B.Tech. (Fifth Semester) Examination 2013

Analysis and Design of Algorithm (IT3105N)
(Information Technology)

Model Answer

....

Section A

Q.1 - **(20×1=10)**

1. Merge Sort uses……………………approach to algorithm design.
   Ans: Divide and conquer
2. Rod Cutting problem uses……………………approach to algorithm design.
   Ans: Dynamic Programming
3. Edit distance problem uses……………………approach to algorithm design.
   Ans: Dynamic Programming
4. Big O notation gives _____________ bound of the function.
   Ans: Upper bound
5. ________ Notation works on lower bound of the function.
   Ans: Big omega
6. ________ Notation is bounded by both upper and lower bound.
   Ans: Theta Notation
7. The worst case complexity of Merge sort algorithm is…………………….. .
   Ans: O (n log n)
8. Famous Algorithm used to find All-pair shortest path is _____________________.
   Ans: Flloyd-Warshall Algorithm.
9. Dijkstra Algorithm is a ________________ Algorithm~~, used to find All-pair shortest path~~.
   Ans: Single Source Shortest path.
10. In All-pair shortest path we create two matrices, they are ____________and ________.
    Ans: Weight matrics and predecessor matrics.
11. Kruskal Algorithm is used for ______________.
    Ans: Minimum spanning tree.
12. According …….………………theory at every stage each node has a bound.
    Ans: Branch and bound
13. To remove problem with recusion stack in devide and conqer ________________.

Ans: sorting the smaller side and simultaneously removing the tail recursion.

14. NP Complete problems are ……………………

Ans: NP problems for which L$\in$NP and all problems in NP can be reduced to P

15. To implement Tree we need _________ Data Structure.

Ans: Linked List

16. Prim's Algorithm is used for ______________.

Ans: Minimum Spanning Tree

17. Floyd-Warshall Algorithm is used for ________.

Ans: All pair shortest Path

18. Time Complexity of Binary Search in average case is __________.

Ans: O ($\log_2$ n)

19. Collection of similar type of data elements is known as _____________.

Ans: Array

20. The items can be broken into smaller pieces to fill knapsack, is known as …………..…… knapsack problem.

Ans: Fractional Knapsack problem

Section B

Note: Attempt any one question from each unit. Each question carries 8 marks.

UNIT 1

Q 1. Explain Quick sort algorithm. What are the two drawbacks of quick sort algorithm and how they can be removed?

Ans:

The algorithm. Quicksort follows the general paradigm of divide-and-conquer, which means it **divides** the unsorted array into two, it **recurses** on the two pieces, and it finally **combines** the two sorted pieces to obtain the sorted array. An interesting feature of quicksort is that the divide step separates small from large items. As a consequence, combining the sorted pieces happens automatically without doing anything extra.

```
void QUICKSORT(int $\ell, r$)
  if $\ell < r$ then $m$ = SPLIT($\ell, r$);
                 QUICKSORT($\ell, m-1$);
                 QUICKSORT($m+1, r$)
  endif.
```

We assume the items are stored in $A[0..n-1]$. The array is sorted by calling QUICKSORT$(0, n-1)$.

Splitting. The performance of quicksort depends heavily on the performance of the split operation. The effect of splitting from ℓ to r is:

- $x = A[\ell]$ is moved to its correct location at $A[m]$;
- no item in $A[\ell..m-1]$ is larger than x;
- no item in $A[m+1..r]$ is smaller than x.

Figure 1 illustrates the process with an example. The nine items are split by moving a pointer i from left to right and another pointer j from right to left. The process stops when i and j cross. To get splitting right is a bit delicate, in particular in special cases. Make sure the algorithm is correct for (i) x is smallest item, (ii) x is largest item, (iii) all items are the same.

```
int SPLIT(int $\ell, r$)
  $x = A[\ell]$; $i = \ell$; $j = r + 1$;
  repeat repeat $i$++ until $x \le A[i]$;
         repeat $j$-- until $x \ge A[j]$;
         if $i < j$ then SWAP($i, j$) endif
  until $i \ge j$;
  SWAP($\ell, j$); return $j$.
```

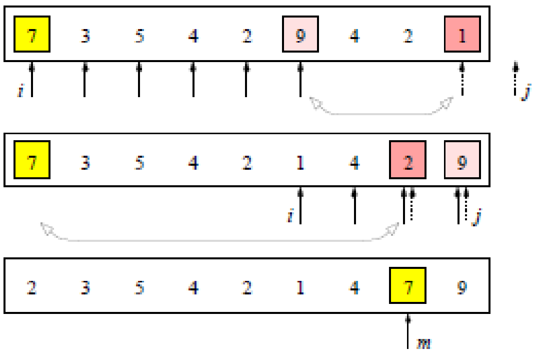


Figure 1: First, i and j stop at items 9 and 1, which are then swapped. Second, i and j cross and the pivot, 7, is swapped with item 2.

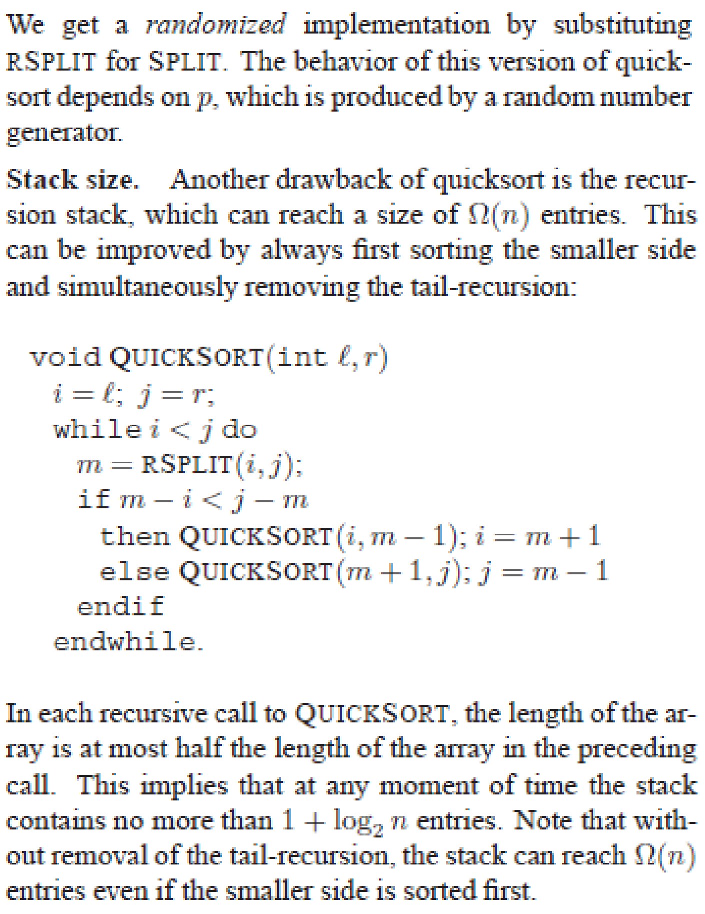
Special cases (i) and (iii) are ok but case (ii) requires a stopper at $A[r+1]$. This stopper must be an item at least as large as x. If $r < n-1$ this stopper is automatically given. For $r = n-1$, we create such a stopper by setting $A[n] = +\infty$.

Randomization. One of the drawbacks of quicksort, as described until now, is that it is slow on rather common almost sorted sequences. The reason are pivots that tend to create unbalanced splittings. Such pivots tend to occur in practice more often than one might expect. Human and often also machine generated data is frequently biased towards certain distributions (in this case, permutations), and it has been said that 80% of the time or more, sorting is done on either already sorted or almost sorted files. Such situations can often be helped by transferring the algorithm's dependence on the input data to internally made random choices. In this particular case, we use randomization to make the choice of the pivot independent of the input data. Assume RANDOM(ℓ, r) returns an integer $p \in [\ell, r]$ with uniform probability:

$$\text{Prob}[\text{RANDOM}(\ell, r) = p] = \frac{1}{r - \ell + 1}$$

for each $\ell \le p \le r$. In other words, each $p \in [\ell, r]$ is equally likely. The following algorithm splits the array with a random pivot:

```
int RSPLIT(int $\ell, r$)
  $p$ = RANDOM($\ell, r$); SWAP($\ell, p$);
  return SPLIT($\ell, r$).
```



We get a *randomized* implementation by substituting RSPLIT for SPLIT. The behavior of this version of quicksort depends on p, which is produced by a random number generator.

Stack size. Another drawback of quicksort is the recursion stack, which can reach a size of $\Omega(n)$ entries. This can be improved by always first sorting the smaller side and simultaneously removing the tail-recursion:

```
void QUICKSORT(int $\ell, r$)
  $i = \ell$; $j = r$;
  while $i < j$ do
    $m = $ RSPLIT($i, j$);
    if $m - i < j - m$
      then QUICKSORT($i, m - 1$); $i = m + 1$
      else QUICKSORT($m + 1, j$); $j = m - 1$
    endif
  endwhile.
```

In each recursive call to QUICKSORT, the length of the array is at most half the length of the array in the preceding call. This implies that at any moment of time the stack contains no more than $1 + \log_2 n$ entries. Note that without removal of the tail-recursion, the stack can reach $\Omega(n)$ entries even if the smaller side is sorted first.

Q 2. Write and explain algorithm of Selection sort with proper example..

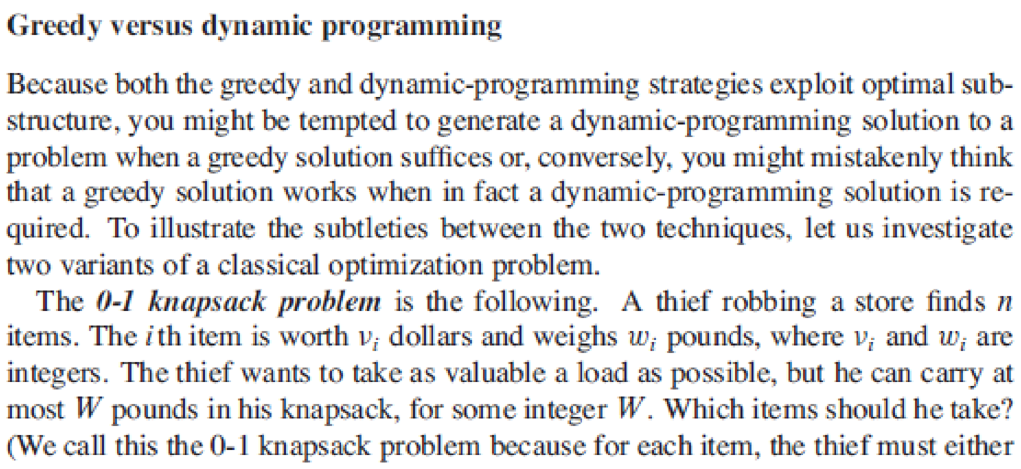
ANS: answer should contain following points

1. algorithm of Selection sort
2. explanation of the algorithm
3. one proper example with diagram.

UNIT 2

Q 3. Explain 0/1 Knapsack problem with proper example and prove that dynamic programming is better for 0/1 knapsack problem.

Ans:



Greedy versus dynamic programming

Because both the greedy and dynamic-programming strategies exploit optimal substructure, you might be tempted to generate a dynamic-programming solution to a problem when a greedy solution suffices or, conversely, you might mistakenly think that a greedy solution works when in fact a dynamic-programming solution is required. To illustrate the subtleties between the two techniques, let us investigate two variants of a classical optimization problem.

The ***0-1 knapsack problem*** is the following. A thief robbing a store finds n items. The ith item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack, for some integer W. Which items should he take? (We call this the 0-1 knapsack problem because for each item, the thief must either take it or leave it behind; he cannot take a fractional amount of an item or take an item more than once.)

In the ***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 ingot 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, by sorting the items by value per pound, the greedy algorithm runs in $O(n \lg n)$ time. We leave the proof that the fractional knapsack problem has the greedy-choice property as Exercise 16.2-1.

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.

For the comparable fractional problem, however, the greedy strategy, which takes item 1 first, does yield an optimal solution, as shown in Figure 16.2(c). Taking item 1 doesn't work in the 0-1 problem because the thief is unable to fill his knapsack to capacity, and the empty space lowers the effective value per pound of his load. In the 0-1 problem, when we consider whether to include an item in the knapsack, we must compare the solution to the subproblem that includes the item with the solution to the subproblem that excludes the item before we can make the

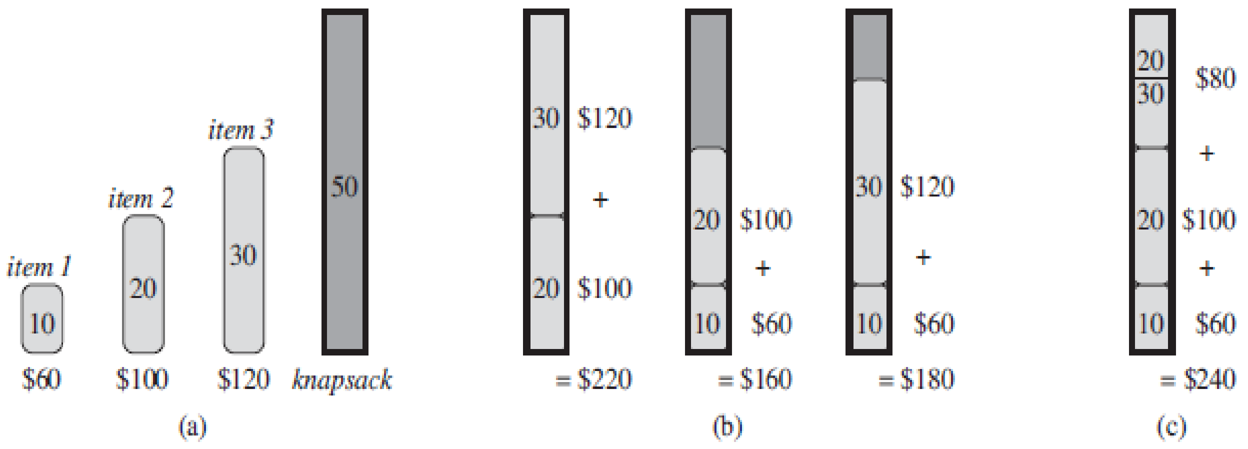
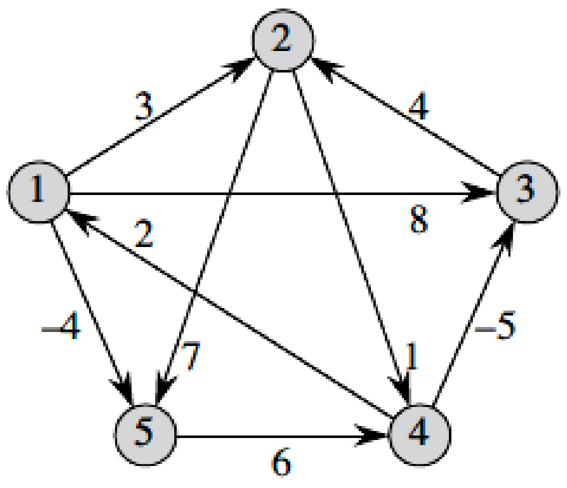


Figure 16.2 An example showing that the greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though 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.

Q 4. Write and Explain floyed-warshall Algorithm. Find out the weight metrics and predecessor metrics for given graph using floyed-warshall Algorithm:



The Floyd-Warshall algorithm

In this section, we shall use a different dynamic-programming formulation to solve the all-pairs shortest-paths problem on a directed graph $G = (V, E)$. The resulting algorithm, known as the ***Floyd-Warshall algorithm***, runs in $\Theta(V^3)$ time. As before, negative-weight edges may be present, but we assume that there are no negative-weight cycles. As in Section 25.1, we follow the dynamic-programming process to develop the algorithm. After studying the resulting algorithm, we present a similar method for finding the transitive closure of a directed graph.

The structure of a shortest path

In the Floyd-Warshall algorithm, we characterize the structure of a shortest path differently from how we characterized it in Section 25.1. The Floyd-Warshall algorithm considers the intermediate vertices of a shortest path, where an ***intermediate*** vertex of a simple path $p = \langle v_1, v_2, \ldots, v_l \rangle$ is any vertex of p other than v_1 or v_l, that is, any vertex in the set $\{v_2, v_3, \ldots, v_{l-1}\}$.

The Floyd-Warshall algorithm relies on the following observation. Under our assumption that the vertices of G are $V = \{1, 2, \ldots, n\}$, let us consider a subset $\{1, 2, \ldots, k\}$ of vertices for some k. For any pair of vertices $i, j \in V$, consider all paths from i to j whose intermediate vertices are all drawn from $\{1, 2, \ldots, k\}$, and let p be a minimum-weight path from among them. (Path p is simple.) The Floyd-Warshall algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. The relationship depends on whether or not k is an intermediate vertex of path p.

- If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set $\{1, 2, \ldots, k-1\}$. Thus, a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$ is also a shortest path from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$.
- If k is an intermediate vertex of path p, then we decompose p into $i \overset{p_1}{\leadsto} k \overset{p_2}{\leadsto} j$, as Figure 25.3 illustrates. By Lemma 24.1, p_1 is a shortest path from i to k with all intermediate vertices in the set $\{1, 2, \ldots, k\}$. In fact, we can make a slightly stronger statement. Because vertex k is not an intermediate vertex of path p_1, all intermediate vertices of p_1 are in the set $\{1, 2, \ldots, k-1\}$. Therefore,

Ans:

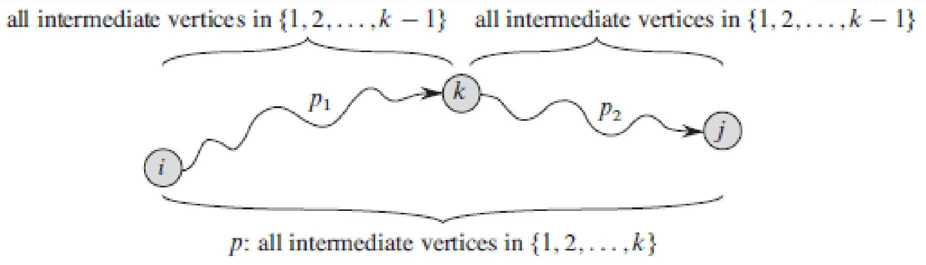

all intermediate vertices in $\{1, 2, \ldots, k-1\}$
all intermediate vertices in $\{1, 2, \ldots, k-1\}$
p_1
k
p_2
j
i
p: all intermediate vertices in $\{1, 2, \ldots, k\}$

Figure 25.3 Path p is a shortest path from vertex i to vertex j, and k is the highest-numbered intermediate vertex of p. Path p_1, the portion of path p from vertex i to vertex k, has all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. The same holds for path p_2 from vertex k to vertex j.

fore, p_1 is a shortest path from i to k with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Similarly, p_2 is a shortest path from vertex k to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$.

A recursive solution to the all-pairs shortest-paths problem

Based on the above observations, we define a recursive formulation of shortest-path estimates that differs from the one in Section 25.1. Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set $\{1, 2, \ldots, k\}$. When $k = 0$, a path from vertex i to vertex j with no intermediate vertex numbered higher than 0 has no intermediate vertices at all. Such a path has at most one edge, and hence $d_{ij}^{(0)} = w_{ij}$. Following the above discussion, we define $d_{ij}^{(k)}$ recursively by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 , \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 . \end{cases} \tag{25.5}$$

Because for any path, all intermediate vertices are in the set $\{1, 2, \ldots, n\}$, the matrix $D^{(n)} = \left(d_{ij}^{(n)}\right)$ gives the final answer: $d_{ij}^{(n)} = \delta(i, j)$ for all $i, j \in V$.

Computing the shortest-path weights bottom up

Based on recurrence (25.5), we can use the following bottom-up procedure to compute the values $d_{ij}^{(k)}$ in order of increasing values of k. Its input is an $n \times n$ matrix W defined as in equation (25.1). The procedure returns the matrix $D^{(n)}$ of shortest-path weights.

```
FLOYD-WARSHALL($W$)
1  $n = W.rows$
2  $D^{(0)} = W$
3  for $k = 1$ to $n$
4      let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
5      for $i = 1$ to $n$
6          for $j = 1$ to $n$
7              $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
8  return $D^{(n)}$
```

Figure 25.4 shows the matrices $D^{(k)}$ computed by the Floyd-Warshall algorithm for the graph in Figure 25.1.

The running time of the Floyd-Warshall algorithm is determined by the triply nested **for** loops of lines 3–7. Because each execution of line 7 takes $O(1)$ time, the algorithm runs in time $\Theta(n^3)$. As in the final algorithm in Section 25.1, the code is tight, with no elaborate data structures, and so the constant hidden in the Θ-notation is small. Thus, the Floyd-Warshall algorithm is quite practical for even moderate-sized input graphs.

Solution for the graph in figure:

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

UNIT 3

Q 5. Write and explain the algorithm for Rod Cutting Problem in detail using proper example.

Ans:

15.1 Rod cutting

Our first example uses dynamic programming to solve a simple problem in deciding where to cut steel rods. Serling Enterprises buys long steel rods and cuts them into shorter rods, which it then sells. Each cut is free. The management of Serling Enterprises wants to know the best way to cut up the rods.

We assume that we know, for $i = 1, 2, \ldots$, the price p_i in dollars that Serling Enterprises charges for a rod of length i inches. Rod lengths are always an integral number of inches. Figure 15.1 gives a sample price table.

The ***rod-cutting problem*** is the following. Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \ldots, n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces. Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

Consider the case when $n = 4$. Figure 15.2 shows all the ways to cut up a rod of 4 inches in length, including the way with no cuts at all. We see that cutting a 4-inch rod into two 2-inch pieces produces revenue $p_2 + p_2 = 5 + 5 = 10$, which is optimal.

We can cut up a rod of length n in 2^{n-1} different ways, since we have an independent option of cutting, or not cutting, at distance i inches from the left end,

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Figure 15.1 A sample price table for rods. Each rod of length i inches earns the company p_i dollars of revenue.

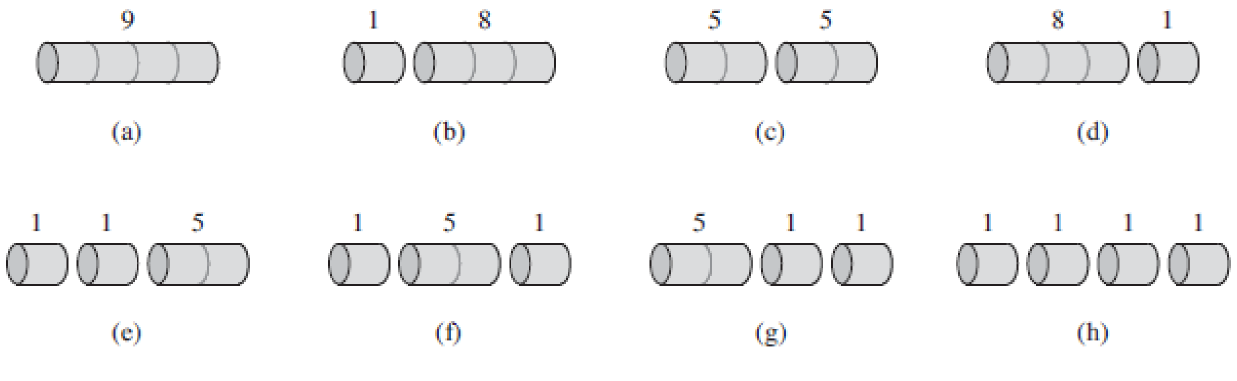


Figure 15.2 The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart of Figure 15.1. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

Recursive top-down implementation

The following procedure implements the computation implicit in equation (15.2) in a straightforward, top-down, recursive manner.

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = −∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n − i))
6  return q
```

Procedure CUT-ROD takes as input an array $p[1\,..\,n]$ of prices and an integer n, and it returns the maximum revenue possible for a rod of length n. If $n = 0$, no revenue is possible, and so CUT-ROD returns 0 in line 2. Line 3 initializes the maximum revenue q to $-\infty$, so that the **for** loop in lines 4–5 correctly computes $q = \max_{1 \le i \le n}(p_i + \text{CUT-ROD}(p, n-i))$; line 6 then returns this value. A simple induction on n proves that this answer is equal to the desired answer r_n, using equation (15.2).

There are usually two equivalent ways to implement a dynamic-programming approach. We shall illustrate both of them with our rod-cutting example.

The first approach is ***top-down with memoization***.[2] In this approach, we write the procedure recursively in a natural manner, but modified to save the result of each subproblem (usually in an array or hash table). The procedure now first checks to see whether it has previously solved this subproblem. If so, it returns the saved value, saving further computation at this level; if not, the procedure computes the value in the usual manner. We say that the recursive procedure has been ***memoized***; it "remembers" what results it has computed previously.

Here is the the pseudocode for the top-down CUT-ROD procedure, with memoization added:

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = −∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if $r[n] \geq 0$
2      return $r[n]$
3  if $n == 0$
4      $q = 0$
5  else $q = -\infty$
6      for $i = 1$ to $n$
7          $q = \max(q, p[i] + \text{MEMOIZED-CUT-ROD-AUX}(p, n - i, r))$
8  $r[n] = q$
9  return $q$
```

Here, the main procedure MEMOIZED-CUT-ROD initializes a new auxiliary array $r[0..n]$ with the value $-\infty$, a convenient choice with which to denote "unknown." (Known revenue values are always nonnegative.) It then calls its helper routine, MEMOIZED-CUT-ROD-AUX.

The procedure MEMOIZED-CUT-ROD-AUX is just the memoized version of our previous procedure, CUT-ROD. It first checks in line 1 to see whether the desired value is already known and, if it is, then line 2 returns it. Otherwise, lines 3–7 compute the desired value q in the usual manner, line 8 saves it in $r[n]$, and line 9 returns it.

The bottom-up version is even simpler:

```
BOTTOM-UP-CUT-ROD(p, n)
1  let $r[0..n]$ be a new array
2  $r[0] = 0$
3  for $j = 1$ to $n$
4      $q = -\infty$
5      for $i = 1$ to $j$
6          $q = \max(q, p[i] + r[j - i])$
7      $r[j] = q$
8  return $r[n]$
```

For the bottom-up dynamic-programming approach, BOTTOM-UP-CUT-ROD uses the natural ordering of the subproblems: a problem of size i is "smaller" than a subproblem of size j if $i < j$. Thus, the procedure solves subproblems of sizes $j = 0, 1, \ldots, n$, in that order.

Line 1 of procedure BOTTOM-UP-CUT-ROD creates a new array $r[0..n]$ in which to save the results of the subproblems, and line 2 initializes $r[0]$ to 0, since a rod of length 0 earns no revenue. Lines 3–6 solve each subproblem of size j, for $j = 1, 2, \ldots, n$, in order of increasing size. The approach used to solve a problem of a particular size j is the same as that used by CUT-ROD, except that line 6 now

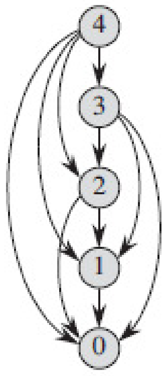


Figure 15.4 The subproblem graph for the rod-cutting problem with $n = 4$. The vertex labels give the sizes of the corresponding subproblems. A directed edge (x, y) indicates that we need a solution to subproblem y when solving subproblem x. This graph is a reduced version of the tree of Figure 15.3, in which all nodes with the same label are collapsed into a single vertex and all edges go from parent to child.

directly references array entry $r[j - i]$ instead of making a recursive call to solve the subproblem of size $j - i$. Line 7 saves in $r[j]$ the solution to the subproblem of size j. Finally, line 8 returns $r[n]$, which equals the optimal value r_n.

The bottom-up and top-down versions have the same asymptotic running time. The running time of procedure BOTTOM-UP-CUT-ROD is $\Theta(n^2)$, due to its doubly-nested loop structure. The number of iterations of its inner **for** loop, in lines 5–6, forms an arithmetic series. The running time of its top-down counterpart, MEMOIZED-CUT-ROD, is also $\Theta(n^2)$, although this running time may be a little harder to see. Because a recursive call to solve a previously solved subproblem returns immediately, MEMOIZED-CUT-ROD solves each subproblem just once. It solves subproblems for sizes $0, 1, \ldots, n$. To solve a subproblem of size n, the **for** loop of lines 6–7 iterates n times. Thus, the total number of iterations of this **for** loop, over all recursive calls of MEMOIZED-CUT-ROD, forms an arithmetic series, giving a total of $\Theta(n^2)$ iterations, just like the inner **for** loop of BOTTOM-UP-CUT-ROD. (We actually are using a form of aggregate analysis here. We shall see aggregate analysis in detail in Section 17.1.)

Subproblem graphs

When we think about a dynamic-programming problem, we should understand the set of subproblems involved and how subproblems depend on one another.

The ***subproblem graph*** for the problem embodies exactly this information. Figure 15.4 shows the subproblem graph for the rod-cutting problem with $n = 4$. It is a directed graph, containing one vertex for each distinct subproblem. The sub-

Reconstructing a solution

Our dynamic-programming solutions to the rod-cutting problem return the value of an optimal solution, but they do not return an actual solution: a list of piece sizes. We can extend the dynamic-programming approach to record not only the optimal *value* computed for each subproblem, but also a *choice* that led to the optimal value. With this information, we can readily print an optimal solution.

Here is an extended version of BOTTOM-UP-CUT-ROD that computes, for each rod size j, not only the maximum revenue r_j, but also s_j, the optimal size of the first piece to cut off:

```
EXTENDED-BOTTOM-UP-CUT-ROD$(p, n)$
1   let $r[0..n]$ and $s[0..n]$ be new arrays
2   $r[0] = 0$
3   for $j = 1$ to $n$
4       $q = -\infty$
5       for $i = 1$ to $j$
6           if $q < p[i] + r[j-i]$
7               $q = p[i] + r[j-i]$
8               $s[j] = i$
9       $r[j] = q$
10  return $r$ and $s$
```

This procedure is similar to BOTTOM-UP-CUT-ROD, except that it creates the array s in line 1, and it updates $s[j]$ in line 8 to hold the optimal size i of the first piece to cut off when solving a subproblem of size j.

The following procedure takes a price table p and a rod size n, and it calls EXTENDED-BOTTOM-UP-CUT-ROD to compute the array $s[1..n]$ of optimal first-piece sizes and then prints out the complete list of piece sizes in an optimal decomposition of a rod of length n:

```
PRINT-CUT-ROD-SOLUTION$(p, n)$
1   $(r, s)$ = EXTENDED-BOTTOM-UP-CUT-ROD$(p, n)$
2   while $n > 0$
3       print $s[n]$
4       $n = n - s[n]$
```

In our rod-cutting example, the call EXTENDED-BOTTOM-UP-CUT-ROD$(p, 10)$ would return the following arrays:

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

A call to PRINT-CUT-ROD-SOLUTION$(p, 10)$ would print just 10, but a call with $n = 7$ would print the cuts 1 and 6, corresponding to the first optimal decomposition for r_7 given earlier.

Q 6. What is Greedy Algorithm? Explain with Proper example. Also write the general characteristics of Greedy Algorithm.

Ans:

5 Greedy Algorithms

The philosophy of being greedy is shortsightedness. Always go for the seemingly best next thing, always optimize the presence, without any regard for the future, and never change your mind about the past. The greedy paradigm is typically applied to optimization problems. In this section, we first consider a scheduling problem and second the construction of optimal codes.

A scheduling problem. Consider a set of activities $1, 2, \ldots, n$. Activity i starts at time s_i and finishes at time $f_i > s_i$. Two activities i and j *overlap* if $[s_i, f_i] \cap [s_j, f_j] \neq \emptyset$. The objective is to select a maximum number of pairwise non-overlapping activities. An example is shown in Figure 6. The largest number of activities can be scheduled by choosing activities with early finish times first. We first sort and reindex such that $i < j$ implies $f_i \leq f_j$.

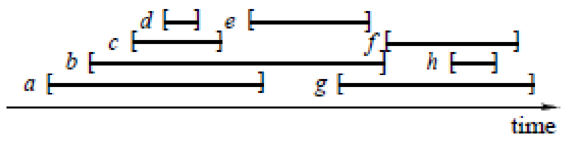


Figure 6: A best schedule is c, e, f, but there are also others of the same size.

```
$S = \{1\}$; $last = 1$;
for $i = 2$ to $n$ do
  if $f_{last} < s_i$ then
    $S = S \cup \{i\}$; $last = i$
  endif
endfor.
```

The running time is $O(n \log n)$ for sorting plus $O(n)$ for the greedy collection of activities.

It is often difficult to determine how close to the optimum the solutions found by a greedy algorithm really are. However, for the above scheduling problem the greedy algorithm always finds an optimum. For the proof let $1 = i_1 < i_2 < \ldots < i_k$ be the greedy schedule constructed by the algorithm. Let $j_1 < j_2 < \ldots < j_\ell$ be any other feasible schedule. Since $i_1 = 1$ has the earliest finish time of any activity, we have $f_{i_1} \leq f_{j_1}$. We can therefore add i_1 to the feasible schedule and remove at most one activity, namely j_1. Among the activities that do not overlap i_1, i_2 has the earliest finish time, hence $f_{i_2} \leq f_{j_2}$. We can again add i_2 to the feasible schedule and remove at most one activity, namely j_2 (or possibly j_1 if it was not removed before). Eventually, we replace the entire feasible schedule by the greedy schedule without decreasing the number of activities. Since we could have started with a maximum feasible schedule, we conclude that the greedy schedule is also maximum.

General Characteristics of Greedy algorithms:

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. This heuristic strategy does not always produce an optimal solution, but as we saw in the activity-selection problem, sometimes it does. This section discusses some of the general properties of greedy methods.

The process that we followed in Section 16.1 to develop a greedy algorithm was a bit more involved than is typical. We went through the following steps:

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution. (For the activity-selection problem, we formulated recurrence (16.2), but we bypassed developing a recursive algorithm based on this recurrence.)
3. Show that if we make the greedy choice, then only one subproblem remains.
4. Prove that it is always safe to make the greedy choice. (Steps 3 and 4 can occur in either order.)
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.

In going through these steps, we saw in great detail the dynamic-programming underpinnings of a greedy algorithm. For example, in the activity-selection problem, we first defined the subproblems S_{ij}, where both i and j varied. We then found that if we always made the greedy choice, we could restrict the subproblems to be of the form S_k.

Alternatively, we could have fashioned our optimal substructure with a greedy choice in mind, so that the choice leaves just one subproblem to solve. In the activity-selection problem, we could have started by dropping the second subscript and defining subproblems of the form S_k. Then, we could have proven that a greedy choice (the first activity a_m to finish in S_k), combined with an optimal solution to the remaining set S_m of compatible activities, yields an optimal solution to S_k. More generally, we design greedy algorithms according to the following sequence of steps:

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.
3. Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

UNIT 4

Q 7. What is Binary Search Tree? How searching is performed in Binary Search Tree? How to find out Minima Maxima in the Binary Search Tree?

Ans:

A binary search tree is organized, as the name suggests, in a binary tree, as shown in Figure 12.1. We can represent such a tree by a linked data structure in which each node is an object. In addition to a *key* and satellite data, each node contains attributes *left*, *right*, and *p* that point to the nodes corresponding to its left child,

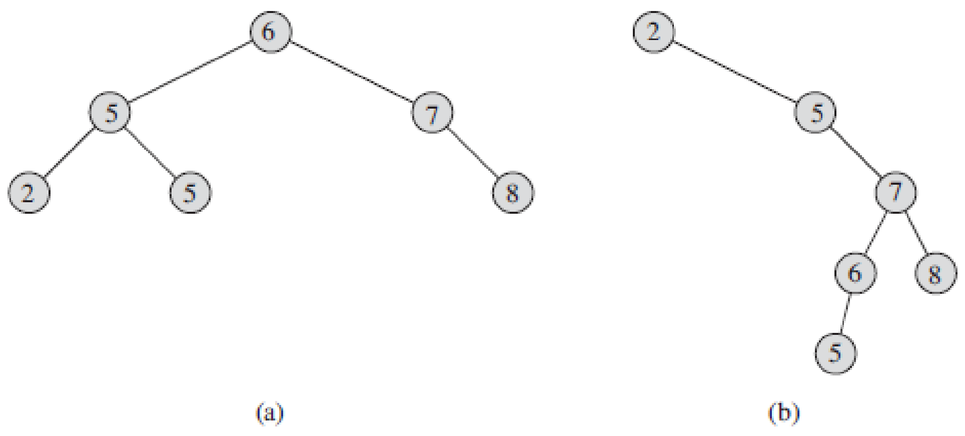


Figure 12.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $x.key$, and the keys in the right subtree of x are at least $x.key$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. **(a)** A binary search tree on 6 nodes with height 2. **(b)** A less efficient binary search tree with height 4 that contains the same keys.

its right child, and its parent, respectively. If a child or the parent is missing, the appropriate attribute contains the value NIL. The root node is the only node in the tree whose parent is NIL.

The keys in a binary search tree are always stored in such a way as to satisfy the ***binary-search-tree property***:

Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \le x.key$. If y is a node in the right subtree of x, then $y.key \ge x.key$.

Searching

We use the following procedure to search for a node with a given key in a binary search tree. Given a pointer to the root of the tree and a key k, TREE-SEARCH returns a pointer to a node with key k if one exists; otherwise, it returns NIL.

```
TREE-SEARCH$(x, k)$
1  if $x$ == NIL or $k$ == $x.key$
2      return $x$
3  if $k < x.key$
4      return TREE-SEARCH$(x.left, k)$
5  else return TREE-SEARCH$(x.right, k)$
```

The procedure begins its search at the root and traces a simple path downward in the tree, as shown in Figure 12.2. For each node x it encounters, it compares the key k with $x.key$. If the two keys are equal, the search terminates. If k is smaller than $x.key$, the search continues in the left subtree of x, since the binary-search-tree property implies that k could not be stored in the right subtree. Symmetrically, if k is larger than $x.key$, the search continues in the right subtree. The nodes encountered during the recursion form a simple path downward from the root of the tree, and thus the running time of TREE-SEARCH is $O(h)$, where h is the height of the tree.

```
ITERATIVE-TREE-SEARCH$(x, k)$
1  while $x \neq$ NIL and $k \neq x.key$
2      if $k < x.key$
3          $x = x.left$
4      else $x = x.right$
5  return $x$
```

Minimum and maximum

We can always find an element in a binary search tree whose key is a minimum by following *left* child pointers from the root until we encounter a NIL, as shown in Figure 12.2. The following procedure returns a pointer to the minimum element in the subtree rooted at a given node x, which we assume to be non-NIL:

```
TREE-MINIMUM$(x)$
1  while $x.left \neq$ NIL
2      $x = x.left$
3  return $x$
```

The binary-search-tree property guarantees that TREE-MINIMUM is correct. If a node x has no left subtree, then since every key in the right subtree of x is at least as large as $x.key$, the minimum key in the subtree rooted at x is $x.key$. If node x has a left subtree, then since no key in the right subtree is smaller than $x.key$ and every key in the left subtree is not larger than $x.key$, the minimum key in the subtree rooted at x resides in the subtree rooted at $x.left$.

The pseudocode for TREE-MAXIMUM is symmetric:

```
TREE-MAXIMUM(x)
1  while x.right ≠ NIL
2      x = x.right
3  return x
```

Both of these procedures run in $O(h)$ time on a tree of height h since, as in TREE-SEARCH, the sequence of nodes encountered forms a simple path downward from the root.

Q 8. Explain minimum spanning tree problem. Explain Prim's algorithm for the solution of minimum spanning tree problem with proper example.

Ans:

We can model this wiring problem with a connected, undirected graph $G = (V, E)$, where V is the set of pins, E is the set of possible interconnections between pairs of pins, and for each edge $(u, v) \in E$, we have a weight $w(u, v)$ specifying the cost (amount of wire needed) to connect u and v. We then wish to find an acyclic subset $T \subseteq E$ that connects all of the vertices and whose total weight

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

is minimized. Since T is acyclic and connects all of the vertices, it must form a tree, which we call a ***spanning tree*** since it "spans" the graph G. We call the problem of determining the tree T the ***minimum-spanning-tree problem***.[1] Figure 23.1 shows

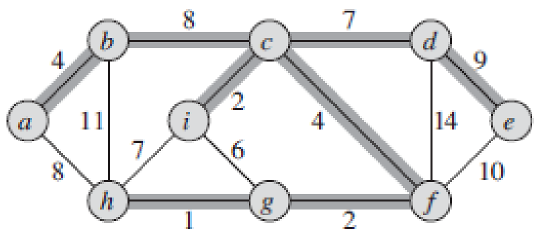


Figure 23.1 A minimum spanning tree for a connected graph. The weights on edges are shown, and the edges in a minimum spanning tree are shaded. The total weight of the tree shown is 37. This minimum spanning tree is not unique: removing the edge (b, c) and replacing it with the edge (a, h) yields another spanning tree with weight 37.

Prim's algorithm

Like Kruskal's algorithm, Prim's algorithm is a special case of the generic minimum-spanning-tree method from Section 23.1. Prim's algorithm operates much like Dijkstra's algorithm for finding shortest paths in a graph, which we shall see in Section 24.3. Prim's algorithm has the property that the edges in the set A always form a single tree. As Figure 23.5 shows, the tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V. Each step adds to the tree A a light edge that connects A to an isolated vertex—one on which no edge of A is incident. By Corollary 23.2, this rule adds only edges that are safe for A; therefore, when the algorithm terminates, the edges in A form a minimum spanning tree. This strategy qualifies as greedy since at each step it adds to the tree an edge that contributes the minimum amount possible to the tree's weight.

In order to implement Prim's algorithm efficiently, we need a fast way to select a new edge to add to the tree formed by the edges in A. 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* attribute. For each vertex v, the attribute $v.key$ is the minimum weight of any edge connecting v to a vertex in the tree; by convention, $v.key = \infty$ if there is no such edge. The attribute $v.\pi$ names the parent of v in the tree. The algorithm implicitly maintains the set A from GENERIC-MST as

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

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

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

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

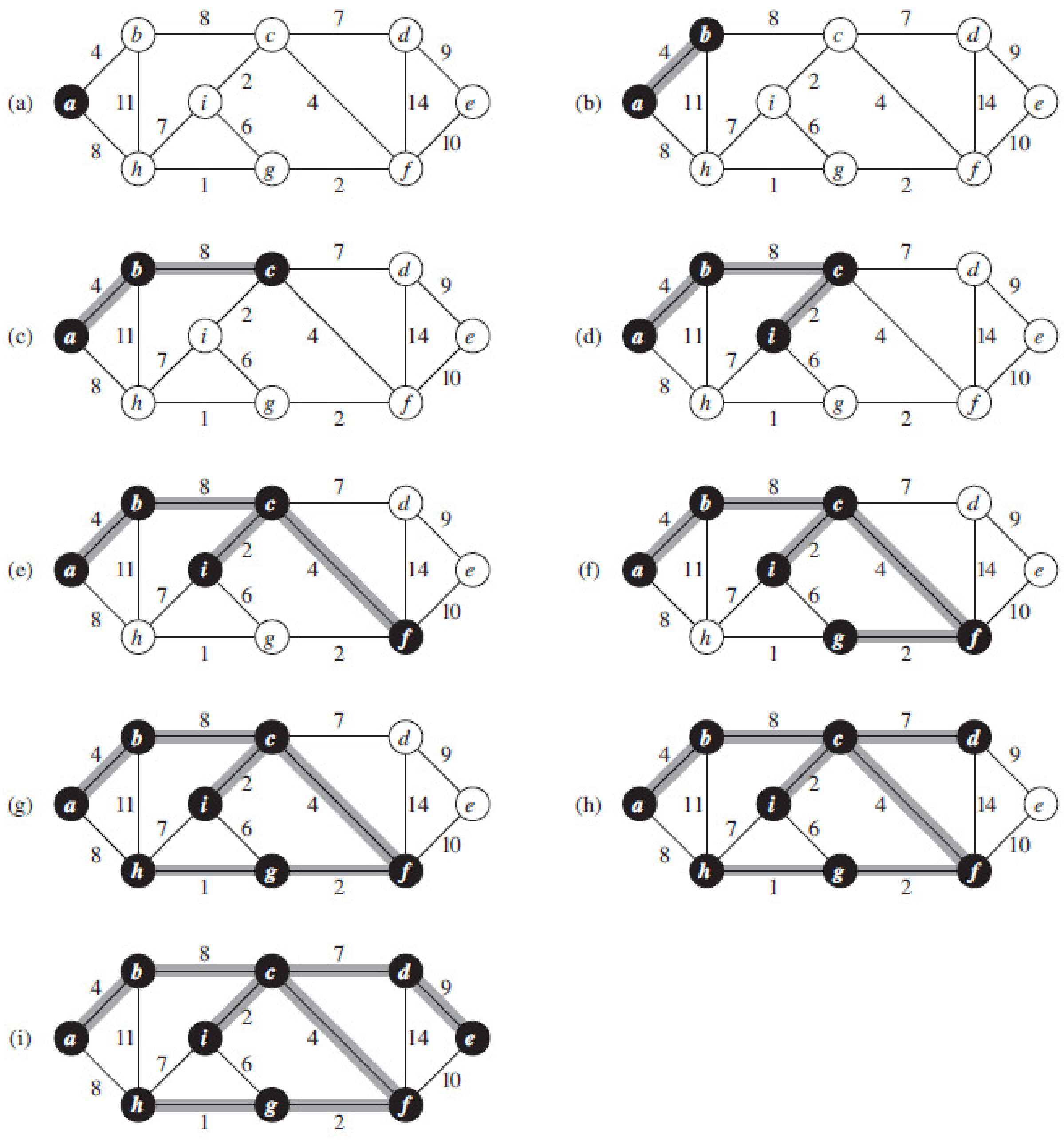


Figure 23.5 The execution of Prim's algorithm on the graph from Figure 23.1. The root vertex is a. Shaded edges are in the tree being grown, and black vertices are in the tree. At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing the cut is added to the tree. In the second step, for example, the algorithm has a choice of adding either edge (b, c) or edge (a, h) to the tree since both are light edges crossing the cut.

Figure 23.5 shows how Prim's algorithm works. Lines 1–5 set the key of each vertex to ∞ (except for the root r, whose key is set to 0 so that it will be the first vertex processed), set the parent of each vertex to NIL, and initialize the min-priority queue Q to contain all the vertices. The algorithm maintains the following three-part loop invariant:

Prior to each iteration of the **while** loop of lines 6–11,

1. $A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$.
2. The vertices already placed into the minimum spanning tree are those in $V - Q$.
3. For all vertices $v \in Q$, if $v.\pi \neq$ NIL, then $v.key < \infty$ and $v.key$ is the weight of a light edge $(v, v.\pi)$ connecting v to some vertex already placed into the minimum spanning tree.

Line 7 identifies a vertex $u \in Q$ incident on a light edge that crosses the cut $(V - Q, Q)$ (with the exception of the first iteration, in which $u = r$ due to line 4). Removing u from the set Q adds it to the set $V - Q$ of vertices in the tree, thus adding $(u, u.\pi)$ to A. The **for** loop of lines 8–11 updates the *key* and π attributes of every vertex v adjacent to u but not in the tree, thereby maintaining the third part of the loop invariant.

The running time of Prim's algorithm depends on how we implement the min-priority queue Q. If we implement Q as a binary min-heap (see Chapter 6), we can use the BUILD-MIN-HEAP procedure to perform lines 1–5 in $O(V)$ time. The body of the **while** loop executes $|V|$ 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 executes $O(E)$ times altogether, since the sum of the lengths of all adjacency lists is $2|E|$. Within the **for** loop, we can implement the test for membership in Q in line 9 in constant time by keeping a bit for each vertex that tells whether or not it is in Q, and updating the bit when the vertex is removed from Q. The assignment in line 11 involves an implicit DECREASE-KEY operation on the min-heap, which a binary min-heap supports 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.

We can improve the asymptotic running time of Prim's algorithm by using Fibonacci heaps. Chapter 19 shows that if a Fibonacci heap holds $|V|$ elements, an EXTRACT-MIN operation takes $O(\lg V)$ amortized time and a DECREASE-KEY operation (to implement line 11) takes $O(1)$ amortized time. Therefore, if we use a Fibonacci heap to implement the min-priority queue Q, the running time of Prim's algorithm improves to $O(E + V \lg V)$.

UNIT 5

Q 9. Explain Polynomial time verification using Hamiltonian cycle.

ANS:

Polynomial-time verification

We now look at algorithms that verify membership in languages. For example, suppose that for a given instance $\langle G, u, v, k \rangle$ of the decision problem PATH, we are also given a path p from u to v. We can easily check whether p is a path in G and whether the length of p is at most k, and if so, we can view p as a "certificate" that the instance indeed belongs to PATH. For the decision problem PATH, this certificate doesn't seem to buy us much. After all, PATH belongs to P—in fact, we can solve PATH in linear time—and so verifying membership from a given certificate takes as long as solving the problem from scratch. We shall now examine a problem for which we know of no polynomial-time decision algorithm and yet, given a certificate, verification is easy.

Hamiltonian cycles

The problem of finding a hamiltonian cycle in an undirected graph has been studied for over a hundred years. Formally, a ***hamiltonian cycle*** of an undirected graph $G = (V, E)$ is a simple cycle that contains each vertex in V. A graph that contains a hamiltonian cycle is said to be ***hamiltonian***; otherwise, it is ***nonhamiltonian***. The name honors W. R. Hamilton, who described a mathematical game on the dodecahedron (Figure 34.2(a)) in which one player sticks five pins in any five consecutive vertices and the other player must complete the path to form a cycle

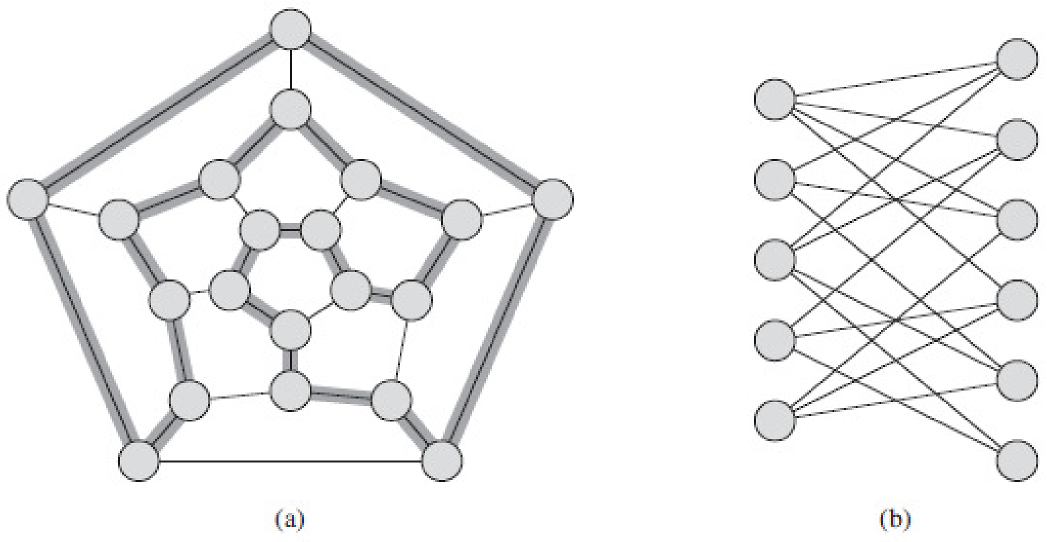


Figure 34.2 **(a)** A graph representing the vertices, edges, and faces of a dodecahedron, with a hamiltonian cycle shown by shaded edges. **(b)** A bipartite graph with an odd number of vertices. Any such graph is nonhamiltonian.

containing all the vertices.[7] The dodecahedron is hamiltonian, and Figure 34.2(a) shows one hamiltonian cycle. Not all graphs are hamiltonian, however. For example, Figure 34.2(b) shows a bipartite graph with an odd number of vertices. Exercise 34.2-2 asks you to show that all such graphs are nonhamiltonian.

We can define the ***hamiltonian-cycle problem***, "Does a graph G have a hamiltonian cycle?" as a formal language:

$$\text{HAM-CYCLE} = \{\langle G \rangle : G \text{ is a hamiltonian graph}\} .$$

How might an algorithm decide the language HAM-CYCLE? Given a problem instance $\langle G \rangle$, one possible decision algorithm lists all permutations of the vertices of G and then checks each permutation to see if it is a hamiltonian path. What is the running time of this algorithm? If we use the "reasonable" encoding of a graph as its adjacency matrix, the number m of vertices in the graph is $\Omega(\sqrt{n})$, where $n = |\langle G \rangle|$ is the length of the encoding of G. There are $m!$ possible permutations

of the vertices, and therefore the running time is $\Omega(m!) = \Omega(\sqrt{n}\,!) = \Omega(2^{\sqrt{n}})$, which is not $O(n^k)$ for any constant k. Thus, this naive algorithm does not run in polynomial time. In fact, the hamiltonian-cycle problem is NP-complete, as we shall prove in Section 34.5.

Verification algorithms

Consider a slightly easier problem. Suppose that a friend tells you that a given graph G is hamiltonian, and then offers to prove it by giving you the vertices in order along the hamiltonian cycle. It would certainly be easy enough to verify the proof: simply verify that the provided cycle is hamiltonian by checking whether it is a permutation of the vertices of V and whether each of the consecutive edges along the cycle actually exists in the graph. You could certainly implement this verification algorithm to run in $O(n^2)$ time, where n is the length of the encoding of G. Thus, a proof that a hamiltonian cycle exists in a graph can be verified in polynomial time.

We define a ***verification algorithm*** as being a two-argument algorithm A, where one argument is an ordinary input string x and the other is a binary string y called a ***certificate***. A two-argument algorithm A ***verifies*** an input string x if there exists a certificate y such that $A(x, y) = 1$. The ***language verified*** by a verification algorithm A is

$$L = \{x \in \{0,1\}^* : \text{ there exists } y \in \{0,1\}^* \text{ such that } A(x, y) = 1\} .$$

Intuitively, an algorithm A verifies a language L if for any string $x \in L$, there exists a certificate y that A can use to prove that $x \in L$. Moreover, for any string $x \notin L$, there must be no certificate proving that $x \in L$. For example, in the hamiltonian-cycle problem, the certificate is the list of vertices in some hamiltonian cycle. If a graph is hamiltonian, the hamiltonian cycle itself offers enough information to verify this fact. Conversely, if a graph is not hamiltonian, there can be no list of vertices that fools the verification algorithm into believing that the graph is hamiltonian, since the verification algorithm carefully checks the proposed "cycle" to be sure.

Q 10. Write and explain the algorithm for Edit Distance Problem in detail using proper example.
ANS:

Edit distance. We illustrate dynamic programming using the edit distance problem, which is motivated by questions in genetics. We assume a finite set of *characters* or *letters*, Σ, which we refer to as the *alphabet*, and we consider *strings* or *words* formed by concatenating finitely many characters from the alphabet. The *edit distance* between two words is the minimum number of letter insertions, letter deletions, and letter substitutions required to transform one word to the other. For example, the edit distance between FOOD and MONEY is at most four:

FOOD $\rightarrow$ MOOD $\rightarrow$ MOND $\rightarrow$ MONED $\rightarrow$ MONEY

A better way to display the editing process is the *gap representation* that places the words one above the other, with a gap in the first word for every insertion and a gap in the second word for every deletion:

```
F O O   D
M O N E Y
```

Columns with two different characters correspond to substitutions. The number of editing steps is therefore the number of columns that do not contain the same character twice.

Prefix property. It is not difficult to see that you cannot get from FOOD to MONEY in less than four steps. However, for longer examples it seems considerably more difficult to find the minimum number of steps or to recognize an optimal edit sequence. Consider for example

```
A L G O R   I   T H M
A L   T R U I S T I C
```

Is this optimal or, equivalently, is the edit distance between ALGORITHM and ALTRUISTIC six? Instead of answering this specific question, we develop a dynamic programming algorithm that computes the edit distance between an m-character string $A[1..m]$ and an n-character string $B[1..n]$. Let $E(i, j)$ be the edit distance between the prefixes of length i and j, that is, between $A[1..i]$ and $B[1..j]$. The edit distance between the complete strings is therefore $E(m, n)$. A crucial step towards the development of this algorithm is the following observation about the gap representation of an optimal edit sequence.

PREFIX PROPERTY. If we remove the last column of an optimal edit sequence then the remaining columns represent an optimal edit sequence for the remaining substrings.

We can easily prove this claim by contradiction: if the substrings had a shorter edit sequence, we could just glue the last column back on and get a shorter edit sequence for the original strings.

Recursive formulation. We use the Prefix Property to develop a recurrence relation for E. The dynamic programming algorithm will be a straightforward implementation of that relation. There are a couple of obvious base cases:

- **Erasing:** we need i deletions to erase an i-character string, $E(i, 0) = i$.
- **Creating:** we need j insertions to create a j-character string, $E(0, j) = j$.

In general, there are four possibilities for the last column in an optimal edit sequence.

- **Insertion:** the last entry in the top row is empty, $E(i, j) = E(i, j - 1) + 1$.
- **Deletion:** the last entry in the bottom row is empty, $E(i, j) = E(i - 1, j) + 1$.
- **Substitution:** both rows have characters in the last column that are different, $E(i, j) = E(i - 1, j - 1) + 1$.
- **No action:** both rows end in the same character, $E(i, j) = E(i - 1, j - 1)$.

Let P be the logical proposition $A[i] \neq B[j]$ and denote by $|P|$ its indicator variable: $|P| = 1$ if P is true and $|P| = 0$ if P is false. We can now summarize and for $i, j > 0$ get the edit distance as the smallest of the possibilities:

$$E(i, j) = \min \left\{ \begin{array}{l} E(i, j-1) + 1 \\ E(i-1, j) + 1 \\ E(i-1, j-1) + |P| \end{array} \right\}.$$

The algorithm. If we turned this recurrence relation directly into a divide-and-conquer algorithm, we would have the following recurrence for the running time:

$$\begin{aligned} T(m,n) &= T(m,n-1)+T(m-1,n) \\ &\quad + T(m-1,n-1)+1. \end{aligned}$$

The solution to this recurrence is exponential in m and n, which is clearly not the way to go. Instead, let us build an $m+1$ times $n+1$ table of possible values of $E(i,j)$. We can start by filling in the base cases, the entries in the 0-th row and column. To fill in any other entry, we need to know the values directly to the left, directly above, and both to the left and above. If we fill the table from top to bottom and from left to right then whenever we reach an entry, the entries it depends on are already available.

```
int EDITDISTANCE(int $m, n$)
  for $i = 0$ to $m$ do $E[i, 0] = i$ endfor;
  for $j = 1$ to $n$ do $E[0, j] = j$ endfor;
  for $i = 1$ to $m$ do
    for $j = 1$ to $n$ do
      $E[i, j] = \min\{E[i, j-1]+1, E[i-1, j]+1,$
                $E[i-1, j-1]+|A[i] \neq B[j]|\}$
    endfor
  endfor;
  return $E[m, n]$.
```

Since there are $(m+1)(n+1)$ entries in the table and each takes a constant time to compute, the total running time is in $O(mn)$.

An example. The table constructed for the conversion of ALGORITHM to ALTRUISTIC is shown in Figure 5. Boxed numbers indicate places where the two strings have equal characters. The arrows indicate the predecessors that define the entries. Each direction of arrow corresponds to a different edit operation: horizontal for insertion, vertical for deletion, and diagonal for substitution. Dotted diagonal arrows indicate free substitutions of a letter for itself.

Recovering the edit sequence. By construction, there is at least one path from the upper left to the lower right corner, but often there will be several. Each such path describes an optimal edit sequence. For the example at hand, we have three optimal edit sequences:

```
A L G O R I   T H M
A L T R U I S T I C
```

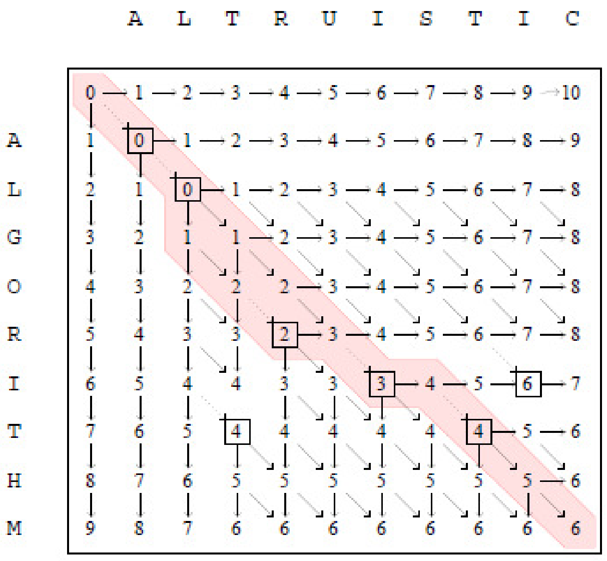


Figure 5: The table of edit distances between all prefixes of ALGORITHM and of ALTRUISTIC. The shaded area highlights the optimal edit sequences, which are paths from the upper left to the lower right corner.

```
A L G O R   I   T H M
A L T   R U I S T I C

A L G O R   I   T H M
A L   T R U I S T I C
```

They are easily recovered by tracing the paths backward, from the end to the beginning. The following algorithm recovers an optimal solution that also minimizes the number of insertions and deletions. We call it with the lengths of the strings as arguments, $R(m,n)$.

```
void R(int $i, j$)
  if $i > 0$ or $j > 0$ then
    switch incoming arrow:
      case $\searrow$: $R(i-1, j-1)$; print($A[i], B[j]$)
      case $\downarrow$: $R(i-1, j)$; print($A[i]$, _)
      case $\rightarrow$: $R(i, j-1)$; print(_, $B[j]$).
    endswitch
  endif.
```

Summary. The structure of dynamic programming is again similar to divide-and-conquer, except that the subproblems to be solved overlap. As a consequence, we get different recursive paths to the same subproblems. To develop a dynamic programming algorithm that avoids redundant solutions, we generally proceed in two steps: