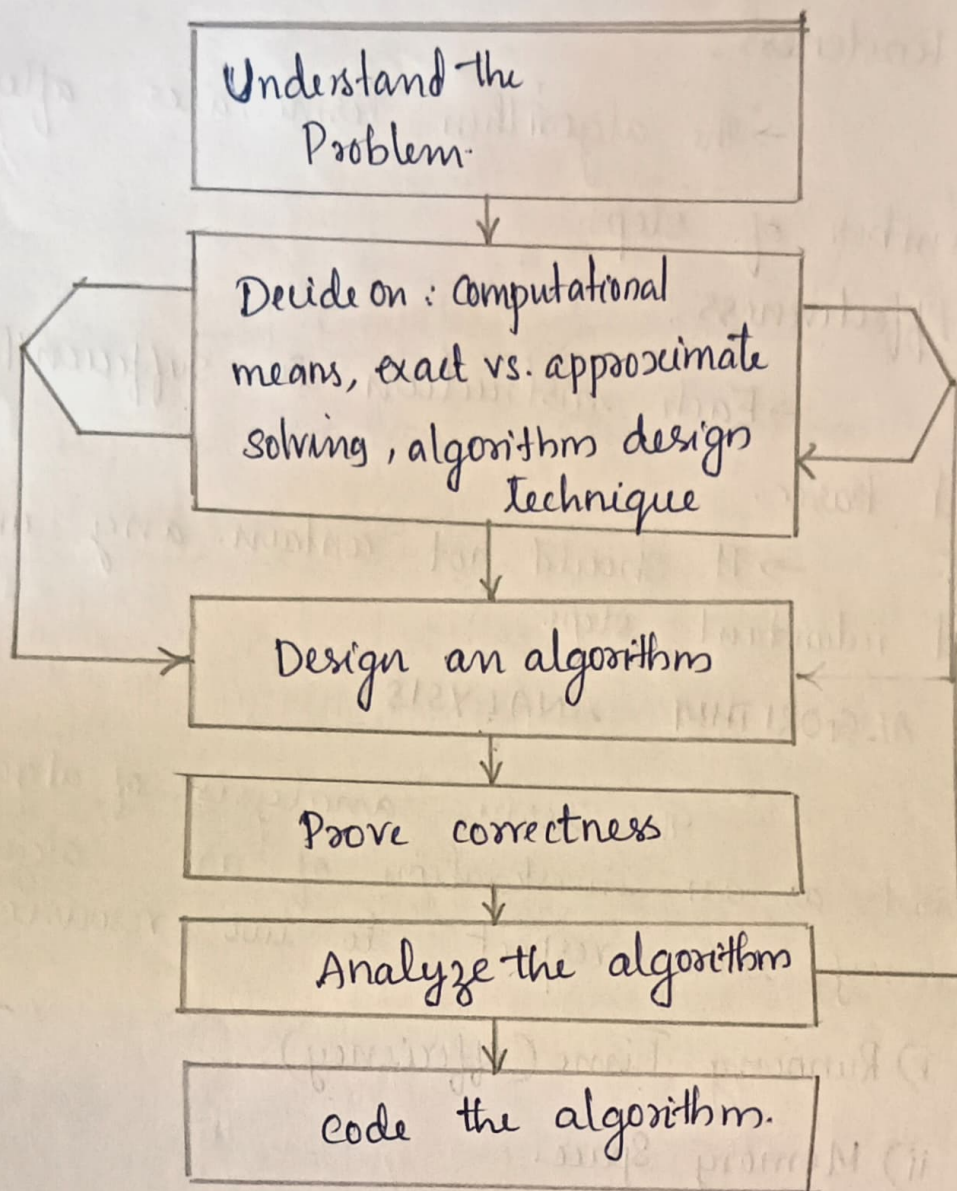


Fig: Algorithm design and analysis process

The steps that need to be followed while designing and analyzing an algorithm are,

1) Understanding the problem.

2) Decision making on,

(a) Capabilities of a computational device

- ⑦
- b) Choosing between exact and approximate problem solving.
 - c) Appropriate data structure.
 - d) Algorithms design Techniques

3) Designing an algorithm.

4) Proving the correctness of the algorithm.

5) Analyzing an algorithm.

6) Coding the algorithm.

1) Understanding the problem:

→ This is the first step in the designing of algorithm

→ In this step, we need to understand the problem statement.

→ Read the problem's description carefully and ask questions for clarifying the doubts about the problem.

→ After understanding the problem clearly, we have to find out the necessary inputs for solving the problem.

→ An input to an algorithm specifies an instance of the problem the algorithm solves.

→ It is important to specify exactly the range of inputs so that the boundary values of algorithms gets fixed.

→ The algorithm should work correctly for all valid inputs.

→ This step is important in algorithm's problem solving and it should not be skipped at all.

2) Decision Making :

(a) Ascertaining the capabilities of a computational device :

→ It is necessary to know the computational capabilities of devices on which the algorithm is running.

→ From execution point of view, an algorithm can be classified as,

- * Sequential algorithm

- * Parallel algorithm.

Sequential algorithms specifically runs on RAM in which the instructions are executed one after the another.

Parallel algorithms are executed on the machine in which the instructions are executed in parallel.

There are certain complex problems which require huge amount of memory or the problems for which execution time is an important factor.

For solving such problems, it is essential to have proper choice of computational device which is space and time efficient.

b) Choosing between exact and approximate problem solving:

If the problem needs to be solved correctly, then we need exact algorithms.

If the problem is so complex, we won't get the exact solution. In this situation, we need to choose approximate algorithms. eg: Extracting square roots, solving non-linear equation.

c) Deciding on approximate data structures:

Algorithms + data structures = programs

→ Data structure is a data organization that enables efficient access of data and modification of data.

→ Proper choice of data structure is required before designing the algorithm.

d) Algorithm Design Techniques:

→ An algorithm design technique is a general approach for solving problems algorithmically that is applicable to a variety of problems from different areas of Computing.

→ Some of the design techniques are,

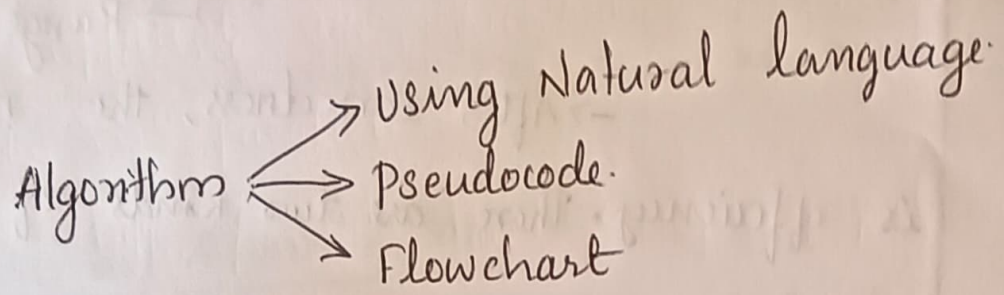
- * Brute Force

- * Divide and Conquer

- * Greedy Techniques.

3) Designing an algorithm:

After designing an algorithm, it must be specified.



→ It is simple to specify an algorithm using Natural language. But, sometimes it is not clear.

→ Pseudocode is compact and informal high level description of an algorithm. Pseudocode consists of short english phrases that explains specific tasks within a program's algorithm.

→ A flowchart is a graphical or symbolic representation of an algorithm.

4) Proving Correctness (Algorithmic Verification)

→ Algorithm verification means checking correctness of an algorithm.

→ Once an algorithm has been specified, you have to prove its correctness. i.e., The algorithm yields a required result for a legitimate input in a finite amount of time.

→ For some algorithms, it is quite easy for others, it can be quite complex.

§. Analyzing an algorithm ::

- Efficiency
- Simplicity.
- Generality.
- Range of Inputs.

→ After correctness, the most important is efficiency. There are two kinds of algorithm efficiency

* Time Efficiency → how fast the algorithm runs

* Space Efficiency → how much extra memory the algorithm needs.

→ Another desirable characteristics of an algorithm is simplicity. Simplicity means generating sequence of instructions which are easy to understand.

→ Generality shows that sometimes it becomes easier to design an algorithm in more general way rather than designing it for a particular sets of inputs.

→ Range of inputs comes in picture when we can execute an algorithm. The design of

the algorithm should be such that it should handle the range of inputs.

If the above factors (time complexity, space complexity, simplicity, generality, range of inputs) are not satisfactory, then we must redesign the algorithm.

6) Coding an Algorithm:

Algorithms are designed to be implemented as computer programs. The validity of the programs are established by testing.

A good algorithm is a result of repeated effort and rework.

while writing a program, it is essential to write an optimized code. Algorithm's optimality should be minimum. This will reduce the burden on computer.

* Rules for writing an Algorithm:

The algorithm is broadly divided into two sections.

Algorithm Heading

It consists of name of algorithm, problem description, input and output.

Algorithm Body.

It consists of logical body of the algorithm by making use of various programming constructs.

1) Algorithm heading consists of keyword algorithm, name of the algorithm and parameter list.

* Syntax:

Algorithm name ($P_1, P_2, \dots, P_n$)

2) In the heading, we should write the following

// Problem Description

// Input

// Output

3) The body of an algorithm is written, in which various programming constructs like if, for, while some assignment statements may be written.

4) The compound statements should be enclosed within { and } brackets.

5) Single line comments are written using // as the beginning of the comment.

6) The ^{name of var, fn, label} identifiers should begin by letter and not by digit. It can be a combination of alphanumeric string.

7) Using assignment operator ($\leftarrow$) an assignment statement can be given.

Variable $\leftarrow$ expression

8) The other types of operators are,

Boolean operators (true or false)

Logical operators (and or not)

Relational operators ($<$, $<=$, $>$, $>=$, $=$, $\neq$),

9) Array indices are stored within square brackets []

10) Inputs and outputs can be given using read and write.
eg: write("This message will be displayed on console");
read(val);

11) Conditional statements such as if-then or if-then-else are written in the following form.

if (Condition) then statement

if (condition) then statement else statement.

12) while statement can be written as,

while (condition) do

{

statement 1

statement 2

statement n.

}

13) The general form for writing for loop is -

for variable ← value 1 to value n do

{

statement 1

statement 2

statement n.

}

14) The repeat-until statement can be written as,

repeat

statement 1

statement 2

⋮

statement n

until (condition)

15) The break statement is used to exit from inner loop. The return statement is used to return control from one point to another.

Examples.

1) Algorithm to check whether a given number is odd or even-

Algorithm evenodd (val)

|| problem Description: This algorithm test whether the given number is even or odd.

|| Input : The number to be tested i.e., val.

|| output : Messages indicating even or odd.

if (val % 2 == 0) then

 write ("Given number is even").

else

 write ("Given number is odd").

2. Algorithm to find factorial of a number.

Algorithm fact(n)

// Problem Description: This algorithm finds the factorial

// Input: The number n of which the factorial is to be calculated.

// Output: Factorial value of the given number n .

if ($n \leq 1$) Then

return 1

else

return $n * \text{fact}(n-1)$.

3. Algorithm to count the sum of n numbers.

Algorithm sum($1, n$)

// problem Description: This algorithm is for finding the sum of given ' n ' numbers.

// Input: 1 to n numbers.

// Output: The sum of n numbers.

result $\leftarrow 0$

for $i = 1$ to n do $i \leftarrow i + 1$

result $\leftarrow$ result + i

return result.

1.2 TIME AND SPACE COMPLEXITY :

* TIME COMPLEXITY :

→ The time complexity of an algorithm is the amount of computer time required by an algorithm to complete a task.

→ It is difficult to compute the time complexity in terms of physically clocked time.

→ Execution Time depends on many factors such as,

- * System Load

- * Number of other programs running

- * Instruction set used

- * Speed of underlying hardware.

→ Time complexity is given in terms of frequency count

Frequency count is a count denoting number of times of execution of statement.

* Types of Time Complexity: The program requires

→ Constant time Complexity $\uparrow$ fixed amount of time for all input values.

→ Linear time complexity → If the input values increases, time complexity will change.

* Example 1: Calculating time complexity
to find the sum of n numbers in
an array

```
for(i=0; i<n; i++)
```

```
{
```

```
    sum = sum + a[i];
```

```
}
```

soln:

Frequency count $\rightarrow$ denotes how many times
the particular statement is
executed.

Statements	Frequency Count
$i=0$	1
$i<n$	$n+1$
$i++$	n
$sum = sum + a[i]$	n
Total	$3n+2$

Complexity is denoted in terms of Oh notation (O)

Neglecting the constants, Time complexity = $O(n)$.

* Example 2 : Addition of Two Matrices.

```
for (i=0; i<n; i++)
```

```
{
```

```
  for (j=0; j<n; j++)
```

```
  {
```

```
    c[i][j] = a[i][j] + b[i][j]
```

```
  }
```

```
}
```

Soln :

Statements	Frequency Count
i=0	1
i<n	n+1
i++	n
j=0	$n \times 1 = n$
j<n	$n(n+1) = n^2 + n$
j++	$n \times n = n^2$
c[i][j] = a[i][j] + b[i][j]	$n \times n = n^2$
Total	$3n^2 + 4n + 2$

Neglecting the constants, Time complexity = $O(n^2)$.

* SPACE COMPLEXITY:

→ Space Complexity indicates how much memory the algorithm needs for execution.

2 factors to compute Space Complexity:

- constant characteristics
- instance characteristics

* space requirement (space complexity) can be calculated by,

$$S(P) = C + S_p$$

where,

$S(P) \rightarrow$ Space Complexity

$C \rightarrow$ Constant / Independent part

$S_p \rightarrow$ dependent part / variable part

Example:

Calculate the Space Complexity for the algorithm for sum of two numbers.

Algorithm Add (a, b)

// Problem Description: This algorithm compute the addition of two numbers

// input : a, b

// output: The sum is $c = a + b$

space complexity, $S(P) = C + S_p$

Variable	[Constant/variable]	Space Required
a	constant	1 word
b	constant	1 word
c	constant	1 word

$$S(P) = C + S_p$$

$$= 3 + 0$$

$$= 3 \Rightarrow \text{constant}$$

Space Complexity $\Rightarrow O(1)$

Example 2: Sum of elements in an array

Algorithm sum(s, n)

{
total = 0;

for $i = 0$ to n do

total = total + s[i];

}

variables	value. (constant (variable))	Space Required
s	dependent variable	depends on 'i' value n units
n	constant	1 unit
total	constant	1 unit
i	constant	1 unit
		$SP) = C + S_p$ $= 3 + n$

$\therefore$ Space Complexity = $O(n)$

1.1.2 ASYMPTOTIC NOTATIONS AND ITS PROPERTIES:

→ To choose the best algorithm, we need to check efficiency of each algorithm.

→ Efficiency can be measured by computing the time complexity of each algorithm.

→ Asymptotic notation is a short way to represent the time complexity.

→ Using asymptotic notations we can give time complexity as,

$$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 \dots 2^n < 3^n < \dots n^n \quad (25)$$

* fastest possible

* slowest possible.

* average time

Definition - Asymptotic Notations:

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value. The various asymptotic notations used are,

→ Big oh ('O') Notation - upper bound

→ Omega ('Ω') Notation - lower bound

→ Theta ('Θ') Notation - average bound.

1) Big oh Notation:

→ Big oh notation is denoted by 'O'.

→ It is a method of representing the upper bound of algorithm's running time.

Definition:

Let $f(n)$ and $g(n)$ be two non-negative functions. A function $f(n)$ is said to be in $O(g(n))$, denoted by $f(n) \in O(g(n))$, if $f(n)$ is bounded above by some positive constant c and some non negative integer n_0 such that

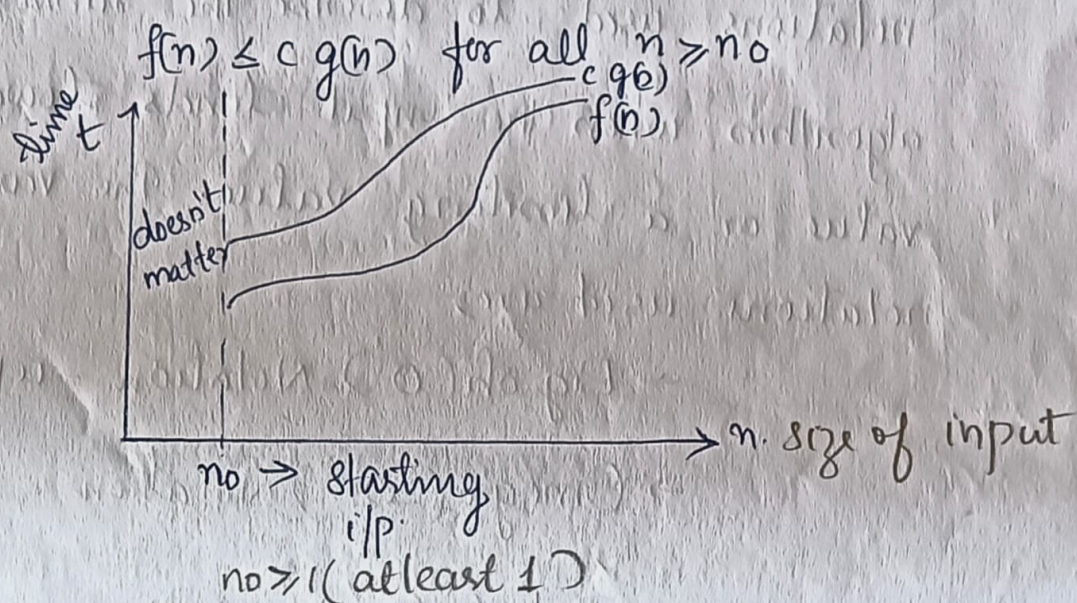


Fig: Big-oh notation: $f(n) \in O(g(n))$

* Example

Consider the functions $f(n) = 2n+2$ and $g(n) = n^2$

Soln: $f(n) \leq c g(n)$

Given, $f(n) = 2n+2$, $g(n) = n^2$ let $c=1$

Let $n=1 \Rightarrow f(1) = 2 \times 1 + 2$

$f(1) = 4$.

$$g(n) = 1^2$$

$$g(n) = 1$$

$$4 > 1 \Rightarrow f(n) > c g(n) \Rightarrow \text{False.}$$

$$\text{Let } n=2 \Rightarrow f(n) = 2 \times 2 + 2$$

$$f(n) = 6$$

$$g(n) = 2^2$$

$$g(n) = 4.$$

$$6 > 4 \Rightarrow f(n) > c g(n) \Rightarrow \text{False.}$$

$$\text{Let } n=3 \Rightarrow f(n) = 2 \times 3 + 2$$

$$f(n) = 8$$

$$g(n) = 3^2$$

$$g(n) = 9.$$

$$8 < 9 \Rightarrow f(n) < c g(n) \Rightarrow \text{True.}$$

Hence we conclude that, for $n > 2$, we obtain

$$\underline{\underline{f(n) < c g(n)}}$$

2. Omega Notation:

→ Omega notation is denoted by ' Ω '.

→ It is a method of representing the lower bound of algorithms running time.

• Definition:

Let $f(n)$ and $g(n)$ be two non-negative functions. A function $f(n)$ is said to be in $\Omega(g(n))$, denoted by $f(n) \in \Omega(g(n))$, if $f(n)$ is bounded below by some positive constant c and some nonnegative integer n_0 such that $f(n) \geq c(g(n))$ for all $n \geq n_0$.

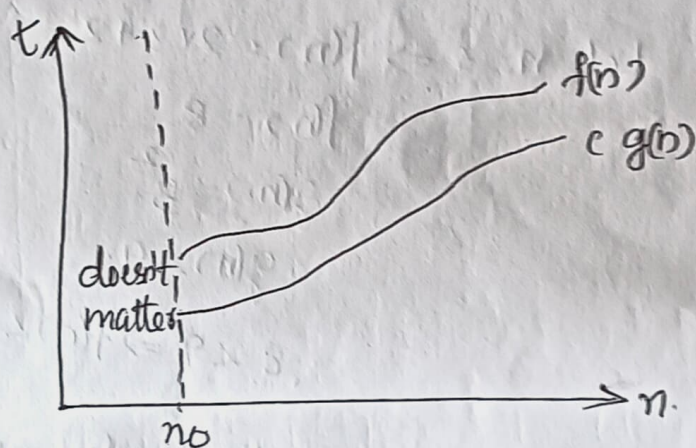


Fig: Big-omega notation: $f(n) \in \Omega(g(n))$

* Example:

Consider the functions $f(n) = 3n + 2$ and $g(n) = n$

Soln:

$$f(n) \geq c(g(n))$$

$$\text{Given, } f(n) = 3n + 2, \quad g(n) = n.$$

Let $c = 4$

$$\text{Let } n=1 \Rightarrow f(n) = 3n+2$$

$$= 3 \times 1 + 2$$

$$f(n) = 5$$

$$g(n) = 1$$

$$5 > 1 \Rightarrow f(n) > c g(n) \Rightarrow \text{True}$$

$$\text{Let } n=2 \Rightarrow f(n) = 3 \times 2 + 2$$

$$f(n) = 8$$

$$g(n) = 2$$

$$8 > 2 \Rightarrow f(n) > c g(n) \Rightarrow \text{True}$$

$$\text{Let } n=3 \Rightarrow f(n) = 9+2$$

$$f(n) = 11$$

$$g(n) = 3$$

$$11 > 3 \Rightarrow f(n) > c g(n) \Rightarrow \text{True}$$

Hence we conclude that, for $n \geq 1$, we obtain

$$f(n) > c g(n)$$

3) Theta Notation:

→ Theta notation is denoted by ' Θ '.

→ The running time of the algorithm is

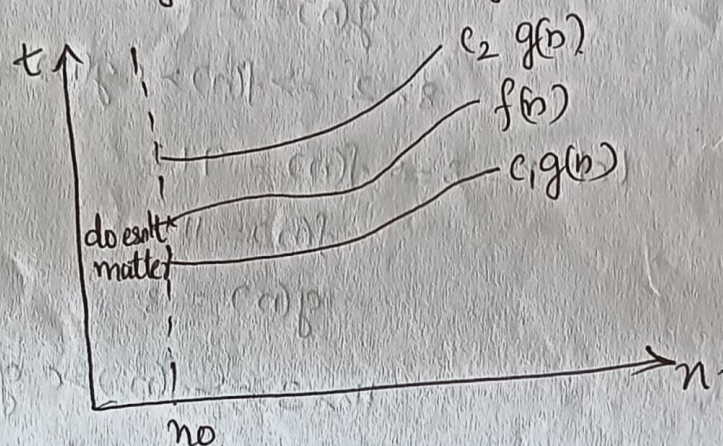
between upper bound and lower bound.

* Definition:

Let $f(n)$ and $g(n)$ be two non-negative

functions. A function $f(n)$ is said to be in $\Theta(g(n))$, denoted by $f(n) \in \Theta(g(n))$, if $f(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large n .
i.e., if there exist some positive constants c_1 and c_2 and some non negative integer n_0 such that

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0.$$



* Example

Consider the functions $f(n) = 3n + 8$ and $g(n) = 7n$

* Soln:

$$c_1 g(n) \leq f(n) \leq c_2 g(n), \quad n \geq n_0$$

Given, $f(n) = 3n + 8$, $g(n) = 7n$.

Let $c_1 = 3$ and $c_2 = 4$

stage 1 - $n = 1 \Rightarrow f(n) = 3 \times 1 + 8 = 11$
 $f(n) = 11$
 $g(n) = 7 \times 1 = 7$

$$1 \times 7 \leq 11 \leq 3 \times 7$$

$$7 \leq 11 \leq 21 \Rightarrow c_2 g(n) \leq f(n) \leq c_1 g(n) \Rightarrow \text{True}$$

Hence, we conclude that, for $n \geq 1$, we obtain

$$c_2 g(n) \leq f(n) \leq c_1 g(n)$$

* Properties of Asymptotic Notations:

* Big oh notation is used for upper bound.

* Big omega notation is used for lower bound.

* Big theta notation is used for average bound.

1. General properties

(i) If $f(n) = O(g(n))$ then $a \times f(n) = O(g(n))$

(ii) If $f(n) = \Omega(g(n))$ then $a \times f(n) = \Omega(g(n))$

(iii) If $f(n) = \Theta(g(n))$ then $|a \times f(n)| = \Theta(g(n))$

eg:- $f(n) = 2n^2 + 5 \Rightarrow O(n^2)$

then, $7 \cdot f(n) = 7(2n^2 + 5)$

$$= 14n^2 + 35$$

$$\Rightarrow O(n^2)$$

2) Reflexive:

i) if $f(n)$ is given then $f(n) = O(f(n))$

eg:- $f(n) = n^2$ then $f(n) = O(n^2)$

ii) If $f(n)$ is given then $f(n) = \Omega(f(n))$.

iii) If $f(n)$ is given then $f(n) = \Theta(f(n))$.

3. Transitive Property

i) If $f(n) = O(g(n))$ and if $g(n) = O(h(n))$ then
 $f(n) = O(h(n))$

ii) If $f(n) = \Theta(g(n))$ and if $g(n) = \Theta(h(n))$ then
 $f(n) = \Theta(h(n))$.

iii) If $f(n) = \Omega(g(n))$ and if $g(n) = \Omega(h(n))$ then
 $f(n) = \Omega(h(n))$.

eg: $f(n) = n$, $g(n) = n^2$, $h(n) = n^3$ then

If $n = O(n^2)$ and $n^2 = O(n^3)$ then $n = O(n^3)$.

4. Symmetry Property

If $f(n) = O(g(n))$ then $g(n) = O(f(n))$

eg:- $f(n) = n^2$, $g(n) = n^2$ then

$$f(n) = O(n^2)$$

$$g(n) = O(n^2)$$

Symmetry property is true only for ' Θ ' notation.

5. Transpose Symmetry

$f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.

eg: if $f(n) = n$, $g(n) = n^2$ - then

$n = O(n^2)$ and $n^2 = \Omega(n)$
 ↙ upper bound ↘ lower bound

6. If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ then
 $f(n) = \Theta(g(n))$

7. $\Theta(f(n) + g(n)) = \max(f(n), g(n))$

* Properties of Order of Growth.

i) If $f_1(n)$ is order of $g_1(n)$ and $f_2(n)$ is order of $g_2(n)$, then $f_1(n) + f_2(n) \in O(\max(g_1(n), g_2(n)))$.

ii) Polynomials of degree $m \in O(n^m)$ i.e., maximum degree is considered.

(iii) $O(1) < O(\log n) < O(n) < O(n^2) < O(2^n)$

(iv) Exponential functions a^n has different orders of growth for different values of a .

→ $1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \dots$
 Measuring the performance of an alg. in relation with the input size 'n' is called order of growth.

Example.

1. show the following equalities are correct.

i) $5n^2 - 6n = O(n^2)$.

ii) $n! = O(n^n)$

iii) $n^3 + 10^6 n^2 = O(n^3)$

(iv) $2n^2 2^n + n \log n = O(n^2 2^n)$

* Soln:-

i) $5n^2 - 6n = O(n^2)$

The given function $f(n) \in O(g(n))$ only when $f(n)$ can be represented as,

$$c_1 \times g(n) \leq f(n) \leq c_2 g(n).$$

$$f(n) = 5n^2 - 6n, \quad g(n) = n^2$$

$$\text{i.e., } n^2 \leq 5n^2 - 6n \leq 5n^2 \text{ for all } n \geq 2$$

$$\text{where, } g(n) = n^2, \quad c_1 = 1, \quad c_2 = 5.$$

Hence the given equality holds true.

(ii) $n! = O(n^n)$

The given function $f(n) \in O(g(n))$ only when $f(n)$ can be represented by,

$$f(n) \leq c * g(n)$$

$$f(n) = n! \text{ and } g(n) = n^n \text{ (let } c=1)$$

$$\text{let } n=1 \Rightarrow f(n) = 1!$$

$$f(n) = 1$$

$$g(n) = 1$$

$$g(n) = 1$$

$$1 \leq 1 \times 1 \Rightarrow 1 \leq 1 \Rightarrow f(n) \leq c g(n) \Rightarrow \text{True}$$

$$(iii) n^3 + 10^6 n^2 = \Theta(n^3)$$

The given function is $f(n) \in \Theta(g(n))$, only

when

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

$$f(n) = n^3 + 10^6 n^2, \quad g(n) = n^3$$

$$\text{let } c_1 = 1, \quad c_2 = 10^7$$

$$n=1 \Rightarrow f(n) = 1^3 + 10^6$$

$$= 1 + 10^6$$

$$g(n) = 1^3$$

$$1 \times 1 \leq 1 + 10^6 \leq 10^7 \times 1 \Rightarrow \text{True}$$

$$(iv) 2^n 2^n + n \log n = \Theta(n^2 2^n)$$

The given function $f(n) \in \Theta(g(n))$, only

when $c_1 g(n) \leq f(n) \leq c_2 g(n)$

$$f(n) = 2n^2 + n \log n, \quad g(n) = n^2$$

$$\text{Let } c_1 = 2, c_2 = 3$$

$$n=1 \Rightarrow f(n) = 2 \times 1 + 2 + 1 \log 1 \quad g(n) = 1^2 \times 2 = 2$$

$$= 4 + 1 \log 1$$

$$2 \times 2 \leq 4 + 0 \leq 3 \times 2$$

$$4 \leq 4 \leq 6 \Rightarrow \text{True}$$

2. Derive a loose bound on the following equation.

$$f(x) = 35x^8 - 22x^7 + 14x^5 - 2x^4 - 4x^2 + x - 15$$

Soln:

Tight bound $\Rightarrow$ if $f(n) \leq c * g(n)$ Then $f(n) \in O(g(n))$

Loose bound $\Rightarrow$ if $f(n) < c * g(n)$ Then $f(n) \in O(g(n))$

Given,

$$f(x) = 35x^8 - 22x^7 + 14x^5 - 2x^4 - 4x^2 + x - 15$$

$$g(x) = x^8, \quad c = 35$$

$$\text{Let } x=1 \Rightarrow f(x) = 35(1)^8 - 22(1)^7 + 14(1)^5 - 2(1)^4 - 4(1)^2 + 1 - 15$$

$$= 35 - 22 + 14 - 2 - 4 + 1 - 15$$

$$f(x) = 7, \quad g(x) = 1$$

$$7 < 35 \times 1 \Rightarrow 7 < 35$$

$$\therefore f(x) < g(x) \text{ for all } x \geq 1.$$

3 State whether the following equality is correct or wrong?

$$5n^3 + 4n = \Omega(n^2)$$

Soln: $f(n) \in \Omega(g(n))$ if $f(n) \geq c * g(n)$ where $n \geq n_0$

$$f(n) = 5n^3 + 4n \quad g(n) = n^2$$

$$\text{let } n=1 \Rightarrow f(n) = 5+4 \quad g(n)=1 \\ = 9$$

$$\text{Let } c=5$$

$$9 \geq 5 \times 1$$

$$9 \geq 5 \Rightarrow \text{True}$$

The given equality is true.

4. Is Big Oh notation transitive? Discuss with an example.

Soln: Yes, Big Oh notation is transitive.

i.e., if $f(n) = O(g(n))$ and if $g(n) = O(h(n))$

then $f(n) = O(h(n))$

eg: Consider, $f(n) = n^3$, $g(n) = 4n^3 + 4n$ and $h(n) = 105n^3 + 4n^2 + 6n$

Then, $f(n) = n^3$, $O(g(n)) = n^3$, $O(h(n)) = n^3$

$$n^3, 4n^3 + 4n, 105n^3 + 4n^2 + 6n = O(n^3)$$

1.13 BEST CASE, WORST CASE AND AVERAGE CASE ANALYSIS:

* eg:- Linear Search:

Algorithm:

Algorithm Linear($A[0 \dots n-1], \text{key}$)

// problem Description: Searching an element in an array.

// Input: The array to be searched and the key element

// Output: Message displaying the position of the element.

for $i=0$ to $n-1$ do

if ($A[i] = \text{key}$) then

return i

* Analysis:

1) Best case Analysis:

→ In the above searching algorithm, the key element is searched from the list of n elements.

→ If the key element is present at first location in the list $A[0 \dots n-1]$ then the algorithm runs for a very short time and we will get the best case time complexity.

Hence in terms of Big-oh notation the time complexity can be denoted as,

$$T(n) = O(1)$$

2) Worst Case Analysis:

Worst case time complexity is a time complexity when algorithm runs for a longest time.

In the above searching algorithm, the element key is searched from the list of n elements.

If the key is present at n th location then the algorithm will run for longest time and we get the worst case complexity.

Hence, in terms of big-oh notation the time complexity can be denoted as,

$$T(n) = O(n)$$

3. Average Case Analysis:

This type of complexity gives information about the behaviour of an algorithm on specific or random input.

Let, 'p' be a probability of getting successful search.

$n \rightarrow$ total no. of elements in the list.

First match of element will occur at i th location.

$\therefore$ probability of first match = P/n .

probability of getting unsuccessful search = $1-p$

Average Case Complexity:

$T(n) =$ Probability of successful search + Probability of unsuccessful search.

$$= \left(1 \cdot \frac{P}{n} + 2 \cdot \frac{P}{n} + \dots + i \cdot \frac{P}{n}\right) + n(1-p)$$

$$= \frac{P}{n} (1 + 2 + \dots + i) + n(1-p)$$

$$= \frac{P}{n} \left[\frac{n(n+1)}{2} \right] + n(1-p)$$

$$\boxed{T(n) = \frac{p(n+1)}{2} + n(1-p)} \quad \text{--- (1)}$$

This is the general formula for computing the average case time complexity.

Substitute $p=0$ and $p=1$ in ①

$$p=0 \Rightarrow T(n) = \frac{0(n+1)}{2} + n(1-0)$$

$$T(n) = n$$

$$p=1 \Rightarrow T(n) = \frac{1(n+1)}{2} + n(1-1)$$

$$T(n) = \frac{(n+1)}{2}$$

$\therefore$ Average Case complexity is $T(n) = O(n)$

* Complexity of linear (sequential) Search :

Best case	Worst case	Average Case
$O(1)$	$O(n)$	$O(n)$

1.1.4 RECURRENCE RELATION:

→ The recurrence equation is an equation that defines a sequence recursively.

→ It is normally in the following form

$$T(n) = T(n-1) + 1 \quad \text{for } n > 0 \quad \text{--- ①}$$

$$T(0) = 0 \quad \text{--- ②}$$

where, eqn ① → recurrence relation

eqn ② → initial condition.

→ The recurrence equation can have infinite number of sequences.

→ The general solution to the recursive function specifies some formula.

→ It is useful for recursive functions

+ Solving Recurrence Equations:

Recurrence relation can be solved by the following methods.

1. Substitution Method.

2. Tree Method.

3. Master's method

4. Iteration Method.

SUBSTITUTION METHOD:

→ In this method, we first make the intelligent guesswork about the solution and then prove by induction that our guess is correct.

* Two types of Substitution

- 1) Forward substitution and
- 2) Backward substitution.

1. Forward Substitution:

→ This method makes use of an initial condition in the initial ~~time~~ term and the value for the next term is generated.

→ This process is continued until some formula is guessed.

Eg:- Consider the recurrence relation $T(n) = T(n-1) + n$ with initial condition $T(0) = 0$

Soln:

$$T(n) = T(n-1) + n \quad \text{--- (1)} \quad T(0) = 0$$

$$\text{if } n=1 \Rightarrow T(1) = T(1-1) + 1$$

$$= T(0) + 1$$

$$= 0 + 1$$

$$T(1) = 1$$

$$\text{If } n=2 \Rightarrow T(2) = T(2-1) + 2 \\ = T(1) + 2 = 1 + 2$$

$$\boxed{T(2) = 3}$$

$$\text{If } n=3 \Rightarrow T(3) = T(3-1) + 3 \\ = T(2) + 3 \\ = 3 + 3$$

$$\boxed{T(3) = 6}$$

By observing above generated values, we can derive a formula,

$$T(n) = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$$

$$\boxed{T(n) = O(n^2)}$$

It is difficult to guess the pattern from the forward substitution.

Hence this method is not very often used.

(ii) Backward Substitution:

In this method, backward values are substituted recursively in order to derive some formula.

Eg:- Consider, a recurrence relation $T(n) = T(n-1) + n$ with initial condition $T(0) = 0$.

Soln:

$$T(n) = T(n-1) + n \quad \text{--- (1)}$$

sub $n = n-1$ in (1)

$$\begin{aligned} T(n-1) &= T(n-1-1) + (n-1) \\ &= T(n-2) + (n-1) \quad \text{--- (2)} \end{aligned}$$

sub (2) in (1) $\Rightarrow T(n) = T(n-2) + (n-1) + n \quad \text{--- (3)}$

sub $n = n-2$ in (1)

$$T(n-2) = T(n-2-1) + (n-2) \quad \text{--- (4)}$$

sub (4) in (3),

$$T(n) = T(n-3) + (n-2) + (n-1) + n \quad \text{--- (5)}$$

$$\text{(1)} \Rightarrow T(n-1) + n \quad \text{---}$$

$$\text{(3)} \Rightarrow T(n) = T(n-2) + (n-1) + n$$

$$\text{(5)} \Rightarrow T(n) = T(n-3) + (n-2) + (n-1) + n$$

$\vdots$

$$T(n) = T(n-k) + (n-k+1) + (n-k+2) + \dots + n.$$

$$\begin{aligned} \text{if } k = n \Rightarrow T(n) &= T(n-k) + (n-n+1) + (n-n+2) + \dots + n \\ &= T(0) + 1 + 2 + \dots + n \quad [T(0) = 0] \\ &= 0 + 1 + 2 + \dots + n. \end{aligned}$$

$$= \frac{n(n+1)}{2}$$

$$T(n) = \frac{n^2}{2} + \frac{n}{2}$$

$$\therefore T(n) \in O(n^2)$$

Example:

1. Solve the following recurrence relation

i) $x(n) = x(n-1) + 5$ for $n > 1$ $x(1) = 0$

ii) $x(n) = 3x(n-1)$ for $n > 1$ $x(1) = 4$

iii) $x(n) = x(n-1) + n$ for $n > 0$ $x(0) = 0$

(iv) $x(n) = x(n/2) + n$ for $n > 1$ $x(1) = 1$ (solve for $n = 2^k$)

(v) $x(n) = x(n/3) + 1$ for $n > 1$ $x(1) = 1$ (solve for $n = 3^k$)

(i) $x(n) = x(n-1) + 5$

Soln: $x(n) = x(n-1) + 5$ ——— (1)

sub $n = n-1$ in (1)

$$x(n-1) = x(n-1-1) + 5$$

$$x(n-1) = x(n-2) + 5$$
 ——— (2)

sub (2) in (1)

$$x(n) = x(n-2) + 5 + 5$$
 (10)

$$x(n) = x(n-2) + 2 \times 5$$
 ——— (3)

sub $n=n-2$ in ①

$$x(n-2) = x(n-2-1) + 5$$

$$x(n-2) = x(n-3) + 5 \quad \text{--- ④}$$

sub ④ in ③

$$x(n) = x(n-3) + 5 + 2 \times 5$$

$$x(n) = x(n-3) + 3 \times 5 \quad \text{--- ⑤}$$

$$\text{①} \Rightarrow x(n) = x(n-1) + 5$$

$$\text{②} \Rightarrow x(n) = x(n-2) + 2 \times 5$$

$$\text{⑤} \Rightarrow x(n) = x(n-3) + 3 \times 5$$

$$\vdots$$

$$x(n) = x(n-k) + k \times 5$$

$$\text{if } k=n-1 \Rightarrow x(n) = x(n-(n-1)) + (n-1) \times 5$$

$$= x(n-n+1) + (n-1) \times 5$$

$$= x(1) + (n-1) \times 5$$

$$= 0 + (n-1) \times 5$$

$$= (n-1) \times 5$$

$$x(n) = 5 \times (n-1)$$

$$\therefore x(n) \in O(n)$$

Solve the recurrence relation,

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ 2T(n/2) + n & \text{otherwise} \end{cases}$$

* Soln:

$$T(n) = 2T(n/2) + n \quad \text{--- (1)}$$

$$n = n/2 \Rightarrow T(n/2) = 2T(n/4) + n/2 \quad \text{--- (2)}$$

$$n = n/4 \Rightarrow T(n/4) = 2T(n/8) + n/4 \quad \text{--- (3)}$$

sub (2) in (1)

$$T(n) = 2[2T(n/4) + n/2] + n$$

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

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

$$T(n) = 2^2 T(n/4) + 2n \quad \text{--- (4)}$$

sub (3) in (4)

$$\Rightarrow T(n/4) = 2^2 [2T(n/8) + n/4] + 2n$$

$$= 2^3 T(n/2^3) + n + 2n$$

$$T(n) = 2^3 T(n/2^3) + 3n \quad \text{--- (5)}$$

The Soln

The sequence continues,

$$T(n) = 2^4 T(n/2^4) + 4n$$

$$T(n) = 2^5 T(n/2^5) + 5n$$

!

$$T(n) = 2^k T(n/2^k) + kn \quad \text{--- (6)}$$

we have to make $T(n/2^k)$ to $T(1)$ to substitute the initial value,

$$\frac{n}{2^k} = 1$$

$$\boxed{n = 2^k}$$

Taking log on both sides.

$$\log n = \log_2 2^k$$

$$\log n = k \cdot \log 2$$

$$\boxed{k = \log n}$$

$$\log_2 2 = 1$$

$$\log_a a = 1$$

Sub $2^k, n$ in (6)

$$T(n) = 2^k T(1) + kn$$

$$= 2^k \cdot 1 + kn \quad [\because n = 2^k, T(1) = 1, k = \log n]$$

$$= n \cdot 1 + \log n \cdot n$$

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

(Complexity)

Lower Bound

→ One approach for solving recurrence relation is using upper bound or lower bound.

* Lower bound:

→ Let $f(n)$ is a running time of algorithm A then $g(n)$ is the lower bound of algorithm A if there exists two constants C and N such that $f(n) \geq C \times g(n)$ for $n > N$.

→ The lower bound algorithm is shown by asymptotic notation called Big Omega (Ω).

* Steps to find the recurrence using lower bound.

(i) guess the solution.

(ii) Use the mathematical induction to find the boundary condition and show that the guess is correct.

* Example:

Solve the following recurrence using lower bound

$$T(n) = 2T\left(\frac{n}{2}\right) + n, \quad n > 1$$

$$\text{Soln: } T(n) = 2T\left(\frac{n}{2}\right) + n \quad \text{--- (1)}$$

we will guess the solution as $O(n \log n)$.

Thus for constant C ,

$$T(n) \leq Cn \log n.$$

Sub in ①

$$\Rightarrow T(n) \leq 2C\left(\frac{n}{2}\right) \log\left(\frac{n}{2}\right) + n$$

$$\leq Cn \log\left(\frac{n}{2}\right) + n.$$

$$\leq Cn \log n - Cn \log 2 + n$$

$$\leq Cn \log n - n(C \log 2 - 1)$$

$$\leq Cn \log n - n(C \times 1 - 1)$$

$$\leq Cn \log n \quad \text{for } C \geq 1$$

$$\text{Thus } T(n) = \Omega(n \log n)$$

1.2 SEARCHING

→ Searching is a technique in which the location of desired element is obtained.

→ The searching technique is based on search key.

→ This key is compared with array elements.

→ If the key and the current element matches, then position of that element is returned.

example: Linear Search

(1)

(2)

(3)

Linear Search

Linear search is a searching technique in which every element is compared with key linearly from first to last element of list.

The time complexity of this algorithm is $O(n)$. The time complexity will increase linearly with this value of n .

eg:

10	20	30	40	50	60	70
----	----	----	----	----	----	----

Enter the element to be searched 50, the element is present in location 4.

Advantages:

1. Simple to implement
2. Doesn't require specific ordering before applying the method.

Disadvantages:

* less efficient.

Time Complexity:

Best case	worst case	Average case.
$O(1)$	$O(n)$	$O(n)$

Algorithm:

Algorithm Linear(key, i, A[])

// problem Description: Searching an element in an array.

// Input: Element to be searched.

// output: Message displaying location of element.

for ($i=0$; $i < n$; $i++$)

if ($A[i] == \text{key}$)

write("Element is found at position", $A[i]$)

else

write("Element is not found")

1.2.2

Binary Search:

→ Binary search is a searching algorithm in which the list of elements are divided into two sublists.

→ The key element is compared with the middle element.

→ If a match is found then, the location of middle element is returned. Otherwise, we search into either of the halves (sublists) depending upon the result produced through the match.

→ This algorithm is considered as an efficient searching algorithm.

→ The prerequisite for this searching technique is that the list should be sorted.

*Example:

Apply Binary search method to search 60 from the list
10, 20, 30, 40, 50, 60, 70.

Soln:

Consider a list of elements stored in array A as

0	1	2	3	4	5	6
10	20	30	40	50	60	70
↑ low						↑ high

Step 1: key element = 60

Step 2: Find the middle element of the array

$$\boxed{\text{mid} = (\text{low} + \text{high}) / 2}$$

$$\text{mid} = (0 + 6) / 2$$

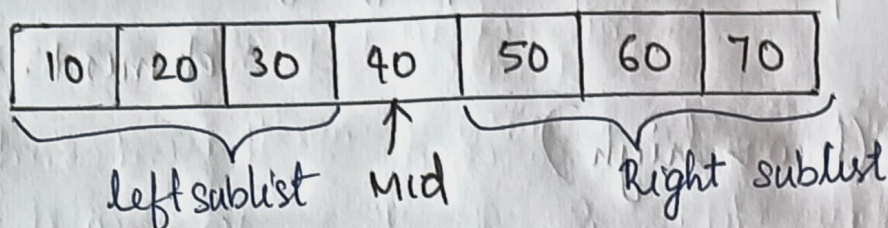
$$\boxed{\text{mid} = 3} \Rightarrow A[3] = 40$$

Compare the middle element with the key element

$$40 < 60$$

i.e., middle element < key element

∴ search the right sublist.



The right sublist is

4	5	6
50	60	70

step 4: Divide the sublist again and find out the mid element.

$$\text{mid} = (\text{low} + \text{high}) / 2$$

$$\text{mid} = (4 + 6) / 2$$

$$\text{mid} = 5 \Rightarrow A[5] = 60$$

step 5: Compare the middle element and the key element

$$60 = 60$$

$\therefore$ middle element = key element

$\therefore$ The search element 60 is present at array index $A[5]$.

Algorithm:

Algorithm Binary($A[0, \dots, n-1], \text{key}$)

// problem Description : Searching an element in the sorted array.

// Input : The array to be searched and the key element.

// output: Message displaying the position of the element.

```
if (low > high)
```

```
    return;
```

```
    mid = (low + high) / 2;
```

```
    if (key == A[mid])
```

```
        return mid;
```

```
    if (key < A[mid])
```

```
        search for x in A[low] to A[mid-1];
```

```
    else
```

```
        search for x in A[mid+1] to A[high];
```

* Advantages:

→ It is an efficient technique.

* Disadvantages:

→ It requires specific ordering before applying the method.

→ It is complex to implement.

* Time Complexity:

Best Case	Worst Case	Average Case
$O(1)$	$O(\log n)$	$O(\log n)$

* Best case:

→ The best case of binary search is, if the element to be searched is present exactly at the middle of the array.

→ i.e. the element is found in the first comparison itself.

∴ Best case time complexity is $O(1)$.

* Worst and Average case:

→ After one comparison the array is divided into 2 sublists.

∴ Worst case time complexity is given by

$$\Rightarrow T(n) = T(n/2) + 1$$

↑
time required
to compare left or
right sublist.

↑ One comparison made
with middle element

$$\therefore T(n) = 1 + \log_2 n$$

∴ Worst, Average case time complexities can be given by, $T(n) = O(\log n)$

1.2.3 INTERPOLATION SEARCH:

→ Interpolation search is an improved variant of binary search.

→ It is an algorithm for searching a key in an array that has been ordered by numerical values.

→ Binary search always goes to the middle element to check.

→ Interpolation search may go to different locations according to the value of the key being searched.

→ The list should be sorted, similar to binary search.

→ The formula used for finding the mid element is,

$$\text{mid} = \text{low} + \left[\frac{(\text{high} - \text{low}) * (\text{key} - A[\text{low}])}{A[\text{high}] - A[\text{low}]} \right]$$

Example:

Apply Interpolation search method to search 22 from the list 10, 12, 14, 16, 18, 20, 22, 24.

* Soln:

step 1: Consider an array of numbers as

0	1	2	3	4	5	6	7
10	12	14	16	18	20	22	24

step 2:

$n=8$, $key=22$

$low=0$, $high=7$

0	1	2	3	4	5	6	7
10	12	14	16	18	20	22	24

low

high

step 3:

$$mid = low + \left[\frac{(high - low) * (key - A[low])}{A[high] - A[low]} \right]$$

$$= 0 + \left[\frac{(7 - 0) * (22 - 10)}{24 - 10} \right]$$

$$= \frac{7 * 12}{14}$$

$$mid = 6$$

Step 4: Check if $A[\text{mid}] = \text{key}$ element

$A[6] = 22 \Rightarrow \text{key element found.}$

Algorithm:

Algorithm Interpolation($A[0 \dots n-1], \text{key}$)

// problem description: searching an element in the sorted array.

// Input: The array to be searched and key element

// output: Message displaying the position of the element.

$\text{low} \leftarrow 0, \text{high} \leftarrow n-1$

$$\text{mid} = \text{low} + \left\lfloor \frac{(\text{high} - \text{low}) * (\text{key} - A[\text{low}])}{A[\text{high}] - A[\text{low}]} \right\rfloor$$

if ($A[\text{mid}] == \text{key}$)

write("Element Found");

else if ($\text{key} > A[\text{mid}]$)

$\text{low} \leftarrow \text{mid} + 1$

else $\text{high} \leftarrow \text{mid} - 1$

Time Complexity

Best case	worst case	Average case.
$O(1)$	$O(n)$	$O(\log(\log n))$

Pattern Search

In software design or in system design-text processing is a vital activity.
 → Text processing is normally done

in compilation of program.

→ Finding the occurrences of all strings in the text is generally known as string matching. These occurrences are called as pattern.

→ String matching algorithms (pattern matching algorithms) are used for text processing.

Let, $\text{Text}(T) \rightarrow t_0, t_1, \dots, t_{n-1}$.

$\text{pattern}(P) \rightarrow p_0, p_1, \dots, p_{m-1}$.

$t_0 \dots t_i \dots t_{i+j} \dots t_{i+m-1} \dots t_{n-1} \leftarrow \text{Text}$

$p_0 \dots p_j \dots p_{m-1} \leftarrow \text{pattern}$

matched pattern
found in text.

Pattern (string) Matching Algorithms

1. The Naive string - matching Algo.
2. Rabin-Karp Algo.
3. Knuth-Morris-Pratt Algo.

1. The Naive String matching Algo.

→ Simplest pattern matching algo. that works using Brute force approach, which is a straight forward approach to solve a problem.

→ This algo performs a checking at all positions in the text, from 0 to $n-m$, whether an occurrence of the pattern starts there or not.

Then for each attempt, it shifts the pattern exactly by one position to the right.

If the pattern match is found, it returns the starting position otherwise the matching process is continued by shifting one character to the right.

If there are no matches at all in the text for the given pattern, we have to do n comparisons.


```

for (i ← 0 to n-m) do
{
    if (P[1..m] = T[i+1..i+m]) then
        print("pattern found at index", i)
    else
        return -1
}

```

Time Complexity

Best case	worst case	Avg case
$O(m)$	$O(mn)$	$O(n)$

2. Rabin-Karp Algorithm

→ Developed by Michael O. Rabin and Richard M. Karp in the year 1987.

→ This alg uses hashing to find an exact match of a pattern string in the text.

* Valid hit

If the hash value matches, it searches if all characters of pattern (P) and Text (T) matches. If a match is there, it is said to be valid hit.

spurious hit

If the hash value matches, it searches if all the characters of pattern (P) and text (T) match. If there is no match, it is said to be spurious hit.

eg:-

T = 2359023141526739921

P = 31415

Soln

let 'q' be a random prime number.

$$q = 13$$

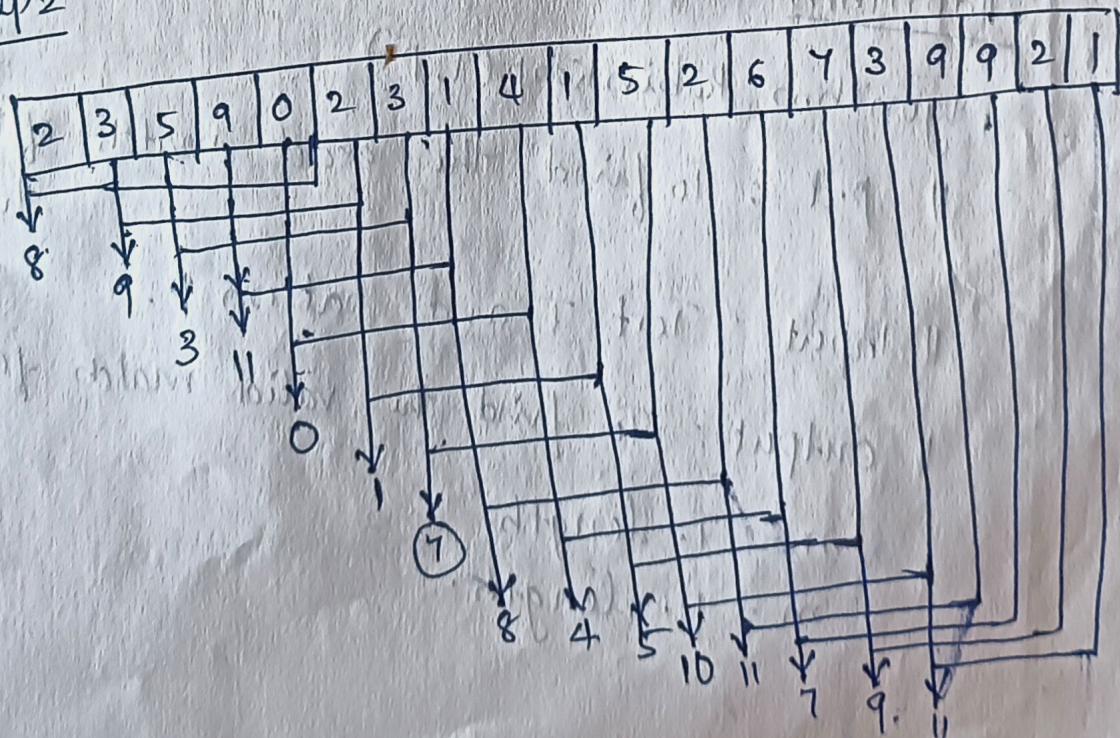
step 1:

$$\text{Hash} = P \bmod q$$

$$= 31415 \bmod 13$$

$$= 7$$

step 2



Handwritten calculations on the right margin:

$$13 \overline{) 2415}$$

$$\begin{array}{r} 185 \\ 13 \overline{) 2415} \\ \underline{26} \\ 54 \\ \underline{52} \\ 21 \\ \underline{13} \\ 85 \\ \underline{78} \\ 7 \end{array}$$

$$23590 \bmod 13 = 8 \neq 7$$

$$35902 \bmod 13 = 9 \neq 7$$

$$59023 \bmod 13 = 3 \neq 7$$

$$90231 \quad " \quad = 11 \neq 7$$

$$02314 \quad " \quad = 0 \neq 7$$

$$23141 \quad " \quad = 1 \neq 7$$

$$31415 \quad " \quad = 7 = 7 \Rightarrow \text{valid bit / valid match}$$

$$14152 \quad " \quad = 8 \neq 7$$

$$41526 \quad " \quad = 4 \neq 7$$

$$15267 \quad " \quad = 5 \neq 7$$

$$52673 \quad " \quad = 10 \neq 7$$

$$26739 \quad " \quad = 11 \neq 7$$

$$p \neq 67399 \quad " \quad = 7 = 7 \Rightarrow \text{spurious match / spurious bit}$$

$$73992 \quad " \quad = 9 \neq 7$$

$$39921 \quad " \quad = 11 \neq 7$$

Algorithm

Algorithm RabinKarp(T, P)

// p.d : To find the exact match of the pattern from the text.

// Input : Text T and pattern P.

// output : To find the valid match for pattern P

$n = T.length$

$m = P.length$

$$h^P = \text{Hash}(P[0 \dots m-1]) \quad // p \bmod q$$

$$h^T = \text{Hash}(T[s \dots s+m-1]) \quad // t \bmod q$$

for $s=0$ to $n-m$.

if $(h^P == h^T)$

if $(P[0 \dots m-1] == T[s \dots s+m-1])$

print ("pattern found with shift", s)

if $(s < n-m)$

$$h^T = \text{Hash}(T[s+1 \dots s+m])$$

* Time Complexity

Best case.	worst case	Average case.
$O(m+n)$	$O(mn)$	$O(m+n)$

3. Knuth - Morris - Pratt Algorithm (KMP - Algo)

KMP algorithm is named after the

scientists Knuth, Morris and Pratt.

Knuth - Morris - Pratt algorithm avoids the repeated comparisons of characters which the other pattern matching algorithms followed.

The basic idea behind this algorithm is to build a prefix-table.

Prefix table is also said to be array table for failure function - table or Π -Table.

or LPS (Longest prefix which is also a suffix)

Prefix-table is built using the prefix and suffix info of pattern.

eg:- Text: abc x abcd ab x abcd abcd abcy.

Pattern: abcd abcy.

Step 1: Construct prefix-table or Π table

0	1	2	3	4	5	6	7
a	b	c	d	a	b	c	y
0	0	0	0	1	2	3	0

Step 2: Start matching search for pattern against the text with the help of prefix-table.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
a	b	c	x	a	b	c	d	a	b	x	a	b	c	d	a	b	c	d	a	b	c	y
✓	✓	✓	x																			
a	b	c	d	a	b	c	y															

pattern[3] ≠ Text[3]

∴ position = text-index-of-unmatched-character - prefix-table[pattern-index - 1]

[1-3] = 3 - prefix-table[3-1]

[2] = 3 - prefix-table[2]

= 3 - 0

= 3

∴, shift the pattern starting from index 3.

Step 3

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
a	b	c	x	a	b	c	d	a	b	x	a	b	c	d	a	b	c	d	a	b	c	y
			x																			
			a	b	c	d	a	b	c	y												

pattern[0] ≠ Text[3]

Since there is a mismatch in the first character itself, we simply shift the pattern by one position.

Step 4.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
a	b	c	x	a	b	c	d	a	b	x	a	b	c	d	a	b	c	d	a	b	c	y	
					✓	✓	✓	✓	✓	✓	x												
				a	b	c	d	a	b	c	y												

pattern[6] ≠ Text[10]

∴ position = 10 - prefix table [6-1]

[5] = 10 - prefix table [5]

= 10 - 2

= 8

i.e., shift the pattern starting from index 8.

Step 5.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
a	b	c	x	a	b	c	d	a	b	x	a	b	c	d	a	b	c	d	a	b	c	y
								✓	✓	x								x				
								a	b	c	d	a	b	c	y							

pattern[2] ≠ Text[10]

∴ position = 10 - prefix table [2-1]

= 10 - prefix table [1]

= 10 - 0

= 10

ii, shift the pattern starting from index 10.

steps:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
a	b	c	x	a	b	c	d	a	b	x	a	b	c	d	a	b	c	d	a	b	c	y
										x												
											a	b	c	d	a	b	c	y				
											0	1	2	3	4	5	6	7				

Pattern[0] $\neq$ Text[10]

Since there is a mismatch in the first position
ii, we simply shift the pattern by one position

step 7

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
a	b	c	x	a	b	c	d	a	b	x	a	b	c	d	a	b	c	d	a	b	c	y
												✓	✓	✓	✓	✓	✓	✓	x			
												a	b	c	d	a	b	c	y			
												0	1	2	3	4	5	6	7			

Pattern[7] $\neq$ Text[18]

$$\begin{aligned} \text{shift} &= \text{position} - \text{prefixtable}[7-1] \\ &= 18 - \text{prefixtable}[6] \\ &= 18 - 3 \end{aligned}$$

$= 15$ $\therefore$ Shift the pattern starting

from index 15

steps: in most plants growth is first in

Thus we found a pattern match at starting position `Text[15]`.

∴ Algorithm - prefix table.

Algorithm $\text{prefix}(\text{char}, p[\text{size}])$

11 p-d : This alg computes prefextable.

11 Input: pattern P.

11 output: prefix table for the given pattern.

prefix-table[0] $\leftarrow$ 0

for ($q \leftarrow 1$ to m) do

```
while (k > 0 and p[k] != p[q])
```

$$k \leftarrow \text{prefix-table}[k-1]$$

if $P[k] = P[q]$ then

$$k \leftarrow k + 1$$

prefix-table[9] $\in K$.

return prefix - table;

$$q \in \text{pattern}$$

$p[g] = \text{array}$
↓
table. patterns

Algorithm : KMP

Algorithm kmp (char t[size], char p[size])

// problem Description: To find the exact match of pattern from the text.

// Input: The array of text t and pattern p.

// Output: The starting location at which the matched pattern is found.

j ← 0;

n ← strlen(t);

m ← strlen(p);

Prefix-table = Create-prefix-table(p);

for (i = 0 to i < n) do

{ while (j > 0 and P[j] != t[i]) do

j ← Prefix-table[j-1]

if (P[j] == t[i]) then

j++;

if (j == m) then

{ write ("pattern is present in the text")

write (i - m + 1)

j ← Prefix-table[j-1];

}

}

Best case

$O(n)$

Time Complexity

worst case

$O(m+n)$

Average case

$O(m+n)$

SORTING

Sorting is a technique by which the elements are arranged in some particular order. i.e., Ascending order or descending order.

Sorting is useful for arranging the data in desired order. After sorting, the element can be located easily.

Applications of Sorting:

- (i) Sorting is useful in database applications for arranging the data in desired order.
- (ii) In dictionary like applications, data are arranged in sorted order.
- (iii) Sorting is required for searching the elements from the list of elements.
- (iv) Sorting is required to check the uniqueness of the element.
- (v) It is used for finding the ^{closest} pair from the list of elements the sorting is required.

Types of sorting:

- 1) Insertion sort-
- 2) Merge sort-
- 3) Quick sort-

- 4) Bubble sort-
- 5) Selection sort
- 6) Heap sort
- 7) Bucket sort
- 8) Shell sort.

INSERTION SORT :

→ In this method, the elements are inserted at their appropriate place.

→ Insertion sort method sorts a list of elements by inserting each successive elements in the previously sorted sublist.

Example

Sort the following elements using Insertion sort

0	1	2	3	4	5	6
30	70	20	50	40	10	60

Soln:

0	1	2	3	4	5	6
30	70	20	50	40	10	60

sorted zone

Unsorted zone.

30	70
----	----

sorted zone.

20	50	40	10	60
----	----	----	----	----

Unsorted zone

0	1	2
20	30	70

Sorted Zone

3	4	5	6
50	40	10	60

Unsorted zone

0	1	2	3
20	30	50	70

Sorted zone

4	5	6
40	10	60

Unsorted zone

0	1	2	3	4
20	30	40	50	70

Sorted zone

5	6
10	60

Unsorted zone

0	1	2	3	4	5
10	20	30	40	50	70

Sorted zone

6
60

Unsorted zone

0	1	2	3	4	5	6
10	20	30	40	50	60	70

Sorted elements

Algorithm:

Algorithm Insertion($A[0 \dots n-1]$)

// problem description: To sort the given elements using Insertion Sort

// Input: An array of n elements

// output: Sorted array $A[0 \dots n-1]$ in ascending order.


```

for i ← 1 to n-1 do
{
    temp ← A[i]
    j ← i-1
    while (j >= 0 and A[j] > temp) do
    {
        A[j+1] ← A[j]
        j ← j-1
    }
    A[j+1] ← temp
}

```

* Analysis

Best case time complexity :- When an array of elements is almost sorted $\Rightarrow O(n)$

Average case time complexity :- If an array is randomly distributed $\Rightarrow O(n^2)$

Worst case time complexity :- If the list of elements is in descending order and if we want to sort the elements in ascending order $\Rightarrow O(n^2)$

Advantages of Insertion Sort:

- (i) Simple to implement
- (ii) Efficient to sort small number of elements.

(iii) More efficient than other $O(n^2)$ algorithms such as selection sort or bubble sort.

(iv) ^{This algorithm} ~~More efficient~~ is stable

(v) It requires a constant amount i.e., $O(1)$ of extra memory space. It is also said to be in-place algorithm. i.e., input is overwritten by output and to execute the sorting method, it does not require any additional space.

HEAP SORT

Heap:

Heap is a complete binary tree or almost complete binary tree in which every parent node is greater than or lesser than its child nodes.

Types of Heaps:

1) Max Heap.

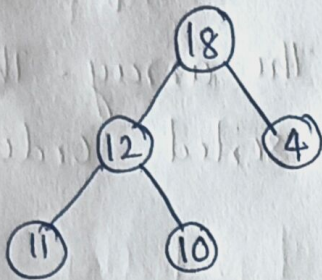
2) Min Heap.

1. Max Heap:

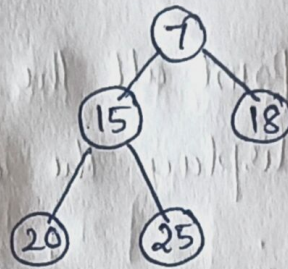
Max heap is a heap structure in which each node is greater than or equal to its child nodes.

2) Min Heap:

Min heap is a heap structure in which each node is lesser than or equal to its child nodes.



Max heap



Min heap.

It works in two stages

1) Heap Construction:

First construct a heap for given numbers.

2) Deletion of Maximum key:

Delete root key always for $(n-1)$ times for remaining heap.

Hence, we will get the elements in decreasing order.

* Steps for Heap sort:

step 1: Construct max heap from given set of elements

step 2: Swap the root key with the last node key of the heap.

step 3: Delete the last node and store this key at the end of the array in which heap is stored.

step 4: Heapify the remaining heap structure.

step 5: Repeat step 2 to 4 until the last remaining node.

step 6: Delete the last node and store it in the array of heap.

step 7: Print all the elements of the array. This will display the elements in sorted order.

Example:

Sort the elements of a given array in ascending order using heap sort. The elements are 15, 20, 7, 9, 30

Soln:

1) Heap Construction:

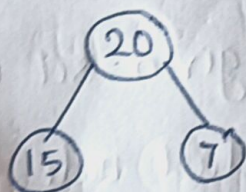
step 1: Insert 15



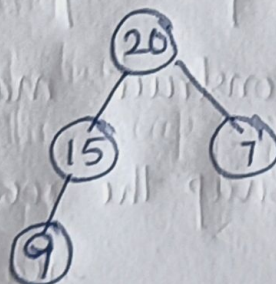
step 2: Insert 20



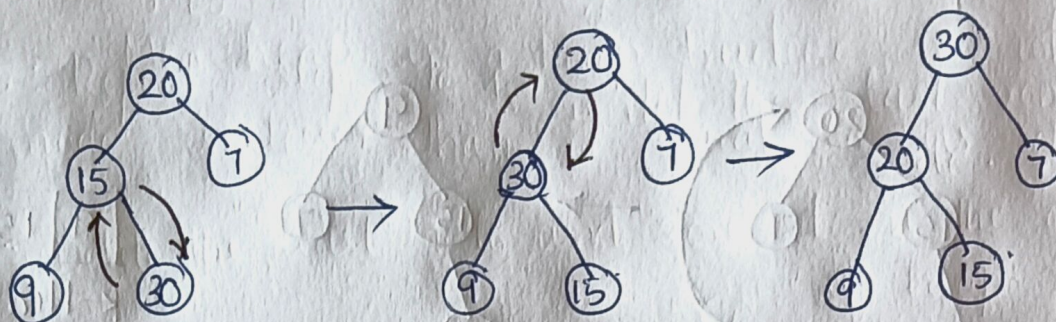
step 3: Insert 7



step 4: Insert 9



step 5: Insert 30

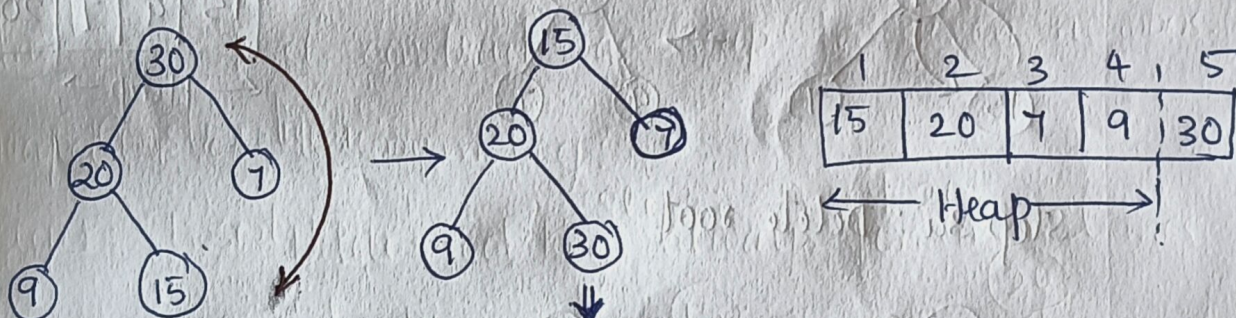


* Heap elements sorted in array

1	2	3	4	5
30	20	7	9	15

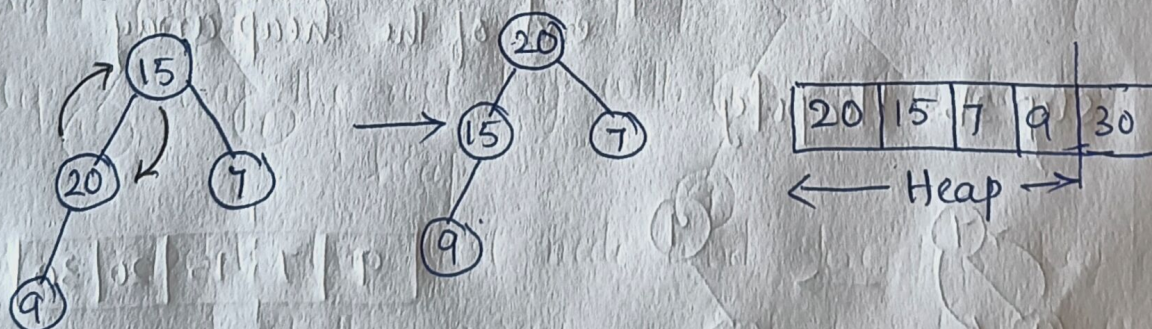
Delete root node and heapify repeatedly

step 1a: Delete root

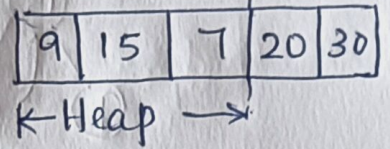
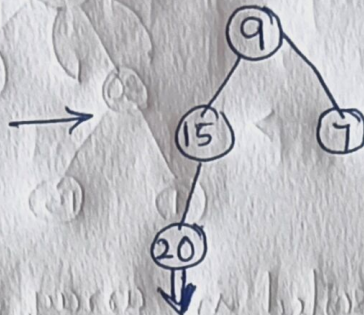
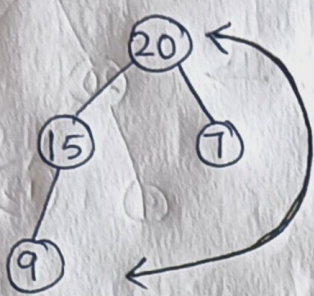


Delete 30 and place it at the end of the array

step 1b: Heapify

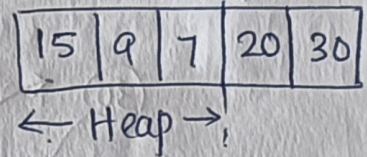
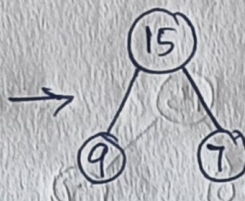
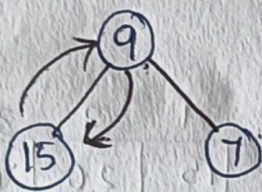


step 2a: Delete root

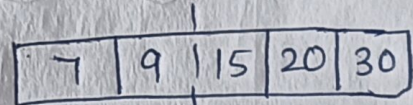
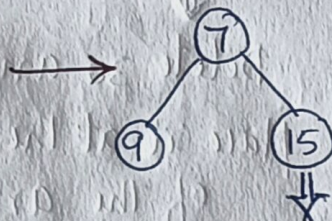
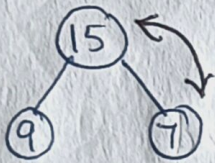


Delete 20 and
place it at the
end of the heap
array

step 2b: Heapify

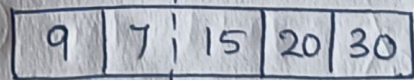
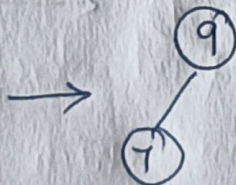


step 3a: Delete root

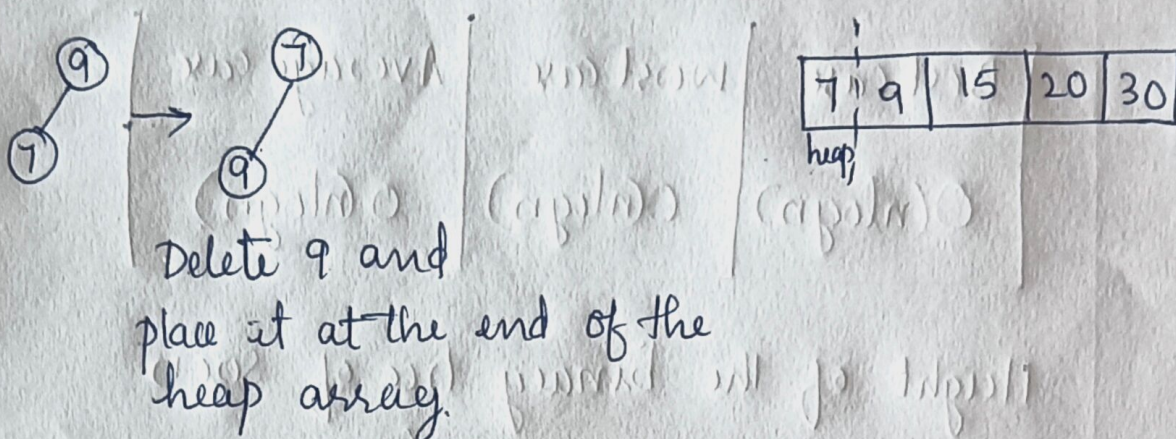


Delete 15 and
place it at the
end of the heap array

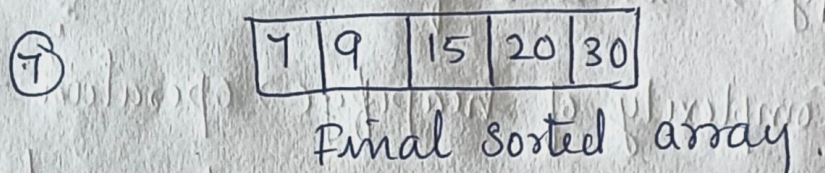
step 3b: Heapify



Step 4a : Delete root



Step 4b : Heapify.



* Algorithm:

Algorithm Heapsort(int arr[], int n)

// problem Description: Sort the elements using heap sort

// Input : An array of elements

// output : Sorted array.

for $i = n/2 - 1$ to $i \geq 1$ do

 heapify(arr, n, i);

for $i = n - 1$ to $i \geq 1$ do

{

 swap-elements(&arr[1], &arr[i])

 heapify(arr, i, 1);

}

Time Complexity :

Best case	Worst case	Average case
$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Height of the binary tree of size n
 $= \log n$

Heapify() is done from root to bottom.

$\therefore$ Complexity of heapify() operation $= O(\log n)$

heapify() method is called $n-1$ times

$\therefore$ Total complexity $= O(n \log n)$