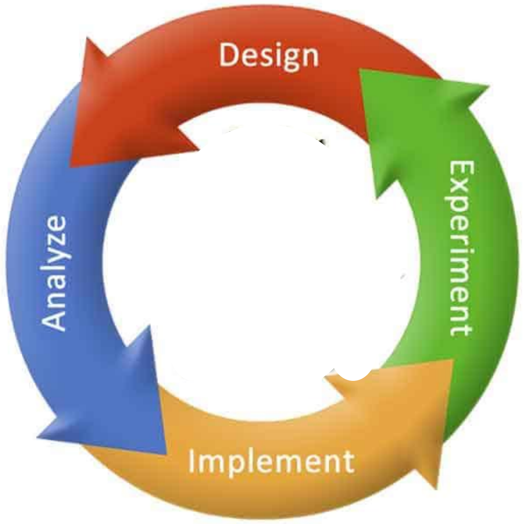
ATME College of Engineering

ANALYSIS AND DESIGN OF ALGORITHM
BCS401



Dr. Nasreen Fathima
Associate Professor & Head
Dept. of Computer Science & Design
ATMECE, Mysuru

Course Outcomes(COs)

1. Apply asymptotic notational method to analyze the performance of the algorithms in terms of time complexity.
2. Demonstrate divide & conquer approaches and decrease & conquer approaches to solve computational problems.
3. Make use of transform & conquer and dynamic programming design approaches to solve the given real world or complex computational problems.
4. Apply greedy and input enhancement methods to solve graph & string based computational problems.
5. Analyze various classes (P,NP and NP Complete) of problems
6. Illustrate backtracking, branch & bound and approximation methods.

What is an algorithm?

Algorithmic: The sprit of computing – David Harel.

Another reason for studying algorithms is their usefulness in developing analytical skills.

Algorithms can be seen as special kinds of solutions to problems – not answers but rather precisely defined procedures for getting answers.

Recipe, process, method, technique, procedure, routine,… with the following requirements:

1. Finiteness
    - terminates after a finite number of steps
2. Definiteness
    - rigorously and unambiguously specified
3. Clearly specified input
    - valid inputs are clearly specified
4. Clearly specified/expected output
    - can be proved to produce the correct output given a valid input
5. Effectiveness
    - steps are sufficiently simple and basic

Algorithm

- Can be represented in various forms
- Unambiguity/clearness
- Effectiveness
- Finiteness/termination
- Correctness

Module 1

Introduction: What is an Algorithm? It's Properties. Algorithm Specification-using natural language, using Pseudo code convention, Fundamentals of Algorithmic Problem solving, Analysis Framework-Time efficiency and space efficiency, Worst-case, Best-case and Average case efficiency.

What is an algorithm?

Algorithmic: The sprit of computing – David Harel.

Another reason for studying algorithms is their usefulness in developing analytical skills.

Algorithms can be seen as special kinds of solutions to problems – not answers but rather precisely defined procedures for getting answers.

Euclid's Algorithm

Problem: Find gcd(m,n), the greatest common divisor of two nonnegative, not both zero integers m and n

Examples: gcd(60,24) = 12, gcd(60,0) = 60, gcd(0,0) = ?

Euclid's algorithm is based on repeated application of equality

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

m mod n is the reminder of the division m &n.

until the second number becomes 0, which makes the problem trivial.

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

Sieve of Eratosthenes

Input: Integer $n \geq 2$

Output: List of primes less than or equal to n

for $p \leftarrow 2$ to n do $A[p] \leftarrow p$

for $p \leftarrow 2$ to sqrt(n) do

    if $A[p] \neq 0$ //p hasn't been eliminated on previous passes

      $j \leftarrow p * p$

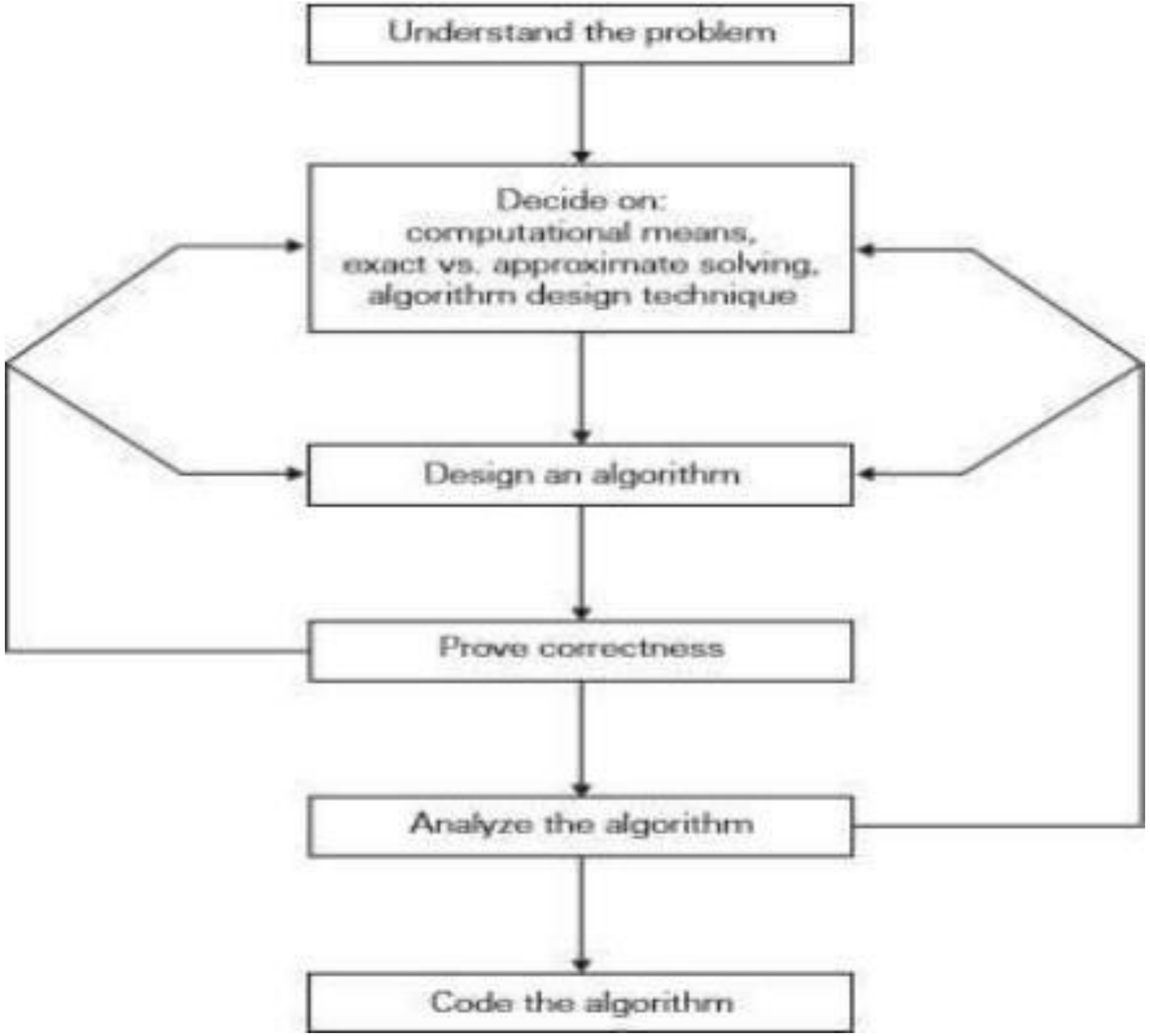
     while $j \leq n$ do

        $A[j] \leftarrow 0$ //mark element as eliminated

        $j \leftarrow j + p$

Example: 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Output: 2 3 5 7 11 13 17 19


Understand the problem
Decide on:
computational means,
exact vs. approximate solving,
algorithm design technique
Design an algorithm
Prove correctness
Analyze the algorithm
Code the algorithm

Fundamental steps in solving problems

- ✓ Statement of the problem
- ✓ Development of mathematical model
- ✓ Design of the algorithm
- ✓ Correctness of the algorithm
- ✓ Analysis of algorithm for its time and space complexity
- ✓ Implementation
- ✓ Program testing and debugging
- ✓ Documentation

Fundamental steps in solving problems

- ✓ Statement of the problem
- ✓ Development of mathematical model
- ✓ Design of the algorithm
- ✓ Correctness of the algorithm
- ✓ Analysis of algorithm for its time and space complexity
- ✓ Implementation
- ✓ Program testing and debugging
- ✓ Documentation

Important problem types

- Sorting
- Searching
- String processing
- Graph problems
- Combinatorial problems
- Geometric problems
- Numerical problems

- Informal definition
  - A graph is a collection of points called vertices, some of which are connected by line segments called edges.
- Modeling real-life problems
  - Modeling WWW
  - Communication networks
  - Project scheduling …
- Examples of graph algorithms
  - Graph traversal algorithms
  - Shortest-path algorithms
  - Topological sorting

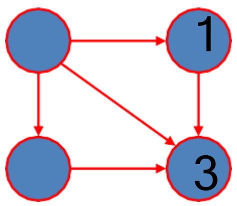
ATME College of Engineering

Linear Data Structures

- Arrays
  - A sequence of n items of the same data type that are stored contiguously in computer memory and made accessible by specifying a value of the array's index.
- Linked List
  - A sequence of zero or more nodes each containing two kinds of information: some data and one or more links called pointers to other nodes of the linked list.
  - Singly linked list (next pointer)
  - Doubly linked list (next + previous pointers)

- Arrays
  - fixed length (need preliminary reservation of memory)
  - contiguous memory locations
  - direct access
  - Insert/delete
- Linked Lists
  - dynamic length
  - arbitrary memory locations
  - access by following links
  - Insert/delete

- Cycle
  - A simple path of a positive length that starts and ends a the same vertex.
- Acyclic graph
  - A graph without cycles
  - DAG (Directed Acyclic Graph)



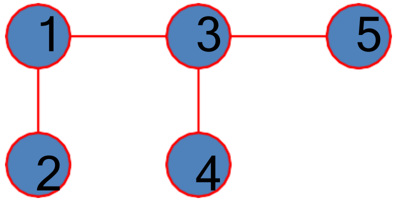
2

4

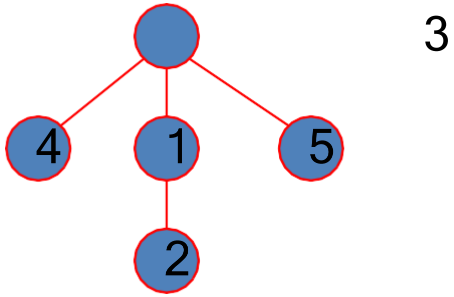
Trees

- Trees
  - A tree (or free tree) is a connected acyclic graph.
  - Forest: a graph that has no cycles but is not necessarily connected.
- Properties of trees
  - For every two vertices in a tree there always exists exactly one simple path from one of these vertices to the other. Why?
    - Rooted trees: The above 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.
    - Levels in a rooted tree.

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



rooted

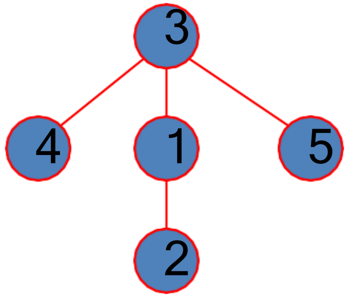


Rooted Trees (I)

- Ancestors
  - For any vertex *v* in a tree *T*, all the vertices on the simple path from the root to that vertex are called ancestors.
- Descendants
  - All the vertices for which a vertex *v* is an ancestor are said to be descendants of *v*.
- Parent, child and siblings
  - If *(u, v)* is the last edge of the simple path from the root to vertex *v*, *u* is said to be the parent of *v* and *v* is called a child of *u*.
  - Vertices that have the same parent are called siblings.
- Leaves
  - A vertex without children is called a leaf.
- Subtree
  - A vertex *v* with all its descendants is called the subtree of *T* rooted at *v*.

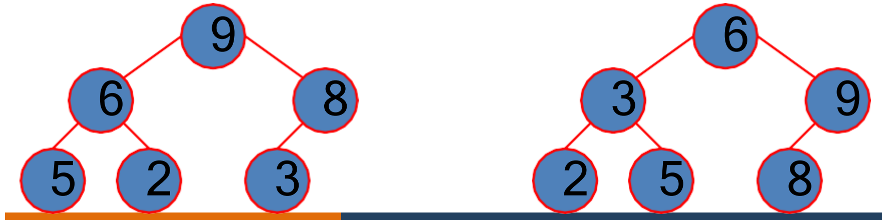
- Depth of a vertex
  - The length of the simple path from the root to the vertex.
- Height of a tree
  - The length of the longest simple path from the root to a leaf.

h = 2



Ordered Trees

- Ordered trees
  - An ordered tree is a rooted tree in which all the children of each vertex are ordered.
- Binary trees
  - A binary tree is an ordered tree in which every vertex has no more than two children and each children is designated s either a left child or a right child of its parent.
- Binary search trees
  - Each vertex is assigned a number.
  - A number assigned to each parental vertex is larger than all the numbers in its left subtree and smaller than all the numbers in its right subtree.
- $\lfloor \log_2 n \rfloor \leq h \leq n - 1$, where h is the height of a binary tree and n the size.



1	constant
$\log n$	logarithmic
n	linear
$n \log n$	n-log-n
n^2	quadratic
n^3	cubic
2^n	exponential
$n!$	factorial

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}		

Table 2.1 Values (some approximate) of several functions important for analysis of algorithms

Order of growth

- Most important: Order of growth within a constant multiple as $n \rightarrow \infty$

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

Best-case, average-case, worst-case

For some algorithms efficiency depends on form of input:

- Worst case: $C_{worst}(n)$ – maximum over inputs of size n
- Best case: $C_{best}(n)$ – minimum over inputs of size n
- Average case: $C_{avg}(n)$ – "average" over inputs of size n

  - Number of times the basic operation will be executed on typical input.
  - NOT the average of worst and best case.

Asymptotic order of growth

A way of comparing functions that ignores constant factors and small input sizes

- O(g(n)): class of functions f(n) that grow <u>no faster</u> than g(n)
- Θ(g(n)): class of functions f(n) that grow <u>at same rate</u> as g(n)
- Ω(g(n)): class of functions f(n) that grow <u>at least as fast</u> as g(n)

Establishing order of growth using the definition

Definition: $f(n)$ is in $O(g(n))$ if order of growth of $f(n) \leq$ order of growth of $g(n)$ (within constant multiple),
i.e., there exist positive constant c and non-negative integer n_0 such that

$$f(n) \leq c\, g(n) \text{ for every } n \geq n_0$$

Example:

- $5n+2$ is $O(n)$; $c = 7$ and $n_0 = 1$

Note : The Upper Bound indicates that the function will be the worst case that it does not consume more than this computing time.

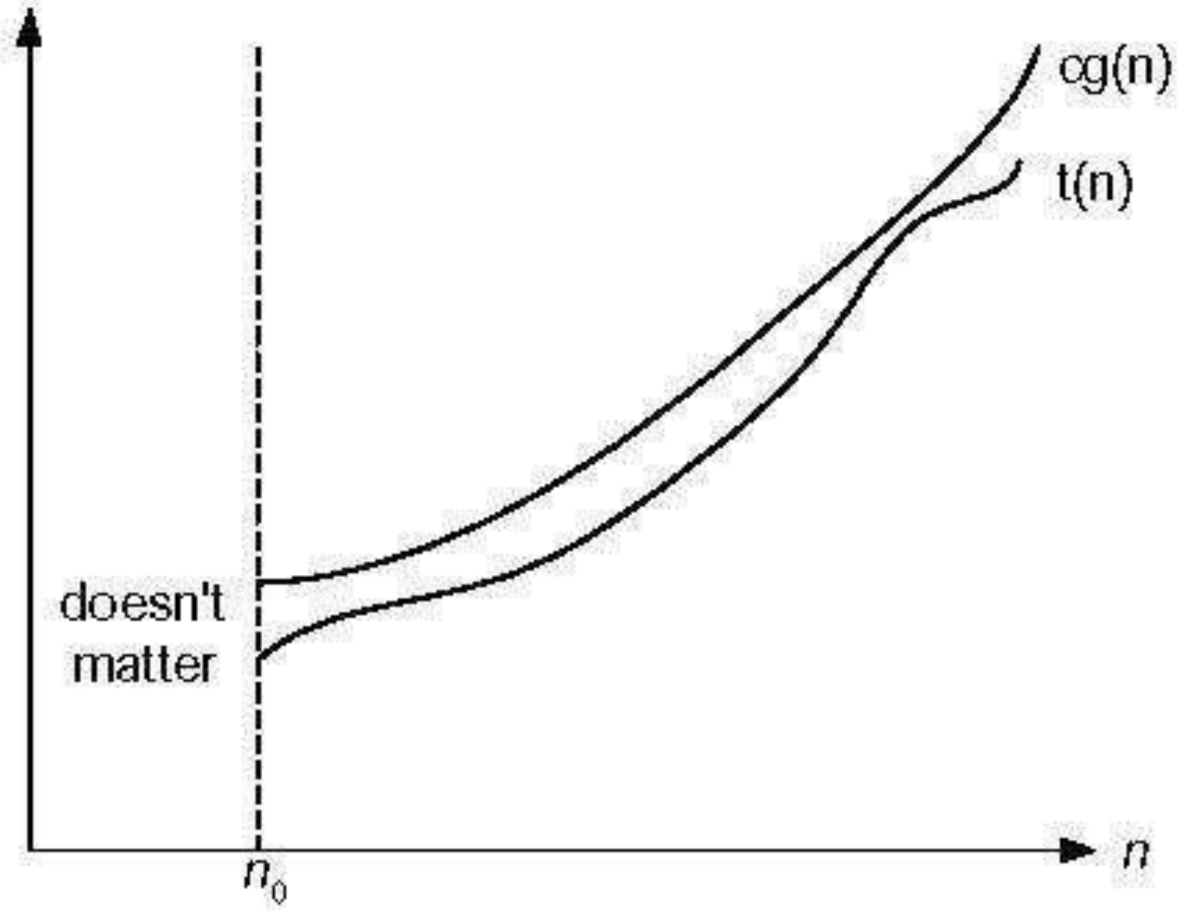


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Definition: $f(n)$ is in $\Omega(g(n))$ if order of growth of $f(n) \geq$ order of growth of $g(n)$ (within constant multiple),
i.e., there exist positive constant c and non-negative integer n_0 such that

$$f(n) \geq c\, g(n) \text{ for every } n \geq n_0$$

Example:

- $5n+2$ is $\Omega(n)$; $c = 5$ and $n_0 = 1$

Big-omega

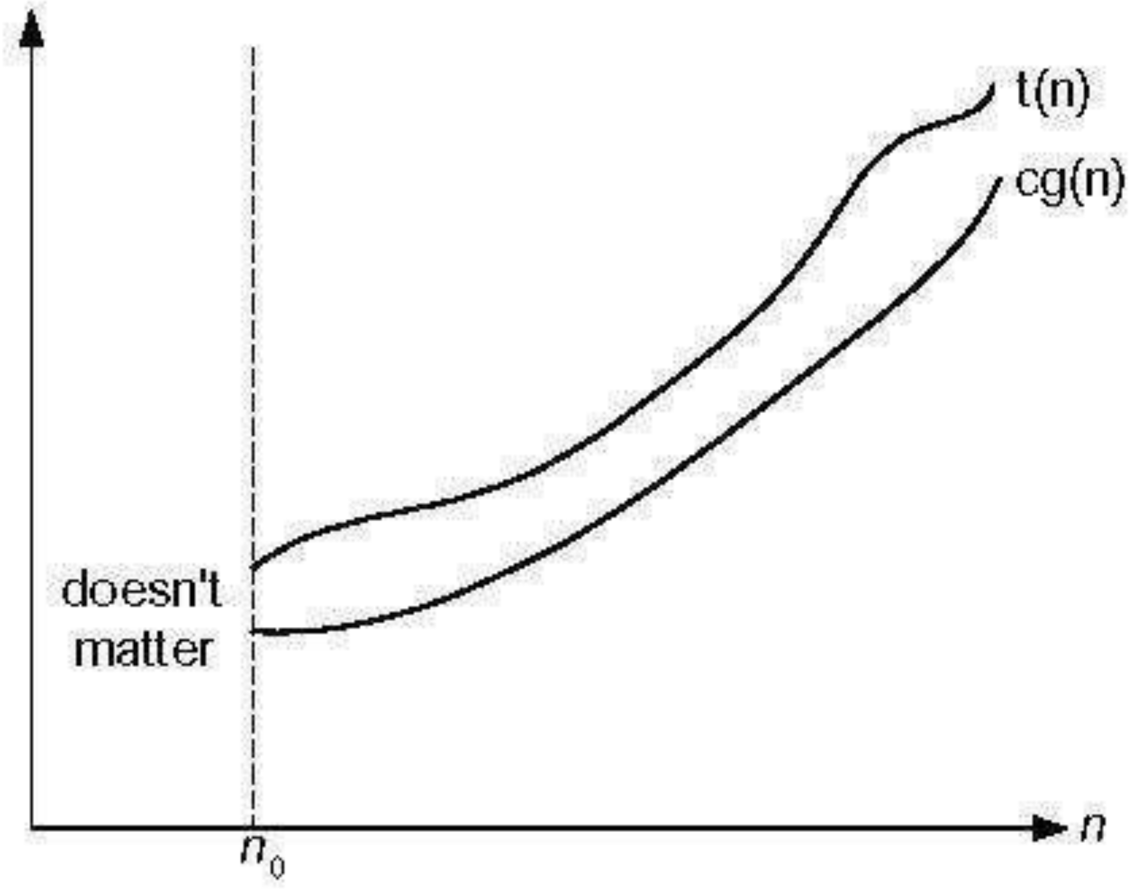


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Establishing order of growth using the definition

Definition: $f(n)$ is in $\Theta(g(n))$ iff there exists three positive constants c1,c2 and n_0 with the constraint that c1 $g(n) \leq f(n) \leq$ c2 $g(n)$ for every $n \geq n_0$.

Example:

- $3n+2$ is $\Theta(n)$
- c1 $g(n) \leq f(n) \leq$ c2 $g(n)$ for every $n \geq n_0$
- $3n \leq 3n+2 \leq 4n$ for every $n_0 = 2$, c1 = 3, c2 = 4

Big-theta

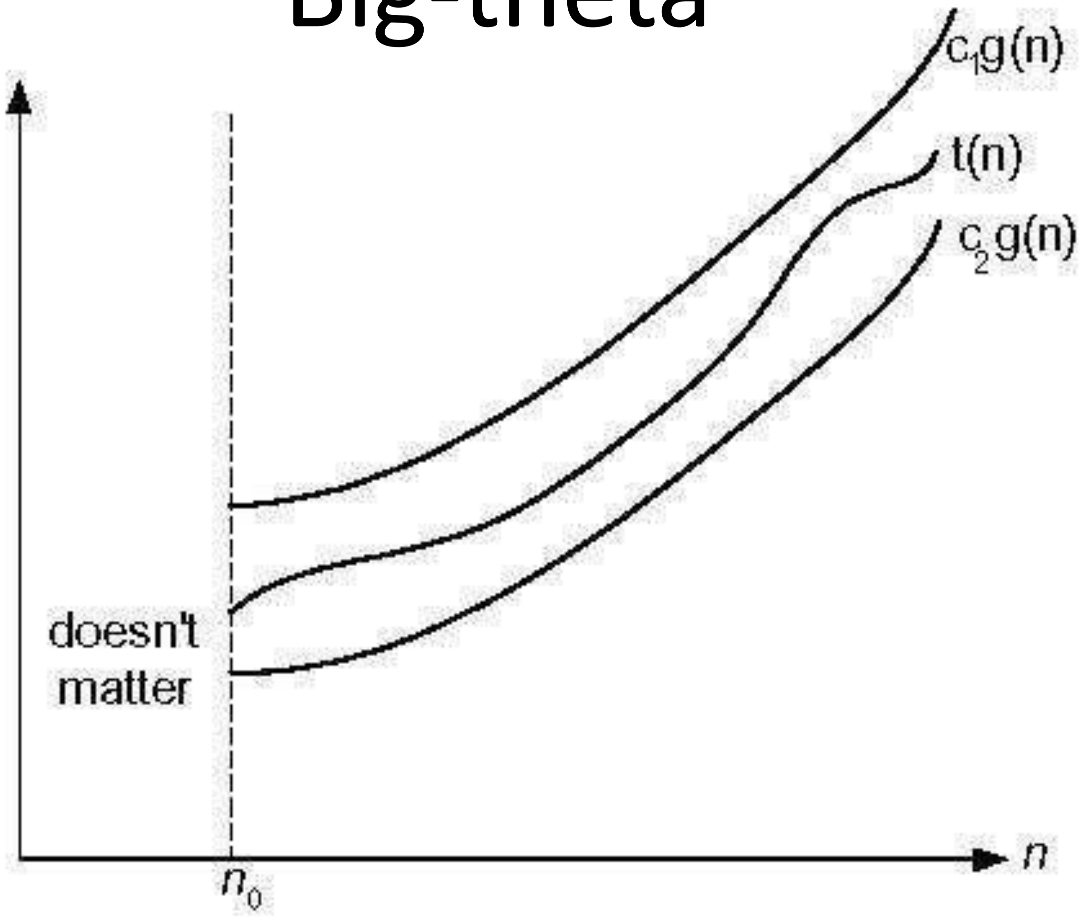


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Properties of asymptotic order of growth

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

- $f(n) \in O(g(n))$ iff $g(n) \in \Omega(f(n))$

- If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$

  Note similarity with $a \leq b$

- If $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$, then

$$f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

General Plan for Analysis

- Decide on parameter *n* indicating ***input size***
- Identify algorithm's ***basic operation***
- Determine ***worst***, ***average***, and ***best*** cases for input of size *n*
- Set up a sum for the number of times the basic operation is executed
- Simplify the sum using standard formulas and rules

Establishing order of growth using limits

$$\lim_{n\to\infty} T(n)/g(n) = \begin{cases} 0 & \text{order of growth of } T(n) < \text{order of growth of } g(n) \\ c > 0 & \text{order of growth of } T(n) = \text{order of growth of } g(n) \\ \infty & \text{order of growth of } T(n) > \text{order of growth of } g(n) \end{cases}$$

Examples:

- **$10n$ vs. n^2**
- **$n(n+1)/2$ vs. n^2**

Example: Sequential search

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

//search for a given value in a given array by sequential search

//Input: An array A[0..n-1] and a search key k

//Output: The index of the first elements of A that matches k or -1 if there are no matching element .

	θ	Type equation here.		
for I ← 0 to n do	θ n	1	n/2	
If (A[i] == k)	n	1	n/2	
{				
found;	1 0	1	1	0
break;				
}				
not found	0 1	0	0	1

Worst case - O(n)

Average case – Θ(n/2)

Best case - Ω(n)

Example 1: Maximum element

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

$maxval \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
    if $A[i] > maxval$
        $maxval \leftarrow A[i]$
return $maxval$

Analysis

1. Input parameter : n
2. Basic operation:

   Comparison

   A[i] > max

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

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

Example 2: Element uniqueness problem

ALGORITHM *UniqueElements*$(A[0..n-1])$

//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
//   and "false" otherwise

for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[i] = A[j]$ **return false**
return true

Analysis

1. Input parameter is input size n
2. Basic operation: Comparison   A[i] == A[j]

3.

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

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

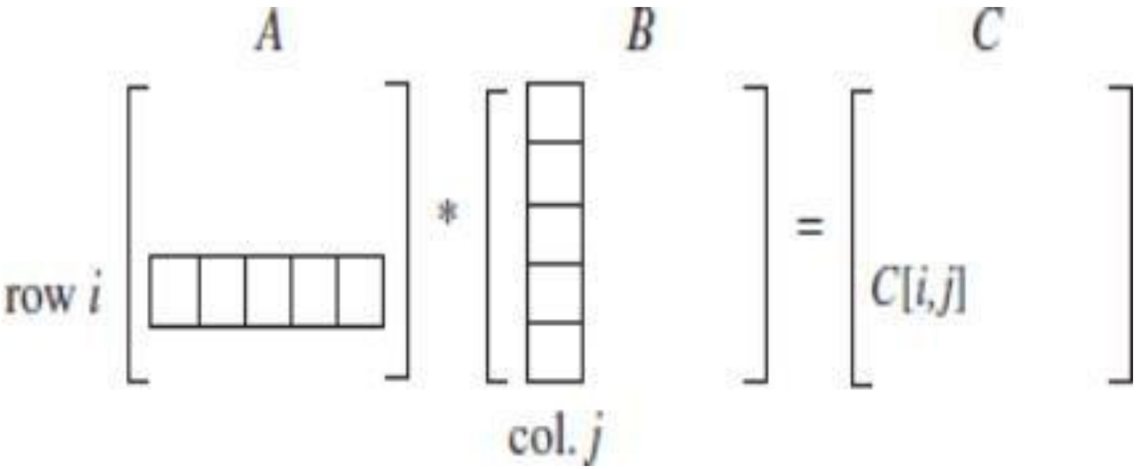
$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2).$$

4. We also could have computed the sum $\sum_{i=0}^{n-2}(n-1-i)$ faster as follows:

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

Example 3: Matrix multiplication

```
ALGORITHM  MatrixMultiplication(A[0..n − 1, 0..n − 1], B[0..n − 1, 0..n − 1])
    //Multiplies two n-by-n matrices by the definition-based algorithm
    //Input: Two n-by-n matrices A and B
    //Output: Matrix C = AB
    for i ← 0 to n − 1 do
        for j ← 0 to n − 1 do
            C[i, j] ← 0.0
            for k ← 0 to n − 1 do
                C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
    return C
```



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

Analysis

1. Input parameter is input size n^2 X n^2
2. Basic operation: Comparison C[i,j] == C[i,j] + A[i,k] * B[k,j]
3.

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

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

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

Now, we can compute this sum by using formula (S1) and rule (R1) given above. Starting with the innermost sum $\sum_{k=0}^{n-1} 1$, which is equal to n (why?), we get

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

$\in O(n^3)$

Selection Sort

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

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

Time efficiency: $\Theta(n^2)$ comparisons (in the worst case)

Solve recurrence relations

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

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

$X(n) = x(n-2) + 5+5$

$= x(n-2) + 2 *5$

$X(n) = x(n-3) + 3*5$

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

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

$= O(n-1) \quad = O(n)$

Plan for Analysis of Recursive Algorithms

- Decide on a parameter indicating an input's size.
- Identify the algorithm's basic operation.
- Check whether the number of times the basic op. is executed may vary on different inputs of the same size. (If it may, the worst, average, and best cases must be investigated separately.)
- Set up a recurrence relation with an appropriate initial condition expressing the number of times the basic op. is executed.
- Solve the recurrence (or, at the very least, establish its solution's order of growth) by backward substitutions or another method.

Example 1: Recursive evaluation of n!

Definition: $n\,! = 1 * 2 * \ldots *(n-1) * n$ for $n \geq 1$ and $0! = 1$

Recursive definition of n!: $F(n) = F(n-1) * n$ for $n \geq 1$ and

$$F(0) = 1$$

ALGORITHM $F(n)$

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

Size:

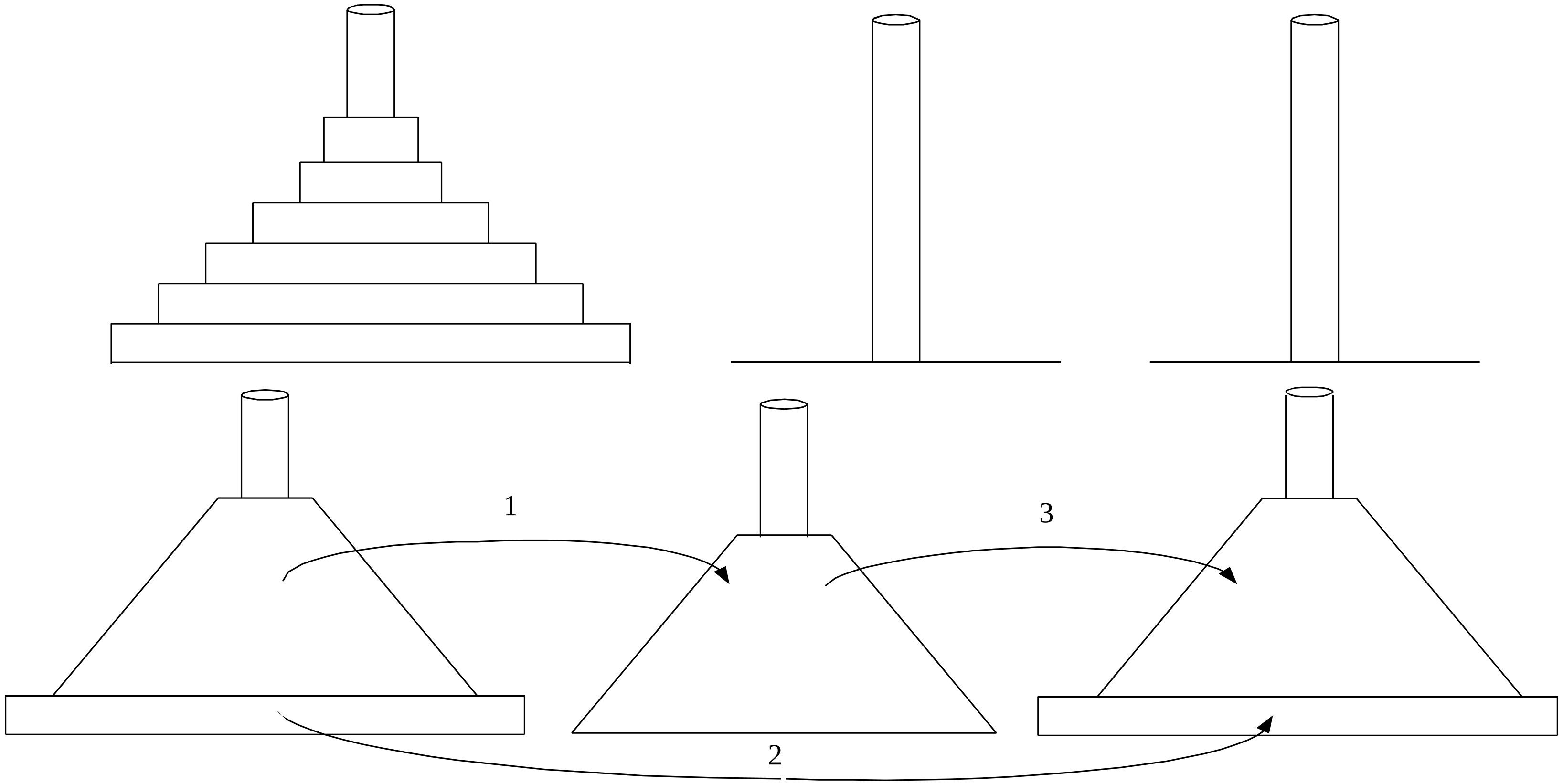
Basic operation: Recurrer

relation:

Tower of Hanoi Problem

- In this problem, we have n disks of different sizes and three pegs.
- Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top.
- The goal is to move all the disks to the third peg, using the second one as an auxiliary if necessary.
- We can move only one disk at a time, and it is forbidden to place a larger disk on top of a smaller one.

Example 2: The Tower of Hanoi Puzzle



Recurrence for number of moves:

Algorithm : TOH (n, S, T, D)

```
// Solving Tower of Hanoi Problems
// Input : Number of discs n
// Output : The sequence of movements.
{
    if n > 0
    {
        TOH (n - 1, S, D, T);
        move disk from S to D
        TOH (n - 1, T, S, D)
    }
}
```

Solving recurrence for number of moves

$M(n) = 2M(n-1) + 1, \ M(0) = 0$

$2[2M(n-2)+1]+1$

$2^2M(n-2) +2 +1$

$2^2 [2M(n-3) +1]+2 +1$

$2^3 M(n-3) + 2^2+2 +1$

...

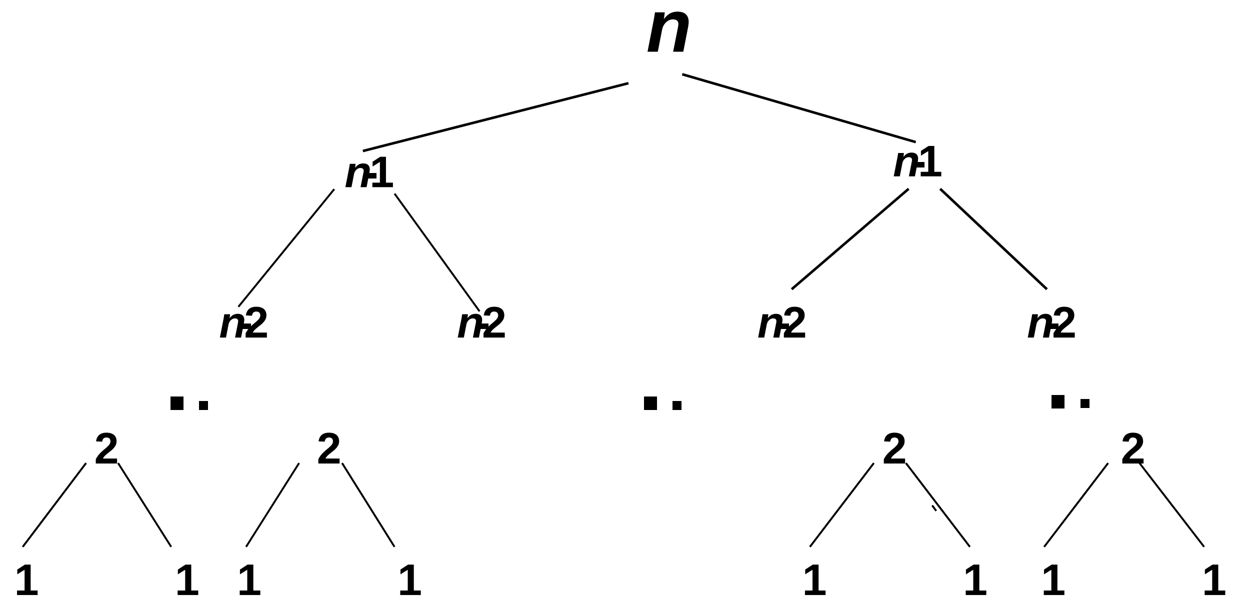
$2^n M(n-n) + 2^{n-1} +2^{n-2} + ... 2^2+2 +1$

Standard formula used is G.P sequence

$$(1 + 2 + 2^2 + \cdots 2^{n.-2} + 2^{n-1}) = \frac{a(r-1)}{r-1}$$

Here a=1 and r=2 => $O(2^n)$

Tree of calls for the Tower of Hanoi Puzzle



Fibonacci numbers

The Fibonacci numbers:

0, 1, 1, 2, 3, 5, 8, 13, 21, …

The Fibonacci algorithm (recursive)

```
Fib(n)
{
If n<=1
     return n
Else
Return F(n-1) + F(n-2)
```

- The recurrence equation for this problem is:

$T(n) = T(n-1) + T(n-2)$ for $n>1$ and the initial conditions are $T(0) =0$, $T(1) = 1$

Solution to recurrence relation:

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

$T(n) - T(n-1) - T(n-2) = 0$

This is of the form $ax(n) +bx(n-1) +cx(n-2) =0$

Which is a homogeneous second order linear

- Where a = 1, b= -1, c=-1

Consider it as a quadratic equation

$ar^2 + br + c = 0$

Then the roots of the equation

$r^2 - r - 1 = 0$

$r1,2 = (1 \pm \sqrt{5})/2$

These roots r1 and r2 are real and distinct.

The recurrence relation can be given as

$$T(n) = \propto r_1^{\,n} + \beta r_2^{\,n}$$

$$T(n) = \propto [(1+\sqrt{5})/2\,]^n + \beta[(1-\sqrt{5})/2\,]^n$$

Substituting T(0) = 0

$$= \propto [(1+\sqrt{5})/2\,]^0 + \beta[(1-\sqrt{5})/2\,]^0$$

$$= \propto = -\beta \quad \text{or } \beta = -\propto$$

T(1) = 1

$$= \propto [(1+\sqrt{5})/2\,]^1 + \beta[(1-\sqrt{5})/2\,]^1 = 1$$

$$\propto = 1/\sqrt{5} \text{ and } \beta = -1/\sqrt{5}$$

Little oh Notation (o)

- The asymptotic upper bound provided by O-notation may or may not be asymptotically tight. The bound $2n^2 = O(n^2)$ is asymptotically tight but the bound $2n = o(n^2)$ is not.
- We use o-notation to denote an upper bound that is not asymptotically tight
- f(n) = o(g(n)); f(n) is equal to the little oh of g(n), iff $f(n) < c, g(n)$ for any +ve constant $c>0$, $n_0>0$ and $n>n_0$

Establishing order of growth using limits

$$\lim_{n\to\infty} T(n)/g(n)$$

0 order of growth of < order of growth of

***c* > 0** order of growth of = order of growth of

∞ order of growth of >order of growth of

Examples:

10*n* vs. n^2

5*n* + 2 vs. *n*

Property of the Asymptotic Notations

1. Theorem : If $t1(n) \in O(g1(n))$ and $t2(n) \in O(g2(n))$, then

   $t1(n) + t2(n) \in O(\max\{g1(n), (g2(n)\})$

2. Theorem: If $f(n) = a_m n^m + - - - - + a_1 n + a_0$ and $a_m > 0$, then $f(n) = O(n^m)$

Brute Force

A straightforward approach, usually based directly on the problem's statement and definitions of the concepts involved

Examples:

1. Computing a^n (a > 0, n a nonnegative integer)

2. Computing n!

3. Multiplying two matrices

4. Searching for a key of a given value in a list

Brute-Force Sorting Algorithm

<u>*Selection Sort*</u> Scan the array to find its smallest element and swap it with the first element. Then, starting with the second element, scan the elements to the right of it to find the smallest among them and swap it with the second elements. Generally, on pass i ($0 \leq i \leq n$-2), find the smallest element in $A[i..n-1]$ and swap it with $A[i]$:

$$A[0] \leq \ . \ . \ . \ \leq A[i-1] \ | \ A[i], \ . \ . \ . \ , A[min], . \ . \ ., A[n-1]$$

in their final positions

Example: 7 3 2 5

Analysis of Selection Sort

ALGORITHM *SelectionSort*$(A[0..n-1])$

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

Time efficiency: $\Theta(n^2)$

In place: **Yes**

Stability: yes

Brute-Force String Matching

- *pattern*: a string of *m* characters to search for
- *text*: a (longer) string of *n* characters to search in
- problem: find a substring in the text that matches the pattern

Brute-force algorithm

Step 1 Align pattern at beginning of text

Step 2 Moving from left to right, compare each character of pattern to the corresponding character in text until

- all characters are found to match (successful search); or
- a mismatch is detected

Step 3 While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

Examples of Brute-Force String Matching

1. Pattern: 001011

   Text: 10010101101001100101111010

2. Pattern: happy

   Text: It is never too late to have a happy childhood.

Pseudocode and Efficiency

ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

//Implements brute-force string matching
//Input: An array $T[0..n-1]$ of n characters representing a text and
// an array $P[0..m-1]$ of m characters representing a pattern
//Output: The index of the first character in the text that starts a
// matching substring or -1 if the search is unsuccessful
for $i \leftarrow 0$ **to** $n-m$ **do**
  $j \leftarrow 0$
  while $j < m$ **and** $P[j] = T[i+j]$ **do**
    $j \leftarrow j+1$
  if $j = m$ **return** i
return -1

Time efficiency: **Θ(mn) comparisons (in the worst case)**

Brute-Force Strengths and Weaknesses

- **Strengths**
  - wide applicability
  - simplicity
  - yields reasonable algorithms for some important problems (e.g., matrix multiplication, sorting, searching, string matching)

- **Weaknesses**
  - rarely yields efficient algorithms
  - some brute-force algorithms are unacceptably slow
  - not as constructive as some other design techniques

DIVIDE AND CONQUER

Divide-and-Conquer

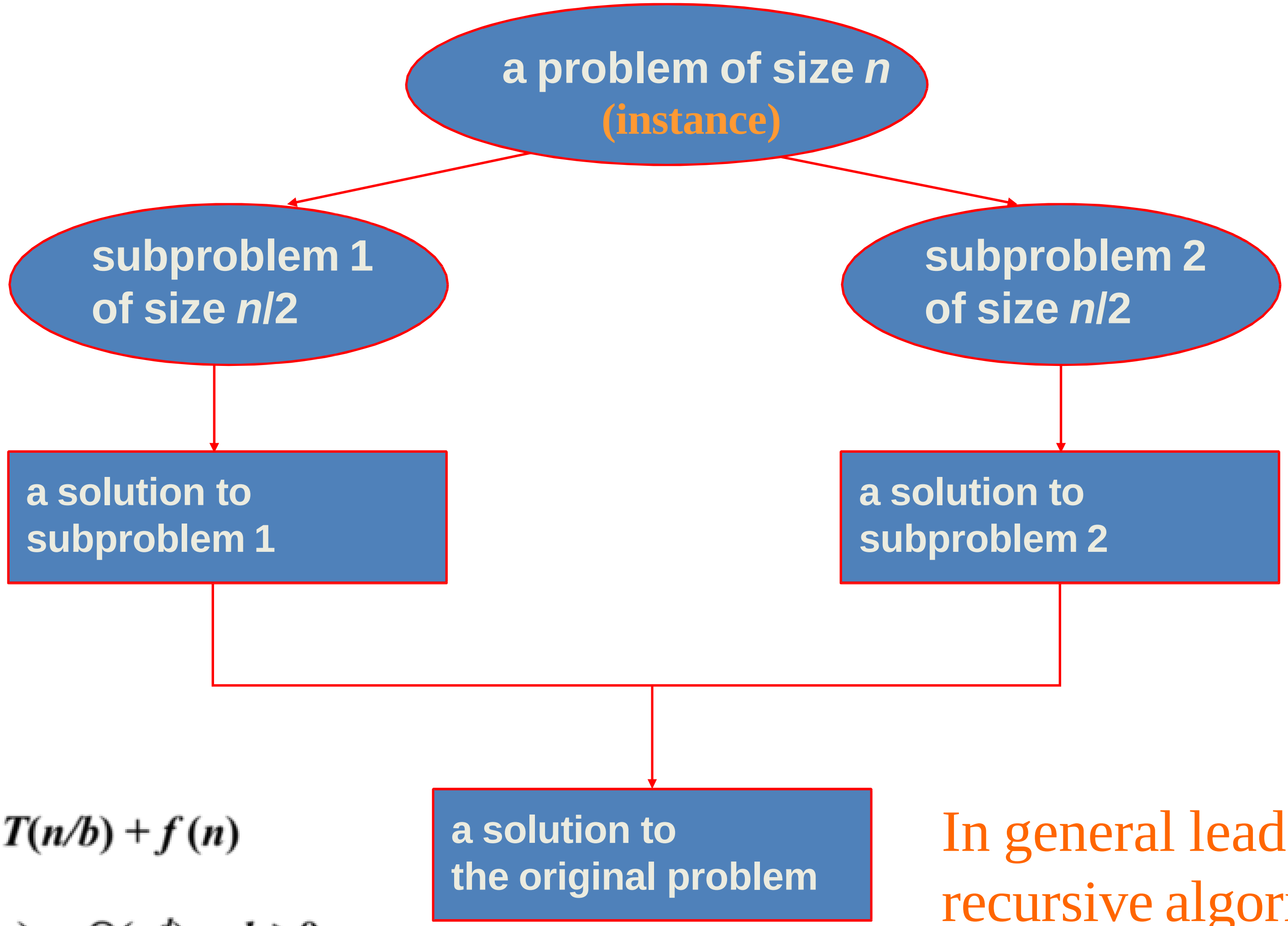
The most-well known algorithm design strategy

1. Divide instance of problem into two or more smaller instances
2. Solve smaller instances recursively
3. Obtain solution to original (larger) instance by combining these solutions

Divide and conquer involves three steps, at each level of recursion

- **Divide:** Divide the problem into a number of sub problems
- **Conquer:** Conquer the sub problems by solving them recursively. If the sub – problem sizes are small enough, then solve the sub- problem in a straight forward manner.
- **Combine:** combine the solutions to the sub- problems to get the solution to the original problem.

Divide-and-Conquer Technique (cont.)



$$T(n) = a\,T(n/b) + f(n)$$

where $f(n) \in \Theta(n^d),\quad d \geq 0$

In general leads to a recursive algorithm!

Divide-and-Conquer Examples

- Sorting: merge sort and quicksort
- Finding min and max element in an array
- Binary search
- Multiplication of large integers
- Matrix multiplication: Strassen's algorithm

General Divide-and-Conquer Recurrence

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

Master Theorem: If $a < b^d$, $T(n) \in \Theta(n^d)$

If $a = b^d$, $T(n) \in \Theta(n^d \log n)$

If $a > b^d$,

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

Note: The same results hold with O instead of Θ.

Examples: $T(n) = 4T(n/2) + n \Rightarrow T(n) \in ?$ Θ(n^2)

$T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ?$ Θ(n^2log n)

$T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ?$ Θ(n^3)

Merge Sort Algorithm

Mergesort(low, high)

//Given an array A of n elements. This algorithm sorts the elements in //ascending order. The variables low and high are used to identify the //positions of first and last element in each partition.

1. *If (low< high)*
2. *mid = (low+high)/2;*
3. *Mergesort (low,mid);*
4. *Mergesort(mid+1,high);*
5. *Merge(low,mid,high);*
6. *End if*
7. *Exit*

Merge Algorithm

Merge(low, mid, high)

/ The variables low, mid, and high are used to identify the portions of elements in each partition.

```
1.  Initialize i=low, j= mid+1, h=low;
2.      while ((h <= mid) && (j <= high))
3.      if (a[h] < a[j])
         b[i++] = a[h++];
         else
         b[i++] = a[j++];
```

Cont…

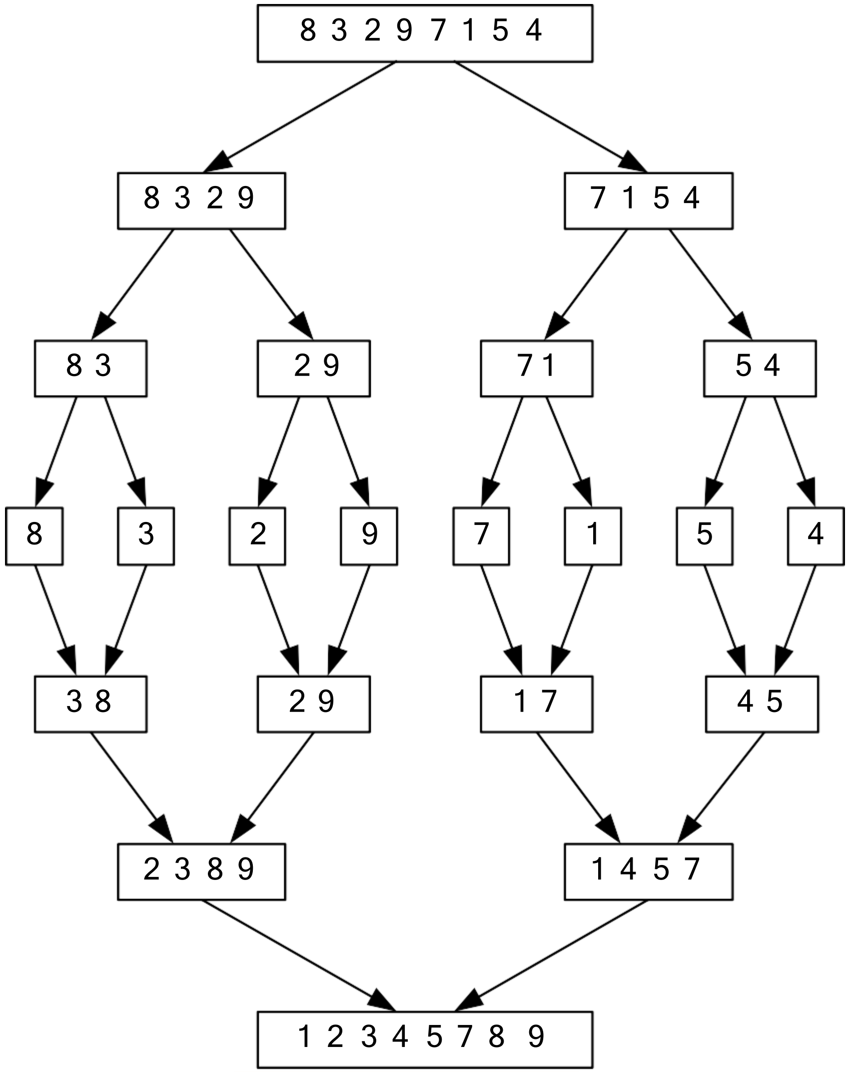
```
4.      if (h > mid)
            for(k = j; k <= high; k++)
                b[i++] = a[k];
        else
            for (k = h; k <= mid; k++)
                b[i++] = a[k];
5.      for (k = low; k <= high; k++)
        a[k] = b[k];
```

Mergesort

- Split array A[0..*n*-1] into about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:

  - Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Mergesort Example



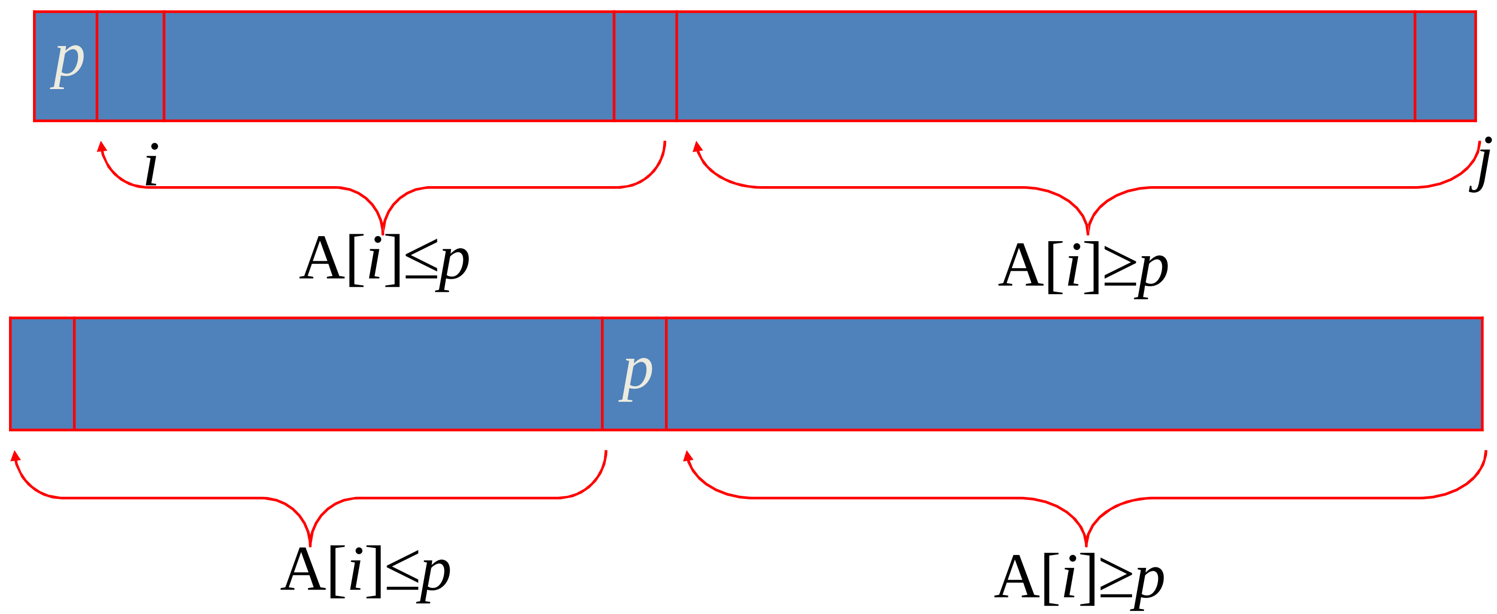
The non-recursive version of Mergesort starts from merging single elements into sorted pairs.

- All cases have same efficiency: $\Theta(n \log n)$
- Number of comparisons in the worst case is close to theoretical minimum for comparison-based sorting:

$$\lceil \log_2 n! \rceil \approx n \log_2 n - 1.44n$$

- Space requirement: $\Theta(n)$ (<u>not</u> in-place)
- Can be implemented without recursion

Quicksort

- Select a *pivot* (partitioning element) – as the first element
- 

- Exchange the pivot with the last element in the first (i.e., ≤) subarray — the pivot is now in its final position
- Sort the two subarrays recursively
- Note : Invented by

Partition Algorithm

Partition(low, high)

//This procedure partitions the element into two lists and places the pivot
//element into a appropriate place. Low = first element of the array, high =
//last element of the array, a[low] = pivot.

Step 1. Set pivot = a[low];

i = low +1;

j = high;

Step 2. Repeat step 3 while (*a[i] < pivot && i < high)*

Step 3. i++;

Step 4. Repeat step 5 while (a[j] > pivot)

Step 5. j--;

Step 6. If(i<j)

swap a[i] and a[j]

go to step 2

else

swap a[j] and pivot

Step 7. Return (j)

Quicksort Example

5 3 1 9 8 2 4 7

2 3 1 4 5 8 9 7

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

Analysis of Quicksort

- Best case: split in the middle — $\Theta(n \log n)$
- Worst case: sorted array! — $\Theta(n^2)$
- Average case: random arrays — $\Theta(n \log n)$

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

- Improvements:
  - better pivot selection: median of three partitioning
  - switch to insertion sort on small subfiles
  - elimination of recursion

  These combine to 20-25% improvement

- Considered the method of choice for internal sorting of large files ($n \geq 10000$)

Binary Search

Algorithm Binary_Search(A[0…n-1], Key)

Input: Given an array of n elements in sorted order and key is an element to be searched.

Output: Returns the position of key element, if successful and returns -1 otherwise.

```
1.   Set first = 0, last = n-1
2.   While ( first <= last )
     mid = (first +last) / 2
     if (key == A[mid])
        return (mid+1); // successful
     else if ( key < A[mid] )
          last = mid – 1
     else
          first = mid+1
     end while
3.   return -1 // unsuccessful
```

Analysis

Best Case: Best case occurs, when we are searching the middle element itself. In that case, total number of comparisons required is 1. there fore best case time complexity of binary search is Ω(1).

Worst Case: Let T(n) be the cost involved to search ‘n’ elements. Let T(n/2) be the cost involved to search either left part or the right part of an array.

Analysis

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

$T(n/2)$ → Time required to search either the left part or the right part of the array.

b → Time required to compare the middle element.

Where a and b are some positive integer constants.

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

Analysis

Average Case:

The average case occurs when an element is found some where in the recursive calls, but not till the recursive call ends.

The average number of key comparisons made by binary search is only slightly smaller than that in this worst case.

$T(n) = \log_2 n$

The average number of comparison in a successful search is

$T(n) = \log_2 n - 1$

The average number of comparison in a unsuccessful search is

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

Analysis

- This algorithm requires 2(n-1) element comparisons in the best, average, and worst cases.
- Now the Best case occurs when the elements are in increasing order. The number of element comparisons is n-1.
- The worst case occurs when the element are in decreasing order. In this case number of comparisons is 2(n-1).

Finding maximum and minimum using divide and conquer technique

Algorithm max_min(i, j, max, min)

{

// Input: a[1:n] is a global array. Parameters i and j are integers, 1<=i <= j<= n.

// output: to set max and min to the largest and smallest values in a[i: j], respectively.

If (i == j) then // Small(P)

{ max = min ← A[i];

}

```
else if ( i =j-1) then   // Another case of Small(P)
{
        if (A[i] < A[j]) then
        {
            max ← A[j]
            min ← A[i]
        }
        else
        {
            max ← A[i]
            min ← A[j]
        }
}
```

```
else
{
  // if P is not small, divide P into sub problems.
  // Find where to split the set
 mid := (i+j)/2;
 // Solve the sub problems.
max_min(i,mid,max,min); max_min(mid+1, j,
max1,min1);
// Combine the solutions
  if( max < max1) then max := max1; if( min >
  min1) then min:= min1;
  }
}
```

Analysis

$$T(n) = \begin{cases} 0 & n=1 \\ 1 & n=2 \\ T(n/2) + T(n/2) + 2 & n>2 \end{cases}$$

When n is a power of two, $n = 2^k$ for some positive integer k, then

$$\begin{aligned}
T(n) &= 2T(n/2) + 2 \\
&= 2T(2^{k-1}) + 2 \\
&= 2(2T(2^{k-2}) + 2) + 2 \\
&= 2^2T(2^{k-2}) + 2^2 + 2 \\
&= 2^3T(2^{k-3}) + 2^3 + 2^2 + 2 \\
&\;\;\ldots \\
&= 2^{k-1}\,T(2^{k-(k-1)}) + 2^{k-2} + 2^{k-1} + - - - + 2^1 \\
&= 2^{k-1} + 2^{k-2} + - - - + 2^1 \\
&= 2.\,(2^{k-1} - 1)/\,2-1 = \mathbf{O(n)}
\end{aligned}$$

Multiplication of Large Integers

Consider the problem of multiplying two (large) n-digit integers represented by arrays of their digits such as:

A = 12345678901357986429   B = 87654321284820912836

The grade-school algorithm:

$$
\begin{array}{r}
a_1\ a_2 \ldots\ a_n \\
b_1\ b_2 \ldots\ b_n \\
\hline
(d_{10})\ d_{11} d_{12} \ldots d_{1n} \\
(d_{20})\ d_{21} d_{22} \ldots d_{2n} \\
\ldots \ldots \ldots \ldots \ldots \ldots \ldots \\
(d_{n0})\ d_{n1} d_{n2} \ldots d_{nn} \\
\hline
\end{array}
$$

Efficiency: $\Theta(n^2)$ single-digit multiplications

First Divide-and-Conquer Algorithm

A small example: $A * B$ where $A = 2135$ and $B = 4014$

$A = (21 \cdot 10^2 + 35)$, $B = (40 \cdot 10^2 + 14)$

So, $A * B = (21 \cdot 10^2 + 35) * (40 \cdot 10^2 + 14)$

$= 21 * 40 \cdot 10^4 + (21 * 14 + 35 * 40) \cdot 10^2 + 35 * 14$

In general, if $A = A_1A_2$ and $B = B_1B_2$ (where A and B are n-digit, A_1, A_2, B_1, B_2 are $n/2$-digit numbers),

$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$

Recurrence for the number of one-digit multiplications $M(n)$:

$$M(n) = 4M(n/2), \quad M(1) = 1$$

Solution: $M(n) = n^2$

Second Divide-and-Conquer Algorithm

$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$

The idea is to decrease the number of multiplications from 4 to 3:

$(A_1 + A_2) * (B_1 + B_2) = A_1 * B_1 + (A_1 * B_2 + A_2 * B_1) + A_2 * B_2,$

I.e., $(A_1 * B_2 + A_2 * B_1) = (A_1 + A_2) * (B_1 + B_2) - A_1 * B_1 - A_2 * B_2,$
which requires only 3 multiplications at the expense of (4-1) extra add/sub.

Recurrence for the number of multiplications M(n):

$$M(n) = 3M(n/2), \quad M(1) = 1$$

Solution: $M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}$

What if we count both multiplications and additions?

Example of Large-Integer Multiplication

2135 * 4014

= (21*10^2 + 35) * (40*10^2 + 14)

= (21*40)*10^4 + c1*10^2 + 35*14

where c1 = (21+35)*(40+14) - 21*40 - 35*14, and

21*40 = (2*10 + 1) * (4*10 + 0)

= (2*4)*10^2 + c2*10 + 1*0

where c2 = (2+1)*(4+0) - 2*4 - 1*0, etc.

This process requires 9 digit multiplications as opposed to 16.

Matrix Multiplication

- Brute-force algorithm

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} * \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$= \begin{pmatrix} a_{11} * b_{11} + a_{12} * b_{21} & a_{11} * b_{12} + a_{12} * b_{22} \\ a_{21} * b_{11} + a_{22} * b_{21} & a_{21} * b_{12} + a_{22} * b_{22} \end{pmatrix}$$

8 multiplications

4 additions

Efficiency class in general: $\Theta(n^3)$

Strassen's Matrix Multiplication

- Strassen's algorithm for two 2x2 matrices (1969):

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} * \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$= \begin{pmatrix} C1 = E + I + J - G & C2 = D + G \\ C3 = E + F & C4 = D + H + J - F \end{pmatrix}$$

D = A1(B2 – B4)

E = A4(B3 – B1)

F = (A3 + A4) B1

G = (A1 + A2) B4

H = (A3 – A1) (B1 + B2)

I = (A2 – A4) (B3 +B4)

J = (A1 +A4)(B1 +B4)

7 multiplications

18 additions

Strassen's Matrix Multiplication

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad B = \begin{pmatrix} 1 & 1 \\ 2 & 2 \end{pmatrix}$$

A1 = 1, A2 =2, A3 = 3, A4 = 4

B1 = 1, B2 = 2, B3 = 2, B4 = 2

1. D = A1(B2 – B4) = 1(1 – 2) = -1
2. E = A4(B3-B1) = 4(2-1) = 4
3. F = (A3 + A4) B1 = (3+4)1 = 7
4. G = (A1 + A2)B4 = (1+2)2 = 6

5. H = (A3 – A1) (B1 + B2) = (3-1)(1+1) = 4

6. I = (A2 – A4)(B3+B4) = (2-4)(2+2) = -8

7. J = (A1+A4)(B1+B4) = (1+4)(1+2) = 15

C1 = E +I+J-G = 4+(-8) +15-6 = 5

C2 = D + G = -1 +6 = 5

C3 = E + F = 4 + 7 = 11

C4 = D + H + J – F = -1 +4 +15 -7 = 11

$$C = \begin{pmatrix} 5 & 5 \\ 11 & 11 \end{pmatrix}$$

Strassen's Matrix Multiplication

Strassen observed [1969] that the product of two matrices can be computed in general as follows:

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} * \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$= \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{pmatrix}$$

Formulas for Strassen's Algorithm

$M_1 = (A_{00} + A_{11}) * (B_{00} + B_{11})$

$M_2 = (A_{10} + A_{11}) * B_{00}$ $M_3 = A_{00} *$

$(B_{01} - B_{11})$ $M_4 =$ $A_{11} * (B_{10} - B_{00})$

$M_5 = (A_{00} + A_{01}) * B_{11}$

$M_6 = (A_{10} - A_{00}) * (B_{00} + B_{01})$

$M_7 = (A_{01} - A_{11}) * (B_{10} + B_{11})$

$$\begin{array}{cc} A00 & A01 \\ \begin{Bmatrix} 1 \end{Bmatrix} & \begin{Bmatrix} 2 \end{Bmatrix} \\ \begin{Bmatrix} 3 \end{Bmatrix} & \begin{Bmatrix} 4 \end{Bmatrix} \\ A10 & A11 \end{array} \qquad \begin{array}{cc} B00 & B01 \\ \begin{Bmatrix} 1 \end{Bmatrix} & \begin{Bmatrix} 1 \end{Bmatrix} \\ \begin{Bmatrix} 2 \end{Bmatrix} & \begin{Bmatrix} 2 \end{Bmatrix} \\ B10 & B11 \end{array}$$

$M_1 = (A_{00} + A_{11}) * (B_{00} + B_{11})$
$= (1 + 4) * (1+2) = 15$

$M_2 = (A_{10} + A_{11}) * B_{00}$
$= (3 + 4) * 1 = 7$

$M_3 = A_{00} * (B_{01} - B_{11})$
$= 1 * (1 - 2) = -1$

$M_4 = A_{11} * (B_{10} - B_{00})$
$= 4 * (2 - 1) = 4$

$M_5 = (A_{00} + A_{01}) * B_{11}$
$= (1 + 2) * 2 = 6$

$M_6 = (A_{10} - A_{00}) * (B_{00} + B_{01})$
$= (3 - 1) * (1 + 1) = 4$

$M_7 = (A_{01} - A_{11}) * (B_{10} + B_{11})$
$= (2 - 4) * (2 + 2) = -8$

$$\begin{array}{cc} C00 & C01 \\ \begin{Bmatrix} M_1 + M_4 - M_5 + M_7 \end{Bmatrix} & \begin{Bmatrix} M_3 + M_5 \end{Bmatrix} \\ \begin{Bmatrix} M_2 + M_4 \end{Bmatrix} & \begin{Bmatrix} M_1 + M_3 - M_2 + M_6 \end{Bmatrix} \\ C10 & C11 \end{array} = \begin{Bmatrix} 5 \end{Bmatrix} \begin{Bmatrix} 5 \end{Bmatrix} \atop \begin{Bmatrix} 11 \end{Bmatrix} \begin{Bmatrix} 11 \end{Bmatrix}$$

$$\begin{array}{cc} A00 & A01 \\ \begin{pmatrix} 2 \end{pmatrix} & \begin{pmatrix} 1 \end{pmatrix} \\ \begin{pmatrix} 3 \end{pmatrix} & \begin{pmatrix} 4 \end{pmatrix} \\ A10 & A11 \end{array} \qquad \begin{array}{cc} B00 & B01 \\ \begin{pmatrix} 5 \end{pmatrix} & \begin{pmatrix} 2 \end{pmatrix} \\ \begin{pmatrix} 1 \end{pmatrix} & \begin{pmatrix} 2 \end{pmatrix} \\ B10 & B11 \end{array}$$

$$M_1 = (A_{00} + A_{11}) * (B_{00} + B_{11})$$
$$= (2 + 4) * (5+2) = 42$$

$$M_2 = (A_{10} + A_{11}) * B_{00}$$
$$= (3 + 4) * 5 = 35$$

$$M_3 = A_{00} * (B_{01} - B_{11})$$
$$= 2 * (2 - 2) = 0$$

$$M_4 = A_{11} * (B_{10} - B_{00})$$
$$= 4 * (1 - 5) = -16$$

$$M_5 = (A_{00} + A_{01}) * B_{11}$$
$$= (2 + 1) * 2 = 6$$

$$M_6 = (A_{10} - A_{00}) * (B_{00} + B_{01})$$
$$= (3 - 2) * (5 + 2) = 7$$

$$M_7 = (A_{01} - A_{11}) * (B_{10} + B_{11})$$
$$= (1 - 4) * (1 + 2) = -9$$

$$\begin{array}{cc} C00 & C01 \\ \begin{pmatrix} M_1 + M_4 - M_5 + M_7 \end{pmatrix} & \begin{pmatrix} M_3 + M_5 \end{pmatrix} \\ \begin{pmatrix} M_2 + M_4 \end{pmatrix} & \begin{pmatrix} M_1 + M_3 - M_2 + M_6 \end{pmatrix} \\ C10 & C11 \end{array} = \begin{pmatrix} 11 & 6 \\ 19 & 14 \end{pmatrix}$$

Solve

$$A = \begin{pmatrix} 1 & 0 & 2 & 1 \\ 4 & 1 & 1 & 0 \\ 0 & 1 & 3 & 0 \\ 5 & 0 & 2 & 1 \end{pmatrix} \quad B = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 4 \\ 2 & 0 & 1 & 1 \\ 1 & 3 & 5 & 0 \end{pmatrix}$$

$$A1 = \begin{pmatrix} 1 & 0 \\ 4 & 1 \end{pmatrix} \quad A2 = \begin{pmatrix} 2 & 1 \\ 1 & 0 \end{pmatrix} \quad B1 = \begin{pmatrix} 0 & 1 \\ 2 & 1 \end{pmatrix} \quad B2 = \begin{pmatrix} 0 & 1 \\ 0 & 4 \end{pmatrix}$$

$$A3 = \begin{pmatrix} 0 & 1 \\ 5 & 0 \end{pmatrix} \quad A4 = \begin{pmatrix} 3 & 0 \\ 2 & 1 \end{pmatrix} \quad B3 = \begin{pmatrix} 2 & 0 \\ 1 & 3 \end{pmatrix} \quad B4 = \begin{pmatrix} 1 & 1 \\ 5 & 0 \end{pmatrix}$$

1. D = A1 (B2 - B4)

$$\begin{pmatrix} 1 & 0 \\ 4 & 1 \end{pmatrix} * \left\{ \begin{pmatrix} 0 & 1 \\ 0 & 4 \end{pmatrix} - \begin{pmatrix} 1 & 1 \\ 5 & 0 \end{pmatrix} \right\}$$

$$\begin{pmatrix} 1 & 0 \\ 4 & 1 \end{pmatrix} * \begin{pmatrix} -1 & 0 \\ -5 & 4 \end{pmatrix}$$

$$\begin{pmatrix} -6 & 0 \\ -9 & 4 \end{pmatrix}$$

2. E = A4 (B3 - B1)

Analysis of Strassen's Algorithm

If n is not a power of 2, matrices can be padded with zeros.

Number of multiplications:

$$M(n) = 7M(n/2), \quad M(1) = 1$$

$$M(n) = 7M(2^{k-1})$$

$$= 7[7M(2^{k-2})] = 7^2 M(2^{k-2})]$$

$$= 7^k M(2^{k-k})] = 7^k \ (1)$$

Solution: $M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807}$ vs. n^3 of brute-force alg.

Advantages and Disadvantages

- Difficult problems is broken down into sub problems and each sub problem is solved independently.
- It gives efficient algorithms like quick sort, merge sort, streassen's matrix multiplication.
- Sub problems can be executed on parallel processor.

Disadvantage

- It makes use of recursive methods and the recursion is slow and complex.

Decrease-and-Conquer

The decrease and conquer technique is almost similar to the divide and conquer technique, but instead of dividing the problem into size n/2, it is decremented by a constant or constant factor.

There are three variations of decrease and conquer

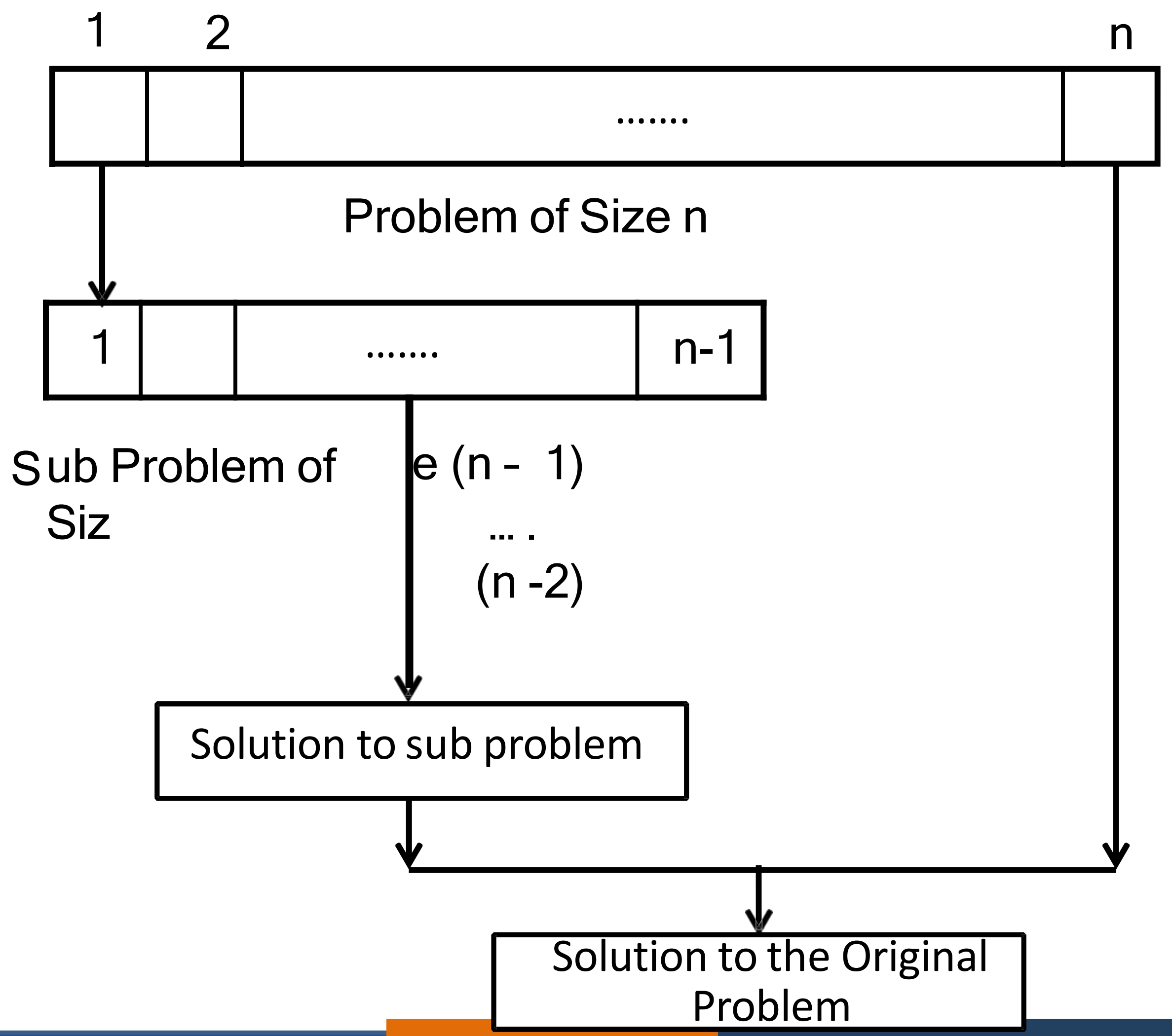
- Decrease by a constant
- Decrease by a constant factor
- Variable size decrease

The problems can be solved either top down (recursively) or bottom up (without recursion)

Decrease by a constant

- In this type of variation, the size of an instance is reduced by the same constant ‘1’ on each iteration. So, if a problem is of size ‘n’ , then a sub problem of size ‘n-1’ is solved first but before a sub sub problem of size ‘n-2’ is solved and so on.

Decrease by a constant



Decrease by a constant

Example: Consider a problem for computing a^n where n is a positive integer exponent

Let $f(n) = a^n$

$$a^n = a^{n-1} . a$$
$$= a^{n-2} . a . a$$
$$= a^{n-3} . a . a . a$$
$$= a. a. a. a. . . n \text{ times}$$

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

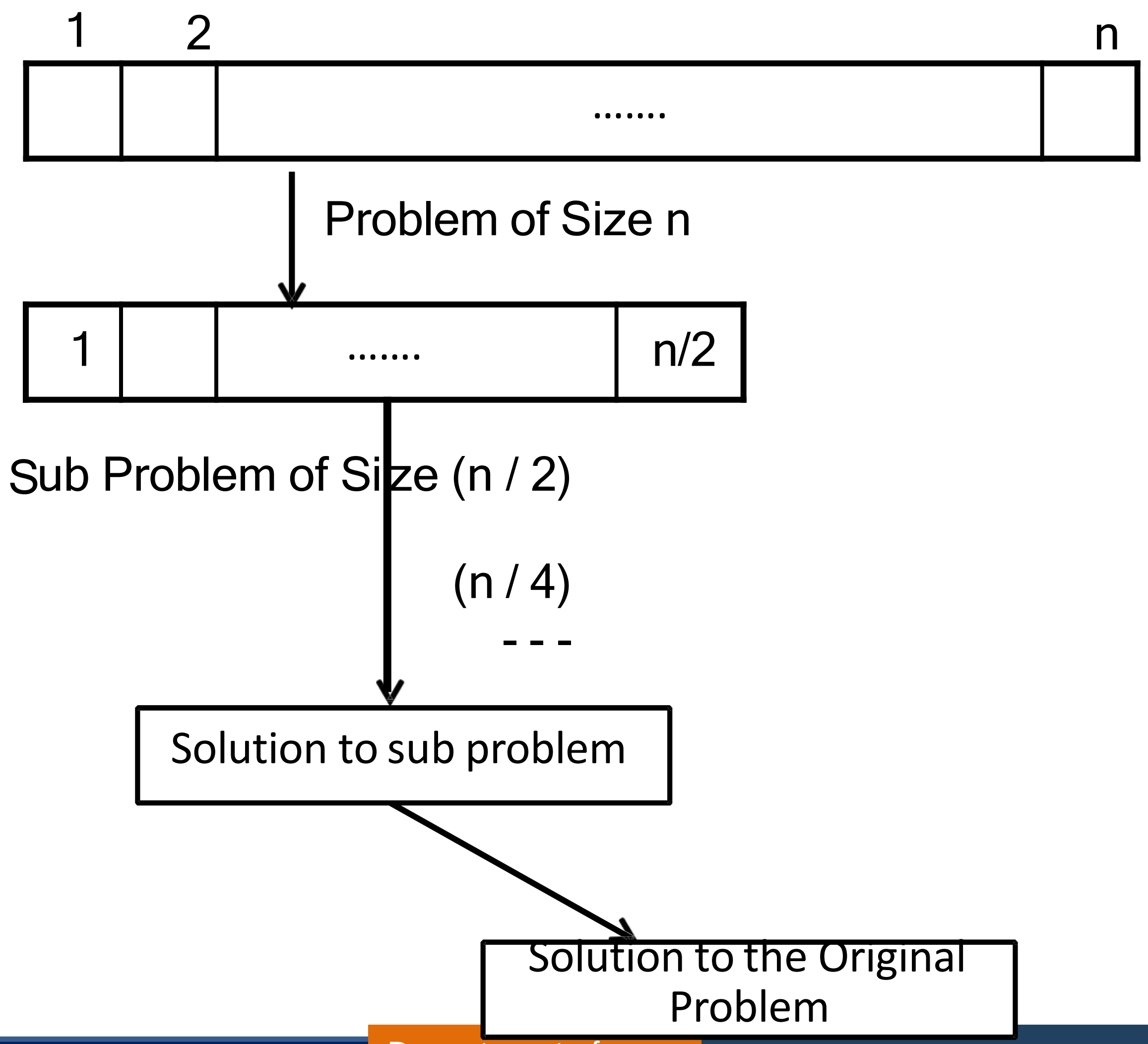
The above definition is a recursive definition i.e, a top down approach

Eg: Insertion sort, Depth First Search, Breath First Search, Topological Sort

Decrease by a constant factor

- In this type of variation, the size of instance is reduced by a constant factor on each iteration (most of the case it is 2).
- So, if a problem of size ‘n’ is to be solved then first the sub problem of size n/2 is to be solved which in-turn requires the solution for the sub sub problem n/4 and so on.

Decrease by a constant factor



Decrease by a constant factor

Example: Consider a problem for computing a^n

As the problem is to be halved each time (Since the constant factor is 2, to solve a^n, first solve $a^{n/2}$, but before solve $a^{n/4}$ and so on.

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

Decrease by a constant factor

The efficiency of this variation i.e decrease by a constant factor is O(log n) because, the size is reduced by at least one half at the expense of no more than two multiplications on each iteration

Eg: Binary search and the method of bisection, Fake coin problem

Variable size decrease

In this type, the reduction in the size of the problem instance is varied from one iteration to another.

Eg: Euclid's algorithm for computing GCD of two nos.

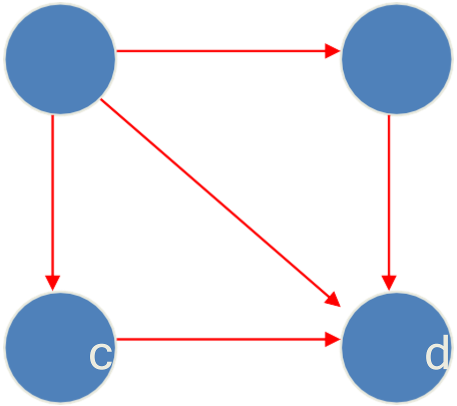
$$gcd(m,n) = \begin{cases} gcd(n, m \bmod n) & \text{if } n > 0 \\ m & \text{if } n = 0 \end{cases}$$

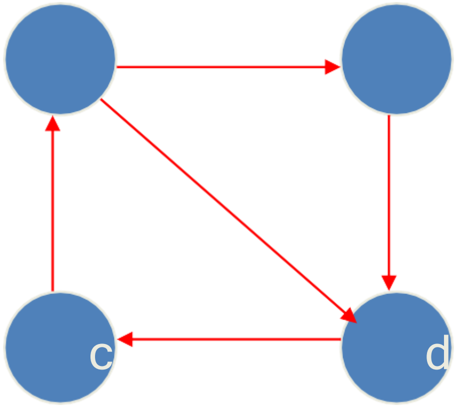
Eg: Computing a median, Interpolation Search and Binary Search Tree

DAGs and Topological Sorting

A *dag*: a directed acyclic graph, i.e. a directed graph with no (directed) cycles

a dag



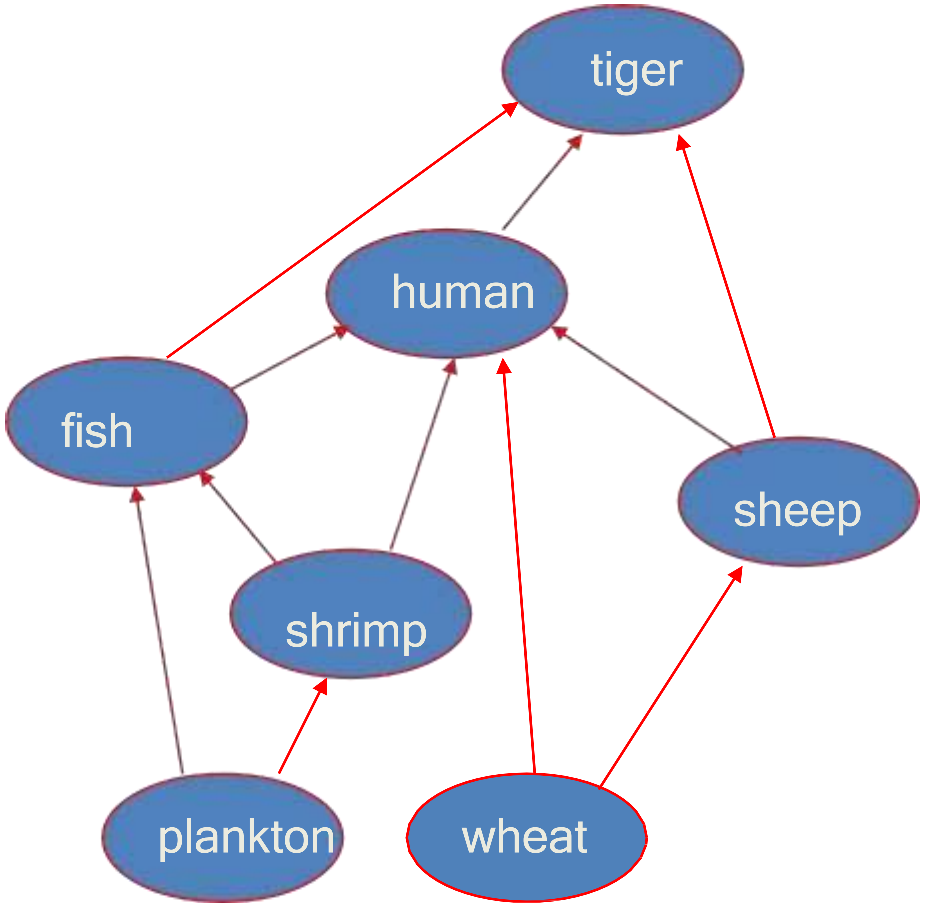


not a dag

Arise in modeling many problems that involve prerequisite constraints (construction projects, document version control)

Vertices of a dag can be linearly ordered so that for every edge its starting vertex is listed before its ending vertex (*topological sorting*). Being a dag is also a necessary condition for topological sorting to be possible.

Topological Sorting Example
Order the following items in a food chain

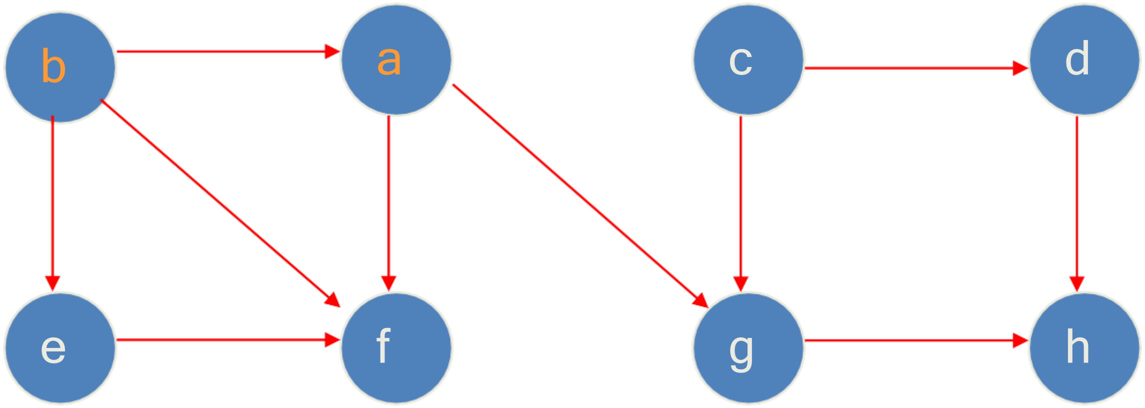


DFS-based Algorithm

DFS-based algorithm for topological sorting

- Perform DFS traversal, noting the order vertices are popped off the traversal stack
- Reverse order solves topological sorting problem
- Back edges encountered?→ NOT a dag!

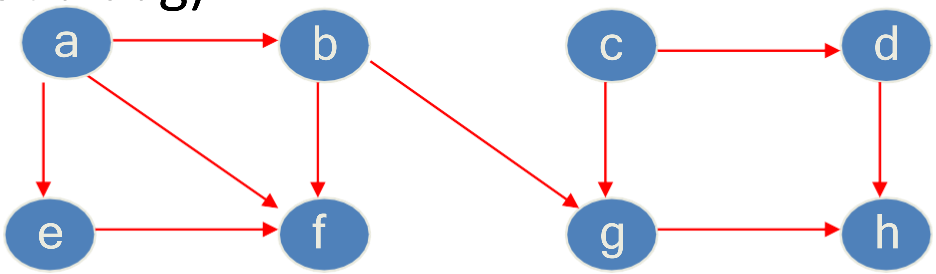
Example:



Source Removal Algorithm

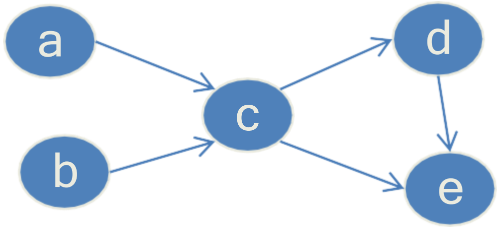
Repeatedly identify and remove a *source* (a vertex with no incoming edges) and all the edges incident to it until either no vertex is left or there is no source among the remaining vertices (not a dag)

Example: 1



Efficiency: same as efficiency of the DFS-based algorithm, but how would you identify a source? How do you remove a source from the dag?

Example 2



Source Removal Algorithm

Topological Sort(G)

1. Find the indegree INDG(n) of each node n of G.
2. Put in a queue Q all the nodes with zero indegree.
3. Repeat step 4 and 5 until G becomes empty.
4. Repeat the element n of the queue Q and add it to T (Set Front = Front +1).

Source Removal Algorithm

5. Repeat the following for each neighbour, m of the node n

   a) Set INDEG(m) = INDG(m)-1

   b) If INDEG(m) = 0 then add m to the rear end of the Q.

6. Exit.

Note: For Problems refer class notes

GREEDY METHOD

Greedy Method

- Approach for Solving problem
- Used for Solving Optimization Problem
- **Optimization Problem** : Problems which demands minimum/maximum results
- Example:

12 hrs

Minimum cost

$A \xrightarrow{\text{12 hrs}} B$

S1 S2 S3 S4 S5……

Optimal solution

Feasible solutions

There will be only one minimum solution

Strategies used for Optimization Problem

- Greedy Method
- Dynamic Programming
- Branch and Bound

Greedy Technique

Greedy algorithms, construct a solution through a sequence of steps, each step expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached.

- ***feasible*** *-it has to satisfy the problem's constraints*
- ***locally optimal*** ***-*** *it has to be the best local choice among all feasible choices available on that step*
- ***irrevocable*** ***-*** *once made, it cannot be changed on subsequent steps of the algorithm*

General method control abstraction

```
Algorithm Greedy(a, n)
// a[1..n] contains the 'n' inputs
{
Solution := 0;       //Initialize the solution
    for i:= 1 to n do
    {
        X : = Select(a);
        If Feasible(Solution, x) then
        Solution:= Union(Solution, x);
    }
Return Solution;
}
```

Applications of the Greedy Strategy

- Optimal solutions:
  - change making for “normal” coin denominations
  - minimum spanning tree (MST)
  - single-source shortest paths
  - simple scheduling problems
  - Huffman codes
- Approximations/heuristics:
  - traveling salesman problem (TSP)
  - knapsack problem
  - other combinatorial optimization problems

Change-Making Problem

- Problem Statement: Given coins of several denominations find out a way to give a customer an amount with fewest number of coins.
- Example: if denominations are 1,5,10, 25 and 100 and the change required is 30, the solutions are,
- Amount : 30
- Solutions : 3 x 10 (3 coins)

  6 x 5 (6 coins)

  1 x 25 + 5 x 1 (6 coins)

  1 x 25 + 1 x 5 (2 coins)

The last solution is the optimal one as it gives us change only with 2 coins.

Change-Making Problem

Given unlimited amounts of coins of denominations $d_1 > \ldots > d_n$, give change for amount n with the least number of coins

Example: $d_1 = 25c$, $d_2 = 10c$, $d_3 = 5c$, $d_4 = 1c$ and $n = 48c$

Greedy solution is optimal for any amount and "normal" set of denominations

Solution: <1, 2, 0, 3>

cashier's Algorithm

Algorithm coinchange()

//Input: Denomination d[1] > d[2] > d*3+… d*n+

//Amount to obtain change – C

// Output: The optimal number of coins for change of C, is stored in Coins[i]

```
for i ← 1 to n do
    {
        Coins[i] = C/d[i];
        C = C mod d[i]
        Print coins[i]
    }
```

{25,10,1} for 30c

n: 3 , C: 30

i=1
Coins[1] : 30|d[1] : 30|25 = 1
C = C % d[i] : 30 % 25 = 5

i=2
Coins[2] : 5|d[2] : 5|10 = 0
C : C % d[i] : 5|10 = 5

i=3
Coins[3] : 5|1 = 5
C = 5 % 1 = 0

Coins {1, 0, 5}

Change-Making Problem

For example, $d1$ = 25c, $d2$ = 10c, $d3$ = 1c, and n = 30c

Solution: <1, 0, 5>

May not be optimal for all denominations

Knapsack Problem (Fractional knapsack problem)

Given n objects and a knapsack or bag. Object i has a weight wi and the knapsack has a capacity m. if the fraction Xi, 0<=Xi<=1, of object i is placed into the knapsack, then a profit of Pi*Xi is earned.

The objective is to maximize the total profit earned. Since the knapsack capacity is m, we require the total weight of all chosen objects to be at most m.

Knapsack Problem - 1

Obtain the optimal solution for the knapsack problem using greedy method given the following:

M = 15

n = 7

p1,p2,p3,p4,p5,p6,p7 = 10,5,15,7,6,18,3

w1,w2,w3,w4,w5,w6,w7= 2,3,5,7,1,4,1

$n=7$, $M=15$

$(P_1, P_2, P_3, P_4, P_5, P_6, P_7) = (10, 5, 15, 7, 6, 18, 3)$

$(w_1, w_2, w_3, w_4, w_5, w_6, w_7) = (2, 3, 5, 7, 1, 4, 1)$

Number of objects $= n = 7$

Capacity of bag $= M = 15$

Object	1	2	3	4	5	6	7
P_i	10	5	15	7	6	18	3
w_i	2	3	5	7	1	4	1
P_i/w_i	5	1.67	3	1	6	4.5	3

There are several greedy methods to obtain the feasible solutions.

Method 1: Select object with Maximum profit (P_i)

Object	profit (P_i)	weight (w_i)	Remaining weight
-	-	-	15
6	18	4	15-4=11
3	15	5	11-5=6
1	10	2	6-2=4
4	4x1=4	4	4-4=0

Object	1	2	3	4	5	6	7
P_i	10	5	15	7	6	18	3
w_i	2	3	5	7	1	4	1
P_i/w_i	5	1.67	3	1	6	4.5	3

Profits = 47

Solution Vector = (1, 0,1, 4/7,0,1,0)= (1, 0,1, 0.57,0,1,0)

Optimal solution using this method is (x1, x2, x3,x4,x5,x6,x7) = (1, 0,1, 0.57,0,1,0) with profit = 47

Method 2: Select Object with minimum weight (w_i)

Object	profit (P_i)	weight (w_i)	Remaining weight
–	–	–	15
5	6	1	15-1=14
7	3	1	14-1=13
1	10	2	13-2=11
2	5	3	11-3=8
6	18	4	8-4=4
3	3X4=12	4	4-4=0

Object	1	2	3	4	5	6	7
P_i	10	5	15	7	6	18	3
w_i	2	3	5	7	1	4	1
P_i/w_i	5	1.67	3	1	6	4.5	3

Profits = 54

Solution Vector = (1, 1,4/5,0,1,1,1)= (1, 1,0.8,0,1,1,1)

Optimal solution using this method is (x1, x2, x3,x4,x5,x6,x7) = (1, 1,0.8,0,1,1,1) with profit = 54

Optimal solution is not guaranteed using method 1 and 2

Method 3: Select object with maximum (Pi/wi)

Object	Profit (P_i)	Weight (w_i)	Remaining weight
-	-	-	15
5	6	1	15-1=14
1	10	2	14-2=12
6	18	4	12-4=8
3	15	5	8-5=3
7	3	1	3-1=2
2	2x1.67=3.34	2	2-2=0

Object	1	2	3	4	5	6	7
P_i	10	5	15	7	6	18	3
w_i	2	3	5	7	1	4	1
P_i/w_i	5	1.67	3	1	6	4.5	3

Profits = $\boxed{55.34}$ Solution Vector = (1, 2/3,1,0, 1,1,1)= (1, 0.67,1,0, 1,1,1)

Optimal solution is (x1, x2, x3,x4,x5,x6,x7) = (1, 0.67,1,0, 1,1,1)

with profit [1*10+0.67*5+1*15+0*7+1*6+1*18+1*3]= **55.34**

Weight=[1*2+0.67*3+1*5+0*7+1*1+1*4+1*1]=**15**

This greedy approach always results optimal solution

Knapsack Problem (Fractional knapsack problem)

Given n objects and a knapsack or bag. Object i has a weight wi and the knapsack has a capacity m. if the fraction Xi, 0<=Xi<=1, of object i is placed into the knapsack, then a profit of Pi*Xi is earned.

The objective is to maximize the total profit earned. Since the knapsack capacity is m, we require the total weight of all chosen objects to be at most m.

Maximize $\sum_{1 \le i \le n} PiXi$

Subject to $\sum_{1 \le i \le n} wiXi \le$ m

The profits and weights are positive numbers.

Knapsack Algorithm

Algorithm Greedy Knapsack(m,n)

//p[1:n] and w[1:n] contain the profits and weights respectively, of the n objects ordered such that p[i]/w[i] >= p[i+1]/w[i+1].

// m is the knapsack size and x[1:n] is the solution vector

```
{
for i := 1 to n do x[i] := 0.0; //Initialize x
U := m;//sack capacity
for i:=1 to n do
{
    if (w[i] > U) then break; // weight of an object is greater than sack capacity
    x[i] := 1.0; U:=U-w[i];
}
If(i<=n) then x[i]:=U/w[i];
}
```

Analysis: Disregarding the time to initially sort the object, each of the above strategies use O(n) time

Problem - 2

Obtain the optimal solution for the knapsack problem using greedy method given the following:

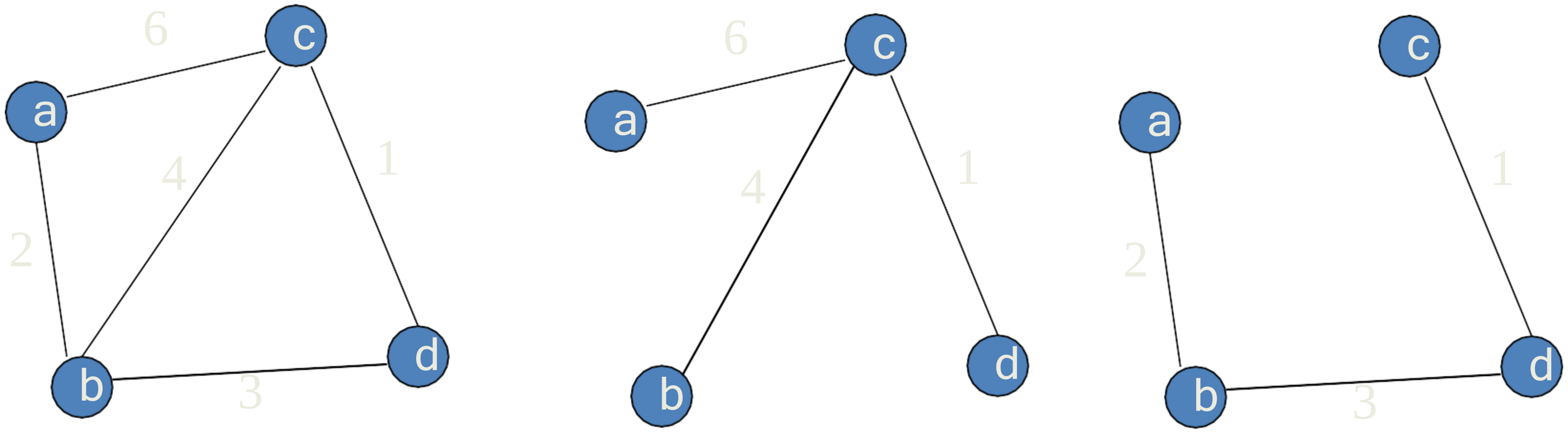
M=40 , n=3

w1,w2,w3 = 20,25,10

p1,p2,p2 = 30,40,35

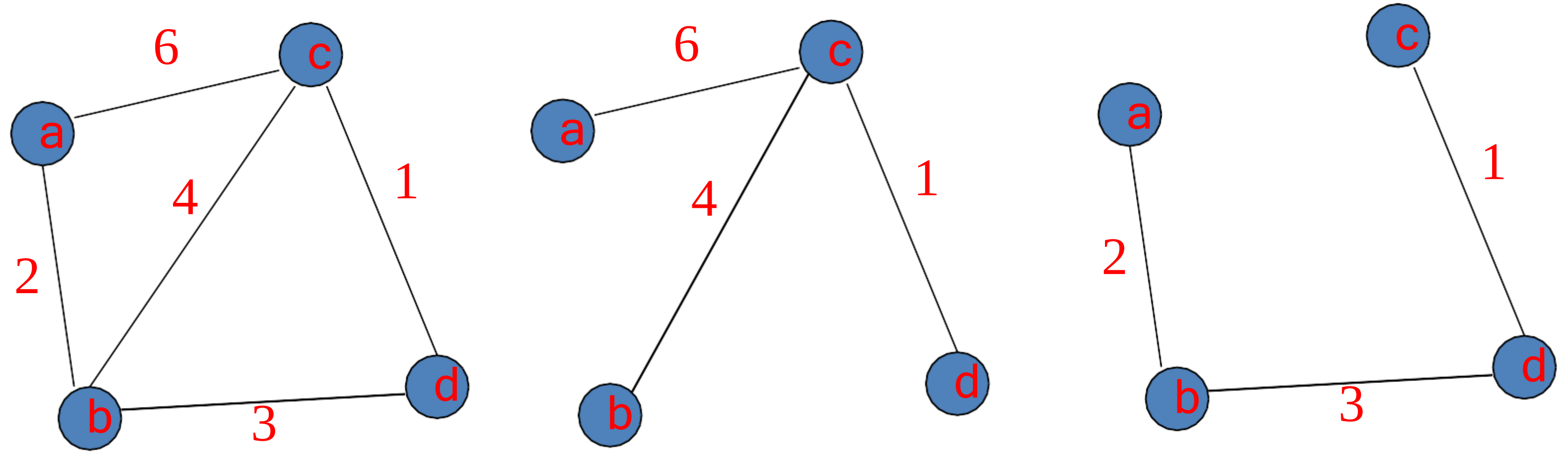
Minimum Spanning Tree (MST)

<u>Spanning tree</u> of a connected graph *G*: a connected acyclic subgraph of *G* that includes all of *G*'s vertices



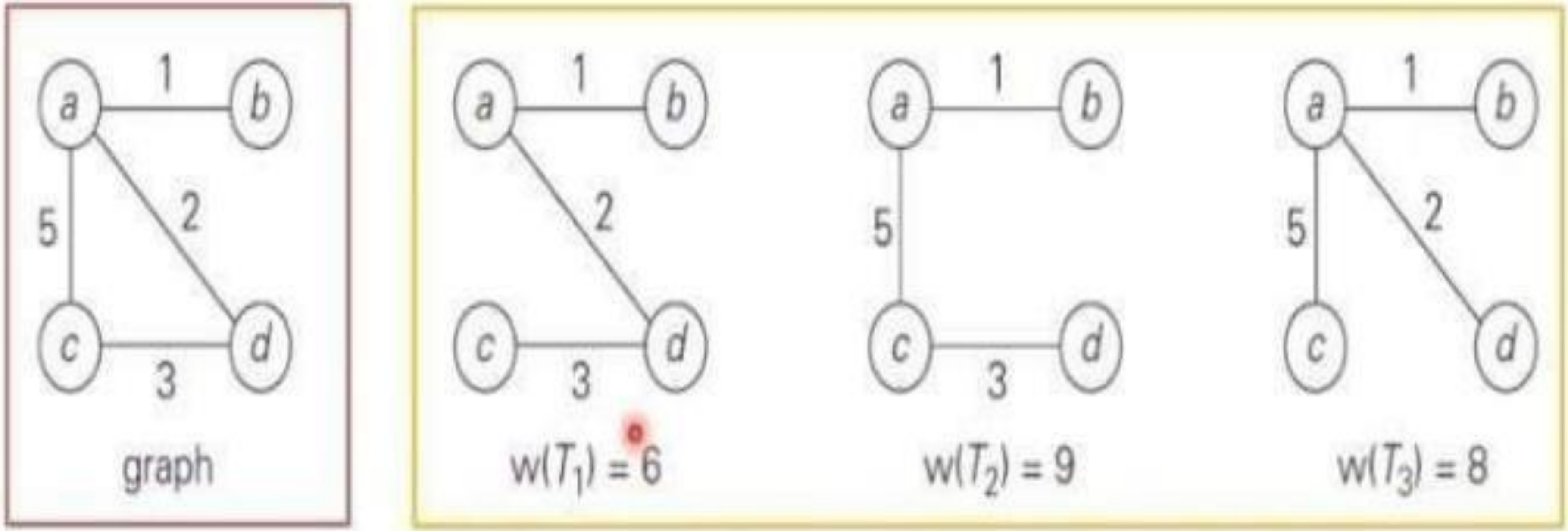
Minimum spanning tree of a weighted, connected graph G: a spanning tree of G of the minimum total weight

Example:



COST=11

COST=6

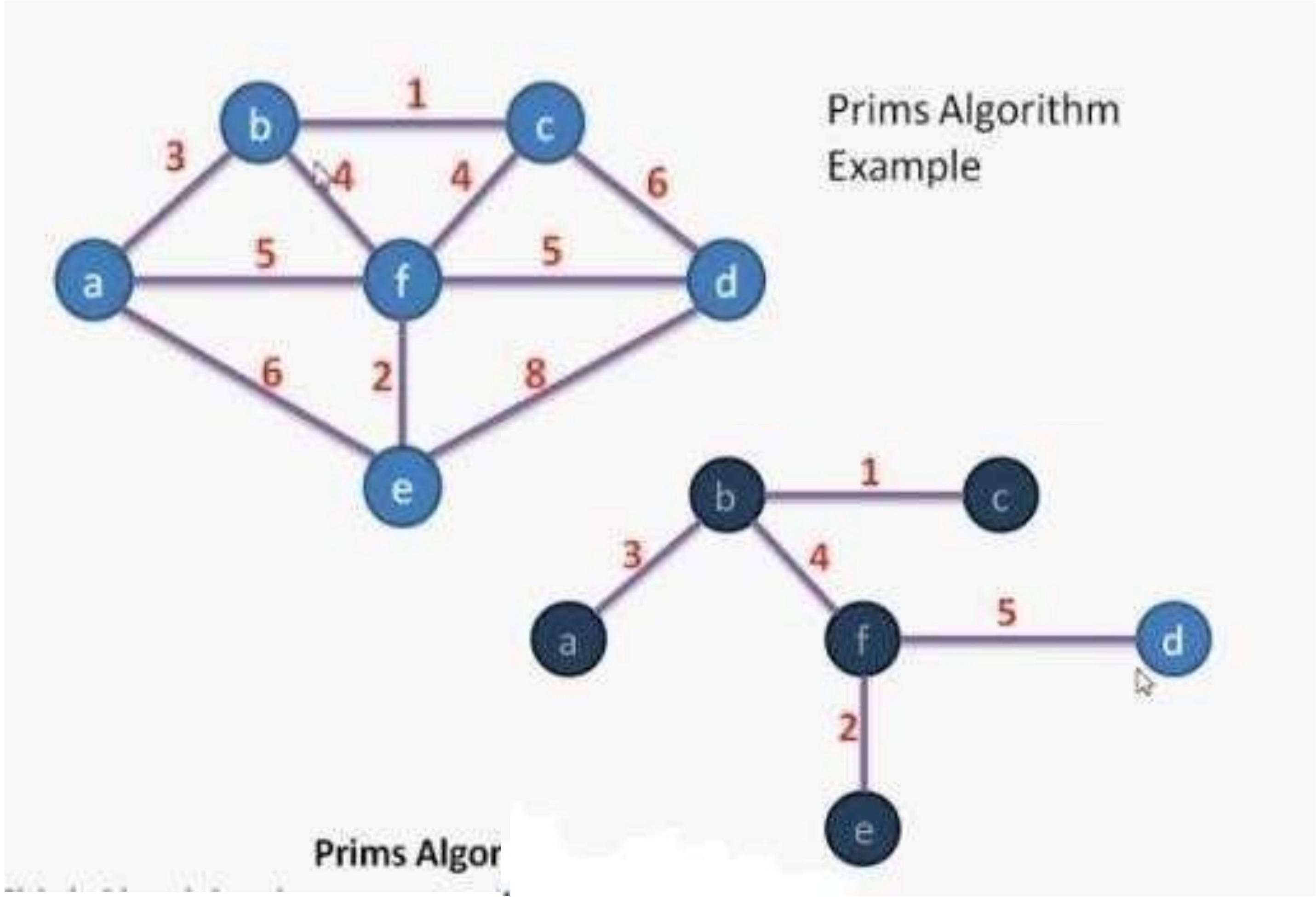


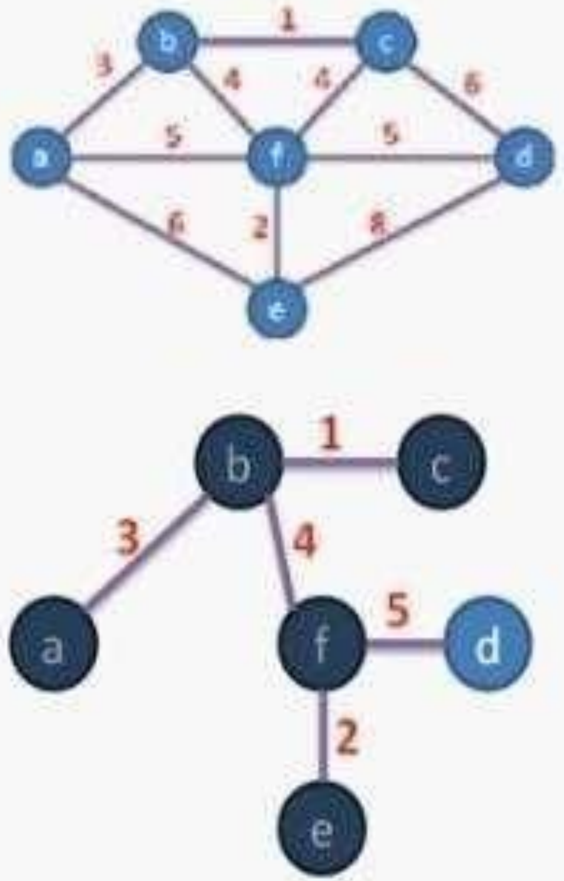
Graph and its spanning trees, with T_1 being the minimum spanning tree.

Note: MST of graph with **n** vertices will have exactly **n-1** edges

- Prim’s algorithm
- Kruskal’s algorithm

MST – Prim's algorithm





Algorithm Prim(G)

Vt ← {v0} //Set of visited vertices

Et ← ϕ

for i ← 1 to |V| -1 do

find minimum edge e between vertices v and u such that v is in Vt and u is in V - Vt

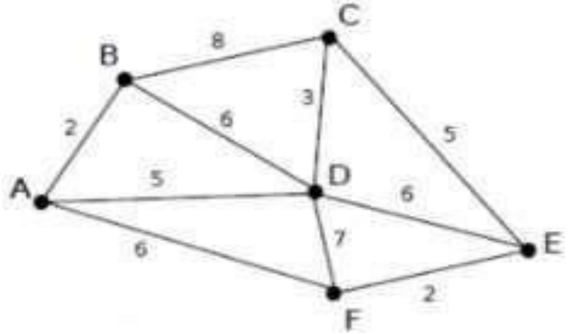
//Add u to Vt

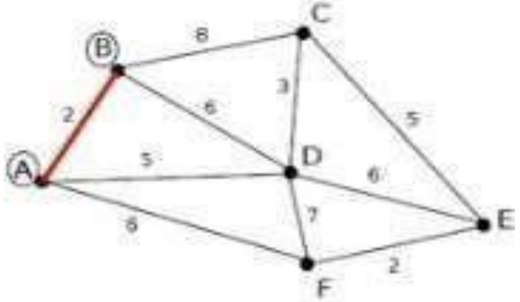
Vt ← Vt U {u}

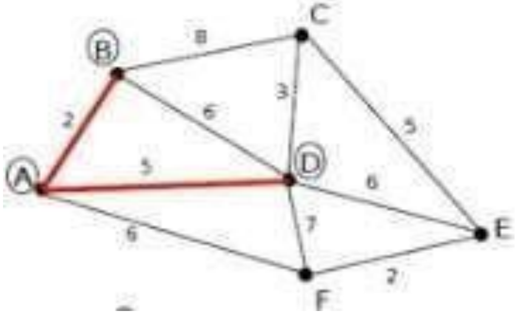
//Add the edge to the spanning tree

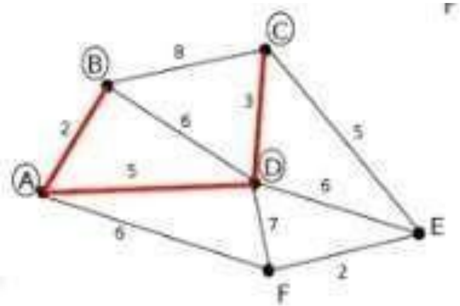
Et ← Et U {e}

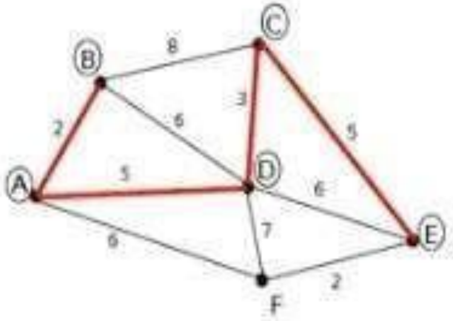
Prims Algorithm

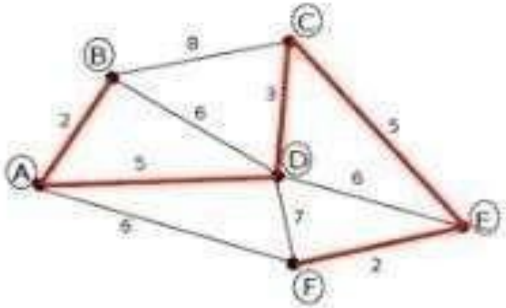

A
B
C
D
E
F
2
8
6
3
5
5
6
6
7
2

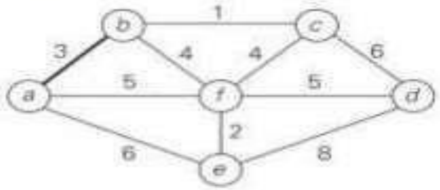
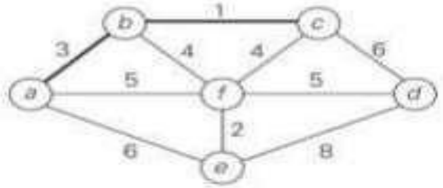
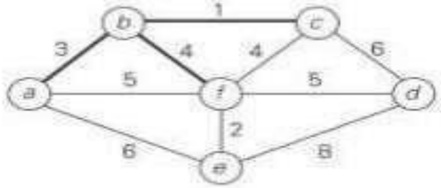
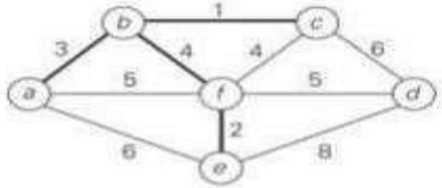
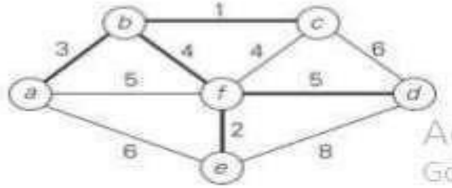

A
B
C
D
E
F
2
8
6
3
5
5
6
6
7
2


A
B
C
D
E
F
2
8
6
3
5
5
6
6
7
2


A
B
C
D
E
F
2
8
6
3
5
5
6
6
7
2


A
B
C
D
E
F
2
8
6
3
5
5
6
6
7
2


A
B
C
D
E
F
2
8
6
3
5
5
6
6
7
2

Tree vertices	*Remaining vertices*	*Illustration*
a(−, −)	**b(a, 3)** c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c(b, 1)** d(−, ∞) e(a, 6) f(b, 4)	
c(b, 1)	d(c, 6) e(a, 6) **f(b, 4)**	
f(b, 4)	d(f, 5) **e(f, 2)**	
e(f, 2)	**d(f, 5)**	
d(f, 5)		

Efficiency

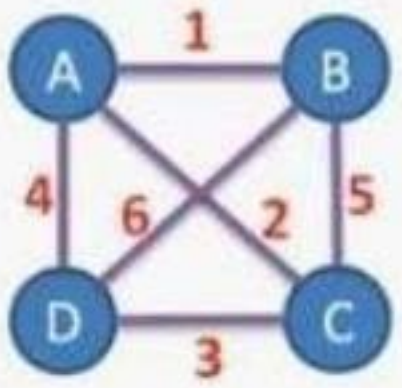
- The time efficiency of depends on the data structures used for implementing the priority queue and for representing the input graph.
- Since we have implemented using weighted matrix and unordered array, the efficiency is $O(|V^2|)$.
- If we implement using adjacency list and the priority queue for min-heap, the efficiency is $O(|E|\log|V|)$.

Kruskal's algorithm

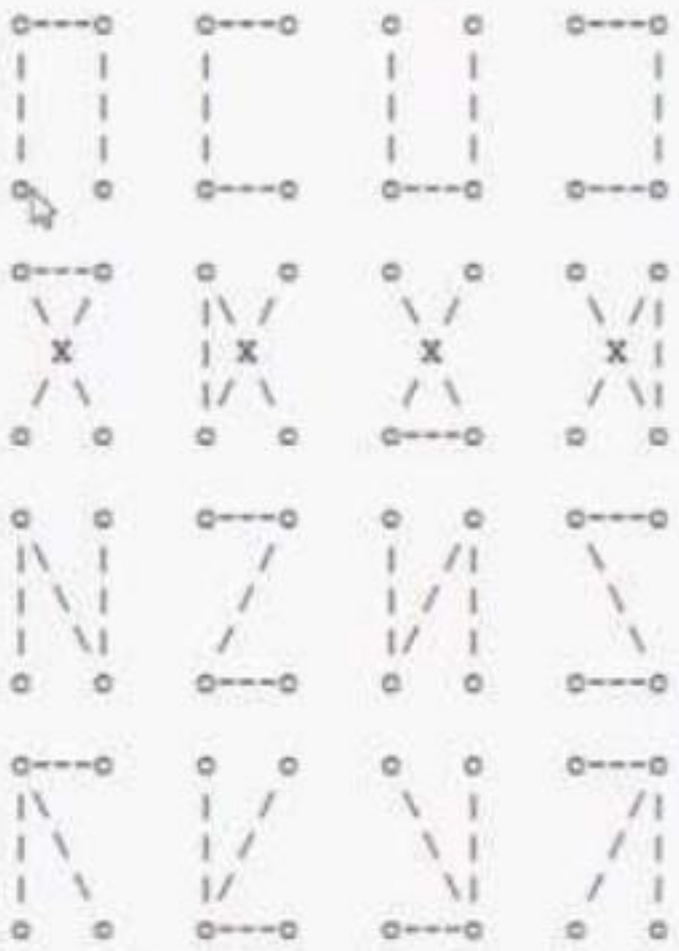
Kruskal's algorithm finds MST of a weighted connected graph G=<V,E> as an acyclic subgraph with |V| - 1 edges. Sum of all the edges weight should be minimum.

The algorithm begins by sorting the graph's edges in increasing order of their weights.

Then it scans this sorted list starting with the empty sub graph and it adds the next edge on the list to the current sub graph, if such an inclusion doesn't create a cycle and simply skipping the edges.



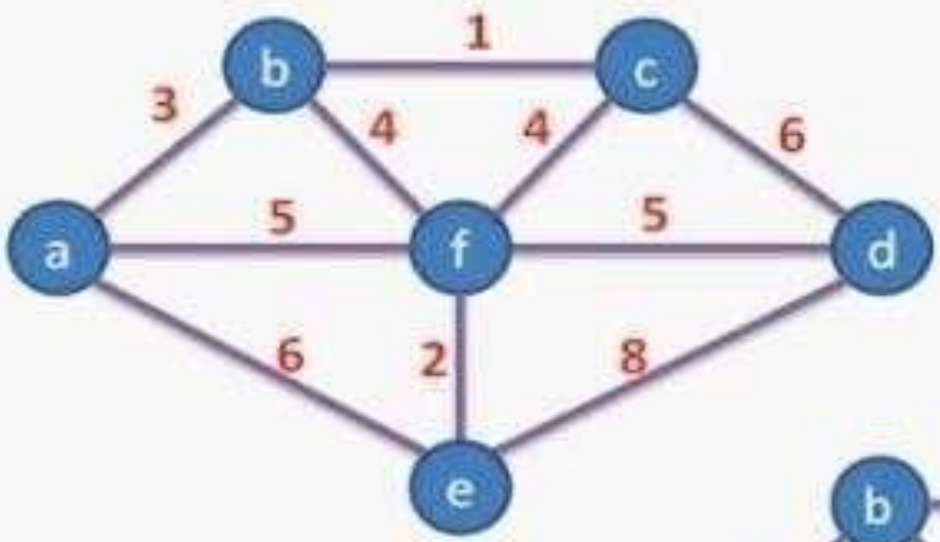
Brute Fore way:
There are 16 possibilites.
List out all possibilites
and choose the smallest

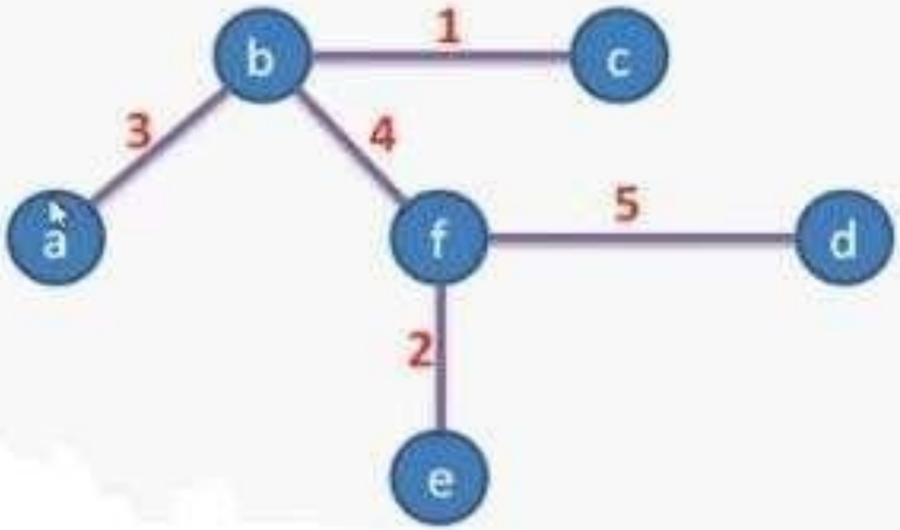


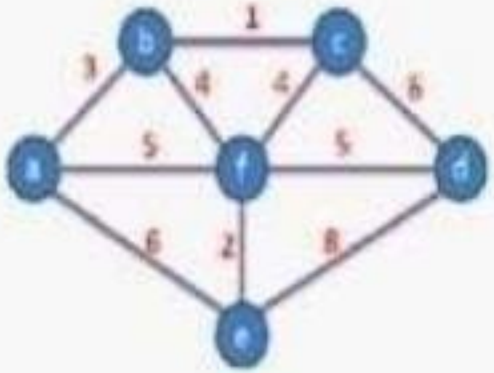
Kruskal's Algorithm

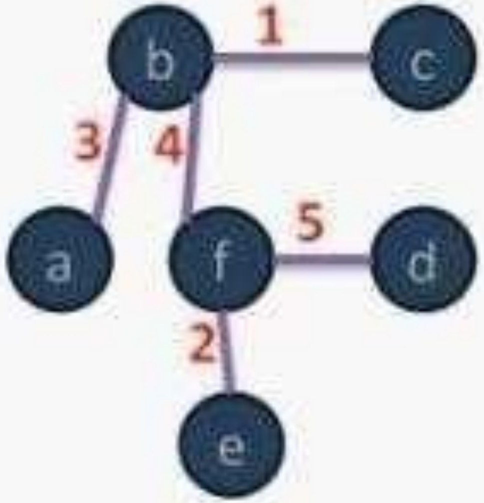
Kruskal's Algorithm Example

bc	ef	ab	bf	cf	af	df	ae	cd	de
1	2	3	4	4	5	5	6	6	8









Edge	Lengths
b→c	1
f→e	2
a→b	3
b→f	4
c→f	4
a→f	5
c→f	5
a→e	6
c→d	6
e→d	8

Algorithm Kruskal (G)

```
Sort E in ascending order of weights
Et ← 0 //no edges selected
encounter ← 0 //no of edges selected
k ← 0
while encounter < |V| - 1
    k ← k + 1
    if Et U {eik} is acyclic
        Et ← Et U {eik}
        encounter += 1
return Et
```

Kruskal's Algorithm

Time Complexity

The crucial check whether two vertices belong to the same tree can be found out using union -find algorithms.

- If the graph is represented as an adjacency matrix then the complexity of kruskal algorithm is V^2
- If you use binary heap and adjacency list the complexity can be of the order of ElogV.

Shortest paths – Dijkstra's algorithm

The Dijkstra's algorithm finds the shortest path from a given vertex to all the remaining vertices in a diagraph.

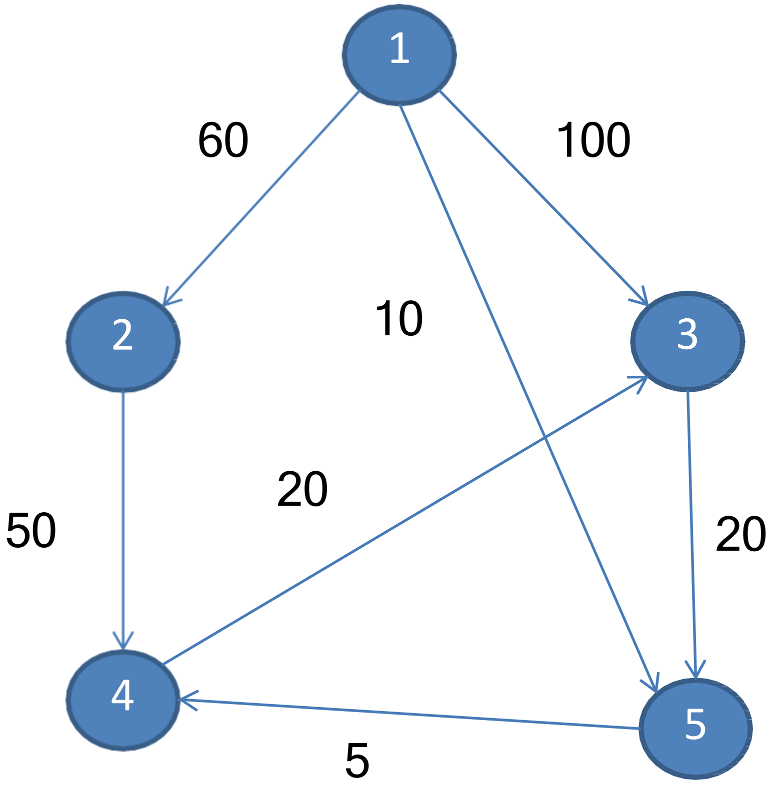
The constraint is that each edge has non-negative cost. The length of the path is the sum of the costs of the edges on the path.

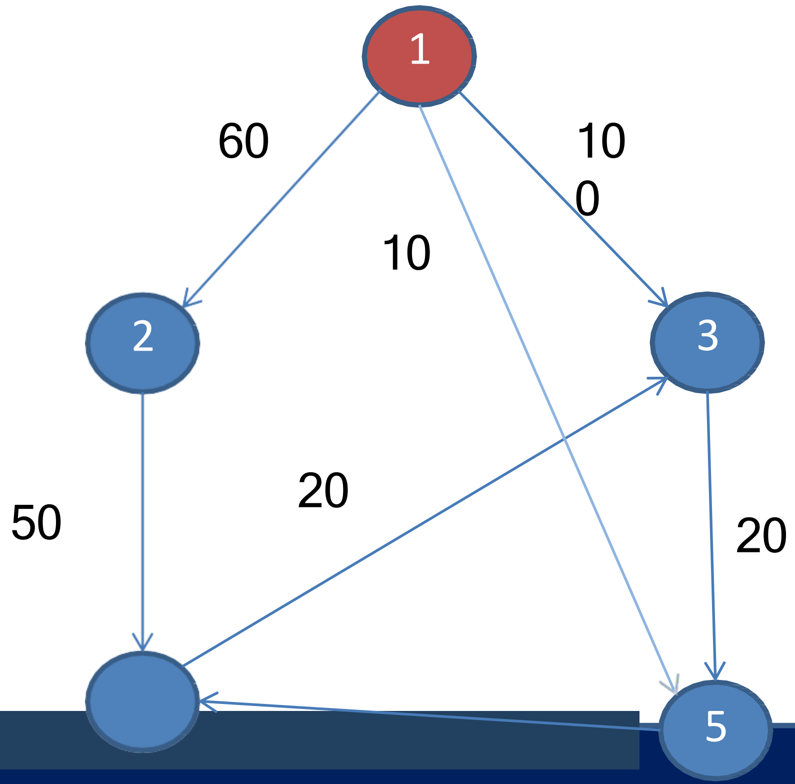
We have to find out the shortest path from a given source vertex 'S' to each of the destinations (other vertices) in the graph.

Example

Initially

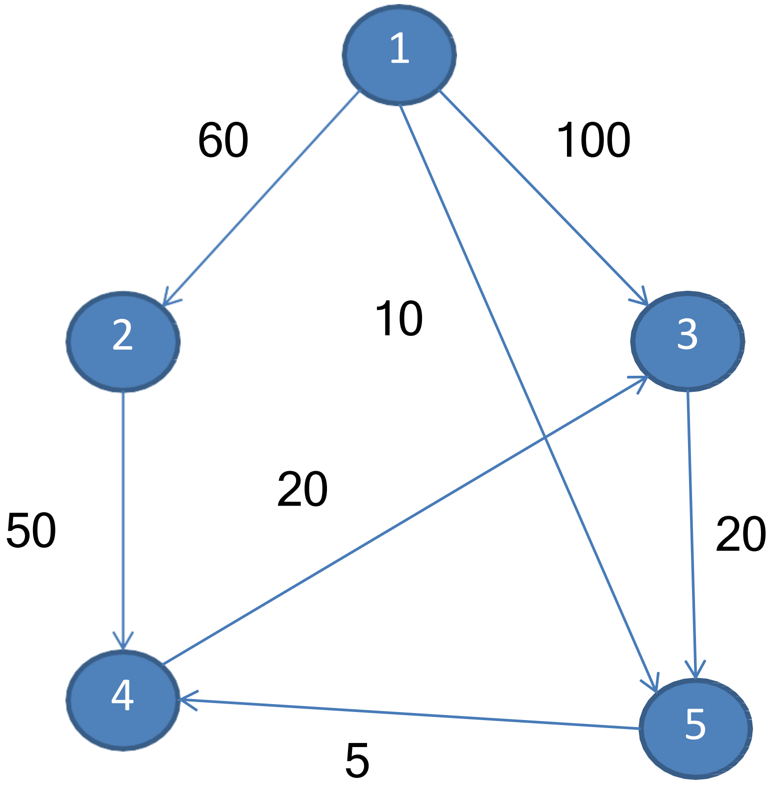
	1	2	3	4	5
S	~~0~~ 1	0	0	0	0
d	~~∞~~ 1	60	100	∞	10

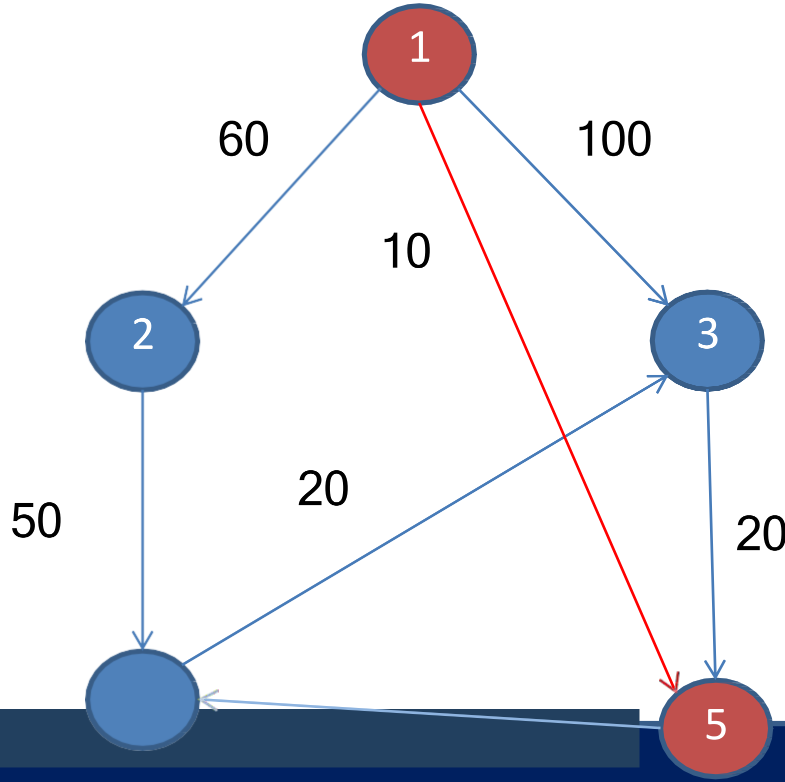




Example

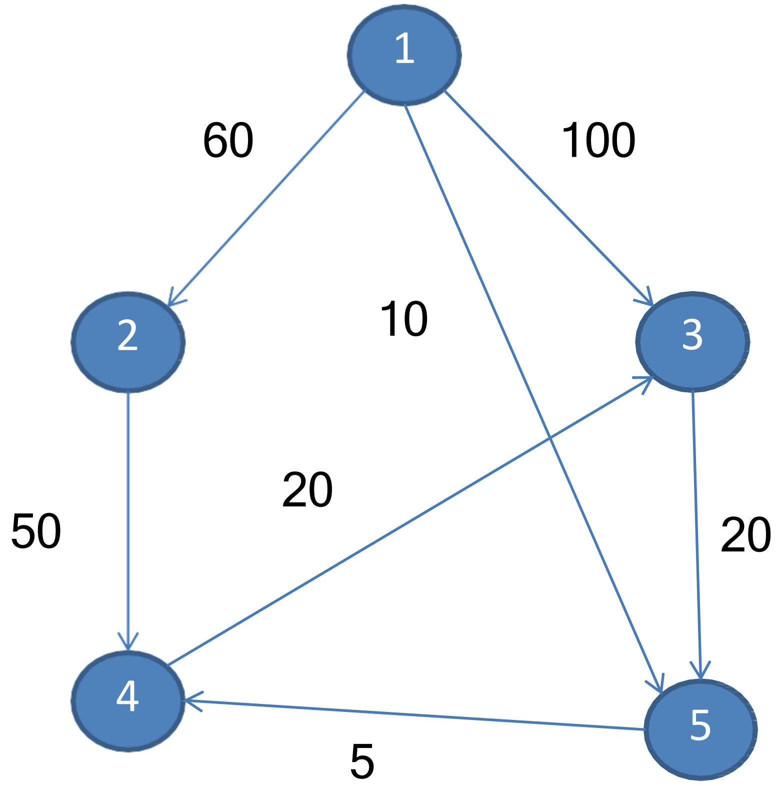
	1	2	3	4	5
S	1	0	0	0	1
d	1	60	100	∞	10

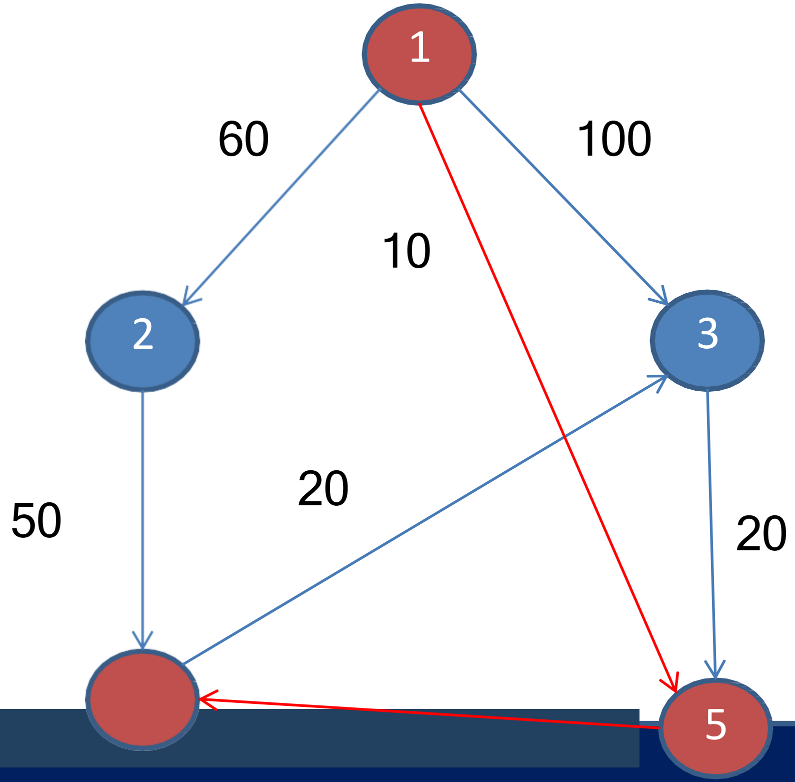




Example

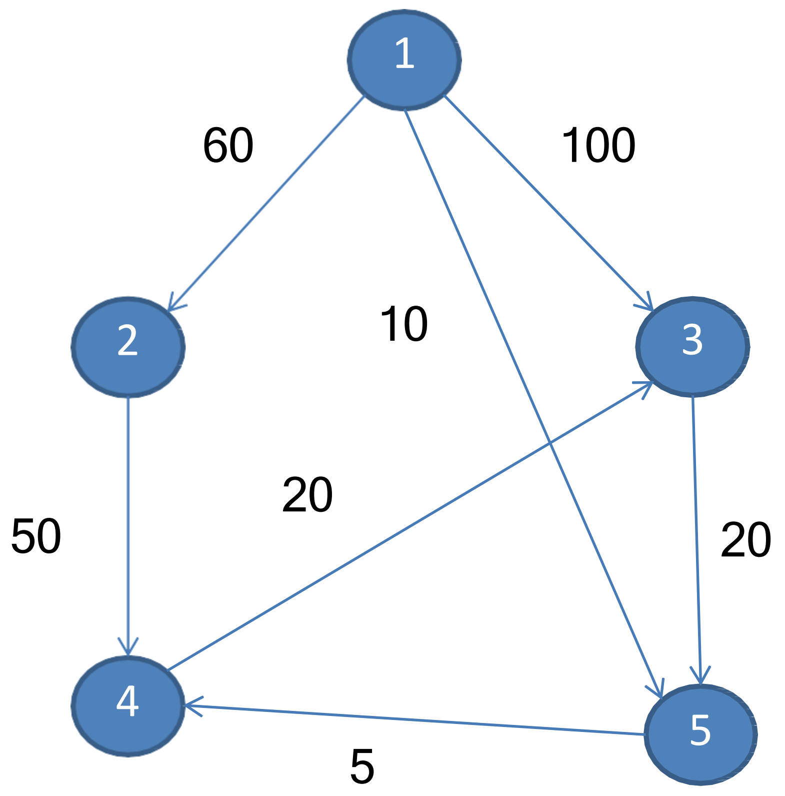
	1	2	3	4	5
S	1	0	0	1	1
d	1	60	100	15	10

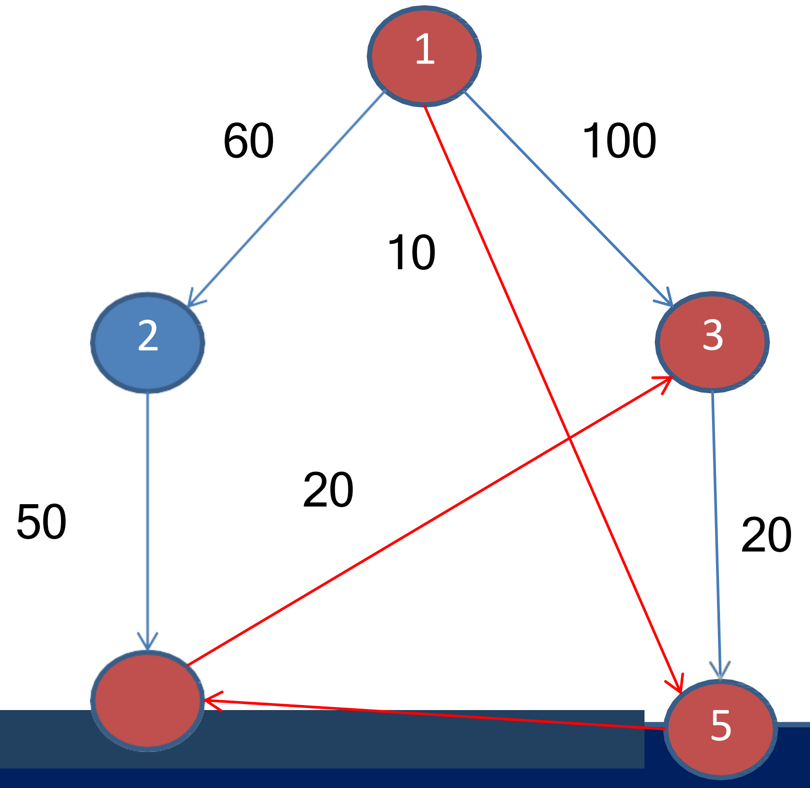




Example

	1	2	3	4	5
S	1	0	1	1	1
d	1	60	35	15	10

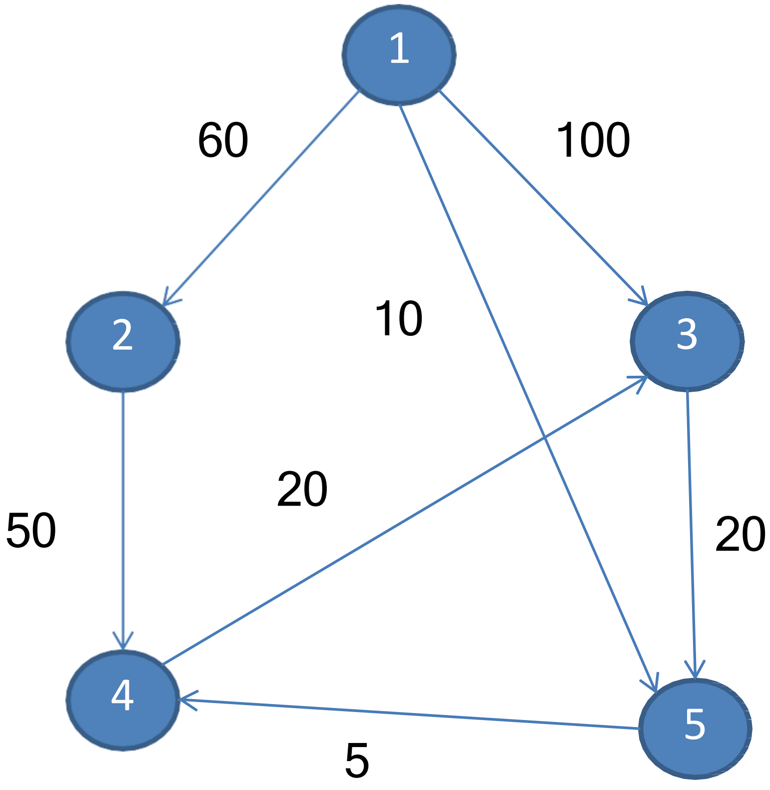


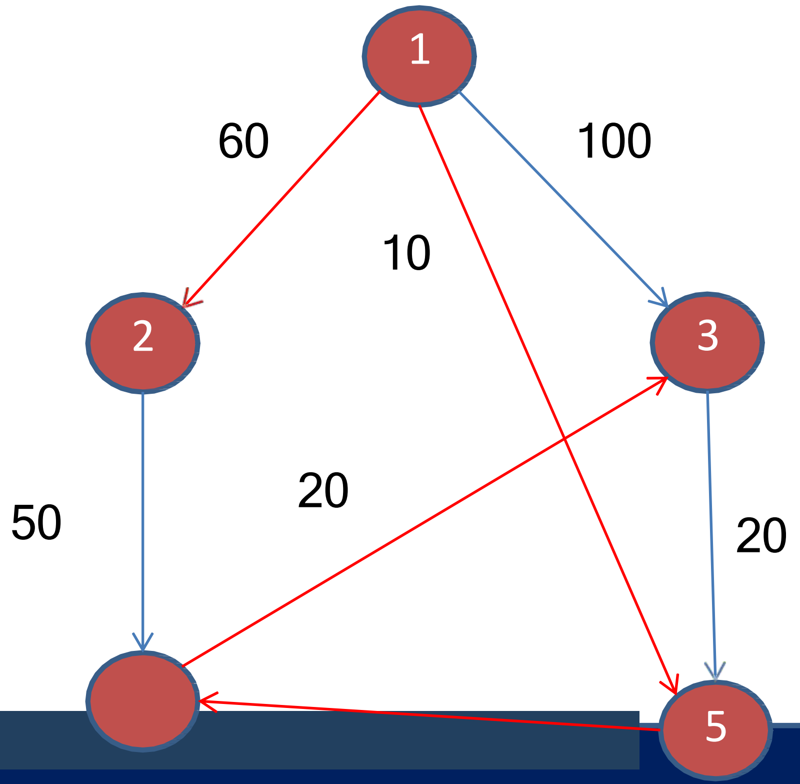


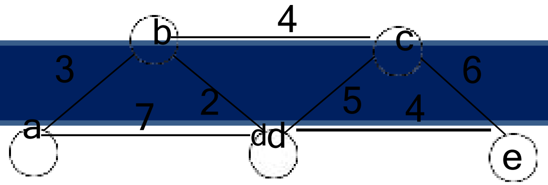
Example

Final distance form note 1 to all other nodes

	1	2	3	4	5
S	1	1	1	1	1
d	1	60	35	15	10

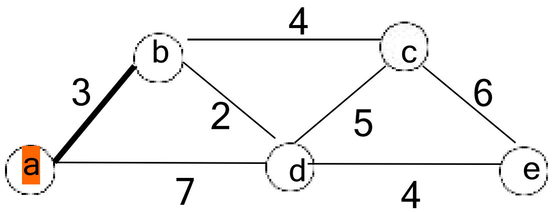


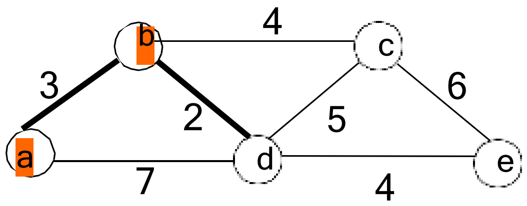


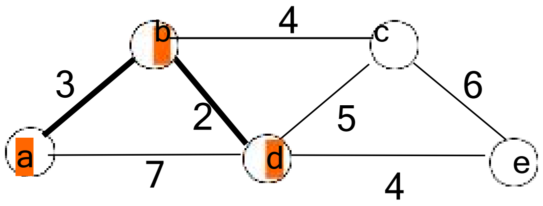


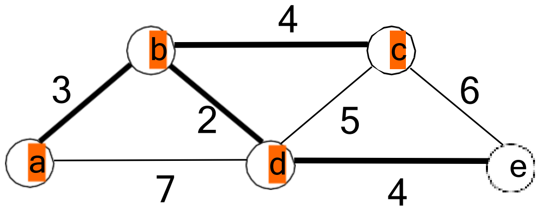
Example2

Tree vertices	Remaining vertices
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)	









Dijkstra's(s)

// Finds shortest path from source vertex to all other vertices

//Input: Weighted connected graph G=<V,E> with nonnegative weights and its vertices s

//Output: The length of distance   of a shortest path from s to v

{

1. for i = 1 to n do // Intialize

S[i] = 0;

d[i] = *a[s][i];*

2. S[s] = 1; //Assume 1 as the source vertex

d[s] = 1;

Dijkstra's algorithm

```
3. for i = 1 to n do
   {
   Choose a vertex u in v-s such that d[u] is
   minimum
   S = s ∪ u
        for each vertex v in v-s do
             d[v] = min{ d[u], d[u]+c[u,v]}
   }
}
```

Key points on Dijkstra's algorithm

Doesn't work for graphs with negative weights (whereas Floyd's algorithm does, as long as there is no negative cycle).

Applicable to both undirected and directed graphs.

Efficiency **O(|V²|)** for graphs represented by weight matrix and array implementation of priority queue

O(|E|log|V|) for graphs represented by adj. lists and min-heap implementation of priority queue

DYNAMIC PROGRAMMING

Dynamic Programming

Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems

Optimization Problem : Problems which demands minimum/maximum results

Dynamic " means "changing"
Programming" means "planning"

Dynamic Programming is a general algorithm design technique for solving problems with overlapping sub-problems.

Main idea:

- -Solve smaller instances once.
- -Record solutions in a table.
- -Get solution to a larger instance from some smaller instances.
- -Optimal solution for the initial instance is obtained from that table.

Principle of Optimality

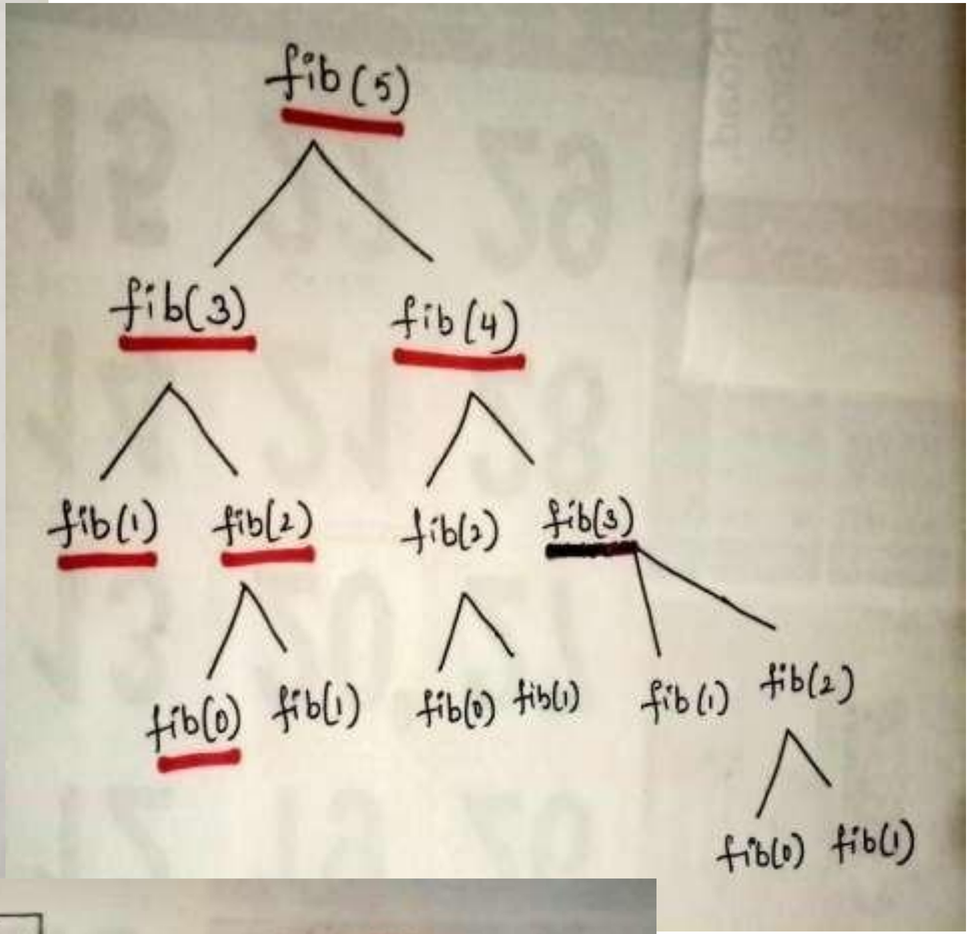
Definition [Principle of optimality] The principle of optimality states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.

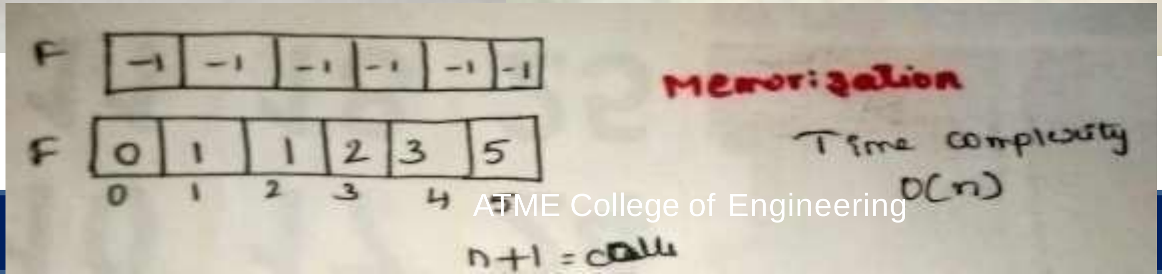
Problems can be solved by taking sequence of decisions to get optimal solutions

Example: Fibonacci numbers

$$fib(n) = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ fib(n-2)+fib(n-1) & \text{if } n>1 \end{cases}$$

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





Example: Fibonacci numbers

Tabulation Method

```
int fib(int n)
{
    if (n<=1)
        return n;
    F[0]=0; F[1]=1;
    for (int i=2; i<=n; i++)
    {  F[i]=F[i-2]+F[i-1];
    }
    return F[n];
}
```

F	0	1	1	2	3	5
	0	1	2	3	4	5
			i	i	i	i

fib(5)

return f(5) = 5

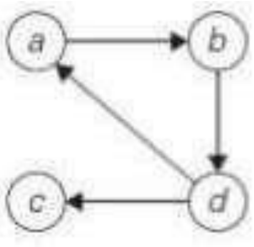
Bottom up approa

Examples of DP algorithms

- Computing a binomial coefficient
- Longest common subsequence
- Warshall's algorithm for transitive closure
- Floyd's algorithm for all-pairs shortest paths
- Constructing an optimal binary search tree
- Some instances of difficult discrete optimization problems:
  - traveling salesman
  - knapsack

Warshall’s Algorithm: Transitive Closure

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



(a) Digraph.

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

(b) Its adjacency matrix.

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

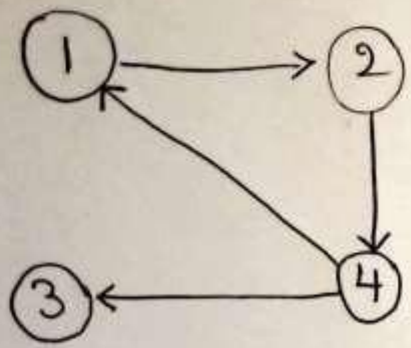
(c) Its transitive closure.

Warshall's Algorithm: Transitive Closure

Constructs transitive closure T as the last matrix in the sequence of n-by-n matrices $R^{(0)}, \dots, R^{(k)}, \dots, R^{(n)}$ where $R^{(k)}[i,j] = 1$ iff there is nontrivial path from i to j with only the first k vertices allowed as intermediate

Note: that $R^{(0)} = A$ (adjacency matrix), $R^{(n)} = T$ (transitive closure)

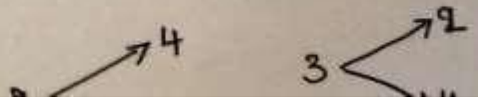
Warshall's Algorithm



$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 1 & 0 & 1 & 0 \end{array}$$

A

$$R^{(0)} = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 1 & 0 & 1 & 0 \end{array}$$



$$R^{(1)} = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 1 & 1 & 1 & 0 \end{array}$$

R[4,2] = 0 OR(R[4,1] and R[1,2])
= 0 OR (1 and 1) = 1

$$R^{(2)} = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 1 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 1 & 1 & 1 & 1 \end{array}$$

R[2,4] = 0 OR(R[1,2] AND R[2,4])
= 0 OR (1 and 1)
= 1

$$R^{(3)} = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 1 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 1 & 1 & 1 & 1 \end{array}$$

$$R^{(4)} = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 1 & 1 & 1 & 1 \\ 2 & 1 & 1 & 1 & 1 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 1 & 1 & 1 & 1 \end{array}$$

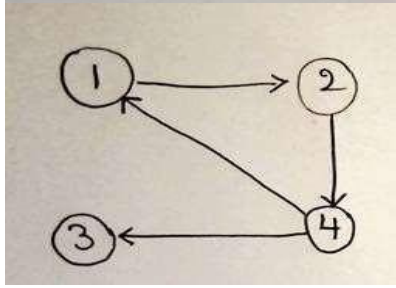
R [1,1] = 0 OR(R(1,4) and R(4,1))
= 0 OR (1 and 1) = 1

R [1,3] = 0 OR [R(1,4) and R[4,3)
= 0 OR [1 and 1] = 1

R[2,1] = 0 OR [R(2,4) and R(4,1)
= 0 OR (1 and 1) = 1

R[2,2] = 0 OR [R (2,4) and R(4,2)
= 0 OR [1 and 1] = 1

R[2,3] = 0 OR [R(2,4) and R(4,3)
= 0 OR [1 and 1] = 1



$$R^{(0)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

$$R^{(1)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

R[2,2] = 0 OR [R(2,1) AND R(1,2)]
= 0 OR [0 AND 1] = 0

R[2,3] = 0 OR [R[2,1] AND R(1,3)]
= 0 OR [0 AND 0] = 0

R[2,4] = 1 OR [R(2,1) AND R(1,4)]
= 1 OR [0 AND 0] = 1

R[4,2] = 0 OR [R(4,1) AND R(1,2)]
= 0 OR (1 AND 1) = 1

$$R^{(2)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

R[2.4] : 0 OR (R[1,2] AND R[2,4])
: 0 OR (1 and 1)
: 1

$$R^{(3)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

$$R^{(4)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

R[1,1] = 0 OR (R(1,4) and R(4,1))
= 0 OR (1 and 1) = 1

R[1,3] = 0 OR [R(1,4) and R[4,3]]
= 0 OR [1 and 1] = 1

R[2,1] = 0 OR [R(2,4) and R(4,1)]
= 0 OR (1 and 1) = 1

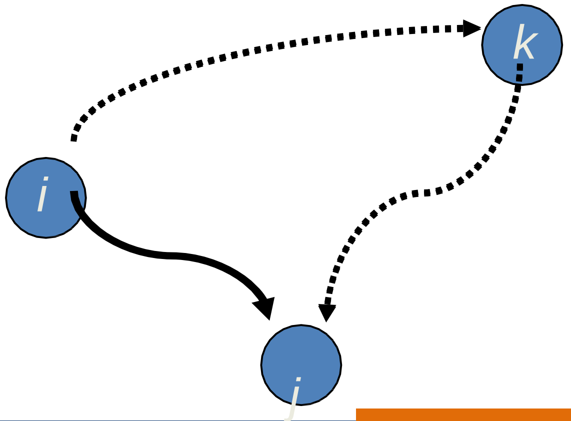
R[2,2] = 0 OR [R(2,4) and R(4,2)]
= 0 OR [1 and 1] = 1

R[2,3] = 0 OR [R(2,4) and R(4,3)]
= 0 OR [1 and 1] = 1

Warshall's Algorithm (recurrence)

On the *k*-th iteration, the algorithm determines for every pair of vertices *i, j* if a path exists from *i* and *j* with just vertices 1,…,*k* allowed as intermediate

$R^{(k)}[i,j] = \begin{cases} R^{(k-1)}[i,j] & \text{(path using just } 1,\ldots,k\text{-}1) \\ R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j] & \text{(path from } i \text{ to } k \text{ and from } k \text{ to } j \text{ using just } 1,\ldots,k\text{-}1) \end{cases}$



Initial condition?

Warshall's Algorithm (matrix generation)

Recurrence relating elements $R^{(k)}$ to elements of $R^{(k-1)}$ is:

$$R^{(k)}[i,j] = R^{(k-1)}[i,j] \text{ or } (R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j])$$

It implies the following rules for generating $R^{(k)}$ from $R^{(k-1)}$:

Rule 1 If an element in row i and column j is 1 in $R^{(k-1)}$,
it remains 1 in $R^{(k)}$

Rule 2 If an element in row i and column j is 0 in $R^{(k-1)}$,
it has to be changed to 1 in $R^{(k)}$ if and only if
the element in its row i and column k and the element
in its column j and row k are both 1's in $R^{(k-1)}$

Warshall's Algorithm (pseudocode and analysis)

ALGORITHM *Warshall*$(A[1..n, 1..n])$

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

$R^{(0)} \leftarrow A$
for $k \leftarrow 1$ **to** n **do**
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **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 $R^{(n)}$

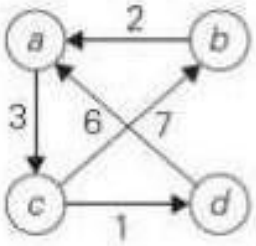
Time efficiency: $\Theta(n^3)$

Floyd's Algorithm: All pairs shortest paths

Problem: In a weighted (di)graph, find shortest paths between every pair of vertices

Same idea: construct solution through series of matrices $D^{(0)}, \ldots, D^{(n)}$ using increasing subsets of the vertices allowed as intermediate

Example:



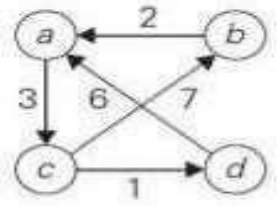
(a) Digraph.

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

(b) Its weight matrix.

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

(c) Its distance matrix



$$D^{(0)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \left[\begin{array}{cccc} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{array}\right] \end{array}$$

$$D^{(1)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \left[\begin{array}{cccc} 0 & \infty & 3 & \infty \\ 2 & 0 & \mathbf{5} & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \mathbf{9} & 0 \end{array}\right] \end{array}$$

$$D^{(2)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \left[\begin{array}{cccc} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \mathbf{9} & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{array}\right] \end{array}$$

$$D^{(3)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \left[\begin{array}{cccc} 0 & \mathbf{10} & 3 & \mathbf{4} \\ 2 & 0 & 5 & \mathbf{6} \\ 9 & 7 & 0 & 1 \\ 6 & \mathbf{16} & 9 & 0 \end{array}\right] \end{array}$$

$$D^{(4)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \left[\begin{array}{cccc} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ \mathbf{7} & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{array}\right] \end{array}$$

$$D[b,c] = \min \begin{cases} \infty \\ D(b,a) + D(a,c) = 2+3 = 5 \end{cases} = 5$$

$$D[d,c] = \min \begin{cases} \infty \\ D(d,a) + D(a,c) = 6+3 = 9 \end{cases}$$

$$D[c,a] = \min \begin{cases} \infty \\ D(c,b) + D[b,a] = 7+2 = 9 \end{cases}$$

$$D[a,b] = \min \begin{cases} \infty \\ D(a,c) + D(c,b) = 3+7 = 10 \end{cases}$$

$$D[a,d] = \min \begin{cases} \infty \\ D[a,c] + D(c,d) = 3+1 = 4 \end{cases}$$

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

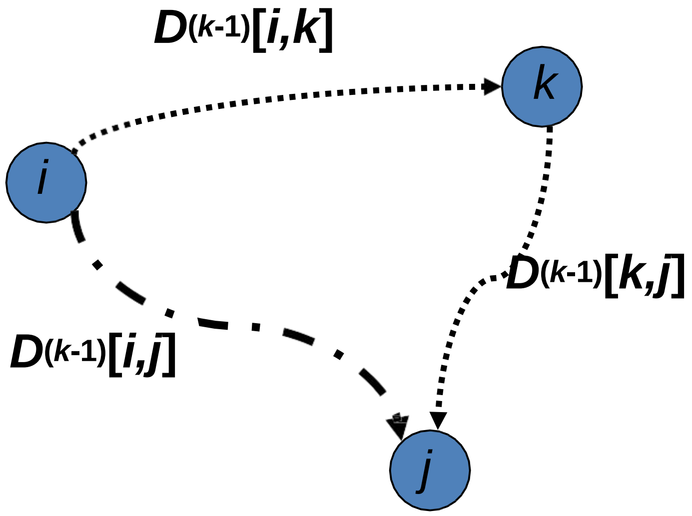
$$D[d,b] = \min \begin{cases} \infty \\ D[d,c] + D[c,b] = 9+7 = 16 \end{cases}$$

$$D[c,a] = \min \begin{cases} 9 \\ D[c,d] + D[d,a] = 1+6 = 7 \end{cases}$$

Floyd's Algorithm (matrix generation)

On the k-th iteration, the algorithm determines shortest paths between every pair of vertices i, j that use only vertices among 1,…,k as intermediate

$$D^{(k)}[i,j] = \min \{D^{(k-1)}[i,j],\ D^{(k-1)}[i,k] + D^{(k-1)}[k,j]\}$$



Initial condition?

ALGORITHM *Floyd*($W[1..n, 1..n]$)

//Implements Floyd's algorithm for the all-pairs shortest-paths problem
//Input: The weight matrix W of a graph with no negative-length cycle
//Output: The distance matrix of the shortest paths' lengths

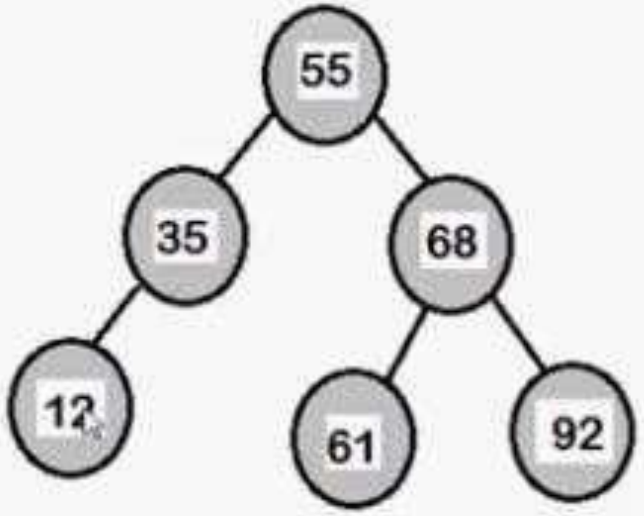
$D \leftarrow W$ //is not necessary if W can be overwritten
for $k \leftarrow 1$ **to** n **do**
    for $i \leftarrow 1$ **to** n **do**
        for $j \leftarrow 1$ **to** n **do**
            $D[i, j] \leftarrow \min\{D[i, j], D[i, k] + D[k, j]\}$
return D

Time efficiency: $\Theta(n^3)$

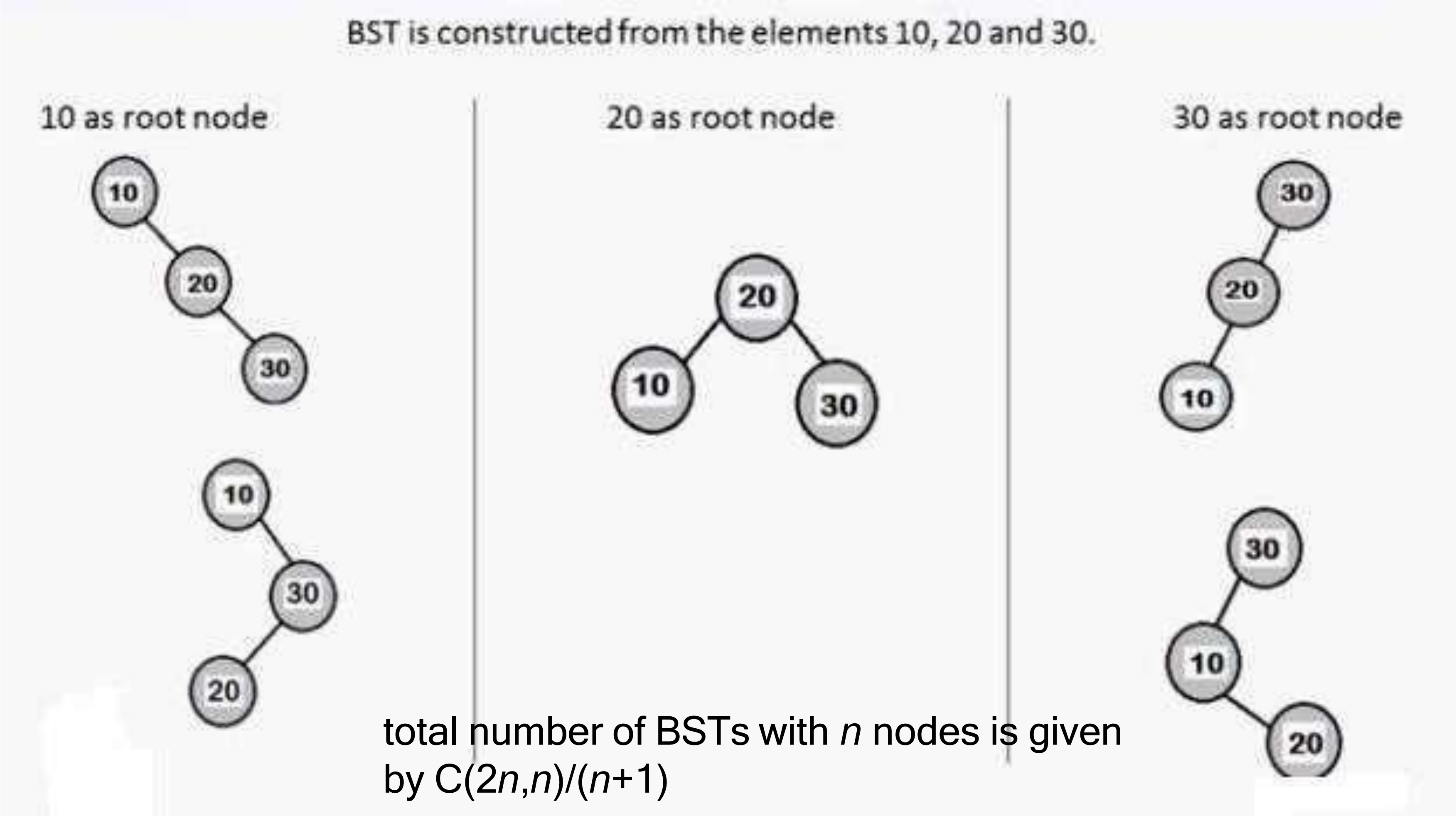
Note: Works on graphs with negative edges but without negative cycles.

Optimal Binary Search Tree

Binary Tree



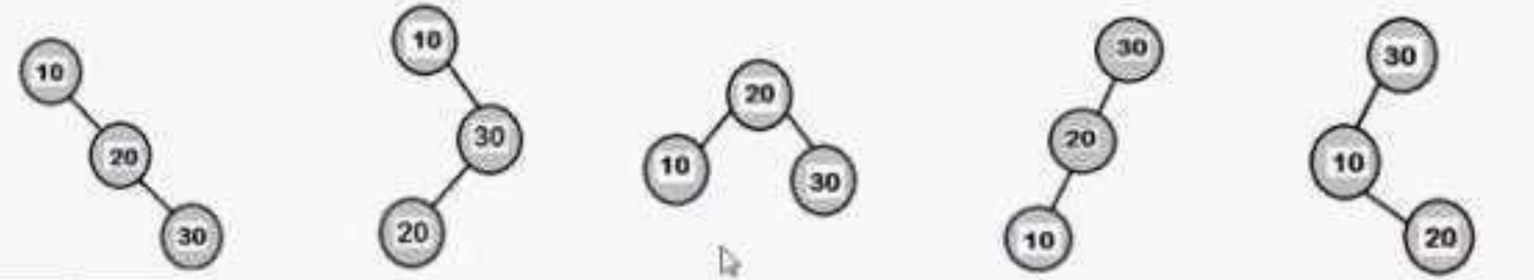
All elements in the left subtree of root are less than root and all elements in the right subtree of root are greater than root.



total number of BSTs with n nodes is given by $C(2n,n)/(n+1)$

Cost of searching any Key

- Cost of searching any key is dependent on comparisons required for searching any key element in the tree.



Key	C
10	1
20	2
30	3
Avg	6/3=2

Key	C
10	1
20	3
30	2
Avg	6/3=2

Key	C
10	2
20	1
30	2
Avg	5/3=1.66

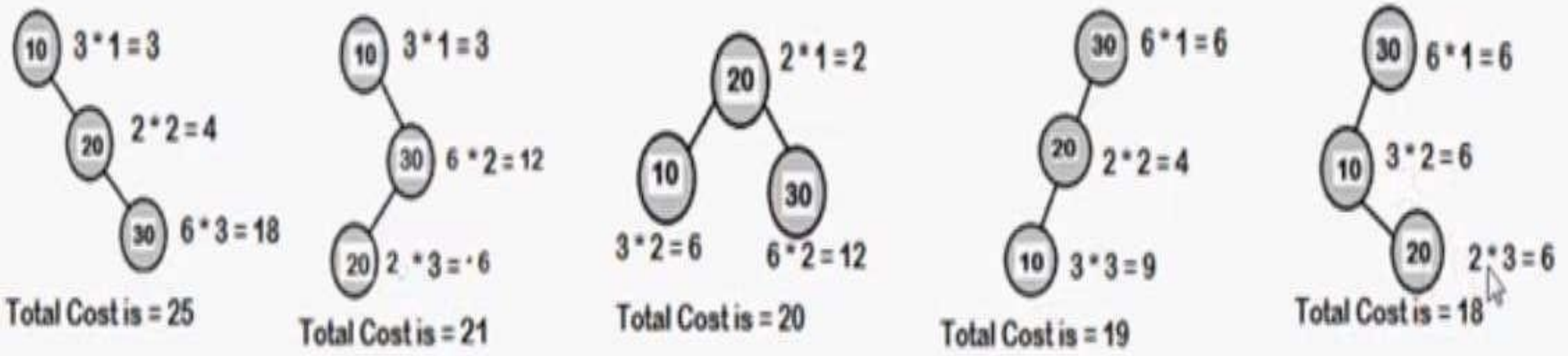
Key	C
10	3
20	2
30	1
Avg	6/3=2

Key	C
10	2
20	3
30	1
Avg	6/3=2

- Third tree is balanced tree because,
  - Average of comparison is less
  - Height is less

Cost of BST-Frequencies

Keys	10	20	30
Frequencies for Searching	3	2	6



- Minimum searching cost is low means it's a Optimal BST.
- Tree 5 is having minimum searching cost = 15.
- Tree 5 is OBST .
- Though it is not height balanced, tree 5 is OBST which is based on frequencies the cost of BST is minimum.
- Key Point to Remember : "Highest frequency key must be root node and lowest frequency key must be a child (Leaf) node."

18

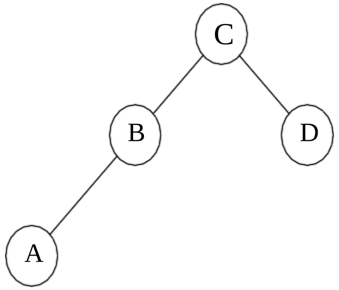
Optimal Binary Search Tree

Problem: Given n keys $a_1 < \ldots < a_n$ and probabilities $p_1, \ldots, p_n$ searching for them, find a BST with a minimum average number of comparisons in successful search.

Since total number of BSTs with n nodes is given by $C(2n,n)/(n+1)$, which grows exponentially, brute force is not recommended.

- BST is a tree which is mainly constructed for searching a key from it.
- For searching any key from a given BST, it should take optimal time.
- For this, we need to construct a BST in such a way that it should take optimal time to search any of the key from given BST.
- To construct OBST, frequency of searching of every key is required.

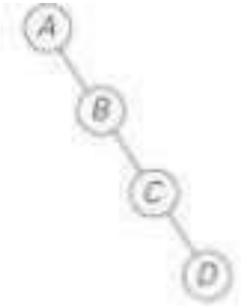
Example: What is an optimal BST for keys *A*, *B*, *C*, and *D* with search probabilities 0.1, 0.2, 0.4, and 0.3, respectively?



Average # of comparisons

= 1*0.4 + 2*(0.2+0.3) + 3*0.1

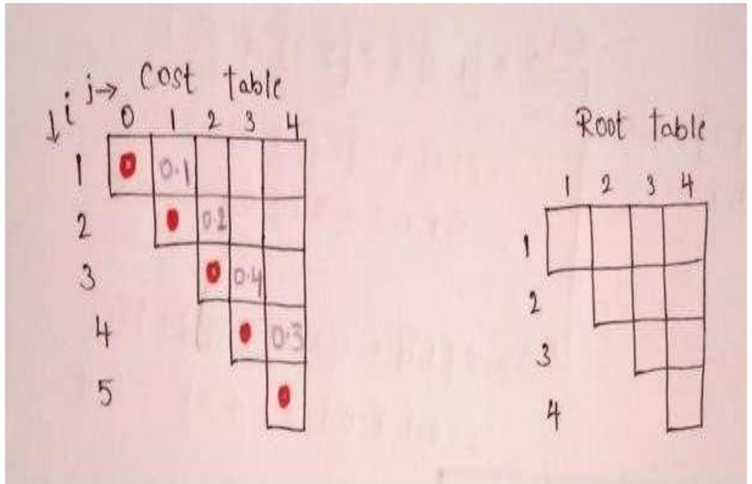
= 1.7



$0.1 * 1 + 0.2 * 2 + 0.4 * 3 + 0.3 * 4 = 2.9.$

ALGORITHM *OptimalBST(P[1..n])*

```
//Finds an optimal binary search tree by dynamic programming
//Input: An array P[1..n] of search probabilities for a sorted list of n keys
//Output: Average number of comparisons in successful searches in the
//        optimal BST and table R of subtrees' roots in the optimal BST
for i ← 1 to n do
    C[i, i − 1] ← 0
    C[i, i] ← P[i]
    R[i, i] ← i
C[n + 1, n] ← 0
for d ← 1 to n − 1 do    //diagonal count
    for i ← 1 to n − d do
        j ← i + d
        minval ← ∞
        for k ← i to j do
            if C[i, k − 1] + C[k + 1, j] < minval
                minval ← C[i, k − 1] + C[k + 1, j];  kmin ← k
        R[i, j] ← kmin
        sum ← P[i];  for s ← i + 1 to j do sum ← sum + P[s]
        C[i, j] ← minval + sum
return C[1, n], R
```



Knapsack Problem

Given n objects of known weights w1,w2…wn and profit p1, p2, … pn for those n objects and a knapsack of capacity M i.e is not exceeding the weight M. Let a variable xi be ‘0’ if we do not select the object ‘i’ or ‘1’ if we include the object ‘i’ into the knapsack.

The objective is to maximize the total profit earned. Since the knapsack capacity is M, we require the total weight of all chosen objects to be at most M.

Knapsack problem

Maximize $\sum_{1 \le i \le n} PiXi$

Subject to $\sum_{1 \le i \le n} wiXi \le$ m

The profits and weights are positive numbers.

Knapsack Problem by DP (pseudocode)

```
Algorithm DPKnapsack(int: w[1..n], int: p[1..n], M)
    int: V[0..n,0..M]
    for j := 0 to M do
                                        V[0,j] := 0
                                    for i := 0 to n do
        V[i,0] := 0
    for i := 1 to n do
        for j := 1 to M do
            if  w[i] ≤ j and p[i] + V[i-1,j-w[i]] > V[i-1,j] then
                V[i,j] := p[i] + V[i-1,j-w[i]];
            else
                V[i,j] := V[i-1,j];
    return V[n,M]
```

Running time and space: O(nW).

Analysis

- The classic dynamic programming approach, works bottom up: it fills a table with solutions to all smaller subproblems, each of them is solved only once.
- Drawback: Some unnecessary subproblems are also solved

- The time efficiency and space efficiency of this algorithm are both in **Θ(nW).**
- The time needed to find the composition of an optimal solution is in **O(n).**

table [4, 5] With $i = 4$, $j = 5$, $w_i = 5$ and $v_i = 6$

As $j \geq w_i$, we will obtain table [4, 5] as

$$\begin{aligned}\text{table}\ [4, 5] &= \text{maximum}\left\{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\right\} \\ &= \text{maximum}\ \{\text{table}\ [3, 5]\ ,\ 6 + \text{table}\ [3, 0]\} \\ &= \text{maximum}\ \{7, 6 + 0\}\end{aligned}$$

$\therefore$ $\boxed{\text{table}\ [4, 5] = 7}$

Thus the table can be finally as given below

	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

This is the total value of selected items

Discussion

- The direct top-down approach to finding a solution to such a recurrence leads to an algorithm that solves common subproblems more than once and hence is very inefficient.
- Since this drawback is not present in the top-down approach, it is natural to try to combine the strengths of the top-down and bottom-up approaches.
- The goal is to get a method that solves only subproblems that are necessary and does so only once. Such a method exists; it is based on using **memory functions**.

Algorithm MFKnapsack(i, j)

- Implements the memory function method for the knapsack problem
- **Input**: A nonnegative integer **i** indicating the number of the first items being considered and a nonnegative integer **j** indicating the knapsack capacity
- **Output:** The value of an optimal feasible subset of the first i items
- **Note:** Uses as global variables input arrays **Weights[1..n], V alues[1..n],** and **table F[0..n, 0..W]** whose entries are initialized with −1's except for row 0 and column 0 initialized with 0's

Memory Function Knapsack

Example: Knapsack of capacity $M = 5$

item	weight	value
1	2	\$8
2	1	\$6
3	3	\$16
4	2	\$11

capacity j

	0							
$w_1 = 2,\ p_1 = 8$	1	0	-1	-1	-1	-1	-1	
$w_2 = 1,\ p_2 = 6$	2	0	-1	-1	-1	-1	-1	
$w_3 = 3,\ p_3 = 16$	3	0	-1	-1	-1	-1	-1	
$w_4 = 2,\ p_4 = 11$	4	0	-1	-1	-1	-1	-1	?

Memory Function Knapsack

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

i=4, j=5, p[i]=11, wi =2

J-wi = 5-2 = 3 (able to fit into knapsack)

Find V[4,5] = max{ mfk[i-1,j], p[i] + mfk[i-1, j-wi]}

= max{ mkf(3,5), 11+mfk(3,3)}

= max{ ---------, 11+ -------)}

Find V[3,5] = max{ mkf(2,5), 16+mfk(2,2)}

= max{______, 16+_________}

Find V[3,3] = max{ mfk[i-1,j], p[i] + mfk[i-1, j-wi]}

= max{ mkf(2,3), 16+mfk(2,0)}

= max{ ----, 16+ -----)}

Find V[2,5] = max{ mkf(1,5), 6+mfk(1,4)}

= max{______, 6+________}

Find V[2,2] = max{ mkf(1,2), 6+mfk(1,1)}

= max{______, 6 +________}

Find V[2,3] = max{ mkf(1,3), 6+mfk(1,2)}

= max{______, 6 +________-}

Find V[1,5] = max{ mkf(0,5), 8+mfk(0,3)}

= max{ 0 , 8 + 0} = 8

Find V[1,4] = max{ mkf(0,4), 8+mfk(0,2)}

= max{ 0 , 8 + 0} = 8

Find V[1,2] = max{ mkf(0,2), 8+mfk(0,0)}

= max{ 0 , 8 + 0} = 8

Find V[1,3] = max{ mkf(0,3), 8+mfk(0,0)}

= max{ 0 , 8 + 0} = 8

Find V[1,1] = max{ mkf(0,1)} = 0

Back Substitute these values

$$\text{Find } V[2,3] = \max\{ mkf(1,3),\ 6+mfk(1,2)\}$$
$$= \max\{ 8,\ \ 6 + 8\} \ = 14$$

$$\text{Find } V[1,5] = \max\{ mkf(0,5),\ 8+mfk(0,3)\}$$
$$= \max\{ \ 0\ ,\ \ 8 + 0\} = 8$$
$$\text{Find } V[1,4] = \max\{ mkf(0,4),\ 8+mfk(0,2)\}$$
$$= \max\{ \ 0\ ,\ \ 8 + 0\} = 8$$
$$\text{Find } V[1,2] = \max\{ mkf(0,2),\ 8+mfk(0,0)\}$$
$$= \max\{ \ 0\ ,\ \ 8 + 0\} = 8$$
$$\text{Find } V[1,3] = \max\{ mkf(0,3),\ 8+mfk(0,0)\}$$
$$= \max\{ \ 0\ ,\ \ 8 + 0\} = 8$$
$$\text{Find } V[1,1] = \max\{ mkf(0,1)\} = 0$$

Back Substitute these values

Memory Function Knapsack

Example: Knapsack of capacity M = 5

item	weight	value
1	2	$8
2	1	$6
3	3	$16
4	2	$11

capacity j

		0	1	2	3	4	5						
	0	0	0	0	0	0	0	0	0	0	0	0	0
w = 2, p = 8	1	0	$\cancel{-1}$	-1	-1	-1	-1	0	0	8	8	8	8
w = 1, p = 6	2	0	-1	-1	-1	-1	-1	0	-1	8	14	-1	14
w = 3, p = 16	3	0	-1	-1	-1	-1	-1	0	-1	-1	16	-1	24
w_4 = 2, p_4 = 11	4	0	-1	-1	-1	-1	-1 ?	0	-1	-1	-1	-1	27

Memory Function Knapsack Algo.

ALGORITHM MFKnapsack(i, j) *//Implements the memory function method for the knapsack problem //Input: A nonnegative integer i indicating the number of the first // items being considered and a nonnegative integer j indicating // the knapsack capacity //Output: The value of an optimal feasible subset of the first i items //Note: Uses as global variables input arrays Weights[1..n], Values[1..n], //and table V[0..n, 0..W]whose entries are initialized with −1's except for //row 0 and 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]

MODULE – 5

BACKTRACKING

Backtracking

Backtrack' the Word was first introduced by Dr. D.H. Lehmer in 1950s.

- R.J Walker Was the First man who gave algorithmic description in 1960.
- Later developed by S. Golamb and L. Baumert.

Backtracking technique resembles a depth-first – search in a directed graph. The graph concerned here is usually a tree, the aim of backtracking is to search the state space tree systematically. The aim of the search is to find solutions to some problems.

What is Backtracking?

When the search begins, solution to the problem is unknown. Each move along an edge of the tree corresponds to adding a new element to a partial solution, that is to narrowing down the remaining possibilities for a complex solution.

The search is successful if, a solution can be completely defined. At this stage an algorithm may terminate or it may continue for an alternative solution.

The search is unsuccessful if at some stage the partial solution constructed so far cannot be completed. In this case the search backtracks like a depth first search, removing elements that were added at each stage.

State Space Tree

In state space tree, root represents an initial state before the search for a solution begins. The nodes of the first level in the tree represent the choice made for the first component of a solution, the nodes of the second level represent the choices for the second components, and so on. A node in a state space tree is said to be **promising** if it corresponds to a partially constructed solution that may lead to a complete solution; otherwise a node is said to be **non promising**.

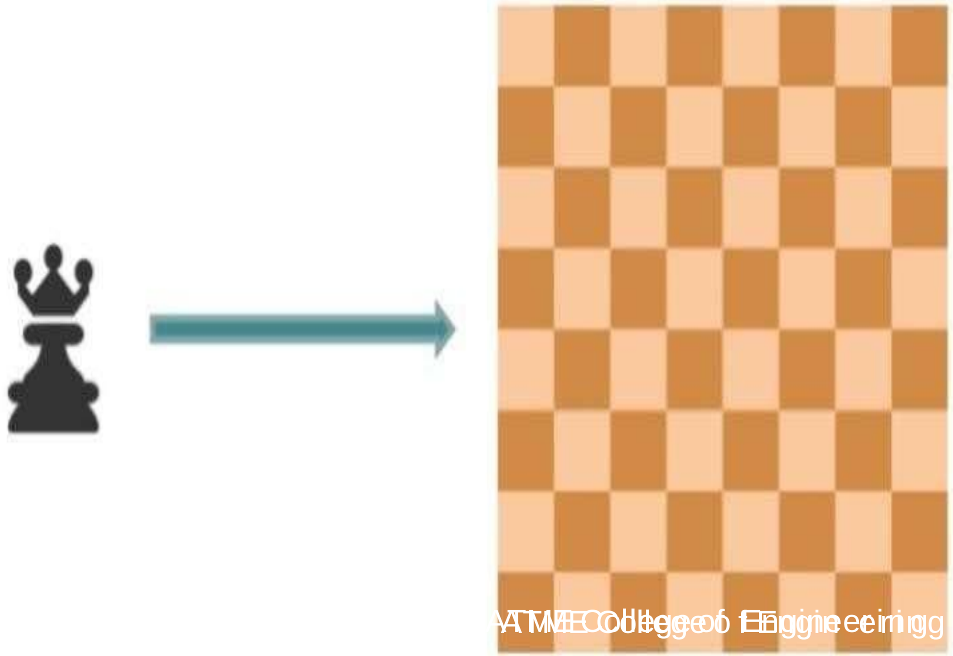
Application of Backtracking

- Optimization and tactical problems
- Constraints Satisfaction Problem
- Electrical Engineering
- Robotics
- Artificial Intelligence
- Genetic and bioinformatics Algorithm
- Materials Engineering
- Network Communication
- Solving puzzles and path

N-Queen Problem

History:

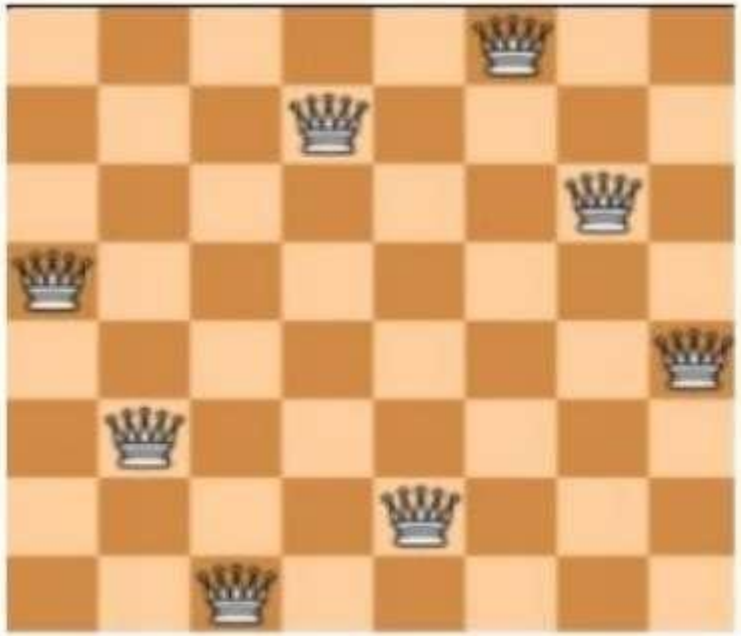
First Introduced in 1848 which was known as 8- queens Puzzle. Surprisingly, The First Solution was created in 1950 by Franz Nauck. Nauck made an 8X8 Chessboard to find the first Feasible Solution.



Problem Description

In a NxN square board N – number of queens need to be placed considering three Condition ---

- No two Queens can be placed in same row.
- No two Queens Can be places in same Column
- No two queens Can be placed in same Diagonal.



Constraints

Explicit Constraints: All ‘n’ queens must be placed on the chessboard in the columns 1,2,3, …. N. Xi belongs to S where S = {1,2,3, …. N }

Implicit Constraints: In this all Xi Values must be distinct
No two queens can be on the same row
No two queens can be on the same column
No two queens can be on the same diagonal

Horizontal Attack:

Row wise attacking is avoided by placing 1st queen in 1st row, 2nd queen in 2nd row and so on.

By placing ith queen in ith row, horizontal attacking can be avoided

Vertical Attack:

(i, x[i])→ means the position of ith queen in row i and column x[i]

(k, x[k])→ means the position of kth queen in row k and column x[k]

If ith & kth queen are in same column then

X[i] == x[k] --------------- (1)

Hence indicate that queens attack vertically

Q1			
Q4			

(1,1) & (4,1) x[i] == x[k] to be avoided

Diagonal Attack:

Top left corner to bottom right corner: The difference between row value and column value is same.

1,1	1,2	1,3	1,4
2,1	2,2	2,3	2,4
3,1	3,2	3,3	3,4
4,1	4,2	4,3	4,4

(1,3) & (2,4) |i - x[i]| = |k - x[k]|-------(2) to be avoided

Diagonal Attack:

Top right corner to bottom left corner: The difference between row value and column value is same.

1,1	1,2	1,3	1,4
2,1	2,2	2,3	2,4
3,1	3,2	3,3	3,4
4,1	4,2	4,3	4,4

(1,3) & (3,1) i + x[i] = k + x[k] ------(3) to be avoided

Using eqn. (2) and (3)

i – k = x[i] - x[k] -----------------(4)

i – k = - x[i] + x[k] ----------------(5)

|i – k| = |x[i] - x[k]| indicates queens attack diagonally.

X[i] == x[k] | | abs(i –k) = abs(x[i] – x[k]) → two queens attack each other and cannot be placed.

Algorithm

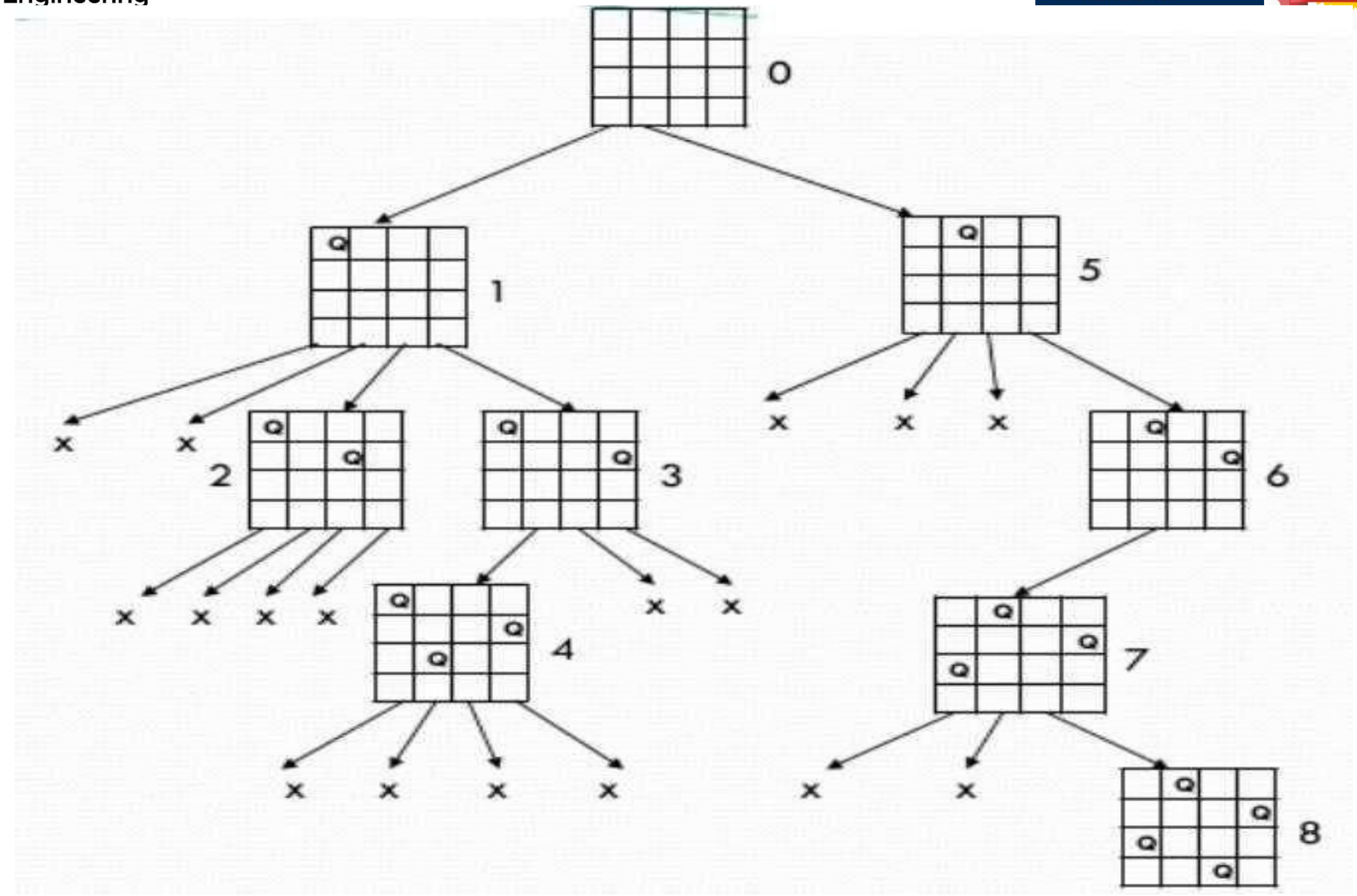
Backtracking approach for solution

```
Algorithm Nqueen(K,n){
    For i= 1 to n {
        If Place(K,i){
            X[k] = I;
            If(k = n) then
            Write x[1:n];
          Else
            Nqueen(k+1, n) ;
        }
    }
}
```

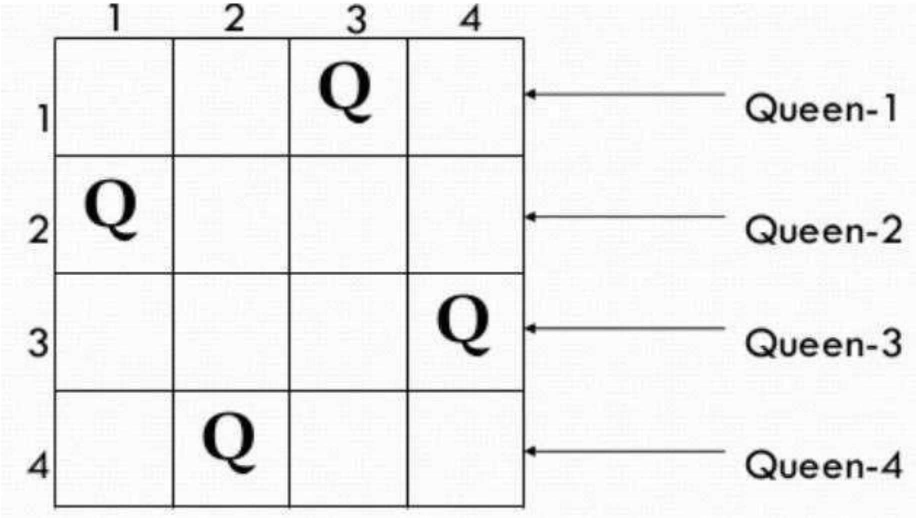
```
Place(k,i){
    For j=1 to k-1{
        If((x[j] = i) or abs(x[j] – 1 = abs(j-k)))
            Return false;
    }
    Return true;
}
```

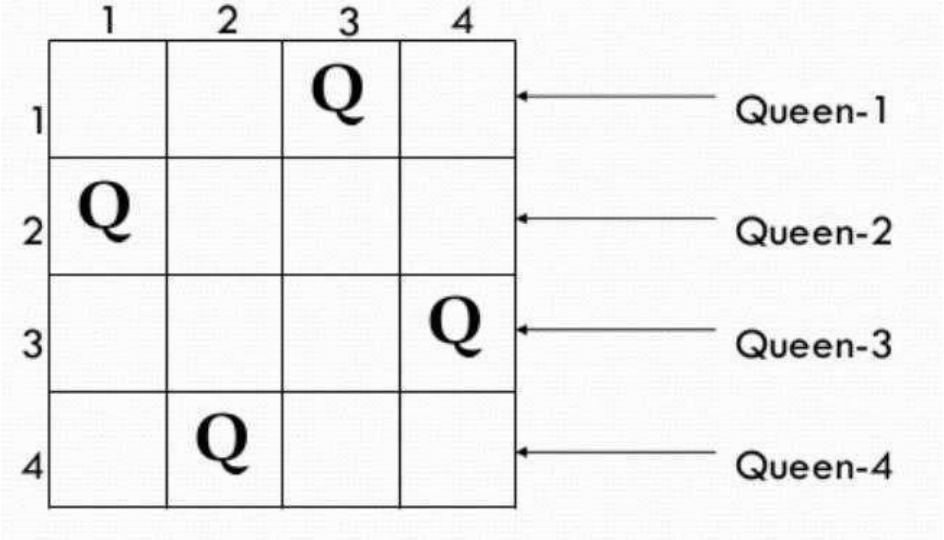
-- The Algorithm will check each position [i , j] for each queens . If any Suitable places found , It will place a queen on that position. If not Algorithm will try same approach for next position.

State Space Tree for 4 Queens



Two Solutions of 4 Queen Problem





Hamiltonian Cycle

Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then print the path. Following are the input and output of the required function.

Input:

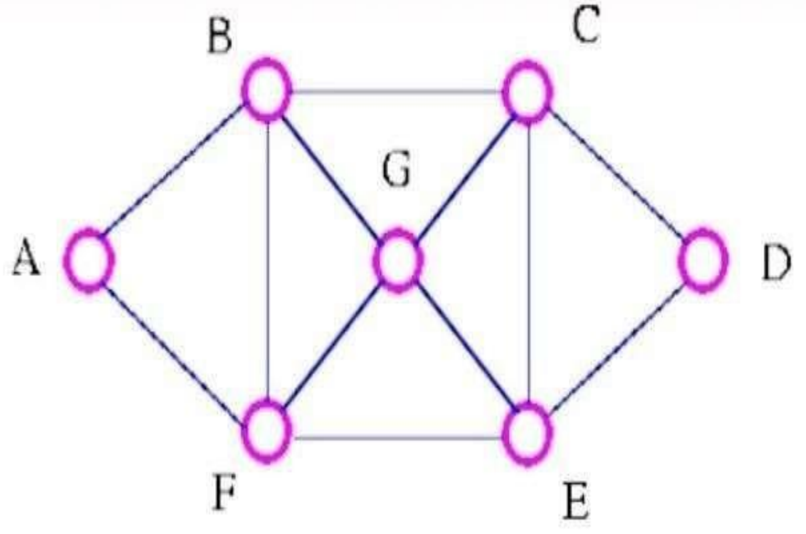
A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.

Output:

An array path[V] that should contain the Hamiltonian Path. path[i] should represent the ith vertex in the Hamiltonian Path. The code should also return false if there is no Hamiltonian Cycle in the graph.

Example 2

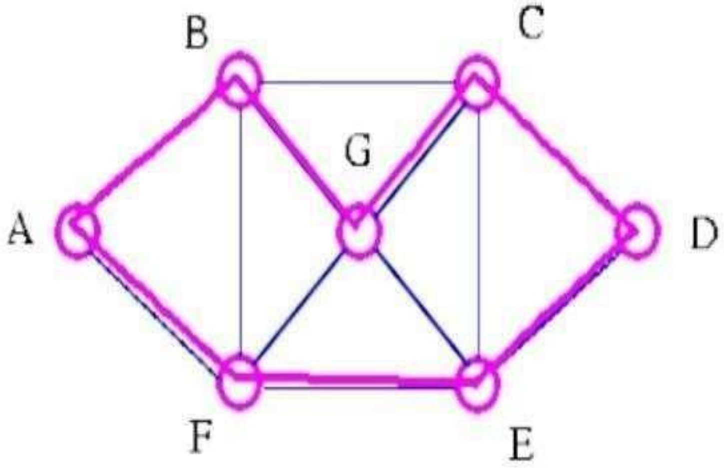
Hamiltonian Cycle is a graph theory problem where the graph cycle through a graph can visit each node only once. The puzzle was first devised by Sir William Rowan Hamilton and the Problem is named after Him.



Condition: The Cycle Started with a Starting Node, and visit all the Nodes in the Graph (Not necessary to visit in sequential Order and not creating edge that is not given) And Stop at The Starting Point/Node.

The Backtracking Approach

The Algorithm First Check the Starting Node, if there is any edge to the next node. If yes, then the Algorithm will check that node for the edge to the next Node. It will Also Check If any Node is visited twice by the previous Node. If there is any then the Algorithm Will Ignore One and Choose the Optimal One for the Solution.



The Important thing is the tour must finish at the starting point.

Example

For example consider the given graph and evaluate the mechanism:-

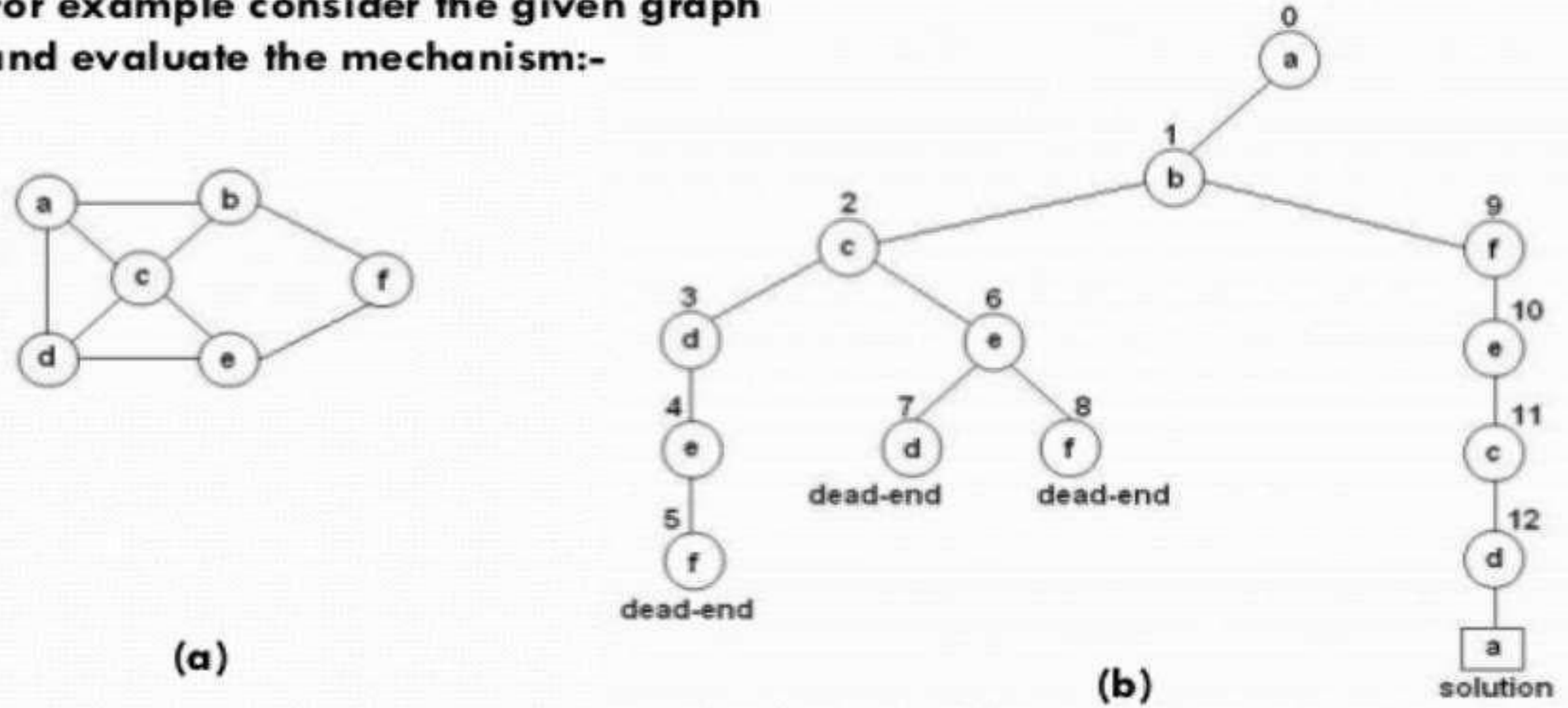


Figure: • (a) Graph.

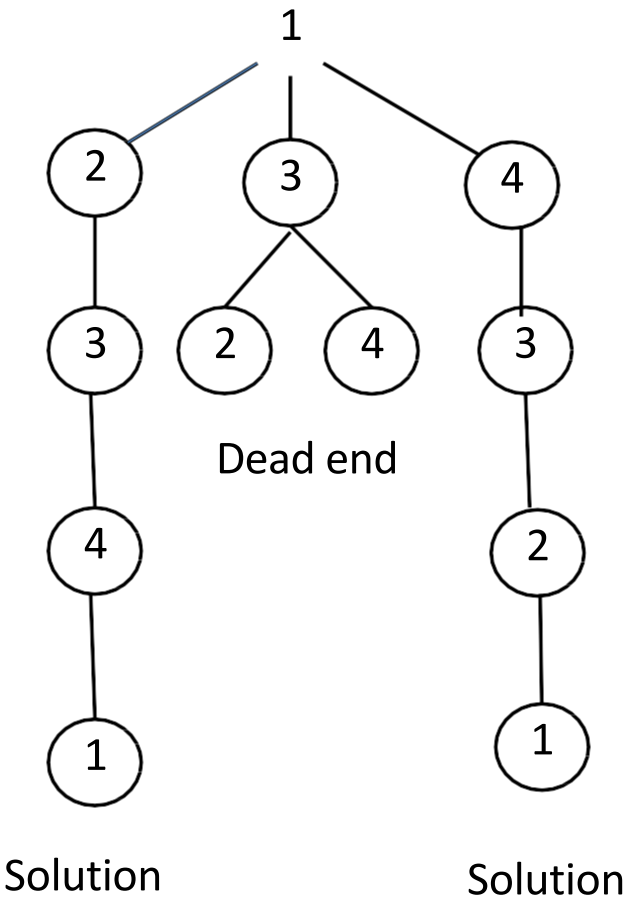
• (b) State-space tree for finding a Hamiltonian circuit. The numbers above the nodes of the tree indicate the order the order in which nodes are generated.

Hamiltonian Cycle

X[1] =1	X[2] = 0	X[3]=0	X[4]=0

1 2

4 3



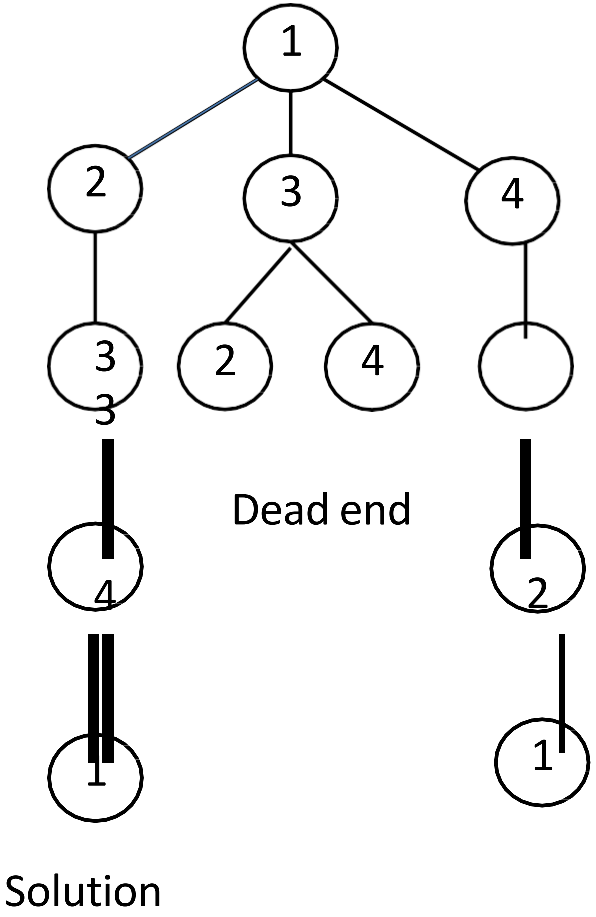
```
Algorithm Hamilton_Cycle(k){
Repeat{
      NextVal(k);
      If(x[k]==0) then
        Return;
       If (k ==n) then
          Write (x[1:n]);
       Else
         Hamilton_Cycle(k+1);
  }
  Until (false);
}
```

```
AlgortihmNextVal(k){
Repeat{
        X[k] = (x[k]+1) mod (n+1);
        If(x[k]=0) then return;
        If (G[x[k-1],x[k]] != 0 ) then{
             For j = 1 to k-1 do
               If(x[j]=x[k]) then
                   Break;
               If(j = k) then
                 If ((k<n or k=n) && G[x[n],x[1]] != 0)
                    Then return;
        }
    }
    Until (false);
}
```

```
for (int i = 1; i <= n; i++)
        x[i] = 0;
x[1] = 1;

void HamiltonianMethod(int k) {
while (true) {
    NextValue(k, G, x, n);
    if (x[k] == 0)

    return;
        if (k == n) {
        for (int i = 1; i <= k; i++)
        System.out.print(x[i] + " ");
        System.out.println(x[1]);
        System.out.println();
        found = true;
        return;
        } else
        HamiltonianMethod(k + 1);
    }
    }
```



```
void NextValue(int k, int G[][], int x[], int n) {
while (true) {
x[k] = (x[k] + 1) % (n + 1);
if (x[k] == 0)
return;
if (G[x[k - 1]][x[k]] != 0) {
int j;
for (j = 1; j < k; j++)
if (x[k] == x[j])
break;
if (j == k)
if ((k < n) || ((k == n) && G[x[n]][x[1]] != 0))
return;
}
}
}
```

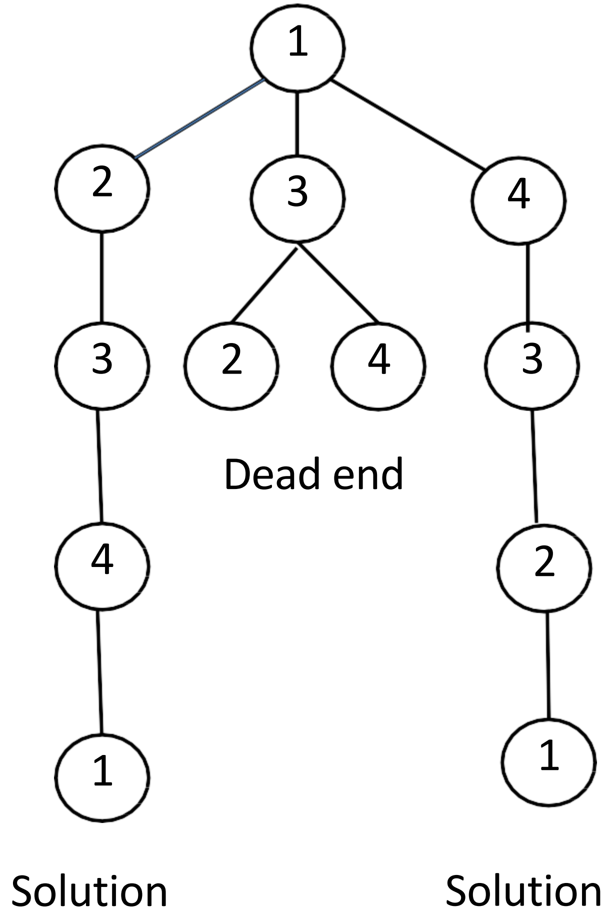
Hamiltonian Cycle

Enter the number of the vertices: 4

If edge between the following vertices enter 1 else 0:
1 and 2: 1
1 and 3: 1
1 and 4: 1
2 and 3:
1
2 and 4: 0
3 and 4: 1

Solution:
1 2 3 4 1

1 4 3 2 1


1
2
3
4
3
2
4
3
4
2
1
1
Dead end
Solution
Solution

C:\Windows\system32\cmd.exe

```
Solution exists
 0
 1
 2
 4
 3
 0
No solution
Press any key to continue . . .
```

Sum of subsets

- **Subset-sum Problem:** The problem is to find a subset of a given set $S = \{s_1, s_2, - - -, s_n\}$ of **'n'** positive integers whose sum is equal to a given positive integer **'d'**.

- **Observation :** It is convenient to sort the set's elements in increasing order, $S_1 \leq S_2 \leq \ldots\ldots \leq S_n$. And each set of solutions don't need to be necessarily of fixed size.

- **Example :** For $S = \{3, 5, 6, 7\}$ and $d = 15$, the solution is shown below :-

  Solution = $\{3, 5, 7\}$

Sum of subsets

The Sum of Subset Problem is, there will be a set of distinct positive Numbers X and a given Number N. Now, we have to find all the combination of numbers from X that sum up to N. Make a Set of those Number.

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

$$M = 13$$

Sum of Subsets

Find a subset of a given set S= {S1, S2, S3, S4, ------ Sn}
Of n +ve integers whose sum is equal to given +ve integer **d** subject to the constrains

1. **Implicit:** All Xi values should be distinct and should belong to the set S
2. **Explicit:** optimal solution be $\sum_{i=1} i = d$

Xi of the solution vector is either **1 or 0** depending on weather the weight Wi is included or not.

For a node at level i, the left child corresponds to Xi = 1 and the right child to Xi = 0

The bounding function X[X1, X2, X3, -----Xn]   = true iff

$$\sum_{i=1} i \qquad i + \sum_{i=+1} i \qquad \geq$$

X1, X2, -----Xk cannot lead to an promising node if this condition is not satisfied.

The bounding function can be strengthened if we assume that Wi's are initially in increasing order.

In this case X1 – Xk can not lead to promising node if
X[X1, X2, X3, -----Xn] = true iff

$$\sum_{i=1} i \quad i+ \quad + 1 \qquad \leq$$

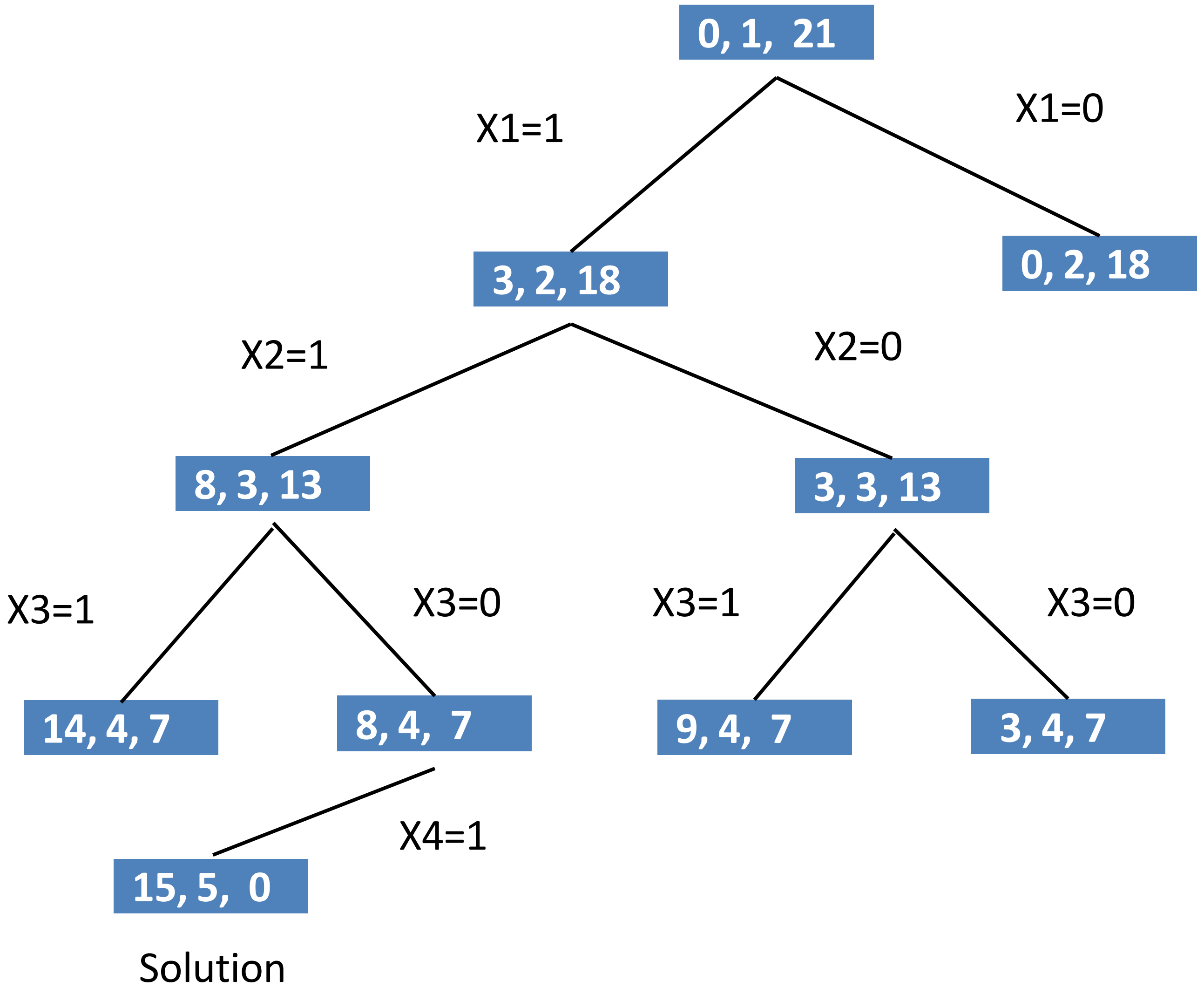
X1, X2, -----Xk cannot lead to an promising node if this condition is not satisfied.

Therefore the bounding function will be

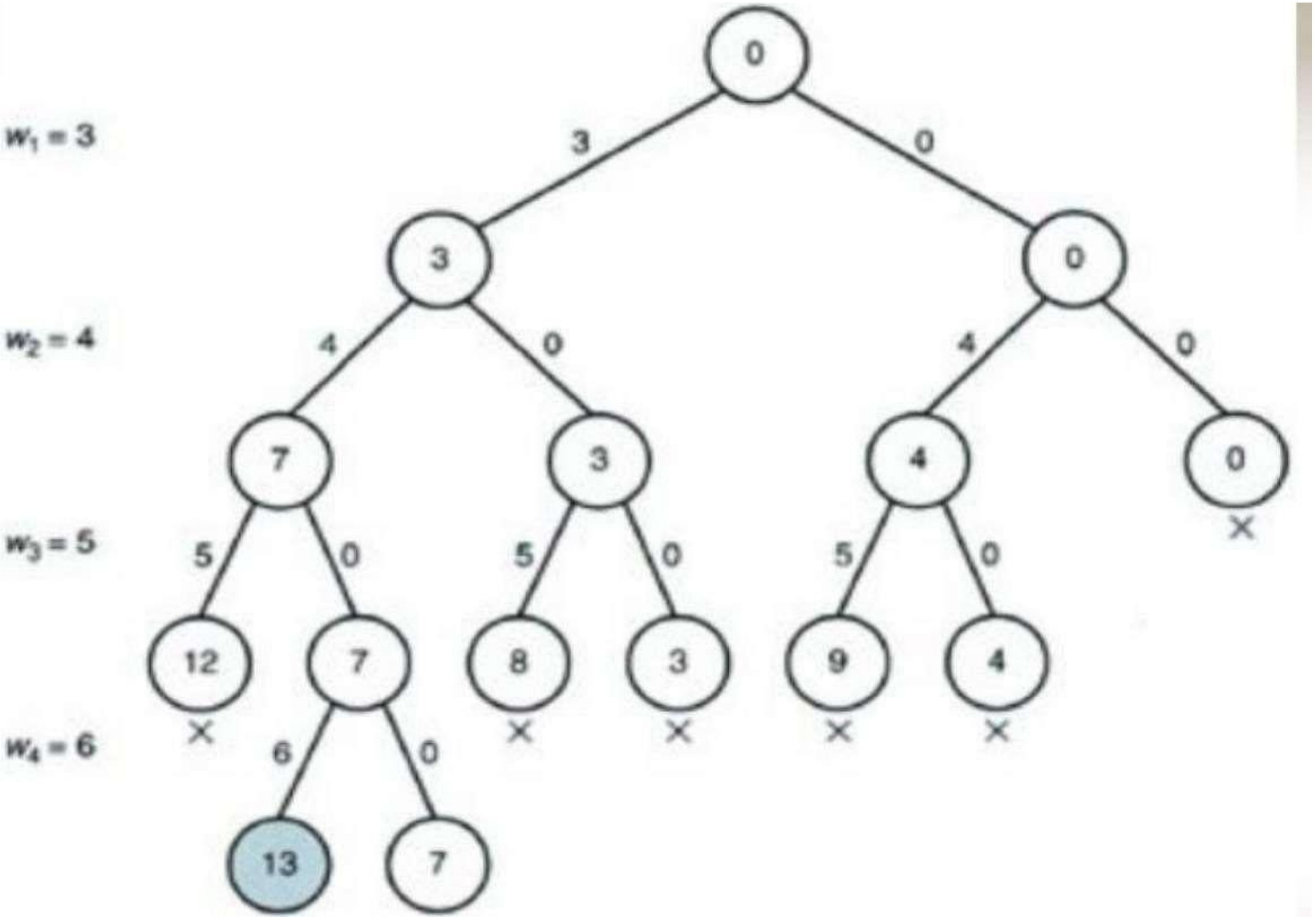
$$\sum_{i=1} i \quad i + \sum_{i=+1} i \quad \geq$$

$$\sum_{i=1} i \quad i + \quad {}_{+\,1} \quad \leq$$

State space tree



Sum of subsets



Sum of subsets

Backtracking Approach

First, organize the numbers in non decreasing order. Then generating a tree, we can find the solution by adding the weight of the nodes. Note that, here more than necessary nodes can be generated. The algorithm will search for the weighted amount of the nodes. If it goes beyond given Number N, then it stops generating nodes and move to other parent nodes for finding solution.

```
Algorithm SumOfSubset(s,k,y){
   X[k] = 1;
   If(s+w[k] = m)
        Write (x[1:n]);
    Else if((s+w[k] + w[k+1]) <= m)
SumOfSubset(s+w[k], k+1, y-w[k]);
    If ((s+ y-w[k]>=m) &&(s =w[k+1] <=m)) {
      X[k] =0;
SumOfSubset(s,k+1,y-w[k]);
    }
}
```

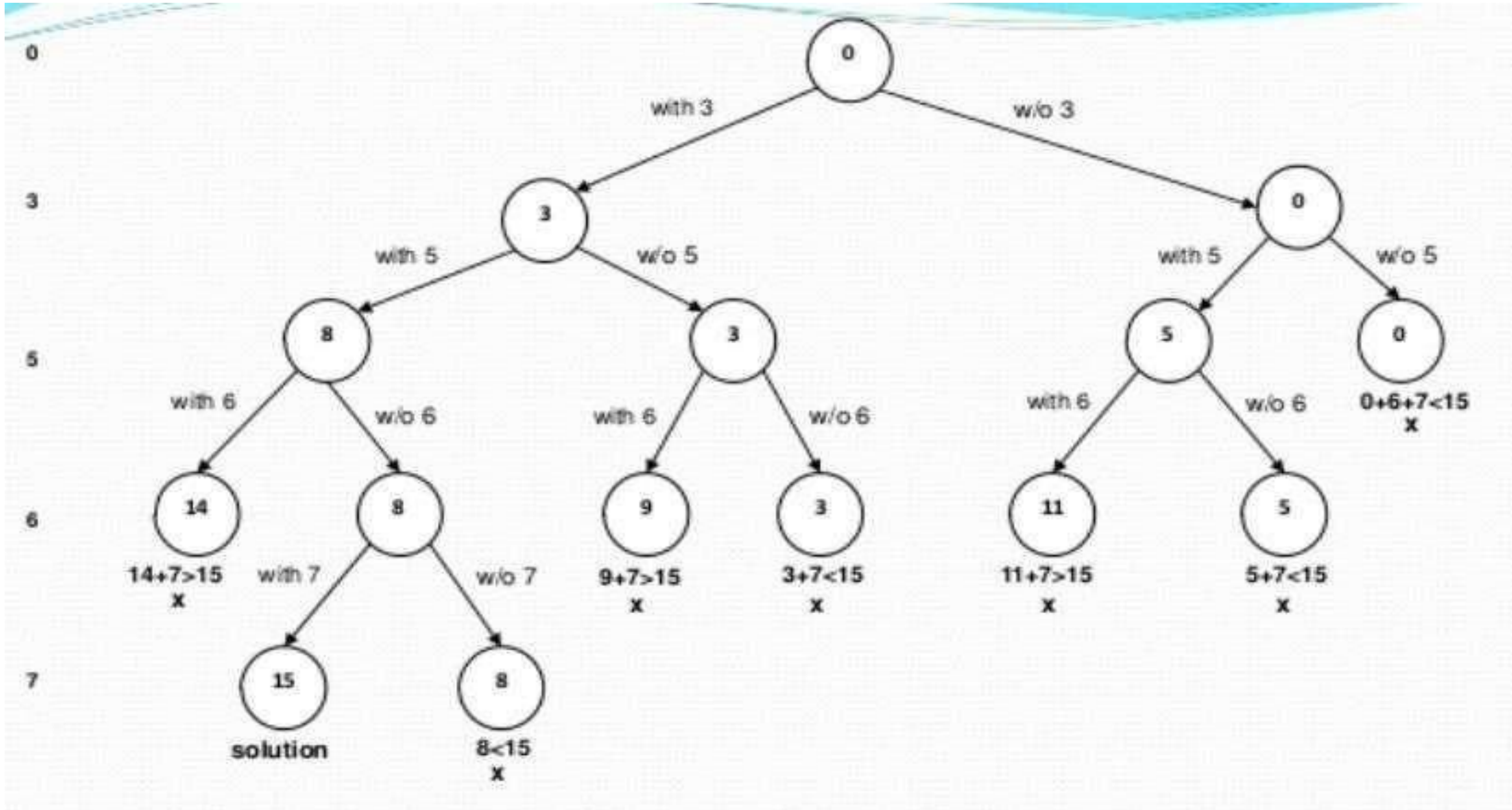


Figure : Compete state-space tree of the backtracking algorithm applied to the instance **S =** **{3, 5, 6, 7}** and **d = 15** of the subset-sum problem. The number inside a node is the sum of the elements already included in subsets represented by the node. The inequality below a leaf indicates the reason for its termination.

Coding

```
for (int i = 1; i <= n; i++)
sum = sum + S[i];
if (sum < d || S[1] > d)
System.out.println("No Subset possible");
else
SumofSub(0, 0, sum);
```

ATME College of Engineering

```
static void SumofSub(int i, int weight, int total)
{ if (promising(i, weight, total) == true)
if (weight == d) {
for (int j = 1; j <= i; j++) {
if (soln[j] == 1)
System.out.print(S[j] + "   ");
}
System.out.println();
} else {
soln[i + 1] = 1;
SumofSub(i + 1, weight + S[i + 1], total - S[i + 1]);
soln[i + 1] = 0;
SumofSub(i + 1, weight, total - S[i + 1]);
}
}
```

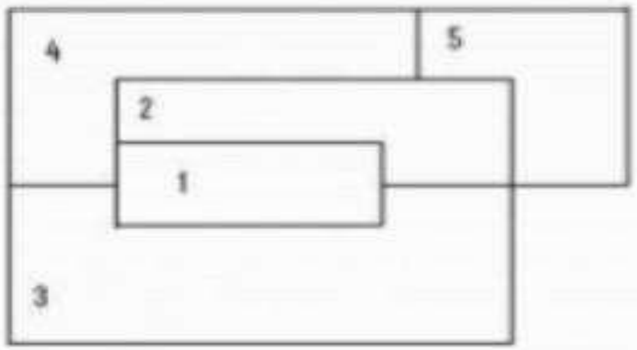
```
static boolean promising(int i, int weight, int total) {
return ((weight + total >= d) && (weight == d ||
weight + S[i + 1] <= d));
```

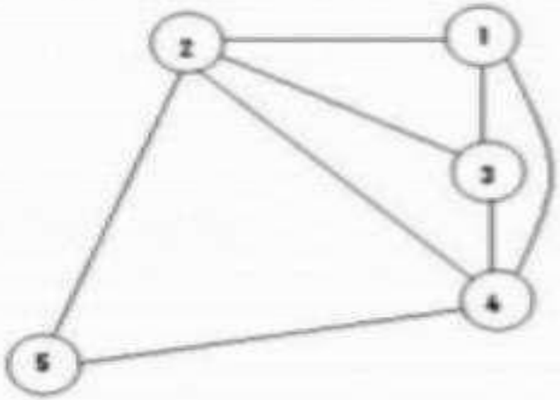
Coloring a map

Problem:

Let **G** be a graph and **m** be a given positive integer. We want to discover whether the nodes of **G** can be colored in such a way that no two adjacent node have the same color yet only **m** colors are used. This technique is broadly used in "map-coloring"; Four-color map is the main objective.

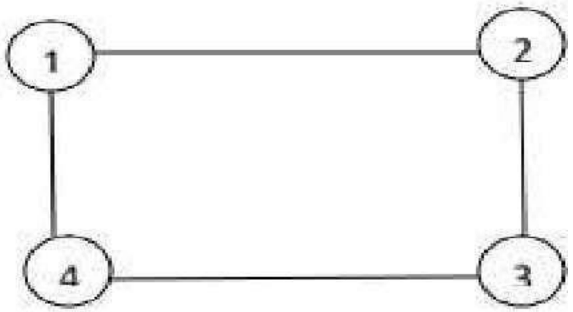
Consider the following map and it can be easily decomposed into the following planner graph beside it :





ALGORITHM FOR GRAPH COLORING

```
Algorithm mcolor(k)
{
    The graph is represented in the form of matrix nxn
    "k" is an index of vertex to be colored
    {
        repeat
        {
                Nextvalue(k)
                If (x[k] = 0) then return
                If (k=n) then
                        Write(x[1:n])
                Else
                        Mcolor(k+1)
        } until(false)
    }
}
```



```
Algorithm nextvalue(k)
Assume x[1..k-1] are assigned integer in the range of [1,m] such that no
two adjacent vertices are in the same color
x[k] is assigned the next value such that distinctness is maintained
If no such color exists then x[k]=0
{
    repeat
    {
        x[k] = (x[k] + 1) mod m+1
        if x[k] = 0 then return
        for j = 1 to n do
        {
            If (G[k,j]!=0) and (x[k]=x[j]) then
                Break
        }
        If (j=n+1) then
            Return
    }until (false)
}
```

Branch and Bound

The term Branch means the way in which we search the state space tree and Bound means assigning bounding function at each node. This bounding function is used to prevent the expansion of nodes that cannot possibly lead to an answer node.

Basically there are two methods used in branch and bound technique.

1. FIFO based Branch & Bound
2. In this method, the live node form a queue (FIFO Structure) & each live node will be taken from the queue and next live node is selected.

Least Cost Branch and Bound

At each node, an intelligent ranking function is used to assign a value to that node. The next live node is selected on the basis of the least cost.

Travelling sales man problem, a sales man must visit n cities. The sales man visits each city exactly once and comes back to the starting city.

The travelling sales man problem is minimization problem and hence we require to find the lower bound.

Example 1

Assignment Problem : given n jobs <j1,j2,--- jn> and n persons <p1,p2,p3 ---pn>, it is required to assign all n jobs to all n persons with the constraint that one job has to be assigned to one person and the cost involved in completing all the jobs should be minimum.

	J1	J2	j3	j4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

Example 1

	J1	J2	j3	j4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

Take minimum in each row

2
3
1
4

10

	a → J1	a → J2	a → J3	a → J4
a	9	2	7	8
b	3	3	4	3
c	8	5	5	5
d	4	4	4	6
	24	14	20	22

Example 1

	J1	J2	J3	J4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

Take minimum in each row

2
3
1
4

10

	b → J1	b → J3	b → J4
a	2	2	2
b	6	3	7
c	1	5	1
d	4	4	7
	13	14	17

Example 1

	J1	J2	J3	J4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

Take minimum in each row

2
3
1
4
10

	C → J3	C → J4
a	2	2
b	6	6
c	1	8
d	4	9
	13	25

	C → J4
a	2
b	6
c	1
d	4
	13

Start
Lb=10

a –> J1
Lb=24

a – >J2
Lb=14

a –> J3
Lb=20

a –> J4
Lb=22

b –> J1
Lb=13

b –> J3
Lb=14

c –> J4
Lb=17

c –> J3
Lb=13

c –> J4
Lb=25

d –> J4
Lb=13

Example 2

	J1	J2	j3	j4
A	10	3	8	9
B	7	5	4	8
C	6	9	2	9
D	8	7	10	5

Knapsack Problem

Knapsack Problem: Given n items of known weights wi and values vi, i=1, 2, . . . , n,and a knapsack of capacity W, find the most valuable subset of the items that fit in the knapsack. It is convenient to order the items of a given instance in descending order by their value-to-weight ratios. Then the first item gives the best payoff per weight unit and the last one gives the worst payoff per weight unit

item	weight	value	$\frac{\text{value}}{\text{weight}}$
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4

The knapsack's capacity W is 10.

item	weight	value	$\frac{\text{value}}{\text{weight}}$
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

The knapsack's capacity W is 10.

First arrange in V/W in decreasing order

Since it is a maximization problem. The upper bound is calculated using the function $ub = v + (W - w)(v_{i+1}/w_{i+1})$.

i =0 , v=0, w=0 v_{i+1}/w_{i+1} = 10

Ub = 0 + (10) 10 = 100

$w=0.\ v=0$

$Ub = 100$

With item 1

$i=1$, $w=4.\ v=40$, $v_{i+1}/w_{i+1} = 6$

$Ub = v + (W-w)(v_{i+1}/w_{i+1})$

$= 40 + 6.\ 6$

$= 76$

Without item 1

$i=1$, $w=0.\ v=0$, $v_{i+1}/w_{i+1} = 6$

$Ub = v + (W-w)(v_{i+1}/w_{i+1})$

$= 0 + 10\ .\ 6$

$= 60$

With item 2

$i=2$, $w=7.\ v=42$, $v_{i+1}/w_{i+1} = 5$

$Ub = v + (W-w)(v_{i+1}/w_{i+1})$

$= 42 + (10 - 11)\ .\ 5$

= Not Feasible

With out item 2

$i=2$, $w=0+4.\ v=40+0$, $v_{i+1}/w_{i+1} = 5$

$Ub = v + (W-w)(v_{i+1}/w_{i+1})$

$= 40 + 6.\ 5$

$= 70$

w=0. v=0

Ub = 100

With item 3
i =3, w=5+4. v=40+25, v_{i+1}/w_{i+1} = 4

Ub = v + (W-w)(v_{i+1}/w_{i+1})
= 65 + 1. 4
= 69

Without item 3
i =3, w=4+0. v=40+0, v_{i+1}/w_{i+1} = 4

Ub = v + (W-w)(v_{i+1}/w_{i+1})
= 40 + 6 . 4
= 64

With item 4
i =4, w=9+3 =12.

Ub = v + (W-w)(v_{i+1}/w_{i+1})
= Not Feasible

With out item 4
i =4, w=0+9. v=65+0, v_{i+1}/w_{i+1} = 1

Ub = v + (W-w)(v_{i+1}/w_{i+1})
= 65 + 1. 1
= 66

w=0. v=0

Ub = 100

With item 1
i =1, w=4. v=40, v $_{i+1}/w_{i+1}$ = 6
= 76

Without item 2
i =1, w=0. v=0, v $_{i+1}/w_{i+1}$ = 6
= 60

With item 2
i =2, w=7. v=42, v $_{i+1}/w_{i+1}$ = 5
= Not Feasible

With out item 2
i =2, w=0+4. v=40+0, v $_{i+1}/w_{i+1}$ = 5
= 70

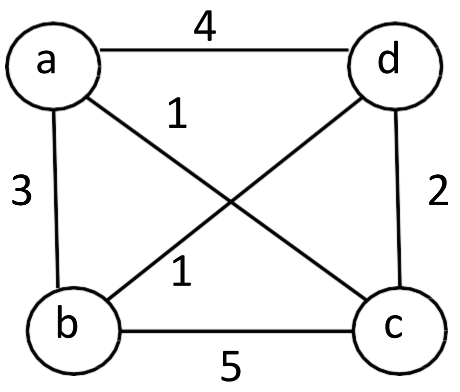
With item 3
i =3, w=5+4. v=40+25, v $_{i+1}/w_{i+1}$ = 4
= 69

Without item 3
i =3, w=4+0. v=40+0, v $_{i+1}/w_{i+1}$ = 4
= 64

With item 4
i =4, w=9+3 =12.
Not Feasible

With out item 4
i =4, w=0+9. v=65+0, v $_{i+1}/w_{i+1}$ = 0
= 65

Travelling Sales Person Problem



In the travelling sales man problem, a sales man must visit n cities. The sales man visits each city exactly once and comes back to the starting city.
The travelling sales man problem is minimization problem and hence we require to find the lower bound.

Lower bound = lb = S / 2;
Where S = [Va+ Vb+Vc+Vd]
Va = sum of distances from vertex a to the nearest vertices 1 + 3 = 4
Vb = 1+3 = 4
Vc = 1+2= 3
Vd = 1+2 = 3
Lb = [4 +4 +3+3] / 2 = 14 / 2 = 7

Now find,

a → b = (3+1)+(3+1)+(1+2)+(1+2) = 14/ 2 = 7

a → c = (1+3)+(3+1)+(1+2)+(1+2) = 14/ 2 = 7

a → d = (4+1)+(1+3)+(1+2)+(4+1) = 17/2 = 8

Start

Lb=7

a –> b

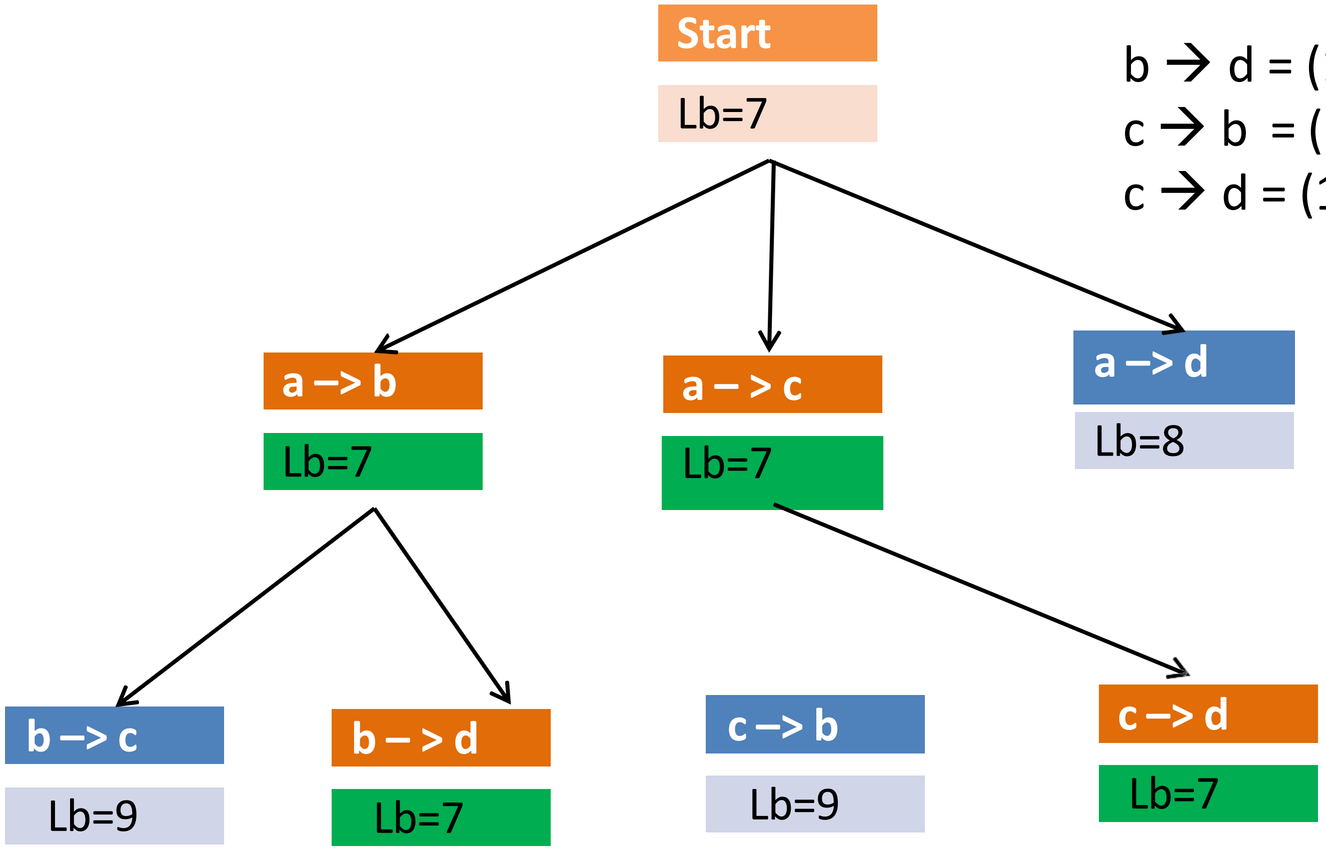
Lb=7

a – > c

Lb=7

a –> d

Lb=8



Now find,

$b \rightarrow c = (3+1)+(5+1)+(5+1)+(1+2) = 19/2 = 9$

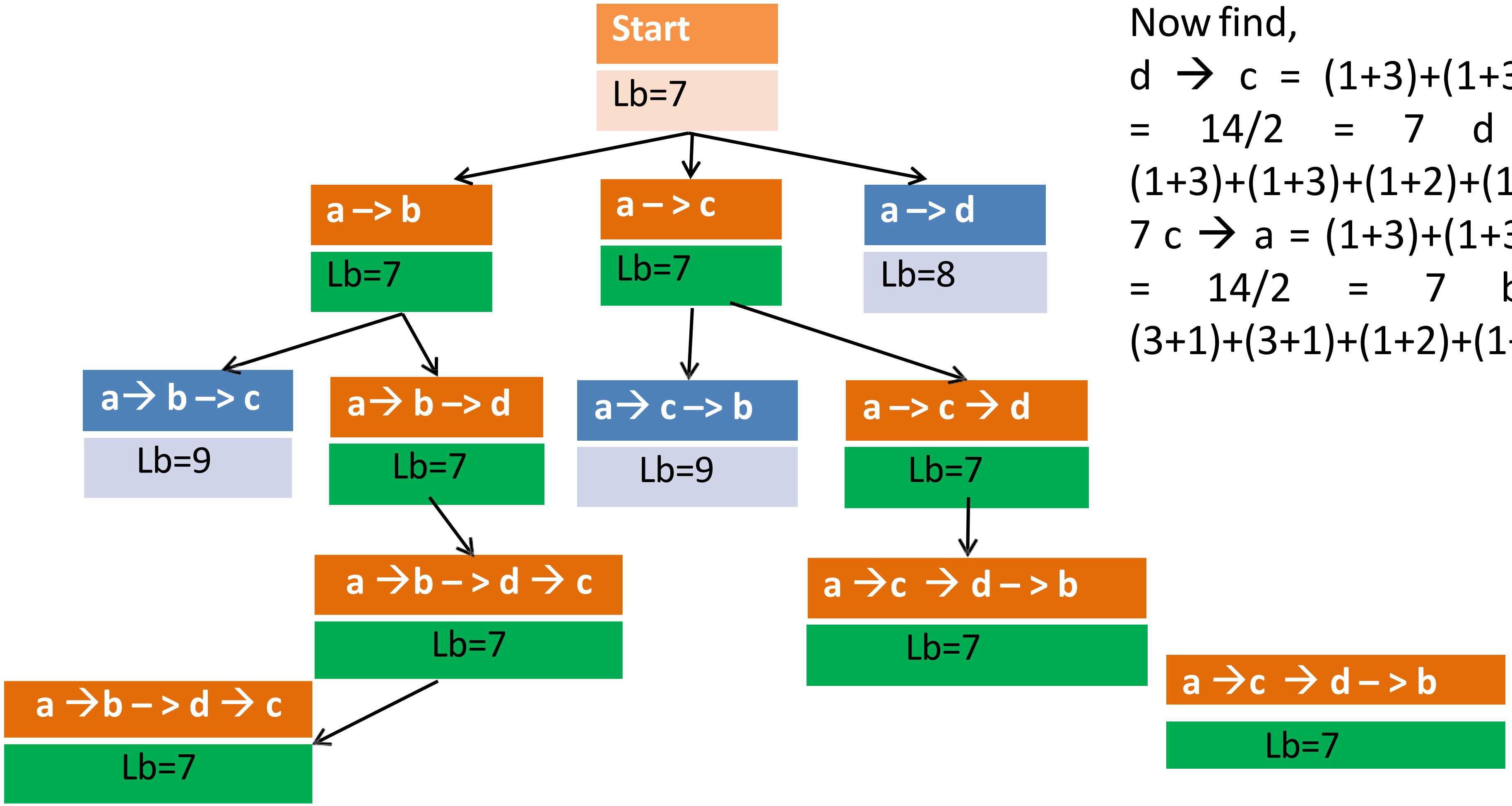
$b \rightarrow d = (1+3)+(1+3)+(1+2)+(1+2) = 14/2 = 7$

$c \rightarrow b = (1+3)+(5+1)+(5+1)+(1+2) = 19/2 = 9$

$c \rightarrow d = (1+3)+(3+1)+(2+1)+(2+1) = 14/2 = 7$

Now find,

d → c = (1+3)+(1+3)+(2+1)+(1+2) = 14/2 = 7 d → b = (1+3)+(1+3)+(1+2)+(1+2) = 14/2 = 7 c → a = (1+3)+(1+3)+(1+2)+(1+2) = 14/2 = 7 b → a = (3+1)+(3+1)+(1+2)+(1+2) = 14/2 = 7



Thank you