

Unit-3

Divide and Conquer with Examples Such as Sorting, Matrix Multiplication, Convex Hull and Searching.

Greedy Methods with Examples Such as Optimal Reliability Allocation, Knapsack, Minimum Spanning Trees – Prim's and Kruskal's Algorithms, Single Source Shortest Paths - Dijkstra's and Bellman Ford Algorithms.

Matrix multiplication

SQUARE-MATRIX-MULTIPLY(A, B)

```
1   $n = A.rows$ 
2  let  $C$  be a new  $n \times n$  matrix
3  for  $i = 1$  to  $n$ 
4      for  $j = 1$  to  $n$ 
5           $c_{ij} = 0$ 
6          for  $k = 1$  to  $n$ 
7               $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$ 
8  return  $C$ 
```

Strassen's algorithm

To keep things simple, when we use a divide-and-conquer algorithm to compute the matrix product $C = A \cdot B$, we assume that n is an exact power of 2 in each of the $n \times n$ matrices. We make this assumption because in each divide step, we will divide $n \times n$ matrices into four $n/2 \times n/2$ matrices, and by assuming that n is an exact power of 2, we are guaranteed that as long as $n \geq 2$, the dimension $n/2$ is an integer.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix},$$

so that we rewrite the equation $C = A \cdot B$ as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}.$$

Equation (4.10) corresponds to the four equations

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21},$$

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22},$$

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21},$$

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}.$$

SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)

```
1   $n = A.rows$ 
2  let  $C$  be a new  $n \times n$  matrix
3  if  $n == 1$ 
4       $c_{11} = a_{11} \cdot b_{11}$ 
5  else partition  $A, B$ , and  $C$  as in equations (4.9)
6       $C_{11} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{11})$ 
            $+ \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{21})$ 
7       $C_{12} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{12})$ 
            $+ \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{22})$ 
8       $C_{21} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{11})$ 
            $+ \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{21})$ 
9       $C_{22} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{12})$ 
            $+ \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{22})$ 
10 return  $C$ 
```

The key to Strassen's method is to make the recursion tree slightly less bushy. That is, instead of performing eight recursive multiplications of $n/2 \times n/2$ matrices, it performs only seven. The cost of eliminating one matrix multiplication will be several new additions of $n/2 \times n/2$ matrices, but still only a constant number of additions. As before, the constant number of matrix additions will be subsumed by Θ -notation when we set up the recurrence equation to characterize the running time.

Strassen's Matrix Multiplication

$$\begin{vmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{vmatrix} = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix}$$

$$P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$P_2 = (A_{21} + A_{22}) * B_{11}$$

$$P_3 = A_{11} * (B_{12} - B_{22})$$

$$P_4 = A_{22} * (B_{21} - B_{11})$$

$$P_5 = (A_{11} + A_{12}) * B_{22}$$

$$P_6 = (A_{21} - A_{11}) * (B_{11} + B_{12})$$

$$P_7 = (A_{12} - A_{22}) * (B_{21} + B_{22})$$

$$C_{11} = P_1 + P_4 - P_5 + P_7$$

$$C_{12} = P_3 + P_5$$

$$C_{21} = P_2 + P_4$$

$$C_{22} = P_1 + P_3 - P_2 + P_6$$

Use Strassen's algorithm to compute the matrix product

$$\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix} \begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}.$$

Greedy Algorithms

- Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step.
- A greedy algorithm always makes the choice that looks best at the moment.
- That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution

An activity-selection problem

Our first example is the problem of scheduling several competing activities that require exclusive use of a common resource, with a goal of selecting a maximum-size set of mutually compatible activities. Suppose we have a set $S = \{a_1, a_2, \dots, a_n\}$ of n proposed *activities* that wish to use a resource, such as a lecture hall, which can serve only one activity at a time. Each activity a_i has a *start time* s_i and a *finish time* f_i , where $0 \leq s_i < f_i < \infty$. If selected, activity a_i takes place during the half-open time interval $[s_i, f_i)$. Activities a_i and a_j are *compatible* if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. That is, a_i and a_j are compatible if $s_i \geq f_j$ or $s_j \geq f_i$. In the *activity-selection problem*, we wish to select a maximum-size subset of mutually compatible activities. We assume that the activities are sorted in monotonically increasing order of finish time:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

For this example, the subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities. It is not a maximum subset, however, since the subset $\{a_1, a_4, a_8, a_{11}\}$ is larger. In fact, $\{a_1, a_4, a_8, a_{11}\}$ is a largest subset of mutually compatible activities; another largest subset is $\{a_2, a_4, a_9, a_{11}\}$.

Elements of the greedy strategy





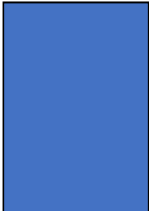
- A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices.
- At each decision point, the algorithm makes choice that seems best at the moment.

Optimal substructure

- A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to sub problems

0-1 Knapsack problem:

This is a knapsack
Max weight: $W = 20$

Items	Weight w_i	Benefit value b_i
	2	3
	3	4
	4	5
	5	8
	9	10

- Fractional knapsack problem, the setup is the same, but the thief can take fractions of items, rather than having to make a binary (0-1) choice for each item.
- You can think of an item in the 0-1 knapsack problem as being like a gold coin and an item in the fractional knapsack problem as more like gold dust

- Both knapsack problems exhibit the optimal-substructure property.
- For the 0-1 problem, consider the most valuable load that weighs at most W pounds.
- If we remove item j from this load, the remaining load must be the most valuable load weighing at most $W - w_j$ that the thief can take from the $n - 1$ original items excluding j .
- For the comparable fractional problem, consider that if we remove a weight w of one item j from the optimal load, the remaining load must be the most valuable load weighing at most $W - w$ that the thief can take from the $n - 1$ original items plus $w_j - w$ pounds of item j .

Although the problems are similar, we can solve the fractional knapsack problem by a greedy strategy, but we cannot solve the 0-1 problem by such a strategy. To solve the fractional problem, we first compute the value per pound v_i/w_i for each item. Obeying a greedy strategy, the thief begins by taking as much as possible of the item with the greatest value per pound. If the supply of that item is exhausted and he can still carry more, he takes as much as possible of the item with the next greatest value per pound, and so forth, until he reaches his weight limit W . Thus,

To see that this greedy strategy does not work for the 0-1 knapsack problem, consider the problem instance illustrated in Figure 16.2(a). This example has 3 items and a knapsack that can hold 50 pounds. Item 1 weighs 10 pounds and is worth 60 dollars. Item 2 weighs 20 pounds and is worth 100 dollars. Item 3 weighs 30 pounds and is worth 120 dollars. Thus, the value per pound of item 1 is 6 dollars per pound, which is greater than the value per pound of either item 2 (5 dollars per pound) or item 3 (4 dollars per pound). The greedy strategy, therefore, would take item 1 first. As you can see from the case analysis in Figure 16.2(b), however, the optimal solution takes items 2 and 3, leaving item 1 behind. The two possible solutions that take item 1 are both suboptimal.

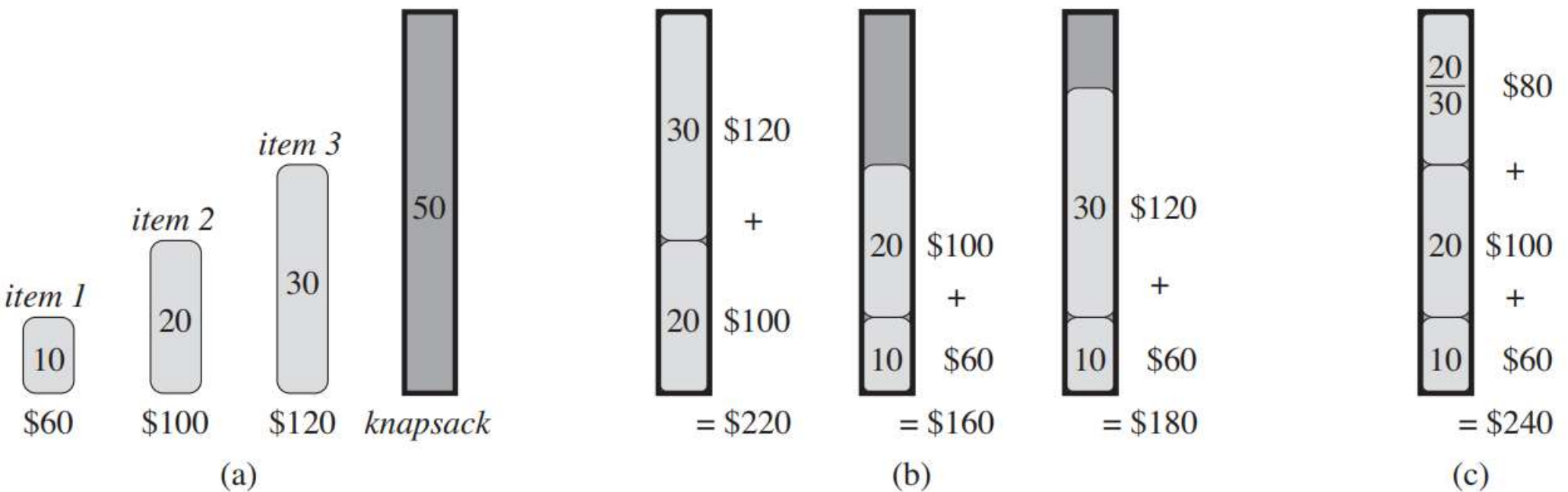


Figure 16.2 An example showing that the greedy strategy does not work for the 0-1 knapsack problem. (a) The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. (b) The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. (c) For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

Huffman codes

Huffman codes compress data very effectively: savings of 20% to 90% are typical, depending on the characteristics of the data being compressed. We consider the data to be a sequence of characters. Huffman's greedy algorithm uses a table giving how often each character occurs (i.e., its frequency) to build up an optimal way of representing each character as a binary string.

Suppose we have a 100,000-character data file that we wish to store compactly. We observe that the characters in the file occur with the frequencies given by Figure 16.3. That is, only 6 different characters appear, and the character **a** occurs 45,000 times.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Figure 16.3 A character-coding problem. A data file of 100,000 characters contains only the characters a–f, with the frequencies indicated. If we assign each character a 3-bit codeword, we can encode the file in 300,000 bits. Using the variable-length code shown, we can encode the file in only 224,000 bits.

codeword. If we use a *fixed-length code*, we need 3 bits to represent 6 characters: a = 000, b = 001, ..., f = 101. This method requires 300,000 bits to code the entire file. Can we do better?

We have many options for how to represent such a file of information. Here, we consider the problem of designing a *binary character code* (or *code* for short)

A *variable-length code* can do considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent characters long codewords. Figure 16.3 shows such a code; here the 1-bit string 0 represents a, and the 4-bit string 1100 represents f. This code requires

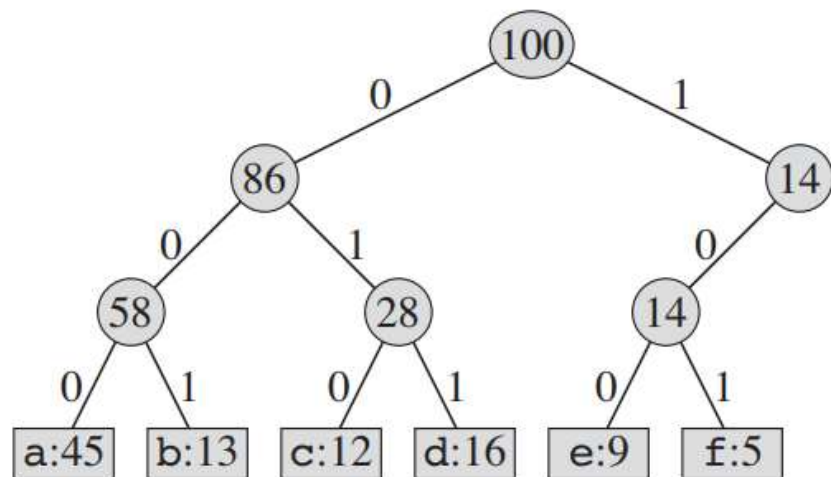
$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1,000 = 224,000 \text{ bits}$$

to represent the file, a savings of approximately 25%. In fact, this is an optimal character code for this file, as we shall see.

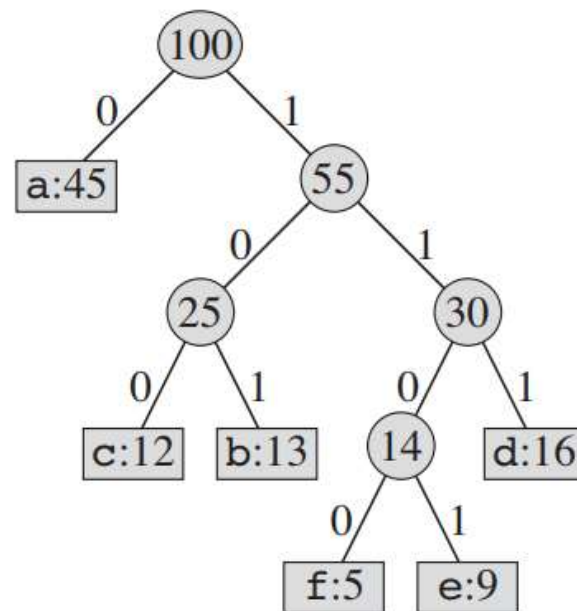
Prefix codes

We consider here only codes in which no codeword is also a prefix of some other codeword. Such codes are called *prefix codes*.³ Although we won't prove it here, a prefix code can always achieve the optimal data compression among any character code, and so we suffer no loss of generality by restricting our attention to prefix codes.

Encoding is always simple for any binary character code; we just concatenate the codewords representing each character of the file. For example, with the variable-length prefix code of Figure 16.3, we code the 3-character file `abc` as $0 \cdot 101 \cdot 100 = 0101100$, where “ \cdot ” denotes concatenation.



(a)



(b)

Figure 16.4 Trees corresponding to the coding schemes in Figure 16.3. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree. (a) The tree corresponding to the fixed-length code $a = 000, \dots, f = 101$. (b) The tree corresponding to the optimal prefix code $a = 0, b = 101, \dots, f = 1100$.

Constructing a Huffman code

In the pseudocode that follows, we assume that C is a set of n characters and that each character $c \in C$ is an object with an attribute $c.freq$ giving its frequency. The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner. It begins with a set of $|C|$ leaves and performs a sequence of $|C| - 1$ “merging” operations to create the final tree. The algorithm uses a min-priority queue Q , keyed on the $freq$ attribute, to identify the two least-frequent objects to merge together. When we merge two objects, the result is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

HUFFMAN(C)

1 $n = |C|$

2 $Q = C$

3 **for** $i = 1$ **to** $n - 1$

4 allocate a new node z

5 $z.left = x = \text{EXTRACT-MIN}(Q)$

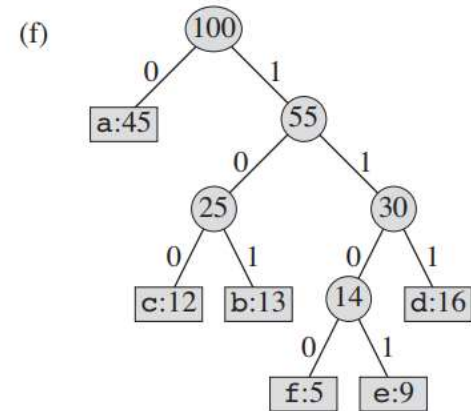
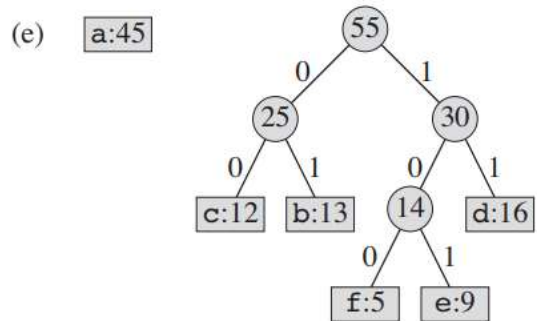
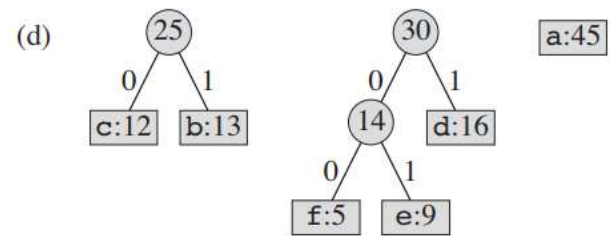
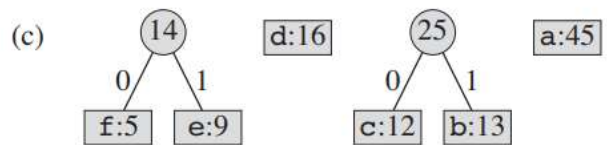
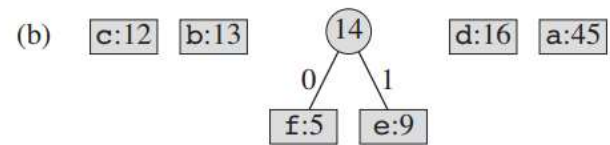
6 $z.right = y = \text{EXTRACT-MIN}(Q)$

7 $z.freq = x.freq + y.freq$

8 INSERT(Q, z)

9 **return** EXTRACT-MIN(Q) // return the root of the tree

(a) f:5 e:9 c:12 b:13 d:16 a:45



What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers?

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

A task-scheduling problem

- An interesting problem that we can solve is the problem of optimally scheduling unit-time tasks on a single processor, where each task has a deadline, along with a penalty paid if the task misses its deadline.
- A unit-time task is a job, such as a program to be run on a computer, that requires exactly one unit of time to complete.
- Given a finite set S of unit-time tasks, ***A schedule for S is a permutation of S specifying the order in which these tasks are*** to be performed.
- The first task in the schedule begins at time 0 and finishes at time 1, the second task begins at time 1 and finishes at time 2, and so on.

The problem of *scheduling unit-time tasks with deadlines and penalties for a single processor* has the following inputs:

- a set $S = \{a_1, a_2, \dots, a_n\}$ of n unit-time tasks;
- a set of n integer **deadlines** d_1, d_2, \dots, d_n , such that each d_i satisfies $1 \leq d_i \leq n$ and task a_i is supposed to finish by time d_i ; and
- a set of n nonnegative weights or **penalties** w_1, w_2, \dots, w_n , such that we incur a penalty of w_i if task a_i is not finished by time d_i , and we incur no penalty if a task finishes by its deadline.

	Task						
a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

Figure 16.7 demonstrates an example of the problem of scheduling unit-time tasks with deadlines and penalties for a single processor. In this example, the greedy algorithm selects, in order, tasks a_1 , a_2 , a_3 , and a_4 , then rejects a_5 (because $N_4(\{a_1, a_2, a_3, a_4, a_5\}) = 5$) and a_6 (because $N_4(\{a_1, a_2, a_3, a_4, a_6\}) = 5$), and finally accepts a_7 . The final optimal schedule is

$$\langle a_2, a_4, a_1, a_3, a_7, a_5, a_6 \rangle ,$$

which has a total penalty incurred of $w_5 + w_6 = 50$.

Minimum Spanning Trees

- Assume that we have a connected, undirected graph $G = (V, E)$ with a weight function $w : E \rightarrow \mathbf{R}$, **and we wish to find a minimum spanning tree for G .**
- we will examine two algorithms for solving the minimum spanning-tree problem: **Kruskal's** algorithm and **Prim's** algorithm.

Growing a minimum spanning tree

- The two algorithms we consider in this **use a greedy approach** to the problem, although they differ in how they apply this approach.
- This greedy strategy is captured by the following “generic” algorithm, which grows the minimum spanning tree one edge at a time.
- The algorithm manages a set of edges A , *maintaining the following loop invariant:*
Prior to each iteration, A is a subset of some minimum spanning tree.

Growing a minimum spanning tree

- At each step, we determine an edge (u, v) that can be added to A without violating this invariant, in the sense that $A \cup \{(u, v)\}$ is also a subset of a minimum spanning tree.
- We call such an edge a safe edge for A , since it can be safely added to A while maintaining the invariant.

Growing a minimum spanning tree

GENERIC-MST(G, w)

1 $A \leftarrow \emptyset$

2 **while** A does not form a spanning tree

3 **do** find an edge (u, v) that is safe for A

4 $A \leftarrow A \cup \{(u, v)\}$

5 **return** A

Kruskal algorithm

- In Kruskal's algorithm, the set A is a forest. The safe edge added to A is always a *least-weight edge in the graph that connects two distinct components*.
- Kruskal's algorithm is a greedy algorithm, because at each step it adds to the forest an edge of least possible weight.
- It uses a disjoint-set data structure to maintain several disjoint sets of elements. Each set contains the vertices in a tree of the current forest.

Disjoint-set data structure

- A disjoint-set data structure maintains a collection $S = \{S_1, S_2, \dots, S_k\}$ of disjoint dynamic sets. Each set is identified by a representative, which is some member of the set.
- there may be a pre specified rule for choosing the representative, such as choosing the smallest member in the set.
- Each element of a set is represented by an object. Letting x *denote an object*, we wish to support the following operations:

Disjoint-set data structure

- **MAKE-SET**(x) creates a new set whose only member (and thus representative) is x . Since the sets are disjoint, we require that x not already be in some other set.
- **UNION**(x, y) unites the dynamic sets that contain x and y , say S_x and S_y , into a new set that is the union of these two sets. The two sets are assumed to be disjoint prior to the operation. Since we require the sets in the collection to be disjoint, we “**destroy**” sets S_x and S_y , *removing them* from the collection S .
- **FIND-SET**(x) returns a pointer to the representative of the (unique) set containing x .

An application of disjoint-set data structures

- Determining the connected components of an undirected graph. The procedure **CONNECTED-COMPONENTS** that follows uses the disjoint-set operations to compute the connected components of a graph.
- Once **CONNECTED-COMPONENTS** has been run as a preprocessing step, the procedure **SAMECOMPONENT** answers queries about whether two vertices are in the same connected component

Algorithm

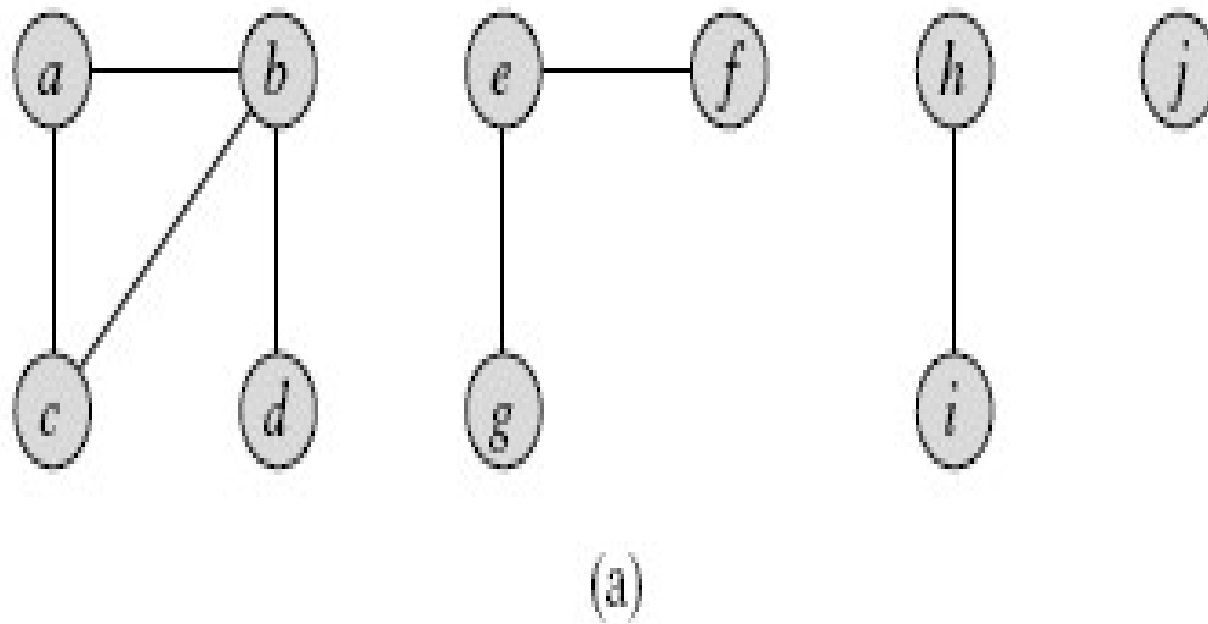
CONNECTED-COMPONENTS(G)

```
1 for each vertex  $v \in V[G]$ 
2     do MAKE-SET( $v$ )
3 for each edge  $(u, v) \in E[G]$ 
4     do if FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ )
5         then UNION( $u, v$ )
```

SAME-COMPONENT(u, v)

```
1 if FIND-SET( $u$ ) = FIND-SET( $v$ )
2     then return TRUE
3     else return FALSE
```

Example



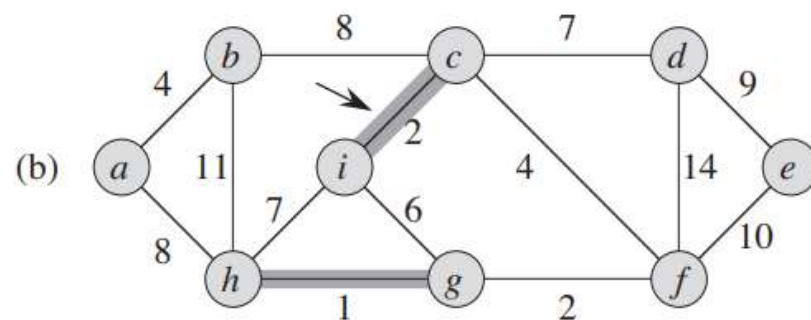
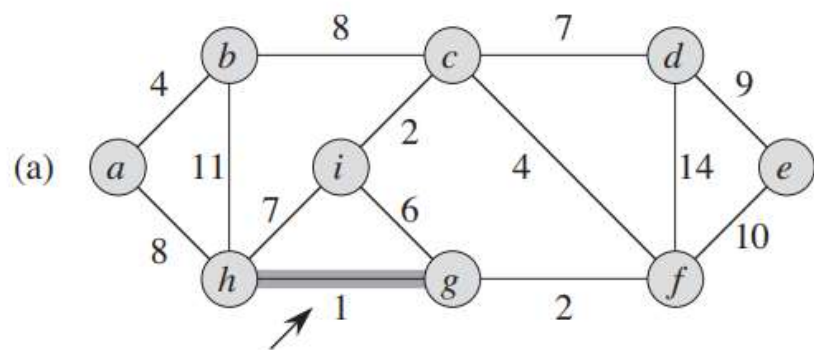
Example

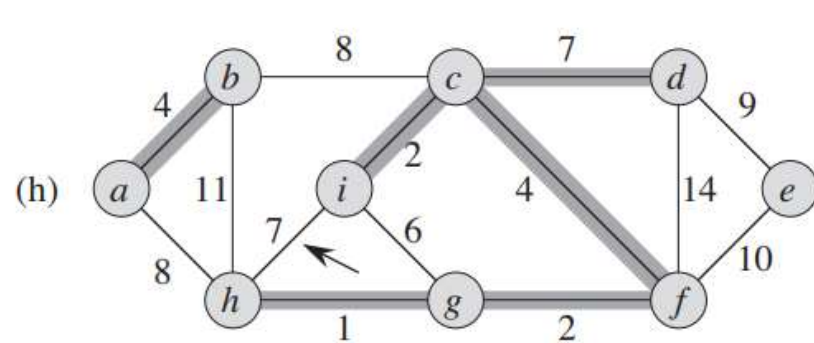
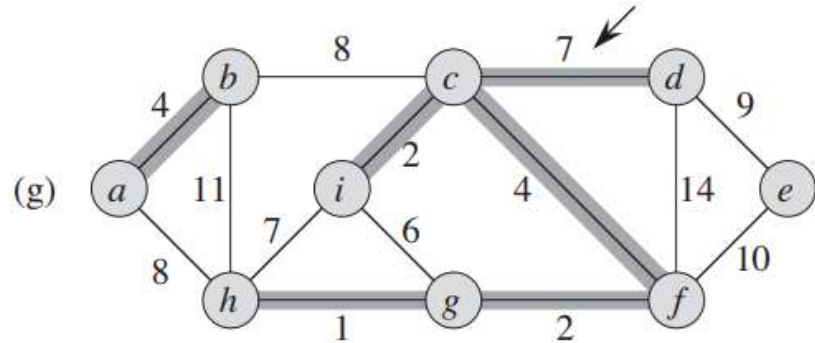
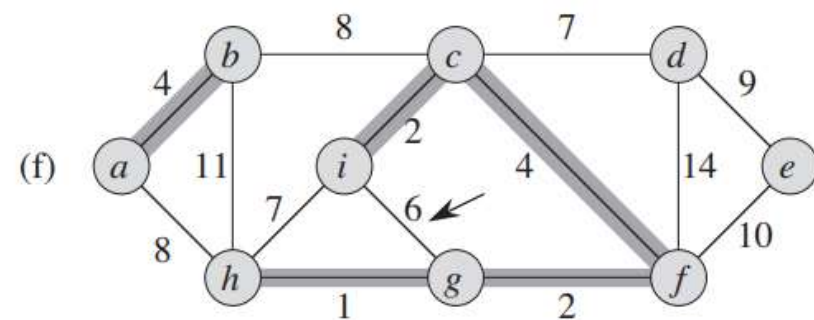
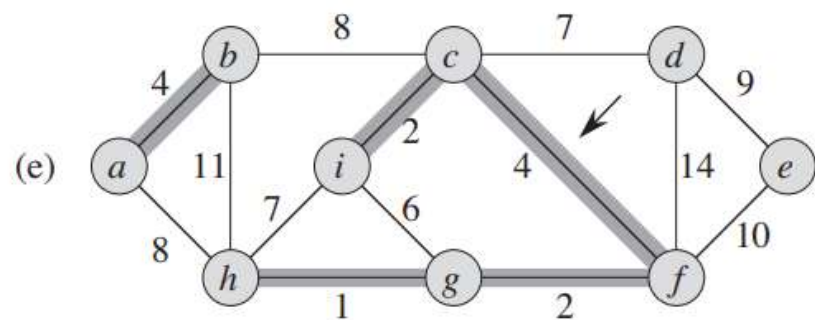
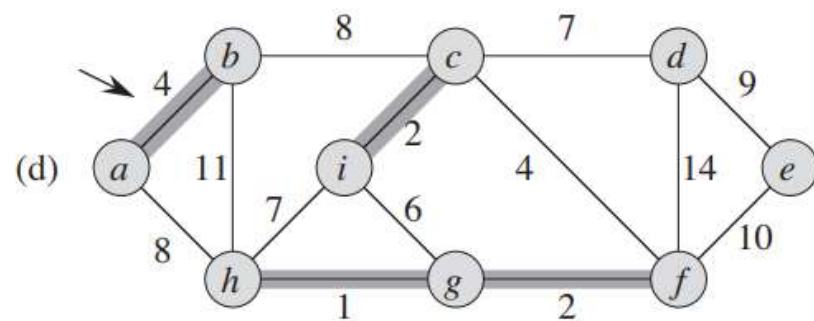
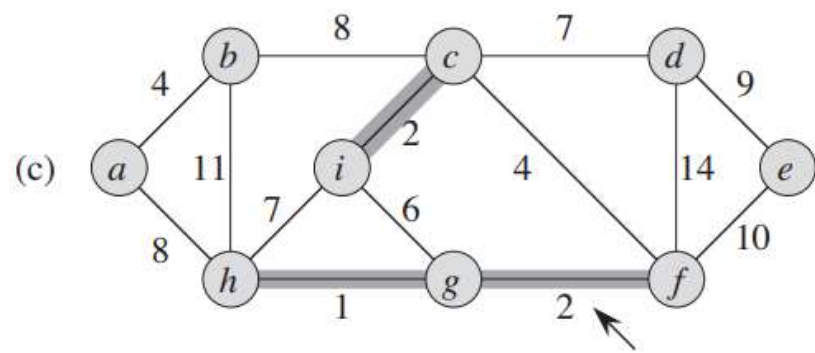
Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(e,f)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

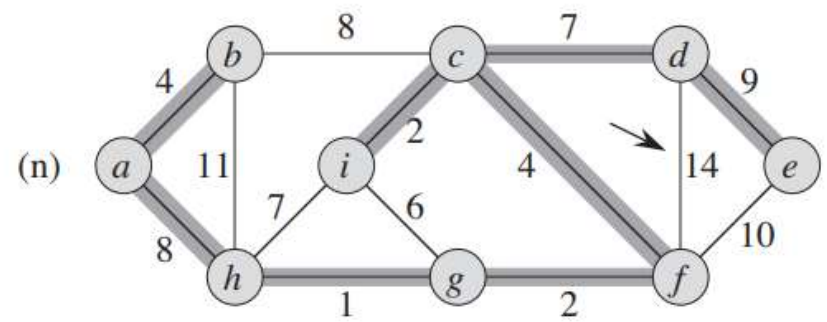
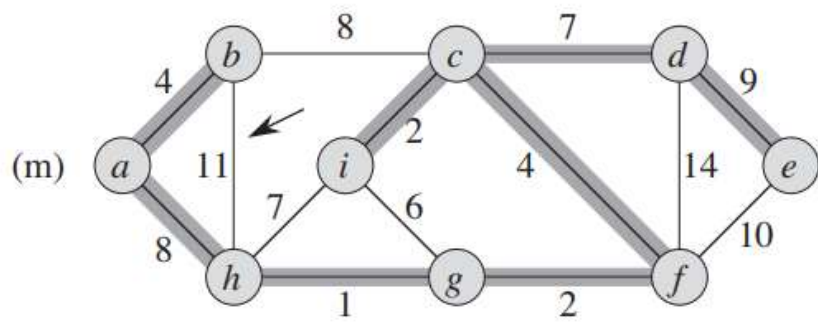
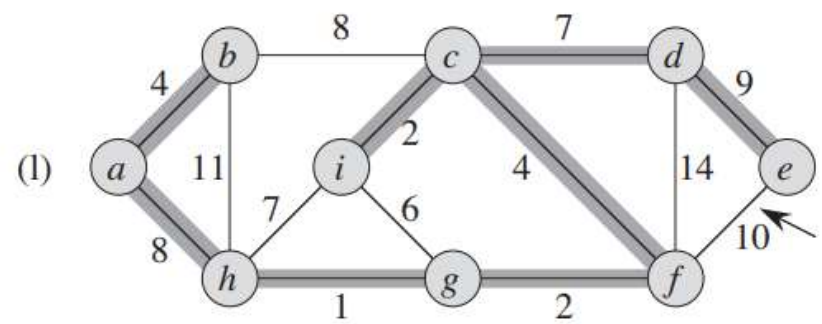
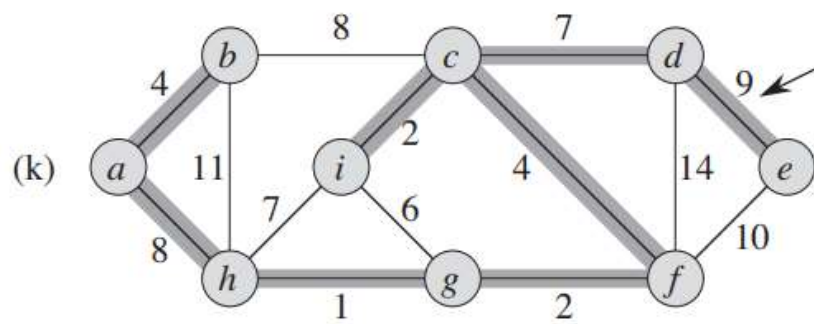
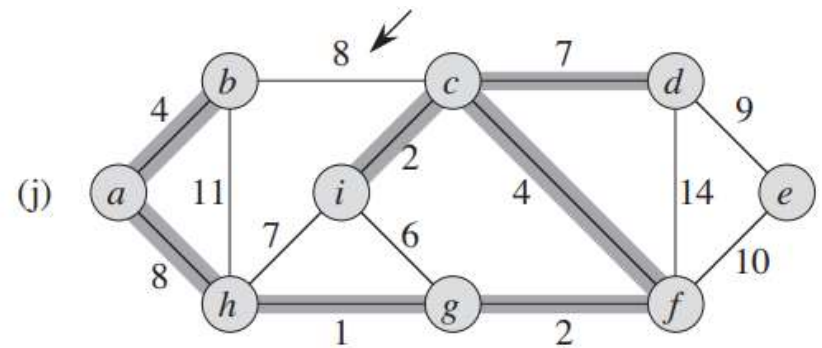
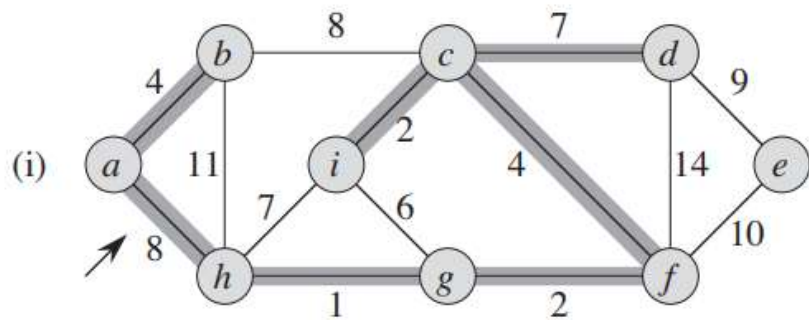
Kruskal's algorithm

MST-KRUSKAL(G, w)

```
1   $A \leftarrow \emptyset$ 
2  for each vertex  $v \in V[G]$ 
3      do MAKE-SET( $v$ )
4  sort the edges of  $E$  into nondecreasing order by weight  $w$ 
5  for each edge  $(u, v) \in E$ , taken in nondecreasing order by weight
6      do if FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ )
7          then  $A \leftarrow A \cup \{(u, v)\}$ 
8              UNION( $u, v$ )
9  return  $A$ 
```







Prim's algorithm

- Prim's algorithm has the property that the edges in the set A always form a single tree.
- the tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V .
- At each step, a light edge is added to the tree A that connects A to an isolated vertex of $G_A = (V, A)$.
- This strategy is greedy since the tree is augmented at each step with an edge that contributes the minimum amount possible to the tree's weight.

Prim's algorithm

- In the pseudocode below, the connected graph G and the root r of the minimum spanning tree to be grown are inputs to the algorithm.
- During execution of the algorithm, all vertices that are not in the tree reside in a **min-priority queue Q** based on a key field.
- For each vertex v , $\text{key}[v]$ is the minimum weight of any edge connecting v to a vertex in the tree;
- by convention, $\text{key}[v] = \infty$ if there is no such edge. The field $\pi[v]$ names the parent of v in the tree

Prim's algorithm

- During the algorithm, the set A from *GENERIC-MST* is kept implicitly as

$$A = \{(v, \pi[v]) : v \in V - \{r\} - Q\}.$$

- When the algorithm terminates, the min-priority queue Q is empty; the minimum spanning tree A for G is thus

$$A = \{(v, \pi[v]) : v \in V - \{r\}\}.$$

Prim's Algorithm

MST-PRIM(G, w, r)

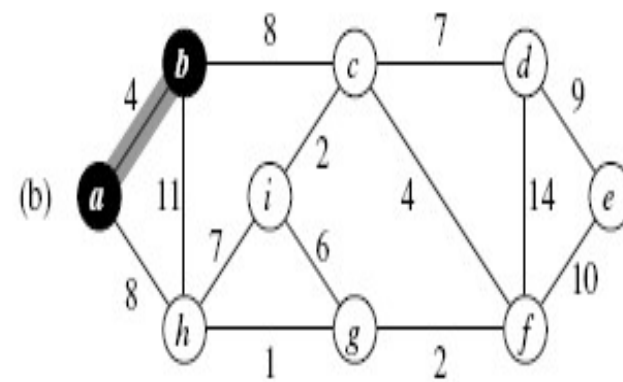
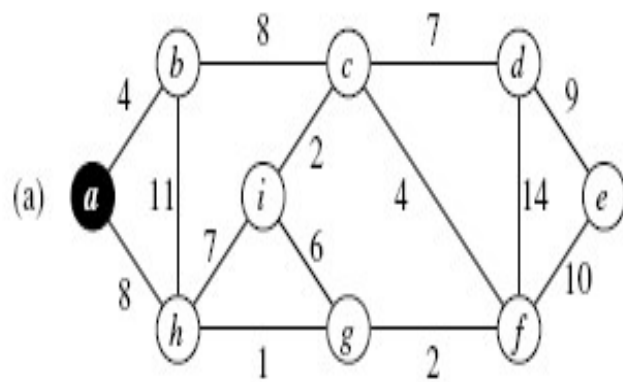
```
1 for each  $u \in V[G]$ 
2     do  $key[u] \leftarrow \infty$ 
3      $\pi[u] \leftarrow NIL$ 
4  $key[r] \leftarrow 0$ 
5  $Q \leftarrow V[G]$ 
6 while  $Q \neq \emptyset$ 
7     do  $u \leftarrow EXTRACT-MIN(Q)$ 
8     for each  $v \in Adj[u]$ 
9     do if  $v \in Q$  and  $w(u, v) < key[v]$ 
10    then  $\pi[v] \leftarrow u$ 
11     $key[v] \leftarrow w(u, v)$ 
```

Salient features of algorithm

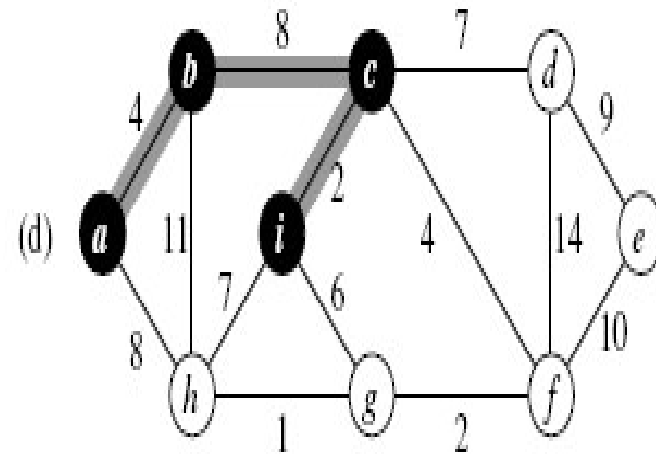
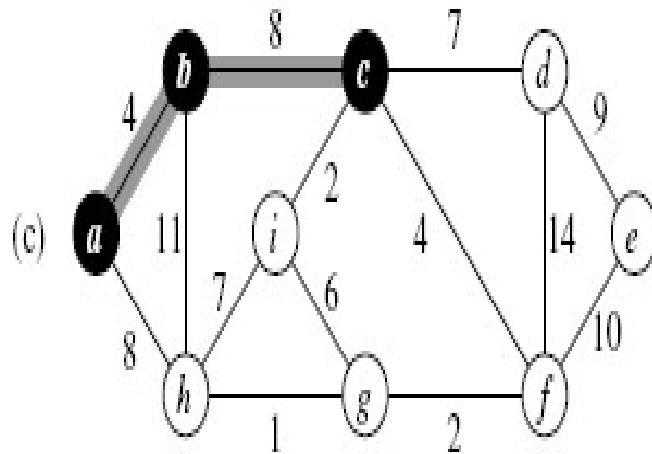
The algorithm maintains the following three-part loop invariant:

- Prior to each iteration of the **while loop of lines 6–11**,
 1. $A = \{(v, \pi[v]) : v \in V - \{r\} - Q\}$.
 2. The vertices already placed into the minimum spanning tree are those in $V - Q$.
 3. For all vertices $v \in Q$, if $\pi[v] = \text{NIL}$, then $\text{key}[v] < \infty$ and $\text{key}[v]$ is the weight of a light edge $(v, \pi[v])$ connecting v to some vertex already placed into the minimum spanning tree.

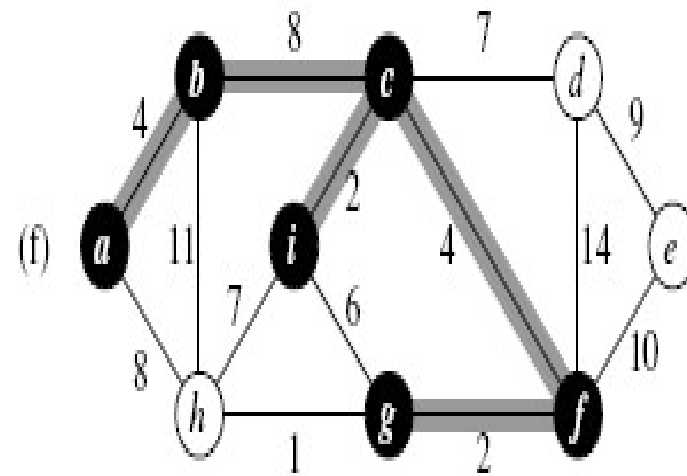
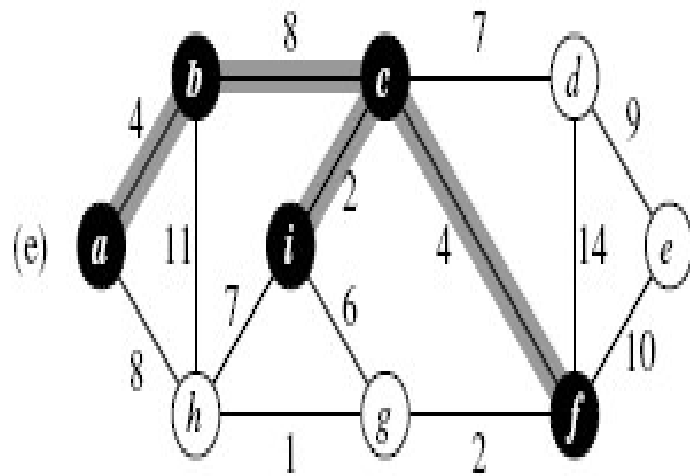
Example



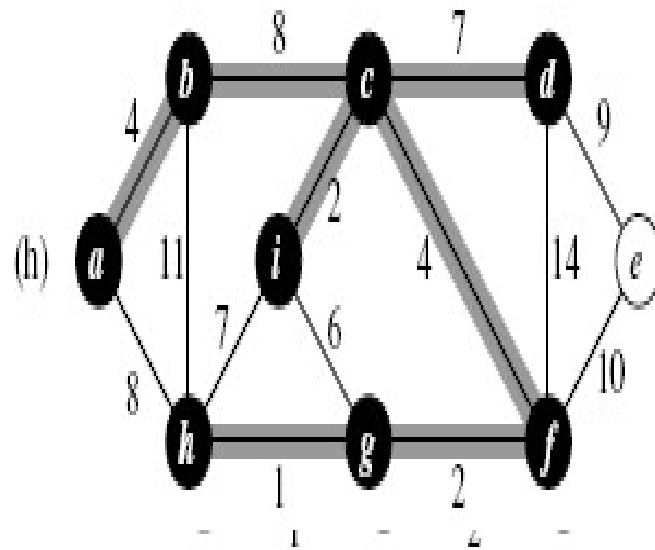
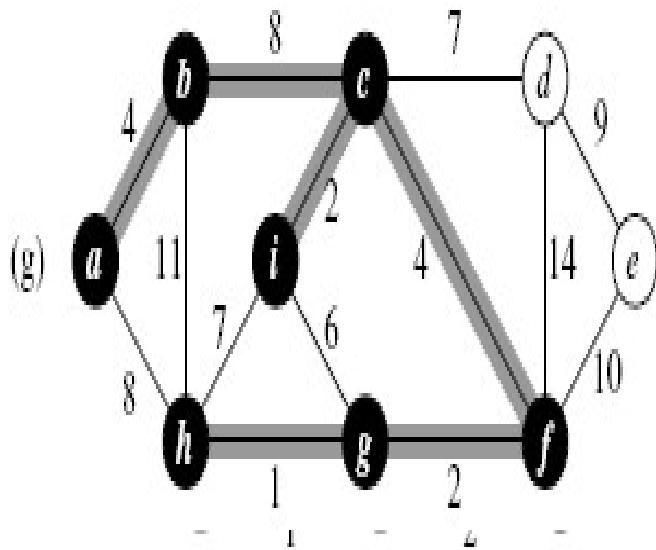
Example



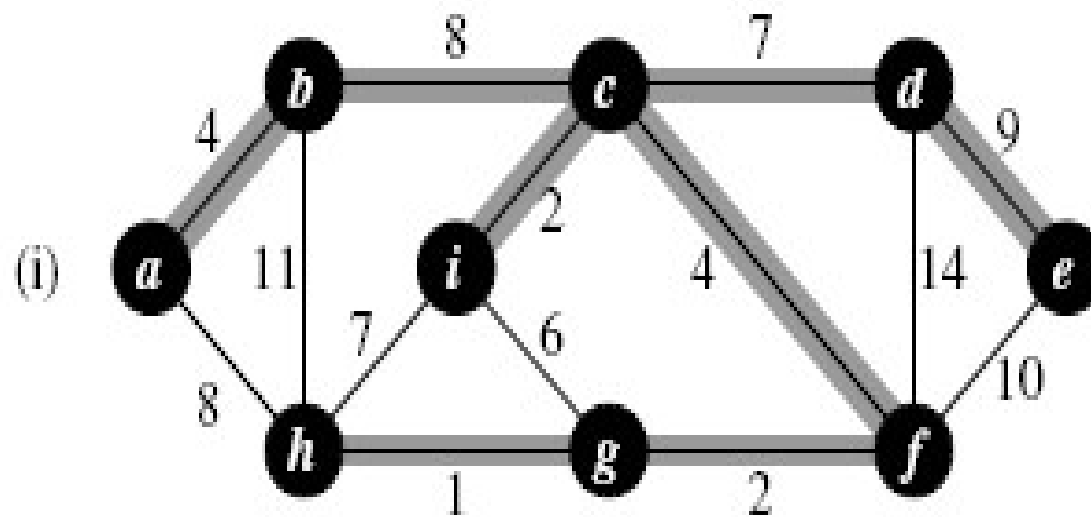
Example



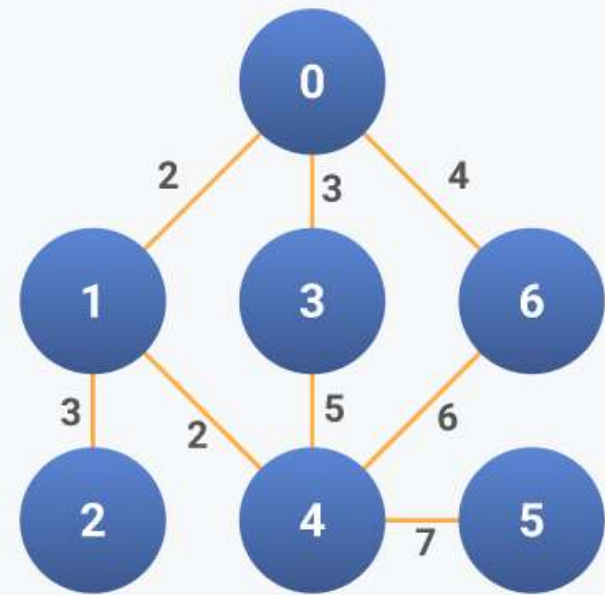
Example



Example



- Solve for given graph using Prim's And Kruskal's



Graph

Analysis

- If Q is implemented as a binary min-heap we can use the BUILD-MIN-HEAP procedure to perform the initialization in lines in $O(V)$ time.
- The body of the while loop is executed $|V|$ times, and since each EXTRACT-MIN operation takes $O(\lg V)$ time, the total time for all calls to EXTRACT-MIN is $O(V \lg V)$.
- The for loop in lines 8–11 is executed $O(E)$ times
- altogether, since the sum of the lengths of all adjacency lists is $2|E|$.
- The assignment in line 11 involves an implicit DECREASE-KEY operation on the min-heap, which can be implemented in a binary min-heap in $O(\lg V)$ time.
- Thus, the total time for Prim's algorithm is $O(V \lg V + E \lg V) = O(E \lg V)$, which is asymptotically the *same as for our* implementation of Kruskal's algorithm.

