80+20=100

Design & Analysis of Algorithms

Sub Code : 10CS43

UNIT-I

Introduction: Notion of Algorithm, Review of Asymptotic Notations, Mathematical Analysis of Non-Recursive & Recursive Algorithms.
Brute Force Approaches: Introduction, Selection Sort & Bubble Sort, Sequential Search & Brute Force String Matching.

What is an Algorithm? (Notion of Algorithm)

* Algorithm is a sequence of unambiguous instructions for solving a problem. ie for obtaining required o/p for any legitimate i/p in finite amount of time.
* This definition can be illustrated by a simple diagram as

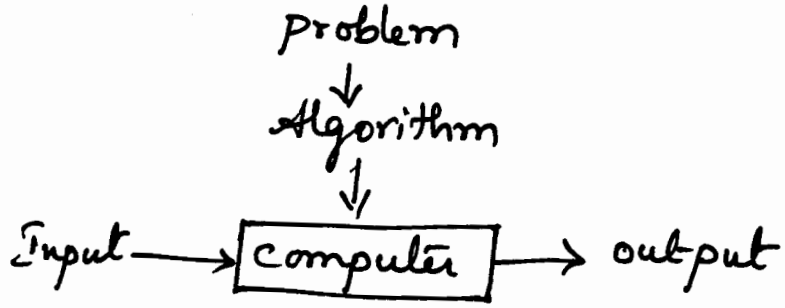


fig: Notion of Algorithm.

* To illustrate notion of an Algorithm, we consider three methods for solving same problem [Case study: GCD of two numbers]

[Prof. T. SATISH KUMAR
Dept of CSE, RNSIT]

These examples will help us to illustrate several important Points [Important characteristics of Algorithms]

→ The Non ambiguity requirement for each step of an algorithm cannot be compromised.

→ Range of input for which an algorithm works has to be specified carefully.

→ The same algorithm can be represent in several different ways.

→ Several algorithm for solving the same problem may exist.

→ Different algorithms for same problem can be based on different ideas & can solve the problem with dramatically different speeds.

Euclid's Algorithm for computing GCD of two no:s

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

Ex $gcd(60,24) = gcd(24,12)$

$= gcd(12,0)$

$\therefore gcd(60,24) = 12$

Algorithm Euclid (m,n)
// Computes gcd (m,n) by Euclid's Algorithm
// I/P: Two non negative, not-both-zero integers m & n
// O/P: GCD of m & n
while $n \neq 0$ do
    $r \leftarrow m \bmod n$
    $m \leftarrow n$
    $n \leftarrow r$
return m

Consecutive integer checking algorithm for computing gcd(m,n)

[unlike Euclid's Algorithm this algorithm will not work if i/p is zero]

Step1: Assign the value of min{m,n} to t.

Step2: Divide m by t. If the remainder of this division is 0, goto Step3; otherwise, goto step4.

Step3: Divide n by t. If the remainder of this division is 0, return the value of t as the answer & stop; otherwise proceed to step4.

Step4: Decrease the value of t by 1. Goto Step2.

Middle School Procedure

Step1: Find the prime factors of m

Step2: Find the prime factors of n

Step3: Identify all the common factors in two prime expansion found in step1 & step2.

Step4: Compute the product of all common factors & return it as the gcd of the numbers given.

Ex1: $60 = (2) \times (2) \times 3 \times 5$

$24 = (2) \times (2) \times 2 \times 3$

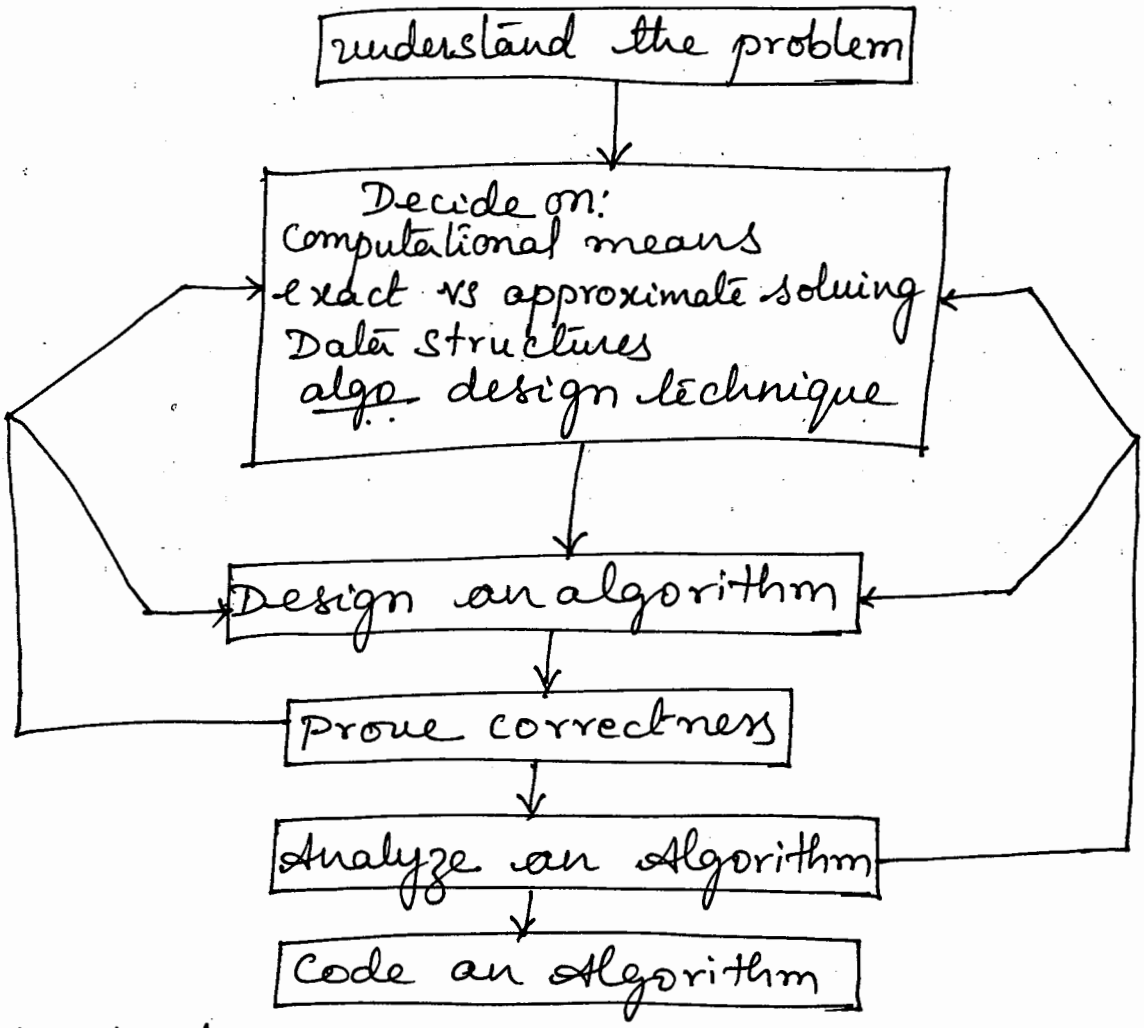
ie $2 \times 2 \times 3 = 12$.

Q. Discuss the various stages of Algorithm design & analysis process using flowchart

OR

Q with a neat figure, explain algorithm development process.

Below figure shows the sequence of steps involved in designing & analysing an Algorithm



understand the problem

* Given a problem; we have to understand it completely & clearly. In the desciption of the problem we should not be having any doubts
* The designer should be capable to understand what information is missing.
* we should know what i/p will be given & what o/p should we get.
* Importantly, we should specify the exact range of the i/p.

Algorithm Design Techniques

- An ***algorithm design technique*** (or "strategy" or "paradigm") is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.
- Various algorithmic design techniques are available such as Brute Force, Divide & Conquer, Dynamic Programming, Decease & Conquer, Back Tracking, etc..
- We can select any one of the suitable design technique to our algorithm.

Designing an Algorithm and Data Structures

- While the algorithm design techniques do provide a powerful set of general approaches to algorithmic problem solving, designing an algorithm for a particular problem may still be a challenging task. Some design techniques can be simply inapplicable to the problem in question.
- Sometimes, several techniques need to be combined, and there are algorithms that are hard to pinpoint as applications of the known design techniques.
-

Methods of Specifying an Algorithm

- These are the two options that are most widely used nowadays for specifying algorithms.
- Using a natural language has an obvious appeal; however, the inherent ambiguity of any natural language makes a succinct and clear description of algorithms surprisingly difficult.
- Nevertheless, being able to do this is an important skill that you should strive to develop in the process of learning algorithms.
- ***Pseudocode*** is a mixture of a natural language and programming language like constructs.
- Pseudocode is usually more precise than natural language, and its usage often yields more succinct algorithm descriptions.
- In the earlier days of computing, the dominant vehicle for specifying algorithms was a ***flowchart***, a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps.
- This representation technique has proved to be inconvenient for all but very simple algorithms; nowadays, it can be found only in old algorithm books.

Proving an Algorithm's Correctness

- Once an algorithm has been specified, you have to prove its ***correctness***.
- We have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time.
- For example, the correctness of Euclid's algorithm for computing the greatest common divisor stems from the correctness of the equality gcd*(m, n)* = gcd*(n, m* mod *n)*, the simple observation that the second integer gets smaller on every iteration of the algorithm, and the fact that the algorithm stops when the second integer becomes 0.
- For some algorithms, a proof of correctness is quite easy; for others, it can be quite complex. A common technique for proving correctness is to use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs.

T. SATISH KUMAR
Asst. Prof.
Dept. of CSE
RNS Institute of Technology
Bangalore - 560 098.
Mob :

Understanding the Problem

- The first thing you need to do before designing an algorithm is to understand completely the problem given. Read the problem's description carefully and ask questions if you have any doubts about the
- An input to an algorithm specifies an ***instance*** of the problem the algorithm solves.
- It is very important to specify exactly the set of instances the algorithm needs to handle. (As an example, recall the variations in the set of instances for the three greatest common divisor algorithms discussed in the previous section.)
- If you fail to do this, your algorithm may work correctly for a majority of inputs but crash on some "boundary" value.
- Remember that a correct algorithm is not one that works most of the time, but one that works correctly for *all* legitimate inputs.

Ascertaining the Capabilities of the Computational Device

- Once you completely understand a problem, you need to ascertain the capabilities of the computational device the algorithm is intended for.
- The vast majority of algorithms in use today are still destined to be programmed for a computer closely resembling the von Neumann machine—a computer architecture outlined by the prominent Hungarian-American mathematician John von Neumann, in collaboration with A. Burks and H. Goldstine.
- 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, i.e., in parallel. Algorithms that take advantage of this capability are called ***parallel algorithms***.
- Even the "slow" computers of today are almost unimaginably fast. Consequently, in many situations you need not worry about a computer being too slow for the task.
- There are important problems, however, that are very complex by their nature, or have to process huge volumes of data, or deal with applications where the time is critical. In such situations, it is imperative to be aware of the speed and memory available on a particular computer system.

Choosing between Exact and Approximate Problem Solving

- The next principal decision is to choose between solving the problem exactly or solving it approximately. In the former case, an algorithm is called an ***exact algorithm***; in the latter case, an algorithm is called an ***approximation algorithm***.
- Why would one opt for an approximation algorithm?
    - First, there are important problems that simply cannot be solved exactly for most of their instances; examples include extracting square roots, solving nonlinear equations, and evaluating definite integrals.
    - Second, available algorithms for solving a problem exactly can be unacceptably slow because of the problem's intrinsic complexity.

Analyzing an Algorithm

- After correctness, by far the most important is ***efficiency***.
- In fact, there are two kinds of algorithm efficiency: ***time efficiency***, indicating how fast the algorithm runs, and ***space efficiency***, indicating how much extra memory it uses.
- A general framework and specific techniques for analyzing an algorithm's efficiency appear in Chapter 2.
- Another desirable characteristic of an algorithm is ***simplicity***.
- Unlike efficiency, which can be precisely defined and investigated with mathematical rigor, simplicity, like beauty, is to a considerable degree in the eye of the beholder.
- Yet another desirable characteristic of an algorithm is ***generality***.
- There are, in fact, two issues here: generality of the problem the algorithm solves and the set of inputs it accepts. On the first issue, note that it is sometimes easier to design an algorithm for a problem posed in more general terms.

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.

Important Problem Types

The important problem types are:

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

Sorting

- The ***sorting problem*** is to rearrange the items of a given list in nondecreasing order.
- As a practical matter, we usually need to sort lists of numbers, characters from an alphabet, character strings, and, most important, records similar to those maintained by schools about their students, libraries about their holdings, and companies about their employees.
- In the case of records, we need to choose a piece of information to guide sorting. For example, we can choose to sort student records in alphabetical order of names or by student number or by student grade-point average. Such a specially chosen piece of information is called a ***key***.
- Why would we want a sorted list? To begin with, a sorted list can be a required output of a task such as ranking Internet search results or ranking students by their GPAscores.
- By now, computer scientists have discovered dozens of different sorting algorithms.
- Two properties of sorting algorithms deserve special mention.
    - A sorting algorithm is called ***stable*** if it preserves the relative order of any two equal elements in its input. In other words, if an input list contains two equal elements in positions *i* and *j* where *i* < *j*, then in the sorted list they have to be in positions *I* and *j*, respectively, such that i < *j*.
    - The second notable feature of a sorting algorithm is the amount of extra memory the algorithm requires. An algorithm is said to be ***in-place*** if it does not require extra memory, except, possibly, for a few memory units. There are important sorting algorithms that are in-place and those that are not.

Searching

- The ***searching problem*** deals with finding a given value, called a ***search key***, in a given set (or a multiset, which permits several elements to have the same value).
- 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.
- The latter algorithms are of particular importance for real-world applications because they are indispensable for storing and retrieving information from large databases.
- For searching, too, 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. Unlike with sorting algorithms, there is no stability problem, but different issues arise.

T. SATISH KUMAR
Asst. Prof.
Dept. of CSE
RNS Institute of Tech...gy
Bangalore - 560 0...
Mob : ...

String Processing

- A ***string*** is a sequence of characters from an alphabet.
- Strings of particular interest are text strings, which comprise letters, numbers, and special characters; bit strings, which comprise zeros and ones; and gene sequences, which can be modeled by strings of characters from the four-character alphabet {A, C, G, T}.
- It should be pointed out, however, that string-processing algorithms have been important for computer science for a long time in conjunction with computer languages and compiling issues One particular problem—that of searching for a given word in a text—has attracted special attention from researchers. They call it ***string matching***. Several algorithms that exploit the special nature of this type of searching have been invented.

Graph Problems

- One of the oldest and most interesting areas in algorithmics is graph algorithms.
- A ***graph*** can be thought of as a collection of points called vertices, some of which are connected by line segments called edges.
- Graphs are an interesting subject to study, for both theoretical and practical reasons. Graphs can be used for modeling a wide variety of applications, including transportation, communication, social and economic networks, project scheduling, and games. Studying different technical and social aspects of the Internet in particular is one of the active areas of current research involving computer scientists, economists, and social scientists.
- Basic graph algorithms include graph-traversal algorithms (how can one reach all the points in a network?), shortest-path algorithms (what is the best route between two cities?), and topological sorting for graphs with directed edges (is a set of courses with their prerequisites consistent or self-contradictory?).
- Some graph problems are computationally very hard; the most well-known examples are the traveling salesman problem and the graph-coloring problem.
- The ***traveling salesman problem (TSP)*** is the problem of finding the shortest tour through n cities that visits every city exactly once. In addition to obvious applications involving route planning, it arises in such modern applications as circuit board and VLSI chip fabrication, X-ray crystallography, and genetic engineering.
- The ***graph-coloring problem*** seeks to assign the smallest number of colors to the vertices of a graph so that no two adjacent vertices are the same color.
- This problem arises in several applications, such as event scheduling: if the events are represented by vertices that are connected by an edge if and only if the corresponding events cannot be scheduled at the same time, a solution to the graph-coloring problem yields an optimal schedule.

Combinatorial Problems

- From a more abstract perspective, the traveling salesman problem and the graph coloring problem are examples of ***combinatorial problems***.
- These are problems that ask, explicitly or implicitly, to find a combinatorial object—such as a permutation, a combination, or a subset—that satisfies certain constraints.
- A desired combinatorial object may also be required to have some additional property such as a maximum value or a minimum cost. Generally speaking, combinatorial problems are the most difficult problems in computing, from both a theoretical and practical standpoint.

- Their difficulty stems from the following facts. First, the number of combinatorial objects typically grows extremely fast with a problem's size, reaching unimaginable magnitudes even for moderate-sized instances. Second, there are no known algorithms for solving most such problems exactly in an acceptable amount of time. Moreover, most computer scientists believe that such algorithms do not exist.
- This conjecture has been neither proved nor disproved, and it remains the most important unresolved issue in theoretical computer science.

Geometric Problems

- ***Geometric algorithms*** deal with geometric objects such as points, lines, and polygons.
- The ancient Greeks were very much interested in developing procedures (they did not call them algorithms, of course) for solving a variety of geometric problems, including problems of constructing simple geometric shapes—triangles, circles, and so on—with an unmarked ruler and a compass.
- Then, for about 2000 years, intense interest in geometric algorithms disappeared, to be resurrected in the age of computers—no more rulers and compasses, just bits, bytes, and good old human ingenuity. Of course, today people are interested in geometric algorithms with quite different applications in mind, such as computer graphics, robotics, and tomography.
- The ***closest-pair problem*** is self-explanatory: given n points in the plane, find the closest pair among them.
- The ***convex-hull problem*** asks to find the smallest convex polygon that would include all the points of a given set. If you are interested in other geometric algorithms.

Numerical Problems

- ***Numerical problems***, another large special area of applications, are problems that involve mathematical objects of continuous nature: solving equations and systems of equations, computing definite integrals, evaluating functions, and so on.
- The majority of such mathematical problems can be solved only approximately. Another principal difficulty stems from the fact that such problems typically require manipulating real numbers, which can be represented in a computer only approximately.
- Moreover, a large number of arithmetic operations performed on approximately represented numbers can lead to an accumulation of the round-off error to a point where it can drastically distort an output produced by a seemingly sound algorithm.
- Many sophisticated algorithms have been developed over the years in this area, and they continue to play a critical role in many scientific and engineering applications.
- But in the last 30 years or so, the computing industry has shifted its focus to business applications. These new applications require primarily algorithms for information storage, retrieval, transportation through networks, and presentation to users.

Graph Representations

- Graphs for computer algorithms are usually represented in one of two ways: the adjacency matrix and adjacency lists.
- The ***adjacency matrix*** of a graph with n vertices is an $n \times n$ 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.

- The ***adjacency lists*** of a graph or a digraph is a collection of linked lists, one for each vertex, that contain all the vertices adjacent to the list's vertex (i.e., all the vertices connected to it by an edge).

Weighted Graphs A ***weighted graph*** (or weighted digraph) is a graph (or digraph) with numbers assigned to its edges. These numbers are called ***weights*** or ***costs***.

Paths and Cycles Among the many properties of graphs, two are important for a great number of applications: ***connectivity*** and ***acyclicity***. Both are based on the notion of a path.A***path*** from vertex u to vertex v of a graph G can be defined as a sequence of adjacent (connected by an edge) vertices that starts with u and ends with v. If all vertices of a path are distinct, the path is said to be ***simple***. The ***length***
of a path is the total number of vertices in the vertex sequence defining the path minus 1, which is the same as the number of edges in the path.

A ***directed path*** 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.
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. If we make a model of a connected graph by connecting some balls representing the graph's vertices with strings representing the edges, it will be a single piece. If a graph is not connected, such a model will consist of several connected pieces that are called connected components of the graph.
Formally, a ***connected component*** is a maximal (not expandable by including another vertex and an edge) connected subgraph2 of a given graph.

Trees
A ***tree*** (more accurately, a ***free tree***) is a connected acyclic graph. A graph that has no cycles but is not necessarily connected is called a ***forest***: each of its connected components is a tree.
Trees have several important properties other graphs do not have. In particular, the number of edges in a tree is always one less than the number of its vertices: $|E| = |V| - 1$.

Rooted Trees Another very important property of trees is the fact that for every two vertices in a tree, there always exists exactly one simple path from one of these vertices to the other. This property makes it possible to select an arbitrary vertex in a free tree and consider it as the ***root*** of the so-called ***rooted tree***.

As examples illustrating the notion of the algorithm, we consider in this section three methods for solving the same problem: computing the greatest common divisor of two integers. These examples will help us to illustrate several important points:

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

Euclid's algorithm for computing gcd*(m, n)*
Step 1 If $n = 0$, return the value of m as the answer and stop; otherwise, proceed to Step 2.
Step 2 Divide m by n and assign the value of the remainder to r.
Step 3 Assign the value of n to m and the value of r to n. Go to Step 1.

Alternatively, we can express the same algorithm in pseudocode:
ALGORITHM *Euclid(m, n)*

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

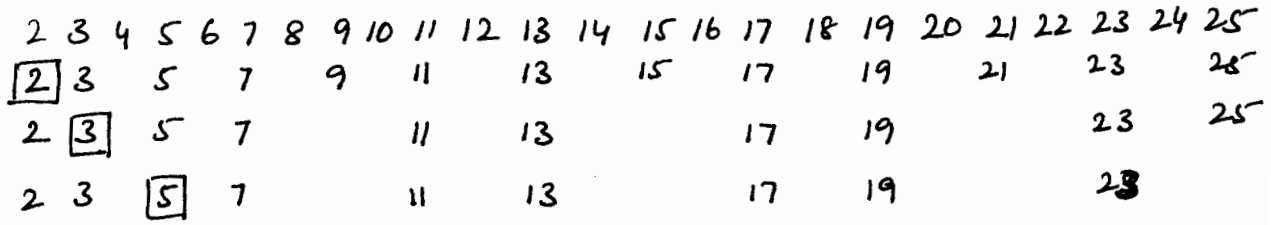
Consecutive integer checking algorithm for computing gcd*(m, n)*
Step 1 Assign the value of min{*m, n*} to *t*.
Step 2 Divide m by *t*. If the remainder of this division is 0, go to Step 3; otherwise, go to Step 4.
Step 3 Divide n by *t*. If the remainder of this division is 0, return the value of *t* as the answer and stop; otherwise, proceed to Step 4.
Step 4 Decrease the value of *t* by 1. Go to Step 2.
Note that unlike Euclid's algorithm, this algorithm, in the form presented, does not work correctly when one of its input numbers is zero. This example illustrates why it is so important to specify the set of an algorithm's inputs explicitly and carefully.

The third procedure for finding the greatest common divisor should be familiar to you from middle school.
Middle-school procedure for computing gcd*(m, n)*
Step 1 Find the prime factors of m.
Step 2 Find the prime factors of n.
Step 3 Identify all the common factors in the two prime expansions found in Step 1 and Step 2. (If p is a common factor occurring pm and pn times in m and n, respectively, it should be repeated min{pm, pn} times.)
Step 4 Compute the product of all the common factors and return it as the greatest common divisor of the numbers given.

Thus, for the numbers 60 and 24, we get
$60 = 2 \cdot 2 \cdot 3 \cdot 5$
$24 = 2 \cdot 2 \cdot 2 \cdot 3$
$\gcd(60, 24) = 2 \cdot 2 \cdot 3 = 12$.

a simple algorithm for generating consecutive primes not exceeding any given integer $n > 1$. It was probably invented in ancient Greece and is known as the ***sieve of Eratosthenes*** (ca. 200 b.c.). The algorithm starts by initializing a list of prime candidates with consecutive integers from 2 to n. Then, on its first iteration, the algorithm eliminates from the list all multiples of 2, i.e., 4, 6, and so on. Then it moves to the next item on the list, which is 3, and eliminates its multiples. (In this straightforward version, there is an overhead because some numbers, such as 6, are eliminated more than once.) No pass for number 4 is needed: since 4 itself and all its multiples are also multiples of 2, they were already eliminated on a previous pass. The next remaining number on the list, which is used on the third pass, is 5.

As an example, consider the application of the algorithm to finding the list of primes not exceeding $n = 25$:



ALGORITHM *Sieve(n)*
//Implements the sieve of Eratosthenes
//Input: A positive integer $n > 1$
//Output: Array L of all prime numbers less than or equal to n

```
for p←2 to n do A[p]←p
for p←2 to ⌊√n⌋ do //see note before pseudocode
        if A[p] ≠ 0 //p hasn't been eliminated on previous passes
                j ← p □ p
                while j ≤ n do
                        A[j ]←0 //mark element as eliminated
                        j ←j + p
                //copy the remaining elements of A to array L of the primes
i ←0
for p←2 to n do
        if A[p] ≠ 0
                L[i]←A[p]
                i ←i + 1
return L
```

Brute Force String Matching

- String Matching is a problem of detecting the occurence of a particular substring called "pattern" in another string called the "text"

```
Algorithm BruteForceStringMatch(T[0..n-1], P[0..m-1])
// Imp Brute force string Matching
// I/P: An array T[0..n-1] of n characters
   representing a text & an array P[0..m-1]
   representing a pattern
// O/P: Position of the first character in the text
        if search successful or -1 otherwise

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

Efficiency.

* If the pattern appears at the beginning of the text, m comparisions are done

  ie $T_{best}(n) = O(m)$ <Best Case>

* The worst case occurs when the outer loop is executed $n-m+1$ times. ie the algorithm shifts the pattern almost always after a single two character comparison. Therefore we have to make 'm' comparisons before shifting the pattern.

$$\text{ie } T_{worst}(n) = \sum_{i=0}^{n-m} \sum_{j=1}^{m} 1$$

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

$$= mn - m^2 + m$$

$$\text{ie } \boxed{T_{worst}(n) = O(mn)}$$

Exhaustive Search.

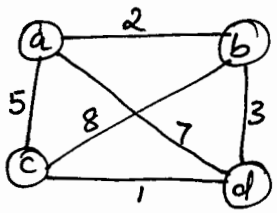
* Exhaustive search is simply a Brute Force approach to Combinational problem. It suggests generating each & every element of the problem domain, selecting those of them that satisfy the problem Constraint & then finding a desired Element.

Travelling Salesman Problem (TSP)

* In TSP, a salesman must visit n cities
* If the salesman wishes to make a tour, visiting each city exactly once & finishing at the same city from where he has started

* There will be an integer cost c(i,j) to travel from i to j.
* The salesman should complete the tour in such a way that the total cost spent should be minimum where total cost is the sum of the individual costs along the edges (paths) of the tour.

Example



If there are n vertices, there will be (n-1)! combinations of possible tours
Here (4-1)! = 3! = 6 tours.

Tours	Costs	
① a → b → c → d → a	2+8+1+7 = 18	Feasible Solutions
② a → b → d → c → a	2+3+1+5 = 11	
③ a → c → b → d → a	5+8+3+7 = 23	
④ a → c → d → b → a	5+1+3+2 = 11	
⑤ a → d → b → c → a	7+3+8+5 = 23	
⑥ a → d → c → b → a	7+1+8+2 = 18	

* out of 6 possible tours, the 2nd & 4th are optimal with a minimum cost of 11 units
* Therefore, the possible optimal solutions paths are

a → b → d → c → a & a → c → d → b → a

T. SATISH KUMAR
Asst. Prof.
Dept. of CSE
RNS Institute of Technology
Bangalore - 560 0[illegible].
Mob : [illegible]

Knapsack Problem

* In knapsack problem, we will be given n objects & a knapsack (bag/sack)
* Each object has a weight w_i & will also be associated with a profit P_i that will be earned when particular object is placed in a knapsack.
* we have an option of placing an object as a whole or as a part if fraction is not allowed, then it will be called as 0/1 knapsack. 0 indicates that the object is not added on to the knapsack & 1 indicates that it is added on to the knapsack.
* The Exhausline search approach to this problem leads to considering all the subsets of set n items given, Computing the total weight of the each subset in order to identify the feasible solutions.

Example: $n=4$, $w=10$

$\{w_1, w_2, w_3, w_4\} = \{7, 3, 4, 5\}$

$\{P_1, P_2, P_3, P_4\} = \{42, 12, 40, 25\}$

If there are n objects than there will be $2^n - 1$ Subsets.

SLNO:	Subset	Total weight	Profit	
1	{1}	7	42	Feasible Soln
2	{2}	3	12	
3	{3}	4	40	
4	{4}	5	25	
5	{1,2}	10	54	
6	{1,3}	11	82	
7	{1,4}	12	67	
8	{2,3}	7	52	Feasible Soln
9	{2,4}	8	37	
10	{3,4}	9	65	Optimal soln.
11	{1,2,3}	14	94	
12	{1,2,4}	15	79	
13	{1,3,4}	16	107	
14	{2,3,4}	12	77	
15	{1,2,3,4}	19	119	

Assignment Problem

* In assignment problem, there will be n Persons $(P_1, P_2, \ldots P_n)$ who need to be assigned to execute n jobs $(J_1, J_2, \ldots J_n)$
* But the constraint here is each person should be assigned only one job
* The cost of execeuting j^{th} job by i^{th} person is denoted by $c(i,j)$ i-person j-job.
* The problem is to find an assignment with the smallest total cost, which means that assigning all the persons to all the jobs optimally. ie total cost should be minimum.

Example : Consider the following cost-metrix of the assignment problem.

	J_1	J_2	J_3
P_1	5	3	2
P_2	8	7	5
P_3	4	9	12

(·X·) we can obtain n! assignments here 3! = 6 assignments.

SL NO	P_1 P_2 P_3	Cost
1	⟨1,2,3⟩	5+7+12 = 24
2	⟨1,3,2⟩	5+5+9 = 19
3	⟨2,1,3⟩	3+8+12 = 23
4	⟨2,3,1⟩	3+5+4 = (12) optimal soln.
5	⟨3,1,2⟩	2+8+9 = 19
6	⟨3,2,1⟩	2+7+4 = 13

The optimal soln is with a cost of 12 where
I person is assigned 2nd job
II person is assigned 3rd job
III person is assigned 1st job.

Brute Force

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

* "Just do it" would be the another way to describe the prescription of the brute force approach.
* This technique is used only when the size of the problems instances is less. ie to sort 100 elements brute force technique is not opted.
* Disadvantage :→ more expensive with respect to time & space.
  → not so efficient & effective compared to other designing techniques
* Advantage :→ Simple Design.

Ex: a^n is calculated using Brute Force as follows

$$a^n = \underbrace{a * a * a * \dots * a}_{n \text{ times}}$$

Selection Sort

* We start selection sort by scanning the entire given list to find its smallest element & exchange it with the first element putting the smallest element in its final position in the sorted list.
* Then we scan the list starting from the second element to find the smallest among last n-1 elements & exchange it with second element, putting the second smallest element in its final position.

* Generally, on the i^{th} 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 & swaps it with A_i :

$A_0 \le A_1 \le \cdots < A_{i-1} \mid A_i, \ldots A_{min} \cdots A_{n-1}$

in their final position — the last n-i element

* After n-1 passes, the list is sorted.

Example.

89	45	68	90	29	34	(17)	
17 |	45	68	90	(29)	34	89	Pass-0
17	29 |	68	90	45	(34)	89	pass-1
17	29	34 |	90	(45)	68	89	pass-2
17	29	34	45 |	90	(68)	89	pass-3
17	29	34	45	68 |	90	(89)	pass-4
17	29	34	45	68	89 |	90	pass-5

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

// Sorts a given array by SelectionSort
// 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
    min ← i
    for j ← i+1 to n-1 do
        if A[j] < A[min]
            min ← j
    swap A[i] and A[min]
```

Analysis

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

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

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

$$C(n) = (n-1) + (n-2) + \cdots + 1$$

$$= 1 + 2 + 3 \cdots + n-1$$

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

$$= \frac{n^2}{2} - \frac{n}{2} = n^2 - n$$

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

Bubble Sort [Comparison of adjacent Elements]

* In Bubble sort, we Compare adjacent Elements of the list & Exchange them if they are out of order.
* By repeadely doing it, we end up "Bubbling up" the largest element to the last position of the list.
* In the next iteration, the second largest element will be bubbled up & so on until, all the elements have been sorted.

Example 5 4 3 2 1

(5	4	4	4	4	4	3	3	3	3	2	2	2	1						
4	(5	3	3	3	3	4	2	2	2	3	1	1	2						
3	3	(5	2	2	2	2	4	1	1	1	3	3	3						
2	2	2	(5	1	1	1	1	4	4	4	4	4	4						
1	1	1	1	5	5	5	5	5	5	5	5	5	5						
pass-1					pass-2				pass-3				pass-4						

ie no: of passes = no: of Elements - 1.

Algorithm Bubble Sort (A[0..n-1])
// Sorts the elements of array A using bubble sort
// I/P: An unsorted array A[0..n-1]
// O/P: An Sorted array A[0..n-1]

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

Efficiency.

-> Efficiency depends on how many times Comparisons has been done

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

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

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

$$C(n) = (n-1) + (n-2) + \cdots + 1$$

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

$$\Rightarrow \boxed{\therefore C(n) = O(n^2)}$$

Sequential Search & Brute Force String Matching

Sequential Search

* This 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).
* While implementing Sequential Search, the search key is appended to the end of the list. Thus, the search for the key will have to be successful, & therefore we can eliminate a check for the list's end on each iteration of the algorithm.

Algorithm SequentialSearch(A[0..n], K)
// Implements Sequential Search with Search key as a Sentinel
// I/P: An array 'A' of n elements & a key.
// O/P: The index of the first element in A[0..n-1] whose value is Equal to K or -1 if no such Element is found

```
A[n] ← K
i ← 0
while A[i] ≠ K do
    i ← i+1
if i < n return i
Else return -1
```

Theorem (Master Theorem):
If $f(n) \in \Theta(n^d)$ with $d \geq 0$ in recurrence Equation ($T(n) = aT(n/b) + f(n)$), then

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

(Analogous results holds for the O & Ω notations too).

Example 1: To use the master theorem, we simply determine which case (if any) of the master theorem applies & write down the answer
Consider the recurrence: $T(n) = 9T(n/3) + n$.
For this recurrence, we have $a=9$, $b=3$, $f(n)=n$
& thus we have that

$$n^{\log_b a} = n^{\log_3 9} = n^{\log_3 3^2} = \Theta(n^2).$$

Ex 2: Consider the following recurrence

$$T(n) = T(2n/3) + 1,$$

in which $a=1$, $b=3/2$, $f(n)=1$ & $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$

Case 2 applies, since $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$
& thus the solution to the recurrence is given by

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a} \log_2 n) \\ &= \Theta(1 \times \log_2 n) \\ &= \Theta(\log_2 n) \end{aligned}$$

Recurrence	solution.
$T(n) = T(n/2) + d$	$T(n) = O(\log_2 n)$
$T(n) = T(n/2) + n$	$T(n) = O(n)$
$T(n) = 2T(n/2) + d$	$T(n) = O(n)$
$T(n) = 2T(n/2) + n$	$T(n) = O(n \log_2 n)$
$T(n) = T(n-1) + d$	$T(n) = O(n)$

Fibonacci Numbers

0, 1, 1, 2, 3, 5, 8, 13, ... can be defined by the simple recurrence

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

& two initial conditions

$$F(0) = 0, \quad F(1) = 1$$

Explicit formula for the n^{th} Fibonnaci number

* Here we donot apply back substitution to solve recurrence.
* We shall take advantage of a theorem that describes solution to a homogeneous second-order linear recurrence with constant co-efficients

$$ax(n) + bx(n-1) + cx(n-2) = 0 \quad \text{---} \quad (1)$$

* where a, b & c are some fixed real no: ($a \neq 0$) called the coefficients of recurrence.
* $x(n)$ is an unknown sequence to be found

$$ar^2 + br + c = 0 \quad \text{---} \quad (2)$$

is called the characteristic Equation for recurrence

* Let us apply this theorem to the case of the fibonacci no: The recurrence is rewritten as

$$F(n) - F(n-1) - F(n-2) = 0 \quad \text{---} \quad (3)$$

* Its characteristic Equation is

$$r^2 - r - 1 = 0 \quad \text{---} \quad (4)$$

* with roots

$$r_{1,2} = \frac{1 \pm \sqrt{1 - 4(-1)}}{2} = \frac{1 \pm \sqrt{5}}{2}$$

* Since this characteristic Equation has two distinct real roots, we can rewrite it as

$$F(n) = \alpha\left[\frac{1+\sqrt{5}}{2}\right]^n + \beta\left[\frac{1-\sqrt{5}}{2}\right]^n$$

* we shall take the advantage of initial condition to find specific values of parameters α & β

$$F(0) = \alpha\left[\frac{1+\sqrt{5}}{2}\right]^0 + \beta\left[\frac{1-\sqrt{5}}{2}\right]^0 = 0$$

$$F(1) = \alpha\left[\frac{1+\sqrt{5}}{2}\right]^1 + \beta\left[\frac{1-\sqrt{5}}{2}\right]^1 = 1$$

* After some standard algebraic simplification we get the following system of 2 linear Equations in two unknown α & β

$$\alpha + \beta = 0$$

$$\alpha\left[\frac{1+\sqrt{5}}{2}\right] + \beta\left[\frac{1-\sqrt{5}}{2}\right] = 1$$

* solving the system ($\beta = 1-\alpha$), we get the value $\alpha = 1/\sqrt{5}$ & $\beta = -1/\sqrt{5}$ for the unknowns

Thus

$$F(n) = \frac{1}{\sqrt{5}}\left[\frac{1+\sqrt{5}}{2}\right]^n - \frac{1}{\sqrt{5}}\left[\frac{1-\sqrt{5}}{2}\right]^n = \frac{1}{\sqrt{5}}\left[\phi^n - \hat{\phi}^n\right]$$

where $\phi = (1+\sqrt{5})/2 \approx 1.61803$

$\hat{\phi} = (1-\sqrt{5})/2 \approx -0.61803$

* for every non negative integer n

$F(n) = \frac{1}{\sqrt{5}}\phi^n$ rounded to the nearest integer.

UNIT-2

Divide and Conquer

Divide & Conquer: General method, Defective chess Board, Binary Search, Merge Sort, Quick Sort and its performance.

[Ref: Fundamentals of Computer Algorithms by Horowitz, Sahni & Rajasekaran]

* **Divide & Conquer strategy divides the problem into subproblems.**
* These subproblems must be solved & then a method must be found to Combine sub-solution of the whole.
* If the subproblems are still relatively large, then the divide-&-conquer strategy can possibly be reapplied.

Algo: Control abstraction for divide & Conquer.

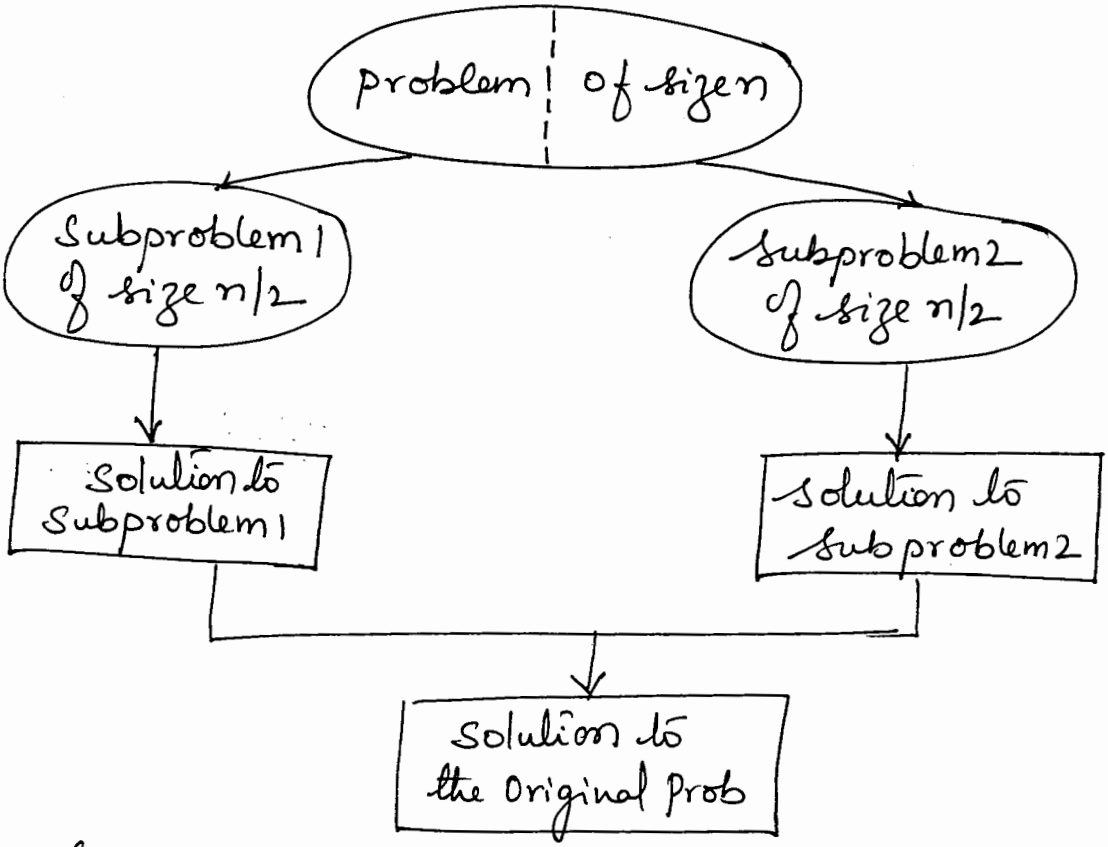
```
Algorithm DAandC (P)
{
    if Small(P) then return S(P);
    else
    {
        divide P into smaller instances
                P1, P2, .... Pk, k ≥ 1;
        Apply DAandC to each of these subproblems.
        return Combine (DAandC (P1), DAandC(P2) ...
                        DAandC (Pk));
    }
}
```

* By a control abstraction we mean a procedure whose flow of control is clear but whose primary operations are specified by other

procedures whose precise meanings are left undefined.

* DAandC is initially invoked as DAandC(P), where P is the problem to be solved.
* Small(P) is a Boolean function that determines whether the i/p size is small enough that the answer can be computed without splitting.

Divide-&-Conquer technique (typical case)



* DAC (Divide & Conquer) algo work according to the following general plan.

1. A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size.
2. The smaller instances are solved.
3. The necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance.

```
Algorithm BinSearch (a, i, l, x)
// Given an array a[i:l] of Elements in Ascending
// order 1≤i≤l, determine whether x is present,
// and if so, return j such that x = a[j]; else return 0.
{
    if (l = i) then  // if Small(P)
    {
        if (x = a[i]) then return i;
        Else return 0;
    }
    else
    { // Reduce P into a small subproblem
        mid := ⌊(i+l)/2⌋;
        if (x = a[mid]) then return mid;
        Else if (x < a[mid]) then
                return BinSearch(a, i, mid-1, x);
            Else
                return BinSearch(a, i, mid+1, x);
    }
}
```

Iterative Binary Search.

```
Algorithm BinSearch (a, n, x)
// Given an array a[1:n] of elements in Ascending
// order. n≥0, determine whether x is present
// & if so, return j such that x = a[j]; else return 0.
{
    low := 1; high := n;
    while (low ≤ high) do
    {
        mid := ⌊(low + high)/2⌋;
        if (x < a[mid]) then high := mid-1;
        Else if (x > a[mid]) then low := mid+1;
            else return mid;
    }
    return 0;
}
```

Example:

-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151

$x = 151$

low	high	mid
1	14	7
8	14	11
12	14	13
14	14	14

found ✓

$x = -14$

low	high	mid
1	14	7
1	6	3
1	2	1
2	2	2
2	1	not find ✓

$x = 9$

low	high	mid
1	14	7
1	6	3
4	6	5

found.

Algorithm Efficiency / Time Complexity

Successful Searches			Unsuccessful Searches
$O(1)$,	$O(\log n)$,	$O(\log n)$	$O(\log n)$
best	Average	worst	best, Average, worst

Binary Search Analysis

Base case: Time Complexity = $\Omega(1)$

Worst Case:

Worst case occurs when key to be searched is either at the first position or at the last Position.

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

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

Substituting for $n = 2^k$

$$T(2^k) = T(2^{k-1}) + 1$$
$$= T(2^{k-2}) + 1 + 1$$
$$= T(2^{k-3}) + 3$$
$$\vdots$$
$$= T(2^{k-k}) + k \qquad 2^k = n$$
$$= T(1) + k \qquad k = \log_2 n$$
$$= \log_2 n$$

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

T. SATISH KUMAR
Asst. Prof.
Dept. of CSE
RNS Institute

Average Case

Let n be the no. of elements in the array & let no: of elements in the array be integral power of 2.

Situation S_1, the no: of Comparison = 1 for 2^0

—"— S_2, —"— —"— = 2 for 2^1

S_c, —"— —"— = c for 2^{c-1} item

$$\therefore T(n) = \sum_{c=1}^{k} \frac{S_c}{n} * c$$

$$= \frac{1}{n} \sum_{c=1}^{k} c\,(2^c - 2^{c-1})$$

$$= 1/n \left\{ \sum_{c=1}^{k} c\,2^c - \sum_{c=1}^{k} c\,2^{c-1} \right\}$$

$$= 1/n \left\{ \sum_{c=1}^{k} c\,2^c - \sum_{c=0}^{k-1} c\,2^c - \sum_{c=0}^{k-1} 2^c \right\}$$

$$= \frac{1}{n} \left\{ k\,2^k - \sum_{c=0}^{k-1} 2^c \right\} = \frac{1}{n} \left\{ k\,2^k - [1 + 2^1 + 2^2 + \cdots 2^{k-1}] \right\}$$

$$= 1/n \left\{ k\,2^k - 1 \left[\frac{2^k - 1}{2-1} \right] \right\}$$

$$= 1/n \left\{ k\,n - 1\,(n-1) \right\}$$

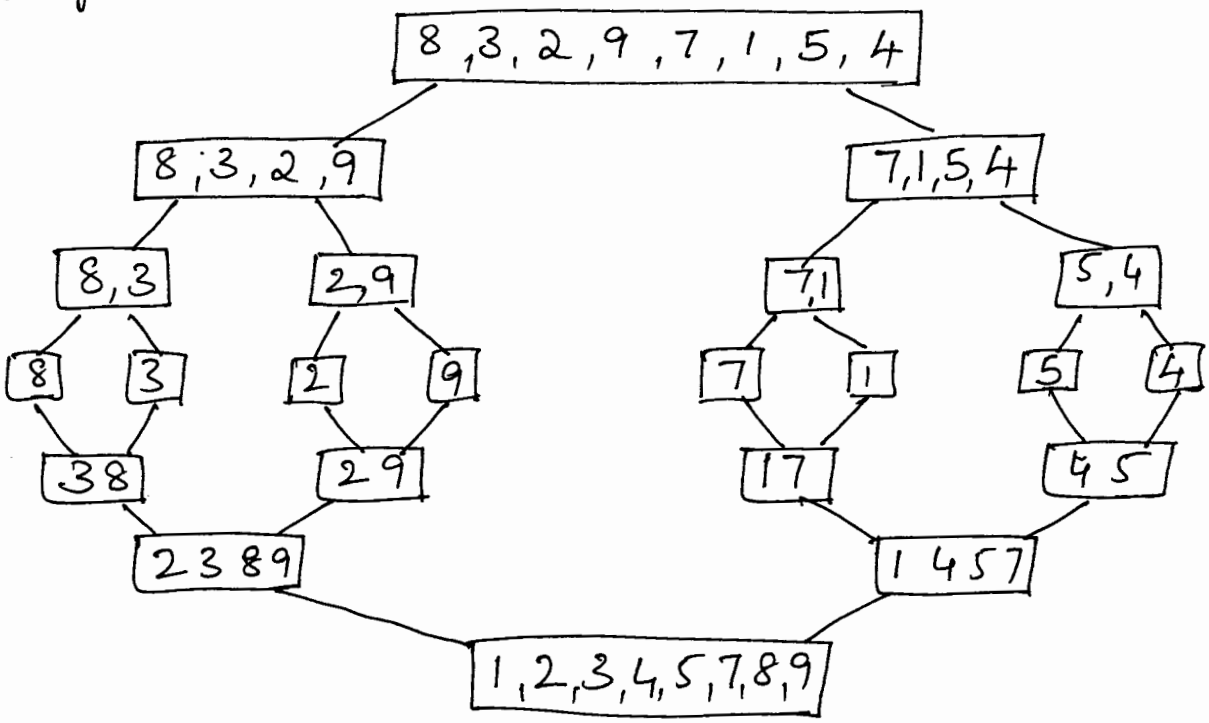
$$= 1/n \left\{ kn - n + 1 \right\}$$

$$= k - 1 + \frac{1}{n}$$

$$= k = \log_2 n$$

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

Merge Sort. Example of Merge sort Operation.



Example 2: 179, 254, 285, 310, 351, 423, 450, 520, 652, 861

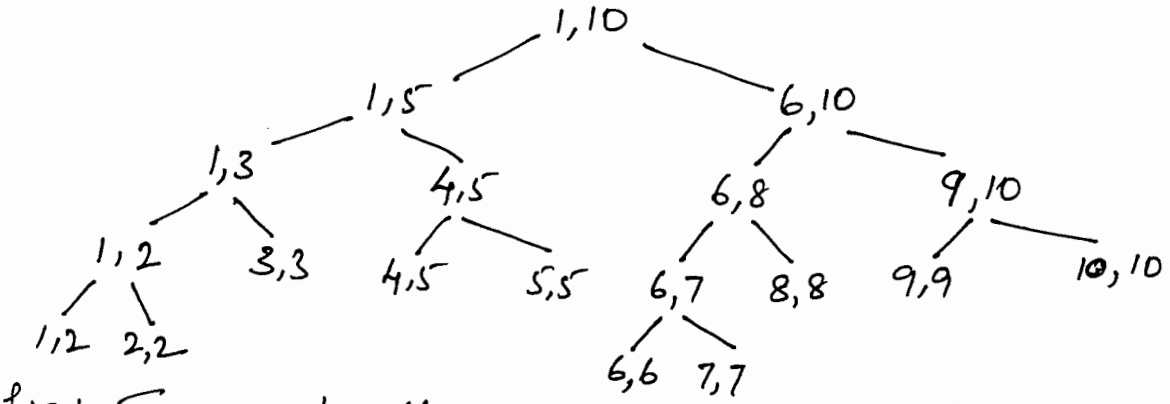


fig: Tree of calls of MergeSort(1,10)

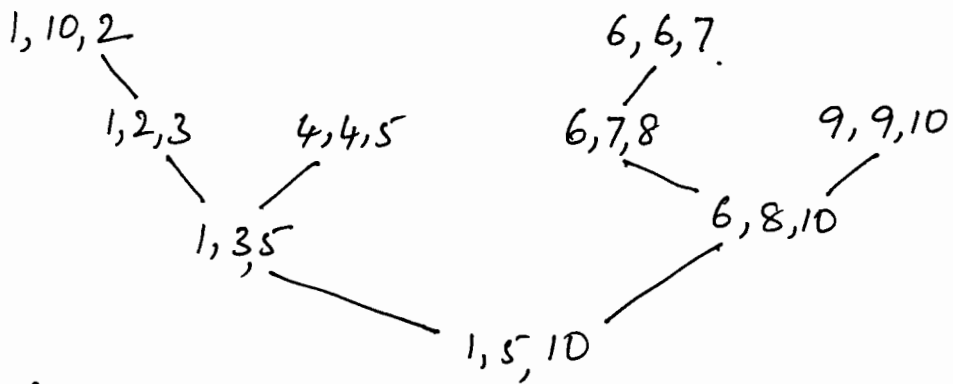


fig: Tree of calls Merge.

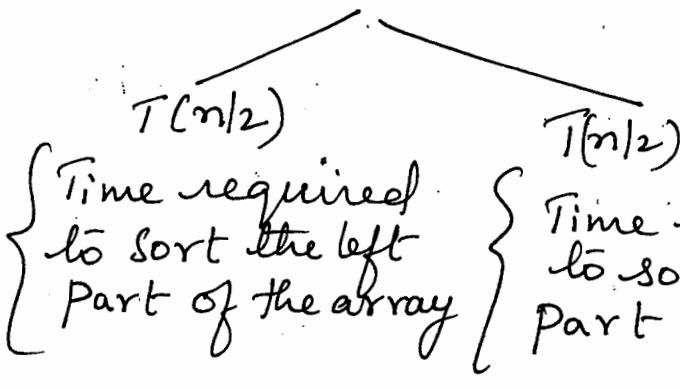
```
Algorithm MergeSort (low, high)
// a [low:high] is a global array to be sorted.
// Small(P) is true if there is only one Element
// to sort. In this case the list is already sorted.
{
    if (low < high) then
    {
        // Divide P into subproblems.
            // Find where to split the set.
                mid := ⌊(low+high)/2⌋;
        // Solve the subproblems
            MergeSort (low, mid);
            MergeSort (mid+1, high);
        // Combine the solutions.
            Merge (low, mid, high);
    }
}
Algorithm Merge (low, mid, high)
// a [low:high] is a global array containing 2 sorted
subsets in a[low:mid] and in a [mid+1:high].
The goal is to merge these two sets into a
single set residing in a [low:high]. b[] is
an auxiliary global array.
{
    h := low; i := low; j := mid+1;
    while ((h ≤ mid) and (j ≤ high)) do
    {
        if (a[h] ≤ a[j]) then
        {
            b[i] := a[h]; h := h+1;
        }
        else
        {
            b[i] := a[j]; j := j+1;
        }
        i := i+1;
    }
```

```
if (h > mid) then
        for k:= j to high do
{       b[i]:= a[k]; i:= i+1;
}
else    for k:= h to mid do
{       b[i]:= a[k]; i:= i+1;
}
for k:= low to high do a[k]:= b[k];
}
```

Recurrence Relation for Merge Sort.

Method1

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



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

$a = 2$, $b = 2$, $f(n) = cn = cn^1 = cn^d$

$\therefore d = 1 \quad (b = b^d)$

by Master's Theorem.

$$T(n) = O(n \log_2 n) = \Omega(n \log_2 n) = \Theta(n \log_2 n)$$

Method2 Time Complexity By using Substitution Method.

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

let $n = 2^k$

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

$$T(2^{k-1}) = 2[2T(2^{k-2}) + c2^{k-1}] + c2^k$$
$$= 2^2 T(2^{k-2}) + 2.c.2^{k-1} + c.2^k$$
$$= 2^3 T(2^{k-2}) + 2c2^k$$
$$= 2^3 T(2^{k-3}) + 3c2^k.$$

$$T(2^k - Ck = 2^k T(2^{k-k}) + k C 2^k$$

$$= 2^k T(2^0) + C k.2^k$$

$$= 2^k (0) + C.k.2^k$$

$$= C.n.k \quad (\because 2^k = n)$$

$$= C.n.\log_2 n \quad (\text{neglect constant})$$

$$= n \log_2 n$$

apply log both sides

$k \log_2 2 = n \log_2 n$

$k = \log_2 n.$

$\therefore O(n \log_2 n) = \Omega(n \log_2 n) = \Theta(n \log_2 n).$

Quick Sort.

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	i	P
65	(70)	75	80	85	60	55	50	(45)	+∞	2	9
65	45	(75)	80	85	60	55	(50)	70	+∞	3	8
65	45	50	(80)	85	60	(55)	75	70	+∞	4	7
65	45	50	55	(85)	(60)	80	75	70	+∞	5	6
(65)	45	50	55	(60)	85	80	75	70	+∞	6	5
60	45	50	55	65	85	80	75	70	+∞		

```
Algorithm Quicksort (P, q)
// Sorts the elements a[p], ..., a[q] which reside
// in the global array a[1:n] into ascending order;
// a[n+1] is considered to be defined & must be
// ≥ all the elements in a[1:n]
{
    if (P < q) then // if there are more than
                    one Element
    {
        // divide P into two subproblems
        j := Partition (a, P, q+1);
            // j is the position of the partitioning
               Element.
        // solve the subproblems.
        Quicksort (P, j-1);
        Quicksort (j+1, q);
        // Therefore is no need for Combining
           solutions.
    }
}
```

```
Algorithm Partition(a, m, p)
// within a[m], a[m+1], ... a[p-1] the elements are
// rearranged in such a manner that if initially
// t = a[m], then after completion a[q] = t for some
// q between m & p-1, a[k] ≤ t for m ≤ k ≤ q
// & a[k] ≥ t for q < k < p. q is returned. Set a[p] = ∞.
{
    v := a[m]; i := m; j := p;
    repeat
    {
        repeat
            i := i+1;
        until (a[i] ≥ v);
        repeat
            j := j-1;
        until (a[j] ≤ v);
        if (i < j) then interchange(a, i, j);
    } until (i ≥ j);
    a[m] := a[j]; a[j] := v; return j;
}

Algorithm Interchange(a, i, j)
// Exchange a[i] with a[j]
{
    p := a[i];
    a[i] := a[j]
    a[j] := p;
}
```

Analysis of Quick Sort.

The Recurrence relation of quicksort is given by.

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$

Time Complexity By Using Master Theorem

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

Recurrence relation of quicksort

$$\therefore T(n) = 2T(n/2) + cn \qquad a=2,\ b=2$$

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

$f(n) = cn = cn^1 = cn^d$

$\therefore d$ is one

$$\therefore \theta(n^d \log_b n) = \theta(n \log_2 n)$$

Useful Property Involving the Asymptotic Notations

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

(Note: The analogous assertions are true for the Ω & Θ)

Proof: → Let a_1, b_1, a_2 & b_2 be four arbitrary real numbers where if $a_1 \le b_1$ & $a_2 \le b_2$, then $a_1 + a_2 \le 2\max\{b_1, b_2\}$

→ Since $t_1(n) \in O(g_1(n))$, there exist some positive constant c_1 & some nonnegative integer n_1 such that $t_1(n) \le c_1 g_1(n)$ for all $n \ge n_1$.

→ Similarly, since $t_2(n) \in O(g_2(n))$,

$$t_2(n) \le c_2 g_2(n) \text{ for all } n \ge n_2.$$

→ Let us denote $c_3 = \max\{c_1, c_2\}$ & consider $n \ge \max\{n_1, n_2\}$ so that we can use both inequalities.

→ Adding the two inequalities above yields the following:

$$t_1(n) + t_2(n) \le c_1 g_1(n) + c_2 g_2(n)$$

$$\le c_3 g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)]$$

$$\le c_3 2\max\{g_1(n), g_2(n)\}.$$

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

Example1 : Compare the orders of growth of $\frac{1}{2}n(n-1)$ and n^2.

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

Since the limit is equal to a positive constant the functions have the same order of growth or symbolically $\frac{1}{2}n(n-1) \in \Theta(n^2)$

Example2 : Compare the orders of growth of $n!$ and 2^n.

Taking the advantage of Stirling's formula we get

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

Thus though 2^n grows very fast, $n!$ grows still faster. We can write symbolically that $n! \in \Omega(2^n)$.

Stirling's formula: $\lim_{n\to\infty} \frac{n!}{n^{n+1/2}e^{-n}} = \sqrt{2\pi}.$

The following asserlions are all true:

⟨1⟩ $n \in O(n^2)$ ⟨2⟩ $100n+5 \in O(n^2)$ ⟨3⟩ $\frac{1}{2}n(n-1) \in O(n^2)$

⟨4⟩ $n^3 \notin O(n^2)$ ⟨5⟩ $0.00001n^3 \notin O(n^2)$ ⟨6⟩ $n^4+n+1 \notin O(n^2)$

⟨7⟩ $n^3 \in \Omega(n^2)$ ⟨8⟩ $\frac{1}{2}n(n-1) \in \Omega(n^2)$ ⟨9⟩ $100n+5 \notin \Omega(n^2)$.

Examples for Big "oh"

⟨1⟩ The function $3n+2 = O(n)$ as $3n+2 \leq 4n$ for all $n \geq 2$

⟨2⟩ $3n+3 = O(n)$ as $3n+3 \leq 4n$ for all $n \geq 3$.

⟨3⟩ $10n^2+4n+2 = O(n^4)$ as $10n^2+4n+2 \leq 10n^4$ for $n \geq 2$.

⟨4⟩ $3n+2 \neq O(1)$ as $3n+2$ is not less than or Equal to c for any constant c & all $n \geq n_0$.

Examples for Big "Omega".

⟨1⟩ The function $3n+2 = \Omega(n)$ as $3n+2 \geq 3n$ for $n \geq 1$

⟨2⟩ $3n+3 = \Omega(n)$ as $3n+3 \geq 3n$ for $n \geq 1$.

⟨3⟩ $6*2^n+n^2 = \Omega(2^n)$ as $6*2^n+n^2 \geq 2^n$ for $n \geq 1$.

Examples for Big "Theta"

⟨1⟩ The function $3n+2 = \Theta(n)$ as $3n+2 \geq 3n$ for all $n \geq 2$, & $3n+2 \leq 4n$ for all $n \geq 2$, so $c_1=3$, $c_2=4$ & $n_0=2$.

⟨2⟩ $3n+2 \neq \Theta(1)$

⟨3⟩ $3n+3 \neq \Theta(n^2)$

⟨4⟩ $10n^2+4n+2 \neq \Theta(n)$

⟨5⟩ $6*2^n+n^2 \neq \Theta(n^{100})$

⟨6⟩ $6*2^n+n^2 \neq \Theta(1)$.

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

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

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

$$= (n-1) \sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i \qquad \left| \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \right.$$

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

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

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

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

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

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

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

$$= \frac{n^2-n}{2}$$

$\therefore C(n) \in \theta(n^2)$

Quicksort [Average Case]

→ under the assumption, the partitioning element v has an equal probability of being the i^{th} smallest element, $1 \le i \le p-m$ in $a[m:p-1]$

→ Hence the two subarrays remaining to be sorted are $a[m:j]$ and $a[j+1:p-1]$ with probability $1/(p-m)$, $m \le j < p$

→ From this we obtain the recurrence

$$C_A(n) = n+1 + \frac{1}{n} \sum_{1 \le k \le n} [C_A(k-1) + C_A(n-k)] \quad \text{— (1)}$$

→ The no: of elements comparisons required by partition on its first call is $n+1$.
also note that $C_A(0) = C_A(1) = 0$

→ xly both sides of (1) by n, we obtain

$$nC_A(n) = n(n+1) + 2[C_A(0) + C_A(1) + \cdots + C_A(n-1)] \quad \text{— (2)}$$

Replacing n by $n-1$ in (2) gives

$$(n-1)C_A(n-1) = n(n-1) + 2[C_A(0) + \cdots C_A(n-2)]$$

Subtracting this from (2) we get

$$nC_A(n) - (n-1)C_A(n-1) = 2n + 2C_A(n-1)$$

or

$$C_A(n)/(n+1) = C_A(n-1)/n + 2/(n+1)$$

→ Repeatedly using this Equation to substitute for $C_A(n-1), C_A(n-2), \ldots$ we get.

SCHEME AND SOLUTION

SUBJECT CODE: **SUBJECT:**

Question No		Marks

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

$$= \frac{C_A(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

$$\vdots$$

$$= \frac{C_A(1)}{2} + 2 \sum_{3 \le k \le n+1} \frac{1}{k} \quad \text{— (3)}$$

$$= 2 \sum_{3 \le k \le n+1} 1/k.$$

Since

$$\sum_{3 \le k \le n+1} 1/k \le \int_2^{n+1} \frac{1}{x}\, dx = \log_e(n+1) - \log_e 2$$

(3) yields.

$$C_A(n) \le 2(n+1)\left[\log_e(n+2) - \log_e 2\right] = \frac{\cancel{O(n \log n)}}{O(n \log n)}$$

Quick sort (Best Case Analysis)

Recurrence Equation for Best case is given as

$$T(n) = \begin{cases} 0 & \\ T(n/2) + T(n/2) + n & \text{otherwise} \end{cases}$$

Time required to sort left part of the array | Time required to sort right part of the array | Time required to Partition n elements

Applying Master's Theorem to the recurrence

$a = 2, \; b^d = 2^1 \quad \therefore T(n) = \Theta(n \log_2 n)$

Quick Sort [Worst Case Analysis]

we have $k=1$ & $n-k=n-1$ (one Element in an array & $n-1$ elements in the other). In such Case the Recurrence Equation is given by

$$T(n) = T(1) + T(n-1) + \alpha n$$

Solving the recurrence as follows

$$T(n) = T(n-1) + T(1) + \alpha n$$

$$= [T(n-2) + T(1) + \alpha(n-1)] + T(1) + \alpha n$$

[note: $T(n-1) = T(1) + T(n-2) + \alpha(n-1)$ by simplification]

$$= T(n-2) + 2T(1) - \alpha(n-1+n)$$

$$= T(n-3) + T(1) + \alpha(n-2) + 2T(1) + \alpha(n-1+n)$$

$$= T(n-3) + 3T(1) + \alpha(n-2+n-1+n)$$

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

$$\vdots \qquad \vdots \qquad \vdots$$

$$= T(n-i) + iT(1) + \alpha(n-i+1+\cdots+n-2+n-1+n)$$

$$= T(n-i) + iT(1) + \alpha\left[\sum_{j=0}^{i-1}(n-j)\right]$$

→ Now clearly such a recurrence can only go on until $i=n-1$ (why? becoz otherwise $n-i$ would be less then 1).

→ So substitute $i=n-1$ in the above Equation which gives us

$$T(n) = T(n-n-1) + (n-1)T(1) + \alpha\sum_{j=0}^{n-1-1}(n-j)$$

$$T(n) = nT(1) + \alpha\left(n(n-2) - (n-2)(n-1)/2\right)$$

Notice $\sum_{j=0}^{n-2} j = \sum_{j=1}^{n-2} j = (n-2)(n-1)/2$

which is $\boxed{O(n^2)}$

Defective chessboard.

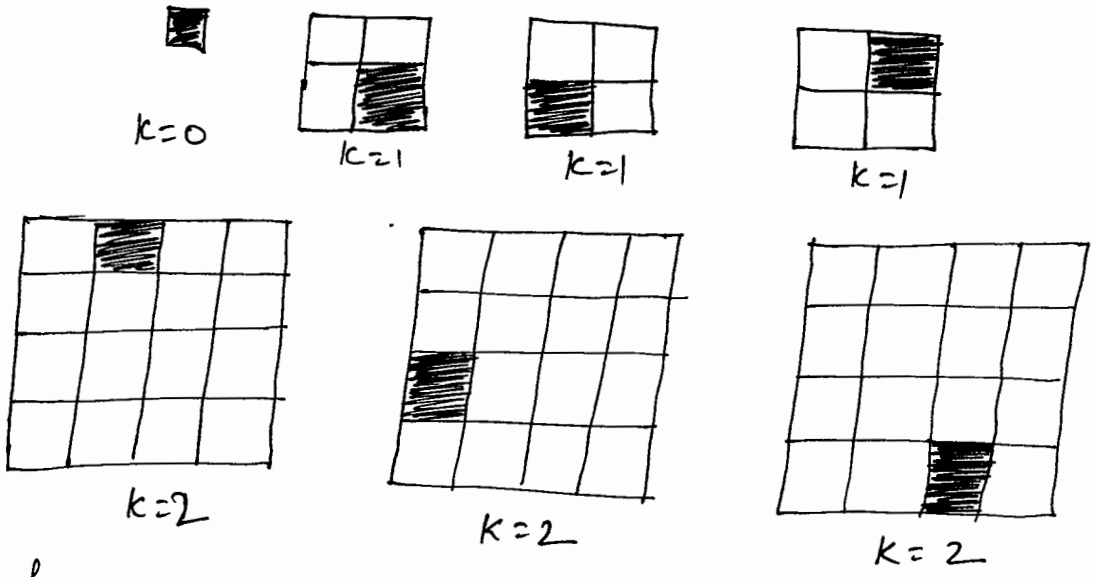


fig: Defective chessboard.

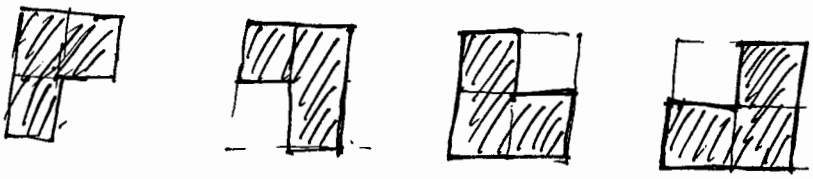


fig Triominoes with different orientations

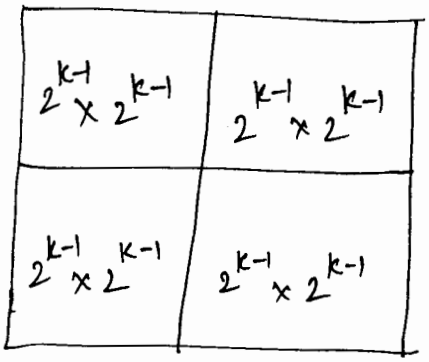
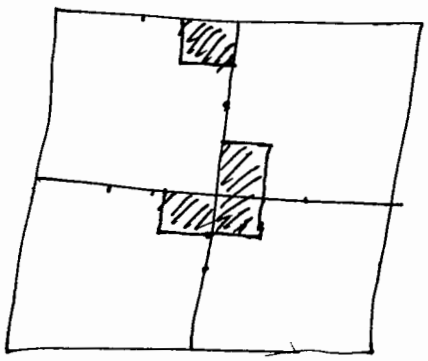


fig: Partitioning



Triomino placed.

```
Algorithm TileBoard (topRow, topColumn,
                     defectRow, defectColumn,
                     Size)
// topRow is row no: of top-left corner of board
// topColumn is Column no: of top-left corner of board.
// defectRow is row no: of defective Square.
// defectColumn is column no: of defective Square.
// Size is length of one side of chess board.
{
    if (Size = 1) return;
    tileToUse := tile++;
    quadrantSize := Size/2;
    // tile top-left quadrant
    if (defectRow < topRow + quadrantSize &&
        defectColumn < topColumn + quadrantSize) then
     // defect is in this quadrant
        TileBoard (topRow, topColumn, defectRow,
                   defectColumn, quadrantSize);
    Else
    {
        // no defect in their quadrant.
        // place a tile in bottom-right corner
        board [topRow + quadrantSize - 1][topColumn
              + quadrantSize - 1] := tileToUse;
        // tile the rest
        TileBoard (topRow, topColumn, topRow +
        quadrantSize - 1, topColumn + quadrantSize - 1,
        quadrantSize);
    }
    // Code for remaining three quadrants is Similar
}
```

FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY

Analysis of algorithms means to investigate an algorithm's efficiency with respect to resources:

- **running time (time efficiency)** and
- **memory space (space efficiency)**

Time being more critical than space, we concentrate on Time efficiency of algorithms. The theory developed, holds good for space complexity also.

Experimental Studies: requires writing a program implementing the algorithm and running the program with inputs of varying size and composition. It uses a function, like the built-in clock() function, to get an accurate measure of the actual running time, then analysis is done by plotting the results.

Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used

Theoretical Analysis: It uses a high-level description of the algorithm instead of an implementation. Analysis characterizes running time as a function of the input size, *n*, and takes into account all possible inputs. This allows us to evaluate the speed of an algorithm independent of the hardware/software environment. Therefore theoretical analysis can be used for analyzing any algorithm

Framework for Analysis

We use a hypothetical model with following assumptions

- Total time taken by the algorithm is given as a function on its input size
- Logical units are identified as one step
- Every step require ONE unit of time
- Total time taken = Total Num. of steps executed

Input's size: Time required by an algorithm is proportional to size of the problem instance. For e.g., more time is required to sort 20 elements than what is required to sort 10 elements.

Units for Measuring Running Time: Count the number of times an algorithm's **basic operation** is executed. (**Basic operation**: The most important operation of the algorithm, the operation contributing the most to the total running time.) For e.g., The basic operation is usually the most time-consuming operation in the algorithm's innermost loop.

Consider the following example:

ALGORITHM sum_of_numbers (A[0… n-1])
// Functionality : Finds the Sum
// Input : Array of n numbers
// Output : Sum of 'n' numbers
i ← 0
sum ← 0
while i < n
    sum ← sum + A[i] ⟶ n
    i ← i + 1
return sum

Total number of steps for basic operation execution, C (n) = n

NOTE:
Constant of fastest growing term is insignificant: Complexity theory is an Approximation theory. We are not interested in exact time required by an algorithm to solve the problem. Rather we are interested in order of growth. i.e

- How much faster will algorithm run on computer that is twice as fast?
- How much longer does it take to solve problem of double input size?

We can crudely estimate running time by

T (n) ≈ Cop ∗ C (n)

Where,
T (n): running time as a function of n.
Cop : running time of a single operation.
C (n): number of basic operations as a function of n.

Order of Growth: For order of growth, consider only the leading term of a formula and ignore the constant coefficient. The following is the table of values of several functions important for analysis of algorithms.

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

Worst-case, Best-case, Average case efficiencies

Algorithm efficiency depends on the **input size n.** And for some algorithms efficiency depends on **type of input**. We have best, worst & average case efficiencies.

Worst-case efficiency: Efficiency (number of times the basic operation will be executed) **for the worst case input of size n.** *i.e.* The algorithm runs the longest among all possible inputs of size n.

Best-case efficiency: Efficiency (number of times the basic operation will be executed) **for the best case input of size n.** ***i.e.*** The algorithm runs the fastest among all possible inputs of size n.

Average-case efficiency: Average time taken (number of times the basic operation will be executed) **to solve all the possible instances (random) of the input.** NOTE: NOT the average of worst and best case

Asymptotic Notations

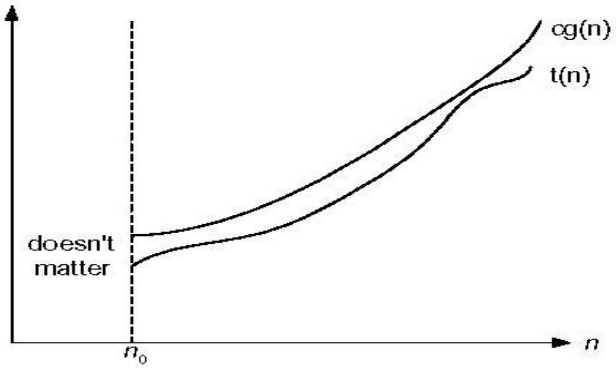
Asymptotic notation is a way of comparing functions that ignores constant factors and small input sizes. Three notations used to compare orders of growth of an algorithm's basic operation count are: **O, Ω, Θ notations**

Big Oh- O notation

Definition:

A function t(n) is said to be in O(g(n)), denoted **t(n) ∈O(g(n))**, if t(n) is bounded above by some constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n0 such that

t(n) ≤ cg(n) for all n ≥ n0



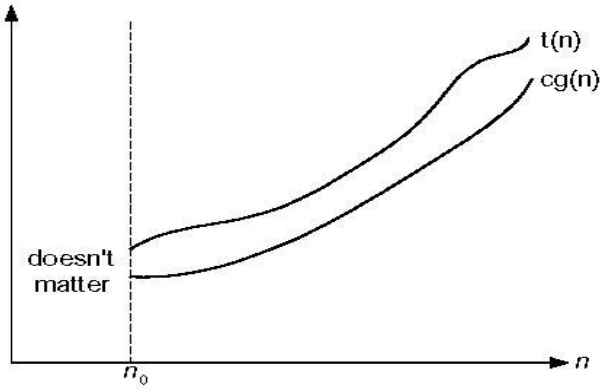
Big-oh notation: $t(n) \in O(g(n))$

Big Omega- Ω notation

Definition:

A function t (n) is said to be in Ω (g(n)), denoted **t(n) ∈ Ω (g (n)),** if t (n) is bounded below by some constant multiple of g (n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n0 such that

t(n) ≥ cg(n) for all n ≥ n0



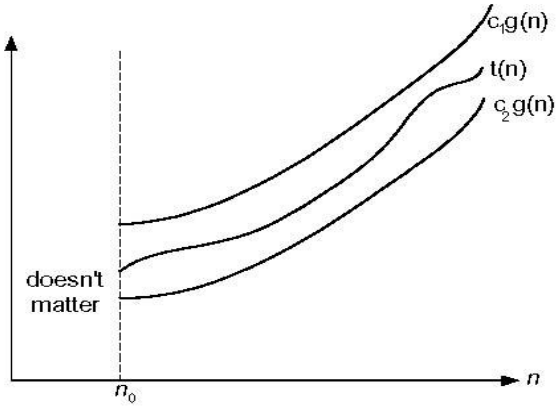
Big-omega notation: $t(n) \in \Omega(g(n))$

Big Theta- Θ notation

Definition:

A function t (n) is said to be in Θ(g (n)), denoted **t(n) ∈ Θ(g (n)),** if t (n) is bounded both above and below by some constant multiple of g (n) for all large n, i.e., if there exist some positive constant c1 and c2 and some nonnegative integer n0 such that

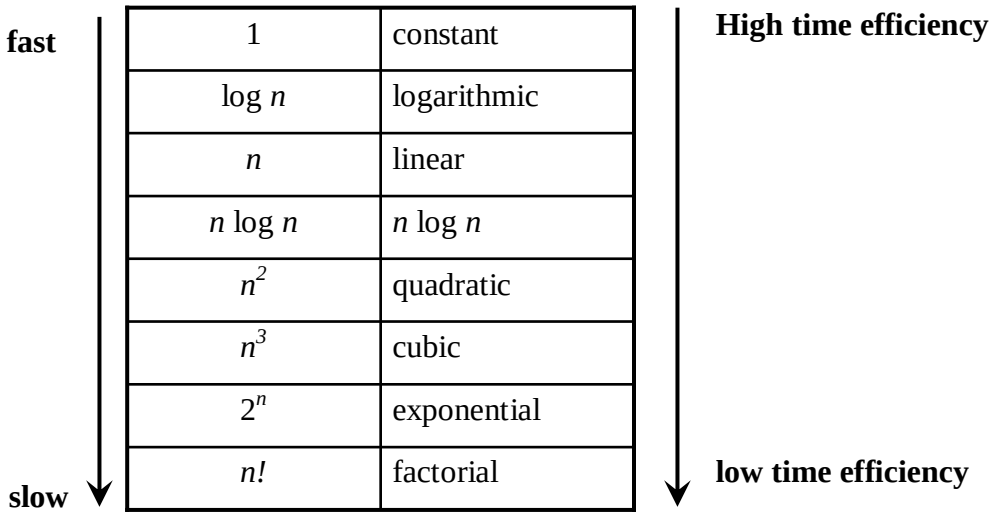
c2 g (n) ≤ t (n) ≤ c1 g (n) for all n ≥ n0



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

Basic Efficiency classes

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



1	constant
log *n*	logarithmic
n	linear
n log *n*	*n* log *n*
n^2	quadratic
n^3	cubic
2^n	exponential
n!	factorial

Mathematical analysis (Time Efficiency) of Non-recursive Algorithms

General plan for analyzing efficiency of non-recursive algorithms:

1. Decide on parameter n indicating **input size**
2. Identify algorithm's **basic operation**
3. Check whether the number of times the basic operation is executed depends only on the input size n. If it also depends on the type of input, investigate **worst, average, and best case efficiency** separately.
4. Set up **summation** for C(n) reflecting the number of times the algorithm's basic operation is executed.
5. Simplify summation using standard formulas

Example: Finding the largest element in a given array

ALOGORITHM *MaxElement*(A[*0..n*-1])

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

currentMax ← *A*[0]
for *i* ← 1 to *n* – 1 do
    if *A*[*i*] > *currentMax*
        currentMax ← *A*[*i*]
return *currentMax*

Analysis:

1. Input size: number of elements = n (size of the array)
2. Basic operation:
   a) Comparison
   b) Assignment
3. NO best, worst, average cases.
4. Let C (*n*) denotes number of comparisons: Algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bound between 1 and *n* – 1.

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

5. **Simplify summation** using standard formulas

$$\mathbf{C}(n) = \sum_{i=1}^{n-1} 1 \qquad \mathbf{1 + 1 + 1 + \ldots + 1 \quad [(n\text{-}1) \text{ number of times}]}$$

$$\mathbf{C}(n) = \mathbf{n\text{-}1}$$

$$\mathbf{C}(n) \in \Theta(\mathbf{n})$$

Example: Element uniqueness problem

Algorithm *UniqueElements (A[0..n-1])*
//Checks whether all the elements in a given array are distinct
//Input: An array A[0..n-1]
//Output: Returns true if all the elements in A are distinct and false otherwise

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

Analysis

1. Input size: number of elements = n (size of the array)
2. Basic operation: Comparison
3. NO Best, worst, average cases EXISTS.
   Worst case input is an array giving largest comparisons.
   - Array with no equal elements
   - Array with last two elements are the only pair of equal elements
4. Let C (*n*) denotes number of comparisons in worst case: Algorithm makes one comparison for each repetition of the innermost loop i.e., for each value of the loop's variable j between its limits *i + 1 and n – 1*; and this is repeated for each

value of the outer loop i.e, for each value of the loop's variable i between its limits *0 and n – 2*

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

5. **Simplify summation** using standard formulas

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

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

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

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

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

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

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

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

$$C(n) = (n-1)(n)/2$$
$$= (n^2 - n)/2$$
$$= (n^2)/2 - n/2$$

$$\mathbf{C}(n) \in \Theta(\mathbf{n^2})$$

Mathematical analysis (Time Efficiency) of recursive Algorithms

General plan for analyzing efficiency of recursive algorithms:

1. Decide on parameter n indicating **input size**
2. Identify algorithm's **basic operation**
3. Check whether the number of times the basic operation is executed depends only on the input size n. If it also depends on the type of input, investigate **worst, average, and best case efficiency** separately.
4. Set up **recurrence relation**, with an appropriate initial condition, for the number of times the algorithm's basic operation is executed.
5. **Solve** the recurrence.

Example: Factorial function

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

Analysis:

1. Input size: given number = n
2. Basic operation: multiplication
3. NO best, worst, average cases.
4. Let M (n) denotes number of multiplications.

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0$$
$$M(0) = 0 \quad \textit{initial condition}$$

Where: M $(n - 1)$: to compute Factorial $(n - 1)$
1 :to multiply Factorial $(n - 1)$ by n

5. Solve the recurrence: Solving using *"Backward substitution method":*

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

In the ith recursion, we have

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

When i = n, we have

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

Since M (0) = 0

$$= n$$

$$\mathbf{M(n) \in \Theta(n)}$$

Example: Find the number of binary digits in the binary representation of a positive decimal integer

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

Analysis:

1. Input size: given number = n
2. Basic operation: addition
3. NO best, worst, average cases.
4. Let A (*n*) denotes number of additions.

$$A(n) = A(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1$$
$$A(1) = 0 \quad \textit{initial condition}$$

Where: A ($\lfloor n/2 \rfloor$) : to compute *BinRec* ($\lfloor n/2 \rfloor$)
1 : to increase the returned value by 1

5. Solve the recurrence:

$A(n) = A(\lfloor n/2 \rfloor) + 1$ for $n > 1$

Assume $n = 2^k$ (smoothness rule)

$A(2^k) = A(2^{k-1}) + 1$ for $k > 0$; $A(2^0) = 0$

Solving using "Backward substitution method":

$$A(2^k) = A(2^{k-1}) + 1$$
$$= [A(2^{k-2}) + 1] + 1$$
$$= A(2^{k-2}) + 2$$
$$= [A(2^{k-3}) + 1] + 2$$
$$= A(2^{k-3}) + 3$$
$$\dots$$

In the ith recursion, we have

$$= A(2^{k-i}) + i$$

When i = k, we have

$$= A(2^{k-k}) + k = A(2^0) + k$$

Since $A(2^0) = 0$

$$\mathbf{A(2^k) = k}$$

Since $n = 2^k$, HENCE $k = \log_2 n$

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

UNIT-3

The Greedy Method

The General Method, Knapsack Problem, Job Sequencing with Deadlines, Minimum-cost Spanning Trees: Prim's Algorithm, Kruskal's Algorithm Single Source shortest Path.

[Reference: Fundamentals of Computer Algorithms by Ellis Horowitz, Sahini, Sanguthevar Rajasekaran & Design & Analysis of Algorithms by V.V. muniswamy]

→ The greedy method suggests that one can devise an algorithm that works in stages, considering one i/p at a time.

→ At each stage, a decision is made regarding whether a particular i/p is in an optimal solution.

→ This is done by considering the i/ps in an order determined by some selection procedure.

→ If the inclusion of the next i/p into the partially constructed optimal solution will result in an infeasible solution, then this i/p is not added to the partial solution, otherwise it is added.

→ The greedy method control abstraction for the subset paradigm algorithm is given in the next page.

→ The function "select" selects an i/p from a[] & removes it.

→ The selected i/p's value is assigned to x.

```
Algorithm Greedy(a,n)
// a[1:n] Contains the n i/p
{
    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;
}
```

→ 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 updes the objective function.

Knapsack Problem

→ We are given n objects and a knapsack or bag.

→ Object i has a weight w_i and the knapsack has a capacity m.

→ If a fraction x_i, $0 \leq x_i \leq 1$ of object i is placed into the knapsack, then a profit of $p_i x_i$ is earned.

→ The objective is to obtain a filling of the knapsack that maximizes the total profit earned.

Formally, the problem can be stated as

$$\text{maximize} \sum_{1 \le i \le n} P_i x_i$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le m$$

and $0 \le x_i \le 1, \quad 1 \le i \le n$

The profits & weights are positive numbers.

* **Greedy strategies for the knapsack problem are:**

(1) From the remaining items, select the item with maximum profit that fits into the knapsack.

(2) From the remaining items, select the item that has minimum weight & also fits into the knapsack.

(3) From the remaining items, select the one with maximum P_i/w_i that fits into the knapsack.

* **Algorithm for greedy strategies for the knapsack prob.**

```
Algorithm Greedy knapsack (m, n)
// P[1:n] & w[1:n] contains profits & weights
// of the n objects ordered such that P[i]/w[i]
// >= P[i+1]/w[i+1], m is the knapsack size &
// x[1:n] is the solution vector.
{
    for i := 1 to n do x[i] := 0.0; //initialize x
    U := m;
    for i := 1 to n do
    {
        if (w[i] > U) then break;
        x[i] := 1.0; U := U - w[i];
    }
    if (i <= n) then x[i] := U / w[i];
}
```

Example: Consider the following instance of the knapsack problem: $n=3$, $m=20$, $(P_1, P_2, P_3) = (25, 24, 15)$ and $(w_1, w_2, w_3) = (18, 15, 10)$.

Four feasible solutions are

	(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum P_i x_i$
1.	$(1/2, 1/3, 1/4)$	16.5	24.25
2.	$(1, 2/15, 0)$	20	28.2
3.	$(0, 2/3, 1)$	20	31
4.	$(0, 1, 1/2)$	20	31.5

* Of these four feasible solution, solution 4 yields the maximum profit.

Imp points to note: * In case the sum of all the weights is $\leq m$ then $x_i = 1$, $1 \leq i \leq n$ is an optimal solution.

* All optimal solutions will fill the knapsack exactly.

* If $P_1/w_1 \geq P_2/w_2 \geq \cdots \geq P_n/w_n$, then Greedy knapsack generates an optimal solution to the given instance of the knapsack problem.

Job Sequencing with Deadlines.

* We are given a set of n objects.
* Associated with job i is an integer deadline $d_i \geq 0$ and a profit $P_i > 0$.
* For any job i the profit P_i is earned iff the job is completed by its deadline.

* To complete a job, one has to process the job on a machine for one unit of time.
* only one machine is available for processing jobs.

Example: Let $n=4$, $(P_1, P_2, P_3, P_4) = (100, 10, 15, 27)$ & $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$

The feasible solutions & their values are.

	Feasible Solution	Processing sequence	value.	
1.	(1, 2)	2, 1	110	
2.	(1, 3)	1, 3 or 3, 1	115	
3.	(1, 4)	4, 1	(127)	optimal solution.
4.	(2, 3)	2, 3	25	
5.	(3, 4)	4, 3	42	
6.	(1)	1	100	
7.	(2)	2	10	
8.	(3)	3	15	
9.	(4)	4	27	

```
Algorithm JS (d, j, n)
// d[i] >= 1, 1 <= i <= n are the deadlines, n >= 1
// The jobs are ordered such that P[1] >= P[2] ---- >= P[n].
// J[i] is the i^th job in the optimal soln, 1 <= i <= k.
// Also, at termination d[J[i]] <= d[J[i+1]], 1 <= i <= k.
{
    d[0] := J[0] := 0; // initialize.
    J[1] := 1; // Include job 1.
    K := 1;
    for i := 2 to n do
    {
        // Consider jobs in nonincreasing order of P[i].
        // Find position for i & check feasibility of insertion.
```

```
    r := k;
    while ((d[J[r]] > d[i]) and (d[J[r]] ≠ r)) do r := r-1;
    if ((d[J[r]] ≤ d[i]) and (d[i] > r)) then
    {
        // Insert i into J[].
        for q := k to (r+1) step -1 do
            J[q+1] := J[q];
        J[r+1] := i;
        k := k+1;
    } // end of if.
  } // end of for
    return k;
} // end of the Algorithm.
```

Example 2: Let $n=5$, $(P_1,..,P_5) = (20,15,10,5,1)$ and $(d_1,..,d_5) = (2,2,1,3,3)$. Using the above feasibility rule, we have

J	assigned slots	job Considered	action	Profit.
ϕ	none	1	assign to [1,2]	0
{1}	[1,2]	2	assign to [0,1]	20
{1,2}	[0,1], [1,2]	3	Cannot fit; reject	35
{1,2}	[0,1], [1,2]	4	assign to [2,3]	35
{1,2,4}	[0,1], [1,2], [2,3]	5	reject	40.

* The Optimal Solution is $J = \{1,2,4\}$ with a profit of 40.
* The Computing time of JS algo is $O(n^2)$.

Minimum-cost Spanning Time

Spanning Tree: Let $G = (V, E)$ be an undirected connected graph. A subgraph $t = (V, E')$ of G is a spanning tree of G iff t is a tree.

OR

A spanning tree of a connected graph is its connected acyclic subgraph that contains all the vertices of a graph.

Minimum Spanning tree: The minimum spanning tree of a weighted connected graph is its spanning tree of the smallest weight, where the weight of tree is defined as sum of weights of all its Edges.

Algorithm Prim's (G)
// prim's algorithm for constructing a min spanning tree
// I/P: A weighted connected graph $G = \langle V, E \rangle$
// O/P: E_T, the set of Edges composing a minimum Spanning tree of G.

$V_T \leftarrow \{V_0\}$ // Initialize the vertex set
$E_T \leftarrow \{\phi\}$
for $i \leftarrow 1$ to $|V| - 1$ do
    find a min-weight edge $e^* = (v^*, u^*)$ among all the Edges (v, u) such that v is in V_T & u is in $V - V_T$
    $V_T \leftarrow V_T \cup \{u^*\}$
    $E_T \leftarrow E_T \cup \{e^*\}$
return E_T.

Example 1:

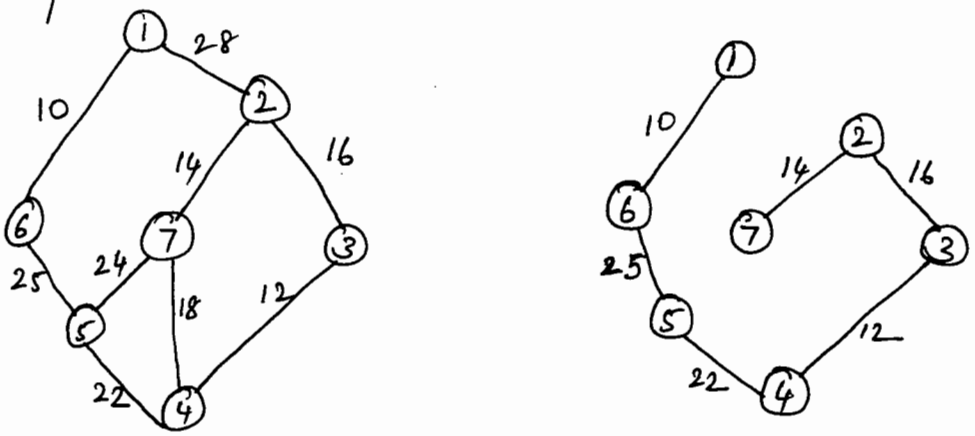
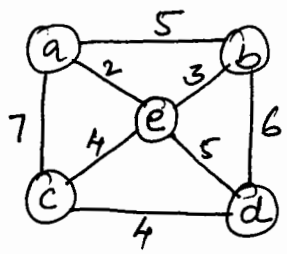


Fig: A graph & its minimum cost spanning tree.

Algorithm Prim (E, cost, n, t) (Ref: Horowitz & Sahni)
// E is the set of edges in G. cost[1:n, 1:n] is the cost
// adjacency matrix of an n vertex graph.
// A min spanning tree is stored in t[1:n-1, 1:2]
// Finally cost is returned.
{
    Let (k, l) be an edge of min 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;
}

Example.2 Find the minimum Spanning tree by prim's algorithm.



Sol^n: Let source vertex be (a)

Step1: ⟨Iteration-1⟩

$V_T \leftarrow \{a\}$

$E_T \leftarrow \phi$

$V - V_T \leftarrow \{b, c, d, e\}$

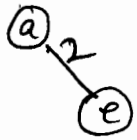
sum = 0 (cost involved)

min {⟨a,b⟩ ⟨a,c⟩ ⟨a,d⟩, ⟨a,e⟩}

min { 5, 7, ∞, 2}

= 2 (a→e)

sum ← 0+2 = 2



Step2: $V_T = \{a, e\}$

$E_T = \{\langle a, e\rangle\}$

$V - V_T = \{b, c, d\}$

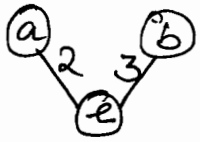
sum = 2

min(⟨a,b⟩ ⟨a,e⟩ ⟨a,d⟩ ⟨e,b⟩⟨e,c⟩⟨e,d⟩)

min(5, 7, ∞, 3, 4, 5)

= 3 (e→b)

sum = 2+3 = 5



Step3: $V_T = \{a, e, b\}$

$E_T = \{\langle a, e\rangle \langle e, b\rangle\}$

$V - V_T = \{c, d\}$

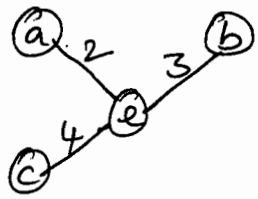
sum = 5

min {⟨a,c⟩ ⟨a,d⟩ ⟨e,c⟩ ⟨e,d⟩ ⟨b,c⟩ ⟨b,d⟩}

= min {7, ∞, 4, 5, ∞, 6}

= 4 (e→c)

sum = 5+4 = 9



Step4: $V_T = \{a, e, b, c\}$

$E_T = \{\langle a, e\rangle \langle e, b\rangle \langle e, c\rangle\}$

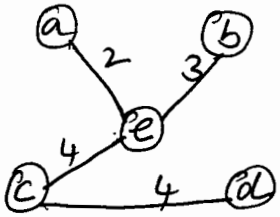
$V - V_T = \{d\}$

sum = 9

min {⟨a,d⟩ ⟨e,d⟩ ⟨b,d⟩ ⟨c,d⟩}

min (∞, 5, 6, 4)

= 4 (c→d) sum = 9+4 = 13



Finally $E_T = \{\langle a, e\rangle \langle e, b\rangle \langle e, c\rangle \langle c, d\rangle\}$

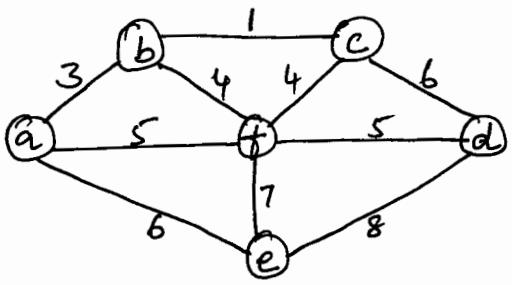
Cost of Constructing the spanning tree = (13)

Ex: 3 : Find the min Spanning tree using Prim's algo..

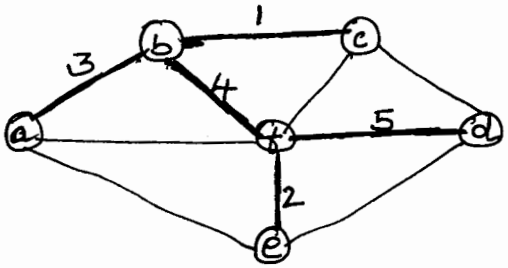
Ans : 3 + 1 + 4 + 2 + 5

E_T = {⟨a,b⟩ ⟨b,c⟩ ⟨b,f⟩ ⟨f,e⟩ ⟨f,d⟩}

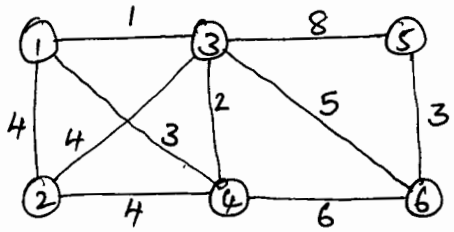
Sum = 15



=>

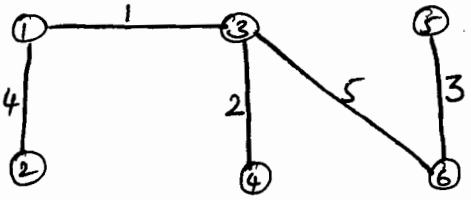


Ex: 4 : Find the min Spanning tree using Prim's algo..



Ans: {⟨1,2⟩ ⟨1,3⟩ ⟨3,4⟩ ⟨3,6⟩ ⟨5,6⟩}

4 + 1 + 2 + 5 + 3



Kruskal's Algorithm

* Kruskal's algo begins by sorting the graphs edges in increasing order of their weights. Then it scans this sorted list, starting with the empty subgraph it adds the next edge on the list to the current subgraph if such an inclusion does not create a cycle, & if it creates a cycle, it has to be skipped.

Algorithm Kruskal(G) {Ref: Anany Levitin}
// kruskal's algo for constructing a min spanning tree
// I/p: A weighted connected graph $G = \langle V, E \rangle$
// O/p: E_T, the set of edges composing a min spanning tree of G sort E in ascending order

$E_T \leftarrow \phi$; ecounter $\leftarrow 0$ // init the set of tree edges & its size

$k \leftarrow 0$ // init the no: of processed edges.

while ecounter $< |V| - 1$ do

$k \leftarrow k + 1$

if $E_T \cup \{e_{ik}\}$ is acyclic

$E_T \leftarrow E_T \cup \{e_{ik}\}$;

ecounter $\leftarrow$ encounter $+ 1$

return E_T.

Example: 1

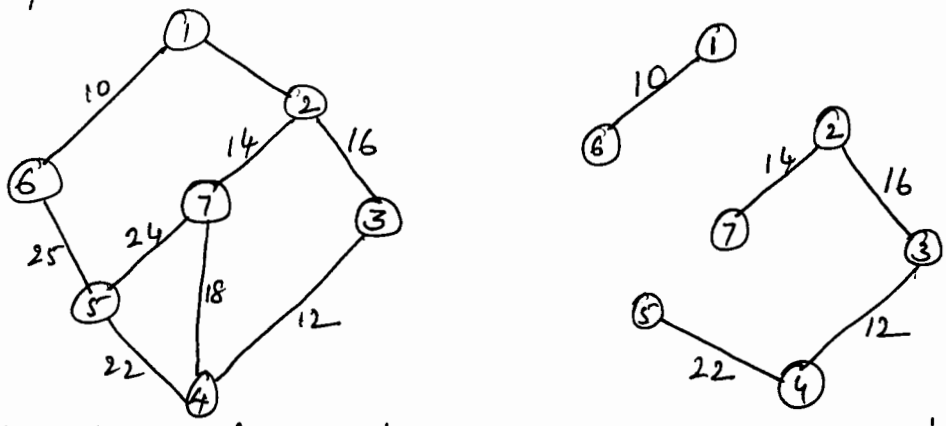
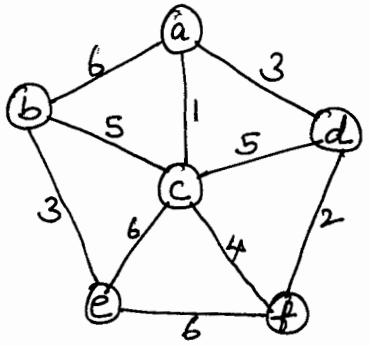


fig: A graph & its minimum spanning tree using kruskal's algo

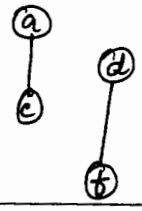
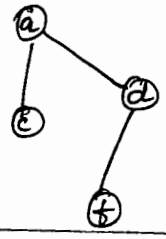
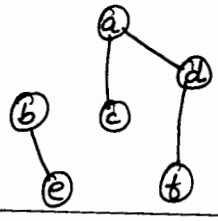
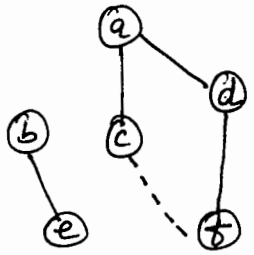
```
Algorithm Kruskal (E, cost, n, t)  {ref: Horowitz & Sahani}
// E is the set of edges in G. G has n vertices.
// cost[u,v] is the cost of edge (u,v). t is the
// set of edges in the minimum-cost spanning
// tree. The final cost is returned.
{
    Construct a heap out of the edge costs using Heapify
    for i := 1 to n do parent[i] := -1;
    // Each vertex is in a different set.
    i := 0; mincost := 0.0;
    while ((i < n-1) and (heap not Empty)) do
    {
        Delete a min 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;
}
```

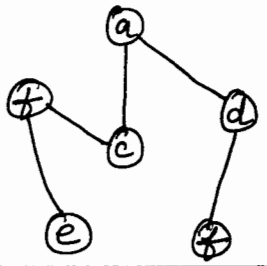
Example 2: Find the minimum spanning tree using kruskal's algorithm.



Initial step: sort the Edges w.r.t edges

Edges	Cost
a→c	1
d→f	2
a→d	3
b→e	3
c→f	4
b→c	5
c→d	5
a→b	6
e→c	6
e→f	6

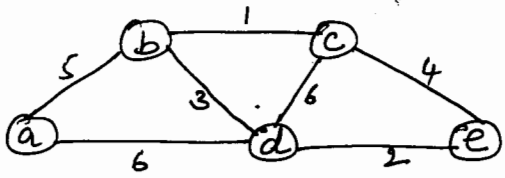
Sl No	Edges	Graph	Accept/Reject
1	a→c		A.
2	d→f		A.
3	a→d		A.
4	b→e		A.
5	c→f		R.

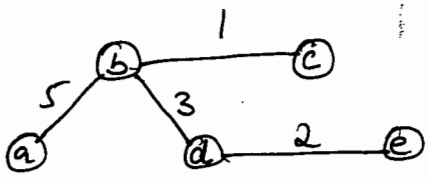
6.	b→c		A.
7.	c→d	—	R
8.	a→b	—	R
9.	e→c	—	R
10.	e→f	—	R

Total Cost } = 14

Example: 3

Find the min Spanning tree using kruskal's algo.





Ans: Cost = 11

Dijkstra's Algorithm.

* It is a single source shortest path problem which is generally used in the applications of Computer Science. (Networks).
* The Dijkstra's algo finds the shortest paths from a given source vertex to all the remaining vertices in a graph. The length of the path is the sum of the cost of the edges on the path.

Algorithm Dijkstra(G, s) {Ref: Anany Levitin}
// Dijkstra's algo for single-source shortest path
// I/p: A weighted connected graph with nonnegative weights & its vertex s
// O/p: The length d_v of a shortest path from s to v & its penultimate vertex p_v for every vertex v in V

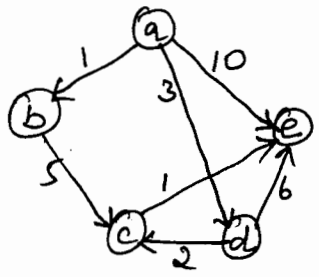
```
Initialize(Q) //initialize vertex priority queue to empty
for every vertex v in V do
    dv ← ∞, Pv ← null
    Insert(Q, v, dv) //init vertex priority in the
                       priority queue.
ds ← 0; Decrease(Q, s, ds) //update priority of s
                             with ds
VT ← ϕ
for i ← 0 to |V| -1 do
    u* ← DeleteMin(Q) //delete the minimum priority
                        element
    VT ← VT ∪ {u*}
    for every vertex u in V-VT that is adjacent
                                to U* do
        if du* + w(u*, u) < du
            du ← du* + w(u*, u); Pu ← u*.
            Decrease(Q, u, du).
```

OR

```
Algorithm Dijkstra(G, s)
// Imp SSSP from the given src vertex
// I/P & O/P.
    S ← {1}
    for i ← 2 to n do
        D[i] ← C[1, i]
    for i ← 1 to n do
        Choose a vertex w in V-S such that D[w]
        is minimum
        S ← S ∪ w
        for each vertex v in V-S do
            D[V] ← min {D[V], D[w] + C[w, V]}
    return Di
```

```
Algorithm ShortestPath(v, cost, dist, n)   {Ref: Horowitz & Sahni}
// dist[j], 1 ≤ j ≤ n, is set to the length of
// the shortest path from vertex v to vertex j in a
// diagraph 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; dist[v] := 0.0; // 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];
    }
}
```

Example:1: Solve the following instances of Single Source Shortest path problem with Vertex (a) as Source.



Cost matrix, C

	a	b	c	d	e
a	0	1	∞	3	10
b	∞	0	5	∞	∞
c	∞	∞	0	∞	1
d	∞	∞	2	0	6
e	∞	∞	∞	∞	0

<u>initial Step:</u>

V = {a, b, c, d, e, f} set of Vertices

S = {a} source

initial distance D [b, c, d, e] = D [1, ∞, 3, 10]

<u>Step 1:</u> * Find w {the Edge with the min cost from a}

min (D[b], D[c], D[d], D[e]) = min (1, ∞, 3, 10) = 1

∴ w = b

* Add w to S

S = {a, b}

V = V - S = {c, d, e}

* Now, find D [v] ie $D[v] = \min\{D[v], D[w] + c[w,v]\}$

$D[c] = \min\{D[c], D[b] + c[b,c]\} = \min\{\infty, 1+5\} = 6$

$D[d] = \min\{D[d], D[b] + c[b,d]\} = \min\{3, 1+\infty\} = 3$

$D[e] = \min\{D[e], D[b] + c[b,e]\} = \min\{10, 1+\infty\} = 10$

Iteration	S	w	D(b)	D(c)	D(d)	D(e)
initial	{a}	—	1	∞	3	∞
Step 1	{a, b}	b	1	6	3	10
Step 2	{a, b, d}	d	1	5	3	9
Step 3	{a, b, d, c}	c	1	5	3	6

Step2: * Find w

$\min(D[c], D[d], D[e]) = \min(6,3,10) = 3$ ie w=d

* Add w to S

$S = \{a, b, d\}$

$V = V - S = \{c, e\}$

* Now, find D[V]

$D[c] = \min\{D[c], D[d] + C[d,c]\} = \min\{6, 3+2\} = 9$

Step3: * Find w

$\min\{D[c], D[e]\} = \min\{5, 9\} = 5$ ie w=c

* Add w to S

$S = \{a, b, c, d\}$

$V = V - S = \{e\}$.

* Now find D[V]

$D[e] = \min\{D[e], D[c] + C[c,e]\} = \min\{9, 5+1\} = 6$

Finally

* Find w

$\min(D[e]) = 6$ w=e

* Add w to S

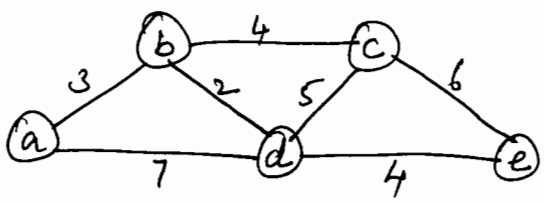
$S = \{a, b, c, d, e\}$

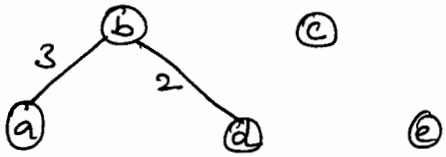
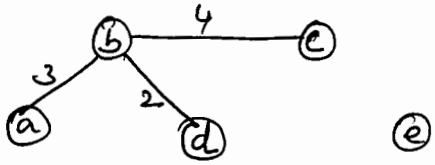
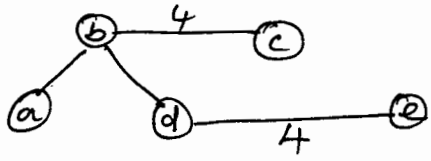
$V = V - S = \phi$

$\therefore D[b]=1, D[c]=5, D[d]=3, D[e]=6$

from (a) Vertex.

Example: find the single source shortest path from (a) in the graph shown below using Dijkstra's Algo..



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)		

Dynamic Programming

unit - iv

Syllabus: The General Method, Warshall's Algo.. Floyd's algo.. (all-pairs shortest path), Single Source Shortest paths: General weights - BellMan-Ford algo.., 0/1 knapsack, The Travelling Salesperson Problem.

The General Method:

→ The word "programming" in the name of this technique stands for "planning".

→ Dynamic programming is an algorithm design method that can be used when the solution to a problem can be viewed as the results of a sequence of decisions.

Examples: knapsack problem, Optimal merge patterns, Shortest path, Principle of Optimality.

Warshall's algorithm

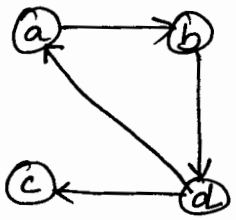
→ Warshall's algorithm constructs the transitive closure of a given digraph with n vertices through a series of n-by-n boolean matrices:

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

→ Transitive closure: The transitive closure of a directed graph with n vertices can be defined as the n-by-n boolean matrix $T = \{t_{ij}\}$,

in which the element in the i^{th} row ($1 \le i \le n$) & the j^{th} column ($1 \le j \le n$) is 1 if there exits a non trivial directed path from the i^{th} vertex to the j^{th} vertex; otherwise t_{ij} is 0.

Example 1



Digraph

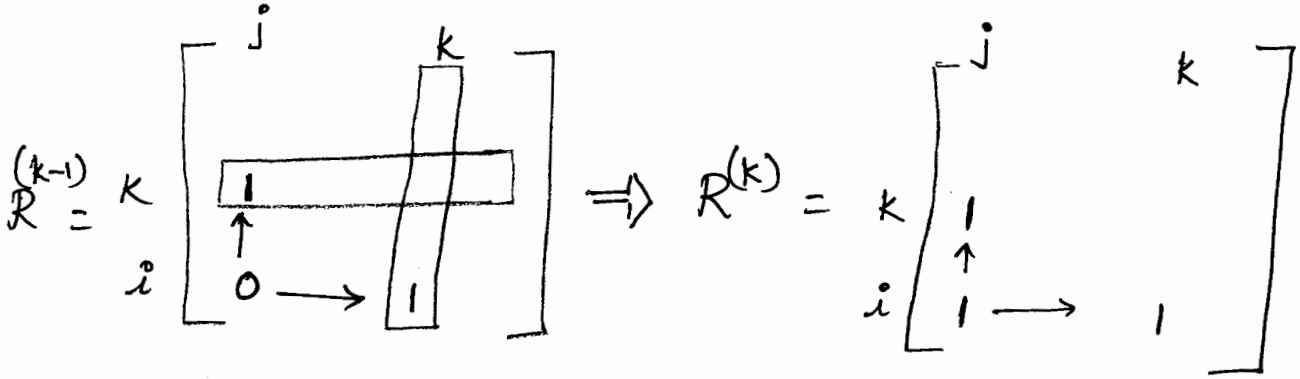
$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Adjacency Matrix

$$T = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Transitive closure

→ The Rule for changing zeros in warshall's algorithm.



→ The following formula ~~is~~ used for generating the elements of matrix $R^{(k)}$ from the elements of matrix $R^{(k-1)}$

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

R^0 =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

R^1 =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	1	1	0

R^2 =

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

R^3 =

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

R^4 =

	a	b	c	d
a	1	1	1	1
b	1	1	1	1
c	0	0	0	0
d	1	1	1	1

Algorithm Warshalls (A[1..n, 1..n])
// Implements Warshall's algo for Computing the Transitive closure
// Input: The adjacency matrix A of a digraph with n vertices
// Output: The transitive closure of the digraph

$R^{(0)} \leftarrow A$
for k ← 1 to n do
  for i ← 1 to n do
    for j ← 1 to n do
      $R^{(k)}[i,j] \leftarrow R^{k-1}_{[i,j]}$ or $R^{k-1}_{[i,k]}$ and $R^{k-1}_{[k,j]}$
Return $R^{(n)}$.

Efficiency.

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

$$= n^2 \sum_{k=0}^{n-1} 1 \qquad = n^2(n-1-0+1) = n^2 * n = n^3$$

$\therefore$ it is $\theta(n^3)$

Example2 Apply warshall algorithm to find the transitive closureness of a digraph defined by the adjacency matrix.

$R^0 =$

	a	b	c	d
a	0	1	0	0
b	0	0	1	0
c	0	0	0	1
d	0	0	0	0

$R^1 =$

	a	b	c	d
a	0	1	0	0
b	0	0	1	0
c	0	0	0	1
d	0	0	0	0

$R^2 =$

	a	b	c	d
a	0	1	1	0
b	0	0	1	0
c	0	0	0	1
d	0	0	0	0

$R^3 =$

	a	b	c	d
a	0	1	1	1
b	0	0	1	1
c	0	0	0	1
d	0	0	0	0

$R^4 =$

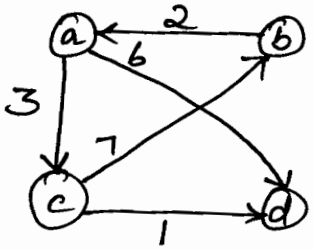
	a	b	c	d
a	0	1	1	1
b	0	0	1	1
c	0	0	0	1
d	0	0	0	0

Floyd's Algorithm.

→ This algorithm is used to calculate the all pair shortest path for a weighted graph

→ offcourse the graph may be either directed or undirected.

Example1



$$d[i,j] = \min\left[d[i,j], d[i,k] + d[k,j]\right]$$

$D^0 =$

	a	b	c	d
a	0	∞	3	6
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

$D^1 =$

	a	b	c	d
a	0	∞	3	6
b	2	0	5	8
c	∞	7	0	1
d	6	∞	9	0

$D^2 =$

	a	b	c	d
a	0	∞	3	6
b	2	0	5	8
c	9	7	0	1
d	6	∞	9	0

$D^3 =$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	9	7	0	1
d	6	16	9	0

$D^4 =$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	7	7	0	1
d	6	16	9	0

Example2 : solve the all pair shortest path for the diagraph with weighted matrix given.

$D^0 =$

	a	b	c	d	e
a	0	2	∞	1	∞
b	6	0	3	2	∞
c	∞	∞	0	4	∞
d	∞	∞	2	0	3
e	3	∞	∞	∞	0

$D^1 =$

	a	b	c	d	e
a	0	2	∞	1	∞
b	6	0	3	2	∞
c	∞	∞	0	4	∞
d	∞	∞	2	0	3
e	3	5	∞	∞	0

$D^2 =$

	a	b	c	d	e
a	0	2	5	1	∞
b	6	0	3	2	∞
c	∞	∞	0	4	∞
d	∞	∞	2	0	3
e	8	5	8	4	0

$D^3 =$

	a	b	c	d	e
a	0	2	5	1	∞
b	6	0	3	2	∞
c	∞	∞	0	4	∞
d	∞	∞	2	0	3
e	3	5	8	4	0

$D^4 =$

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

$D^5 =$

	a	b	c	d	e
a	0	2	5	1	4
b	6	0	3	2	5
c	10	12	0	4	7
d	6	8	2	0	3
e	3	5	6	4	0

```
Algorithm Floyd (w[1..n, 1..n])
// Implements Floyd's algo for the all pair
   Shortest-path problem
// I/p: The weight matrix w of a graph
// O/p: The distance matrix of the shortest
   path lengths
D ← w // is not necessary if w can be
          over written
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}
return D.
```

Analysis

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

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

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

$$= \sum_{k=0}^{n-1} n\,(n-1-0+1)$$

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

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

$$= n^2 \sum_{k=0}^{n-1} 1$$

$$= n^2 (n-1-0+1)$$

$$= n^2 * n$$

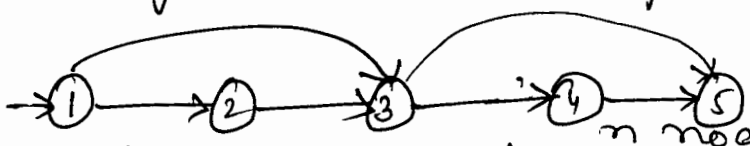
$$= n^3$$

$\Rightarrow$ ∴ the time Complexity of warshall's algo is $\theta(n^3)$.

Single-Source Shortest paths: General weights

Problems with generic SSSP algo are.

① Complexity could be Exponential time



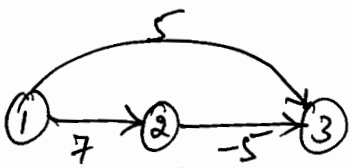
n nodes $O(2^{n/2})$.

{This problem is fixed by Dijkstra's algo}

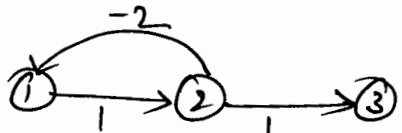
② Will not even terminate if there is a -ve cycle reachable by the source.

{This problem is overcome by Bellman-Ford algo.}

```
Algorithm BellmanFord(v, cost, dist, n)
// Single-Source // all-destinations shortest
// paths with negative edge costs.
{
    for i := 1 to n do // initialize dist.
        dist[i] := cost[v, i];
    for k := 2 to n-1 do
        for each u such that u ≠ v & u has.
            at least one incoming edge do
            for each <i, u> in the graph do
                if dist[u] > dist[i] + cost[i,u] then
                    dist[u] := dist[i] + cost[i,u];
}
```

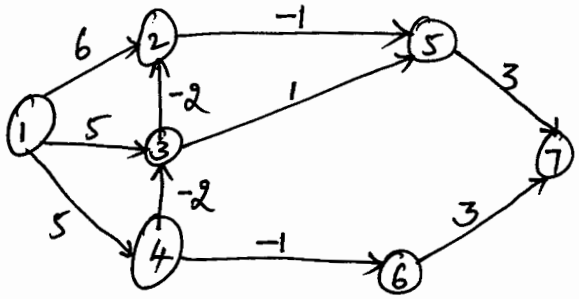


dijkstra's algo will not work for this ~~d~~graph



dijkstra's algo will work for this graph.

Example: Find the shortest path from node 1 to every node in the graph following graph using Bellman-Ford algo.



Soln

dist[1,7]

K	1	2	3	4	5	6	7
1	0	6	5	5	∞	∞	∞
2	∞	0	∞	∞	-1	∞	∞
3	∞	-2	0	∞	1	∞	∞
4	∞	∞	-2	0	∞	-1	∞
5	∞	∞	∞	∞	0	∞	3
6	∞	∞	∞	∞	∞	0	3
7	∞	∞	∞	∞	∞	∞	0

d [0 6 5 5 ∞ ∞ ∞]

The value of $d^k_{[x]} = \min\{ d^{k-1}_{[u]}, d^{k-1}_{[i]} + Cost[i,u] \}$

Compute d for k = 2, 3, 4, 5, 6 as shown below.

When k=2.

d	i	6	5	5	∞	∞	∞
0	1	0+6	0+5	0+5	0+∞	0+∞	0+∞
6	2	6+0	6+∞	6+∞	6-1	6+∞	6+∞
5	3	**5-2**	5+0	5+∞	5+1	5+∞	5+∞
5	4	5+∞	5-2	5-0	5+∞	5-1	5+∞
∞	5	∞+∞	∞+∞	∞+∞	∞+0	∞+∞	∞+3
∞	6	∞+∞	∞+∞	∞+∞	∞+∞	∞+0	∞+3
∞	7	∞+∞	∞+∞	∞+∞	∞+∞	∞+∞	∞+0

	2	3	4	5	6	7
1	6	5	5	∞	∞	∞
2	6	∞	∞	5	∞	∞
3	3	5	∞	6	∞	∞
4	∞	3	5	∞	4	∞
5	∞	∞	∞	∞	∞	∞
6	∞	∞	∞	∞	∞	∞
7	∞	∞	∞	∞	∞	∞
min →	3	3	5	5	4	∞

When k=3

d	i	3	3	5	5	4	∞
0	1	0+6	0+5	0+5	0+∞	0+∞	0+∞
3	2	3+0	3+∞	3+∞	3-1	3+∞	3+∞
3	3	3-2	3+0	3+∞	3+1	3+∞	3+∞
5	4	5-∞	5-2	5+0	5+∞	5-1	5+∞
5	5	5+∞	5+∞	5+∞	5+0	5+∞	5+3
4	6	4+∞	4+∞	4+∞	4+∞	4+0	4+3
∞	7	∞+∞	∞+∞	∞+∞	∞+∞	∞+∞	∞+0
min:		1	3	5	2	4	7

when k=4

d	i	1	3	5	2	4	7
0	1	0+6	0+5	0+5	0+∞	0+∞	0+∞
1	2	1+0	1+∞	1+∞	1-1	1+∞	1+∞
3	3	3-2	3+0	3+∞	3+1	3+∞	3+∞
5	4	5+∞	5-2	5+0	5+∞	5-1	5+∞
2	5	2+∞	2+∞	2+∞	2+0	2+∞	2+3
4	6	4+∞	4+∞	4+∞	4+∞	4+0	4+3
7	7	7+∞	7+∞	7+∞	7+∞	7+∞	7+0
min=		1	3	5	0	4	5

when k=5.

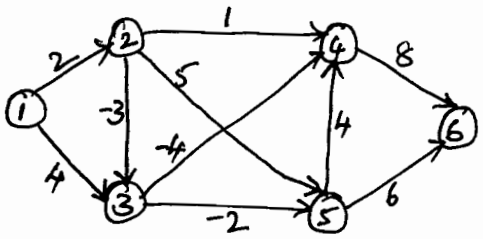
d	i	1	3	5	0	4	5
0	1	0+6	0+5	0+5	0+∞	0+∞	0+∞
1	2	1+0	1+∞	1+∞	1-1	1+∞	1+∞
3	3	3-2	3+0	3+∞	3+1	3+∞	3+∞
5	4	5+∞	5-2	5+0	5+∞	5-1	5+∞
0	5	0+∞	0+∞	0+∞	0+0	0+∞	0+3
4	6	4+∞	4+∞	4+∞	4+∞	4+0	4+3
5	7	5+∞	5+∞	5+∞	5+∞	5+∞	5+0
min=		1	3	5	0	4	3

when k=6

d	i	1	3	5	0	4	3
0	1	0+6	0+5	0+5	0+∞	0+∞	0+∞
1	2	1+0	1+∞	1+∞	1-1	1+∞	1+∞
3	3	3-2	3+0	3+∞	3+1	3+∞	3+∞
5	4	5+∞	5-2	5-0	5+∞	5-1	5+∞
0	5	0+∞	0+∞	0+∞	0+0	0+∞	0+3
4	6	4+∞	4+∞	4+∞	4+∞	4+0	4+3
3	7	3+∞	3+∞	3+∞	3+∞	3+∞	3+0
min =		1	3	5	0	4	3

∴ the final distance from ① to all other vertices are 1, 3, 5, 0, 4, 3 respectively.

Ex2: Find the shortest path from node 1 to every other node in the graph following graph using Bellman-Ford algorithm.



Soln

	1	2	3	4	5	6
initial d=0	2	4	∞	∞	∞	
k=2, d=0	2	-1	0	2	∞	
k=3, d=0	2	-1	-5	-3	8	
k=4, d=0	2	-1	-5	-3	3	
k=5, d=0	2	-1	-5	-3	3	

The knapsack problem and Memory Function.

→ The Top-down approach finds a solution to the recurrence leading to algorithm that solves common subproblems more than once and hence is inefficient.

→ The classic dynamic programming approach works bottom-up, it fills a table with soln to all smaller subproblem, but each of them is solved only once.

→ The goal is to get a method that solves only subproblems which are necessary & does it only once.

→ The recurrence for knapsack with memory function is

$$V[i,j] = \begin{cases} \max\{V[i-1,j],\ V[i-1,j-w_i]+P_i & \text{if } j-w_i \geq 0 \quad -(1) \\ V[i-1,j] & \text{if } j-w_i < 0 \quad -(2) \end{cases}$$

with its initial conditions

$V[0,j]=0$ for $j \geq 0$

&

$V[i,0]=0$ for $i \geq 0$

→ The parameters used in the algorithm

n : no: of objects / items

m : Capacity of the knapsack

w : is an array consisting of weights of all objects

P : is an array consisting of profits of all objects

V : is 2-D array of size n rows & m columns

→ Call the knapsack recursive function with $i=n$ & $j=m$

→ The algorithm is given in the next page.

Algorithm MFknapsack (i, j)
// Implement the memory function method for the knapsack problem.
// I/p: A non-ve int i indicating the number of the first items being considered & a non-ve int j indicating the knapsack's capacity.
// o/p: The value of an optimal feasible subset of the first i items.
// Note: Uses a global variable I/p array weights[], values[1..n] and table V[0..n, 0..W] whose entries are initialized with -1's except the row 0 & column 0 initialized with 0's

```
if V[i,j] < 0
    if j < weights[i]
        value ← MFknapsack(i-1, j)
    Else
        value ← max(MFknapsack(i-1, j),
                values[i] + MFknapsack(i-1, j-weights[i]))
    V[i,j] ← value
return V[i,j]
```

Problem1: Apply the memory function method to solve the following instance of the knapsack problem with capacity m = 5

Item	weight	value
1	2	₹12
2	1	10
3	3	20
4	2	15

Soln The following data is given

$n=4$, $m=5$, $(w_1\ w_2\ w_3\ w_4)=(2,1,3,2)$, $(P_1\ P_2\ P_3\ P_4)=(12,10,20,15)$

Step1 : Compute $V[4,5]$: $i=4$, $P_4=15$, $w_4=2$, $j=5$

Since $w_4<j$ use Equation ① from the recurrence

$V[4,5] = \max\{V[3,5], V[3,5-2]+15\}$

$= \max\{V[3,5], V[3,3]+15\}$ ∴ Compute $V[3,5]$ & $V[3,3]$

Step2 : Compute $V[3,5]$: $i=3$, $P_3=20$, $w_3=3$, $j=5$

Since $w_3<j$ use Equation ①

$V[3,5] = \max\{V[2,5], V[2,5-3]+20\}$

$= \max\{V[2,5], V[2,2]+20\}$ ∴ Compute $V[2,5]$ & $V[2,2]$

Step3 : Compute $V[3,3]$: $i=3$, $P_3=20$, $w_3=3$, $j=3$

Since $w_3<j$ use Equation ①

$V[3,3] = \max\{V[2,3], V[2,3-3]+20\}$

$= \max\{V[2,3], V[2,0]+20\}$ ∴ Compute $V[2,3]$ & $V[2,0]$

Step4 : Compute $V[2,5]$

follow the procedure for the consecutive steps ↵

∴ Compute $V[1,5]$ & $V[1,4]$

Step5 : Compute $V[2,2]$

Compute $V[1,2]$ & $V[1,1]$

Step6 : Compute $V[2,3]$

$= \max\{V[1,3], V[1,2]+10\}$

Compute $V[1,3]$ & $V[1,2]$

Step7 : Compute $V[2,0]$ But $V[2,0]=0$

Step8 : Compute $V[1,5]$

$= \max\{V[0,5], V[0,3]+12\} = 12$

Step9 : Compute $V[1,4]=12$

Step10 : Compute $V[1,2]=12$

Step11 : Compute $V[1,3]=12$

Step12 : $V[1,1]=V[0,1]=0$

Step13 : $V[2,3]=22$

Step14 : $V[2,2]=12$

Step15 : $V[2,5]=22$

Step16 : $V[3,3]=22$

Step17 : $V[3,5]=32$

Step18 : $V[4,5]=37$

$j \longrightarrow m$.

i ↓	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	-	12	22	-	22
3	0	-	-	22	-	32
n=4 4	0	-	-	-	-	(37) — optimal solution.

prob2 Solve the following instance of the knapsack problem with capacity $m=8$ & $(w_1, w_2, w_3, w_4) = (1, 5, 3, 4)$, $(P_1, P_2, P_3, P_4) = (15, 10, 9, 5)$

Soln

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	15	15	15	15	15	15	15	15
2	0	15	15	15	15	15	25	25	25
3	0	15	15	15	24	24	25	25	25
4	0	15	15	15	24	24	25	25	(29) — optimal soln.

Note: without memory func :-

prob3 Solve the following instance of the knapsack problem with capacity $m=5$, $n=4$, $(w, P) = (2,3)\ (3,4)\ (4,5)\ (5,6)$

Soln

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	(7) — optimal soln

Travelling Salesperson problem. (TSP)

→ Let's consider the source as 1, so the tour starts from 1 & ends at 1.

→ Various points to remember are

① The tour may start from any of the Edge $(1,k)$ where $k \in \underbrace{V-\{1\}}_{\text{all nodes except Vertex 1}}$ (Source Vertex)

② There is a path from vertex k to 1 & it goes through each vertex in $V-\{1,k\}$ (nodes b/w k & 1)

③ Let $g(i,S)$ be the cost of a shortest path starting at vertex i, going through all vertices in S & terminating at Vertex 1.

④ The function $g(1, V-\{1\})$ is the least cost of an optimal salesperson tour. From the principle of optimality we can write

$$g(1, V-\{1\}) = \min_{2 \le k \le n} \{c_{1k} + g(k, V\{1,k\})\}$$

when $|V-\{1\}| = n-1$.

*Note: The relation gives the min cost of the tour from Source Vertex 1.

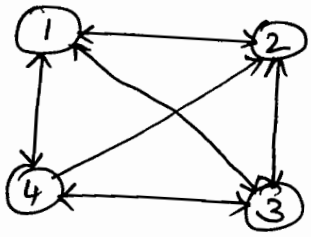
⑤ The above relation can be generalized as

$$g(i,S) = \min_{j \in S} \{c_{ik} + g(j, S-\{j\})\} \text{ where } |S| < n-1, \; i \neq 1, \; i \notin S, \; 1 \notin S$$

⑥ The base case is given by the following relation

$$g(i,\phi) = c_{i1} \text{ for } 1 \le i \le n \text{ and } i \neq 1 \text{ when } |S| = 0$$

Exe:1 Solve the following TSP which is represented as directed graph & whose edge lengths are given by cost adjacency matrix.



	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

Soln

Step1: $S = \{\phi\}$, so, $|S| = 0$ In this case let us solve the following relation

$$g(i, \phi) = c_{i1} \quad \text{where } 1 \le i \le n \text{ \& } i \neq 1$$

$g(2, \phi) = c_{21} = 5$
$g(3, \phi) = c_{31} = 6$
$g(4, \phi) = c_{41} = 8$

Step2: $|s| = 1$, In this case $i \neq 1, 1 \notin S, i \notin S$ solve the recurrence $g(i, s) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\}$

when $i = 2$:

$g(2, \{3\}) = \min_{j \in \{3\}} \{c_{23} + g(3, \phi)\} = \min\{9 + 6\} = 15$
$g(2, \{4\}) = \min_{j \in \{4\}} \{c_{24} + g(4, \phi)\} = \min\{10 + 8\} = 18$

when $i = 3$:

$g(3, \{2\}) = \min_{j \in \{2\}} \{c_{32} + g(2, \phi)\} = \min\{13 + 5\} = 18$
$g(3, \{4\}) = \min_{j \in \{4\}} \{c_{34} + g(4, \phi)\} = \min\{12 + 8\} = 20$

when $i = 4$:

$g(4, \{2\}) = \min_{j \in \{2\}} \{c_{42} + g(2, \phi)\} = \min\{8 + 5\} = 13$
$g(4, \{3\}) = \min_{j \in \{3\}} \{c_{43} + g(3, \phi)\} = \min\{9 + 6\} = 15$

Step3: $|s| = 2$. In this case also $i \neq 1, 1 \notin S, i \notin S$ solve the recurrence $g(i, s) = \min_{j \in S} \{c_{ij} + g(j, s - \{j\})\}$

when $i=2$: $g(2,\{3,4\}) = \min_{j\in\{3,4\}} \{c_{ij} + g(j, s-\{j\})\}$

$= \min\{c_{23} + g(3,4), c_{24} + g(4,3)\}$

$= \min\{9+20, 10+15\}$

$= \min\{29, 25\} = 25$

when $i=3$: $g(3,\{2,4\}) = \min_{j\in\{2,4\}} \{c_{ij} + g(j, s-\{j\})\}$

$= \min\{c_{32} + g(2,4), c_{34} + g(4,2)\}$

$= \min\{13+18, 12+13\}$

$= \min\{31, 25\} = 25$

when $i=4$: $g(4,\{2,3\}) = \min_{j\in\{2,3\}} \{c_{ij} + g(j, s-\{j\})\}$

$= \min\{c_{4,2} + g(2,3), c_{4,3} + g(3,2)\}$

$= \min\{8+15, 9+18\}$

$= \min\{23, 27\} = 23$

Step 4

Since $|s| = n-1$ i.e. we should find

$$g(1, V-\{1\}) = \min_{2\le k\le n} \{c_{1k} + g(k, V-\{1,k\})\}$$

$$g(1,\{1,2,3,4\}-\{1\}) = \min_{k=2,3,4} \{c_{1k} + g(k,\{1,2,3,4\}-\{1,k\})\}$$

$= \min\{c_{12} + g(2,\{3,4\}),$
$\quad c_{13} + g(3,\{2,4\}),$
$\quad c_{14} + g(4,\{2,3\})\}$

$= \min\{10+25, 15+25, 20+23\}$

$= \min\{35, 40, 43\}$

$\therefore g(1,\{2,3,4\}) = \underline{35}$ is the cost of optimal tour.

$g(1,\{2,3,4\}) = 35$

$g(1,\{2,3,4\}) = c_{12} + g(2,\{3,4\})$

$\Downarrow$

$c_{24} + g(4,\{3\})$

$\Downarrow$

$c_{43} + g(3,\phi)$

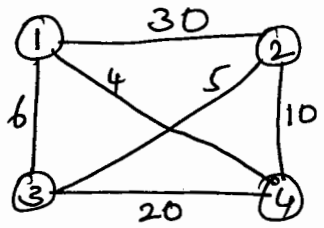
$\Downarrow$

c_{31}

optimal path : $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1$

The cost of Tour : $10 + 10 + 9 + 6 = 35$.

Problem:2 Solve the following TSP which is represented as a graph.



Soln $g(1,\{2,3,4\}) = 25$

$g(1,\{2,3,4\}) = c_{13} + g(3,\{2,4\})$

$\Downarrow$

$c_{32} + g(2,\{4\})$

$\Downarrow$

$c_{24} + g(4,\phi)$

$\Downarrow$

c_{41}

$1 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 1$

$6 \quad 5 \quad 10 \quad 4 = 25$.

Decrease-AND-Conquer Approaches
Space-Time Tradeoffs:
unit-5.

Syllabus: Decrease-and-Conquer Approaches: Introduction, Insertion sort, Depth First Search (DFS), Breadth First Search (BFS), Topological Sorting Space-Time Tradeoffs: Introduction, Sorting by Counting, I/p enhancement in String Matching.

Introduction

→ The Decrease-and-Conquer (D-&-C) technique is based on exploiting the relationship b/w a solution to a given instance of a problem & a solution to a smaller instance of the same Problem.

→ There are Three major variations of D-&-C

① Decrease by a Constant: The size of an instance is reduced by the same constant on each iteration of the algorithm.

Ex Computing a^n in this method

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

② Decrease-by-Constant factor: is a technique suggests reducing a problem instance by the same constant factor on each iteration of algorithm

Ex Computing a^n

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

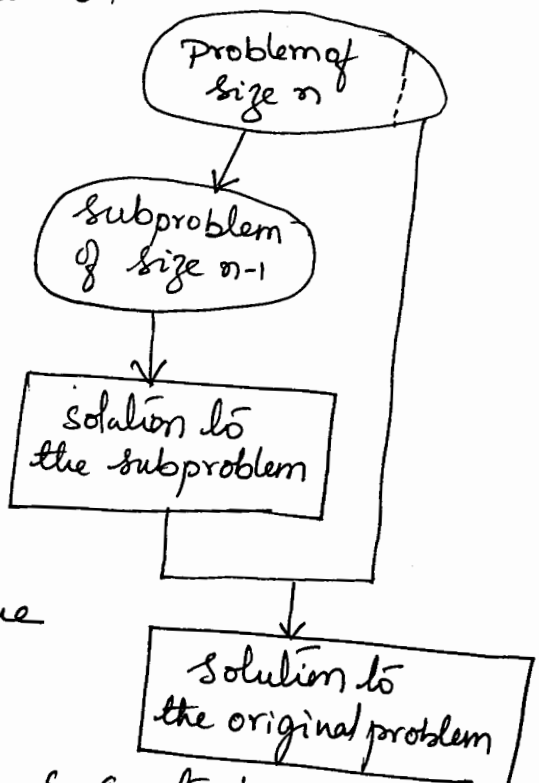


fig: D(by one)-&-C technique

③ Variable size decrease:

Variety of D&C, a size reduction pattern varies from one iteration of an algorithm to another.

Ex Euclid's algorithm to calculate gcd

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

Note: a^n can be calculated using divide & conquer as

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

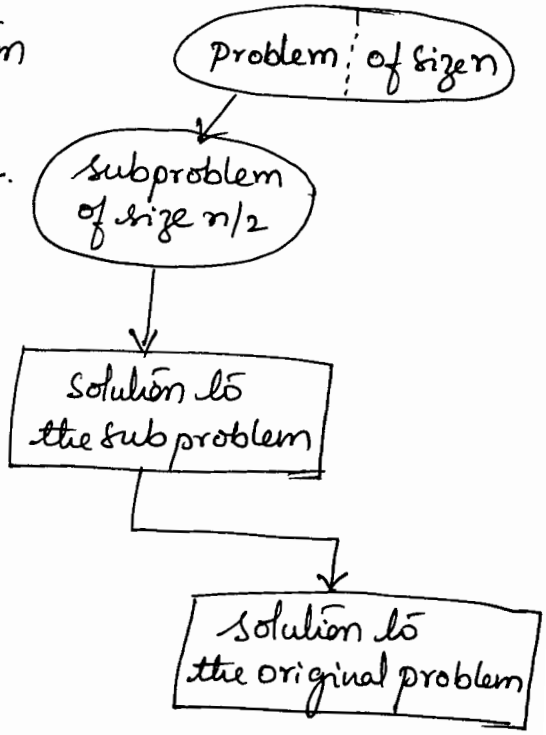


Fig: Decrease (by half) - & - Conquer technique.

Inserlion Sort.

Algorithm Inserlion Sort (A[0..n-1])
// sorts a given array be Inserlion Sort.
// I/P: An array A[0..n-1] of n orderable elements
// O/P: Array A[0..n-1] sorted in ascending 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
```

Example

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
29 45 68 89 90 | 34 17
29 34 45 68 89 90 | 17
17 29 34 45 68 89 90

Algorithm Analysis

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

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

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

Depth-First Search.

→ There are two principal algorithms for doing traversal on graphs: depth-first-search (DFS)
Breadth-First-Search (BFS)

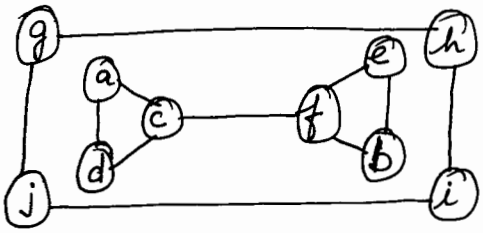


fig: graph.

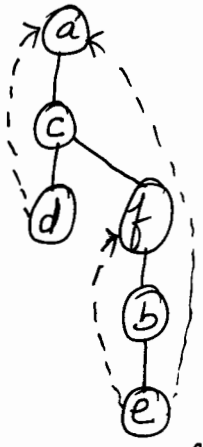
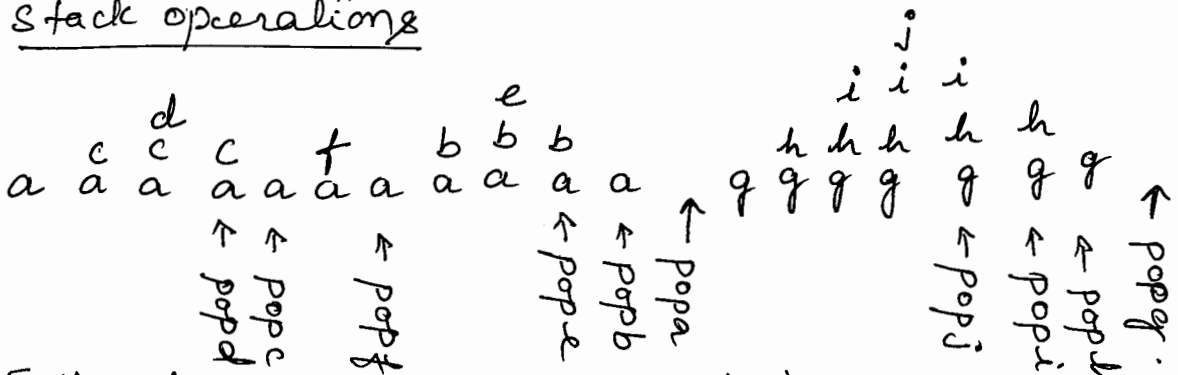


fig: DFS forest with solid line tree edge & dashed line the back edge.

$e_{6,2}$
$b_{5,3}$ $j_{10,7}$
$d_{3,1}$ $f_{4,4}$ $i_{9,8}$
$c_{2,5}$ $h_{8,9}$
$a_{1,6}$ $g_{7,10}$

[Traversal stack, 1st subscript indicates order in which a vertex was visited, 2nd indicates order in which it became dead end]

Stack operations



[if not pop operation then it is push]

```
Algorithm DFS(G)
Mark each vertex in V as 0 //initialize
count ← 0
for each vertex v in V do
    if v is marked with 0
        dfs(v)
```

```
Function dfs(v)
    count ← count + 1
    mark v with count
    for each w in V adjacent to v
        if w is marked 0
            dfs(w)
```

Traversal time Efficiency.

① For Adjacency Matrix representation it is $O(|V|^2)$
② For linked list representation it is $O(|V| + |E|)$
(|V| = vertices, |E| = edges)

Breadth-First-Search (BFS)

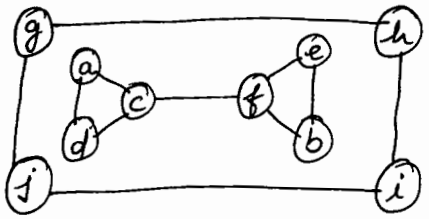
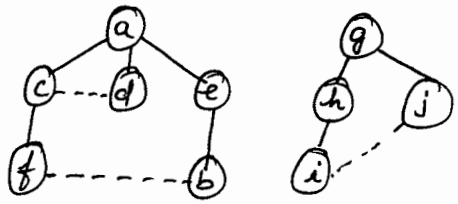


Fig: Graph

$a_1 c_2 d_3 e_4 f_5 b_6$
$g_7 h_8 i_9 j_{10}$

Traversal Queue



BFS forest
→ tree edges
→ dotted lines are cross edges

Algorithm BFS(G)
// Implements a BFS traversal of a given graph
// I/P: Graph $G = \langle V, E \rangle$
// O/P: Graph G with its vertices marked with consecutive integers mark each vertex in V with 0 as 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 & assigns them the numbers in the
// order they are visited via global variable count
count ← count + 1;
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.

Algorithm Efficiency

→ for adjacency matrix representation it is $\Theta(|V|^2)$
→ for linked list representation it is $\Theta(|V| + |E|)$

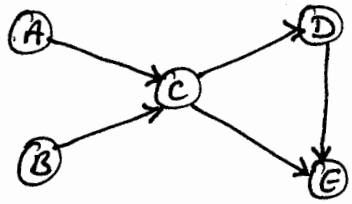
Facts about DSF & BSF

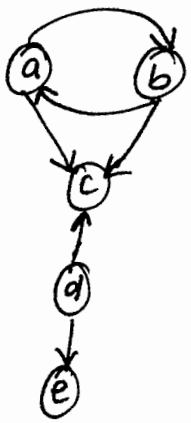
	DFS	BFS
Data structure	stack	queue
No of vertex orderings	2 orderings	1 orderings
Edge types (undirected graphs)	tree & back edges	tree & Cross Edge
Applications	connectivity, acyclicity	connectivity
	articulation points	acyclicity
		min-edge path
Efficiency for adjacency matrix	$\Theta(\|V\|^2)$	$\Theta(\|V^2\|)$
Efficiency for adjacency list	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\| + \|E\|)$

Topological Sorting Algorithm.

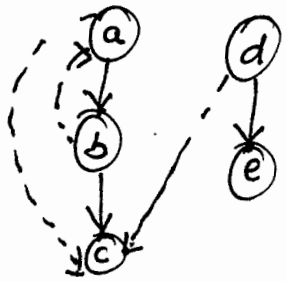
→ if we have a directed graph & then topological sorting/ordering of the graph is a linear order of the vertices of the graph such that if 'uv' is an edge in the graph then 'u' should come before "v" in the topological ordering.

Case study: Lets assume that a student has to take five courses/subjects before he gets a degree. & they can be represented as below dependency graph. In what order should he study the courses so that the dependencies are satisfied.





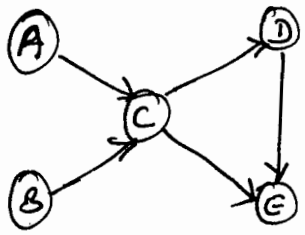
(a) Digraph



(b) DFS forest of the diagraph for the DFS traversal started at a.

→ If a DFS forest of a digraph has no back Edges, the digraph is a dag. [Directed ACyclic graph].

Algorithm 1: Topological Sorting using Depth First Search



STACK

E
D
C
A
B

{ Pop Elements E D C A B

Reading Backwards B A C D E

Algo (Wikipedia)

L ← Empty list that will contain the sorted nodes
S ← Set of all nodes with no outgoing edges
for each node n in S do
    visit(n)

function visit (node n)
    if n has not been visited yet then
        mark n as visited
        for each node m with an edge from m to n do
            visit(m)
        add n to L

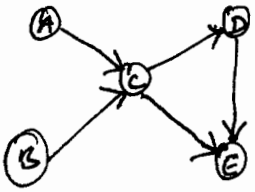
S E

L B A E D E

A V
B V
C V
D V
E V

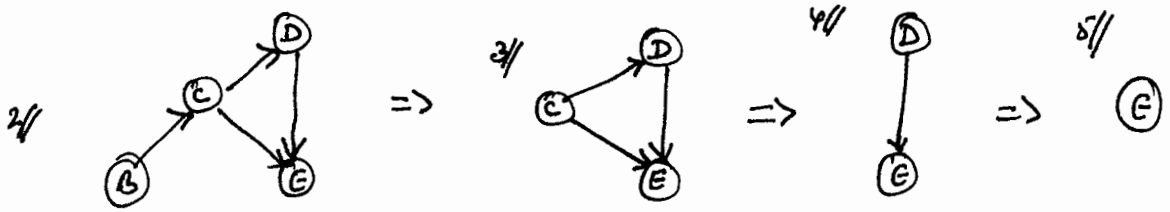
Start from E

Algorithm 2 — Topological Sorting using Source Removal



Remove the vertex & the corresponding Edge which doesn't have any incomming Edge.

sorted list (A) (B) (C) (D) (E) or (B) (A) (C) (D) (E)

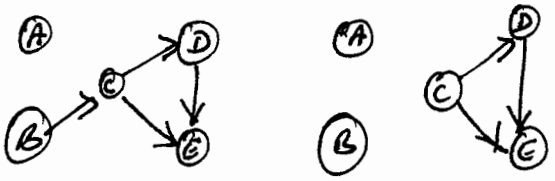


Algorithm

L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edges

while S is non-empty do
    remove a node n from S insert n into L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
if graph has edges then
    return error (graph has at least one cycle)
Else
    return L (a topologically sorted order)

S ~~A~~ B	S ~~A~~ ~~B~~	S D	S E
L A	L A B	L A B C	L A B C D E



Limitations of Algorithm Power

Objectives
We now move into the third and final major theme for this course.

1. *Tools* for analyzing algorithms.
2. *Design strategies* for designing algorithms.
3. Identifying and coping with the *limitations* of algorithms.

Efficiency of an algorithm

- By establishing the asymptotic efficiency class

Problem	Algorithm	Time efficiency
sorting	selection sort	$\Theta(n^2)$
Towers of Hanoi	recursive	$\Theta(2^n)$

- The efficiency class for selection sort (quadratic) is lower. Does this mean that selection sort is a "better" algorithm?
  - Like comparing "apples" to "oranges"
- By analyzing how efficient a particular algorithm is compared to other algorithms for the same problem
  - It is desirable to know the best possible efficiency any algorithm solving this problem may have – establishing a *lower bound*

Lower Bounds
Lower bound: an estimate on a minimum amount of work needed to solve a given problem
Examples:

- number of comparisons needed to find the largest element in a set of *n* numbers
- number of comparisons needed to sort an array of size *n*
- number of comparisons necessary for searching in a sorted array
- number of multiplications needed to multiply two *n*-by-*n* matrices

Lower bound can be

- an exact count
- an efficiency class (Ω)

- *Tight* lower bound: there exists an algorithm with the same efficiency as the lower bound

Problem	**Lower bound**		**Tightness**
sorting	Ω(*n*log *n*)	yes	
searching in a sorted array	Ω(log *n*)		yes
element uniqueness	Ω(*n*log *n*)		yes
n-digit integer multiplication	Ω(*n*)		unknown
multiplication of *n*-by-*n* matrices	Ω(*n*2)		unknown

Methods for Establishing Lower Bounds

- trivial lower bounds
- information-theoretic arguments (decision trees)

- adversary arguments
- problem reduction

Trivial Lower Bounds

Trivial lower bounds: based on counting the number of items that must be processed in input and generated as output

Examples

- finding max element
- polynomial evaluation
- sorting
- element uniqueness
- computing the product of two n-by-n matrices

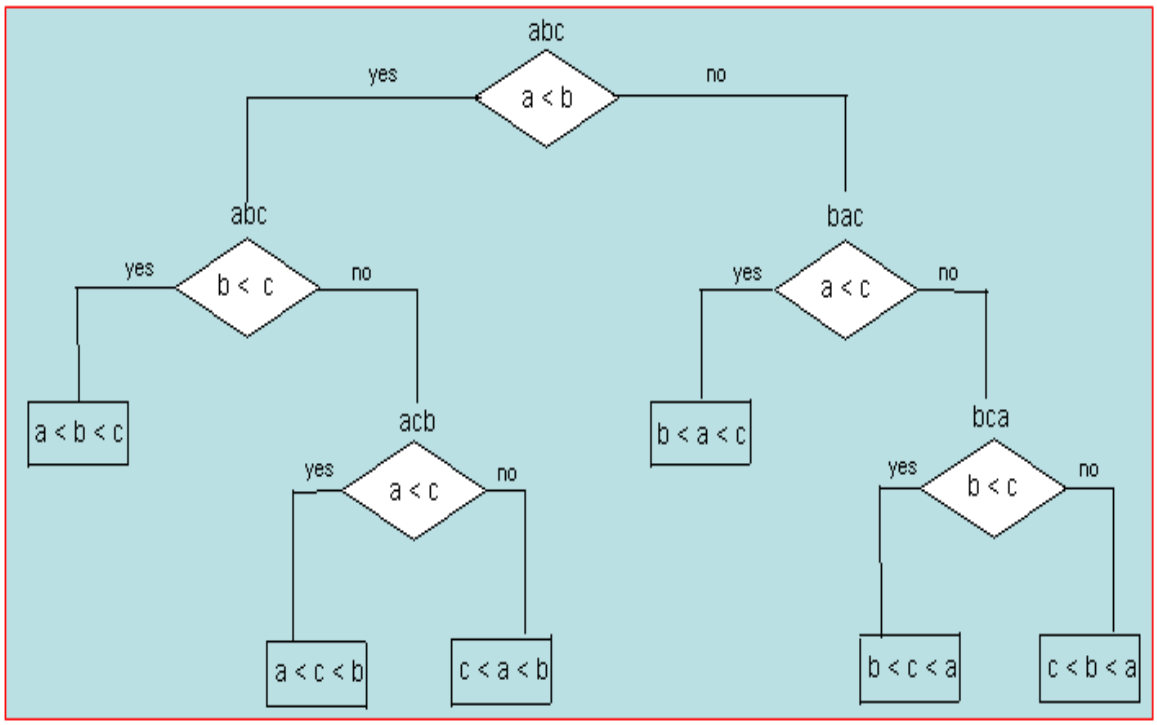
Conclusions

- may and may not be useful
- be careful in deciding how many elements must be processed

Decision Trees

Decision tree — a convenient model of algorithms involving Comparisons in which: (i) internal nodes represent comparisons (ii) leaves represent outcomes

Decision tree for 3-element insertion sort



Deriving a Lower Bound from Decision Trees

- How does such a tree help us find lower bounds?
  - There must be at least one leaf for each correct output.
  - The tree must be tall enough to have that many leaves.
- In a binary tree with l leaves and height h,

$$h \geq \lceil \log_2 l \rceil$$

Decision Tree and Sorting Algorithms

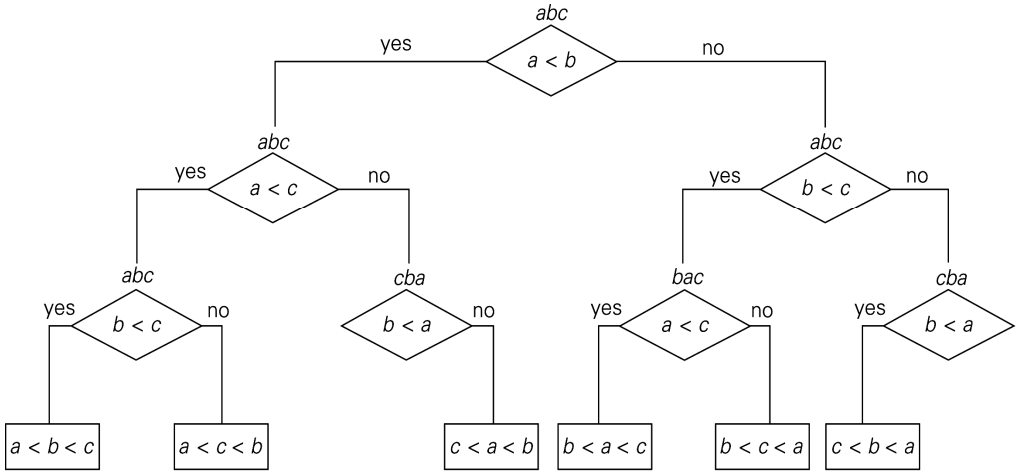


FIGURE 11.2 Decision tree for the three-element selection sort. A triple above a node indicates the state of the array being sorted. Note the two redundant comparisons $b < a$ with a single possible outcome because of the results of some previously made comparisons.

Decision Tree and Sorting Algorithms

- Number of leaves (outcomes) $\geq n!$
- Height of binary tree with $n!$ leaves $\geq \lceil \log_2 n! \rceil$
- Minimum number of comparisons in the worst case $\geq \lceil \log_2 n! \rceil$ for any comparison-based sorting algorithm
- $\lceil \log_2 n! \rceil \approx n \log_2 n$
- This lower bound is tight (e.g. mergesort)

Decision Tree and Searching a Sorted Array

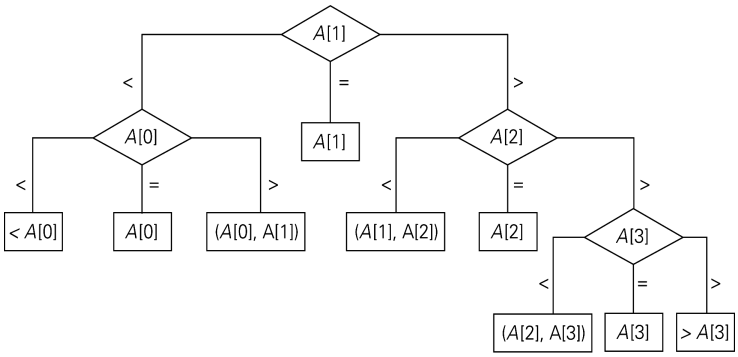


FIGURE 11.4 Ternary decision tree for binary search in a four-element array

Decision Tree and Searching a Sorted Array

- Number of leaves (outcomes) = $n + n+1 = 2n+1$
- Height of ternary tree with $2n+1$ leaves $\geq \lceil \log_3 (2n+1) \rceil$
- This lower bound is NOT tight (the number of worst-case comparisons for binary search is $\lceil \log_2 (n+1) \rceil$, and $\lceil \log_3 (2n+1) \rceil \leq \lceil \log_2 (n+1) \rceil$)
- Can we find a better lower bound or find an algorithm with better efficiency than binary search?

Decision Tree and Searching a Sorted Array

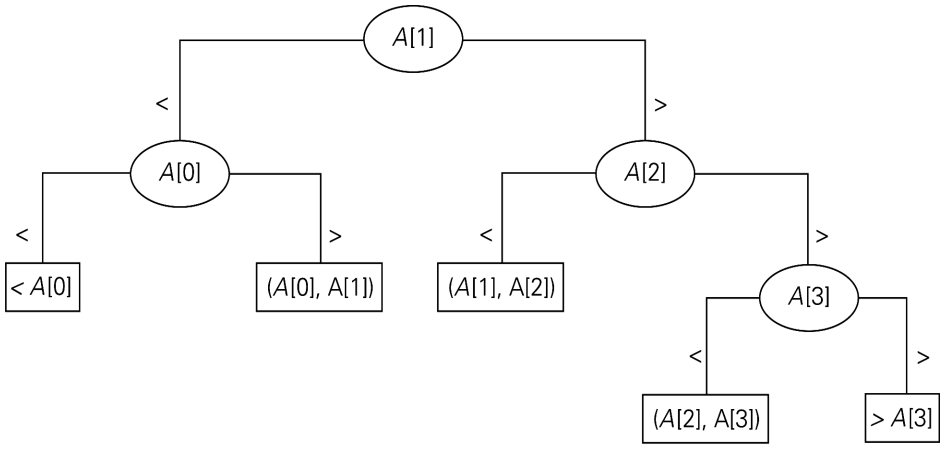
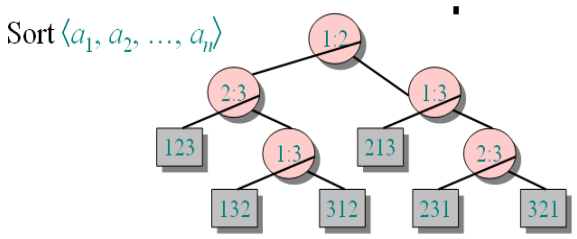


FIGURE 11.5 Binary decision tree for binary search in a four-element array

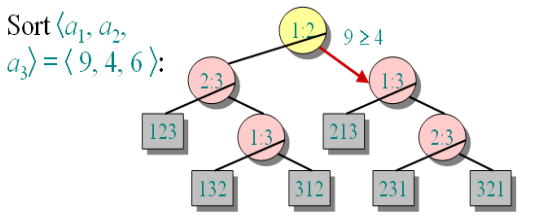
Decision Tree and Searching a Sorted Array

- Consider using a binary tree where internal nodes also serve as successful searches and leaves only represent unsuccessful searches
- Number of leaves (outcomes) = $n + 1$
- Height of binary tree with $n+1$ leaves $\geq \lceil \log_2 (n+1) \rceil$
- This lower bound is tight

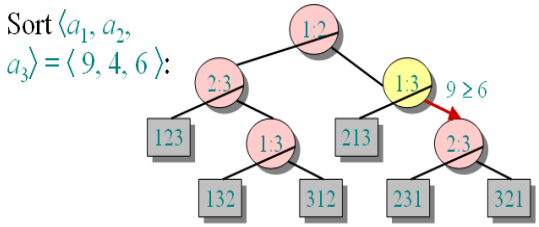
Decision-tree example



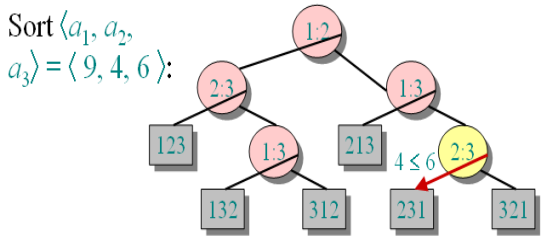
Each internal node is labeled $i:j$ for $i, j \in \{1, 2, \ldots, n\}$.
- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.



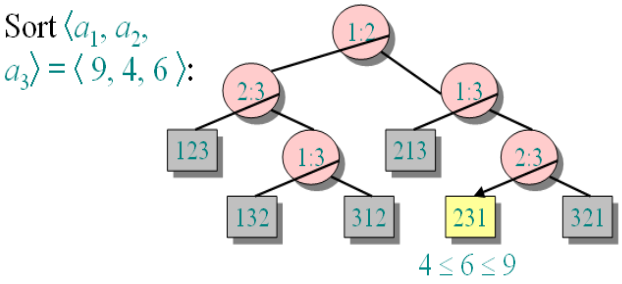
Each internal node is labeled $i:j$ for $i, j \in \{1, 2, \ldots, n\}$.
- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.



Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.
- The left subtree shows subsequent comparisons if $a_i \le a_j$.
- The right subtree shows subsequent comparisons if $a_i \ge a_j$.



Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.
- The left subtree shows subsequent comparisons if $a_i \le a_j$.
- The right subtree shows subsequent comparisons if $a_i \ge a_j$.



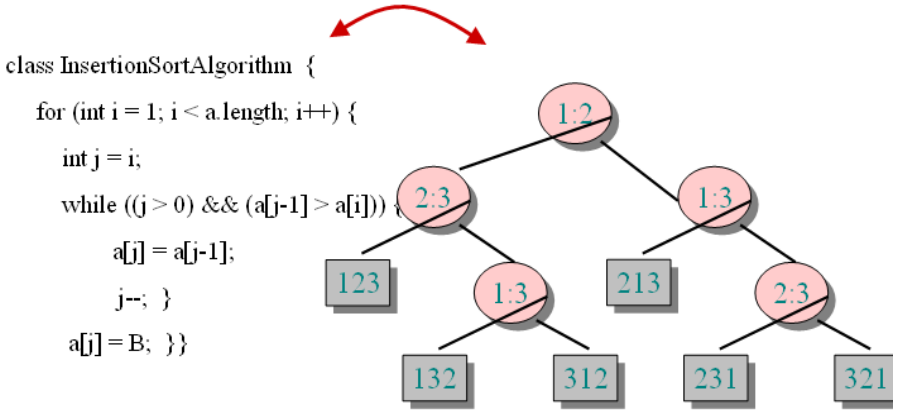
Each leaf contains a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ to indicate that the ordering $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$ has been established.

Decision-tree model

A decision tree can model the execution of any comparison sort:

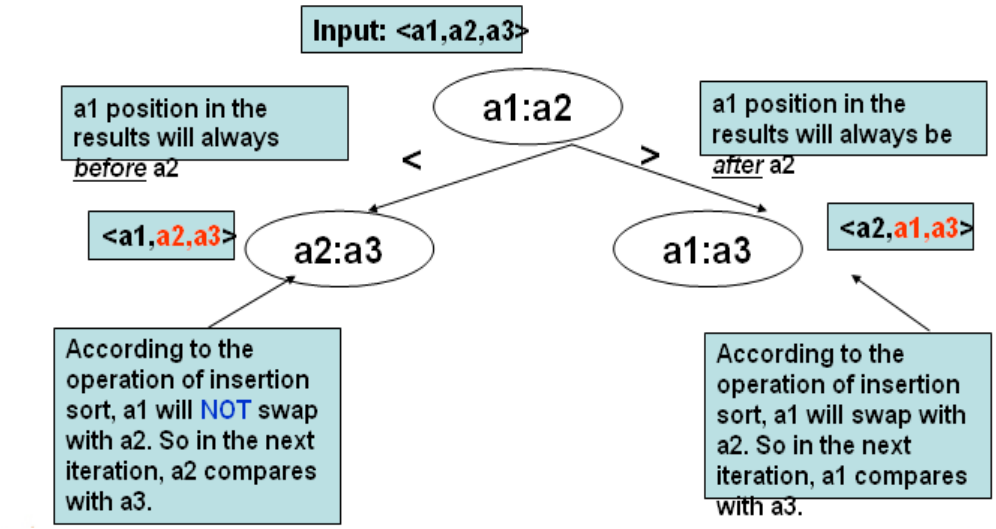
- One tree for each input size n.
- View the algorithm as splitting whenever it compares two elements.
- The tree contains the comparisons along all possible instruction traces.
- The running time of the algorithm = the length of the path taken.
- Worst-case running time = height of tree.

Any comparison sort can be turned into a Decision tree

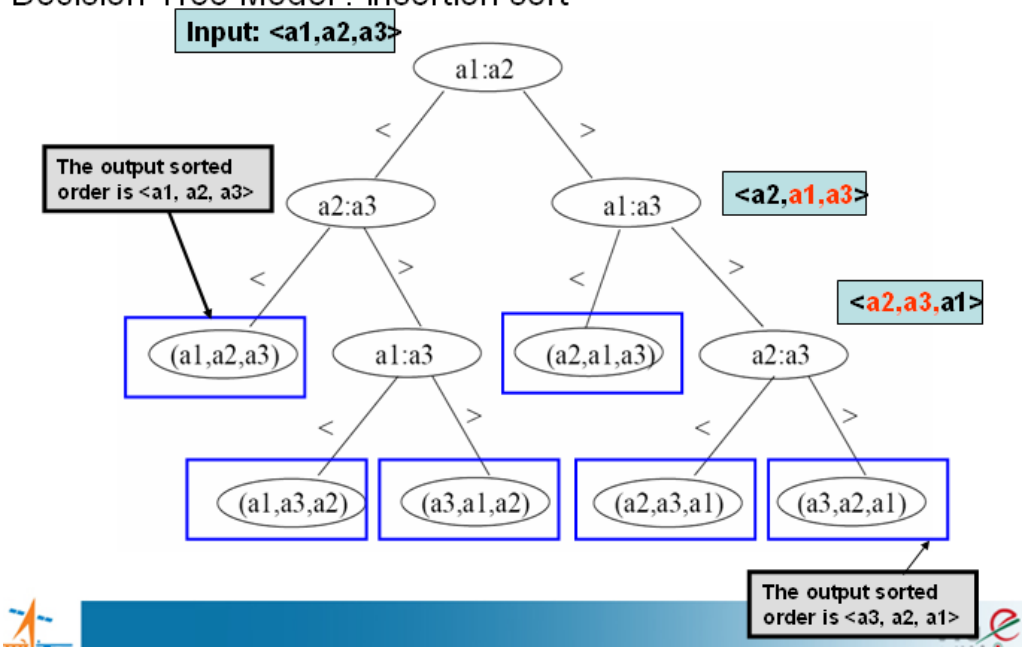


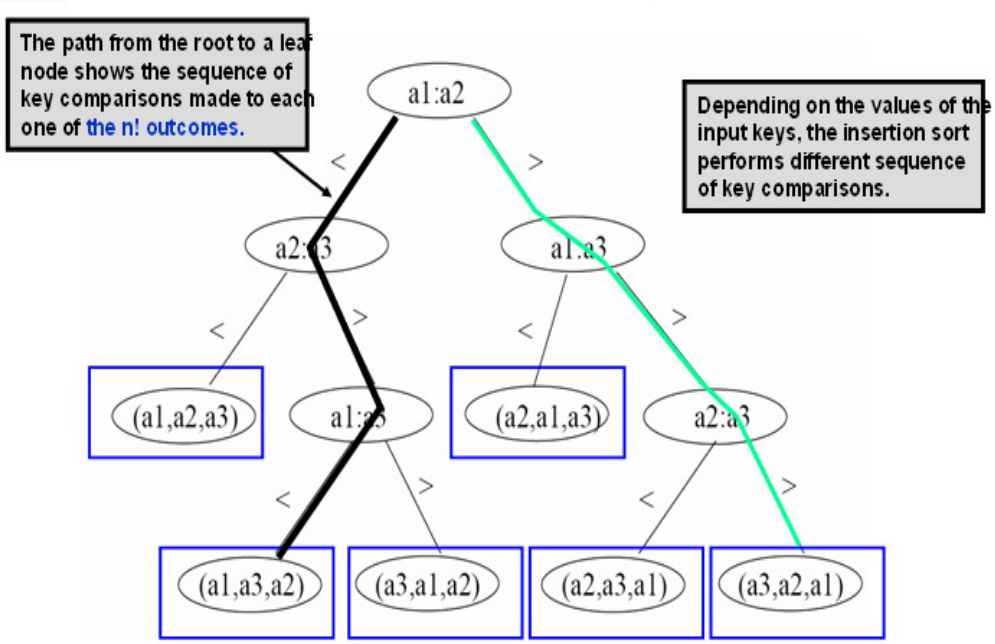
Decision Tree Model : insertion sort

- For example, we trace the sequence of key comparisons of insertion sort on <a1, a2, a3> by a decision tree.



Decision Tree Model : insertion sort





Decision Tree Model

- In the insertion sort example, the decision tree reveals all possible key comparison sequences for 3 distinct numbers.
- There are exactly 3!=6 possible output sequences.
- Different comparison sorts should generate different decision trees.
- It should be clear that, in theory, we should be able to draw a decision tree for ANY comparison sort algorithm.
- Given a particular input sequence, the path from root to the leaf path traces a particular key comparison sequence performed by that comparison sort.
  - The length of that path represented the number of key comparisons performed by the sorting algorithm.
- When we come to a leaf, the sorting algorithm has determined the sorted order.
- Notice that a correct sorting algorithm should be able to sort EVERY possible output sorted order.
- Since, there are n! possible sorted order, there are n! leaves in the decision tree.
- Given a decision tree, the height of the tree represent the longest length of a root to leaf path.
- It follows the height of the decision tree represents the largest number of key comparisons, which is the worst-case running time of the sorting algorithm.

"Any comparison based sorting algorithm takes $\Omega(n \log n)$ to sort a list of n distinct elements in the worst-case."

- any comparison sort ← model by a decision tree
- worst-case running time ← the height of decision tree

"Any comparison based sorting algorithm takes $\Omega(n \log n)$ to sort a list of n distinct elements in the worst-case."

- We want to find a lower bound (Ω) of the **height** of a binary tree that has n! Leaves.
  ◊ What is the minimum height of a binary tree that has n! leaves?

- The binary tree must be a complete tree (recall the definition of complete tree).
- Hence the minimum (lower bound) height is θ(log2(n!)).

- $\log_2(n!)$
  $= \log_2(n) + \log_2(n-1) + \ldots + \log_2(n/2) + \ldots$.
  $\geq n/2 \log_2(n/2) = n/2 \log_2(n) - n/2$
  So, $\log_2(n!) = \Omega(\boldsymbol{n}\ \boldsymbol{\log n})$.
- It follows the height of a binary tree which has n! leaves is at least $\Omega(\boldsymbol{n}\ \boldsymbol{\log n})$ ◊ worst-case running time is at least $\Omega(\boldsymbol{n}\ \boldsymbol{\log n})$
- Putting everything together, we have
  "Any comparison based sorting algorithm takes $\Omega(\boldsymbol{n}\ \boldsymbol{\log n})$ to sort a list of n distinct elements in the worst-case."

Adversary Arguments
Adversary argument: a method of proving a lower bound by playing role of adversary that makes algorithm work the hardest by adjusting input
Example: "Guessing" a number between 1 and *n* with yes/no questions
Adversary: Puts the number in a larger of the two subsets generated by last question

Lower Bounds by Problem Reduction
Idea: If problem *P* is at least as hard as problem *Q*, then a lower bound for *Q* is also a lower bound for *P*.
Hence, find problem *Q* with a known lower bound that can be reduced to problem *P* in question.

Example: Euclidean MST problem
- Given a set of *n* points in the plane, construct a tree with minimum total length that connects the given points. (considered as problem *P*)
- To get a lower bound for this problem, reduce the *element uniqueness problem* to it. (considered as problem *Q*)
- If an algorithm faster than *n* log *n* exists for Euclidean MST, then one exists for element uniqueness also. Aha! A contradiction! Therefore, any algorithm for Euclidean MST must take $\Omega(n \log n)$ time.

Classifying Problem Complexity
Is the problem *tractable*, i.e., is there a polynomial-time ($O(p(n))$ algorithm that solves it?
Possible answers:
- yes (give examples)
- no
  - because it's been proved that no algorithm exists at all (e.g., Turing's *halting problem*)
  - because it's been proved that any algorithm for the problem takes exponential time
- unknown

Problem Types: Optimization and Decision

- *Optimization problem*: find a solution that maximizes or minimizes some objective function
- *Decision problem*: answer yes/no to a question
  Many problems have decision and optimization versions.
  E.g.: traveling salesman problem
- *optimization*: find Hamiltonian cycle of minimum length
- *decision*: find Hamiltonian cycle of length ≤ m

Decision problems are more convenient for formal investigation of their complexity.

Class *P*

P: the class of decision problems that are solvable in O($p(n)$) time, where $p(n)$ is a polynomial of problem's input size n

Examples:

- searching
- element uniqueness
- graph connectivity
- graph acyclicity
- primality testing (finally proved in 2002)

Class *NP*

NP (*nondeterministic polynomial*): class of decision problems whose proposed solutions can be verified in polynomial time = solvable by a *nondeterministic polynomial algorithm*

A *nondeterministic polynomial algorithm* is an abstract two-stage procedure that:

- generates a random string purported to solve the problem
- checks whether this solution is correct in polynomial time

By definition, it solves the problem if it's capable of generating and verifying a solution on one of its tries

Why this definition?

- led to development of the rich theory called "computational complexity"

Example: CNF satisfiability

Problem: Is a boolean expression in its conjunctive normal form (CNF) satisfiable, i.e., are there values of its variables that makes it true?

This problem is in *NP*. Nondeterministic algorithm:

- Guess truth assignment
- Substitute the values into the CNF formula to see if it evaluates to true

Example: (A | ¬B | ¬C) & (A | B) & (¬B | ¬D | E) & (¬D | ¬E)

Truth assignments:

A B C D E

0 0 0 0 0

. . .

1 1 1 1 1

Checking phase: O(n)

What problems are in *NP*?

- Hamiltonian circuit existence
- Partition problem: Is it possible to partition a set of n integers into two disjoint subsets with the same sum?
- Decision versions of TSP, knapsack problem, graph coloring, and many other combinatorial optimization problems. (Few exceptions include: MST, shortest paths)
- All the problems in *P* can also be solved in this manner (but no guessing is necessary), so we have:

  $P \subseteq NP$
- Big question: $P = NP$?

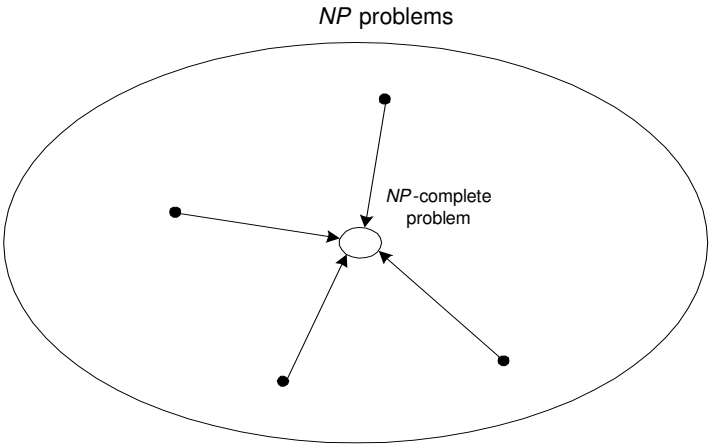
***P = NP* ?**

- One of the most important unsolved problems is computer science is whether or not *P=NP*.
  - If *P=NP*, then a ridiculous number of problems currently believed to be very difficult will turn out have efficient algorithms.
  - If *P≠NP*, then those problems definitely do not have polynomial time solutions.
- Most computer scientists suspect that P ≠ NP. These suspicions are based partly on the idea of NP-completeness.

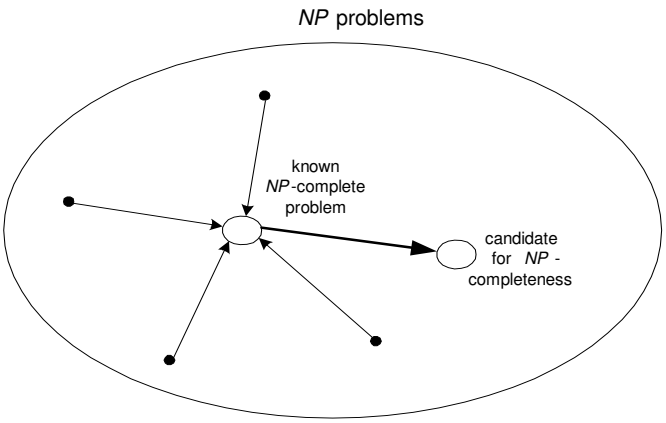
***NP*-Complete Problems**

A decision problem *D* is *NP*-complete if it's as hard as any problem in *NP*, i.e.,

- *D* is in *NP*
- every problem in *NP* is polynomial-time reducible to *D*



Other *NP*-complete problems obtained through polynomial-time reductions from a known *NP*-complete problem



Examples: TSP, knapsack, partition, graph-coloring and hundreds of other problems of combinatorial nature

General Definitions: P, NP, NP-hard, NP-easy, and NP-complete

- Problems
  - Decision problems (yes/no)
  - Optimization problems (solution with best score)
- P
  - Decision problems (decision problems) that can be solved in polynomial time
  - can be solved "efficiently"
- NP
  - Decision problems whose "*YES*" answer can be verified in polynomial time, if we already have the *proof* (or *witness*)
- co-NP
  - Decision problems whose "*NO*" answer can be verified in polynomial time, if we already have the *proof* (or *witness*)
- e.g. The satisfiability problem (SAT)

- Given a boolean formula

$$\left(x_1 \vee x_2 \vee x_3 \vee x_4\right) \wedge \left(x_5 \vee x_6 \vee x_7\right) \wedge x_8 \wedge \overline{x}_9$$

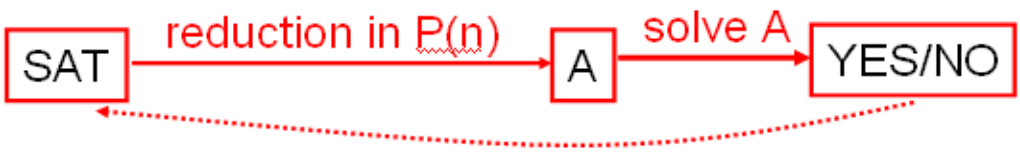
is it possible to assign the input x1...x9, so that the formula evaluates to TRUE?

- If the answer is YES with a proof (i.e. an assignment of input value), then we can check the proof in polynomial time (SAT is in NP)
- We may not be able to check the NO answer in polynomial time (Nobody really knows.)

- NP-hard
  - A problem is NP-hard iff an polynomial-time algorithm for it implies a polynomial-time algorithm for every problem in NP
  - NP-hard problems are at least *as hard as* NP problems
- NP-complete
  - A problem is NP-complete if it is NP-hard, and is an element of NP (NP-easy)
- Relationship between decision problems and optimization problems

- every optimization problem has a corresponding decision problem
- optimization: minimize x, subject to constraints
- yes/no: is there a solution, such that x is less than c?
- an optimization problem is NP-hard (NP-complete)
if its corresponding decision problem is NP-hard (NP-complete)

- How to know another problem, A, is NP-complete?
  - To prove that A is NP-complete, reduce a known NP-complete problem to A



- Requirement for Reduction
  - Polynomial time
  - YES to A also implies YES to SAT, while
    NO to A also implies No to SAT

Examples of NP-complete problems

Vertex cover

- Vertex cover
  - given a graph G=(V,E), find the *smallest* number of vertexes that cover *each edge*
  - Decision problem: is the graph has a vertex cover of size K?
- Independent set

- independent set: a set of vertices in the graph with no edges between *each pair* of nodes.
- given a graph G=(V,E), find the *largest independent set*
- reduction from vertex cover:
  - Set cover
    - given a universal set *U*, and several subsets *S1*,...*Sn*
    - find the least number of subsets that contains each elements in the universal set

Polynomial (P) Problems

- Are solvable in polynomial time
- Are solvable in O(nk), where k is some constant.
- Most of the algorithms we have covered are in P

Nondeterministic Polynomial (NP) Problems

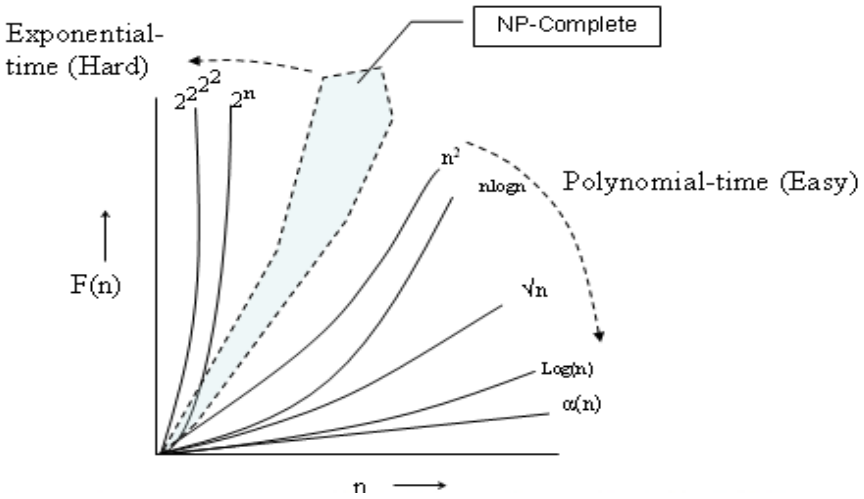
- This class of problems has solutions that are verifiable in polynomial time.
  - Thus any problem in P is also NP, since we would be able to solve it in polynomial time, we can also verify it in polynomial time

NP-Complete Problems

- Is an NP-Problem
- Is at least as difficult as an NP problem (is reducible to it)
- More formally, a decision problem C is NP-Complete if:

- C is in NP
- Any known NP-hard (or complete) problem ≤p C
- Thus a proof must show these two being satisfied

Exponential Time Algorithms



Examples

- Longest path problem: (similar to Shortest path problem, which requires polynomial time) suspected to require exponential time, since there is no known polynomial algorithm.
- Hamiltonian Cycle problem: Traverses all vertices exactly once and form a cycle.

Reduction

- P1 : is an unknown problem (easy/hard ?)
- P2 : is known to be difficult

If we can easily solve P2 using P1 as a subroutine then P1 is difficult
Must create the inputs for P1 in polynomial time.

* P1 is definitely difficult because you know you cannot solve P2 in polynomial time unless you use a component that is also difficult (it cannot be the mapping since the mapping is known to be polynomial)

Decision Problems

- Represent problem as a decision with a boolean output
  - Easier to solve when comparing to other problems
  - Hence all problems are converted to decision problems.

P = {all decision problems that can be solved in polynomial time}
NP = {all decision problems where a solution is proposed, can be verified in polynomial time}
NP-complete: the subset of NP which are the "hardest problems"

Alternative Representation

- Every element p in P1 can map to an element q in P2 such that p is true (decision problem) if and only if q is also true.
- Must find a mapping for such true elements in P1 and P2, as well as for false elements.
- Ensure that mapping can be done in polynomial time.
- *Note: P1 is unknown, P2 is difficult

Cook's Theorem

- Stephen Cook (Turing award winner) found the first NP-Complete problem, 3SAT.
  - ❖ Basically a problem from Logic.
  - ❖ Generally described using Boolean formula.
  - ❖ A Boolean formula involves AND, OR, NOT operators and some variables.

  Ex: (x or y) and (x or z), where x, y, z are boolean variables.
  - ❖ Problem Definition – Given a boolean formula of m clauses, each containing 'n' boolean variables, can you assign some values to these variables so that the formula can be true?
    - ❖ Boolean formula: $(x \vee y \vee \hat{z}) \Lambda (x \vee y \vee \hat{z})$
  - ❖ *Try all sets of solutions. Thus we have exponential set of possible solutions. So it is a NPC problem.*
- Having one definite NP-Complete problem means others can also be proven NP-Complete, using reduction.

Space-Time Trade offs:

Introduction, Sorting by counting, I/P Enhancement in String Matching.

Def: An algorithm must be time efficient & Space efficient. To achieve both may not be possible for some of the Algorithms. In some situations, space may be an important factor & in some other situation, time may be an important factor.

Thus, space & time tradeoff is a situation in which either time efficiency can be achieved at the cost of extra money usage or space efficiency can be achieved at the cost of execution speed.

Sorting by Comparison

```
Algorithm ComparisonCountingSort (A[0..n-1])
// Sorts an array by comparison counting
// I/p : Array A[0..n-1] of orderable values.
// O/p: Array S[0..n-1] of A's elements sorted in non-
       decreasing 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.
```

Eg4

Array A[0..5]

62	31	84	96	19	47

Initially

After pass $i = 0$

$i = 1$

$i = 2$

$i = 3$

$i = 4$

Final state

Count[]	0	0	0	0	0	0
	3	0	1	1	0	0
		1	2	2	0	1
			4	3	0	1
				5	0	1
					0	2
	3	1	4	5	0	2

Array S[0..5]

19	31	47	62	84	96

Time Complexity.

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

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

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

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

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

Problem2 Give the tracing for the Comparison Count algorithm to sort the elements 10, 20, 10, 20, 10, 5. Will the Comparison counting algorithm work Correctly for arrays with Equal values? Is this algorithm stable.

	0	0	0	0	0	initialize.
3	1	0	1	0	0	$i=0$
	5	0	1	0	0	$i=1$
		2	2	0	0	$i=2$
			4	0	0	$i=3$
				1	0	$i=4$
3	5	2	4	1	0	

5	10	10	10	20	20	sorted array.

→ yes, Comparison Counting algorithm works well for arrays with Equal values

→ Since the relative position of duplicate Elements are not the same before & after sorting the algorithm is not stable.

Distribution Counting Algorithm.

```
Algorithm DistributionCounting(A[0..n-1])
// sorts an array of integers from a limited range
   by distribution counting.
// I/p: Array A[0..n-1] of integer b/w l & u (l≤u)
      l: lower bound & u: upper bound.
// O/p: Array S[0..n-1] of A's elements sorted
      in nondecreasing order
for j ← 0 to u-l do D[j] ← 0    //initialize frequencies
for i ← 0 to n-1 do
    D[A[i]-l] ← D[A[i]-l] + 1    //Computing frequencies
for j ← 1 to u-l do
    D[j] ← D[j-1] + D[j]    //reuse for distribution
for i ← n-1 downto 0 do
    j ← A[i] - l
    S[D[j]-1] ← A[i]
    D[j] ← D[j] - 1
return S
```

Problem1 Sort the numbers using distributed counting method.

13	11	12	13	12	12

Soln The frequency & distribution array are as follows.

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

D[0..2]

A[5] = 12	1	(4)	6
A[4] = 12	1	(3)	6
A[3] = 13	1	2	(6)
A[2] = 12	1	(2)	5
A[1] = 11	(1)	1	5
A[0] = 13	0	1	(5)

S[0..5]

			12		
		12			
					13
	12				
11					
				13	

Input Enhancement in String Matching.

Horspool's Algorithm.

→ The technique used here is to first align the pattern string to the beginning of the text string.

→ Instead of comparing from left to right, we compare it from right to left. This results in two cases.

① If all the characters of the pattern string matched with the corresponding characters of the text string & found same, then search is successful.

② If there is a mismatch, then the pattern string has to be shifted towards right. How many positions the pattern string has to be moved towards right is depends on the shift value present in the shift table for the corresponding character of the text string.

Computing the shift table.

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern} \\ \\ \text{the distance from the rightmost } c \text{ among the first} \\ m-1 \text{ characters of the pattern to its last} \\ \text{character, otherwise.} \end{cases}$$

There can be four possibilities in shifting the pattern.

Case 1:

$S_0 \ldots\ldots S \ldots\ldots S_{n-1}$ [TEXT]

BARBER

BARBER [PATTERN]

{shift by 6 (size of pattern)}

If there are no c's in the pattern, we can safely shift the pattern by its entire length.

Case 2:

$S_0 \ldots\ldots B \ldots\ldots S_{n-1}$ [TEXT]

BARBER

BARBER [PATTERN]

BARBER {shift by 2, shift(B) is 2.}

If there are occurences of character c in the pattern but it is not the last one there, the shift should align the rightmost occurence of c in the pattern with the c in the text.

Case 3:

$S_0 \ldots\ldots$ MER $\ldots\ldots S_{n-1}$ [TEXT]

LEADER

LEADER {shift by the pattern size}

If c happens to be the last character in the pattern but there are no c's among its other m-1 characters, the shift should be similar to that of Case 1: the pattern should be shifted by the entire Pattern's length m.

Case 4:

$S_0 \ldots\ldots OR \ldots\ldots S_{n-1}$ [TEXT]

≠ ||

REORDER

REORDER [PATTERN]

Finally, if c happens to be the last character in the pattern & there are other c's among its first m-1 characters, the shift should be similar to that of case 2 the rightmost occurence of c among the first m-1 characters in the pattern should be aligned with the text's c.

Algorithm shiftTable(P[0..m-1])
// Fills the shift table used by Horspool's & Boyer-Moore Algorithms
// I/P: Pattern P[0..m-1] & an alphabet of possible chars
// O/P: Table[0..size-1] indexed by the alphabet's chars & filled with shift sizes computed by formula (1)
initialize all the Elements of Table with m
for j ← 0 to m-2 do
    Table[P[j]] ← m-1-j
return Table.

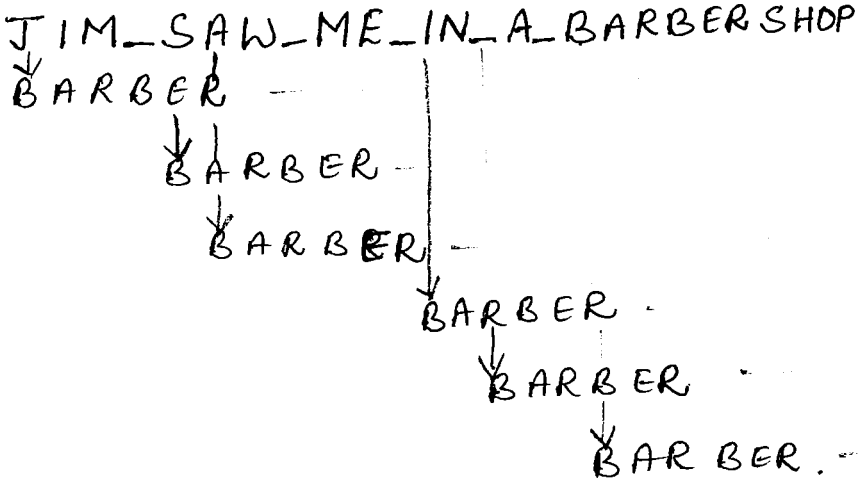
```
Algorithm HorspoolMatching(P[0..m-1], T[0..n-1])
//Implements Horspool's algorithm for string matching
//I/P: pattern P[0..m-1] and Text T[0..n-1]
//O/P: The index of the left end of the first matching
      substring or -1 if there are no matches
ShiftTable(P[0..m-1]) //generate Table of shifts
i ← m-1                    //position of the pattern's right end
while i ≤ n-1 do
    K ← 0
    while K ≤ m-1 and P[m-1-K] = T[i-K] do
        K ← K+1
    if K = m
        return i-m+1
    Else
        i ← i + Table[T[i]]
return -1
```

Example1: Consider searching for the pattern BARBER in a text that comprises English letters & spaces (denoted by underscores). Use Horspool's algorithm to search the pattern.

Soln

Let us construct the shift table considering the pattern & the text.

Character c	A	B	C	D	E	F	...	R	.....	Z	_
Shift t(c)	4	2	6	6	1	6	6	3	6	6	6



shift(A) = 4

shift(E) = 1

no match ⑥ shift

shift(B) = 2

shift(R) = 3

Not found.

Boyer-Moore Algorithm

Boyer moore Algorithm

Steps

(1) For a given pattern & the alphabet used in both the pattern & text construct the bad symbol shift table (a)

(2) Using the pattern, construct the good suffix table (b)

(3) Align the pattern against the beginning of the text

(4) Repeat the following step until either a matching substring is found or the pattern reaches beyond the last character of the text

Starting with the last character in the pattern, compare the corresponding characters in the pattern & the text until either all 'm' character pairs are matched (then stop) or a mismatching pair is encountered after $k > 0$ character pairs are matched successfully

On the latter case retrieve the entry $t_1(c)$ from the bad symbol Table(a) where 'c' is the text's mismatched character

If $k > 0$, also retrive the d_2 entry from good suffix table

Shift the pattern ~~by~~ to the right by the number of positions computed by the formula

$$d = \begin{cases} d_1 & \text{if } k = 0 \\ \max(d_1, d_2) & \text{if } k > 0 \end{cases}$$

where $d_1 = \max(t_1(c) - k, 1)$.

→ Search for BAOBAB in a text made of English letters and spaces

* Table (a) the Bad symbol Table

c	A	B	C	D	..	O	..	2	..	-	
$t_1(c)$	1	2	6	6		3		6		6	

* Table (b) the Good suffix Table

k	pattern	d_2
1	BAOBA<u>B</u>	2
2	A<u>B</u>AOBA<u>B</u>	5
3	BA<u>B</u>AOBA<u>B</u>	5
4	OBA<u>B</u>AOBA<u>B</u>	5
5	<u>B</u>AOBA<u>B</u>	5

Teks

B E S S _ K N E W _ A B O U T _ B A O B A B S

Pattern

B A O B A B

Solusi

B E S S _ K N E W _ A B O U T _ B A O B A B S

B A O B A B

$d_1 = t_1(k) - 0 = 6$

B A O B A B

$d_1 = t_1(\_) - 2 = 4$

$d_2 = 5$

$d = \max(4,5) = 5$

B A O B A B

$d_1 = t_1(\_) - 1 = 5$

$d_2 = 2$

$d = \max(5,2) = 5$

B A O B A B

* construct the good suffix table for the pattern

ABCBAB

k	pattern	d_2
1	ABCBAB	2
2	ABCBAB	4
3	ABCBAB	4
4	ABCBAB	4
5	ABCBAB	4

Copying with the limitations of Algorithm Power unit-VII

Topics to cover

* BackTracking
  - ↳ n Queens Problem
  - ↳ Hamiltonian Circuit Problem
  - ↳ Subset-Sum Problem.
* Branch-and-Bound
  - ↳ Assignment Problem
  - ↳ Knapsack Problem
  - ↳ Travelling Salesman Problem.
* Approximation Algorithm for NP-hard Problem.

Backtracking.

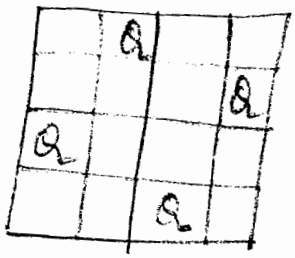
It is a general algorithm for finding all (or some) solution to some computational problem, that incrementally builds candidates to the solutions & abandons each partial candidate c ("backtracks") as soon as it determines that c cannot possibly be completed to a valid solution.

Backtracking technique can be used to solve n queens problem. This technique is used to solve Constraint satisfaction problems such as crosswords, Verbal arithmetic, Sudoku, It is also used in Parsing, knapsack problem & other combinatorial optimization problem.

N-queens Problem

→ Given N×N chess board, it is required to place all N-queens on the chessboard such that no two queens attack each other.
ie two or more queens should not be placed in the same row, same column or same diagonal.

→ for Example for 4×4 chess board the solution can be as shown below



(x_1, x_2, x_3, x_4)

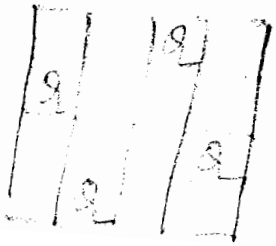
R: row
C: column

→ The algorithm should comply two constraints
(1) Implicit Constraints (2) Explicit Constraints.

→ Implicit Constraint is that "no two queens should attack each other. Each x_i should be chosen so that no two queens lie in same column or same row or same diagonal.

→ Explicit Constraint is that, a queen can be placed on any of the columns 1, 2, 3 or 4. So in a 4-tuple (x_1, x_2, x_3, x_4), each $x_i \in S$ where $S = \{1, 2, 3, 4\}$

The 2nd solution for 4×4 chess board (4-queens problem)



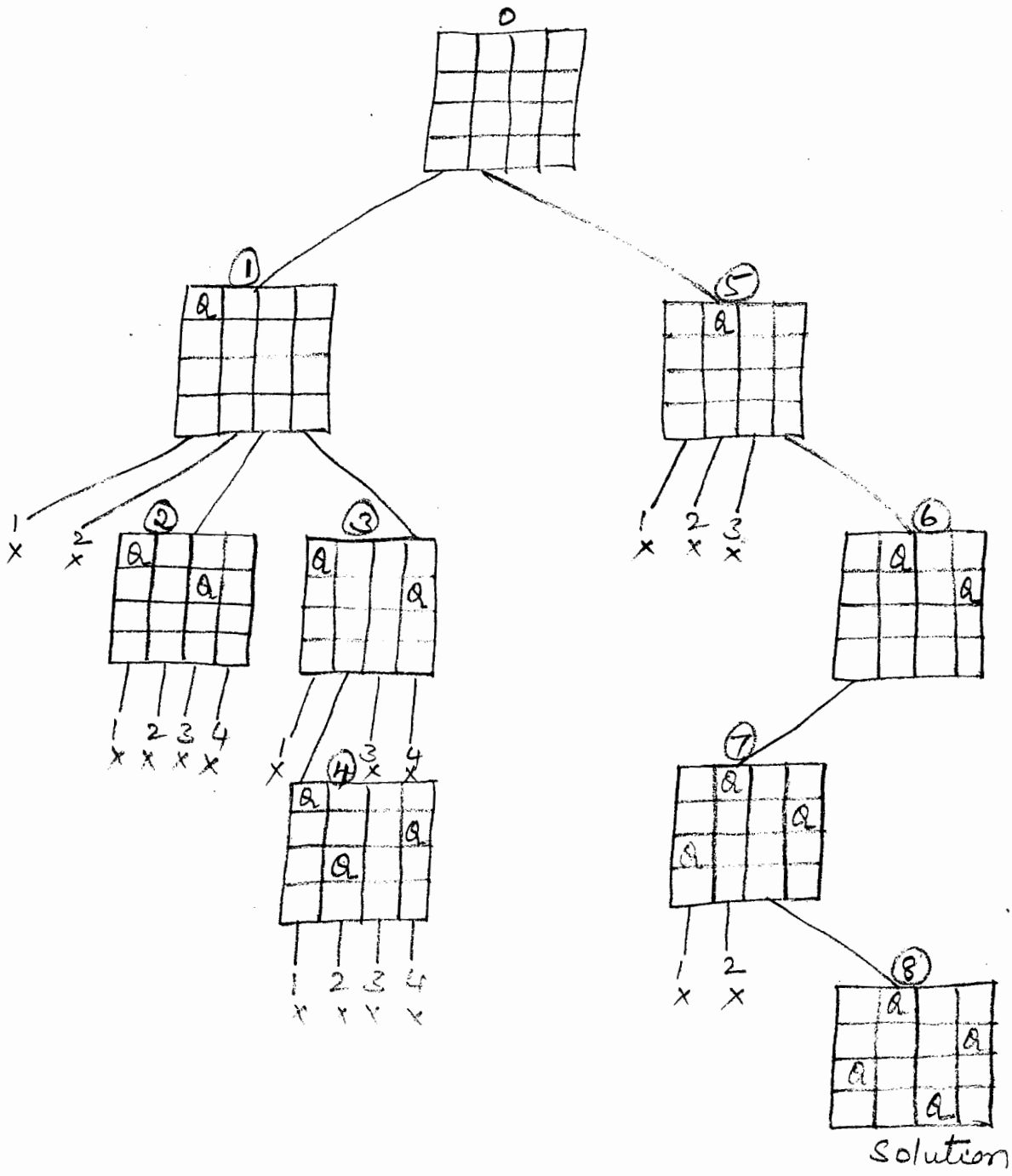


figure: State-space tree of solving the four queens problem by backtracking. x denotes an unsuccessful attempt to place a queen in the indicated column. The number above the nodes indicate the order in which the nodes are generated.

Algorithm: Can a new queen be placed?

```
Algorithm Place (k, i)
// Returns true if a queen can be placed in kth row &
// ith column. Otherwise it return 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) // Two in the same column
            or (Abs(x[j]-i) = Abs(j-k)))
                // or in the same diagonal
            then return false;
    return true;
}
```

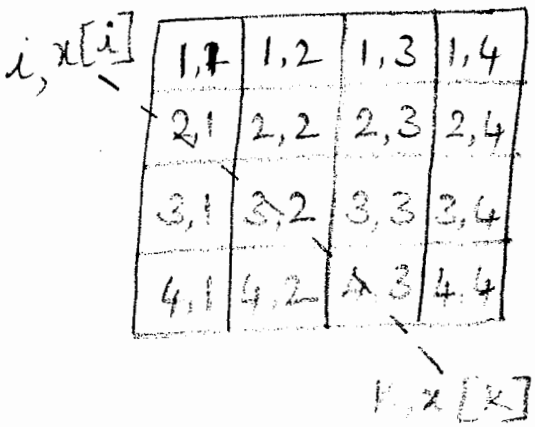
Algorithm: All solutions to the n-queens problem.

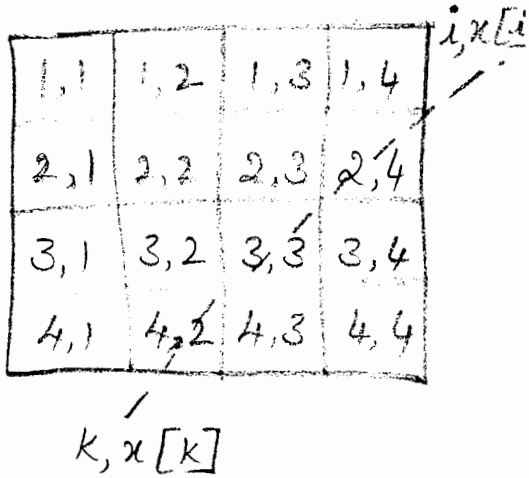
```
Algorithm NQueens (k, n)
// using backtracking, this procedure prints all
// possible placements of n queens on an n x n
// chessboard so that they are nonattacking.
{
    for i := 1 to n do
    {
        if Place (k, i) then
        {
            x[k] := i;
            if (k == n) then write (x[1:n]);
            Else NQueens (k+1, n);
        }
    }
}
```

The following points have to be noted while designing n-queens solution.

* $(i, x[i])$ gives the position of i^{th} queen in row i & column $x[i]$.
* $(k, x[k])$ gives the position of k^{th} queen in row k & column $x[k]$
* if i^{th} queen & k^{th} queen are in same column, then $x[i] = x[k]$ hence

$x[i] = x[k]$ indicate that queens attack Verlically — (1)





$i - x[i] = k - x[k]$ — (2)

Can be rewritten as

$i - k = x[i] - x[k]$ — (4)

$i + x[i] = k + x[k]$ — (3)

Can be rewritten as

$i - k = -x[i] + x[k]$ — (5)

Equation (4) & (5) Can be generalized as

$$|i - k| = |x[i] - x[k]|$$

This indicates the queens attack directly

ie if ($x[i] = x[k]$ OR $abs(i-k) = abs(x[i] - x[k])$

return 0

End if.

Subset Problem

Definition: The sum of subset problem is stated as follows "Given n positive numbers $w_1, w_2, \ldots w_n$ & m it is required to find all subsets of w_i where $1 \le i \le n$ whose sum is m."

for example, let $W = \{3, 5, 6, 7\}$ and $m = 15$. Now, we have to select one or more items from a set W such that their sum is 15. with this constraint there is one solution ie. $\{3, 5, 7\}$

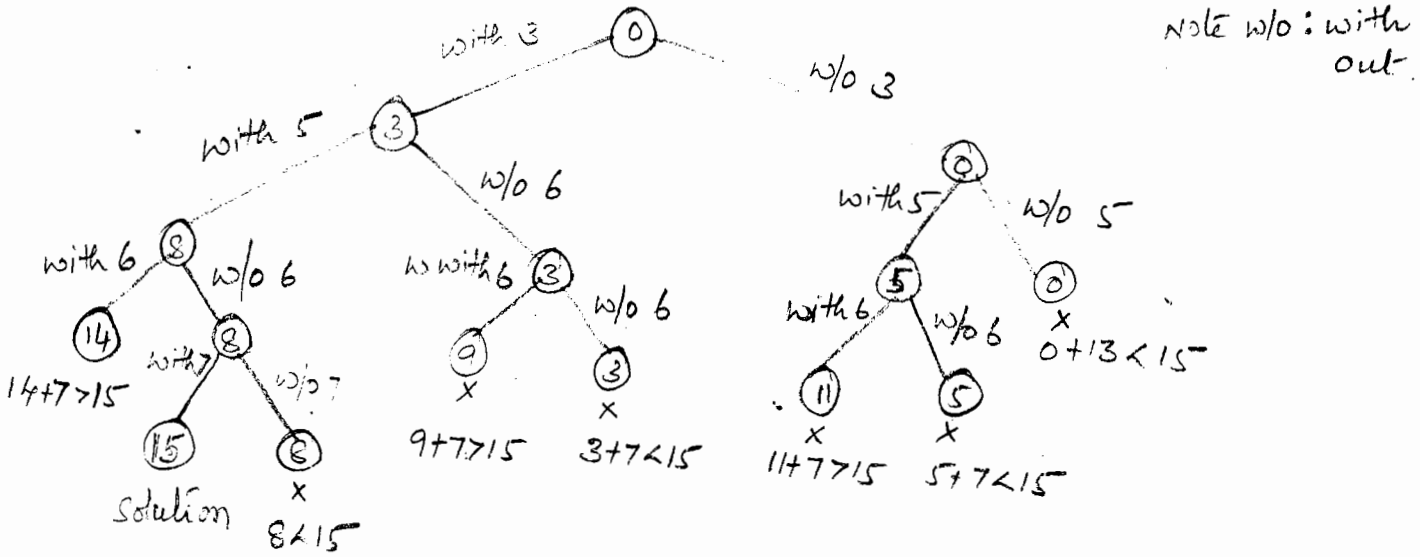


Figure: Complete state-space tree of the backtracking algo applied to the instance $W = \{3, 5, 6, 7\}$ & $m = 15$ of the subset-sum problem.
The number inside a node is the sum of the elements already included in subsets represented by the node.
The inEquality below a leaf indicates the reason for its termination.

```
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 deter-
// mined. s = Σ_{j=1}^{k-1} w[j]*x[j] & r = Σ_{j=k}^{n} w[j].
// The w[j]'s are in non decreasing 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 & 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]);
    }
}
```

Branch and Bound (B&B)

It is a general algorithm for finding optimal solution of various optimization problems, especially in discrete & combinatorial optimization. A branch-and-bound algorithm consists of a systematic enumeration of all candidate solutions, where large subsets of fruitless canditates are discarded en masse, by using upper & lower estimated bounds of the quantity being optimized.

The method was first proposed by A H Land & A.G. Doig in 1960 for discrete programming (wikipedia.org

Knapsack problem using Branch-&-Bound.

→ To obtain the optimal solution, it is convenient to order the items in descending order of their Profit-to-weight ratios.

→ In such situation, the selection of the first item gives the maximum profit & last one gives the least profit.

→ All the items are arranged as shown below.

$$P_1/w_1 \geq P_2/w_2 \geq P_3/w_3 \ldots\ldots P_n/w_n$$

→ Construct the state-space tree for the knapsack problem using branch-&-bound method as shown below:

* Each node of the tree on i^{th} level where $i=0$ to n represent all subsets of n items consisting of items selected.

* A selection is determined by a path from root to the node where a branch going to the left indicates the inclusion of the next item
* A branch going to the right indicates its exclusion from the solution.
* For a particular selection, record the total weight w & the total profit value v in the node along with some upper bound ub.
* The upper bound ub can be calculated using the following relation

$$ub = p + (M - w)(P_{i+1}/w_{i+1})$$

where

w is the total weight of all objects placed into knapsack.

v is the total profit or value of all objects placed into knapsack

Ex:1 Solve the following knapsack problem using brand-&-bound given the following data & Capacity of knapsack $m = 10$

item	weight	profit	Profit/weight
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4

Soln: The given data is shown below

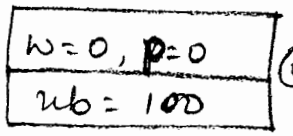
weight	Value (profit)	Profit/weight.
$w_1 = 4$	$P_1 = 40$	$P_1/w_1 = 10$
$w_2 = 7$	$P_2 = 42$	$P_2/w_2 = 6$
$w_3 = 5$	$P_3 = 25$	$P_3/w_3 = 5$
$w_4 = 3$	$P_4 = 12$	$P_4/w_4 = 4$

The capacity of the knapsack = m = 10
The upper bound is calculated using the following relation : $ub = p + (m-w)\, P_{i+1}/w_{i+1}$

Item i = 0 Initial state-space tree $P_i/w_i = 10$

w = 0, P = 0, m − w = 10 − 0 = 10

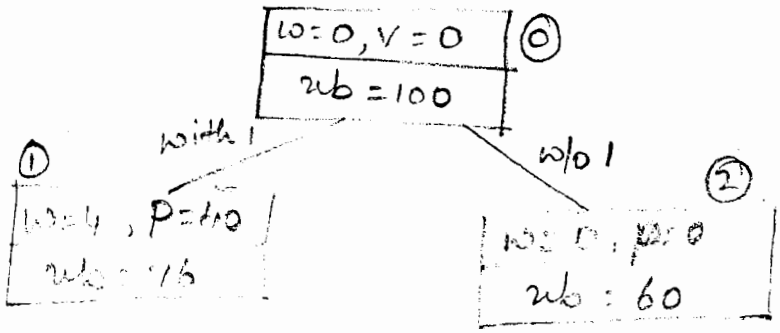
So ub = p + (m − w) P_1/w_1
= 0 + 10 * 10 = 100



Item i = 1 : $P_2/w_2 = 6$

[with item1]
w = weight of item 1 + w = 4+0 = 4
P = P + P_1 = 0 + 40 = 40
m − w = 10 − 4 = 6
ub = p + (m − w) P_{i+1}/w_{i+1}
= 40 + 6 * 6
= 76.

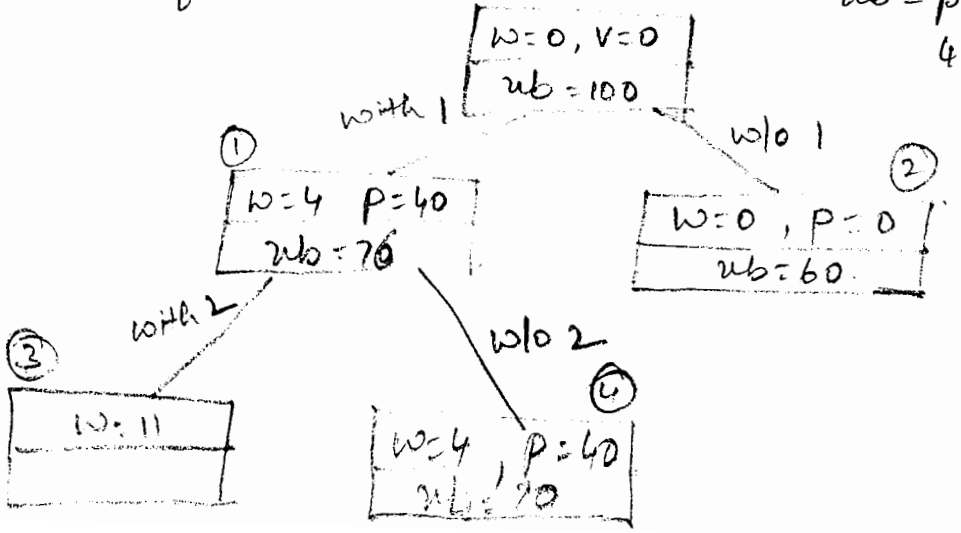
[w/o item 1]
w = 0+0 = 0
P = P + w/o P_1 = 0+0 = 0
m − w = 10 − 0 = 10
ub = p + (m − w) P_2/w_2
= 0 + 10 * 6 = 60.



Item i = 2 $P_3/w_3 = 5$

(with item 2)
w = weight of item 2 + w.
= 7 + 4 = 11
Not feasible.

(w/o item 2)
w = w/o wt of item 2 + w = 0 + 4
P = P + w/o P_2 = 40 + 0 = 40
m − w = 10 − 4 = 6
ub = p + (m − w) P_3/w_3
40 + 6 * 5 = 70.



→ out of nodes 2, 3, 4 Consider node 4 with highest bound = 70

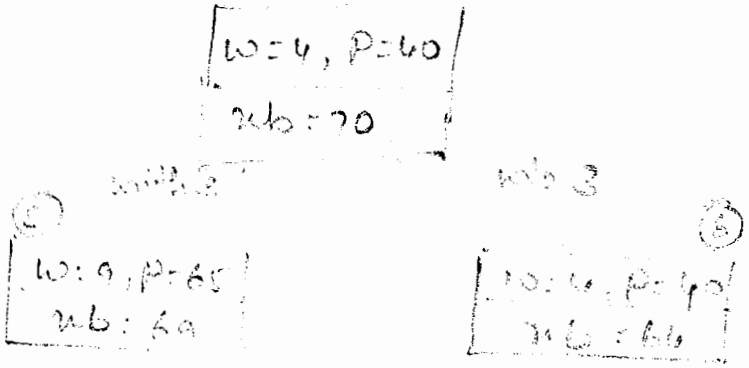
Item i=3 : P_4/w_4

(with item3)

w = wt of item3 + w = 5+4=9
P = P+P_3 = 40 + 25 = 65
m-w = 10-9 = 1
ub = p + (m-w) P_4/w_4
= 65 + 1 * 4 = 69

(with out w/o item3)

w = w/o wt of item3 + w = 0+4=4
P = P + w/o P_3 = 40+0 = 40
m-w = 10-4 = 6
ub = p + (m-w) P_4/w_4
= 40 + 6*4 = 64.



→ out of nodes 5, 6 & 2 Consider node 5 with highest bound = 69.

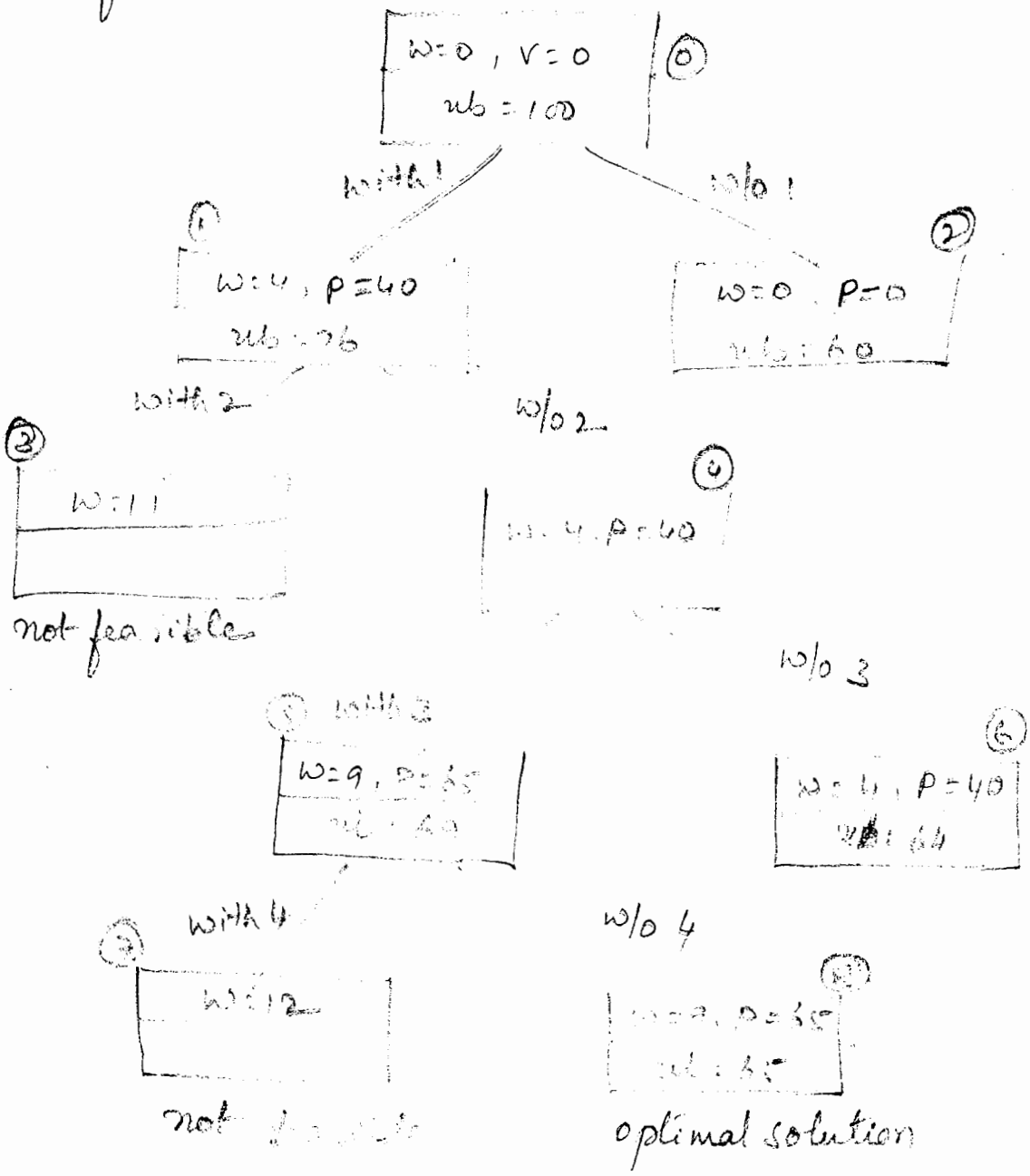
Item i=4 : P_5/w_5 =

with item 4
w = wt of item4 + w
= 3 + 9 = 12
Not feasible.

(w/o item 4)

w = w/o wt of item4 + w = 0+9 = 9
P = P + w/o P_4 = 65 + 0 = 65
m-w = 10-9 = 1
ub = 65 (because it is the last item so ub = P)

The final tree.



So, final optimal solution is obtained by considering the objects selected while moving from root node 0 down to node 8. ∴ final solution vector = {1, 3}

Knapsack problem is maximization problem, since we are interested in maximum profit.

Assignment Problem

-> The worst case analysis for Assignment Problem when using Exhaustive approach is $f(n) \in O(n!)$

Prob: Find the optimal solution for the given assignment problem which is represented as a matrix as shown below using branch & bound method.

	J_1	J_2	J_3	J_4
a	9	2	7	8
b	6	4	3	7
c	5	8	1	8
d	7	6	9	4

Solution: Now let us find the lower bound for the problem given. The cost of any solution, including the optimal solution cannot be smaller than the sum of the smallest elements in row of matrix. So take minimum of each row & add them to get the initial lower bound as shown below:

	J_1	J_2	J_3	J_4	
a	9	2	7	8	2
b	6	4	3	7	3
c	5	8	1	8	1
d	7	6	9	4	4

lower bound lb = (10)

start
lb = 10

Consider Person a: Assign various jobs to a & Compute the lower bound as shown below

Let a -> 1 where cost = 9
Since a -> 1 leave row a column 1

	J_1	J_2	J_3	J_4	min
a	9	2	7	8	9
b	6	4	3	7	3
c	5	8	1	8	1
d	7	6	9	4	4 (1)

lb = 9+3+1+4 = (17)

	J_1	J_2	J_3	J_4	
a	9	2	7	8	2
b	6	4	3	7	3
c	5	8	1	8	1
d	7	6	9	4	4

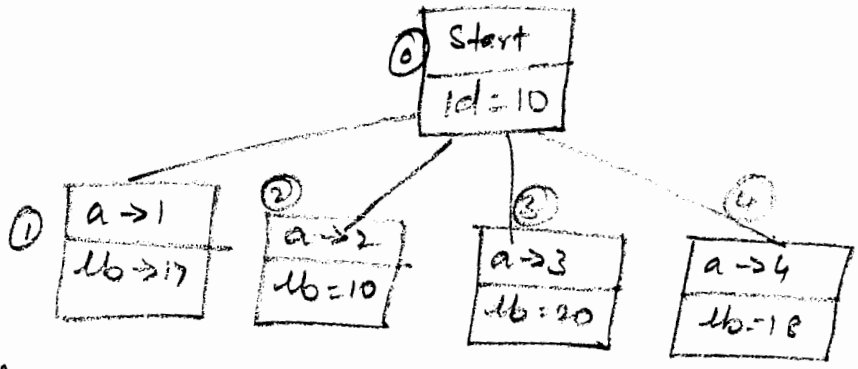
lb = 2+3+1+4 = (10)

	J_1	J_2	J_3	J_4	min
a	~~9~~	~~2~~	7	~~8~~	7
b	6	4	~~3~~	7	4
c	5	8	1	8	5
d	7	6	9	4	4.

$lb = 7+4+5+4 = (20)$

	J_1	J_2	J_3	J_4	
a	~~9~~	~~2~~	~~7~~	8	8
b	6	4	3	~~7~~	3
c	5	8	1	~~8~~	1
d	7	6	9	4	6

$lb = 8+3+1+6 = (18)$



Consider Person b

	J_1	J_2	J_3	J_4	
a		2			2
b	6		~~3~~	~~7~~	6
c	~~5~~		1	8	~~1~~
d	7		9	4	4.

$lb = 2+6+1+4 = (13)$

	J_1	J_2	J_3	J_4
a		2		
b	~~6~~		3	~~7~~
c	5			8
d	7		9	4

$lb = 2+3+5+4 = (14)$

	J_1	J_2	J_3	J_4	
a		2			2
b	~~6~~		3	7	7
c	5		1	~~8~~	1
d	7		9	4	7

$lb = 2+7+1+7 = (17)$

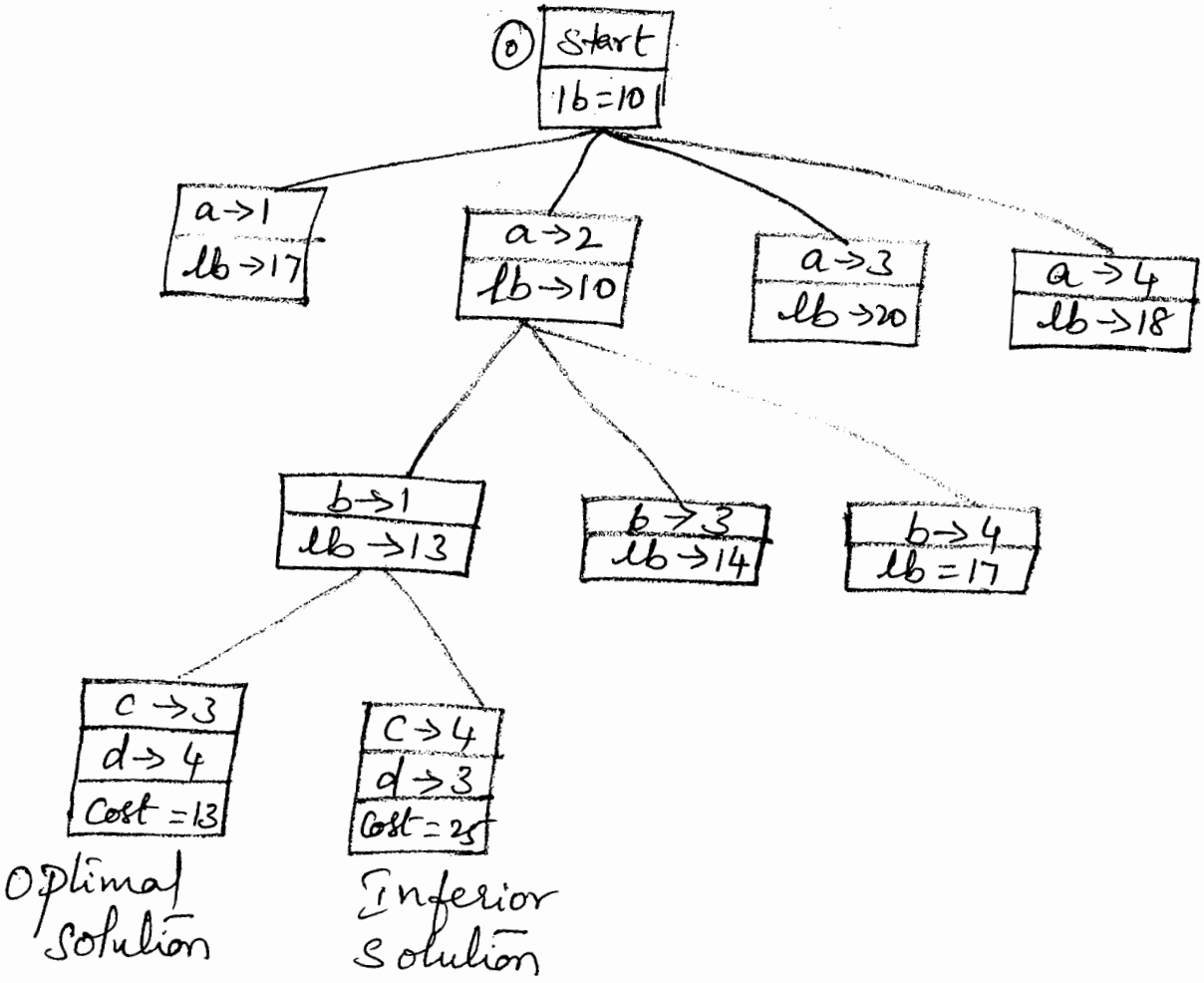
Consider person C.

	J_1	J_2	J_3	J_4	
a		2			2
b	6				6
c			1	~~8~~	1
d			9	4	4

$lb = 2 + 6 + 1 + 4 = (13)$

	J_1	J_2	J_3	J_4
a		2		
b	6			
c			+	8
d			9	4

$lb = 2 + 6 + 8 + 9 = (25)$



The optimal solution is $\{a \to 2, b \to 1, c \to 3, d \to 4\}$

Cost $2 + 6 + 1 + 4 = 13$

The Branch-and-bound strategy

- This strategy can be used to solve optimization problems without an exhaustive search in the average case.
- 2 mechanisms:
  - A mechanism to generate branches when searching the solution space
  - A mechanism to generate a bound so that many braches can be terminated
- It is efficient in the average case because many branches can be terminated very early.
- Although it is usually very efficient, a very large tree may be generated in the worst case.
- Many NP-hard problem can be solved by B&B efficiently in the average case; however, the worst case time complexity is still exponential.

Bounding

- A bound on a node is a guarantee that any solution obtained from expanding the node will be:
  - Greater than some number (lower bound)
  - Or less than some number (upper bound)
- If we are looking for a minimal optimal, as we are in weighted graph coloring, then we need a lower bound
  - For example, if the best solution we have found so far has a cost of 12 and the lower bound on a node is 15 then there is no point in expanding the node
    - The node cannot lead to anything better than a 15
- We can compute a lower bound for weighted graph color in the following way:
  - The actual cost of getting to the node
  - Plus a bound on the future cost
    - Min weight color * number of nodes still to color
      - That is, the future cost cannot be any better than this
- Recall that we could either perform a depth-first or a breadth-first search
  - Without bounding, it didn't matter which one we used because we had to expand the entire tree to find the optimal solution
  - Does it matter with bounding?
    - Hint: think about when you can prune via bounding
- We prune (via bounding) when:
  (currentBestSolutionCost <= nodeBound)
- This tells us that we get more pruning if:
  - The currentBestSolution is low
  - And the nodeBound is high
- So we want to find a low solution quickly and we want the highest possible lower bound
  - One has to factor in the extra computation cost of computing higher lower bounds vs. the expected pruning savings

The Assignment Problem

- In many business situations, management needs to assign - personnel to jobs, - jobs to machines, - machines to job locations, or - salespersons to territories.
- Consider the situation of assigning n jobs to n machines.
- When a job i (=1,2,....,n) is assigned to machine j (=1,2, ...n) that incurs a cost Cij.
- The objective is to assign the jobs to machines at the least possible total cost.

- This situation is a special case of the Transportation Model And it is known as the *assignment problem*.
- Here, jobs represent "sources" and machines represent "destinations."
- The supply available at each source is 1 unit And demand at each destination is 1 unit.

		Machine				
		1	2	.....	n	Source
Job	1	C11	C12	.....	C1n	1
	2	C21	C22	.....	C2n	1
	.	.	.		.	.
	.	.	.		.	.
	.	.	.		.	.
	n	Cn1	Cn2	.....	Cnn	1
Destination		1	1	.....	1	

The assignment model can be expressed mathematically as follows:

Xij= 0, if the job j is not assigned to machine i

1, if the job j is assigned to machine i

$$\text{Min} \quad \sum_{i=1}^{n}\sum_{j=1}^{n} C_{ij} X_{ij}$$

(Sum of assignments from a source should be exactly equal to 1):

$$\sum_{j=1}^{n} X_{ij} = 1 \qquad \text{For } i=1,2,\ldots,n$$

(Sum of assignments to a destination should be equal to the demanded quantity by that destination):

$$\sum_{i=1}^{n} X_{ij} = 1 \qquad \text{For } j=1,2,\ldots,n$$

(Quantities to be assigned can be either 0 or 1):

$$X_{ij} = 0 \text{ or } 1 \qquad \text{For all i and j.}$$

The Assignment Problem Example

- Ballston Electronics manufactures small electrical devices.
- Products are manufactured on five different assembly lines (1,2,3,4,5).
- When manufacturing is finished, products are transported from the assembly lines to one of the five different inspection areas (A,B,C,D,E).
- Transporting products from five assembly lines to five inspection areas requires different times (in minutes)

	Inspection Area				
Assembly Line	A	B	C	D	E
1	10	4	6	10	12
2	11	7	7	9	14
3	13	8	12	14	15
4	14	16	13	17	17
5	19	11	17	20	19

Under current arrangement, assignment of inspection areas to the assembly lines are 1 to A, 2 to B, 3 to C, 4 to D, and 5 to E. This arrangement requires 10+7+12+17+19 = 65 man minutes.

- Management would like to determine whether some other assignment of production lines to inspection areas may result in less cost.
- This is a typical assignment problem. n = 5 And each assembly line is assigned to each inspection area.
- It would be easy to solve such a problem when n is 5, but when n is large all possible alternative solutions are n!, this becomes a hard problem.
- Assignment problem can be either formulated as a linear programming model, or it can be formulated as a transportation model.
- However, An algorithm known as *Hungarian Method* has proven to be a quick and efficient way to solve such problems.

Assignment Problem Revisited:

Select one element in each row of the cost matrix *C* so that:

- **no two selected elements are in the same column**
- **the sum is minimized**

Example

	Job 1	Job 2	Job 3	Job 4
Person *a*	9	2	7	8
Person *b*	6	4	3	7
Person *c*	5	8	1	8
Person *d*	7	6	9	4

Lower bound: Any solution to this problem will have total cost at least: 2 + 3 + 1 + 4 (or 5 + 2 + 1 + 4)

Example: First two levels of the state-space tree

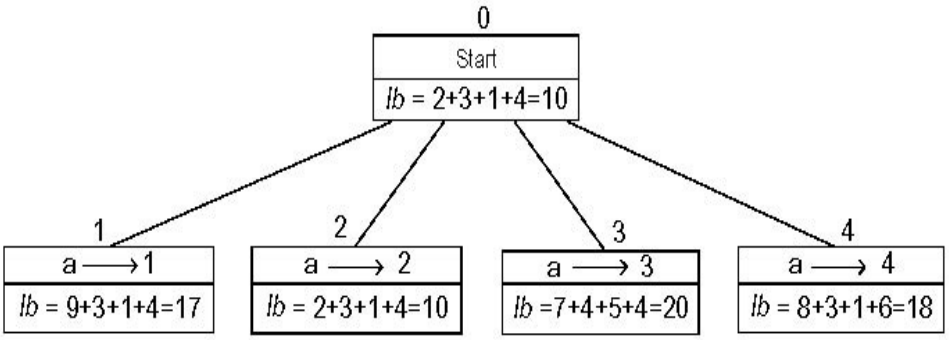


Figure 11.5 Levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated. A node's fields indicate the job number assigned to person a and the lower bound value, lb, for this node.

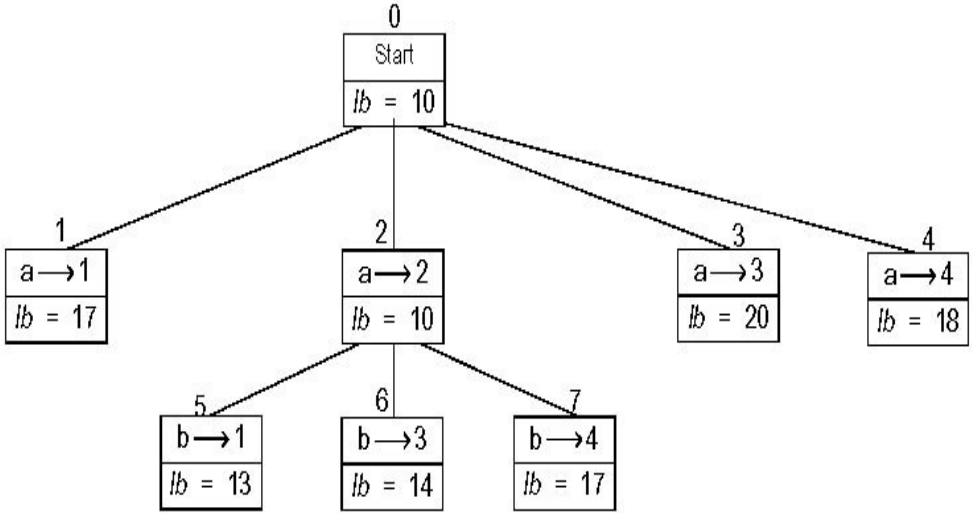


Figure 11.6 Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm

Example: Complete state-space tree

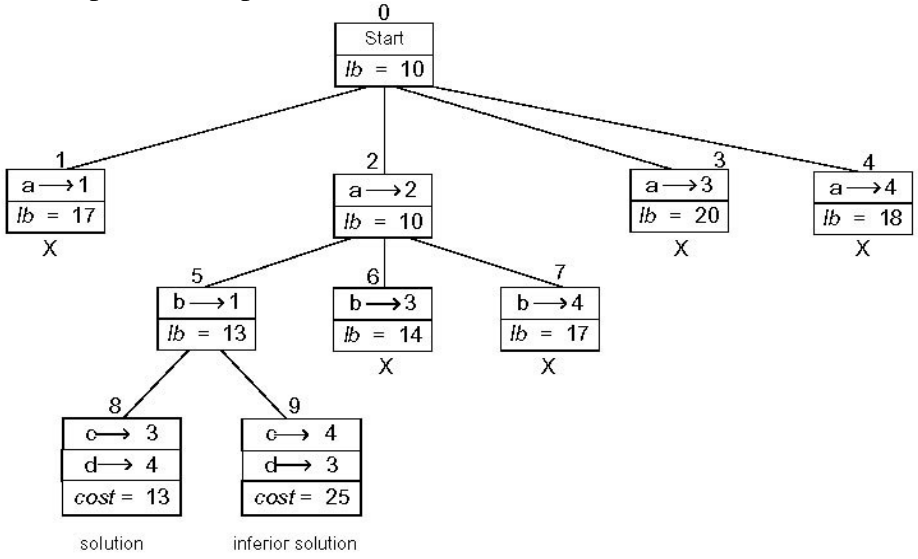
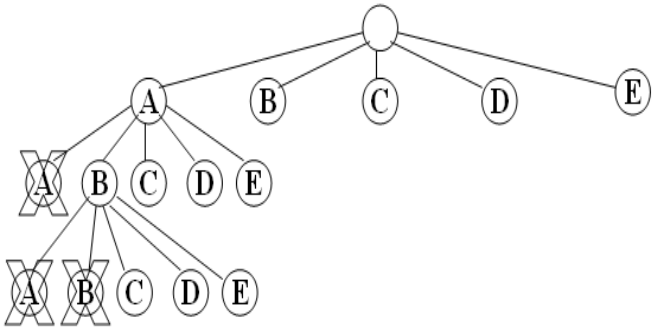


Figure 11.7 Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm

Traveling Salesperson Problem

- This is a classic CS problem
- Given a graph (cities), and weights on the edges (distances) find a minimum weight tour of the cities
  - Start in a particular city
  - Visit all other cities (*exactly* once each)
  - Return to the starting city
- Cannot be done by brute-force as this is worst-case exponential or worse running time
  - So we will look to backtracking with pruning to make it run in a reasonable amount of time in most cases
- We will build our state space by:
  - Having our children be all the potential cities we can go to next
  - Having the depth of the tree be equal to the number of cities in the graph
    - we need to visit each city exactly once
- So given a fully connected set of 5 nodes we have the following state space
  - only partially completed

Traveling Salesperson Problem



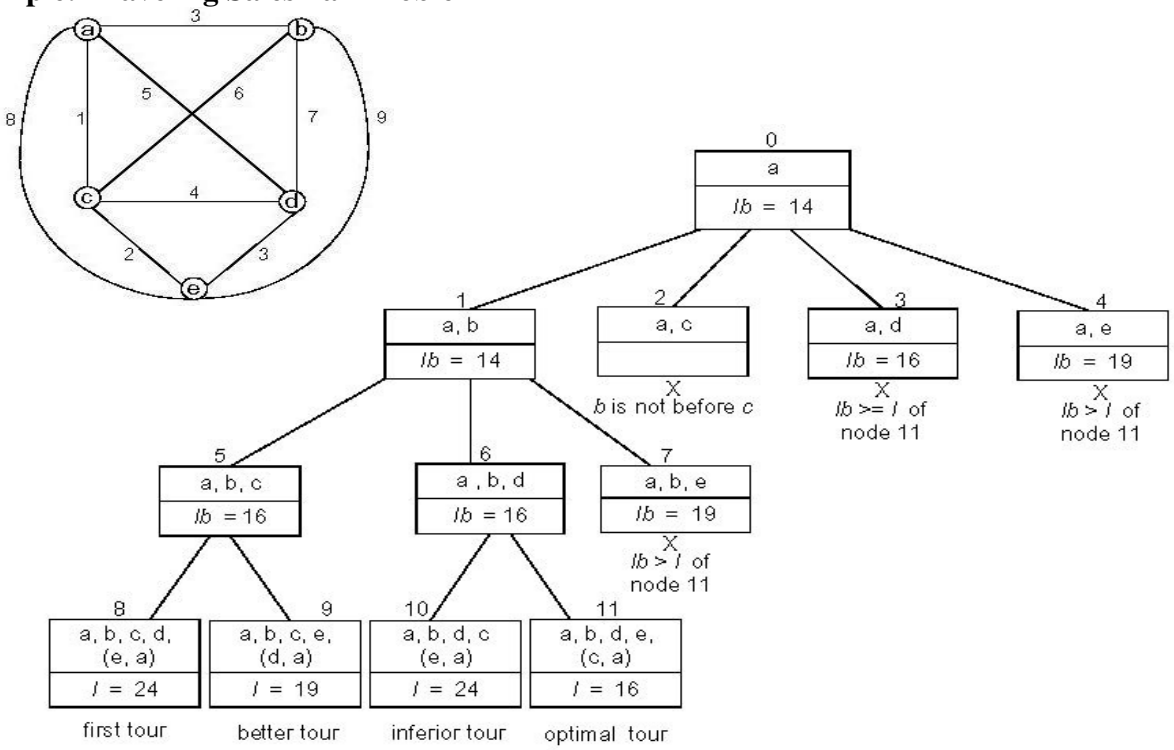
- Now we need to add bounding to this problem
  - It is a minimization problem so we need to find a lower bound
- We can use:
  - The current cost of getting to the node plus
  - An underestimate of the future cost of going through the rest of the cities
    - The obvious choice is to find the minimum weight edge in the graph and multiply that edge weight by the number of remaining nodes to travel through
- As an example assume we have the given adjacency matrix
- If we started at node A and have just traveled to node B then we need to compute the bound for node B
  - Cost 14 to get from A to B
  - Minimum weight in matrix is 2 times 4 more legs to go to get back to node A = 8
  - For a grand total of 14 + 8 = 22

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0

- Recall that if we can make the lower bound higher then we will get more pruning
- Note that in order to complete the tour we need to leave node B, C, D, and E
  - The min edge we can take leaving B is min(14, 7, 8, 7) = 7
  - Similarly, C=4, D=2, E=4
- This implies that at best the future underestimate can be 7+4+2+4=17
- 17 + current cost of 14 = 31
  - This is much higher than 8 + 14 = 22

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0

Example: Traveling Salesman Problem



The 0/1 knapsack problem

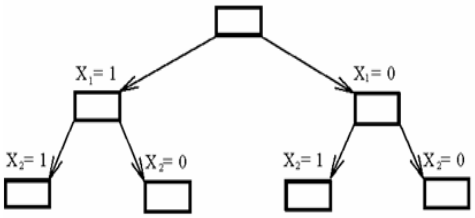
- Positive integer $P_1, P_2, \ldots, P_n$ (profit)
  $W_1, W_2, \ldots, W_n$ (weight)
  M (capacity)

maximize $\sum_{i=1}^{n} P_i X_i$

subject to $\sum_{i=1}^{n} W_i X_i \le M \quad X_i = 0 \text{ or } 1,\ i = 1, \ldots, n.$

The problem is modified:

minimize $-\sum_{i=1}^{n} P_i X_i$



The Branching Mechanism in the Branch-and-Bound Strategy to Solve 0/1 Knapsack Problem.

How to find the upper bound?

- Ans: by quickly finding a feasible solution in a greedy manner: starting from the smallest available i, scanning towards the largest i's until M is exceeded. The upper bound can be calculated.

The 0/1 knapsack problem

- E.g. n = 6, M = 34

i	1	2	3	4	5	6
P_i	6	10	4	5	6	4
W_i	10	19	8	10	12	8

$$(P_i/W_i \geq P_{i+1}/W_{i+1})$$

- A feasible solution: X1 = 1, X2 = 1, X3 = 0, X4 = 0, X5 = 0, X6 = 0
  -(P1+P2) = -16 (upper bound)
  Any solution higher than -16 can not be an optimal solution.

How to find the lower bound?

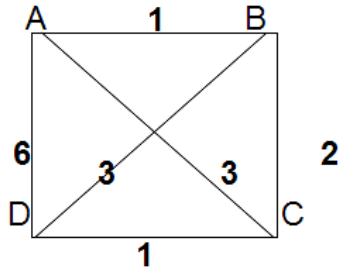
- Ans: by relaxing our restriction from Xi = 0 or 1 to $0 \leq Xi \leq 1$ (knapsack problem)

Let $-\sum_{i=1}^{n} P_i X_i$ be an optimal solution for 0/1 knapsack problem and $-\sum_{i=1}^{n} P_i X'_i$ be an optimal solution for fractional knapsack problem. Let $Y = -\sum_{i=1}^{n} P_i X_i$, $Y' = -\sum_{i=1}^{n} P_i X'_i$.
$\Rightarrow Y' \leq Y$

Approximation Approach
Apply a fast (i.e., a polynomial-time) approximation algorithm to get a solution that is not necessarily optimal but hopefully close to it
Accuracy measures:
accuracy ratio of an approximate solution *sa*
$r(sa) = f(sa) / f(s^*)$ for minimization problems
$r(sa) = f(s^*) / f(sa)$ for maximization problems
where $f(sa)$ and $f(s^*)$ are values of the objective function f for the approximate solution *sa* and actual optimal solution s^*
performance ratio of the algorithm A the lowest upper bound of $r(sa)$ on all instances

Nearest-Neighbor Algorithm for TSP
Starting at some city, always go to the nearest unvisited city, and, after visiting all the cities, return to the starting one

Note: Nearest-neighbor tour may depend on the starting city



s_a : A – B – C – D – A of length 10

s^* : A – B – D – C – A of length 8

Accuracy: RA = ∞ (unbounded above) – make the length of AD arbitrarily large in the above example

Multifragment-Heuristic Algorithm
Stage 1: Sort the edges in nondecreasing order of weights. Initialize the set of tour edges to be constructed to empty set
Stage 2: Add next edge on the sorted list to the tour, skipping those whose addition would've created a vertex of degree 3 or a cycle of length less than *n*. Repeat this step until a tour of length *n* is obtained

Note: RA = ∞, but this algorithm tends to produce better tours than the nearest-neighbor algorithm

Twice-Around-the-Tree Algorithm
Stage 1: Construct a minimum spanning tree of the graph(e.g., by Prim's or Kruskal's algorithm)
Stage 2: Starting at an arbitrary vertex, create a path that goes twice around the tree and returns to the same vertex
Stage 3: Create a tour from the circuit constructed in Stage 2 by making shortcuts to avoid visiting intermediate vertices more than once

Note: RA = ∞ for general instances, but this algorithm tends to produce better tours than the nearest-neighbor algorithm

Christofides Algorithm
Stage 1: Construct a minimum spanning tree of the graph
Stage 2: Add edges of a minimum-weight matching of all the odd vertices in the minimum spanning tree
Stage 3: Find an Eulerian circuit of the multigraph obtained in Stage 2
Stage 3: Create a tour from the path constructed in Stage 2 by making shortcuts to avoid visiting intermediate vertices more than once

RA = ∞ for general instances, but it tends to produce better tours than the twice-around-the-minimum-tree alg.

Euclidean Instances
Theorem If $P \neq NP$, there exists no approximation algorithm for TSP with a finite performance ratio.

Definition An instance of TSP is called *Euclidean*, if its distances satisfy two conditions:
1. *symmetry* $d[i, j] = d[j, i]$ for any pair of cities i and j
2. *triangle inequality* $d[i, j] \le d[i, k] + d[k, j]$ for any cities i, j, k

For Euclidean instances:

approx. tour length / optimal tour length $\le 0.5(\lceil \log_2 n \rceil + 1)$ for nearest neighbor and multifragment heuristic;
approx. tour length / optimal tour length ≤ 2 for twice-around-the-tree;
approx. tour length / optimal tour length ≤ 1.5 for Christofides

Local Search Heuristics for TSP
Start with some initial tour (e.g., nearest neighbor). On each iteration, explore the current tour's neighborhood by exchanging a few edges in it. If the new tour is shorter, make it the current tour; otherwise consider another edge change. If no change yields a shorter tour, the current tour is returned as the output.

Greedy Algorithm for Knapsack Problem
Step 1: Order the items in decreasing order of relative values:
$v_1/w_1 \ge \ldots \ge v_n/w_n$
Step 2: Select the items in this order skipping those that don't fit into the knapsack

Example: The knapsack's capacity is 16

item	weight	value	v/w
1	2	$40	20
2	5	$30	6
3	10	$50	5
4	5	$10	2

Accuracy
- R_A is unbounded (e.g., $n = 2$, $C = m$, $w_1=1$, $v_1=2$, $w_2=m$, $v_2=m$)
- yields exact solutions for the continuous version

Approximation Scheme for Knapsack Problem
Step 1: Order the items in decreasing order of relative values:
$v_1/w_1 \ge \ldots \ge v_n/w_n$
Step 2: For a given integer parameter k, $0 \le k \le n$, generate all subsets of k items or less and for each of those that fit the knapsack, add the remaining items in decreasing order of their value to weight ratios
Step 3: Find the most valuable subset among the subsets generated in Step 2 and return it as the algorithm's output

Bin Packing Problem: First-Fit Algorithm
First-Fit (*FF*) Algorithm: Consider the items in the order given and place each item in the first available bin with enough room for it; if there are no such bins, start a new one
Example: $n = 4$, $s_1 = 0.4$, $s_2 = 0.2$, $s_3 = 0.6$, $s_4 = 0.7$

Accuracy

- Number of extra bins never exceeds optimal by more than 70% (i.e., $R_A \leq 1.7$)
- Empirical average-case behavior is much better. (In one experiment with 128,000 bins, the relative error was found to be no more than 2%.)

Bin Packing: First-Fit Decreasing Algorithm

First-Fit Decreasing (FFD) Algorithm: Sort the items in decreasing order (i.e., from the largest to the smallest). Then proceed as above by placing an item in the first bin in which it fits and starting a new bin if there are no such bins

Example: $n = 4,\ s_1 = 0.4,\ s_2 = 0.2,\ s_3 = 0.6,\ s_4 = 0.7$

Accuracy

- Number of extra bins never exceeds optimal by more than 50% (i.e., $R_A \leq 1.5$)
- Empirical average-case behavior is much better, too

The End

RNS Institute of Technology,

Department of Computer Science & Engg
Bangalore.

SUB: Analysis and Design of Algorithm LAB.

VIVA Questions

1. What is an algorithm?
2. What do you mean by analysis and design of algorithm?
3. Explain algorithm design and analysis process.
4. Explain the differences between sequential algorithms and parallel algorithms.
5. Explain the differences between exact algorithm and approximation algorithm.
6. What is a program?
7. Explain pseudo code and flowchart.
8. Explain two kinds of algorithms efficiency.
9. Explain
    a. Sorting
    b. Searching
    c. String processing
    d. Graph problems
    e. Combinational problems
    f. Geometric problems
    g. Numerical problems
10. Describe Konigsberg bridge puzzle.
11. Explain Hamilton circuit.
12. What is a data structure?
13. Explain different fundamental data structures.
14. Differentiate singly linked list and doubly linked list.
15. Differentiate stacks and queue.
16. What is a graph?
17. Differentiate undirected graph and digraph.
18. Explain different types of graph representations.
19. What is a tree?
20. What is binary tree?
21. What is an ordered tree?
22. Explain worst case, best case and average case efficiencies of an algorithm.
23. Explain different asymptotic notations.
24. Explain O - notation.
25. Explain Ω - notation.
26. Explain Θ - notation.
27. Describe the general plan for analyzing efficiency of non recursive algorithms.
28. Describe the general plan for analyzing efficiency of recursive algorithms.
29. Explain Tower of Hanoi.

30. Explain Fibonacci number series.
31. Describe an algorithm to compute fibonacci numbers.
32. Describe the general plan for empirical analysis of algorithm efficiency.
33. Differentiate static algorithm visualization and dynamic algorithm visualization.
34. Explain brute force approach.
35. Differentiate selection sort and bubble sort.
36. Explain brute force approach in sequential search and string matching.
37. Explain closest pair and convex hull problems by brute force approach.
38. What is exhaustive search?
39. Explain traveling salesman problem.
40. Explain knapsack problem.
41. Explain assignment problem.
42. Which algorithms can be considered by brute force approach?
43. Explain divide and conquer concept.
44. Explain the general strategy used by divide and conquer algorithm.
45. Explain merge sort using divide and conquer technique.
46. Explain quick sort using divide and conquer technique.
47. Explain binary search using divide and conquer technique.
48. Explain binary tree traversals using divide and conquer technique.
49. Explain multiplication of large integers and strassen's matrix multiplication.
50. Explain closest pair and convex hull problem by divide and conquer technique.
51. Explain the Seive of Eratosthenis with an eg.
52. Define:
    a) Time Efficiency
    b) Space Efficiency
53. What is Order of Growth?
54. What is Exhaustive Search?
55. What is Traveling Salesman Problem (TSP)? Explain with an eg.
56. Explain Divide and Conquer technique and give the general divide and conquer recurrence.
57. Define:
    a) Eventually non-decreasing function
    b) Smooth function
    c) Smoothness rule
    d) Master's Theorem
58. Discuss the different methods of generating Permutations.
59. Discuss the different methods of generating Subsets.
60. What is Heap? What are the different types of heaps?
61. Explain the concept of input enhancement in String Matching
62. What is Hashing? Explain with an eg.
63. Explain the concept of Dynamic programming with an eg.
64. What is n-Queen's problem? Generate the state space tree for n = 4.
65. Explain the subset sum problem with an eg.
66. What are Decision Trees? Explain.
67. Define P, NP and NP-Complete problems.
68. Explain the Branch and Bound technique with an eg.
69. Give the Approximation Algorithm for NP-Hard problems.
70. Explain the concept of Greedy technique.