NEHRU COLLEGE OF ENGINEERING AND RESEARCH CENTRE
(NAAC Accredited)
(Approved by AICTE, Affiliated to APJ Abdul Kalam Technological University, Kerala)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COURSE MATERIALS

CS 302 DESIGN AND ANALYSIS OF ALGORITHM

VISION OF THE INSTITUTION

To mould true citizens who are millennium leaders and catalysts of change through excellence in education.

MISSION OF THE INSTITUTION

NCERC is committed to transform itself into a center of excellence in Learning and Research in Engineering and Frontier Technology and to impart quality education to mould technically competent citizens with moral integrity, social commitment and ethical values.

We intend to facilitate our students to assimilate the latest technological know-how and to imbibe discipline, culture and spiritually, and to mould them in to technological giants, dedicated research scientists and intellectual leaders of the country who can spread the beams of light and happiness among the poor and the underprivileged.

ABOUT DEPARTMENT

- Established in: 2002
- Course offered : B.Tech in Computer Science and Engineering

  M.Tech in Computer Science and Engineering

  M.Tech in Cyber Security
- Approved by AICTE New Delhi and Accredited by NAAC
- Affiliated to the University of A P J Abdul Kalam Technological University.

DEPARTMENT VISION

Producing Highly Competent, Innovative and Ethical Computer Science and Engineering Professionals to facilitate continuous technological advancement.

DEPARTMENT MISSION

1. To Impart Quality Education by creative Teaching Learning Process
2. To Promote cutting-edge Research and Development Process to solve real world problems with emerging technologies.
3. To Inculcate Entrepreneurship Skills among Students.
4. To cultivate Moral and Ethical Values in their Profession.

PROGRAMME EDUCATIONAL OBJECTIVES

PEO 1: Graduates will be able to Work and Contribute in the domains of Computer Science and Engineering through lifelong learning.

PEO 2: Graduates will be able to Analyze, design and development of novel Software Packages, Web Services, System Tools and Components as per needs and specifications.

PEO 3: Graduates will be able to demonstrate their ability to adapt to a rapidly changing environment by learning and applying new technologies.

PEO 4: Graduates will be able to adopt ethical attitudes, exhibit effective communication skills, Team work and leadership qualities.

PROGRAM OUTCOMES (POS)

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSO)

PSO1: Ability to Formulate and Simulate Innovative Ideas to provide software solutions for Real-time Problems and to investigate for its future scope.

PSO2: Ability to learn and apply various methodologies for facilitating development of high quality System Software Tools and Efficient Web Design Models with a focus on performance optimization.

PSO3: Ability to inculcate the Knowledge for developing Codes and integrating hardware/software products in the domains of Big Data Analytics, Web Applications and Mobile Apps to create innovative career path and for the socially relevant issues.

COURSE OUTCOMES

C302.1	To Analyze a given algorithm and express its time and space complexities and also analyze different recurrence methods.
C302.2	Tousethe Master's Theorem to find the complexity and to design different types of trees.
C302.3	To Apply traversals, shortest path finding algorithms into graphs.
C302.4	To Analyze different algorithm methods like dynamic programming and divide and conquer strategies.
C302.5	To Implement Optimization problems using Greedy strategy.
C302.6	To Design efficient algorithms using Back Tracking and Branch Bound Techniques for solving problemsand to apply computational problems into P, NP, NP-Hard and NP-Complete.

MAPPING OF COURSE OUTCOMES WITH PROGRAM OUTCOMES

CO'S	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
C302.1	3	3	3	2	2	-	-	-	-	-	-	-
C302.2	3	3	3	2	-	-	-	-	-	-	-	-
C302.3	3	3	3	2	2	-	-	-	-	-	-	-
C302.4	3	3	3	2	2	-	-	-	-	-	-	-
C302.5	3	3	3	2	2	-	-	-	-	-	-	-
C302.6	3	3	3	2	-	-	-	-	-	-	-	-
C302	3	3	3	2	2							

CO'S	PSO1	PSO2	PSO3
C302.1	3	3	-
C302.2	3	3	-
C302.3	3	3	-
C302.4	3	3	-
C302.5	3	3	-
C302.6	3	3	-
C302	3	3	

Note: H-Highly correlated=3, M-Medium correlated=2, L-Less correlated=1

SYLLABUS

Course code	Course Name	L-T-P - Credits	Year of Introduction
CS302	Design and Analysis of Algorithms	3-1-0-4	2016
Prerequisite: Nil			

Course Objectives

- To introduce the concepts of Algorithm Analysis, Time Complexity, Space Complexity.
- To discuss various Algorithm Design Strategies with proper illustrative examples.
- To introduce Complexity Theory.

Syllabus

Introduction to Algorithm Analysis, Notions of Time and Space Complexity, Asymptotic Notations, Recurrence Equations and their solutions, Master's Theorem, Divide and Conquer and illustrative examples, AVL trees, Red-Black Trees, Union-find algorithms, Graph algorithms, Divide and Conquer, Dynamic Programming, Greedy Strategy, Back Tracking and Branch and Bound, Complexity classes

Expected outcome

The students will be able to

i. Analyze a given algorithm and express its time and space complexities in asymptotic notations.
ii. Solve recurrence equations using Iteration Method, Recurrence Tree Method and Master's Theorem.
iii. Design algorithms using Divide and Conquer Strategy.
iv. Compare Dynamic Programming and Divide and Conquer Strategies.
v. Solve Optimization problems using Greedy strategy.
vi. Design efficient algorithms using Back Tracking and Branch Bound Techniques for solving problems.
vii. Classify computational problems into P, NP, NP-Hard and NP-Complete.

Course code	Course Name	L-T-P Credits	Year of Introduction
CS304	**COMPILER DESIGN**	**3-0-0-3**	**2016**

Prerequisite: Nil

Course Objectives

- To provide a thorough understanding of the internals of Compiler Design.

Syllabus

Phases of compilation, Lexical analysis, Token Recognition, Syntax analysis, Bottom Up and Top Down Parsers, Syntax directed translation schemes, Intermediate Code Generation, Triples and Quadruples, Code Optimization, Code Generation.

Expected Outcome

The students will be able to

i. Explain the concepts and different phases of compilation with compile time error handling.
ii. Represent language tokens using regular expressions, context free grammar and finite automata and design lexical analyzer for a language.
iii. Compare top down with bottom up parsers, and develop appropriate parser to produce parse tree representation of the input.
iv. Generate intermediate code for statements in high level language.
v. Design syntax directed translation schemes for a given context free grammar.
vi. Apply optimization techniques to intermediate code and generate machine code for high level language program.

Text Books

1. Aho A. Ravi Sethi and D Ullman. Compilers – Principles Techniques and Tools, Addison Wesley, 2006.
2. D. M.Dhamdhare, System Programming and Operating Systems,Tata McGraw Hill & Company, 1996.

References

1. Kenneth C. Louden, Compiler Construction – Principles and Practice, Cengage Learning Indian Edition, 2006.
2. Tremblay and Sorenson, The Theory and Practice of Compiler Writing, Tata McGraw Hill & Company,1984.

Text Books

1. Ellis Horowitz, SartajSahni, SanguthevarRajasekaran, Computer Algorithms, Universities Press, 2007 [Modules 3,4,5]
2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, MIT Press, 2009 [Modules 1,2,6]

References

1. Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, The Design and Analysis of Computer Algorithms, Pearson Education, 1999.
2. Anany Levitin, Introduction to the Design and Analysis of Algorithms, Pearson, 3rd Edition, 2011.
3. Gilles Brassard, Paul Bratley, Fundamentals of Algorithmics, Pearson Education, 1995.
4. Richard E. Neapolitan, Kumarss Naimipour, Foundations of Algorithms using C++ Psuedocode, Second Edition, 1997.

Course Plan

Module	Contents	Hours	End Sem. Exam Marks

I	***Introduction to Algorithm Analysis***Time and Space Complexity- Elementary operations and Computation of Time Complexity- Best, worst and Average Case Complexities- Complexity Calculation of simple algorithms ***Recurrence Equations:***Solution of Recurrence Equations – Iteration Method and Recursion Tree Methods	**04** **04**	**15 %**
II	***Master's Theorem***(Proof not required) – examples, Asymptotic Notations and their properties- Application of Asymptotic Notations in Algorithm Analysis- Common Complexity Functions ***AVL Trees*** – rotations, Red-Black Trees insertion and deletion (Techniques only; algorithms not expected). B-Trees – insertion and deletion operations. Sets- Union and find operations on disjoint sets.	**05** **05**	**15%**
FIRST INTERNAL EXAM			
III	***Graphs*** – DFS and BFS traversals, complexity, Spanning trees – Minimum Cost Spanning Trees, single source shortest path algorithms, Topological sorting, strongly connected components.	**07**	**15%**
IV	***Divide and Conquer:***The Control Abstraction, 2 way Merge sort, Strassen's Matrix Multiplication, Analysis ***Dynamic Programming :*** The control Abstraction- The Optimality Principle- Optimal matrix multiplication, Bellman-Ford Algorithm	**04** **05**	**15%**
SECOND INTERNAL EXAM			
V	Analysis, Comparison of Divide and Conquer and Dynamic Programming strategies ***Greedy Strategy: -*** The Control Abstraction- the Fractional Knapsack Problem, Minimal Cost Spanning Tree Computation- Prim's Algorithm – Kruskal's Algorithm.	**02** **04** **03**	**20%**
VI	***Back Tracking: -***The Control Abstraction – The N Queen's Problem, 0/1 Knapsack Problem ***Branch and Bound:***Travelling Salesman Problem. ***Introduction to Complexity Theory :-***Tractable and Intractable Problems- The P and NP Classes- Polynomial Time Reductions - The NP- Hard and NP-Complete Classes	**03** **03** **03**	**20%**
END SEMESTER EXAM			

Question Paper Pattern

1. There will be *five* parts in the question paper – A, B, C, D, E
2. Part A
    a. Total marks : 12
    b. *Four* questions each having *3* marks, uniformly covering modules I and II; All*four* questions have to be answered.
3. Part B
    a. Total marks : 18
    b. *Three* questions each having *9* marks, uniformly covering modules I and II; *Two* questions have to be answered. Each question can have a maximum of three subparts.
4. Part C

   a. Total marks : 12
   b. *Four* questions each having *3* marks, uniformly covering modules III and IV; All*four* questions have to be answered.
5. Part D
   a. Total marks : 18
   b. *Three* questions each having *9* marks, uniformly covering modules III and IV; *Two* questions have to be answered. Each question can have a maximum of three subparts
6. Part E
   a. Total Marks: 40
   b. *Six* questions each carrying 10 marks, uniformly covering modules V and VI; *four* questions have to be answered.
   c. A question can have a maximum of three sub-parts.
7. There should be at least 60% analytical/numerical questions.

QUESTION BANK

MODULE I				
Q:NO:	**QUESTIONS**	**CO**	**KL**	**PAGE NO:**
1	What are the different types of algorithm design techniques? Explain in detail.	CO1	K2	13
2	Discuss in detail about Space Complexity with example.	CO1	K2	17
3	Discuss in detail about time complexity with example.	CO1	K2	21
4	Describe Best, Worst and Average case complexities in detail.	CO1	K3	25
5	Find the best case of Linear Search Algorithm.	CO1	K2	26
6	Discuss any 2 method to solve recurrence equation in detail.	CO1	K2	44
7	Solve the recurrence equation T(n)= 3T(n/4)+n using iteration method.	CO1	K3	44
8	Solve the recurrence equation T(n)= 2T(n/2)+4n using recursion tree method.	CO1	K2	58
MODULE II				
1	Define master's theorem with example.	CO2	K2	62
2	Solve the recurrence equation T(n)= 2T(n/2)+4n using masters method.	CO2	K4	63
3	Describe asymptotic notation in detail.	CO2	K2	66
4	Find the O notation of the given equation $5n^3 + n^2 + 6n+2 = f(n)$.	CO2	K5	68
5	Compare Little Oh and Little Omega Notations with examples.	CO2	K5	72
6	Elucidate AVL tree rotations in detail.	CO2	K3	82
7	Insert 1,2,3,4,5,6,7,8 into an AVL tree.	CO2	K5	86
8	Construct an AVL tree having the following elements H,I,J,B,A,E,C.	CO2	K2	89
9	Discuss the properties of red black tree.	CO2	K2	96
10	Narrate the insertion of red black tree with example.	CO2	K3	101
11	Explain any 4 cases of red black tree deletion with example.	CO2	K2	110
12	Explain in detail about B Tree with example.	CO2	K2	119

MODULE III				
1	Discuss in detail about DFS with example.	CO3	K3	125
2	Discuss in detail about BFS with example.	CO3	K3	134
3	Example Minimum Cost Spanning Tree with example.	CO3	K2	141
4	Discuss in detail about Kruskal's algorithm with example.	CO3	K3	141
5	Discuss in detail about Prim's algorithm with example.	CO3	K5	144
6	Discuss in detail about Dijkstra's algorithm.	CO3	K3	146
7	Explain in detail about Bellman Ford algorithm.	CO3	K5	147
8	Explain Topological Sorting	CO3	K2	149
MODULE IV				
1	Briefly explain the control abstraction of divide and conquer.	CO4	K2	154
2	Explain the concept of 2- way merge sort.	CO4	K1	155
3	Briefly explain about strassen's algorithm for matrix multiplication.	CO4	K2	158
4	Briefly explain the control abstraction of divide and conquer.	CO4	K3	161
5	Explain the working of Bellman ford algorithm.	CO4	K5	170
MODULE V				
1	Compare and contrast divide and conquer with dynamic programming.	CO5	K4	173
2	Explain about Greedy strategy.	CO5	K2	174
3	Write a short note on Fractional Knapsack problem.	CO5	K3	175
4	State MST with examples.	CO5	K2	177
5	Write about Kruskal's algorithm.	CO5	K3	178
6	Briefly explain about Prim's algorithm.	CO5	K2	179

APPENDIX 1

CONTENT BEYOND THE SYLLABUS

MODULE 1

Introduction to Algorithm Analysis. Time and Space complexity - Elementary Operations and computation of Time Complexity - Best, worst and Average case complexities - Complexity calculations of simple algorithms.
Recurrence Equations. Solution of Recurrence Equations - Iteration Method and Recursion Tree Methods.

① INTRODUCTION TO ALGORITHMS

Algorithm is a persian name derived from Abu Jafar Mohammad ibn Musba al khowarizmi.

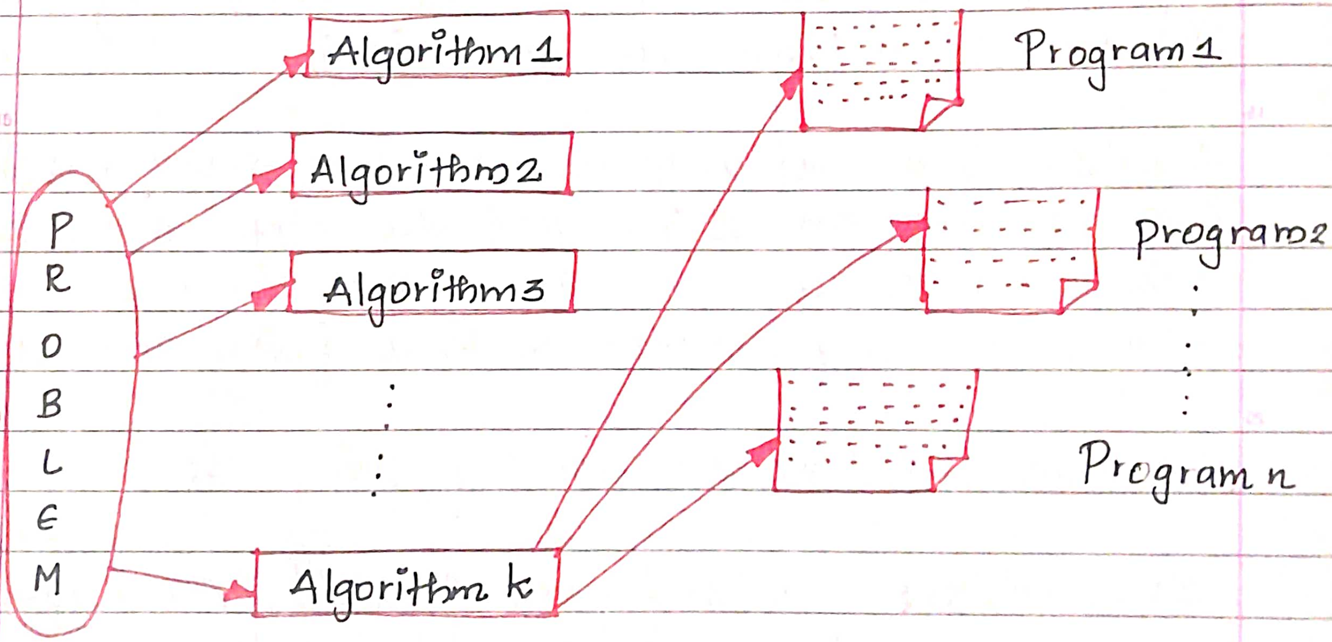
Algorithm is defined as formula or set of steps for solving a particular problem. It is a good s/w engg Practice to design algorithms before we write a program. It has a step by step method for solving the problem in a finite amount of time.

Algorithm must have the following properties:

① Finiteness: Algorithm must complete after a finite no. of instructions have been executed.
② Absence of Ambiguity: Each step must be clearly defined, having only one interpretation.
③ Definition of Sequence: Each step must have a unique defined preceding and succeeding step. The first step (start step) and last step (halt step) must be ~~cleared~~ clearly noted.
④ Input/Output: No. of types of required i/p & results must be specified.
⑤ Feasibility: It must be possible to perform each instruction.

Characteristics of an Algorithm :

1. Input
2. Output
3. Definiteness: Each operation must be definite meaning.
4. Effectiveness: Each operation be effective ie., each step must be such that it can be done by a person using pencil and paper in a finite amount of time.
5. Termination: Algorithm terminates after a finite no.of operations.



Algorithm Design Techniques:

For a given problem, there are many ways to design algorithms for it.
The following is a list of several popular design approaches.

① Divide and Conquer: (merge sort, quick sort, ...)

- Divide the original problem into a set of sub pblms.
- Solve every sub problem individually, recursively
- Combine the solutions of the subproblems into a solution of the whole original problem.

② Dynamic Programming:

Dynamic programming is a technique for efficiently computing recurrences by storing partial results. It is a method of solving problems exhibiting the properties of overlapping subproblems and optimal substructure that takes much less time than naive methods.

③ Greedy Strategy: (Prims, Kruskals, knapsack ...)

Greedy algorithms seek to optimize a function by making choices which are the best locally but do not look at the global problem. The result is a good solution but not necessarily the best one. The greedy algorithm does not always guarantee the optimal solution however it generally produces solutions that are very close in value to the optimal.

④ Backtracking:

Backtracking algorithms try each possibility until they find the right one. It is a depth-first search of the set of possible solutions. During the search, if an alternative doesn't work, the search backtracks to the choice point, the place which presented different alternatives, and tries the next alternative. When the alternatives are exhausted, the search returns to the previous choice point and try the next alternative there. If there are no more choice points, the search fails.

⑤ Branch-and-Bound:
In a branch and bound algorithm a given sub problem which cannot be bounded, has to be divided into at least two new restricted sub problems. Branch and bound algorithms are methods for global optimization in nonconvex problems.

② INTRODUCTION TO ALGORITHM ANALYSIS

Purpose: When a programmer builds an algorithm during design phase of software development life cycle, he/she might not be able to implement it immediately. This is because programming comes in later part of the sw dvt life cycle. There is a need to analyze the algorithm at that stage. This will help in forecasting time of execution and amount of primary memory might occupy when it is implemented.

Analysis of Algorithm: Analysis of Algorithm means developing a formula or how fast the algorithm works based on the problem size.

The problem size could be:
- The no. of inputs/outputs in an algorithm.
  Eg: For a sorting algorithm, the no. of inputs is the total no. of elements to be arranged in a specific order. The no. of outputs is the total no. of sorted elements.
- The no. of operations involved in the algorithm.
  Eg: For a searching algorithm, the no. of operations is equal to the total no. of comparisons made with the search element.

In other cases, the analysis of an algorithm is to predict the resources that the algorithm requires, and such analysis is based on individual computational models. In the following several popular computational models are listed.

1. RAM (Random Access Machine): time and space (traditional serial computers).
2. PRAM: Parallel time, no. of processors, and read-and-write restrictions (SIMD type of parallel computers).
3. Message Passing Model: communication cost (no. of message), and computational cost (Distributed computing, Peer-to-Peer n/w)
4. Turing Machine: time and space (abstract theoretical machine).

The analysis of an algorithm is to evaluate the performance of the algorithm based on the given model and metrics.

1. Input size
2. Running time (worst-case and average case): The running time of an algorithm on a particular input is the no. of primitive operations or steps executed. Unless otherwise specified, we shall concentrate on finding only the worst case running time.
3. Order of growth: To simplify the analysis of algorithms, we are interested in the growth rate of the running time. ie., we only consider the leading terms of a time formula.

   eg: The leading term is n^2 in the expression

   $$n^2 + 100n + 50000.$$

Date / /

Complexity of Algorithm: It is very convenient to classify algorithms based on the relative amount of time or relative amount of space they require and specify the growth of time/space requirements as a function of the input size.

① **Space complexity:** Space complexity of an algorithm is the amount of memory it needs to run to completion. Generally, space needed by an algorithm is the sum of following two components:

- A fixed part that is independent of the characteristics (eg: number, size) of the inputs and outputs. This part typically includes the instruction space (ie., space for the code), space for simple variables and fixed-size component variables (also called aggregate), space for constants, and so on.
- A variable part that consist of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space ~~referenced by~~ needed by referenced variables, and the recursion stack space.

The space requirement $S(P)$ of any algorithm P may therefore be written as $S(P) = c + S_P$(instance characteristics), where c is a constant.

When analyzing the space complexity of an algorithm, we concentrate solely on estimating S_P(instance characteristics). For any given problem, we need first to determine which instance characteristics to use to ~~determine~~ measure the space requirements. This is very problem specific, and we resort to examples to illustrate the various possibilities.

To calculate the space complexity, we must know the memo required to store different datatype values.

1. 2 bytes to store Integer value,
2. 4 bytes to store floating point value,
3. 1 byte to store character value,
4. 6 OR 8 bytes to store double value.

Examples: 1

```
int square (int a)
{
    return a*a;
}
```

It requires 2 bytes of memory to store variable 'a' and another 2 bytes of memory is used for 'return value'.

That means, totally it requires 4 bytes of memory to complete its execution. And this 4 bytes of memory is fixed for any input value of 'a'. This Space complexity is said to be constant space complexity.

Example 2:

```
int sum (int A[], int n)
{
    int sum=0, i;
    for (i=0; i<n; i++)
        sum = sum + A[i];
    return sum;
}
```

It requires 'n*2' bytes of memory to store array variable 'a[]'.

2 bytes of memory for integer parameter 'n'.

4 bytes of memory for local integer variables 'sum' and 'i' (2 bytes each).

2 bytes of memory for 'return value'.

That means, totally it requires '2n+8' bytes of memory to complete its execution. Here, the amount of memory depends on the input value of 'n'. This space complexity is said to be Linear space complexity.

Example 3:

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

The problem instance is characterized by the specific values of a,b and c. Making the assumption that one word (2 bytes) is adequate to store the values of each of a,b,c, and the result, we see that the space needed by abc is independent of the instance characteristics. consequently, Sp(instance characteristics)=0.

Example 4:

```
Algorithm Sum(a,n)
{
     s:=0.0;
     for i:=1 to n do
          s:=s+a[i];
     return s;
}
```

This problem instances are characterized by 'n', the no.of elements to be summed. The space needed by 'n' is one word, since it is of type integer.

The space needed by 'a' is the space needed by variables of type array of floating point numbers. This is atleast 'n' words, since 'a' must be large enough to hold the 'n' elements to be summed. So we obtain $S_{sum}(n) \geq (n+3)$ (n for a[], one each for n, i, and s).

Example 5:

```
Algorithm Rsum(a,n)
{
    if (n≤0) then return 0.0;
    else return RSum (a,n-1)+ a[n];
```

In Algorithm RSum, as in the case of Sum, the instances are characterized by 'n'. The recursion Stack space includes space for the formal parameters, the local variables, and the return address. Assume that the return address requires only one word of memory. Each call to RSum requires at least 3 words (including space for the values of n, the return address, and a pointer to a[]). since the depth of recursion is n+1, the recursion stack space needed is $\geq 3(n+1)$.

Example 6:

```
int findMin (int[] x) {
    int k=0; int n=x.length;
  for (int i=1, i<n; i++) {
    if (x[i]<x[k]) {
        k =i;
  }}
    return k;
}
```

Space complexity of T(findMin, n) = n+2

(2) Time Complexity: The time $T(P)$ taken by a program P is the sum of the compile time and run (or execution) time. The compile time does not depend on the instance characteristics. Also, we may assume that a compiled program will be run several times without recompilation. Consequently, we concern ~~over~~ ourselves with just the run time of a program. This run time is denoted by t_p (instance characteristics).

Because many of the factors t_p depends on are not known at the time a program is conceived, it is reasonable to attempt only to estimate t_p. If we knew the characteristics of the compiler to be used, we could proceed to determine the no. of additions, subtractions, multiplications, divisions, compares, loads, stores, and so on, that would be made by the code for P.

So, we could obtain an expression for $t_p(n)$ of the form

$$t_p(n) = C_a\,ADD(n) + C_s\,SUB(n) + C_m\,MULL(n) + C_d\,DIV(n) + \cdots\cdots$$

where n denotes the instance characteristics, and C_a, C_s, C_m, C_d, and so on, respectively, denote the time needed for an addition, subtraction, multiplication, division, and so on, and ADD, SUB, MUL, DIV, and so on, are functions whose values are the numbers of additions, subtractions, multiplications, divisions, and so on, that are performed when the code for P is used on an instance with characteristic n.

The number of steps any program statement is assigned depends on the kind of statement.

Example:

1. Comments count as zero steps.
2. An assignment statement which doesnot involve any calls to other algorithms is counted as one step.
3. In an iterative statement such as for, while and repeat until statements, we consider the step counts only for the control part of the statement.

We can determine the number of steps needed by a program to solve a particular problem instance in one of 2 ways.

Method 1: We introduce a new variable, count, into the program. This is a global variable with initial value 0. Statements to increment count by the appropriate amount are introduced into the program. This is done so that each time a statement in the original problem is executed, count is incremented by the step count of that statement.

Examples: 1

→ This is b/s i has to be incremented to n+1 before the for loop can terminate

→ steps per execution

Statement	s/e	frequency	total steps
1. Algorithm Sum(a,n)	0	—	0
2. {	0	—	0
3. s:=0.0;	1	1	1
4. for i:=1 to n do	1	n+1	n+1
5. s:=s+a[i];	1	n	n
6. return s;	1	1	1
7. }	0	—	0
Total			2n+3

Example 2:

i need to be incremented upto m+1 before termination

Statement	s/e	frequency	total steps
1. Algorithm Add (a,b,c,m,n)	0	—	0
2. {	0	—	0
3 for i:=1 to m do	1	m+1	m+1
4 for j:=1 to n do	1	m(n+1)	mn+m
5 c[i,j]:= a[i,j]+b[i,j];	1	mn	mn
6 }	0	—	0
Total			2mn+2m+1

Example 3:

```
Algorithm Fibonacci (n)
// compute the nth Fibonacci number.
{
    if (n≤1) then
        write(n);
    else
    {
        fnm2:=0; fnm1:=1;
        for i:=2 to n do
        {
            fn:= fnm1 +fnm2;
            fnm2:= fnm1; fnm1:= fn;
        }
        write (fn);
    }
}
```

To analyze the time complexity of this algorithm, we need to consider the 2 cases:

① n=0 or 1

② $n>1$.

When $n=0$ or 1, lines 4 and 5 get executed once each. Since each line has an s/e of 1, the total step count for this case is 2.

When $n>1$, lines 4, 8, and 14 are each executed once. Line 9 gets executed n times, and lines 11 and 12 get executed $n-1$ times each (note that the last time line 9 is executed, i is incremented to $n+1$, and the loop exited). Line 8 has an s/e of 2, Line 12 has an s/e of 2, and line 13 has an s/e of 0. The remaining lines that get executed have s/e's of 1.

The total step for the case $n>1$ is therefore $4n+1$.

Ex:

```
int fun1 (int n) {
if (n <= 1) return n;
return 2 * fun1 (n-1);
}
int fun2 (int n) {
if (n <= 1) return n;
return fun2 (n-1) + fun2 (n-1);
}
```

Date / /

Best, Worst and Average Case complexity Analysis

The worst case running time of an algorithm is the function defined by the maximum number of steps taken on any instance of size n.
The worst case running time of an algorithm is an upper bound on the running time for any input. Knowing it gives us a guarantee that the algorithm will never take any longer time

The Average case running time of an algorithm is the function defined by an average number of steps taken any instance of Size n.

The best case running time of an algorithm is the function defined by the minimum number of steps taken on any instance of size n.

Example 1: Linear search / sequential search

Search a key 'k' in a set $A = \{a_1, a_2, a_3 \ldots a_n\}$
$A = \{5, 7, 2, 1, 6, 8, 10, 4, 15, 13\}$ $n = 10$

```
int search (int i) {
      if (A[i]==k)
      return (i);
     else
       return (-1); // not found
  }
```

Performance of search = No.of comparisons needed to search

Best case:

Search(5) in set A[i]   k=5
```
for (int i =1 to n) {
    if (A[i] = =5)
    return (i); }
```
In this case, we can find the key in first search. So this is the best case comparison.

Worst Case:

Search (13) in set A[i]

In this case, we find the key in 10th position. So this is the worst case comparison. Search value not in the list is also included in worst case.

Average Case:

Average case is some where in b/w around n/2 elements to get the search elements.

Search (6) in set A[i]

So, Best case linear search is 1 comparison.
Worst case linear search is n comparisons.
Average case linear search is n/2 comparisons.

Linear search is also called as sequential search, is a very simple method used for searching an array for a particular value. It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found. Linear search is mostly used to search an unsorted list of elements.

Complexity of Linear search Algorithms:

Linear search executes in O(n) time where n is the number of elements in the array. Obviously, the best case of linear search is when the search value is equal to the first element of the array. In this case only one comparison will be made. Likewise, the worst case will happen when either search value is not present in the array or it is equal to the last element of the array. In both cases, n comparisons will have to be made.

Example 2: Binary Search

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. 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 until the value is found or the interval is empty.

If searching for 23 in the 10-element array:

2	5	8	12	16	23	38	56	72	91

23 > 16, take 2nd half

L	2	5	8	12	16	23	38	56	72	91	H

23 < 56, take 1st half

					L				H
					23	38	56	72	91

Found 23, Return 5th position

					L	H			
					23	38			

```
binary_search (A, target)
  lo = 1, hi = size(A)
  while lo <= high
     mid := lo+(hi -lo)/2
   if  A[mid] == target
     return mid
  else if  A [mid] < target
      lo = mid+1
  else
       hi = mid -1
// target was not found.
```

Complexity of Binary Search Algorithm:

The complexity of the binary search algorithm can be expressed as $f(n)$, where n is the number of elements in the array. The complexity of the algorithm is calculated depending on the number of comparisons that are made. In the binary search algorithm, we see that with each comparisons, the size of the segment where search has to be made is reduced to half. Thus, we can say that, inorder to locate a particular ~~pr~~ value in the array, the total number of comparisons that will be made is given as $2f(n) > n$ or $f(n) = \log_2 n$

Best case performance $\rightarrow O(1)$

Worst case and Average case performance is $O(\log n)$

Example 3: Bubble Sort

Bubble sort is a very simple method that sorts the array elements by repeatedly moving the largest element to the highest index position of the array segment. In bubble sorting, consecutive adjacent pairs of elements in the array are compared with each other. If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one. This process will continue till the list of unsorted elements exhausts.

First Pass →

54	26	93	17	77	31	44	55	20	Exchange.
26	54	93	17	77	31	44	55	20	No exchange
26	54	93	17	77	31	44	55	20	Exchange
26	54	17	93	77	31	44	55	20	Exchange.
26	54	17	77	93	31	44	55	20	exchange.
26	54	17	77	31	93	44	55	20	Exchange.
26	54	17	77	31	44	93	55	20	exchange.
26	54	17	77	31	44	55	93	20	exchange.
26	54	17	77	31	44	55	20	93	93 in place after first pass.

The main advantage of Bubble sort is the simplicity of the algorithm.

Camlin Page
Date / /

Observe that after the end of the 1st pass, the largest element is placed at the highest index of the array. All the other elements are still unsorted. This process will continue through many passes till the list of unsorted elements exhausts.

```
BUBBLE-SORT (A,N)
Step1 : Repeat step 2 for I = 0 to N-1
Step2 :         Repeat for J = 0 to N-I
Step3 :                 IF A[J] > A[J+1]
                        SWAP A[J] and A[J+1]
                [END OF INNER LOOP]
        [END OF OUTER LOOP]
Step4: EXIT
```

Complexity of Bubble sort:

Space complexity of Bubble sort is $O(1)$, because only single additional memory space is required i.e for temp variable (for swapping).

The best-case time complexity will be $O(n)$, it is when the list is already sorted.

The complexity of any sorting algorithm depends upon the number of comparisons. In bubble sort, there are N-1 passes in total. N-1 comparisons will be done in 1st pass, N-2 in 2nd pass, N-3 in 3rd pass and so on. So the total number of comparisons will be

$$\boxed{\begin{array}{l}(N-1)+(N-2)+(N-3)+\cdots\cdots+3+2+1 \\ \text{Sum} = N(N-1)/2 \Rightarrow O(N^2).\end{array}}$$

Hence the Worst and Average case complexity is $O(n^2)$.

Example 4: Insertion Sort

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hand.
The main idea behind insertion sort is that it inserts each item into its proper place in the final list. To save memory, most implementations of the insertion sort algorithm work by moving the current data element pass the already sorted values and repeatedly interchanging it with the preceding value until it is in its correct place.

9	7	6	15	16	5	10	11
9	7	6	15	16	5	10	11
7	9	6	15	16	5	10	11
6	7	9	15	16	5	10	11
6	7	9	15	16	5	10	11
6	7	9	15	16	5	10	11
5	6	7	9	15	16	10	11
5	6	7	9	10	15	16	11
5	6	7	9	10	11	15	16

```
void insertion_sort (int A[], int n)
{
    for(int i=0;i<n;i++) {
        int temp = A[i];
        int j =i; //check whether the adj. elt in left is > or < than current elt
          while (j>0 && temp < A[j-1]) { //moving the left side elt to 1 position forward.
             A[j] = A[j-1];
             j =j-1;
          } //moving current elt to its correct position
          A[j] = temp;
    }
}
```

Complexity of Insertion sort:

For insertion sort, the best case occurs when the array is already sorted. In this case, the running time of the algorithm is $O(n)$. This is because, ~~the~~ during each iteration, the 1st element from the unsorted set is compared only with the last element of the sorted set of the array.

Similarly, the worst case of the insertion sort alg^n occurs when the array is sorted in the reverse order. In this worst case, the first element of the unsorted set has to be compared with almost every element in the sorted set. Furthermore, every iteration of the inner loop will have to shift the elements of the sorted set of the array before inserting the next elements. Therefore, in the worst case, insertion sort ~~is~~ has a quadratic running time $O(n^2)$.

Average case is also $O(n^2)$.

Example 5: Selection Sort

Selection sort is a simple sorting algorithm. The sorting algorithm is an in-place comparison-based algorithm in which the list is divided into 2 parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one elt to the right.

14	33	27	10	35	19	42	44

For the 1st position in the sorted list, the whole list is scanned sequentially. The 1st position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

14	33	27	10	35	19	42	44

So we replace 14 with 10. After 1 iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

10	33	27	14	35	19	42	44

For the 2nd position, where 33 is residing, we start scanning the rest of the list in a linear manner. We find 14 is the second lowest value in the list and it should appear at the second place we swap these values.

10	33	27	14	35	19	42	44

After two iterations, two least values are positioned at the beginning in a sorted manner.

10	14	27	33	35	19	42	44	lowest is 19.
10	14	19	33	35	27	42	44	lowest is 27.
10	14	19	27	35	33	42	44	lowest is 33.
10	14	19	27	33	35	42	44	sorted list

```
void selection_sort (int A[], int n) {
    //temporary variable to store the position of min elt
  int minimum;
   // reduces the effective size of the array by one in each
    iteration.
  for (int i=0; i< n-1; i++) {
  // assuming the 1st elt to be the min of the unsorted array
  minimum = i;
  // gives the effective size of the unsorted array.
  for (int j = i+1 ; j<n; j++) {
       if ( A[j] < A[minimum]) {
  // find the minimum element
             minimum = j; }}
  // Putting minimum element on its proper position.
          swap (A[minimum], A[i]); }}
```

Complexity of ~~the~~ Selection Sort:

To find the minimum element from the array of N elements, N-1 comparisons are required. After putting the min elt in its proper position, the size of an unsorted array reduces to N-1 and then N-2 comparisons are required to find the minimum in the unsorted array.

Therefore, $(N-1)+(N-2)+\cdots+1$
$= (N(N-1))/2$ comparisons and N swaps
swaps result in the overall complexity of $O(N^2)$.

The worst case, Best case and Average case complexity of selection sort is $O(N^2)$.

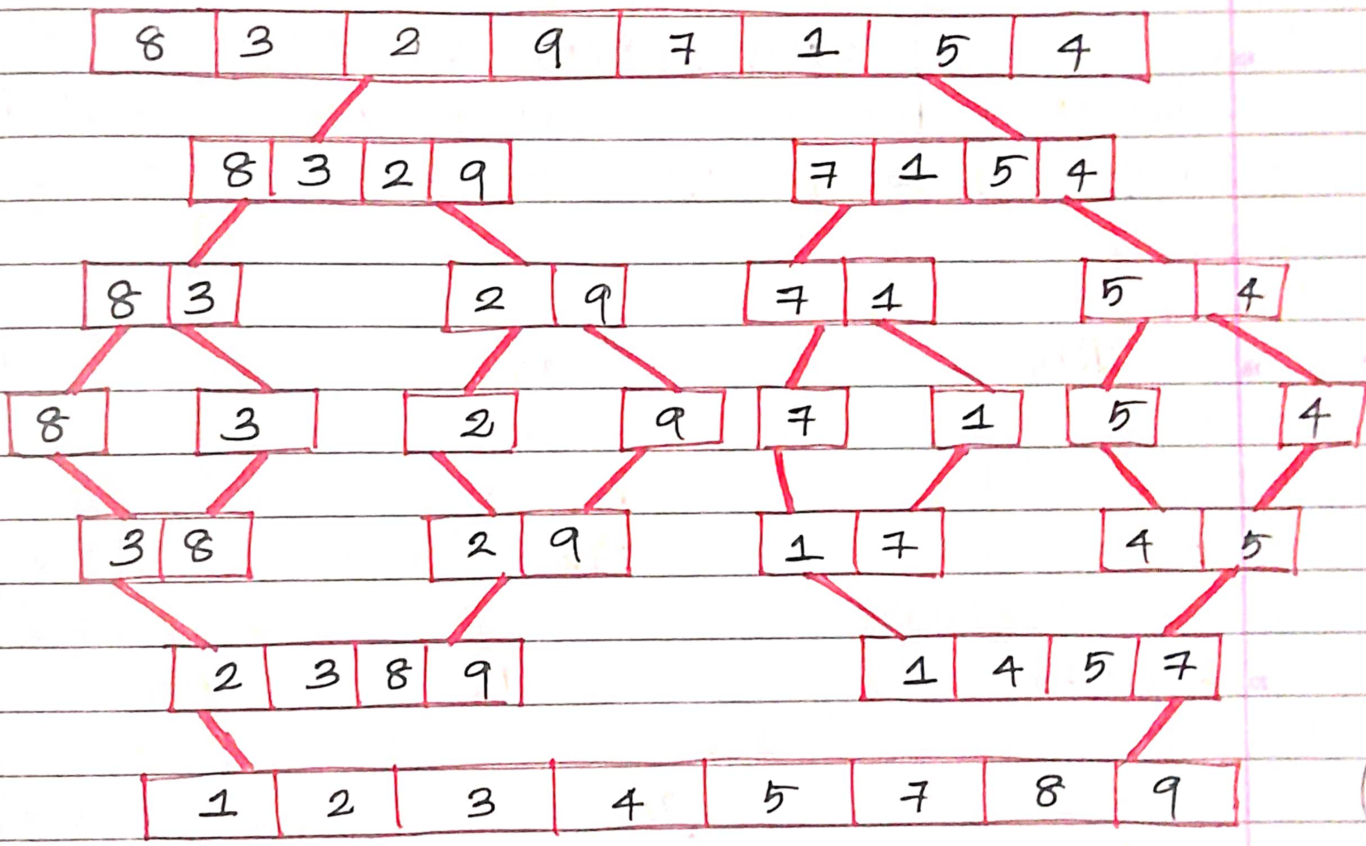
Example 6: MERGE Sort

The Merge Sort algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

- **Divide**: Divide the n-element sequence to be sorted into 2 ~~seqe~~ subsequences of n/2 elements each.
- **Conquer**: Sort the 2 subsequences recursively using Merge Sort.
- **Combine**: Merge the two sorted subsequences to produce the sorted answer.

```
MERGE (A, P, q, r)
  n1 = q - P + 1
  n2 = r - q
  let L[1 .... n1+1] and R[1 .... n2+1] be new arrays
  for i = 1 to n1
      L[i] = A[P+i-1]
  for j = 1 to n2
      R[j] = A[q+j]
  L[n1+1] = ∞
  R[n2+1] = ∞
  i = 1
  j = 1
```

```
for k=p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i+1
    else A[k] = R[j]
        j = j+1
```



Complexity of Merge Sort:

Divide: The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n) = \Theta(1)$.

Conquer: We recursively solve two subproblems each of size $n/2$, which contributes $2T(n/2)$ to the running time.

Combine: MERGE procedure on an n-element subarray takes $\Theta(n)$ time, and so $C(n) = \Theta(n)$.

When we add the functions $D(n)$ and $C(n)$ for the merge sort analysis, we are adding a function that is $\Theta(n)$ and a function that is $\Theta(1)$. This sum is a linear function of n, that is $\Theta(n)$. Adding it to the $2T(n/2)$ term from the 'conquer' step gives the recurrence for the worst case running time $T(n)$ of Merge sort:

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

$T(n)$

Fig (a)

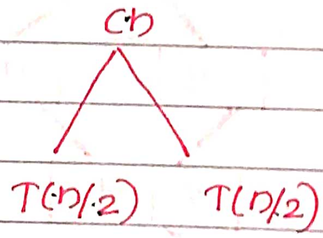


Fig (b)

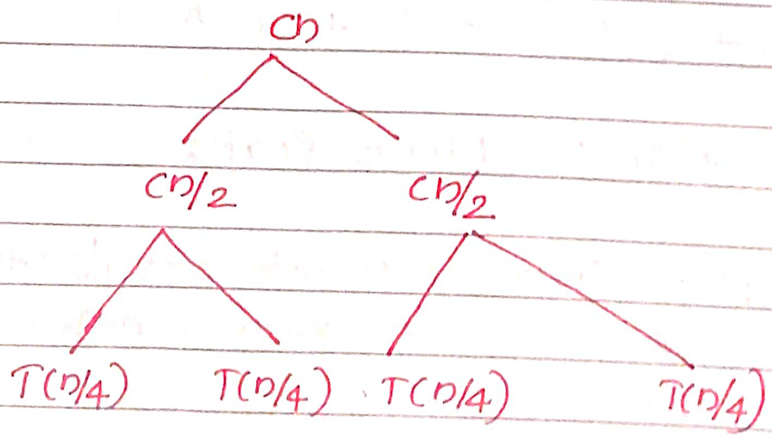


Fig (c)

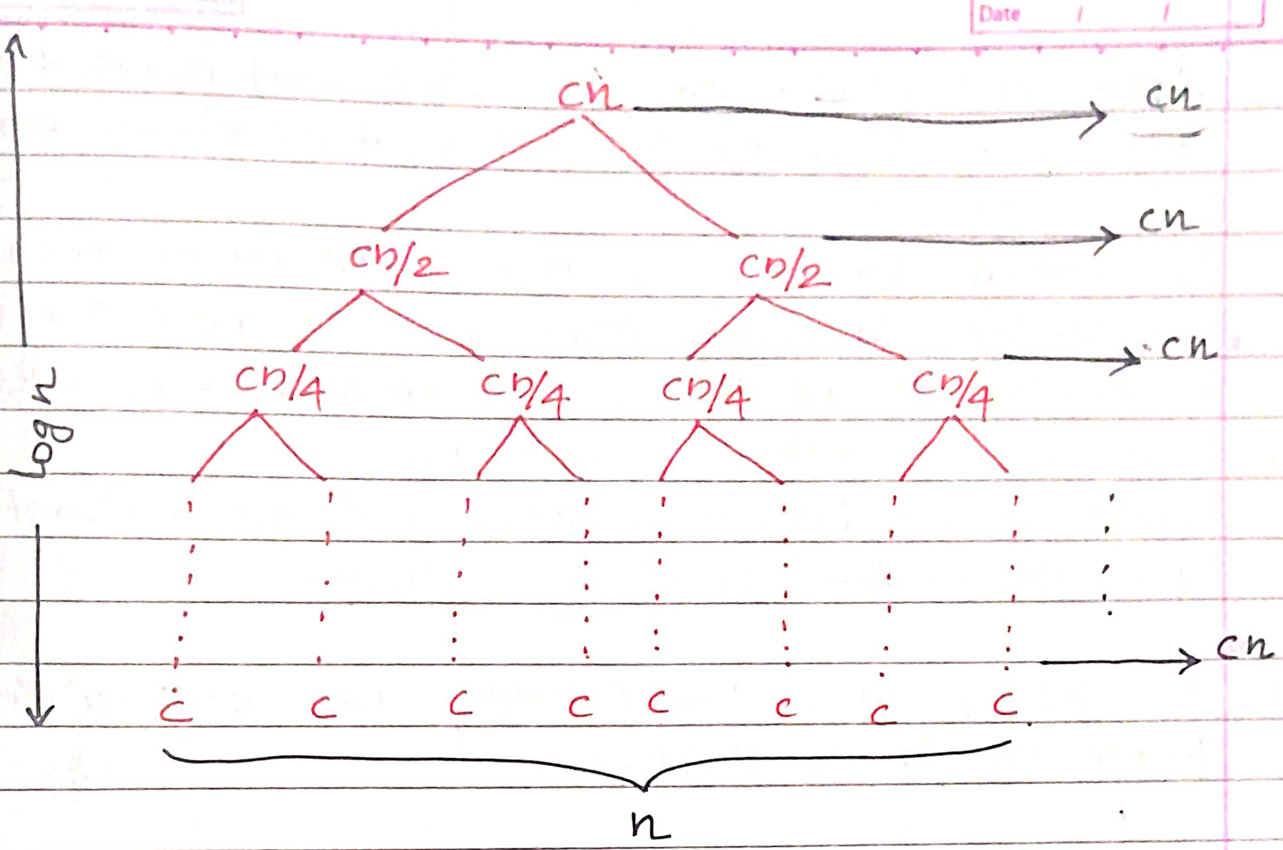


Fig (d)

Total : cn log n + cn

Ignoring low order term & constant, will get

→ $O(n \log n)$

Part (a) of the fig shows T(n), which we expand in Part (b) into an equivalent tree representing the recurrence. The 'cn' term is the root (the cost incurred at the top level of recursion), and the 2 subtrees of the root are the two smaller recurrences T(n/2).
Part (c) shows this process carried one step further by expanding T(n/2). The cost incurred at each of the two subnodes at the second level of recursion is n/2.
We continue expanding each node in the tree by breaking it into its constituent parts as determined by the recurrence, until the problem sizes get down to 1, each with a cost of c. Part (d) shows the recursion tree.

Next, we add the costs across each level of the tree. The top level has total cost cn, the next level down has total cost $c(n/2) + c(n/2) = cn$, the level after that has total cost $c(n/4) + c(n/4) + c(n/4) + c(n/4) = cn$, and so on.

To compute the total cost represented by the recurrence, we simply add up the costs of all the levels. The recursion tree has $\log n + 1$ levels, each costing cn for a total cost of $cn(\log n + 1) = cn\log n + cn$.
Ignoring the low-order term and the constant c gives the desired result of $O(n\log n)$.

The Best, Worst and Average case complexities of Merge sort is $\Theta(n\log n)$.

Example 7 : Quick Sort

It is used on the principle of divide-and-conquer. Quicksort algorithm is also known as partition exchange sort.
The Quick sort algorithm works as follows:

1. Select an element Pivot from the array element.
2. Rearrange the elements in the array in such a way that all elements that are less than the pivot appear before the Pivot and all elements that ar greater than the Pivot element come after it. After such a partitioning, the Pivot is placed in its final Position. This is called the Partition Operation.
3. Recursively sort the two sub-arrays thus obtained.

```
QUICKSORT (A,P,r)
    if P<r
        q = PARTITION (A,P,r)
        QUICKSORT (A, P, q-1)
        QUICKSORT (A, q+1, r)
```

```
PARTITION(A, p, r)
x = A[r]          → Pivot
i = p-1
for j = p to r-1
    if A[j] ≤ x
        i = i+1
        exchange A[i] with A[j]
exchange A[i+1] with A[r]
return
```

i

p,j							r,x
2	8	7	1	3	5	6	4

Set A[r] = x = 4 as the Pivot element.

j = 2, if 2 ≤ 4 ⇒ i = 1 then exchange 2 with 2, ↑ j

p,i	j						r
2	8	7	1	3	5	6	4

j = 8, if 8 ≰ 4, then no change, increment j

p,i		j					r
2	8	7	1	3	5	6	4

j = 7, if 7 ≰ 4 then no exchange, increment j

p,i			j				r
2	8	7	1	3	5	6	4

j = 1, if 1 ≤ 4, increment i, exchange 1 with 8, ↑ j

p	i			j			r
2	1	7	8	3	5	6	4

p		i			j		r
2	1	3	8	7	5	6	4

p		i				j	r
2	1	3	8	7	5	6	4

p		i					r
2	1	3	8	7	5	6	4

increment stops.

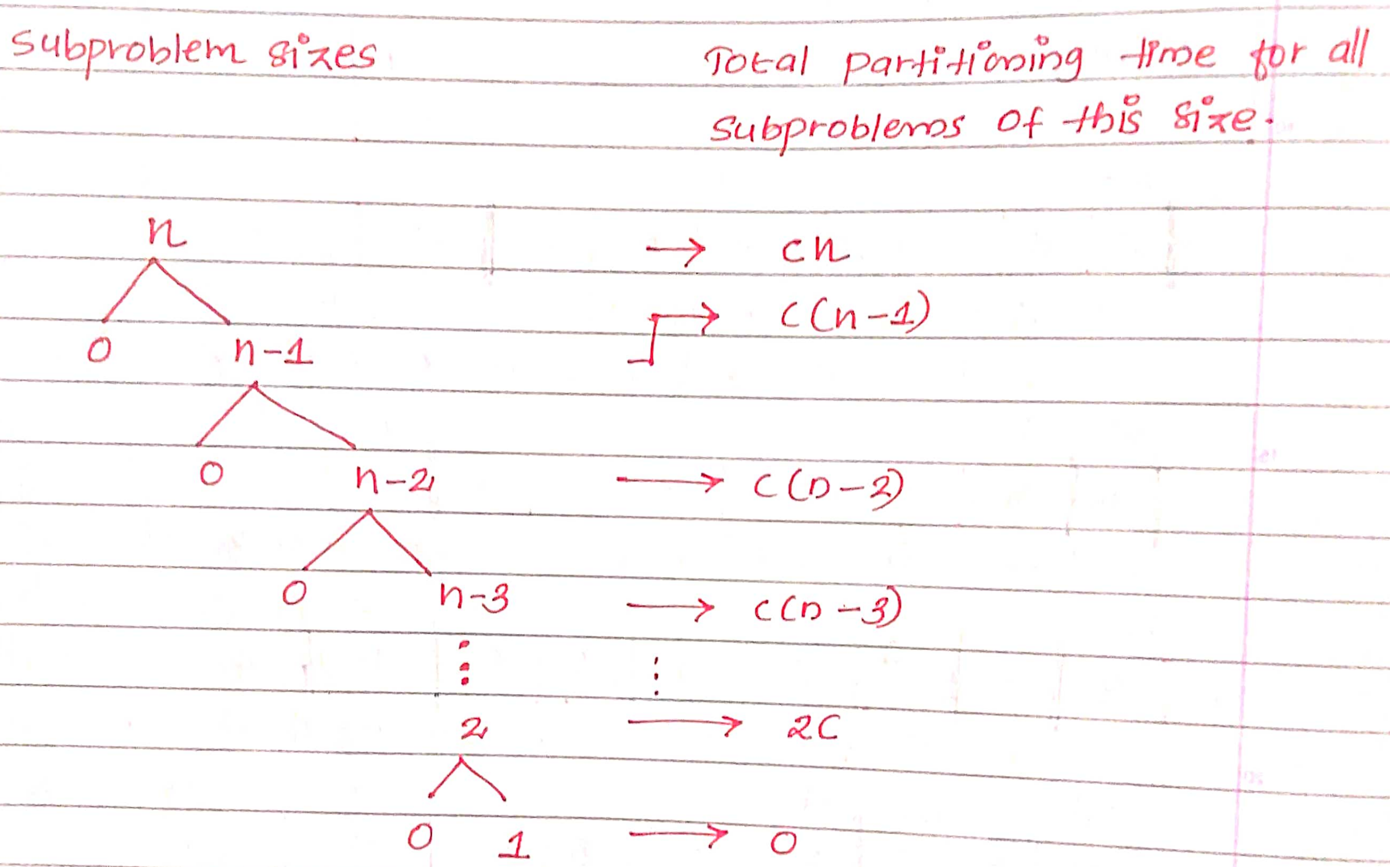
exchange A[i+1] with A[r]

2	1	3	4	7	5	6	8

Recursively call this QUICKSORT algm at RIGHT & LEFT SUB ARRAYS

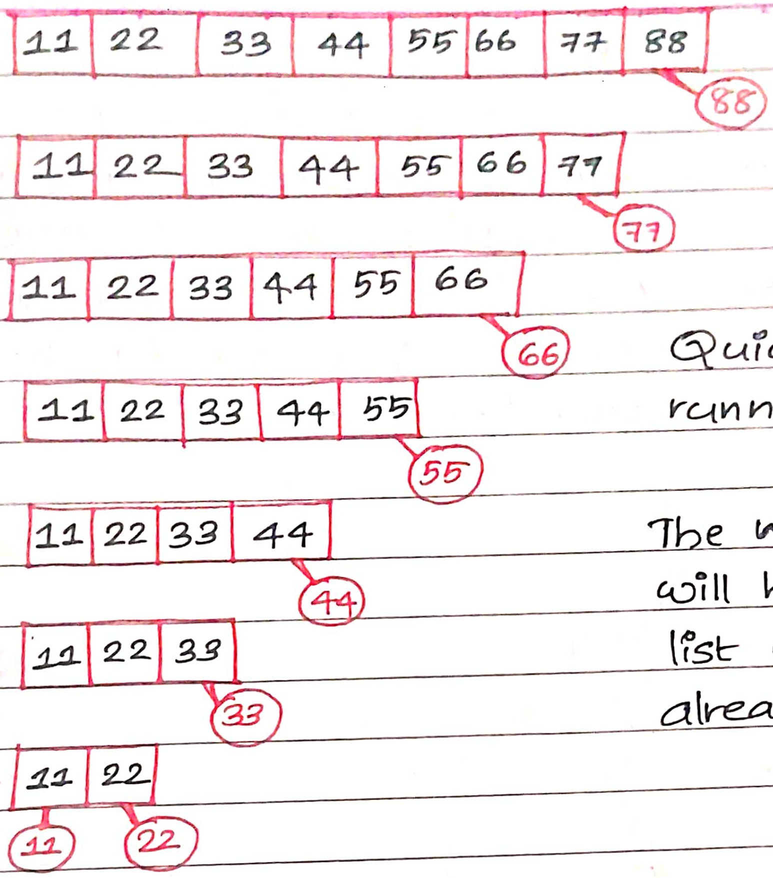
Worst-case running Time:

When quicksort always has the most unbalanced partitions possible, then the original call takes 'cn' times for some constant c, the recursive call on $n-1$ elements takes $c(n-1)$ time, the recursive call on $n-2$ elements takes $c(n-2)$ time, and so on. Here's a tree of the subproblem sizes with their partitioning times:



When we total up the partitioning times for each level, we get

$$cn + c(n-1) + c(n-2) + \cdots + 2c =$$
$$c(n + (n-1) + (n-2) + \cdots + 2)$$
$$= c((n+1)(n/2) - 1)$$
$$= (n+1)(n/2) - 1$$
$$= \frac{2(n+1)n}{2} - 1$$
$$= \frac{(2n+2)n}{2} - 1$$
$$= 2n^2 + 2n - 2 \quad = \Theta(n^2)$$

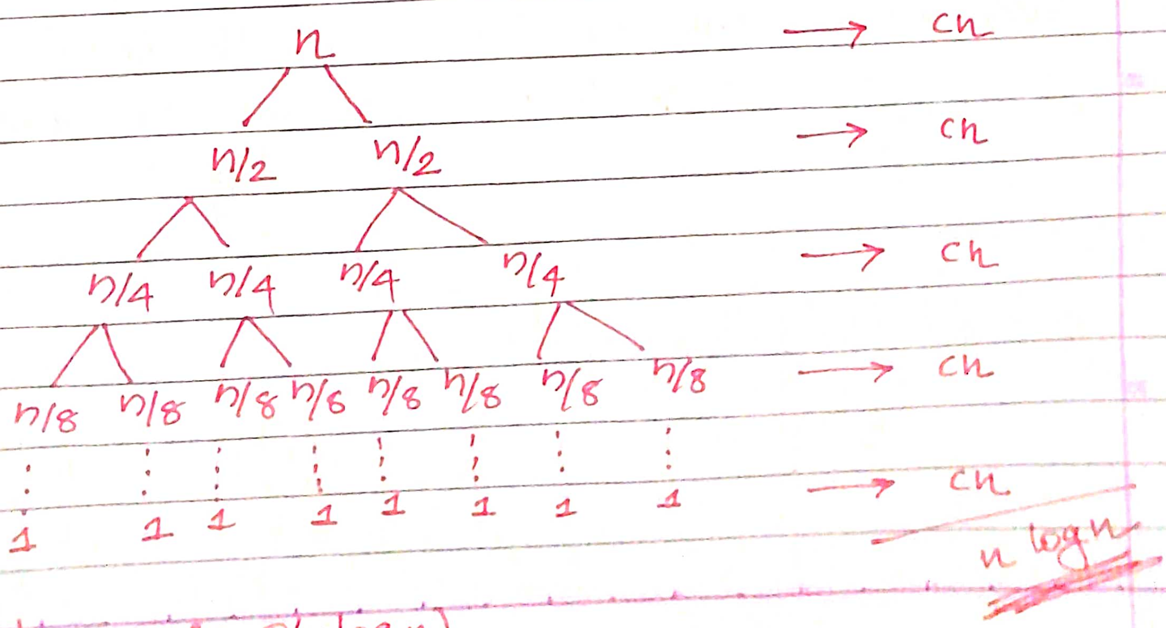


Quicksort's worst-case running time is $\Theta(n^2)$.

The worst-case scenario will happen when the list of elements are already sorted.

Best-Case running time:

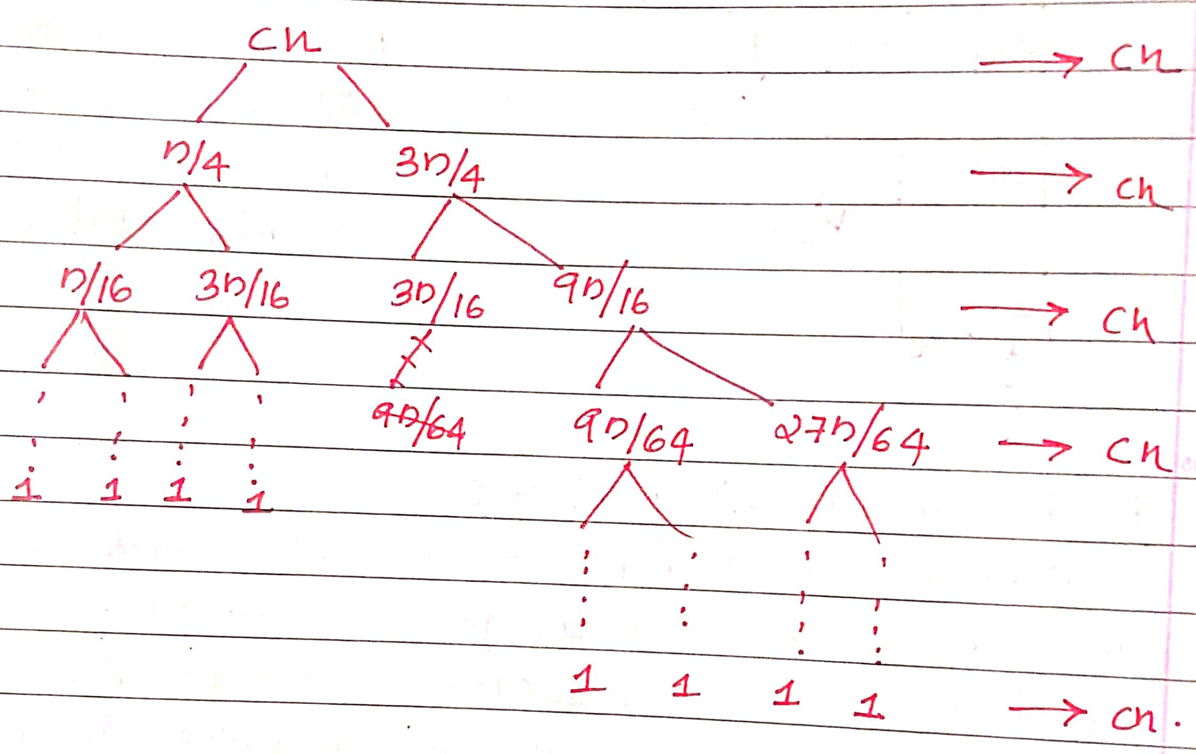
Quicksort's best case occurs when the partitions are as evenly balanced as possible: their sizes either are equal or are within 1 of each other. This case occurs if the subarray has an even number 'n' of elements and one partition has $n/2$ elements with the other having $n/2-1$.



Best case is $\Theta(n \log n)$

Average-Case running time:

First, lets imagine that we don't always get evenly balanced partitions, but that we always get at worst a 3-1 split. That is, imagine that each time we partition, one side gets $3n/4$ elements and the other side gets $n/4$. Then the tree of subproblem sizes and partitioning times would like this:



Total $\Theta(n\log n)$

So, Worst-case of Quick sort is $\Theta(n^2)$.

Best-case & Average case of Quick sort is $\Theta(n\log n)$.

RECURRENCE EQUATIONS

A recurrence is an equation or inequality that describes a function in terms of ~~the~~ its value on smaller inputs. To solve the recurrence relation means to obtain a function defined on the natural numbers that satisfies the recurrence. For example, the worst-case running time $T(n)$ of the MERGE-SORT procedure is described by the recurrence.

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

There are many methods for solving recurrence relations.

1. Iteration Method.
2. Recursion Tree Method.
3. Master's Theorem.

Iteration Method:

In iteration method the basic idea is to ~~express~~ expand the recurrence and express it as a summation of terms dependent only on 'n' and the initial conditions.

Example 1:

Consider the recurrence: $T(n) = 3T(\lfloor \frac{n}{4} \rfloor) + n$.

Solution

We iterate it as follows:

$$T(n) = n + 3T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) = n + 3\left(\left\lfloor \frac{n}{4} \right\rfloor + 3T\left(\left\lfloor \frac{n}{16} \right\rfloor\right)\right)$$

$$= n + 3\left(\left\lfloor \frac{n}{4} \right\rfloor + 3\left(\left\lfloor \frac{n}{16} \right\rfloor + 3T\left(\left\lfloor \frac{n}{64} \right\rfloor\right)\right)\right)$$

$$= n + 3\left\lfloor \frac{n}{4} \right\rfloor + 9\left\lfloor \frac{n}{16} \right\rfloor + 27T\left(\left\lfloor \frac{n}{64} \right\rfloor\right)$$

$$\leq n + \frac{3n}{4} + \frac{9n}{16} + \cdots + 3^i T\left(\frac{n}{4^i}\right)$$

The series terminates when $\frac{n}{4^i} = 1 \Rightarrow$

$n = 4^i$ or $i = \log_4 n$.

$$T(n) \leq n + \frac{3n}{4} + \frac{9n}{16} + \frac{27n}{64} + \cdots + 3^{\log_4 n} T(1)$$

$$\leq n + \frac{3n}{4} + \frac{9n}{16} + \frac{27n}{64} + \cdots + 3^{\log_4 n}\, \Theta(1)$$

$$\leq n \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i + \Theta\left(n^{\log_4 3}\right) \text{ as } 3^{\log_4 n} = n^{\log_4 3}$$

$$\leq n \cdot \frac{1}{1 - \frac{3}{4}} + O(n) \text{ as } \log_4 3 < 1$$

i.e., $\Theta\left(n^{\log_4 3}\right) = O(n)$

Example 2:

Solve the recurrence relation by iteration: $T(n) = T(n-1) + n^4$

Solution

$$T(n) = T(n-1) + n^4$$

$$= [T(n-2) + (n-1)^4] + n^4$$

$$= T(n-2) + (n-1)^4 + n^4$$

$$= T(n-3) + (n-2)^4 + (n-1)^4 + n^4$$

$$\cdots\cdots\cdots$$

$$= n^4 + (n-1)^4 + (n-2)^4 + \cdots + 2^4 + 1^4 + T(0)$$

$$= \sum_{i=1}^{n} i^4 + T(0)$$

$$T(n) = \Theta(n^4)$$

Example 3: Solve the following recurrence relation:

$$T(n) = 2T(n/2) + 3n^2 \qquad T(1) = 11.$$

Solution:

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

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

$$= 3n^2 + 2\left[3\left(\frac{n}{2}\right)^2 + 2T\left(\frac{n}{4}\right)\right]$$

$$= 3n^2 + 2\cdot 3\left(\frac{n}{2}\right)^2 + 4\left[3\cdot\left(\frac{n}{4}\right)^2 + 2T\left(\frac{n}{8}\right)\right]$$

$$= 3n^2 + \frac{3n^2}{2} + 4\cdot 3\frac{n^2}{4^2} + 2^3\cdot T\left(\frac{n}{2^3}\right)$$

$$= 3n^2 + \frac{3n^2}{2} + \frac{3n^2}{4} + 2^3 T\left(\frac{n}{2^3}\right)$$

$$= 3n^2 + \frac{3n^2}{2} + \frac{3n^2}{2^2} + 2^3 T\left(\frac{n}{2^3}\right)$$

$$\therefore \quad T(n) = 3n^2 + \frac{3n^2}{2^1} + \frac{3n^2}{2^2} + \cdots + 2^i T\left(\frac{n}{2^i}\right)$$

The series terminates when

$$\frac{n}{2^i} = 1 \Rightarrow n = 2^i \text{ or } i = \log_2 n$$

$$\therefore \quad T(n) = 3n^2 + \frac{3n^2}{2} + \frac{3n^2}{2^2} + \cdots + 2^{\log_2 n} T(1)$$

$$= 3n^2 + \frac{3n^2}{2} + \frac{3n^2}{2^2} + \cdots + n^{\log_2 2} T(1)$$

$$= 3n^2 \left[1 + \frac{1}{2} + \frac{1}{2^2} + \cdots\right] + n \cdot 11 \qquad \because T(1) = 11$$

$$\leq 3n^2 \cdot \left[\frac{1}{1 - 1/2}\right] + 11n$$

$$\leq 3n^2 \cdot 2 + 11n$$

$$\leq 6n^2 + 11n$$

$$\therefore T(n) = O(n^2).$$

Exercise

(1) Solve the recurrence relation

$T(n) = 1$ for $n = 1$

$T(n) = 2T(n-1)$ for $n > 1$

(2) Solve $T(n) = T(n-1) + 1$ and $T(1) = \Theta(1)$.

(3) Solve $T(n) = T(n/3) + n^{4/3}$

(4) Solve $T(n) = T(n-1) + 1/n$.

Solve the following recurrence relation in terms of 'n' using the iteration technique.

$$\boxed{\begin{aligned} T(n) &= 4T(n/4)+4 \\ T(1) &= 1 \end{aligned}}$$

K	T(n)
1	$T(n) = 4T(n/4)+4$
	$T(n/4) = 4T\left(\frac{n/4}{4}\right)+4$
	$\boxed{T(n/4) = 4T(n/4^2)+4}$
2	$T(n) = 4\left(4T(n/4^2)+4\right)+4$
	$T(n) = 4^2T(n/4^2)+4^2+4$
	$T(n/4^2) = 4T\left(\frac{n/4^2}{4}\right)+4$
	$\boxed{T(n/4^2) = 4T(n/4^3)+4}$
3	$T(n) = 4^2\left(4T(n/4^3)+4\right)+4^2+4$
	$= 4^3T(n/4^3)+4^3+4^2+4$
⋮	⋮

$\vdots \qquad \vdots$

$k \qquad T(n) = 4^k T(n/4^k) + \underbrace{4^k + 4^{k-1} + \dots 4^1}_{\text{Summation}}$

$$T(n) = 4^k T(n/4^k) + \sum_{i=1}^{k} 4^i$$

$$\boxed{\sum_{i=1}^{k} ar^i = a\left(\frac{1-r^k}{1-r}\right)}$$

↳ Geomatric Series Summation

$$T(n) = 4^k T(n/4^k) + \left[\frac{1-4^k}{1-4}\right]$$

$$= 4^k T(n/4^k) + \left[\frac{1-4^k}{-3}\right]$$

$$T(n) = 4^k T(n/4^k) + \left[\frac{4^k - 1}{3}\right]$$

$\because$ Stop when $n=1$. ie., $\underline{\underline{T(1) = 1}}$.

$$\boxed{\begin{aligned} (n/4^k) &= 1 \\ n &= 4^k \\ \log_4 n &= k \end{aligned}}$$

$$T(n) = 4^{\log_4 n} T(n/4^{\log_4 n}) + \left[\frac{4^{\log_4 n} - 1}{3}\right]$$

$$T(n) = n T(n/n) + \left[\frac{n-1}{3}\right]$$

$$\Rightarrow n \cdot T(1) + \left[\frac{n-1}{3}\right]$$

$$\Rightarrow n \cdot 1 + \left[\frac{n-1}{3}\right] \qquad \because \underline{\underline{T(1) = 1}}$$

$$\Rightarrow n + \left[\frac{n-1}{3}\right]$$

$$\Rightarrow \left[\frac{4n-1}{3}\right]$$

Running Time = $\underline{\underline{O(n)}}$.

Solve the following recurrence relation using the iteration method.

$$\boxed{T(n) = 2T(n/2) + n}$$

k	$T(n)$
$k=1$	$T(n) = 2T(n/2) + n$

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

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

$$\boxed{T(n/2) = 4T(n/4) + 2n}$$

$k=2$

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

$T(n) = 8T(n/4) + 4n + n$

$T(n/4) = 2T\left(n/4/2\right) + \frac{n}{4}$

$$\boxed{T(n/4) = 2T(n/8) + n/4}$$

$k=3$

$T(n) = 8\left[2T(n/8) + n/4\right] + 4n + n$

$= 16T(n/8) + 4n + 4n + n$

$= 16T(n/8) + 8n + n$

$$T(n) = 2^3\left[2T\left(n/2^4\right) + 4\left(n/2^3\right)\right] + 4n + 4n + 4n$$

$$T(n) = 2^4\,T\left(n/2^4\right) + 4n + 4n + 4n + 4n.$$

$$\vdots$$

$$\boxed{T(n) = 2^i\,T\left(n/2^i\right) + i(4n).}$$ General form.

$$\frac{n}{2^i} = 1 \longrightarrow n = 2^i \longrightarrow \log_2 n = i$$

$$T(n) = 2^{\log_2 n}\,T\left(n/2^{\log_2 n}\right) + \log_2 n\,(4n)$$

$$\left\{\boxed{2^{\log_2 n}} \Rightarrow n^{\log_2 2} = \underline{\underline{n}}\right\}$$

$$n\,T(n/n) + 4n\log_2 n$$

$$n \cdot T(1) + 4n\log_2 n$$

$$n \cdot (4) + 4n\log_2 n$$

$$4n + 4n\log_2 n = \underline{O(n\log n)}$$

$$\boxed{T(n) = 2T(n/2) + 4n} \qquad \boxed{T(1) = 4}$$

$$T(n) = 2\underline{\underline{T(n/2)}} + 4n$$

$$\hookrightarrow T(n/2) = 2T(n/2/2) + 4(n/2)$$

$$\hookrightarrow \boxed{T(n/2) = 2T(n/2^2) + 4(n/2)}$$

$$T(n) = 2\left[2T(n/2^2) + 4(n/2)\right] + 4n$$

$$T(n) = 2^2 \underline{\underline{T(n/2^2)}} + 4n + 4n$$

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

$$\hookrightarrow \boxed{T(n/2^2) = 2T(n/2^3) + 4n}$$

$$T(n) = 2^2\left[2T(n/2^3) + 4n\right] + 4n + 4n$$

$$= 2^3 \underline{\underline{T(n/2^3)}} + 4n + 4n + 4n$$

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

$$\boxed{T(n/2^3) = 2T(n/2^4) + 4(n/2^3)}$$

$$T(1) = 4 \qquad T(n) = 2T(n/2) + 4n$$

$T(n) \Rightarrow 2T(n/2) + 4n$ $\rightarrow T(n/2) \Rightarrow 2[T(n/2/2) + 4(n/2)]$

$\boxed{T(n/2) \Rightarrow 2T(n/2^2) + 4(n/2)}$

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

$T(n) \Rightarrow 2^2 T(n/2^2) + 4n + 4n$ $\rightarrow T(n/2^2) \Rightarrow 2T(n/2^2/2) + 4(n/2^2)$

$\boxed{T(n/2^2) \Rightarrow 2T(n/2^3) + 4(n/2^2)}$

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

$T(n) = 2^3 T(n/2^3) + 4n + 4n + 4n$ $\rightarrow T(n/2^3) \Rightarrow 2T(n/2^3/2) + 4(n/2^3)$

$\boxed{T(n/2^3) = 2T(n/2^4) + 4(n/2^3)}$

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

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

⋮

$$\boxed{T(n) = 2^i T(n/2^i) + i(4n)}$$

Stop this procedure when $T(1) = 4$

$\therefore \boxed{\frac{n}{2^i} = 1 \rightarrow n = 2^i \Rightarrow \log_2 n = i}$ // apply this in $T(n)$.

$\boxed{2^{\log_2 n} \rightarrow n^{\log_2 2} = n}$

$\therefore T(n) = 2^{\log_2 n} T(n/2^{\log_2 n}) + \log_2 n\,(4n).$

$T(n) = 2^{\log_2 n} T(n/n) + \log_2 n\,(4n)$

$T(n) = n\,T(1) + 4n \cdot \log_2 n$

$T(n) = n \cdot 4 + 4n \log_2 n$

$$\boxed{T(n) = 4n + 4n \log_2 n}$$

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

$$\boxed{T(n) = 2T(n/2) + n}$$

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

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

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

$T(n) = 2\left[2T(n/2^2) + n/2\right] + n$

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

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

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

$\boxed{T(n/2^2) = 2T(n/2^3) + n/2^2}$

$= 2^2\left[2T(n/2^3) + n/2^2\right] + 2n + n$

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

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

$\boxed{T(n/2^3) = 2T(n/2^4) + n/2^3}$

$\vdots$

$$\boxed{\text{General Form } T(n) = 2^i T(n/2^i) + i(n)}$$

Stop this iteration when $T(n) = 1$

$$\therefore \boxed{\frac{n}{2^i} = 1 \Rightarrow n = 2^i \Rightarrow \log_2 n = i}$$

$\therefore T(n) = 2^{\log_2 n} T\left(n/2^{\log_2 n}\right) + \log_2 n\,(n)$

$\boxed{2^{\log_2 n} \rightarrow n^{\log_2 2} = n}$

$= n \cdot T(n/n) + \log_2 n\,(n)$

$= n \cdot T(1) + \log_2 n\,(n)$

$= n + n\log_2 n$

$$\boxed{T(n) = O(n\log n)}$$

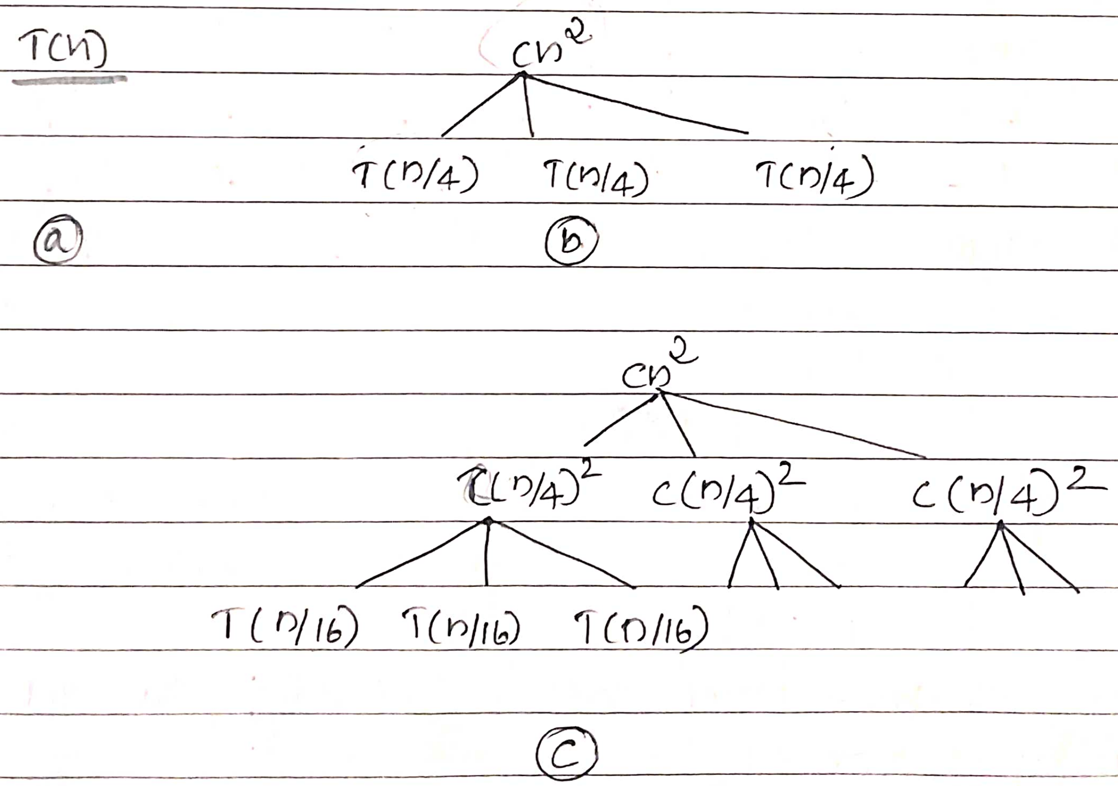
RECURSION TREE METHOD

In a recursion tree, each node represents the cost of a single problem somewhere in the set of recursive function invocations. We sum the costs within each level of the tree to obtain the set of per-level costs, and then we su all the per-level costs to determine the total cost of all t levels of the recursion.

Example 1:

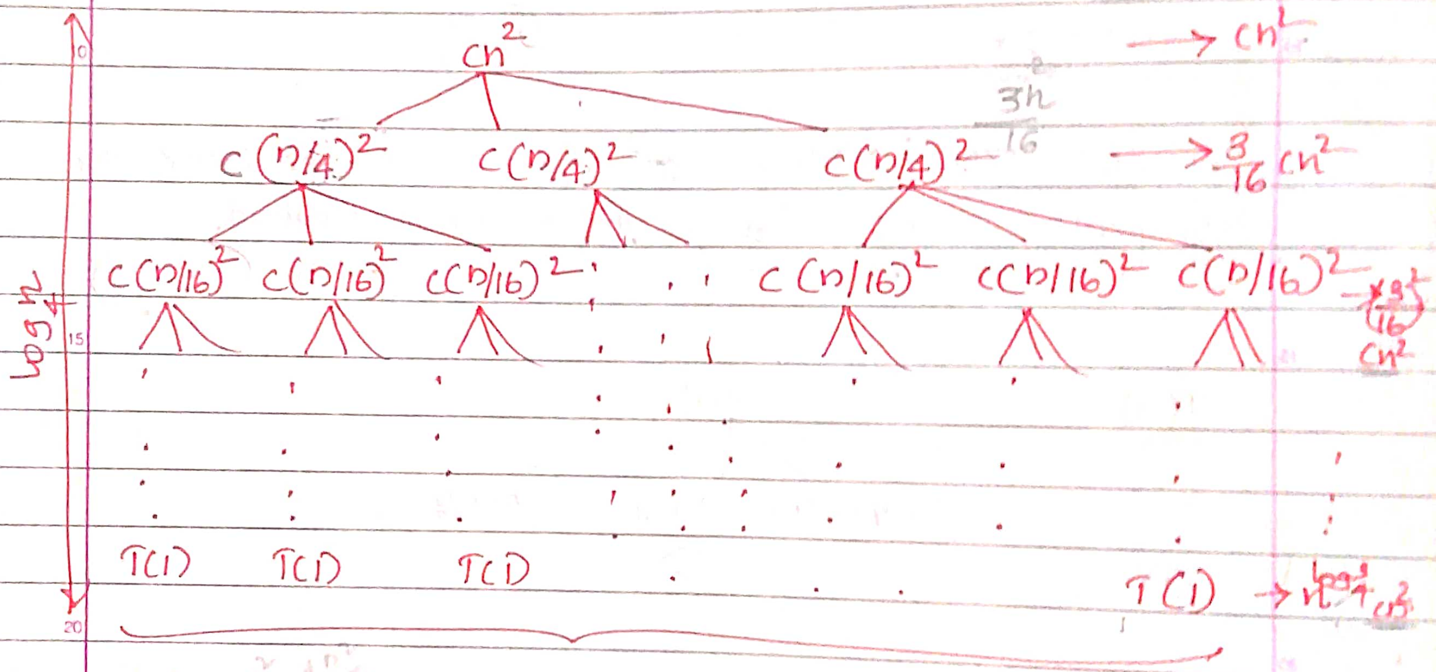
Consider $T(n) = 3T(n/4) + cn^2$. We have to obtain the asymptotic bound using recursion tree method.

Solution:



Part (a) of the figure shows $T(n)$, which we expand in Part (b) into an equivalent tree representing the recurrence. The cn^2 term at the root represents the cost at the top level of recursion, and the 3 subtrees of the root represent the costs incurred by the subproblems of size n/4. Part (c) shows this process carried one step further by expanding each node with cost T(n/4) from part (b). The cost for each of the three children of the root is $c(n/4)^2$.

We continue expanding each node in the tree by breaking it into its constituent parts as determined by the recurrence, until the problem sizes get down to 1, each with a cost of c.



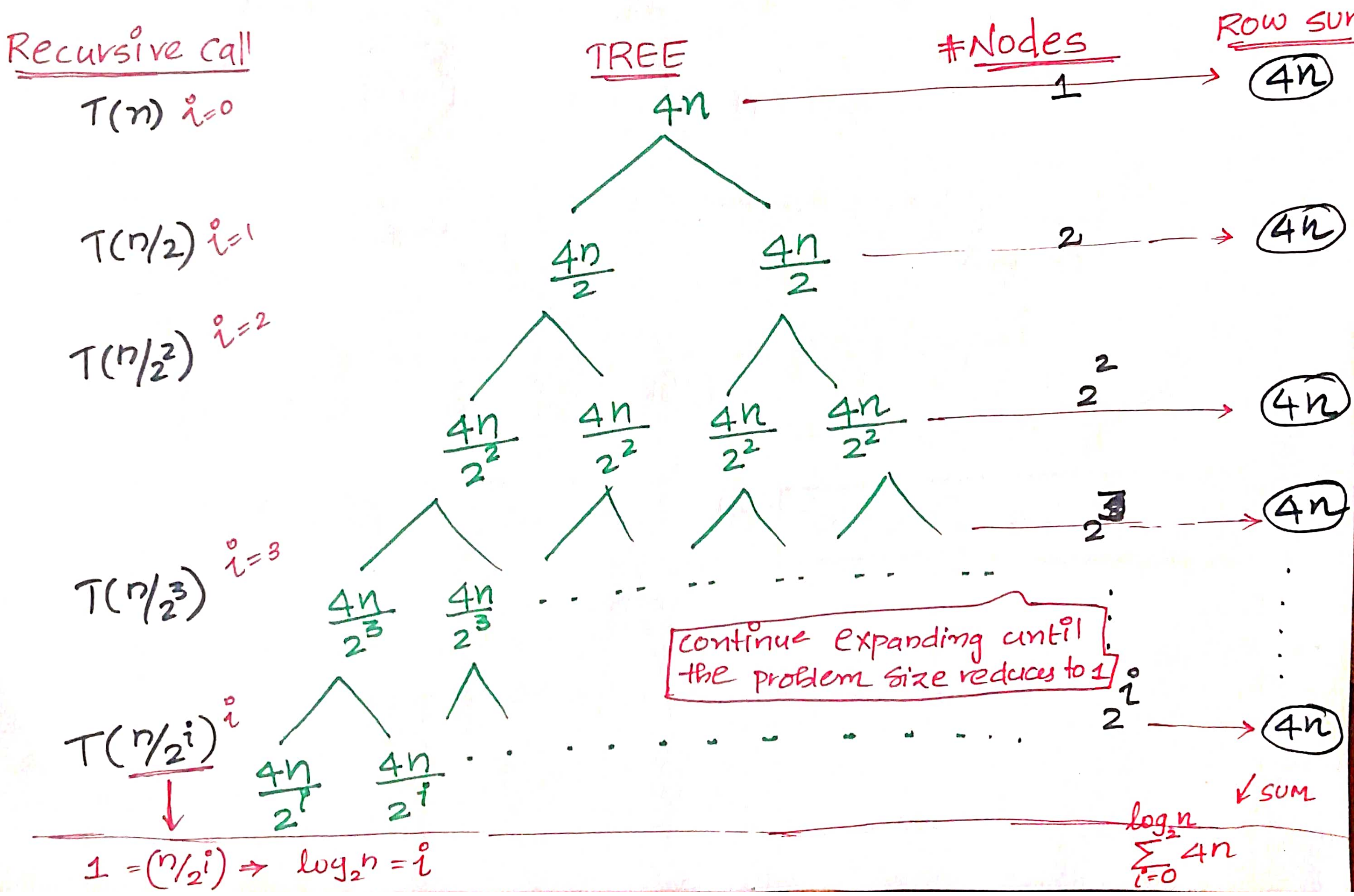
$n^{\log_4 3}$

Total: $O(n^2)$

Add all per-level costs, Finally we get the leading factor as n^2.

So, the complexity is $O(n^2)$

$\boxed{T(1) = 4}$ $\boxed{T(n) = 2T(n/2) + 4n}$



Sum $\rightarrow \sum_{i=0}^{\log_2 n} 4n \Rightarrow 4n \sum_{i=0}^{\log_2 n} 1$

$\underbrace{4n + 4n + \cdots\cdots 4n}_{\log_2 n+1} \Rightarrow 4n(\underbrace{1 + 1 \cdots\cdots + 1}_{\log_2 n+1})$

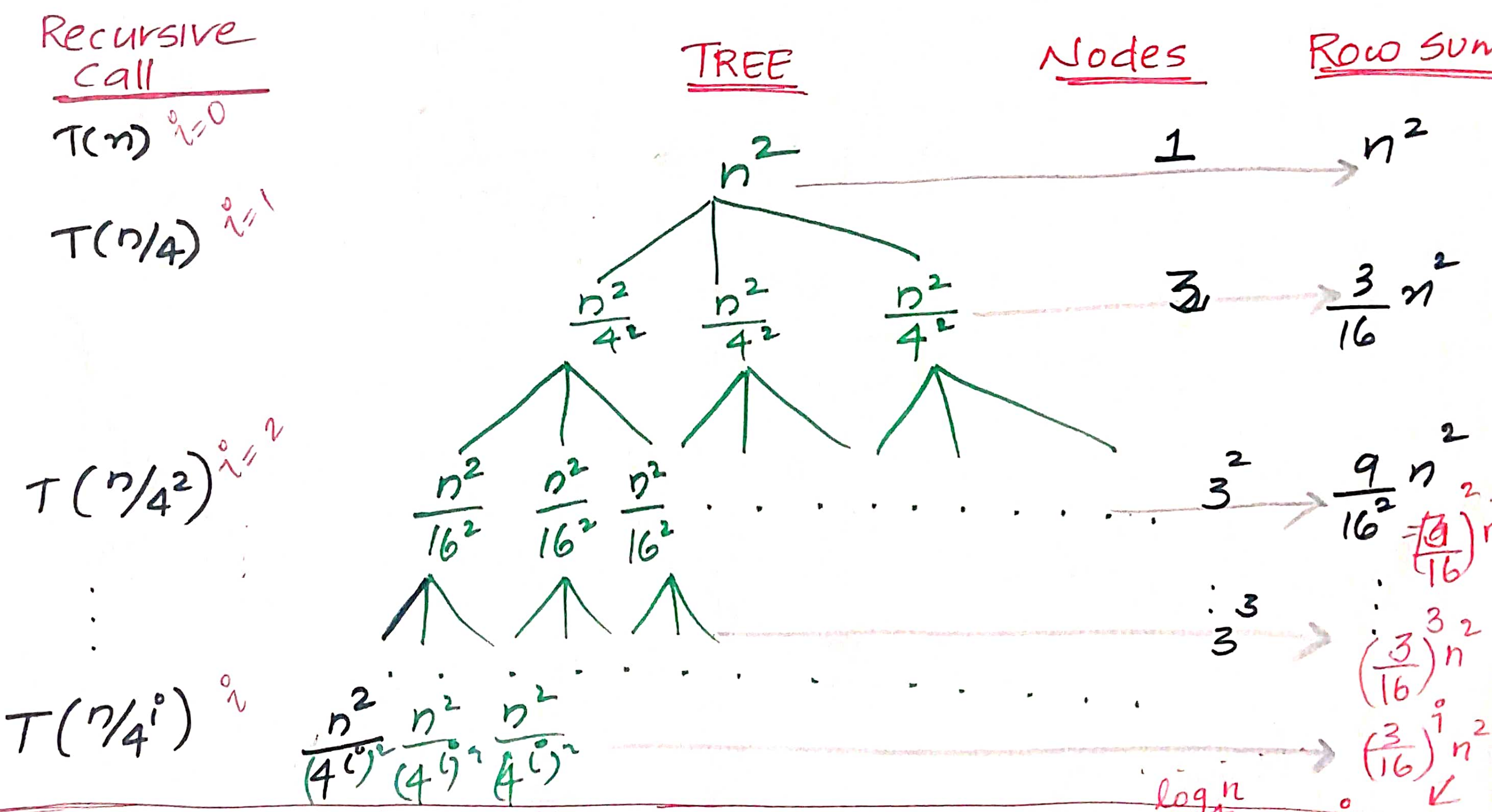
$\Rightarrow 4n(\log_2 n + 1)$

$\Rightarrow 4n \log_2 n + 4n$

$\Rightarrow$ $\boxed{\text{Complexity} \rightarrow O(n \log n)}$

Solve the recurrence equation using Recursion Tree Method.

$$\boxed{T(n) = 3T(n/4) + cn^2}$$



$\frac{n}{4^i} = 1$, $n = 4^i$ $\log_4 n = i$

$$\sum_{i=0}^{\log_4 n} \left(\frac{3}{16}\right)^i n^2$$

$$\Rightarrow n^2 \cdot \sum_{i=0}^{\log_4 n} \left[\left(\frac{3}{16}\right)^0 + \left(\frac{3}{16}\right)^1 + \left(\frac{3}{16}\right)^2 + \cdots \left(\frac{3}{16}\right)^i\right]$$

$$n^2 \cdot \left[\frac{1}{1-\left(\frac{3}{16}\right)}\right] \Rightarrow n^2\left(\frac{1}{\frac{16-3}{16}}\right)$$

$$n^2\left(\frac{16}{13}\right) = \underline{\Theta(n^2)}$$

MODULE II

Master's Theorem - Examples, Asymptotic Notations and their properties - Applications of Asymptotic Notations in Algorithm Analysis - Common complexity functions.

AVL Trees - rotations, Red - Black Trees insertion and deletion. B-Trees - insertion and deletion operations. Sets - Union and find operations on disjoint sets.

1 MASTER'S THEOREM

The master method provides a "cookbook" method for solving recurrences of the form

$$\boxed{T(n) = aT(n/b) + f(n)}$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

The recurrence describes the running time of an algorithm that divides a problem of size 'n' into 'a' subproblems, each of size n/b, where 'a' and 'b' are positive constants. The 'a' subproblems are solved recursively, each in time $T(n/b)$. The function $f(n)$ encompases the cost of dividing the problem and combining the result of the subproblems.

Master Theorem:

The master method depends on the following theorem.

Theorem: Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence.

$T(n) = aT(n/b) + f(n)$.
where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds.

① If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

② If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.

③ If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

In each of the three cases, we compare the function $f(n)$ with the function $n^{\log_b a}$. Intuitively, the larger of the two functions determines the solution to the recurrence.
If, as in case 1, the function $n^{\log_b a}$ is the larger, then the solution is $T(n) = \Theta(n^{\log_b a})$.
If, as in case 3, the function $f(n)$ is the larger, then the solution is $T(n) = \Theta(f(n))$.
If, as in case 2, the two functions are the same size, we multiply by a logarithmic factor, and the solution is $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$.

Example 1 : Use the master method to give tight asymptotic bound of the following recurrences.

$\boxed{T(n) = 9T(n/3) + n}$

A ⇒ The general form of recurrence is

$$T(n) = aT(n/b) + f(n).$$

For this recurrence, we have $a=9, b=3, f(n)=n$, and thus we have that

$$n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$$

$$\boxed{\begin{aligned} &\log_3 9 = \log(9) \div \log(3). \\ &\log(9) = \cdot 9542 \qquad \log(3) = \cdot 477 \\ \therefore\ &\log_3 9 = \cdot 9542/\cdot 477 = 2. \end{aligned}}$$

Since $f(n) = O(n^{\log_3 9 - \epsilon})$, where $\epsilon = 1$, we can apply case 1 of the master theorem (because function $n^{\log_b a}$ is the larger than $f(n)$.) and conclude that the solution is $T(n) = \Theta(n^2)$.

Example 2: $\boxed{T(n) = T(2n/3) + 1}$

⇒ In which $a=1$, $b=3/2$, $f(n)=1$, and $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$.

Case 2 applies, since $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, and thus the solution to the recurrence is $T(n) = \Theta(\lg n)$.

Example 3: $\boxed{T(n) = 3T(n/4) + n\lg n}$

⇒ We have $a=3, b=4, f(n)=n\lg n$. and $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$.

Since $f(n) = \Omega(n^{\log_4 3 + \epsilon})$, where $\epsilon \approx 0.2$, case 3 applies if we can show that the regularity condition holds for $f(n)$. For sufficiently large n, we have that $af(n/b) = 3(n/4)\lg(n/4) \le (3/4)n\lg n = cf(n)$ for $c = 3/4$. Consequently, by case 3, the solution to the recurrence is $T(n) = \Theta(n\lg n)$.

Example 4: $\boxed{T(n) = 2T(n/2) + \Theta(n)}$ //Merge sort

⇒ Here, we have $a=2, b=2, f(n) = \Theta(n)$, and thus we have that $n^{\log_b a} = n^{\log_2 2} = n$. Case 2 applies, since $f(n) = \Theta(n)$, and so we have the solution $T(n) = \Theta(n \lg n)$

Example 5: $\boxed{T(n) = 8T(n/2) + \Theta(n^2)}$ //Matrix Xion.

⇒ Now we have $a=8, b=2$, and $f(n) = \Theta(n^2)$, and so $n^{\log_b a} = n^{\log_2 8} = n^3$. Since n^3 is polynomially larger than $f(n)$, case 1 applies, and

$$T(n) = \Theta(n^3).$$

Exercises

1. $T(n) = 2T(n/4) + 1$
2. $T(n) = 2T(n/4) + \sqrt{n}$
3. $T(n) = 2T(n/4) + n$
4. $T(n) = 2T(n/4) + n^2$.
5. $T(n) = 16T(n/4) + n^3$.
6. $T(n) = 4T(n/3) + n^2$
7. $T(n) = 2T(n/2) + n\lg n$.

Example 6: Can the Master method be applied to solve $T(n) = 4T(n/2) + n^2 \log n$? Why or why not?

⇒ $T(n) = 4T(n/2) + n^2 \log n$
Here, $a=4, b=2, f(n) = n^2 \log n$.
Now $n^{\log_b a} = n^{\log_2 4} = n^2$
Since $f(n) = n^2 \log n$ is asymptotically larger than $n^{\log_b a} = n^2$, so it might seen that case 3 should apply.

The problem is that it is not polynomially larger. The ratio $\frac{f(n)}{n^{\log_b a}} = \frac{n^2 \log n}{n^2} = \log n$, is asymptotically lesser than n^{ϵ} for any positive constant ϵ.
So, the recurrence falls into the gap b/w case 2 and case 3. Thus the master theorem cannot apply for this recurrence.

(2) ASYMPTOTIC NOTATIONS AND THEIR PROPERTIES

Asymptotic means a line that tends to converge to a curve, which may or may not eventually touch the curve.
Asymptotic notation gives a simple characterization of an algorithms efficiency. They allow the comparison of the performances of various algorithms.

Asymptotic notation is a shorthand way to write down and talk about 'fastest possible' and 'slowest possible' running times for an algorithm, using high and low bounds on speed. These is also referred as 'best case' and 'worst case' scenarios respectively.

Asymptotic Notations:

(1) Big-Oh Notation: Worst-case

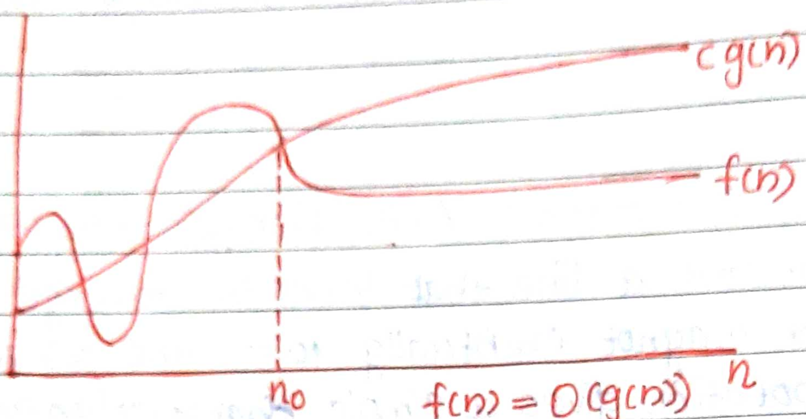
Big-Oh is the formal method of expressing the upper bound of an algorithms running time. It is the measure of the longest amount of time it could possibly take for the algorithm to complete.
More formally, for non-negative functions, $f(n)$ and $g(n)$, if there exists an integer n_0 and a constant $c > 0$ such that for all integers $n > n_0$.

$$f(n) \le c\,g(n)$$

Then f(n) is Big-oh of g(n). This is denoted as

$f(n) \in O(g(n))$

ie., the set of functions which as 'n' gets large, grow no faster than a constant times f(n).



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

Big-Oh Notation —— Asymptotically Upper bound.

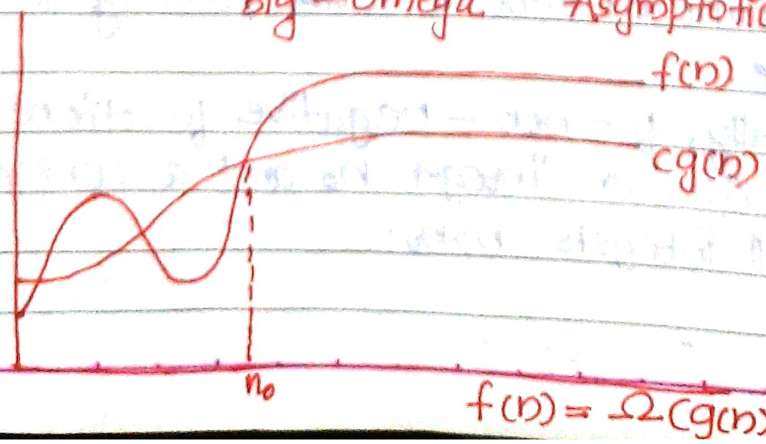
② Big-Omega Notation: Best case

For non-negative functions, f(n) and g(n), if there exist an integer n_0 and a constant $c>0$ such that for all integers $n>n_0$, $f(n) \geq cg(n)$ then f(n) is big omega of g(n).

$f(n) \in \Omega(g(n))$

This is almost the same definition as Big-oh except that $f(n) \geq cg(n)$, this makes g(n) a lower bound function instead of an upper bound function. It describes the best that can happen for a given data size.

Big-Omega Asymptotically lower bound



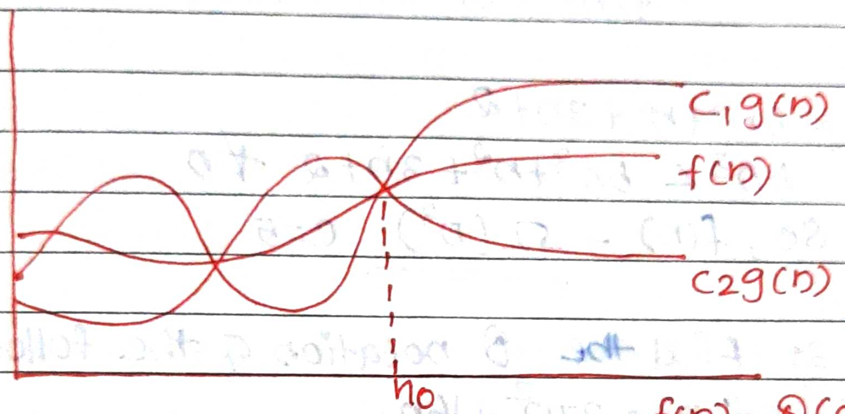
$f(n) = \Omega(g(n))$

Camlin Page
Date / /

③ Theta Notation: Average case

The lower and upper bound for the function 'f' is provided by the theta notation (θ). For non-negative functions f(n) and g(n), if there exist an integer n_0 and positive constant c_1 and c_2 ie; $c_1 > 0$ and $c_2 > 0$ such that for all integers $n > n_0$.

$c_1 g(n) \le f(n) \le c_2 g(n)$ then f(n) is theta of g(n).
This is denoted as $f(n) \in \theta(g(n))$.



Big-theta Notation — Asymptotically tight bound.

PROBLEMS

Example 1: Find the O-notation for the following functions:
ⓐ $5n^3 + n^2 + 6n + 2 = f(n)$
ⓑ $f(n) = 6n^2 + 3n + 2^n$

$\boxed{f(n) \le g(n)}$ $\boxed{(n > n_0)}$.

⇒

ⓐ $f(n) = 5n^3 + n^2 + 6n + 2$. $n_0 = 2$.

for $n \ge 2$

$5n^3 + n^2 + 6n + 2 \le 5n^3 + n^2 + 6n + n \le 5n^3 + n^2 + 7n$.

for $n^2 \ge 7n$

$5n^3 + n^2 + 7n \le 5n^3 + n^2 + n^2 \le 5n^3 + 2n^2$

for $n^3 \ge 2n^2$

$5n^3 + n^3 \le 6n^3$

Thus $c = 6$, $n_0 = 2$. So $f(n) = O(n^3)$

$\boxed{f(n) \le cg(n).}$

(5) $f(n) = 2^n + 6n^2 + 3n$

for $n^2 \ge 3n$

$$2^n + 6n^2 + 3n \le 2^n + 6n^2 + n^2 \le 2^n + 7n^2$$

for $2^n \ge n^2$

$$2^n + 7n^2 \le 2^n + 7 \cdot 2^n \le 8 \cdot 2^n$$

So, $n_0 = 4$, $c = 8$ Thus $f(n) = O(2^n)$.

Example 2: Find the Ω notation for the following equation.

$$f(n) = 5n^3 + n^2 + 3n + 2$$

$\Rightarrow$

$f(n) = 5n^3 + n^2 + 3n + 2$

$5n^3 < 5n^3 + n^2 + 3n + 2 \quad \forall n$

So, $f(n) = \Omega(n^3)$ $c = 5$

$\boxed{cg(n) \le f(n)}$

Example 3: Find the θ notation of the following function

$$f(n) = 27n^2 + 16n.$$

$\Rightarrow$

$f(n) = 27n^2 + 16n.$

Now, first find lower bound for $f(n)$

$\therefore 27n^2 \le 27n^2 + 16n \quad \forall n$

So, $c_1 = 27$. $g(n) = n^2$

Now, we find upper bound for $f(n)$

$27n^2 + 16n \le 27n^2 + n^2$ for $n^2 \ge 16n$ / $n \ge 4$.

$\le 28n^2$

$c_2 = 28$, $g(n) = n^2$

$\therefore f(n) = \theta(n^2)$ $c_1 = 27$, $c_2 = 28$, $n_0 = 4$

Exercises

① Find O-notation (a) $f(n) = 6n^2+3n+2$ (b) $4n^3+2n+3$

② Find Ω notation (a) 3^n+6n^2+3n (b) $6n^3+n^2+4n+3$

③ Find Θ notation (a) $15n^2+22n$ (b) $3 * 2^n + 4n^2+5n+2$

Example 4: PT $\sum_{i=1}^{n} \log(i)$ is $\Theta(n \log n)$.

⇒

$$\text{Let } f(n) = \sum_{i=1}^{n} \log i = \log 1 + \log 2 + \log 3 + \cdots + \log n$$

$$= 0 + \log 2 + \log 3 + \cdots + \log n$$

$$= \log 2 \cdot 3 \cdot 4 \cdots\cdots n$$

$$= \log (n-1)! \leq \log n!$$

We know $\log n! = \Theta(n \log n)$

$\therefore f(n) = \underline{\Theta(n \log n)}$

Example 5: Show that $\log x = O(x)$.

⇒

$$\lim_{x\to\infty} \frac{\log x}{x} = \lim_{x\to\infty} \frac{1/x}{1} = \lim_{x\to\infty} \frac{1}{x} = 0$$

$\therefore \log x = O(x)$.

Example 6: ST $(\sqrt{2})^{\log n} = O(\sqrt{n})$, where log means $\log_2$.

⇒

$$(\sqrt{2})^{\log n} = n^{\log \sqrt{2}} = n^{\log 2^{1/2}} = n^{\frac{1}{2}\log 2} = n^{1/2} = \sqrt{n}$$

Thus it is clear that $(\sqrt{2})^{\log n} = \underline{O(\sqrt{n})}$

Example 7: if $f(n) = a_m n^m + \cdots + a_1 n + a_0$, then $f(n) = O(n^m)$.

$$\Rightarrow f(n) \leq \sum_{i=0}^{m} |a_i| n^i \leq n^m \sum_{i=0}^{m} |a_i| n^{i-m}$$

$$\leq n^m \sum_{i=0}^{m} |a_i| \qquad \text{for } n \geq 1$$

so, $f(n) = O(n^m)$

$\boxed{\text{Exercises}}$ Show that the following equalities are correct.

(a) $5n^2 - 6n = \Theta(n^2)$

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

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

(d) $\sum_{i=0}^{\infty} i^2 = \Theta(n^3)$

(e) $\sum_{i=0}^{\infty} i^3 = \Theta(n^4)$

(f) $n2^n + 6 * 2^n = \Theta(n2^n)$

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

(h) $6n^3/(\log n + 1) = O(n^3)$

(i) $n^{1.001} + n\log n = \Theta(n^{1.001})$

(j) $10n^3 + 15n^4 + 100n^2 2^n = O(100n^2 2^n)$

(k) $33n^3 + 4n^2 = \Omega(n^2)$

(l) $33n^3 + 4n^2 = \Omega(n^3)$.

(m) Show that ~~the~~ for any real constants a and b, where $b > 0$,

$$(n+a)^b = \Theta(n^b).$$

(n) Is $2^{n+1} = O(2^n)$? Is $2^{2n} = O(2^n)$?

(4) Little-oh Notation: (o)

Asymptotic upper bound provided by O-notation may or may not be asymptotically tight. So o-notation is used to denote an upper bound that is not asymptotically tight.

$$o(g(n)) = \left\{ \begin{array}{l} f(n): \text{for any +ve constant } c>0, \text{ if a} \\ \quad \text{constant } n_0>0 \text{ such that} \\ \quad 0 \le f(n) < cg(n) \ \forall \ n \ge n_0 \end{array} \right.$$

The function f(n) becomes insignificant relative to g(n) as 'n' approaches infinity, ie.,

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

ie.,

$$\boxed{f(n) = o(g(n)) \text{ iff } \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0}$$

whenever f(n) is little-oh of g(n), the ratio b/w f(n) and g(n) is zero.

g(n) is variably bigger than f(n).

Whenever g(n) is variably bigger than f(n), if we cancel them still some part of the g(n) will remain in the denomenator that will leads to zero.

Example 1: $f(n) = n$ $g(n) = n^2$. find the little-oh notation for the following functions.

$\Rightarrow$ g(n) is variably bigger than f(n). ie., $n^2 > n$

if we apply $\lim_{n\to\infty} \frac{n}{n^2} = \frac{1}{n} = \frac{1}{\infty} = 0$

$\therefore n = o(n^2)$ // $f(n) = o(g(n))$.

Example 2: $f(n) = \log n$ $g(n) = n$.

$\Rightarrow \lim_{n\to\infty} \frac{\log n}{n} = \frac{\log \infty}{\infty} = \frac{\infty}{\infty}$ $\frac{\infty}{\infty}, \frac{0}{0} \rightarrow$ Indefinite forms.

if $\frac{\infty}{\infty}$ and $\frac{0}{0}$ appears in the result, then we use L-hospitals rule:

ie., whenever $\frac{f(n)}{g(n)}$ leads to indefinite forms, we use L-hospitals rules.

ie., $\boxed{\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}}$

$\lim_{n\to\infty} \frac{\log n}{n} = \lim_{n\to\infty} \frac{1/n}{1} \Rightarrow \lim_{n\to\infty} 1/n \Rightarrow 1/\infty = 0$

$\therefore \log n = o(n)$

⑤ little-Omega Notations (ω):

~~As~~ As o-notation is to O-notation, we have ω-notation as Ω notation. Little-omega (ω) is used to denote a lower bound that is not asymptotically tight.

$\omega(g(n)) = \{ f(n)$: for any positive constant $c>0$ if a constant $n_0>0$ such that $0 \le cg(n) < f(n)$

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

Camlin Page
Date / /

Example 1: $f(n) = 3^n \quad g(n) = 2^n$

$$\Rightarrow \lim_{n\to\infty} \frac{3^n}{2^n} = \left(\frac{3}{2}\right)^n = (1.5)^n = (1.5)^\infty = 1.5 * 1.5 * \cdots = \infty$$

$\therefore f(n) = \omega(g(n))$

$\therefore 3^n = \omega(2^n)$

Example 2: $f(n) = n^2 \quad g(n) = \log n.$

$$\Rightarrow \lim_{n\to\infty} \frac{n^2}{\log n} = \frac{\infty^2}{\log \infty} = \frac{\infty}{\infty}$$ indefinite form.

Apply L-Hospital's rule.

$$\lim_{n\to\infty} \frac{f'(n)}{g'(n)} = \lim_{n\to\infty} \frac{2n}{1/n} = 2n^2 = 2\infty^2 = \infty$$

$\therefore n^2 = \omega(\log n)$

Asymptotic Properties:

Following are the properties of asymptotic notations.

① Transitivity:

- $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ imply $f(n) = \Theta(h(n))$.
- $f(n) = O(g(n))$ and $g(n) = O(h(n))$ imply $f(n) = O(h(n))$.
- $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ imply $f(n) = \Omega(h(n))$
- $f(n) = o(g(n))$ and $g(n) = o(h(n))$ imply $f(n) = o(h(n))$.
- $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$ imply $f(n) = \omega(h(n))$.

② Reflexivity:

- $f(n) = \Theta(f(n))$
- $f(n) = O(f(n))$.
- $f(n) = \Omega(f(n))$.

Camlin Page
Date / /

(3) Symmetry:

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

(4) Transpose Symmetry:

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

We say that $f(n)$ is asymptotically smaller than $g(n)$ if $f(n) = o(g(n))$, and
$f(n)$ is asymptotically larger than $g(n)$ if $f(n) = \omega(g(n))$.

(3) APPLICATIONS OF ASYMPTOTIC NOTATIONS IN ALGORITHM ANALYSIS

We write $O(1)$ to mean a computing time that is a constant. $O(n)$ is called linear, ie., printing a list of n items to the screen looking at each item once. $O(n^2)$ is called quadratic, ie., taking a list of n items and comparing every item to every other item. $O(n^3)$ is called cubic, and $O(2^n)$ is called exponential.
If an algorithm takes time $O(\log n)$, it is faster, for sufficiently large n, than if it had taken $O(n)$. Similarly, $O(n\log n)$ is better than $O(n^2)$ but not as good as $O(n)$.

Function	Name	Example Algorithms
$O(1)$	constant time	Array lookup
$O(\log n)$	Logarithmic Time	Binary Search
$O(n)$	Linear Time	Searching an unsorted array
$O(n \log n)$	Log linear time	Sorting using a comparison sort
$O(n^2)$	Quadratic Time	Sorting using Bubble sort
$O(n^c)$ for $c>1$	Polynomial Time	Multiplying 2 Matrices
$O(c^n)$ for $c>1$	Exponential Time	Proving a sorting n/w is correct
$O(n!)$	factorial	Solving TSP via brute force search

4 COMMON COMPLEXITY FUNCTIONS

Here is a list of classes of functions that are commonly encountered when analyzing the running time of an algorithm. In each case, 'c' is a constant and 'n' increases without bound. The slower-growing functions are generally listed first.

NB: Write the ~~same~~ above table.

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4,096	65,536
5	32	160	1,024	32,768	4,294,967,296.

⑤ ASYMPTOTIC COMPLEXITY OF SIMPLE ALGORITHMS

Example 1: Sum (a,n).

Statement	s/e	frequency	total steps
1. Algorithm Sum(a,n)	0	—	$\theta(0)$
2. {	0	—	$\theta(0)$
3 s:=0.0;	1	1	$\theta(1)$
4 for i:=1 to n do	1	n+1	$\theta(n)$
5 s:=s+a[i];	1	n	$\theta(n)$
6 return	1	1	$\theta(1)$
7 }	0	—	$\theta(0)$
Total			$\theta(n)$.

Example 2: RSum (a,n)

Statement	s/e	frequency n=0	frequency n>0	total steps n=0	total steps n>0
1 Algorithm RSum(a,n)	0	—	—	0	$\theta(0)$
2 {	0	—	—	0	$\theta(0)$
3 if (n≤0) then	1	1	1	1	$\theta(1)$
4 return 0.0;	1	1	0	1	$\theta(1)$
5 else return					
6 RSum(a,n-1)+a[n];	1+x	0	1	0	$\theta(1+x)$
7 }	0	—	—	0	$\theta(0)$
Total				2	$\theta(1+x)$

Example 3 : Matrix Add

Statement	s/e	frequency	total steps
1 Algorithm Add (a,b,c,m,n)	0	–	$\Theta(0)$
2 {	0	–	$\Theta(0)$
3 for i:=1 to m do	1	$\Theta(m)$	$\Theta(m)$
4 for j:=1 to n do	1	$\Theta(mn)$	$\Theta(mn)$
5 c[i,j]:= a[i,j]+b[i,j];	1	$\Theta(mn)$	$\Theta(mn)$
6 }	0	–	$\Theta(0)$
Total			$\Theta(mn) = \Theta(n^2)$

Example 4 : Matrix Multiplication

Statement	s/e	frequency	Total Steps
1 Algorithm Mult (a,b,c,m,n,p)	0	0	$\Theta(0)$
2 {	0	0	$\Theta(0)$
3 for i:=1 to m do	1	$\Theta(m)$	$\Theta(m)$
4 for j:= 1 to p do	1	$\Theta(mp)$	$\Theta(mp)$
5 ~~for~~ {	0	0	$\Theta(0)$
6 c[i,j]:=0;	1	$\Theta(1)$	$\Theta(1)$
7 for k:=1 to n do	1	$\Theta(mpn)$	$\Theta(mpn)$
8 c[i,j]:= c[i,j] + a[i,k] * b[k,j];	1	$\Theta(mpn)$	$\Theta(mpn)$
9 }	0	0	$\Theta(0)$
10 }	0	0	$\Theta(0)$
Total			$\Theta(mpn) = \Theta(n^3)$

Camlin Page
Date / /

Example 5: For the following program give Big oh analysis of the running time.

①
```
for (i=0;i<n;i++)
A[1] = t;
```

②
```
for (i=0; i<n; i++)
  for (j=i; j<n; j++)
    for (k=j; k<n; k++)
      S++;
```

③
```
for (i=0; i<n*n; i++)
A[i]=i;
```

Solution:

① for (i=0; i<n; i++) $\longrightarrow$ n+1 times
A[i] = t; $\longrightarrow$ n times.

Suppose each statement takes 1 cost.
Hence total cost is

$$t(n) = (n+1)*1 + n*1$$
$$= n+1+n$$
$$= 2n+1$$
$$t(n) = O(n)$$

② for (i=0; i<n; i++) $\rightarrow$ n+1 times
for (j=i; j<n; j++) $\rightarrow$ n*(n+1) times
for (k=j; k<n; k++) $\rightarrow$ n*(n+1)*n times
S++; $\rightarrow$ n*(n+1)*n times

Total cost $= t(n) = (n+1)*1 + n*(n+1)*1 + n*(n+1)*n*1 + n*(n+1)*n*1$

$= n+1 + n^2+n + n^3+n^2+n^3+n^2$

$= 2n^3+3n^2+2n+1$

$= O(n^3)$

$\therefore t(n) = O(n^3).$

③ `for (i=0; i<n*n; i++)` $\rightarrow n^2+1$ times

`A[i] = i;` $\rightarrow n^2$ times

Total cost is

$$t(n) = (n^2+1)*1 + n^2*1$$
$$= n^2+1+n^2$$
$$= 2n^2+1$$
$$= O(n^2)$$

$$\therefore t(n) = O(n^2)$$

Exercises

① Obtain the running time for the following "for" loops:

ⓐ
```
for(i <- 1 to k)
{
    for (i <- 1 to k^2)
    {
        for (i <- 1 to k^3)
        {
            P <- P*P;
        }
    }
}
```

ⓑ
```
for (i <- 2 to k-1)
{
    for (j <- 3 to k-2)
    {
        A <- A+2;
    }
}
```

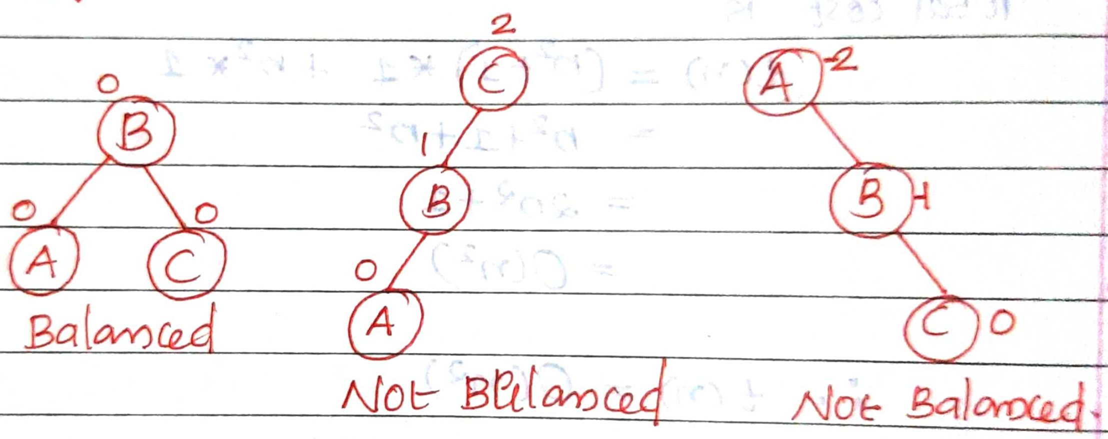
② Arrange the following growth rates in the ~~decre~~ increasing order.

$O(n^4), O(1), O(n^3), O(n), O(n\log n), O(n^2\log n), \Omega(n^{0.5}),$
$\Omega(n^2\log n), \Theta(n^2), \Theta(n^{1.5}), \Theta(n\log n).$

⑥ AVL TREES

Named after their inventor Adelson, Velski & Landis, AVL trees are height balancing binary search tree. AVL tree checks the height of the left and right sub-trees and assures that the difference is not more than 1. This difference is called the Balance Factor.

Example:



To implement an AVL tree each node must contain a balance factor, which indicates its states of balance relative to its sub-trees.
If balance is defined by

$$\boxed{\text{(Height of left subtree)} - \text{(Height of right subtree)}}$$

then the balance factor in balanced tree can only have values -1, 0 or 1. Thus AVL tree may have the following balance factor for a node $\{-1, 0, 1\}$

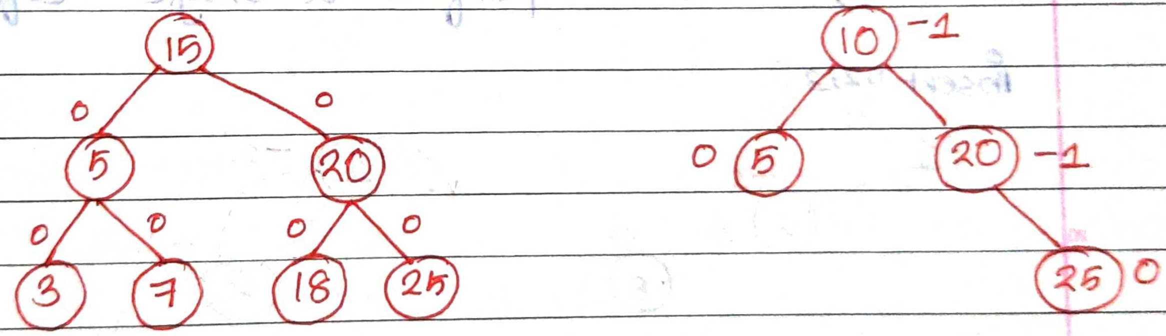
If a node 'x' has balance factor 1, then that means left subtree is at greater hight than the right subtree, then it is said to be left balanced.

If a node 'x' has balance factor −1 then that means right subtree is greater hight than the left subtree then it is said to be right balanced.
For balance factor 0 the tree is said to be (completely) balanced.

If the binary tree is the binary search tree then it is AVL search tree iff,

1. Binary search tree B_L and B_R are AVL search tree where B_L and B_R are left and right subtree respectively.
2. $|h_L - h_R| \leq 1$ where h_L and h_R are the hights of left and right subtrees respectively.

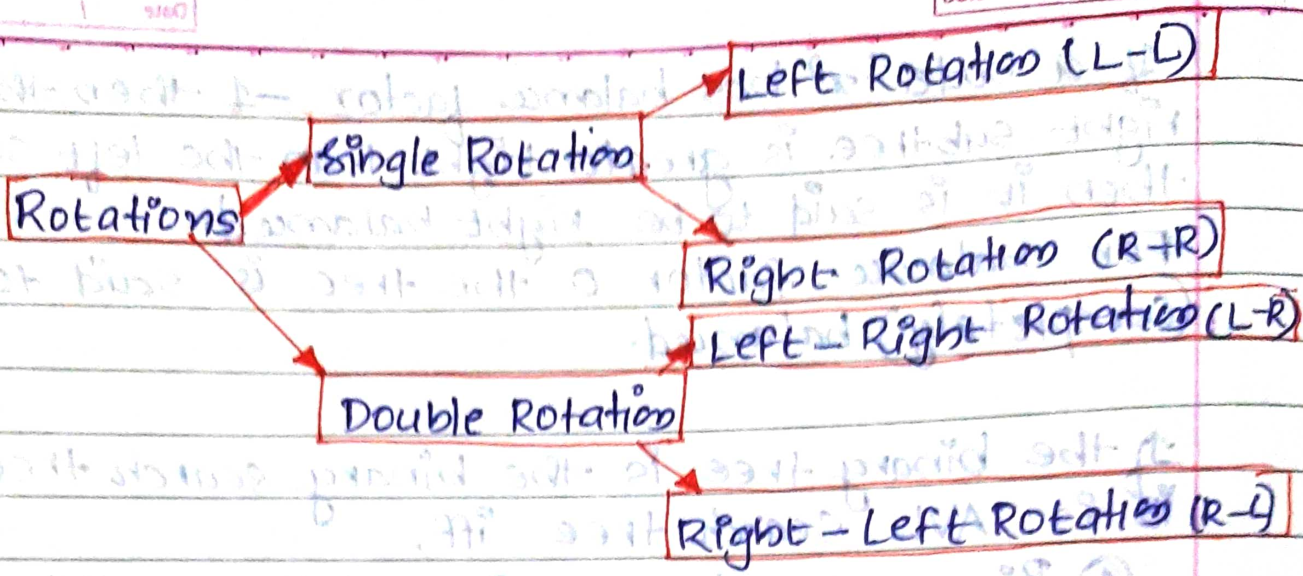
Example:



AVL Rotations:

To balance itself, an AVL tree may perform the 4 kinds of rotations.

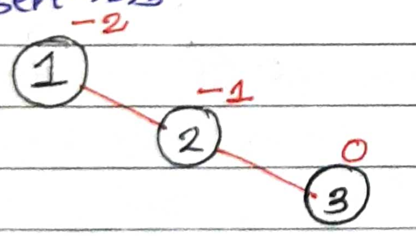
1. Left − Left (L−L) Rotation.
2. Right − Right (R−R) Rotation.
3. Left − Right (L−R) Rotation.
4. Right − Left (R−L) Rotation.



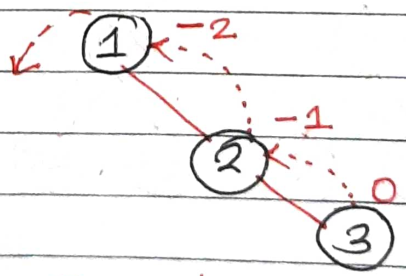
Single - Left Rotation (Left - Left Rotation):

In L-L Rotation every node moves one position to left from the current position.
~~To~~ If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single - Left operation

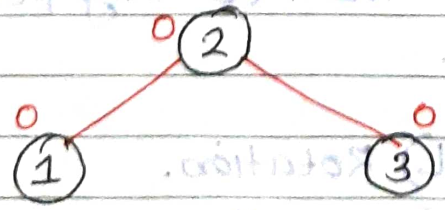
insert 1,2,3



Tree is unbalanced



To make balanced we use LL rotation which moves nodes one position to left.

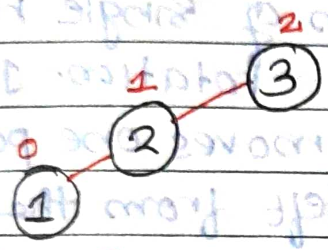


After LL Rotation Tree is balanced.

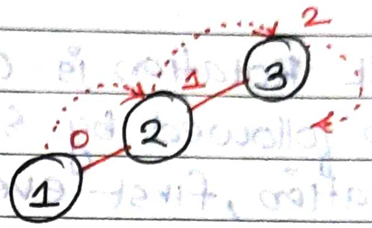
Single – Right Rotation (Right – Right Rotation):

In RR Rotation every node moves one position to right from the current Position. AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then need a right rotation.

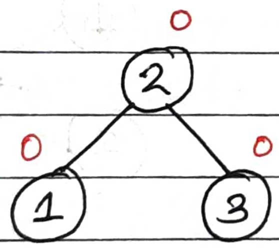
insert 3, 2, 1



Tree is unbalanced



To make balanced we use RR Rotation which moves nodes one position to right

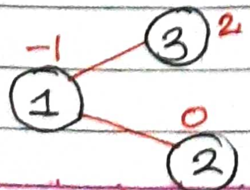


After RR rotation
Tree is balanced.

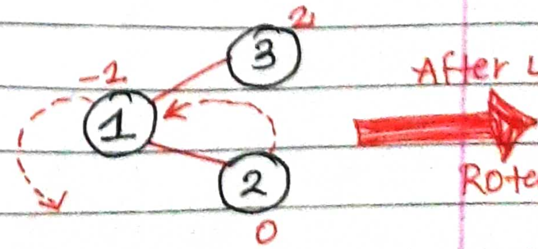
Left – Right Rotation:

The LR rotation is combination of single left rotation followed by single right rotation. In LR ~~oper~~ rotation, first every node moves one position to left then one position to right from the current position.

insert 3, 1 and 2

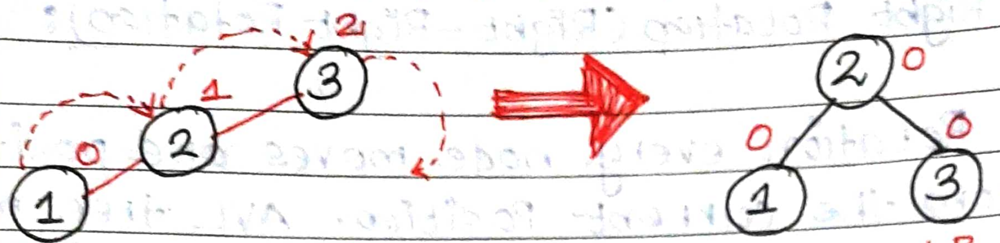


Tree is unbalanced



After LL Rotation

LL Rotation

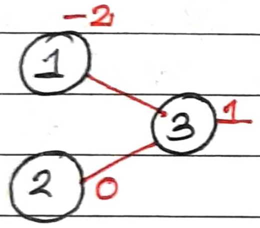


RR Rotation

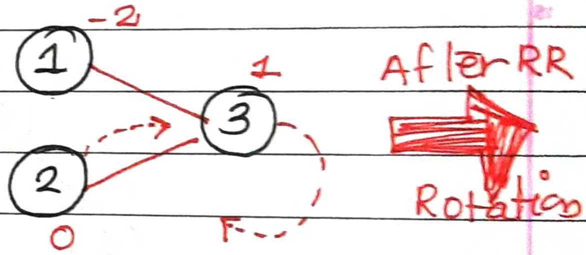
After LR Rotation Tree is balanced.

Right — Left Rotation:

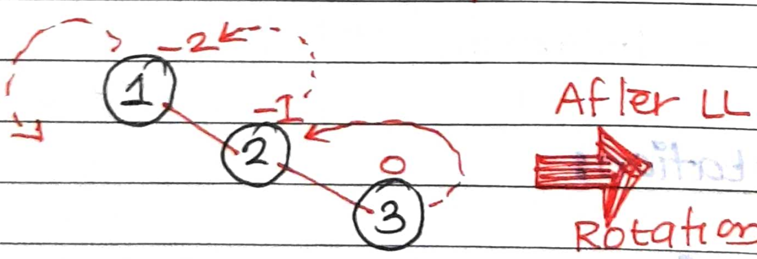
The RL Rotation is combination of single right rotation followed by single left rotation. In RL Rotation, first every node moves one position to right then one position to left from the current position.



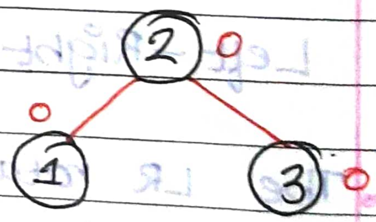
Tree is unbalanced.



RR rotation



LL Rotation



After RL rotation tree is balanced.

Insertion Operation AVL Tree:

In an AVL tree, the insertion operation is performed with O(log n) time complexity. In AVL tree new node is always inserted as a leaf node.

The insertion operation is performed as follows:

Step 1: Insert a new element into the tree using Binary search tree insertion logic

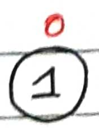
Step 2: After insertion, check the Balance factor of every node.

Step 3: If the balance factor of every node is 0,1 or −1, then go for next operation.

Step 4: Otherwise, the tree is said to be unbalanced. Then perform the suitable Rotation to make it balanced. And go for next operation.

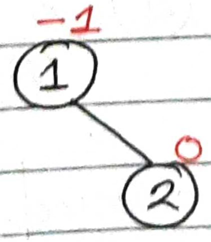
Example: construct an AVL Tree by inserting number from 1 to 8.

Insert 1



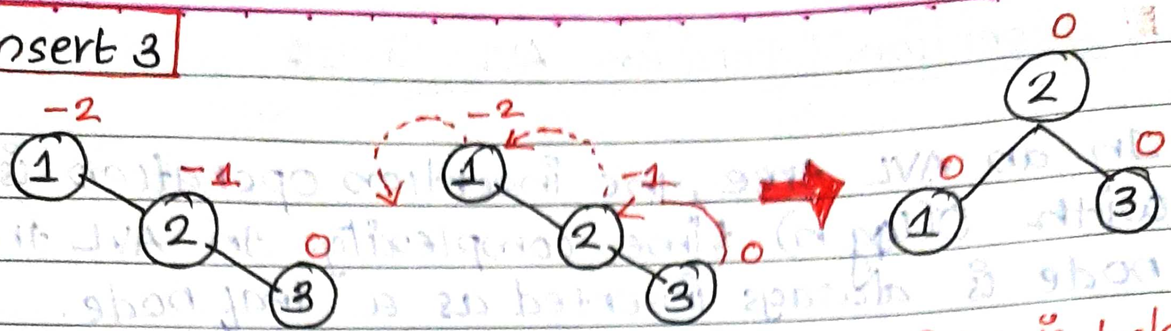
Tree is balanced.

Insert 2



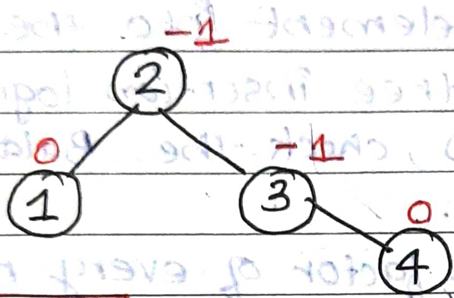
Tree is balanced.

Insert 3



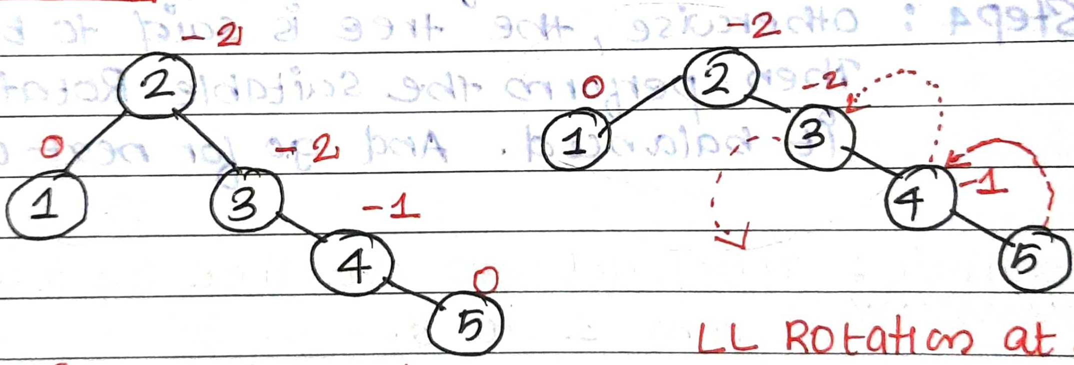
Tree is unbalanced   LL Rotation   Tree is balanced.

Insert 4

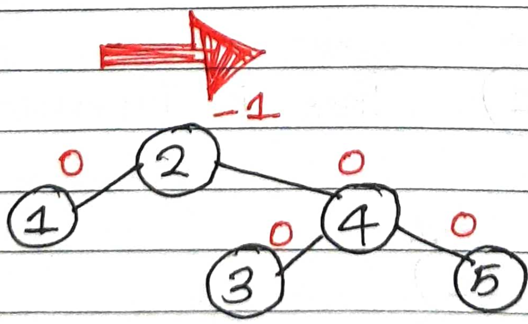


Tree is balanced.

Insert 5



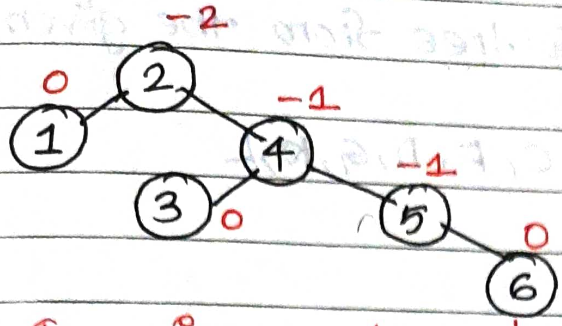
Tree is unbalanced   LL Rotation at 3



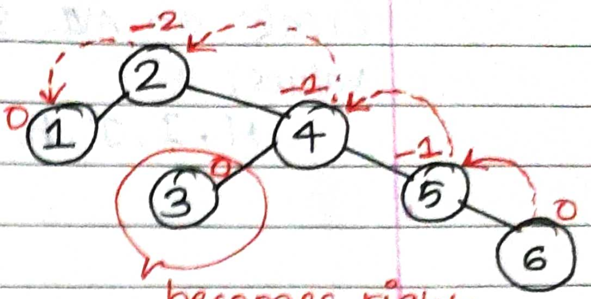
Tree is balanced.

Date / /

Insert 6

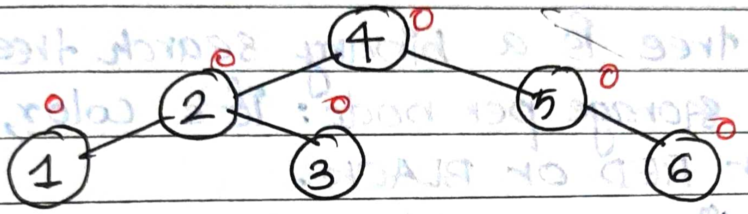


Tree is unbalanced



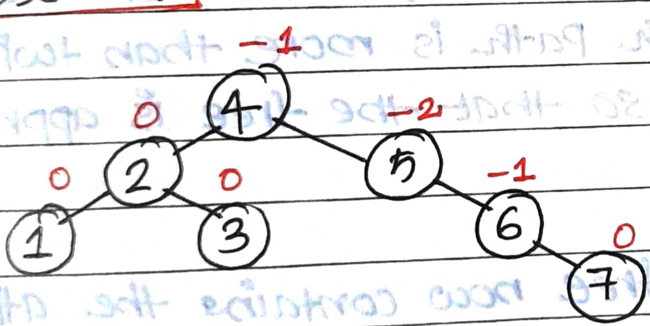
becomes right child of 2

LL Rotation at 2

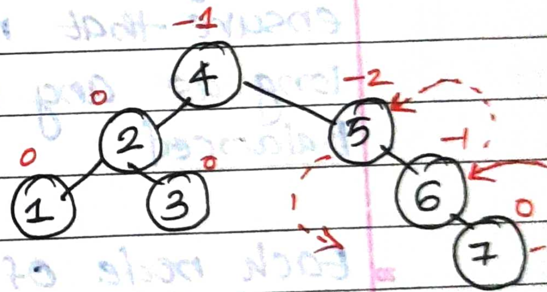


Tree is balanced.

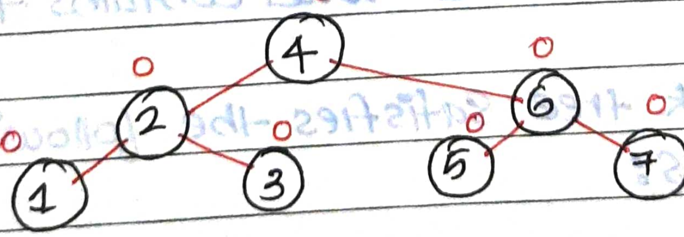
Insert 7



Tree is unbalanced

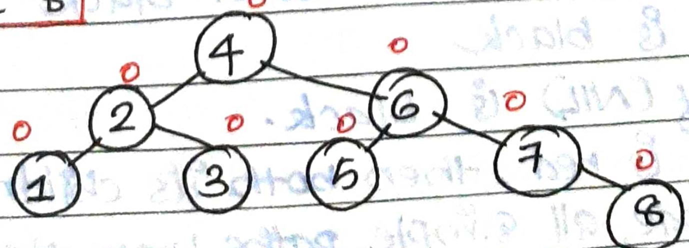


LL Rotation at 5



Tree is balanced.

Insert 8

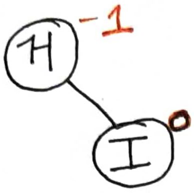


Tree is balanced.

construct an AVL tree having the following elements

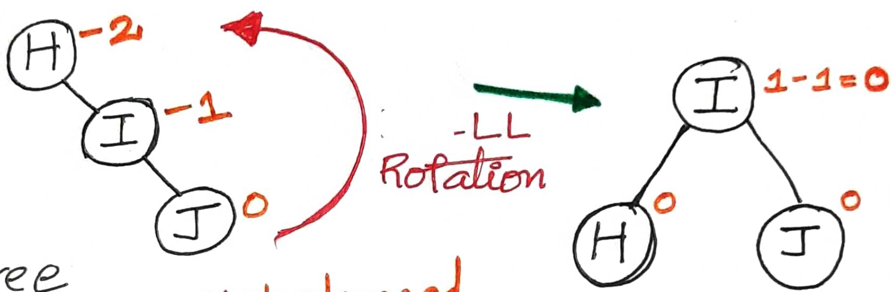
H, I, J, B, A, E, C, F, D

INSERT H and I



Balanced

INSERT J



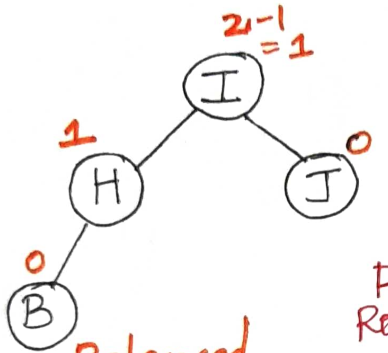
Unbalanced

Balanced

→ Binary Search Tree becomes unbalanced as the BF of $\boxed{H = -2}$.

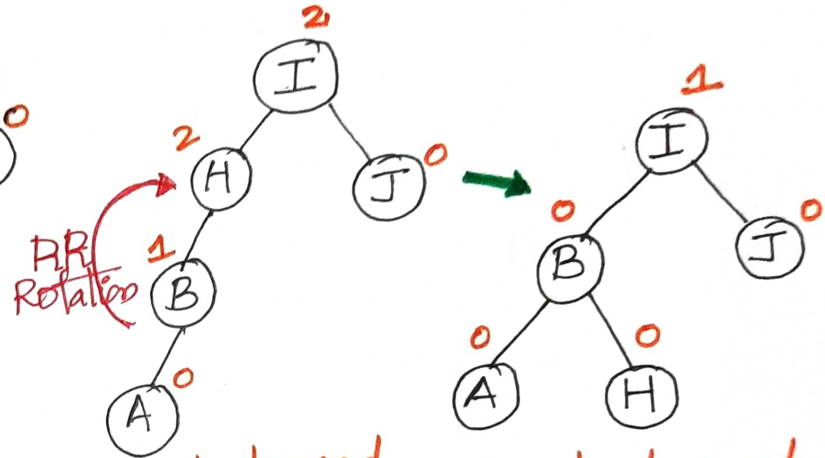
→ Since the BST is Right-skewed, Perform ~~RR~~ LL Rotation on node H.

INSERT B



Balanced

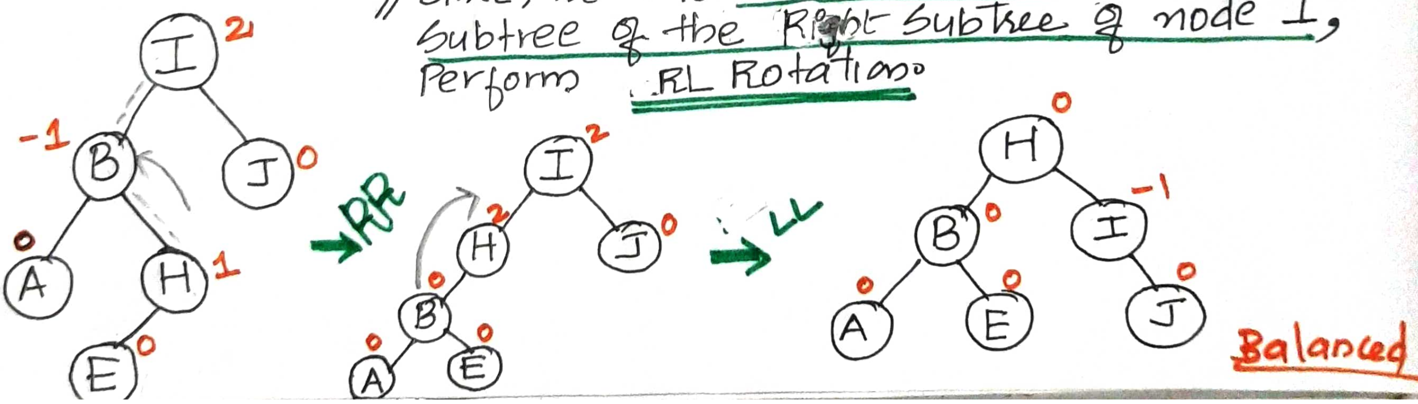
INSERT A



unbalanced

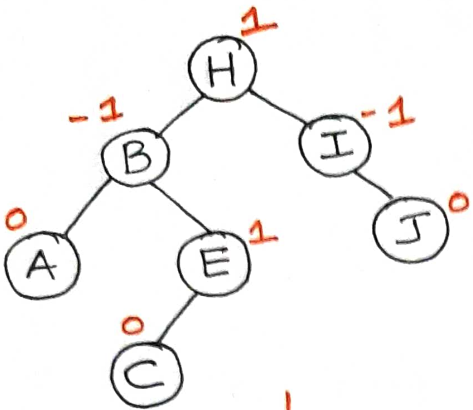
balanced

INSERT E



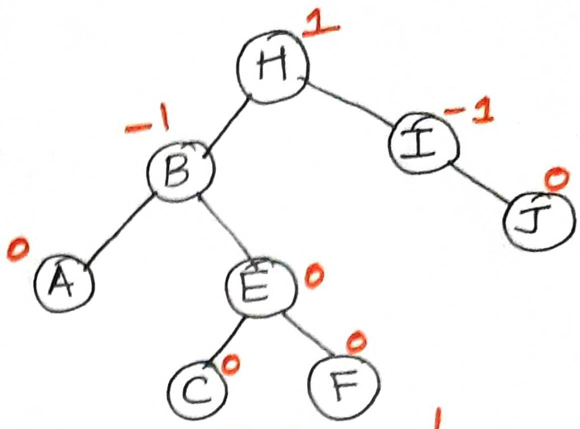
// Since, the node E is inserted into the Left subtree of the Right subtree of node I, Perform RL Rotation.

INSERT C



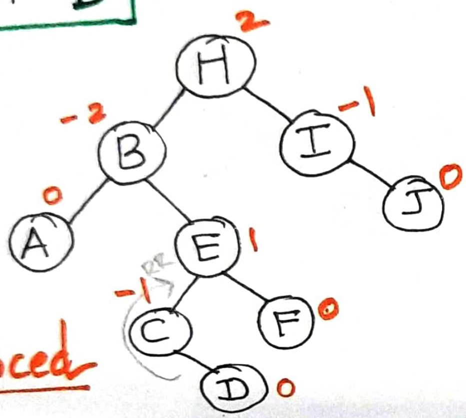
Balanced

INSERT F



Balanced

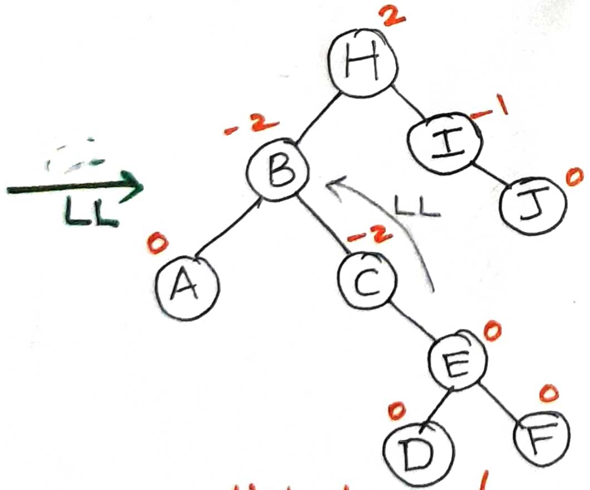
INSERT D



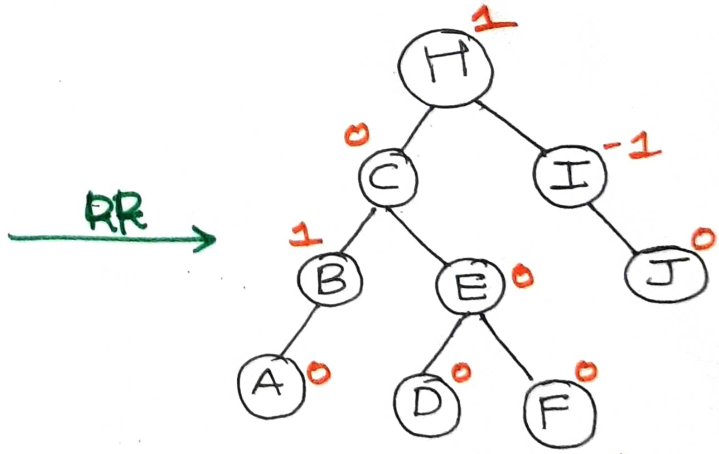
Unbalanced

// Since the node D is inserted into the left subtree of the Right sub tree → use RL Rotation

① LL
② RR



Unbalanced.



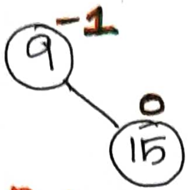
Balanced

Construct an AVL tree for the following elements:

9, 15, 20, 8, 7, 13, 10

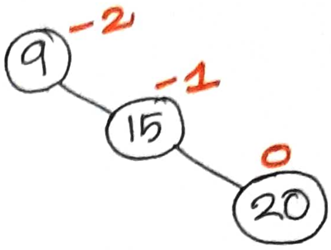
INSERT 9

INSERT 15

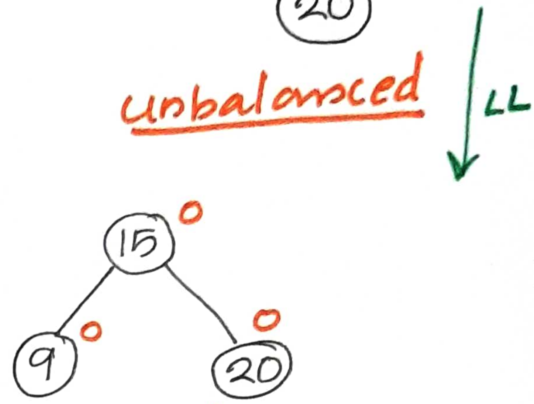


Balanced

INSERT 20

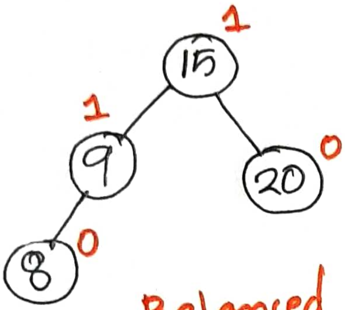


unbalanced



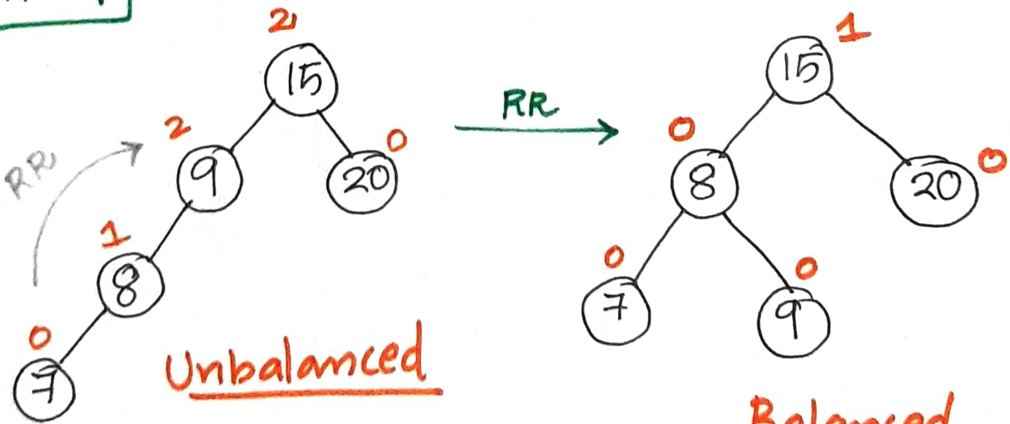
Balanced

INSERT 8



Balanced

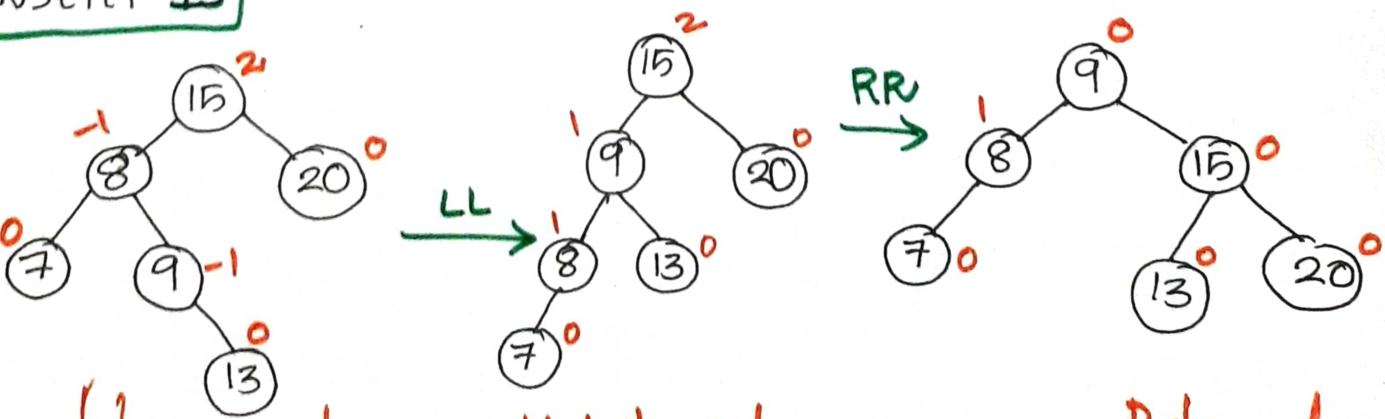
INSERT 7



Unbalanced

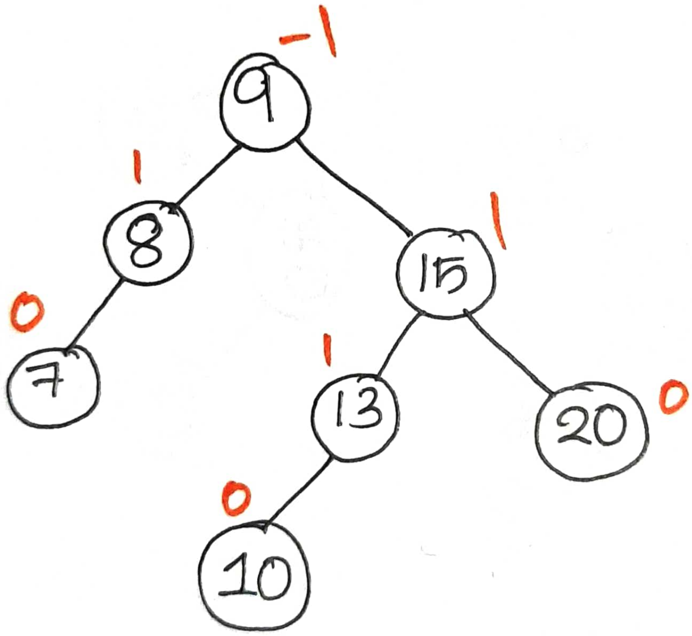
Balanced

INSERT 13



Unbalanced LR Rotation Unbalanced Balanced

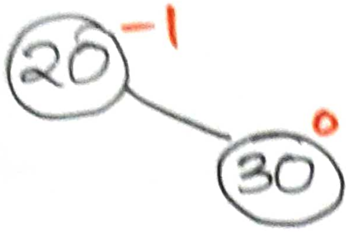
INSERT 10



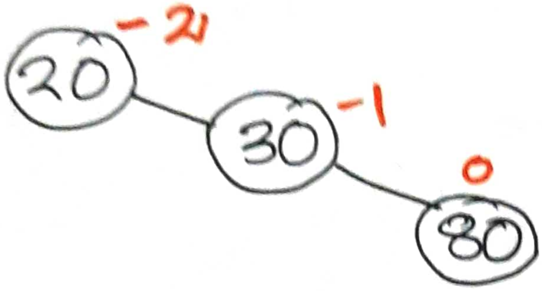
20, 30, 80, 40, 10, 60, 50, 70

1 20

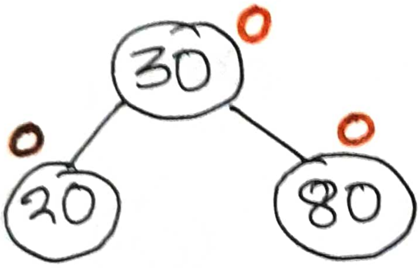
2 30



3 80

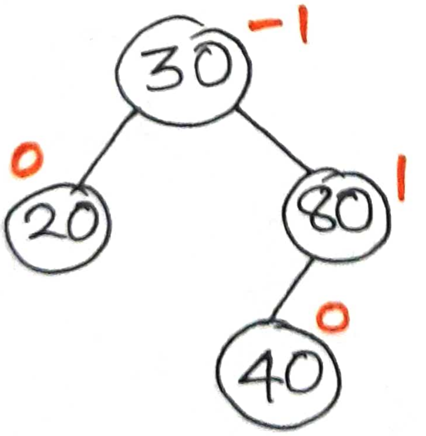


Unbalanced.
LL Rotation.



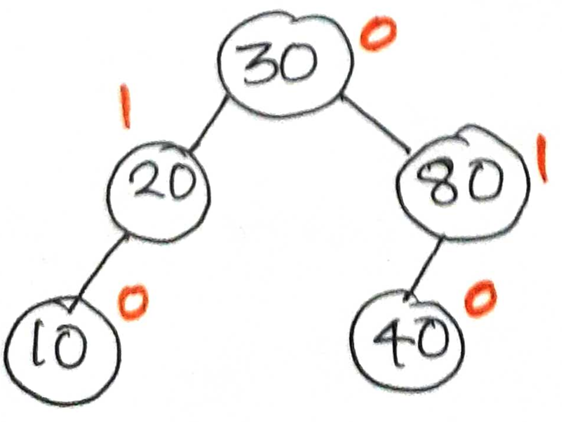
Balanced.

4 40



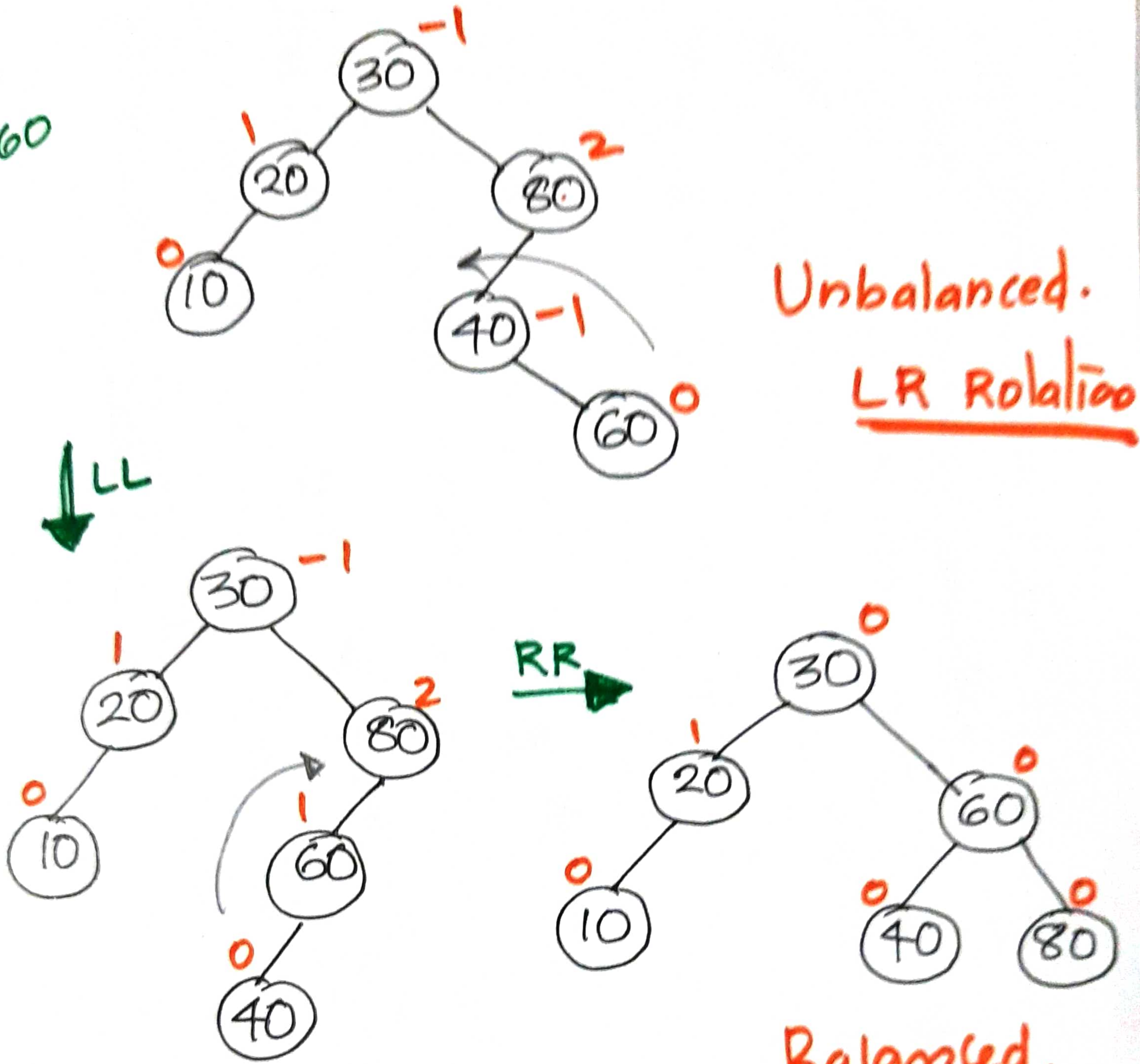
Balanced.

5 10



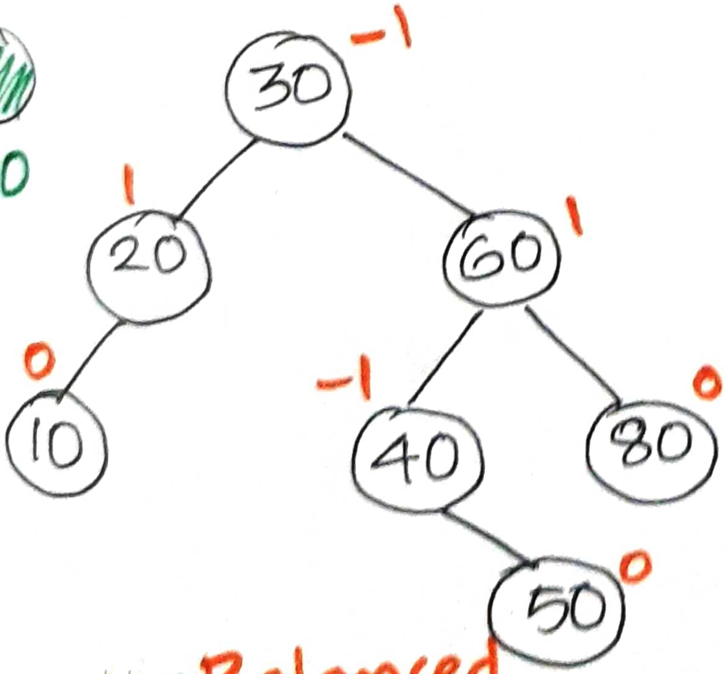
Balanced.

6 60



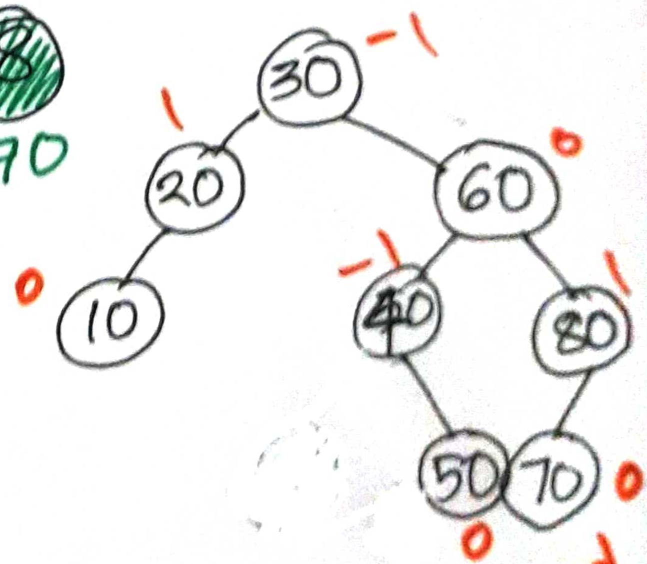
Balanced.

7 50



Balanced

8 70



Balanced

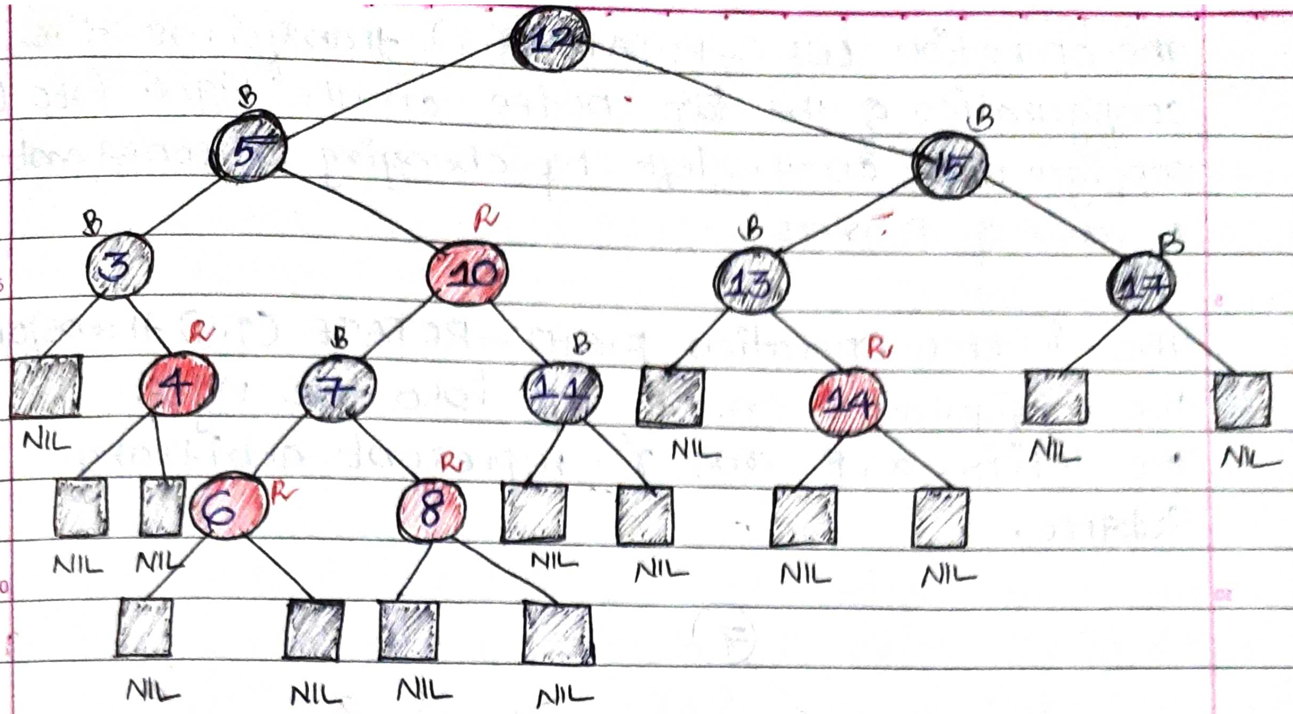
⑦ RED — BLACK TREE

Red-Black tree is a binary search tree with one extra bit of storage per node: it's color, which can be either RED or BLACK.
By constrainting the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately balanced.

Each node of the tree now contains the attributes color, key, left, right, and P. If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL.

A Red-Black tree satisfies the following red-black tree properties:

1. Every node is either red or black
2. The root is black
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.



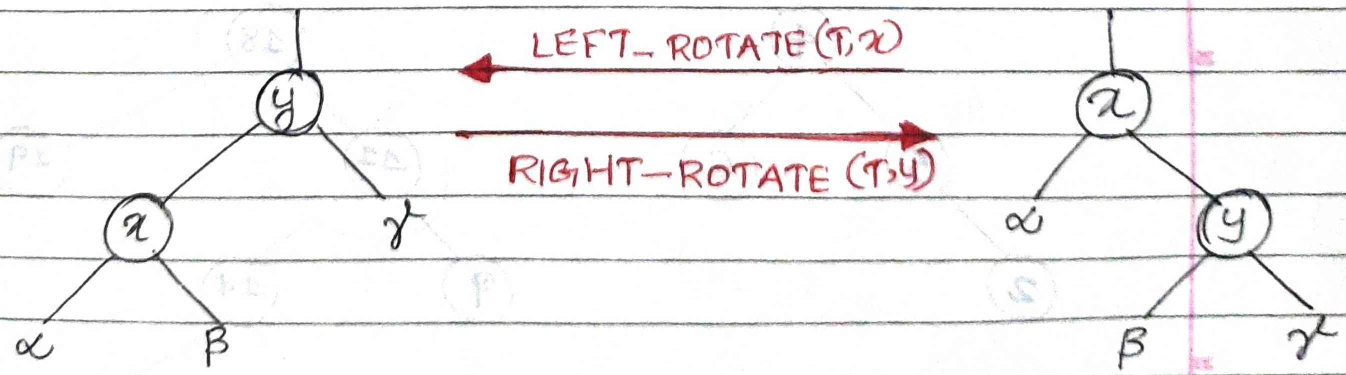
Red-Black Tree.

Rotations:

Rotation is a local operation in a search tree that preserves the binary search tree property.

2 kinds of operations.

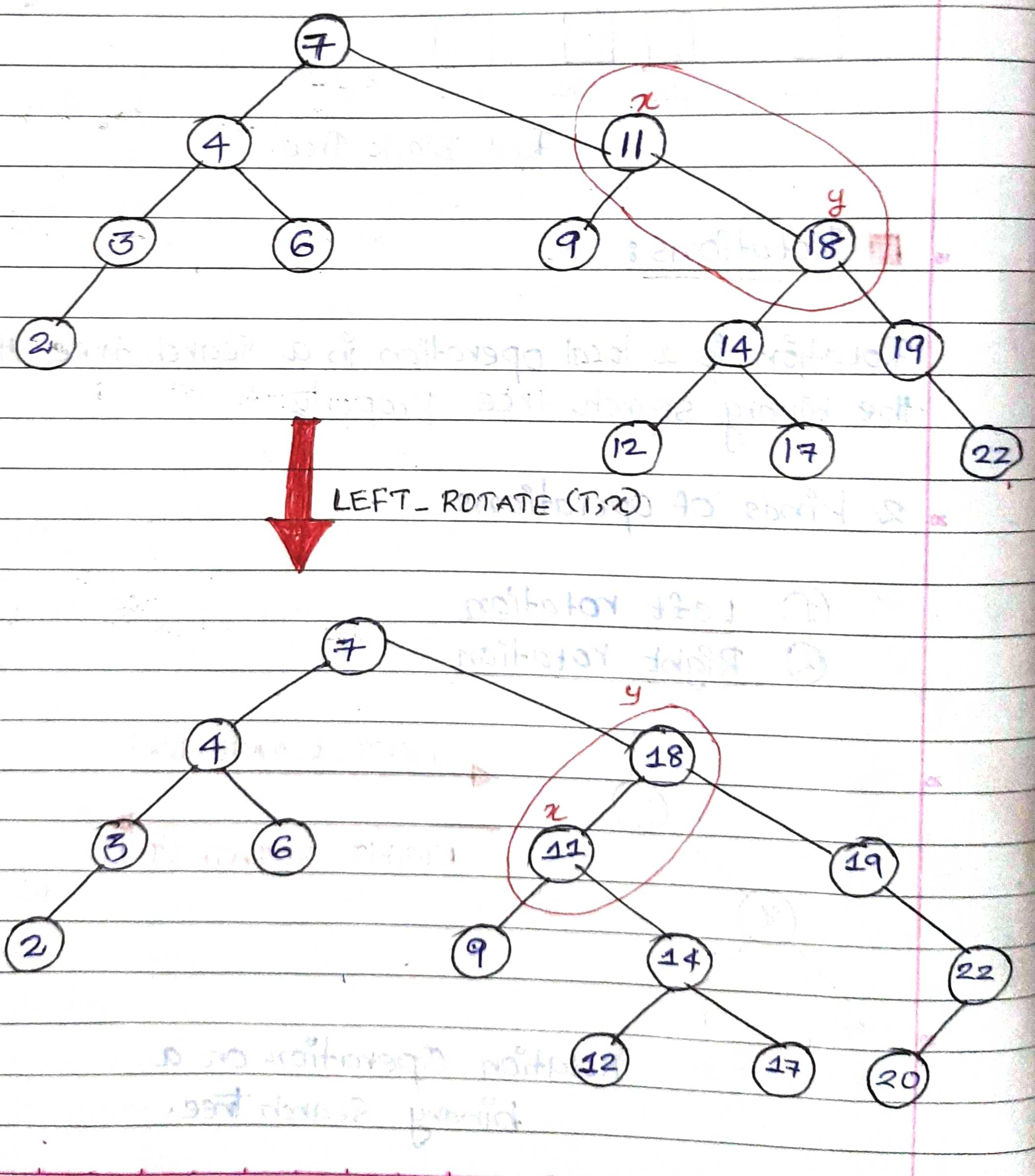
1. Left rotation
2. Right rotation



Rotation Operation on a binary search tree.

The operation LEFT-ROTATE (T, x) transforms the configuration of the two nodes on the right into the configuration on the left by changing a constant number of pointers.

The inverse operation RIGHT-ROTATE (T, y) transforms the configuration on the left into the right. The letters α, β, and γ represent arbitrary subtree.



RED-BLACK TREE INSERTION:

We insert a node into an n-node red-black tree in O(lg n) time. In Red-Black tree, we use two tools to do balancing after insertion caused imbalance.

1. Recoloring.
2. Rotation.

We try recoloring first, if recoloring doesn't work, then we go for rotation. Following is detailed algorithm. The algorithm has mainly two cases depending upon the color of uncle.

→ If uncle is red, we do recoloring.
→ If uncle is black, we do rotations and/or recoloring.

Algorithm:

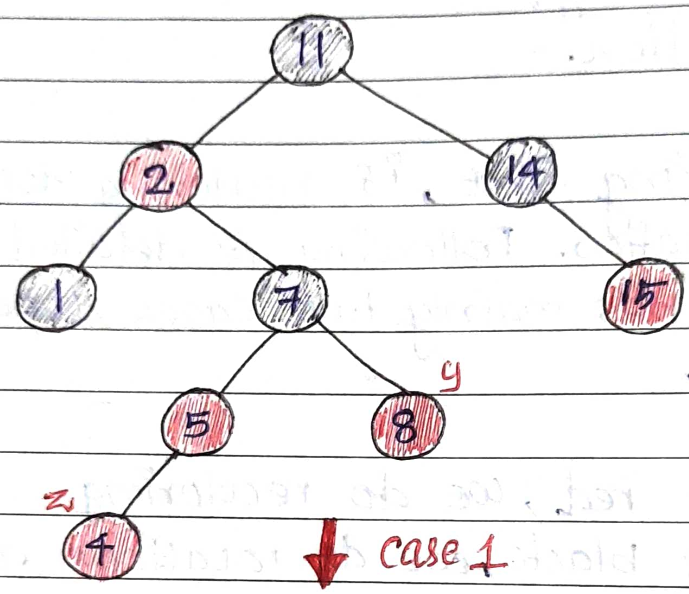
```
RB-INSERT-FIXUP (T, z)
    while z.p.color == RED
        if z.p == z.p.p.left
            y = z.p.p.right
            if y.color == RED
                z.p.color = BLACK
                y.color = BLACK            Case 1
                z.p.p.color = RED
                z = z.p
            else if z == z.p.right
                z = z.p                    Case 2
                LEFT-ROTATE (T, z)
            z.p.color = BLACK
            z.p.p.color = RED              Case 3
            RIGHT-ROTATE (T, z)
```

else (same as then clause with right & left exchanged)

T.root.color = BLACK

Operation of RB – INSERT – FIXUP : Example 1

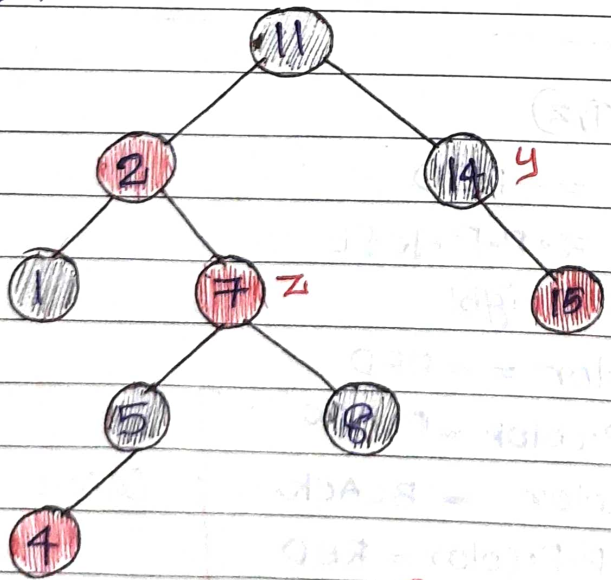
(a) →



A node z after insertion. B/s both z and its Parent z.p are red, a violation of Property occurs. z's uncle y is red. so, case 1 applies.

↓ case 1

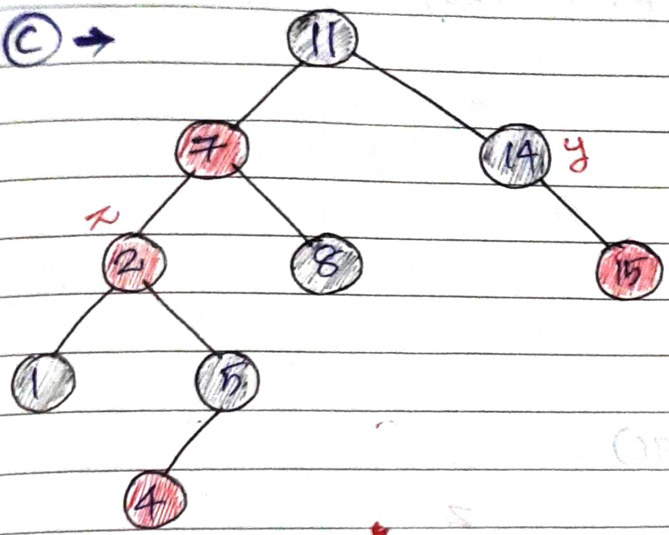
(b) →



z and its Parent are both red, but z's uncle y is black. since z is the right child of z.P, case 2 applies.

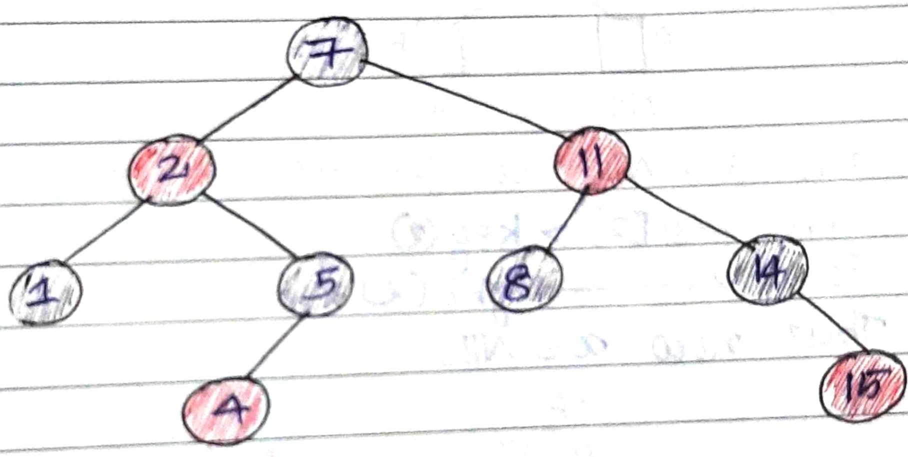
↓ Case 2

(c) →



z is the left child of its parent, and uncle is black so case 3 applies

↓ Case 3



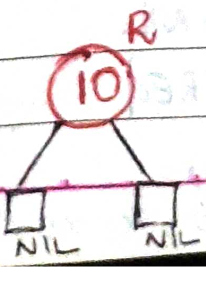
Legal Red-Black tree

Example 2:

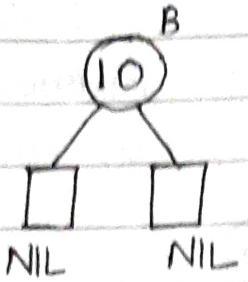
Show that the red-Black tree to be resulted after successively inserting the values 10, 90, 5, 20, 6, 9 into an initially empty red-black tree.

Solution:

Insert 10:

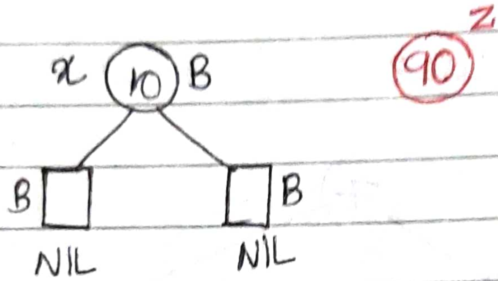


Now call RB-INSERT-FIXUP, we get



$\boxed{\text{Insert } 90}$

Call RB-INSERT $(T, 90)$



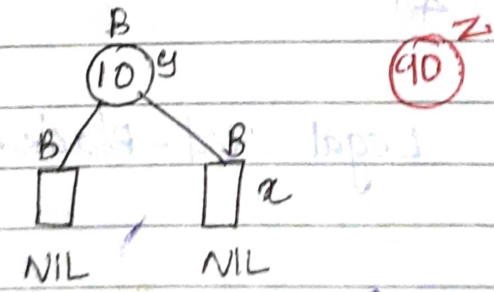
Here $x \neq NIL$ so $y \leftarrow x$

Here $key[z] > key(x)$

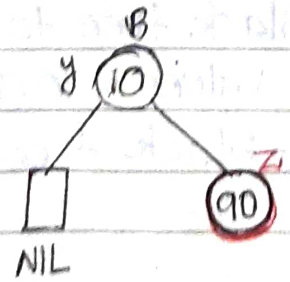
So $x \leftarrow right[x]$

Thus now $x = NIL$

So,



make $p[z] \leftarrow y$, and $right[y] \leftarrow z$

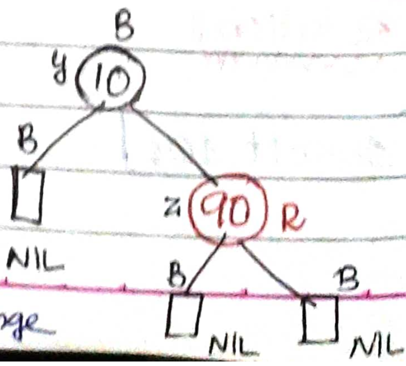


Also make $left[z] = NIL$

$right[z] = NIL$

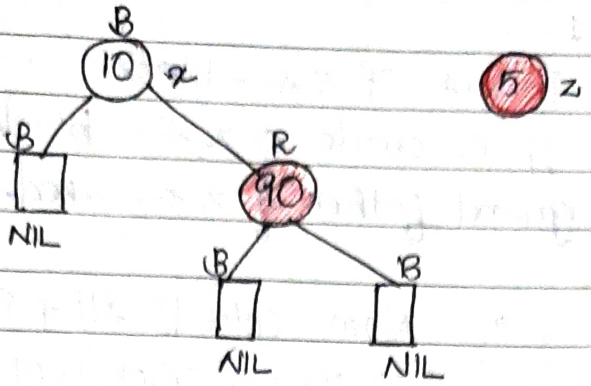
$color[z] = Red$

ie.,

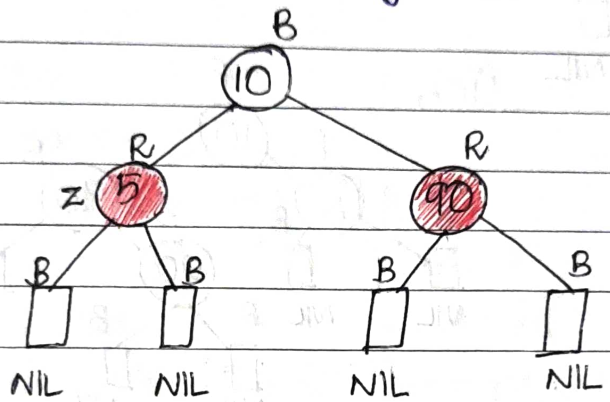


Here $color[p(z)] \neq RED$, so no change

Insert 5



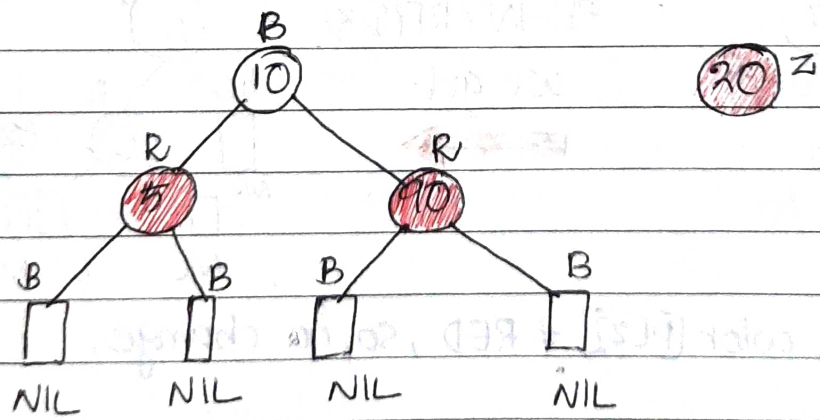
Call RB-INSERT (T, z) we get



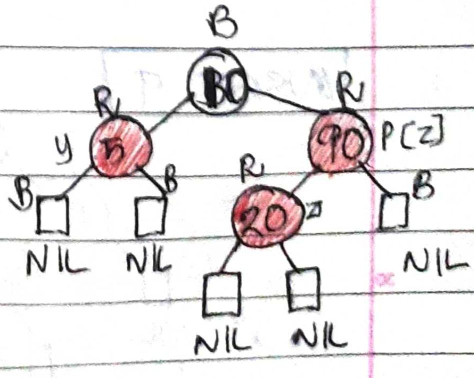
Now call RB-INSERT-FIXUP (T, z)

Here parent of z color is not RED, so no change.

Insert 20



Call RB-INSERT (T, z) →



Now call RB-INSERT-FIXUP (T, z)

Here color [P[z]] = Red

and P[z] = left [P[P[z]]]

So, y ← right [P[P[z]]]

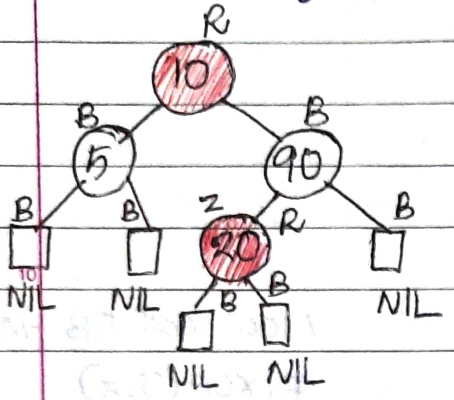
and Color[y] = RED

so, by case 1

color of Parent of z ← black

color of y ie, uncle of z ← black

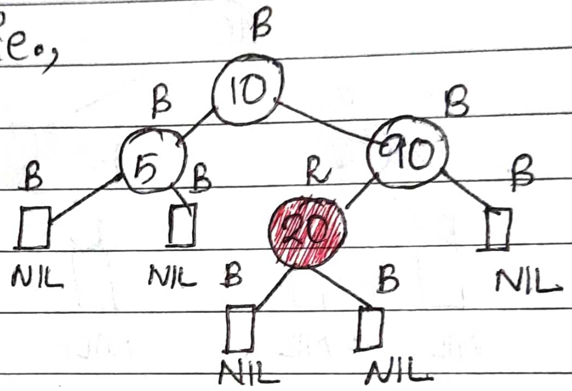
color of grand father of z ← red.



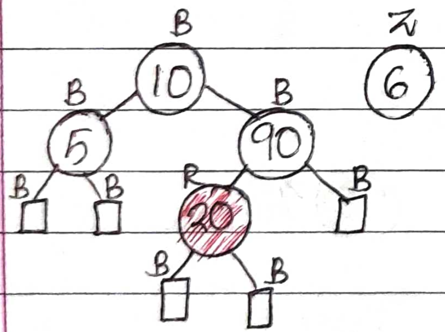
Now color [P[z]] ≠ Red

so, make color[root [T]] = Black

ie,



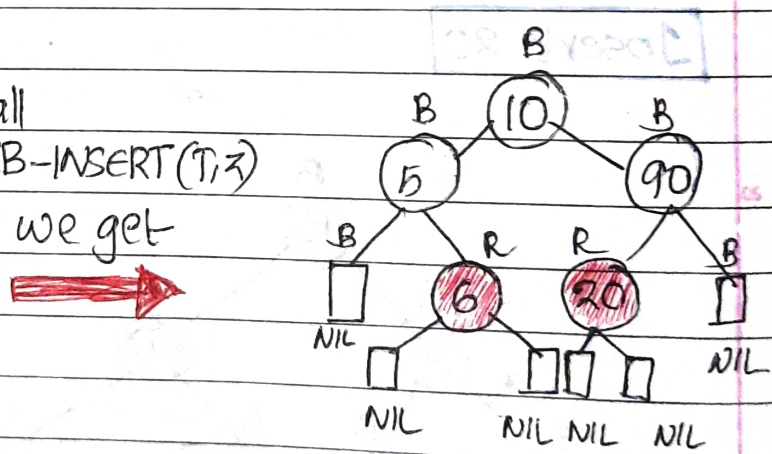
Insert 6



call

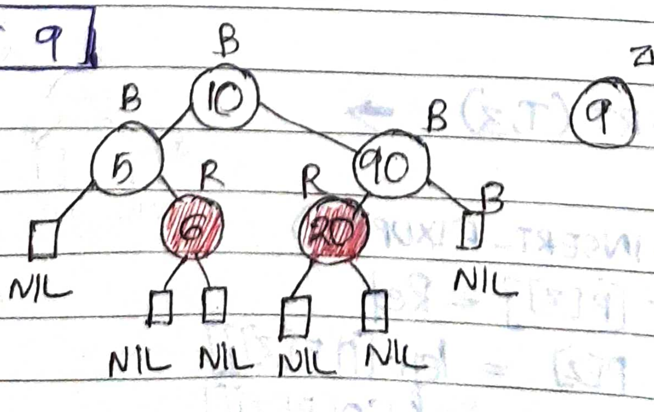
RB-INSERT(T,z)

we get



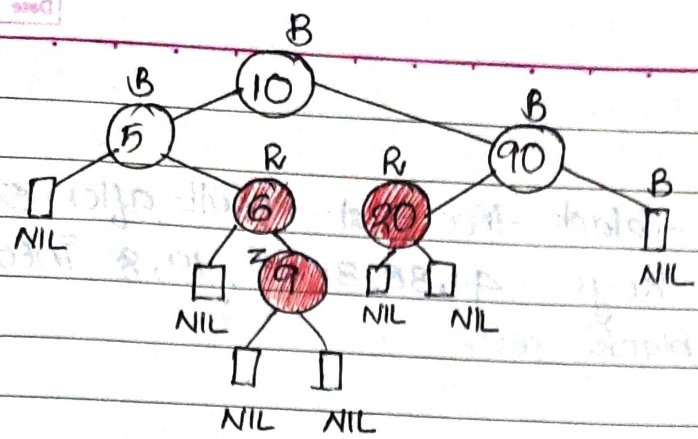
Here, color[P[z]] ≠ RED, so, no change.

INSERT 9



Call RB-INSERT(T,z)

we get



Now call RB-INSERT-FIXUP (T, z)

Here color$[P[z]] = $ RED

and $P[z] = right[P[P[z]]]$

then $y \longleftarrow left[P[P[z]]]$

i.e., leaf node.

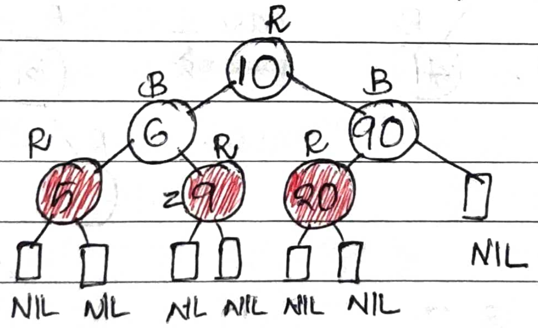
So, color$[y]$ = Black

Here, $z \rightleftharpoons right[P[z]]$

So, Color$[P[z]] \longleftarrow$ Black

Color$[P[P[z]]] \longleftarrow$ RED

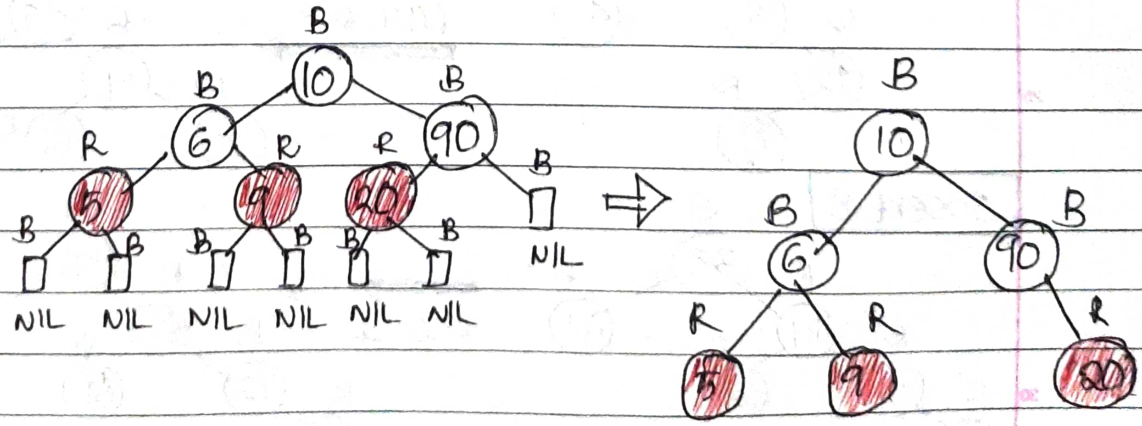
and left-rotate$[T, P[P[z]]]$



Now color$[P[z]] \neq$ RED

So, color$[Root[T]] \longleftarrow$ BLACK

and the final tree is,

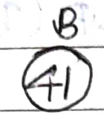


Example 3:

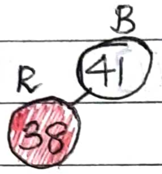
Show the red-black tree that result after successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty red-black tree.

Solution:

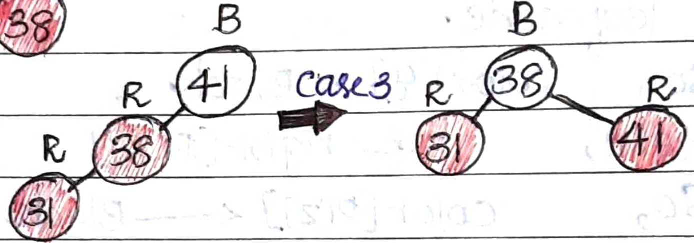
Insert 41



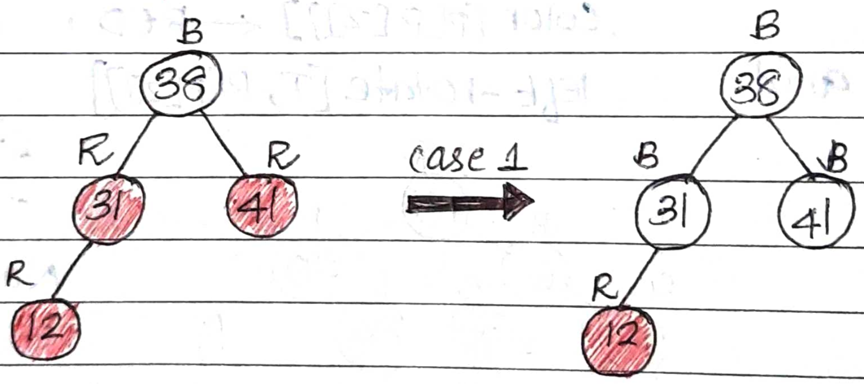
Insert 38



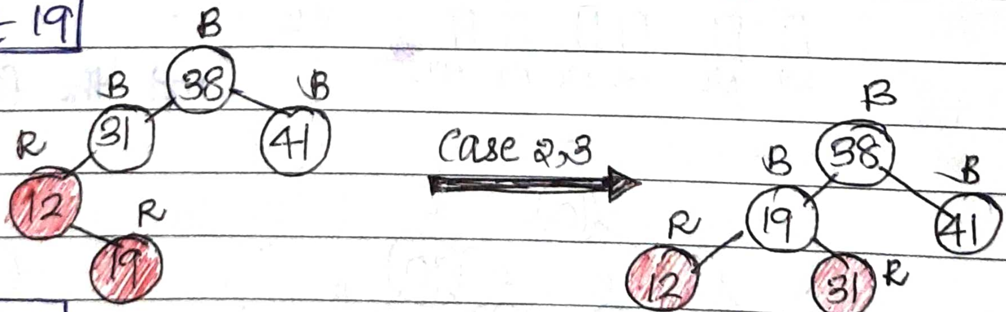
Insert 31



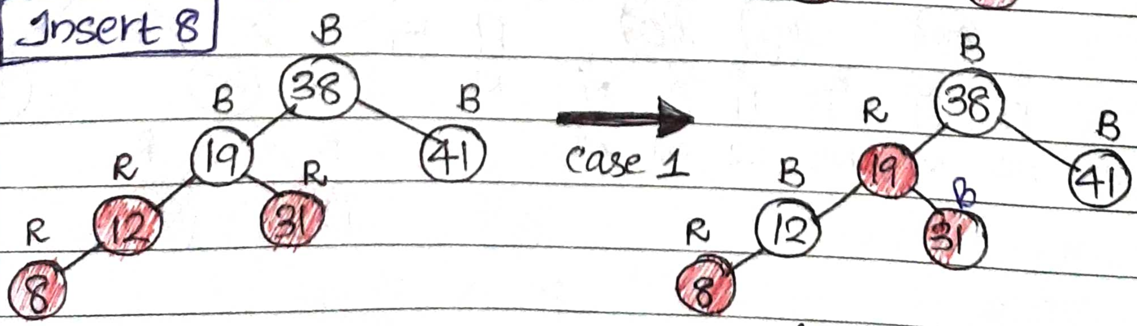
Insert 12



Isert 19



Insert 8



Final Tree.

Insertion — Red-Black Tree

Create a Red-Black Tree by inserting following sequence of numbers:

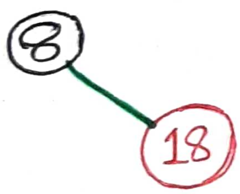
8, 18, 5, 15, 17, 25, 40, 80

INSERT 8

Tree is empty. So insert new node as Root Node with Black color.

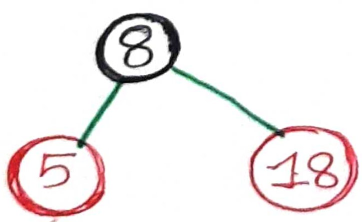
INSERT 18

Tree is not empty. So insert new node with Red color.

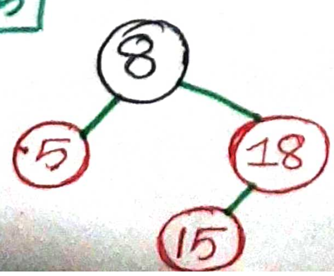


INSERT 5

Tree is not empty. So insert new node with Red color.



INSERT 15



① If Tree is empty, create new node as root node with color Black.

② If tree is not empty, create new node as leaf node with color Red.

③ If Parent of new node is black, then exit.

④ If Parent of new node is Red, then check the color of Parent's sibling of new node.

- ⓐ If color is black or null then do suitable rotation and recolor.
- ⓑ If color is Red then recolor & also check if Parent's parent of new node is not root, then recolor it & recheck.

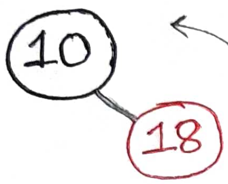
RED - BLACK TREE INSERTION

Create a Red Black tree by inserting following sequence of numbers:

10, 18, 7, 15, 16, 30, 25

Insert 10 :- Tree is empty → Black Node

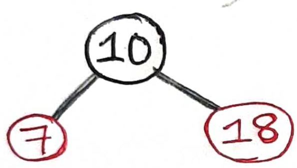
Insert 18 :- Tree is not empty → Red Node



check parent of New node ?

BLACK → ✓

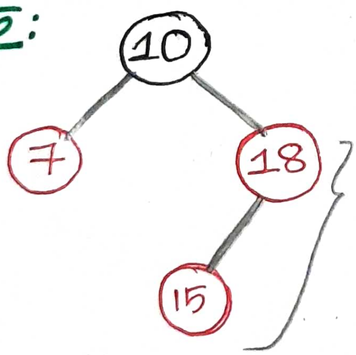
Insert 7:



Parent is BLACK

↳ EXIT.

Insert 15:

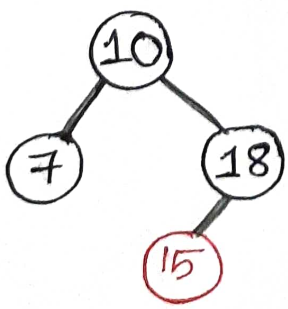


Violating Property 4

↓

Do Rule 4

↓ Recolor.



① If Tree is empty, create new node as Root Node with color BLACK.

② If Tree is not empty, create new node as leaf node with color RED.

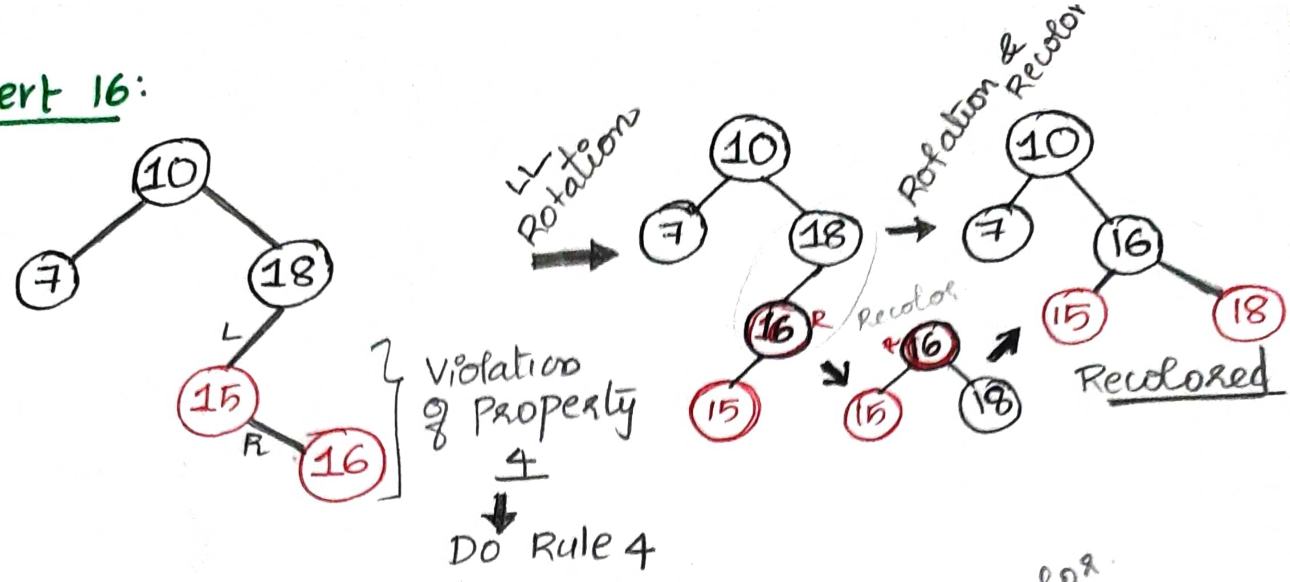
③ If Parent of new node is black, then EXIT.

④ If Parent of new node is RED, then check the color of Parent's Siblings of new node.

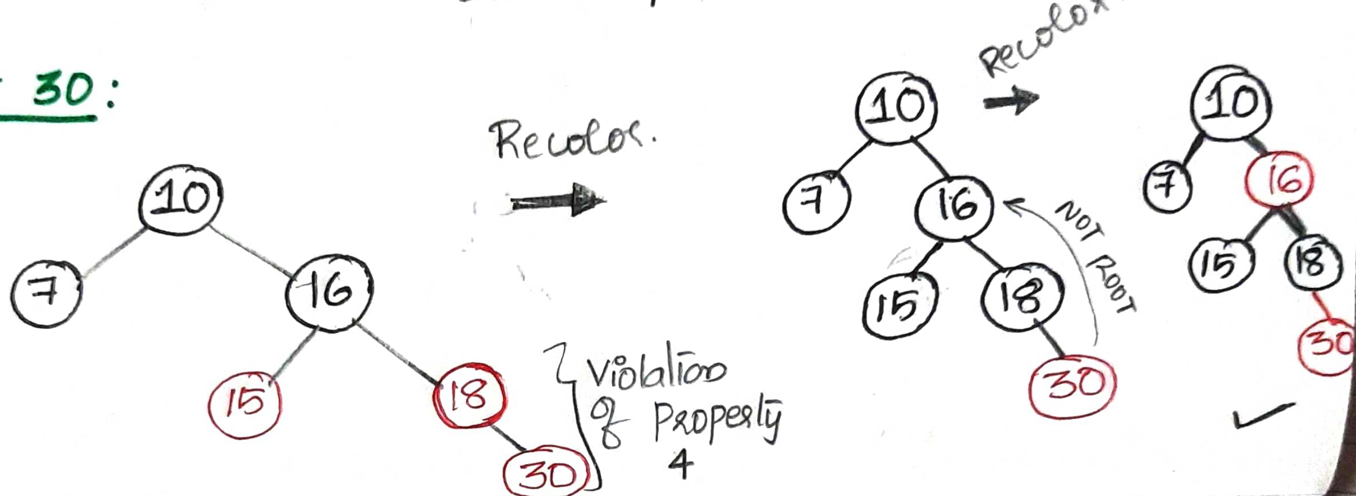
↳ ⓐ If uncle's color is Black, or NULL then do suitable rotation & recolor.

↳ Ⓑ If uncle's color is Red, then recolor & also check if Parent's Parent of new node is not root, then recolor it and recheck.

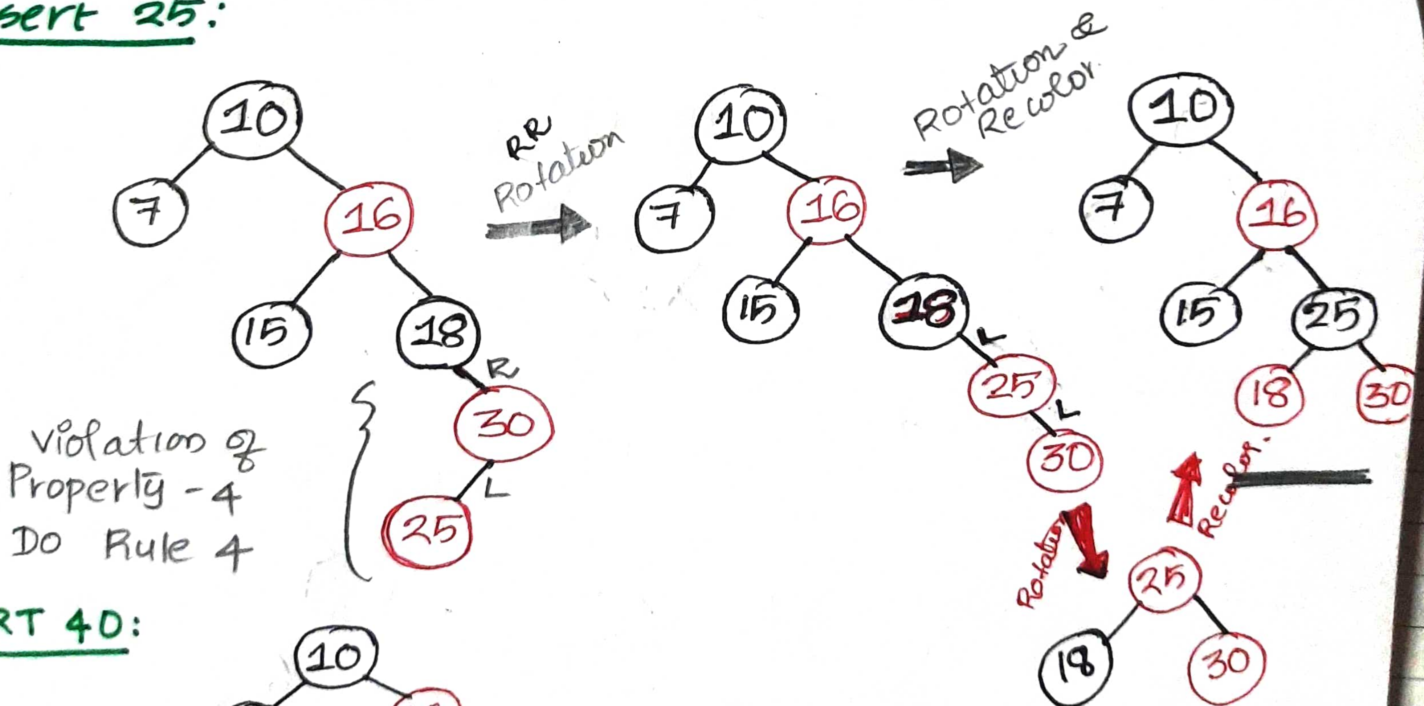
Insert 16:



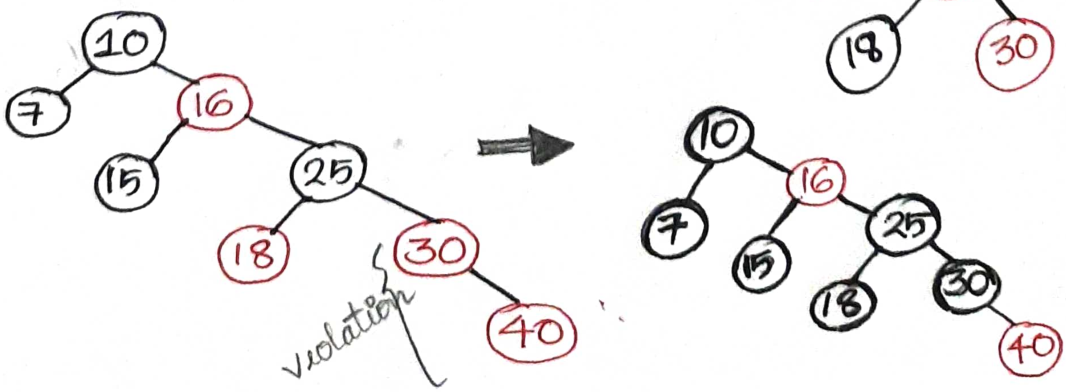
Insert 30:



Insert 25:



INSERT 40:



Red-Black Tree Deletion

Like Insertion, recoloring and rotations are used to maintain the Red-Black Properties.
In insert operation, we check color of uncle to decide the appropriate case.
In delete operation, we check color of sibling to decide the appropriate case.

The main property that violates after insertion is two consecutive reds. In delete, the main violated property is, change of black height in subtrees as deletion of a black node may cause reduced black height in one root to leaf path.

Deletion is fairly complex process. To understand deletion, notion of double black is used. When a black node is deleted and replaced by a black child, the child is marked as double black.
The main task now becomes to convert this double black to single black.

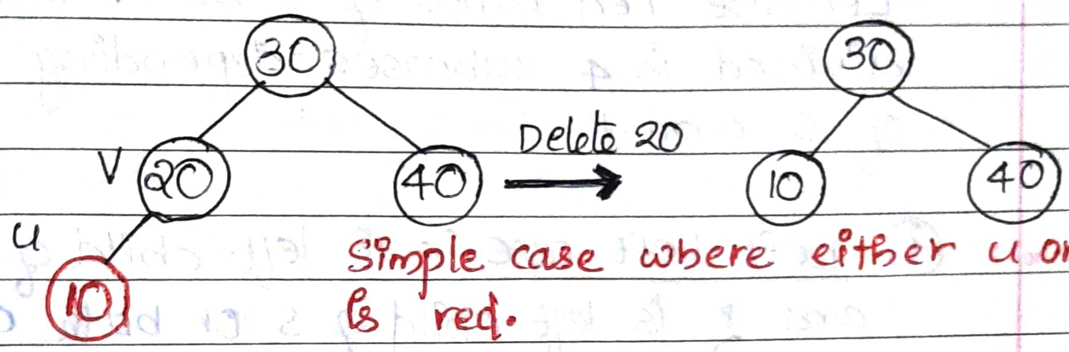
Deletion steps:

Following are detailed steps for deletion.

1. When we perform standard delete operation in BST, we always end up deleting a node which is either leaf or has only one child. So we only need to handle cases where a node is leaf or has one child.
   Let v be the node to be deleted and u be the child that replaces v (Note that u is NULL when v is a leaf and color of NULL is considered as Black).

(2) Simple Case : If either u or v is red

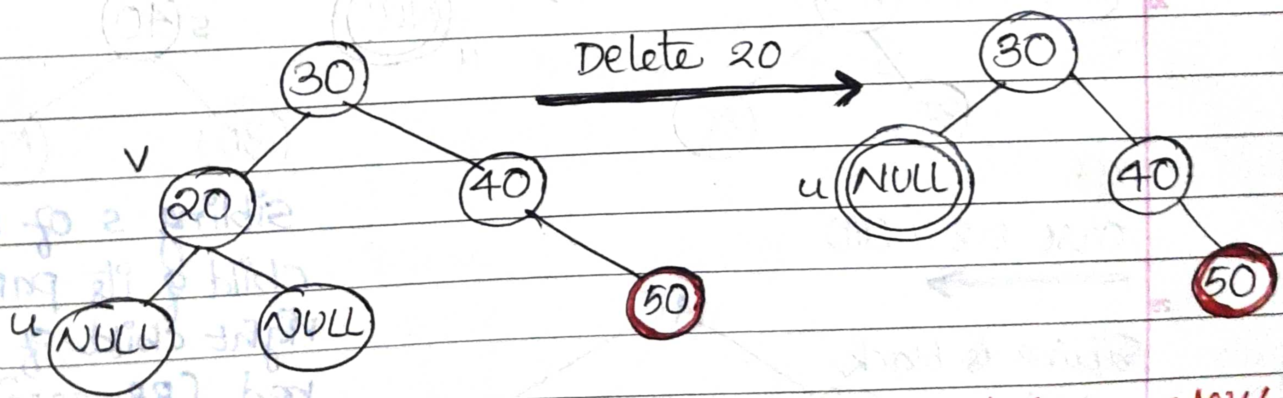
We mark the replaced child as black (No change in black height). Note that both u and v cannot be red as v is parent of u and two consecutive reds are not allowed in red-black tree.



Simple case where either u or v is red.

(3) If Both u and v are Black

(3.1) Color u as double black. Now our task reduces to convert this double black to single black. Note that if v is leaf, then u is NULL and color of NULL is considered as black. So the deletion of a black leaf also causes a double black.



When 20 is deleted, it is replaced by a NULL, so the NULL becomes double black. Note that deletion is not done yet, this double black must become single black.

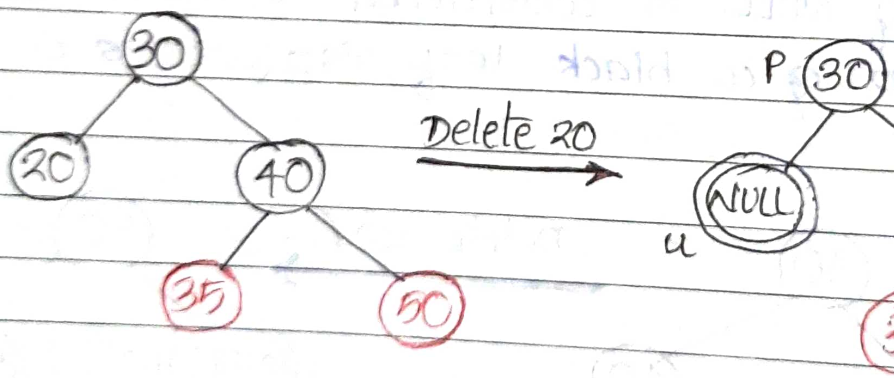
(3·2) Do following while the current node u is double black and it is not root.
Let sibling of node be s.

(a) if sibling s is black and at least one of sibling's children is red, perform rotation(s). Let the red child of s be r. This case can be divided in 4 subcases depending upon positions of s and r.

(i) Left Left case (s is left child of its parent and r is left child of s or both children of s are red).

(ii) Left Right case (s is left child of its parent and r is right child).

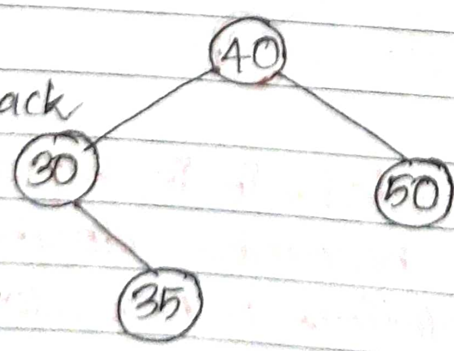
(iii) Right Right case (s is right child of its parent and r is right child of s or both children of s are red)



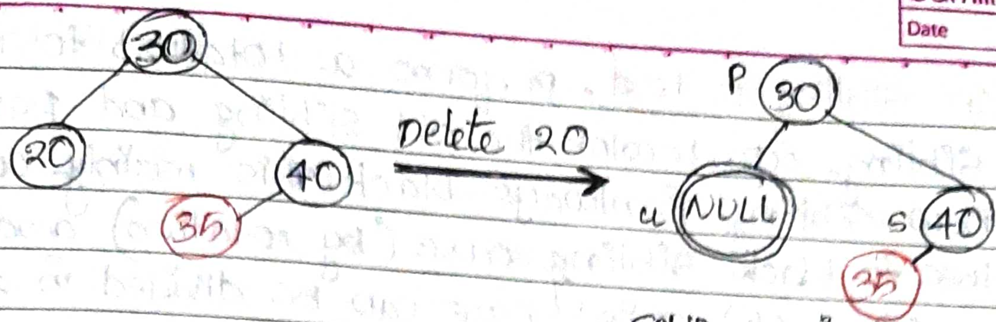
Sibling s of right child of its parent & right child of s is red (RR case).

case 3·2 a (iii)

Sibling is black with at least one red child.

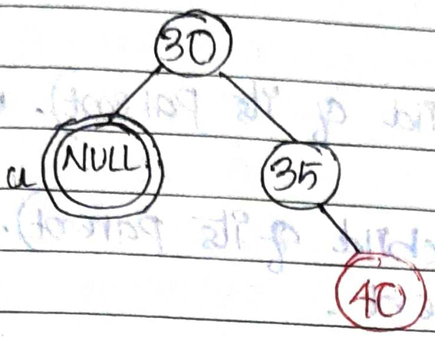


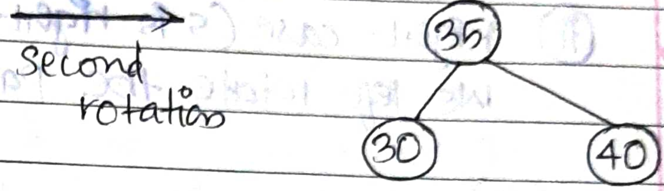
(iv) Right Left case (s is right child of its parent and r is left child of s)



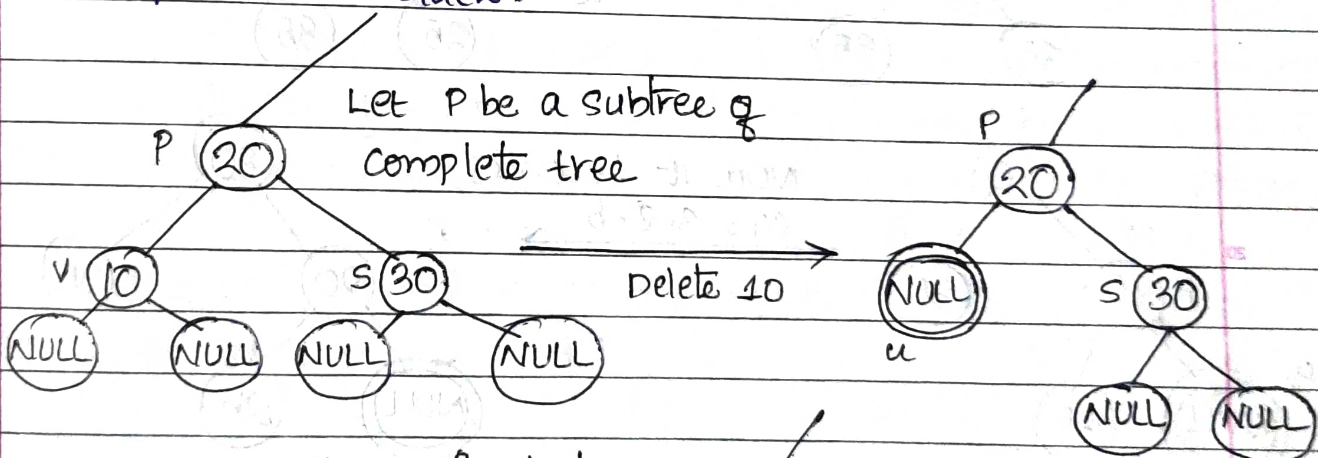
case 3.2 a (iv) sibling is black with atleast one red child.

Sibling s is right child of its Parent and the red child r of sibling is left child of it (RL case).

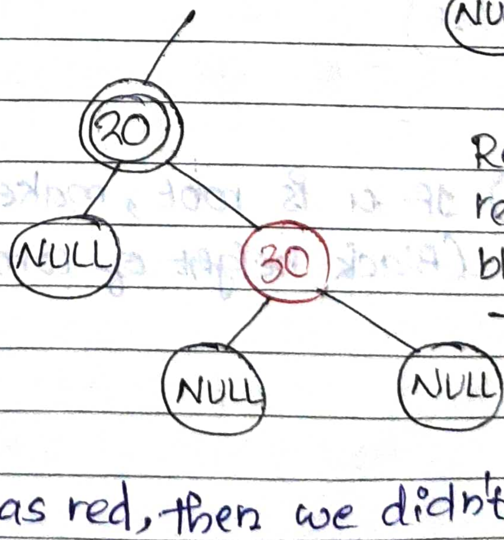




→ ⓑ If sibling is black and its both children are black, Perform recoloring, and recur for the parent if Parent is black.



cas 3.2 b sibling s is Black and both of its children are also black



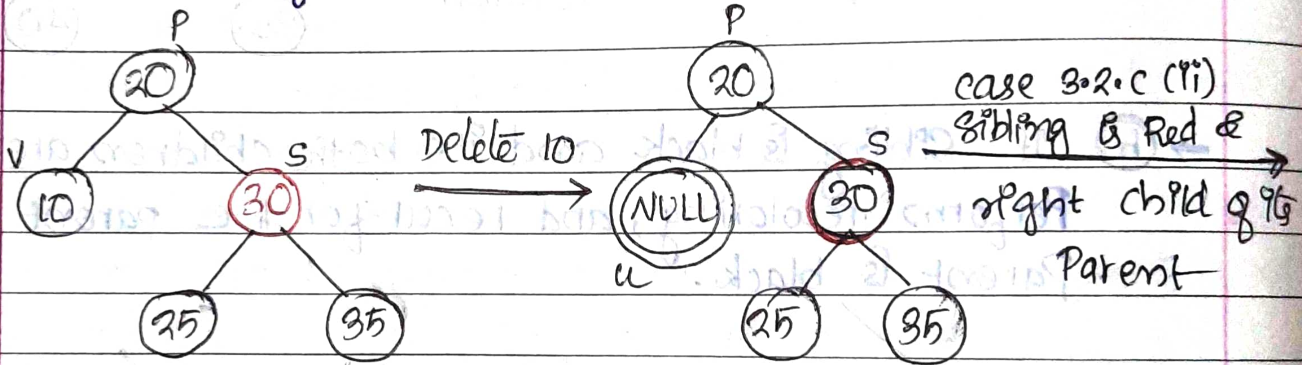
Recur for 20 to remove double black from it.

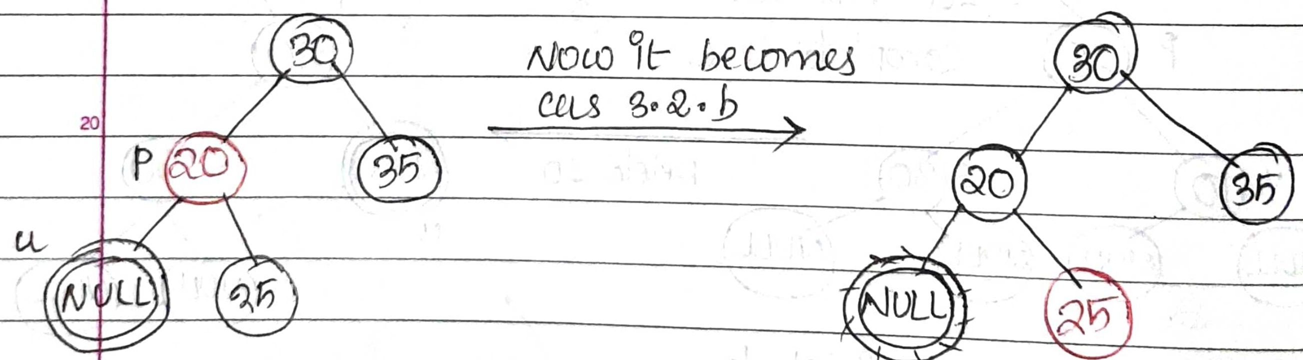
In this case, if Parent was red, then we didn't need to recur for Parent, we can simply make it black (red + double black = single black).

(c) If sibling is red, perform a rotation to move old sibling up, recolor the old sibling and parent. The new sibling is always black. This mainly converts the tree to black sibling case (by rotation) and leads to case (a) or (b). This case can be divided in 2 subcases.

(i) Left case (s is left child of its parent). We right rotate the parent.

(ii) Right case (s is right child of its parent). We left rotate the parent.

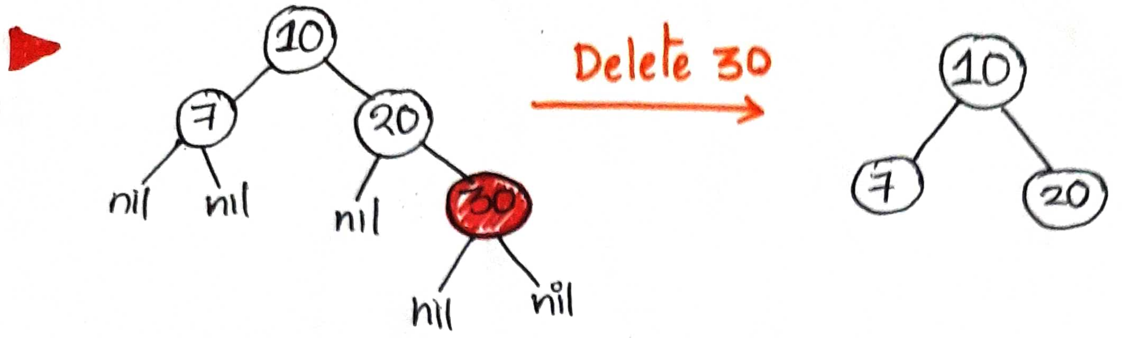


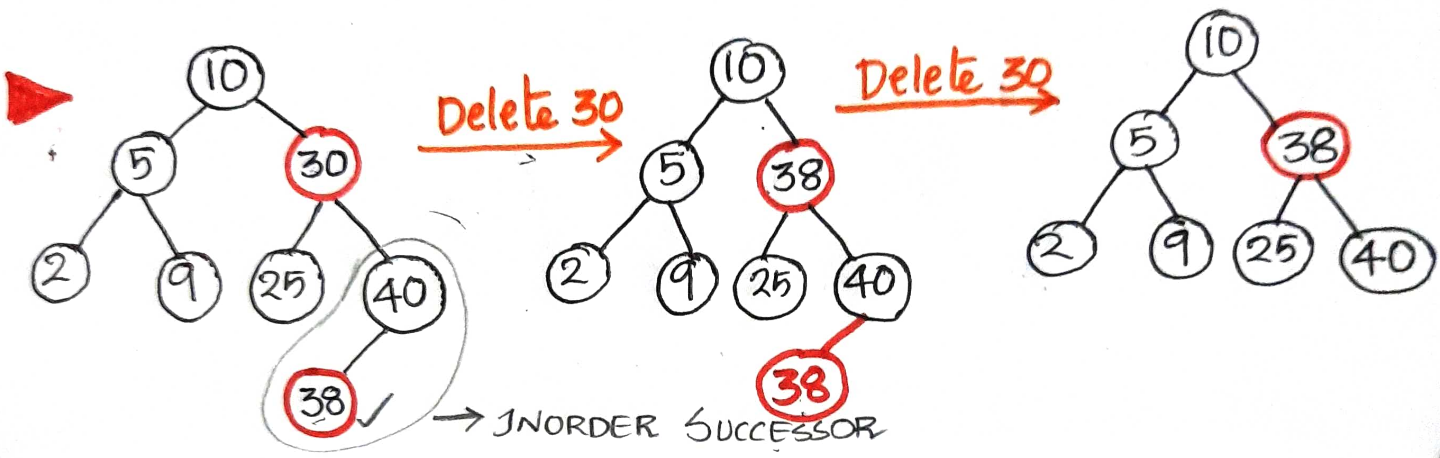


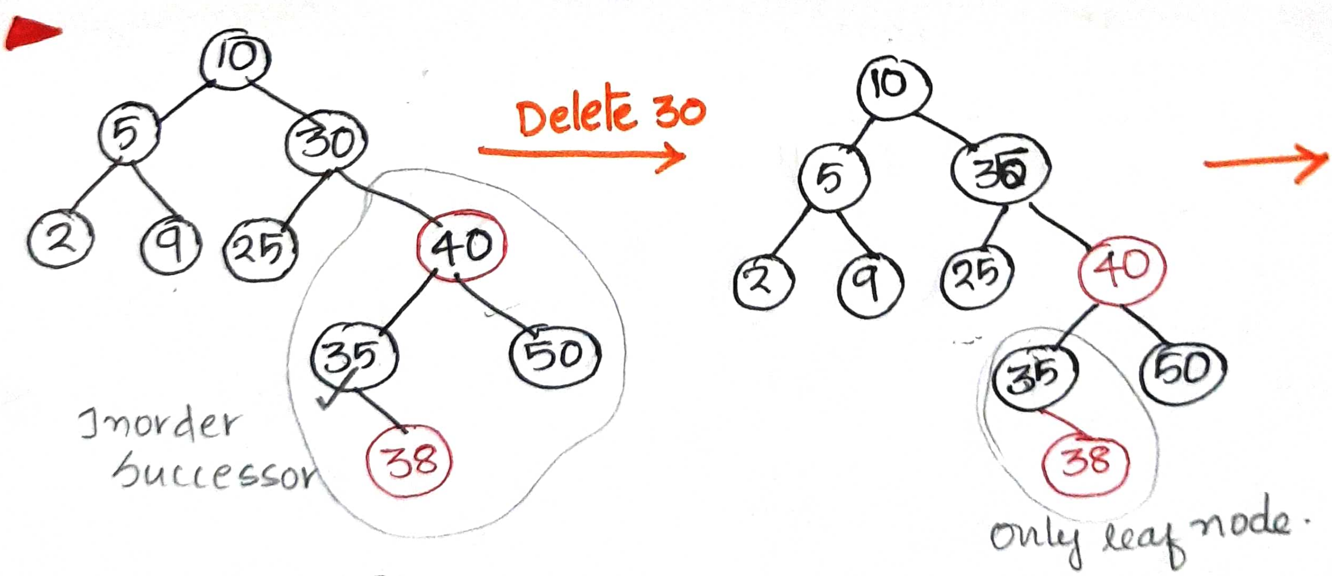
(3.3) If u is root, make it single black and return (Black height of complete tree reduces by 1).

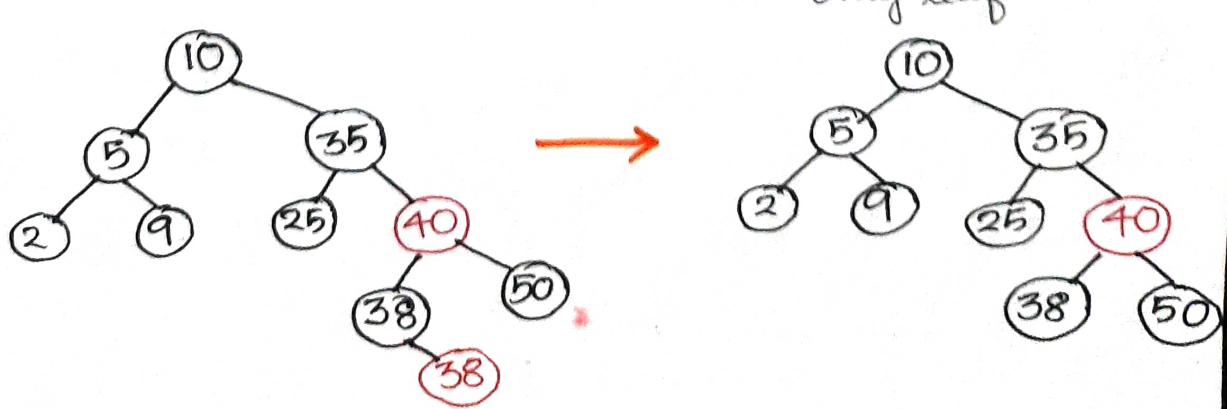
DELETION IN RED-BLACK TREE

CASE 1 If node to be deleted is RED, just delete it.

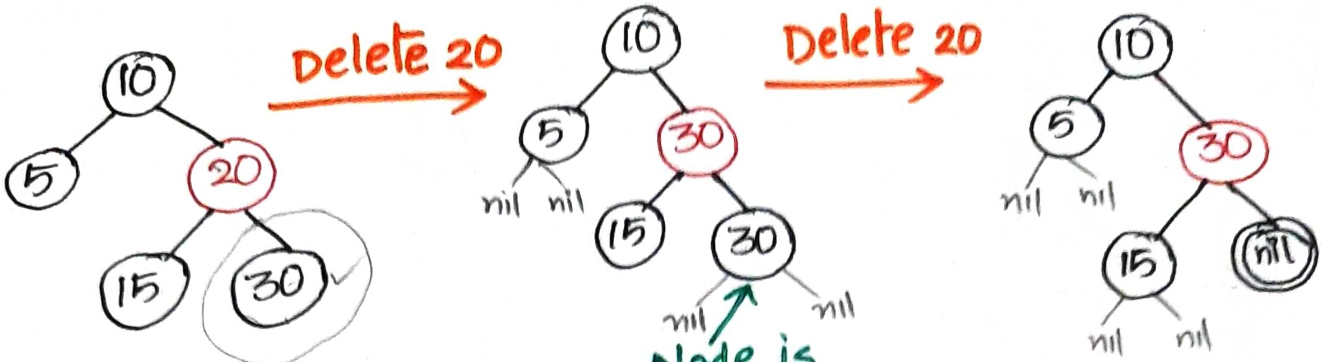




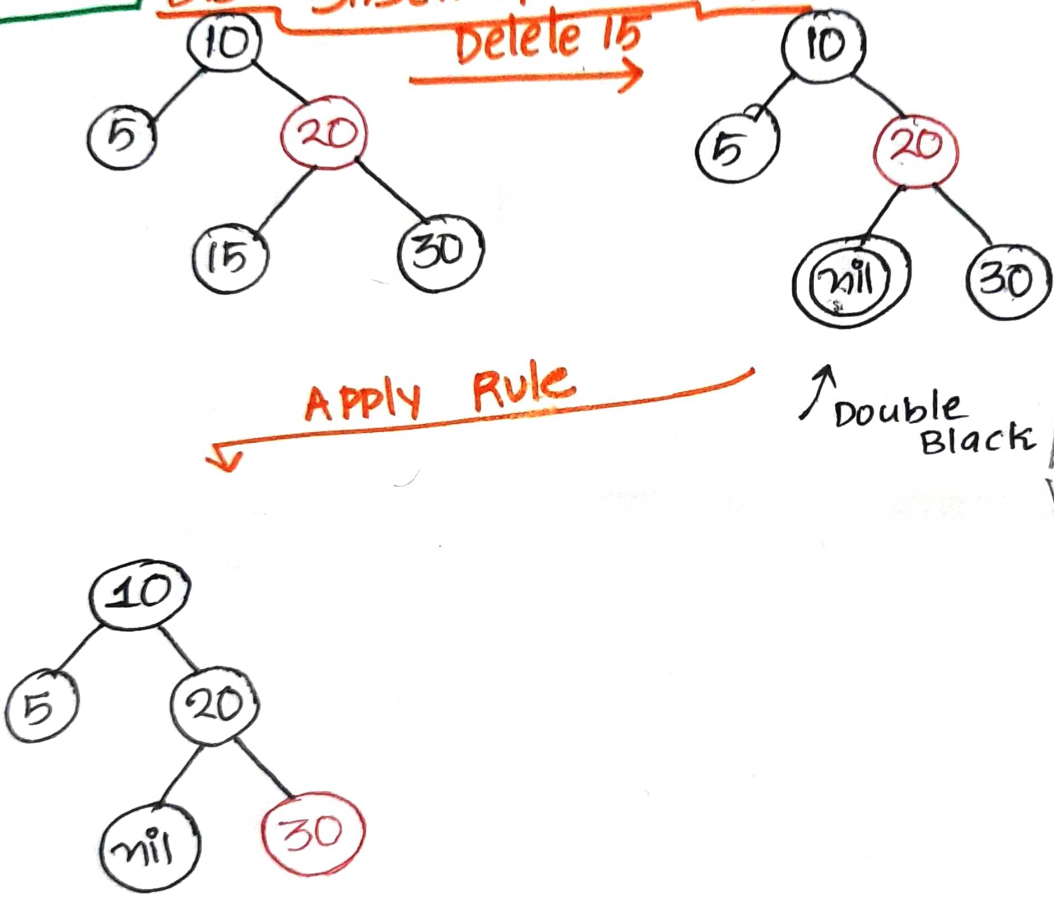




CASE 2 Double Black Condition:



CASE 3 DB's SIBLING IS BLACK

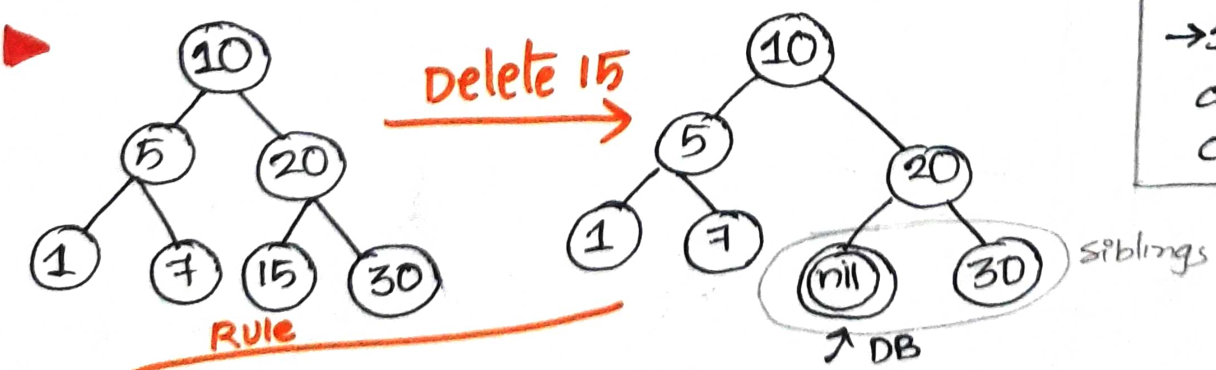


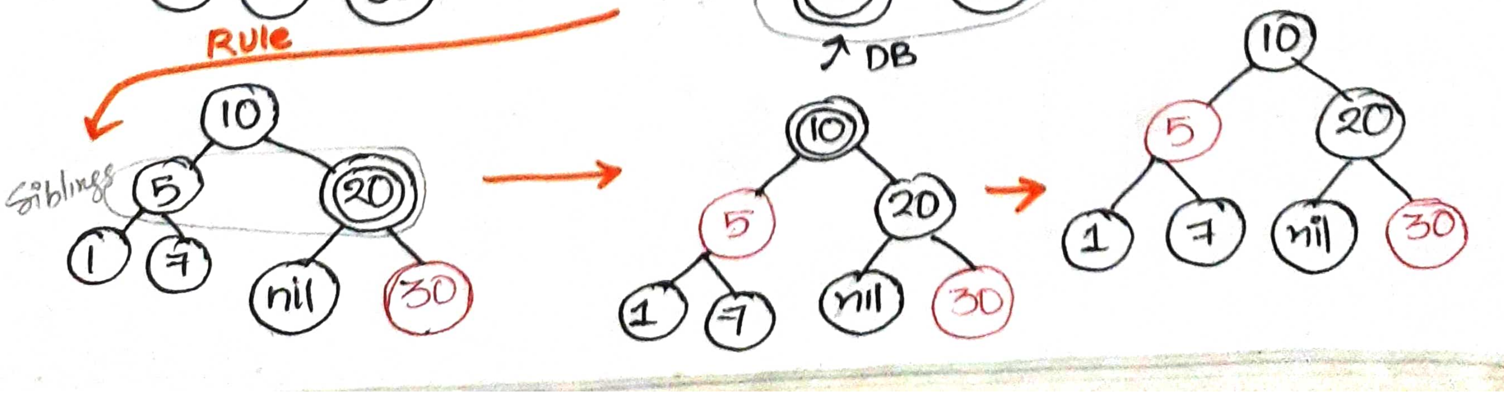
Rules

1. If Root is DB, Just Remove it

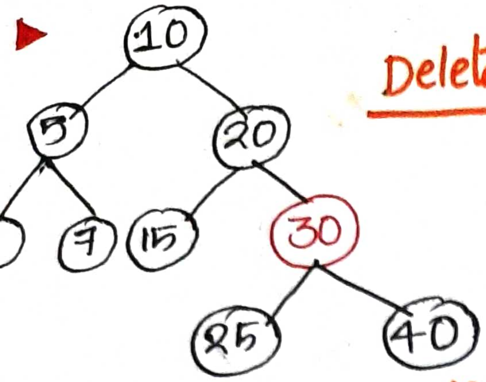
CASE 3

2. If DB's sibling is black & both its children are black
   - → Remove DB
   - → Add Black to its DB's Parent
     - → If Parent is red, → Black
     - → If Parent is black → DB
   - → Make Sibling red
   - → If still DB exists, apply other conditions.

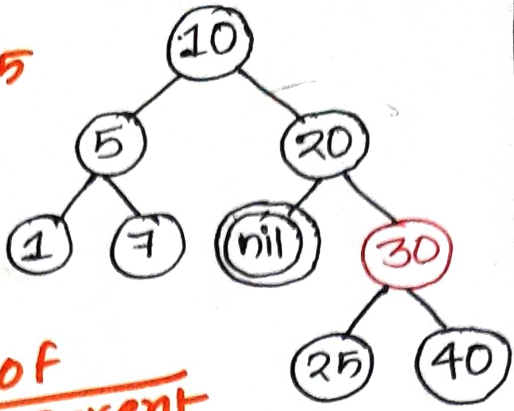




CASE 4 IF DB's SIBLING IS RED



Delete 15



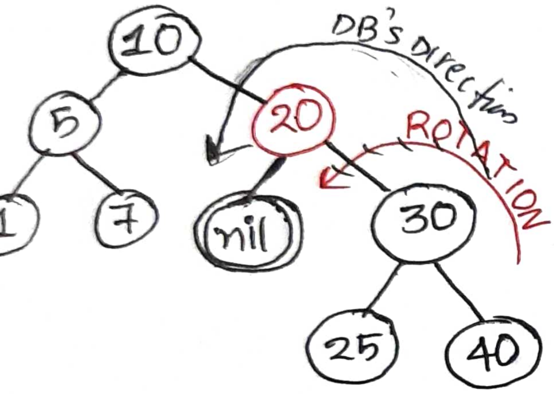
Rule

(4) Swap color of Sibling & Parent of DB.

↳ Rotate Parent in DB direction.

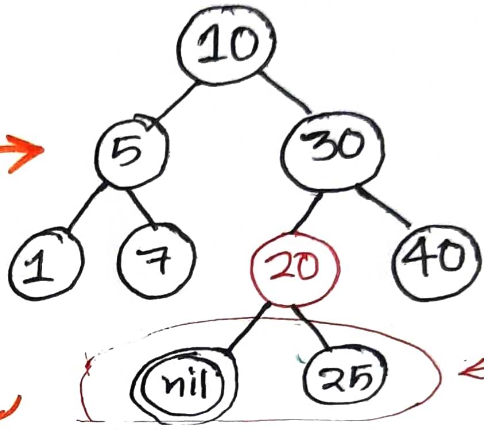
↳ Reapply cases.

Swap color of Sibling & parent



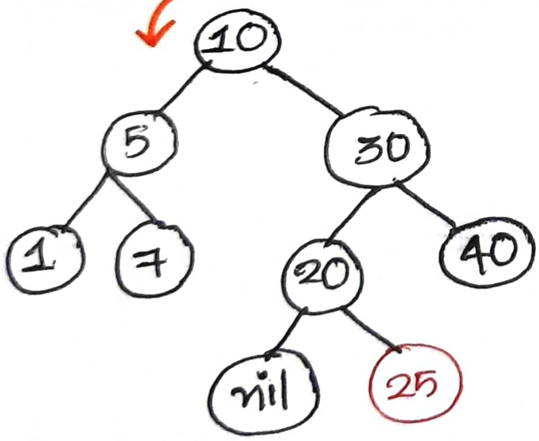
DB's Direction

ROTATION



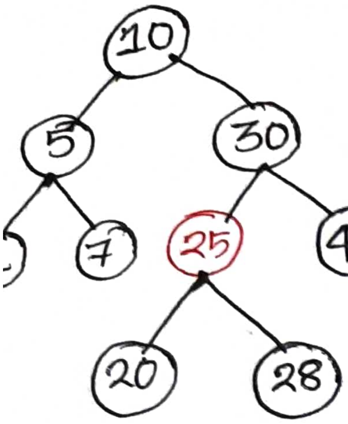
← case 3 // sibling is black

① remove DB
② Parent Black
③ Sibling Red

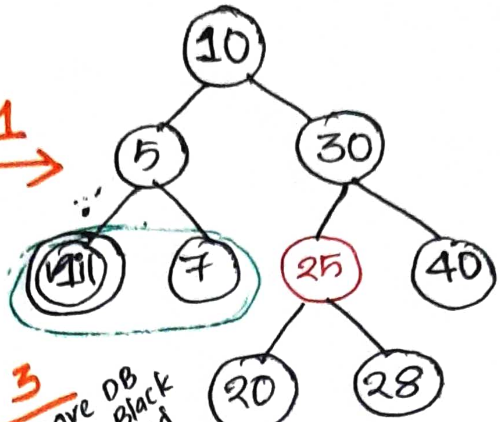


CASE 5

DB's sibling is **Black**, siblings child who is far from DB is **Black**, but near child to DB is RED.



Delete 1



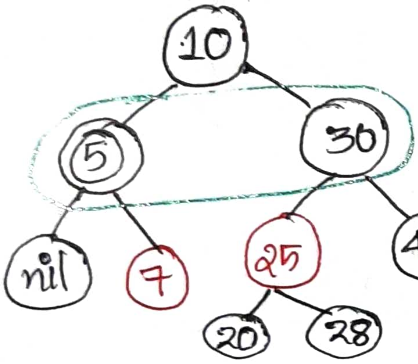
Rules

→ Swap colors of sibling and sibling's child who is near to DB

→ Rotate sibling in opposite direction to DB

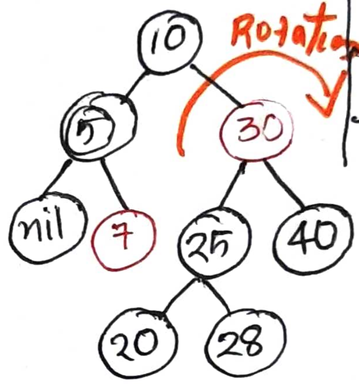
→ Apply case 6.

CASE 3 {Remove DB Parent Black Sibling Red



✓ Farthest child is black.

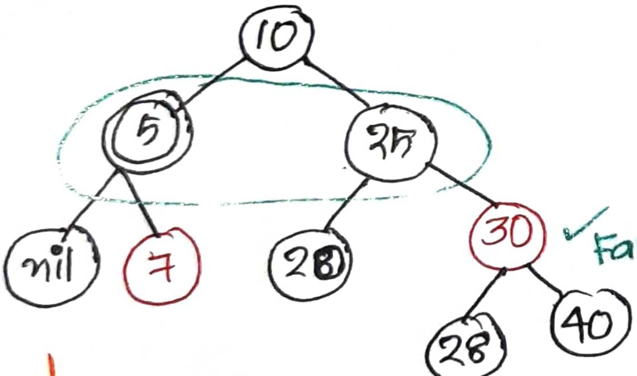
Swap color (Rule 1)



Rotation

CASE 6

DB's sibling is **Black**, sibling's child who is far from DB is RED



✓ Farthest child is red

Rules

① Swap colors of DB's Parent & sibling.

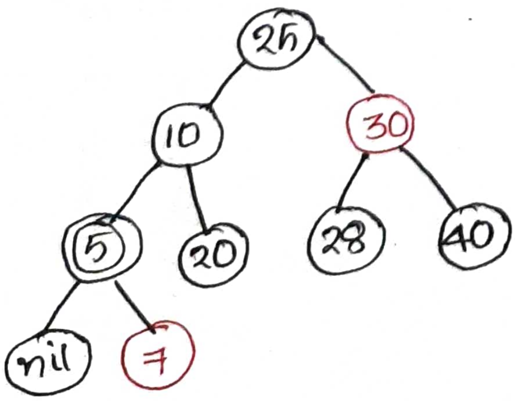
② Rotate Parent of DB in DB's direction.

③ Remove DB

④ change color of red child to Black.

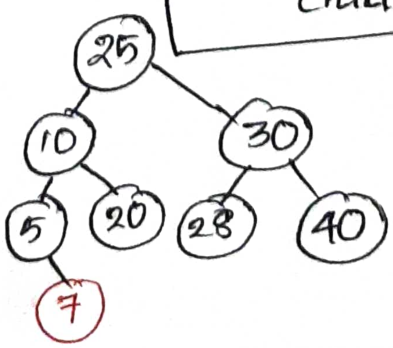
swap colors of Parent & sibling of DB. //Both are Black → 10 & 25

→ Do Rotation



Remove DB

change color of farthest node.



(8) B-TREES

B-Tree is a self-Balancing search tree. In most of the other self-balancing search trees, it is assumed that everything is in main memory. To understand use of B-trees, we must think of huge amount of data that cannot fit in main memory. When the no. of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to main memory access time.

The main idea of using B-Trees is to reduce the no. of disk accesses. Most of the tree operations require O(h) disk accesses where h is height of the tree. B-tree is a fat tree. Height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, a B-Tree node size is kept equal to the disk block size. Since h is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary search trees like AVL tree, Red Black tree.

A B-tree T is a rooted tree (whose root is T.root) having the following properties:

(1) Every node x has the following attributes:

- (a) x.n → the no. of keys currently stored in node x,
- (b) the x.n keys themselves, x.key₁, x.key₂, ... x.key$_{x.n}$ stored in nondecreasing order, so that $x.key_1 \leq x.key_2 \leq \ldots \leq x.key_{x.n}$.
- (c) x.leaf, a boolean value that is TRUE if x is a leaf and FALSE if x is an internal medial node.

② Each internal node x also contains $x.n+1$ Pointers $x.c_1, x.c_2, \ldots \ldots x.c_{x.n+1}$ to its children. Leaf node have no children, and so their c_i attributes are undefined.

③ The keys $x.key_i$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $x.c_i$, then

$$k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \ldots \leq x.key_{x.n} \leq k_{x.n+1}$$

④ All leaves have the same depth, which is the tree's height h.

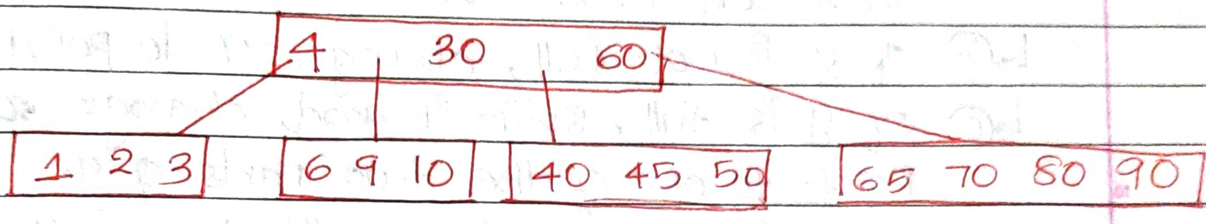
⑤ Nodes have lower and upper bound on the no. of keys ~~that~~ they can contain. We express these bounds in terms of a fixed integer $t \geq 2$ called the minimum degree of the B-tree.

↳ ⓐ Every node other than the root must have atleast $t-1$ keys. Every internal node other than the root thus has at least t children.
If the tree is nonempty, the root must have atleast one key.

↳ ⓑ Every node may contain at most $2t-1$ keys. Therefore, an internal node may have at most $2t$ children. We say that a node is full if it contains exactly $2t-1$ keys.

The simplest B-Tree occurs when t=2. Every internal node then has either 2, 3, or 4 children, and we have a 2-3-4 tree.

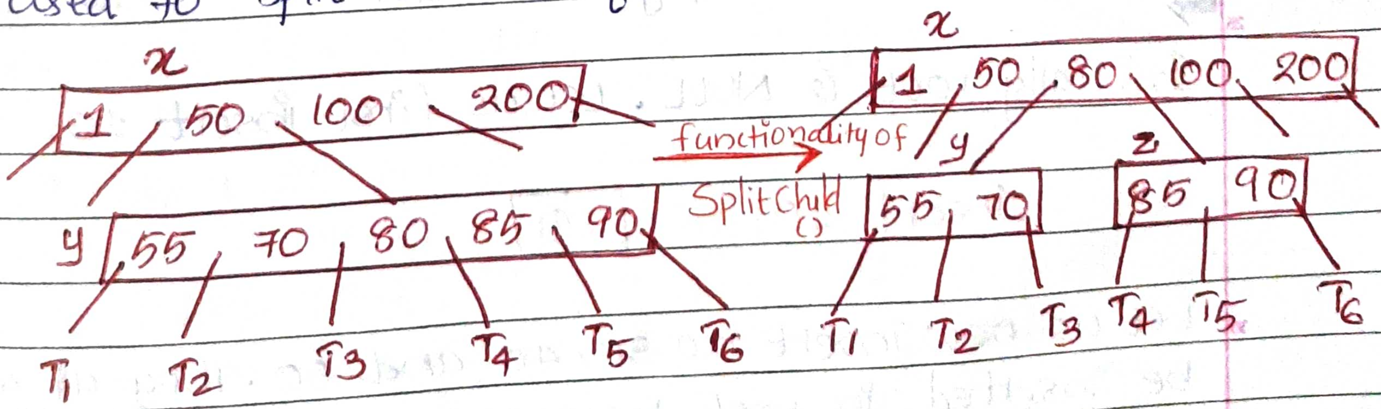
Example: B-Tree with minimum degree 3.



INSERTION:

A new key is always inserted at leaf node. Let the key to be inserted be k. Like BST, we start from root and traverse down till we reach a leaf node. Once we reach a leaf node, we insert the key in that leaf node. Unlike BSTs, we have a predefined range on number of keys that a node can contain. So before inserting a key to node, we make sure that the node has extra space.

We use an operation called splitChild() that is used to split a child of a node.



Algorithm:

1. Initialize x as root.
2. While x is not leaf, do following
   - (a) Find the child of x that is going to be traversed next. Let the child be y.
   - (b) If y is not full, change x to point y.
   - (c) If y is full, split it and change x to point to one of the two parts of y. If k is smaller than mid key in y, then set x as first part of y. Else second part of y. When we split y, we move a key from y to its parent x.
3. The loop in step 2 stops when x is leaf. x must have space for one extra key as we have been splitting all nodes in advance. So simply insert k to x.

Example: Let us understand the algorithm with an example tree of minimum degree 't' as 3 and a sequence of integers 10, 20, 30, 40, 50, 60, 70, 80 and 90 in an initial empty ~~to~~ B-Tree.

→

Initially root is NULL. Let us first insert 10.

Insert 10 $\boxed{10}$

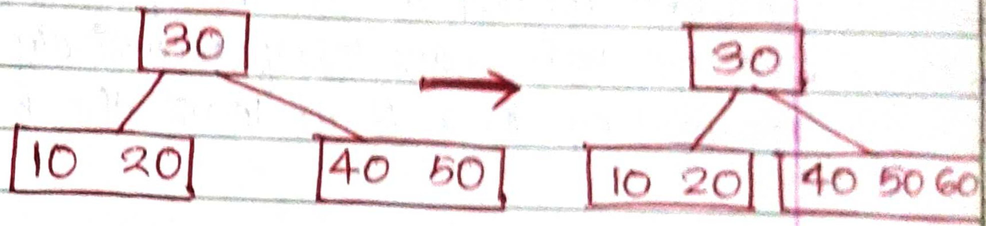
Let us now insert 20, 30, 40 and 50. They all will be inserted in root because maximum no. of keys a node can accomodate is $2t-1$ which is 5.

Insert 20, 30, 40 and 50

10 20 30 40 50

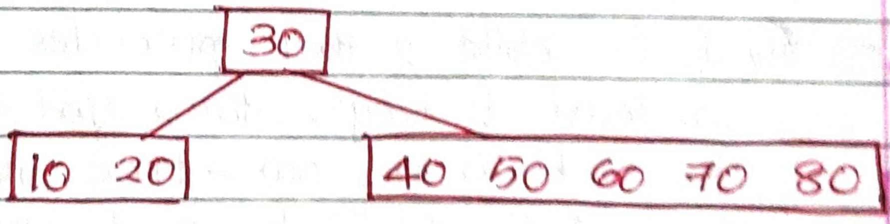
Let us now insert 60. Since root node is full, it will first split into 2, then 60 will be inserted into the appropriate child.

Insert 60



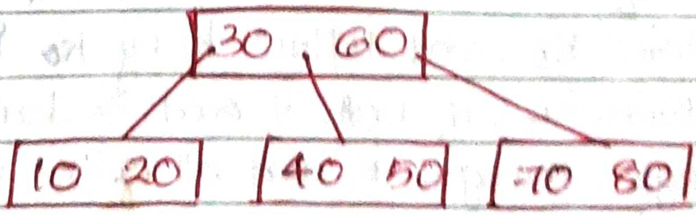
Let us now insert 70 and 80. These new keys will be inserted into the appropriate leaf without any split.

Insert 70 and 80



Let us now insert 90. This insertion will cause a split. The middle key will go up to the parent.

Insert 90



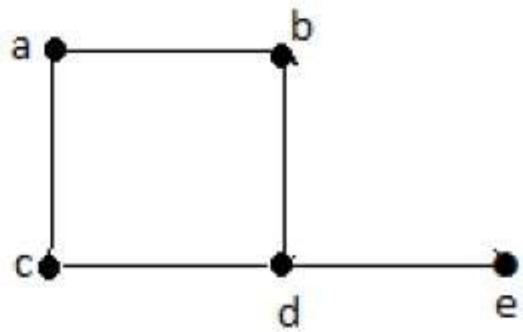
MODULE III

Graphs – DFS and BFS traversals, complexity, Spanning trees – Minimum Cost Spanning Trees, single source shortest path algorithms, Topological sorting, strongly connected components.

GRAPHS

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices,** and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets **(V, E),** where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph −



In the above graph,

V = {a, b, c, d, e}

E = {ab, ac, bd, cd, de}

GRAPH TRAVERSAL

Graph traversal is technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices to be visit in the search process. A graph traversal finds the egdes to be used in the search process without creating loops that means using graph traversal we visit all vertices of graph without getting into looping path.

There are two types of graph traversal techniques and they are as follows...

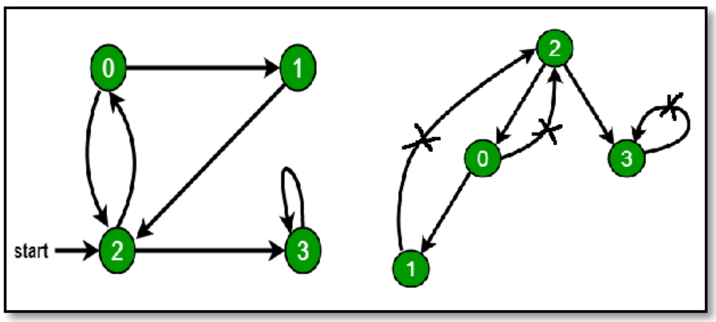
- DFS (Depth First Search)
- BFS (Breadth First Search)

Depth First Traversal or DFS for a Graph

Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.

DFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without any loops. We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal of a graph.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Depth First Traversal of the following graph is 2, 0, 1, 3.



We use the following steps to implement DFS traversal...

Step 1: Define a Stack of size total number of vertices in the graph.

Step 2: Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.

Step 3: Visit any one of the adjacent vertex of the verex which is at top of the stack which is not visited and push it on to the stack.

Step 4: Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.

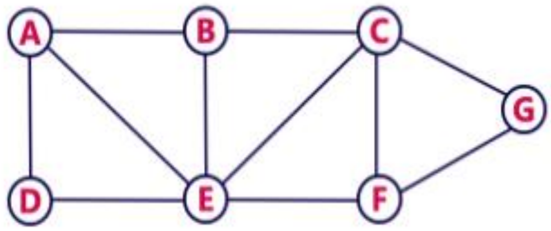
Step 5: When there is no new vertex to be visit then use back tracking and pop one vertex from the stack.

Step 6: Repeat steps 3, 4 and 5 until stack becomes Empty.

Step 7: When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

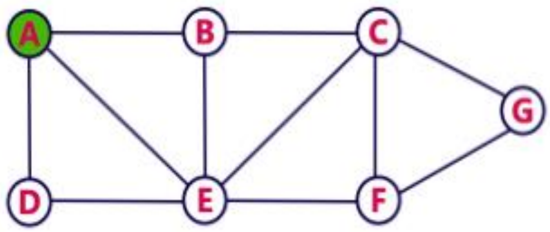
EXAMPLE 1

Consider the following example graph to perform DFS traversal



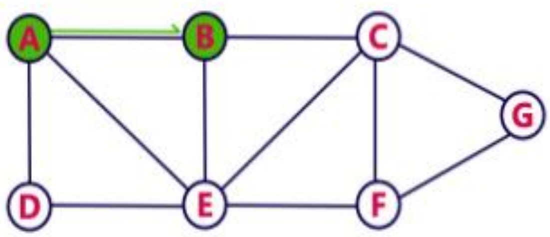
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



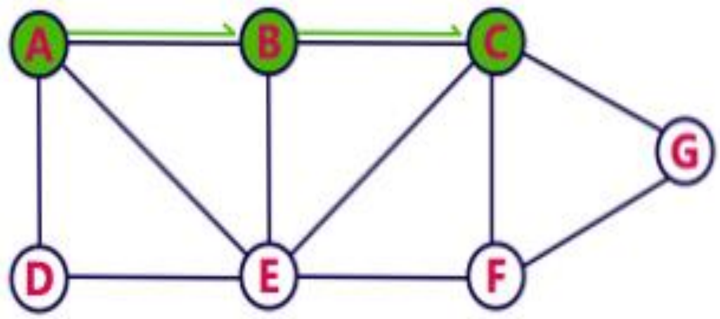
Step 2:

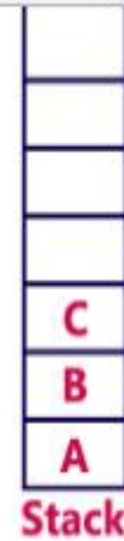
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



Step 3:

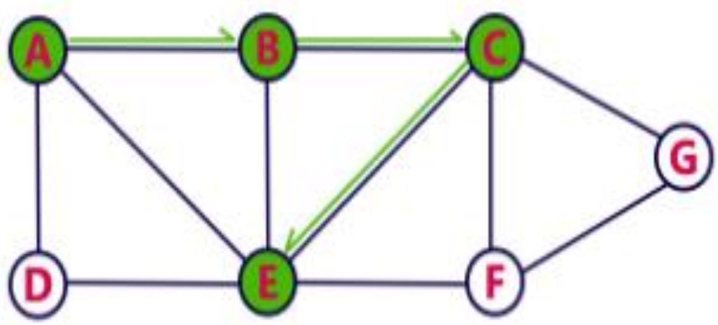
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

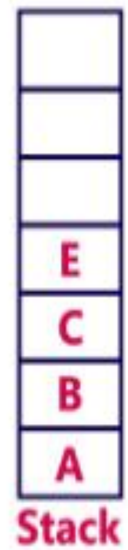




Step 4:

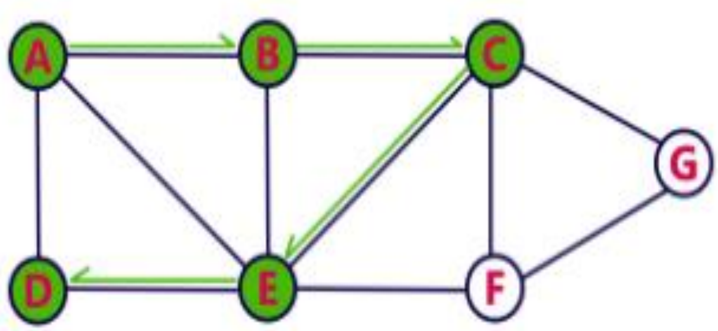
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

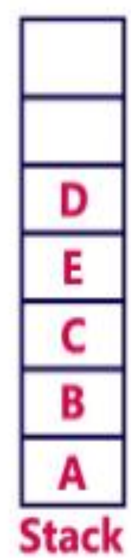




Step 5:

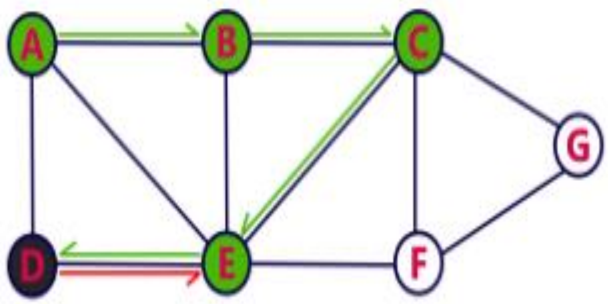
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

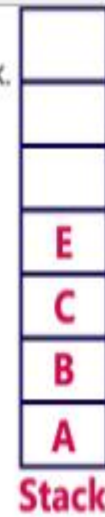




Step 6:

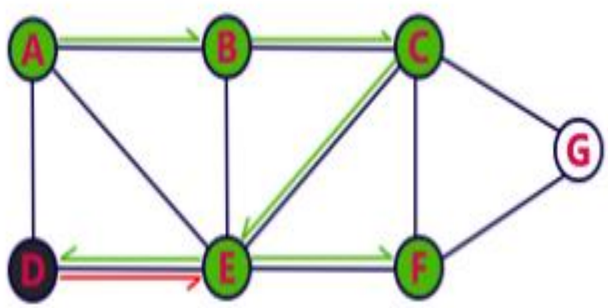
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

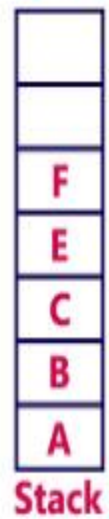




Step 7:

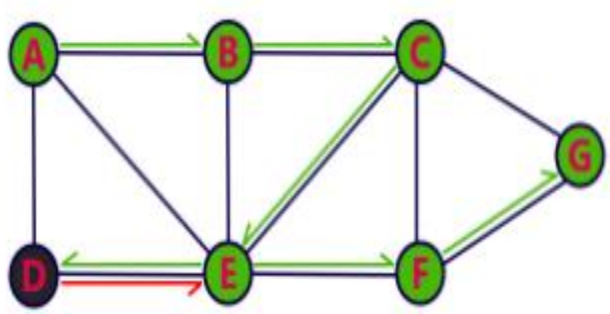
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

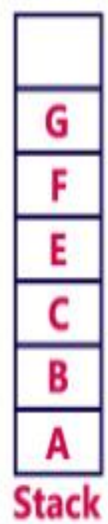




Step 8:

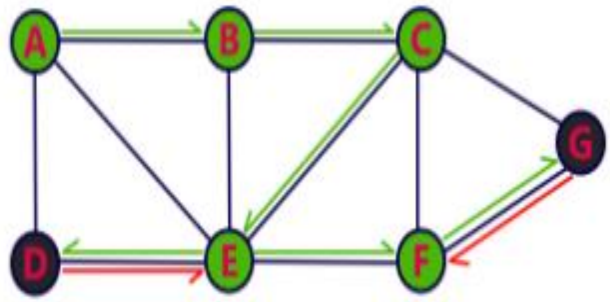
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

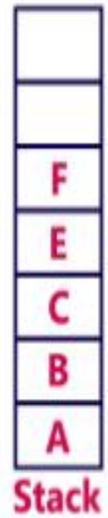




Step 9:

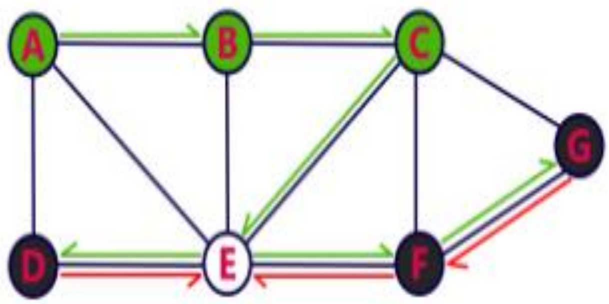
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.





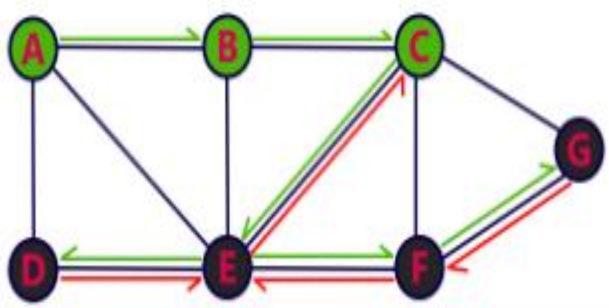
Step 10:

- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.



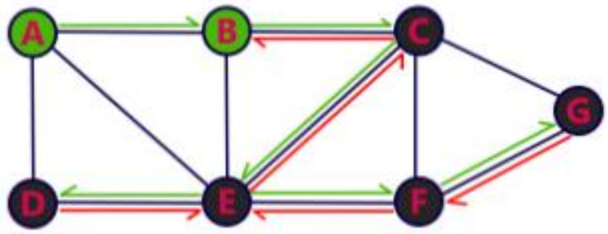
Step 11:

- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.



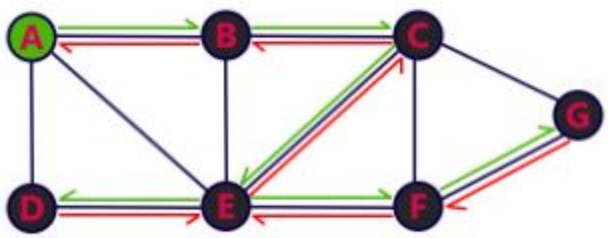
Step 12:

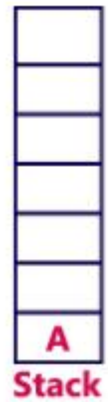
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.



Step 13:

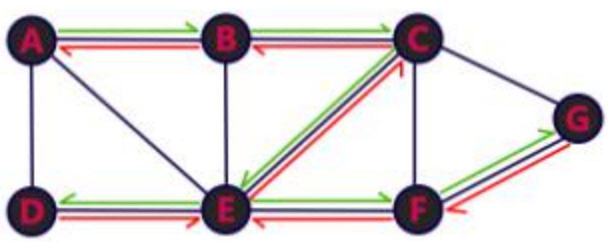
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

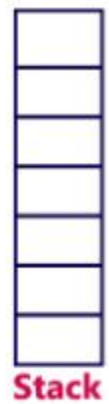




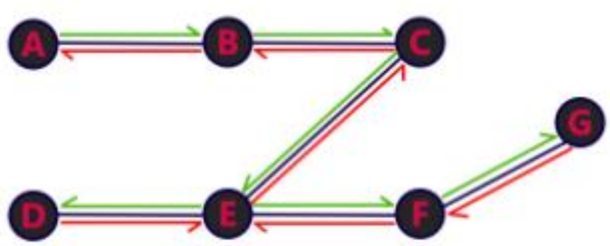
Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.

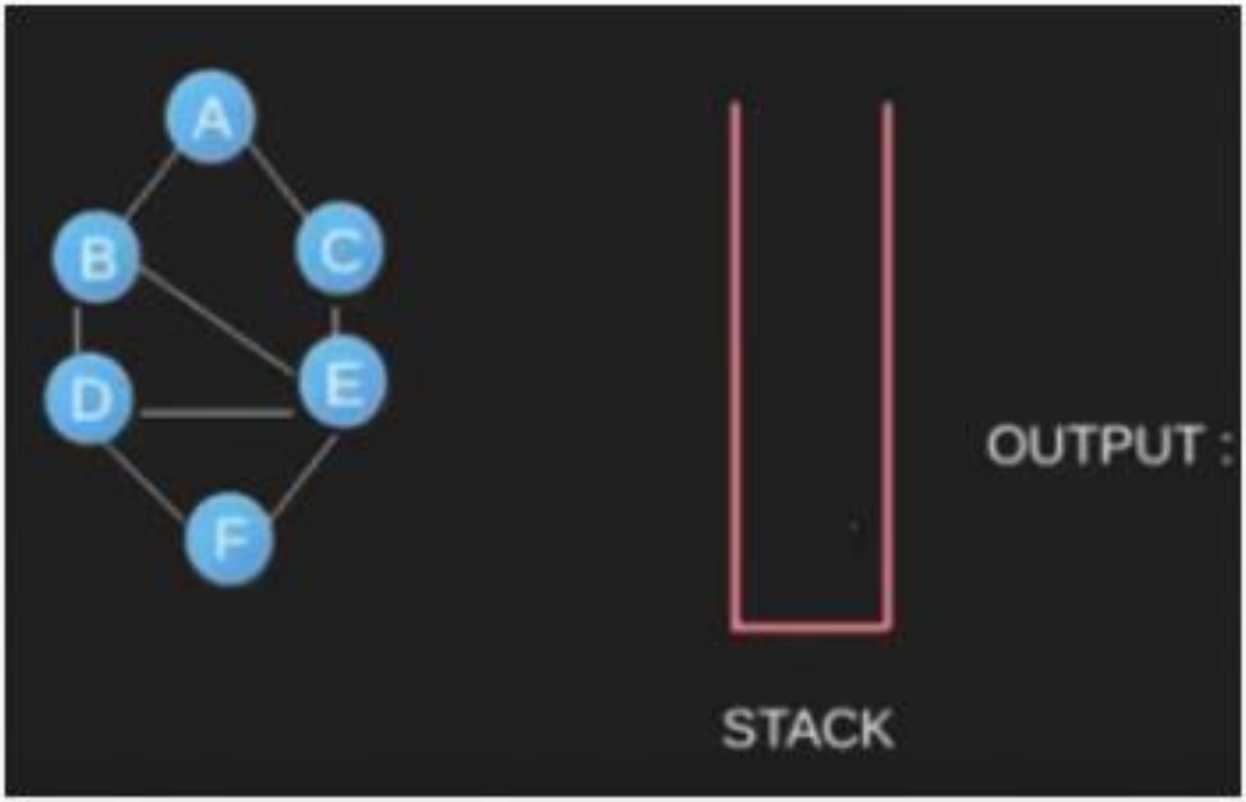


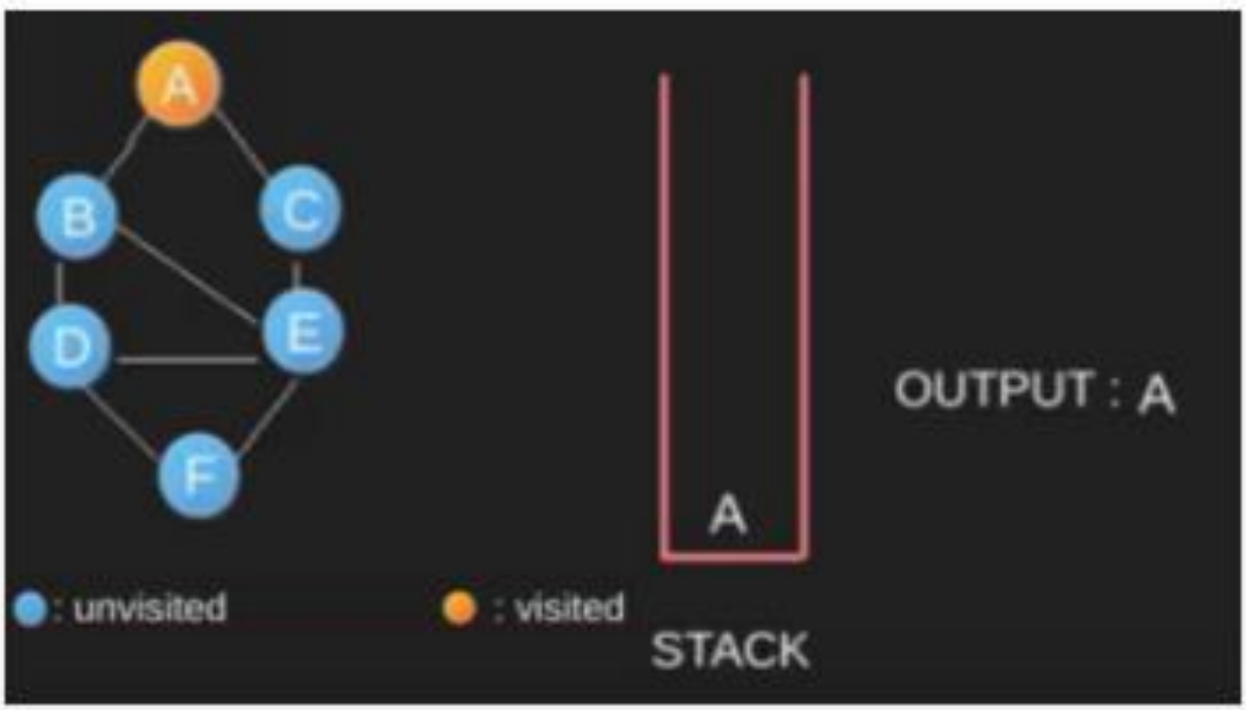


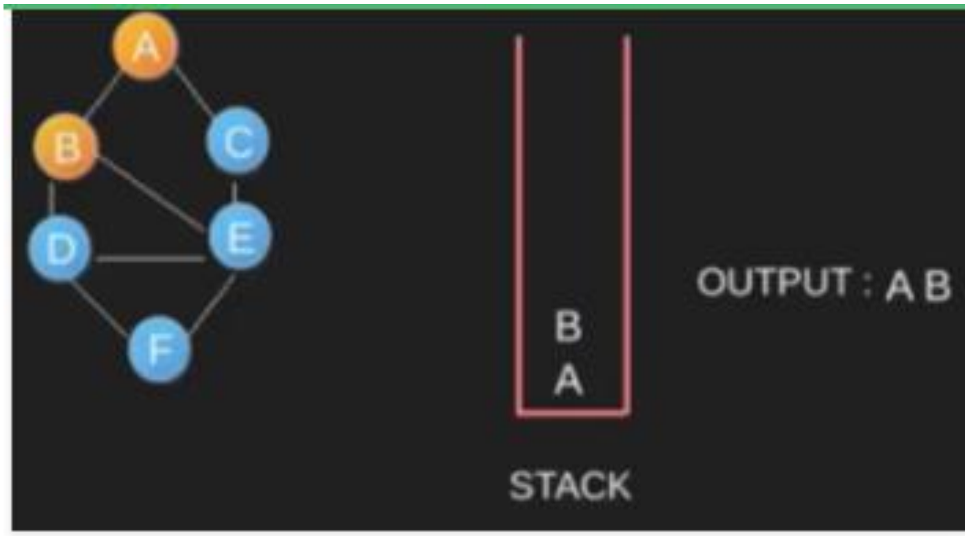
- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.

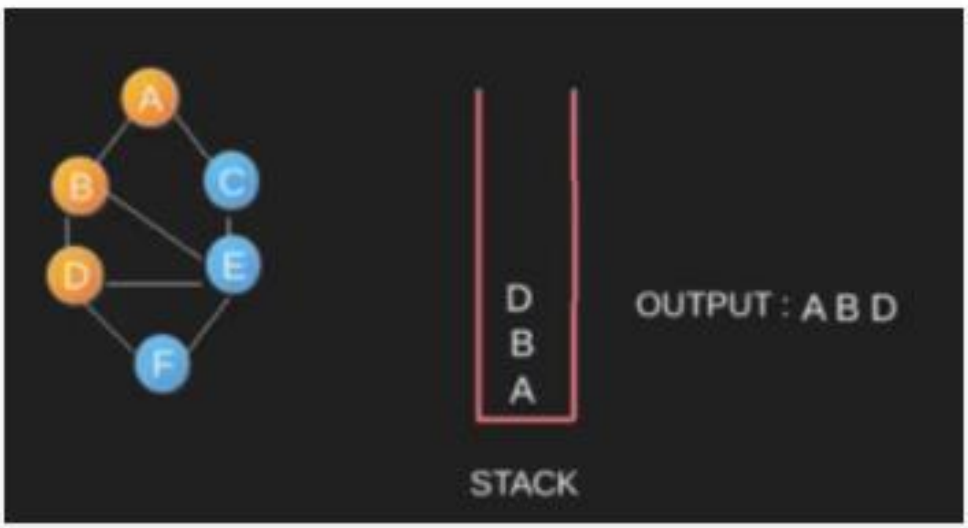


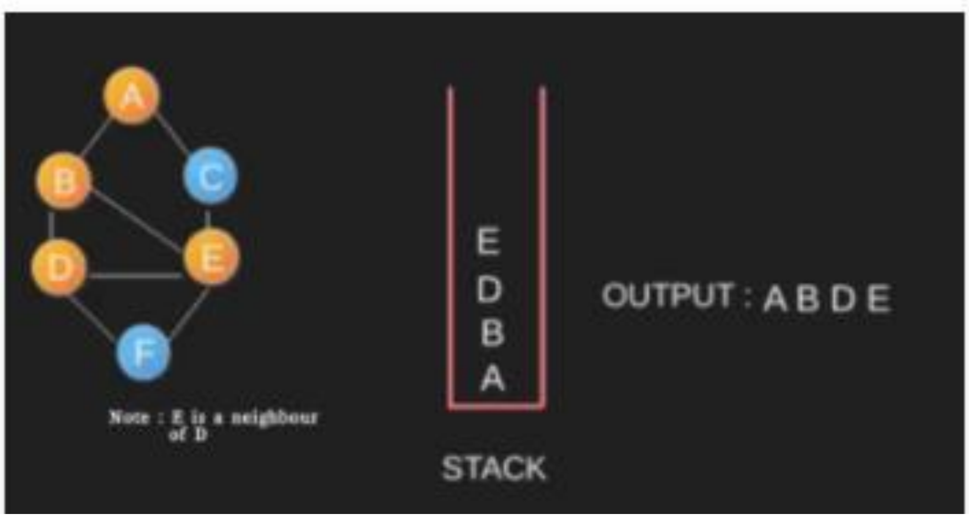
EXAMPLE 2

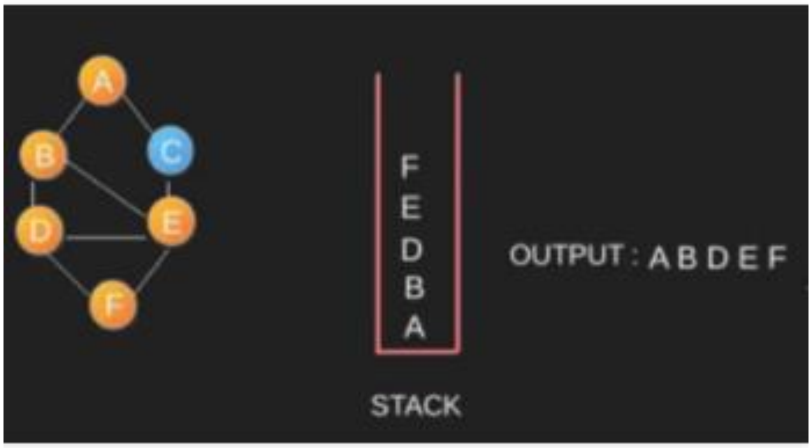

A
B
C
D
E
F
OUTPUT :
STACK

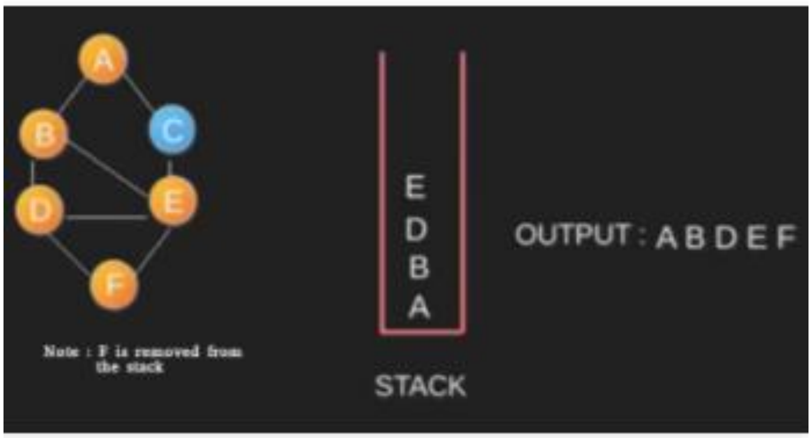

A
B
C
D
E
F
OUTPUT : A
A
: unvisited
: visited
STACK

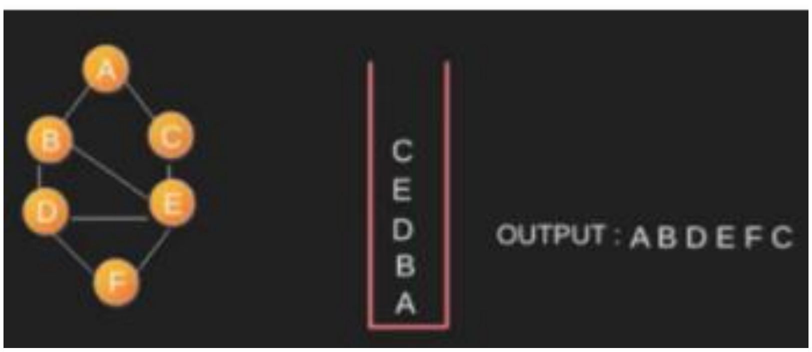

A
B
C
D
E
F
B
A
STACK
OUTPUT : A B

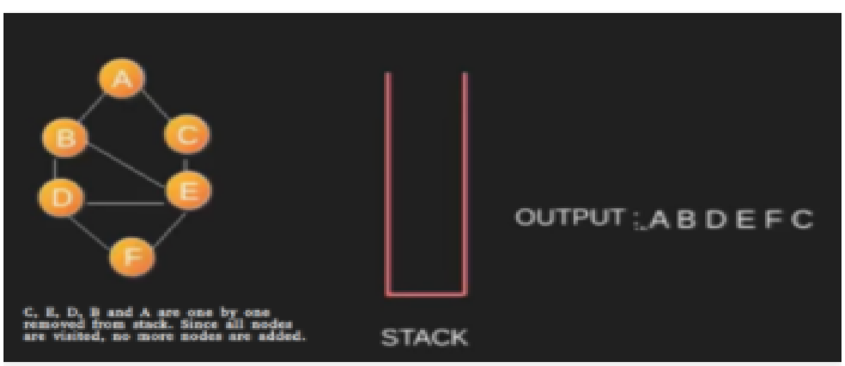

A
B
C
D
E
F
D
B
A
STACK
OUTPUT : A B D


A
B
C
D
E
F
Note : E is a neighbour of D
E
D
B
A
STACK
OUTPUT : A B D E


A
B
C
D
E
F
F
E
D
B
A
STACK
OUTPUT : A B D E F


A
B
C
D
E
F
Note : F is removed from the stack
E
D
B
A
STACK
OUTPUT : A B D E F


A
B
C
D
E
F
C
E
D
B
A
OUTPUT : A B D E F C


A
B
C
D
E
F
C, E, D, B and A are one by one removed from stack. Since all nodes are visited, no more nodes are added.
STACK
OUTPUT : A B D E F C

BFS (Breadth First Search)

BFS traversal of a graph, produces a spanning tree as final result. Spanning Tree is a graph without any loops. We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal of a graph.

We use the following steps to implement BFS traversal...

Step 1: Define a Queue of size total number of vertices in the graph.

Step 2: Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.

Step 3: Visit all the adjacent vertices of the verex which is at front of the Queue which is not visited and insert them into the Queue.

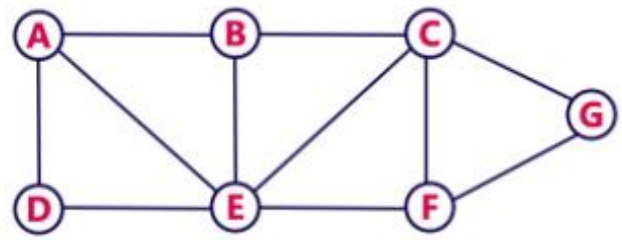
Step 4: When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.

Step 5: Repeat step 3 and 4 until queue becomes empty.

Step 6: When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph

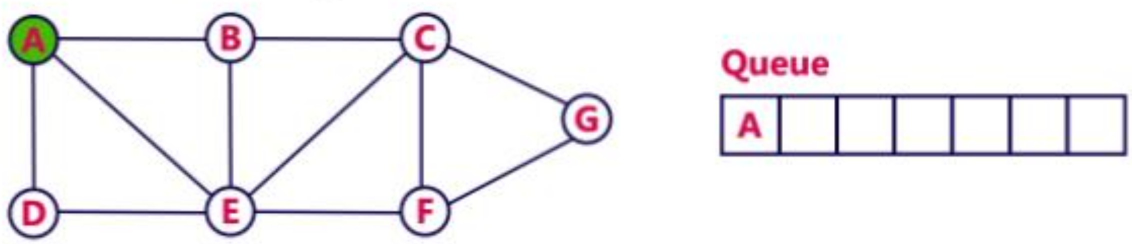
EXAMPLE

Consider the following example graph to perform BFS traversal



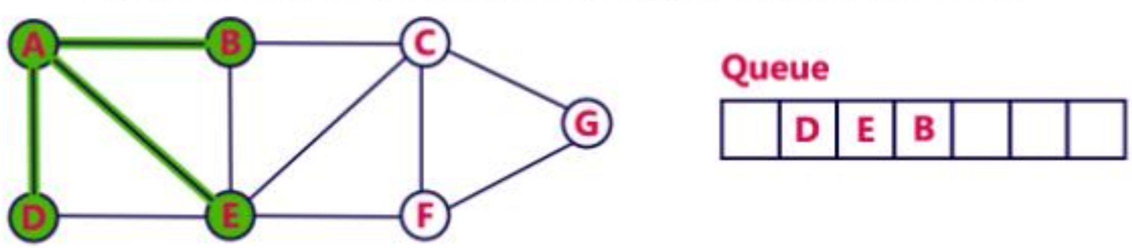
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.



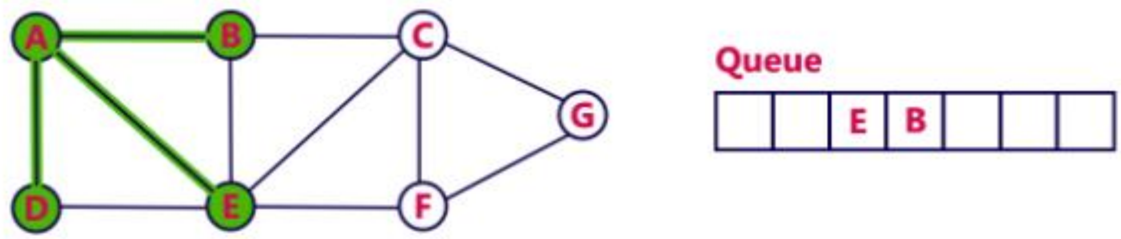
Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..



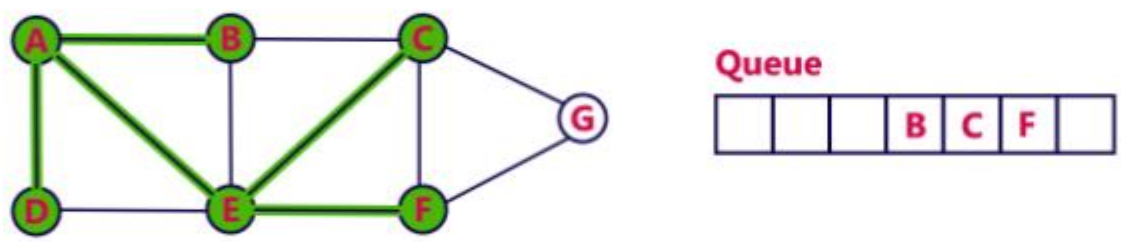
Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.



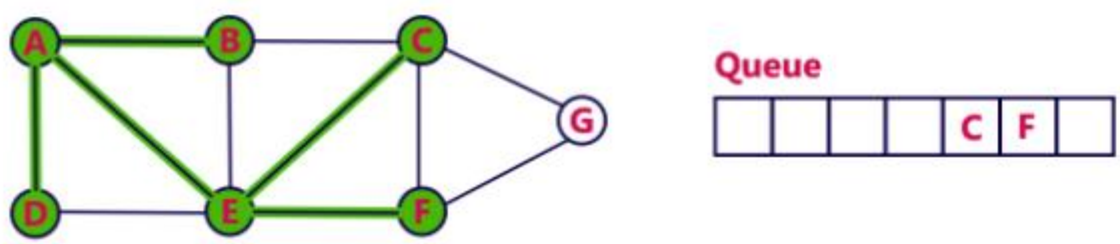
Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.



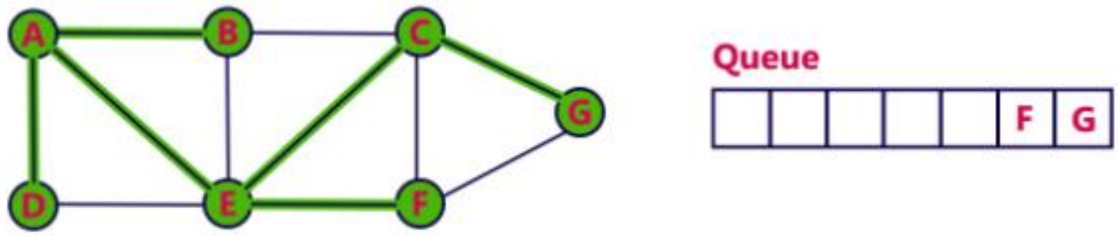
Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.



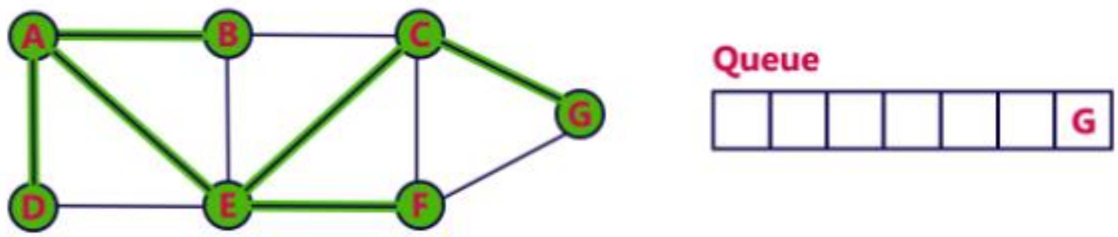
Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.



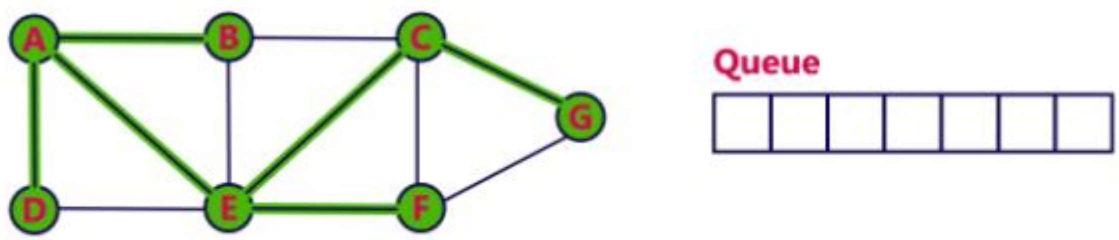
Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

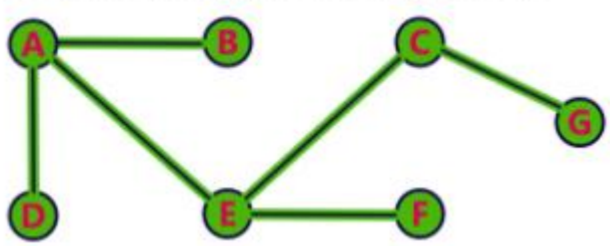


Step 8:

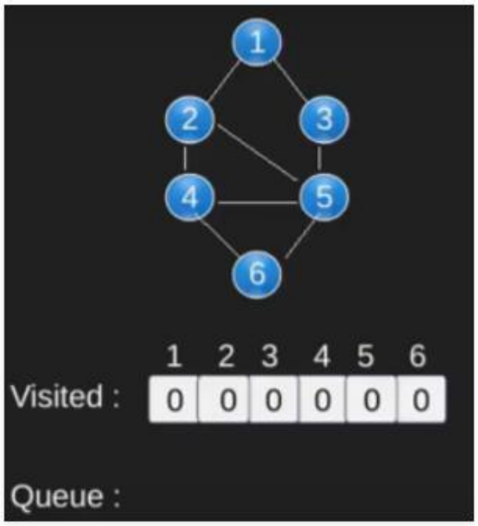
- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.

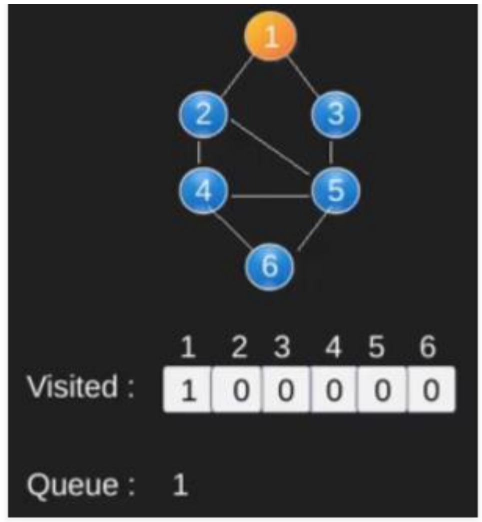


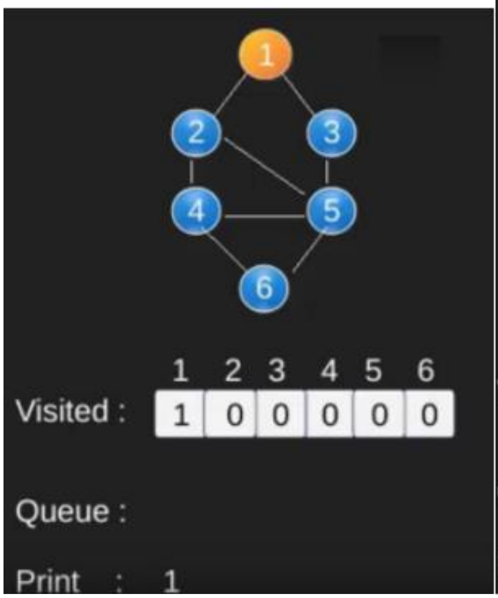
- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...

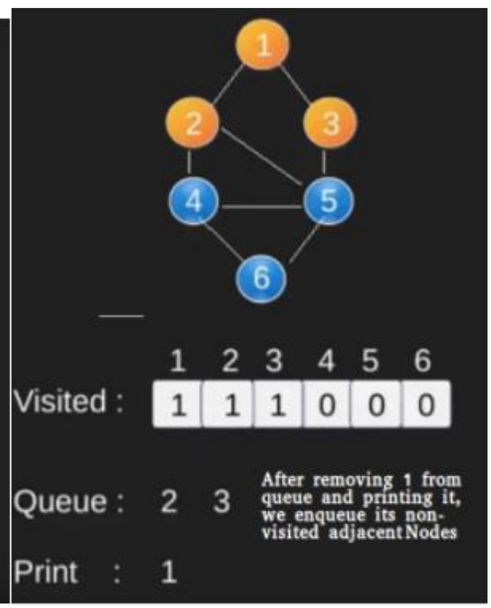


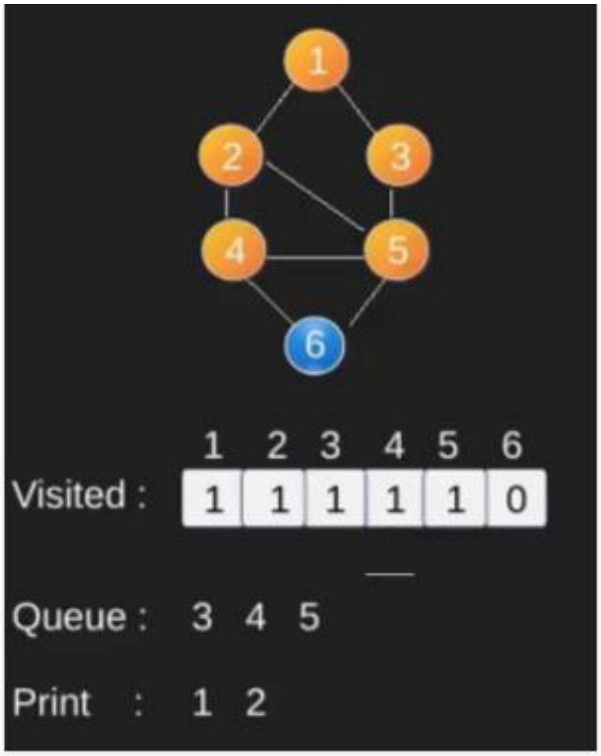
EXAMPLE 2

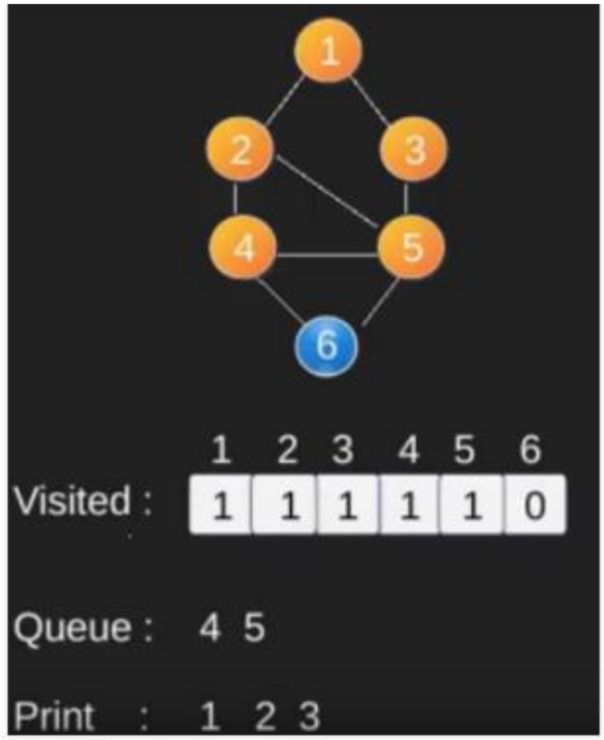

1
2
3
4
5
6
1 2 3 4 5 6
Visited : 0 0 0 0 0 0
Queue :

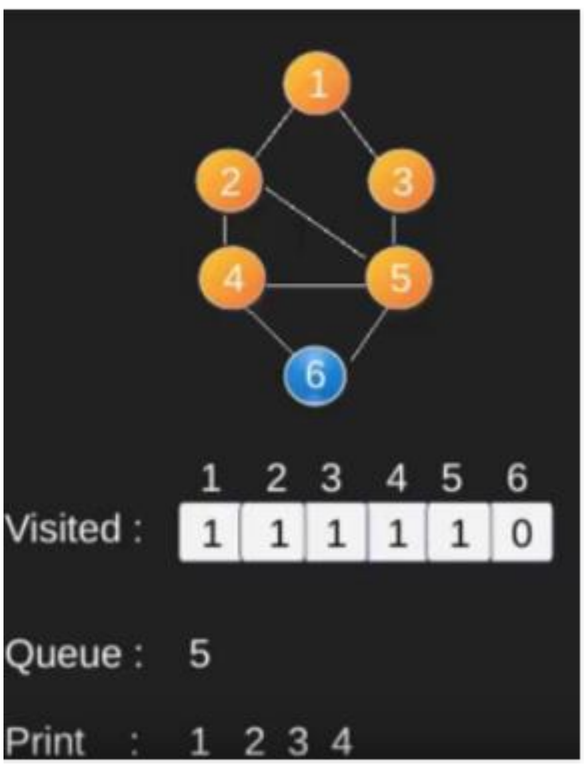

1
2
3
4
5
6
1 2 3 4 5 6
Visited : 1 0 0 0 0 0
Queue : 1

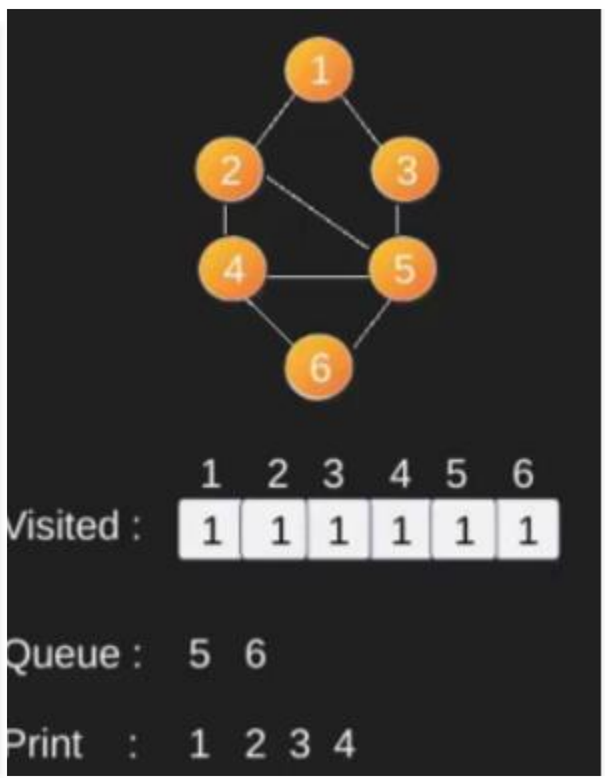

1
2
3
4
5
6
1 2 3 4 5 6
Visited : 1 0 0 0 0 0
Queue :
Print : 1

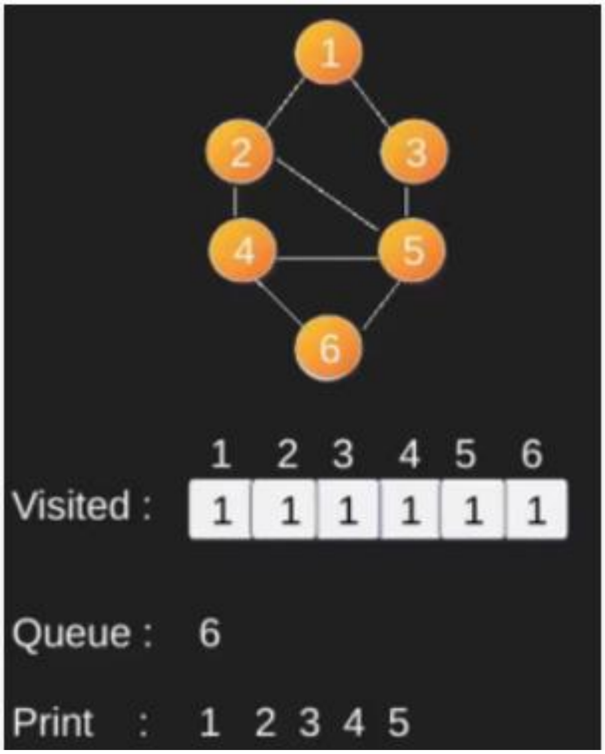

1
2
3
4
5
6
1 2 3 4 5 6
Visited : 1 1 1 0 0 0
Queue : 2 3
After removing 1 from queue and printing it, we enqueue its non-visited adjacent Nodes
Print : 1

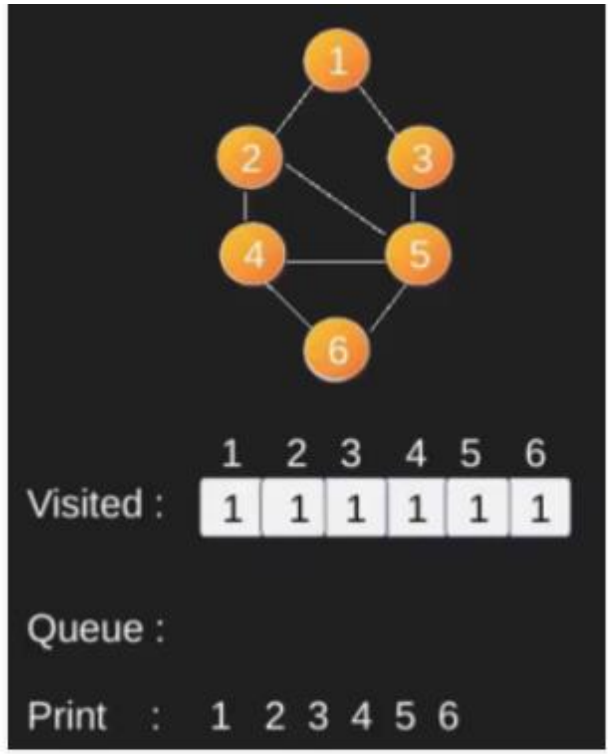

1 2 3 4 5 6
Visited : 1 1 1 1 1 0
Queue : 3 4 5
Print : 1 2


1 2 3 4 5 6
Visited : 1 1 1 1 1 0
Queue : 4 5
Print : 1 2 3


1 2 3 4 5 6
Visited : 1 1 1 1 1 0
Queue : 5
Print : 1 2 3 4


1 2 3 4 5 6
Visited : 1 1 1 1 1 1
Queue : 5 6
Print : 1 2 3 4





COMPLEXITIES OF DFS AND BFS

BFS:

Time complexity is O(|V|) where |V| is the number of nodes,you need to traverse all nodes.

Space complecity is O(|V|) as well - since at worst case you need to hold all vertices in the queue.

DFS:

Time complexity is again O(|V|), you need to traverse all nodes.

Space complexity - depends on the implementation, a recursive implementation can have a O(h) space complexity [worst case], where h is the maximal depth of your tree.

Using an iterative solution with a stack is actually the same as BFS, just using a stack instead of a queue - so you get both O(|V|) time and space complexity.

(*) Note that the space complexity and time complexity is a bit different for a tree then for a general graphs becase you do not need to maintain a visited for a tree, and |E| = O(|V|), so the |E| factor is actually redundant.

MINIMUM COST SPANNING TREES

What is a Spanning Tree?

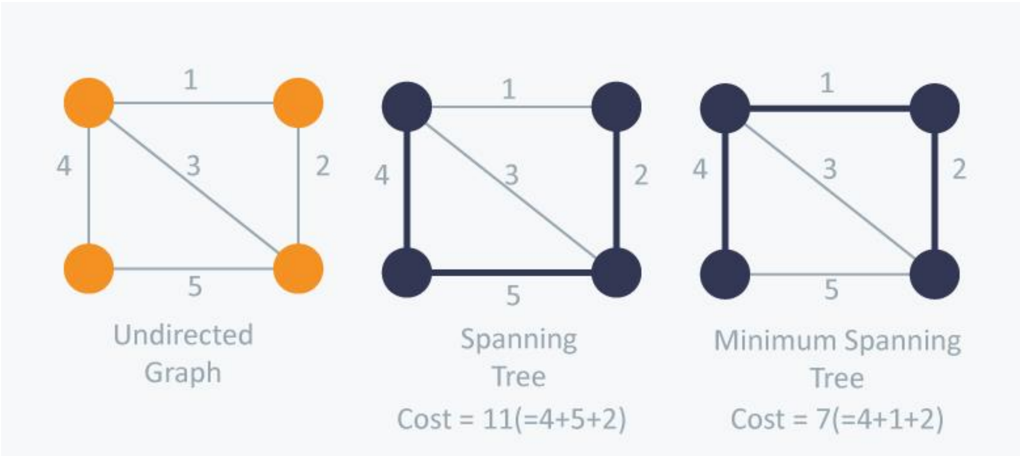
Given an undirected and connected graph G=(V,E), a spanning tree of the graph G is a tree that spans G(that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G)

Minimum Spanning Tree

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.

Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Other practical applications are:

1. Cluster Analysis
2. Handwriting recognition
3. Image segmentation



There are two famous algorithms for finding the Minimum Spanning Tree:

Kruskal's Algorithm

Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

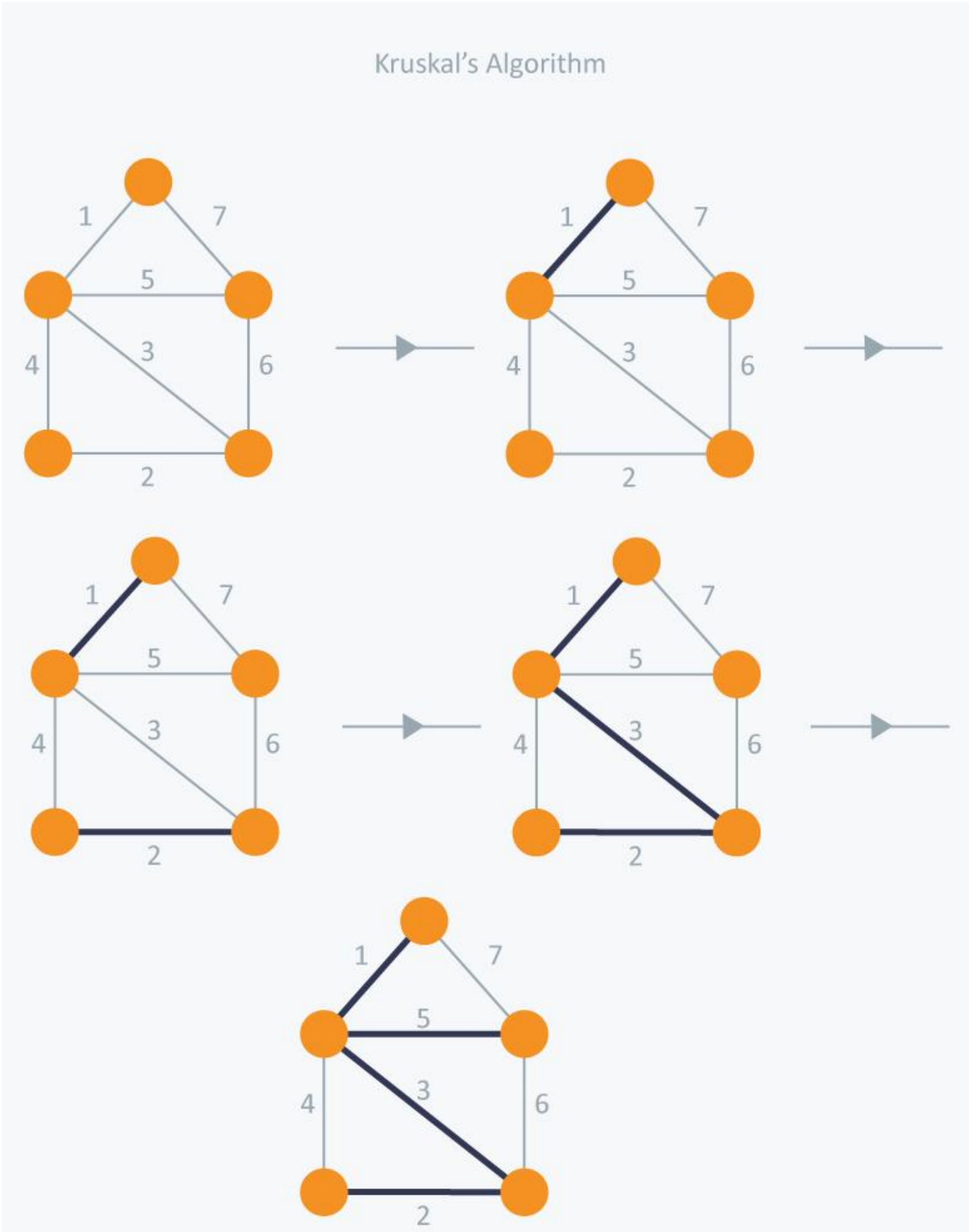
Algorithm Steps:

- Sort the graph edges with respect to their weights.
- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which doesn't form a cycle , edges which connect only disconnected components.

So now the question is how to check if 2 vertices are connected or not ?

This could be done using DFS which starts from the first vertex, then check if the second vertex is visited or not. But DFS will make time complexity large as it has an order of O(V+E) where V is the number of vertices, E is the number of edges. So the best solution is "Disjoint Sets":
Disjoint sets are sets whose intersection is the empty set so it means that they don't have any element in common.

Consider following example:



In Kruskal's algorithm, at each iteration, we will select the edge with the lowest weight. So, we will start with the lowest weighted edge first i.e., the edges with weight 1. After that we will select the second lowest weighted edge i.e., edge with weight 2. Notice these two edges are totally disjoint. Now, the next edge will be the third lowest weighted edge i.e., edge with weight 3, which connects the two disjoint pieces of the graph. Now, we are not allowed to pick the edge with weight 4, that will create a cycle and we can't have any cycles. So we will select the fifth lowest weighted edge i.e., edge with weight 5. Now the other two edges will create cycles so we will ignore them. In the end, we end up with a minimum spanning tree with total cost 11 (= 1 + 2 + 3 + 5).

TimeComplexity:
In Kruskal's algorithm, most time consuming operation is sorting because the total complexity of the Disjoint-Set operations will be O(ElogV), which is the overall Time Complexity of the algorithm.

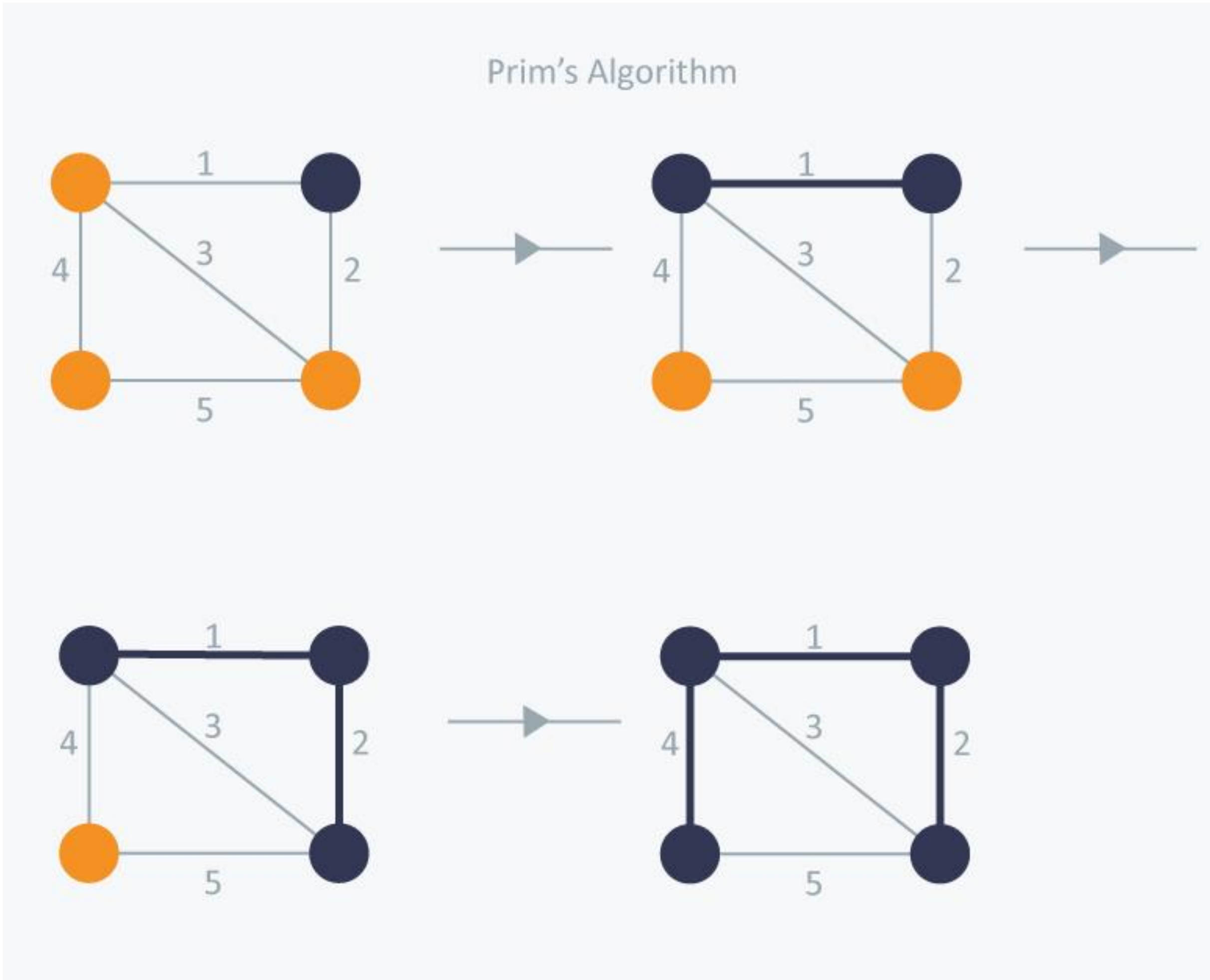
Prim's Algorithm

Prim's Algorithm also use Greedy approach to find the minimum spanning tree. In Prim's Algorithm we grow the spanning tree from a starting position. Unlike an **edge** in Kruskal's, we add **vertex** to the growing spanning tree in Prim's.

Algorithm Steps:

- Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.
- Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

Consider the example below:



In Prim's Algorithm, we will start with an arbitrary node (it doesn't matter which one) and mark it. In each iteration we will mark a new vertex that is adjacent to the one that we have already marked. As a greedy algorithm, Prim's algorithm will select the cheapest edge and mark the vertex. So we will simply choose the edge with weight 1. In the next iteration we have three options, edges with weight 2, 3 and 4. So, we will select the edge with weight 2 and mark the vertex. Now again we have three options, edges with weight 3, 4 and 5. But we can't choose edge with weight 3 as it is creating a cycle. So we will select the edge with weight 4 and we end up with the minimum spanning tree of total cost 7 (= 1 + 2 +4).

SINGLE SOURCE SHORTEST PATH ALGORITHMS

(a) DIJKSTRA'S ALGORITHM

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph G = (V, E) for the case in which all edge weights are nonnegative. In this section, therefore, we assume that $w(u, v) \geq 0$ for each edge $(u, v) \in E$. As we shall see, with a good implementation, the running time of Dijkstra's algorithm is lower than that of the Bellman-Ford algorithm. Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u. In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S ← Ø
3  Q ← V[G]
4  while Q ≠ Ø
5      do u ← EXTRACT-MIN(Q)
6         S ← S ∪ {u}
7         for each vertex v ∈ Adj[u]
8             do RELAX(u, v, w)
```

Dijkstra's algorithm relaxes edges as shown in Figure. Line 1 performs the usual initialization of d and π values, and line 2 initializes the set S to the empty set. The algorithm maintains the invariant that $Q = V - S$ at the start of each iteration of the while loop of lines 4–8. Line 3 initializes the min-priority queue Q to contain all the vertices in V; since $S = \emptyset$ at that time, the invariant is true after line 3. Each time through the while loop of lines 4–8, a vertex u is extracted from $Q = V - S$ and added to set S, thereby maintaining the invariant. (The first time through this loop, u = s.) Vertex u, therefore, has the smallest shortest-path estimate of any vertex in $V - S$. Then, lines 7–8 relax each edge (u, v) leaving u, thus updating the estimate d[v] and the predecessor π[v] if the shortest path to v can be improved bygoing through u. Observe that vertices are never insertedinto Q after line 3 and that each vertex is extracted from Q and added to S exactlyonce, so that the while loop of lines 4–8 iterates exactly |V| times.

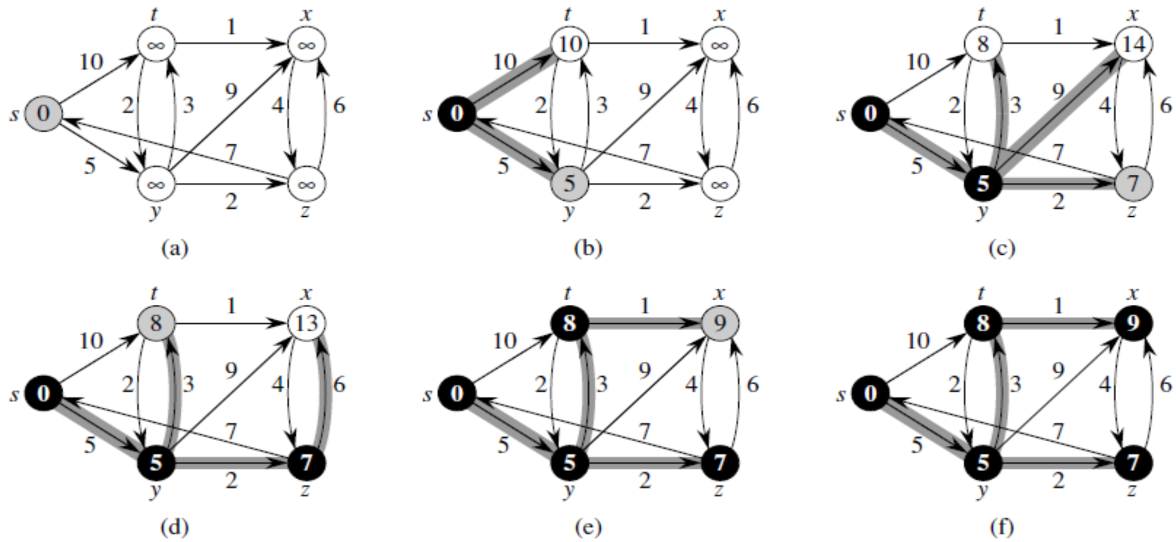


Figure: The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates are shown within the vertices, and shaded edges indicate predecessor values.Black vertices are in the set S, and white vertices are in the min-priority queue Q = V − S.(a) The situation just before the first iteration of the while loop of lines 4–8. The shaded vertex hasthe minimum d value and is chosen as vertex u in line 5. (b)–(f) The situation after each successiveiteration of the while loop. The shaded vertex in each part is chosen as vertex u in line 5 of the nextiteration. The d and π values shown in part (f) are the final values.

Because Dijkstra's algorithm always chooses the "lightest" or "closest" vertex in V − S to add to set S, we say that it uses a greedy strategy.

(b) THE BELLMAN-FORD ALGORITHM

The Bellman-Ford algorithm solves the single-source shortest-paths problem in the general case in which edge weights may be negative. Given a weighted, directed graph G = (V, E) with source s and weight function w : E → R, the Bellman-Ford algorithm returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest paths and their weights.

The algorithm uses relaxation, progressively decreasing an estimate d[v] on the weight of a shortest path from the source s to each vertex v ∈ V until it achieves the actual shortest-path weight δ(s, v). The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i ← 1 to |V[G]| − 1
3       do for each edge (u, v) ∈ E[G]
4              do RELAX(u, v, w)
5  for each edge (u, v) ∈ E[G]
6       do if d[v] > d[u] + w(u, v)
7             then return FALSE
8  return TRUE
```

Figure shows the execution of the Bellman-Ford algorithm on a graph with 5 vertices. After initializing the d and π values of all vertices in line 1, the algorithm makes |V| − 1 passes over the edges of the graph. Each pass is one iteration of the for loop of lines 2–4 and consists of relaxing each edge of the graph once. Figures (b)–(e) show the state of the algorithm after each of the four passes over the edges. After making |V|−1 passes, lines 5–8 check for a negative weight cycle and return the appropriate boolean value. (We'll see a little later why this check works.)

The Bellman-Ford algorithm runs in time O(V E), since the initialization in line 1 takes (V) time, each of the |V|−1 passes over the edges in lines 2–4 takes (E) time, and the for loop of lines 5–7 takes O(E) time.

To prove the correctness of the Bellman-Ford algorithm, we start by showing that if there are no negative-weight cycles, the algorithm computes correct shortest-path weights for all vertices reachable from the source.

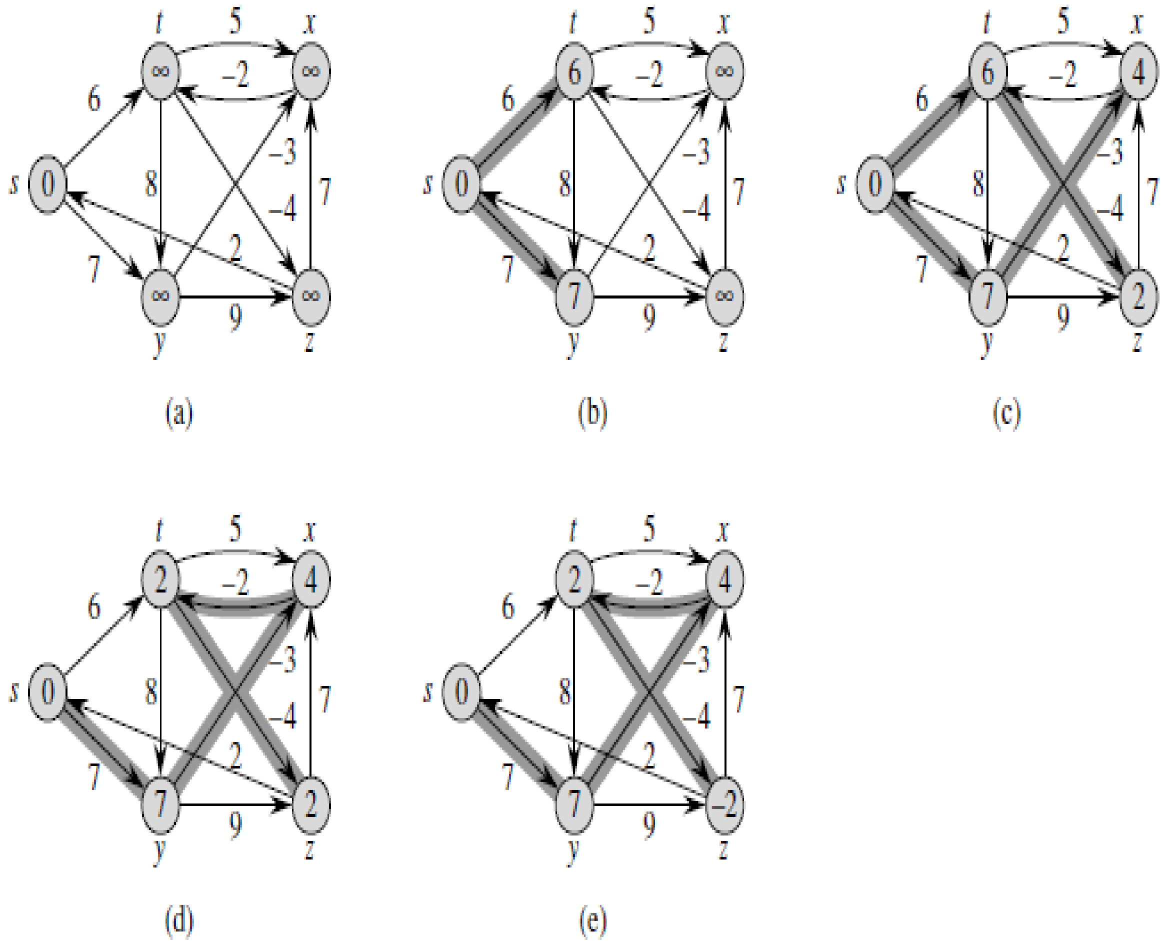
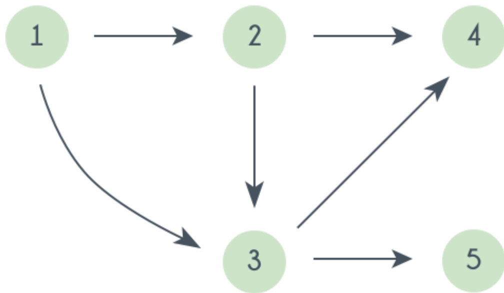


FIGURE: The execution of the Bellman-Ford algorithm. The source is vertex s. The d values are shown within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then π[v] = u. In this particular example, each pass relaxes the edges in the order (t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y). (a) The situation just before the first pass over the edges. (b)–(e) The situation after each successive pass over the edges. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

TOPOLOGICAL SORTING

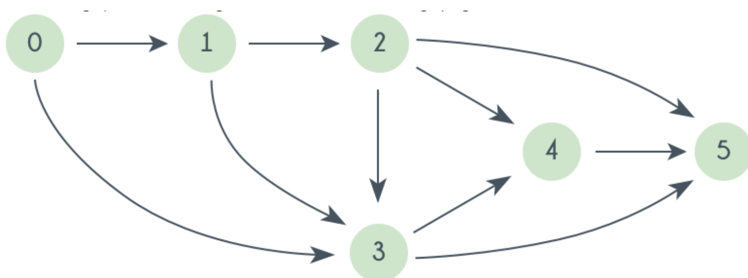
Topological sorting of vertices of a **Directed Acyclic Graph** is an ordering of the vertices v1,v2,...vn in such a way, that if there is an edge directed towards vertex vj from vertex vi, then vi comes before vj. For example consider the graph given below:

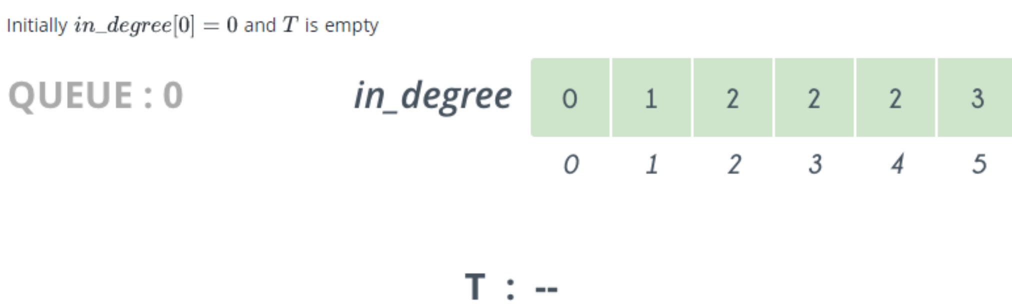


A topological sorting of this graph is: 1 2 3 4 5. There are multiple topological sorting possible for a graph. For the graph given above one another topological sorting

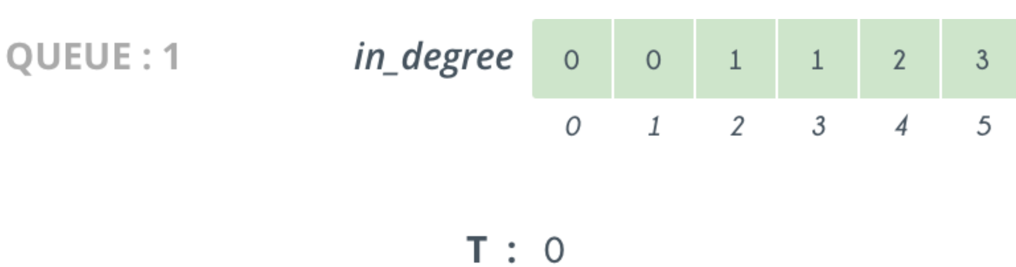
is: 1 2 3 5 4

Let's take a graph and see the algorithm in action. Consider the graph given below:

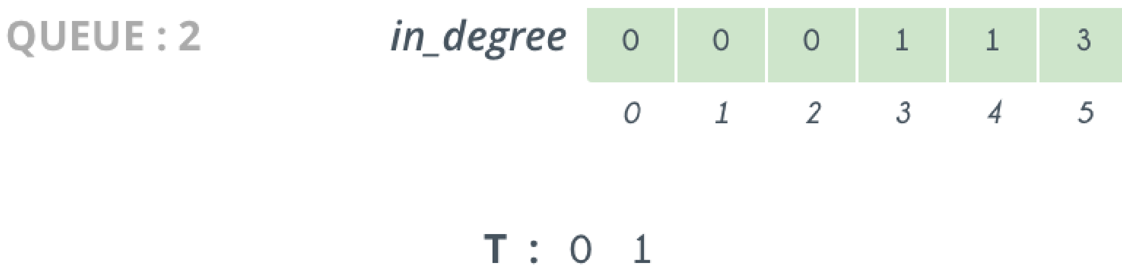




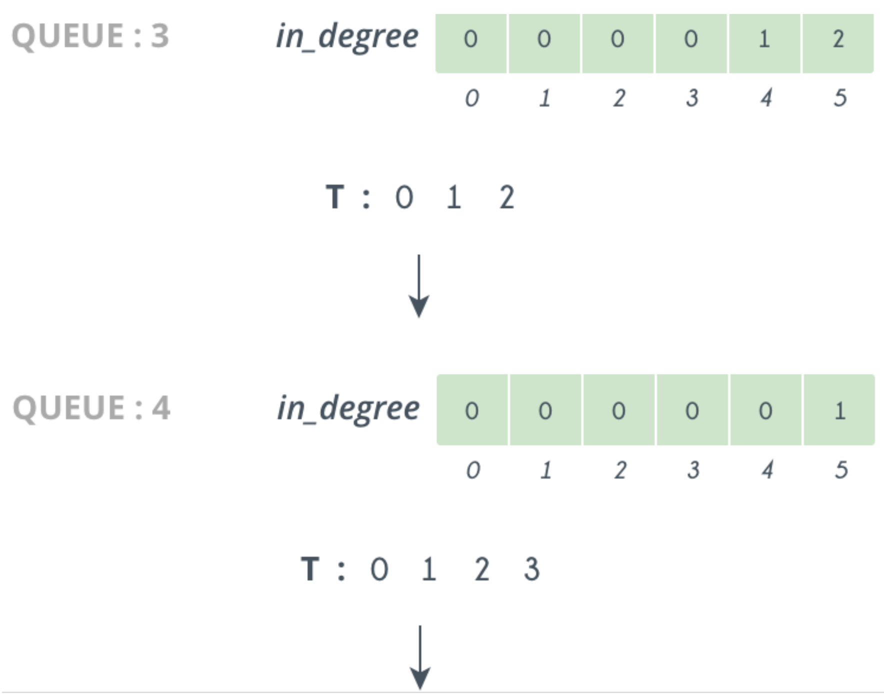
So, we delete 0 from Queue and append it to T. The vertices directly connected to 0 are 1 and 2 so we decrease their in_degree[] by 1. So, now in_degree[1]=0 and so 1 is pushed in Queue.

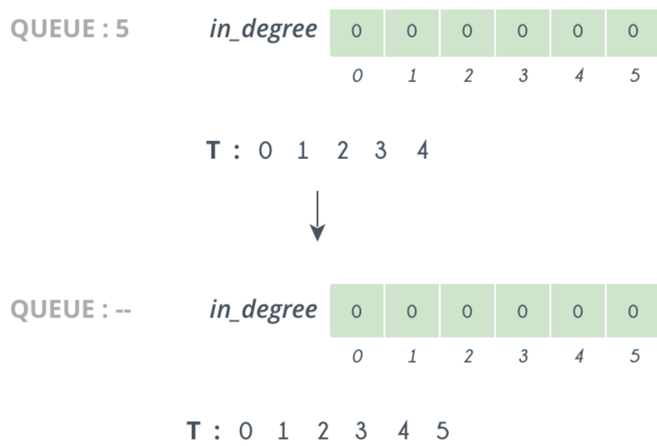


Next we delete 1 from Queue and append it to T. Doing this we decrease in_degree[2] by 1, and now it becomes 0 and 2 is pushed into Queue.



So, we continue doing like this, and further iterations looks like as follows:





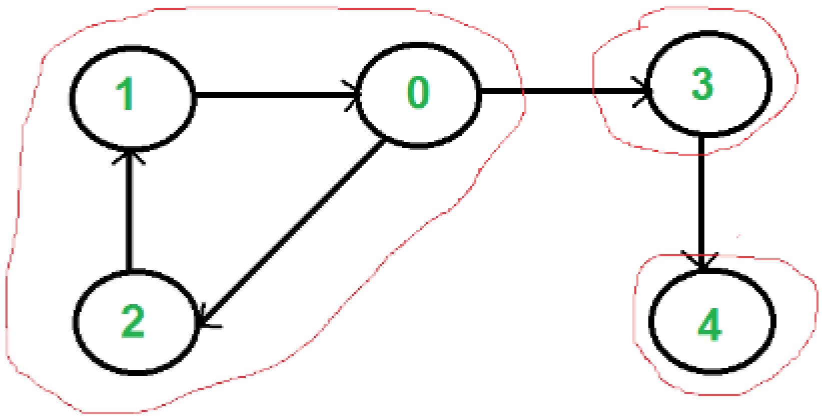
So at last we get our Topological sorting in T i.e. : 0, 1, 2, 3, 4, 5

Solution using a DFS traversal, unlike the one using BFS, does not need any special in_degree[] array. Following is the pseudo code of the DFS solution:

```
T = []

visited = []

topological_sort( cur_vert, N, adj[][] ){

    visited[cur_vert] = true

    for i = 0 to N

        if adj[cur_vert][i] is true and visited[i] is false

topological_sort(i)

T.insert_in_beginning(cur_vert)

}
```

STRONGLY CONNECTED COMPONENTS

A directed graph is called strongly connected if there is a path in each direction between each pair of vertices of the graph. In a directed graph G that may not itself be strongly connected, a pair of vertices u and v are said to be strongly connected to each other if there is a path in each direction between them.


1
0
3
2
4

MODULE IV

Divide and Conquer*:* The Control Abstraction, 2 way Merge sort, Strassen's Matrix Multiplication, Analysis
Dynamic Programming: The control Abstraction- The Optimality Principle- Optimal matrix multiplication, Bellman-Ford Algorithm.

CONTROL ABSTRACTION OF DIVIDE AND CONQUER

A control abstraction is a procedure that reflects the way an actual program based on DAndC will look like. A control abstraction shows clearly the flow of control but the primary operations are specified by other procedures. The control abstraction can be written either iteratively or recursively.

If we are given a problem with 'n' inputs and if it is possible for splitting the 'n' inputs into 'k' subsets where each subset represents a sub problem similar to the main problem then it can be achieved by using divide and conquer strategy.

If the sub problems are relatively large then divide and conquer strategy is reapplied. The sub problem resulting from divide and conquer design are of the same type as the original problem. Generally divide and conquer problem is expressed using recursive formulas and functions.

A general divide and conquer design strategy(control abstraction) is illustrated as given below-

Algorithm DAndC (P) {

if small(P) then return S(P) //termination condition

else {

Divide P into smaller instances P_1, P_2,P_3...P_k $k\geq1$; or $1\leq k\leq n$

Apply DAndC to each of these sub problems.

Return Combine (DAndC(P_1), DAndC (P_2),DAndC (P_3)...DAndC (P_k)}}

The above blocks of code represents a control abstraction for divide and conquer strategy. Small (P) is a Boolean valued function that determines whether the input size is small enough that the answer can be computed without splitting. If small (P) is true then function 'S' is invoked. Otherwise the problem 'P' is divided into sub problems. These sub problems are solved by recursive application of Divide-and-conquer. Finally the solution from k sub problems is combined to obtain the solution of the given problem.

If the size of 'P' is 'n' and if the size of 'k' sub problems is $n_{1,}n_{2,\ldots}n_k$

Respectively then the computing time of DAndC is described by the recurrence relation.

$$\begin{aligned} \mathbf{T(n)} &= \mathbf{g(n)} && \mathbf{,when\ n\ is\ small} \\ &= \mathbf{T(n_1)+ T(n_1)+ T(n_2)+\ldots\ldots+ T(n_k)+ f(n)} && \mathbf{,otherwise.} \end{aligned}$$

T(n) denotes the time for DAndC on any input of size 'n'.

G(n) is the time to compute the answer directly for small inputs.

F(n) is the time for dividing 'P' and combining the solutions of sub problems.

****

2-WAY MERGE SORT

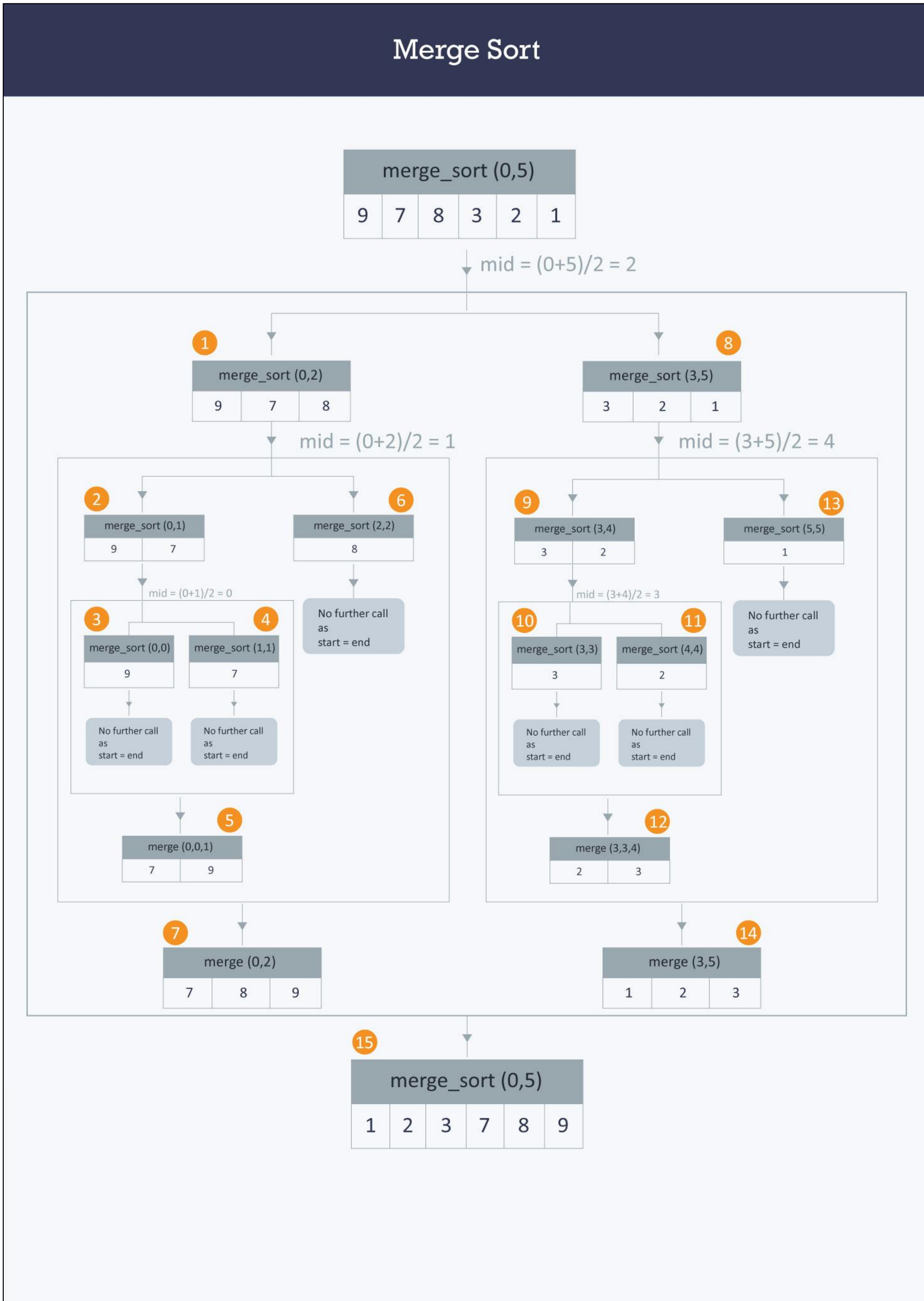
Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

Idea:

- Divide the unsorted list into N sublists, each containing 1 element.
- Take adjacent pairs of two singleton lists and merge them to form a list of 2 elements. N will now convert into N/2 lists of size 2.
- Repeat the process till a single sorted list of obtained.

While comparing two sublists for merging, the first element of both lists is taken into consideration. While sorting in ascending order, the element that is of a lesser value becomes a new element of the sorted list. This procedure is repeated until both the smaller sublists are empty and the new combined sublist comprises all the elements of both the sublists.

Let's consider the following image:


Merge Sort
merge_sort (0,5)
9 7 8 3 2 1
mid = (0+5)/2 = 2
1
merge_sort (0,2)
9 7 8
8
merge_sort (3,5)
3 2 1
mid = (0+2)/2 = 1
mid = (3+5)/2 = 4
2
merge_sort (0,1)
9 7
6
merge_sort (2,2)
8
9
merge_sort (3,4)
3 2
13
merge_sort (5,5)
1
mid = (0+1)/2 = 0
No further call as start = end
mid = (3+4)/2 = 3
No further call as start = end
3
merge_sort (0,0)
9
4
merge_sort (1,1)
7
10
merge_sort (3,3)
3
11
merge_sort (4,4)
2
No further call as start = end
No further call as start = end
No further call as start = end
No further call as start = end
5
merge (0,0,1)
7 9
12
merge (3,3,4)
2 3
7
merge (0,2)
7 8 9
14
merge (3,5)
1 2 3
15
merge_sort (0,5)
1 2 3 7 8 9

- As one may understand from the image above, at each step a list of size M is being divided into 2 sublists of size M/2, until no further division can be done. To understand better, consider a smaller array A containing the elements (9,7,8).
- At the first step this list of size 3 is divided into 2 sublists the first consisting of elements (9,7) and the second one being (8). Now, the first list consisting of elements (9,7) is further divided into 2 sublists consisting of elements (9) and (7) respectively.
- As no further breakdown of this list can be done, as each sublist consists of a maximum of 1 element, we now start to merge these lists. The 2 sub-lists formed in the last step are then merged together in sorted order using the procedure mentioned above leading to a new list (7,9). Backtracking further, we then need to merge the list consisting of element (8) too with this list, leading to the new sorted list (7,8,9).

An implementation has been provided below:

```
 void merge(int A[ ] , int start, int mid, int end) {
 //stores the starting position of both parts in temporary variables.
int p = start ,q = mid+1;

int Arr[end-start+1] , k=0;

for(int i = start ;i <= end ;i++) {
    if(p > mid)      //checks if first part comes to an end or not .
       Arr[ k++ ] = A[ q++] ;

   else if ( q > end)   //checks if second part comes to an end or not
       Arr[ k++ ] = A[ p++ ];

   else if( A[ p ] < A[ q ])     //checks which part has smaller element.
      Arr[ k++ ] = A[ p++ ];

   else
      Arr[ k++ ] = A[ q++];
 }
  for (int p=0 ; p< k ;p ++) {
   /* Now the real array has elements in sorted manner including both
        parts.*/
     A[ start++ ] = Arr[ p ] ;
  }
}
```

Here, in merge function, we will merge two parts of the arrays where one part has starting and ending positions from start to mid respectively and another part has positions from mid+1 to the end.

A beginning is made from the starting parts of both arrays. i.e. p and q. Then the respective elements of both the parts are compared and the one with the smaller value will be stored in the auxiliary array (Arr[ ]). If at some condition, one part comes to end ,then all the elements of another part of array are added in the auxiliary array in the same order they exist.

Now consider the following 2 branched recursive function:

```
    void merge_sort (int A[ ] , int start , int end )
  {
        if( start < end ) {
        int mid = (start + end ) / 2 ;          // defines the current array in 2 parts .
        merge_sort (A, start , mid ) ;                // sort the 1st part of array .
        merge_sort (A,mid+1 , end ) ;             // sort the 2nd part of array.

      // merge the both parts by comparing elements of both the parts.
       merge(A,start , mid , end );
  }
}
```

Time Complexity:

The list of size N is divided into a max of logN parts, and the merging of all sublists into a single list takes O(N) time, the worst case run time of this algorithm is O(NLogN)

****

STRASSENS ALGORITHM FOR MATRIX MULTIPLICATION

Consider two matrices A and B with 4x4 dimension each as shown below,

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

The matrix multiplication of the above two matrices A and B is Matrix C,

$$\begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}$$

where,

c11=a11∗b11+a12∗b21+a13∗b31+a14∗b41(1)
c12=a11∗b12+a12∗b22+a13∗b32+a14∗b42(2)
c21=a21∗b11+a22∗b21+a23∗b31+a24∗b41(3)
c22=a21∗b12+a22∗b22+a23∗b32+a24∗b42(4) and so on.

Now, let's look at the Divide and Conquer approach to multiply two matrices.

Take two submatrices from the above two matrices A and B each as (A11 & A12) and (B11 & B21) as shown below,

$$\overset{A}{\begin{bmatrix} \boxed{\begin{matrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{matrix}}_{A11} & \boxed{\begin{matrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{matrix}}_{A12} \\ \begin{matrix} a_{31} & a_{32} \\ a_{41} & a_{42} \end{matrix} & \begin{matrix} a_{33} & a_{34} \\ a_{43} & a_{44} \end{matrix} \end{bmatrix}} \quad \overset{B}{\begin{bmatrix} \boxed{\begin{matrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{matrix}}_{B11} & \begin{matrix} b_{13} & b_{14} \\ b_{23} & b_{24} \end{matrix} \\ \boxed{\begin{matrix} b_{31} & b_{32} \\ b_{41} & b_{42} \end{matrix}}_{B21} & \begin{matrix} b_{33} & b_{34} \\ b_{43} & b_{44} \end{matrix} \end{bmatrix}}$$

And the matrix multiplication of the two 2x2 matrices A11 and B11 is,

$$\underset{\textbf{A11}}{\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}} * \underset{\textbf{B11}}{\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}} = \begin{bmatrix} a_{11} * b_{11} + a_{12} * b_{21} & a_{11} * b_{12} + a_{12} * b_{22} \\ a_{21} * b_{11} + a_{22} * b_{21} & a_{21} * b_{12} + a_{22} * b_{22} \end{bmatrix}$$

Also, the matrix multiplication of two 2x2 matrices A12 and B21 is as follows,

$$\underset{\textbf{A12}}{\begin{bmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{bmatrix}} * \underset{\textbf{B21}}{\begin{bmatrix} b_{31} & b_{32} \\ b_{41} & b_{42} \end{bmatrix}} = \begin{bmatrix} a_{13} * b_{31} + a_{14} * b_{41} & a_{13} * b_{32} + a_{14} * b_{42} \\ a_{23} * b_{32} + a_{24} * b_{42} & a_{23} * b_{32} + a_{24} * b_{42} \end{bmatrix}$$

So if you observe, I can conclude the following,

$$A11*B11+A12*B21 = \begin{bmatrix} c_{11} & c_{12} & . & . \\ c_{21} & c_{22} & . & . \\ . & . & . & . \\ . & . & . & . \end{bmatrix}$$

Where '+' is Matrix Addition,

And c11, c12, c21 and c22 are equal to equations 1, 2, 3 and 4 respectively.

So the idea is to recursively divide n x n matrices into n/2 x n/2 matrices until they are small enough to be multiplied in the naive way, more specifically into 8 multiplications and 4 matrix additions.

Recurrence Relation of Divide and Conquer Method:

For multiplying two matrices of size n x n, we make 8 recursive calls above, each on a matrix/subproblem with size n/2 x n/2. Each of these recursive calls multiplies two n/2 x n/2 matrices, which are then added together. For addition, we add two matrices of size $n^2/4$, so each addition takes $\Theta(n^2/4)$ time.

We can write this recurrence in the form of the following equations,

$$\boxed{T(n) = \begin{cases} \Theta(1), & \text{if } n = 1 \\ 8T(\frac{n}{2}) + \Theta(n^2), & \text{if } n > 1 \end{cases}}$$

From the Case 1 of Master's Theorem, the time complexity of the above approach is $O(n^{\log_2 8})$ or $O(n^3)$.

The Advantage of using Divide and Conquer over the naive method is that we can parallelize the multiplication over different cores and/or cpu's as the 8 multiplications can be carried out independently.

Strassen's Algorithm:

Strassen's algorithm makes use of the same divide and conquer approach as above, but instead uses only 7 recursive calls rather than 8 as shown in the equations below. Here we save one recursive call, but have several new additions of n/2 x n/2 matrices.

M1=(A11+A22)(B11+B22)
M2=(A21+A22)B11
M3=A11(B12−B22)
M4=A22(B21−B−11)
M5=(A11+A12)B22
M6=(A21−A11)(B11+B12)
M7=(A12−A22)(B21+B22)

C11=M1+M4−M5+M7
C12=M3+M5
C21=M2+M4
C22=M1−M2+M3+M6

From the above equations, the recurrence relation of the Strassen's approach is,

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1 \\ 7T(\frac{n}{2}) + \Theta(n^2), & \text{if } n > 1 \end{cases}$$

So, from Case 1 of Master's Theorem, the time complexity of the above approach is $O(n^{\log_2 7})$ or $O(n^{2.81})$ which beats the divide and conquer approach asymptotically.

****

CONTROL ABSTRACTION OF DYNAMIC PROGRAMMING

Dynamic Programming is also used in optimization problems. Like divide-and-conquer method, Dynamic Programming solves problems by combining the solutions of subproblems. Moreover, Dynamic Programming algorithm solves each sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time.

Two main properties of a problem suggest that the given problem can be solved using Dynamic Programming. These properties are **overlapping sub-problems and optimal substructure**.

Overlapping Sub-Problems:

Similar to Divide-and-Conquer approach, Dynamic Programming also combines solutions to sub-problems. It is mainly used where the solution of one sub-problem is needed repeatedly. The computed solutions are stored in a table, so that these don't have to be re-computed. Hence, this technique is needed where overlapping sub-problem exists.

For example, Binary Search does not have overlapping sub-problem. Whereas recursive program of Fibonacci numbers have many overlapping sub-problems.

Optimal Sub-Structure:

A given problem has Optimal Substructure Property, if the optimal solution of the given problem can be obtained using optimal solutions of its sub-problems.

For example, the Shortest Path problem has the following optimal substructure property −

If a node **x** lies in the shortest path from a source node **u** to destination node **v**, then the shortest path from **u** to **v** is the combination of the shortest path from **u** to **x**, and the shortest path from **x** to **v**.

The standard All Pair Shortest Path algorithms like Floyd-Warshall and Bellman-Ford are typical examples of Dynamic Programming.

Steps of Dynamic Programming Approach:

Dynamic Programming algorithm is designed using the following four steps −

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution, typically in a bottom-up fashion.
- Construct an optimal solution from the computed information.

****

OPTIMALITY PRINCIPLE

The principle of optimality states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining states must constitute an optimal decision sequence with regard to the state resulting from the first decision.

MATRIX CHAIN MULTIPLICATION

Given following matrices $\{A_1,A_2,A_3,...A_n\}$ and we have to perform the matrix multiplication, which can be accomplished by a series of matrix multiplications

$A_1 \text{ x} A_2 \text{ x}, A_3 \text{ x.....x } A_n$

Matrix Multiplication operation is **associative** in nature rather commutative. By this, we mean that we have to follow the above matrix order for multiplication but we are free to **parenthesize** the above multiplication depending upon our need.

Three Matrices can be multiplied in two ways:

1. **$A_1,(A_2,A_3)$:** First multiplying(A_2 and A_3) then multiplying and resultant withA_1.
2. **$(A_1,A_2),A_3$:** First multiplying(A_1 and A_2) then multiplying and resultant withA_3

To find the best possible way to calculate the product, we could simply parenthesis the expression in every possible fashion and count each time how many scalar multiplication are required. Matrix Chain Multiplication Problem can be stated as "find the optimal parenthesization of a chain of matrices to be multiplied such that the number of scalar multiplication is minimized".

Number of ways for parenthesizing the matrices:

There are very large numbers of ways of parenthesizing these matrices. If there are n items, there are (n-1) ways in which the outer most pair of parenthesis can place.

(A_1) $(A_2,A_3,A_4,................A_n)$
Or (A_1,A_2) $(A_3,A_4\ ....A_n)$
Or (A_1,A_2,A_3) $(A_4\ ....A_n)$
....

Or$(A_1,A_2,A_3.............A_{n-1})$ (A_n)

It can be observed that after splitting the kth matrices, we are left with two parenthesized sequence of matrices: one consist 'k' matrices and another consist 'n-k' matrices.

Development of Dynamic Programming Algorithm

1. Characterize the structure of an optimal solution.

2. Define the value of an optimal solution recursively.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct the optimal solution from the computed information.

Step1: Structure of an optimal parenthesization:

Our first step in the dynamic paradigm is to find the optimal substructure and then use it to construct an optimal solution to the problem from an optimal solution to subproblems.

Let $A_{i....j}$ where $i \leq j$ denotes the matrix that results from evaluating the product $A_i\ A_{i+1}....A_j$. If $i < j$ then any parenthesization of the product $A_i\ A_{i+1}......A_j$ must split that the product between A_k and A_{k+1} for some integer k in the range $i \leq k \leq j$. That is for some value of k, we first compute the matrices $A_{i.....k}$ & $A_{k+1....j}$ and then multiply them together to produce the final product $A_{i....j}$. The cost of computing $A_{i....k}$ plus the cost of computing $A_{k+1....j}$ plus the cost of multiplying them together is the cost of parenthesization.

Step 2: A Recursive Solution: Let m [i, j] be the minimum number of scalar multiplication needed to compute the matrix$A_{i....j}$.

If i=j the chain consist of just one matrix $A_{i....i}=A_i$ so no scalar multiplication are necessary to compute the product. Thus m [i, j] = 0 for i= 1, 2, 3....n.

If i<j we assume that to optimally parenthesize the product we split it between A_k and A_{k+1} where $i\leq k \leq j$. Then m [i,j] equals the minimum cost for computing the subproducts $A_{i....k}$ and $A_{k+1....j}$+ cost of multiplying them together. We know A_i has dimension p_{i-1} x p_i, so computing the product $A_{i....k}$ and $A_{k+1....j}$takes $p_{i-1}\ p_k\ p_j$ scalar multiplication, we obtain

$$m[i,j] = m[i, k] + m[k + 1, j] + p_{i-1}\ p_k\ p_j$$

There are only (j-1) possible values for 'k' namely k = i, i+1.....j-1. Since the optimal parenthesization must use one of these values for 'k' we need only check them all to find the best.

So the minimum cost of parenthesizing the product $A_i\ A_{i+1}......A_j$ becomes

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \leq k < j}\{m[i,k] + m[k+1,j] + p_{i-1}\, p_k p_j\} & \text{if } i < j \end{cases}$$

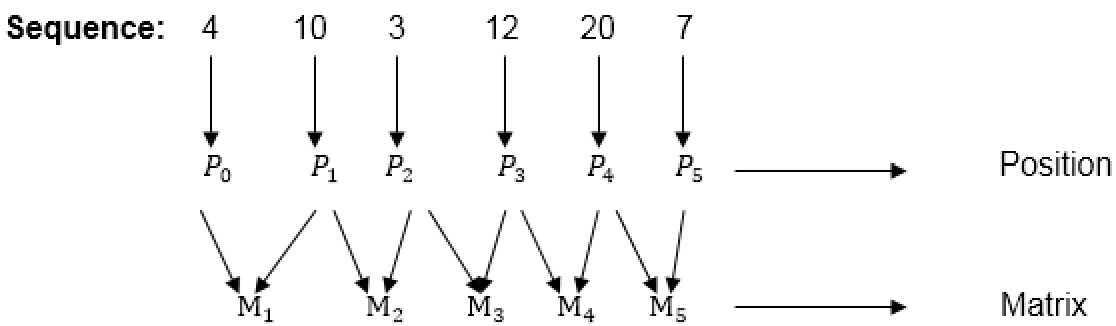
To construct an optimal solution, let us define s [i,j] to be the value of 'k' at which we can split the product $A_i\ A_{i+1}.....A_j$ To obtain an optimal parenthesization i.e. s [i, j] = k such that

$$m[i,j] = m[i, k] + m[k + 1, j] + p_{i-1}\ p_k\ p_j$$

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], $0 \le i, j \le 5$. 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 to p_{i-1})

On the basis of sequence, we make a formula

For M_i $\longrightarrow$ 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:

1. m (1,2) = m_1 x m_2

= 4 x 10 x 10 x 3

= 4 x 10 x 3 = 120

2. m (2, 3) = m_2 x m_3

= 10 x 3 x 3 x 12

= 10 x 3 x 12 = 360

3. m (3, 4) = m_3 x m_4

= 3 x 12 x 12 x 20

= 3 x 12 x 20 = 720

4. m (4,5) = m_4 x m_5

= 12 x 20 x 20 x 7

= 12 x 20 x 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 x M_2) + M_3, M_1+ (M_2x M_3)
2. After solving both cases we choose the case in which minimum output is there.

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

M [1, 3] =264

As Comparing both output **264** is minimum in both cases so we insert **264** in table and (M_1 x 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_2x M_3)+M_4, M_2+(M_3 x M_4)
2. After solving both cases we choose the case in which minimum output is there.

$$\mathrm{M}\ [2,\ 4] = \min \begin{Bmatrix} M[2,3] + M[4,4] + \ p_1p_3p_4 = 360 + 0 + 10.12.20 = 2760 \\ M[2,2] + \ M[3,4] + \ p_1p_2p_4 = 0 + 720 + 10.3.20 = \ 1320 \end{Bmatrix}$$

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 x 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 x M_4) + M_5, M_3+ (M_4xM_5)
2. After solving both cases we choose the case in which minimum output is there.

$$\mathrm{M}\ [3,\ 5] = \min \begin{Bmatrix} M[3,4] + M[5,5] + p_2p_4p_5 = 720 + 0 + 3\ .20.7 \ = \ \ 1140 \\ M[3,3] + \ M\ [4,5] + \ p_2p_3p_5 = 0 + 1680 + 3.12.7 = 1932 \end{Bmatrix}$$

M [3, 5] = 1140

As Comparing both output **1140** is minimum in both cases so we insert **1140** in table and (M_3 x M_4) + M_5this 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

$\longrightarrow$

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

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

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

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

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

→

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

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 x M_2 xM_3 x M_4)x M_5
2. M_1 x(M_2 xM_3 x M_4 xM_5)
3. (M_1 x M_2 xM_3)x M_4 xM_5
4. M_1 x M_2x(M_3 x M_4 xM_5)

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

$$M\,[1,\,5] = \min \begin{Bmatrix} M[1,4] + M[5,5] + p_0p_4p_5 = 1080 + 0 + 4.20.7 = \quad 1544 \\ M[1,3] + M[4,5] + p_0p_3p_5 = 264 + 1680 + 4.12.7 = 2016 \\ M[1,2] + M[3,5] + p_0p_2p_5 = 120 + 1140 + 4.3.7 = \ 1344 \\ M[1,1] + M[2,5] + \ p_0p_1p_5 = 0 + 1350 + 4.10.7 = \quad 1630 \end{Bmatrix}$$

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 x M_2 x(M_3 x M_4 x 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

$\longrightarrow$

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}x p_i for i=1, 2, 3....n. The input is a sequence (p_0,p_1,......p_n) where length [p] = n+1. The procedure uses an auxiliary table m [1....n, 1.....n] for storing m [i, j] costs an auxiliary table s [1.....n, 1.....n] that record which index of k achieved the optimal costs in computing m [i, j].

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

Algorithm of Matrix Chain Multiplication

Step 1: Constructing an Optimal Solution:

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 ")"

****

THE BELLMAN-FORD ALGORITHM

The Bellman-Ford algorithm solves the single-source shortest-paths problem in the general case in which edge weights may be negative. Given a weighted, directed graph G = (V, E) with source s and weight function w : E → R, the Bellman-Ford algorithm returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest paths and their weights.

The algorithm uses relaxation, progressively decreasing an estimate d[v] on the weight of a shortest path from the source s to each vertex v ∈ V until it achieves the actual shortest-path weight δ(s, v). The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i ← 1 to |V[G]| − 1
3      do for each edge (u, v) ∈ E[G]
4             do RELAX(u, v, w)
5  for each edge (u, v) ∈ E[G]
6      do if d[v] > d[u] + w(u, v)
7            then return FALSE
8  return TRUE
```

Figure shows the execution of the Bellman-Ford algorithm on a graph with 5 vertices. After initializing the d and π values of all vertices in line 1, the algorithm makes |V| − 1 passes over the edges of the graph. Each pass is one iteration of the for loop of lines 2–4 and consists of relaxing each edge of the graph once. Figures (b)–(e) show the state of the algorithm after each of the four passes over the edges. After making |V|−1 passes, lines 5–8 check for a negative weight cycle and return the appropriate boolean value. (We'll see a little later why this check works.)

The Bellman-Ford algorithm runs in time O(V E), since the initialization in line 1 takes (V) time, each of the |V|−1 passes over the edges in lines 2–4 takes (E) time, and the for loop of lines 5–7 takes O(E) time.

To prove the correctness of the Bellman-Ford algorithm, we start by showing that if there are no negative-weight cycles, the algorithm computes correct shortest-path weights for all vertices reachable from the source.

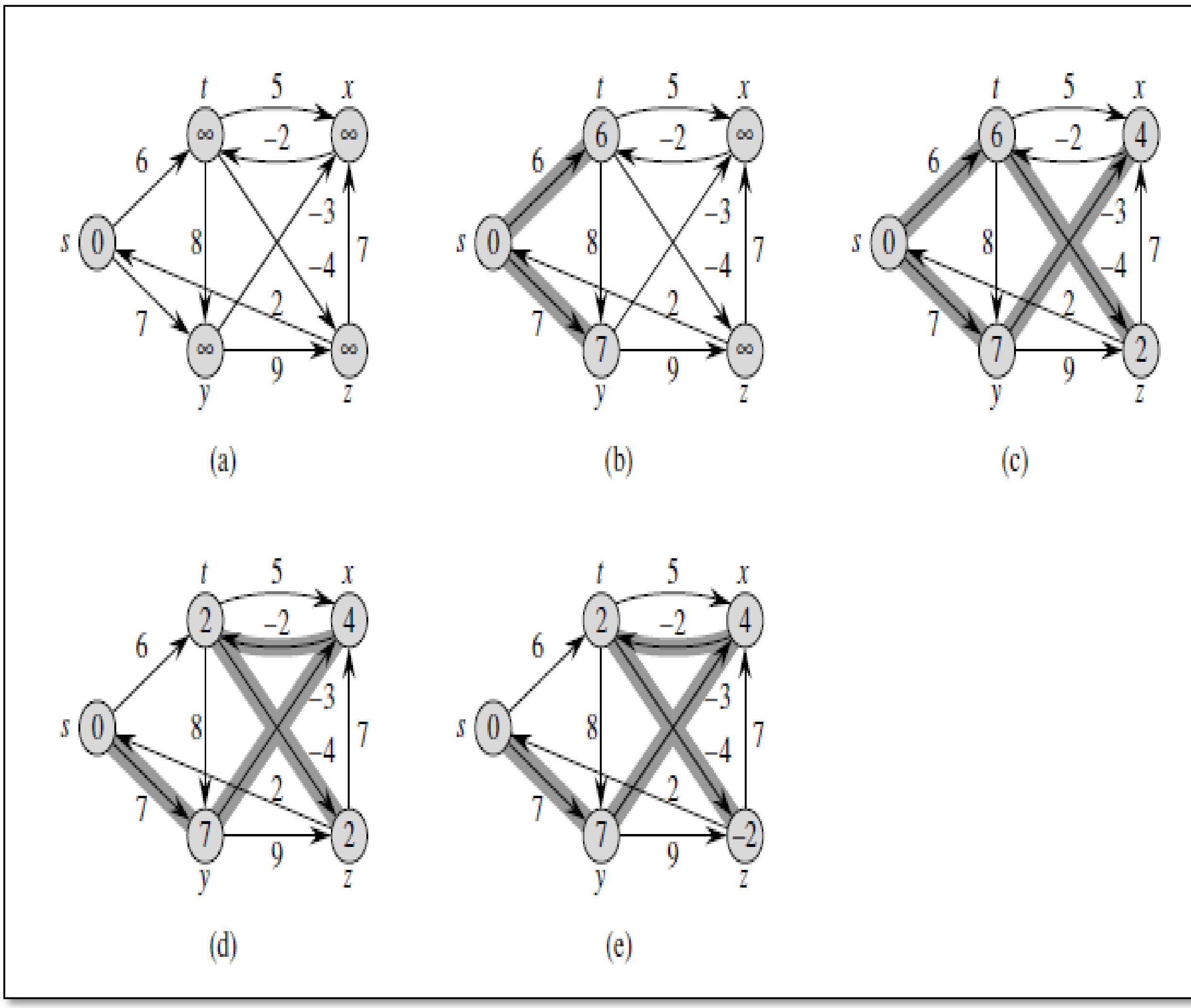


FIGURE: The execution of the Bellman-Ford algorithm. The source is vertex s. The d values are shown within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then π[v] = u. In this particular example, each pass relaxes the edges in the order (t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y). (a) The situation just before the first pass over the edges. (b)–(e) The situation after each successive pass over the edges. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

MODULE V

Analysis, Comparison of Divide and Conquer and Dynamic Programming strategies Greedy Strategy: - The Control Abstraction- the Fractional Knapsack Problem, Minimal Cost Spanning Tree Computation- Prim's Algorithm – Kruskal's Algorithm.

COMPARISON OF DIVIDE AND CONQUER AND DYNAMIC PROGRAMMING STRATEGIES

The **main difference** between divide and conquer and dynamic programming is that the **divide and conquer combines the solutions of the sub-problems to obtain the solution of the main problem while dynamic programming uses the result of the sub-problems to find the optimum solution of the main problem.** Divide and conquer and dynamic programming are two algorithms or approaches to solving problems. Divide and conquer algorithm divides the problem into subproblems and combines those solutions to find the solution to the original problem. However, dynamic programming does not solve the subproblems independently. It stores the answers of subproblems to use them for similar problems.

DIVIDE AND CONQUER	DYNAMIC PROGRAMMING
An algorithm that recursively breaks down a problem into two or more sub-problems of the same or related type until it becomes simple enough to be solved directly	An algorithm that helps to efficiently solve a class of problems that have overlapping subproblems and optimal substructure property
Subproblems are independent of each other	Subproblems are interdependent
Recursive	Non-recursive
More time-consuming as it solves each subproblem independently	Less time-consuming as it uses the answers of the previous subproblems
Less efficient	More efficient
Used by merge sort, quicksort, and binary search	Used by matrix chain multiplication, optimal binary search tree

GREEDY STRATEGY

Among all the algorithmic approaches, the simplest and straightforward approach is the Greedy method. In this approach, the decision is taken on the basis of current available information without worrying about the effect of the current decision in future.

A greedy algorithm, as the name suggests, **always makes the choice that seems to be the best at that moment**. This means that it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution. Greedy algorithms are quite successful in some problems, such as Huffman encoding which is used to compress data, or Dijkstra's algorithm, which is used to find the shortest path through a graph.

The Control Abstraction:

```
Algorithm  Greedy(a, n)
// a[1:n] contains the n inputs
{
   solution=  //Initialize solution
   for i=1 to n do
   {
        x:=Select(a);
        if  Feasible(solution, x) then
                solution=Union(solution, x)
   }
return solution;
}
```

- Selects an input from a[] and removes it.
- The selected input's value is assigned to **x**.
- **Feasible** is a Boolean-valued function that determines whether **x** can be included into the **solution** vector or not.
- **Union** combines **x** with the **solution** and updates the objective function.

KNAPSACK PROBLEM

A list of items is given, each item has its own value and weight. Items can be placed in a knapsack whose maximum weight limit is W. The problem is to find the weight that is less than or equal to W, and value is maximized.

There are two types of Knapsack problem.

- 0 – 1 Knapsack
- Fractional Knapsack

0 – 1 Knapsack

In 0-1 Knapsack you can either put the item or discard it, there is no concept of putting some part of item in the knapsack.

Example:

```
Items = {A, B, C}
Value of items = {20, 25, 40}
Weights of items = {25, 20, 30}
Capacity of the bag = 50
```

The Maximum capacity of the bag is 50. So, we can choose only items **B and C**.

Weight of B=20 and weight of C=30. So, Total weight = 20+30=50. Total Value = 25+40=65.

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 $\mathbf{i^{th}}$ item $w_i>0$
- Profit for $\mathbf{i^{th}}$ item $p_i>0$and
- Capacity of the Knapsack is **W**

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 (P_i / W_i)	7	10	6	5

As the provided items are not sorted based on (P_i / W_i). 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 (P_i / W_i)	10	7	6	5

Solution

- After sorting all the items according to (P_i / W_i), 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.

MINIMAL COST SPANNING TREE COMPUTATION

What is a Spanning Tree?

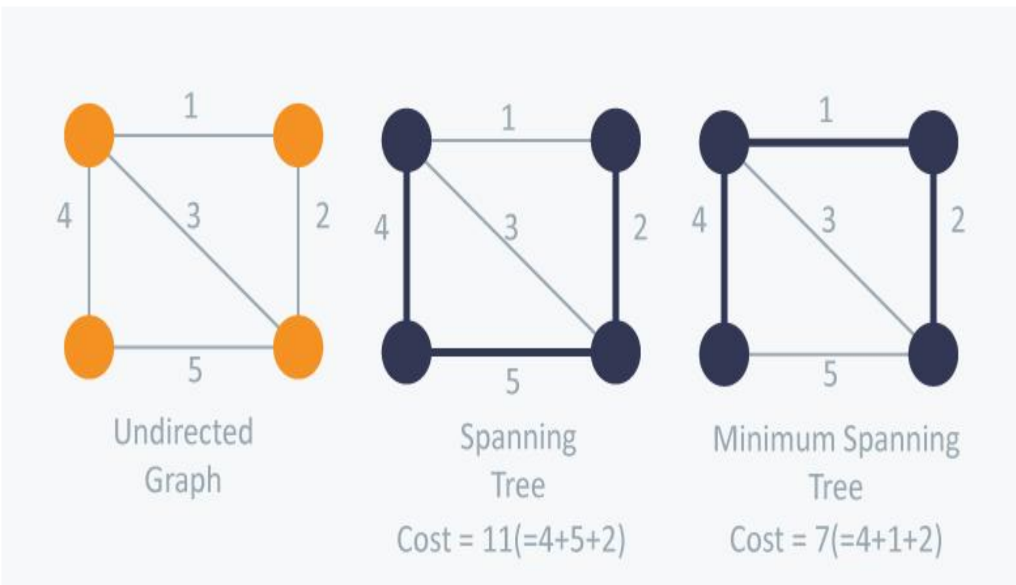
Given an undirected and connected graph G=(V,E), a spanning tree of the graph G is a tree that spans G(that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G)

Minimum Spanning Tree

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.

Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Other practical applications are:

1. Cluster Analysis
2. Handwriting recognition
3. Image segmentation



There are two famous algorithms for finding the Minimum Spanning Tree:

Kruskal’s Algorithm

Kruskal’s Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

Algorithm Steps:

- Sort the graph edges with respect to their weights.
- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which doesn't form a cycle , edges which connect only disconnected components.

So now the question is how to check if 2 vertices are connected or not ?

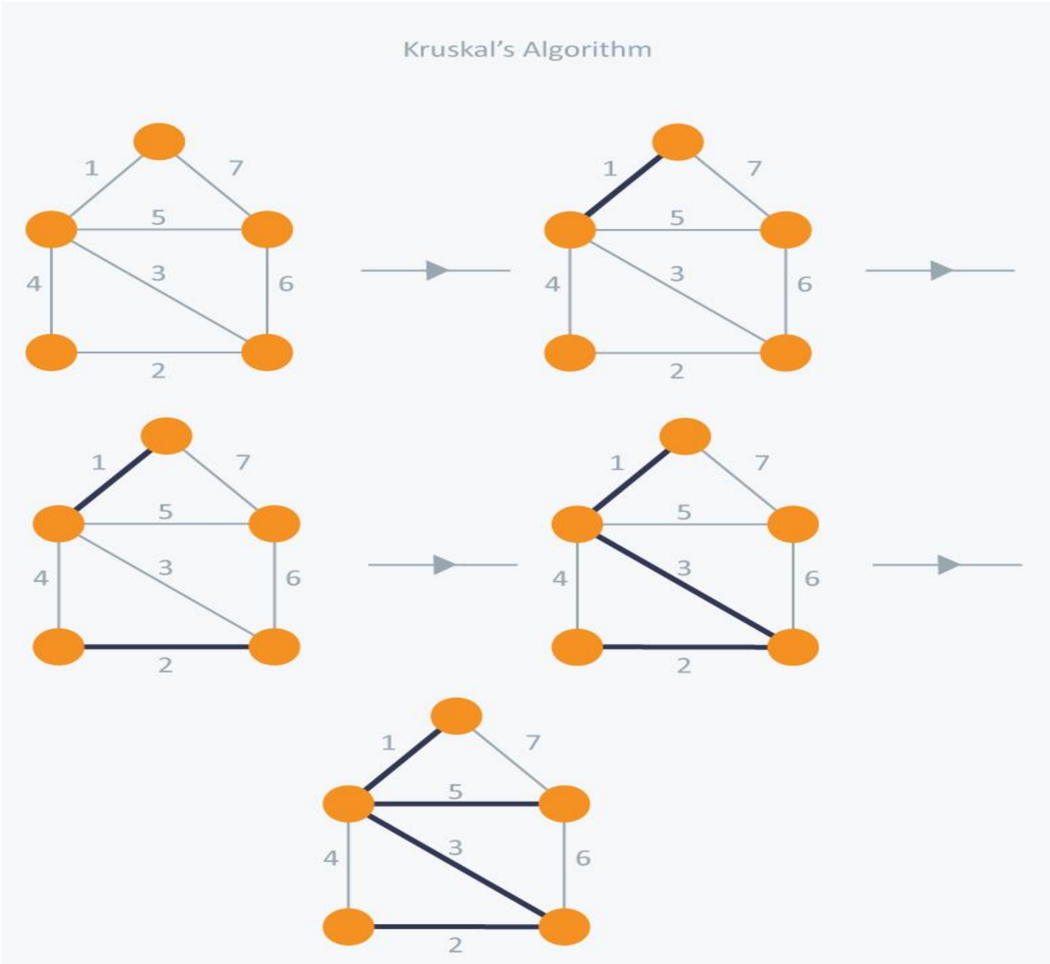
This could be done using DFS which starts from the first vertex, then check if the second vertex is visited or not. But DFS will make time complexity large as it has an order of O(V+E) where V is the number of vertices, E is the number of edges. So the best solution is "Disjoint Sets":
Disjoint sets are sets whose intersection is the empty set so it means that they don't have any element in common.

Consider following example:

In Kruskal’s algorithm, at each iteration, we will select the edge with the lowest weight. So, we will start with the lowest weighted edge first i.e., the edges with weight 1. After that we will select the second lowest weighted edge i.e., edge with weight 2. Notice these two edges are totally disjoint. Now, the next edge will be the third lowest weighted edge i.e., edge with weight 3, which connects the two disjoint pieces of the graph.

Now, we are not allowed to pick the edge with weight 4, that will create a cycle and we can’t have any cycles. So we will select the fifth lowest weighted edge i.e., edge with weight 5. Now the other two edges will create cycles so we will ignore them. In the end, we end up with a minimum spanning tree with total cost 11 (= 1 + 2 + 3 + 5).

TimeComplexity:
In Kruskal’s algorithm, most time consuming operation is sorting because the total complexity of the Disjoint-Set operations will be O(E log V), which is the overall Time Complexity of the algorithm.



Prim’s Algorithm

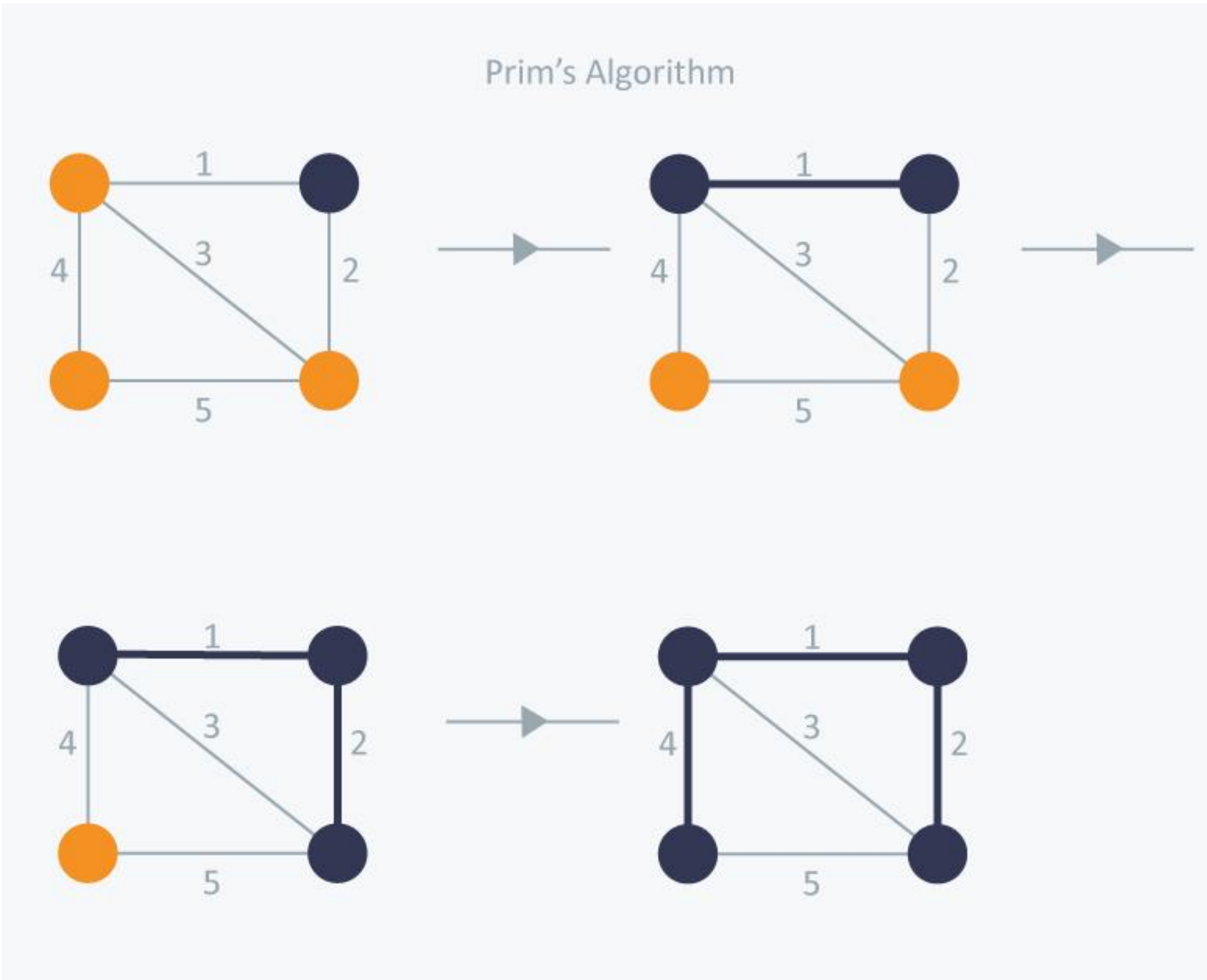
Prim’s Algorithm also use Greedy approach to find the minimum spanning tree. In Prim’s Algorithm we grow the spanning tree from a starting position. Unlike an **edge** in Kruskal's, we add **vertex** to the growing spanning tree in Prim's.

Algorithm Steps:

- Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.

- Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

Consider the example below:



In Prim's Algorithm, we will start with an arbitrary node (it doesn't matter which one) and mark it. In each iteration, we will mark a new vertex that is adjacent to the one that we have already marked. As a greedy algorithm, Prim's algorithm will select the cheapest edge and mark the vertex. So we will simply choose the edge with weight 1. In the next iteration we have three options, edges with weight 2, 3 and 4. So, we will select the edge with weight 2 and mark the vertex. Now again we have three options, edges with weight 3, 4 and 5. But we can't choose edge with weight 3 as it is creating a cycle. So we will select the edge with weight 4 and we end up with the minimum spanning tree of total cost 7 (= 1 + 2 +4).

MODULE VI

BackTracking: -The Control Abstraction – The N Queen's Problem, 0/1 Knapsack Problem. Branch and Bound: Travelling Salesman Problem. Introduction to Complexity Theory :-Tractable and Intractable Problems- The P and NP Classes- Polynomial Time Reductions - The NP- Hard and NP-Complete Classes

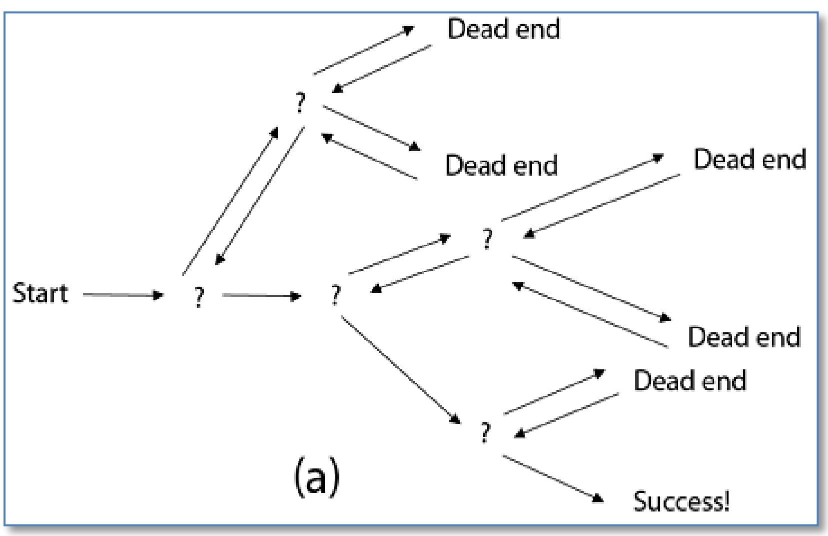
BACKTRACKING

The Backtracking is an algorithmic-method to solve a problem with an additional way. It uses a recursive approach to explain the problems. We can say that the backtracking is needed to find all possible combination to solve an optimization problem.

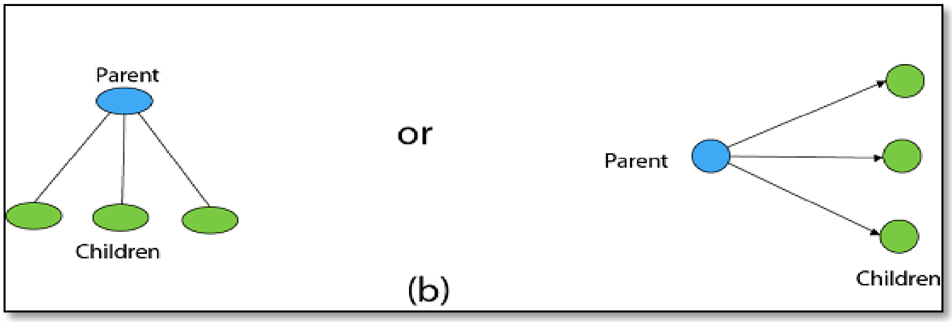
Backtracking is a systematic way of trying out different sequences of decisions until we find one that "works."

In the following Figure:

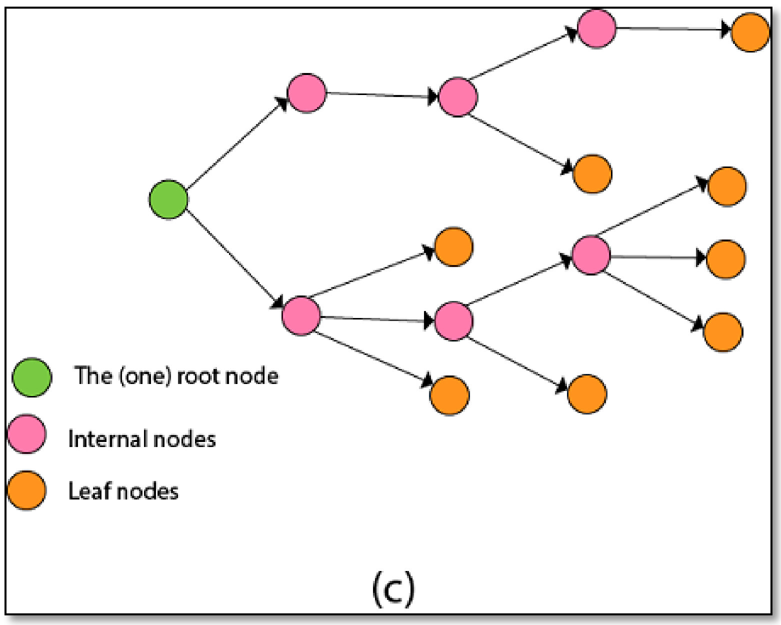
- Each non-leaf node in a tree is a parent of one or more other nodes (its children)
- Each node in the tree, other than the root, has exactly one parent



Generally, however, we draw our trees downward, with the root at the top.



A tree is composed of nodes.



Backtracking can understand of as searching a tree for a particular "goal" leaf node.

Backtracking is undoubtedly quite simple - we "explore" each node, as follows:

To "explore" node N:

1. If N is a goal node, return "success"
2. If N is a leaf node, return "failure"
3. For each child C of N,

   Explore C

   If C was successful, return "success"
4. Return "failure"

The Control Abstraction:

```
Algorithm Backtrack (v1,Vi)
  If (V1,……., Vi) is a Solution Then
              Return (V1,…, Vi)
  For each v DO
    If (V1,…….,Vi) is acceptable vector THEN
      Sol = try (V1,…,Vi, V)
      If sol != () Then
                    RETURN sol
    End
  End
  Return ( )
```

N- QUEEN PROBLEM

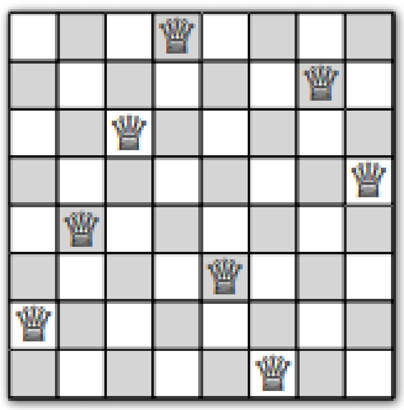
The prototypical backtracking problem is the classical n Queens Problem, first proposed by German chess enthusiast Max Bezzel in 1848 (under his pseudonym “Schachfreund”) for the standard 8 × 8 board and by François-Joseph Eustache Lionnet in 1869 for the more general n × n board.

The problem is to **place n queens on an n × n chessboard**, **so that no two queens can attack each other**. For readers not familiar with the rules of chess, this means that **no two queens are in the same row, column, or diagonal**. Obviously, in any solution to the n-Queens problem, there is exactly one queen in each row. So we will represent our possible solutions using an array Q[1 .. n], where Q[i] indicates which square in row i contains a queen, or 0 if no queen has yet been placed in row i.

To find a solution, we put queens on the board row by row, starting at the top. A partial solution is an array Q[1 .. n] whose first r − 1 entries are positive and whose last n − r + 1 entries are all zeros, for some integer r. The following recursive algorithm, essentially due to Gauss (who called it “methodical groping”), recursively enumerates all complete n-queens solutions that are consistent with a given partial solution.

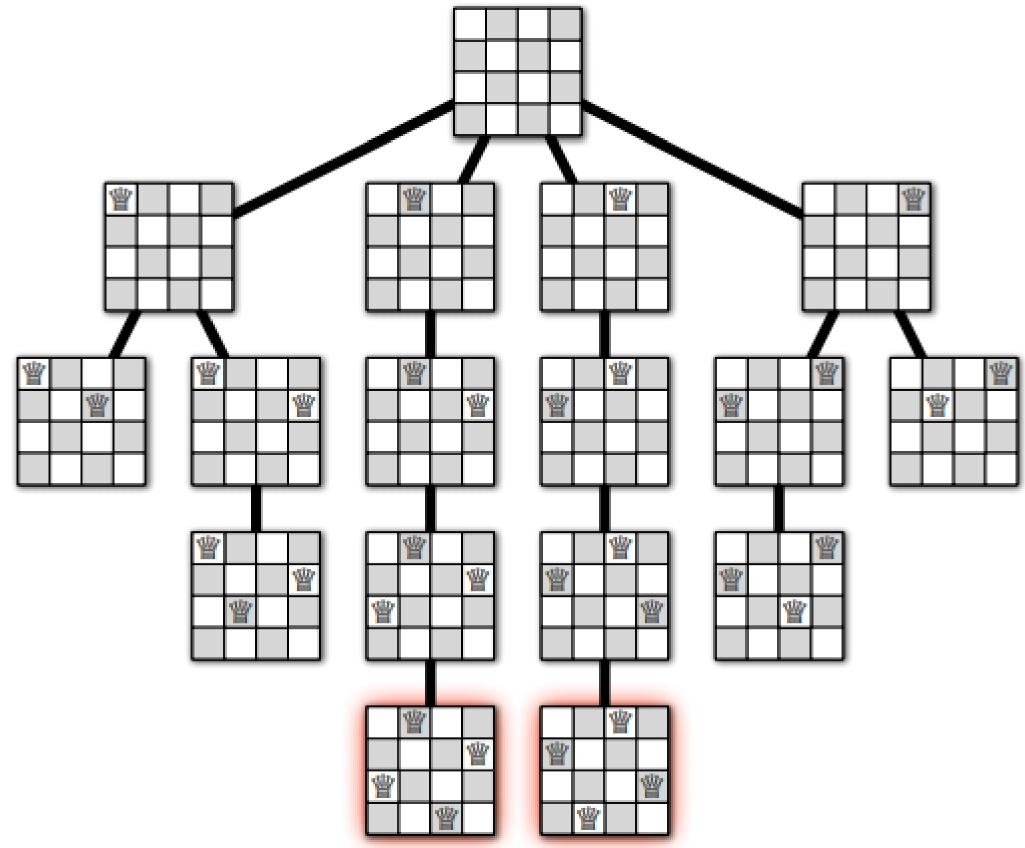
The input parameter r is the first empty row. Thus, to compute all n-queens solutions with no restrictions, we would call RECURSIVENQUEENS(Q[1 .. n], 1).

```
RecursiveNQueens(Q[1..n], r):
  if r = n + 1
      print Q
  else
      for j ← 1 to n
          legal ← True
          for i ← 1 to r − 1
              if (Q[i] = j) or (Q[i] = j + r − i) or (Q[i] = j − r + i)
                  legal ← False
          if legal
              Q[r] ← j
              RecursiveNQueens(Q[1..n], r + 1)
```



One solution to the 8 queens problem, represented by the array [4,7,3,8,2,5,1,6]

Like most recursive algorithms, the execution of a backtracking algorithm can be illustrated using a recursion tree. The root of the recursion tree corresponds to the original invocation of the algorithm; edges in the tree correspond to recursive calls. A path from the root down to any node shows the history of a partial solution to the n-Queens problem, as queens are added to successive rows. The leaves correspond to partial solutions that cannot be extended, either because there is already a queen on every row, or because every position in the next empty row is in the same row, column, or diagonal as an existing queen. The backtracking algorithm simply performs a depth-first traversal of this tree.

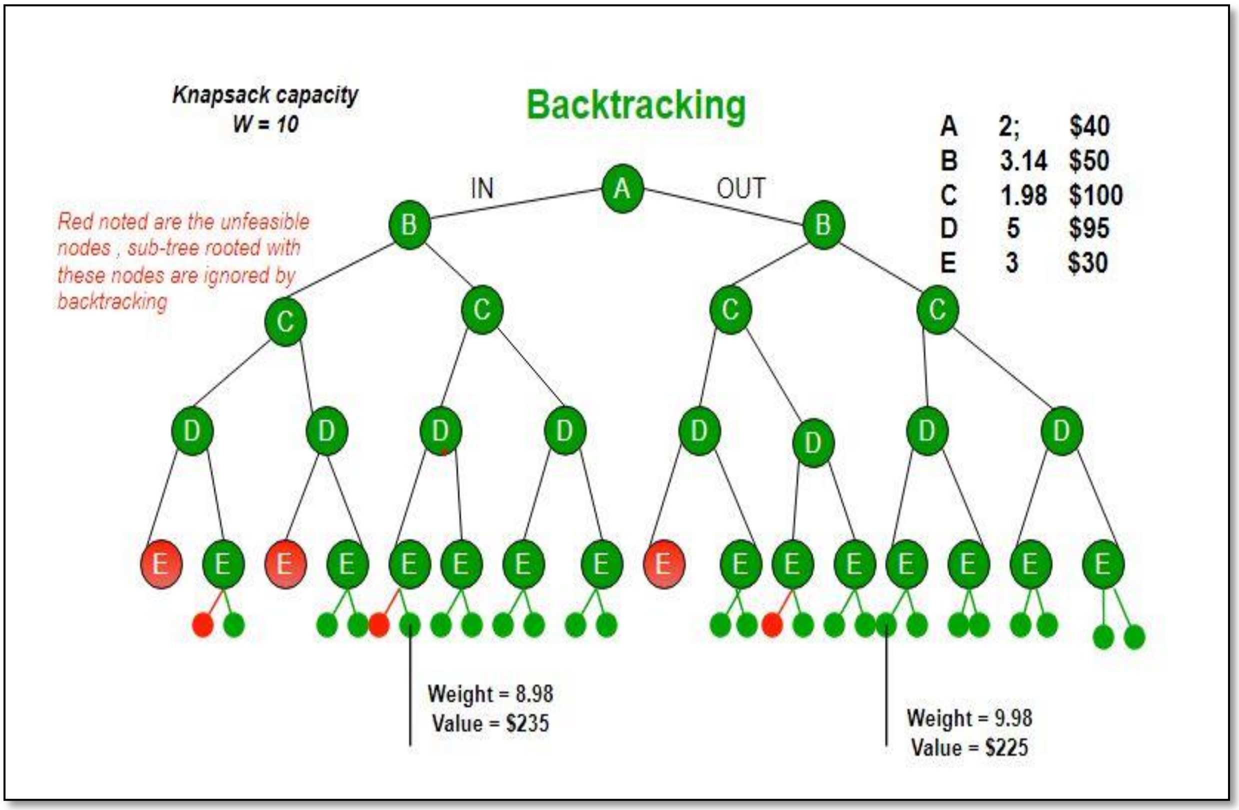


The complete recursion tree for our algorithm for the 4 queens problem.

0-1 KNAPSACK USING BACKTRACKING

1. A **Greedy** approach is to pick the items in decreasing order of value per unit weight. The Greedy approach works only for fractional knapsack problem and may not produce correct result for 0/1 knapsack.
2. We can use **D**ynamic **P**rogramming (**DP**) for 0/1 Knapsack problem. In DP, we use a 2D table of size n x W. The **DP Solution doesn't work if item weights are not integers**.
3. Since DP solution doesn't alway work, a solution is to use **Brute Force**. With n items, there are 2^n solutions to be generated, check each to see if they satisfy the constraint, save maximum solution that satisfies constraint. This solution can be expressed as **tree**.

We can use **Backtracking** to optimize the Brute Force solution. In the tree representation, we can do DFS of tree. If we reach a point where a solution no longer is feasible, there is no need to continue exploring. In the given example, backtracking would be much more effective if we had even more items or a smaller knapsack capacity.



BRANCH AND BOUND

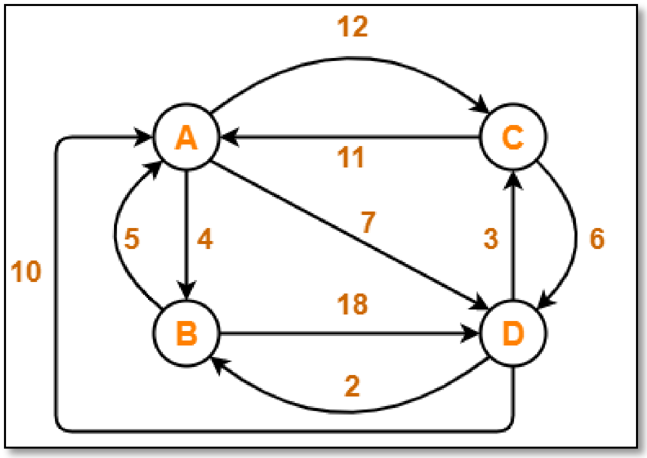
The branch and bound algorithm is similar to backtracking but is used for optimization problems. It performs a graph transversal on the space-state tree, but general searches BFS instead of DFS.

During the search **bounds** for the objective function on the partial solution are determined. At each level the best bound is explored first, the technique is called **best bound first**. If a complete solution is found then that value of the objective function can be used to prune partial solutions that exceed the bounds.

The difficult of designing branch and bound algorithm is finding good bounding function. The bounding the function should be inexpensive to calculate but should be effective at selecting the most promising partial solution.

TRAVELLING SALESMAN PROBLEM USING BRANCH AND BOUND

Solve Travelling Salesman Problem using Branch and Bound Algorithm in the following graph-



Solution-

Step-01:

Write the initial cost matrix and reduce it-

$$
\begin{array}{c c}
 & \begin{array}{cccc} A & B & C & D \end{array} \\
\begin{array}{c} A \\ B \\ C \\ D \end{array} & \begin{bmatrix} \infty & 4 & 12 & 7 \\ 5 & \infty & \infty & 18 \\ 11 & \infty & \infty & 6 \\ 10 & 2 & 3 & \infty \end{bmatrix}
\end{array}
$$

Rules

- To reduce a matrix, perform the row reduction and column reduction of the matrix separately.
- A row or a column is said to be reduced if it contains at least one entry ‘0’ in it.

Row Reduction-

Consider the rows of above matrix one by one.

If the row already contains an entry ‘0’, then-

- There is no need to reduce that row.

If the row does not contains an entry ‘0’, then-

- Reduce that particular row.
- Select the least value element from that row.
- Subtract that element from each element of that row.
- This will create an entry ‘0’ in that row, thus reducing that row.

Following this, we have-

- Reduce the elements of row-1 by 4.
- Reduce the elements of row-2 by 5.
- Reduce the elements of row-3 by 6.
- Reduce the elements of row-4 by 2.

Performing this, we obtain the following row-reduced matrix-

$$
\begin{array}{c c}
 & \begin{array}{cccc} A & B & C & D \end{array} \\
\begin{array}{c} A \\ B \\ C \\ D \end{array} & \begin{bmatrix} \infty & 0 & 8 & 3 \\ 0 & \infty & \infty & 13 \\ 5 & \infty & \infty & 0 \\ 8 & 0 & 1 & \infty \end{bmatrix}
\end{array}
$$

Column Reduction-

Consider the columns of above row-reduced matrix one by one.

If the column already contains an entry ‘0’, then-

- There is no need to reduce that column.

If the column does not contains an entry ‘0’, then-

- Reduce that particular column.
- Select the least value element from that column.
- Subtract that element from each element of that column.
- This will create an entry ‘0’ in that column, thus reducing that column.

Following this, we have-

- There is no need to reduce column-1.
- There is no need to reduce column-2.
- Reduce the elements of column-3 by 1.
- There is no need to reduce column-4.

Performing this, we obtain the following column-reduced matrix-

$$\begin{array}{c c} & \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{array}{c} A \\ B \\ C \\ D \end{array} & \begin{bmatrix} \infty & 0 & 7 & 3 \\ 0 & \infty & \infty & 13 \\ 5 & \infty & \infty & 0 \\ 8 & 0 & 0 & \infty \end{bmatrix} \end{array}$$

Finally, the initial distance matrix is completely reduced.

Now, we calculate the cost of node-1 by adding all the reduction elements.

Cost(1)= Sum of all reduction elements

= 4 + 5 + 6 + 2 + 1= **18**

Step-02:

We consider all other vertices one by one.

- We select the best vertex where we can land upon to minimize the tour cost.

Choosing To Go To Vertex-B: Node-2 (Path A → B)

From the reduced matrix of step-01, M[A,B] = 0

- Set row-A and column-B to ∞
- Set M[B,A] = ∞

Now, resulting cost matrix is-

$$\begin{array}{c c} & \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{array}{c} A \\ B \\ C \\ D \end{array} & \left[\begin{array}{cccc} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 13 \\ 5 & \infty & \infty & 0 \\ 8 & \infty & 0 & \infty \end{array}\right] \end{array}$$

Now,

- We reduce this matrix.
- Then, we find out the cost of node-02.

Row Reduction-

We can not reduce row-1 as all its elements are ∞.

- Reduce all the elements of row-2 by 13.
- There is no need to reduce row-3.
- There is no need to reduce row-4.

Performing this, we obtain the following row-reduced matrix-

$$
\begin{array}{c|cccc}
 & A & B & C & D \\
\hline
A & \infty & \infty & \infty & \infty \\
B & \infty & \infty & \infty & 0 \\
C & 5 & \infty & \infty & 0 \\
D & 8 & \infty & 0 & \infty
\end{array}
$$

Column Reduction-

Reduce the elements of column-1 by 5.

- We can not reduce column-2 as all its elements are ∞.
- There is no need to reduce column-3.
- There is no need to reduce column-4.

Performing this, we obtain the following column-reduced matrix-

$$
\begin{array}{c|cccc}
 & A & B & C & D \\
\hline
A & \infty & \infty & \infty & \infty \\
B & \infty & \infty & \infty & 0 \\
C & 0 & \infty & \infty & 0 \\
D & 3 & \infty & 0 & \infty
\end{array}
$$

Finally, the matrix is completely reduced.

Now, we calculate the cost of node-2.

Cost(2)= Cost(1) + Sum of reduction elements + M[A,B]

= 18 + (13 + 5) + 0= **36**

Choosing To Go To Vertex-C: Node-3 (Path A → C)

From the reduced matrix of step-01, M[A,C] = 7

- Set row-A and column-C to ∞
- Set M[C,A] = ∞

Now, resulting cost matrix is-

$$
\begin{array}{c|cccc}
 & A & B & C & D \\
\hline
A & \infty & \infty & \infty & \infty \\
B & 0 & \infty & \infty & 13 \\
C & \infty & \infty & \infty & 0 \\
D & 8 & 0 & \infty & \infty
\end{array}
$$

Now,

- We reduce this matrix.
- Then, we find out the cost of node-03.

Row Reduction-

We can not reduce row-1 as all its elements are ∞.

- There is no need to reduce row-2.
- There is no need to reduce row-3.
- There is no need to reduce row-4.

Thus, the matrix is already row-reduced.

Column Reduction-

There is no need to reduce column-1.

- There is no need to reduce column-2.
- We can not reduce column-3 as all its elements are ∞.
- There is no need to reduce column-4.

Thus, the matrix is already column reduced.

Finally, the matrix is completely reduced.

Now, we calculate the cost of node-3.

Cost(3)= Cost(1) + Sum of reduction elements + M[A,C]

= 18 + 0 + 7= **25**

Choosing To Go To Vertex-D: Node-4 (Path A → D)

From the reduced matrix of step-01, M[A,D] = 3

- Set row-A and column-D to ∞
- Set M[D,A] = ∞

Now, resulting cost matrix is-

$$
\begin{array}{c|cccc|}
 & A & B & C & D \\
A & \infty & \infty & \infty & \infty \\
B & 0 & \infty & \infty & \infty \\
C & 5 & \infty & \infty & \infty \\
D & \infty & 0 & 0 & \infty
\end{array}
$$

Now,

- We reduce this matrix.
- Then, we find out the cost of node-04.

Row Reduction-

We can not reduce row-1 as all its elements are ∞.

- There is no need to reduce row-2.
- Reduce all the elements of row-3 by 5.
- There is no need to reduce row-4.

Performing this, we obtain the following row-reduced matrix-

$$
\begin{array}{c|cccc|}
 & A & B & C & D \\
A & \infty & \infty & \infty & \infty \\
B & 0 & \infty & \infty & \infty \\
C & 0 & \infty & \infty & \infty \\
D & \infty & 0 & 0 & \infty
\end{array}
$$

Column Reduction-

There is no need to reduce column-1.

- There is no need to reduce column-2.
- There is no need to reduce column-3.
- We can not reduce column-4 as all its elements are ∞.

Thus, the matrix is already column-reduced.

Finally, the matrix is completely reduced.

Now, we calculate the cost of node-4.

Cost(4)= Cost(1) + Sum of reduction elements + M[A,D]

= 18 + 5 + 3= **26**

Thus, we have-

- Cost(2) = 36 (for Path A → B)
- Cost(3) = 25 (for Path A → C)
- Cost(4) = 26 (for Path A → D)

We choose the node with the lowest cost.

Since cost for node-3 is lowest, so we prefer to visit node-3.

Thus, we choose node-3 i.e. path **A** → **C**.

Step-03:

We explore the vertices B and D from node-3.

We now start from the cost matrix at node-3 which is-

$$
\begin{array}{c c}
 & \begin{array}{cccc} A & B & C & D \end{array} \\
\begin{array}{c} A \\ B \\ C \\ D \end{array} & \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 13 \\ \infty & \infty & \infty & 0 \\ 8 & 0 & \infty & \infty \end{bmatrix}
\end{array}
$$

Cost(3) = 25

Choosing To Go To Vertex-B: Node-5 (Path A → C → B)

From the reduced matrix of step-02, M[C,B] = ∞

- Set row-C and column-B to ∞
- Set M[B,A] = ∞

Now, resulting cost matrix is-

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & \infty & \infty & \infty \\ B & \infty & \infty & \infty & 13 \\ C & \infty & \infty & \infty & \infty \\ D & 8 & \infty & \infty & \infty \end{array}$$

Now,

- We reduce this matrix.
- Then, we find out the cost of node-5.

Row Reduction-

We can not reduce row-1 as all its elements are ∞.

- Reduce all the elements of row-2 by 13.
- We can not reduce row-3 as all its elements are ∞.
- Reduce all the elements of row-4 by 8.

Performing this, we obtain the following row-reduced matrix-

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & \infty & \infty & \infty \\ B & \infty & \infty & \infty & 0 \\ C & \infty & \infty & \infty & \infty \\ D & 0 & \infty & \infty & \infty \end{array}$$

Column Reduction-

There is no need to reduce column-1.

- We can not reduce column-2 as all its elements are ∞.
- We can not reduce column-3 as all its elements are ∞.
- There is no need to reduce column-4.

Thus, the matrix is already column reduced.

Finally, the matrix is completely reduced.

Now, we calculate the cost of node-5.

Cost(5)= cost(3) + Sum of reduction elements + M[C,B]

= 25 + (13 + 8) + ∞= ∞

Choosing To Go To Vertex-D: Node-6 (Path A → C → D)

From the reduced matrix of step-02, M[C,D] = ∞

- Set row-C and column-D to ∞
- Set M[D,A] = ∞

Now, resulting cost matrix is-

$$\begin{array}{c c} & \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{array}{c} A \\ B \\ C \\ D \end{array} & \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \end{bmatrix} \end{array}$$

Now,

- We reduce this matrix.
- Then, we find out the cost of node-6.

Row Reduction-

We can not reduce row-1 as all its elements are ∞.

- There is no need to reduce row-2.
- We can not reduce row-3 as all its elements are ∞.
- We can not reduce row-4 as all its elements are ∞.

Thus, the matrix is already row reduced.

Column Reduction-

There is no need to reduce column-1.

- We can not reduce column-2 as all its elements are ∞.
- We can not reduce column-3 as all its elements are ∞.
- We can not reduce column-4 as all its elements are ∞.

Thus, the matrix is already column reduced.

Finally, the matrix is completely reduced.

Now, we calculate the cost of node-6.

Cost(6)= cost(3) + Sum of reduction elements + M[C,D]

= 25 + 0 + 0= **25**

Thus, we have-

- Cost(5) = ∞ (for Path A → C → B)
- Cost(6) = 25 (for Path A → C → D)

We choose the node with the lowest cost.

Since cost for node-6 is lowest, so we prefer to visit node-6.

Thus, we choose node-6 i.e. path **C** → **D**.

Step-04:

We explore vertex B from node-6.

We start with the cost matrix at node-6 which is-

$$\begin{array}{cc} & \begin{matrix} A & B & C & D \end{matrix} \\ \begin{matrix} A \\ B \\ C \\ D \end{matrix} & \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \end{bmatrix} \end{array}$$

Cost(6) = 25

<u>Choosing To Go To Vertex-B: Node-7 (Path A → C → D → B)</u>

From the reduced matrix of step-03, M[D,B] = 0

- Set row-D and column-B to ∞
- Set M[B,A] = ∞

Now, resulting cost matrix is-

$$\begin{array}{cc} & \begin{matrix} A & B & C & D \end{matrix} \\ \begin{matrix} A \\ B \\ C \\ D \end{matrix} & \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \end{bmatrix} \end{array}$$

Now,

- We reduce this matrix.
- Then, we find out the cost of node-7.

<u>Row Reduction-</u>

- We can not reduce row-1 as all its elements are ∞.
- We can not reduce row-2 as all its elements are ∞.
- We can not reduce row-3 as all its elements are ∞.
- We can not reduce row-4 as all its elements are ∞.

<u>Column Reduction-</u>

- We can not reduce column-1 as all its elements are ∞.

- We can not reduce column-2 as all its elements are ∞.
- We can not reduce column-3 as all its elements are ∞.
- We can not reduce column-4 as all its elements are ∞.

Thus, the matrix is already column reduced.

Finally, the matrix is completely reduced.

All the entries have become ∞.

Now, we calculate the cost of node-7.

Cost(7)= cost(6) + Sum of reduction elements + M[D,B]

=25+0+0=25.

Thus,

Optimal Path is : A->C->D->B->A

Cost of Optimal Path = 25 units.

The classes P and NP :

The class P consists of those problems that are solvable in polynomial time. More specifically, they are problems that can be solved in time $O(n^k)$ for some constant k, where n is the size of the input to the problem. Most of the problems examined in previous modules are in P.

The class NP consists of those problems that are 'verifiable' in polynomial time. If we were somehow given a 'certificate' of a solution, then we could verify that the certificate is correct in time polynomial in the size of the input to the problem.

n

Example:

PATH
INPUT : graph G, nodes a and b
Question : Is there a path from a to b in G ?

This problem is in P. To see if there is a path from node a to node b, one might determine all the nodes reachable from a by doing for instance a breadth-first search or Dijkstra's algorithm.

Verification Algorithm:

A verification algorithm is an algorithm A, that takes two inputs: an ordinary input x, and a certificate y, and outputs a 1 on certain combinations of x and y.

Verification algorithm A verifies an input string x if there exists a certificate y such that $A(x, y) = 1$.

The language verified by verification algorithm A is

$$L = \{\text{input string } x \mid \text{there exists certificate string } y \text{ such that } A(x,y) = 1\}$$

Example:

In the hamiltonian cycle problem, given a directed graph $G = (V, E)$, a certificate would be sequence $\langle V_1, V_2, V_3 \ldots V_{|V|} \rangle$ of $|V|$ vertices. We would easily check in polynomial time that $(V_i, V_{i+1}) \in E$ for $i = 1, 2, 3 \ldots |V| - 1$ and that $(V_{|V|}, V_1) \in E$ as well.

Polynomial-time Verification Algorithm:

A verification algorithm A for a language L is a polynomial-time verification algorithm for L if

- for each $x \in L$, there is a certificate y of size polynomial in the size of x such that $A(x,y) = 1$, and $A(x,y)$ returns 1 in time polynomial in x.
- Since A is a verification algorithm for L, for every x not in L there is no certificate y for which $A(x,y) = 1$.

P, NP and NP-complete:

Any problem in P is also in NP, since if a problem is in P then we can solve it in polynomial time without even being supplied a certificate. So we can believe that $P \subseteq NP$. The open question is whether or not P is a proper subset of NP.

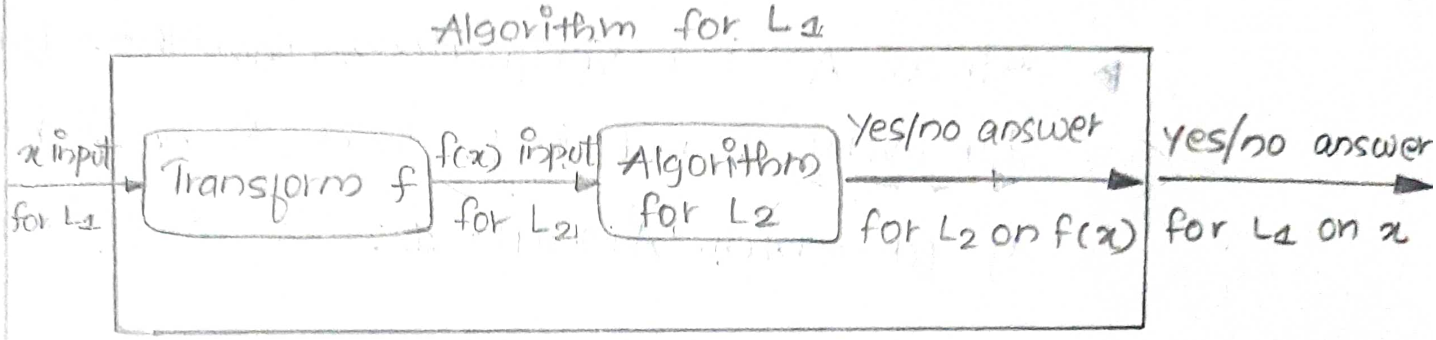
pg no 6. PTO

Informally, a problem is in the class NPC — and we refer to it as being NP-complete — if it is in NP and is as "hard" as any problem in NP. If any NP-complete problem can be solved in polynomial time, then every problem in NP has a polynomial time algorithm.

Reduction:

Let L_1 and L_2 be two decision problems. Suppose algorithms A_2 solves L_2. That is, if y is an input for L_2 then algorithm A_2 will answer Yes or No depending upon whether $y \in L_2$ or not.

The idea is to find a transformation f from L_1 to L_2 so that the algorithm A_2 can be part of an algorithm A_1 to solve L_1.



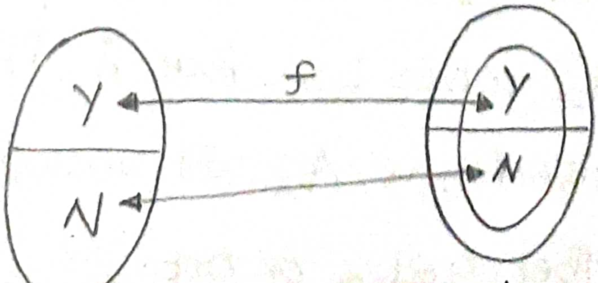
Polynomial – time Reduction:

Let L_1 and L_2 be languages that are subsets of $\{0,1\}^*$. We say that L_1 is polynomial – time reducible to L_2 if there exists a function f

$$f : \{0,1\}^* \longrightarrow \{0,1\}^*$$

with the following properties.

- f transforms an input x for L_1 into an input $f(x)$ for L_2 such that $f(x)$ is a yes – input for L_2 if and only if x is a yes input for L_1. We require a yes – input of L_1 maps to a yes-input of L_2, and a no-input of L_1 maps to a no-input of L_2.



- $f(x)$ is computable in polynomial time.

If such an f exists, we say that L_1 is Polynomial-time reducible to L_2, and write $L_1 \leq_p L_2$.

$L_1 \leq_p L_2$

Languages in NP:

Let us consider the following examples of decision problems.

- HAM-CYCLE $= \{\langle G \rangle \mid G$ is a Hamiltonian graph$\}$
- CIRCUIT-SAT $= \{\langle C \rangle \mid C$ is a satisfiable boolean ckt$\}$
- SAT $= \{\langle \phi \rangle \mid \phi$ is satisfiable boolean formula$\}$
- CNF-SAT $= \{\langle \phi \rangle \mid \phi$ is a satisfiable boolean formula in CNF$\}$
- 3-CNF-SAT $= \{\langle \phi \rangle \mid \phi$ is a satisfiable boolean formula in CNF$\}$
- CLIQUE $= \{\langle G, K \rangle \mid G$ is an undirected graph with a clique of size $k\}$
- IS $= \{\langle G, K \rangle \mid G$ is an undirected graph with an independent set of size $k\}$
- VERTEX-COVER $= \{\langle G, K \rangle \mid$ undirected graph G has a vertex cover of size $k\}$

- $TSP = \{\langle G, c, k\rangle \mid G = (V, E)$ is a complete graph, $c : V \times V \to \mathbb{Z}$ is a cost function, $k \in \mathbb{Z}$ and G has a traveling salesman tour with cost at most $k\}$
- $SUBSET\text{-}SUM = \{\langle S, t\rangle \mid$ there is a subset $S' \subseteq S$ such that $t = \sum_{s \in S'} s\}$

Is P = NP?

One of the most important problems in computer science is whether P=NP or P≠NP? Observe that $P \subseteq NP$. Given a problem $A \in P$, and a certificate to verify the ~~validate~~ validity of a yes-input (an instance of A), we can simply solve A in polynomial time (since $A \in P$). It implies $A \in NP$.

Intuitively, $NP \subseteq P$ is doubtful. After all, just able to verify a certificate in polynomial time does not necessary mean we can able to tell whether an input is an yes-input or no-input in polynomial time.

However, 30 years after the P=NP? problem was first proposed, we are still no closer to solving it and do not know the answer. The search for a solution though, has provided us with deep insights into what distinguishes an 'easy' problem ~~for~~ from a 'hard' one.

The class CO-NP:

$$L \in NP$$
$$\bar{L} \in NP$$

Note that if $L \in NP$, there is no guarantee that $\bar{L} \in NP$ (since having certificates for yes-inputs, does not mean that we have certificates for the no-inputs).

The class of decision problems L such that $\bar{L} \in NP$ is called CO-NP.

Prime $\in NP$ 2, 3, 5, 7
Composite $\in NP$

Example: COMPOSITE $\in NP$ so PRIME $= \overline{\text{COMPOSITE}} \in$ CO-NP

256

The complexity class NP is the class of languages that can be verified by a polynomial-time algorithm. More precisely, a language L belongs to NP if and only if there exist a two-input polynomial-time algorithm 'A' and a constant 'c' such that

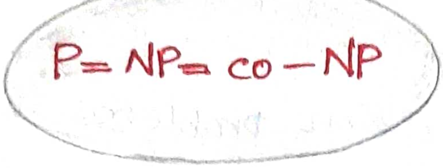
$$L = \left\{ x \in \{0,1\}^* : \text{there exist a certificate } y \text{ with } |y| = O(|x|^c) \text{ such that } A(x,y) = 1 \right\}.$$

We say that algorithm A verifies language L in polynomial time.

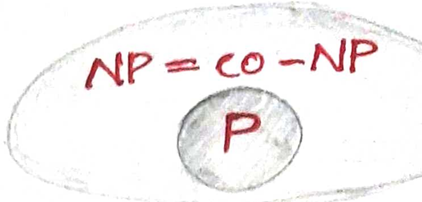
We can define the complexity class CO-NP as the set of languages L such that $\bar{L} \in NP$.

Once again, no one knows whether $P = NP \cap$ CO-NP or whether there is some language in $NP \cap$ CO-NP-P

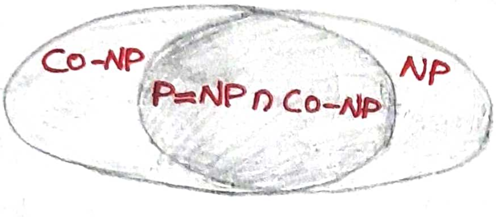
Possibilities for relationships among complexity class



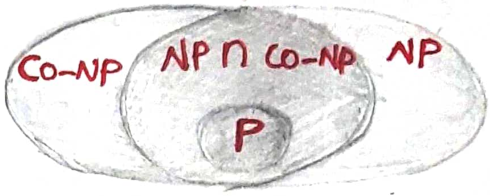
(a)



(b)



(c)



(d)

In each diagram, one region enclosing another indicates a proper-subset relation.

(a) $P = NP = Co\text{-}NP$. Most researchers regard this possibility as the most unlikely.

(b) If NP is closed under complement, then $NP = Co\text{-}NP$, but it need not be the case that $P = NP$.

(c) $P = NP \cap co\text{-}NP$, but NP is not closed under complement.

(d) $NP \neq Co\text{-}NP$ and $P \neq NP \cap Co\text{-}NP$. Most researchers regard this possibility as the most likely.

NP-Hard:

A language $L \subseteq \{0,1\}^*$ is NP-complete if

1. $L \in NP$, and
2. $L' \leq_p L$ for every $L' \in NP$

If a language L satisfies property 2, but not necessarily property 1, we say that L is NP-Hard.

NP-hardness is a class of problems that are, informally, "at least as hard as the hardest problems in NP". More precisely, a problem H is NP-Hard when every problem L in NP can be reduced in polynomial time to H.

Theorem:

If any NP-complete problem is polynomial time solvable, then $P = NP$. If any problem in NP is not polynomial time solvable, then all NP-complete problems are not polynomial time solvable.

Proof:

Suppose that $L \in P$ and also that $L \in NPC$. For any $L' \in NP$, we have $L' \leq_p L$ by property 2 of the definition of NP-completeness.

> A language $L \subseteq \{0,1\}^*$ is NP complete if it satisfies the following two properties:
>
> 1. $L \in NP$; and
> 2. For every $L' \in NP$, $L' \leq_p L$
>
> We use the notation $L \in NPC$ to denote that L is NP-complete.

We know if $L' \leq_p L$ then $L \in P$ implies $L' \in P$, which proves the first statement.

To proves the second statement, ... that there exists an $L \in NP$ such that $L \in$ ... Let $L' \in NPC$ be any NP-complete language, and for the purpose of contradiction, assume that $L' \in P$. But then we have $L_p \leq_p L'$ and thus $L \in P$.

Proving NP-completeness:

To prove that a problem P is NP-complete, we have following methods:

Method 1: (direct proof)

(a) P is in NP

(b) All problems in NP-complete can be reduced to P.

Method 2: (equivally general but potentially easier)

(a) P is in NP

(b) Find a problem P' that has already been Proven to be in NP-complete

(c) Show that $P' \leq P$.

NP-Complete Problems

Examples of NP-complete problems

1. Formula Satisfiability
2. Circuit satisfiability
3. 3-CNF Satisfiability
4. clique
5. Vertex cover
6. Subset-Sum
7. Hamiltonian cycle
8. Traveling Salesman

Formula Satisfiability:

Formula satisfiability problem is the historical honor of being the first problem ever shown to be NP-complete.

We formulate the (formula) satisfiability problem in terms of the language SAT as follows.

An instance of SAT is a boolean formula ϕ composed of

(1) n boolean variables: $x_1, x_2 \ldots x_n$

(2) m boolean connectives: any boolean function with one or two inputs and one output, such as $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT), $\rightarrow$ (implication), $\leftrightarrow$ (if and only if); and

(3) Parentheses

We can easily encode a boolean formula ϕ in a length that is polynomial in $n+m$. As in boolean combinational circuits, a "truth assignment" for a boolean formula ϕ is a set of values for the variables of ϕ, and a "satisfying assignment" is a truth assignment that causes it to evaluate to 1. A formula with a satisfying assignment is a satisfiable formula. The satisfiability problem asks whether a given boolean formula ~~ϕ~~ is satisfiable; in formal terms,

~~SAT = $\{\langle\phi\rangle\}$~~ SAT = $\{\langle\phi\rangle : \phi$ is a satisfiable boolean formula$\}$.

$x_1=0, x_2=1, x_3=0$
$(x_1 \wedge x_2 \wedge x_3) \vee (x_1 \vee \neg x_2 \vee x_3)$
$(0 \wedge 1 \wedge 0) \vee (0 \vee 0 \vee 0)$
$0 \vee 0 = 0$

$x_1=1, x_2=1, x_3=1$
$(1 \wedge 1 \wedge 1) \vee (1 \vee 0 \vee 1)$
$1 \vee 1 = 1$

algo... able

Example:

$$\phi = (x_1 \to x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

has the satisfying assignment

$\langle x_1 = 0,\ x_2 = 0,\ x_3 = 1,\ x_4 = 1 \rangle$, since

$$\phi = (0 \to 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0 \qquad \text{①}$$

$$= (1 \vee \neg(1 \vee 1)) \wedge 1$$

$$= (1 \vee 0) \wedge 1$$

$$= 1.$$

and thus this formula ϕ belongs to SAT.

Theorem

Satisfiability of boolean formulas is NP-complete.

Proof:

To prove SAT is NP-complete, we prove that

- SAT $\in$ NP
- SAT is NP-Hard

To show that SAT $\in$ NP, we show that a certificate consisting of a satisfying assignment for an input formula ϕ can be verified in polynomial time. The verifying algorithm simply replaces each variable in the formula with its corresponding value and then evaluate the expression, much as we did in eq—① above. This task is easy to do in polynomial time. If the expression evaluate to 1, then the algorithm has verified that the formula is satisfiable.

To prove that SAT is NP-hard, we show that CIRCUIT-SAT $\leq_p$ SAT. In other words, we need to show how to reduce any instance of circuit satisfiability to an instance of formula satisfiability in polynomial time. We can use induction to express any boolean combinational circuit as a boolean formula. We simply look at the gate that produces the circuit output and inductively express each of the gate's inputs as formula's. We then obtain the formula for the circuit by writing an expression that applies the gate's function to its input's formula's.

CIRCUIT-SATISFIABILITY:

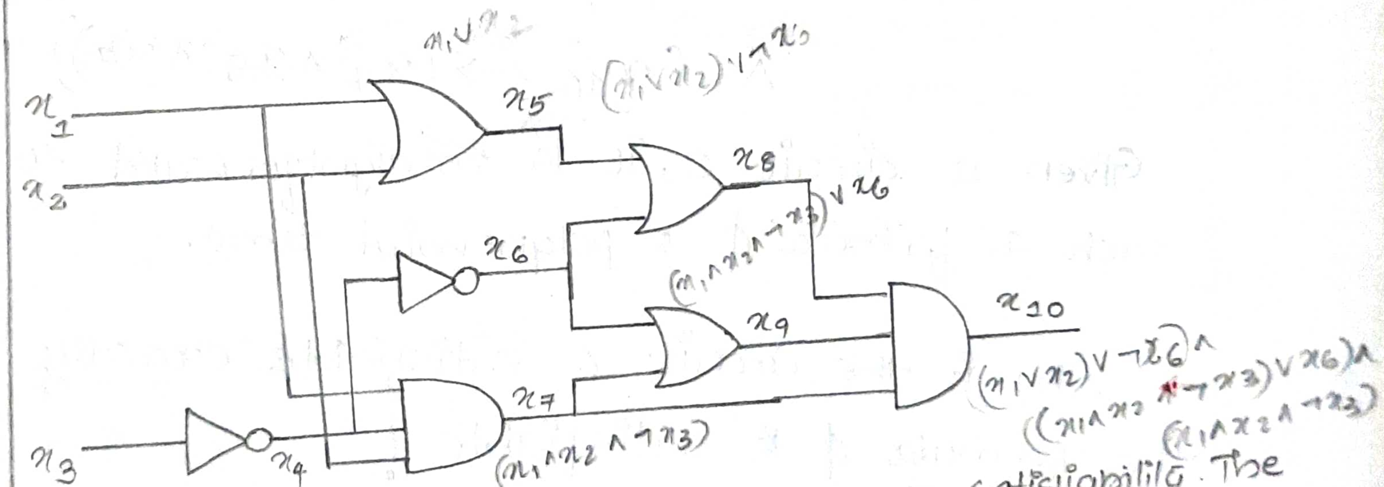


Fig: Reducing circuit satisfiability to formula satisfiability. The formula produced by the reduction algorithm has a variable for each wire in the circuit.

Figure illustrates that, for each wire x_i in the circuit C, the formula ϕ has a variable x_i. We can now express how each gate operates as a small formula involving the variables of its incident wires. For example, the operation of the output AND gate is $x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)$. We call each of these small formulas a clause.

The formula ϕ produced by the reduction al[...] the circuit output variable with is the AND of the circuit output variable with the conjunction of clauses describing the operatio[n] of each gate.

For circuit in the figure, the formula is

$$\phi = x_{10} \wedge (x_4 \longleftrightarrow \neg x_3)$$
$$\wedge (x_5 \longleftrightarrow (x_1 \vee x_2))$$
$$\wedge (x_6 \longleftrightarrow \neg x_4)$$
$$\wedge (x_7 \longleftrightarrow (x_1 \wedge x_2 \wedge x_4))$$
$$\wedge (x_8 \longleftrightarrow (x_5 \vee x_6))$$
$$\wedge (x_9 \longleftrightarrow (x_6 \vee x_7))$$
$$\wedge (x_{10} \longleftrightarrow (x_7 \wedge x_8 \wedge x_9))$$

Given a circuit C, it is straightforward to produce such a formula ϕ in polynomial time.

→ Why is the circuit C satisfiable exactly when the formula ϕ is satisfiable?

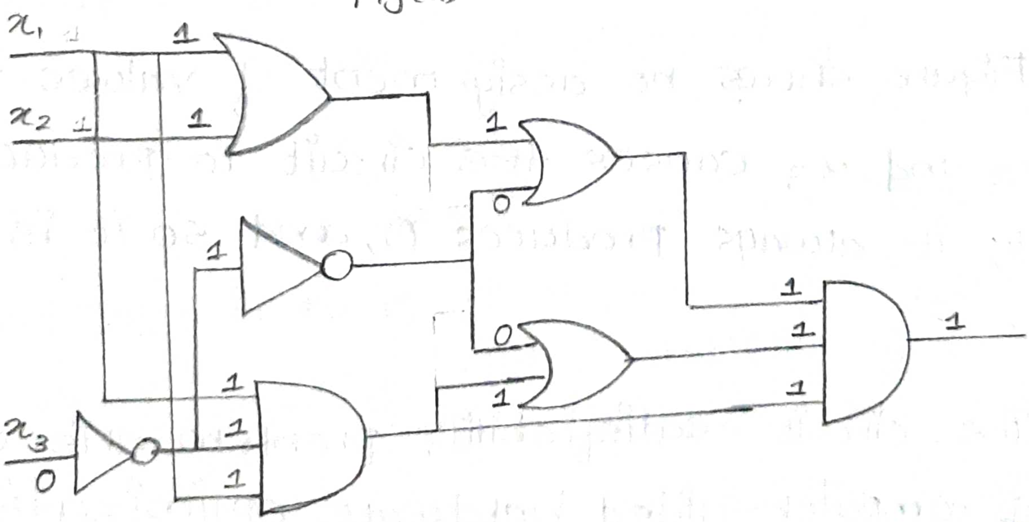
If C has a satisfying assignment, then each wire of the circuit has a well-defined value, and the output of the circuit is 1. Therefore, when we assign wire values to variables in ϕ, each clause of ϕ evaluates to 1, and thus the conjunction of all evaluates to 1. conversely, if some assignment causes ϕ to evaluate to 1, the circuit C is satisfiable by an analogous argument. Thus, we show that CIRCUIT_SAT $\leq_p$ SAT, which completes the Proof.

Circuit satisfiability:

Boolean combinational circuits are built from boolean combinational elements that are interconnected by wires. A boolean combinational element is any circuit elements that has a constant number of boolean inputs and outputs and that performs a well-defined function. Boolean values are drawn from the set $\{0,1\}$, where 0 represents FALSE and 1 represent TRUE.

A truth assignment for a boolean combinational circuit is a set of boolean values. We say that a one-output boolean combinational circuit is satisfiable if it has a satisfying assignment: a truth assignment that causes the output of the circuit to be 1.

Fig(a)



For example, the circuit in Fig(a) has the satisfying assignment $\langle x_1=1, x_2=1, x_3=0\rangle$, and so it is satisfiable.

Theorem 2

The CIRCUIT-SAT is NP-complete.

Proof

To prove CIRCUIT-SAT is NP-complete, we prove that,

- CIRCUIT-SAT $\in$ NP
- CIRCUIT-SAT is NP-hard.

The first part is proved in previous theorem. For proving CIRCUIT-SAT is NP-hard, we have to reducing ~~circuit~~ circuit satisfiability to formula satisfiability. The formula produced by the reduction algorithm has a variable for each wire in the circuit.

From fig (a), for each wire x_i in the circuit C, the formula ϕ has a variable x_i. We can now express how each gate operates as a small formula involving the variables of its incident wires. For example, the operation of the output AND gate is $x_{10} \longleftrightarrow (x_7 \wedge x_8 \wedge x_9)$. We call each of these small formulas a clause.

The formula ϕ produced by the reduction algorithm is the AND of the circuit-output variable with the conjunction of clauses describing the operation of each gate.

For the circuit in the figure, the formula is

$$\phi = x_{10} \wedge (x_4 \longleftrightarrow \neg x_3)$$
$$\wedge (x_5 \longleftrightarrow (x_1 \vee x_2))$$
$$\wedge (x_6 \longleftrightarrow \neg x_4)$$
$$\wedge (x_7 \longleftrightarrow (x_1 \wedge x_2 \wedge x_4))$$
$$\wedge (x_8 \longleftrightarrow (x_5 \vee x_6))$$
$$\wedge (x_9 \longleftrightarrow (x_6 \vee x_7))$$
$$\wedge (x_{10} \longleftrightarrow (x_7 \wedge x_8 \wedge x_9))$$

Given a circuit C, it is straightforward to produce such a formula ϕ in polynomial time.

If C has a satisfying assignment, then each wire of the circuit has a well-defined value, and the output of the circuit is 1. Therefore, when we assign wire values to variables in ϕ, each clause of ϕ evaluates to 1, and thus the conjunction of all evaluates to 1. Conversely, if some assignment causes ϕ to evaluate to 1, the circuit ~~sat~~ C is satisfiable by an analogous argument. Thus, we have shown that CIRCUIT-SAT $\leq_p$ SAT, which completes the proof.

3-CNF Satisfiability:

A literal in a boolean formula is an occurrence of a variable or its negation. A boolean formula is in conjunctive normal form, or CNF, if it is expressed as a AND of clauses, each of which is the OR of one or more literals. A boolean formula is in 3-conjunctive normal form, or 3-CNF, if each clause has exactly three distinct literals.

For example, the boolean formula

$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

is in 3-CNF. The first of its three clauses is $(x_1 \vee \neg x_1 \vee \neg x_2)$, which contains the three literals x_1, $\neg x_1$, and $\neg x_2$.

The 3-CNF-SAT problem is whether a given boolean formula ϕ in 3-CNF is satisfiable.

The formal language for 3-CNF-SAT is

$$\text{3-CNF-SAT} = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable boolean formula in CNF}\}$$

Theorem

Satisfiability of boolean formulas in 3-conjunctive normal form is NP-complete.

Proof

The proof SAT $\in$ NP applies equally well here to show that 3-CNF-SAT $\in$ NP.

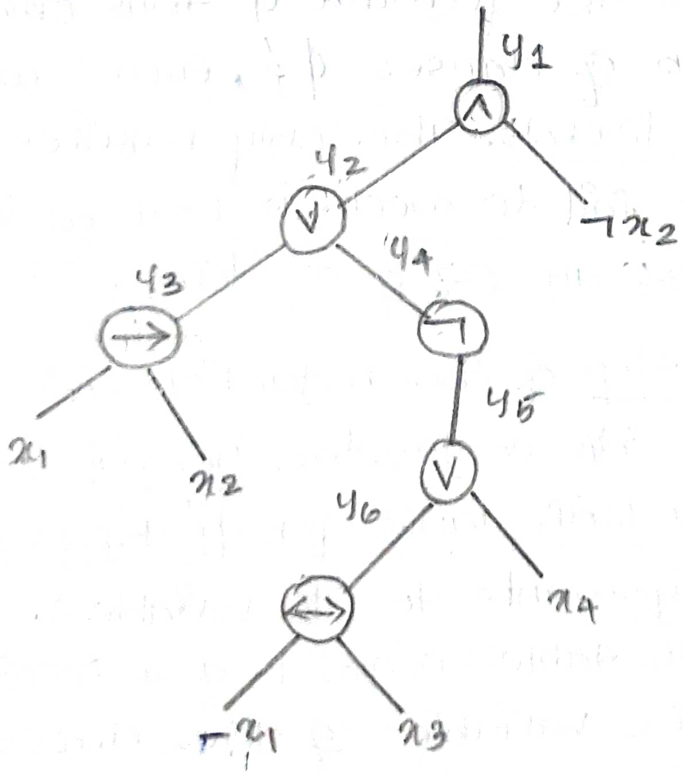
Lemma:

If L is a language such that $L' \leq_p L$ for some $L' \in NPC$, then L is NP-hard. If, in addition, $L \in NP$, then $L \in NPC$

By lemma, therefore, we need only show that SAT $\leq_p$ 3-CNF-SAT.

We break the reduction algorithm into 3 basic steps. Each step progressively transforms the input formula ϕ closer to the desired 3-CNF.

▶ The first step is similar to the one to prove CIRCUIT-SAT $\leq_p$ SAT. First we construct a binary parse tree for the input formula ϕ, with literals as leaves & connectives as internal nodes.



Fig(a): The tree corresponding to the formula $\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$

Formula for the given parse tree is

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2 .$$

We rewrite the original formula ϕ as the AND of the root variable and a conjunction of clauses describing the operation of each node.

For the formula, the resulting expression is

$$\phi' = y_1 \wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2))$$
$$\wedge (y_2 \leftrightarrow (y_3 \vee y_4))$$
$$\wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2))$$
$$\wedge (y_4 \leftrightarrow \neg y_5)$$
$$\wedge (y_5 \leftrightarrow (y_6 \vee x_4))$$
$$\wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3)).$$

Observe that the formula ϕ' thus obtained is a conjunction of clauses ϕ_i', each of which has at most 3 literals. The only requirement that we might fail to meet is that each clause has to be an OR of 3 literals.

▶ The second step of the reduction converts each clause ϕ_i' into conjunctive normal form. We construct a truth table for ϕ_i' by evaluating all possible assignments to its variables. Each row of the truth table consist of a possible assignment of the variables of the clause, together with the value of the clause under that assignment.

Material implication → $A \rightarrow B$ is true only in the case that either A is false or B is true.

implies → (same as →)

Material equivalence ⟺ $A \Leftrightarrow B$ is true only if both A & B are true or both A & B are false.

Using the truth-table entries that evaluates to 0, we ~~would~~ build a formula in disjunctive normal form (or DNF) — an OR of ANDs — that is equivalent to $\neg\phi_i'$. We then negate this formula and convert it into a CNF formula ϕ_i'' by using De Morgan's laws for propositional logic,

$$\neg(a \wedge b) = \neg a \vee \neg b$$

$$\neg(a \vee b) = \neg a \wedge \neg b$$

to complement all literals, change ORs into ANDs, and change ANDs into ORs.

In our example, we convert the clause $\phi_1' = (y_1 \leftrightarrow (y_2 \wedge \neg x_2))$ into CNF as follows. The truth table for ϕ_1' appears in figure.

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

$A \rightarrow B$ ✓

A false
B true

The DNF formula equivalent to $\neg\phi_1'$ is

$(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee$
$(y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)$

Negating and applying DeMorgan's laws, we get the CNF formula

$$\phi_1'' = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge$$
$$(\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2).$$

which is equivalent to the original clause ϕ_1'.

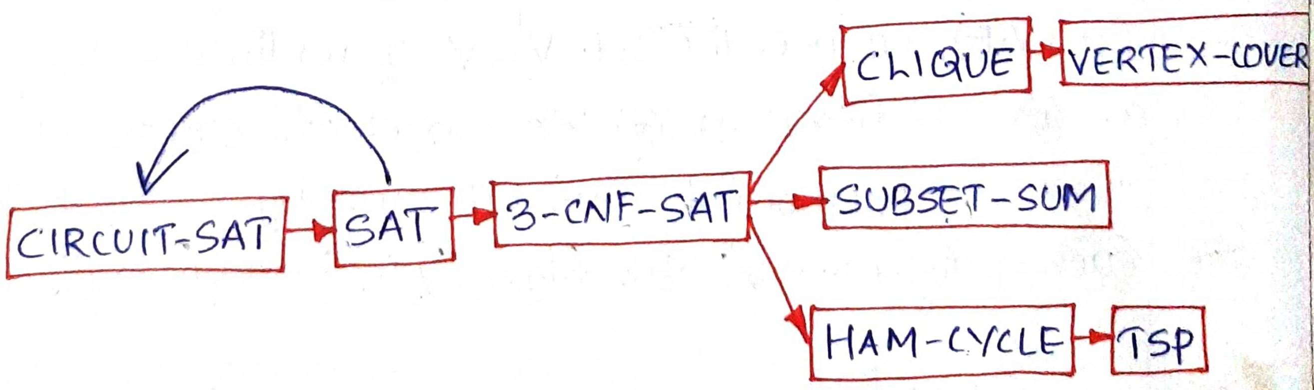
At this point, we have converted each clause ϕ_i' of the formula ϕ' into a CNF formula ϕ_i'', and thus ϕ' is equivalent to the CNF formula ϕ'' consisting of the conjunction of the ϕ_i''. Moreover each clause of ϕ'' has at most 3 literals.

The third and final step of the reduction further transforms the formula so that each clause has exactly 3 distinct literals. We construct the final 3-CNF formula ϕ''' from the clause of the CNF formula ϕ''. The formula ϕ''' also uses 2 auxiliary variables that we shall call p and q.

For each clause C_i of ϕ'', we include the following clauses in ϕ''':

- If C_i has 3 distinct literals, then simply includes C_i as a clause of ϕ'''.
- If C_i has 2 distinct literals, that is, if $C_i = (l_1 \vee l_2)$, where l_1 and l_2 are literals, then include $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$ as clauses of ϕ'''. The literals p and $\neg p$ merely fulfill the syntactic requirements that each clause of ϕ''' has exactly 3 distinct literals. Whether $p=0$ or $p=1$, one of the clauses is equivalent to $l_1 \vee l_2$, and the other evaluates to 1, which is the identity for AND.
- If C_i has just 1 distinct literal l, then include $(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$ as clauses of ϕ'''.
  Regardless of the values of p and q, one of the four clauses is equivalent to l, and the other 3 equivalent to 1.

These steps of the algorithm preserve satisfiability.
Thus, SAT $\leq_p$ 3-CNF-SAT.



All proofs ultimately follow by reduction from the NP-completeness of CIRCUIT-SAT.

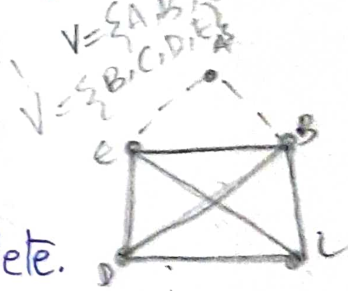
Clique:

A clique in an undirected graph $G=(V,E)$ is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in E. In other words, a clique is a complete subgraph of G. The size of a clique is the number of vertices it contains. The clique problem is the optimization problem of finding a clique of maximum size in a graph.

As a decision problem, we ask simply whether a clique of a given size k exists in the graph.

The formal definition is

CLIQUE $= \{\langle G, k\rangle : G$ is a graph containing a clique of size $k\}$



Theorem

The clique problem is NP-complete.

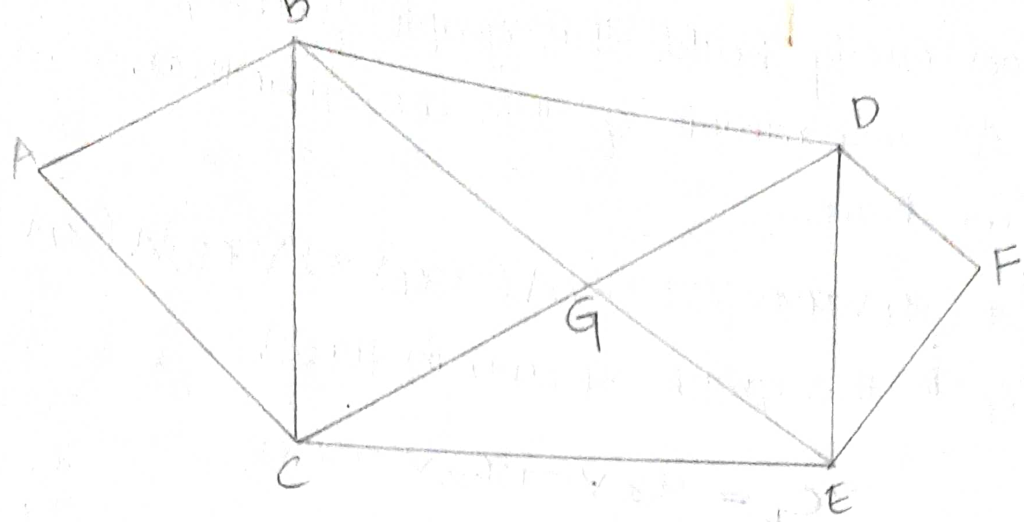
Proof

To show that CLIQUE $\in$ NP, for a given graph $G=(V,E)$, we use the set $V' \subseteq V$ of vertices in the clique as a certificate for G. We can check whether V' is a clique in polynomial time by checking whether, for each pair $u, v \in V'$, the edge (u,v) belongs to E.

Example:

Fig(a) $V' = \{A, B, D, E, C, A\}$ ✓ → CLIQUE

$V' = \{A, B, G, D, F, G\}$ ✗



Fig(a).

Next prove that 3-CNF-SAT $\leq_p$ CLIQUE, which shows that the clique problem is NP-hard.

The reduction algorithm begins with an instance of 3-CNF-SAT. Let $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ be a boolean formula in 3-CNF with k clauses. For $r = 1, 2 \ldots k$, each clause C_r has exactly three distinct literals l_1^r, l_2^r, and l_3^r. We shall construct a graph G such that ϕ is satisfiable if and only if G has a clique of size k.

We construct the graph $G = (V, E)$ as follows. For each clause $C_r = (l_1^r \vee l_2^r \vee l_3^r)$ in ϕ, we place a triple of vertices v_1^r, v_2^r, and v_3^r into V. We put an edge between two vertices v_i^r and v_j^s if both of the following hold:

- v_i^r and v_j^s are in different triples, that is, $r \neq s$, and
- their corresponding literals are consistent, that is, l_i^r is not the negation of l_j^s.

We can easily build this graph from ϕ in polynomial time. As an example of this construction, if we have

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

then G is the graph shown in fig (b).

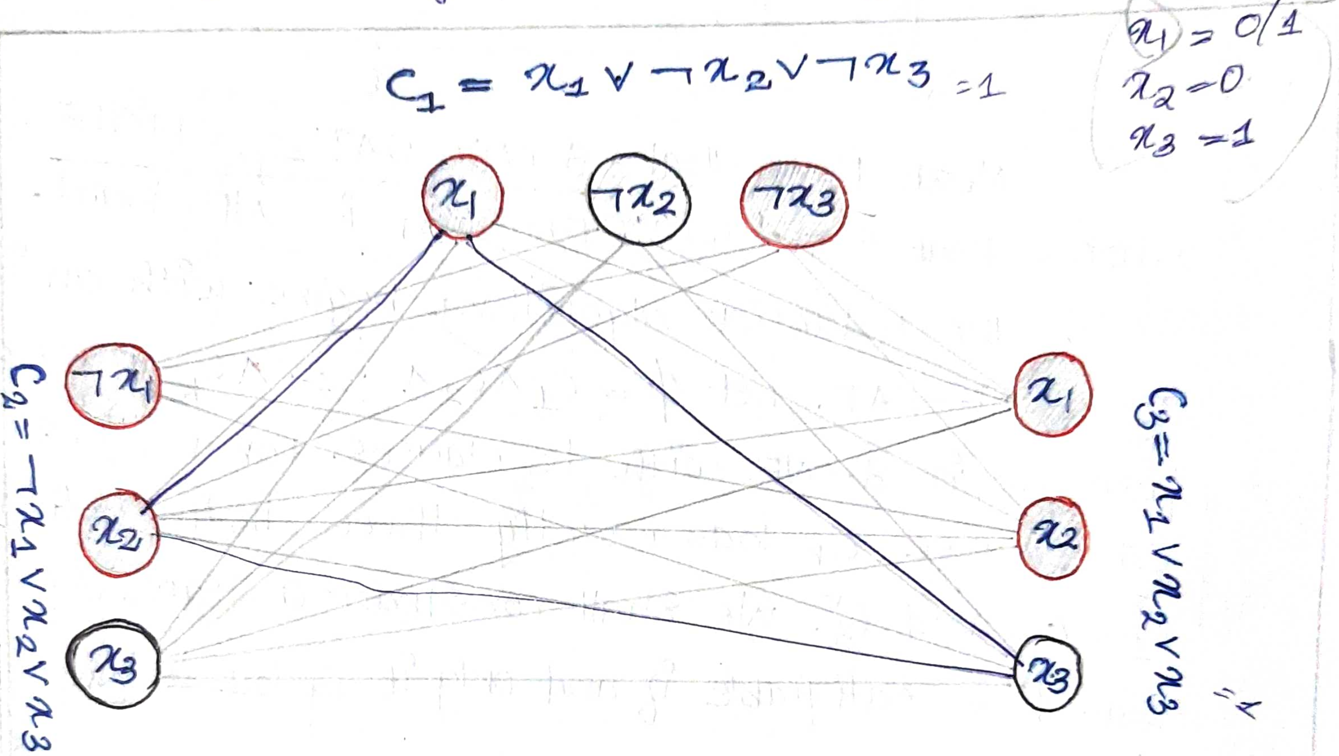


Fig (b): The graph G derived from the 3-CNF formula $\phi = C_1 \wedge C_2 \wedge C_3$, where $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee x_2 \vee x_3)$, and $C_3 = (x_1 \vee x_2 \vee x_3)$, in reducing 3-CNF-SAT to CLIQUE.
A satisfying assignment of the formula has $x_2 = 0$, $x_3 = 1$, and x_1 either 0 or 1. This assignment satisfies C_1 with $\neg x_2$, and it satisfies C_2 and C_3 with x_3, corresponding to the clique with lightly shaded vertices.

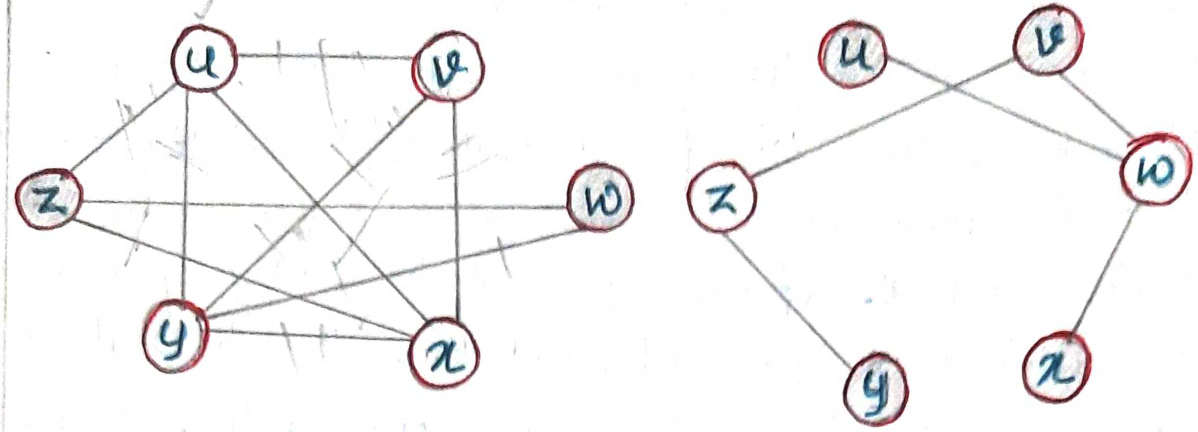
We must show that this transformation of ϕ into G is a reduction. First, suppose that ϕ has a satisfying assignment. Then each clause C_r contains at least one literal l_i^r that is assigned 1, and each such literal corresponds to a vertex v_i^r. Picking one such 'true' literal from each clause yields a set V' of k vertices. We claim that V' is a clique. For any two vertices $v_i^r, v_j^s \in V'$, where $r \neq s$, both corresponding literals l_i^r and l_j^s map to 1 by the given satisfying assignment, and thus the literals cannot be complements. Thus, by the construction of G, the edge (v_i^r, v_j^s) belongs to E.

Conversely, suppose that G has a clique V' of size k. No edges in G connect vertices in the same triple, and so V' contains exactly one vertex per triple. We can assign 1 to each literal l_i^r such that $v_i^r \in V'$ without fear of assigning 1 to both a literal and its complement, since G contains no edges between inconsistent literals. Each clause is satisfied, and so ϕ is satisfied.

The Vertex Cover Problem:

A vertex cover of an undirected graph $G=(V,E)$ is a subset $V' \subseteq V$ such that if $(u,v) \in E$, then $u \in V'$ or $v \in V'$. That is, each vertex 'covers' its incident edges, and a vertex cover for G is a set of vertices that covers all the edges in E.

The size of a vertex cover is the number of vertices in it. For example, the graph in fig (b) has a vertex cover $\{w, z\}$ of size 2.



(a) (b)

Fig: Reducing CLIQUE to VERTEX-COVER.

(a) An undirected graph $G = (V, E)$ with clique $V' = \{u, v, x, y\}$.

(b) The graph $\overline{G}$ produced by the reduction algorithm that has vertex cover $V - V' = \{w, z\}$

The vertex-cover problem is to find a vertex cover of minimum size in a given graph. Restating this optimization problem as a decision problem, we wish to determine whether a graph has a vertex cover of a given size k.

As a language, we define

VERTEX-COVER $= \{\langle G, k \rangle :$ graph G has a vertex cover of size k $\}$

Theorem

The vertex-cover problem is NP-complete.

Proof

We first show that VERTEX-COVER $\in$ NP.

Suppose we are given a graph $G = \{V, E\}$ and an integer k. The certificate we choose is the vertex cover $V' \subseteq V$ itself. The verification algorithm affirms that $|V'| = k$, and then it checks, for each edge $(u, v) \in E$, that $u \in V'$ or $v \in V'$. We can easily verify the certificate in polynomial time.

We prove that the vertex-cover problem is NP-hard by showing that CLIQUE $\leq_p$ VERTEX-COVER. This reduction relies on the notion of the 'complement' of a graph. Given an undirected graph $G = (V, E)$, we define the complement of G as $\overline{G} = (V, \overline{E})$, where $\overline{E} = \{(u, v) : u, v \in V, u \neq v, \text{and } (u, v) \notin E\}$. In other words, $\overline{G}$ is the graph containing exactly those edges that are not in G. Figure (a) and (b) shows a graph and its complement and illustrates the reduction from CLIQUE to VERTEX-COVER.

The reduction algorithm takes as input an instance $\langle G, k \rangle$ of the clique problem. It computes the complement $\overline{G}$, which we can easily do in polynomial time. The output of the reduction algorithm is the instance $\langle \overline{G}, |V| - k \rangle$ of the vertex-cover problem. To complete the proof, we show that this transformation is indeed a reduction: the graph G has a clique of size k if and only if the graph $\overline{G}$ has a vertex cover of size $|V| - k$.

Subset-Sum Problem:

In the subset-sum problem, we are given a finite set S of positive integers and an integer target $t > 0$ we ask whether there exists a subset $S' \subseteq S$ whose elements sum to t.

For example,

If $S = \{1, 2, 7, 9, 10, 18, 21, 29, 56, 100\}$

and $t = \{5846\}$

then the subset $S' = \{1, 7, 9, 10, 100\}$ is a solution.

As usual, we define the problem as language:

$$\text{SUBSET-SUM} = \{\langle S,t\rangle : \text{there exists a subset } S' \subseteq S \text{ such that } t = \textstyle\sum_{s \in S'} s\}.$$

Theorem

The subset-sum problem is NP-Complete.

Proof

To show that SUBSET-SUM is in NP, for an instance $\langle S,t\rangle$ of the problem, we let the subset S' be the certificate. A verification algorithm can check whether $t = \sum_{s \in S'} s$ in polynomial time.

We now show that 3-CNF-SAT $\leq_p$ SUBSET-SUM.

Given a 3-CNF formula ϕ over variables $x_1, x_2 \cdots x_n$
with clauses $C_1, C_2 \cdots C_k$, each containing exactly
3 distinct literals, the reduction algorithm constructs
an instance $\langle S,t \rangle$ of the subset-sum problem such
that ϕ is satisfiable if and only if there exist a
Subset of S whose sum is exactly t.

Without loss of generality, we make two
simplifying assumptions about the formula ϕ.

1. No clause contains both a variable and
its negation, for such a clause is automatically satisfied by any assignment of values
to the variables.

2. Each variable appears in at least one
clause, because it doesnot matter what
value is assigned to a variable that appears
in no clauses.

The reduction creates 2 numbers in set S for each
variable x_i (v_i and v_i') and 2 numbers in S for each
clause C_j (s_j and s_j').

Fig (a) shows, we construct set S and target t
as follows. We label each digit position by either a
variable or a clause. The least significant k ~~bits are~~
digits are labeled by the clauses, and the most significant n digits are labeled by variables.

		n-bits variables			k-bits clauses			
		x_1	x_2	x_3	C_1	C_2	C_3	C_4
x_1	v_1 =	1	0	0	1	0	0	1
	v_1' =	1	0	0	0	1	1	0
x_2	v_2 =	0	1	0	0	0	0	1
	v_2' =	0	1	0	1	1	1	0
x_3	v_3 =	0	0	1	0	0	1	1
	v_3' =	0	0	1	1	1	0	0
C_1	s_1 =	0	0	0	1	0	0	0
	s_1' =	0	0	0	2	0	0	0
C_2	s_2 =	0	0	0	0	1	0	0
	s_2' =	0	0	0	0	2	0	0
C_3	s_3 =	0	0	0	0	0	1	0
	s_3' =	0	0	0	0	0	2	0
C_4	s_4 =	0	0	0	0	0	0	1
	s_4' =	0	0	0	0	0	0	2
	t =	1	1	1	4	4	4	4

Fig:The reduction of 3-CNF-SAT to SUBSET-SUM. The formula in 3-CNF is $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$, where $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee \neg x_2 \vee \neg x_3)$, $C_3 = (\neg x_1 \vee \neg x_2 \vee x_3)$, $C_4 = (x_1 \vee x_2 \vee x_3)$. A satisfying assignment of ϕ is $\langle x_1 = 0, x_2 = 0, x_3 = 1 \rangle$. The set $S = \{1001001, 1000110, 100001, 101110, 10011, 11100, 1000, 2000, 100, 200, 10, 20, 1, 2\}$. The target, $t = 1114444$. The subset $S' \subseteq S$ is not shaded, and it contains v_1', v_2' and v_3, corresponding to the Satisfying assignment. It also contains slack variables $s_1, s_1', s_2', s_3, s_4$ and s_4' to achieve the target value of 4 in the digits labeled by C_1 through C_4.

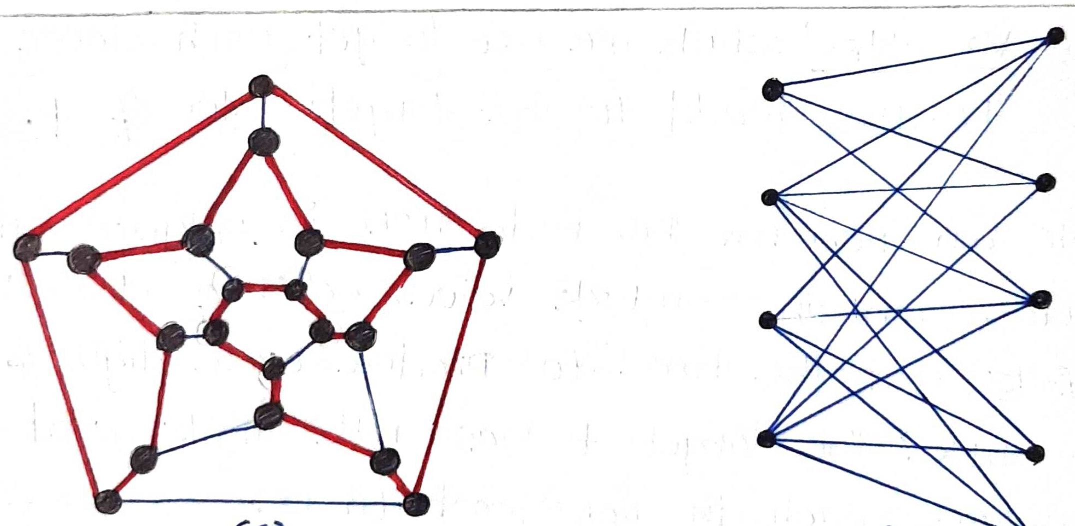
- The target t has a 1 in each digit labeled by a variable and a 4 in each digit labeled by a clause.
- For each variable x_i, set S contains two integers v_i & v_i'. Each of v_i and v_i' has a 1 in the digit labeled by x_i and 0s in the other variable digits. If literal x_i appears in clause C_j, then the digit labeled by C_j in v_i contains a 1. All other digits labeled by clauses in v_i and v_i' are 0.
- For each clause C_j, set S contains two integers s_j and s_j'. Each of s_j and s_j' has 0s in all digits other than the one labeled by C_j. For s_j, there is a 1 in the C_j digit, and s_j' has a 2 in this digit. These integers are 'slack variables' which we use to get each clause-labeled digit position to add to the target value of 4.

We can perform the reduction in polynomial time. The set S contains $2n+2k$ values, each of which has $n+k$ digits, and the time to produce each digit is polynomial in $n+k$. The target t has $n+k$ digits, and the reduction produces each in constant time.

Hamiltonian Cycle Problem:

A hamiltonian cycle of an undirected graph $G=(V,E)$ is a simple cycle that contains each vertex in V. A graph that contains a hamiltonian cycle is said to be hamiltonian; otherwise, it is nonhamiltonian.

The name honors W.R. Hamilton, who described a mathematical game on the dodecahedron in which one player sticks five pins in any five consecutive vertices and the other player must complete the path to form a cycle containing all the vertices. The dodecahedron is hamiltonian, however.



(a) (b)

Fig: (a) A graph representing the vertices, edges, and faces of a dodecahedron, with a hamiltonian cycle shown by red edges.

(b) A bipartite graph with an odd number of vertices. Any such graph is nonhamiltonian.

We can define the hamiltonian-cycle problem, 'Does a graph G have a hamiltonian cycle ?'

As a formal language:

HAM-CYCLE = {<G> : G is a hamiltonian graph}

Theorem

The hamiltonian cycle problem is NP-complete.

Proof

We first show that HAM-CYCLE ∈ NP. Given a graph
G=(V,E), our certificate is the sequence of |V| vertices that
makes up the hamiltonian cycle. The verification
algorithm checks that this sequence contains each
vertex in V exactly once and that with the first
vertex repeated at the end, it forms a cycle in G.
~~That is, there is an edge between~~ That is, it checks
that there is an edge between each pair of consecutive
vertices and between the first and last vertices. We
can verify the certificate in polynomial time.

We now prove that VERTEX-COVER $\le_p$ HAM-CYCLE,
which shows that HAM-CYCLE is NP-complete.
Given an undirected graph G=(V,E) and an integer
k, we construct an undirected graph G'=(V', E') that
has a hamiltonian cycle if and only if G has a vertex
cover of size k.

Our construction uses a widget, which is a piece of a graph that enforces certain properties.

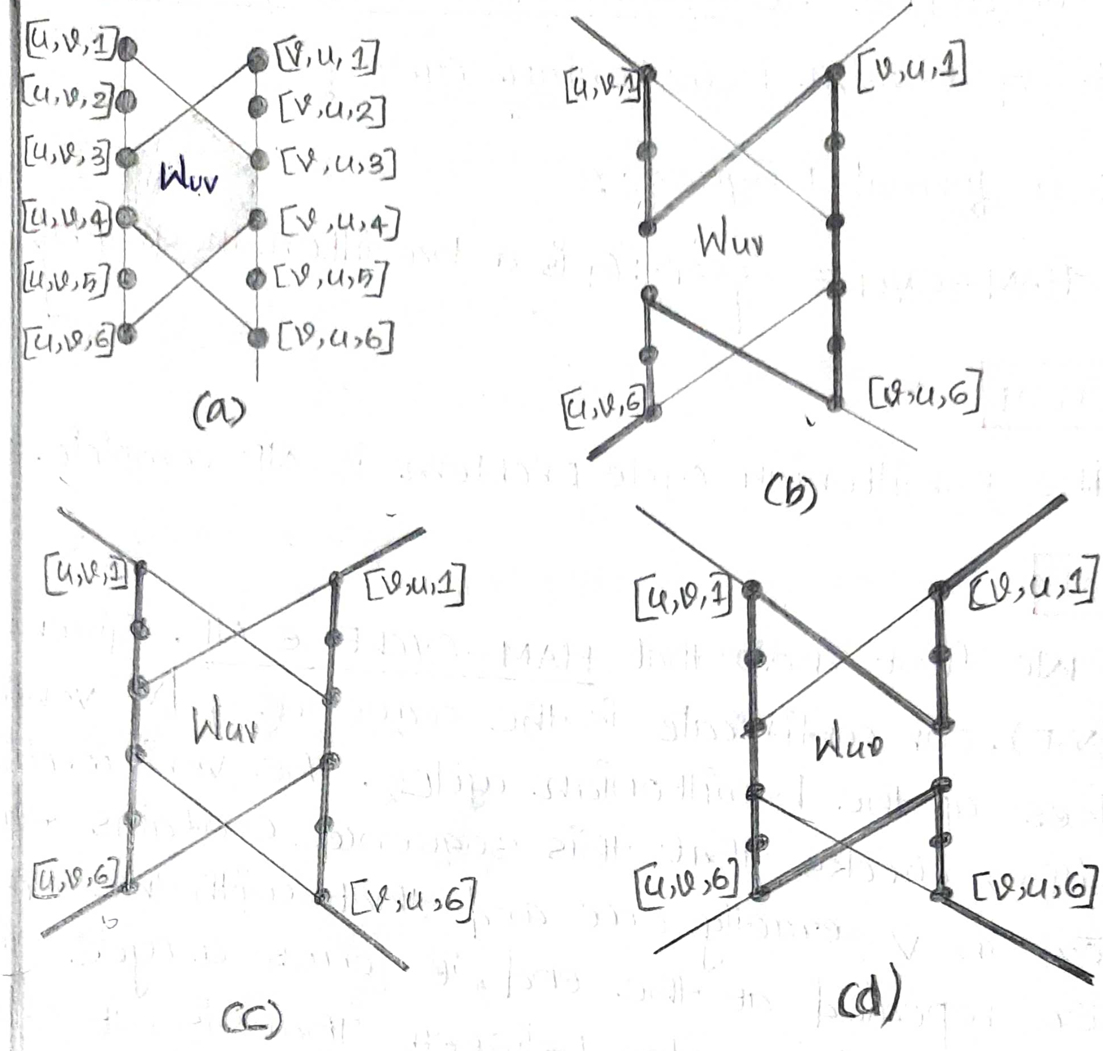


Fig: The widget used in reducing the vertex-cover problem to the hamiltonian-cycle problem. An edge (u,v) of graph G corresponds to widget W_{uv} in the graph G' created in the reduction.
(a) The widget, with individual vertices labeled.
(b) – (d) The shaded paths are the only possible ones through the widget that includes all vertices, assuming that the only connections from the widget to the remainder of G' are through vertices $[u,v,1]$, $[u,v,6]$, $[v,u,1]$, and $[v,u,6]$.

Fig (a) shows the widget we use. For each edge $(u,v) \in E$, the graph G' that we construct will contain one copy of this widget, which we denote by W_{uv}.

We denote each vertex in W_{uv} by $[u,v,i]$ or $[v,u,i]$, where $1 \le i \le 6$, so that each widget W_{uv} contains 12 vertices. Widget W_{uv} also contains the 14 edges.

Along with the internal structure of the widget, we enforce the properties we want by limiting the connection b/w the widget and the remainder of the graph G' that we construct. In particular, only vertices $[u,v,1]$, $[u,v,6]$, $[v,u,1]$ and $[v,u,6]$ will have edges incident from outside W_{uv}.

Any hamiltonian cycle of G' must traverse the edges of W_{uv} in one of the 3 ways shown in fig (b)–(d). If the cycle enters through vertex $[u,v,1]$, it must exit through vertex $[u,v,6]$, and it either visits all 12 of the widget's vertices or the 6 vertices $[u,v,1]$ through $[u,v,6]$ (fig(c)). No other paths through the widget that visits all vertices as possible. In particular, it is impossible to construct two vertex-disjoint paths, one of which connects $[u,v,1]$ to $[v,u,6]$ and the other of which connects $[v,u,1]$ to $[u,v,6]$, such that the union of the two paths contains all of the widgets vertices.

The only other vertices in V' other than those of widgets are selector vertices $s_1, s_2 \dots s_k$. We use edges incident on selector vertices in G' to select the k vertices of the cover in G.

Reducing an instance of the vertex-cover problem to an instance of the hamiltonian cycle problem.

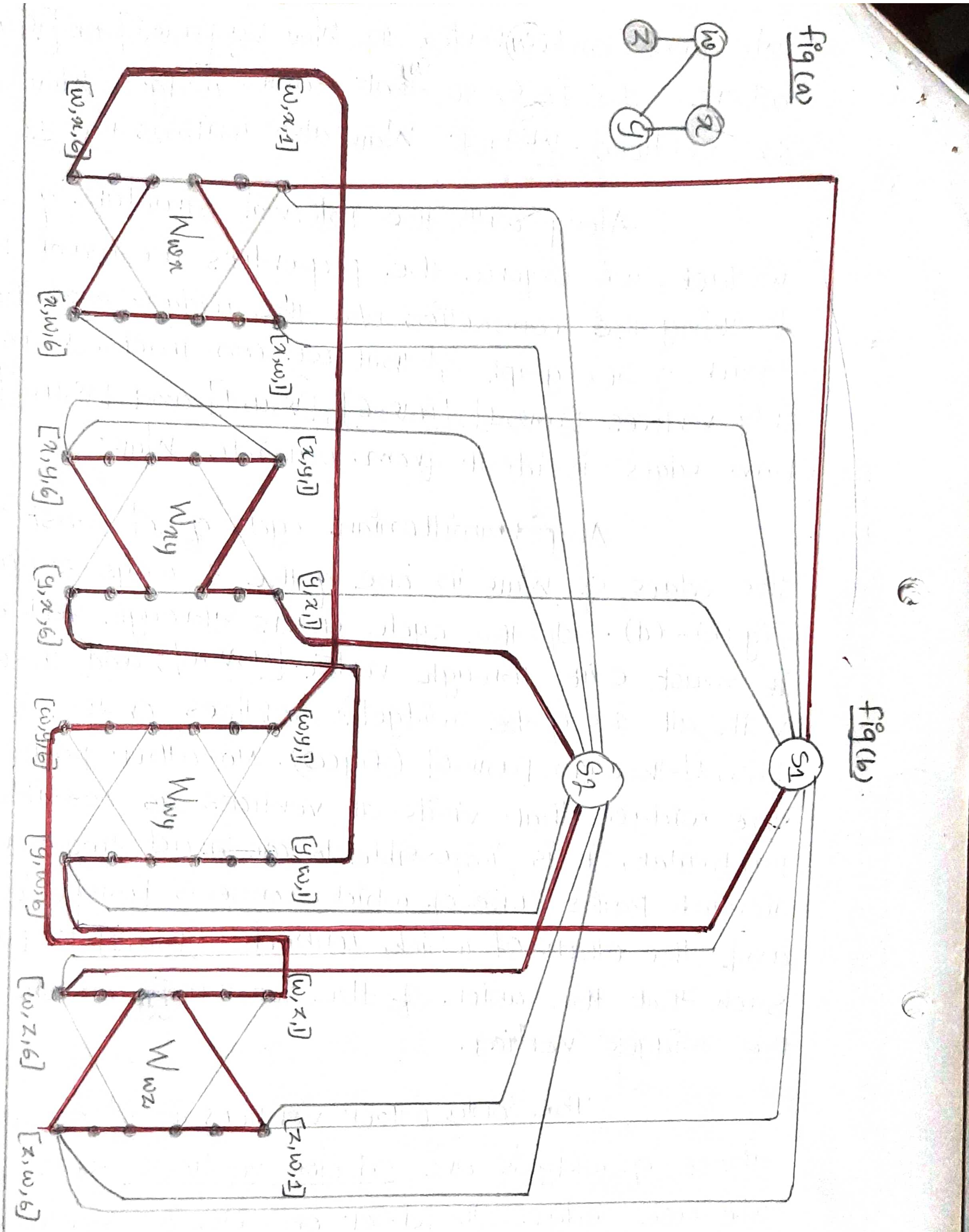


Fig: Reducing an instance of the vertex-cover problem to an instance of the hamiltonian-cycle problem. (a) An undirected graph G with a vertex cover of size 2, consisting of the lightly shaded vertices w and y. (b) The undirected graph G' Produced by the reduction, with the hamiltonian path correspo-nding to the vertex cover shaded. The vertex-cover $\{w, y\}$ corresponds to edges $(S_1, [w, x, 1])$ & $(S_2, [y, x, 1])$ appearing hamilton cycle.

The Travelling – Salesman Problem:

In the travelling-salesman problem, which is closely related to the hamiltonian cycle problem, a salesman must visit 'n' cities. Modeling the Problem as a complete graph with 'n' vertices, we can say that salesman wishes to make a tour, or hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from.

The salesman incurs a nonnegative integer cost $c(i,j)$ to travel from city 'i' to city 'j', and the salesman wishes to make the tour whose total cost is minimum, where the total cost is the sum of the individual costs along the edges of the tour.

Example:

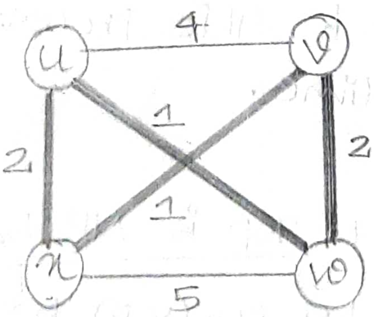


Fig: An instance of the traveling-salesman problem. Shaded edges represent a minimum cost tour, with cost 7.

The Formal language for the corresponding decision problem is

$TSP = \{\langle G, c, k\rangle : G = \langle V, E\rangle$ is a complete graph,

c is a function from $V \times V \to \mathbb{Z}$,

$k \in \mathbb{Z}$, and

G has a traveling-salesman tour with cost atmost $k\}$

Theorem

The traveling-salesman problem is NP-complete

Proof

We first show that TSP belongs to NP. Given an instance of the problem, we use as a certificate the sequence of 'n' vertices in the tour. The verification algorithm checks that this sequence contains each vertex exactly once, sums up the edge costs, and checks whether the sum is at most k. This process can certainly be done in polynomial time.

To prove that TSP is NP-hard, we show that HAM-CYCLE $\leq_p$ TSP. Let $G = (V, E)$ be an instance of HAM-CYCLE. We construct an instance of TSP as follows. We form the complete graph $G' = (V, E')$, where $E' = \{(i,j) : i,j \in V \text{ and } i \neq j\}$, and we define the cost function 'c' by

$$c(i,j) = \begin{cases} 0 & \text{if } (i,j) \in E, \\ 1 & \text{if } (i,j) \notin E. \end{cases}$$

We now show that graph G has a hamiltonian cycle if and only if graph G' has a tour of cost at most 0. Suppose that graph G has a hamiltonian cycle h, each edge in 'h' belongs to E and thus has cost 0 in G'. Thus, h is a tour in G' with cost 0. Conversely, Suppose that graph G' has a tour h' of cost at most 0. Since the costs of the edges in E' are 0 and 1, the cost of tour h' is exactly 0 and each edge on the tour must have cost 0. Therefore, h' contains only edges in E. We conclude that h' is a hamiltonian cycle in graph G.

CONTENT BEYOND SYLLABUS

RANDOMIZED ALGORITHMS

A *randomized algorithm* is an algorithm that employs a degree of randomness as part of its logic. The algorithm typically uses uniformly random bits as an auxiliary input to guide its behavior, in the hope of achieving good performance in the "average case" over all possible choices of random determined by the random bits; thus either the running time, or the output (or both) are random variables.

One has to distinguish between algorithms that use the random input so that they always terminate with the correct answer, but where the expected running time is finite (Las Vegas algorithms, for example Quicksort), and algorithms which have a chance of producing an incorrect result (Monte Carlo algorithms, for example the Monte Carlo algorithm for the MFAS problem) or fail to produce a result either by signaling a failure or failing to terminate. In some cases, probabilistic algorithms are the only practical means of solving a problem.

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(nLogn).

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 Fermet Method for Primality Testing.

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 expect 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.