Chapter 3
Dynamic programming

- Dynamic programming also solve a problem by combining the solutions to subproblems.
- But dynamic programming considers the situation that some subproblems will be called repeatedly an thus need to avoid repeated work.

A typical application of the dynamic programming is for optimization problems.

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, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Rod cutting problem

The rod cutting problem is the following. Given a rod of length n inches and a table of price p_i for $i = 1, \ldots, n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces. The following is an example of price table.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	19	17	17	20	24	30

- For $n = 4$, we may cut as: $(1, 1, 1, 1), (1, 1, 2), (2, 2), (1, 3), (4)$, the correspondent prices are: $4, 7, 10, 9, 9$, respectively.
- So the optimal revenue is cutting the 4-inch rod into two 2-inch pieces.

By inspection, we can obtain the optimal decomposition as follows.

$$
\begin{aligned}
r_1 &= 1 && \text{from solution } 1 = 1 \text{ (no cuts)} \\
r_2 &= 5 && \text{from solution } 2 = 2 \text{ (no cuts)} \\
r_3 &= 8 && \text{from solution } 3 = 3 \text{ (no cuts)} \\
r_4 &= 10 && \text{from solution } 4 = 2 + 2 \\
r_5 &= 13 && \text{from solution } 5 = 2 + 3 \\
r_6 &= 17 && \text{from solution } 6 = 6 \text{ (no cut)} \\
r_7 &= 18 && \text{from solution } 7 = 1 + 6 \text{ or } 7 = 2 + 2 + 3 \\
r_8 &= 22 && \text{from solution } 8 = 2 + 6 \\
r_9 &= 25 && \text{from solution } 9 = 3 + 6 \\
r_{10} &= 30 && \text{from solution } 10 = 10 \text{ (no cuts)}
\end{aligned}
$$

In general, for a rod of length n, we can consider 2^{n-1} different cutting ways, since we have an independent option of cutting or not cutting at distance i inches from one end.

Suppose an optimal solution cuts the rod into k pieces with lengths $i_1, i_2, \ldots, i_k$. Then

$$n = i_1 + i_2 + \cdots + i_k$$

and the corresponding optimal revenue is

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}.$$

Our purpose is to compute r_n for given n and $p_i, i = 1, \ldots, n$.

When we consider dividing the problem, we can use the following method:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \cdots, r_{n-1} + r_1)$$

The first case is no cutting. The other cases consider optimal substructure: optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently.

A little simplify the above method, we can consider the cases that the first cut is of length i then

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i}).$$

In this formulation, the solution embodies the solution to only one related subproblem.

The following procedure implements the method. The inputs of the procedure are the the length n and the price $p[1, \ldots, n]$

```
1: procedure CUT-ROD(p, n)
2:     if n == 0 then
3:         return 0
4:     end if
5:     q = −∞
6:     for i = 1 to n do
7:         q = max(q, p[i]+CUT-ROD(p, n − i))
8:     end for
9:     return q
10: end procedure
```

Using a simple induction on n proves that the answer of the procedure is equal to r_n.

This procedure is very inefficient. This is because the CUT-ROD procedure calls itself recursively again and again.

Suppose the running time of the procedure is $T(n)$. Then we have the recurrence

$$T(n) \quad = \quad 1 + \sum_{j=0}^{n-1} T(j).$$

It is easy to prove that $T(n) = 2^n$ by mathematical induction. So the running time for CUT-ROD is exponential in n.

To see why the procedure is inefficient, we draw the recursion tree of CUT-ROD for $n = 4$ in Figure 1. In the tree, each vertex is a procedure calling. The number in the vertex is the parameter n.

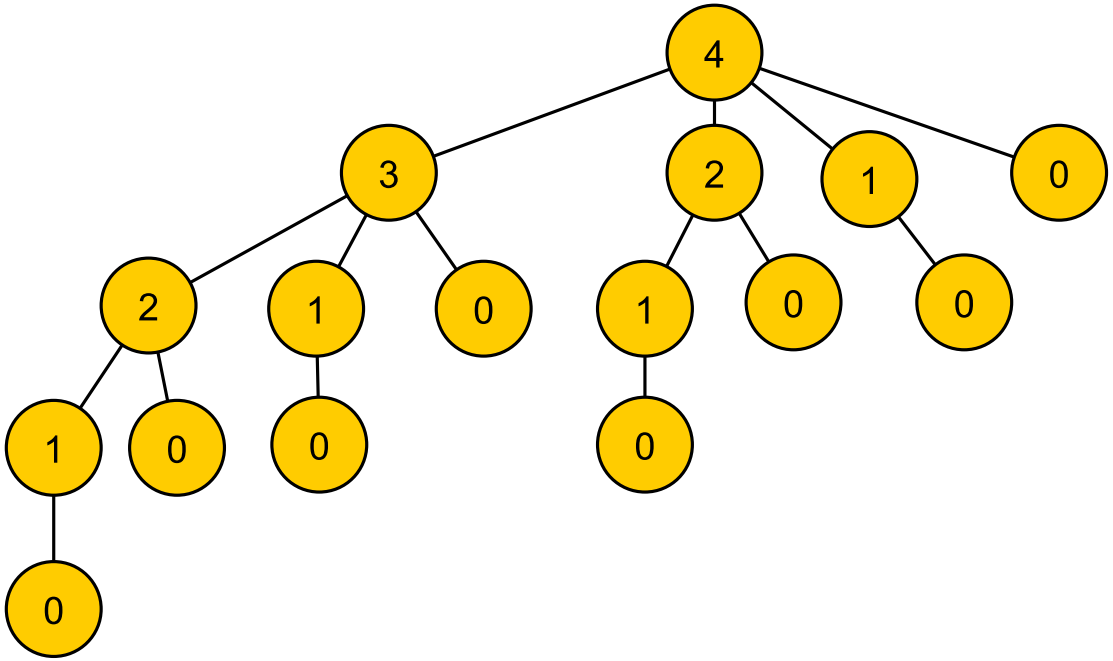

Figure 1: Recursion tree for CUT-ROD$(p, 4)$

- From the recursion tree, we see that the same subproblem is computed again and again.
- In this example, CUT-ROD$(p, 1)$ computed 4 times, CUT-ROD$(p, 0)$ computed 8 times, etc.
- To improve the method, we will use dynamic-programming method.
- The main idea of the dynamic-programming is to arrange for each subproblem to be solved only once. Each time a subproblem is solved, the result will be stored for the next calling. So next time, when we need to solve this subproblem, we need just look it up. Dynamic-programming uses additional memory to save the computation time.

There are two ways to implement a dynamic-programming approach.

- The first approach is top-down with memoization. In this approach, the procedure runs recursively in a nature manner, but modified to save the result of each subproblem (in an array or hash table). The procedure now first checks to see if the subproblem has previous solved or not. If so, it just returns the saved result; if not, the procedure computes the result in the usual manner and returns and saves the result.

- The second approach is the bottom-up method. This approach typically depends on some natural notion of the "size" of a subproblem, such that solving any particular subproblem depends only on solving "smaller" subproblems. We sort the subproblems by size and solve them in order, smallest first. Each subproblem is solved once. When solve a subproblem, the prerequisite subproblems are already solved.

The top-down approach for the cut rod problem is as follows.

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

```
1: procedure MEMOIZED-CUT-ROD-AUX((p, n))
2:     if r[n] ≥ 0 then
3:         return r[n]
4:     end if
5:     if n == 0 then
6:         q = 0
7:     else
8:         q = −∞
9:         for i = 1 to n do
10:            q = max(q, p[i]+MEMOIZED-CUT-ROD-AUX(p, n − i, r))
11:        end for
12:    end if
13:    r[n] = q
14:    return q
15: end procedure
```

- The main procedure MEMOIZED-CUT-ROD just initializes an auxiliary array r and then calls MEMOIZED-CUT-ROD-AUX.
- The later is the memoized version of the CUT-ROD. It returns the result from the auxiliary array if the result exists. Otherwise it computes the result.

The bottom-up version is as follows.

```
1: procedure BOTTOM-UP-CUT-ROD-AUX((p, n))
2:     let r[0...n] be a new array
3:     r[0] = 0
4:     for j = 1 to n do
5:         q = −∞
6:         for i = 1 to j do
7:             q = max(q, p[i] + r[j − i])
8:         end for
9:         r[j] = q
10:    end for
11:    return r[n]
12: end procedure
```

- The above procedure first creates a new array r, then calculate the values of r from the smallest to the largest.
- When compute $r[j]$, all the values of $r[j-i]$ have been computed. Therefor the line 7 just use these value instead of using recursive calling.

- The running time of the BOTTOM-UP-CUT-ROD is $\Theta(n^2)$, because there is a double-nested **for** loop.
- The running time of the top-down approach is also $\Theta(n^2)$.
- Although the line 10 of MEMOIZED-CUT-ROD-AUX uses recursive calling, each value of $r[i]$ just computes once. Therefore the total number of iterations of its for loop forms an arithmetic series, which gives total of $\Theta(n^2)$ iterations.

When we think about a dynamic-programming problem, it is important for us to understand the set of subproblems involved and how they depend on one another. We can use subproblem graph for these information. The Figure 2 is the subproblem graph for the cut rod problem with $n = 4$.

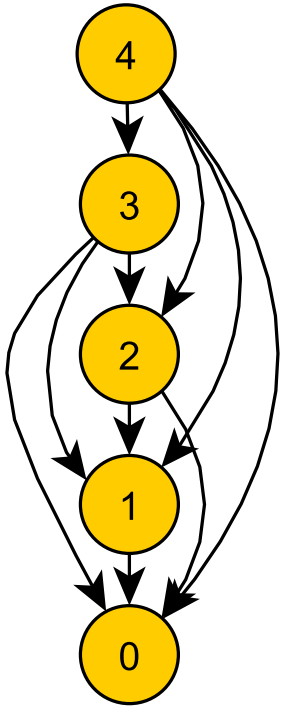

Figure 2: Subproblem graph for cut-rod problem

- The subproblem graph is a digraph, in which each vertex represents a distinct subproblem, and each arc represents that an optimal solution of a subproblem needs the solution of the other subproblem.
- For the top-down approach, in Figure 2 shows the vertex 4 needs a solution of vertex 3, the vertex 3 needs a solution of 2, etc.
- The bottom-up approach first solves the vertex 1 from vertex 0, then solves vertex 2 from vertices 0 and 1, etc.

- The size of the subproblem graph can help us determine the running time of the dynamic programming algorithm.
- Since each subproblem is solved only once, the running time is the sum of the times needed to solve each subproblem.
- Typically, the time to compute the solution of a subproblem is proportional to the degree of the corresponding vertex in the subproblem graph, and the number of subproblems is equal to the number of vertices in the graph. In this common case, the running time of dynamic programming is linear in the number of vertices and edges.

The above dynamic programming solutions of the cut rod problem just give the value of the optimal revenue, but not the actual solutions (how to cut the rod).

The following extended version of BOTTOM-UP-CUT-ROD not only returns the optimal value, but also returns a choice that led to the optimal value.

```
1:  procedure EXTENDED-BOTTOM-UP-CUT-ROD((p, n))
2:      let r[0...n] and s[0...n] be a new arrays
3:      r[0] = 0
4:      for j = 1 to n do
5:          q = -∞
6:          for i = 1 to j do
7:              if q < p[i] + r[j - i] then
8:                  q = p[i] + r[j - i]
9:                  s[j] = i     ▷ s[j] records the size of the first cut to
    the rod of size j
10:             end if
11:         end for
12:         r[j] = q
13:     end for
14:     return  r and s
15: end procedure
```

```
1: procedure PRINT-CUT-ROD-SOLUTION((p, n))
2:     (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
3:     while n > 0 do
4:         print s[n]
5:         n = n − s[n]
6:     end while
7: end procedure
```

The PRINT-CUT-ROD-SOLUTION prints the solution: it prints the size of first cut of the optimal solution, and then recursively prints out the first cut of the remainder rod for its optimal solution.

Matrix-chain multiplication

Suppose n matrices $A_1, A_2, \ldots, A_n$ are given, where the matrices are not necessary square. We need to compute the product

$$A_1 A_2 \cdots A_n.$$

```
1: procedure MATRIX-MULTIPLY(A, B)
2:     if A.columns ≠ B.rows then
3:         error "incompatible dimensions"
4:     else
5:         let C be a new A.rows × B.columns matrix
6:         for i = 1 to A.rows do
7:             for j = 1 to B.columns do
8:                 c_ij = 0
9:                 for k = 1 to A.columns do
10:                    c_ij = c_ij + a_ik · b_kj
11:                end for
12:            end for
13:        end for
14:        return C
15:    end if
16: end procedure
```

- The main cost of the procedure is from line 6 to line 13, which is $A.rows \times B.columns \times A.columns$ scalar multiplications.
- When we compute the multiplication of more than two matrices, the order of the multiplication will effect the cost.

Consider the multiplication $A_1A_2A_3$. Suppose the dimensions of A_1, A_2, A_3 are $10 \times 100, 100 \times 5, 5 \times 50$, respectively.

If we multiply according to the parenthesization $((A_1A_2)A_3)$, then we first perform $10 \cdot 100 \cdot 5 = 5000$ scalar multiplications to compute a 10×5 matrix. Then multiply the resulting matrix with A_3, which needs $10 \cdot 5 \cdot 50 = 2500$ scalar multiplications. In this way, we need total 7500 multiplications.

But if we compute the multiply as $(A_1(A_2A_3))$, then a simple calculation shows that we need a total 75000 scalar multiplications.

Matrix-chain multiplication problem: Given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \ldots A_n$ in a way that minimizes the number of scalar multiplications.

This problem is to find out an optimal order of products for a matrix-chain multiplications.

First let us see the number of possible parenthesizations.

Denote the number of alternative parenthesizations of a sequence of n matrices by $P(n)$. Then $P(1) = 1$. 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 kth and $(k+1)$st matrices for any $k = 1, 2, \ldots, n-1$. Thus we have

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

Using the substitution method, we can show that the solution to the recurrence (1) is $\Omega(2^n)$. So a exhaustive search will be exponential in n.

We can see that in the recurrence (1), many value of $P(i)$ is computed repeatedly. Therefore we can apply the dynamic programming.

Step 1: The structure of an optimal parenthesization

Let $A_{i..j}$, where $i \leq j$, denote the matrix that results from evaluating the product $A_i A_{i+1} \cdots A_j$. When $i < j$, we need to split the product into two products and compute $A_{i..k}$ and $A_{k+1..j}$ for some $i \leq k < j$, and then compute $A_{i..k} A_{k+1..j}$. The cost thus is the sum of the costs of compute $A_{i..k}, A_{k+1..j}$ and $A_{i..k} A_{k+1..j}$.

The optimal substructure of this problem is as follows. Suppose that to optimally parenthesize $A_i A_{i+1} \cdots A_j$, we split the product between A_k and A_{k+1}. Then the subchain $A_i A_{i+1} \cdots A_k$ within this parenthesize must be optimal.

Otherwise, if we have a better parenthesize of $A_i A_{i+1} \cdots A_k$ then we can get a better parenthesize of $A_i A_{i+1} \cdots A_j$. By the similar argument, the parenthesize of $A_{k+1} A_{k+2} \cdots A_j$ is also optimal.

We can split the matrix-chain problem into two subproblems and find out the optimal solutions of these two subproblems.

Step 2: A recursive solution

Let $m[i,j]$ be the minimum number of scalar multiplications needed to compute $A_i A_{i+1} \cdots A_j$. Then we want to build $m[i,j]$ recursively. If we split $A_i A_{i+1} \cdots A_j$ between A_k and A_{k+1}, then the minimum costs will be $m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j. \end{cases}$$

- The $m[i, j]$ value gives the costs of optimal solution to problems, but they do not provide the construction of the optimal solution.
- We need to know the value of k which we used to split the product.
- We define $s[i, j]$ to be a value of k at which we split the product $A_i A_{i+1} \cdots A_j$ in an optimal parenthesization.

Step 3: Computing the optimal costs

From the recurrence, now the task is to find out the value $m[1, n]$, which depends on $m[i, j]$ for smaller chains with length $j - i$. So it is suitable to use the bottom-up method, i.e., start from computing $m[i, j]$ for smaller $l = j - i$.

Instead of use a recursive algorithm based on recurrence, we use a tabular, bottom-up method to compute the costs.

Suppose $p = \langle p_0, p_1, \ldots, p_n \rangle$, where $p.length = n + 1$, which defines the dimensions of the matrices. Let $m[1..n, 1..n]$ be an auxiliary table for store the values of $m[i, j]$ and $s[1..n-1, 2, ..n]$ be a table for store the indexes k that achieved the optimal cost in computing $m[i, j]$.

```
 1: procedure MATRIX-CHAIN-ORDER(p)
 2:     n = p.length − 1
 3:     let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 4:     for i = 1 to n do
 5:         m[i, i] = 0                                  ▷ only has one matrix
 6:     end for
 7:     for l = 2 to n do                                ▷ l is the chain length
 8:         for i = 1 to n − l + 1 do
 9:             j = i + l − 1
10:             m[i, j] = ∞
11:             for k = i to j − 1 do
12:                 q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
13:                 if a < m[i, j] then
14:                     m[i, j] = q
15:                     s[i, j] = k
16:                 end if
17:             end for
18:         end for
19:     end for
20:     return m and s
21: end procedure
```

- The main cost for the procedure is the three nested **for** loops, which yields a running time of $O(n^3)$.
- We can prove that the running time is also $\Omega(n^3)$.
- The space requirement is $\Theta(n^2)$.

Step 4: Constructing an optimal solution

Now we are able to give the optimal solution (the optimal parenthesizing the chain), because we have determined the value k in $s[i, j]$.

```
1: procedure PRINT-OPTIMAL-PARENS(s, i, j)
2:     if i == j then
3:         print "A_i"
4:     else
5:         print "("
6:         PRINT-OPTIMAL-PARENS(s, i, s[i, j])
7:         PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
8:         print ")"
9:     end if
10: end procedure
```

Example Suppose the following matrix-chain is given (p given)

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

Call MATRIX-CHAIN-ORDER(p) and
PRINT-OPTIMAL-PARENS($s, 1, 6$) prints the parenthesization
$((A_1(A_2A_3))((A_4A_5)A_6))$.

Elements of dynamic programming

There are two key ingredients that an optimization problem must have to apply dynamic programming: optimal substructure and overlapping subproblems.

Optimal substructure means some algorithm that solves the optimal problem will depends on some optimal solutions of the subproblems. So we need not only to find some recursive method, but the recursive also on optimal problems.

Two factors may effect the running time for a dynamic programming algorithm: the number of subproblems overall and how many choices we look at for each subproblem. The subproblem graph gives a way to do the analysis.

Dynamic programming often uses optimal substructure in a bottom-up fashion. That is, first find optimal solutions for subproblems and then solve the original problem based on these solutions of subproblems. One property of these solutions of subproblems is that they are independent to the problem.

Typically, the total number of distinct subproblems is a polynomial in the input size, while the recursive algorithm revisits the same subproblem repeatedly.

In contrast, a problem for which a divide-and-conquer approach is suitable usually generates brand-new problems at each step of recursion.

For the matrix-chain product problem, if we use the simple recursive method without using the auxiliary table, then the running time will be $\Omega(2^n)$. In fact, the distinct subproblems in the matrix-chain product problem is $\Theta(n^2)$.

Longest common subsequence

Biological applications often need to compare the DNA of two different organisms. A strand of DNA consists of a string of molecules called bases, where the possible bases are adenine, guanine, cytosine and thymine. Usually DNA strands are expressed as a string over the finite set {`A`,`C`,`G`,`T`}. For example, the DNA of one organism may be

$S_1 =$ `ACCGGTCGAGTGCGCGGAAGCCGGCCGAA`,

Another organism may be

$S_2 =$ `GTCGTTCGGAATGCCGTTGCTCTGTAAA`.

One reason to compare two strands of DNA is to determine how closely related the two organisms are.

There are different ways to define the similarity of DNA strands. Here we consider one of the definitions. The method is to find a strand S_3 in which the bases in S_3 appear in each of S_1 and S_2, these bases must appear in the same order, but not consecutively. The longer the strand S_3 we can find, the more similar S_1 and S_2 are. In the above example, the longest strand S_3 is `GTCGTCGGAAGCCGGCCGAA`.

Formally, given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, another sequence $Z = \langle z_1, z_2, \ldots, z_n \rangle$ is a subsequence of X if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of index of X such that $x_{i_j} = z_j$ for $j = 1, 2, \ldots, k$.

For example, $Z = \langle B, C, D, B \rangle$ is a subsequence of $X = \langle A, B, C, B, D, A, B \rangle$ with corresponding index sequence $\langle 2, 3, 5, 7 \rangle$. 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.

Now we consider the longest-common-subsequence problem (LCS problem). We are given two sequences X and Y, and wish to find a maximum length common subsequence of X and Y. We will use the dynamic programming to solve the problem step by step.

Step 1: Characterizing a longest common subsequence

It is not suitable to use a brute-force method the check all possible subsequences of a sequence when we try to solve the LCS problem, because there are 2^m subsequences of the sequence of length m.

So first we need to look for some optimal-substructures. we have the following theorem to use. For a sequence $X = \langle x_1, x_2 \ldots, x_m \rangle$, we will use X_i to denote the sequence $\langle x_1, x_2, \ldots, x_i \rangle$ for $i = 0, 1, \ldots, m-1$, where X_0 is an empty sequence.

Theorem [Optimal substructure of an LCS]

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y. Then

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}.

Step 2: A recursive solution

Above Theorem tells us, if $x_m = y_n$, then we need to find the LCS for X_{m-1} and Y_{n-1}. If $x_m \neq y_n$, then we need to find two LCSs for X_{m-1} and Y, and X and Y_{n-1}.

Using a similar idea for solving the matrix-chain product problem, we define $c[i, j]$ be the length of an LCS of the sequence X_i and Y_j. Then we have the following.

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

Note that in this problem, a condition in the problem restricts which subproblem we need to consider, that is different from the previous examples.

Step 3: Computing the length of an LCS

The idea of the procedure:

- The input are $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$.
- Table c used to record $c[i, j]$. We need another table b to help us construct the optimal solution.
- If $x_i = y_j$, then put x_i into the solution sequence. Otherwise, we need to reduce the value of i or j according to the values of the two subproblems.
- We use arrows in table c to indicate how to construct the LCS.

```
1:  procedure LCS-Length(X, Y)
2:      m = X.length
3:      n = Y.length
4:      let b[1..m, 1..n] and c[0..m, 0..n] be new tables
5:      for i = 1 to m do
6:          c[i, 0] = 0
7:      end for
8:      for j = 0 to n do
9:          c[0, j] = 0
10:     end for
11:     for i = 1 to m do
12:         for j = 1 to n do
13:             if x_i == y_j then
14:                 c[i, j] = c[i − 1, j − 1] + 1
15:                 b[i, j] = " ↖ " ▷ put x_i to sequence, reduce i and j
16:             else if c[i − 1, j] ≥ c[i, j − 1] then
```

```
17:                 c[i, j] = c[i − 1, j]
18:                 b[i, j] = " ↑"                          ▷ reduce i
19:             else
20:                 c[i, j] = c[i, j − 1]
21:                 b[i, j] = " ←"                          ▷ reduce j
22:             end if
23:         end for
24:     end for
25:     return c and b
26: end procedure
```

The running time for this procedure is $\Theta(mn)$, since each table entry takes constant time $\Theta(1)$.

Step 4: Constructing an LCS

We use an example to explain the notations of b:

Let $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. Then the table b is as follows.

	1	2	3	4	5	6
1	↑	↑	↑	↖	←	↖
2	↖	←	←	↑	↖	←
3	↑	↑	↖	←	↑	↑
4	↖	↑	↑	↑	↖	←
5	↑	↖	↑	↑	↑	↑
6	↑	↑	↑	↖	↑	↖
7	↖	↑	↑	↑	↖	↑

To construct the LCS, we start at the right bottom corner of the table. When $b[i, j]$ is "$\uparrow$", we go to up row (reduce i), when $b[i, j]$ is "$\leftarrow$", we go to left column (reduce j), and when $b[i, j]$ is "$\nwarrow$", we record x_i (which equals to y_j) and go up left (reduce both i and j). For this example, the red arrows indicate the path we follow. We will record (in a reverse order): x_6, x_4, x_3, x_2. So the LCS is $\langle B, C, B, A \rangle$.

We can use the follow procedure to print out the LCS. The initial call is PRINT-LCS$(b, X, X.length, Y.length)$.

```
1:  procedure PRINT-LCS(b, X, i, j)
2:      if i == 0 or j == 0 then
3:          return
4:      end if
5:      if b[i, j] == "↖" then
6:          PRINT-LCS(b, X, i − 1, j − 1)
7:          print x_i
8:      else if b[i, j] == "↑" then
9:          PRINT-LCS(b, X, i − 1, j)
10:     else
11:         PRINT-LCS(b, X, i, j − 1)
12:     end if
13: end procedure
```

- The procedure takes time $O(m+n)$, since it decrements at least one of i and j in each recursive call.
- We need $\Theta(mn)$ space to store tables c and b.
- The procedure can be slightly improved, since just small part of the table b are useful for the solution. We can use just $m+n$ storage to store the useful part of the information of b.
- If we just need to know the length of LCS, then we can reduce the asymptotic space requirement.

Optimal binary search trees

Binary search tree is a binary tree, in which the keys in the left subtree is less than the key in the root while keys in the right subtree is greater than the key in the root, and a subtree of binary search tree is also a binary search tree. There are some methods to make the binary search tree balance so that the search time will be more efficient, such an AVL tree.

Now we consider a more general case. Suppose we have a sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct keys in sorted order (i.e., $k_1 < k_2 < \cdots < k_n$). For each key k_i, the probability a search will be on k_i is p_i. We wish to build a binary search tree for these keys such that the expected search time (the average search time) is optimal. We also need to consider the search values that are not in K. So we have $n + 1$ dummy keys $d_0, d_1, d_2 \ldots d_n$, where, d_i, $0 < i < n$, represents the values between k_i and k_{i+1}, d_0 represent the values less than k_1 and d_n represents the values greater than k_n. For each dummy key d_j, we assume the probability for searching according to it is q_j. So we have

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

Suppose we have already established the binary search tree T (in the tree, dummy keys should be leaves). Then we have the expected cost of a search in T is

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

where $depth_T$ denotes a node's depth in the tree T. If the expected search cost is the smallest, then we call T an optimal binary search tree.

Example A small binary search tree for a set of $n = 5$ keys.

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

Two binary search trees are displayed in Figure 3. The first tree has the expected search cost 2.80 and the second tree has the expected search cost 2.75, which is optimal.

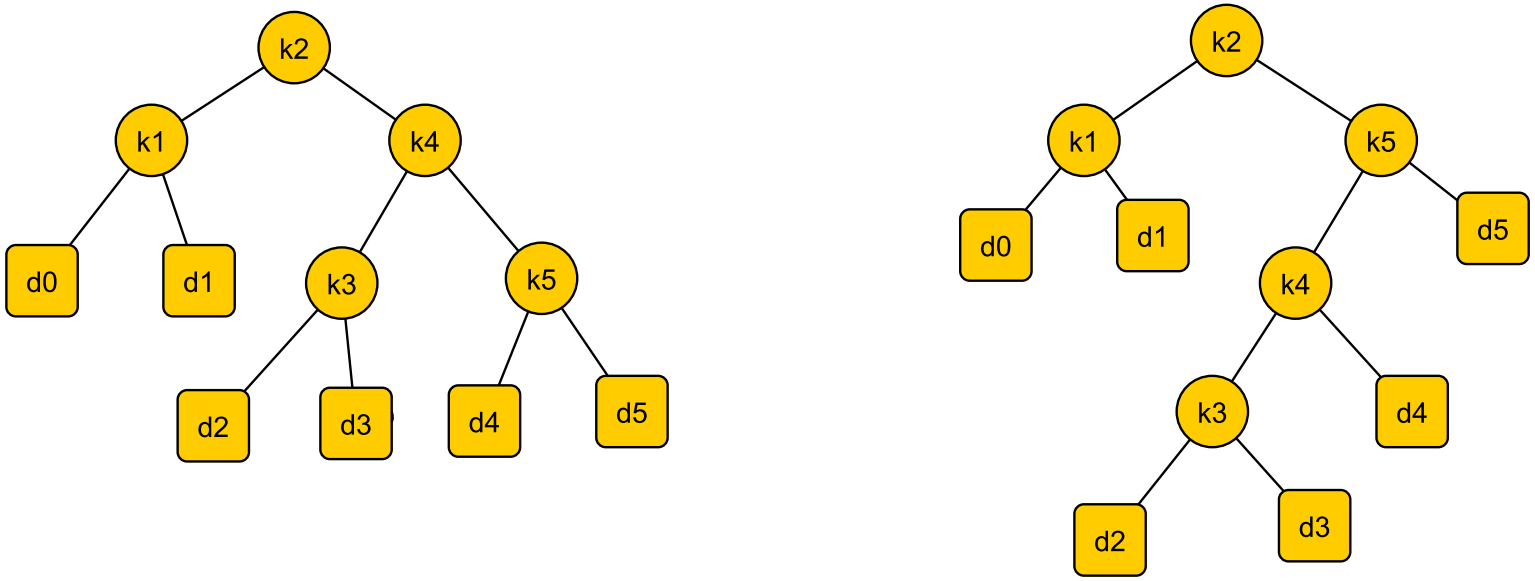

Figure 3: BSTs for a set of $n = 5$

To construct the tree, we can first construct a binary search tree with the n keys, then add the dummy nodes to leaves. But the number of binary search tree with n nodes is $\Theta(4^n/n^{3/2})$. So exhaustive search is not feasible. We can consider to use dynamic programming.

Step 1: The structure of an optimal binary search tree

Suppose we have constructed an optimal binary search tree. Then each subtree must contain keys in a contiguous range $k_i, k_{i+1}, \ldots, k_j$, for some $1 \leq i \leq j \leq n$. In addition, that subtree must also contains the leaves of dummy keys $d_{i-1}, d_i, \ldots, d_j$.

Therefore we have the optimal substructure: if an optimal binary search tree T has a subtree T' containing keys $k_i, \ldots, k_j$, then T' must be optimal as well for subproblem with keys $k_i, \ldots, k_j$ and dummy keys $d_{i-1}, \ldots, d_j$. Otherwise we can replace the subtree with better expected cost and that means that T is not optimal.

Considering the recursive method, if a subtree contains keys $k_i, \ldots, k_j$ and the root is k_r, then the left subtree contains keys $k_i, \ldots, k_{r-1}$ (and dummy keys $d_{i-1}, \ldots, d_{r-1}$) and the right subtree contains keys $k_{r+1}, \ldots, k_j$ (and dummy keys $d_r, \ldots, d_j$). When the root is i, then the left subtree contains only d_{i-1} and when k_j is the root, its right subtree contains only d_j. We may try every possible key as the root to obtain the optimal subtree.

Step 2: A recursive solution

We can define the values of optimal solution for subtrees as follows. For a subtree with keys $k_i, \ldots, k_j$, define $e[i, j]$ to be the optimal expected cost of searching, where $i \geq 1, i - 1 \leq j \leq n$. Here we define $e[i, i-1]$ as the subtree with d_{i-1} as a only node. So $e[i, i-1] = q_{i-1}$.

When $j \geq i$, we need to select a root k_r, which forms two subtrees, one with the keys $d_i \ldots, d_{r-1}$ and another with the keys $d_{r+1}, \ldots, j$. For a tree containing keys $k_s, \ldots, k_t$ the optimal value is $e[s,t]$. But when it becomes a subtree, the depth of each vertex will increase one. Therefore the the expected costs for this subtree will be $e[s,t] + \sum_{l=s}^{t} p_l + \sum_{l=s-1}^{t} q_l$. Define

$$w(s,t) = \sum_{l=s}^{t} p_l + \sum_{l=s-1}^{t} q_l \qquad (2)$$

Then if k_r is the root of an optimal subtree containing keys $k_i, \ldots, k_j$, we have

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

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

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

Now we have the recursive formula for $e[i,j]$.

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

To help us to keep the track of the structure of optimal binary search tree, we define $root[i, j]$ to be the index r for which k_r is the root of an optimal binary search tree containing keys $k_i, \ldots, k_j$.

Step 3: Computing the expected search cost of an optimal BST

Similar to other dynamic programming, we need to use some tables to store the solutions for subproblems. So we define tables e, w and *root* in the following procedure. For e and w we need to define $1 \leq i \leq n+1, 0 \leq j \leq n$, because we need to record the values of "empty" subtrees (e.g., $e[i, i-1], 1 \leq i \leq n$).

```
1:  procedure OPTIMAL-BST(p, q, n)
2:      let e[1..n+1, 0..n], w[1..n+1, 0..n] and root[1..n, 1..n] be
    new tables
3:      for i = 1 to n + 1 do                    ▷ initial empty subtrees
4:          e[i, i − 1] = q_{i−1}
5:          w[i, i − 1] = q_{i−1}
6:      end for
7:      for l = 1 to n do
8:          for i = 1 to n − l + 1 do
9:              j = i + l − 1
10:             e[i, j] = ∞
11:             w[i, j] = w[i, j − 1] + p_j + q_j
12:             for r = i to j do
13:                 t = e[i, r − 1] + e[r + 1, j] + w[i, j]
14:                 if t < e[i, j] then
15:                     e[i, j] = t
```

```
16:                     root[i, j] = r
17:                 end if
18:             end for
19:         end for
20:     end for
21:     return e and root
22: end procedure
```

The OPTIMAL-BST procedure takes $\Theta(n^3)$ time. Because the main costs are the three nested **for** loops, each loop index takes at most n values, the running time is $O(n^3)$. On the other hand, we can also see that the procedure takes $\Omega(n^3)$ time.