# Methods of Minimum Spanning Tree

There are two methods to find Minimum Spanning Tree

1. Kruskal's Algorithm
2. Prim's Algorithm

---

## Kruskal's Algorithm:

An algorithm to construct a Minimum Spanning Tree for a connected weighted graph. It is a Greedy Algorithm. The Greedy Choice is to put the smallest weight edge that does not because a cycle in the MST constructed so far.

**If the graph is not linked, then it finds a Minimum Spanning Tree.**

**Steps for finding MST using Kruskal's Algorithm:**

1. Arrange the edge of G in order of increasing weight.
2. Starting only with the vertices of G and proceeding sequentially add each edge which does not result in a cycle, until (n - 1) edges are used.
3. EXIT.

```
MST- KRUSKAL (G, w)
 1. A ← ∅
 2. for each vertex v ∈ V [G]
 3. do MAKE - SET (v)
 4. sort the edges of E into non decreasing order by weight w
 5. for each edge (u, v) ∈ E, taken in non decreasing order by weight
 6. do if FIND-SET (μ) ≠ if FIND-SET (v)
 7. then A  ←  A ∪ {(u, v)}
 8. UNION (u, v)
 9. return A
```

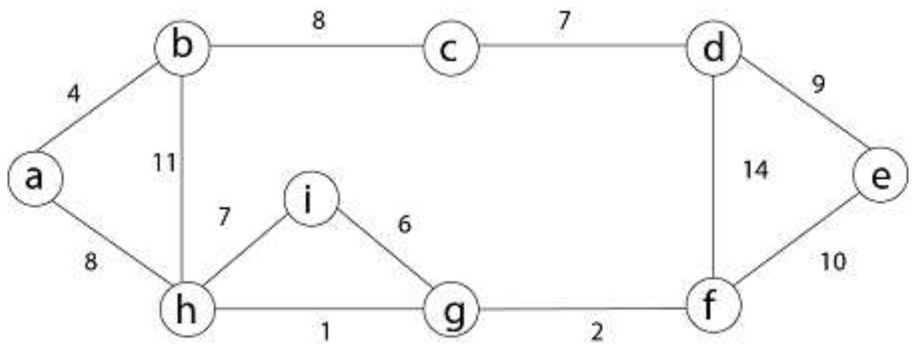
**Analysis:** Where E is the number of edges in the graph and V is the number of vertices, Kruskal's Algorithm can be shown to run in O (E log E) time, or simply, O (E log V) time, all with simple data structures. These running times are equivalent because:

- E is at most $V^2$ and log $V^2$= 2 x log V is O (log V).
- If we ignore isolated vertices, which will each their components of the minimum spanning tree, V ≤ 2 E, so log V is O (log E).

Thus the total time is

1. O (E log E) = O (E log V).

**For Example:** Find the Minimum Spanning Tree of the following graph using Kruskal's algorithm.



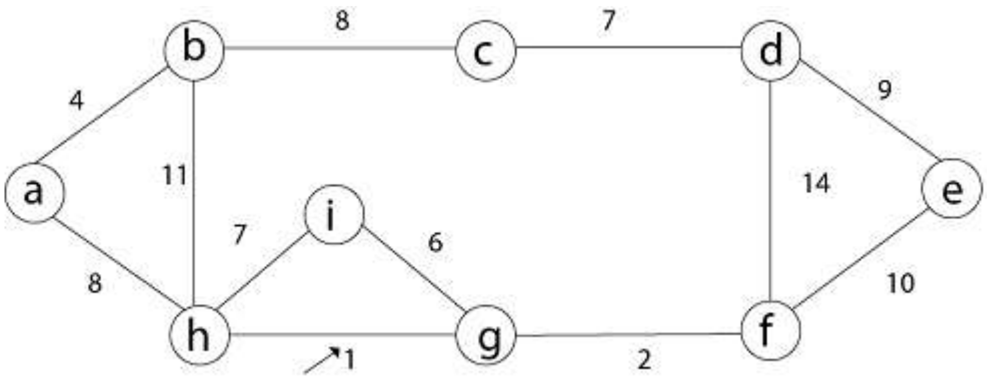
**Solution:** First we initialize the set A to the empty set and create |v| trees, one containing each vertex with MAKE-SET procedure. Then sort the edges in E into order by non-decreasing weight.

There are 9 vertices and 12 edges. So MST formed (9-1) = 8 edges

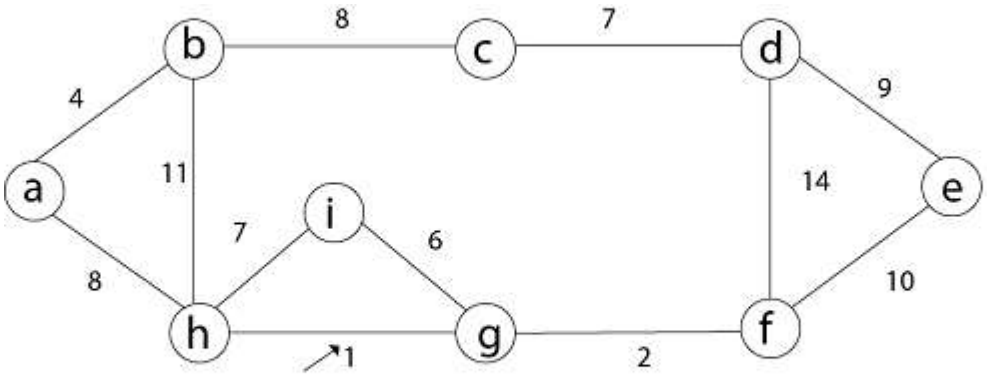
| Weight | Source | Destination |
|---|---|---|
| 1 | h | g |
| 2 | g | f |
| 4 | a | b |
| 6 | i | g |
| 7 | h | i |
| 7 | c | d |
| 8 | b | c |
| 8 | a | h |
| 9 | d | e |
| 10 | e | f |
| 11 | b | h |
| 14 | d | f |

Now, check for each edge (u, v) whether the endpoints u and v belong to the same tree. If they do then the edge (u, v) cannot be supplementary. Otherwise, the two vertices belong to different trees, and the edge (u, v) is added to A, and the vertices in two trees are merged in by union procedure.

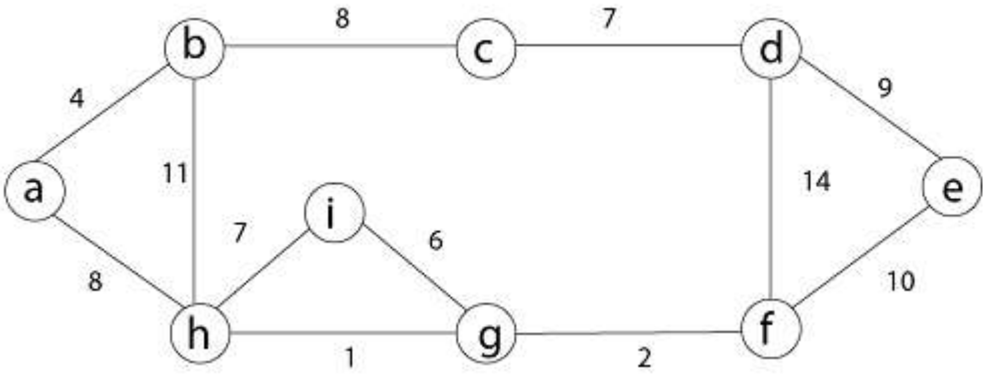
**Step1:** So, first take (h, g) edge



**Step 2:** then (g, f) edge.

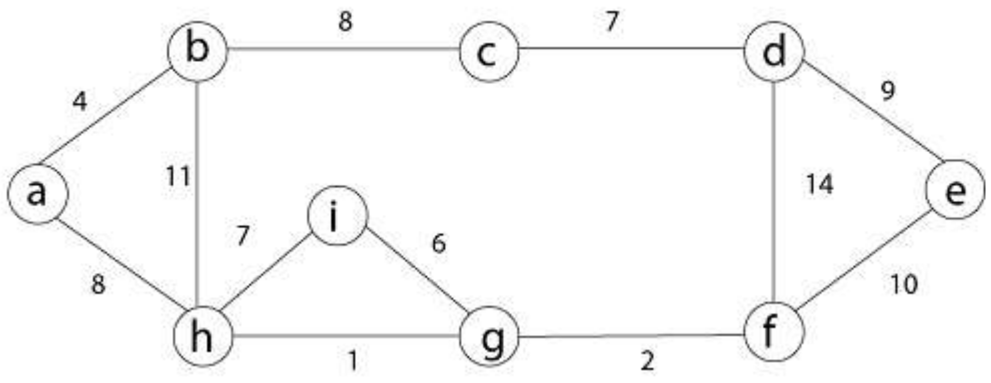


**Step 3:** then (a, b) and (i, g) edges are considered, and the forest becomes



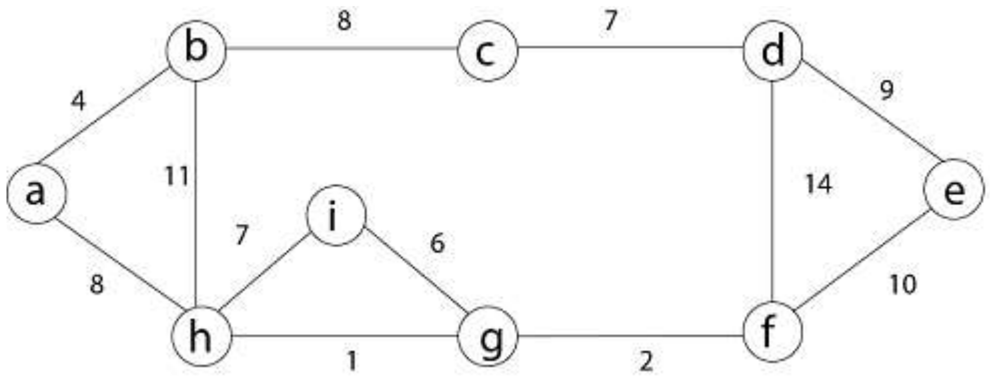
**Step 4:** Now, edge (h, i). Both h and i vertices are in the same set. Thus it creates a cycle. So this edge is discarded.

Then edge (c, d), (b, c), (a, h), (d, e), (e, f) are considered, and the forest becomes.



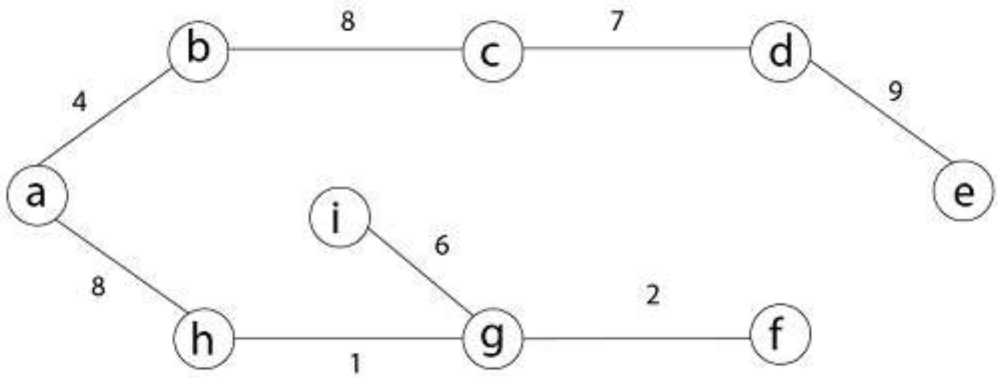
**Step 5:** In (e, f) edge both endpoints e and f exist in the same tree so discarded this edge. Then (b, h) edge, it also creates a cycle.

**Step 6:** After that edge (d, f) and the final spanning tree is shown as in dark lines.



**Step 7:** This step will be required Minimum Spanning Tree because it contains all the 9 vertices and (9 - 1) = 8 edges

1. e → f, b → h, d → f [cycle will be formed]



Minimum Cost MST

# Prim's Algorithm

It is a greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices:

- Contain vertices already included in MST.
- Contain vertices not yet included.

At every step, it considers all the edges and picks the minimum weight edge. After picking the edge, it moves the other endpoint of edge to set containing MST.

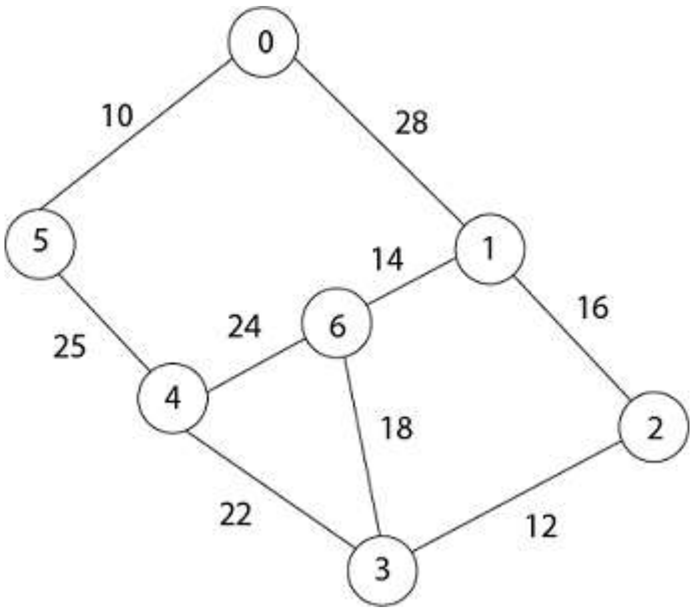
## Steps for finding MST using Prim's Algorithm:

1. Create MST set that keeps track of vertices already included in MST.
2. Assign key values to all vertices in the input graph. Initialize all key values as INFINITE (∞). Assign key values like 0 for the first vertex so that it is picked first.
3. While MST set doesn't include all vertices.
   a. Pick vertex u which is not is MST set and has minimum key value. Include 'u'to MST set.
   b. Update the key value of all adjacent vertices of u. To update, iterate through all adjacent vertices. For every adjacent vertex v, if the weight of edge u.v less than the previous key value of v, update key value as a weight of u.v.

**MST-PRIM (G, w, r)**

```
1. for each u ∈ V [G]
2. do key [u] ← ∞
3. π [u] ← NIL
4. key [r] ← 0
5. Q ← V [G]
6. While Q ? Ø
7. do u ← EXTRACT - MIN (Q)
8. for each v ∈ Adj [u]
9. do if v ∈ Q and w (u, v) < key [v]
10. then π [v] ← u
11. key [v] ← w (u, v)
```

**Example:** Generate minimum cost spanning tree for the following graph using Prim's algorithm.



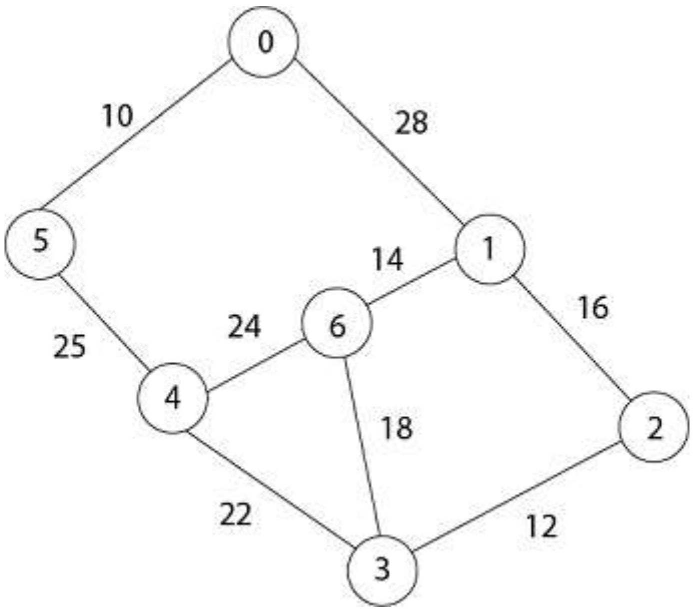
**Solution:** In Prim's algorithm, first we initialize the priority Queue Q. to contain all the vertices and the key of each vertex to ∞ except for the root, whose key is set to 0. Suppose 0 vertex is the root, i.e., r. By EXTRACT - MIN (Q) procure, now u = r and Adj [u] = {5, 1}.

Removing u from set Q and adds it to set V - Q of vertices in the tree. Now, update the key and π fields of every vertex v adjacent to u but not in a tree.

| Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| Key Value | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| Parent | NIL | NIL | NIL | NIL | NIL | NIL | NIL |

1. Taking 0 as starting vertex
2. Root = 0
3. Adj [0] = 5, 1
4. Parent, π [5] = 0 and π [1] = 0
5. Key [5] = ∞ and key [1] = ∞
6. w [0, 5) = 10 and w (0,1) = 28
7. w (u, v) < key [5] , w (u, v) < key [1]
8. Key [5] = 10 and key [1] = 28
9. So update key value of 5 and 1 is:

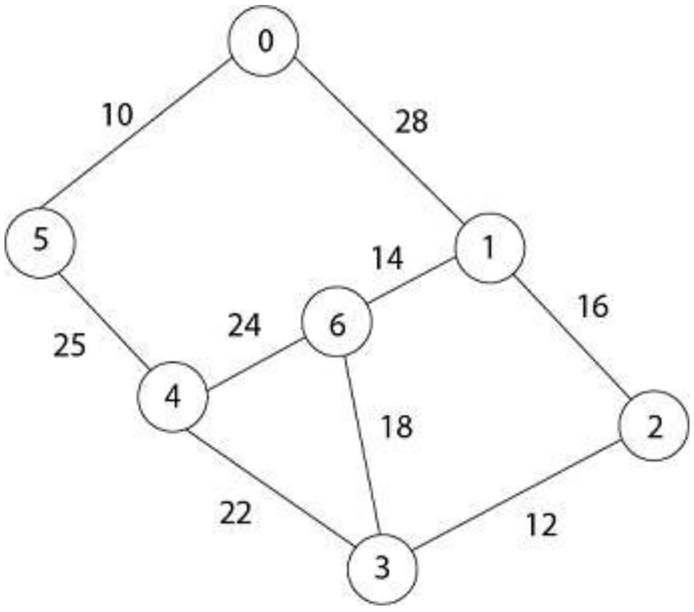
| Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| Key Value | 0 | 28 | ∞ | ∞ | ∞ | 10 | ∞ |
| Parent | NIL | 0 | NIL | NIL | NIL | 0 | NIL |



Now by EXTRACT_MIN (Q) Removes 5 because key [5] = 10 which is minimum so u = 5.

1. Adj [5] = {0, 4} and 0 is already in heap
2. Taking 4, key [4] = ∞      π [4] = 5
3. (u, v) < key [v] then key [4] = 25
4. w (5,4) = 25
5. w (5,4) < key [4]
6. date key value and parent of 4.

| Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| Key Value | 0 | 28 | ∞ | ∞ | 25 | 10 | ∞ |
| Parent | NIL | 0 | NIL | NIL | 5 | 0 | NIL |



Now remove 4 because key [4] = 25 which is minimum, so u =4

1. Adj [4] = {6, 3}
2. Key [3] = ∞ key [6] = ∞
3. w (4,3) = 22 w (4,6) = 24
4. w (u, v) < key [v] w (u, v) < key [v]
5. w (4,3) < key [3] w (4,6) < key [6]

Update key value of key [3] as 22 and key [6] as 24.

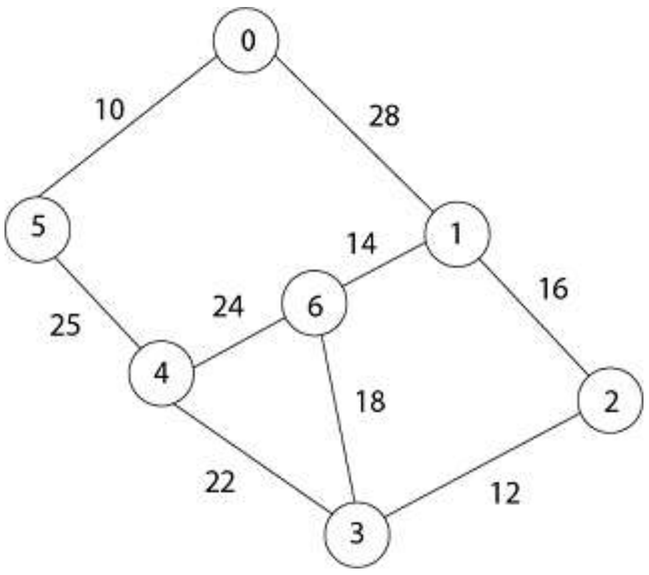
And the parent of 3, 6 as 4.

1. п[3]= 4 п[6]= 4

| Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| Key Value | 0 | 28 | ∞ | 22 | 25 | 10 | 24 |
| Parent | NIL | 0 | NIL | 4 | 5 | 0 | 4 |

1. u = EXTRACT_MIN (3, 6) [key [3] < key [6]]
2. u = 3 i.e. 22 < 24

Now remove 3 because key [3] = 22 is minimum so u =3.



1. Adj [3] = {4, 6, 2}
2. 4 is already in heap
3. 4 ≠ Q key [6] = 24 now becomes key [6] = 18
4. Key [2] = ∞ key [6] = 24
5. w (3, 2) = 12 w (3, 6) = 18
6. w (3, 2) < key [2] w (3, 6) < key [6]

Now in Q, key [2] = 12, key [6] = 18, key [1] = 28 and parent of 2 and 6 is 3.

1. п [2] = 3      п[6]=3

Now by EXTRACT_MIN (Q) Removes 2, because key [2] = 12 is minimum.

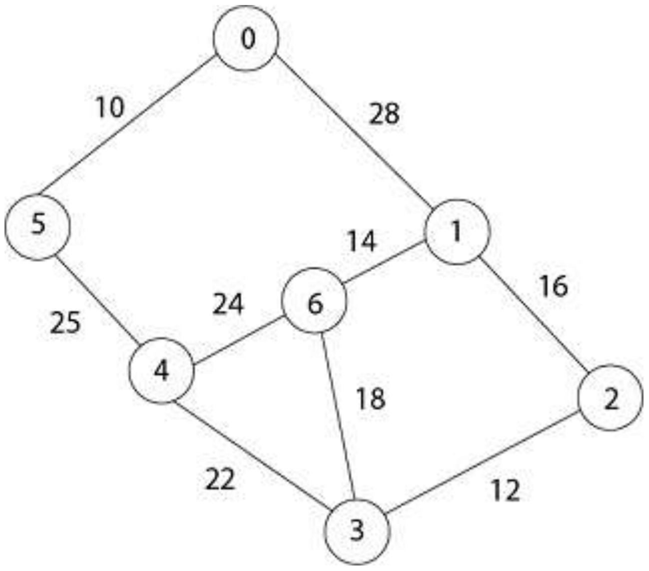
| Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| Key Value | 0 | 28 | 12 | 22 | 25 | 10 | 18 |
| Parent | NIL | 0 | 3 | 4 | 5 | 0 | 3 |

1. u = EXTRACT_MIN (2, 6)
2. u = 2          [key [2] < key [6]]
3. 12 < 18
4. Now the root is 2
5. Adj [2] = {3, 1}
6. 3 is already in a heap
7. Taking 1, key [1] = 28
8. w (2,1) = 16
9. w (2,1) < key [1]

So update key value of key [1] as 16 and its parent as 2.

1. п[1]= 2

| Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| Key Value | 0 | 16 | 12 | 22 | 25 | 10 | 18 |
| Parent | NIL | 2 | 3 | 4 | 5 | 0 | 3 |



Now by EXTRACT_MIN (Q) Removes 1 because key [1] = 16 is minimum.

1. Adj [1] = {0, 6, 2}

2. 0 and 2 are already in heap.
3. Taking 6, key [6] = 18
4. w [1, 6] = 14
5. w [1, 6] < key [6]

Update key value of 6 as 14 and its parent as 1.

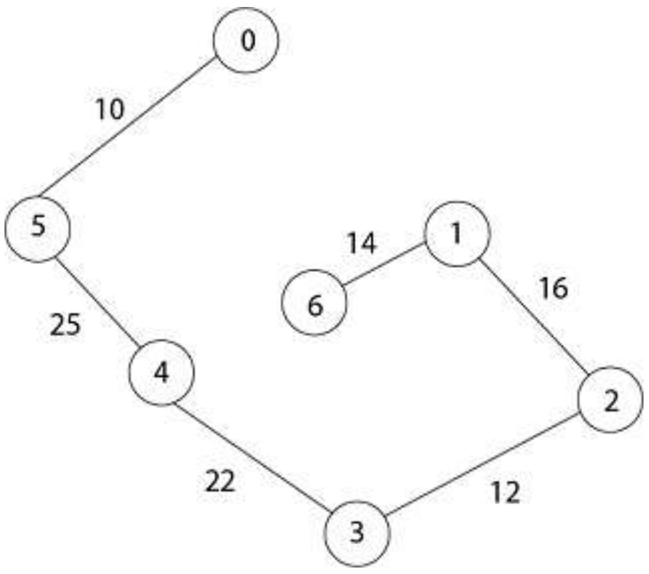
1. Π [6] = 1

| Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| Key Value | 0 | 16 | 12 | 22 | 25 | 10 | 14 |
| Parent | NIL | 2 | 3 | 4 | 5 | 0 | 1 |

Now all the vertices have been spanned, Using above the table we get Minimum Spanning Tree.

1. 0 → 5 → 4 → 3 → 2 → 1 → 6
2. [Because Π [5] = 0, Π [4] = 5, Π [3] = 4, Π [2] = 3, Π [1] =2, Π [6] =1]

**Thus the final spanning Tree is**



**Total Cost = 10 + 25 + 22 + 12 + 16 + 14 = 99**

## Strassen’s Method

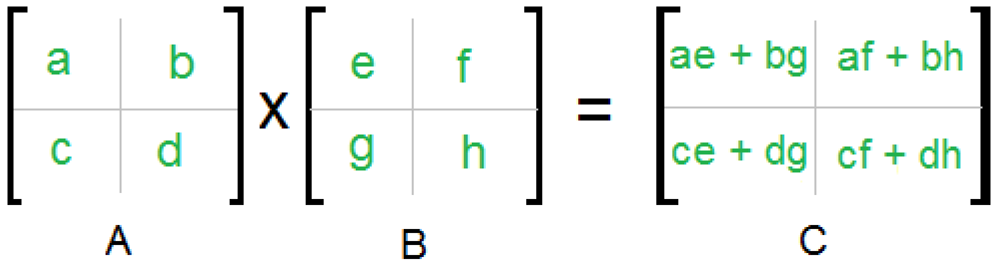
Given two square matrices A and B of size n x n each, find their multiplication matrix.

***Naive 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 $O(N^3)$.

***Divide and Conquer***
Following is simple Divide and Conquer method to multiply two square matrices.
1) Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.
2) Calculate following values recursively. ae + bg, af + bh, ce + dg and cf + dh.



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

In the above method, we do 8 multiplications for matrices of size N/2 x N/2 and 4 additions. Addition of two matrices takes $O(N^2)$ time. So the time complexity can be written as

```
T(N) = 8T(N/2) + O(N²)

From Master's Theorem, time complexity of above method is O(N³)
which is unfortunately same as the above naive 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. Strassen’s method is similar to above simple divide and conquer method in the sense that this method also divide matrices to sub-

matrices of size N/2 x N/2 as shown in the above diagram, but in Strassen’s method, the four sub-matrices of result are calculated using following formulae.

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

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(N²)

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

Generally Strassen’s Method is not preferred for practical applications for following reasons.
1) The constants used in Strassen’s method are high and for a typical application Naive method works better.
2) For Sparse matrices, there are better methods especially designed for them.
3) The submatrices in recursion take extra space.
4) Because of the limited precision of computer arithmetic on noninteger values, larger errors accumulate in Strassen’s algorithm than in Naive Method

## Strassen’s Matrix Equation

Strassen’s matrix is a Divide and Conquer method that helps us to multiply two matrices(of size n X n).

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

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

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

X , Y 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

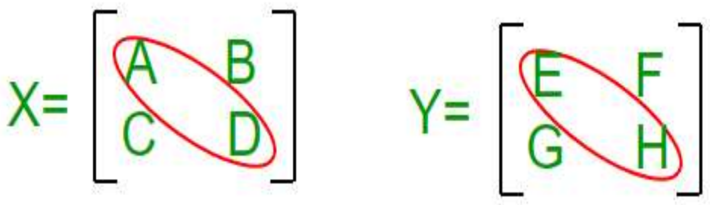
You just need to remember 4 Rules :

- AHED (Learn it as ‘Ahead’)
- Diagonal
- Last CR
- First CR

Also, consider X as (Row +) and Y as (Column -) matrix

Follow the Steps :

- Write P1 = A; P2 = H; P3 = E; P4 = D
- For P5 we will use Diagonal Rule i.e.
  (Sum the Diagonal Elements Of Matrix X ) * (Sum the Diagonal Elements Of Matrix Y ), we get
  P5 = (A + D)* (E + H)

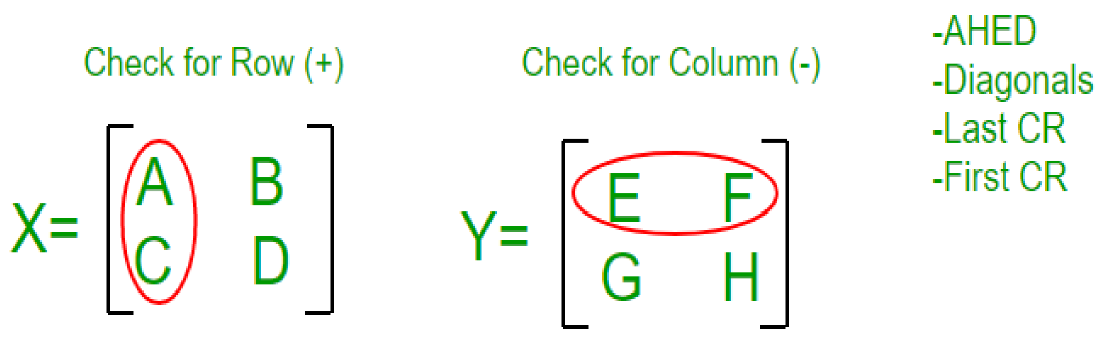


-AHED
-Diagonals
-Last CR
-First CR

P1 = A

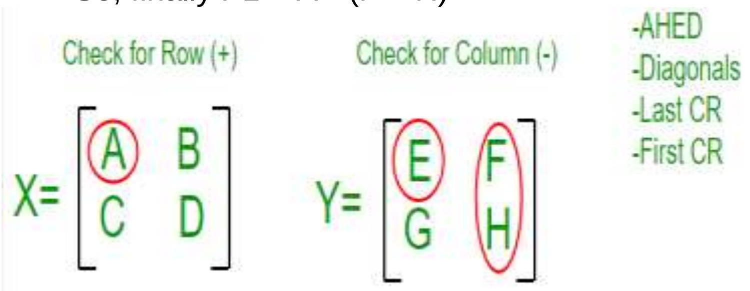
P2= H
P3= E
P4= D
P5= ( A + D ) * ( E + H )

- For P6 we will use Last CR Rule i.e. Last Column of X and Last Row of Y and remember that Row+ and Column- so i.e. (B – D) * (G + H), we get
  P6 = (B – D) * (G + H)
- For P7 we will use First CR Rule i.e. First Column of X and First Row of Y and remember that Row+ and Column- so i.e. (A – C) * (E + F), we get
  P6 = (A – C) * (E + F)



P1 = A
P2= H
P3= E
P4= D
P5= ( A + D ) * ( E + H )
P6= ( B – D ) * ( G + H)
P7= ( A – C ) * ( E + F)

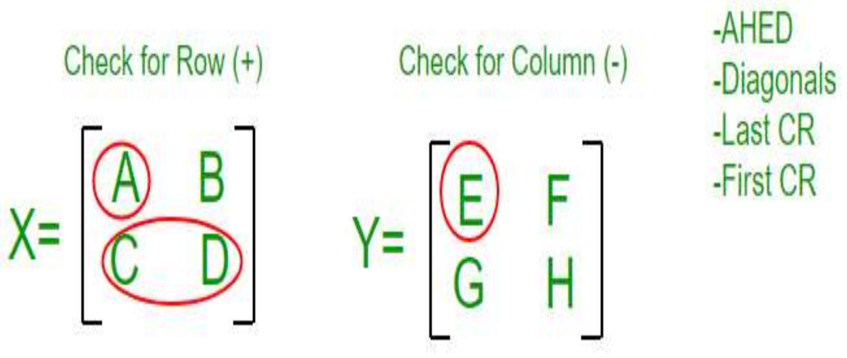
- Come Back to P1 : we have A there and it's adjacent element in Y Matrix is E, since Y is Column Matrix so we select a column in Y such that E won't come, we find F H Column, so multiply A with (F – H)
  So, finally P1 = A * (F – H)



P1 = A * ( F – H)
P2= H
P3= E
P4= D
P5= ( A + D ) * ( E + H )
P6= ( B – D ) * ( G + H)
P7= ( A – C ) * ( E + F)

- Come Back to P2 : we have H there and it's adjacent element in X Matrix is D, since X is Row Matrix so we select a Row in X such that D won't come, we find A B Column, so multiply H with (A + B)
  So, finally P2 = H * (A + B)

- Come Back to P3 : we have E there and it's adjacent element in X Matrix is A, since X is Row Matrix so we select a Row in X such that A won't come, we find C D Column, so multiply E with (C + D)
  So, finally P3 = E * (C + D)



P1= A * ( F – H )
P2= H * ( A + B )
P3= E * ( C + D )
P4= D
P5= ( A + D ) * ( E + H )
P6= ( B – D ) * ( G + H)
P7= ( A – C ) * ( E + F)

- Come Back to P4 : we have D there and it's adjacent element in Y Matrix is H, since Y is Column Matrix so we select a column in Y such that H won't come, we find G E Column, so multiply D with (G – E)
  So, finally P4 = D * (G – E)

We are done with P1 – P7 equations, so now we move to C1 – C4 equations in Final Matrix C :

- Remember Counting : Write P1 + P2 at C2
- Write P3 + P4 at its diagonal Position i.e. at C3
- Write P4 + P5 + P6 at 1st position and subtract P2 i.e. C1 = P4 + P5 + P6 – P2
- Write odd values at last Position with alternating – and + sign i.e. P1 P3 P5 P7 becomes
  C4 = P1 – P3 + P5 – P7

$$XY= \begin{bmatrix} P6 + P5 + P4 - P2 & P1 + P2 \\ P3+ P4 & P1 + P5 - P3 - P7 \end{bmatrix}$$