

# Chapter 15 Solution

[github.com/yirong-c/CLRS](https://github.com/yirong-c/CLRS)

11/3/2021

## 15.1

### 15.1-1

**Proof.** We prove by substitution method. For  $n = 0$ ,  $T(0) = 2^0 = 1$ . For  $n > 0$ ,

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) = 1 + \sum_{j=0}^{n-1} 2^j = 1 + (2^n - 1) = 2^n$$

□

### 15.1-2

Consider the following case:

length $i$	1	2	3
price $p_i$	1	6	8
density $p_i/i$	1	3	2.67

If we use “greedy” strategy, our solution will be “2 1”, and the total price will be 7. However, the optimal way is “3”, and the total price is 8.

### 15.1-3

```
1      /**
2      * running time:  $O(n^2)$ 
3      *  $p$ : table of prices (index start from 0)
4      *  $n$ : length of rod
5      *  $c$ : cost of each cut
6      * return maximum revenue
7      */
8      int BottomUpCutRodWithCost(const std::vector<int>& p, int n, int c)
9      {
10         int *r, q, i, j;
11         r = new int[n + 1];
12         r[0] = 0;
13         for (j = 1; j <= n; ++j)
14         {
```

```
15         q = p[j - 1];
16         for (i = 0; i < j - 1; ++i)
17             q = std::max(q, p[i] + r[j - i - 1] - c);
18         r[j] = q;
19     }
20     delete[] r;
21     return q;
22 }
```

## 15.1-4

```
1  /**
2   * p: table of prices (index start from 0)
3   * n: length of rod
4   * r: table of maximum revenue (index start from 1)
5   * s: table of optimal size i of the first piece to cut off (index start from 1)
6   * return maximum revenue
7   */
8  int ExtendedMemoizedCutRodAux(const std::vector<int>& p, int n, int *r, int *s)
9  {
10     int q, i, reminder_r;
11     if (r[n] >= 0) return r[n];
12     q = INT_MIN;
13     for (i = 0; i < n; ++i)
14     {
15         reminder_r = ExtendedMemoizedCutRodAux(p, n - i - 1, r, s);
16         if (q < p[i] + reminder_r)
17         {
18             q = p[i] + reminder_r;
19             s[n] = i + 1;
20         }
21     }
22     r[n] = q;
23     return q;
24 }
25
26 /**
27  * running time:  $O(n^2)$ 
28  * p: table of prices (index start from 0)
29  * n: length of rod
```

```
30     * return (r, s)
31     * r: table of maximum revenue (index start from 1)
32     * s: table of optimal size i of the first piece to cut off (index start from 1)
33     * caller is responsible to deallocate return value r and s
34     */
35     std::pair<int*, int*> ExtendedMemoizedCutRod(const std::vector<int>& p, int n)
36     {
37         int *r, *s, i;
38         r = new int[n + 1];
39         s = new int[n + 1];
40         r[0] = 0;
41         s[0] = 0;
42         for (i = 1; i <= n; ++i) r[i] = INT_MIN;
43         ExtendedMemoizedCutRodAux(p, n, r, s);
44         return std::make_pair(r, s);
45     }
```

### 15.1-5

```
1     /**
2     * running time:  $O(n)$ 
3     * n: n-th fibonacci number (must greater than 0)
4     */
5     int FibonacciNumber(int n)
6     {
7         int *f, i, result;
8         f = new int[n + 1];
9         f[0] = 0;
10        f[1] = 1;
11        for (i = 2; i <= n; ++i)
12            f[i] = f[i - 1] + f[i - 2];
13        result = f[n];
14        delete[] f;
15        return result;
16    }
```

## 15.2

### 15.2-1

Optimal parenthesization:  $((1, 2), ((3, 4), (5, 6)))$

Minimum cost: 2010

## 15.2-2

```
1      /**
2      * s: table (2d) storing index of k achieved the optimal cost
3      *      (index start by 1)
4      * caller is responsible to deallocate the return value
5      */
6      Matrix* MatrixChainMultiply
7          (const std::vector<Matrix*>& matrices, const Table* s, int i, int j)
8      {
9          Matrix *matrix_a, *matrix_b, *matrix_c;
10         if (i == j)
11         {
12             return matrices[i - 1];
13         }
14         matrix_a = MatrixChainMultiply(matrices, s, i, (*s)[i][j]);
15         matrix_b = MatrixChainMultiply(matrices, s, (*s)[i][j] + 1, j);
16         matrix_c = MatrixMultiply(matrix_a, matrix_b);
17         if (i != (*s)[i][j]) delete matrix_a;
18         if ((*s)[i][j] + 1 != j) delete matrix_b;
19         return matrix_c;
20     }
```

## 15.2-3

**Proof.** We prove by substitution method. For  $n = 1$ ,  $P(1) = 1 \geq 2^k$  for  $k \leq 0$ . For  $n \geq 2$ ,

$$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k) \geq \sum_{k=1}^{n-1} (c \cdot 2^k)(c \cdot 2^{n-k}) = \sum_{k=1}^{n-1} (c^2 \cdot 2^n) = (n-1)(c^2 \cdot 2^n) \geq c^2 \cdot 2^n$$

for some constant  $c$ . □

## 15.2-4

For all vertices  $v_{i,j}$  in the graph, it contains edge  $(v_{i,j}, v_{i,k})$  and  $(v_{i,j}, v_{k+1,j})$  for all  $i \leq k < j$ .

Vertices:

$$\binom{n}{2} + n = \frac{n(n-1)}{2} + n = \frac{n(n+1)}{2}$$

Edges:

$$\begin{aligned} \sum_{i=1}^n \sum_{j=i}^n (j-i) &= \sum_{i=1}^n \left( \sum_{j=i}^n (j) - \sum_{j=i}^n (i) \right) = \sum_{i=1}^n \left( \sum_{j=1}^n (j) - \sum_{j=1}^{i-1} (j) - (n-i+1)i \right) \\ &= \sum_{i=1}^n \left( \frac{n(n+1)}{2} - \frac{(i-1)i}{2} - (n-i+1)i \right) = \frac{n^2(n+1)}{2} - \sum_{i=1}^n \left( \frac{i-i^2+2ni}{2} \right) \end{aligned}$$

$$\begin{aligned}
 &= \frac{n^2(n+1)}{2} + \frac{1}{2} \sum_{i=1}^n (i^2) - \frac{1}{2}(1+2n) \sum_{i=1}^n (i) = \frac{n^2(n+1)}{2} + \frac{n(n+1)(2n+1)}{12} - \frac{n(n+1)(2n+1)}{4} \\
 &= \frac{n^2(n+1)}{2} - \frac{n(n+1)(2n+1)}{6} = \frac{(n-1)n(n+1)}{6}
 \end{aligned}$$

### 15.2-5

**Proof.** Notice that  $\sum_{i=1}^n \sum_{j=i}^n R(i, j)$  is equal to the total times of any entries are referenced during the entire call of MATRIX-CHAIN-ORDER. In other words, it is equal to twice the times of line 10 was executed during the entire call.

Hence, we have

$$\begin{aligned}
 \sum_{i=1}^n \sum_{j=i}^n R(i, j) &= \sum_{l=2}^n \sum_{i=1}^{n-l+1} \sum_{k=i}^{i+l-2} 2 = 2 \sum_{l=2}^n (n-l+1)(l-1) = 2((n+2) \sum_{l=2}^n l - \sum_{l=2}^n l^2 - (n-1)(n+1)) \\
 &= 2((n+2)(\frac{n(n+1)}{2} - 1) - (\frac{n(n+1)(2n+1)}{6} - 1)) = \frac{n^3-n}{3} \quad \square
 \end{aligned}$$

### 15.2-6

**Proof.** We prove by induction. Let  $P(n)$  be the claim: A full parenthesization of an  $n$ -element expression has exactly  $n-1$  pairs of parentheses.

(Base Case) A 2-element full parenthesization  $(A_1, A_2)$  has only one pair of parentheses clearly. Hence, we have proved  $P(2)$  is true.

(Induction Step) Suppose that  $P(n)$  is true. Let  $C$  be a sequence with  $n+1$  elements:  $A_1 A_2 \dots A_n A_{n+1}$ . Delete one arbitrary element from  $C$ , we have a sequence with  $n$  elements. By induction hypothesis,  $C$  ( $n$ -element) has exactly  $n-1$  pairs of parentheses now. Add the deleted element back, we can add one pair of parentheses to surround the deleted element and one of the element's neighbor element or one of the element's neighbor parenthesization. This says,  $C$  ( $n+1$ -element) has exactly  $n$  pairs of parentheses now. We have proved  $P(n+1)$  is true.  $\square$

## 15.3

### 15.3-1

RECURSIVE-MATRIX-CHAIN is a more efficient way.

**Proof.** By recurrence (15.6) in section 15.2, there are  $P(n)$  alternative parenthesizations of a sequence of matrices where

$$P(n) = \begin{cases} 1 & \text{if } n = 1. \\ \sum_{k=1}^{n-1} P(k)(n-k) & \text{if } n \geq 2. \end{cases} \quad (15.6)$$

By problem 12-4, we proved that  $P(n) = \Omega(4^n/n^{3/2})$ . This says enumerating takes  $\Omega(4^n/n^{3/2})$  time. In order to prove that RECURSIVE-MATRIX-CHAIN is a more efficient than enumerating, we just need to prove that RECURSIVE-MATRIX-CHAIN takes  $o(4^n/n^{3/2})$  time.

By recurrence (15.7) in section 15.2, RECURSIVE-MATRIX-CHAIN takes  $T(n)$  times where

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

$$T(n) = 2 \sum_{i=1}^{n-1} T(i) + n$$

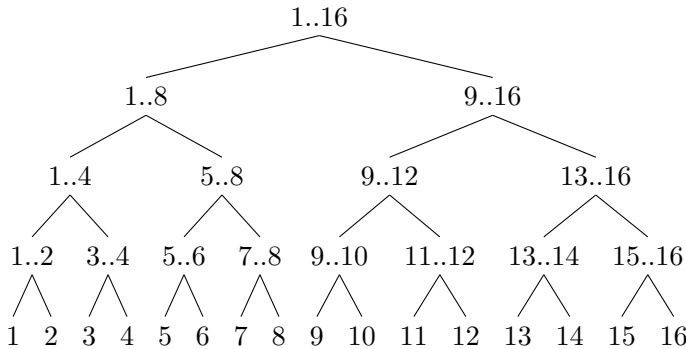
This says we want to prove that  $T(n) = o(4^n/n^{3/2})$ . Since  $\lim_{n \rightarrow \infty} \frac{4^n/n^{3/2}}{3.5^n} = \infty$ , we just need to prove that  $T(n) = O(3.5^n)$ . ( $T(n) = O(3^n)$  is false, so we try  $T(n) = O(3.5^n)$ .)

We claim that  $T(n) = O(3.5^n)$  and prove this by substitution method. Let  $c$  be some constant. Assume  $T(n) \leq c \cdot 3.5^n$ .

$$\begin{aligned} T(n) &= 2 \sum_{i=1}^{n-1} T(i) + n \\ &\leq 2 \sum_{i=1}^{n-1} (c \cdot 3.5^i) + n \\ &= 2c \left( \frac{3.5^n - 1}{3.5 - 1} - 1 \right) + n \\ &= 2c \left( \frac{3.5^n - 3.5}{2.5} \right) + n \\ &= 0.8c \cdot 3.5^n - 2.8c + n \end{aligned}$$

Let  $c = 1$ . We have  $0.8c \cdot 3.5^n - 2.8c + n \leq c \cdot 3.5^n$ . We have proved  $T(n) = O(3.5^n)$ . Hence,  $T(n) = o(4^n/n^{3/2})$ .  $\square$

### 15.3-2



We notice that there is no overlapping subproblem, so memoization does not help to speed up the algorithm.

### 15.3-3

Yes.

**Proof.** Let  $A_{i...j}$  denotes sequence of matrices  $A_i A_{i+1} \dots A_j$ .

The subproblems in maximize multiplication are independent. (for more information, refer to page 383) An optimal parenthesization of  $A_{i...j}$  that splits the product between  $A_k$  and  $A_{k+1}$  contains within it optimal solutions to the problems of parenthesizing  $A_{i...k}$  and  $A_{k+1...j}$ .

Given parenthesization  $P_{ij}$  maximize the number of scalar multiplications to  $A_{i...j}$ , and  $P_{ij}$  splits the product between  $A_k$  and  $A_{k+1}$ . Making the choice (splits the product between  $A_k$  and  $A_{k+1}$ ) leaves subproblems  $A_{i...k}$  and  $A_{k+1...j}$  to solve. Let  $P_{ik}$  and  $P_{k+1,j}$  be the parenthesization on  $A_{i...k}$  and  $A_{k+1...j}$  respectively. We want to prove that  $P_{ik}$  and  $P_{k+1,j}$  maximize the number of scalar multiplications to  $A_{i...k}$  and  $A_{k+1...j}$  by contradiction. Suppose that  $P_{ik}$  and  $P_{k+1,j}$  does not maximize the number of scalar multiplications to  $A_{i...k}$  and  $A_{k+1...j}$ . Let  $Q_{ik}$  and  $Q_{k+1,j}$  be the optimal parenthesization (maximize number of multiplications) to  $A_{i...k}$  and  $A_{k+1...j}$ . Then, by “cutting out”  $P_{ik}$  and  $P_{k+1,j}$  and “pasting in”  $Q_{ik}$  and  $Q_{k+1,j}$ , we get a better solution (more number of multiplications) to the original problem than  $P_{ij}$ . This contradicts to parenthesization  $P_{ij}$  maximize the number of scalar multiplications.  $\square$

### 15.3-4

Consider the following  $p$ 's:

$p_0$	$p_1$	$p_2$	$p_3$
1	10	20	100

By the approach of greedy algorithm, we choose  $k = 1$  for  $[i, j] = [1, 3]$  since  $p_0 p_1 p_3 = 1000$  and  $p_0 p_2 p_3 = 2000$ . Hence, the solution of greedy algorithm is  $(A_1(A_2 A_3))$ .

However,  $((A_1 A_2) A_3)$  ( $k = 2$ ) is the optimal solution, which takes  $p_0 p_1 p_2 + p_0 p_2 p_3 = 200 + 2000 = 2200$  multiplications. Greedy solution  $(A_1(A_2 A_3))$  ( $k = 1$ ) takes  $p_1 p_2 p_3 + p_0 p_1 p_3 = 20000 + 1000 = 21000$  multiplications.

How can we find the counterexample? We start to try to find a counterexample in a sequence with 3 matrices. This says  $[i, j] = [1, 3]$ , and there are two choices for  $k$ : 1 or 2.

The algorithm perform  $m[1, 3]$  times multiplication.

$$m[1, 3] = \begin{cases} m[1, 1] + m[2, 3] + p_0 p_1 p_3 & k = 1 \\ m[1, 2] + m[3, 3] + p_0 p_2 p_3 & k = 2 \end{cases} = \begin{cases} p_1 p_2 p_3 + p_0 p_1 p_3 & k = 1 \\ p_0 p_1 p_2 + p_0 p_2 p_3 & k = 2 \end{cases}$$

We try to make the greedy algorithm choose  $k = 1$ . This says we want  $p_0 p_1 p_3 < p_0 p_2 p_3 \iff p_1 < p_2$ .

In order to make the greedy approach ( $k = 1$ ) yields a suboptimal solution, we want  $k = 2$  to be the optimal approach. This says we want  $p_1 p_2 p_3 + p_0 p_1 p_3 > p_0 p_1 p_2 + p_0 p_2 p_3$ .

Hence, our goal is to find  $p_0, p_1, p_2, p_3$  such that  $p_1 < p_2$  and  $p_1 p_2 p_3 + p_0 p_1 p_3 > p_0 p_1 p_2 + p_0 p_2 p_3$ . We try to let  $p_1 = 10$  and  $p_2 = 20$ . By a sloppy way, we can try to let  $p_3$  much larger than  $p_0$  since  $p_3$  appears twice and  $p_0$  appears once on the LHS, and  $p_0$  appears twice and  $p_3$  appears once on the RHS. We try to let  $p_0 = 1$  and  $p_3 = 100$ . After testing, we find that this is a good counterexample.

### 15.3-5

If we have limit  $l_i$  on the number of pieces of length  $i$  that we are allowed to produce, We can not find the optimal subproblems indenpendently. We show the optimal-substructure property does not

hold by providing a counterexample. (Recall optimal substructure on page 374)

Consider the following case:

$i$	1	2	3
$p_i$	5	8	9
$l_i i$	2	2	1

The optimal solution of cutting the rod where  $i = 3$  is lengths 1 and 2 with price  $5 + 8 = 13$ . However, the optimal solution of cutting the rod where  $i = 2$  is lengths 1 and 1 with price  $5 + 5 = 10$ . We have showed that there is a way of cutting that does not contain in the optimal solution to the original problem but does contain in the optimal solution to the subproblem, which violates the optimal-substructure property.

### 15.3-6

Note: For this question, we assume that  $r_{ij}r_{ji} = 1$  for any  $1 \leq i, j \leq n$ .

**Claim 1.** If  $c_k = 0$  for all  $k = 1, 2, \dots, n$ , then the problem of finding the best sequence of exchanges from currency 1 to currency  $n$  exhibits optimal substructure.

**Proof.** Let the sequence of currencies  $k_1, k_2, k_3, \dots, k_{n-1}, k_n$  be the best sequence to exchange from currency from  $k_1$  to currency  $k_n$ , which means  $r_{k_1 k_2} r_{k_2 k_3} \dots r_{k_{n-1} k_n}$  is maximized. We show that the sequence of currencies  $k_i, k_{i+1}, \dots, k_{j-1}, k_j$  is the best sequence to exchange from currency from  $k_i$  to currency  $k_j$  by contradiction. Assume  $k_i, q_{i+1}, \dots, q_{j-1}, k_j$  is the best sequence to exchange from currency from  $k_i$  to currency  $k_j$ . By using the “cut-and-paste” technique to replace  $k_i, k_{i+1}, \dots, k_{j-1}, k_j$  with  $k_i, q_{i+1}, \dots, q_{j-1}, k_j$ , we get a better sequence of currencies to exchange from currency from  $k_1$  to currency  $k_n$  (e.g.  $k_1, k_2, k_3, \dots, k_i, q_{i+1}, \dots, q_{j-1}, k_j, \dots, k_{n-1}, k_n$ ), which contradicts to  $k_1, k_2, k_3, \dots, k_{n-1}, k_n$  is the best sequence.  $\square$

Now, we show that if  $c_k$  are arbitrary values, then the problem of finding the best sequence of exchanges from currency 1 to currency  $n$  does not necessarily exhibit optimal substructure.

Consider the following case:

We want to exchange from currency from  $k_1$  to currency  $k_4$ .

$r_{k_1 k_2}$	$r_{k_2 k_3}$	$r_{k_3 k_4}$	$r_{k_1 k_3}$	$r_{k_2 k_4}$	$r_{k_1 k_4}$	$c_1$	$c_2$	$c_3$
6	2	5	2	8	10	4	5	20

We try to find the optimal sequence from  $k_1$  to  $k_4$  by list all possible sequences:

$$\begin{aligned}
 k_1 k_4: & r_{k_1 k_4} - c_1 = 10 - 4 = 6 \\
 k_1 k_2 k_4: & r_{k_1 k_2} r_{k_2 k_4} - c_2 = 6 \cdot 8 - 5 = 43 \\
 k_1 k_3 k_4: & r_{k_1 k_3} r_{k_3 k_4} - c_2 = 2 \cdot 5 - 5 = 5 \\
 k_1 k_2 k_3 k_4: & r_{k_1 k_2} r_{k_2 k_3} r_{k_3 k_4} - c_3 = 6 \cdot 2 \cdot 5 - 20 = 40 \\
 k_1 k_3 k_2 k_4: & r_{k_1 k_3} r_{k_3 k_2} r_{k_2 k_4} - c_3 = 2 \cdot \frac{1}{2} \cdot 8 - 20 = -12
 \end{aligned}$$

The optimal sequence from  $k_1$  to  $k_4$  is  $k_1 k_2 k_4$ .

Now, we try to show that  $k_2 k_4$  is not the optimal sequence from  $k_2$  to  $k_4$  by list all possible sequences:

$$k_2 k_4: r_{k_2 k_4} - c_1 = 8 - 4 = 4$$



$k_2k_3k_4$ :  $r_{k_2k_3}r_{k_3k_4} - c_2 = 2 \cdot 5 - 5 = 5$

$k_2k_1k_4$ : (unnecessary)

$k_2k_1k_3k_4$ : (unnecessary)

$k_2k_3k_1k_4$ : (unnecessary)

It is unnecessary to solve results for  $k_2k_1k_4$ ,  $k_2k_1k_3k_4$ , and  $k_2k_3k_1k_4$  since we already find that  $k_2k_3k_4$  is a better sequence than  $k_4$  is  $k_2k_4$ .

## 15.4

### 15.4-1

$\langle 1, 0, 0, 1, 1, 0 \rangle$

(the solution is not unique)

### 15.4-2

```
1  template <typename T>
2  std::list<T> OutputLCS(LengthTable length_table,
3      const std::vector<T>& sequence_x, const std::vector<T>& sequence_y)
4  {
5      size_t it_x, it_y;
6      std::list<T> common_sequence;
7      it_x = sequence_x.size();
8      it_y = sequence_y.size();
9      while (it_x > 0 && it_y > 0)
10     {
11         const T& element_x = sequence_x[it_x - 1];
12         const T& element_y = sequence_y[it_y - 1];
13         if (element_x == element_y)
14         {
15             common_sequence.push_front(element_y);
16             --it_x;
17             --it_y;
18         }
19         else if (length_table[it_x - 1][it_y] >= length_table[it_x][it_y - 1])
20         {
21             --it_x;
22         }
23         else
24         {
25             --it_y;
```

```
26         }
27     }
28     return common_sequence;
29 }
```

### 15.4-3

```
1     template <typename T>
2     int RecursiveLCSLengthMemoized
3         (const std::vector<T>& sequence_x, const std::vector<T>& sequence_y,
4          size_t it_x, size_t it_y,
5          LengthTable& length_table, PointerTable& pointer_table)
6     {
7         int sequence_length_x_prev, sequence_length_y_prev;
8         if (length_table[it_x][it_y] >= 0)
9             return length_table[it_x][it_y];
10        if (sequence_x[it_x - 1] == sequence_y[it_y - 1])
11        {
12            length_table[it_x][it_y] = RecursiveLCSLengthMemoized(sequence_x, sequence_y,
13                it_x - 1, it_y - 1, length_table, pointer_table) + 1;
14            pointer_table[it_x][it_y] = PointerType::X_Y_PREV;
15        }
16        else
17        {
18            sequence_length_x_prev = RecursiveLCSLengthMemoized(sequence_x, sequence_y,
19                it_x - 1, it_y, length_table, pointer_table);
20            sequence_length_y_prev = RecursiveLCSLengthMemoized(sequence_x, sequence_y,
21                it_x, it_y - 1, length_table, pointer_table);
22            if (sequence_length_x_prev >= sequence_length_y_prev)
23            {
24                length_table[it_x][it_y] = sequence_length_x_prev;
25                pointer_table[it_x][it_y] = PointerType::X_PREV;
26            }
27            else
28            {
29                length_table[it_x][it_y] = sequence_length_y_prev;
30                pointer_table[it_x][it_y] = PointerType::Y_PREV;
31            }
32        }
33        return length_table[it_x][it_y];
}
```

```
34     }
35
36     template <typename T>
37     std::pair<LengthTable, PointerTable> LCSLengthMemoized
38         (const std::vector<T>& sequence_x, const std::vector<T>& sequence_y)
39     {
40         size_t size_x, size_y, it_x, it_y,
41             sequence_length_x_prev, sequence_length_y_prev;
42         size_x = sequence_x.size();
43         size_y = sequence_y.size();
44         // note this line init all elements in length_table with -1
45         LengthTable length_table(size_x + 1, LengthTableRow(size_y + 1, -1));
46         PointerTable pointer_table(size_x + 1, PointerTableRow(size_y + 1, NIL));
47         for (it_x = 0; it_x <= size_x; ++it_x)
48             length_table[it_x][0] = 0;
49         for (it_y = 1; it_y <= size_y; ++it_y)
50             length_table[0][it_y] = 0;
51         RecursiveLCSLengthMemoized(sequence_x, sequence_y, size_x, size_y,
52             length_table, pointer_table);
53         return std::make_pair(std::move(length_table), std::move(pointer_table));
54     }
```

#### 15.4-4

Solution for space of  $2 \cdot \min(m, n) + O(1)$ :

```
1     template <typename T>
2     std::pair<int, PointerTable> LCSLengthByTwoRowLengthTable
3         (const std::vector<T>& sequence_x, const std::vector<T>& sequence_y)
4     {
5         size_t size_x, size_y, it_x, it_y;
6         int sequence_length_x_prev, sequence_length_y_prev;
7         size_x = sequence_x.size();
8         size_y = sequence_y.size();
9         if (size_x < size_y)
10             return LCSLengthByTwoRowLengthTable(sequence_y, sequence_x);
11         LengthTableRow length_table_row_1(size_y + 1, length_table_row_2(size_y + 1));
12         PointerTable pointer_table(size_x + 1, PointerTableRow(size_y + 1));
13         for (it_y = 0; it_y <= size_y; ++it_y)
14             length_table_row_1[it_y] = 0;
15         for (it_x = 1; it_x <= size_x; ++it_x)
```

```
16     {
17         for (it_y = 1; it_y <= size_y; ++it_y)
18         {
19             if (sequence_x[it_x - 1] == sequence_y[it_y - 1])
20             {
21                 length_table_row_2[it_y] =
22                     length_table_row_1[it_y - 1] + 1;
23                 pointer_table[it_x][it_y] = PointerType::X_Y_PREV;
24             }
25             else
26             {
27                 sequence_length_x_prev = length_table_row_1[it_y];
28                 sequence_length_y_prev = length_table_row_2[it_y - 1];
29                 if (sequence_length_x_prev >= sequence_length_y_prev)
30                 {
31                     length_table_row_2[it_y] = sequence_length_x_prev;
32                     pointer_table[it_x][it_y] = PointerType::X_PREV;
33                 }
34                 else
35                 {
36                     length_table_row_2[it_y] = sequence_length_y_prev;
37                     pointer_table[it_x][it_y] = PointerType::Y_PREV;
38                 }
39             }
40         }
41         length_table_row_1.swap(length_table_row_2);
42     }
43     return std::make_pair(length_table_row_2[size_y], std::move(pointer_table));
44 }
```

Solution for space of  $\min(m, n) + O(1)$ :

```
1     template <typename T>
2     std::pair<int, PointerTable> LCSLengthByOneRowLengthTable
3         (const std::vector<T>& sequence_x, const std::vector<T>& sequence_y)
4     {
5         size_t size_x, size_y, it_x, it_y;
6         int sequence_length_x_y_prev, sequence_length_x_y_now;
7         size_x = sequence_x.size();
8         size_y = sequence_y.size();
9         if (size_x < size_y)
```

```
10     return LCSLengthByOneRowLengthTable(sequence_y, sequence_x);
11     LengthTableRow length_table_row(size_y + 1);
12     PointerTable pointer_table(size_x + 1, PointerTableRow(size_y + 1));
13     for (it_y = 0; it_y <= size_y; ++it_y)
14         length_table_row[it_y] = 0;
15     for (it_x = 1; it_x <= size_x; ++it_x)
16     {
17         sequence_length_x_y_prev = 0;
18         for (it_y = 1; it_y <= size_y; ++it_y)
19         {
20             sequence_length_x_y_now = length_table_row[it_y];
21             if (sequence_x[it_x - 1] == sequence_y[it_y - 1])
22             {
23                 length_table_row[it_y] =
24                     sequence_length_x_y_prev + 1;
25                 pointer_table[it_x][it_y] = PointerType::X_Y_PREV;
26             }
27             else
28             {
29                 if (length_table_row[it_y] >= length_table_row[it_y - 1])
30                 {
31                     // length_table_row[it_y] = length_table_row[it_y];
32                     pointer_table[it_x][it_y] = PointerType::X_PREV;
33                 }
34                 else
35                 {
36                     length_table_row[it_y] = length_table_row[it_y - 1];
37                     pointer_table[it_x][it_y] = PointerType::Y_PREV;
38                 }
39             }
40             sequence_length_x_y_prev = sequence_length_x_y_now;
41         }
42     }
43     return std::make_pair(length_table_row[size_y], std::move(pointer_table));
44 }
```

## 15.4-5

```
1 // O(n^2)
2 std::vector<int> LongestIncreasingSubsequence
```

```
3         (const std::vector<int>& sequence)
4     {
5         size_t i, j, size, sequence_length_it, max_length_index;
6         size = sequence.size();
7         std::vector<size_t> length(size), prev(size);
8         // compute
9         max_length_index = 0;
10        for (i = 0; i < size; ++i)
11        {
12            length[i] = 1;
13            for (j = 0; j < i; ++j)
14            {
15                if (sequence[j] < sequence[i])
16                {
17                    sequence_length_it = length[j] + 1;
18                    if (sequence_length_it > length[i])
19                    {
20                        length[i] = sequence_length_it;
21                        prev[i] = j;
22                    }
23                }
24            }
25            if (length[i] > length[max_length_index])
26                max_length_index = i;
27        }
28        // output
29        std::vector<int> result(length[max_length_index]);
30        for (i = length[max_length_index] - 1; i < length[max_length_index]; --i)
31        {
32            result[i] = sequence[max_length_index];
33            max_length_index = prev[max_length_index];
34        }
35        return result;
36    }
```

We also can sort the sequence and find the longest common sequence between the sequence and the sorted sequence.

## 15.4-6

```
1      // O(nlgn)
2      std::vector<int> LongestIncreasingSubsequenceBinarySearch
3          (const std::vector<int>& sequence)
4      {
5          size_t i, size, seq_index;
6          int lower, upper, middle;
7          size = sequence.size();
8          /**
9           * prev
10          * index is the element index of sequence
11          * value is the element index of sequence
12          *      such that the element is the prev element
13          *      in the increasing subsequence
14          * -----
15          * sub_seq
16          * index is length of the increasing subsequence
17          * value is the element index of sequence
18          */
19          std::vector<size_t> prev(size), sub_seq;
20          sub_seq.reserve(size);
21          // compute
22          for (i = 0; i < size; ++i)
23          {
24              lower = 0;
25              upper = sub_seq.size() - 1;
26              while (lower <= upper)
27              {
28                  middle = lower + ((upper - lower) >> 1);
29                  if (sequence[sub_seq[middle]] > sequence[i])
30                  {
31                      upper = middle - 1;
32                  }
33                  else if (sequence[sub_seq[middle]] < sequence[i])
34                  {
35                      lower = middle + 1;
36                  }
37                  else
38                  {
```

```
39         lower = middle;
40         break;
41     }
42 }
43 // sequence[i] <= sequence[sub_seq[lower]] must be true
44 if (lower >= sub_seq.size())
45     sub_seq.push_back(i);
46 else
47     sub_seq[lower] = i;
48 if (lower > 0)
49     prev[i] = sub_seq[lower - 1];
50 }
51 // output
52 size = sub_seq.size();
53 std::vector<int> result(size);
54 seq_index = sub_seq[size - 1];
55 for (i = size - 1; i < size; --i)
56 {
57     result[i] = sequence[seq_index];
58     seq_index = prev[seq_index];
59 }
60 return result;
61 }
```

## 15.5

### 15.5-1

```
1 void ConstructOptimalBST(std::ostream& os, const Table& root,
2     const std::string& suffix, size_t i, size_t j)
3 {
4     if (j == i - 1)
5     {
6         os << "d_" << j << " is the " << suffix << "\n";
7     }
8     else
9     {
10         size_t r = root[i][j];
11         os << "k_" << r << " is the " << suffix << "\n";
12         ConstructOptimalBST(os, root,
```



```
13         "left child of k_" + std::to_string(r), i, r - 1);
14         ConstructOptimalBST(os, root,
15         "right child of k_" + std::to_string(r), r + 1, j);
16     }
17 }
18
19 void ConstructOptimalBST(std::ostream& os, const Table& root)
20 {
21     ConstructOptimalBST(os, root, "root", 1, root.size() - 1);
22 }
```

## 15.5-2

cost: 3.12

structure:

$k_5$  is the root

$k_2$  is the left child of  $k_5$

$k_1$  is the left child of  $k_2$

$d_0$  is the left child of  $k_1$

$d_1$  is the right child of  $k_1$

$k_3$  is the right child of  $k_2$

$d_2$  is the left child of  $k_3$

$k_4$  is the right child of  $k_3$

$d_3$  is the left child of  $k_4$

$d_4$  is the right child of  $k_4$

$k_7$  is the right child of  $k_5$

$k_6$  is the left child of  $k_7$

$d_5$  is the left child of  $k_6$

$d_6$  is the right child of  $k_6$

$d_7$  is the right child of  $k_7$

## 15.5-3

The OPTIMAL-BST would still take  $\Theta(n^3)$ .

According to equation (15.12), it takes  $\Theta(j - i)$  to compute  $w(i, j)$  each time. Hence the total time contributed by line 9 in OPTIMAL-BST is

$$\begin{aligned}\sum_{l=1}^n \sum_{i=1}^{n-l+1} (c \cdot (j-i)) &= c \cdot \sum_{l=1}^n \sum_{i=1}^{n-l+1} ((i+l-1) - i) \\ &= c \cdot \sum_{l=1}^n (n - (l-1))(l-1) \\ &= c \cdot \sum_{z=0}^{n-1} (n-z)z \\ &= c \cdot \sum_{z=0}^{n-1} (n-z)z \\ &= \Theta(n^3)\end{aligned}$$

## 15.5-4

```
1      // O(n^2)
2      std::pair<Table, Table> OptimalBST(const Row& p, const Row& q)
3      {
4          size_t q_size, i, j, l, r;
5          int t;
6          q_size = q.size(); // n + 1
7          Table e(q_size + 1, Row(q_size)), w(q_size + 1, Row(q_size)),
8              root(q_size, Row(q_size));
9          for (i = 1; i <= q_size; ++i) // 1 to (n + 1)
10             {
11                 e[i][i - 1] = q[i - 1];
12                 // w[i][i - 1] = q[i - 1];
13             }
14         // l = 1
15         for (i = 1; i < q_size; ++i) // 1 to n
16             {
17                 // w[i][i] = w[i][i - 1] + p[i] + q[i];
18                 w[i][i] = q[i - 1] + p[i] + q[i];
19                 // e[i][i] = e[i][i - 1] + e[i + 1][i] + w[i][i];
20                 e[i][i] = q[i - 1] + q[i] + w[i][i];
21                 root[i][i] = i;
22             }
23         // l = 2 ... n
24         for (l = 2; l < q_size; ++l) // 2 to n
25             {
26                 for (i = 1; i <= q_size - l; ++i) // 1 to (n - l + 1)
27                     {
```

```
28         j = i + 1 - 1;
29         e[i][j] = INT_MAX;
30         // root[i][j - 1] to root[i + 1][j]
31         for (r = root[i][j - 1]; r <= root[i + 1][j]; ++r)
32         {
33             t = e[i][r - 1] + e[r + 1][j];
34             if (t < e[i][j])
35             {
36                 e[i][j] = t;
37                 root[i][j] = r;
38             }
39         }
40         w[i][j] = w[i][j - 1] + p[j] + q[j];
41         e[i][j] += w[i][j];
42     }
43 }
44 return std::make_pair(std::move(e), std::move(root));
45 }
```

The main principle is to reduce the loop range of line 10 of OPTIMAL-BST from  $i...j$  to  $root[i, j - 1]...root[i + 1, j]$ . This says modify line 10 of OPTIMAL-BST to

**for**  $r = root[i, j - 1]$  **to**  $root[i + 1, j]$

Note the modification of case of  $l = 1$  is skipped here. We primary consider  $l \geq 2$ .

**Claim 2.** OPTIMAL-BST run in  $\Theta(n^2)$  time

**Proof.** Denote the **for** loop of line 5-14 as  $L_1$ , and denote the **for** loop of line 6-14 as  $L_2$ . In order to show OPTIMAL-BST run in  $\Theta(n^2)$  time, we just need to show each iteration of  $L_1$  takes  $\Theta(n)$  time.

Let  $t(l, i)$  be the running time of each iteration of  $L_2$  with variable  $l$  and  $i$ .

$$\begin{aligned} t(l, i) &= r[i + 1, j] - r[i, j - 1] + 1 \\ &= r[i + 1, i + l - 1] - r[i, i + l - 2] + 1 \end{aligned}$$

Each iteration of  $L_1$  takes  $\sum_{i=1}^{n-l+1} t(l, i)$  time. Notice  $1 \leq root[a, b] \leq n$  for all  $a, b$ . Also, by Knuth [212],  $root[a + k, b - 1 + k] \leq root[a + h, b - 1 + h]$  for all  $k \leq h$ . Hence we have  $root[a + h, b - 1 + h] - root[a + k, b - 1 + k] \leq n$  for all  $k \leq h$ .

$$\begin{aligned}\sum_{l=1}^n \sum_{i=1}^{n-l+1} t(l, i) &= \sum_{l=1}^n \sum_{i=1}^{n-l+1} (r[i+1, i+l-1] - r[i, i+l-2] + 1) \\ &= \sum_{l=1}^n (r[n-l+2, n] - r[1, l-1] + (n-l+1))\end{aligned}$$

We claim  $r[n-l+2, n] - r[1, l-1] \leq n$  by setting  $a = 1$ ,  $b = l$ ,  $k = 0$ , and  $h = n-l+1$ . Hence  $\sum_{l=1}^n \sum_{i=1}^{n-l+1} t(l, i) = \Theta(n^2)$ . □

## Chapter 15 Problems

### 15-1

The memoized, recursive version algorithm run in  $O(V + E)$ .

```
1      // O(E)
2      int RecursiveLongestSimplePath(const Graph& graph, int start, int dist,
3          Row& length, Row& next)
4      {
5          int curr_length;
6          if (length[start] >= 0)
7              return length[start];
8          for (const AdjListElement& adj_element : graph[start])
9              {
10                 curr_length = adj_element.weight + RecursiveLongestSimplePath(graph,
11                     adj_element.vertex, dist, length, next);
12                 if (curr_length > length[start])
13                     {
14                         length[start] = curr_length;
15                         next[start] = adj_element.vertex;
16                     }
17             }
18          return length[start];
19      }
20
21      // O(V + E)
22      std::list<int> LongestSimplePath(const Graph& graph, int start, int dist)
23      {
24          int vertex;
25          size_t size;
```

```
26     std::list<int> result;
27     size = graph.size();
28     Row length(size, INT_MIN), next(size);
29     length[dist] = 0;
30     RecursiveLongestSimplePath(graph, start, dist, length, next);
31     if (length[start] < 0)
32         return result;
33     vertex = start;
34     while (vertex != dist)
35     {
36         result.push_back(vertex);
37         vertex = next[vertex];
38     }
39     result.push_back(dist);
40     return result;
41 }
```

We can implement a bottom-up version algorithm by utilizing the topological sort.

## 15-2

Denote  $X_{ij} = \langle x_i, x_{i+1}, \dots, x_{j-1}, x_j \rangle$  and  $Z_{kt} = \langle z_k, z_{k+1}, \dots, z_{t-1}, z_t \rangle$ . Let  $Z_{kt}$  be the LPS of  $X_{ij}$ . For all  $X_{ij}$  where  $n > 2$ , we claim the following:

1. If  $x_i = x_j$ , then  $z_k = z_t = x_i = x_j$  and  $Z_{k+1,t-1}$  is the LPS of  $X_{i+1,j-1}$ .
2. If  $x_i \leq x_j$ , then  $z_k = z_t \neq x_i$  implies  $Z_{kt}$  is the LPS of  $X_{i+1,j}$ .
3. If  $x_i \leq x_j$ , then  $z_k = z_t \neq x_j$  implies  $Z_{kt}$  is the LPS of  $X_{i,j-1}$ .

Denote  $c[i, j]$  as the length of LPS of  $X_{ij}$ . For all  $X_{ij}$  where  $n > 2$ , we have the following recursive solution:

$$c[i, j] = \begin{cases} c[i+1, j-1] + 2 & \text{if } x_i = x_j. \\ \max(c[i+1, j], c[i, j-1]) & \text{if } x_i \neq x_j. \end{cases}$$

Therefore, we have the following bottom-up algorithm which runs in  $\Theta(n^2)$ :

```
1     // O(n^2)
2     std::string LPS(const std::string& str)
3     {
4         size_t size, i, j, lps_left, lps_right;
5         size = str.size();
6         std::string result;
7         LengthTable lengths(size, LengthTableRow(size, INT_MIN));
8         PointerTable pointers(size, PointerTableRow(size, NIL));
9         // compute
```

```
10     for (i = 0; i < size - 1; ++i)
11     {
12         lengths[i][i] = 1;
13         j = i + 1;
14         if (str[i] == str[j])
15         {
16             lengths[i][j] = 2;
17             pointers[i][j] = COMMON;
18         }
19         else
20         {
21             lengths[i][j] = 1;
22             pointers[i][j] = I_SUCC;
23         }
24     }
25     lengths[i][i] = 1; // i == size - 1
26     for (i = size - 3; i < size; --i)
27     {
28         for (j = i + 2; j < size; ++j)
29         {
30             if (str[i] == str[j])
31             {
32                 lengths[i][j] = lengths[i + 1][j - 1] + 2;
33                 pointers[i][j] = COMMON;
34             }
35             else if (lengths[i + 1][j] >= lengths[i][j - 1])
36             {
37                 lengths[i][j] = lengths[i + 1][j];
38                 pointers[i][j] = I_SUCC;
39             }
40             else
41             {
42                 lengths[i][j] = lengths[i][j - 1];
43                 pointers[i][j] = J_PREV;
44             }
45         }
46     }
47     // output
48     i = 0;
49     j = size - 1;
```

```
50     lps_left = 0;
51     lps_right = lengths[i][j] - 1;
52     result.resize(lengths[i][j]);
53     while (i < j)
54     {
55         if (pointers[i][j] == I_SUCC)
56         {
57             ++i;
58         }
59         else if (pointers[i][j] == J_PREV)
60         {
61             --j;
62         }
63         else
64         {
65             result[lps_left] = str[i];
66             result[lps_right] = str[j];
67             ++i;
68             --j;
69             ++lps_left;
70             --lps_right;
71         }
72     }
73     if (i == j)
74     {
75         result[lps_left] = str[i];
76     }
77     return result;
78 }
```

### 15-3

Sort points by x-coordinate first, and put the sorted points in a sequence  $\langle p_1, p_2, \dots, p_n \rangle$ . Let  $d(i, j)$  be the distance between  $p_i$  and  $p_j$ . Denote  $c[i, j]$  be the shortest length of bitonic path from  $p_i$  to  $p_j$ . That is from  $p_i$  go left to  $p_1$  then go right to  $p_j$ . We have the the optimal substructure and have the following preliminary recursive solution

$$c[i, j] = \min(c[a, b] + d(a, i) + d(b, j))$$

where  $1 \leq a \leq i$  and  $1 \leq b \leq j$ . Note we have to travel all the points since we do not want to skip a point, but it is not covered by the above recursive solution, and we want to add restrictions to  $a$

and  $b$ . Our goal is to determine subproblems and restrict choices of subproblems.

Consider  $i < j$  only.  $p_i$  and  $p_j$  does not connect for all  $i > 1$  or  $j > 2$ . When  $i < j - 1$ , we must connect  $p_{j-1}$  and  $p_j$ . When  $i = j - 1$ , all points  $p_b$  where  $b < j - 1$  can connect to  $p_j$ . Hence we have the following complete recursive solution:

$$c[i, j] = \begin{cases} d(1, 2) & \text{if } i = 1 \text{ and } j = 2. \\ c[i, j - 1] + d(j - 1, j) & \text{if } i < j - 1. \\ \min_{1 \leq b < j-1} (c[b, j - 1] + d(b, j)) = \min_{1 \leq b < i} (c[b, i] + d(b, j)) & \text{if } i = j - 1. \end{cases}$$

Therefore, we have the following bottom-up algorithm which runs in  $O(n^2)$ :

```
1      std::vector<size_t> BitonicTour(std::vector<Point>& points)
2      {
3          int l_length;
4          size_t size, i, j, b, result_left_p, result_right_p;
5          size = points.size();
6          Table t_length(size, Row(size, INT_MAX)),
7              t_left_neighbour(size, Row(size));
8          // r_neighbour_distance[j] == Distance(points[j - 1], points[j])
9          Row r_neighbour_distance(size);
10         std::vector<size_t> result(size);
11         // sort points by x-coordinate
12         QuickSort(points, 0, (int)size - 1);
13         // compute
14         for (j = 2; j < size - 1; ++j)
15         {
16             r_neighbour_distance[j] = Distance(points[j - 1], points[j]);
17         }
18         t_length[0][1] = Distance(points[0], points[1]);
19         t_left_neighbour[0][1] = 0;
20         for (i = 0; i < size - 1; ++i)
21         {
22             j = i + 1;
23             for (b = 0; b < i; ++b)
24             {
25                 l_length = t_length[b][i] + Distance(points[b], points[j]);
26                 if (l_length < t_length[i][j])
27                 {
28                     t_length[i][j] = l_length;
29                     t_left_neighbour[i][j] = b;
30                 }
31             }
32         }
33     }
```



```
32         for (j = i + 2; j < size - 1; ++j)
33         {
34             t_length[i][j] = t_length[i][j - 1] + r_neighbour_distance[j];
35             t_left_neighbour[i][j] = j - 1;
36         }
37     }
38     // output
39     result_left_p = 0;
40     result_right_p = size - 1;
41     i = size - 2;
42     j = size - 1;
43     while (i != j)
44     {
45         if (i < j)
46         {
47             result[result_left_p] = j;
48             ++result_left_p;
49             j = t_left_neighbour[i][j];
50         }
51         else
52         {
53             result[result_right_p] = i;
54             --result_right_p;
55             i = t_left_neighbour[j][i];
56         }
57     }
58     result[result_left_p] = 0;
59     return result;
60 }
```

## 15-4

Denote  $c[i, j]$  as the minimum sum of extra space (characters). For  $j < n$ , we have the following preliminary recursive solution:

$$c[i, j] = \min_{i < k \leq j} (c[i, k - 1] + c[k, j])$$

Let  $e(i, j) = M - j + i - \sum_{k=1}^j l_k$ . Fix  $j$ , and let  $t$  be the smallest possible integer such that  $e(t, j) \geq 0$ . For  $j < n$ , we have the following improved recursive solution:

$$c[1, j] = \begin{cases} e(1, j) & \text{if } t = 1. \\ \min_{t \leq k \leq j} (c[1, k - 1] + e(k, j)) & \text{if } t \geq 2. \end{cases}$$

Therefore, we have the following bottom-up algorithm which runs in  $O(n^2)$ :

```
1      std::list<int> PrintingNeatly(int line_max_char,
2          const std::vector<int>& lengths)
3      {
4          int size, j, k, extra;
5          size = (int)(lengths.size());
6          /**
7           * min_sum[k + 1] is the minimum extra char
8           * from the first word to the word which the length is lengths[k]
9           */
10         std::vector<int> min_sum(size + 1);
11         std::vector<int> line_start_index(size);
12         std::list<int> result;
13         // compute
14         min_sum[0] = 0;
15         for (j = 0; j < size - 1; ++j)
16         {
17             min_sum[j + 1] = INT_MAX;
18             extra = line_max_char + 1;
19             for (k = j; k >= 0; --k)
20             {
21                 extra = extra - 1 - lengths[k];
22                 if (extra < 0)
23                     break;
24                 if (min_sum[k] + extra < min_sum[j + 1])
25                 {
26                     min_sum[j + 1] = min_sum[k] + extra;
27                     line_start_index[j] = k;
28                 }
29             }
30         }
31         extra = line_max_char + 1;
32         min_sum[size] = INT_MAX;
33         for (k = size - 1; k >= 0; --k)
34         {
35             extra = extra - 1 - lengths[k];
```

```
36         if (extra < 0)
37             break;
38         if (min_sum[k] < min_sum[size])
39         {
40             min_sum[size] = min_sum[k];
41             line_start_index[size - 1] = k;
42         }
43     }
44     // output
45     k = size;
46     while (k != 0)
47     {
48         k = line_start_index[k - 1];
49         result.push_front(k);
50     }
51     return result;
52 }
```

## 15-5

(a)

Denote  $c[i, j]$  as edit distance from  $x[1 \cdots i]$  to  $y[1 \cdots j]$ . We have the following preliminary recursive solution

$$c[i, j] = \min_{\substack{1 \leq k < i \\ 1 \leq t < j}} (c[k, t] + \alpha)$$

where there is a single operation from  $x[k], y[t]$  to  $x[i], y[j]$  which cost  $\alpha$ . After adding details, we have the following improved recursive solution:

$$c[i, j] = \min \begin{cases} c[i-2, j-2] + \text{cost}(\text{twiddle}) & \text{if } x[i-1] = y[j] \text{ and } x[i] = y[j-1] \\ c[i-1, j-1] + \text{cost}(\text{copy}) & \text{if } x[i] = y[j] \\ c[i-1, j-1] + \text{cost}(\text{replace}) & \text{if } x[i] \neq y[j] \\ c[i, j-1] + \text{cost}(\text{insert}) \\ c[i-1, j] + \text{cost}(\text{delete}) \\ c[k, j]_{0 \leq k < i} + \text{cost}(\text{kill}) & \text{if } i = m \text{ and } j = n \end{cases}$$

Therefore, we have the following bottom-up algorithm which runs in  $\Theta(mn)$ :

```
1  enum Operation { COPY, REPLACE, DELETE, INSERT, TWIDDLE, KILL };
2
3  // i and j are index of edit_distance and operations,
```

```
4  // instead of source and target
5  inline void CheckCopyReplaceInsertDelete(const std::string& source,
6      const std::string& target, const std::vector<int>& cost,
7      std::vector< std::vector<int> >& edit_distance,
8      std::vector< std::vector<Operation> >& operations,
9      int i, int j)
10 {
11     int l_cost;
12     if (source[i - 1] == target[j - 1])
13     {
14         l_cost = edit_distance[i - 1][j - 1] + cost[COPY];
15         if (l_cost < edit_distance[i][j])
16         {
17             edit_distance[i][j] = l_cost;
18             operations[i][j] = COPY;
19         }
20     }
21     if (source[i - 1] != target[j - 1])
22     {
23         l_cost = edit_distance[i - 1][j - 1] + cost[REPLACE];
24         if (l_cost < edit_distance[i][j])
25         {
26             edit_distance[i][j] = l_cost;
27             operations[i][j] = REPLACE;
28         }
29     }
30     l_cost = edit_distance[i][j - 1] + cost[INSERT];
31     if (l_cost < edit_distance[i][j])
32     {
33         edit_distance[i][j] = l_cost;
34         operations[i][j] = INSERT;
35     }
36     l_cost = edit_distance[i - 1][j] + cost[DELETE];
37     if (l_cost < edit_distance[i][j])
38     {
39         edit_distance[i][j] = l_cost;
40         operations[i][j] = DELETE;
41     }
42 }
43
```

```
44 // O(mn)
45 std::list<Operation> EditDistance(const std::string& source,
46     const std::string& target, const std::vector<int>& cost)
47 {
48     int m, n, i, j, kill_i_pos;
49     int l_cost;
50     m = (int)(source.size());
51     n = (int)(target.size());
52     std::list<Operation> result;
53     /**
54      * edit_distance[i + 1][j + 1] is the edit distance
55      * from source[0 ... i] to target[0 ... j]
56      */
57     std::vector< std::vector<int> > edit_distance(m + 1,
58         std::vector<int>(n + 1));
59     /**
60      * operations[i + 1][j + 1] is the operation
61      * which incremented offset of source and target to i and j
62      */
63     std::vector< std::vector<Operation> > operations(m + 1,
64         std::vector<Operation>(n + 1));
65     // compute
66     edit_distance[0][0] = 0;
67     for (i = 1; i <= m; ++i) // j = 0
68     {
69         edit_distance[i][0] = edit_distance[i - 1][0] + cost[DELETE];
70         operations[i][0] = DELETE;
71     }
72     for (j = 1; j <= n; ++j) // i = 0
73     {
74         edit_distance[0][j] = edit_distance[0][j - 1] + cost[INSERT];
75         operations[0][j] = INSERT;
76     }
77     for (i = 1; i <= m; ++i) // j = 1
78     {
79         edit_distance[i][1] = INT_MAX;
80         CheckCopyReplaceInsertDelete(source, target, cost,
81             edit_distance, operations, i, 1);
82     }
83     for (j = 2; j <= n; ++j) // i = 1
```

```
84     {
85         edit_distance[1][j] = INT_MAX;
86         CheckCopyReplaceInsertDelete(source, target, cost,
87             edit_distance, operations, 1, j);
88     }
89     for (i = 2; i <= m; ++i)
90     {
91         for (j = 2; j <= n; ++j)
92         {
93             edit_distance[i][j] = INT_MAX;
94             CheckCopyReplaceInsertDelete(source, target, cost,
95                 edit_distance, operations, i, j);
96             if (source[i - 2] == target[j - 1] &&
97                 source[i - 1] == target[j - 2])
98             {
99                 l_cost = edit_distance[i - 2][j - 2] + cost[TWIDDLE];
100                 if (l_cost < edit_distance[i][j])
101                 {
102                     edit_distance[i][j] = l_cost;
103                     operations[i][j] = TWIDDLE;
104                 }
105             }
106         }
107     }
108     for (i = 0; i < m; ++i)
109     {
110         l_cost = edit_distance[i][n] + cost[KILL];
111         if (l_cost < edit_distance[m][n])
112         {
113             edit_distance[m][n] = l_cost;
114             operations[m][n] = KILL;
115             kill_i_pos = i;
116         }
117     }
118     // output
119     i = m;
120     j = n;
121     while (i != 0 || j != 0)
122     {
123         result.push_front(operations[i][j]);
```

```
124     switch (operations[i][j])
125     {
126     case COPY:
127         --i;
128         --j;
129         break;
130     case REPLACE:
131         --i;
132         --j;
133         break;
134     case DELETE:
135         --i;
136         break;
137     case INSERT:
138         --j;
139         break;
140     case TWIDDLE:
141         i -= 2;
142         j -= 2;
143         break;
144     case KILL:
145         i = kill_i_pos;
146         break;
147     }
148 }
149 return result;
150 }
```

(b)

Disable twiddle and delete operations, and let the remaining operations have the following costs:

$\text{cost}(\text{copy}) = -1$   
 $\text{cost}(\text{replace}) = +1$   
 $\text{cost}(\text{delete}) = +2$   
 $\text{cost}(\text{insert}) = +2$

## 15-6

Let  $r(x)$  be the conviviality rating of  $x$ . Denote  $c[x]$  as the sum of the conviviality ratings of the subtree root at  $x$  if  $x$  will attend. Denote  $d[x]$  as the sum of the conviviality ratings of the subtree root at  $x$  if  $x$  will not attend. We have the following recursive solution:

$$c[x] = r(x) + \sum_{y \text{ is child of } x} d[y]$$

$$d[x] = \sum_{y \text{ is child of } x} \max(c[y], d[y])$$

Therefore, we have the following algorithm which runs in  $\Theta(n)$ :

```
1  /* declarations of function are omitted */
2
3  struct Node
4  {
5      Node *left_child = nullptr;
6      Node *right_sibling = nullptr;
7      std::string name;
8      int conviviality_rating;
9      int dp_attend_subtree_r_sum = INT_MIN;
10     int dp_not_attend_subtree_r_sum = INT_MIN;
11     bool dp_attend_if_parent_not_attend;
12 };
13
14 void DpAuxRootAttend(Node *root)
15 {
16     Node *child;
17     root->dp_attend_subtree_r_sum = root->conviviality_rating;
18     for (child = root->left_child; child; child = child->right_sibling)
19     {
20         if (child->dp_not_attend_subtree_r_sum < 0)
21             DpAuxRootNotAttend(child);
22         root->dp_attend_subtree_r_sum += child->dp_not_attend_subtree_r_sum;
23     }
24 }
25
26 void DpAuxRootNotAttend(Node *root)
27 {
28     Node *child;
29     root->dp_not_attend_subtree_r_sum = 0;
30     for (child = root->left_child; child; child = child->right_sibling)
31     {
32         if (child->dp_attend_subtree_r_sum < 0)
33             DpAuxRootAttend(child);
```



```
34     if (child->dp_not_attend_subtree_r_sum < 0)
35         DpAuxRootNotAttend(child);
36     if (child->dp_attend_subtree_r_sum > child->dp_not_attend_subtree_r_sum)
37     {
38         root->dp_not_attend_subtree_r_sum += child->dp_attend_subtree_r_sum;
39         child->dp_attend_if_parent_not_attend = true;
40     }
41     else
42     {
43         root->dp_not_attend_subtree_r_sum += child->dp_not_attend_subtree_r_sum;
44         child->dp_attend_if_parent_not_attend = false;
45     }
46 }
47 }
48
49 void OutputListRootAttend(Node *root, std::list<Node*>& result)
50 {
51     Node *child;
52     result.push_back(root);
53     for (child = root->left_child; child; child = child->right_sibling)
54     {
55         OutputListRootNotAttend(child, result);
56     }
57 }
58
59 void OutputListRootNotAttend(Node *root, std::list<Node*>& result)
60 {
61     Node *child;
62     for (child = root->left_child; child; child = child->right_sibling)
63     {
64         if (child->dp_attend_if_parent_not_attend)
65             OutputListRootAttend(child, result);
66         else
67             OutputListRootNotAttend(child, result);
68     }
69 }
70
71 std::list<Node*> PlanningCompanyParty(Node *root)
72 {
73     std::list<Node*> result;
```

```
74     DpAuxRootAttend(root);
75     DpAuxRootNotAttend(root);
76     if (root->dp_attend_subtree_r_sum > root->dp_not_attend_subtree_r_sum)
77         OutputListRootAttend(root, result);
78     else
79         OutputListRootNotAttend(root, result);
80     return result;
81 }
```

## 15-7

Note that path in this book is referred "walk" by some other authors. Path is not defaulted to simple path. This says there might exist same vertices in a path. For more information, refer to page 1170.

(a)

Denote  $c[v, i] = 1$  where  $0 \leq i \leq k$  if there exists path that has  $\langle \sigma_1, \sigma_2, \dots, \sigma_i \rangle$  as its label; if there does not exist such path, then  $c[v, i] = 0$ . Let  $S = \exists u \in V$  such that  $(u, v) \in E$  and  $\sigma(u, v) = \sigma_i$ . We have the following recursive solution:

$$c[v, i] = \begin{cases} 1 & \text{if } i = 0 \\ \max_{\substack{(u, v) \in E \\ \sigma(u, v) = \sigma_i}} (c[u, i - 1]) & \text{if } 0 < i \leq k \text{ and } S \text{ is true} \\ 0 & \text{if } 0 < i \leq k \text{ and } S \text{ is false} \end{cases}$$

In another way, we denote  $d[u, i] = 1$  where  $0 \leq i \leq k$  if there exists path that has  $\langle \sigma_{i+1}, \dots, \sigma_k \rangle$  as its label; if there does not exist such path, then  $d[u, i] = 0$ . Let  $H = \exists v \in V$  such that  $(u, v) \in E$  and  $\sigma(u, v) = \sigma_{i+1}$ . We have the following recursive solution:

$$d[u, i] = \begin{cases} 1 & \text{if } i = k \\ \max_{\substack{(u, v) \in E \\ \sigma(u, v) = \sigma_{i+1}}} (d[v, i + 1]) & \text{if } 0 \leq i < k \text{ and } H \text{ is true} \\ 0 & \text{if } 0 \leq i < k \text{ and } H \text{ is false} \end{cases}$$

Therefore, we have the following memoized algorithm run in  $O(k|V|^2)$ :

```
1  bool ViterbiAux(Graph& graph, int v,
2      const std::vector<Sound>& sigma,
3      int sigma_index, std::list<int>& result,
4      std::vector< std::vector<bool> >& dp_checked)
5  {
6      bool exist_path;
7      if (sigma_index >= sigma.size())
```

```
8         return true;
9     if (dp_checked[v][sigma_index])// dp
10        return false;
11    dp_checked[v][sigma_index] = true;
12    for (const AdjListElement& edge : graph[v].adj_list)
13    {
14        if (sigma[sigma_index] == edge.sound)
15        {
16            if (ViterbiAux(graph, edge.vertex_index, sigma,
17                sigma_index + 1, result, dp_checked))
18            {
19                result.push_front(edge.vertex_index);
20                return true;
21            }
22        }
23    }
24    return false;
25 }
26
27 // index of sigma start from 0
28 // return empty list if NO-SUCH-PATH
29 std::list<int> Viterbi(Graph& graph, int v0,
30     const std::vector<int>& sigma)
31 {
32     std::list<int> result;
33     std::vector< std::vector<bool> > dp_checked(graph.size(),
34         std::vector<bool>(sigma.size(), false));
35     if (ViterbiAux(graph, v0, sigma, 0, result, dp_checked))
36         result.push_front(v0);
37     return result;
38 }
```

(b)

Denote  $c[v, i]$  where  $0 \leq i \leq k$  as the maximum probability of the path that has  $\langle \sigma_1, \sigma_2, \dots, \sigma_i \rangle$  as its label; if there does not exist such path, then  $c[v, i] = 0$ . Let  $S = \exists u \in V$  such that  $(u, v) \in E$  and  $\sigma(u, v) = \sigma_i$ . We have the following recursive solution:

$$c[v, i] = \begin{cases} 1 & \text{if } i = 0 \\ \max_{\substack{(u, v) \in E \\ \sigma(u, v) = \sigma_i}} (c[u, i-1] \cdot p(u, v)) & \text{if } 0 < i \leq k \text{ and } S \text{ is true} \\ 0 & \text{if } 0 < i \leq k \text{ and } S \text{ is false} \end{cases}$$

In another way, we denote  $d[u, i]$  where  $0 \leq i \leq k$  as the maximum probability of the path that has  $\langle \sigma_{i+1}, \dots, \sigma_k \rangle$  as its label; if there does not exist such path, then  $d[v, i] = 0$ . Let  $H = \exists v \in V$  such that  $(u, v) \in E$  and  $\sigma(u, v) = \sigma_{i+1}$ . We have the following recursive solution:

$$d[u, i] = \begin{cases} 1 & \text{if } i = k \\ \max_{\substack{(u, v) \in E \\ \sigma(u, v) = \sigma_{i+1}}} (d[v, i+1] \cdot p(u, v)) & \text{if } 0 \leq i < k \text{ and } H \text{ is true} \\ 0 & \text{if } 0 \leq i < k \text{ and } H \text{ is false} \end{cases}$$

Therefore, we have the following memoized algorithm run in  $O(k|V|^2)$ :

```
1  double ViterbiProbabilityAux(Graph& graph, int v,
2      const std::vector<Sound>& sigma,
3      int sigma_index, std::vector< std::vector<int> >& dp_path,
4      std::vector< std::vector<double> >& dp_probability)
5  {
6      double l_probability;
7      if (sigma_index >= sigma.size())
8          return 1;
9      if (dp_probability[v][sigma_index] >= 0) // dp
10         return dp_probability[v][sigma_index];
11     dp_probability[v][sigma_index] = 0;
12     for (const AdjListElement& edge : graph[v].adj_list)
13     {
14         if (sigma[sigma_index] == edge.sound)
15         {
16             l_probability = ViterbiProbabilityAux(graph, edge.vertex_index,
17                 sigma, sigma_index + 1, dp_path, dp_probability)
18                 * edge.probability;
19             if (l_probability > dp_probability[v][sigma_index])
20             {
21                 dp_probability[v][sigma_index] = l_probability;
22                 dp_path[v][sigma_index] = edge.vertex_index;
23             }
24         }
25     }
```

```
26     return dp_probability[v][sigma_index];
27 }
28
29 // index of sigma start from 0
30 // return empty list if NO-SUCH-PATH
31 std::list<int> ViterbiProbability(Graph& graph, int v0,
32     const std::vector<int>& sigma)
33 {
34     int l_vertex;
35     size_t vertex_size, sigma_size, i;
36     std::list<int> result;
37     vertex_size = graph.size();
38     sigma_size = sigma.size();
39     std::vector< std::vector<int> > dp_path(vertex_size,
40         std::vector<int>(sigma_size));
41     std::vector< std::vector<double> > dp_probability(vertex_size,
42         std::vector<double>(sigma_size, -1));
43     if (ViterbiProbabilityAux(graph, v0, sigma, 0, dp_path,
44         dp_probability) > 0)
45     {
46         l_vertex = v0;
47         result.push_back(l_vertex);
48         for (i = 0; i < sigma_size; ++i)
49         {
50             l_vertex = dp_path[l_vertex][i];
51             result.push_back(l_vertex);
52         }
53     }
54     return result;
55 }
```

## 15-8

### (a)

We form the seam from the top row to the bottom row. Assuming that  $n > 1$ , there are at least two choices of the column (same or adjacent column) of the next row for each pixel. This says the number of possible seams will be doubled at least if we increase  $m$  by 1. Thus, we obtain

$$T(r) \geq 2T(r - 1)$$

Hence, the number of possible seams grows in  $T(m) = \Omega(2^m)$ .

(b)

Denote  $c[i, j]$  as the lowest disruption of the seam from  $A[i, j]$  to the bottom row. We have the following recursive solution

$$c[i, j] = \begin{cases} \max(c[i + 1, j], c[i + 1, j + 1]) + d[i, j] & \text{if } 1 \leq i < m \text{ and } j = 1 \\ \max(c[i + 1, j - 1], c[i + 1, j], c[i + 1, j + 1]) + d[i, j] & \text{if } 1 \leq i < m \text{ and } 1 < j < n \\ \max(c[i + 1, j - 1], c[i + 1, j]) + d[i, j] & \text{if } 1 \leq i < m \text{ and } j = n \\ d[i, j] & \text{if } i = m \end{cases}$$

Therefore, we have the following bottom-up algorithm run in  $\Theta(mn)$ :

```
1  std::vector<int> LowestDisruptionSeam(const std::vector< std::vector<int> >& disruption)
2  {
3      int m, n, i, j;
4      m = disruption.size();
5      n = disruption[0].size();
6      std::vector< std::vector<int> > dp_disruption_sum(m, std::vector<int>(n)),
7          dp_col_of_next_row(m - 1, std::vector<int>(n));
8      std::vector<int> result;
9      result.reserve(m);
10     // compute
11     for (j = 0; j < n; ++j)
12     {
13         dp_disruption_sum[m - 1][j] = disruption[m - 1][n];
14     }
15     for (i = m - 2; i >= 0; --i)
16     {
17         if (dp_disruption_sum[i + 1][0] < dp_disruption_sum[i + 1][1])
18         {
19             dp_disruption_sum[i][0] = dp_disruption_sum[i + 1][0] + disruption[i][0];
20             dp_col_of_next_row[i][0] = 0;
21         }
22         else
23         {
24             dp_disruption_sum[i][0] = dp_disruption_sum[i + 1][1] + disruption[i][0];
25             dp_col_of_next_row[i][0] = 1;
26         }
27         for (j = 1; j < n - 1; ++j)
28         {
29             if (dp_disruption_sum[i + 1][j - 1] < dp_disruption_sum[i + 1][j])
```

```
30         {
31             dp_disruption_sum[i][j] = dp_disruption_sum[i + 1][j - 1];
32             dp_col_of_next_row[i][j] = j - 1;
33         }
34         else
35         {
36             dp_disruption_sum[i][j] = dp_disruption_sum[i + 1][j];
37             dp_col_of_next_row[i][j] = j;
38         }
39         if (dp_disruption_sum[i + 1][j + 1] < dp_disruption_sum[i][j])
40         {
41             dp_disruption_sum[i][j] = dp_disruption_sum[i + 1][j + 1];
42             dp_col_of_next_row[i][j] = j + 1;
43         }
44         dp_disruption_sum[i][j] += disruption[i][j];
45     }
46     if (dp_disruption_sum[i + 1][n - 2] < dp_disruption_sum[i + 1][n - 1])
47     {
48         dp_disruption_sum[i][n - 1] = dp_disruption_sum[i + 1][n - 2] +
49             disruption[i][n - 1];
50         dp_col_of_next_row[i][n - 1] = n - 2;
51     }
52     else
53     {
54         dp_disruption_sum[i][n - 1] = dp_disruption_sum[i + 1][n - 1] +
55             disruption[i][n - 1];
56         dp_col_of_next_row[i][n - 1] = n - 1;
57     }
58 }
59 // output
60 j = std::max_element(dp_disruption_sum.begin(), dp_disruption_sum.end()) -
61     dp_disruption_sum.begin();
62 result.push_back(j);
63 for (i = 0; i < m - 1; ++i)
64 {
65     j = dp_col_of_next_row[0][j];
66     result.push_back(j);
67 }
68 return result;
69 }
```

## 15-9

Denote  $c[i, j]$  where  $1 \leq i \leq j \leq n$  as the least cost to break the string  $S[i..j]$  by the break points  $x \in L$  such that  $x \in [i, j)$ . We have the following recursive solution

$$c[i, j] = \begin{cases} 0 & \text{if } \forall x \in [i, j), x \notin L \\ \min_{\substack{x \in L \\ x \in [i, j)}} (c[i, x] + c[x + 1, j]) + j - i + 1 & \text{otherwise} \end{cases}$$

In another way, we sort the array  $L[1..m]$  first, and let  $L[0] = 0$ ,  $L[m + 1] = n$ . Denote  $d[i, j]$  where  $0 \leq i < j \leq m + 1$  as the least cost to break the string  $S[(L[i] + 1)..L[j]]$  by the break points in  $L[(i + 1)..(j - 1)]$ . We have the following recursive solution

$$d[i, j] = \begin{cases} 0 & \text{if } j - i = 1 \\ \min_{i < x < j} (d[i, x] + d[x, j]) + L[j] - L[i] & \text{if } j - i > 1 \end{cases}$$

Therefore, we have the following bottom-up algorithm run in  $\Theta(m^3)$  (process is similar to 15.2-5):

```
1  void OutputSequence(const std::vector<int>& breaking_point,
2      const std::vector< std::vector<int> >& dp_break,
3      int i, int j, std::vector<int>& result)
4  {
5      int x;
6      if (j - i == 1)
7          return;
8      x = dp_break[i][j];
9      result.push_back(breaking_point[x]);
10     OutputSequence(breaking_point, dp_break, i, x, result);
11     OutputSequence(breaking_point, dp_break, x, j, result);
12 }
13
14 std::pair<int, std::vector<int> > BreakingString(std::vector<int>& breaking_point, int n)
15 {
16     int m, l, i, j, x, l_cost;
17     m = (int)(breaking_point.size());
18     std::vector< std::vector<int> > dp_cost(m + 2, std::vector<int>(m + 2)),
19         dp_break(m + 2, std::vector<int>(m + 2));
20     std::vector<int> result;
21     result.reserve(m);
22     breaking_point.push_back(breaking_point[0]);
23     breaking_point[0] = 0;
24     breaking_point.push_back(n);
```



```
25     Quicksort(breaking_point, 1, m);
26     for (i = 0; i <= m; ++i) // l = 1
27     {
28         dp_cost[i][i + 1] = 0;
29     }
30     for (l = 2; l <= m + 2; ++l)
31     {
32         for (i = 0; i <= m + 1 - l; ++i)
33         {
34             j = i + l;
35             dp_cost[i][j] = INT_MAX;
36             for (x = i + 1; x < j; ++x)
37             {
38                 l_cost = dp_cost[i][x] + dp_cost[x][j];
39                 if (l_cost < dp_cost[i][j])
40                 {
41                     dp_cost[i][j] = l_cost;
42                     dp_break[i][j] = x;
43                 }
44             }
45             dp_cost[i][j] += breaking_point[j] - breaking_point[i];
46         }
47     }
48     // output
49     OutputSequence(breaking_point, dp_break, 0, m + 1, result);
50     return std::make_pair(dp_cost[0][m + 1], result);
51 }
```

## 15-10

(a)

**Claim 3.** There exists an optimal investment strategy that, in each year, puts all the money into a single investment.

**Proof.** Assume there does not exist an optimal investment strategy that, in each year, puts all the money into a single investment. This says for all optimal investment strategy, there exists a year such that money is put in more than one investment.

For each year such that we shift money into  $k$  investments where  $1 < k \leq n$ . Let  $d$  be how many dollars we have at the begin of the year. Let  $\sum_{s=1}^k g_s = d - f_2$ . We put money  $g_1, g_2, \dots, g_k$  into

$i_1, i_2, \dots, i_k$  respectively in year  $j$ , At the end of the year  $j$ , we have  $u = \sum_{s=1}^k g_s \cdot r_{i_s j}$  dollars. Let  $i_m$  be the investment with the highest return rate in year  $j$ ; i.e.  $r_{i_m j} = \max(r_{i_1 j}, r_{i_2 j}, \dots, r_{i_n j})$ . If we put all the money into investment  $i_m$ , we have  $v = \sum_{s=1}^k g_s \cdot r_{i_m j}$  dollars at the end of the year  $j$ . We observed  $u \leq v$ . Contradiction  $\square$

(b)

**Proof.** Suppose that the optimal investment strategy for year 10 where year 10 put all money in  $i_{10}$  is contained in the sequence  $s_{10} = \langle i_1, i_2, \dots, i_9, i_{10} \rangle$ ; i.e. year  $k$  put all money in single investment  $i_k$ . Consider a year  $t$  such that  $1 < t < 10$ . Let strategy  $s_t = \langle i_1, i_2, \dots, i_{t-1}, i_t \rangle$ . Assume  $s_t$  is not the optimal strategy for year  $t$ . where year  $t$  put all money in  $i_t$ . Then there exists a optimal investment strategy for year  $t$  where year  $t$  put all money in  $i_t$ :  $s'_t = \langle i'_1, i'_2, \dots, i'_{t-1}, i_t \rangle$ . We can utilize “cut-and-paste” technique to replace  $i_1, i_2, \dots, i_{t-1}$  with  $i'_1, i'_2, \dots, i'_{t-1}$ , and let  $s'_{10} = \langle i'_1, i'_2, \dots, i'_{t-1}, i_t, \dots, i_{10} \rangle$ . The strategy  $s'_{10}$  result a greater amount of money after 10 years than the strategy  $s_{10}$ . Contradiction.  $\square$

(c)

Denote  $c[i, j]$  as the largest amount of money after year  $j$  where year  $j$  put all money in  $i$ . Then we have the following recursive solution:

$$c[i, j] = \begin{cases} 10000 \cdot r_{i1} & \text{if } j = 1 \\ \max_{1 \leq k \leq n} \begin{cases} (c[i, j-1] - f_1) \cdot r_{ij} \\ (c[k, j-1] - f_2) \cdot r_{ij} \end{cases} & \text{if } j > 1 \end{cases}$$

The following algorithm run in  $\Theta(10 \cdot n^2) = \Theta(n^2)$ :

```
1  std::pair<double, std::vector<int>> InvestmentStrategy
2      (const std::vector< std::vector<double>> & rate, int f_1, int f_2)
3  {
4      int i, j, k, n;
5      double l_money;
6      n = (int)(rate.size());
7      std::vector< std::vector<double>> dp_money(n, std::vector<double>(10));
8      std::vector< std::vector<int>> dp_investment(n, std::vector<int>(10));
9      std::vector<int> result(10);
10     // compute
11     for (i = 0; i < n; ++i) // j == 0
12     {
13         dp_money[i][0] = rate[i][0] * 10000;
14     }
```

```
15     for (j = 1; j < 10; ++j)
16     {
17         for (i = 0; i < n; ++i)
18         {
19             dp_money[i][j] = (dp_money[i][j - 1] - f_1) * rate[i][j];
20             dp_investment[i][j] = i;
21             for (k = 0; k < n; ++k)
22             {
23                 l_money = (dp_money[k][j - 1] - f_2) * rate[i][j];
24                 if (l_money > dp_money[i][j])
25                 {
26                     dp_money[i][j] = l_money;
27                     dp_investment[i][j] = k;
28                 }
29             }
30         }
31     }
32     // output
33     l_money = dp_money[0][9];
34     result[9] = 0;
35     for (i = 1; i < n; ++i)
36     {
37         if (dp_money[i][9] > l_money)
38         {
39             l_money = dp_money[i][9];
40             result[9] = i;
41         }
42     }
43     for (j = 9; j > 0; --j)
44     {
45         result[j - 1] = dp_investment[result[j]][j];
46     }
47     return std::make_pair(l_money, std::move(result));
48 }
```

(d)

If the restriction is imposed, the subproblems are depend on initial amount of money. We cannot solve the subproblems for each initial amount of money.

## 15-11

Let  $D_t = \sum_{i=1}^t d_i$  and  $h(0) = 0$ . Denote  $cost[i, j]$  as the minimum cost to fulfill  $j$  ( $j \geq D_i$ ) demand in month 1 to  $i$ . We have the following recursive solution:

$$c[i, j] = \begin{cases} h(j - d_1) + c \cdot \max(j - m, 0) & \text{if } i = 1 \\ \min_{D_{i-1} \leq k \leq j} (cost[i-1, k] + h(j - D_i) + c \cdot \max(j - k - m, 0)) & \text{if } i > 1 \end{cases}$$

The following algorithm run in  $O(n \cdot D_n^2) = O(n \cdot D^2)$ :

```
1  std::pair<int, std::vector<int> > InventoryPlanning
2      (const std::vector<int>& demands, int m, int c,
3      const std::vector<int>& holding_cost)
4  {
5      int j, k, l_cost;
6      size_t n, i;
7      n = demands.size();
8      std::vector<int> sum_demands(n), result(n);
9      // compute
10     sum_demands[0] = demands[0];
11     for (i = 1; i < n; ++i)
12     {
13         sum_demands[i] += sum_demands[i - 1] + demands[i];
14     }
15     std::vector< std::vector<int> > dp_cost(n, std::vector<int>(sum_demands[n - 1])),
16         dp_plan(n, std::vector<int>(sum_demands[n - 1]));
17     for (j = sum_demands[0]; j <= sum_demands[n - 1]; ++j)
18     {
19         dp_cost[0][j] = holding_cost[j - sum_demands[0]] +
20             c * std::max(j - m, 0);
21     }
22     for (i = 1; i < n; ++i)
23     {
24         for (j = sum_demands[i]; j <= sum_demands[n - 1]; ++j)
25         {
26             dp_cost[i][j] = INT_MAX;
27             for (k = sum_demands[i - 1]; k <= j; ++k)
28             {
29                 l_cost = dp_cost[i - 1][k] + c * std::max(j - k - m, 0);
30                 if (l_cost < dp_cost[i][j])
31                 {
```

```
32         dp_cost[i][j] = l_cost;
33         dp_plan[i][j] = k;
34     }
35 }
36 dp_cost[i][j] += holding_cost[j - sum_demands[i]];
37 }
38 }
39 // output
40 result[n - 1] = sum_demands[n - 1];
41 for (i = n - 2; i < n; --i)
42 {
43     result[i] = dp_plan[i + 1][result[i + 1]];
44     result[i + 1] -= result[i];
45 }
46 return std::make_pair(dp_cost[n - 1][sum_demands[n - 1]], std::move(result));
47 }
```

## 15-12

Let *players* be the array contains the finite sets of availble players for each postion. That is let *players*[*i*] be the finite set of availble players for position *i*. Hence the size of array *players* is *N*. For all  $i \in [1, N]$ , the cardinality of finite set *players*[*i*] is *P*. Denote *max\_vorp*[*i*, *c*] as the maximum VORP to sign players at position 1..*i* by at most a budget of \$*c*. We have the following recursive solution:

$$\text{max\_vorp}[i, c] = \begin{cases} \max_{\substack{p \in \text{players}[1] \\ p.\text{cost} \leq c}} (p.\text{vorp}) & \text{if } i = 1 \\ \max \begin{cases} \text{max\_vorp}[i - 1, c] \\ \max_{\substack{p \in \text{players}[i] \\ p.\text{cost} \leq c}} (\text{max\_vorp}[i - 1, c - p.\text{cost}] + p.\text{vorp}) \end{cases} & \text{if } i > 1 \end{cases}$$

The following algorithm run in  $\Theta(NXP)$ :

```
1 Result SigningPlayers(const std::vector< std::vector<Player> >& players, int budget)
2 {
3     int i, j, c, pos_size, player_size, l_vorp, l_budget;
4     pos_size = (int)(players.size());
5     player_size = (int)(players[0].size());
6     std::vector< std::vector<int> > dp_vorp(pos_size, std::vector<int>(budget + 1)),
7         dp_player(pos_size, std::vector<int>(budget + 1));
8     std::vector<int> result(pos_size);
9     // compute
```

```
10     for (c = 0; c <= budget; ++c)
11     {
12         dp_vorp[0][c] = INT_MIN;
13         for (j = 0; j < player_size; ++j)
14         {
15             if (players[0][j].cost <= c && players[0][j].vorp > dp_vorp[0][c])
16             {
17                 dp_vorp[0][c] = players[0][j].vorp;
18                 dp_player[0][c] = j;
19             }
20         }
21     }
22     for (i = 1; i < pos_size; ++i)
23     {
24         for (c = 0; c <= budget; ++c)
25         {
26             dp_vorp[i][c] = dp_vorp[i - 1][c];
27             dp_player[i][c] = INT_MIN; // means no player for position c
28             for (j = 0; j < player_size; ++j)
29             {
30                 if (players[i][j].cost <= c)
31                 {
32                     l_vorp = dp_vorp[i - 1][c - players[i][j].cost] + players[i][j].vorp;
33                     if (l_vorp > dp_vorp[i][c])
34                     {
35                         dp_vorp[i][c] = l_vorp;
36                         dp_player[i][c] = j;
37                     }
38                 }
39             }
40         }
41     }
42     // output
43     l_budget = budget;
44     for (i = pos_size - 1; i >= 0; --i)
45     {
46         result[i] = dp_player[i][l_budget];
47         if (result[i] >= 0)
48             l_budget -= players[i][result[i]].cost;
49     }
```

```
50     return { dp_vorp[pos_size - 1][budget], // total_vorp
51             budget - l_budget, // total_cost
52             std::move(result) // players
53     };
54 }
```