Greedy Method

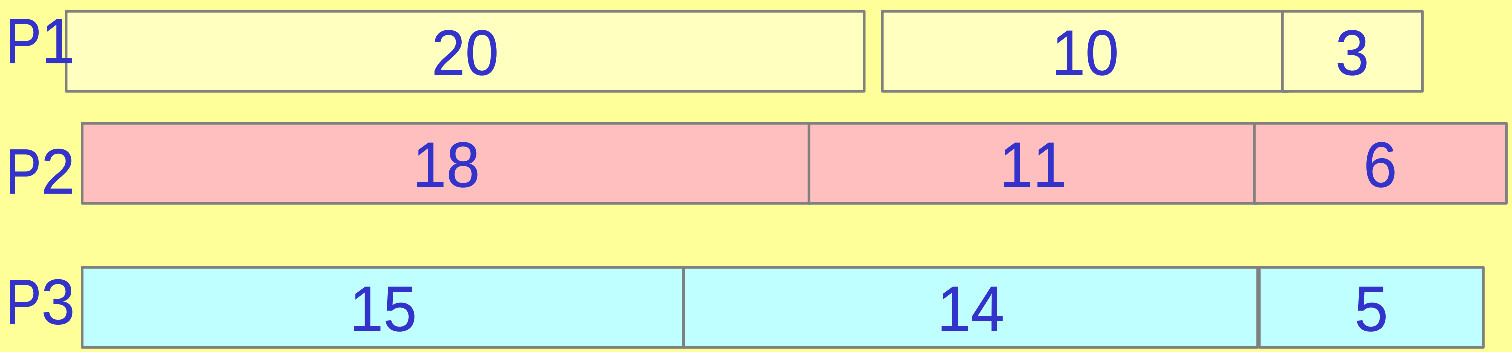
- A greedy algorithm always makes the choice that looks best at the moment.
- It makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.
- Greedy algorithms do not always yield optimal solutions, but for many problems they do.

Optimization problems

- An optimization problem is one in which you want to find, not just *a* solution, but the *best* solution
- A "greedy algorithm" sometimes works well for optimization problems
- A greedy algorithm works in phases. At each phase:
  - You take the best you can get right now, without regard for future consequences
  - You hope that by choosing a ***local*** optimum at each step, you will end up at a ***global*** optimum

A scheduling problem

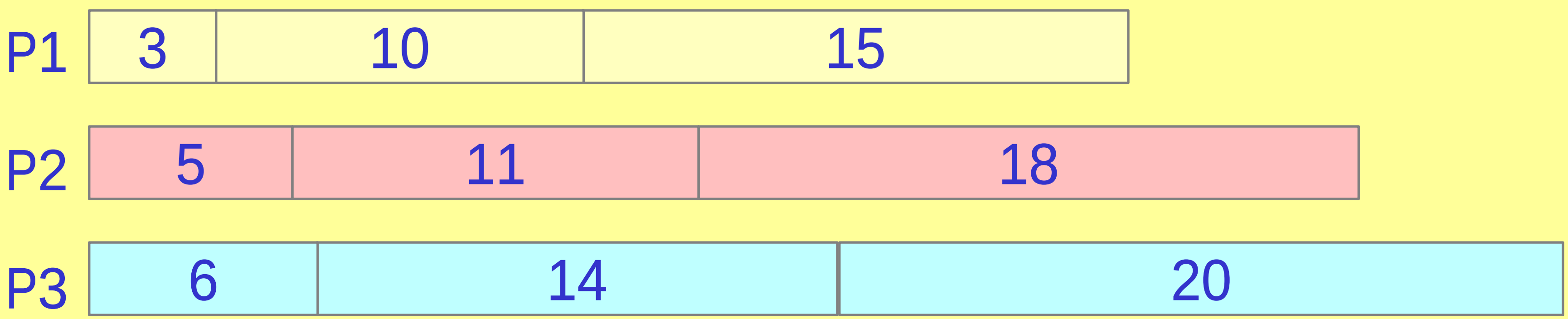
- You have to run nine jobs, with running times of 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes
- You have three processors on which you can run these jobs
- You decide to do the longest-running jobs first, on whatever processor is available



- Time to completion: 18 + 11 + 6 = 35 minutes
- This solution isn't bad, but we might be able to do better

Another approach

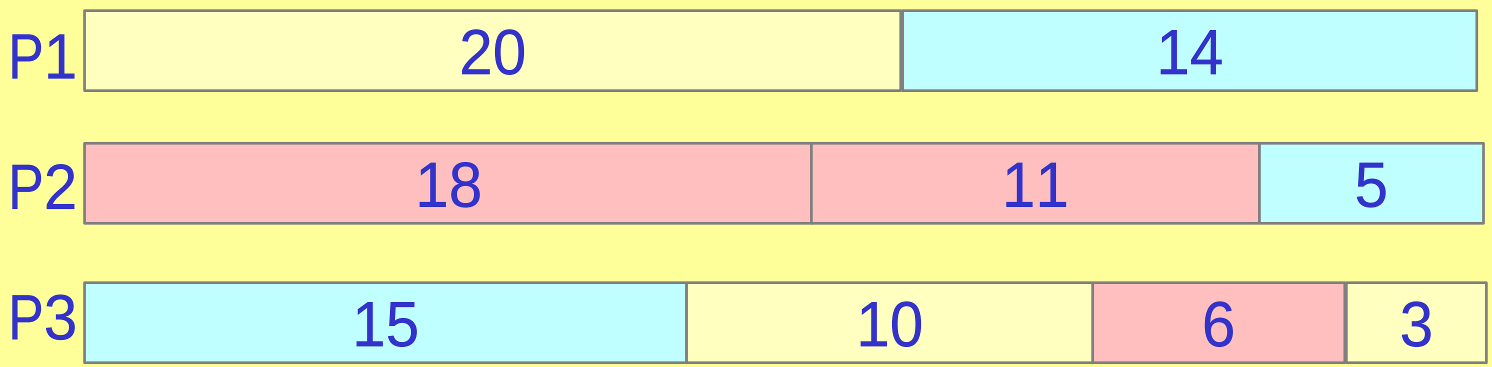
- What would be the result if you ran the *shortest* job first?
- Again, the running times are 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes



- That wasn't such a good idea; time to completion is now 6 + 14 + 20 = 40 minutes
- Note, however, that the greedy algorithm itself is fast
  - All we had to do at each stage was pick the minimum or maximum

An optimum solution

- Better solutions do exist:



- This solution is clearly optimal (why?)
- Clearly, there are other optimal solutions (why?)
- How do we find such a solution?
  - **One way: Try all possible assignments of jobs to processors**
  - **Unfortunately, this approach can take exponential time**

Fractional Knapsack Problem

We are given n objects and a bag or knapsack.

Object i has a weight w_i

the knapsack has a capacity m.

If a fraction x_I , $0 \le x_i \le 1$ of

Object i is placed into the bag

then a profit of $p_i x_i$ is earned.

The objective is to obtain a filling of the knapsack that maximizes the total profit earned.

Maximize $$\sum_{1 \le i \le n} p_i x_i$$

$$subject\ to$$

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

$$and$$

$$o \le x_i \le 1, \qquad 1 \le i \le n$$

A feasible solution is any set (x $_1$,…. x $_n$) satisfying the above inequalities. An optimal solution is a feasible solution for which objective function is maximized.

0-1 knapsack problem

$$\max imize \sum_{1 \le i \le n} p_i x_i$$

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

$$and \qquad 0 \le x_i \le 1, \quad 1 \le i \le n$$

The prifits and weights are positive numbers.

Xi are o or 1

Greedy Algorithms for Knapsack Problem

Several simple greedy strategies are there.

- We can fill the bag by including next object with largest profit.

- Example: $n = 3$, $m = 20$,
- $(p_1, p_2, p_3) = (25, 24, 15)$
- $(w_1, w_2, w_3) = (18, 15, 10)$

By above strategy solution is

$$x_1 = 1, \; x_2 = 2/15, \qquad x_3 = 0$$

Total weight = 20, profit = 28.2

But this solution is not the best one.

(0,1,1/2) is a solution

having profit 31.5

Another greedy approach

To become greedy w.r.t. capacity:

Start with lightest object and so on:

(0 ,2/3 , 1)   with profit = 31

Again not an optimal solution.

New greedy approach

With a balance between weight and profit

At each step we include that object which has the maximum profit per unit of capacity used.

Using this strategy solution obtained is:

(0, 1 , ½) with profit = 31.5

Which is an optimal solution. For this example.

So this strategy gave optimal solution for this data:

In fact we can prove that this strategy will give optimal solution for any data.

.

There seem to be 3 obvious greedy strategies:

(Max value) Sort the objects from the highest value to the lowest, then pick them in that order.

(Min weight) Sort the objects from the lowest weight to the highest, then pick them in that order.

(Max value/weight ratio) Sort the objects based on the value to weight ratios, from the highest to the lowest, then select.

Example: Given $n = 5$ objects and a knapsack capacity $W = 100$ as in Table I. The three solutions are given in Table II.

w	10	20	30	40	50
v	20	30	66	40	60
v/w	2.0	1.5	2.2	1.0	1.2

select	x_i					value
Max v_i	0	0	1	0.5	1	146
Min w_i	1	1	1	1	0	156
Max v_i/w_i	1	1	1	0	0.8	164

The Optimal Knapsack Algorithm:

Algorithm (of time complexity $O(n \lg n)$)

(1) Sort the n objects from large to small based on the ratios v_i/w_i . We assume the arrays $w[1..n]$ and $v[1..n]$

(2) store the respective weights and values after sorting.

(3) initialize array $x[1..n]$ to zeros.
weight = 0; i = 1

(4) while ($i \le n$ and weight < W) do
(4.1) if weight + $w[i] \le W$ then $x[i] = 1$
(4.2) else $x[i]$ = (W – weight) / w[i]
(4.3) weight = weight + x[i] * w[i]
(4.4) i++

Theorem: If p1 /w1 $\geq$ p2/ w2 $\geq$ …pn/wn .then selecting items w.r.t. this ordering gives optimal solution.

Proof: Let $x = (x_1, \ldots x_n)$ be the solution so generated.

By obvious reasons $\Sigma wi\ yi = m$

If all the xi's are one then clearly the solution is optimal.

let j be the least index such that $x_j \neq 1$.

So $x_i = 1$ for all $1 \leq i < j$ and

$x_i = 0$ for $j < i \leq n$ and $0 \leq x_j < 1$.

Let $y = (y_1, y_2, \ldots y_n)$ be an optimal solution

$1 \quad 1 \quad 1 \quad \ldots\ldots 1 \quad X_j \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0$

$$1 \quad 1 \quad 1 \quad \ldots\ldots 1 \quad X_j \quad 0 \; 0 \; 0 \quad 0 \quad 0 \; 0$$

$$y_1 \quad y_2 \qquad y_k \; y_{k+1} \ldots\ldots\ldots\ldots\ldots$$

Let k be the least index such that $yk \neq xk$.

It is easy to reason that $yk < xk$.

- Now suppose we increase yk to xk and decrease as many of (yk+1, …. Yn) as necessary so that total capacityof bag is still m.

This results in a new solution

$$Z = (z1 , z2, \ldots zn)$$

$$zi = xi \qquad\qquad 1 \le i \le k$$

And

$$\sum_{k < i \le n} wi \ (yi - zi) = wk(zk - yk)$$

$$\sum_{1\le i\le n} p_i z_i = \sum_{1\le i\le n} p_i y_i + (z_k - y_k) w_k p_k / w_k$$

$$- \sum_{k<i\le n} (y_i - z_i) w_i p_i / w_i$$

$$\ge \sum_{1\le i\le n} p_i y_i + \Big[(z_k - y_k) w_k - \sum_{k<i\le n} (y_i - z_i) w_i\Big] p_k / w_k$$

$$= \sum_{1\le i\le n} p_i y_i$$

If Σ pi zi > Σ pi yi then y could not have been optimal solution.

- If these sums are equal then either $z = x$ and x is optimal or $z \neq x$.

- In the latter case repeated use of above argument y can be transformed into x without changing the utility value,and thus x too is optimal

Optimal 2-way Merge patterns and Huffman Codes:

Example. Suppose there are 3 sorted lists L_1, L_2, and L_3, of sizes 30, 20, and 10, respectively, which need to be merged into a combined sorted list, but we can merge only two at a time.

We intend to find an optimal merge pattern which minimizes the total number of comparisons.

For example, we can merge L_1 and L_2, which uses 30 + 20 = 50 comparisons resulting in a list of size 50.

We can then merge this list with list L_3, using another 50 + 10 = 60 comparisons,

so the total number of comparisons 50 + 60 = 110.

Alternatively, we can first merge lists L_2 and L_3,

using 20 + 10 = 30 comparisons,

the resulting list (size 30) can then be merged with list L_1,

for another 30 + 30 = 60 comparisons.

So the total number of comparisons is 30 + 60 = 90.

It doesn't take long to see that this latter merge pattern is the optimal one.

Binary Merge Trees:

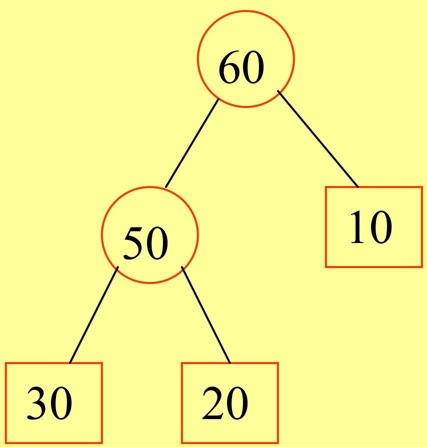
We can depict the merge patterns using a binary tree, built from the leaf nodes (the initial lists) towards the root in which each merge of two nodes creates a parent node whose size is the sum of the sizes of the two children. For example, the two previous merge patterns are depicted in the following two figures:

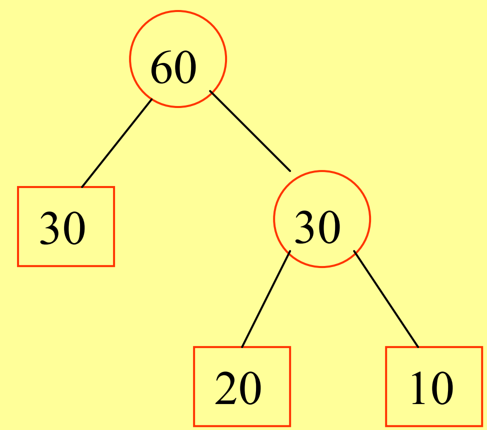
Cost = 30*2 + 20*2 + 10*1 = 110

Cost = 30*1 + 20*2 + 10*2 = 90

Merge L_1 and L_2, then with L_3

Merge L_2 and L_3, then with L_1





merge cost = sum of all weighted external path lengths

Optimal Binary Merge Tree Algorithm:

Input: n leaf nodes each have an integer size, $n \geq 2$.

Output: a binary tree with the given leaf nodes which has a minimum total weighted external path lengths

Algorithm:

(1) create a min-heap $T[1..n]$ based on the n initial sizes.
(2) while (the heap size ≥ 2) do
    (2.1) delete from the heap two smallest values, call them a and b, create a parent node of size $a + b$ for the nodes corresponding to these two values
    (2.2) insert the value $(a + b)$ into the heap which corresponds to the node created in Step (2.1)

When the algorithm terminates, there is a single value left in the heap whose corresponding node is the root of the optimal binary merge tree. The algorithm's time complexity is $O(n \lg n)$ because Step (1) takes $O(n)$ time; Step (2) runs $O(n)$ iterations, in which each iteration takes $O(\lg n)$ time.

Example of the optimal merge tree algorithm:

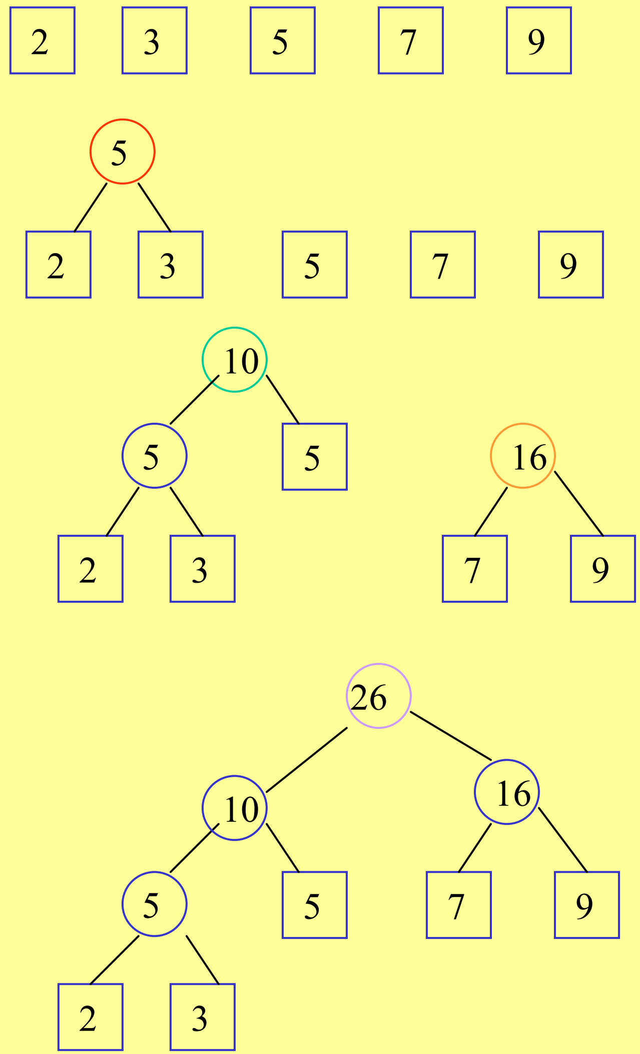
Initially, 5 leaf nodes with sizes

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.

Proof of optimality of the binary merge tree algorithm:

We use induction on $n \geq 2$ to show that the binary merge tree is *optimal* in that it gives the minimum total weighted external path lengths (among all possible ways to merge the given leaf nodes into a binary tree).

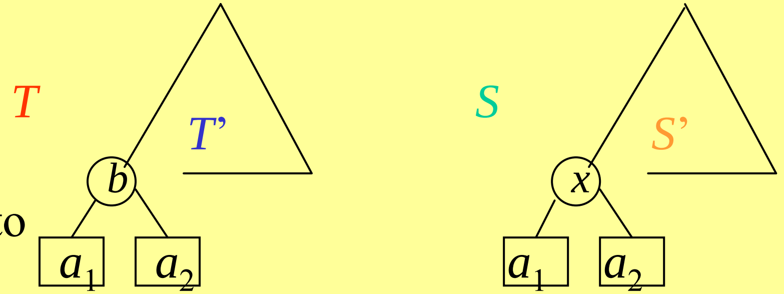
(Basis) When $n = 2$. There is only one way to merge two nodes.

(Induction Hypothesis) Suppose the merge tree is optimal when there are k leaf nodes, for some $k \geq 2$.

(Induction) Consider $(k + 1)$ leaf nodes. Call them a_1, a_2, …, and a_{k+1}. We may assume nodes a_1, a_2 are of the smallest values, which are merged in the first step of the merge algorithm into node b. We call the merge tree T, the part excluding a_1, a_2 T' (see figure). Suppose an optimal binary merge tree is S. We make two observations.

(1) If node x of S is a deepest internal node, we may swap its two children with nodes a_1, a_2 in S without increasing the total weighted external path lengths. Thus, we may assume tree S has a subtree S' with leaf nodes x, a_2, …, and a_{k+1}.

(2) The tree S' must be an optimal merge tree for k nodes x, a_2, …, and a_{k+1}. By induction hypothesis, tree S' has a total weighted external path lengths equal to that of tree T'. Therefore, the total weighted external path lengths of T equals to that of tree S, proving the optimality of T.



Minimum Spanning Tree problem

- Kruskal's Algorithm
- Prim's algorithm

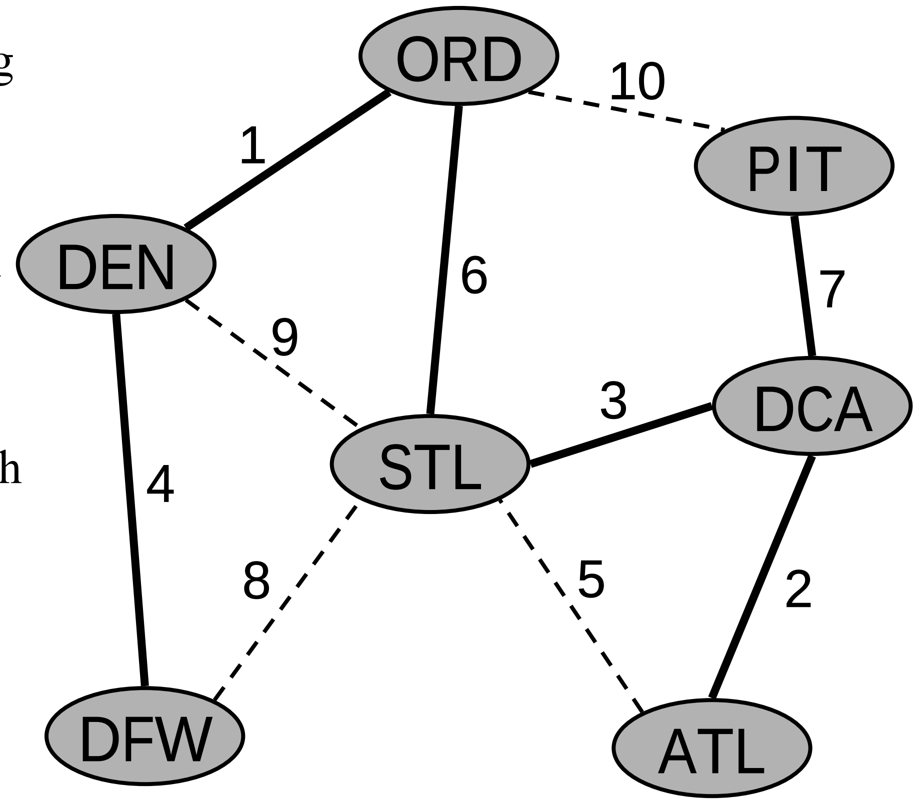
Both are based on greedy algorithm

Algorithm and proof of getting optimal solution

- Already done so prepare your self

Minimum Spanning Tree

- Spanning subgraph
  - Subgraph of a graph ***G*** containing all the vertices of ***G***
- Spanning tree
  - Spanning subgraph that is itself a (free) tree
- Minimum spanning tree (MST)
  - Spanning tree of a weighted graph with minimum total edge weight
- Applications
  - Communications networks
  - Transportation networks

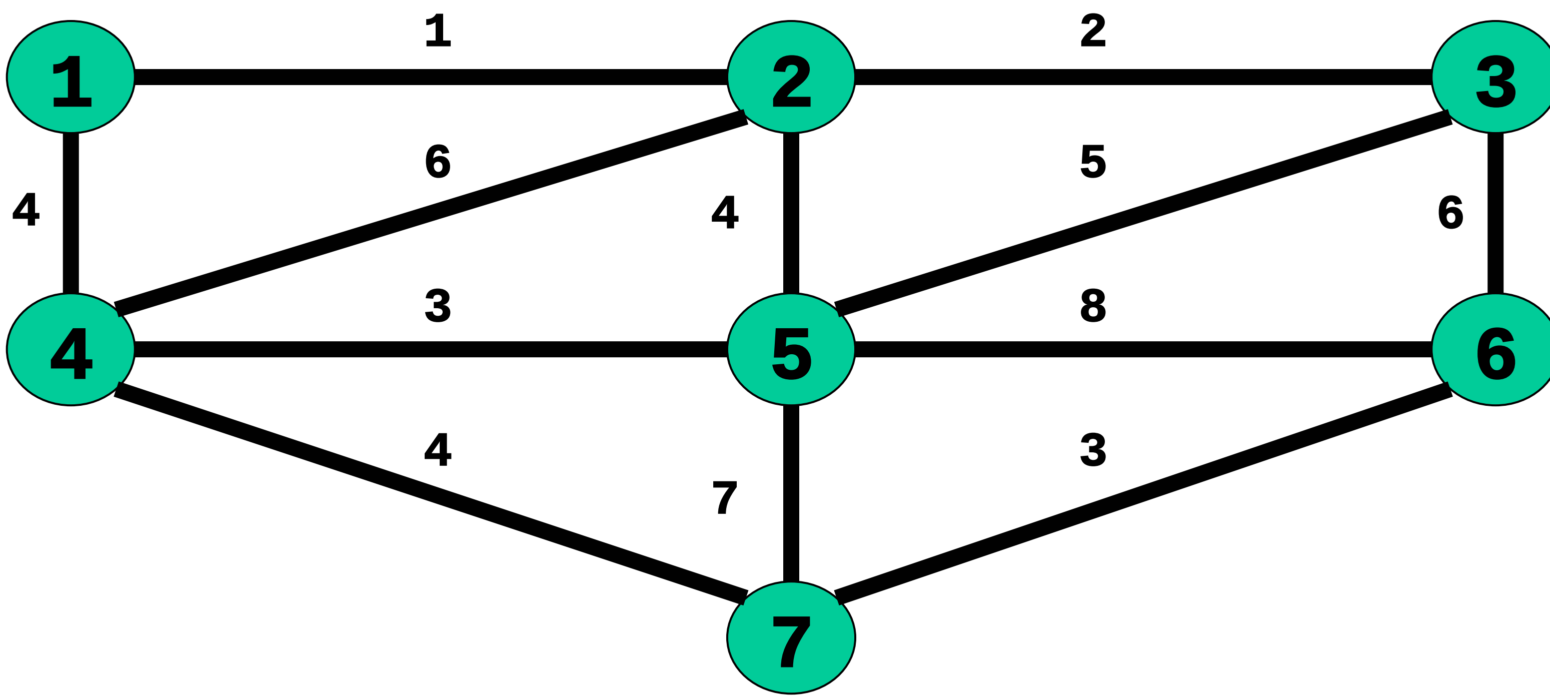


Greedy Algorithms

- We are trying to solve a problem in an optimal way.
- We start with a set of candidates
- As the algorithm proceeds, we keep two sets, one for candidates already chosen, and one for candidates rejected.
- Functions exist for choosing the next element and testing for solutions and feasibility.
- Let's identify these points for the Making Change problem.

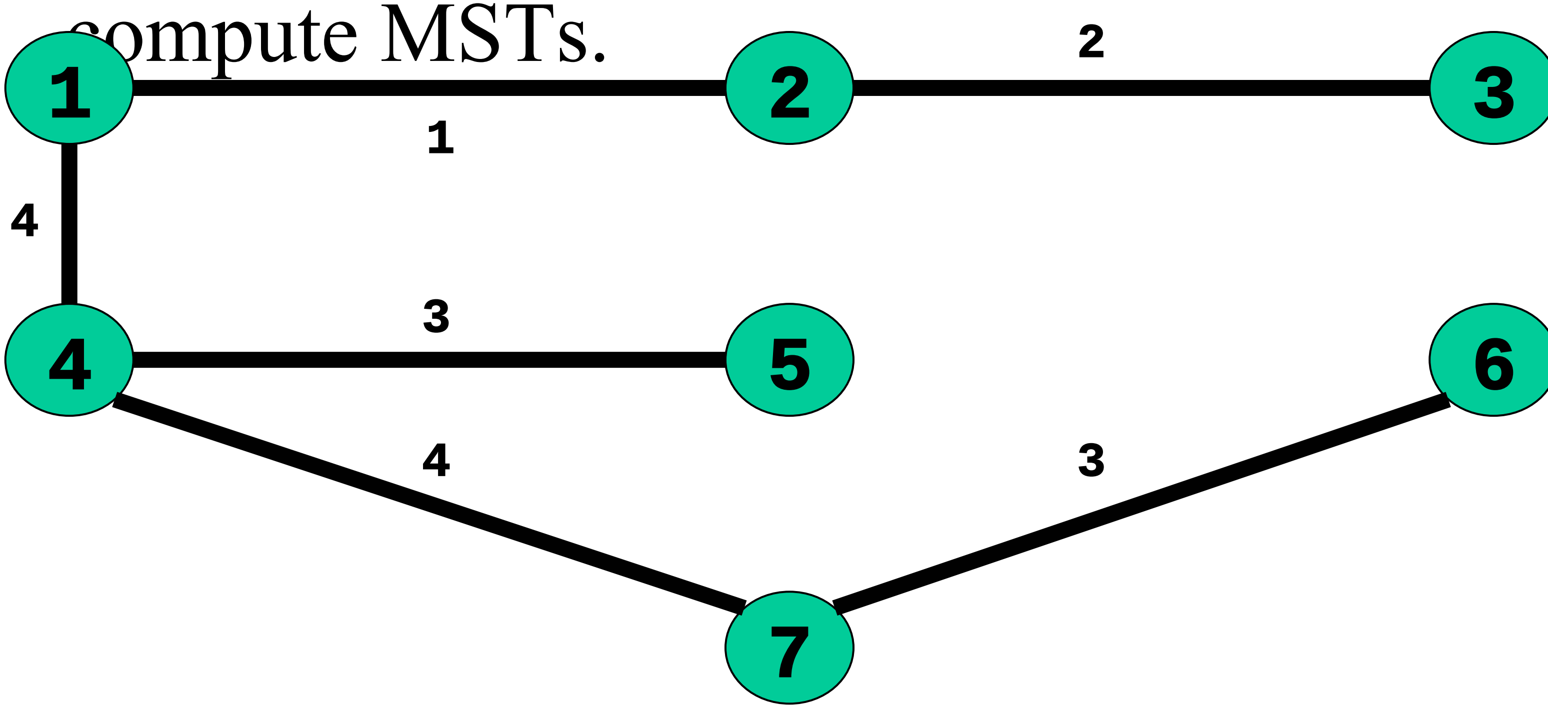
Minimum Spanning Trees

- Given G={N,A }, G is undirected and connected
- Find T, a subset of A, such that the nodes are connected and the sum of the edge weights is minimized. T is a minimum spanning tree for G.



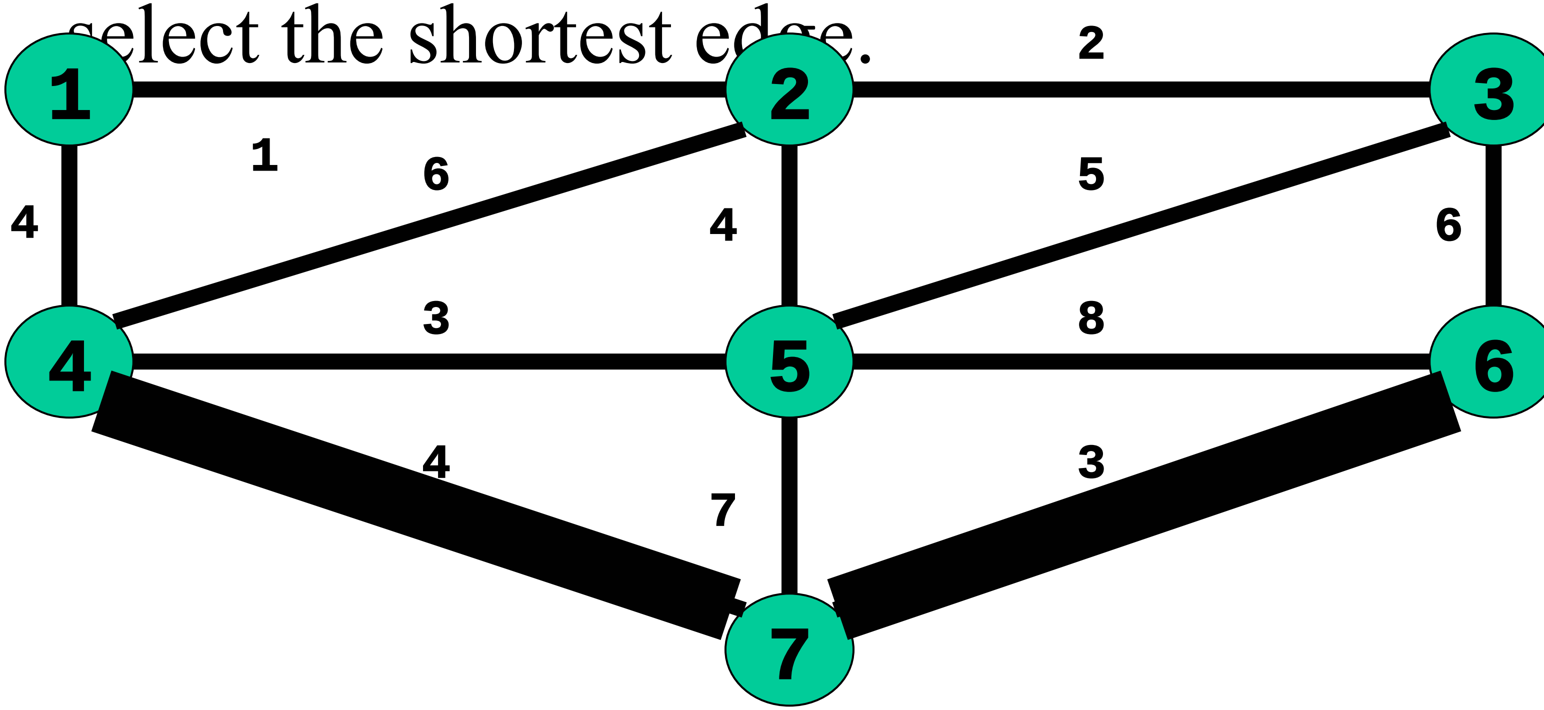
Minimum Spanning Trees

- How many edges in *T*?
- Let's come up with some algorithms to compute MSTs.



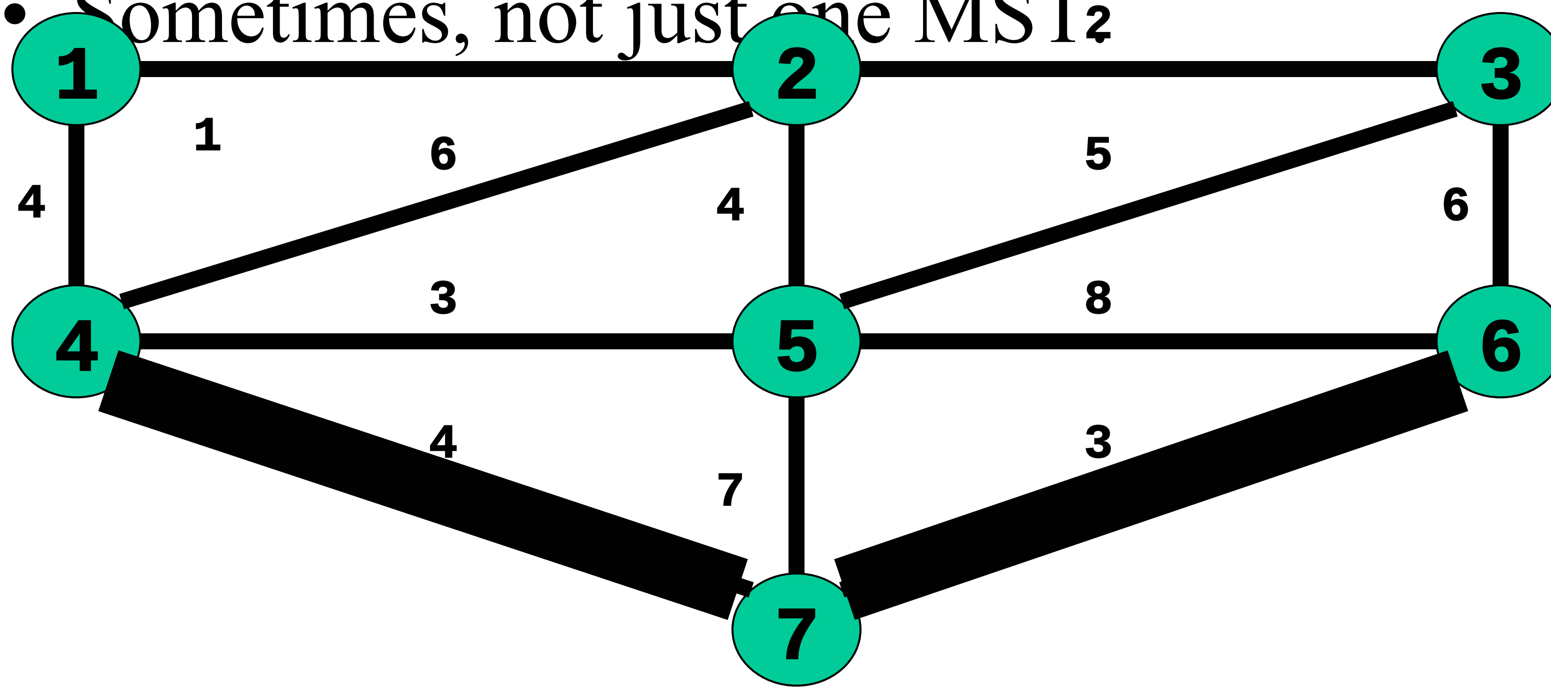
Algorithm #1

- Start with an empty set.
- From all the unchosen and unrejected edges, select the shortest edge.



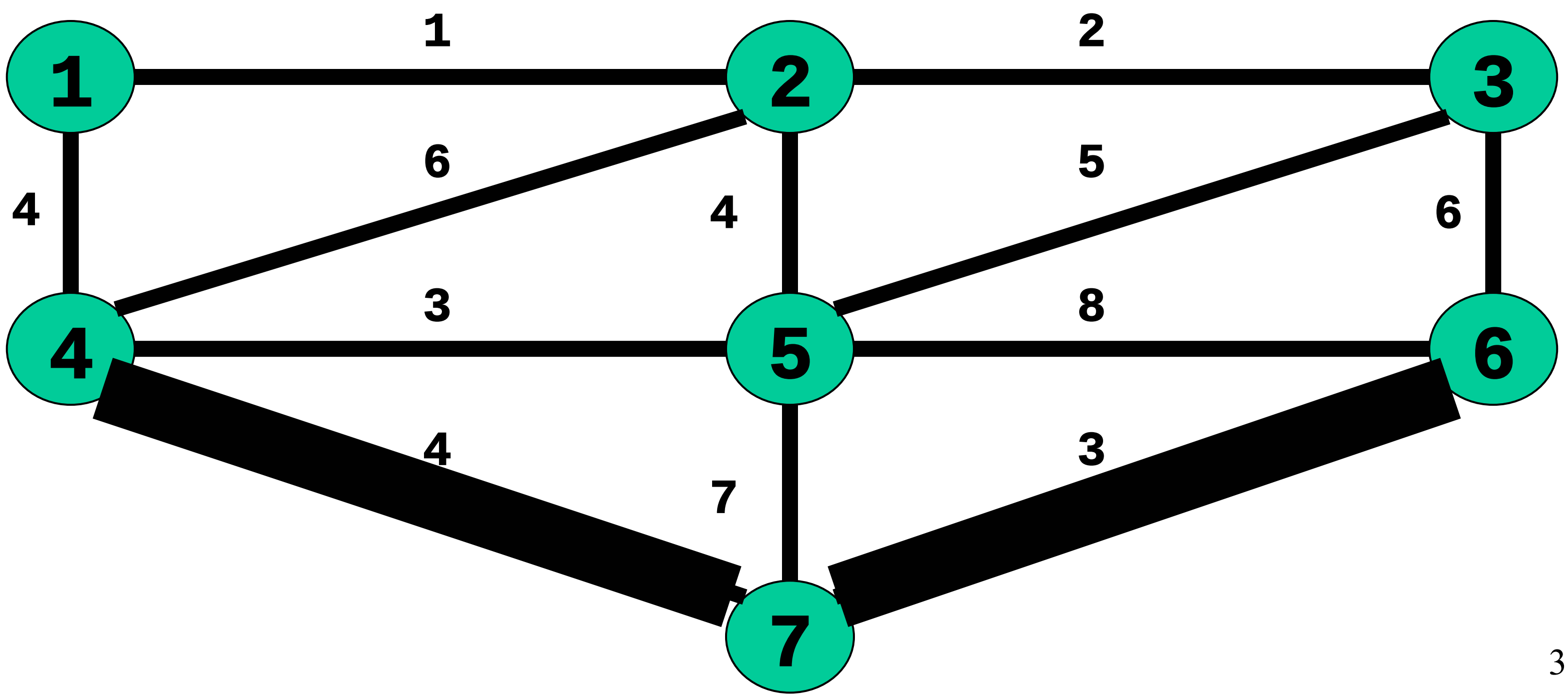
Algorithm #1

- Note we actually have multiple "correct" answers.
- Sometimes, not just one MST.



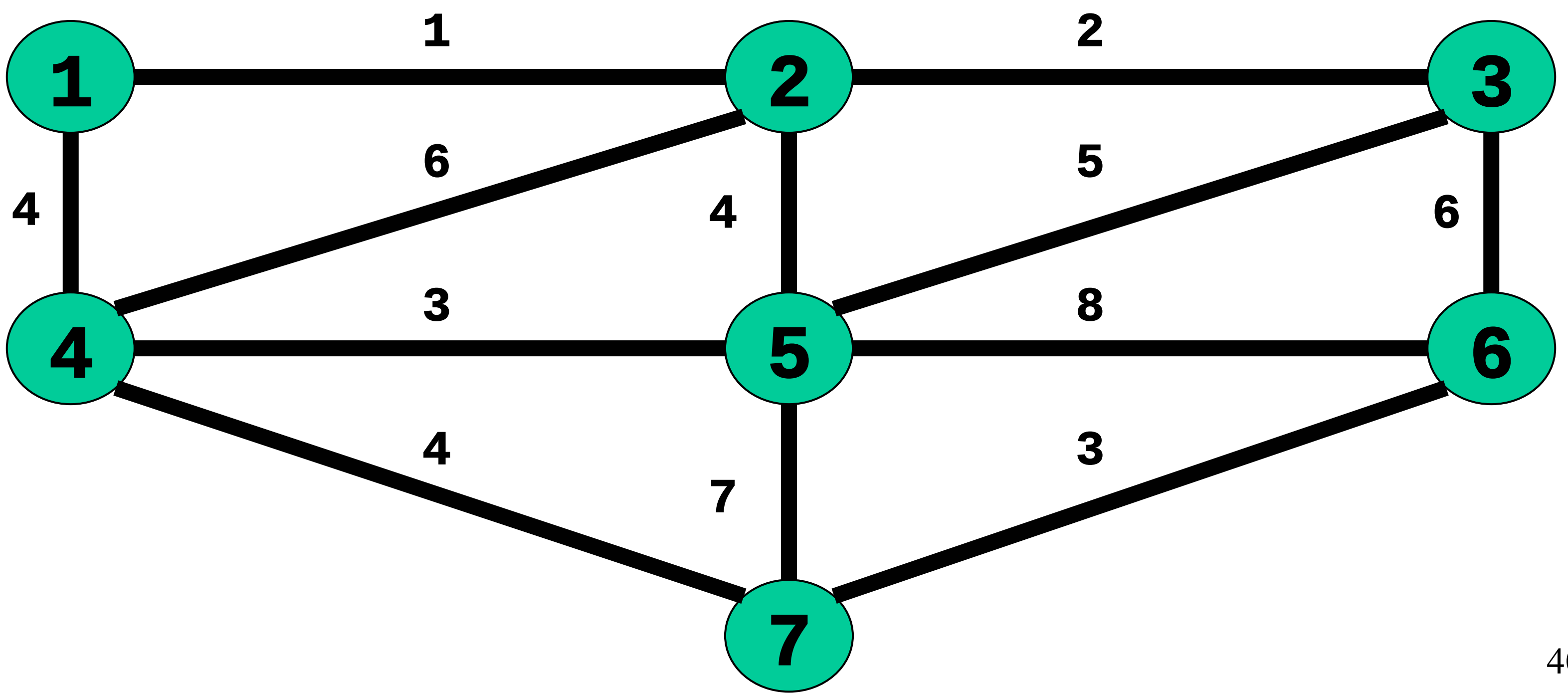
MST

- We keep two sets, one for candidates already chosen, and one for candidates rejected.
- Functions exist for choosing the next element and testing for solutions and feasibility.

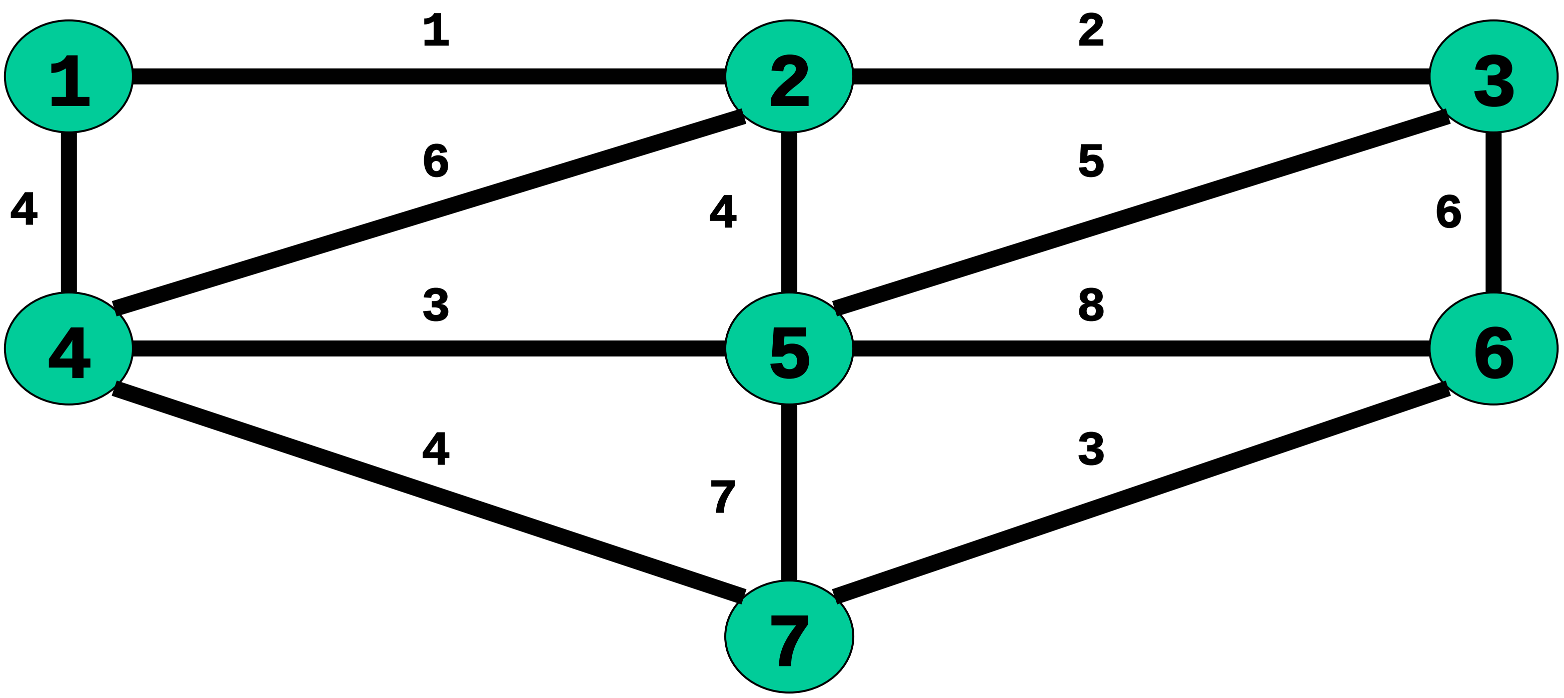


Kruskal’s Algorithm

- Each node is in its own set
- Sort edges in increasing order
- Add shortest edge that connects two sets



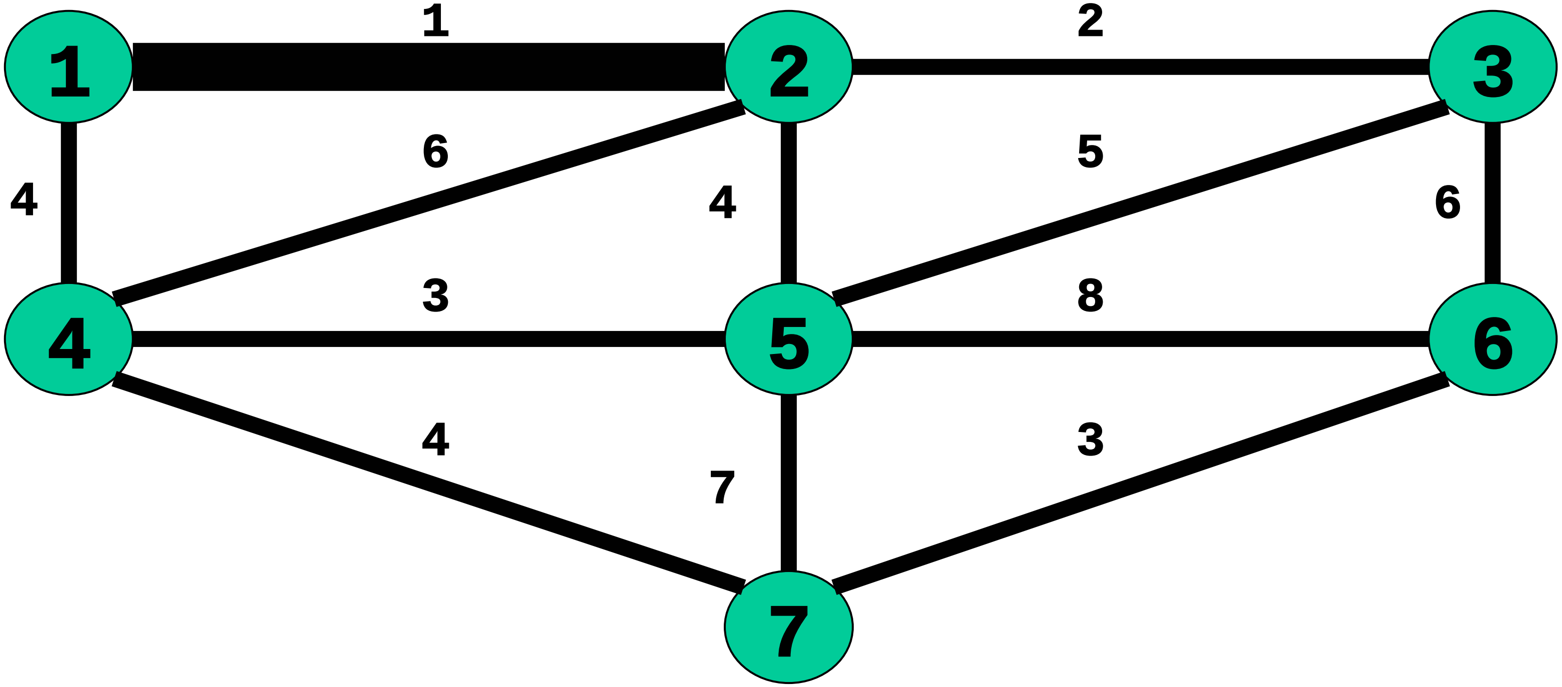
Kruskal's Algorithm



S = {1} {2} {3} {4} {5} {6} {7}

E = (1-2) (2-3) (6-7) (4-5) (1-4) (2-5) (4-7) (3-5) (2-4) (3-6) (6-7)

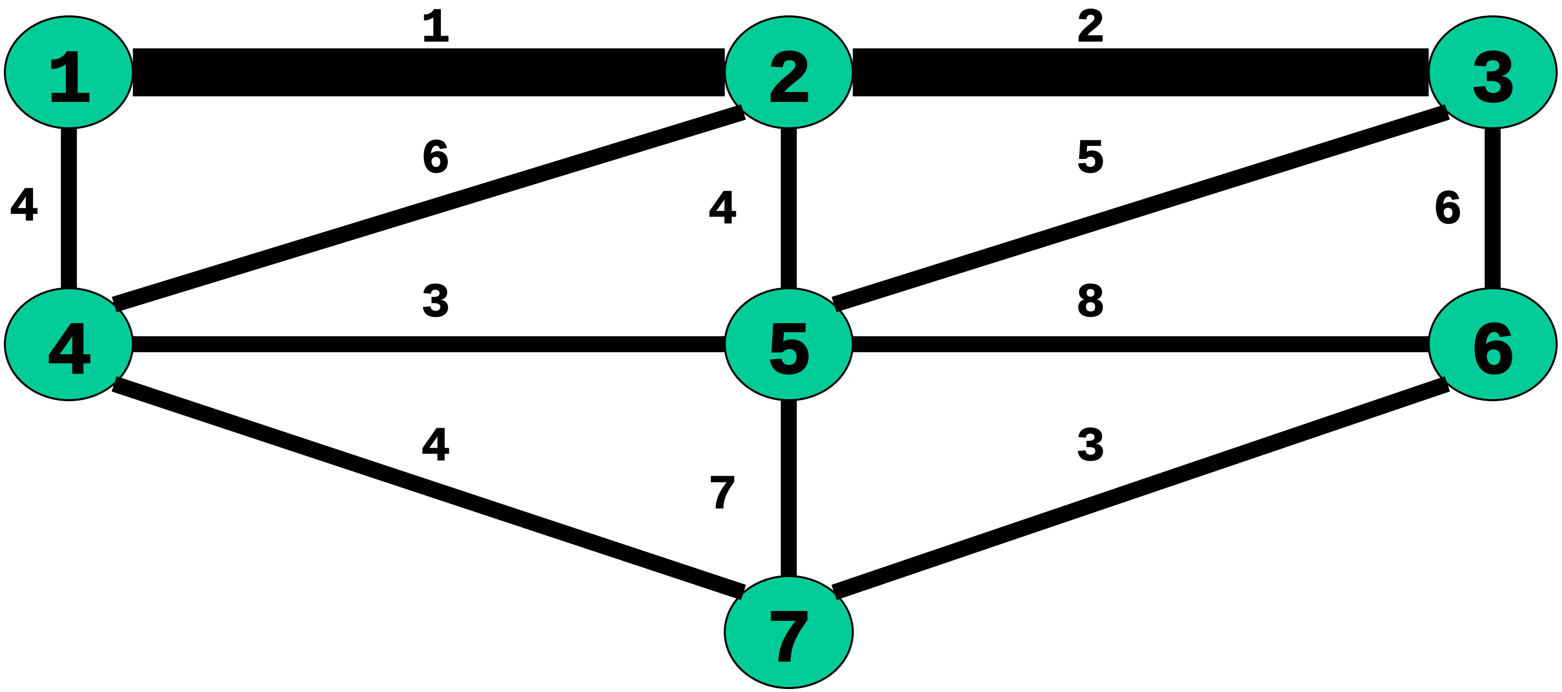
Kruskal's Algorithm



S = {1, 2} {3} {4} {5} {6} {7}

E = **(1-2)** (2-3) (6-7) (4-5) (1-4) (2-5) (4-7) (3-5) (2-4) (3-6) (6-7)

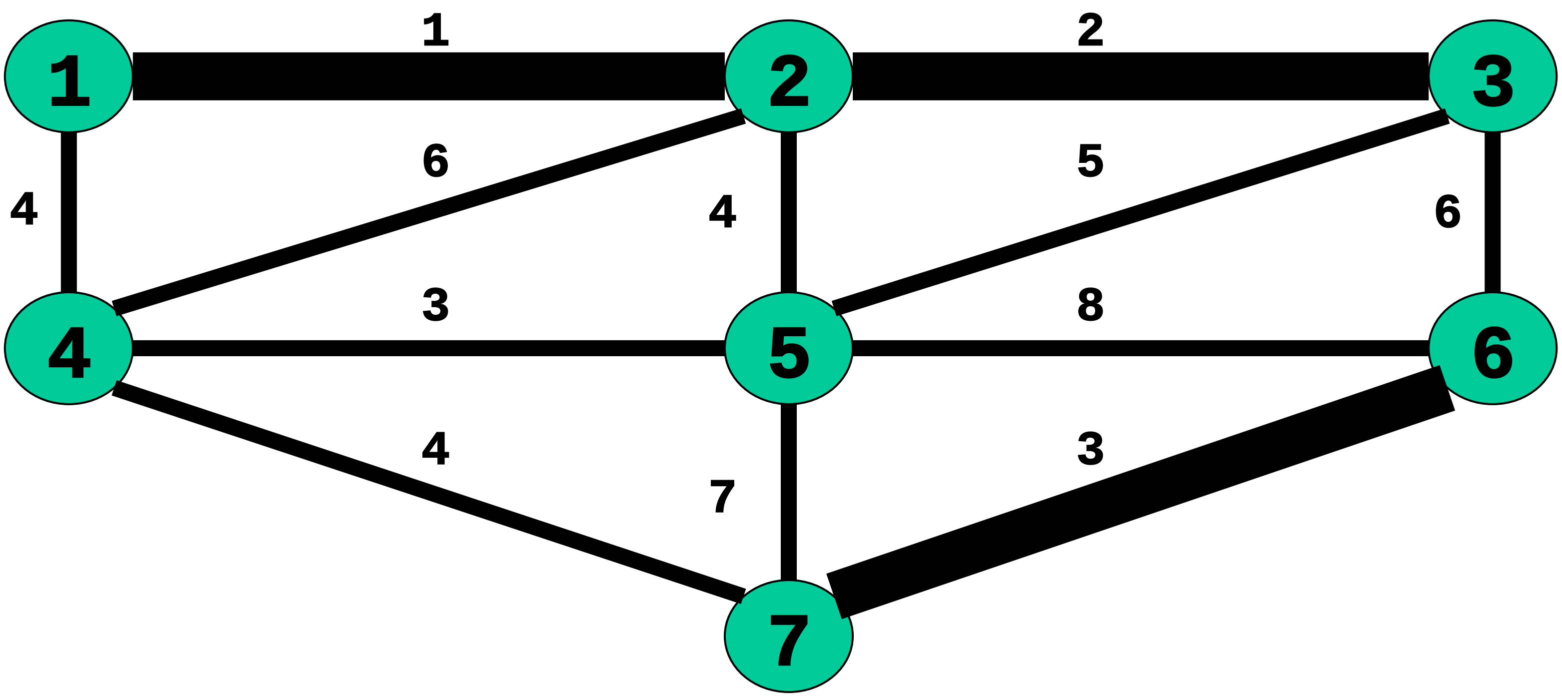
Kruskal's Algorithm



S = {1, 2, 3} {4} {5} {6} {7}

E = **(1-2) (2-3)** (6-7) (4-5) (1-4) (2-5) (4-7) (3-5) (2-4) (3-6) (6-7)

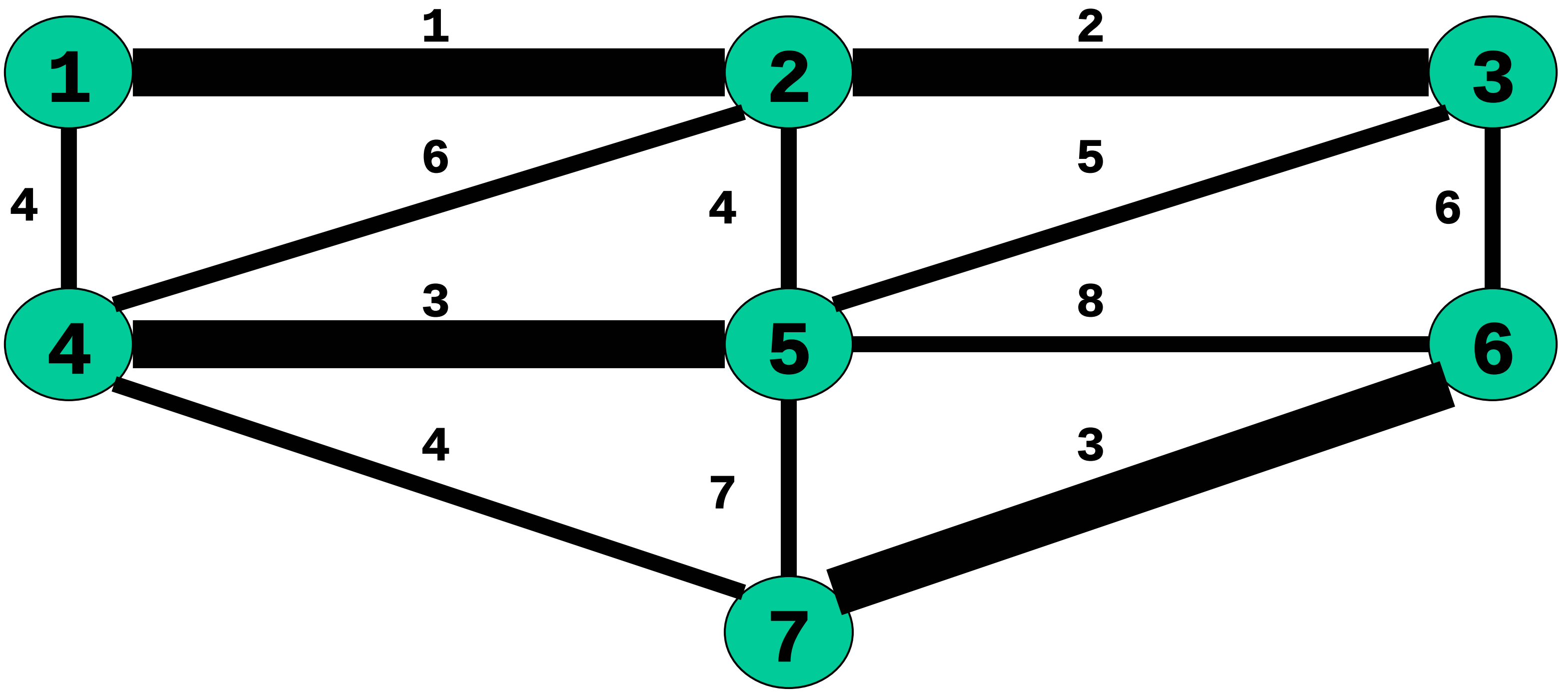
Kruskal's Algorithm



S = {1, 2, 3} {4} {5} {6, 7}

E = **(1-2) (2-3) (6-7)** (4-5) (1-4) (2-5) (4-7) (3-5) (2-4) (3-6) (6-7)

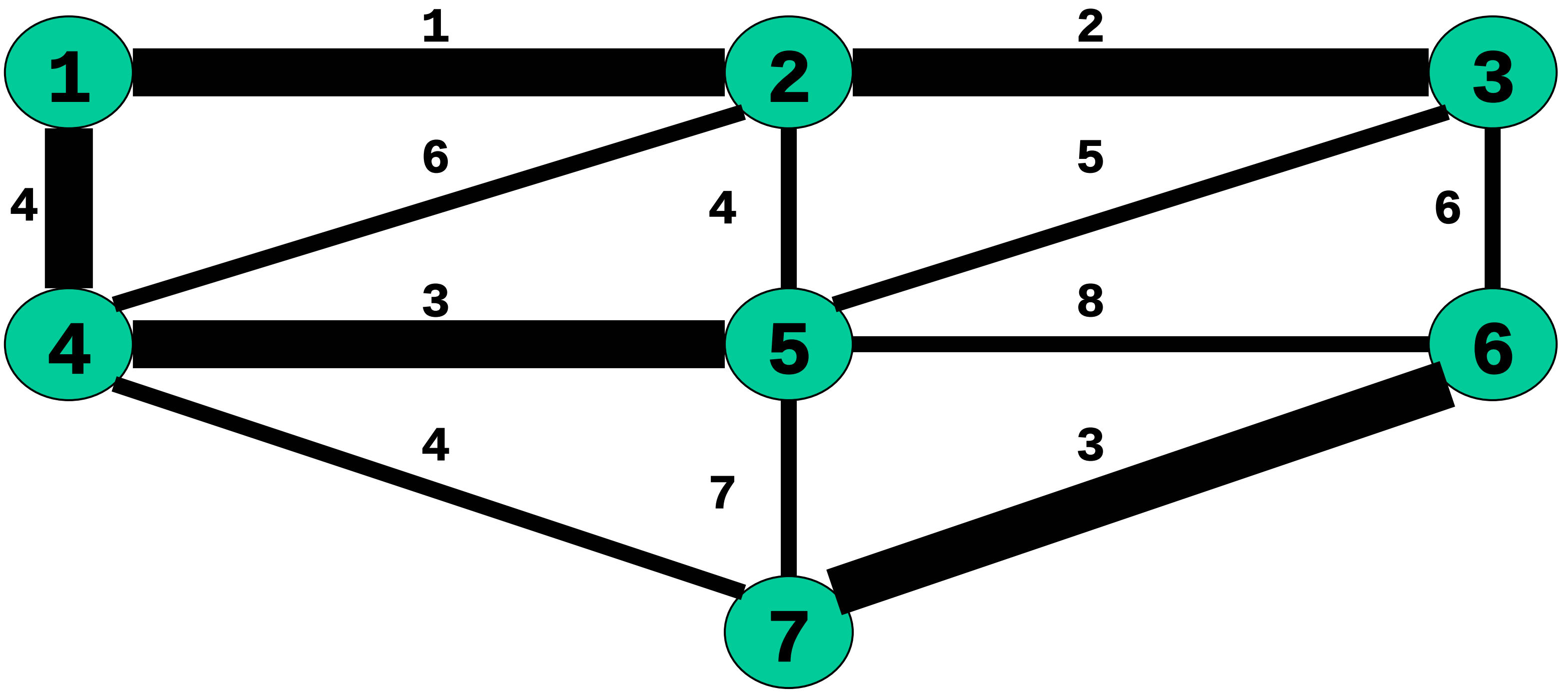
Kruskal's Algorithm



S = {1, 2, 3} {4, 5} {6, 7}

E = **(1-2) (2-3) (6-7) (4-5)** (1-4) (2-5) (4-7) (3-5) (2-4) (3-6) (6-7)

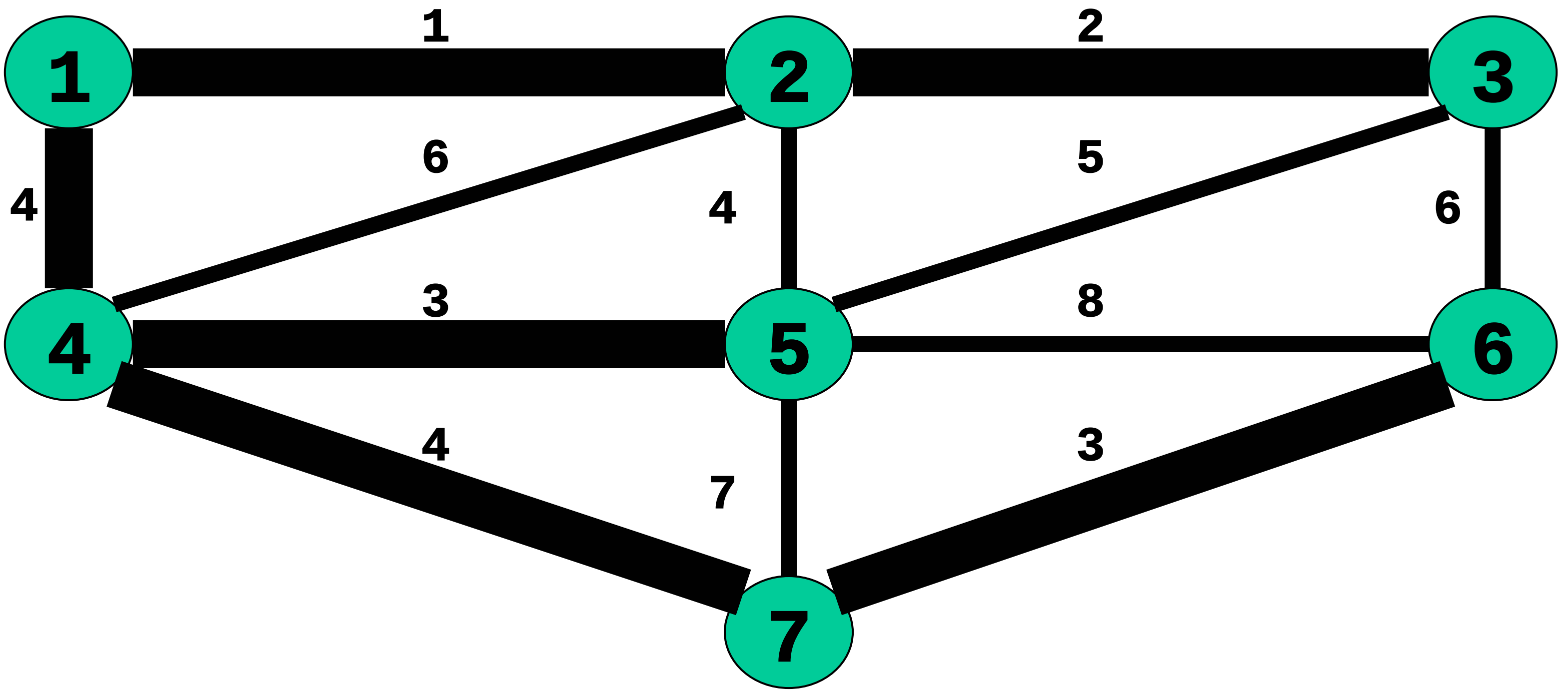
Kruskal's Algorithm



S = {1, 2, 3, 4, 5} {6, 7}

E = **(1-2) (2-3) (6-7) (4-5) (1-4)** (2-5) (4-7) (3-5) (2-4) (3-6) (6-7)

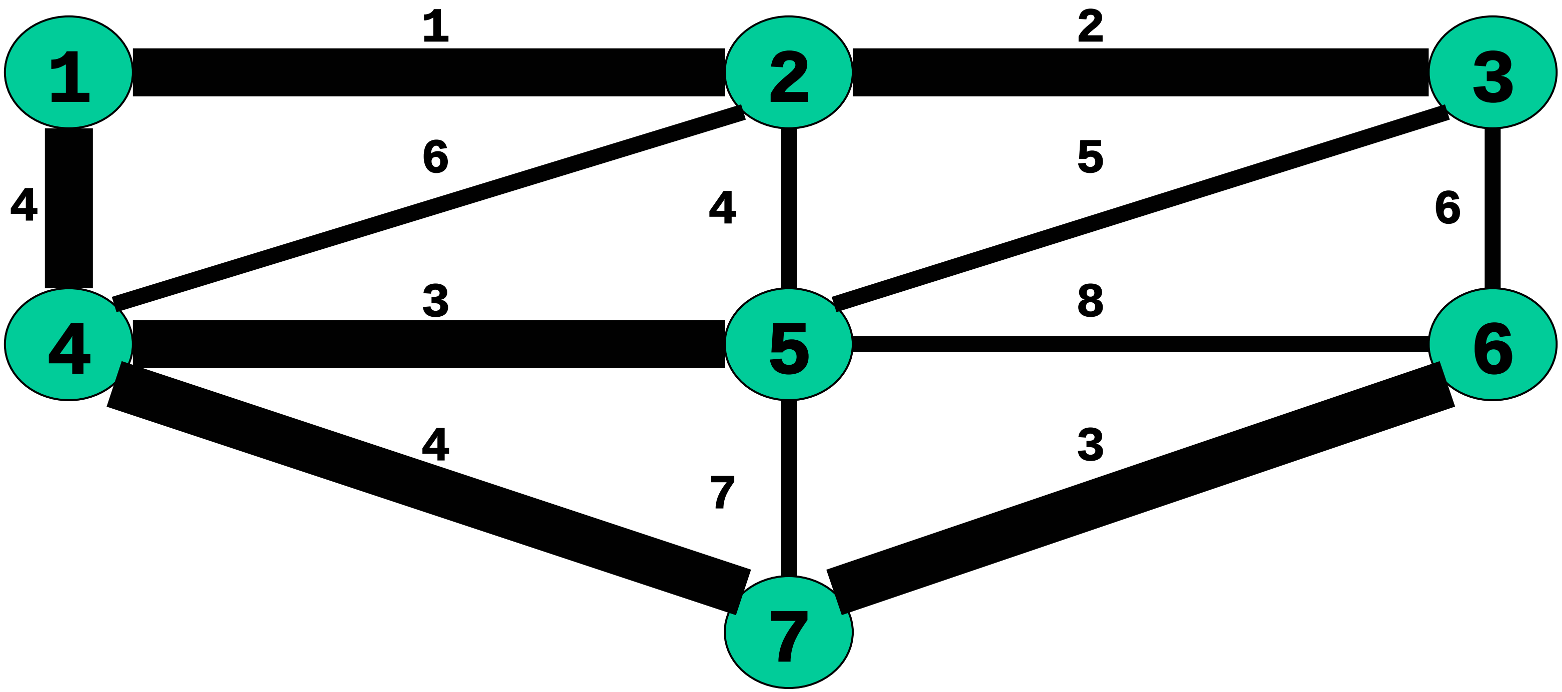
Kruskal's Algorithm



S = {1, 2, 3, 4, 5, 6, 7}

E = **(1-2) (2-3) (6-7) (4-5) (1-4)** (2-5) **(4-7)** (3-5) (2-4) (3-6) (6-7)

Kruskal's Algorithm

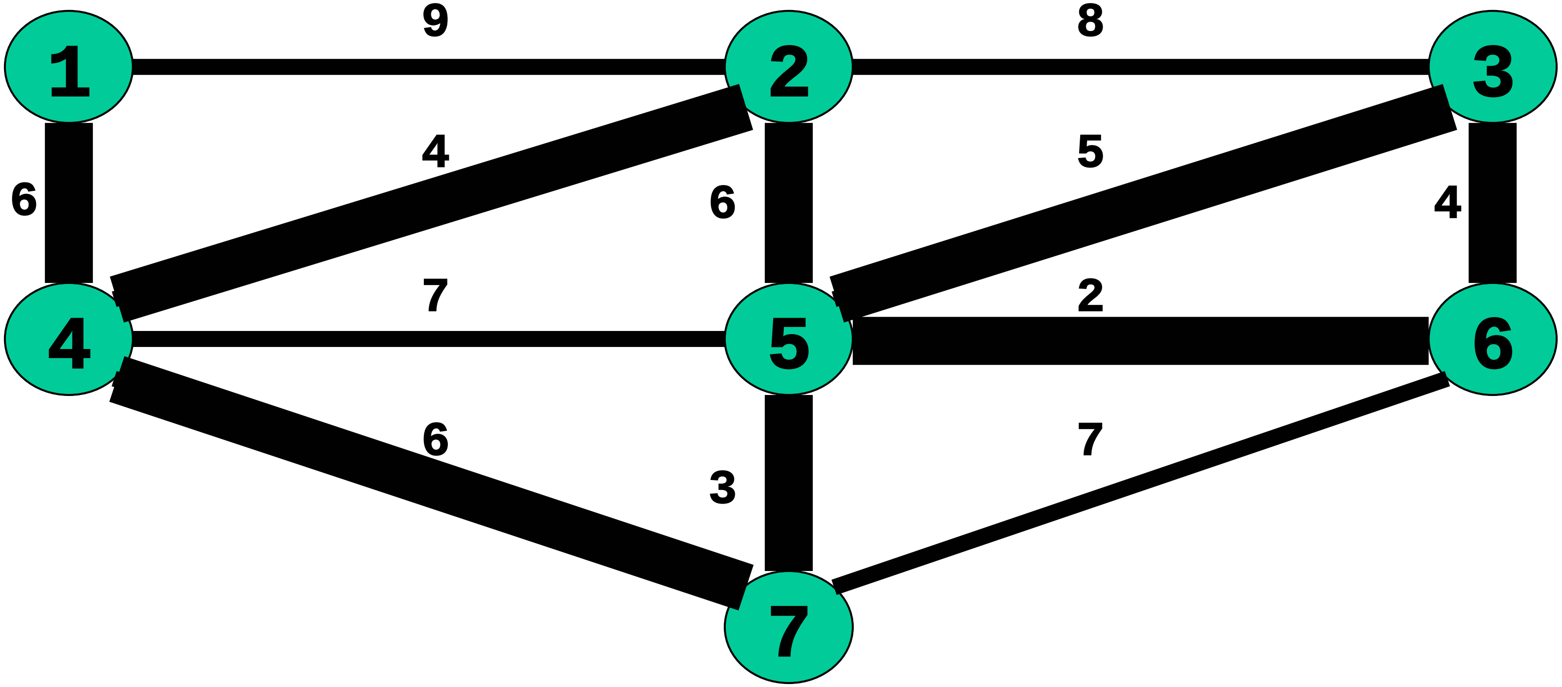


S = {1, 2, 3, 4, 5, 6, 7}

E = **(1-2) (2-3) (6-7) (4-5) (1-4) (4-7)**

Total = 1 + 2 + 3 + 3 + 4 + 4 = 17

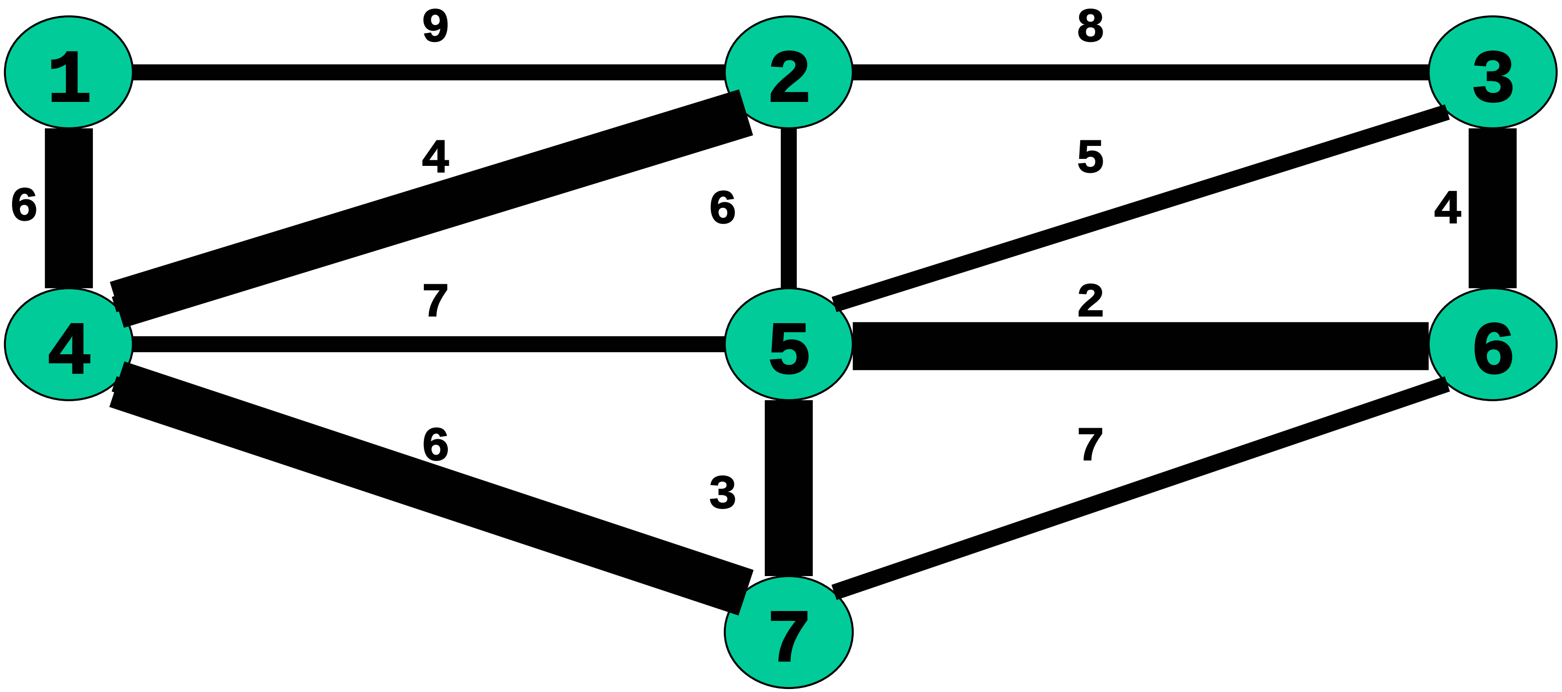
Kruskal’s Algorithm



E = (5-6), (5-7), (6-3), (4-2), (1-4), (2-5)

Total = 2 + 3 + 4 + 4 + 5 + 6 = 24

Kruskal’s Algorithm



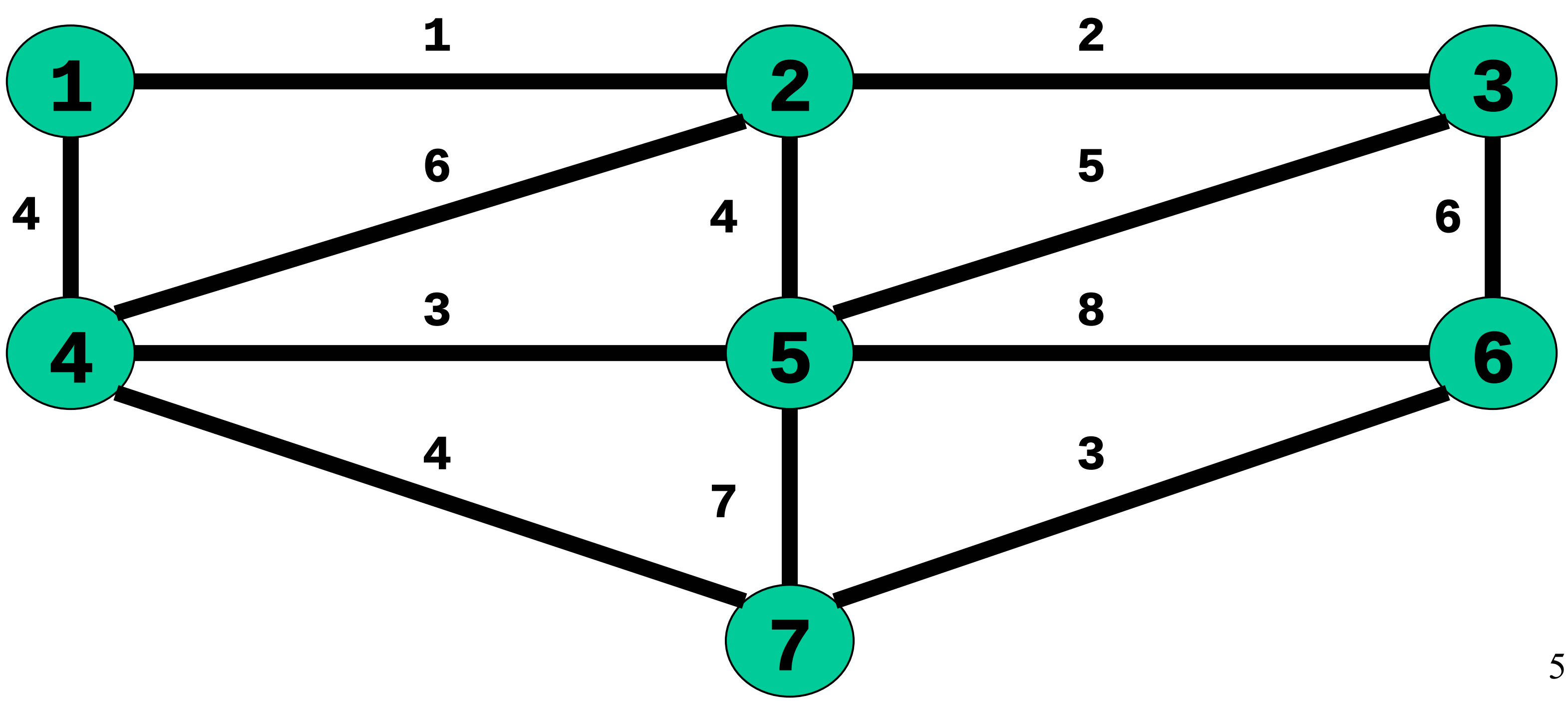
Another possible Minimum Spanning Tree:

E = (5-6), (5-7), (6-3), (4-2), (1-4), (4-7)

Total = 2 + 3 + 4 + 4 + 5 + 6 = 24

Timing Kruskal's Algorithm

- Each node is in its own set
- Sort edges in increasing order
- Add shortest edge that connects two sets

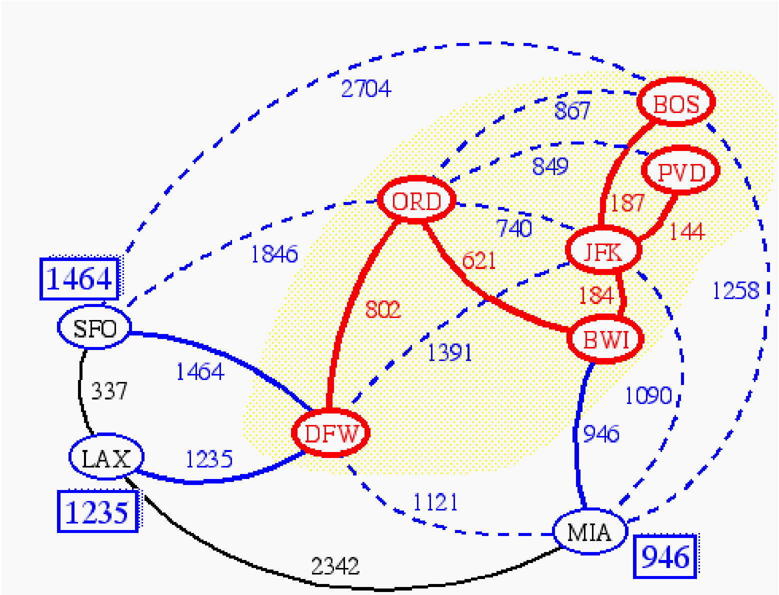


Prim-Jarnik's Algorithm

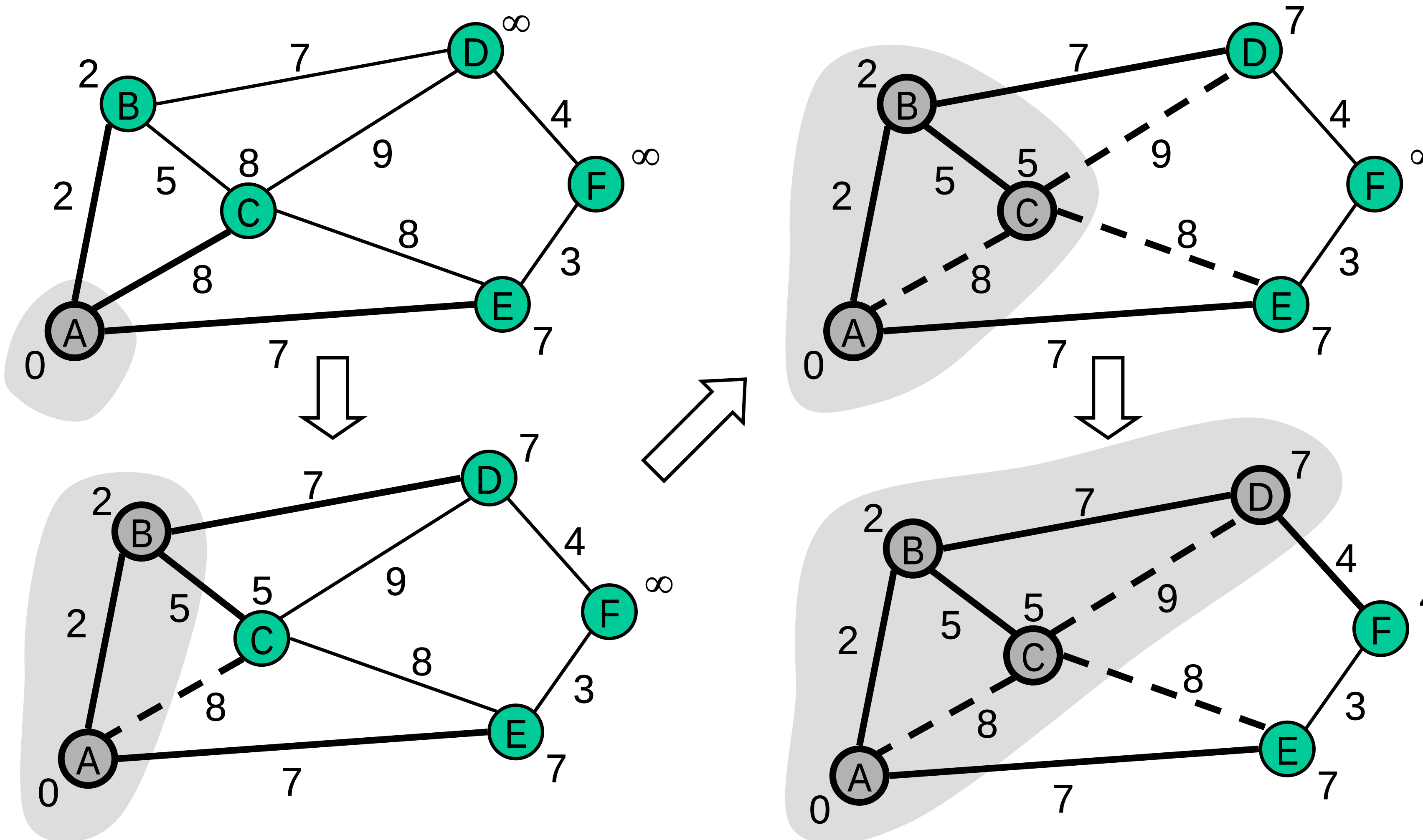
- We pick an arbitrary vertex ***s*** and we grow the MST as a cloud of vertices, starting from ***s***
- We store with each vertex ***v*** a label ***d***(***v***) = the smallest weight of an edge connecting ***v*** to a vertex in the cloud

◆ At each step:

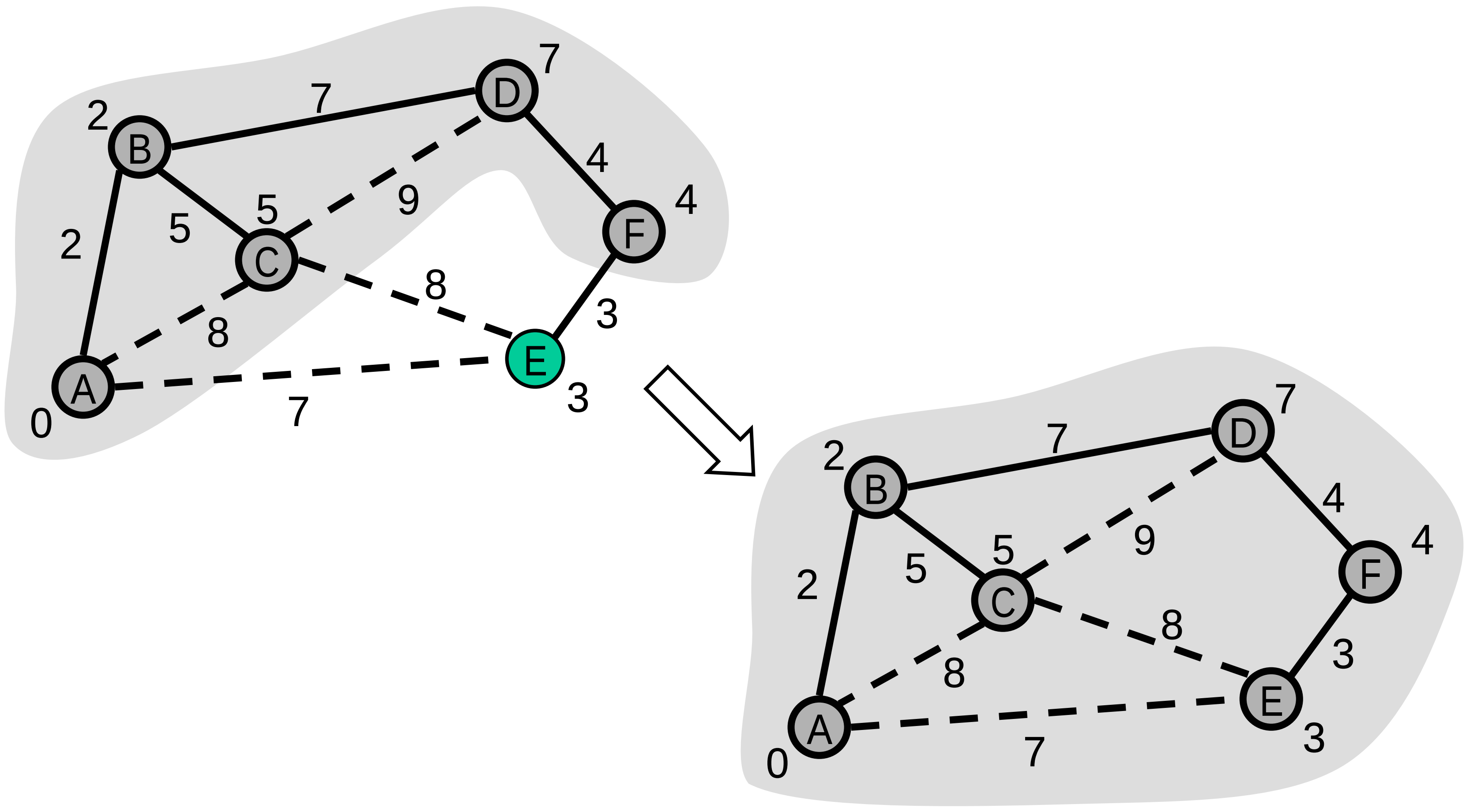
- We add to the cloud the vertex ***u*** outside the cloud with the smallest distance label
- We update the labels of the vertices adjacent to ***u***



Example



Example (contd.)



Huffman Codes

- Very effective technique for compressing data; saving of 20% to 90%.
- It uses a table of frequencies of occurrence of characters.

Fixed length code

- If only six different characters are used in a text then we need 3 bits to represent six characters.
- a = 000; b = 001; …f = 101.
- This method thus requires 300,000 bits to code the entire file having 100,000 characters.

Variable Length code

We can do considerable better by giving frequent characters short code words and infrequent characters log code words.

frequency	45	13	12	16	9	5
Fixed len codeword	000	001	010	011	100	101
Var. length code	0	101	100	111	1101	1100

Thus using variable length code it requires

(45.1 +13.3 12.3 16.3 9.4 5.4)1000

= 224,000 Bits

Thus saving approx 25%.

Prefix code: no code word is also a prefix of some other codeword.

These prefix codes are desirable because they simplify decoding.

Like in our example

001011101 can be parsed uniquely as

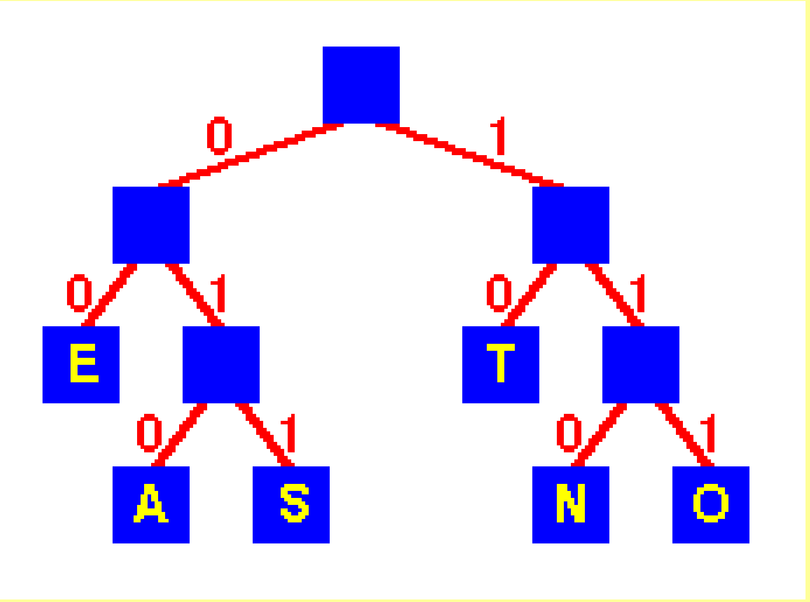
0-0-101-1101 which decodes as aabe

Huffman Encoding

- Compression
  - Typically, in files and messages,
    - Each character requires 1 byte or 8 bits
    - Already wasting 1 bit for most purposes!
- Question
  - What’s the smallest number of bits that can be used to store an arbitrary piece of text?
- Idea
  - Find the frequency of occurrence of each character
  - Encode Frequent characters short bit strings
  - Rarer characters longer bit strings

Huffman Encoding

- Encoding
  - Use a tree
  - Encode by following tree from root to leaf
  - eg
    - E is 00
    - S is 011
  - Frequent characters
    E, T 2 bit encodings
  - Others
    A, S, N, O 3 bit encodings



Huffman Encoding

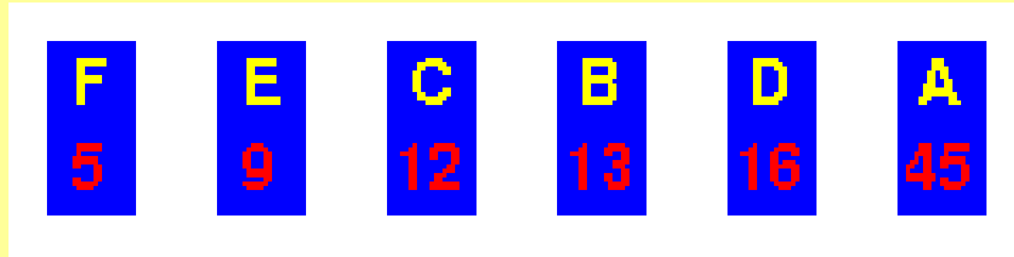
- Encoding
  - Use a tree
    - Inefficient in practice
  - Use a direct-addressed lookup table

A	010
B	
:	
E	00
:	
N	110
:	
S	001
T	10

? Finding the optimal encoding
  - Smallest number of bits to represent arbitrary text

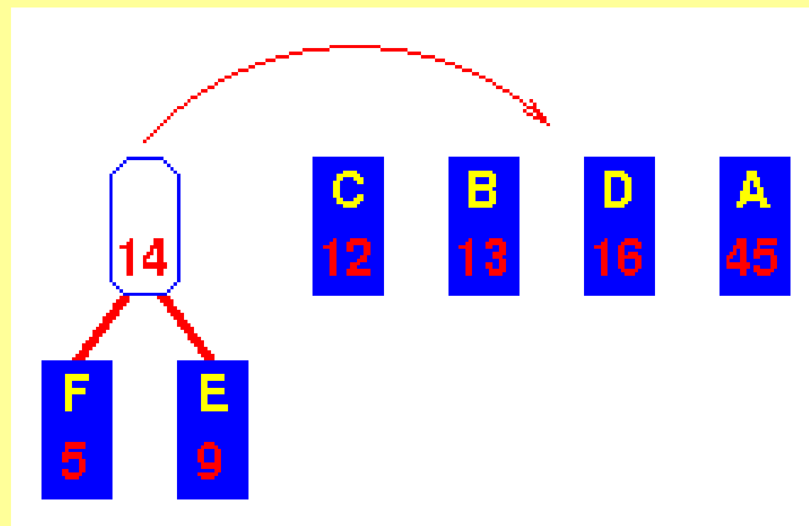
Huffman Encoding - Operation

Initial sequence
Sorted by frequency



Combine lowest two into sub-tree

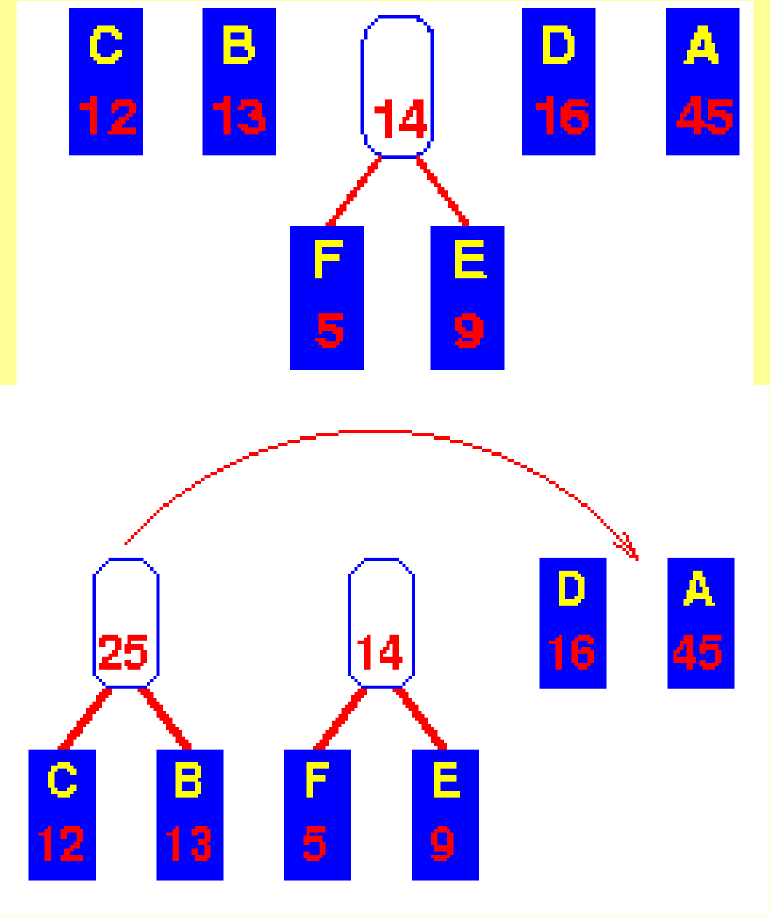
Move it to correct place



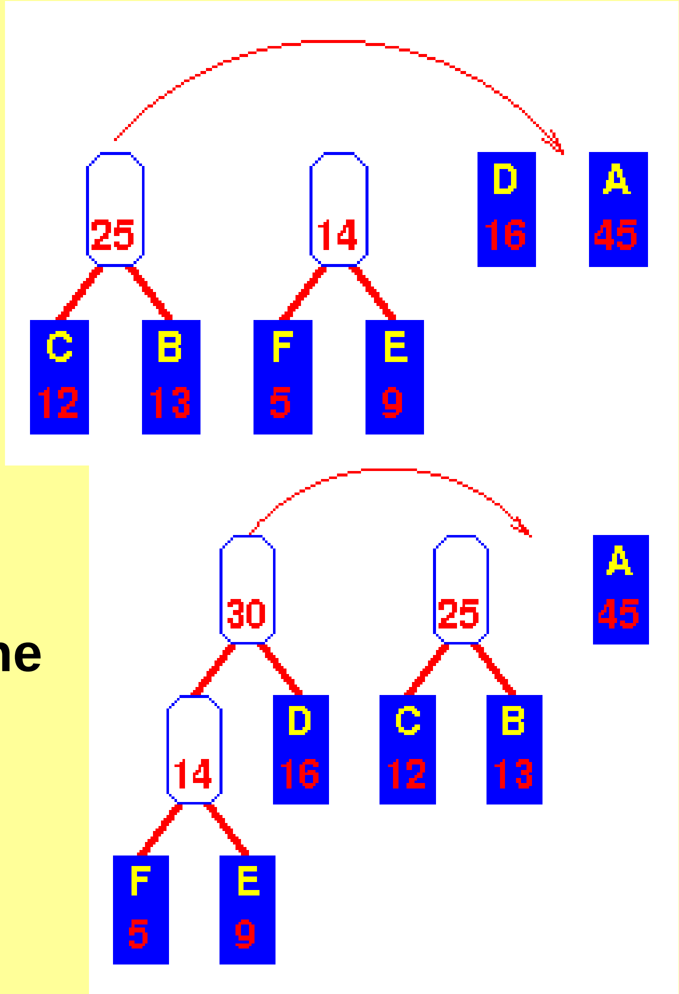
After shifting sub-tree to its correct place ...

Combine next lowest pair

Move sub-tree to correct place

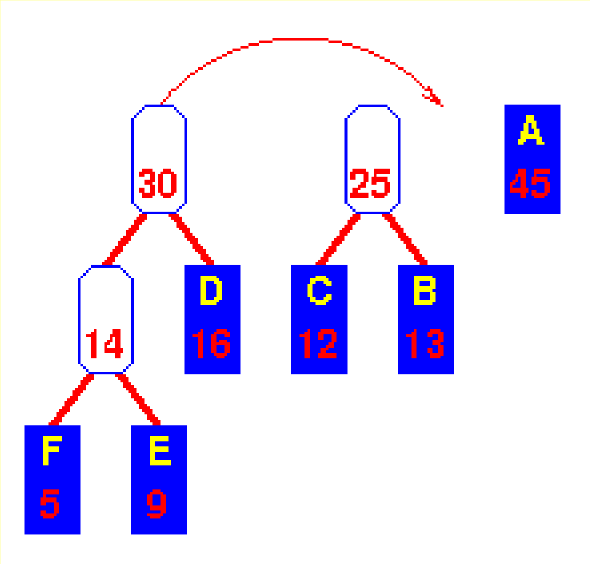


Move the new tree to the correct place ...

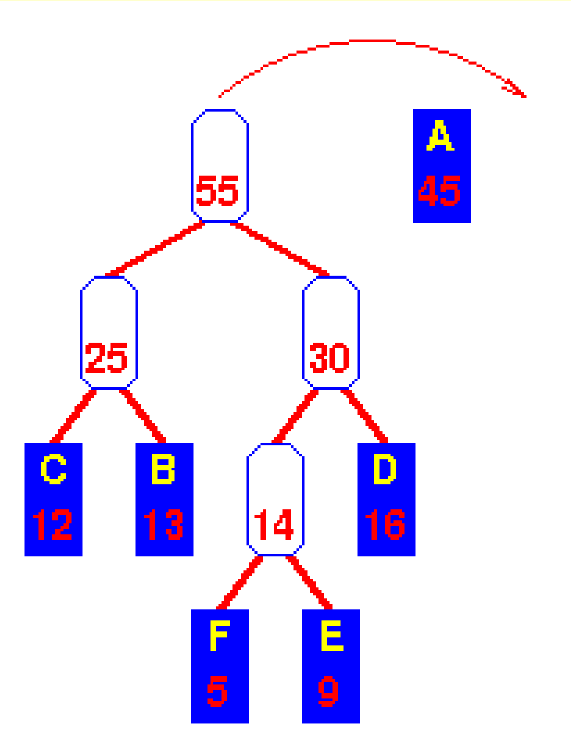


Now the lowest two are the "14" sub-tree and D

Combine and move to correct place

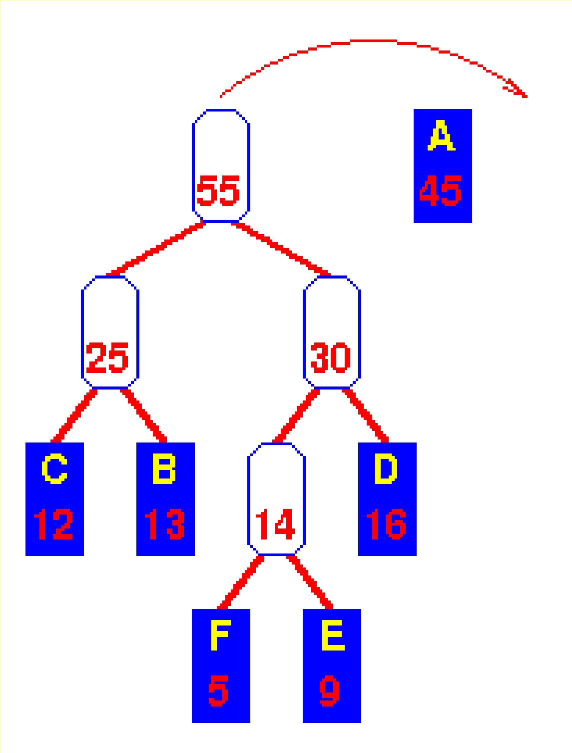


Move the new tree to the correct place ...

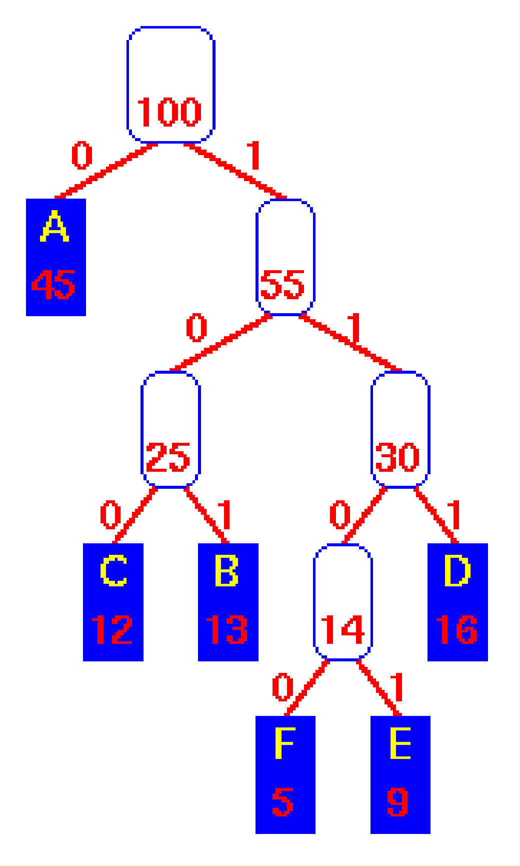


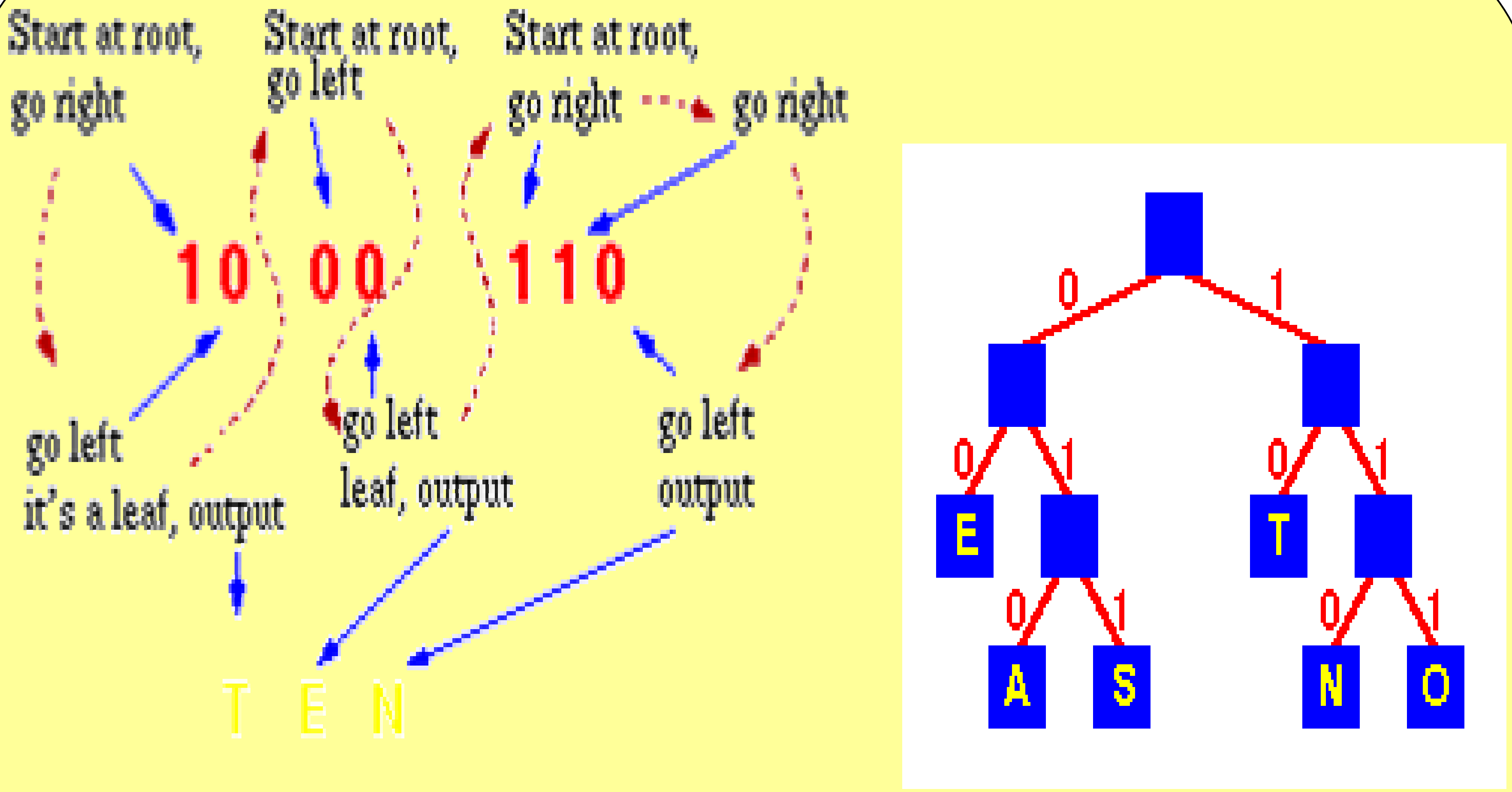
Now the lowest two are the the “25” and “30” trees

Combine and move to correct place



Combine last two trees





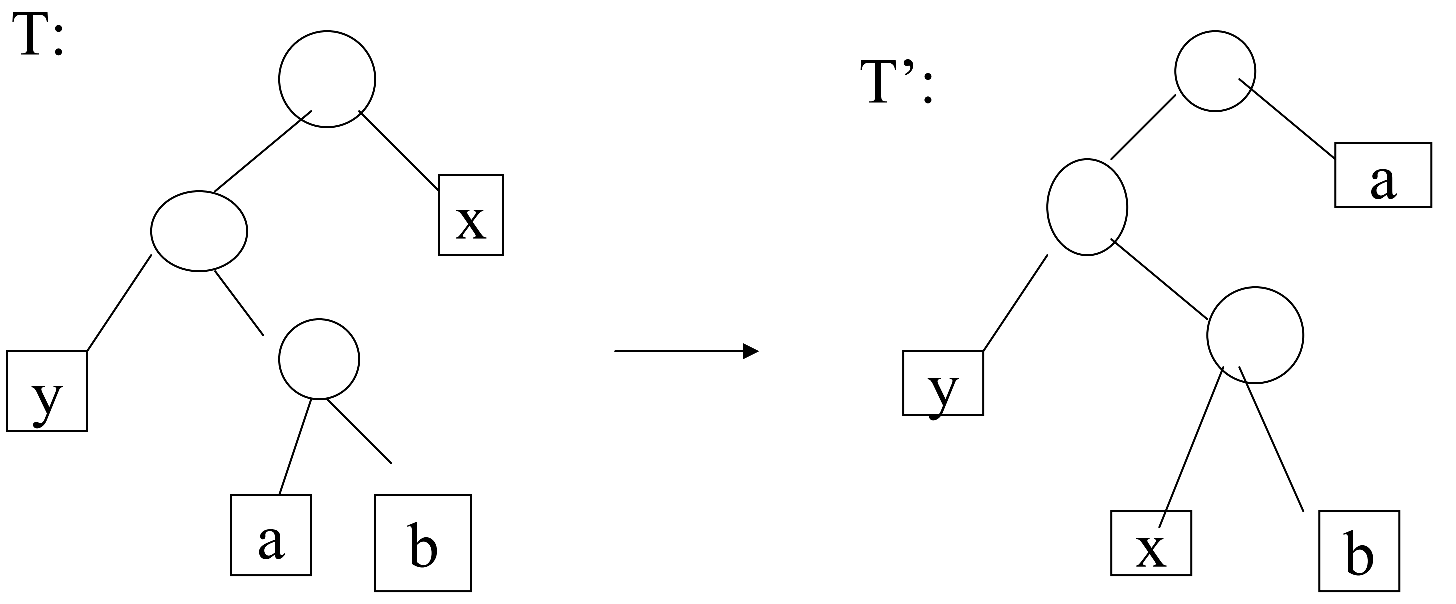
Start at root,
go right
Start at root,
go left
Start at root,
go right
go right
10
00
110
go left
it's a leaf, output
go left
leaf, output
go left
output
T
E
N
0
1
0
1
0
1
E
T
0
1
0
1
A
S
N
O

Huffman Encoding - Time Complexity

- Sort keys $O(n \log n)$
- Repeat n times
  - Form new sub-tree $O(1)$
  - Move sub-tree (binary search) $O(\log n)$
  - Total $O(n \log n)$
- Overall $O(n \log n)$

Theorem

- Let C be an alphabet in which each character c of C has frequency f(c) .
- Let x and y be two characters in C having lowest frequencies.
- Then there exists *an optimal prefix code* for C in which the code words for x and y have the same length and differ only in the last bit.



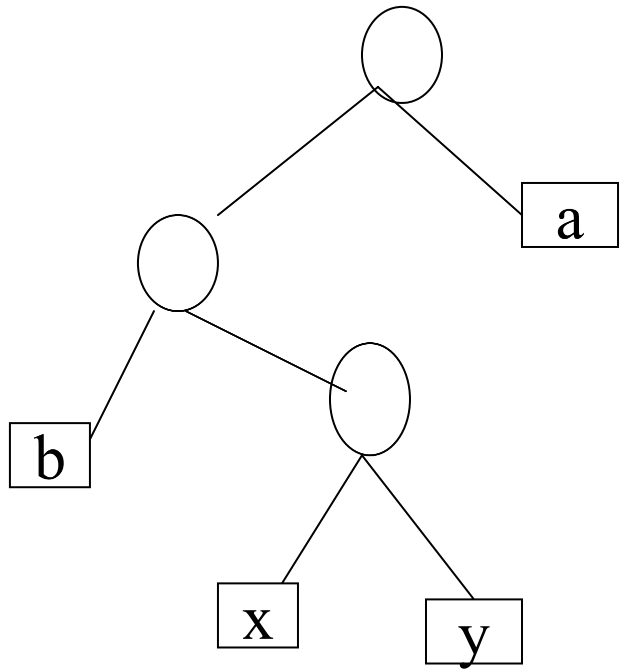
Let a and b be two characters that are sibling leaves of maximum depth in T.

We can assume $f(a) \leq f(b)$ and $f(x) \leq f(y)$

Since x and y are of lowest frequencies

$f(x) \leq f(a)$ and $f(y) \leq f(b)$

T'':



- The number of bits required to encode a file is
- $B(T) = \sum f(c)\, d_T(c)$
- we can call it the cost of the tree T.
- $B(T) – B(T') = \sum f(c)\, d_T(c) - \sum f(c)\, d_{T'}(c)$

$= f(x)\, d_T(x) + f(a)\, d_T(a) - f(x)\, d_T'(x) – f(a)\, d_T'(a)$

$= f(x)\, d_T(x) + f(a)\, d_T(a) - f(x)\, d_T(a) – f(a)\, d_T(x)$

$=(f(a) – f(x))(d_T(a) – d_T(x))$

$>= 0$ because both

$f(a)-f(x)$ and $d_T(a) – d_T(x)$ are nonnegative.

- Similarly exchanging y and b does not increase the cost so

$B(T') - B(T'')$ is non negative.

There fore

$B(T'') \leq B(T)$

- And since T was taken to be optimal
- $B(T) \leq B(T'')$
- Which implies
- $B(T'') = B(T)$
- Thus T'' is an optimal tree in which x and y appear as sibling leaves of maximum depth.

Theorem - Let C be a given alphabet with frequency f(c). Let x and y be two characters in C with minimum frequencies. Let C' be the alphabet obtained from C as

$C' = C - \{x,y\} \cup \{z\}$.

Frequencies for new set is same as for C except that $f(z) = f(x) + f(y)$.

Let T' be any optimal tree representing optimal code for C', then the tree T obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C

Proof: For any $c \,\varepsilon\, C - \{x, y\}$

$d_T(c) = d_T'(c)$ but for x and y

$d_T(x) = d_T(y) = d_T'(z) + 1$

We have

$f(x)d_T(x) + f(y)\, d_T(y) = (f(x) + f(y))(d_T(z) + 1)$

$= f(z)\, d_T(z) + (f(x) + f(y))$

$B(T) = B(T') + f(x) + f(y)$

From which we conclude that

$B(T) = B(T') + f(x) + f(y)$

Or

$B(T') = B(T) - f(x) - f(y)$

We now prove by contradiction.

Suppose that T is not optimal for C then there is another optimal tree T'' such that

$B(T'') < B(T)$

Without any loss of generality we can assume that x and y are siblings here.

Let T''' be the tree obtained from T''with the common parent of x and y replaced by a leaf z with freq $f(z) = f(x) + f(y)$

then

- $B(T''') = B(T'') - f(x) - f(y)$

  $< B(T) - f(x) - f(y)$

  $= B(T')$

This contradicts the assumption that T' represents an optimal prefix code for C'

Thus T must represent an optimal prefix code for the alphabet C.

Job Sequencing with dead lines

We are given n jobs,

associated with each job I,here is an (integer) deadline $d_i \geq 0$ and a profit $p_i > 0$.

This profit pi will be earned only if job is completed before its deadline.

Each job needs processing for one unit time on a single machine available.

Feasible solution-is a subset of jobs each of which can be completed without crossing any deadline.

Optimal solution is a feasible solution with maximum profit

Example: n=4 , p=(100,10,15,27) and d=(2,1,2,1)

Possible feasible solutions:

Subset	sequence	value
{1,2}	2,1	110
{1,3}	1,3 or 3,1	115
{1,4}	4,1	127
{2,3}	2,3	25
{3,4}	4,3	42
{1}	1	100

Greedy approach

objective is to maximize $\sum pi$

So next job to select in J is the one that increases

$\sum pi$ the most, subject to the restriction that set J remains feasible.

Feasibility test becomes very easy through the following result

Theorem: Let J be a set of k jobs and

$S = i_1, i_2, \ldots. i_k$ a permutation of jobs in J such that $d_{i1} \leq d_{i2} \leq \quad \leq d_{ik}$. Then J is feasible iff the jobs in J can be processed in the order S without violating any deadline.

Proof:

We need to show that if J is feasible set then a permutation $S = i_1, i_2 \ldots i_k$ with $d_{i1} \leq d_{i2} \leq d_{ik}$ is also feasible.

If J is taken to be a feasible set then there exists S` = $r_1, r_2 \ldots. r_k$ such that $d_{rq} \geq q, \; 1 \leq q \leq k$.

Assume $S \neq$ S`,

Let a be the least index such that $r_a \neq i_a$.

Let $i_a = r_b$ for some b > a

In S' we can interchange r_a and r_b .since d_{ra} d_{rb} the resulting permutation s''

S: i_1 i_2 . . . ia … i_k

S': r_1 r_2 r_a r_b r_k

(b)

$d_{ia} <= d_{ra}$

Since S is a permutation of jobs with nondecreasing deadlines.and r_a is positioned later than i_a

If we interchange r_a and r_b we get another feasible sequence. And new sequence is closer to S.

b

Continuing in this way S` can be transformed into S without violating any dead line.

Theorem: greedy approach always obtains an optimal solution for job sequencing with dead lines

Proof: Let I be the set of jobs obtained by greedy method.

And let J be the set of jobs in an optimal solution.

We will show that I and J both have same profit values.

We assume that I $\neq$ J.

J cannot be subset of I as J is optimal

Also I can not be subset of J by the algo.

So there exists a job a in I such that a is not in J

And a job b in J Which is not in I

Let us take a “a highest profit job” such that a is in I and not in J.

Clearly $p_a \geq p_b$ for all jobs that are in J but not in I.

Since if $p_b > p_a$ then greedy approach would consider job b before job a and included into I.

Let Si and Sj be the feasible sequences for feasible sets I and J.

We will assume that common jobs of I and J can be processed in same time intervals.

Let i be a job scheduled in t to t+1 in S i and t' to t'+1 in Sj

If t < t' then interchange the job if any in [t' , t'+1] in Si with job i.

Similarly if t' < t similar transformation can be done in Sj

Now consider the interval [ta, ta+1] in Si in which job a is scheduled.

Let b the job(if any) scheduled in Sj in this interval.

$p_a \geq p_b$ from the choice of a

So scheduling a from ta to ta+1 in Sj and discarding job b gives a feasible schedule for the set J' = J- {b} + {a}.

Clearly J' has profit no less than that of J and differs from I in one less job than J does.

By repeatedly using this approach J can be transformed into I without decreasing the profit value.