CS8451 DESIGN AND ANALYSIS OF ALGORITHMS

OBJECTIVES:

- To understand and apply the algorithm analysis techniques.
- To critically analyze the efficiency of alternative algorithmic solutions for the same problem
- To understand different algorithm design techniques.
- To understand the limitations of Algorithmic power.

UNIT I INTRODUCTION 9

Notion of an Algorithm – Fundamentals of Algorithmic Problem Solving – Important Problem Types – Fundamentals of the Analysis of Algorithmic Efficiency –Asymptotic Notations and their properties. Analysis Framework – Empirical analysis - Mathematical analysis for Recursive and Non-recursive algorithms - Visualization

UNIT II BRUTE FORCE AND DIVIDE-AND-CONQUER 9

Brute Force – Computing an – String Matching - Closest-Pair and Convex-Hull Problems - Exhaustive Search - Travelling Salesman Problem - Knapsack Problem - Assignment problem. Divide and Conquer Methodology – Binary Search – Merge sort – Quick sort – Heap Sort - Multiplication of Large Integers – Closest-Pair and Convex - Hull Problems.

UNIT III DYNAMIC PROGRAMMING AND GREEDY TECHNIQUE 9

Dynamic programming – Principle of optimality - Coin changing problem, Computing a Binomial Coefficient – Floyd‘s algorithm – Multi stage graph - Optimal Binary Search Trees – Knapsack Problem and Memory functions. Greedy Technique – Container loading problem - Prim‘s algorithm and Kruskal's Algorithm – 0/1 Knapsack problem, Optimal Merge pattern - Huffman Trees.

UNIT IV ITERATIVE IMPROVEMENT 9

The Simplex Method - The Maximum-Flow Problem – Maximum Matching in Bipartite Graphs, Stable marriage Problem.

UNIT V COPING WITH THE LIMITATIONS OF ALGORITHM POWER 9

Lower - Bound Arguments - P, NP NP- Complete and NP Hard Problems. Backtracking – n-Queen problem - Hamiltonian Circuit Problem – Subset Sum Problem. Branch and Bound – LIFO Search and FIFO search - Assignment problem – Knapsack Problem – Travelling Salesman Problem - Approximation Algorithms for NP-Hard Problems – Travelling Salesman problem – Knapsack problem.

TOTAL: 45 PERIODS

OUTCOMES:

At the end of the course, the students should be able to:

- Design algorithms for various computing problems.
- Analyze the time and space complexity of algorithms.
- Critically analyze the different algorithm design techniques for a given problem.
- Modify existing algorithms to improve efficiency.

TEXT BOOKS:

1. Anany Levitin, ―Introduction to the Design and Analysis of Algorithms‖, Third Edition, Pearson Education, 2012.
2. Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran, Computer Algorithms/ C++, Second Edition, Universities Press, 2007.

REFERENCES:

1. Thomas H.Cormen, Charles E.Leiserson, Ronald L. Rivest and Clifford Stein, ―Introduction to Algorithms‖, Third Edition, PHI Learning Private Limited, 2012.
2. Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, ―Data Structures and Algorithms‖, Pearson Education, Reprint 2006.
3. Harsh Bhasin, ―Algorithms Design and Analysis‖, Oxford university press, 2016.
4. S. Sridhar, ―Design and Analysis of Algorithms‖, Oxford university press, 2014.
5. http://nptel.ac.in/

UNIT-1
INTRODUCTION

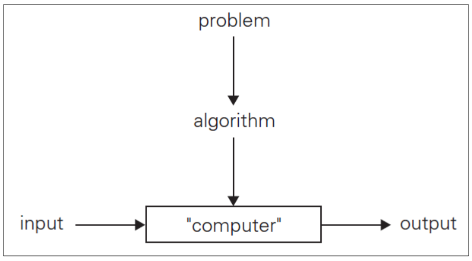
Notion of an Algorithm – Fundamentals of Algorithmic Problem Solving – Important Problem Types – Fundamentals of the Analysis of Algorithmic Efficiency – Asymptotic Notations and their properties. Analysis Framework – Empirical analysis - Mathematical analysis for Recursive and Non-recursive algorithms – Visualization.

1.1 Notion of an Algorithm

1.1.1 Algorithm

- Algorithm is a sequence of unambiguous instructions for solving a problem i.e) for obtaining a required output for any legitimate input in a finite amount of time.

- **Diagram:** The notion of the Algorithm



1.1.2 Need for the analysis of Algorithms:

Example: - computing the greatest common divisor of two integers:

gcd(m,n) – defined as the largest integer that divides both m and n evenly.

Three methods for solving the same problem:

1. Euclid's Algorithm
2. Consecutive Integer Checking Algorithm
3. Middle School Procedure

Method 1: Euclid's Algorithm

Step 1 : If n = 0, return the value of m as the answer and stop; otherwise, proceed to Step 2.
Step 2 : Divide m by n and assign the value of the remainder to r.
Step 3 : Assign the value of n to m and the value of r to n. Go to Step 1.

$$\boxed{\gcd(m,n) = \gcd(n, m \bmod n)}$$

- based on applying repeatedly the equality

Example:

gcd(60,24) = gcd(24,12) = gcd(12,0) = 12

pseudocode:

```
ALGORITHM Euclid(m, n)
//Computes gcd(m, n) by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers m and n
//Output: Greatest common divisor of m and n
    while n ≠ 0 do
        r ←m mod n
        m←n
        n←r
    return m
```

Method 2: Consecutive Integer Checking Algorithm

Step 1: Assign the value of min{m, n} to t.
Step 2: Divide m by t. If the remainder of this division is 0, go to Step 3; otherwise, go to Step 4.
Step 3: Divide n by t. If the remainder of this division is 0, return the value of t as the answer and stop; otherwise, proceed to Step 4.
Step 4: Decrease the value of t by 1. Go to Step 2.

-> **more complex and slower than Euclid's Algorithm**

Example: **gcd(60,24)**
Step 1: t = min{60,24} = 24 ; m=60 ; n=24
Step 2 : Divide m by t;
Divide 60 by 24 ; remainder $\neq 0$; Decrease the value of 24 by 1 i.e) 23.
Divide 60 by 23 ; remainder $\neq 0$; Decrease the value of 23 by 1 i.e) 22
Divide 60 by 22 ; remainder $\neq 0$; Decrease the value of 22 by 1 i.e) 21
Divide 60 by 21 ; remainder $\neq 0$; Decrease the value of 21 by 1 i.e) 20
Divide 60 by 20 ; remainder $= 0$;
now t=20
Step 3: Divide n by t;
Divide 24 by 20; remainder $\neq 0$; Decrease the value of 20 by 1 i.e) 19.
Divide m by t;
Divide 60 by 19; remainder $\neq 0$; Decrease the value of 19 by 1 i.e) 18.
Divide 60 by 18; remainder $\neq 0$; Decrease the value of 18 by 1 i.e) 17.
Divide 60 by 17; remainder $\neq 0$; Decrease the value of 17 by 1 i.e) 16.
Divide 60 by 16; remainder $\neq 0$; Decrease the value of 16 by 1 i.e) 15.
Divide 60 by 15; remainder $= 0$;
now t=15
Step 4: Divide n by t;
Divide 24 by 15; remainder $\neq 0$; Decrease the value of 15 by 1 i.e) 14.
Divide m by t;
Divide 60 by 14; remainder $\neq 0$; Decrease the value of 14 by 1 i.e) 13.
Divide 60 by 13; remainder $\neq 0$; Decrease the value of 13 by 1 i.e) 12.
Divide 60 by 12; remainder $= 0$;
Step 4: Divide n by t;
Divide 24 by 12; remainder $= 0$;
Step 5: Return the value of t as answer: t = 12;
So **gcd(60,24) = 12.**

Method 3: Middle School Procedure .

Step 1: Find the prime factors of m.
Step 2: Find the prime factors of n.
Step 3: Identify all the common factors in the two prime expansions found in Step 1 and Step 2.
(If p is a common factor occurring p_m and p_n times in m and n, respectively, it should be repeated min{p_m, p_n} times.)
Step 4: Compute the product of all the common factors and return it as the greatest common divisor of the numbers given.

Example: **gcd(60,24)**
Step 1: the prime factors of 60 = 2 . 2 . 3 . 5
Step 2: the prime factors of 24 = 2 . 2 . 2 . 3
Step 3: Identify all the common factors : 2, 2, 3
Step 4: Compute the product of all the common factors and return;
gcd(60, 24) = 2 . 2 . 3 = 12.

Finding Prime Numbers: (sieve of Eratosthenes)

- simple algorithm for generating consecutive primes not exceeding any given integer n > 1.
- It was probably invented in ancient Greece and is known as the **sieve of Eratosthenes**
- **Steps:**
    - The algorithm starts by initializing a list of prime candidates with consecutive integers from 2 to n.
    - Then, on its first iteration, the algorithm eliminates from the list all multiples of 2, i.e., 4, 6, and so on.
    - Then it moves to the next item on the list, which is 3, and eliminates its multiples.
    - No pass for number 4 is needed: since 4 itself and all its multiples are also multiples of 2, they were already eliminated on a previous pass.
    - The next remaining number on the list, which is used on the third pass, is 5.
    - The algorithm continues in this fashion until no more numbers can be eliminated from the list.
    - The remaining integers of the list are the primes needed.
        - **Example:** the algorithm to finding the list of primes not exceeding n = 25:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	3		5		7		9		11		13		15		17		19		21		23		25
2	**3**		5		7				11		13				17		19				23		25
2	3		**5**		7				11		13				17		19				23		

- The remaining numbers on the list are the consecutive primes less than or equal to 25.

ALGORITHM Sieve(n)
//Input: A positive integer n > 1
//Output: Array L of all prime numbers less than or equal to n
for p←2 **to** n **do** A[p]←p
for p←2 **to** $\sqrt{n}$
    do
        if A[p] ≠ 0   //p hasn't been eliminated on previous passes
            j ← p ∗ p
        while j ≤ n **do**
            A[j]←0   //mark element as eliminated
            j ←j + p
            //copy the remaining elements of A to array L of the primes
        i ←0
        for p←2 **to** n **do**
            if A[p] ≠ 0
            L[i]←A[p]
            i ←i + 1
return L

Example:

List the prime numbers not exceeding 10

Step 1:	2	3	4	5	6	7	8	9	10
Step 2:	**2**	3		5		7		9	
Step 3:	2	**3**		5		7			

Execution steps of the Algorithm:

Step 1:

	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
A[P] =	2	3	4	5	6	7	8	9	10

for p ← 2 to $\sqrt{10}$ i.e) p ← 2 to 3 do

Step 2:

p=2

if A[2] ≠ 0 ==> 2 ≠ 0 // Not eliminated
j = 2 x 2 = 4
while 4 ≤ 10 do
A[4] = 0 // eliminated and j=4+2= 6 i.e) ≤ 10
A[6] = 0 // eliminated and j=6+2= 8 i.e) ≤ 10
A[8] = 0 // eliminated and j=8+2= 10 i.e) ≤ 10
A[10] = 0 // eliminated and j=10+2= 12 i.e) ≥ 10
Hence comes out of while loop and increments "p"

Step 3:

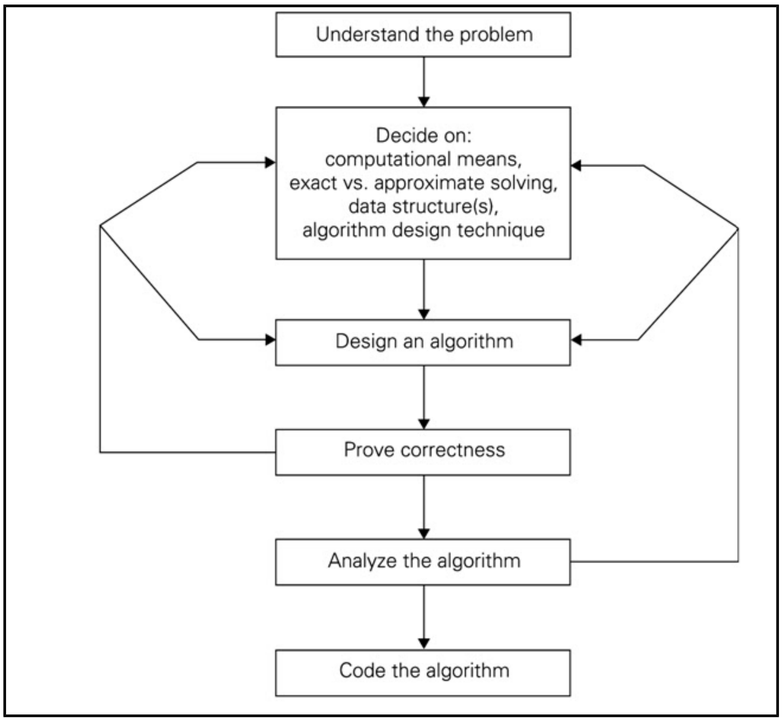
p=3

if A[3] ≠ 0 ==> 3 ≠ 0 // Not eliminated
j = 3 x 3 = 9
while 9 ≤ 10 do
A[9] = 0 // eliminated and j=9+3= 12 i.e) ≥ 10
Hence comes out of while loop and increments "p"

- Now, After elimination, the array A contains only prime numbers which is copied to the array L.

1.2. Fundamentals of Algorithmic problem solving

Algorithm Design and Analysis Process



1) Understand the problem:

- It is done by reading the problem statement thoroughly and ask questions for clarifying the doubts about the problem.
- Find out what are the necessary inputs for solving the problem

2) a) Ascertaining the capabilities of computational devices:

- It is necessary to ascertain (decide) the computational capabilities of devices on which the algorithm will be running.
- From execution point of view algorithm
  1. Sequential algorithm
  2. Parallel algorithm
  - ✔ Sequential algorithm - runs on a machine in which the instructions are executed one after another. Such a machine is called Random Acess Machine(RAM).
  - ✔ Parallel algorithm – Algorithm that take advantage of operations that can be executed concurrently. i.e) The algorithm that can be executed simultaneously on many different processing devices and then combined together to get correct result.
- There are certain problems which require huge amount of memory or the problems for which execution time is an important factor.
- For such problems it is essential to have a proper choice of a computational device which is space and time efficient.

b) Choosing between exact and approximate problem solving:

- To decide whether the problems is to be solved exactly or approximately.

i) exact algorithm – solving the problem exactly

ii) approximation algorithm - solving the problem approximately.

Ex: Travelling salesman problem – finding shortest tour through n cities

c) Deciding on Appropriate Data Structures:

- Data Structure is important for both design and analysis of algorithm
- Choice of proper data structure is required

Algorithms + Data Structures = Programs

- Data structure and algorithm work together and these are interdependent.
- Program is possible with the help of algorithm and data structure.

d) Algorithm Design Techniques:

- Algorithm Design Technique is a general approach to solving problems algorithmically.
- They provide guidance for designing algorithms for new problems
- They are used to classify the algorithms based on the design idea.
- Algorithmic strategies also called as algorithmic techniques or algorithmic paradigm.
    - Brute force
    - Divide and conquer
    - Dynamic programming
    - Greedy Technique
    - Back Tracking

3) Methods of specifying an Algorithm:

There are various ways for specifying an algorithm.

- ➢ Using Natural Language – Clear description of an algorithm
- ➢ Pseudo Code – mixture of natural language and programming language
- ➢ Flow Chart – diagramatic representation of an algorithm

4) Proving an Algorithm's correctness:

- to prove the correctness of the algorithm. i.e) to prove that the algorithm yields a required result for every legitimate input in a finite amount of time.
- The common technique is to use mathematical induction (2 Steps)
- If the algorithm is found to be incorrect, it is needed to redesign regarding tha data structures, the design techniques and so on.

5) Analyzing an Algorithm:

- The following factors should be considerd while analysing an algorithm

✔ Time efficiency - Speed (how fast the algorithm runs)
✔ Space efficieny - memory (howmuch memory the algorithm needs)
✔ Simplicity - easy to understand
✔ Generality - which range of input is accepted

6) Coding an Algorithm:

- Programming an algorithm
- transition from an algorithm to a program
- Test and debug the program

1.3. Important Problem Types

1. Sorting
2. Searching
3. String processing
4. Graph problems
5. Combinatorial problems
6. Geometric problems
7. Numerical problems

1) Sorting:

- Rearranging the items of a given list in ascending order
- key – chosen piece of information to sort
- Example: For student records, the key is the alphabets (Name)

- Properties:

  i) stable – preserves the relative order of any two equal elements in its input
  ii) in place – does not require extra memory

2) Searching:

- finding a value (search key) in a given list of elements
- two types:
  1. Sequential Search
  2. Binary Search

3) String processing:

- String – a sequence of characters
- Types:
  1. text string – comprises letters, numbers and special characters
  2. bit string – comprises zeros and ones
  3. gene sequence – strings of characters of {A,C,G,T}
- String Matching – Searching for a given word in a text

4) Graph problems:

- Graph – collection of points(vertices) are connected by line segments(edges)
- used for modeling a variety of real-life applications
- Basic Graph Algorithms:
  1. Graph Traversal Algorithm (visiting all the points in a network)
  2. Shortest path Algorithm (Finding best route between two cities)
  3. Topological sorting for graphs (Ordering the vertices)
- Example:
  - Traveling salesman problems (finding shortest tour through n cities)
  - Graph coloring problem (Assigning smallest number of colors to vertices such that no two adjacent vertices are the same)

5) Combinatorial problems:

- finding a combinatorial objects
- i.e) computing permutations and combinations
- Ex:
  1. Travelling Salesman Problem
  2. Graph Coloring Problem
- difficult problems because the number of combinatorial objects grows extremely fast with a problem's size.

- No known algorithms for solving the problems exactly in an acceptable amount of time
- Many problems are unsolvable problems

6) Geometric problems:

- deal with geometric objects such as points, lines, and polygons
- problems of constructing simple geometric shapes such as triangles, circles and so on.
- Ex:
  1. Closest Pair Problem – finding closest pair among n points
  2. Convex Hull Problem – finding smallest convex polygon

7) Numerical problems:

- problems that involve mathematical objects of continuous nature.
- can be solved only approximately
- Ex:
  - Solving equations and systems of equations
  - Computing definite integrals
  - evaluating functions

1.4. Fundamentals of the Analysis of Algorithmic Efficiency

Analysis of algorithm – investigation of an algorithm's efficiency with respect to two resources:

i) running time

ii) memory space

Efficiency – determined by measuring time and space, the algorithm uses for executing the program

Time Efficiency :

- how fast the algorithm runs
- The time taken by a program to complete its task depends on the number of steps in an algorithm

  Two types:

  Compilation time – time for compilation

  Run Time – Execution time depends on the size of the algorithm

Space Efficiency :

- The number of units the algorithm requires for memory storage

1.4.1 Analysis framework:

Two kinds of Efficiency:

i) Time Efficiency

ii) Space Efficiency

General Framework:

i) Measuring an input's size

ii) Units for measuring Running Time

iii) Ordres of Growth

iv) Worst-case, Best-case and Average – case Efficiency

v) Recapitulation of the Analysis Framework

i) Measuring an input's size:

- Algorithms run longer on larger inputs
- parameter n – indicating the algorithm's input size (Ex: sorting, searching)

- Ex:
- i) problem of evaluating a polynomial $p(x) = a_n x^n + ... + a_0$:
  - input's size – polynomial's degree or number of coefficients
- ii) computing the product of two n-by-n matrices
  - input's size – total number of elements N in the matrices
- Measuring size of the inputs by the number of bits in the n's binary representation:
  - number of bits b; $b = \lfloor \log_2 n \rfloor + 1$

  - Ex:

n	$Log_2 n$	$\lfloor Log_2 n \rfloor$	b
1	0.0000	0	1
9	3.1699	3	4
15	3.9069	3	4

ii) Units for measuring Running Time:

- use standard units of time measurement – seconds, milliseconds

- count the number of times each of the algorithm's operation is executed
    - identify the basic operation (most important operation)
    - number of times the basic operation is executed

- Ex: i) For sorting algorithm, the basic operation is comparison
  ii) For matrix multiplication, the basic operation is multiplication

- Estimating the running time:
  $T(n) \approx C_{op} C(n)$
  C_{op} – Basic operation's execution time
  C(n) – number of times the Basic operation needs to be executed

- 10 times faster machine - 10 times faster
- Double the input – 4 times longer

- Ex:

$$C(n) = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \approx \frac{1}{2}n^2$$

$$\frac{T(2n)}{T(n)} \approx \frac{c_{op}C(2n)}{c_{op}C(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} = 4.$$

iii) Orders of Growth:

- Measuring the performance of an algorithm in relation with input size.

Values (some approximate) of several functions important for analysis of algorithms

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3{\cdot}10^1$	10^2	10^3	10^3	$3.6{\cdot}10^6$
10^2	6.6	10^2	$6.6{\cdot}10^2$	10^4	10^6	$1.3{\cdot}10^{30}$	$9.3{\cdot}10^{157}$
10^3	10	10^3	$1.0{\cdot}10^4$	10^6	10^9		
10^4	13	10^4	$1.3{\cdot}10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7{\cdot}10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0{\cdot}10^7$	10^{12}	10^{18}		

- The function growing the slowest is the logarithmic function.
- the exponential function 2^n and the factorial function n! grow so fast

iv) Worst-case, Best-case and Average – case Efficiency

- Ex: **Sequential Search**
    - searches for a given item (search key K) in a list of n elements by checking successive elements of the list until either a match with the search key is found or the list is exhausted.

- **ALGORITHM** SequentialSearch(A[0..n − 1], K)
  //Searches for a given value in a given array by sequential search
  //Input: An array A[0..n − 1] and a search key K
  //Output: The index of the first element in A that matches K
  // or −1 if there are no matching elements

```
i ←0
while i < n and A[i] ≠ K do
        i ←i + 1
if i < n return i
else return −1
```

- **Worst-case Efficiency** – The worst-case efficiency of an algorithm is its efficiency for the worst-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the longest among all possible inputs of that size.

  $$\mathbf{C_{worst}(n) = n}$$

- **Best-case Efficiency** - The best-case efficiency of an algorithm is its efficiency for the best-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the fastest among all possible inputs of that size.

  $$\mathbf{C_{best}(n) = 1}$$

- **Average-case Efficiency** – make some assumptions about possible inputs of size n
  i) successful search- the probability of the first match occurring in the ith position of the list is p/n
  ii) unsuccessful search - the number of comparisons will be n with the probability (1− p).

$$\begin{aligned} C_{avg}(n) &= [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1-p) \\ &= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1-p) \\ &= \frac{p}{n}\frac{n(n+1)}{2} + n(1-p) = \frac{p(n+1)}{2} + n(1-p). \end{aligned}$$

- Successful search: p=1, The average number of key comparisons is $\frac{n+1}{2}$
- Unsuccessful search: p=0, The average number of key comparisons is n
- the average-case efficiency cannot be obtained by taking the average of the worst-case and the best-case efficiencies.

- **Amortized efficiency:**
  - It applies not to a single run of an algorithm but rather to a sequence of operations performed on the same data structure.
  - The total time for an entire sequence of n such operations is always significantly better than the worst-case efficiency of that single operation multiplied by n.

1.5 Asymptotic Notations and their properties

- To choose best algorithm, it is needed to check the efficiency of the algorithms
- The efficiency of an algorithm can be measured by computing time complexity of each algorithm
- Using asymptotic notations time complexity can be rated as
    1. Fastest Possible
    2. Slowest Possible
    3. Average Time
- asymptotic notations:
    - O (Big – oh)
    - Ω (Big Omega)
    - Θ (Big - Theta)

- t(n) will be an algorithm's running time and
- g(n) will be some simple function to compare the count with.

i) Big – oh Notation (O)

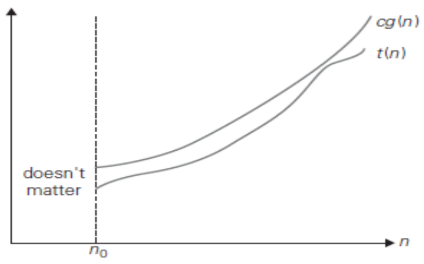
- Method of representing the upper bound of algorithm's running time

Definition:

- A function t(n) is said to be in O(g(n)) denoted as **t(n) ϵ O(g(n))**, if t(n) is **bounded above** by some constant multiple of g(n) for all large n i.e) if there exists some positive constant C and some non-negative integer n_0 such that

$$\mathbf{t(n) \le Cg(n) \text{ for all } n \ge n_0}$$

- **Diagram**



Big-oh notation: $t(n) \in O(g(n))$.

Ex:

t(n) = 4n; g(n) = 5n

ii) Big Omega Notation (Ω)

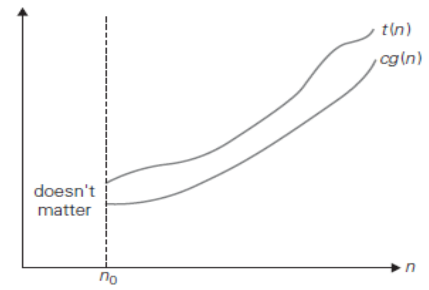
- Method of representing the lower bound of algorithm's running time
- Describes the best case running time of algorithms

Definition:

- A function t(n) is said to be in Ω (g(n)) denoted as **t(n) ϵ Ω(g(n))**, if t(n) is **bounded below** by some positive constant multiple of g(n) for all large n i.e) if there exists some positive constant C and some non-negative integer n_0, such that

$$\mathbf{t(n) \ge Cg(n) \text{ for all } n \ge n_o}$$

- **Diagram:**



Big-omega notation: $t(n) \in \Omega(g(n))$.

Ex:

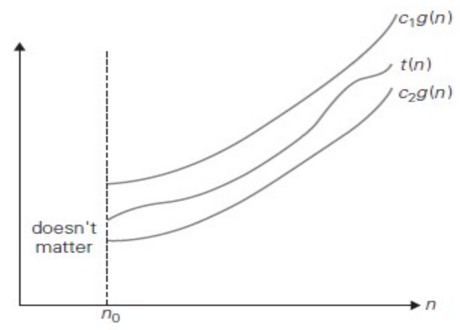
t(n) = 5n; g(n) = 4n

iii) Big - Theta Notation - Θ:

- A function t(n) is said to be in Θ(g(n)),denoted by **t(n) ϵ Θ(g(n))**, if t(n) is **bounded both above and below** by some positive constant multiples of g(n) for all large n i.e) if there exists some positive constants 'C_1' and' C_2' and some non-negative integer n_o such that

$$C_2\, g(n) \le t(n) \le C_1 g(n) \text{ for all } n > n_0$$

- **Diagram:**



Big-theta notation: $t(n) \in \Theta(g(n))$.

Note:

Θ(g(n)) = o(g(n)) ∩ Ω(g(n))

Properties:

Useful Property involving the Asymptotic Notations

- Theorem: If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$.
- Proof **:**

Let, four arbitrary real numbers a_1, b_1, a_2, b_2: if $a_1 \le b_1$ and $a_2 \le b_2$, then $a_1 + a_2 \le 2 \max\{b_1, b_2\}$.
$t_1(n) \in O(g_1(n))$, $t_1(n) \le c_1g_1(n)$ for all $n \ge n_1$ and
$t_2(n) \in O(g_2(n))$, $t_2(n) \le c_2g_2(n)$ for all $n \ge n_2$.
Consider, $c_3 = \max\{c_1, c_2\}$; $n \ge \max\{n_1, n_2\}$

$$\begin{aligned} t_1(n) + t_2(n) &\le c_1g_1(n) + c_2g_2(n) \\ &\le c_3g_1(n) + c_3g_2(n) = c_3[g_1(n) + g_2(n)] \\ &\le c_3\, 2 \max\{g_1(n), g_2(n)\}. \end{aligned}$$

Hence, $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$,
with the constants c and n_0 required by the O definition being $2c_3 = 2 \max\{c_1, c_2\}$ and $\max\{n_1, n_2\}$, respectively.

$$\left.\begin{array}{|c|}\hline t_1(n) \in O(g_1(n)) \\ \hline t_2(n) \in O(g_2(n)) \\ \hline \end{array}\right\} \quad t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

- Ex:

  $t_1(n) = \frac{1}{2} n(n-1)$, $t_2(n) = n-1$

  $t_1(n) \in O(n^2)$, $t_2(n) \in O(n)$; i.e) $g_1(n) = n^2$, $g_2(n) = n$

  $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$

  So, $t_1(n) + t_2(n) \in O(\max\{n^2, n\}) = O(n^2)$

Using Limits for Comparing Orders of Growth:

Three principal cases

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n).^3 \end{cases}$$

L'Hospital's rule :

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$$

Stirling's formula:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \quad \text{for large values of } n$$

EXAMPLE 1: Compare the orders of growth of $\frac{1}{2} n(n-1)$ and n^2.

$$\lim_{n\to\infty} \frac{\frac{1}{2}n(n-1)}{n^2} = \frac{1}{2}\lim_{n\to\infty} \frac{n^2-n}{n^2} = \frac{1}{2}\lim_{n\to\infty}(1-\frac{1}{n}) = \frac{1}{2}.$$

- Limit is equal to a constant, the functions have the same order of growth or, symbolically,

$$\frac{1}{2} \mathbf{n(n-1) \in \Theta(n^2)}.$$

EXAMPLE 2 Compare the orders of growth of $\log_2 n$ and $\sqrt{n}$.

$$\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n\to\infty} \frac{(\log_2 n)'}{(\sqrt{n})'} = \lim_{n\to\infty} \frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2\log_2 e \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0.$$

- limit is equal to zero, $\log_2 n$ has a smaller order of growth than $\sqrt{n}$.

$$\mathbf{\log_2 n \in O(\sqrt{n})}.$$

EXAMPLE 3: Compare the orders of growth of n! and 2^n

$$\lim_{n\to\infty} \frac{n!}{2^n} = \lim_{n\to\infty} \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n} = \lim_{n\to\infty} \sqrt{2\pi n}\frac{n^n}{2^n e^n} = \lim_{n\to\infty} \sqrt{2\pi n}\left(\frac{n}{2e}\right)^n = \infty$$

- n! and 2^n have the larger order of growth

$$\mathbf{n! \in \Omega(2^n)}$$

Properties of Big – oh:

1. If there are 2 functions $t_1(n)$ and $t_2(n)$, such that $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$ then

   $t_1(n) + t_2(n) = O(\max\{g_1(n), g_2(n)\})$

2. $t(n) \in O(t(n))$
3. If there are 2 functions $t_1(n)$ and $t_2(n)$, such that $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$ then

   $t_1(n) * t_2(n) = O(g_1(n) * g_2(n))$

4. If $t(n) \in O(g(n))$ and $g(n) \in O(h(n))$ then $t(n) \in O(h(n))$
5. In a polynomial the highest power term dominates other terms i.e) maximum degree is considered

   Eg: for $3n^3+2n^2+10$

   Time complexity is $O(n^3)$

6. Any constant values leads to O(1) time complexity. ie, if t(n) = c, then it belongs to O(1) time complexity
7. $O(1) < O(\log n) < O(n) < O(n^2) < O(2^n)$
8. $t(n) = \Theta(g(n))$ iff $t(n) = O(g(n))$ and $t(n) = \Omega(g(n))$

- **Basic efficiency classes**:

Class	Name	Comments
1	constant	- Short of best-case efficiencies, - an algorithm's running time typically goes to infinity when its input size grows infinitely large.
log n	logarithmic	- a result of cutting a problem's size by a constant factor on each iteration of the algorithm - linear running time.
n	linear	- Algorithms that scan a list of size n
n log n	linearithmic	- Many divide-and-conquer algorithms in the average case
n^2	quadratic	- characterizes efficiency of algorithms with two embedded loops - example : n × n matrices
n^3	cubic	- characterizes efficiency of algorithms with three embedded loops
2^n	exponential	- algorithms that generate all subsets of an n-element set
n!	factorial	- algorithms that generate all permutations of an n-element set.

1.6 Empirical Analysis

- Some simple algorithms are very difficult to analyze with mathematical precision and certainty.
- The principal alternative to the mathematical analysis of an algorithm's efficiency is empirical analysis.
- Empirical analysis of an algorithm is performed by running a program implementing the algorithm on a sample of inputs and analyzing the data observed.

General Plan for the Empirical Analysis of Algorithm Time Efficiency

1. Understand the experiment's purpose.
2. Decide on the efficiency metric *M* to be measured and the measurement unit (an operation count vs. a time unit).
3. Decide on characteristics of the input sample (its range, size, and so on).
4. Prepare a program implementing the algorithm (or algorithms) for the experimentation.
5. Generate a sample of inputs.
6. Run the algorithm (or algorithms) on the sample's inputs and record the data observed.
7. Analyze the data obtained.

Goals in analyzing algorithms empirically: They include

- checking the accuracy of a theoretical assertion about the algorithm's efficiency,
- comparing the efficiency of several algorithms for solving the same problem or different implementations of the same algorithm,
- developing a hypothesis about the algorithm's efficiency class, and
- ascertaining the efficiency of the program implementing the algorithm on a particular machine.

How the algorithm's efficiency is to be measured:

- ➢ The first alternative is to insert a counter (or counters) into a program implementing the algorithm to count the number of times the algorithm's basic operation is executed.
- ➢ The second alternative is to time the program implementing the algorithm in question.
  - ➢ The easiest way to do this is to use a system's command, such as the time command in UNIX.
  - ➢ Alternatively, one can measure the running time of a code fragment by asking for the system time right before the fragment's start (t_{start}) and just after its completion (t_{finish}), and then computing the difference between the two ($t_{finish} - t_{start}$)

Profiling

- measuring time spent on different segments of a program
- Getting such data called profiling
- is an important resource in the empirical analysis of an algorithm's running time;
- the data in question can usually be obtained from the system tools available in most computing environments.

Decide on a sample of inputs:

- use a sample representing a "typical" input - a set of instances they use for benchmarking
- to make decisions about the sample size
- and a procedure for generating instances in the range chosen.

Generating Pseudo Random Numbers:

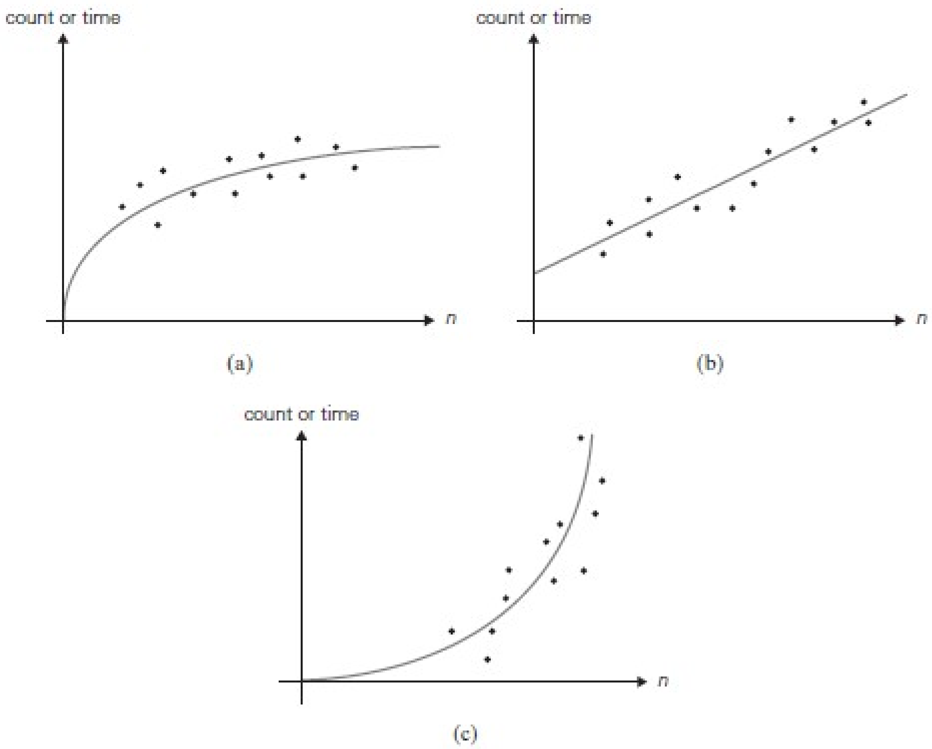
- an empirical analysis requires generating random numbers.
- the problem can be solved only approximately
- its output will be a value of a (pseudo)random variable uniformly distributed in the interval between 0 and
- Algorithms for generating (pseudo)random numbers **linear congruential method**

ALGORITHM *Random(n, m, seed, a, b)*
//Generates a sequence of *n* pseudorandom numbers according to the linear congruential method
//Input: A positive integer *n* and positive integer parameters *m, seed, a, b*

//Output: A sequence $r1, \ldots, rn$ of n pseudorandom integers uniformly distributed among integer values between 0 //and $m - 1$
//Note: Pseudorandom numbers between 0 and 1 can be obtained by treating the integers generated as digits after the //decimal point
$r0 \leftarrow seed$
for $i \leftarrow 1$ to n do
$ri \leftarrow (a * ri-1 + b)$ mod m

- ✔ *The empirical data obtained as the result of an experiment need to be recorded* and then presented for an analysis.
- ✔ Data can be presented numerically in a table or graphically in a *scatterplot*, i.e., by points in a Cartesian coordinate system.
- ✔ *the form of a scatterplot may also help in ascertaining* the algorithm's probable efficiency class.
  a) For a logarithmic algorithm, the scatterplot will have a concave shape
  b) For a linear algorithm, the points will tend to aggregate around a straight line or, more generally, to be contained between two straight lines
  c) Scatterplots of functions in $(n \lg n)$ and $(n2)$ will have a convex shape making them difficult to differentiate

Typical scatter plots. (a) Logarithmic. (b) Linear. (c) One of the convex functions.



Applications of the empirical analysis
- ○ is to predict the algorithm's performance on an instance not included in the experiment sample.
- ○ *Extrapolation:* Predicting the values of n outside the sample range.
- ○ *Interpolation*, which deals with values within the sample range.)

Basic differences between mathematical and empirical analyses of algorithms.
- The principal strength of the mathematical analysis is its independence of specific inputs;
- its principal weakness is its limited applicability, especially for investigating the average-case efficiency.
- The principal strength of the empirical analysis lies in its applicability to any algorithm,
- but its results can depend on the particular sample of instances and the computer used in the experiment.

1.7 Mathematical analysis for Recursive and Non-recursive algorithms

1.7. 1. Mathematical Analysis for Recursive Algorithms

- **Recursive Algorithm:**
  - The same operation or function is executed a number of times to obtain the result
  - Recurrence Equation: Equation that defines a sequence recursively
    - **T(n) = T(n-1) + n**

- **General Plan for Analyzing the Time Efficiency of Recursive Algorithms**

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation.
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.
4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or, at least, ascertain the order of growth of its solution

Examples:

1. Computing factorial for a number
2. Tower of Hanoi
3. Finding the number of digits

- **Example 1: Computing factorial for a number**
  - Compute the factorial function $F(n) = n!$ for an arbitrary nonnegative integer n.

    - $n! = 1 \ldots (n-1) \cdot n$
      $= (n-1)! \cdot n$ for $n \geq 1$
      and $0! = 1$
    - compute $F(n) = F(n-1) \cdot n$

    - **ALGORITHM** F(n)
      //Computes n! recursively
      //Input: A nonnegative integer n
      //Output: The value of n!
      if n = 0 return 1
      else return F (n − 1) ∗ n

    - Ex: Compute 3!
      Solution:
      F(3) = F(3-1) * 3 = F(2) * 3
      F(2) = F(2-1) * 2 = F(1) * 2
      F(1) = F(1-1) * 1 = F(0) * 1
      F(0) = 1
      F(1) = 1 * 1 = 1
      F(2) = 1 * 2 = 2
      F(3) = 2 * 3 = 6

    - **Analysis:**
      i) Measuring the input's size:
        - input size - n

      ii) Basic operation:
        - multiplication

iii) the number of times the basic operation (Multiplication) is executed - M(n).
iv) Recurrence relation:

$$M(n) = \underset{\substack{\text{to compute}\\ F(n-1)}}{M(n-1)} + \underset{\substack{\text{to multiply}\\ F(n-1)\text{ by } n}}{1} \quad \text{for } n > 0.$$

1. M(n − 1) multiplications are spent to compute F (n − 1), and one more multiplication is needed to multiply the result by n.
2. M(n) is a function of n, but implicity as a function of its value at another point, n-1. Such equations are called recurrence relations or recurrences.

v) Solve the recurrence:

To solve the recurrences, an initial condition is needed.

If n= 0 return 1

$$\underset{\text{the calls stop when } n=0}{M(0)} = \underset{\text{no multiplications when } n=0}{0.}$$

Hence

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

The first is the factorial function F(n) itself,it is defined by the recurrence

$$F(n) = F(n-1) \cdot n \quad \text{for every } n > 0,$$
$$F(0) = 1.$$

- Solution:
- ✔ **Method of backward substitution:**

$$\begin{aligned} M(n) &= M(n-1) + 1 && \text{substitute } M(n-1) = M(n-2) + 1 \\ &= [M(n-2) + 1] + 1 = M(n-2) + 2 && \text{substitute } M(n-2) = M(n-3) + 1 \\ &= [M(n-3) + 1] + 2 = M(n-3) + 3. \end{aligned}$$

- ✔ Genaral Formula: M(n) = M(n-i) + i
- ✔ Mathematucal Induction: (Correctness of the formula)

Substitute i=n

M(n) = M(n-n)+n
= M(0) +n = 0 + n
= n

✔ The time complexity of factorial function is **Θ**(n)

Example 2: **Tower of Hanoi puzzle:**

- *n* disks of different sizes that can slide onto any of three pegs.
- Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top.
- The goal is to move all the disks to the third peg, using the second one as an auxiliary, if necessary.
- Only one disk can be moved at a time, and it is forbidden to place a larger disk on top of a smaller one.

- **Steps:**
  1. To move *n*>1 disks from peg 1 to peg 3 (with peg 2 as auxiliary),
     - first move recursively *n* − 1 disks from peg 1 to peg 2 (with peg 3 as auxiliary),
     - then move the largest disk directly from peg 1 to peg 3, and, finally,
     - move recursively *n* − 1 disks from peg 2 to peg 3 (using peg 1 as auxiliary)
  2. if *n* = 1, simply move the single disk directly from the source peg to the destination peg.

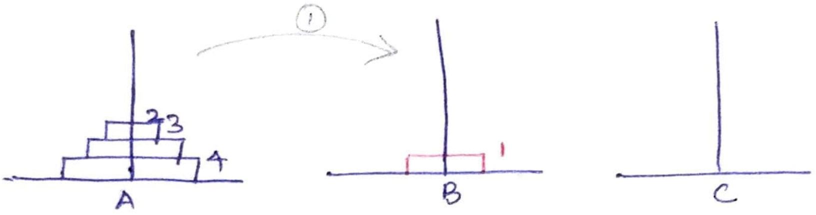
Tower of Hanoi

Ex: 4 disks

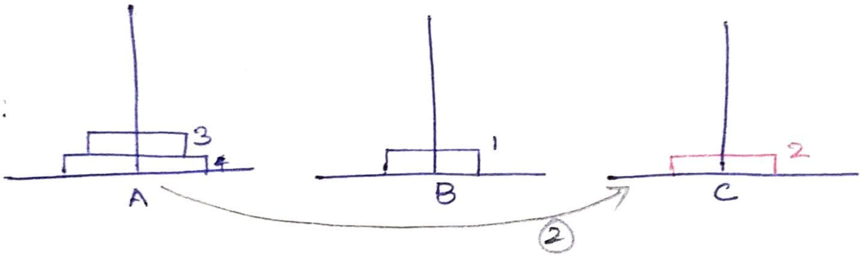
Prbm:-

Solution:

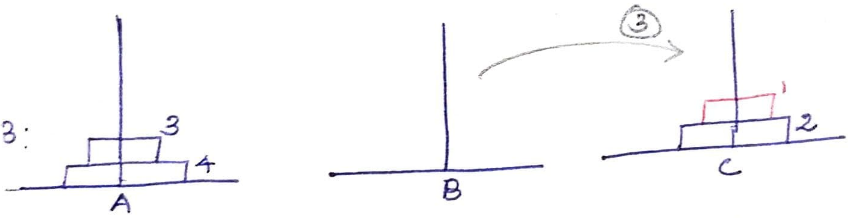
Move 1:



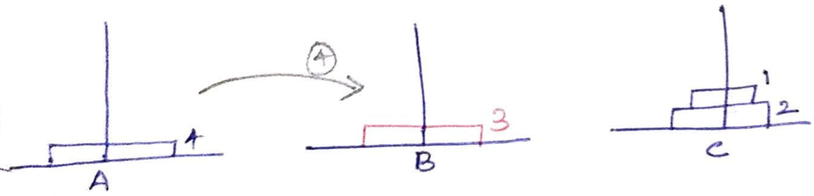
Move 2:



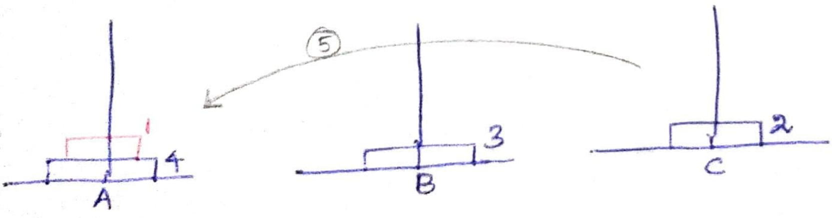
Move 3:



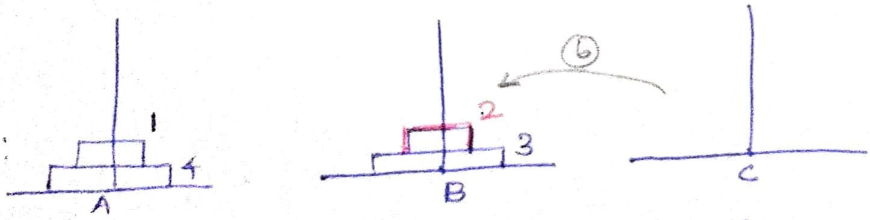
Move 4:



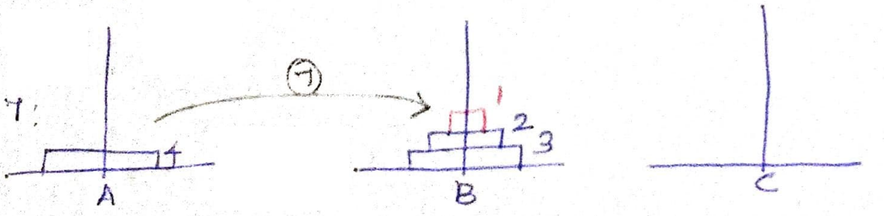
Move 5:



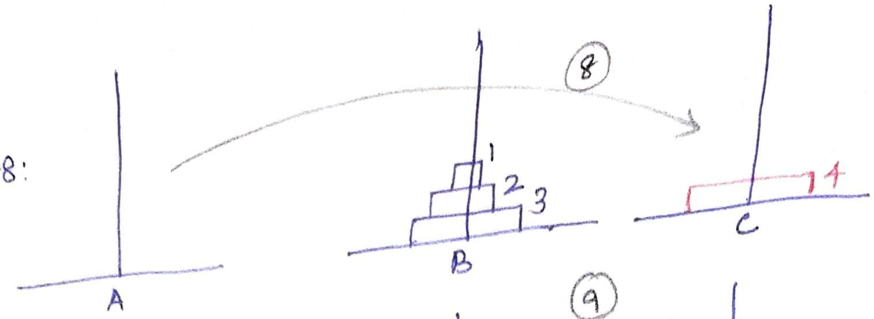
Move 6:



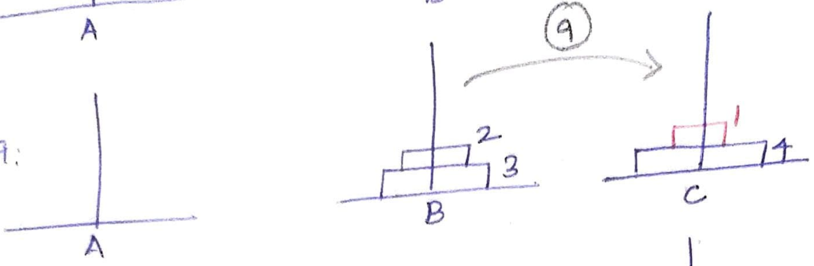
Move 7:



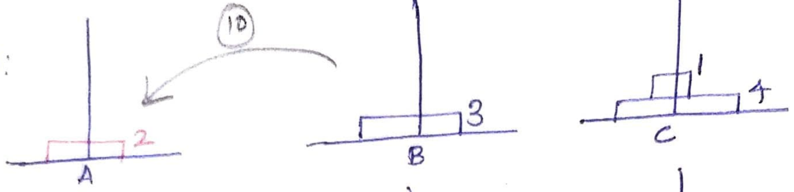
Move 8:



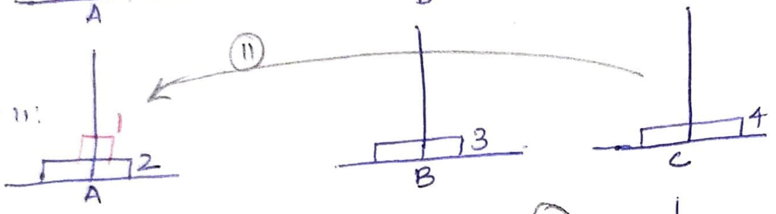
Move 9:



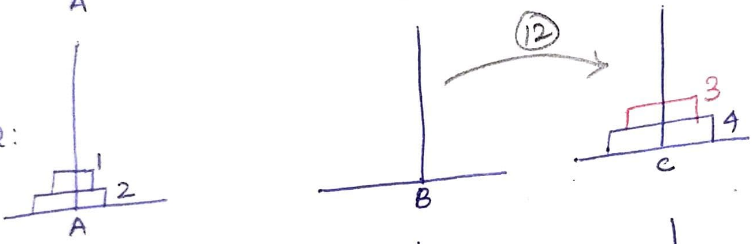
Move 10:



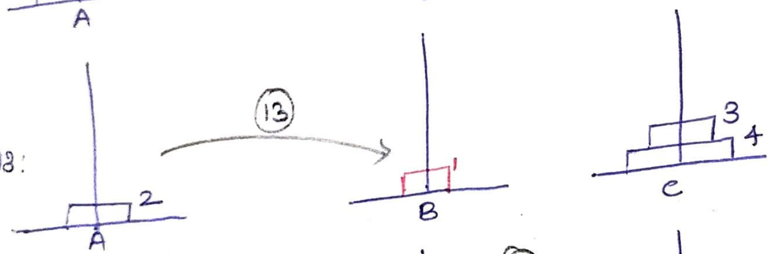
Move 11:



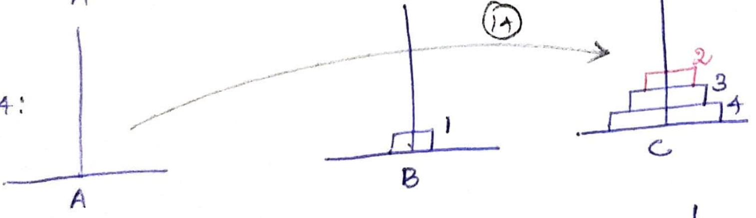
Move 12:



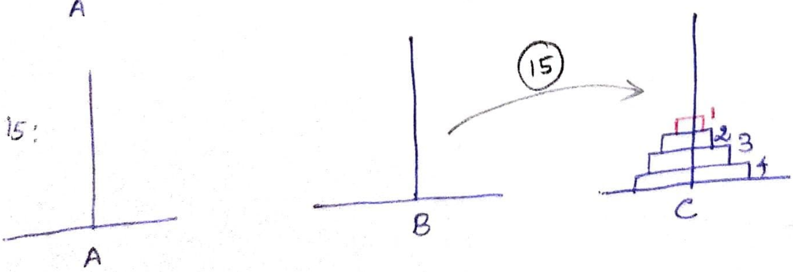
Move 13:



Move 14:



Move 15:



Total number of moves = 15.

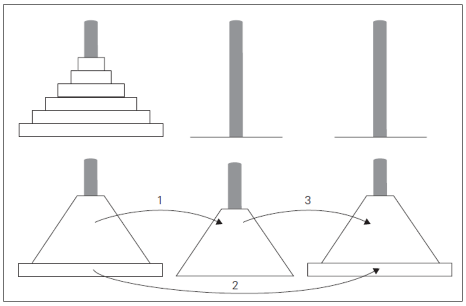
ie) $M(4) = 15$

$$M(n) = 2^n - 1$$
$$= 2^4 - 1$$
$$= 16 - 1$$
$$\boxed{M(4) = 15}$$

- **ALGORITHM Hanoi(n,A,C,B)** // n – number of disks, A – Peg 1, C – Peg 3, B – Peg 2

```
if n==1
        Move the disk from A to C
else
        Hanoi( n-1,A,B,C)
        Move the disk from A to C
        Hanoi( n-1,B,C,A)
```

- **Recursive solution:**



- **Analysis:**

i) Measuring the input's size: input size – n (number of disks)
ii) Basic operation: moving one disk
iii) the number of times the basic operation (Moves) is executed - M(n).
iv) Recurrence relation

- The number of moves M(n) depends on n only. The recurrence equation is

$$M(n) = M(n-1) + 1 + M(n-1) \quad \text{for } n > 1.$$

Initial condition: M(1) = 1

v) Solve the Recurrence Relation

$$M(n) = 2M(n-1) + 1 \qquad \text{sub. } M(n-1) = 2M(n-2) + 1$$
$$= 2[2M(n-2) + 1] + 1 = 2^2 M(n-2) + 2 + 1 \quad \text{sub. } M(n-2) = 2M(n-3) + 1$$
$$= 2^2[2M(n-3) + 1] + 2 + 1 = 2^3 M(n-3) + 2^2 + 2 + 1.$$

Substitute i,

$M(n) = 2^i M(n-i) + 2^{i-1} + 2^{i-2} + \ldots + 2 + 1$
$= 2^i M(n-i) + 2^i - 1$

Initial condition is specified for n=1, for i = n-1,

$$M(n) = 2^{n-1} M(n-(n-1)) + 2^{n-1} - 1$$
$$= 2^{n-1} M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1.$$

- The order of growth is **O(2^n)**

- **Example 3: Counting The number of binary digits**
  - Finds the number of binary digits in the binary representation of a positive decimal integer.

- **ALGORITHM** BinRec(n)

```
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
        if n = 1 return 1
        else return BinRec( n/2 ) + 1
```

- **Analysis:**

i) Measuring the input's size: input size - n

ii) Basic operation: Addition

iii) the number of times the basic operation (Addition) is executed - A(n).

iv) Recurrence relation

$$A(n) = A(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1.$$

Initial condition: A(1) = 0

v) Solve the Recurrence Relation

- let n= 2^k, the order of growth for all values of *n*.

$$A(2^k) = A(2^{k-1}) + 1 \quad \text{for } k > 0,$$
$$A(2^0) = 0.$$

- backward substitutions

$$\begin{aligned}
A(2^k) &= A(2^{k-1}) + 1 && \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1 \\
&= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 && \text{substitute } A(2^{k-2}) = A(2^{k-3}) + 1 \\
&= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 && \ldots \\
&\ldots \\
&= A(2^{k-i}) + i \\
&\ldots \\
&= A(2^{k-k}) + k.
\end{aligned}$$

$$A(2^k) = A(1) + k = k,$$

after returning to the original variable $n = 2^k$ and hence $k = \log_2 n$,

$$\boxed{A(n) = \log_2 n \in \Theta(\log n).}$$

1.7.2 Mathematical Analysis of Nonrecursive Algorithms

General Plan for Analyzing the Time Efficiency of Nonrecursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation. (As a rule, it is located in the innermost loop.)
3. Check whether the number of times the basic operation is executed, depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
4. Set up a sum, expressing the number of times the algorithm's basic operation is executed.
5. Using standard formulas and rules of sum manipulation, either find a closed form formula for the count or, at the very least, establish its order of growth.

i) Basic rules for Sum manipulation:

$$\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i,$$
$$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i,$$

ii) Summation formulas

$$\sum_{i=l}^{u} 1 = u - l + 1 \quad \text{where } l \le u \text{ are some lower and upper integer limits,}$$
$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2).$$

Examples:

1. Finding largest element in a list of n numbers
2. Element Uniqueness Problem
3. Matrix Multiplication

Example 1: Finding largest element:

- The problem of finding the value of the largest element in a list of n numbers.

ALGORITHM MaxElement(A[0..n − 1])
//Determines the value of the largest element in a given array
//Input: An array A[0..n − 1] of real numbers
//Output: The value of the largest element in A

maxval ← A[0]
for i ← 1 to n − 1 do
if A[i] > maxval
maxval ← A[i]
return maxval

- Ex: Determine the value of the largest element in an array
  A={34, 65, 100, 67}

Illustration of example:

// MaxElementA[4] Determines the value of the largest element in a given array
//Input: An array A={34,65,100,67}
maxval ←34
for i ←1 to 3 do
if 65>34 // here i=1
maxval←65
if 100 >65 // here i=2
maxval ←100
if 100 >65 // here i=3 //end of elements in the list
return 100
//Output : 100

- **Analysis:**

i) Measuring the input's size:

- number of elements in the array, i.e., n

ii) Basic operation:

- two operations :
  - comparison
  - assignment
- the comparison is executed on each repetition

iii) the number of comparisons:

C(n) - The number of times the comparison is executed

iv) Set up a sum expression: i.e) Find a formula expressing it as a function of size n.

○ The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds 1 and n − 1
C(n):

$$C(n) = \sum_{i=1}^{n-1} 1.$$

v) Find a closed form formula and establish its order of growth:

✔ sum to compute because it is nothing other than 1 repeated n – 1 times.

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

$$\sum_{i=l}^{u} 1 = u - l + 1$$

Example 2: Element Uniqueness Problem:

- Check whether all the elements in a given array of n elements are distinct.

- **ALGORITHM** *UniqueElements(A*[0..*n* − 1]*)*
  //Determines whether all the elements in a given array are distinct
  //Input: An array *A*[0..*n* − 1]
  //Output: Returns "true" if all the elements in *A* are distinct
  // and "false" otherwise
  for *i* ←0 **to** *n* − 2 **do**
  for *j* ←*i* + 1 **to** *n* − 1 **do**
  if *A*[*i*]= *A*[*j*] **return false**
  return true
- Ex: A={54,78,56,2}
  Illustration of example :
  //UniqueElements(A[4])
  //Input: An array A={54,78,56,2}
  i ←0 do // the range of i is from 0 to 2
  j← 1 do // the range of j is from 1 to 3
  54!=78
  j←2
  54!=56
  j←3
  54!=2
  i ←1 do
  j←2
  78!=56
  j←3
  78!=2
  i ←2 do
  j←3
  56!=2
  Return true
  //Output: true. All the elements in the array are distinct.

- **Analysis:**
  i) Measuring the input's size:
  - number of elements in the array, i.e., n

  ii) Basic operation:
  - comparison

  iii) the number of comparisons
  C(n) - The number of times the comparison is executed
  - depends on the number if elements and theirpositions

  iv) Find a formula expressing it as a function of size n.
  Worst-case:
  - the number of element comparisons is the largest among all arrays of size n.

  two kinds of worst-case inputs:
  i) arrays with no equal elements
  ii) arrays in which the last two elements are the only pair of equal elements.

  - one comparison is made for each repetition of the innermost loop, i.e., for each value of the loop variable *j* between its limits *i* + 1 and *n* − 1;
  - this is repeated for each value of the outer loop, i.e., for each value of the loop variable *i* between its limits 0 and *n* − 2.

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

v) Find a closed form formula and establish its order of growth:

$$\begin{aligned} C_{worst}(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 \\ &= \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] \\ &= \sum_{i=0}^{n-2} (n-1-i) \\ &= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i \\ &= (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} \\ &= (n-1)^2 - \frac{(n-2)(n-1)}{2} \\ &= \frac{(n-1)n}{2} \end{aligned}$$

$$C_{worst}(n) \quad \boxed{\approx \frac{1}{2}n^2 \in \Theta(n^2).}$$

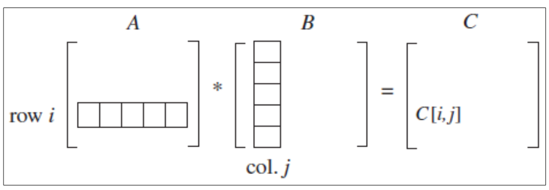
Compute the sum:(Another Method)

$$\begin{aligned} \sum_{i=0}^{n-2} (n-1-i) &= (n-1) + (n-2) + \cdots + 1 \\ &= \frac{(n-1)n}{2} \end{aligned}$$

- The algorithm needs to compare all n(n-1)/2 distinct pairs of its n elements.

Example 3: Matrix Multiplication:

- Given two $n \times n$ matrices A and B, find the time efficiency of the definition-based algorithm for computing their product $C = AB$.
- By definition, C is an $n \times n$ matrix whose elements are computed as the scalar (dot) products of the rows of matrix A and the columns of matrix B:



- where $C[i, j] = A[i, 0]B[0, j] + \ldots + A[i, k]B[k, j] + \ldots + A[i, n-1]B[n-1, j]$ for every pair of indices $0 \leq i, j \leq n-1$.

- **ALGORITHM** *MatrixMultiplication(A*[0..*n* − 1, 0..*n* − 1], *B*[0..*n* − 1, 0..*n* − 1])

  //Multiplies two square matrices of order *n* by the definition-based algorithm

  //Input: Two *n* × *n* matrices *A* and *B*

  //Output: Matrix *C* = *AB*

```
for i ←0 to n − 1 do
    for j ←0 to n − 1 do
        C[i, j ]←0.0
    for k←0 to n − 1 do
        C[i, j ]←C[i, j ]+ A[i, k] ∗ B[k, j]
return C
```

- **Example :**

  A[2][2]={1,2,4,6}

  B[2][2]={6,7,8,9}

  C[2][2]={22,25,72,82}

Illustration for example :

//Algorithm: Matrix multiplication (A[2][2], B[2][2])

//Multiplies two square matrices of order n.

```
//Input: A[2][2]={1,2,4,6} and B[2][2]={6,7,8,9}
for i ←0            //range of i={0,1}
for j ←0            //range of j={0,1}
    C[0, 0 ]←0.0
    for k←0         //range of k={0,1}
        C[0, 0 ]←0+ 1*6         // A[0,0]=1 and B[0,0]=6
        C[0,0] ←6
    For k←1
        C[0,0] ← 6+2*8          // A[0,1]=2 and B[1,0]=8
        C[0,0] ←22
For j ←1
    C[0,1 ]←0.0
    for k←0
        C[0, 1 ]←0+1*7          // A[0, 0]=1 and B[0,1]=7
        C[0,1] ←7
    For k ←1
        C[0,1 ]←7+ 2*9          // A[0,1]=2 and B[1, 1]=9
        C[0,1] ←18
For i ←1 for j ←0
    C[1, 0 ]←0.0
    for k←0
        C[1, 0 ]←0+4*6          // A[1,0]=4 and B[0,0]=6
        C[1,0] ←24
    For k←1
        C[1,0] ← 24+6*8         // A[1,1]=6 and B[1,0]=8
        C[1,0] ←72
For j ←1
    C[1,1 ]←0.0
    for k←0
        C[1, 1 ]←0+4*7          // A[1, 0]=4 and B[0,1]=7
        C[0,1] ←28
    For k ←1
        C[1,1 ]←28+ 6*9         // A[1,1]=6 and B[1, 1]=9
        C[0,1] ←82
```

//Output: Matrix C [2][2]={22,25,72,82}

- **Analysis:**

  i) Measuring the input's size:

  - matrix order, i.e., n

ii) Basic operation:

2 operations:

i) Multiplication

ii) Addition

- First consider, the basic operation is multiplication

iii) the total number of multiplications:

M(n) - The number of times the multiplication is executed

iv) Set up a sum for the total number of multiplications M(n):

- one multiplication executed on each repetition of the algorithm's innermost loop

$$\sum_{k=0}^{n-1} 1$$

- total number of multiplications *M(n)* is expressed by the following triple sum:

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$$

v) Find a closed form formula and establish its order of growth:

Compute the sum:

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$$

$$= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n \qquad = \sum_{i=0}^{n-1} n^2$$

$$M(n) \quad = n^3.$$

Estimate the running time of the algorithm:

- Total number of Multiplications $M(n)=n^3$

$$T(n) \approx c_m M(n) = c_m n^3$$

c_m ------> Time of one multiplication

- Total number of Additions $A(n)=n^3$

$T(n) \approx c_a A(n) = c_a n^3$

c_a ------> Time of one addition

- Total Running Time:

$$T(n) \approx c_m M(n) + c_a A(n)$$

$$= c_m n^3 + c_a n^3$$

$$= (c_m + c_a) n^3$$

- Time complexity of Matrix Multiplication is $\mathbf{\Theta(n^3)}$

Example 4: Counting the binary digits:

- Finds the number of binary digits in the binary representation of a positive decimal integer

- **ALGORITHM** *Binary(n)*

```
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
    count ←1
    while n > 1 do
        count ←count + 1
        n←n/2
    return count
```

- **Analysis:**
  - Measuring the input's size: - input size is n
  - Basic operation:
  - – most frequently executed operation is not inside the **while** loop but rather the comparison $n > 1$ that determines whether the loop's body will be executed.
  - – Since the number of times the comparison will be executed is larger than the number of repetitions of the loop's body by exactly 1.

  - value n is halved on each repetition of the loop
- formula for the number of times the comparison $n>1$ will be executed is actually $\log_2 n + 1$
- Time complexity for counting number of bits of given number is $\boldsymbol{\Theta}(\log_2 n)$

1.8 Visualization

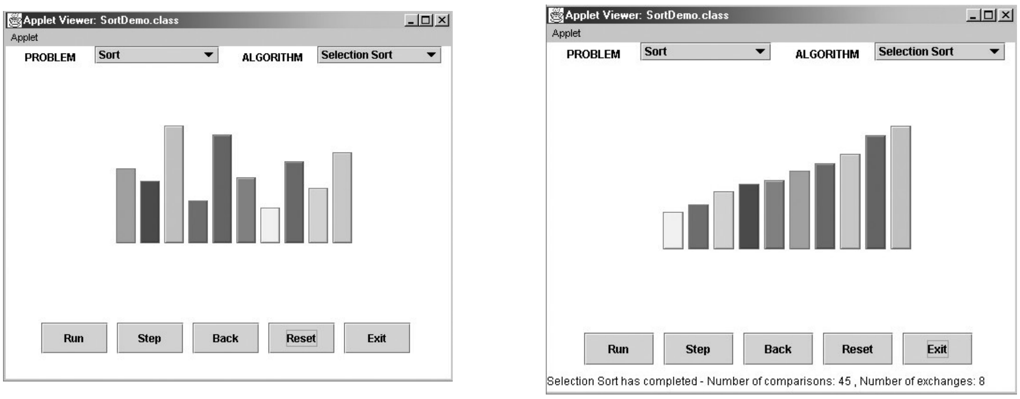
- Third way to study algorithms.
- *Algorithm visualization* -can be defined as the use of images to convey some useful information about algorithms.
- That information can be a visual illustration of an algorithm's operation, of its performance on different kinds of inputs, or of its execution speed versus that of other algorithms for the same problem.
- To accomplish this goal, an algorithm visualization uses graphic elements—points, line segments, two- or three-dimensional bars, and so on—to represent some "interesting events" in the algorithm's operation.

Two principal variations of algorithm visualization:

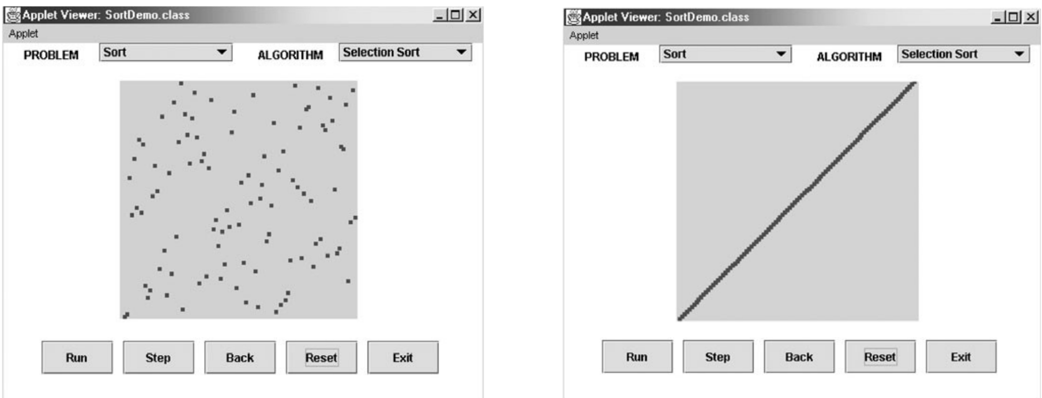
- Static algorithm visualization
- Dynamic algorithm visualization, also called *algorithm animation*

- Static algorithm visualization shows an algorithm's progress through a series of still images.
- Algorithm animation, on the other hand, shows a continuous, movie-like presentation of an algorithm's operations.

Initial and final screens of a typical visualization of a sorting algorithm using the bar representation



Initial and final screens of a typical visualization of a sorting algorithm using the scatterplot representation.



Two principal applications of algorithm visualization:

1. Research and
2. Education.

- Potential benefits for researchers are based on expectations that algorithm visualization may help uncover some unknown features of algorithms.
- The application of algorithm visualization to education seeks to help students learning algorithms.

Nov/Dec - 2016

Unit - I

(1) Solve the recurrence relation:

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

Soln :-

Method I: Backward Substitution:

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

$[\because x(n-1) = x[(n-1)-1]+5$
$= x(n-2)+5]$

$= [x(n-2)+5]+5$

$[x(n-2) = x[(n-2)-1]+5$
$= x(n-3)+5]$

$= [x(n-3)+5]+5+5$

ie)
$x(n) = x(n-3)+3\times5$
$x(n) = x(n-4)+4\times5$
$\vdots$

ith
$\boxed{x(n) = x(n-i)+i\times5}$

* As per initial condition, $x(1)=0$.
$\boxed{n-i=1}$ ie) $i=n-1$

$x(n) = x(n-[n-1]) + (n-1)\times5$
$= x(n-n+1) + (n-1)\times5$
$= x(1)+(n-1)\times5$
$= 0+(n-1)\times5$

Ans: $\boxed{x(n) = 5(n-1)}$

Method 2:

Forward Substitution:

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

$x(1) = 0$

$x(2) = x(2-1)+5 = x(1)+5 = 0+5$

ie) $x(2) = 5$

$x(3) = x(3-1)+5 = x(2)+5 = 5+5$

$x(3) = 10$.

$x(4) = 15$

ie) $x(2) = 1 \times 5$

$x(3) = 2 \times 5$

$x(4) = 3 \times 5$

$\vdots$

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

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

ie) $\boxed{x(n) = 5(n-1)}$

(2) $x(n) = 3x(n-1)$ for $n > 1$, $x(1) = 4$.

Soln:

Method 1:

Backward Substitution

$x(n) = 3x(n-1)$

$= 3[3x(n-2)]$

$= 3^2 x(n-2)$

$= 3^2 [3x(n-3)]$

$= 3^3 x(n-3)$

$\vdots$

$[\because x(n-1) = 3x[(n-1)-1]$
$= 3x[n-2]]$

$[\because x(n-2) = 3x(n-2-1)$
$= 3x(n-3)$

i^{th} $\quad x(n) = 3^{i} x(n-i)$

As per initial condition, $x(1) = 4$

$n - i = 1$

ie) $i = n - 1$

$x(n) = 3^{n-1} x[n - (n-1)]$

$= 3^{n-1} x[n - n + 1]$

$= 3^{n-1} \cdot x(1)$

$= 3^{n-1} \cdot 4$

ie) $\boxed{x(n) = 4 \times 3^{(n-1)}}$

Method 2: Forward Substitution:

$x(n) = 3x(n-1)$

$x(1) = 4$

$x(2) = 3x(2-1) = 3x(1) = 3 \times 4$

$x(3) = 3x(3-1) = 3x(2) = 3(3 \times 4) = 3^2 \cdot 4$

$x(4) = 3x(4-1) = 3x(3) = 3[3^2 \cdot 4] = 3^3 \cdot 4$

$\vdots$

$x(5) = 3^4 \cdot 4$

$\vdots$

$x(i) = 3^{i-1} \cdot 4$

$x(n) = 3^{n-1} \cdot 4$

ie) $\boxed{x(n) = 4 \times 3^{n-1}}$

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

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

Soln:-

Method 1: Backward substitution:

$x(n) = x(n-1) + n$ $\qquad x(n-1) = x(n-2) + (n-1)$

$= [x(n-2) + (n-1)] + n$ $\qquad x(n-2) = x(n-3) + (n-2)$

$x(n) = x(n-3) + (n-2) + (n-1) + n$

$\vdots$

i^{th}

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

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

$\vdots$

As per initial condition, $x(0) = 0$

$\boxed{i=n}$ $\quad x(n) = x(n-n) + (n-n+1) + (n-n+2) + \cdots + n$

$= x(0) + 1 + 2 + \cdots + n$

$= 0 + 1 + 2 + \cdots + n$

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

Method 2: Forward substitution

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

$x(0) = 0$

$x(1) = x(1-1) + 1 = x(0) + 1 = 1$

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

$x(3) = x(3-1) + 3 = x(2) + 3 = 1+2+3$

$\vdots$

$x(n) = 1+2+3+\cdots+n$

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

UNIT II

BRUTE FORCE AND DIVIDE-AND-CONQUER

Brute Force – Computing a^n – String Matching - Closest-Pair and Convex-Hull Problems - Exhaustive Search - Travelling Salesman Problem - Knapsack Problem - Assignment problem. Divide and Conquer Methodology – Binary Search – Merge sort – Quick sort – Heap Sort - Multiplication of Large Integers – Closest-Pair and Convex - Hull Problems.

ALGORITHM CLASSIFICATION

- ✓ Algorithms that use a similar problem-solving approach can be grouped together. Some of the famous algorithm types include:
  - Backtracking algorithms
  - Divide and conquer algorithms
  - Dynamic programming algorithms
  - Greedy algorithms
  - Branch and bound algorithms
  - Brute force algorithms
  - Randomized algorithms

2.1 BRUTE FORCE ALGORITHMS

- Brute force is a straightforward approach to solving a problem directly based on the problem's statement and definitions of the concepts involved.
- the brute-force strategy is indeed the one that is easiest to apply
  - A brute force algorithm simply tries all possibilities until a satisfactory solution is found.
  - It includes techniques for finding optimal solutions to hard problems quickly.
  - Brute force algorithms can be:
    - Optimizing: Finding the best solution among all solutions
      - Example: Finding the best path for a traveling salesman.
    - Satisfying: Finding a satisfying or good solution
      - Example: Finding a traveling salesman path that is within 10% of optimal solution.
  - Problems that can be solved by brute force technique include String Matching, Closest-Pair and Convex-Hull Problems, Selection Sort, Bubble Sort and Sequential Search

Advantages

- Simplicity
- Wide applicability
- useful for solving small-size instances of a problem
- It is a good method for developing better algorithms.

Disadvantages

- Rarely produces efficient algorithms
- Some brute force algorithms are extremely slow
- Not as creative when compared with other design techniques

2.1.1 Computing a^n

Definition:

- Compute a^n for a nonzero number a and a nonnegative integer n.

Method: Brute – Force

- By the definition of exponentiation,
- computing a^n by multiplying 1 by a **n times**

Ex: Compute 5^3

$$5^3 = 5*5*5 = 125$$

Analysis:

- The brute force algorithm requires n-1 multiplications.
- The recursive algorithm for the same problem, based on the observation that $a^n = a^{n/2} * a^{n/2}$ requires **Θ(log(n))** operations.

2.1.2 String Matching

- Given a string of n characters called the **text** and a string of m characters ($m \leq n$) called the **pattern**; find a substring of the text that matches the pattern.
- find *i*—the index of the leftmost character of the first matching substring in the text
- If matches other than the first one need to be found, a string-matching algorithm can simply continue working until the entire text is exhausted.

- A brute-force algorithm:
  - Align the pattern against the first m characters of the text and start matching the corresponding pairs of characters from left to right until either all the m pairs of the characters match or a mismatching pair is encountered.

Algorithm BruteForceStringMatch(T [0..n − 1], P[0..m − 1])
//Input: An array T [0…n − 1] of n characters representing a text and an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a matching substring or −1 if the search is unsuccessful

```
for i ←0 to n − m do
j ←0
while j <m and P[j ]= T [i + j ] do
j ←j + 1
if j = m return i
return −1
```

Example: Finding "NOT" in "NOBODY_NOTICED _HIM"

```
N O B O D Y _ N O T I C E D _ H I M
N O T
  N O T
    N O T
      N O T
        N O T
          N O T
            N O T
              N O T
```

The pattern's characters that are compared with their text counterparts are in bold type.

Analysis:

Worst-case:

- m(n-m+1) number of comparisons are made
- the worst case complexity is O(nm)

Average-case:

- the average case efficiency being Θ(n).

2.1.3 Closest-Pair Problem Definition

- The closest pair problem is to find the two closest points in a set of n points.
- the points *(x, y)* Cartesian coordinates and that the distance between two points $p_i(x_i,y_i)$ and $p_j(x_j, y_j)$ is the standard Euclidean distance

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

- Brute force algorithm:
  - computes the distance between every pair of distinct points and
  - return the indexes of the points for which the distance is the smallest.

ALGORITHM *BruteForceClosestPair(P)*
//Finds distance between two closest points in the plane by brute force
//Input: A list P of n ($n \geq 2$) points $p_1(x_1, y_1), \ldots, p_n(x_n, y_n)$
//Output: The distance between the closest pair of points
$d \leftarrow \infty$
for $i \leftarrow 1$ **to** $n - 1$ **do**
for $j \leftarrow i + 1$ **to** n **do**
$d \leftarrow$ min(d, $sqrt((x_i - x_j)^2 + (y_i - y_j)^2)$) //*sqrt* is square root
return d

Example

// *BruteForceClosestPair(P)*
//Input: List P with points p1 (3,9) ,p2(6,4) and p3(7,3) d←∞
i ←1 // range of i={1, 2}
j ←2 // range of j={2,3}
d ←min(∞, sqrt(34)) d ← 5.83
j ←3
d ←min(5.83, sqrt(1)) d ← 1
i ←2
j ←3 // range of j={3} d ←min(1, sqrt(52)) d ← 1
return 1
//Output: The index of the closest pair of points are p1 (3,9)and p3(7,3)

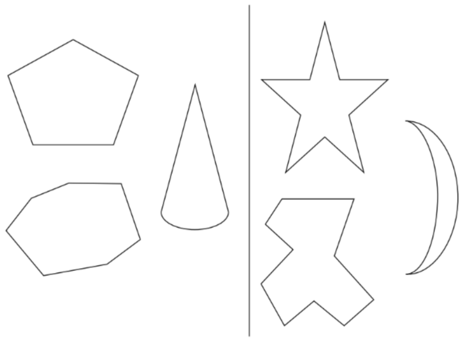
Analysis:

- input size is n points
- the basic operation is computing the square root

2.1.4 CONVEX-HULL PROBLEM

Definition: A set of points (finite or infinite) on the plane is called convex if for any two points p and q in the set ,the entire line segment with the end points at p and q belongs to the set.

(a) Convex sets (b) Sets that are not convex



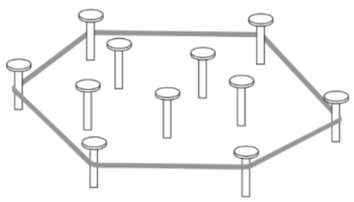
(a) (b)

- The convex hull of a set of n point in the plane is the smallest convex polygone that contains all of them.

Method : Solved by Brute force method**.**

Example: A rubber band interpretation of the convex hull

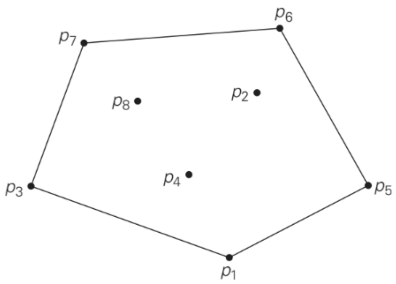
- Take a rubber band and stretch it to include all the nails, then let it snap into place.The convex hull is the area bounded by the snapped rubber band.



- A formal definition of the convex hull that is applicable to arbitrary set ,including sets of points that happens to lie on the same line, follows.

Definition: The convex hull of a set of points is the smallest convex set containing S.

- If S is convex , its convex hull is obviously S itself
- If S is a set of two points , its convex hull is the line segment connecting these points.
- If S is a set of three points not on the same line, its convex hull is the triangle with the vertices at the three points given.
- If three points do lie on the same line, the convex hull is the line segment with its end points at the two points that are farthest apart.



- The convex hull for this set of eight points is the convex polygon with its vertices at p_1, p_5, p_6, p_7, and p_3.

Theorem:

- The convex hull of any set S of n>2 points is a convex polygon with the vertices at some of the points of S.

Convex hull problem → is the problem of constructing the convex hull for a given set S of n points.

- To solve, to find the points that will serve as the Vertices of the polygon in question.
- Extreme points.

Definition: A extreme point of a convex set is a point of the set that is not a middle point of any line segment with end points in the set.

Property:

- Simplex method-algorithm
- Solves linear programming problems, which are problems of finding a minimum or a maximum of a linear function of n variables subject to linear constraints.

Algorithm:

Analytical geometry are needed to implement the algorithm:

Step 1: First, the straight line through two points (x_1,y_1), (x_2,y_2) in the coordinate plane can be defined by the equation $ax+by=c$ where $a=y_2-y_1$, $b=x_1-x_2$, $c=x_1y_2-y_1x_2$

Step 2: Second, a line divides the plane into two half-planes: for all the points in one of them $ax+by>c$, while for all the points on the other $ax+by<c$.

Step 3: To check whether the points lie on the same side of the line, to check the sign of the expression.

→ $\frac{n(n-1)}{2}$ pairs of distinct points.

→ other n-2 points

No of checks: $\frac{n(n-1)}{2}(n-2)$

Analysis: Time efficiency → $O(n^2)$

2.2 Exhaustive search method

- Exhaustive search is a brute –force approach to combinational problems. (permutations, combinations or subset of a set)
- It suggests generating each and every element of the problem's domain, selecting those of them that satisfy the problem's constraints, and then finding a desired element.
  1. Listing all possible solution.
  2. Evaluate solutions, disqualifying infeasible ones
  3. Find the best solution.

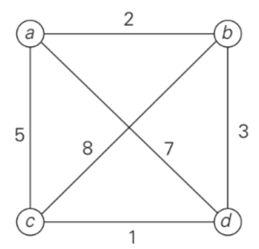
2.2.1: TRAVELING SALESMAN PROBLEM

Definition: To find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city where it started.

- Modeled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifying the distances.
- The problem can be stated as the problem of finding the shortest **Hamiltonian circuit** of the graph.(A Hamiltonian circuit defined as a cycle that passes through all the vertices of the graph exactly once)
- Hamiltonian circuit can also be defined as a sequence of n + 1 adjacent vertices v_{i0}, v_{i1},...,v_{in-1}, v_{i0}, where the first vertex of the sequence is the same as the last one and all the other n − 1 vertices are distinct.
- All circuits start and end at one particular vertex.

Method: Solved by Exhaustive search method.
Algorithm:
Step 1: Get all the tours by generating all the permutations of n-1 intermediate cities.
Step 2: Compute all the tour lengths.
Step 3: Find the shortest among them.

Example: Find the tour using Exhaustive search for the graph.
Problem:



Solution:

Tour	Length	
a --> b --> c --> d --> a	l = 2 + 8 + 1 + 7 = 18	
a --> b --> d --> c --> a	l = 2 + 3 + 1 + 5 = 11	optimal
a --> c --> b --> d --> a	l = 5 + 8 + 3 + 7 = 23	
a --> c --> d --> b --> a	l = 5 + 1 + 3 + 2 = 11	optimal
a --> d --> b --> c --> a	l = 7 + 3 + 8 + 5 = 23	
a --> d --> c --> b --> a	l = 7 + 1 + 8 + 2 = 18	

A solution to a small instance of the traveling salesman problem by exhaustive search .

Approach:

- **i.** Find out all (n-1)! Possible solution.
- **ii.** Determine the minimum cost.

Possible solution: (n-1)!

Example: 4: (4-1)! = 3!

2.2.2 : KNAPSACK PROBLEM

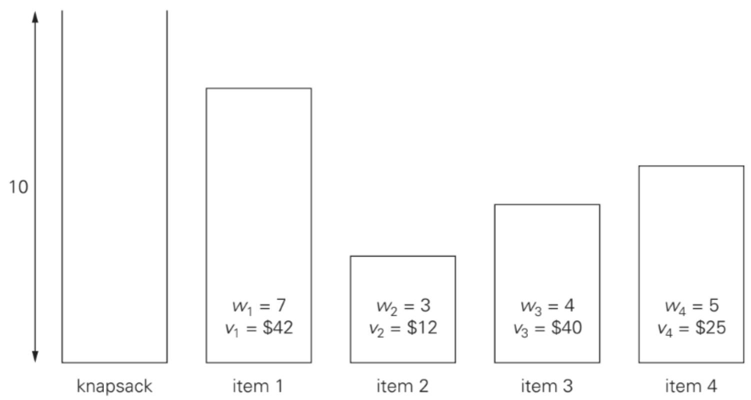
Definition: Given n items of known weights $w_1, w_2,...,w_n$ and values $v_1, v_2,...,v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack**.**

- To pick up the most valuable objects to fill the knapsack to its capacity.

Method: Solved by Exhaustive search method.

Example:

Problem: (a) Inside of the Knapsack problem.



(a)

Solution: (b) exhaustive search.

Subset	Total weight	Total value
∅	0	$ 0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

(b)

Algorithm:

Step 1: Find all the subset of set of n items.
Step 2: Compute the total weight of each subset.
Step 3: Find the subset of the largest value.

Exhaustive Search approach:

Step 1: Consider all the subset of the set of n items given computing the total weight of each subset in order to identify feasible subset.
Step 2: Finding a subset of the target value among them.

- The number of subset of an n-element set is 2^n
- The exhaustive search leads to a $\Omega(2^n)$ algorithm.

- For both traveling salesman and Knapsack problem, exhaustive search leads to algorithms that are inefficient on every input.
- Two problems are the best-known examples of **NP-hard problems**.
- Sophisticated approaches→ backtracking and branch-and-bound.

2.2.3: ASSIGNMENT PROBLEM

Definition:

- There are n people who need to be assigned to execute n jobs, one person per job.
- Each person is assigned to exactly one job and each job is assigned to exactlyone person.
- If the i^{th} person is assigned to the j^{th} job, the cost is a known quantity C[i, j] for each pair i, j = 1, 2,...,n.
- The problem is to find an assignment with the minimum total cost.

Method: Solved by Exhaustive Search method.

Example:

A small instance of this problem follows, with the table entries representing the assignment costs C[i, j]:

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

- Cost matrix C.
- The problem calls for a selection of one element in each row of the matrix so that all selected element are in different columns and the total sum of the selected elements is the smallest possible.

Feasible solution:

- n-tuples < $j_1,...,j_n$ > in which the i^{th} component, i = 1,...,n, indicates the column of the element selected in the i^{th} row.

Example: cost matrix <2,3,4,1> - feasible assignment.
Person 1 to job 2
Person 2 to job 3
Person 3 to job 4

Person 4 to job 1

→There is a one-to-one correspondence between feasible assignment and permutation of the first n integers.

Exhaustive approach:

Step 1: Generating all the permutation of integers 1,2,….n.

Step 2: Computing the total cost of each assignment by summing up the corresponding elements of the cost matrix.

Step 3: Finally, selecting the one with the smallest sum.

Example : First few iterations of solving a small instance of the assignment problem by exhaustive search.

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

<1, 2, 3, 4> cost = 9 + 4 + 1 + 4 = 18
<1, 2, 4, 3> cost = 9 + 4 + 8 + 9 = 30
<1, 3, 2, 4> cost = 9 + 3 + 8 + 4 = 24
<1, 3, 4, 2> cost = 9 + 3 + 8 + 6 = 26
<1, 4, 2, 3> cost = 9 + 7 + 8 + 9 = 33
<1, 4, 3, 2> cost = 9 + 7 + 1 + 6 = 23

etc.

<2,1,3,4> cost = 2 + 6 + 1 +4 = 13 **Optimal**

Permutation → n! eg: 4! = 4.3.2.1 = 24

Efficient algorithm for this problem called the Hungarian method.

Time Complexity: O(n!)

2.3 Divide and Conquer Methodology

✓ Divide-and-conquer is probably the best-known general algorithm design technique.

General plan:

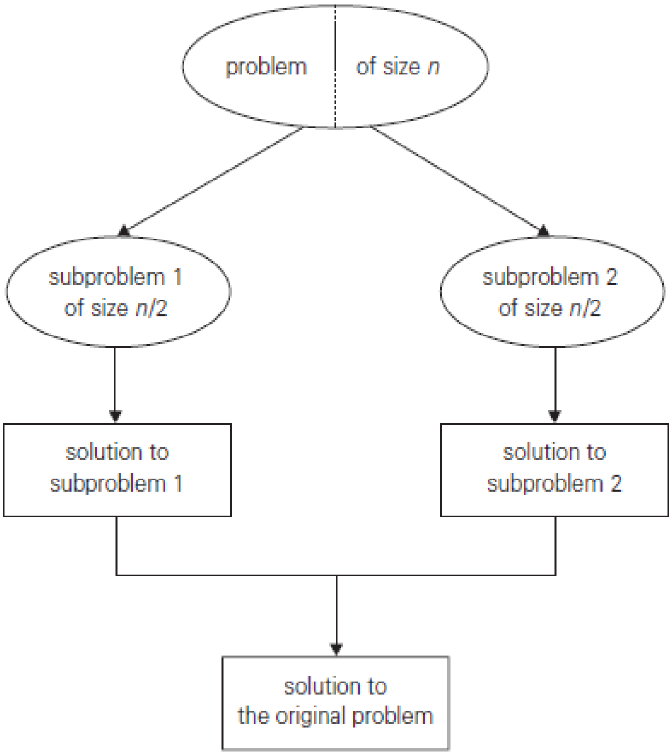
Divide-and-conquer algorithms work according to the following general plan:

1. A problem is divided into several sub problems of the same type, ideally of about equal size.

2. The sub problems are solved (typically recursively, though sometimes a different algorithm is employed, especially when sub problems become small enough).

3. If necessary, the solutions to the sub problems are combined to get a solutionto the original problem.

Divide and conquer technique



- Dividing a sub problem into two smaller sub problems.
  Example: The problem of computing the sum of n numbers $a_{0,\ldots\ldots,}a_{n-1.}$
- If $n > 1$, we can divide the problem into two instances of the same problem:
  - to compute the sum of the first $\lfloor n/2 \rfloor$ numbers and to compute the sumof the remaining $\lceil n/2 \rceil$ numbers.
- Once each of these two sums is computed, add their values to get the sum:
  $a_0+\ldots\ldots+a_{n-1} = (a_0+\ldots\ldots+a_{\lfloor n/2 \rfloor-1}) + (a_{\lfloor n/2 \rfloor}+\ldots+a_{n-1})$.
  - a problem's instance of size n is divided into two instances of size n/2.
  - an instance of size n can be divided into b instances of size n/b.
- Size n is a power of b, recurrence for the running time T (n):

General Divide and Conquer recurrence:

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

f(n) is a function that accounts for the time spent on dividing an instanceof size n into instances of size n/b and combining their solutions.

The order of growth of its solution T (n) depends on the values of the constants a and b and the order of growth of the function f (n).

- **Master Theorem:**

If $f(n) \in \theta(n^d)$ where $d \geq 0$ in recurrence equation, then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Example: Computing the sum of n numbers:

The recurrence for the number of additions A(n) made by the divide-and-conquer summation computation algorithm on inputs of sizen = 2k is

$$A(n) = 2A(n/2) + 1.$$

a = 2, b = 2, and d = 0; hence, since $a > b^d$,

$$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n).$$

this approach can only establish a solution's order of growth to within an unknown multiplicative constant , while solving a recurrence equation with a specific initial condition yields an exact answer.

Examples for divide and conquer:

- Binary Search
- Merge Sort
- Quick Sort
- Heap Sort
- Multiplication of large integers
- Closest pair problem
- Convex Hull Problem

Binary Search:

Definition: Binary Search is an efficient algorithm for searching an element in a sorted array.

Method: Divide and conquer.

Working:

comparing a search key K with the array's middle element A*m+.

If they match, the algorithm stops.

Otherwise, the same operation is repeated recursively for the first half of the array if K <A[m], and for the second half if K >A[m].

<u>Three conditions:</u>

$$\underbrace{A[0]\dots A[m-1]}_{\substack{\text{search here if}\\ K<A[m]}} \quad \overset{\overset{K}{\updownarrow}}{A[m]} \quad \underbrace{A[m+1]\dots A[n-1]}_{\substack{\text{search here if}\\ K>A[m]}}.$$

<u>Steps:</u>

Step 1: First find the middle element.

Step2: Compare the searching element with middle element. If they match the algorithm stops.

Step3: If k<A[m],search in the left side of the middle element.

Step4: If k>A[m], search in the right side of the middle element.

Step5: Recursively do the process until the element is found. If the element is not found in the listreturn -1.

<u>Algorithm:</u>

```
Binary Search(A*0..n − 1+, K)
l←0;
r ←n – 1
while l ≤ r do
  if K = A[m] return m
  else if  K <A[m]
        r ←m – 1
  else
       l ←m + 1
return −1
```

Example: binary search to searching for K = 70 in the array

3	14	27	31	39	42	55	70	74	81	85	93	98

iterations of the algorithm:

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1	*l*						*m*						*r*
iteration 2								*l*		*m*			*r*
iteration 3								*l,m*	*r*				

Analysis:

- count the number of times the search key is compared with an element of the array.three-way comparisons**:** k with A[m]

  i)k=A[m]

  ii)k<A[m

  iii)k>A[m]

Worst case:

- find the number of key comparisons
- inputs include all arrays that do not contain a given search key, as well as some successful searches.

Recurrence relation:

$$C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1$$

$$C_{worst}(1) = 1.$$

substitute $n=2^k$

Cworst (2k) = Cworst(2^{k-1})+1

=Cworst(2^{k-1})+2

.

.

.

$=C_{worst}(2^{k-k})+k=C_{worst}(1)+k=1+k$

$C_{worst}(n) = 1 + \log_2 n = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2(n+1) \rceil.$

worst-case time efficiency of binary search is $\Theta(\log n).$

Average case:

$$C_{avg}(n) \approx \log_2 n.$$

- Successful search: $C^{yes}_{avg}(n) \approx \log_2 n - 1$
- Unsuccessful search: $C^{no}_{avg}(n) \approx \log_2(n+1)$

Time Complexity:

Best Case	Average Case	Worst Case
$\theta(1)$	$\theta(\log_2 n)$	$\theta(\log_2 n)$

18

① - Divides the array into two equal halves and sorts the halves separately, then it merges the sorted halves. ⑪.

MERGE SORT

* Sorts a given array $A[0 \ldots n-1]$ by dividing it into two halves $A[0 \ldots \lfloor n/2 \rfloor - 1]$ and $A[\lfloor n/2 \rfloor \ldots n-1]$, sorting each of them recursively and then merging the two smaller sorted arrays into a single sorted one.

⑤

* ALGORITHM Mergesort($A[0 \ldots n-1]$)

// Sorts array $A[0 \ldots n-1]$ by recursive mergesort.
// Input : An array $A[0 \ldots n-1]$ of orderable elements
// Output : Array $A[0 \ldots n-1]$ sorted in nondecreasing order.

```
if n > 1
    copy A[0.. ⌊n/2⌋-1] to B[0.. ⌊n/2⌋ -1]
    copy A[⌊n/2⌋ .. n -1] to C[0.. ⌈n/2⌉ -1]
    Mergesort(B[0.. ⌊n/2⌋-1])
    Mergesort (C[0.. ⌈n/2⌉ -1])
    Merge (B, C, A)
```

③ → The merging of two sorted arrays can be done as follows:

Merging operation

* Two pointers are initialized to point to the first elements of the arrays being merged.
* Then the elements pointed to, are compared and the smaller of them is added to a new array being constructed;
* After that, the index of that smaller element is incremented to point to its immediate successor in the array it was copied from.
* This operation is continued until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.

* ALGORITHM Merge($B[0 \ldots p-1]$, $C[0 \ldots q-1]$, $A[0 \ldots p+q-1]$)

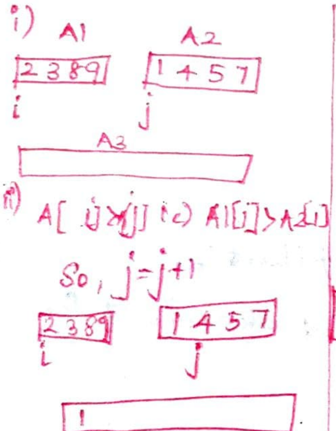
// Merges two sorted arrays into one sorted array.
// Input: Arrays $B[0 \ldots p-1]$ and $C[0 \ldots q-1]$ both sorted
// Output: Sorted array $A[0 \ldots p+q-1]$ of the elements of B and C

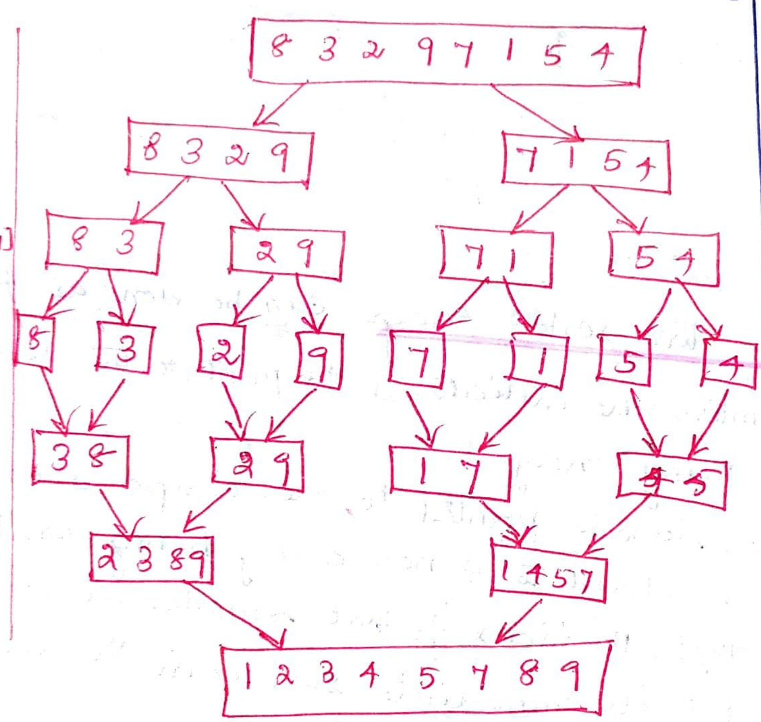
```
i ← 0; j ← 0; k ← 0
while i < p and j < q do
    if B[i] ≤ C[j]
        A[k] ← B[i]; i ← i+1
    else A[k] ← C[j]; j ← j+1
    k ← k+1
if i = p
    copy C[j..q-1] to A[k..p+q-1]
else copy B[i..p-1] to A[k..p+q-1]
```

(4) **Ex:**

An example of merge sort operation:

- The operation of the algorithm on the list 8, 3, 2, 9, 7, 1, 5, 4.





(2) **Method:**

Step 1: Divide the list into two.
→ even - equal sublist
→ odd - 1st one more elt.

Step 2: Split the sublists into two and go on until each of the sublists are of one.

Step 3: Merging the individual sublists to obtain a sorted list.

Efficient:

n is a power of 2.

- The recurrence relation for the number of key comparisons $C(n)$ is

$$C(n) = 2C(n/2) + C_{merge}(n) \text{ for } n > 1, \; C(1) = 0$$

$C_{merge}(n)$ - the number of key comparisons performed during the merging stage.

* At each step, exactly one comparison is made

Worst case: $\boxed{C_{merge}(n) = n-1}$, the recurrence is

$$\boxed{C_{worst}(n) = 2\,C_{worst}(n/2) + n - 1} \text{ for } n > 1, \; C_{worst}(1) = 0$$

* According to the Master Theorem, $C_{worst}(n) \in \boxed{\Theta(n \log n)}$

$n = 2^k$

$C_{worst}(n) = n \log_2 n - n + 1$

disadvantage: - linear amount of extra storage the algorithm requires.

Analysis:

In merge sort algorithm, 2 recursive calls are made.

Recurrence relation:

$$T(n) = T(n/2) + T(n/2) + cn \quad \text{for } n>1.$$
$$T(1) = 0$$

Two methods:

- obtain the complexity.

i) Master Theorem

ii) Substitution Method.

① Master Theorem:

Recurrence relation for merge sort:

$$T(n) = T(n/2) + T(n/2) + cn.$$
$$T(n) = 2T(n/2) + cn \quad \text{—— ①}$$
$$T(1) = 0.$$

Master Theorem:

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

Then $T(n) = \Theta(n^d \log n)$ if $a = b$.

eqn ①, $a = 2$, $b = 2$

$f(n) = cn$

$\Rightarrow n^d$ where $d = 1$

$\therefore T(n) = \Theta(n \log_2 n)$

The average and worst case complexity of merge sort is $\boxed{\Theta(n \log_2 n)}$

② Substitution Method:

Recurrence relation: $T(n) = 2T(n/2) + cn$ for $n>1$ —①

$T(1) = 0$

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

Substitute $n = 2^k$ in ①

$$T(n) = 2T(2^k/2) + c2^k$$
$$T(2^k) = 2T(2^{k-1}) + c2^k$$

By substitution method,

$$T(2^k) = 2\left[2T(2^{k-2}) + c\cdot 2^{k-1}\right] + c\cdot 2^k$$

$$= 2^2 T(2^{k-2}) + 2\cdot c\cdot 2^k\cdot 2^{-1} + c\cdot 2^k$$

$$= 2^2 T(2^{k-2}) + \cancel{2}\cdot c\cdot 2^k\cdot \frac{1}{\cancel{2}} + c\cdot 2^k$$

$$= 2^2 T(2^{k-2}) + c\cdot 2^k + c\cdot 2^k$$

$$T(2^k) = 2^2 T(2^{k-2}) + 2c\cdot 2^k$$

$$= 2^3 T(2^{k-3}) + 3\cdot c\cdot 2^k$$

$$= 2^4 T(2^{k-4}) + 4\cdot c\cdot 2^k$$

$$\vdots$$

$$= 2^k T(2^{k-k}) + k\cdot c\cdot 2^k$$

$$= 2^k T(2^0) + k\cdot c\cdot 2^k$$

$$T(2^k) = 2^k T(1) + k\cdot c\cdot 2^k$$

$$= 2^k\cdot(0) + k c 2^k$$

$$T(2^k) = k\cdot c\cdot 2^k$$

replace $2^k = n$.

$$n = 2^k$$

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

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

$$\log_2 n = k$$

$$T(n) = \log_2 n\cdot c\cdot n.$$

$$= n\log_2 n.$$

$$\boxed{T(n) = \Theta(n\log_2 n)}$$

$\therefore$ Average and worst case time Complexity is $\boxed{\Theta(n\log_2 n)}$

Time complexity of merge sort.

Best Case	Average Case	Worst Case
$\Theta(n\log_2 n)$	$\Theta(n\log_2 n)$	$\Theta(n\log_2 n)$

Examples: ① 123, 23, 1, 43, 54, 36, 75, 34

④ 38, 27, 43, 3, 9, 82, 10

② 286, 45, 278, 368, 475, 389, 656, 788, 503, 126

③ 12, 24, 8, 71, 4, 23, 6, 89, 56

QUICKSORT

- sorting algorithm - based on the divide and conquer approach.

mergesort:
* Divides its input's elements according to their position in the array

quicksort - divides the input's elements according to their value.
→ it rearranges elements of a given array $A[0..n-1]$ to achieve its partition, a situation where all the elements before some position s are smaller than or equal to $A[s]$ and all the elements after position s are greater than or equal to $A[s]$:

$$\underbrace{A[0] \ldots A[s-1]}_{\text{all are } \leq A[s]} \quad A[s] \quad \underbrace{A[s+1] \ldots A[n-1]}_{\text{all are } \geq A[s]}$$

- after a partition has been achieved, $A[s]$ will be in its final position in the sorted array, and we can continue sorting the two subarrays of the elements preceding and following $A[s]$ independently.

ALGORITHM: Quicksort ($A[l..r]$)
// Sorts a subarray by quicksort
// Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its
// left and right indices l and r
// Output: The subarray $A[l..r]$ sorted in nondecreasing order

```
if l < r
    s ← Partition (A[l..r])  // s is a split position
    Quicksort (A[l..s-1])
    Quicksort (A[s+1..r])
```

→ A partition of $A[0..n-1]$ can be achieved by the following algorithm:
i) First, select an element with respect to whose value that are going to divide the subarray. This element is pivot.
- Selecting the subarray's first element: $p = A[l]$

→ Procedures for rearranging elements to achieve a partition:
* two scans of the subarray. — left-to-right / right-to-left.
↳ comparing the subarray's elements with the pivot.

left-to-right scan → starts with the second element
- elements smaller than the pivot to be in the first part of the subarray.
- scan stops on encountering the first element greater than or equal to the pivot.

right-to-left scan → starts with the last element of the subarray.
→ elements larger than the pivot to be in the second part of the subarray.
→ skips over elements that are larger than the pivot
→ stops on encountering the first element smaller than or equal to the pivot.

* Three situations:-

i) If scanning indices i and j have not crossed, i.e) $\boxed{i<j}$ exchange A[i] and A[j] and resume the scans by incrementing i and decrementing j respectively.

p	all are $\le p$	$\xrightarrow{}i$ $\ge p$	...	$j\leftarrow$ $\le p$	all are $\ge p$

ii) If the scanning indices have crossed over, i.e) $\boxed{i>j}$, partition the array after exchanging the pivot with A[j]:

p	all are $\le p$	$j\leftarrow$ $\le p$	$\rightarrow i$ $\ge p$	all are $\ge p$

iii) If the scanning indices stop while pointing to the same element i.e) $\boxed{i=j}$, the value they are pointing to must be equal to p. Thus partitioned the array:

p	all are $\le p$	$\leftarrow i=j \rightarrow$ $=p$	all are $\ge p$

Pseudocode:

ALGORITHM: Partition (A[l..r])
// Partitions a subarray by using its first element as a pivot
// Input : A subarray A[l..r] of A[0..n-1], defined by its left and right
//   indices l and r (l < r)
// Output : A partition of A[l..r], with the split position returned as
//   this function's value
//

```
p ← A[l]
i ← l ; j ← r+1
repeat
    repeat i ← i+1  until A[i] ≥ p
    repeat j ← j-1  until A[j] ≤ p
    swap(A[i], A[j])
until i ≥ j
swap(A[i], A[j])   // undo last swap when i ≥ j
swap(A[l], A[j])
return j
```

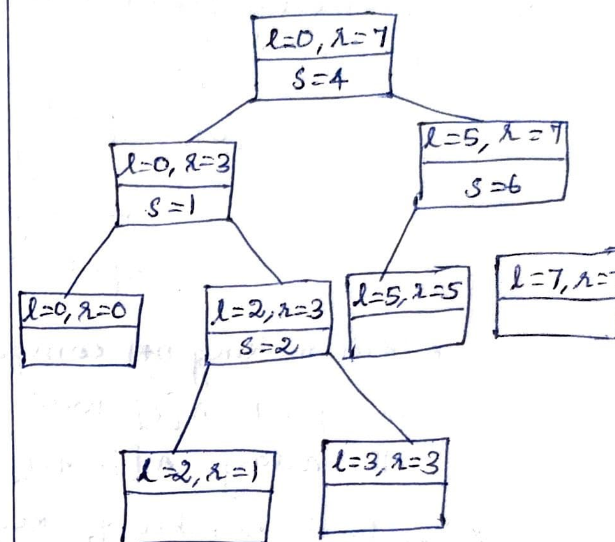
Example:

Sorting an array by quicksort

a) The array's transformations with pivots

0	1	2	3	4	5	6	7
5	3	1	9	8	2	4	7
5	3	1	9	8	2	4	7
5	3	1	4	8	2	9	7
5	3	1	4	8	2	9	7
5	3	1	4	2	8	9	7
5	3	1	4	2	8	9	7
				5	8	9	7
2	3	1	4				
2	3	1	4				
2	3	1	4				
2	1	3	4				
2	1	3	4				
1	2	3	4				
1							
		3	4				
		3	4				
			4				
					8	9	7
					8	7	9
					8	7	9
					7	8	9
					7		
							9

b) The tree of recursive calls to Quicksort with input values l and r of subarray bounds and split position s of a partition obtained.



Quicksort's efficiency:

- The number of key comparisons made before a partition is achieved is n+1, if the scanning indices cross over, n if they coincide.

* **Best case:**

- all the splits happen in the middle of corresponding subarrays.
- The number of key comparisons in the best case, $C_{best}(n)$ will satisfy the recurrence.

$$C_{best}(n) = 2C_{best}(n/2) + n \quad \text{for } n>1,$$

$$C_{best}(1) = 0$$

According to the Master Theorem, $C_{best}(n) \in \Theta(n \log_2 n)$

$n = 2^k$, yields $\boxed{C_{best}(n) = n \log_2 n}$

* worst case:
  - all the splits will be skewed to the extreme
  - One of the two subarrays will be empty while size of the other will be just one less than the size of a subarray being partitioned.

  * $A[0..n-1]$ - increasing array.

    $A[0]$ - pivot.

    → left to right scan - stop on $A[1]$

    → right to left scan - go all the way to reach $A[0]$

    split at position 0:

$$\begin{array}{c} 6\,(55) \\ \swarrow \quad \searrow \\ 2\,(27) \quad 9\,(81) \end{array}$$

* After making $n+1$ comparisons to get the partition and exchanging the pivot $A[0]$ with itself, the algorithm will find itself with the array $A[1..n-1]$ to sort.
* The total number of key comparisons made will be equal to

$$C_{worst}(n) = (n+1) + n + \dots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \boxed{\Theta(n^2)}$$

* Average Case:
  - The partition split can happen in each position $s\,(0 \le s \le n-1)$ with the same probability $1/n$, the recurrence relation

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n>1$$

$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$$

Its solution turns out to be

$$C_{avg}(n) \approx 2n \ln n \approx 1.38 n \log_2 n$$

- 38% more comparisons than in the best case.

Advantages:-
- better pivot selection method
- switching to a simpler sort on smaller subfiles
- recursion elimination.

Example: Sort the elements : 50 30 10 90 80 20 40 70

50 30 10 90 80 20 40 70

Step 1: Select the pivot element : P = A[l]

[50] 30 10 90 80 20 40 70
p i j

Step 2: - Increment i while A[i] < pivot.
- stop incrementing when it encounters the element larger than pivot

[50] 30 10 90 80 20 40 70
p i j

Step 3: Decrement j while A[j] > pivot.
- Stop decrement when it encounters the element smaller than pivot.

[50] 30 10 90 80 20 40 70
p i j

Step 4: i<j, Exchange A[i] and A[j] and start incrementing and decrementing i and j respectively.

[50] 30 10 40 80 20 90 70
p i j

Step 5: Increment i

[50] 30 10 40 80 20 90 70
p i j

Step 6: Decrement j

[50] 30 10 40 80 20 90 70
p i j

Step 7: i<j, exchange A[i] and A[j]

[50] 30 10 40 20 80 90 70
p i j

Step 8: Increment i

[50] 30 10 40 20 80 90 70
p ij

Step 9: Decrement j

[50] 30 10 40 20 80 90 70
p j i

Step 10: $\boxed{i > j}$, Exchange the pivot with A[j] and partition the array after exchanging.

$$\underbrace{20 \quad 30 \quad 10 \quad 40}_{\text{Left sub list}} \quad \underset{j}{50} \quad \underbrace{80 \quad 90 \quad 70}_{\text{Right sublist}}$$

Step 11: <u>Left sublist</u>

Increment i for the left sub list

$$\underbrace{\underset{p}{\boxed{20}} \quad \underset{i}{30} \quad 10 \quad \underset{j}{40}} \quad 50 \quad 80 \quad 90 \quad 70$$

Step 12: Decrement j for the left sublist

$$\underbrace{\underset{p}{\boxed{20}} \quad \underset{i}{30} \quad \underset{j}{10} \quad 40} \quad 50 \quad 80 \quad 90 \quad 70$$

Step 13: i < j, Exchange A[i] and A[j]

$$\underbrace{\underset{p}{\boxed{20}} \quad \underset{i}{10} \quad \underset{j}{30} \quad 40} \quad 50 \quad 80 \quad 90 \quad 70$$

Step 14: Increment i

$$\underbrace{\underset{p}{\boxed{20}} \quad 10 \quad \underset{ij}{30} \quad 40} \quad 50 \quad 80 \quad 90 \quad 70$$

Step 15: Decrement j

$$\underbrace{\underset{p}{\boxed{20}} \quad \underset{j}{10} \quad \underset{i}{30} \quad 40} \quad 50 \quad 80 \quad 90 \quad 70$$

Step 16: i > j, Exchange pivot with A[j]

$$\underbrace{10 \quad \underset{j}{20} \quad 30 \quad 40} \quad 50 \quad 80 \quad 90 \quad 70$$

Step 17: <u>Right Sub list</u>

Increment i & Decrement j

$$10 \quad 20 \quad 30 \quad 40 \quad 50 \quad \underbrace{\underset{p}{\boxed{80}} \quad \underset{i}{90} \quad \underset{j}{70}}$$

Step 18: i < j, Exchange A[i] and A[j]

$$10 \quad 20 \quad 30 \quad 40 \quad 50 \quad \underbrace{\underset{P}{\boxed{80}} \quad \underset{i}{70} \quad \underset{j}{90}}$$

Step 19: Increment i for the right sublist

10 20 30 40 50 $\boxed{80}$ 70 90

p (under 80), ij (under 90)

Step 20: Decrement j

10 20 30 40 50 $\boxed{80}$ 70 90

p (under 80), j (under 70), i (under 90)

Step 21: $i > j$, → Exchange $A[j]$ and pivot

→ Partition the Array.

10 20 30 40 50 $\boxed{70}$ ~~80~~ 80 90

p (under 70), j (under 80), i (under 90)

The final sorted list is

10 20 30 40 50 70 80 90

Analysis:

* Best case:

Recurrence relation

$$\boxed{\begin{array}{l} C(n) = C(n/2) + C.(n/2) + n \\ C(1) = 0 \end{array}}$$

Method : 1 : Master Theorem :

$$C(n) = 2\,C(n/2) + n.$$

$$f(n) = n$$

As per master Theorem,

$$C(n) = \Theta(n \log n)$$

Best case time complexity is $\boxed{\Theta(n \log_2 n)}$

Method 2 : Substitution Method :

$$C(n) = 2\,C(n/2) + n$$

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

$$C(2^k) = 2\,C(2^k/2) + 2^k$$

$$= 2\,C(2^{k-1}) + 2^k$$

Substitute, $C(2^{k-1})$

$$C(2^{k-1}) = 2\,C(2^{k-1}/2) + 2^{k-1}$$

$$= 2\,C(2^{k-2}) + 2^{k-1}$$

$$c(2^k) = 2\left[2\,c(2^{k-2}) + 2^{k-1}\right] + 2^k$$
$$= 2^2 c(2^{k-2}) + 2 \cdot 2^k \cdot 2^{-1} + 2^k$$
$$= 2^2 c(2^{k-2}) + \cancel{2} \cdot \frac{2^k}{\cancel{2}} + 2^k$$
$$= 2^2 c(2^{k-2}) + 2 \cdot 2^k$$

Similarly,

$$c(2^k) = 2^3 c(2^{k-3}) + 3 \cdot 2^k$$
$$c(2^k) = 2^4 c(2^{k-4}) + 4 \cdot 2^k$$
$$\vdots$$
$$c(2^k) = 2^k c(2^{k-k}) + k \cdot 2^k$$
$$= 2^k c(2^0) + k \cdot 2^k$$
$$= 2^k c(1) + k \cdot 2^k$$
$$= 2^k \cdot 0 + k \cdot 2^k$$
$$\boxed{c(2^k) = k \cdot 2^k}$$

replace. $2^k = n$

$$n = 2^k$$
$$\log_2 n = \log 2^k$$
$$= k \log 2$$
$$\boxed{\log_2 n = k}$$

$$c(n) = \log_2 n \cdot n$$
$$c(n) = n \log_2 n$$

Best case time complexity of Quick sort is $\boxed{\Theta(n \log_2 n)}$

Worst case: → when pivot is max or min of all the elts.

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

Time complexity:

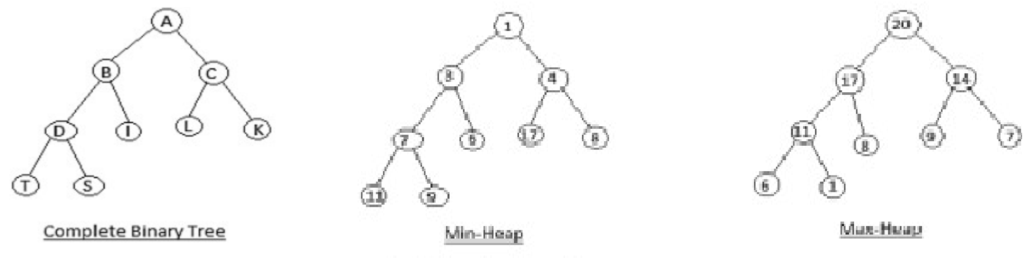
Best case	Average case	Worst Case
$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$\Theta(n^2)$

Example: ① 39, 20, 40, 14, 69, 61, 97, 31

HEAP SORT

Definition:

- Heap sort is a comparison based sorting technique based on Binary Heap data structure.
- First find the maximum element and place the maximum element at the end. Repeat the same process for remaining element.
- Heap sort is an efficient sorting algorithm with average and worst case time complexities are in O(n*log n).
- Heap sort is an in-place algorithm i.e. does not use any extra space, like merge sort.
- A heap can be defined as a binary tree with the following two conditions :
    - **The shape property**—the binary tree is complete,
        - i.e., all its levels are full except possibly the last level, where only some rightmost leaves may be missing.



    - **The heap property**—
        - Max heap - the key in each node is greater than or equal to the keys in its children
        - Minheap - the key in each node is Smaller than or equal to the keys in its children.

Method: Divide and Conquer

Steps: Consider an array Arr which is to be sorted using Heap Sort.

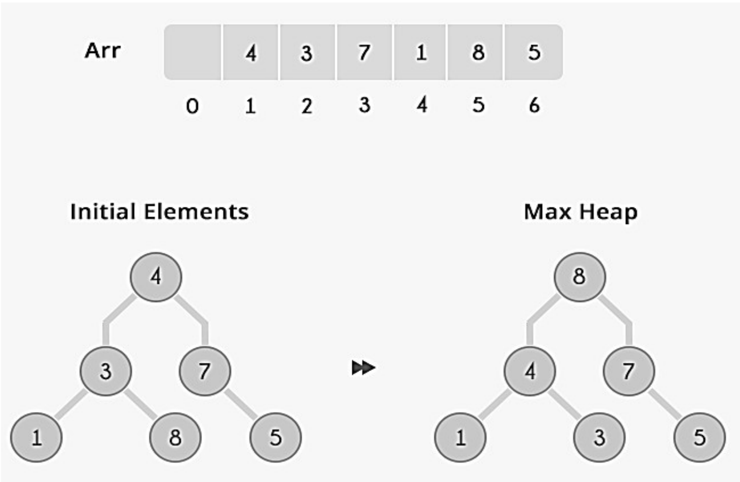
1. Initially build a max heap of elements in Arr.
2. The root element, that is Arr[1], will contain maximum element of Arr.
3. After that, swap this element with the last element of Arr and heapify the max heap excluding the last element which is already in its correct position and then decrease the length of heap by one.
4. Repeat the step 2, until all the elements are in their correct position

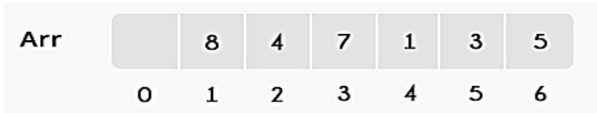
ALGORITHM

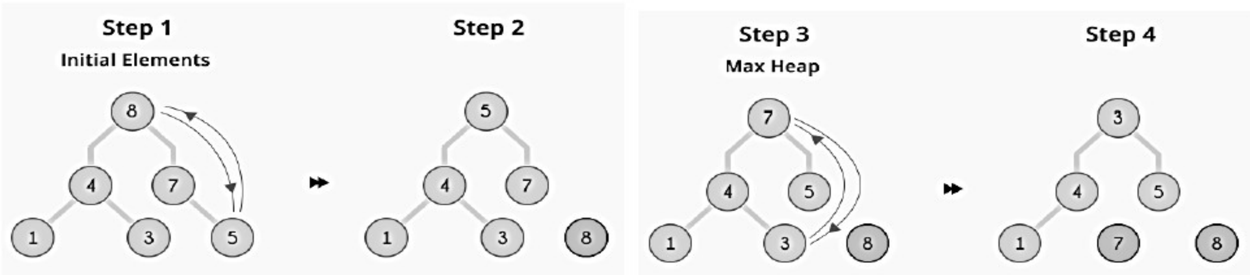
```
HeapBottomUp(H[1..n])
        //Input: An array H[1..n] of orderable items
        //Output: A heap H[1..n]
for i ←[n/2] downto 1 do k←i; v←H[k] heap←false
while not heap and 2 * k ≤ n do
j ←2 * k
if j <n          //there are two children
if H[j ]<H[j + 1]
        j ←j + 1
if v ≥ H[j ]
        heap←true
else
        H[k]←H[j ];   k←j H[k]←v
```

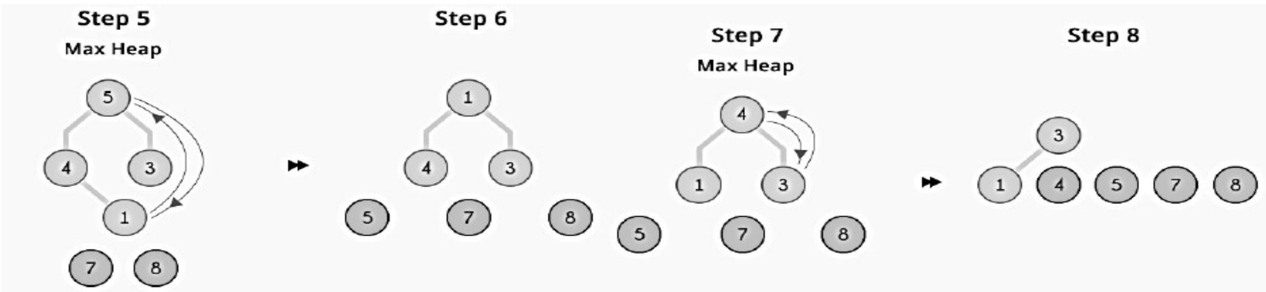
Example:

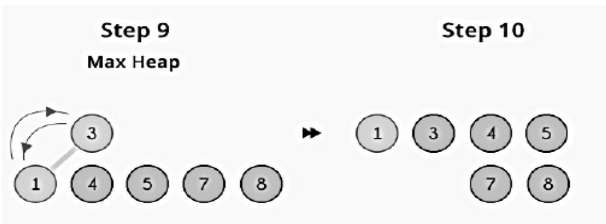
- Initially there is an unsorted array Arr having 6 elements and then max-heap will be built.
- After building max-heap, the elements in the array Arr will be:



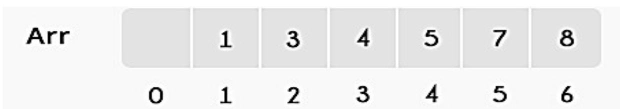








After all the steps, a sorted array is.



Analysis:

Worst Case Time Complexity	Best Case Time Complexity	Average Time Complexity
O(n*log n)	O(n*log n)	O(n*log n)

Space Complexity: O(1)

- Heap sort is not a Stable sort, and requires a constant space for sorting a list.
- Heap Sort is very fast and is widely used for sorting

CLOSEST - PAIR AND CONVEX- HULL PROBLEMS

CLOSEST- PAIR PROBLEM:

* Let P be a set of $n>1$ points in the Cartesian plane.
  - The points are distinct
  - The points are ordered in nondecreasing order of their x coordinate.
* Q - the points sorted in separate list in nondecreasing order of y coordinate.

→ If $2 \le n \le 3$, the problem can be solved by brute-force algm.

Divide & conquer method: → If $n>3$, divide the points into two subsets P_1 and P_2 of $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ points respectively, by drawing a vertical line through the median m of their x coordinates so that $\lceil n/2 \rceil$ points lie to the left of or on the line itself and $\lfloor n/2 \rfloor$ points lie to the right of or on the line.

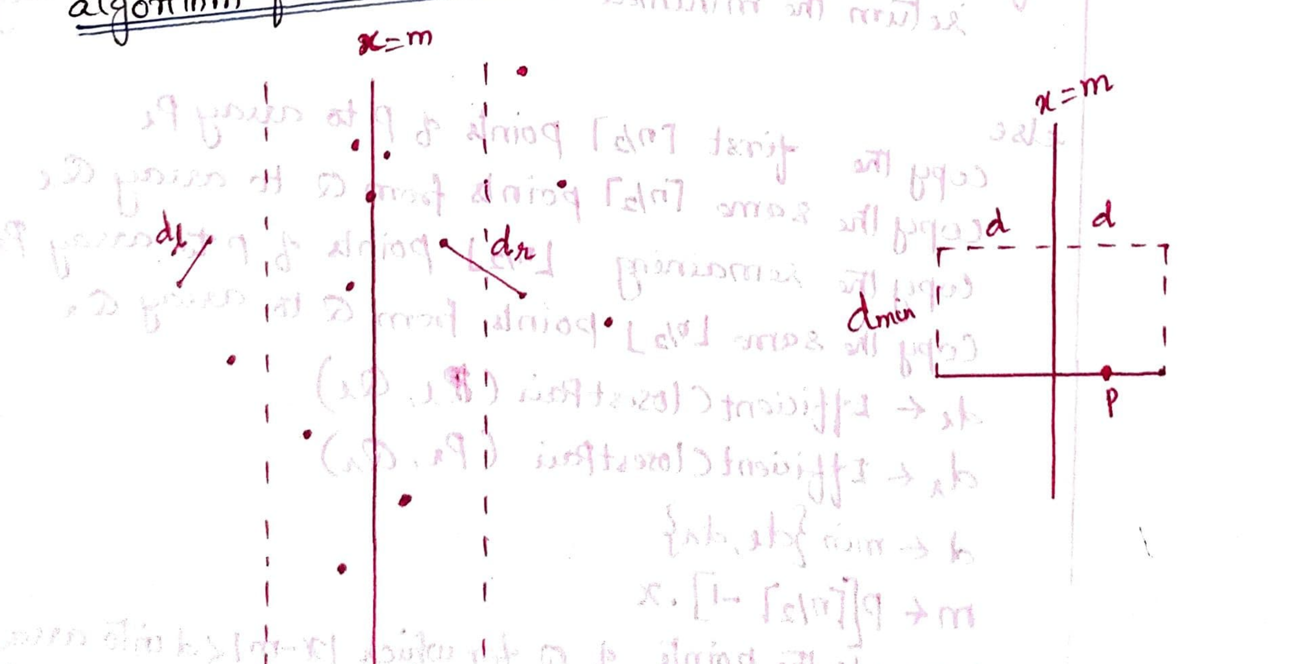
→ Solve the closest-pair problem recursively for subsets P_l and P_r

→ Let d_l - smallest distances between pairs of points in P_l.

d_r - smallest distances between pairs of points in P_r.

a) Idea of the divide and conquer algorithm for the closest pair problem

b) Rectangle that may contain points closer than d_{min} to point p



- The distance between any other pair of points is at least d.

→ Let S be the list of points inside the strip of width $2d$ around the separating line, obtained from Q and hence ordered in non decreasing order of their y coordinate.

d_{min} - the minimum distance

* Initially $d_{min} = d$ and subsequently $d_{min} \le d$.
* $p(x, y)$ - point on the list
* $p'(x', y')$ - closer to p than d_{min}; belong to the rectangle.
* the points in each half of the rectangle must be at least distance d apart.

Pseudocode:

ALGORITHM EfficientClosestPair (P, Q)

// Solves the closest-pair problem by divide and conquer

// Input : An array P of $n \ge 2$ points in the Cartesian plane sorted in

// nondecreasing order of their x coordinates and an array Q of the

// same points sorted in nondecreasing order of the y coordinates

// Output : Euclidean distance between the closest pair of points

```
if n ≤ 3
    return the minimal distance found by the brute-force algorithm
else
    copy the first ⌈n/2⌉ points of P to array Pl
    copy the same ⌈n/2⌉ points from Q to array Ql
    copy the remaining ⌊n/2⌋ points of P to array Pr
    copy the same ⌊n/2⌋ points from Q to array Qr
    dl ← EfficientClosestPair (Pl, Ql)
    dr ← EfficientClosestPair (Pr, Qr)
    d ← min {dl, dr}
    m ← P[⌈n/2⌉ - 1].x
    copy all the points of Q for which |x - m| < d into array S[0..num - 1]
    dminsq ← d²
```

```
for i ← 0 to num - 2 do
    k ← i+1
    while k ≤ num - 1 and (S[k].y - S[i].y)^2 < dminsq
        dminsq ← min ((S[k].x - S[i].x)^2 + (S[k].y - S[i].y)^2,
                                                        dminsq)
        k ← k+1
return sqrt(dminsq)
```

Analysis:

- linear time both for dividing the problem into two problems half the size and combining the obtained solutions.

Recurrence relation:

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

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

$f(n) \in \Theta(n)$

Master's theorem:

$a = 2, \quad b = 2, \quad d = 1$

$T(n) \in \Theta(n \log n)$

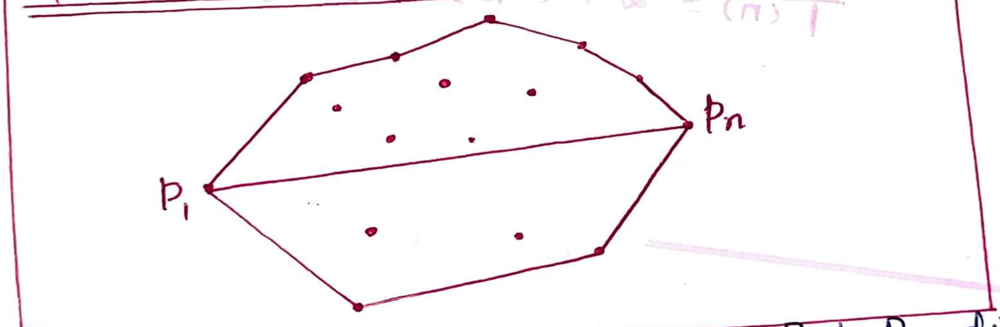
CONVEX - HULL PROBLEM

* Find the smallest convex polygon that contains n given points in the plane.
  → quick hull. - divide and conquer algorithm

* Let S be a set of $n > 1$ points $p_1(x_1, y_1), \ldots, p_n(x_n, y_n)$ in the Cartesian plane.
  - The points are sorted in nondecreasing order of their x coordinates, with ties resolved by increasing order of the y coordinates of the points involved.

→ the leftmost point p_1 and the rightmost point p_n are two distinct extreme points of the set's convex hull.

Upper and lower hulls of a set of points



* Let $\overrightarrow{P_1 P_n}$ - the straight line through points P_1 to P_n directed from P_1 to P_n.
  - This line separates the points into two sets:
    S_1 - the set of points to the left of or on this line
    S_2 - the set of points to the right of or on this line.

upper hull:
→ The convex hull of S_1 consists of the line segment with the end points at P_1 and P_n and an upper boundary made up of a polygonial chain
  i.e) a sequence of line segments connecting some points of S_1.
  - The upper boundary is called the upper hull.

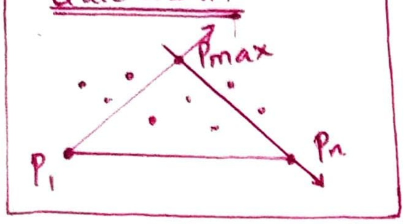
Lower hull:
→ The polygonial chain, which serves as the lower boundary of the convex hull of set S_2 is called the lower hull.

* The convex hull of the entire set S is composed of the upper and lower hulls

Construction of upper hull & lower hull :

* First, the algorithm identifies vertex P_{max} in S_1, which is the farthest from the line $\overrightarrow{P_1P_n}$.
  - If there is a tie, the point that maximizes the angle $\angle P_{max}P_1P_n$ can be selected.
* Then, the algorithm identifies all the points of set S_1 that are to the left of the line $\overrightarrow{P_1P_{max}}$;
  - these are the points that, along with P_1 and P_{max}, will make up the set $S_{1,1}$
  - The points of S_1 to the left of the line $\overrightarrow{P_{max}P_n}$ will make up, along with P_{max} and P_n, the set $S_{1,2}$.

Quick hull.



* The points inside $\Delta P_1P_{max}P_n$ can be eliminated
* The algorithm can continue constructing the upper hulls of $S_{1,1}$ and $S_{1,2}$ recursively
* Then, concatenate them to get the upper hull of the entire S_1.

Algorithm's geometric operations :

→ if $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$ and $P_3 = (x_3, y_3)$ are three arbitrary points in the plane, then the area of the triangle $\Delta P_1P_2P_3$ is equal to one half of the magnitude of the determinant

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1y_2 + x_3y_1 + x_2y_3 - x_3y_2 - x_2y_1 - x_1y_3$$

- the sign of the expression is positive if and only if the point $P_3 = (x_3, y_3)$ is to the left of the line $\overrightarrow{P_1P_2}$

Analysis :

~~Best~~ Average-case efficiency - $\Theta(n \log n)$

Worst-case efficiency - ~~$\Theta(n \log n)$~~ $\Theta(n^2)$

improvement → $\Theta(n \log n)$.

MULTIPLICATION OF LARGE INTEGERS

* manipulation of integers that are over 100 decimal digits long.

Multiplication :- each of the n digits of the first number is multiplied by each of the n digits of the second number for the total of n^2 digit multiplications.

Ex: Two-digit integers : 23 and 14.

23 can be represented as : $23 = 2.10^1 + 3.10^0$

$14 = 1.10^1 + 4.10^0$

multiply:

$$23 * 14 = (2.10^1 + 3.10^0) * (1.10^1 + 4.10^0)$$
$$= (2*1)\,10^2 + (2*4 + 3*1)\,10^1 + (3*4)\,10^0$$
$$= 2.10^2 + (8+3).10^1 + 12*1$$
$$= 2.100 + 11.10^1 + 12$$
$$= 200 + 110 + 12$$
$$= 322$$

$\rightarrow 2*4 + 3*1 = (2+3) * (1+4) - 2*1 - 3*4.$

* For any pair of two-digit numbers $a = a_1a_0$ and $b = b_1b_0$, their product c can be computed by the formula:

$$\boxed{c = a * b = c_2 10^2 + c_1 10^1 + c_0.}$$

where

$c_2 = a_1 * b_1$ is the product of their first digits

$c_0 = a_0 * b_0$ is the product of their second digits

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's digits and the sum of the b's digits minus the sum of c_2 and c_0

$\rightarrow$ Apply this to multiplying two n-digit integers a and b where n is a positive even number.

Divide and conquer Method:-

Step 1: * Divide the numbers in the middle

Step 2: — denote the first half of the a's digits by a_1 and the second half by a_0

— for b, the notations are b_1 and b_0

$a = a_1 a_0$ implies that $a = a_1 . 10^{n/2} + a_0$.

$b = b_1 b_0$ implies that $b = b_1 . 10^{n/2} + b_0$

Step 3: Multiplication is carried out using the formula.

$$c = a * b = (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0)$$

$$= (a_1 * b_1) 10^n + (a_1 * b_0 + a_0 * b_1) 10^{n/2} + (a_0 * b_0)$$

$$\boxed{c = c_2 10^n + c_1 . 10^{n/2} + c_0 .}$$

where

$c_2 = a_1 * b_1 \rightarrow$ product of first halves.

$c_0 = (a_0 * b_0) \rightarrow$ product of second halves.

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0) \rightarrow$ product of the sum of the a's halves and the sum of the b's halves minus the sum of c_2 and c_0

EX2: $2101 * 1130$

Example:- Multiply $2135 * 4014$

Soln: $c_2 = 840$, $c_1 = 1694$, $c_0 = 490$

$c = 840 \times 10^4 + 1694 \times 10^2 + 490$

Analysis:

* Multiplication of n-digit numbers requires three multiplications of $n/2$ -digit numbers

→ Recurrence for the number of multiplications $M(n)$ is:

$$\boxed{\begin{aligned} &M(n) = 3 M(n/2) \quad \text{for } n > 1, \\ &M(1) = 1 \end{aligned}}$$

⇒ Backward substitution: $\boxed{n = 2^k}$

$$M(2^k) = 3 . M(2^k/2) = 3 M(2^{k-1})$$

$$= 3 [3 . M(2^{k-1}/2)] = 3^2 M(2^{k-2})$$

$$\vdots$$

$$= 3^i M(2^{k-i})$$

$$\vdots$$

$$= 3^k M(2^{k-k})$$

$$M(2^k) = 3^k = 3^{\log n}$$

$$\boxed{M(n) = 3^{\log_2 n} = n^{\log_2 3}} \approx n^{1.585} \quad [\because a^{\log_b c} = c^{\log_b a}]$$

* Additions & Subtractions: $A(n)$

$3A(n/2)$ - needed to compute the three products of $n/2$ digit numbers

→ Recurrence:

$$\boxed{\begin{aligned} &A(n) = 3A(n/2) + cn \quad \text{for } n > 1, \\ &A(1) = 1 \end{aligned}}$$

require 5 additions & one subtraction

⇒ Apply Master theorem. $\boxed{A(n) \in \Theta(n^{\log_2 3})}$

$a = 3$, $b = 2$ } $a > b^d$ ⇓ $\Theta(n^{\log_b a})$

* Total number of additions & substractions have the same asymptotic order of growth as the number of multiplications

Multiplication of Large Integers:

Example 1:

Multiply $2135 * 4014$.

Soln:-

$a = 2135$, $b = 4014$

Step 1:

* Divide the numbers in the middle.

ie) $21 \mid 35$ $\quad 40 \mid 14$

Step 2:

* Denote the first half of a's digit by a_1 and the second half by a_0.
  - For b, the notations are b_1 and b_0.

$a = a_1 \; a_0$

$b = b_1 \; b_0$

$a = \underset{a_1}{21} \mid \underset{a_0}{35}$ $\quad b = \underset{b_1}{40} \mid \underset{b_0}{14}$

Step 3:

* Multiplication is carried out using the formula:

$$c = a * b = (a_1 * b_1) 10^n + (a_1 * b_0 + a_0 b_1) 10^{n/2} + (a_0 * b_0)$$

$$\therefore \boxed{c = c_2 10^n + c_1 . 10^{n/2} + c_0}$$

where

$c_2 = a_1 * b_1$

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$

(or)

$(a_1 * b_0) + (a_0 * b_1)$

$c_0 = a_0 * b_0$

$n \rightarrow$ number of digits
- positive even number.

Use Divide and Conquer Method.

i) Compute C_2:

$$C_2 = a_1 * b_1$$

$$C_2 = \underline{21 * 10}$$

→ 2-digit number.
→ Use D & C Method.

ie) Multiply $\underline{21 * 10}$. (2-digit no.)

Step 1: Divide the numbers in the middle.

ie) $a = 2|1 \quad b = 1|0$

Step 2: Denote. $a_1 a_0$ & $b_1 b_0$

ie) $a = \underset{a_1}{2}|\underset{a_0}{1} \quad b = \underset{b_1}{1}|\underset{b_0}{0}$

Step 3: $c = c_2 10^2 + c_1 10^1 + c_0$

$c_2 = a_1 * b_1$

$c_1 = (a_1 * b_0) + (a_0 * b_1)$

$c_0 = a_0 * b_0$.

a) Compute C_2

$C_2 = a_1 * b_1$

ie) $C_2 = 2 * 1 = 8$

b) Compute: C_1:

$C_1 = (a_1 * b_0) + (a_0 * b_1)$

$= 2 * 0 + 1 * 1$

$C_1 = 1$

c) Compute C_0:

$C_0 = a_0 * b_0$

$= 1 * 0$

$C_0 = 0$

$C = 8 \times 10^2 + 1 \times 10^1 + 0 = 800 + 10$

$C = 810$

$\boxed{C_2 = 810}$

ii) Compute C_1 :

$$C_1 = (a_1 * b_0) + (a_0 * b_1)$$

ie) $C_1 = (21 * 14) + (35 * 40)$

* Compute $21 * 14$:

Step 1: Divide

ie) $2 | 1 \quad 1 | 4$

Step 2: Denote :

$$\underset{a_1}{2} \,|\, \underset{a_0}{1} \qquad \underset{b_1}{1} \,|\, \underset{b_0}{4}$$

Step 3:

$$c = C_2 10^2 + C_1 10^1 + C_0$$

$$C_2 = a_1 * b_1 = 2 * 1 = 2$$

$$C_1 = (a_1 * b_0) + (a_0 * b_1) = (2*4) + (1*1)$$
$$= 8 + 1$$
$$= 9$$

$$C_0 = a_0 * b_0 = 1 \times 4 = 4$$

$$C = 2 \times 10^2 + 9 \times 10^1 + 4 = 200 \times 90 + 4 = \underline{C = 294}$$

* Compute $35 * 40$:-

Step 1: Divide

ie) $3 | 5 \quad 4 | 0$

Step 2: Denote

$$\underset{a_1}{3} \,|\, \underset{a_0}{5} \qquad \underset{b_1}{4} \,|\, \underset{b_0}{0}.$$

Step 3:

$$C = C_2 10^2 + C_1 10^1 + C_0$$

$$C_2 = (a_1 * b_1) = 3 * 4 = 12$$

$$C_1 = (a_1 * b_0) + (a_0 * b_1) = (3*0) + (5*4)$$
$$= 20$$

$$C_0 = a_0 * b_0 = 5 * 0 = 0$$

$$C = 12 \times 10^2 + 20 \times 10^1 + 0 = 1200 + 200 + 0 = 1400$$

$$C = 1400$$

$$\Rightarrow C_1 = 294 + 1400$$

$$\boxed{C_1 = 1694}$$

iii) Compute C_0 :

$C_0 = a_0 * b_0$

ie) $C_0 = 35 * 14$

Step 1:

Divide.

3 | 5   1 | 4

Step 2: Denote :

$\underset{a_1}{3} \,|\, \underset{a_0}{5}$ $\underset{b_1}{1} \,|\, \underset{b_0}{4}$

Step 3:

$c = c_2 10^2 + c_1 10^1 + c_0$

$c_2 = a_1 * b_1 = 3 * 1 = 3$

$c_1 = (a_1 * b_0) + (a_0 * b_1) = (3*4) + (5*1)$
$= 12 + 5 = 17$

$c_0 = a_0 * b_0 = 5 * 4 = 20$

$c = 3 \times 10^2 + 17 \times 10^1 + 20 = 300 + 170 + 20$

$c = 490$

$\boxed{C_0 = 490}$

$C = C_2 10^n + C_1 10^{n/2} + C_0$

$C = 840 \times 10^4 + 1694 \times 10^2 + 490$

$= 8400000 + 169400 + 490$

Ans: $\boxed{C = 8569890}$

Example 2:

Multiply 2101 * 1130.

UNIT III

DYNAMIC PROGRAMMING AND GREEDY TECHNIQUE

Dynamic programming – Principle of optimality - Coin changing problem, Computing a Binomial Coefficient – Floyd‘s algorithm – Multi stage graph - Optimal Binary Search Trees – Knapsack Problem and Memory functions. Greedy Technique – Container loading problem - Prim‘s algorithm and Kruskal's Algorithm – 0/1 Knapsack problem, Optimal Merge pattern - Huffman Trees.

2m

3.1 DYNAMIC PROGRAMMING

→ algorithm design technique.

→ general method for optimizing multistage decision processes

* programming – planning.

2marks

* Dynamic programming is a technique for solving problems with overlapping subproblems.

→ The subproblems arise from a recurrence relating a solution to a given problem with solutions to its smaller subproblems of the same type.

→ Rather than solving overlapping subproblems again and again, dynamic programming suggests solving each of the smaller subproblems only once and recording the results in a table, from which we can then obtain a solution to the original problem.

Example:

→ Fibonacci numbers – elements of the sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Recurrence:

$$F(n) = F(n-1) + F(n-2) \text{ for } n > 1$$

Two Initial Conditions:

$$F(0) = 0$$
$$F(1) = 1$$

* The problem of computing $F(n)$ is expressed in terms of its smaller and overlapping subproblems of computing $F(n-1)$ & $F(n-2)$

→ An algorithm based on the bottom-up dynamic programming approach.

2m
3.1.1 Principle of optimality:

- An optimal solution to any instance of an optimization problem is composed of an optimal solution to its subinstances.
- In an optimal sequence of choices or decisions, each subsequence must also be optimal

2m
Differences

	Divide and Conquer	Dynamic Programming
1.	The problem is divided into small subproblems. - The subproblems are solved independently - all the solutions of subproblems are collected together to get the solution to the given problem.	- many decision sequences are generated and all the overlapping sub-instances are considered.
2.	less efficient	- efficient than divide & Conquer strategy.
3.	top-down approach	- bottom-up approach.
4.	splits its input at specific deterministic points.	- splits its input at every possible points

x ———————— x

3.1.2 Coin Changing problem

- Give change for amount n using the minimum number of coins of denominations $d1<d2 < . . .<dm$.
- Dynamic programming algorithm
  - *assuming* availability of unlimited quantities of coins for each of the m denominations $d1< d2 < . . . < dm$ where $d1 = 1$.
  - Let $F(n)$ be the minimum number of coins whose values add up to n;
  - define $F(0) = 0$.
  - The amount n can only be obtained by adding one coin of denomination dj to the amount $n - dj$ for $j = 1, 2, . . . , m$ such that $n \geq dj$.
  - consider all denominations and select the one minimizing $F(n - dj) + 1$.
    - 1 is a constant
    - find the smallest $F(n - dj)$ first and then add 1 to it.
- Recurrence for $F(n)$:

$$F(n) = \min_{j:n \geq d_j} \{F(n - d_j)\} + 1 \quad \text{for } n > 0,$$
$$F(0) = 0.$$

- compute $F(n)$ by filling a one-row table left to right
- computing a table entry here requires finding the minimum of up to m numbers.

Example:

- Amount $n = 6$ and denominations 1, 3,4. Find the denominations of coins.

$F[0] = 0$

n	0	1	2	3	4	5	6
F	0						

$F[1] = \min\{F[1 - 1]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1					

$F[2] = \min\{F[2 - 1]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2				

$F[3] = \min\{F[3 - 1], F[3 - 3]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1			

$F[4] = \min\{F[4 - 1], F[4 - 3], F[4 - 4]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1	1		

$F[5] = \min\{F[5 - 1], F[5 - 3], F[5 - 4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	

$F[6] = \min\{F[6 - 1], F[6 - 3], F[6 - 4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	**2**

- To find the coins of an optimal solution
  - backtrace the computations to see which of the denominations produced the minima
  - the minimum was produced by $d2 = 3$.
  - The second minimum (for $n = 6 - 3$) was also produced for a coin of that denomination.
  - Thus, the minimum-coin set for $n = 6$ is **two 3's**.
- The answer it yields is **two coins.**

ALGORITHM

ChangeMaking(D[1..*m*]*, n)*

//Applies dynamic programming to find the minimum number of coins

//of denominations $d1 < d2 < \ldots < dm$ where $d1 = 1$ that add up to a

//given amount n

//Input: Positive integer n and array $D[1..m]$ of increasing positive

// integers indicating the coin denominations where $D[1] = 1$

//Output: The minimum number of coins that add up to n

$F[0] \leftarrow 0$

for $i \leftarrow 1$ **to** n **do**

$temp \leftarrow \infty$; $j \leftarrow 1$

while $j \leq m$ **and** $i \geq D[j]$ **do**

$temp \leftarrow \min(F[i - D[j]], temp)$

$j \leftarrow j + 1$

$F[i] \leftarrow temp + 1$

return $F[n]$

Analysis:

- The time efficiency of the algorithm =O(nm) and
- space efficiency of the algorithm ɵ(n)

3.1.3 COMPUTING A BINOMIAL COEFFICIENT

- nonoptimization problem
- example of dynamic programming.

Binomial Coefficient:

* A binomial is an algebraic expression that contains two terms

$x+y$.

$$(x+y)^2 = 1\cdot x^2 + 2\cdot xy + 1y^2$$

$$(x+y)^3 = 1x^3 + 3x^2y + 3xy^2 + 1y^3$$

* The numbers that appear as the coefficients of the terms in a binomial expression, called binomial coefficients

→ It is denoted as $\boxed{C(n,k) \text{ or } \binom{n}{k}}$

— the number of combinations (subsets) of k elements from an n-element set $(0 \le k \le n)$

→ binomial formula:

$$\boxed{(a+b)^n = C(n,0)a^n + \dots + C(n,k)a^{n-k}b^k + \dots + C(n,n)b^n}$$

→ properties of binomial coefficients

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

and

$$C(n,0) = C(n,n) = 1$$

dynamic programming:

* Computing $C(n,k)$ in terms of the smaller and overlapping problems of computing $C(n-1,k-1)$ and $C(n-1,k)$.

→ Record the values of the binomial coefficients in a table of n+1 rows and k+1 columns, numbered from 0 to n and from 0 to k respectively.

Table

* To compute $C(n,k)$, fill the table row by row, starting with row 0 and ending with row n.
* Each row i $(0 \le i \le n)$ is filled left to right, starting with 1 because $C(n,0) = 1$

* Rows 0 through k also end with 1 on the table's main diagonal.

$C(i, i) = 1$ for $0 \le i \le k$.

Table: Computing binomial coefficient $C(n,k)$ by the dynamic programming algorithm

	0	1	2	...	k-1	k
0	1					
1	1	1				
2	1	2	1			
⋮						
k	1					1
⋮						
n-1	1				$C(n-1,k-1)$	$C(n-1,k)$
n	1					$C(n,k)$

* Compute other entries by the formula, adding the contents of the cells in the preceding row and the previous column and in the preceding row and the same column.

Pseudocode :-

```
ALGORITHM Binomial (n,k)
// Computes C(n,k) by the dynamic programming algorithm
// Input : A pair of nonnegative integers n ≥ k ≥ 0
// output : The value of C(n,k)
for i ← 0 to n do
    for j ← 0 to min (i,k) do
        if j = 0 or j = k
            C[i,j] ← 1
        else C[i,j] ← C[i-1,j-1] + C(i-1,j]
return C[n,k]
```

Analysis :

Time efficiency of the algorithm.

→ The algorithm's basic operation is addition.

* $A(n,k)$ → total number of additions in computing $C(n,k)$

→ k+1 rows of the table form a triangle

→ remaining n−k rows form a rectangle.

* Split the sum expressing A(n,k) into two parts:

$$A(n,k) = \sum_{i=1}^{k}\sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^{n}\sum_{j=1}^{k} 1$$

$$= \sum_{i=1}^{k}(i-1) + \sum_{i=k+1}^{n} k$$

$$= \frac{(k-1)k}{2} + k(n-k) = \frac{k^2-k}{2} + nk - k^2$$

[∵ k is a constant]
[take only n term]

$$= \boxed{\Theta(nk)}$$

Formula

$$\boxed{\begin{array}{l} C(n,k) = C(n-1,k-1) + C(n-1,k) \\ C(n,0) = C(n,n) = 1, \quad n > k > 0 \end{array}}$$

Example:

① → Compute C(3,2)

Step 1

C(3,2) = C(2,1) + C(2,2)

Step 2

C(2,1) = C(1,0) + C(1,1)

= 1+1 = 2

C(2,2) = 1

Step 3

C(3,2) = 2+1 = 3

C(3,1) = C(2,0)+C(2,1) = 1+2 = 3

n/k	0	1	2
0	1		
1	1	1	
2	1	2	1
3	1	3	3

② Compute C(4,2)

⇒ C(4,2) = C(3,1)+C(3,2)

⇒ C(3,1) = C(2,0)+C(2,1)

⇒ C(2,1) = C(1,0)+C(1,1).

= 1+1 = 2

$\boxed{C(2,1) = 2}$

C(3,1) = 1+2 = 3

$\boxed{C(3,1) = 3}$

⇒ C(3,2) = C(2,1) + C(2,2)

= 2+1 = 3

$\boxed{C(3,2) = 3}$

C(4,2) = 3+3 = 6

$\boxed{C(4,2) = 6}$

n/k	0	1	2
0	1	1	
1	1		
2	1	2	1
3	1	3	3
4	1	4	6

Ex 3 C(10,5)

UNIT III

Problems

1) Computing Binomial coefficient:

C(10,5)

Step 1: Table:

column → 0 to k ie) 0 to 5

row → 0 to n ie) 0 to 11

n/k	0	1	2	3	4	5
0	1					
1	1	1				
2	1	2	1			
3	1	$\boxed{3}$	3	1		
4	1	4	6	4	1	
5	1	5	10	10	5	1
6	1	6	15	20	15	6
7	1	7	21	35	35	21
8	1	8	28	56	70	56
9	1	9	36	84	126	126
10	1	10	45	120	210	$\boxed{252}$

← c(n,k) C(10,5)

126+126 = 252

C(n,0) = 1

C(n,n) = 1

c(n,k) = c(n-1,k-1) + c(n-1,k)

ie) adding contents of preceding row same column (n-1,k) & previous column (n-1,k-1)

C(10,5) = C(9,4) + c(9,5)

compute c(9,4) + c(9,5)

3.1.4 FLOYD'S ALGORITHM

Defn:

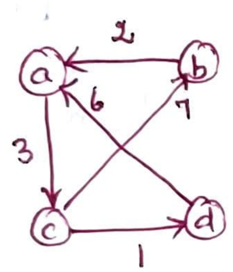
* Given a weighted connected graph (undirected or directed), the all-pairs shortest-paths problem asks to find the distances (the lengths of the shortest paths) from each vertex to all other vertices

Distance matrix:

* Record the lengths of shortest paths in an n-by-n matrix D called the distance matrix.
  - The element d_{ij} in the i^{th} row and the j^{th} column of the matrix indicates the length of the shortest path from the i^{th} vertex to the j^{th} vertex $(1 \le i, j \le n)$.

Example:

a) Digraph



b) Weight matrix

$$W = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

$a \to a, b \to b, c \to c, d \to d = 0$

c) Distance matrix

$$D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

Floyd's algorithm:

- generate the distance matrix.
- inventor → R. Floyd.
- applicable to both undirected and directed weighted graphs provided that they do not contain a cycle of a negative length.

* Floyd's algorithm computes the distance matrix of a weighted graph with n vertices through a series of n-by-n matrices:

$$D^{(0)}, \ldots, D^{(k-1)}, D^{(k)}, \ldots, D^{(n)}$$

- Each of the matrices contains the lengths of shortest paths with certain constraints on the paths.

* The element $d_{ij}^{(k)}$ in the i^{th} row and the j^{th} column of matrix $D^{(k)}$ $(k = 0, 1, \ldots n)$ is equal to the lengths of the shortest path

among all paths from the i^{th} vertex to the j^{th} vertex with each intermediate vertex, numbered not higher than k.

- * The series starts with $D^{(0)}$ → does not allow any intermediate vertices in the paths.

Step 1: Construct $D^{(0)}$ - weight matrix of the graph.

Step 4: $D^{(n)}$ - Contains the lengths of the shortest paths among all paths that can use all n vertices as intermediate.

Step 3: * Compute all the elements of each matrix $D^{(k)}$ from its immediate predecessor $D^{(k-1)}$.

$d_{ij}^{(k)}$ - the element in the i^{th} row and the j^{th} column of matrix $D^{(k)}$.

Step 2: $D^{(1)}$ from $D^{(0)}$ - include intermediate vertex.

- is equal to the length of the shortest path among all paths from the i^{th} vertex v_i to the j^{th} vertex v_j with their intermediate vertices numbered not higher than k.

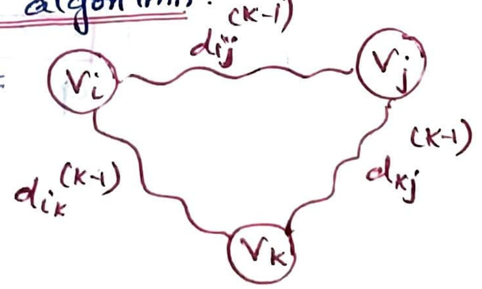
v_i, a list of intermediate vertices each numbered not higher than k, v_j

- partition all paths into two disjoint subsets.
  i) do not use the k^{th} vertex v_k as intermediate
  ii) use k^{th} vertex v_k as intermediate.

- Paths have the following form:
  v_i, vertices numbered $\leq k-1$, v_k, vertices numbered $\leq k-1$, v_j.

Idea of Floyd's algorithm:

Graphical Representation:



* Each of the paths is made up of a path from v_i to v_k with each intermediate vertex numbered not higher than k-1 and a path from v_k to v_j with each intermediate vertex numbered not higher than k-1.

- length of the shortest path from v_i to v_k → $d_{ik}^{(k-1)}$
- length of the shortest path from v_k to v_j → $d_{kj}^{(k-1)}$

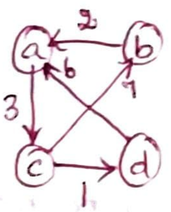
* <u>Recurrence</u> for finding the lengths of the shortest paths

$$d_{ij}^{(k)} = \min\left\{ d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right\} \text{ for } k \geq 1,\ d_{ij}^{(0)} = w_{ij}$$

* The element in the i^{th} row and the j^{th} column of the current matrix $D^{(k-1)}$ is replaced by the sum of the elements in the same row i and the k^{th} column and in the same column j and the k^{th} column if and only if the latter sum is smaller than its current value.

3.3.3 Application of Floyd's algorithm.

Example:



Step 1:

$$D^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \infty & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \infty & 0 \end{array}$$

→ lengths of the shortest paths with no intermediate vertices.

Step 2:

$k=1$ $i=1, j=1$

$d_{11}^{(1)} = \min\{ d_{11}^{(0)}, d_{11}^{(0)} + d_{11}^{(0)} \}$

$= \min\{0, 0+0\} = 0$

$d_{12}^{(1)} = \min\{ d_{12}^{(0)}, d_{11}^{(0)} + d_{12}^{(0)} \}$

$= \min\{\infty, 0+\infty\} = \infty$

⋮

$d_{23}^{(1)} = \min\{ d_{23}^{(0)}, d_{21}^{(0)} + d_{13}^{(0)} \}$

$= \min\{\infty, 2+3\}$

$= \min\{\infty, 5\}$

$= 5$

$$D^{(1)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & 5 & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & 9 & 0 \end{array}$$

→ Lengths of the shortest paths with intermediate vertices numbered not higher than 1.

Step 3:

$$D^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & 5 & \infty \\ c & 9 & 7 & 0 & 1 \\ d & 6 & \infty & 9 & 0 \end{array}$$

→ Lengths of the shortest paths with intermediate vertices numbered not higher than 2.
i.e) a & b.

Step 4:

$$D^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & 9 & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{array}$$

→ Lengths of the shortest paths with intermediate vertices numbered not higher than 3.
i.e) a, b & c.

Step 5:

$$D^{(4)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{cccc} a & b & c & d \\ \end{array}$$

$$\begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & 7 & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{array}$$

→ Lengths of the shortest paths with intermediate vertices numbered not higher than 4

→ ie) a, b, c & d

Pseudocode:

ALGORITHM Floyd(W[1..n, 1..n])

// Implements Floyd's algorithm for the all-pairs shortest-paths problem

// Input : The weight matrix W of a graph

// Output : The distance matrix of the shortest paths' lengths

```
D ← W
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            D[i,j] ← min{ D[i,j], D[i,k] + D[k,j] }
return D
```

Analysis:

Time efficiency = $\boxed{\Theta(n^3)}$

* Basic operation - computation of $D^{(k)}[i,j]$

$$C(n) = \sum_{k=1}^{n} \sum_{i=1}^{n} \sum_{j=1}^{n} 1$$

$$= \sum_{k=1}^{n} \sum_{i=1}^{n} (n-1+1)$$

$$= n \sum_{k=1}^{n} \sum_{i=1}^{n} 1 = n \sum_{k=1}^{n} (n-1+1)$$

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

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

$$= n^3$$

$$C(n) \in \boxed{\Theta(n^3)}$$

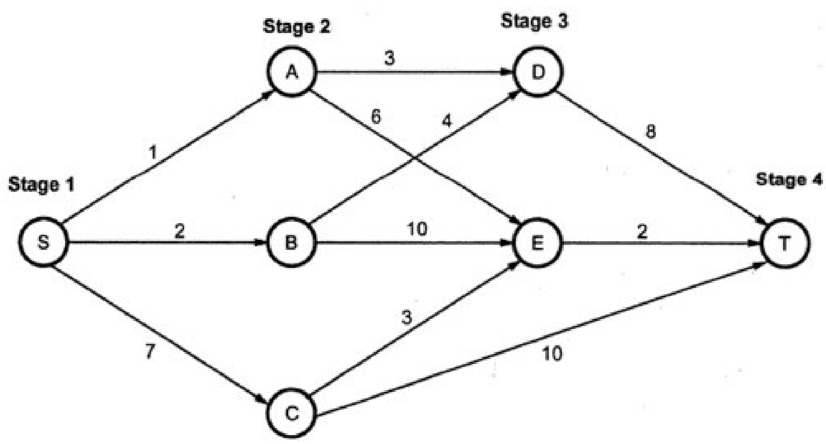
x ——————— x

3.1.5 Multi-Stage Graph(Finding Shortest path)

- To find the shortest path from source(S) to sink(T) in a multistage graph of G=(V,E) which is a directed graph.
- A **Multistage graph** is a directed graph in which the nodes can be divided into a set of stages such that all edges are from a stage to next stage only
  - All the vertices are partitioned into the k stages where k>=2.
  - Each stage consists of set of vertices
  - The cost of a path from source (denoted by S) to sink (denoted by T) is the sum of the costs of edges on the path.
- Dynamic Programming Method
  - obtain the minimum path at each current stage by considering the path length of each vertex obtained in earlier stage.
  - The sequence of decisions is taken by considering overlapping solutions.
- The multistage graph can be solved using
  - Forward approach
  - Backward approach.

Example:

- **Stage 1 consists of node S, Stage 2 consists of nodes A,B,C, Stage 3 consists of nodes D and E, Stage 4 consists of node T**



i)Backward approach:

d(S, T)=min {1+d(A, T),2+d(B,T),7+d(C,T)} …(1)

Compute d(A,T), d(B,T) and d(C,T).

d(A,T)=min{3+d(D,T),6+d(E,T)} …(2)
d(B,T)=min{4+d(D,T),10+d(E,T)} …(3)
d(C,T)=min{3+d(E,T),d(C,T)} …(4)

Compute d(D,T) and d(E,T).

d(D,T)=8
d(E,T)=2

backward vertex=E

- Put these values in equations (2), (3) and (4)

d(A,T)=min{3+8, 6+2}
d(A,T)=8 and the Path is **A-E-T**
d(B,T)=min{4+8,10+2}
d{B,T}=12 and the Path is **A-D-T**
d(C,T)=min(3+2,10)
d(C,T)=5 and the Path is **C-E-T**

Substitute these values of equations (2), (3) and (4) in (1),

d(S,T)	=	min{1+d(A,T), 2+d(B,T), 7+d(C,T)}
	=	min{1+8, 2+12,7+5}
	=	min{9,14,12}

d(S,T)=9 and the Path is **S-A-E-T**

Solution:

- Shortest distance from Source node(S) to Sink Node(T) is:
  The path with minimum cost is **S-A-E-T with the cost 9**.

Algorithm for Backward Approach

Algorithm Backward_Graph (G, K, n, p)
//solve multistage graph using forward approach
//**Input**:Given a weighted Graph G
//**output**: Path with minimum cost using Backward approach
b_cost [1]<- 0
For j = 2 to n **do**
r<-get-min(j,n)
b_cost[r]<- b_cost [r] + c [r, j];
D[j] = r;
// **find a minimum cost path**
P[1] = 1;
p[k] = n;
For j = k-1 to 2 **do**
p[j] = d[p(j+1)];

Analysis:

- Time complexity O(|V| + |E|).
  - |V| is the number of vertices and
  - |E| is the number of edges.

ii)Forward approach

d(S,A)=1
d(S,B)=2
d(S,C)=7
d(S,D) = min{1+d(A,D),2+d(B,D)}
= min{1+3,2+4}
d(S,D)=4

d(S,E) = min{1+d(A,E), 2+d(B,E),7+d(C,E)}
= min {1+6,2+10,7+3}
= min {7,12,10}
d(S,E) = 7 i.e. Path S-A-E is chosen.

d(S,T) = min{d(S,D)+d(D,T),d(S,E)+d(E,T),d(S,C)+d(C,T)}
= min {4+8,7+2,7+10}
d(S,T) = 9
Path S-E, E-T is chosen.

Solution:

- Shortest path and distance from Source node(S) to Sink Node(T) is:
  The minimum **cost=9 with the path S-A-E-T.**

Algorithm for Forward Approach:

Algorithm Forward_graph (G, K, n, p[])
//solve multistage graph using forward approach
// **Inpu**t:Given a weighted Graph G
// **output: path with minimum cost**
For j = n-1 to 1 do
Let r be a vertex such that is an edge of G and
C[j][r]+ cost[r] is minimum;
Cost [j] = C[j][r] + Cost[r]
D [j] = r
P [1] = 1
P[k] = n
For j = 2 to K-1 do
P[j] = d[P(j-1)];

Analysis:

- Time complexity O(|V| + |E|). Where the |V| is the number of vertices and |E| is the number of edges

3.1.6 OPTIMAL BINARY SEARCH TREES

- A binary search tree is one of the data structures.

application → to implement a dictionary, a set of elements with the operations of searching, insertion and deletion.

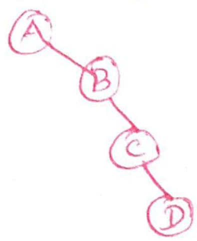
- probabilities of searching for elements of a set are known.

optimal binary search tree → the average number of comparisons in a search is the smallest possible.

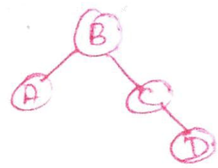
Example

* Four keys A, B, C & D. to be searched for with probabilities 0.1, 0.2, 0.4 and 0.3 respectively.
  - two out of 14 possible binary search trees containing keys.

(a)



(b)



→ Average number of comparisons in the successful search.

$= 0.1(1) + 0.2(2) + 0.4(3) + 0.3(4)$

$= 0.1 + 0.4 + 1.2 + 1.2$

$= 2.9$

→ $= 0.1(2) + 0.2(1) + 0.4(2) + 0.3(3)$

$= 0.2 + 0.2 + 0.8 + 0.9$

$= 2.1$

→ find the optimal tree by generating all 14 binary search trees with keys

* Catalan number.
  * Total number of binary search trees with n keys is equal to the n^{th} catalan number.

$$C(n) = \binom{2n}{n}\frac{1}{n+1} \text{ for } n > 0, \quad C(0) = 1$$

which grows to infinity as fast as $4^n / n^{1.5}$

$a_1, \dots, a_n$ → keys ordered from the smallest to the largest

$p_1, \dots, p_n$ → probabilities of searching

$C[i,j]$ → smallest average number of comparisons made in a successful search in a binary tree T_i^j made up of keys $a_i, \dots, a_j$, $1 \le i \le j \le n$

$n = 4$

$C(n) = \binom{8}{4}\frac{1}{5}$

$= \frac{8 \cdot 7 \cdot 6 \cdot 5}{1 \cdot 2 \cdot 3 \cdot 4}\left(\frac{1}{5}\right)$

$= \frac{70}{5}$

$= 14$

dynamic programming approach

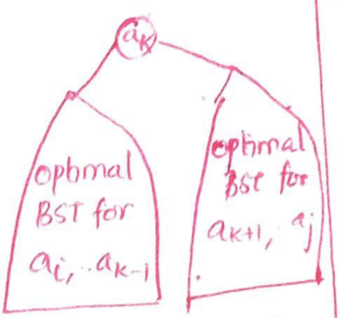
- find values of $c[i,j]$ for all smaller instances of the problem.

Recurrence Relation:

- consider all possible ways to choose a root a_K among the keys $a_i, \ldots a_j$.
  - * the root contains key a_K.
  - * the left subtree T_i^{k-1} contains keys $a_i, \ldots, a_{k-1}$ optimally arranged.
  - * the right subtree T_{k+1}^{j} contains keys $a_{k+1}, \ldots a_j$ optimally arranged

Structure: - Count tree levels starting with 1,

BST with root a_K



- two optimal binary search subtrees T_i^{k-1} & T_{k+1}^{j}

$$c[i,j] = \min_{i \le k \le j} \left\{ p_K \cdot 1 + \sum_{s=i}^{k-1} p_s \cdot (\text{levels of } a_s \text{ in } T_i^{k-1} + 1) + \sum_{s=k+1}^{j} p_s \, (\text{levels of } a_s \text{ in } T_{k+1}^{j} + 1) \right\}$$

$$= \min_{i \le k \le j} \left\{ p_K + \sum_{s=i}^{k-1} p_s \cdot \text{level of } a_s \text{ in } T_i^{k-1} + \sum_{s=i}^{k-1} p_s + \sum_{s=k+1}^{j} p_s \cdot \text{level of } a_s \text{ in } T_{k+1}^{j} + \sum_{s=k+1}^{j} p_s \right\}$$

$$= \min_{i \le k \le j} \left\{ \sum_{s=i}^{k-1} p_s \cdot \text{level of } a_s \text{ in } T_i^{k-1} + \sum_{s=k+1}^{j} p_s \cdot \text{level of } a_s \text{ in } T_{k+1}^{j} + \sum_{s=i}^{j} p_s \right\}$$

$$= \min_{i \le k \le j} \left\{ c[i, k-1] + c[k+1, j] \right\} + \sum_{s=i}^{j} p_s$$

$$\boxed{c[i,j] = \min_{i \le k \le j} \{ c[i,k-1] + c[k+1,j] \} + \sum_{s=i}^{j} p_s \quad \text{for } 1 \le i \le j \le n}$$

$c[i, i-1] = 0$ for $1 \le i \le n+1$ → number of comparisons in the empty tree.

$c[i,i] = p_i$ for $1 \le i \le n$ → one-node binary tree containing a_i

Two-dimensional table: for computing $c[i,j]$.

Cost Table: - values
- → in row i and the columns to the left of column j
- & in column j and the rows below row i.

arrows → point to the pairs of entries whose sums are computed in order to find the smallest one to be recorded as the value of $c[i,j]$.

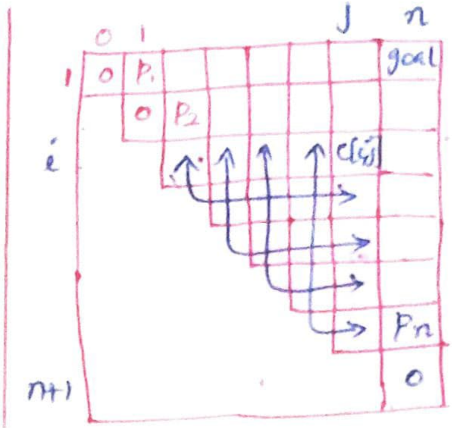
* filling the table along its diagonals, starting with all zeros on the main diagonal and given probabilities p_i, $1 \le i \le n$, right above it and moving toward the upper right corner.

* $C[1,n]$ – the average number of comparisons for successful searches in the optimal binary tree.

Root Table

- record the value of k for which the minimum is achieved
- starting with entries $R[i,i] = i$ for $1 \le i \le n$

* When the table is filled, its entries indicate indices of the roots of the optimal subtrees.



3 Example:

Applying the algorithm

key	A	B	C	D
probability	0.1	0.2	0.4	0.3

~~Initial ta~~

Solution:

Formula

$$C[i, i-1] = 0$$

$$C[i,i] = P_i$$

$$C[i,j] = \min_{i \le k \le j} \{ C[i,k-1] + C[k+1,j] \} + \sum_{s=i}^{j} P_s \quad \text{for } 1 \le i \le j \le n$$

Step 1:

$C[1,0] = C[2,1] = C[3,2] = C[4,3] = C[5,4] = 0$

$C[1,1] = P_1 = 0.1$

$C[2,2] = P_2 = 0.2$

$C[3,3] = P_3 = 0.4$

$C[4,4] = P_4 = 0.3$.

Initial table:

main table

	0	1	2	3	4
1	0	0.1			
2		0	0.2		
3			0	0.4	
4				0	0.3
5					0

root table

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

Step 2:

Compute remaining cells.

Compute $c[1,2]$: $i=1, j=2$

$$c[1,2] = \min \begin{cases} k=1: & c[1,0]+c[2,2]+\sum_{s=1}^{2} p_s \\ k=2: & c[1,1]+c[3,2]+\sum_{s=1}^{2} p_s \end{cases}$$

$$= \min \begin{cases} k=1: & 0+0.2+0.1+0.2 = 0.5 \\ k=2: & 0.1+0+0.1+0.2 = 0.4 \end{cases}$$

$\boxed{c[1,2] = 0.4}$, $\boxed{k=2}$ ie) $\boxed{R[1,2]=2}$

Compute $c[2,3]$:

$$c[2,3] = \min \begin{cases} k=2: & c[2,1]+c[3,3]+\sum_{s=2}^{3} p_s \\ k=3: & c[2,2]+c[4,3]+\sum_{s=2}^{3} p_s \end{cases}$$

$$= \min \begin{cases} k=2: & 0+0.4+0.2+0.4 = 1.0 \\ k=3: & 0.2+0+0.2+0.4 = 0.8 \end{cases}$$

$\boxed{c[2,3] = 0.8}$ $\boxed{k=3}$ ie) $\boxed{R[2,3]=3}$

Compute $c[3,4]$

$$c[3,4] = \min \begin{cases} k=3: & c[3,2]+c[4,4]+\sum_{s=3}^{4} p_s \\ k=4: & c[3,3]+c[5,4]+\sum_{s=3}^{4} p_s \end{cases}$$

$$= \min \begin{cases} k=3: & 0+0.3+0.4+0.3 = 1.0 \\ k=4: & 0.4+0+0.4+0.3 = 1.1 \end{cases}$$

$\boxed{c[3,4] = 1.0}$ $\boxed{k=3}$ $\boxed{R[3,4]=3}$

Compute $c[1,3]$:

$$c[1,3] = \min \begin{cases} k=1: & c[1,0]+c[2,3]+\sum_{s=1}^{3} p_s \\ k=2: & c[1,1]+c[3,3]+\sum_{s=1}^{3} p_s \\ k=3: & c[1,2]+c[4,3]+\sum_{s=1}^{3} p_s \end{cases}$$

$$= \min \begin{cases} k=1: & 0+0.8+0.1+0.2+0.4 = 1.5 \\ k=2: & 0.1+0.4+0.1+0.2+0.4 = 1.2 \\ k=3: & 0.4+0+0.1+0.2+0.4 = 1.1 \end{cases}$$

$\boxed{c[1,3] = 1.1}$ $\boxed{k=3}$ $\boxed{R[1,3]=3}$

Similarly Compute c[2,4] : $\boxed{c[2,4] = 1.4}$ $\boxed{k=3}$ $\boxed{R[2,4]=3}$

Compute c[1,4] :

$$c[1,4] = \min \begin{cases} k=1; & c[1,0]+c[2,4]+\sum_{s=1}^{4} p_s \\ k=2: & c[1,1]+c[3,4]+\sum_{s=1}^{4} p_s \\ k=3: & c[1,2]+c[4,4]+\sum_{s=1}^{4} p_s \\ k=4: & c[1,3]+c[5,4]+\sum_{s=1}^{4} p_s \end{cases}$$

$$= \min \begin{cases} k=1: & 0+1.4+0.1+0.2+0.4+0.3 = 2.4 \\ k=2: & 0.1+1.0+0.1+0.2+0.4+0.3 = 2.1 \\ k=3: & 0.4+0.3+0.1+0.2+0.4+0.3 = 1.7 \\ k=4: & 1.1+0+0.1+0.2+0.4+0.3 = 2.1 \end{cases}$$

$\boxed{c[1,4] = 1.7}$ $\boxed{k=3}$ $\boxed{R[1,4]=3}$

Step 3:

→ fill in the table

Final tables

Main table

	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2		0	0.2	0.8	1.4
3			0	0.4	1.0
4				0	0.3
5					0

Root table

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

* The average number of key comparisons in the optimal tree is equal to 1.7

* $\boxed{R[1,4] = 3}$ — the root of the optimal tree contains the third key, C

- left subtree is made up of keys A and B
- right subtree contains the key D

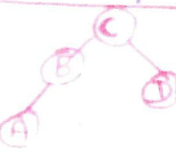
Construction of tree

R[1,2] = 2, the root of the optimal tree containing A & B is B

A - left child , R[1,1] = 1

R[4,4] = 4, the root in the right subtree.

Optimal BST



Pseudocode:

ALGORITHM optimalBST (P[1..n])

// Finds an optimal binary search tree by dynamic programming.
// Input: An array P[1..n] of search probabilities for a sorted list of n keys
// Output: Average number of comparisons in successful searches in the
// optimal BST and table R of subtrees' roots in the optimal BST

```
for i <- 1 to n do
    c[i,i-1] <- 0
    c[i,i] <- P[i]
    R[i,i] <- i
c[n+1,n] <- 0
for d <- 1 to n-1 do    // diagonal count
    for i <- 1 to n-d do
        j <- i+d
        minval <- ∞
        for k <- i to j do
            if c[i,k-1] + c[k+1,j] < minval
                minval <- c[i,k-1]+c[k+1,j]; kmin <- k
        R[i,j] <- k
        sum <- P[i];
        for s <- i+1 to j do sum <- sum + P[s]
            c[i,j] <- minval + sum
return c[1,n], R
```

Analysis:

Time efficiency = $\boxed{O(n^3)}$

× ———————— ×

Method:
Steps:

1) Find the cost c[1,n] using cost table and find the root R[1,n] using root table.
2) Construction of cost table
   a) c[i, i-1] = 0 b) c[i,i] = p_i c) c[i,j] - using formula.
3) Construction of root table
   a) R[i,i] = i b) R[1,n] from step 2.
4) Draw the optimal Binary Search Tree.

3.1.7 KNAPSACK PROBLEM AND MEMORY FUNCTIONS

Defn. Knapsack problem:

2 marks

* Given n items of known weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.
  - the weights and the knapsack capacity are positive integers
  - the item values do not have to be integers.

Method:

Dynamic Programming algorithm:

- Derive a recurrence relation that expresses a solution to an instance of the knapsack problem in terms of solutions to its smaller subinstances

→ instances defined by 'i' items, $1 \le i \le n$

→ weights $w_1, \ldots, w_i$

→ values $v_1, \ldots, v_i$

→ knapsack capacity j, $1 \le j \le W$

→ $V[i,j]$ – value of an optimal solution to the instance.
ie) the value of the most valuable subset of the first i items that fit into the knapsack of capacity j.

→ Divide all subsets

- do not include i^{th} item
- include i^{th} item

i) Among the subsets that do not include all i^{th} item, the value of an optimal subset is, $V[i-1,j]$

ii) Among the subsets that do include the i^{th} item, an optimal subset is made up of the item and an optimal subset of the first $i-1$ item that fit into the knapsack of capacity $j-w_i$
The value of an optimal subset is $v_i + V[i-1, j-w_i]$

* The value of an optimal solution among all feasible subsets of the first i items is the maximum of two values.

Formula:

Recurrence:

$$V[i,j] = \begin{cases} \max\{V[i-1,j],\ v_i + V[i-1, j-w_i]\} & \text{if } j - w_i \ge 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

Initial Conditions:

$V[0, j] = 0 \quad$ for $j \geq 0$
$V[i, 0] = 0 \quad$ for $i \geq 0$

goal → to find $V[n, W]$, the maximal value of a subset of the n given items that fit into the knapsack of capacity W and an optimal subset.

Table: for solving the knapsack problem:

		0	$j-w_i$	j	W
	0	0	0	0	0
	$i-1$	0	$V[i-1, j-w_i]$	$V[i-1, j]$	
w_i, v_i	i	0		$V[i, j]$	
	n	0			goal

* For $i, j > 0$, to compute the entry in the ith row and the jth column
* $V[i, j]$ – compute the maximum of the entry in the previous row and the same column and the sum of v_i and the entry in the previous row and w_i columns to the left.

- The table can be filled either row by row or column by column

Example: Consider the instance given by the following data:

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity $W = 5$

Solution:

$V[0, j] = 0$
$V[i, 0] = 0$

$$V[i,j] = \begin{cases} \max\{V[i-1, j], v_i + V[i-1, j-w_i]\} & \text{if } j - w_i \geq 0 \\ V[i-1, j] & \text{if } j - w_i < 0 \end{cases}$$

Step 1:

dynamic programming table:

initial table:

capacity j

	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1=2, v_1=12$	1	0					
$w_2=1, v_2=10$	2	0					
$w_3=3, v_3=20$	3	0					
$w_4=2, v_4=15$	4	0					

Step 2: Filling the remaining cells.

→ $V[1,1]$:

$i=1,\ j=1,\ w_1=2,\ v_1=12$

$j-w_1 = 1-2 = -1 < 0$

$V[1,1] = V[0,1] = 0$

→ $V[1,2]$: $j-w_i = 2-2 = 0$

$V[1,2] = \max\{V[0,2], v_1 + V[0, 2-2]\}$

$= \max\{0, 12+V[0,0]\} = \max\{0, 12+0\} = 12$

$\boxed{V[1,2]=12}$

→ $V[1,3]$ $j-w_i = 3-2 = 1$

$V[1,3] = \max\{V[1-1,3], v_1 + V[1-1, 3-2]\}$

$= \max\{V[0,3], 12+V[0,1]\} = \max\{0, 12+0\} = 12$

$\boxed{V[1,3]=12}$

→ $V[1,4]$ $j-w_i = 4-2 = 2$

$V[1,4] = \max\{V[1-1,4], v_1 + V[1-1, 4-2]\} = \max\{V[0,4], 12+V[0,2]\}$

$= \max\{0, 12+0\} = 12$

$\boxed{V[1,4]=12}$

→ $V[1,5]$ $j-w_i = 5-2 = 3$

$V[1,5] = \max\{V[1-1,5], v_1 + V[1-1, 5-2]\} = \max\{V[0,5], 12+0\}$

$= \max\{0, 12\}$

$\boxed{V[1,5]=12}$

Now, the table is

	i	capacity j 0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0					
$w_3 = 3, v_3 = 20$	3	0					
$w_4 = 2, v_4 = 15$	4	0					

→ $V[2,1]$: $j - w_i = 1 - w_2 = 1 - 1 = 0$

$V[2,1] = \max\{V[2-1,1], V_2 + V[2-1, 1-1]\}$
$= \max\{V[1,1], 10 + V[1,0]\} = \max\{0, 10+0\}$

$\boxed{V[2,1] = 10}$

* Similary find for $V[2,2], V[2,3], V[2,4], V[2,5]$

$V[2,2] = 12$
$V[2,3] = 22$
$V[2,4] = 22$
$V[2,5] = 22$

→ $V[3,1]$: $j - w_i = 1 - w_3 = 1-3 = -2$

$V[3,1] = V[i-1,j] = V[3-1,1] = V[2,1]$

$\boxed{V[3,1] = 10}$

→ $V[3,2]$: $j - w_i = 2 - 3 = -1$

$V[3,2] = V[3-1,2] = V[2,2]$

$\boxed{V[3,2] = 12}$

→ $V[3,3]$: $j - w_i = 3 - w_3 = 3-3 = 0$

$V[3,3] = \max\{V[3-1,3], V_3 + V[3-1, 3-3]\}$
$= \max\{V[2,3], 20 + V[2,0]\} = \max\{22, 20+0\}$

$\boxed{V[3,3] = 22}$

* Similarly find for $V[3,4]$ & $V[3,5]$

$V[3,4] = 30$
$V[3,5] = 32$

→ $V[4,1]$: $j - w_i = 1 - w_4 = 1 - 2 = -1$

$V[4,1] = V[i-1,j] = V[4-1,1] = V[3,1]$

$\boxed{V[4,1] = 10}$

→ $V[4,2]$: $j - w_i = 2 - w_4 = 2 - 2 = 0$

$V[4,2] = \max\{V[4-1,2], V_4 + V[4-1, 2-w_4]\}$

$= \max\{V[3,2], 15 + V[3, 2-2]\} = \max\{12, 15+V[3,0]\}$

$= \max\{12, 15+0\} = 15$

$\boxed{V[4,2] = 15}$

* Similarly find for $V[4,3]$, $V[4,4]$ & $V[4,5]$

$V[4,3] = 25$

$V[4,4] = 30$

→ $V[4,5]$: $j - w_i = 5 - w_4 = 5 - 2 = 3 > 0$

$V[4,5] = \max\{V[4-1,5], V_4 + V[4-1, 5-2]\}$

$= \max\{V[3,5], 15 + V[3,3]\} = \max\{32, 15+22\}$

$= \max\{32, 37\}$

$\boxed{V[4,5] = 37}$

Step 3: Final Table:

capacity j

	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, V_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, V_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3, V_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2, V_4 = 15$	4	0	10	15	25	30	$\boxed{37}$

Maximum value = \$37

Step 4:

* Find the composition of an optimal subset by tracing back the computations of the entry in the table.

i) $V[4,5] \neq V[3,5]$

- item 4 is included in 1 optimal solution along with an optimal subset for filling $5-2=3$ remaining units of the knapsack capacity.

ii) V[3,3] = V[2,3]

- item 3 is not a part of an optimal subset

iii) V[2,3] ≠ V[1,3]

- item 2 is a part of an optimal selection

iv) V[1,3-1] = V[1,2] - remaining composition

V[1,2] ≠ V[0,2]

- item 1 is the final part of the optimal solution

Solution is :

{item 1, item 2, item 4}

ie) {1, 2, 4} = $ 37

Analysis:

- Time efficiency
- Space efficiency

$\left.\right\}$ = $\boxed{\Theta(nW)}$

→ Time needed to find the composition of an optimal solution is $O(n+W)$

* MEMORY FUNCTIONS :

* dynamic programming - deals with problems whose solutions satisfy a recurrence relation with overlapping subproblems.

top-down approach → finding a solution to a recurrence leads to an algorithm that solves common subproblems more than once

→ very inefficient.

* classic dynamic programming :

bottom-up approach → it fills a table with solutions to all smaller subproblems but each of them is solved only once.

* Solutions to some of the smaller subproblems are not necessary for getting a solution to the problem.

→ memory functions:

→ The goal is to get a method that solves only subproblems that are necessary and does it only once.

- it is based on using memory functions.

Steps :-

Virtual initialization * Initially, the table's entries are initialized with a special "null" symbol to indicate that they have not yet been calculated.

* Whenever, a new value needs to be calculated, the method checks the corresponding entry in the table first.
    - if this entry is not "null", it is simply retrieved from the table
    - otherwise, it is computed by the recursive call whose result is then recorded in the table.

Example: Knapsack problem:

- initializing the table
- The recursive function calls with $i = n$ and $j = W$

3.5.6 ALGORITHM MFKnapsack(i, j)

```
// Implements the memory function method for the knapsack problem.
// Input : A nonnegative integer i indicating the number of the first items
//         being considered and a nonnegative integer j indicating
//         the knapsack capacity.
// Output: The value of an optimal feasible subset of the first i items
// Note : global variables input arrays Weights[1..n], values[1..n] and
//         table V [0..n, 0..W]

if V[i, j] < 0
    if j < Weights[i]
        value ← MFKnapsack (i-1, j)
    else
        value ← max(MFKnapsack(i-1, j),
                    values[i] + MFKnapSack (i-1, j-Weights[i]))
    V[i, j] ← value
return V[i, j]
```

- Apply the memory function method to the instance.

capacity j

	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	–	12	12
$w_2 = 1, v_2 = 10$	2	0	–	12	22	–	22
$w_3 = 3, v_3 = 20$	3	0	–	–	22	–	32
$w_4 = 2, v_4 = 15$	4	0	–	–	–	–	37

* Only 10 out of 20 values have been computed.
  eg $V[1,2]$ is retrieved, rather than be recomputed.

- memory function method may be less space - efficient

× ——————— ×

3.2 GREEDY TECHNIQUE

→ Change-making problem is called greedy.

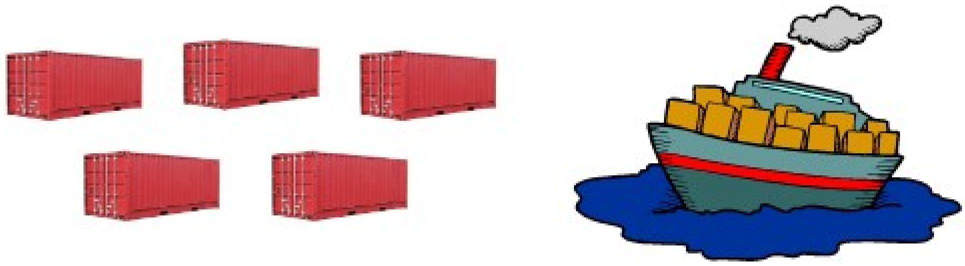
defn :-

* The greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached.

→ On each step, the choice made must be

* feasible, i.e. it has to satisfy the problem's constraints
* locally optimal, i.e, it has to be the best local choice among all feasible choices available on that step.
* irrevocable, i.e., once made, it cannot be changed on subsequent steps of the algorithm.

3.2.1 Container Loading



- A large ship is to be loaded with containers of cargos.
- Different containers will have different weights.
  - Let *wi* be the weight of the *i*th container,
  - $1 \leq i \leq n$, and the capacity of the ship is *C*
- To find out how could the ship can be loaded with the maximum number of containers.

- Greedy Technique:
  - The ship may be loaded in stages; one container per stage.
  - At each stage ***select the one with least weight***.
  - Then the one with the next smallest weight, and so on until either all containers have been loaded or there is not enough capacity for the next one.
  - This results in loading maximum number of containers.
- **Example :**

  Suppose that $n = 8$, $[w1, \dots, w8] = [100, 200, 50, 90, 150, 50, 20, 80]$, and $c = 400$.

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	c
100	200	50	90	150	50	20	80	400
1	1	1	0	0	1	0	0	400

- Only 4 containers are loaded for the capacity 400
- Not the optimal solution

- Applying Greedy technique
  - The containers are added in the increasing weight order
  - 6 containers (greater than 4) are loaded with capacity 390 - Optimal Solution

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	c
20	50	50	80	90	100	150	200	400
1	1	1	1	1	1	0	0	390

- The available capacity is now(400-390= 10 units), which is inadequate for any of the remaining containers.
- Greedy solution we have $[x1, , , , , x8] = [1, 0, 1, 1, 0, 1, 1, 1]$ and $\Sigma xi = 6$.

Algorithm

```
void containerLoading(container* c, int capacity,
                      int numberOfContainers, int* x)
{// Greedy algorithm for container loading.
 // Set x[i] = 1 iff container i, i >= 1 is loaded.
   // sort into increasing order of weight
   heapSort(c, numberOfContainers);

   int n = numberOfContainers;

   // initialize x
   for (int i = 1; i <= n; i++)
      x[i] = 0;

   // select containers in order of weight
   for (int i = 1; i <= n && c[i].weight <= capacity; i++)
   {// enough capacity for container c[i].id
      x[c[i].id] = 1;
      capacity -= c[i].weight;  // remaining capacity
   }
}
```

Analysis:

- Time complexity = O(n log n)

3.2.2 PRIM'S ALGORITHM

* Given n points, connect them in the cheapest possible way so that there will be a path between every pair of points.

points - vertices.
connections - edges
cost - weight.

Definition:

spanning tree:

- A spanning tree of a connected graph is its connected acyclic subgraph (i.e) a tree) that contains all the vertices of the graph.

Minimum spanning tree:

- A minimum spanning tree of a weighted connected graph is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the weights on all its edges.

Minimum spanning tree problem:

- finding a minimum spanning tree for a given weighted connected graph.

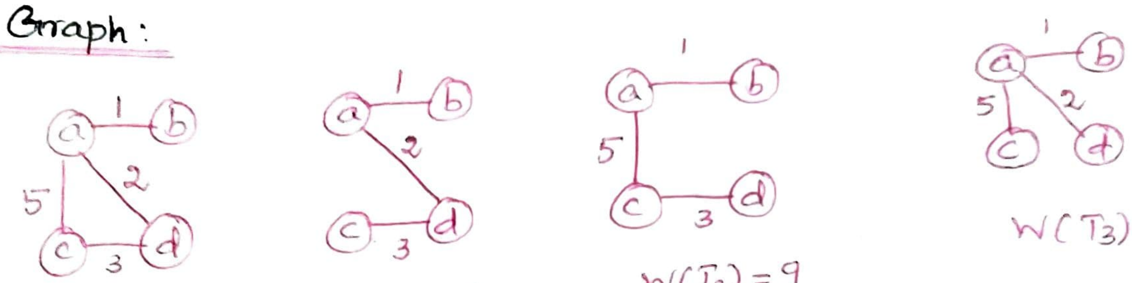
DisAdvantages in Exhaustive search approach.

i) the number of spanning trees grows exponentially with the graph size.

ii) generating all spanning trees for a given graph is not easy.

Graph and its spanning trees:

Graph:



$W(T_1) = 6$ Minimum spanning tree

$W(T_2) = 9$

$W(T_3) = 8$

Prim's Algorithm:

* Constructs a minimum spanning tree through a sequence of expanding subtrees.

Steps:-

- Initial subtree → a sequence consists of a single vertex selected from the set V of the graph's vertices.
- On each iteration, expand the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree.
- The algorithm stops after all the graph's vertices have been included in the tree being constructed.

→ Number of iterations = n − 1. Since the algorithm expands a tree by exactly one vertex on each of its iterations.

- The tree generated is obtained as the set of edges used for the tree expansions.

Pseudocode:

ALGORITHM Prim(G)

// Prim's algorithm for constructing a minimum spanning tree.

// Input : A weighted connected graph $G = (V, E)$

// Output : E_T, the set of edges composing a MST of G.

$V_T \leftarrow \{v_0\}$
$E_T \leftarrow \phi$
for $i \leftarrow 1$ to $|V|-1$ do
find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u)
such that v is in V_T and u is in $V - V_T$
$V_T \leftarrow V_T \cup \{u^*\}$
$E_T \leftarrow E_T \cup \{e^*\}$
return E_T

contd...

two labels:

- the name of the nearest tree vertex
- length (weight) of the corresponding edge.

* Vertices that are not adjacent to any of the tree vertices can be given the ∞ label indicating their "infinite" distance to the tree vertices and
* a null label for the name of the nearest tree vertex.

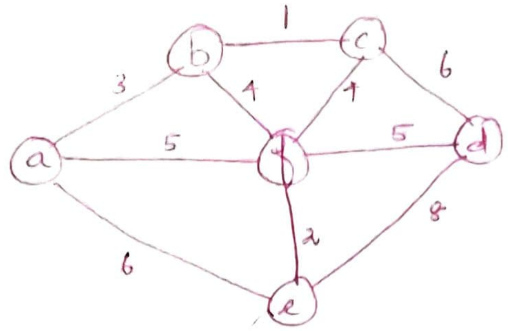
vertices (two sets) — fringe, unseen

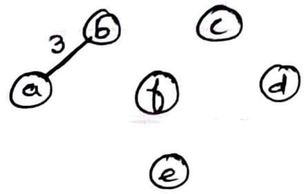
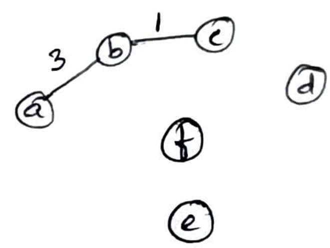
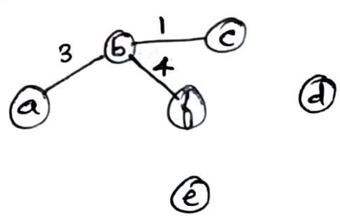
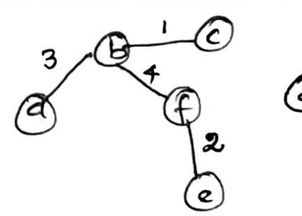
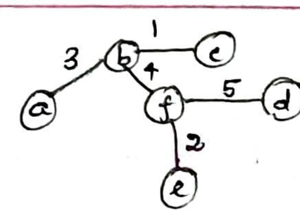
fringe - contains only the vertices that are not in the tree but are adjacent to at least one tree vertex
- the candidates from which the next tree vertex is selected.

unseen - vertices are all the other vertices of the graph

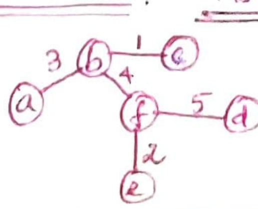
* Finding the next vertex to be added to the current tree $T = (V_T, E_T)$ becomes a task of finding a vertex with the smallest distance label in the set $V - V_T$.
* After identifying a vertex u^* to be added to the tree, perform two operations:
  i) Move u^* from the set $V - V_T$ to the set of tree vertices V_T
  ii) For each remaining vertex u in $V - V_T$ that is connected to u^* by a shorter edge than the u's current distance label, update its labels by u^* and the weight of the edge between u^* and u respectively.

3.6.3 Application of Prim's algorithm:



	Tree vertices	Remaining vertices	Illustration
Step 1:	a(–, –)	b(a, 3) c(–, ∞) d(–, ∞) e(a, 6) f(a, 5)	
Step 2:	b(a, 3)	c(b, 1) d(–, ∞) e(a, 6) f(b, 4)	
Step 3:	c(b, 1)	d(c, 6) e(a, 6) f(b, 4)	
Step 4:	f(b, 4)	d(f, 5) e(f, 2)	
Step 5:	e(f, 2)	d(f, 5)	
Step 6:	d(f, 5)		

Optimal Solution: MST



$w(T) = 3 + 1 + 4 + 2 + 5 = 15$

* Proof of correctness of Prim's Algorithm:

Theorem:

Prim's algorithm yields a minimum spanning tree always.

Proof:

- Let $G = (V, E)$ be a weighted connected graph. Let T be the edge set that is grown in Prim's algorithm.

* The proof is by mathematical induction on the number of edges in T and using the MST Lemma.

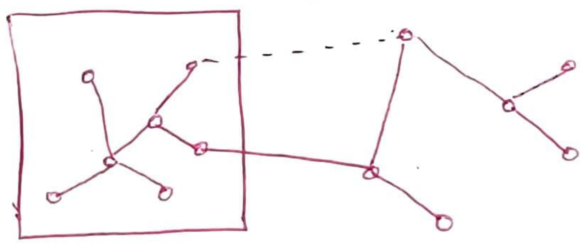
Basis:

- T_0 consists of a single vertex and hence must be a part of any minimum spanning tree.

Induction step:

- Let $e_i = (v, u)$ - minimum weight edge from a vertex in T_{i-1} to a vertex not in T_{i-1} used by Prim's algorithm to expand T_{i-1} to T_i.
- e_i cannot belong to any MST including T. if we add e_i to T, a cycle must be formed

Correctness proof of Prim's algorithm.



* In addition to edge $e_i = (v, u)$, this cycle must contain another edge (v', u') connecting a vertex $v' \in T_{i-1}$ to a vertex u' which is not in T_{i-1}.
* if deletion of edge (v', u'), obtain another spanning tree of the entire graph.

- Hence, this spanning tree is a minimum spanning tree

* How efficient is Prim's algorithm:?

- depends on the data structures chosen for the graph itself and for the priority queue of the set $V - V_T$ whose vertex priorities are the distances to the nearest tree vertices.

weight matrix + priority queue →

* the running time $= \Theta(|V|^2)$

* On each of the $|V|-1$ iterations, the array implementing the priority queue is traversed to find and delete the minimum and then to update.

priority queue with a min heap →

* A min heap is a complete binary tree in which every element is less than or equal to its children.

* Deletion + insertion — $O(\log n)$ operations.

adjacency Linked List + Priority queue (min heap)

* Running time $= O(|E| \log |V|)$

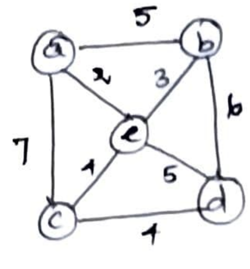
Analysis:

Running time of Prim's algorithm

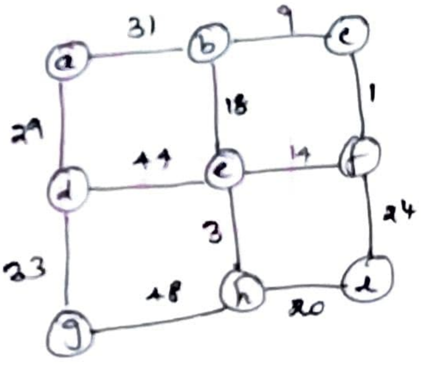
$$(|V|-1+|E|)\,O(\log |V|) = \boxed{O(|E| \log |V|)}$$

in a connected graph; $|V|-1 \le |E|$

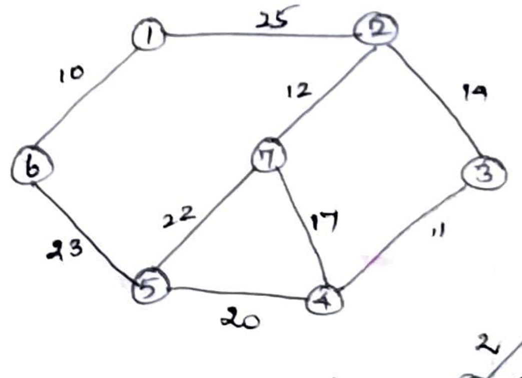
Ex: ①



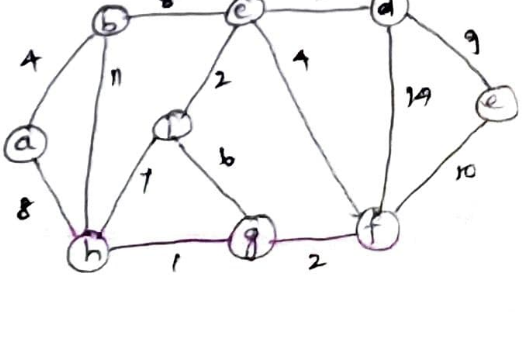
2)



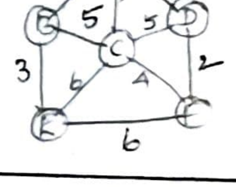
3)



4)



5)



× ——————————— ×

3.2.3 KRUSKAL'S ALGORITHM

Info

2m.

* Kruskal's algorithm looks at a minimum spanning tree for a weighted connected graph $G = (V, E)$ as an acyclic subgraph with $|V| - 1$ edges for which the sum of the edge weights is the smallest

→ The algorithm constructs a minimum spanning tree as an expanding sequence of subgraphs, which are always acyclic but are not necessarily connected on the intermediate stages of the algorithm

Steps:

i) The algorithm begins by sorting the graph's edges in nondecreasing order of their weights

ii) Then, starting with the empty subgraph

iii) scans the sorted list adding the next edge on the list to the current subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise

Pseudocode

ALGORITHM Kruskal(G)

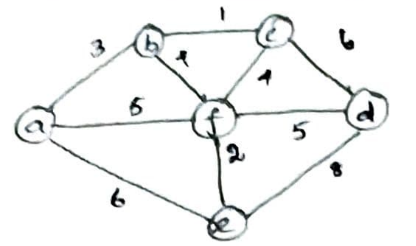
// Kruskal's algorithm for constructing a MST

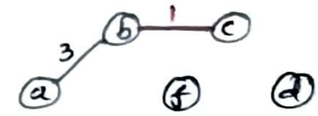
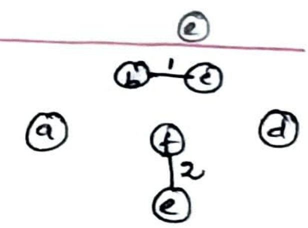
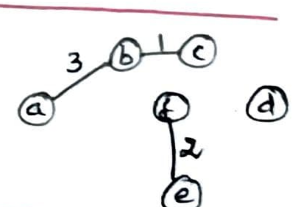
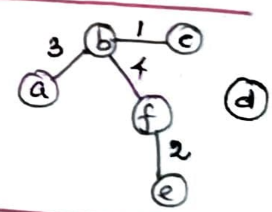
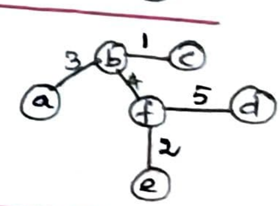
// Input: A weighted connected graph $G = (V, E)$

// Output: E_T, the set of edges composing a MST of G

```
Sort E in nondecreasing order of the edge weights w(e_i1) ≤ ... w(e_i|E|)
  E_T ← Φ ; ecounter ← 0
  k ← 0
  while ecounter < |V| - 1
      k ← k+1
      if E_T ∪ {e_ik} is acyclic
          E_T ← E_T ∪ {e_ik};
              ecounter ← ecounter + 1
  return E_T
```

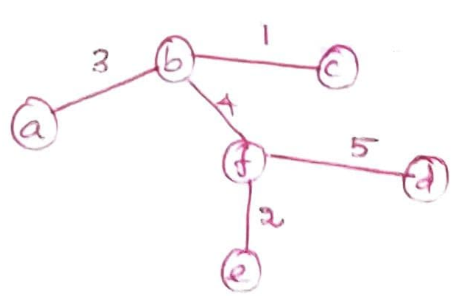
Application of kruskal's algorithm:



	Tree edges	Sorted list of edges	Illustration
step 1		**[bc 1]** ef 2, ab 3, bf 4, cf 4, af 5, df 5, ae 6, cd 6, de 8	
step 2	bc 1	bc 1, **[ef 2]** ab 3, bf 4, cf 4, af 5, df 5, ae 6, cd 6, de 8	
step 3	ef 2	bc 1, ef 2, **[ab 3]** bf 4, cf 4, af 5, df 5, ae 6, cd 6, de 8	
step 4	ab 3	bc 1, ef 2, ab 3, **[bf 4]** cf 4, af 5, df 5, ae 6, cd 6, de 8	
step 5	bf 4	bc 1, ef 2, ab 3, bf 4, cf 4, af 5, **[df 5]** ae 6, cd 6, de 8	
step 6	df 5		

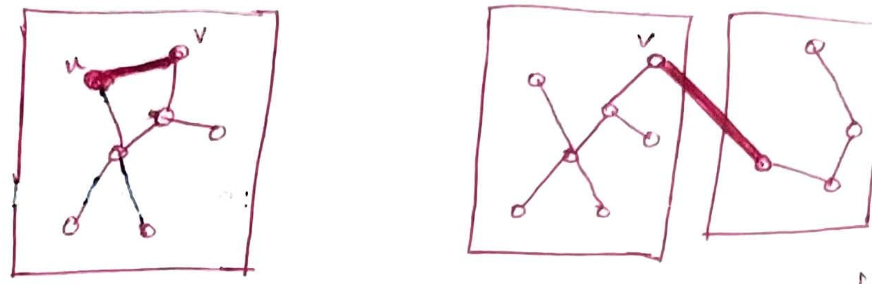
optimal solution:

MST



Correctness of kruskal's algorithm : → same as Prim's algorithm.

* Kruskal's algorithm has to check whether the addition of the next edge to the edges already selected would create a cycle.
- a new cycle is created if and only if the new edge connects two vertices already connected by a path.



New edge connecting two vertices may create a cycle

New edge connecting two vertices may not create a cycle.

initial : → single vertex

final → single tree, which is a minimum spanning tree

* On each iteration, the algorithm takes the next edge (u,v) from the sorted list of the graph's edges, finds the trees containing the vertices u and v.

union-find algorithm – check whether two vertices belong to the same tree.

* Analysis:

* Time efficiency of kruskal's algorithm $\boxed{O(|E| \log |E|)}$

Disjoint subsets and Union-find Algorithms:

* Requires a dynamic partition of some n-element set S into a collection of disjoint subsets $S_1, S_2, \ldots S_k$.

* After being initialized as a collection of n one-element subsets, each containing a different element of S, the collection is subjected to a sequence of intermixed union and find operations.

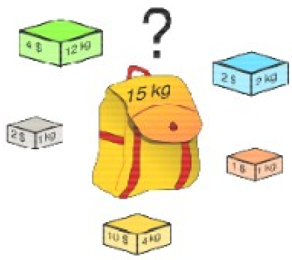
Operations:

i) makeset(x) – creates a one-element set {x}

ii) find(x) – returns a subset containing x

iii) union(x, y) – constructs the union of the disjoint subsets S_x and S_y containing x and y respectively and adds it to the collection to replace S_x and S_y, which are deleted from it.

3.2.4 0/1 Knapsack Problem



- Given n objects and a knapsack or bag. Object i has a weight w_i and the knapsack has a capacity m. Object i is placed into the knapsack, then a profit is earned.
- The objective is to obtain a filling of the knapsack that maximizes the total profit earned.
- Since the knapsack capacity is m, it is required that the total weight of all chosen objects to beat most m.
- Formally, the problem can be stated as

$$\max \sum_{1}^{n} p_i x_i$$

$$\text{subject to} \sum_{1}^{n} w_i x_i \leq m$$

$$\text{and} \quad x_i = 0 \text{ or } 1, \;\; 1 \leq i \leq n$$

- The profits and weights are positive numbers.
- A feasible solution is any set (xi,.x..n,) satisfying the conditions.
- An optimal solution is a feasible solution for which the objective function is maximized.

Example:

- Use the following instances of the knapsack problem, find the subset for maximizing the profit.

Knapsack capacity = 8

Items	Weight	Profit
1	1	15
2	5	10
3	3	9
4	4	5

Solution:

- Step 1:

Find Profit/Weight ratio

Items	Weight	Profit	Profit/Weight
1	1	15	15
2	5	10	2
3	3	9	3
4	4	5	1.25

- Step 2:

  Arrange in the descending order

Items	Weight	Profit	Profit/Weight
1	1	15	15
3	3	9	3
2	5	10	2
4	4	5	1.25

➢ Select the item which has the maximum profit/weight ratio and the weight must be less than or equal to the capacity of the knapsack

- Step 3:

  Use Greedy Technique, find the optimal solution

✔ Take Item 1

weight of the item 1 = 1 $\leq$ 8

Add item 1 into knapsack

{1}

8-1 = 7 ------> Remaining need to fill

✔ Next, Take Item 3

weight of the item 3 = 3 $\leq$ 7

Add item 3 into knapsack

{1,3}

7-3 = 4 ------> need to fill

✔ Next, Take Item 2

weight of the item 2 = 5 $\neq$ 4

Can't add item 2 into knapsack

{1,3}

7-3 = 4 ------> need to fill

✔ Next, Take Item 4

weight of the item 4 = 4 = 4

Add item 4 into knapsack

{1,3,4} i.e) {1,0,1,1} [1 – Included, 0 – Not included]

7-4 = 0 ------> knapsack is full

Answer:

Optimal Solution is {1,0,1,1}

Profit = 29

Analysis:

Time complexity – O(n)

3.2.4 Optimal Merge Pattern

- **Definition**
  - The problem is to merge a set of sorted files of different length into a single sorted file with minimum time.
  - This merge can be performed pair wise. Hence, this type of merging is called as **2- way merge patterns**.

- To merge a **p-record file** and a **q-record file** requires possibly **p + q** record moves, the better choice is merge the two smallest files together at each step.
- Two-way merge patterns can be represented by binary merge trees.

- Consider a set of **n** sorted files **{f1, f2, f3, …, fn}**.
- Initially, each element of this is considered as a single node binary tree.

Algorithm: TREE (n)
for i := 1 to n – 1 do
declare new node
node.leftchild := least (list)
node.rightchild := least (list)
node.weight := ((node.leftchild).weight) + ((node.rightchild).weight)
insert (list, node);
return least (list);
At the end of this algorithm, the weight of the root node represents the optimal cost.

Example

- Consider the given files, f1, f2, f3, f4 and f5 with 20, 30, 10, 5 and 30 number of elements respectively.

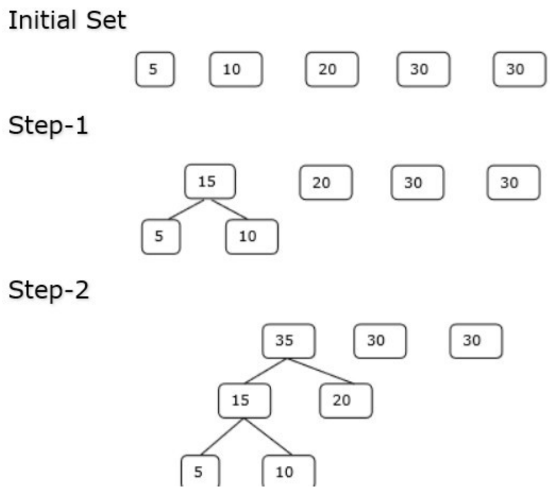
Solution 1:

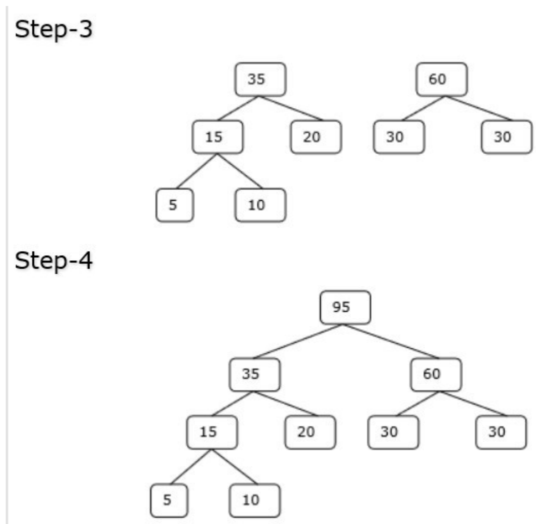
- Merge operations are performed according to the provided sequence, then
  M1 = merge f1 and f2 => 20 + 30 = 50
  M2 = merge M1 and f3 => 50 + 10 = 60
  M3 = merge M2 and f4 => 60 + 5 = 65
  M4 = merge M3 and f5 => 65 + 30 = 95
- The total number of operations is 50 + 60 + 65 + 95 = 270

Solution 2:

- Sorting the numbers according to their size in an ascending order
- Sequence - f4, f3, f1, f2, f5
- Merge operations can be performed on this sequence
  M1 = merge f4 and f3 => 5 + 10 = 15
  M2 = merge M1 and f1 => 15 + 20 = 35
  M3 = merge M2 and f2 => 35 + 30 = 65
  M4 = merge M3 and f5 => 65 + 30 = 95
- The total number of operations is 15 + 35 + 65 + 95 = 210

Solution 3:





- The solution takes 15 + 35 + 60 + 95 = 205 number of comparisons
- This is Optimal Solution

Analysis:

- Time complexity = O(n log n)

3.2.6 HUFFMAN TREE

→ Constructed for encoding a given text of n characters.

Codeword:

* Encoding a text that comprises symbols from n-symbol alphabet by assigning to each of the text's symbols some sequence of bits called the codeword.

Types:

→ ① Fixed-length encoding:
  - assigns to each symbol a bit string of the same length m ($m \geq \log_2 n$)
  - frequent letters : e (·)
    a (· –)
  - infrequent letters q (– – · –)
    z (– – · ·)

② → Variable-length encoding:
  - assigns codewords of different lengths to different symbols

Problem → how many bits of an encoded text represent the first symbol.

Avoid: Prefix-free codes (Prefix codes)
  → no codeword can simply scan a bit string

* Construct a tree that would assign shorter bit strings to high frequency symbols and longer ones to low-frequency symbols.

Huffman's algorithm:

Step 1: Initialize n one-node trees and label them with the symbols of the alphabet given.
  - Record the frequency of each symbol on its tree's root to indicate the tree's weight.

Step 2: Repeat the following operation until a single tree is obtained.
  - Find two trees with the smallest weight
  - Make them the left and right subtree of a new tree
  - Record the sum of their weights in the root of the new tree as its weights.

* Huffman tree – A tree constructed by Huffman algorithm
* Huffman code

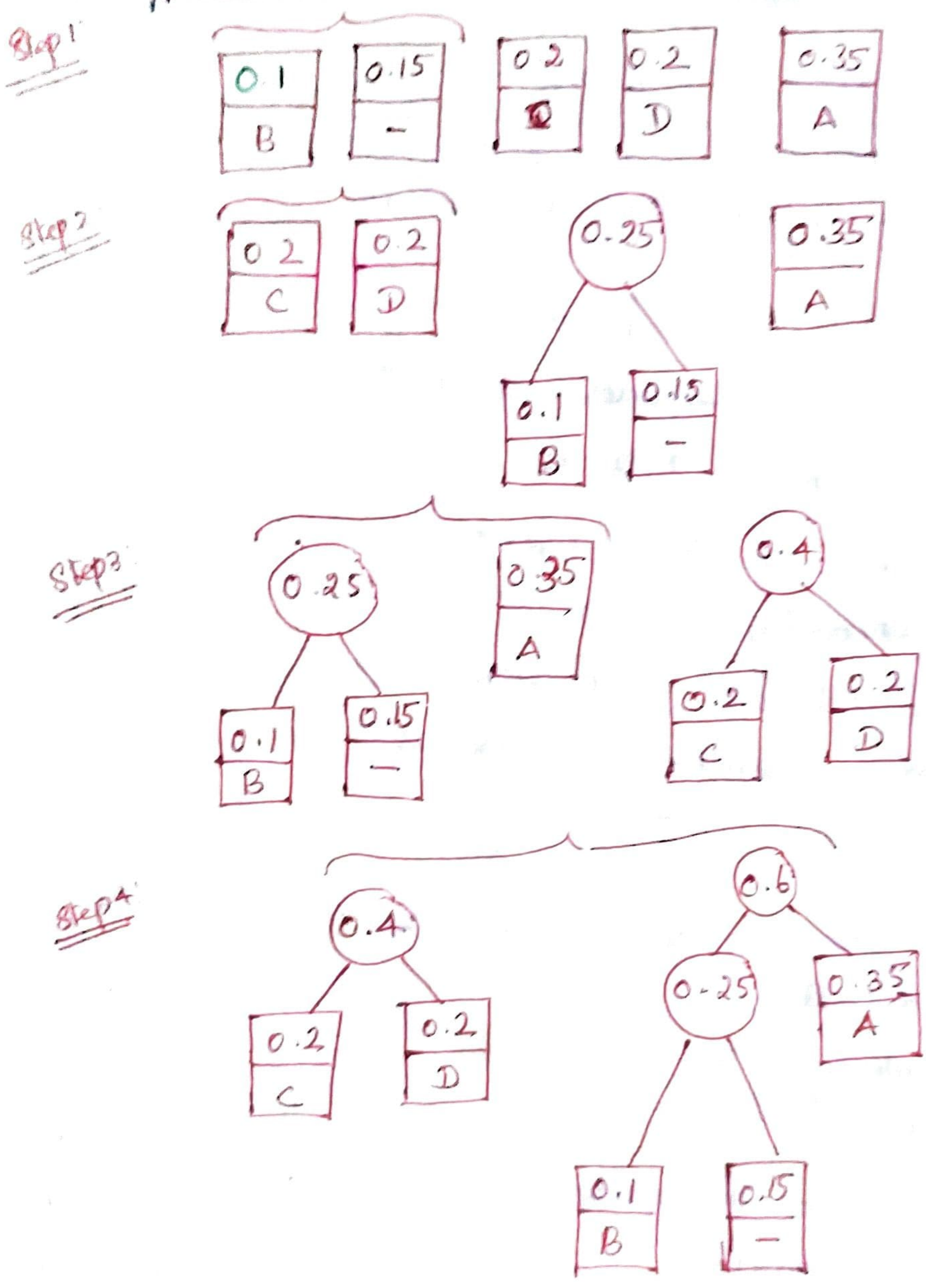
Example

* The five symbol alphabet { A, B, C, D, _ } with the following occurrence frequencies in a text made up of the symbols:

Symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15

Huffman tree Construction: i) Arrange the characters in ascending to their probabilities

ii) Combine nodes to form a subtree based on probabilities

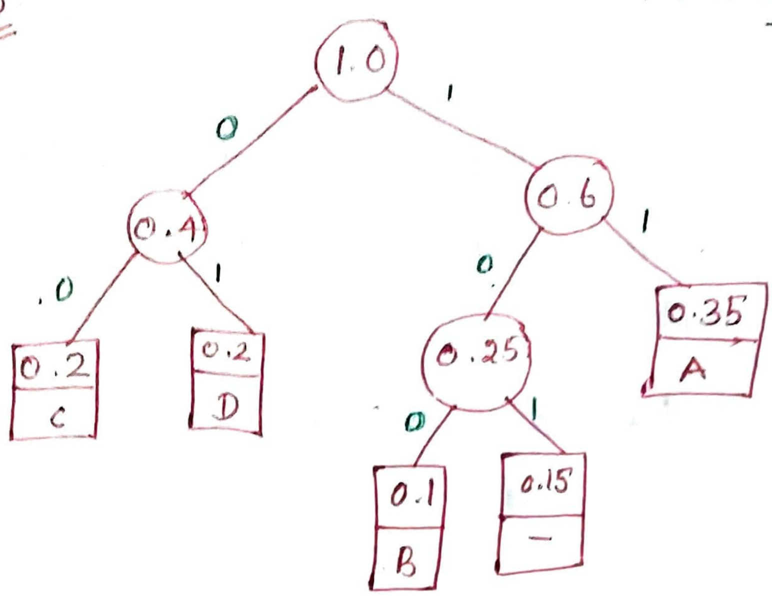


Step 5

Huffman tree

→ encode the tree

* left branch should be assigned with 0
* right branch should be assigned with 1.



The resulting codewords:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

* DAD is encoded as 011101
* 100110110111101 is decodes as BAD_AD

→ The average number of bits per symbol in the code

$\sum$ (length of codeword × Prob)

$2 \times 0.35 + 3 \times 0.1 + 2 \times 0.2 + 2 \times 0.2 + 3 \times 0.15 = 2.25$

→ fixed length encoding - to use at least 3 bits per symbol.

* Huffman code achieves compression ratio → a standard measure of a compression algorithm's effectiveness

$$\frac{3 - 2.25}{3} \times 100\,\% = 25\,\%$$

- Huffman encoding will use 25% less memory less than fixed-length encoding.

adv: - simplicity & versatility
- optimal encoding.

disadv: - to include the coding table into the encoded text to make its decoding possible

Dynamic Huffman encoding: - overcomes the disadvantage

- coding tree is updated each time a new symbol is read from the source text.

Lempel - Ziv algorithms : → assign codewords not to individual symbols but to strings of symbols

- achieves better and more robust compressions

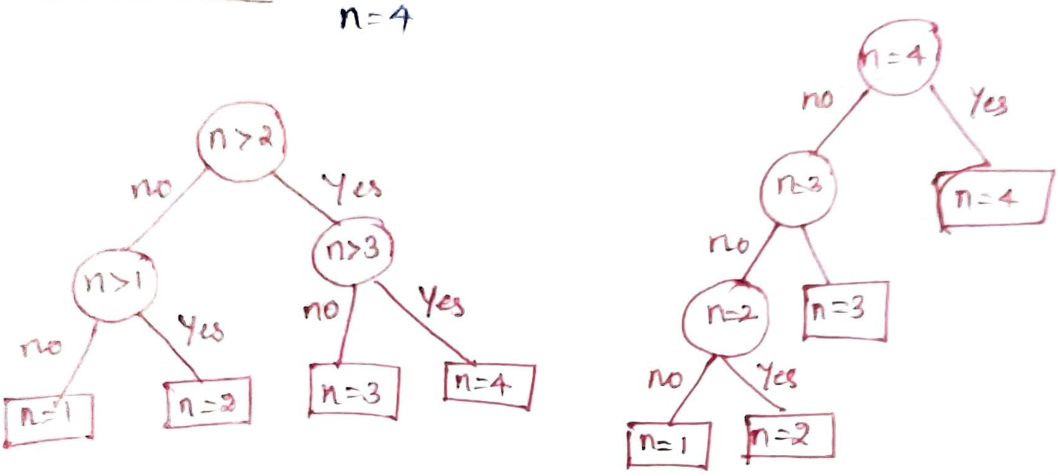
weighted path length :

sum $\sum_{i=1}^{n} l_i w_i$

l_i - length of the path from root to the i^{th} leaf

decision trees : Two decision trees

n = 4



* if number i is chosen with probability p_i, the sum

$\sum_{i=1}^{n} l_i p_i$, l_i - length of the path from the root to the i^{th} leaf.

ex: if $n=4$ and $p_1 = 0.1$, $p_2 = 0.2$, $p_3 = 0.3$ and $p_4 = 0.4$

- the minimum weighted path tree is the rightmost one in figure.

Applications :

1. Huffman encoding is used in file compression algorithms
2. Huffman's code is used in transmission of data in an encoded form.
3. used in game playing method

Ex: Construct a Huffman code for the following data

character	A	B	C	D	E
Probability	0.4	0.1	0.2	0.15	0.15

a) Encode the text ABACABAD

b) Decode the text whose encoding is 100010111001010

2) Construct the Huffman tree and give the Huffman encoding for the following:

Value	1	2	3	4	5	6
frequency.	5	7	10	15	20	45

3) Find the Huffman encoding for the following data:

Value	a	b	c	d	e
Frequency	20	15	5	15	45

x ———————— x

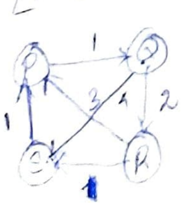
defn

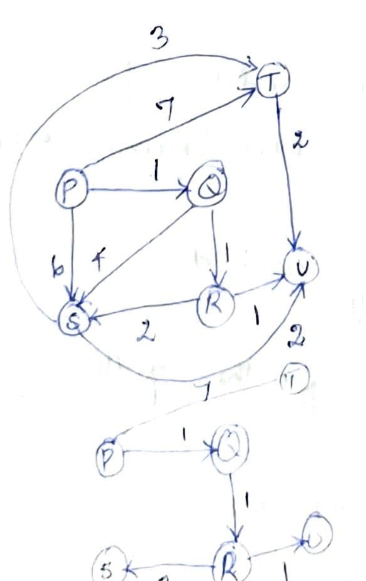
* A Huffman tree is a binary tree that minimizes the weighted path length from the root to the leaves containing a set of predefined weights.
* Huffman code → encoding scheme that assigns bit strings to characters based on their frequencies in a given text.

leaves → characters

edges - 0's and 1's

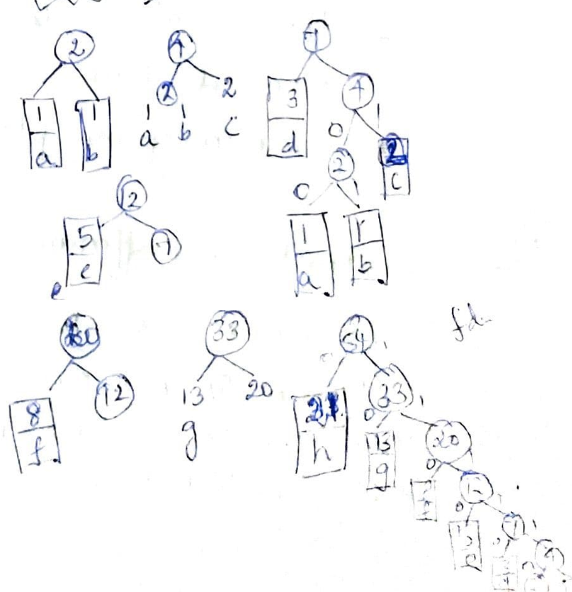
$$\begin{array}{c|cccc} & P & Q & R & S \\ \hline P & 0 & 1 & \infty & \infty \\ Q & \infty & 0 & 2 & 4 \\ R & 3 & \infty & 0 & 1 \\ S & 1 & \infty & \infty & 0 \end{array}$$





a b c d e f g h

1 1 2 3 5 8 13 21



UNIT- IV

ITERATIVE IMPROVEMENT

The Simplex Method - The Maximum Flow Problem - Maximum Matching in Bipartite Graphs - The Stable Marriage Problem.

INTRODUCTION

Iterative Improvement Algorithm:

→ algorithm design technique for solving optimization problems.

Steps:

1. Start with a feasible solution.
2. Repeat the following step until no improvement can be found:
   - i) change the current feasible solution to a feasible solution with a better value of the objective function.
3. Return the last feasible solution as optimal.

Examples:

1. Simplex Method
2. Ford - Fulkerson algorithm for maximum flow problem.
3. Maximum matching of graph vertices.
4. Gale- Shapley algorithm for the stable marriage problem.

4.1 THE SIMPLEX METHOD

Linear Programming:

– The general problem of optimizing a linear function of several variables subject to a set of linear constraints:

maximize (or minimize) $c_1 x_1 + \dots + c_n x_n$

subject to $a_{i1} x_1 + \dots + a_{in} x_n \leq (\text{or} \geq \text{or} =) b_i$ for $i = 1, \dots, m$

$x_1 \geq 0, \dots, x_n \geq 0$

→ U.S mathematician G.B. Dantzig – father of linear Programming
– inventor of the Simplex method

Geometric Interpretation of Linear Programming:

Fundamental Properties of the problems.

Ex: 1

Linear Programming Problem in two variables:

maximize $3x + 5y$

subject to $x + y \leq 4$

$x + 3y \leq 6$

$x \geq 0, y \geq 0$

feasible solution → any point (x, y) satisfies all the constraints of the problem.

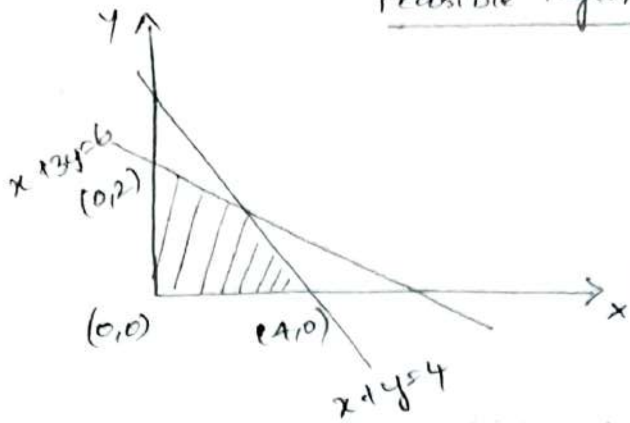
feasible region – set of all feasible points

* The points of the feasible region must satisfy all the constraints of the problem.

task:

* To find an optimal solution, a point in the feasible region with the largest value of the objective function $z = 3x + 5y$

Feasible region

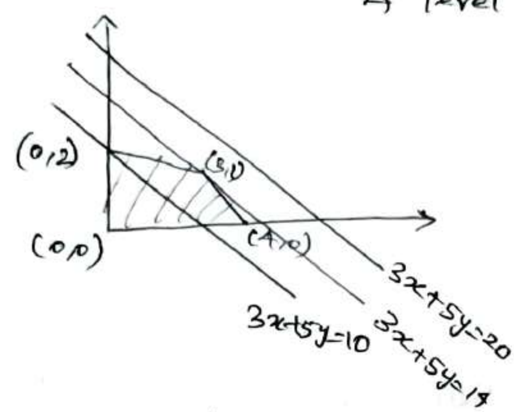


* Iterative Improvement Algorithms

* Iterative Improvement: algorithms is the algorithm design technique for solving optimization problems
- start with a feasible solution
- Repeat the following step until no improvement can be found.
  → change the current feasible solution to a feasible solution with a better value of the objective function.
- Return the last feasible solution as optimal.

ex: $3x+5y=20$
$3x+5y=10$ $\quad$ $3x+5y=z$.

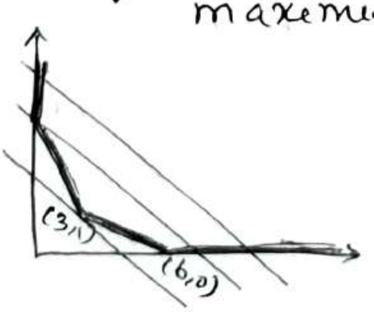
↳ level lines of the objective function



infeasible → Linear programming problems with the empty feasible region are called infeasible.
- infeasible problems donot have optimal solutions

Example: 2 Problem's feasible region is unbounded.

inequalities: $x+y \geq 4$
$x+3y \geq 6$
feasible region will become unbounded.
maximize $z = 3x+5y$



* extreme Points:
* An optimal solution to a linear programming problem can be found at one of the extreme points of its feasible region.

THE SIMPLEX METHOD

* Method used for the solution of Linear Programming Problems (LPP)

Linear Programming Problems:

* LPP consists of a linear objective function to be maximized or minimized subject to certain constraints in the form of linear equations or inequalities.

General Form:

maximize (or minimize) $c_1x_1 + \dots + c_nx_n$

subject to $a_{i1}x_1 + \dots + a_{in}x_n \leq (\text{or} \geq \text{or} =) b_i$ for $i = 1, \dots, m$

$x_1 \geq 0, \dots, x_n \geq 0.$

Example:

Linear Programming Problem in two variables:

maximize $3x + 5y$

subject to $x + y \leq 4$

$x + 3y \leq 6$

$x \geq 0, y \geq 0$

Procedure:

Step 1: Set up the initial simplex tableau.

Step 2: Determine whether the optimal solution has been reached by examining all entries in the last row

a) If all the entries are nonnegative, the optimal solution has been reached. Proceed to step 4.

b) If there are one or more negative entries, the optimal solution has not been reached. Proceed to step 3.

Step 3: Perform the pivot operation. Return to step 2

Step 4: Determine the optimal solution.

Ex:

Maximize $3x+5y$

Subject to
$$x+y \le 4$$
$$x+3y \le 6$$
$$x, y \ge 0$$

Solution:

* **Step 1:** Convert into standard form.
→ adding slack variables.

Maximize $3x+5y+0u+0v$

Subject to
$$x+y+u=4$$
$$x+3y+v=6$$
$$x, y, u, v \ge 0$$

* **Step 2:** Simplex tableau: → storing info about BFS corresponding to the extreme point.

$\boxed{x=0,\ y=0}$ — initial

	x	y	u	v	
basic variables u	1	1	1	0	4
basic variables v	1	3	0	1	6
objective row →	−3	−5	0	0	0

value of Z at $(0,0,4,6)$

↳ coefficients of obj. fn with sign reversed.

* Basic feasible solution $(0, 0, 4, 6)$
$x\ y\ u\ v$

→ $Z=0$

→ This table is not optimal.

optimality test → Because the entries in the objective row have negative number.

* **Step 3:** Next iteration

i) — find pivot column, pivot row, pivot element.

pivot column — entering variable

pivot row — departing variable

pivot column → column of most negative value in the objective row

$(-3, -5) \to -5$

	x	y	u	v	
u	1	1	1	0	4
v	1	3	0	1	6
	−3	−5	0	0	0

↑ pivot column (entering variable)

— choosing a pivot row (departing variable)

$\theta_u = \frac{4}{1} = 4$ ie) $\frac{\text{constant}}{\text{pivot column}}$

$\theta_v = \frac{6}{3} = 2$

— choose minimum value ie) 2

so departing variable is v

	x	y	u	v	
u	1	1	1	0	4
← v	1	$\boxed{3}$	0	1	6
	−3	−5	0	0	0

← pivot row

pivot element

ii) Perform Pivoting

— makes pivot element value to be 1

— divide the pivot row by pivot element

ie) $\frac{1}{3}$ 1 0 $\frac{1}{3}$ 2

	x	y	u	v	
u	1	1	1	0	4
y	$\frac{1}{3}$	1	0	$\frac{1}{3}$	2
	−3	−5	0	0	0

— Replacing other rows

Row1 = Row1 − c · Row2

Row3 = Row3 − c · Row2

c — constant of pivot column

ie) Row1 = Row1 − 1 · Row2

Row3 = Row3 − (−5) · Row2

$$\begin{array}{c|cccc|c} & x & y & u & v & \\ \hline u & 2/3 & 0 & 1 & -1/3 & 2 \\ y & 1/3 & 1 & 0 & 1/3 & 2 \\ \hline & -4/3 & 0 & 0 & 5/3 & 10 \end{array}$$

→ $\boxed{Z = 10}$

- This table is not optimal

optimality test

→ entries - negative number

→ Basic feasible solution $\boxed{(0, 2, 2, 0)}$

Row1:

$1 - 1(1/3) = 2/3$

$1 - 1(1) = 0$

$1 - 1(0) = 1$

$0 - 1(1/3) = -1/3$

$4 - 1(2) = 2$

Row3:

$-3 - (-5)\cdot 1/3 = -4/3$

$-5 - (-5)\cdot 1 = 0$

$0 - (-5)\cdot 0 = 0$

$0 - (-5)\cdot(1/3) = 5/3$

$0 - (-5)\cdot 2 = 10$

* Step 4 Next Iteration

- Find pivot column, pivot row, pivot element

pivot column → $-4/3$ (entering variable) x.

pivot row:

$$\theta_u = \frac{2}{2/3} = 3$$

$$\theta_y = \frac{2}{1/3} = 6$$

So. row $\underline{u}$ (departing variable)

$$\begin{array}{c|cccc|c} & x & y & u & v & \\ \hline \leftarrow u & 2/3 & 0 & 1 & -1/3 & 2 \\ y & 1/3 & 1 & 0 & 1/3 & 2 \\ \hline & -4/3 & 0 & 0 & 5/3 & 10 \end{array}$$

↑

pivot elt → $2/3$

- makes pivot elt to be 1

ie) $1 \quad 0 \quad 3/2 \quad -1/2 \quad 3$

Pivoting : Replacing other rows

Row2 = Row 2 − c. Row1 = Row2 − $1/3$ · Row1

Row3 = Row 3 − c · Row1 = Row 3 − $(-4/3)$. Row1

$$\begin{array}{c|cccc|c} & x & y & u & v & \\ \hline x & 1 & 0 & 3/2 & -1/2 & 3 \\ y & 0 & 1 & -1/2 & 1/2 & 1 \\ \hline & 0 & 0 & 2 & 1 & 14 \end{array}$$

value of z at $(3,1,0,0)$

$\boxed{Z=14}$ → Basic feasible solution $\boxed{(3,1,0,0)}$

Row2:

$1/3 - 1/3(1) = 0$

$1 - 1/3(0) = 1$

$0 - 1/3(3/2) = -1/2$

$1/3 - 1/3(-1/2) = 1/2$

$2 - 1/3(3) = 1$

Row 3:

$-4/3 - (-4/3)(1) = 0$

$0 - (-4/3)(0) = 0$

$0 - (-4/3)(3/2) = 2$

$5/3 - (-4/3)(-1/2) = 1$

$10 - (-4/3)(3) = 14$

optimality test

* This table is optimal

→ Because all entries in the objective row are non negative

* The maximal value of the objective function is $\boxed{14}$

x ———— x

Ex:

① linear Programming problem

maximize $3x+y$

subject to $-x+y \le 1$

$2x+y \le 4$

$x \ge 0, y \ge 0$

② maximize $6x_1 + 5x_2$

subject to $x_1 + x_2 \le 5$

$3x_1 + 2x_2 \le 12$

$x_1, x_2 \ge 0$

using tabular form

③ max. $2x - 3y + 4z$

subject to $4x - 3y + z \le 3$

$x + y + z \le 10$

$2x + y - z \le 10$

④ max. $2x - 3y + 4z$

subject to $4x - 3y + z \le 3$

$x + y - z \ge 5$

$x, y, z \ge 0$

Dual Problem.

Primal Problem

If A Linear Programming Problem

$$\text{maximize } \sum_{j=1}^{n} c_j x_j$$

$$\text{subject to } \sum_{j=1}^{n} a_{ij} x_j \le b_i \text{ for } i=1,2,\ldots,m$$

$$x_1, x_2, \ldots, x_n \ge 0$$

is considered as primal, then its dual is defined as the linear Programming Problem

Dual Problem

$$\text{minimize } \sum_{i=1}^{m} b_i y_i$$

$$\text{subject to } \sum_{i=1}^{m} a_{ij} y_i \ge c_j \text{ for } j=1,2,\ldots,n$$

$$y_1, y_2, \ldots, y_m \ge 0$$

Example:

Write the dual problem associated with this problem

$$\text{Minimize } 6x+8y$$

$$\text{subject to } 40x+10y \ge 2400$$

$$10x+15y \ge 2100$$

$$5x+15y \ge 1500$$

$$x, y \ge 0.$$

Solution.

Step 1. Write down a tableau for the primal problem

x	y	Constant
40	10	2400
10	15	2100
5	15	1500
6	8	

Step 2: Interchange the columns and rows of the tableau, and head the three columns of the resulting array with the three variables u, v and w

u	v	w	Constant
40	10	5	6
10	15	15	8
2400	2100	1500	

Step 3: Consider the tableau as initial simplex tableau, write in equation (i.e) standard maximization problem.

Dual Problem:

$$\text{maximize } 2400u + 2100v + 1500w$$

$$\text{subject to } 40u + 10v + 5w \le 6$$

$$10u + 15v + 15w \le 8$$

$$u, v, w \ge 0$$

① Ex: (NOV/DEC 2015)

Determine the Dual linear program for the following LP,

$$\text{Maximize } 3a + 2b + c$$

subject to

$$2a + b + c \le 3$$

$$a + b + c \le 4$$

$$3a + 3b + 6c \le 6$$

$$a, b, c \ge 0$$

a	b	c	
2	1	1	3
1	1	1	4
3	3	6	6
3	2	1	

Solution:

$$\text{Minimize } 3u + 4v + 6w$$

subject to

$$2u + v + 3w \ge 3$$

$$u + v + 3w \ge 2$$

$$u + v + 6w \ge 1$$

$$u, v, w \ge 0$$

u	v	w	
2	1	3	3
1	1	3	2
1	1	6	1
3	4	6	

② Ex: Find the dual of the linear programming problem

$$\text{maximize } x_1 + 4x_2 - x_3$$

$$\text{subject to } x_1 + x_2 + x_3 \le 6$$

$$x_1 - x_2 - 2x_3 \le 2$$

$$x_1, x_2, x_3 \ge 0$$

* Theorem (Extreme Point Theorem)

* Any linear Programming problem with a nonempty bounded feasible region has an optimal solution.
* An optimal solution can always be found at an extreme point of the problem's feasible region.

→ Solve a problem by computing the value of the objective function at each extreme point and selecting the one with the best value

Simplex Method:

* Inspects only a small fraction of the extreme points of the feasible region before reaching an optimal one

Steps: Idea

i) Start by identifying an extreme point of the feasible region

ii) Then check whether one can get an improved value of the objective function by going to an adjacent extreme point.

iii) If it is not the case, the current point is optimal, stop

iv) If it is the case, proceed to an adjacent extreme point with an improved value of the objective function.

v) After a finite number of steps, the algorithm will either reach an extreme point where an optimal solution occurs or determine that no optimal solution exists.

* An outline of the Simplex Method:

Task → to translate the geometric description of the simplex method into algorithmically language of algebra.

* To apply simplex method, the problem has to be represented in a special form called the standard form

The standard form has the following requirements:

*) It must be a maximization problem.

*) All the constraints must be in the form of linear equations with non negative right-hand side.

*) All the variables must be required to be non negative.

* General linear Programming Problem in standard form:

- m constraints & n unknowns $(n \geq m)$ is

$$\begin{cases} \text{maximize } c_1 x_1 + \dots + c_n x_n \\ \text{subject to } a_{i1} x_1 + \dots + a_{in} x_n = b_i, \text{ where } b_i \geq 0 \text{ for } i = 1, 2, \dots, m \\ \quad x_1 \geq 0, \dots, x_n \geq 0 \end{cases}$$

matrix notations:

$$\begin{aligned} &\text{maximize } cx \\ &\text{subject to } Ax = b \\ &\qquad\qquad x \geq 0 \end{aligned}$$

where

$$c = [c_1 c_2 \dots c_n], \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ \vdots & & & \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

* Any linear programming problem can be transformed into an equivalent problem in standard form.

→ objective function – minimized → replaced by equivalent problem of maximizing the same objective function

c_j replaced by $-c_j$

→ Constraints – inequality → replaced by an equivalent equation by adding a slack variable

inequalities:

$$\left.\begin{aligned} x + y &\leq 4 \\ x + 3y &\leq 6 \end{aligned}\right\} \Rightarrow \quad \text{equality: } \begin{aligned} &x + y + u = 4 \text{ where } u \geq 0 \\ &x + 3y + v = 6 \text{ where } v \geq 0 \end{aligned}$$

Step 1

Standard form:

$$\boxed{\begin{array}{ll} \text{maximize} & 3x+5y+0u+0v \\ \text{subject to} & x+y+u \qquad\quad = 4 \\ & x+3y+ \qquad + v = 6 \\ & x, y, u, v \geq 0 \end{array}}$$

- find the optimal solution;
  then obtain an optimal solution to problem.

adv → it provides for identifying extreme points of the feasible region.

→ basic solution
→ non basic - coordinates set to zero before solving the system
→ basic - coordinates obtained by solving the system.

Rewrite the system of constraint equations:

$$x\begin{bmatrix}1\\1\end{bmatrix} + y\begin{bmatrix}1\\3\end{bmatrix} + u\begin{bmatrix}1\\0\end{bmatrix} + v\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}4\\6\end{bmatrix}$$

Basic feasible solution (BFS)

→ If all the coordinates of a basic solution are nonnegative, the basic solution is called a basic feasible solution.

Simplex tableau:-

* The simplex method progresses through a series of adjacent extreme points with increasing values of the objective function.
  - Each point can be represented by simplex tableau
    ↓
    - a table storing the information about basic feasible solution corresponding to the extreme point.
    - Table has m+1 rows and n+1 columns
- m rows of the table contains the coefficients of a constraint equation, with the last column's entry containing the equation's right-hand side.
- columns are labelled by the names of the variables
- rows are labeled by the basic variables
- the values are in the last column.

- last row of a simplex tableau is called the objective row.

* It is initialized by the coefficients of the objective function with their signs reversed and the value of the objective function at the initial point
* On subsequent iterations, the objective row is transformed

Simplex Tableau

	x	y	u	v		
basic variables u	1	1	1	0	4	basic feasible solution $(0,0,4,6)$
basic variables v	1	3	0	1	6	
objective row →	-3	-5	0	0	0	value of z at $(0,0,4,6)$

* The objective row is used to check whether the current tableau represents an optimal solution
  - it does if all the entries in the objective row, except the one in the last column - are nonnegative
  - if this is not the case, any of the non negative entries indicates a non basic variable that can become basic in the next tableau.

* The tableau is not optimal.
  - negative value in the x-column → we can increase the value of the objective function $Z = 3x+5y+0u+0v$

  by increasing the value of the x-coordinate in the current basic feasible solution $(0,0,4,6)$

* Compensate an increase in x by adjusting the values of the basic variables u and v so that the new point is still feasible.

$$x + u = 4 \quad \text{where } u \geq 0$$
$$x + v = 6 \quad \text{where } v \geq 0$$

must be satisfied,

$$x \leq \min\{4, 6\}$$
$$= 4$$

- increase the value of x from 0 to 4, the largest amount possible, the point $(4,0,0,2)$, an adjacent to $(0,0,4,6)$ extreme point of the feasible region $z=12$.

- negative value in the y-column of the objective row
  ↳ we can increase the value of the objective function by increasing the value of the y-coordinate in the initial basic feasible solution $(0,0,4,6)$
  - This requires

$$y + u = 4 \quad \text{where } u \geq 0$$
$$3y + v = 6 \quad \text{where } v \geq 0$$

means, $y \leq \min \left\{ \frac{4}{1}, \frac{6}{3} \right\} = 2$

- increase the value of y from 0 to 2, the largest amount possible. the point $(0,2,2,0)$, another adjacent to $(0,0,4,6)$ extreme point with $z=10$.

* negative entries in the objective row.
  ↳ select the most negative one.
  → The rule yields the largest increase in the objective function's value per unit of change in a variable's value.

- feasibility constraints impose different limits on how much each of the variables

Entering variable → A new basic variable

Pivot column → column of entering variable.
→ mark the pivot column by ↑

departing variable
- basic variable to become non basic in the next tableau.

* To get to an adjacent extreme point with a larger value of the objective function, need to increase the entering variable by the largest amount possible.

Choosing a departing variable:

- for each positive entry in the pivot column, compute θ-ratio by dividing the row's last entry by the entry in the pivot column.

ex: θ-ratios are

$$\theta_u = \frac{4}{1} = 4 \; ; \qquad \theta_v = \frac{6}{3} = 2$$

- The row with the smallest θ-ratio determines the departing variable. ie) variable to become nonbasic

→ Mark the row of the departing variable, called the pivot row, by ← and denote it $\overleftarrow{row}$

* if there are no positive entries in the pivot column, no θ-ratio can be computed. *

Steps: to transform a current tableau into the next one.

transformation → pivoting

i) First, divide all the entries of the pivot row by the pivot, its entry in the pivot column, to obtain $\overleftarrow{row}_{new}$

$$\overleftarrow{row}_{new} : \frac{1}{3} \quad 1 \quad 0 \quad \frac{1}{3} \quad 2$$

ii) Then replace each of the other rows, including the objective row, by the difference

$$row - c \cdot \overleftarrow{row}_{new}.$$

c → row's entry in the pivot column

$$row\,1 - 1 \cdot \overleftarrow{row}_{new} : \frac{2}{3} \quad 0 \quad 1 \quad -\frac{1}{3} \quad 2$$

$$row\,3 - (-5) \cdot \overleftarrow{row}_{new} : -\frac{4}{3} \quad 0 \quad 0 \quad \frac{5}{3} \quad 10$$

→ The simplex method transforms tableau into the following tableau:

$$
\begin{array}{c|cccc|c}
 & x & y & u & v & \\
\hline
\leftarrow u & \frac{2}{3} & 0 & 1 & -\frac{1}{3} & 2 \\
y & \frac{1}{3} & 1 & 0 & \frac{1}{3} & 2 \\
\hline
 & -\frac{1}{3} & 0 & 0 & \frac{5}{3} & 10 \\
 & \uparrow & & & &
\end{array}
$$

→ basic feasible solution $(0, 2, 2, 0)$ with an increased value of the objective function, which is equal to 10.

– It is not optimal.

Next iteration:

$$
\begin{array}{c|cccc|c}
 & x & y & u & v & \\
\hline
x & 1 & 0 & \frac{3}{2} & -\frac{1}{2} & 3 \\
y & 0 & 1 & -\frac{1}{2} & \frac{1}{2} & 1 \\
\hline
 & 0 & 0 & 2 & 1 & 14
\end{array}
$$

→ basic feasible solution $(3, 1, 0, 0)$

– It is optimal, all the entries in the objective row are nonnegative

* The maximal value of the objective function is equal to 14.

Summary of the Simplex method:

Step 0: Initialization:

* Present a given linear programming problem in standard form

– Set up an initial tableau with nonnegative entries in the rightmost column and m other columns composing the $m \times m$ identity matrix.

– m columns define the basic variables of the initial basic feasible solution.

Step 1: optimality test

* If all the entries in the objective row are non-negative stop.
  - The tableau represents an optimal solution
  - basic variables' values are in the right most column
  - remaining, non basic variable's values are zeros.

Step 2: Finding the entering variable

* Select a negative entry from among the first n elements of the objective row
  - Mark its column to indicate the entering variable and the pivot column.

Step 3: Finding the departing variable:

* For each positive entry in the pivot column, calculate the θ-ratio by dividing that row's entry in the right-most column by its entry in the pivot column.

- Find the row with the smallest θ-ratio
- mark the row to indicate the departing variable and the pivot row.

Step 4: Forming the new tableau:

* Divide all the entries in the pivot row by its entry in the pivot column.

- subtract from each of the other rows, including the objective row, the new pivot row multiplied by the entry in the pivot column of the row.
- Replace the label of the pivot row by the variable's name of the pivot column and go back to step 1.

Analysis:

* The number of operations per iteration : $\boxed{O(nm)}$

Example Problems

1. The cannon Hill furniture company produces tables and chairs. Each table takes four hours of labor from the carpentry department. Each chair requires 3 hours of carpentry and 1 hour of finishing. During the current week, 240 hours of carpentry time are available and 100 hours of finishing time. Each table produced gives a profit of \$70 and each chair a profit of \$50. Howmany chairs and tables should be made?

Soln:-

Resource	Tables x_1	Chairs x_2	constraints
carpentry (hr)	4	3	240
Finishing (hr)	2	1	100
Unit Profit	\$70	\$50	

Objective function:

$$\text{Maximize } 70x_1 + 50x_2$$

constraints:

$$4x_1 + 3x_2 \le 240$$
$$2x_1 + x_2 \le 100$$

Non-negativity conditions:

$$x_1, x_2 \ge 0$$

LPP: Maximize $70x_1 + 50x_2$

subject to
$$4x_1 + 3x_2 \le 240$$
$$2x_1 + x_2 \le 100$$
$$x_1, x_2 \ge 0$$

→ Solve using simplex method.

* Std form
* initial BFS
* Initial simplex table.
* Finding optimal solution.

$$\boxed{\begin{array}{c} x_1 = 30, \; x_2 = 40 \\ Z = \$ 4100. \end{array}}$$

Example 2:

A farmer owns a 100 acre farm and plans to plant at most three crops. The seed for crops A, B and C costs \$40, \$20, and \$30 per acre respectively. A maximum of \$3200 can be spent on seed. Crops A, B and C require 1, 2 and 1 workdays per acre, respectively, and there are maximum of 160 workdays available. If the farmer can make a profit of \$100 per acre on crop A, \$300 per acre on crop B and \$200 per acre on crop C, how many acres of each crop should be planted to maximize profit?

4.2 THE MAXIMUM - FLOW PROBLEM

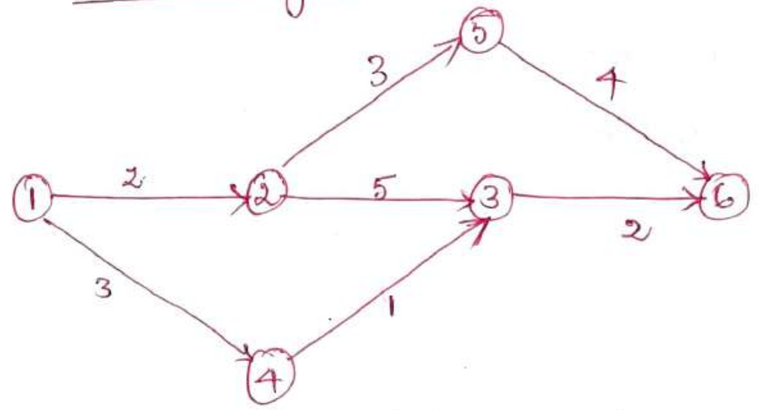
Defn:-
* maximizing the flow of a material through a transportation network

→ Flow N/w:
* Transportation network can be represented by a connected weighted digraph with n vertices numbered from 1 to n and a set of edges E. with the following properties:
   i) contains exactly one vertex with no entering edges;
      - this vertex is called source and assumed to be numbered 1
   ii) contains exactly one vertex with no leaving edges; this vertex is called the sink and assumed to be numbered n.
   iii) The weight u_{ij} of each directed edge (i,j) is a positive integer, called the edge capacity

flow network → A digraph satisfying the properties.

ex:

Network graph



Vertex number → name
edge number → edge capacities

→ Flow:-
* A flow can be redirected without consuming or adding any amount of the material.

→ Flow-Conservation requirement:
→ total amount of the material entering an intermediate vertex must be equal to the total amount of the material leaving the vertex.

x_{ij} - amount sent through edge (i,j)

* The flow-conservation requirement can be expressed by equality constraint:

$$\sum_{j:(j,i)\in E} x_{ji} = \sum_{j:(i,j)\in E} x_{ij} \quad \text{for } i = 2, 3, \dots n-1$$

- Sums in the left and right-hand sides express the total inflow and outflow entering and leaving vertex i, respectively.
- The total amount of the material leaving the source must end up at the sink.

$$\sum_{j:(1,j)\in E} x_{1j} = \sum_{j:(j,n)\in E} x_{jn}$$

Value of the flow: → total outflow from the source or total inflow into the sink.
→ denoted by v.
↳ maximize over all possible flows in a network.

flow:-
- an assignment of real numbers x_{ij} to edges (i,j) of a given network that satisfy flow-conservation constraints and capacity constraints

$$0 \le x_{ij} \le u_{ij}$$

Maximum-flow problem:-
stated as optimization problem:

maximize $v = \sum_{j:(1,j)\in E} x_{1j}$

subject to $\sum_{j:(j,i)\in E} x_{ji} - \sum_{j:(i,j)\in E} x_{ij} = 0$ for $i = 2, 3, \ldots n-1$

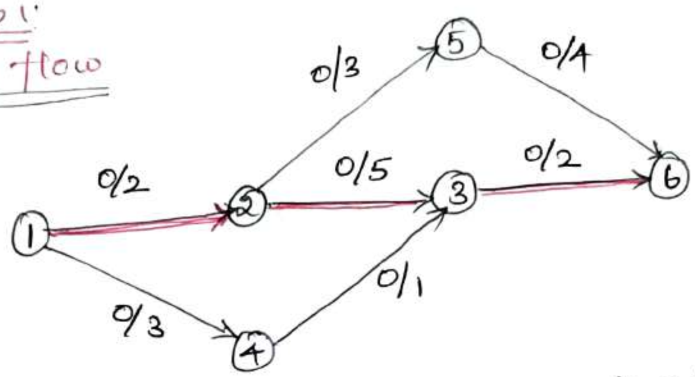
$0 \le x_{ij} \le u_{ij}$ for every edge $(i,j) \in E$

Ford-Fulkerson Method:-
Idea: Iterative Improvement: Augmenting-Path Method
i) Always start with the zero flow
ie) set $x_{ij} = 0$ for every edge (i,j)
ii) Then, on each iteration, try to find a path from source to sink
- path is called flow augmenting.

iii) If a flow-augmenting path is found, adjust the flow along the edges of the path to get a flow of an increased value and try to find an augmenting path for the new flow.

iv) if no flow-augmenting path can be found, the current flow is optimal.

Example: Step 1: Zero flow



i) Zero flow
ii) Path
iii) flow
iv) path
v) flow
⋮

Iteration: 1

– Zero amounts sent through each edge are separated from the edge capacities by the slashes.

Step 2

* Search for a flow-augmenting path from source to sink by following directed edges (i,j) for which the current flow x_{ij} is less than the edge capacity u_{ij}.

$\min\{2,5,2\} = 2$

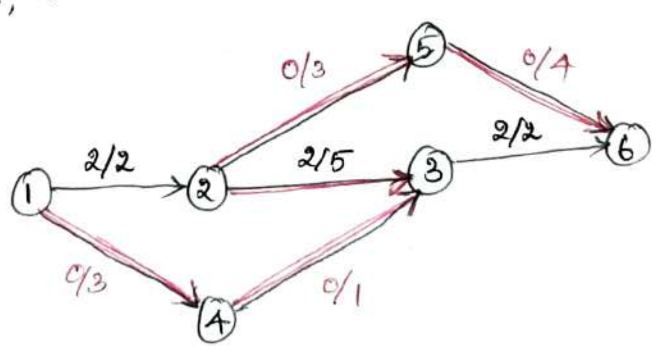
– Identify the augmenting path $1 \to 2 \to 3 \to 6$

→ increase the flow along the path by a maximum of 2 units, which is the smallest unused capacity of the edges

–s not optimal.

New flow:

Iteration 2

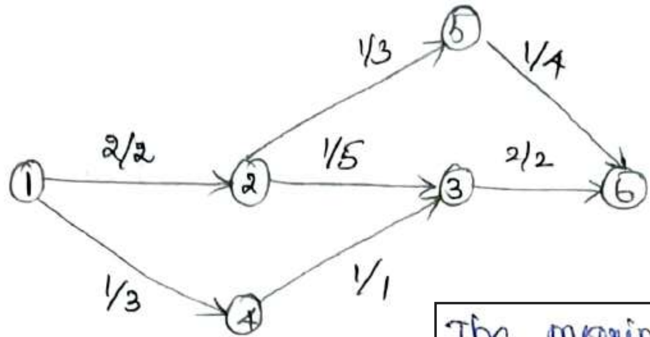


$\min\{3,1,5,3,4\} = 1$

Step 3:

– The value can be increased along the path by increasing the flow by 1 on edges $1 \to 4 \to 3 \leftarrow 2 \to 5 \to 6$ $(1,4)$, $(4,3)$, $(2,5)$ and $(5,6)$ and decreasing it by 1 on edge $(2,3)$

$2/5$ becomes $2-1/5$ ie) $1/5$
(backward edge)



$\boxed{\text{The maximal Flow is 3}}$

- It is maximal.

* To find a flow-augmenting path for a flow x, need to consider paths from source to sink in the underlying undirected graph in which any two consecutive vertices i, j are either

forward edges
i) Connected by a directed edge from i to j with some positive unused capacity $r_{ij} = u_{ij} - x_{ij}$. or

Backward edges
ii) Connected by a directed edge from j to i with some positive flow x_{ji}

forward edges:

$1 \rightarrow \cdots i \rightarrow j \cdots n$

backward edges:

$1 \rightarrow \cdot \; i \leftarrow j \cdots \rightarrow n$

eg. $1 \rightarrow 4 \rightarrow 3 \leftarrow 2 \rightarrow 5 \rightarrow 6$

$\left.\begin{array}{l}(1,4), (4,3), (2,5) \\ \quad\quad\quad 5,6\end{array}\right\} \rightarrow$ forward edges

(3, 2) $\rightarrow$ backward edge.

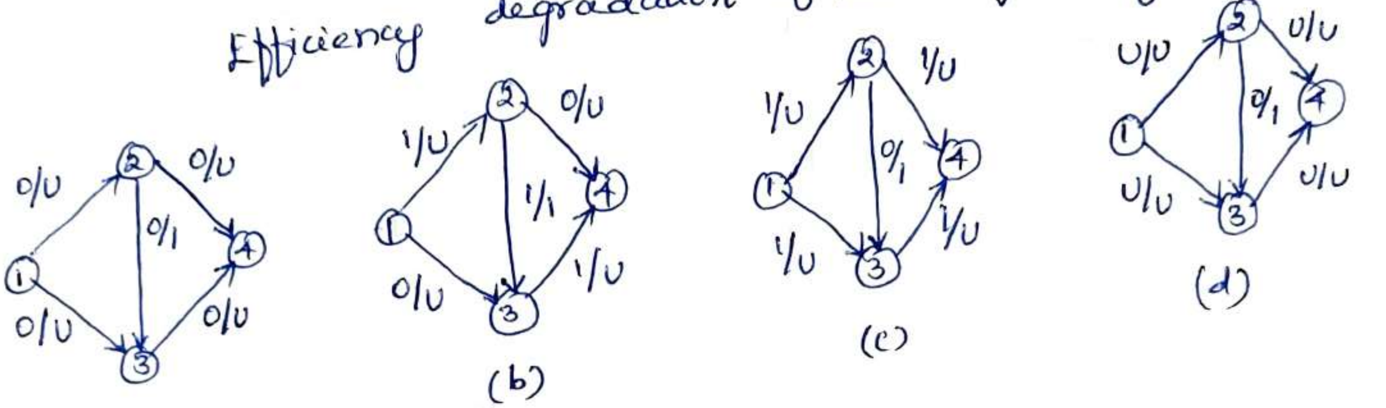
* For a given flow-augmenting path,
  r - minimum of all the unused capacities r_{ij} of its forward edges and all the flows x of its backward edges
  - increase the current flow by r on each forward edge + decrease it by the amount on each backward edge.
  - obtain a feasible flow whose value is r units greater than the value of its predecessor.

$\rightarrow i$ - intermediate vertex on a flow-augmenting path.
4 possible combinations of forward and backward edges

$\xrightarrow{+r} i \xrightarrow{+r}$, $\xrightarrow{+r} i \xleftarrow{-r}$, $\xleftarrow{-r} i \xrightarrow{+r}$, $\xleftarrow{-r} i \xleftarrow{-r}$

- the new flow will satisfy the capacity constraints
* Adding r to the flow on the first edge of the augmenting path will increase the value of the flow by r.
→ the flow value increases at least by 1 on each iteration of the augmenting-path method.
→ The augmenting-path method has to stop after a finite number of iterations.
→ The final flow always turns out to be maximal, irrespective of a sequence of augmenting paths.

Efficiency degradation of the augmenting-path method



(a) (b) (c) (d)

U → large positive integer.

* Augment the zero flow along the path 1→2→3→4
  - obtain the flow of value 1 (c)
* Augment that flow along the path 1→3←2→4 will increase the flow value to 2. (c)
* Continue selecting the pair of flow-augmenting paths, need a total of 2U iterations to reach the maximum flow of value 2U (d)

→ Augmenting Initial zero flow: 1→2→4
→ Augmenting new flow along the path 1→3→4.

Shortest-augmenting-path or first-labeled-first scanned algorithm:
→ Use breadth-first search to generate augmenting paths with the least number of edges
→ augmenting-path method

Labeling:

- marking a new vertex with two labels.

first label → amount of additional flow that can be brought from the source to the vertex being labeled.

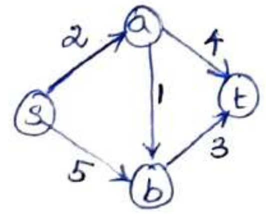
Second label → name of the vertex from which the vertex being labeled was reached.

→ add + or - sign to the second label to indicate whether the vertex was reached.

* The source can be always labeled with $\infty, -$
* For the other vertices, the labels are computed as follows:
  - If unlabeled vertex j is connected to the front vertex i of the traversal queue by a directed edge from i to j with positive unused capacity $r_{ij} = u_{ij} - x_{ij}$, then vertex j is labeled with l_j, i^+, where $l_j = \min\{l_i, r_{ij}\}$
  - if unlabeled vertex j is connected to the front vertex i of the traversal queue by a directed edge from j to i with positive flow x_{ji}, then vertex j is labeled with l_j, i^-, where $l_j = \min\{l_i, x_{ji}\}$
  - If this labeling-enhanced traversal ends up labeling the sink, the current flow can be augmented by the amount indicated by the sink's first label.
    * Augmentation is performed along the augmenting path traced by following the vertex second labels from sink to source:
      - the current flow quantities are increased on the forward edges and decreased on the backward edges of this path.
      - if the sink remains unlabeled after the traversal queue becomes empty, the algorithm returns the current flow as maximum and stops.

no. of augmenting paths - never exceeds $\frac{nm}{2}$

Example.

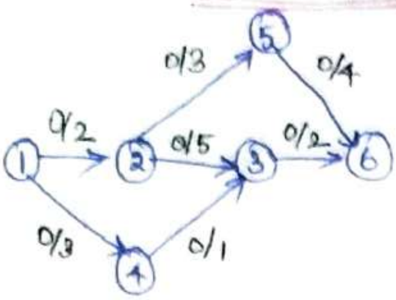


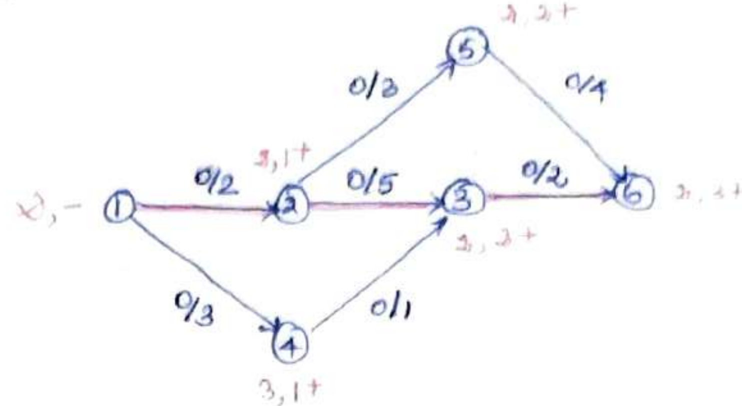
max flow = 5
(1+1+1+2)

s - a - b - t
s - a - t
s - b - a - t
s - b - t

Application of the algorithm to the network:

current flow



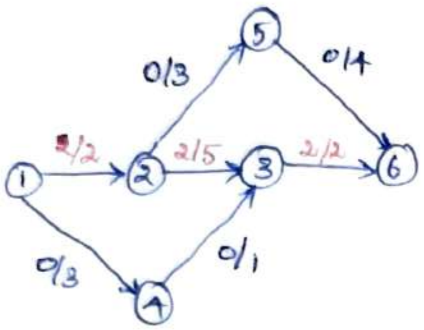


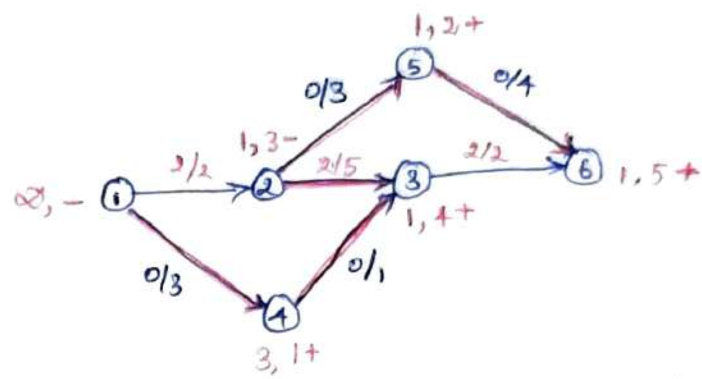
Queue : 1 2 4 3 5 6

↑ ↑ ↑ ↑

→ BFS used

Augment the flow by 2 (the sink's first label) along the path 1→2→3→6

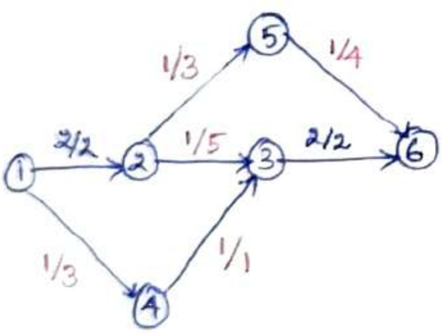


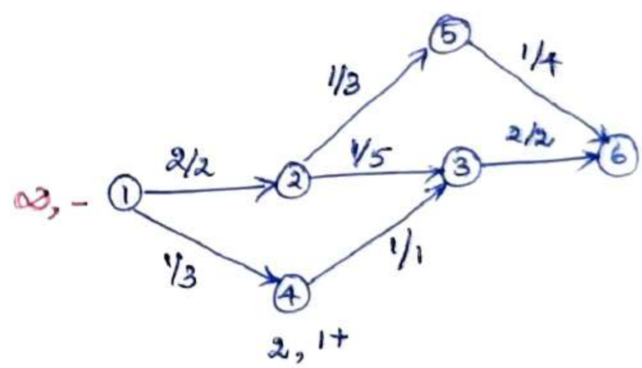


Queue : 1 4 3 2 5 6

↑ ↑ ↑ ↑ ↑

Augment the flow by 1 (the sink's first label) along the path 1→4→3←2→5→6





Queue : 1 4

↑ ↑

No augmenting path (the sink is unlabelled); the current flow is maximal.

i.e) 2+1 = 3

Note: 4 $\frac{1}{3}$ [remaining 2] so 1→4 is possible but 4→3 not possible

ALGORITHM Shortest AugmentingPath (G)

// Implements the shortest - augmenting - path algorithm

// Input : A network with single source 1, single sink n, and

// positive integer capacities u_{ij} on its edges (i, j)

// Output : A maximum flow x

assign $x_{ij} = 0$ to every edge (i,j) in the network
label the source with ∞, $-$ and add the source to the empty queue Q

```
While not Empty(Q) do
    i ← Front(Q); Dequeue(Q)
    for every edge from i to j do          // forward edges
        if j is unlabeled
            r_ij ← u_ij − x_ij
            if r_ij > 0
                l_j ← min{l_i, r_ij}; label j with l_j, i+
                Enqueue(Q, j)
    for every edge from j to i do          // backward edges
        if j is unlabeled
            if x_ji > 0
                l_j ← min{l_i, x_ij}; label j with l_j, i−
                Enqueue(Q, j)
    if the sink has been labeled
        j ← n
        while j ≠ 1
            if the second label of vertex j is i−
                x_ij ← x_ij + l_n
            else
                x_ji ← x_ji − l_n
            j ← i; i ← the vertex indicated by i's second label
        erase all vertex labels except the ones of the source
        reinitialize Q with the source
return x
```

* **Network cut:**
A cut induced by partitioning vertices of a network into some subset X containing the source and $\bar{X}$, the complement of X, containing the sink is the set of all the edges with a tail

in X and a head in $\bar{X}$.

* denoted by $C(X, \bar{X})$

ex

if $X = \{1\}$; $\bar{X} = \{2,3,4,5,6\}$; $C(X, \bar{X}) = \{(1,2), (1,4)\}$

if $X = \{1,2,3,4,5\}$; $\bar{X} = \{6\}$; $C(X, \bar{X}) = \{(3,6), (5,6)\}$;

if $X = \{1,2,4\}$; $\bar{X} = \{3,5,6\}$; $C(X, \bar{X}) = \{(2,3), (2,5), (4,3)\}$

capacity of a cut

* The capacity of a cut $C(X, \bar{X})$ denoted $c(X, \bar{X})$

- defined as the sum of capacities of the edges that compose the cut.

n - no of vertices
m - no of edges

ex - capacities are equal to 5, 6 & 9.

Minimum cut → cut with the smallest capacity.

Find path = $O(nm)$ = $O(m)$

Analysis: Time efficiency of the shortest augmenting path algm = $\boxed{O(nm^2)}$

Theorem: Max-Flow Min-Cut Theorem:

* The value of a maximum flow in a network is equal to the capacity of its minimum cut.

Proof: let x - feasible flow of value v

let $C(X, \bar{X})$ - cut of capacity c in the network.

→ flow across the cut defined as the difference between the sum of the flows on the edges from X to $\bar{X}$ and the sum of the flows on the edges from $\bar{X}$ to X.

→ v, the value of the flow

$$v = \sum_{i \in X, j \in \bar{X}} x_{ij} - \sum_{j \in \bar{X}, i \in X} x_{ji}$$

$$v \leq \sum_{i \in X, j \in \bar{X}} x_{ij} \leq \sum_{i \in X, j \in \bar{X}} u_{ij}$$

ie) $v \leq c$

- The value of the flow cannot exceed the capacity of any cut in the network.

$$v^* = \sum_{i \in X^*, j \in \bar{X}^*} x_{ij}^* - \sum_{j \in \bar{X}^*, i \in X^*} x_{ji}^* = \sum_{i \in X^*, j \in \bar{X}^*} u_{ij} - 0 = c(X^*, \bar{X}^*)$$

which proves the theorem.

Edmond.

4.3 MAXIMUM MATCHING IN BIPARTITE GRAPHS

* Represent elements of two given sets by vertices of a graph, with edges between vertices that can be paired.

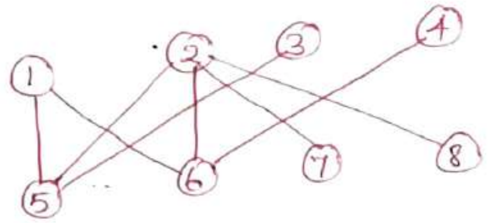
✓ **matching** :-

- A matching in a graph is a subset of its edges with the property that no two edges share a vertex.

✓ **maximum matching**:

- a maximum cardinality matching
- is a matching with the largest number of edges.

Ex: Bipartite graph



✓ **maximum -matching problem**:

- The problem of finding a maximum matching in a given graph.
- solved by Jack Edmonds in 1965.

✓ **Bipartite graph**:

- all the vertices can be partitioned into two disjoint sets V and U, not necessarily of the same size, so that every edge connects a vertex in one of the sets to a vertex in the other set.

* A graph is bipartite if its vertices can be colored in two colors so that every edge has its vertices colored in different colors

* Graphs are also said to be 2-colorable.

* **Iterative-improvement technique**:

* Let M - matching in a bipartite Graph G = (V, U, E)

Find a new matching with more edges:

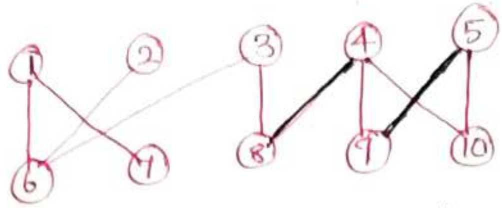
* if every vertex in either V or U is matched (has a mate),

i.e) serves as an endpoint of an edge in M, this cannot be done and M is a maximum matching.

* To improve current matching, both V and U must contain unmatched (free) vertices. i.e) vertices that are not incident to any edge in M.

* Ex:

Step 1:



$M_a = \{(4,8),(5,9)\}$

Vertices 1, 2, 3, 6, 7, 10 – free

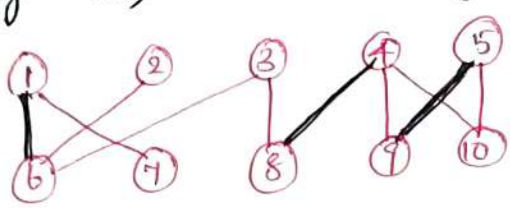
Vertices 4, 5, 8, 9 – matched.

→ increase a current matching by adding an edge between two free vertices.

– adding (1,6) to the matching

Step 2:

$M_b = \{(1,6),(4,8),(5,9)\}$



Augmenting path : 1, 6

→ find a matching larger than M_b by matching vertex 2

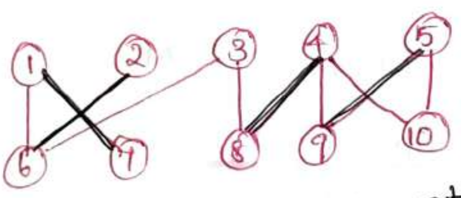
– include the edge (2,6) in a new matching.

– requires removal of (1,6)

– inclusion of (1,7) in the new matching.

Step 3:

$M_c = \{(1,7),(2,6),(4,8),(5,9)\}$



Augmenting path : 2, 6, 1, 7

→ increase the size of a current matching M by constructing a simple path from a free vertex in V to a free vertex in U whose edges are alternately in E−M and in M.

* The first edge of the path does not belong to M, the second one does and so on, until the last edge that does not belong to M.

– a path is called augmenting with respect to the matching M.

Augmentation:

* Since the length of an augmenting path (path from a free vertex in V to a free vertex in U) is always odd, adding to the matching M the path's edges in the odd-numbered positions and deleting from it the path's edges in the even-numbered positions yields a matching with one more edge than in M.

- such a matching adjustment is called augmentation

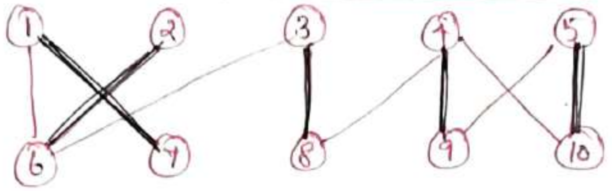
→ 3, 8, 4, 9, 5, 10 is an augmenting path for the matching M_c

→ After adding to M_c the edges (3,8), (4,9) and (5,10) and deleting (4,8) and (5,9)

- obtain the matching $M_d = \{(1,7), (2,6), (3,8), (4,9), (5,10)\}$

Step 4:

Maximum Matching



Augmenting path: 3, 8, 4, 9, 5, 10

* The matching M_d is not only a maximum matching but also Perfect

ie) matching that matches all the vertices of the graph.

Theorem: A matching M is a maximum matching if and only if there exists no augmenting path with respect to M.

Proof:

* If an augmenting path with respect to matching M exists, then the size of the matching can be increased by augmentation.
* if no augmenting path with respect to a matching M exists, then the matching is a maximum matching.

M^* – maximum matching in G

$|M^*| > |M|$

$M \oplus M^* = (M - M^*) \cup (M^* - M)$

Steps: * General Method: for constructing a maximum matching

- start with some initial matching
- Find an augmenting path
- augment the current matching along the path
- When no augmenting path can be found, terminate the algorithm
- return the last matching, which is maximum.

Specific algorithm:

- Search for an augmenting path for a matching M by a BFS - like traversal of the graph.
- starts simultaneously at all the free vertices in one of the sets V and U

- augmenting path:
  - if it exists, is an odd-length path that connects a free vertex in V with a free vertex in U and which, unless it consists of a single edge "zigs" from a vertex in V to another vertex' mate in U, then "zags" back to V along the uniquely defined edge from M and so on until a free vertex in U is reached.

Rules for labeling vertices during the BFS-like traversal of the graph.

Case 1 (the queue's front vertex w is in V)
- if u is a free vertex adjacent to w, it is used as the other endpoint of an augmenting path; so the labeling stops and augmentation of the matching commences.
- if u is not free and connected to w by an edge not in M, label u with w unless it has been already labeled.

Case 2: (the front vertex w is in U)
- w must be matched and label its mate in V with w.

* Pseudocode:

ALGORITHM Maximum Bipartite Matching (G)

// Finds a maximum matching in a bipartite graph by a BFS-like traversal

// Input: A bipartite graph G = (V, U, E)

// Output: A maximum-cardinality matching M in the input graph

initialize set M of edges with some valid matching

\# initialize queue Q with all the free vertices in V

\#

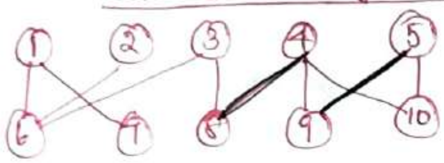
```
while not Empty(Q) do
    w ← Front(Q); Dequeue(Q)
    if w ∈ V
        for every vertex u adjacent to w do
            if u is free
                M ← M ∪ {w, u}
                v ← w
                while v is labeled do
                    u ← vertex indicated by v's label;
                        M ← M − (v, u)
```

```
            v ← vertex indicated by u's label;
                M ← M ∪ {(v, u)}
            remove all vertex labels
            reinitialize Q with all free vertices in V
            break
    else
        if (w, u) ∉ M and u is unlabeled
            label u with w
            Enqueue(Q, u)
        else
            label the mate v of w with w
            Enqueue(Q, v)
return M
```

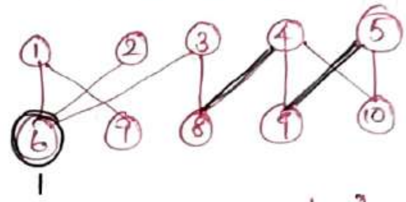
* Application of the algorithm

current matching & initialized queue

vertex labeling generated by the algorithm

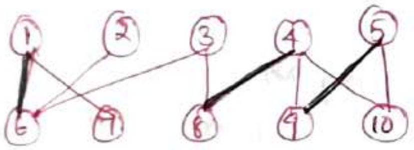


Queue 1 2 3

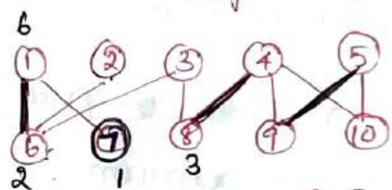


Queue: 1 2 3

Augment from 6

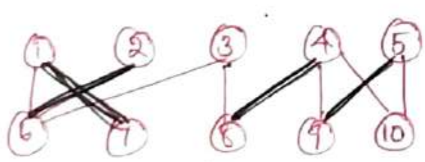


Queue 2 3

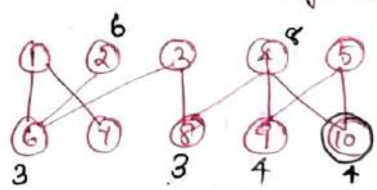


Queue 2 3 6 8 1 4

Augment from 7

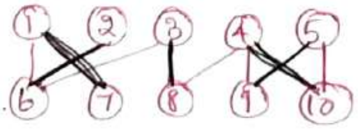


Queue 3



Queue 3 6 8 2 4 9

Augment from 10



Queue : empty ⇒ maximum matching

Analysis: Time efficiency of the algorithm is in $\boxed{O(n(n+m))}$

4.4 THE STABLE MARRIAGE PROBLEM

– version of bipartite matching called the stable marriage problem.

* Consider a set $Y = \{m_1, m_2, \ldots, m_n\}$ of n men and set $X = \{w_1, w_2, \ldots, w_n\}$ of n women.
* Each man has a preference list ordering the women as potential marriage partners with no ties allowed.
* Each woman has a preference list of the men, also with no ties.

Ex: Data for an instance of the stable marriage problem

men's preferences

	1st	2nd	3rd
Bob:	Lea	Ann	Sue
Jim:	Lea	Sue	Ann
Tom:	Sue	Lea	Ann

women's preferences

	1st	2nd	3rd
Ann:	Jim	Tom	Bob
Lea:	Tom	Bob	Jim
Sue:	Jim	Tom	Bob

ranking matrix

	Ann	Lea	Sue
Bob:	$\boxed{2,3}$	1,2	3,3
Jim:	3,1	$\boxed{1,3}$	2,1
Tom:	3,2	2,1	$\boxed{1,2}$

first position → position of w in the m's preference list

Second → position of m in the w's preference list

ex: pair 3,1

Marriage matching M

* is a set of n (m, w) pairs whose members are selected from disjoint n-element sets Y and X in a one-one fashion.

ie) each man m from Y is paired with exactly one woman w from X and vice versa.

Blocking pair:

* A pair (m, w), where $m \in Y$, $w \in X$, is said to be a blocking pair for a marriage matching M if man m and woman w are not matched in M but they prefer each other to their mates in M.

ex: (Bob, Lea) – blocking pair

- Bob prefers Lea to Ann
- Lea prefers Bob to Jim.

Stable: – A marriage matching M is called stable if there is no blocking pair for it

Unstable – blocking pair for it.

Stable marriage problem:

— to find a stable marriage matching for men's and women's preferences.

* Stable Marriage Algorithm:

Input: A set of n men and a set of n women along with rankings of the women by each man and rankings of the men by each woman with no ties allowed in the rankings

Output: A stable marriage matching

Step 0: Start with all the men and women being free

Step 1: While there are free men, arbitrarily select one of them and do the following:

Proposal:

* The selected free man m proposes to w, the next woman on his preference list

Response:

* If w is free, she accepts the proposal to be matched with m.
* If she is not free, she compares m with her current mate.
  - If she prefers m to him, she accepts m's proposal making her former mate free; otherwise, she simply rejects m's proposal, leaving m free.

Step 2: Return the set of n matched pairs.

* Application of the Stable marriage Algorithm:

Step 1

Free men: Bob, Jim, Tom

	Ann	Lea	Sue
Bob	2,3	$\boxed{1,2}$	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Bob proposed to Lea

Lea accepted

Step 2:

Free men:
Jim, Tom

	Ann	Lea	Sue
Bob	2,3	$\boxed{1,2}$	3,3
Jim	3,1	<u>1,3</u>	2,1
Tom	3,2	2,1	1,2

Jim proposed to Lea
Lea rejected.

Step 3

Free men:
Jim, Tom

	Ann	Lea	Sue
Bob	2,3	$\boxed{1,2}$	3,3
Jim	3,1	1,3	$\boxed{2,1}$
Tom	3,2	2,1	1,2

Jim proposed to Sue
Sue accepted.

Step 4:

Free men:
Tom

	Ann	Lea	Sue
Bob	2,3	$\boxed{1,2}$	3,3
Jim	3,1	1,3	$\boxed{2,1}$
Tom	3,2	2,1	<u>1,2</u>

Tom proposed to Sue
Sue rejected.

Step 5:

Free men:
Tom

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	$\boxed{2,1}$
Tom	3,2	$\boxed{2,1}$	1,2

Tom proposed to Lea
Lea replaced Bob with Tom

Step 6:

Free men:
Bob

	Ann	Lea	Sue
Bob	$\boxed{2,3}$	1,2	3,3
Jim	3,1	1,3	$\boxed{2,1}$
Tom	3,2	$\boxed{2,1}$	1,2

Bob proposed to Ann
Ann accepted.

boxed cell → accepted proposal
underlined cell → rejected proposal.

Bob – Ann
Jim – Sue
Tom – Lea

Properties of the stable marriage problem:

Theorem: The stable marriage algorithm terminates after no more than n^2 iterations with a stable marriage output.

Proof:
- The algorithm starts with n men having the total of n^2 women on their ranking lists.
- On each iteration, one man makes a proposal to a woman.
- The algorithm must stop after no more than n^2 iterations

Prove → the final matching M is a stable marriage matching

unstable → blocking pair of man m and a woman w, who are unmatched in M.

- m must have proposed to w on some iteration
- whether w refused m's proposal or accepted it but replaced him on a subsequent iteration with a higher-ranked match, w's mate in M must be higher on w's preference list than m

* Analysis: Time complexity is $\boxed{O(n^2)}$. n → number of men or women

disadvantage:

* favors men's preferences over women's preferences.

ex:

	woman 1	woman 2
man 1	1,2	2,1
man 2	2,1	1,2

* The algorithm always yields man-optimal & gender-optimal stable matching.

Ex:

1) Men's Preference

1	2	4	1	3
2	3	1	4	2
3	2	3	1	4
4	4	1	3	2

Women's Pref.

1	2	1	4	3
2	4	3	1	2
3	1	4	3	2
4	2	1	4	3

2) Men's Preference

	1	2	3	4
A	γ	β	δ	α
B	β	α	γ	δ
C	β	δ	α	γ
D	γ	α	δ	β

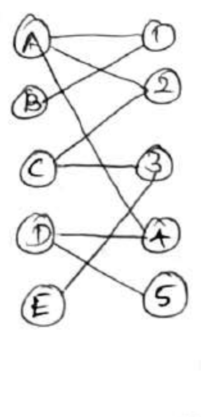
Women's Pref

	1	2	3	4
α	A	B	D	C
β	C	A	D	B
γ	C	B	D	A
δ	B	A	C	D

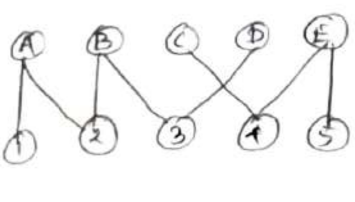
Maximum Matching

1)

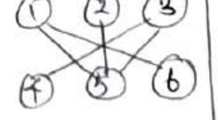
2)



3)



4)



3) Men's Preference

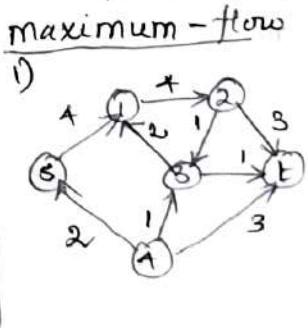
	1	2	3	4
A	Y	W	X	Z
B	W	X	Y	Z
C	Y	Z	W	X
D	Y	Z	W	X

Women's Pref.

	1	2	3	4
X	B	C	D	A
Y	C	D	B	A
Z	C	B	A	D
W	C	B	D	A

maximum-flow

1)



	A	B	C	D
α	1,3	2,3	3,2	4,3
β	1,4	4,1	3,4	2,2
γ	2,2	1,4	3,3	4,1
δ	4,1	2,2	3,1	1,4

UNIT – V

COPING WITH THE LIMITATIONS OF ALGORITHM POWER

Lower - Bound Arguments – P, NP, NP - Complete - and NP- Hard Problems. Backtracking – n - Queen Problem - Hamiltonian Circuit Problem - Subset Sum Problem – Branch and Bound - LIFO Search and FIFO Search - Assignment Problem - Knapsack Problem - Traveling Salesman Problem - Approximation Algorithms for NP-Hard Problems – Traveling Salesman Problem – Knapsack Problem.

Limitations of Algorithm Power

* Algorithms for solving a variety of different problems.
* Power of algorithm is not unlimited.

→ Some problems cannot be solved by any algorithm.

→ Some problems can be solved algorithmically but not in polynomial time

→ Some problems can be solved in polynomial time but there are lower bounds for efficiency of the algorithms.

LOWER-BOUND ARGUMENTS.

* The efficiency of an algorithm can be expressed in two ways:
  i) Asymptotic efficiency class
  ii) Comparing efficiency of a particular algorithm with respect to other algorithms for the same problem.

→ to know the <u>best possible efficiency</u> class for a problem among the algorithms that could solve the problem.

<u>Lower bound</u>:

→ an estimate on a minimum amount of work needed to solve a given problem.

- can be exact count or an efficiency class (O)

Tight.

- A bound is tight if there exists an algorithm with the same efficiency as the lower bound.

Problem	Lower bound	Tightness
i) Sorting	$\Omega(n \log n)$	Yes
ii) Searching	$\Omega(\log n)$	Yes
iii) n-digit integer multiplication	$\Omega(n)$	Unknown.

Methods for establishing lower bounds.

i) Trivial lower bounds

ii) Information-Theoretic Arguments

iii) Adversary Arguments.

iv) Problem Reduction.

① Trivial Lower Bounds:

- based on counting the number of items that must be processed in input and the number of output items that need to be produced.
- Algorithm must at least "read" all the items it needs to process an "write all its outputs, such a count yields a trivial lower bound.

ex: finding max element.

* Problem of evaluating a polynomial of degree n.

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_0.$$

- all the coefficients have to be processed

$\rightarrow \Omega(n)$ – linear.

② Information – Theoretic Arguments:

- based on the amount of information the algorithm has to produce as an output rather than the input/output.

ex: game of deducing a positive integer between 1 & n

- Answers can be either yes or no
- $\log_2 n$

→ connection to information theory ; → problems – comparisons – sorting, searching

Adversary Arguments

- proving a lower bound by playing role of adversary that makes algorithm work the hardest by adjusting input.

- forces the algorithm to work hard by given the worst possible answer.

ex:
- Guessing a number between 1 and n using yes/no questions
- Merging two sorted lists of size n:

  $2n-1$ comparisons.

Problem Reduction

- If problem P is at least as hard as problem Q, then a lower bound for Q is also a lower bound for P.
- find problem Q with a known lower bound that can be reduced to problem P.
- any algorithm that solves P will also solve Q

ex:
- finding Minimum spanning tree.
- element uniqueness problem.

→ lower bound of element uniqueness problem – $O(n \log n)$

  " " MST – $O(n \log n)$

P, NP and NP-Complete Problems:

*** DEFINITION 1:**

* An algorithm solves a problem in polynomial time if its worst-case time efficiency belongs to $O(p(n))$ where $p(n)$ is a polynomial of the problem's input size n.

→ tractable – Problems that can be solved in polynomial time.

→ intractable – Problems that cannot be solved in polynomial time

* Computational complexity → seeks to classify problems according to their inherent difficulty.

* P and NP Problems:

P:

ex: → computing the product, gcd of two integers, sorting a list, searching for a key in a list. finding MST.

→ More formal definition includes in P only decision problems, which are problems with yes/no answers

*** DEFINITION 2:**

- Class P is a class of decision problems that can be solved in polynomial time by (deterministic) algorithms.
- This class of problems is called polynomial.

* undecidable → problems that cannot be solved at all by any algorithm

* decidable → problems that can be solved by an algorithm

* halting problem: Given a computer program and an input to it, determine whether the program will halt on that input or continue working indefinitely on it.

– A is an algorithm that solves the halting problem. any program P and input I

$$A(P, I) = \begin{cases} 1, & \text{if Program } P \text{ halts on input } I. \\ 0, & \text{if program } P \text{ does not halt on input } I. \end{cases}$$

* **DEFINITION 3:**

  * A nondeterministic algorithm is a two-stage procedure that takes as its input an instance I of a decision problem

    i) Nondeterministic ("guessing") stage:

    ii) Deterministic ("verification") stage:

      - takes both I and S as its input and outputs yes if S represents a solution to instance I.

  * Nondeterministic algorithm solves a decision problem if and only if for every yes instance of the problem it returns yes on some execution.

  * A nondeterministic algorithm is said to be nondeterministic polynomial if the time efficiency of its verification stage is polynomial.

* **DEFINITION 4:**

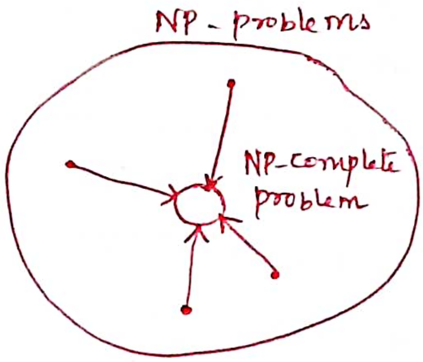
  * Class NP is the class of decision problems that can be solved by nondeterministic polynomial algorithms. This class of problems is called nondeterministic polynomial.

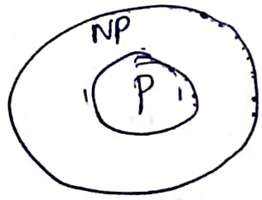
    - Most decision problems are in NP.
    - This class includes all the problems in P.

$$P \subseteq NP$$

NP-complete:

* An NP-complete problem is a problem in NP that can be reduced to it in polynomial time.





* **DEFINITION 5:**

  * A decision problem D_1 is said to be polynomially reducible to a decision problem D_2 if there exists a function t that transforms instances of D_1 to instances of D_2 such that.

1. t maps all yes instances of D_1 to yes instances of D_2 and all no instances of D_1 to no instances of D_2;
2. t is computable by a polynomial-time algorithm.

* If a problem D_1 is polynomially reducible to some problem D_2 that can be solved in polynomial time, then problem D_1 can also be solved in polynomial time.

NP-Complete Problems:

DEFINITION 6:

* A decision problem D is said to be NP-complete if
  1. it belongs to class NP.
  2. every problem in NP is polynomially reducible to D.

ex:

- The Hamiltonian circuit problem is polynomially reducible to the decision version of the traveling salesman problem.

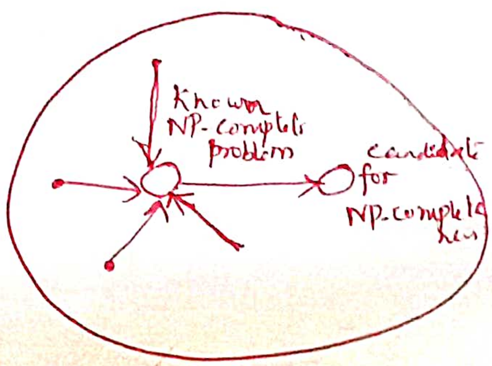
* map a graph G of a given instance of the Hamiltonian circuit problem to a complete weighted graph G' representing an instance of the traveling salesman problem.

* The notion of NP-completeness requires polynomial reducibility of all problems in NP.

→ showing that a decision problem is NP-complete can be done in two steps:

i) To show that the problem is in NP
ii) To show that every problem in NP is reducible to the problem in polynomial time.

ex: Hamiltonian circuit problem is NP-complete.



NP-completeness by reduction.

P = NP

* find a deterministic polynomial-time algorithm for one NP-complete problem, then every problem in NP can be solved in polynomial-time by a deterministic algorithm.

COPING WITH THE LIMITATIONS OF ALGORITHM POWER

Two algorith design techniques:

* backtracking
* brand - and - bound

– improvement over exhaustive search

↳ They construct candidate solutions one component at a time and evaluate the partially constructed solutions.

– The approach makes it possible to solve large instances of difficult combinatorial problems.

→ based on the construction of a state-space tree whose reflect specific choices made for a solution's components.

→ terminate a node as soon as it can be guaranteed that no solution to the problem can be obtained

BACKTRACKING

Idea:

* To construct solutions one component at a time and evaluate partially constructed candidate as follows:
  → if a partially constructed solution can be developed further without violating the problem's constraints, then remaining options are considered.
  → If there is no legitimate option for the next component, no alternatives for any remaining component need to be considered
* The algorithm backtracks to replace the component of the partially constructed solution with its next option.

State-space tree:

– Implement Backtracking

– constructing a tree of choices being made.

root → initial state before the search for a solution begins.

first-level nodes → choices made for the first component of a solution

Second-level nodes → choices for the second component.

Promising:

* A node in a state-space tree is said to be promising if it corresponds to a partially constructed solution that may still lead to a complete solution.

non-promising

* does not lead to a complete solution.

Leaves → nonpromising dead ends or complete solutions.

Construction state space tree → DFS

→ If the current node is promising, its child is generated by adding the first remaining legitimate option for the next component of a solution and processing moves to this child.

→ If the current node turns out to be nonpromising, the algorithm backtracks to the node's parent to consider the next possible option for its last component

→ if there is no such option, it backtracks one more level up the tree, and so on.

→ Finally, if the algorithm reaches a complete solution to the problem, it either stops or continues searching for other possible solutions.

---

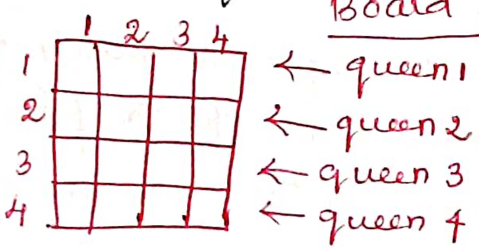
n - Queens Problem

* The problem is to place n queens on an n×n chess board so that no two queens attack each other by being in the same row or in the same column or on the same diagonal

Ex: Four - Queens Problem: (i) $n = 4$

* Each of the 4 queens has to be placed in its own row
- to assign a column for each queen on the board.

Board for the four-queens problem



* If other solutions need to be found, the algorithm can resume its operations at the leaf at which it stopped.

- A single solution to the n-queens problem for any $n \geq 4$ can be found in linear time.

Analysis Time Complexity is $\boxed{O(n)}$.

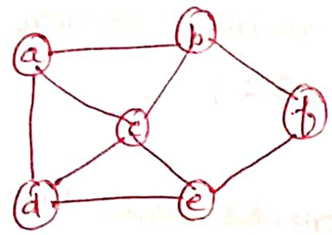
Hamiltonian Circuit Problem

→ finding a Hamiltonian circuit in the graph.

↓ is a cycle in an undirected graph which visits each vertex exactly once and also returns to the starting vertex.

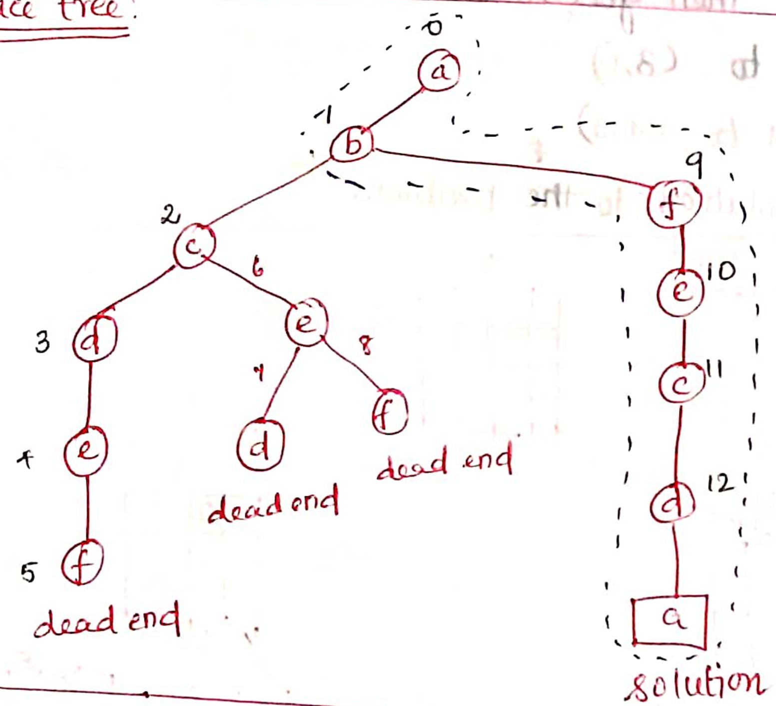
→ Hamiltonian path - is a path in an undirected graph which visits each vertex exactly once.

Ex: Graph:



* if a Hamiltonian circuit exists, it starts at vertex a.
- make vertex a the root of the state-space tree.

State-Space tree:



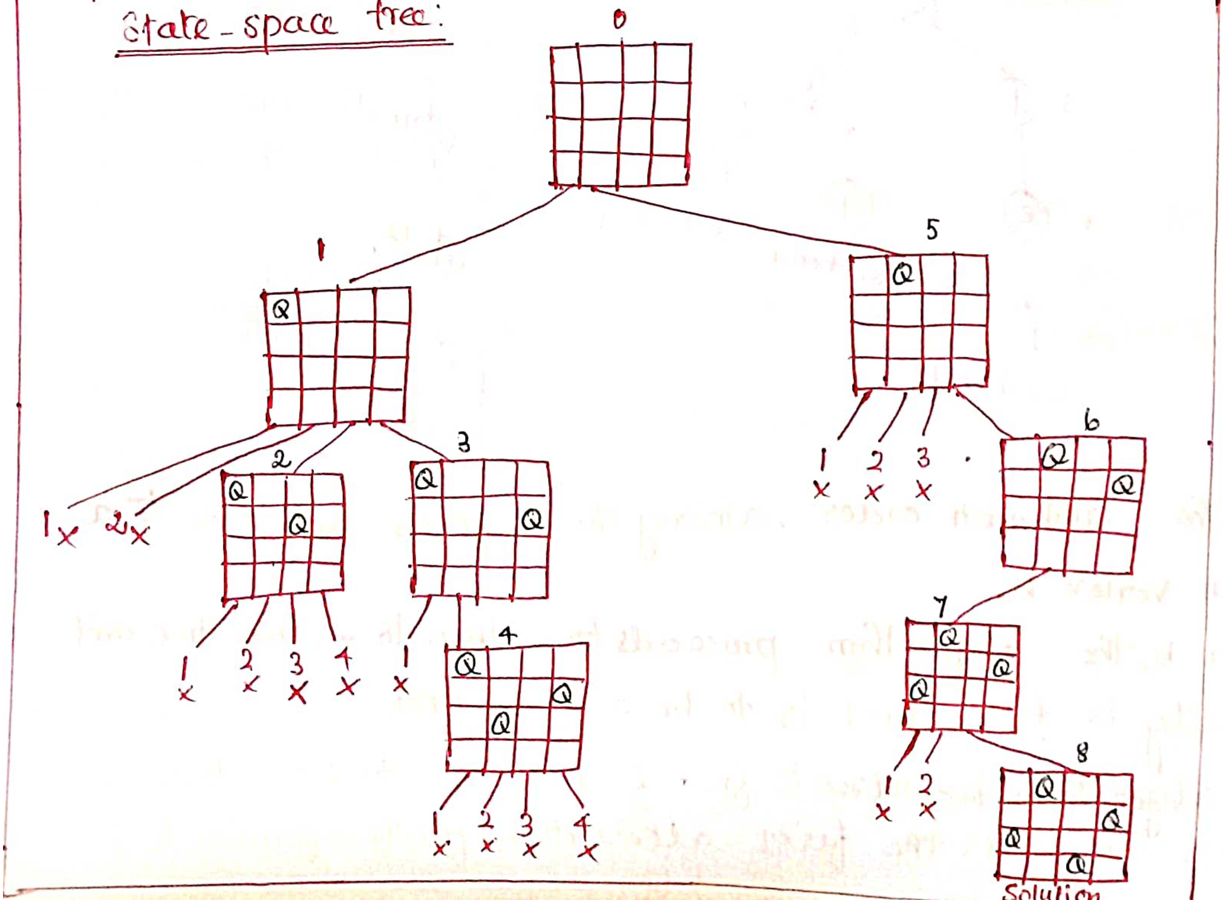
* Use the alphabet order, among the vertices adjacent to a
- Select vertex b.
- From b, the algorithm proceeds to c, then to d, then to e and finally to f, which is to be a dead end

* The algorithm backtracks from f to e, then to d and then to c, which provides the first alternative for the algorithm to pursue.

Procedure

* Start with the empty board
* place queen 1 in the first possible position of its row, which is in column 1 of row 1.
* Place queen 2, after trying unsuccessfully columns 1 and 2, in the first acceptable position for it, which is square (2,3), the square in row 2 and column 3
  → dead end because there is no acceptable position for queen 3.
* So, the algorithm backtracks
* place queen 2 in the next possible position at (2,4).
* Then, queen 3 is placed at (3,2)
  → dead end
* Then, the algorithm backtracks all the way to queen 1 and moves it to (1,2)
* Queen 2 then goes to (2,4)
* Queen 3 to (3,1)
* Queen 4 to (4,3)
  — Solution to the problem

State-space tree:



* Going from c to e, — dead end.
* The algorithm has to backtrack from e to c and then to b.
- From b, it goes to the vertices f, e, c and d, from which it can return to a.

Soln: $\boxed{\text{Hamiltonian circuit : } a, b, f, e, c, d, a.}$

→ To find another Hamiltonian circuit, continue this process by backtracking from the leaf of the solution found.

Time complexity is $\boxed{O(d^n)}$

d → Out going edges

SUBSET - SUM PROBLEM

* Find a subset of a given set $A = \{a_1, \ldots, a_n\}$ of n positive integers whose sum is equal to a given positive integer d.

ex:

$A = \{1, 2, 5, 6, 8\}$ and $d = 9$.

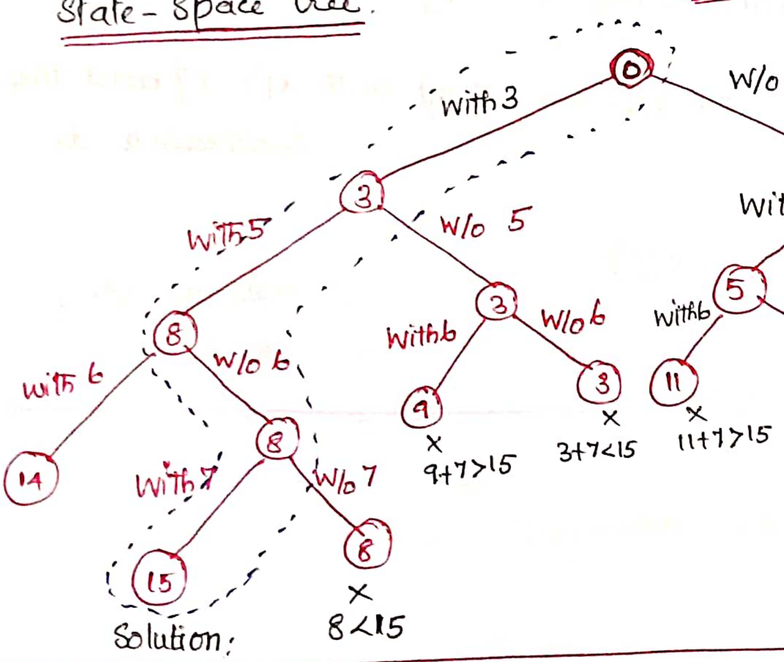
Two solutions: $\{1, 2, 6\}$ & $\{1, 8\}$

- Sort the set's elements in increasing order

Assume: $a_1 < a_2 < \cdots < a_n$

Ex: $A = \{3, 5, 6, 7\}$ and $d = 15$

State-Space tree:



Solution: $\{3, 5, 7\}$

→ The number inside a node - sum of the elements already included in the subsets represented by the node.

→ root of the tree - starting point, with no decisions about the given elements

* left children → inclusion of a_1
right children → exclusion of a_1
* going to the left from a node of the first level - inclusion of a_2
going to the right corresponds to exclusion of a_2
and so on.
* A path from the root to a node on the i th level of the tree → first i numbers have been included in the subsets represented by that node.
- record the value of s, the sum in the node.
* If s is equal to d → solution of the problem.
* To find all solutions, continue by backtracking to the node's parent.
* If s is not equal to d → terminate the node as nonpromising two inequalities hold:

i) $s + a_{i+1} > d$ (sum is too large)

ii) $s + \sum_{j=i+1}^{n} a_j < d$ (sum s is too small)

$O(2^n n)$

Backtracking:

ALGORITHM Backtrack (X[1..i])

if X[1..i] is a solution write X[1..i]
else
for each element $x \in S_{i+1}$ Consistent with X[1..i] and the constraints do
X[i+1] ← x
Backtrack (X[1..i+1])

BRANCH - AND - BOUND

terminologies

feasible solution → is a point in the problem's search space that satisfies all the problem's constraints

optimal Solution → is a feasible solution with the best value of the objective function

→ Compared to backtracking, branch-and-bound requires two additional items

* a way to provide, for every node of a state-space tree, a bound on the best value of the objective function on any solution that can be obtained by adding further components to the partially constructed solution represented by the node.
* the value of the best solution seen so far.

→ Terminate a search path at the current node in a state-space tree of a branch-and-bound algorithm for any one of the following three reasons:

* The value of the node's bound is not better than the value of the best solution seen so far.
* The node represents no feasible solutions because the constraints of the problem are already violated.
* The subset of feasible solutions represented by the node consists of a single point.

ASSIGNMENT PROBLEM

Defn:

* The problem of assigning n people to n jobs so that the total cost of the assignment is as small as possible.

→ An instance of the assignment problem is specified by an $n \times n$ cost matrix C

* Select one element in each row of the matrix so that no two selected elements are in the same Column and their sum is the smallest possible.

Branch-and-bound technique:

Ex:

$$C = \begin{array}{c} \\ \\ \\ \\ \\ \end{array}\begin{array}{cccc} \text{Job1} & \text{Job2} & \text{Job3} & \text{Job 4} \\ \end{array}$$

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \begin{array}{l} \text{person a} \\ \text{person b} \\ \text{person c} \\ \text{person d} \end{array}$$

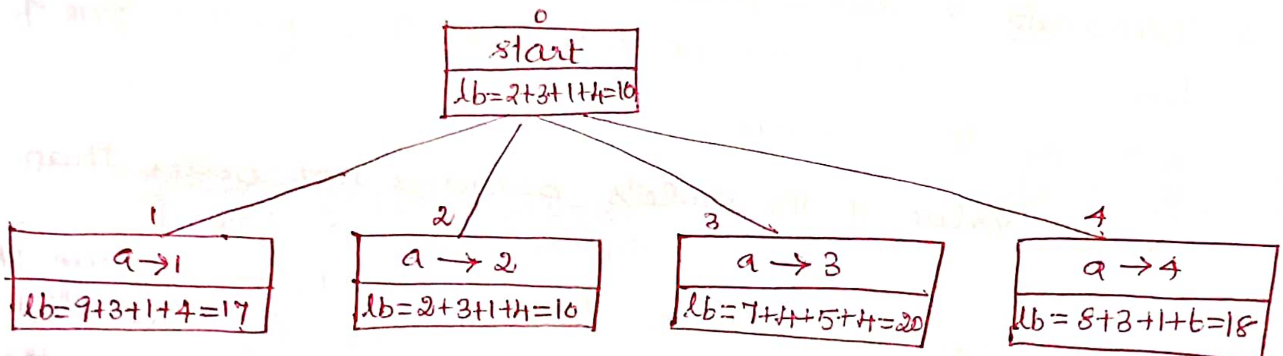
Solution:

Step 1

* Find a lower bound on the cost of an optimal selection.
  - sum of the smallest elements in each of the matrix's rows.

lower bound $\boxed{lb = 2+3+1+4 = 10}$

Step 2: Levels 0 and 1 of the state-space tree



- start with the root that corresponds to no elements selected from the cost matrix.
- the lower-bound value for the root, lb is 10

→ The nodes on the first level of the tree correspond to selections of an element in the first row of the matrix
  ie) a job for person a.

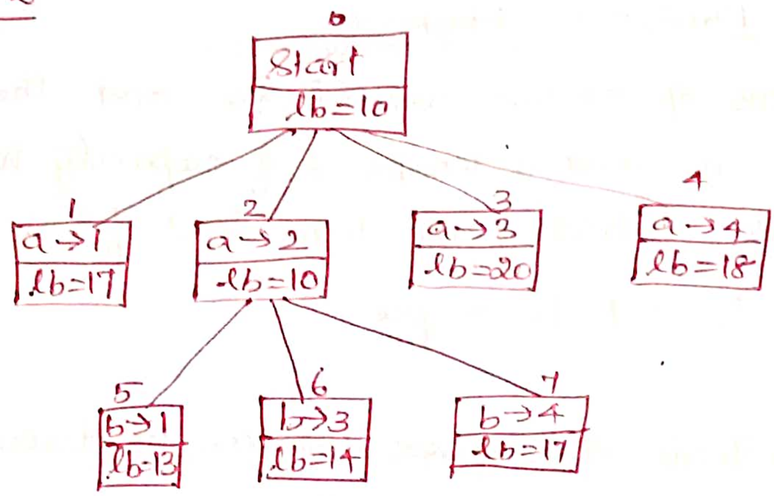
→ four live leaves.

- The most promising node is 2 because it has the smallest lower bound value.

Step 3:

- branch out from that node first by considering the three different ways of selecting an element from the second row and not in the second column
- three different jobs that can be assigned to person b.

Levels 0, 1, 2



Node 5

2+6+1+4 = 13

Node 6

2+3+5+4 = 14

Node 7

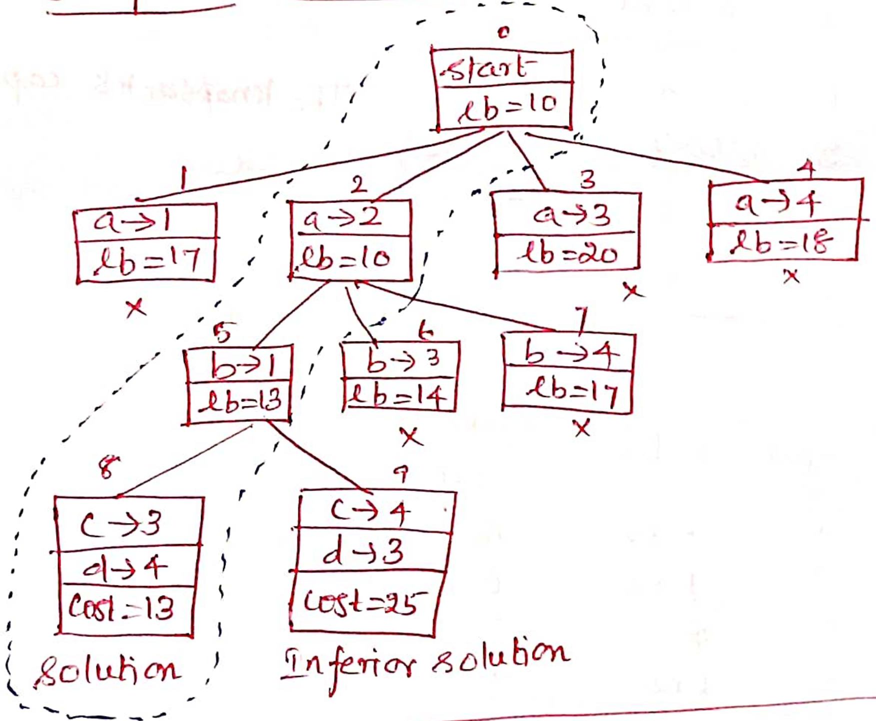
2+7+1+7 = 17

Step 4: node 5.

- choose the one with the smallest lower bound, node 5.

* selecting the third column's element from c's row.
  ie) person c – Job 3

Complete State-space tree:



Node 8

2+6+1+4 = 13

Node 9

2+6+8+9 = 25

leaf 8: feasible solution: { a→2, b→1, c→3, d→4 }
total cost = 13

node 9: feasible solution: { a→2, b→1, c→4, d→3 }
total cost = 25.
→ terminated.

* leaf 8 → optimal solution
  ie) Person a is assigned with Job 2
  Person b is assigned with Job 1
  Person c is assigned with Job 3

cost = 2+6+1+4 = 13

KNAPSACK PROBLEM

* Given n items of known weights w_i and the values v_i, $i = 1, 2, \ldots, n$ and a knapsack of capacity W, find the most valuable subset of the items that fit in the knapsack.

Branch-and-Bound Technique.

Step 1:

- order the items of a given instance in descending order by their value-to-weight ratios.
* first item gives the best pay off per weight unit
  last one gives the worst pay off per weight unit

$$v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n$$

Example: Problem:-

item	Weight	Value
1	4	$40
2	7	$42
3	5	$25
4	3	$12

The knapsack's capacity W is 10.

Step 1:

item	weight	value	$\frac{\text{Value}}{\text{weight}}$
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

Step 2:

State space tree

* Each node on the ith level → represents all the subsets of n items that include a particular selection made from the first i ordered items.

- particular selection is determined by the path from the root to the node.

→ left branch – inclusion of the next item
→ right branch – exclusion of the next item

node → total weight w,
total value v,
upper bound ub

computing upper bound:

$$\boxed{ub = v + (W - w)(v_{i+1}/w_{i+1})}$$

v – total value of the items selected

W−w → remaining capacity of the knapsack.

v_{i+1}/w_{i+1} → best per unit payoff among the remaining items.

0

w=0, v=0
ub=100

node 0 : Root.

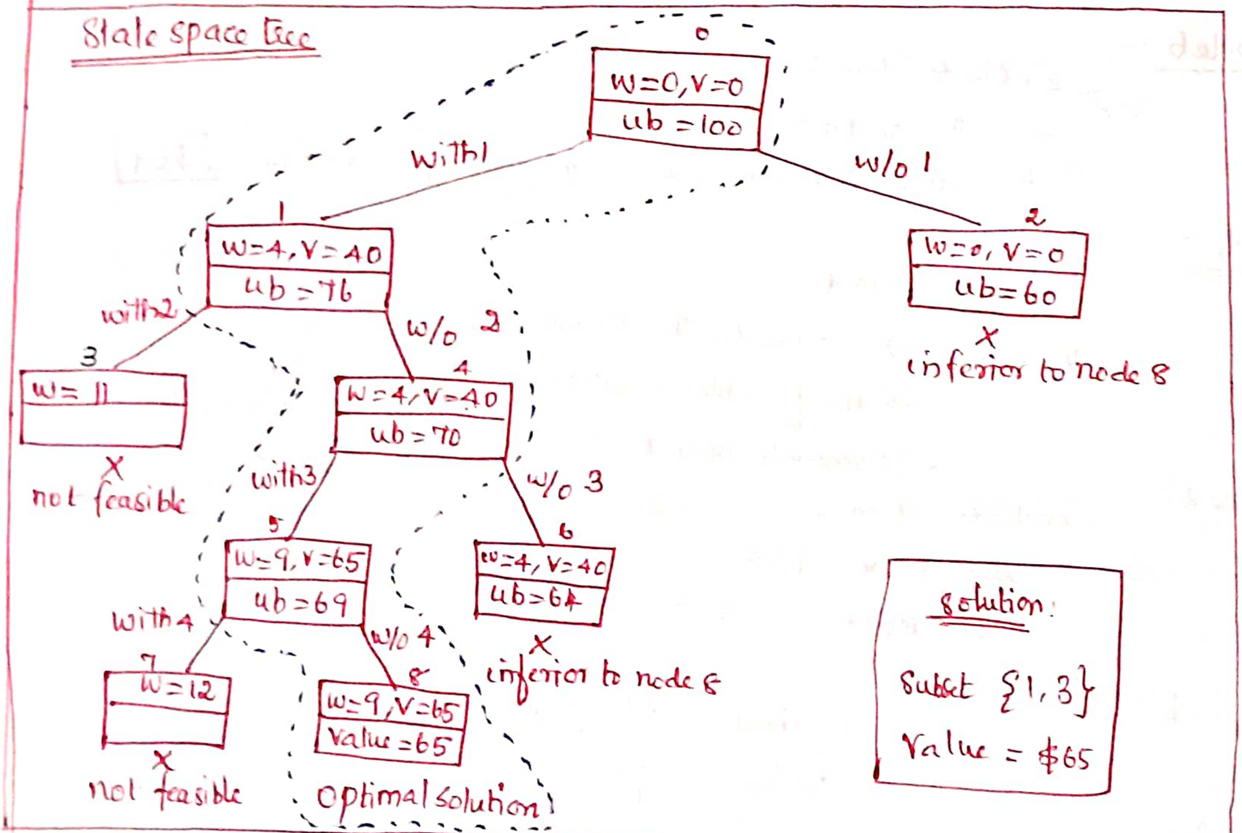
→ no items have been selected as yet.

→ total weight = 0
total value = 0

$$ub = v + (W-w)(v_{i+1}/w_{i+1})$$
$$= 0 + (10-0)(10)$$

$$\boxed{ub = 100}$$

State space tree



node 1:

left child → subsets that include item 1.
total weight → 4, total value → 40

upper bound ub = $40 + (10-4) * 6 = \boxed{\$76}$

Node 2:

- subsets that do not include item 1

$w = 0, v = \$0$

$ub = 0 + (10-0) * 6 = \boxed{\$60}$

* Node 1 has a larger upper bound than node 2.
- So Node 1 is a promising node., branch from node 1

Node 3:

- include item 2

$w = 11 \rightarrow$ exceeds the Knapsack's capacity

$\rightarrow$ terminate node 3.

Node 4:

- without item 2.

$w = 4, v = \$40$

$ub = 40 + (10-4) * 5 = \boxed{\$70}$

- promising node.

Node 5:

- include item 3

$w = 9, v = \$65$

$ub = 65 + (10-9) * (4) = 65 + 4 = \boxed{\$69}$

Node 6:

- exclude item 3

$w = 4, v = \$40$

$ub = 40 + (10-4) * 4 = 40 + (6 * 4) = 40 + 24 = \boxed{\$64}$

Node 7:

- include item 4

$w = 12 \rightarrow$ exceeds the Knapsack's capacity

$\rightarrow$ no feasible solution.

$\rightarrow$ terminate node 7.

Node 8:

- exclude item 4

$w = 9, v = \$65$

$ub = 65 + (10-9) * 0 = \boxed{\$65}$

Step 3:

optimal solution

subset = $\{1, 3\}$
value = $65

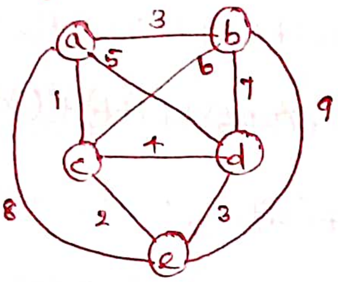
TRAVELING SALESMAN PROBLEM

- Apply branch and bound technique to instances of the traveling salesman problem.
  - lower bound on tour lengths.
    - can be obtained by finding the smallest element in the intercity distance matrix D and multiplying it by the number of cities n.

* For each city, i, $1 \le i \le n$, find the sum s_i of the distances from city i to the two nearest cities; compute the sum s of n numbers, divide the result by 2.
  - if all the distances are integers, round up the result to the nearest integer:

$$\boxed{lb = \lceil s/2 \rceil} \qquad lb = \sum_{v \in V} (\text{sum of costs of the two least cost edges adjacent to } v) / 2$$

Example: Weighted graph



$$lb = \lceil [\overset{a}{(1+3)} + \overset{b}{(3+6)} + \overset{c}{(1+2)} + \overset{d}{(3+4)} + \overset{e}{(2+3)}]/2 \rceil = 14$$

→ lower bound by summing up the lengths of the two shortest edges incident with each of the vertices, with the required inclusion of edges.

Branch and bound technique:
- find the shortest Hamiltonian circuit of the graph

* **State - space Tree:**
  i) - tours - start at a.
  ii) - Graph is undirected, b is visited before c. X.
  → After visiting $n-1 = 4$ cities, a tour has no choice but to visit the remaining unvisited city and returned to the starting one.

node 0:
- starting vertex a.

$lb = 14$

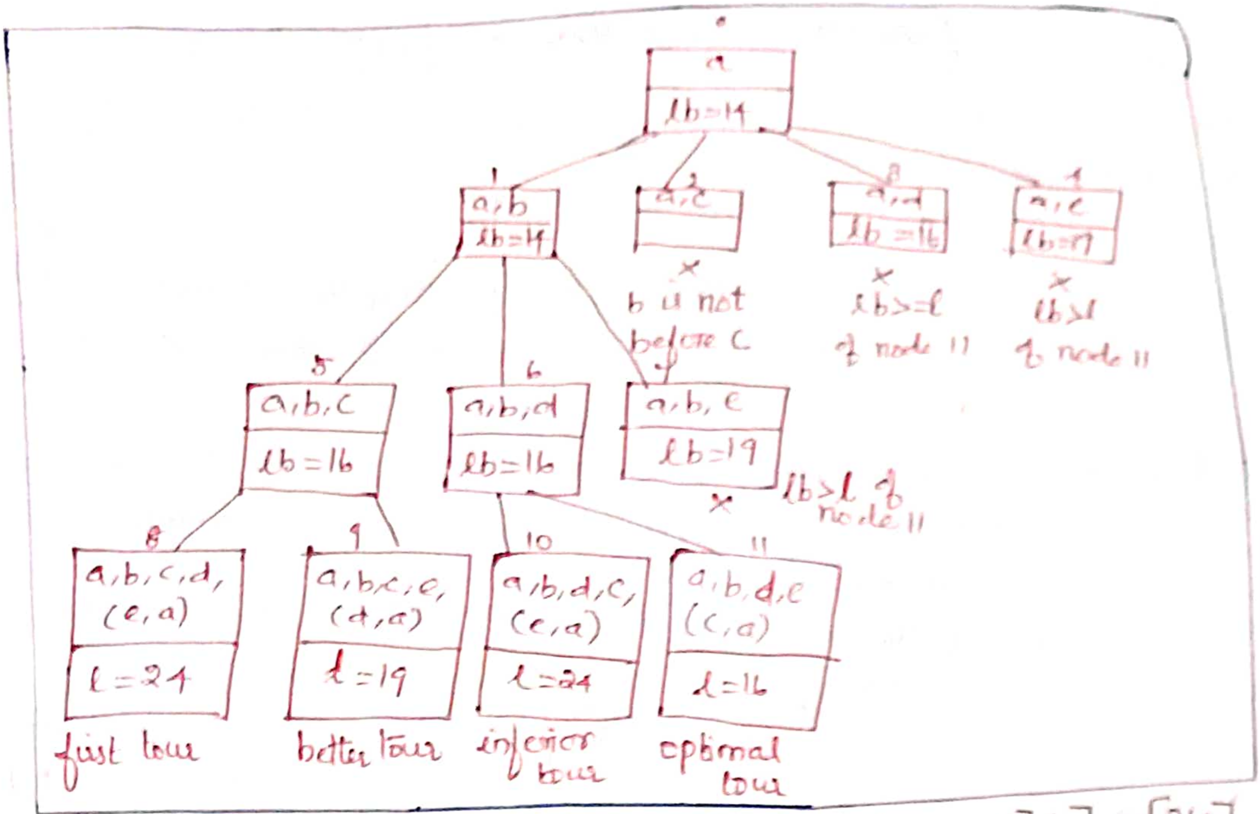
node 1: a, b

$lb = 14$.

Node 2:

a, c

→ b is not before c.



node 3: a,d: $lb = \lceil [(1+5)+(3+6)+(1+2)+(3+5)+(2+3)]/2 \rceil = \lceil 31/2 \rceil = 16$

node 4: a,e: $lb = \lceil [(1+8)+(3+6)+(1+2)+(3+4)+(2+8)]/2 \rceil = \lceil 38/2 \rceil = 19$

node 5: a,b,c: $lb = \lceil [(1+3)+(3+6)+(1+6)+(3+4)+(2+3)]/2 \rceil = \lceil 32/2 \rceil = 16$

node 6: a,b,d: $lb = \lceil [(1+3)+(3+7)+(1+2)+(3+7)+(2+3)]/2 \rceil = \lceil 32/2 \rceil = 16$

node 7: a,b,e: $lb = \lceil [(1+3)+(3+9)+(1+2)+(3+4)+(2+9)]/2 \rceil = \lceil 37/2 \rceil = 19$

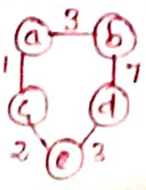
node 8:
a,b,c,d: (e,a)
$lb = \lceil [(3+8)+(3+6)+(4+6)+(3+4)+(3+8)]/2 \rceil = \lceil 48/2 \rceil = 24$

node 9: a,b,c,e: (d,a)
$lb = \lceil [(3+5)+(3+6)+(2+6)+(3+5)+(2+3)]/2 \rceil = \lceil 38/2 \rceil = 19$

node 10: a,b,d,c: (e,a)
$lb = \lceil [(3+8)+(3+7)+(4+2)+(7+4)+(2+8)]/2 \rceil = \lceil 48/2 \rceil = 24$

node 11: a,b,d,e: (c,a)
$lb = \lceil [(1+3)+(3+7)+(1+2)+(7+3)+(3+2)]/2 \rceil = \lceil 32/2 \rceil = 16$

optimal solution: $a \to b \to d \to e \to c \to a$



optimal tour = 3+7+3+2+1 = 16

APPROXIMATION ALGORITHMS FOR NP-HARD PROBLEMS

NP-hard Problems → optimization problems
- at least as hard as NP-complete problems

* An algorithm whose output is an approximation of the actual optimal solution. i.e) O/P is an approximation of actual soln.

① **Accuracy ratio:** – minimization problem. S_a - approximate soln. S^* - exact soln.

→ accuracy of the approximate soln is measured by Accuracy ratio.

$$\boxed{r(S_a) = \frac{f(S_a)}{f(S^*)}}$$

S^* - exact solution to the problem.

→ approximate solutions to maximization problems.

$$\boxed{r(S_a) = \frac{f(S^*)}{f(S_a)}}$$

$r(S_a) = 1$ → better the approximate solution

Definition: ②

* A polynomial - time approximation algorithm is said to be a c - approximation algorithm, where $c \geq 1$, if the accuracy ratio of the approximation it produces does not exceed c for any instance of the problem:

③ **Performance ratio:**

$$r(S_a) \leq c$$

- The best value of c → holds for all instances of the problem is called the performance ratio of the algorithm
- denoted as R_A. ↳ quality of the approximation alg

→ best upper bound of possible $r(S_a)$ values taken over all instances of the probm

* Traveling Salesman Problem:

Theorem 1: If $P \neq NP$, there exists no c-approximation algorithm for the traveling salesman problem. i.e) there exists no polynomial - time approximation algorithm for the problem so that for all instances

$$f(S_a) \leq c\, f(S^*)$$

for some constant c.

Proof:

Let G → graph with n vertices

* map G to a complete weighted Graph G' by assigning weight 1 to

each edge in G and adding an edge of weight cn+1 between each pair of vertices not adjacent in G.

* If G has a Hamiltonian circuit, its length in G' is n
- hence it is the exact solution s^* to the traveling salesman problem for G'
* if S_a is an approximate solution obtained for G' by algorithm A, then $f(S_a) \leq cn$
- G does not have Hamiltonian circuit

$$f(S_a) \geq f(s^*) > cn$$

* Hamiltonian circuit problem is NP-complete.

* Greedy Algorithms for the TSP:

approximation algorithm - based on greedy technique.

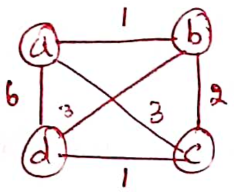
i) Nearest-neighbor algorithm:

Step 1: Choose a city as the start.

Step 2: Repeat the following operation until all the cities have been visited:
- go to the unvisited city nearest the one visited last.

Step 3: Return to the starting city.

Example: 1



* a → starting vertex.
* Nearest-neighbor algorithm yields the tour.

S_a : a - b - c - d - a of length 10

optimal solution:

s^* : a - b - d - c - a of length 8.

Accuracy ratio: $r(S_a) = \frac{f(S_a)}{f(s^*)} = \frac{10}{8} = 1.25$

tour S_a is 25% longer than the optimal tour s^*.

ii) Multifragment-heuristic algorithm:

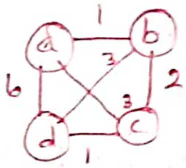
- the problem of finding a minimum-weight collection of edges in a given complete weighted graph so that all the vertices have degree 2.

Step 1: Sort the edges in increasing order of their weights.
Initialize the set to be empty set.

Step 2: Repeat the step n times, where n is the number of cities in the instance being solved.
- add the next edge on the sorted edge list to the set of tour edges, provided the addition does not create a vertex of degree 3 or a cycle of length less than n; otherwise, skip the edge.

Step 3: Return the set of tour edges.

Example:



- Apply the algorithm
- {(a,b), (c,d), (b,c), (a,d)}
- better tour than nearest-neighbor algorithm

* The instances in which intercity distances satisfy the natural conditions:
→ triangle inequality: $d[i,j] \le d[i,k] + d[k,j]$
→ Symmetry: $d[i,j] = d[j,i]$

→ better tour than Nearest-neighbor
→ performance unbounded

Accuracy ratios: $n \ge 2$ cities

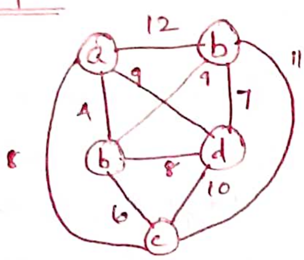
$$\frac{f(s_a)}{f(s^*)} \le \frac{1}{2}\left(\lceil \log_2 n \rceil + 1\right)$$

$f(s_a)$ → length of heuristic tour
$f(s^*)$ → length of shortest tour.

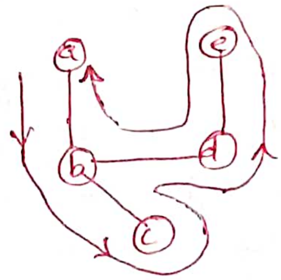
Minimum-Spanning-Tree-Based Algorithms:
- approximation algorithm.
→ exploit a connection between Hamiltonian circuits & spanning tree.
- Removing an edge from Hamiltonian circuit yields a spanning tree.

(i) Twice-around-the-tree algorithm:
Step 1: Construct a minimum spanning tree
Step 2: Starting at a vertex, perform a walk around the MST recording all the vertices passed by.
Step 3: Scan the vertex list obtained in step 2 and eliminate from it all repeated occurrences of the same vertex except the starting one at the end of the list

Example



Minimum Spanning tree:
(a,b), (b,c), (b,d), (d,e)



→ twice around the tree walk that starts and ends at a is

a, b, c, b, d, e, d, b, a.

* Eliminating the second b, the second d, and the third b yields the Hamiltonian circuit

a, b, c, d, e, a of length 39.

→ not optimal.

Theorem 2:

The twice-around-the-tree algorithm is a 2-approximation algorithm for the traveling salesman problem with Euclidean distances.

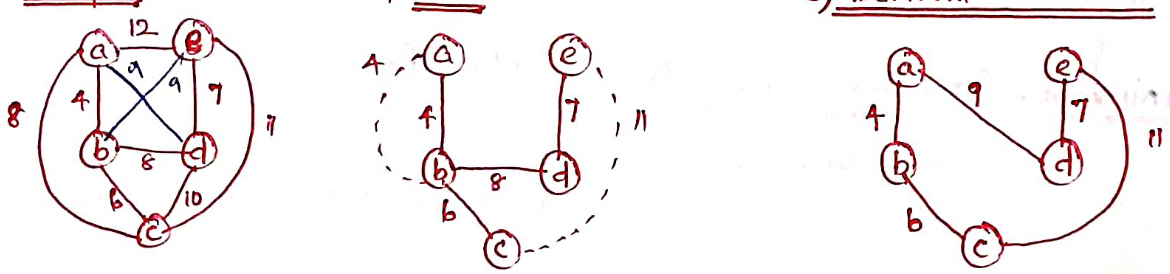
(iv) Christofides Algorithm:

- better performance ratio.
- uses minimum spanning tree.
- obtains a multigraph by adding to the graph the edges of a minimum-weight matching of all the odd-degree vertices in its minimum spanning tree.
- finds an Eulerian circuit in the multigraph and transforms it into a Hamiltonian circuit by shortcuts.

Example:

Application of the christofides algorithm.

a) Graph: b) MST c) Hamiltonian circuit



* The traversal of the multigraph, starting at vertex a, produces the Eulerian circuit a-b-c-e-d-b-a, which, after one shortcut, yields the tour a-b-c-e-d-a of length 37.

* Performance ratio: 1.5

Eulerian circuit → uses every edge of the graph exactly once.

Degree → number of edges connecting it

Even degree → even number of edges.

(iv) Local Search Heuristics:

- optimal tours can be obtained by iterative - improvement algorithms, which are called local Search heuristics.

best-known algorithms

→ 2 - opt

→ 3-opt

→ Lin - kernighan. → best algm to obtain high-quality approximations of optimal tours.

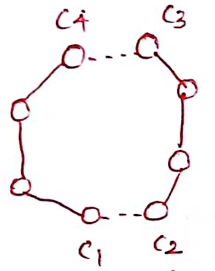
* Start with some initial tour
* on each iteration, the algorithm explores a neighborhood around the current tour by replacing a few edges in the current tour by other edges.
* If the changes produce a shorter tour, the algorithm makes it the current tour and continues by exploring its neighborhood.
* otherwise, the current tour is returned as the algorithm's output and the algorithm stops.

(v) 2-opt algorithm

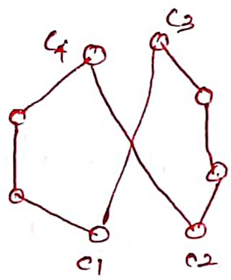
→ works by deleting a pair of nonadjacent edges in a tour and reconnecting their endpoints by the different pair of edges to obtain another tour.

- This operation is called 2-change.

Ex:



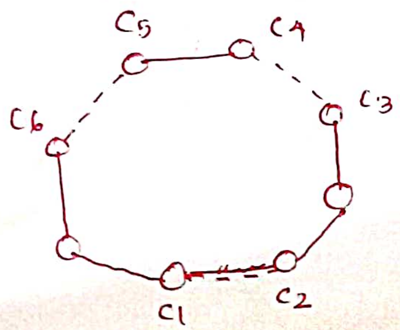
original tour.



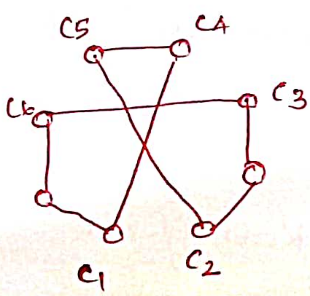
New tour - 2change

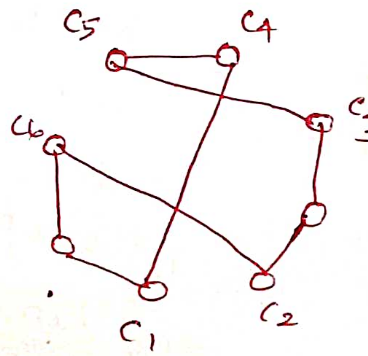
(vi) 3-change:-

Original tour



New tours





Average tour quality and running times:

Heuristic	% Excess over the Held-Karp bound	Running Time (seconds)
nearest neighbor	24.79	0.28
multifragment	16.42	0.20
christofides	9.81	1.04
2-opt	4.70	1.41
3-opt	2.88	1.50
Lin-Kernighan	2.00	2.06

Held-Karp bound → lower bound on the length of the shortest tour
— very close to the length of an optimal tour.
→ HK(s*)

I Greedy algm
  i) Nearest Neighbor algm
  ii) Multifragment - heuristic algm

II Minimum Spanning Tree-based algm
  i) Twice Around the tree algm
  ii) christofides algm

III Local Search heuristics
  i) 2-opt
  ii) 3-opt
  iii) Lin - Kernighan.

APPROXIMATION ALGORITHMS FOR THE KNAPSACK PROBLEM

Knapsack problem → NP-hard problem.

* Given n items of known weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ and a Knapsack of weight capacity W, find the most valuable sub-set of the items that fits into the Knapsack.

Approximation Algorithms: Greedy Algorithms for the knapsack Problem:

* To select the items in decreasing order of their weights
  - heavier items may not be the most valuable in the set.
  - if we pick up the items in decreasing order of their value, there is no guarantee that the knapsack's capacity will be used efficiently.

Greedy strategy → computing the value-to-weight ratios v_i / w_i, $i = 1, 2, \ldots, n$

→ selecting the items in decreasing order of the ratios.

(i) Greedy algorithm for the discrete knapsack problem:

Step 1: Compute the value-to-weight ratios $r_i = v_i / w_i$, $i = 1, \ldots, n$ for the items given

Step 2: Sort the items in nonincreasing order of the ratios computed.

Step 3: Repeat the following operation until no item is left in the sorted list:

- if the current item on the list fits into the knapsack, place it in the knapsack and proceed to the next item;
- otherwise, proceed to the next item.

Ex:

Instance of the knapsack problem:

Knapsack capacity = 10.

item	weight	value
1	7	\$42
2	3	\$12
3	4	\$40
4	5	\$25

→ Computing the value-to-weight ratios and sorting the items in nonincreasing order of the ratios:

item	weight	value	Value/weight
3	4	\$40	10
1	7	\$42	6
4	5	\$25	5
2	3	\$12	4

→ Select the first item of weight 4, skip the next item of weight 7, Select the next item of weight 5, and skip the last item of weight 3

item 3 – selected
1 – not selected
4 – selected
2 – not selected

i) $4+7 > 10$

ii) $4+5 = 9$

solution: {item 3, item 4}

Profit = \$40 + \$25 = $\boxed{\$65}$

→ Performance ratio = 2

(ii) Greedy Algorithm for the Continuous Knapsack problem

Step 1: Compute the value-to-weight ratios v_i/w_i, $i = 1, \ldots, n$ for the items given

Step 2: Sort the items in nonincreasing order of the ratios computed.

Step 3: Repeat the following operation until the knapsack is filled to its full capacity or no item is left in the sorted list.

- if the current item on the list fits into the knapsack in its entirety, take it and proceed to the next item;
- otherwise, take its largest fraction to fill the knapsack to its full capacity and stop.

ex:

① Compute the value-to-weight ratios

② Sort the items in nonincreasing order of the ratios

item	weight	value	Value/weight
3	4	\$40	10
1	7	\$42	6
4	5	\$25	5
2	3	\$12	4

– Take the first item of weight 4.

– Then 6/7 of the next item on the sorted list to fill the knapsack to its full capacity.

Profit = $\$40 + \frac{6}{7} \times 42 = \$40 + \$36 = \boxed{\$76}$

* Approximation schemes: — Polynomial time approximation scheme. (k)
- to get approximations S_a with any predefined accuracy level:

$$\frac{f(s^*)}{f(s_a^{(k)})} \leq 1 + 1/k \text{ for any instance of size } n.$$

i) First Approximation scheme:

— suggested by S. Sahni in 1975.

* Algorithm:
  - generates all subsets of k items or less and for each one that fits into the knapsack it adds the remaining items.
  - the subsets of the highest value obtained is returned as the algorithm's output.

Ex: a) Instance

item	weight	value	value/weight
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	1	\$4	4

W = 10

k=2 b) Subsets generated by the algorithm

Subset	added items	value
ϕ	1, 3, 4	\$69
{1}	3, 4	\$69
{2}	4	\$46
{3}	1, 4	\$69
{4}	1, 3	\$69
{1,2}	not feasible	
{1,3}	4	\$69
{1,4}	3	\$69
{2,3}	not feasible	
{2,4}		\$46
{3,4}	1	\$69

{1, 3, 4} → optimal solution.

Analysis:

- Determine the subset → O(n) time
- algorithm's efficiency → $O(kn^{k+1})$.
- Time efficiency of Sahni's scheme — exponential in k.

Knapsack Problem using Branch and Bound Algorithm.

NOV/Dec-2016

Problem:

① Solve the following instance of knapsack problem by branch and bound algorithm.

Item	weight	profit
1	5	$ 40
2	7	$ 35
3	2	$ 18
4	4	$ 4
5	5	$ 10
6	1	$ 2

$W = 15$

Solution:

Step 1: compute profit / weight ratios

Step 2: Sort the items in descending order of their profit / weight ratios.

Item	weight	profit	$\frac{\text{Profit}}{\text{weight}}$
3	2	18	9
1	5	40	8
2	7	35	5
5	5	10	2
6	1	2	2
4	4	4	1

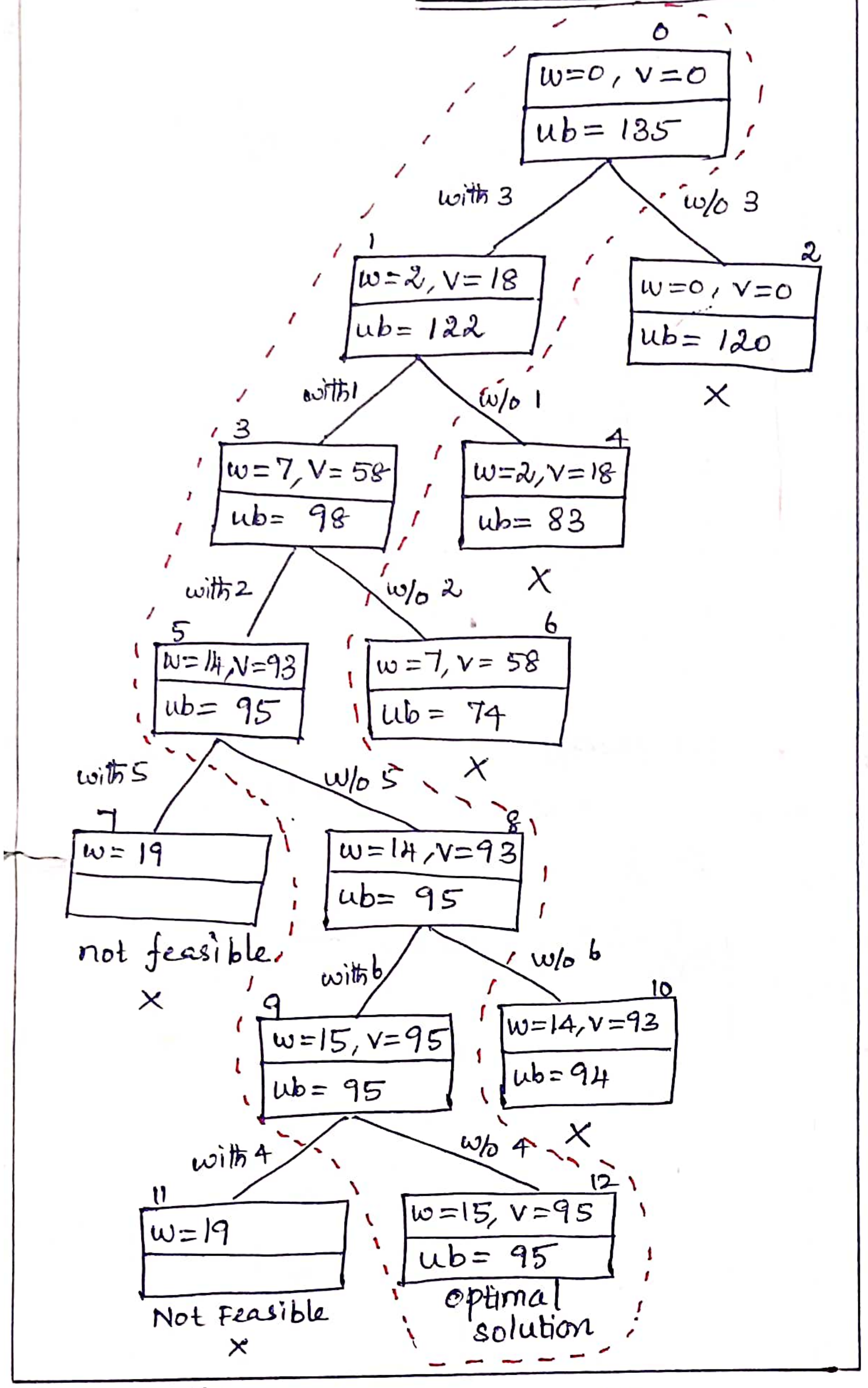
Step 3: Draw state space tree

Step 4: Find optimal solution.

* **Finding upper bound:**

$$\boxed{ub = v + (W - w)\left(V_{i+1}/w_{i+1}\right)}$$

State space Tree



node 0:

$ub = 0 + (15-0) \cdot 9$
$= 15(9) =$

node 1

$ub = 18 + (15-2) \cdot 8$
$= 18 + 13(8)$

node 2

$ub = 0 + (15-0) \cdot 8$
$= 15(8) = 120$

node 3

$ub = 58 + (15-7) \cdot (5)$
$= 58 + (8)(5)$
$= 58 + 40 = 98$

node 4

$ub = 18 + (15-2) \cdot (5)$
$= 18 + (13) \cdot 5$

node 5

$ub = 93 + (15-14) \cdot 2$
$= 93 + 2 = 95$

node 6

$ub = 58 + (15-7) \cdot 2$
$= 58 + 8(2)$
$= 58 + 16 = 74$

node 8:

$ub = 93 + (15-14) \cdot 2$
$= 93 + 2 = 95$

node 9:

$ub = 95 + (15-15) \cdot 1 = 95$

node 10:

$ub = 93 + (15-14) \cdot 1 = 93 + 1$

node 12:

$ub = 95 + (15-15) \cdot 0 = 95$

→ **Answer:**

Subset items = $\{1, 2, 3, 6\}$
Maximum Profit = \$95