Dynamic Programming Notes

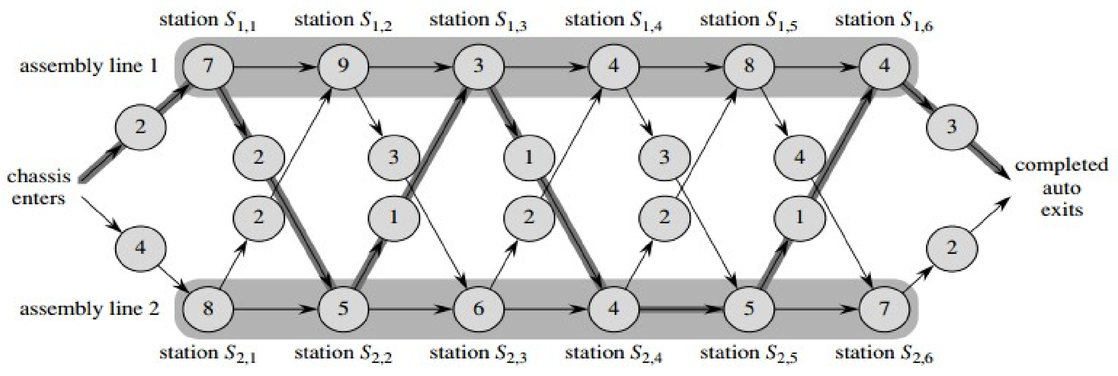
- Introduction to Algorithms

1. Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. (“Programming” in this context refers to a tabular method, not to writing computer code.)
2. Dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share subsubproblems.
3. A dynamic-programming algorithm solves every subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answe every time the subsubproblem is encountered.
4. Dynamic programming is typically applied to ***optimization problems***. In such problems there can be many possible solutions.
5. The development of a dynamic-programming algorithm can be broken into a sequence of four steps.
    1. Characterize the structure of an optimal solution.
    2. Recursively define the value of an optimal solution.
    3. Compute the value of an optimal solution in a bottom-up fashion.
    4. Construct an optimal solution from computed information.

15.1 Assembly-line scheduling

1. A manufacturing problem to find the fastest way through a factory. There are two assembly lines, each with n stations; the jth station on line i is denoted $S_{i,j}$ and the assembly time at that station is $a_{i,j}$.
2. An automobile chassis enters the factory, and goes onto line i (where i = 1 or 2), taking e_i time.
3. After going through the jth station on a line, the chassis goes on to the $(j+1)$st station on either line. There is no transfer cost if it stays on the same line, but it takes time $t_{i,j}$ to transfer to the other line after station $S_{i,j}$.
4. After exiting the nth station on a line, it takes x_i time for the completed auto to exit the factory.

5. The problem is to determine which stations to choose from line 1 and which to choose from line 2 in order to minimize the total time through the factory for one auto.
6. In the example of Figure 15.2(a), the fastest total time comes from choosing stations 1, 3, and 6 from line 1 and stations 2, 4, and 5 from line 2.



(a)

j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

$f^* = 38$

j	2	3	4	5	6
$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

$l^* = 1$

(b)

Figure 15.2 **(a)** An instance of the assembly-line problem with costs e_i, $a_{i,j}$, $t_{i,j}$, and x_i indicated. The heavily shaded path indicates the fastest way through the factory. **(b)** The values of $f_i[j]$, f^*, $l_i[j]$, and l^* for the instance in part (a).

7. Let us consider the fastest possible way for a chassis to get from the starting point through station $S_{1,j}$. If j = 1, there is only one way that the chassis could have gone, and so it is easy to determine how long it takes to get through station $S_{1,j}$.
8. For $j = 2, 3, \dots, n$, however, there are two choices: the chassis could have come from station $S_{1,j-1}$ and then directly to station $S_{1,j}$, the time for going from station $j - 1$ to station j on the same line being negligible.
9. Alternatively, the chassis could have come from station $S_{2,j-1}$ and then been transferred to station $S_{1,j}$, the transfer time being $t_{2,j-1}$.
10. First, let us suppose that the fastest way through station $S_{1,j}$ is through station $S_{1,j-1}$.
11. Similarly, let us now suppose that the fastest way through station $S_{1,j}$ is through station $S_{2,j-1}$.
12. Now we observe that the chassis must have taken a fastest way from the starting point through station $S_{2,j-1}$.

13. We can say that for assembly-line scheduling, an optimal solution to a problem (finding the fastest way through station $S_{i,j}$) contains within it an optimal solution to subproblems (finding the fastest way through either $S_{1,j-1}$ or $S_{2,j-1}$).
14. We refer to this property as ***optimal substructure***, and it is one of the hallmarks of the applicability of dynamic programming.
15. Let $f_i[j]$ denote the fastest possible time to get a chassis from the starting point through station $S_{i,j}$.
16. Our ultimate goal is to determine the fastest time to get a chassis all the way through the factory, which we denote by f^{*}.

    $f^{*} = \min(f_1[n] + x_1, f_2[n] + x_2)$
17. To get through station 1 on eitherline, a chassis just goes directly to that station. Thus,

    $f_1[1] = e_1 + a_{1,1}$,

    $f_2[1] = e_2 + a_{2,1}$.
18. $f_1[j] = \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j})$
19. for j = 2, 3, . . . , n. Symmetrically, we have

    $f_2[j] = \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j})$
20. We obtain the recursive equations

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j = 1 , \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{if } j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j = 1 , \\ \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{if } j \geq 2 \end{cases}$$

21. Let us define $l_i[j]$ to be the line number, 1 or 2, whose station $j-1$ is used in a fastest way through station $S_{i,j}$.
22. We also define l^* to be the line whose station n is used in a fastest way through the entire factory.
23. The $l_i[j]$ values help us trace a fastest way.
24. FASTEST-WAY(*a, t, e, x, n*)

```
1.  $f_1[1] \leftarrow e_1 + a_{1,1}$
2.  $f_2[1] \leftarrow e_2 + a_{2,1}$
3.  for $j \leftarrow 2$ to $n$
4.          do if $f_1[j-1] + a_{1,j} \leq f_2[j-1] + t_{2,j-1} + a_{1,j}$
```

```
5.              then $f_1[j] \leftarrow f_1[j-1] + a_{1,j}$
6.                   $l_1[j] \leftarrow 1$
7.              else $f_1[j] \leftarrow f_2[j-1] + t_{2,j-1} + a_{1,j}$
8.                   $l_1[j] \leftarrow 2$
9.          if $f_2[j-1] + a_{2,j} \le f_1[j-1] + t_{1,j-1} + a_{2,j}$
10.            then $f_2[j] \leftarrow f_2[j-1] + a_{2,j}$
11.                 $l_2[j] \leftarrow 2$
12.            else $f_2[j] \leftarrow f_1[j-1] + t_{1,j-1} + a_{2,j}$
13.                 $l_2[j] \leftarrow 1$
14. if $f_1[n] + x_1 \le f_2[n] + x_2$
15.     then $f^* = f_1[n] + x_1$
16.          $l^* = 1$
17.     else $f^* = f_2[n] + x_2$
18.          $l^* = 2$
```

25. PRINT-STATIONS(l, n)

```
1. $i \leftarrow l^*$
2. print “line ” $i$ “, station ” $n$
3. for $j \leftarrow n$ downto 2
4. do $i \leftarrow l_i[j]$
5.  print “line ” $i$ “, station ” $j-1$
```

15.2 Matrix-chain multiplication

1. Matrix multiplication is associative, and so all parenthesizations yield the same product.
2. The way we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product.
3. MATRIX-MULTIPLY(A, B)

```
1. if columns[A] $\neq$ rows[B]
2.        then error “incompatible dimensions”
3.        else for $i \leftarrow 1$ to rows[A]
4.                 do for $j \leftarrow 1$ to columns[B]
5.                         do C[$i, j$] $\leftarrow 0$
6.                            for $k \leftarrow 1$ to columns[A]
7.                                do C[$i, j$] $\leftarrow$ C[$i, j$] + A[$i, k$] $\cdot$ B[$k, j$]
8.           return C
```

4. The time to compute C is dominated by the number of scalar multiplications in line 7.
5. The **matrix-chain multiplication problem** can be stated as follows: given a chain $(A_1, A_2, . . . , A_n)$ of n matrices, where for $i = 1, 2, . . . , n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.
6. Denote the number of alternative parenthesizations of a sequence of *n* matrices by *P(n)*.
7. When $n = 1$, there is just one matrix and therefore only one way to fully parenthesize the matrix product.
8. When $n \geq 2$, a fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts, and the split between the two subproducts may occur between the *k*th and (*k* + 1)st matrices for any $k = 1, 2, . . . , n - 1$.

$$P(n) = \begin{cases} 1 & \text{if } n = 1 , \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 . \end{cases}$$

9. Let us adopt the notation $Ai..j$, where $i \leq j$, for the matrix that results from evaluating the product $A_i A_{i+1} \cdots A_j$.
10. The optimal substructure of this problem is as follows. Suppose that an optimal parenthesization of $A_i A_{i+1} \cdots A_j$ splits the product between A_k and A_{k+1}.
11. Then the parenthesization of the “prefix” subchain $A_i A_{i+1} \cdots A_k$ within this optimal parenthesization of $A_i A_{i+1} \cdots A_j$ must be an optimal parenthesization of $A_i A_{i+1} \cdots A_k$.
12. Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$; for the full problem, the cost of a cheapest way to compute $A_{1..n}$ would thus be $m[1, n]$.
13. Let us assume that the optimal parenthesization splits the product $A_i A_{i+1} \cdots A_j$ between A_k and A_{k+1}, where $i \leq k < j$. Then, $m[i, j]$ is equal to the minimum cost for computing the subproducts $A_{i..k}$ and $A_{k+1..j}$, plus the cost of multiplying these two matrices together.
14. Recalling that each matrix A_i is $p_{i-1} \times p_i$, we see that computing the matrix product $A_{i..k} A_{k+1..j}$ takes $pi_{-1} p_k p_j$ scalar multiplications.

   $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$
15. This recursive equation assumes that we know the value of *k*, which we do not. There are only $j - i$ possible values for *k*, however, namely $k = i, i + 1, . . . , j - 1$.

16. To help us keep track of how to construct an optimal solution, let us define $s[i, j]$ to be a value of k at which we can split the product $A_i A_{i+1} \cdots A_j$ to obtain an optimal parenthesization.

17. That is, $s[i, j]$ equals a value k such that $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$

18. The following pseudocode assumes that matrix A_i has dimensions $p_{i-1} \times p_i$ for $i = 1, 2, \ldots, n$.

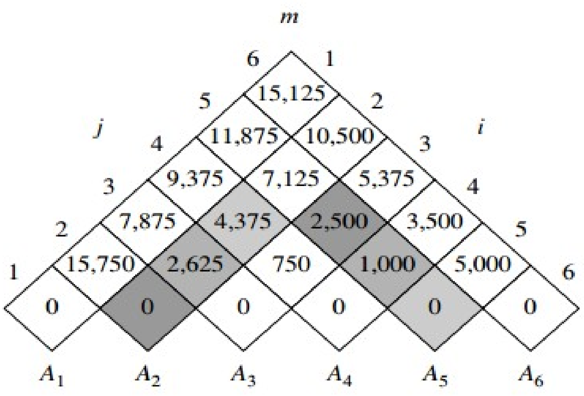
19. The input is a sequence $p = (p0, p1, \ldots, pn)$, where $length[p] = n + 1$.

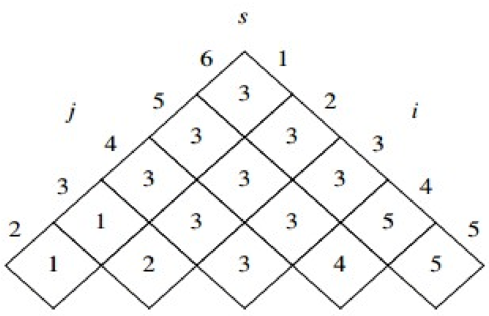
20. The procedure uses an auxiliary table $m[1 \ldots n, 1 \ldots n]$ for storing the $m[i, j]$ costs and an auxiliary table $s[1 \ldots n, 1 \ldots n]$ that records which index of k achieved the optimal cost in computing $m[i, j]$.

21. The algorithm should fill in the table m in a manner that corresponds to solving the parenthesization problem on matrix chains of increasing length.

22. MATRIX-CHAIN-ORDER(p)

```
1.  n ← length[p] − 1
2.  for i ← 1 to n
3.          do m[i, i] ← 0
4.  for l ← 2 to n  //l is the chain length.
5.          do for i ← 1 to n − l + 1
6.                  do j ← i + l − 1
7.                     m[i, j] ← ∞
8.                     for k ← i to j − 1
9.                         do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10.                           if q < m[i, j]
11.                              then m[i, j] ← q
12.                                   s[i, j] ← k
13. return m and s
```





23. Figure 15.3 illustrates this procedure on a chain of *n* = 6 matrices. Since we have defined *m[i, j]* only for $i \leq j$, only the portion of the table m strictly above the main diagonal is used.
24. The figure shows the table rotated to make the main diagonal run horizontally.

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 & = 13000 , \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 & = 7125 , \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 & = 11375 \end{cases}$$
$$= 7125 .$$

25. Each entry *s[i, j]* records the value of *k* such that the optimal parenthesization of $A_i A_{i+1} \cdots A_j$ splits the product between A_k and A_{k+1}.
26. PRINT-OPTIMAL-PARENS(s, i, j)

```
1.  if $i = j$
2.        then print “$A$”$_i$
3.        else print “(”
4.              PRINT-OPTIMAL-PARENS($s, i, s[i, j]$)
5.              PRINT-OPTIMAL-PARENS($s, s[i, j] + 1, j$)
6.              print “)
```

15.3 Elements of dynamic programming

1. Recall that a problem exhibits ***optimal substructure*** if an optimal solution to the problem contains within it optimal solutions to subproblems.
2. Whenever a problem exhibits optimal substructure, it is a good clue that dynamic programming might apply.
3. To characterize the space of subproblems, a good rule of thumb is to try to keep the space as simple as possible, and then to expand it as necessary.
4. Dynamic programming uses optimal substructure in a bottom-up fashion. That is, we first find optimal solutions to subproblems and, having solved the subproblems, we find an optimal solution to the problem.

5. There is a variation of dynamic programming that often offers the efficiency of the usual dynamic-programming approach while maintaining a top-down strategy. The idea is to memoize the natural, but inefficient, recursive algorithm.

6. A memoized recursive algorithm maintains an entry in a table for the solution to each subproblem.

7. Each table entry initially contains a special value to indicate that the entry has yet to be filled in. When the subproblem is first encountered during the execution of the recursive algorithm, its solution is computed and then stored in the table.

8. Each subsequent time that the subproblem is encountered, the value stored in the table is simply looked up and returned.

9. MEMOIZED-MATRIX-CHAIN(p)

```
1. $n \leftarrow length[p] - 1$
2. for $i \leftarrow 1$ to $n$
3.         do for $j \leftarrow$ i to $n$
4.                 do $m[i, j] \leftarrow \infty$
5. return LOOKUP-CHAIN($p$, 1, $n$)
```

10. LOOKUP-CHAIN(p, i, j)

```
1. if $m[i, j] < \infty$
2.     then return $m[i, j]$
3. if $i = j$
4.     then $m[i, j] \leftarrow 0$
5.     else for $k \leftarrow i$ to $j - 1$
6.             do $q \leftarrow$ LOOKUP-CHAIN($p, i, k$)
                   + LOOKUP-CHAIN($p, k + 1$, j) + $p_{i-1} p_k p_j$
7.                 if $q < m[i, j]$
8.                     then $m[i, j] \leftarrow q$
9. return $m[i, j]$
```

11. In general practice, if all subproblems must be solved at least once, a bottomu dynamic-programming algorithm usually outperforms a top-down memoized algorithm by a constant factor, because there is no overhead for recursion and less overhead for maintaining the table.

12. Moreover, there are some problems for which the regular pattern of table accesses in the dynamic-programming algorithm can be exploited to reduce time or space requirements even further.

15.4 Longest common subsequence

1. Formally, given a sequence $X = (x_1, x_2, \dots, x_m)$, another sequence $Z = (z_1, z_2, \dots, z_k)$ is a ***subsequence*** of X if there exists a strictly increasing sequence $(i_1, i_2, \dots, i_k)$ of indices of X such that for all $j = 1, 2, \dots, k$, we have $x_{i_j} = z_j$.
2. For example, $Z = (B, C, D, B)$ is a subsequence of $X = (A, B, C, B, D, A, B)$ with corresponding index sequence $(2, 3, 5, 7)$.
3. Given two sequences X and Y, we say that a sequence Z is a ***common subsequence*** of X and Y if Z is a subsequence of both X and Y.
4. In the ***longest-common-subsequence problem***, we are given two sequences $X = (x_1, x_2, \dots, x_m)$ and $Y = (y_1, y_2, \dots, y_n)$ and wish to find a maximum-length common subsequence of X and Y.
5. Given a sequence $X = (x_1, x_2, \dots, x_m)$, we define the *i*th prefix of X, for $i = 0, 1, \dots, m$, as $X_i = (x_1, x_2, \dots, x_i)$.
6. For example, if $X = (A, B, C, B, D, A, B)$, then $X_4 = (A, B, C, B)$ and X_0 is the empty sequence.
7. Let $X = (x_1, x_2, \dots, x_m)$ and $Y = (y_1, y_2, \dots, y_n)$ be sequences, and let $Z = (z_1, z_2, \dots, z_k)$ be any LCS of X and Y.
    1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
    2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
    3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.
8. The optimal substructure of the LCS problem gives the recursive formula

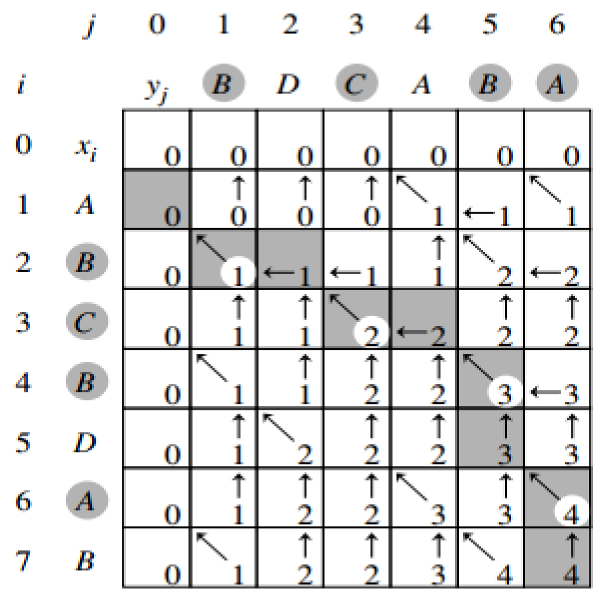
$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j\,. \end{cases}$$

9. Procedure LCS-LENGTH takes two sequences $X = (x_1, x_2, \dots, x_m)$ and $Y = (y_1, y_2, \dots, y_n)$ as inputs.
10. It stores the $c[i, j]$ values in a table $c[0 \,..\, m, 0 \,..\, n]$ whose entries are computed in row-major order.
11. It also maintains the table $b[1 \,..\, m, 1 \,..\, n]$ to simplify construction of an optimal solution. Intuitively, $b[i, j]$ points to the table entry corresponding to the optimal subproblem solution chosen when computing $c[i, j]$.

12. LCS-LENGTH(*X*, *Y*)

```
1.  $m \leftarrow length[X]$
2.  $n \leftarrow length[Y]$
3.  for $i \leftarrow 1$ to $m$
4.      do $c[i, 0] \leftarrow 0$
5.  for $j \leftarrow 0$ to $n$
6.      do $c[0, j] \leftarrow 0$
7.  for $i \leftarrow 1$ to $m$
8.      do for $j \leftarrow 1$ to $n$
9.          do if $x_i = y_j$
10.             then $c[i, j] \leftarrow c[i - 1, j - 1] + 1$
11.                 $b[i, j] \leftarrow$ “”
12.             else if $c[i - 1, j] \geq c[i, j - 1]$
13.                 then $c[i, j] \leftarrow c[i - 1, j]$
14.                     $b[i, j] \leftarrow$ “↑”
15.                 else $c[i, j] \leftarrow c[i, j - 1]$
16.                     $b[i, j] \leftarrow$ “←”
17. return $c$ and $b$
```

13. Here “&” equals titled arrow in figure below.



	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

14. PRINT-LCS(*b, X, i, j*)

```
1. if $i = 0$ or $j = 0$
2.     then return
3. if $b[i, j]$ = “&”
4.     then PRINT-LCS($b, X, i - 1, j - 1$)
5.          print $x_i$
6. elseif $b[i, j]$ = “↑”
7.     then PRINT-LCS($b, X, i - 1, j$)
8. else PRINT-LCS($b, X, i, j - 1$)
```

15.5 Optimal binary search trees

1. The number of nodes visited when searching for a key in a binary search tree is one plus the depth of the node containing the key.
2. We are given a sequence $K = (k_1, k_2, \ldots, k_n)$ of n distinct keys in sorted order (so that $k_1 < k_2 < \cdots < k_n$), and we wish to build a binary search tree from these keys.
3. For each key k_i, we have a probability p_i that a search will be for k_i.
4. Some searches may be for values not in K , and so we also have n + 1 “dummy keys $d_0, d_1, d_2, \ldots, d_n$ representing values not in K.
5. In particular, d_0 represents all values less than k_1, d_n represents all values greater than k_n and for $i = 1, 2, \ldots, n-1$, the dummy key d_i represents all values between k_i and k_{i+1}.
6. For each dummy key d_i, we have a probability q_i that a search will correspond to d_i.
7. Figure 15.7 shows two binary search trees for a set of n = 5 keys. Each key k_i is an internal node, and each dummy key d_i is a leaf. Every search is either successful (finding some key k_i) or unsuccessful (finding some dummy key d_i) and so we have

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1 .$$

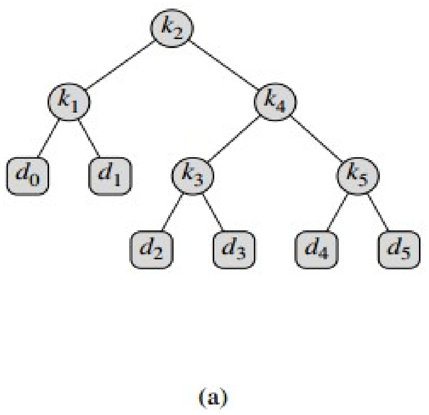
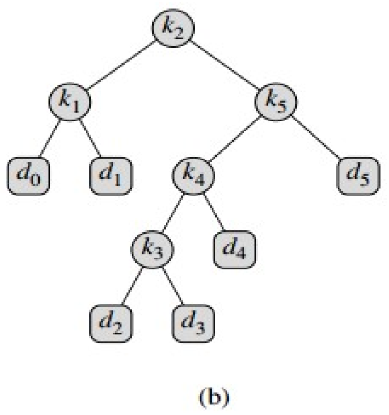



Figure 15.7 Two binary search trees for a set of $n = 5$ keys with the following probabilities:

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

(a) A binary search tree with expected search cost 2.80. **(b)** A binary search tree with expected search cost 2.75. This tree is optimal.

$$
\begin{aligned}
\mathrm{E}[\text{search cost in } T] &= \sum_{i=1}^{n} (\mathrm{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\mathrm{depth}_T(d_i) + 1) \cdot q_i \\
&= 1 + \sum_{i=1}^{n} \mathrm{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \mathrm{depth}_T(d_i) \cdot q_i \,, \qquad (15.16)
\end{aligned}
$$

where depth_T denotes a node's depth in the tree T. The last equality follows from equation (15.15). In Figure 15.7(a), we can calculate the expected search cost node by node:

node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

8. For a given set of probabilities, our goal is to construct a binary search tree whose expected search cost is smallest. We call such a tree an ***optimal binary search tree***.
9. We need to use the optimal substructure to show that we can construct an optimal solution to the problem from optimal solutions to subproblems.

10. Given keys $k_i, \dots, k_j$, one of these keys, say k_r ($i \le r \le j$), will be the root of an optimal subtree containing these keys.
11. The left subtree of the root k_r will contain the keys $k_i, \dots, k_{r-1}$ (and dummy keys $d_{i-1}, \dots, d_{r-1}$), and the right subtree will contain the keys $k_{r+1}, \dots, k_j$ (and dummy keys $d_r, \dots, d_j$).
12. Let us define $e[i, j]$ as the expected cost of searching an optimal binary search tree containing the keys $k_i, \dots, k_j$. Ultimately, we wish to compute $e[1, n]$.
13. The easy case occurs when $j = i - 1$. Then we have just the dummy key d_{i-1}. The expected search cost is $e[i, i - 1] = q_{i-1}$.
14. For a subtree with keys $k_i, \dots, k_j$, let us denote this sum of probabilities as

$$w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l \ .$$

15. Thus, if k_r is the root of an optimal subtree containing keys $k_i, \dots, k_j$, we have

$$e[i, j] = p_r + (e[i,r - 1] + w(i,r - 1)) + (e[r + 1, j] + w(r + 1, j))$$

16. Noting that

$$w(i, j) = w(i,r - 1) + p_r + w(r + 1, j)$$

17. we rewrite e[i, j] as

$$e[i, j] = e[i,r - 1] + e[r + 1, j] + w(i, j)$$

$$e[i, j] = \begin{cases} q_{i-1} & \text{if } j = i - 1 \ , \\ \min\limits_{i \le r \le j} \{e[i, r - 1] + e[r + 1, j] + w(i, j)\} & \text{if } i \le j \ . \end{cases}$$

18. OPTIMAL-BST(*p*, *q*, *n*)

```
1.  for $i \leftarrow 1$ to $n + 1$
2.      do $e[i, i - 1] \leftarrow q_i-1$
3.         $w[i, i - 1] \leftarrow q_i-1$
4.  for $l \leftarrow 1$ to $n$
5.      do for $i \leftarrow 1$ to $n - l + 1$
6.             do $j \leftarrow i + l - 1$
7.                $e[i, j] \leftarrow \infty$
8.                $w[i, j] \leftarrow w[i, j - 1] + p_j + q_j$
9.                for $r \leftarrow i$ to $j$
10.                   do $t \leftarrow e[i,r - 1] + e[r + 1, j] + w[i, j]$
11.                      if $t < e[i, j]$
12.                         then $e[i, j] \leftarrow t$
13.                              $root[i, j] \leftarrow r$
14. return e and root
```

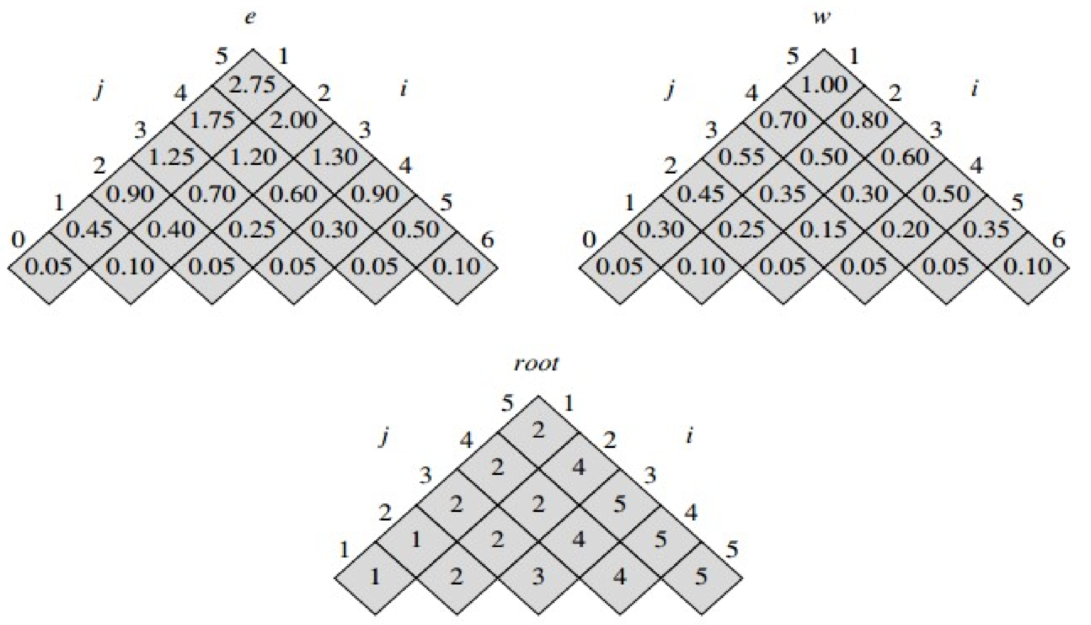


Figure 15.8 The tables $e[i, j]$, $w[i, j]$, and $root[i, j]$ computed by OPTIMAL-BST on the key distribution shown in Figure 15.7. The tables are rotated so that the diagonals run horizontally.

Reference – Introduction to Algorithms , CLSR