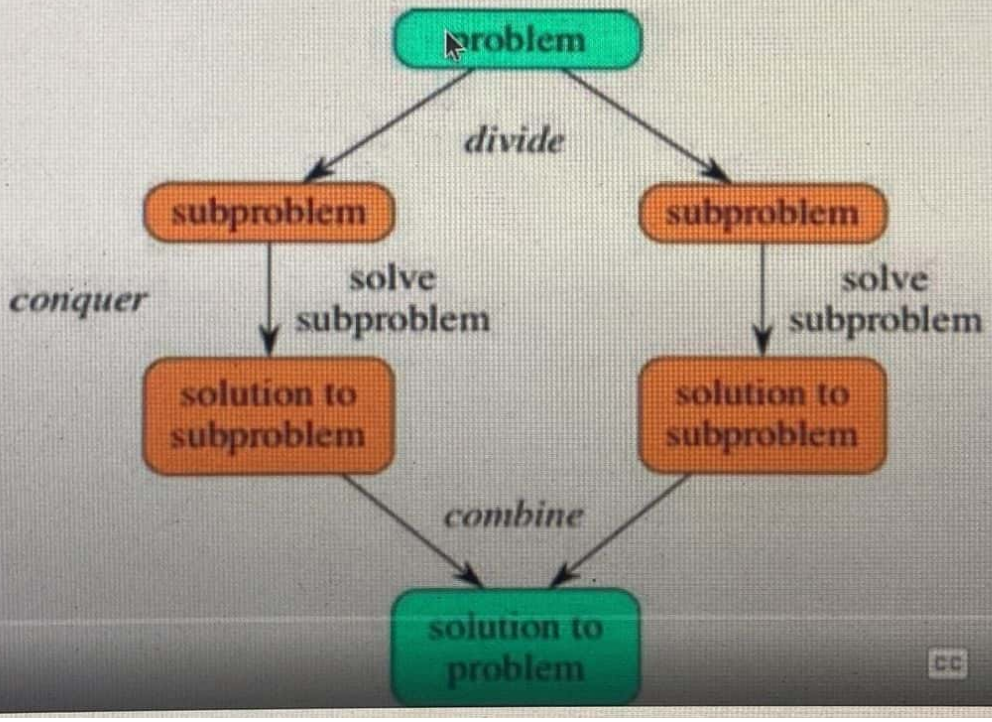
Divide and Conquer Strategy

- **Divide** the problem into a number of sub-problems that are smaller instances of the same problem.
- **Conquer** the sub-problems by solving them recursively. If they are small enough, solve the sub-problems as base cases.
- **Combine** the solutions to the sub-problems into the solution for the original problem.



Control Abstraction

- It is a procedure whose flow of control is clear but whose primary operations are specified by other procedure whose precise meanings are left undefined.

Divide and Conquer Control Abstraction

Algorithm DAndC(P)

```
{
    if Small(P) then
            return S(P)
    else
    {
        Divide P into smaller instances P1, P2, . . . . Pk,  k≥1;
        apply DAndC to each of these sub-problems;
        return Combine(DAndC(P1), DAndC(P2), . .  ,DAndC(Pk));
    }
}
```

Divide and Conquer – Time Complexity

Complexity of many divide and conquer algorithms are given by the following recurrence relation

$$T(n) = \begin{cases} T(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases}$$

2 - Way Merge Sort / Merge Sort

Imp

⇒ is an example of divide and conquer.

⇒ Worst case time complexity is $O(n \log n)$.

Problem

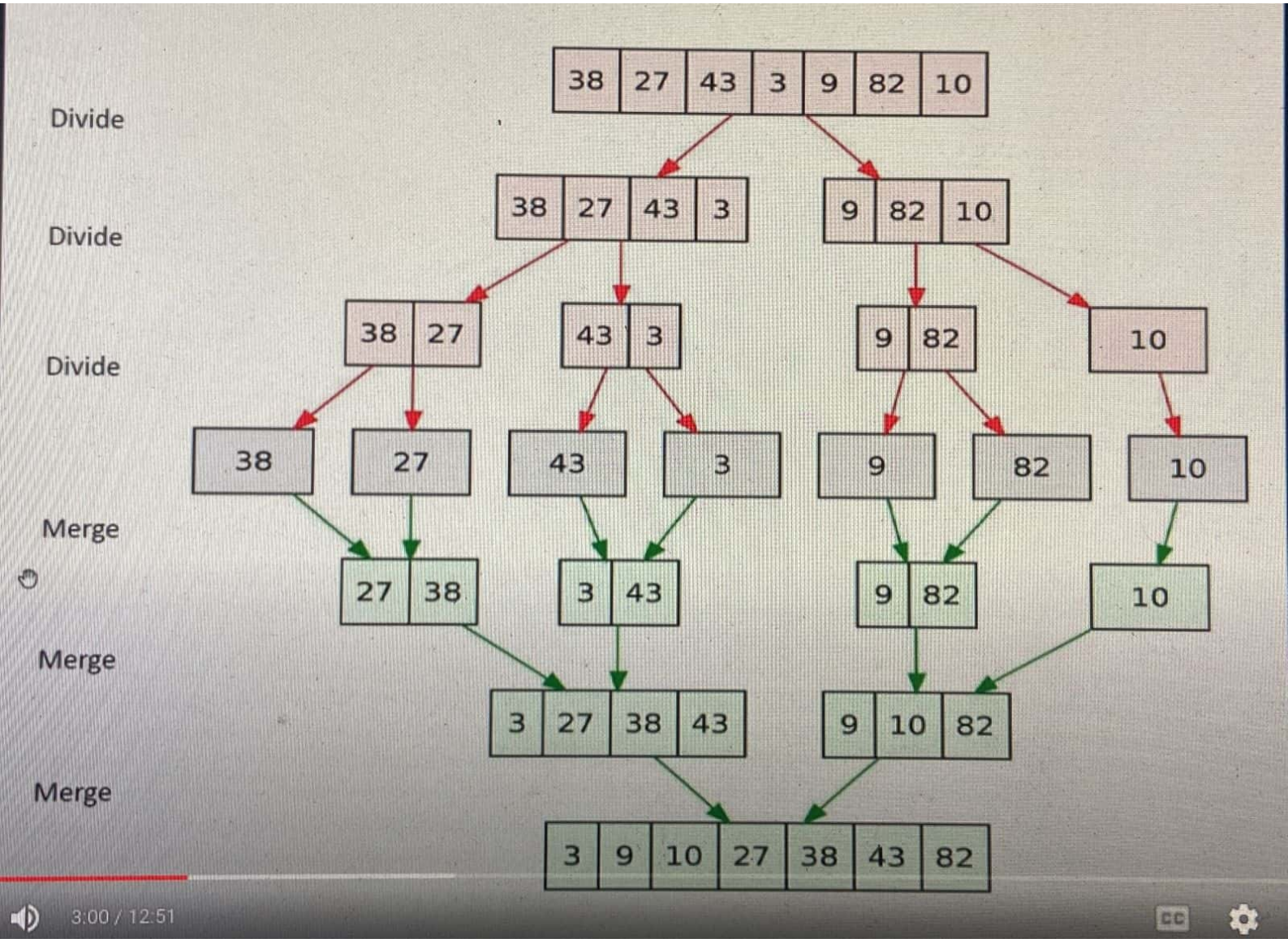
⇒ Given a sequence of n elements a[1] to a[n].

⇒ The general idea is to split the array into 2 sets [ie call 2 way merge sort]

a[1] .... a[n/2] &

a[n/2+1] .... a[n].

⇒ Each set is individually sorted & resulting sorted sequence are merged to



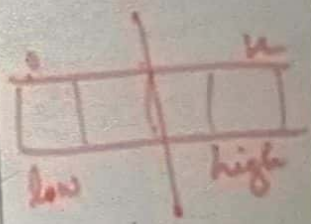
produce a simple sorted sequence of n elements.

Algorithm

```
Algorithm MergeSort (low, high)
{
    if (low < high) then. // if there are more than 1 elements
    {
        mid := (low + high)/2.
        // divide it into subproblems.
        // find where to split the set
```

mergesort()

↳ merge



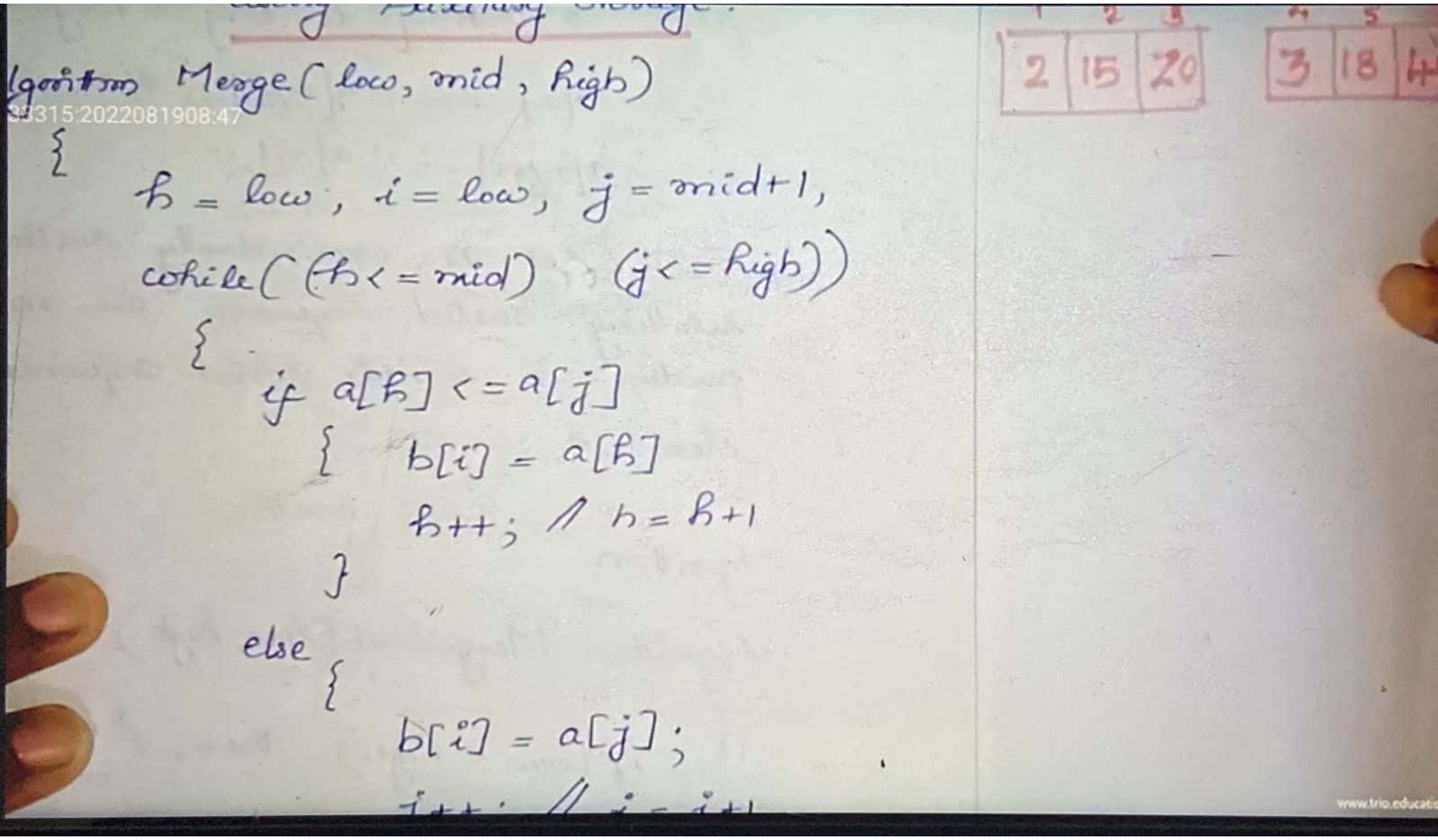
mergesort (low, mid)

mergesort (mid+1, high)

```
        MergeSort(low, mid);
        MergeSort(mid+1, high);
        Merge(low, mid, high); // Combine the
                               Solutions
    }
}
```

Algorithm : Merging 2 Sorted Subarrays using Auxiliary Storage.

```
Algorithm Merge(low, mid, high)
{
```

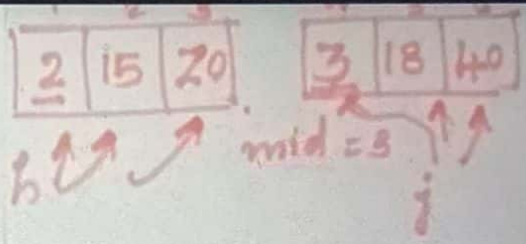


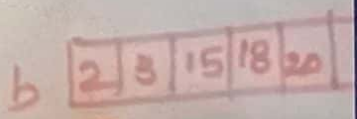
```
Algorithm Merge(low, mid, high)
{
    h = low, i = low, j = mid+1,
    while((h<=mid) (j<=high))
    {
        if a[h] <= a[j]
        { b[i] = a[h]
          h++; // h = h+1
        }
        else
        {
            b[i] = a[j];
            i++; // i = i+1
```

1	2	3
2	15	20

4	5	
3	18	4

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





if (b > mid) then

for k = j to high do

{

b[i] = a[k]

i = i + 1;

}

else

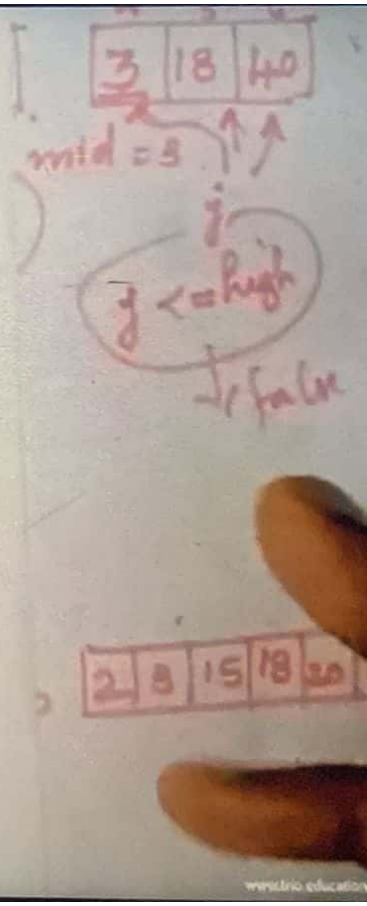
for k = b to mid do

{

b[i] = a[k]

i = i + 1

}



2 Way Merge Sort – Time Complexity

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

a is the time to sort an array of size 1

cn is the time to merge two sub-arrays

2 T(n/2) is the complexity of two recursion calls

2 Way Merge Sort – Time Complexity

$$
\begin{aligned}
T(n) &= 2\,T(n/2) + c\,n \\
&= 2(2\,T(n/4)+c(n/2)) + c\,n \\
&= 2^2T(n/2^2) + 2\,c\,n \\
&= 2^3T(n/2^3) + 3\,c\,n \\
&\ldots\ldots\ldots\ldots\ldots \\
&= 2^kT(n/2^k) + k\,c\,n \qquad [\text{Assume that } n/2^k = 1,\ \ k=\log n] \\
&= n\,T(1) + c\,n\log n \\
&= a\,n + c\,n\log n \\
&= \mathbf{O(n \log n)}
\end{aligned}
$$

Best Case, Average Case and Worst Case Complexity of Merge Sort = O(n log n)

Trick:

Strassen Matrix Multiplication Formulae

$P = (A_{11} + A_{22})(B_{11} + B_{22})$

$Q = B_{11}(A_{21} + A_{22})$

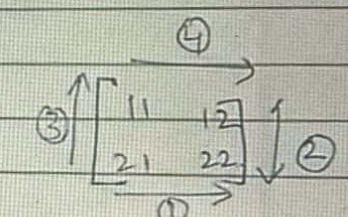
$R = A_{11}(B_{12} - B_{22})$

$S = A_{22}(B_{21} - B_{11})$

$T = B_{22}(A_{11} + A_{12})$

$U = (A_{21} - A_{11})(B_{11} + B_{12})$

$V = (A_{12} - A_{22})(B_{22} + B_{21})$



$B_{11} A_{11} A_{22} B_{22}$.

$T = B_{22}(A_{11} + A_{12})$

$U = (A_{21} - A_{11})(B_{11} + B_{12})$

$V = (A_{12} - A_{22})(B_{22} + B_{21})$

$C_{11} = P + S - T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

$C_{22} = P + R - Q + U$

$$T(n) = \begin{cases} b & \text{if } n<2 \\ 7\,T(n/2) + c\,n^2 & \text{Otherwise} \end{cases}$$

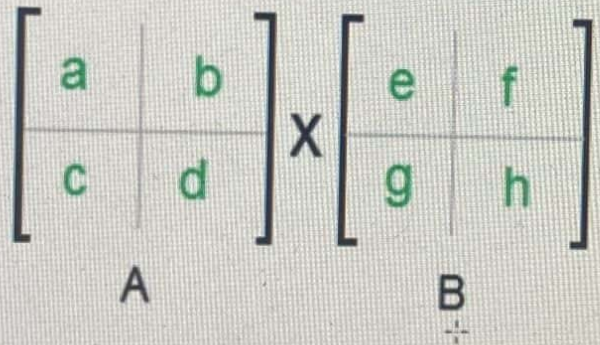
$$T(n) = 7\,T(n/2) + c\,n^2$$
$$= 7[7\,T(n/4) + c\,n^2/4\,] + c\,n^2$$
$$= 7^2T(n/2^2) + 7\,c\,n^2/4 + c\,n^2$$
$$= 7^3T(n/2^3) + 7^2\,c\,n^2/4^2 + 7\,c\,n^2/4 + c\,n^2$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$= 7^kT(n/2^k) + (7^{k-1}/4^{k-1})\,c\,n^2 + \ldots\ldots + (7/4)\,c\,n^2 + c\,n^2$$
$$= 7^kT(n/2^k) + [1+(7/4)+\ldots\ldots+(7^{k-1}/4^{k-1})\,]\,c\,n^2$$
$$\leq 7^kT(n/2^k) + [1+(7/4)+\ldots\ldots\ldots]\,c\,n^2$$
$$= 7^kT(n/2^k) + [1/(1-(7/4))]\,c\,n^2$$

[Assume that $n/2^k = 1 \rightarrow k = \log n$]

$$= 7^{\log n}T(1) - [4/3]\,c\,n^2$$
$$= n^{\log 7}O(1) - [4/3]\,c\,n^2$$
$$= \mathbf{O(n^{\log 7})}$$
$$= \mathbf{O(n^{2.81})}$$

Strassen's Matrix Multiplication Algorithm

1. A and B are the matrices with dimension nxn
2. If n is not a power of 2, then enough rows and columns of 0's can be added to both A and B so that the resulting dimensions are the power of two.
3. Partition A and B in to 4 square matrices of size n/2 x n/2



a, b, c and d are sub-matrices of A, of size n/2 x n/2

e, f, g and h are sub-matrices of B, of size n/2 x n/2

Greedy Method

- There is a problem with 'n' inputs
- ***Feasible solution*** : Obtain a subset that satisfies some constraints
- ***Optimal Solution*** : Find a feasible solution that either maximizes or minimizes a given objective function
- This method finds the feasible solutions as per the constraints and **finds the optimal solution** from it. The **optimum value can be a maximum or minimum value** depending upon the problem constraints

Greedy Method

```
Algorithm Greedy(a,n)        // n inputs
{
  solution = ϕ               // initialize the solution
  for(i=1 to n) do
   {
     x= Select (n)
     if (Feasible (solution,x) then
          solution = Union(solution,x)
   }
   return Solution
}
```

Fractional Knapsack Problem

- m - Knapsack(or bag) of capacity
- n - Number of objects
- W_i - Weight of object i
- P_i - Profit of object i
- X_i - Fraction of i^{th} object placed in the knapsack
- P_iX_i - Profit earned from i^{th} object

- The objective is to obtain an optimal solution of the knapsack that maximizes the total profit earned.
- The total weight of all the chosen objects should not be more than m.

Fractional Knapsack Problem

- Fractional knapsack problem can be stated as

Maximize $\sum_{i=1}^{n} PiXi$ ——— (1)

Subject to $\sum_{i=1}^{n} WiXi \leq m$ ——— (2)

$0 \leq Xi \leq 1$ and $1 \leq i \leq n$ ——— (3)

- A **feasible solution** satisfies equation 2 and 3.
- An **optimal solution** is a feasible solution that satisfies equation 1.

Find the optimal solution for the following fractional Knapsack problem. Given number of items(n)=4, capacity of sack(m) = 60, W={40,10,20,24} and P={280,100,120,120}

m = 60

n = 4

i ➔ { 1, 2, 3, 4 }

P = {280, 100, 120, 120}

W = {40, 10, 20, 24 }

P/W= {7, 10, 6, 5 }

Sort p[i]/w[i] ≥ p[i+1]/w[i+1]

i ➔ { 2, 1, 3, 4 }

P = {100, 280, 120, 120}

W= {10, 40, 20, 24}

i	Pi	Wi	Xi	U = U-Wi
2	100	10	1	50
1	280	40	1	10
3	120	20	1/2	0
4	120	24	0	

Total Profit = Σ Pi * Xi

$= 100x1 + 280x1 + 120x1/2 = \mathbf{440}$

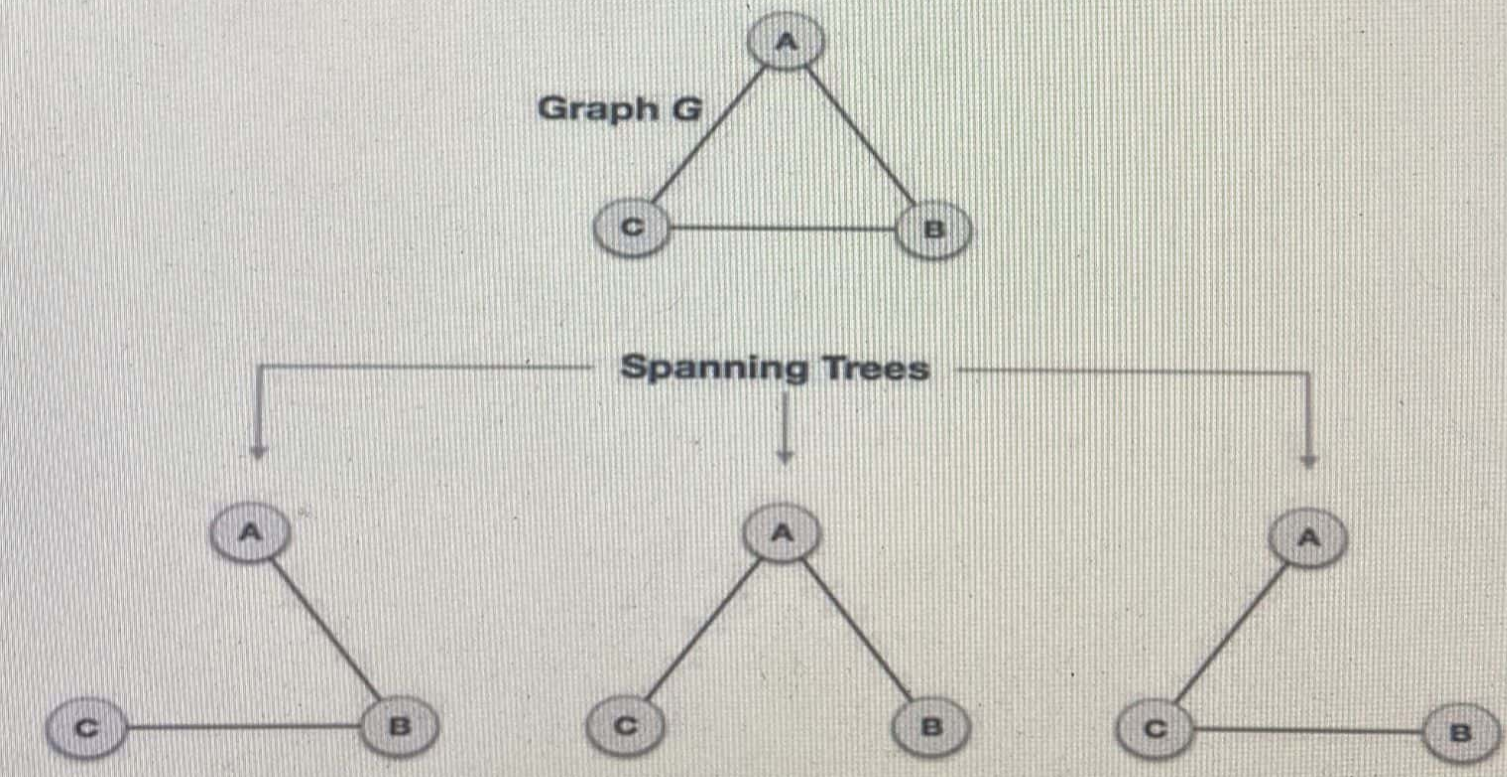
Solution vector **X ={1, 1, ½, 0}**

Fractional Knapsack Problem - Algorithm

```
Algorithm GreedyKnapsack(m, n)   // m → knapsack capacity
{   for i= 1 to n do
        x[i] = 0.0;                    // x[1:n] → solution vector
    U = m;
    for i=1 to n do
    {       if w[i] > U then
                break;
            x[i] = 1.0
            U = U – w[i];
    }
    If i ≤ n then
            x[i] = U / w[i];
}
```

Spanning Trees

- A spanning tree is a subset of undirected connected Graph G=(V,E), which has all the vertices covered with minimum possible number of edges.



Properties of a Spanning Tree

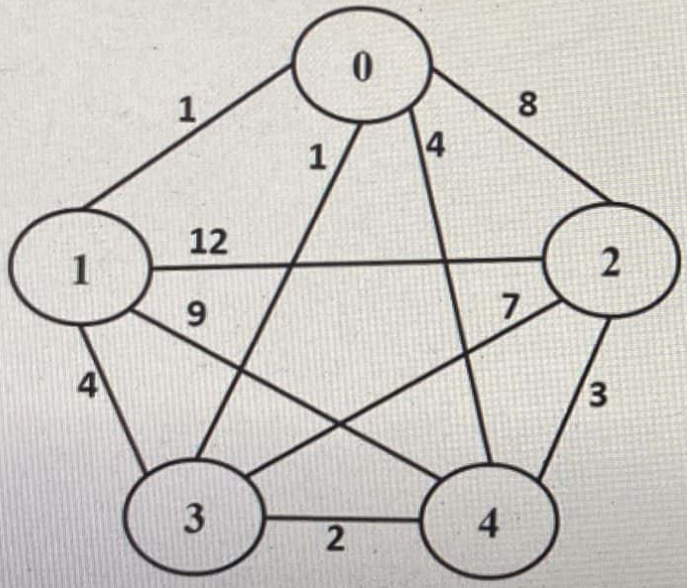
- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops)
- **Spanning tree is minimally connected:** Removing one edge from the spanning tree will make the graph disconnected.
- **Spanning tree is maximally acyclic:** Adding one edge to the spanning tree will create a circuit or loop.
- Spanning tree has **n-1** edges, where **n** is the number of nodes.

Minimum Spanning Tree(MST)

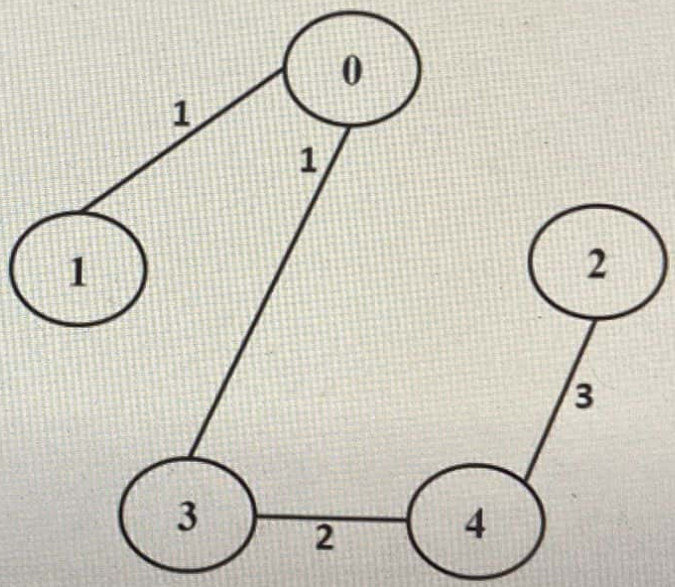
- A minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph.

- **Minimum Spanning-Tree Algorithms**
  - Prim's Algorithm
  - Kruskal's Algorithm

Minimum Spanning Tree Construction



G

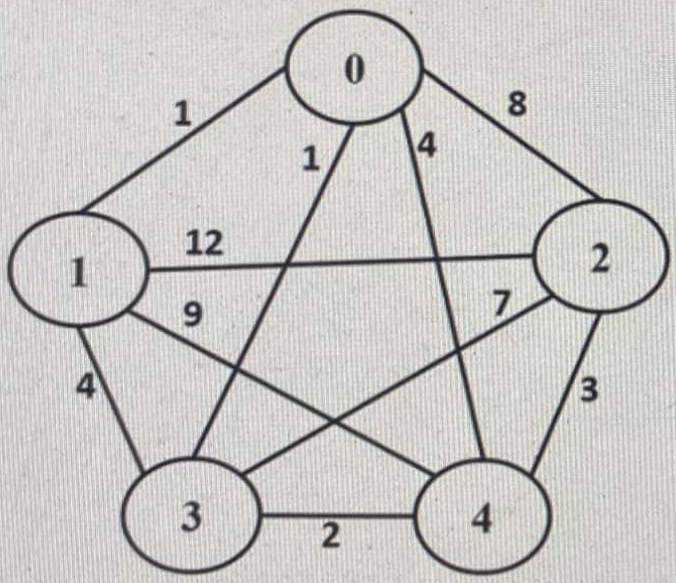


MST

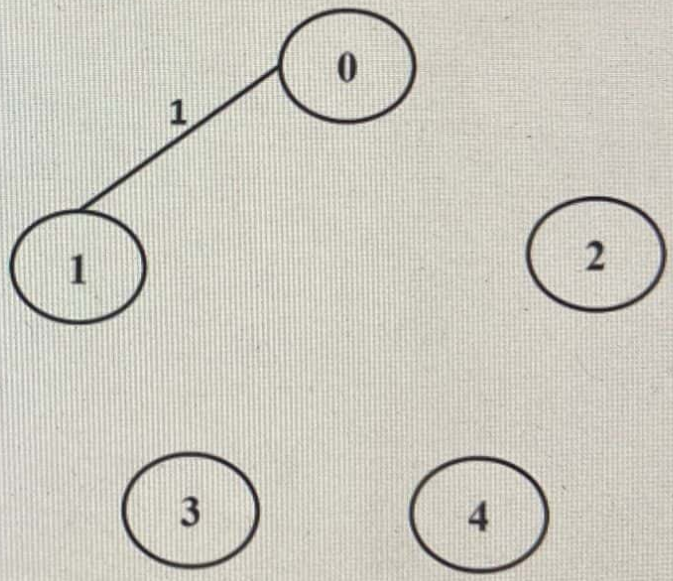
Cost = 7

5:54 / 9:57

Qtn) What is the minimum possible weight of a spanning tree T in this graph such that vertex 0 is a leaf node in the tree T?



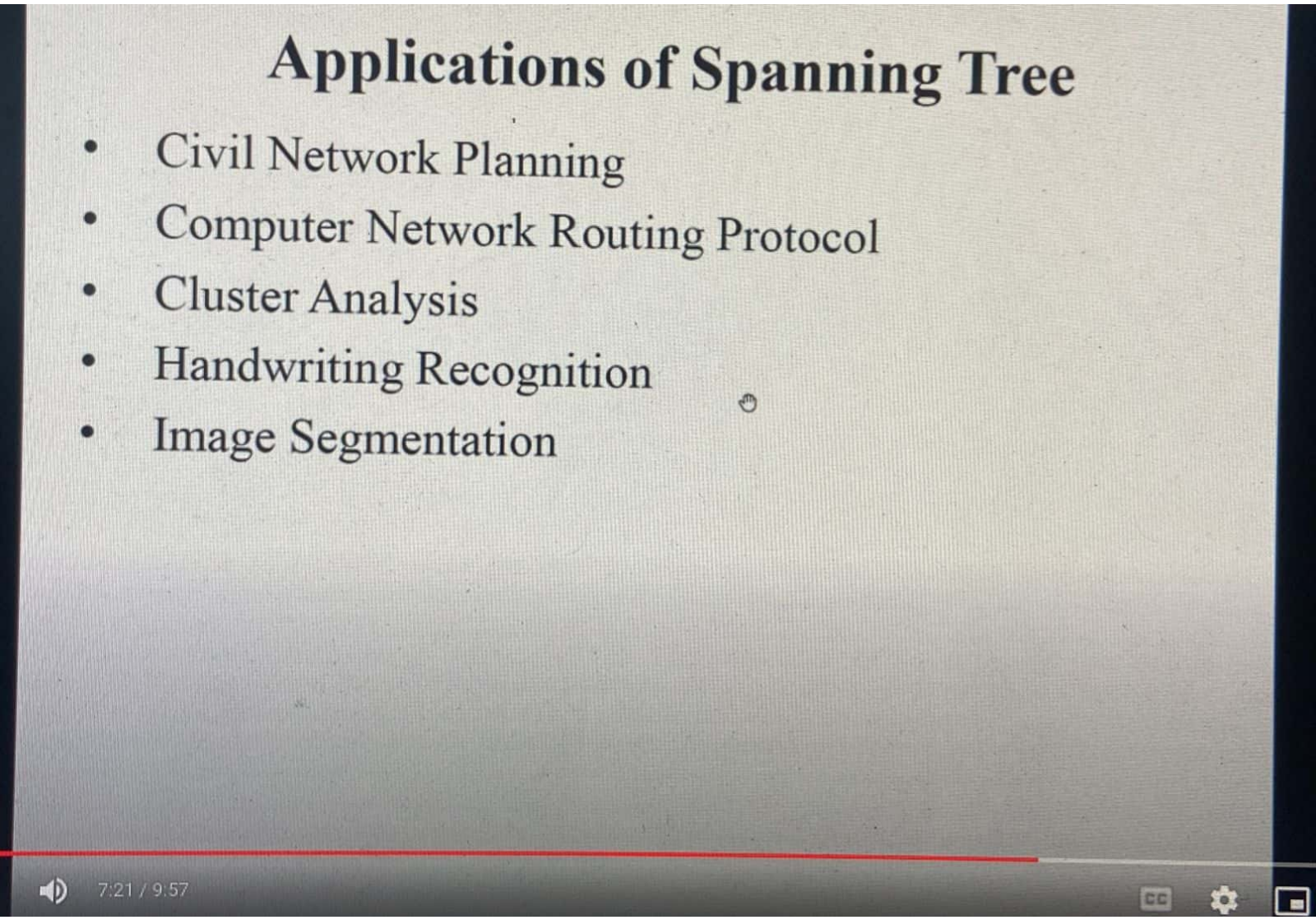
G



MST

Applications of Spanning Tree

- Civil Network Planning
- Computer Network Routing Protocol
- Cluster Analysis
- Handwriting Recognition
- Image Segmentation



Kruskal's Algorithm

- Builds this tree edge by edge.
- Edges are considered in the increasing order of cost.
- If the selected edge will form a cycle, then discard it.
- This selection process continues until there are n-1 edges

Kruskal's Algorithm

```
Algorithm kruskals(G, w)
{
    MST = Φ
    for(each vertex V ∈ V(G)) do
        Make_set(V)
    Sort all edges E(G) in
    increasing order of weight.
    for(each sorted edge(u,v)
        in E(G)) do
    if(find-set(u) ≠ find-set(v))
    {
        MST = MST ∪ (u,v)
        mincost = mincost + cost[u,v]
        union(u,v)
    }
}
```

```
Algorithm Make_set(v)
{
    p[v] = v
}
```

```
Algorithm find-set(x)
{
    While(p[x] != x) do
        x = p[x]
    return x
}
```

```
Algorithm union(x,y)
{
    xp = find-set(x)
    yp = find-set(y)
    if(xp != yp)
        p[xp] = yp
}
```

Kruskal's Algorithm - Complexity

- Construction of heap itself takes **O(|E|)** time.
- The edges are maintained as a minheap, then the nex edge to consider can be obtained in **O(log |E|)** time.
- We iterate through all edges
- Complexity of while loop = **O(|E|Log|E|)**

- The value of E can be at most $O(|V|^2)$

  So $O(Log|E|) = O(Log|V|^2) = O(2Log|V|) = O(Log|V|)$

- Therefore, the overall time complexity

  = **O(|E|log|V|) or O(|E|log|E|)**

Shortest Path Problem

- Finding a path between two vertices in a graph such that the sum of the weights of its constituent edges is minimized
- Different shortest path problems are:
  - **Single Source Shortest Path Problem**:
    - To find shortest paths from a source vertex to all other vertices in the directed graph.
  - **Single Destination Shortest Path Problem**:
    - To find shortest paths from all vertices in the directed graph to a single destination vertex v
  - **All Pairs Shortest Path Problem**:
    - To find shortest paths between every pair of vertices in the graph

Single Source Shortest Path Problem

- Given a connected weighted graph G=(V,E), find the shortest path from a given source vertex s to every other vertices (V-{s}) in the graph.
- The weight of any path(**w(p)**) is the sum of the weights of its constituted edges
- The weight of the shortest path from u to v = **min{w(p): p is a path from u to v}**

- Single Source Shortest Path Algorithms are:
  - Dijkstra's Algorithm
  - Bellman Ford Algorithm

Dijkstra's Algorithm

Assumes **no negative-weight edges**.

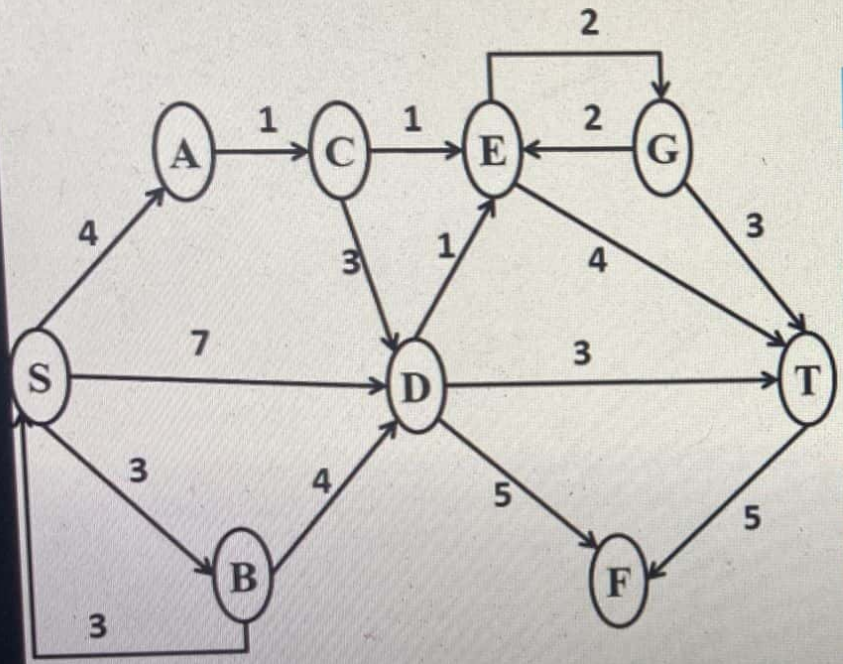
Maintains a set S of vertices whose SP from s has been determined.

Repeatedly selects u in V–S with minimum SP estimate (greedy choice).

Store V–S in priority queue Q.

```
Initialize(G, s);
S := ∅;
Q := V[G];
while Q ≠ ∅ do
      u := Extract-Min(Q);
      S := S ∪ {u};
      for each v ∈ Adj[u] do
            Relax(u, v, w)
      od
od
```

Dijkstra's Algorithm : Example



	S	A	B	C	D	E	F	G	T
S	0	α	α	α	α	α	α	α	α
B		4	3	α	7	α	α	α	α
A		4		α	7	α	α	α	α
C				5	7	α	α	α	α
E					7	6	α	α	α
D					7		α	8	10
G							12	8	10
T							12		10
F							12		

Min(preVal,lastFixed Val+distNeighbour)

Previous Node

S S A S C D E E