# KIET Group of Institutions, Ghaziabad

## Department of Computer Applications

**(An ISO – 9001: 2015 Certified & 'A' Grade accredited Institution by NAAC)**

# NOTES

**Subject Name**: Data Structures & Analysis of Algorithms **Subject Code**: KCA-205 **Semester**: II

# UNIT-5

Divide and Conquer with Examples Such as Merge Sort, Quick Sort, Matrix Multiplication: Strassen' s Algorithm Dynamic Programming: Dijikstra Algorithm, Bellman Ford Algorithm, Allpair
Shortest Path: Warshal Algorithm, Longest Common Sub-sequence Greedy Programming: Prims and Kruskal algorithm.

### The divide-and-conquer approach

There are many ways to design algorithms. Insertion sort uses an ***incremental*** approach. Now we will see an alternative design approach, known as "divide-and-conquer."

Many useful algorithms are ***recursive*** in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems. These algorithms typically follow a ***divide-and-conquer*** approach: they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem. The divide-and-conquer paradigm involves three steps at each level of the recursion:

**Divide** the problem into a number of subproblems.

**Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

**Combine** the solutions to the subproblems into the solution for the original problem.

The ***merge sort*** algorithm closely follows the divide-and-conquer paradigm. Intuitively,

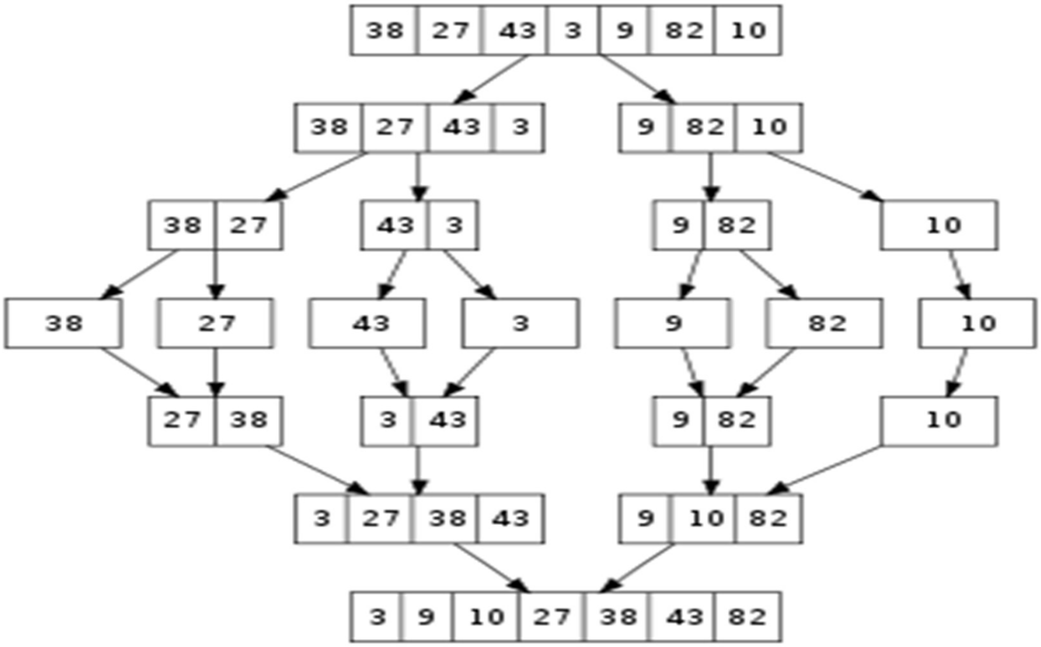
it operates as follows.

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

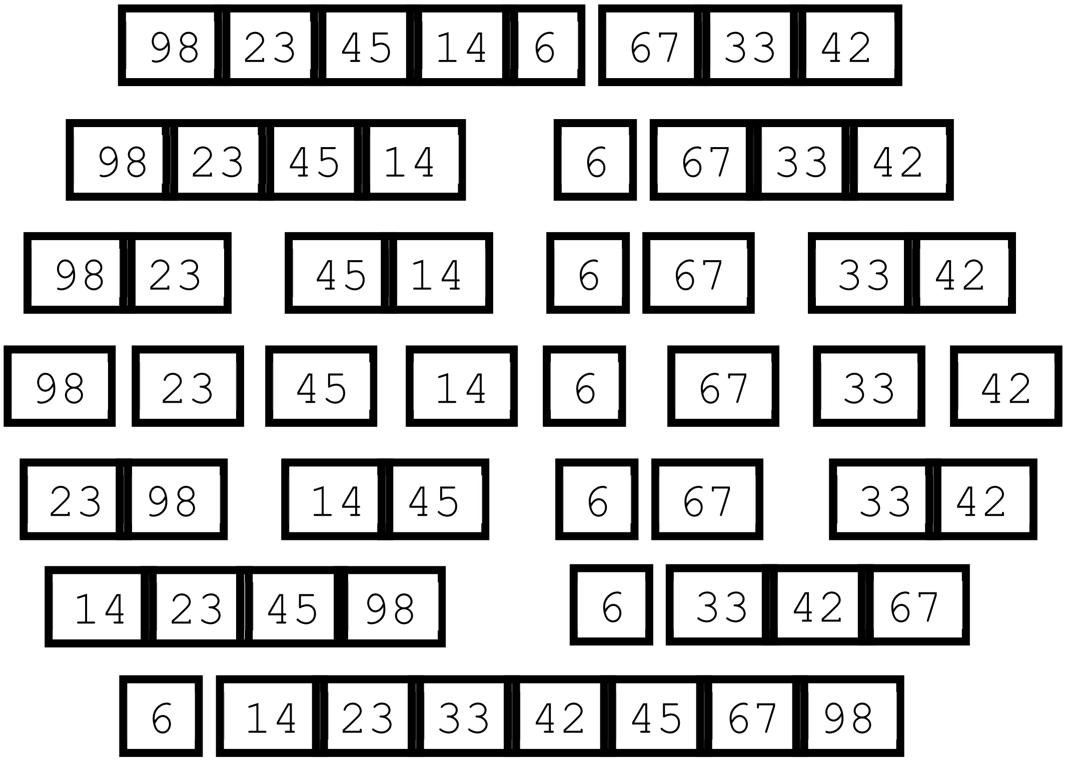
**Conquer:** Sort the two subsequences recursively using merge sort.

**Combine:** Merge the two sorted subsequences to produce the sorted answer.

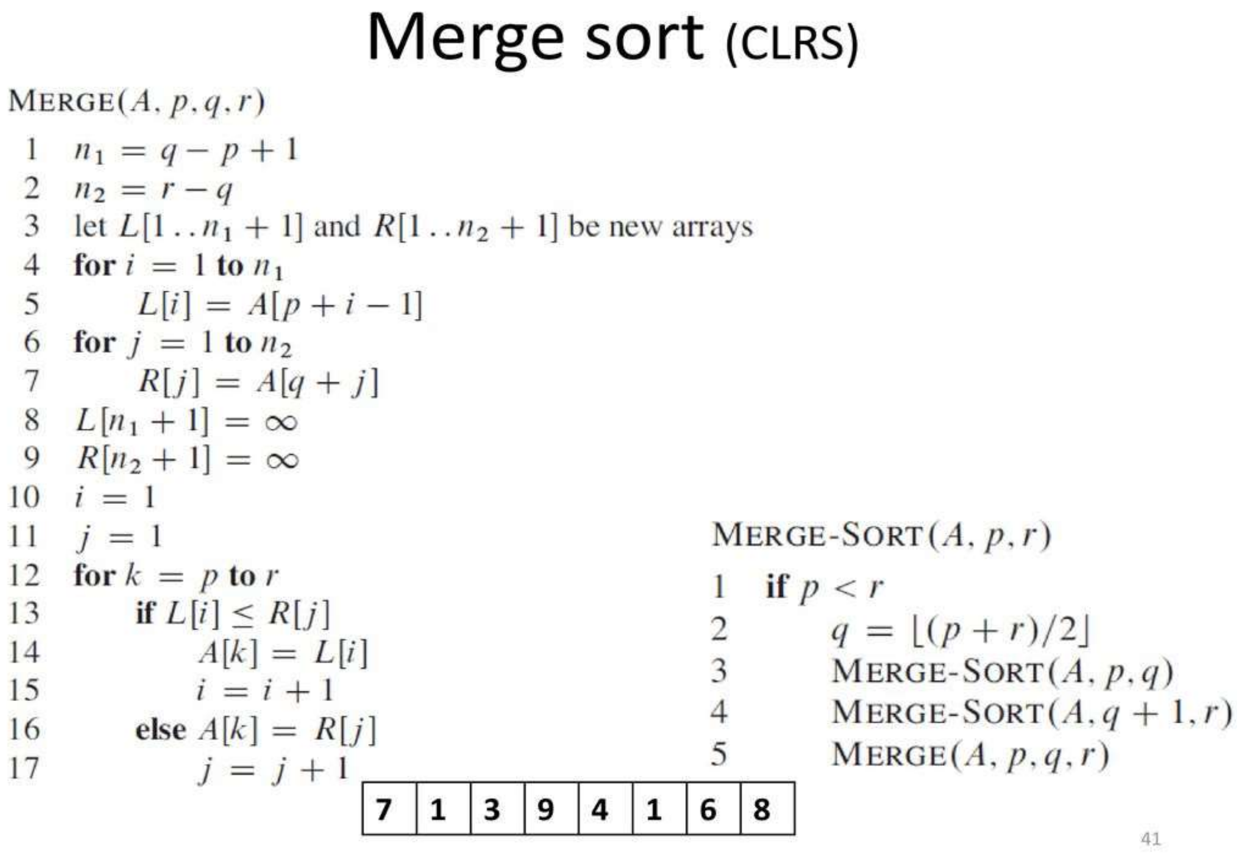
Example-1



Example-2


98 23 45 14 6 | 67 33 42
98 23 45 14 | 6 67 33 42
98 23 | 45 14 | 6 67 | 33 42
98 | 23 | 45 | 14 | 6 | 67 | 33 | 42
23 98 | 14 45 | 6 67 | 33 42
14 23 45 98 | 6 33 42 67
6 14 23 33 42 45 67 98

**Algorithm**



## Merge sort (CLRS)

```
MERGE(A, p, q, r)
1  $n_1 = q - p + 1$
2  $n_2 = r - q$
3  let $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$ be new arrays
4  for $i = 1$ to $n_1$
5      $L[i] = A[p + i - 1]$
6  for $j = 1$ to $n_2$
7      $R[j] = A[q + j]$
8  $L[n_1 + 1] = \infty$
9  $R[n_2 + 1] = \infty$
10 $i = 1$
11 $j = 1$
12 for $k = p$ to $r$
13     if $L[i] \leq R[j]$
14         $A[k] = L[i]$
15         $i = i + 1$
16     else $A[k] = R[j]$
17         $j = j + 1$
```

```
MERGE-SORT(A, p, r)
1  if $p < r$
2      $q = \lfloor (p + r)/2 \rfloor$
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 1 | 3 | 9 | 4 | 1 | 6 | 8 |

41

**Analysis of MERGE function**

- Lines from 1 to 3 will run only once and took constant time $\mathbf{C_1}$
- Line 4 has a loop and line 5 is inside the loop. So the running time of line 4 and 5 will be $\mathbf{C_4.n/2}$
- Similarly line 6 and 7 will took $\mathbf{C_6.n/2}$
- Lines from 8 to 11 will run only once and took constant time $\mathbf{C_8}$
- Line 12 has a loop statement and lines 13 to 17 are inside the loop. This loop will run for maximum size of **n** element so maximum running time is $\mathbf{C_{11}.n}$
- Total running time is $M(n) = C_1 + C_4.n/2 + C_6.n/2 + C_8 + C_{11}.n$

$$= (C_4/2 + C_6/2 + C_{11})n + (C_1 + C_8)$$

$$= an+b \text{ ( equation of line )}$$

$$= cn$$

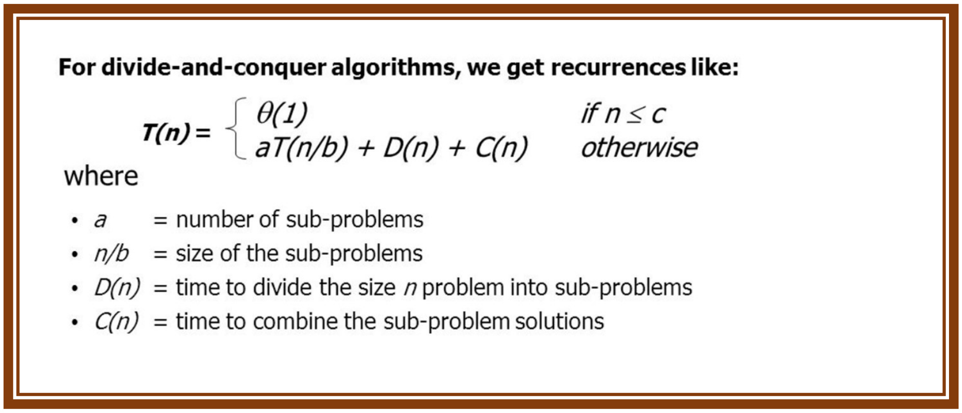
**Analyzing divide-and-conquer algorithms**

When an algorithm contains a **recursive call to itself,** its running time can often be described by a ***recurrence equation*** or ***recurrence***, which describes the overall running time on a problem of size ***n*** in terms of the running time on smaller inputs.

We can then use **mathematical tools to solve the recurrence** and provide bounds on the performance of the algorithm.

A recurrence for the running time of a divide-and-conquer algorithm is based on the three steps. let ***T (n)*** be the running time on a problem of size ***n***.

- If the problem size is small enough, say ***n ≤ c*** for some constant ***c***, the straightforward solution takes constant time, which we write as **Θ(1).**
- If problem divided into ***a*** subproblems, each of which is ***1/b*** the size of the original.



**For divide-and-conquer algorithms, we get recurrences like:**

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

where

- $a$ = number of sub-problems
- $n/b$ = size of the sub-problems
- $D(n)$ = time to divide the size $n$ problem into sub-problems
- $C(n)$ = time to combine the sub-problem solutions

- If we take ***D(n)*** time to divide the problem into subproblems and ***C(n)*** time to combine

The solutions to the subproblems into the solution to the original problem, we get the recurrence

**Analysis of merge sort**

we assume that the original problem size is a **power of 2**. Each divide step then yields **two subsequences** of size exactly ***n/*2**. To set up the recurrence for ***T (n),*** the worst-case running time of merge sort on ***n* numbers**.

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

Merge sort on just **one** element takes **constant time**. When we have ***n* > 1** elements, we break down the running time as follows.

**Divide:** The divide step just computes the middle of the subarray, which takes constant time. Thus, *D(n)* = **Θ(1).**

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

**Combine:** We have computed the running time of **MERGE** procedure on an ***n*-element** subarray takes ***cn*** time

So the recurrence for the worst-case running time ***T (n)*** of merge sort:

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

This can also be written as

Whose solution can be found as **T(n)=Ө(nlog n)**

## Description of Quicksort

Quicksort is based on the three-step process of divide-and-conquer.

**To sort the subarray *A*[*p . . r* ]:**
**Divide:** Partition ***A*[*p . . r* ],** into two (possibly empty) subarrays ***A*[*p . . q* − 1]** and ***A*[*q* + 1 . . *r* ],** such that each element in the first subarray ***A*[*p . . q* − 1]** is ≤ ***A*[*q*]** and ***A*[*q*]** is ≤ each element in the second subarray ***A*[*q* + 1 . . *r* ].**
**Conquer:** Sort the two subarrays by recursive calls to QUICKSORT.
**Combine:** No work is needed to combine the subarrays, because they are sorted in place.

Perform the divide step by a procedure PARTITION, which returns the index *q* that marks the position separating the subarrays.

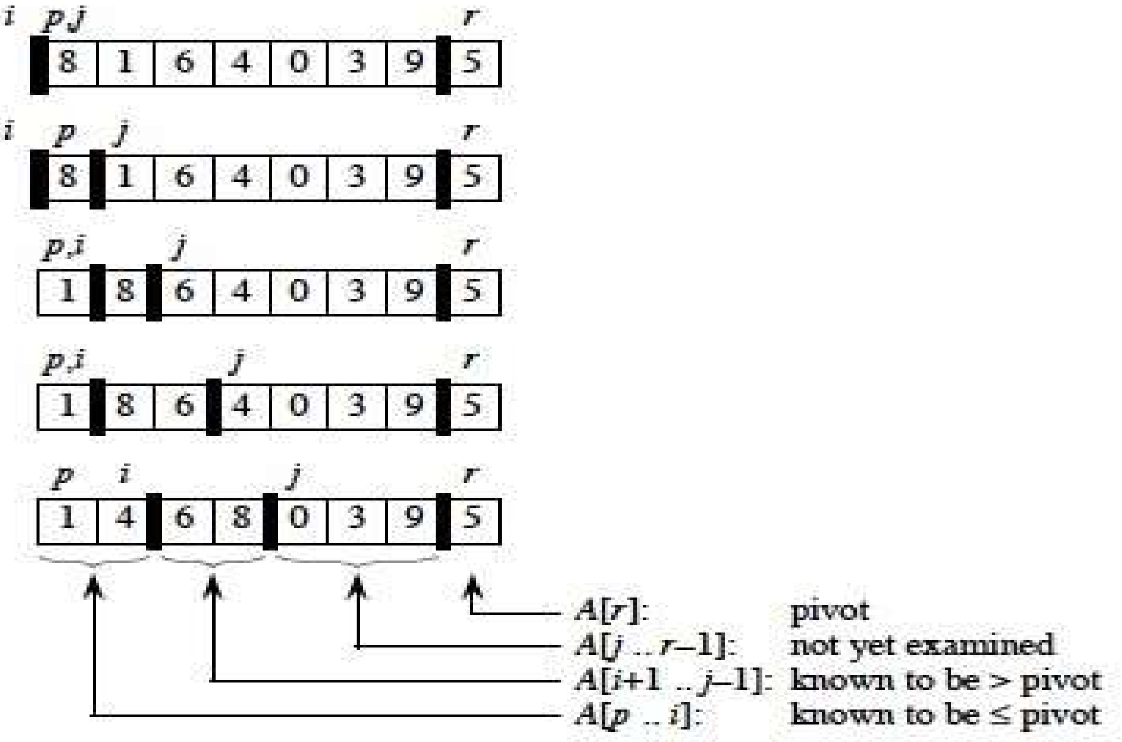
**Initial call is QUICKSORT *(A,* 1*, n)***

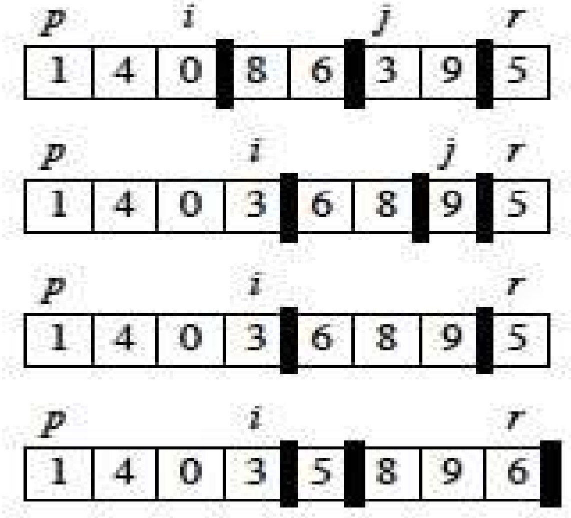
```
QUICKSORT (A, p, r)
    if $p < r$
    then $q \leftarrow$ PARTITION(A, p, r )
    QUICKSORT (A, p, $q - 1$)
    QUICKSORT (A, $q + 1$, r)
```

**Partitioning**
Partition subarray *A* [*p . . . r]* by the following procedure:

```
PARTITION (A, p, r)
    $x \leftarrow A[r]$
    $i \leftarrow p - 1$
    for $j \leftarrow p$ to $r - 1$
        if $A[j] \le x$
            $i \leftarrow i + 1$
            exchange $A[i] \leftrightarrow A[j]$
    exchange $A[i + 1] \leftrightarrow A[r]$
    return $i + 1$
```

PARTITION always selects the **last element *A*[*r* ]** in the subarray *A*[*p . . r* ] as the ***pivot*** the element around which to partition.
As the procedure executes, the array is partitioned into four regions, some of which may be empty:





**Performance of Quicksort**

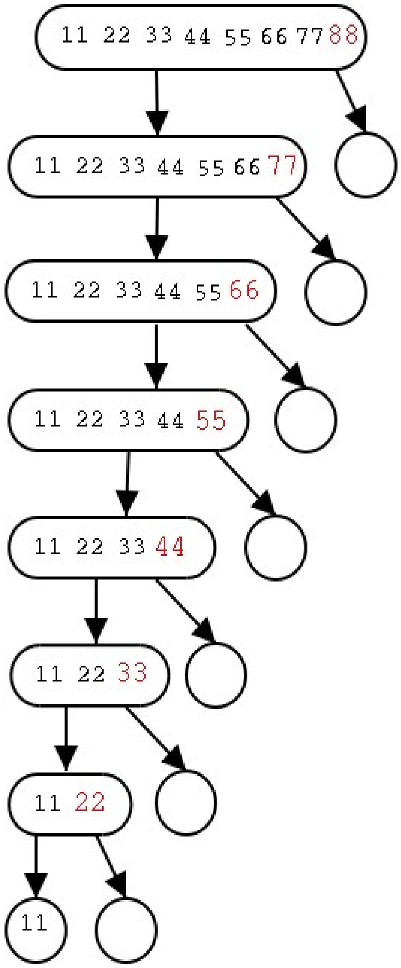
The running time of Quicksort depends on the partitioning of the subarrays:

- If the subarrays are balanced, then Quicksort can run as fast as mergesort.
- If they are unbalanced, then Quicksort can run as slowly as insertion sort.

**Worst case**

- Occurs when the subarrays are completely unbalanced.
- Have 0 elements in one subarray and *n* − 1 elements in the other subarray.

Look at following example where input data is reversely sorted and we are trying to partition it around pivot element



**Get the recurrence**

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= O(n^2) \end{aligned}$$

**Same running time as insertion sort.**

*Note: In fact, the worst-case running time occurs when Quicksort takes a sorted array as input, but insertion sort runs in O(n) time in this case.*

**Best case**

- Occurs when the subarrays are completely balanced every time.
- Each subarray has ≤ $n/2$ elements.
- Get the recurrence

$$T(n) = 2T(n/2) + \Theta(n)$$

$$= O(n \lg n).$$

**Balanced partitioning**

- QuickPort's average running time is much closer to the best case than to the worst case.
- Imagine that PARTITION always produces a 9-to-1 split.
- Get the recurrence

$$T(n) \leq T(9n/10) + T(n/10) + \Theta(n)$$

$$= O(n \lg n).$$

*Recall : It's like the one for* $T(n) = T(n/3) + T(2n/3) + O(n)$.

Except that here the constants are different; we get $\log_{10} n$ full levels and $\log_{10/9} n$ levels that are nonempty. • As long as it's a constant, the base of the log doesn't matter in asymptotic notation. • Any split of constant proportionality will yield a recursion tree of depth $O(\lg n)$.

## Matrix Multiplication

Given two square matrices A and B of size n x n each, find their multiplication matrix.

***Classical Method***

Following is a simple way to multiply two matrices.

```
void multiply(int A[][N], int B[][N], int C[][N])
{
   for (int i = 0; i < N; i++)
   {
      for (int j = 0; j < N; j++)
      {
         C[i][j] = 0;
         for (int k = 0; k < N; k++)
         {
            C[i][j] += A[i][k]*B[k][j];
         }
      }
   }
}
```

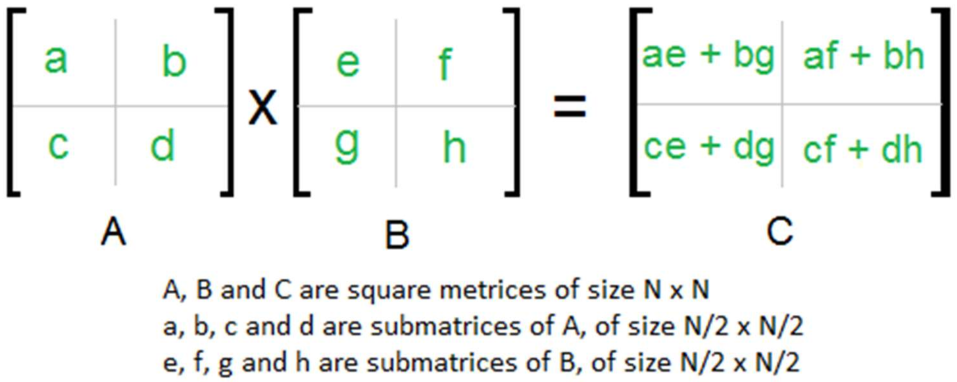
**Time Complexity of above method is O(N$^3$).**

## Matrix Multiplication using Divide and Conquer

Following is simple Divide and Conquer method to multiply two square matrices.

1. Divide a matrix of order of 2*2 recursively till we get the matrix of 2*2.
2. Use the previous set of formulas to carry out 2*2 matrix multiplication.
3. In this eight multiplication and four additions are performed.
4. Combine the result of two matrixes to find the final product or final matrix.

Calculate following values recursively. ae + bg, af + bh, ce + dg and cf + dh.



$$\underbrace{\begin{bmatrix} a & b \\ c & d \end{bmatrix}}_{A} \times \underbrace{\begin{bmatrix} e & f \\ g & h \end{bmatrix}}_{B} = \underbrace{\begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}}_{C}$$

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

**Complexity of multiplication of two matrix using divide and conquer**

Addition of two matrices takes O(N$^2$) time. So the time complexity can be written as
T(N) = 8T(N/2) + O(N$^2$)
From Master's Theorem, time complexity of above method is O(N$^3$). which is unfortunately same as the above classical method.
**Simple Divide and Conquer also leads to O(N$^3$), can there be a better way?** In the above divide and conquer method, the main component for high time complexity is 8 recursive calls. The idea of **Strassen's method** is to reduce the number of recursive calls to 7

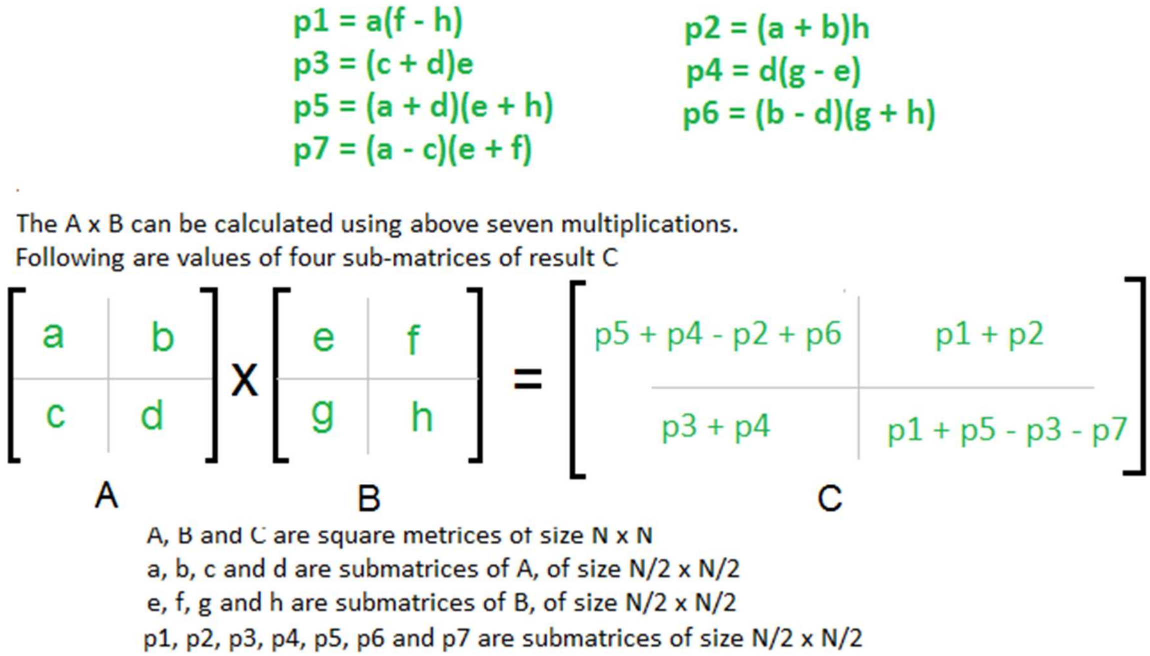
**Introduction to Strassen**

**Strassen** in 1969 which gives an overview that how we can find the multiplication of two **2*2 dimension matrix by the brute-force algorithm**. But by using divide and conquer technique the overall complexity for multiplication two matrices is reduced. This happens by decreasing the total number if multiplication performed at the expenses of a slight increase in the number of addition.

For multiplying the two 2*2 dimension matrices **Strassen's** used some formulas in which there are seven multiplication and eighteen addition, subtraction, and in brute force algorithm, there is eight multiplication and four addition. The utility of Strassen's formula is shown by its asymptotic superiority when order **n** of matrix reaches infinity. Let us consider two matrices **A** and **B**, **n*n** dimension, where **n** is a power of two. It can be observed that we can contain four **n/2*n/2** submatrices from **A**, **B** and their product **C**. **C** is the resultant matrix of **A** and **B**.

**Procedure of Strassen matrix multiplication**

In **Strassen's matrix multiplication** there are seven multiplication and four addition, subtraction in total.



$p1 = a(f - h)$ $\quad$ $p2 = (a + b)h$
$p3 = (c + d)e$ $\quad$ $p4 = d(g - e)$
$p5 = (a + d)(e + h)$ $\quad$ $p6 = (b - d)(g + h)$
$p7 = (a - c)(e + f)$

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\underset{A}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} \times \underset{B}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}} = \underset{C}{\begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}}$$

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

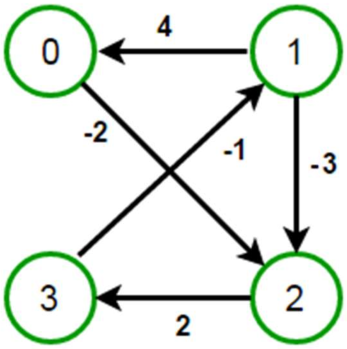
**Time Complexity of Strassen's Method**

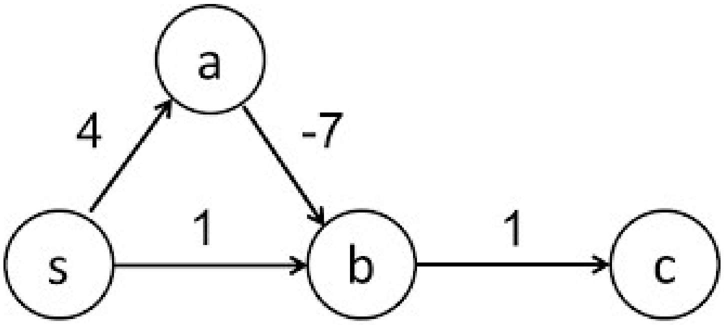
Addition and Subtraction of two matrices takes O($N^2$) time. So time complexity can be written as T(N) = 7T(N/2) + O($N^2$)

From Master's Theorem, time complexity of above method is $O(N^{Log7})$ which is approximately **$O(N^{2.8074})$**

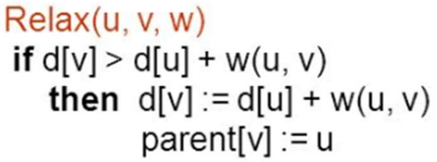
The ***Bellman-Ford algorithm*** solves the single-source shortest-paths problem in the general case in which edge weights may be negative. Given a weighted, directed graph *G = (V, E)* with source *s* and weight function *w* : *E* → **R**, the Bellman-Ford algorithm returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest paths and their weights.

A **negative weight cycle** is a **cycle** with **weights** that sum to a **negative** number. The Bellman-Ford algorithm propagates correct distance estimates to all nodes in a graph in **V**-1 steps, unless there is a **negative weight cycle**. If there is a **negative weight cycle**, you can go on relaxing its nodes indefinitely.



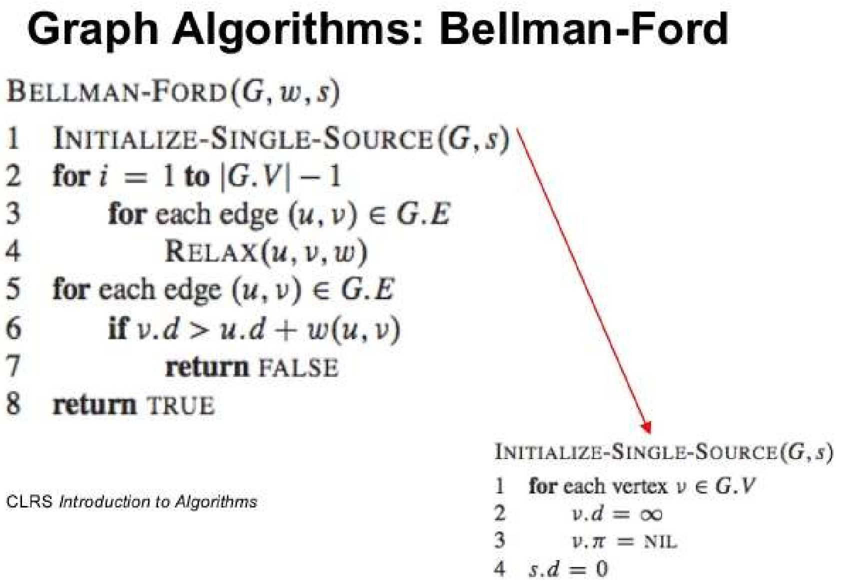


The process of ***relaxing*** an edge *(u, v)* consists of testing whether we can improve the shortest path to *v* found so far by going through *u* and, if so, updating *d*[*v*] and *π*[*v*]. A relaxation step may decrease the value of the shortest-path estimate *d*[*v*] and update *v*'s predecessor field *π*[*v*]. The following code performs a relaxation step on edge *(u, v)*.



```
Relax(u, v, w)
 if d[v] > d[u] + w(u, v)
    then  d[v] := d[u] + w(u, v)
          parent[v] := u
```

## Bellman Ford Algorithm



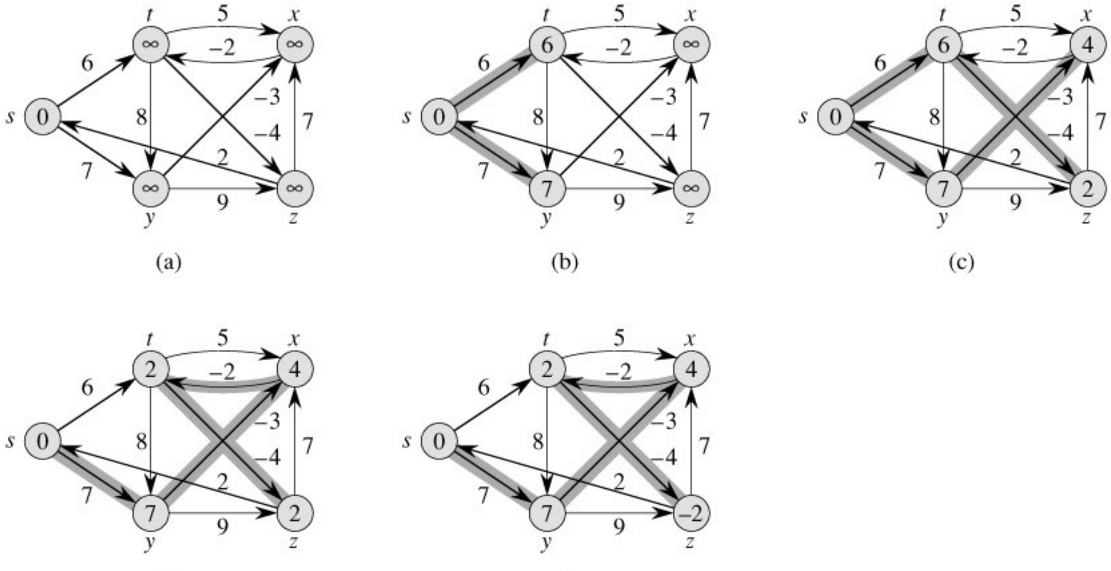
### Graph Algorithms: Bellman-Ford

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| − 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

```
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```

CLRS *Introduction to Algorithms*

## Example



(a) (b) (c)

(d) (e)

## Analysis of Algorithm

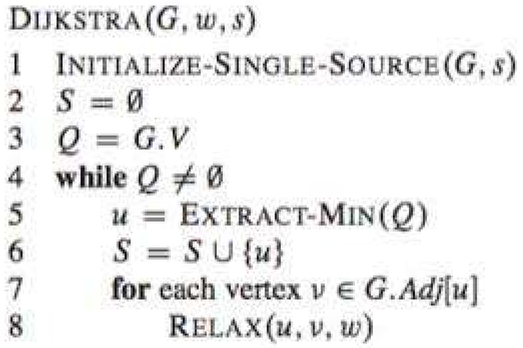
The Bellman-Ford algorithm runs in time *O(V E)*, since the initialization in line 1 takes *Θ(V)* time, each of the |*V*|−1 passes over the edges in lines 2–4 takes Θ*(E)* time, and the **for** loop of lines 5–7 takes *O(E)* time.

**Dijkstra's algorithm**

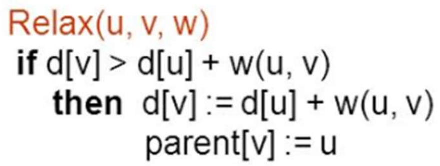
Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph *G* = *(V, E)* for the case in which all edge weights are **nonnegative.**

therefore, we assume that *w(u, v)* ≥ 0 for each edge *(u, v)* ɛ̇ *E*.

Dijkstra's algorithm maintains a set *S* of vertices whose final shortest-path weights from the source *s* have already been determined. The algorithm repeatedly selects the vertex *u* ɛ̇ *V* − *S* with the minimum shortest-path estimate, adds *u* to *S*, and relaxes all edges leaving *u*. In the following implementation, we use a min-priority queue *Q* of vertices, keyed by their *d* values.



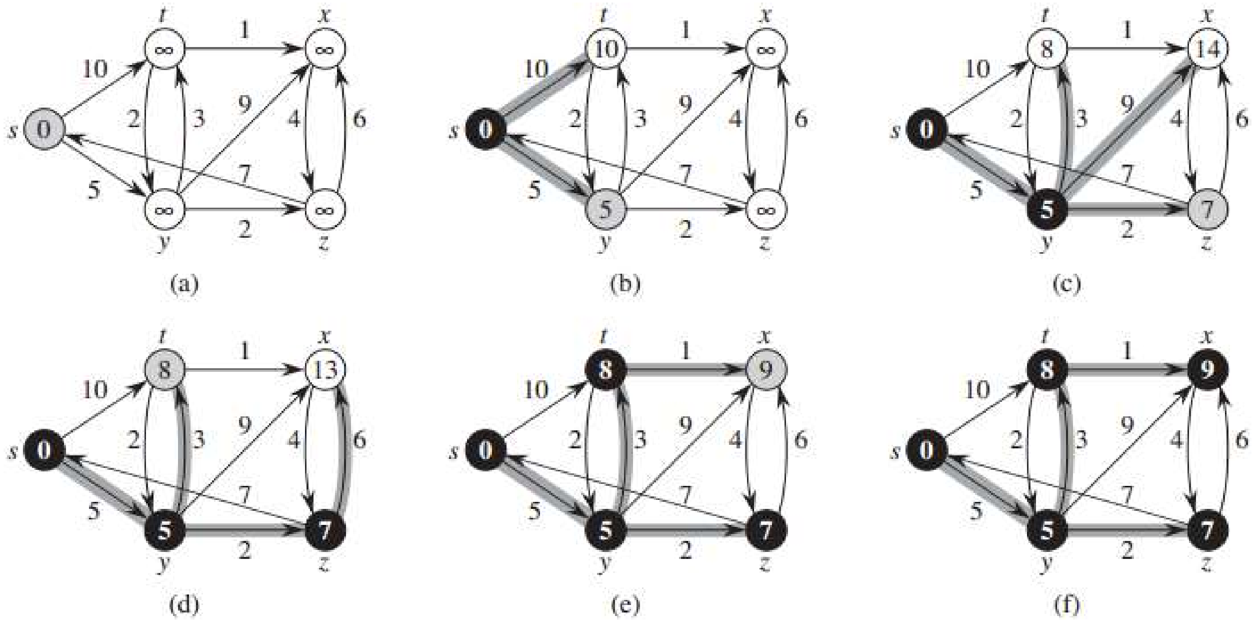
```
DIJKSTRA($G, w, s$)
INITIALIZE-SINGLE-SOURCE($G, s$)
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$
    $u$ = EXTRACT-MIN($Q$)
    $S = S \cup \{u\}$
    for each vertex $v \in G.Adj[u]$
        RELAX($u, v, w$)
```



```
Relax(u, v, w)
if d[v] > d[u] + w(u, v)
   then  d[v] := d[u] + w(u, v)
         parent[v] := u
```

**Because Dijkstra's algorithm always chooses the "lightest" or "closest" vertex in V − S to add to set S, we say that it uses a greedy strategy.**

**<u>Example</u>**



(a) (b) (c)

(d) (e) (f)

**Analysis of Algorithm**

The running time of Dijkstra's algorithm depends on how the min-priority queue is implemented. Suppose the graph has V vertices and E edges. We perform V "find minimum-distance vertex" operations, and at most E "update distance" operations. If we use a priority queue implemented with a min heap (the distance is the priority), each operation takes O(logV) time, so the runtime is O((V+E)logV).

**Dynamic programming,**

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. divide-and-conquer algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share subsubproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems. A dynamic-programming algorithm solves every subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subsubproblem is encountered.

Dynamic programming is typically applied to ***optimization problems***. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution *an* optimal solution to the problem, as opposed to *the* optimal solution, since there may be several solutions that achieve the optimal value.

The development of a dynamic-programming algorithm can be broken into a sequence of four steps.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Steps 1-3 form the basis of a dynamic-programming solution to a problem. Step 4 can be omitted if only the value of an optimal solution is required. When we do perform step 4, we sometimes maintain additional information during the computation in step 3 to ease the construction of an optimal solution.

**Longest common subsequence**

The next problem we shall consider is the longest-common-subsequence problem. A subsequence of a given sequence is just the given sequence with some elements (possibly none) left out. Formally, given a sequence $X = \langle x_1, x_2, . . . , x_m\rangle$ , another sequence $Z = \langle z_1, z_2, . . . , z_k\rangle$ is a ***subsequence*** of $X$ if there exists a strictly increasing sequence $\langle i_1, i_2, . . . , i_k\rangle$ of indices of $X$ such that for all $j = 1, 2, . . . , k$, we have $x_{ij} = z_j$. For example, $Z = \langle B, C, D, B\rangle$ is a subsequence of $X = \langle A, B, C, B, D, A, B\rangle$ with corresponding index sequence $\langle 2, 3, 5, 7\rangle$ .

Given two sequences $X$ and $Y$, we say that a sequence $Z$ is a ***common subsequence*** of $X$ and $Y$ if $Z$ is a subsequence of both $X$ and $Y$. For example, if $X = \langle A, B, C, B, D, A, B\rangle$ and $Y = \langle B, D, C, A, B, A\rangle$, the sequence $\langle B, C, A\rangle$ is a common subsequence of both $X$ and $Y$. The sequence $\langle B, C, A\rangle$ is not a *longest* common subsequence $\langle$LCS$\rangle$ of $X$ and $Y$, however, since it has length 3 and the sequence $\langle B, C, B, A\rangle$, which is also common to both $X$ and $Y$, has length 4. The sequence $\langle B, C, B, A\rangle$ is an LCS of $X$ and $Y$, as is the sequence $\langle B, D, A, B\rangle$, since there is no common subsequence of length 5 or greater.

Like the matrix-chain multiplication problem, our recursive solution to the LCS problem involves establishing a recurrence for the cost of an optimal solution. Let us define $c[i, j]$ to be the length of an LCS of the sequences $X_i$ and $Y_j$. If either $i = 0$ or $j = 0$, one of the sequences has length 0, so the LCS has length 0. The optimal substructure of the LCS problem gives the recursive formula

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 , \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j , \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j . \end{cases} \qquad (16.5)$$

**Computing the length of an LCS**

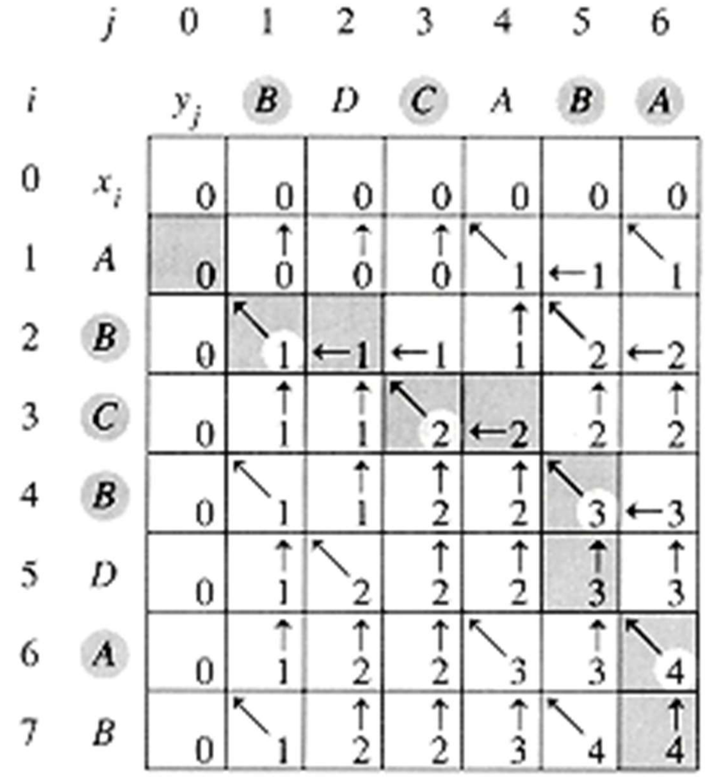
Based on equation we could easily write an exponential-time recursive algorithm to compute the length of an LCS of two sequences. Since there are only $\Theta(mn)$ distinct subproblems, however, we can use dynamic programming to compute the solutions bottom up.

```
LCS-LENGTH(X,Y)
m ← length[X]
n ← length[Y]
for i ← 1 to m
    do c[i,0] ← 0
for j ← 0 to n
    do c[0, j] ← 0
for i ← 1 to m
    do for j ← 1 to n
        do if x_i = y_j
            then c[i, j] ← c[i - 1, j -1] + 1
                b[i, j] ← "↖"
            else if c[i - 1, j] ≥ c[i, j - 1]
                then c[i, j] ← c[i - 1, j]
                    b[i, j] ← "↑"
                else c[i, j] ← c[i, j -1]
                    b[i, j] ← "←"
return c and b
```

the tables produced by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The running time of the procedure is $O(mn)$, since each table entry takes $O(1)$ time to compute.

### Constructing an LCS

The *b* table returned by LCS-LENGTH can be used to quickly construct an LCS of $X = \langle x_1, x_2, ..., x_m \rangle$ and $Y = \langle y_1, y_2, ..., y_n \rangle$. We simply begin at *b*[*m, n*] and trace through the table following the arrows. Whenever we encounter a "↖ in entry *b*[*i,j*], it implies that $x_i = y_j$ is an element of the LCS. The elements of the LCS are encountered in reverse order by this method. The following recursive procedure prints out an LCS of *X* and *Y* in the proper, forward order. The initial invocation is PRINT-LCS(*b, X*, *length*[*X*], *length*[*Y*]).



|  | j | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|
| i |  | $y_j$ | B | D | C | A | B | A |
| 0 | $x_i$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A | 0 | ↑ 0 | ↑ 0 | ↑ 0 | ↖ 1 | ← 1 | ↖ 1 |
| 2 | B | 0 | ↖ 1 | ← 1 | ← 1 | ↑ 1 | ↖ 2 | ← 2 |
| 3 | C | 0 | ↑ 1 | ↑ 1 | ↖ 2 | ← 2 | ↑ 2 | ↑ 2 |
| 4 | B | 0 | ↖ 1 | ↑ 1 | ↑ 2 | ↑ 2 | ↖ 3 | ← 3 |
| 5 | D | 0 | ↑ 1 | ↖ 2 | ↑ 2 | ↑ 2 | ↑ 3 | ↑ 3 |
| 6 | A | 0 | ↑ 1 | ↑ 2 | ↑ 2 | ↖ 3 | ↑ 3 | ↖ 4 |
| 7 | B | 0 | ↖ 1 | ↑ 2 | ↑ 2 | ↑ 3 | ↖ 4 | ↑ 4 |

***To reconstruct the elements of an LCS, follow the b[i, j] arrows from the lower right-hand corner; the path is shaded. Each "↖ on the path corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.***

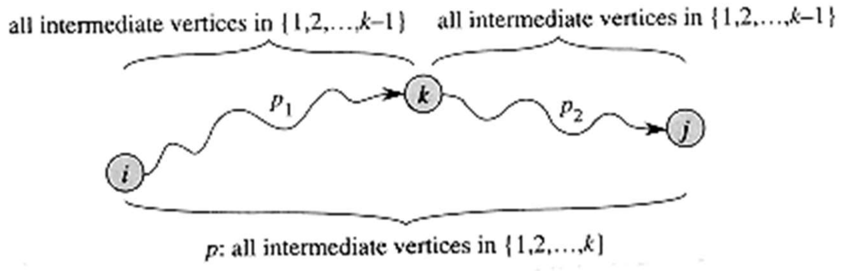
```
PRINT-LCS(b,X,i,j)
if i = 0 or j = 0
    then return
if b[i, j] = "↖"
    then PRINT-LCS(b,X,i - 1, j - 1)
         print x_i
elseif b[i,j] = "↑"
```

```
    then PRINT-LCS($b,X,i$ - 1,$j$)
else PRINT-LCS($b,X,i, j$ - 1)
```

For the *b* table this procedure prints "*BCBA*" The procedure takes time *O*(*m* + *n*), since at least one of *i* and *j* is decremented in each stage of the recursion.

**The Floyd-Warshall algorithm**

The Floyd-Warshall algorithm is based on the following observation. Let the vertices of *G* be *V* = {1, 2, . . . , *n*}, and consider a subset {1, 2, . . . , *k*} of vertices for some *k*. For any pair of vertices *i, j* ∈ *V*, consider all paths from *i* to *j* whose intermediate vertices are all drawn from {1, 2, . . . , *k*}, and let *p* be a minimum-weight path from among them. (Path *p* is simple, since we assume that *G* contains no negative-weight cycles.) The Floyd- Warshall algorithm exploits a relationship between path *p* and shortest paths from *i* to *j* with all intermediate vertices in the set {1, 2, . . . , *k* - 1}. The relationship depends on whether or not *k* is an intermediate vertex of path *p*.


all intermediate vertices in {1,2,…,k−1}
all intermediate vertices in {1,2,…,k−1}
p1
k
p2
i
j
p: all intermediate vertices in {1,2,…,k}

- If *k* is not an intermediate vertex of path *p*, then all intermediate vertices of path *p* are in the set {1, 2, . . . , *k* - 1}. Thus, a shortest path from vertex *i* to vertex *j* with all intermediate vertices in the set {1, 2, . . . , *k* - 1} is also a shortest path from *i* to *j* with all intermediate vertices in the set {1, 2, . . . , *k*}.

    - If *k* is an intermediate vertex of path *p*, then we break *p* down into $i \overset{p_1}{\leadsto} k \overset{p_2}{\leadsto} j$ as shown in Figure 26.3. By Lemma 25.1, $p_1$ is a shortest path from *i* to *k* with all intermediate vertices in the set {1, 2, . . . , *k*}. In fact, vertex *k* is not an intermediate vertex of path $p_1$, and so $p_1$ is a shortest path from *i* to *k* with all intermediate vertices in the set {1, 2, . . . , *k* - 1}. Similarly, $p_2$ is a shortest path from vertex *k* to vertex *j* with all intermediate vertices in the set {1, 2, . . . , *k* - 1}.

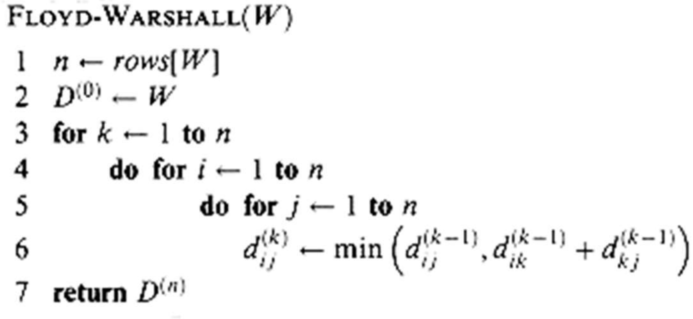
A recursive solution to the all-pairs shortest-paths problem

Based on the above observations, we define a different recursive formulation of shortest-path estimates than we did in Section 26.1. Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex *i* to vertex *j* with all intermediate vertices in the set {1, 2, . . . , *k*}. When *k* = 0, a path from vertex *i* to vertex *j* with no intermediate vertex numbered higher than 0 has no intermediate vertices at all. It thus has at most one edge, and hence $d_{ij}^{(0)} = w_{ij}$. A recursive definition is given by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 , \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 . \end{cases} \qquad (26.5)$$

The matrix $D^{(n)} = \left(d_{ij}^{(n)}\right)$ gives the final answer--$d_{ij}^{(n)} = \delta(i, j)$ for all *i, j* ∈ *V*--because all intermediate vertices are in the set {1, 2, . . . , *n*}.

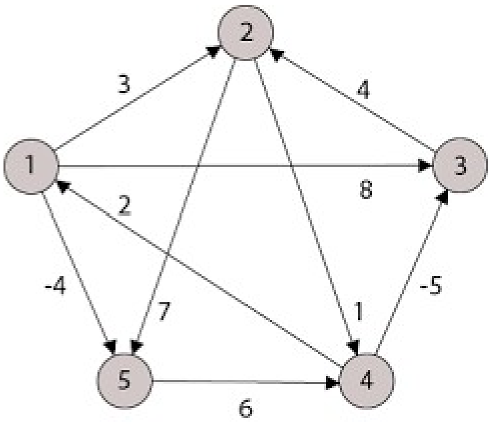
The following bottom-up procedure can be used to compute the values $d_{ij}^{(k)}$ in order of increasing values of *k*. Its input is an *n* × *n* matrix *W* . The procedure returns the matrix $D^{(n)}$ of shortest-path weights.



```
FLOYD-WARSHALL($W$)
1  $n \leftarrow rows[W]$
2  $D^{(0)} \leftarrow W$
3  for $k \leftarrow 1$ to $n$
4       do for $i \leftarrow 1$ to $n$
5               do for $j \leftarrow 1$ to $n$
6                       $d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
7  return $D^{(n)}$
```

The running time of the Floyd-Warshall algorithm is determined by the triply nested **for** loops of lines 3-6. Each execution of line 6 takes *O*(1) time. The algorithm thus runs in time $\Theta(n^3)$.

Example



$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(0)} = \begin{pmatrix} \mathrm{NIL} & 1 & 1 & \mathrm{NIL} & 1 \\ \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} & 2 & 2 \\ \mathrm{NIL} & 3 & \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} \\ 4 & \mathrm{NIL} & 4 & \mathrm{NIL} & \mathrm{NIL} \\ \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} & 5 & \mathrm{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(1)} = \begin{pmatrix} \mathrm{NIL} & 1 & 1 & \mathrm{NIL} & 1 \\ \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} & 2 & 2 \\ \mathrm{NIL} & 3 & \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} \\ 4 & 1 & 4 & \mathrm{NIL} & 1 \\ \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} & 5 & \mathrm{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(2)} = \begin{pmatrix} \mathrm{NIL} & 1 & 1 & 2 & 1 \\ \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} & 2 & 2 \\ \mathrm{NIL} & 3 & \mathrm{NIL} & 2 & 2 \\ 4 & 1 & 4 & \mathrm{NIL} & 1 \\ \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} & 5 & \mathrm{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(3)} = \begin{pmatrix} \mathrm{NIL} & 1 & 1 & 2 & 1 \\ \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} & 2 & 2 \\ \mathrm{NIL} & 3 & \mathrm{NIL} & 2 & 2 \\ 4 & 3 & 4 & \mathrm{NIL} & 1 \\ \mathrm{NIL} & \mathrm{NIL} & \mathrm{NIL} & 5 & \mathrm{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(4)} = \begin{pmatrix} \mathrm{NIL} & 1 & 4 & 2 & 1 \\ 4 & \mathrm{NIL} & 4 & 2 & 1 \\ 4 & 3 & \mathrm{NIL} & 2 & 1 \\ 4 & 3 & 4 & \mathrm{NIL} & 1 \\ 4 & 3 & 4 & 5 & \mathrm{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(5)} = \begin{pmatrix} \mathrm{NIL} & 3 & 4 & 5 & 1 \\ 4 & \mathrm{NIL} & 4 & 2 & 1 \\ 4 & 3 & \mathrm{NIL} & 2 & 1 \\ 4 & 3 & 4 & \mathrm{NIL} & 1 \\ 4 & 3 & 4 & 5 & \mathrm{NIL} \end{pmatrix}$$

Computation of π Matrix

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty , \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty . \end{cases} \qquad (26.6)$$

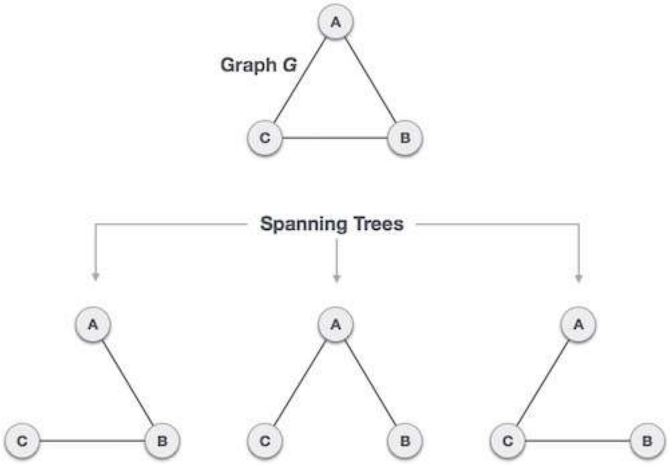
For $k \geq 1$, if we take the path $i \leadsto k \leadsto j$, then the predecessor of *j* we choose is the same as the predecessor of *j* we chose on a shortest path from *k* with all intermediate vertices in the set {1, 2, . . . , *k* - 1}. Otherwise, we choose the same predecessor of *j* that we chose on a shortest path from *i* with all intermediate vertices in the set {1, 2, . . . , *k* - 1}. Formally, for $k \geq 1$,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} , \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} . \end{cases} \qquad (26.7)$$

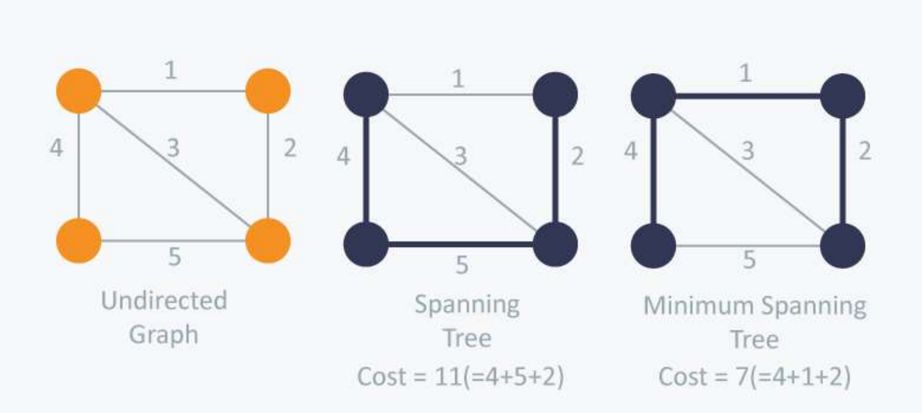
**Minimum Spanning Tree**

**Spanning tree**

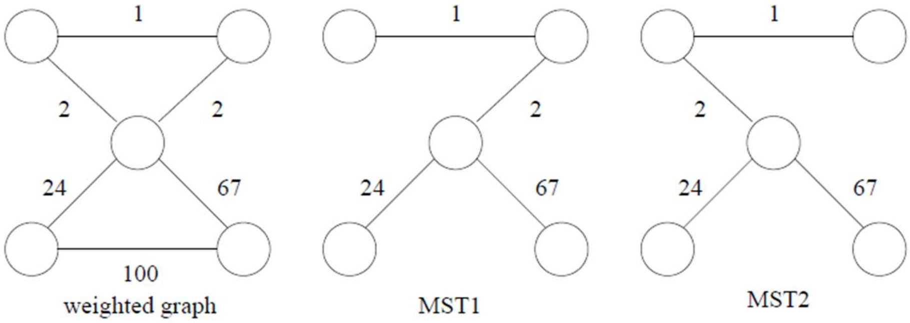
A **spanning tree** is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a **spanning tree** does not have cycles and it cannot be disconnected.. By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one **spanning tree**.



A **Minimum Spanning Tree** in an undirected connected weighted graph is a spanning tree of minimum weight (among all spanning trees).



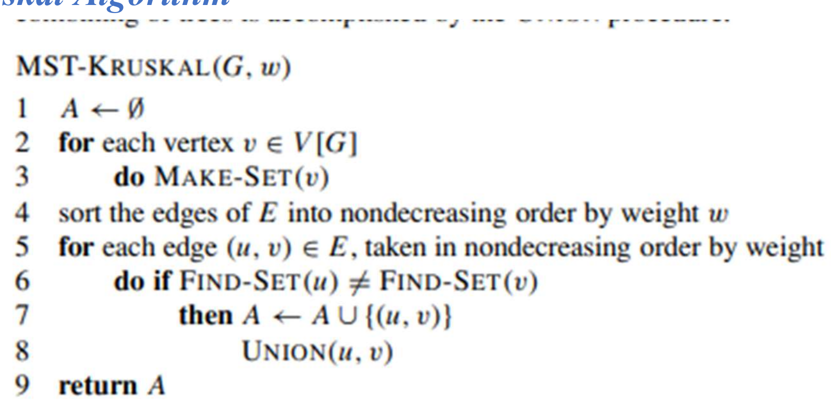
**Remark:** The minimum spanning tree may not be unique. However, if the weights of all the edges are pairwise distinct, it is indeed unique



Two algorithm by which we can find minimum spanning tree

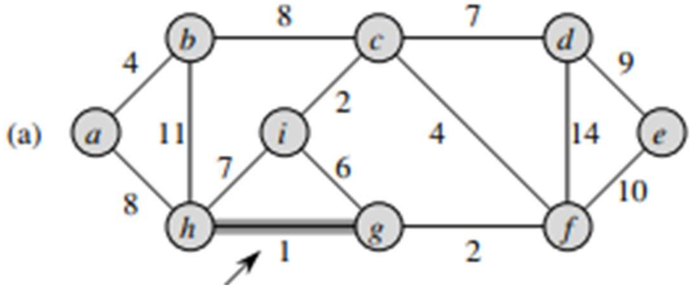
1. KRUSKAL’S
2. PRIM’S

## Kruskal Algorithm



```
MST-KRUSKAL(G, w)
A ← Ø
for each vertex $v \in V[G]$
    do MAKE-SET($v$)
sort the edges of $E$ into nondecreasing order by weight $w$
for each edge $(u, v) \in E$, taken in nondecreasing order by weight
    do if FIND-SET($u$) $\neq$ FIND-SET($v$)
        then $A \leftarrow A \cup \{(u, v)\}$
             UNION($u, v$)
return $A$
```

Example



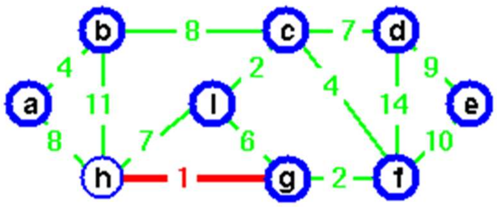
| EDGES SET E | WEIGHT | FIND _SET (U) | FIND_SET (V) | MAKE_SET(V) | ACTION | SET A | Selected weight |
|---|---|---|---|---|---|---|---|
| | | | | **{a}{b}{c}{d}{e}{f}{g}{h} {i}** | | **SET A= { }** | |
| (g,h) | 1 | {g} | {h} | {g,h} | SELECTED | A=AU (g,h) | 1 |
| (f,g) | 2 | | | | | | |
| (c,i) | 2 | | | | | | |
| (a,b) | 4 | | | | | | |
| (c,f) | 4 | | | | | | |
| (I,g) | 6 | | | | | | |
| (c,d) | 7 | | | | | | |
| (h,i) | 7 | | | | | | |
| (a,h) | 8 | | | | | | |
| (b,c) | 8 | | | | | | |
| (d,e) | 9 | | | | | | |
| (e,f) | 10 | | | | | | |
| (b,h) | 11 | | | | | | |
| (d,f) | 14 | | | | | | |

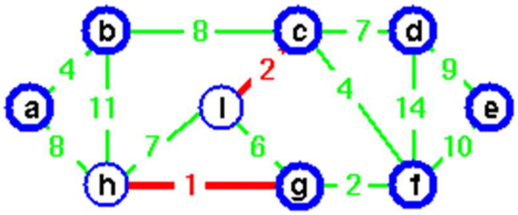
| EDGES SET E | WEIGHT | FIND_SET (U) | FIND_SET (V) | UNION (U,V) | ACTION | SET A | Selected weight |
|---|---|---|---|---|---|---|---|

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |
| | | | | | | SET A= { } | | |
| (g,h) | 1 | {g} | {h} | {g,h} | SELECTED | A=AU (g,h) | 1 | |
| (f,g) | 2 | {f} | {g,h} | {f,g,h} | SELECTED | a=AU (f,g,) | 2 | |
| (c,i) | 2 | {c} | {i} | {c,i} | SELECTED | A=A U (c,i) | 2 | |
| (a,b) | 4 | {a} | {b} | {a,b} | SELECTED | A=AU(a,b) | 4 | |
| (c,f) | 4 | {c,i} | {f,g,h} | {c,f,g,h,i} | SELECTED | A=AU(c,f) | 4 | |
| (I,g) | 6 | {c,f,g,h,i} | {c,f,g,h,i} | | unselected | | | |
| (c,d) | 7 | {c,f,g,h,i} | {d} | {c,d,f,g,h,i} | SELECTED | A=AU(c,d) | 7 | |
| (h,i) | 7 | {c,d,f,g,h,i} | {c,d,f,g,h,i} | | unselected | | | |
| (a,h) | 8 | {a,b} | {c,d,f,g,h,i} | {a,b,c,d,f,g,h,i} | SELECTED | A=AU(a,h) | 8 | |
| (b,c) | 8 | {a,b,c,d,f,g,h,i} | {a,b,c,d,f,g,h,i} | | unselected | | | |
| (d,e) | 9 | {a,b,c,d,f,g,h,i} | {e} | {a,b,c,d,e,f,g,h,i } | SELECTED | A=AU(d,e) | 9 | |
| (e,f) | 10 | | | | | | | |
| (b,h) | 11 | | | | | | | |
| (d,f) | 14 | | | | | | | |
| | | | | | | **Total** | **37** | |

Step 1. In the graph, the Edge(*g*, *h*) is shortest. Either vertex *g* or vertex *h* could be representative. Lets choose vertex g arbitrarily.

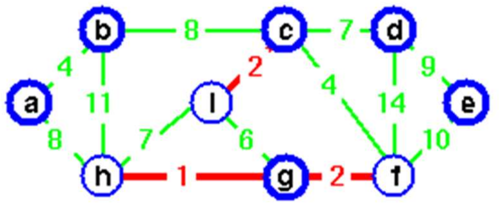
Step 2. The edge (*c*, *i*) creates the second tree. Choose vertex *c* as representative for second tree.

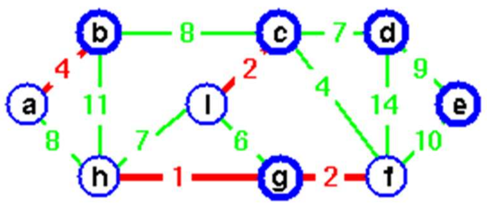




Step 3. Edge (g, g) is the next shortest edge. Add this edge and choose vertex g as representative.

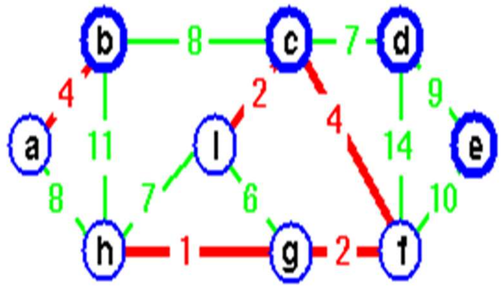
Step 4. Edge (a, b) creates a third tree.

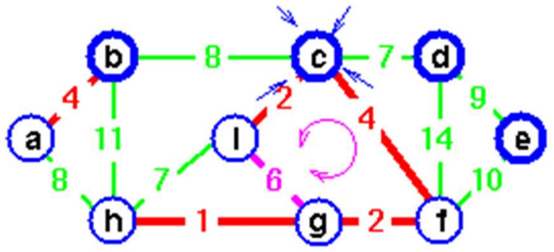




Step 5. Add edge (c, f) and merge two trees. Vertex c is chosen as the representative.

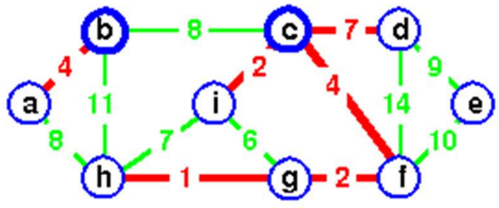
Step 6. Edge (g, i) is the next next cheapest, but if we add this edge a cycle would be created. Vertex c is the representative of both.

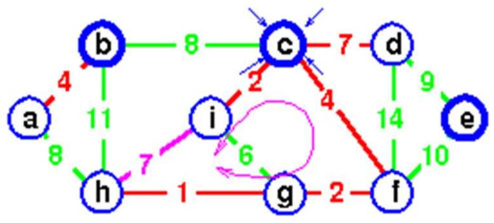




Step 7. Instead, add edge (c, d).

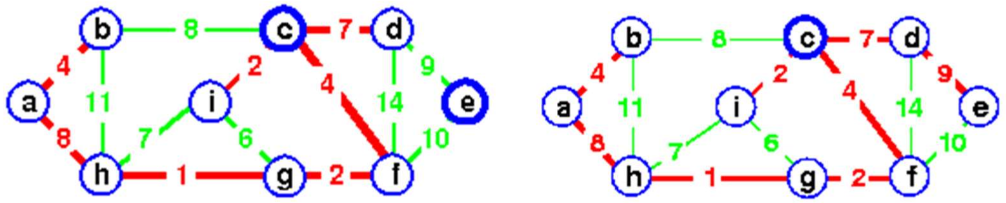
Step 8. If we add edge (h, i), edge(h, i) would make a cycle.





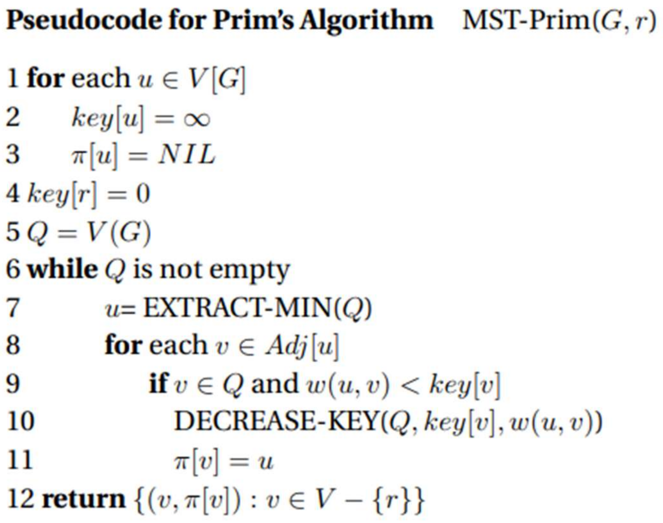
Step 9. Instead of adding edge (h, i) add edge (a, h).

Step 10. Again, if we add edge (b, c), it would create a cycle. Add edge (d, e) instead to complete the spanning tree. In this spanning tree all trees joined and vertex c is a sole representative.



### *Prim's Algorithm*

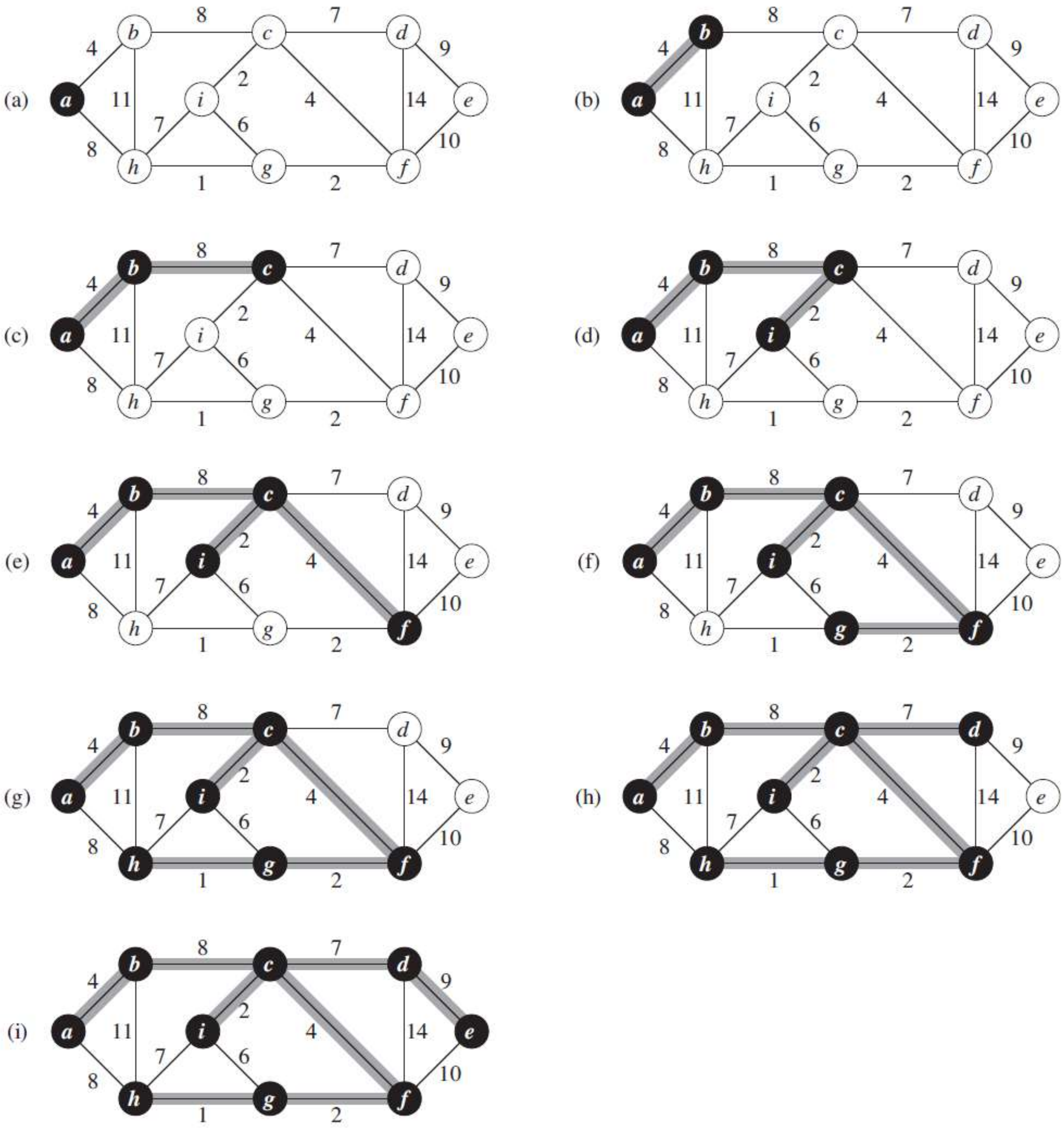
Prim's algorithm is very similar to Kruskal's: whereas Kruskal's "grows" a forest of trees, Prim's algorithm grows a single tree until it becomes the minimum spanning tree. Both algorithms use the greedy approach - they add the cheapest edge that will not cause a cycle. But rather than choosing the cheapest edge that will connect *any* pair of trees together, Prim's algorithm only adds edges that join nodes to the existing tree



**Pseudocode for Prim's Algorithm** MST-Prim$(G, r)$

```
for each $u \in V[G]$
    $key[u] = \infty$
    $\pi[u] = NIL$
$key[r] = 0$
$Q = V(G)$
while $Q$ is not empty
        $u$= EXTRACT-MIN$(Q)$
        for each $v \in Adj[u]$
            if $v \in Q$ and $w(u, v) < key[v]$
                DECREASE-KEY$(Q, key[v], w(u, v))$
                $\pi[v] = u$
return $\{(v, \pi[v]) : v \in V - \{r\}\}$
```

The time complexity is *O(V*log*V + E*log*V) = O(E*log*V)*, making it the same as Kruskal's algorithm. However, Prim's algorithm can be improved using Fibonacci Heaps to *O(E +* log*V).*


(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
8
10
1
2