Chapter 4
The Greedy Approach

Greedy Approach vs Dynamic Programming

- Like dynamic programming, greedy algorithms are often used to solve optimization problems.
- Greedy – problem is not divided into smaller sub-problems
- Obtain a solution by making a sequence of choices, each choice appears to be the best choice at that step
- Once a choice is made, it cannot be reconsidered
- Choice made without regard to past or future choice

Greedy Approach

- Initially the solution set $=\phi$

```
Repeat
    Add an item to the solution set
Until the set represents a solution to that instance
```

- An **optimization problem** is the problem of finding the *best* solution from all feasible solutions.
- *Greedy algorithm* : each choice is locally optimal.
- The hope is that a globally optimal solution will be obtained, but this is not always the case.
- For a given algorithm, we must determine whether the solution is always optimal.

Greedy Algorithm

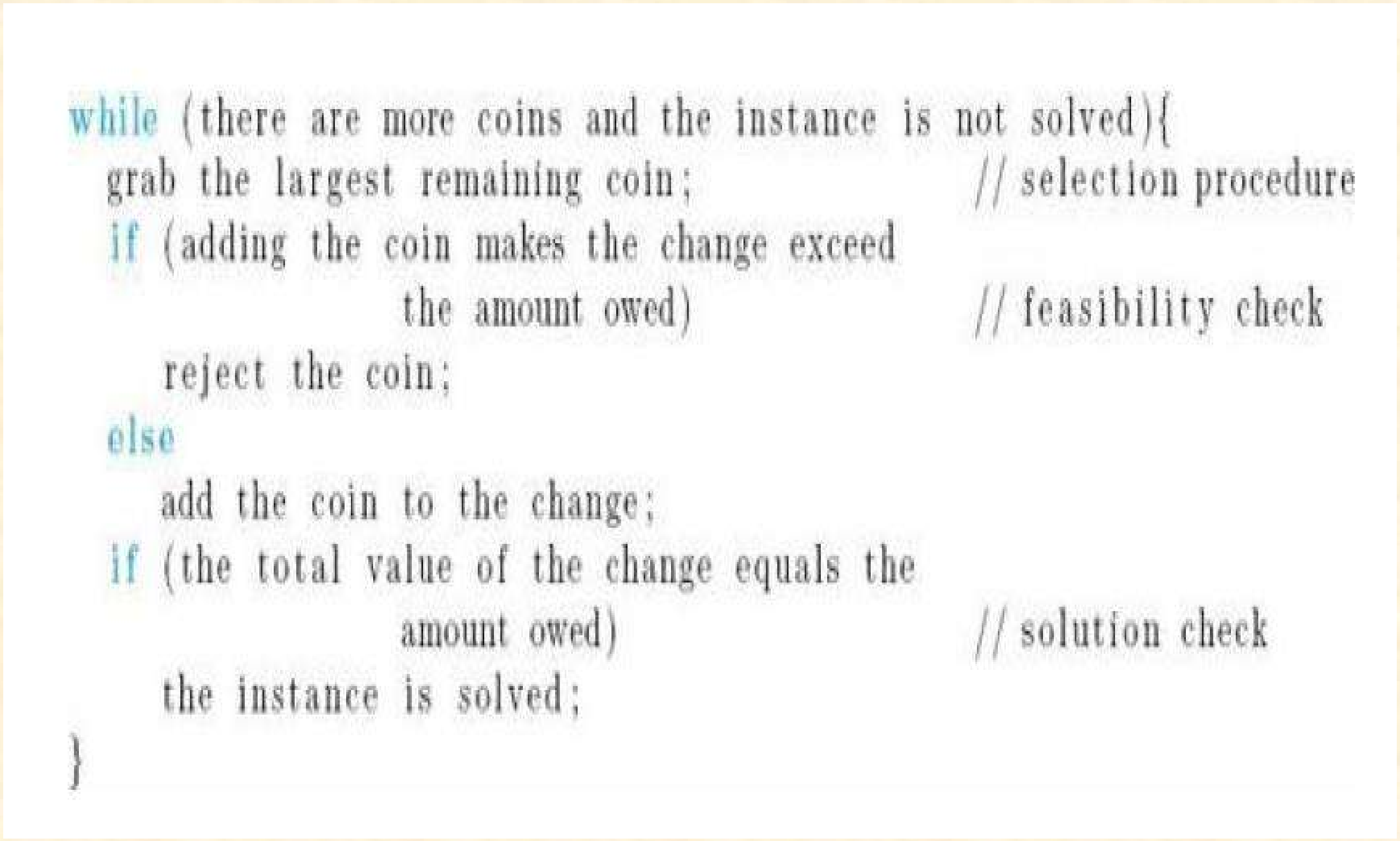
- ***Selection procedure***: Choose the next item to add to the solution set according to the greedy criterion satisfying the locally optimal consideration
- ***Feasibility Check***: Determine if the new set is feasible by determining if it is possible to complete this set to provide a solution to the problem instance
- ***Solution Check:*** Determine whether the new set produced is a solution to the problem instance.

Joe’s change problem

- The problem of giving change for a purchase. Customers usually don’t want to receive a lot of coins.

Joe’s change problem

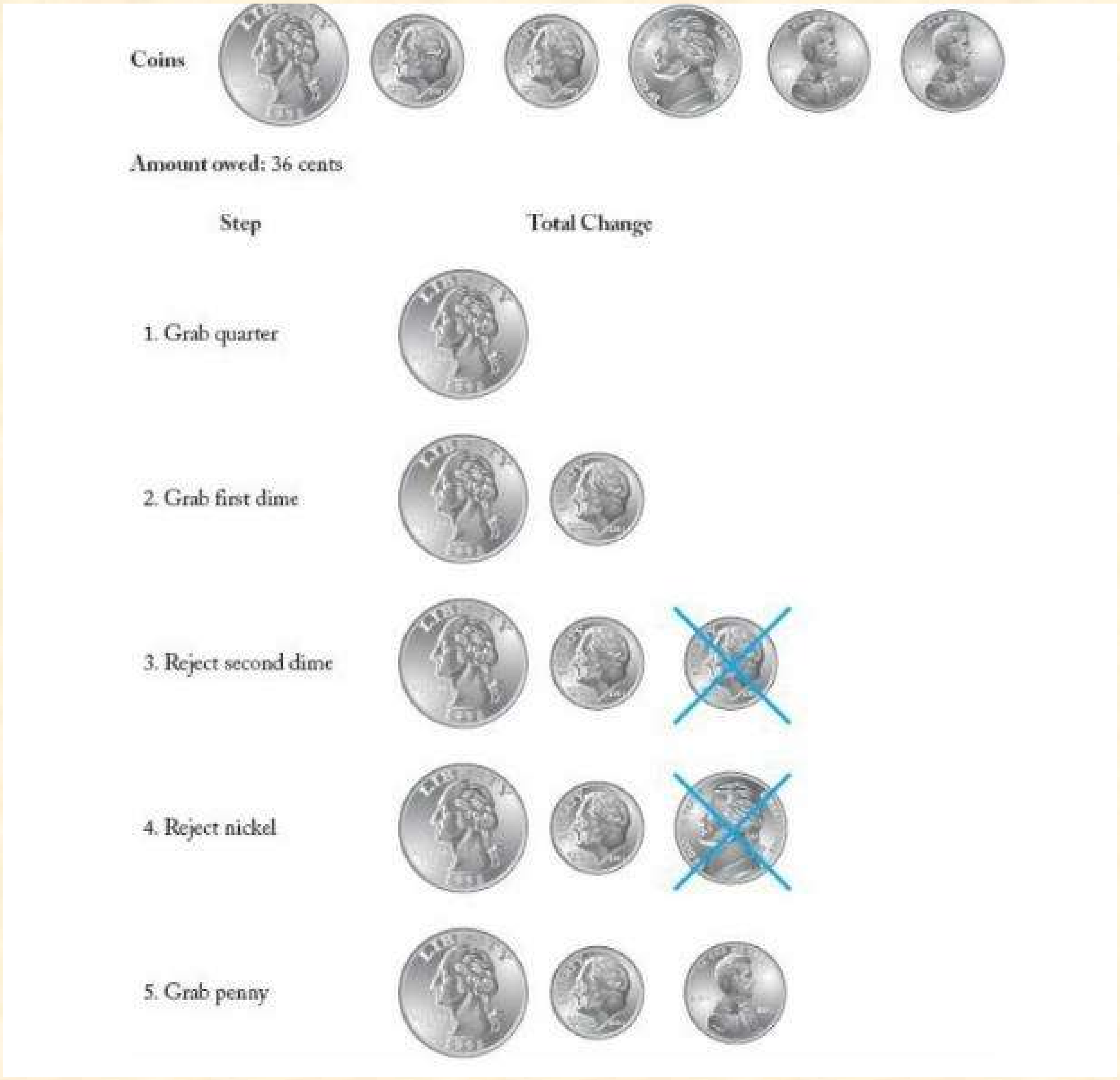
- ***Selection procedure***: Joe starts by looking for the largest coin (in value) he can find. That is, his criterion for deciding which coin is best (locally optimal) is the value of the coin
- .***Feasibility Check***: he sees if adding this coin to the change would make the total value of the change exceed the amount owed.
- ***Solution Check:*** he checks to see if the value of the change is now equal to the amount owed.



```
while (there are more coins and the instance is not solved){
  grab the largest remaining coin;                  // selection procedure
  if (adding the coin makes the change exceed
                    the amount owed)                // feasibility check
     reject the coin;
  else
     add the coin to the change;
  if (the total value of the change equals the
                    amount owed)                    // solution check
     the instance is solved;
}
```

Greedy algorithm

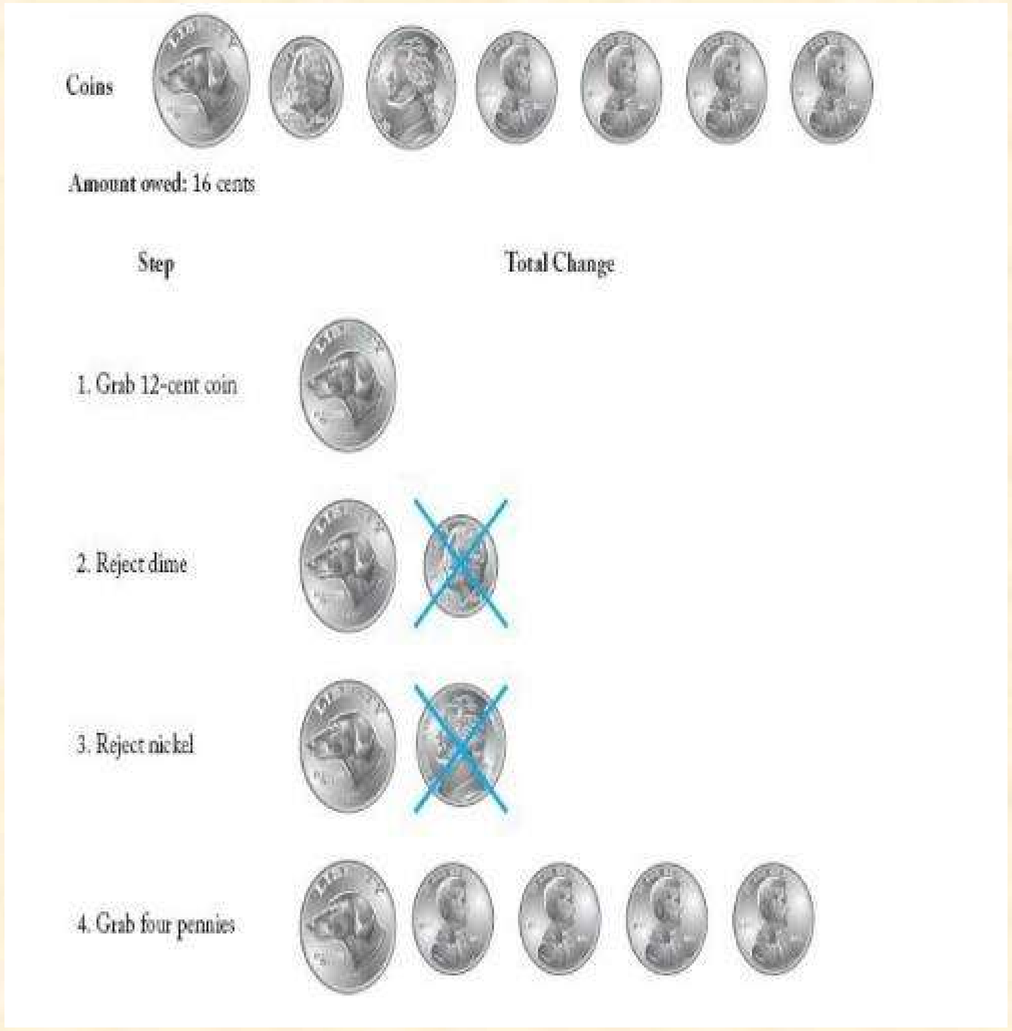
- the algorithm is called “greedy” because the selection procedure simply consists of greedily grabbing the next-largest coin without considering the potential drawbacks of making such a choice.


Coins
Amount owed: 36 cents
Step
Total Change
1. Grab quarter
2. Grab first dime
3. Reject second dime
4. Reject nickel
5. Grab penny

Greedy algorithm

- does the solution provided by the algorithm contain the minimum number of coins necessary to give the correct change?
- If the coins consist of U.S. coins (penny, nickel, dime, quarter, half dollar) and if there is at least one type of each coin available, the greedy algorithm always returns an optimal solution when a solution exists. This is proven in the exercises.

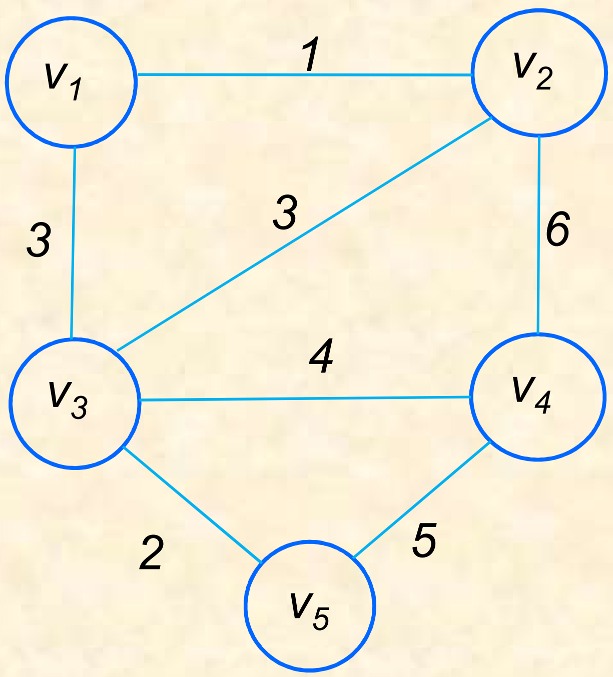
Greedy algorithm does not always give an optimal solution.



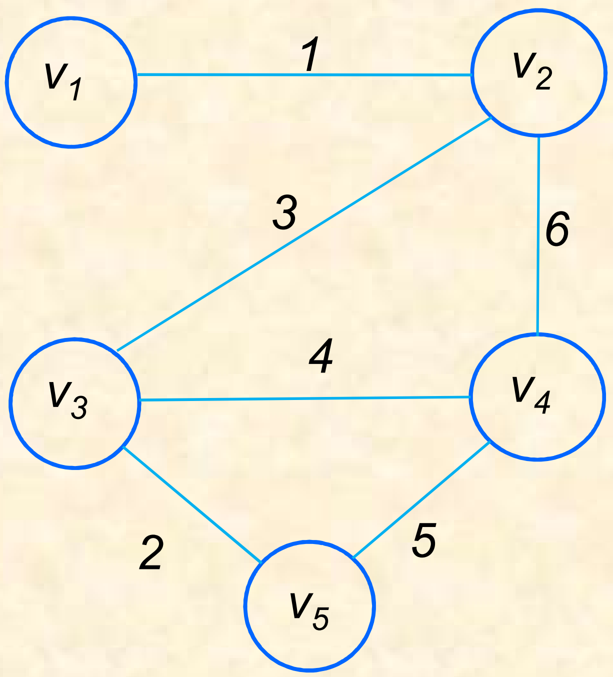
- An undirected graph G consists of a finite set **V** whose members are called the vertices of G, together with a set **E** of pairs of vertices in V. These pairs are called the edges of G. We denote G by G = (V , E)
- An undirected graph is called **connected** if there is a path between every pair of vertices.
- ***Simple cycle :*** In an undirected graph, a path from a vertex to itself, which contains at least three vertices.
- An undirected graph with **no** simple cycles is called ***acyclic.***
- A **tree** is an acyclic, connected, undirected graph.
- **Rooted tree** is defined as a tree with one vertex designated as the root.

4.1 Minimum Spanning Trees

- Applications: road construction .
- G : weighted, undirected graph.
- A ***spanning tree for G*** is a connected subgraph that contains **all the vertices** in G and is a tree.
- ***minimum spanning tree*** a spanning tree of minimum weight.
- ***A*** graph can have more than one minimum spanning tree.

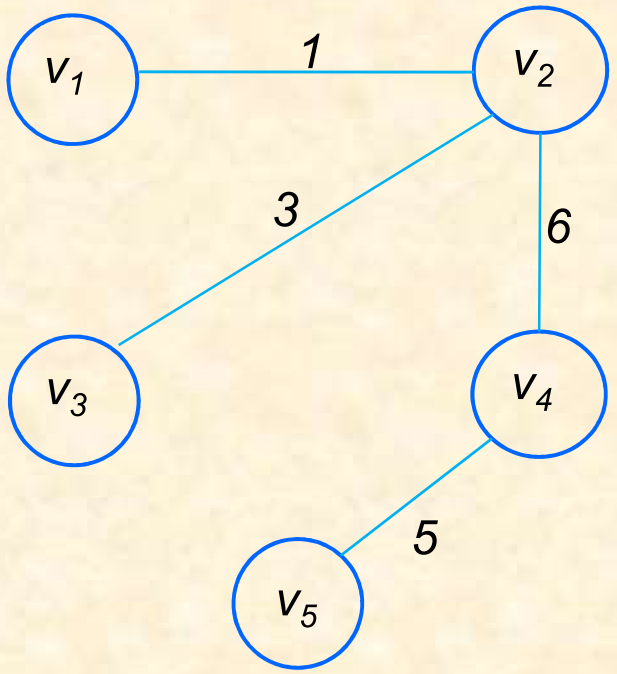


(a) A connected weighted, undirected graph G



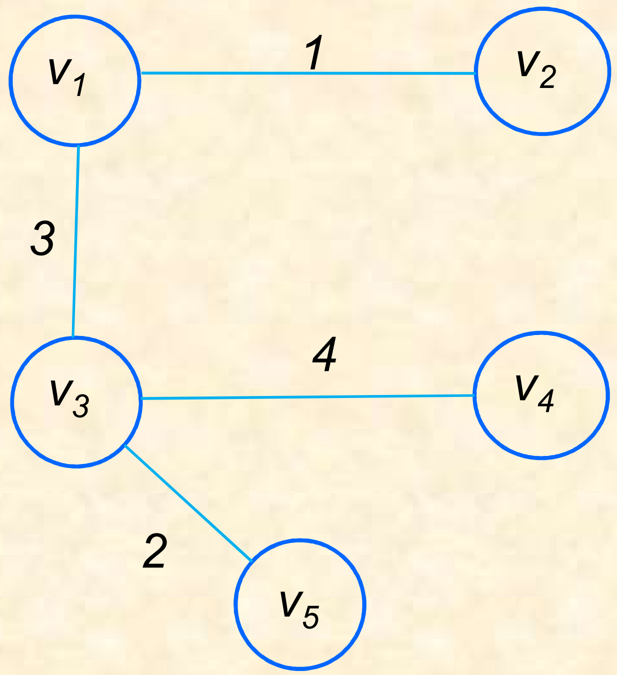
(b) G is connected

(c) spanning tree that does not have minimum weight



Total weight=15

(d) A minimum spanning tree



Total weight=10

- A subgraph with minimum weight must be a tree, because if a subgraph were not a tree, it would contain a simple cycle, and we could remove any edge on the cycle, resulting in a connected graph with a smaller weight.
- The subgraph in Figure 4.3(b) cannot have minimum weight because, *For example* if we remove *v4,v5,* the subgraph remains connected.

- Let G = (V , E)
- Let T be a spanning tree for G: T = (V, F) where F $\subseteq$ E
- Find T such that the sum of the weights of the edges in F is minimal

Greedy Algorithms for finding a Minimum Spanning Tree

- Prim’s Algorithm
- Kruskal’s Algorithm
- Each uses a different locally optimal property
- Must prove each algorithm

```
F = ∅                                                     // Initialize set of
                                                          // edges to empty.
while (the instance is not solved){
  select an edge according to some locally
  optimal consideration;                                  // selection procedure
  if (adding the edge to F does not create a cycle)
    add it;                                               // feasibility check
  if (T = (V,F) is a spanning tree)                       // solution check
    the instance is solved;
}
```

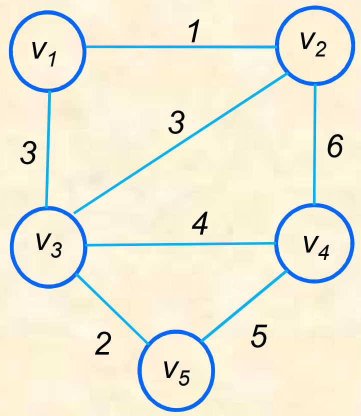
- This algorithm simply says “select an edge according to some locally optimal consideration.”
- There is no unique locally optimal property for a given problem. We will investigate two different greedy algorithms for this problem, Prim’s algorithm and Kruskal’s algorithm. Each uses a different locally optimal property.
- Recall that there is no guarantee that a given greedy algorithm always yields an optimal solution.

4.1.1 Prim's Algorithm

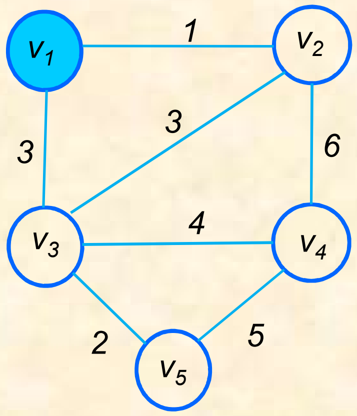
A high-level greedy algorithm for the problem could proceed

```
F = ∅        //Empty subset of edges
 Y = {$v_1$}, $v_1$ is an arbitrary vertex Y
While the instance is not solved
{
   Select a vertex in V-Y  that is nearest to Y   // Selection procedure and feasibility check
     A vertex nearest to Y is a vertex in V-Y that is connected to  a
    vertex in Y by an edge of minimum weight
    Add the selected vertex to Y
    Add the edge connecting the selected vertex to F
   If (Y==V) the instance is solved   //solution check
}
```

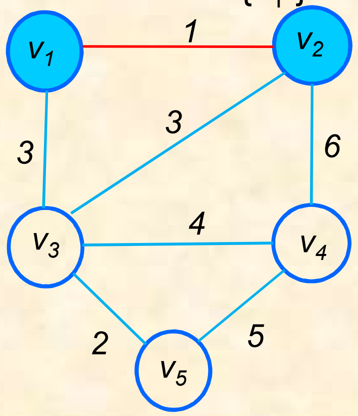
Figure 4.4



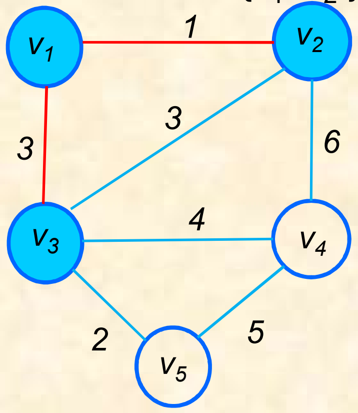
1. v_1 is selected first



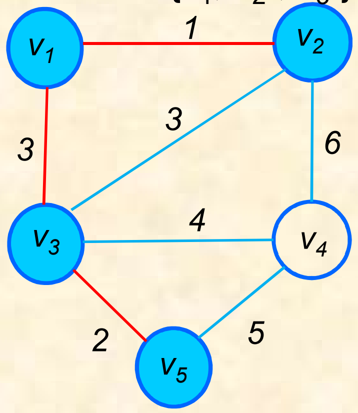
2. v_2 is selected : it is nearest to $\{v_1\}$



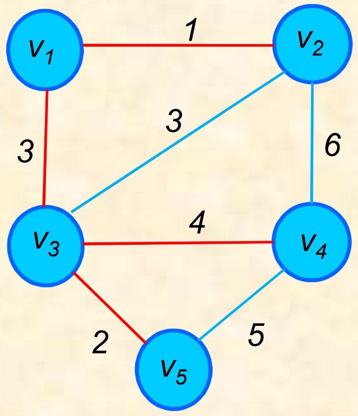
3. v_3 is selected : it is nearest to $\{v_1, v_2\}$



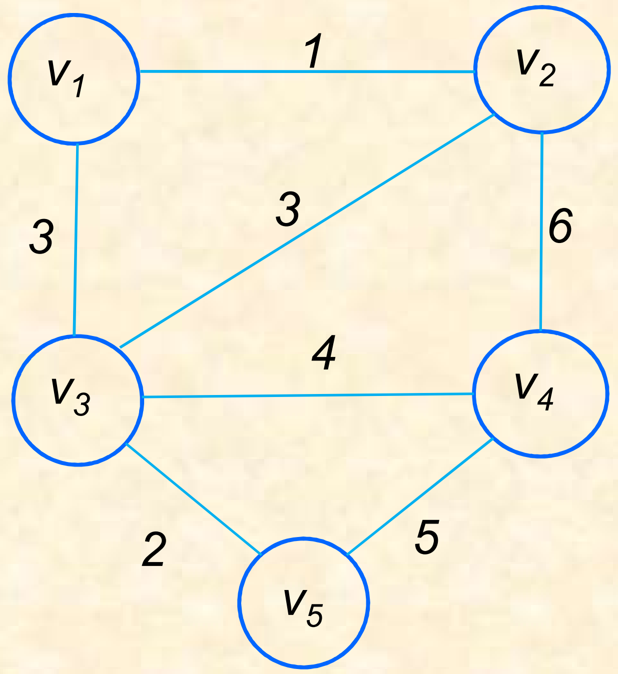
4. v_5 is selected : it is nearest to $\{v_1, v_2, v_3\}$

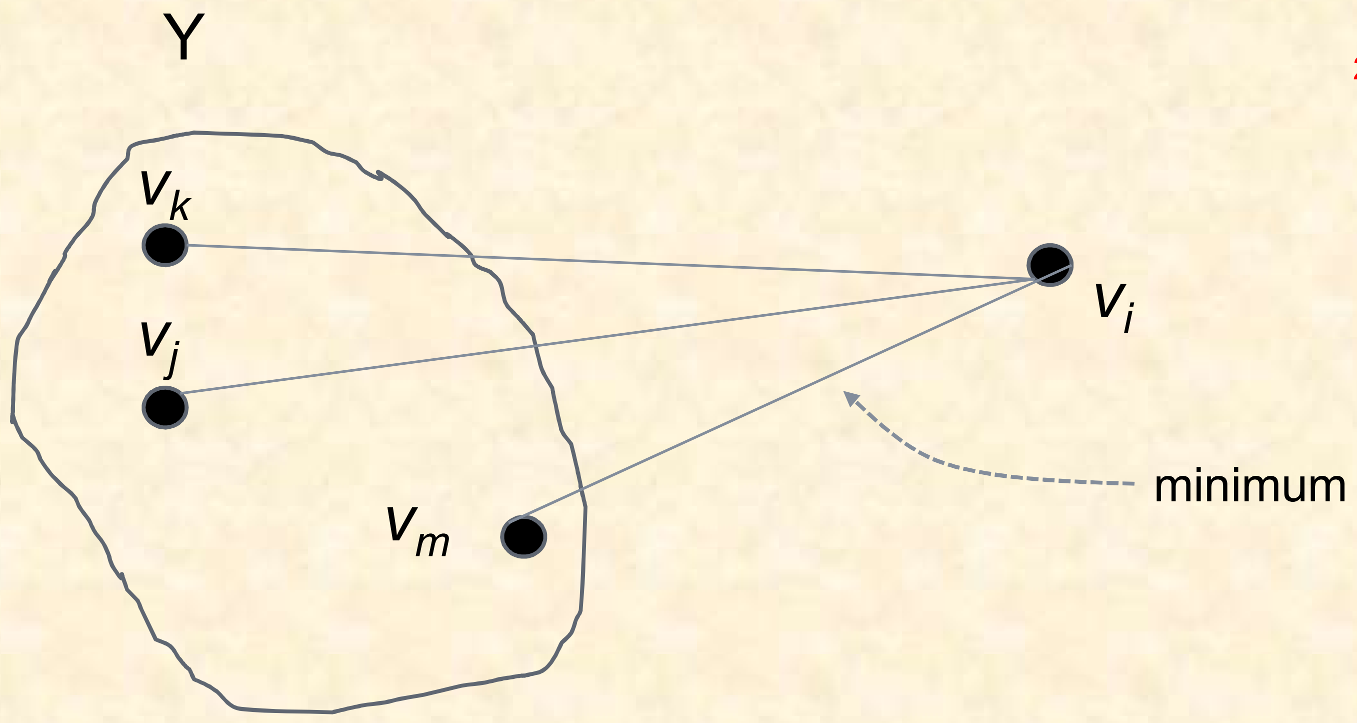


5. v_5 is selected



$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 3 & \infty & \infty \\ 2 & 1 & 0 & 3 & 6 & \infty \\ 3 & 3 & 3 & 0 & 4 & 2 \\ 4 & \infty & 6 & 4 & 0 & 5 \\ 5 & \infty & \infty & 2 & 5 & 0 \end{array}$$





nearest [*i*] =m
distance [*i*] = weight on edge $v_i\ v_m$

two arrays, for $i = 2, \ldots, n$,

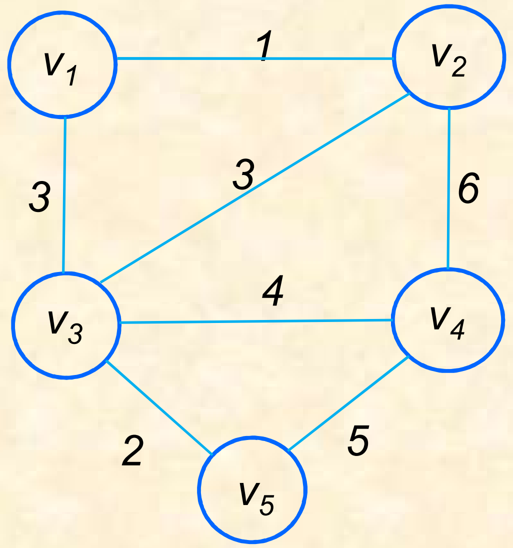
- *nearest* [*i*] = index of the vertex in *Y* nearest to v_i
- *distance* [*i*] = weight on edge between v_i and $v_{nearest\,[i]}$.

Because at the start *Y* = {*v*1}, *nearest*[*i*] is initialized to 1 and *distance*[*i*] is initialized to the weight on the edge between *v*1 and *vi*.

Example

Y={v1}

nearest [2]=1; *nearest* [3]=1;
nearest [4]=1; *nearest* [5]=1;
distance [2] = *W* [1][2]=1; *distance* [3] = 3;
distance [4] = ∞; *distance* [5] = ∞;



- To determine which vertex to add to *Y* , in each iteration we compute:
  1. the index for which *distance[i]* is the smallest. We call this index v_{vnear} .
  2. The vertex v_{vnear} is added to *Y* by setting *distance* [v_{vnear}] *to −1.*

Algorithm 4.1
Prim's Algorithm

- Problem: Determine a minimum spanning tree.
- **Inputs**: integer $n \geq 2$, and a connected, weighted, undirected graph containing n vertices. The graph is represented by a two-dimensional array W indexed (1..n,1..n)

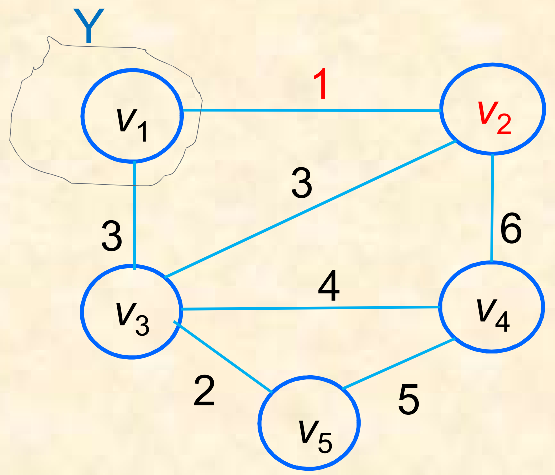
  where $W[i][j]$ is the weight on the edge between the ith vertex and the *jth* vertex.
- **Outputs**: set of edges F in a minimum spanning tree for the graph.

```
void prim (int n, const number W[ ][ ], set_of_edges&  F)
{
   index i, vnear; number min; edge e; index nearest[2..n]; number distance [2..n];
   F=ϕ;                                  //Empty subset of edges
   for (i=2;i<=n; i++) {  nearest[i]=1; distance [i]= W[1 ][i ];  }
   repeat (n-1 times) {
      min=∞;
      for (i=2;i<=n; i++)
         if (0<=distance [i] <min);
            {    min= distance [i] ;  vnear=i }
       e=edge conecting $v_{vnear}$  and  $v_{nearest[vnear]}$
       add e to F;
       distance [vnear]=-1;
       for (i=2;i<=n; i++)                                   // for each vertex not in Y
          if (W[i][vnear]<distance [i] )                      // update its distance from Y
             {distance [i] = W[i][vnear]; nearest[i]=vnear;}
   }
}
```

Initialization:

nrt[2]=1; nrt[3]=1; nrt[4]=1; nrt[5]=1;

dist [2]=1; dist [3]=3; dist [4]= ∞; dist [5]= ∞;



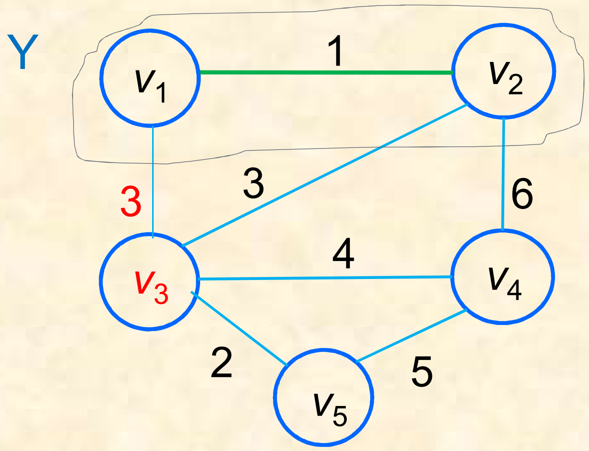
Step 1

for: min=1, vnear=2; add edge v_2 v_1

dist [2]=-1;

Update: dist [3]=3; dist [4]= 6; dist [5]= ∞;

nrt [2]=1; nrt[3]=1; nrt[4]=2; nrt[5]=1;



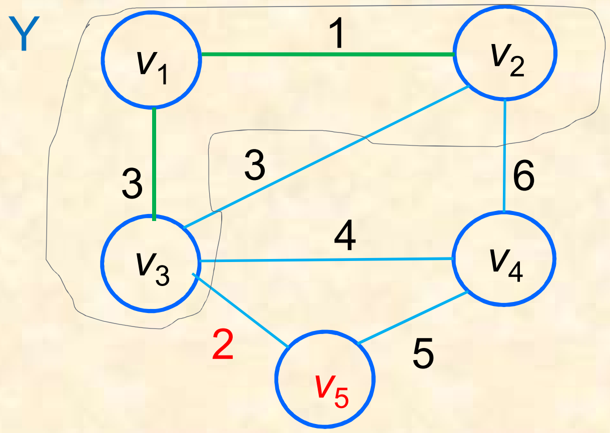
Step 2

for: min=3, vnear=3; add edge v_1 v_3

dist [2]=-1; dist [3]=-1;

Update: dist [4]= 4; dist [5]= 2;

nrt [2]=1; nrt[3]=1; nrt[4]=3; nrt[5]=3;



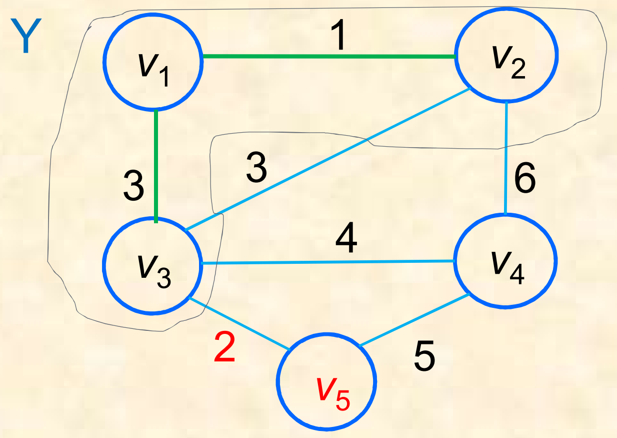
Step 2

for: min=3, vnear=3; add edge v_1 v_3

dist [2]=-1; dist [3]=-1;

Update: dist [4]= 4; dist [5]= 2;

nrt [2]=1; nrt[3]=1; nrt[4]=3; nrt[5]=3;

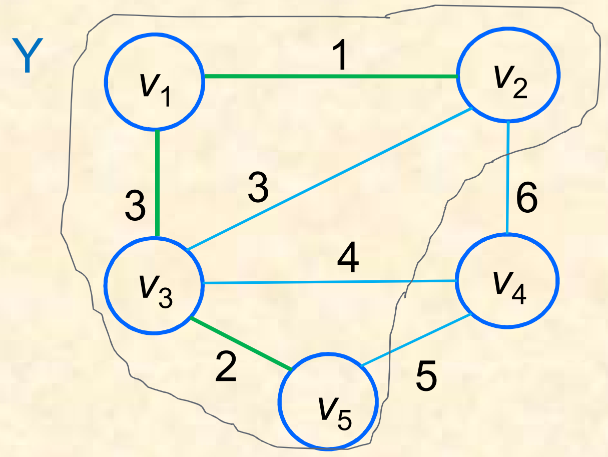


Step 3

min=2, vnear=5; add edge F={v_5 v_3 }

dist [2]=-1; dist [3]=-1; dist [4]= 4; dist [5]= -1;

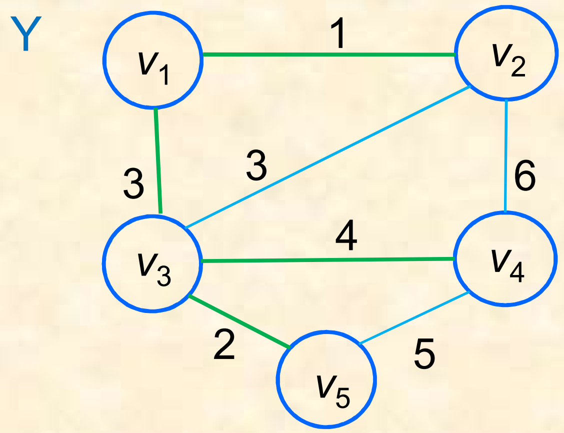
nrt [2]=1; nrt[3]=1; nrt[4]=3; nrt[5]=3;



Step 4

min=4, vnear=4; add edge F={v_4 v_3 }

dist [4]= -1;



Every-Case Time Complexity of Prim's Algorithm 4.1

- Input Size: n (the number of vertices)
- Basic Operation : Two loops with $n - 1$ iterations inside repeat loop
- Repeat loop has $n-1$ iterations
- Time complexity:
  - $T(n) = 2(n - 1)(n - 1) \in \theta(n^2)$

proof for Prim’s algorithm

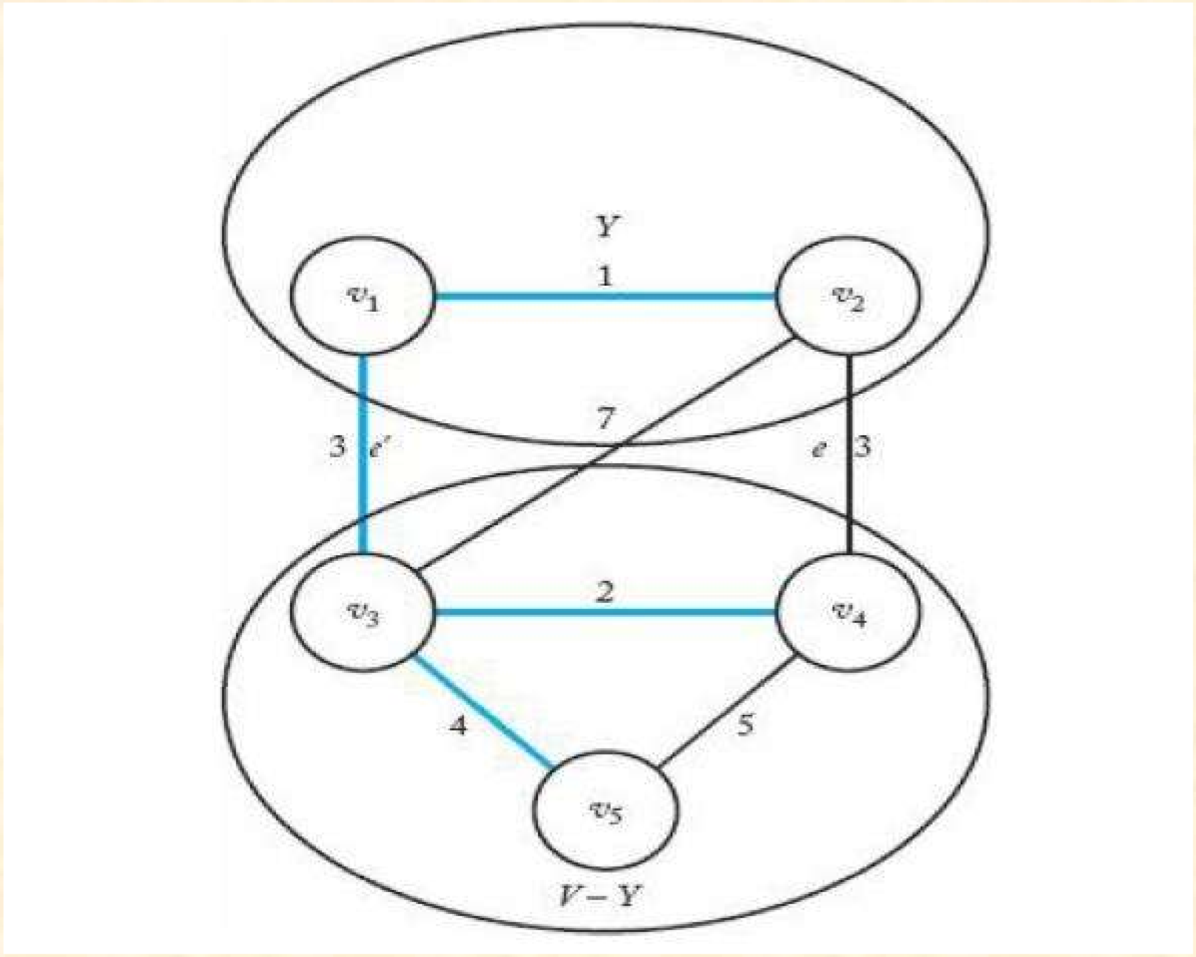
- A subset *F* and *E* is called ***promising*** if edges can be added to it so as to form a minimum spanning tree.

- The subset {(*v*1, *v*2) , (*v*1, *v*3)} is promising, and the subset {(*v*2, *v*4)} is not promising.

- **Lemma 4.1**

Let *G* = (*V* , *E*) be a connected, weighted, undirected graph; let *F* be a promising subset of *E*; and let *Y* be the set of vertices connected by the edges in *F*. If *e* is an edge of minimum weight that connects a vertex in *Y* to a vertex in *V* − *Y* , then *F* ∪ {*e*} is promising.

proof



proof for Prim's algorithm

Theorem 4.1
Prim's algorithm always produces a minimum spanning tree.

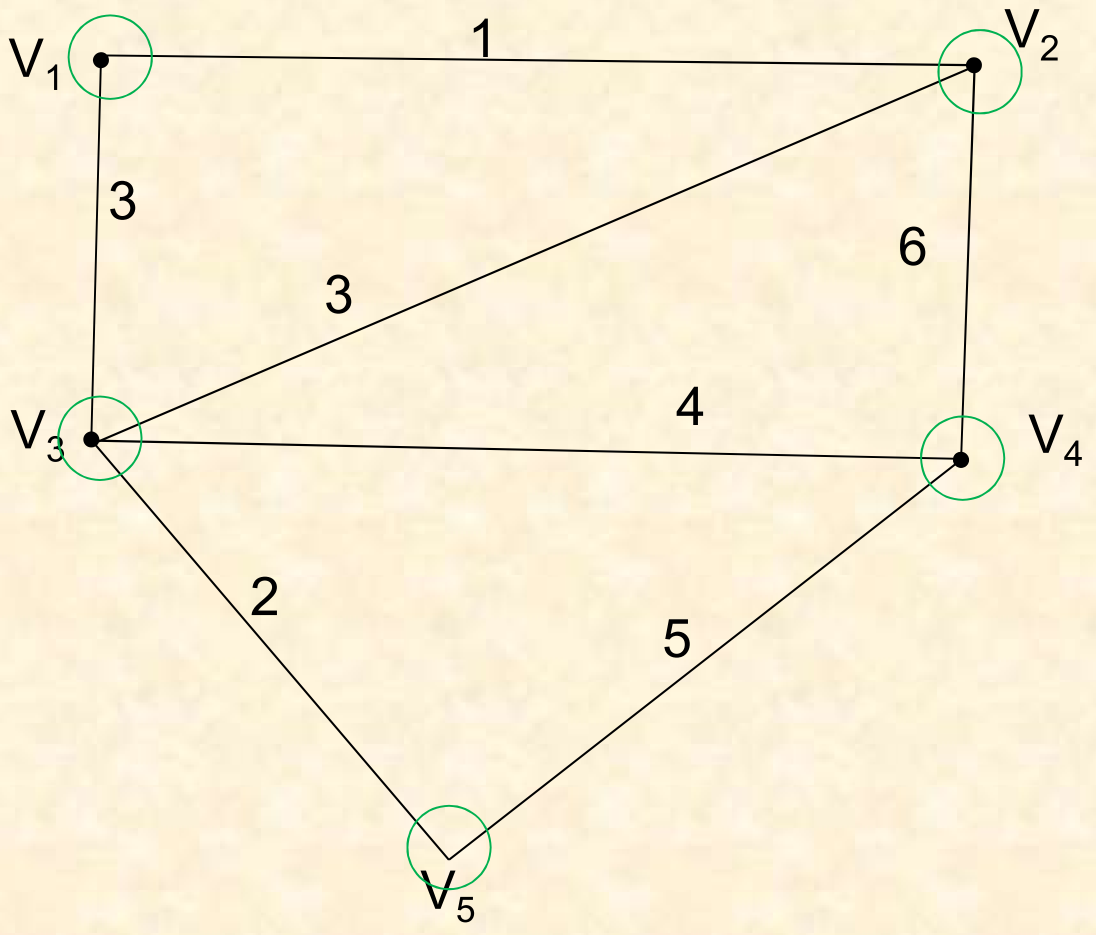
Proof:
We use induction to show that the set *F* is promising after each iteration of the **repeat** loop.

- Induction base: Clearly the empty set $\emptyset$ is promising.
- Induction hypothesis: Assume that, after a given iteration of the **repeat** loop, the set of edges so far selected—namely, *F*—is promising.
- Induction step: We need to show that the set $F \cup \{e\}$ is promising, where *e* is the edge selected in the next iteration. Because the edge *e* selected in the next iteration is an edge of minimum weight that connects a vertex in *Y* to one on $V - Y$, $F \cup \{e\}$ is promising, by Lemma 4.1. This completes the induction proof.

By the induction proof, the final set of edges is promising. Because this set consists of the edges in a spanning tree, that tree must be a minimum spanning tree.

4.1.2 Kruskal’s Minimum Spanning Tree Algorithm

```
F = ∅;                                                  //F is set of edges
Create disjoint subsets of V;
Sort the edges in E in non-decreasing order;
while (the instance is not solved)
{
    select next edge;                                   //selection procedure
    if (edge connects 2 vertices in disjoint subsets)
    {
        merge the subsets;
        add the edge to F;
    }
    if (all the subsets are merged into a single set)   //solution check
        instance is solved;
}
```



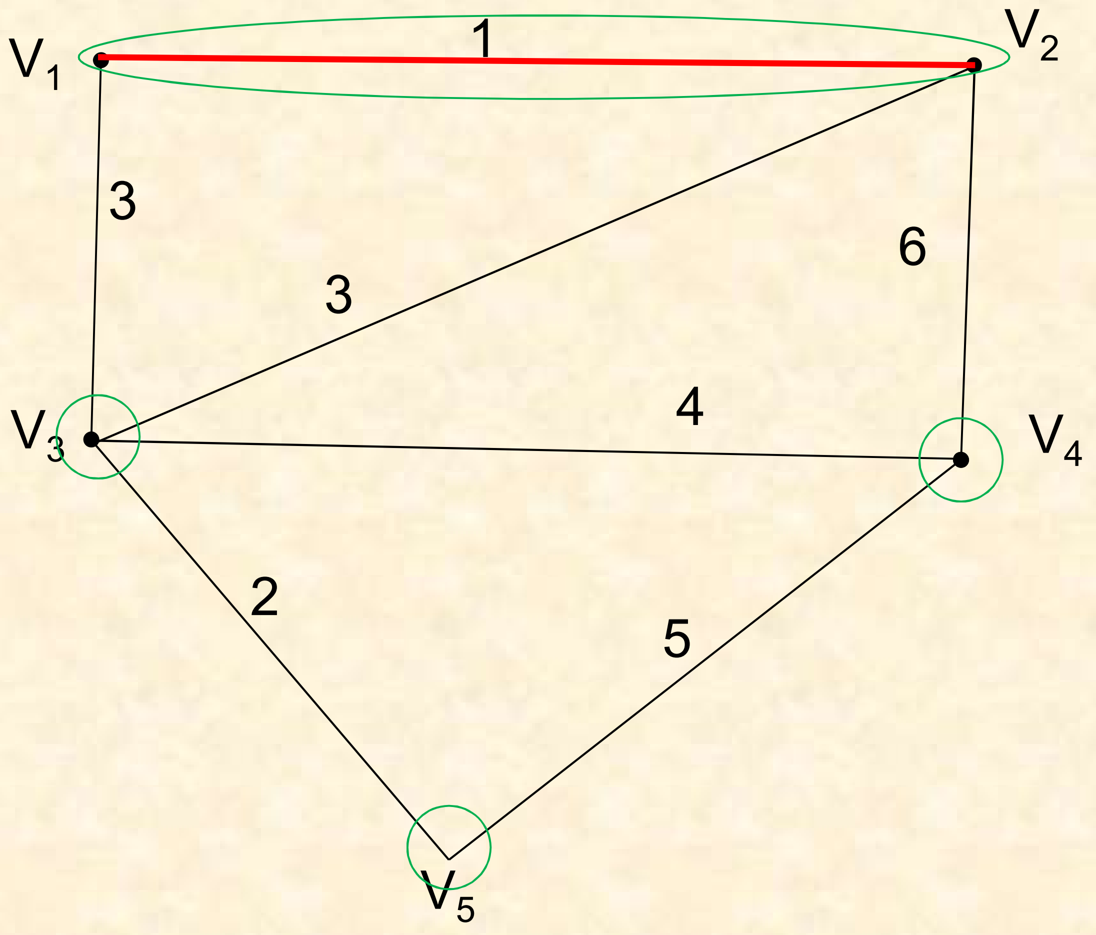
List the edges in order of size:

V_1V_2	1
V_3V_5	2
V_1V_3	3
V_2V_3	3
V_3V_4	4
V_4V_5	5
V_2V_4	6

Disjoint sets

$\{V_1\}, \{V_2\}, \{V_3\}, \{V_4\}, \{V_5\}$

$F=\varnothing$



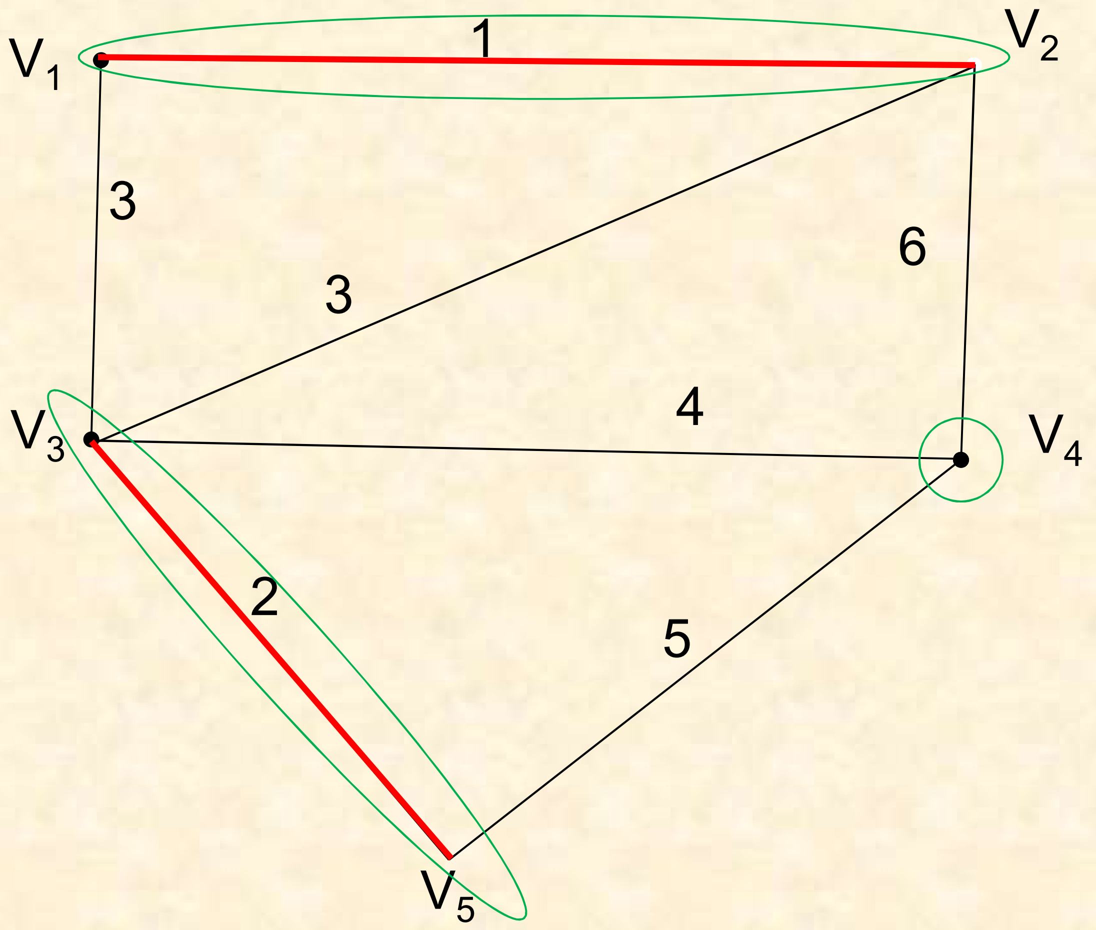
Select the shortest edge :

V_1V_2	1
V_3V_5	2
V_1V_3	3
V_2V_3	3
V_3V_4	4
V_4V_5	5
V_2V_4	6

Disjoint sets

$\{V_1, V_2\}, \{V_3\}, \{V_4\}, \{V_5\}$

$F=\{V_1V_2\}$



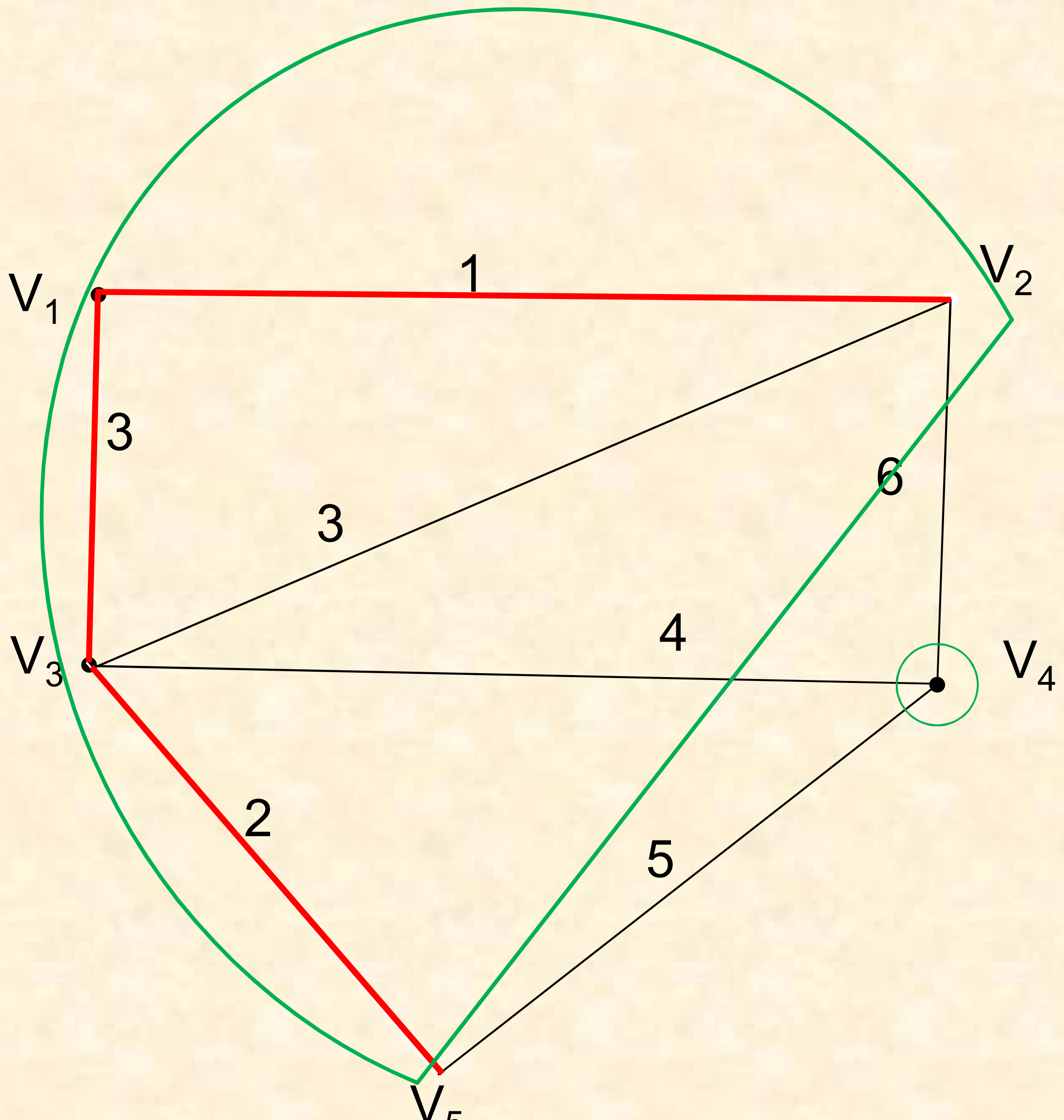
Select the next shortest edge :

V_1V_2	1
V_3V_5	2
V_1V_3	3
V_2V_3	3
V_3V_4	4
V_4V_5	5
V_2V_4	6

Disjoint sets

$\{V_1, V_2\}, \{V_3, V_5\}, \{V_4\}$

$F=\{V_1V_2 , V_3V_5\}$



Select the next shortest Edge

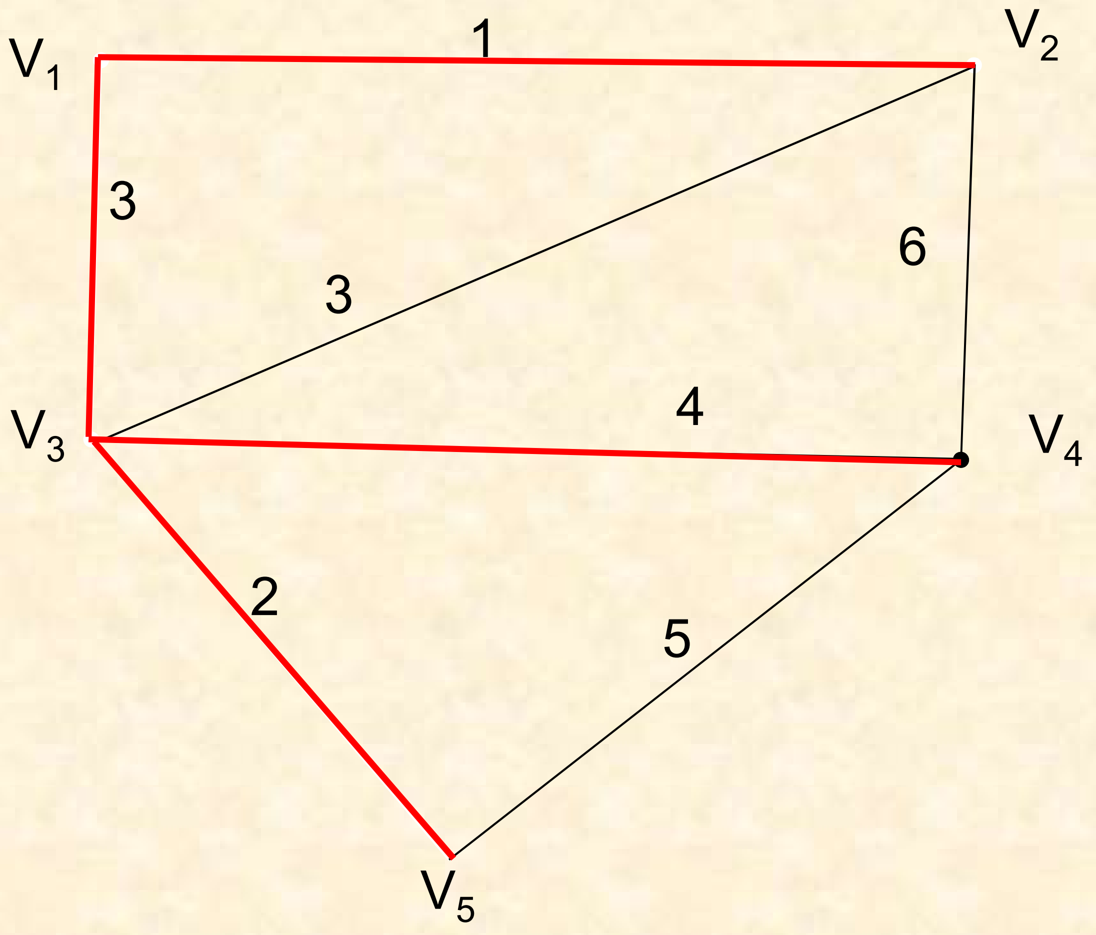
V_1V_2	1
V_3V_5	2
V_1V_3	3
V_2V_3	3
V_3V_4	4
V_4V_5	5
V_2V_4	6

Disjoint sets

$\{V_1, V_2, V_3, V_5\}, \{V_4\}$

$F=\{V_1V_2, V_3V_5, V_1V_3\}$

Select the next shortest edge



V_1V_2	1
V_3V_5	2
V_1V_3	3
V_2V_3	3
V_3V_4	4
V_4V_5	5
V_2V_4	6

Disjoint sets

$\{V_1, V_2, V_3, V_5, V_4\}$

$F=\{V_1V_2, V_3V_5, V_1V_3, V_3V_4\}$

Algorithm 4.2 Kruskal

```
void kruskal (int $n$, int $m$, set_of_edges $E$, set_of_edges& $F$ )
{ index $i, j$ ;
  set_pointer $p, q$ ;
  edge $e$ ;
  sort the $m$ edges in $E$  in non-decreasing order
  initial ($n$);                    // initialize n disjoint subsets
  while (number of edges in $F$ is less than $n$ -1{
      e=edge $v_i$ $v_j$ with least weight not yet considered ;
      $p$ = find ($i$ );              // makes p point to the set containing index i.
     $q$ = find ( $j$ );
     if ( ! equal ($p$, $q$)) {
        merge ($p$, $q$);
        add $e$ to  $F$;
     }
  }
}
```

Worst-case Time Complexity Kruskal

1. Sort the edges : We obtained a sorting algorithm in Chapter 2 (Mergesort) that is worst-case $\Theta(m \lg m)$. $W(m) \in \theta(m \lg m)$

2. initial(n) : the time complexity for the initialization is given by $T(n) \in \theta(n)$

3. while loop:

- In the worst case, every edge is considered before the **while** loop is exited, which means there are *m* passes through the loop
  - p = find(i) sets p to point at the set containing index i, find $\in \theta(\lg m)$
  - merge(p,q) merges 2 sets into 1 set, merge $\in \theta(c)$, c is a constant
  - equal(p,q) where p and q point to sets returns true p and q point to the same set, equal $\in \theta(c)$ where c is a constant

- while loop: the time complexity for *m passes* through the loop is given by

  $W(m) \in \theta(m \lg m)$ (appendix C)

- Because $m \geq n-1$, the sorting and the manipulations of the disjoint sets dominate the initialization time, which means that To connect n nodes requires at least n-1 edges: $m >= n-1$, $W(m, n) \in \theta(m \lg m)$
- G fully connected $m = n(n-1)/2 \in \theta(n^2)$

  $W(m,n) \in \theta(n^2 \lg n^2) = \theta(n^2 2\lg n) = \theta(n^2 \lg n)$

See appendix C – disjoint set ADT

Spanning Tree Produced by Kruskal’s Algorithm Minimal?

- Lemma 4.2
- Theorem 4.2

Lemma 4.2
Let

- G = (V, E) be a connected, weighted, undirected graph
- F is a promising subset of E
- Let e be an edge of minimum weight in E – F
- $F \cup \{e\}$ has no cycles
- $F \cup \{e\}$ is promising
- Proof of Lemma 4.2 is similar to proof of Lemma 4.1

Theorem 4.2

- Kruskal’s Algorithm always produces a minimum spanning tree

Theorem Proof Continued

- e selected in next iteration, it has a minimum weight
- e connects vertices in disjoint sets
- Because e is selected, it is minimum and connects two vertices in disjoint sets
- By Lemma 4.2 $F \cup \{e\}$ is promising

Prim vs Kruskal

- Prim's Algorithm: $T(n) \in \Theta(n^2)$
- Kruskal's Algorithm: $W(m, n) \in \Theta(m \lg m)$

  $n-1 <= m <= (n)(n-1)/2$
- Sparse graph
  - m close to $n - 1$
  - Kruskal $\theta(n \lg n)$
  - Kruskal's faster than Prim
- Highly connected graph
  - Kruskal $\theta(n^2 \lg n)$
  - Prim's faster than kruskal

Huffman Code

- Given a data file, it would therefore be desirable to find a way to store the file as efficiently as possible.
- The problem of ***data compression*** is to find an efficient method for encoding a data file.
- we discuss the encoding method, called ***Huffman code***, and a greedy algorithm for finding a Huffman encoding for a given file.
- A common way to represent a file is to use a ***binary code***. In such a code, each character is represented by a unique binary string, called the ***codeword***.
- A ***fixed-length binary code*** represents each character using the same number of bits.(example)

A ***fixed-length binary code***

- A common way to represent a file is to use a ***binary code***. In such a code, each character is represented by a unique binary string, called the ***codeword***.

- A ***fixed-length binary code*** represents each character using the same number of bits
- Example: suppose our character set is {a, b, c}. Then we could use 2 bits to code each character, since this gives us four possible codewords and we need only three. We could code as follows

a: 00   b:01   c:11

ababcbbbc   000100011101010111 (18 bits)

A variable-length binary code.

- We can obtain a more efficient coding using a ***variable-length binary code***. Such a code can represent different characters using different numbers of bits.

a: 10   b:0   c:11

ababcbbbc   1001001100011   (13 bits)

Optimal Binary Code

- Given a file, the Optimal Binary Code problem is to find a binary character code for the characters in the file, which represents the file in the least number of bits.

Prefix Codes

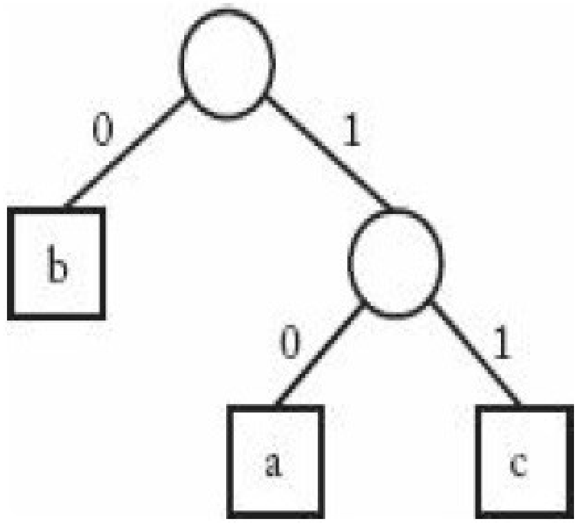
- In a prefix code no codeword for one character constitutes the beginning of the codeword for another character

- For example, if 01 is the code word for ‘*a*’, then 011 could not be the codeword for ‘*b*’.

- Every prefix code can be represented by a binary tree whose leaves are the characters that are to be encoded.

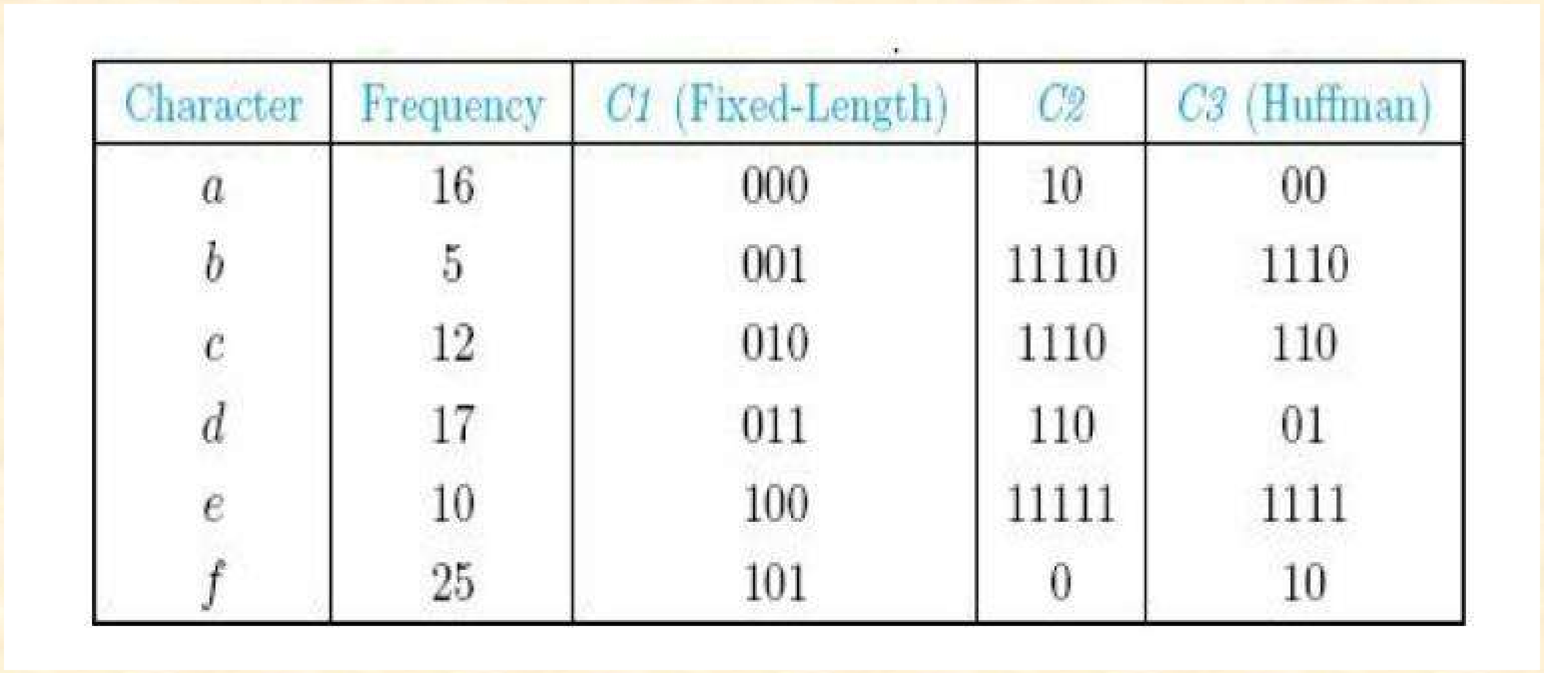
- The advantage of a prefix code is that we need not look ahead when parsing the file.

Prefix Codes

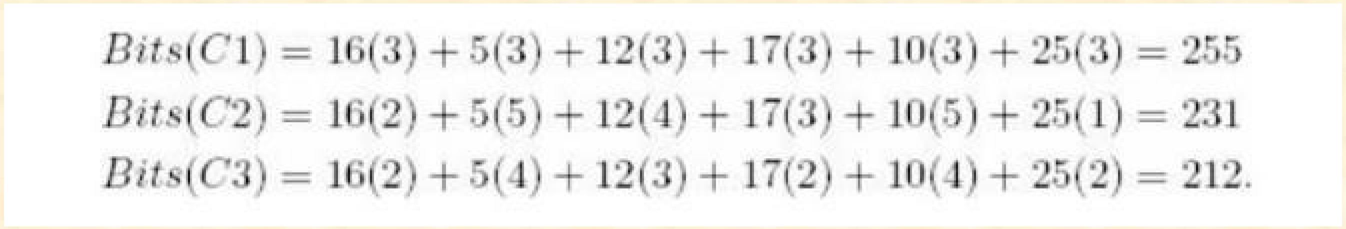


Example

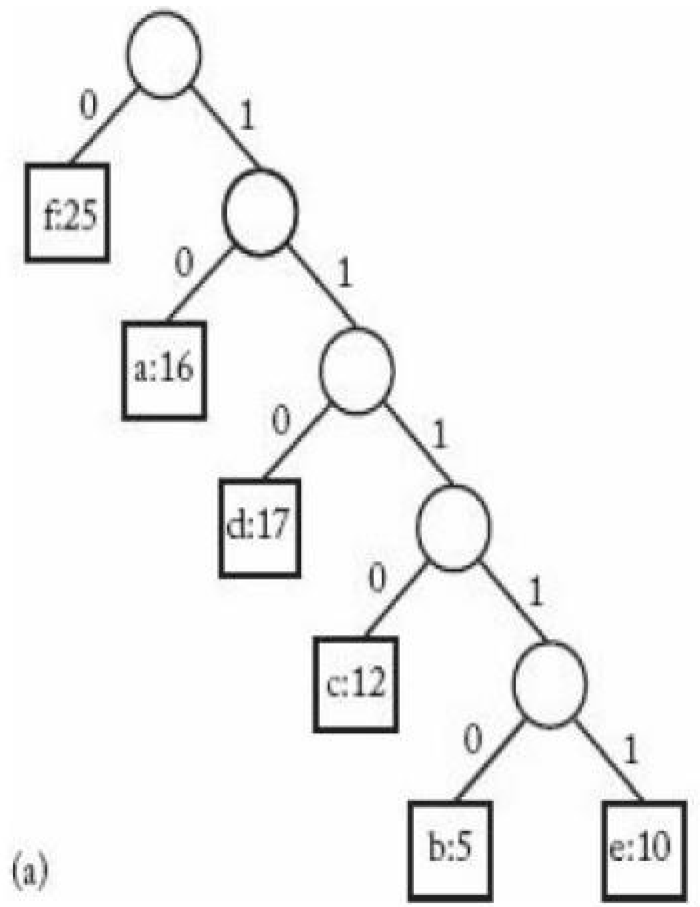
Let’s compute the number of bits for each encoding:

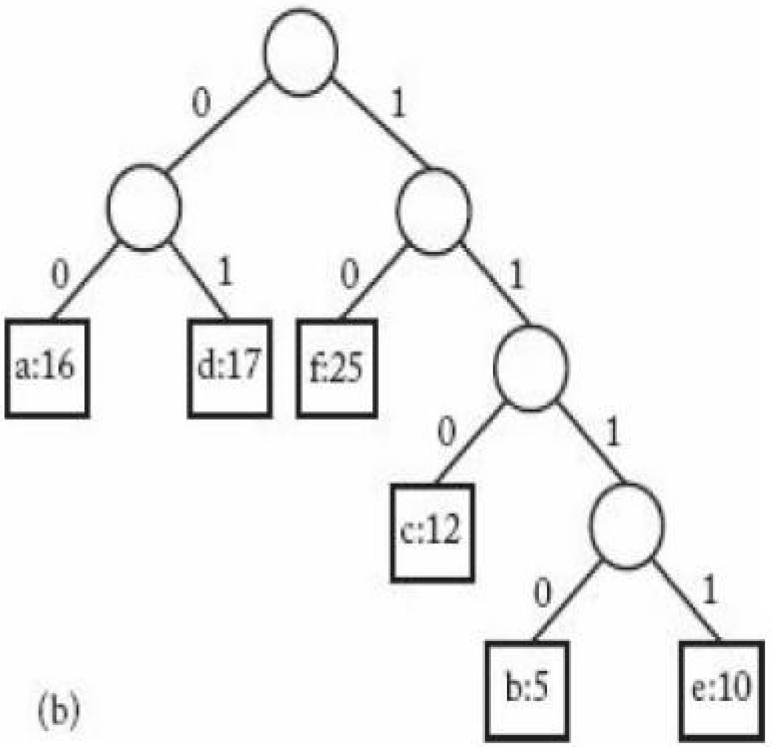


Character	Frequency	*C1* (Fixed-Length)	*C2*	*C3* (Huffman)
a	16	000	10	00
b	5	001	11110	1110
c	12	010	1110	110
d	17	011	110	01
e	10	100	11111	1111
f	25	101	0	10



$$Bits(C1) = 16(3) + 5(3) + 12(3) + 17(3) + 10(3) + 25(3) = 255$$
$$Bits(C2) = 16(2) + 5(5) + 12(4) + 17(3) + 10(5) + 25(1) = 231$$
$$Bits(C3) = 16(2) + 5(4) + 12(3) + 17(2) + 10(4) + 25(2) = 212.$$


0
1
f:25
0
1
a:16
0
1
d:17
0
1
c:12
0
1
b:5
e:10
(a)


0
1
0
1
0
1
a:16
d:17
f:25
0
1
c:12
0
1
b:5
e:10
(b)

- the number of bits it takes to encode a file given the binary tree *T* corresponding to some code is given by

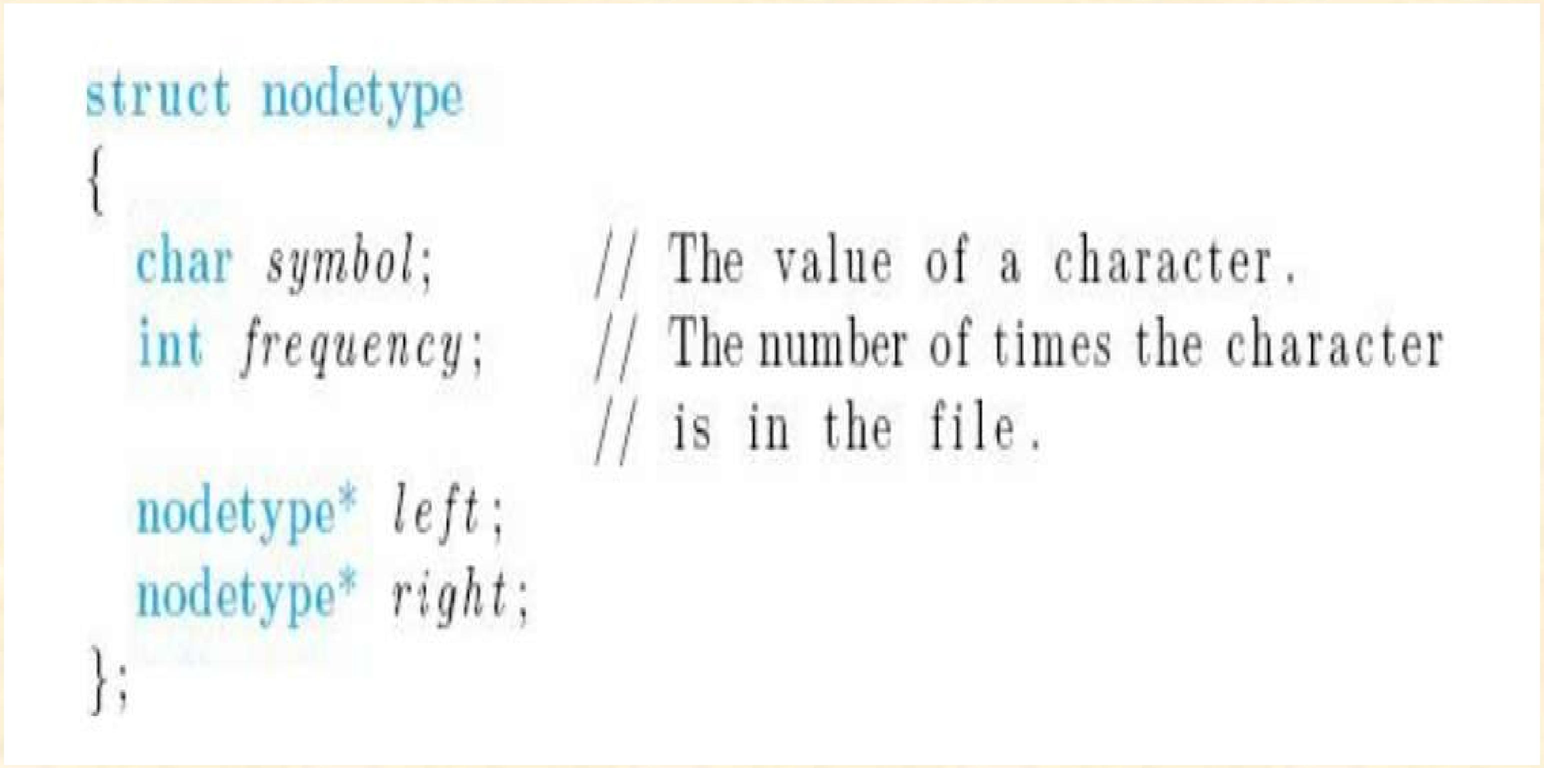
$$bits(T) = \sum_{i=1}^{n} frequency(v_i) depth(v_i),$$

- where {*v*1*, v*2*, … vn*} is the set of characters in the file, *frequency*(*vi*) is the number of times *vi* occurs in the file, and *depth*(*vi*) is the depth of *vi* in *T*.

Huffman’s Algorithm

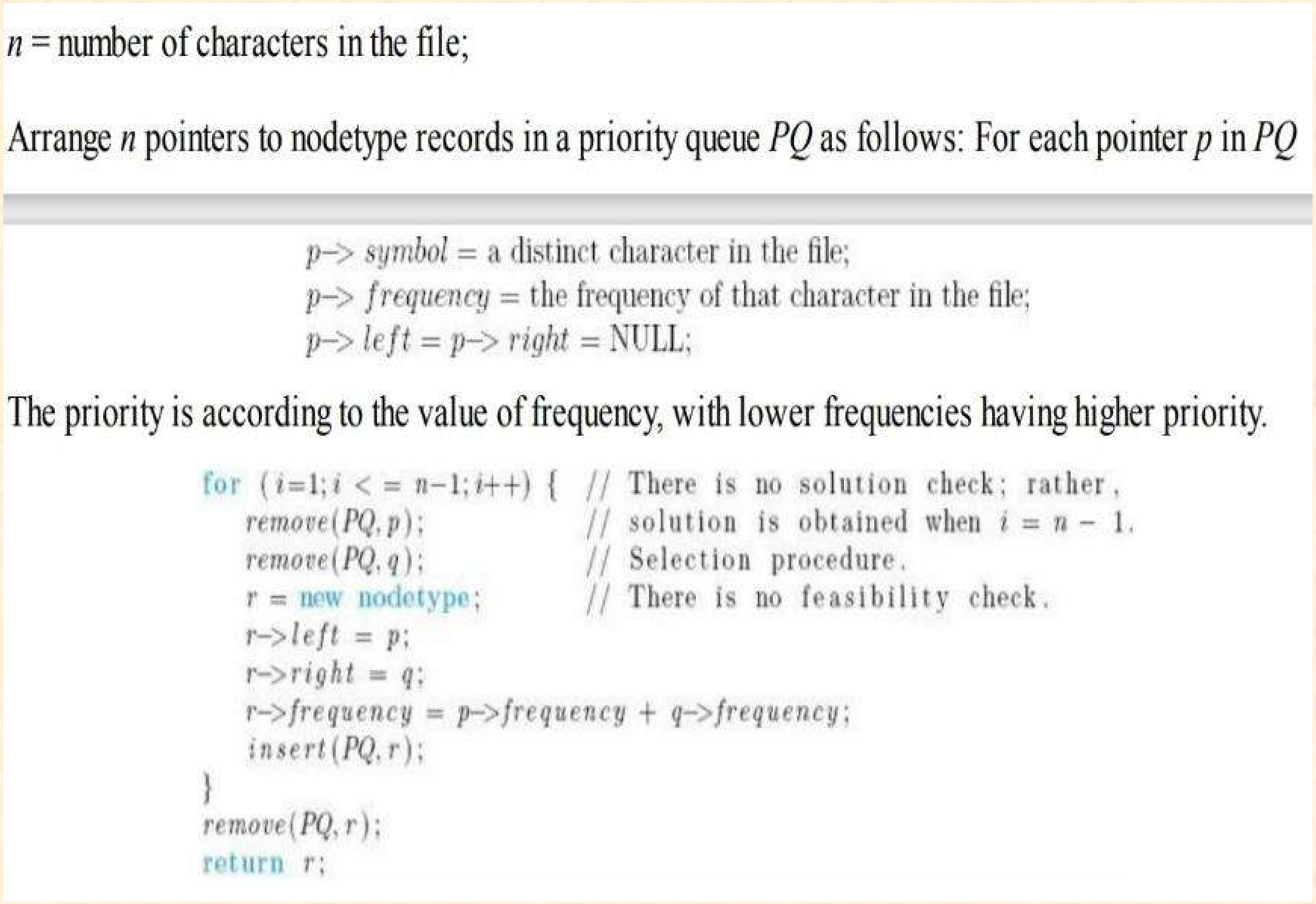
- Huffman developed a greedy algorithm that produces an optimal binary character code by constructing a binary tree corresponding to an optimal code. A code produced by this algorithm is called a **Huffman code**.

Huffman’s Algorithm



```
struct nodetype
{
  char symbol;          // The value of a character.
  int frequency;        // The number of times the character
                        // is in the file.
  nodetype* left;
  nodetype* right;
};
```

Huffman's Algorithm



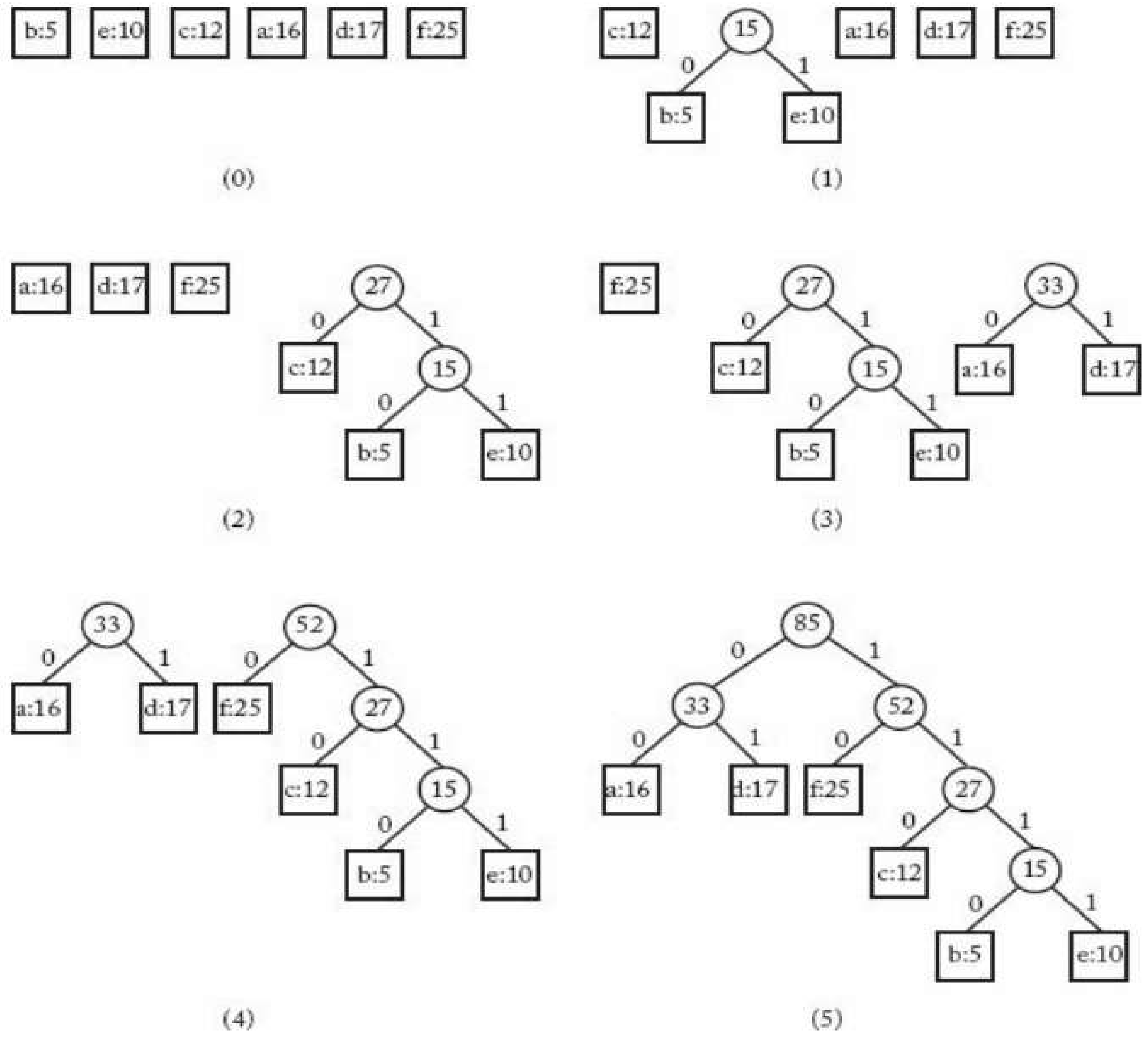
n = number of characters in the file;

Arrange n pointers to nodetype records in a priority queue PQ as follows: For each pointer p in PQ

```
p-> symbol = a distinct character in the file;
p-> frequency = the frequency of that character in the file;
p-> left = p-> right = NULL;
```

The priority is according to the value of frequency, with lower frequencies having higher priority.

```
for (i=1; i < = n-1; i++) {   // There is no solution check; rather,
    remove(PQ, p);            // solution is obtained when i = n - 1.
    remove(PQ, q);            // Selection procedure.
    r = new nodetype;         // There is no feasibility check.
    r->left = p;
    r->right = q;
    r->frequency = p->frequency + q->frequency;
    insert(PQ, r);
}
remove(PQ, r);
return r;
```


b:5
e:10
c:12
a:16
d:17
f:25
(0)
15
(1)
27
(2)
33
(3)
52
(4)
85
(5)
0
1

Huffman’s Algorithm

- If a priority queue is implemented as a heap, it can be initialized in θ(n) time. Furthermore, each heap operation requires θ(lg n) time. Since there are n−1 passes through the for-i loop, the algorithm runs in θ(n lg n) time.