



Swami Keshvanand Institute of Technology, Management & Gramothan,

Ramnagar, Jagatpura, Jaipur-302017, INDIA

Approved by AICTE, Ministry of HRD, Government of India

Recognized by UGC under Section 2(f) of the UGC Act, 1956

Tel. : +91-0141- 5160400 Fax: +91-0141-2759555

E-mail: info@skit.ac.in Web: www.skit.ac.in

A
Course File
on
(Analysis of Algorithms: 5CS4-05)

Programme: B. Tech

Semester: V

Session: 2021-22

Loveleen Kumar
(Assistant Professor)
CS



**Swami Keshvanand Institute of Technology, Management & Gramothan,
Ramnagar, Jagatpura, Jaipur-302017, INDIA**

Approved by AICTE, Ministry of HRD, Government of India

Recognized by UGC under Section 2(f) of the UGC Act, 1956

Tel. : +91-0141- 5160400 Fax: +91-0141-2759555

E-mail: info@skit.ac.in Web: www.skit.ac.in

Vision and Mission of Institute

Vision:“To promote higher learning in technology and industrial research to make our country a global player.”

Mission:“To promote quality education, training and research in the field of engineering by establishing effective interface with industry and to encourage the faculty to undertake industry sponsored projects for the students.”



Vision of CSE Department

Vision of CSE Department is to:

V1: Produce quality computer engineers trained in the latest tools and technologies.

V2: Be a leading department in the region and country by imparting in-depth knowledge to the students in an emerging technologies in computer science & engineering.

Mission of CSE Department

Mission of CSE Department is to:

Delivering resources in IT enable domain through:

M1: Effective Industry interaction and project based learning.

M2: Motivating our students for employability, entrepreneurship, research and higher education.

M3: Providing excellent engineering skills in a state-of-the-art infrastructure.



Syllabus

III Year-V Semester: B.Tech. Computer Science and Engineering

5CS4-05: Analysis of Algorithms

Credit: 3 Max. Marks: 150(IA:30, ETE:120)

SN	Contents	Hours
1	Introduction: Objective, scope and outcome of the course	01
2	Background: Review of Algorithm, Complexity Order Notations: definitions and calculating complexity. Divide And Conquer Method: Binary Search, Merge Sort, Quick sort and Strassen's matrix multiplication algorithms.	06
3	Greedy Method: Knapsack Problem, Job Sequencing, Optimal Merge Patterns and Minimal Spanning Trees. Dynamic Programming: Matrix Chain Multiplication. Longest Common Subsequence and 0/1 Knapsack Problem.	10
4	Branch And Bound: Traveling Salesman Problem and Lower Bound Theory. Backtracking Algorithms and queens problem. Pattern Matching Algorithms: Naïve and Rabin Karp string matching algorithms, KMP Matcher and Boyer Moore Algorithms.	08
5	Assignment Problems: Formulation of Assignment and Quadratic Assignment Problem. Randomized Algorithms- Las Vegas algorithms, Monte Carlo algorithms, randomized algorithm for Min-Cut, randomized algorithm for 2- SAT. Problem definition of Multicommodity flow, Flow shop scheduling and Network capacity assignment problems	08
6	Problem Classes Np, Np-Hard And Np-Complete: Definitions of P, NP-Hard and NP-Complete Problems. Decision Problems. Cook's Theorem. Proving NP- Complete Problems - Satisfiability problem and Vertex Cover Problem. Approximation Algorithms for Vertex Cover and Set Cover Problem.	08



Prerequisites

Students should have basic knowledge of:-

- Programming languages like C or C++ or Java or Python etc.
- Data Structure
- Mathematics.

Students should have basic knowledge of programming and mathematics. The student should know data structure very well. Moreover, it is preferred if the students have basic understanding of Formal Language and Automata Theory. A version of what is normally called [discrete mathematics](#), combined with first-year (university) level calculus are the primary requirements to understanding many (basic) algorithms and their analysis.

Specialized or advanced algorithms can require additional or advanced mathematical background, such as in statistics / probability (scientific and financial programming), abstract algebra, and number theory (i.e. for cryptography).

it is comfort and fluency with the mathematical formalism. Learn basic set terminology and the corresponding formalism.

The analysis of algorithms, especially in the context of complexity theory in which you study the underlying computational problem (if you're attempting to do something more substantial than "Big-Oh" notation), does require a significant investment in time into graph theory and abstract algebra, all in addition to a huge dose of innate cleverness.



Text And Reference Book

Text Book:

1. Cormen, Leiserson, Rivest: "Introduction to Algorithm", Second Edition - PHI
2. Gilles Brassard, Paul Bratley: "Fundamental of Algorithmics" – PHI
3. "Data Structures and Their Algorithms" by Harry R Lewis and Larry Denenberg
4. Computer Algorithms by Horowitz E., Sahni S., Rajasekaran S., Galgotia Publications, 2001

Reference Books:

1. Jon Kleinberg, Eva Tardos: "Algorithm Design", Pearson Edition
2. Harsh Bhasin: "Algorithm Design and Analysis", Oxford Higher Education.
3. "Data Structures and Algorithms Made Easy: Data Structures and Algorithmic Puzzles" by Narasimha Karumanchi
4. "The Algorithm Design Manual" by Steven S Skiena



Course Outcomes Competency Level

Course Code	Course name	Course Outcomes	Bloom Level	PO Indicators
5CS4-05	Analysis and Design Of Algorithm	CO1: describe asymptotic analysis concepts and use them to evaluate the time-complexity of different algorithms	1,2	1.1,1.2,1.5,3.1,3.9 PSO 1.1,1.2
		CO2:explain, apply and analyze the divide and conquer, greedy method and dynamic programming techniques to solve various engineering problems	3,6	1.1,1.2,1.4,1.5,2.1,2.3,2.4, 2.5,2.6,2.7, 3.1,3.6,3.7, 3.8,3.9,5.2,5.3,5.5,5.6 PSO 1.1,1.2
		CO3:discuss and use Branch and Bound, and pattern-matching algorithms	3,4,6	2.3,2.4,2.5,2.7,2.8 3.1,3.6,3.7,3.8,3.9, 5.2,5.3,5.5,5.6 PSO 1.1,1.2
		CO4:discuss randomized algorithms for min-cut and 2-SAT problem	5,6	1.1,1.2,2.3,2.4 PSO 1.1,1.2
		CO5:understand the concepts of NP-Hard and NP-Complete problems	2,5	PSO 1.1,1.2



**Swami Keshvanand Institute of Technology, Management & Gramothan,
Ramnagar, Jagatpura, Jaipur-302017, INDIA**

Approved by AICTE, Ministry of HRD, Government of India

Recognized by UGC under Section 2(f) of the UGC Act, 1956

Tel. : +91-0141- 5160400 Fax: +91-0141-2759555

E-mail: info@skit.ac.in Web: www.skit.ac.in

CO PO\PSO Mapping

	PO 1	PO 2	PO 3	PO 4	PO 5	PO6	PO7	PO8	PO9	PO 10	PO 11	PO 12	PSO 1	PSO 2	PSO 3
CO1	2		1										3		
CO2	3	2	2		3								3		
CO3	2	2			3								3		
CO4	2	1											3		
CO5													3		



QUIZ

1. Consider a linked list of n elements. What is the time taken to insert an element after an element pointed by some pointer?

- A. $O(1)$ B. $O(n)$ C. $O(\log_2 n)$ D. $O(n \log_2 n)$

2. An algorithm is made up of two independent time complexities $f(n)$ and $g(n)$. Then the complexities of the algorithm is in the order of

- A. $f(n) \times g(n)$ B. $\max(f(n), g(n))$ C. $\min(f(n), g(n))$ D. $f(n) + g(n)$

3. Two main measures for the efficiency of an algorithm are

- A. Processor and memory C. Time and space
B. Complexity and capacity D. Data and space

4. The total number of comparisons in a bubble sort is

- A. $O(\log n)$ B. $O(n \log n)$ C. $O(n)$ D. None

5. The space factor when determining the efficiency of algorithm is measured by

- A. Counting the maximum memory needed by the algorithm
B. Counting the minimum memory needed by the algorithm
C. Counting the average memory needed by the algorithm
D. Counting the maximum disk space needed by the algorithm

6. What's the minimum time needed to merge all n

Files ? If $n=5$ and $(F_1, F_2, F_3, F_4, F_5) = (20, 30, 10, 5, 30)$.

- a. 140 b. 270 c. 205 d. none

7. The optimal solution to the knapsack instance $n=3, M=15$ $(P_1, P_2, P_3) = (25, 24, 15)$

$(W_1, W_2, W_3) = (18, 15, 10)$ is

- a) 28.2 b) 31.0 c) 31.5 d) 41.5

8. The solution set $X = (X_1, X_2, X_3)$ for the problem given in Q.7 is

- a) $(1/15, 2/15, 0)$ b) $(0, 2/3, 1)$ c) $(0, 1, 1/2)$
d) None of these

9. Total number of spanning tree in a complete graph with 5 nodes are

- a) 5^2 b) 5^3 c) 10 d) 100

10. The concept of order BigO is important because

- A. It can be used to decide the best algorithm that solves a given problem
B. It determines the maximum size of a problem that can be solved in a given given amount of time
C. It is the lower bound of the growth rate of algorithm
D. Both A and B

11. The time factor when determining the efficiency of algorithm is measured by

- A. Counting microseconds
B. Counting the number of key operations
C. Counting the number of statements
D. Counting the kilobytes of algorithm



12. In worst case Quick Sort has order

- A. $O(n \log n)$ B. $O(n^2/2)$ C. $O(\log n)$ D. $O(n^2/4)$

13. The sorting technique where array to be sorted is partitioned again and again in such a way that all elements less than or equal to partitioning element appear before it and those which are greater appear after it, is called

- A. Merge sort B. Quick sort C. Selection sort
D. None of these

14. The best average behaviour is shown by

- A. Quick Sort B. Merge Sort C. Insertion Sort
D. Heap Sort

15) Which of the following algorithm have same time complexity in Best, average and worst case:

- a) Quick sort b) Merge sort c) Binary search d) all

16) Suppose the input array $A[1...n]$ is already in sorted order (increasing or decreasing) then it is _____ case situation for QUICKSORT algorithm

- a) Best b) worst c) average d) may be best or worst

17) Which one of the following algorithm design techniques is used in Strassen's matrix multiplication algorithm?

- (a) Dynamic programming (c) Greedy method
(b) Backtracking approach (d) D&C strategy

18) Strassen's algorithm is able to perform matrix multiplication in time _____.

- (a) $O(n^{2.61})$ (b) $O(n^{2.71})$ (c) $O(n^{2.81})$ (d) $O(n^3)$

19) Strassen's matrix multiplication algorithm ($C = AB$), if the matrices A and B are not of type $2^n \times 2^n$, the missing rows and columns are filled with _____.

- (a) 0's (b) 1's (c) -1's (d) 2's

20) Strassen's matrix multiplication algorithm ($C = AB$), the matrix C can be computed using only 7 block multiplications and 18 block additions or subtractions. How many additions and how many subtractions are there out of 18?

- (a) 9 and 9 (b) 6 and 12 (c) 12 and 6 (d) none

21) Which of following is Pattern Matching Algorithm.

- a) Naive Matching b) RabinKarp c) KMP matching d) All of above

22) "This is a text" Pattern=TEXT. Find the index of Pattern

- a) 8 b) 11 c) 9 d) 10

23) Hash Value is calculated by

- a) RabinKarp b) KMP Matching c) Boyer Moyer d) None

24) Bad Match Table is constructed in

- a) KMP b) Rabin Karp c) Boyer Moore d) None

25) Complexity of KMP in worst case

- a) $O(N)$ b) $O(n+m)$ c) $O(m)$ d) none of above



COURSE PLAN

LECTURE NUMBER	TOPICS TO BE COVERED
UNIT-1	
1	Objective, Scope and Outcome of the Subject
UNIT-2	
2	Review of Algorithm, Complexity Order Notations
3	Asymptotic Notations and Calculating Complexity
4	Complexity of Recurrence Relation
5	Complexity of Recurrence Relation cont..
6	Intro to Divide and Conquer, Binary Search
7	Merge Sort, Quick Sort
8	Quick Sort cont., Strassen Matrix Multiplication
UNIT-3	
9	Intro. To Greedy method , Knapsack Problem
10	Job Sequencing
11	Optimal Merge Patterns
12	Minimal Spanning Trees
13	Intro. To Dynamic Programming, Matrix Chain Multiplication
14	Longest Common Subsequence
15	Longest Common Subsequence cont.
16	0/1 Knapsack Problem.
Unit-4	
17	Intro. To Branch and Bound , Traveling Salesman Problem
18	Lower Bound Theory. Intro. To Backtracking Algorithms
19	Queens problem
20	Queens problem cont.
21	Pattern Matching, Naïve String Matching Algorithm
22	Rabin Karp string matching algorithms
23	KMP Matcher
24	Boyer Moore Algorithms
UNIT-5	
25	Formulation of Assignment and Quadratic Assignment Problem.
26	Las Vegas algorithms, Monte Carlo algorithms
27	Randomized algorithm for Min-Cut
28	Randomized algorithm for 2- SAT
29	Problem definition of Multicommodity flow
30	Flow shop scheduling
31	Flow shop Scheduling cont.
32	Network capacity assignment problems
UNIT-6	
33	Definitions of P, NP-Hard and NP-Complete Problems
34	P, NP-Hard and NP-Complete Problems cont.
35	Decision Problems.
36	Cook's Theorem
37	Proving NP- Complete Problems - Satisfiability problem
38	Vertex Cover Problem
39	Approximation Algorithms
40	Approximation Algorithms for Vertex Cover and Set Cover Problem.



**Swami Keshvanand Institute of Technology, Management & Gramothan,
Ramnagar, Jagatpura, Jaipur-302017, INDIA**

Approved by AICTE, Ministry of HRD, Government of India
Recognized by UGC under Section 2(f) of the UGC Act, 1956
Tel. : +91-0141- 5160400 Fax: +91-0141-2759555
E-mail: info@skit.ac.in Web: www.skit.ac.in

COURSE COVERAGE

Lect. No.	Date	Topic Covered
1		Introduction to Analysis of Algorithm
2		Analysis of Algorithm, Space Complexity, Time Complexity
3		Time Complexity of Insertion Sort
4		Asymptotic Notations, Big Oh, Omega and theta notation
5		Recurrence Relation, Substitution Method, Iteration Method
6		Master's Theorem, Recurrence Tree
7		Examples of Master's Method, Example of Recurrence Tree
8		Introduction To Divide and Conquer
9		Practice Worksheet-1
10		Binary Search, Merge Sort
11		Quick Sort, Analysis of Quick Sort
12		Strassen's Matrix Multiplication
13		Introduction to Greedy Approach, Fractional Knapsack problem
14		Job Sequencing Problem with deadline, Optimal Merge Pattern
15		Minimum Spanning Tree, Prim's Algorithm, Kruskal's Algorithm
16		Dynamic Programming, Difference between Dynamic, Greedy and D & C
17		0/1 Knapsack Problem, 0/1 Knapsack using set method
18		Longest Common Subsequence
19		Matrix Chain Multiplication
20		Practice Worksheet-2 and Live Quiz using Mentimeter
21		Introduction to Branch and Bound, Travelling Salesman problem
22		Travelling Salesman Problem
23		Introduction To Backtracking
24		Lower Bound Theory
25		N Queen's problem
26		N Queen's Problem Cont...
27		Pattern Matching: Naïve String matching Algorithm
28		Rabin Karp, KMP matcher
29		Boyer Moore Algorithm, Example of Pattern Matching
30		Unit test-I
31		Flow Shop Scheduling
32		Flow Shop Scheduling cont.....
33		Assignment problem using Branch and Bound
34		Revision Worksheet-3
35		Revision Worksheet-4
36		Revision Worksheet-4 cont..
37		Revision Worksheet-5
38		Revision Worksheet-5 cont..
39		Discussion of Question paper



Analysis of Assignment-I
Assignment-I, 2022-23

Branch/Semester: CSE/ V	Subject: Analysis of Algorithm	Subject Code: 5CS4-05
Assignment: I	Session (I/II/III): I	Max Marks: 20
Submitted By:	Mr. Loveleen Kumar	

A. Distribution of Course Outcome and Bloom's Taxonomy in Question Paper

Q. No	Questions	Marks	CO	BL
1	Define Space and Time Complexity. State three cases of Master's theorem.	1	CO1	L1
2	What are Asymptotic Notations? Name them.	1	CO1	L1
3	Differentiate Greedy and Dynamic Programming Technique.	1	CO2	L2
4	Differentiate Kruskal's and Prim's Algorithm.	1	CO2	L2
5	Explain Quick sort algorithm along with its analysis.	2	CO1	L2
6	Find complexity of $T(n)=3T(n/4)+cn^2$ using Recurrence Tree.	2	CO1	L3
7	Show all the steps to multiply two matrices using Strassen's Matrix Multiplication. $X = \begin{bmatrix} 3 & 2 \\ 4 & 8 \end{bmatrix} \text{ and } Y = \begin{bmatrix} 1 & 5 \\ 9 & 6 \end{bmatrix}$	2	CO2	L3
8	Explain Greedy approach with suitable example. Consider a Knapsack of capacity 60 and items with profits as (280,100,120,120) and weight (40,10,20,24). What is the maximum Profit earned. (use greedy approach)	2	CO2	L2
9	Explain Prim's algorithm and Kruskal's Algorithm using the following graph 	4	CO2	L3
10	Find the optimal parenthesizing of Matrix chain product whose sequence of dimensions are $\langle 5,10,3,12,5 \rangle$	4	CO2	L3

BL – Bloom's Taxonomy Level

(1- Remembering, 2- Understanding, 3 – Applying, 4 – Analyzing, 5 – Evaluating, 6 - Creating)



Analysis of Assignment-I
Assignment-I, 2022-23

Branch/Semester: CSE/ V	Subject: Analysis of Algorithm (AOA)	Subject Code: 5CS4-05
Assignment: I	Session (I/II/III): I	Max Marks: 20
Submitted By:	Mr. Loveleen Kumar	

CO – Course Outcome

B. Questions and Course Outcomes (COs) Mapping in terms of correlation

COs	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
CO1	3	3	-	-	3	3	-	-	-	-
CO2	-	-	3	3	-	-	3	3	3	3
CO3	-	-	-	-	-	-	-	-	-	-
CO4	-	-	-	-	-	-	-	-	-	-
CO5	-	-	-	-	-	-	-	-	-	-

1-Low Correlation; 2- Moderate Correlation; 3- Substantial Correlation

C. Mapping of Bloom's Level and Course Outcomes with Question Paper

Bloom's Level Mapping		CO Mapping	
Bloom's Level	Percentage	CO	Percentage
BL1	10.00% (2)	CO1	30.00% (6)
BL2	30.00% (6)	CO2	70.00% (14)
BL3	60.00% (12)	CO3	-
BL4		CO4	-
BL5		CO5	-
BL6		-	-

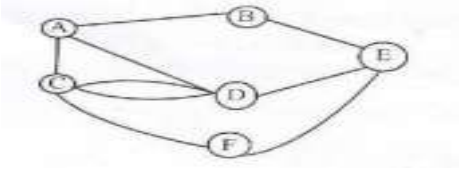
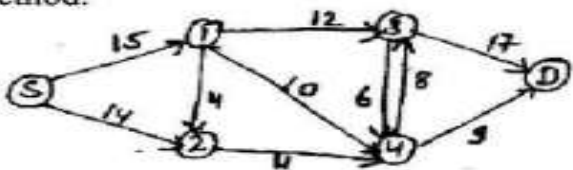


Analysis of Assignment-II

Assignment-II, 2022-23

Branch/Semester: CSE/ V	Subject: Analysis of Algorithm (AOA)	Subject Code: 5CS4-05
Assignment: II	Session (I/II/III): I	Max Marks: 40
Submitted By:	Mr. Loveleen Kumar	

A. Distribution of Course Outcome and Bloom's Taxonomy in Question Paper

Q. No	Questions	Marks	CO	BL
1	Define Cook's Theorem.	1	CO5	L1
2	Differentiate between Feasible and Optimal solution.	1	CO5	L2
3	What are Decision problems.	1	CO5	L1
4	What do you understand by Satisfiability problem(SAT).	1	CO5	L1
5	What is MIN-CUT?	1	CO4	L1
6	Name different String Matching Algorithms.	1	CO3	L1
7	Define Assignment problem.	1	CO3	L1
8	What is vertex cover?	1	CO5	L1
9	Differentiate between P and NP problem.	1	CO5	L2
10	What is Approximation Algorithm?	1	CO4	L1
11	Compare Las Vegas and Monte Carlo Approach.	2	CO4	L2
12	Explain Approximation Algorithm for Vertex Cover.	2	CO5	L2
13	Suppose $T = 1011101110$ $P = 111$. Find all the Valid Shift using Naïve String Matching Algorithm	2	CO3	L3
14	Give Randomized Algorithm to find Min cut for the following Graph. 	2	CO4	L3
15	Prove SAT is a NP complete Problem.	2	CO5	L2
16	Find Max flow using Ford Fulkerson method for the following method. 	5	CO4	L3



Analysis of Assignment-II

Assignment-II, 2022-23

17	Solve Assignment Problem using Branch and Bound for the following cost matrix. $\begin{matrix} & 4 & 7 & 5 \\ \text{Cost} = & 2 & 6 & 1 \\ & 3 & 9 & 8 \end{matrix}$	5	CO3	L3
18	Explain KMP Matcher along with lps construction for the following example txt[] = "AAAAABAAABA" pat[] = "AAAA"	5	CO3	L3
19	Explain Flow shop scheduling with suitable example.	5	CO4	L2

BL – Bloom's Taxonomy Level

(1- Remembering, 2- Understanding, 3 – Applying, 4 – Analyzing, 5 – Evaluating, 6 - Creating)



Analysis of Assignment-II

Assignment-II, 2022-23

Branch/Semester: CSE/ V	Subject: Analysis of Algorithm(AOA)	Subject Code: 5CS4-05
Assignment: II	Session (I/II/III): I	Max Marks: 40
Submitted By:	Mr. Loveleen Kumar	

CO – Course Outcome

B. Questions and Course Outcomes (COs) Mapping in terms of correlation

COs	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
CO1	-	-	-	-	-	-	-	-	-	-
CO2	-	-	-	-	-	-	-	-	-	-
CO3	-	-	-	-	-	3	3	-	-	-
CO4	-	-	-	-	3	-	-	-	-	3
CO5	3	3	3	3	-	-	-	3	3	-

COs	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19
CO1	-	-	-	-	-	-	-	-	-
CO2	-	-	-	-	-	-	-	-	-
CO3	-	-	3	-	-	-	3	3	-
CO4	3	-	-	3	-	3	-	-	3
CO5	-	3	-	-	3	-	-	-	-

1-Low Correlation; 2- Moderate Correlation; 3- Substantial Correlation



Analysis of Assignment-II

Assignment-II, 2022-23

Branch/Semester: CSE/ V	Subject: Analysis of Algorithm(AOA)	Subject Code: 5CS4-05
Assignment: II	Session (I/II/III): I	Max Marks: 40
Submitted By:	Mr. Loveleen Kumar	

C. Mapping of Bloom's Level and Course Outcomes with Question Paper

Bloom's Level Mapping		CO Mapping	
Bloom's Level	Percentage	CO	Percentage
BL1	20.00% (8)	CO1	-
BL2	32.50% (13)	CO2	-
BL3	47.50%(19)	CO3	35.00% (14)
BL4	-	CO4	40.00%(16)
BL5	-	CO5	25.00%(10)
BL6	-	-	-



Unit-1

Algorithm

An algorithm is a finite set of instructions, those if followed, accomplishes a particular task. It is not language specific, we can use any language and symbols to represent instructions.

The criteria of an algorithm

- **Input:** Zero or more inputs are externally supplied to the algorithm.
- **Output:** At least one output is produced by an algorithm.
- **Definiteness:** Each instruction is clear and unambiguous.
- **Finiteness:** In an algorithm, it will be terminated after a finite number of steps for all different cases.
- **Effectiveness:** Each instruction must be very basic, so the purpose of those instructions must be very clear to us.

Analysis of algorithms

Algorithm analysis is an important part of computational complexities. The complexity theory provides the theoretical estimates for the resources needed by an algorithm to solve any computational task. Analysis of the algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of the analysis of the algorithm is the required time or performance.

Complexities of an Algorithm

The complexity of an algorithm computes the amount of time and spaces required by an algorithm for an input of size (n). The complexity of an algorithm can be divided into two types. The **time complexity** and the **space complexity**.

Time Complexity of an Algorithm

The time complexity is defined as the process of determining a formula for total time required towards the execution of that algorithm. This calculation is totally independent of implementation and programming language.

Space Complexity of an Algorithm

Space complexity is defining as the process of defining a formula for prediction of how much memory space is required for the successful execution of the algorithm. The memory space is generally considered as the primary memory.



Asymptotic Notations

I. Asymptotic Analysis: Big-O Notation and More

In this tutorial, you will learn what asymptotic notations are. Also, you will learn about Big-O notation, Theta notation and Omega notation.

The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm. The efficiency is measured with the help of asymptotic notations.

An algorithm may not have the same performance for different types of inputs. With the increase in the input size, the performance will change.

The study of change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis.

II. 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.

For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

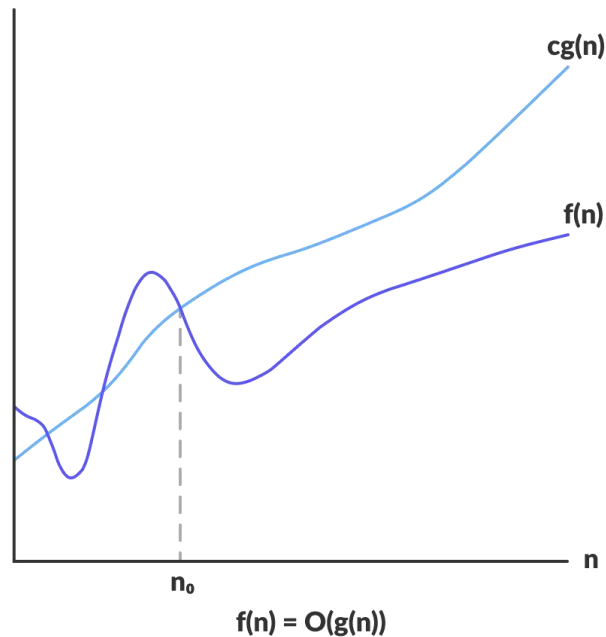
When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

There are mainly three asymptotic notations:

- Big-O notation
- Omega notation
- Theta notation

III. Big-O Notation (O-notation)

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



Big-O gives the upper bound of a function

$O(g(n)) = \{ f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}$

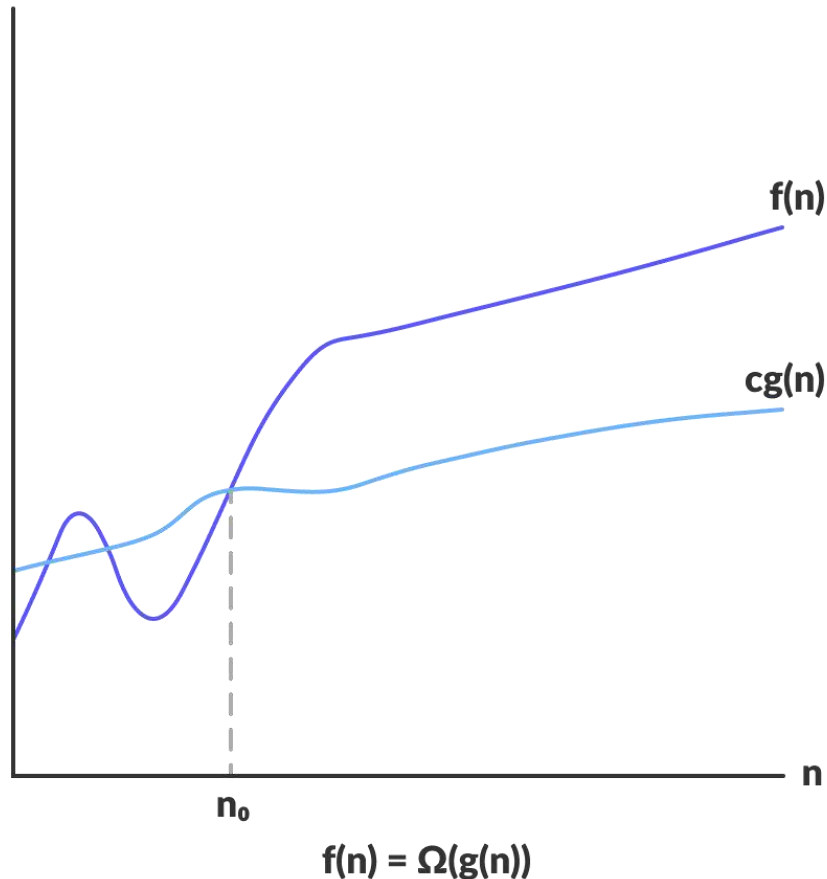
The above expression can be described as a function $f(n)$ belongs to the set $O(g(n))$ if there exists a positive constant c such that it lies between 0 and $cg(n)$, for sufficiently large n .

For any value of n , the running time of an algorithm does not cross the time provided by $O(g(n))$.

Since it gives the worst-case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst-case scenario.

IV. Omega Notation (Ω -notation)

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



Omega gives the lower bound of a function

$\Omega(g(n)) = \{ f(n): \text{there exist positive constants } c \text{ and } n_0$

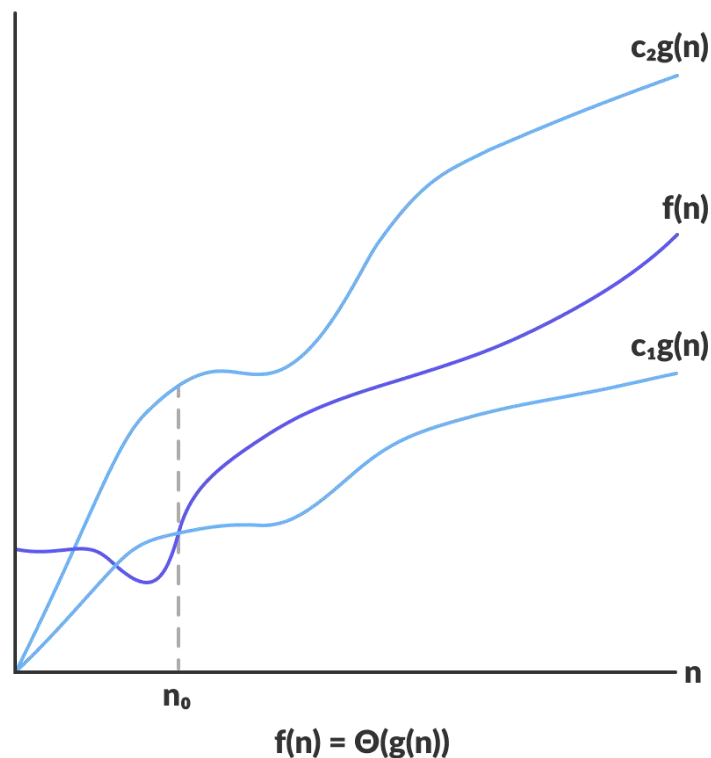
such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0 \}$

The above expression can be described as a function $f(n)$ belongs to the set $\Omega(g(n))$ if there exists a positive constant c such that it lies above $cg(n)$, for sufficiently large n .

For any value of n , the minimum time required by the algorithm is given by Omega $\Omega(g(n))$.

V. Theta Notation (Θ -notation)

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



Theta bounds the function within constants factors

For a function $g(n)$, $\Theta(g(n))$ is given by the relation:

$\Theta(g(n)) = \{ f(n): \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0 \}$

The above expression can be described as a function $f(n)$ belongs to the set $\Theta(g(n))$ if there exist positive constants c_1 and c_2 such that it can be sandwiched between $c_1g(n)$ and $c_2g(n)$, for sufficiently large n .

If a function $f(n)$ lies anywhere in between $c_1g(n)$ and $c_2g(n)$ for all $n \geq n_0$, then $f(n)$ is said to be asymptotically tight bound.



Recurrence Relation

A recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs. To solve a Recurrence Relation means to obtain a function defined on the natural numbers that satisfy the recurrence.

For Example, the Worst-Case Running Time $T(n)$ of the MERGE SORT Procedures is described by the recurrence.

$$T(n) = \theta(1) \text{ if } n=1$$

$$2T\left(\frac{n}{2}\right) + \theta(n) \text{ if } n>1$$

There are four methods for solving Recurrence:

1. Substitution Method
2. Iteration Method
3. Recursion Tree Method
4. Master Method

Substitution Method:

The Substitution Method Consists of two main steps:

1. Guess the Solution.
2. Use the mathematical induction to find the boundary condition and shows that the guess is correct.

For Example1 Solve the equation by Substitution Method.

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

We have to show that it is asymptotically bound by $O(\log n)$.

Solution:

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

We have to show that for some constant c

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

Put this in given Recurrence Equation.

$$\begin{aligned} T(n) &\leq c \log\left(\frac{n}{2}\right) + 1 \\ &\leq c \log\left(\frac{n}{2}\right) + 1 = c \log n - c \log_2 2 + 1 \\ &\leq c \log n \text{ for } c \geq 1 \end{aligned}$$

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

Example2 Consider the Recurrence

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

Find an Asymptotic bound on T .

Solution:

We guess the solution is $O(n \log n)$. Thus for constant ' c '.

$$T(n) \leq c n \log n$$

Put this in given Recurrence Equation.

Now,

$$\begin{aligned} T(n) &\leq 2c \left(\frac{n}{2}\right) \log\left(\frac{n}{2}\right) + n \\ &\leq cn \log n - cn \log_2 2 + n \\ &= cn \log n - n(c \log_2 2 - 1) \\ &\leq cn \log n \text{ for } (c \geq 1) \end{aligned}$$

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

Iteration Methods

It means to expand the recurrence and express it as a summation of terms of n and initial condition.

Example1: Consider the Recurrence

1. $T(n) = 1$ if $n=1$
2. $= 2T(n-1)$ if $n > 1$



Solution:

$$\begin{aligned}T(n) &= 2T(n-1) \\&= 2[2T(n-2)] = 2^2T(n-2) \\&= 4[2T(n-3)] = 2^3T(n-3) \\&= 8[2T(n-4)] = 2^4T(n-4) \quad (\text{Eq.1})\end{aligned}$$

Repeat the procedure for i times

$$\begin{aligned}T(n) &= 2^i T(n-i) \\ \text{Put } n-i=1 \text{ or } i=n-1 \text{ in (Eq.1)} \\ T(n) &= 2^{n-1} T(1) \\ &= 2^{n-1} \cdot 1 \quad \{T(1)=1 \text{given}\} \\ &= 2^{n-1}\end{aligned}$$

Example2: Consider the Recurrence

1. $T(n) = T(n-1) + 1$ and $T(1) = \theta(1)$.

Solution:

$$\begin{aligned}T(n) &= T(n-1) + 1 \\&= (T(n-2) + 1) + 1 = (T(n-3) + 1) + 1 + 1 \\&= T(n-4) + 4 = T(n-5) + 1 + 4 \\&= T(n-5) + 5 = T(n-k) + k\end{aligned}$$

Where $k = n-1$

$$\begin{aligned}T(n-k) &= T(1) = \theta(1) \\ T(n) &= \theta(1) + (n-1) = 1+n-1=n = \theta(n).\end{aligned}$$

Recursion Tree Method

Steps to Solve Recurrence Relations Using Recursion Tree Method-

Step-01:

Draw a recursion tree based on the given recurrence relation.

Step-02:

Determine-

- Cost of each level

- Total number of levels in the recursion tree
- Number of nodes in the last level
- Cost of the last level

Step-03:

Add cost of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation.

Following problems clearly illustrates how to apply these steps.

PRACTICE PROBLEMS BASED ON RECURSION TREE-

Problem-01:

Solve the following recurrence relation using recursion tree method-

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

Solution-

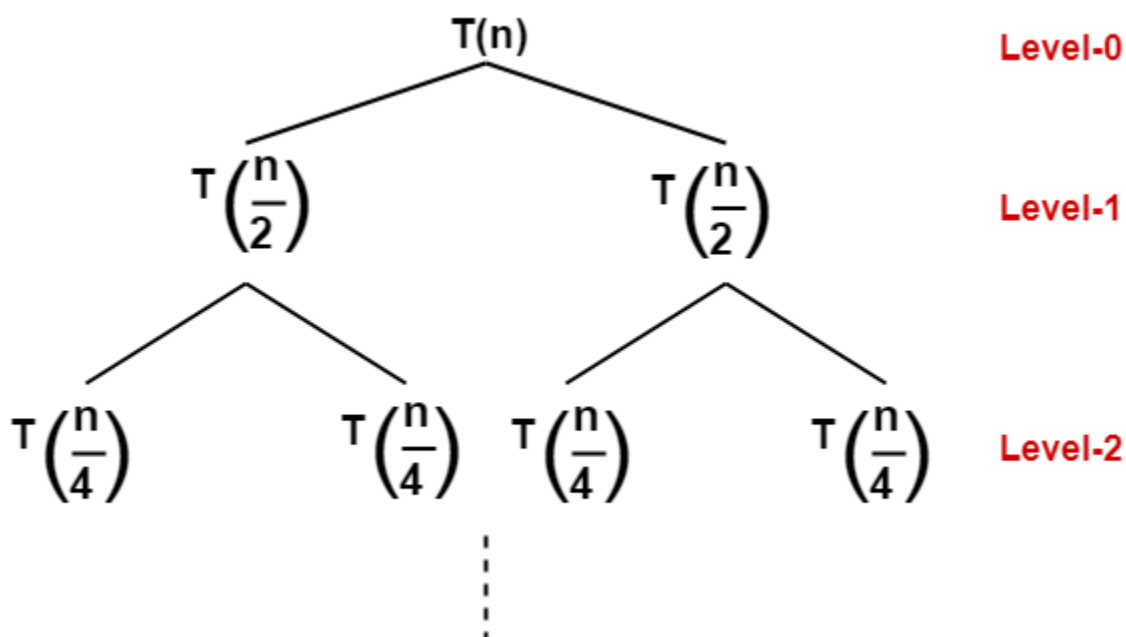
Step-01:

Draw a recursion tree based on the given recurrence relation.

The given recurrence relation shows-

- A problem of size n will get divided into 2 sub-problems of size $n/2$.
- Then, each sub-problem of size $n/2$ will get divided into 2 sub-problems of size $n/4$ and so on.
- At the bottom most layer, the size of sub-problems will reduce to 1.

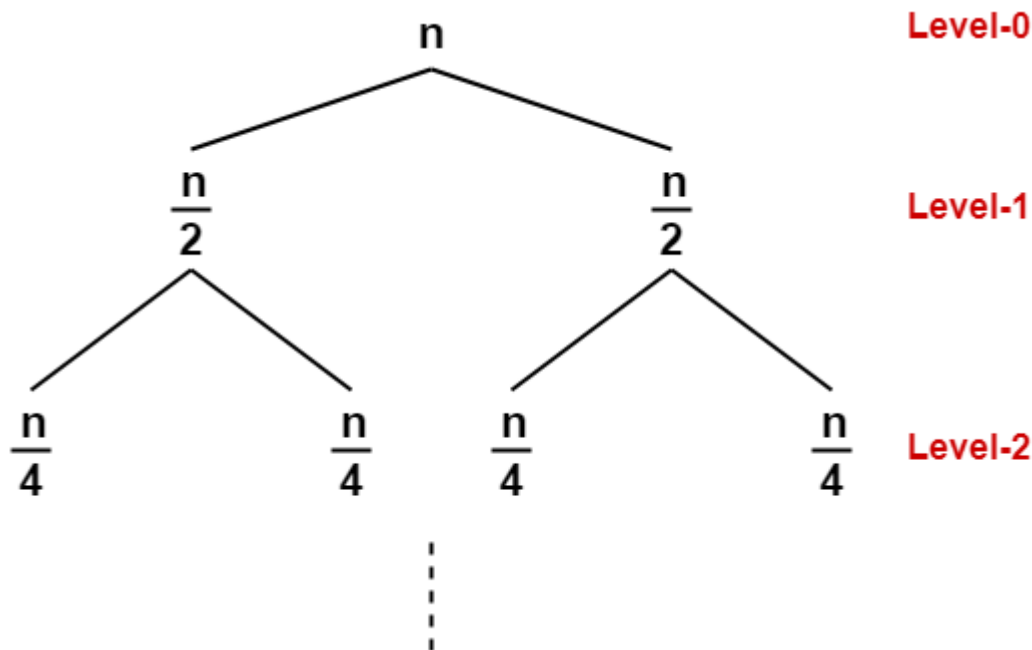
This is illustrated through following recursion tree-



The given recurrence relation shows-

- The cost of dividing a problem of size n into its 2 sub-problems and then combining its solution is n .
- The cost of dividing a problem of size $n/2$ into its 2 sub-problems and then combining its solution is $n/2$ and so on.

This is illustrated through following recursion tree where each node represents the cost of the corresponding sub-problem-



Step-02:

Determine cost of each level-

- Cost of level-0 = n
- Cost of level-1 = $n/2 + n/2 = n$
- Cost of level-2 = $n/4 + n/4 + n/4 + n/4 = n$ and so on.

Step-03:

Determine total number of levels in the recursion tree-

- Size of sub-problem at level-0 = $n/2^0$
- Size of sub-problem at level-1 = $n/2^1$
- Size of sub-problem at level-2 = $n/2^2$

Continuing in similar manner, we have-

Size of sub-problem at level- i = $n/2^i$

Suppose at level- x (last level), size of sub-problem becomes 1. Then-

$$n / 2^x = 1$$

$$2^x = n$$

Taking log on both sides, we get-

$$x \log 2 = \log n$$

$$x = \log_2 n$$

$$\therefore \text{Total number of levels in the recursion tree} = \log_2 n + 1$$

Step-04:

Determine number of nodes in the last level-

- Level-0 has 2^0 nodes i.e. 1 node
- Level-1 has 2^1 nodes i.e. 2 nodes
- Level-2 has 2^2 nodes i.e. 4 nodes

Continuing in similar manner, we have-

Level- $\log_2 n$ has $2^{\log_2 n}$ nodes i.e. n nodes

Step-05:

Determine cost of last level-

$$\text{Cost of last level} = n \times T(1) = \theta(n)$$

Step-06:

Add costs of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation-

$$T(n) = \{ n + n + n + \dots \} + \theta(n)$$

For $\log_2 n$ levels

$$= n \times \log_2 n + \theta(n)$$

$$= n \log_2 n + \theta(n)$$

$$= \theta(n \log_2 n)$$

Problem-02:

Solve the following recurrence relation using recursion tree method-

$$T(n) = T(n/5) + T(4n/5) + n$$

Solution-

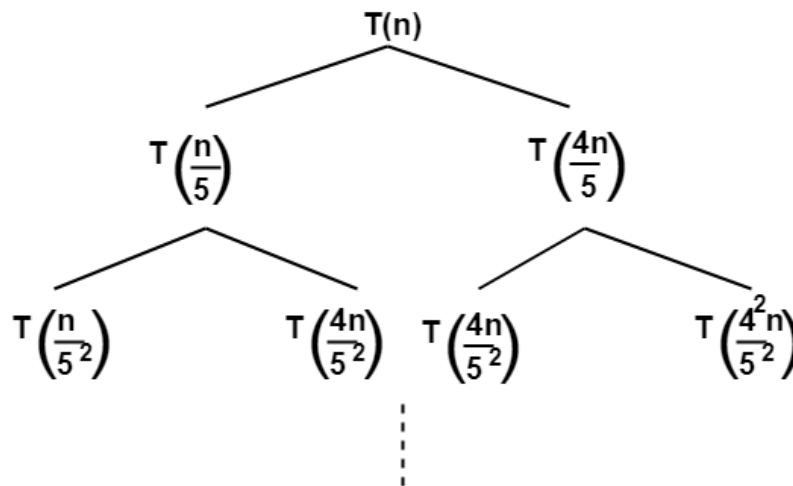
Step-01:

Draw a recursion tree based on the given recurrence relation.

The given recurrence relation shows-

- A problem of size n will get divided into 2 sub-problems- one of size $n/5$ and another of size $4n/5$.
- Then, sub-problem of size $n/5$ will get divided into 2 sub-problems- one of size $n/5^2$ and another of size $4n/5^2$.
- On the other side, sub-problem of size $4n/5$ will get divided into 2 sub-problems- one of size $4n/5^2$ and another of size $4^2n/5^2$ and so on.
- At the bottom most layer, the size of sub-problems will reduce to 1.

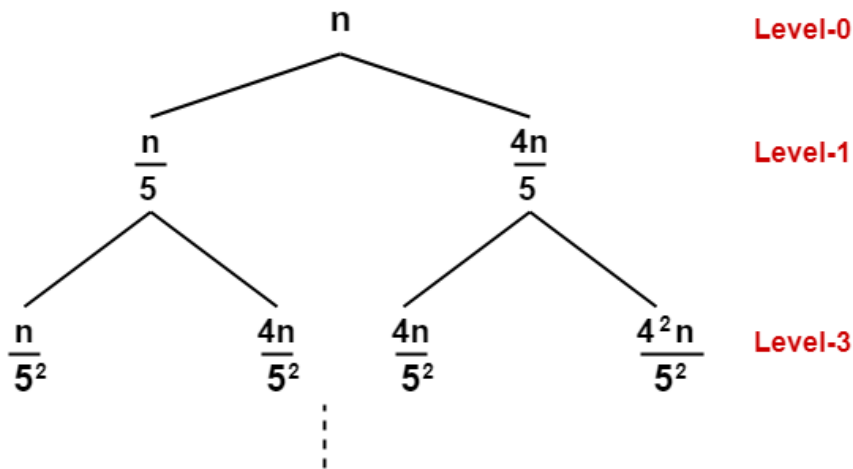
This is illustrated through following recursion tree-



The given recurrence relation shows-

- The cost of dividing a problem of size n into its 2 sub-problems and then combining its solution is n .
- The cost of dividing a problem of size $n/5$ into its 2 sub-problems and then combining its solution is $n/5$.
- The cost of dividing a problem of size $4n/5$ into its 2 sub-problems and then combining its solution is $4n/5$ and so on.

This is illustrated through following recursion tree where each node represents the cost of the corresponding sub-problem-



Step-02:

Determine cost of each level-

- Cost of level-0 = n
- Cost of level-1 = $\frac{n}{5} + \frac{4n}{5} = n$
- Cost of level-2 = $\frac{n}{5^2} + \frac{4n}{5^2} + \frac{4n}{5^2} + \frac{4^2n}{5^2} = n$

Step-03:

Determine total number of levels in the recursion tree. We will consider the rightmost sub tree as it goes down to the deepest level-

- Size of sub-problem at level-0 = $(4/5)^0n$
- Size of sub-problem at level-1 = $(4/5)^1n$
- Size of sub-problem at level-2 = $(4/5)^2n$

Continuing in similar manner, we have-

Size of sub-problem at level- i = $(4/5)^in$

Suppose at level- x (last level), size of sub-problem becomes 1. Then-

$$(4/5)^xn = 1$$

$$(4/5)^x = 1/n$$

Taking log on both sides, we get-

$$x \log(4/5) = \log(1/n)$$

$$x = \log_{5/4}n$$

$$\therefore \text{Total number of levels in the recursion tree} = \log_{5/4}n + 1$$

Step-04:

Determine number of nodes in the last level-

- Level-0 has 2^0 nodes i.e. 1 node
- Level-1 has 2^1 nodes i.e. 2 nodes
- Level-2 has 2^2 nodes i.e. 4 nodes

Continuing in similar manner, we have-

Level- $\log_{5/4} n$ has $2^{\log_{5/4} n}$ nodes

Step-05:

Determine cost of last level-

$$\text{Cost of last level} = 2^{\log_{5/4} n} \times T(1) = \theta(2^{\log_{5/4} n}) = \theta(n^{\log_{5/4} 2})$$

Step-06:

Add costs of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation-

$$T(n) = \{ n + n + n + \dots \} + \theta(n^{\log_{5/4} 2})$$

For $\log_{5/4} n$ levels

$$= n \log_{5/4} n + \theta(n^{\log_{5/4} 2})$$

$$= \theta(n \log_{5/4} n)$$

Problem-03:

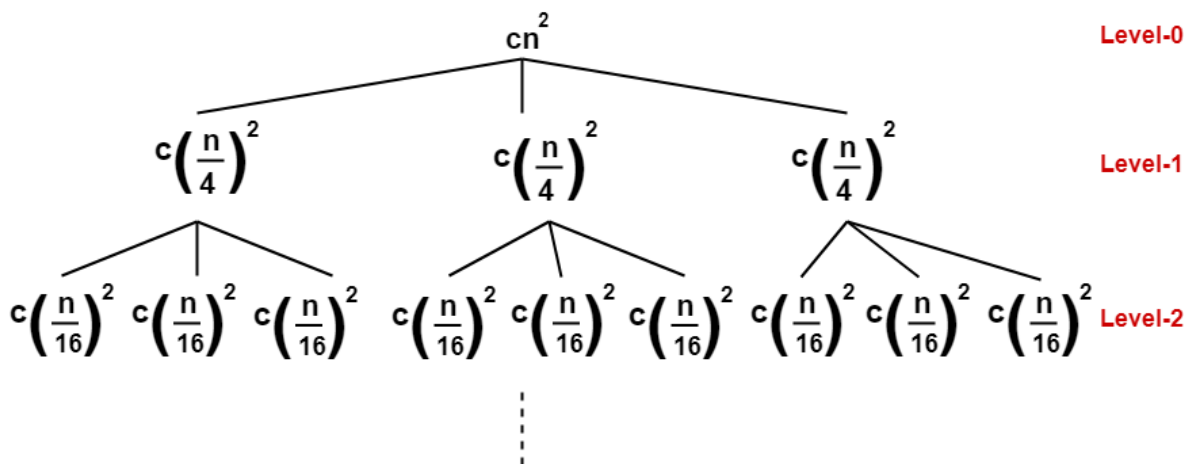
Solve the following recurrence relation using recursion tree method-

$$T(n) = 3T(n/4) + cn^2$$

Solution-

Step-01:

Draw a recursion tree based on the given recurrence relation-



(Here, we have directly drawn a recursion tree representing the cost of sub problems)

Step-02:

Determine cost of each level-

- Cost of level-0 = cn^2
- Cost of level-1 = $c(n/4)^2 + c(n/4)^2 + c(n/4)^2 = (3/16)cn^2$
- Cost of level-2 = $c(n/16)^2 \times 9 = (9/16^2)cn^2$

Step-03:

Determine total number of levels in the recursion tree-

- Size of sub-problem at level-0 = $n/4^0$
- Size of sub-problem at level-1 = $n/4^1$
- Size of sub-problem at level-2 = $n/4^2$

Continuing in similar manner, we have-

Size of sub-problem at level- i = $n/4^i$

Suppose at level- x (last level), size of sub-problem becomes 1. Then-

$$n/4^x = 1$$

$$4^x = n$$

Taking log on both sides, we get-

$$x \log 4 = \log n$$

$$x = \log_4 n$$

$$\therefore \text{Total number of levels in the recursion tree} = \log_4 n + 1$$

Step-04:

Determine number of nodes in the last level-

- Level-0 has 3^0 nodes i.e. 1 node
- Level-1 has 3^1 nodes i.e. 3 nodes
- Level-2 has 3^2 nodes i.e. 9 nodes

Continuing in similar manner, we have-

Level- $\log_4 n$ has $3^{\log_4 n}$ nodes i.e. $n^{\log_4 3}$ nodes

Step-05:

Determine cost of last level-

$$\text{Cost of last level} = n^{\log_4 3} \times T(1) = \theta(n^{\log_4 3})$$

Step-06:

Add costs of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation-

$$T(n) = \left\{ cn^2 + \frac{3}{16}cn^2 + \frac{9}{(16)^2}cn^2 + \dots \right\} + \theta(n^{\log_4 3})$$

For $\log_4 n$ levels

$$= cn^2 \{ 1 + (3/16) + (3/16)^2 + \dots \} + \theta(n^{\log_4 3})$$

Now, $\{ 1 + (3/16) + (3/16)^2 + \dots \}$ forms an infinite Geometric progression.

On solving, we get-

$$= (16/13)cn^2 \{ 1 - (3/16)^{\log_4 n} \} + \theta(n^{\log_4 3})$$

$$= (16/13)cn^2 - (16/13)cn^2 (3/16)^{\log_4 n} + \theta(n^{\log_4 3})$$

$$= O(n^2)$$

Divide and conquer algorithms

The two sorting algorithms we've seen so far, selection sort and insertion sort, have worst-case running times of $\Theta(n^2)$. When the size of the input array is large, these algorithms can take a long time to run. In this tutorial and the next one, we'll see two other sorting algorithms, merge sort and quicksort, whose running times are better. In particular, merge sort runs in $\Theta(n \lg n)$ time in all cases, and quicksort runs in $\Theta(n \lg n)$ time in the best case and on average, though its worst-case running time is $\Theta(n^2)$. Here's a table of these four sorting algorithms and their running times:

Algorithm	Worst-case running time	Best-case running time	Average-case running time
Selection sort	$\Theta(n^2)$ $\Theta(n^2)$, left parenthesis, n, squared, right parenthesis	$\Theta(n^2)$ $\Theta(n^2)$, left parenthesis, n, squared, right parenthesis	$\Theta(n^2)$ $\Theta(n^2)$, left parenthesis, n, squared, right parenthesis
Insertion sort	$\Theta(n^2)$ $\Theta(n^2)$, left parenthesis, n, squared, right parenthesis	$\Theta(n)$ $\Theta(n)$, left parenthesis, n, right parenthesis	$\Theta(n^2)$ $\Theta(n^2)$, left parenthesis, n, squared, right parenthesis

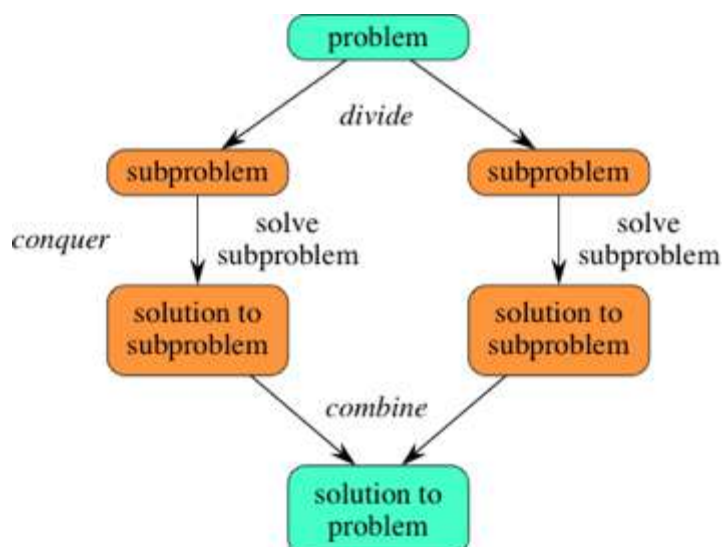
Merge sort	$\Theta(n \lg n)$ \Theta($n \lg n$) $\Theta(n \lg n)$ \Theta, left parenthesis, n , \lg , n , right parenthesis	$\Theta(n \lg n)$ \Theta($n \lg n$) $\Theta(n \lg n)$ \Theta, left parenthesis, n , \lg , n , right parenthesis	$\Theta(n \lg n)$ \Theta($n \lg n$) $\Theta(n \lg n)$ \Theta, left parenthesis, n , \lg , n , right parenthesis
Quicksort	$\Theta(n^2)$ \Theta(n^2) $\Theta(n^2)$ \Theta, left parenthesis, n , squared, right parenthesis	$\Theta(n \lg n)$ \Theta($n \lg n$) $\Theta(n \lg n)$ \Theta, left parenthesis, n , \lg , n , right parenthesis	$\Theta(n \lg n)$ \Theta($n \lg n$) $\Theta(n \lg n)$ \Theta, left parenthesis, n , \lg , n , right parenthesis

Divide-and-conquer

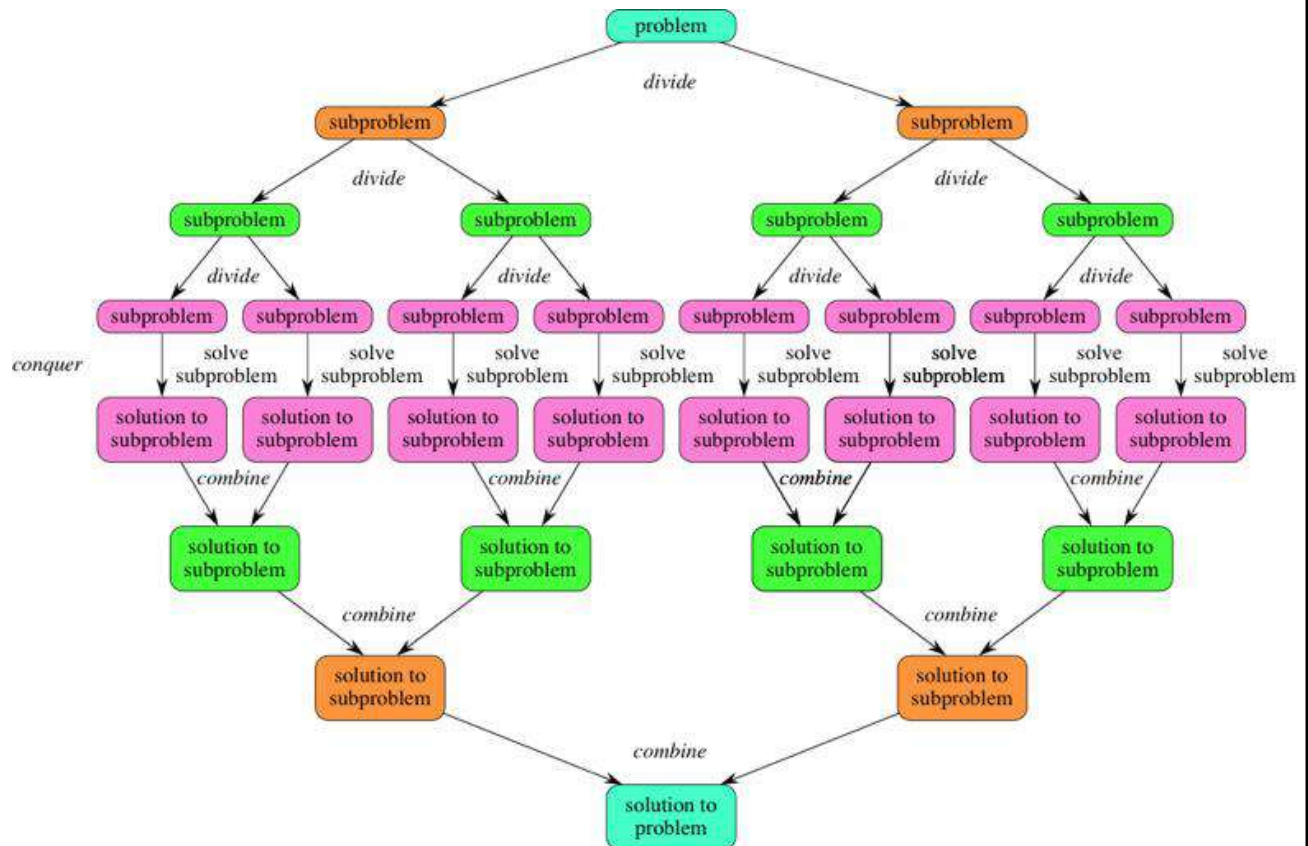
Both merge sort and quicksort employ a common algorithmic paradigm based on recursion. This paradigm, **divide-and-conquer**, breaks a problem into subproblems that are similar to the original problem, recursively solves the subproblems, and finally combines the solutions to the subproblems to solve the original problem. Because divide-and-conquer solves subproblems recursively, each subproblem must be smaller than the original problem, and there must be a base case for subproblems. You should think of a divide-and-conquer algorithm as having three parts:

1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
2. **Conquer** the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.
3. **Combine** the solutions to the subproblems into the solution for the original problem.

You can easily remember the steps of a divide-and-conquer algorithm as *divide, conquer, combine*. Here's how to view one step, assuming that each divide step creates two subproblems (though some divide-and-conquer algorithms create more than two):



If we expand out two more recursive steps, it looks like this:



Because divide-and-conquer creates at least two subproblems, a divide-and-conquer algorithm makes multiple recursive calls.

Binary Search

Problem: Given a sorted array `arr[]` of `n` elements, write a function to search a given element `x` in `arr[]` and return the index of `x` in the array.

Consider array is 0 base index.

Examples:

Input: `arr[] = { 10, 20, 30, 50, 60, 80, 110, 130, 140, 170}`, `x = 110`

Output: 6

Explanation: Element `x` is present at index 6.

Input: `arr[] = { 10, 20, 30, 40, 60, 110, 120, 130, 170}`, `x = 175`

Output: -1

Explanation: Element `x` is not present in `arr[]`.

Linear Search Approach: A simple approach is to do a **linear search**. The time complexity of the Linear search is $O(n)$. Another approach to perform the same task is using *Binary Search*.

Binary Search Approach:

Binary Search is a searching algorithm used in a sorted array by **repeatedly dividing the search**

interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to $O(\log n)$.

Binary Search Algorithm: The basic steps to perform Binary Search are:

- Begin with the mid element of the whole array as a search key.
- If the value of the search key is equal to the item then return an index of the search key.
- Or if the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half.
- Otherwise, narrow it to the upper half.
- Repeatedly check from the second point until the value is found or the interval is empty.

Binary Search Algorithm can be implemented in the following two ways

1. Iterative Method
2. Recursive Method

1. Iteration Method

```
binarySearch(arr, x, low, high)
```

```
repeat till low = high
```

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

```
if (x == arr[mid])
```

```
return mid
```

```
else if (x > arr[mid]) // x is on the right side
```

```
low = mid + 1
```

```
else // x is on the left side
```

```
high = mid - 1
```

2. Recursive Method (The recursive method follows the divide and conquers approach)

```
binarySearch(arr, x, low, high)
```

```
if low > high
```

```
return False
```

```
else
```

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

```
if x == arr[mid]
```

```
return mid
```


else if $x > arr[mid]$ // x is on the right side

return binarySearch(arr, x, mid + 1, high)

else // x is on the left side

return binarySearch(arr, x, low, mid - 1)

Illustration of Binary Search Algorithm:



Merge Sort Algorithm

The **Merge Sort** algorithm is a sorting algorithm that is based on the **Divide and Conquer** paradigm. In this algorithm, the array is initially divided into two equal halves and then they are combined in a sorted manner.

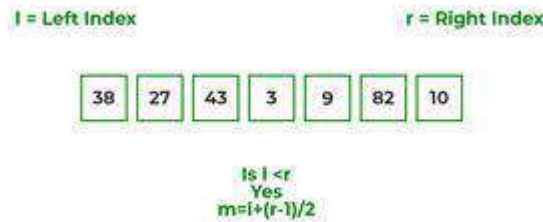
Merge Sort Working Process:

Think of it as a recursive algorithm continuously splits the array in half until it cannot be further divided. This means that if the array becomes empty or has only one element left, the dividing will stop, i.e. it is the base case to stop the recursion. If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves. Finally, when both halves are sorted, the merge operation is applied. Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.

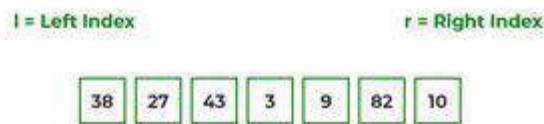
Illustration:

To know the functioning of merge sort, let's consider an array $arr[] = \{38, 27, 43, 3, 9, 82, 10\}$

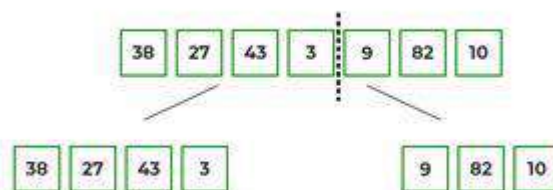
- At first, check if the left index of array is less than the right index, if yes then calculate its mid point



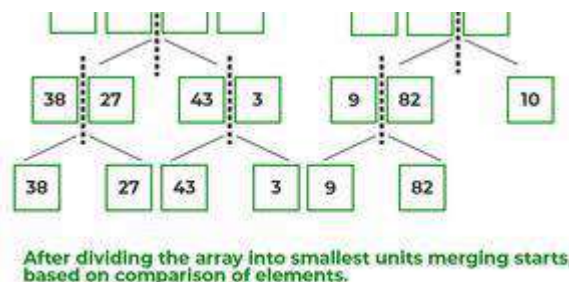
- Now, as we already know that merge sort first divides the whole array iteratively into equal halves, unless the atomic values are achieved.
- Here, we see that an array of 7 items is divided into two arrays of size 4 and 3 respectively.



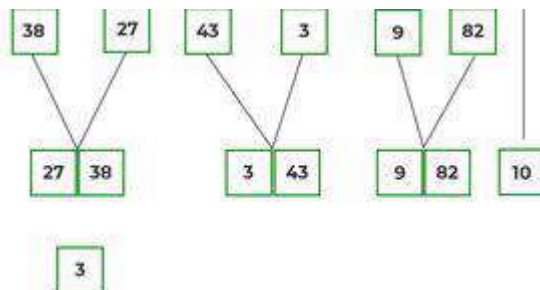
Now, again find that is left index is less than the right index for both arrays, if found yes, then again calculate mid points for both the arrays.



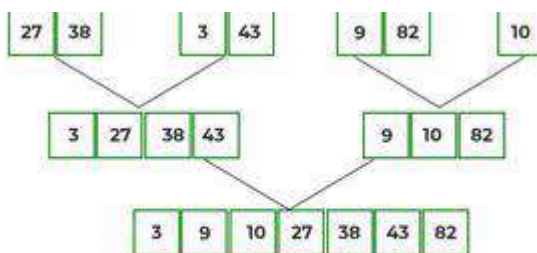
- Now, further divide these two arrays into further halves, until the atomic units of the array is reached and further division is not possible.



- After dividing the array into smallest units, start merging the elements again based on comparison of size of elements
- Firstly, compare the element for each list and then combine them into another list in a sorted manner.



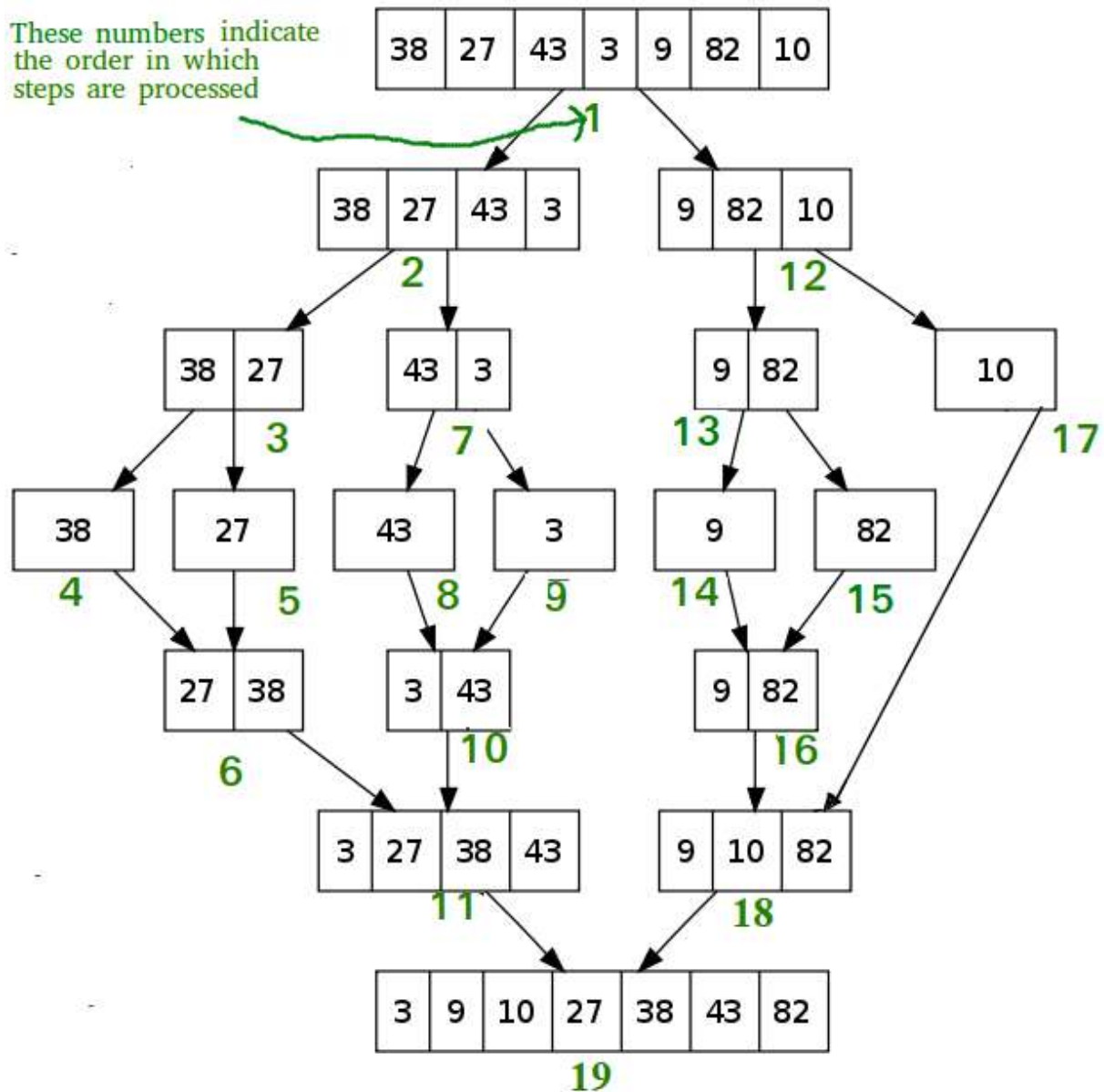
- After the final merging, the list looks like this:



The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}.

If we take a closer look at the diagram, we can see that the array is recursively divided into two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.

These numbers indicate
the order in which
steps are processed



Recursive steps of merge sort

Algorithm:

step 1: start

step 2: declare array and left, right, mid variable

step 3: perform merge function.

if left > right

return

mid= (left+right)/2

mergesort(array, left, mid)

mergesort(array, mid+1, right)

merge(array, left, mid, right)

step 4: Stop

Follow the steps below to solve the problem:

MergeSort(arr[], l, r)

If $r > l$

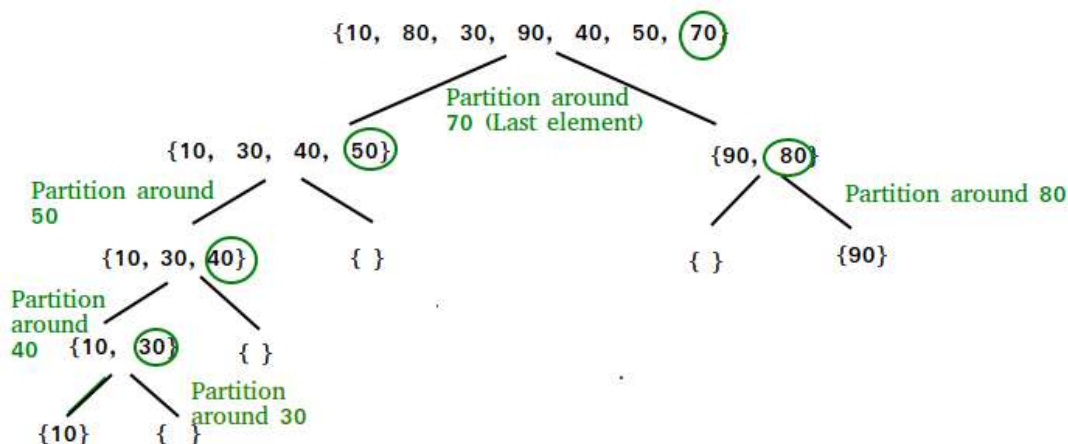
- Find the middle point to divide the array into two halves:
 - middle $m = l + (r - l)/2$
- Call mergeSort for first half:
 - Call mergeSort(arr, l, m)
- Call mergeSort for second half:
 - Call mergeSort(arr, m + 1, r)
- Merge the two halves sorted in steps 2 and 3:
 - Call merge(arr, l, m, r)

QuickSort

Like Merge Sort, **QuickSort** is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

- Always pick the first element as a pivot.
- Always pick the last element as a pivot (implemented below)
- Pick a random element as a pivot.
- Pick median as the pivot.

The key process in **quickSort** is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a 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.



Partition Algorithm:

There can be many ways to do partition, following pseudo-code adopts the method given in the CLRS book. The logic is simple, we start from the leftmost element and keep track of the index of smaller (or equal to) elements as i . While traversing, if we find a smaller element, we swap the current element with $arr[i]$. Otherwise, we ignore the current element.

Pseudo Code for recursive QuickSort function:

/* low \rightarrow Starting index, high \rightarrow Ending index */

quickSort(arr[], low, high) {

 if (low < high) {

 /* pi is partitioning index, arr[pi] is now at right place */

 pi = partition(arr, low, high);

 quickSort(arr, low, pi - 1); // Before pi

 quickSort(arr, pi + 1, high); // After pi

 }

}

Pseudo code for partition()

/* This function takes last element as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than pivot) to left of pivot and all greater elements to right of pivot */

partition (arr[], low, high)

{

 // pivot (Element to be placed at right position)

 pivot = arr[high];

 i = (low - 1) // Index of smaller element and indicates the

 // right position of pivot found so far

```
for (j = low; j <= high- 1; j++){
    // If current element is smaller than the pivot
    if (arr[j] < pivot){
        i++; // increment index of smaller element
        swap arr[i] and arr[j]
    }
}
swap arr[i + 1] and arr[high])
return (i + 1)
}
```

Illustration of partition() :

Consider: arr[] = { 10, 80, 30, 90, 40, 50, 70 }

- Indexes: 0 1 2 3 4 5 6
- low = 0, high = 6, pivot = arr[h] = 70
- Initialize index of smaller element, i = -1

Partition



Counter variables

I: Index of smaller element

J: Loop variable

We start the loop with initial values

Test Condition	Actions	Value of variables
arr [J] <= pivot		I = -1 J = 0

- Traverse elements from j = low to high-1
 - j = 0: Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
 - i = 0
- arr[] = { 10, 80, 30, 90, 40, 50, 70 } // No change as i and j are same
- j = 1: Since arr[j] > pivot, do nothing

Partition



↑
Pivot

Counter variables

I: Index of smaller element

J: Loop variable

Pass 2

Test Condition	Actions	Value of variables
arr [J] <= pivot		I = 0 J = 1
80 < 70 false	No Action	

- **j = 2** : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
- **i = 1**
- arr[] = { 10, 30, 80, 90, 40, 50, 70 } // We swap 80 and 30

Partition



↑
Pivot

Counter variables

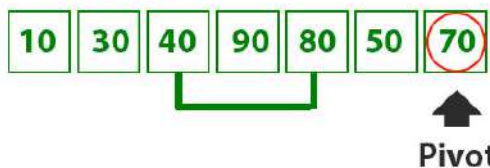
I: Index of smaller element

J: Loop variable

Test Condition	Actions	Value of variables
arr [J] <= pivot		I = 1 J = 2
30 < 70 true	i++ Swap(arr[i],arr[j])	

- **j = 3** : Since arr[j] > pivot, do nothing // No change in i and arr[]
- **j = 4** : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
- **i = 2**
- arr[] = { 10, 30, 40, 90, 80, 50, 70 } // 80 and 40 Swapped

Partition



Counter variables

I: Index of smaller element

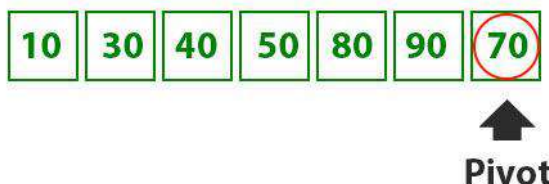
J: Loop variable

Pass 5

Test Condition	Actions	Value of variables
arr [J] <= pivot		I = 2 J = 4
40 < 70 true	i++ Swap(arr[i],arr[j])	

- **j = 5** : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]
- **i = 3**
- arr[] = { 10, 30, 40, 50, 80, 90, 70 } // 90 and 50 Swapped

Partition



Counter variables

I: Index of smaller element

J: Loop variable

Before Pass 7, J becomes 6
so we come out of the loop

Test Condition	Actions	Value of variables
arr [J] <= pivot		I = 3 J = 6

- We come out of loop because j is now equal to high-1.
- **Finally we place pivot at correct position by swapping arr[i+1] and arr[high] (or pivot)**
- arr[] = { 10, 30, 40, 50, 70, 90, 80 } // 80 and 70 Swapped

Partition



Counter Variable

I : Index of smaller element

J : Loop variable

We know swap arr[i+1] and pivot

I = 3

- Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.
- Since quick sort is a recursive function, we call the partition function again at left and right partitions

Quick sort left



Since quick sort is a recursion function,
we call the Partition function again

First 50 is the pivot.

As it is already at its correct position
we call the quicksort function again on the left part.

- Again call function at right part and swap 80 and 90

Quick sort Right



80 is the Pivot

80 and 90 are swapped to bring pivot
to correct position

Strassen's Matrix Multiplication



Problem Statement

Let us consider two matrices X and Y . We want to calculate the resultant matrix Z by multiplying X and Y .

Naïve Method

First, we will discuss naïve method and its complexity. Here, we are calculating $Z = X \times Y$. Using Naïve method, two matrices (X and Y) can be multiplied if the order of these matrices are $p \times q$ and $q \times r$. Following is the algorithm.

Algorithm: Matrix-Multiplication (X, Y, Z)

for $i = 1$ to p do

 for $j = 1$ to r do

$Z[i,j] := 0$

 for $k = 1$ to q do

$Z[i,j] := Z[i,j] + X[i,k] \times Y[k,j]$

Complexity

Here, we assume that integer operations take $O(1)$ time. There are three **for** loops in this algorithm and one is nested in other. Hence, the algorithm takes $O(n^3)$ time to execute.

Strassen's Matrix Multiplication Algorithm

In this context, using Strassen's Matrix multiplication algorithm, the time consumption can be improved a little bit.

Strassen's Matrix multiplication can be performed only on **square matrices** where n is a **power of 2**. Order of both of the matrices are $n \times n$.

Divide X, Y and Z into four $(n/2) \times (n/2)$ matrices as represented below –

$Z = [IKJL]$

$X = [ACBD]$ and $Y = [EGFH]$

Using Strassen's Algorithm compute the following –

$$M1 := (A+C) \times (E+F)$$

$$M2 := (B+D) \times (G+H)$$

$$M3 := (A-D) \times (E+H)$$

$$M4 := A \times (F-H)$$

$$M5 := (C+D) \times (E)$$

$$M6 := (A+B) \times (H)$$

$$M7 := D \times (G - E)$$

Then,

$$I := M2 + M3 - M6 - M7$$

$$J := M4 + M6$$

$$K := M5 + M7$$

$$L := M1 - M3 - M4 - M5$$

Analysis

$$T(n) = \begin{cases} c_7 \times T(n/2) + d \times n^2 & \text{if } n > 1 \\ \text{otherwise} \end{cases}$$

where c and d are constants

Using this recurrence relation, we get $T(n) = O(n \log 7)$

Hence, the complexity of Strassen's matrix multiplication algorithm is $O(n \log 7)$

Following is simple Divide and Conquer method to multiply two square matrices.

1. Divide matrices A and B in 4 sub-matrices of size $N/2 \times N/2$ as shown in the below diagram.
2. Calculate following values recursively. $ae + bg$, $af + bh$, $ce + dg$ and $cf + dh$.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

A B C

A, B and C are square matrices of size $N \times N$

a, b, c and d are submatrices of A, of size $N/2 \times N/2$

e, f, g and h are submatrices of B, of size $N/2 \times N/2$



Unit-2

Greedy Algorithm

In this tutorial, you will learn what a Greedy Algorithm is. Also, you will find an example of a greedy approach.

A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result.

The algorithm never reverses the earlier decision even if the choice is wrong. It works in a top-down approach.

This algorithm may not produce the best result for all the problems. It's because it always goes for the local best choice to produce the global best result.

However, we can determine if the algorithm can be used with any problem if the problem has the following properties:

1. Greedy Choice Property

If an optimal solution to the problem can be found by choosing the best choice at each step without reconsidering the previous steps once chosen, the problem can be solved using a greedy approach. This property is called greedy choice property.

2. Optimal Substructure

If the optimal overall solution to the problem corresponds to the optimal solution to its subproblems, then the problem can be solved using a greedy approach. This property is called optimal substructure.

Advantages of Greedy Approach

- The algorithm is **easier to describe**.
- This algorithm can **perform better** than other algorithms (but, not in all cases).

Drawback of Greedy Approach

As mentioned earlier, the greedy algorithm doesn't always produce the optimal solution. This is the major disadvantage of the algorithm



Knapsack Problem

Given a set of items, each with a weight and a value, determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

The knapsack problem is in combinatorial optimization problem. It appears as a subproblem in many, more complex mathematical models of real-world problems. One general approach to difficult problems is to identify the most restrictive constraint, ignore the others, solve a knapsack problem, and somehow adjust the solution to satisfy the ignored constraints.

Applications

In many cases of resource allocation along with some constraint, the problem can be derived in a similar way of Knapsack problem. Following is a set of example.

- Finding the least wasteful way to cut raw materials
- portfolio optimization
- Cutting stock problems

Problem Scenario

A thief is robbing a store and can carry a maximal weight of W into his knapsack. There are n items available in the store and weight of i^{th} item is w_i and its profit is p_i . What items should the thief take?

In this context, the items should be selected in such a way that the thief will carry those items for which he will gain maximum profit. Hence, the objective of the thief is to maximize the profit.

Based on the nature of the items, Knapsack problems are categorized as

- Fractional Knapsack
- Knapsack

Fractional Knapsack

In this case, items can be broken into smaller pieces, hence the thief can select fractions of items.

According to the problem statement,

- There are n items in the store
- Weight of i^{th} item $w_i > 0$
- □ Profit for i^{th} item $p_i > 0$
- and
- Capacity of the Knapsack is W

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction x_i of i^{th} item.

$$0 \leq x_i \leq 1$$

The i^{th} item contributes the weight $x_i.w_i$
to the total weight in the knapsack and profit $x_i.p_i$
to the total profit.

Hence, the objective of this algorithm is to

maximize $\sum_{n=1}^n (x_i.p_i)$

subject to constraint,

$\sum_{n=1}^n (x_i.w_i) \leq W$

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by

$\sum_{n=1}^n (x_i.w_i) = W$

In this context, first we need to sort those items according to the value of p_i/w_i

, so that $p_{i+1}/w_{i+1} \leq p_i/w_i$

. Here, x is an array to store the fraction of items.

Algorithm: Greedy-Fractional-Knapsack ($w[1..n]$, $p[1..n]$, W)

for $i = 1$ to n

do $x[i] = 0$

weight = 0

for $i = 1$ to n

if weight + $w[i] \leq W$ then

$x[i] = 1$

weight = weight + $w[i]$

else

$x[i] = (W - \text{weight}) / w[i]$

weight = W

break

return x

Analysis

If the provided items are already sorted into a decreasing order of p_i/w_i

, then the whileloop takes a time in $O(n)$; Therefore, the total time including the sort is in $O(n \log n)$.



Example

Let us consider that the capacity of the knapsack $W = 60$ and the list of provided items are shown in the following table –

Item	A	B	C	D
Profit	280	100	120	120
Weight	40	10	20	24
Ratio ($piwi$)				

7 10 6 5

As the provided items are not sorted based on $piwi$

. After sorting, the items are as shown in the following table.

Item	B	A	C	D
Profit	100	280	120	120
Weight	10	40	20	24
Ratio ($piwi$)				

10 7 6 5

Solution

After sorting all the items according to $piwi$

. First all of **B** is chosen as weight of **B** is less than the capacity of the knapsack. Next, item **A** is chosen, as the available capacity of the knapsack is greater than the weight of **A**. Now, **C** is chosen as the next item. However, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of **C**.

Hence, fraction of **C** (i.e. $(60 - 50)/20$) is chosen.

Now, the capacity of the Knapsack is equal to the selected items. Hence, no more item can be selected.

The total weight of the selected items is $10 + 40 + 20 * (10/20) = 60$

And the total profit is $100 + 280 + 120 * (10/20) = 380 + 60 = 440$

This is the optimal solution. We cannot gain more profit selecting any different combination of items.



Job Sequencing with Deadline

Problem Statement

In job sequencing problem, the objective is to find a sequence of jobs, which is completed within their deadlines and gives maximum profit.

Solution

Let us consider, a set of n given jobs which are associated with deadlines and profit is earned, if a job is completed by its deadline. These jobs need to be ordered in such a way that there is maximum profit.

It may happen that all of the given jobs may not be completed within their deadlines.

Assume, deadline of i^{th} job J_i is d_i and the profit received from this job is p_i . Hence, the optimal solution of this algorithm is a feasible solution with maximum profit.

Thus, $D(i) > 0$

for $1 \leq i \leq n$

.

Initially, these jobs are ordered according to profit, i.e. $p_1 \geq p_2 \geq p_3 \geq \dots \geq p_n$

.

Algorithm: Job-Sequencing-With-Deadline (D, J, n, k)

$D(0) := J(0) := 0$

$k := 1$

$J(1) := 1$ // means first job is selected

for $i = 2 \dots n$ do

$r := k$

 while $D(J(r)) > D(i)$ and $D(J(r)) \neq r$ do

$r := r - 1$

 if $D(J(r)) \leq D(i)$ and $D(i) > r$ then

 for $l = k \dots r + 1$ by -1 do

$J(l + 1) := J(l)$

$J(r + 1) := i$

$k := k + 1$

Analysis

In this algorithm, we are using two loops, one is within another. Hence, the complexity of this algorithm is $O(n^2)$



Example

Let us consider a set of given jobs as shown in the following table. We have to find a sequence of jobs, which will be completed within their deadlines and will give maximum profit. Each job is associated with a deadline and profit.

Job	J ₁	J ₂	J ₃	J ₄	J ₅
-----	----------------	----------------	----------------	----------------	----------------

Deadline	2	1	3	2	1
----------	---	---	---	---	---

Profit	60	100	20	40	20
--------	----	-----	----	----	----

Solution

To solve this problem, the given jobs are sorted according to their profit in a descending order. Hence, after sorting, the jobs are ordered as shown in the following table.

Job	J ₂	J ₁	J ₄	J ₃	J ₅
-----	----------------	----------------	----------------	----------------	----------------

Deadline	1	2	2	3	1
----------	---	---	---	---	---

Profit	100	60	40	20	20
--------	-----	----	----	----	----

From this set of jobs, first we select **J₂**, as it can be completed within its deadline and contributes maximum profit.

- Next, **J₁** is selected as it gives more profit compared to **J₄**.
- In the next clock, **J₄** cannot be selected as its deadline is over, hence **J₃** is selected as it executes within its deadline.
- The job **J₅** is discarded as it cannot be executed within its deadline.

Thus, the solution is the sequence of jobs (**J₂**, **J₁**, **J₃**), which are being executed within their deadline and gives maximum profit.

Total profit of this sequence is **100 + 60 + 20 = 180**.



Optimal Merge Pattern (Algorithm and Example)

Optimal merge pattern is a pattern that relates to the merging of two or more sorted files in a single sorted file. This type of merging can be done by the two-way merging method.

If we have two sorted files containing n and m records respectively then they could be merged together, to obtain one sorted file in time $O(n+m)$.

There are many ways in which pairwise merge can be done to get a single sorted file. Different pairings require a different amount of computing time. The main thing is to pairwise merge the n sorted files so that the number of comparisons will be less.

The formula of external merging cost is:

$$\sum_{i=1}^n f(i)d(i)$$

Where, $f(i)$ represents the number of records in each file and $d(i)$ represents the depth.

Algorithm for optimal merge pattern

Algorithm Tree(n)

//list is a global list of n single node

```
{
    For i=1 to i= n-1 do
    {
        // get a new tree node
        Pt: new treenode;

        // merge two trees with smallest length
        (Pt = lchild) = least(list);
        (Pt = rchild) = least(list);
        (Pt =weight) = ((Pt = lchild) = weight) = ((Pt = rchild) = weight);
        Insert (list , Pt);
    }
    // tree left in list
    Return least(list);
}
```

An optimal merge pattern corresponds to a binary merge tree with minimum weighted external path length. The function tree algorithm uses the greedy rule to get a two- way merge tree for n files. The algorithm contains an input list of n trees. There are three field child, rchild, and weight in each

node of the tree. Initially, each tree in a list contains just one node. This external node has lchild and rchild field zero whereas weight is the length of one of the n files to be merged. For any tree in the list with root node t, $t =$ it represents the weight that gives the length of the merged file. There are two functions least (list) and insert (list, t) in a function tree. Least (list) obtains a tree in lists whose root has the least weight and return a pointer to this tree. This tree is deleted from the list. Function insert (list, t) inserts the tree with root t into the list.

The main for loop in this algorithm is executed in $n-1$ times. If the list is kept in increasing order according to the weight value in the roots, then least (list) needs only $O(1)$ time and insert (list, t) can be performed in $O(n)$ time. Hence, the total time taken is $O(n^2)$. If the list is represented as a minheap in which the root value is less than or equal to the values of its children, then least (list) and insert (list, t) can be done in $O(\log n)$ time. In this condition, the computing time for the tree is $O(n \log n)$.

Advertisement

Example:

Given a set of unsorted files: 5, 3, 2, 7, 9, 13

Now, arrange these elements in ascending order: 2, 3, 5, 7, 9, 13

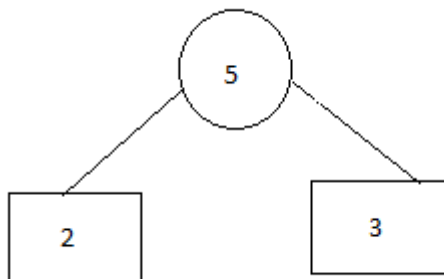
After this, pick two smallest numbers and repeat this until we left with only one number.

Now follow following steps:

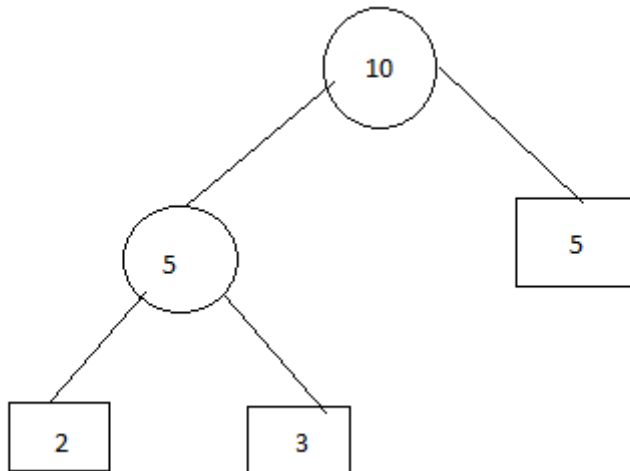
Step 1: Insert 2, 3



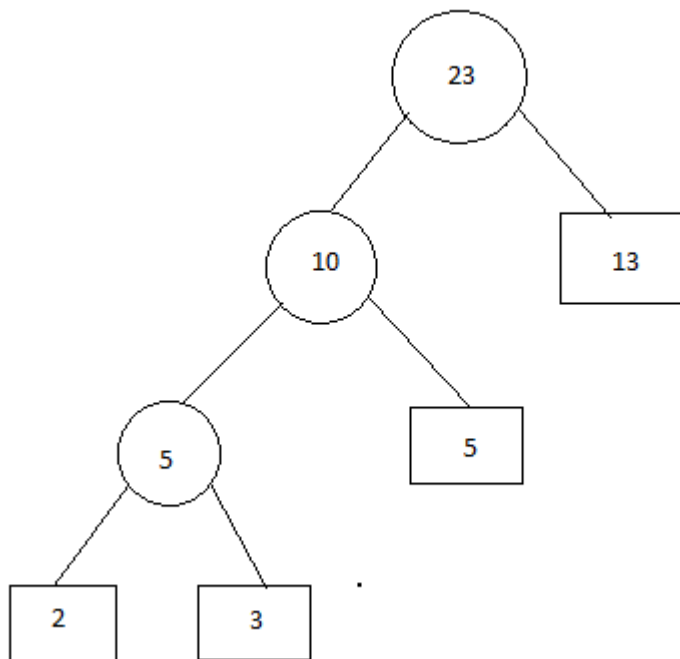
Step 2:



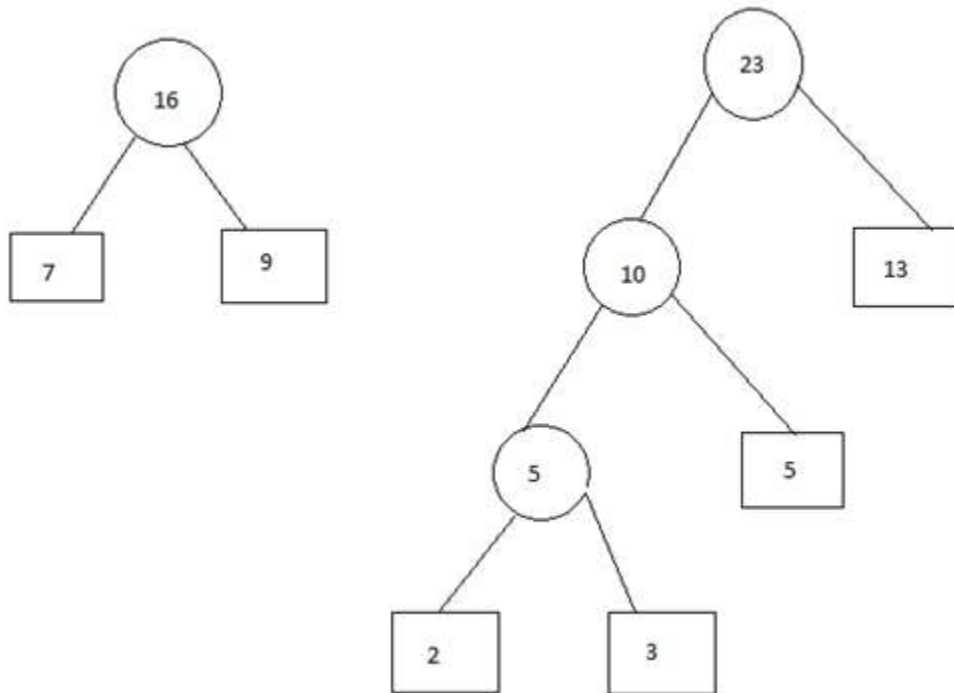
Step 3: Insert 5



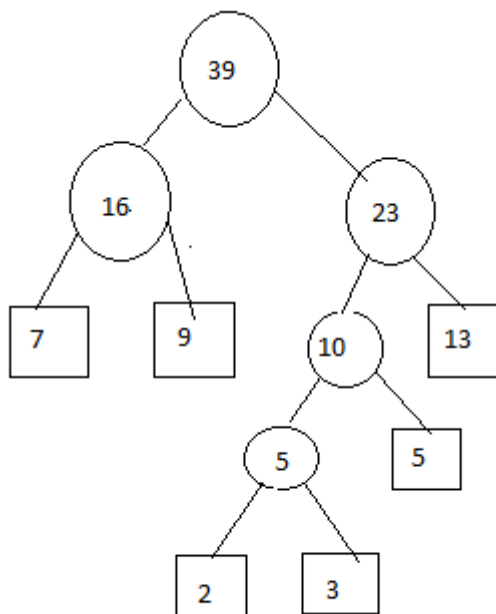
Step 4: Insert 13



Step 5: Insert 7 and 9



Step 6:



So, The merging cost = $5 + 10 + 16 + 23 + 39 = 93$

Methods of Minimum Spanning Tree

There are two methods to find Minimum Spanning Tree

1. Kruskal's Algorithm
2. Prim's Algorithm

Kruskal's Algorithm:

An algorithm to construct a Minimum Spanning Tree for a connected weighted graph. It is a Greedy Algorithm. The Greedy Choice is to put the smallest weight edge that does not because a cycle in the MST constructed so far.

If the graph is not linked, then it finds a Minimum Spanning Tree.

Steps for finding MST using Kruskal's Algorithm:

1. Arrange the edge of G in order of increasing weight.
2. Starting only with the vertices of G and proceeding sequentially add each edge which does not result in a cycle, until $(n - 1)$ edges are used.
3. EXIT.

MST- KRUSKAL (G, w)

1. $A \leftarrow \emptyset$
2. for each vertex $v \in V [G]$
3. do MAKE - SET (v)
4. sort the edges of E into non decreasing order by weight w
5. for each edge $(u, v) \in E$, taken in non decreasing order by weight
6. do if FIND-SET (μ) \neq if FIND-SET (v)
7. then $A \leftarrow A \cup \{(u, v)\}$
8. UNION (u, v)
9. return A

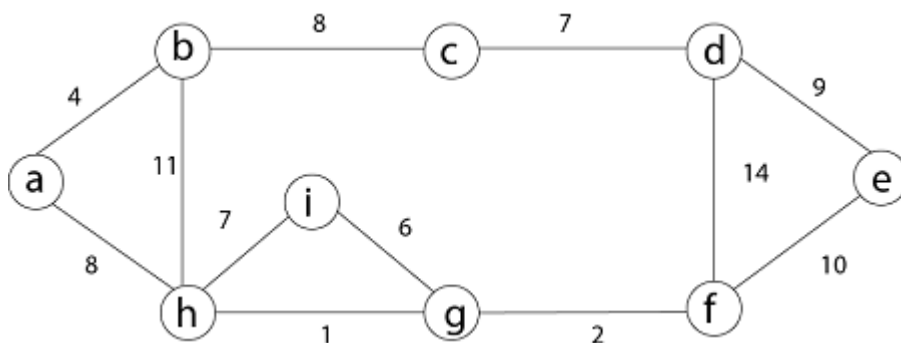
Analysis: Where E is the number of edges in the graph and V is the number of vertices, Kruskal's Algorithm can be shown to run in $O(E \log E)$ time, or simply, $O(E \log V)$ time, all with simple data structures. These running times are equivalent because:

- E is at most V^2 and $\log V^2 = 2 \times \log V$ is $O(\log V)$.
- If we ignore isolated vertices, which will each their components of the minimum spanning tree, $V \leq 2 E$, so $\log V$ is $O(\log E)$.

Thus the total time is

1. $O(E \log E) = O(E \log V)$.

For Example: Find the Minimum Spanning Tree of the following graph using Kruskal's algorithm.



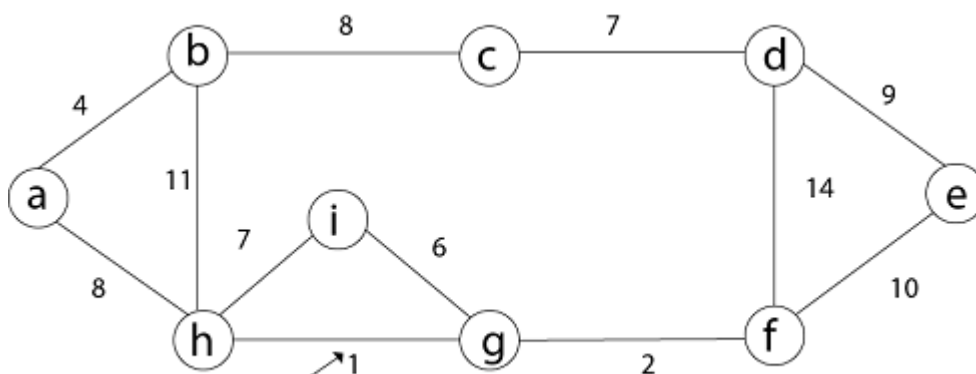
Solution: First we initialize the set A to the empty set and create $|V|$ trees, one containing each vertex with MAKE-SET procedure. Then sort the edges in E into order by non-decreasing weight.

There are 9 vertices and 12 edges. So MST formed $(9-1) = 8$ edges

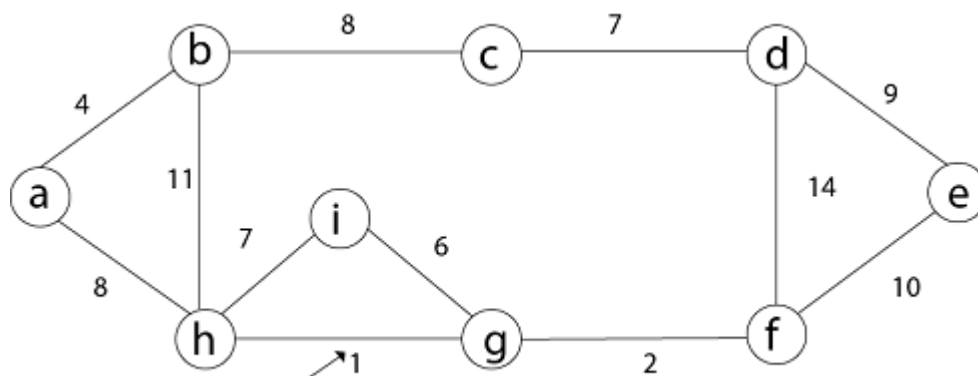
Weight	Source	Destination
1	h	g
2	g	f
4	a	b
6	i	g
7	h	i
7	c	d
8	b	c
8	a	h
9	d	e
10	e	f
11	b	h
14	d	f

Now, check for each edge (u, v) whether the endpoints u and v belong to the same tree. If they do then the edge (u, v) cannot be supplementary. Otherwise, the two vertices belong to different trees, and the edge (u, v) is added to A, and the vertices in two trees are merged in by union procedure.

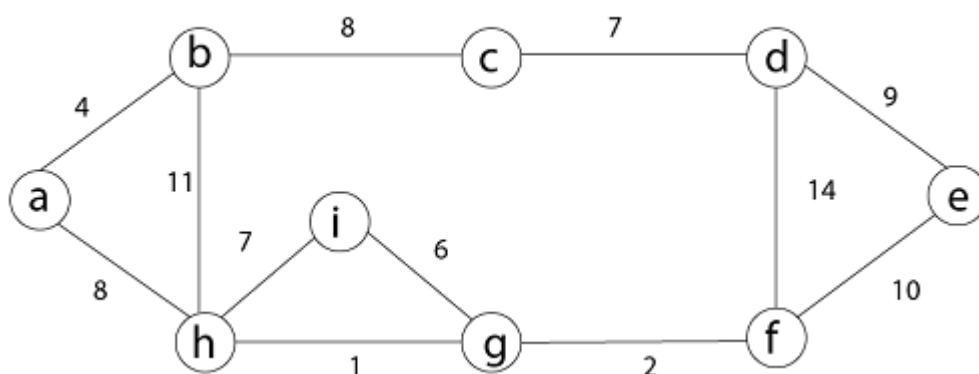
Step1: So, first take (h, g) edge



Step 2: then (g, f) edge.

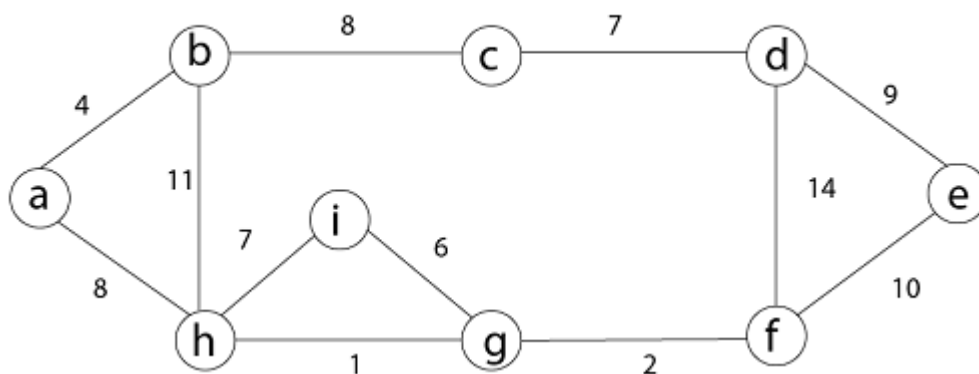


Step 3: then (a, b) and (i, g) edges are considered, and the forest becomes



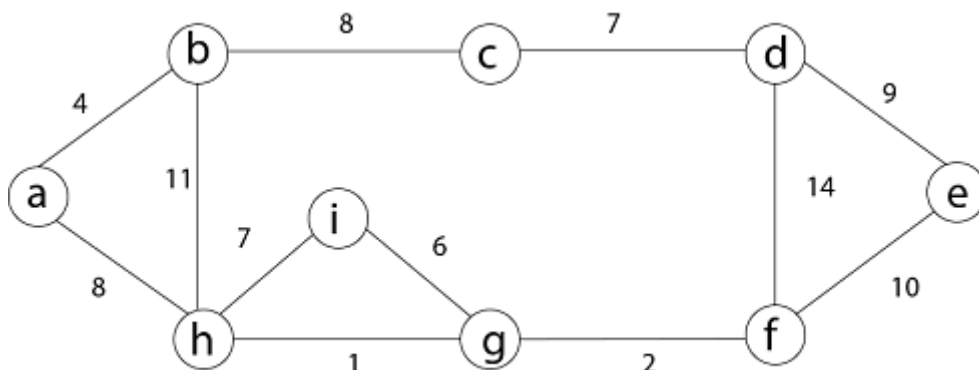
Step 4: Now, edge (h, i). Both h and i vertices are in the same set. Thus it creates a cycle. So this edge is discarded.

Then edge (c, d), (b, c), (a, h), (d, e), (e, f) are considered, and the forest becomes.



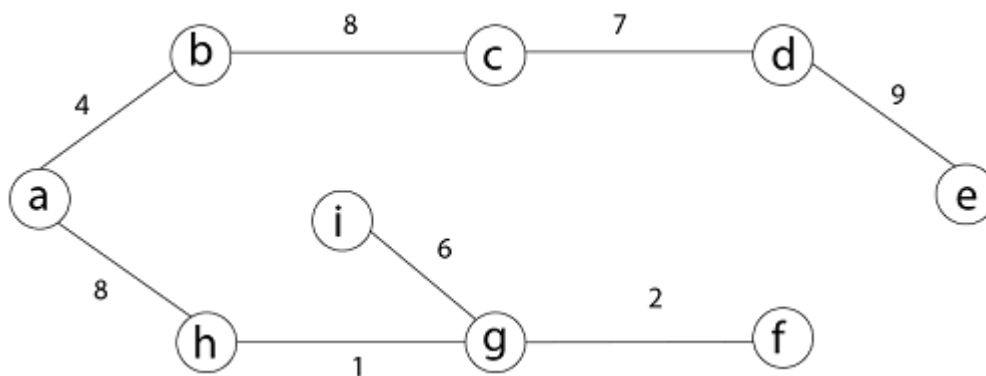
Step 5: In (e, f) edge both endpoints e and f exist in the same tree so discarded this edge. Then (b, h) edge, it also creates a cycle.

Step 6: After that edge (d, f) and the final spanning tree is shown as in dark lines.



Step 7: This step will be required Minimum Spanning Tree because it contains all the 9 vertices and $(9 - 1) = 8$ edges

1. $e \rightarrow f$, $b \rightarrow h$, $d \rightarrow f$ [cycle will be formed]



Minimum Cost MST

Prim's Algorithm

It is a greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices:

- Contain vertices already included in MST.
- Contain vertices not yet included.

At every step, it considers all the edges and picks the minimum weight edge. After picking the edge, it moves the other endpoint of edge to set containing MST.

Steps for finding MST using Prim's Algorithm:

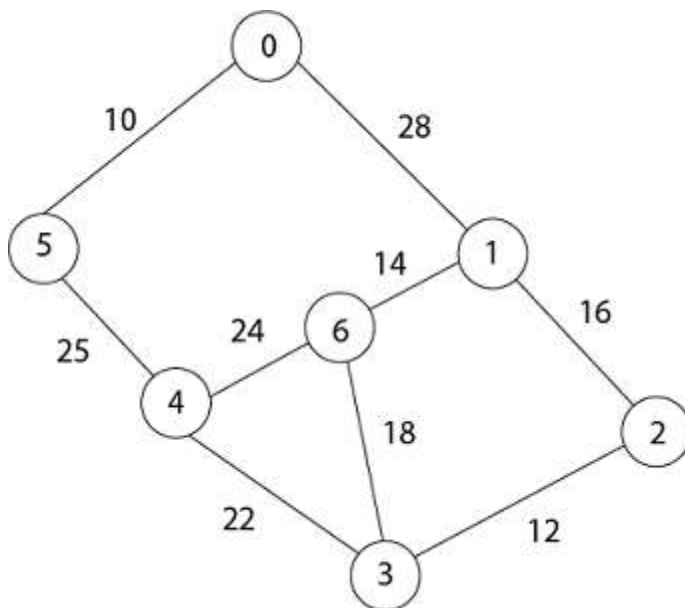
1. Create MST set that keeps track of vertices already included in MST.
2. Assign key values to all vertices in the input graph. Initialize all key values as INFINITE (∞). Assign key values like 0 for the first vertex so that it is picked first.
3. While MST set doesn't include all vertices.

1. Pick vertex u which is not in MST set and has minimum key value. Include ' u ' to MST set.
2. Update the key value of all adjacent vertices of u . To update, iterate through all adjacent vertices. For every adjacent vertex v , if the weight of edge $u.v$ less than the previous key value of v , update key value as a weight of $u.v$.

MST-PRIM (G, w, r)

1. for each $u \in V [G]$
2. do $key [u] \leftarrow \infty$
3. $\pi [u] \leftarrow NIL$
4. $key [r] \leftarrow 0$
5. $Q \leftarrow V [G]$
6. While $Q \neq \emptyset$
7. do $u \leftarrow EXTRACT - MIN (Q)$
8. for each $v \in Adj [u]$
9. do if $v \in Q$ and $w (u, v) < key [v]$
10. then $\pi [v] \leftarrow u$
11. $key [v] \leftarrow w (u, v)$

Example: Generate minimum cost spanning tree for the following graph using Prim's algorithm.



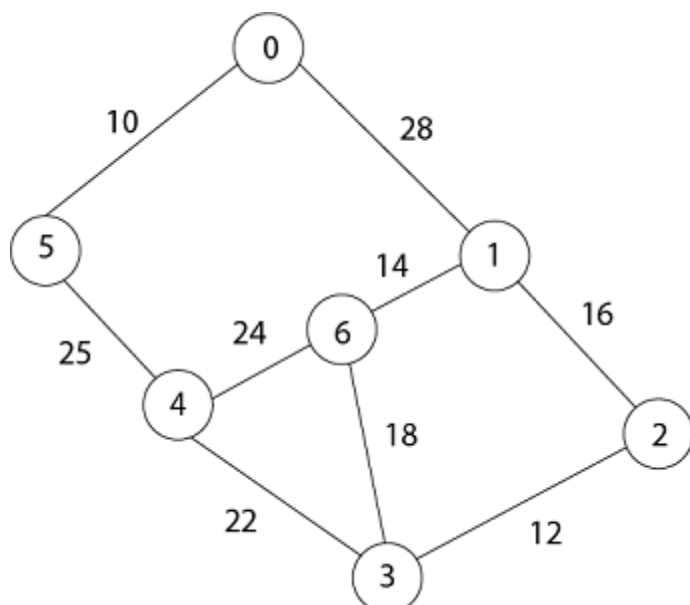
Solution: In Prim's algorithm, first we initialize the priority Queue Q . to contain all the vertices and the key of each vertex to ∞ except for the root, whose key is set to 0. Suppose 0 vertex is the root, i.e., r . By $EXTRACT - MIN (Q)$ procure, now $u = r$ and $Adj [u] = \{5, 1\}$.

Removing u from set Q and adds it to set $V - Q$ of vertices in the tree. Now, update the key and π fields of every vertex v adjacent to u but not in a tree.

Vertex	0	1	2	3	4	5	6
Key Value	0	∞	∞	∞	∞	∞	∞
Parent	NIL	NIL	NIL	NIL	NIL	NIL	NIL

1. Taking 0 as starting vertex
2. Root = 0
3. Adj [0] = 5, 1
4. Parent, π [5] = 0 and π [1] = 0
5. Key [5] = ∞ and key [1] = ∞
6. w [0, 5] = 10 and w (0,1) = 28
7. w (u, v) < key [5] , w (u, v) < key [1]
8. Key [5] = 10 and key [1] = 28
9. So update key value of 5 and 1 is:

Vertex	0	1	2	3	4	5	6
Key Value	0	28	∞	∞	∞	10	∞
Parent	NIL	0	NIL	NIL	NIL	0	NIL

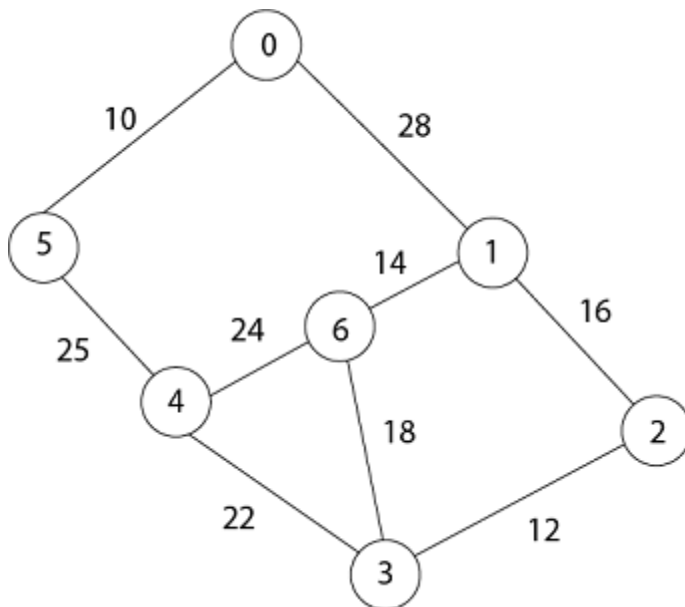


Now by EXTRACT_MIN (Q) Removes 5 because key [5] = 10 which is minimum so $u = 5$.

1. Adj [5] = {0, 4} and 0 is already in heap

2. Taking 4, key [4] = ∞ π [4] = 5
3. $(u, v) < \text{key}[v]$ then key [4] = 25
4. $w(5,4) = 25$
5. $w(5,4) < \text{key}[4]$
6. date key value and parent of 4.

Vertex	0	1	2	3	4	5	6
Key Value	0	28	∞	∞	25	10	∞
Parent	NIL	0	NIL	NIL	5	0	NIL



Now remove 4 because key [4] = 25 which is minimum, so $u = 4$

1. $\text{Adj}[4] = \{6, 3\}$
2. Key [3] = ∞ key [6] = ∞
3. $w(4,3) = 22$ $w(4,6) = 24$
4. $w(u, v) < \text{key}[v]$ $w(u, v) < \text{key}[v]$
5. $w(4,3) < \text{key}[3]$ $w(4,6) < \text{key}[6]$

Update key value of key [3] as 22 and key [6] as 24.

And the parent of 3, 6 as 4.

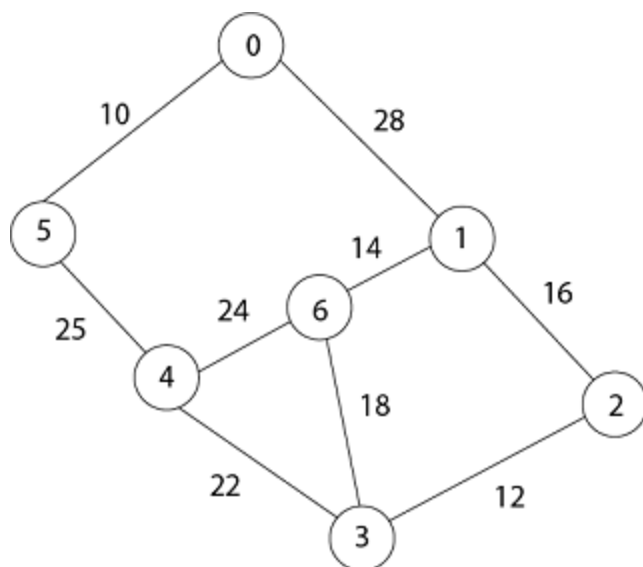
1. $\pi[3] = 4$ $\pi[6] = 4$

Vertex	0	1	2	3	4	5	6
Key Value	0	28	∞	22	25	10	24
Parent	NIL	0	NIL	4	5	0	4

1. $u = \text{EXTRACT_MIN}(3, 6)$ [key [3] < key [6]]

2. $u = 3$ i.e. $22 < 24$

Now remove 3 because key [3] = 22 is minimum so $u = 3$.



1. $\text{Adj}[3] = \{4, 6, 2\}$

2. 4 is already in heap

3. $4 \neq Q$ key [6] = 24 now becomes key [6] = 18

4. Key [2] = ∞ key [6] = 24

5. $w(3, 2) = 12$ $w(3, 6) = 18$

6. $w(3, 2) < \text{key}[2]$ $w(3, 6) < \text{key}[6]$

Now in Q, key [2] = 12, key [6] = 18, key [1] = 28 and parent of 2 and 6 is 3.

1. $\pi[2] = 3$ $\pi[6] = 3$

Now by EXTRACT_MIN (Q) Removes 2, because key [2] = 12 is minimum.

Vertex	0	1	2	3	4	5	6
Key Value	0	28	12	22	25	10	18
Parent	NIL	0	3	4	5	0	3

1. $u = \text{EXTRACT_MIN}(2, 6)$

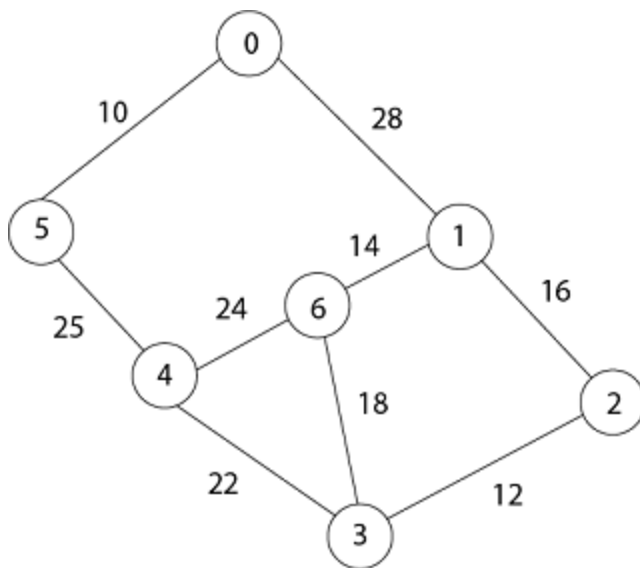
2. $u = 2$ [key [2] < key [6]]

3. $12 < 18$
4. Now the root is 2
5. $\text{Adj}[2] = \{3, 1\}$
6. 3 is already in a heap
7. Taking 1, $\text{key}[1] = 28$
8. $w(2,1) = 16$
9. $w(2,1) < \text{key}[1]$

So update key value of key [1] as 16 and its parent as 2.

1. $\pi[1] = 2$

Vertex	0	1	2	3	4	5	6
Key Value	0	16	12	22	25	10	18
Parent	NIL	2	3	4	5	0	3



Now by EXTRACT_MIN (Q) Removes 1 because $\text{key}[1] = 16$ is minimum.

1. $\text{Adj}[1] = \{0, 6, 2\}$
2. 0 and 2 are already in heap.
3. Taking 6, $\text{key}[6] = 18$
4. $w[1, 6] = 14$
5. $w[1, 6] < \text{key}[6]$

Update key value of 6 as 14 and its parent as 1.

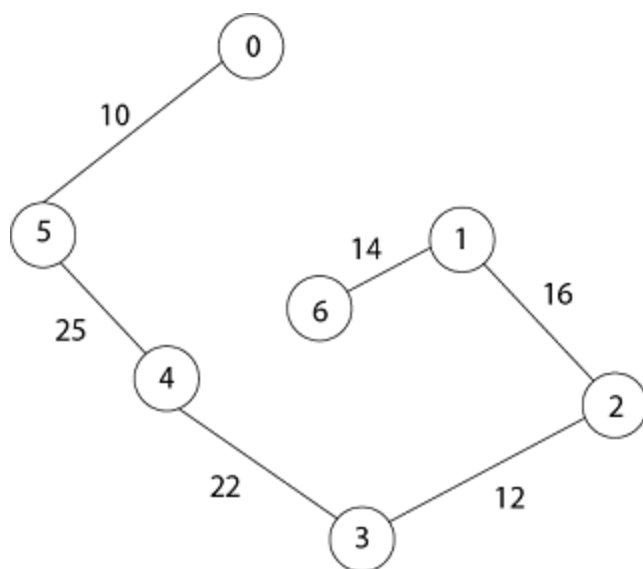
1. $\Pi[6] = 1$

Vertex	0	1	2	3	4	5	6
Key Value	0	16	12	22	25	10	14
Parent	NIL	2	3	4	5	0	1

Now all the vertices have been spanned, Using above the table we get Minimum Spanning Tree.

1. $0 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 6$
2. [Because $\Pi[5] = 0, \Pi[4] = 5, \Pi[3] = 4, \Pi[2] = 3, \Pi[1] = 2, \Pi[6] = 1$]

Thus the final spanning Tree is



$$\text{Total Cost} = 10 + 25 + 22 + 12 + 16 + 14 = 99$$



Dynamic Programming

Dynamic Programming is a technique in computer programming that helps to efficiently solve a class of problems that have overlapping subproblems and [optimal substructure](#) property.

If any problem can be divided into subproblems, which in turn are divided into smaller subproblems, and if there are overlapping among these subproblems, then the solutions to these subproblems can be saved for future reference. In this way, efficiency of the CPU can be enhanced. This method of solving a solution is referred to as dynamic programming.

Such problems involve repeatedly calculating the value of the same subproblems to find the optimum solution.

Dynamic Programming Example

Let's find the fibonacci sequence upto 5th term. A fibonacci series is the sequence of numbers in which each number is the sum of the two preceding ones. For example, 0,1,1, 2, 3. Here, each number is the sum of the two preceding numbers.

Algorithm

Let n be the number of terms.

1. If $n \leq 1$, return 1.
2. Else, return the sum of two preceding numbers.

We are calculating the fibonacci sequence up to the 5th term.

1. The first term is 0.
2. The second term is 1.
3. The third term is sum of 0 (from step 1) and 1(from step 2), which is 1.
4. The fourth term is the sum of the third term (from step 3) and second term (from step 2) i.e. $1 + 1 = 2$.
5. The fifth term is the sum of the fourth term (from step 4) and third term (from step 3) i.e. $2 + 1 = 3$.

Hence, we have the sequence 0,1,1, 2, 3. Here, we have used the results of the previous steps as shown below. This is called a **dynamic programming approach**.

$$F(0) = 0$$

$$F(1) = 1$$

$$F(2) = F(1) + F(0)$$

$$F(3) = F(2) + F(1)$$



$$F(4) = F(3) + F(2)$$

How Dynamic Programming Works

Dynamic programming works by storing the result of subproblems so that when their solutions are required, they are at hand and we do not need to recalculate them.

This technique of storing the value of subproblems is called memoization. By saving the values in the array, we save time for computations of sub-problems we have already come across.

```
var m = map(0 → 0, 1 → 1)
```

```
function fib(n)
```

```
  if key n is not in map m
```

```
    m[n] = fib(n - 1) + fib(n - 2)
```

```
  return m[n]
```

Dynamic programming by memoization is a top-down approach to dynamic programming. By reversing the direction in which the algorithm works i.e. by starting from the base case and working towards the solution, we can also implement dynamic programming in a bottom-up manner.

```
function fib(n)
```

```
  if n = 0
```

```
    return 0
```

```
  else
```

```
    var prevFib = 0, currFib = 1
```

```
    repeat n - 1 times
```

```
      var newFib = prevFib + currFib
```

```
      prevFib = currFib
```

```
      currFib = newFib
```

```
  return currFib
```

Recursion vs Dynamic Programming

Dynamic programming is mostly applied to recursive algorithms. This is not a coincidence, most optimization problems require recursion and dynamic programming is used for optimization.

But not all problems that use recursion can use Dynamic Programming. Unless there is a presence of overlapping subproblems like in the fibonacci sequence problem, a recursion can only reach the solution using a divide and conquer approach.

That is the reason why a recursive algorithm like Merge Sort cannot use Dynamic Programming,



because the subproblems are not overlapping in any way.

Greedy Algorithms vs Dynamic Programming

[Greedy Algorithms](#) are similar to dynamic programming in the sense that they are both tools for optimization.

However, greedy algorithms look for locally optimum solutions or in other words, a greedy choice, in the hopes of finding a global optimum. Hence greedy algorithms can make a guess that looks optimum at the time but becomes costly down the line and do not guarantee a globally optimum.

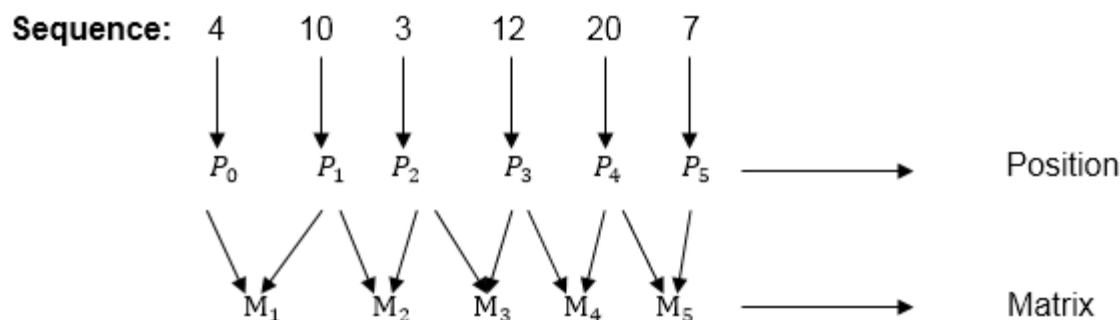
Dynamic programming, on the other hand, finds the optimal solution to subproblems and then makes an informed choice to combine the results of those subproblems to find the most optimum solution.

Matrix Chain Multiplication

Example: We are given the sequence {4, 10, 3, 12, 20, and 7}. The matrices have size 4 x 10, 10 x 3, 3 x 12, 12 x 20, 20 x 7. We need to compute $M[i,j]$, $0 \leq i, j \leq 5$. We know $M[i, i] = 0$ for all i .

1	2	3	4	5	
0					1
	0				2
		0			3
			0		4
				0	5

Let us proceed with working away from the diagonal. We compute the optimal solution for the product of 2 matrices.



Here P_0 to P_5 are Position and M_1 to M_5 are matrix of size $(p_i \text{ to } p_{i-1})$

On the basis of sequence, we make a formula

For M_i → $p[i]$ as column
 $p[i-1]$ as row

In Dynamic Programming, initialization of every method done by '0'. So we initialize it by '0'. It will sort out diagonally.

We have to sort out all the combination but the minimum output combination is taken into consideration.

Calculation of Product of 2 matrices:

$$\begin{aligned}
 1. \ m(1,2) &= m_1 \times m_2 \\
 &= 4 \times 10 \times 10 \times 3 \\
 &= 4 \times 10 \times 3 = 120
 \end{aligned}$$

$$2. m(2, 3) = m_2 \times m_3$$

$$= 10 \times 3 \times 3 \times 12$$

$$= 10 \times 3 \times 12 = 360$$

$$3. m(3, 4) = m_3 \times m_4$$

$$= 3 \times 12 \times 12 \times 20$$

$$= 3 \times 12 \times 20 = 720$$

$$4. m(4,5) = m_4 \times m_5$$

$$= 12 \times 20 \times 20 \times 7$$

$$= 12 \times 20 \times 7 = 1680$$

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

- We initialize the diagonal element with equal i,j value with '0'.
- After that second diagonal is sorted out and we get all the values corresponded to it

Now the third diagonal will be solved out in the same way.

Now product of 3 matrices:

$$M[1, 3] = M_1 M_2 M_3$$

1. There are two cases by which we can solve this multiplication: $(M_1 \times M_2) + M_3, M_1 + (M_2 \times M_3)$
2. After solving both cases we choose the case in which minimum output is there.

$$M[1, 3] = \min \left\{ \begin{array}{l} M[1,2] + M[3,3] + p_0 p_2 p_3 = 120 + 0 + 4.3.12 = 264 \\ M[1,1] + M[2,3] + p_0 p_1 p_3 = 0 + 360 + 4.10.12 = 840 \end{array} \right\}$$

$$M[1, 3] = 264$$

As Comparing both output **264** is minimum in both cases so we insert **264** in table and $(M_1 \times M_2) +$

M_3 this combination is chosen for the output making.

$$M[2, 4] = M_2 M_3 M_4$$

1. There are two cases by which we can solve this multiplication: $(M_2 \times M_3) + M_4$, $M_2 + (M_3 \times M_4)$
2. After solving both cases we choose the case in which minimum output is there.

$$M[2, 4] = \min \begin{cases} M[2,3] + M[4,4] + p_1 p_3 p_4 = 360 + 0 + 10.12.20 = 2760 \\ M[2,2] + M[3,4] + p_1 p_2 p_4 = 0 + 720 + 10.3.20 = 1320 \end{cases}$$

$$M[2, 4] = 1320$$

As Comparing both output **1320** is minimum in both cases so we insert **1320** in table and $M_2 + (M_3 \times M_4)$ this combination is chosen for the output making.

$$M[3, 5] = M_3 M_4 M_5$$

1. There are two cases by which we can solve this multiplication: $(M_3 \times M_4) + M_5$, $M_3 + (M_4 \times M_5)$
2. After solving both cases we choose the case in which minimum output is there.

$$M[3, 5] = \min \begin{cases} M[3,4] + M[5,5] + p_2 p_4 p_5 = 720 + 0 + 3.20.7 = 1140 \\ M[3,3] + M[4,5] + p_2 p_3 p_5 = 0 + 1680 + 3.12.7 = 1932 \end{cases}$$

$$M[3, 5] = 1140$$

As Comparing both output **1140** is minimum in both cases so we insert **1140** in table and $(M_3 \times M_4) + M_5$ this combination is chosen for the output making.

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

→

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

Now Product of 4 matrices:

$$M[1, 4] = M_1 M_2 M_3 M_4$$

There are three cases by which we can solve this multiplication:

1. $(M_1 \times M_2 \times M_3) M_4$
2. $M_1 \times (M_2 \times M_3 \times M_4)$
3. $(M_1 \times M_2) \times (M_3 \times M_4)$

After solving these cases we choose the case in which minimum output is there

$$M[1, 4] = \min \begin{cases} M[1,3] + M[4,4] + p_0p_3p_4 = 264 + 0 + 4.12.20 = 1224 \\ M[1,2] + M[3,4] + p_0p_2p_4 = 120 + 720 + 4.3.20 = 1080 \\ M[1,1] + M[2,4] + p_0p_1p_4 = 0 + 1320 + 4.10.20 = 2120 \end{cases}$$

$$M[1, 4] = 1080$$

As comparing the output of different cases then '1080' is minimum output, so we insert 1080 in the table and $(M_1 \times M_2) \times (M_3 \times M_4)$ combination is taken out in output making,

$$M[2, 5] = M_2 M_3 M_4 M_5$$

There are three cases by which we can solve this multiplication:

1. $(M_2 \times M_3 \times M_4) \times M_5$
2. $M_2 \times (M_3 \times M_4 \times M_5)$
3. $(M_2 \times M_3) \times (M_4 \times M_5)$

After solving these cases we choose the case in which minimum output is there

$$M[2, 5] = \min \begin{cases} M[2,4] + M[5,5] + p_1p_4p_5 = 1320 + 0 + 10.20.7 = 2720 \\ M[2,3] + M[4,5] + p_1p_3p_5 = 360 + 1680 + 10.12.7 = 2880 \\ M[2,2] + M[3,5] + p_1p_2p_5 = 0 + 1140 + 10.3.7 = 1350 \end{cases}$$

$$M[2, 5] = 1350$$

As comparing the output of different cases then '1350' is minimum output, so we insert 1350 in the table and $M_2 \times (M_3 \times M_4 \times M_5)$ combination is taken out in output making.

1	2	3	4	5		1	2	3	4	5		1	2	3	4	5
0	120	264			1	0	120	264	1080		1	0	120	264	1080	
	0	360	1320		2		0	360	1320	1350	2		0	360	1320	1350
		0	720	1140	3			0	720	1140	3			0	720	1140
			0	1680	4				0	1680	4				0	1680
				0	5					0	5					0

Now Product of 5 matrices:

$$M[1, 5] = M_1 M_2 M_3 M_4 M_5$$

There are five cases by which we can solve this multiplication:

1. $(M_1 \times M_2 \times M_3 \times M_4) \times M_5$
2. $M_1 \times (M_2 \times M_3 \times M_4 \times M_5)$
3. $(M_1 \times M_2 \times M_3) \times M_4 \times M_5$
4. $M_1 \times M_2 \times (M_3 \times M_4 \times M_5)$

After solving these cases we choose the case in which minimum output is there

$$M[1, 5] = \min \begin{cases} M[1,4] + M[5,5] + p_0 p_4 p_5 = 1080 + 0 + 4.20.7 = 1544 \\ M[1,3] + M[4,5] + p_0 p_3 p_5 = 264 + 1680 + 4.12.7 = 2016 \\ M[1,2] + M[3,5] + p_0 p_2 p_5 = 120 + 1140 + 4.3.7 = 1344 \\ M[1,1] + M[2,5] + p_0 p_1 p_5 = 0 + 1350 + 4.10.7 = 1630 \end{cases}$$

$$M[1, 5] = 1344$$

As comparing the output of different cases then '1344' is minimum output, so we insert 1344 in the table and $M_1 \times M_2 \times (M_3 \times M_4 \times M_5)$ combination is taken out in output making.

Final Output is:

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

→

1	2	3	4	5	
0	120	264	1080	1344	1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

Step 3: Computing Optimal Costs: let us assume that matrix A_i has dimension $p_{i-1} \times p_i$ for $i=1, 2, 3, \dots, n$. The input is a sequence (p_0, p_1, \dots, p_n) where $\text{length}[p] = n+1$. The procedure uses an auxiliary table $m[1, \dots, n, 1, \dots, n]$ for storing $m[i, j]$ costs an auxiliary table $s[1, \dots, n, 1, \dots, n]$ that record which index of k achieved the optimal costs in computing $m[i, j]$.

The algorithm first computes $m[i, j] \leftarrow 0$ for $i=1, 2, 3, \dots, n$, the minimum costs for the chain of length 1.

Algorithm of Matrix Chain Multiplication

MATRIX-CHAIN-ORDER (p)

1. $n \leftarrow \text{length}[p]-1$
2. for $i \leftarrow 1$ to n
3. do $m[i, i] \leftarrow 0$
4. for $l \leftarrow 2$ to n // l is the chain length
5. do for $i \leftarrow 1$ to $n-l+1$
6. do $j \leftarrow i+l-1$
7. $m[i, j] \leftarrow \infty$
8. for $k \leftarrow i$ to $j-1$
9. do $q \leftarrow m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$
10. If $q < m[i, j]$



11. then $m[i,j] \leftarrow q$

12. $s[i,j] \leftarrow k$

13. return m and s .

PRINT-OPTIMAL-PARENS (s, i, j)

1. if $i=j$

2. then print "A"

3. else print "("

4. PRINT-OPTIMAL-PARENS ($s, i, s[i, j]$)

5. PRINT-OPTIMAL-PARENS ($s, s[i, j] + 1, j$)

6. print ")"



Longest Common Subsequence

In this tutorial, you will learn how the longest common subsequence is found. Also, you will find working examples of the longest common subsequence in C, C++, Java and Python.

The longest common subsequence (LCS) is defined as the longest subsequence that is common to all the given sequences, provided that the elements of the subsequence are not required to occupy consecutive positions within the original sequences.

If $S1$ and $S2$ are the two given sequences then, Z is the common subsequence of $S1$ and $S2$ if Z is a subsequence of both $S1$ and $S2$. Furthermore, Z must be a **strictly increasing sequence** of the indices of both $S1$ and $S2$.

In a strictly increasing sequence, the indices of the elements chosen from the original sequences must be in ascending order in Z .

If

$S1 = \{B, C, D, A, A, C, D\}$

Then, $\{A, D, B\}$ cannot be a subsequence of $S1$ as the order of the elements is not the same (ie. not strictly increasing sequence).

Let us understand LCS with an example.

If

$S1 = \{B, C, D, A, A, C, D\}$

$S2 = \{A, C, D, B, A, C\}$

Then, common subsequences are $\{B, C\}$, $\{C, D, A, C\}$, $\{D, A, C\}$, $\{A, A, C\}$, $\{A, C\}$, $\{C, D\}$, ...

Among these subsequences, $\{C, D, A, C\}$ is the longest common subsequence. We are going to find this longest common subsequence using dynamic programming.

Before proceeding further, if you do not already know about dynamic programming, please go through [dynamic programming](#).

Using Dynamic Programming to find the LCS

Let us take two sequences:

X



The first sequence

Y

C	B	D	A
---	---	---	---

Second Sequence

The following steps are followed for finding the longest common subsequence.

1. Create a table of dimension $n+1*m+1$ where n and m are the lengths of X and Y respectively. The first row and the first column are filled with zeros.

		C	B	D	A
	0	0	0	0	0
A	0				
C	0				
A	0				
D	0				
B	0				

Initialise a table

2. Fill each cell of the table using the following logic.
3. If the character corresponding to the current row and current column are matching, then fill the current cell by adding one to the diagonal element. Point an arrow to the diagonal cell.
4. Else take the maximum value from the previous column and previous row element for filling the current cell. Point an arrow to the cell with maximum value. If they are equal,

		C	B	D	A
	0	0	0	0	0
A	0	0	0	0	1
C	0				
A	0				
D	0				
B	0				

point to any of them.

Fill the values

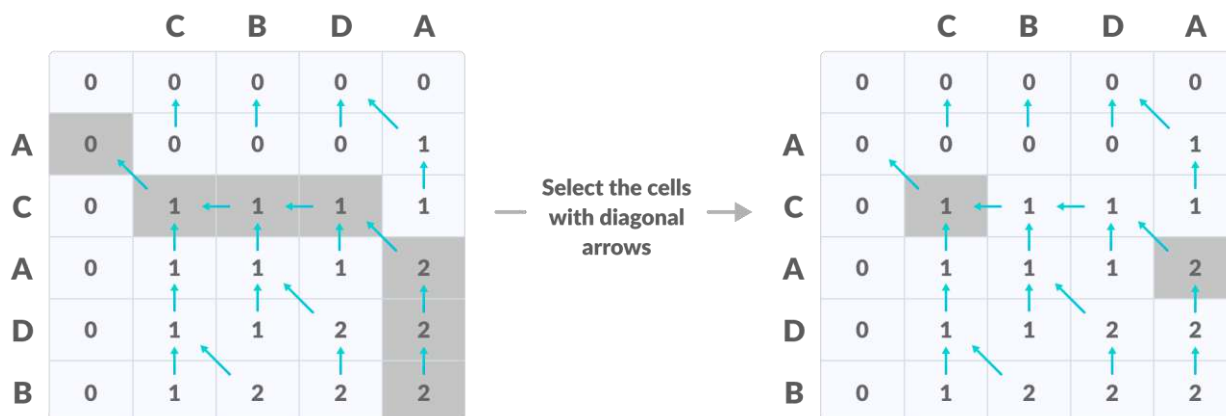
5. **Step 2** is repeated until the table is filled. Fill all the values

	C	B	D	A
A	0	0	0	0
C	0	1	1	1
A	0	1	1	2
D	0	1	1	2
B	0	1	2	2

6. The value in the last row and the last column is the length of the longest common subsequence. The bottom right corner is the length of the LCS

	C	B	D	A
A	0	0	0	0
C	0	1	1	1
A	0	1	1	2
D	0	1	1	2
B	0	1	2	2

7. In order to find the longest common subsequence, start from the last element and follow the direction of the arrow. The elements corresponding to () symbol form the longest common subsequence. Create a path according to the arrows



Thus, the longest common subsequence is CA.



LCS

How is a dynamic programming algorithm more efficient than the recursive algorithm while solving an LCS problem?

The method of dynamic programming reduces the number of function calls. It stores the result of each function call so that it can be used in future calls without the need for redundant calls.

In the above dynamic algorithm, the results obtained from each comparison between elements of X and the elements of Y are stored in a table so that they can be used in future computations.

So, the time taken by a dynamic approach is the time taken to fill the table (ie. $O(mn)$). Whereas, the recursion algorithm has the complexity of $2^{\max(m, n)}$.

Longest Common Subsequence Algorithm

X and Y be two given sequences

Initialize a table LCS of dimension $X.length * Y.length$

$X.label = X$

$Y.label = Y$

$LCS[0][] = 0$

$LCS[][0] = 0$

Start from $LCS[1][1]$

Compare $X[i]$ and $Y[j]$

If $X[i] = Y[j]$

$LCS[i][j] = 1 + LCS[i-1, j-1]$

Point an arrow to $LCS[i][j]$

Else

$LCS[i][j] = \max(LCS[i-1][j], LCS[i][j-1])$

Point an arrow to $\max(LCS[i-1][j], LCS[i][j-1])$



0/1 Knapsack problem

Here knapsack is like a container or a bag. Suppose we have given some items which have some weights or profits. We have to put some items in the knapsack in such a way total value produces a maximum profit.

For example, the weight of the container is 20 kg. We have to select the items in such a way that the sum of the weight of items should be either smaller than or equal to the weight of the container, and the profit should be maximum.

There are two types of knapsack problems:

- 0/1 knapsack problem
- Fractional knapsack problem

We will discuss both the problems one by one. First, we will learn about the 0/1 knapsack problem.

x

What is the 0/1 knapsack problem?

The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack. For example, we have two items having weights 2kg and 3kg, respectively. If we pick the 2kg item then we cannot pick 1kg item from the 2kg item (item is not divisible); we have to pick the 2kg item completely. This is a 0/1 knapsack problem in which either we pick the item completely or we will pick that item. The 0/1 knapsack problem is solved by the dynamic programming.

What is the fractional knapsack problem?

The fractional knapsack problem means that we can divide the item. For example, we have an item of 3 kg then we can pick the item of 2 kg and leave the item of 1 kg. The fractional knapsack problem is solved by the Greedy approach.

Example of 0/1 knapsack problem.

Consider the problem having weights and profits are:

Weights: {3, 4, 6, 5}

Profits: {2, 3, 1, 4}

The weight of the knapsack is 8 kg

The number of items is 4

The above problem can be solved by using the following method:

$x_i = \{1, 0, 0, 1\}$

$= \{0, 0, 0, 1\}$

$= \{0, 1, 0, 1\}$

The above are the possible combinations. 1 denotes that the item is completely picked and 0 means



that no item is picked. Since there are 4 items so possible combinations will be:

$2^4 = 16$; So. There are 16 possible combinations that can be made by using the above problem. Once all the combinations are made, we have to select the combination that provides the maximum profit.

Another approach to solve the problem is dynamic programming approach. In dynamic programming approach, the complicated problem is divided into sub-problems, then we find the solution of a sub-problem and the solution of the sub-problem will be used to find the solution of a complex problem.

How this problem can be solved by using the Dynamic programming approach?

First,

we create a matrix shown as below:

	0	1	2	3	4	5	6	7	8
0									
1									
2									
3									
4									

In the above matrix, columns represent the weight, i.e., 8. The rows represent the profits and weights of items. Here we have not taken the weight 8 directly, problem is divided into sub-problems, i.e., 0, 1, 2, 3, 4, 5, 6, 7, 8. The solution of the sub-problems would be saved in the cells and answer to the problem would be stored in the final cell. First, we write the weights in the ascending order and profits according to their weights shown as below:

$w_i = \{3, 4, 5, 6\}$

$p_i = \{2, 3, 4, 1\}$

The first row and the first column would be 0 as there is no item for $w=0$

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0								
2	0								
3	0								
4	0								



When $i=1$, $W=1$

$w_1 = 3$; Since we have only one item in the set having weight 3, but the capacity of the knapsack is 1. We cannot fill the item of 3kg in the knapsack of capacity 1 kg so add 0 at $M[1][1]$ shown as below:

0 1 2 3 4 5 6 7 8

0 0 0 0 0 0 0 0 0 0

1 0 0

2 0

3 0

4 0

When $i = 1$, $W = 2$

$w_1 = 3$; Since we have only one item in the set having weight 3, but the capacity of the knapsack is 2. We cannot fill the item of 3kg in the knapsack of capacity 2 kg so add 0 at $M[1][2]$ shown as below:

0 1 2 3 4 5 6 7 8

0 0 0 0 0 0 0 0 0 0

1 0 0 0

2 0

3 0

4 0

When $i=1$, $W=3$

$w_1 = 3$; Since we have only one item in the set having weight equal to 3, and weight of the knapsack is also 3; therefore, we can fill the knapsack with an item of weight equal to 3. We put profit corresponding to the weight 3, i.e., 2 at $M[1][3]$ shown as below:

0 1 2 3 4 5 6 7 8

0 0 0 0 0 0 0 0 0 0

1 0 0 0 2

2 0

3 0



4 0

When $i=1$, $W = 4$

$w_1 = 3$; Since we have only one item in the set having weight equal to 3, and weight of the knapsack is 4; therefore, we can fill the knapsack with an item of weight equal to 3. We put profit corresponding to the weight 3, i.e., 2 at $M[1][4]$ shown as below:

0 1 2 3 4 5 6 7 8

0 0 0 0 0 0 0 0 0 0

1 0 0 0 2 2

2 0

3 0

4 0

When $i=1$, $W = 5$

$w_1 = 3$; Since we have only one item in the set having weight equal to 3, and weight of the knapsack is 5; therefore, we can fill the knapsack with an item of weight equal to 3. We put profit corresponding to the weight 3, i.e., 2 at $M[1][5]$ shown as below:

0 1 2 3 4 5 6 7 8

0 0 0 0 0 0 0 0 0 0

1 0 0 0 2 2 2

2 0

3 0

4 0

When $i = 1$, $W=6$

$w_1 = 3$; Since we have only one item in the set having weight equal to 3, and weight of the knapsack is 6; therefore, we can fill the knapsack with an item of weight equal to 3. We put profit corresponding to the weight 3, i.e., 2 at $M[1][6]$ shown as below:

0 1 2 3 4 5 6 7 8

0 0 0 0 0 0 0 0 0 0

1 0 0 0 2 2 2 2

2 0



3 0

4 0

When $i=1$, $W = 7$

$w_1 = 3$; Since we have only one item in the set having weight equal to 3, and weight of the knapsack is 7; therefore, we can fill the knapsack with an item of weight equal to 3. We put profit corresponding to the weight 3, i.e., 2 at $M[1][7]$ shown as below:

0 1 2 3 4 5 6 7 8

0 0 0 0 0 0 0 0 0 0

1 0 0 0 2 2 2 2 2

2 0

3 0

4 0

When $i =1$, $W =8$

$w_1 = 3$; Since we have only one item in the set having weight equal to 3, and weight of the knapsack is 8; therefore, we can fill the knapsack with an item of weight equal to 3. We put profit corresponding to the weight 3, i.e., 2 at $M[1][8]$ shown as below:

0 1 2 3 4 5 6 7 8

0 0 0 0 0 0 0 0 0 0

1 0 0 0 2 2 2 2 2 2

2 0

3 0

4 0

Now the value of 'i' gets incremented, and becomes 2.

When $i =2$, $W = 1$

The weight corresponding to the value 2 is 4, i.e., $w_2 = 4$. Since we have only one item in the set having weight equal to 4, and the weight of the knapsack is 1. We cannot put the item of weight 4 in a knapsack, so we add 0 at $M[2][1]$ shown as below:

0 1 2 3 4 5 6 7 8

0 0 0 0 0 0 0 0 0 0

1 0 0 0 2 2 2 2 2 2



2 0 0

3 0

4 0

When $i = 2$, $W = 2$

The weight corresponding to the value 2 is 4, i.e., $w_2 = 4$. Since we have only one item in the set having weight equal to 4, and the weight of the knapsack is 2. We cannot put the item of weight 4 in a knapsack, so we add 0 at $M[2][2]$ shown as below:

0 1 2 3 4 5 6 7 8

0 0 0 0 0 0 0 0 0 0

1 0 0 0 2 2 2 2 2 2

2 0 0 0

3 0

4 0

When $i = 2$, $W = 3$

The weight corresponding to the value 2 is 4, i.e., $w_2 = 4$. Since we have two items in the set having weights 3 and 4, and the weight of the knapsack is 3. We can put the item of weight 3 in a knapsack, so we add 2 at $M[2][3]$ shown as below:

0 1 2 3 4 5 6 7 8

0 0 0 0 0 0 0 0 0 0

1 0 0 0 2 2 2 2 2 2

2 0 0 0 2

3 0

4 0

When $i = 2$, $W = 4$

The weight corresponding to the value 2 is 4, i.e., $w_2 = 4$. Since we have two items in the set having weights 3 and 4, and the weight of the knapsack is 4. We can put item of weight 4 in a knapsack as the profit corresponding to weight 4 is more than the item having weight 3, so we add 3 at $M[2][4]$ shown as below:

0 1 2 3 4 5 6 7 8

0 0 0 0 0 0 0 0 0 0



1 0 0 0 2 2 2 2 2 2

2 0 0 0 2 3

3 0

4 0

When $i = 2$, $W = 5$

The weight corresponding to the value 2 is 4, i.e., $w_2 = 4$. Since we have two items in the set having weights 3 and 4, and the weight of the knapsack is 5. We can put item of weight 4 in a knapsack and the profit corresponding to weight is 3, so we add 3 at $M[2][5]$ shown as below:

0 1 2 3 4 5 6 7 8

0 0 0 0 0 0 0 0 0 0

1 0 0 0 2 2 2 2 2 2

2 0 0 0 2 3 3

3 0

4 0

When $i = 2$, $W = 6$

The weight corresponding to the value 2 is 4, i.e., $w_2 = 4$. Since we have two items in the set having weights 3 and 4, and the weight of the knapsack is 6. We can put item of weight 4 in a knapsack and the profit corresponding to weight is 3, so we add 3 at $M[2][6]$ shown as below:

0 1 2 3 4 5 6 7 8

0 0 0 0 0 0 0 0 0 0

1 0 0 0 2 2 2 2 2 2

2 0 0 0 2 3 3 3

3 0

4 0

When $i = 2$, $W = 7$

The weight corresponding to the value 2 is 4, i.e., $w_2 = 4$. Since we have two items in the set having weights 3 and 4, and the weight of the knapsack is 7. We can put item of weight 4 and 3 in a knapsack and the profits corresponding to weights are 2 and 3; therefore, the total profit is 5, so we add 5 at $M[2][7]$ shown as below:

0 1 2 3 4 5 6 7 8



0 0 0 0 0 0 0 0 0 0

1 0 0 0 2 2 2 2 2 2

2 0 0 0 0 3 3 3 5

3 0

4 0

When $i = 2$, $W = 8$

The weight corresponding to the value 2 is 4, i.e., $w_2 = 4$. Since we have two items in the set having weights 3 and 4, and the weight of the knapsack is 7. We can put item of weight 4 and 3 in a knapsack and the profits corresponding to weights are 2 and 3; therefore, the total profit is 5, so we add 5 at $M[2][7]$ shown as below:

0 1 2 3 4 5 6 7 8

0 0 0 0 0 0 0 0 0 0

1 0 0 0 2 2 2 2 2 2

2 0 0 0 2 3 3 3 5 5

3 0

4 0

Now the value of 'i' gets incremented, and becomes 3.

When $i = 3$, $W = 1$

The weight corresponding to the value 3 is 5, i.e., $w_3 = 5$. Since we have three items in the set having weights 3, 4, and 5, and the weight of the knapsack is 1. We cannot put neither of the items in a knapsack, so we add 0 at $M[3][1]$ shown as below:

0 1 2 3 4 5 6 7 8

0 0 0 0 0 0 0 0 0 0

1 0 0 0 2 2 2 2 2 2

2 0 0 0 2 3 3 3 5 5

3 0 0

4 0

When $i = 3$, $W = 2$

The weight corresponding to the value 3 is 5, i.e., $w_3 = 5$. Since we have three items in the set having weight 3, 4, and 5, and the weight of the knapsack is 1. We cannot put neither of the items in



a knapsack, so we add 0 at $M[3][2]$ shown as below:

0 1 2 3 4 5 6 7 8

0 0 0 0 0 0 0 0 0 0

1 0 0 0 2 2 2 2 2 2

2 0 0 0 2 3 3 3 5 5

3 0 0 0

4 0

When $i = 3$, $W = 3$

The weight corresponding to the value 3 is 5, i.e., $w_3 = 5$. Since we have three items in the set of weight 3, 4, and 5 respectively and weight of the knapsack is 3. The item with a weight 3 can be put in the knapsack and the profit corresponding to the item is 2, so we add 2 at $M[3][3]$ shown as below:

0 1 2 3 4 5 6 7 8

0 0 0 0 0 0 0 0 0 0

1 0 0 0 2 2 2 2 2 2

2 0 0 0 2 3 3 3 5 5

3 0 0 0 2

4 0

When $i = 3$, $W = 4$

The weight corresponding to the value 3 is 5, i.e., $w_3 = 5$. Since we have three items in the set of weight 3, 4, and 5 respectively, and weight of the knapsack is 4. We can keep the item of either weight 3 or 4; the profit (3) corresponding to the weight 4 is more than the profit corresponding to the weight 3 so we add 3 at $M[3][4]$ shown as below:

0 1 2 3 4 5 6 7 8

0 0 0 0 0 0 0 0 0 0

1 0 0 0 2 2 2 2 2 2

2 0 0 0 2 3 3 3 5 5

3 0 0 0 1 3

4 0

When $i = 3$, $W = 5$



The weight corresponding to the value 3 is 5, i.e., $w_3 = 5$. Since we have three items in the set of weight 3, 4, and 5 respectively, and weight of the knapsack is 5. We can keep the item of either weight 3, 4 or 5; the profit (3) corresponding to the weight 4 is more than the profits corresponding to the weight 3 and 5 so we add 3 at $M[3][5]$ shown as below:

```
0 1 2 3 4 5 6 7 8
0 0 0 0 0 0 0 0 0
1 0 0 0 2 2 2 2 2
2 0 0 0 2 3 3 3 5
3 0 0 0 1 3 3
4 0
```

When $i = 3$, $W = 6$

The weight corresponding to the value 3 is 5, i.e., $w_3 = 5$. Since we have three items in the set of weight 3, 4, and 5 respectively, and weight of the knapsack is 6. We can keep the item of either weight 3, 4 or 5; the profit (3) corresponding to the weight 4 is more than the profits corresponding to the weight 3 and 5 so we add 3 at $M[3][6]$ shown as below:

```
0 1 2 3 4 5 6 7 8
0 0 0 0 0 0 0 0 0
1 0 0 0 2 2 2 2 2
2 0 0 0 2 3 3 3 5
3 0 0 0 1 3 3
4 0
```

When $i = 3$, $W = 7$

The weight corresponding to the value 3 is 5, i.e., $w_3 = 5$. Since we have three items in the set of weight 3, 4, and 5 respectively, and weight of the knapsack is 7. In this case, we can keep both the items of weight 3 and 4, the sum of the profit would be equal to $(2 + 3)$, i.e., 5, so we add 5 at $M[3][7]$ shown as below:

```
0 1 2 3 4 5 6 7 8
0 0 0 0 0 0 0 0 0
1 0 0 0 2 2 2 2 2
2 0 0 0 2 3 3 3 5
```



3 0 0 0 1 3 3 3 5

4 0

When $i = 3$, $W = 8$

The weight corresponding to the value 3 is 5, i.e., $w_3 = 5$. Since we have three items in the set of weight 3, 4, and 5 respectively, and the weight of the knapsack is 8. In this case, we can keep both the items of weight 3 and 4, the sum of the profit would be equal to $(2 + 3)$, i.e., 5, so we add 5 at $M[3][8]$ shown as below:

0 1 2 3 4 5 6 7 8

0 0 0 0 0 0 0 0 0 0

1 0 0 0 2 2 2 2 2 2

2 0 0 0 2 3 3 3 5 5

3 0 0 0 1 3 3 3 5 5

4 0

Now the value of 'i' gets incremented and becomes 4.

When $i = 4$, $W = 1$

The weight corresponding to the value 4 is 6, i.e., $w_4 = 6$. Since we have four items in the set of weights 3, 4, 5, and 6 respectively, and the weight of the knapsack is 1. The weight of all the items is more than the weight of the knapsack, so we cannot add any item in the knapsack; Therefore, we add 0 at $M[4][1]$ shown as below:

0 1 2 3 4 5 6 7 8

0 0 0 0 0 0 0 0 0 0

1 0 0 0 2 2 2 2 2 2

2 0 0 0 2 3 3 3 5 5

3 0 0 0 1 3 3 3 5 5

4 0 0

When $i = 4$, $W = 2$

The weight corresponding to the value 4 is 6, i.e., $w_4 = 6$. Since we have four items in the set of weights 3, 4, 5, and 6 respectively, and the weight of the knapsack is 2. The weight of all the items is more than the weight of the knapsack, so we cannot add any item in the knapsack; Therefore, we add 0 at $M[4][2]$ shown as below:

0 1 2 3 4 5 6 7 8



0 0 0 0 0 0 0 0 0 0

1 0 0 0 2 2 2 2 2 2

2 0 0 0 2 3 3 3 5 5

3 0 0 0 1 3 3 3 5 5

4 0 0 0

When $i = 4$, $W = 3$

The weight corresponding to the value 4 is 6, i.e., $w_4 = 6$. Since we have four items in the set of weights 3, 4, 5, and 6 respectively, and the weight of the knapsack is 3. The item with a weight 3 can be put in the knapsack and the profit corresponding to the weight 4 is 2, so we will add 2 at $M[4][3]$ shown as below:

0 1 2 3 4 5 6 7 8

0 0 0 0 0 0 0 0 0 0

1 0 0 0 2 2 2 2 2 2

2 0 0 0 2 3 3 3 5 5

3 0 0 0 1 3 3 3 5 5

4 0 0 0 2

When $i = 4$, $W = 4$

The weight corresponding to the value 4 is 6, i.e., $w_4 = 6$. Since we have four items in the set of weights 3, 4, 5, and 6 respectively, and the weight of the knapsack is 4. The item with a weight 4 can be put in the knapsack and the profit corresponding to the weight 4 is 3, so we will add 3 at $M[4][4]$ shown as below:

0 1 2 3 4 5 6 7 8

0 0 0 0 0 0 0 0 0 0

1 0 0 0 2 2 2 2 2 2

2 0 0 0 2 3 3 3 5 5

3 0 0 0 1 3 3 3 5 5

4 0 0 0 2 3

When $i = 4$, $W = 5$

The weight corresponding to the value 4 is 6, i.e., $w_4 = 6$. Since we have four items in the set of weights 3, 4, 5, and 6 respectively, and the weight of the knapsack is 5. The item with a weight 4



can be put in the knapsack and the profit corresponding to the weight 4 is 3, so we will add 3 at $M[4][5]$ shown as below:

```
0 1 2 3 4 5 6 7 8
0 0 0 0 0 0 0 0 0
1 0 0 0 2 2 2 2 2
2 0 0 0 2 3 3 3 5
3 0 0 0 1 3 3 3 5
4 0 0 0 2 3 3
```

When $i = 4$, $W = 6$

The weight corresponding to the value 4 is 6, i.e., $w_4 = 6$. Since we have four items in the set of weights 3, 4, 5, and 6 respectively, and the weight of the knapsack is 6. In this case, we can put the items in the knapsack either of weight 3, 4, 5 or 6 but the profit, i.e., 4 corresponding to the weight 6 is highest among all the items; therefore, we add 4 at $M[4][6]$ shown as below:

```
0 1 2 3 4 5 6 7 8
0 0 0 0 0 0 0 0 0
1 0 0 0 2 2 2 2 2
2 0 0 0 2 3 3 3 5
3 0 0 0 1 3 3 3 5
4 0 0 0 2 3 3 4
```

When $i = 4$, $W = 7$

The weight corresponding to the value 4 is 6, i.e., $w_4 = 6$. Since we have four items in the set of weights 3, 4, 5, and 6 respectively, and the weight of the knapsack is 7. Here, if we add two items of weights 3 and 4 then it will produce the maximum profit, i.e., $(2 + 3)$ equals to 5, so we add 5 at $M[4][7]$ shown as below:

```
0 1 2 3 4 5 6 7 8
0 0 0 0 0 0 0 0 0
1 0 0 0 2 2 2 2 2
2 0 0 0 2 3 3 3 5
3 0 0 0 2 3 3 3 5
4 0 0 0 2 3 3 4 5
```

When $i = 4$, $W = 8$

The weight corresponding to the value 4 is 6, i.e., $w_4 = 6$. Since we have four items in the set of weights 3, 4, 5, and 6 respectively, and the weight of the knapsack is 8. Here, if we add two items of weights 3 and 4 then it will produce the maximum profit, i.e., $(2 + 3)$ equals to 5, so we add 5 at $M[4][8]$ shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2	3	3	3	5	5
4	0	0	0	2	3	3	4	5	5

As we can observe in the above table that 5 is the maximum profit among all the entries. The pointer points to the last row and the last column having 5 value. Now we will compare 5 value with the previous row; if the previous row, i.e., $i = 3$ contains the same value 5 then the pointer will shift upwards. Since the previous row contains the value 5 so the pointer will be shifted upwards as shown in the below table:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2	3	3	3	5	5
4	0	0	0	2	3	3	4	5	5

Again, we will compare the value 5 from the above row, i.e., $i = 2$. Since the above row contains the value 5 so the pointer will again be shifted upwards as shown in the below table:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2	3	3	3	5	5
4	0	0	0	2	3	3	4	5	5



Again, we will compare the value 5 from the above row, i.e., $i = 1$. Since the above row does not contain the same value so we will consider the row $i=1$, and the weight corresponding to the row is 4. Therefore, we have selected the weight 4 and we have rejected the weights 5 and 6 shown below:

$$x = \{1, 0, 0\}$$

The profit corresponding to the weight is 3. Therefore, the remaining profit is $(5 - 3)$ equals to 2. Now we will compare this value 2 with the row $i = 2$. Since the row ($i = 1$) contains the value 2; therefore, the pointer shifted upwards shown below:

```
0 1 2 3 4 5 6 7 8
0 0 0 0 0 0 0 0 0
1 0 0 0 2 2 2 2 2
2 0 0 0 2 3 3 3 5 5
3 0 0 0 2 3 3 3 5 5
4 0 0 0 2 3 3 4 5 5
```

Again we compare the value 2 with a above row, i.e., $i = 1$. Since the row $i = 0$ does not contain the value 2, so row $i = 1$ will be selected and the weight corresponding to the $i = 1$ is 3 shown below:

$$X = \{1, 1, 0, 0\}$$

The profit corresponding to the weight is 2. Therefore, the remaining profit is 0. We compare 0 value with the above row. Since the above row contains a 0 value but the profit corresponding to this row is 0. In this problem, two weights are selected, i.e., 3 and 4 to maximize the profit.



Lower Bound Theory

Lower Bound Theory Concept is based upon the calculation of minimum time that is required to execute an algorithm is known as a lower bound theory or Base Bound Theory.

Lower Bound Theory uses a number of methods/techniques to find out the lower bound.

Concept/Aim: The main aim is to calculate a minimum number of comparisons required to execute an algorithm.

Techniques:

The techniques which are used by lower Bound Theory are:

1. Comparisons Trees.
2. Oracle and adversary argument
3. State Space Method

1. Comparison trees:

In a comparison sort, we use only comparisons between elements to gain order information about an input sequence ($a_1; a_2, \dots, a_n$).

Given a_i, a_j from (a_1, a_2, \dots, a_n) We Perform One of the Comparisons

- $a_i < a_j$ less than
- $a_i \leq a_j$ less than or equal to
- $a_i > a_j$ greater than
- $a_i \geq a_j$ greater than or equal to
- $a_i = a_j$ equal to

To determine their relative order, if we assume all elements are distinct, then we just need to consider $a_i \leq a_j$ '=' is excluded & $\geq, \leq, >, <$ are equivalent.

Consider sorting three numbers a_1, a_2 , and a_3 . There are $3! = 6$ possible combinations:

1. $(a_1, a_2, a_3), (a_1, a_3, a_2),$
2. $(a_2, a_1, a_3), (a_2, a_3, a_1)$
3. $(a_3, a_1, a_2), (a_3, a_2, a_1)$

The Comparison based algorithm defines a decision tree.

Decision Tree: A decision tree is a full binary tree that shows the comparisons between elements that are executed by an appropriate sorting algorithm operating on an input of a given size. Control, data movement, and all other conditions of the algorithm are ignored.

In a decision tree, there will be an array of length n .

So, total leaves will be $n!$ (I.e. total number of comparisons)

If tree height is h , then surely

$$n! \leq 2^h \text{ (tree will be binary)}$$

Taking an Example of comparing a_1, a_2 , and a_3 .

Left subtree will be true condition i.e. $a_i \leq a_j$

Right subtree will be false condition i.e. $a_i > a_j$

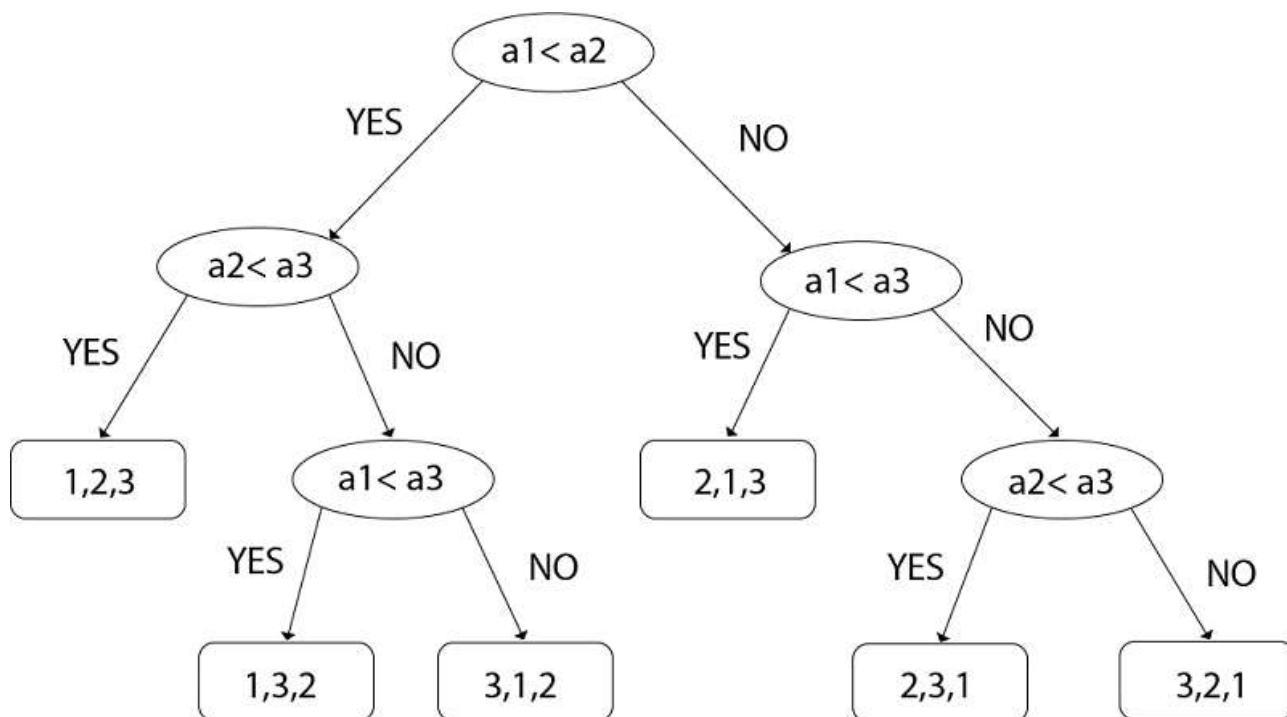


Fig: Decision Tree

So from above, we got

$$n! \leq 2^h$$

Taking Log both sides

$$\log n! \leq h \log 2$$

$$h \log 2 \geq \log n!$$

$$h \geq \log_2 n!$$

$$h \geq \log_2 [1, 2, 3, \dots, n]$$

$$h \geq \log_2 1 + \log_2 2 + \log_2 3 + \dots + \log_2 n$$

$$h \geq \sum_{i=1}^n \log_2 i$$

$$h \geq \int_1^n \log_2 i - 1 \, di$$

$$h \geq \log_2 i \cdot x^0 \int_1^n - \int_1^n \frac{1}{i} x^i \, di$$

$$h \geq n \log_2 n - \int_1^n 1 \, di$$

$$h \geq n \log_2 n - i \int_1^n$$

$$h \geq n \log_2 n - n + 1$$

Ignoring the Constant terms

$$h \geq n \log_2 n$$

$$h = \pi n (\log n)$$

Comparison tree for Binary Search:

Example: Suppose we have a list of items according to the following Position:

1. 1,2,3,4,5,6,7,8,9,10,11,12,13,14

$$\text{Mid} = \left\lfloor \left(\frac{1+14}{2} \right) \right\rfloor = \frac{15}{2} = 7.5 = 7$$

Note: Choose the greatest integer

1, 2, 3, 4, 5, 6	8, 9, 10, 11, 12, 13, 14
$\text{Mid} = \left(\frac{1+6}{2} \right) = 3$	$\text{Mid} = \left(\frac{8+14}{2} \right) = 11$

1, 2	4, 5, 6	8, 9, 10	12, 13, 14
$\text{Mid} = \left(\frac{1+2}{2} \right) = 1$	$\text{Mid} = \left(\frac{4+6}{2} \right) = 5$	$\text{Mid} = \left(\frac{8+10}{2} \right) = 9$	$\text{Mid} = \left(\frac{12+14}{2} \right) = 13$

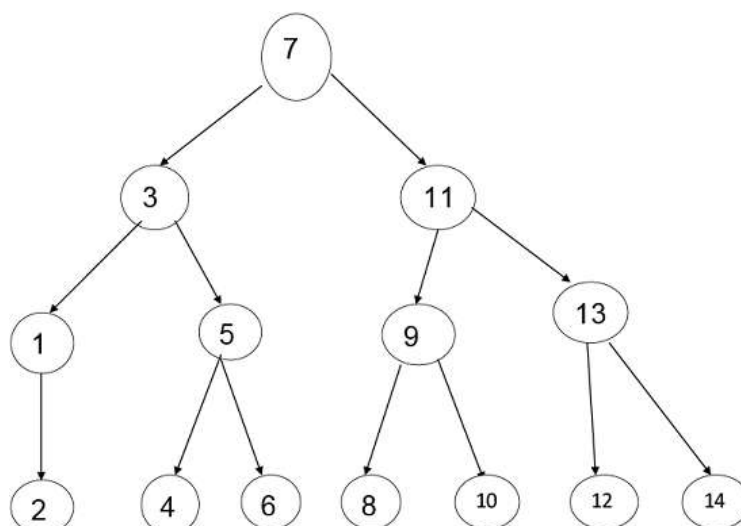
And the last midpoint is:

- 2, 4, 6, 8, 10, 12, 14

Thus, we will consider all the midpoints and we will make a tree of it by having stepwise midpoints.

The Bold letters are Mid-Points Here

According to Mid-Point, the tree will be:



Step1: Maximum number of nodes up to k level of the internal node is $2^k - 1$

For Example

$$2^k - 1$$



$$2^3 - 1 = 8 - 1 = 7$$

Where $k = \text{level} = 3$

Step2: Maximum number of internal nodes in the comparisons tree is $n!$

Note: Here Internal Nodes are Leaves.

Step3: From Condition 1 & Condition 2 we get

$$N! \leq 2^k - 1$$

$$14 < 15$$

Where $N = \text{Nodes}$

Step4: Now, $n+1 \leq 2^k$

Here, Internal Nodes will always be less than 2^k in the Binary Search.

Step5:

$$n+1 \leq 2^k$$

$$\log(n+1) = k \log 2$$

$$k \geq \frac{\log(n+1)}{\log 2}$$

$$k \geq \log_2(n+1)$$

Step6:

$$1. T(n) = k$$

Step7:

$$T(n) \geq \log_2(n+1)$$

Here, the minimum number of Comparisons to perform a task of the search of n terms using Binary Search

2. Oracle and adversary argument:

Another technique for obtaining lower bounds consists of making use of an "oracle."

Given some model of estimation such as comparison trees, the oracle tells us the outcome of each comparison.



In order to derive a good lower bound, the oracle efforts it's finest to cause the algorithm to work as hard as it might.

It does this by deciding as the outcome of the next analysis, the result which matters the most work to be needed to determine the final answer.

And by keeping step of the work that is finished, a worst-case lower bound for the problem can be derived.

Example: (Merging Problem) given the sets A (1: m) and B (1: n), where the information in A and in B are sorted. Consider lower bounds for algorithms combining these two sets to give an individual sorted set.

Consider that all of the $m+n$ elements are specific and $A(1) < A(2) < \dots < A(m)$ and $B(1) < B(2) < \dots < B(n)$.

Elementary combinatory tells us that there are $C((m+n), n)$ ways that the A's and B's may merge together while still preserving the ordering within A and B.

Thus, if we need comparison trees as our model for combining algorithms, then there will be $C((m+n), n)$ external nodes and therefore at least $\log C((m+n), n)$ comparisons are needed by any comparison-based merging algorithm.

If we let $MERGE(m, n)$ be the minimum number of comparisons used to merge m items with n items then we have the inequality

$$1. \log C((m+n), n) \leq MERGE(m, n) \leq m+n-1.$$

The upper bound and lower bound can get promptly far apart as m gets much smaller than n .

3. State Space Method:

1. State Space Method is a set of rules that show the possible states (n -tuples) that an algorithm can assume from a given state of a single comparison.
2. Once the state transitions are given, it is possible to derive lower bounds by arguing that the finished state cannot be reached using any fewer transitions.
3. Given n distinct items, find winner and loser.
4. Aim: When state changed count it that is the aim of State Space Method.



5. In this approach, we will count the number of comparison by counting the number of changes in state.

6. Analysis of the problem to find out the smallest and biggest items by using the state space method.

7. State: It is a collection of attributes.

8. Through this we sort out two types of Problems:

1. Find the largest & smallest element from an array of elements.
2. To find out largest & second largest elements from an array of an element.

9. For the largest item, we need 7 comparisons and what will be the second largest item?

Now we count those teams who lose the match with team A

Teams are: B, D, and E

So the total no of comparisons are: 7

Let n is the total number of items, then

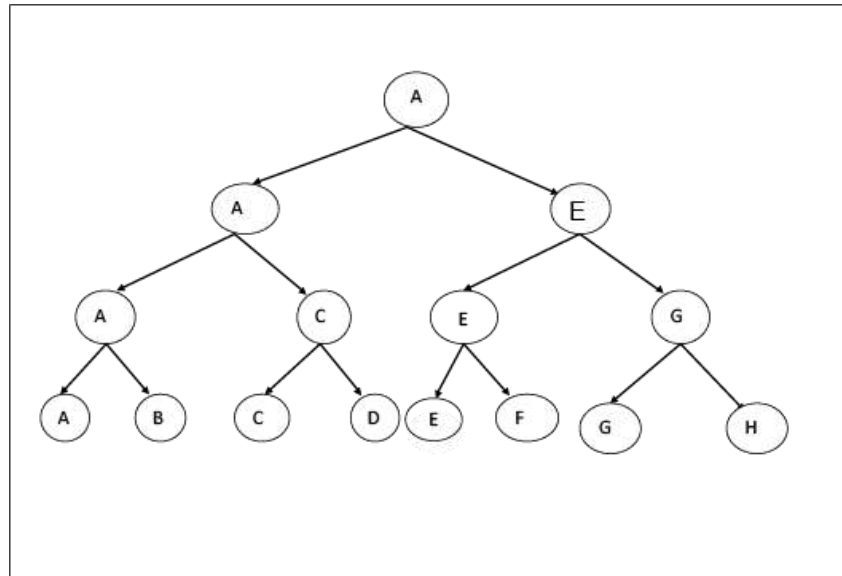
Comparisons = $n-1$ (to find the biggest item)

No of Comparisons to find out the 2nd biggest item = $\log_2 n - 1$

10. In this **no of comparisons** are equal to the **number of changes of states** during the execution of the algorithm.

Example: State (A, B, C, D)

1. No of items that can never be compared.
2. No of items that win but never lost (ultimate winner).
3. No of items that lost but never win (ultimate loser)
4. No of items who sometimes lost & sometimes win.



Firstly, A, B compare out or match between them. A wins and in C, D.C wins and so on. We can assume that B wins and so on. We can assume that B wins in place of A it can be anything depending on our self.

Phase-1:

A	B	C	D
---	---	---	---

8 0 0 0

No match occurs; there are total 8 external nodes or 8 teams

6 1 1 0

Match between A & B, 6 left in A state as only one match is done between A & B. 1 add to B and 1 add to C .

4 2 2 0

C & D match out 4 left out 1 for C wins and 1 for D's lost and 0 for there is no team that sometimes wins or lost.

2 3 3 0

Match between E & F

0 4 4 0

Match between E & F and Match between G & H are compare out.

In Phase-1 there are 4 states

If the team is 8 then 4 states

As if n team the $n/2$ states.

Phase-2

A	B	C	D
---	---	---	---

0 4 4 0

Started at point where Phase 1 ends. Initial States for Phase-2

0 3 4 1

A & C both wins from lower level. But now A wins and C lost. But, C wins from lower level so C will be counted as sometimes wins & sometimes lost.

0 2 4 2

E & G are match E wins but G lost but G wins from lower level. It will be counted as sometimes wins & sometimes lost.

0 1 4 3

A & C both wins at Phase-1 but now A wins and C lost. So, C wins before and last now A will come in state- D

4 is Constant in C-State as B, D, F, H are lost teams that never win.

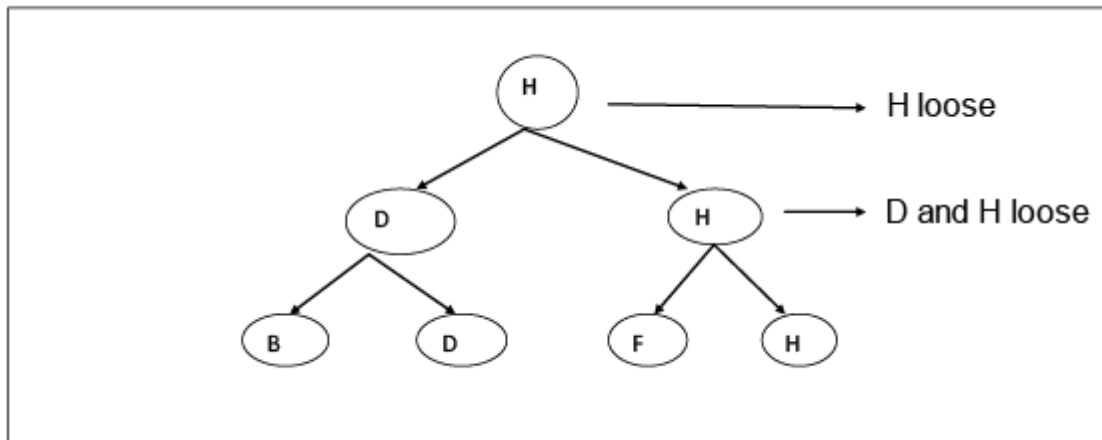
Thus there are 3 states in Phase-2,

If n is 8 then states are 3

If n teams that are $\left(\frac{n}{2} - 1\right)$ states.

Phase-3: This is a Phase in which teams which come under C-State are considered and there will be matches between them to find out the team which is never winning at all.

In this Structure, we are going to move upward for denoting who is not winning after the match.



Here H is the team which is never winning at all. By this, we fulfill our second aim to.

A	B	C	D
---	---	---	---

0 1 4 3

0 1 3 4

0 1 2 5

0 1 1 6

Here by match out of B & D .We find that B wins and D loses But B at lower level loses. It will be counted as sometimes wins and lost and it will be removed from lost team. ∴ 4 will turn to 3 from lost team.

After match out F & H, F wins and H loses but F loses at lower level ∴ F will be counted as sometimes wins & lost also removed from lost team thus 2 turn to 1 from lost team.

Match between F & H

After match out of D & H.H wins and D lost but H loses from lower level ∴ it will be counted as sometimes wins & lost also removed from lost team thus 2 turn to 1 from lost team.

Thus there are 3 states in Phase -3

If n teams that are $\left(\frac{n}{2} - 1\right)$ states

Note: the total of all states value is always equal to 'n'.

Thus, by adding all phase's states we will get:-

Phase1 + Phase 2 + Phase 3

$$\frac{n}{2} + \left(\frac{n}{2} - 1\right) + \left(\frac{n}{2}\right) - 1$$

$$\frac{n}{2} + \frac{n}{2} - 1 + \frac{n}{2} - 1$$

$$\frac{n+n-2+n-2}{2} = \frac{3n-4}{2}$$

$$\frac{3n-2}{2}$$

Suppose we have 8 teams then $\left[3\left(\frac{8}{2}\right) - 2\right] = 10$ states are there (as minimum) to find out which one is never wins team.

Thus, Equation is:

$$T(n) = \frac{3n-2}{2}$$

Lower bound ($L(n)$) is a property of the particular issue i.e. the sorting problem, matrix multiplication not of any particular algorithm solving that problem.

Lower bound theory says that no calculation can carry out the activity in less than that of ($L(n)$) times the units for arbitrary inputs i.e. that for every comparison based sorting algorithm must take at least $L(n)$ time in the worst case.

$L(n)$ is the base overall conceivable calculation which is greatest finished.

Trivial lower bounds are utilized to yield the bound best alternative is to count the number of elements in the problems input that must be prepared and the number of output items that need to be produced.

The lower bound theory is the method that has been utilized to establish the given algorithm in the most efficient way which is possible. This is done by discovering a function $g(n)$ that is a lower bound on the time that any algorithm must take to solve the given problem. Now if we have an algorithm whose computing time is the same order as $g(n)$, then we know that asymptotically we cannot do better.

If $f(n)$ is the time for some algorithm, then we write $f(n) = \Omega(g(n))$ to mean that $g(n)$ is the **lower bound of $f(n)$** . This equation can be formally written, if there exists positive constants c and n_0 such that $|f(n)| \geq c|g(n)|$ for all $n > n_0$. In addition for developing lower bounds within the



constant factor, we are more conscious of the fact to determine more exact bounds whenever this is possible.

Deriving good **lower bounds** is more challenging than arrange efficient algorithms. This happens because a lower bound states a fact about all possible algorithms for solving a problem. Generally, we cannot enumerate and analyze all these algorithms, so lower bound proofs are often hard to obtain.



N-Queens Problem

N - Queens problem is to place n - queens in such a manner on an n x n chessboard that no queens attack each other by being in the same row, column or diagonal.

It can be seen that for n =1, the problem has a trivial solution, and no solution exists for n =2 and n =3. So first we will consider the 4 queens problem and then generate it to n - queens problem.

Given a 4 x 4 chessboard and number the rows and column of the chessboard 1 through 4.

	1	2	3	4
1				
2				
3				
4				

4x4 chessboard

Since, we have to place 4 queens such as q_1 q_2 q_3 and q_4 on the chessboard, such that no two queens attack each other. In such a conditional each queen must be placed on a different row, i.e., we put queen "i" on row "i."

Now, we place queen q_1 in the very first acceptable position (1, 1). Next, we put queen q_2 so that both these queens do not attack each other. We find that if we place q_2 in column 1 and 2, then the dead end is encountered. Thus the first acceptable position for q_2 in column 3, i.e. (2, 3) but then no position is left for placing queen ' q_3 ' safely. So we backtrack one step and place the queen ' q_2 ' in (2, 4), the next best possible solution. Then we obtain the position for placing ' q_3 ' which is (3, 2). But later this position also leads to a dead end, and no place is found where ' q_4 ' can be placed safely. Then we have to backtrack till ' q_1 ' and place it to (1, 2) and then all other queens are placed safely by moving q_2 to (2, 4), q_3 to (3, 1) and q_4 to (4, 3). That is, we get the solution (2, 4, 1, 3). This is one possible solution for the 4-queens problem. For another possible solution, the whole method is repeated for all partial solutions. The other solutions for 4 - queens problems is (3, 1, 4, 2) i.e.

	1	2	3	4
1			q_1	
2	q_2			
3				q_3
4		q_4		

The implicit tree for 4 - queen problem for a solution (2, 4, 1, 3) is as follows:

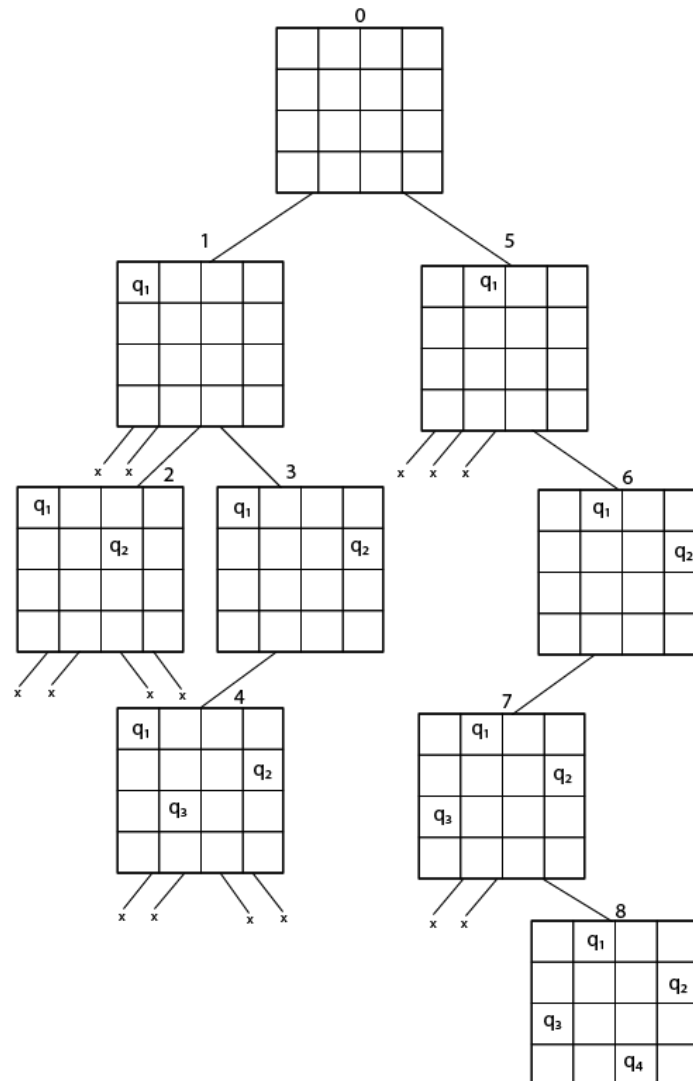
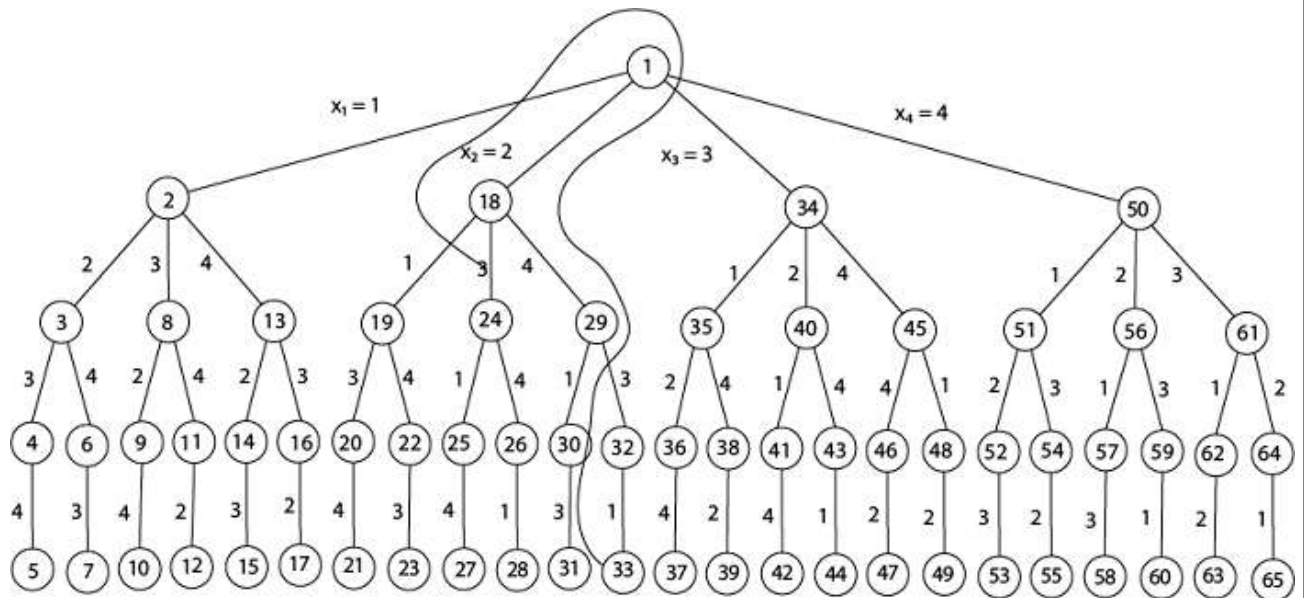


Fig shows the complete state space for 4 - queens problem. But we can use backtracking method to

generate the necessary node and stop if the next node violates the rule, i.e., if two queens are attacking.



4 - Queens solution space with nodes numbered in DFS

It can be seen that all the solutions to the 4 queens problem can be represented as 4 - tuples (x_1, x_2, x_3, x_4) where x_i represents the column on which queen " q_i " is placed.

One possible solution for 8 queens problem is shown in fig:

	1	2	3	4	5	6	7	8
1				q_1				
2						q_2		
3								q_3
4		q_4						
5						q_5		
6	q_6							
7			q_7					
8					q_8			

1. Thus, the solution for 8 -queen problem for (4, 6, 8, 2, 7, 1, 3, 5).
2. If two queens are placed at position (i, j) and (k, l) .
3. Then they are on same diagonal only if $(i - j) = k - l$ or $i + j = k + l$.



4. The first equation implies that $j - 1 = i - k$.
5. The second equation implies that $j - 1 = k - i$.
6. Therefore, two queens lie on the duplicate diagonal if and only if $|j - i| = |i - k|$

Place (k, i) returns a Boolean value that is true if the kth queen can be placed in column i. It tests both whether i is distinct from all previous costs x_1, x_2, \dots, x_{k-1} and whether there is no other queen on the same diagonal.

Using place, we give a precise solution to then n- queens problem.

1. Place (k, i)
2. {
3. For $j \leftarrow 1$ to $k - 1$
4. do if $(x[j] = i)$
5. or $(\text{Abs } x[j] - i) = (\text{Abs } (j - k))$
6. then return false;
7. return true;
8. }

Place (k, i) return true if a queen can be placed in the kth row and ith column otherwise return is false.

$x[]$ is a global array whose final $k - 1$ values have been set. Abs (r) returns the absolute value of r.

1. N - Queens (k, n)
2. {
3. For $i \leftarrow 1$ to n
4. do if Place (k, i) then
5. {
6. $x[k] \leftarrow i$;
7. if $(k == n)$ then
8. write $(x[1 \dots n])$;



9. else
10. N - Queens (k + 1, n);
11. }
12. }



Naive Pattern Searching

Naïve pattern searching is the simplest method among other pattern searching algorithms. It checks for all character of the main string to the pattern. This algorithm is helpful for smaller texts. It does not need any pre-processing phases. We can find substring by checking once for the string. It also does not occupy extra space to perform the operation.

The time complexity of Naïve Pattern Search method is $O(m*n)$. The m is the size of pattern and n is the size of the main string.

Input and Output

Input:

Main String: "ABAAABCDBBABCDDDEBCABC", pattern: "ABC"

Output:

Pattern found at position: 4

Pattern found at position: 10

Pattern found at position: 18

Algorithm

naivePatternSearch(pattern, text)

Input – The text and the pattern

Output – location, where patterns are present in the text

Begin

patLen := pattern Size

strLen := string size

for $i := 0$ to $(strLen - patLen)$, do

for $j := 0$ to $patLen$, do

if $text[i+j] \neq pattern[j]$, then

break the loop

done



if j == patLen, then

display the position i, as there pattern found

done

End

Example

```
#include<iostream>
```

```
using namespace std;
```

```
void naivePatternSearch(string mainString, string pattern, int array[], int *index) {
```

```
    int patLen = pattern.size();
```

```
    int strLen = mainString.size();
```

```
    for(int i = 0; i<=(strLen - patLen); i++) {
```

```
        int j;
```

```
        for(j = 0; j<patLen; j++) {    //check for each character of pattern if it is matched
```

```
            if(mainString[i+j] != pattern[j])
```

```
                break;
```

```
        }
```

```
        if(j == patLen) {    //the pattern is found
```

```
            (*index)++;
```

```
            array[(*index)] = i;
```

```
        }
```

```
    }
```

```
}
```



```
int main() {  
  
    string mainString = "ABAAABCDBBABCDDDEBCABC";  
  
    string pattern = "ABC";  
  
    int locArray[mainString.size()];  
  
    int index = -1;  
  
    naivePatternSearch(mainString, pattern, locArray, &index);  
  
    for(int i = 0; i <= index; i++) {  
        cout << "Pattern found at position: " << locArray[i]<<endl;  
    }  
}
```

Output

Pattern found at position: 4

Pattern found at position: 10

Pattern found at position: 18



The Rabin-Karp-Algorithm

The Rabin-Karp string matching algorithm calculates a hash value for the pattern, as well as for each M-character subsequences of text to be compared. If the hash values are unequal, the algorithm will determine the hash value for next M-character sequence. If the hash values are equal, the algorithm will analyze the pattern and the M-character sequence. In this way, there is only one comparison per text subsequence, and character matching is only required when the hash values match.

RABIN-KARP-MATCHER (T, P, d, q)

1. $n \leftarrow \text{length}[T]$
2. $m \leftarrow \text{length}[P]$
3. $h \leftarrow d^{m-1} \bmod q$
4. $p \leftarrow 0$
5. $t_0 \leftarrow 0$
6. for $i \leftarrow 1$ to m
7. do $p \leftarrow (dp + P[i]) \bmod q$
8. $t_0 \leftarrow (dt_0 + T[i]) \bmod q$
9. for $s \leftarrow 0$ to $n-m$
10. do if $p = t_s$
11. then if $P[1.....m] = T[s+1.....s+m]$
12. then "Pattern occurs with shift" s
13. If $s < n-m$
14. then $t_{s+1} \leftarrow (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$

Example: For string matching, working module $q = 11$, how many spurious hits does the Rabin-Karp matcher encounters in Text $T = 31415926535.....$

1. $T = 31415926535.....$
2. $P = 26$
3. Here $T.Length = 11$ so $Q = 11$

4. And $P \bmod Q = 26 \bmod 11 = 4$

5. Now find the exact match of $P \bmod Q$...

Solution:

T =

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

P =

2	6
---	---

S = 0 →

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$31 \bmod 11 = 9$ not equal to 4

S = 1 →

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$14 \bmod 11 = 3$ not equal to 4

S = 2 →

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$41 \bmod 11 = 8$ not equal to 4

S = 3 →

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$15 \bmod 11 = 4$ equal to 4 SPURIOUS HIT

$S = 4$ →

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$59 \bmod 11 = 4$ equal to 4 SPURIOUS HIT

$S = 5$ →

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$92 \bmod 11 = 4$ equal to 4 SPURIOUS HIT

$S = 6$ →

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$26 \bmod 11 = 4$ EXACT MATCH

$S = 7$ →

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---



S = 7

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$65 \bmod 11 = 10$ not equal to 4

S = 8

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$53 \bmod 11 = 9$ not equal to 4

S = 9

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$35 \bmod 11 = 2$ not equal to 4

The Pattern occurs with shift 6.

Complexity:

The running time of **RABIN-KARP-MATCHER** in the worst case scenario $O((n-m+1)m)$ but it has a good average case running time. If the expected number of strong shifts is small $O(1)$ and prime q is chosen to be quite large, then the Rabin-Karp algorithm can be expected to run in time $O(n+m)$ plus the time to require to process spurious hits.



The Knuth-Morris-Pratt (KMP) Algorithm

Knuth-Morris and Pratt introduce a linear time algorithm for the string matching problem. A matching time of $O(n)$ is achieved by avoiding comparison with an element of 'S' that have previously been involved in comparison with some element of the pattern 'p' to be matched. i.e., backtracking on the string 'S' never occurs

Components of KMP Algorithm:

1. The Prefix Function (Π): The Prefix Function, Π for a pattern encapsulates knowledge about how the pattern matches against the shift of itself. This information can be used to avoid a useless shift of the pattern 'p.' In other words, this enables avoiding backtracking of the string 'S.'

2. The KMP Matcher: With string 'S,' pattern 'p' and prefix function ' Π ' as inputs, find the occurrence of 'p' in 'S' and returns the number of shifts of 'p' after which occurrences are found.

The Prefix Function (Π)

Following pseudo code compute the prefix function, Π :

COMPUTE- PREFIX- FUNCTION (P)

1. $m \leftarrow \text{length}[P]$ // 'p' pattern to be matched
2. $\Pi[1] \leftarrow 0$
3. $k \leftarrow 0$
4. for $q \leftarrow 2$ to m
5. do while $k > 0$ and $P[k+1] \neq P[q]$
6. do $k \leftarrow \Pi[k]$
7. If $P[k+1] = P[q]$
8. then $k \leftarrow k+1$
9. $\Pi[q] \leftarrow k$
10. Return Π

Running Time Analysis:

In the above pseudo code for calculating the prefix function, the for loop from step 4 to step 10 runs 'm' times. Step 1 to Step 3 take constant time. Hence the running time of computing prefix function is $O(m)$.

Example: Compute Π for the pattern 'p' below:

P :

a	b	a	b	a	c	a
---	---	---	---	---	---	---

Solution:

Initially: $m = \text{length}[p] = 7$

$$\Pi[1] = 0$$

$$k = 0$$

Step 1: $q = 2, k = 0$

$$\Pi[2] = 0$$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0					

Step 2: $q = 3, k = 0$

$$\Pi[3] = 1$$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1				

Step3: $q = 4, k = 1$

$$\Pi[4] = 2$$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	A
Π	0	0	1	2			

Step4: $q = 5, k = 2$

$$\pi[5] = 3$$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3		

Step5: $q = 6, k = 3$

$$\pi[6] = 0$$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3	0	

Step6: $q = 7, k = 1$

$$\pi[7] = 1$$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3	0	1

After iteration 6 times, the prefix function computation is complete:

q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
π	0	0	1	2	3	0	1

The KMP Matcher:

The KMP Matcher with the pattern 'p,' the string 'S' and prefix function ' π ' as input, finds a match of p in S. Following pseudo code compute the matching component of KMP algorithm:

KMP-MATCHER (T, P)

1. $n \leftarrow \text{length}[T]$
2. $m \leftarrow \text{length}[P]$
3. $\pi \leftarrow \text{COMPUTE-PREFIX-FUNCTION}(P)$
4. $q \leftarrow 0$ // numbers of characters matched
5. for $i \leftarrow 1$ to n // scan S from left to right

6. do while $q > 0$ and $P[q + 1] \neq T[i]$
7. do $q \leftarrow \Pi[q]$ // next character does not match
8. If $P[q + 1] = T[i]$
9. then $q \leftarrow q + 1$ // next character matches
10. If $q = m$ // is all of p matched?
11. then print "Pattern occurs with shift" $i - m$
12. $q \leftarrow \Pi[q]$ // look for the next match

Running Time Analysis:

The for loop beginning in step 5 runs 'n' times, i.e., as long as the length of the string 'S.' Since step 1 to step 4 take constant times, the running time is dominated by this for the loop. Thus running time of the matching function is $O(n)$.

Example: Given a string 'T' and pattern 'P' as follows:

T:

b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

P:

a	b	a	b	a	c	a
---	---	---	---	---	---	---

Let us execute the KMP Algorithm to find whether 'P' occurs in 'T.'

For 'p' the prefix function, π was computed previously and is as follows:

q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
π	0	0	1	2	3	0	1

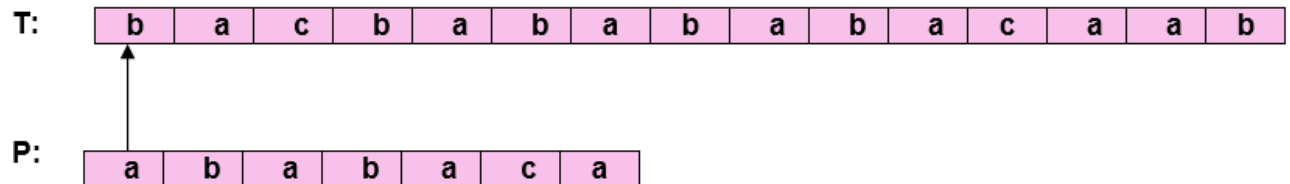
Solution:

Initially: $n = \text{size of } T = 15$

$m = \text{size of } P = 7$

Step1: $i=1, q=0$

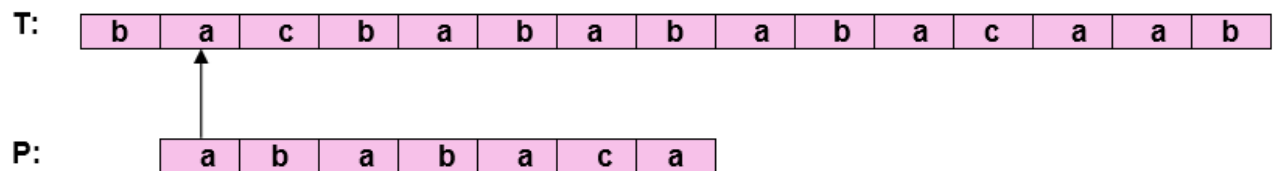
Comparing P [1] with T [1]



P [1] does not match with T [1]. 'p' will be shifted one position to the right.

Step2: $i = 2, q = 0$

Comparing P [1] with T [2]

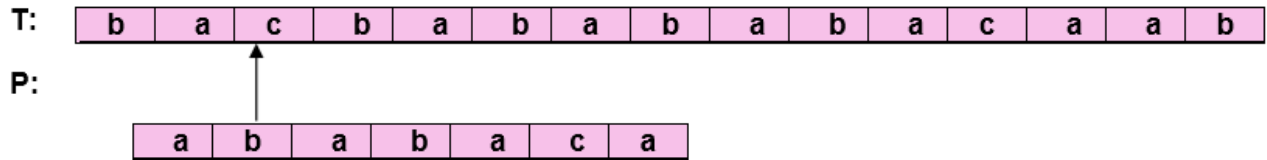


P [1] matches T [2]. Since there is a match, p is not shifted.

Step 3: $i = 3, q = 1$

Comparing P [2] with T [3]

P [2] doesn't match with T [3]

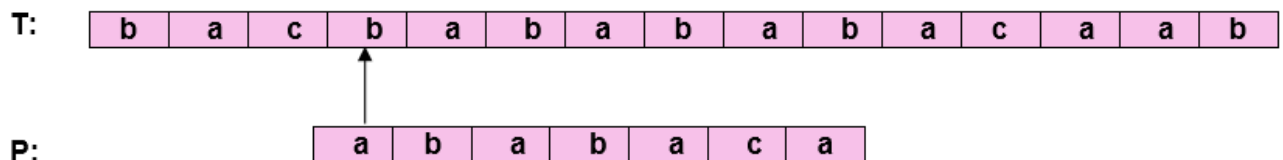


Backtracking on p, Comparing P [1] and T [3]

Step4: $i = 4, q = 0$

Comparing P [1] with T [4]

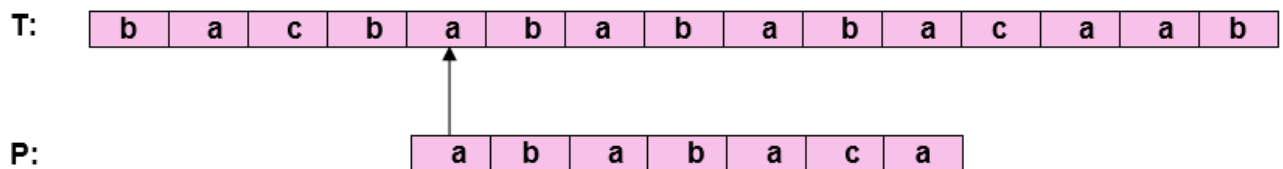
P [1] doesn't match with T [4]



Step5: $i = 5, q = 0$

Comparing P [1] with T [5]

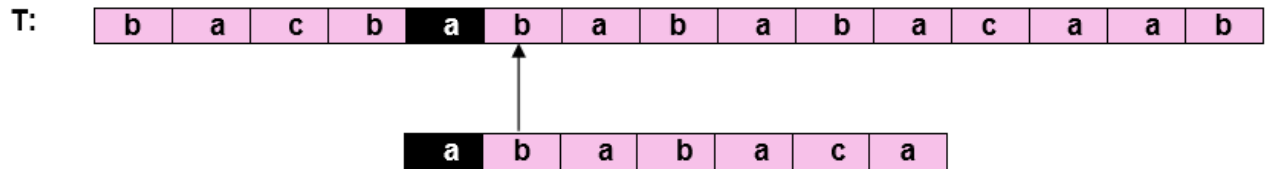
P [1] match with T [5]



Step6: $i = 6, q = 1$

Comparing P [2] with T [6]

P [2] matches with T [6]

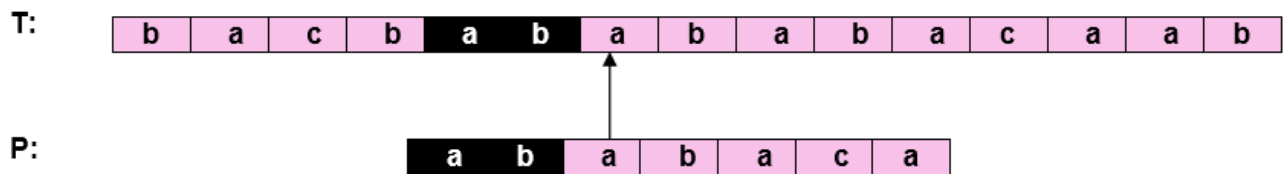


P:

Step7: $i = 7, q = 2$

Comparing P [3] with T [7]

P [3] matches with T [7]

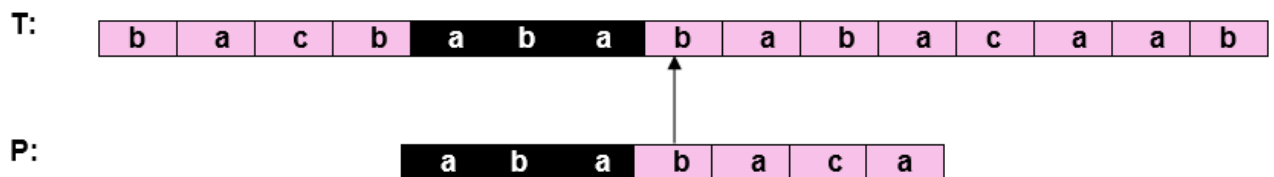


P:

Step8: $i = 8, q = 3$

Comparing P [4] with T [8]

P [4] matches with T [8]

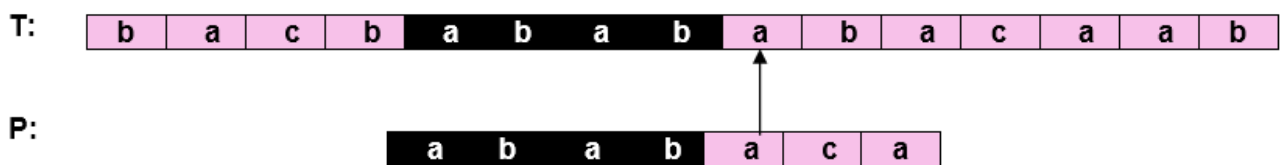


P:

Step9: $i = 9, q = 4$

Comparing P [5] with T [9]

P [5] matches with T [9]

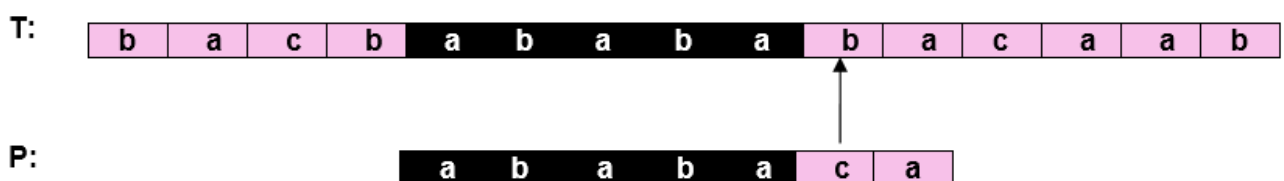


P:

Step10: $i = 10, q = 5$

Comparing P [6] with T [10]

P [6] doesn't match with T [10]



P:

Backtracking on p, Comparing P [4] with T [10] because after mismatch $q = \pi [5] = 3$

Step11: $i = 11, q = 4$

Comparing P [5] with T [11]

P [5] match with T [11]

T:

b a c b a b a b a c a a b

P:

a b a b a c a

Step12: $i = 12, q = 5$

Comparing P [6] with T [12]

P [6] matches with T [12]

T:

b a c b a b a b a c a a b

P:

a b a b a c a

Step13: $i = 3, q = 6$

Comparing P [7] with T [13]

P [7] matches with T [13]

T:

b a c b a b a b a c a a a b

P:

a b a b a c a

Pattern 'P' has been found to complexity occur in a string 'T.' The total number of shifts that took place for the match to be found is $i - m = 13 - 7 = 6$ shifts.

Unit-4

Job Assignment Problem using Branch And Bound

Let there be N workers and N jobs. Any worker can be assigned to perform any job, incurring some cost that may vary depending on the work-job assignment. It is required to perform all jobs by assigning exactly one worker to each job and exactly one job to each agent in such a way that the total cost of the assignment is minimized.

	Job 1	Job 2	Job 3	Job 4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

Worker A takes 8 units of time to finish job 4.

An example job assignment problem. Green values show optimal job assignment that is A-Job2, B-Job1, C-Job3 and D-Job4

Let us explore all approaches for this problem.

Solution 1: Brute Force

We generate $n!$ possible job assignments and for each such assignment, we compute its total cost and return the less expensive assignment. Since the solution is a permutation of the n jobs, its complexity is $O(n!)$.

Solution 2: Hungarian Algorithm

The optimal assignment can be found using the Hungarian algorithm. The Hungarian algorithm has worst case run-time complexity of $O(n^3)$.

Solution 3: DFS/BFS on state space tree

A state space tree is a N-ary tree with property that any path from root to leaf node holds one of many solutions to given problem. We can perform depth-first search on state space tree and but successive moves can take us away from the goal rather than bringing closer. The search of state space tree follows leftmost path from the root regardless of initial state. An answer node may never be found in this approach. We can also perform a Breadth-first search on state space tree. But no matter what the initial state is, the algorithm attempts the same sequence of moves like DFS.



Solution 4: Finding Optimal Solution using Branch and Bound

The selection rule for the next node in BFS and DFS is “blind”. i.e. the selection rule does not give any preference to a node that has a very good chance of getting the search to an answer node quickly. The search for an optimal solution can often be speeded by using an “intelligent” ranking function, also called an approximate cost function to avoid searching in sub-trees that do not contain an optimal solution. It is similar to BFS-like search but with one major optimization. Instead of following FIFO order, we choose a live node with least cost. We may not get optimal solution by following node with least promising cost, but it will provide very good chance of getting the search to an answer node quickly.

There are two approaches to calculate the cost function:

1. For each worker, we choose job with minimum cost from list of unassigned jobs (take minimum entry from each row).
2. For each job, we choose a worker with lowest cost for that job from list of unassigned workers (take minimum entry from each column).

In this article, the first approach is followed.

Let's take below example and try to calculate promising cost when Job 2 is assigned to worker A.

	Job 1	Job 2	Job 3	Job 4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

Since Job 2 is assigned to worker A (marked in green), cost becomes 2 and Job 2 and worker A becomes unavailable (marked in red).

	Job 1	Job 2	Job 3	Job 4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

Now we assign job 3 to worker B as it has minimum cost from list of unassigned jobs. Cost becomes $2 + 3 = 5$ and Job 3 and worker B also becomes unavailable.

	Job 1	Job 2	Job 3	Job 4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

Finally, job 1 gets assigned to worker C as it has minimum cost among unassigned jobs and job 4 gets assigned to worker D as it is only Job left. Total cost becomes $2 + 3 + 5 + 4 = 14$.

	Job 1	Job 2	Job 3	Job 4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

Below diagram shows complete search space diagram showing optimal solution path in green.

Level 0

ROOT

Level 1

A -> 1
Cost = 24

A -> 2
Cost = 14

A -> 3
Cost = 20

A -> 4
Cost = 22

Level 2

B -> 1
Cost = 13

B -> 3
Cost = 14

B -> 4
Cost = 17

Level 3

C -> 3
D -> 4
Cost = 13

C -> 4
D -> 3
Cost = 25



Randomized Algorithms | Set 1 (Introduction and Analysis)

What is a Randomized Algorithm?

An algorithm that uses random numbers to decide what to do next anywhere in its logic is called a Randomized Algorithm. For example, in Randomized Quick Sort, we use a random number to pick the next pivot (or we randomly shuffle the array). And in Karger's algorithm, we randomly pick an edge.

How to analyse Randomized Algorithms?

Some randomized algorithms have deterministic time complexity. For example, this implementation of Karger's algorithm has time complexity is $O(E)$. Such algorithms are called Monte Carlo Algorithms and are easier to analyse for worst case. On the other hand, time complexity of other randomized algorithms (other than Las Vegas) is dependent on value of random variable. Such Randomized algorithms are called Las Vegas Algorithms. These algorithms are typically analysed for expected worst case. To compute expected time taken in worst case, all possible values of the used random variable needs to be considered in worst case and time taken by every possible value needs to be evaluated. Average of all evaluated times is the expected worst case time complexity. Below facts are generally helpful in analysis of such algorithms.

Linearity of Expectation

Expected Number of Trials until Success.

For example consider below a randomized version of QuickSort.

A **Central Pivot** is a pivot that divides the array in such a way that one side has at-least $1/4$ elements.

// Sorts an array arr[low..high]

randQuickSort(arr[], low, high)

1. If $low \geq high$, then EXIT.

2. While pivot 'x' is not a Central Pivot.



- (i) Choose uniformly at random a number from [low..high].

Let the randomly picked number be x .

- (ii) Count elements in arr[low..high] that are smaller than arr[x]. Let this count be sc .

- (iii) Count elements in arr[low..high] that are greater than arr[x]. Let this count be gc .

- (iv) Let $n = (high-low+1)$. If $sc \geq n/4$ and $gc \geq n/4$, then x is a central pivot.

3. Partition arr[low..high] around the pivot x .

4. // Recur for smaller elements

randQuickSort(arr, low, sc-1)

5. // Recur for greater elements

randQuickSort(arr, high-gc+1, high)

The important thing in our analysis is, time taken by step 2 is $O(n)$.

How many times while loop runs before finding a central pivot?

The probability that the randomly chosen element is central pivot is $1/n$.

Therefore, expected number of times the while loop runs is n

Thus, the expected time complexity of step 2 is $O(n)$.

What is overall Time Complexity in Worst Case?

In worst case, each partition divides array such that one side has $n/4$ elements and other side has $3n/4$ elements. The worst case height of recursion tree is $\log_{3/4} n$ which is $O(\log n)$.

$$T(n) < T(n/4) + T(3n/4) + O(n)$$

$$T(n) < 2T(3n/4) + O(n)$$



Solution of above recurrence is $O(n \log n)$

Note that the above randomized algorithm is not the best way to implement randomized Quick Sort.

The idea here is to simplify the analysis as it is simple to analyse.



Randomized Algorithms | Set 2 (Classification and Applications)

We strongly recommend to refer below post as a prerequisite of this. Randomized Algorithms | Set 1 (Introduction and Analysis)

Classification

Randomized algorithms are classified in two categories. **Las Vegas:** These algorithms always produce correct or optimum result. Time complexity of these algorithms is based on a random value and time complexity is evaluated as expected value. For example, Randomized QuickSort always sorts an input array and expected worst case time complexity of QuickSort is $O(n \log n)$. **Monte Carlo:** Produce correct or optimum result with some probability. These algorithms have deterministic running time and it is generally easier to find out worst case time complexity. For example this implementation of Karger's Algorithm produces minimum cut with probability greater than or equal to $1/n^2$ (n is number of vertices) and has worst case time complexity as $O(E)$. Another example is Fermat Method for Primality Testing. **Example to Understand Classification:**

Consider a binary array where exactly half elements are 0 and half are 1. The task is to find index of any 1.

A Las Vegas algorithm for this task is to keep picking a random element until we find a 1. A Monte Carlo algorithm for the same is to keep picking a random element until we either find 1 or we have tried maximum allowed times say k . The Las Vegas algorithm always finds an index of 1, but time complexity is determined as expected value. The expected number of trials before success is 2, therefore expected time complexity is $O(1)$. The Monte Carlo Algorithm finds a 1 with probability $[1 - (1/2)^k]$. Time complexity of Monte Carlo is $O(k)$ which is deterministic

Applications and Scope:

- Consider a tool that basically does sorting. Let the tool be used by many users and there are few users who always use tool for already sorted array. If the tool uses simple (not randomized) QuickSort, then those few users are always going to face worst case situation. On the other hand if the tool uses Randomized QuickSort, then there is no user that always gets worst case. Everybody gets expected $O(n \log n)$ time.
- Randomized algorithms have huge applications in Cryptography.
- Load Balancing.



- Number-Theoretic Applications: Primality Testing
- Data Structures: Hashing, Sorting, Searching, Order Statistics and Computational Geometry.
- Algebraic identities: Polynomial and matrix identity verification. Interactive proof systems.
- Mathematical programming: Faster algorithms for linear programming, Rounding linear program solutions to integer program solutions
- Graph algorithms: Minimum spanning trees, shortest paths, minimum cuts.
- Counting and enumeration: Matrix permanent Counting combinatorial structures.
- Parallel and distributed computing: Deadlock avoidance distributed consensus.
- Probabilistic existence proofs: Show that a combinatorial object arises with non-zero probability among objects drawn from a suitable probability space.
- Derandomization: First devise a randomized algorithm then argue that it can be derandomized to yield a deterministic algorithm.

Randomized algorithms are algorithms that use randomness as a key component in their operation. They can be used to solve a wide variety of problems, including optimization, search, and decision-making. Some examples of applications of randomized algorithms include:

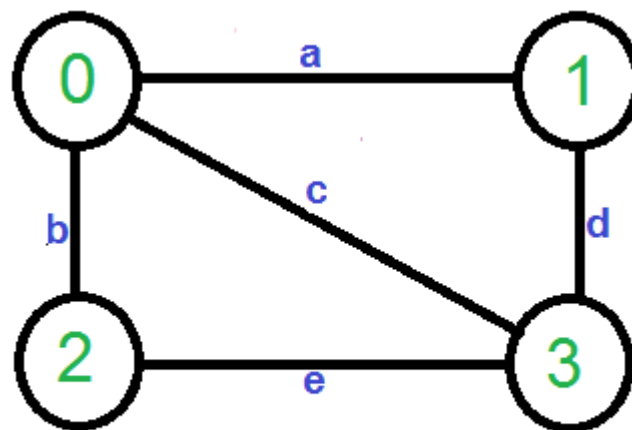
1. Monte Carlo methods: These are a class of randomized algorithms that use random sampling to solve problems that may be deterministic in principle, but are too complex to solve exactly. Examples include estimating pi, simulating physical systems, and solving optimization problems.
2. Randomized search algorithms: These are algorithms that use randomness to search for solutions to problems. Examples include genetic algorithms and simulated annealing.
3. Randomized data structures: These are data structures that use randomness to improve their performance. Examples include skip lists and hash tables.
4. Randomized load balancing: These are algorithms used to distribute load across a network of computers, using randomness to avoid overloading any one computer.
5. Randomized encryption: These are algorithms used to encrypt and decrypt data, using randomness to make it difficult for an attacker to decrypt the data without the correct key.

Introduction and implementation of Karger's algorithm for Minimum Cut

Given an undirected and unweighted graph, find the smallest cut (smallest number of edges that disconnects the graph into two components).

The input graph may have parallel edges.

For example consider the following example, the smallest cut has 2 edges.



Min-Cut for above graph is either {a, d} OR {b, e}

A Simple Solution use Max-Flow based s-t cut algorithm to find minimum cut. Consider every pair of vertices as source 's' and sink 't', and call minimum s-t cut algorithm to find the s-t cut. Return minimum of all s-t cuts. Best possible time complexity of this algorithm is $O(V^5)$ for a graph.

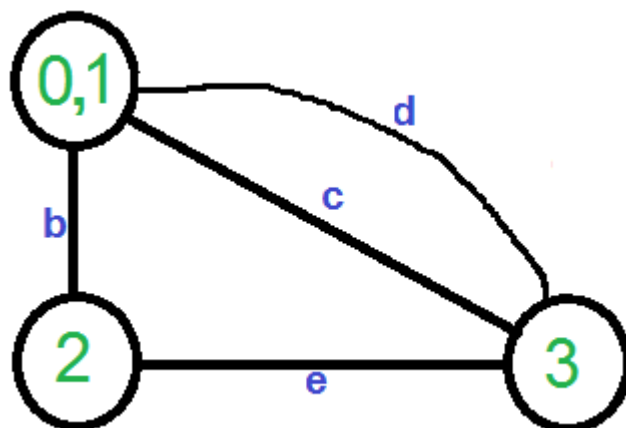
[How? there are total possible V^2 pairs and s-t cut algorithm for one pair takes $O(V \cdot E)$ time and $E = O(V^2)$].

Below is simple Karger's Algorithm for this purpose. Below Karger's algorithm can be implemented in $O(E) = O(V^2)$ time.

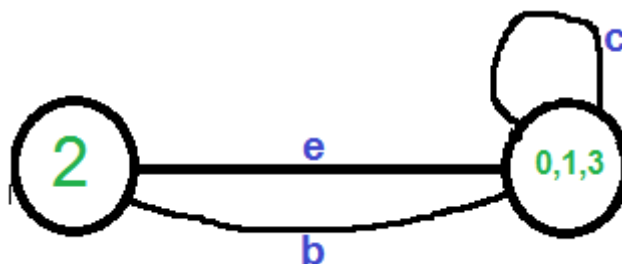
- 1) Initialize contracted graph CG as copy of original graph
- 2) While there are more than 2 vertices.
 - a) Pick a random edge (u, v) in the contracted graph.
 - b) Merge (or contract) u and v into a single vertex (update the contracted graph).
 - c) Remove self-loops
- 3) Return cut represented by two vertices.

Let us understand above algorithm through the example given.

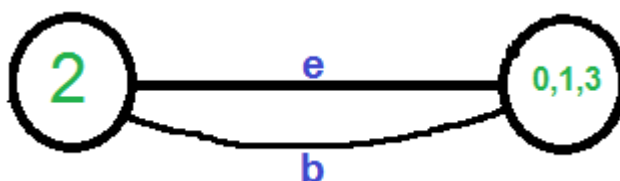
Let the first randomly picked vertex be 'a' which connects vertices 0 and 1. We remove this edge and contract the graph (combine vertices 0 and 1). We get the following graph.



Let the next randomly picked edge be 'd'. We remove this edge and combine vertices (0,1) and 3.

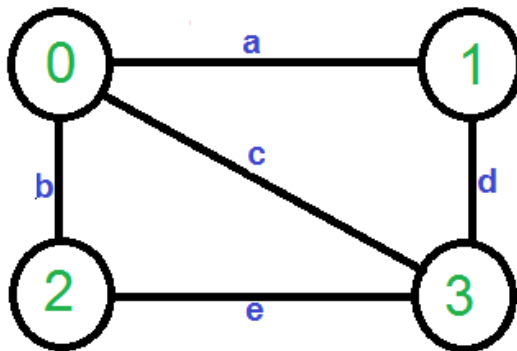


We need to remove self-loops in the graph. So we remove edge 'c'

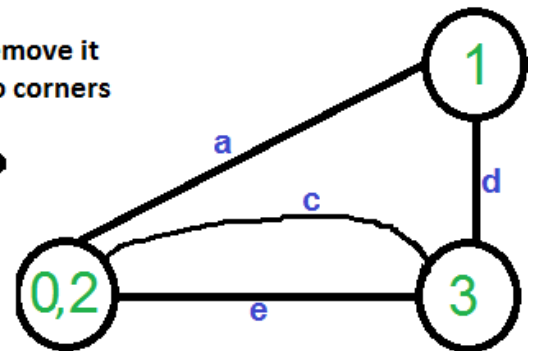


Now graph has two vertices, so we stop. The number of edges in the resultant graph is the cut produced by Karger's algorithm.

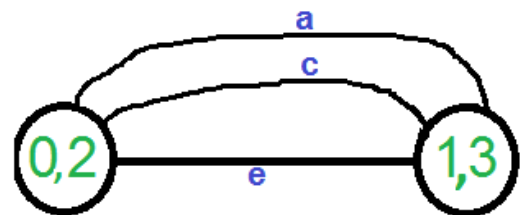
Karger's algorithm is a Monte Carlo algorithm and cut produced by it may not be minimum. For example, the following diagram shows that a different order of picking random edges produces a min-cut of size 3.



Pick edge 'b', remove it and fuse its two corners into one.



Pick edge 'd', remove it and fuse its two corners into one



An example run of Karger's Randomized algorithm where random edges are picked in a way that the output cut is not minimal cut. The output cut produced here is {a, c, e}, but the minimal cut is either {b, e} or {a, d}

Network Flow Problems

The most obvious flow network problem is the following:

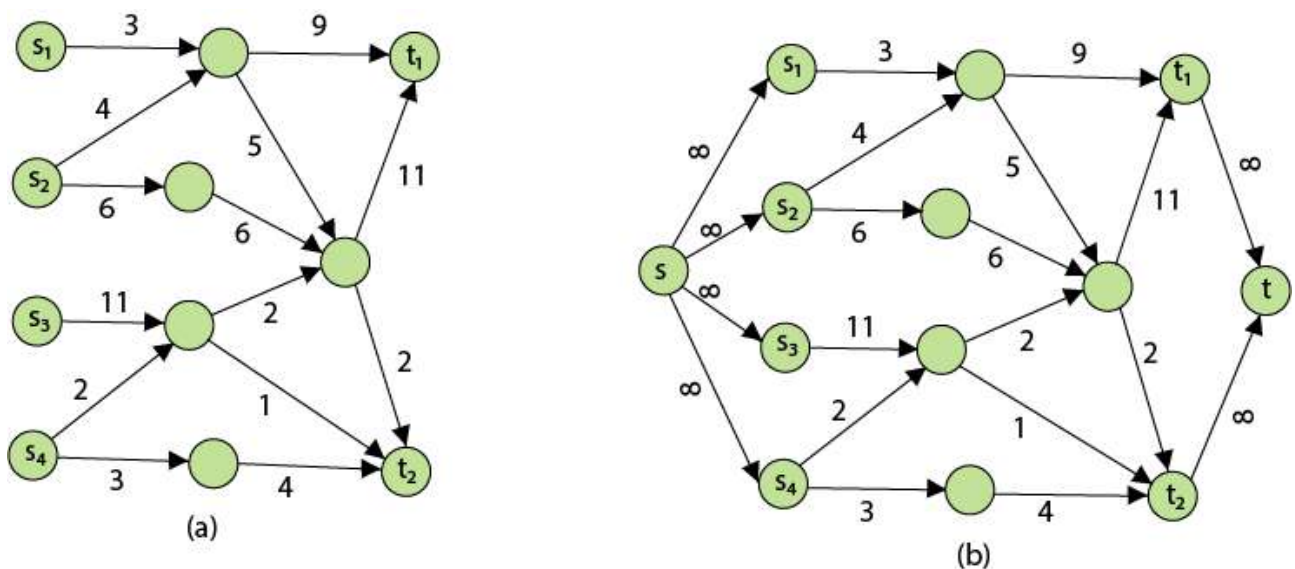
Problem1: Given a flow network $G = (V, E)$, the maximum flow problem is to find a flow with maximum value.

Problem 2: The multiple source and sink maximum flow problem is similar to the maximum flow problem, except there is a set $\{s_1, s_2, s_3, \dots, s_n\}$ of sources and a set $\{t_1, t_2, t_3, \dots, t_n\}$ of sinks.

Fortunately, this problem is no solid than regular maximum flow. Given multiple sources and sink flow network G , we define a new flow network G' by adding

- A super source s ,
- A super sink t ,
- For each s_i , add edge (s, s_i) with capacity ∞ , and
- For each t_i , add edge (t_i, t) with capacity ∞

Figure shows a multiple sources and sinks flow network and an equivalent single source and sink flow network



Residual Networks: The Residual Network consists of an edge that can admit more net flow. Suppose we have a flow network $G = (V, E)$ with source s and sink t . Let f be a flow in G , and examine a pair of vertices $u, v \in V$. The sum of additional net flow we can push from u to v before exceeding the capacity $c(u, v)$ is the residual capacity of (u, v) given by

$$c_f(u, v) = c(u, v) - f(u, v).$$

When the net flow $f(u, v)$ is negative, the residual capacity $c_f(u, v)$ is greater than the capacity $c(u, v)$.

For Example: if $c(u, v) = 16$ and $f(u, v) = 16$ and $f(u, v) = -4$, then the residual capacity $c_f(u, v)$ is 20.

Given a flow network $G = (V, E)$ and a flow f , the residual network of G induced by f is $G_f = (V, E_f)$, where

$$E_f = \{(u, v) \in V \times V : C_f(u, v) \geq 0\}$$

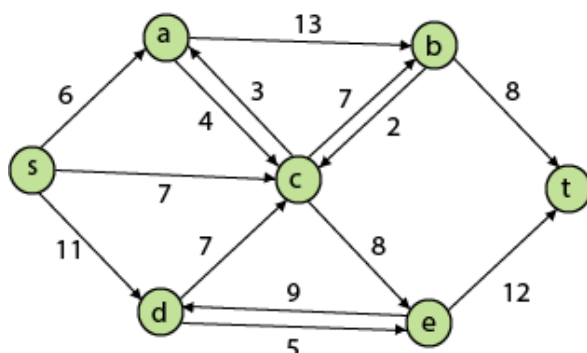
That is, each edge of the residual network, or residual edge, can admit a strictly positive net flow.

Augmenting Path: Given a flow network $G = (V, E)$ and a flow f , an **augmenting path** p is a simple path from s to t in the residual network G_f . By the solution of the residual network, each edge (u, v) on an augmenting path admits some additional positive net flow from u to v without violating the capacity constraint on the edge.

Let $G = (V, E)$ be a flow network with flow f . The **residual capacity** of an augmenting path p is

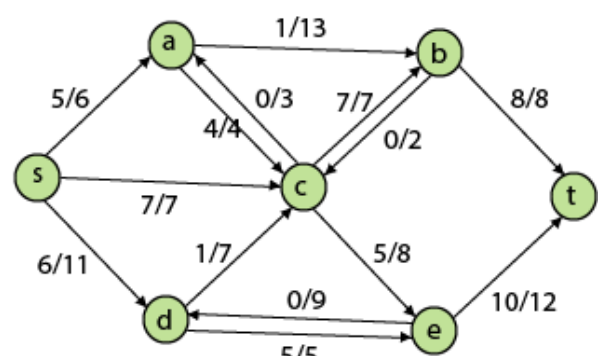
$$C_f(p) = \min \{C_f(u, v) : (u, v) \text{ is on } p\}$$

The residual capacity is the maximal amount of flow that can be pushed through the augmenting path. If there is an augmenting path, then each edge on it has a positive capacity. We will use this fact to compute a maximum flow in a flow network.



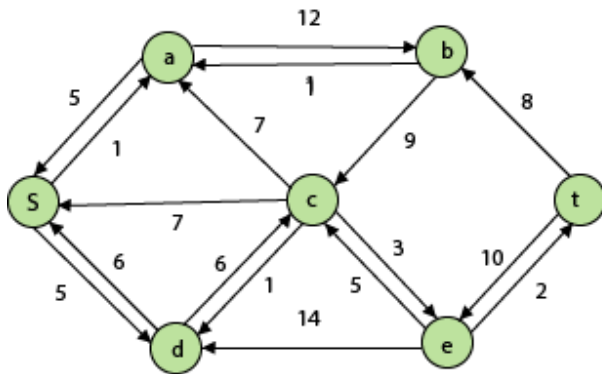
(a)

A flow network $G = (V, E)$

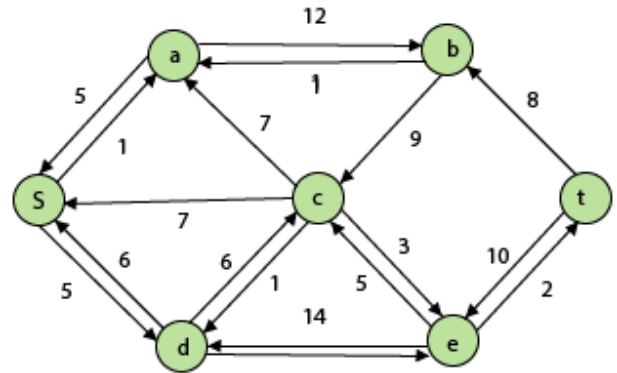


(b)

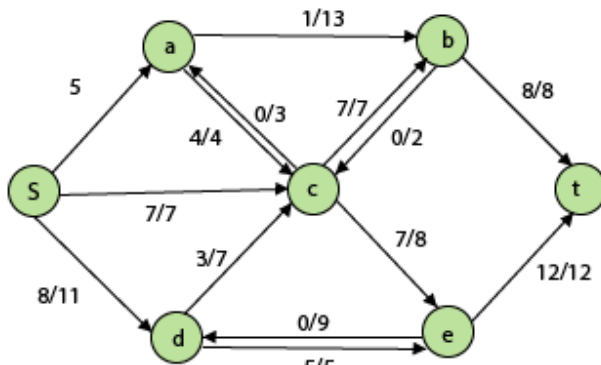
A flow in f in G



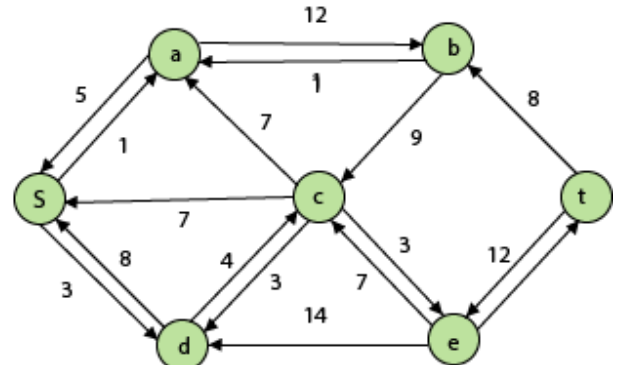
(c)
The residual network G_r



(d)
The grey edges form an augmenting path with capacity z .



(e)
A new flow $f' = f + f_p$



(f)
The residual network G_r



Ford-Fulkerson Algorithm

Initially, the flow of value is 0. Find some augmenting Path p and increase flow f on each edge of p by residual Capacity $c_f(p)$. When no augmenting path exists, flow f is a maximum flow.

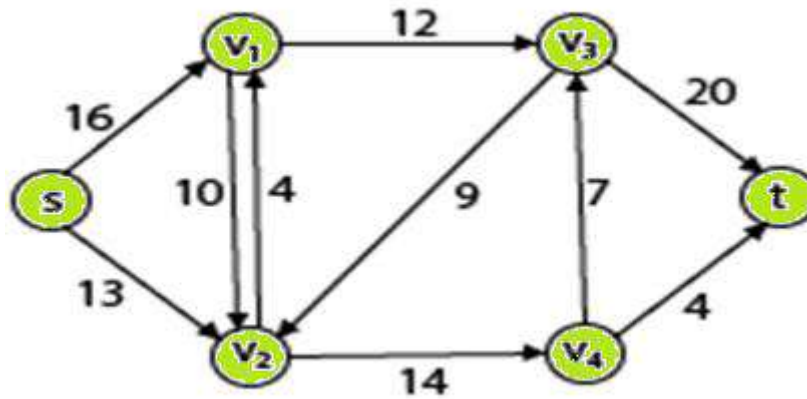
FORD-FULKERSON METHOD (G, s, t)

1. Initialize flow f to 0
2. while there exists an augmenting path p
3. do argument flow f along p
4. Return f

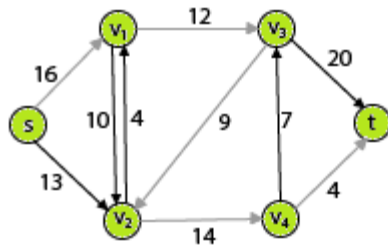
FORD-FULKERSON (G, s, t)

1. for each edge $(u, v) \in E[G]$
2. do $f[u, v] \leftarrow 0$
3. $f[u, v] \leftarrow 0$
4. while there exists a path p from s to t in the residual network G_f .
5. do $c_f(p) \leftarrow \min\{C_f(u, v) : (u, v) \text{ is on } p\}$
6. for each edge (u, v) in p
7. do $f[u, v] \leftarrow f[u, v] + c_f(p)$
8. $f[u, v] \leftarrow -f[u, v]$

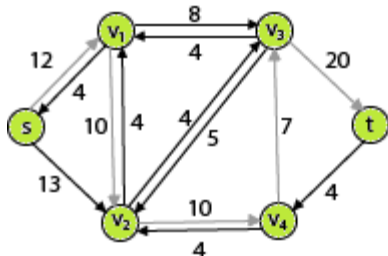
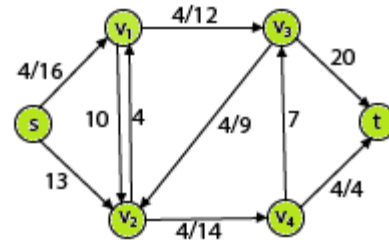
Example: Each Directed Edge is labeled with capacity. Use the Ford-Fulkerson algorithm to find the maximum flow.



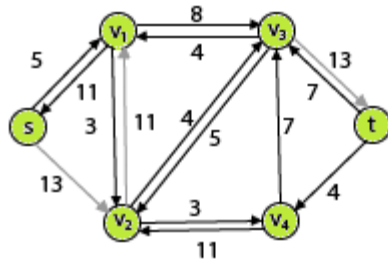
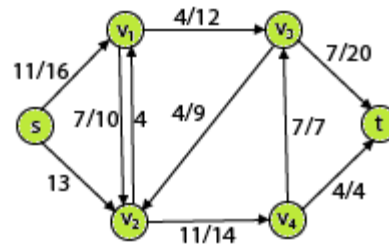
Solution: The left side of each part shows the residual network G_f with a shaded augmenting path p , and the right side of each part shows the net flow f .



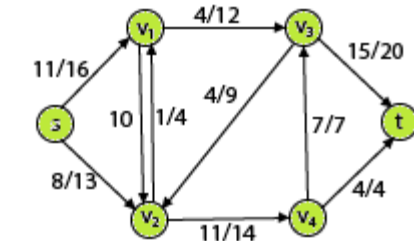
(a)



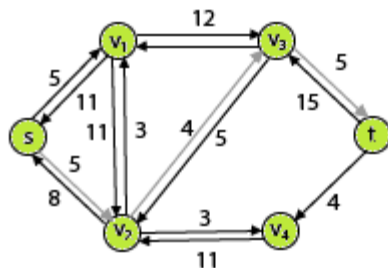
(b)



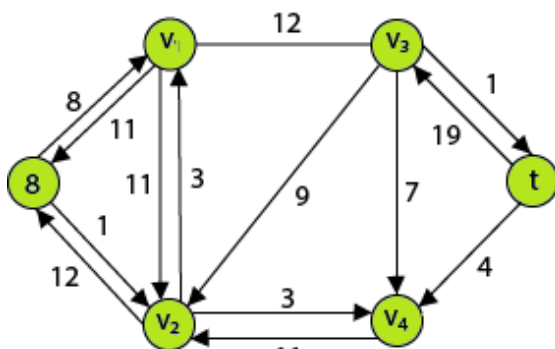
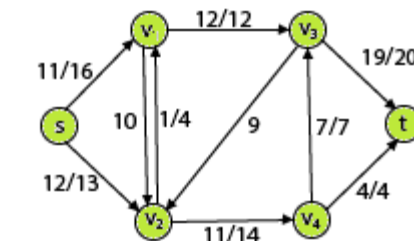
(c)



(In this, 8 is break into 7 and 1 and 7 is cancelled by v_1v_2 flow)



(d)



(e)

Now, it has no augmenting paths. So, the maximum flow shown in (d) is 23 is a maximum flow.



Types of Complexity Classes | P, NP, CoNP, NP hard and NP complete

In computer science, there exist some problems whose solutions are not yet found, the problems are divided into classes known as **Complexity Classes**. In complexity theory, a Complexity Class is a set of problems with related complexity. These classes help scientists to group problems based on how much time and space they require to solve problems and verify the solutions. It is the branch of the theory of computation that deals with the resources required to solve a problem.

The common resources are time and space, meaning how much time the algorithm takes to solve a problem and the corresponding memory usage. The time complexity of an algorithm is used to describe the number of steps required to solve a problem, but it can also be used to describe how long it takes to verify the answer. The space complexity of an algorithm describes how much memory is required for the algorithm to operate.

Complexity classes are useful in organizing similar types of problems.

Types of Complexity Classes

This article discusses the following complexity classes:

1. **P Class**
2. **NP Class**
3. **CoNP Class**
4. **NP hard**
5. **NP complete**

P Class

The P in the P class stands for **Polynomial Time**. It is the collection of decision problems (problems with a “yes” or “no” answer) that can be solved by a deterministic machine in polynomial time.

Features:



1. The solution to P problems is easy to find.
2. P is often a class of computational problems that are solvable and tractable. Tractable means that the problems can be solved in theory as well as in practice. But the problems that can be solved in theory but not in practice are known as intractable.

This class contains many natural problems like:

1. **Calculating the greatest common divisor.**
2. **Finding a maximum matching.**
3. **Decision versions of linear programming.**

NP Class

The NP in NP class stands for **Non-deterministic Polynomial Time**. It is the collection of decision problems that can be solved by a non-deterministic machine in polynomial time.

Features:

1. The solutions of the NP class are hard to find since they are being solved by a non-deterministic machine but the solutions are easy to verify.
2. Problems of NP can be verified by a Turing machine in polynomial time.

Example:

Let us consider an example to better understand the NP class. Suppose there is a company having a total of 1000 employees having unique employee IDs. Assume that there are 200 rooms available for them. A selection of 200 employees must be paired together, but the CEO of the company has the data of some employees who can't work in the same room due to some personal reasons. This is an example of an NP problem. Since it is easy to check if the given choice of 200 employees proposed by a coworker is satisfactory or not i.e. no pair taken from the coworker list appears on the list given by the CEO. But generating such a list from scratch seems to be so hard as to be completely impractical.

It indicates that if someone can provide us with the solution to the problem, we can find the correct



and incorrect pair in polynomial time. Thus for the NP class problem, the answer is possible, which can be calculated in polynomial time.

This class contains many problems that one would like to be able to solve effectively:

1. **Boolean Satisfiability Problem (SAT).**
2. **Hamiltonian Path Problem.**
3. **Graph coloring.**

Co-NP Class

Co-NP stands for the complement of NP Class. It means if the answer to a problem in Co-NP is No, then there is proof that can be checked in polynomial time.

Features:

1. If a problem X is in NP, then its complement X' is also in CoNP.
2. For an NP and CoNP problem, there is no need to verify all the answers at once in polynomial time, there is a need to verify only one particular answer “yes” or “no” in polynomial time for a problem to be in NP or CoNP.

Some example problems for CO-NP are:

1. **To check prime number.**
2. **Integer Factorization.**

NP-hard class

An NP-hard problem is at least as hard as the hardest problem in NP and it is the class of the problems such that every problem in NP reduces to NP-hard.

Features:

1. All NP-hard problems are not in NP.



2. It takes a long time to check them. This means if a solution for an NP-hard problem is given then it takes a long time to check whether it is right or not.
3. A problem A is in NP-hard if, for every problem L in NP, there exists a polynomial-time reduction from L to A.

Some of the examples of problems in NP-hard are:

1. **Halting problem.**
2. **Qualified Boolean formulas.**
3. **No Hamiltonian cycle.**

NP-complete class

A problem is NP-complete if it is both NP and NP-hard. NP-complete problems are the hard problems in NP.

Features:

1. NP-complete problems are special as any problem in NP class can be transformed or reduced into NP-complete problems in polynomial time.
2. If one could solve an NP-complete problem in polynomial time, then one could also solve any NP problem in polynomial time.

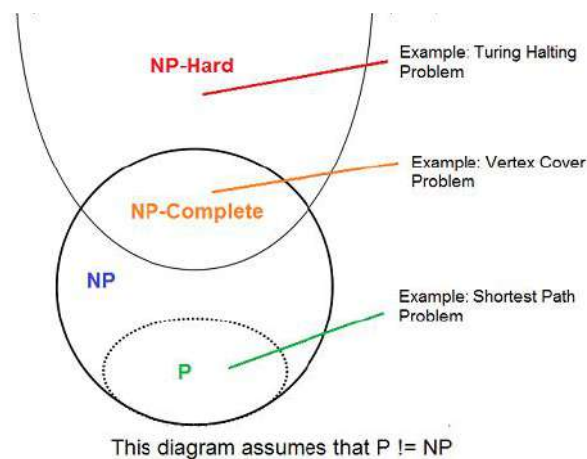
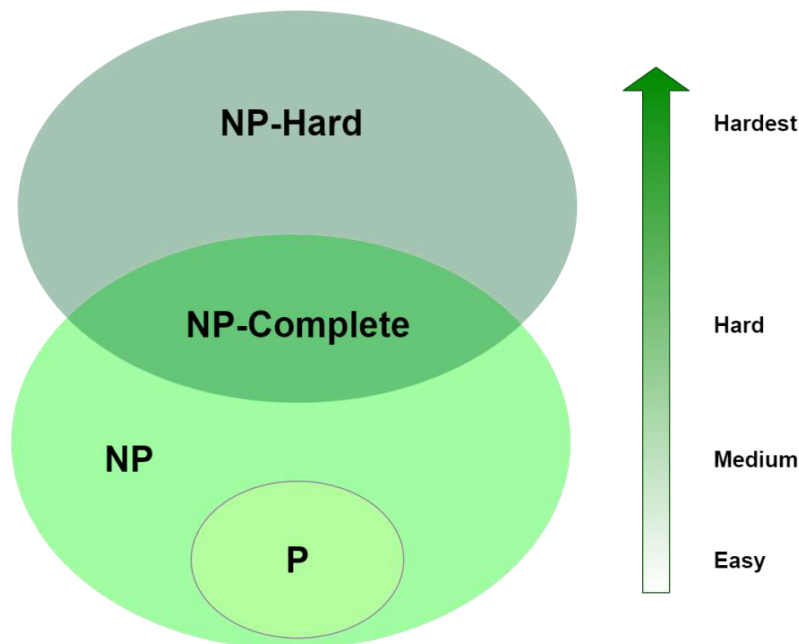
Some example problems include:

1. **Decision version of 0/1 Knapsack.**
2. **Hamiltonian Cycle.**
3. **Satisfiability.**
4. **Vertex cover.**

Complexity Class Characteristic feature

P	Easily solvable in polynomial time.
NP	Yes, answers can be checked in polynomial time.

Co-NP	No, answers can be checked in polynomial time.
NP-hard	All NP-hard problems are not in NP and it takes a long time to check them.
NP-complete	A problem that is NP and NP-hard is NP-complete.



Difference between NP hard and NP complete problem

NP Problem:

The NP problems set of problems whose solutions are hard to find but easy to verify and are solved by Non-Deterministic Machine in polynomial time.



NP-Hard Problem:

A Problem X is NP-Hard if there is an NP-Complete problem Y, such that Y is reducible to X in polynomial time. NP-Hard problems are as hard as NP-Complete problems. NP-Hard Problem need not be in NP class.

NP-Complete Problem:

A problem X is NP-Complete if there is an NP problem Y, such that Y is reducible to X in polynomial time. NP-Complete problems are as hard as NP problems. A problem is NP-Complete if it is a part of both NP and NP-Hard Problem. A non-deterministic Turing machine can solve NP-Complete problem in polynomial time.

Difference between NP-Hard and NP-Complete:

NP-hard	NP-Complete
NP-Hard problems(say X) can be solved if and only if there is a NP-Complete problem(say Y) that can be reducible into X in polynomial time.	NP-Complete problems can be solved by a non-deterministic Algorithm/Turing Machine in polynomial time.
To solve this problem, it do not have to be in NP .	To solve this problem, it must be both NP and NP-hard problems.
Time is unknown in NP-Hard.	Time is known as it is fixed in NP-Hard.
NP-hard is not a decision problem.	NP-Complete is exclusively a decision problem.
Not all NP-hard problems are NP-complete.	All NP-complete problems are NP-hard
Do not have to be a Decision problem.	It is exclusively a Decision problem.
Example: Halting problem, Vertex cover problem, etc.	Example: Determine whether a graph has a Hamiltonian cycle, Determine whether a Boolean formula is satisfiable or not, Circuit-satisfiability problem, etc.



Cook–Levin theorem or Cook’s theorem

In computational complexity theory, the Cook–Levin theorem, also known as Cook’s theorem, states that the Boolean satisfiability problem is NP-complete. That is, it is in NP, and any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the Boolean satisfiability problem.

*Stephen Arthur Cook and L.A. Levin in 1973 independently proved that the **satisfiability problem(SAT)** is NP-complete. Stephen Cook, in 1971, published an important paper titled ‘The complexity of Theorem Proving Procedures’, in which he outlined the way of obtaining the proof of an NP-complete problem by reducing it to SAT. He proved **Circuit-SAT** and **3CNF-SAT** problems are as hard as SAT. Similarly, Leonid Levin independently worked on this problem in the then Soviet Union. The proof that SAT is NP-complete was obtained due to the efforts of these two scientists. Later, Karp reduced 21 optimization problems, such as Hamiltonian tour, vertex cover, and clique, to the SAT and proved that those problems are NP-complete.*

Hence, an SAT is a significant problem and can be stated as follows:

Given a boolean expression **F** having **n** variables **x₁, x₂, ..., x_n**, and Boolean operators, is it possible to have an assignment for variables true or false such that binary expression **F** is true?

This problem is also known as the **formula – SAT**. An **SAT(formula-SAT or simply SAT)** takes a Boolean expression **F** and checks whether the given expression(or formula) is satisfiable. A Boolean expression is said to be satisfactory for some valid assignments of variables if the evaluation comes to be true. Prior to discussing the details of SAT, let us now discuss some important terminologies of a Boolean expression.

- **Boolean variable:** A variable, say **x**, that can have only two values, true or false, is called a boolean variable
- **Literal:** A literal can be a logical variable, say **x**, or the negation of it, that is **x** or **\bar{x}** ; **x** is called a positive literal, and **\bar{x}** is called the negative literal
- **Clause:** A sequence of variables(**x₁, x₂, ..., x_n**) that can be separated by a logical **OR** operator is called a clause. For example, (**x₁ V x₂ V x₃**) is a clause of three literals.



- **Expressions:** One can combine all the preceding clauses using a Boolean operator to form an expression.
- **CNF form:** An expression is in CNF form (conjunctive normal form) if the set of clauses are separated by an **AND** (\wedge), operator, while the literals are connected by an **OR** (\vee) operator. The following is an example of an expression in the CNF form:
 - $f = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee \bar{x}_3 \vee x_2)$
- **3 – CNF:** An expression is said to be in 3-CNF if it is in the conjunctive normal form, and every clause has exact three literals.

Thus, an **SAT** is one of the toughest problems, as there is no known algorithm other than the brute force approach. A brute force algorithm would be an exponential-time algorithm, as 2^n possible assignments need to be tried to check whether the given Boolean expression is true or not. Stephen Cook and Leonid Levin proved that the **SAT** is NP-complete.

Cook demonstrated the reduction of other hard problems to SATs. Karp provided proof of 21 important problems, Such as Hamiltonian tour, vertex cover, and clique, by reducing it to SAT using Karp reduction.

Let us briefly introduce the three types of SATs. They are as follows:

1. **Circuit- SAT:** A circuit-SAT can be stated as follows: given a Boolean circuit, which is a collection of gates such as **AND**, **OR**, and **NOT**, and n inputs, is there any input assignments of Boolean variables so that the output of the given circuit is true? Again, the difficulty with these problems is that for n inputs to the circuit. 2^n possible outputs should be checked. Therefore, this brute force algorithm is an exponential algorithm and hence this is a hard problem.
2. **CNF-SAT:** This problem is a restricted problem of **SAT**, where the expression should be in a conjunctive normal form. An expression is said to be in a conjunction form if all the clauses are connected by the Boolean **AND** operator. Like in case of a **SAT**, this is about assigning truth values to n variables such that the output of the expression is true.
3. **3-CNF-SAT(3-SAT):** This problem is another variant where the additional restriction is that the expression is in a conjunctive normal form and that every clause should contain



exactly three literals. This problem is also about assigning n assignments of truth values to n variables of the Boolean expression such that the output of the expression is true. In simple words, given an expression in 3-CNF, a 3-SAT problem is to check whether the given expression is satisfiable.

These problems can be used to prove the NP-completeness of some important problems.



Proof that SAT is NP Complete

SAT Problem: SAT(Boolean Satisfiability Problem) is the problem of determining if there exists an interpretation that satisfies a given boolean formula. It asks whether the variables of a given boolean formula can be consistently replaced by the values **TRUE** or **FALSE** in such a way that the formula evaluates to **TRUE**. If this is the case, the formula is called *satisfiable*. On the other hand, if no such assignment exists, the function expressed by the formula is **FALSE** for all possible variable assignments and the formula is *unsatisfiable*.

Problem: Given a boolean formula f , the problem is to identify if the formula f has a satisfying assignment or not.

Explanation: An instance of the problem is an input specified to the problem. An instance of the problem is a boolean formula f . Since an [NP-complete](#) problem is a problem which is both **NP** and **NP-Hard**, the proof or statement that a problem is NP-Complete consists of two parts:

1. The problem itself is in NP class.
2. All other problems in NP class can be polynomial-time reducible to that.
(B is polynomial-time reducible to C is denoted as $\leq P^C$)

If the 2nd condition is only satisfied then the problem is called **NP-Hard**.

But it is not possible to reduce every NP problem into another NP problem to show its NP-Completeness all the time i.e., to show a problem is NP-complete then prove that the problem is in NP and any NP-Complete problem is reducible to that i.e. if B is NP-Complete and $B \leq P^C$ For C in NP, then C is NP-Complete. Thus, it can be verified that the **SAT Problem** is NP-Complete using the following propositions:

SAT is in NP:

If any problem is in NP, then given a 'certificate', which is a solution to the problem and an instance of the problem (a boolean formula f) we will be able to check (identify if the solution is correct or not) certificate in polynomial time. This can be done by checking if the given assignment of variables satisfies the boolean formula.

SAT is NP-Hard:

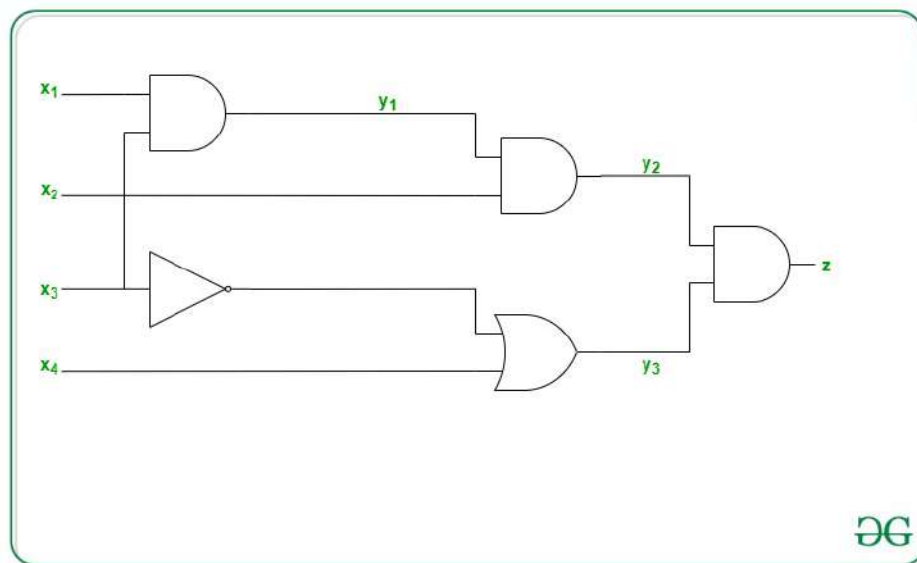
In order to prove that this problem is NP-Hard then reduce a known problem, Circuit-SAT in this case to our problem. The boolean circuit C can be corrected into a boolean formula as:

- For every input wire, add a new variable y_i .
- For every output wire, add a new variable Z .
- An equation is prepared for each gate.
- These sets of equations are separated by \cap values and adding $\cap Z$ at the end.

This transformation can be done in linear time. The following propositions now hold:

- If there is a set of input, variable values satisfying the circuit then it can derive an assignment for the formula f that satisfies the formula. This can be simulated by computing the output of every gate in the circuit.
- If there is a satisfying assignment for the formula f , this can satisfy the boolean circuit after the removal of the newly added variables.

For Example: If below is the circuit then:



Therefore, the **SAT Problem** is NP-Complete.



Proof that vertex cover is NP complete

Problem – Given a graph $G(V, E)$ and a positive integer k , the problem is to find whether there is a subset V' of vertices of size at most k , such that every edge in the graph is connected to some vertex in V' .

Explanation –

First let us understand the notion of an instance of a problem. An instance of a problem is nothing but an input to the given problem. An instance of the Vertex Cover problem is a graph $G(V, E)$ and a positive integer k , and the problem is to check whether a vertex cover of size at most k exists in G . Since an NP Complete problem, by definition, is a problem which is both in NP and NP hard, the proof for the statement that a problem is NP Complete consists of two parts:

1. Proof that vertex cover is in NP –

If any problem is in NP, then, given a 'certificate' (a solution) to the problem and an instance of the problem (a graph G and a positive integer k , in this case), we will be able to verify (check whether the solution given is correct or not) the certificate in polynomial time.

The certificate for the vertex cover problem is a subset V' of V , which contains the vertices in the vertex cover. We can check whether the set V' is a vertex cover of size k using the following strategy (for a graph $G(V, E)$):

```
let count be an integer
set count to 0
for each vertex v in V'
    remove all edges adjacent to v from set E
    increment count by 1
if count = k and E is empty
    then
        the given solution is correct
    else
```




the given solution is wrong

It is plain to see that this can be done in polynomial time. Thus the vertex cover problem is in the class NP.

2. Proof that vertex cover is NP Hard –

To prove that Vertex Cover is NP Hard, we take some problem which has already been proven to be NP Hard, and show that this problem can be reduced to the Vertex Cover problem. For this, we consider the Clique problem, which is NP Complete (and hence NP Hard).

“In computer science, the clique problem is the computational problem of finding cliques (subsets of vertices, all adjacent to each other, also called complete subgraphs) in a graph.”

Here, we consider the problem of finding out whether there is a clique of size k in the given graph. Therefore, an instance of the clique problem is a graph $G(V, E)$ and a non-negative integer k , and we need to check for the existence of a clique of size k in G .

Now, we need to show that any instance (G, k) of the Clique problem can be reduced to an instance of the vertex cover problem. Consider the graph G' which consists of all edges not in G , but in the complete graph using all vertices in G . Let us call this the complement of G . Now, the problem of finding whether a clique of size k exists in the graph G is the same as the problem of finding whether there is a vertex cover of size $|V| - k$ in G' . We need to show that this is indeed the case.

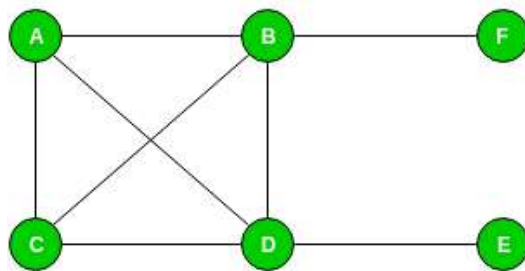
Assume that there is a clique of size k in G . Let the set of vertices in the clique be V' . This means $|V'| = k$. In the complement graph G' , let us pick any edge (u, v) . Then at least one of u or v must be in the set $V - V'$. This is because, if both u and v were from the set V' , then the edge (u, v) would belong to V' , which, in turn would mean that the edge (u, v) is in G . This is not possible since (u, v) is not in G . Thus, all edges in G' are covered by vertices in the set $V - V'$.

Now assume that there is a vertex cover V'' of size $|V| - k$ in G' . This means that all edges in G' are connected to some vertex in V'' . As a result, if we pick any edge (u, v) from G' , both of

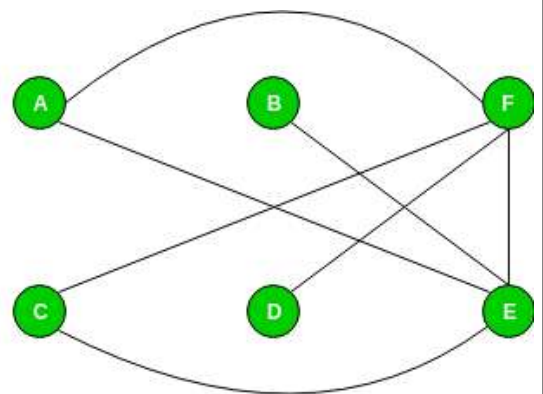
them cannot be outside the set V'' . This means, all edges (u, v) such that both u and v are outside the set V'' are in G , i.e., these edges constitute a clique of size k .

Thus, we can say that there is a clique of size k in graph G if and only if there is a vertex cover of size $|V| - k$ in G' , and hence, any instance of the clique problem can be reduced to an instance of the vertex cover problem. Thus, vertex cover is NP Hard. Since vertex cover is in both NP and NP Hard classes, it is NP Complete.

To understand the proof, consider the following example graph and its complement:



G
 $V' = \{A, B, C, D\}$



G'
 $V'' = \{E, F\}$