INTRODUCTION

1.1 Notion of Algorithm

1.2 Review of Asymptotic Notation

1.3 Mathematical Analysis of Non-Recursive and Recursive Algorithms

1.4 Brute Force Approaches: Introduction

1.5 Selection Sort and Bubble Sort

1.6 Sequential Search and Brute Force String Matching.
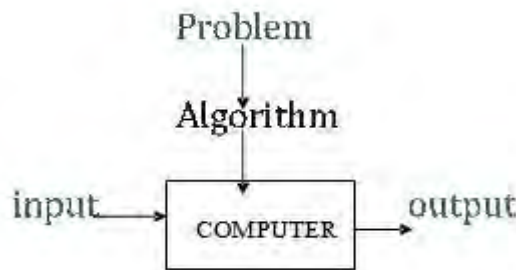
1.1 Notion of Algorithm

Need for studying algorithms:

The study of algorithms is the cornerstone of computer science. It can be recognized as the core of computer science. Computer programs would not exist without algorithms. With computers becoming an essential part of our professional & personal life's, studying algorithms becomes a necessity, more so for computer science engineers. Another reason for studying algorithms is that if we know a standard set of important algorithms, They further our analytical skills & help us in developing new algorithms for required applications

Algorithm

An algorithm is finite set of instructions that is followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

1. Input. Zero or more quantities are externally supplied.

2. Output. At least one quantity is produced.

3. Definiteness. Each instruction is clear and produced.

4. Finiteness. If we trace out the instruction of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.

5. Effectiveness. Every instruction must be very basic so that it can be carried out, in principal, by a person using only pencil and paper. It is not enough that each operation be definite as in criterion 3; it also must be feasible.

An algorithm is composed of a finite set of steps, each of which may require one or more operations. The possibility of a computer carrying out these operations necessitates that certain constraints be placed on the type of operations an algorithm can include. The fourth criterion for algorithms we assume in this book is that they terminate after a finite number of operations.

Criterion 5 requires that each operation be effective; each step must be such that it can, at least in principal, be done by a person using pencil and paper in a finite amount of time. Performing arithmetic on integers is an example of effective operation, but arithmetic with real numbers is not, since some values may be expressible only by infinitely long decimal expansion. Adding two such numbers would violet the effectiveness property.

- Algorithms that are definite and effective are also called computational procedures.
- The same algorithm can be represented in same algorithm can be represented in several ways
- Several algorithms to solve the same problem
- Different ideas different speed

Example:

Problem:GCD of Two numbers m,n

Input specifiastion :Two inputs,nonnegative,not both zero

Euclids algorithm

-gcd(m,n)=gcd(n,m mod n)

Untill m mod n =0,since gcd(m,0) =m

Another way of representation of the same algorithm

Euclids algorithm

Step1:if n=0 return val of m & stop else proceed step 2

Step 2:Divide m by n & assign the value of remainder to r

Step 3:Assign the value of n to m,r to n,Go to step1.

Another algorithm to solve the same problem

Euclids algorithm

Step1:Assign the value of min(m,n) to t

Step 2:Divide m by t.if remainder is 0,go to step3 else goto step4

Step 3: Divide n by t.if the remainder is 0,return the value of t as the answer and stop,otherwise proceed to step4

Step4 :Decrease the value of t by 1. go to step 2

1.1 Review of Asymptotic Notation

Fundamentals of the analysis of algorithm efficiency

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

- **running time (time efficiency)**

- **memory space (space efficiency)**

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

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

Limitations of Experiments

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

Theoretical Analysis: It uses a high-level description of the algorithm instead of an implementation. Analysis characterizes running time as a function of the input size, n, and takes into account all possible inputs. This allows us to evaluate the speed of an

algorithm independent of the hardware/software environment. Therefore theoretical analysis can be used for analyzing any algorithm

Framework for Analysis

We use a hypothetical model with following assumptions

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

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

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

Consider the following example:

ALGORITHM sum_of_numbers (A[0… n-1])

// Functionality : Finds the Sum

// Input : Array of n numbers

// Output : Sum of „n‟numbers

i   0

sum   0

while i < n

sum   sum + A[i] $\longrightarrow$ n

i   i + 1

return sum

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

NOTE:

Constant of fastest growing term is insignificant: Complexity theory is an Approximation theory. We are not interested in exact time required by an algorithm to solve the problem. Rather we are interested in order of growth. i.e How much faster will algorithm run on computer that is twice as

fast? How much longer does it take to solve problem of double input size? We can crudely estimate running time by

$$T(n) \approx C_{op} \, \diamond \, C(n)$$

Where,

T (n): running time as a function of n.

C_{op} : running time of a single operation.

C (n): number of basic operations as a function of n.

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

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

Worst-case, Best-case, Average case efficiencies

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

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

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

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

Asymptotic Notations

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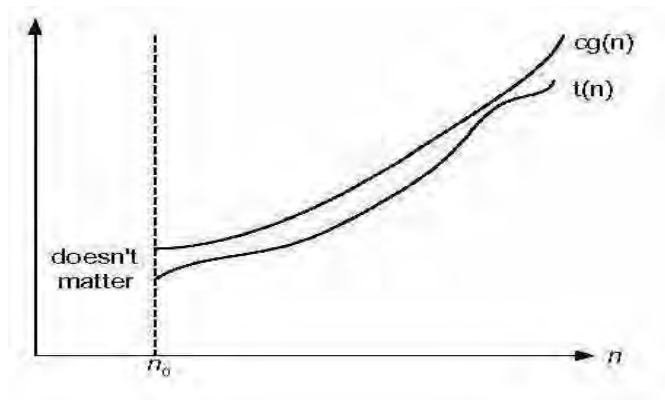
O, Ω, Θ notations

Big Oh- O notation

Definition:

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

t(n) ≤ cg(n) for all n ≥ n0

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

Big Omega- Ω notation

Definition:

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

t(n) ≥ cg(n) for all n ≥ n0

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

Big Theta- Θ notation

Definition:

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

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

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

Basic Efficiency classes

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

High time efficiency

fast

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

low time efficiency

slow

1.2 Mathematical Analysis of Non-Recursive and Recursive Algorithms

Mathematical analysis (Time Efficiency) of Non-recursive Algorithms

General plan for analyzing efficiency of non-recursive algorithms:

1. Decide on parameter n indicating **input size**

2. Identify algorithm's **basic operation**

3. Check whether the number of times the basic operation is executed depends only on the input size n. If it also depends on the type of input, investigate **worst, average, and best case efficiency** separately.

4. Set up **summation** for C(n) reflecting the number of times the algorithm's basic operation is executed.

5. Simplify summation using standard formulas

Example: Finding the largest element in a given array

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

//Determines the value of largest element in a given array

//nput: An array A[0..n-1] of real numbers

//Output: The value of the largest element in A

currentMax ← A[0]

for i ← 1 to n - 1 do

if A[i] > currentMax

currentMax ← A[i]

return currentMax

Analysis:

1. Input size: number of elements = n (size of the array)

2. Basic operation:

   a) Comparison

   b) Assignment

3.
4. NO best, worst, average cases.

   Let C (n) denotes number of comparisons: Algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bound between 1 and $n - 1$.

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

5. **Simplify summation** using standard formulas

$$C(n) = \sum_{i=1}^{n-1} 1 \qquad \begin{array}{c} 1 + 1 + 1 + \dots + 1 \\ [(n-1) \text{ number of times}] \end{array}$$

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

$$C(n) \; \square \; \Theta(n)$$

Example: Element uniqueness problem

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

//Checks whether all the elements in a given array are distinct

//Input: An array A[0..n-1]

//Output: Returns true if all the elements in A are distinct and false otherwise

for i   0 to n - 2 do

for j   i + 1 to n − 1 do

if A[i] = = A[j]

return *false*

return *true*

Analysis

1. Input size: number of elements $= n$ (size of the array)

2. Basic operation: Comparison

3. Best, worst, average cases EXISTS.

Worst case input is an array giving largest comparisons.

- Array with no equal elements

- Array with last two elements are the only pair of equal elements

4. Let C (n) denotes number of comparisons in worst case: Algorithm makes one comparison for each repetition of the innermost loop i.e., for each value of the loop's variable j between its limits $i + 1$ *and* $n - 1$; and this is repeated for each value of the outer loop i.e, for each value of the loop's variable i between its limits 0 *and* $n - 2$

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

5. **Simplify summation** using standard formulas

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

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

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

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

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

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

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

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

$$C(n) = (n-1)(n)/2$$
$$= (n^2 - n)/2$$
$$= (n^2)/2 - n/2$$
$$C(n) \in \Theta(n^2)$$

Mathematical analysis (Time Efficiency) of recursive Algorithms

General plan for analyzing efficiency of recursive algorithms:

1. Decide on parameter n indicating **input size**

2. Identify algorithm's **basic operation**

3. Check whether the number of times the basic operation is executed depends only on the input size n. If it also depends on the type of input, investigate **worst, average, and best case efficiency** separately.

4. Set up **recurrence relation**, with an appropriate initial condition, for the number of times the algorithm's basic operation is executed.

5. **Solve** the recurrence.

Example: Factorial function

ALGORITHM *Factorial (n)*

//Computes n! recursively

//Input: A nonnegative integer n

//Output: The value of n!

if *n == 0*

return 1

else

return Factorial $(n - 1) * n$

Analysis:

1. Input size: given number = n

2. Basic operation: multiplication

3. NO best, worst, average cases.

4. Let M (n) denotes number of multiplications.

M (n) = M $(n - 1)$ + 1   for $n > 0$

M (0) = 0   *initial condition*

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

1   :to multiply Factorial $(n - 1)$ by n

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

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

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

= M $(n - 2)$ + 2

= [M $(n - 3)$ + 1] + 3

= M $(n - 3)$ + 3

…

In the ith recursion, we have

= M $(n - i)$ + i

When i = n, we have

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

Since M (0) = 0

= n

M (n) = Θ (n)

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

ALGORITHM *BinRec (n)*

//Input: A positive decimal integer n

//Output: The number of binary digits in n"sbinary representation

if *n* = = 1

return 1

else

return *BinRec* ($\lfloor n/2 \rfloor$) + 1

Analysis:

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

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

A (1) = 0   *initial condition*

Where: A ($\lfloor n/2 \rfloor$) : to compute *BinRec* ($\lfloor n/2 \rfloor$)

1 : to increase the returned value by 1

5. Solve the recurrence:

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

Assume n = 2^k (smoothness rule)

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

Solving using "Backward substitution method":

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

= [A (2^{k-2}) + 1] + 1

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

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

= A (2^{k-3}) + 3

…

In the ith recursion, we have

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

When i = k, we have

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

Since $A(2^0) = 0$

$A(2^k) = k$

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

$A(n) = \log_2 n$

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

1.2 Brute Force Approaches:

Introduction

Brute force is a straightforward approach to problem solving, usually directly based on the problem's statement and definitions of the concepts involved. Though rarely a source of clever or efficient algorithms, the brute-force approach should not be overlooked as an important algorithm design strategy. Unlike some of the other strategies, **brute force** is applicable to a very wide variety of problems. For some important problems (e.g., sorting, searching, string matching), the brute-force approach yields reasonable algorithms of at least some practical value with no limitation on instance size Even if too inefficient in general, a brute-force algorithm can still be useful for solving small-size instances of a problem. A brute-force algorithm can serve an important theoretical or educational purpose.

1.3 Selection Sort and Bubble Sort

Problem: Given a list of *n* orderable items (e.g., numbers, characters from some alphabet, character strings), rearrange them in nondecreasing order.

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 $A[i]$ an

Example:

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

Selection sort's operation on the list 89, 45, 68, 90, 29, 34, 17. Each line corresponds to one iteration of the algorithm, i.e., a pass through the list's tail to the right of the vertical bar; an element in bold indicates the smallest element found. Elements to the left of the vertical bar are in their final positions and are not considered in this and subsequent iterations.

Performance Analysis of the selection sort algorithm:

The input's size is given by the number of elements n.

The algorithm's basic operation is the key comparison $A[j]<A[min]$. The number of times it is executed depends only on the array's size and is given by

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

Thus, selection sort is a $O(n^2)$ algorithm on all inputs. The number of key swaps is only $O(n)$ or, more precisely, n-1 (one for each repetition of the i loop).This property distinguishes selection sort positively from many other sorting algorithms.

Bubble Sort

Compare adjacent elements of the list and exchange them if they are out of order.Then we repeat the process,By doing it repeatedly, we end up _bubbling up' the largest element to the last position on the list

ALGORITHM

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

//The algorithm sorts array $A[0..n - 1]$ by bubble sort

//Input: An array $A[0..n - 1]$ of orderable elements

//Output: Array $A[0..n - 1]$ sorted in ascending order

for i=0 **to** n - 2 **do**

for j=0 **to** n - 2 - i **do**

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

    swap $A[j]$ and $A[j + 1]$

Example

etc.

The first 2 passes of bubble sort on the list 89, 45, 68, 90, 29, 34, 17. A new line is shown after a swap of two elements is done. The elements to the right of the vertical bar are in their final positions and are not considered in subsequent iterations of the algorithm

Bubble Sort the analysis

Clearly, the outer loop runs **n** times. The only complexity in this analysis in the inner loop. If we think about a single time the inner loop runs, we can get a simple bound by noting that it can never loop more than **n** times. Since the outer loop will make the inner loop complete *n* times, the comparison can't happen more than $O(n^2)$ times.

The number of key comparisons for the bubble sort version given above is the same for all arrays of size *n*.

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

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

The number of key swaps depends on the input. For the worst case of decreasing arrays, it is the same as the number of key comparisons.

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

Observation: if a pass through the list makes no exchanges, the list has been sorted and we can stop the algorithm Though the new version runs faster on some inputs, it is still in $O(n^2)$ in the worst and average cases. Bubble sort is not very good for big set of input. How ever bubble sort is very **simple to code**.

General Lesson From Brute Force Approach

A first application of the brute-force approach often results in an algorithm that can be improved with a modest amount of effort. Compares successive elements of a given list with a given search key until either a match is encountered (successful search) or the list is exhausted without finding a match (unsuccessful search)

1. 4 Sequential Search and Brute Force String Matching.

Sequential Search

ALGORITHM *SequentialSearch2(A* [0..n], *K)*

//The algorithm implements sequential search with a search key as a sentinel

//Input: An array *A* of *n* elements and a search key *K*

//Output: The position of the first element in *A*[0..*n* - 1] whose value is

// equal to *K* or -1 if no such element is found

A[*n*]=*K*

i=0

while *A*[*i*] = *K* **do**

i=*i* + 1

if *i* < *n* **return** *i*

else return

Brute-Force String Matching

Given a string of *n* characters called the *text* and a string of *m* characters (*m* = *n)* called the *pattern*, find a substring of the text that matches the pattern. To put it more precisely, we want to find *i*—the index of the leftmost character of the first matching

substring in the text—such that

$$t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}:$$

$$t_0 \ldots t_i \ldots t_{i+j} \ldots t_{i+m-1} \ldots t_{n-1} \qquad \textbf{text } \textbf{\textit{T}}$$

$$p_0 \ldots p_j \ldots p_{m-1} \qquad\qquad \textbf{pattern } \textbf{\textit{P}}$$

1. Pattern: 001011

Text: 10010101101001100101111010

2. Pattern: happy

Text: It is never too late to have a happy childho
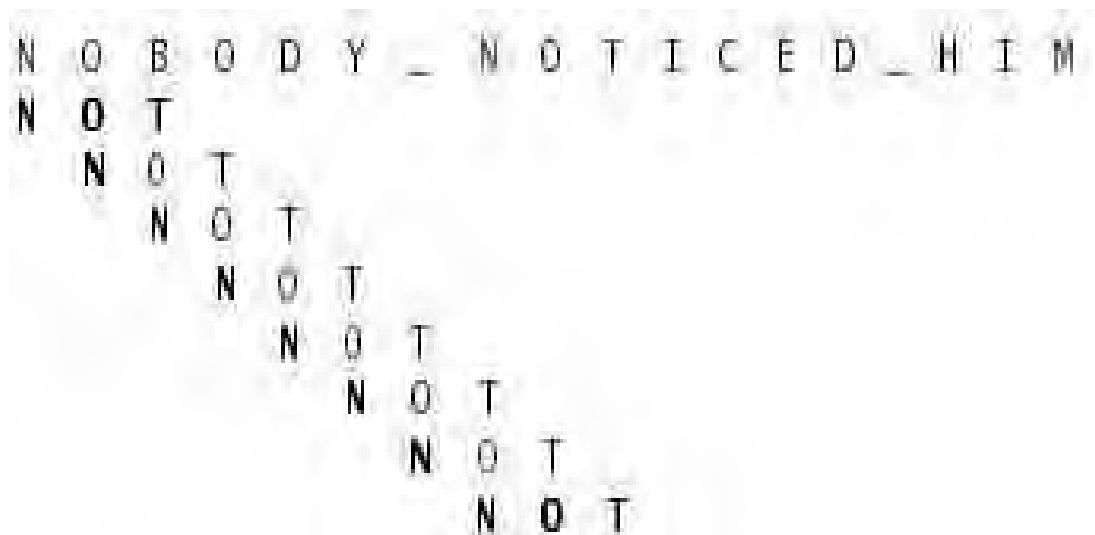
```
N O B O D Y _ N O T I C E D _ H I M
N O T
  N O T
    N O T
      N O T
        N O T
          N O T
            N O T
              N O T
```

The algorithm shifts the pattern almost always after a single character comparison. in the worst case, the algorithm may have to make all *m* comparisons before shifting the pattern, and this can happen for each of the *n - m* + 1 tries. Thus, in the worst case, the algorithm is in θ*(nm)*.

DIVIDE & CONQUER

1.1 Divide and Conquer

1.2 General Method

1.3 Binary Search

1.4 Merge Sort

1.5 Quick Sort and its performance

1.1 Divide and Conquer

Definition:
Divide & conquer is a general algorithm design strategy with a general plan as follows:
1. DIVIDE:
    A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size.
2. RECUR:
    Solve the sub-problem recursively.
3. CONQUER:
    If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance.

Diagram 1 shows the general divide & conquer plan
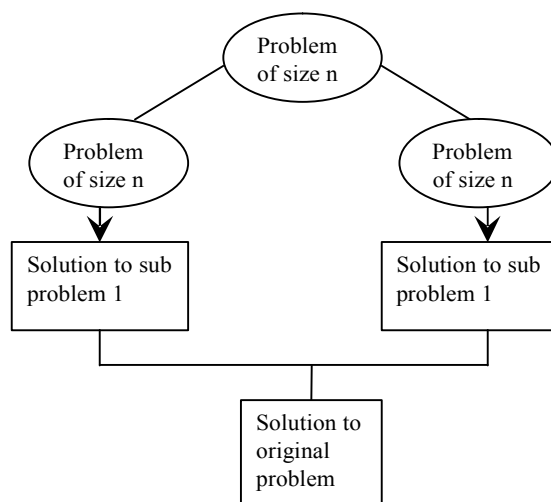
NOTE:

The base case for the recursion is sub-problem of constant size.

Advantages of Divide & Conquer technique:
- For solving conceptually difficult problems like Tower Of Hanoi, divide & conquer is a powerful tool
- Results in efficient algorithms
- Divide & Conquer algorithms are adapted foe execution in multi-processor machines
- Results in algorithms that use memory cache efficiently.

Limitations of divide & conquer technique:
- Recursion is slow
- Very simple problem may be more complicated than an iterative approach. Example: adding n numbers etc

1.2 General Method

General divide & conquer recurrence:
An instance of size n can be divided into b instances of size n/b, with ―a‖ of them needing to be solved. [$a \geq 1$, $b > 1$].
Assume size n is a power of b. The recurrence for the running time T(n) is as follows:

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

where:

$f(n)$ – a function that accounts for the time spent on dividing the problem into smaller ones and on combining their solutions

Therefore, the order of growth of T(n) depends on the values of the constants a & b and the order of growth of the function f(n).

Master theorem

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

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

Example:

Let $T(n) = 2T(n/2) + 1$, solve using master theorem.
Solution:
Here: $a = 2$
$b = 2$
$f(n) = \Theta(1)$

$$d = 0$$

Therefore:

$$a > b^d \quad \text{i.e., } 2 > 2^0$$

Case 3 of master theorem holds good. Therefore:

$$T(n) \in \Theta \ (n^{\log_b a})$$
$$\in \Theta \ (n^{\log_2 2})$$
$$\in \Theta \ (n)$$

1.3 Binary Search

Description:
Binary tree is a dichotomic divide and conquer search algorithm. Ti inspects the middle element of the sorted list. If equal to the sought value, then the position has been found. Otherwise, if the key is less than the middle element, do a binary search on the first half, else on the second half.

Algorithm:
Algorithm can be implemented as recursive or non-recursive algorithm.

ALGORITHM BinSrch (A[0 … n-1], key)
//implements non-recursive binary search
//i/p: Array A in ascending order, key k
//o/p: Returns position of the key matched else -1

l   0
r   n-1

while l ≤ r do
       m   (l + r) / 2
       if key = = A[m]
               return m
       else
               if key < A[m]
                       r   m-1
               else
                       l   m+1
return -1

Analysis:
- **Input size:** Array size, n
- **Basic operation:** key comparison
- **Depend on**
  Best – key matched with mid element
  Worst – key not found or key sometimes in the list
- Let $C(n)$ denotes the number of times basic operation is executed. Then
  $C_{worst}(n)$ = Worst case efficiency. Since after each comparison the algorithm divides the problem into half the size, we have
  $$C_{worst}(n) = C_{worst}(n/2) + 1 \qquad \text{for } n > 1$$

$$C(1) = 1$$

- Solving the recurrence equation using master theorem, to give the number of times the search key is compared with an element in the array, we have:

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

$$a = 1$$

$$b = 2$$

$$f(n) = n^0 \; ; \; d = 0$$

case 2 holds:

$$C(n) = \Theta \, (n^d \log n)$$
$$= \Theta \, (n^0 \log n)$$
$$= \Theta \, (\log n)$$

Applications of binary search:
- Number guessing game
- Word lists/search dictionary etc

Advantages:
- Efficient on very big list
- Can be implemented iteratively/recursively

Limitations:
- Interacts poorly with the memory hierarchy
- Requires given list to be sorted
- Due to random access of list element, needs arrays instead of linked list.

1.4 Merge Sort

Definition:

Merge sort is a sort algorithm that splits the items to be sorted into two groups, recursively sorts each group, and merges them into a final sorted sequence.

Features:
- Is a comparison based algorithm
- Is a stable algorithm
- Is a perfect example of divide & conquer algorithm design strategy
- It was invented by John Von Neumann

Algorithm:

```
ALGORITHM Mergesort ( A[0… n-1] )
//sorts array A by recursive mergesort
//i/p: array A
//o/p: sorted array A in ascending order

if n > 1
        copy A[0… (n/2 -1)] to B[0… (n/2 -1)]
        copy A[n/2… n -1)] to C[0… (n/2 -1)]
        Mergesort ( B[0… (n/2 -1)] )
        Mergesort ( C[0… (n/2 -1)] )
        Merge ( B, C, A )
```

ALGORITHM Merge (B[0… p-1], C[0… q-1], A[0… p+q-1])
//merges two sorted arrays into one sorted array
//i/p: arrays B, C, both sorted
//o/p: Sorted array A of elements from B & C

I →0
j→0
k→0
while i < p and j < q do
        if B[i] ≤ C[j]
                A[k] →B[i]
                i→i + 1
        else
                A[k] →C[j]
                j→j + 1
        k→k + 1
if i == p
        copy C [j… q-1] to A [k… (p+q-1)]
else
        copy B [i… p-1] to A [k… (p+q-1)]

Example:

Apply merge sort for the following list of elements: 6, 3, 7, 8, 2, 4, 5, 1
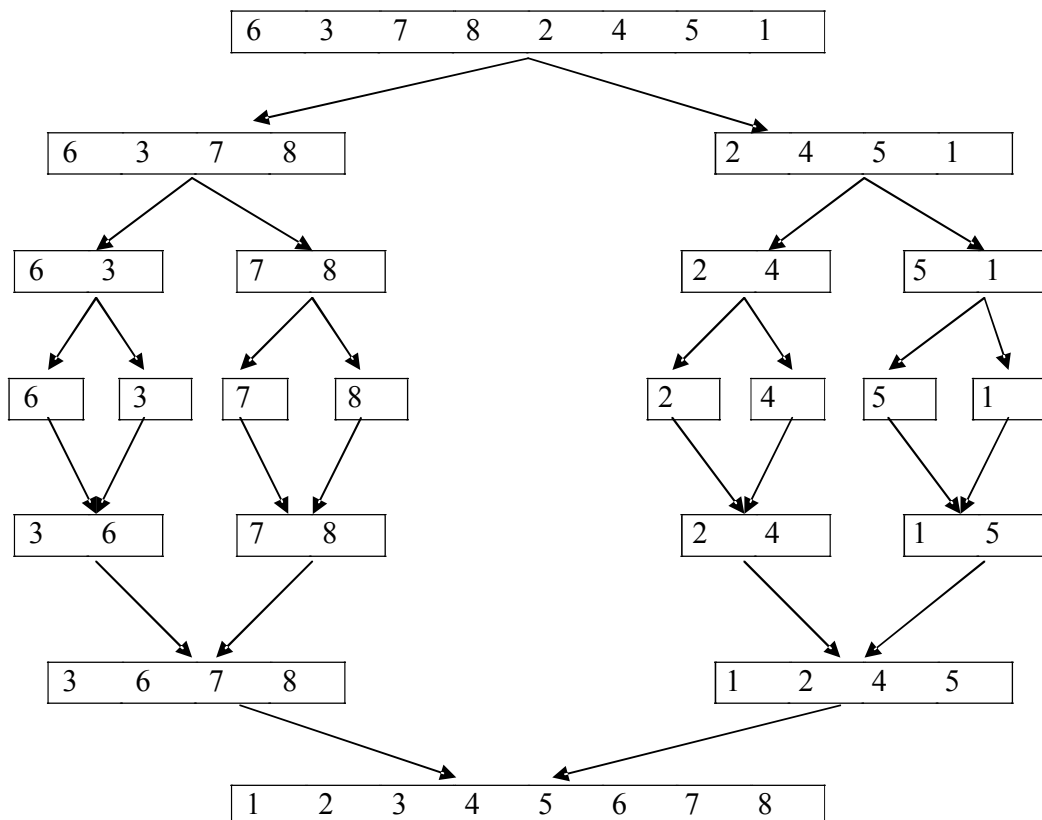
Analysis:

- **Input size:** Array size, n
- **Basic operation:** key comparison
- Best, worst, average case exists:
  Worst case: During key comparison, neither of the two arrays becomes empty before the other one contains just one element.
- Let $C(n)$ denotes the number of times basic operation is executed. Then
  $C(n) = 2C(n/2) + C_{merge}(n)$ for $n > 1$
  $C(1) = 0$
  where, $C_{merge}(n)$ is the number of key comparison made during the merging stage.
  In the worst case:
  $\qquad C_{merge}(n) = 2\ C_{merge}(n/2) + n\text{-}1$ for $n > 1$
  $\qquad C_{merge}(1) = 0$
- Solving the recurrence equation using master theorem:
  $C(n) = 2C(n/2) + n\text{-}1$ for $n > 1$
  $C(1) = 0$

  Here $a = 2$
  $\qquad b = 2$
  $\qquad f(n) = n; d = 1$
  Therefore $2 = 2^1$, case 2 holds
  $\qquad C(n) = \quad \Theta\ (n^d \log n)$
  $\qquad\qquad = \quad \Theta\ (n^1 \log n)$
  $\qquad\qquad = \quad \Theta\ (n \log n)$

Advantages:

- Number of comparisons performed is nearly optimal.
- Mergesort will never degrade to $O(n^2)$
- It can be applied to files of any size

Limitations:

- Uses $O(n)$ additional memory.

1.6 Quick Sort and its performance

Definition:

Quick sort is a well –known sorting algorithm, based on divide & conquer approach. The steps are:

1. Pick an element called pivot from the list
2. Reorder the list so that all elements which are less than the pivot come before the pivot and all elements greater than pivot come after it. After this partitioning, the pivot is in its final position. This is called the partition operation
3. Recursively sort the sub-list of lesser elements and sub-list of greater elements.

Features:

- Developed by C.A.R. Hoare
- Efficient algorithm
- NOT stable sort
- Significantly faster in practice, than other algorithms

ALGORITHM Quicksort (A[l …r])

//sorts by quick sort
//i/p: A sub-array A[l..r] of A[0..n-1],defined by its left and right indices l and r
//o/p: The sub-array A[l..r], sorted in ascending order
if l < r

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

ALGORITHM Partition (A[l ..r])

//Partitions a sub-array by using its first element as a pivot
//i/p: A sub-array A[l..r] of A[0..n-1], defined by its left and right indices l and r (l < r)
//o/p: A partition of A[l..r], with the split position returned as this function's value
p→A[l]
i→l
j→r + 1;
Repeat

       repeat i→i + 1 until A[i] >=p   //left-right scan
       repeat j→j – 1 until A[j] < p   //right-left scan
       if (i < j)   //need to continue with the scan

           swap(A[i], a[j])
until i >= j   //no need to scan
swap(A[l], A[j])
return j

Example: Sort by quick sort the following list: 5, 3, 1, 9, 8, 2, 4, 7, show recursion tree.

Analysis:

- **Input size:** Array size, n
- **Basic operation:** key comparison
- Best, worst, average case exists:
  Best case: when partition happens in the middle of the array each time.
  Worst case: When input is already sorted. During key comparison, one half is empty, while remaining n-1 elements are on the other partition.
- Let C(n) denotes the number of times basic operation is executed in worst case:
  Then
  $C(n) = C(n-1) + (n+1)$ for n > 1 (2 sub-problems of size 0 and n-1 respectively)
  $C(1) = 1$

  Best case:
  $C(n) = 2C(n/2) + \Theta(n)$ (2 sub-problems of size n/2 each)

- Solving the recurrence equation using backward substitution/ master theorem,
  we have:
  $C(n) = C(n-1) + (n+1)$ for n > 1; $C(1) = 1$
  $$C(n) = \Theta(n^2)$$

  $C(n) = 2C(n/2) + \Theta(n).$
  $$= \Theta(n^1 \log n)$$
  $$= \Theta(n \log n)$$

NOTE:
The quick sort efficiency in average case is $\Theta(n \log n)$ on random input.

UNIT - 3
THE GREEDY METHOD

3.1 The General Method
3.2 Knapsack Problem
3.3 Job Sequencing with Deadlines
3.4 Minimum-Cost Spanning Trees
3.5 Prim'sAlgorithm
3.6 Kruskal's Algorithm
3.7 Single Source Shortest Paths.

3.1The General Method

Definition:

Greedy technique is a general algorithm design strategy, built on following elements:
- **configurations**: different choices, values to find
- **objective function**: some configurations to be either maximized or minimized

The method:

- Applicable to **optimization problems** ONLY
- Constructs a solution through a sequence of steps
- Each step expands a partially constructed solution so far, until a complete solution to the problem is reached.
  On each step, the choice made must be
- **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

NOTE:
- Greedy method works best when applied to problems with the **greedy-choice** property
- A globally-optimal solution can always be found by a series of local improvements from a starting configuration.

Greedy method vs. Dynamic programming method:

- **LIKE** dynamic programming, greedy method **solves optimization problems.**
- **LIKE** dynamic programming, greedy method **problems exhibit optimal substructure**
- **UNLIKE** dynamic programming, greedy method problems exhibit the **greedy choice** property -avoids back-tracing.

Applications of the Greedy Strategy:

- **Optimal solutions:**
  - Change making
  - Minimum Spanning Tree (MST)
  - Single-source shortest paths
  - Huffman codes
- **Approximations:**
  - Traveling Salesman Problem (TSP)
  - Fractional Knapsack problem

3.2 Knapsack problem

- One wants to pack n items in a luggage
  - The ith item is worth v_i dollars and weighs w_i pounds
  - Maximize the value but cannot exceed W pounds
  - v_i, w_i, W are integers
- 0-1 knapsack → each item is taken or not taken
- Fractional knapsack → fractions of items can be taken
- Both exhibit the optimal-substructure property
  - 0-1: If item j is removed from an optimal packing, the remaining packing is an optimal packing with weight at most W-w_j
  - Fractional: If w pounds of item j is removed from an optimal packing, the remaining packing is an optimal packing with weight at most W-w that can be taken from other n-1 items plus w_j – w of item j

Greedy Algorithm for Fractional Knapsack problem

- Fractional knapsack can be solvable by the greedy strategy
  - Compute the value per pound v_i/w_i for each item
  - Obeying a greedy strategy, take as much as possible of the item with the greatest value per pound.
  - If the supply of that item is exhausted and there is still more room, take as much as possible of the item with the next value per pound, and so forth until there is no more room
  - O(n lg n) (we need to sort the items by value per pound)

O-1 knapsack is harder

- knapsack cannot be solved by the greedy strategy

- Unable to fill the knapsack to capacity, and the empty space lowers the effective value per pound of the packing
- We must compare the solution to the sub-problem in which the item is included with the solution to the sub-problem in which the item is excluded before we can make the choice
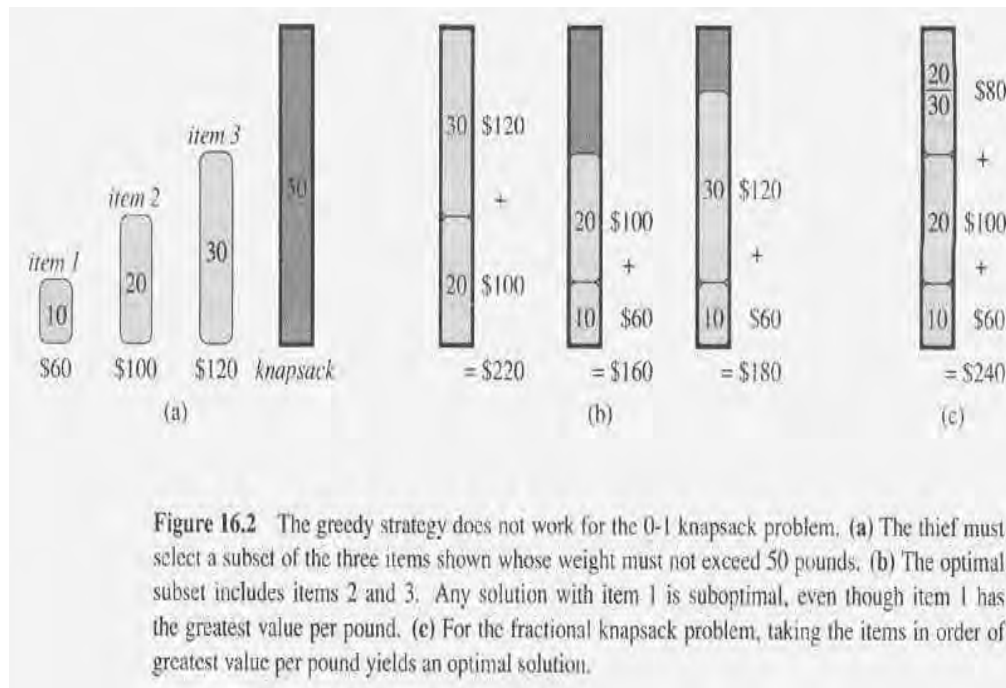
Figure 16.2 The greedy strategy does not work for the 0-1 knapsack problem. (a) The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. (b) The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. (c) For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

3.3 Job sequencing with deadlines

The problem is stated as below.

- There are n jobs to be processed on a machine.
- Each job i has a deadline $d_i \geq 0$ and profit $p_i \geq 0$.
- Pi is earned iff the job is completed by its deadline.
- The job is completed if it is processed on a machine for unit time.
- Only one machine is available for processing jobs.
- Only one job is processed at a time on the machine
- A feasible solution is a subset of jobs J such that each job is completed by its deadline.
- An optimal solution is a feasible solution with maximum profit value.

    Example : Let n = 4, (p_1, p_2, p_3, p_4) = (100,10,15,27), (d_1, d_2, d_3, d_4) = (2,1,2,1)

Sr.No.	Feasible Solution	Processing Sequence	Profit value
(i)	(1,2)	(2,1)	110
(ii)	(1,3)	(1,3) or (3,1)	115
(iii)	(1,4)	(4,1)	127 ,is the optimal one
(iv)	(2,3)	(2,3)	25
(v)	(3,4)	(4,3)	42
(vi)	(1)	(1)	100
(vii)	(2)	(2)	10
(viii)	(3)	(3)	15
(ix)	(4)	(4)	27

- Consider the jobs in the non increasing order of profits subject to the constraint that the resulting job sequence J is a feasible solution.
- In the example considered before, the non-increasing profit vector is

$$(100 \quad 27 \quad 15 \quad 10) \qquad (2 \quad 1 \quad 2 \quad 1)$$

$$p_1 \ p_4 \ p_3 \ p_2 \qquad\qquad d_1 \ d \ d_3 \ d_2$$

J = { 1} is a feasible one

J = { 1, 4} is a feasible one with processing   sequence (4,1)

J = { 1, 3, 4} is not feasible

J = { 1, 2, 4} is not feasible

J = { 1, 4} is optimal

Theorem: Let J be a set of K jobs and

$\Sigma = (i_1,i_2,\ldots.i_k)$ be a permutation of jobs in J such that $di_1 \leq di_2 \leq\ldots\leq di_k$.

- J is a feasible solution iff the jobs in J can be processed in the order Σ without violating any deadly.

Proof:

- By definition of the feasible solution if the jobs in J can be processed in the order without violating any deadline then J is a feasible solution.

- So, we have only to prove that if J is a feasible one, then Σ represents a possible order in which the jobs may be processed.

- Suppose J is a feasible solution. Then there exists $\Sigma^1 = (r_1, r_2, \ldots, r_k)$ such that

$$d_{rj} \geq j, \qquad 1 \leq j < k$$

i.e. $d_{r1} \geq 1, d_{r2} \geq 2, \ldots, d_{rk} \geq k.$

each job requiring an unit time.

- $\Sigma = (i_1, i_2, \ldots, i_k)$ and $\Sigma^1 = (r_1, r_2, \ldots, r_k)$

- Assume $\Sigma^1 \neq \Sigma$. Then let a be the least index in which Σ^1 and Σ differ. i.e. a is such that $r_a \neq i_a$.

- Let $r_b = i_a$, so $b > a$ (because for all indices j less than a $r_j = i_j$).

- In Σ^1 interchange r_a and r_b.

- $\Sigma = (i_1, i_2, \ldots i_a \quad i_b \quad i_k)$ 　　　　[r_b occurs before r_a

- 　　　　　　　　　　　　　　　　　　　　　　in $i_1, i_2, \ldots, i_k$]

- $\Sigma^1 = (r_1, r_2, \ldots r_a \quad r_b \ldots r_k)$

- 　　　　$i_1 = r_1, i_2 = r_2, \ldots i_{a-1} = r_{a-1}, i_a \neq r_b$ but $i_a = r_b$

- We know $d_{i_1} \leq d_{i_2} \leq \ldots d_{i_a} \leq d_{i_b} \leq \ldots \leq d_{i_k}$.

- Since $i_a = r_b$, $d_{r_b} \leq d_{r_a}$ or $d_{r_a} \geq d_{r_b}$.

- In the feasible solution $d_{r_a} \geq a \quad d_{r_b} \geq b$

- So if we interchange r_a and r_b, the resulting permutation $\Sigma^{11} = (s_1, \ldots s_k)$ represents an order with the least index in which Σ^{11} and Σ differ is incremented by one.

- Also the jobs in Σ^{11} may be processed without violating a deadline.

- Continuing in this way, Σ^1 can be transformed into Σ without violating any deadline.

- Hence the theorem is proved

GREEDY ALGORITHM FOR JOB SEQUENSING WITH DEADLINE

Procedure greedy job (D, J, n)　　　　　　　　J may be represented by

// J is the set of n jobs to be completed　　//　　one dimensional array J (1: K)

// by their deadlines //　　　　　　　　　　The deadlines are

　J $\leftarrow$ {1}　　　　　　　　　　D (J(1)) $\leq$ D(J(2)) $\leq$ .. $\leq$ D(J(K))

　for I $\leftarrow$ 2 to n do　　　　　　　　To test if JU {i} is feasible,

If all jobs in JU{i} can be completed   we insert i into J and verify

 by their deadlines   D(J®) ≤ r   1 ≤ r ≤ k+1

  then J ← JU{I}

end if

   repeat

 end greedy-job

Procedure JS(D,J,n,k)

// D(i) ≥ 1, 1≤ i ≤ n are the deadlines //

// the jobs are ordered such that //

// $p_1 \geq p_2 \geq$ ……. $\geq p_n$ //

// in the optimal solution ,D(J(i) ≥ D(J(i+1)) //

                                    // 1 ≤ i ≤ k //

integer D(o:n), J(o:n), i, k, n, r

D(0) ←J(0) ← 0

// J(0) is a fictious job with D(0) = 0 //

K←1; J(1) ←1   // job one is inserted into J //

for i ←2 to do // consider jobs in non increasing order of pi //

// find the position of i and check feasibility of insertion //

   r← k   // r and k are indices for existing job in J //

// find r such that i can be inserted after r //

while D(J(r)) > D(i) and D(i) ≠ r do

// job r can be processed after i and //

// deadline of job r is not exactly r //

   r← r-1 // consider whether job r-1 can be processed after i //

repeat

if D(J(r)) ≥ d(i) and D(i) > r then

// the new job i can come after existing job r; insert i into J at position r+1 //

for I ← k to r+1 by –1 do

J(I+1)← J(l) // shift jobs(r+1) to k right by//

//one position //

repeat

if D(J(r)) ≥ d(i) and D(i) > r then

// the new job i can come after existing job r; insert i into J at position r+1 //

for I ← k to r+1 by –1 do

J(I+1)← J(l) // shift jobs(r+1) to k right by//

//one position //

Repeat

COMPLEXITY ANALYSIS OF JS ALGORITHM

- Let n be the number of jobs and s be the number of jobs included in the solution.
- The loop between lines 4-15 (the for-loop) is iterated (n-1)times.
- Each iteration takes O(k) where k is the number of existing jobs.
- ∴   The time needed by the algorithm is 0(sn) s ≤ n so the worst case time is $0(n^2)$.

If d_i = n - i+1   1 ≤ i ≤ n, JS takes $\theta(n^2)$ time

D and J need θ(s) amount of space.

3.4 Minimum-Cost Spanning Trees
Spanning Tree
Spanning tree is a connected acyclic sub-graph (tree) of the given graph (*G)* that includes all of *G*'s vertices

Example: Consider the following graph
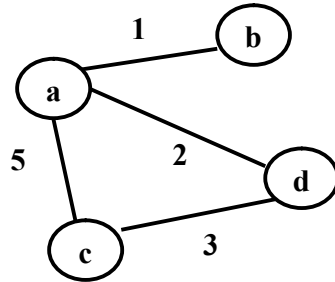
The spanning trees for the above graph are as follows:
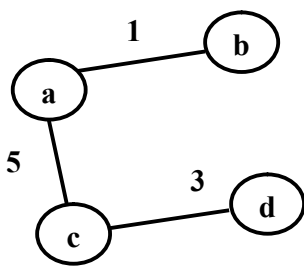
Weight (T₁) = 9
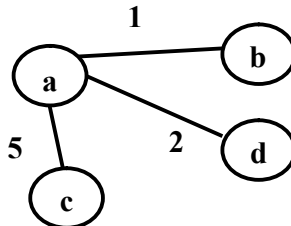
Weight (T₂) = 8
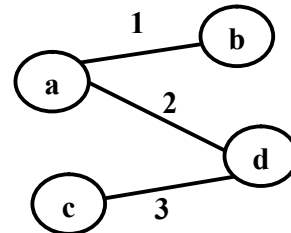
Weight (T₃) = 6

Minimum Spanning Tree (MST)

Definition:

MST of a weighted, connected graph *G* is defined as: A spanning tree of *G* with **minimum total weight.**

Example: Consider the example of spanning tree:

For the given graph there are three possible spanning trees. Among them the spanning tree with the minimum weight 6 is the MST for the given graph

Question: Why can't we use **BRUTE FORCE** method in constructing MST ?

Answer: **If we use Brute force method-**
- Exhaustive search approach has to be applied.
- Two serious obstacles faced:
    1. **The number of spanning trees grows exponentially with graph size.**
    2. **Generating all spanning trees for the given graph is not easy.**

MST Applications:

- **Network design.**
  Telephone, electrical, hydraulic, TV cable, computer, road
- **Approximation algorithms for NP-hard problems.**
  Traveling salesperson problem, Steiner tree
- Cluster analysis.
- Reducing data storage in sequencing amino acids in a protein
- Learning salient features for real-time face verification
- Auto config protocol for Ethernet bridging to avoid cycles in a network, etc

3.5 Prim's Algorithm

Some useful definitions:

- **Fringe edge:** An edge which has one vertex is in partially constructed tree Ti and the other is not.

- **Unseen edge:** An edge with both vertices not in Ti

Algorithm:

ALGORITHM Prim (G)

//Prim's algorithm for constructing a MST

//Input: A weighted connected graph G = { V, E }

//Output: ET the set of edges composing a MST of G

// the set of tree vertices can be initialized with any vertex

$V_T \rightarrow$ { v_0 }

$E_T \rightarrow$ Ø

for i$\rightarrow$ 1 to |V| - 1 do

   Find a minimum-weight edge e* = (v*, u*) among all the edges (v, u) such that v is in V_T and u is in V - V_T

   $V_T \rightarrow$ V_T U { u* }

$E_T \rightarrow \quad E_T \cup \{ e* \}$

return E_T

STEP 1: Start with a tree, T_0, consisting of one vertex
STEP 2: —Grow" tree one vertex/edge at a time
- Construct a series of expanding sub-trees $T_1, T_2, \ldots T_{n-1}$.
- At each stage construct T_{i+1} from T_i by adding the minimum weight edge connecting a vertex in tree (Ti) to one vertex not yet in tree, **choose from "fringe" edges (this is the "greedy" step!)**

Algorithm stops when all vertices are included
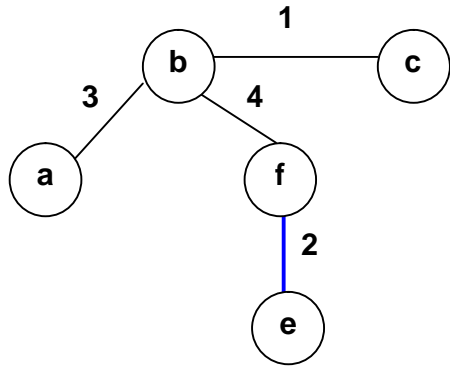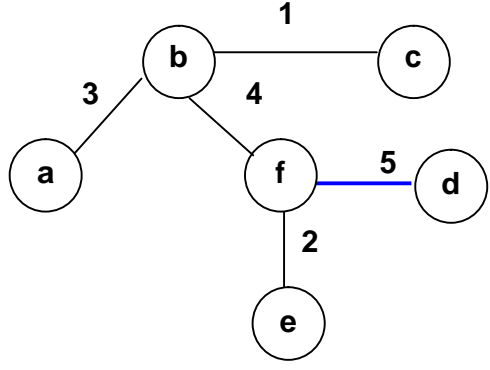
Example:

Apply Prim's algorithm for the following graph to find MST.
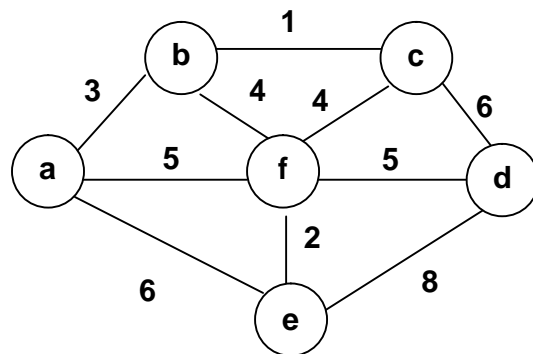
Solution:

Tree vertices	Remaining vertices	Graph
a (-, -)	b (a , 3) c (- , ∞) d (- , ∞) e (a , 6) f (a , 5)	
b (a, 3)	c (b , 1) d (- , ∞) e (a , 6) f (b , 4)	
c (b, 1)	d (c , 6) e (a , 6) f (b , 4)	

f (b, 4)	d (f , 5) e (f , 2)	
e (f, 2)	d (f , 5)	
d(f, 5)		**Algorithm stops since all vertices are included.** The weight of the minimum spanning tree is **15**

Efficiency:

Efficiency of Prim's algorithm is based on data structure used to store priority queue.

- Unordered array: **Efficiency: $\Theta(n^2)$**
- Binary heap: **Efficiency: $\Theta(m \log n)$**
- Min-heap: For graph with n nodes and m edges: **Efficiency: $(n + m) \log n$**

Conclusion:

- Prim's algorithm is a ―vertex based algorithm"
- Prim's algorithm ―Needs priority queue for locating the nearest vertex." The choice of priority queue matters in Prim implementation.
  - Array - optimal for dense graphs
  - Binary heap - better for sparse graphs
  - Fibonacci heap - best in theory, but not in practice

3.6 Kruskal's Algorithm

Algorithm:

ALGORITHM Kruskal (G)

//Kruskal's algorithm for constructing a MST
//Input: A weighted connected graph G = { V, E }
//Output: ET the set of edges composing a MST of G

Sort E in ascending order of the edge weights

// initialize the set of tree edges and its size
$E_T \rightarrow \emptyset$
edge_counter $\rightarrow 0$

//initialize the number of processed edges
$K \rightarrow 0$

while edge_counter < |V| - 1
      $k \rightarrow k + 1$
      if E_T U { $e_{i\ k}$} is acyclic
            $E_T \rightarrow E_T$ U { $e_{i\ k}$ }
            edge_counter $\rightarrow$ edge_counter + 1
return E_T

The method:

STEP 1: Sort the edges by increasing weight
STEP 2: Start with a forest having |V| number of trees.
STEP 3: Number of trees are reduced by ONE at every inclusion of an edge
      At each stage:
- Among the edges which are not yet included, select the one with minimum weight AND which does not form a cycle.
- the edge will reduce the number of trees by one by combining two trees of the forest

Algorithm stops when |V| -1 edges are included in the MST i.e : when the number of trees in the forest is reduced to ONE.

Example:

Apply Kruskal's algorithm for the following graph to find MST.
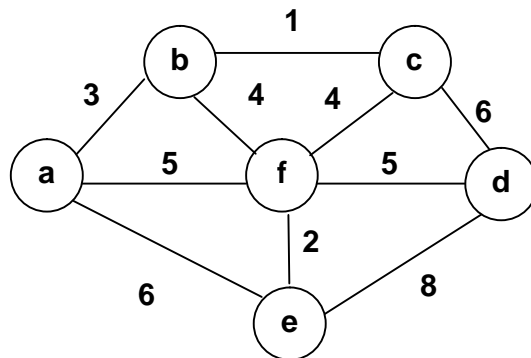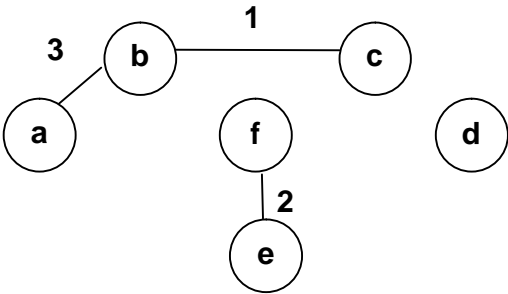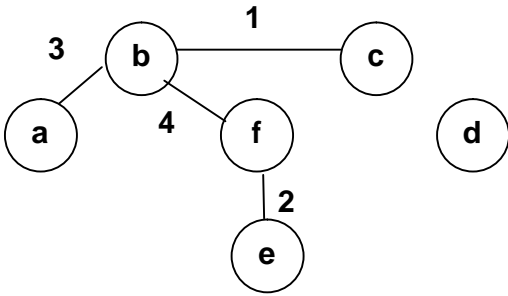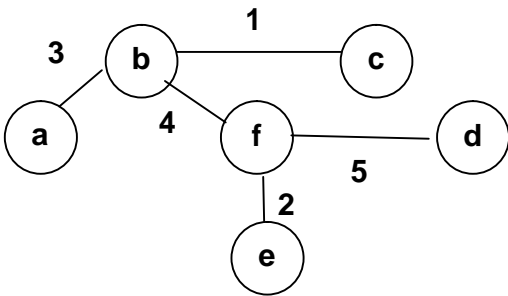
Solution:

The list of edges is:

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

Sort the edges in ascending order:

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

Edge	bc	
Weight	1	
Insertion status	YES	
Insertion order	1	
Edge	ef	
Weight	2	
Insertion status	YES	
Insertion order	2	

Edge	ab	
Weight	3	
Insertion status	YES	
Insertion order	3	
Edge	bf	
Weight	4	
Insertion status	YES	
Insertion order	4	
Edge	cf	
Weight	4	
Insertion status	NO	
Insertion order	-	
Edge	af	
Weight	5	
Insertion status	NO	
Insertion order	-	
Edge	df	
Weight	5	
Insertion status	YES	
Insertion order	5	
Algorithm stops as |V| -1 edges are included in the MST		

Efficiency:

Efficiency of Kruskal's algorithm is based on the time needed for sorting the edge weights of a given graph.

- With an efficient sorting algorithm: **Efficiency: $\Theta(|E| \log |E|)$**

Conclusion:

- Kruskal's algorithm is an —edge based algorithm"
- Prim's algorithm with a heap is faster than Kruskal's algorithm.

3.7 Single Source Shortest Paths.

Some useful definitions:

- **Shortest Path Problem:** Given a connected directed graph G with non-negative weights on the edges and a root vertex r, find for each vertex x, a directed path P (x) from r to x so that the sum of the weights on the edges in the path P (x) is as small as possible.

Algorithm

- By Dutch computer scientist Edsger Dijkstra in 1959.
- Solves the single-source shortest path problem for a graph with nonnegative edge weights.
- This algorithm is often used in routing.

   E.g.: Dijkstra's algorithm is usually the working principle behind link-state routing protocols

ALGORITHM Dijkstra(G, s)

//Input: Weighted connected graph G and source vertex s

//Output: The length Dv of a shortest path from s to v and its penultimate vertex Pv for every vertex v in V

//initialize vertex priority in the priority queue

Initialize (Q)

for every vertex v in V **do**

   $D_v \rightarrow \infty$; $P_v \rightarrow$ **null**   // Pv , the parent of v

   insert(Q, v, D_v) //initialize vertex priority in priority queue

$d_s \rightarrow 0$

//update priority of s with ds, making ds, the minimum

Decrease(Q, s, d_s)

$V_T \rightarrow \mathbf{O}$

for i→0 to |V| - 1 do

      u*→DeleteMin(Q)

      //expanding the tree, choosing the locally best vertex

      V_T→V_T U {u*}

      for every vertex u in V − V_T that is adjacent to u* do

          if Du* + w (u*, u) < Du

              Du→Du + w (u*, u); Pu   u*

              Decrease(Q, u, Du)

The method

Dijkstra's algorithm solves the single source shortest path problem in 2 stages.

Stage 1: A greedy algorithm computes the shortest distance from **source to all other nodes** in the graph and saves in a data structure.

Stage 2 : Uses the data structure for finding a **shortest path from source to any vertex v.**

- **At each step, and for each vertex x, keep track of a "distance" D(x) and a directed path P(x) from root to vertex x of length D(x).**
- **Scan first from the root and take initial paths P(r, x) = (r, x) with**

      D(x) = w(rx) when rx is an edge,

      D(x) = ∞ when rx is not an edge.

      For each temporary vertex y distinct from x, set

      D(y) = min{ D(y), D(x) + w(xy) }

Example:

Apply Dijkstra's algorithm to find Single source shortest paths with vertex **a** as the source.

Solution:

Length Dv of shortest path from source (s) to other vertices v and Penultimate vertex Pv for every vertex v in V:

    Da = 0 , Pa = null

    Db = ∞ , Pb = null

    Dc = ∞ , Pc = null

    Dd = ∞ , Pd = null

    De = ∞ , Pe = null

    Df = ∞ , Pf = null

Tree vertices	Remaining vertices	Distance & Path vertex	Graph
a (-, 0)	b (a , 3) c (- , ∞) d (- , ∞) e (a , 6) f (a , 5)	Da = 0 Pa = a **Db = 3 Pb = [a, b]** Dc = ∞ Pc = null Dd = ∞ Pd = null De = 6 Pe = [a, e] Df = 5 Pf = [a, f]	
b (a, 3)	c (b , 3+1) d (- , ∞) e (a , 6) f (a , 5)	Da = 0 Pa = a Db = 3 Pb = [a, b] **Dc = 4 Pc = [a,b,c]** Dd = ∞ Pd = null De = 6 Pe = [a, e] Df = 5 Pf = [a, f]	
c (b, 4)	d (c , 4+6) e (a , 6) f (a , 5)	Da = 0 Pa = a Db = 3 Pb = [a, b] Dc = 4 Pc = [a,b,c] Dd = 10 Pd = [a,b,c,d] De = 6 Pe = [a, e] **Df = 5 Pf = [a, f]**	
f (a, 5)	d (c , 10) e (a , 6)	Da = 0 Pa = a Db = 3 Pb = [a, b] Dc = 4 Pc = [a,b,c] Dd = 10 Pd = [a,b,c,d] **De = 6 Pe = [a, e]** Df = 5 Pf = [a, f]	
e (a, 6)	d (c, 10)	Da = 0 Pa = a Db = 3 Pb = [a, b] Dc = 4 Pc = [a,b,c] Dd = 10 Pd = [a,b,c,d] **De = 6 Pe = [a, e]** Df = 5 Pf = [a, f]	
d(c, 10)			**Algorithm stops since no edges to scan**

Conclusion:

- Doesn't work with negative weights
- Applicable to both undirected and directed graphs
- Use unordered array to store the priority queue: **Efficiency = $\Theta(n^2)$**
- Use min-heap to store the priority queue: **Efficiency = O(m log n)**

Dynamic Programming

4.1 The General Method
4.2 Warshall's Algorithm
4.3 Floyd's Algorithm for the All-Pairs Shortest Paths Problem
4.4 Single-Source Shortest Paths
4.5 General Weights 0/1 Knapsack
4.6 The Traveling Salesperson problem.

4.1 The General Method

Definition
Dynamic programming (DP) is a general algorithm design technique for solving problems with overlapping sub-problems. This technique was invented by American mathematician ―Richard Bellman" in 1950s.

Key Idea
The key idea is to save answers of overlapping smaller sub-problems to avoid re-computation.

Dynamic Programming Properties
- An instance is solved using the solutions for smaller instances.
- The solutions for a smaller instance might be needed multiple times, so store their results in a table.
- Thus each smaller instance is solved only once.
- Additional space is used to save time.

Dynamic Programming vs. Divide & Conquer
LIKE divide & conquer, dynamic programming solves problems by combining solutions to sub-problems. UNLIKE divide & conquer, sub-problems are NOT independent in dynamic programming.

Divide & Conquer	Dynamic Programming
1. Partitions a problem into independent smaller sub-problems	1. Partitions a problem into overlapping sub-problems
2. Doesn't store solutions of sub-problems. (Identical sub-problems may arise - results in the same computations are performed repeatedly.)	2. Stores solutions of sub-problems: thus avoids calculations of same quantity twice
3. Top down algorithms: which logically progresses from the initial instance down to the smallest sub-instances via intermediate sub-instances.	3. Bottom up algorithms: in which the smallest sub-problems are explicitly solved first and the results of these used to construct solutions to progressively larger sub-instances

Dynamic Programming vs. Divide & Conquer: EXAMPLE
Computing Fibonacci Numbers

1. Using standard recursive formula:

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

Algorithm F(n)
// Computes the nth Fibonacci number recursively by using its definitions
// Input: A non-negative integer n
// Output: The nth Fibonacci number
if n==0 || n==1 then
      return n
else
      return F(n-1) + F(n-2)

Algorithm F(n): Analysis
- Is too expensive as it has repeated calculation of smaller Fibonacci numbers.
- Exponential order of growth.

```
                        F(n)
                       /    \
            F(n-1)    +    F(n-2)
           /      \          /      \
      F(n-2)  +  F(n-3)  F(n-3)  +  F(n-4)
```
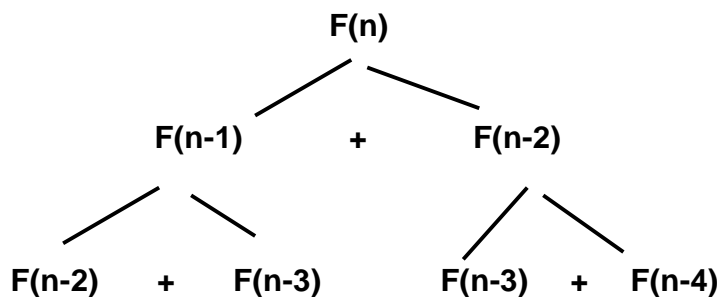
2. Using Dynamic Programming:
Algorithm F(n)
// Computes the nth Fibonacci number by using dynamic programming method
// Input: A non-negative integer n
// Output: The nth Fibonacci number
A[0]   0
A[1]   1
for i   2 to n do
      A[i]   A[i-1] + A[i-2]
return A[n]

Algorithm F(n): Analysis
- Since it caches previously computed values, saves time from repeated computations of same sub-instance
- Linear order of growth

Rules of Dynamic Programming

1. **OPTIMAL SUB-STRUCTURE:** An optimal solution to a problem contains optimal solutions to sub-problems
2. **OVERLAPPING SUB-PROBLEMS:** A recursive solution contains a ―small‖ number of distinct sub-problems repeated many times
3. **BOTTOM UP FASHION:** Computes the solution in a bottom-up fashion in the final step

Three basic components of Dynamic Programming solution

The development of a dynamic programming algorithm must have the following three basic components

1. A recurrence relation
2. A tabular computation
3. A backtracking procedure

Example Problems that can be solved using Dynamic Programming method

1. Computing binomial co-efficient
2. Compute the longest common subsequence
3. Warshall‗s algorithm for transitive closure
4. Floyd‗s algorithm for all-pairs shortest paths
5. Some instances of difficult discrete optimization problems like knapsack problem And traveling salesperson problem

4.2 Warshall's Algorithm

Some useful definitions:

- **Directed Graph:** A graph whose every edge is directed is called directed graph OR digraph
- **Adjacency matrix:** The adjacency matrix $A = \{a_{ij}\}$ of a directed graph is the boolean matrix that has
    - o 1 - if there is a directed edge from ith vertex to the jth vertex
    - o 0 - Otherwise
- **Transitive Closure:** Transitive closure of a directed graph with n vertices can be defined as the n-by-n matrix T={tij}, in which the elements in the ith row (1≤ i ≤ n) and the jth column(1≤ j ≤ n) is 1 if there exists a nontrivial directed path (i.e., a directed path of a positive length) from the ith vertex to the jth vertex, otherwise tij is 0. The transitive closure provides reach ability information about a digraph.

Computing Transitive Closure:

- We can perform DFS/BFS starting at each vertex
    - Performs traversal starting at the ith vertex.
    - Gives information about the vertices reachable from the ith vertex
    - **Drawback:** This method traverses the same graph several times.
    - **Efficiency** : (O(n(n+m)))
- Alternatively, we can use dynamic programming: the Warshall‗s Algorithm

Underlying idea of Warshall's algorithm:

- Let A denote the initial boolean matrix.
- The element r(k) [i, j] in ith row and jth column of matrix Rk (k = 0, 1, …, n) is equal to 1 if and only if there exists a directed path from ith vertex to jth vertex with intermediate vertex if any, numbered not higher than k
- **Recursive Definition:**
    - **Case 1:** A path from *vi* to *vj* restricted to using only vertices from {*v1,v2,…,vk*} as intermediate vertices does not use *vk*, Then

      R(*k*) [*i, j*] = R(*k*-1) [*i, j*].
    - **Case 2:** A path from *vi* to *vj* restricted to using only vertices from {*v1,v2,…,vk*} as intermediate vertices do use vk. Then

      R(*k*) [*i, j*] = R(*k*-1) [*i, k*] **AND** R(*k*-1) [*k, j*].

   R(*k*)[*i, j*] = R(*k*-1) [*i, j*] OR (R(*k*-1) [*i, k*] AND R(*k*-1) [*k, j*])

Algorithm:

Algorithm Warshall(A[1..n, 1..n])
// Computes transitive closure matrix
// Input: Adjacency matrix A
// Output: Transitive closure matrix R
R(0)   A
for k→1 to n do
    for i→   1 to n do
        for j→   1 to n do
            R(k)[i, j]→   R(k-1)[i, j] OR (R(k-1)[i, k] AND R(k-1)[k, j])
return R(n)

Find Transitive closure for the given digraph using Warshall's algorithm.
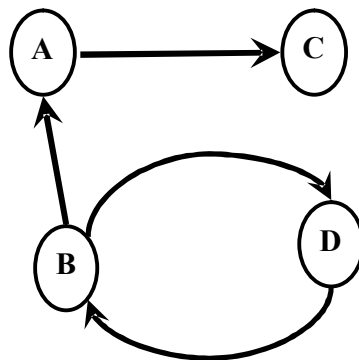
Solution:

		A	B	C	D
R(0) =	A	0	0	1	0
	B	1	0	0	1
	C	0	0	0	0
	D	0	1	0	0

R(0)	k = 1 Vertex 1 can be intermediate node		A	B	C	D			A	B	C	D
		A	0	0	1	0	A	0	0	1	0
		B	1	0	0	1	B	1	0	**1**	1
		C	0	0	0	0	C	0	0	0	0
		D	0	1	0	0	D	0	1	0	0

R1[2,3]
= R0[2,3] OR
 R0[2,1] AND **R0[1,3]**
= 0 OR (**1** AND **1**)
= 1

R(1)	k = 2 Vertex {1,2 } can be intermediate nodes		A	B	C	D			A	B	C	D
		A	0	0	1	0	A	0	0	1	0
		B	1	0	1	1	B	1	0	1	1
		C	0	0	0	0	C	0	0	0	0
		D	0	1	0	0	D	**1**	**1**	**1**	**1**

R2[4,1]
= R1[4,1] OR
 R1[4,2] AND **R1[2,1]**
= 0 OR (**1** AND **1**)
= 1

R2[4,3]
= R1[4,3] OR
 R1[4,2] AND **R1[2,3]**
= 0 OR (**1** AND **1**)
= 1

R2[4,4]
= R1[4,4] OR
 R1[4,2] AND **R1[2,4]**
= 0 OR (**1** AND **1**)
= 1

R(2)	k = 3 Vertex {1,2,3 } can be intermediate nodes		A	B	C	D			A	B	C	D
		A	0	0	1	0	A	0	0	1	0
		B	1	0	1	1	B	1	0	1	1
		C	0	0	0	0	C	0	0	0	0
		D	1	1	1	1	D	1	1	1	1

NO CHANGE

			A	B	C	D		A	B	C	D
R(3)	k = 4 Vertex {1,2,3,4 } can be intermediate nodes	**A**	0	0	1	0	**A**	0	0	1	0
		B	1	0	1	1	**B**	1	**1**	1	1
		C	0	0	0	0	**C**	0	0	0	0
		D	1	1	1	1	**D**	1	1	1	1

R4[2,2]
= R3[2,2] OR
  R3[2,4] AND **R3[4,2]**
= 0 OR (**1** AND **1**)
= 1

			A	B	C	D	
R(4)		**A**	0	0	1	0	TRANSITIVE CLOSURE for the given graph
		B	1	1	1	1	
		C	0	0	0	0	
		D	1	1	1	1	

Efficiency:
- Time efficiency is $\Theta(n^3)$
- Space efficiency: Requires extra space for separate matrices for recording intermediate results of the algorithm.

4.3 Floyd's Algorithm to find -ALL PAIRS SHORTEST PATHS
Some useful definitions:
- **Weighted Graph:** Each edge has a weight (associated numerical value). Edge weights may represent costs, distance/lengths, capacities, etc. depending on the problem.
- **Weight matrix:** W(i,j) is
   - 0 if i=j
   - ∞ if no edge b/n i and j.
   - ―wieght of edge" if edge b/n i and j.

Problem statement:
Given a weighted graph G(V, Ew), the all-pairs shortest paths problem is to find the shortest path between every pair of vertices (vi, vj) Є V.
Solution:
A number of algorithms are known for solving All pairs shortest path problem
- **Matrix multiplication based algorithm**
- **Dijkstra's algorithm**
- **Bellman-Ford algorithm**
- **Floyd's algorithm**

Underlying idea of Floyd's algorithm:

- Let W denote the initial weight matrix.
- Let D(k) [i, j] denote cost of shortest path from i to j whose intermediate vertices are a subset of {1,2,…,k}.
- **Recursive Definition**

  Case 1:

  A shortest path from v_i to v_j restricted to using only vertices from {$v_1,v_2,…,v_k$} as intermediate vertices does not use v_k. Then

  $$D(k) [i, j] = D(k-1) [i, j].$$

  Case 2:

  A shortest path from v_i to v_j restricted to using only vertices from {$v_1,v_2,…,v_k$} as intermediate vertices do use v_k. Then

  $$D(k) [i, j] = D(k-1) [i, k] + D(k-1) [k, j].$$

We conclude:

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

Algorithm:

Algorithm Floyd(W[1..n, 1..n])

// Implements Floyd's algorithm

// Input: Weight matrix W

// Output: Distance matrix of shortest paths' length

D   W

for k   →   1 to n do

        for i→   1 to n do

        for j→   1 to n do

            D [i, j]→   min { D [i, j], D [i, k] + D [k, j]

return D

Example:

Find All pairs shortest paths for the given weighted connected graph using Floyd's algorithm.
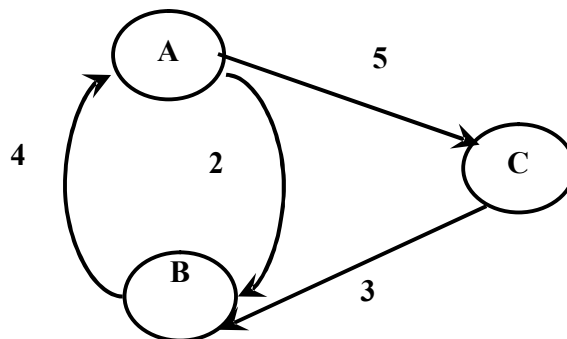
Solution:

D(0)   =

	A	B	C
A	0	2	5
B	4	0	∞
C	∞	3	0

D(0)	k = 1 Vertex 1 can be intermediate node			

	A	B	C			A	B	C
A	0	2	5		A	0	2	5
B	4	0	∞		B	4	0	**9**
C	∞	3	0		C	∞	3	0

D1[2,3]
= min { D0 [2,3],
**   D0 [2,1] + D0 [1,3] }**
= min { ∞, (4 + 5) }
= 9

| D(1) | k = 2
Vertex 1,2
can be |

intermediate nodes

	A	B	C			A	B	C
A	0	2	5		A	0	2	5
B	4	0	9		B	4	0	9
C	∞	3	0		C	**7**	3	0

D2[3,1]
= min { D1 [3,1],
**   D1 [3,2] + D1 [2,1] }**
= min { ∞, (4 + 3) }
= 7

| D(2) | k = 3
Vertex 1,2,3
can be |

intermediate nodes

	A	B	C			A	B	C
A	0	2	5		A	0	2	5
B	4	0	9		B	4	0	9
C	7	3	0		C	7	3	0

| D(3) | | | **NO Change** |

	A	B	C	
A	0	2	5	
B	4	0	9	
C	7	3	0	

ALL PAIRS SHORTEST PATHS for the given graph

4.4 0/1 Knapsack Problem Memory function
Definition:
Given a set of n items of known weights w1,…,wn and values v1,…,vn and a knapsack of capacity W, the problem is to find the most valuable subset of the items that fit into the knapsack.
Knapsack problem is an OPTIMIZATION PROBLEM

Dynamic programming approach to solve knapsack problem

Step 1:
Identify the smaller sub-problems. If items are labeled 1..n, then a sub-problem would be to find an optimal solution for Sk = {items labeled 1, 2, .. k}

Step 2:
Recursively define the value of an optimal solution in terms of solutions to smaller problems.

      Initial conditions:
$$V[\,0, j\,] = 0 \quad \text{for } j \geq 0$$
$$V[\,i, 0\,] = 0 \quad \text{for } i \geq 0$$

      Recursive step:

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

Step 3:
Bottom up computation using iteration

Question:
Apply bottom-up dynamic programming algorithm to the following instance of the knapsack problem Capacity W= 5

Item #	Weight (Kg)	Value (Rs.)
1	2	3
2	3	4
3	4	5
4	5	6

Solution:
Using dynamic programming approach, we have:

Step	Calculation	Table						

Step 1

Calculation:

Initial conditions:
V[0, j] = 0   for j ≥ 0
V[i, 0] = 0   for i ≥ 0

V[i,j]	j=0	1	2	3	4	5
i=0	**0**	**0**	**0**	**0**	**0**	**0**
1	**0**					
2	**0**					
3	**0**					
4	**0**					

Step 2

Calculation:

W1 = 2,
Available knapsack capacity = 1
W1 > WA,   CASE 1 holds:
V[i, j] = V[i-1, j]
V[1,1] = V[0, 1] = 0

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	**0**				
2	0					
3	0					
4	0					

Step 3

Calculation:

W1 = 2,
Available knapsack capacity = 2
W1 = WA,   CASE 2 holds:
V[i, j] = max { V[i-1, j],
             vi +V[i-1, j - wi] }
V[1,2] = max { V[0, 2],
             3 +V[0, 0] }
= max { 0, 3 + 0 } = 3

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	0	**3**			
2	0					
3	0					
4	0					

Step 4

Calculation:

W1 = 2,
Available knapsack capacity =
                     3,4,5
W1 < WA,   CASE 2 holds:
V[i, j] = max { V[i-1, j],
             vi +V[i-1, j - wi] }
V[1,3] = max { V[0, 3],
             3 +V[0, 1] }
= max { 0, 3 + 0 } = 3

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	0	3	**3**	**3**	**3**
2	0					
3	0					
4	0					

Step 5

Calculation:

W2 = 3,
Available knapsack capacity = 1
W2 >WA,   CASE 1 holds:
V[i, j] = V[i-1, j]
V[2,1] = V[1, 1] = 0

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	**0**				
3	0					
4	0					

6	W2 = 3, Available knapsack capacity = 2 W2 >WA, CASE 1 holds: V[i, j] = V[i-1, j] V[2,2] = V[1, 2] = 3	**V[i,j]**	**j=0**	**1**	**2**	**3**	**4**	**5**
		i=0	0	0	0	0	0	0
		1	0	0	3	3	3	3
		2	0	0	**3**			
		3	0					
		4	0					

7	W2 = 3,
	Available knapsack capacity = 3
	W2 = WA, CASE 2 holds:
	V[i, j] = max { V[i-1, j],
	            vi +V[i-1, j - wi] }
	V[2,3] = max { V[1, 3],
	            4 +V[1, 0] }
	= max { 3, 4 + 0 } = 4
		V[i,j]	**j=0**	**1**	**2**	**3**	**4**	**5**

		i=0	0	0	0	0	0	0
		1	0	0	3	3	3	3
		2	0	0	3	**4**		
		3	0					
		4	0					

8	W2 = 3,
	Available knapsack capacity = 4
	W2 < WA, CASE 2 holds:
	V[i, j] = max { V[i-1, j],
	            vi +V[i-1, j - wi] }
	V[2,4] = max { V[1, 4],
	            4 +V[1, 1] }
	= max { 3, 4 + 0 } = 4
		V[i,j]	**j=0**	**1**	**2**	**3**	**4**	**5**

		i=0	0	0	0	0	0	0
		1	0	0	3	3	3	3
		2	0	0	3	4	**4**	
		3	0					
		4	0					

9	W2 = 3,
	Available knapsack capacity = 5
	W2 < WA, CASE 2 holds:
	V[i, j] = max { V[i-1, j],
	            vi +V[i-1, j - wi] }
	V[2,5] = max { V[1, 5],
	            4 +V[1, 2] }
	= max { 3, 4 + 3 } = 7
		V[i,j]	**j=0**	**1**	**2**	**3**	**4**	**5**

		i=0	0	0	0	0	0	0
		1	0	0	3	3	3	3
		2	0	0	3	4	4	**7**
		3	0					
		4	0					

10	W3 = 4, Available knapsack capacity = 1,2,3 W3 > WA, CASE 1 holds: V[i, j] = V[i-1, j]	**V[i,j]**	**j=0**	**1**	**2**	**3**	**4**	**5**
		i=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**		
		4	0					

11	W3 = 4,
	Available knapsack capacity = 4
	W3 = WA, CASE 2 holds:
	V[i, j] = max { V[i-1, j],
	       vi +V[i-1, j - wi] }
	V[3,4] = max { V[2, 4],
	       5 +V[2, 0] }
	= max { 4, 5 + 0 } = 5						

Step 11:

V[i,j]	j=0	1	2	3	4	5
i=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**	
4	0					

Step 12:

W3 = 4,
Available knapsack capacity = 5
W3 < WA,   CASE 2 holds:
V[i, j] = max { V[i-1, j],
      vi +V[i-1, j - wi] }
V[3,5] = max { V[2, 5],
      5 +V[2, 1] }
     = max { 7, 5 + 0 } = 7

V[i,j]	j=0	1	2	3	4	5
i=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					

Step 13:

W4 = 5,
Available knapsack capacity = 1,2,3,4
W4 < WA,   CASE 1 holds:
V[i, j] = V[i-1, j]

V[i,j]	j=0	1	2	3	4	5
i=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**	

Step 14:

W4 = 5,
Available knapsack capacity = 5
W4 = WA,   CASE 2 holds:
V[i, j] = max { V[i-1, j],
      vi +V[i-1, j - wi] }
V[4,5] = max { V[3, 5],
      6 +V[3, 0] }
= max { 7, 6 + 0 } = 7

V[i,j]	j=0	1	2	3	4	5
i=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**

Maximal value is V [4, 5] = 7/-

What is the composition of the optimal subset?

The composition of the optimal subset if found by tracing back the computations for the entries in the table.

Step	Table	Remarks
1	<table><tr><td>V[i,j]</td><td>j=0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><td>i=0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td><td>3</td><td>3</td><td>3</td><td>3</td></tr><tr><td>2</td><td>0</td><td>0</td><td>3</td><td>4</td><td>4</td><td>7</td></tr><tr><td>3</td><td>0</td><td>0</td><td>3</td><td>4</td><td>5</td><td>7</td></tr><tr><td>4</td><td>0</td><td>0</td><td>3</td><td>4</td><td>5</td><td>7</td></tr></table>	V[4, 5] = V[3, 5] **ITEM 4 NOT included in the subset**

V[i,j] table (Step 1):

V[i,j]	j=0	1	2	3	4	5
i=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

Step 1 Remarks: V[4, 5] = V[3, 5] — **ITEM 4 NOT included in the subset**

Step 2 Table:

V[i,j]	j=0	1	2	3	4	5
i=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

Step 2 Remarks: V[3, 5] = V[2, 5] — **ITEM 3 NOT included in the subset**

Step 3 Table:

V[i,j]	j=0	1	2	3	4	5
i=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

Step 3 Remarks: V[2, 5] ≠ V[1, 5] — **ITEM 2 included in the subset**

Step 4: **Since item 2 is included in the knapsack:** Weight of item 2 is **3kg**, therefore, remaining capacity of the knapsack is

(5 - 3 =) **2**kg

V[i,j]	j=0	1	2	3	4	5
i=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

Step 4 Remarks: V[1, 2] ≠ V[0, 2] — **ITEM 1 included in the subset**

Step 5: **Since item 1 is included in the knapsack:** Weight of item 1 is 2kg, therefore, remaining capacity of the knapsack is

(2 - 2 =) 0 kg.

Step 5 Remarks: **Optimal subset: { item 1, item 2 }**

Total weight is: 5kg   (2kg + 3kg)
Total profit is: 7/-   (3/- + 4/-)

Efficiency:
- Running time of Knapsack problem using dynamic programming algorithm is: $O(n * W)$
- Time needed to find the composition of an optimal solution is: $O(n + W)$

Memory function

- Memory function combines the strength of top-down and bottom-up approaches
- It solves ONLY sub-problems that are necessary and does it ONLY ONCE.

The method:
- Uses top-down manner.
- Maintains table as in bottom-up approach.
- Initially, all the table entries are initialized with special —null" symbol to indicate that they have not yet been calculated.
- Whenever a new value needs to be calculated, the method checks the corresponding entry in the table first:
- If entry is NOT —null", it is simply retrieved from the table.
- Otherwise, it is computed by the recursive call whose result is then recorded in the table.

Algorithm:

Algorithm MFKnap(i, j)
if V[i, j] < 0
      if j < Weights[i]
          value   →   MFKnap(i-1, j)
      else
          value   →   max {MFKnap(i-1, j),
                   Values[i] + MFKnap(i-1, j - Weights[i])}
      V[i, j]→   value
return V[i, j]

Example:
Apply memory function method to the following instance of the knapsack problem
Capacity W= 5

Item #	Weight (Kg)	Value (Rs.)
1	2	3
2	3	4
3	4	5
4	5	6

Solution:
Using memory function approach, we have:

	Computation	**Remarks**
1	Initially, all the table entries are initialized with special —null" symbol to indicate that they have not yet been calculated. Here null is indicated with -1 value.	<table><tr><td>V[i,j]</td><td>j=0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><td>i=0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>1</td><td>0</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td></tr><tr><td>2</td><td>0</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td></tr><tr><td>3</td><td>0</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td></tr><tr><td>4</td><td>0</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td></tr></table>

Row 1 table (initialization):

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	-1	-1	-1	-1	-1
2	0	-1	-1	-1	-1	-1
3	0	-1	-1	-1	-1	-1
4	0	-1	-1	-1	-1	-1

2. Computation:

```
MFKnap( 4, 5 )
   ↙        ↘
MFKnap( 3, 5 )    6 + MFKnap( 3, 0 )
   ↙        ↘
MFKnap( 2, 5 )    5 + MFKnap( 2, 1 )
   ↙        ↘
MFKnap( 1, 5 )    4 + MFKnap( 1, 2 )
   0        3
   ↙    ↘
MFKnap( 0, 5 )    3 + MFKnap( 0, 3 )
   0                3 + 0
```

Remarks:

V[1, 5] = 3

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	-1	-1	-1	-1	**3**
2	0	-1	-1	-1	-1	-1
3	0	-1	-1	-1	-1	-1
4	0	-1	-1	-1	-1	-1

3. Computation:

```
MFKnap( 4, 5 )
   ↙        ↘
MFKnap( 3, 5 )    6 + MFKnap( 3, 0 )
   ↙        ↘
MFKnap( 2, 5 )    5 + MFKnap( 2, 1 )
   ↙        ↘
MFKnap( 1, 5 )    4 + MFKnap( 1, 2 )
   3              0        3
              ↙    ↘
       MFKnap( 0, 2 )    3 + MFKnap( 0, 0 )
          0                3 + 0
```

Remarks:

V[1, 2] = 3

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	-1	**3**	-1	-1	**3**
2	0	-1	-1	-1	-1	-1
3	0	-1	-1	-1	-1	-1
4	0	-1	-1	-1	-1	-1

4. Computation:

```
MFKnap( 4, 5 )
   ↙        ↘
MFKnap( 3, 5 )    6 + MFKnap( 3, 0 )
   ↙        ↘
MFKnap( 2, 5 )    5 + MFKnap( 2, 1 )
   3              7
   ↙        ↘
MFKnap( 1, 5 )    4 + MFKnap( 1, 2 )
   3                3
```

Remarks:

V[2, 5] = 7

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	-1	**3**	-1	-1	**3**
2	0	-1	-1	-1	-1	**7**
3	0	-1	-1	-1	-1	-1
4	0	-1	-1	-1	-1	-1

5	MFKnap(4, 5) ↙ ↘ MFKnap(3, 5) 6 + MFKnap(3, 0) 7 5 ↙ ↘ MFKnap(2, 5) 5 + MFKnap(2, 1) 7 ↓ 0 MFKnap(1, 1) ↓ 0 MFKnap(0, 1) 0	**V[2, 1] = 0** **V[3, 5] = 7**

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	**0**	**3**	-1	-1	**3**
2	0	**0**	-1	-1	-1	**7**
3	0	-1	-1	-1	-1	**7**
4	0	-1	-1	-1	-1	-1

6	7 MFKnap(4, 5) 7 6 ↙ ↘ MFKnap(3, 5) 6 + MFKnap(3, 0) 7 0	**V[4, 5] = 7**

V[i,j]	j=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	**0**	**3**	-1	-1	**3**
2	0	**0**	-1	-1	-1	**7**
3	0	-1	-1	-1	-1	**7**
4	0	-1	-1	-1	-1	**7**

The composition of the optimal subset if found by **tracing back the computations for the entries in the table as done with the early knapsack problem**

Conclusion:
Optimal subset: { item 1, item 2 }

Total weight is: 5kg   (2kg + 3kg)
Total profit is: 7/-   (3/- + 4/-)

Efficiency:

- Time efficiency same as bottom up algorithm: O(n * W) + O(n + W)
- Just a constant factor gain by using memory function
- Less space efficient than a space efficient version of a bottom-up algorithm

UNIT-5
DECREASE-AND-CONQUER APPROACHES, SPACE-TIMETRADEOFFS

5.1DECREASE-AND-CONQUER APPROACHES: INTRODUCTION
5.2 INSERTION SORT
5.3 DEPTH FIRST SEARCH AND BREADTH FIRST SEARCH
5.4TOPOLOGICAL SORTING
5.5SPACE-TIME TRADEOFFS: INTRODUCTION
5.6SORTING BY COUNTING
5.7 INPUT ENHANCEMENT IN STRING MATCHING.

5.1 INTRODUCTION:

Decrease & conquer is a general algorithm design strategy based on exploiting the relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem. The exploitation can be either top-down (recursive) or bottom-up (non-recursive).

The major variations of decrease and conquer are
1. Decrease by a constant :(usually by 1):
    a. insertion sort
    b. graph traversal algorithms (DFS and BFS)
    c. topological sorting
    d. algorithms for generating permutations, subsets

2. Decrease by a constant factor (usually by half)
    a. binary search and bisection method

3. Variable size decrease
    a. Euclid's algorithm

Decrease by a constant :(usually by 1):
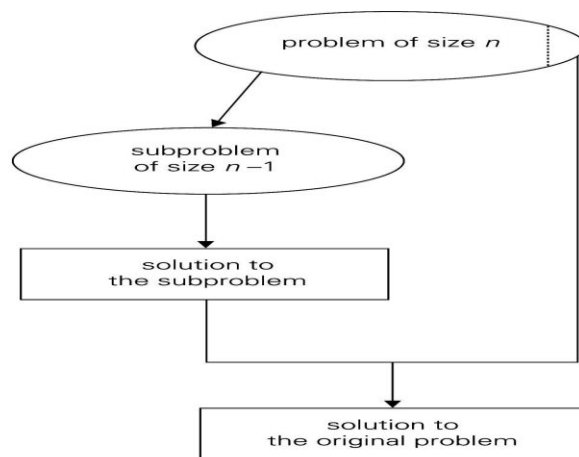
Decrease by a constant factor (usually by half)
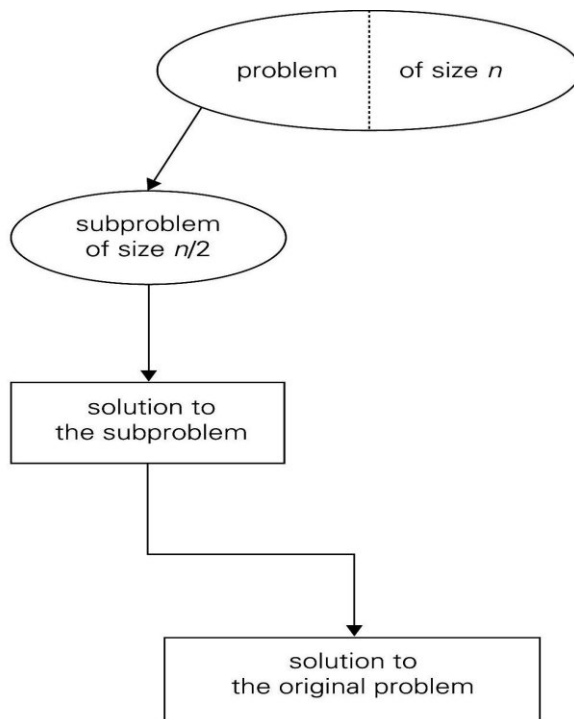
5.2 INSERTION SORT

Description:
Insertion sort is an application of decrease & conquer technique. It is a comparison based sort in which the sorted array is built on one entry at a time

$$A[0] \leq \ldots \leq A[j] < A[j+1] \leq \ldots \leq A[i-1] \mid A[i] \ldots A[n-1]$$

smaller than or equal to $A[i]$ greater than $A[i]$

Algorithm:

ALGORITHM Insertionsort(A [0 … n-1])
//sorts a given array by insertion sort
//i/p: Array A[0…n-1]
//o/p: sorted array A[0…n-1] in ascending order

for i   1 to n-1
   V   A[i]
   j   i-1

```
        while j ≥ 0 AND A[j] > V do
                A[j+1]    A[j]
                j    j – 1
     A[j + 1]    V
```

Analysis:

- **Input size:** Array size, n
- **Basic operation:** key comparison
- Best, worst, average case exists
  Best case: when input is a sorted array in ascending order:
  Worst case: when input is a sorted array in descending order:
- Let $C_{worst}(n)$ be the number of key comparison in the worst case. Then

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

- Let $C_{best}(n)$ be the number of key comparison in the best case.
  Then

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

Example:
Sort the following list of elements using insertion sort:
        89, 45, 68, 90, 29, 34, 17

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

Advantages of insertion sort:

- Simple implementation. There are three variations
  - o Left to right scan
  - o Right to left scan
  - o Binary insertion sort
- Efficient on small list of elements, on almost sorted list
- Running time is linear in best case
- Is a stable algorithm
- Is a in-place algorithm

5.3 DEPTH-FIRST SEARCH (DFS) AND BREADTH-FIRST SEARCH (BFS)

DFS and BFS are two graph traversing algorithms and follow decrease and conquer approach – decrease by one variation to traverse the graph

Some useful definition:
- **Tree edges:** edges used by DFS traversal to reach previously unvisited vertices
- **Back edges:** edges connecting vertices to previously visited   vertices other than their immediate predecessor in the traversals
- **Cross edges:** edge that connects an unvisited vertex to vertex other than its immediate predecessor. (connects siblings)
- **DAG:** Directed acyclic graph

Depth-first search (DFS)
Description:
- DFS starts visiting vertices of a graph at an arbitrary vertex by marking it as visited.
- It visits graph's vertices by always moving away from last visited vertex to an unvisited one, backtracks if no adjacent unvisited vertex is available.
- Is a recursive algorithm, it uses a stack
- A vertex is pushed onto the stack when it's reached for the first time
- A vertex is popped off the stack when it becomes a dead end, i.e., when there is no adjacent unvisited vertex
- ―Redraws" graph in tree-like fashion (with tree edges and back edges for undirected graph)

Algorithm:
ALGORITHM DFS (G)
//implements DFS traversal of a given graph
//i/p: Graph G = { V, E}
//o/p: DFS tree

Mark each vertex in V with 0 as a mark of being ―unvisited"
count   0
for each vertex v in V do
        if v is marked with 0
                dfs(v)

dfs(v)
count   count + 1
mark v with count
for each vertex w in V adjacent to v do
        if w is marked with 0
                dfs(w)
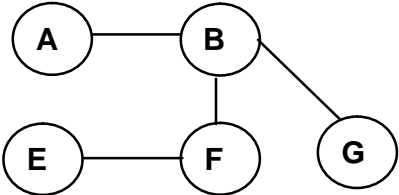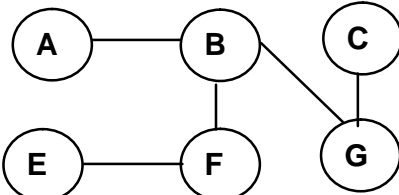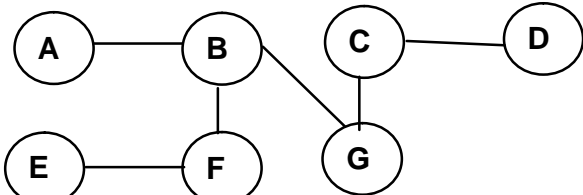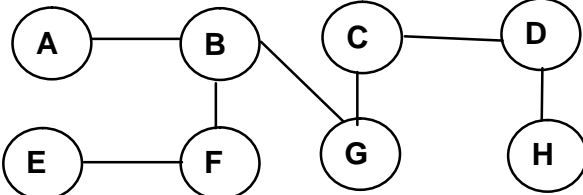
Example:

Starting at vertex A traverse the following graph using DFS traversal method:
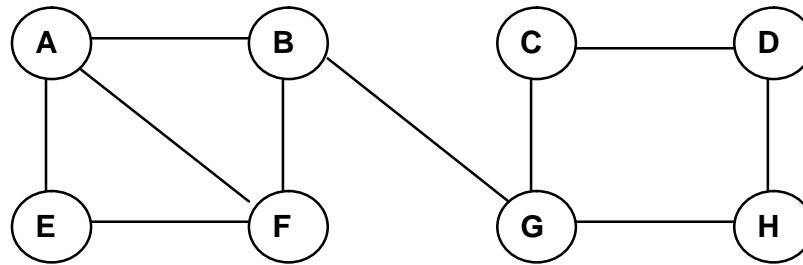
Solution:

Step	Graph	Remarks
1	A	Insert A into stack A(1)
2	A — B	Insert B into stack B (2) A(1)
3	A — B F	Insert F into stack F (3) B (2) A(1)
4	A — B E — F	Insert E into stack E (4) F (3) B (2) A(1)
5	NO unvisited adjacent vertex for E, backtrack	Delete E from stack E (4, 1) F (3) B (2) A(1)

6	NO unvisited adjacent vertex for F, backtrack	Delete F from stack E (4, 1) F (3, 2) B (2) A(1)
7		Insert G into stack E (4, 1) F (3, 2) G (5) B (2) A(1)
8		Insert C into stack E (4, 1) C (6) F (3, 2) G (5) B (2) A(1)
9		Insert D into stack D (7) E (4, 1) C (6) F (3, 2) G (5) B (2) A(1)

10		Insert H into stack
		        H (8)
		        D (7)
		E (4, 1) C (6)
		F (3, 2) G (5)
		B (2)
		A(1)

11	NO unvisited adjacent vertex for H, backtrack	Delete H from stack
		        H (8, 3)
		        D (7)
		E (4, 1) C (6)
		F (3, 2) G (5)
		B (2)
		A(1)

12	NO unvisited adjacent vertex for D, backtrack	Delete D from stack H (8, 3) D (7, 4) E (4, 1) C (6) F (3, 2) G (5) B (2) A(1)
13	NO unvisited adjacent vertex for C, backtrack	Delete C from stack H (8, 3) D (7, 4) E (4, 1) C (6, 5) F (3, 2) G (5) B (2) A(1)
14	NO unvisited adjacent vertex for G, backtrack	Delete G from stack H (8, 3) D (7, 4) E (4, 1) C (6, 5) F (3, 2) G (5, 6) B (2) A(1)
15	NO unvisited adjacent vertex for B, backtrack	Delete B from stack H (8, 3) D (7, 4) E (4, 1) C (6, 5) F (3, 2) G (5, 6) B (2, 7) A(1)
16	NO unvisited adjacent vertex for A, backtrack	Delete A from stack H (8, 3) D (7, 4) E (4, 1) C (6, 5) F (3, 2) G (5, 6) B (2, 7) A(1, 8)
	Stack becomes empty. Algorithm stops as all the nodes in the given graph are visited	

The DFS tree is as follows: (dotted lines are back edges)
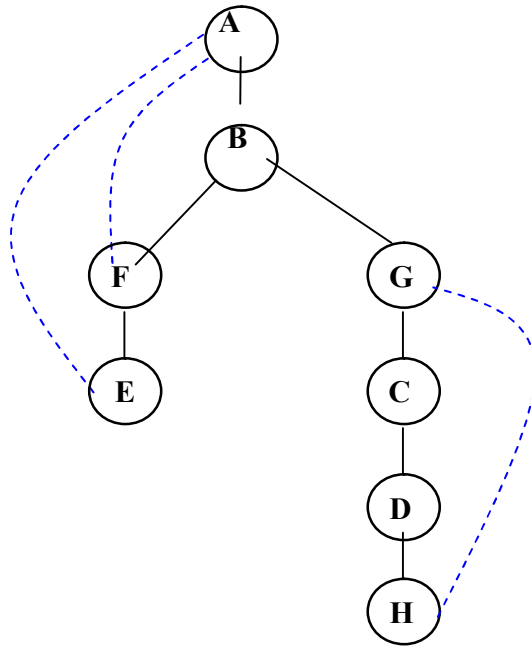
Applications of DFS:
- The two orderings are advantageous for various applications like topological sorting, etc
- To check connectivity of a graph (number of times stack becomes empty tells the number of components in the graph)
- To check if a graph is acyclic. (no back edges indicates no cycle)
- To find articulation point in a graph

Efficiency:
- Depends on the graph representation:
  - Adjacency matrix : $\Theta(n^2)$
  - Adjacency list: $\Theta(n + e)$

Breadth-first search (BFS)
Description:
- BFS starts visiting vertices of a graph at an arbitrary vertex by marking it as visited.
- It visits graph's vertices by across to all the neighbors of the last visited vertex
- Instead of a stack, BFS uses a queue
- Similar to level-by-level tree traversal
- ―Redraws‖ graph in tree-like fashion (with tree edges and cross edges for undirected graph)

Algorithm:
ALGORITHM BFS (G)
//implements BFS traversal of a given graph
//i/p: Graph G = { V, E}
//o/p: BFS tree/forest
Mark each vertex in V with 0 as a mark of being ―unvisited‖

count   0
for each vertex v in V do
      if v is marked with 0
          bfs(v)
bfs(v)
count   count + 1
mark v with count and initialize a queue with v
while the queue is NOT empty do
      for each vertex w in V adjacent to front's vertex v do
          if w is marked with 0
              count   count + 1
              mark w with count
              add w to the queue
      remove vertex v from the front of the queue
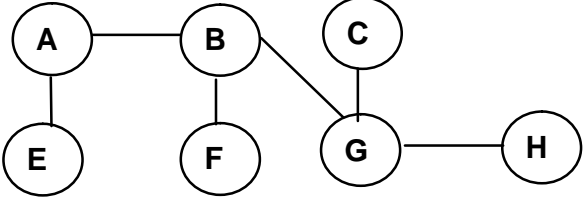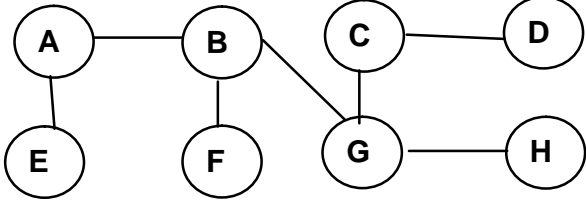
Example:

Starting at vertex A traverse the following graph using BFS traversal method:
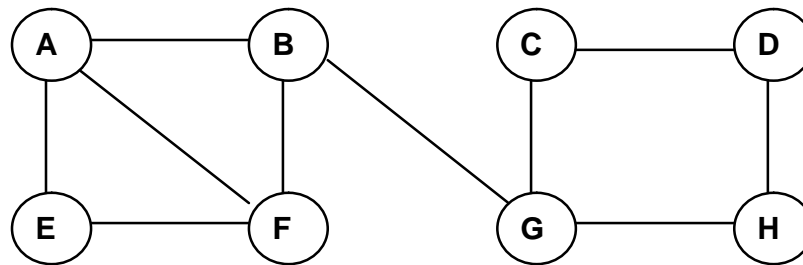
Solution:

Step	Graph	Remarks
1		Insert A into queue A(1)
2		Insert B, E into queue A(1), B (2), E(3) B (2), E(3)
3		Insert F, G into queue B(2), E(3), F(3), G(4) E(3), F(3), G(4)

4	NO unvisited adjacent vertex for E, backtrack	Delete E from queue F(3), G(4)
5	NO unvisited adjacent vertex for F, backtrack	Delete F from queue G(4)
6		Insert C, H into queue G(4), C(5), H(6) C(5), H(6)
7		Insert D into queue C(5), H(6), D(7) H(6), D(7)
8	NO unvisited adjacent vertex for H, backtrack	Delete H from queue D(7)
9	NO unvisited adjacent vertex for D, backtrack	Delete D from queue
	Queue becomes empty. Algorithm stops as all the nodes in the given graph are visited	

The BFS tree is as follows: (dotted lines are cross edges)
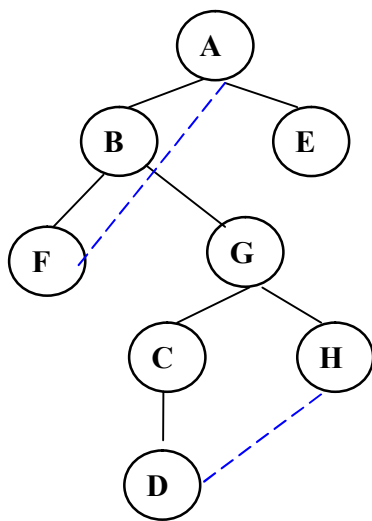
Applications of BFS:

- To check connectivity of a graph (number of times queue becomes empty tells the number of components in the graph)
- To check if a graph is acyclic. (no cross edges indicates no cycle)
- To find minimum edge path in a graph

Efficiency:

- Depends on the graph representation:
  - Array : $\Theta(n^2)$
  - List: $\Theta(n+e)$

Difference between DFS & BFS:

	DFS	BFS
Data structure	Stack	Queue
No. of vertex orderings	2 orderings	1 ordering
Edge types	Tree edge	Tree edge
	Back edge	Cross edge
Applications	Connectivity	Connectivity
	Acyclicity	Acyclicity
	Articulation points	Minimum edge paths
Efficiency for adjacency matrix	$\Theta(n^2)$	$\Theta(n^2)$
Efficiency for adjacency lists	$\Theta(n+e)$	$\Theta(n+e)$

5.4 Topological Sorting

Description:

Topological sorting is a sorting method to list the vertices of the graph in such an order that for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends.

NOTE: There is no solution for topological sorting if there is a cycle in the digraph .
[MUST be a DAG]

Topological sorting problem can be solved by using
1. DFS method
2. Source removal method

DFS Method:

- Perform DFS traversal and note the order in which vertices become dead ends (popped order)
- Reverse the order, yield the topological sorting.

Example:

Apply DFS – based algorithm to solve the topological sorting problem for the given graph:

Step	Graph	Remarks
1	C1	Insert C1 into stack C1(1)
2	C1 → C3	Insert C2 into stack C2 (2) C1(1)
3	C1 → C3 → C4	Insert C4 into stack C4 (3) C2 (2) C1(1)
4	C1 → C3 → C4 → C5	Insert C5 into stack C5 (4) C4 (3) C2 (2) C1(1)

5	NO unvisited adjacent vertex for C5, backtrack	Delete C5 from stack C5 (4, 1) C4 (3) C2 (2) C1(1)
6	NO unvisited adjacent vertex for C4, backtrack	Delete C4 from stack C5 (4, 1) C4 (3, 2) C2 (2) C1(1)

7	NO unvisited adjacent vertex for C3, backtrack	Delete C3 from stack C5 (4, 1) C4 (3,2) C2 (2, 3) C1(1)
8	NO unvisited adjacent vertex for C1, backtrack	Delete C1 from stack C5 (4, 1) C4 (3,2) C2 (2, 3) C1(1, 4)
Stack becomes empty, but there is a node which is unvisited, therefore start the DFS again from arbitrarily selecting a unvisited node as source		
9	C2	Insert C2 into stack C5 (4, 1) C4 (3,2) C2 (2, 3) C1(1, 4) C2(5)
10	NO unvisited adjacent vertex for C2, backtrack	Delete C2 from stack C5 (4, 1) C4 (3,2) C2 (2, 3) C1(1, 4) C2(5, 5)
Stack becomes empty, NO unvisited node left, therefore algorithm stops. The popping – off order is: C5, C4, C3, C1, C2, Topologically sorted list (reverse of pop order): C2, C1 C3 C4 C5		

Source removal method:

- Purely based on decrease & conquer
- Repeatedly identify in a remaining digraph a source, which is a vertex with no incoming edges
- Delete it along with all the edges outgoing from it.

Example:

Apply Source removal – based algorithm to solve the topological sorting problem for the given graph:
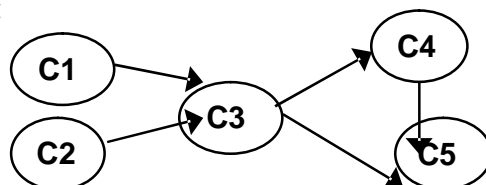
Solution:
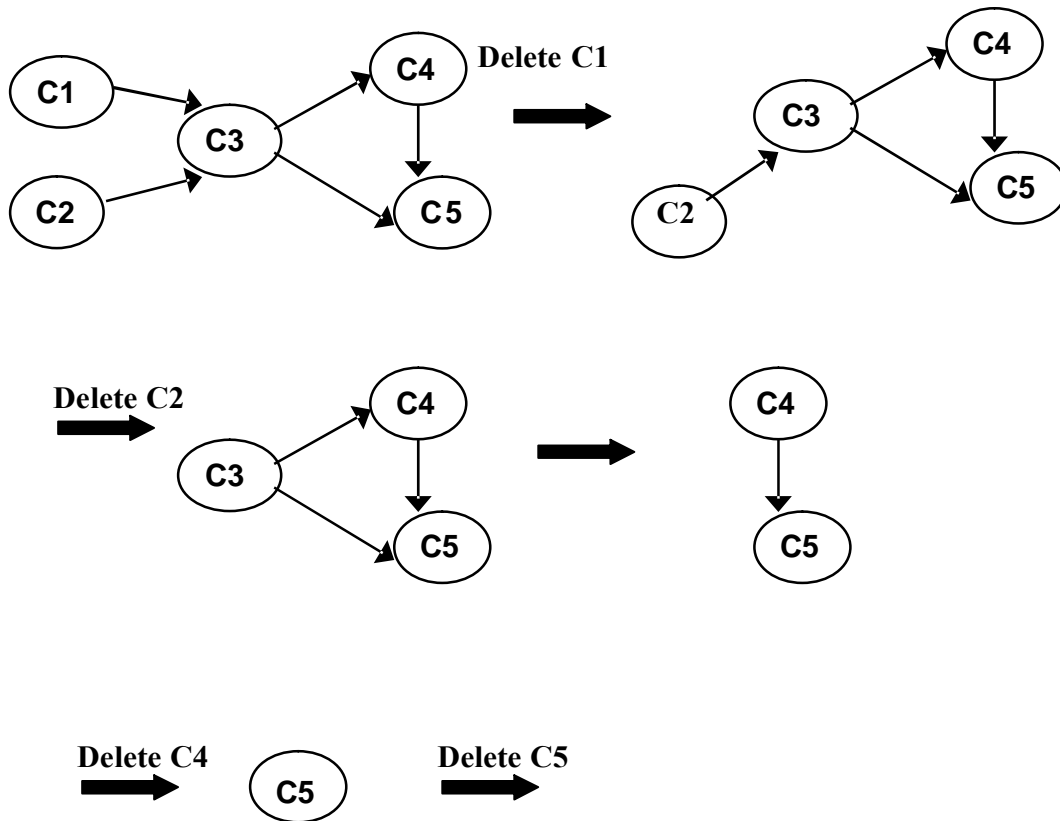
The topological order is C1, C2, C3, C4, C5

5.5 SPACE-TIME TRADEOFFS:

Introduction

Two varieties of space-for-time algorithms:

- <u>input enhancement</u> — preprocess the input (or its part) to store some info to be used later in solving the problem

    - counting sorts
    - string searching algorithms

- <u>pre structuring</u> — preprocess the input to make accessing its elements easier   1)hashing   2)indexing schemes (e.g., B-trees)

5.6 SORTING BY COUNTING

Assume elements to be sorted belong to a known set of small values between l and u, with potential duplication

Constraint: we cannot overwrite the original list Distribution Counting: compute the frequency of each

element and later accumulate sum of frequencies (distribution)

Algorithm:

    for j ← 0 **to** u-l **do** D[j] ← 0   // init frequencies

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

    for j ← 1 **to** u-l **do** D[j] ← D[j-1] + D[j]   // reuse for distribution

    for i ← n-1 **downto** 0 **do**

        j ← A[i] - l

        S[D[j] - 1] ← A[i]

        D[j] ← D[j] - 1

        return S

Example: A =

13	11	12	13	12	12

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

D[0]	D[1]	D[2]		S[0]	S[1]	S[2]	S[3]	S[4]	S[5]
1	4	6					12		
1	3	6				12			
1	2	6							13
1	2	5			12				
1	1	5		11					
0	1	5						13	

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

Efficiency: Θ(n)

Best so far but only for specific types of input

5.7 INPUT ENHANCEMENT IN STRING MATCHING.

Horspool's Algorithm

A simplified version of Boyer-Moore algorithm: preprocesses pattern to generate a shift table that determines how much to shift the patter when a mismatch occurs. Always makes a shift based on the text's character c aligned with the last compared (mismatched) character in the pattern according to the shift table's entry for c

```
void horspoolInitocc()
{
    int j;
    char a;

    for (a=0; a<alphabetsize; a++)
        occ[a]=-1;

    for (j=0; j<m-1; j++)
    {
        a=p[j];
        occ[a]=j;
```

```
    }
}
void horspoolSearch()
{
    int i=0, j;
    while (i<=n-m)
    {
        j=m-1;
        while (j>=0 && p[j]==t[i+j]) j--;
        if (j<0) report(i);
        i+=m-1;
        i-=occ[t[i]];
    }
}
```

Shift table

- Shift sizes can be precomputed by the formula

$$t(c) = \begin{cases} \text{distance from } c\text{'s rightmost occurrence in pattern} \\ \text{among its first } m\text{-1 characters to its right end} \\ \\ \text{pattern's length } m, \text{ otherwise} \end{cases}$$

  by scanning pattern before search begins and stored in a table called *shift table*. After the shift, the right end of pattern is $t(c)$ positions to the right of the last compared character in text.

- Shift table is indexed by text and pattern alphabet
  Eg, for BAOBAB :

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	2	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6	6	6	6

Example of Horspool's algorithm

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	_
1	2	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6	6	6	6	6

```
BARD  LOVED  BANANAS
BAOBAB
        BAOBAB
            BAOBAB
            BAOBAB  (unsuccessful search)
```

If k characters are matched before the mismatch, then the shift distance is $d_1 = t(c) - k$.

```
.................czyx.........
... ....bzyx
```
k

$t(c)$

```
... ....bzyx
```

Note that the shift could be negative!
  E.g. if text = ...ABAB B...

Time complexity

- preprocessing phase in $O(m+$ п$)$ time and $O($п$)$ space complexity.

- searching phase in $O(mn)$ time complexity.

- the average number of comparisons for one text character is between $1/$п and $2/($п$+1)$.

  (п is the number of storing characters)

UNIT 6
LIMITATIONS OF ALGORITHMIC POWER AND COPING WITH THEM

6.1 LOWER-BOUND ARGUMENTS

6.2 DECISION TREES

6.3 P, NP, AND NP-COMPLETE PROBLEMS

Objectives
We now move into the third and final major theme for this course.
1. *Tools* for analyzing algorithms.
2. *Design strategies* for designing algorithms.
3. Identifying and coping with the *limitations* of algorithms.

Efficiency of an algorithm
- By establishing the asymptotic efficiency class

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

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

6.1 LOWER BOUNDS ARGUMENTS

Lower bound: an estimate on a minimum amount of work needed to solve a given problem
Examples:
- number of comparisons needed to find the largest element in a set of *n* numbers
- number of comparisons needed to sort an array of size *n*
- number of comparisons necessary for searching in a sorted array
- number of multiplications needed to multiply two *n*-by-*n* matrices
Lower bound can be
  - an exact count
  - an efficiency class (Ω)
- *Tight* lower bound: there exists an algorithm with the same efficiency as the lower bound

Problem	Lower bound		Tightness
sorting	$\Omega(n\log n)$	yes	
searching in a sorted array	$\Omega(\log n)$		yes
element uniqueness	$\Omega(n\log n)$		yes
n-digit integer multiplication	$\Omega(n)$		unknown
multiplication of n-by-n matrices	$\Omega(n2)$		unknown

Methods for Establishing Lower Bounds
- Trivial lower bounds
- Information-theoretic arguments (decision trees)
- Adversary arguments
- Problem reduction

Trivial Lower Bounds
Trivial lower bounds: based on counting the number of items that must be processed in input and generated as output
Examples
- finding max element
- polynomial evaluation
- sorting
- element uniqueness
- computing the product of two n-by-n matrices
Conclusions
- may and may not be useful
- be careful in deciding how many elements must be processed

6.2 DECISION TREES
Decision tree — a convenient model of algorithms involving Comparisons in which: (i) internal nodes represent comparisons (ii) leaves represent outcomes
Decision tree for 3-element insertion sort
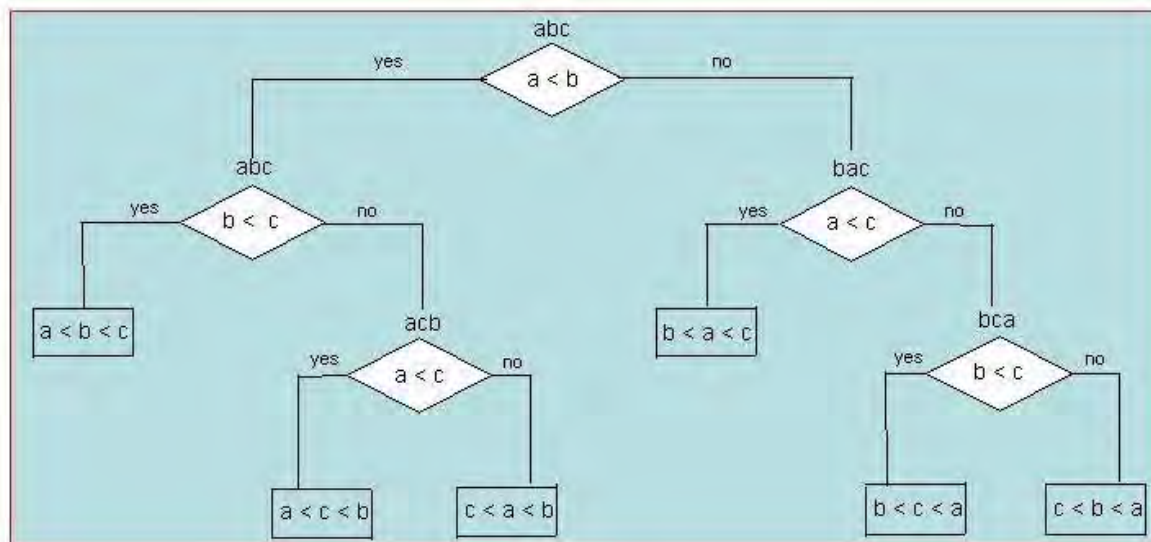
Deriving a Lower Bound from Decision Trees
- How does such a tree help us find lower bounds?
  - There must be at least one leaf for each correct output.
  - The tree must be tall enough to have that many leaves.
- In a binary tree with l leaves and height h,

$$h \geq \log_2 l$$

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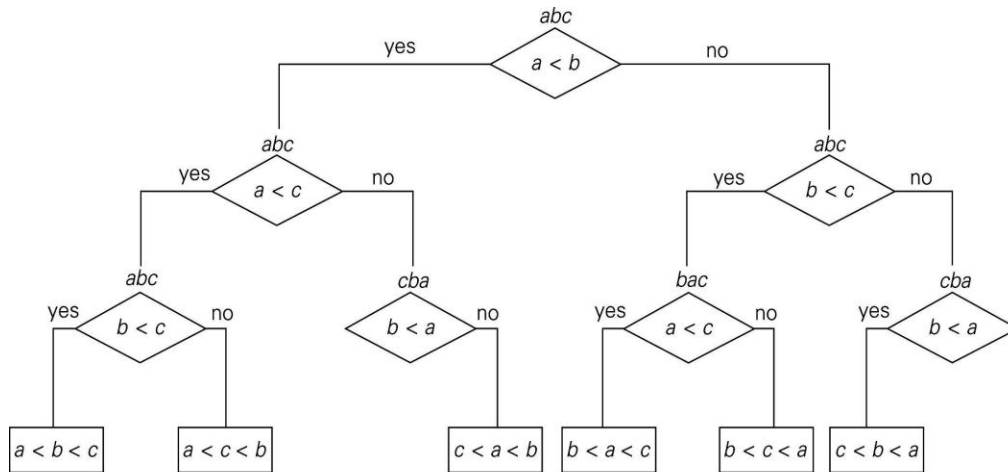
FIGURE 11.2 Decision tree for the three-element selection sort. A triple above a node indicates the state of the array being sorted. Note the two redundant comparisons $b < a$ with a single possible outcome because of the results of some previously made comparisons.

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

Decision Tree and Searching a Sorted Array
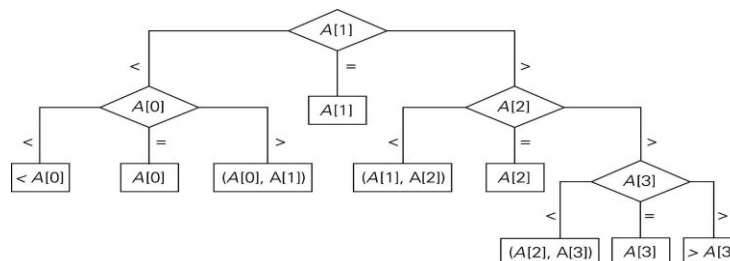
FIGURE 11.4 Ternary decision tree for binary search in a four-element array

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

Decision Tree and Searching a Sorted Array
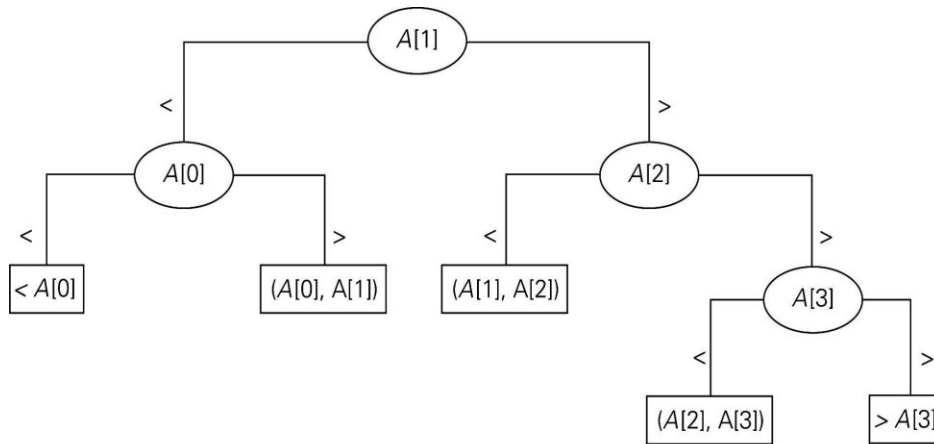
FIGURE 11.5 Binary decision tree for binary search in a four-element array

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

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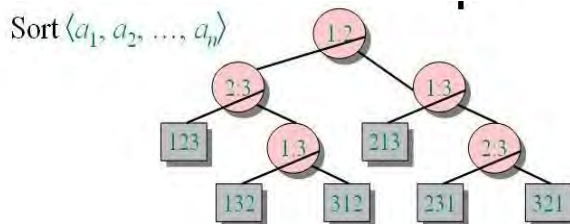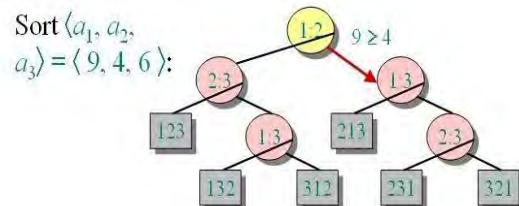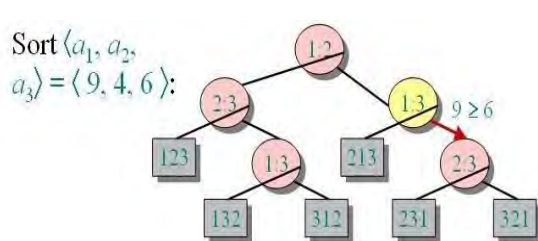
Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.
- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.

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

Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$:

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

Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$:

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

Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$:
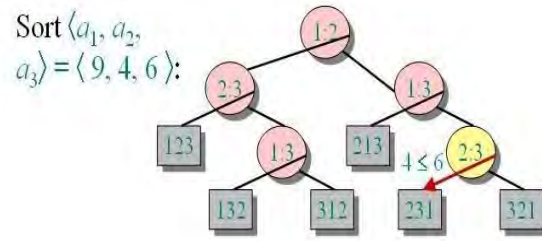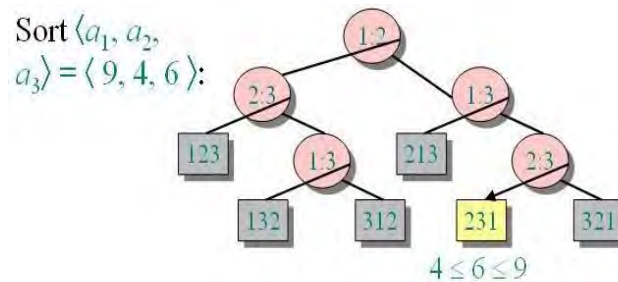
Each leaf contains a permutation $\langle \pi(1), \pi(2),..., \pi(n) \rangle$ to indicate that the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$ has been established.

Decision-tree model

A decision tree can model the execution of any comparison sort:
  • One tree for each input size n.
  • View the algorithm as splitting whenever it compares two elements.
  • The tree contains the comparisons along all possible instruction traces.
  • The running time of the algorithm = the length of the path taken.
  • Worst-case running time = height of tree.

Any comparison sort can be turned into a Decision tree


```
class InsertionSortAlgorithm  {
    for (int i = 1; i < a.length; i++) {
        int j = i;
        while ((j > 0) && (a[j-1] > a[i])) {
            a[j] = a[j-1];
            j--; }
        a[j] = B;  }}
```
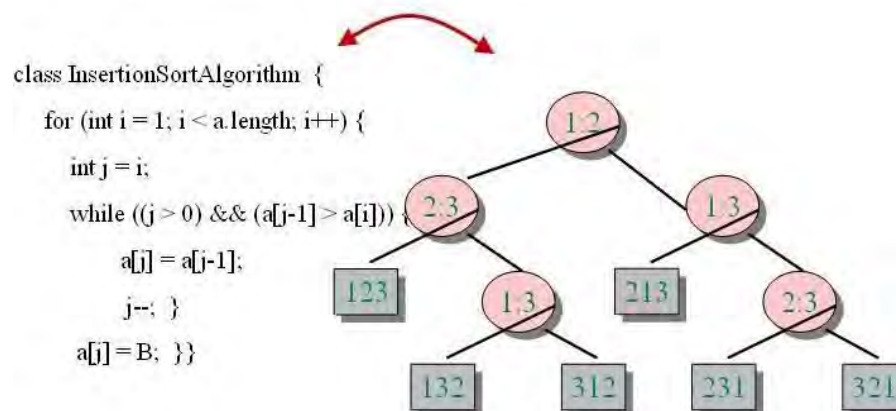
Decision Tree Model : insertion sort

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

Input: <a1,a2,a3>

a1:a2

a1 position in the results will always *before* a2

a1 position in the results will always be *after* a2

<a1,a2,a3>

a2:a3

a1:a3

<a2,a1,a3>

According to the operation of insertion sort, a1 will NOT swap with a2. So in the next iteration, a2 compares with a3.

According to the operation of insertion sort, a1 will swap with a2. So in the next iteration, a1 compares with a3.

Decision Tree Model : insertion sort

Input: <a1,a2,a3>

a1:a2

The output sorted order is <a1, a2, a3>

a2:a3

a1:a3

<a2,a1,a3>

(a1,a2,a3)

a1:a3

(a2,a1,a3)

a2:a3

<a2,a3,a1>

(a1,a3,a2)

(a3,a1,a2)

(a2,a3,a1)

(a3,a2,a1)

The output sorted order is <a3, a2, a1>

The path from the root to a leaf node shows the sequence of key comparisons made to each one of the n! outcomes.

Depending on the values of the input keys, the insertion sort performs different sequence of key comparisons.

a1:a2

< >

a2:a3   a1:a3

< >   < >

(a1,a2,a3)   a1:a3   (a2,a1,a3)   a2:a3

< >   < >

(a1,a3,a2)   (a3,a1,a2)   (a2,a3,a1)   (a3,a2,a1)

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

—Any comparison based sorting algorithm takes $\Omega(n\ log n)$ to sort a list of n   distinct elements in the worst-case."
    – any comparison sort ← model by a decision tree
    – worst-case running time ← the height of decision tree
—Any comparison based sorting algorithm takes $\Omega(n\ log n)$ to sort a list of n distinct elements in the worst-case."
- We want to find a lower bound (Ω) of the **height** of a binary tree that has n! Leaves.
    ◊ What is the minimum height of a binary tree that has n! leaves?

- The binary tree must be a complete tree (recall the definition of complete tree).
- Hence the minimum (lower bound) height is $\theta(\log 2(n!))$.

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

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

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

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

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

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

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

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

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

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

A *nondeterministic polynomial algorithm* is an abstract two-stage procedure that:
- generates a random string purported to solve the problem
- checks whether this solution is correct in polynomial time

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

Why this definition?
- led to development of the rich theory called ―computational complexity"

Example: CNF satisfiability
Problem: Is a boolean expression in its conjunctive normal form (CNF) satisfiable, i.e., are there values of its variables that makes it true?
This problem is in *NP*. Nondeterministic algorithm:
- Guess truth assignment
- Substitute the values into the CNF formula to see if it evaluates to true

Example: $(A \mid \neg B \mid \neg C) \& (A \mid B) \& (\neg B \mid \neg D \mid E) \& (\neg D \mid \neg E)$
Truth assignments:
A B C D E
0 0 0 0 0

. . .

1 1 1 1 1
Checking phase: $O(n)$

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

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

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

6.5 NP-COMPLETE PROBLEMS
A decision problem *D* is <u>*NP*-complete</u> if it's as hard as any problem in *NP,* i.e.,
- *D* is in *NP*
- every problem in *NP* is polynomial-time reducible to *D*

NP problems

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

NP problems

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

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

- Given a boolean formula

$$\left(x_1 \lozenge x_2 \lozenge x_3 \lozenge x_4\right) \lozenge \left(x_5 \lozenge x_6 \lozenge x_7\right) \lozenge x_8 \lozenge \overline{x_9}$$

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

- If the answer is YES with a proof (i.e. an assignment of input value), then we can check the proof in polynomial time (SAT is in NP)
- We may not be able to check the NO answer in polynomial time (Nobody really knows.)
- NP-hard
  - A problem is NP-hard iff an polynomial-time algorithm for it implies a polynomial-time algorithm for every problem in NP
  - NP-hard problems are at least *as hard as* NP problems
- NP-complete
  - A problem is NP-complete if it is NP-hard, and is an element of NP (NP-easy)

- Relationship between decision problems and optimization problems

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

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

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

Examples of NP-complete problems
Vertex cover
- Vertex cover
  - given a graph G=(V,E), find the *smallest* number of vertexes that cover *each edge*
  - Decision problem: is the graph has a vertex cover of size K?
- Independent set
- independent set: a set of vertices in the graph with no edges between *each pair* of nodes.
- given a graph G=(V,E), find the *largest independent set*
- reduction from vertex cover:
  - Set cover
    - given a universal set *U*, and several subsets *S1,...Sn*
    - find the least number of subsets that contains each elements in the universal set

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

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

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

  – C is in NP
  – Any known NP-hard (or complete) problem $\leq_p$ C
  – Thus a proof must show these two being satisfied

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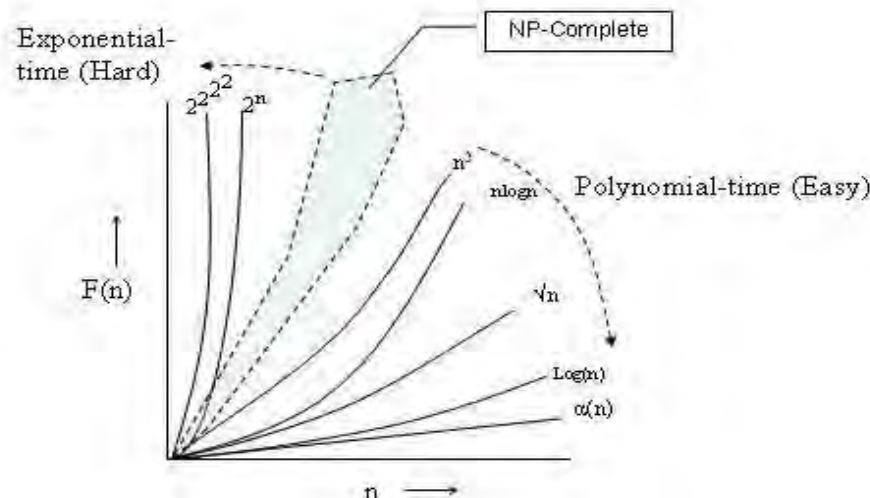
Examples
- Longest path problem: (similar to Shortest path problem, which requires polynomial time) suspected to require exponential time, since there is no known polynomial algorithm.
- Hamiltonian Cycle problem: Traverses all vertices exactly once and form a cycle.

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

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

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

Decision Problems
Represent problem as a decision with a boolean output
  – Easier to solve when comparing to other problems
  – Hence all problems are converted to decision problems.
P = {all decision problems that can be solved in polynomial time}

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

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

Cook's Theorem
- Stephen Cook (Turing award winner) found the first NP-Complete problem, 3SAT.
    - Basically a problem from Logic.
    - Generally described using Boolean formula.
    - A Boolean formula involves AND, OR, NOT operators and some variables.
  - Ex: (x or y) and (x or z), where x, y, z are boolean variables.
    - Problem Definition – Given a boolean formula of m clauses, each containing _n' boolean variables, can you assign some values to these variables so that the formula can be true?
        - Boolean formula: $(x \vee y \vee \hat{z}) \wedge (x \vee y \vee \hat{z})$
    - *Try all sets of solutions. Thus we have exponential set of possible solutions. So it is a NPC problem.*
- Having one definite NP-Complete problem means others can also be proven NP-Complete, using reduction.

Unit 7
COPING WITH LIMITATIONS OF ALGORITHMIC POWER

7.1 Backtracking: n - Queens problem,
7.2 Hamiltonian Circuit Problem,
7.3 Subset –Sum Problem.
7.4 Branch-and-Bound: Assignment Problem,
7.5 Knapsack Problem,
7.6 Traveling Salesperson Problem.
7.7 Approximation Algorithms for NP-Hard Problems – Traveling Salesperson Problem,
    Knapsack Problem

Introduction
Tackling Difficult Combinatorial Problems
   • There are two principal approaches to tackling difficult combinatorial problems (NP-hard problems):
   • Use a strategy that guarantees solving the problem exactly but doesn't guarantee to find a solution in polynomial time
   • Use an approximation algorithm that can find an approximate (sub-optimal) solution in polynomial time

Exact Solution Strategies
   • *exhaustive search* (brute force)
       – useful only for small instances
   • *dynamic programming*
       – applicable to some problems (e.g., the knapsack problem)
   • *backtracking*
       – eliminates some unnecessary cases from consideration
       – yields solutions in reasonable time for many instances but worst case is still exponential
   • *branch-and-bound*
       – further refines the backtracking idea for optimization problems

7.1 Backtracking
   • Suppose you have to make a series of *decisions,* among various *choices,* where
       – You don't have enough information to know what to choose
       – Each decision leads to a new set of choices
       – Some sequence of choices (possibly more than one) may be a solution to your problem

   • Backtracking is a methodical way of trying out various sequences of decisions, until you find one that ―works"

Backtracking : A Scenario

A tree is composed of nodes
Backtracking can be thought of as searching a tree for a particular ―goal‖ leaf node
- Each non-leaf node in a tree is a parent of one or more other nodes (its children)
- Each node in the tree, other than the root, has exactly one parent

The backtracking algorithm
- Backtracking is really quite simple--we ―explore‖ each node, as follows:
- To ―explore‖ node N:
  - 1. If N is a goal node, return ―success‖
  - 2. If N is a leaf node, return ―failure‖
  - 3. For each child C of N,
    - 3.1. Explore C
      - 3.1.1. If C was successful, return ―success‖
  - 4. Return ―failure‖

- Construct the *state-space tree*
  - nodes: partial solutions
  - edges: choices in extending partial solutions

- Explore the state space tree using depth-first search

- ―Prune‖ *nonpromising nodes*
  - dfs stops exploring subtrees rooted at nodes that cannot lead to a solution and backtracks to such a node‘s parent to continue the search

Example:
n-Queens Problem
Place *n* queens on an *n*-by-*n* chess board so that no two of them are in the same row, column, or diagonal
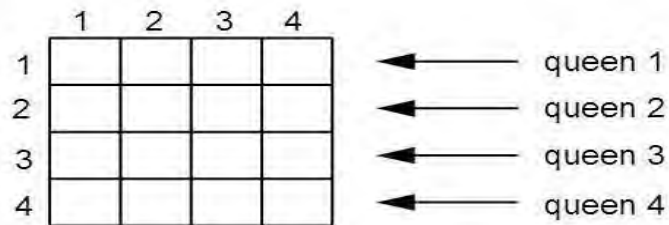
State-Space Tree of the 4-Queens Problem

7.1.1N-Queens Problem:
- The object is to place queens on a chess board in such as way as no queen can capture another one in a single move
    - Recall that a queen can move horz, vert, or diagonally an infinite distance
        - This implies that no two queens can be on the same row, col, or diagonal
    - We usually want to know how many different placements there are

4-Queens
- Lets take a look at the simple problem of placing queens 4 queens on a 4x4 board
- The brute-force solution is to place the first queen, then the second, third, and forth
    - After all are placed we determine if they are placed legally
- There are 16 spots for the first queen, 15 for the second, etc.
    - Leading to 16*15*14*13 = 43,680 different combinations
- Obviously this isn't a good way to solve the problem
- First lets use the fact that no two queens can be in the same col to help us
    - That means we get to place a queen in each col

- So we can place the first queen into the first col, the second into the second, etc.
- This cuts down on the amount of work
  - Now there are 4 spots for the first queen, 4 spots for the second, etc.
    - 4*4*4*4 = 256 different combinations
- However, we can still do better because as we place each queen we can look at the previous queens we have placed to make sure our new queen is not in the same row or diagonal as a previously place queen
- Then we could use a Greedy-like strategy to select the next valid position for each col

- So now what do we do?
- Well, this is very much like solving a maze

  - As you walk though the maze you have to make a series of choices
  - If one of your choices leads to a dead end, you need to back up to the last choice you made and take a different route
    - That is, you need to change one of your earlier selections
  - Eventually you will find your way out of the maze

- **This type of problem is often viewed as a state-space tree**

- – A tree of all the states that the problem can be in
- **We start with an empty board state at the root and try to work our way down to a leaf node**
  - – Leaf nodes are completed boar
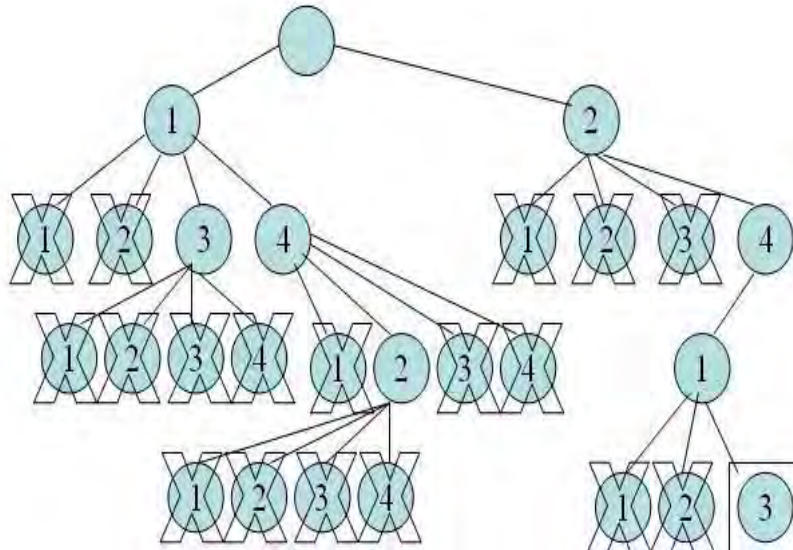
Eight Queen Problem

- The solution is a vector of length 8 (a(1), a(2), a(3), ..., a(8)).

   a(i) corresponds to the column where we should place the i-th queen.
- The solution is to build a partial solution element by element until it is complete.
- We should backtrack in case we reach to a partial solution of length k, that we couldn't expand any more.

Eight Queen Problem: Algorithm

```
putQueen(row) {
  for every position col on the same row
    if position col is available
      place the next queen in position col
    if (row<8)
      putQueen(row+1);
    else   success;
    remove the queen from position col
}
putQueen(row) {
  for every position col on the same row
    if position col is available
      place the next queen in position col
    if (row<8)
      putQueen(row+1);
    else   success;
    remove the queen from position col
}
```

Eight Queen Problem: Implementation
- Define an 8 by 8 array of 1s and 0s to represent the chessboard
- The array is initialized to 1s, and when a queen is put in a position (c,r), board[r][c] is set to zero
- Note that the search space is very huge: 16,772, 216 possibilities.
- Is there a way to reduce search space? Yes Search Pruning.
- We know that for queens:
  each row will have exactly one queen
  each column will have exactly one queen
  each diagonal will have at most one queen
- This will help us to model the chessboard not as a 2-D array, but as a set of rows, columns and diagonals.

7.2 HAMILTONIAN CYCLE
- Hamiltonian Cycle: a cycle that contains every node exactly once
- Problem:Given a graph, does it have a Hamiltonian cycle?

Background
- **NP-complete** problem:
  - Most difficult problems in NP (non- deterministic polynomial time)
- A decision problem *D* is NP-complete if it is complete for NP, meaning that:
  - it is in NP
  - it is NP-hard (every other problem in NP is reducible to it.)
- As they grow large, we are not able to solve them in a reasonable time (polynomial time)

Alternative Definition
- . **NP Problem such as Hamiltonian Cycle** :
  - Cannot be solved in Poly-time
  - Given a solution, easy to verify in poly-time
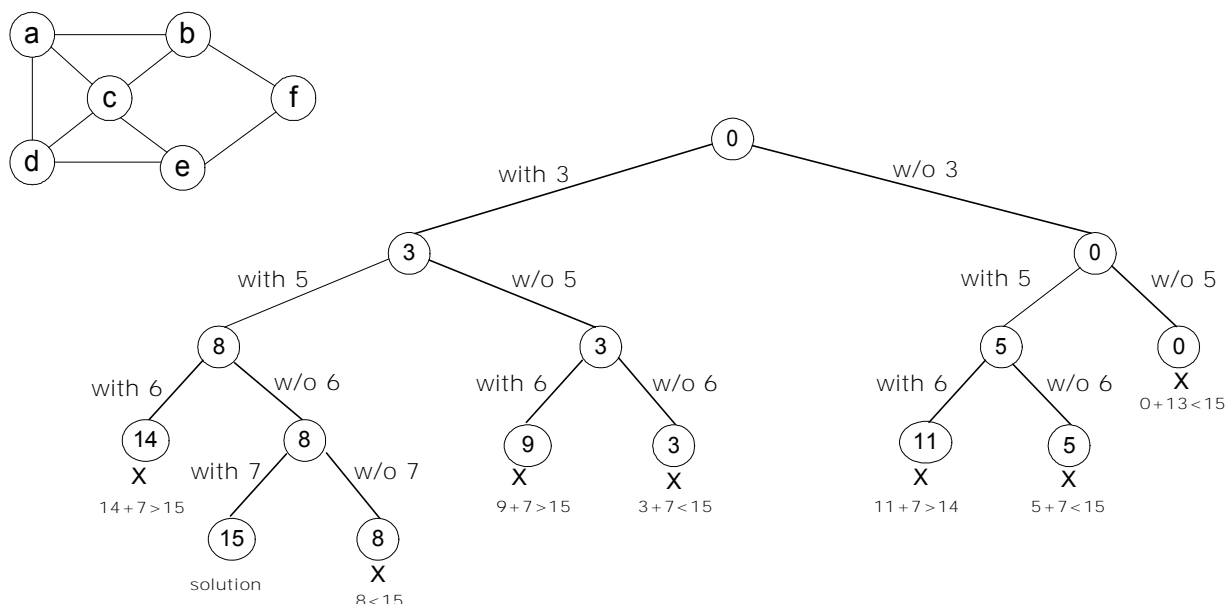
7.3 SUBSET –SUM PROBLEM.

- Problem: **Given *n* positive integers *w*1, … *wn* and a positive integer S. Find all subsets of *w*1, … *wn* that sum to S.**
- Example:
  n=3, S=6, and w1=2, w2=4, w3=6
- Solutions:
   {2,4} and {6}
- **We will assume a binary state space tree.**
- **The nodes at depth 1 are for including (yes, no) item 1, the nodes at depth 2 are for item 2, etc.**
- **The left branch includes *wi*, and the right branch excludes *wi*.**
- **The nodes contain the sum of the weights included so far**

Sum of subset Problem: State SpaceTree for 3 items

The sum of the included integers is stored at the node.
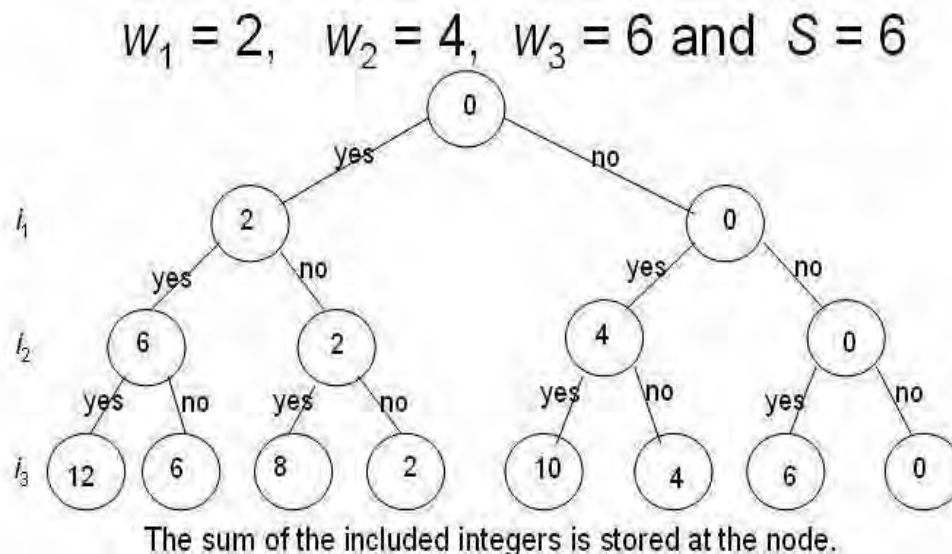
A Depth First Search solution
- Problems can be solved using depth first search of the (implicit) state space tree.
- Each node will save its depth and its (possibly partial) current solution
- DFS can check whether node v is a leaf.
  - If it is a leaf then check if the current solution satisfies the constraints
  - Code can be added to find the optimal solution

A DFS solution
- Such a DFS algorithm will be very slow.
- It does not check for every solution state (node) whether a solution has been reached, or whether a *partial* solution can lead to a *feasible* solution
- Is there a more efficient solution?

Backtracking solution to Sum of Subsets
- Definition: We call a node *nonpromising* if it cannot lead to a feasible (or optimal)

solution, otherwise it is *promising*
- Main idea: Backtracking consists of doing a DFS of the state space tree, checking whether each node is promising and if the node is nonpromising backtracking to the node's parent
- The state space tree consisting of expanded nodes only is called the *pruned state space tree*
- The following slide shows the pruned state space tree for the sum of subsets example
- There are only 15 nodes in the pruned state space tree
- The full state space tree has 31 nodes

Pruned State Space Tree (find all solutions)
$w_1 = 3,\ w_2 = 4,\ w_3 = 5,\ w_4 = 6;\ S = 13$

Sum of subsets problem
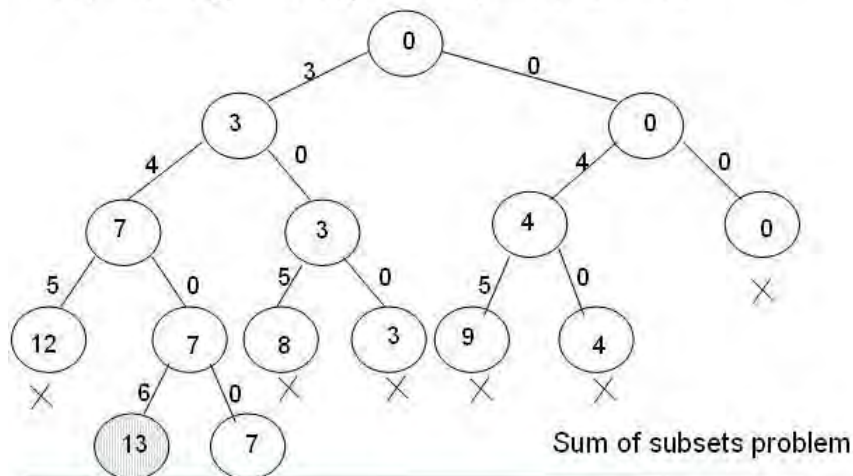
Backtracking algorithm
void *checknode* (node *v*) {
node *u*

if (*promising* (*v*))
      if (*aSolutionAt*(*v*))
            write the solution
      else //expand the node
            for (each child *u* of *v*)
                  checknode (*u*)

Checknode
- Checknode uses the functions:
  - *promising*(*v*) which checks that the partial solution represented by *v* can lead to the required solution
  - *aSolutionAt*(*v*) which checks whether the partial solution represented by node *v* solves the problem.

Sum of subsets – when is a node "promising"?
- Consider a node at depth i
- *weightSoFar* = weight of node, i.e., sum of numbers included in partial solution node represents
- *totalPossibleLeft* = weight of the remaining items i+1 to n (for a node at depth i)
- A node at depth i is non-promising
  if   (*weightSoFar* + *totalPossibleLeft* < S)
  or (*weightSoFar*   + w[i+1] > S)
- To be able to use this —promising function" the w_i must be sorted in non-decreasing order

A Pruned State Space Tree

$W_1 = 3,\ W_2 = 4,\ W_3 = 5,\ W_4 = 6;\ S = 13$

X - backtrack
Nodes numbered in "call" order
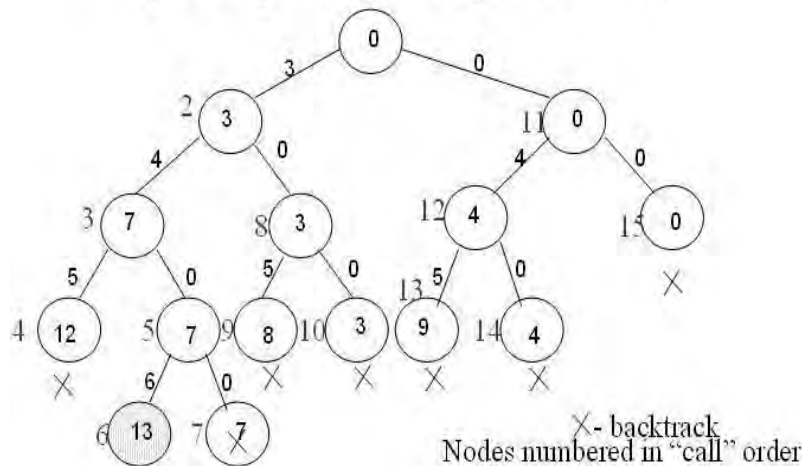
sumOfSubsets (*i*, *weightSoFar*, *totalPossibleLeft*)
1) **if** (promising (*i*))   //may lead to solution
2)   **then if** (*weightSoFar* == S)
3)   **then** print *include*[1] to *include*[*i*]   //found solution
4)   **else**   //expand the node when *weightSoFar* < S
5)   include [*i* + 1] = "yes"   //try including
6)   sumOfSubsets (*i* + 1,
                         weightSoFar + w[*i* + 1],
                    totalPossibleLeft - w[*i* + 1])
7)   include [*i* + 1] = "no"   //try excluding
8)   sumOfSubsets (*i* + 1, *weightSoFar* ,
                          totalPossibleLeft - w[*i* + 1])
boolean promising (*i*)
1) return (*weightSoFar* + *totalPossibleLeft* ≥ S) &&
          (*weightSoFar* == S || *weightSoFar* + w[*i* + 1] ≤ S)
Prints all solutions!

Initial call sum Of Subsets (0, 0,   ) $\sum\limits_{i=1}^{n} w_i$

7.4 Branch and Bound Searching Strategies

Feasible Solution vs. Optimal Solution
- DFS, BFS, hill climbing and best-first search can be used to solve some searching problem for searching a feasible solution.
- However, they cannot be used to solve the optimization problems for searching an (the) optimal solution.

The Branch-and-bound strategy
- This strategy can be used to solve optimization problems without an exhaustive search in the average case.
- 2 mechanisms:
    - A mechanism to generate branches when searching the solution space
    - A mechanism to generate a bound so that many braches can be terminated
- It is efficient in the average case because many branches can be terminated very early.
- Although it is usually very efficient, a very large tree may be generated in the worst case.
- Many NP-hard problem can be solved by B&B efficiently in the average case; however, the worst case time complexity is still exponential.

Bounding
- A bound on a node is a guarantee that any solution obtained from expanding the node will be:
    - Greater than some number (lower bound)
    - Or less than some number (upper bound)
- If we are looking for a minimal optimal, as we are in weighted graph coloring, then we need a lower bound
    - For example, if the best solution we have found so far has a cost of 12 and the lower bound on a node is 15 then there is no point in expanding the node
        - The node cannot lead to anything better than a 15
- We can compute a lower bound for weighted graph color in the following way:
    - The actual cost of getting to the node
    - Plus a bound on the future cost
        - Min weight color * number of nodes still to color
            - That is, the future cost cannot be any better than this
- Recall that we could either perform a depth-first or a breadth-first search
    - Without bounding, it didn't matter which one we used because we had to expand the entire tree to find the optimal solution
    - Does it matter with bounding?
        - Hint: think about when you can prune via bounding
- We prune (via bounding) when:
    (currentBestSolutionCost <= nodeBound)
- This tells us that we get more pruning if:
    - The currentBestSolution is low
    - And the nodeBound is high
- So we want to find a low solution quickly and we want the highest possible lower bound

     – One has to factor in the extra computation cost of computing higher lower bounds vs. the expected pruning savings

The Assignment Problem

- In many business situations, management needs to assign - personnel to jobs, - jobs to machines, - machines to job locations, or - salespersons to territories.
- Consider the situation of assigning n jobs to n machines.
- When a job i (=1,2,....,n) is assigned to machine j (=1,2, ...n) that incurs a cost Cij.
- The objective is to assign the jobs to machines at the least possible total cost.
- This situation is a special case of the Transportation Model And it is known as the *assignment problem*.
- Here, jobs represent —sources" and machines represent —destinations."
- The supply available at each source is 1 unit And demand at each destination is 1 unit.

		Machine				
		1	2	....	n	Source
	1	C11	C12	....	C1n	1
	2	C21	C22	....	C2n	1
Job	.	.	.	.	.	.
	.	.	.	.	.	.
	.	.	.	.	.	.
	n	Cn1	Cn2	....	Cnn	1
Destination		1	1	....	1	

The assignment model can be expressed mathematically as follows:

Xij=   0, if the job j is not assigned to machine i
           1, if the job j is assigned to machine i

$$\text{Min} \quad \sum_{i=1}^{r} \sum_{j=1}^{n} C_{ij} X_{ij}$$

(Sum of assignments from a source should be exactly equal to 1):

$$\sum_{j=1}^{n} X_{ij} = 1 \quad \text{For } i=1,2,...,r$$

(Sum of assignments to a destination should be equal to the demanded quantity by that destination):

$$\sum_{i=1}^{n} X_{ij} = 1 \quad \text{For } j=1,2,...,n$$

(Quantities to be assigned can be either 0 or 1):

$$X_{ij} = 0 \text{ or } 1 \quad \text{For all i and j.}$$

The Assignment Problem Example
- Ballston Electronics manufactures small electrical devices.
- Products are manufactured on five different assembly lines (1,2,3,4,5).
- When manufacturing is finished, products are transported from the assembly lines to one of the five different inspection areas (A,B,C,D,E).
- Transporting products from five assembly lines to five inspection areas requires different times (in minutes)

| | Inspection Area | | | | |
Assembly Line	A	B	C	D	E
1	10	4	6	10	12
2	11	7	7	9	14
3	13	8	12	14	15
4	14	16	13	17	17
5	19	11	17	20	19

Under current arrangement, assignment of inspection areas to the assembly lines are 1 to A, 2 to B, 3 to C, 4 to D, and 5 to E. This arrangement requires 10+7+12+17+19 = 65 man minutes.
- Management would like to determine whether some other assignment of production lines to inspection areas may result in less cost.
- This is a typical assignment problem. $n = 5$ And each assembly line is assigned to each inspection area.
- It would be easy to solve such a problem when n is 5, but when n is large all possible alternative solutions are n!, this becomes a hard problem.
- Assignment problem can be either formulated as a linear programming model, or it can be formulated as a transportation model.
- However, An algorithm known as *Hungarian Method* has proven to be a quick and efficient way to solve such problems.

Assignment Problem Revisited:
Select one element in each row of the cost matrix *C* so that:
- no two selected elements are in the same column
- the sum is minimized

Example

	Job 1	Job 2	Job 3	Job 4
Person a	9	2	7	8
Person b	6	4	3	7
Person c	5	8	1	8
Person d	7	6	9	4

Lower bound: Any solution to this problem will have total cost
          at least: 2 + 3 + 1 + 4 (or 5 + 2 + 1 + **4)**

Example: First two levels of the state-space tree
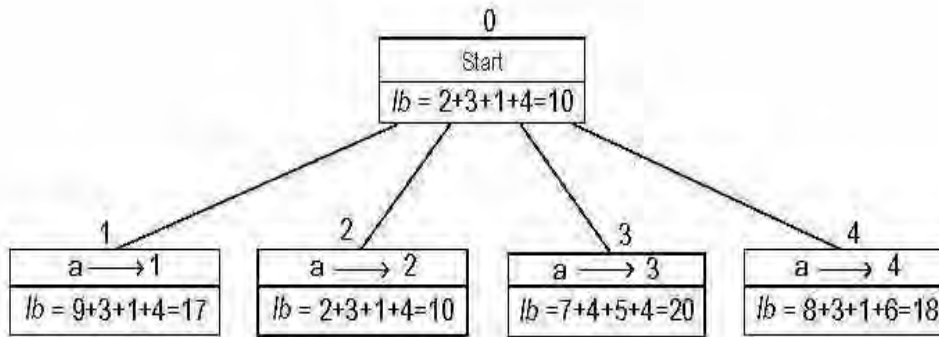
Figure 11.5 Levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated. A node's fields indicate the job number assigned to person a and the lower bound value, lb, for this node.
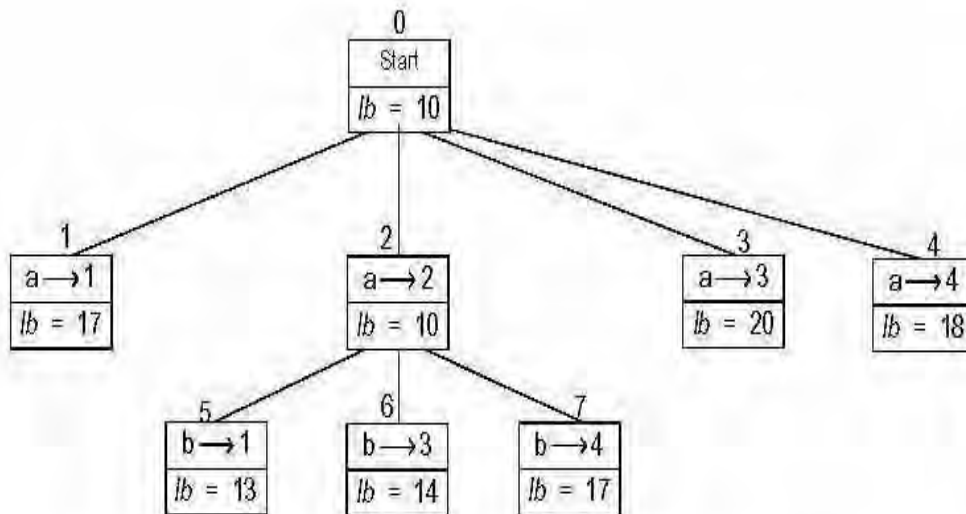
Figure 11.6 Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm

Example: Complete state-space tree
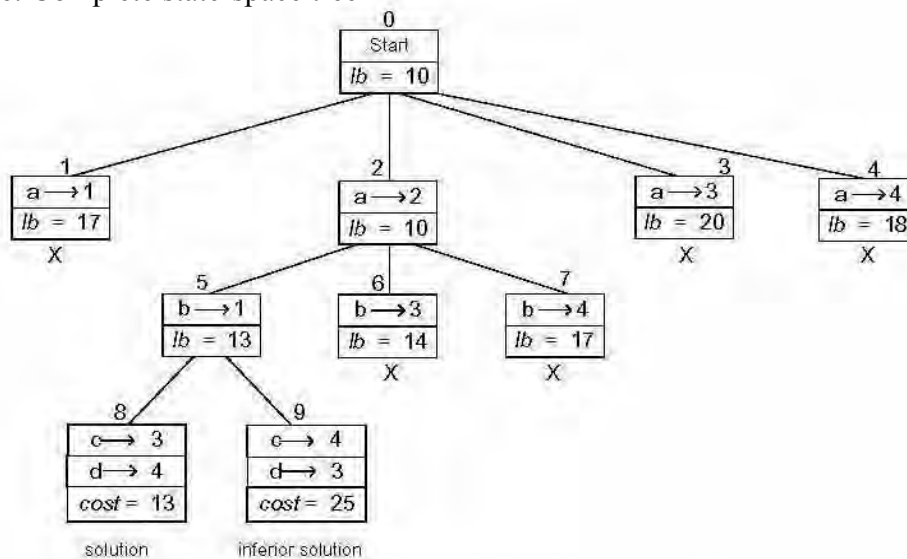
Figure 11.7 Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm

7.5 TRAVELING SALESPERSON PROBLEM

- This is a classic CS problem
- Given a graph (cities), and weights on the edges (distances) find a minimum weight tour of the cities
    - Start in a particular city
    - Visit all other cities (*exactly* once each)
    - Return to the starting city
- Cannot be done by brute-force as this is worst-case exponential or worse running time
    - So we will look to backtracking with pruning to make it run in a reasonable amount of time in most cases
- We will build our state space by:
    - Having our children be all the potential cities we can go to next
    - Having the depth of the tree be equal to the number of cities in the graph
        - we need to visit each city exactly once
- So given a fully connected set of 5 nodes we have the following state space
    - only partially completed

Traveling Salesperson Problem
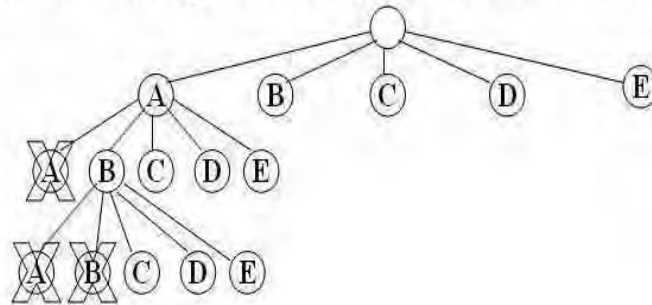
- Now we need to add bounding to this problem
  - It is a minimization problem so we need to find a lower bound
- We can use:
  - The current cost of getting to the node plus
  - An underestimate of the future cost of going through the rest of the cities
    - The obvious choice is to find the minimum weight edge in the graph and multiply that edge weight by the number of remaining nodes to travel through
- As an example assume we have the given adjacency matrix
- If we started at node A and have just traveled to node B then we need to compute the bound for node B
  - Cost 14 to get from A to B
  - Minimum weight in matrix is 2 times 4 more legs to go to get back to node A = 8
  - For a grand total of 14 + 8 = 22

0	14	4	10	20	
14	0	7	8	7	
4	5	0	7	16	
11	7	9	0	2	
18	7	17	4	0	

Recall that if we can make the lower bound higher then we will get more pruning
- Note that in order to complete the tour we need to leave node B, C, D, and E
  - The min edge we can take leaving B is min(14, 7, 8, 7) = 7
  - Similarly, C=4, D=2, E=4

This implies that at best the future underestimate can be 7+4+2+4=17
- 17 + current cost of 14 = 31
  - This is much higher than 8 + 14 = 22

0	14	4	10	20	
14	0	7	8	7	
4	5	0	7	16	
11	7	9	0	2	
18	7	17	4	0	

Example: Traveling Salesman Problem

7.6 THE 0/1 KNAPSACK PROBLEM

- Positive integer $P_1, P_2, \ldots, P_n$ (profit)
  $W_1, W_2, \ldots, W_n$ (weight)
  M (capacity)

maximize $\sum\limits_{i=1}^{n} P_i X_i$

subject to $\sum\limits_{i=1}^{n} W_i X_i \leq M$ $X_i = 0$ or $1, i = 1, \ldots, n.$

The problem is modified:

minimize $-\sum\limits_{i=1}^{n} P_i X_i$
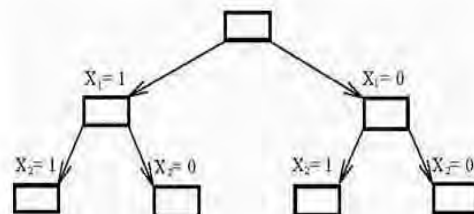
The Branching Mechanism in the Branch-and-Bound Strategy to
Solve 0/1 Knapsack Problem.

How to find the upper bound?

- Ans: by quickly finding a feasible solution in a greedy manner: starting from the smallest available i, scanning towards the largest i's until M is exceeded. The upper bound can be calculated.

The 0/1 knapsack problem

- E.g. n = 6, M = 34

i	1	2	3	4	5	6
P_i	6	10	4	5	6	4
W_i	10	19	8	10	12	8

$$(P_i/W_i \geq P_{i+1}/W_{i+1})$$

- A feasible solution: X1 = 1, X2 = 1, X3 = 0, X4 = 0, X5 = 0, X6 = 0
  -(P1+P2) = -16 (upper bound)
  Any solution higher than -16 can not be an optimal solution.

How to find the lower bound?

- Ans: by relaxing our restriction from Xi = 0 or 1 to $0 \leq Xi \leq 1$ (knapsack problem)

$$\text{Let} \quad - \sum_{i=1}^{n} P_i X_i \quad \text{be an optimal}$$

solution for 0/1 knapsack

$$\text{problem and} \quad - \sum_{i=1}^{n} P_i X_i' \quad \text{be an}$$

optimal solution for fractional knapsack problem. Let

$$Y = - \sum_{i=1}^{n} P_i X_i, \quad Y' = - \sum_{i=1}^{n} P_i X_i'.$$

$$? \quad Y' \leq Y$$

7.7 APPROXIMATION APPROACH

Apply a fast (i.e., a polynomial-time) approximation algorithm to get a solution that is not necessarily optimal but hopefully close to it
Accuracy measures:
accuracy ratio of an approximate solution *sa*
  r(sa) = f(sa) / f(s*) for minimization problems
  r(sa) = f(s*) / f(sa) for maximization problems
where f(sa) and f(s*) are values of the objective function f for the approximate solution *sa* and

actual optimal solution $s*$

performance ratio of the algorithm A the lowest upper bound of $r(sa)$ on all instances

Nearest-Neighbor Algorithm for TSP

Starting at some city, always go to the nearest unvisited city, and, after visiting all the cities, return to the starting one

Note: Nearest-neighbor tour may depend on the starting city

$s_a: A - B - C - D - A$ of length 10
$s^*: A - B - D - C - A$ of length 8
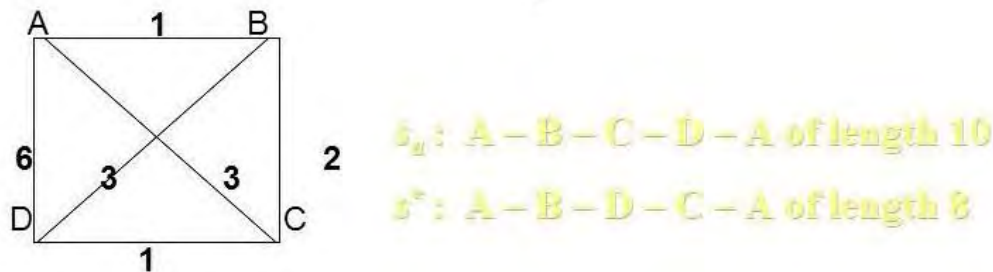
Accuracy: RA = ∞ (unbounded above) – make the length of AD
          arbitrarily large in the above example

Multifragment-Heuristic Algorithm

Stage 1: Sort the edges in nondecreasing order of weights. Initialize the set of tour edges to be constructed to empty set

Stage 2: Add next edge on the sorted list to the tour, skipping those whose addition would've created a vertex of degree 3 or a cycle of length less than n. Repeat this step until a tour of length n is obtained

Note: RA = ∞, but this algorithm tends to produce better tours than the nearest-neighbor algorithm

Twice-Around-the-Tree Algorithm

Stage 1: Construct a minimum spanning tree of the graph(e.g., by Prim's or Kruskal's algorithm)

Stage 2: Starting at an arbitrary vertex, create a path that goes twice around the tree and returns to the same vertex

Stage 3: Create a tour from the circuit constructed in Stage 2 by making shortcuts to avoid visiting intermediate vertices more than once

Note: RA = ∞ for general instances, but this algorithm tends to produce better tours than the nearest-neighbor algorithm

Christofides Algorithm

Stage 1: Construct a minimum spanning tree of the graph

Stage 2: Add edges of a minimum-weight matching of all the odd vertices in the minimum spanning tree

Stage 3: Find an Eulerian circuit of the multigraph obtained in Stage 2

Stage 3: Create a tour from the path constructed in Stage 2 by making shortcuts to avoid visiting intermediate vertices more than once

RA = ∞ for general instances, but it tends to produce better tours than the twice-around-the-minimum-tree alg.

Euclidean Instances

<u>Theorem</u> If $P \neq NP$, there exists no approximation algorithm for TSP with a finite performance ratio.

<u>Definition</u> An instance of TSP is called *Euclidean,* if its distances satisfy two conditions:
1. *symmetry* $d[i, j] = d[j, i]$ for any pair of cities i and j
2. *triangle inequality* $d[i, j] \leq d[i, k] + d[k, j]$ for any cities i, j, k

For Euclidean instances:

approx. tour length / optimal tour length $\leq 0.5(\lceil \log 2\ n \rceil + 1)$ for nearest neighbor and multifragment heuristic;
approx. tour length / optimal tour length ≤ 2 for twice-around-the-tree;
approx. tour length / optimal tour length ≤ 1.5 for Christofides

Local Search Heuristics for TSP

Start with some initial tour (e.g., nearest neighbor). On each iteration, explore the current tour's neighborhood by exchanging a few edges in it. If the new tour is shorter, make it the current tour; otherwise consider another edge change. If no change yields a shorter tour, the current tour is returned as the output.

Greedy Algorithm for Knapsack Problem

Step 1: Order the items in decreasing order of relative values:
$$v_1/w_1 \geq \ldots \geq v_n/w_n$$
Step 2: Select the items in this order skipping those that don't fit into the knapsack

Example: The knapsack's capacity is 16

item	weight	value	v/w
1	2	$40	20
2	5	$30	6
3	10	$50	5
4	5	$10	2

Accuracy
- R_A is unbounded (e.g., $n = 2$, $C = m$, $w_1=1$, $v_1=2$, $w_2=m$, $v_2=m$)
- yields exact solutions for the continuous version

Approximation Scheme for Knapsack Problem

Step 1: Order the items in decreasing order of relative values:
$$v_1/w_1 \geq \ldots \geq v_n/w_n$$
Step 2: For a given integer parameter k, $0 \leq k \leq n$, generate all subsets of k items or less and for each of those that fit the knapsack, add the remaining items in decreasing order of their value to weight ratios
Step 3: Find the most valuable subset among the subsets generated in Step 2 and return it as the

algorithm's output

Bin Packing Problem: First-Fit Algorithm

First-Fit (FF) Algorithm: Consider the items in the order given and place each item in the first available bin with enough room for it; if there are no such bins, start a new one

Example: $n = 4$, $s_1 = 0.4$, $s_2 = 0.2$, $s_3 = 0.6$, $s_4 = 0.7$

Accuracy
- Number of extra bins never exceeds optimal by more than 70% (i.e., $R_A \leq 1.7$)
- Empirical average-case behavior is much better. (In one experiment with 128,000 bins, the relative error was found to be no more than 2%.)

Bin Packing: First-Fit Decreasing Algorithm

First-Fit Decreasing (FFD) Algorithm: Sort the items in decreasing order (i.e., from the largest to the smallest). Then proceed as above by placing an item in the first bin in which it fits and starting a new bin if there are no such bins

Example: $n = 4$, $s_1 = 0.4$, $s_2 = 0.2$, $s_3 = 0.6$, $s_4 = 0.7$

Accuracy
- Number of extra bins never exceeds optimal by more than 50% (i.e., $R_A \leq 1.5$)
- Empirical average-case behavior is much better, too

UNIT-8

PRAM ALGORITHMS

8.1 Introduction,

8.2 Computational Model,

8.3 Parallel Algorithms for Prefix Computation,

 8.4 List Ranking, and Graph Problems,

8.1 INTRODUCTION

In this section a few basic facts about parallel processing in general. One very basic fact that applies to parallel computation, regardless of how it is implemented, is the following:

Suppose the fastest sequential algorithm for doing a computation with parameter n has execution time of T(n). Then the fastest parallel algorithm with m processors (each comparable to that of the sequential computer) has execution time ¸ T(n)/m. The idea here is: If you could find a faster parallel algorithm, you could execute it sequentially by having a sequential computer *simulate* parallelism and get a faster sequential algorithm. This would contradict the fact that the given sequential algorithm is the fastest possible. We are making the assumption that the *cost* of simulating parallel algorithms by sequential ones is negligible. This claim is called the ―Principle of Unitary Speedup". As usual, the parameter *n* represents the relative size of the instance of the problem being considered. For instance, if the problem was that of sorting, *n* might be the number of items to be sorted and T(*n*) would be $O(n \lg n)$ for a sorting algorithm based upon comparisons.

As simple as this claim is, it is a bit controversial. It makes the tacit assumption that the algorithm in question is *deterministic*. In other words, the algorithm is like the usual idea of a computer program — it performs calculations and makes decisions based on the results of these calculations.

There is an interesting area of the theory of algorithms in which statement is *not necessarily true* — this is the theory of *randomized* algorithms. Here, a solution to a problem may involve making random ―gesses" at some stage of the calculation. In this case, the parallel algorithm using *m* processors can run faster than $m \times$ the speed of the sequential algorithm (―Super-unitary speedup"). This phenomenon occurs in certain problems in which random search is used, and

most guesses at a solution quickly lead to a valid solution, but there are a few guesses that execute for a long time without producing any concrete results.

The expected execution-time of a *single (sequential) attempt* to find a solution is the *average* of all of these times, or 10.99 time-units.

8.2. MODELS OF PARALLEL COMPUTATION

If we attack this problem with a parallel computer that has 2 processors that try *distinct* possibilities, the expected time (and even the *worst-case* time) is 1 unit, since at least one of these two distinct possibilities will be a *fast solution*, consequently, see a *super-unitary* speedup in the parallel algorithm. In other words the *expected*1 running-time of the algorithm is divided by > 10, which is much greater than the ratio of processors.

The opponents of the concept of super-unitary speedups (including the author) would argue that the original sequential algorithm was not *optimal* — and that the optimal sequential algorithm would have attempted two possible solutions with two distinct processes, run *concurrently*). A sequential algorithm that created two processes would have a running time of 2 units. The speedup that results by going to the sample parallel algorithm is 2, which is exactly equal to the ratio of processors. Thus, by modifying the sequential algorithm used, the validity is restored.

In this section, consider some simple algorithms that can be implemented when a SIMD computer in which every processor can access common RAM. In general, a computer in which many processors can access common RAM in a single program-step is called the **PRAM** model of computer. This is one of the oldest models of parallel processing to be considered, although there have never been any large parallel computers that implement it. The PRAM model is a kind of mathematical idealization of a parallel computer that eliminates many of the low-level details and allows a researcher to concentrate on the ―purely parallel‖ aspects of a problem.

Traditionally, the PRAM model of computation has been regarded as more of theoretical than practical interest. This is due to the fact that it requires large numbers of processors to be physically connected to the same memory location. The two examples cited above (the Sequent and the Encore Multimax) aren't an exception to this statement: they only have small numbers of processors. Several researchers are exploring the possibility of physically realizing of PRAM computers using optical interfaces.

There exist several different models of program control. Flynn listed several basic schemes:

SIMD: Single Instruction Multiple Data. In this model the processors are controlled by a program whose instructions are applied to all of them simultaneously (with certain qualifications). Assume that each of the processors has a unique number that is ―known‖ to the processor in the sense that instructions to the parallel computer can refer to processor numbers. An example of this type of machine is: 1Recall that the expected running-time of an algorithm like the one in the example is the average of actual running times, weighted by probabilities that these running times occur.

MIMD: Multiple Instruction Multiple Data. In this model processors can each have independent programs that are read from the common RAM. This model is widely used in several settings:

The data-movement and communications problems that occur in all parallel computation are more significant here because the instructions to the processors as well as the data must be passed between the common memory and the processors. Due to these data-movement problems, commercial MIMD computers tend to have a relatively small number of processors (¼ 20). In general, it is easier to program a MIMD machine if one is only interested in a very limited form of parallelism—namely the formation of processes. Conventional operating systems like UNIX form separate processes to carry out many functions, and these processes *really* execute in parallel on commercially-available MIMD machines. It follows that, with one of these MIMD machines, one can reap *some* of the benefits of parallelism without explicitly doing any parallel programming. For this reason, most of the parallel computers in commercial use today tend to be MIMD machines, run as general-purpose computers. On the surface, it would appear that MIMD machine are strictly more powerful than SIMD machines with the same number of processors. Interestingly enough, this is not the case—it turns out that SIMD machines are more suited to performing computations with a very *regular* structure. MIMD machines are not as suited to solving such problems because their processors must be *precisely synchronized* to implement certain algorithms. Pure MIMD machines have no hardware features to guarantee synchronization of processors. In general, it is not enough to simply load multiple copies of a program into all of the processors and to start all of these copies at the same time. In fact many such computers have hardware features that tend to *destroy* synchronization, once it

has been achieved. For instance, the manner in which memory is accessed in the Sequent Symmetry series, generally causes processes to run at different rates even if they are synchronized at some time. Many Sequent's even have processors that run at different clock-rates.

8.2.1 MODELS OF PARALLEL COMPUTATION

Three other terms that fill out this list are:

SISD Single Instruction, Single Data. This is nothing but conventional *sequential computing*.

MISD This case is often compared to computation that uses *Systolic Arrays*. These are arrays of processors that are developed to solve specific problems — usually on a single VLSI chip. A clock coordinates the data movement operations of all of the processors, and output from some processors are pipelined into other processors. The term ―Systolic‖ comes from an analogy with an animal's circulatory system — the data in the systolic array playing the part of the blood in the circulatory system. In a manner of speaking, one can think of the different processors in a systolic array as constituting ―multiple processors‖ that work on one set of

(pipelined) data.

SIMD-MIMD Hybrids This is a new category of parallel computer that is becoming very significant. These machines are also called SAMD machines (Synchronous-Asynchronous Multiple Data). The first announced commercial SAMD computer is the new Connection Machine, the CM-5.

This is essentially a MIMD computer with hardware features to allow:

• Precise synchronization of processes to be easily achieved.

• Synchronization of processors to be maintained with little or no overhead,

Once it has been achieved (assuming that the processors are all executing the same instructions in corresponding program steps). It differs from pure MIMD machines in that the hardware maintains a uniform ―heartbeat‖ throughout the machine, so that when the same program is run on all processors, and all copies of this program are started at the same time, it is possible to the execution of all copies to be kept in lock-step with essentially no overhead. Such computers allow efficient execution of MIMD and SIMD programs.

8.3 PARALLEL ALGORITHMS FOR PREFIX COMPUTATION

PRAM Recursive Prefix Sum Algorithm
Input: Array of ($x1$, $x2$, ..., xn) elements, $n = 2k$
Output: Prefix sums si, $1 < i < n$
begin
if n = 1 **then** $s1 = x1$; **exit**
for $1 < i < n/2$ **pardo**
$yi := x2i–1 + x2i$
Recursively compute prefix sums of y and store in z
for $1 < i < n$ **pardo**
if i is even **then** $si := zi/2$
if $i > 1$ is odd **then** $si := z(i–1)/2 + xi$
if $i = 1$ **then** $s1 := x1$

Theorem: An algorithm that runs in T time on the p-processor priority CRCW PRAM can be simulated by EREW PRAM to run in O(T log
p) time. A concurrent read or write of an p-processor CRCW PRAM can be implemented on a p processor EREW PRAM to execute in O(log p) time
$Q1,…,Qp$ CRCW processors, such that Qi has to read (write) $M[ji]$
$P1,…,Pp$ EREW processors
$M1,…,Mp$ denote shared memory locations for special use Pi stores $<ji,i>$ in Mi
Sort pairs in lexicographically non-decreasing order in O(log p) time using EREW merge sort algorithm
Pick representative from each block of pairs that have same firs component in O(1) time
 Representative Pi reads (writes) from $M[k]$ with $<k,\_>$ in Mi and copies data to each M in the block in O(log p) time using EREW segmented parallel prefix algorithm
 Pi reads data from Mi

 For n x n matrix M with non-negative integer coefficients, define M and give an algorithm for computing m. prove that M cnan be computed from n x n matrix M in O(log n)time using CRCW PRAM processors for any fixed € > 0

Consider $n{\times}n$ matrix multiplication $C = A B$ using $n3$ processors.
 Each element of C
 $cij = \Sigma k{=}1..n\ aik\ bkj$
can be computed on the CREW PRAM in parallel using n processors in O(log n) time All cij can be computed using $n3$ processors in O(log n)

Matrix multiply with $n3$ processors (i,j,l)
Each processor (i,j,l) executes:
Input: $n{\times}n$ matrices A and B, $n = 2k$
Output: $C = A B$
begin
$C'[i,j,l] := A[i,l]B[l,j]$
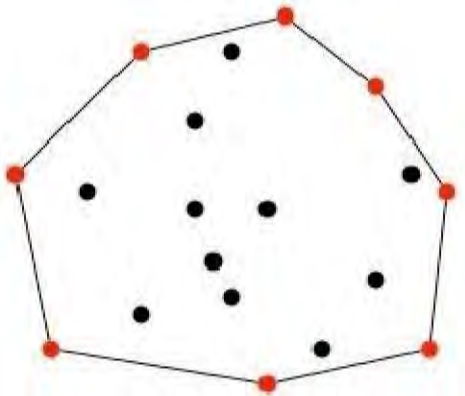**for** $h = 1$ **to** log n **do**

if $i < n/2h$ **then**
$C'[i,j,l] := C'[i,j,2l-1] + C'[i,j,2l]$
if $l = 1$ **then**
$C[i,j] := C'[i,j,1]$
End

8.4 LIST RANKING AND GRAPH PROBLEMS.
Convex Hull Problem
 The *planar convex hull* of a set of points $S = \{p1, p2, …, pn\}$ of $pi = (x,y)$ coordinates is the smallest convex polygon that encompasses all points S on the *x-y* plane

Preprocess the points by sorting in O(log n) parallel time
(pipelined merge sort), such that $x(p1) < x(p2) < … < x(pn)$
Remove duplicates $x(pi) = x(pj)$ (for UCH and LCH)

Divide-and-conquer:
1. Split S into $S1$ $S2$ and recursively compute the UCH of $S1$ and $S2$
2. Combine UCH($S1$) and UCH($S2$) by computing the upper common tangent in O(1) time to form UCH(S) Repeat to compute the LCH
 Parallel time (assuming $p = O(n)$ processors)
$T(n) = T(n/2) + O(1)$ gives
$T(n) = O(\log n)$

Unit 1

1. **Give an algorithm for selection sort. If C(n) denotes the no of times the algorithm**
1. **is executed   (n denotes input size),obtain an expression for C(n)**

 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] an

Performance Analysis of the selection sort algorithm:

The input's size is given by the number of elements *n*.

The algorithm's basic operation is the key comparison $A[j]<A[min]$. The number of

times it is executed depends only on the array's size and is given by

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n - 1 - i).$$
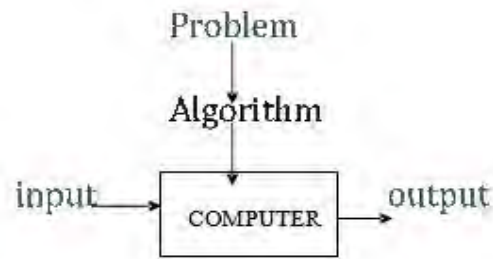
Thus, selection sort is a $O(n^2)$ algorithm on all inputs. The number of key swaps is
only $O(n)$ or, more precisely, *n*-1 (one for each repetition of the *i* loop).This property
distinguishes selection sort positively from many other sorting algorithms.

2. **With the help of a flowchart, explain the various stages of algorithm and design**
   process

An algorithm is finite set of instructions that is followed, accomplishes a particular
task. In   addition, all algorithms must satisfy the following criteria:

1. Input. Zero or more quantities are externally supplied.

2. Output. At least one quantity is produced.

3. Definiteness. Each instruction is clear and produced.

4. Finiteness. If we trace out the instruction of an algorithm, then for all cases, the

    algorithm terminates after a finite number of steps.

5. Effectiveness. Every instruction must be very basic so that it can be carried
    out, in principal, by a person using only pencil and paper. It is not enough that each

operation be definite as in criterion 3; it also must be feasible.

3. Write an algorithm which computes the no of binary digits in the binary representation of n.

ALGORITHM BinRec (n)

//Input: A positive decimal integer n

//Output: The number of binary digits in n's binary representation

if n = = 1

return 1

else

return BinRec ($\lfloor$ n/2 $\rfloor$) + 1

Analysis:

1. Input size: given number = n

2. Basic operation: addition

3. NO best, worst, average cases.

4. Let A (n) denotes number of additions.

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

A (1) = 0   initial condition

Where: A ($\lfloor$ n/2 $\rfloor$) : to compute BinRec ($\lfloor$ n/2 $\rfloor$)

 1 : to increase the returned value by 1

5. Solve the recurrence:

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

Assume n = 2^k (smoothness rule)

$A(2^k) = A(2^{k-1}) + 1$ for $k > 0$; $A(20) = 0$

Solving using "Backward substitution method":

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

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

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

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

$= A(2^{k-3}) + 3$

…

In the ith recursion, we have

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

When $i = k$, we have

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

Since $A(2^0) = 0$

$A(2^k) = k$

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

$A(n) = \log_2 n$

$A(n)$ $\Theta(\log n)$

4. **Explain the worst-case, best-case and average case efficiencies of an algorithm with help of an example.**

   Worst-case, Best-case, Average case efficiencies

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

- **Worst-case efficiency:** Efficiency (number of times the basic operation will be executed) **for the worst case input of size n.** *i.e.* The algorithm runs the longest among all possible inputs of size n.
- **Best-case efficiency:** Efficiency (number of times the basic operation will be executed) **for the best case input of size n. *i.e.*** The algorithm runs the fastest among all possible inputs of size n.
- **Average-case efficiency:** Average time taken (number of times the basic operation will be executed) **to solve all the possible instances (random) of the input.** NOTE: NOT the average of worst and best case

5. **Explain the general plan for analyzing the efficiency of a recursive algorithm.**

   General plan for analyzing efficiency of recursive algorithms:

1. Decide on parameter n indicating **input size**

2. Identify algorithm's **basic operation**

3. Check whether the number of times the basic operation is executed depends only on the input size n. If it also depends on the type of input, investigate **worst, average, and best case** efficiency separately.

4. Set up **recurrence relation**, with an appropriate initial condition, for the number of times the algorithm's basic operation is executed.

5. **Solve** the recurrence.

Example: Factorial function

ALGORITHM Factorial (n)

//Computes n! recursively

//Input: A nonnegative integer n

//Output: The value of n!

if n = = 0

return 1

else

return Factorial (n – 1) * n

Analysis:

1. Input size: given number = n

2. Basic operation: multiplication

3. NO best, worst, average cases.

4. Let M (n) denotes number of multiplications.

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

$$M(0) = 0 \qquad \text{initial condition}$$

Where: M (n – 1): to compute Factorial (n – 1)

1: t o multiply Factorial (n – 1) by n

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

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

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

$$= M(n-2) + 2$$

$$= [\,M(n-3) + 1\,] + 3$$

= M (n – 3) + 3

…

In the ith recursion, we have

= M (n – i) + i

When i = n, we have

= M (n – n) + n = M (0) + n

Since M (0) = 0

= n

M (n) ⬚ Θ (n)

6. **Give three different algorithms to compute the GCD of 2 positive numbers**

Problem: GCD of Two numbers m,n
Input specifiastion: Two inputs, nonnegative, not both zero

Euclid's algorithm

gcd(m,n)=gcd(n,m mod n)

Untill m mod n =0, since gcd (m,0) =m

Another way of representation of the same algorithm

Euclid's algorithm

Step1:if n=0 return val of m & stop else proceed step 2

Step 2: Divide m by n & assign the value of remainder to r

Step 3: Assign the value of n to m,r to n,Go to step1.

Another algorithm to solve the same problem

Euclid's algorithm

Step1: Assign the value of min (m,n) to t

Step 2: Divide m by t.if remainder is 0,go to step3 else goto step4

Step 3: Divide n by t.if the remainder is 0, return the value of t as the answer and Stop, otherwise proceed to step4

Step4: Decrease the value of t by 1. go to step 2

7. Explain all asymptotic notations used in algorithm analysis
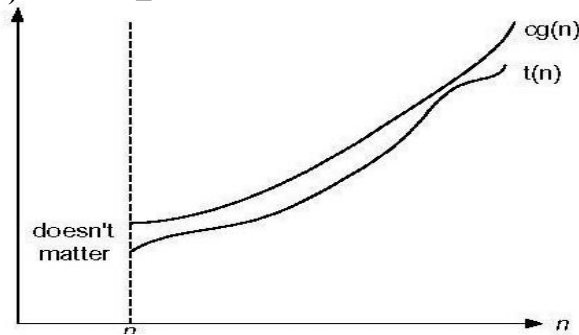
Asymptotic Notations

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

Big Oh- O notation

Definition:

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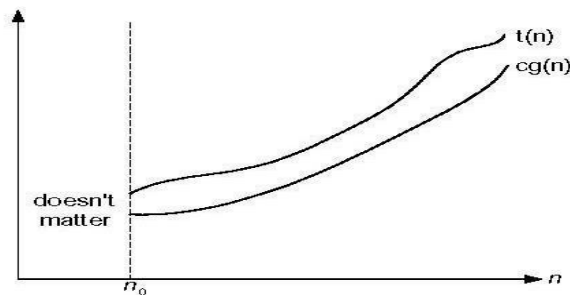
t(n) ≤ cg(n) for all n ≥ n0

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

Big Omega- Ω notation

Definition:

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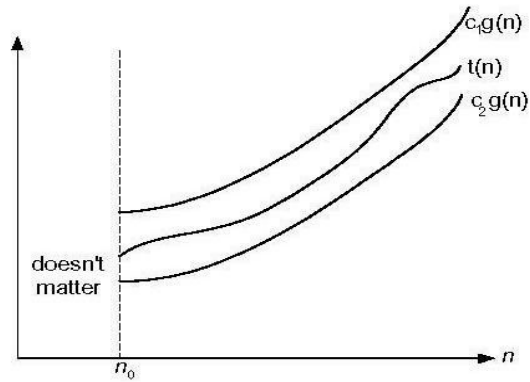
t(n) ≥ cg(n) for all n ≥ n0

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

Big Theta- Θ notation

Definition:

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

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

Unit 2

1. What is brute-force method? Write a brute-force string matching algorithm. Analyze its complexity.

Brute force is a straightforward approach to problem solving, usually directly based on the

problem's statement and definitions of the concepts involved. Though rarely a source of clever or efficient algorithms, the brute-force approach should not be overlooked as an important algorithm design strategy. Unlike some of the other strategies, **brute force** is applicable to a very wide variety of problems. For some important problems (e.g., sorting, searching, string matching), the brute-force approach yields reasonable algorithms of at least some practical value with no limitation on instance size Even if too inefficient in general, a brute-force algorithm can still be useful for solving small-size instances of a problem. A brute-force algorithm can serve an important theoretical or educational purpose.

Given a string of *n* characters called the *text* and a string of *m* characters (*m = n)* called the *pattern*, find a substring of the text that matches the pattern. To put it more precisely, we want to find *i*—the index of the leftmost character of the first matching

substring in the text—such that

$t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}$:

$t_0 \ldots t_i \ldots t_{i+j} \ldots t_{i+m-1} \ldots t_{n-1}$ **text *T***

$p_0 \ldots p_j \ldots p_{m-1}$ **pattern *P***

1. Pattern: 001011

Text: 10010101101001100101111010

2. Pattern: happy

Text: It is never too late to have a happy child

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

The algorithm shifts the pattern almost always after a single character comparison. in the worst case, the algorithm may have to make all *m* comparisons before shifting the pattern, and this can happen for each of the *n - m* + 1 tries. Thus, in the worst case, the algorithm is in θ*(nm)*.

2. **Write the quick sort algorithm. Analyze its efficiency. Apply the algorithm to sort the list : 5, 3, 1, 9, 8, 2, 4, 7, .**
   ALGORITHM Quicksort (A[l …r])

   //sorts by quick sort

   //i/p: A sub-array A[l..r] of A[0..n-1],defined by its left and right indices l and r
   //o/p: The sub-array A[l..r], sorted in ascending order
   if l < r
       Partition (A[l..r]) // s is a split position
       Quicksort(A[l..s-1])
       Quicksort(A[s+1..r])

ALGORITHM Partition (A[l ..r])
       //Partitions a sub-array by using its first element as a pivot
       //i/p: A sub-array A[l..r] of A[0..n-1], defined by its left and right indices l and r (l < r)
       //o/p: A partition of A[l..r], with the split position returned as this function's value
       p→A[l]
       i→l
       j→r + 1;
       Repeat
       repeat i→i + 1 until A[i] >=p   //left-right scan
       repeat j→j – 1 until A[j] < p   //right-left scan
       if (i < j)   //need to continue with the scan
       swap(A[i], a[j])
       until i >= j   //no need to scan
       swap(A[l], A[j])
       return j

Example: Sort by quick sort the following list: 5, 3, 1, 9, 8, 2, 4, 7, show recursion tree.

3. Write the algorithm for binary search and find the average case efficiency

ALGORITHM BinSrch (A[0 … n-1], key)
    //implements non-recursive binary search
    //i/p: Array A in ascending order, key k
    //o/p: Returns position of the key matched else -1
    $l \rightarrow 0$
    $r \rightarrow n-1$
    while $l \leq r$ do
    $m \rightarrow (l + r) / 2$
    if key $== A[m]$
    return m
    else
    if key $< A[m]$
    $r \rightarrow m-1$
    else
    $l \rightarrow m+1$
    return -1

Analysis:
- **Input size:** Array size, n
- **Basic operation:** key comparison
- **Depend on**

Best – key matched with mid element

Worst – key not found or key sometimes in the list

- Let C(n) denotes the number of times basic operation is executed. Then

$C_{worst}(n)$ = Worst case efficiency. Since after each comparison the algorithm
divides the problem into half the size, we have

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

$C(1) = 1$

- Solving the recurrence equation using master theorem, to give the number of
times the search key is compared with an element in the array, we have:

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

$a = 1$

$b = 2$

$f(n) = n^0 ; d = 0$

case 2 holds:

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

$= \Theta (n^0 \log n)$

$= \Theta (\log n)$

4. Discuss the merge sort algorithm with recursive tree and its efficiency. Apply the same algorithm to sort the list {4,6,1,3,9,5,2,7}.
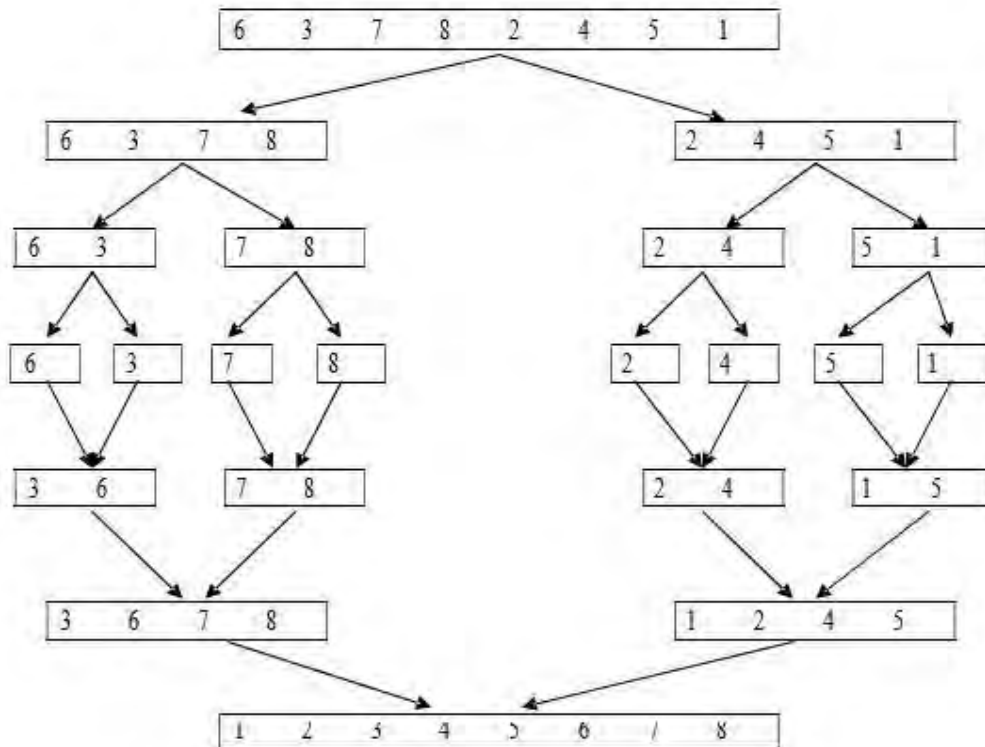
```
ALGORITHM Mergesort ( A[0… n-1] )
        //sorts array A by recursive mergesort
        //i/p: array A
        //o/p: sorted array A in ascending order
        if n > 1

        copy A[0… (n/2 -1)] to B[0… (n/2 -1)]
        copy A[n/2… n -1)] to C[0… (n/2 -1)]
        Mergesort ( B[0… (n/2 -1)] )
        Mergesort ( C[0… (n/2 -1)] )
        Merge ( B, C, A )


ALGORITHM Merge ( B[0… p-1], C[0… q-1], A[0… p+q-1] )
        //merges two sorted arrays into one sorted array
        //i/p: arrays B, C, both sorted
        //o/p: Sorted array A of elements from B & C
        I →0
        j→0
        k→0
        while i < p and j < q do
        if B[i] ≤ C[j]
                A[k] →B[i]
                i→i + 1
        else
                A[k] →C[j]
                j→j + 1
        k→k + 1
        if i == p
        copy C [ j… q-1 ] to A [ k… (p+q-1) ]
        else
        copy B [ i… p-1 ] to A [ k… (p+q-1) ]
```

5. **Using bubble sort algorithm. Arrange the letters of the word a" QUESTION" in alphabetical order**

Bubble Sort

Compare adjacent elements of the list and exchange them if they are out of order.Then we repeat the process,By doing it repeatedly, we end up 'bubbling up' the largest element to the last position on the list

ALGORITHM

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

//The algorithm sorts array *A*[0..*n* - 1] by bubble sort

//Input: An array *A*[0..*n* - 1] of orderable elements

//Output: Array *A*[0..*n* - 1] sorted in ascending order

for *i*=0 **to** *n* - 2 **do**

for *j*=0 **to** *n* - 2 - *i* **do**

if *A*[*j* + 1]<*A*[*j*]

swap *A*[*j*] and *A*[*j* + 1]

```
89 ⇄? 45    68      90      29      34      17
45      89 ⇄? 68      90      29      34      17
45      68      89 ⇄? 90  ⇄? 29      34      17
45      68      89      29      90 ⇄? 34      17
45      68      89      29      34      90 ⇄? 17
45      68      89      29      34      17     |90

45 ⇄? 68 ⇄? 89 ⇄? 29      34      17     |90
45      68      29      89 ⇄? 34      17     |90
45      68      29      34      89 ⇄? 17     |90
45      68      29      34      17     |89      90
```

etc.

The first 2 passes of bubble sort on the list 89, 45, 68, 90, 29, 34, 17. A new line is shown after a swap of two elements is done. The elements to the right of the vertical bar are in their final positions and are not considered in subsequent iterations of the algorithm

6. **Give the general divide and conquer recurrence and explain the same. Give the master's theorem .**

General divide & conquer recurrence:

An instance of size n can be divided into b instances of size n/b, with "a" of them needing
to be solved. [a ≥ 1, b > 1].
Assume size n is a power of b. The recurrence for the running time T(n) is as follows:

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

where:

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

Therefore, the order of growth of T(n) depends on the values of the constants a & b and
the order of growth of the function f(n).

Master theorem

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

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

Example:
Let T(n) = 2T(n/2) + 1, solve using master theorem.
Solution:

Here: a = 2
     b = 2
     $f(n) = \Theta(1)$

$$d = 0$$

Therefore:
$$a > b^d \text{ i.e., } 2 > 2^0$$

Case 3 of master theorem holds good. Therefore:

$T(n) \in \Theta (n^{\log_b a})$

$\quad \in \Theta (n^{\log_2 2})$

$\quad \in \Theta (n)$

7. Explain the concept of divide and conquer methodology indicating three major variations.(Jan 2015).

Divide & conquer is a general algorithm design strategy with a general plan as follows:

1. DIVIDE:

A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size.

2. RECUR:

Solve the sub-problem recursively.

3. CONQUER:

If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance.

The base case for the recursion is sub-problem of constant size.

Advantages of Divide & Conquer technique:

• For solving conceptually difficult problems like Tower Of Hanoi, divide & conquer is a powerful tool

• Results in efficient algorithms

• Divide & Conquer algorithms are adapted foe execution in multi-processor machines

• Results in algorithms that use memory cache efficiently.

Limitations of divide & conquer technique:

• Recursion is slow

• Very simple problem may be more complicated than an iterative approach. Example: adding n numbers etc

Diagram 1 shows the general divide & conquer plan
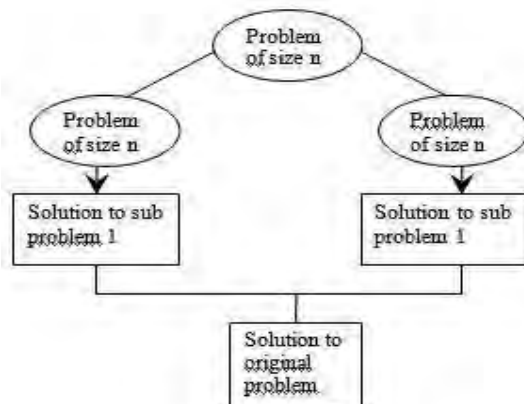
Unit-3

1. Justify the statement "prim's algorithm always yields minimum cost spanning tree". Give the prim's algorithm and discuss about its time complexity.

ALGORITHM Prim (G)
//Prim's algorithm for constructing a MST
//Input: A weighted connected graph G = { V, E }
//Output: ET the set of edges composing a MST of G
// the set of tree vertices can be initialized with any vertex

$V_T \rightarrow$ { v_0}

$E_T \rightarrow$ Ø

for I $\rightarrow$ 1 to |V| - 1 do

Find a minimum-weight edge e* = (v*, u*) among all the edges (v, u) such that

v is in V_T and u is in V - V_T
$V_T \rightarrow$ V_T U { u*}

$E_T \rightarrow$ E_T U { e*}

return E_T

Efficiency of Prim's algorithm is based on data structure used to store priority queue.
- Unordered array: **Efficiency: $\Theta(n^2)$**
- Binary heap: **Efficiency: $\Theta(m \log n)$**
- Min-heap: For graph with n nodes and m edges: **Efficiency: (n + m) log n**
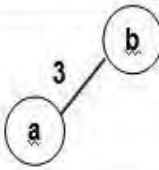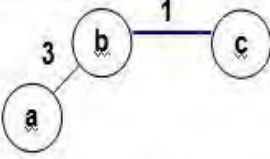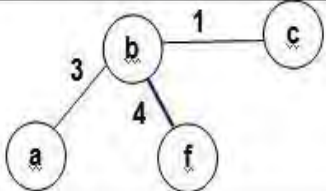
Conclusion:

- Prim's algorithm is a "vertex based algorithm"
- Prim's algorithm "Needs priority queue for locating the nearest vertex." The choice of priority queue matters in Prim implementation.

Array - optimal for dense graphs

Binary heap - better for sparse graphs

Fibonacci heap - best in theory, but not in practice

Tree vertices	Remaining vertices	Graph
a (-, -)	b (a , 3) c (- , ∞) d (- , ∞) e (a , 6) f (a , 5)	
b (a, 3)	c (b , 1) d (- , ∞) e (a , 6) f (b , 4)	
c (b, 1)	d (c , 6) e (a , 6) f (b , 4)	
f (b, 4)	d (f , 5) e (f , 2)	
e (f, 2)	d (f , 5)	
d(f, 5)		Algorithm stops since all vertices are included. The weight of the minimum spanning tree is 15

2. Give the dijkstra's algorithm. What is its complexity? Discuss with a simple example Jan 2015/ Jan 2014/ June 13)

ALGORITHM Dijkstra(G, s)

//Input: Weighted connected graph G and source vertex s

//Output: The length Dv of a shortest path from s to v and its penultimate vertex Pv

    for every vertex v in V
    //initialize vertex priority in the priority queue
    Initialize (Q)
    `**for** every vertex v in V **do**
        $D_v→∞$; $P_v→$**null**   // Pv , the parent of v
        insert(Q, v, D_v) //initialize vertex priority in priority queue
        $d_s→0$
        //update priority of s with ds, making ds, the minimum
        Decrease(Q, s, d_s)
        $V_T→$⬚

        for i→0 to |V| - 1 do

        u*→DeleteMin(Q)
        //expanding the tree, choosing the locally best vertex
        $V_T→V_T$ U {u*}
        for every vertex u in V – V_T that is adjacent to u* do
                if Du* + w (u*, u) < Du
                        Du→Du + w (u*, u); Pu   u*
                        Decrease(Q, u, Du)

Conclusion:

- Doesn't work with negative weights
- Applicable to both undirected and directed graphs
- Use unordered array to store the priority queue: **Efficiency = $\Theta(n^2)$**
- Use min-heap to store the priority queue: **Efficiency = O(m log n)**

 3. **Use the kruskal's method to solve the min cost spanning tree to the above graph.**

The method:
    STEP 1: Sort the edges by increasing weight
    STEP 2: Start with a forest having |V| number of trees.
    STEP 3: Number of trees are reduced by ONE at every inclusion of an edge At each
            stage:
        • Among the edges which are not yet included, select the one with minimum
            weight AND which does not form a cycle.
        • the edge will reduce the number of trees by one by combining two trees
            of the forest
    Algorithm stops when |V| -1 edges are included in the MST i.e : when the
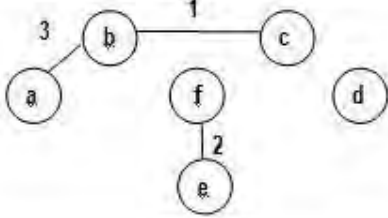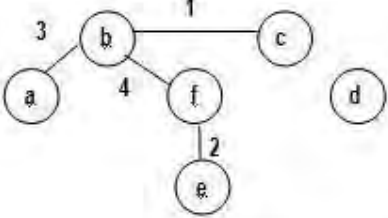   **number of trees in the forest is reduced to ONE.**
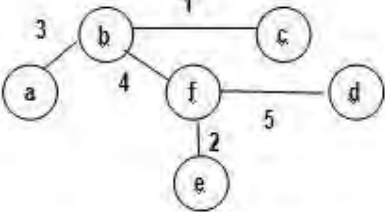
Solution:

The list of edges is:

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

Sort the edges in ascending order:

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

Edge	ab	
Weight	3	
Insertion status	YES	
Insertion order	3	

Edge	bf	
Weight	4	
Insertion status	YES	
Insertion order	4	

Edge	cf
Weight	4
Insertion status	NO
Insertion order	-

Edge	af
Weight	5
Insertion status	NO
Insertion order	-

Edge	df	
Weight	5	
Insertion status	YES	
Insertion order	5	

Algorithm stops as |V| –1 edges are included in the MST

4. Solve the following instances of the single source shortest path problem with vertex 'a' as the source.

Solution:

Length Dv of shortest path from source (s) to other vertices v and Penultimate vertex Pv

for every vertex v in V:

Da = 0 ,   Pa = null
Db = ∞ , Pb = null
Dc = ∞ , Pc = null
Dd = ∞ , Pd = null
De = ∞ , Pe = null
Df = ∞ ,   Pf = null

Tree vertices	Remaining vertices	Distance & Path vertex		Graph
a (-, 0)	b (a , 3) c (- , ∞) d (-, ∞) e (a , 6) f (a , 5)	Da = 0 Db = 3 Dc = ∞ Dd = ∞ De = 6 Df = 5	Pa = a Pb = [a, b] Pc = null Pd = null Pe = [a, e] Pf = [a, f]	
b (a, 3)	c (b , 3+1) d (-, ∞) e (a , 6) f (a , 5)	Da = 0 Db = 3 Dc = 4 Dd = ∞ De = 6 Df = 5	Pa = a Pb = [a, b] Pc = [a, b, c] Pd = null Pe = [a, e] Pf = [a, f]	
c (b, 4)	d (c , 4+6) e (a , 6) f (a , 5)	Da = 0 Db = 3 Dc = 4 Dd = 10 De = 6 Df = 5	Pa = a Pb = [a, b] Pc = [a, b, c] Pd = [a, b, c, d] Pe = [a, e] Pf = [a, f]	

f(a,5)	d(c,10) e(a,6)	Da=0 Pa=a Db = 3 Pb=[a,b] Dc=4 Pc=[a,b,c] Dd=10 Pd=[a,b,c,d] De= 6 Pe=[a, e] Df=5 Pf=[a,f]	
e(a,6)	d(c,10)	Da=0 Pa=a Db = 3 Pb=[a,b] Dc=4 Pc=[a,b,c] Dd=10 Pd=[a,b,c,d] De= 6 Pe=[a, e] Df=5 Pf=[a,f]	
d(c, 10)			Algorithm stops since no edges to scan

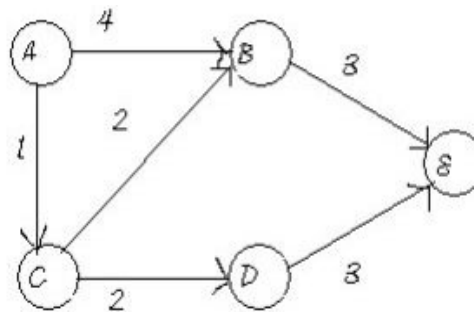**5. Solve the following instances of the single source shortest path problem with vertex 'a' as the source.**

Figure: Weighted-directed graph

The above weighted graph has 5 vertices from A-E. The value between the two
vertices is known as the edge cost between two vertices. For example the edge
cost between A and C is 1. Using the above graph the Dijkstra's algorithm is
used to determine the shortest path from the source A to the remaining
vertices in the graph.

The example is solved as follows:

- Initial step
  sDist[A]=0 ; *the value to the source itself*
  sDist[B]= ∞, sDist[C]= ∞, sDist[D]= ∞, sDist[E]= ∞; *the nodes not*
  processed yet

- Step 1
  Adj[A]={B,C}; *computing the value of the adjacent vertices of the graph*
  sDist[B]=4;

sDist[C]=2;

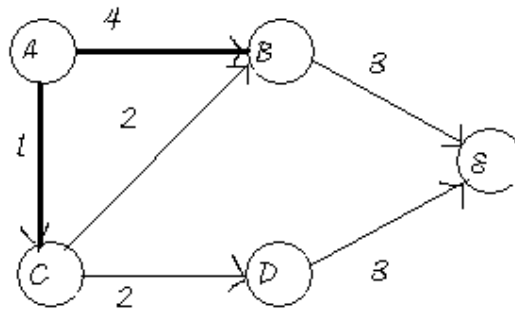*Figure: shortest path to vertices B, C from A*

- Step 2
  Computation from vertex C
  Adj[C] = {B, D};
  sDist[B] > sDist[C]+ EdgeCost[C,B]
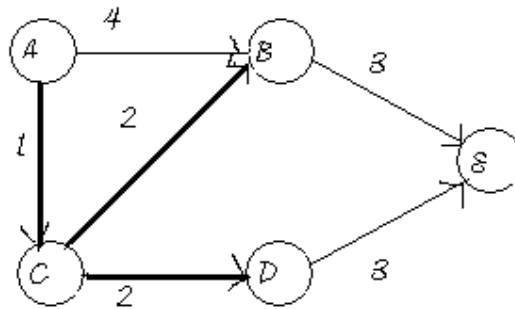   4   > 1+2 *(True)*
  Therefore, sDist[B]=3;

  sDist[D]=2;

  Figure: Shortest path from B, D using C as intermediate
vertex
  Adj[B]={E};
  sDist[E]=sDist[B]+EdgeCost[B,E]
        =3+3=6;
    Adj[D]={E};
    sDist[E]=sDist[D]+EdgeCost[D,E]
          =3+3=6

    This is same as the initial value that was computed so sDist[E]
value is
not changed.

- Step 4
  Adj[E]=0; means there is no outgoing edges from E
  And no more vertices, algorithm terminated. Hence the path which
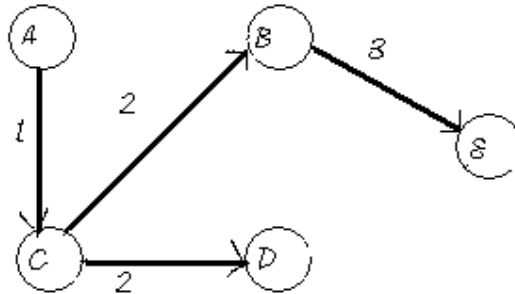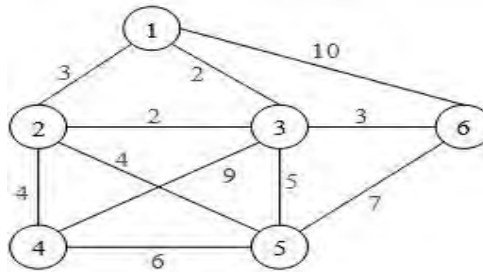 follows the algorithm is

Figure: the path obtained using Dijkstra's Algorithm

6. Using kruskal's algorithm, obtain the min cost spanning tree for the graph given below.

Kruskal's Algorithm

Input: A weighted, connected and undirected graph $G = (V, E)$.

Output: A minimum spanning tree T for G.

Method.

1. $T = \varphi$;

2. **While** T contains less than n-1 edges **do** // n = |V|

   Choose an edge (u, v) from E of the smallest weight;

   If (the adding of (u, v) to T does not form a cycle in T) then

Add

   (u, v) to T;

   Delete edge (u, v) from E;

3. **Output** T.

(c) By the disjoint union and find operations:

   A tree in the forest is used to represent a SET. Two operations on

   disjoint sets are as follows:

   union(S_1, S_2): union S_1 with S_2 to form a new set S_1.

   find(u): return a set name containing element u.

   Either of the above operations needs O(1) time.

   To check whether or not one cycle exists if edge (u, v) is added into

   the current forest, we perform the following operations:

   If (u, v) ☐ E and u, v are in the same set, then the addition of (u, v)

will form a cycle. (perform find(u) and find(v) to get the sets containing u and v, respectively)

If (u, v) □ E and u □ S₁, v □ S₂, then perform union(S₁, S₂), where S₁ and S₂ are distinct sets.

For example, for a spanning forest in the following, we would like to check whether one cycle exists if we add edge (3, 4) we must perform the following operations:
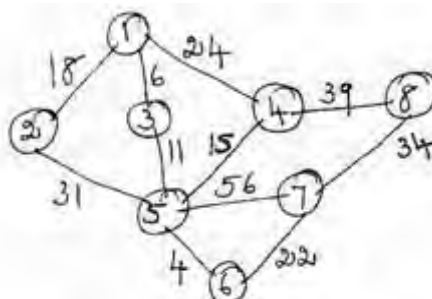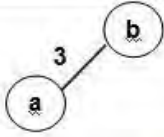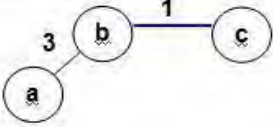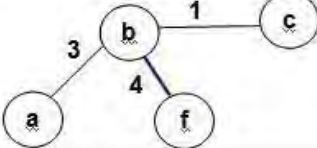

①find(3) = S₁; find(4) = S₁

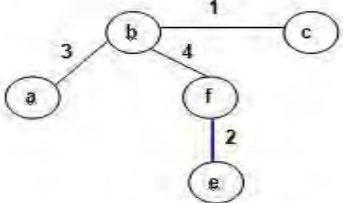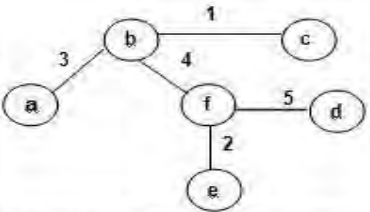②Since vertices 3 and 4 are in the same set, edge (3, 4) cannot be added into the forest.
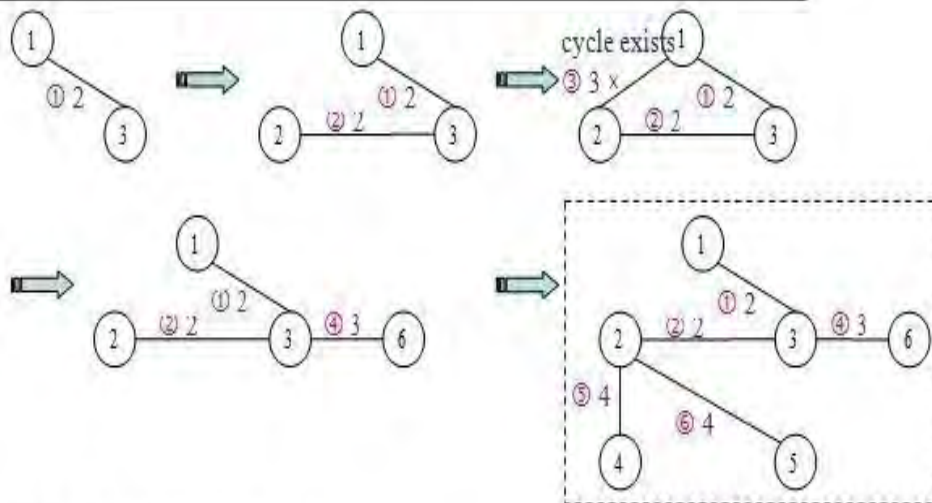
weight(T) = 2+2+3+4+4 = 15.

7. Using prim's algorithm , obtain the min cost spanning tree for the graph given below.(July 2014 / June 2013)

Tree vertices	Remaining vertices	Graph
a (-, -)	b (a , 3) c (-, ∞) d (-, ∞) e (a , 6) f (a , 5)	
b (a, 3)	c (b , 1) d (-, ∞) e (a , 6) f (b , 4)	
c (b, 1)	d (c , 6) e (a , 6) f (b , 4)	
f (b, 4)	d (f , 5) e (f , 2)	
e (f, 2)	d (f , 5)	
d(f, 5)		Algorithm stops since all vertices are included. The weight of the minimum spanning tree is 15

8. Write an algorithm for kruskal method and find its time complexity(jan 2015/jan 2013)
   Algorithm:
   ALGORITHM Kruskal (G)

//Kruskal's algorithm for constructing a MST
//Input: A weighted connected graph G = { V, E }
//Output: ET the set of edges composing a MST of G Sort E in ascending order of the edge weights
// initialize the set of tree edges and its size
$E_T \to \emptyset$
edge_counter →0
//initialize the number of processed edges
K →0

while edge_counter < |V| - 1

k→k + 1

if E_T U { $e_{i\ k}$} is acyclic

   E_T →E_T U { $e_{i\ k}$ }

   edge_counter →edge_counter + 1

return E_T

Efficiency:

Efficiency of Kruskal's algorithm is based on the time needed for sorting the
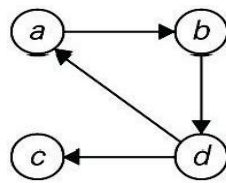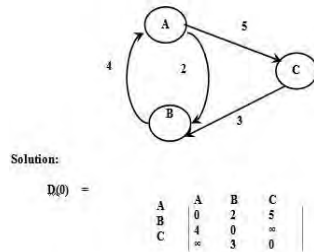
edge weights of a given graph.

• With an efficient sorting algorithm: **Efficiency: $\Theta(|E| \log |E|)$**

Unit 4

1. What are memory functions? Explain how they are used to solve the knapsack problem. Solve the instance of the knapsack problem below. Capacity W= 5

Item #	Weight (Kg)	Value (Rs.)
1	2	3
2	3	4
3	4	5
4	5	6

Step	Calculation	Table
1	**Initial conditions:** $V[0,j]=0$ for $j\geq 0$ $V[i,0]=0$ for $i\geq 0$	V[i,j] with j=0..5, i=0..4: i=0: 0 0 0 0 0 0 i=1: 0 i=2: 0 i=3: 0 i=4: 0
2	W1 = 2, Available knapsack capacity = 1 W1 > WA, CASE 1 holds: $V[i,j]=V[i-1,j]$ $V[1,1]=V[0,1]=0$	i=0: 0 0 0 0 0 0 i=1: 0 0 i=2: 0 i=3: 0 i=4: 0
3	W1 = 2, Available knapsack capacity = 2 W1 = WA, CASE 2 holds: $V[i,j]=\max\{V[i-1,j],$ $vi+V[i-1,j-wi]\}$ $V[1,2]=\max\{V[0,2],$ $3+V[0,0]\}$ $=\max\{0,3+0\}=3$	i=0: 0 0 0 0 0 0 i=1: 0 0 3 i=2: 0 i=3: 0 i=4: 0
4	W1 = 2, Available knapsack capacity = 3,4,5 W1 < WA, CASE 2 holds: $V[i,j]=\max\{V[i-1,j],$ $vi+V[i-1,j-wi]\}$ $V[1,3]=\max\{V[0,3],$ $3+V[0,1]\}$ $=\max\{0,3+0\}=3$	i=0: 0 0 0 0 0 0 i=1: 0 0 3 3 3 3 i=2: 0 i=3: 0 i=4: 0
5	W2 = 3, Available knapsack capacity = 1 W2 > WA, CASE 1 holds: $V[i,j]=V[i-1,j]$ $V[2,1]=V[1,1]=0$	i=0: 0 0 0 0 0 0 i=1: 0 0 3 3 3 3 i=2: 0 0 i=3: 0 i=4: 0
6	W2 = 3, Available knapsack capacity = 2 W2 > WA, CASE 1 holds: $V[i,j]=V[i-1,j]$ $V[2,2]=V[1,2]=3$	i=0: 0 0 0 0 0 0 i=1: 0 0 3 3 3 3 i=2: 0 0 3 i=3: 0 i=4: 0
7	W2 = 3, Available knapsack capacity = 3 W2 = WA, CASE 2 holds: $V[i,j]=\max\{V[i-1,j],$ $vi+V[i-1,j-wi]\}$ $V[2,3]=\max\{V[1,3],$ $4+V[1,0]\}$ $=\max\{3,4+0\}=4$	i=0: 0 0 0 0 0 0 i=1: 0 0 3 3 3 3 i=2: 0 0 3 4 i=3: 0 i=4: 0
8	W2 = 3, Available knapsack capacity = 4 W2 < WA, CASE 2 holds: $V[i,j]=\max\{V[i-1,j],$ $vi+V[i-1,j-wi]\}$ $V[2,4]=\max\{V[1,4],$ $4+V[1,1]\}$ $=\max\{3,4+0\}=4$	i=0: 0 0 0 0 0 0 i=1: 0 0 3 3 3 3 i=2: 0 0 3 4 4 i=3: 0 i=4: 0

2. State all-pairs shortest path algorithm. Solve the given problem using the below figure

Solution:

$$D_s(0) = \quad \begin{array}{c|ccc} & A & B & C \\ \hline A & 0 & 2 & 5 \\ B & 4 & 0 & \infty \\ C & \infty & 3 & 0 \end{array}$$

FIGURE 8.2 (a) Digraph. (b) Its adjacency matrix. (c) Its transitive closure.

3. Using Warshall's algorithm, obtain the transitive closure of the matrix given below

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

Ones reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

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

Ones reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex a (note a new path from d to b); boxed row and column are used for getting $R^{(2)}$.

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

Ones reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., a and b (note two new paths); boxed row and column are used for getting $R^{(3)}$.

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

Ones reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., a, b, and c (no new paths); boxed row and column are used for getting $R^{(4)}$.

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

Ones reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., a, b, c, and d (note five new paths).

FIGURE 8.4 Application of Warshall's algorithm to the digraph shown. New ones are in bold.

4. **State all-pairs shortest path algorithm. Solve the given problem using the matrix given below**

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

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

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

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e. just a (note two new shortest paths from b to c and from d to c).

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

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e. a and b (note a new shortest path from c to a).

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

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e. a, b, and c (note four new shortest paths from a to b, from a to d, from b to d, and from d to b).

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

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e. a, b, c, and d (note a new shortest path from c to a).

FIGURE 8.7 Application of Floyd's algorithm to the graph shown. Updated elements are shown in bold.
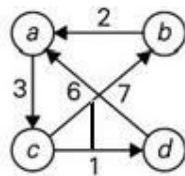
(a) Digraph. (b) Its weight matrix. (c) Its distance matrix.

5. Using the Dynamic programming .solve the following Knapsack instance.

$$\cdot V[i, j] = \max \left(V[i-1, j], V[i-1, j-w_i] + v_i\right)$$

object i not used   object i used

Example: Knapsack of capacity $W = 5$

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

capacity j

0 1 2 3 4 5

0

$w_1 = 2, v_1 = 12$ 1

$w_2 = 1, v_2 = 10$ 2

$w_3 = 3, v_3 = 20$ 3

$w_4 = 2, v_4 = 15$ 4   ?

$$\cdot \ V[i, j] = \max \ (V[i-1, j], \ V[i-1, j-w_i] + v_i)$$

$$\underbrace{\quad}_{object_i \ not \ used} \qquad \underbrace{\quad}_{object_i \ used}$$

capacity j

i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$ 1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$ 2	0	10	12	22	22	22
$w_3 = 3, v_3 = 20$ 3	0	10	12	22	30	32
$w_4 = 2, v_4 = 15$ 4	0	10	15	25	30	37

6. Write the algorithm to find the shortest path using Floyd's approach
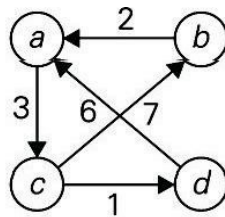
$$W = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \infty & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \infty & 0 \end{array}$$

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

(a)   (b)   (c)

FIGURE 8.5 (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.

Underlying idea of Floyd's algorithm:

- Let W denote the initial weight matrix.
- Let D(k) [i, j] denote cost of shortest path from i to j whose intermediate vertices
  are a subset of {1,2,…,k}.
- **Recursive Definition**
  Case 1:
  A shortest path from vi to vj restricted to using only vertices from
  {v1,v2,…,vk} as intermediate vertices does not use vk. Then
  $D(k) [i, j] = D(k-1) [i, j]$.
  Case 2:
  A shortest path from vi to vj restricted to using only vertices from
  {v1,v2,…,vk} as intermediate vertices do use vk. Then
  $D(k) [i, j] = D(k-1) [i, k] + D(k-1) [k, j]$.
  We conclude:
  $D(k)[i, j] = \min \{ D(k-1) [i, j], D(k-1) [i, k] + D(k-1) [k, j] \}$

Algorithm:
Algorithm Floyd(W[1..n, 1..n])
// Implements Floyd's algorithm
// Input: Weight matrix W
// Output: Distance matrix of shortest paths' length
D →W

```
for k→1 to n do
    for i→    1 to n do
        for j→    1 to n do
            D [ i, j]→    min { D [ i, j], D [ i, k] +  D [ k, j]
return D
```

7. **Explain dynamic programming. list out the differences between divide and conquer and dynamic programming.**
   Dynamic Programming Properties

   • An instance is solved using the solutions for smaller instances.
   • The solutions for a smaller instance might be needed multiple times, so store their results in a table.
   • Thus each smaller instance is solved only once.
   • Additional space is used to save time.

Divide & Conquer	Dynamic Programming
1. Partitions a problem into independent smaller sub-problems	1. Partitions a problem into overlapping sub-problems
2. Doesn't store solutions of sub-problems. (Identical sub-problems may arise - results in the same computations are performed repeatedly.)	2. Stores solutions of sub-problems: thus avoids calculations of same quantity twice
3. Top down algorithms: which logically progresses from the initial instance down to the smallest sub-instances via intermediate sub-instances.	3. Bottom up algorithms: in which the smallest sub-problems are explicitly solved first and the results of these used to construct solutions to progressively larger sub-instances

Unit 5

1. Differentiate between DFS and BFS tree traversals. Explain how DFS algorithm can be used to obtain the topological sorting with an example

Difference between DFS & BFS:

	DFS	BFS
Data structure	Stack	Queue
No. of vertex orderings	2 orderings	1 ordering
Edge types	Tree edge Back edge	Tree edge Cross edge
Applications	Connectivity Acyclicity Articulation points	Connectivity Acyclicity Minimum edge paths
Efficiency for adjacency matrix	$\Theta(n^2)$	$\Theta(n^2)$
Efficiency for adjacency lists	$\Theta(n + e)$	$\Theta(n + e)$

Topological Sorting
Description:

Topological sorting is a sorting method to list the vertices of the graph in such an order
that for every edge in the graph, the vertex where the edge starts is listed before the
vertex where the edge ends.
NOTE: There is no solution for topological sorting if there is a cycle in the digraph .
[MUST be a DAG]

Topological sorting problem can be solved by using
1. DFS method
2. Source removal method

DFS Method:
  • Perform DFS traversal and note the order in which vertices become dead ends
       (popped order)
  • Reverse the order, yield the topological sorting.

Example:

Apply DFS – based algorithm to solve the topological sorting problem for the given graph:

Step	Graph	Remarks
1		Insert C1 into stack C1(1)
2		Insert C2 into stack C2 (2) C1(1)
3		Insert C4 into stack C4 (3) C2 (2) C1(1)
4		Insert C5 into stack C5 (4) C4 (3) C2 (2) C1(1)
5	NO unvisited adjacent vertex for C5, backtrack	Delete C5 from stack C5 (4, 1) C4 (3) C2 (2) C1(1)
6	NO unvisited adjacent vertex for C4, backtrack	Delete C4 from stack C5 (4, 1) C4 (3, 2) C2 (2) C1(1)

7	NO unvisited adjacent vertex for C3, backtrack	Delete C3 from stack C5 (4, 1) C4 (3,2) C2 (2, 3) C1(1)
8	NO unvisited adjacent vertex for C1, backtrack	Delete C1 from stack C5 (4, 1) C4 (3,2) C2 (2, 3) C1(1, 4)
Stack becomes empty, but there is a node which is unvisited, therefore start the DFS again from arbitrarily selecting a unvisited node as source		
9	C2	Insert C2 into stack C5 (4, 1) C4 (3,2) C2 (2, 3) C1(1, 4) C2(5)
10	NO unvisited adjacent vertex for C2, backtrack	Delete C2 from stack C5 (4, 1) C4 (3,2) C2 (2, 3) C1(1, 4) C2(5, 5)
Stack becomes empty, NO unvisited node left, therefore algorithm stops. The popping – off order is: C5, C4, C3, C1, C2, Topologically sorted list (reverse of pop order): C2 C1 C3 C4 C5		

2. **Write an algorithm for topological sort of a diagraph using DFS algorithm. Prove the correctness of the algorithm.**

   ALGORITHM DFS (G)

//implements DFS traversal of a given graph
//i/p: Graph G = { V, E}
//o/p: DFS tree
Mark each vertex in V with 0 as a mark of being "unvisited"
Count →0
for each vertex v in V do
   if v is marked with 0
       dfs(v)
   dfs(v)
   count→   count + 1
   mark v with count
   for each vertex w in V adjacent to v do
   if w is marked with 0
   dfs(w)

3. **Show how insertion sort arranges the following members in increasing order. 89, 45, 68, 90, 29, 34, 17**

Algorithm:

ALGORITHM Insertionsort(A [0 … n-1])
//sorts a given array by insertion sort
//i/p: Array A[0…n-1]
//o/p: sorted array A[0…n-1] in ascending order
for i→   1 to n-1
      V→   A[i]
      j→   i-1
      while j ≥ 0 AND A[j] > V do
          A[j+1]   A[j]
          j→   j – 1
    A[j + 1]→   V

Analysis:

- **Input size:** Array size, n
- **Basic operation:** key comparison
- Best, worst, average case exists
  Best case: when input is a sorted array in ascending order:
  Worst case: when input is a sorted array in descending order:
- Let $C_{worst}(n)$ be the number of key comparison in the worst case. Then

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

- Let $C_{best}(n)$ be the number of key comparison in the best case.
  Then

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

Example:
Sort the following list of elements using insertion sort:
    89, 45, 68, 90, 29, 34, 17

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

4. **What are the three major variations of decrease and conquer technique explain each with an example.(june 2013).**

Decrease & conquer is a general algorithm design strategy based on exploiting the relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem. The exploitation can be either top-down

(recursive) or bottom-up (non-recursive).

The major variations of decrease and conquer are

1. Decrease by a constant :(usually by 1):
   a. insertion sort
   b. graph traversal algorithms (DFS and BFS)
   c. topological sorting
   d. algorithms for generating permutations, subsets

2. Decrease by a constant factor (usually by half)

   a. binary search and bisection method
3. Variable size decrease
   a. Euclid's algorithm

Following diagram shows the major variations of decrease & conquer approach.

Decrease by a constant :(usually by 1):
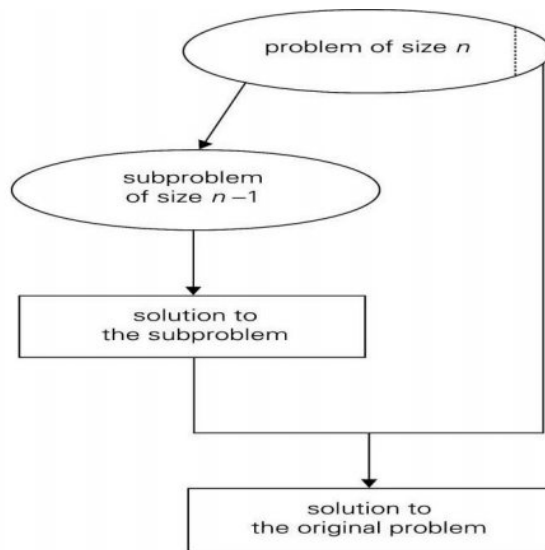
Decrease by a constant factor (usually by half)
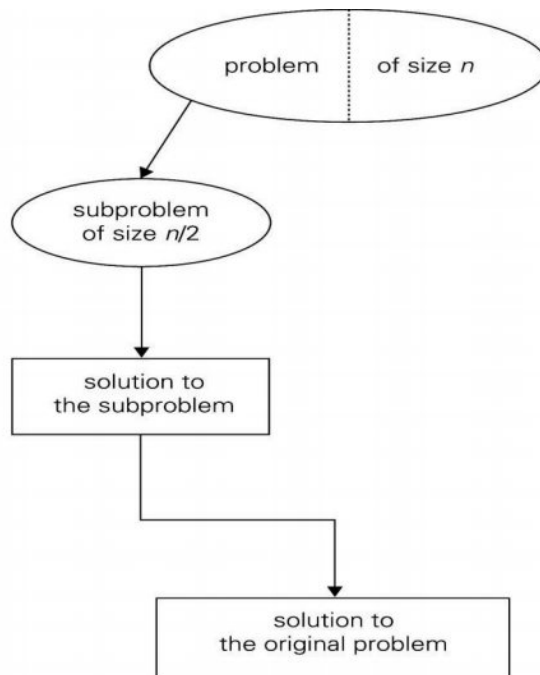
5. What do you mean by topological sort? Give its applications. Explain source removal method to find topological sorting.

Source removal method:

- Purely based on decrease & conquer
- Repeatedly identify in a remaining digraph a source, which is a vertex with no
  incoming edges
- Delete it along with all the edges outgoing from it.

Example:
Apply Source removal – based algorithm to solve the topological sorting problem for the given graph:
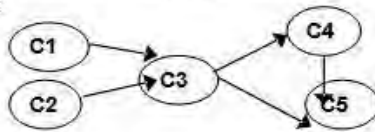
Solution:
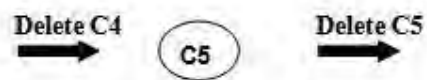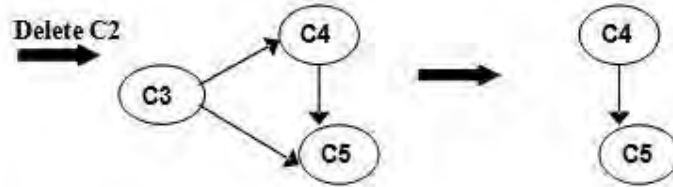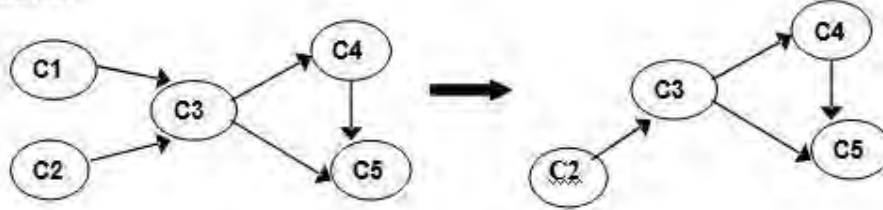
The topological order is C1, C2, C3, C4, C5

UNIT 6

1. **Draw the decision tree for the 3-elements insertion sort**

   Decision Trees

   Decision tree — a convenient model of algorithms involving Comparisons in which: (i)internal nodes represent comparisons (ii) leaves represent outcomes Decision tree for 3-element insertion sort
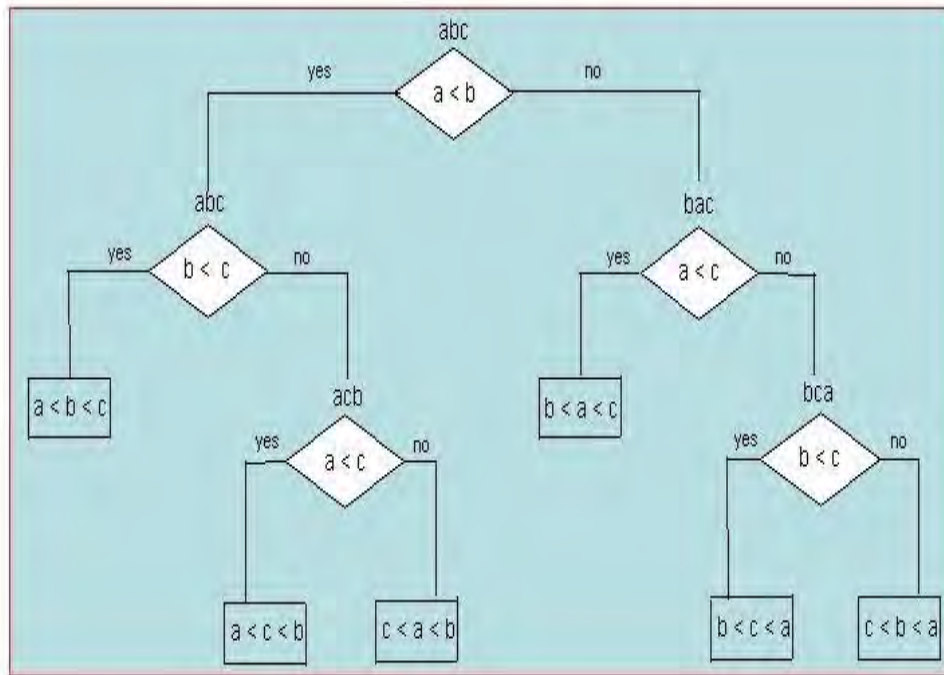
2. **Solve the instance of knapsack problem using branch and bound algorithm**

	1	2	3	4
Weight	4	7	5	3
Value	$40	$42	$25	$12

The knapsack capacity is W=10. (july/june2011)

The 0/1 knapsack problem

- Positive integer $P_1, P_2, \ldots, P_n$ (profit)
  $W_1, W_2, \ldots, W_n$ (weight)
  M (capacity)

maximize $\sum_{i=1}^{n} P_i X_i$

subject to $\sum_{i=1}^{n} W_i X_i \leq M$ $X_i = 0$ or $1, i = 1, \ldots, n.$

The problem is modified:

minimize $-\sum_{i=1}^{n} P_i X_i$
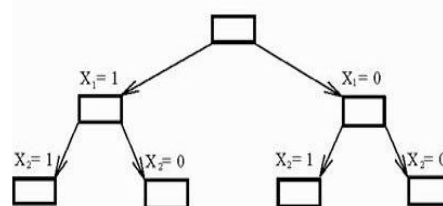
The Branching Mechanism in the Branch-and-Bound Strategy to Solve 0/1 Knapsack Problem.

How to find the upper bound?

- Ans: by quickly finding a feasible solution in a greedy manner: starting from the smallest available i, scanning towards the largest i's until M is exceeded. The upper bound can becalculated.

The 0/1 knapsack problem

- E.g. n = 6, M = 34

	1	2	3	4	5	6
P_i	6	10	4	5	6	4
W_i	10	19	8	10	12	8

$(P_i/W_i \geq P_{i+1}/W_{i+1})$

- A feasible solution: X1 = 1, X2 = 1, X3 = 0, X4 = 0, X5 = 0, X6 = 0
-(P1+P2) = -16 (upper bound)
Any solution higher than -16 can not be an optimal solution.

How to find the lower bound?

- Ans: by relaxing our restriction from Xi = 0 or 1 to $0 \leq Xi \leq 1$ (knapsack problem)

Let $-\sum_{i=1}^{n} P_i X_i$ be an optimal solution for 0/1 knapsack problem and $-\sum_{i=1}^{n} P_i X'_i$ be an optimal solution for fractional knapsack problem. Let

$Y = -\sum_{i=1}^{n} P_i X_i$, $Y' = -\sum^{n} P_i X'_i$.

☐ $Y' \leq Y$

3. **Define the following i) tractable problem ii) class p iii) class NP iv) polynomial reduction v) NP complete problems.**

ii) Class *P*

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

Examples:
- searching

- Element uniqueness
- graph connectivity
- graph acyclicity
- Primality testing (finally proved in 2002)

iii)**Class *NP***

NP (*nondeterministic polynomial*): class of decision problems whose proposed solutions can be
verified in polynomial time = solvable by a *nondeterministic polynomial algorithm*
A *nondeterministic polynomial algorithm* is an abstract two-stage procedure that:
- generates a random string purported to solve the problem
- checks whether this solution is correct in polynomial time

By definition, it solves the problem if it's capable of generating and verifying a solution on one
of its tries
Why this definition?
- led to development of the rich theory called "computational complexity"

Example: CNF satisfiability
Problem: Is a Boolean expression in its conjunctive normal form (CNF) satisfiable, i.e., are there
values of its variables that makes it true?
This problem is in *NP*. Nondeterministic algorithm:
- Guess truth assignment
- Substitute the values into the CNF formula to see if it evaluates to true

Example: (A | ¬B | ¬C) & (A | B) & (¬B | ¬D | E) & (¬D | ¬E)
Truth assignments:
A B C D E
0 0 0 0 0
. . .
1 1 1 1 1
Checking phase: $O(n)$

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

***P = NP* ?**
- One of the most important unsolved problems is computer science is

whether or not
> *P=NP*.

> – If *P=NP*, then a ridiculous number of problems currently believed to be very

> > difficult will turn out have efficient algorithms.

> – If *P≠NP*, then those problems definitely do not have polynomial time solutions.

• Most computer scientists suspect that P ≠ NP. These suspicions are based partly on the idea of NP-completeness.

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

Nondeterministic Polynomial (NP) Problems
- • This class of problems has solutions that are verifiable in polynomial time.

> – Thus any problem in P is also NP, since we would be able to solve it in

> > polynomial time, we can also verify it in polynomial time

V) NP-Complete Problems

A decision problem *D* is <u>*NP*-complete</u> if it's as hard as any problem in *NP*, i.e.,
- • *D* is in *NP*
- • every problem in *NP* is polynomial-time reducible to *D*
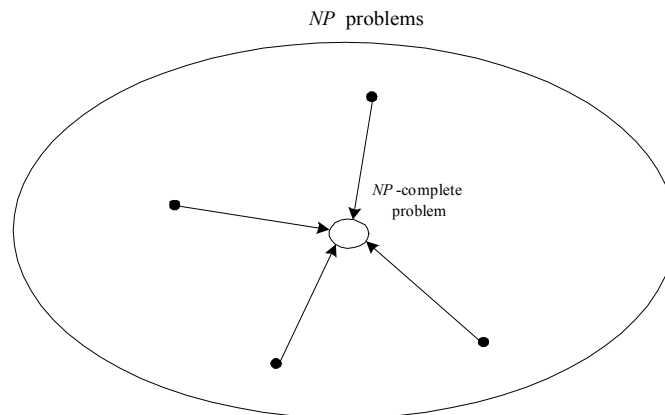
Other *NP*-complete problems obtained through polynomial- time reductions from a known *NP*-complete problem

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

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

- Problems
    - Decision problems (yes/no)
    - Optimization problems (solution with best score)
- P
    - Decision problems (decision problems) that can be solved in polynomial time
    - can be solved "efficiently"
- NP
    - Decision problems whose "*YES*" answer can be verified in

polynomial

    time, if we already have the *proof* (or *witness*)

- co-NP
    - Decision problems whose "*NO*" answer can be verified in polynomial time,
        if we already have the *proof* (or *witness*)
- e.g. The satisfiability problem (SAT)
- Given a boolean formula

$$(x_1 \lor x_2 \lor x_3 \lor x_4) \land (x_5 \lor x_6 \lor x_7) \land x_8 \lor x_9$$

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

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

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

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

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

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

    – C is in NP
    – Any known NP-hard (or complete) problem $\leq_p$ C
    – Thus a proof must show these two being satisfied

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

P = {all decision problems that can be solved in polynomial time}

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

4. **Explain the concept of decision tree for sorting algorithms**

   Decision Trees:

   There must be at least one leaf for each correct output.
    – The tree must be tall enough to have that many leaves.
- In a binary tree with l leaves and height h, $h \geq \log_2 l$

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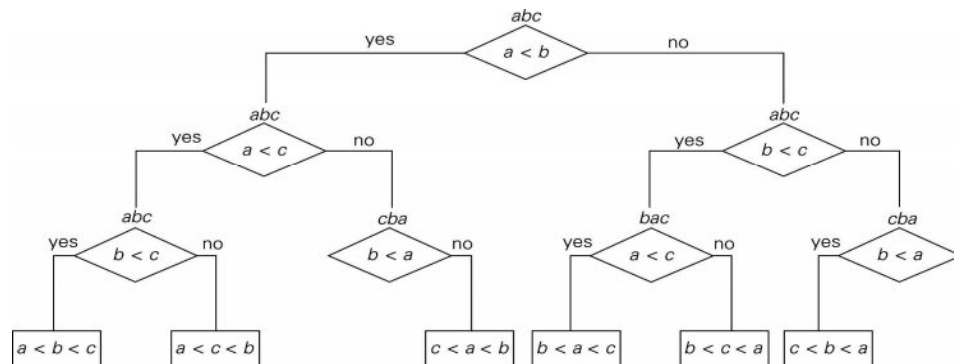
FIGURE 11.2 Decision tree for the three-element selection sort. A triple above a node indicates the state of the array being sorted. Note the two redundant comparisons $b < a$ with a single possible outcome because of the results of some previously made comparisons.

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

UNIT-7

   1. Explain how backtracking is used for solving 4- queens problem. Show the state space tree.

   N-Queens Problem:

• The object is to place queens on a chess board in such as way as no queen can capture another one in a single move

   Recall that a queen can move horz, vert, or diagonally an infinite distance.

   This implies that no two queens can be on the same row, col, or diagonal

   We usually want to know how many different placements there are

4-Queens

• Lets take a look at the simple problem of placing queens 4 queens on a 4x4 board

• The brute-force solution is to place the first queen, then the second, third, and forth

   – After all are placed we determine if they are placed legally

• There are 16 spots for the first queen, 15 for the second, etc.

   – Leading to 16*15*14*13 = 43,680 different combinations

• Obviously this isn't a good way to solve the problem

• First lets use the fact that no two queens can be in the same col to help us

   – That means we get to place a queen in each col

• So we can place the first queen into the first col, the second into the second,

• This cuts down on the amount of work

   – Now there are 4 spots for the first queen, 4 spots for the second, etc.

• 4*4*4*4 = 256 different combinations

• However, we can still do better because as we place each queen we can look at the previous queens we have placed to make sure our new queen is not in the same row or diagonal as a previously place queen

• Then we could use a Greedy-like strategy to select the next valid position for each col

- So now what do we do?
- Well, this is very much like solving a maze

  – As you walk though the maze you have to make a series of choices
  – If one of your choices leads to a dead end, you need to back up to the last choice you made and take a different route .

  - That is, you need to change one of your earlier selections
  – Eventually you will find your way out of the maze

2. Differentiate between back tracking and branch- and –bound algorithms

Backtracking : A Scenario

A tree is composed of nodes

Backtracking can be thought of as searching a tree for a particular "goal" leaf node
• Each non-leaf node in a tree is a parent of one or more other nodes (its children)
• Each node in the tree, other than the root, has exactly one parent

The backtracking algorithm
• Backtracking is really quite simple--we "explore" each node, as follows:
• To "explore" node N:
       1. If N is a goal node, return "success"
       2. If N is a leaf node, return "failure"
       3. For each child C of N,
           3.1. Explore C
               3.1.1. If C was successful, return "success"
       4. Return "failure"

• Construct the _state-space tree_
    – nodes: partial solutions
    – edges: choices in extending partial solutions
• Explore the state space tree using depth-first search
• "Prune" _nonpromising nodes_
    – dfs stops exploring sub-trees rooted at nodes that cannot lead to a solution and backtracks to such a node's parent to continue the search.

Branch and Bound Searching Strategies

Feasible Solution vs. Optimal Solution
• DFS, BFS, hill climbing and best-first search can be used to solve some searching problem for searching a feasible solution.
• However, they cannot be used to solve the optimization problems for searching an (the) optimal solution.

The Branch-and-bound strategy
• This strategy can be used to solve optimization problems without an

exhaustive search in the average case.
   • 2 mechanisms:
 A mechanism to generate branches when searching the solution space
  A mechanism to generate a bound so that many braches can be terminated
   • It is efficient in the average case because many branches can be terminatedvery early.
   • Although it is usually very efficient, a very large tree may be generated in the worst case.
   • Many NP-hard problem can be solved by B&B efficiently in the average case; however,
   the worst case time complexity is still exponential.

Bounding

A bound on a node is a guarantee that any solution obtained from expanding the node will be:
   – Greater than some number (lower bound)
   – Or less than some number (upper bound)
If we are looking for a minimal optimal, as we are in weighted graph coloring, then we need a lower bound
For example, if the best solution we have found so far has a cost of 12 and the lower bound on a node is 15 then there is no point in expanding the node
The node cannot lead to anything better than a 15
   • We can compute a lower bound for weighted graph color in the following way:
      – The actual cost of getting to the node
      – Plus a bound on the future cost
         • Min weight color * number of nodes still to color
         That is, the future cost cannot be any better than this
   • Recall that we could either perform a depth-first or a breadth-first search
      – Without bounding, it didn't matter which one we used because we had to expand the entire tree to find the optimal solution
      – Does it matter with bounding?
         • Hint: think about when you can prune via bounding
   • We prune (via bounding) when:
      (currentBestSolutionCost <= nodeBound)
   • This tells us that we get more pruning if:
      – The currentBestSolution is low
   – And the nodeBound is high
   • So we want to find a low solution quickly and we want the highest possible lower bound
      – One has to factor in the extra computation cost of computing higher lower bounds vs. the expected pruning savings

3. **State subset-sum problem. Using backtracking obtain a solution to the subset-sum problem by backtracking S={6,8,2,14} and d=16**
   b. Apply backtracking to solve the following instance of the subset sum problem

      S={5,1,4,3}   and d=11.
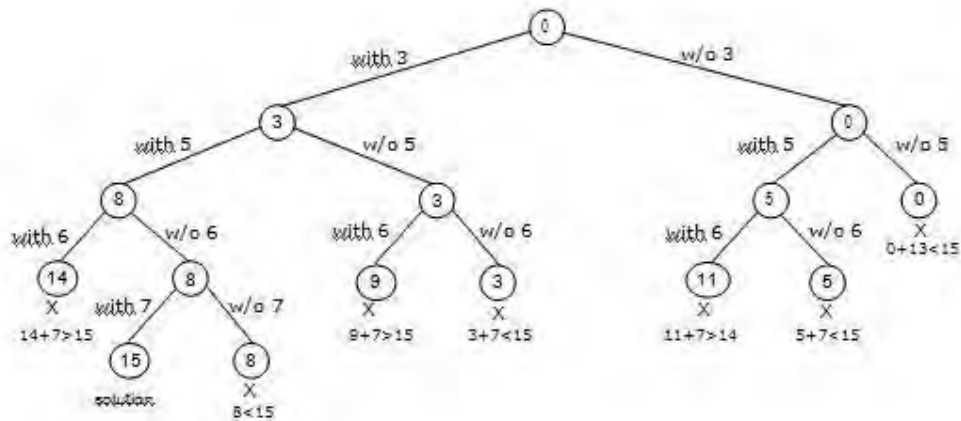
4. **Explain approximation algorithm for NP- hard problem in general. Also discuss the approximation algorithms for knapsack problem**

Approximation Approach

Apply a fast (i.e., a polynomial-time) approximation algorithm to get a solution that is not
necessarily optimal but hopefully close to it
Accuracy measures:
accuracy ratio of an approximate solution *sa*
$r(sa) = f(sa) / f(s^*)$ for minimization problems
$r(sa) = f(s^*) / f(sa)$ for maximization problems
where f(*sa*) and f(*s**) are values of the objective function f for the approximate solution *sa* and
actual optimal solution *s**
performance ratio of the algorithm A the lowest upper bound of $r(sa)$ on all i

5. **Solve the following instance of Assignment problem using Branch-Bound technique.**

Job1	Job2	Job3	Job4
9	2	7	8
6	4	3	7
5	8	1	8
7	6	9	4

Assignment Problem:
Select one element in each row of the cost matrix *C* so that:
- no two selected elements are in the same column
- the sum is minimized

	Job 1	Job 2	Job 3	Job 4
Person *a*	9	2	7	8
Person *b*	6	4	3	7
Person *c*	5	8	1	8
Person *d*	7	6	9	4

<u>**Lower bound:**</u> Any solution to this problem will have total cost
at least: $2 + 3 + 1 + 4$ (or $5 + 2 + 1 + 4$)
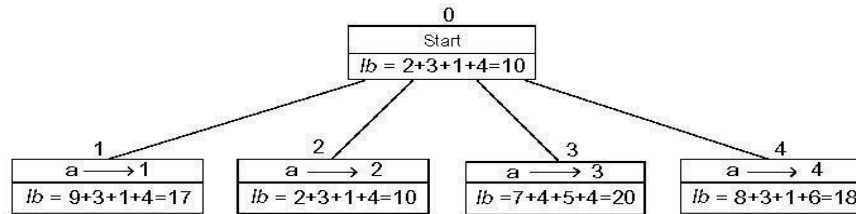
Example: Complete state-space tree

Figure 11.5 Levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated. A node's fields indicate the job number assigned to person a and the lower bound value, lb, for this node.
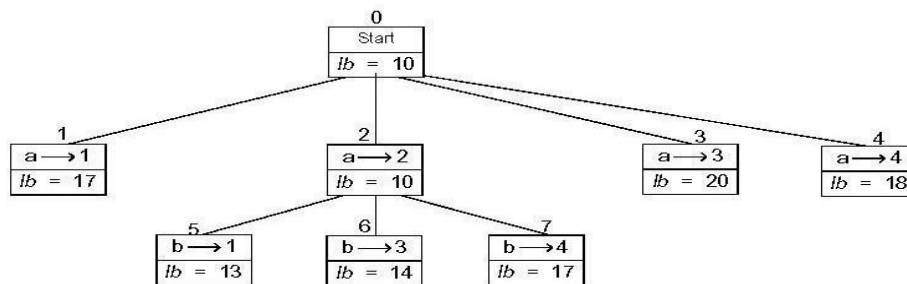

Figure 11.6 Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm
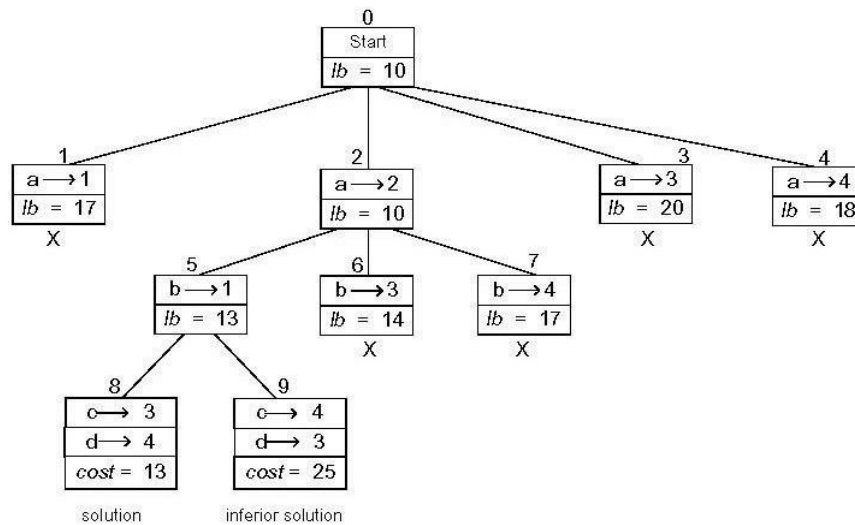

Figure 11.7 Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm

UNIT 8

1. What is prefix computation problem? Give the algorithm for prefix computation which uses i)n processors. ii)n/log n processors. Obtain the time complexities of these algorithms.

PRAM Recursive Prefix Sum Algorithm

Input: Array of ($x1, x2, ..., xn$) elements, $n = 2k$
Output: Prefix sums $si, 1 < i < n$
begin
if n = 1 **then** $s1 = x1$; **exit**
for $1 < i < n/2$ **pardo**
$yi := x2i–1 + x2i$
Recursively compute prefix sums of y and store in z
for $1 < i < n$ **pardo**
if i is even **then** $si := zi/2$
if $i > 1$ is odd **then** $si := z(i–1)/2 + xi$
if $i = 1$ **then** $s1 := x1$

Theorem: An algorithm that runs in T time on the p-processor priority CRCW PRAM can be simulated by EREW PRAM to run in O(T log
p) time. A concurrent read or write of an p-processor CRCW PRAM can be implemented on a p processor EREW PRAM to execute in O(log p) time
$Q1,…,Qp$ CRCW processors, such that Qi has to read (write) $M[ji]$
$P1,…,Pp$ EREW processors
$M1,…,Mp$ denote shared memory locations for special use Pi stores $<ji,i>$ in Mi
Sort pairs in lexicographically non-decreasing order in O(log p) time using EREW merge sort algorithm
Pick representative from each block of pairs that have same firs component in O(1) time
 Representative Pi reads (writes) from $M[k]$ with $<k,\_>$ in Mi and copies data to each M in the block in O(log p) time using EREW segmented parallel prefix algorithm
 Pi reads data from Mi

2. **For n x n matrix M with non-negative integer coefficients, define M and give an algorithm for computing m. prove that M cnan be computed from n x n matrix M in O(log n)time using CRCW PRAM processors for any fixed € > 0**

Consider $n×n$ matrix multiplication $C = A B$ using $n3$ processors.

 Each element of C
 $cij = \Sigma k=1..n\ aik\ bkj$
can be computed on the CREW PRAM in parallel using n processors in O(log n) time All cij can be computed using $n3$ processors in O(log n)

Matrix multiply with $n3$ processors (i,j,l)
Each processor (i,j,l) executes:
Input: $n×n$ matrices A and $B, n = 2k$
Output: $C = A B$
begin
$C'[i,j,l] := A[i,l]B[l,j]$
for $h = 1$ **to** log n **do**
if $i < n/2h$ **then**

C'[i,j,l] := *C'[i,j,2l–1]* + *C'[i,j,2l]*
if *l* = 1 **then**
C[i,j] := *C'[i,j,1]*
End

CREW versus EREW PRAM
- Algorithm on the CREW PRAM requires O(log *n*) time and O(*n*3) operations (*n*2 processors perform O(*n*) ops)
- On the EREW PRAM, the exclusive reads of *aij* and *bij* values can be satisfied by making *n* copies of *a* and *b*, which takes O(log *n*) time with *n* processors (broadcast
- tree)
- Total time is still O(log *n*).But requires more work and total memory requirement is huge!