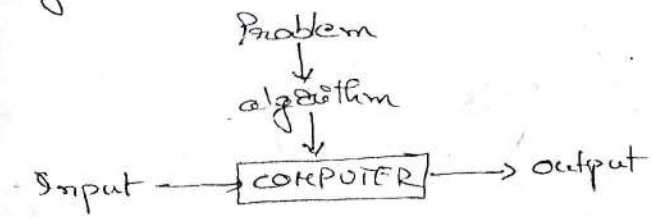
1. Jyothi Priyanka

INTRODUCTION

1. What is an Algorithm?

An algorithm is a sequence of unambiguous instructions for solving a problem. i.e, for obtaining a requested output for any legitimate input in a finite amount of time. This algorithm can be illustrated by a simple diagram in



Important points

1. The nonambiguity requirement for each step of an algorithm can't be compromised.
2. The range of input for which an algorithm works has to be specified carefully.
3. The same algorithm can be represented in several different ways.
4. Several algorithms for solving the same problem may exist
5. Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds.

Greatest Common Divisor of Two Integers

Three methods for solving the problem.
gcd(m, n) is defined as the largest integer that devides both m and n evenly, i.e with a remainder of zero.

Method 1: Euclid's Algorithm

Step1: If n=0, return the value of m as the answer and stop; otherwise proceed to step2.

Step2: Divide m by n and assign the value of the remainder to r.

Step3: Assign the value of n to m and the value of r to n. Go to Step 1.

Method 2: Consecutive Integer Checking Algorithm

S1: Assign the value of min[m, n] to t.

S2: Divide m by t. If the remainder of this divison is 0, goto step 3; otherwise goto step 4.

S3: 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.

S4: Decrease the value of t by 1. Go to step 2.

Method 3: Middle-School Procedure

S1: Find the prime factors of m.

S2: Find the prime factors of n

S3: Identify all the common factors in the two prime expansions found in step 1 and step 2.

S4: Compute the product of all the common factors and return it as the greatest common divisor of the numbers given.

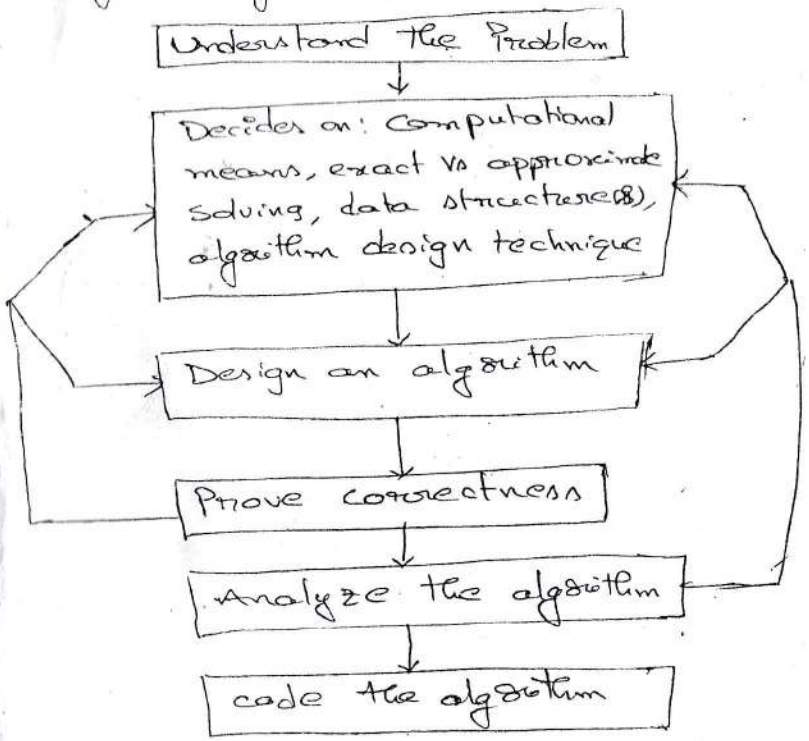
Pseudocode for Euclid's Algorithm

```
// Compute 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.
```

2. Fundamentals of Algorithmic Problem Solving

Steps for Design and analyzing an algorithm

1. Understanding the problem
2. Ascertaining the Capabilities of Computational Device
3. choosing between Exact and Approximate problem solving
4. Deciding on Appropriate Data Structures
5. Algorithm Design Techniques
6. Methods of Specifying an Algorithm
7. Proving Algorithm's correctness
8. Analyzing an Algorithm
9. Coding an Algorithm



1. Understanding the Problem

- The first thing you need to do before designi an algorithm is to understand completely the problem given.
- Read the problem's description carefully an ask questions if you have any doubts about the problem, do a few small examples by hand, think about special cases, and ask questions again if needed.
- An input to an algorithm specifies an instance of the problem the algorithm solves.
- It is very important to specify exactly the range of instances the algorithm needs to hand

2 Ascertaining the Capabilities of Computationa Device:

- The essence of this architecture is captured by the so-called random-access machine(RAM
- Its central assumption is that instructions are executed one after another, one operation at a time.
- Accordingly, algorithms designed to be executed on such machines are called Sequential algorithms
- The central assumption of the RAM model does not hold for some newer computers that can execute operations concurrently ie in parallel.

- Algorithms that take advantage of this capability are called parallel algorithms.

3. Choosing between Exact and Approximate Problem Solving

- The next principal decision is to choose between solving the problem exactly or solving it approximately.
- An algorithm solves the problem exactly is called Exact Algorithm.
- An algorithm solves the problem approximately is called Approximation algorithm.

Why would one opt for an approximation algorithm?

1. There are important problems that simply can't be solved exactly for most of their instance.
   ex square roots, solving nonlinear equations, evaluating definite integrals.
2. Available algorithms for solving a problem exactly can be unacceptably slow because of the problem's intrinsic complexity.
3. An approximation algorithm can be a part of a more sophisticated algorithm that solves a problem exactly.

4 Deciding on Appropriate Data Structures

- Some algorithms do not demand any ingenuity in representing their inputs.
- But others are, in fact, predicated on ingenious data structures.
- Many years ago, an influential textbook proclaimed (declared) the fundamental importance of both algorithms and data structures for computer programming by its very title Algorithms + Data Structures = Programs.
- In object oriented programming, data structures remain crucially important for both design and analysis of algorithms.

5. Algorithm Design Techniques.

An algorithm design technique is a general approach to solving a problems algorithmic that is applicable to a variety of problem from different areas of computing.

1. Algorithms provide guidance for designing algorithms for new problems. i.e problem for which there is no known satisfactory algorithm.
2. Algorithms are the cornerstone of computer science.

Algorithm design techniques make it possible to classify algorithms according to an underlying design idea, therefore they can serve as a natural way to both categorize and study algorithms.

6. Methods of Specifying an Algorithm.

Pseudocode

- A pseudocode is a mixture of a natural language and programming language like constructs.
- It is usually more precise than a natural language, and its usage often yields more succint algorithm descriptions.

Flowchart

A method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps.

7. Proving an Algorithm's Correctness.

- Once an algorithm has been specified, go to prove its correctness. i.e to prove that an algorithm yields a required result for every legitimate input in a finite amount of time.
- A common technique for proving correctness is to use the mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs.
- But in order to show that an algorithm is incorrect, you need just one instance of its input for which the algorithm fails.
- If the algorithm is found to be incorrect, either redesign it under the same decision regarding the data structures, the design technique, and so on, or, in a more dramatic reversal, to reconsider one or more of those decisions.

8. Analyzing an algorithm.

- After correctness, the most important is efficiency.
- There are two kinds of algorithm efficiency
  1. Time efficiency 2. Space efficiency
- Time efficiency indicates how fast the algorithm runs.
- Space efficiency indicates how much extra memory the algorithm needs.
- Another desirable characteristic of an algorithm is simplicity
- Simplicity is an important algorithm characteristic because simpler algorithms are easier to understand and easier to program, consequently, the resulting programs usually contain fewer bugs.

- Another desirable characteristic of an algorithm is generality
- Generality of the problem the algorithm solves and the range of inputs it accepts.

9. Coding an Algorithm.

- Most algorithms are destined to be ultimately implemented as computer programs.
- Programming an algorithm presents both a peril and an opportunity.
- The Peril lies in the possibility of making the transition from an algorithm to a program either incorrectly or very inefficiently.

Problems.

1. Design an algorithm to compute the area and circumference of a circle.
2. Write a Pseudocode for an algorithm for finding real roots of equations $ax^2+bx+c=0$ for arbitrary real coefficients a, b and c.
3. Design an algorithm to find the prime number.

3. Important Problem Types.

Most important problem types are

1. Sorting 2. Searching 3. String processi
4. Graph Problems 5. Combinatorial proble
6. Geometric problems 7. Numerical probl

1. Sorting

- In sorting problem to rearrange the items of given list in ascending order.
- We usually need to sort list of numbers, chara from an alphabet, character strings, libraries about their holdings, records in schools abou their students, and companies about their employees.

ex Dictionaries, telephone books, class list etc

- Sorting is used as an auxiliary step in several important algorithms in other area e.g. geometric algorithms.
- There are a few good sorting algorithms that sort an arbitrary array of size n using about $n \log_2 n$ comparisons.
- Two properties of sorting algorithms.
  1. Stable. 2. In place.
- An algorithm is called stable if it preserve the relative order of any two equal elements in its input. ex Insertion sort, Merge sort
- An algorithm is said to be in place if it doe not require extra memory, except, possib for a few memory units. ex Insertion, bubble, heap

2. Searching

- The searching problem deals with finding a given value, called a search key, in a given set.
- There are plenty of searching algorithms to choose from.
- They range from the straightforward sequential search to a spectacularly efficient but limited binary search and algorithms based on representing the underlying set in a different form more conducive to searching.
- For searching, there is no single algorithm that fits all situations best.
- Some algorithms work faster than others but require more memory; some are very fast but applicable only to sorted arrays and so on.

3. String Processing

- A string is a sequence of characters from an alphabet.
- Text strings compromise letters, numbers, and special characters.
- Bit strings compromise zeros and ones
- Gene sequences, which can be modeled by strings of characters from the four-character alphabet [A, C, G, T].
- String-processing algorithms have been impor for computer science for a long time in conjunction with computer languages and compiling issues.
- String matching searching for a given word in a text.

4. Graph Problems

- A Graph is a collection of points called vertices, some of which are connected by line segments called edges.
- Graphs can be used for modeling a wide variety of real-life applications, including transportation and communication networks, project scheduling, and games.
- Basic graph algorithms include graph traversal algorithms, shortest-path algorithms, and topological sorting for graphs with directed edges
- Some graph problems are computationally very hard.

examples

1. Traveling salesman problem (TSP) is the pro of finding the shortest tour through n citi that visits every city exactly once.

2. Graph-coloring problem asks us to assign the smallest number of colors to vertices of a graph so that no two adjacent vertices are the same color.

5. Combinatorial Problems.

- These are problems that ask (explicitly or implicitly) to find a combinatorial object such as permutation, a combination, or a subset that satisfies certain constraints and has some desired property (e.g. maximizes a value or minimizes a cost)

examples.

1. Traveling salesman problem.
2. Graph coloring problem.

6 Geometric Problems.

- It deal with geometric objects such as points, lines, and polygons.
- Two classic problems of computational geometry algorithms.

1. The closest-pair problem. 2. Convex-hull problem

closest-pair problem.

It is self-explanatory, given n points in the plane, find the closest pair among them.

Convex-hull problem.

To find the smallest convex polygon that would include all the points of a given set.

7. Numerical Problems.

- These are the another large special area o applications, are problems that involve mathem objects of continuous nature.
- solving equations and systems of equations computing definite integrals, evaluating functions, and so on.
- The majority of such mathematical problem can be solved only approximately.

1.4. Fundamental Data Structures.

- A data structure can be defined as a particular scheme of organizing related data items.
- The data items can range from elemente data types (e.g integers or characters) to date structures.

1. Linear Data Structures.

- The two most important elementary data structures are the array and the linked list.
- A array is a sequence of n items of the same data type that are stored contiguous in computer memory and made accessible by specifying a value of the array's index

item[0]	item[1]	...	item[n-1]

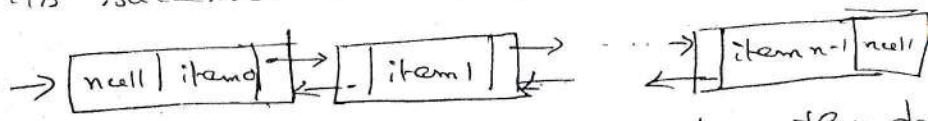
The index is an integer either between 0 and n-1 or between 1 and n.

- Each and every element of an array can be accessed in the same constant amount of time regardless of where in the array the element in question is located.
- String is a sequence of characters from an alphabet terminated by a special character indicating the string's end.
- Strings composed of zeros and ones are called binary strings or bit strings
- operations we usually perform on strings differ from those we typically perform on other arrays.

- A Linked List is a sequence of zero or more elements called nodes each containing two kinds of information, some data and one or more links called pointers to other nodes of the linked list.
- In singly linked list, each node except the last one contains a single pointer to the next element.

- To start a linked list with a special node called the header.
- This node often contains information about the linked list such as its current length, it may also contain, in addition to a pointer to the first element, a pointer to the linked list's last element.

- In doubly linked list, every node, except the first and last, contains pointers to both its successor and its predecessor.



- The basic operations performed on this data structure are searching for, inserting, and deleting an element.

Stack

- A stack is a list in which insertions and deletions can be done only at the end.
- This end is called the top
- when elements are added to a stack and deleted from it, the structure operates in the LIFO fashion.

Queue

- A queue is a list from which elements are deleted from one end of the structure called front, and new elements are added to the other end, called the rear.
- A queue operates in FIFO fashion

Priority Queue

- Many important applications require selection of an item of the highest priority among a dynamically changing set of candidates
- A data structure that seeks to satisfy the needs of such application is called a priority queue.

- The principal operations on a priority queue are finding its largest element, deleting its largest element, and adding a new element.
- A better implementation of a priority queue is based on an ingenious data structure called the heap.

2. Graphs

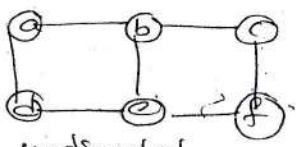
A graph $G = \langle V, E \rangle$ is defined by a pair of two sets: a finite set V of items called vertices and a set E of pairs of these items called edges.

Undirected Graph.

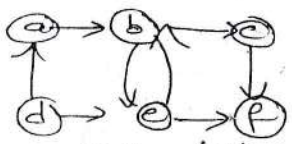
- A graph G is called undirected if every edge in it is undirected.

Directed Graph

- A graph whose every edge is directed is called directed.
- Directed graphs are also called digraphs



undirected



Directed

The graph has six vertices and seven edges.

$V = \{a, b, c, d, e, f\}$

$E = \{(a,b), (b,c), (b,e), (c,f), (d,e), (e,b), (e,f), (d,a)\}$

Complete.

- A graph with every pair of its vertices connected by an edge is called complete
- A standard notation for the complete graph with $|V|$ vertices is $K_{|V|}$.

Dense.

A graph with relatively few possible edges missing is called dense.

Sparse

A graph with few edges relative to the number of its vertices is called sparse.

Graph representations.

Graph can be represented in two way

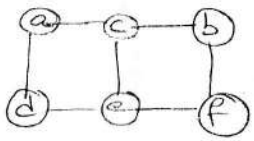
1. The adjacency matrix.
2. The adjacency list.

Adjacency Matrix.

- The adjacency matrix of a graph with n vertices is an n-by-n boolean matrix with one row and one column for each of the graph's vertices, in which the element in the ith row and the jth column is equal to 1 if there is an edge from the ith vertex to the jth vertex, and equal to 0 if there is no such edge.
- Adjacency matrix of an undirected graph is always symmetric. i.e. $A[i, j] = A[j, i]$, for every $0 \le i, j \le n-1$

Adjacency List:

The adjacency lists of a graph is a collection of linked lists, one for each vertex, that contain all vertices adjacent to the list's vertex.



Adjacency Matrix.

$$\begin{array}{c|cccccc} & a & b & c & d & e & f \\ \hline a & 0 & 0 & 1 & 1 & 0 & 0 \\ b & 0 & 0 & 1 & 0 & 0 & 1 \\ c & 1 & 1 & 0 & 0 & 1 & 0 \\ d & 1 & 0 & 0 & 0 & 0 & 0 \\ e & 0 & 0 & 1 & 1 & 0 & 1 \\ f & 0 & 1 & 0 & 0 & 1 & 0 \end{array}$$

adjacency List.

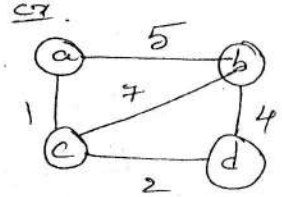
a	→ c → d
b	→ c → f
c	→ a → b → e
d	→ a → e
e	→ c → d → f
f	→ b → e

Weighted Graphs.

- A weighted graph is a graph with numbers assigned to its edges. These numbers are called weights or costs.
- If a weighted graph is represented by its adjacency matrix, then the element A[i, j] will simply contain the weight of the edge from the ith to the jth vertex. If there is such an edge and a special symbol e.g. ∞, if there is no such edge.

Such a matrix is called the weight matrix or cost matrix.

eg.



weighted graph

$$\begin{array}{c|cccc} & a & b & c & d \\ \hline a & \infty & 5 & 1 & \infty \\ b & 5 & \infty & 7 & 4 \\ c & 1 & 7 & \infty & 2 \\ d & \infty & 4 & 2 & \infty \end{array}$$

weighted matrix.

Path and Cycles.

Path: A path from vertex u to vertex v of a graph G can be defined as a sequence of adjacent vertices that starts with u and ends with v.

Simple:
If all vertices of a path are distinct the path is said to be Simple.

length.
The total no of vertices in a vertex sequence defining the path minus one, which is the same as the number of edges in the path.

Directed Path
It is a sequence of vertices in which every consecutive pair of the vertices is connected by an edge directed from the vertex listed first to the vertex listed next.

Connected Graph
A graph is said to be connected if for every pair of its vertices u and v there is a path from u to v.

Cycle

A cycle is a path of a positive length that starts and ends at the same vertex and does not traverse the same edge more than once.

Acyclic

A graph with no cycles is said to be acyclic.

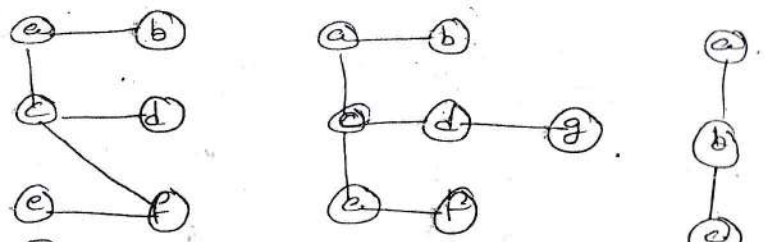
3. Trees

A tree is a connected acyclic graph.

Forest

A graph that has no cycles but is not necessarily connected is called a forest.

ex.



Tree

forest -

Properties of Trees

1. The no of edges in a tree is always one less than the no of its vertices.

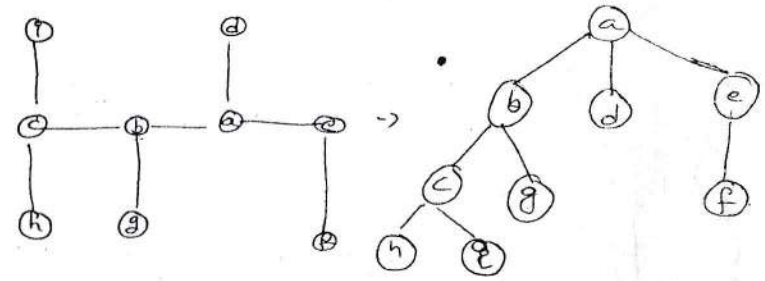
$$|E| = |V| - 1$$

2. For every two vertices in a tree, there always exists exactly one simple path from one of these vertices to the other.

Rooted Tree

To select an arbitrary vertex in a free tree and consider it as the root of the so-called rooted tree.

A rooted tree is usually depicted by placing its root on the top (level 0), the vertex adjacent to the root below it (level 1), the vertices two edges apart from the root below that (level 2), and so on.



Ancestors

For any vertex v in a tree T, all the vertices on the simple path from the root to that vertex are called ancestors.

The vertex itself is usually considered its own ancestor.

Proper Ancestors

The set of ancestors that excludes the vertex itself is referred to as proper ancestors.

If (u, v) is the last edge of the simple path from the root to vertex v ($u \neq v$), u is said to be parent of v and v is called a child of u, vertices that have the same parent are said to be siblings.

Leaf: A vertex with no child is called a leaf.

Parental: A vertex with at least one child is called parental.

Descendants: All the vertices for which a vertex v is an ancestor are said to be descendants of v; the proper descendants exclude the vertex v itself.

Subtree: All the descendants of a vertex v with all the edges connecting them for the subtree of T rooted at that vertex.

Depth:
The depth of a vertex v is the length of the simple path from the root to v.

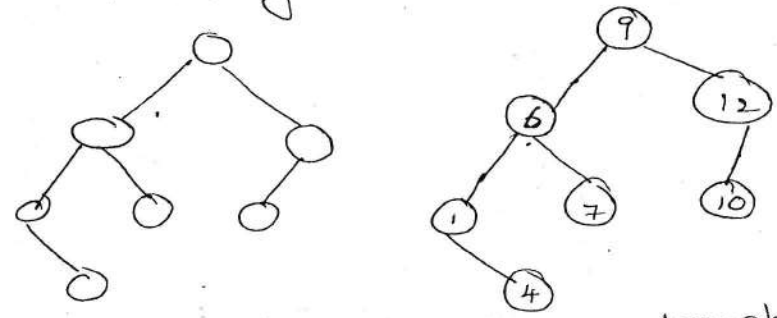
Height
The height of a tree is the length of the longest simple path from the root to a leaf.

Ordered Trees:
An ordered tree is a rooted tree in which all the children of each vertex are ordered

Binary Tree.
A binary tree can be defined as an ordered tree in which every vertex has no more than two children and each child is designated as either a left child or a right child of its parent.

Subtree
The subtree with its root at the left (right) child of a vertex is called left (right) subtree of that vertex.

Binary Search Trees.
A number assigned to each parental vertex is larger than all the numbers in its left subtree and smaller than all the numbers in its right subtree. Such trees are called binary search trees.



Binary Tree. Binary search tree.

4. Sets and Dictionaries.

Sets:
- A set can be described as an unordered collection of distinct items called elements of the set.
- A specific set is defined either by an explicit listing of its elements or by specifying a property that all the set's elements and only they must satisfy.
- The most important set operations are checking membership of a given item in a given set, union and intersection of two sets.

- Sets can be implemented in computer applications in two ways.

1. Sets that are subsets of some large set U, called the universal set.
   - If set U has n elements, then any subset S of U can be represented by a bit string of size n, called a bit vector, in which the ith element is 1 if and only if the ith element of U is included in set S.

   ex. $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, $S = \{2, 3, 5, 7\}$

   bit string = 011010100.
2. To use the list structure to indicate the set's elements.

The operations to perform for a set or a multiset most often are

1) searching for a given item.
2) adding a new item.
3) deleting an item from the collection.

A data structure that implements these three operations is called the dictionary

2 Fundamentals of the Analysis of Algorithm Efficiency

- Analysis of algorithms mean an investigation of an algorithm's efficiency with respect to two resources.
  1. Running time. 2. Memory Space.

1. Analysis Framework.

- This specifies a general framework for analyzing the efficiency of algorithms.
- There are two kinds of efficiency.
  1. Time Efficiency 2. Space Efficiency
- Time efficiency indicates how fast an algorithm in question runs.
- Space efficiency deals with the extra space the algorithm requires.

Measuring an Input's Size.

- Almost all algorithms run longer on larger inputs.

  ex. longer to sort larger arrays.
  multiply larger matrices.
- Algorithm's efficiency as a function of some parameter n indicating the algorithm's input size
- In most cases, selecting such a parameter is quite straightforward.

  ex ① It will be the size of the list for problems of sorting, searching, finding the list's smallest element, and problems dealing with lists.

② In evaluating a polynomial $p(x) = a_n x^n + \dots + a_0$ of degree n, it will be the polynomial's degree or the number of its coefficients, which is larger by one than its degree.

- There are situations, where the choice of a parameter indicating an input size does matter.

ex. Computing the product of two n-by-n matrices.

- There are 2 natural measures of size.
  1. The matrix order n.
  2. The total number of elements N in the matrices being multiplied.
- The choice of an appropriate size metric can be influenced by operations of the algorithm in question.
- Measuring the size of inputs for algorithms involving properties of numbers.
- Computer scientists prefere measuring size by the number b of bits in the n's binary representation.

$$b = \lfloor \log_2 n \rfloor + 1$$

Units of Measuring Running Time.

- One possible approach is to count the number of times each of the algorithm's operations is executed
- This approach is difficult and unnecessary.
- To identify the most important operation of the algorithm, called the basic operation, the operation contributing the most to the total running time, and compute the number of times the basic operation is executed.
- It is usually the most time-consuming operation in the algorithm's innermost loop.

ex ① Sorting algorithms work by comparing elements (keys) of a list being sorted with each other, for such algorithms, the basic operation is a key comparison

② In matrix multiplication and polynomial evaluation require two arithmetic operations multiplication and addition.

- In the analysis of an algorithm's time efficiency suggests measuring it by counting the number of times the algorithm's basic operation is executed on inputs of size n.

Let

c_{op} - execution time of an algorithm's basic operation on a particular computer

$C(n)$ - number of times this operation needs to be executed for this algorithm

Running time $T(n) \approx c_{op} C(n)$

Orders of Growth.

- Order of growth in algorithm means how the time for computation increases when you increase the input size.
- It really matters when your input size is very large.
- Order of growth provide only a crude description of the behavior of a process.
- example.

  A process requiring n^2 steps and a process requiring $1000n^2$ steps and a process requiring $3n^2+10n+17$ steps all have $O(n^2)$ order of growth.
- Order of growth provides a useful indication of how we may expect the behavior of the process to change as we change the size of the problem.
- For large values of n, it is the function's order of growth that counts.

n	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3\cdot10^1$	10^2	10^3	10^3	$3.6\cdot10^6$
10^2	6.6	10^2	$6.6\cdot10^2$	10^4	10^6	1.3×10^{30}	$9.3\cdot10^{157}$
10^3	10	10^3	1.0×10^4	10^6	10^9		
10^4	13	10^4	$1.3\cdot10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7\cdot10^6$	10^{10}	10^{25}		
10^6	20	10^6	$20\cdot10^7$	10^{12}	10^{18}		

Worst-Case, Best-Case and Average Case Efficie

- To measure an algorithm's efficiency as a func of a parameter indicating the size of the algorithm's input.
- But there are many algorithms for which running time depends not only on an inpu size but also on the specifics of a particular input.

Ex. Sequential Search Algorithm

This algorithm searches for a given item (some search key k) in a list of n element by checking successive elements of the list until either a match with the search key is found or the list is exhausted.

Alg. 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 of 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
```

- The running time of this algorithm can be quite different for the same list size n.

Worst-case efficiency

- If an algorithm takes maximum amount of time to run to completion for a specific set of input then it is called worst case time complexity.
- The worst case efficiency of an algorithm is its efficiency for the worst-case input of size n, which is an input(s) of size n for which the algorithm runs the longest among all possible inputs of that size.
- We analyze the algorithm to see what kind of inputs yields the largest value of the basic operation's count C(n) among all possible inputs of siz n and then compute this worst-case value $C_{worst}(n)$
- The worst-case analysis provides very important information about an algorithm's efficiency by bounding its running time from above.
- It guarantees that for any instance of size n, the running time will not exceed $C_{worst}(n)$, its running time on the worst-case inputs.

Best-case efficiency

- If an algorithm takes minimum amount of time to run to completion for a specific set of input then it is called best case time complexity.
- The best-case efficiency of an algorithm is its efficiency for the best-case input of size n, which is an input(s) of size n for which the algorithm runs the fastest among all possible inputs of that size.
- We determine the kind of inputs for which the count C(n) will be the smallest among all possible inputs of size n.
- For Sequential search, best-case inputs are lists of size n with their first elements equal to a search key, according $C_{best}(n) =$

Average-case efficiency

- The time complexity that we get for certain set of inputs is as a average same. Then for corresponding input such a time complexity is called average case time complexity.
- It applies not to a single run of an algorithm but rather to a sequence of operations performed on the same data structure.

To analyze the algorithm's average-case efficiency we must make some assumptions about possible input of size n.

eg. Sequential search.

The standard assumptions are.

1) The probability of a successful search is equal to p $(0 \le p \le 1)$.

2)

2) The probability of the first match occuring in the ith position of the list is the same for every i.

The average number of key comparisons $C_{avg}(n)$ as follows

1. In the case of successful search, the probability of the first match occurring in the ith position of the list is p/n for every i, and the number of comparisons made by the algorithm in such a situation is obviously i.

2. In the case of unsuccessful search, the number of comparisons is n with the probability of such a search being $(1-p)$.

$$\therefore C_{avg}(n) = \left[1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}\right] + n \cdot (1-p)$$

$$= \frac{p}{n}[1+2+3 \cdots i + \cdots + n] + n(1-p)$$

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

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

- If $p=1$ (successful), the average no of key comparisons made by sequential search is $(n+1)/2$.
- If $p=0$ (unsuccessful), the average no of key comparisons is n.

Amortized Efficiency

- This is the another type of efficiency.
- It applies not to a single run of an algorithm but rather to a sequence of operations performed on the same data structure.
- Amortized analysis means finding the average running time of operations over a worst sequence of operations.
- Amortized analysis assumes worst case input and does not allow random choice
- Amortized analysis is an analysis technique that provides tight bounds on the worst case running time of an algorith

2. Asymptotic Notations and Basic Efficiency classes

- The efficiency analysis framework concentrates on the order of growth of an algorithm basic operation count as the principal indicator of the algorithm's efficiency.
- To compare and rank such orders of growth, computer scientists use 3 notations O (big oh), Ω (big omega), Θ (big theta)
- $t(n)$ and $g(n)$ can be any nonnegative functions defined on the set of natural numbers.

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

1. Informal Introduction.

- Informally, $O(g(n))$ is the set of all functions with a smaller or same order of growth as $g(n)$
- ex. The following assertions are all true.

  $n \in O(n^2)$, $100n+5 \in O(n^2)$, $\frac{1}{2}n(n-1) \in O(n^2)$
- The first two functions are linear and hence have a smaller order of growth than $g(n) = n^2$
- The last one is quadratic and hence has the same order of growth as n^2
- On the other hand.

  $n^3 \notin O(n^2)$, $0.0001n^2 \notin O(n^2)$, $n^3+n+1 \notin O(n^2)$
- All are cubic and hence have a higher order of growth than n^2.
- $\Omega(g(n))$ stands for the set of all functions with a larger or same order of growth as $g(n)$.

ex

$n^3 \in \Omega(n^2)$, $\frac{1}{2}n(n-1) \in \Omega(n^2)$

but $100n+5 \notin \Omega(n^2)$

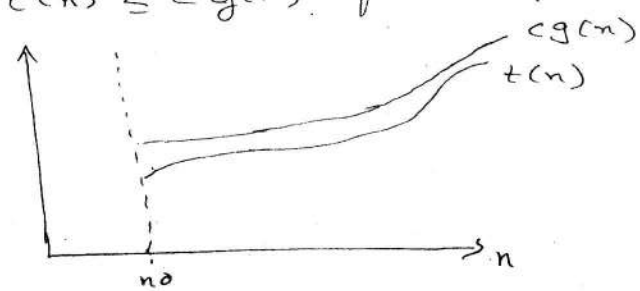
$\Theta(g(n))$ is the set of all functions that have the same order of growth as $g(n)$

Every quadratic function an^2+bn+c with $a>0$ is in $\Theta(n^2)$.

O-notation.

Def. A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n.

i.e. there exist some positive constant c and some nonnegative integer n_0 such that $t(n) \le c\,g(n)$ for all $n \ge n_0$



ex. $100n+5 \in O(n^2)$

$100n+5 \le 100n+n$ (for all $n \ge 5$)

$= 101n \le 101n^2$.

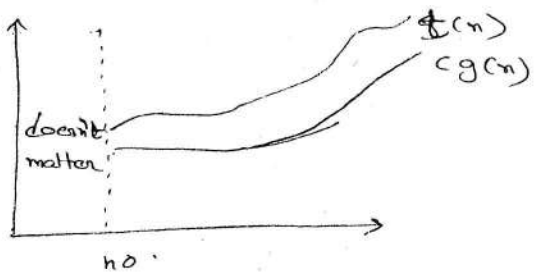
Thus, as values of the constants c and n_0 required by the definition, we can take 101 and 5 respectively.

Ω - notation.

Def. A function $t(n)$ is said to be $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$, if $t(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large n.

i.e. If there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \geq c\,g(n) \text{ for all } n \geq n_0$$



ex. $n^3 \in \Omega(n^2)$.

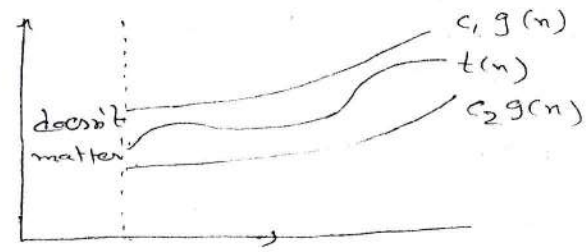
$n^3 \geq n^2$ for all $n \geq 0$.

i.e. we can select $c = 1$ and $n_0 = 0$.

Θ - notation

Def. A function $t(n)$ is said to be $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, 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 constant c_1 and c_2 and some nonnegative integer n_0 such that $c_2 g(n) \leq t(n) \leq c_1 g(n)$, for all $n \geq n_0$



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

① we prove the right inequality (upper bound)

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n$$
$$\leq \frac{1}{2}n^2 \text{ for all } n \geq 0.$$

② we prove the left inequality (lower bound)

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n$$
$$> \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n \quad (\text{for all } n \geq 2)$$
$$= \frac{1}{4}n^2$$

Hence we can select $c_2 = \frac{1}{4}$, $c = \frac{1}{2}$ and $n_0 = 2$

Useful Property Involving the Asymptotic Notations:

- If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then $t_1(n) + t_2(n) \in O(\max[g_1(n), g_2(n)])$

Using Limits for Comparing Orders of Growth

- Formal definitions of O, Ω and Θ are rarely used for comparing the orders of growth of two specific functions.

- A much more convenient method for doing so is based on comparing the limit of the ratio of two functions in question.
- Three principal cases may arise.

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

- The first two cases mean that $t(n) \in O(g(n))$
- The last two mean that $t(n) \in \Omega(g(n))$.
- The second case means that $t(n) \in \Theta(g(n))$.
- The limit-based approach is often more convenient than the one based on the definition.
- L. Hopital's Rule.

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

Stirling's formula

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

<u>example</u>

compute the order of growth of $\frac{1}{2}n(n-1)$ and n^2.

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

The limit equal to positive constant, the function have the same order of growth.

Symbolically $\frac{1}{2}n(n-1) \in O(n^2)$.

Basic Efficiency classes

The time efficiencies of a large number of algorithms fall into only a few classes.

Class	Name	Comments
1	constant	short of best-case efficiencies, very few reasonable examp can be given since an algorit running time typically goes f infinitely when its input size go infinitely large.
logn.	logarithm.	A logarithmic algorithm can't tak into account all its input; any algorithm that does so will hav at least linear running time
n.	linear	Algorithms that scan a list of size n belong to this class.
nlogn.	n-log-n.	many divide-and-conquer algorit including merge sort & quick sor in the average case fall into this category.
n^2	quadratic.	Typically characterizes efficiency algorithms with two embedded loo (matrices)
n^3	cube	Typically characterizes efficiency algorithms with three embedded loo (Linear algebra)
2^n	Exponential	Typically for algorithms that genera all subsets of n-element set.
n!	factorial	Typically for algorithms all permutations of an n-element se

3. Mathematical Analysis of Nonrecursive Algorithms.

Example. Finding the value of the largest element in a list of n numbers.

Algorithm MaxElement(A[1..n-1])

// Determines the value of the largest element
// in a given array.
// Input: An array A[0..n-1] of real number
// 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.
```

- The obvious measure of an input's size here is the no of elements in the array i.e. n.
- The operations that are going to be executed most often are in the algorithm's for loop.
- There are two operations in the loop's body the comparison A[i] > maxval and the assignment maxval ← A[i].
- which of these two operations should we consider basic?
- Since the comparison is executed on each repetition of the loop and the assignment is not,
- we should consider the comparison to be the algorithm's basic operation.
- Let C(n) the no of times this comparison is executed.
- The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i with the bounds 1 and n-1.

∴ The sum for $C(n) = \sum_{i=1}^{n-1} 1$

This is the an easy sum to compute because it is nothing else but 1 repeated n-1 times

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

General Plan for Analyzing Time Efficiency of Nonrecursive Algorithms.

1. Decide on a parameter(s) indicating an input size.
2. Identify the algorithm's basic operation. (As a rule, it is located in its innermost loop)
3. Check whether the no of times the basic operation is executed depends only on the size of an input.

It is also depends on some additional property the worst-case, average-case, and if necessary, best-case efficiencies have to be investigated separately.

*. Set up a sum expressing the no of times the algorithm's basic operation is executed.

5. Using standard formula and rules of sum manipulation, either find a closed form formula for the count, at the very least establish its order of growth.

Examples

1. Element uniqueness problem.
2. Matrix Multiplication.

4. Mathematical Analysis of Recursive Algorithms

Example Compute the factorial function $F(n) = n!$ for an arbitrary nonnegative integer n.

$$n! = 1 \cdot 2 \cdot \cdots (n-1) \cdot n$$
$$= (n-1)! \cdot n \quad \text{for } n \geq 1$$

$$0! = 1$$

By definition, we can compute $F(n) = F(n-1) \cdot n$

Algorithm F(n).

// compute n! recursively

// Input: A nonnegative integer n.

// Output: The value of n!

if n = 0 return 1

else return F(n-1) * n.

- Input Size = n.
- Basic operation = multiplication.

- The function F(n) is computed according to the formula $F(n) = F(n-1) \cdot n$ for $n \geq 0$.
- The no of multiplications M(n) needed to compute it must satisfy the equality

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

- M(n-1) multiplications are spent to compute F(n-1) and one more multiplication is needed to multiply the result by n.
- Indeed, M(n-1) multiplications are spent to compute F(n-1) and one more multiplication is needed to multiply the result by n.
- We need to find M(n).
- This equation defines M(n) not explicitly i.e. as a function of n, but implicitly as a function of its value at another point n-1.
- Such equations are called recurrence relations or recurrences.
- Our goal is to solve the recurrence relation

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

i.e. to find an explicit formula for M(n) in terms of n only.

- To determine a solution uniquely, we need an initial condition that tells us the value with which the sequence starts

if $n=0$ return 1

- This tells 2 things.

1. The call stop when $n=0$, the smallest value of n for which this algorithm is executed hence $M(n)$ defined is 0.
2. when $n=0$, the algorithm performs no multiplication.

- Thus initial condition is $M(0)=0$.
- Find the algorithm's no of multiplications $M(n)$.

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

- The factorial function

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

- $M(n)$ is defined by recurrence

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

- General formula is

$$\begin{aligned} M(n) &= M(n-i)+i \\ &\vdots \\ &= M(n-n)+n \\ &= M(0)+n \\ &= 0+n \\ &= n. \end{aligned}$$

General plan for analyzing Time Efficiency of Recursive Algorithms

1. Decide on a parameter(s) indicating an input's size.
2. Identify the algorithm's basic operation.
3. Check whether the no of times the basic operation is executed can vary on different inputs of the same size.
   if it can, 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 no of times the basic operation is executed.
5. Solve the recurrence or at least ascertain the order of growth of its solution.

5. Empirical Analysis of Algorithms

- The principal alternative to the mathematical analysis of an algorithm's efficiency is its empirical analysis.

General plan for 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's count vs. a time unit)
3. Decide on characteristics of the input (its range, size, and so on).
4. Prepare a program implementing the algorithm(s) for the experimentation.
5. Generate a sample of inputs.
6. Run the algorithm(s) on the sample's input and record the data observed.
7. Analysis the data obtained.

- There are several different goals one can pursue in analyzing algorithms empirically.

1. Checking the accuracy of a theoretical assertion about the algorithm's efficiency.
2. Comparing the efficiency of several algorithms for solving the same problem or different implementations of the same algorithm
3. Developing a hypothesis about the algorithm's efficiency class.
4. ~~Ascertaining~~ Ascertaining the efficiency of the program implementing the algorithm on a particular machine.

How the algorithm's efficiency is to be measured

1. To insert a counter into a program implementing the algorithm to count the no of times the algorithm's basic operation is executed.
2. To time the program implementing the algorithm in question.

- The easiest way to do this is to use system command.
- In C and C++, you can use the function clock for this purpose.
- In Java, the method currentTimeMillis() in the System class is available.
- The principal advantage of size changing searching to a pattern is that its impact is easier to Analyze.
- An empirical analysis of an algorithm's efficiency requires generating random numbers.
- Even if we decide to use a pattern for input sizes, we typically want instances themselves generated randomly.

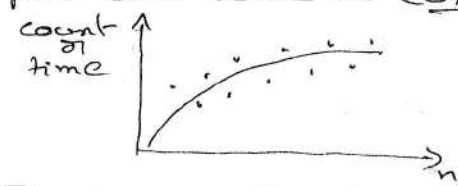
- Generating random numbers on a digital computer is known to present a difficult problem because, in principle, the problem can be solved only approximately.
- This is the reason computer scientist prefer to call such numbers pseudorandom
- Its output will be a value of a pseudorandom variable formly distributed in the interval between 0 and 1.
- If a different pseudorandom variable is desired, an appropriate transformation needs to be made.
- The most widely used and thoroughly studied of such algorithm is the linear congruential method.

Algorithm. Random(n, m, seed, a, b)

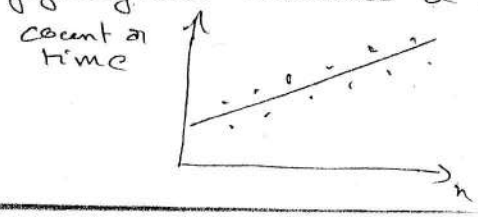
// Generate a sequence of n pseudorandom
// numbers according to the linear
// congruential method.
// input: A positive integer n and a positive
// integer parameters m, seed, a, b
// output: A sequence $r_1 \ldots r_n$ of n pseudorandom
// integers uniformly distributed among
// integer values between 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.

$r_0 \leftarrow$ seed.
for $i \leftarrow 1$ to n do
$r_i \leftarrow (a * r_{i-1} + b) \bmod m$.

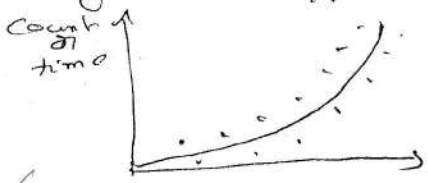
- Seed may be choosen arbitrarily and is often set to the current date and time
- m should be large and may be conveniently taken as 2^n.
- a should be selected as integer between 0.01m and 0.99m with no particular pattern in its digit
- The value of b can be chosen as 1.
- The empirical data obtained 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
- For a logarithmic algorithm, the scatter plot will have a concave shape.



- For linear algorithm the points will tend to aggregate around a straight line



- Scatterplots of function in $\Theta(n \log n)$ and $\Theta(n^2)$ will have a convex shape, making them difficult to differentiate.



6. Algorithm Visualization.

- Algorithm visualization defined as the use of images to convey some useful information about algorithms.
- An algorithm visualization uses graphic elements (points, line segments, two-or-three dimensional bars, and so on) to represent some "interesting events" in the algorithms operation.
- There are two principal variations of algorithm visualization
  1. Static algorithm visualization.
  2. Dynamic algorithm visualization, also called algorithm animation.
- Static algorithm visualization shows an algorithm's progress through a series of still images.
- Dynamic algorithm visualization a continuous, movie-like presentation of an algorithm's operations.

- The algorithm visualization classic was produced at the university of Toronto by Ronald Baecker with assistance of D. Sherm
- The success of Sorting Out Sorting made sorti algorithms a perennial favorite for algorithm animation.
- The sorting problem lends itself quite natu to visual presentation via vertical or horizontal bars or sticks of different height or lengths, which are rearranged according to their sizes.
- A good general-purpose animation system should allow a user to not only watch and interact with existing animations of a wide variety of algorithms, it should also provid functions for creating new animations.

Features of animation's

1) Be consistent. 2) Be interactive. 3) Be clear and concise. 4) Be forgiving to the user. 5) Adapt to the knowledge level of the user. 6) Emphasize the visual component. 7) Keep the user interested. 8) Incorporate both symbolic and iconic representations. 9) Include algorithm's analysis and comparison with other algorithms for the same problem. 10) Include execution history.

2 applications.

1 - Research 2 - Education.

3. BRUTE FORCE

Brute Force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.

1. Brute force is applicable to a very wide variety of problems.
2. For some important problems (e.g. sorting, searching, matrix multiplication, string matching) the brute force approach yields reasonable algorithms of at least some practical value with no limitation on instance size.
3. The expense of designing a more efficient algorithm may be unjustifiable if only a few instances of a problem need to be solved and a brute-force algorithm can solve those instances with acceptable speed.
4. Even if too inefficient in general, a brute-force algorithm can still be useful for solving small-size instances of a problem.
5. A brute-force algorithm can serve an important theoretical or educational purpose

1. Selection Sort and Bubble Sort

These two are applications of Brute-Force

Selection Sort

1. Scanning the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list.
2. Then we scan the list, starting with the second element, to find the smallest among the last n-1 elements and exchange it with the second element, putting the second smallest element in its final position.
3. Generally, on the ith pass through the list, which we number from 0 to n-2, the algorithm searches for the smallest item among the last n-i elements and swaps it with A_i.
4. After n-1 passes, the list is sorted.

Algorithm SelectionSort(A[0..n-1])

```
//Sorts a given array by selection sort.
//Input: An array A[0..n-1] of orderable elements
//Output: An array A[0..n-1] sorted in Ascending order
for i ← 0 to n-2 do
    min ← i
    for j ← i+1 to n-1 do
        if A[j] < A[min]  min ← j.
    swap A[i] and A[min]
```

Example

$\overset{0}{89}, \overset{1}{45}, \overset{2}{68}, \overset{3}{90}, \overset{4}{29}, \overset{5}{24}, \overset{6}{17}$

Execution

$i = 0 \quad n = 7$

min = 0

j = 1 to 6.

A[1] < A[0] => 45 < 89 yes then min = 1

j = 2, min = 1

A[2] < A[1] => 68 < 45 no.

~~swap (A[2], A[1])~~ j = 3, min = 1

=> A[3] < A[1]

90 < 45 min = 3.

j = 4 min 3

A[4] < A[3].

29 < 90 min = 4.

j = 5, min = 4.

A[5] < A[4].

24 < 29 min = 5

j = 6 min = 5

A[6] < A[5]. m = 6

17 < 24.

i = 0, m = 6

Example 89, 45, 68, 90, 29, 34, 17

$n = 7$.

$A[7] = \overset{0}{89}, \overset{1}{45}, \overset{2}{68}, \overset{3}{90}, \overset{4}{29}, \overset{5}{34}, \overset{6}{17}$.

for $i = 0$ to 5 ; $i = 0$

$min = i = 0$.

for $j = 1$ to 6 , $j = 1$.

$A[j] < A[min] \Rightarrow 45 < 89 \overset{yes}{\Rightarrow} min = 1$

$j = 2$, $min = 1$

$A[2] < A[1] \Rightarrow 68 < 45 \overset{no}{\Rightarrow}$

$j = 3$, $min = 1$

$A[3] < A[1] \Rightarrow 90 < 45 \overset{no}{\Rightarrow}$

$j = 4$, $min = 1$

$A[4] < A[1] \Rightarrow 29 < 45 \overset{yes}{\Rightarrow} min = 4$

$j = 5$, $min = 4$

$A[5] < A[4] \Rightarrow 34 < 29 \overset{no}{\Rightarrow}$

$j = 6$, $min = 4$

$A[6] < A[4] \Rightarrow 17 < 29 \overset{yes}{\Rightarrow} min = 6$.

j loop completed then swap $A[i]$, $A[min]$.
$\Rightarrow A[0], A[6]$.

$\Rightarrow$

Ans:

17, 45, 68, 90, 29, 34, 89

17, 29, 68, 90, 45, 34, 89

17, 29, 34, 90, 45, 68, 89

17, 29, 34, 45, 90, 68, 89

17, 29, 34, 45, 68, 90, 89

17, 29, 34, 45, 68, 89, 90

Analysis

Input size = 7.

Basic operation is comparison

$A[j] < A[min]$

Number of time it is executed depends only the array's size.

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

$$= \sum_{i=0}^{n-2} \left[(n-1) - (i+1) + 1\right]$$

$$= \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i$$

$$= (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

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

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

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

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

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

Time complexity of Best $= \Omega(n^2)$

worst $= \Theta(n^2)$

Bubble Sort

1. To compare adjacent elements of the list and exchange them if they are out of order.
2. By doing it repeatedly, we end up "bubbling up" the largest element to the last position on the list.
3. The next pass bubbles up the second largest element, and so on until, after n-1 passes, the list is sorted.

Algorithm. BubbleSort(A[0..n-1])

// Sorts a given array by bubble sort.
// Input: An array A[0..n-1] of orderable elements
// Output: Array A[0..n-1] sorted in ascending
// order

```
for i ← 0 to n-2 do
    for j ← 0 to n-2-i do
        if A[j+1] < A[j] swap A[j] and A[j+1]
```

Example 89, 45, 68, 90, 29, 34, 17

$i = 0$ to 5

$j = 0$ to 5, $j = 0$

$A[1] < A[0] \Rightarrow 45 < 89 \Rightarrow$ swap [A[1], [A0]]

$j = 1 \Rightarrow A[2] < A[1] \Rightarrow 68 < 89 \Rightarrow$ swap.

$j = 2 \Rightarrow A[3] < A[2] \Rightarrow 90 < 89$ No

$j = 3 \Rightarrow A[4] < A[3] \Rightarrow 29 < 90 \Rightarrow$ swap

89 ↔ 45 68 90 29 34 17.

45 89 ↔ 68 90 29 34 17

45 68 89 ↔ 90 29 34 17

45 68 89 90 ↔ 29 34 17

45 68 89 29 90 ↔ 34 17

45 68 89 29 34 90 ↔ 17

45 68 89 29 34 17 90.

45 ↔ 68 ↔ 89 ↔ 29 34 17 90

45 68 29 89 ↔ 34 17 90.

45 68 29 34 89 ↔ 17 90

45 68 29 34 17 89 90.

etc.

Analysis

Input size = 7.

Basic operation is comparison.

$A[j+1] < A[j]$.

The no of key comparisons is

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

$$= \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1]$$

$$= \sum_{i=0}^{n-2} (n-1-i)$$

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

Time complexity Best = $\Omega(n)$

worst = $\Theta(n^2)$

2. Sequential Search and Brute-Force String Matching

Two applications of the brute-force approach to the searching problem.

1. Sequential Search.

It deals with the canonical problem of searching for an item of a given value in a given list.

2. Brute-Force String Matching

It deals with the string matching problem.

Sequential Search

- The algorithm simply compares successive elements of a given list with a given search key until either a match is encountered (successful search) or the list is exhausted without finding a match (unsuccessful search).

Algorithm SequentialSearch(A[0..n-1], k)

// Implementing sequential search with a
// search key.
// Input: An array A[0..n-1] and a search
// key k.
// Output: The index of the first element
// of 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
```

ex. 58, 62, 75, 13, 98, 12. n = 6

search key = 75

i = 0

i < 6 & A[0] = 58 ≠ 75

i = 0+1 = 1

1 < 6 & A[1] = 62 ≠ 75

i = 1+1 = 2

2 < 6 & A[2] = 75 = 75

if i < n ⇒ 2 < 6 then return 2.

The element found at 2nd element.

Analysis

Input size = n.

Basic operation = A[i] ≠ k - comparison

$$C(n) = \sum_{i=0}^{n-1} 1$$

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

$$= n.$$

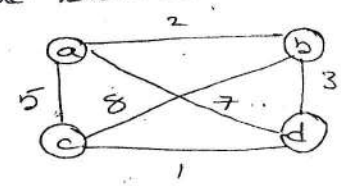
$$\in \Theta(n)$$

3. Exhaustive Search

- It is simply a brute-force approach to combinatorial problem i.e permutations, combinations, and subsets of a given set.
- The idea of exhaustive search is quite straightforward, its implementation typically requires an algorithm for generating certain combinatorial objects.
- Exhaustive search is applying to 3 important problems.
  1. Traveling Salesman Problem.
  2. The knapsack problem.
  3. The assignment problem.

Traveling Salesman Problem

- In Dalyman's terms, the problem asks 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.
- The problem can be conveniently modeled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifically specifying the distances.
- Then the problem can be stated as the problem of finding the shortest Hamiltonian Circuit of the graph.
- A Hamiltonian circuit is defined as a cycle that pass through all the vertices of the graph exactly once.
- A Hamiltonian circuit is defined as a sequence of n+1 adjacent vertices $v_{i_1}, v_{i_2} \cdots v_{i_{n-1}}, v_{i_n}$, where the first vertex of the sequence is the same as the last one while all the other n-1 vertices are distinct.
- We assume, with no loss of generality, that all circuits start and end at one particular vertex.
- Thus, we can get all the tours by generating all the permutations of n-1 intermediate cities, compute the tour lengths, and find the shortest among them.



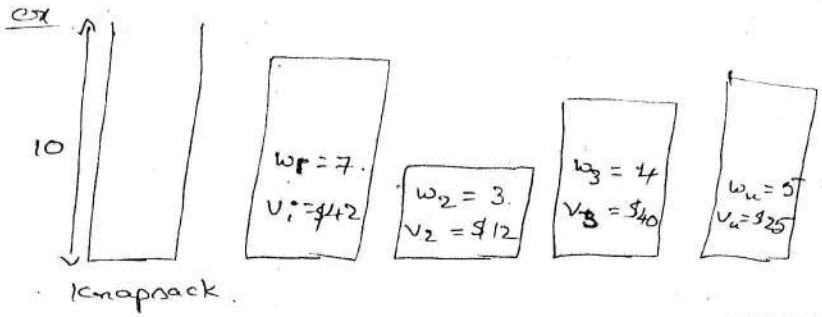
Tour	Length	
a → b → c → d → a	2 + 8 + 1 + 7 = 18	
a → b → d → c → a	2 + 3 + 1 + 5 = 11	optimal
a → c → b → d → a	5 + 8 + 3 + 7 = 23	
a → c → d → b → a	5 + 1 + 3 + 2 = 11	optimal
a → d → b → c → a	7 + 3 + 8 + 5 = 23	
a → d → c → b → a	7 + 1 + 8 + 2 = 18	

- Three pairs of tours that differ only their direction.
- Hence, we could cut the number of vertex permutations by half.
- This improvement can't brighten the efficiency picture much, however.
- The total no of permutations needed will still be $(n-1)!/2$
- which makes the exhaustive approach impractical for all but very small values of n.
- Also note that had we not limited our investigation to the circuits starting at the same vertex, the number of permutations would have been even larger, by a factor of n.

2. Knapsack Problem

- Given n items of known weights $w_1, \dots w_n$ and values $v_1 \dots v_n$ and a knapsack of capacity W.
- Find the most valuable subset of the items that fit into the knapsack.
- The exhaustive-search approach to this problem leads to generating all the subsets of the set of n items given, computing the total weight of each subset to identify feasible subsets (i.e. the ones with the total weight not exceeding the knapsack's capacity) and finding a subset of the largest value among them.

ex



Subset	Total Weight	Total Value
ϕ	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
$\{2,3,4\}$	12	not feasible
$\{1,2,3,4\}$	19	not feasible

- The number of subsets of an n-element set is 2^n.
- The exhaustive search leads to a $\Omega(2^n)$ algorithm no matter how efficiency individual subsets are generated.
- For both the traveling salesman and knapsack problems, exhaustive search leads to algorithms that are extremely inefficient on every input.
- These two problems are the best-known examples of <u>NP-hard</u> problems.
- Backtracking and branch-and-bound enable us to solve some but not all instances of these.

Divide and conquer

Quick sort

↳ Binary search

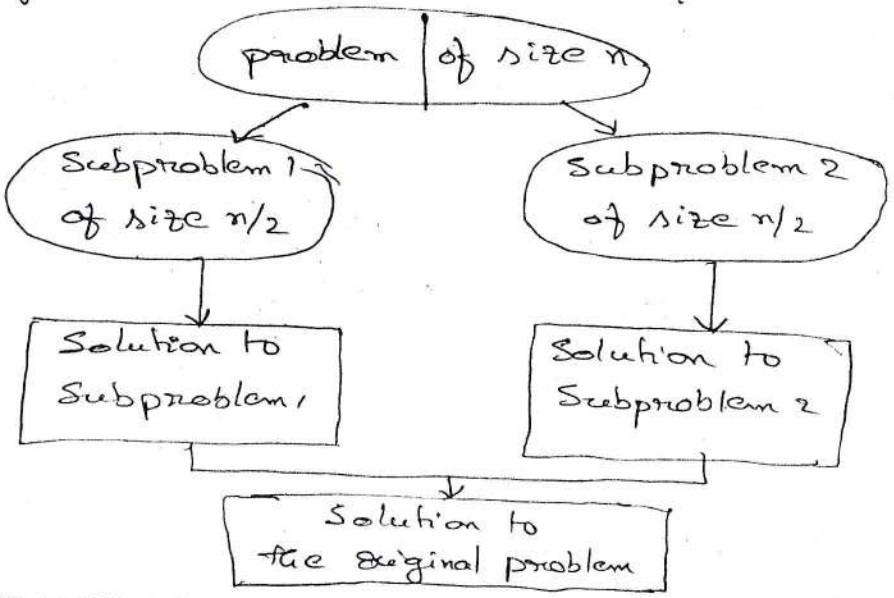
↳ Binary tree traversal

↳

DIVIDE - AND - CONQUER.

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

1. A problem's instance is divided into several smaller instances of the same problem, ideally of about the same siz
2. The smaller instances are solved (typica recursively, though sometimes a differen algorithm is employed when instances become small enough)
3. If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance.



Example

- Computing the sum of n numbers.
- The n numbers are $a_0 \dots a_{n-1}$
- 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 sum of the remaining $\lceil n/2 \rceil$ numbers.
- We can add their values to get the sum in question.

$$a_0 + \dots a_{n-1} = (a_0 + \dots a_{\lfloor n/2 \rfloor - 1}) + (a_{\lceil n/2 \rceil} + \dots + a_{n-1})$$

- Thus, not every divide-and-conquer algorithm is necessarily more efficient than even a brute-force solution.
- The time spent on executing the divide-and-conquer plan turns out to be smaller than solving problem by a different method.
- The divide-and-conquer approach yields some of the most important and efficient algorithms in computer science.
- In divide-and-conquer, a problem's intance of size n is divided into two instances of size n/2.
- More generally, an instance of size n can be divided into b instances of size n/b, with a of them needing to be solved.
- Assume that size n is a power of b, to simplify our analysis, we get the following recurrence for the running time T(n)

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

f(n) - function that accounts for the time spent on dividing the problem into smaller once and on combining their solutions

Master Theorem

- The efficiency analysis of many divide-and-conquer algorithms is if $f(n) \in \Theta(n^d)$, $d \geq 0$ then

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

1. MergeSort

- It is the application of the divide-and-conquer technique.
- It sorts a given array $A[0..n-1]$ by dividing it into two halves $A[0..\lfloor n/2 \rfloor - 1]$ and $A[\lceil n/2 \rceil .. n-1]$.
- Sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

Algorithm. Mergesort(A[0..n-1])

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

if $n > 1$

copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$

copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$

Mergesort($B[0..\lfloor n/2 \rfloor - 1]$)

Mergesort($C[0..\lceil n/2 \rceil - 1]$)

Merge(B, C, A)

- The <u>Merge</u> of two sorted arrays can be done as follows.
- Two pointers (array indices) are initialized to point to the first elements of the arrays being merged.
- The elements pointed to are compared, and the smaller of them is added to a new array being constructed.
- After that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from.
- This operation is repeated 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.

<u>Algorithm</u>. Merge($B[0..p-1]$), $C[0..q-1]$, $A[0..p+q-1]$)

// Merges two sorted arrays into one sorted

// array.

// Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted.

// Output: Sorted array $A[0..p+q-1]$ of the elements of B and C

$i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$

while $i < p$ and $j < q$ do

  if $B[i] \leq C[j]$

    $A[k] \leftarrow B[i]$; $i \leftarrow i+1$

  else

    $A[k] \leftarrow C[j]$; $j \leftarrow j+1$

  $k \leftarrow k+1$

if $i = p$

  copy $C[j..q-1]$ to $A[k..p+q-1]$

else

  copy $B[i..p-1]$ to $A[k..p+q-1]$

<u>Efficiency</u>

Assuming for simplicity that n is a power of 2

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

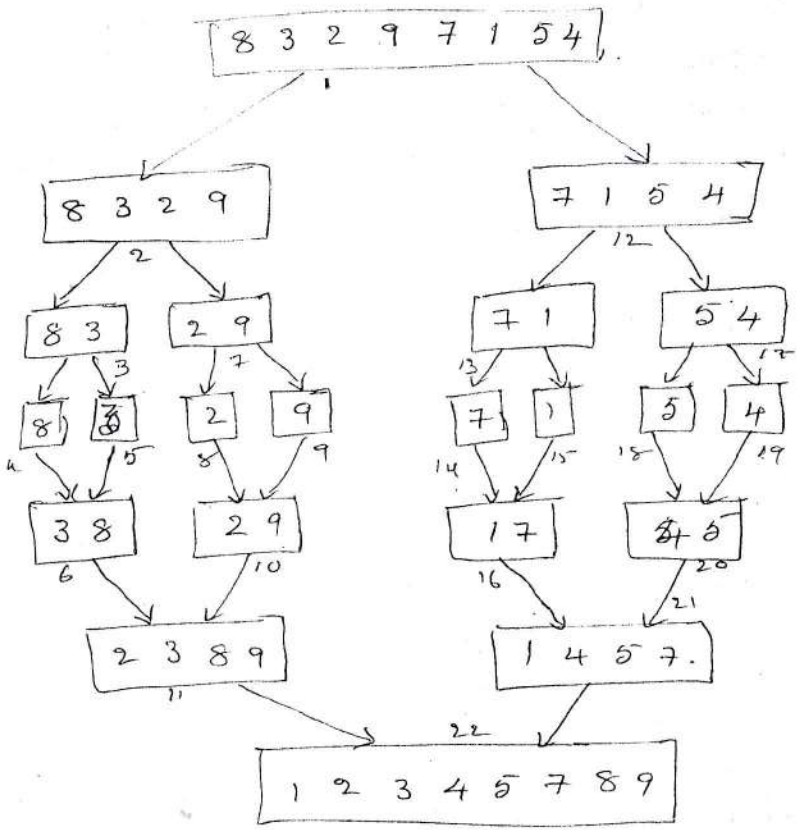
$C_{merge}(n)$ = The no of key comparisons performed during the merging stage.

In worst case $C_{merge}(n) = n-1$

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1$$

$$= n\log_2 n - n + 1$$

$\therefore C_{worst} = \Theta(n \log n)$ for $n = 2^k$.



① Merge sort the following list.
38, 27, 43, 3, 9, 82, 10

② Apply mergesort to sort the list
E, X, A, M, P, L, E in aphabetical order

2. Quicksort

- Quicksort is another application of divide-and-conquer approach.
- Quicksort divides them according to their value.
- Specifically, it rearranges elements of a given array $A[0..n-1]$ to achieve its partition, a situation where all the elemen before some position s are smaller than or equal to $A[s]$ and all the elemen after position's are greater than or equal to $A[s]$.

$$\underbrace{A[0] \cdots A[s-1]}_{\text{all are } \le A[s]}\ A[s].\ \underbrace{A[s+1] \cdots A[n-1]}_{\text{all are } \ge A[s]}$$

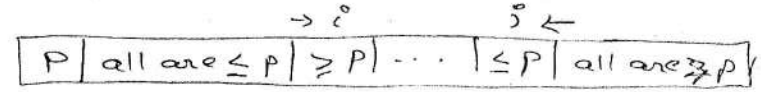
- After a position has been achieved, $A[s]$ will be in its final position in the sorted array, and we can continue sorting the two subarrays of the elements preceding and following $A[s]$ independently.
- A partition of $A[0..n-1]$ and more generally, of its subarray $A[l..r]$, $0 \le l \le r \le n-1$) can be achieved as.
- First, we select an element with respect to whose value we are going to divide the subarray.
- We call this element the pivot
- There are several different strategies for selecting a pivot.

- We use the simplest strategy of selecting the subarray's first element.

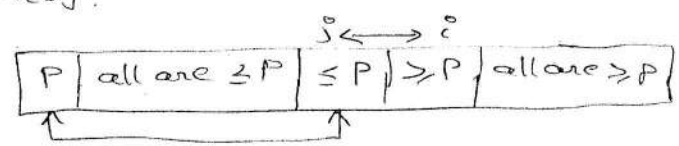
  $P = A[l]$.

- There are also several alternative procedures for rearranging elements.
- An efficient method based on two scans of the subarray.
- One is left-to-right scan, and the other right-to-left scan, each comparing the subarray's elements with the pivot.
- The left-to-right scan, index i starts with the second element.
- Since we want elements smaller than the pivot to be in the first part of the subarray, this scan skips over elements that are smaller than the pivot and stops on encountering the first element greater than or equal to the pivot.
- The right-left scan, index j starts with the last element of the subarray,
- this scan skips over elements that are larger than the pivot and stops on encountering the first element smaller than or equal to the pivot.
- After both scans stop, 3 situations may arise, depending on whether or not the scanning indices have crossed.

1. If scanning indices i and j have not crossed i.e $i < j$, we simply exchange $A[i]$ and $A[j]$ and resume the scans by incrementing i and decrementing j, respectively.

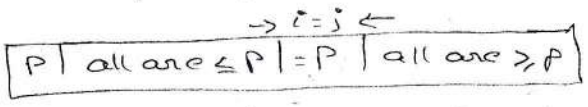


2. If the scanning indices have crossed over i.e. $i > j$, we will have partitioned the array after exchanging the pivot with $A[j]$.



3. If the scanning indices stop while pointing to the same element i.e. $i = j$, the value they are pointing to must be equal to P.

Thus, we have the array partitioned, with the split position $s = i = j$.



- We can combine the last case with the case of crossed-over indices ($i > j$) by exchanging the pivot with $A[j]$ whenever $i \geq j$

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: 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]).
```

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

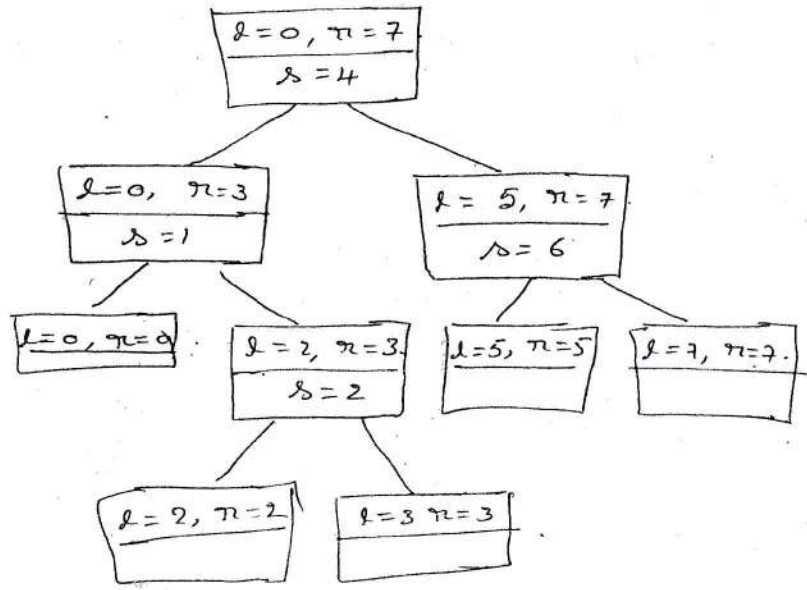
```
P ← A[l]
i ← l ; j ← r+1
repeat
    repeat i ← i+1 until A[i] ≥ P.
    repeat j ← j-1 until A[j] ≤ P.
    swap(A[i], A[j])
until i ≥ j
swap(A[i], A[j]) // undo last swap when i ≥ j
swap(A[l], A[j])
return j.
```

example list

5, 3, 1, 9, 8, 2, 4, 7

0	1	2	3	4	5	6	7	
i P [5]	3 i	1	9	8	2	4	7 j	
5	3	1	9 i	8	2	4 j	7	
5	3	1	4 i	8	2	9 j	7	swap(i,j)
5	3	1	4	8 i	2 j	9	7	
5	3	1	4	2 i	8 j	9	7	swap(i,j)
5	3	1	4	2 j	8 i	9	7	
i [2]	3	1	4 j	5 s	8	9	7	swap (A[l], A[s])
2	3 i	1 j	4					
2	1 i	3 j	4	swap(i,j)				
2	1 j	3 i	4					
1	[2] s	3 i	4 j	swap(A[l] & A[j])				
		[3] j	4 i					
		3	4					
		[3]	4					
			4					
				i [8]	9 i	7 j		
				8	9 j	7 i		
				8	7 j	9 i		swap(i,j)
				7	8 s	9		swap(l,s)
						9		

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



Efficiency

$C_{best}(n) = 2C_{best}(n/2) + n$ for $n > 1$. $C_{best}(1) = 0$.

$C_{best}(n) \in \Theta(n \log_2 n)$ for $n = 2^k$

$\therefore C_{best}(n) = n \log_2 n$

best case $\Theta(n \log n)$

worst case $\Theta(n^2)$

Avg. case $\Theta(n \log n)$

3 Binary Search.

- Binary search is efficient algorithm for searching in a sorted array.
- It works by 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 $K > A[m]$.

$$\underbrace{A[0], \cdots A[m-1]}_{\text{Search here if } K < A[m]} \quad \overset{K}{A[m]} \quad \underbrace{A[m+1] \cdots A[n-1]}_{\text{Search here if } K > A[m]}$$

Algorithm: BinarySearch($A[0..n-1]$, K)

// Implements nonrecursive binary search
// Input: An array $A[0..n-1]$ sorted in ascending
// order and a search key K.
// Output: An index of the array's element that
// is equal to K. or -1 if there is no such
// element

```
l ← 0 ; r = n-1
while l ≤ r do
    m ← [(l+r)/2]
    if K = A[m] return m
    else if K < K[m]  r ← m-1
    else l ← m+1
return -1.
```

example $k = 70$

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

0	1	2	3	4	5	6	7	8	9	10	11	12
3	14	27	31	39	42	55	70	74	81	85	93	98

$l = 0,\ r = 12$

iteration 1 $l < r \Rightarrow 0 < 12$

$m \leftarrow (0+12)/2 = 6.$

$70 > 55$ then $l = m+1 = 6+1 = 7$

iteration 2 Now $l = 7,\ r = 12.$

$m \leftarrow (7+12)/2 = 8.5 = 9.$

$70 < 81$ $r = m-1 = 9-1 = 8$

iteration 3 Now $l = 7,\ r = 8.$

$m \leftarrow (7+8)/2 = 15/2 = 7.5 \to 7$

$70 = 70$ then return 7

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

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

if $n = 2^k$.

$C_{worst}(2^k) = k+1 = \log_2 n + 1.$

$= \Theta(\log n)$

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

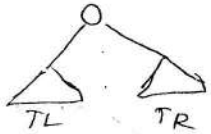
Example

10, 12, 5, 9, 4, 11, 8, 3, 13, 2, 7, 1, 6, key = 2

4. Binary Tree Traversals and Related Properties.

Binary Tree

A Binary Tree T is defined as a finite set of nodes that is either empty or consists of a root and two disjoint binary trees T_L and T_R, the left and right subtree of the root.



Since the definition itself divides a binary tree into two smaller structures of the same type, the left subtree and right subtree.

Height of a Binary Tree.

Height is defined as the length of the longest path from the root to a leaf. Hence, it can be computed as the maximum of the height of the root's left and right subtree plus 1.

- We add 1 to account for the extra level of the root.
- The height of the empty tree as -1.

Algorithm Height(T)

// Computes recursively the height of a
// binary tree.
// Input: A binary Tree
// Output: The height of T.

```
if T = ∅ return -1
else
    return max[Height(T_L), Height(T_R)] + 1
```

- Number of additions $A(n(T))$

$$A(n(T)) = A(n(T_L)) + A(n(T_R)) + 1 \text{ for } n(T) > 0.$$
$$A(\emptyset) = 0$$

- Addition is not the most frequently executed operation of this algorithm.
- For the empty tree, the comparison $T = \emptyset$ is executed once but there are no additions and for single node tree, the comparison and addition numbers are three and one respectively.

1. The most important divide-and-conquer algorithm for binary trees are the three classic traversals.
   1. Preorder 2. Inorder 3. Postorder
- All three traversals visit node of a binary tree recursively. i.e visiting the tree's root and its left and right subtrees.
- They differ just by the timing of the root's visit.

1. Preorder Traversal

In the preorder traversal, the root is visited before the left and right subtrees are visited

Pseudocode.

```
Preorder(tree).
1. visit the root.
2. Traverse the left subtree. i.e.
   call preorder(left_subtree)
3. Traverse the right subtree. i.e.
   call preorder(right_subtree.
```

2. Inorder Traversal.

In the inorder traversal, the root is visited after visiting the left subtree but before visiting the right subtree.

Pseudocode

1. Traverse the left subtree. i.e. call Inorder (left-subtree)
2. Visit the root.
3. Traverse the right subtree i.e. call Inorder (right-subtree)

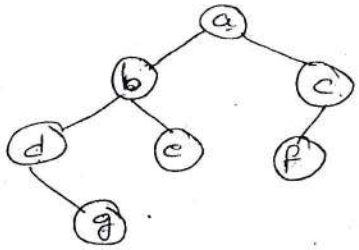
3. Postorder Traversal.

In the postorder traversal, the root is visited after visiting the left and right subtrees (in that order).

Pseudocode

1. Traverse the left subtree i.e. call Postorder (left-subtree).
2. Traverse the right subtree i.e. call Postorder (right-subtree).
3. Visit the root.

Example



Preorder: a, b, d, g, e, c, f. ~~gLR~~ +LR

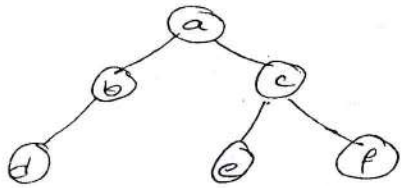
Inorder: d, g, b, e, a, f, c. ~~L+~~ L+R

postorder: g, d, e, b, f, c, a. LR+

Example

Discuss the following binary tree traversal

a) Preorder b) inorder c) Postorder

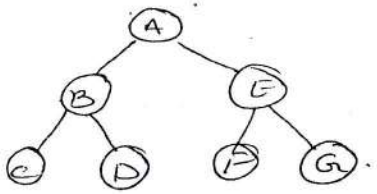


Preorder: a, b, d, c, e, f.

Inorder: d, b, a, e, c, f.

postorder: d, b, e, f, c, a.

Example



Preorder: A, B, C, D, E, F, G.

Inorder: C, B, D, A, F, E, G.

Postorder: C, D, B, F, G, E, A.

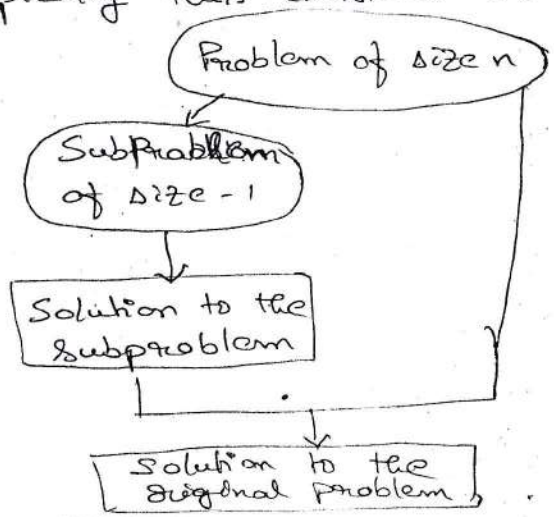
UNIT - II

DECREASE-AND-CONQUER

- The decrease-and-conquer technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem.
- It can be exploited either top down (recursively) or bottom up (without a recursion)
- Three major variations of decrease-and-conquer
  1. Decrease by a constant.
  2. Decrease by a constant factor
  3. Variable size decrease.

Decrease-by-a constant.

- In this variation, the size of an instance is reduced by the same constant on each iteration of the algorithm.
- Typically this constant is equal to one.



- ex. The exponentiation problem of computing a^n for positive integer.
- The relationship between a solution to an instance of size n and an instance of size n-1.
  i.e. $a^n = a^{n-1} \cdot a$.
- The function $f(n) = a^n$ can be computed either top-down by using its recursive definition

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 1 \\ a & \text{if } n = 1 \end{cases}$$

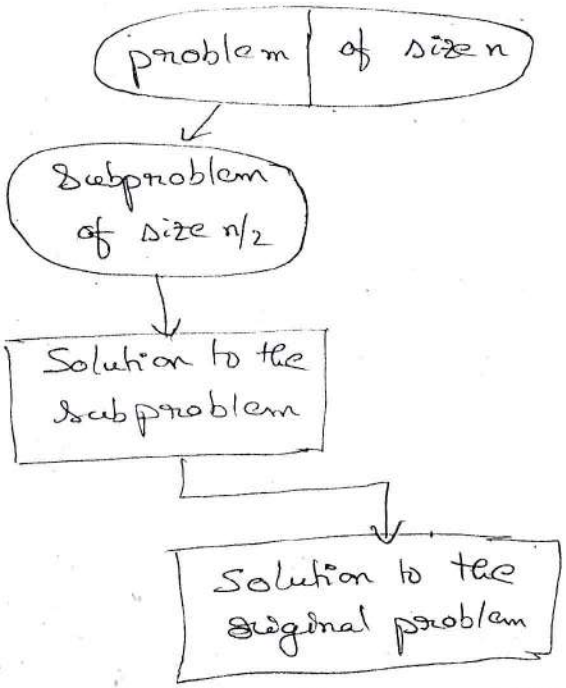
or bottom-up by multiplying a by itself n-1 times.

Decrease-by-a-constant-factor.

- This technique suggests reducing a problem instant by the same constant factor on each iteration of the algorithm
- In most applications this constant factor is equal to two.
  ex. Exponentiation problem.
- If the instance of size n is to compute a^n, the instance of half its size will be to compute $a^{n/2}$.

$$a^n = (a^{n/2})^{1/2}$$

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



- If we compute a^n recursively then efficiency $= O(\log n)$.

Based on Divide-and-conquer.

$$a^n = \begin{cases} a^{(n/2)} \cdot a^{(n/2)} & \text{if } n > 1 \\ a & \text{if } n = 1 \end{cases}$$

Variable-size-decrease.

- In this variation, a size reduction pattern varies from one iteration of an algorithm to another.

ex. $\gcd(m, n) = \gcd(n, m \bmod n)$

- Though the arguments on the right-hand side are always smaller than those on the left-hand side.

1. INSERTION SORT

- This is an application of the decrease-by-one-technique to sorting an array $A[0..n-1]$.
- We need to find an appropriate position for $A[n-1]$ among the sorted elements and insert it there.
- Three reasonable alternatives.

1. We can scan the sorted subarray from left to right until the first element greater than or equal to $A[n-1]$ is encountered, and then insert $A[n-1]$ right before that element.
2. We can scan the sorted subarray from right to left until the first element smaller than or equal to $A[n-1]$ is encountered and then insert $A[n-1]$ right after that element.

- These two alternatives are essentially equivalent, usually, it is the second one that is implemented in practice.
- The resulting algorithm is called <u>straight insertion sort</u> or simply <u>insertion sort</u>.

3. To use binary search to find an appriate position for $A[n-1]$ in the sorted portion of the array.

   The resulting algorithm is called <u>binary insertion sort</u>.

<u>Algorithm</u>: InsertionSort($A[0..n-1]$)

```
// Sorts a given array by insertion sort
// Input: An array A[0..n-1] of n
// orderable elements.
// Output: Array A[0..n-1] sorted in
// nondecreasing order.
for i ← 1 to n-1 do
    v ← A[i]
    j ← i-1
    while j ≥ 0 and A[j] > v do
        A[j+1] ← A[j]
        j ← j-1
    A[j+1] ← v
```

<u>Example</u>: 89, 45, 68, 90, 29, 34, 17.

	0	1	2	3	4	5	6
	$^{j}89$ |	$^{i}45$	68	90	29	34	17.
$i=1$ to 6. $j=0, v=45$ $j=-1$	45	$^{j}89$ |	$^{i}68$	90	29	34	17.
$i=2, v=68$ $j=1, j=0$ $A[j] \not> v$.	$^{j}45$	68	$^{i}89$ |	90	29	34	17.
$i=3, v=90$	45	68	$^{j}89$ |	$^{i}90$	29	34	17.
$i=4, v=29$ $j=3, j=2$	45	68	89	$^{j}90$ |	$^{i}29$	34	17.
$j=1, j=1,$ $j=0, j=-1$	45	68	$^{j}89$	29	$^{i}90$	34	17
	45	$^{j}68$	29	89	$^{i}90$	34	17.
	$^{j}45$	29	68	89	$^{i}90$	34	17.
	29	45	68	89	$^{j}90$ |	$^{i}34$	17
$i=5, v=34$	29	45	68	$^{j}89$	90	$^{i}90$	17.
$j=4, j=3, j=2,$ $j=1, j=0$ $A[j] \not> v$	29	45	$^{j}68$	34	89	$^{i}90$	17.
	29	$^{j}45$	34	68	89	$^{i}90$	17
	$^{j}29$	34	45	68	89	$^{j}90$ |	$^{i}17$.
$i=6, v=17$ $j=5$	29	34	45	68	$^{j}89$	17	$^{i}90$
	29	34	45	$^{j}68$	17	89	$^{i}90$
$j=-1$	29	34	$^{j}45$	17	68	89	$^{i}90$.
	29	$^{j}34$	17	45	68	89	$^{i}90$
	$^{j}29$	17	34	45	68	89	$^{i}90$
	17	29	34	45	68	89	90.

<u>Example</u>

1. 5, 2, 4, 6, 1, 3.

Efficiency

Basic operation : $A[j] > v$

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

$$= \sum_{i=1}^{n-1} i$$

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

$$C_{best}(n) = \sum_{i=1}^{n-1} 1$$

$$= n-1 \in \Theta(n)$$

$$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n)$$

Ex. $A = (3, 41, 59, 26, 41, 58)$

2. Depth-First Search and Breadth-First Search.

- These are the applications of the decrease-by-one technique.
- DFS & BFS are the important graph algorith
- Graph → Undirected, Directed & weighted grap
- Two principal representations of a graph are adjacency matrix & adjacency list.
- Many graph algorithms require processing vertices or edges of a graph in a systematic fashion.
- There are two principal algorithms.
  1. Depth-first Search (DFS)
  2. Breadth-first Search (BFS)

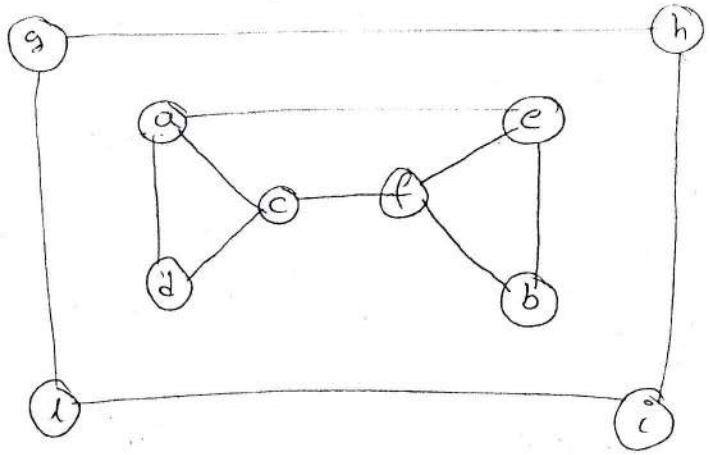
Depth-First-Search.

- DFS starts visiting vertices of a graph at an arbitrary vertex by making it as having been visited.
- On each iteration, the algorithm proceeds to an unvisited vertex that is adjacent to the one it is currently in.
- If there are several such vertices, a tie can be resolved arbitrarily.
- We will always break ties by the alphabetic order of the vertices.

- This process continues until a dead end - a vertex with no adjacent unvisited vertices - is encountered.
- At a dead end, the algorithm backs up one edge to the vertex it came from and tries to continue visiting unvisited vertices from there.
- The algorithm eventually halts after backing up to the starting vertex, with the latter being a dead end.
- It is convenient to use a stack to trace the operation of DFS.
- We push a vertex into the stack when the vertex is reached for the first time (i.e. the visit of the vertex starts), ~~and~~
- we pop a vertex off the stack when it becomes a dead end (i.e. the visit of the vertex ends).

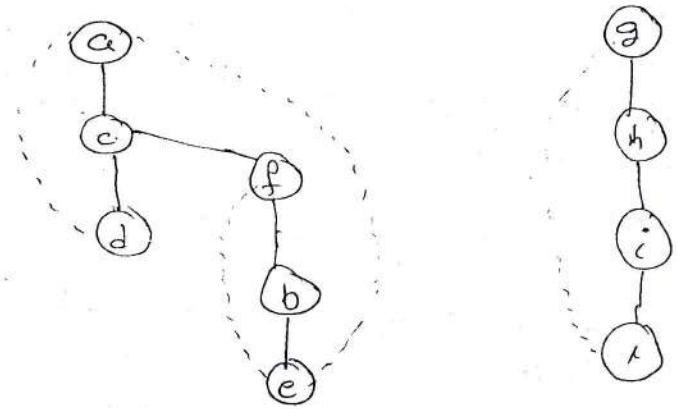
<u>Algorithm</u> DFS(j)

```
// Implements a depth-first-search traversal
// of a given graph.
// Input: Graph G = <V, E>
// Output: Graph G with its vertices marke
// with consecutive integers in the order
// they've been first encountered by the
// DFS traversal
Mark each vertex in V with 0 as a
mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
        dfs(v)

dfs(v)
// visits recursively all the unvisited
// vertices connected to vertex v by a path
// and numbers them in the order they
// are encountered via global variable cou
count ← count+1;
mark v with count.
for each vertex w in V adjacent to v
    do. if w is marked with 0
        dfs(w).
```

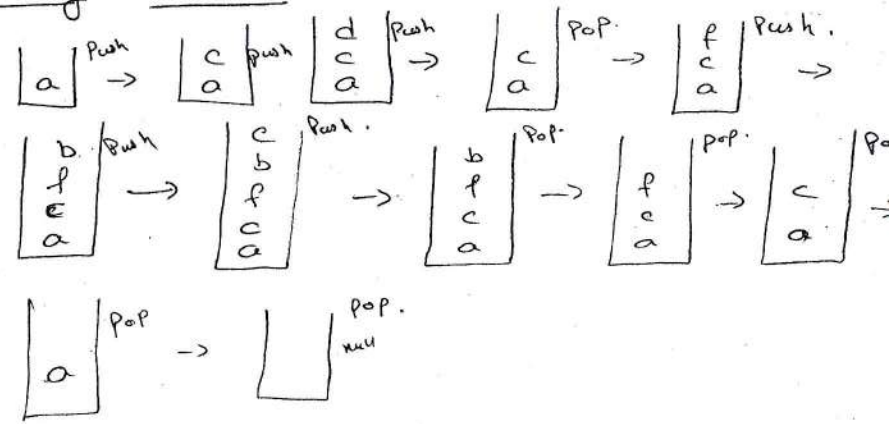


1. Starts with some vertex a.
2. I have 3 options i.e. c, d, e. and I have chosen c.
3. I go to c further, I have 2 options f and d and I have chosen d.
4. I go to d and only possibility is a.
5. I will keep on circle.
5. I can't go to a because a is a ancestor of d.
6. So backtrack on c.
7. I can't go back to a i.e parent vertex.
8. I can't go to d because that where I have gone.
9. I choose f. I have 2 options b, e.
10. I go to b. then I go to e.
11. After, I have 3 options a, b, f.
12. b and f are parent vertices. I have a dead end of this graph.
13. So I will draw two edges through parent f and ancestor a through dashed lines
14. I have to go one step back goto b and go back to f and go back to c and go back to a.

From a I go will go out of the graph



Using Stack.



a 0 1
c 0 2
d 0 3
f 0 4
b 0 5
e 0 6
g 0
h 0
i 0
l 0

1. Initiall start with a, increment count ~~as~~ now a = 1
2. Adjacent of a is c. and is marked as 0. so increment count

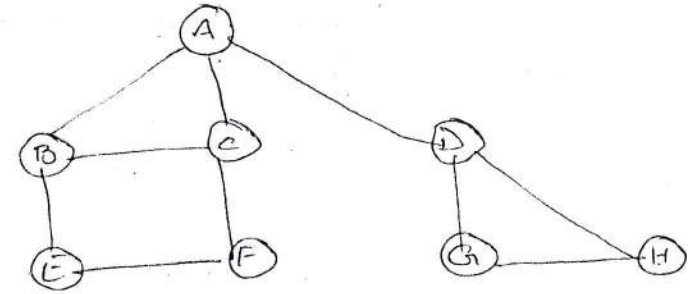
a
|
c

3. adjacent of e is d and is marked as 0. So increment count

a
|
c
|
d

4. Adjacent vertex of d is a, c and is not marked with 0.
So applying back track i.e. go back one step. i.e. c.
5. Adjacent of c is f is ~~not~~ marked with 0.
So increment count.
5 Adjacent of f is b is marked with 0.
So increment count.
6. Adjacent of b is e is marked with 0.
So increment count.

Example.



Breadth-First Search:

- It is a traversal for the cautious.
- It proceeds in a concentric manner by visiting first all the vertices that are adjacent to a starting vertex, then all revisited vertices two edges apart from it, and so on, until all the vertices in the same connected component as the starting vertex are visited.
- If there are still remain unvisited vertices, the algorithm has to be restarted at an arbitrary vertex of another connected component of the graph
- It is convenient to use queue to trace the operation of BFS.
- The queue is initialized with the traversal's starting vertex, which is marked as visited.

- On each iteration, the algorithm identifies all unvisited vertices that are adjacent to the front ~~vertices~~ vertex, marks them as visited, and adds them to the queue after that, the front vertex is removed from the queue.

Algorithm. BFS (g)

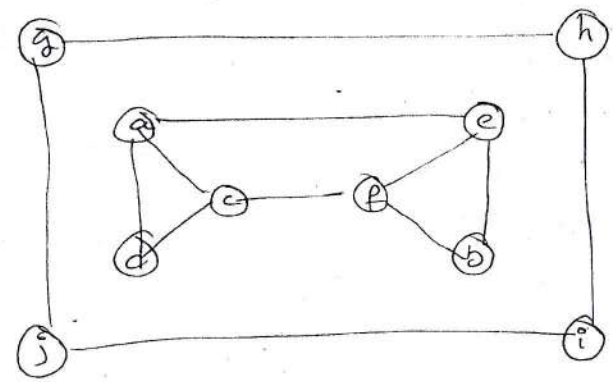
// Implements a bfs traversal of a given
// graph.
// Input: Graph $G = \langle V, E \rangle$
// Output: Graph G with its vertices marked
// with consecutive integers in the order
// they have been visited by the BFS
// traversal

mark each vertex in v with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
        bfs(v).

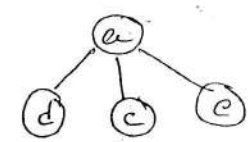
bfs(v).

// visits all the unvisited vertices connected
// to vertex v by a path and assigns
// them the numbers in the order they are
// visited via global variable count.

count ← count + 1
mark v with count and initialize a queue with v.
while the queue is not empty do
    for each vertex w in v adjacent to the front vertex do
        if w is marked with 0
            count ← count + 1
            mark w with count.
            add w to the queue.
    remove the front vertex from the queue.

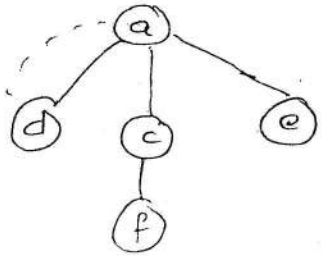


Start with vertex (a).
List down all the children of a.

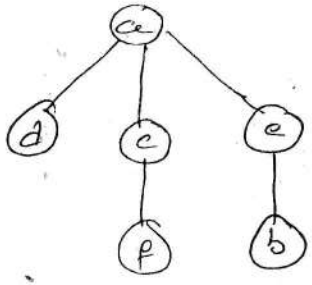


Then checking all the children
d has a, c childrens, already lies

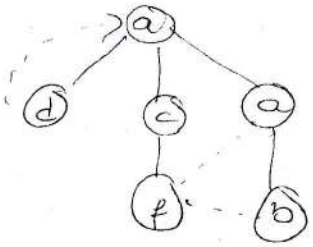
Go back to a.
next child of c. list of all children
a, c, f.
a and c are already present
so we only present f.

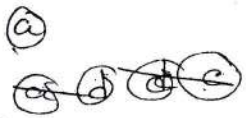


we move on next child of a is c.
List out children a, f, b..
a and f are already present The
only availability is b.



Then we go to next level i.e f
f. List the children.
All are already listed. so we put
dotted lines to e and b
'g b also





drop a

Using Algorithm

a	0	1
c	0	2
d	0	3
e	0	4
f	0	5
b	0	6
g	0	
h	0	
j	0	
i	0	

List down all the vertices of a graph and mark with 0.

Let us start with a.

Queue.
a. -> Queue is not empty.

adjacencent of a is c. increment count
add c to queue.

a c. -> adjacent of c is d and increment count and add queue

a c d

e is adjacent to a. and add to queue

a c d e

Now check any body left whose a of child of a is connected. No body else.
So. I drop a.

c d e

I will go up while queue is not empty. Take the first element i.e. c & find adjacency then add f to queue

c d e f.

No check is there any body left over. No baby else. So I drop c.

d e f

Now queue is not empty:
Now Take d and adjacent to d are a and c those are not marked with 0. So drop d.

e f

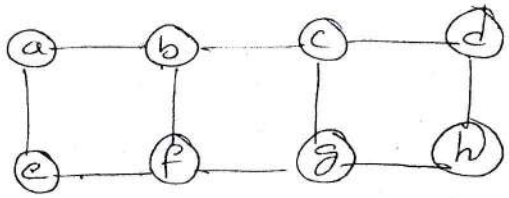
Next. e and adjacent are a, f, b and b is marked with 0 so add b to queue

e f b

No check is there any body left
Then e is drop
Next f is drop. and next drop b.

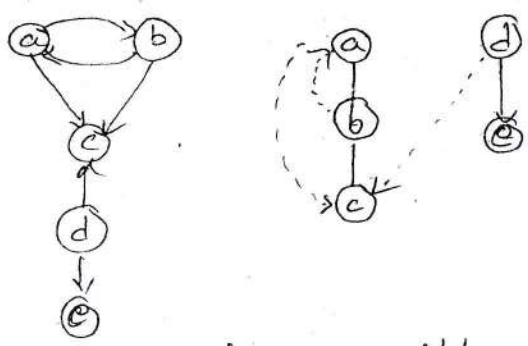
Main Facts about DFS & BFS

	DFS	BFS
1. Data Structure	stack	Queue
2. No. of vertex orderings	2 orderings	1 ordering
3. Edge Types	Tree and back edges	Tree and cross edges
4. Applications	connectivity, acyclicity, articulation points	connectivity, acyclicity, minimum-edge points
5. Efficiency for adjacent matrix.	$\Theta(\lvert V\rvert^2)$	$\Theta(\lvert V\rvert^2)$
6. Efficiency for adjacent list.	$\Theta(\lvert V\rvert+\lvert E\rvert)$	$\Theta(\lvert V\rvert+\lvert E\rvert)$



Topological Sorting

- A directed graph or digraph is a graph with directions specified for all its edges.
- A digraph can be represented by adjacency matrix or adjacency lists.
- Difference between undirected and directed graph are
  1. The adjacency matrix of a directed graph does not have to be symmetric.
  2. An edge in a directed graph has just one (not two) corresponding nodes in the digraph's adjacency lists.
- DFS and BFS search are principal traversal algorithms for traversing digraphs, but the structure of corresponding facts can be more complex.



- 4 types of edges possible in a DFS forest of a directed graph.

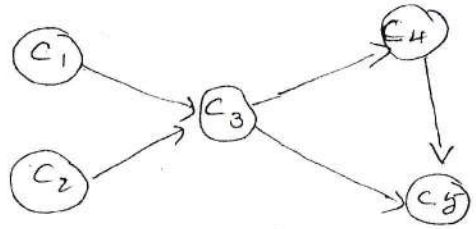
1. Tree edges - ab, bc, de.
2. back edges - ba (3). Forward edges - ac
3. cross edges - dc

- The presence of a back edge indicates that the digraph has a directed cycle.
- A <u>directed cycle</u> in a digraph is a sequence of three or more of its vertices to its immediate predecessor by an edge directed from the predecessor to the successor.
- If a DFS forest of a digraph has no back edges, the digraph is a <u>dag</u> i.e. directed acyclic graph.
- If we have a directed graph G then topological sorting/ordering of the graph is a "linear" order of the vertices of the graph such that if "uv" is an edge in the graph then "u" should come before "v" in the topological ordering.

<u>Example</u>

- Consider a set of 5 required courses [c_1, c_2, c_3, c_4, c_5] a part-time student has to take in some degree program.
- The courses can be taken in any order as long as the following course prerequisites are met.

- c1 and c2 have no prerequisites, c3 requires c1 and c2, c4 requires c3, and c5 requires c3 and c4.
- The student can take only one course per term.
- In which order should the student take the course.



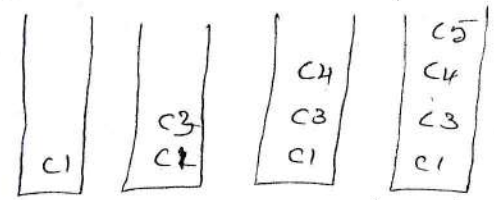
- Whether we can list its vertices in such an order that for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends.
- This problem is called <u>topological sorting</u>
- Topological sorting to be possible, a digraph must be a <u>dag</u>
  i.e If a digraph has no cycles, the topological sorting problem for it has a solution.
- There are <u>two</u> efficient algorithms that both verify whether a digraph is a dag and, if it is, produce an order of vertices that solves the topological sorting problem.

<u>Algorithm 1</u>

Topological sorting using DFS

<u>Stack</u>

start with c1



Then I have dead and then pop elem

c5 c4 c3 c1

- The one vertex which is not in the stack and put in stack.

- Then pop the element and put into list-stack

∴ Stack is

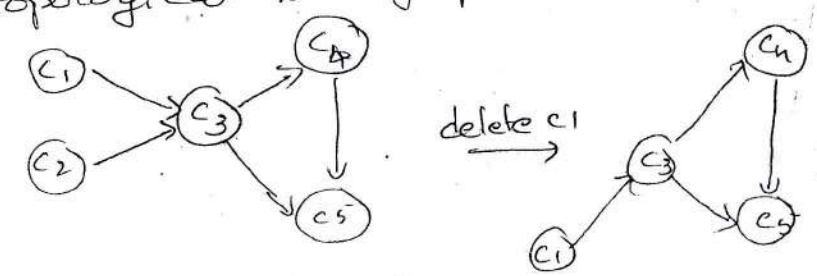
c5 c4 c3 c1 c2.

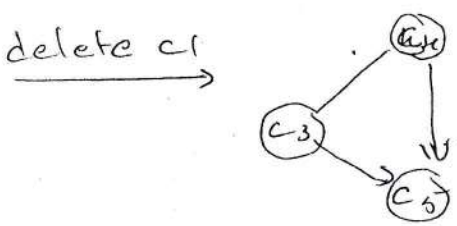
- Reversing this order yields a solution to the topological sorting problem.

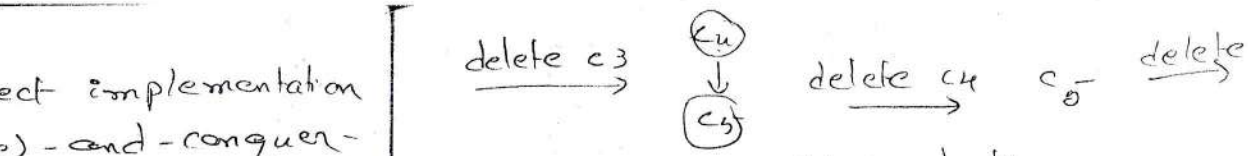
c2 c1 c3 c4 c5

Algorithm 2

- It is based on a direct implementation of the decrease (by one) - and - conquer - technique.
- Topological sorting using Source Removal
- Repeatedly identify in a remaining digraph a <u>source</u>, which is a vertex with no incoming edges, and delete it along with all the edges outgoing from it.
- If there are several sources, break the tie arbitrarily.
- If there is none, stop because the problem can't be solved.
- The order in which the vertices are deleted yields a solution to the topological sorting problem.



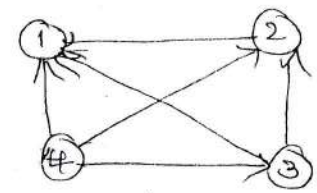




The solution obtained is

c1, c2, c3, c4, c5.

Prob ① Apply the DFS-based algorithm to solve the topological sorting problem for the following graph.



Decrease-by-a-Constant Factor Algorithm

- This technique suggests reducing a problem's instant by the same constant factor on each iteration of the algorithm.

Example: Binary search.

Fake-Coin Problem:

- Among n identically looking coins, one is fake.
- With a balance scale, we can compare any two sets of coins.
- That is, by tipping to the left, to the right, or staying even, the balance scale will tell whether the sets weigh the same or which of the sets is heavier than the other but not by how much.
- The problem is to design an efficient algorithm for detecting the fake coin.
- The most natural idea for solving this problem is to divide n coins into two piles of $\lfloor n/2 \rfloor$ coins each, leaving one extra coin apart if n is odd, and put the two piles on the scale.
- We assume that the fake coin is lighter.
- If the piles weigh the same, the coin put aside must be fake.
- Otherwise, we can proceed in the same manner with the lighter pile, which must be the one with the fake coin.
- Even though we divide the coins into two subsets, after one weighing we are left to solve a single problem of half the original size.

DAA

Jyothi

Decrease-by-a-Constant Factor Algorithm

- This technique suggests reducing a problem's instant by the same constant factor on each iteration of the algorithm.

Example. Binary Search.

1. Fake-Coin Problem

- Among n identically looking coins, one is fake.
- with a balance scale, we can compare any two sets of coins.
- That is, by tipping to the left, to the right, or staying even, the balance scale will tell whether the sets weigh the same or which of the sets is heavier than the other but not by how much.
- The problem is to design an efficient algorithm for detecting the fake coin.
- The most natural idea for solving this problem is to divide n coins into two piles of $\lfloor n/2 \rfloor$ coins each, leaving one extra coin apart if n is odd, and put the two piles on the scale.
- we assume that the fake coin is lighter.
- If the piles weigh the same, the coin put aside must be fake.
- Otherwise, we can proceed in the same manner with the lighter pile, which must be the one with the fake coin.
- Even though we divide the coins into subsets, after one weighing we are to solve a single problem of half the original size.
- It is a decrease(by half)-and-conquer rather than divide-and-conquer algorithm.

$\therefore$ No of weights $W(n)$ is.

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

$$W(1) = 0$$

$$\therefore W(n) = \lceil \log_2 n \rceil$$

2. Multiplication a la Russe

- Now consider a nonorthodox algorithm for multiplying two positive integer called multiplication a la russe or the Russian peasant method.
- Let n and m be positive integers whose product we want to compute.
- The size of the integer is n.

- If n is even, an instance of half the size has to deal with $n/2$

$$\therefore n \cdot m = \frac{n}{2} \cdot 2m$$

- If n is odd, we need only a slight adjustment of this formula

$$n \cdot m = \frac{n-1}{2} \cdot 2m + m$$

- Using these formulas and the trivial case of $1 \cdot m = m$ to stop.
- We can compute product $n \cdot m$ either recursively or iteratively.

ex. computing $50 \cdot 65$

n	m
50	65
25	130
12	260 (+130)
6	520
3	1,040
1	2,080 (+1040)
-	2,080 + (130 + 1040) = 3,250

n	m	
50	65	
25	130	130
12	260	
6	520	
3	1040	1040
1	2080	2080
		3,250.

We can find the product by simply adding all the elements in the m column that have an odd number in the n column.

3. Josephus Problem.

- Flavius Josephus, a famous Jewish histor who participated in and chronicled the Jewish revolt of 66-70 C.E against the Romans.
- Josephus, as a general, managed to hold the fortress of Jotapata for 47 days, but after the fall of the city he took refuge wi 40 diehards in a nearby cave.
- Therefore, the rebels voted to perish rat than surrender.
- Josephus proposed that each man in th should dispatch his neighbor, the order t be determined by casting lots.
- Josephus contrived to draw the last lot, and, as one of the two surviving men in the cave, he prevailed upon his intended victim for surrender to the Romans.
- So let n people numbered 1 to n stan in a circle.
- Starting the grim count with person numb 1.
- We eliminate every second person unt only one survivor is left.

- In this problem is for $k = \lceil n/2 \rceil$ which asks to find an element that is greater than one half of the list's elements and smaller than the other half.
- This middle value is called the <u>Median</u>
- We have an efficient algorithm for partitioning an array's elements into two subsets.
- The one containing the elements that are less than or equal to some value p pivoting the partition and the other containing the elements that are greater than or equal to p.

$$\underbrace{a_{i1} \cdots a_{i\,s-1}}_{\le p}\ p\ \underbrace{a_{i\,s+1} \cdots a_{in}}_{\ge p}$$

- Let s be the partition's split position
- If $s = k$, the pivot p obviously solves the selection problem.
- If $s > k$, the kth smallest element in the entire list can be found as the kth smallest element in the left part of the partitioned array.
- If $s < k$, we can proceed by searching for the $(k-s)^{th}$ smallest element in its right part.
- This process repeated recursively.

- For nonrecursive version, we need not even adjust the value of k but just continue until $s = k$.

Find the median of the following list of 9 numbers

4, 1, 10, 9, 7, 12, 8, 2, 15

Here $k = 9/2 = 5$.

- Our task is to find the 5th smallest element in the array.
- For convenience that the elements of the list are indexed from 1 to 9.
- We Quick Sort
- which selects the first element as a pivot and rearranges elements by two oppositely directed scans of the array.

$\boxed{4}$ 1, 10, 9, 7, 12, 8, 2, 15

4, 1, 2, 9, 7, 12, 8, 10, 15

2, 1, 4, 9, 7, 12, 8, 10, 15

Since $s = 3 < k = 5$

We proceed with the right part of the list

$\boxed{9}$, 7, 12, 8, 10, 15

9 7 8 12 10 15

8 7 9 12 10 15

Since $s = 6 > k = 5$, we continue with the left part of the previous sublist

$\boxed{8}$ 7

7 8

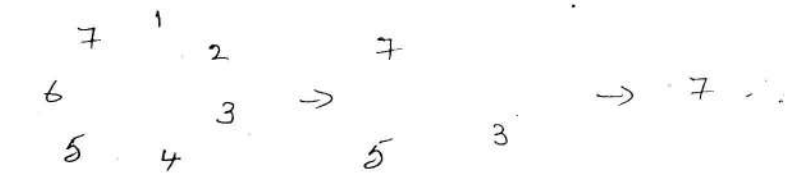
- The problem is to determine the survivor's number $J(n)$.

Example:

- If n is 6, people in position 2, 4 and 6 will be eliminated on the first pass through the circle
- People in initial positions 3 and 1 will be eliminated in the second pass.
- Leaving a sole survivor in initial position 5, thus $J(6) = 5$.

Example

- If $n = 7$, people in positions 2, 4, 6 and 1 will be eliminated on the first pass.
- People in position 5 and, for convenience 3 on the second pass.
- Thus $J(7) = 7$



- If n is even i.e $n = 2k$, the first p through the circle yields an instanc exactly the same problem but half initial size.

$$\therefore J(2k) = 2J(k) - 1$$

- If n is odd i.e. $n = 2k+1$, the first pass eliminates people in all even po
- If we add to this the elimination of the pers in position 1 right after that, we are le with an instance of size k.

$$\therefore J(2k+1) = 2J(k) + 1.$$

VARIABLE-SIZE-DECREASE ALGORITHMS

- In this algorithms the size reduction pattern varies from one iteration of th algorithm to another.

Examples

1. Computing a Median and the Selection Pro

- The Selection Problem is the problem finding the kth smallest element in a li of n numbers.
- This number is called the kth order statis
- For $k = 1$ or $k = n$, we can simply scan the in question to find the smallest or large element respectively.

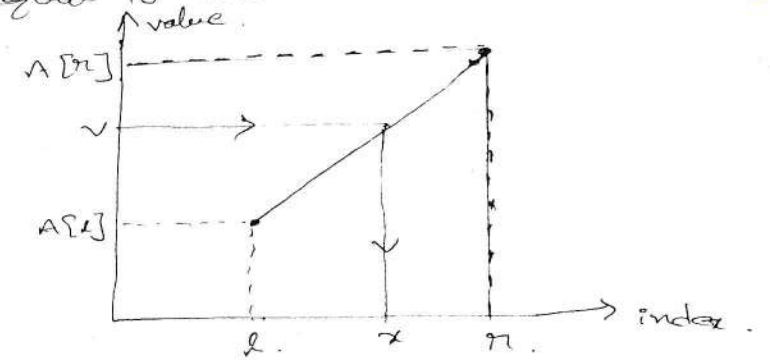
- Now $S = K = 5$, and hence we can stop.
- The found median is 8, which is greater than 2, 1, 4 and 7 but smaller than 9, 12, 10, and 15.

The number of comparisons would have been

$$C(n) = C(n/2) + (n+1)$$

2. Interpolation Search

- An algorithm for searching in a sorted array called Interpolation Search.
- This search takes into account the value of the search key in order to find the array's element to be compared with the search key.
- More precisely, on the iteration dealing with the array's portion between the leftmost element $A[l]$ and the rightmost element $A[r]$, the algorithm assumes that the array's values increase linearly i.e. along the straight line through the points $(l, A[l])$ and $(r, A[r])$.
- Accordingly, the search key's value v is compared with the element whose index is computed as the x coordinate of the point on the straight through the po $(l, A[l])$ and $(r, A[r])$, whose y coord is equal to the search value P. 3



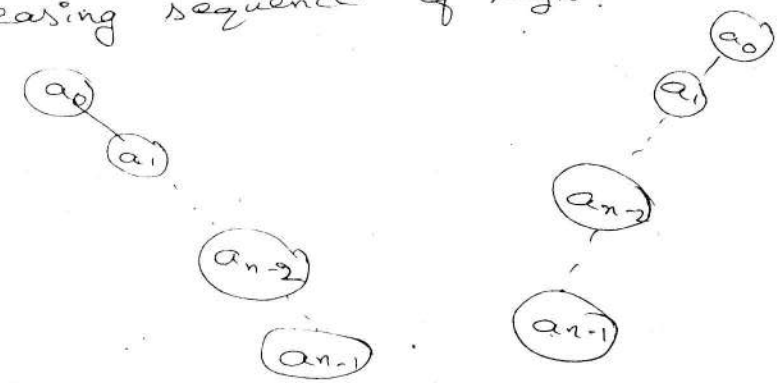
- Writing a standard equation for the st line passing through the points $(l, A$ and $(r, A[r])$, substituting v for y, an solving it for x leads to the follg.

$$x = l + \left[\frac{(v - A[l])(r - l)}{A[r] - A[l]} \right]$$

3. Searching and Insertion in a Binary Search Tree

- In a binary tree whose nodes contain elements of a set of orderable items one element per node, so that for every node all elements in the left subtree are smaller and all the elements in the right subtree are greater than the elements in the subtree's root.
- When we need to search for an element v of a given value in a tree we do it recursively in the following manner.
  1. If the tree is empty the search ends in failure.
  2. If the tree is not empty, we compare v with the tree's root $k(r)$.
     - If they match, a desired element is found the search can be stopped.
     - If they do not match, we continue with the search in the left subtree of the root if $v < k(r)$ and in the right subtree if $v > k(r)$
- Thus, on each iteration of the algorithm, the problem of searching in a binary search tree is reduced to searching in a smaller binary search tree.

- In the worst-case efficiency of the binary tree search, the tree is severely skewed.
- This happen if a tree constructed by successive insertion in increasing or decreasing sequence of keys.



- The worst-case efficiency of the search operation is $\Theta(n)$
- The average-case efficiency is $\Theta(\log n$

4. The Game of Nim

TRANSFORM - and - CONQUER.

- This technique deals with a group of design methods that are based on the idea of transformation.
- This general technique is called transform-and-conquer technique these methods work as two-stage procedures.

1. In transformation stage, the problem's instance is modified to be, for one reason to another, more amenable to solution.
2. Conquering stage, it is solved.

- There are 3 major variations.

1. Transformation to a simpler or more convenient instance of the same problem we call it instance simplification.
2. Transformation to a different representation of the same instance, we call it representation change
3. Transformation to an instance of a different problem for which an algorithm is already available, we call it problem reduction

Problems Instance → Simpler Instance or Another representation or Another problem instance → S

1. PRESORTING

- Presorting is an old idea in computer sc
- The time efficiency of algorithms that inv sorting may depend on the efficiency of sorting algorithm being used.

Example 1

Checking element uniqueness in an array

- This problem considered a brute-force alg
- The brute-force algorithm compared pairs the array's elements until either two eq elements were found or no more pairs we
- We can sort the array first and then che only its consecutive elements.
- If the array has equal elements, a pair them must be next to each other and vi

Algorithm: PresortElementUniqueness(A[0..n-1])

// Solves the element uniqueness problem b
// sorting the array first.
// Input: An array A[0..n-1] of orderable elem
// Output: Returns "true" if A has no equal elem
// "false" otherwise.

```
Sort the array A
for i ← 0 to n-2 do
    if A[i] = A[i+1] return false
return true
```

- The running time of this algorithm is the sum of the time spent on sorting and the time spent on checking consecutive elements.
- The worst-case efficiency of the entire presorting-based algorithm is $\Theta(n \log n)$

$$T(n) = T_{sort}(n) + T_{scan}(n)$$
$$\in \Theta(n \log n) + \Theta(n)$$
$$= \Theta(n \log n)$$

Example 2

Computing a mode

- A mode is a value that occurs most often in a given list of numbers.
- The brute-force approach to computing a mode would scan the list and compute the frequencies of all its distinct values, then find the value with the largest frequency. To implement this idea, we can store the values already encountered, along with their frequencies, in a separate list.
- On each iteration, the ith element of the original list is compared with the values already encountered by traversing this auxiliary list.
- If a matching value is found, its frequency is incrementing.
- Otherwise, the current element is added to the list of distinct values seen so far with a frequency of 1.

```
Algorithm PresortMode(A[0..n-1])
// Computes the mode of an array by sorting it fi
// Input: An array A[0..n-1] of orderable eleme
// Output: The array's mode
Sort the array A
i ← 0
modefrequency ← 0
while i ≤ n-1 do
    runlength ← 1
    runvalue ← A[i]
    while i+runlength ≤ n-1 and A[i+runlength]
                                = runvalue
        runlength ← runlength + 1
    if runlength > modefrequency
        modefrequency ← runlength
        modevalue ← runvalue
    i ← i + runlength
return modevalue
```

· The worst-case number of comparisons made by this algorithm is

$$C(n) = \sum_{i=1}^{n} (i-1)$$
$$= 0 + 1 + \cdots (n-1)$$
$$= \frac{n(n-1)}{2}$$
$$\in \Theta(n^2)$$

Example 3.

Searching Problem.

· Consider the problem of searching for a given value v in a given array of n sortable items.

· The brute-force solution is sequential search, which needs n comparisons in the worst case.

· If the array is sorted first, we can then apply binary search, which requires only $\lfloor \log_2 n \rfloor + 1$ comparisons in the worst case.

· Assuming the most efficient n log n sort, the total running time of such a searching algorithm in the worst case will be

$$T(n) = T_{sort}(n) + T_{search}(n)$$
$$= \Theta(n \log n) + \Theta(\log n)$$
$$= \Theta(n \log n)$$

2. BALANCED SEARCH TREES

· In a binary tree whose nodes contain elements of a set of orderable items one element per node, so that all el in the left subtree are smaller than the element in the subtree's root all the elements in the right subtree are greater than it.

· Computer scientists have expanded a of effort in trying to find a structure that preserves the good properties the classical binary search tree

· They have come up with two approa

1. The instance simplification variety

· An unbalanced binary search tree transformed to a balanced one.

· An AVL Tree requires the difference b the heights of the left and right subt of every node never exceed 1.

· A red-black tree tolerates the height one subtree being twice as large as other subtree of the same node.

· If an insertion and deletion of a node creates a tree with a violated balanc requirement, the tree is restructured by one of a family of special transform called rotations that restore the bal required.

2 The representation-change variety

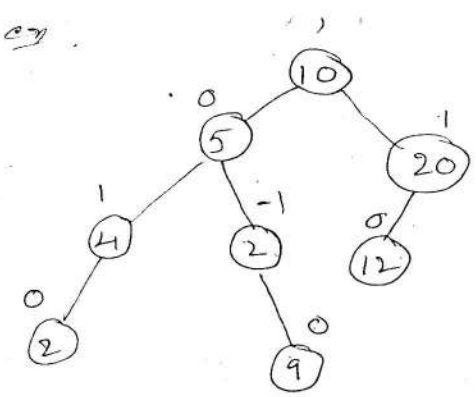
- It allow more than one element in a node of a search tree
- Specific cases of such trees are 2-3 trees, 2-3-4 trees, and more general and important B-trees.

AVL Trees

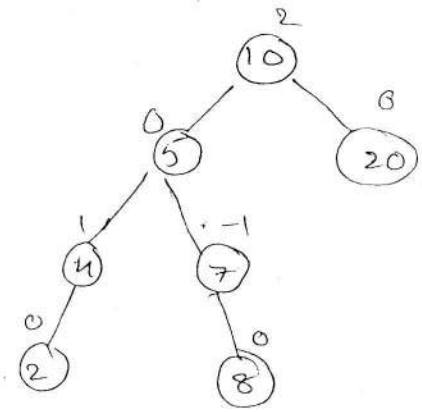
- AVL trees were invented in 1962 by two Russian scientists G. M. Adelson-Velsky and E. M. Landis.

Def A AVL tree is a binary search tree in which the balance factor of every node, which is defined as the difference between the heights of the node's left and right subtree, is either 0 or +1 or -1. (The height of the empty tree is -1)

eg.



AVL Tree

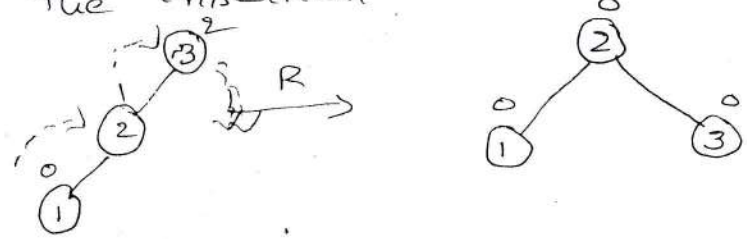


Not an AVL tree.

- If an insertion of a new node makes an AVL tree unbalanced, we transform the tree by a rotation.
- A rotation in an AVL tree is a local transformation of its subtree rooted at a node whose balance has become either +2 or -2, if there are several such nodes, we rotate the tree rooted at the unbalanced node that is the closest to the newly inserted leaf.
- There are only four types of rotations, infact two of them are mirror images of the other two.

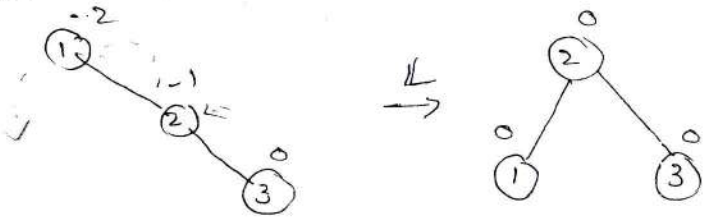
1. Single Right Rotation or R-rotation.

- Imagine rotating the edge connecting the root and its left child in the binary tree
- This rotation is performed after a new key is inserted into the left subtree of the left child of a tree whose root had the balance of +1 before the insertion



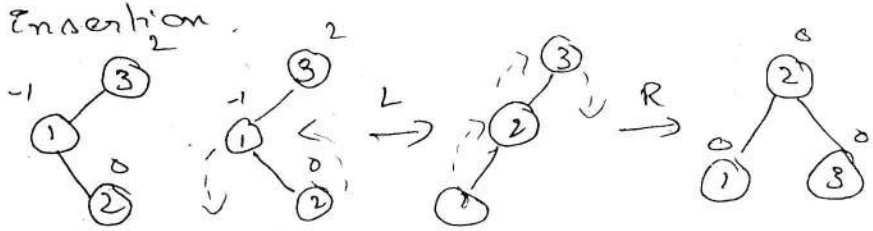
2. Single Left Rotation or L-rotation

- This rotation is the mirror image of the single R-rotation.
- It is performed after a new key is inserted into the right subtree of the right child of a tree whose root had the balance of -1 before the insertion.



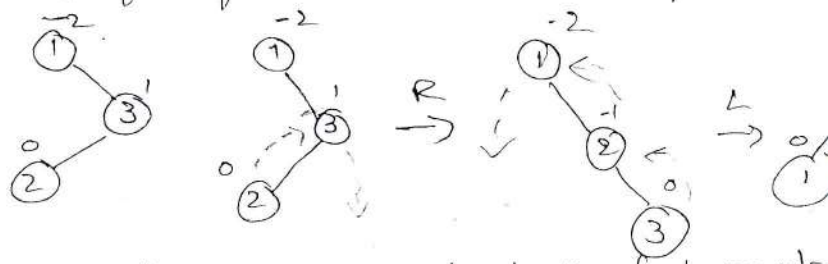
3. Double Left-Right Rotation (LR-rotation)

- It is a combination of two rotations.
- We perform the L-rotation of the left subtree of root r followed by the R-rotation of the new tree rooted at r.
- It is performed after a new key is inserted into the right subtree of the left child, of a tree whose root had the balance of +1 before the insertion.



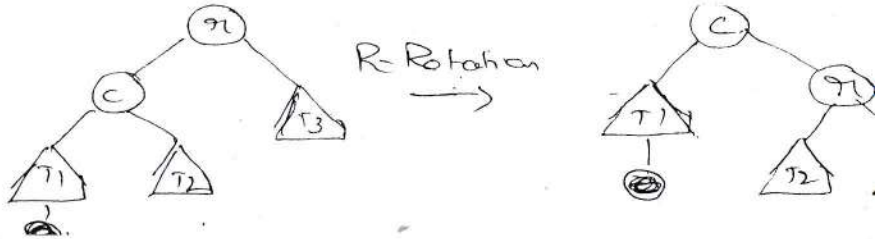
4. Double Right-Left Rotation (RL-rotat

- It is the mirror image of the doubl LR rotation.
- The RL rotation is combination o Single right rotation followed by s left rotation
- In RL rotation, first every node r one position to right then one po to left from the current positi



- The rotations are not trivial transfo Though, fortunately, they can be done constant time.
- Not only should they guarantee that resulting tree is balanced, but te should also preserve the basic requirements of a binary search

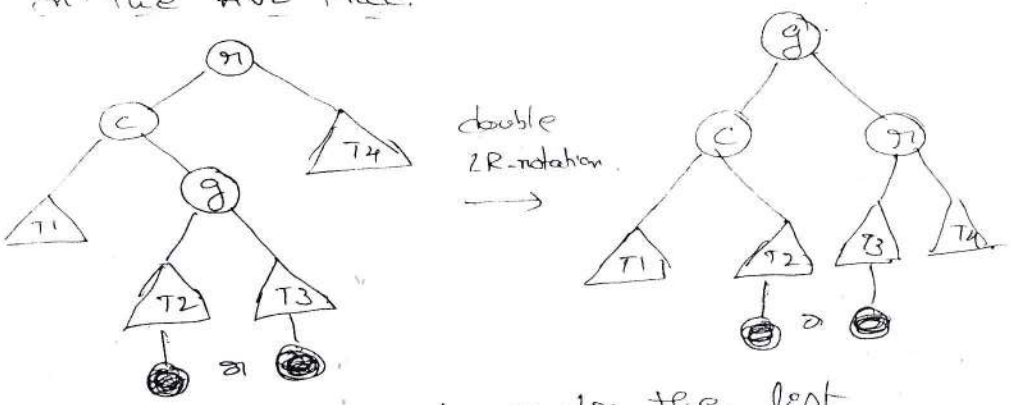
General form of the R-rotation in AV



All the keys of subtree T1 are smaller than c, which is smaller than all the keys of subtree T2, which are smaller than a, which is smaller than all the keys of subtree T3.

The same relationships among the key values hold, as they must, for the balanced tree after the rotation.

General form of the double LR-rotation in the AVL tree.

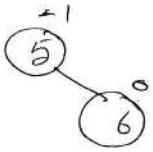


Construct an AVL tree for the list

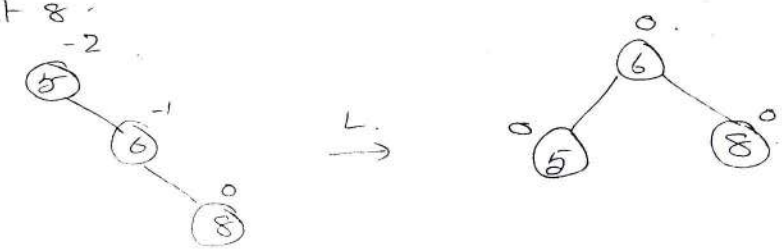
5, 6, 8, 3, 2, 4, 7

insert 5

insert 6.

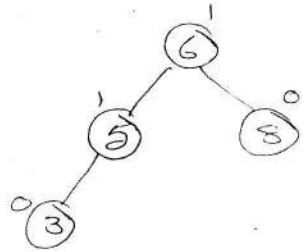


insert 8.

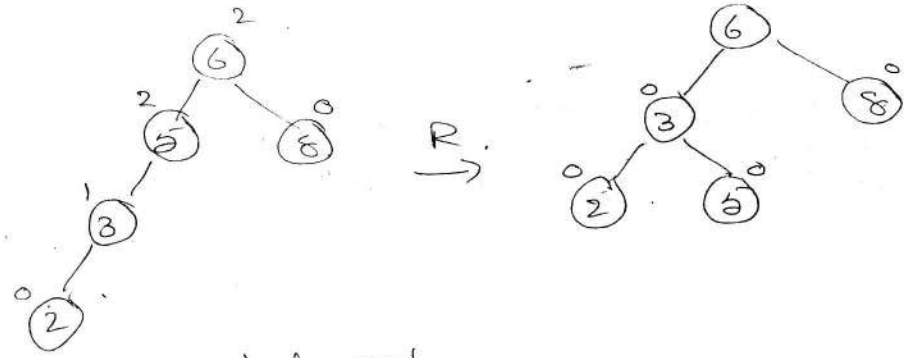


This is unbalanced

Insert 3.

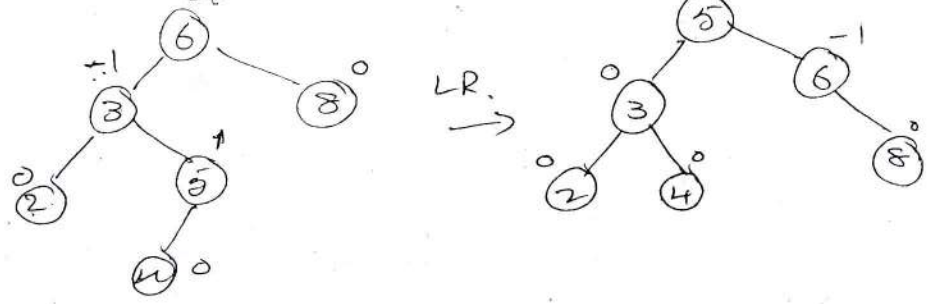


insert 2

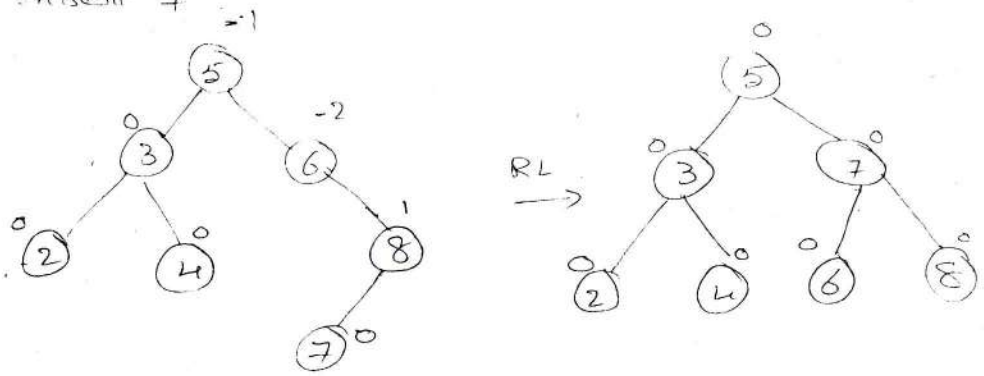


This is unbalanced

Insert 4



Insert 7



Worst case complexity of searching and insertion are $\Theta(\log n)$

Problem.

For each of the following lists, construct an AVL tree by inserting their elements successively, starting with the empty tree

a) 1, 2, 3, 4, 5, 6 b) 6, 5, 4, 3, 2, 1 c) 3, 6, 5, 1, 2, 3

2-3 Trees

- 2-3 trees introduced by U.S. computer scientist John Hopcroft in 1970.
- A 2-3 tree is a tree that can have nodes of two kinds: 2-nodes and 3 nodes.
- A 2-node contains a single key k and has two children, the left child serves as the root of a subtree whose keys are less than k and the right child serves as the root of a subtree whose keys are greater than k.

A 3-node contains two ordered keys k1 and k2 (k1 < k2) and has 3 children

- The leftmost child as the the root of a subtree with keys less than k1, and the middle child serves as the r of a subtree with keys between k1 a and the right child serves as the r of a subtree with keys greater than

2-node

- The last requirement of the 2-3 tree that all the leaves must be on the sa level
- 2-3 tree is always perfectly height-bala the length of a path from the root the tree to a leaf must be the same every leaf
- Searching for a given key in a 2-3 tre is quite straightforward
- We start at the root.
- If the root is a 2-node, we if it a binary search tree, we either stop

k is equal to the root's key or continue the search in the left or right subtree if k is, respectively, smaller or larger than the root's key. If the root is a 3-node, if k is equal to one of the root's keys, the search can be stoped or in which of the which of the root's three subtrees it needs to be continued

Inserting a new key in a 2-3 tree

1. We always insert a new key k in a leaf, except for the empty tree.
2. The appropriate leaf is found by performing a search for k.
3. If the leaf in question is a 2-node, we insert k there as either the first or the second key, depending on whether k is smaller or larger than the node's old key.
4. If the leaf is a 3-node, we split the leaf into two.
   - The smallest of the three keys is put in the first leaf.
   - The largest key is put in the second leaf.
   - The middle key is promoted to the old leaf's parent.

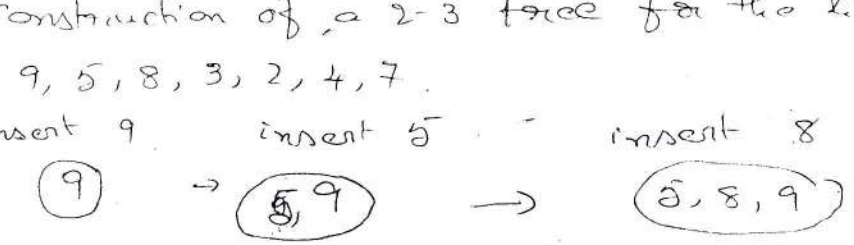
   (if the leaf happens to be the tree's root a new root is created to accept the middle key)

Note

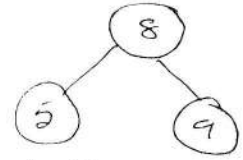
Promotion of a middle key to its parent can cause the parent's overflow (if it was a 3-node) and hence can lead to several node splits along the chain of the leaf's ancestors.

Construction of a 2-3 tree for the list 9, 5, 8, 3, 2, 4, 7.

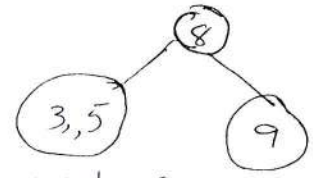
insert 9 → insert 5 → insert 8



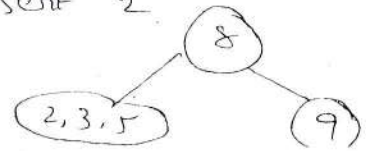
insert 3, Node contain 3 keys so split it



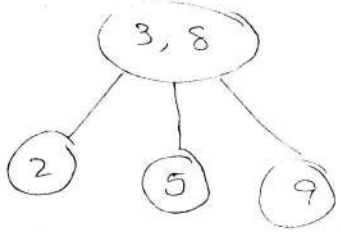
insert 3.



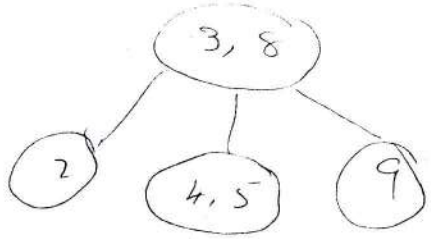
insert 2



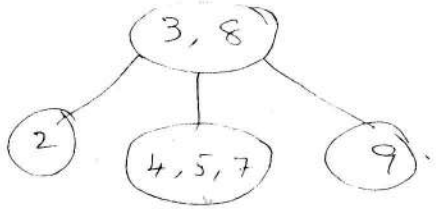
Leaf node contain 3 keys. So split it



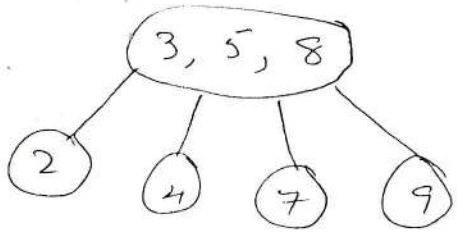
Insert 4.



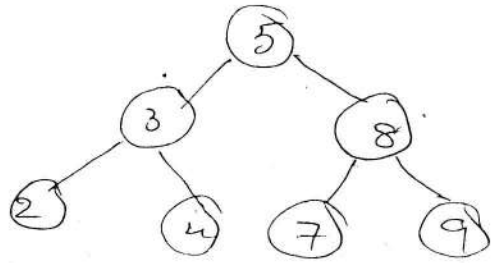
Insert 7



Leaf node contain 3 keys. So split it.



The root node contain 3 keys So split it



The time efficiencies of searching, insertion, and deletion are all in $\Theta(\log n)$ in both the worst and aver case

Prob

construct a 2-3 tree for the list.

C, O, M, P, U, T, E, R

3. HEAPS and HEAPSORT

Heap

Definition

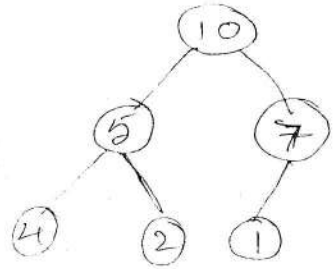
A heap can be defined as a binary which keys assigned to its nodes (one ke node) provided the following two cond are met.

1. The tree's shape requirement

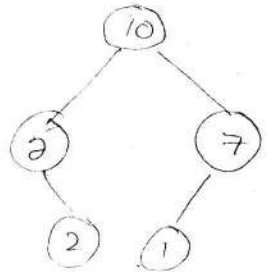
The binary tree is essentially complet (complete), that is, all its levels are fu except possibly the last level, where o some rightmost leaves may be missi

2. The parental dominance requirement.

The key at each node is greater than equal to the keys at its children.



Heap tree



Not Heap tree

Properties of Heaps

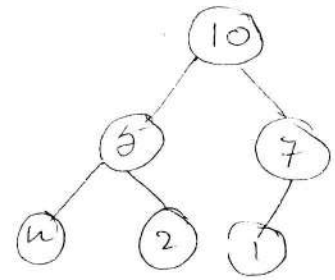
1. There exists exactly one essentially complete binary tree with n nodes. Its height is equal to $\lfloor \log_2 n \rfloor$.
2. The root of a heap always contains the largest element
3. A node of a heap considered with all its descendants is also a heap
4. A heap can be implemented as an array by recording the elements in the heap top-down, left-to-right fashion. It is convenient to store the heap's elements in position 1 through n of such an array, leaving H[1] either unused or putting there a sentinel whose value is greater than every element in the heap.

In such a representation

a) the potential node keys will be in the first $\lfloor n/2 \rfloor$ positions of the array, while the leaf keys will occupy the last $\lceil n/2 \rceil$ positions.

b) the children of a key in the array's parental position i $(1 \le i \le \lfloor n/2 \rfloor)$ will be positions $2i$ and $2i+1$, and, correspondingly the parent of a key in position i $(2 \le i \le n)$ will be in position $\lfloor i/2 \rfloor$

ex. of Array Representation



Index	0	1	2	3	4	5	6	7	8
Values		10	5	7	4	2	1		

parents leaves

Thus we could also define a heap as an array $H[1..n]$ in which every element in position i in the first half of the array is greater than or equal to the elements in positions $2i$ and $2i+1$.

i.e

$$H[i] \geq \max[H[2i], H[2i+1]] \text{ for } i = 1 \ldots n/2$$

Construction of a Heap

- There are two principal alternatives for doing this.
  1. Bottom-up heap construction.
  2. Top-down heap construction.

Bottom-up heap construction

- It initializes the essentially complete binary tree with n nodes by placing keys in the order given and then ~~heap~~ 'heapifies' the tree as follows.
- Starting with the last parental node, the algorithm checks whether the parental dominance holds for the key at this node.
- If it does not, the algorithm exchanges the node's key K with the larger key of its children and checks whether the parental dominance holds for K in its new position.
- This process continues until the parental dominance requires for K is satisfied.
- After completing the "heapification" of the subtree rooted at the current parental node, the algorithm proceeds to do the same for the node's immediate predecessor.
- The algorithm stops after this is done for the tree's root.

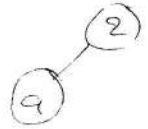
Algorithm HeapBottomUp(H[1..n])

```
// constructs a heap from the elements
// of a given array by the bottom
// algorithm.
// Input: An array H[1..n] of orderable
// Output: A heap H[1..n]

for i ← [n/2] downto 1 do
    k ← 1
    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.
```

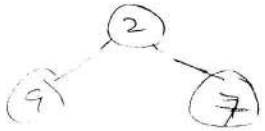
Ex construction of a heap for the list 2, 9, 7, 6, 5, 8.

insert 2

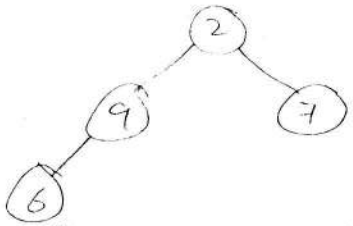
insert 9



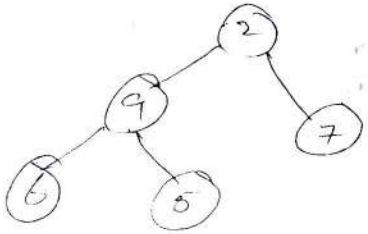
insert 7



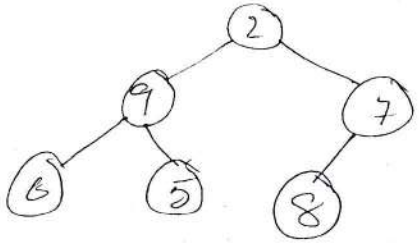
insert 6



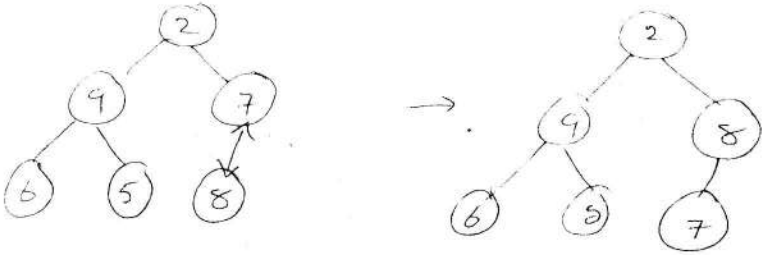
insert 5

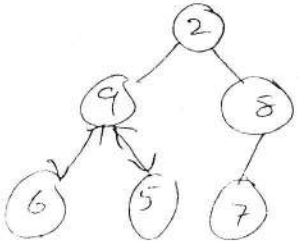


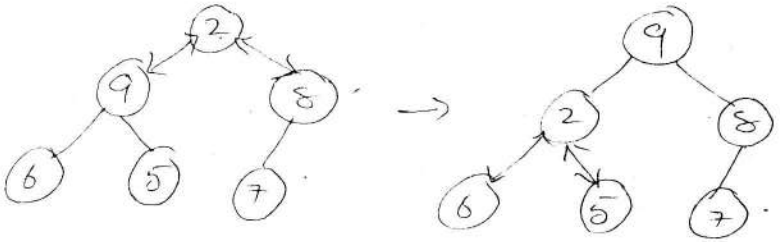
insert 8

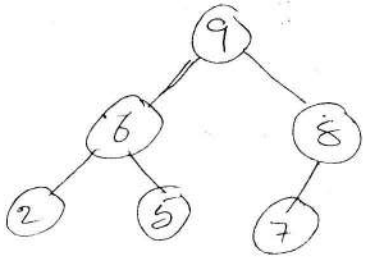


Then apply heapifies









Top-down heap construction

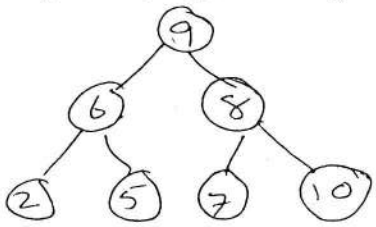
- This algorithm constructs a heap by successive insertion of a new key into a previously constructed heap.

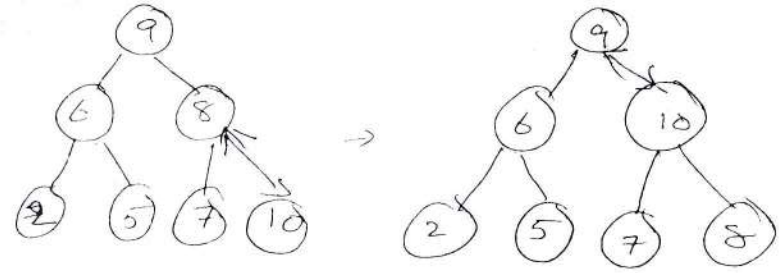
Insert a new key

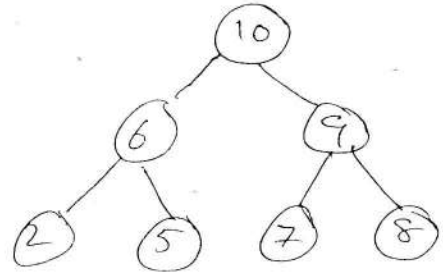
1. First, attach a new node with the key K in it after the last leaf of the existing heap.
2. Then shift K up to its appropriate place in the new heap as follows.
   - Compare K with its parent's key. if the latter is greater than or equal to K, stop
   Otherwise, swap these two keys and compare K with its new parent.

   This swapping continues until K is not greater than its last parent or it reaches the root.

ex
insert new key 10.







Delete

- we consider here only the most im case of deleting the root's key.
- Deleting the root's key from a h can be done with the following algo

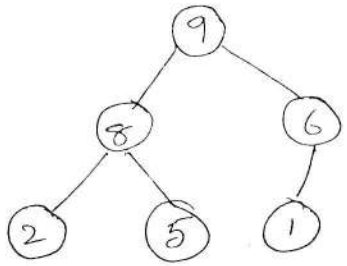
Maximum key Deletion from a heap

S1: Exchange the root's key with the least key K of the heap

S2: Decrease the heap's size by 1.

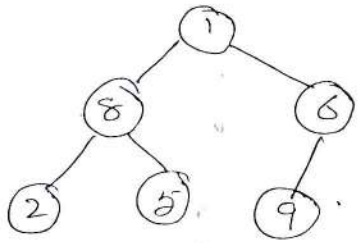
S3: "Heapify" the smaller tree by sl K down the tree exactly in the s way we did in the bottom up ap heap construction algorithm.

- That is, verify the parental dominance for k, if it holds, we are done.
If not, swap k with the larger of its children and repeat this operation until the parental dominance condition holds for k in its new position.

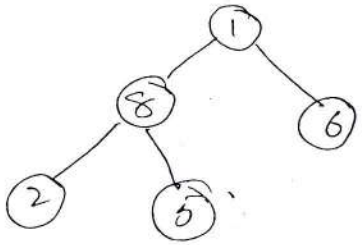


Delete the root 9.

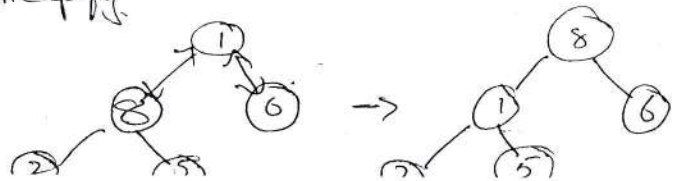
Swap 9 and 1

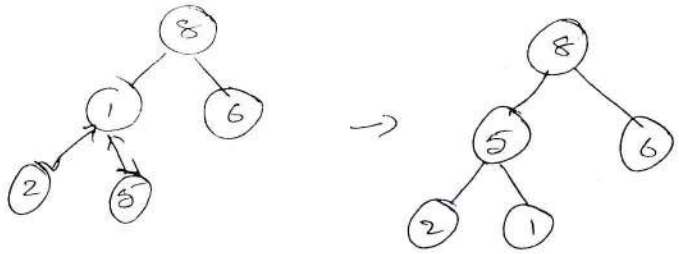


Remove 9



Apply Heapify





Heapsort:

This algorithm discovered by J.W.J. Willia

This is a two-stage algorithm.

Stage 1: Heap Construction.

Construct a heap for a given array

Stage 2: Maximum Deletions.

Apply the root-deletion operation n-1 times to the remaining heap.

- As a result, the array elements are eliminated in decreasing order
- But since under the array implementat of heaps, an element being deleted is placed last, the resulting array will be exactly the original array stored in ascending order.

Ex. Apply Heapsort to 2, 9, 7, 6, 5, 8.

Stage 1 Heap Construction.

2 9 7 6 5 8

2 9 8 6 5 7

2 9 8 6 5 7

9 2 8 6 5 7

9 6 8 2 5 7

Stage 2 Maximum deletions.

9 6 8 2 5 7

7 6 8 2 5 | 9

8 6 7 2 5

5 6 7 2 | 8

7 6 5 2

2 6 5 | 7

6 2 5

5 2 | 6

5 2

2 | 5

2

Sorted order is 2 5 6 7 8 9

Time efficiency for both the worst and average is

$\Theta(n \log n)$.

Ex. Construct a heap for the list 1, 8, 6, 5, 3, 7, 4

Ex. Sort the below given lists by heap using the array representation of heaps.

1) 3, 2, 4, 1, 6, 5

2) H, E, A, P, S, O, R, T.

M. Jyothi Priyanka

UNIT - III

Space and Time Tradeoffs.

Sorting by Counting

- This is the application of input enhancement technique.
- The idea is to count, for each element of a list to be sorted, the total number of elements smaller than this element, and record the results in a table.
- These numbers will indicate the positions of the elements in the sorted list.

ex.

If the count is 10 for some element, it should be in the 11th position (with index 10, if we start counting with 0) in the sorted array.

- Thus, we will be able to sort the list by simply copying its elements to their appropriate positions in a new, sorted list.
- This algorithm is called comparison counting sort.

Algorithm ComparisonCountingSort(A[0..n-1])

```
// Sorts an array by comparison counting
// Input: An array A[0..n-1] of orderable elements
// Output: Array S[0..n-1] of A's elements
// sorted in nondecreasing order.
for i ← 0 to n-1 do
    count[i] ← 0
for i ← 0 to n-2 do
    for j ← i+1 to n-1 do
        if A[i] < A[j]
            count[j] ← count[j] + 1
        else
            count[i] ← count[i] + 1
for i ← 0 to n-1 do
    S[count[i]] ← A[i]
return S.
```

Example

Array A[0..5]

0	1	2	3	4	5
62	31	84	96	19	47

Initially	count[]	0	0	0	0	0	0
After pass i=0	count[]	3	0	1	1	0	0
After pass i=1	count[]		1	2	2	0	1
After pass i=2	count[]			4	3	0	1
After pass i=3	count[]				5	0	1
After pass i=4	count[]					0	2
Final state	count[]	3	1	4	5	0	2

Array S[0..5]

19	31	47	62	84	96

Time efficiency is $\frac{n(n-1)}{2}$

Distribution Counting

- The counting idea does work productively in a situation in which elements to be sorted belong to a known small set of values.
- For ex, we have to sort a list whose values can be either 1 or 2.
- We can scan the list to compute the number of 1's and and the number of 2's in it and then, on the second pass, simply make the appropriate number of the first elements equal to 1 and the remaining elements equal to 2.
- If element values are in integers between some lower bound l and upper bound u, we can compute the frequency of each of those values and store them in an array $F[0..u-l]$.
- Then the first $F[0]$ positions in the sorted list must be filled with l, the next $F[1]$ positions with $l+1$, and so on.
- All this can be done, of course, only if we can overwrite the given elements.
- A more realistic situation of sorting a list of items with some other information associated with thier keys so that we can overwrite the list of elements.
- Then we can copy elements into a new array $S[0..n-1]$ to hold the sorted list as follows.
- The elements of A whose values are equal to the lowest possible value l are copied into the first elements of S, i.e., positions 0 through $F[0]-1$, the elements of value $l+1$ are copied to positions from $F[0]$ to $(F[0]+F[1])-1$, and so on.
- Such accumulated sums of frequencies are called a distribution in statistics, the method itself is known as <u>distribution counting</u>.

<u>example</u>

- Consider sorting the array

13	11	12	13	12	12

- whose values are known to come from the set $\{11, 12, 13\}$ and should not be overwritten in the process of sorting.
- The frequency and distribution arrays are

Array values	11	12	13
Frequencies	1	3	2
Distribution values	1	4	6

<u>Algorithm</u> DistributionCounting$(A[0..n-1], l, u)$

// Sorts an array of integers from a limited
// range by distribution counting
// Input: An array $A[0..n-1]$ of integers between l
// $(l \le u)$
// output: Array $S[0..n-1]$ of A's elements sorted
// in nondecreasing order.

```
for j ← 0 to n-1
  do D[j] ← 0 ~~#~~ // initialize frequencies.

for i ← 0 to n-1
   do D[A[i] - l] ← D[A[i] - l] + 1 //compute
                                     frequencies.

for j ← n-1 down
     D[j] ← D[j-1] + D[j] // recurse for distribution

for i ← n-1 down 0 do
     j ← A[i] - l.
     S[D[j]-1] ← A[i]
     A[j] ← D[j] - 1.
return S.
```

HASHING

- Hashing is based on the idea of distributing keys among a one-dimensional array, $H[0 \cdots m-1]$ called a hash table.
- The distribution is done by computing for each of the keys, the value of some predefined function h called the hash function.
- This function assigns an integer between 0 and m-1 called the hash address to a key.
- ex. If keys are nonnegative integers, a hash function can be of the form $h(k) = k \bmod m$.
- If keys are letters of some alphabet, we can first assign a letter its position in the alphabet (ord(k)) and then apply some kind of a function used for integers.
- If k is a character string $c_0 c_1 \cdots c_{k-1}$, we can use as a very unsophisticated option $\sum_{i=0}^{s-1} ord(c_i) \bmod m$.
- A better option to compute h(k) is

```
h ← 0;
for i ← 0 to s-1 do
    h ← (h * c + ord(c_i)) mod m,   c is constant
```

- A hash function satisfy two conflicting requirements.

1. A hash function needs to distribute keys among the cells of the hash table as evenly as possible.
2. A hash function has to be easy to compute.

- Obviously, if we choose a hash table's size m to be smaller than the no. of keys n, we will get collisions (i.e two or more keys being hashed into the same cell of the hash table)
- Still, every hashing scheme must have a collisi resolution mechanism.
- This mechanism is different in the two princi versions of hashing.

1. open hashing. 2. closed hashing.

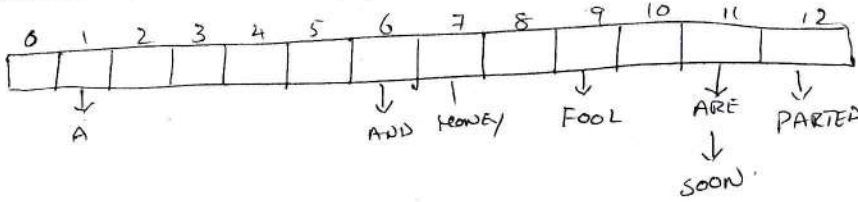
Open Hashing (Separate Chaining)

- In open hashing, keys are stored in linked lists attached to cells of a hash table.
- Each list contains all the keys hashed to its cell.

ex list of words.
A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED

- A hash function is, we will add the position a word's letters in the alphabet and compute the sum's remainder after division by 13.
- We start with empty table.
- The first key is the word A, its hash value is $h(A) = 1 \bmod 13 = 1$.
- The second key is the word FOOL (6+15+15+12) $h(FOOL) = 48 \bmod 13 = 9$. and so on.

keys	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
hash address	1	9	6	10	7	11	11	12



- If we want to search for the key KID in the hash table, we first compute the value of the same hash function for the key h(KID) = 11.
- Since the list attached to cell 11 is not empty, its linked list may contain the search key.
- The efficiency of searching depends on the length of the linked lists.
- If the hash function distributes n keys among m cells of the hash table about evently, each list will be about n/m keys long.
- The ratio $\alpha = n/m$, called the <u>load factor</u> of the hash table.
- The average number of pointers (chain links) for succenful search is $S \sim 1 + \frac{\alpha}{2}$
  for consuccenful search is $U = \alpha$
- <u>Insertions</u> are normally done at the end of the list.
- <u>Deletion</u> is performed by searching for a key to be deleted and then removing it from its list.
- The efficiency of this hashing is $\Theta(1)$

<u>Closed Hashing (Open Addressing)</u>

- In closed hashing, all keys are stored in the hash table itself without the use of linked lis
- Different strategies can be employed for collisi resolution.
- The simplest one - called <u>Linear Probing</u>.
- Linear probing checks the cell following th one whether the collision occurs.
- If that cell is empty, the new key is installe there, if the next cell is already occupied the availability of the cell's immediate successor is checked, and so on.

<u>Example</u>

Keys	A	FOOL	AND	HIS	MONEY	ARE	SOON	PART
hash address	1	9	6	10	7	11	11	12

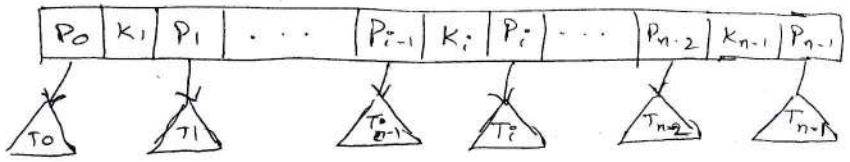
0	1	2	3	4	5	6	7	8	9	10	11	1
	A											
	A								FOOL			
	A					AND			FOOL			
	A					AND			FOOL	HIS		
	A					AND	MONEY		FOOL	HIS		
	A					AND	MONEY		FOOL	HIS	ARE	
	A					AND	MONEY		FOOL	HIS	ARE	SOO
PARTED	A					AND	MONEY		FOOL	HIS	ARE	SOO

- To search for a given key K, we start by compute h(K) where h is the hash function used in the table's construction.

- If the cell h(k) is empty, the search is unsuccessful.
- If the cell is not empty, we must compare k with the cell's occupant.
- If they are equal, we have found a matching key.
- If they are not, we compare k with a key in the next cell and continue in this manner until we encounter either a matching key.
- The search and insertion operations are straightforward for this version of hashing, deletion is not.
- e.g. if we simply delete the key ARE from the last state of the hash table
- We will be unable to find the key SOON afterward.
- After computing h(SOON) = 11, the algorithm would find this location empty, and report the unsuccessful search result.
- A simple solution is to use <u>lazy deletion</u> i.e. to mark previously occupied location by a special symbol to distinguish them from the location that have not been occupied.

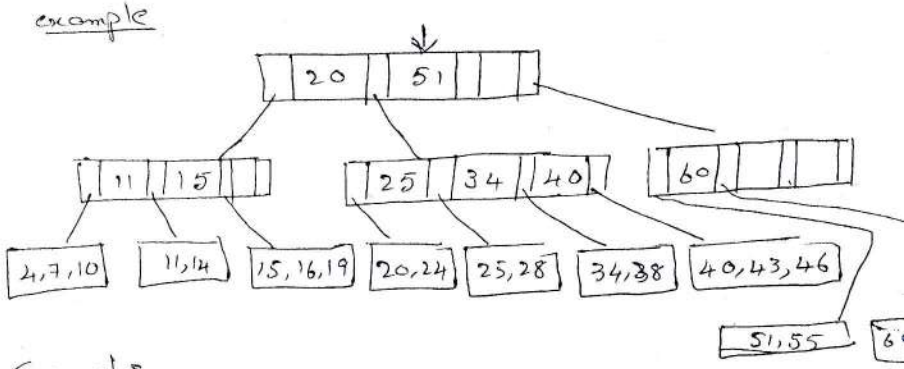
B-TREES

- For data sets of structured records, the most important index organization is the B-tree introduced by R. Bayer and E. McCreight.
- In the B-tree, all data records (or record keys) are stored at the leaves, in increasing order of the keys.
- The parental nodes are used for indexing.
- Specifically, each parental node contains $n-1$ ordered keys $K_1 < K_2 \cdots K_{n-1}$.
- The keys are interpreted with n pointers to the node's children so that all the keys in subtree T_0 are smaller than K_1, all the keys in subtree T_1 are greater than or equal to K_1 and smaller than K_2 with K_1 being equal to the smallest key in T_1 and so on through the last subtree T_{n-1} where keys are greater than or equal than K_{n-1} with K_{n-1} being equal to the smallest key in T_{n-1}



- A B-tree of order $m \geq 2$ must satisfy the following structural properties.
  1. The root is either a leaf or has between 2 and m children.
  2. Each node, except for the root and the leaves, has between $\lceil m/2 \rceil$ and m children.
  3. The tree is (perfectly) balanced, i.e. all its leaves are the same level.

example



Searching

- Searching in a B-tree is very similar to searching in the binary search tree.
- Starting at the root, we follow a chain of pointers to the leaf that may contain the search key.
- Then we search for the search key among the keys of the leaf.

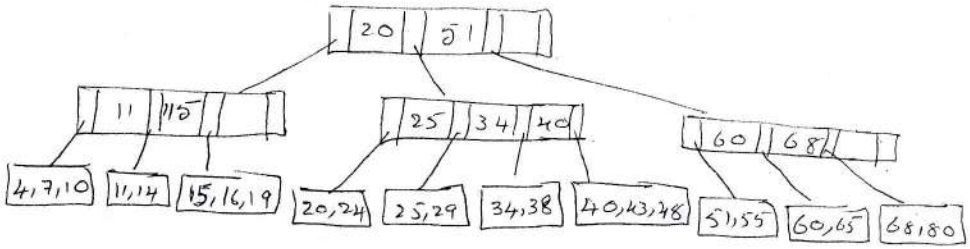
note: keys are stored in sorted order, at both parental nodes and leaves.

- How many nodes of a B-Tree do we need to access during a search for a record with a given key value?
- This number is obviously equal to the height of the tree plus one
- The operations of insertion and deletion are less straightforward than searching, but both can also be done in $O(\log n)$ time.

Inserting a node into B-Tree.

- First, we apply the search procedure to the new record's key k to find the appropriate leaf for the new record.
- If there is room for the record in that leaf, we place it there (in an appropriate position so that the keys remain sorted) and we are done.
- If there is no room for the record, the leaf is split in half by sending the second half of the records to a new node.
- After that, the smallest key k in the new node and the pointer to it have to be inserted into the old leaf's parent.
- This recursive procedure percolate (to parent) upto the tree's root.
- If the root is already full too, a new root is created with the two halves of the old root's keys split between two children of the new root.

ex. Insert 65.



DYNAMIC PROGRAMMING

- Dynamic Programming is an algorithm design technique.
- It was invented by a prominent U.S. Mathematician, Richard Bellman, in the 1950s.
- The word programming in the name of this technique stands for planning and does not refer to computer programming.
- Dynamic programming is a technique for solving problems with overlapping subproblems.
- Typically, these subproblems arise from a recurrence relating a solution to a given problem with solution to its smaller subproblems of the same type.

Warshall's and Floyd's Algorithm.

- Warshall's algorithm for computing the transitive closure of directed graph.
- Floyd's algorithm for all-pairs shortest paths problem.
- These algorithm are based on essentially the same idea.
- These are an application of the dynamic programming technique.

Warshall's Algorithm.

- Adjacency matrix $A = \{a_{ij}\}$ of a directed graph is the boolean matrix that has 1 in its ith row and j^{th} column if and only if there is a directed edge from the ith vertex to the jth vertex.

Def.

The transitive closure of a directed graph with a vertices can be defined as the n-by-n boolean matrix $T = \{T_{ij}\}$, in which the element in the ith row $(1 \le i \le n)$ and the jth column $(1 \le j \le n)$ is 1 if there exists a nontrivial-directed path. (i.e, a directed path of a positive length from the ith vertex to the jth vertex: otherwise t_{ij} is 0.

- We can generate the transitive closure of a digraph with the help of depth-first search & breadth-first search.
- Performing either traversal starting at the ith vertex gives the information about the vertices reachable from the ith vertex and hence the column that contain one in the ith row of the transitive closure.

- Thus doing such a traversal for every vertex as a starting point yields the transitive closure in its entirety.
- Warshall's algorithm constructs the transitive closure of a given digraph with n vertices through a series of n-by-n boolean matrices.

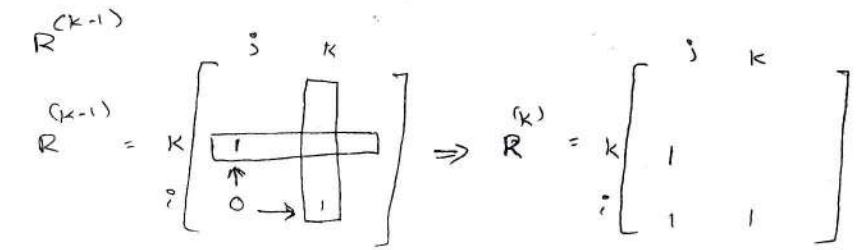
$$R^{(0)} \ldots R^{(k-1)}, R^{(k)} \ldots R^{(n)}$$

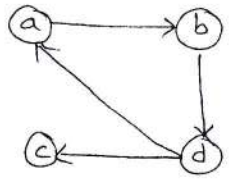
- Each of these matrices provides certain information about directed paths in the digraph.
- The element $r_{ij}^{(k)}$ in the ith row and j^{th} column of matrix $R^{(k)}$ $(k = 0, 1, \ldots n)$ is equal to 1 if and only if there exists a directed path from the ith vertex to the jth vertex with each intermediate vertex, if any, numbered not higher than k.
- $R^{(0)}$ does not allow any intermediate vertices in its paths, hence, $R^{(0)}$ is the adjacency matrix of the digraph.
- $R^{(1)}$ contains the information about paths that can use the first vertex as intermediate.
- It may contain more ones than $R^{(0)}$.
- The last matrix in this series, $R^{(n)}$, reflects paths that can use all n vertices of the digraph as intermediate and hence is nothing else but the digraph's transitive closure.

- The central point of the algorithm is that we can compute all the elements of each matrix $R^{(k)}$ from its immediate predecessor $R^{(k-1)}$.
- Let $r_{ij}^{(k)}$, the element in the ith row and j^{th} column of matrix $R^{(k)}$ be equal to 1.
- This means that there exists a path from the ith vertex v_i to the jth vertex v_j with each intermediate vertex numbered not higher than k.
- If $r_{ij}^{(k)} = 1$, then either $r_{ij}^{(k-1)} = 1$ or both $r_{ik}^{(k-1)} = 1$ and $r_{kj}^{(k-1)} = 1$.
- This formula for generating the elements of matrix $R^{(k)}$ from the elements of matrix $R^{(k-1)}$ is.

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} \text{ or } \left(r_{ik}^{(k-1)} \text{ and } r_{kj}^{(k-1)} \right)$$

- If an element r_{ij} is 1 in $R^{(k-1)}$, it remains 1 i
- If an element r_{ij} is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ if and only if the element in its row i and column k and the element in its column j and row k are both 1's in





The adjacency matrix is

$$\begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

<u>Rule</u> $R^{k}[i,j] = R^{k-1}[i,j]$ or $R^{k-1}[i,k]$ and $R^{k-1}[k,j]$

$$R^0 = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

$$R^1 = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 0 \end{array}$$

$k = 1$

$R(1,1) = 0.$

$R(1,1) \times R(1,1) = 0.$

$R(1,2) = 1$

$R(2,1) = 0$ or $R(2,1) \times R(1,2)$
$= 0 \times 1 = 0.$

$(4,1) \times (2,4)$

$R(4,1) = 0$ or $(1 \times 1) = 1$

$$R^2 = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 1 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

$k = 2$

$R(1,4) = 0$ or $(1,4)$ $(1,2) \times (2,4)$
$1 \times 1 = 1$

$R(4,4) = (4,4)$ or $(4,2) \times (2,4)$
$1 \times 1 = 1$

$$R^3 = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 1 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

$$R^4 = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 1 & 1 & 1 & 1 \\ b & 1 & 1 & 1 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

$k = 4$

$R(1,1) = R(1,1)$ or $(1,4) \times$
0 $1 \times$

$R(1,3) = (1,3)$ or $(1,4)$
0 $1 \times$

$R(2,1) = (2,1)$ or $(2,4) \times$
0 $1 \times$

$R(2,2) = (2,2)$ or $(2,4)$
$1 \times$

$R(2,3) = (2,3)$ or $(2,4)$
1

<u>Algorithm</u> <u>Warshall</u> $A[1..n, 1..n]$

// Implements Warshall's algorithm for computing
// the transitive closure.
// Input: The adjacency matrix A of a digraph
// with n vertices.
// Output: The transitive closure of the digraph.

```
R^(0) ← A.
for k ← 1 to n do
    for i ← 1 to n do.
        for j ← 1 to n do
            R^(k)[i, j] ← R^(k-1)[i, j] or
                          (R^(k-1)[i, k] and R^(k-1)[k, j])
return R^(n)
```

$l = 4$ i.e. $(0, 1, 2, 3)$

Mat$(0,1,2,3)$ → Mat$(0,(1,2,3))$

① → $132 + 2 \times 3 \times 5$

→ $132 + 30 = 162$.

Mat$(0,1,2,3)$ → Mat$((0,1,2),3)$

② → $84 + 2 \times 4 \times 5$

→ $84 + 40 = 124$

Mat$(0,1,2,3)$ → Mat$((0,1),(2,3))$

③ → $36 + 120 + 2 \times 6 \times 5$

→ $36 + 120 + 60 = 216$

from ① & ② & ③ ② is minimum.

Multiplications required = 124

$= [((0,1,2),3]$

$= [((0,1),2),3]$

$= [((A_1, A_2), A3) A4]$

Eg. ① Find the matrix chain multiplication of

$P[\ ] = \{40, 20, 30, 10, 30\}$ i.e.

A(40, 20), B(20, 30), C(30, 10), D(10, 30)

output = 26000 i.e. $[(A(BC))D]$

② Find the matrix chain multiplication of

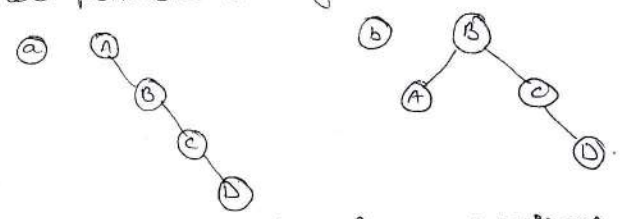
$P[\ ] = \{10, 20, 30, 40, 30\}$ i.e.

A(10, 20), B(20, 30), C(30, 40), D(40, 30)

output = 6000 i.e. $((AB)C)D$.

Optimal Binary Search Trees

- A binary search tree is one of the most important data structures in computer science.
- One of its principal applications is to implement a dictionary, a set of elements with the operations of searching, insertion and deletions.
- If probabilities of searching for elements of a set are known then an optimal binary search tree for which the average number of comparisons in a search is the smallest possible.
- Consider four keys A, B, C and D to be searched for with probabilities 0.1, 0.2, 0.4 and 0.3
- Two possible binary search trees are



No. of possible search trees are 14. $\frac{{}^{2n}C_n}{n+1}$

- The average number of comparison in a successful search (a) $0.1 \times 1 + 0.2 \times 2 + 0.4 \times 3 + 0.3 \times 4 = 0.1 + 0.4 + 1.2 + 1.2 = 2.9$.

(b) $0.1 \times 2 + 0.2 \times 1 + 0.4 \times 2 + 0.3 \times 3$

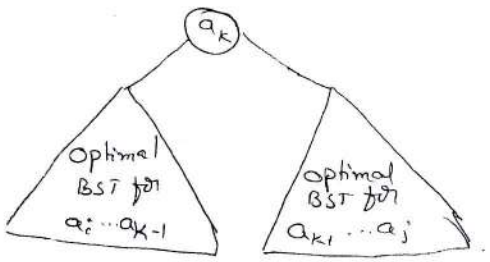
$= 0.2 + 0.2 + 0.8 + 0.9 = 2.1$

- We would find the optimal tree by generating all binary search trees with these keys.
- The total number of binary search trees with n keys is equal to the nth Caralan Number

$$C(n) = \binom{2n}{n} \frac{1}{n+1} \text{ for } n > 0$$

$$C(0) = 1.$$

- We will find the values of $c[i,j]$ for all smaller instances of the problem.
- Although we are interested just in $c[1,n]$.
- We will consider all possible ways to choose a root a_k among the keys $a_i \cdots a_j$.
- The root key a_k, the left subtree T_i^{k-1} contains keys $a_i \cdots a_{k-1}$, and the right subtree T_{k+1}^{j} contains keys $a_{k+1} \cdots a_j$



- If we count tree travels starting with 1, the following recurrence relation is obtained.

$$c[i,j] = \min_{i \le l \le j} \Big[P_k + \sum_{s=1}^{k-1} P_s - (\text{level of } a_s \text{ in } T_i^{k-1} + 1) + \sum_{s=k+1}^{j} P_s - (\text{level of } a_s \text{ in } T_{k+1}^{j} + 1) \Big]$$

$$= \min_{i \le l \le j} \Big[P_k + \sum_{s=1}^{k-1} P_s - \text{level of } a_s \text{ in } T_i^{k-1} + \sum_{s=1}^{k-1} P_s + \sum_{s=k+1}^{j} P_s - \text{level of } a_s \text{ in } T_{k+1}^{j} + \sum_{s=k+1}^{j} P_s \Big]$$

$$= \min_{i \le l \le j} \Big[\sum_{s=1}^{k-1} P_s - \text{level of } a_s \text{ in } T_i^{k-1} + \sum_{s=k+1}^{j} P_s - \text{level of } a_s \text{ in } T_{k+1}^{j} + \sum_{s=l}^{j} P_s \Big]$$

$$= \min_{i \le l \le j} \big[c[i,k-1] + c[k+1,j] \big] + \sum_{s=l}^{j} P_s$$

$$\therefore c[i,j] = \min_{i \le l \le j} \big[c[i,k-1] + c[k+1,j] \big] + \sum_{s=i}^{j} P_s \text{ for } 1 \le i \le j \le n.$$

Example

	0	1	2	3
Keys	10	12	16	21
Frequencies	4	2	6	3

	0	1	2	3
0	4	$8^{(0)}$	$20^{(2)}$	$26^{(2)}$
1		2	$10^{(2)}$	$16^{(2)}$
2			6	$12^{(2)}$
3				3

Find $c[i,j]$. $c[i,i] = P_i$.

$c[i,j] = \text{cost}[i,j] + \min\{\text{total cost of root } i, \text{ total cost of root } j\}$

$l = 1$.

$c[0,0] = 4$

$c[1,1] = 2$

$c[2,2] = 6$

$c[3,3] = 3$.

$l = 2$. $\Rightarrow \{(0,1), (1,2), (2,3)\}$.

case 1.

$$c[0,1] = 4 + 2 + \min \begin{cases} 2 & \text{if root is } 0 \\ 4 & \text{if root is } 1 \end{cases}$$

$$= 6 + 2 = 8^{(0)}$$

$$c[1,2] = 2 + 6 + \min \begin{cases} 6 & \text{if root is } 1 \\ 2 & \text{if root is } 2 \end{cases}$$

$$= 8 + 2 = 10^{(2)}$$

$$c[2,3] = 6 + 3 + \min \begin{cases} 3 & \text{if root is } 2 \\ 6 & \text{if root is } 3 \end{cases}$$

$$= 9 + 3 = 12^{(2)}$$

$l = 3 \quad : \{(0,1,2), (1,2,3)\}$

$$c[0,1,2] = 4+2+6+\min\begin{cases} 10 & \text{if root is } 0 \text{ i.e. cost of } (1,2) \\ 4+6=10 & \text{if root is } 1 \text{ i.e. cost } 0 + \text{cost } 2 \\ 8 & \text{if root in } 2 \text{ i.e. cost of } (0,1) \end{cases}$$

$$= 12+8 = 20^{(2)}$$

$$c[1,2,3] = 2+6+3+\min\begin{cases} 12 & \text{if root is } 1 \text{ i.e. cost of } (2,3) \\ 2+3 & \text{if root is } 2 \text{ i.e cost}[1] + \text{cost}(3) \\ 10 & \text{if root is } 3 \text{ i.e cost}(1,2) \end{cases}$$

$$= 11+5 = 16^{(2)}$$

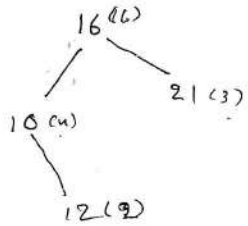
$\therefore$ Minimum $= 16^{(2)}$

$l = 4 \quad \{(0,1,2,3)\}$

$$c[0,1,2,3] = 4+2+6+3+\min\begin{cases} 16 & \text{if root is } 0 \text{ i.e. } c(1,2,3) \\ 4+12=16 & \text{if root is } 1 \text{ i.e. } c[0] + c[2,3] \\ 8+3=11 & \text{if root is } 2 \text{ i.e. } c[0,1] + c[3] \\ 20 & \text{if root is } 3 \text{ i.e } c(0,1,2) \end{cases}$$

$$= 15+11 = 26^{(2)}$$

Binary Search tree is



no of comparisons $= (6\times1)+(4\times2)+(3\times2)+2\times3$

$= 6+8+6+6$

$= \underline{26}$


```
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] = kmin.
                sum = P[i]
                for s = i+1 to j do
                    sum = sum + P[s]
                    c[i,j] = minval + sum.
return c[1,n] R.
```

0/1 Knapsack Problem & Memory Function :-

- Given n items of known weights $w_1 \cdots w_n$ and values $v_1 \cdots v_n$ and knapsack capacity w.
- Find the most valuable subset of the items that fit into the knapsack.
- All the weights and the knapsack's capacity are positive integers.
- Let us consider an instance defined by the first i items, $1 \le i \le n$ with weights w_1 w_i values $v_1 \cdots v_i$ and knapsack capacity j, $1 \le j \le w$.
- Let $v[i,j]$ be the value of an optimal solution to this instance.
  i.e the value of the most valuable subset of the first i items that fit into the knapsack of capacity j.
- We can divide all the subsets of the first i items that the knapsack of capacity j into two categories.
  1. Those that do not include the ith item.
  2. Those that do.

$$v[i,j] = \begin{cases} \max[v[i-1,j],\ v_i + v[i-1, j-w_i]] & \text{if } j-w_i \ge 0 \\ v[i-1,j] & \text{if } j-w_i < 0 \end{cases}$$

Initial conditions.

$v[0,j] = 0$ for $j \ge 0$

$v[i,0] = 0$ for $i \ge 0$.

Table for solving the knapsack problem.

	0	$j-w_i$	j	w
0	0	0	0	0
$i-1$	0	$v[i-1, j-w_i]$	$v[i-1,j]$	
i	0		$v[i,j]$	
n	0			goal

Example.

item.	1	2	3	4
weight.	2	1	3	2
value.	12	10	20	15

capacity $W = 5$.

weight	value	$v[i,j]$	0	1	2	3	4	5
		0	0	0	0	0	0	0
w_1 2	12	1	0	0	12	12	12	12
w_2 1	10	2	0	10	12	22	22	22
w_3 3	20	3	0	10	12	22	30	32
w_4 2	15	4	0	10	15	25	30	37.

Initially $v[0,j] = 0$, $v[i,0] = 0$

1. $v[1,1]$, $j-w_i = 1-2 < 0$

$v[i-1,j] = v[0,1]$

$= 0$

2. $v[1,2]$, $j-w_1 = 2-2 = 0$

$\max[v[i-1,j],\ v_i + v[i-1, j-w_i]]$

$= \max[v[0,2],\ 12 + v[0,0]]$

$= \max[0,\ 12+0=12]$

$= 12$

$v[1,3]$, $j-w_1 = 3-2 > 0$

$\max[v[i-1,j],\ v_i + v[i-1, j-w_i]]$

$= \max[v[0,3],\ 12 + v[0,1]]$

$= \max[0,\ 12+0]$

$= 12$

$v[1,4]$, $j-w_1 = 4-2 > 0$

$\max[v[i-1,j],\ v_i + v[i-1, j-w_i]]$

$= \max[v[0,4],\ 12 + v[0,2]]$

$= \max[0,\ 12+0]$

$= 12$

$v[1,5]$, $j-w_i = 5-2 > 0$

$\max[v[i-1,j],\ v_i + v[i-1, j-w_i]]$

$= \max[v[0,5],\ 12 + v[0,3]]$

$= \max[0,\ 12+0]$

$= 12$

$\underline{V[2,1]}$ $j-w_2 = 1-1 = 0$

$\max(v[i-1,j], v_2+v[i-1, j-w_i])$

$= \max[v[1,1], 10+v[1,0]]$

$= \max[0, 10+0]$

$= 10$

$\underline{V[2,3]}$ $j-w_2 = 3-1>0$

$\max[v[1,3], 10+v[1,2]]$

$= \max[12, 10+12]$ (22)

$= 22$

$\underline{V[2,5]}$, $j-w_2 = 5-1>0$

$\max[v[1,5], 10+v[1,4]]$

$= \max[12, 10+12]$ (22)

$= 22$

$\underline{V[3,1]}$, $j-w_3 = 1-3<0$

$v[i-1,j] = v[2,1]$

$= 10$

$\underline{V[3,3]}$, $j-w_3 = 3-3 = 0$

$\max[v[2,3], 20+v[2,0]]$

$= \max[22, 20+0]$

$= 22$

$\underline{V[3,5]}$, $j-w_3 = 5-3>0$

$\max[v[2,5], 20+v[2,2]]$

$= \max[22, 20+12]$ (32)

$= 32$

$\underline{V[4,3]}$ $j-w_4 = 3-2>0$

$\max[v[3,3], 15+v[3,1]]$

$= \max(22, 15+10)$

$= 25$

$\underline{V[4,4]}$ $j-w_4 = 4-2>0$

$\max[v[3,4], 15+v[3,2]]$

$\max[30, 15+12]$

$= 30$

$\underline{V[2,2]}$, $j-w_2 = 2-1>0$

$\max[v[1,2], 10+v[1,1]]$

$= \max[12, 10+0]$

$= 12$

$\underline{V[2,4]}$ $j-w_2 = 4-1>0$

$\max[v[1,4], 10+v[1,3]]$

$= \max[12, 10+12]$ (22)

$= 22$

$\underline{V[3,2]}$, $j-w_3 = 2-3<0$

$v[i-1,j] = v[2,2]$

$= 12$

$\underline{V[3,4]}$ $j-w_3 = 4-3>0$

$\max[v[2,4], 20+v[2,1]]$

$= \max[22, 20+10]$ (30)

$= \cancel{\max}\ 30$

$\underline{V[4,1]}$, $j-w_4 = 1-2<0$

$v[i-1,j] = v[3,1]$

$= 10$

$\underline{V[4,2]}$, $j-w_4 = 2-2 = 0$

$\max[v[3,2], 15+v[3,0]]$

$= \max[12, 15+0]$

$= 15$

$\underline{V[4,5]}$ $j-w_4 = 5-2>0$

$\max[v[3,5], 15+v[3,3]]$

$= \max[32, 15+22]$ (37)

$= 37$

Maximal value is $v[4,5] = 37$.

We can find the composition of an optimal subset by tracking back the computations of this entry in the table.

$v[4,5] \neq v[3,5]$.

$\therefore$ item 4 is selected for optimal subset.

$\therefore$ $5-2 = 3$ remaining weight.

Latter is represented by element $v[3,3]$

$v[3,3] = v[2,3]$

$\therefore$ 3 is not part of optimal subset.

$v[2,3] \neq v[1,3]$

$\therefore$ item 2 is selected for optimal subset.

$3-1 = 2$ remaining weight.

$v[1, 3-1]$ i.e $v[1,2] \neq v[0,2]$

$\therefore$ item 1 is the final part of the optimal subset

$2-2 = 0$.

$\therefore$ Final set [item 1, item 2, item 4].

item 1	2,	12
item 2	1,	10
item 4	2,	15
	5,	37

Example ②

item	1	2	3	4
weight	1	3	4	5
value	1	4	5	7

Total weight = 7

	weight	value
w1	1	1
w2	3	4
w3	4	5
w4	5	7

v[i,j]	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1
2	1	1	4	5	5	5	5	5
3	1	1	4	5	6	6	9	
4	1	1	4	5	7	8	9	

$v[1,1]$ $j-w_1 = 1-1$
$\max(0,1)+v_1+v[0,0])$
$0+1+0)=1$

$v[1,2] = j-w_2 = 2-3 > 0$
$\max(0,2)+v_1+v[0,1]$
$= 0+1+0 = 1$

$v[1,3] = 3-3 > 0$
$\max(0,3)+v_1+v[0,2]$
$= 0+1+0 = 1$

$v[1,4] = 1$

$v[3,1]$ $j-w_3 = 1-4 < 0$
$v[2,1] = 1$

$v[3,2] = j-w_3 = 2-4 < 0$
$v[2,2] = 1$

$v[3,3] = j-w_3 = 3-4 < 0$
$v[2,3] = 4$

$v[3,4] = j-w_3 = 4-4 = 0$
$\max(v[2,4], 5+v[2,0])$
$= (5, 5+0) = 5$

$v[3,5] = j-w_3 = 5-4 > 0$
$\max(v[2,5], 5+v[2,2])$
$= (5, 5+1) = 6$

$v[3,6] = (5, 5+1) = 6$
$v[3,7] = (5, 5+4) = 9$

$v[2,1]$ $j-w_2 = 1-3 < 0$
$v[i-1,j] = v[1,1] = 1$

$v[2,2]$ $j-w_2 = 2-3 < 0$
$v[i-1,j] = v[1,2] = 1$

$v[2,3] = j-w_2 = 3-3 = 0$
$\max(v[1,3], 4+v[1,0]) = \max(1, 4+0) = 4$

$v[2,4] = j-w_2 = 4-3 > 0$
$\max(v[1,4], 4+v[1,1]) = \max(1, 4+1) = 5$

$v[2,5] = j-w_2 = 5-3 > 0$
$\max(v[1,5], 4+v[1,2]) = \max(1, 4+1) = 5$

$v[2,7] = 7-3 \geq 0$
$\max(v[1,7], 4+v[1,4]) = \max(1, 4+1) = 5$

$v[4,1] = j-w_4 = 1-5 < 0$ i.e $v[3,1] = 1$
$v[4,2] = 2-5 < 0$ i.e $v[3,2] = 1$
$v[4,3] = 3-5 < 0$ i.e $v[3,3] = 4$
$v[4,4] = 4-5 < 0$ i.e $v[3,4] = 5$
$v[4,5] = 5-5 = 0$
$\max(v[3,5], 7+v[3,0]) = (6, 7+0) = 7$

$v[4,6] = 6-5 > 0$
$\max(v[3,6], 7+v[3,1]) = (6, 7+1) = 8$

$4[4,7] = 7-5 > 0$
$\max(v[3,7], 7+v[3,2]) = (9, 7+1) = 9$

$v[4,7] = v[3,7]$
$\therefore$ row 4 is not selected
$3[3,7] \neq v[2,7]$
$\therefore$ item 3 is selected
$7-4 = 3$.

Later to represent by element [2,6]
$v[2,6] \neq v[1,6]$
$\therefore$ item 2 is selected
$3-3 = 0$.
$\therefore$ Final set {item 2, item 3}

item 2	3, 4
item 3	4, 5
	7, 9

Algorithm

```
0/1KNAPSACK(P[ ], w[ ], n, M)
// n number of items, M. capacity
for j := 0 to M  c[0, j] := 0;
for i := 0 to n  c[i, 0] := 0
for i := 1 to n
    for j = 1 to M
        if (w[i] > j)   // can't pick item
            c[i,j] := c[i-1, j];
        else
            if (P[i] + c[i-1, j-w[i]]) > c[i-1, j])
                c[i,j] := P[i] + c[i-1, j-w[i]]
            else
                c[i,j] := c[i-1, j];
return c[n, M];
```

Complexity - $\Theta(nM)$

ALL-PAIRS SHORTEST PATHS

- Let ~~G = V(G~~ $G = (V, E)$ be a directed graph with n vertices.
- Let cost be a cost of adjacency matrix for G such that $cost(i,i) = 0,\ 1 \le i \le n$.

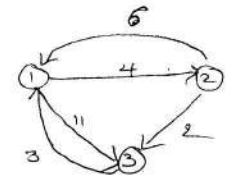
  $cost(i,j) =$ length / cost of edge (i,j) if $(i,j) \in E(G)$.

  $cost(i,j) = \alpha$ if $i \ne j$ and $(i,j) \notin E(G)$.
- The all-pairs shortest-path problem is to determine a matrix A such that $A(i,j)$ is the length of the shortest path from i to j.
- If we allow G to contain a cycle of negative length, then the shortest path between any two vertices on this cycle has length $-\alpha$.
- Let us examine a shortest i to j path in G, $i \ne j$.
- This path originates at vertex i and goes through some intermediate vertices and terminates at vertex j.
- We can assume that this path contains no cycles for if there is a cycle, then this can be deleted without increasing the path length.
- If k is an intermediate vertex on this shortest path, then the subpaths from i to k and from k to j must be shortest paths from i to k and k to j respectively.
- Otherwise, the i to j path is not of minimum length.
- So the principle of optimality holds.
- Using $A^k(i,j)$ to represent the length of a shortest path from i to j going through no vertex of index greater than k.

- Clearly $A^0(i,j) = cost(i,j),\ 1 \le i \le n,\ 1 \le j \le n$.
- We can obtain a recurrence for $A^k(i,j)$ as

$$A^k(i,j) = \min\{A^{k-1}(i,j),\ A^{k-1}(i,k) + A^{k-1}(k,j)\},\ k \ge 1.$$

Example.



Initially $A^0(i,j) = c(i,j),\ 1 \le i \le n,\ 1 \le j \le n$.

A^0	1	2	3
1	0	4	11
2	6	0	3
3	3	α	0

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

$A^1(1,1)$, $i=1, j=1, k=1$
$= \min\{A^{1-1}(1,1),\ A^{1-1}(1,1) + A^{1-1}(1,1)\}$
$= \min\{A^0(1,1),\ A^0(1,1) + A^0(1,1)\}$
$= \min\{0,\ 0+0\} = \underline{0}$

$A^1(1,2)$, $i=1, j=2, k=1$
$= \min\{A^{1-1}(1,2),\ A^{1-1}(1,1) + A^{1-1}(1,2)\}$
$= \min\{A^0(1,2),\ A^0(1,1) + A^0(1,2)\}$
$= \min\{4,\ 0+4\} = \underline{4}$

$A^1(1,3)$, $i=1, j=3, k=1$
$= \min\{A^{1-1}(1,3),\ A^{1-1}(1,1) + A^{1-1}(1,3)\}$
$= \min\{A^0(1,3),\ A^0(1,1) + A^0(1,3)\}$
$= \min\{11,\ 0+11\} = \underline{11}$

$A^1(2,1)$, $i=2, j=1, k=1$
$= \min\{A^{1-1}(2,1),\ A^{1-1}(2,1) + A^{1-1}(1,1)\}$
$= \min\{A^0(2,1),\ A^0(2,1) + A^0(1,1)\}$
$= \min\{6,\ 6+0\} = \underline{6}$

$A^1(2,2)$, $i=2, j=2, k=1$
$= \min\{A^{1-1}(2,2),\ A^{1-1}(2,1) + A^{1-1}(1,2)\}$
$= \min\{A^0(2,2),\ A^0(2,1) + A^0(1,2)\}$
$= \min\{0,\ 6+4\} = \underline{0}$

$A^1(2,3) = \min\{A^{1-1}(2,3),\ A^{1-1}(1,2)+A^{1-1}(1,3)\}$
$i=2, j=3, k=1$
$= \min\{A^0(2,3),\ A^0(1,2)+A^0(1,3)\}$
$= \min\{2,\ 6+11\} = \underline{2}$

$A^1(3,1) = \min\{A^{1-1}(3,1),\ A^{1-1}(3,1)+A^{1-1}(1,1)\}$
$i=3, j=1, k=1$
$= \min\{A^0(3,1),\ A^0(3,1)+A^0(1,1)\}$
$= \min\{3,\ 3+0\} = \underline{3}$

$A^1(3,2) = \min\{A^{1-1}(3,2),\ A^{1-1}(3,1)+A^{1-1}(1,2)\}$
$i=3, j=2, k=1$
$= \min\{A^0(3,2),\ A^0(3,1)+A^0(1,2)\}$
$= \min\{\infty,\ 3+4\} = \underline{7}.$

$A^1(3,3) = 0.$

A^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

calculations for A^2 matrix.

$A^2(1,1) = 0.$

$A^2(1,2) = \min\{A^{2-1}(1,2),\ A^{2-1}(1,2)+A^{2-1}(2,2)\}$
$i=1, j=2, k=2$ $= \min\{A^1(1,2),\ A^1(1,2)+A^1(2,2)\}$
$= \min\{4,\ 4+0\} = \underline{4}$

$A^2(1,3) = \min\{A^{2-1}(1,3),\ A^{2-1}(1,2)+A^{2-1}(2,3)\}$
$i=1, j=3, k=2$ $= \min\{A^1(1,3),\ A^1(1,2)+A^1(2,3)\}$
$= \min\{11,\ 4+2\} = \underline{6}$

$A^2(2,1) = \min\{A^{2-1}(2,1),\ A^{2-1}(2,2)+A^{2-1}(2,1)\}$
$i=2, j=1, k=2$ $= \min\{A^1(2,1),\ A^1(2,2)+A^1(2,1)\}$
$= \min\{6,\ 0+6\} = \underline{6}$

$A^2(2,2) = 0.$

$A^2(2,3) = \min\{A^{2-1}(2,3),\ A^{2-1}(2,2)+A^{2-1}(2,3)\}$
$i=2, j=3, k=2$ $= \min\{A^1(2,3),\ A^1(2,2)+A^1(2,3)\}$
$= \min\{2,\ 0+2\} = \underline{2}$

$A^2(3,1) = \min\{A^{2-1}(3,1),\ A^{2-1}(3,2)+A^{2-1}(2,1)\}$
$i=3, j=1, k=2$ $= \min\{A^1(3,1),\ A^1(3,2)+A^1(2,1)\}$
$= \min\{3,\ 7+6\} = \underline{3}$

$A^2(3,2) = \min\{A^{2-1}(3,2),\ A^{2-1}(3,2)+A^{2-1}(2,2)\}$
$i=3, j=2, k=2$ $= \min\{A^1(3,2),\ A^1(3,2)+A^1(2,2)\}$
$= \min\{7,\ 7+0\} = \underline{7}$

$A^2(3,3) = 0.$

A^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

calculations for A^3 matrix.

$A^3(1,1) = 0$

$A^3(1,2) = \min\{A^{3-1}(1,2),\ A^{3-1}(1,3)+A^{3-1}(3,2)\}$
$i=1, j=2, k=3$ $= \min\{A^2(1,2),\ A^2(1,3)+A^2(3,2)\}$
$= \min\{4,\ 6+7\} = \underline{4}$

$A^3(1,3) = \min\{A^{3-1}(1,3),\ A^{3-1}(1,3)+A^{3-1}(3,3)\}$
$i=1, j=3, k=3.$ $= \min\{A^2(1,3),\ A^2(1,3)+A^2(3,3)\}$
$= \min\{6,\ 6+0\} = \underline{6}$

$A^3(2,1) = \min\{A^{3-1}(2,1),\ A^{3-1}(2,3)+A^{3-1}(3,1)\}$
$i=2, j=1, k=3$ $= \min\{A^2(2,1),\ A^2(2,3)+A^2(3,1)\}$
$= \min\{6,\ 2+3\} = \underline{5}$

$A^3(2,2) = 0$

$A^3(2,3) = \min\{A^{3-1}(2,3),\ A^{3-1}(2,3)+A^{3-1}(3,3)\}$
$i=2, j=3, k=3$ $= \min\{A^2(2,3),\ A^2(2,3)+A^2(3,3)\}$
$= \min\{2,\ 2+0\} = \underline{2}$

$A^3(3,1) = \min\{A^{3-1}(3,1),\ A^{3-1}(3,3)+A^{3-1}(3,1)\}$
$i=3, j=1, k=3$ $= \min\{A^2(3,1),\ A^2(3,3)+A^2(3,1)\}$
$= \min\{3,\ 0+3\} = \underline{3}$

$A^3(3,2) = \min\{A^{3-1}(3,2),\ A^{3-1}(3,3) + A^{3-1}(3,2)\}$ 4-3

$i=3, j=2, k=3$ $= \min\{A^2(3,2),\ A^2(3,3) + A^2(3,2)\}$

$= \min\{7,\ 0+7\} = \underline{\underline{7}}$

$A^3(3,3) = 0$

A^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

Finally, cost matrix A^3 is the shortest path of all the pairs.

<u>Algorithm.</u>

```
Algorithm AllPaths(cost, A, n)
// cost[1:n, 1:n] is the cost adjacency matrix.
// A[i, j] is the cost of a shortest path from i to j
// cost[i, i] = 0.0 for 1 ≤ i ≤ n.
{ for i := 1 to n do
    for j := 1 to n do
      A[i, j] := cost[i, j].
  for k := 1 to n do
  for i := 1 to n do
   for j := 1 to n do
     A[i, j] = min(A[i, j], A[i, k] + A[k, j]);
}.
```

The time complexity is $\Theta(n^3)$

THE GREEDY METHOD

1. THE GENERAL METHOD.

- The greedy method is the most straight forward design technique.
- The problem have n inputs and require us to obtain a subset that satisfies some constraints.
- Any subset that satisfies these constraints is called a feasible solution.
- We need to find a feasible solution that ~~does this is called~~ either maximizes or minimizes a given objective function.
- A feasible solution that does this is called optimal solution.
- The greedy method suggests that one can devise an algorithm that works in stage, considering one input at a time.
- At each stage, a decision is made regarding whether a particular input is an optimal solution.
- This is done by considering the inputs in an order determined by some selection procedure.
- If the inclusion of the next input into the partially constructed optimal solution will result in an infeasible solution, then this input is not added to the partial solution.
- Otherwise, it is added.
- The selection procedure itself is based on some optimization measure.
- This measure may be the objective function.
- This version of the greedy technique is called the subset paradigm.

Algorithm:

Greedy method control abstraction for the subset paradigm.

```
Algorithm Greedy(a, n)
// a[1:n] contains the n inputs.
{
   solution := 0; // Initialize the solution.
   for i := 1 to n do
   {
      x := Select(a);
      if Feasible(solution, x) then
         solution := Union(solution, x);
   }
   return solution;
}.
```

- The function Select() selects an input from a[] and removes it.
- The selected input's value is assigned to x.
- Feasible is a Boolean-valued function that determines whether x can be included into the solution vector.
- The function Union() combines x with the solution and updates the objective function.
- The function Greedy describes the essential way that a greedy algorithm will look, once a particular problem is chosen and the function Select, Feasible and Union are properly implemented.

Example 1. Change Making.

- A child buys candy valued at less than \$1 and gives a \$1 bill to the cashier.
- Assume that an unlimited supply of quarters, dimes, nickels, and pennies is available.
- The cashier constructs the change in stages.
- In each stage a coin is added to the change.
- This coin is selected using the greedy criteria.
- At each stage increase the total amount of change constructed by as much as possible.
- Suppose that 67 cents in change is due to child.

→ The first 2 coins selected are quarters.

→ A quarter can't be selected for third coin because change exceeds 67 cents. (infeasible)

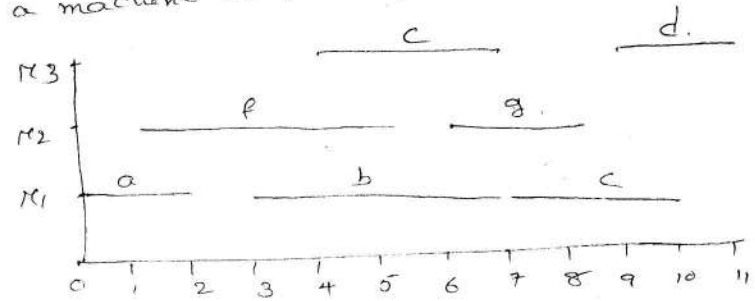
→ The third coin selected is a dime, then a nickel is selected, finally two pennies are added to the change.

Example 2. Machine Scheduling.

- You are given n tasks and an infinite supply of machines on which these tasks can be performed.
- Each task has a start time s_i and a finish time f_i, $s_i < f_i$, $[s_i, f_i]$ is the processing interval for task i.
- Two tasks i & j overlapp iff their processing intervals overlap at a point other than the interval start or end.
- A feasible task-to-machine assignment is an assignment in which no machine is assigned two overlapping tasks.
- Therefore, in a feasible assignment each machine works on at most one task at any time.
- An optimal assignment is a feasible assignment that utilizes the fewest number of machines.
- Suppose we have $n = 7$ tasks labled 'a' through 'g' and their start and finish times are given.
- Assign task a to machine M1, task b to machine M2 ..., task g to machine M7.

task	a	b	c	d	e	f	g
start	0	3	4	9	7	1	6
finish	2	7	7	11	10	5	8

- The greedy algorithm assigns tasks to machines in this order.
- The algorithm has $n = 7$ stages, and in each stage one task is assigned to a machine.
- To assign the tasks in stages, one task per stage and in nondecreasing order of task start time.
- Call a machine old if at least one task has been assigned to it.
- If a machine is not old, it is new.

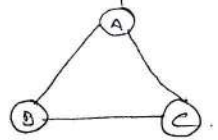


Spanning Tree.

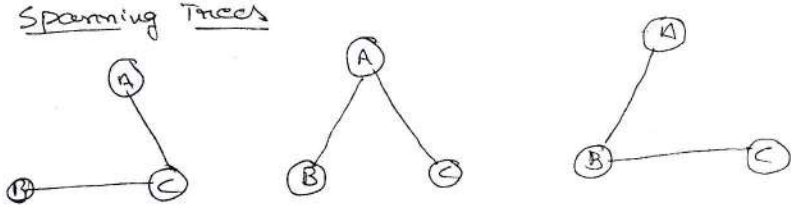
A Spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges.

- Hence a Spanning tree does not have cycles and it cannot be disconnected.
- Every connected and undirected Graph G has at least one spanning tree.

graph.



Spanning Trees



Minimum Spanning Tree. (MST)

A MST or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted (un) directed graph that connects all the vertices together without any cycles and with the minimum possible total edge weight.

We have two algorithms for Minimum Spanning Tree.

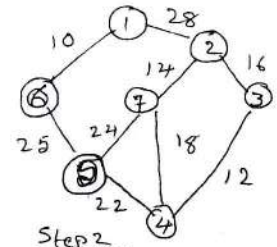
1. Prim's Algorithm
2. Kruskal's Algorithm

Prim's Algorithm.

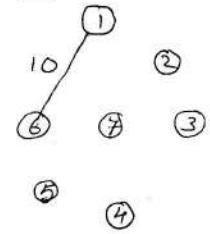
- A greedy method to obtain a minimum-cost spanning tree builds this tree edge by edge.
- The next edge to include is chosen according to some optimization criterion.
- The simplest such criterion is to choose an edge that results in a minimum increase in the sum of the costs of the edges so far included.
- There are two possible ways to interpret this criterion.

1. The set of edges so far selected forms a tree. Thus, if A is the set of edges selected so far, then A forms a tree.
2. The edge (u, v) to be included in A is a minimum cost edge not in A with the property that $A \cup \{(u,v)\}$ is also a tree.

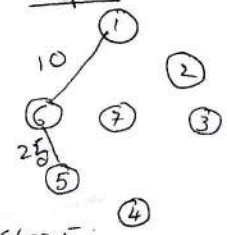
Example



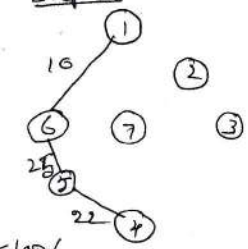
Step 1



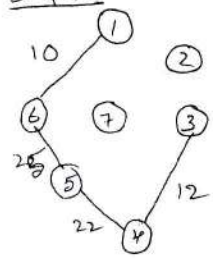
Step 2



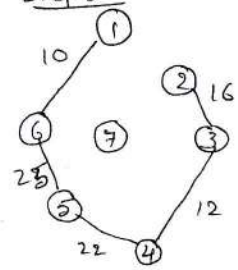
Step 3



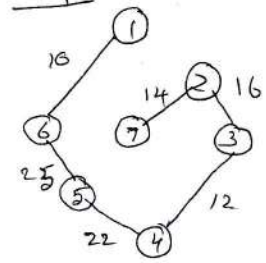
Step 4



Step 5



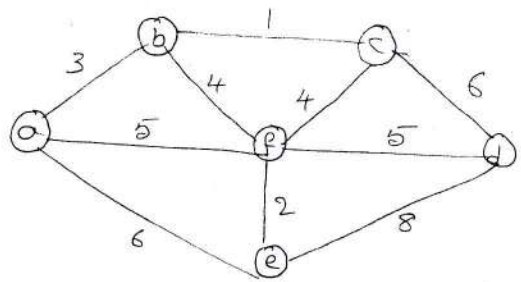
Step 6

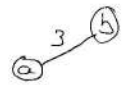
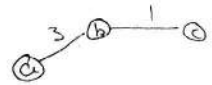
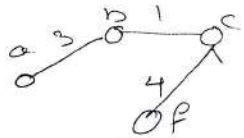
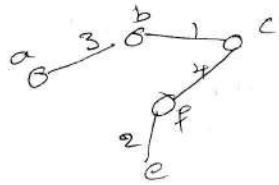
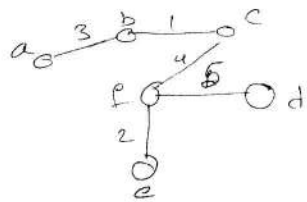


The total minimum cost = 10 + 25 + 22 + 12 + 16 + 14
= 99

- The algorithm will start with a tree that includes only a minimum-cost edge of G.
- Then the edges are added to this tree one by one.
- The next edge(i, j) to be added is such that is a vertex already included in the tree, j is a vertex not yet included, and the cost[i, j] is minimum among all edges (k, l) such that vertex k is in the tree and vertex l is not in the tree.
- To determine this edge(i, j) efficiently, we associate with each vertex j not yet included in the tree a value near[j].
- The value near[j] is a vertex in the tree such that cost[j, near[j]] is minimum among all choices for near[j].
- We define near[j] = 0 for all vertices j that are already in the tree.
- The next edge to include is defined by the vertex j such that near[j] ≠ 0 (j not already in the tree) and cost[j, near[j]] is minimum.

```
Algorithm Prim(E, cost, n, t)
// E is the set of edges in G. cost[1:n, 1:n] is the
// cost adjacency matrix of an n vertex graph such
// that cost[i, j] is either a positive real number
// or ∞ if no edge(i, j) exists. A minimum spanning
// tree is computed and stored as a set of edges
// in the array t[1:n-1, 1:2]. (t[i, 1], t[i, 2]) is an
// edge in the minimum-cost spanning tree.
// The final cost is returned.
{ Let (k, l) be an edge of minimum cost in E;
  mincost := cost[k, l];
  t[1, 1] := k;
  t[1, 2] := l;
  for i := 1 to n do // Initialize near
     if (cost[i, l] < cost[i, k]) then
        near[i] := l;
     else
        near[i] := k;
  near[k] := near[l] := 0;
  for i := 2 to n-1 do
  { // Find n-2 additional edges for t.
    Let j be an index such that near[j] ≠ 0
    and cost[j, near[j]] is minimum.
    t[i, 1] := j;
    t[i, 2] := near[j];
    mincost := mincost + cost[j, near[j]];
    near[j] := 0;
    for k := 1 to n do // update near[]
      if ((near[k] ≠ 0) and (cost[k, near[k]] >
                                cost[k, j]))
      then near[k] := j;
  }
  return mincost;
}
```



Tree Vertices	Remaining Vertices	Illustration
a(- -)	b(a, 3), c(-∞) d(-∞) e(a, 6) f(a, 5)	
b(a, 3)	c(b, 1), d(-∞) e(a, 6), f(b, 4)	
c(b, 1)	d(c, 6), e(a, 6) f(b, 4)	
f(b, 4)	d(f, 5), e(f, 2)	
e(f, 2)	d(f, 5)	
d(f, 5)		

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
// minimum spanning tree of G.

$V_T \leftarrow \{v_0\}$ // The set of tree vertices can be
// initialized with any vertex.

$E_T \leftarrow \emptyset$

for $i \leftarrow 1$ to $|V| - 1$ do

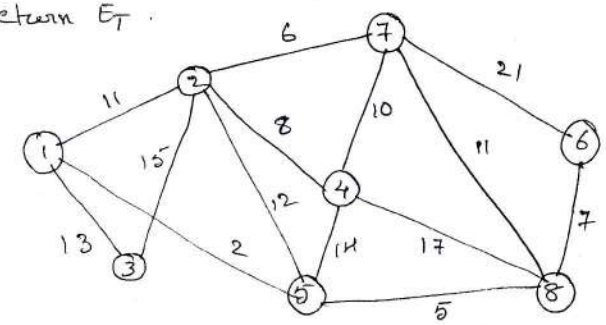
find a minimum weight edge $e^* = (v^*, u^*)$
among all the edges (v, u) such that
v is in V_T and u is in $V - V_T$

$V_T \leftarrow V_T \cup \{u^*\}$

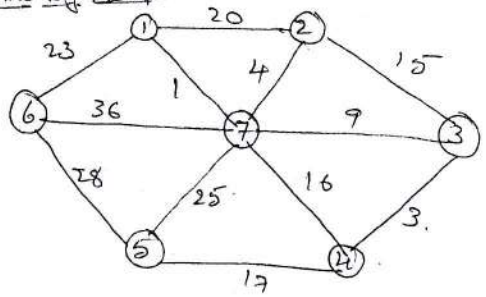
$E_T \leftarrow E_T \cup \{e^*\}$

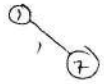
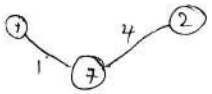
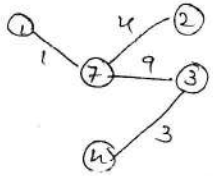
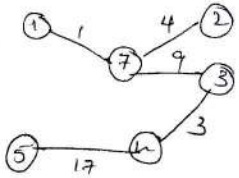
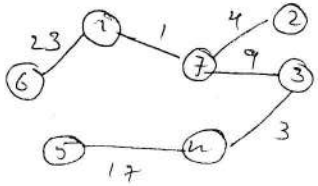
return E_T.

examples



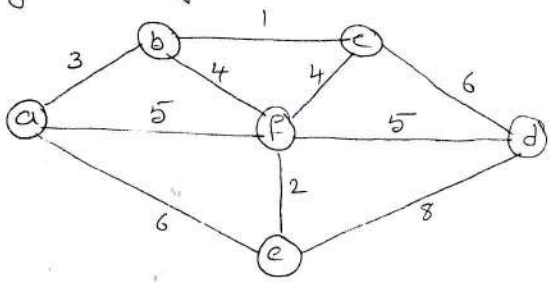
Prims Alg Example

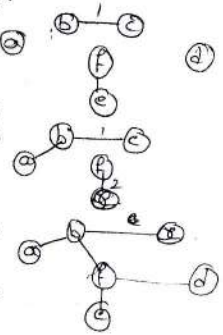


Tree Vertices	Remaing Vertices	Illustration
1(- -)	2(1, 20), 3(-, ∞) 4(-, ∞), 5(-, ∞), 6(1, 23), 7(1, 1)	
7(1, 1)	2(7, 4), 3(7, 9), 4(7, 16) 5(7, 25), 6(1, 23)	
2(7, 4)	3(7, 9) 4(7, 16), 5(7, 25) 6(1, 23)	
3(7, 9)	4(3, 3) 5(7, 25), 6(1, 23)	
4(3, 3)	5(4, 17), 6(1, 23)	
5(4, 17)	6(1, 23)	
6(1, 23).		

Kruskal's Algorithm

- Kruskal's algorithm was developed by Joseph Kruskal.
- A minimum spanning tree for a weighted connected graph $G = (V, E)$ as an acyclic subgraph with $|V|-1$ edges for which sum of the edge weights is the smallest.
- The algorithm begins by sorting the graph's edges in nondecreasing order of their weights.
- Then, starting with the empty subgraph, it scans this 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.



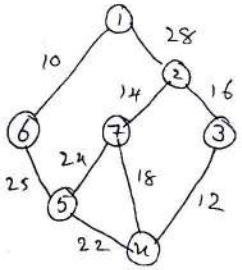
Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bc	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ef	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ab	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bf	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
df		

Algorithm Kruskal(G)

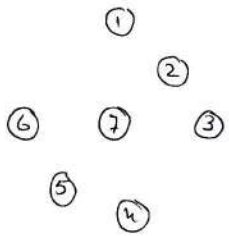
```
// Kruskal's algorithm for constructing a minimum
// spanning tree.
// Input: A weighted connected graph G = (V, E)
// Output: E_T, the set of edges composing a minimum
// spanning tree of G.
Sort E in nondecreasing order of the edge weights
  w(e_1) ≤ ... ≤ w(e_|E|)
E_T ← 0      // initialize the set of tree edges and its size.
ecounter ← 0
k ← 0        // initialize the number of processed edges.
while ecounter < |V| - 1 do
    k ← k + 1
    if E_T ∪ {e_k} is acyclic
        E_T ← E_T ∪ {e_ik};
    ecounter ← ecounter + 1
return E_T
```

1. The edges of the graph are considered in nondecreasing order of cost.
2. The set t of edges so far selected for the spanning tree be such that it is possible to complete t into a tree.
3. Thus t may not be a tree at all stages in the algorithm.
4. The set of edges t can be completed into a tree iff there are no cycles in t.

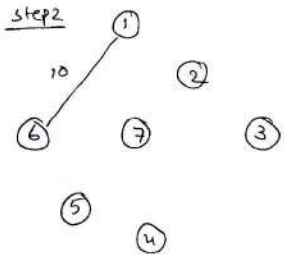
Example Kruskal.



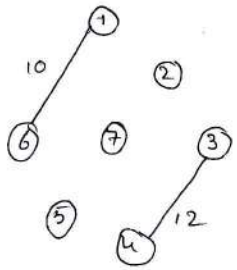
step1 Graph with no edges selected.



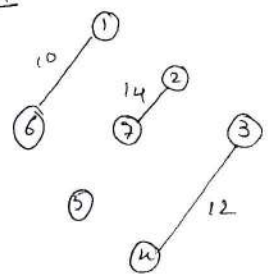
step2



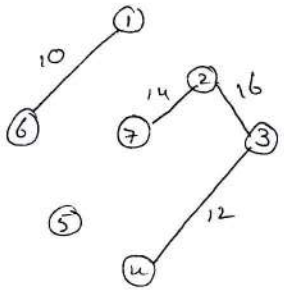
step3



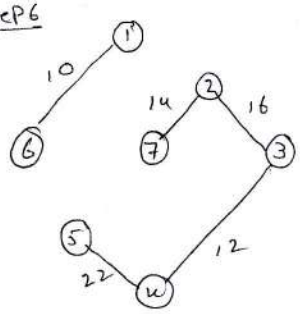
step4



step5

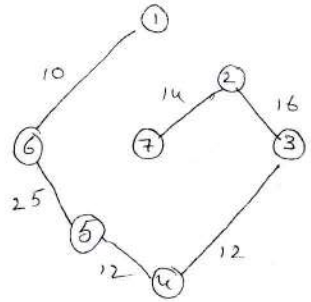


step6



The edge (7,4) is not yet considered because its inclusion in the tree results in a cycle. so this edge is discarded.

step7



Final spanning tree of cost 99

Algorithm

```
1. t := 0;
2. while((t has less than n-1 edges) and (E ≠ 0)) do
3. {
4.     choose an edge (v, w) from E of lowest cost;
5.     Delete (v, w) from E;
6.     if (v, w) does not create a cycle in t
       then add (v, w) to t;
7.     else discard (v, w);
8. }.
```

Explanation

. Initially E is the set of all edges in G.

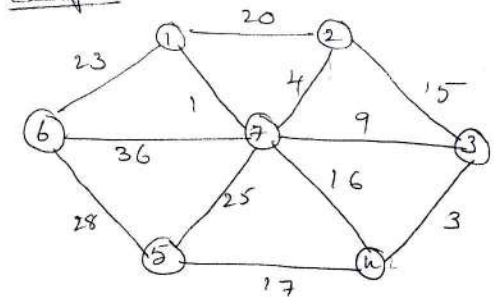
. The functions perform on this set are

1. Determine an edge with minimum cost (line 4)
2. delete this edge (line 5).

Both these functions can be performed efficiently if the edges in E are maintained as a sorted sequential list.

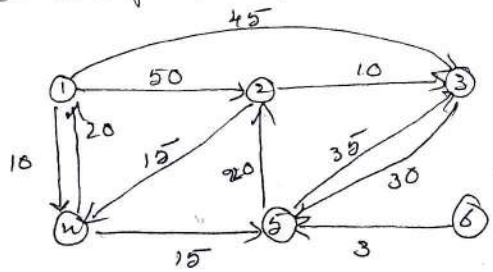
```
Algorithm. Kruskal(E, cost, n, t)
// E is the set of edges in G. G has n vertices.
// cost[u,v] is the cost of edge (u,v). t is the
// set of edges in the minimum-cost spanning
// tree. The final cost is returned.

{ construct a heap out of the edge costs using
  Heapify;
  for i := 1 to n do parent[i] := -1;
  // Each vertex is in a different set.
  t := 0;
  mincost := 0.0;
  while((i < n-1) and (heap not empty)) do
  { Delete a minimum cost edge(u,v) from
    the heap and reheapify using Adjust;
    j := Find(u);
    k := Find(v);
    if (j ≠ k) then
    {
      i := i+1;
      t[i,1] := u;
      t[i,2] = v;
      mincost := mincost + cost[u,v];
      Union(j,k);
    }
  }
  if (i ≠ n-1) then write("no spanning tree");
  else
  return mincost;
}.
```

Example.



SINGLE-SOURCE SHORTEST PATHS (or) DIJKSTRA'S ALGORITHM.

- Given a directed graph $G = (V, E)$, a weighting function cost for the edges of G.
- The starting vertex of the path is referred to as the source and the last vertex the destination.
- The graphs are digraphs.
- The problem is to determine the shortest paths from V_0 to all the remaining vertices of G.
- It is assumed that all the weights are positive.
- The shortest path between V_0 and some other node v is an ordering among a subset of the edges.
- Hence this problem fits the ordering paradigm.



Source = Node 1.

Length-adjacency Matrix

	1	2	3	4	5	6
1	0	50	45	10	-	-
2	-	0	10	15	-	-
3	-	-	0	-	30	-
4	20	-	-	0	15	-
5	-	20	35	-	0	-
6	-	-	-	-	3	0

Illustration	S	Vertex selected	1	2	3	4	5	6
Initial	—	—	0	∞	∞	∞	∞	∞
	{1}	1	0	50	45	10	∞	∞
	{1,4}	4	0	50	45	10	25	∞
	{1,4,5}	5	0	45	45	10	25	∞
	{1,4,5,2}	2	0	45	45	10	25	∞
	{1,4,5,2,3}	3	0	45	45	10	25	∞
	{1,4,5,2,3,6}	6	0	45	45	10	25	∞

If node 1 is source vertex then shortest path from 1 to 2 is {1, 4, 5, 2}.

The length of this path is $10 + 15 + 20 = 45$.

- First, a shortest path to the nearest vertex is generated.
- Then a shortest path to the second nearest vert is generated on so on.
- In order to generate the shortest paths in th order, we need to able to determine.
  1) the next vertex to which a shortest path must be generated. and
  2) A shortest path to this vertex.

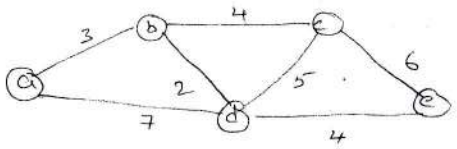
Let S denote the set of vertices (including V_0) a to which shortest paths ~~to~~ ~~the~~ vertex have already been genera

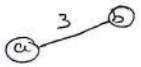
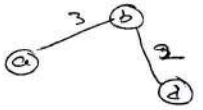
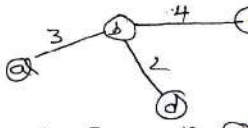
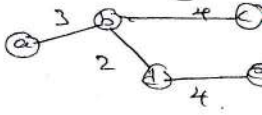
- For w not in S, let dist[w] be the length of the shortest path starting from V_0, going through only those vertices that are in S and ending at w.

```
Algorithm ShortestPaths(v, cost, dist, n)
// dist[j], 1 ≤ j ≤ n, is set to the length of the
// shortest path from vertex v to vertex j in a digraph
// G with n vertices. dist[v] is set to zero. G is
represented by its cost adjacency matrix cost[1:n, 1:n].
{
  for i := 1 to n do
   { // Initialize S.
     S[i] := false;
     dist[i] := cost[v, i];
   }
   S[v] := true;
   dis[v] := 0; // Put v in S
   for num := 2 to n do
    {
      // Determine n-1 paths from v.
      choose u from among those vertices
      not in S such that dist[u] is minimum;
      S[u] := true // Put u in S.
      for(each w adjacent to u with
         S[w] = false) do
      // update distances.
      if dist[w] > dist[u] + cost([u,w]) then
         dist[w] := dist[u] + cost[u,w];
    }
}.
```

Dijkstra's Algorithm

- We consider the Single-source shortest-path problem.
- For a given vertex called the source in a weighted graph, find shortest paths to all its other vertices.
- The single-source-shortest problem asks for a family of paths, each leading from the source to a different vertex in the graph, though some paths may, of course, have edges in common.
- There are several well-known algorithms for solving it, including Floyd's algorithm.
- Dijkstra's algorithm is applicable to graphs with nonnegative weights only.
- This algorithm finds shortest paths to a graph's vertices in order of their distance from a given source.
- First, it finds the shortest path from the source to a vertex nearest to it, then to a second nearest, and so on.
- The set of vertices adjacent to the vertices in T_i can be referred to as fringe vertices, they are the candidates from which Dijkstra's algorithm selects the next vertex nearest to the source.
- To identify the ith nearest vertex, the algorithm computes, for every fringe vertex u, the sum of the distance to the nearest tree vertex v and the length d, of the shortest path from the source to v and then selects the vertex with the smallest such sum.

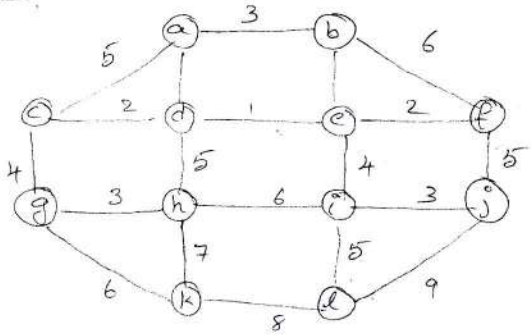


Tree vertices.	Remaining vertices.	Illustration.
a(-, 0).	b(a, 3), c(-, ∞), d(a, 7) e(-, ∞)	
b(a, 3)	c(b, 3+4), d(b, 3+2), e(-, ∞)	
d(b, 5)	c(b, 7), e(d, 5+4)	
c(b, 7)	e(d, 9)	
e(d, 9)		

The shortest paths and their lengths are

from a to b : a - b of length 3.
from a to d : a - b - d of length 5
from a to c : a - b - c of length 7.
from a to e : a - b - d - e of length 9.

Example:



```
Algorithm Dijkastra(G, s)
// Dijkastra's algorithm for Single-Source shortest paths
// Input: A weighted connected graph G = <V, E>
// with nonnegative weights and its vertex v.
// Output: The length d, of a shortest path from
// vertex v in V Initialize(Q)
 for every vertex v in V do
      d_v ← ∞.
      P_v ← null
      Insert(Q, v, d_v)
      d_s ← 0
      Decrease(Q, s, d)
      V_T ← ∅
      for i ← 0 to |V| - 1 do.
        u* ← DeleteMin(Q)
        V_T ← V_T ∪ {u*}
        for every vertex u in V - V_T that is
           adjacent to u* do
             if d_u* + w(u*, u) < d_u.
              d_u ← d_u* + w(u*, u)
              P_u ← u*
              Decrease(Q, u, d_u)
```

HUFFMAN TREES.

Codeword

- Suppose we have to encode a text that comprises characters from some n-character alphabet by assigning to each of the text's characters some sequence of bits called the codeword.

  Two types of encoding.

  1. Fixed-length encoding 2. Variable-length encoding.

Fixed-length encoding.

- This encoding assigns to each character a bit string of the same length m. ($m > \log_2 n$)
- This is exactly what the standard ASCII code does.
- One way of getting a coding scheme that yields a shorter bit string on the average is based on the old idea of assigning shorter codewords to more frequent characters and longer codewords to less frequent characters.

Variable-length encoding.

- This encoding assigns codewords of different lengths to different characters.
- Variable-length encoding introduces a problem that fixed-length encoding does not have.
- To avoid this complication, we can limit ourselves to prefix-tree (or simply prefix) code.
- In prefix code, no codeword is a prefix of a codeword of another character.
- Hence such an encoding, we can simply scan a b string until we get the first group of bits th is a codeword for some character, replace these bits by this character, and repeat th operation until the bit string's end is reache
- We want to create a binary prefix code for some alphabet, it is natural to associate the alphabet's characters with leaves of a bina tree in which all the left edges are labele by 0 and all the right edges are labeled b (or vice versa).
- The codeword of a character can then be obtained by recording the labels on the simp path from the root to the character's leaf

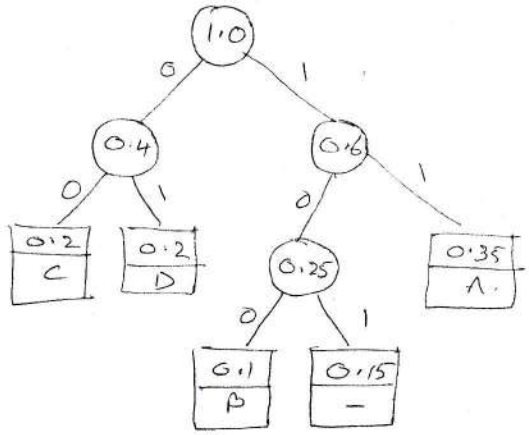
Huffman's Algorithm.

Step1:

- Initialize n one-node trees and label them with the characters of the alphabet.
- Record the frequency of each character in its tree's root to indicate the tree's weight. (weight of a tree will be qual to the sum of t frequencies in the tree's leaves)

Step2:

- 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 and record the sum of their weights in the root of the new tree as its weight.



The resulting codewords are.

Character	A	B	C	D	–
Probability codeword.	11	100	00	01	101

Hence DAD is encoded as 011101

CDAD is encoded as 00011101

BAD-AD is 1001101101101.

The expected number of bits per character in this code is

$2 \times 0.35 + 3 \times 0.1 + 2 \times 0.2 + 2 \times 0.2 + 3 \times 0.15$

$= 2.25$

- In fixed-length encoding for the same alphabet, we would have to use at least 3 bits per each character. Huffman's code achieve the compression ratio.
  i.e. $(3 - 2.25)/3 \times 100\% = 25\%$
- we should expect that Huffman's encoding of a text will use 25% less memory than its fixed length encoding.
- Huffman's encoding is one of the most important file compression methods.

A tree constructed by the above algorithm is called a <u>Huffman tree</u>.

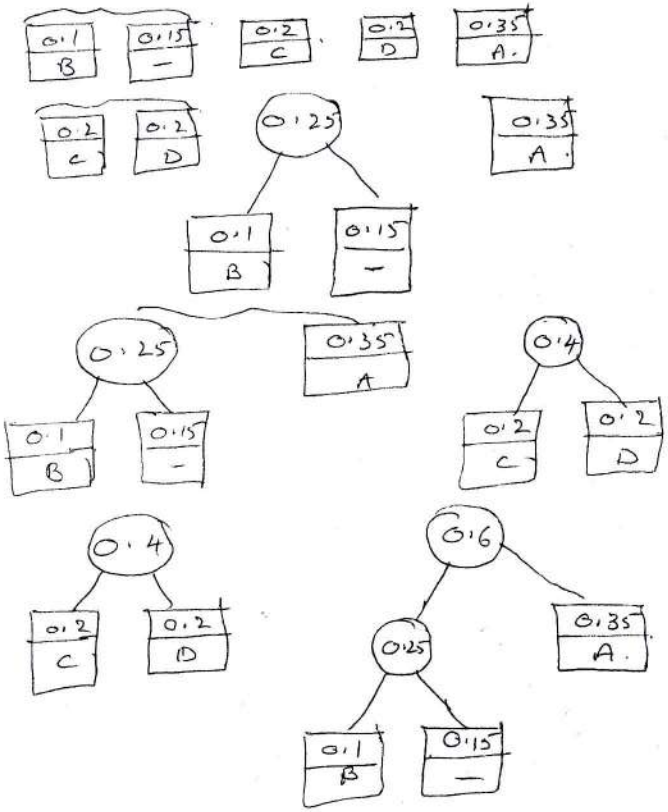
<u>Example</u>

Consider the five-character alphabet {A, B, C, D, –} with the following occurrence probabilities

Character	A	B	C	D	–
Probability	0.35	0.1	0.2	0.2	0.15

Arrange probabilities in ascending order.

Character	B	–	C	D	A
Probability	0.1	0.15	0.2	0.2	0.35



UNIT - IV Coping with the limitations of algorithm power

BACKTRACKING

1. THE GENERAL METHOD.

- In the search for fundamental principles of algorithm design, backtracking represents one of the most general techniques.
- The name backtrack was first coined by D.H. Lehmer in the 1950s.
- In many applications of the backtrack method, the desired solution is expressible as an n-tuple $(x_1, x_2 \cdots x_n)$, where the x_i are chosen from some finite set S_i
- Often the problem to be solved calls for finding one vector that maximizes (or minimizes or satisfies) a ~~criteri~~ criterion function $P(x_1, x_2 \cdots x_n)$.
- Sometimes it seeks all vectors that satisfy P.
- Suppose m_i is the size of set S_i. Then there are $m = m_1 m_2 \cdots m_n$ n-tuples that are possible candidates for satisfying the function P.
- The brute force approach would be to form all these n-tuples, evaluate each one with P, and save those which yield the optimum.
- The backtrack algorithm has the ability to yield the same answer with far fewer than m trails.
- Its basic idea is to build up the solution vector one component at a time and to use modified criterion functions $P_i(x_1, x_2 \cdots x_i)$ to test whether the vector being formed has any chance of success.

Advantage: If it is realized that the partial vector $(x_1, x_2 \cdots x_i)$ can in no way to lead to an optimal solution, then $m_{i+1} \cdots m_n$ possible test vectors can be ignored entirely.

- Many of the problems we solve using backtracking require that all the solutions satisfy a complex set of constraints.
- Constraints can be divided into two categories
  1. Explicit 2. Implicit.
- Explicit constraints are rules that restrict each x_i to take on values only from a given set.

  ex: ① $x_i \geq 0$ or $x_i = 0$ or 1, $l_i \leq x_i \leq u_i$.

  ② $S_i = \{$all nonnegative real nos$\}$, $S_i = \{0, 1\}$.

  Implicit constraints are rules that determine which of the tuples in the solution space of I satisfies the criterion function.
- Thus implicit constraints describe the way in which the x_i must relate to each other.

Example: 8-queens.

- A classic combinatorial problem is to place eight queens on an 8x8 chessboard so that no two attack i.e. no two of them are on the same row column or diagonal.
- All solutions to the 8-queens problem can therefore be represented as 8-tuples $(x_1, x_2 \cdots x_8)$, where x_i the column on which queen i is to be placed on row
- The explicit constraints using this formulation are $S_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$. $1 \leq i \leq 8$.
- The implicit constraints for this problem are that no two x_i's can be the same and no two queens can be on the same diagonal.

Example 2. Sum of Subsets

- Given positive numbers w_i, $1 \le i \le n$, and m, this problem calls for finding all subsets of the w_i whose sums are m.
- All solutions are k-tuples $(x_1, x_2 \cdots x_k)$, $1 \le k \le n$, and different solutions may have different-sized tuples.
- The explicit constraints require $x_i \in \{ j \mid j$ is an integer and $1 \le j \le n \}$.
- The implicit constraints require that no two be the same and that the sum of the corresponding w_i's be m.

ex. $n = 4$, $(w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$, and $m = 31$.
subsets are $(11, 13, 7)$ and $(24, 7)$.

- Backtracking algorithms determine problem solutions by systematically searching the solution space for the given problem instance.
- This search is facilitated by using a tree organization for the solution space.
- For a given solution space many tree organizations may be possible.
- Let $(x_1, x_2 \cdots x_i)$ be a path from the root to a node in a state space tree.
- The tree organization of the solution space is referred to as the state space tree.
- Let $T(x_1, x_2 \cdots x_i)$ be the set of all possible values for x_{i+1} such that $(x_1, x_2 \cdots x_{i+1})$ is also a path to a problem state.
- $T(x_1, x_2 \cdots x_n) = \phi$.

- We assume the existence of bounding function B_{i+1} such that if $B_{i+1}(x_1, x_2 \cdots x_{i+1})$ is false for a path $(x_1, x_2 \cdots x_{i+1})$ from the root node to a problem state, then the path can't be extended to reach an answer node.
- Thus the candidates for position $i+1$ of the solution vector $(x_1, x_2 \cdots x_n)$ are those values which are generated by T and satisfy B_{i+1}.

Recursive Backtracking Algorithm.

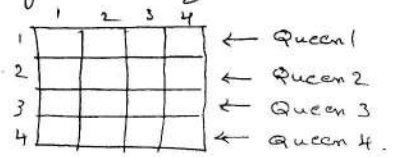
```
Algorithm. Backtrack(k).
// This schema describes the backtracking process
// using recursion. On entering the first k-1
// values x[1], x[2] ... x[k-1] of the solution
// vector x[1:n] have been assigned x[] and n are glo
{
  for (each x[k] ∈ T (x[1], .. x[k-1]) do
  {
    if (B_k(x[1], x[2] .. x[k]) ≠ 0 then
    {
      if (x[1], x[2], ... x[k] is a path
          to an answer node) then
        write (x[1:k]);
      if (k < n) then Backtrack(k+1);
    }
  }
}.
```

General Iterative backtracking method.

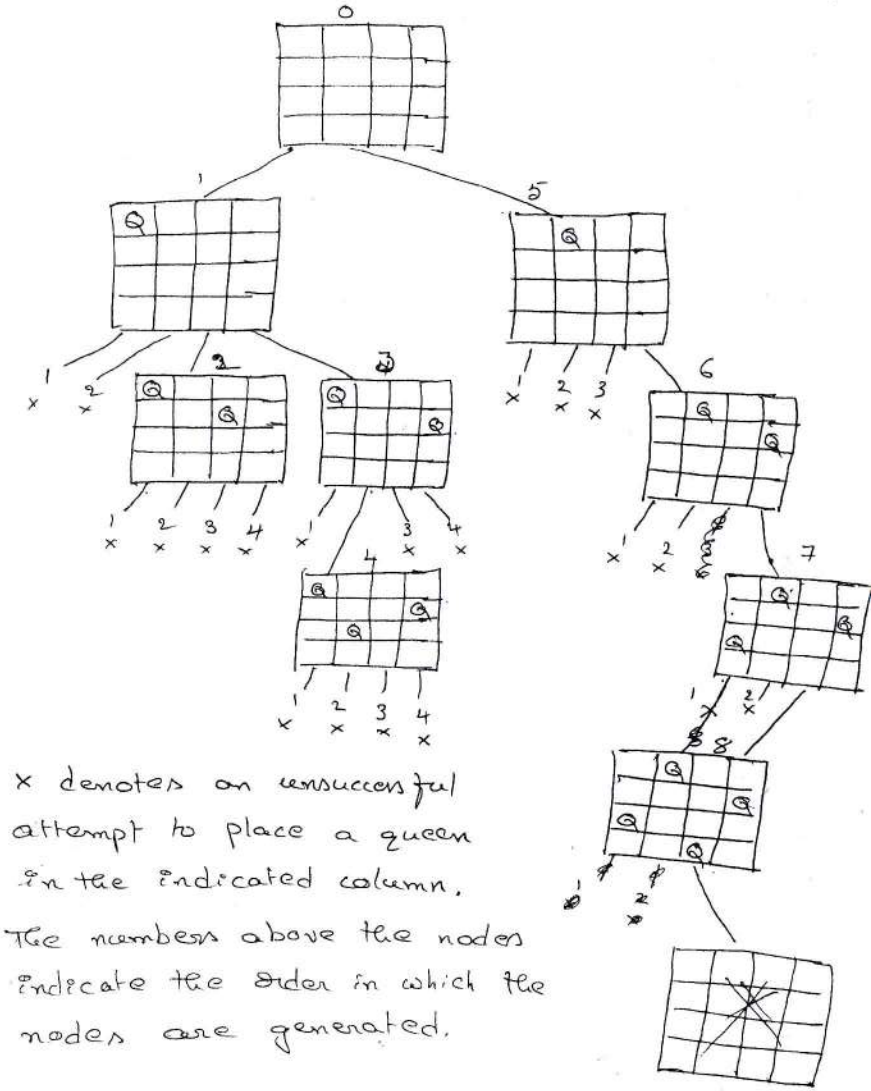
```
Algorithm IBacktrack(n)
// This schema describes the backtracking process.
// All solutions are generated in x[1:n] and
// printed as soon as they are determined.
{ k := 1;
  while(k ≠ 0) do
  {
    if (there remains an untried
      x[k] ∈ T(x[1], x[2] ... x[k-1]) and
      B_k(x[1], ... x[k]) is true) then
    {
      if (x[1], ... x[k] is a path to an
        answer node) then
          write(x[1:k]);
          k := k+1 // consider the next set.
    }
      else
          k := k-1; // Backtrack to the previous set.
  }
}
```

n-Queens Problem.

- The problem is to place n queens on an n-by-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.
- For n=1, the problem has a trivial solution, and it is easy to see that there is no solution for n=2 & n=3
- Let us consider four-queen problem and solve it by backtracking technique
- Since each of the four queens has to be placed in its own row, all we need to do is to assign a column for each queen on the board



- We start with the empty board and then place Queen 1 in the first possible position of its row, which is in column 1 and row 1.
  (1,1) ← Q1
- Then we place Queen 2, after trying unsuccessfully column 1 and 2, in the first acceptable position for it, which is square (2,3), the square in row 2 and column 3
- This proves to be a dead end because there is no acceptable position for queen 3.
- So, the algorithm backtracks and puts queen 2 in the next possible position at (2,4)
- Then queen 3 placed at (3,2), which proves to be another dead end.
- The algorithm then 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) and Queen 4 to (4,3), which is a solution to the problem



- × denotes an unsuccessful attempt to place a queen in the indicated column.
- The numbers above the nodes indicate the order in which the nodes are generated.

- Let $(x_1, x_2 \cdots x_n)$ represent a solution in which x_i is the column of the ith row where the ith queen is placed.
- The x_i's will all be distinct since no two queens ~~are~~ can be placed in the same column.
- If we imagine the chessboard square being numbered as the indices of the two-dimensional array $a[1:n, 1:n]$ then we observe that every element on the same diagonal that runs from the upper left to the lower right has the same row-column value.
- All these squares have a row-column value of 2.
- Also, every element on the same diagonal that goes from the upper right to the lower left has the same row+column value.
- Suppose two queens are placed at positions (i, j) and (k, l).
- Then by the above they are on the same diagonal only if

$$i - j = k - l \quad \text{or} \quad i + j = k + l.$$

The first equation implies

$$j - l = i - k.$$

The second implies

$$j - l = k - i$$

Therefore two queens lie on the same diagonal if and only if $|j - l| = |i - k|$.

- <u>Place(k, i)</u> returns a boolean value that is true if the kth queen can be placed in column i.
- It tests both whether i is distinct from all previous values $x[1] \cdots x[k-1]$ and wether there is no other queen on the same diagonal.
- Its computing time is $O(k-1)$.
- Using Place, we can refine the general backtracking method NQueens(1, n).

<u>Algorithm Place(k, i)</u>

```
// Returns true if a queen can be placed in kth row
// and ith column. Otherwise it returns false. x[] is
// a global array whose first(k-1) values have been set.
// Abs(r) returns the absolute value of r.
{ for j := 1 to k-1 do
    if ((x[j] = i)  or  // Two in the same column.
        (Abs(x[j] - i) = Abs(j-k))) // or in the same diag
    then return false;
  return true;
}.
```

Algorithm NQueens(k, n)

```
// Using backtracking, this procedure prints all
// possible placements of n queens on an nxn
// chessboard so that they are nonattacking.
{ for i := 1 to n do
  {  if Place(k, i) then
     {
        x[k] := i;
        if (k=n) then write (x[1:n]);
        else
           NQueens(k+1, n);
     }
  }
}.
```

Sum of Subsets.

- Given n distinct positive numbers (usually called weights) and we desire to find all combinations of these numbers whose sums are m.
- This is called the sum of subsets problem.
- We could formulate this problem using either fixed- or variable-sized tuples.
- We consider a backtracking solution using the fixed tuple size strategy.
- The element x_i of the solution vector is either one or zero depending on whether the weight w_i is included or not.
- The children of any node are easily generated.
- For a node at level i the left child corresponds to $x_i = 1$ and the right to $x_i = 0$.
- A simple choice for the bounding function is

$B_k(x_1, \cdots x_k)$ = true iff

$$\sum_{i=1}^{k} w_i x_i + \sum_{k=k+1}^{n} w_i \geq m$$

- Clearly $x_1 \cdots x_k$ can't lead to an answer node if this condition is not satisfied.
- The bounding function can be strengthened if we assume that w_i's are initially in nondecreasing order.
- $x_1, \cdots x_k$ can't lead to an answer node if

$$\sum_{i=1}^{k} w_i x_i + w_{k+1} > m$$

- The bounding function is

$B_k(x_1 \cdots x_k)$ = true iff $\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$

and $\sum_{i=1}^{k} w_i x_i + w_{k+1} \leq m$.

Since our algorithm will not make use of B_n, we need not be concerned by the appearance of w_{n+1} in this function.

- Algorithm avoids computing $\sum_{i=1}^{k} w_i x_i$ and $\sum_{i=k+1}^{n} w_i$ each time by keeping these values in variables s and r respectively.
- The algorithm assumes $w_1 \leq m$ and $\sum_{i=1}^{n} w_i \geq m$.
- The initial call is SumOfSub$(0, 1, \sum_{i=1}^{n} w_i)$

```
Algorithm. SumOfSub(s, k, r)
// Find all subsets of w[1:n] that sum to m.
// The values of x[j], 1 ≤ j < k, have already been
// determined. s = Σ_{j=1}^{k-1} w[j]*x[j] and r = Σ_{j=k}^{n} w[j].
// The w[j]'s are in nondecreasing order. It is
// assumed that w[1] ≤ m and Σ_{i=1}^{n} w[i] ≥ m.
{
  // Generate left child. Note s+w[k] ≤ m since B_{k-1} is true.
  x[k] := 1;
  if (s+w[k] = m) then write(x[1:k]);
  // Subset found. There is no recursive call here
  // as w[j] > 0, 1 ≤ j ≤ n.
  else if (s+w[k]+w[k+1] ≤ m)
      then SumOfSub(s+w[k], k+1, r-w[k]);
  // Generate right child and evaluate B_k
  if((s+r-w[k] ≥ m) and (s+w[k+1] ≤ m)) then
  {
    x[k] := 0;
    SumOfSub(s, k+1, r-w[k]);
  }
}
```

Example.

Let $n = 6$, $w[1:6] = \{5, 10, 12, 13, 15, 18\}$ and $m = 30$. Find all possible subsets of w that sum to m. Draw the portion of the state space tree that is generated.

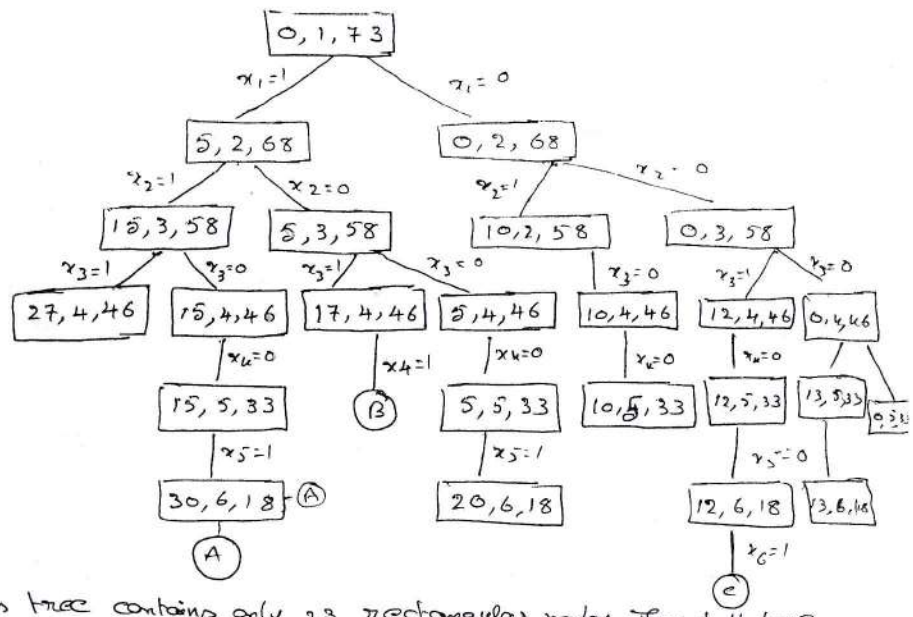
- The rectangular node list the values of s, k and r on each of the calls to SumOfSub.
- Circular nodes represent points at which subsets with sums m are printed out.

$w = \{5, 10, 12, 13, 15, 18\}$.

$A = (1, 1, 0, 0, 1)$, $B = (1, 0, 1, 1)$ $C = (0, 0, 1, 0, 0, 1)$

$\sum_{i=1}^{6} w_i = 73$.

Portion of state space tree.



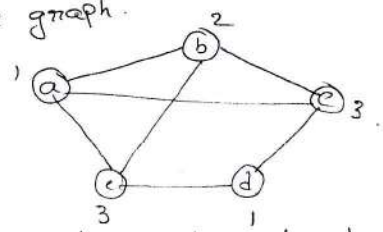
This tree contains only 23 rectangular nodes. The full tree for $n = 6$ contains $2^6 - 1 = 63$ nodes from which calls could be made.

Example

Let $w = \{5, 7, 10, 12, 15, 18, 20\}$ and $m = 35$. Find all possible subsets of w that sum to m. Draw the portion of the state space tree that is generated.

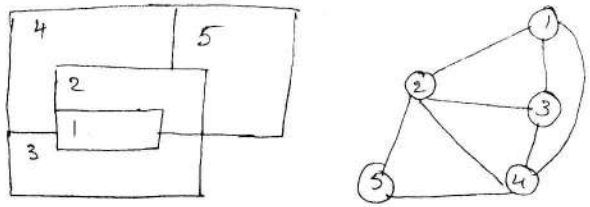
GRAPH COLORING.

- Let G be a graph and m be a given positive integer.
- We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color yet only m colors are used.
- This is termed the m-colorability decision problem.
- If d is the degree of the given graph, then it can be colored with $d+1$ colors.
- The m-colorability optimization problem asks for the smallest integer m for which the graph G can be colored.
- This integer is referred to as the chromatic number of the graph.



- This graph can be colored with three colors 1, 2, and 3.
- The color of each node is indicated next to it.
- It can also be seen that three colors are needed to color this graph and hence this graph's chromatic number is 3.
- A graph is said to be planar iff it can be drawn in a plane in such a way that no two edges cross each other.
- A famous special case of the m-colorability decision problem is the 4-color problem for planar graphs.
- Given a map, can the regions be colored in such a way that no two adjacent regions have the same color ye only four colors are needed?

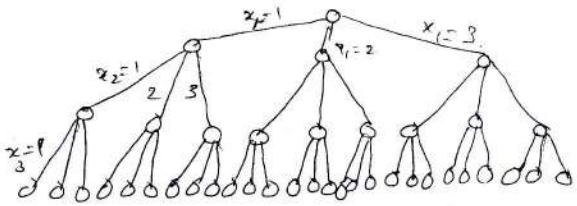
- A map can easily be transformed into graph.
- Each region of the map becomes a node, and if two regions are adjacent then the corresponding nodes are joined by an edge.



Map.

- This map requires four colors.
- We are determining all the different ways in which a given graph can be colored using at most m colors.
- Suppose we represent a graph by its adjacency matrix $G[1:n, 1:n]$, where $G[i,j] = 1$ if if (i,j) is an edge of G and $G[i,j] = 0$ otherwise.
- The colors are represented by the integers $1, 2 \ldots m$ and the solutions are given by the n-tuple $\{x_1, \ldots x_n\}$ where x_i is the color of node i.
- The underlying state space tree used is a tree of degree m and height n+1.
- Each node at level i has m children corresponding to the m possible assignments to x_i, $1 \le i \le n$.
- Nodes at level n+1 are leaf nodes.

The state space tree when n = 3 and m = 3.



Algorithm mColoring(k)

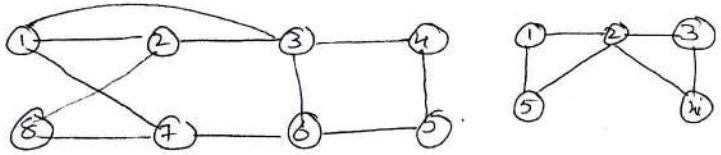
```
// This algorithm was formed using the recursive backtracki
// scheme. The graph is represented by its boolean adjacen
// matrix G[1:n,1:n]. All assignments of 1,2 .. m to the vertices
// of the graph such that adjacent vertices are assigned
// distinct integers are printed. k is the index of the next vertex
// to color.
{ repeat
  { // Generate all legal assignments for x[k]
    NextValue(k); // Assign to x[k] a legal color.
    if (x[k]=0) then return // No new color possible.
    if (k=n) then  // At most m colors have been used to color
                   the n vertices.
      write (x[1:n]);
    else mCololoring(k+1);
  }until (false);
}
```

Algorithm NextValue(k)

```
// x[1] ... x[k-1] have been assigned integer values in the range
// [1, m] such that adjacent vertices have distinct integers. A
// value for x[k] is determined in the range [0,m]. x[k] is assigned
// the next highest numbered color while maintaining distinctness fro
// the adjacent vertices of vertex k. If no such color exists, then x[k] is 0.
{ repeat
  { x[k] := (x[k]+1) mod (m+1); // Next highest color.
    if (x[k]=0) then return; // All colors have been used.
    for j := 1 to n do
    { // check if this color is distinct from adjacent colors.
      if((G[k,j] ≠ 0) and (x[k] = x[j]))
      // If (k,j) is and edge and if adj vertices have the
                 same color.
      the break;
    }
    if (j = n+1) then return; // New color found.
  }until (false); // Otherwise try to find another color.
}
```

HAMILTONIAN CYCLES

- Let $G(V, E)$ be a connected graph with n vertices.
- A Hamiltonian cycle is a round-trip path along 'n' edges of G that visits every vertex once and returns to its starting position.
- In other words, it begins at some vertex $v_1 \in G$ and vertices of G are visited in the order $v_1, v_2 \ldots v_{n+1}$, then the edges (v_i, v_{i+1}) are in E, $1 \le i \le n$, and v_i are distinct except for v_1 and v_{n+1} which are equal.



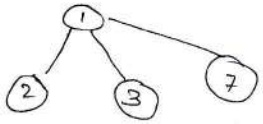
- The Hamiltonian cycle for the graph are

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

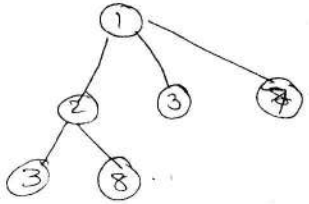
  1 - 3 - 4 - 5 - 6 - 7 - 8 - 2 - 1

- The backtracking solution vector $(x_1, \ldots x_n)$ is defined so that x_i represents the ith visited vertex of the proposed cycle.
- State space tree.

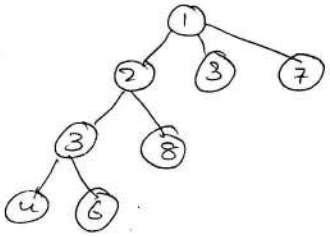
1. The children of node 1 are nodes that are adjacent to 1



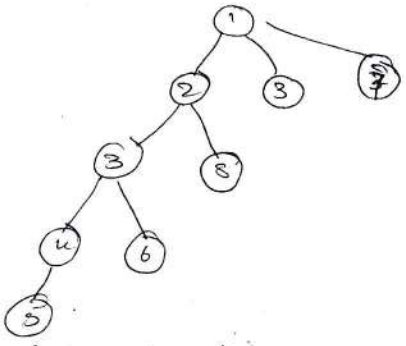
2. The adjacent nodes of 2 are 1, 3, 8. ∵ 1 is already traversed only 3 & 8 are represented



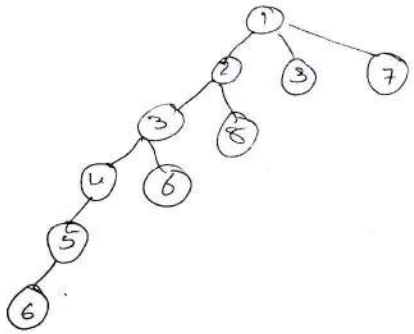
3. The adjacent 3 are 1, 2, 4, 6. Since 1, 2 are traversed. only 4 and 6 are represented.



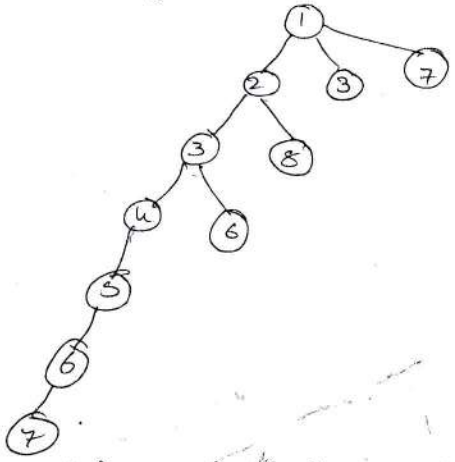
4. The adjacent of 4 are 3, 5. Since 3 are already visited. so 5 only represented



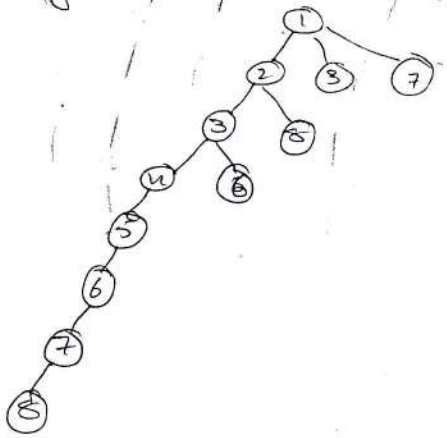
5. The adjacent of are 4, 6. Since 4 is already visited ~~so~~ 6 only represented.



6. The adjacent of 6 are 3, 5, 7. Since 3, 5 are already visited, only 7 is represented.

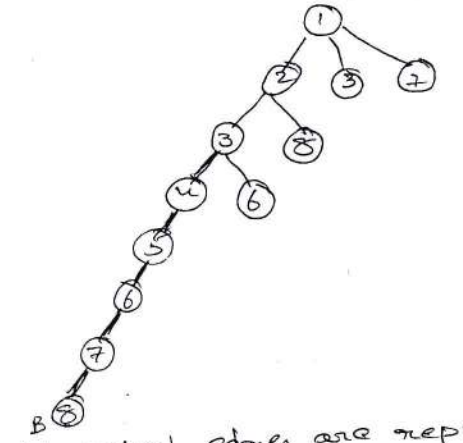


7. The adjacent of 7 are 1, 6, 8. Since 1, 6 are already traversed. only 8 is represented.

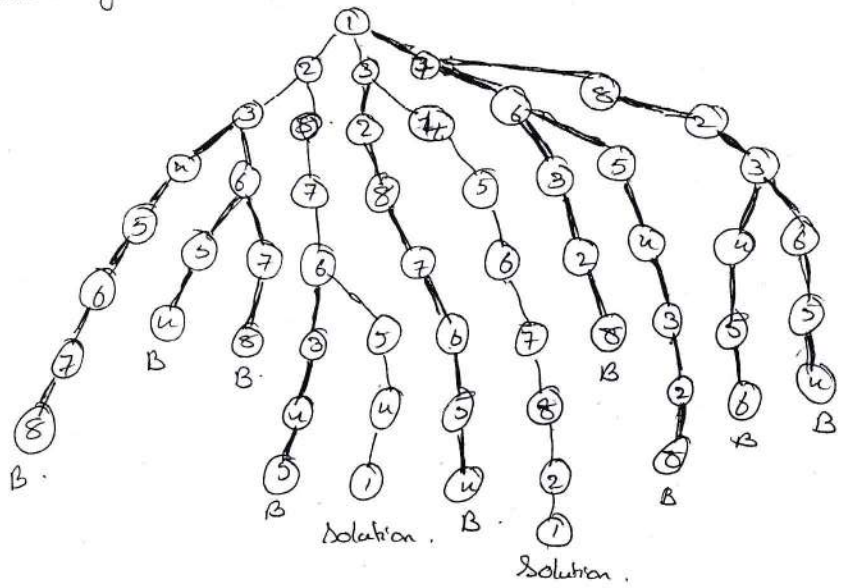


8. The adjacent nodes of 8 are 2, ~~8~~, 7. Since Both are traversed.

9. Backtrack and search for alternate solution.



Backtracked edges are represented with dashed edges. Continuing in the same way we get the following tree.



Algorithm: NextValue(k).

```
// x[1:k-1] is a path of k-1 distinct vertices. If x[k]=0,
// then no vertex has as yet been assigned to x[k].
// After execution, x[k] is assigned to the next
// heighest numbered vertex which does not already
// appear in x[1:k-1] and is connected by an
// edge to x[k-1]. Otherwise x[k] = 0. If k = n then
// in addition x[k] is connected to x[1].
{ repeat
  { x[k] := (x[k]+1) mod (n+1); // next vertex.
    if (x[k] = 0) then return;
    if (G[x[k-1], x[k]) ≠ 0 then
    { // Is there any edge?
      for j := 1 to k-1 do
        if (x[j] = x[k]) then break;
                        // check for distinctness.
        if (j = k) then // If true, then the
                          vertex is distict.
          if ((k < n) or ((k = n) and
                G[x[n], x[1]] ≠ 0)
          then return;
    }
  } until (false);
}.
```

Algorithm Hamiltonian(k)

```
// This algorithm uses the recursive formulation
// of backtracking to find all the Hamiltonian
// cycles of a graph. The graph is stored
// as an adjacency matrix G[1:n, 1:n].
// All cycles begin at node 1.
{
  repeat
  { // Generate values for x[k].
    NextValue(k) // Assign a legal next value to x[
    if (x[k] = 0) then return;
    if (k = n) then write(x[1:n]);
    else Hamiltonian(k+1);
  } until (false);
}.
```

UNIT - IV

P, NP, and NP-Complete Problems.

- The study of the computational complexity of problems, the first concern of both computer scientists and computing profesionals is whether a given problem can be solved in polynomial time by some algorithm.

Def1:-

An algorithm solves the problem in polynomial time if its worst-case time complexity 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 are called tractable.

Intractable. Problems that can't be solved in polynomial time are called intractable.

Reasons for drawing the intractability.

1. We can't solve arbitrary instances of intractable problems in a reasonable amount of time unless such instances are very small.
2. There might be a huge difference between the running times in $O(p(n))$ for polynomials of drastically different degrees.
3. Polynomial functions possess many convenient properties, in particular, both the sum and composition of two polynomials are always polynomials too.

4. The choice of this class has led to a development of an extensive theory called <u>computational complexity</u>, which seeks to classify problems according to their inherent difficulty.

P and NP Problems.

<u>P</u>: Problems that can be solved in polynomial time as the set that computer science theoreticians call P.

- A more formal definition includes in P only <u>decision problems</u>, which are problems with yes/no answers.

Def. 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.

- The restriction of P to decision problems can be justified by the following reasons.

1. It is sensible to exlude problems not solvable in polynomial time because of their componentially large output.
2. Many important problems that are not decision problems in their most natural formulation can be reduced to a series of decision problems that are easier to study.

- Is every decision problem can be solved in

Undecidable.

Some decision problems can't be solved at all by any algorithm. Such problems are called undecidable.

Halting Problem:

The problem in question is called the 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.

- Assume that A is an algorithm that solves the halting problem. That is, for any program P and input I,

$$A(P, I) = \begin{cases} 1, & \text{if program } P \text{ halts on input } I. \\ 0, & \text{if program } P \text{ does not halt on input } I. \end{cases}$$

- There is a large number of important problems, however, for which no polynomial-time algorithm has been found, nor has the impossibility of such an algorithm been proved.

Some best-known problems are.

1. Hamiltonian circuit. Determine whether a given graph has a Hamiltonian circuit.
(A path starts and ends at the same vertex and pass through all the other vertices exactly once)

2 Traveling Salesman. Find the shortest tour through n cities with known positive integer distances between them. (find the shortest Hamiltonian circuit in a complete graph with +ve integer

3. <u>Knapsack problem</u>. Find the most valuable subset of n items of given positive integer weights and values that fit into a knapsack of a given positive integer capacity.

4. <u>Partition Problem</u>. Given n positive integers, determine whether it is possible to partition them into two disjoint subsets with the same sum.

5. <u>Bin Packing</u>. Given n items whose sizes are positive rational numbers not larger than 1, put them into the smallest number of bins of size 1.

6. <u>Graph Coloring</u> For a given graph, find its chromatic number (the smallest number of colors that need to be assigned to the graph's vertices so that no two adjacent vertices are assigned the same color).

7. <u>Integer Linear Programming</u>. Find the maximum (minimum) value of a linear function of several integer-valued variables subject to a finite set of constraints in the form of linear equalities and/or inequalities.

Some of these problems are decision problems.

Def 3.

A nondeterministic algorithm is a two-stage procedure that takes as its input an instance I, of decision problem and does the following.

1. Nondeterministic (Guessing) Stage.

An arbitrary string S is generated that can be thought of as a candidate solution to the given instance I (but may be complete gibberish as well)

2 Deterministic Stage (verification)

A deterministic algorithm takes both I and S as its input and outputs yes if S represents a solution to instance I (if S is not a solution to instance I, the algorithm either return no or allowed not to halt at all)

- A 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 (NP) if the time efficiency of its verification stage is polynomial.

NP Problem.

Def 4

class NP is the class of decision problem that can be solved by nondeterministic polynomial algorithm. This class of problems is called nondeterministic polynomial.

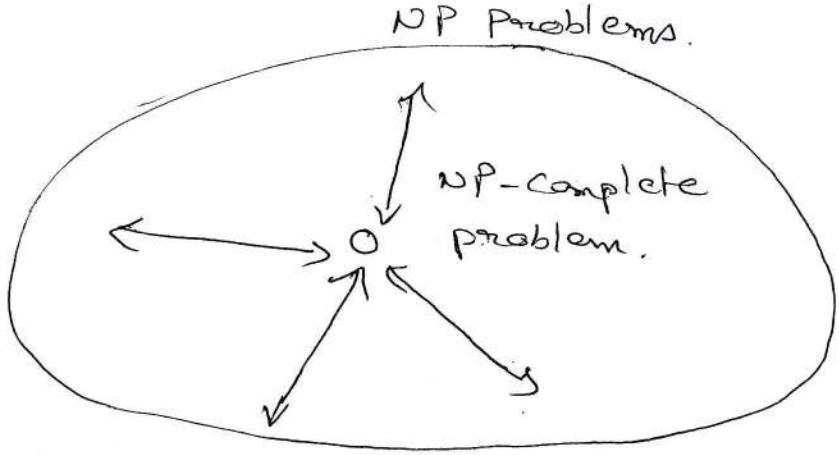
- Most ~~determin.~~ decision problems are in NP.
- First of all, this class includes all the problems in P.

Note that P = NP would imply that each of many hundreds of difficult combinatorial decision problems can be solved by a polynomial-time algorithm.

- Many well known decision problems are known to be NP-complete which seems to cast more doubts on the possibility that P = NP.

NP-Complete Problems

Informally an NP-Complete problem is a problem in NP that is as difficult as any other problem in this class because, by definition, any other problem in NP can be reduced to it in polynomial time.



Def 5.

A decision problem D_1 is said to be <u>polynomially reducible</u> to a decision problem D_2 if there exists a function t that transforms instances of D_1 to instance 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.

Def. 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.

- The notion of NP-completeness requires, however, polynomial reducibility of all problems in NP, both known and unknown, to the problem in question.
- The CNF-satisfiability problem is NP-complete.
- The CNF-satisfiability problem deals with boolean expressions.
- Each boolean expression can be represented in conjunctive normal form, such as the following expression involving three boolean variables x_1, x_2 and x_3 and their negations defined $\bar{x}_1, \bar{x}_2$ and $\bar{x}_3$ respectively.

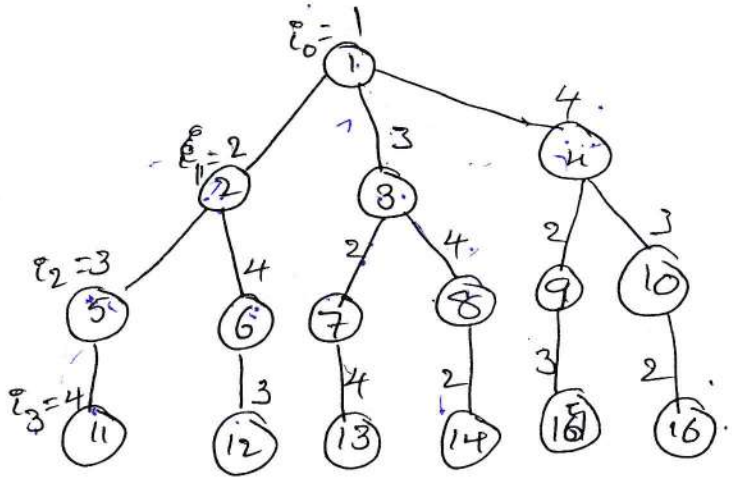
$$(x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$$

- The CNF-satisfiability problem asks whether or not we can assign values true and false to variables of a given boolean expression in the CNF form to make the entire expression true.

$(x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$

TRAVELING SALESPERSON.

- Let $G = (V, E)$ be a directed graph.
- Let c_{ij} equal the cost of edge $\langle i, j \rangle$
- $c_{ij} = \infty$ if $\langle i, j \rangle \notin E$ and let $|V| = n$.
- We can assume that every tour starts and ends at vertex 1.
- The solution space $S = \{ 1, \pi, 1/\pi$ is permutation of $(2, 3 \ldots n) \}$.
- Let $\hat{c}(A)$ be the length of the path defined at node A.
- State space tree for Traveling Salesperson with $n = 4$ & $i_0 = i_4 = 1$.



- The path defined at node 6 is $i_0, i_1, i_2 = 1, 2, 4$ consisting of edges $\langle 1, 2 \rangle$ and $\langle 2, 4 \rangle$.
- A better $\hat{c}(\cdot)$ can be obtained by using the reduced cost matrix of G.
- A row (column) is said to be reduced iff it contains atleast one zero and all remaining entries are non-negative.

- A matrix is reduced iff every row and column is reduced.
- The cost of edge $\langle i, j \rangle$ can be obtained as follows.

1) Change all entries in row i and column j to ∞
2) Set $A(j, i)$ to ∞
3) Reduce the resultant matrix except for rows and columns containing only ∞.

- Solve the following Travelling Salesperson using Branch and Bound.

$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & \infty & 20 & 30 & 10 & 11 \\ B & 15 & \infty & 16 & 4 & 2 \\ C & 3 & 5 & \infty & 2 & 4 \\ D & 19 & 6 & 18 & \infty & 3 \\ E & 16 & 4 & 7 & 16 & \infty \end{array}$$

we have to reduce the row values by using minimum values.

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix} \begin{matrix} 10 \\ 2 \\ 2 \\ 3 \\ 4 \end{matrix} \rightarrow \begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$$

we have to reduce the column values by using minimum values.

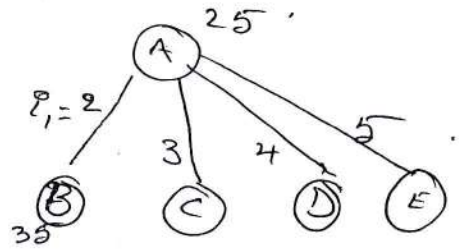
$$\begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 13 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix} \rightarrow \begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 13 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

$$\begin{matrix} 1 & 0 & 3 & 0 & 0. \end{matrix}$$

Total reduced cost

$= 10 + 2 + 2 + 3 + 4 + 1 + 3$

$= 25$



Now cost of edge $\langle 1, 2 \rangle$.

1. set row 1 & col 2 to ∞.
2. Set $(2, 1)$ to ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix} \begin{matrix} -0 \\ -0 \\ -0 \\ -0 \\ -0 \end{matrix}$$

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 \end{matrix}$$

reduced cost $= 0$

cost of $1-2$ = optimal cost + Reduced cost + $M[1][2]$

$= 25 + 0 + 10$

$= 35$

Cost of edge $\langle 1, 3 \rangle$

Set row 1 & col 3 to ∞. set $(3, 1) = \infty$.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 15 & 13 & \infty & \infty & 0 \\ \text{[illegible]} & \text{[illegible]} & \text{[illegible]} & \text{[illegible]} & \text{[illegible]} \end{bmatrix} \begin{matrix} -0 \\ -0 \\ -0 \\ -0 \\ \text{[illegible]} \end{matrix} \rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 13 & \infty & \infty & 0 \\ \text{[illegible]} & \text{[illegible]} & \text{[illegible]} & \text{[illegible]} & \text{[illegible]} \end{bmatrix}$$

reduced cost $= 0 + 11$
$= 11$.

Cost of $\langle 1\text{-}3 \rangle$ = optimal cost + reduced cost + $M[1][3]$
$= 25 + 11 + 17$
$= \underline{53}$.

<u>Cost</u> of edge $\langle 1,4 \rangle$
set row 1 & col 4 to ∞, set $M[4][1] = \infty$.

$$\begin{array}{c|c} \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix} & \begin{matrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{matrix} \\ \begin{matrix} 0 & 0 & 0 & 0 & 0 \end{matrix} & \end{array} \longrightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

Reduced cost $= 0$.

Cost of $\langle 1,4 \rangle$ = Optimal cost + Reduced cost + $M[1][4]$
$= 25 + 0 + 0$
$= \underline{25}$

Cost of edge $\langle 1,5 \rangle$
set row 1 and col 5 to ∞, set $M[5][1] = \infty$.

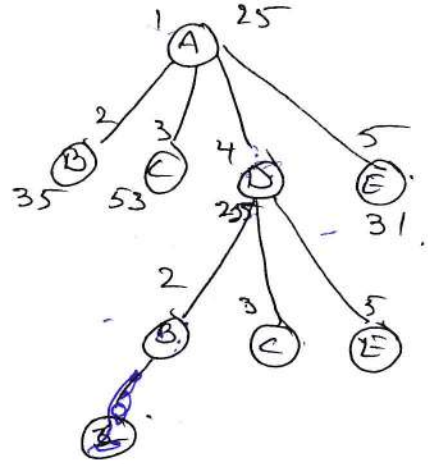
$$\begin{array}{c|c} \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 15 & 3 & 12 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix} & \begin{matrix} 0 \\ 2 \\ 0 \\ 3 \\ 0 \end{matrix} \\ \begin{matrix} 0 & 0 & 0 & 0 & 0 \end{matrix} & \end{array} \longrightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$
$$\begin{matrix} 0 & 0 & 0 & 0 & 0 \end{matrix}$$

Reduced cost $= 2 + 3 = 5$.

Cost of $\langle 1,5 \rangle$ = optimal cost + Reduced cost + $M[1][5]$
$= 25 + 5 + 1$
$= 31$

Among all the paths, path <1, 4> is least cost

The next chances are 1, 4, 2 ; 1, 4, 3 ; 1, 4, 5 .



cost of path 1, 4, 2 . with edges <1, 4> <4 ; 2>

set row 1 & 4, col 4 & 2 to ∞

set $M[4,1]$ & $M[2,4] = \infty$.

$$\begin{array}{c|c} \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix} & \begin{matrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{matrix} \\ \begin{matrix} 0 & 0 & 0 & 0 & 0 \end{matrix} & \end{array}$$

Reduced cost = 0.

cost of (1-4)-2 is

optimal cost + Reduced cost + $M[4, 2]$.

$= 25 + 0 + 3$

$= 28$.

cost of path 1, 4, 3 . with edge <1, 4> <4 3>.

Set rows 1, 4 and col 4, 3 to ∞. $M[4,1] \times M[3,1] = \infty$.

$$\begin{array}{c|c} \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 3 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix} & \begin{matrix} 0 \\ 0 \\ 2 \\ 0 \\ 0 \end{matrix} \end{array} \rightarrow \begin{array}{c} \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix} \\ \begin{matrix} 11 & 0 & 0 & 0 & 0 \end{matrix} \end{array}$$

$$\rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

Reduced cost = 2 + 11 = 13.

cost of (1-4)-3 = optimal cost + Reduced cost + M[4, 3]

= 25 + 13 + 12

= 50

Cost of path (1-4)-5 with edge <1, 4> and <4, 5>

Set rows 1, 4 and col. 4, 5 to ∞. set M[4, 1], M[5, 4] to ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix} \begin{matrix} 0 \\ 11 \\ 0 \\ 0 \\ 0 \end{matrix} \rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 \end{matrix}$$

Reduced cost = 11

cost(1-4)-5 is optimal cost + Reduced cost + M[4, 5]

= 25 + 11 + 0

= 36

Among all the paths (1-4)-2 is minimum.

The next chance is (1-4-2)-3. <1, 4> <4, 2> <2, 3>

Set rows 1, 4, 2 and col. 4, 2, 3 to ∞. set M[4, 1], M[2, 1], M[3, 1] = ∞

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & \infty & \infty & \infty \end{bmatrix} \begin{matrix} 0 \\ 0 \\ 2 \\ 0 \\ 11 \end{matrix} \rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$$

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 \end{matrix}$$

reduced cost = 2 + 11 = 13.

Cost of (1-4-2)-3 is

optimal cost + Reduced cost + M[2, 3]

= 25 + 13 + 11

= 49

The next chance is $(1-4-2)-5$

set rows 1, 4, 2 and columns 4, 2, 5 to α.

Set $M[4,1]$, $M[2,1]$, $M[5,1]$ to α.

$$\begin{array}{cc} \begin{bmatrix} \alpha & \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & \alpha & \alpha \\ 0 & \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & 0 & \alpha & \alpha \end{bmatrix} & \begin{matrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{matrix} \\ \begin{matrix} 0 & 0 & 0 & 0 & 0 \end{matrix} & \end{array}$$

Reduced cost = 0.

$$\text{Cost}(1-4-2)-5 = \text{optimal cost} + \text{Reduced cost} + M[2,5].$$

$$= 25 + 0 + 0$$

$$= 25.$$

9. Find the product of matrices A and B through transformation to a product of two symmetric matrices, if:

$$\text{Transpose of A} = \begin{bmatrix} 1 & -1 \\ 2 & 3 \end{bmatrix}, \quad \text{Transpose of B} = \begin{bmatrix} 0 & 1 \\ -1 & 2 \end{bmatrix}$$

10. a. Can we use this section's formulas that indicate the complexity equivalence of multiplication and squaring of integers to show the complexity equivalence of multiplication and squaring of square matrices?

    b. Show that multiplication of two matrices of order n can be reduced to squaring a matrix of order $2n$.

11. Write a report on the following:

    a. Tight lower bound.

    b. Trivial lower bound.

    c. Information-theoretic lower bounds.

11.2 Decision Trees

Many important algorithms, especially those for sorting and searching, work by comparing items of their inputs. We can study the performance of such algorithms with a device called the ***decision tree***. As an example, Figure 11.1 presents a decision tree of an algorithm for finding a minimum of three numbers. Each internal node of a binary decision tree represents a key comparison indicated in the node, e.g., $k < k'$. The node's left subtree contains the information about subsequent comparisons made if $k < k'$, while its right subtree does the same for the case of $k > k'$. (For the sake of simplicity, we assume throughout this section that all input items are distinct.) Each leaf represents a possible outcome of the algorithm's run on some input of size n. Note that the number of leaves can be greater than the number of outcomes because, for some algorithms, the same outcome

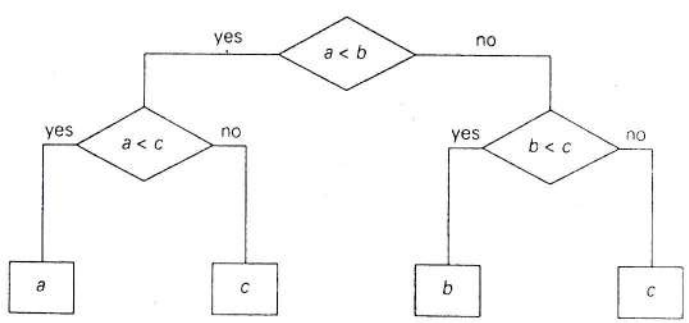


FIGURE 11.1 Decision tree for finding a minimum of three numbers

can be arrived at through a different chain of comparisons. (This happens to be t case for the decision tree in Figure 11.1.) An important point is that the numb of leaves must be at least as large as the number of possible outcomes. The alg rithm's work on a particular input of size n can be traced by a path from the root a leaf in its decision tree, and the number of comparisons made by the algorith on such a run is equal to the number of edges in this path. Hence, the number comparisons in the worst case is equal to the height of the algorithm's decisi tree.

The central idea behind this model lies in the observation that a tree with given number of leaves, which is dictated by the number of possible outcome has to be tall enough to have that many leaves. Specifically, it is not difficult prove that for any binary tree with l leaves and height h,

$$h \geq \lceil \log_2 l \rceil. \qquad (11.$$

Indeed, a binary tree of height h with the largest number of leaves has all its leav on the last level (why?). Hence, the largest number of leaves in such a tree is 2 In other words, $2^h \geq l$, which immediately implies (11.1).

Inequality (11.1) puts a lower bound on the heights of binary decision tre and hence the worst-case number of comparisons made by any comparison-base algorithm for the problem in question. Such a bound is called the ***informatio theoretic lower bound*** (see Section 11.1). We illustrate this technique below c two important problems: sorting and searching in a sorted array.

Decision Trees for Sorting Algorithms

Most sorting algorithms are comparison-based, i.e., they work by comparing el ments in a list to be sorted. Moreover, with the notable exception of bina insertion sort (Problem 9 in Exercises 5.1), comparing two elements is the bas operation of such algorithms. Therefore, by studying properties of decision tre for comparison-based sorting algorithms, we can derive important lower bounc on time efficiencies of such algorithms.

We can interpret an outcome of a sorting algorithm as finding a permutatic of the element indices of an input list that puts the list's elements in ascendir order. For example, for the outcome $a < c < b$ obtained by sorting a list a, c (see Figure 11.2), the permutation in question is 1, 3, 2. Hence, the number possible outcomes for sorting an arbitrary n-element list is equal to $n!$.

Inequality (11.1) implies that the height of a binary decision tree fo any comparison-based sorting algorithm and hence the worst-case number comparisons made by such an algorithm cannot be less than $\lceil \log_2 n! \rceil$:

$$C_{worst}(n) \geq \lceil \log_2 n! \rceil. \qquad (11.$$

Using Stirling's formula for $n!$, we get

$$\lceil \log_2 n! \rceil \approx \log_2 \sqrt{2\pi n}(n/e)^n = n \log_2 n - n \log_2 e + \frac{\log_2 n}{2} + \frac{\log_2 2\pi}{2} \approx n \log_2 n$$

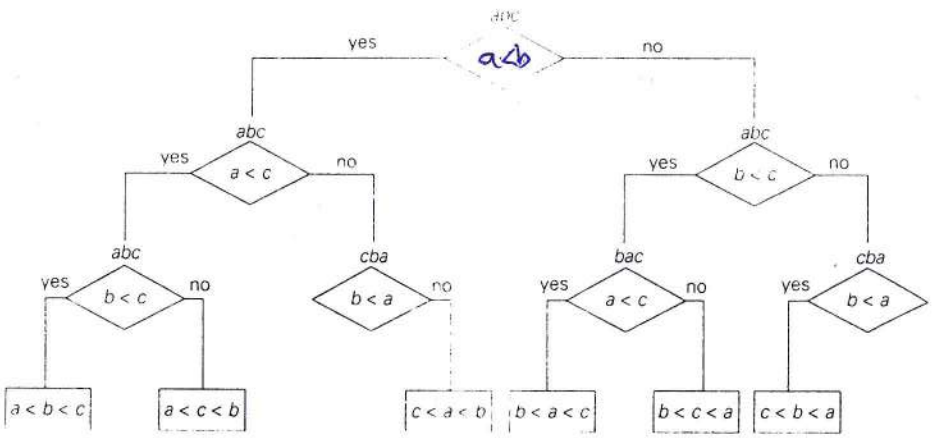


FIGURE 11.2 Decision tree for the three-element selection sort. A triple above a node indicates the state of the array being sorted. Note the two redundant comparisons $b < a$ with a single possible outcome because of the results of some previously made comparisons.

In other words, about $n \log_2 n$ comparisons are necessary to sort an arbitrary n-element list by any comparison-based sorting algorithm. Note that mergesort makes about this number of comparisons in its worst case and hence is asymptotically optimal. This also implies that the asymptotic lower bound $n \log_2 n$ is tight and therefore cannot be substantially improved. We should point out, however, that the lower bound of $\lceil \log_2 n! \rceil$ can be improved for some values of n. For example, $\lceil \log_2 12! \rceil = 29$, but it has been proved that 30 comparisons are necessary (and sufficient) to sort an array of 12 elements in the worst case.

We can also use decision trees for analyzing the average-case behavior of a comparison-based sorting algorithm. We can compute the average number of comparisons for a particular algorithm as the average depth of its decision tree's leaves, i.e., as the average path length from the root to the leaves. For example, for the three-element insertion sort whose decision tree is given in Figure 11.3, this number is $(2 + 3 + 3 + 2 + 3 + 3)/6 = 2\frac{2}{3}$.

Under the standard assumption that all $n!$ outcomes of sorting are equally likely, the following lower bound on the average number of comparisons C_{avg} made by any comparison-based algorithm in sorting an n-element list has been proved:

$$C_{avg}(n) \geq \log_2 n!. \qquad \textbf{(11.3)}$$

As we saw earlier, this lower bound is about $n \log_2 n$. You might be surprised that the lower bounds for the average and worst cases are almost identical. Remember, however, that these bounds are obtained by maximizing the number of comparisons made in the average and worst cases, respectively. For individual sorting

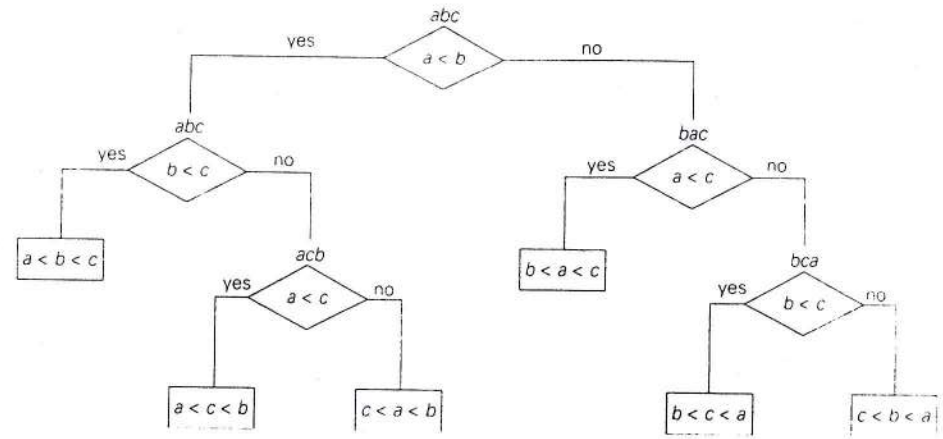


FIGURE 11.3 Decision tree for the three-element insertion sort.

algorithms, the average-case efficiencies can, of course, be significantly better than their worst-case efficiencies.

Decision Trees for Searching a Sorted Array

In this section, we shall see how decision trees can be used for establishing lower bounds on the number of key comparisons in searching a sorted array of n keys: $A[0] < A[1] < \ldots < A[n-1]$. The principal algorithm for this problem is binary search. As we saw in Section 4.3, the number of comparisons made by binary search in the worst case, $C^{bs}_{worst}(n)$, is given by the formula

$$C^{bs}_{worst}(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2 (n+1) \rceil. \qquad \textbf{(11.4)}$$

We will use decision trees to determine whether this is the smallest possible number of comparisons.

Since we are dealing here with three-way comparisons in which search key K is compared with some element $A[i]$ to see whether $K < A[i]$, $K = A[i]$, or $K > A[i]$, it is natural to try using ternary decision trees. Figure 11.4 presents such a tree for the case of $n = 4$. The internal nodes of that tree indicate the array's elements being compared with the search key. The leaves indicate either a matching element in the case of a successful search or a found interval that the search key belongs to in the case of an unsuccessful search.

We can represent any algorithm for searching a sorted array by three-way comparisons with a ternary decision tree similar to that of Figure 11.4. For an array of n elements, all such decision trees will have $2n + 1$ leaves (n for successful searches and $n + 1$ for unsuccessful ones). Since the minimum height h of a ternary tree with l leaves is $\lceil \log_3 l \rceil$, we get the following lower bound on the number of

worst-case comparisons:

$$C_{worst}(n) \geq \lceil \log_3(2n+1) \rceil.$$

This lower bound is smaller than $\lceil \log_2(n+1) \rceil$, the number of worst-case comparisons for binary search, at least for large values of n (and smaller than or equal to $\lceil \log_2(n+1) \rceil$ for every positive integer n—see Problem 7 in the exercises). Can we prove a better lower bound, or is binary search far from being optimal? The answer turns out to be the former. To obtain a better lower bound, we should consider binary rather than ternary decision trees such as the one in Figure 11.5. Internal nodes in such a tree correspond to the same three-way comparisons as before, but they also serve as terminal nodes for successful searches. Leaves therefore represent only unsuccessful searches, and there are $n+1$ of them for searching an n-element array.

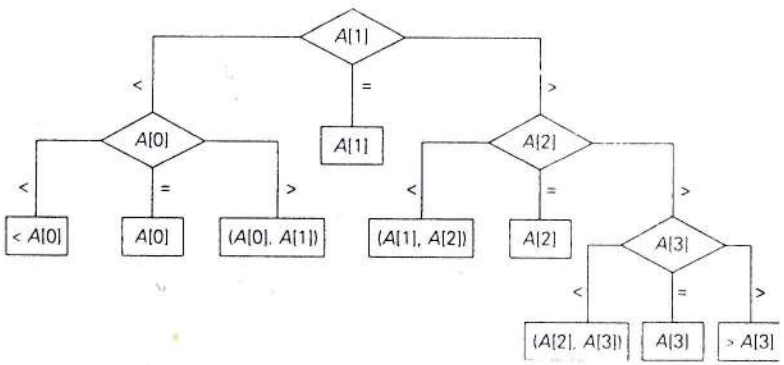


FIGURE 11.4 Ternary decision tree for binary search in a four-element array

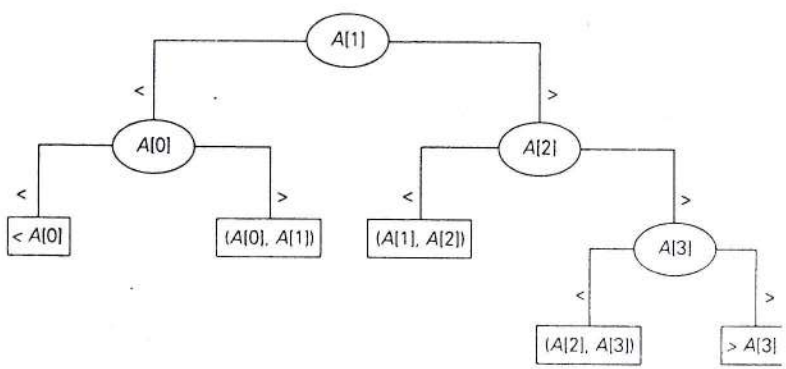


FIGURE 11.5 Binary decision tree for binary search in a four-element array

As comparison of the decision trees of Figures 11.4 and 11.5 illustrates, the binary decision tree is simply the ternary decision tree with all the middle subtrees eliminated. Applying inequality (11.1) to such binary decision trees immediately yields

$$C_{worst}(n) \geq \lceil \log_2(n+1) \rceil. \tag{11.5}$$

This inequality closes the gap between the lower bound and the number of worst-case comparisons made by binary search, which is also $\lceil \log_2(n+1) \rceil$. A much more sophisticated analysis (see, e.g., [Knu11], Section 6.2.1) shows that under the standard assumptions about searches, binary search makes the smallest number of comparisons on the average, as well. The average number of comparisons made by this algorithm turns out to be about $\log_2 n - 1$ and $\log_2(n+1)$ for successful and unsuccessful searches, respectively.

Exercises 11.2

1. Prove by mathematical induction that
   a. $h \geq \lceil \log_2 l \rceil$ for any binary tree with height h and the number of leaves l.
   b. $h \geq \lceil \log_3 l \rceil$ for any ternary tree with height h and the number of leaves l.
2. Consider the problem of finding the median of a three-element set $\{a, b, c\}$.
   a. What is the information-theoretic lower bound for comparison-based algorithms solving this problem?
   b. Draw a decision tree for an algorithm solving this problem.
   c. If the worst-case number of comparisons in your algorithm is greater than the information-theoretic lower bound, do you think an algorithm matching the lower bound exists? (Either find such an algorithm or prove its impossibility.)
3. Draw a decision tree and find the number of key comparisons in the worst and average cases for:
   a. The five-element binary search.
   b. The three-element basic bubble sort.
   c. The three-element enhanced bubble sort (which stops if no swaps have been made on its last pass).
4. Design a comparison-based algorithm for searching a four-element sorted array.
5. Design a comparison-based algorithm for sorting a five-element array with seven comparisons in the worst case.
6. Draw a decision tree for a five-element insertion sort.
7. Compare the two lower bounds for searching a sorted array—$\lceil \log_3(2n+1) \rceil$ and $\lceil \log_2(n+1) \rceil$—to show that

* Problem Reduction

The way professor x approached his task is an example of an important problem-solving strategy called problem reduction. If you need to solve a problem reduce it to another problem that you know how to solve.

Problem 1 (to be solved) $\xrightarrow{\text{reduction}}$ Problem 2 (solvable by alg. A) $\xrightarrow{\text{alg. A}}$ Solution to problem 2

Problem reduction strategy

Computing the Least common multiple

Recall that the least common multiple of two positive integers m and n, denoted Lcm (m,n), is defined as the smallest integer that is divisible by both m and n. For example, Lcm (24, 60) = 120, and Lcm (11, 5) = 55. The least common multiple is one of the most important notions in elementary arithmetic and algebra; perhaps you remember the following middle-school method for computing it.

$$24 = 2.2.2.3$$

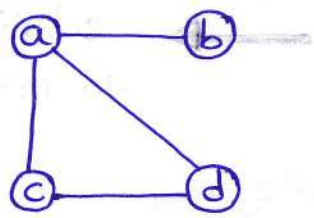
$$60 = 2.2.3.5$$

$$Lcm\ (24, 60) = (2.2.3).2.5 = 120$$

Counting paths in a Graph

The problem of counting a graph's paths can be solved with an algorithm for computing an appropriate power of its adjacency matrix.

figure. Its adjacency matrix A and its square A^2 indicate the number of paths of length 1 and 2, respectively, between the corresponding vertices of the graph. In particular, there are three paths of length 2 that start and end at vertex a: a-b-a, a-c-a, and a-d-a but only one path of length 2 from a to c: a-d-c.



$$A = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 1 & 1 \\ b & 1 & 0 & 0 & 0 \\ c & 1 & 0 & 0 & 1 \\ d & 1 & 0 & 1 & 0 \end{array}$$

$$A^2 = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 3 & 0 & 1 & 1 \\ b & 0 & 1 & 1 & 1 \\ c & 1 & 1 & 2 & 1 \\ d & 1 & 1 & 1 & 2 \end{array}$$

A graph, its adjacency matrix A, and its square A^2. The elements of A and A^2 indicate the number of paths of lengths 1 and 2 respectively.

Reduction of optimization problems

If a problem seeks to find a maximum of some function, it is said to be a maximization problem; if it seeks to find a function's minimum, it is said to be a minimization problem. Suppose now that you have to find a minimum of some function f(x) and you known an algorithm for maximizing the function.

$$\max f(x) = -\min[-f(x)]$$

Linear programming

many problems of optimal decision making can be reduced to an instance of the linear programming problem, which is a problem of optimizing a linear function of several variables subject to constraints in the form of linear equations and linear inequalities.

Example :-

Let us create a mathematical model of this problem. Let x, y, and z be the amounts (in millions of dollars) invested in stocks, bonds, and cash, respectively. By using these variables, we can pose the following optimi-zation problem:

$$\text{maximize } 0.10x + 0.07y + 0.03z$$

$$\text{subject to } x + y + z = 100$$

$$x \le \tfrac{1}{3} y$$

$$z \ge 0.25(x+y)$$

$$x \ge 0,\ y \ge 0,\ z \ge 0.$$

Although this particular problem is both small and simple, it does show how a problem of optimal decision making can be reduced to an instance of the general linear programming problem.

$$\text{maximize (or minimize) } c_1x_1 + \dots + c_nx_n$$

$$\text{subject to } a_{i1}x_1 + \dots + a_{in}x_n \le (\text{or} \ge \text{or} =) b_i$$

* Lower - Bound Arguments

When we want to ascertain the efficiency of an algorithm with respect to other algorithms for the same problem, it is desirable to know the best possible efficiency any algorithm solving the problem may have. Knowing such a lower bound can tell us how much improvement we can hope to achieve in our quest for a better algorithm for the problem in question. If such a bound is tight, i.e, we already known an algorithm in the same efficiency class of the lower bound.

Trivial Lower Bounds

The simplest method of obtaining a lower-bound class is based on counting the number of items in the problem's input that must be processed and the number of output items that need to be produced. Since any algorithm must at least "read" all the items that it needs to process and "write" all its outputs, such a count yields a trivial lower bound.

Information - Theoretic Arguments

The approach we Just exploited is called the information-theoretic ~~arog~~ argument because of its connection to information theory. It has proved to be quite useful for finding the so-called information-theoretic lower bounds for many problems involving comparisons

of decision trees.

Adversary Arguments :-

Let us revisit the same game of "guessing" a number used to introduce the idea of an information-theoretic argument. we can prove that any algorithm that solves this problem must ask at least $\lceil \log_2 n \rceil$ questions in its worst case by playing the role of a hostile adversary who wants to make an algorithm ask as many questions as possible. The adversary starts by considering each of the numbers between 1 and n as being potentially selected.

* Branch-and-Bound

In general, we 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 :

1) The value of the node's bound is not better than the value of the best solution seen so far.

2) The node represents no feasible solutions because the constraints of the problem are already violated.

3) The subset of feasible solutions represented by the node consists of a single point (and hence no further choices can be made) - in this case we compare the value of the objective function for this feasible solution with that of the best solution seen so far

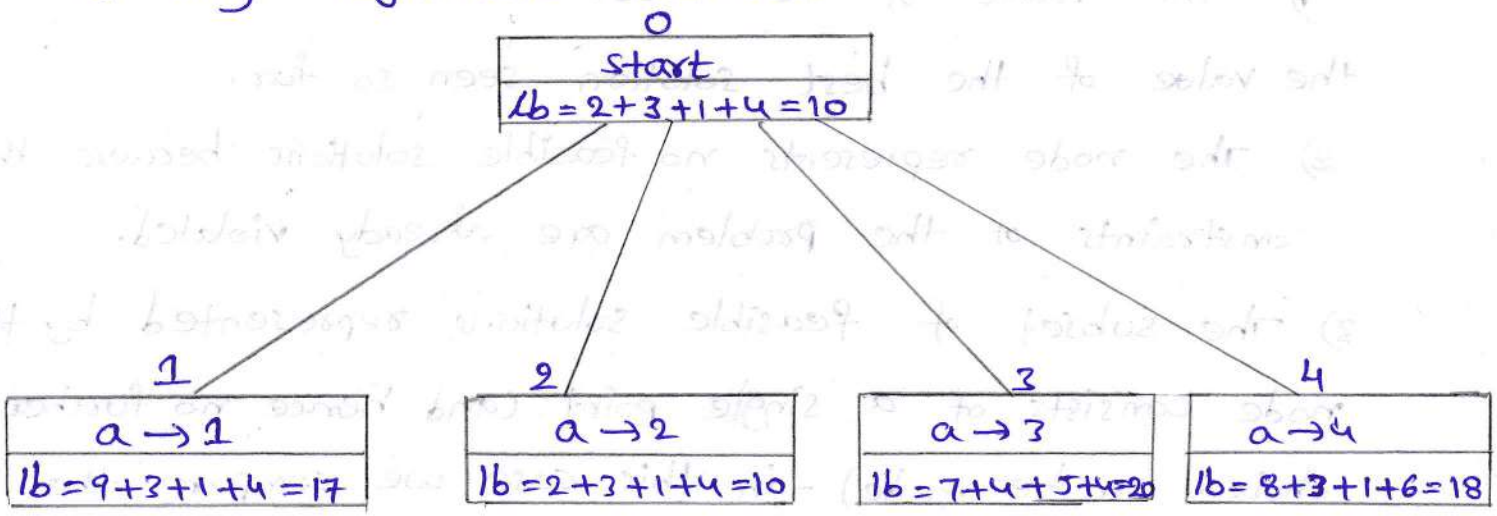
solution is better.

Assignment problem

Let us illustrate the branch-approach by applying it to the problem of assigning n people to n jobs to so that the total cost of the assignment is as small as possible. we can state the problem as follows: 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.

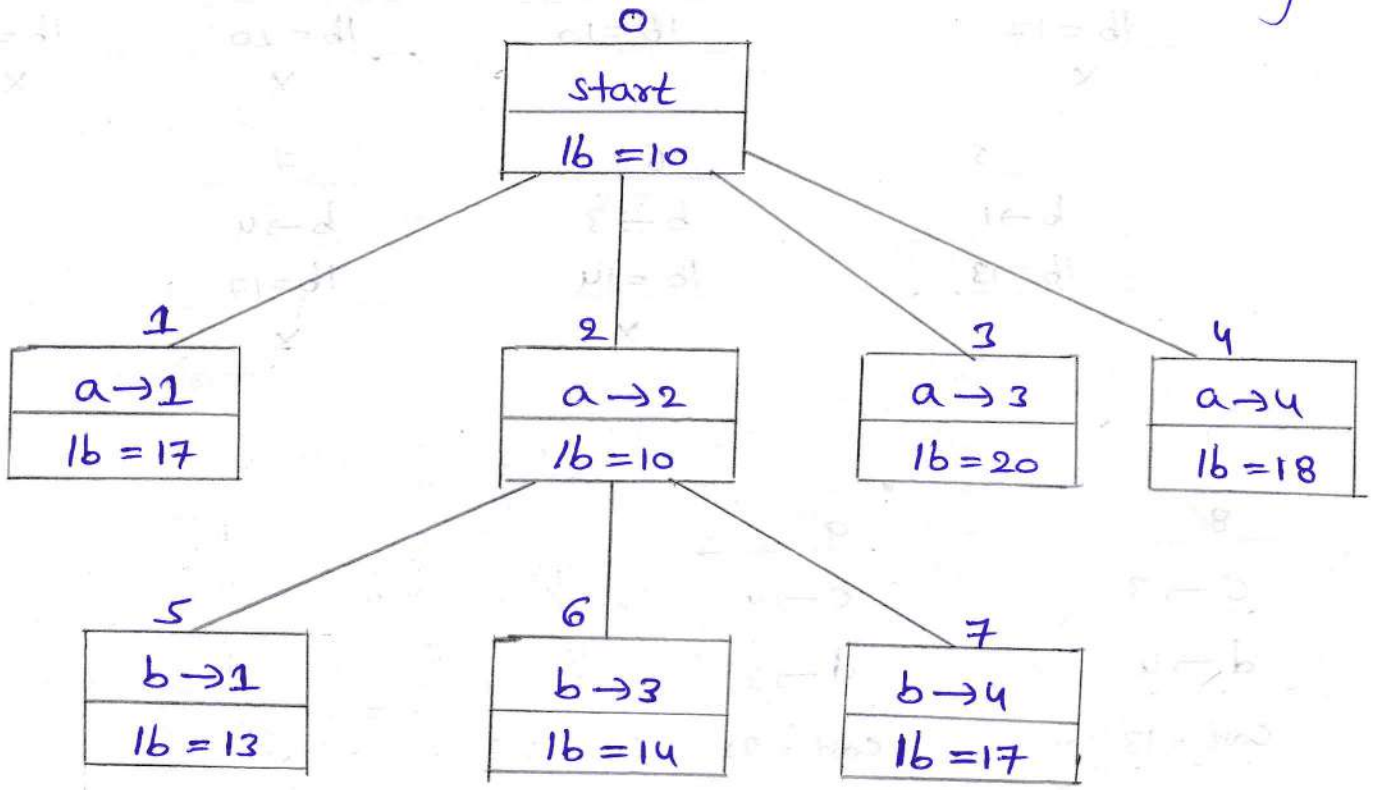
$$
\begin{array}{cc}
 & \begin{array}{cccc} \text{Job1} & \text{Job2} & \text{Job3} & \text{Job4} \end{array} \\
C = & \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}
\end{array}
\begin{array}{l} \text{person a} \\ \text{person b} \\ \text{person c} \\ \text{person d} \end{array}
$$

For the instance here, this sum is $2+3+1+4=10$. It is important to stress that this is not the cost of any legitimate selection.



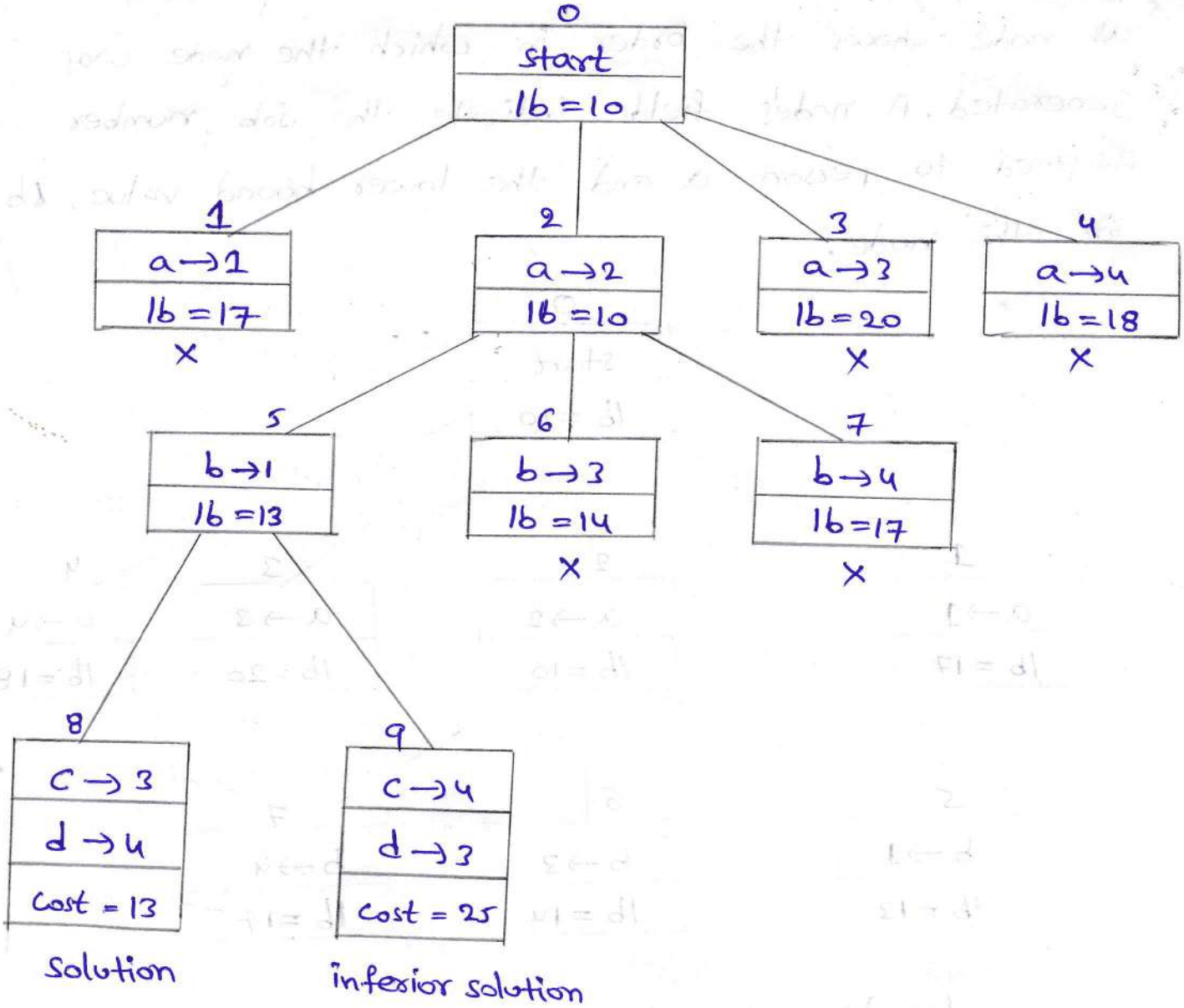
Levels 0 and 1 of the state-space tree for the instance of the assignment problem

a node shows the order in which the node was generated. A node's fields indicate the job number assigned to person a and the lower bound value, lb, for this node.



Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm.

Blank



Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm.

Knapsack problem

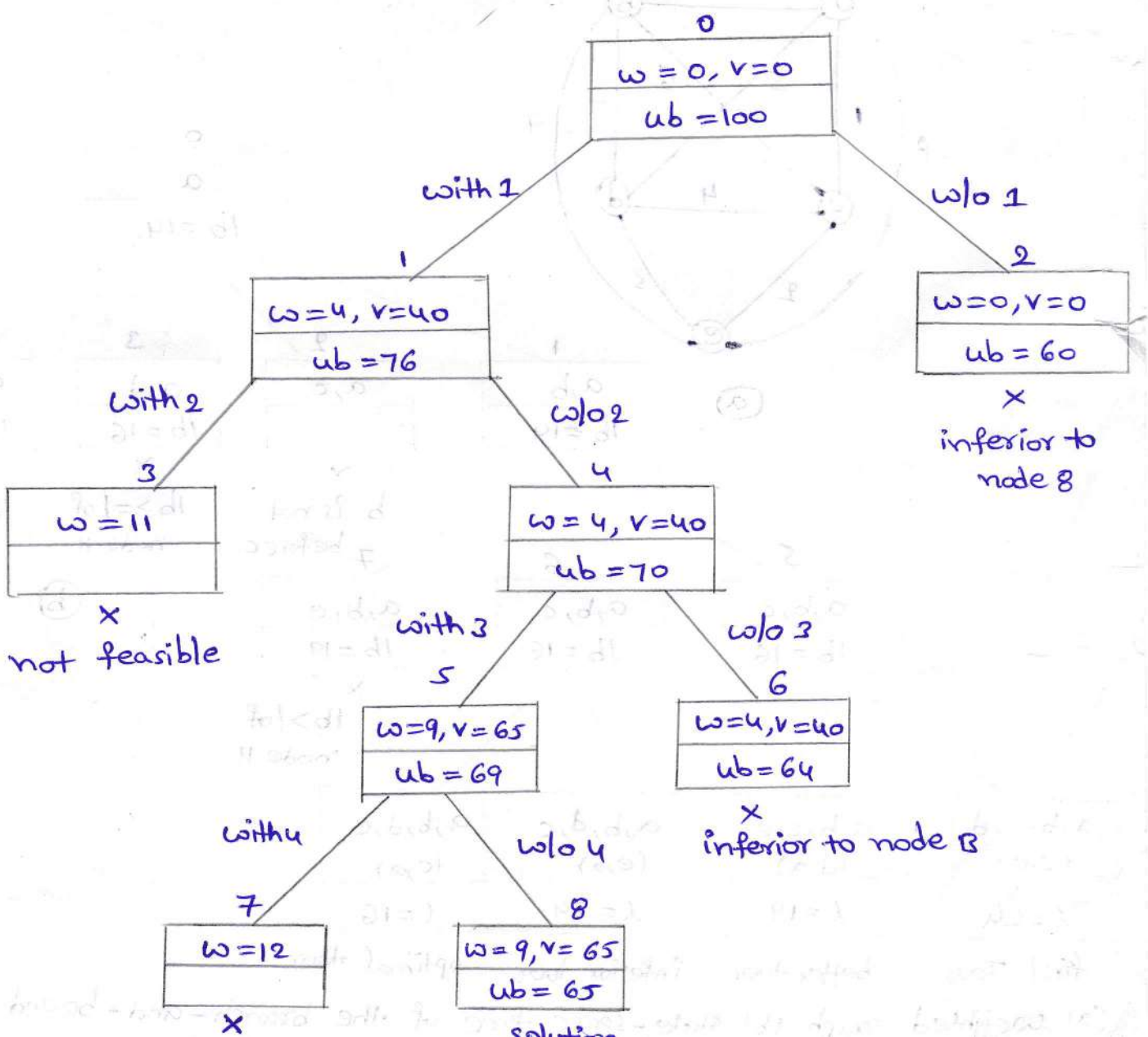
Let us now discuss how we can apply the branch-and-bound technique to solving the knapsack problem. n items of known weights w_i and values v_i, $i = 1, 2, \ldots n$ and a knapsack of capacity w, find the

Knapsack.

$$v_1/w_1 \geq v_2/w_2 \geq \cdots\cdots \geq v_n/w_n$$

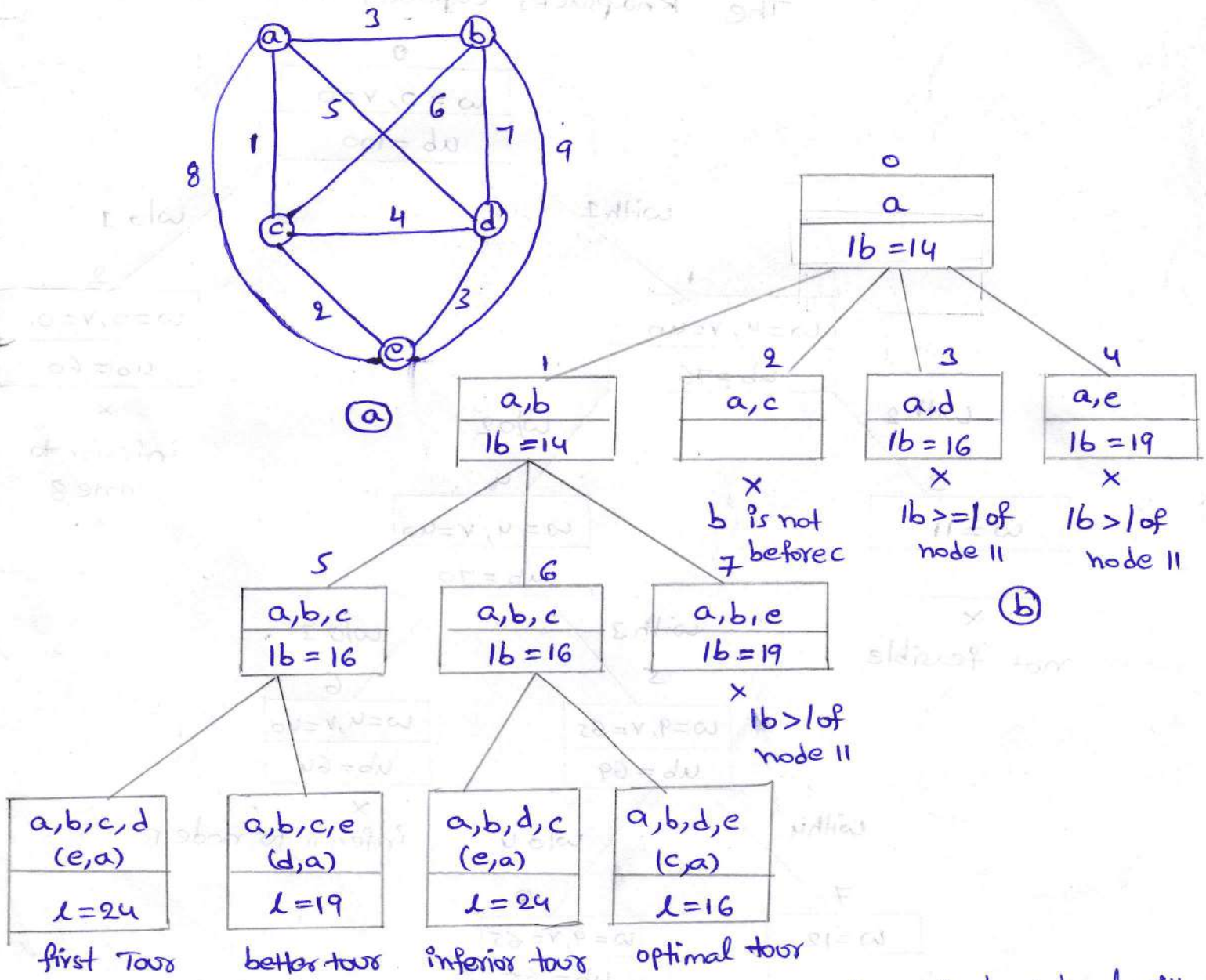
item	weight	value	$\frac{\text{value}}{\text{weight}}$
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4

The knapsack's capacity w is 10.



Travelling salesman problem

we will be able to apply the branch-and-bound technique to instance of the traveling salesman problem if we come up with a reasonable lower bound on tour lengths. one very simple lower bound can be obtained by finding the smallest element in the intercity distance matrix D and multiplying it by the number of cities n. But there is a less obvious and more informative lower bound, which does not require a lot of work to compute.



(a) weighted graph (b) state-space tree of the branch-and-bound algorithm