Greedy Algorithm

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.

JOB SEQUENCING WITH DEADLINES (Contd..)

- 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, $(p_1,p_2,p_3,p_4) = (100,10,15,27)$, $(d_1,d_2,d_3,d_4) = (2,1,2,1)$

JOB SEQUENCING WITH DEADLINES (Contd..)

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

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

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

$$p_1 \; p_4 \quad p_3 \quad p_2 \qquad d_1 \; d_4 \quad d_3 \; d_2$$

GREEDY ALGORITHM TO OBTAIN AN OPTIMAL SOLUTION (Contd..)

J = { 1} is a feasible one

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

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

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

J = { 1, 4} is optimal

GREEDY ALGORITHM FOR JOB SEQUENSING WITH DEADLINE

Procedure greedy job (D, J, n)

// J is the set of n jobs to be completed// K)

// by their deadlines //

J ← {1}

for I ← 2 to n do

If all jobs in JU{i} can be completed

by their deadlines

then J ← JU{I}

end if

repeat

end greedy-job

J may be represented by one dimensional array J (1:

The deadlines are

$D(J(1)) \leq D(J(2)) \leq .. \leq D(J(K))$

To test if JU {i} is feasible, we insert i into J and verify

$D(J®) \leq r \quad 1 \leq r \leq k+1$

GREEDY ALGORITHM FOR SEQUENCING UNIT TIME JOBS

Procedure JS(D,J,n,k)

// D(i) $\geq$ 1, 1$\leq$ i $\leq$ n are the deadlines //

// the jobs are ordered such that //

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

// in the optimal solution ,D(J(i) $\geq$ D(J(i+1)) //

// 1 $\leq$ i $\leq$ k //

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

D(0) ←J(0) ← 0

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

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

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

GREEDY ALGORITHM FOR SEQUENCING UNIT TIME JOBS (Contd..)

```
// find the position of i and check  feasibility of insertion //
   r← k   // r and k are indices for existing job in J //
// find r such that i can be inserted after r //
while D(J(r)) > D(i) and D(i) ≠ r do
// job r can be processed after i and //
// deadline of job r is not exactly r //
   r← r-1 // consider whether job r-1 can be processed after i //
repeat
```

GREEDY ALGORITHM FOR SEQUENCING UNIT TIME JOBS (Contd..)

if $D(J(r)) \geq d(i)$ and $D(i) > r$ then

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

for $I \leftarrow k$ to $r+1$ by -1 do

$J(I+1) \leftarrow J(I)$ // shift jobs(r+1) to k right by//

//one position //

repeat

GREEDY ALGORITHM FOR SEQUENCING UNIT TIME JOBS (Contd..)

J(r+1)$\leftarrow$i ; k $\leftarrow$k+1

// i is inserted at position r+1 //

// and total jobs in J are increased by one //

repeat

end JS

COMPLEXITY ANALYSIS OF JS ALGORITHM

- Let n be the number of jobs and s be the number of jobs included in the solution.
- The loop between lines 4-15 (the for-loop) is iterated (n-1)times.
- Each iteration takes O(k) where k is the number of existing jobs.

$\therefore$ The time needed by the algorithm is $0(sn)$ $s \leq n$ so the worst case time is $0(n^2)$.

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

D and J need $\theta(s)$ amount of space.

Example

EXAMPLE: let n = 5, (p_1,--------p_5) = (20,15,10,5,1) and (d_1,--d_5) = (2,2,1,3,3). Using the above rule

J	assigned slot	jobs being considered	action or
Ø	none	1	assigned to [1, 2]
{1}	[1,2]	2	[0,1]
{1,2}	[0,1],[1,2]	3	cannot fit reject as [0,1] is not free
{1,2}	[0,1],[1,2]	4	assign to [2,3]
{1,2,4}	[0,1],[1,2],[2,3]	5	reject

The optimal solution is {1,2,4} with profit 40

1. **Minimum Spanning Tree** (For Undirected Graph)

The problem:

1) Tree

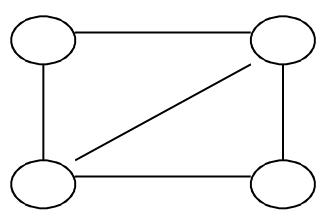
A *Tree* is connected graph with no cycles.

2) Spanning Tree

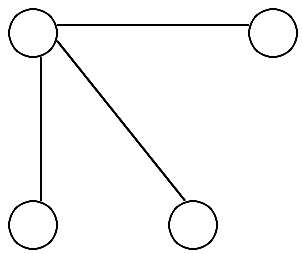
A *Spanning Tree* of *G* is a tree which contains all vertices in *G*.

Example:

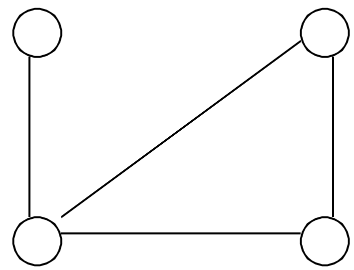
G:



b) Is G a Spanning Tree?



Key: Yes



Key: No

Note: Connected graph with n vertices and exactly $n - 1$ edges is Spanning Tree.

3) Minimum Spanning Tree

Assign weight to each edge of G, then *Minimum Spanning Tree* is the Spanning Tree with minimum total weight.

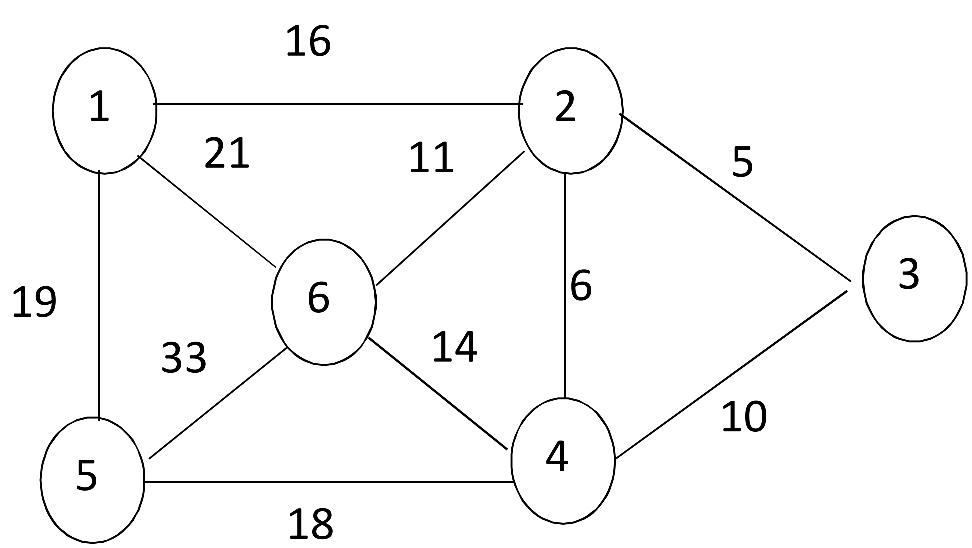
Algorithms:

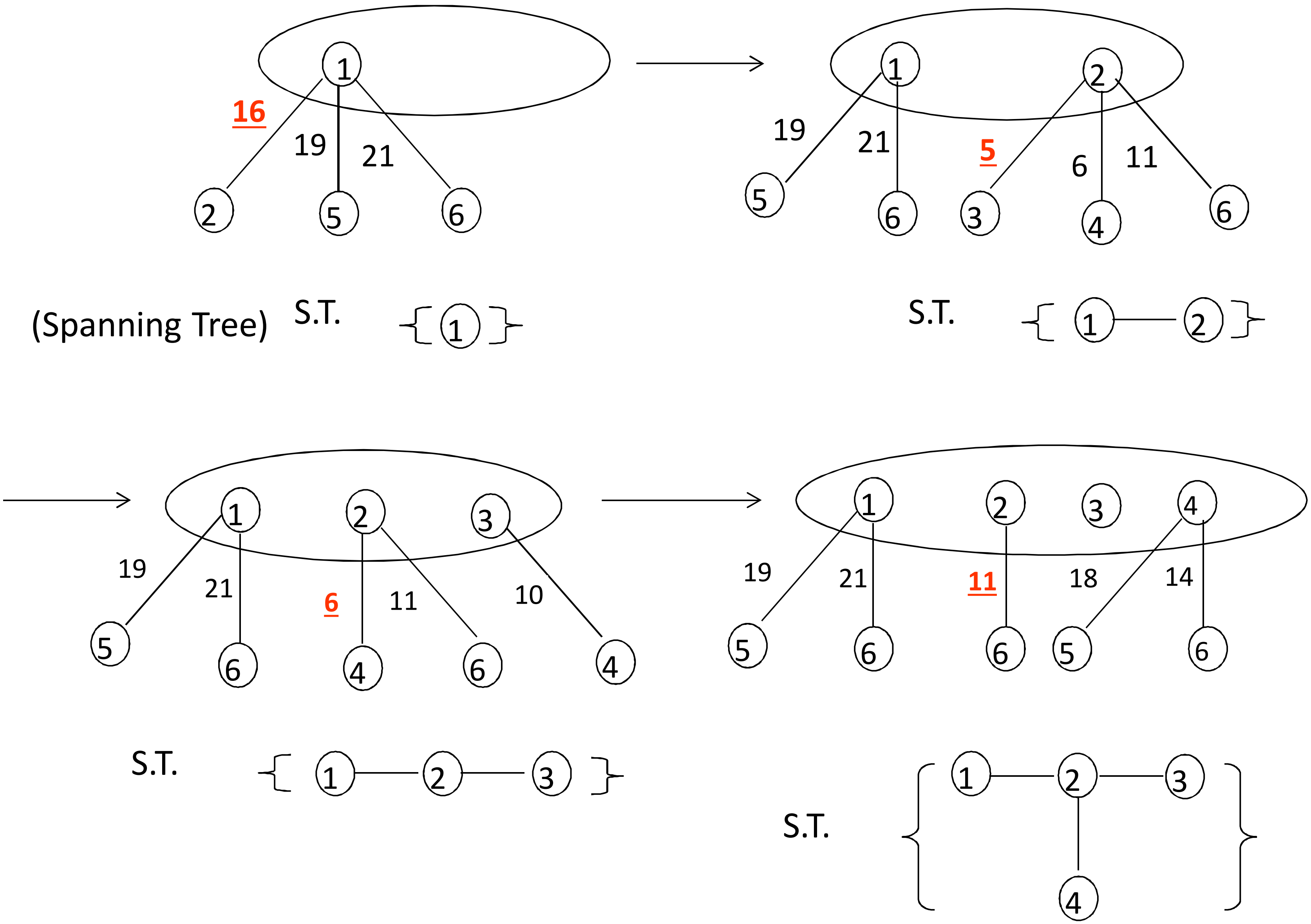
1) *Prim's Algorithm* *(Minimum Spanning Tree)*

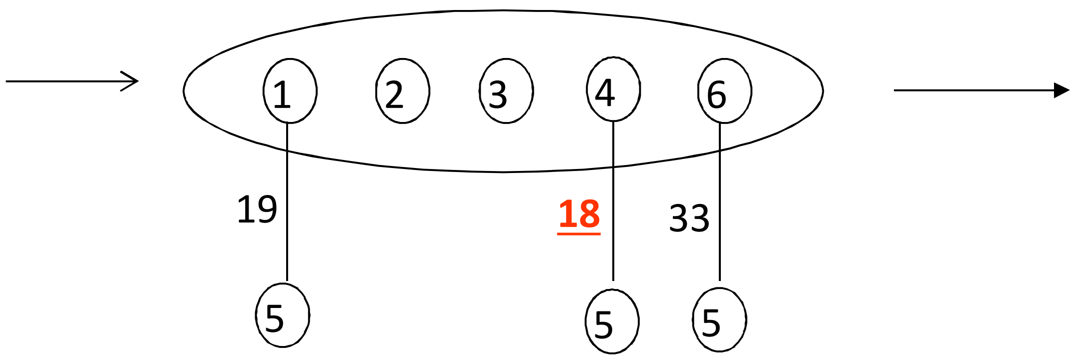
Basic idea:

Start from vertex 1 and let T $\leftarrow$ Ø (T will contain all edges in the S.T.); the next edge to be included in T is the minimum cost edge(u, v), s.t. u is in the tree and v is not.

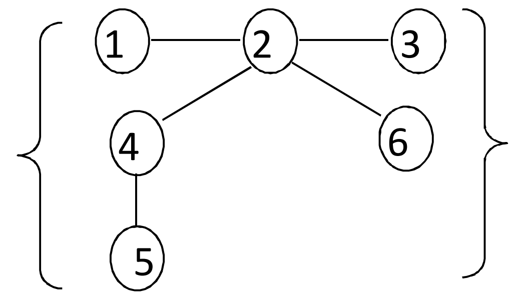
Example*:* *G*




16
19
21
(Spanning Tree) S.T.
19
21
5
6
11
S.T.
19
21
6
11
10
S.T.
19
21
11
18
14
S.T.



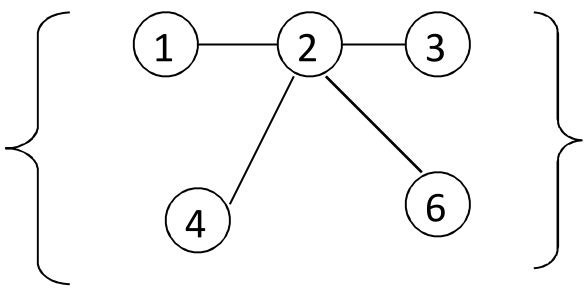
S.T.



Cost = 16+5+6+11+18=56

Minimum Spanning Tree

S.T.



(*n* – # of vertices, *e – # of edges)*

It takes *O(n)* steps. Each step takes *O(e) and e* $\leq$ *n(n-1)/2*

Therefore, it takes O(n^3*) time.*

$$\Rightarrow O(n^2).$$

With clever data structure, it can be implemented in *O(n*^{2}*).*

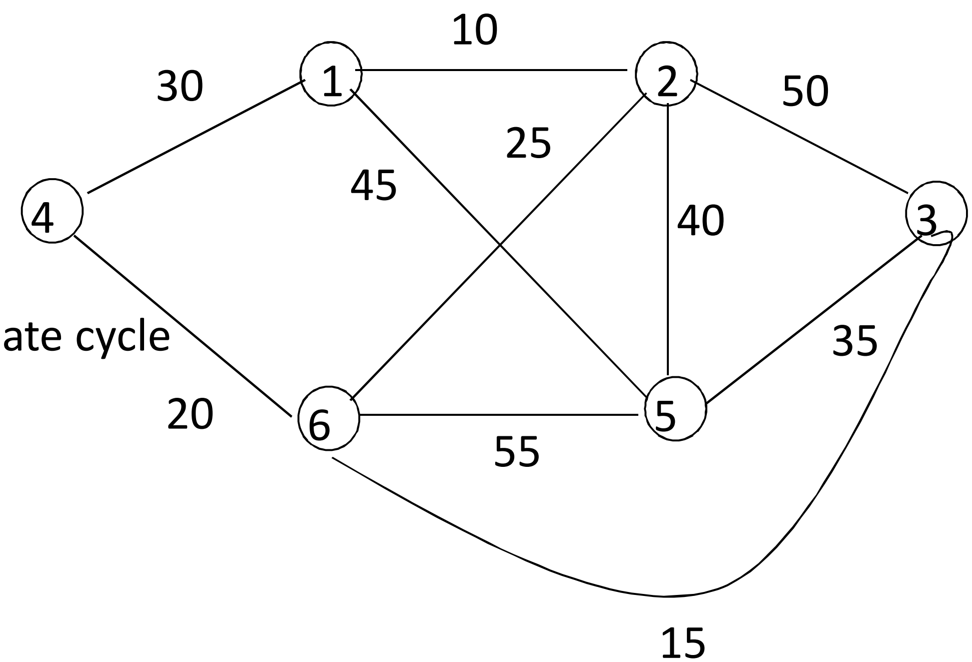
2) Kruskal's Algorithm

Basic idea:

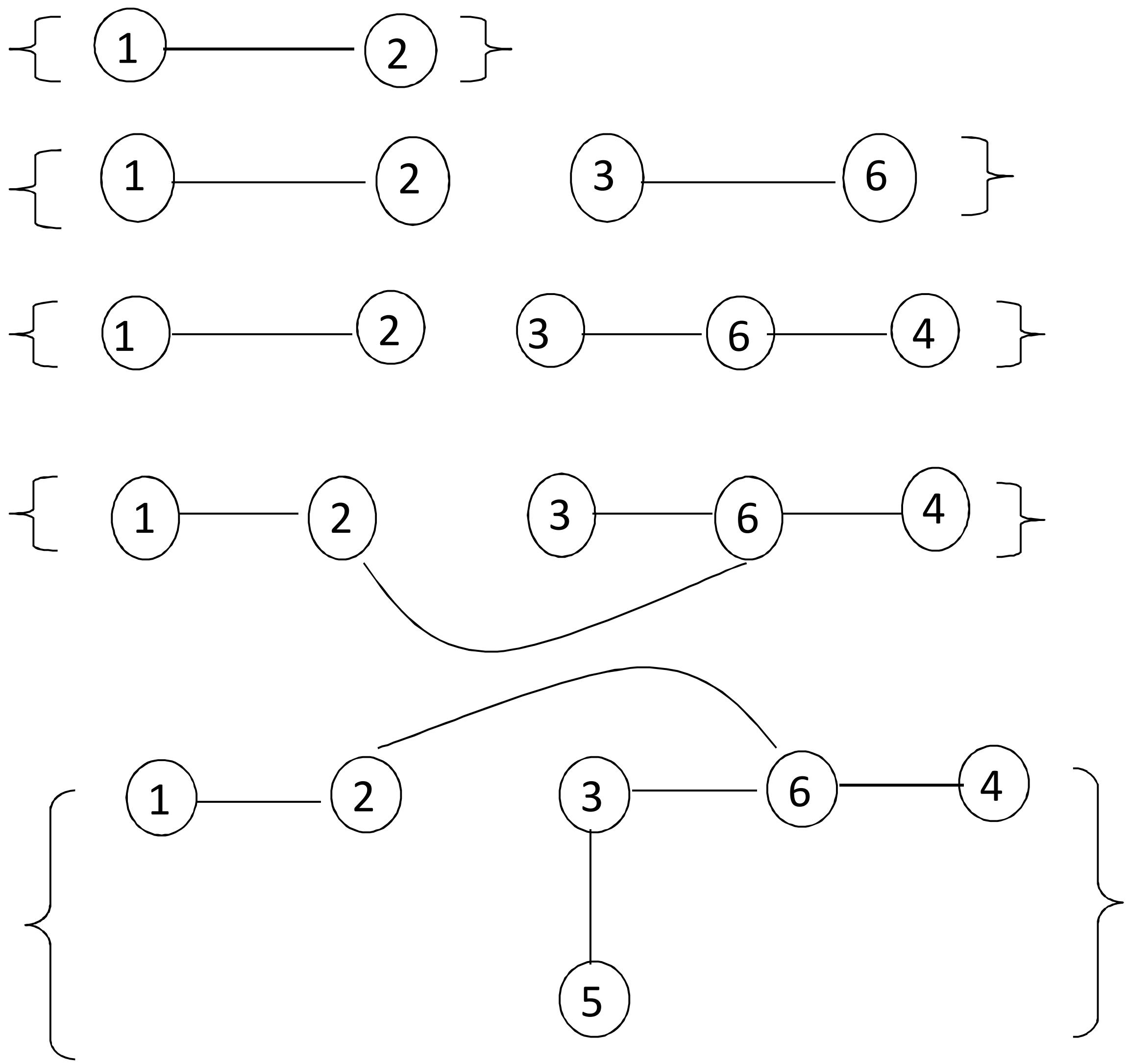
Don't care if *T* is a tree or not in the intermediate stage, as long as the including of a new edge <u>will not create a cycle</u>, we include the minimum cost edge

Example:

Step 1:	Sort all of edges		
(1,2)	10	√	
(3,6)	15	√	
(4,6)	20	√	
(2,6)	25	√	
(1,4)	30	×	reject ∵ create cycle
(3,5)	35	√	



Step 2: T



Kruskal's algorithm

While (T contains fewer than n-1 edges) and ($E \neq \emptyset$) do
Begin

Choose an edge (v,w) from E of lowest cost;
Delete (v,w) from E;
If (v,w) does not create a cycle in T
then add (v,w) to T
else discard (v,w);

End;

With clever data structure, it can be implemented in *O(e log e).*

So, complexity of Kruskal is

$$\left.\begin{array}{r} O(eLoge) \\ \because e \le \dfrac{n(n-1)}{2} \Rightarrow Loge \le Logn^2 = 2Logn \end{array}\right\} \Rightarrow O(eLoge) = O(eLogn)$$

3) Comparing Prim's Algorithm with Kruskal's Algorithm

i. Prim's complexity is $O(n^2)$

ii. Kruskal's complexity is $O(eLogn)$

if G is a complete (dense) graph,
Kruskal's complexity is $O(n^2 Logn)$
if G is a sparse graph,
Kruskal's complexity is $O(nLogn)$.

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

$$t_j = \sum_{k=1}^{j} L_{i_k}$$

Optimal Storage on Tapes

- If all programs are retrieved equally often, then the mean retrieval time (MRT) = $\frac{1}{n}\sum_{j=1}^{n} t_j$

- This problem fits the ordering paradigm. Minimizing the MRT is equivalent to minimizing

$$d(I) = \sum_{j=1}^{n}\sum_{k=1}^{j} L_{i_k}$$

The goal is to minimize MRT (Mean Retrieval Time), $\frac{1}{n}\sum_{j=1}^{n} t_j$

i.e. want to minimize $\sum_{j=1}^{n}\sum_{k=1}^{j} l_{i_k}$

Ex: $n = 3, \quad (l_1, l_2, l_3) = (5,10,3)$

There are n! = 6 possible orderings for storing them.

	order	total retrieval time	MRT
1	1 2 3	5+(5+10)+(5+10+3)=38	38/3
2	1 3 2	5+(5+3)+(5+3+10)=31	31/3
3	2 1 3	10+(10+5)+(10+5+3)=43	43/3
4	2 3 1	10+(10+3)+(10+3+5)=41	41/3
5	3 1 2	3+(3+5)+(3+5+10)=29	29/3 ← Smallest
6	3 2 1	3+(3+10)+(3+10+5)=34	34/3

Note: The problem can be solved using greedy strategy,
just always let the shortest program goes first.
(Can simply get the right order by using any sorting algorithm)

***Analysis*:**

Try all combination: O(n!)

Shortest-length-First Greedy method: O (*n*log*n*)

Shortest-length-First Greedy method:

Sort the programs s.t. $l_1 \leq l_2 \leq \ldots \leq l_n$
and call this ordering *L.*

Next is to show that the ordering *L* is the best

Optimal merge pattern

- Merge a set of sorted files of different length into a single sorted file. We need to find an optimal solution, where the resultant file will be generated in minimum time.
- If the number of sorted files are given, there are many ways to merge them into a single sorted file. This merge can be performed pair wise. Hence, this type of merging is called as **2-way merge patterns**.
- As, different pairings require different amounts of time, in this strategy we want to determine an optimal way of merging many files together. At each step, two shortest sequences are merged.

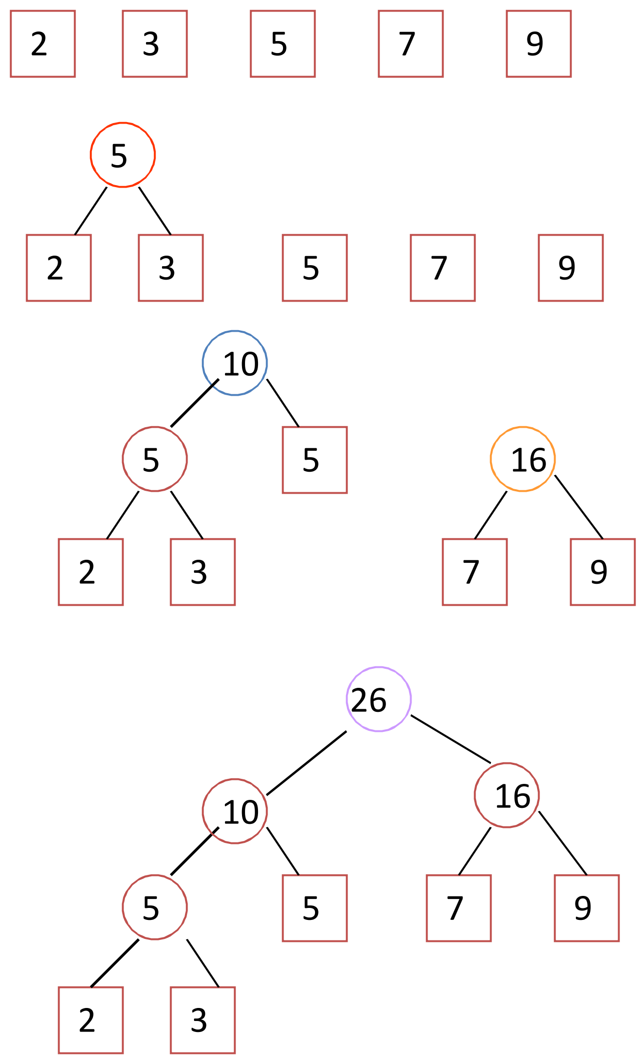
- To merge a **p-record file** and a **q-record file** requires possibly **p + q** record moves, the obvious choice being, merge the two smallest files together at each step.
- Two-way merge patterns can be represented by binary merge trees. Let us consider a set of **n** sorted files $\{f_1, f_2, f_3, \ldots, f_n\}$. Initially, each element of this is considered as a single node binary tree. To find this optimal solution, the following algorithm is used.

- **Algorithm: TREE (n)**
- for i := 1 to n – 1 do
- declare new node
- node.leftchild := least (list)
- node.rightchild := least (list)
- node.weight) := ((node.leftchild).weight) + ((node.rightchild).weight)
- insert (list, node);
- return least (list);
  - At the end of this algorithm, the weight of the root node represents the optimal cost.
  - The complexity of algorithm is O(nlgn)

- Example
- Let us consider the given files, f_1, f_2, f_3, f_4 and f_5 with 20, 30, 10, 5 and 30 number of elements respectively.
- If merge operations are performed according to the provided sequence, then
- **M_1 = merge f_1 and f_2** => 20 + 30 = 50
- **M_2 = merge M_1 and f_3** => 50 + 10 = 60
- **M_3 = merge M_2 and f_4** => 60 + 5 = 65
- **M_4 = merge M_3 and f_5** => 65 + 30 = 95
- Hence, the total number of operations is
- 50 + 60 + 65 + 95 = 270
- Now, the question arises is there any better solution?

- Sorting the numbers according to their size in an ascending order, we get the following sequence −
- **f_4, f_3, f_1, f_2, f_5**
- Hence, merge operations can be performed on this sequence
- **M_1 = merge f_4 and f_3** => 5 + 10 = 15
- **M_2 = merge M_1 and f_1** => 15 + 20 = 35
- **M_3 = merge M_2 and f_2** => 35 + 30 = 65
- **M_4 = merge M_3 and f_5** => 65 + 30 = 95
- Therefore, the total number of operations is
- 15 + 35 + 65 + 95 = 210
- Obviously, this is better than the previous one.

Example of the optimal merge tree algorithm:



Initially, 5 leaf nodes with sizes

Iteration 1: merge 2 and 3 into 5

Iteration 2: merge 5 and 5 into 10

Iteration 3: merge 7 and 9 (chosen among 7, 9, and 10) into 16

Iteration 4: merge 10 and 16 into 26

Cost = 2*3 + 3*3 + 5*2 + 7*2 + 9*2 = 57.

END