CONTENTS

Chapter-1
Introduction

1. 1 what is Algorithm?

It is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. It is thus a sequence of computational steps that transform the input into the output. It is a tool for solving a well - specified computational problem.

Algorithm must have the following criteria:

Input: Zero or more quantities is supplied

Output: At least one quantity is produced.

Definiteness: Each instruction is clear and unambiguous.

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

Effectiveness: Every instruction must be basic so that it can be carried out.

1.2 What is program? Why DAA?

A program is the expression of an algorithm in a programming language. A set of instructions which the computer will follow to solve a problem.
It is learning general approaches to algorithm design.

Divide and conquer
Greedy method
Dynamic Programming
Basic Search and Traversal Technique
Graph Theory
Branch and Bound
NP Problems

1.3 Why do Analyze Algorithms?

To examine methods of analyzing algorithm
Correctness and efficiency
- **-Recursion equations**
- **-Lower bound techniques**
- **-O,Omega and Theta notations for best/worst/average case analysis**

Decide whether some problems have no solution in reasonable time
- **-List all permutations of n objects (takes n! steps)**
- **-Travelling salesman problem**

Investigate memory usage as a different measure of efficiency.

1.4 Importance of Analyzing Algorithms

Need to recognize limitations of various algorithms for solving a problem. Need to understand relationship between problem size and running timeWhen is a running program not good enough? Need to learn how to analyze an algorithm's running time without coding it. Need to learn techniques for writing more efficient code. Need to recognize bottlenecks in code as well as which parts of code are easiest to optimize.

1.4.1 The Selection Problem

Problem: given a group of n numbers, determine the k^{th} largest

Algorithm 1

Store numbers in an array
Sort the array in descending order
Return the number in position k

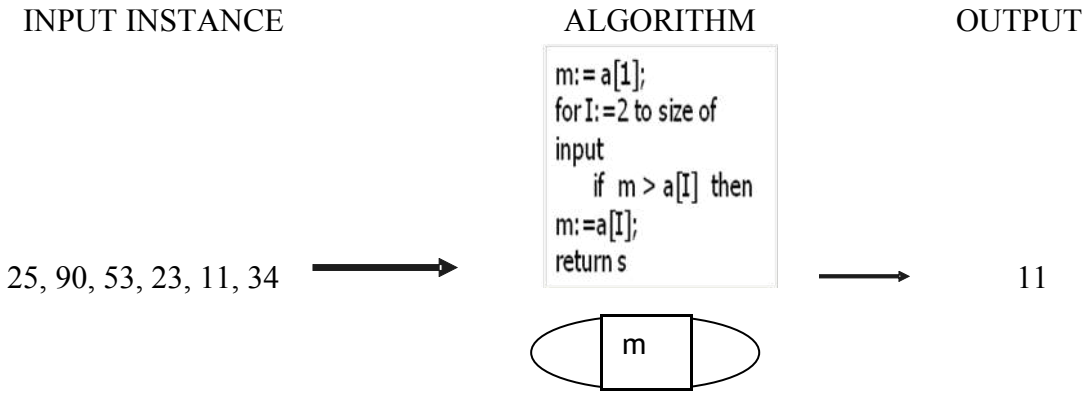
Algorithm 2

Store first k numbers in an array
Sort the array in descending order
For each remaining number, if the number is larger than the k^{th} number, insert the number in the correct position of the array
Return the number in position k

Example

Input is a sequence of integers stored in an array.

Output the minimum.



Problem: Description of Input-Output relationship.

Algorithm: A sequence of computational step that transform the input into the output.

Data Structure: An organized method of storing and retrieving data.

Our task: Given a problem, design a *correct* and *good* algorithm that solves it.

Example Algorithm A

Problem: The input is a sequence of integers stored in array.
Output the minimum.

Algorithm:

$m \leftarrow a[1]$;
For $i \leftarrow 2$ to size of input;
    if $m > a[i]$ then $m \leftarrow a[i]$;
output m.

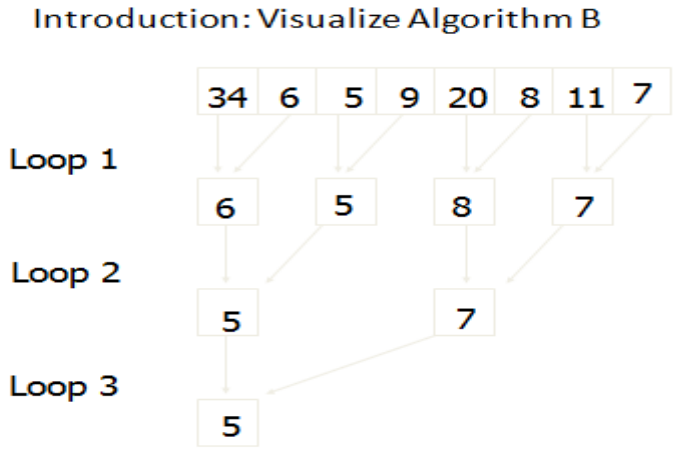
Example Algorithm B

This algorithm uses two temporary arrays.

1. copy the input *a* to array *t1;*
   assign *n* ←size of input;
2. While *n* > 1
   For *i* ←1 to *n* /2
   t2[*i*] ← min (*t1* [2**i*], *t1*[2**i* + 1]);
   copy array *t2* to *t1*;
   n ←*n/2;*
3. Output *t2*[1];

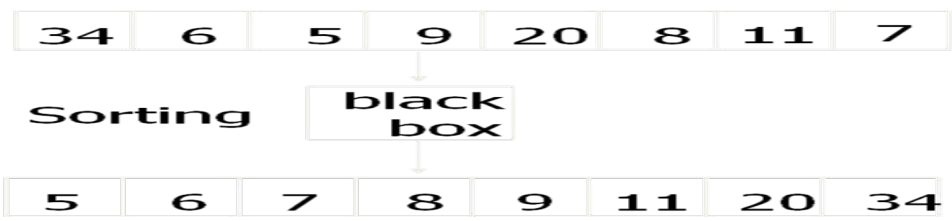
Visualize Algorithm B

34	6	5	9	20	8	11	7



Example Algorithm C

Sort the input in increasing order. Return the first element of the sorted data.

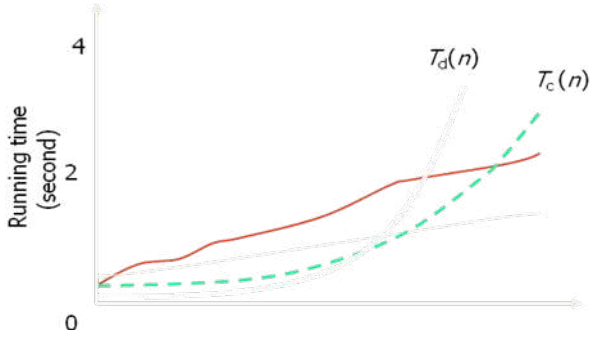


Introduction: Example Algorithm D
For each element, test whether it is the minimum.

```
1.  i ← 0;
    flag ← true;
2.  While flag
        i ← i + 1;
        min ← a[i];
        flag ← false;
        for j ← 1 to size of input
            if min > a[j] then flag ← true;
3.  Output min.
```

1.5 Time vs. Size of Input

Measurement parameterized by the size of the input. The algorihtms A,B,C are *implemented* and run in a PC. Algorithms D is implemented and run in a supercomputer. Let $T_k(n)$ be the amount of time taken by the Algorithm.



1.5.1 Methods of Proof

(a) Proof by Contradiction

Assume a theorem is false; show that this assumption implies a property known to be true is false --therefore original hypothesis must be true

(b) Proof by Counter example

Use a concrete example to show an inequality cannot hold. Mathematical Induction. Prove a trivial base case, assume true for k, and then show hypothesis is true for k+. Used to prove recursive algorithms

(c) Proof by Induction

Claim: S (n) is true for all n >= k

Basis: Show formula is true when n = k

Inductive hypothesis: Assume formula is true for an arbitrary n

Step: Show that formula is then true for n+1

Examples

Gaussian Closed Form

Prove $1 + 2 + 3 + \ldots + n = n (n+1) / 2$
Basis: If n = 0, then $0 = 0 (0+1) / 2$
Inductive hypothesis: Assume $1 + 2 + 3 + \ldots + n = n (n+1) / 2$
Step (show true for n+1): $1 + 2 + \ldots + n + n+1 = (1 + 2 + \ldots + n) + (n+1)$
$= n (n+1)/2 + n+1 = [n (n+1) + 2(n+1)]/2$
$= (n+1) (n+2)/2 = (n+1) (n+1 + 1) / 2$

Geometric Closed Form

Prove $a_0 + a_1 + \ldots + a_n = (a_n+1-1) / (a -1)$ for all a 1
Basis: show that $a_0 = (a_0+1-1) / (a -1)$
$a_0 = 1 = (a1-1) / (a -1)$
Inductive hypothesis: Assume $a_0 + a_1 + \ldots + an = (an+1-1) / (a -1)$
Step (show true for n+1): $a_0 + a_1 + \ldots + a_n+1 = a_0 + a_1 + \ldots + a_n + a_n+1$
$= (a_n+1-1) / (a -1) + a_n+1 = (a_n+1+1-1) / (a -1)$

Strong induction also holds

Basis: show S (0)

Hypothesis: assume S (k) holds for arbitrary k <= n

Step: Show S (n+1) follows

Another variation

Basis: show S (0), S (1)

Hypothesis: assume S (n) and S (n+1) are true

Step: show S (n+2) follows.

1.6 Basic Recursion

Base case: value for which function can be evaluated without recursion
Two fundamental rules:-
1. Must always have a base case
2. Each recursive call must be to a case that eventually leads toward a base case

Problem: Write an algorithm that will strip digits from an integer and print them out one by one

```
void print_out(int n)
if(n < print_digit(n); /*outputs single-digit to terminal*/
else
print_out (n/); /*print the quotient*/
print_digit (n % ); /*print the remainder*/
```

Prove by induction that the recursive printing program works:
Basis: If n has one digit, then program is correct.

hypothesis: Print_out works for all numbers of k or fewer digits

case k+: k+ digits can be written as the first k digits followed by the least significant digit

The number expressed by the first k digits is exactly floor (n /)? Which by hypothesis prints correctly; the last digit is n%; so the (k+)-digit is printed correctly. By induction, all numbers are correctly printed.

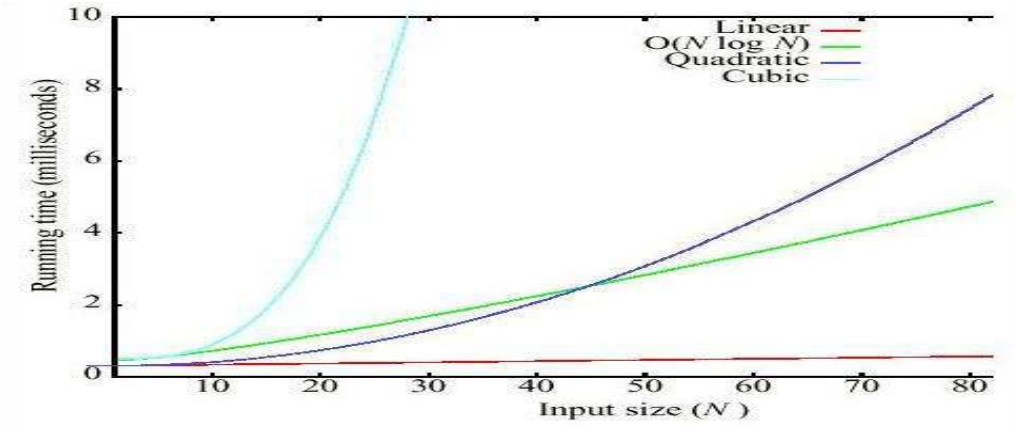
Recursion is expensive in terms of space requirement; avoid recursion if simple loop will do
Last two rules
Assume all recursive calls work
Do not duplicate work by solving identical problem in separated recursive calls
Evaluate fib () --use a recursion tree

Fib (n) = fib (n-1) + fib (n-2)

1.7 Algorithm Analysis and Running Time

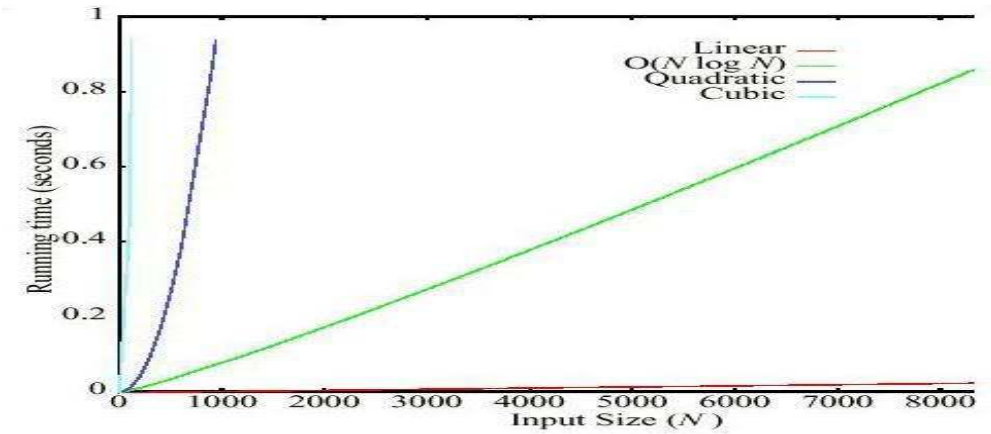
How to estimate the time required for an algorithm. Techniques that drastically reduce the running time of an algorithm. A mathematical framework that more rigorously describes the running time of an algorithm.

Running time for small inputs



Running times for small inputs

Running time for moderate inputs



Running time for moderate inputs

Algorithm Analysis

Measures the efficiency of an algorithm or its implementation as a program as the input size becomes very large. We evaluate a new algorithm by comparing its performance with that of previous approaches. Comparisons are asymtotic analyses of classes of algorithms. We usually analyze the time required for an algorithm and the space required for a data structure.

Many criteria affect the running time of an algorithm, including

-speed of CPU, bus and peripheral hardware
-design think time, programming time and debugging time
-language used and coding efficiency of the programmer
-quality of input (good, bad or average)
-Machine independent
-Language independent
-Environment independent (load on the system)
-Amenable to mathematical study
-Realistic

In lieu of some standard benchmark conditions under which two programs can be run, we estimate the algorithm's performance based on the number of key and basic operations it requires to process an input of a given size. For a given input size n we express the time T to run the algorithm as a function T (n). Concept of growth rate allows us to compare running time of two algorithms without writing two programs and running them on the same compute. Formally, let T (A, L, M) be total run time for algorithm A if it were implemented with language L on machine M. Then the complexity class of algorithm A is O (T(A, L, M) U O(T(A, L, M)) U O(T(A, L, M)) .

Call the complexity class V; then the complexity of A is said to be f if V = O (f).The class of algorithms to which A belongs is said to be of at most linear/quadratic/ etc. The growth in best case if the function TA best(n) is such (the same also for average and worst case).

1.8 Asymptotic Performance

Asymptotic performance means it always concerns with how does the algorithm behave as the problem size gets very large? Running time, Memory/storage requirements, and Band width/power requirements/logic gates/etc.

Asymptotic Notation

By now you should have an intuitive feel for asymptotic (big-O) notation:
What does O (n) running time mean? O (n2)? O (n log n).?
How does asymptotic running time relate to asymptotic memory usage?.
Our first task is to define this notation more formally and completely.

Analysis of Algorithms

Analysis is performed with respect to a computational model we will usually use a generic uni processor random-access machine (RAM).

All memory equally expensive to access.

No concurrent operations.

All reasonable instructions take unit time.

Ex: Except, of course, function calls

Constant word size

Ex: Unless we are explicitly manipulating bits

Input Size

Time and space complexity. This is generally a function of the input size.
E.g., sorting, multiplication
How we characterize input size depends:
Sorting: number of input items

Multiplication: total number of bits

Graph algorithms: number of nodes & edges

Running Time

Number of primitive steps that are executed. Except for time of executing a function call most statements roughly require the same amount of time.

y = m * x + b

c = 5 / 9 * (t -32)

z = f(x) + g(y)

Analysis

Worst case provides an upper bound on running time. An absolute guarantee
Average case provides the expected running time random (equally likely) inputs.

Function of Growth rate

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	N log N
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Functions in order of increasing growth rate

1.9 Space Complexity (S (P)=C+SP(I))

Fixed Space Requirements (C) Independent of the characteristics of the inputs and outputs instruction space. Space for simple variables, fixed-size structured variable, constants. Variable Space Requirements (SP(I))depend on the instance characteristic I–number, size, values of inputs and outputs associated with recursive stack space, formal parameters, local variables, return address.

Program: Simple arithmetic function

```
float abc (float a, float b, float c)
{
return a + b + b * c + (a + b -c) / (a + b) + 4.00;
}
```

Program: Iterative function for summing a list of numbers.

```
float sum(float list[ ], int n)
{
float tempsum = 0;
int i;
for (i = 0; i<n; i++)
tempsum += list [i];return tempsum;
}
```

Sabc(I) = 0
Ssum(I) = 0

Recall: pass the address of the first element of the array & pass by value.

Program: Recursive function for summing a list of numbers.

```
 float rsum(float list[ ], int n)
 {
 if (n) return rsum(list, n-1) + list[n-1];
return 0;
 }
```

Ssum(I)=Ssum(n)=6n

Assumptions

Space needed for one recursive call of

Type	Name	Number of bytes
parameter: float parameter: integer return address:(used internally)	List[n] N	2 2 2
TOTAL per recursive call		6

1.10 Time Complexity

- Compile time (C) : Independent of instance characteristics.
- Run (execution) time TP

Definition: TP (n) = caADD(n) + csSUB(n) + clLDA(n) + cstSTA(n)
A *program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.

Example

abc = a + b + b * c + (a + b -c) / (a + b) + 4.0

abc = a + b + c

Methods to compute the step count

- Introduce variable count into programs
- Tabular method

Determine the total number of steps contributed by each statement step per execution × frequency

add up the contribution of all statements

Program: Iterative summing of a list of numbers:

```
float sum(float list[ ], int n)
{
float tempsum = 0;
count++; /* for assignment */
int i;
for (i = 0; i < n; i++)
{
count++; /*for the for loop */
tempsum += list[i];
count++; /* for assignment */}
count++; /* last execution of for */
return tempsum;
count++; /* for return */
}
```

2n + 3 steps

Program: Simplified version of Program

```
float sum(float list[ ], int n)
{
```

```
float tempsum = 0;
int i;
for (i = 0; i < n; i++)
count += 2;
count += 3;
return 0;
}
```

2n + 3 steps

Program : Recursive summing of a list of numbers

```
float rsum(float list[ ], int n)
{
count++; /*for if conditional */
if (n)
{
count++; /* for return and rsum invocation */
return rsum(list, n-1) + list[n-1];
}
count++;
return list[0];
}
```

2n+2 times

Program : Matrix addition

```
void add( int a[ ] [MAX_SIZE], int b[ ] [MAX_SIZE],int c [ ] [MAX_SIZE], int rows, int
cols)
{
int i, j;
for (i = 0; i < rows; i++)
for (j= 0; j < cols; j++)
c[i][j] = a[i][j] +b[i][j];
}
```

Matrix addition with count statements:

```
void add(int a[ ][MAX_SIZE], int b[ ][MAX_SIZE],int c[ ][MAX_SIZE], int row, int cols )
{
int i, j;
for (i = 0; i < rows; i++)                    (2rows * cols + 2 rows + 1)
{
count++; /* for i for loop */
for (j = 0; j < cols; j++)
 {
count++; /* for j for loop */
c[i][j] = a[i][j] + b[i][j];
count++; /* for assignment statement */}
count++; /* last time of j for loop */
```

```
}
count++; /* last time of i for loop */
}
```

Program: Simplification of Program

```
void add(int a[ ][MAX_SIZE], int b [ ][MAX_SIZE],int c[ ][MAX_SIZE], int rows, int cols)
{
int i, j;for( i = 0; i < rows; i++)
{
for (j = 0; j < cols; j++)
count += 2;
count += 2;
 }
count++;
}
```

2rows cols + 2rows +1 times

Tabular Method

Step count table
Iterative function to sum a list of numbers

Statement	s/e	Frequency	Total steps
float sum(float list[ ], int n)	0	0	0
{	0	0	0
float tempsum = 0;	1	1	1
int i;	0	0	0
for(i=0; i <n; i++)	1	n+1	n+1
tempsum += list[i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
TOTAL			2n+3

Step count table for recursive summing function

Statement	s/e	Frequency	Total steps
float rsum(float list[ ], int n)	0	0	0
{	0	0	0
if (n)	1	n+1	n+1
return rsum(list, n-1)+list[n-1];	1	n	n
return list[0];	1	1	1
}	0	0	0
TOTAL			2n+3

Matrix Addition

Step count table for matrix addition

Statement	s/e	Frequency	Total steps
Void add (int a[ ][MAX_SIZE])	0	0	0
{	0	0	0
int i, j;	0	0	0
for (i = 0; i < row; i++)	1	rows+1	rows+1
for (j=0; j< cols; j++)	1	rows (cols+1)	rows cols+rows
c[i][j] = a[i][j] + b[i][j];	1	rows cols	rows cols
}	0	0	0
TOTAL		2 rows cols+2rows+1	

1.11 Asymptotic Notation (Q, *O*, W, *o*, w)

Defined for functions over the natural numbers.
Ex: $f(n) = Q(n_2)$.

Describes how $f(n)$ grows in comparison to $n2$. Define a *set* of functions; in practice used to compare two function sizes. The notations describe different rate-of-growth relations between the defining function and the defined set of functions.

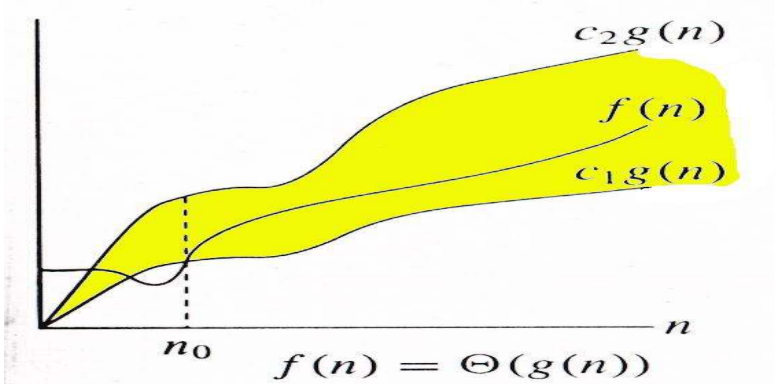
(a) Θ notation

For function $g(n)$, we define $(g(n))$, big-Theta of n, as the set:

$\Theta(g(n)) = \{ f(n) : \exists$ positive constants c_1, c_2, and n_0,such that $\forall n >= n_0$, we have $0 <= c_1 g(n) <= f(n) <= c_2 g(n) \}$

Intuitively:

Set of all functions that have the same rate of growth as g (n). g (n) is an asymptotically tight bound for f(n).



For function $g(n)$, we define $(g(n))$, big-Theta of n, as the set:

$\Theta(g(n)) = \{f(n)$: positive constants c_1, c_2, and n_0,such that $\forall\ n >= n_0$,
we have $0 <= c_1 g(n) < f(n) <= c_2 g(n)\}$

$F(n)$ and $g(n)$ are nonnegative, for large n.

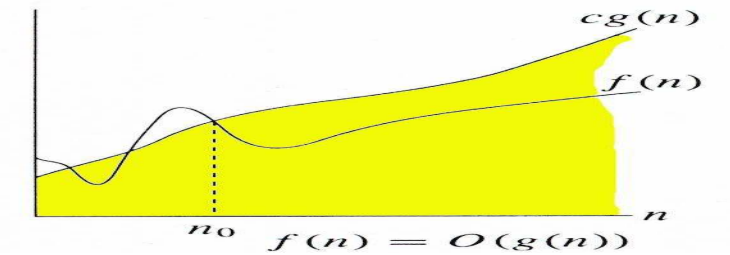
$3n+2 = \Theta(n)$ as $3n+2>=3n$ for all $n>=2$ and $3n+2<=4n$ for all $n>=2$
So $c_1=3$ and $c_2=4$and $n_0=2$. So, $3n+3= \Theta(n)$,
$10n^2+4n+2= \Theta(n^2)$, $6*2n+n^2= \Theta(2^n)$ and
$10*\log n+4= \Theta(\log n)$.$3n+2\# \Theta(1)$,
$3n+3\# \Theta(n2)$, $10n^2+4n+2\# \Theta(n)$, $10n^2+4n+2\# \Theta(1)$

***(b)O*-notation**

For function g (n), we define O (g(n)), big-O of n, as the set:

$O(g(n)) = \{f(n)$: positive constants c and n_0,such that $n>= n_0$, we have$0<= f(n)<=cg(n)\}$

Intuitively: Set of all functions whose rate of grow this same as or lower than that of g (n).
G (n) is an asymptotic upper bound for f(n).
$f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n)).\Theta(g(n)) \subset O(g(n))$.



Example- 1

7n-2
7n-2 is O (n)
need c > 0 and n_{01} such that7n-2 c• n for n n0
this is true for c = 7 and n0= 1

Example - 2

$3n^3+ 20n^2+ 5$
$3n^3+ 20n^2+ 5$ is O (n^3)
need c > 0 and n01 such that$3n^3+ 20n^2+ 5$ c•$n^3$for n n0
this is true for c = 4 and n0= 21

Example: 3
3 log n + log log n
3 log n + log log n is O (log n)
need c > 0 and n01 such that3 log n + log log n c• log n for n n0
this is true for c = 4 and n0= 2

Big-Oh and Growth Rate

The big-Oh notation gives an upper bound on the growth rate of a function. The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$. We can use the big-Oh notation to rank functions according to their growth rate.

	f(n)) is O (g(n))	g(n) is $O(f(n))$
g(n) grows more	Yes	No
f(n) grows more	No	Yes
Same growth	Yes	Yes

Big-Oh Rules

If is $f(n)$a polynomial of degree d, then $f(n)$is $O(nd)$, i.e.,
1. Drop lower-order terms
2. Drop constant factors

Use the smallest possible class of functions Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$". Use the simplest expression of the class. Say "$3n+5$ is $O(n)$" instead of "$3n+5$ is $O(3n)$"

Relatives of Big-Oh

Big-Omega
F (n) is Ω (g(n)) if there is a constant c > 0 and an integer constant $n_{0>=}1$ such that f(n)>= c• g(n) for n>= n0

Big-Theta
f(n) is Θ (g(n)) if there are constants c' > 0 and c'' > 0 and an integer constant n_0>1 such that c'• g(n)<= f(n)<= c''• g(n) for n>= n_0

Little-oh
F (n) is o (g(n)) if, for any constant c > 0, there is an integer constant n_0> 0 such that f(n) < c• g(n) for n>= n_0

Little-omega
f(n) is ω (g(n)) if, for any constant c > 0, there is an integer constant n_0> 0 such that f(n) > c• g(n) for n>= n_0

Intuition for Asymptotic Notation Big-Oh

f (n) is O (g(n)) if f(n) is asymptotically less than or equal to g(n)

Big-Omega
f (n) is Ω (g (n)) if f(n) is asymptotically greater than or equal to g(n)

Big-Theta
f (n) is Θ (g (n)) if f(n) is asymptotically equal to g(n)

Little-oh
f (n) is o (g (n)) if f(n) is asymptotically strictly less than g(n)

Little-omega
f (n) is ω (g (n)) if is asymptotically strictly greater than g(n)

Examples

$5n^2\Omega$ is (n^2)
F (n) is Ω (g(n)) if there is a constant c > 0 and an integer constant n_0>=1 such that f(n)>= c• g(n) for n>= n_0 let c = 5 and n_0= 1

$5n^2\Omega$ is (n)
f(n) is Ω (g(n)) if there is a constant c > 0 and an integer constant n_0>=1 such that f(n) >=c• g(n) for n>= n_0 let c = 1 and n_0= 1

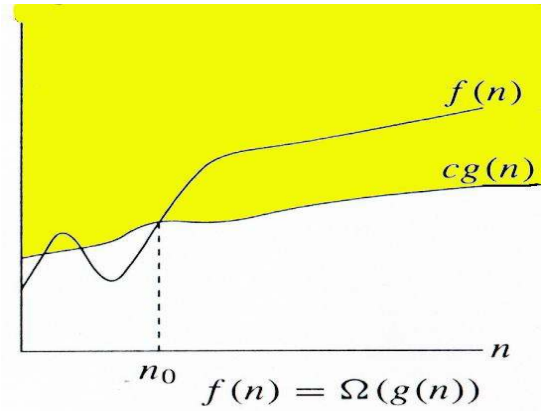
5 n^2 ω (n)
F (n) is ω (g(n)) if, for any constant c > 0, there is an integer constant n_0> 0 such that f(n) > c• g(n) for n >=n_0 need $5n_0^2$> c•n0given c, the $n_0$that satisfies this is n_0> c/5 > 0.

(c) Ω **Notation**

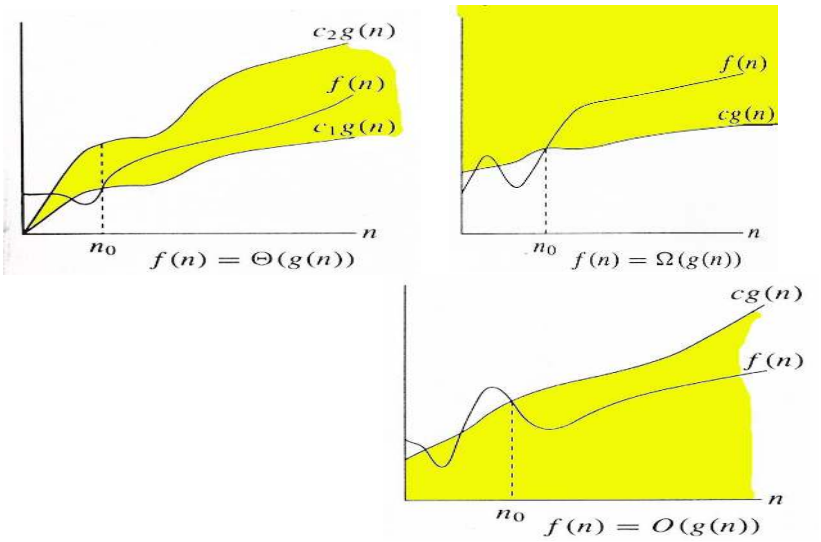
For function *g* (*n*), we define Ω (*g*(*n*)), big-Omega of *n*, as the set:
Ω (*g* (*n*)) ={ *f*(*n*) :∃ positive constants *c* and n_0,such that ∀*n* >=n_0,we have0 <=c*g*(*n*)<=*f*(*n*) }

Intuitively: Set of all functions whose *rate of growth* is the same as or higher than that of *g*(*n*). *g*(*n*) is an *asymptotic lower bound* for *f*(*n*).

$$f(n) = \Theta(g(n)) \; f(n) \Rightarrow f(n) = \Omega(g(n)) \; \Theta(g(n)) \subset \Omega(g(n))$$



Relations Between Θ , *O*, Ω



Theorem : For any two functions $g(n)$ and $f(n)$, $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

i.e., $\Theta(g(n)) = O(g(n))\ (g(n))$
Asymptotically tight bounds are obtained from asymptotic upper and lower bounds.

Running Times

"Running time is $O(f(n))$" Worst case is $O(f(n))$
$O(f(n))$ bound on the worst-case running time $\Rightarrow O(f(n))$ bound on the running time of every input.
$\Theta(f(n))$ bound on the worst-case running time $\Theta(f(n))$ bound on the running time of every input.
"Running time is $(f(n))$" Best case is $(f(n))$ Can still say "Worst-case running time is $\Omega(f(n))$". Means worst-case running time is given by some unspecified function $g(n) \in \Omega(f(n))$.

Asymptotic Notation in Equations

We can use asymptotic notation in equations to replace expressions containing lower-order terms.

Example: $4n_3 + 3n_2 + 2n + 1 = 4n_3 + 3n_2 + (n)$
$= 4n_3 + (n_2) = (n_3)$.
$\Theta(f(n))$ always stands for an *anonymous function* $g(n) \in \Theta(f(n))$

***Little o*-notation**
For a given function $g(n)$, the set little-o: $\forall o(g(n)) = \{f(n): \forall c > 0, \exists\ n_0 > 0$ such that $\forall n >= n_0$, we have $0 <= f(n) < cg(n)\}$.
$F(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity:
$lim\ [f(n) / g(n)] = 0$
$n \to \alpha$
$g(n)$ is an *upper bound* for $f(n)$ that is not asymptotically tight.

Little ω –notation

For a given function $g(n)$, the set little-omega: $(g(n))= \{f(n): \forall c> 0, \exists\ n0> 0$such that $\forall n >= n_0$, we have $0 <= cg(n) < f(n) \}$.
$F(n)$ becomes arbitrarily large relative to $g(n)$as n approaches infinity:
$lim\ [f(n) / g(n)] = \alpha$.
$n-> \alpha$
$g(n)$ is a *lower bound* for $f(n)$that is not asymptotically tight.

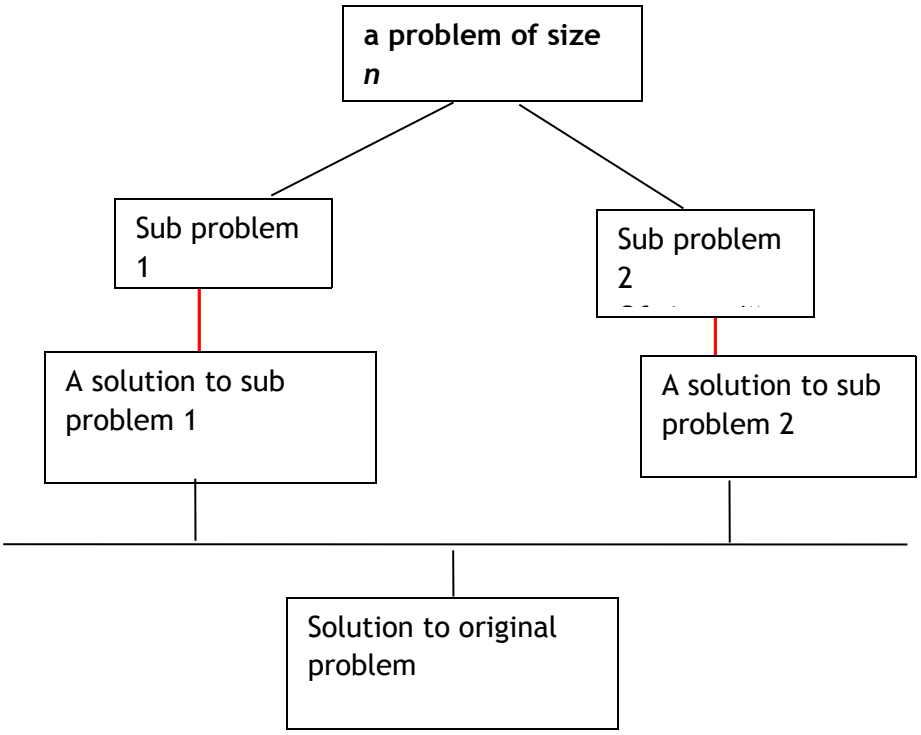
……..

Chapter-2
Divide and Conquer

2.1 General Method

Definition

Divide the problem into a number of sub problems; conquer the sub problems by solving them recursively. If the sub problem sizes are small enough, solve the sub problems recursively, and then combine these solutions to create a solution to the original problem.



Divide-and conquer is a general algorithm design paradigm

Divide: divide the input data S in two or more disjoint subsets S_1, S_2,
Recursively: solve the sub problems recursively
Conquer: combine the solutions for S_1, $S_{2...}$ into a solution for S
The base case for the recursion is sub problems of constant size. Analysis can be done using recurrence equations

For a given function $g(n)$, the set little-omega: $(g(n))= \{f(n): \forall c> 0, \exists\ n0> 0$such that $\forall n >= n_0$, we have $0 <= cg(n) < f(n) \}$.
$F(n)$ becomes arbitrarily large relative to $g(n)$as n approaches infinity:
$lim\ [f(n) / g(n)] = \alpha$.
$n-> \alpha$
$g(n)$ is a *lower bound* for $f(n)$that is not asymptotically tight.

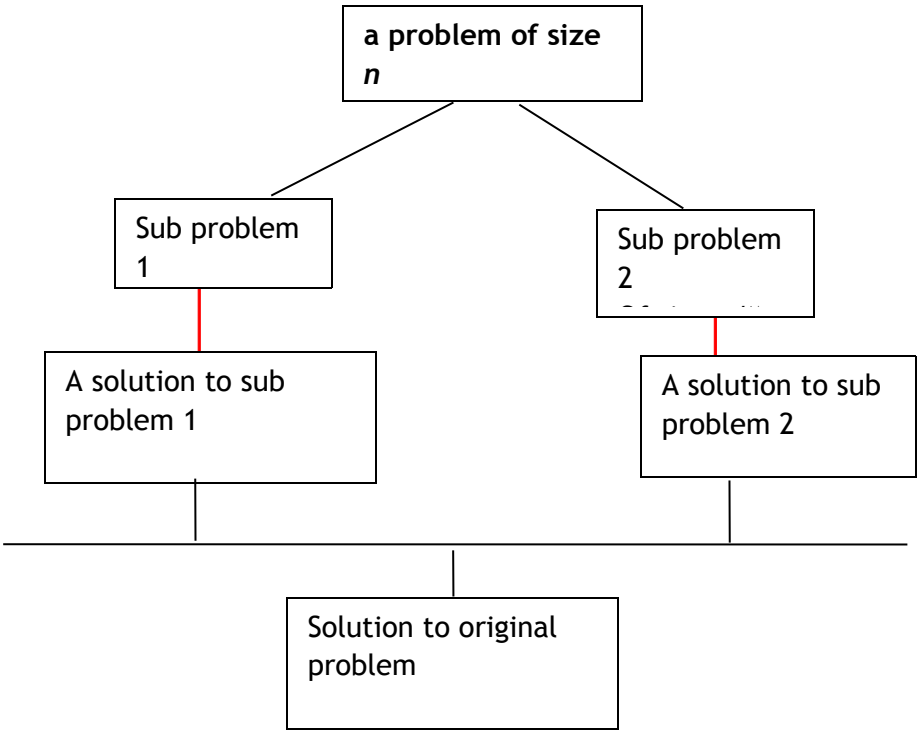
……..

Chapter-2
Divide and Conquer

2.1 General Method

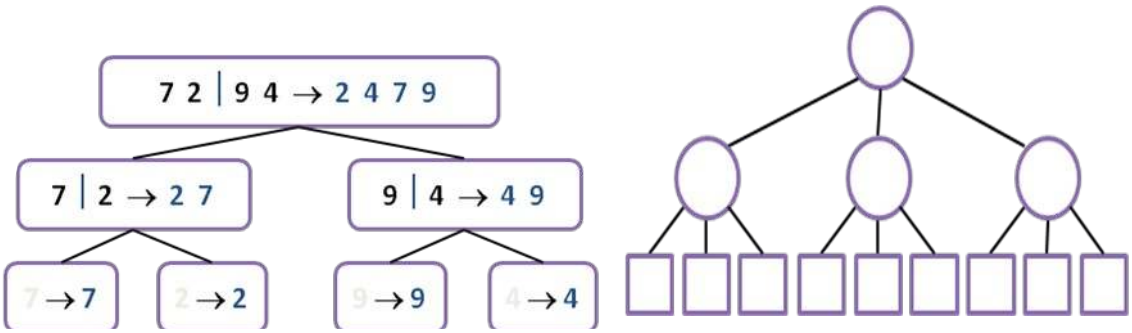
Definition

Divide the problem into a number of sub problems; conquer the sub problems by solving them recursively. If the sub problem sizes are small enough, solve the sub problems recursively, and then combine these solutions to create a solution to the original problem.



Divide-and conquer is a general algorithm design paradigm

Divide: divide the input data S in two or more disjoint subsets S_1, S_2,
Recursively: solve the sub problems recursively
Conquer: combine the solutions for S_1, $S_{2...}$ into a solution for S
The base case for the recursion is sub problems of constant size. Analysis can be done using recurrence equations



Algorithm

Algorithm D-and-C (n: input size)
{
if n ≤ n0 /* small size problem*/
Solve problem without further sub-division;
Else
 {
  Divide into m sub-problems;
  Conquer the sub-problems by solving them
  Independently and recursively; /* D-and-C(n/k) */
 Combine the solutions;
 }
}

Advantage

Straight forward and running times are often easily determined

2.2 Divide-and-Conquer Recurrence Relations

Suppose that a recursive algorithm divides a problem of size n into parts, where each sub-problem is of size n/b. Also, suppose that a total number of g(n) extra operations are needed in the conquer step of the algorithm to combine the solutions of the sub-problems into a solution of the original problem. Let f (n) is the number of operations required to solve the problem of size n. Then f satisfies the recurrence relation and it is called divide-and-conquer recurrence relation.

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

The computing time of Divide and conquer is described by recurrence relation.

$$T(n) = \begin{cases} g(n) & \text{where n small} \\ T(n1) + T(n2) + \ldots\ldots\ldots + T(n_k) + f(n) & \text{other wise} \end{cases}$$

T (n) is the time for Divide and Conquer on any input of size n and g (n) is the time to compute the answer directly for small inputs. The function of f (n) is the time for dividing P combining solutions to sub problems. For divide-and-conquer-based algorithms that produce sub problems of the same type as the original problem, then such algorithm described using recursion.

The complexity of many divide-and-conquer algorithms is given by recurrence of the form.

$$T(n) = \begin{cases} T(1) & n=1 \end{cases}$$

{a T(n/b) + f(n) n>1 where a and b are known constants, and n is a power of b ($n=b^k$). One of the methods for solving any such recurrence relation is called substitution method.

Examples

If a=2 and b=2. Let T (1) =2 and f (n)=n. Than
T (n) = 2T (n/2) +n
=2[2 T (n/4) + n/2] +n
=4 T (n/4) +2n
=4[2 T (n/8) +n/4] + 2n
=8 T (n/8) + 3n
.
.
.
In general, $T(n) = 2^i T(n/2^i) + in$, for any $\log_2 n >= i >= 1$. In Particular, then $T(n) = 2^{\log_2 n} T(n/2^{\log_2 n}) + n \log_2 n$ corresponding to choice of
$i = \log_2 n$. Thus, $T(n) = n T(1) + n \log_2 n = n \log_2 n + 2n$.

2.3 Divide and Conquer Applications

2.3.1 Min and Max
The minimum of a set of elements: The first order statistic i = 1
The maximum of a set of elements: The n^{th} order statistic i = n
The median is the "halfway point" of the set I = (n+1)/2, is unique
When n is odd
$i = \lfloor (n+1)/2 \rfloor = n/2$ (lower median) and $\lceil (n+1)/2 \rceil = n/2+1$ (upper median), when n is even
Finding Minimum or Maximum
Alg: MINIMUM (A, n)

```
min ← A[1]
for i ← 2 to n
  do if min > A[i]
    then min ← A[i]
return min
```

How many comparisons are needed?
n – 1: each element, except the minimum, must be compared to a smaller element at least once. The same number of comparisons is needed to find the maximum. The algorithm is optimal with respect to the number of comparisons performed.

Simultaneous Min, Max

Find min and max independently
Use n – 1 comparisons for each ⇒total of 2n – 2

At most 3n/2 comparisons are needed. Process elements in pairs. Maintain the minimum and maximum of elements seen so far. Don't compare each element to the minimum and maximum separately. Compare the elements of a pair to each other. Compare the larger element to the maximum so far, and compare the smaller element to the minimum so far. This leads to only 3 comparisons for every 2 elements.

Analysis of Simultaneous Min, Max

Setting up initial values:
n is odd: compare the first two elements, assign the smallest one to min and the largest one to max
n is even:
Total number of comparisons:

n is odd: we do 3(n-1)/2 comparisons
n is even: we do 1 initial comparison + 3(n-2)/2 more comparisons = 3n/2 - 2 comparisons

Example

1. n = 5 (odd), array A = {2, 7, 1, 3, 4}
    1. Set min = max = 2
    2. Compare elements in pairs:

$1 < 7 \Rightarrow$ compare 1 with min and 7 with max
$\Rightarrow$ min = 1, max = 7 } 3-comparisions
$3 < 4 \Rightarrow$ compare 3 with min and 4 with max
$\Rightarrow$ min = 1, max = 7
3(n-1)/2 = 6 comparisons } 3-comparisions

2. n = 6 (even), array A = {2, 5, 3, 7, 1, 4}
    1. Compare 2 with 5: 2 < 5
    2. Set min = 2, max = 5
    3. Compare elements in pairs:

$3 < 7 \Rightarrow$ compare 3 with min and 7 with max
$\Rightarrow$ min = 2, max = 7 } 3-comparisons

$1 < 4 \Rightarrow$ compare 1 with min and 4 with max
$\Rightarrow$ min = 1, max = 7
3n/2 - 2 = 7 comparisons
$\Rightarrow$ min = 1, max = 7 } 3-comparisions

2.3.2 Binary Search

The basic idea is to start with an examination of the middle element of the array. This will lead to 3 possible situations: If this matches the target K, then search can terminate successfully, by printing out the index of the element in the array. On the other hand, if K<A[middle], then search can be limited to elements to the left of A[middle]. All elements to

the right of middle can be ignored. If it turns out that K >A[middle], then further search is limited to elements to the right of A[middle]. If all elements are exhausted and the target is not found in the array, then the method returns a special value such as –1.

1st Binary Search function:

```
int BinarySearch (int A[ ], int n, int K)
{
int L=0, Mid, R= n-1;
while (L<=R)
{
Mid = (L +R)/2;
if ( K= =A[Mid] )
return Mid;
else if ( K > A[Mid] )
L = Mid + 1;
else
R = Mid – 1;
}
return –1 ;}
```

Let us now carry out an Analysis of this method to determine its time complexity. Since there are no "for" loops, we cannot use summations to express the total number of operations. Let us examine the operations for a specific case, where the number of elements in the array n is 64. When n= 64 Binary Search is called to reduce size to n=32

When n= 32 Binary Search is called to reduce size to n=16
When n= 16 Binary Search is called to reduce size to n=8
When n= 8 Binary Search is called to reduce size to n=4
When n= 4 Binary Search is called to reduce size to n=2
When n= 2 Binary Search is called to reduce size to n=1.

Thus we see that Binary Search function is called 6 times (6 elements of the array were examined) for n =64. Note that 64 = 26.Also we see that the Binary Search function is called 5 times (5 elements of the array were examined) for n = 32. Note that 32 = 25 Let us consider a more general case where n is still a power of 2. Let us say n = 2k.

Following the above argument for 64 elements, it is easily seen that after k searches, the while loop is executed k times and n reduces to size 1. Let us assume that each run of the while loop involves at most 5 operations. Thus total number of operations: 5k. The value of k can be determined from the expression $2^k = n$ .Taking log of both sides Log 2^k = log n Thus total number of operations = 5 log n. We conclude that the time complexity of the Binary search method is O (log n), which is much more efficient than the Linear Search method.

2nd method Binary Search function

Binary-Search (*A; p; q; x)*
1. if p > *q return -1;*
2. *r = b (p + q)=2 c*
3. if *x = A[r] return r*
4. else if *x < A[r] Binary-Search(A; p; r; x)*
5. else Binary-Search(*A; r + 1; q; x)*

² The initial call is Binary-Search (A; 1; n; x).

List [0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11]

4	8	19	25	34	39	45	48	66	75	89	95

List length=12

Search list

[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11]

4	8	19	25	34	39	45	48	66	75	89	95

mid

Search list, list[0]….list[1]

Middle element

Mid=left + right/2

4	8	19	25	34	39	45	48	66	75	89	95

Sorted list for a binary search

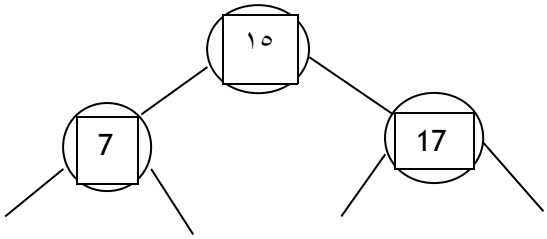
Value of first, last, mid, no of comparisons for search item 89

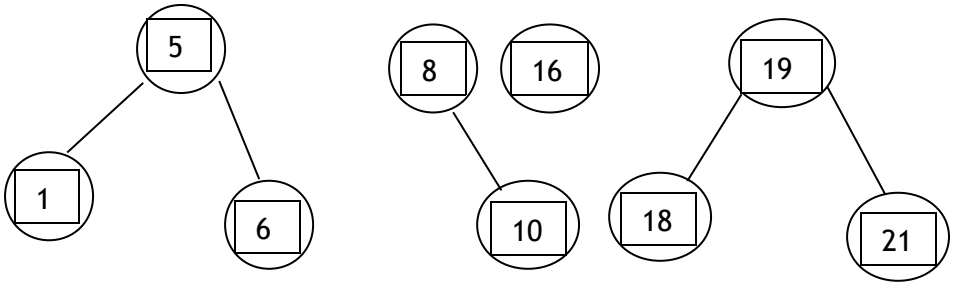
Iteration	First	Last	Mid	Last[mid]	No.of Comparisons
1	0	11	5	39	2
2	6	11	8	66	2
3	9	11	10	89	1(found it is true)

Binary search tradeoffs

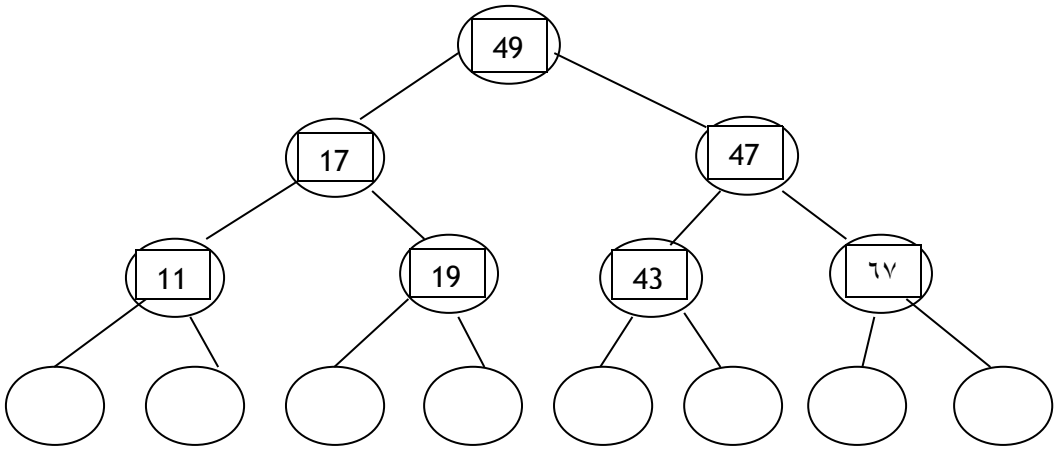
Benefit: More efficient than linear search (for array of N elements performs at most $\log_2 N$ comparisons)

Disadvantages: requires that array elements to be sorted.





Full and balanced Binary search tree



Logarithmic Time Complexity of Binary Search

Our analysis shows that binary search can be done in time proportional to the *log* of the number of items in the list this is considered *very fast* when compared to linear or polynomial algorithms .The table to the right compares the number of operations that need to be performed for algorithms of various time complexities. The computing time binary search by best, average and

Worst cases:
Successful searches
Θ (1) best, Θ (log n) average
Θ (log n) worst
Unsuccessful searches
Θ (log n) for best , average and worst case

2.3.2 Merge Sort Algorithm

Divide: Divide the n-element sequence into two subsequences of n/2 elements each.

Conquer: Sort the two subsequences recursively using merge sort.

Combine: Merge the two sorted sequences.

How to merge two sorted sequences:

We have two sub arrays A [p...q] and A [q+1..r] in sorted order. Merge sort algorithm merges them to form a single sorted sub array that replaces the current sub array A [p...r].

To sort the entire sequence A [1...n], make the initial call to the procedure MERGE-SORT(A,1,n).

```
MERGE-SORT (A, p, r)
{
    1.    IF p<r                    //Check for base case
    2.    THEN q=FLOOR[(p+r)/2]     //Divide step
    3.    MERGESORT (A,p,q)         //Conquer step
    4.    MERGESORT(A,q+1,r)        //Conquer step
    5.    MERGE (A, p, q, r)        //Conquer step.
}
```

The pseudo code of the MERGE procedure is as follow:

MERGE (A, p, q, *and* r)

$n_1 \leftarrow q - p + 1$

$n_2 \leftarrow r - q$

Create arrays L [1 . . . n_1 + 1] and R[1 . . n_2 + 1]

FOR $i \leftarrow 1$ TO n_1

DO L[i] $\leftarrow$ A [$p + i - 1$]

FOR $j \leftarrow 1$ TO n_2

DO R[j] $\leftarrow$ A [$q + j$]

L [$n_1 + 1$] $\leftarrow \infty$

R [$n_2 + 1$] $\leftarrow \infty$

$i \leftarrow 1$

$j \leftarrow 1$

FOR $k \leftarrow p$ TO r

DO IF L [i] $\leq$ R [j]

THEN A [k] $\leftarrow$ L [i]

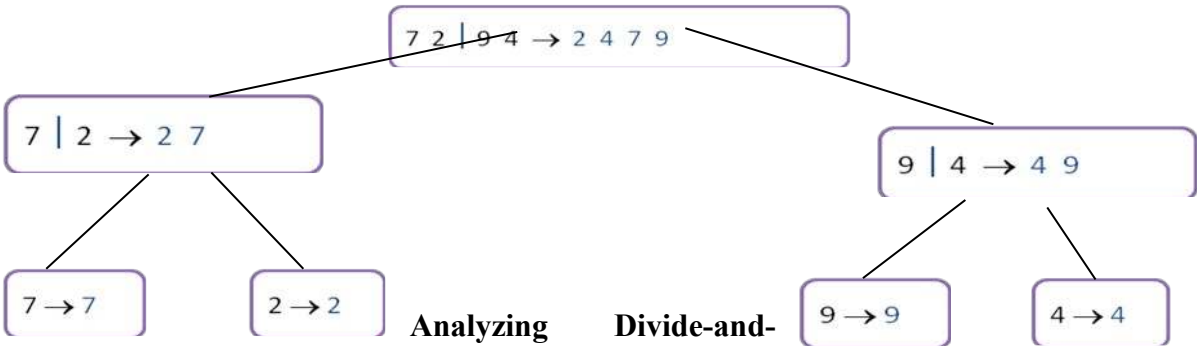
$i \leftarrow i + 1$

ELSE A[k] $\leftarrow$ R[j]

$j \leftarrow j + 1$

Merge-Sort Tree

An execution of merge-sort is depicted by a binary tree each node represents a recursive call of merge-sort and stores unsorted sequence before the execution and its partition sorted sequence at the end of the execution. The root is the initial call. The leaves are calls on subsequences of size 0 or 1.



Analyzing Divide-and-Conquer Algorithm

When an algorithm contains a recursive call to itself, its running time can be described by a recurrence equation or recurrence which describes the running time.

Analysis of Merge-Sort:

The height h of the merge-sort tree is O (log n)

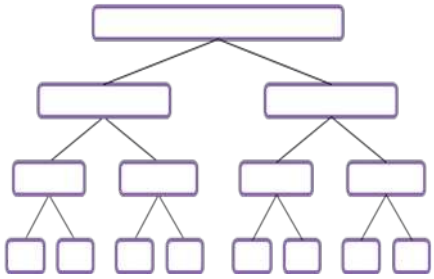
- at each recursive call we divide in half the sequence,

The overall amount or work done at the nodes of depth i is O (n)

- we partition and merge 2^i sequences of size $n/2^i$
- we make 2^{i+1} recursive calls

Thus, the total running time of merge-sort is O (n log n)

Depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
...	…	…



Recurrence

If the problem size is small enough, say n<=c for some constant c, the straightforward solution takes constant time, can be written as θ (1). If we have *a* sub problems, each of which is *1/b* the size of the original. D (n) time to divide the problem and C (n) time to combine the solution.

The recurrence is

$$T(n) = \begin{cases} \theta(1) & \text{if } n <= c \\ a\,T(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Divide: The divide step computes the middle of the sub array which takes constant time, D(n)=θ(1)

Conquer: We recursively solve two sub problems, each of size n/2, which contributes 2T(n/2) to the running time.

Combine: Merge procedure takes θ (n) time on an n-element sub array. C (n)=θ(n)

The recurrence is

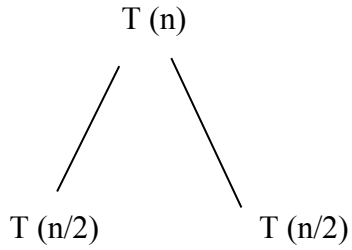
$$T(n) = \begin{cases} \theta(1) & \text{if } n=1 \\ 2T(n/2) + \theta(n) & \text{if } n>1 \end{cases}$$

Let us rewrite the recurrence

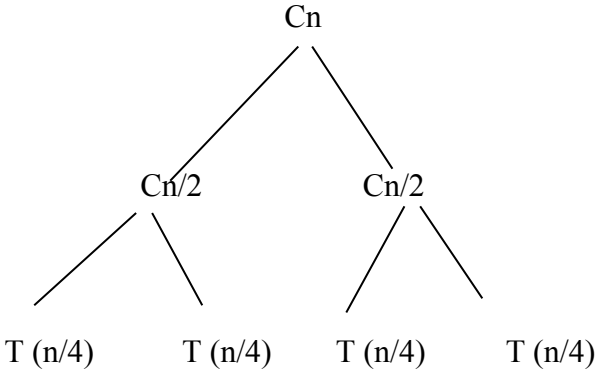
$$T(n) = \begin{cases} C & \text{if } n=1 \\ 2\,T(n/2) + cn & \text{if } n>1 \end{cases}$$

C represents the time required to solve problems of size 1

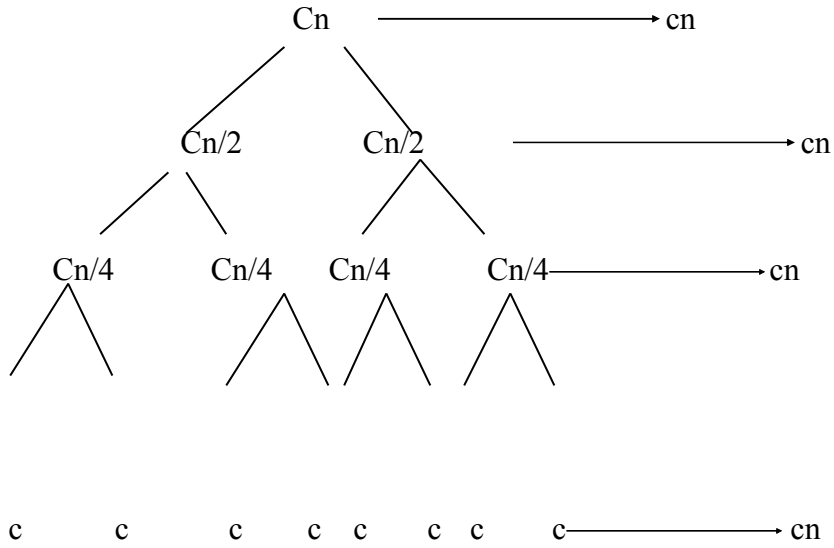
A Recursion Tree for the Recurrence



A Recursion Tree for the Recurrence



A Recursion Tree for the Recurrence



A Recursion Tree for the Recurrence in the above recursion tree, each level has cost *cn*. The top level has cost *cn*. The next level down has 2 sub problems; each contributing cost *cn*/2.The next level has 4 sub problems, each contributing cost *cn*/4. Each time we go down one level, the number of sub problems doubles but the cost per sub problem halves. Therefore, cost per level stays the same. The height of this recursion tree is log *n* and there are log *n* + 1 levels. Total Running Time of a tree for a problem size of 2^i has log $2^i + 1 = i + 1$ levels. The fully expanded tree recursion tree has log n+1 levels. When n=1 than 1 level log 1=0, so correct number of levels log n+1. Because we assume that the problem size is a power of 2, the next problem size up after 2^i is $2^i + 1$. A tree for a problem size of $2^i + 1$ has one more level than the size-2^i tree implying $i + 2$ levels. Since log $2^i + 1 = i + 2$, we are done with the inductive argument. Total cost is sum of costs at each level of the tree. Since we have log *n* +1 levels, each costing *cn*, the total cost is *cn* log n + *cn*. Ignore low-order term of *cn* and constant coefficient *c*, and we have, Θ (*n* log *n*). The fully expanded tree has lg n +1 levels and each level contributes a total cost of cn. Therefore T (n)= cn log n + cn = θ(nlog n). Growth of Functions We look at input sizes large enough to make only the order of growth of the running time relevant.

2.3.4 Divide and Conquer: Quick Sort

Pick one element in the array, which will be the *pivot*. Make one pass through the array, called a *partition* step, re-arranging the entries so that, entries smaller than the pivot are to the left of the pivot. Entries larger than the pivot are to the right. Recursively apply quick sort to the part of the array that is to the left of the pivot, and to the part on its right. No merge step, at the end all the elements are in the proper order. Choosing the Pivot some fixed element: e.g. the first, the last, the one in the middle. Bad choice - may turn to be the smallest or the largest element, and then one of the partitions will be empty. Randomly chosen (by random generator) still a bad choice. The median of the array (if the array has N numbers, the median is the [N/2] largest number). This is difficult to compute - increases the complexity. The median-of-three choice: take the first, the last and the middle element. Choose the median of these three elements.

Quick Sort:
Quick sort is introduced by Hoare in the year 1962.
All elements to the left of pivot are smaller or equal than pivot, and
All elements to the right of pivot are greater or equal than pivot
Pivot in correct place in sorted array/list
Divide: Partition into sub arrays (sub-lists)
Conquer: Recursively sort 2 sub arrays
Combine: Trivial
Problem: Sort *n* keys in non-decreasing order
Inputs: Positive integer *n*, array of keys S indexed from 1 to n
Output: The array S containing the keys in non-decreasing order.

Quick sort (low, high)
1. if high > low
2. then partition(low, high, pivotIndex)
3. quick sort(low, pivotIndex -1)
4. quick sort(pivotIndex +1, high)

Partition array for Quick sort
partition (low, high, pivot)

1. pivotitem = S [low]
2. k=low
3. for j = low +1 to high
4. do if S [j] < pivotitem
5. then k = k + 1
6. exchange S [j] and S [k]
7. pivot = k
8. exchange S[low] and S[pivot]

Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
Divide: pick a random element x (called pivot) and partition S into
L elements less than x
E elements equal x
G elements greater than x
Recur: sort L and G
Conquer: join L, E and G

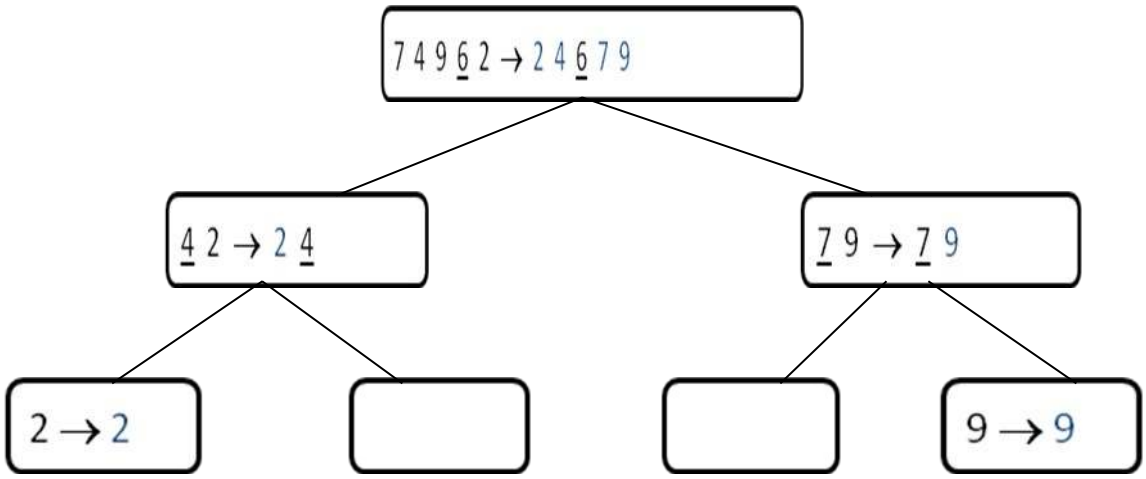
<u>Partition</u>

We partition an input sequence as follows:
-We remove, in turn, each element y from S and
-We insert y into L, E or G, depending on the result of the comparison with the pivot x
Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time. Thus, the partition step of quick-sort takes $O(n)$ time

Quick-Sort Tree

An execution of quick-sort is depicted by a binary tree. Each node represents a recursive call of quick-sort and stores. Unsorted sequence before the execution and its pivot. Sorted sequence at the end of the execution

- The root is the initial call
- The leaves are calls on subsequences of size 0 or 1



Partitioning Array

Given a pivot, partition the elements of the array such that the resulting array consists of:

1. One sub-array that contains elements >= pivot
2. Another sub-array that contains elements < pivot

The sub-arrays are stored in the original data array. Partitioning loops through, swapping elements below/above pivot.

There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:

40	20	10	80	60	50	7	30	100

Partitioning result

7	20	10	30	40	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

<=data [pivot] >data [pivot]

Quick sort: Worst Case

Assume first element is chosen as pivot. Assume we get array that is already in order:

Pivot_index=0

2	4	10	12	13	50	57	63	100

[0] [1] [2] [3] [4] [5] [6] [7] [8]

Complexity of Quick Sort

If we have an array of equal elements, the array index will never increment i or decrement j, and will do infinite swaps. i and j will never cross.

Worst Case: O (N^2)

This happens when the pivot is the smallest (or the largest) element. Then one of the partitions is empty, and we repeat recursively the procedure for N-1 elements.

Worst-case Running Time

The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element. One of *L* and *G* has size *n* - 1 and the other has size 0

The running time is proportional to the sum

n + (*n* - 1) + … + 2 + 1

Thus, the worst-case running time of quick-sort is *O* (n^2)

Depth	time
0	*N*
1	*n* - 1
...	...
n – 1	1

Worst-Case Analysis

The pivot is the smallest (or the largest) element

T (N) = T (N-1) + cN, N > 1

Telescoping:

T (N-1) = T (N-2) + c (N-1)

T (N-2) = T (N-3) + c (N-2)

T (N-3) = T (N-4) + c (N-3)

……...

T (2) = T (1) + c.2

T (N) + T (N-1) + T (N-2) + … + T (2) =

= T (N-1) + T (N-2) + … + T (2) + T (1) +

C (N) + c (N-1) + c (N-2) + … + c.2

T (N) = T (1) +

c times (the sum of 2 thru N)

= T (1) + c (N (N+1) / 2 -1) = O (N^2)

Average-case: O (N logN)

Best-case: O (N logN)

The pivot is the median of the array, the left and the right parts have same size. There are logN partitions, and to obtain each partition we do N comparisons (and not more than N/2 swaps). Hence the complexity is O (NlogN).

Best case Analysis:

T (N) = T (i) + T (N - i -1) + cN

The time to sort the file is equal to the time to sort the left partition with i elements, plus the time to sort the right partition with N-i-1 elements, plus the time to build the partitions.

The pivot is in the middle

T (N) = 2 T (N/2) + cN

Divide by N: T (N) / N = T (N/2) / (N/2) + c

Telescoping:

T (N) / N = T (N/2) / (N/2) + c

T (N/2) / (N/2) = T (N/4) / (N/4) + c

T (N/4) / (N/4) = T (N/8) / (N/8) + c

……

T (2) / 2 = T (1) / (1) + c

Add all equations:

T (N) / N + T (N/2) / (N/2) + T (N/4) / (N/4) + …. + T (2) / 2 =

= (N/2) / (N/2) + T (N/4) / (N/4) + … + T (1) / (1) + c.logN

After crossing the equal terms:

T (N)/N = T (1) + c * LogN

T (N) = N + N * c * LogN = O (NlogN)

Advantages and Disadvantages:

Advantages

One of the fastest algorithms on average

Does not need additional memory (the sorting takes place in the array - this is called in-place processing)

Disadvantages

The worst-case complexity is O (N^2)

Applications

Commercial applications

Quick Sort generally runs fast

No additional memory

The above advantages compensate for the rare occasions when it runs with O (N^2)

2.3.5 Divide and Conquer: Selection Sort

Definition: First find the smallest in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continue in this way until the entire array is sorted.

Selection sort is:

-The simplest sorting techniques.

-a good algorithm to sort a small number of elements

-an incremental algorithm – induction method

Selection sort is Inefficient for large lists.

Incremental algorithms → process the input elements one-by-one and maintain the solution for the elements processed so far.

Let A [1…n] be an array of n elements. A simple and straightforward algorithm to sort the entries in A works as follows. First, we find the minimum element and store it in A [1]. Next, we find the minimum of the remaining n-1 elements and store it in A [2]. We continue this way until the second largest element is stored in A [n-1].

Input: A [1…n];

Output: A [1…n] sorted in non-decreasing order;

1. for $i \leftarrow 1$ to n-1
2. $k \leftarrow i$;
3. for $j \leftarrow i+1$ to n
4. if $A[j]<A[k]$ then $k \leftarrow j$;
5. end for;
6. if $k \neq i$ then interchange $A[i]$ and $A[k]$;
7. end for;

Procedure of selection sort

i.Take multiple passes over the array.
ii.Keep already sorted array at high-end.
iii.Find the biggest element in unsorted part.
iv.Swap it into the highest position in unsorted part.
v.Invariant: each pass guarantees that one more element is in the correct position (same as bubble sort) a lot fewer swaps than bubble sort!

12	8	3	21	99	1	Start- unsorted

Pass 1

12	8	3	21	99	1
12	8	3	21	1	99

Pass 2

12	8	3	21	1	99
12	8	3	1	21	99

Pass 3

12	8	3	1	21	99

1	8	3	12	21	99

Pass 4

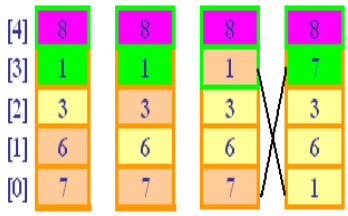
1	8	3	12	21	99
1	3	8	12	21	99

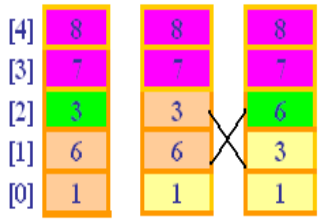
Pass 5

1	3	8	12	21	99

1	3	8	12	21	99	Sorted

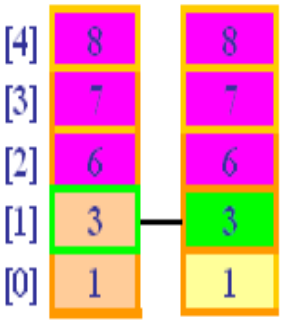
Example Execution of selection sort Tracing

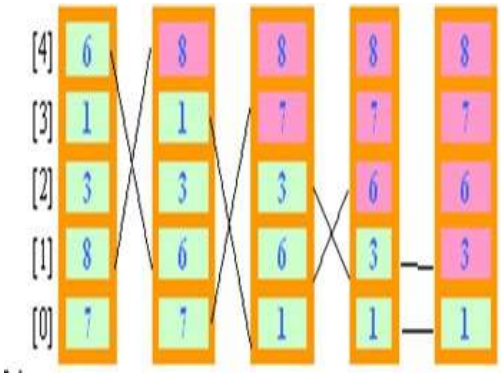




Pass 2
Last 3
Largest index 0, 0, 0
P=1, 2, 3

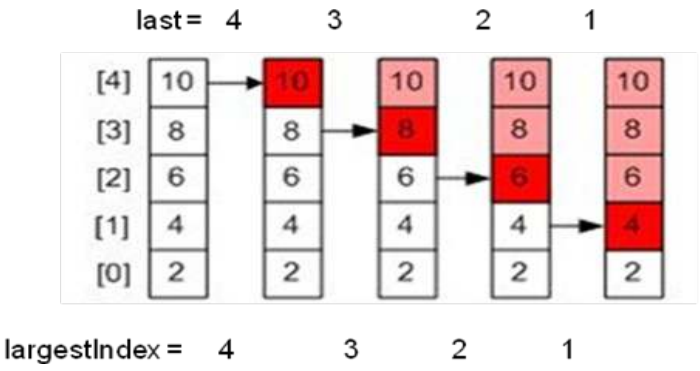
Pass 3
Last 2
largest index 0, 1
p=1, 2





Pass 4
last = 1
Largest Index = 0, 1
p = 1

Selection Sort Implementation for Best Case [2 4 6 8 10]



Selection Sort Analysis

For an array with size n, the external loop will iterate from *n-1 to 1.*
for (int last = n-1; last>=1; --last) For each iteration, to find the largest number in sub array, the number of comparison inside the internal loop must is equal to the value of last. for (int p=1;p <=last; ++p) Therefore the total comparison for Selection Sort in each iteration is *(n-1) + (n-2) + ... 2 + 1*. Generally, the number of comparisons between elements in Selection Sort can be stated as follows:

$$(n-1)+(n-2)+\ldots\ldots+2+1=\frac{n(n-1)}{2}=O(n^2)$$

Selection Sort – Algorithm Complexity

Time Complexity for Selection Sort is the same for all cases - worst case, best case or average case O (n^2). The number of comparisons between elements is the same. The efficiency of Selection Sort does not depend on the initial arrangement of the data.

Alg.: SELECTION-SORT *(A)*	*Cost*	*times*
1. n ← length[A]	c1	1
2. for j ← 1 to n – 1	c2	n
3. do smallest ← j	c3	n-1
4. for i ← j + 1 to n		$\sum_{j=1}^{n-1}(n-j+1)$
5. do if A[i] < A[smallest]		$\sum_{j=1}^{n-1}(n-j)$
6.then smallest ← i		$\sum_{j=1}^{n-1}(n-j)$
7. exchange A[j] ↔ A[smallest]	c7	

2.11 Stressen's Matrix Multiplication:

Multiplication of Large Integer

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

A = 12345678901357986429 B = 87654321284820912836

The grade-school algorithm:

a_1 a_2 … a_n
b_1 b2 … bn
(d_{10}) $d_{11}d_{12}$ … d_{1n}
(d_{20}) $d_{21}d_{22}$ … d_{2n}
… … … … … … … …
(d_{n0}) $d_{n1}d_{n2}$ … d_{nn}

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

First Divide-and-Conquer Algorithm

A small example: A ∗ B where A = 2135 and B = 4014

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

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

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

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

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

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

M (n) = 4 M (n/2),
M (1) = 1
Solution: $M(n) = n^2$

Second Divide-and-Conquer Algorithm:

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

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

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

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

Recurrence for the number of multiplications M (*n*):

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

Solution: $M(n) = 3^{\log 2n} = n^{\log 23} \approx n^{1.585}$

Example of Large-Integer Multiplication:

2135 ∗ 4014

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

= (21*40)*10^4 + c1*10^2 + 35*14
where c1 = (21+35)*(40+14) - 21*40 - 35*14, and
21*40 = (2*10 + 1) * (4*10 + 0)
= (2*4)*10^2 + c2*10 + 1*0
where c2 = (2+1)*(4+0) - 2*4 - 1*0, etc.
This process requires 9 digit multiplications as opposed to 16.

Conventional Matrix Multiplication:
Brute-force algorithm

$$\begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} * \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix}$$

$$\begin{pmatrix} a_{00} * b_{00} + a_{01} * b_{10} & a_{00} * b_{01} + a_{01} * b_{11} \\ a_{10} * b_{00} + a_{11} * b_{10} & a_{10} * b_{01} + a_{11} * b_{11} \end{pmatrix}$$

8 multiplications, 4 additions Efficiency class in general: $\Theta(n^3)$

Strassen's Matrix Multiplication

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

$$\begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} * \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix}$$

$$= \begin{pmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{pmatrix}$$

$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$
$m_2 = (a_{10} + a_{11}) * b_{00}$
$m_3 = a_{00} * (b_{01} - b_{11})$
$m_4 = a_{11} * (b_{10} - b_{00})$
$m_5 = (a_{00} + a_{01}) * b_{11}$
$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$
$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$

7 multiplications, 18 additions

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

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

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

Formulas for Strassen's Algorithm

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

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

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

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

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

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

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

Analysis of Strassen's Algorithm

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

Number of multiplications:

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

Solution: $M(n) = 7^{\log 2n} = n^{\log 27} \approx n^{2.807}$ vs. n^3 of brute-force alg.

Algorithms with better asymptotic efficiency are known but they are even more complex and not used in practice.

******

Chapter-3
Greedy Method

3.1 Greedy Technique Definition

Constructs a solution to an *optimization problem* piece by piece through a sequence of choices that are: feasible, i.e. satisfying the constraints locally optimal (with respect to some neighborhood definition) greedy (in terms of some measure), and irrevocable. For some problems, it yields a globally optimal solution for every instance. For most, does not but can be useful for fast approximations. We are mostly interested in the former case in this class.

Generic Algorithm

```
Algorithm Greedy(a,n)
{
//a[1..n] contains the n inputs.
solution:= ∅;
For i:= 1to n do
{
X=select(a);
If Feasible(solution , x) then
solution:= union(solution, x);
}
return solution;
}
```

Applications of the Greedy Strategy

Optimal solutions:

- change making for "normal" coin denominations
- minimum spanning tree (MST)
- single-source shortest paths
- simple scheduling problems
- Huffman codes

Approximations/heuristics:

- Traveling salesman problem (TSP)
- knapsack problem
- other combinatorial optimization problems

Change-Making Problem:
Given unlimited amounts of coins of denominations $d_1 > \ldots > d_m$, give change for amount n with the least number of coins
Example: $d_1 = 25c$, $d_2 = 10c$, d3 = 5c, d4 = 1c and n = 48c

Greedy solution:
Greedy solution is Optimal for any amount and "normal'' set of denominations
Ex: Prove the greedy algorithm is optimal for the above denominations. It may not be optimal for arbitrary coin denominations.

3.2 The Fractional Knapsack Problem

Given a set S of n items, with each item i having b_i - a positive benefit w_i - a positive weight our goal is to Choose items with maximum total benefit but with weight at most W. If we are allowed to take fractional amounts, then this is the **fractional knapsack problem**. In this case, we let x_i denote the amount we take of item i

Objective: maximize

$$\sum_{i \in S} b_i (x_i / w_i)$$

Constraint:

$$\sum_{i \in S} x_i \leq W$$

Algorithm for greedy strategy for knapsack problem:

Algorithm Greedy***Knapsack***(m,n)

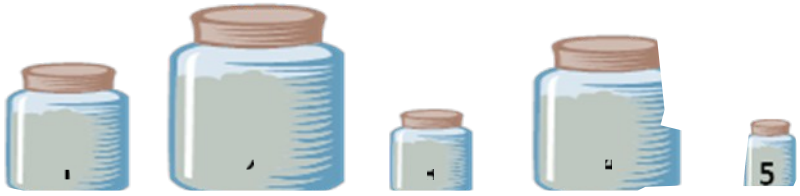
//p[1:n] and w[1:n] contain profits and weights respectively of n objects ordered such that //p[i]/w[i] >=p[i+1]/w[i+1].m is the knapsack size and x[1:n] is the solution vector
{
For i:= 1 to n do x[i]:=0.0; //initialize x
U:=m;

{
If (w[i]>U) then break;
x[i]:=1.0; U:=U-w[i];
}
If (i<=n) then x[i]:=U/w[i];
}

Example model-1

In this model items are arranged by their values, maximum selected first, process continuous till minimum value. Here given a set S of n items, with each item i having b_i - a positive benefit w_i - a positive weight here our goal is to Choose items with maximum total benefit but with weight at most W.

Items:



Weight:	4 ml	8 ml	2 ml	6 ml	1 ml
Benefit:	Rs.12	Rs.32	Rs.40	Rs.30	Rs.50
Value:	3	4	20	5	50

(Rs. per ml)

Solutio
- 1 ml of
- 2 ml of
- 6 ml of
- 1 ml of

Knapsack Problem model-2

In this model items are arranged by their weights, lightest weight selected first, process continuous till the maximum weight. You have a knapsack that has capacity (weight) and You have several items $I_1,\ldots,I_n$..Each item I_j has a weight w_j and a benefit b_j.You want to place a certain number of copies of each item I_j in the knapsack so that:

i)The knapsack weight capacity is not exceeded and

ii)The total benefit is maximal.

Example

Item	Weight	Benefit
A	2	60
B	3	75
C	4	90

f (0), f(1)
f (0) = 0. Why? The knapsack with capacity 0 can have nothing in it.
f (1) = 0. There is no item with weight 1.
f (2)
f (2) = 60. There is only one item with weight 60.then choose A.
f (3)
f(3) = MAX $\{b_j + f(w-w_j) \mid I_j$ is an item$\}$.
= MAX {60+f (3-2), 75 + f (3-3)}
= MAX {60 + 0, 75 + 0}
= 75 then Choose B**.**
F (4)
F (4) = MAX $\{b_j + f (w-w_j) \mid I_j$ is an item$\}$.
= MAX {60 + f (4-2), 75 + f (4-3), 90+f (4-4)}
= MAX {60 + 60, 75 + f (1), 90 + f (0)}
= MAX {120, 75, 90}
=120. Then choose A
F (5)
F (5) = MAX $\{b_j + f(w-w_j) \mid I_j$ is an item$\}$.
= MAX {60 + f (5-2), 75 + f (5-3), 90+f (5-4)}

= MAX {60 + f (3), 75 + f (2), 90 + f (1)}
= MAX {60 + 75, 75 + 60, 90+0}
= 135. Then choose A or B.

Result

Optimal knapsack weight is 135. There are two possible optimal solutions:
Choose A during computation of f (5).
Choose B in computation of f (3).
Choose B during computation of f (5).
Choose A in computation of f (2).
Both solutions coincide. Take A and B.

Procedure to solve the knapsack problem

It is Much easier for item I_j, let $r_j = b_j / w_j$. This gives you the benefit per measure of weight and then Sort the items in descending order of r_j .Pack the knapsack by putting as many of each item as you can walking down the sorted list.

Example model-3

I=<I1,I2,I3,I4,I5> W=<5,10,20,30,40> V=<30,20,100,90,160> knapsack capacity W=60, the solution to the fractional knapsack problem is given as:
Initially

Item	Wi	Vi
I1	5	30
I2	10	20
I3	20	100
I4	30	90
I5	40	160

Taking value per weight ratio

Item	wi	vi	Pi=vi/wi
I1	5	30	6.0
I2	10	20	2.0
I3	20	100	5.0
I4	30	90	3.0
I5	40	160	4.0

Arranging item with decreasing order of Pi

Item	wi	vi	Pi=vi/wi
I1	5	30	6.0
I2	20	100	5.0
I3	40	160	4.0
I4	30	90	3.0
I5	10	20	2.0

Filling knapsack according to decreasing value of Pi, max. value = v1+v2+new (v3)=30+100+140=270

3.3 Greedy Method – Job Sequencing Problem

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 given Input set of jobs 1,2,3,4 have sub sets 2^n so 2^4 =16
It can be written as {1},{2},{3},{4},{Ø},{1,2},{1,3},{1,4},{2,3},{2,4},{3,4},{1,2,3}, {1,2,4},{2,3,4},{1,2,3,4},{1,3,4} total of 16 subsets

Problem:
n=4 , P=(70,12,18,35) , d=(2,1,2,1)

Feasible Solution	Processing Sequence	Profit value	Time Line 0	1	2
1	1	70			
2	2	12			
3	3	18			
4	4	35			
1,2	2,1	82			
1,3	1,3 /3,1	88			
1,4	4,1	105			
2,3	3,2 /2,3	30			
3,4	4,3/3,4	53			

We should consider the pair i,j where di <=dj if di>dj we should not consider pair then reverse the order. We discard pair (2, 4) because both having same dead line(1,1) and cannot process same. Time and discarded pairs (1,2,3), (2,3,4), (1,2,4)…etc since processes are not completed within their deadlines. 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

Problem: n jobs, S = {1, 2… n}, each job i has a deadline $d_i \geq 0$ and a profit $p_i \geq 0$. We need one unit of time to process each job and we can do at most one job each time. We can earn the profit p_i if job i is completed by its deadline.

The optimal solution = {1, 2, 4}.

The total profit = 20 + 15 + 5 = 40.

i	1	2	3	4	5
p_i	20	15	10	5	1
d_i	2	2	1	3	3

Algorithm

Step 1: Sort p_i into non-increasing order. After sorting $p_1 \geq p_2 \geq p_3 \geq \ldots \geq p_i$.

Step 2: Add the next job i to the solution set if i can be completed by its deadline. Assign i to time slot [r-1, r], where r is the largest integer such that $1 \leq r \leq d_i$ and [r-1, r] is free.

Step 3: Stop if all jobs are examined. Otherwise, go to step 2.

Time complexity: $O(n^2)$

Example

I	p_i	d_i	
1	20	2	assign to [1, 2]
2	15	2	assign to [0, 1]
3	10	1	Reject

4	5	3	assign to [2, 3]
5	1	3	Reject

solution = {1, 2, 4}
total profit = 20 + 15 + 5 = 40

Greedy Algorithm to Obtain an Optimal Solution

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 27 15 10) (2 1 2 1)

p_1 p_4 p_3 p_2 d_1 d4 d3 d2

J = {1} is a feasible one

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

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

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

J = {1, 4} is optimal

High level description of job sequencing algorithm

```
Procedure greedy job (D, J, n)
// J is the set of n jobs to be completed by their deadlines
{
J:={1};
   for i:=2 to n do
   {
      if  (all jobs in J U{i} can be completed by their deadlines)
     then J:= ← J U {i};
   }
}
```

Greedy Algorithm for Sequencing unit time jobs

Procedure JS(d,j,n)

// d(i) ≥ 1, 1≤ i ≤ n are the deadlines, n ≥ 1. The jobs are ordered such that

// $p_1 \geq p_2 \geq \ldots\ldots \geq p_n$ .J[i] is the ith job in the optimal solution , i ≤ i ≤ k. Also, at termination d[J[i]]≤ d [J[i+1]] ,1 ≤ i ≤ k

{

d[0]:=J[0]:=0; //initialize and J(0) is a fictious job with d(0) = 0 //

J[1]:=1; //include job 1

K:=1; // job one is inserted into J //

```
for i :=2 to n do // consider jobs in non increasing order of pi //
r:=k;
While ((d[J[r]]>d[i]) and (d[J[r]]#r)) do r:=r-1;
If ((d[J[r] ≤ d[i]) and d[i]>r)) then { //insert i into J[]
For q:=k to (r+1) step-1 do j[q+1]:=j[q];
J[r+1]:=i; k:=k+1;
} } return k;
}
```

3.4 Minimum Cost Spanning Trees

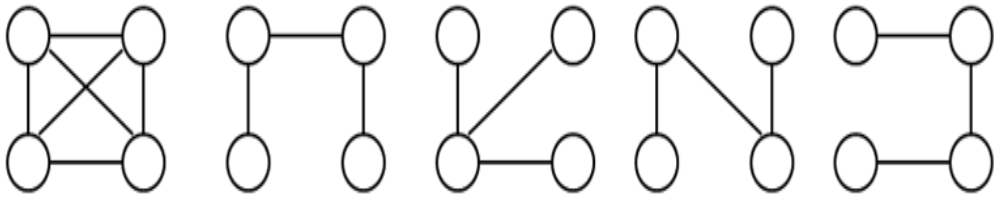
Spanning trees

Suppose you have a connected undirected graph

-Connected: every node is reachable from every other node

-Undirected: edges do not have an associated direction

Then a spanning tree of the graph is a connected subgraph in which there are no cycles

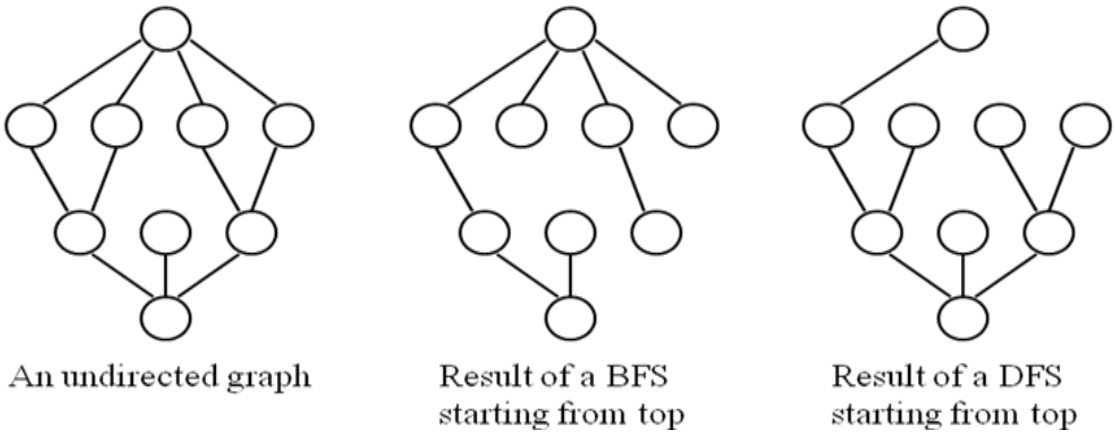


A connected, undirected graph

Four of the spanning trees of the graph

Finding a spanning tree:

To find a spanning tree of a graph, pick an initial node and call it part of the spanning tree do a search from the initial node: each time you find a node that is not in the spanning tree, add to the spanning tree both the new node *and* the edge you followed to get to it .



An undirected graph | Result of a BFS starting from top | Result of a DFS starting from top

Minimizing costs

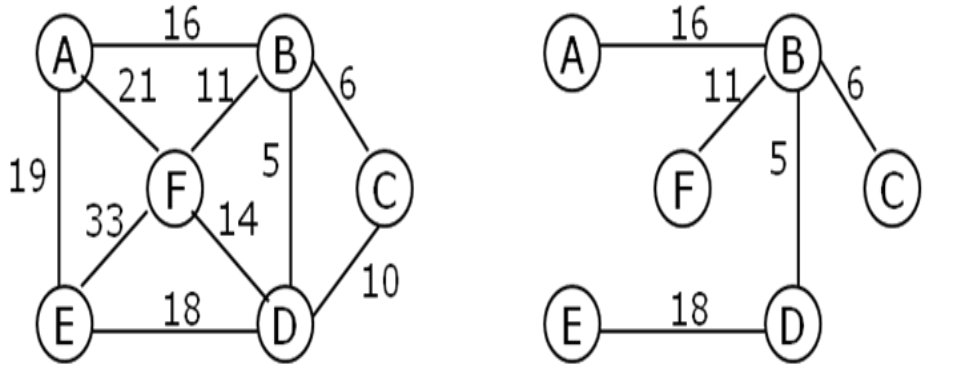
Suppose you want to supply a set of houses (say, in a new subdivision) with:

- electric power
- water
- sewage lines
- telephone lines

To keep costs down, you could connect these houses with a spanning tree (of, for example, power lines) However, the houses are not all equal distances apart. To reduce costs even further, you could connect the houses with a minimum-cost spanning tree.

Minimum-cost spanning trees

Suppose you have a connected undirected graph with a weight (or cost) associated with each edge. The cost of a spanning tree would be the sum of the costs of its edges. A minimum-cost spanning tree is a spanning tree that has the lowest cost.



A connected, undirected graph | A minimum-cost spanning tree

3.5 Greedy Approach for Prim's and kruskal's algorithm:

Both Prim's and Kruskal's algorithms are greedy algorithms. The greedy approach works for the MST problem; however, it does not work for many other problems.

Prim's algorithm:

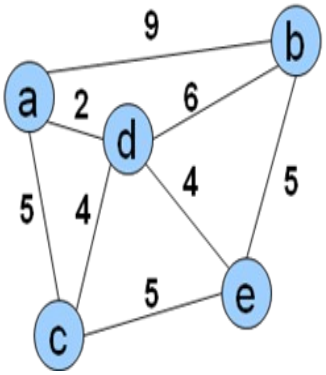
```
T = a spanning tree containing a single node s;
E = set of edges adjacent to s;
while T does not contain all the nodes
{
  remove an edge (v, w) of lowest cost from E
  if w is already in T then discard edge (v, w)
  else
 {
  add edge (v, w) and node w to T
   add to E the edges adjacent to w
  }
}
```

An edge of lowest cost can be found with a priority queue. Testing for a cycle is automatic

Prim's Algorithm:

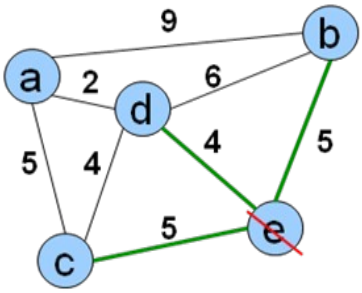
Initialization

a. Pick a vertex *r* to be the root
b. Set $D(r) = 0$, $parent(r) = null$
c. For all vertices $v \in V$, $v \neq r$, set $D(v) = \infty$
d. Insert all vertices into priority queue *P*, using distances as the keys



e	a	b	c	d
0	∞	∞	∞	∞

Vertex	Parent
e	-



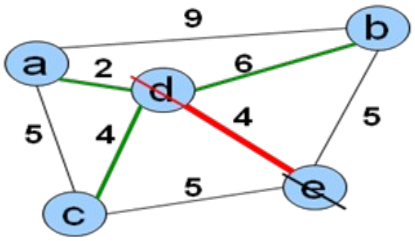
e	d	b	c	a
0	∞	∞	∞	∞

Vertex	Parent
e	-
b	-
c	-
d	-

d	b	c	a
4	**5**	**5**	∞

Vertex	Parent
e	-
b	e
c	e
d	e

The MST initially consists of the vertex *e,* and we update the distances and parent for its adjacent vertices.



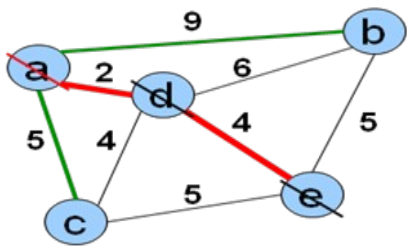
d	b	c	a
4	**5**	**5**	∞

Vertex	Parent
e	-
b	e
c	e
d	e

---

a	c	b
2	**4**	**5**

Vertex	Parent
e	-
b	e
c	d
d	e
a	d



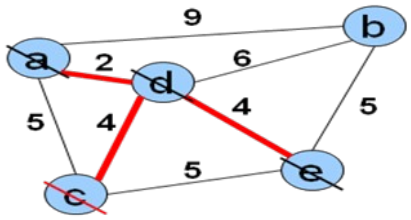
a	c	b
2	**4**	**5**

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

---

c	b
4	**5**

Vertex	Parent
e	-
b	e
c	d
d	e
a	d



c	b
4	5

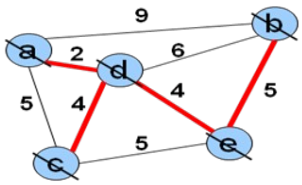
Vertex	Parent
e	-
b	e
c	d
d	e
a	d

---

b
5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

Final Spanning tree



b
5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

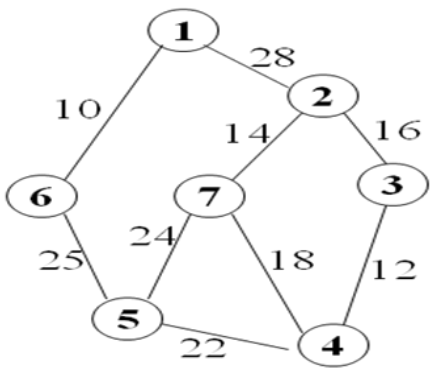
Running time of Prim's algorithm (without heaps):

1. Initialization of priority queue (array): O (|*V*|)
2. Update loop: |*V*| calls
   - Choosing vertex with minimum cost edge: O(|*V*|)
   - Updating distance values of unconnected vertices: each edge is considered only once during entire execution, for a total of O(|*E*|) updates

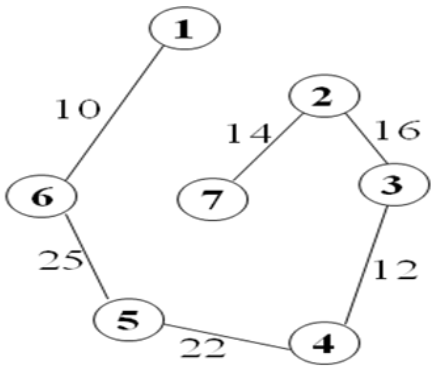
3. Overall cost without heaps:

Minimum-cost Spanning Trees

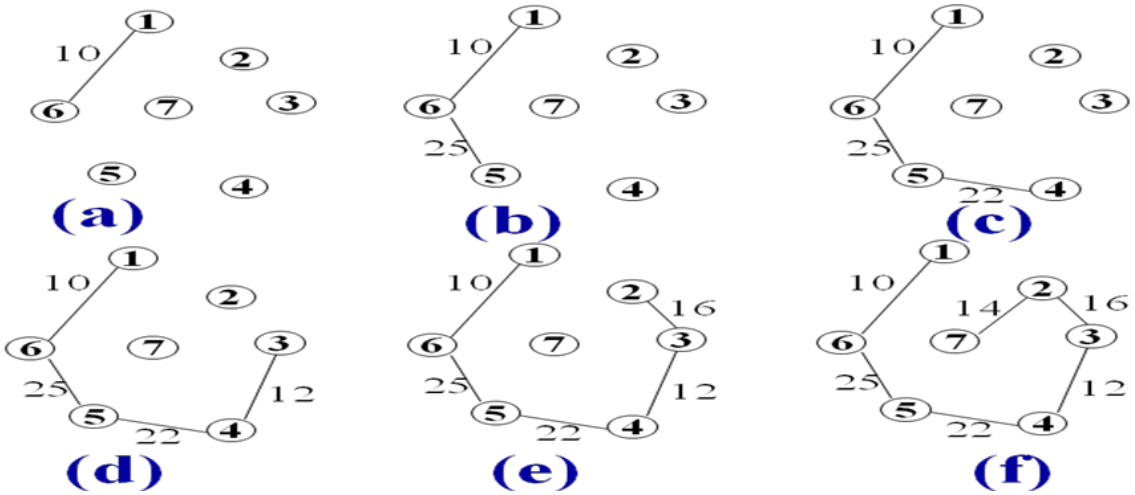
Example of **MCST**: Finding a spanning tree of G with minimum cost



(a)



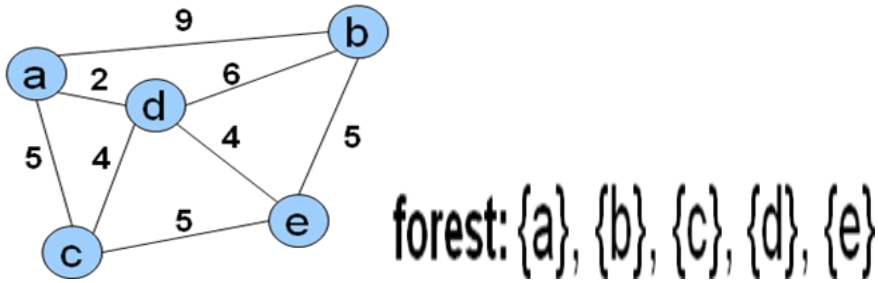
(b)



Prim's Algorithm Invariant:

At each step, we add the edge (u,v) s.t. the weight of (u,v) is minimum among all edges where u is in the tree and v is not in the tree. Each step maintains a minimum spanning tree of the vertices that have been included thus far. When all vertices have been included, we have a MST for the graph.

Another Approach:
Create a forest of trees from the vertices. Repeatedly merge trees by adding "safe edges" until only one tree remains. A "safe edge" is an edge of minimum weight which does not create a cycle.



Kruskal's algorithm:

T = empty spanning tree;

E = set of edges;

N = number of nodes in graph;

while T has fewer than N - 1 edges {

remove an edge (v, w) of lowest cost from E

if adding (v, w) to T would create a cycle

then discard (v, w)

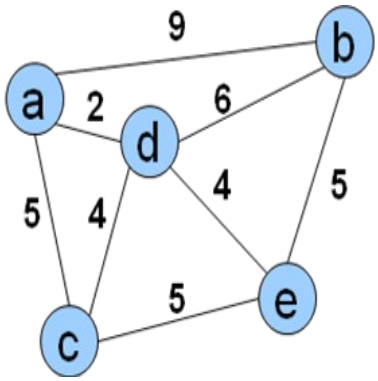
else add (v, w) to T

}

Finding an edge of lowest cost can be done just by sorting the edges
Running time bounded by sorting (or findMin)
O(|*E*|log|*E*|), or equivalently, O(|*E*| log|*V*|)

Initialization

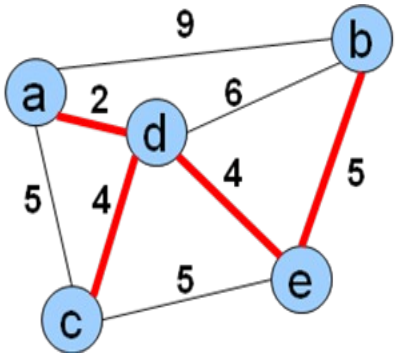
a. Create a set for each vertex $v \in V$
b. Initialize the set of "safe edges" A comprising the MST to the empty set
c. Sort edges by increasing weight



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

$A = \emptyset$

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



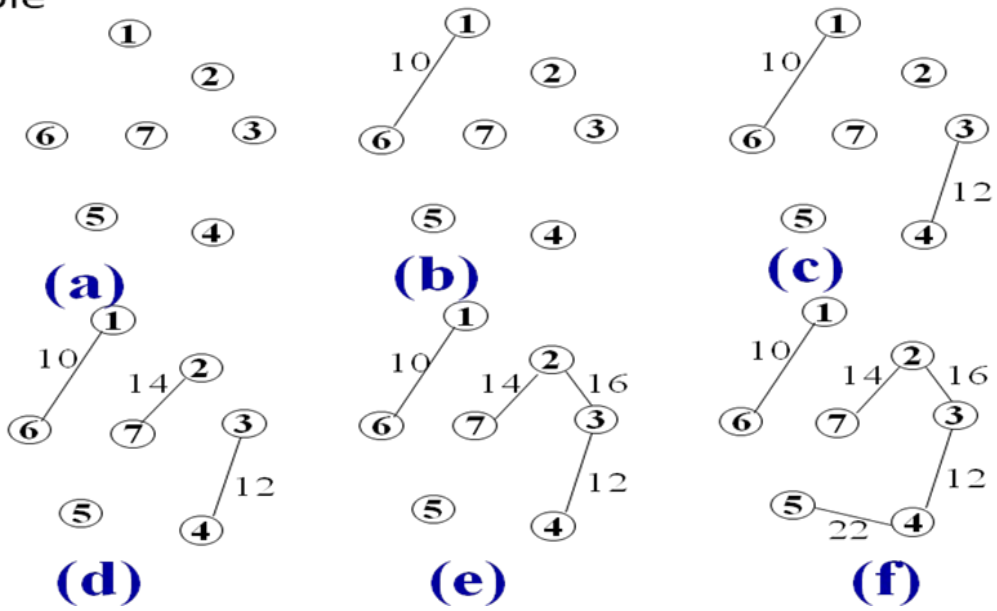
$E = \{(a,d), (c,d), (d,e), (a,c), (b,e), (c,e), (b,d), (a,b)\}$

Forest	A
{a}, {b}, {c}, {d}, {e}	∅
{a,d}, {b}, {c}, {e}	{(a,d)}
{a,d,c}, {b}, {e}	{(a,d), (c,d)}
{a,d,c,e}, {b}	{(a,d), (c,d), (d,e)}
{a,d,c,e,b}	{(a,d), (c,d), (d,e), (b,e)}

Kruskal's algorithm Invariant

After each iteration, every tree in the forest is a MST of the vertices it connects. Algorithm terminates when all vertices are connected into one tree.

- Example



3.6 Optimal Merge Patterns

Problem

Given n sorted files, find an optimal way (i.e., requiring the fewest comparisons or record moves) to pair wise merge them into one sorted file. It fits ordering paradigm.

Example

Three sorted files (x_1, x_2, x_3) with lengths (30, 20, 10)

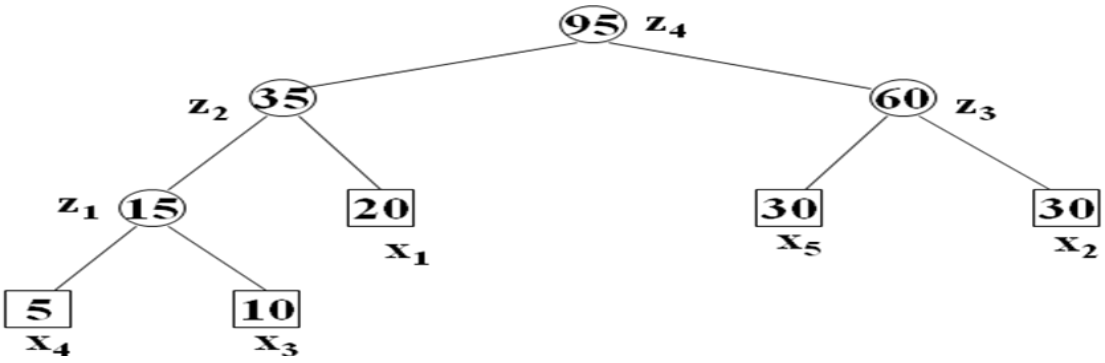
Solution 1: merging x_1 and x_2 (50 record moves), merging the result with x_3 (60 moves) → total 110 moves

Solution 2: merging x_2 and x_3 (30 moves), merging the result with x_1 (60 moves) → total 90 moves

The solution 2 is better.

A greedy method (for 2-way merge problem)

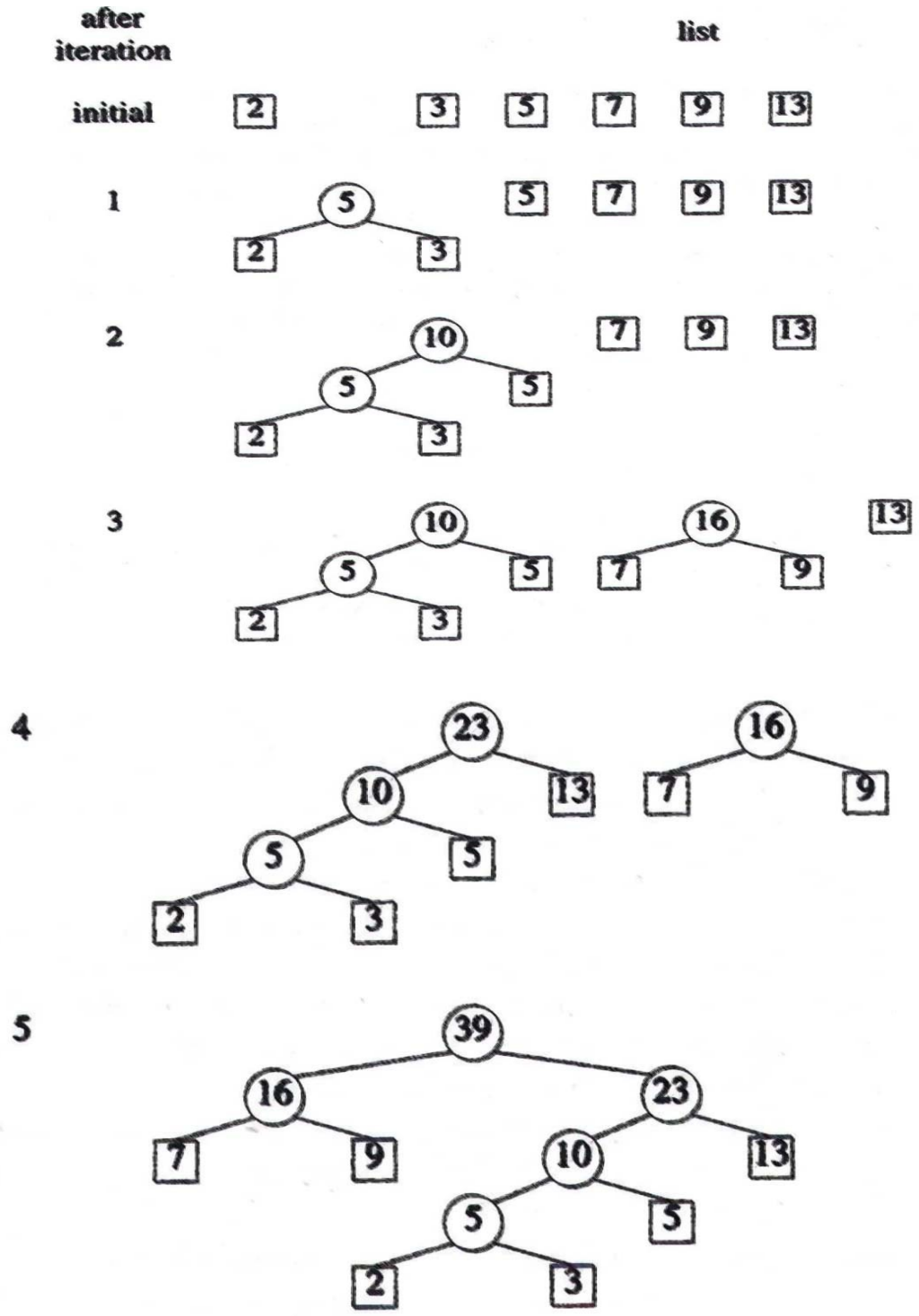
At each step, merge the two smallest files. e.g., five files with lengths (20, 30, 10, 5, 30).



Total number of record moves = weighted external path length
The optimal 2-way merge pattern = binary merge tree with minimum weighted external path length
Algorithm

```
 struct treenode
{
      struct treenode *lchild, *rchild;
      int weight;
};
typedef struct treenode Type;
Type *Tree(int n)
//    list is a global list of n single node
//    binary trees as described above.
{
      for (int i=1; i<n; i++) {
          Type *pt = new Type;
          // Get a new tree node.
          pt -> lchild = Least(list); // Merge two trees with
          pt -> rchild = Least(list); // smallest lengths.
          pt -> weight = (pt->lchild)->weight
                  + (pt->rchild)->weight;
          Insert(list, *pt);
      }
      return (Least(list)); // Tree left in l is the merge tree.
}
```

Example



Time Complexity

If list is kept in non-decreasing order: O (n^2)
If list is represented as a min heap: O (n log n)

3.7 Optimal Storage on Tapes

There are n programs that are to be stored on a computer tape of length L. Associated with each program i is a length L_i. Assume the tape is initially positioned at the front. If the programs are stored in the order $I = i_1, i_2 \ldots i_n$, the time t_j needed to retrieve program i_j

$$t_j = \sum_{k=\backslash}^{j} L_i \frac{1}{n} \sum_{j=l}^{n} t_j$$

If all programs are retrieved equally often, then the mean retrieval time (MRT) =this problem fits the ordering paradigm. Minimizing the MRT is equivalent to minimizing

$$D(I) = \sum_{j=l}^{n} \sum_{k=l}^{j} L_{i_k}$$

Example

n=3 (l1, l2, l3) = (5, 10, 3) 3! =6 total combinations

L1 l2 l3 = l1+ (l1+l2) + (l1+l2+l3) = 5+15+18 = 38/3=12.6
n 3

L1 l3 l2 = l1 + (l1+l3) + (l1+l2+l3) = 5+8+18 = 31/3=10.3
n 3

L2 l1 l3 = l2 + (l2+l1) + (l2+l1+l3) = 10+15+18 = 43/3=14.3
n 3

L2 l3 l1 = 10+13+18 = 41/3=13.6
3

L3 l1 l2 = 3+8+18 = 29/3=9.6 min
3

L3 l2 l1 = 3+13+18 = 34/3=11.3 min
3 permutations at (3, 1, 2)

Example

n = 4, $(p_1, p_2, p_3, p_4) = (100, 10, 1\ 5, 27)$ $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$

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

9 (4) 4 27

Example

Let n = 3, (L_1, L_2, L_3) = (5, 10, 3). 6 possible orderings. The optimal is 3, 1, 2

Ordering I	d(I)
1,2,3	5+5+10+5+10+3 = 38
1,3,2	5+5+3+5+3+10 = 31
2,1,3	10+10+5+10+5+3 = 43
2,3,1	10+10+3+10+3+5 = 41
3,1,2	3+3+5+3+5+10 = 29
3,2,1,	3+3+10+3+10+5 = 34

3.8 TVSP (Tree Vertex Splitting Problem)

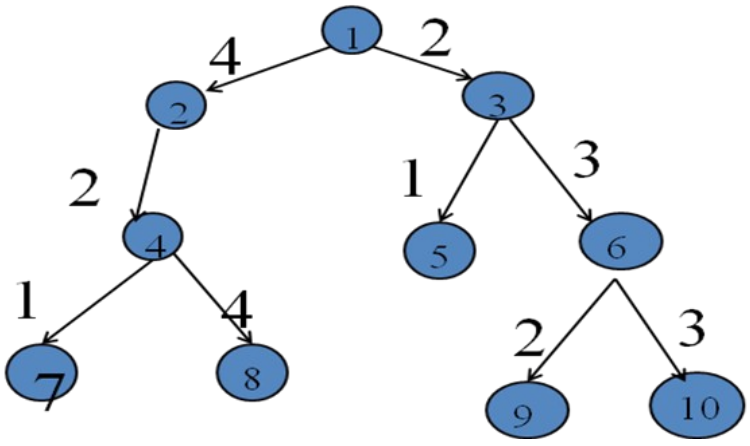
Let T= (V, E, W) be a directed tree. A weighted tree can be used to model a distribution network in which electrical signals are transmitted. Nodes in the tree correspond to receiving stations & edges correspond to transmission lines. In the process of transmission some loss is occurred. Each edge in the tree is labeled with the loss that occurs in traversing that edge. The network model may not able tolerate losses beyond a certain level. In places where the loss exceeds the tolerance value boosters have to be placed. Given a networks and tolerance value, the TVSP problem is to determine an optimal placement of boosters. The boosters can only placed at the nodes of the tree.

d (u) = Max { d(v) + w(Parent(u), u)}

d(u) – delay of node v-set of all edges & v belongs to child(u)

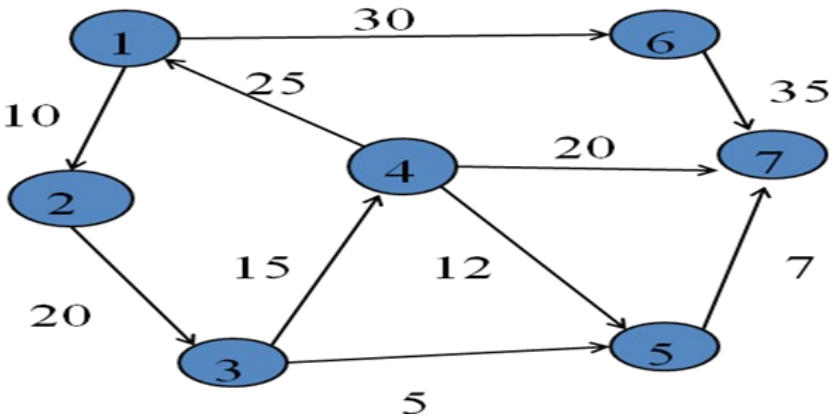
δ tolerance value

TVSP (Tree Vertex Splitting Problem)



If d (u)>= δ than place the booster.
d (7)= max{0+w(4,7)}=1
d (8)=max{0+w(4,8)}=4
d (9)= max{0+ w(6,9)}=2
d (10)= max{0+w(6,10)}=3 d(5)=max{0+e(3.3)}=1
d (4)= max{1+w(2,4), 4+w(2,4)}=max{1+2,4+3}=6> δ ->booster
d (6)=max{2+w(3,6),3+w(3,6)}=max{2+3,3+3}=6> δ->booster
d (2)=max{6+w(1,2)}=max{6+4)=10> δ->booster
d (3)=max{1+w(1,3), 6+w(1,3)}=max{3,8}=8> δ ->booster
Note: No need to find tolerance value for node 1 because from source only power is transmitting.

3.9 Single-source Shortest Paths



Let G=(V,E) be a directed graph and a main function is C(e)(c=cost, e=edge) for the edges of graph 'G' and a source vertex it will represented with V_0 the vertices represents cities and weights represents distance between 2 cities. The objective of the problem find shortest path from source to destination. The length of path is defined to be sum of weights of edges on the path. S[i] =T if vertex i present in set's'. S[i] =F if vertex i is not present in set's'

Formula

Min {distance[w],distance[u]+cost[u, w]}
u-recently visited node w-unvisited node

Step-1 s[1]
s[1]=T dist[2]=10
s[2]=F dist[3]=α
s[3]=F dist[4]= α
s[4]=F dist[5]= α
s[5]=F dist[6]= 30
s[6]=F dist[7]= α
S[7]=F

Step-2 s[1,2] the visited nodes

W={3,4,5,6,7} unvisited nodes

U={2} recently visited node

s[1]=T w=3

s[2]=T dist[3]=α

s[3]=F min {dist[w], dist[u]+cost(u, w)}

s[4]=F min {dist[3], dist[2]+cost(2,3)}

s[5]=F min{α, 10+20}= 30

s[6]=F w=4 dist[4]= α

S[7]=F min{dist(4),dist(2)+cost(2,4)}

min{α,10+ α}= α

W=5 dist[5]= α min{dist(5),dist(2)+cost(2,5)}

min{α,10+ α}= α

W=6 dist[6]=30

Min{dist(6), dist(2)+cost(2,6)}=min{30,10+ α}=30

W=7, dist(7)= α min{dist(7),dist(2)+cost(2,7)}

min{α,10+ α}= α let min. cost is 30 at both 3 and 6 but

Recently visited node 2 have only direct way to 3, so consider 3 is min cost node from 2.

Step-3 w=4,5,6,7

s[1]=T s={1,2,3} w=4 ,dist[4]= α

s[2]=T min{dist[4],dist[3]+cost(3.4)}=min{α,30+15}=45

s[3]=T w=5, dist[5]= α min{dist(5), dist(3)+cost(3,5)}

s[4]=F min{α,30+5}=35 similarity we obtain

s[5]=F w=6, dist(6)=30 w=7 ,dist[7]= α so min cost is 30 at w=6 but

s[6]=F no path from 3 so we consider 5 node so visited nodes 1,2,3, 5

S[7]=F

Step-4 w=4,6,7 s={1,2,3,5}

s[1]=T w=4, dist[4]=45 min {dist[4], dist[5]+cost(5,4)}

s[2]=T min{45,35+ α}=45

s[3]=T w=6,dist[6]=30 min{dist[6],dist[5]+cost(5,6)}

s[4]=F min{30, 35+ α}=30

s[5]=T w=7,dist[7]= α min{dist[7],dist[5]+cost(5,7)}

s[6]=F min{α, 35+7}=42

S[7]=F here min cost is 30 at 6 node but there is no path from 5 yo 6, so we consider 7 , 1,2,3,5,7 nodes visited.

Therefore the graph traveled from source to destination

Single source shortest path is drawn in next slide.

Design of greedy algorithm

Building the shortest paths one by one, in non-decreasing order of path lengths

e.g., 1→4: 10

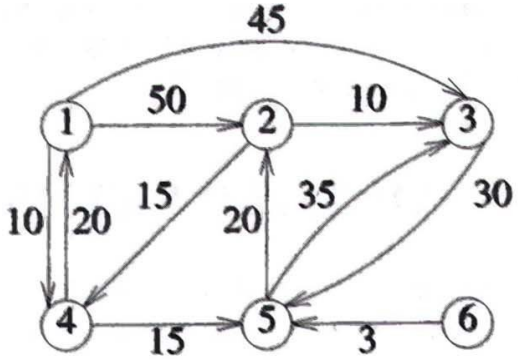
1→4→5: 25

…

We need to determine 1) the next vertex to which a shortest path must be generated and 2) a shortest path to this vertex.

Notations

S = set of vertices (including v_0) to which the shortest paths have already been generated
Dist (w) = length of shortest path starting from v_0, going through only those vertices that are in S, and ending at w.
Three observations
If the next shortest path is to vertex u, then the path begins at v_0, ends at u, and goes through only those vertices that are in S. The destination of the next path generated must be that of vertex u which has the minimum distance, *dist (u),* among all vertices not in S.
Having selected a vertex u as in observation 2 and generated the shortest v_0 to u path, vertex u becomes a member of S.



(a) Graph

Path	Length
1) 1, 4	10
2) 1, 4, 5	25
3) 1, 4, 5, 2	45
4) 1, 3	45

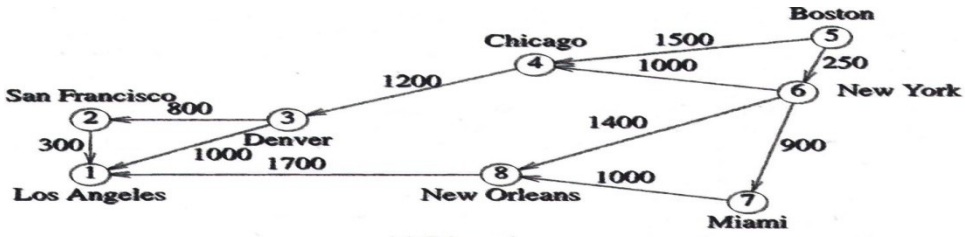
(b) Shortest paths from 1

DIJKSTRA'S Shortest Path Algorithm

Procedure SHORT-PATHS (v, cost, Dist, n)
// Dist (j) is the length of the shortest path from v to j in the //graph G with n vertices; Dist (v)= 0 //
Boolean S (1:n); real cost (1:n,1:n), Dist (1:n); integer u, v, n, num, i, w
// S (i) = 0 if i is not in S and s(i) =1 if it is in S//
// cost (i, j) = +α if edge (i, j) is not there//
// cost (i,j) = 0 if i = j; cost (i, j) = weight of < i, j >
// for i←1 to do // initialize S to empty
// S(i) ←0; Dist (i)← cost(v, i)
Repeat
// initially for no vertex shortest path is available
// S (v)←1; dist(v)←0// Put v in set S //
for num←2 to n-1 do // determine n-1 paths from// //vertex v //
choose u such that Dist (u)=min{dist(w)} and S(w)=0
S (u)←1 // Put vertex u in S //
Dist (w)←min (dist(w),Dist(u) + cost (u, w))
Repeat

repeat
end SHORT - PATHS
Overall run time of algorithm is O ((n+|E|) log n)

Example:



(a) Digraph

	1	2	3	4	5	6	7	8
1	0							
2	300	0						
3	100	800	0					
4			1200	0				
5				1500	0	250		
6				1000		0	900	1400
7							0	1000
8	1700							0

(b) Length-adjacency matrix

Iteration	S	Vertex selected	Distance LA [1]	SF [2]	DEN [3]	CHI [4]	BOST [5]	NY [6]	MIA [7]	NO [8]
Initial	--	----	+∞	+∞	+∞	1500	0	250	+∞	+∞
1	{5}	6	+∞	+∞	+∞	1250	0	250	1150	1650
2	{5,6}	7	+∞	+∞	+∞	1250	0	250	1150	1650
3	{5,6,7}	4	+∞	+∞	2450	1250	0	250	1150	1650
4	{5,6,7,4}	8	3350	+∞	2450	1250	0	250	1150	1650
5	{5,6,7,4,8}	3	3350	3250	2450	1250	0	250	1150	1650
6	{5,6,7,4,8,3}	2	3350	3250	2450	1250	0	250	1150	1650
	{5,6,7,4,8,3,2}									

****

Chapter-4
Dynamic programming

4.1 The General Method

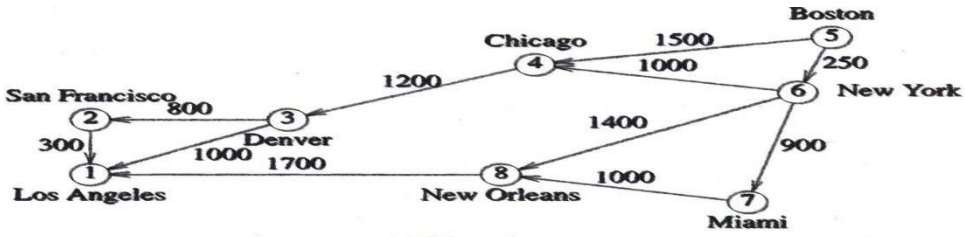
Dynamic Programming: is an algorithm design method that can be used when the solution to a problem may be viewed as the result of a sequence of decisions.

The shortest path

To find a shortest path in a multi-stage graph

repeat
end SHORT - PATHS
Overall run time of algorithm is O ((n+|E|) log n)

Example:



(a) Digraph

	1	2	3	4	5	6	7	8
1	0							
2	300	0						
3	100	800	0					
4			1200	0				
5				1500	0	250		
6				1000		0	900	1400
7							0	1000
8	1700							0

(b) Length-adjacency matrix

Iteration	S	Vertex selected	Distance LA [1]	SF [2]	DEN [3]	CHI [4]	BOST [5]	NY [6]	MIA [7]	NO [8]
Initial	--	----	+∞	+∞	+∞	1500	0	250	+∞	+∞
1	{5}	6	+∞	+∞	+∞	1250	0	250	1150	1650
2	{5,6}	7	+∞	+∞	+∞	1250	0	250	1150	1650
3	{5,6,7}	4	+∞	+∞	2450	1250	0	250	1150	1650
4	{5,6,7,4}	8	3350	+∞	2450	1250	0	250	1150	1650
5	{5,6,7,4,8}	3	3350	3250	2450	1250	0	250	1150	1650
6	{5,6,7,4,8,3}	2	3350	3250	2450	1250	0	250	1150	1650
	{5,6,7,4,8,3,2}									

****

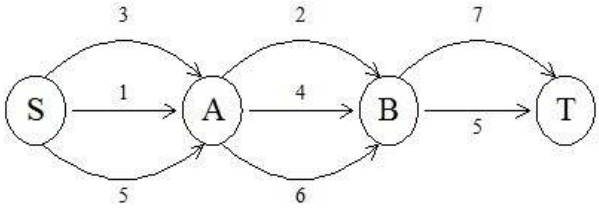
Chapter-4
Dynamic programming

4.1 The General Method

Dynamic Programming: is an algorithm design method that can be used when the solution to a problem may be viewed as the result of a sequence of decisions.

The shortest path

To find a shortest path in a multi-stage graph



Apply the greedy method

the shortest path from S to T
1 + 2 + 5 = 8

4.2 Principle of optimality

Suppose that in solving a problem, we have to make a sequence of decisions D1, D2… Dn. If this sequence is optimal, then the last k decisions, 1 k n must be optimal.

Ex: The shortest path problem

If i1, i2… j is a shortest path from i to j, then i1, i2… j must be a shortest path from i1to j
If a problem can be described by a multistage graph, then it can be solved by dynamic programming.

4.2.1 Forward approach and backward approach

Note that if the recurrence relations are formulated using the forward approach then the relations are solved backwards. i.e., beginning with the last decision

On the other hand if the relations are formulated using the backward approach, they are solved forwards.

To solve a problem by using dynamic programming

- Find out the recurrence relations.
- Represent the problem by a multistage graph.

Backward chaining vs. forward chaining

Recursion is sometimes called "backward chaining": start with the goal you want choosing your sub goals on an as-needed basis.

- Reason backwards from goal to facts (start with goal and look for support for it)
  Another option is "forward chaining": compute each value as soon as you can, in hope that you'll reach the goal.
- Reason forward from facts to goal (start with what you know and look for things you can prove)

Using forward approach to find cost of the path:

Cost (i, j) = min {c(j, *l*) + cost(i+1, *l*)}

L ϵ **Vi+1**
<j, *l*> ϵ **E**

```
Algorithm FGraph(G, k, n, p)
// The input is a k-stage graph G = (V, E) with n vertices
// indexed in order of stages. E is a set of edges and c[i, j]
// is the cost of ⟨i, j⟩. p[1 : k] is a minimum-cost path.
{
    cost[n] := 0.0;
    for j := n − 1 to 1 step −1 do
    { // Compute cost[j].
        Let r be a vertex such that ⟨j, r⟩ is an edge
        of G and c[j, r] + cost[r] is minimum;
        cost[j] := c[j, r] + cost[r];
        d[j] := r;
    }
    // Find a minimum-cost path.
    p[1] := 1; p[k] := n;
    for j := 2 to k − 1 do p[j] := d[p[j − 1]];
}
```

Algorithm 4.1 Multistage graph pseudo code corresponding to the forward approach Using backward approach:

Let bp (i, j) be a minimum cost path from vertex s to vertex j in Vi Let bcost (i,j) be cost of bp(i, j). The backward apporach to find minimum cost is:

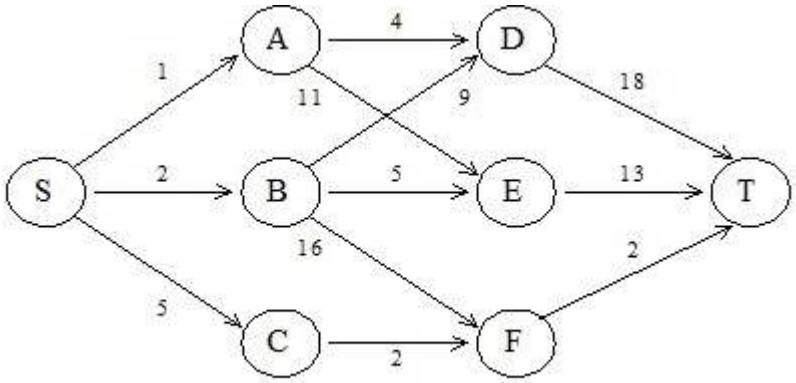
bcost (i, j) = min {bcost (i-1,*l*) +c(*l* ,j)}
l ϵ **Vi+1**
<j,*l*> ϵ **E**

Since bcost (2,j) = c(1,j) if <1,j> ϵ E and bcost (2,j) = ∞
if <i,j> ϵ E, bcost (i,j) can be computed using above formula.

```
Algorithm Bgraph(G,k,n,p)
{
bcost[1]:=0.0;
For j:=2 to n do
{ //compute bcost[j].
Let r be such that <r,j> is an edge of G and bcost[r] + c[r,j] is mimimum;
bcost[j]:=bcost[r]+c[r,j];
d[j]:=r;
}
//Find a minimum-cost path
P[1]:=1;p[k]:=n;
For j:=k-1 to 2 do p[j]:= d[p[j+1]];
}
```

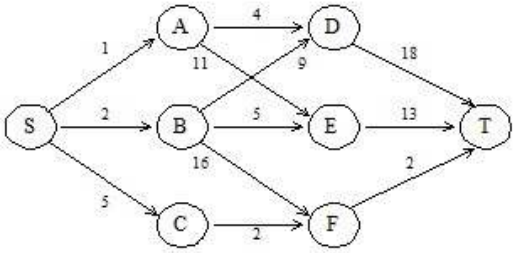
Algorithm: 4.1.1 Multi-stage graph pseudo code for corresponding backward approach.

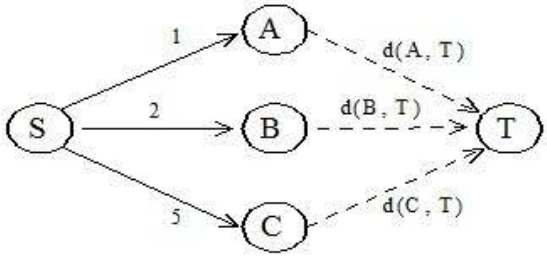
The shortest path in multistage graphs:



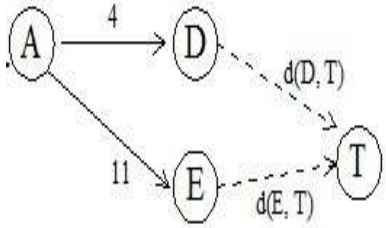
- The greedy method cannot be applied to this case: (S, A, D, T) 1+4+18 = 23.
- The real shortest path is:
  (S, C, F, T) 5+2+2 = 9.

Dynamic programming approach (forward approach)



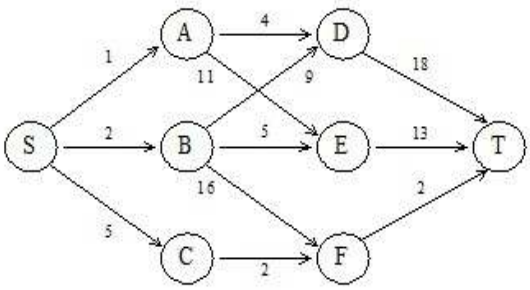


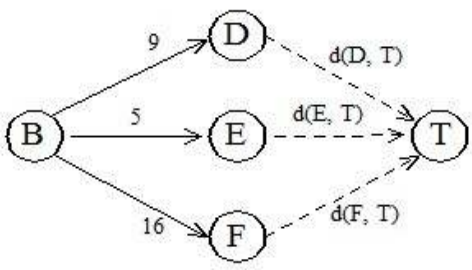
d(S, T) = min{1+d(A, T), 2+d(B, T), 5+d(C, T)}



d(A,T) = min{4+d(D,T), 11+d(E,T)}
= min{4+18, 11+13} = 22.

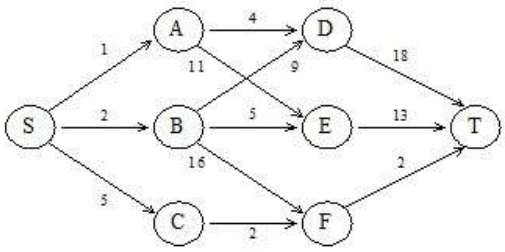
d(B, T) = min{9+d(D, T), 5+d(E, T), 16+d(F, T)}
= min{9+18, 5+13, 16+2} = 18.





d (C, T) = min{ 2+d(F, T) } = 2+2 = 4
d (S, T) = min{1+d(A, T), 2+d(B, T), 5+d(C, T)}
= min{1+22, 2+18, 5+4} = 9.
The above way of reasoning is called backward reasoning.
Backward approach (forward reasoning):
d(S, A) = 1
d(S, B) = 2
d(S, C) = 5
d(S,D) = min{d(S,A)+d(A,D),d(S,B)+d(B,D)}
= min{ 1+4, 2+9 } = 5
d(S,E) = min{d(S,A)+d(A,E), d(S,B)+d(B,E)}
= min{ 1+11, 2+5 } = 7
d(S,F) = min{d(S,B)+d(B,F), d(S,C)+d(C,F)}
= min{ 2+16, 5+2 } = 7
d(S,T) = min{d(S, D)+d(D, T), d(S,E)+d(E,T), d(S, F)+d(F, T)}
= min{ 5+18, 7+13, 7+2 }
= 9

4.3 Multistage Graphs



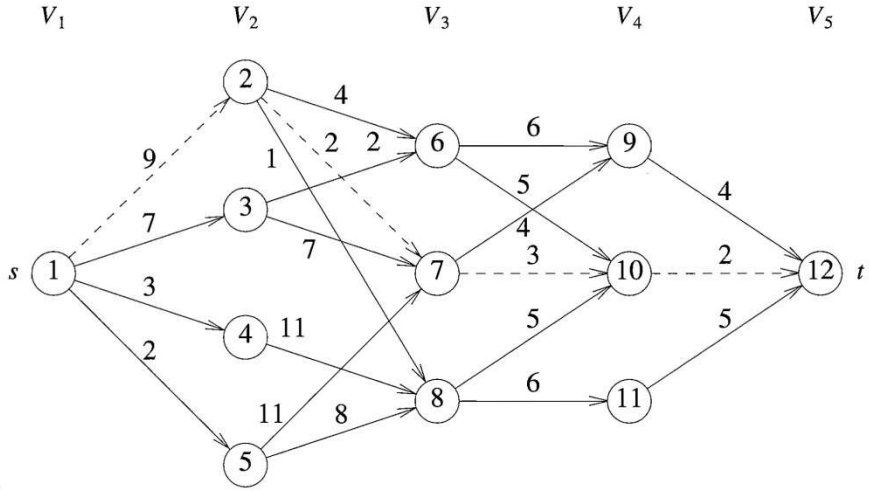
Multistage graph problem is to determine shortest path from source to destination. A multistage graph G=(V,E) is a directed graph in which the vertices are partitioned into k>=2 disjoint sets Vi, 1<=i<=k.

The vertex s is source and t is the sink. Let c(i,j) be the cost of edge <i, j>. The cost of a path from s to t is the sum of costs of the edges on the path. The multistage graph problem is to find a minimum-cost path from s to t.

A dynamic programming formulation for a k-stage graph problem is obtained by first noticing that every s to t path is the result of a sequence of k-2 decisions.

The ith decision involve determining which vertex in Vi+1, 1<=i<=k-2, is on the path. It is easy to see that principal of optimality holds.
Let p(i,j) be a minimum-cost path from vertex j in Vi to vertex t. Let cost(i,j) be the cost of this path.

$$cost(i,j) = \min_{\substack{l \in V_{i+1} \\ \langle j,l \rangle \in E}} \{c(j,l) + cost(i+1,l)\}$$



$$
\begin{aligned}
cost(3,6) &= \min\ \{6 + cost(4,9), 5 + cost(4,10)\} \\
&= 7 \\
cost(3,7) &= \min\ \{4 + cost(4,9), 3 + cost(4,10)\} \\
&= 5 \\
cost(3,8) &= 7 \\
cost(2,2) &= \min\ \{4 + cost(3,6), 2 + cost(3,7), 1 + cost(3,8)\} \\
&= 7 \\
cost(2,3) &= 9 \\
cost(2,4) &= 18 \\
cost(2,5) &= 15 \\
cost(1,1) &= \min\ \{9 + cost(2,2), 7 + cost(2,3), 3 + cost(2,4), \\
&\qquad 2 + cost(2,5)\} \\
&= 16
\end{aligned}
$$

The time for the **for** loop of line 7 is $\Theta(|V| + |E|)$, and the time for the **for** loop of line 16 is $\Theta(k)$, Hence, the total time is $\Theta(|V| + |E|)$.
The backward trace from vertex 1 to *n* also works.
The algorithm also works for the edges crossing more than 1 stage.

4.4 All-pairs Shortest Paths

Let G=(V,E) be a directed graph with n vertices. The cost I,j=0 if i=j , cost I,j is ∞ if i≠j, <i,j> not blelongs E

The cost i,j>0 iif i≠j <i,j> not belongs E

All pairs shortest path problem is to determine the matrix 'A' such that A(i,j) is the length of the shortest path from itoj. The matrix 'A' can be obtained by solving 'n' single source problems by using shortest path algorithm.

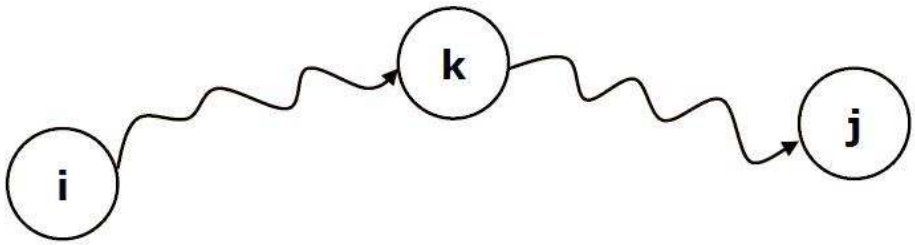
Idea

Label the vertices with integers 1..n

Restrict the shortest paths from i to j to consist of vertices 1..k only (except i and j)

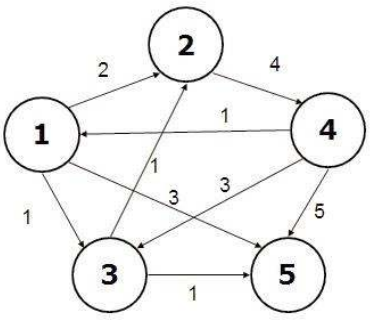
Iteratively relax k from 1 to n.

Find shortest distance from i to j using vertices 1...k

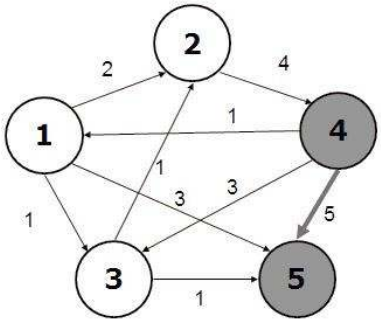


Example

i=4, j=5,k=0

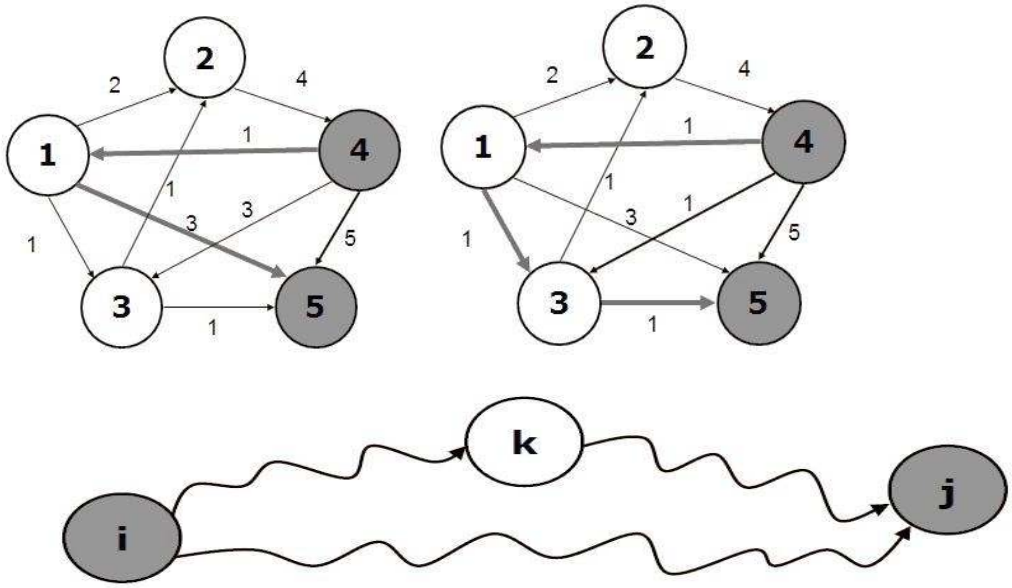


i=4, j=5,k=1



i=4, j=5,k=2

i=4, j=5,k=3



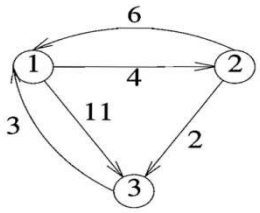
$D^k_{i,j}$: Shortest distance from i to j involving {1..k} only

$$D^k_{i,j} = \begin{cases} \min(D^{k-1}_{i,j}, D^{k-1}_{i,k} + D^{k-1}_{k,j}) & \text{for } k > 0 \\ w_{i,j} & \text{for } k = 0 \end{cases}$$

4.4.1 Shortest Path: Optimal substructure

Let G be a graph, W_{ij} be the length of edge (i, j), where $1<=i, j<= n$, and $d^{(k)}ij$ be the length of the shortest path between nodes I and j, for $1<= i, j, k<=n$, without passing through any nodes numbered greater than k.

Recurrence:



(a) Example digraph

A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

(b) A^0

A^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

(c) A^1

A^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

(d) A^2

A^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

(e) A^3

At d=1

$d^1_{(1,1)} = \min\{d^0(1,1), d^0(1,1,)+ d^0(1,1,)\} = 0$

$d^1_{(1,2)} = \min\{d^0(1,2), d^0(1,1)+d^0(1,2)\} = \min\{4, 0+4\}=4$
$d^1_{(1,3)} = \min\{d^0(1,3), d^0(1,1)+d^0(1,3)\} = \min\{11,0+11\} = 11$
$d^1_{(2,1)} = \min\{d^0(2,1), d^0(2,1)+ d^0(1,1)\}$
$d^1_{(2,2)} = \min\{d^0(2,2), d^0(2,1)+d^0(1,2)\} = 0$
$d^1_{(2,3)} = \min\{d^0(2,3), d^0(2,1)+d^0(1,3)\}= \min\{2,6+11\}=2$
$d^1_{(3,1)} = \min\{d^0(3,1), d^0(3,1) + d^0(1,1)\} = \min\{3, 3+0\} = 3$
$d^1_{(3,2)} = \min\{d^0(3,2), d^0(3,1)+d^0(1,2)\}= \min\{\infty, 3+4\}=7$
$d^1_{(3,3)} = 0$

At d=2

$d^2_{(1,1)} = \min\{d^1(1,1), d^1(1,2)+ d^1(2,1,)\} = \min\{0, \infty\} = 0$
$d^2_{(1,2)} = \min\{d^1(1,2), d^1(1,2)+d^1(2,2)\} =4$
$d^2_{(1,3)} = \min\{d^1(1,3), d^1(1,2)+d^1(2,3)\} = 6$
$d^2_{(2,1)} = \min\{d^1(2,1), d^1(2,2)+ d^1(2,1)\} =6$
$d^2_{(2,2)} = \min\{d^1(2,2), d^1(2,1)+d^1(2,2)\} = \min\{0, 0+0\} = 0$
$d^2_{(2,3)} = \min\{d^1(2,3), d^1(2,1)+d^1(2,3)\}= 2$
$d^2_{(3,1)} = \min\{d^1(3,1), d^1(3,2) + d^1(2,1)\} = \min\{3, 7+6\} = 3$
$d^2_{(3,2)} = \min\{7, 7+0\}=7$
$d^2_{(3,3)} = 0$

At d=3

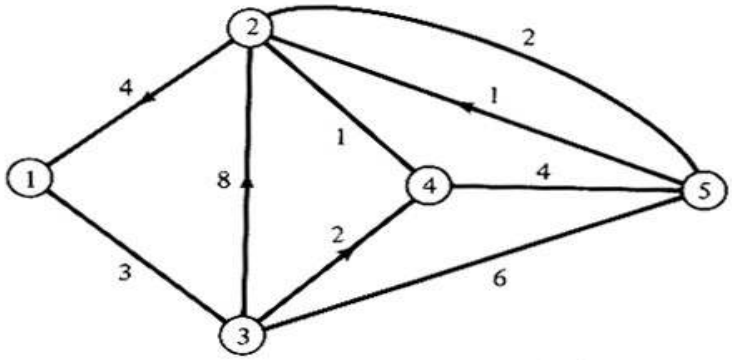
$d^3_{(1,1)} = \min\{0, somevalue\} = 0$
$d^3_{(1,2)} = \min\{4,6+0\}=4$
$d^3_{(1,3)} = \min\{6,6+0)\} = 6$
$d^3_{(2,1)} = \min\{6, 2+3\}=5$
$d^3_{(2,2)} = 0$
$d^3_{(2,3)} = \min\{2,2\}=2$
$d^3_{(3,1)} = \min\{3, 3+0\} = 3$
$d^3_{(3,2)} = \min\{7, 7+0\}=7$
$d^3_{(3,3)} = 0$

```
0   Algorithm AllPaths(cost, A, n)
1   // cost[1 : n, 1 : n] is the cost adjacency matrix of a graph with
2   // n vertices; A[i, j] is the cost of a shortest path from vertex
3   // i to vertex j. cost[i, i] = 0.0, for 1 ≤ i ≤ n.
4   {
5       for i := 1 to n do
6           for j := 1 to n do
7               A[i, j] := cost[i, j]; // Copy cost into A.
8       for k := 1 to n do
9           for i := 1 to n do
10              for j := 1 to n do
11                  A[i, j] := min(A[i, j], A[i, k] + A[k, j]);
12  }
```

Algorithm 4.2 All-Pairs Shortest Paths algorithm

- Find the distance between every pair of vertices in a weighted directed graph G.
- We can make n calls to Dijkstra's algorithm (if no negative edges), which takes O(nmlog n) time.
- Likewise, n calls to Bellman-Ford would take O (n2m) time.
- We can achieve O (n3) time using dynamic programming (similar to the Floyd-Warshall algorithm).

Example for all pairs shortest path:



Initialization:

$D^{(0)}$ =

	1	2	3	4	5
1	0	∞	3	∞	∞
2	4	0	∞	1	2
3	3	8	0	2	6
4	∞	1	∞	0	4
5	∞	1	6	4	0

$P^{(0)}$ =

	1	2	3	4	5
1	—	1	1	1	1
2	2	—	2	2	2
3	3	3	—	3	3
4	4	4	4	—	4
5	5	5	5	5	—

Through Node 1:

$D^{(1)}$ =

	1*	2	3	4	5
*1	0	∞	3	∞	∞
2	4	0	7^+	1	2
3	3	8	0	2	6
4	∞	1	∞	0	4
5	∞	1	6	4	0

$P^{(1)}$ =

	1*	2	3	4	5
*1	—	1	1	1	1
2	2	—	1^+	2	2
3	3	3	—	3	3
4	4	4	4	—	4
5	5	5	5	5	—

Through Node 2:

$D^{(2)}$ =

	1	2*	3	4	5
1	0	∞	3	∞	∞
*2	4	0	7	1	2
3	3	8	0	2	6
4	5+	1	8+	0	3+
5	5+	1	6	2+	0

$P^{(2)}$ =

	1	2*	3	4	5
1	—	1	1	1	1
*2	2	—	1	2	2
3	3	3	—	3	3
4	2+	4	1+	—	2+
5	2+	5	5	2+	—

Through Node 3:

$D^{(3)}$ =

	1	2	3*	4	5
1	1	11+	3	5+	9+
2	4	0	7	1	2
*3	3	8	0	2	6
4	5	1	8	0	3
5	5	1	6	2	0

$P^{(3)}$ =

	1	2	3*	4	5
1	—	3+	1	3+	3+
2	2	—	1	2	2
*3	3	3	—	3	3
4	2	4	1	—	2
5	2	5	5	2	—

Through Node 4:

$D^{(4)}$ =

	1	2	3	4*	5
1	0	6+	3	5	8+
2	4	0	7	1	2
3	3	3+	0	2	5+
*4	5	1	8	0	3
5	5	1	6	2	0

$P^{(4)}$ =

	1	2	3	4*	5
1	—	4+	1	3	2+
2	2	—	1	2	2
3	3	4+	—	3	2+
*4	2	4	1	—	2
5	2	5	5	3	—

Through Node 5:

$D^{(5)}$ =

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

$P^{(5)}$ =

	1	2	3	4	5*
1	—	4	1	3	2
2	2	—	1	2	2
3	3	4	—	3	2
4	2	4	1	—	2
*5	2	5	5	2	—

Note that on the last pass no improvements could be found for $D^{(5)}$ over $D^{(4)}$. The final matrices $D^{(5)}$ and $P^{(5)}$ indicate, for instance, that the shortest path from node 1 to node 5 has length d(1,5) = 8 units and that this shortest path is the path {1, 3, 4, 2, 5}.

To identify that shortest path, we examined row 1 of the $P^{(5)}$ matrix. Entry p_5 says that the predecessor node to 5 in the path from 1 to 5 is node 2; then, entry p_5 (1, 2) says that the predecessor node to 2 in the path from 1 to 2 is node 4; similarly, we backtrack the rest of the path by examining p_5(1, 4) (= 3) and p_5(1, 3) = 1. In general, backtracking stops when the predecessor node is the same as the initial node of the required path.

For another illustration, the shortest path from node 4 to node 3 is d (4, 3) = 8 units long and the path is {4, 2, 1, 3}. The predecessor entries that must be read are, in order, p_5 (4, 3) = 1, p_5 (4, 1) = 2, and finally p_5(4, 2) = 4--at which point we have "returned" to the initial node.

4.5 Single-Source Shortest Paths

4.5.1 General Weights

Let $dist^k[u]$ be the length of a shortest path from the source vertex v to vertex u containing at most k edges.

$$dist^k[u] = \min \{dist^{k-1}[u], \min_i \{dist^{k-1}[i] + cost[i,u]\}\}$$

```
1   Algorithm BellmanFord(v, cost, dist, n)
2   // Single-source/all-destinations shortest
3   // paths with negative edge costs
4   {
5       for i := 1 to n do // Initialize dist.
6           dist[i] := cost[v, i];
7       for k := 2 to n - 1 do
8           for each u such that u ≠ v and u has
9                   at least one incoming edge do
10              for each ⟨i, u⟩ in the graph do
11                  if dist[u] > dist[i] + cost[i, u] then
12                      dist[u] := dist[i] + cost[i, u];
13  }
```

Algorithm 4.3 Bellman and ford algorithm to compute shortest paths

Bellman and ford algorithm: Works even with negative-weight edges

It must assume directed edges (for otherwise we would have negative-weight cycles)

Iteration i finds all shortest paths that use i edges.

Running time: O(nm).
It Can be extended to detect a negative-weight cycle if it exists.

4.6 Optimal Binary Search Trees

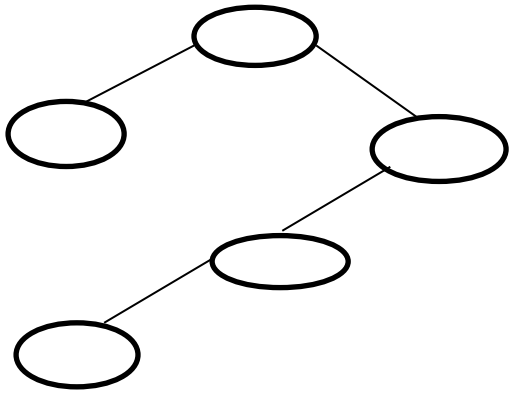
Definition: Binary search tree (BST) A binary search tree is a binary tree; either it is empty or each node contains an identifier and

1. All identifiers in the left sub tree of T are less than the identifiers in the root node T.
2. All the identifiers the right sub tree is greater than the identifier in the root node T.
3. The right and left sub tree are also BSTs.

Algorithm for searching an identifier in the tree 'T'

```
Procedure SEARCH (T X I)
// Search T for X, each node had fields LCHILD, IDENT, RCHILD//
// Return address I pointing to the identifier X// //Initially T is pointing to tree.
//ident(i)=X or i=0
//I ← T
While I ≠ 0 do
  case : X < Ident(i) : I ←LCHILD(i)
       : X = IDENT(i) : RETURN i
         : X > IDENT(i) : I ← RCHILD(i)
  end case
repeat
end SEARCH
```

Optimal Binary Search trees – Example



If each identifier is searched with equal probability the average number of comparisons for the above tree is 1+2+2+3+4/5 = 12/5.

- Let us assume that the given set of identifiers are $\{a_1, a_2...a_n\}$ with $a_1<a_2<.....<a_n$.
- Let P_i be the probability with which we are searching for a_i.
- Let Q_i be the probability that identifier x being searched for is such that $a_i<x<a_{i+1}$ $0\leq i\leq n$, and $a_0=-\infty$ and $a_{n+1}=+\infty$.
- Then $\sum Q_i$ is the probability of an unsuccessful search.

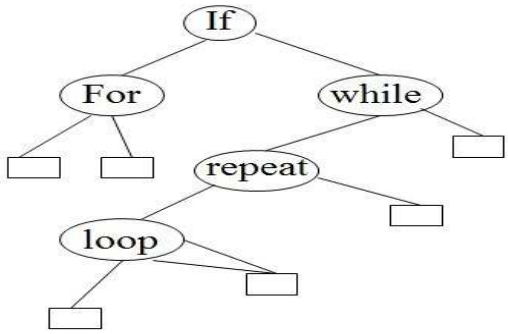
$0\leq i\leq n$

$\sum P(i) + \sum Q(i) = 1$. Given the data,

$1\leq i\leq n \quad 0\leq i\leq n$

let us construct one optimal binary search tree for $(a_1.....a_n)$.

- In place of empty sub tree, we add external nodes denoted with squares.
- Internet nodes are denoted as circles.



4.7 Construction of optimal binary search trees

i) A BST with n identifiers will have n internal nodes and n+ 1 external node.
ii)Successful search terminates at internal nodes unsuccessful search terminates at external nodes.
iii)If a successful search terminates at an internal node at level L, then L iterations of the loop in the algorithm are needed.
iv)Hence the expected cost contribution from the internal nodes for a_i is P (i) * level (a_i).
v)Unsuccessful search terminates at external nodes i.e. at i = 0.
vi)The identifiers not in the binary search tree may be partitioned into n+1 equivalent classes

$E_i \quad 0\leq i\leq n$.

E_o contains all X such that $X\leq a_i$

E_i contains all X such that $a<X<=a_{i+1}$ $1\le i\le n$

E_n contains all X such that $X > a_n$

For identifiers in the same class E_i, the search terminates at the same external node. If the failure node for E_i is at level L, then only L-1 iterations of the while loop are made

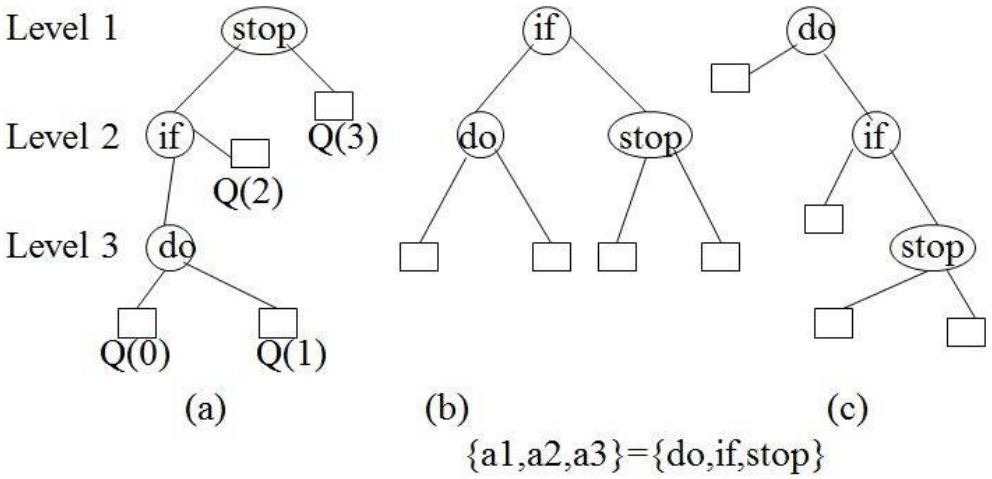
$\therefore$ The cost contribution of the failure node for E_i is Q (i) * level (E_i) -1)

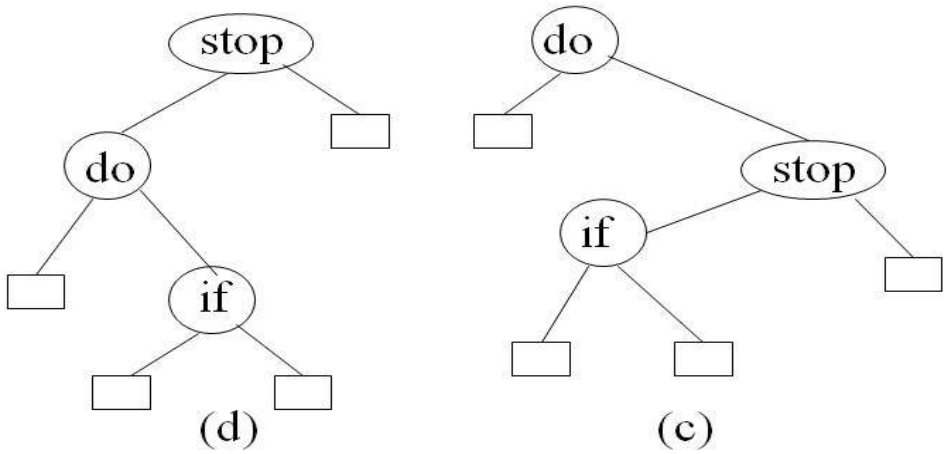
Thus the expected cost of a binary search tree is:

$$\sum_{1\le i\le n} P(i) * \text{level}(a_i) + \sum_{0\le i\le n} Q(i) * \text{level}(E_i) - 1) \quad \ldots\ldots(2)$$

An optimal binary search tree for $\{a_1\ldots\ldots, a_n\}$ is a BST for which (2) is minimum.

Example: Let $\{a_1, a_2, a_3\}$ ={do, if, stop}





With equal probability P (i) = Q(i) = 1/7.

Let us find an OBST out of these.

Cost(tree a) = $\sum_{1\le i\le n} P(i)*\text{level a}(i) + \sum_{0\le i\le n} Q(i)*\text{level (Ei) -1}$

(2-1) (3-1) (4-1) (4-1)

=1/7[1+2+3 + 1 + 2 + 3 + 3] = 15/7

Cost (tree b) =17[1+2+2+2+2+2+2] =13/7

Cost (tree c) =cost (tree d) =cost (tree e) =15/7

∴ tree b is optimal.

If P(1) =0.5 ,P(2) =0.1, P(3) =0.005 , Q(0) =.15 , Q(1) =.1, Q(2) =.05 and Q(3) =.05 find the OBST.

Cost (tree a) = .5 x 3 +.1 x 2 +.05 x 3 +.15x3 +.1x3 +.05x2 +.05x1 = 2.65

Cost (tree b) =1.9 , Cost (tree c) =1.5 ,Cost (tree d) =2.05 , Cost (tree e) =1.6.

Hence tree C is optimal.

To obtain a OBST using Dynamic programming we need to take a sequence of decisions regard. The construction of tree.

First decision is which of a_i is being as root.

Let us choose a_k as the root. Then the internal nodes for $a_1,\ldots\ldots,a_{k-1}$ and the external nodes for classes $E_o,E_1,\ldots\ldots,E_{k-1}$ will lie in the left sub tree L of the root. The remaining nodes will be in the right sub tree R.

Define

$$\text{Cost (L)} = \sum_{1\le i\le k} P(i) * \text{level}(a_i) + \sum_{0\le i\le k} Q(i)*(\text{level}(E_i)-1)$$

$$\text{Cost(R)} = \sum_{k\le i\le n} P(i)*\text{level}(a_i) + \sum_{k\le i\le n} Q(i)*(\text{level}(E_i)-1)$$

Tij be the tree with nodes $a_{i+1},\ldots..,a_j$ and nodes corresponding to $E_i,E_{i+1},\ldots..,E_j$.

Let W (i, j) represents the weight of tree T_{ij}.

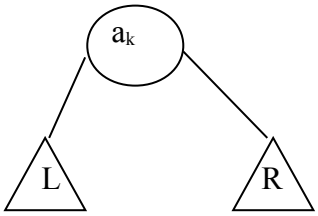
$$W(i, j)=P(i+1) +\ldots+P(j)+Q(i)+Q(i+1)\ldots Q(j)=Q(i) +\sum_{l=i+1}^{j} [Q(l)+P(l)]$$

The expected cost of the search tree in (a) is (let us call it T) is

$$P(k) + \text{cost}(l) + \text{cost}(r) + W(0,k-1) + W(k, n)$$

W (0, k-1) is the sum of probabilities corresponding to nodes and nodes belonging to equivalent classes to the left of a_k.

W (k, n) is the sum of the probabilities corresponding to those on the right of a_k.



(a) OBST with root a_k

4.8 0-1 Knapsack

If we are given 'n' objects and a knapsack or a bag, in which the object ' i' has weight 'w_i' is to be placed, the knapsack has capacity 'N' then the profit that can be earned is $p_i x_i$. The objective is to obtain filling of knapsack with maximum profits is to

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

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

$$\text{and } 0 \le x_i \le 1, \quad 1 \le i \le n$$

n=no of objects i=1,2,….n.; m=capacity of the bag ;
w_i =weight of object I; P_i =profit of the object i.

In solving 0/1 knapsack problem two rules are defined to get the solution.
Rule1: When the weight of object(s) exceeds bag capacity than discard that pair(s).
Rule2: When (p_i,w_i) and (p_j,w_j) where pi $\le p_j$and $w_i \ge$wjthan (p_i,w_i) pair will be discarded. This rule is called purging or dominance rule.Applying dynamic programming method to calculate 0/1 knapsack problem the formula equation is: $S_1^i = \{(P,W)|(P-p_i , W-w_i) \epsilon S^i \}$

Example-1
N=3 ; m=6 (p1,p2,p3)=(1,2,5)
(w1,w2,w3)=(2,3,4) Rule 1:
Initially $S^0 = \{0,0\}$

$S^0{}_1 = \{1,2\}$
$S^1 = S^0 \cup S^0{}_1 = \{(0,0),(1,2);$
$S^1{}_1 = \{(2,3),(3,5)\}$
$S^2 = S^1 \cup S^1{}_1 = \{(0,0),(1,2),(2,3),(3,5)\}$
$S^2{}_1 = \{(5,4),(6,6),(7,7),(8,9)\}$
$S^3 = S^2 \cup S^2{}_1 = \{(0,0),(1,2),(2,3),(3,5),(5,4),(6,6),(7,7),(8,9)\}$
N=3, $2^3 = 8$ pairs

Applying rule to the above pairs, where weight exceeds knapsack capacity discards the pair. In the above (7, 7), (8,9) pairs are discarded.

Rule 2(purging or dominance): Applying rule to the remaining pairs after discarded pairs i.e on 6 pairs

Pairs 3≤5 and 5≥4 pairs in above shown so that pair (3,5) pair discarded.
So the solution pair(s) is (6,6) ;
Solution vector: (p1,p2,p3)=(1,2,5)=>(p1,p3)=(1,5)
(w1,w2,w3)=(2,3,4)=>(w1,w3)=(2,4)
The solution vector is (1,0,1)

```
Alogirthm DKP(p,w,n,m)
{
S^0 :={(0,0)};
for i:= 1 to n-1 do
{
S^i-1 := {(P,W| (P-p_i , W-w_i ) ϵ S^i-1 and W ≤ m};
S^i :=MergePurge(S S^i-1 S^i-1 _1 );
}
(PX,WX):=lastpair in S^n-1 ;
(PX,WX):=(P + p_n , W + w_n ) where W is the largest W in any pair in S^n-1 such
that W + w_n ≤m;
//Trace back for x_n, x_n-1,.....x_1

If (PX>PY) then X_n:= 0;
else X_n :=1;
TraceBackFor(x_n-1,.....x_1 );
}
```

Fig. Informal Knapsack algorithm

4.9 Traveling Salesperson Problem

A salesperson would like to travel from one city to the other (n–1) cities just once then back to the original city, what is the minimum distance for this travel?

The brute-and-force method is trying all possible (n–1)! Permutations of (n–1) cities and picking the path with the minimum distance.

There are a lot of redundant computations for the brute-and-force method such as the permutations of 6 cities are 1234561, …, 1243561, …., 1324561, …, 1342561, …, 1423561, …, 1432561, …

The function g(1,V-{1}) is the length of an optimal salesperson tour. From the principal of optimality it follows that g (1, V-[1}) = $\min_{2\le k\le n}$ {c1k + g{k,V-{1,k})}
Generalizing above one we obtain (for I not belong S) g (i, S) = $\min_{jE\ s}${c_{iJ}+ g(j, S-{j})}
The method computes from vertex 1 backward to the other vertices then return to vertex 1.

Let *g(i,S)* be the length of a shortest path starting at vertex *i*, going through all vertices in *S*, and terminating at vertex 1.

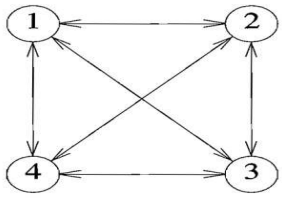
The function g(1,V-{1}) is the length of an optimal salesperson tour. From the principal of optimality it follows that

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

$$g(i, S) = \min_{j\in S}\{c_{ij} + g(j, S-\{j\})\}$$

Eqn. 1 solved for g(1,V-{1}) if we know g (k,V-{1,k}) for all choices of k. The g values can be obtained by using eqn. 2 clearly, g (i, ø) = c_{i1}, 1≤i ≤n. Hence we use g (i, S) for all S size 1.then g (i, S) for S=2 and so on.

Example



(a)

$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

(b)

Thus $g(2,\phi) = c_{21} = 5, g(3,\phi) = c_{31} = 6$, and $g(4,\phi) = c_{41} = 8$. Using (5.21), we obtain

$$\begin{array}{llll} g(2,\{3\}) = c_{23} + g(3,\phi) = 15 & & g(2,\{4\}) = 18 \\ g(3,\{2\}) = 18 & & g(3,\{4\}) = 20 \\ g(4,\{2\}) = 13 & & g(4,\{3\}) = 15 \end{array}$$

Next, we compute $g(i,S)$ with $|S| = 2$, $i \neq 1$, $1 \notin S$ and $i \notin S$.

$$\begin{array}{lll} g(2,\{3,4\}) = \min\{c_{23} + g(3,\{4\}), c_{24} + g(4,\{3\})\} = 25 \\ g(3,\{2,4\}) = \min\{c_{32} + g(2,\{4\}), c_{34} + g(4,\{2\})\} = 25 \\ g(4,\{2,3\}) = \min\{c_{42} + g(2,\{3\}), c_{43} + g(3,\{2\})\} = 23 \end{array}$$

Finally, from (5.20) we obtain

$$\begin{array}{lll} g(1,\{2,3,4\}) = \min\{c_{12} + g(2,\{3,4\}), c_{13} + g(3,\{2,4\}), c_{14} + g(4,\{2,3\})\} \\ = \min\{35, 40, 43\} \\ = 35 \end{array}$$

4.10 Flow shop scheduling

Let n be the no. of jobs, each may be requiring m task.

Let T_{1i}, T_{2i}… T_{mi} where i=1 to n to be performed where T_{mi} is the I^{th} job of m^{th} task. The task 'T_{ji} to be processed on the processor 'T_j' where j=1,2,……m. The time required to complete the task T_{ji} is t_{ji} .

A schedule for 'n' jobs is an assignment of tasks to the time intervals on the processors.

Constraints: No two processors may have more than one task assign to it in anytime interval.

Objective: The objective of flow shop scheduling is to find the optimal finishing time (OFT) of the given schedule 'S'.

Formula: $F(s) = \max_{1 \le i \le n}\{f_i(S)\}$

(Optimal Finishing Time OFT schedule 'S')
Mean flow time for the schedule 'S' is

$$\text{MFT} = 1/n \sum_{1 \le i \le n} (f_i(S))$$

There are 2 possible scheduling:

1) Non-preemptive schedule:-it is in which the processing of a task on any processor is not terminate until the task is completed.
2) Preemptive scheduling:-It is in which the processing of a task on any processor is terminated before the task is completed.

Problem:

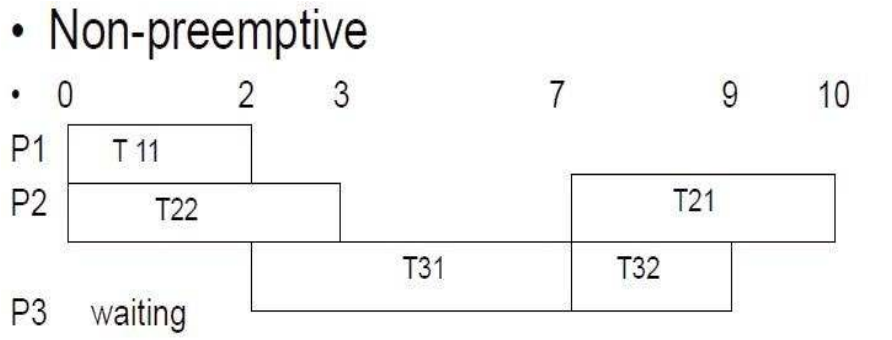
j =

2	0	p1
3	3	p2
5	2	p3

j=schedule t11

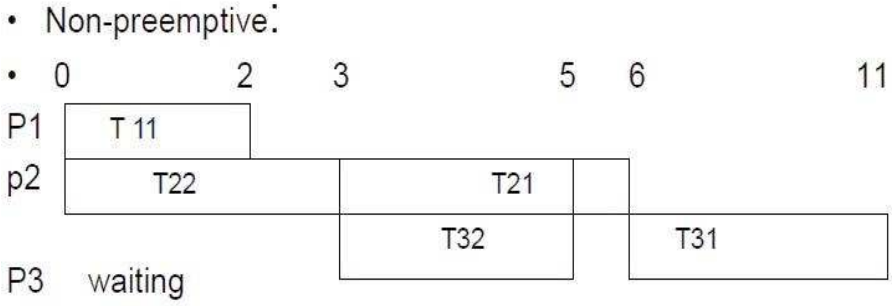
P1, p2, p3= processors or tasks of jobs

T11 –first job of the first task



Optimal Finishing Time=10

Mean Flow Time =1/2(9+10)=9.5



OFS =11

MFT = ½(min+max)=1/2(6+11)=8.5

****

Chapter-5
Basic Traversal and Search Techniques

5.1 Techniques for Binary Trees

Binary Tree

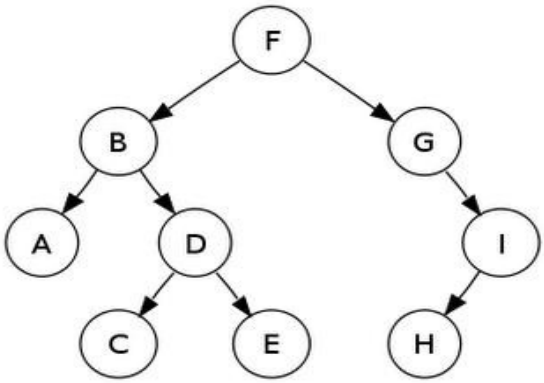
A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint binary trees called the left sub tree and the right sub tree.
In a traversal of a binary tree, each element of the binary tree is visited exactly at once. During the visiting of an element, all actions like clone, display, evaluate the operator etc is taken with respect to the element. When traversing a binary tree, we need to follow linear order i.e. L, D, R where
L->Moving left
D->printing the data
R->moving right

We have three traversal techniques on binary tree. They are

- *In order*
- *Post order*
- *Pre order*

Examples

For fig: 1



In order: A-B-C-D-E-F-G-H-I
Post order: A-C-E-D-B-H-I-G-F
Pre order: F-B-A-D-C-E-G-I-H

Preorder, post order and in order algorithms

Algorithm preorder(x)

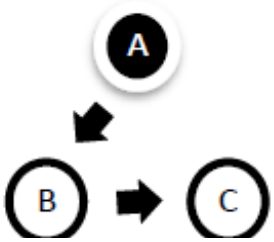
Input: x is the root of a sub tree.

1. If x ≠ NULL

2. **Then** output key(x);

3. Preorder (left(x));

4. Preorder (right(x));



Algorithm postorder(x)

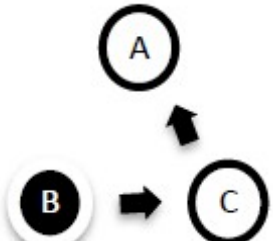
Input: x is the root of a subtree

1. **If** x ≠ NULL

2. **Then** postorder(left(x));;

3. Postorder(right(x));

4. Outputkey(x);



Algorithm inorder(x)

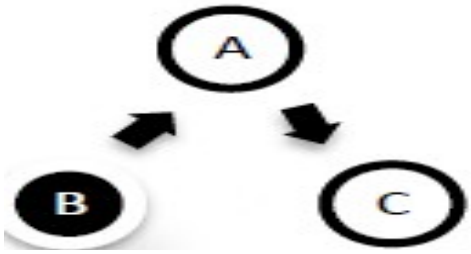
Input: x is the root of a subtree

1. If x≠ null

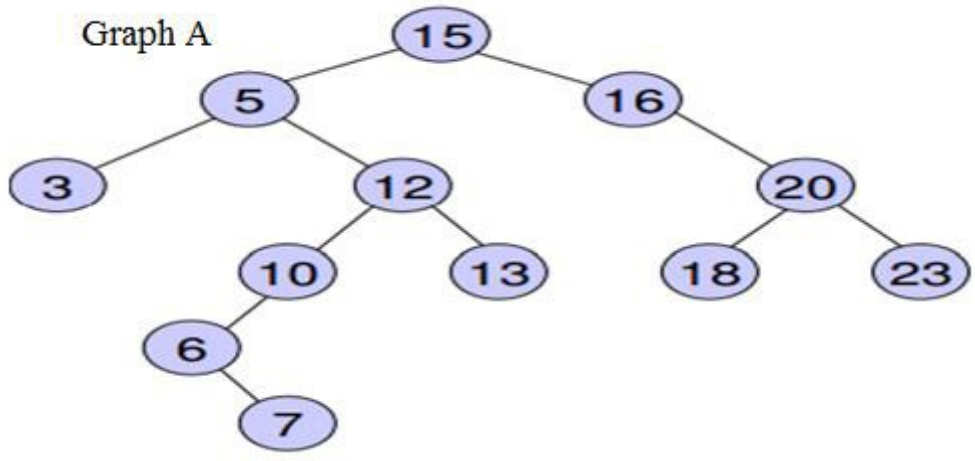
2. **Then** inorder(left(x));

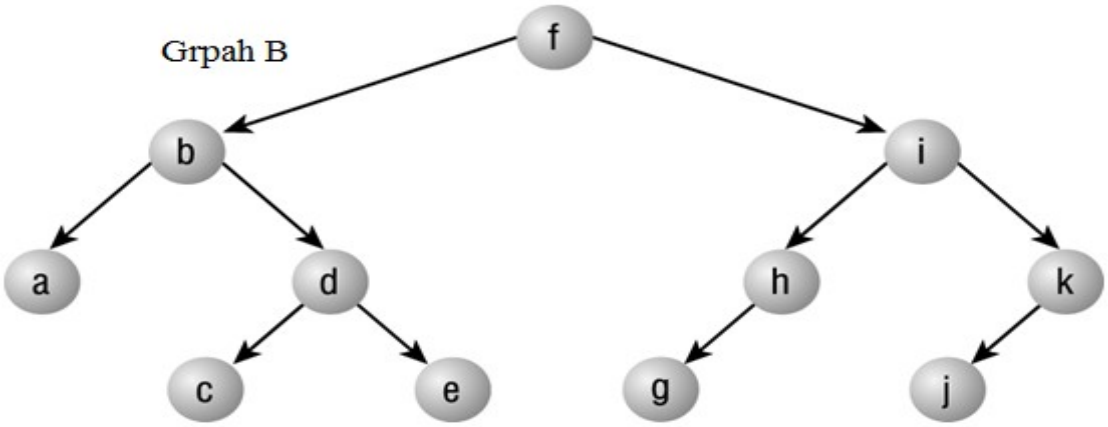
3. Outputkey(x);

4. Inorder(right(x));



Exercises





5.2 Techniques for Graphs

Graph: The sequence of edges that connected two vertices.
A graph is a pair (*V, E*), where
V is a set of nodes, called vertices
E is a collection (can be duplicated) of pairs of vertices, called edges
Vertices and edges are data structures and store elements.

Types of graphs: Graphs are of three types.

a. *Directed/Undirected:* In a directed graph the direction of the edges must be considered

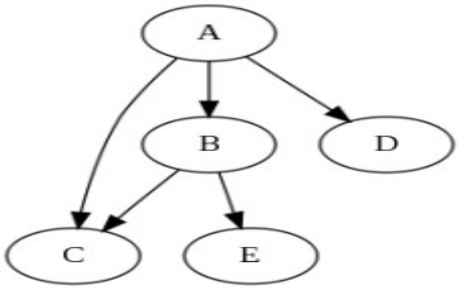


Fig 5.1

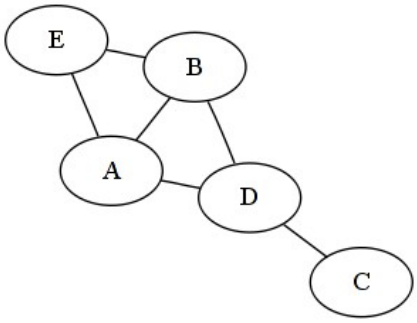


Fig 5.2

b. *Weighted/ Unweighted*: A weighted graph has values on its edge.

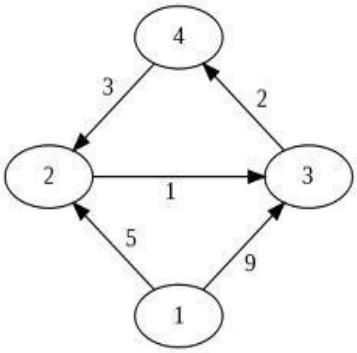


Fig 5.3

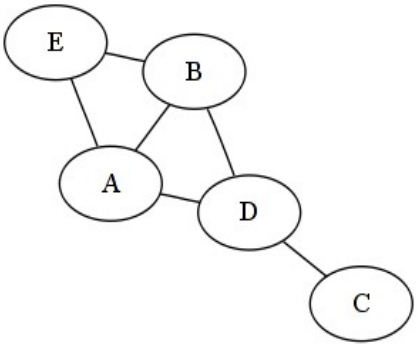
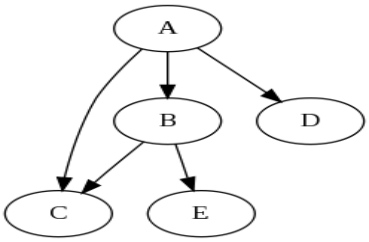
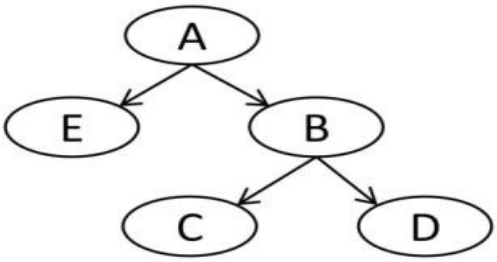


Fig 5.4

c. *Cyclic/Acyclic*: A **cycle** is a path that begins and ends at same vertex and A graph with no cycles is **acyclic.**

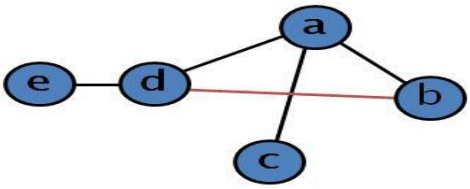




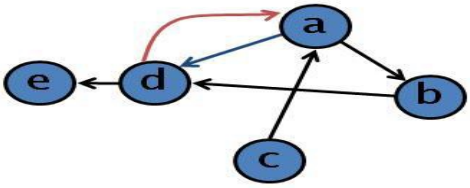
Representation of graphs

Graphs can be represented in three ways

(i) **Adjacency Matrix:** A *V* x *V* array, with matrix[*i*][*j*] storing whether there is an edge between the *ith* vertex and the *jth* vertex. This matrix is also called as "Bit matrix" or "Boolean Matrix"

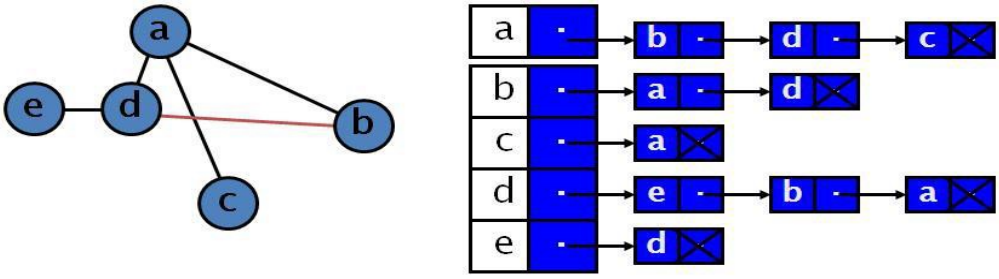


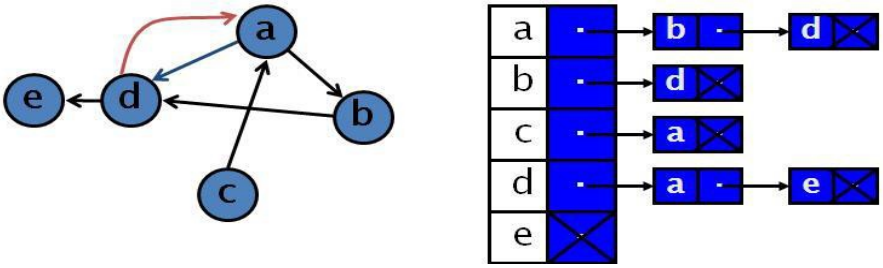
	a	b	c	d	e
a	0	1	1	1	0
b	1	0	0	1	0
c	1	0	0	0	0
d	1	1	0	0	1
e	0	0	0	1	0



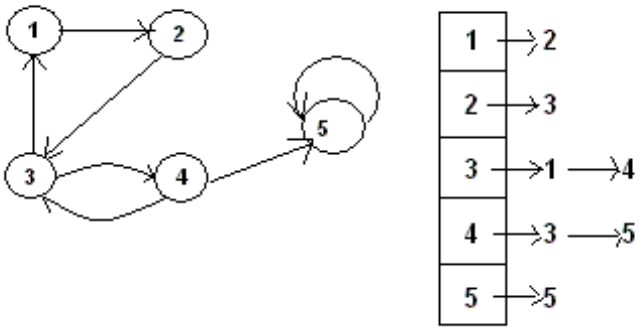
	a	b	c	d	e
a	0	1	0	1	0
b	0	0	0	1	0
c	1	0	0	0	0
d	1	0	0	0	1
e	0	0	0	0	0

(ii) **Adjacency list:** One linked list per vertex, each storing directly reachable vertices .





(iii) Linked List or Edge list:



Graph traversal techniques

"The process of traversing all the nodes or vertices on a graph is called graph traversal".

We have two traversal techniques on graphs
DFS
BFS

Depth First Search

The DFS explore each possible path to its conclusion before another path is tried. In other words go as a far as you can (if u don't have a node to visit), otherwise, go back and try another way. Simply it can be called as "backtracking".

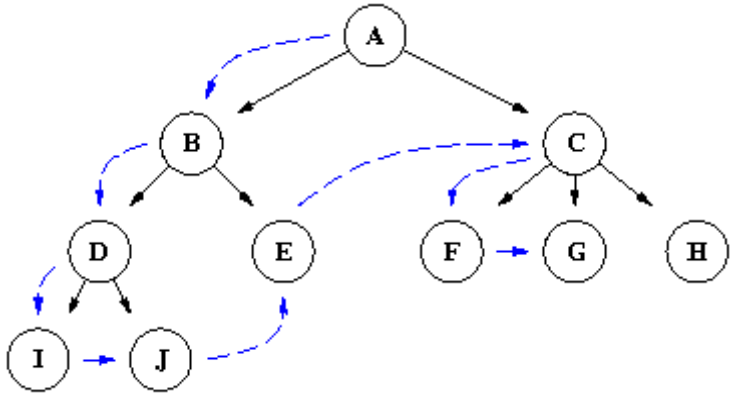
Steps for DFS

(i) Select an unvisited node 'v' visits it and treats it as the current node.
(ii) Find an unvisited neighbor of current node, visit it and make it new current node

(iii) If the current node has no unvisited neighbors, backtrack to its parent and make it as a new current node

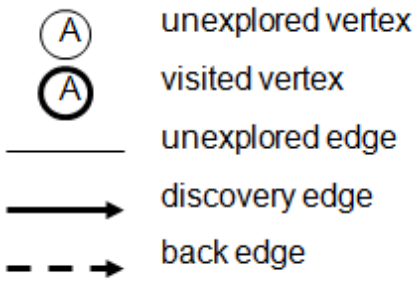
(iv) Repeat steps 2 and 3 until no more nodes can be visited

(v) Repeat from step 1 for remaining nodes also.



Implementation of DFS

DFS (Vertex)
{
Mark **u** as visiting
For each vertex V directly reachable from **u**
If **v** is unvisited
DFS (**v**)
}



Unexplored vertex: The node or vertex which is not yet visited.

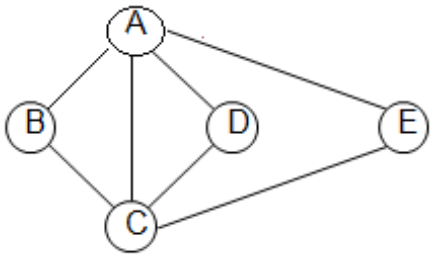
Visited vertex: The node or vertex which is visited is called 'visited vertex' i.e. can be called as "current node".

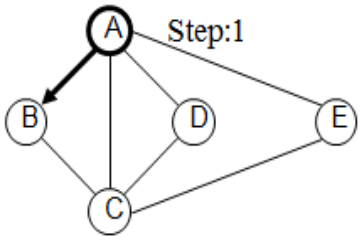
Unexplored edge: The edge or path which is not yet traversed.

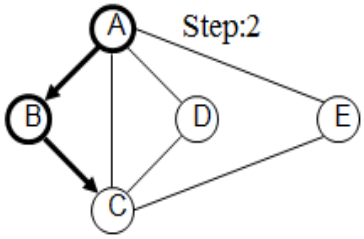
Discovery edge: It is opposite to unexplored edge, the path which is already traversed is known as discovery edge.

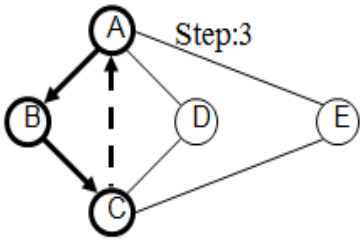
Back edge: If the current node has no unvisited neighbors we need to backtrack to its parent node. The path used in back tracking is called back edge.

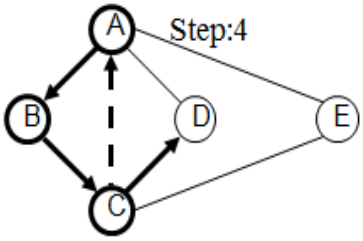
For the following graph the steps for tracing are as follows:

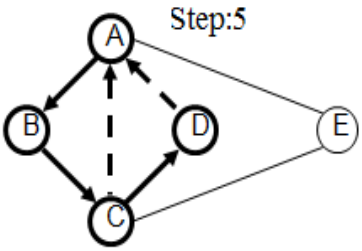


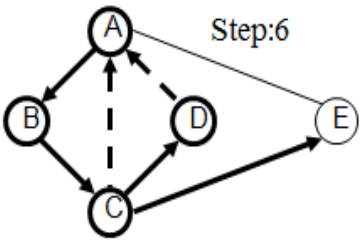


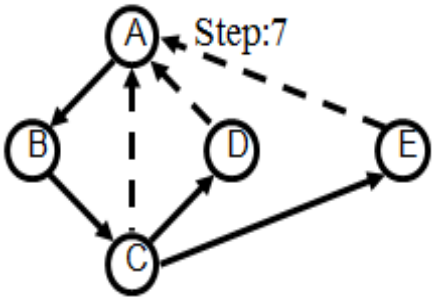








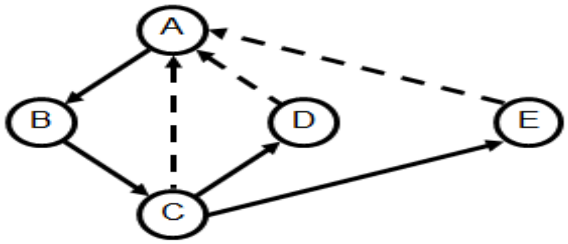




Properties of DFS

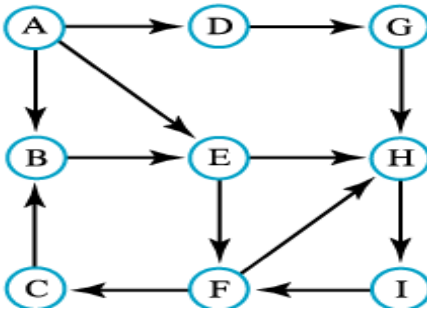
i) DFS (G, *v*) visits all the vertices and edges in the connected component of *v*.

ii) The discovery edges labeled by DFS (G, *v*) form a spanning tree of the connected component of *v*.



Tracing of graph using Depth First Search

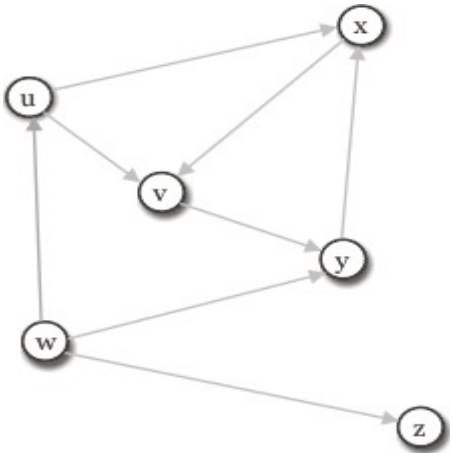
(a)



topVertex	nextNeighbor	Visited vertex	vertexStack (top to bottom)	traversalOrder (front to back)
		A	A	A
A			A	
	B	B	BA	AB
B			BA	
	E	E	EBA	ABE
E			EBA	
	F	F	FEBA	ABEF
F			FEBA	
	C	C	CFEBA	ABEFC
C			FEBA	
F			FEBA	
	H	H	HFEBA	ABEFCH
H			HFEBA	
	I	I	IHFEBA	ABEFCHI
I			HFEBA	
H			FEBA	
F			EBA	
E			BA	
B			A	
A			A	
	D	D	DA	ABEFCHID
D			DA	
	G		GDA	ABEFCHIDG
G			DA	
D			A	
A			*empty*	ABEFCHIDG

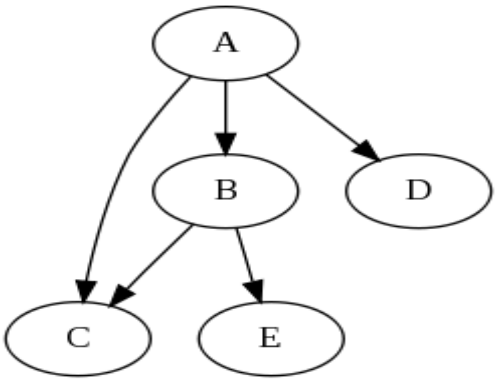
Exercise

1.



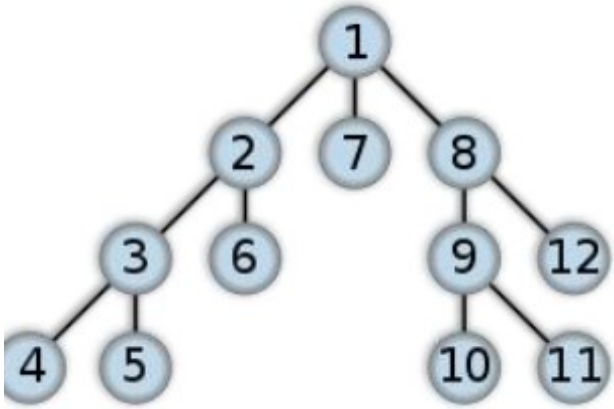
Depth: W-U-V-Y-X-Z

2.



Depth: A-B-C-E-D

3



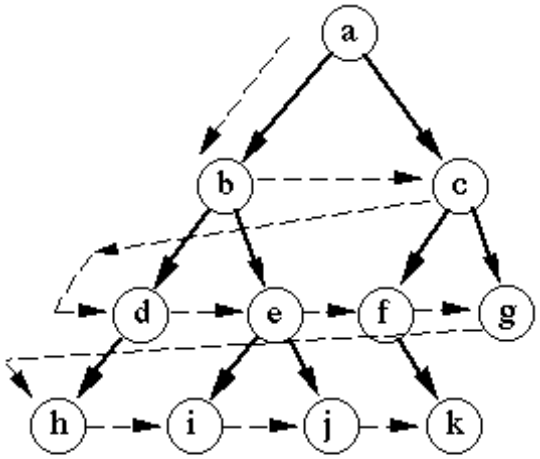
Depth: 1-2-3-4-5-6-7-8-9-10-11-12.

5.3 Breadth First Search

It is one of the simplest algorithms for searching or visiting each vertex in a graph. In this method each node on the same level is checked before the search proceeds to the next level. BFS makes use of a queue to store visited vertices, expanding the path from the earliest visited vertices
Breadth: a-b-c-d-e-f-g-h-i-j-k

Steps for BFS:

1. Mark all the vertices as unvisited.
2. Choose any vertex say 'v', mark it as visited and put it on the end of the queue.
3. Now, for each vertex on the list, examine in same order all the vertices adjacent to 'v'
4. When all the unvisited vertices adjacent to v have been marked as visited and put it on the end (rear of the queue) of the list.
5. Remove a vertex from the front of the queue and repeat this procedure.
6. Continue this procedure until the list is empty.

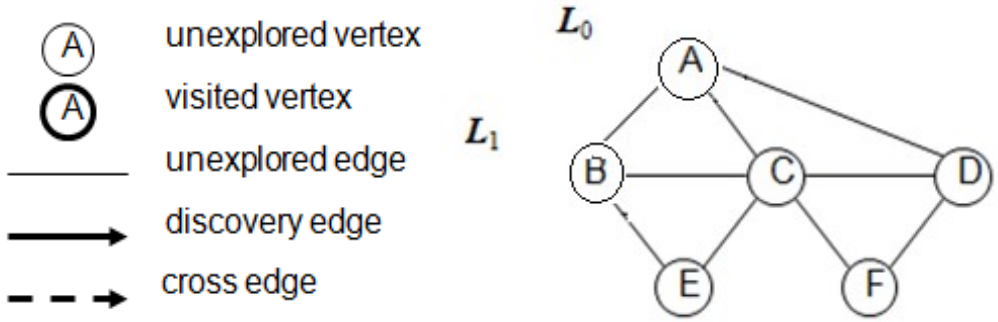


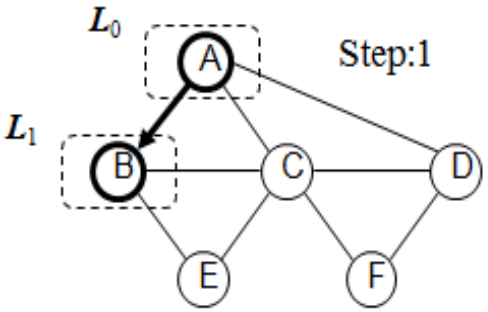
Breadth-first search

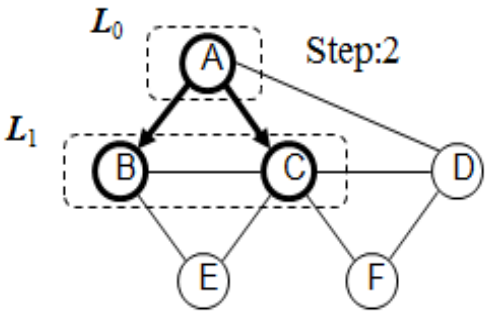
Implementation of BFS

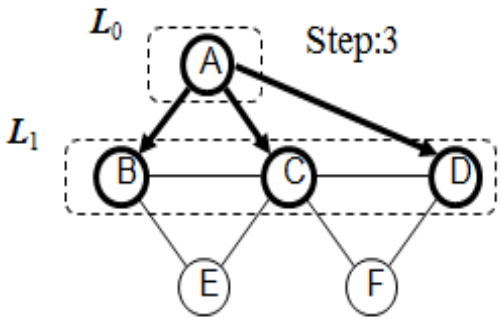
While queue Q not empty
De queue the first vertex **u** from Q
For each vertex **v** directly reachable from **u**
If **v** is unvisited
En queue **v** to Q
Mark **v** as visited
\ Initially all vertices except the start vertex are marked as unvisited and the queue contains the start vertex only.

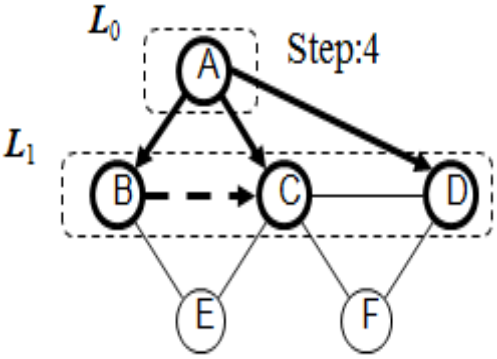
Explored vertex: A vertex is said to be explored if all the adjacent vertices of **v** are visited.
Example 1: Breadth first search for the following graph:

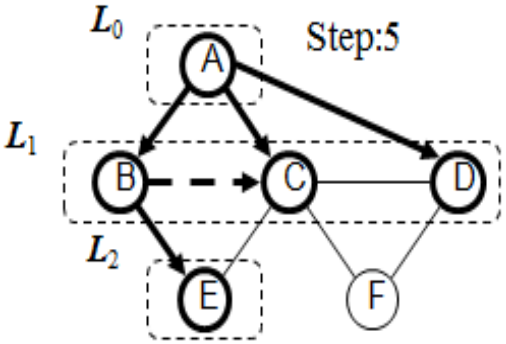

unexplored vertex
visited vertex
unexplored edge
discovery edge
cross edge
L_0
L_1
A
B
C
D
E
F

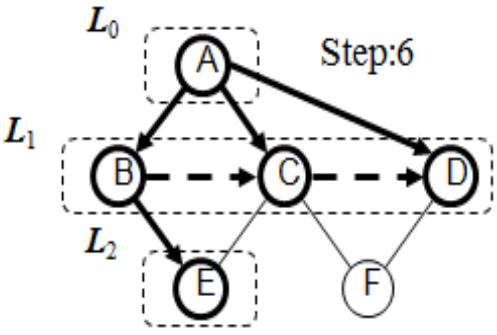

L_0
L_1
Step:1
A
B
C
D
E
F

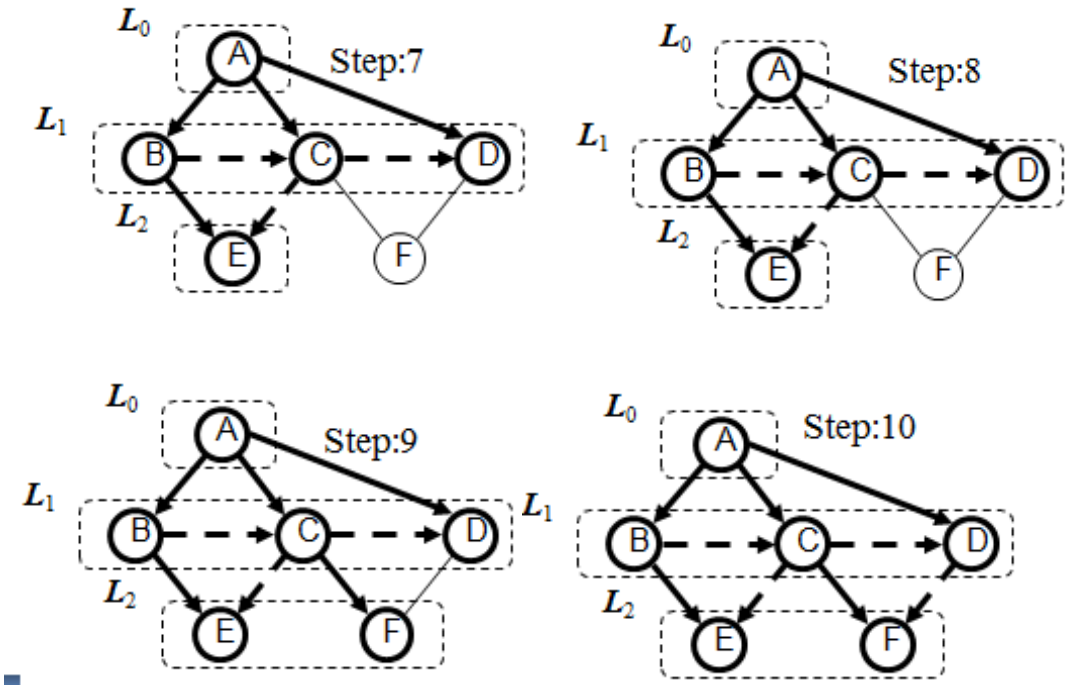

L_0
L_1
Step:2
A
B
C
D
E
F


L_0
L_1
Step:3
A
B
C
D
E
F


L_0
L_1
Step:4
A
B
C
D
E
F


L_0
L_1
L_2
Step:5
A
B
C
D
E
F

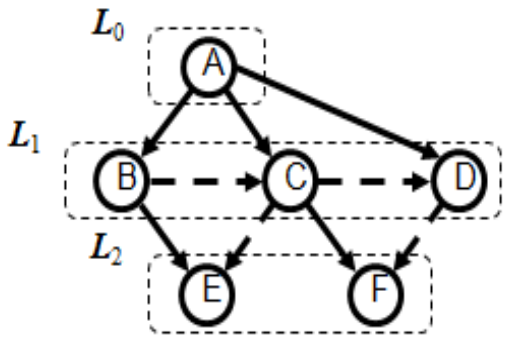

L_0
L_1
L_2
Step:6
A
B
C
D
E
F



Properties of BFS

Notation: ***Gs*** (connected component of ***s***)

i) ***BFS*** **(*G, s*)** visits all the vertices and edges of ***Gs***
ii) The discovery edges labeled by ***BFS*** **(*G, s*)** form a spanning tree ***Ts*** of ***G***
iii) For each vertex ***v*** in ***Li***
a. The path of ***Ts*** from ***s*** to ***v*** has ***i*** edges
b. Every path from ***s*** to ***v*** in ***Gs*** has at least ***i*** edges.

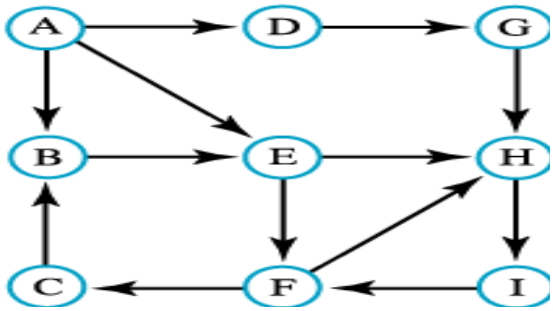


Complexity of BFS

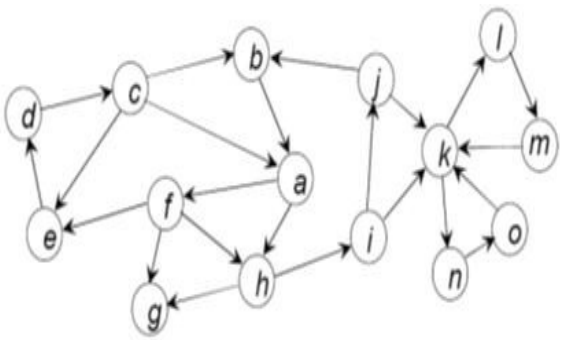
Step1: read a node from the queue O (v) times.
Step2: examine all neighbors, i.e. we examine all edges of the currently read node. Not oriented graph: 2*E edges to examine
*Hence the complexity of BFS is O (V + 2*E)*

Tracing of graph using Breadth first search:

(a)

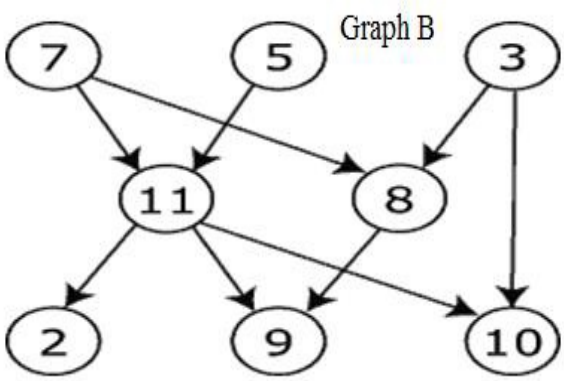


frontVertex	nextNeighbor	Visited vertex	vertexQueue	traversalOrder
		A	A	A
A			*empty*	
	B	B	B	A B
	D	D	B D	A B D
	E	E	B D E	A B D E
B			D E	
D			E	
	G	G	E G	A B D E G
E			G	
	F	F	G F	A B D E G F
	H	H	G F H	A B D E G F H
G			F H	
F			H	
	C	C	H C	A B D E G F H C
H			C	
	I	I	C I	A B D E G F H C I
C			I	
I			*empty*	

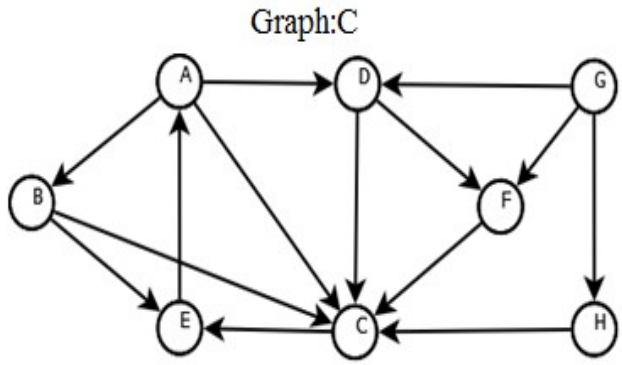


BFS: 7-11-8-2-9-10-5-3

BFS: a f h e g i d j k c l n b m o

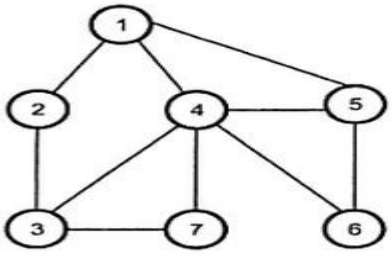


BFS: A-B-C-D-E-F-G-H

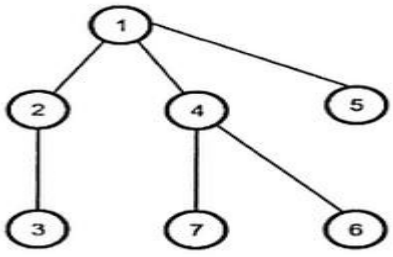


5.4 Connected Components and Spanning Trees

***Connected component*:** If G is connected undirected graph, then we can visit all the vertices of the graph in the first call to BFS. The sub graph which we obtain after traversing the graph using BFS represents the connected component of the graph.



Graph G

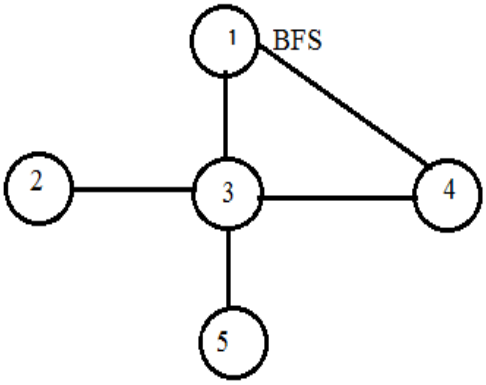


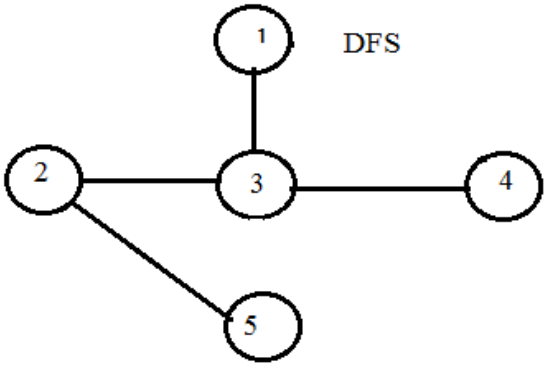
Connected component of G obtained by breadth first search

Thus BFS can be used to determine whether G is connected. All the newly visited vertices on call to BFS represent the vertices in connected component of graph G. The sub graph formed by theses vertices make the connected component.

Spanning tree of a graph*:* Consider the set of all edges (u, w) where all vertices w are adjacent to u and are not visited. According to BFS algorithm it is established that this set of edges give the spanning tree of G, if G is connected. We obtain depth first search spanning tree similarly

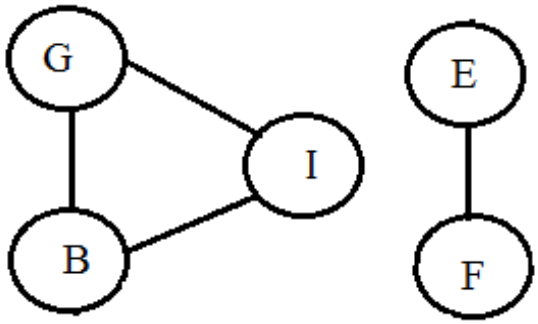
These are the BFS and DFS spanning trees of the graph G

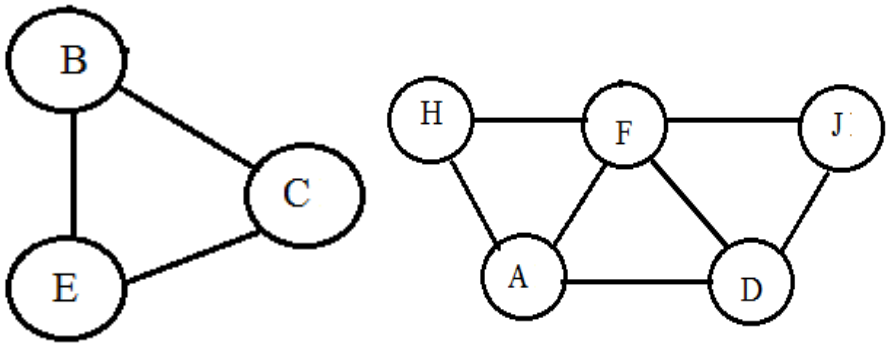




Bi-connected Components

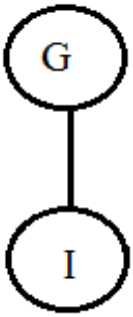
A connected undirected graph is said to be bi-connected if it remains connected after removal of any one vertex and the edges that are incident upon that vertex.
In this we have two components.
i. *Articulation point*: Let G= (V, E) be a connected undirected graph. Then an articulation point of graph 'G' is a vertex whose articulation point of graph is a vertex whose removal disconnects the graph 'G'. It is also known as "cut point".
ii. *Bi-connected graph:* A graph 'G' is said to be bi-connected if it contains no-articulation point.

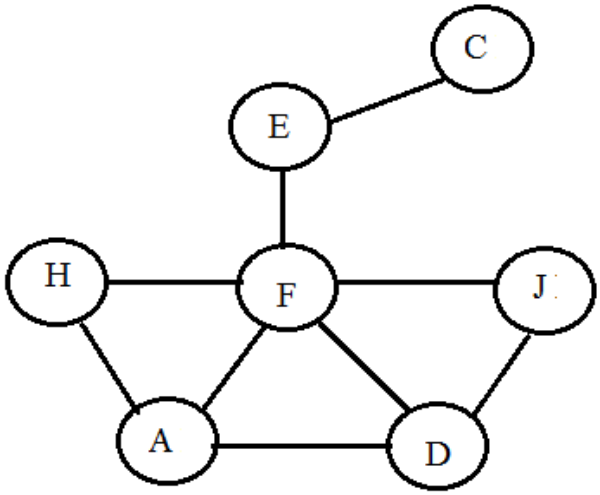




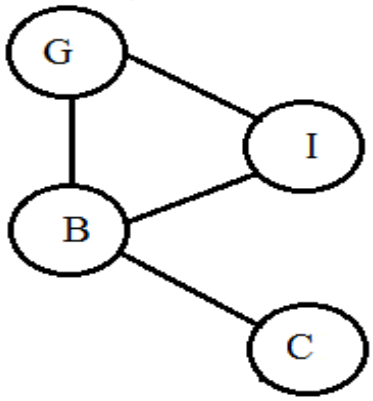
Articulation points for the above undirected graph are B, E, F

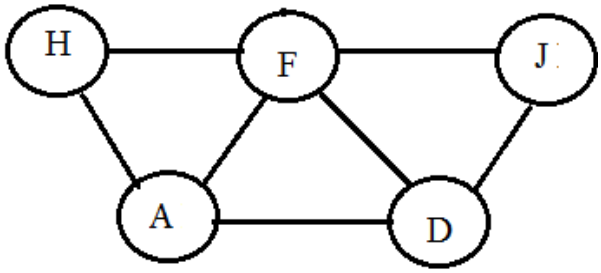
i) After deleting vertex B and incident edges of B, the given graph is divided into two components



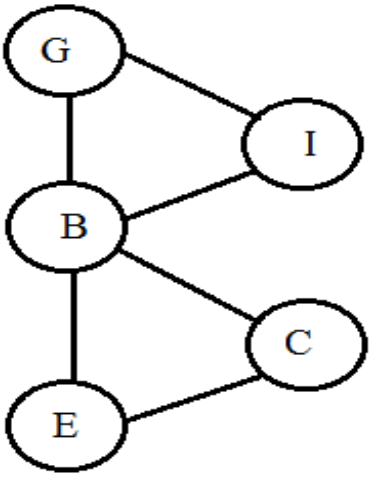


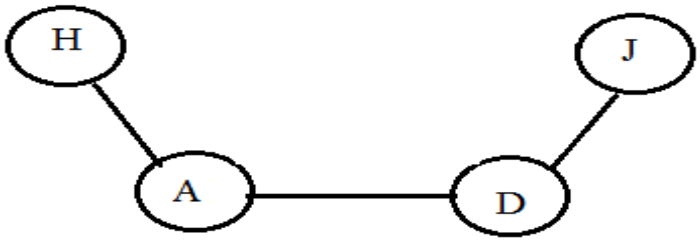
ii) After deleting the vertex E and incident edges of E, the resulting components are





iii) After deleting vertex F and incident edges of F, the given graph is divided into teo components.





Note: If there exists any articulation point, it is an undesirable feature in communication network where joint point between two networks failure in case of joint node fails.

Algorithm to construct the Bi- Connected graph

1. For each articulation point ‘a’ do

2. Let B1, B2, B3 ….Bk are the Bi-connected components

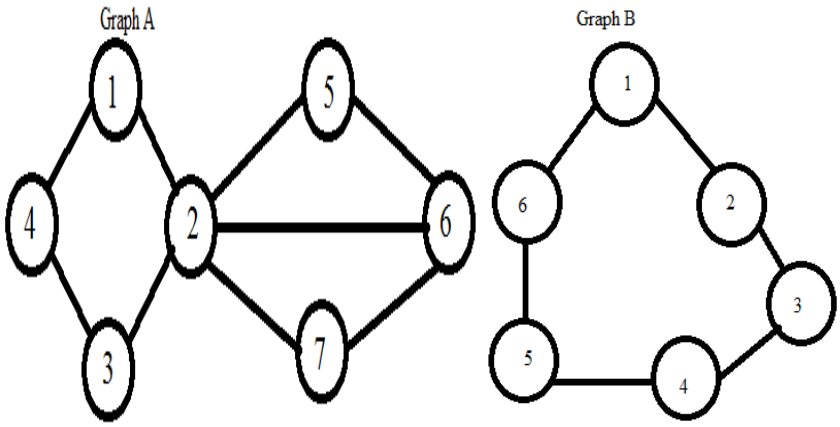
3. Containing the articulation point ‘a’

4. Let Vi E Bi, Vi # a i<=i<=k

5. Add (Vi,Vi+1) to Graph G.

Vi-vertex belong Bi
Bi-Bi-connected component
- i- Vertex number 1 to k
- a- articulation point

Exercise



****

Index