Chapter 15 Solusion

frc6.com

i@frc6.com

https://github.com/frc123/CLRS-code-solution

11/3/2021

15.1

15.1-1

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

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

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

```
/**
* running time: O(n^2)
* p: table of prices (index start from 0)
* n: length of rod
* c: cost of each cut
* return maximum revenue
*/
int BottomUpCutRodWithCost(const std::vector<int>& p, int n, int c)
{
    int *r, q, i, j;
    r = new int[n + 1];
    r[0] = 0;
```

```
    for (j = 1; j <= n; ++j)
    {
        q = p[j - 1];
        for (i = 0; i < j - 1; ++i)
            q = std::max(q, p[i] + r[j - i - 1] - c);
        r[j] = q;
    }
    delete[] r;
    return q;
}
```

15.1-4

```
/**
* p: table of prices (index start from 0)
* n: length of rod
* r: table of maximum revenue (index start from 1)
* s: table of optimal size i of the first piece to cut off (index start from 1)
* return maximum revenue
*/
int ExtendedMemoizedCutRodAux(const std::vector<int>& p, int n, int *r, int *s)
{
    int q, i, reminder_r;
    if (r[n] >= 0) return r[n];
    q = INT_MIN;
    for (i = 0; i < n; ++i)
    {
        reminder_r = ExtendedMemoizedCutRodAux(p, n - i - 1, r, s);
        if (q < p[i] + reminder_r)
        {
            q = p[i] + reminder_r;
            s[n] = i + 1;
        }
    }
    r[n] = q;
    return q;
}

/**
* running time: O(n^2)
```

```
* p: table of prices (index start from 0)
* n: length of rod
* return (r, s)
* r: table of maximum revenue (index start from 1)
* s: table of optimal size i of the first piece to cut off (index start from 1)
* caller is responsible to deallocate return value r and s
*/
std::pair<int*, int*> ExtendedMemoizedCutRod(const std::vector<int>& p, int n)
{
    int *r, *s, i;
    r = new int[n + 1];
    s = new int[n + 1];
    r[0] = 0;
    s[0] = 0;
    for (i = 1; i <= n; ++i) r[i] = INT_MIN;
    ExtendedMemoizedCutRodAux(p, n, r, s);
    return std::make_pair(r, s);
}
```

15.1-5

```
/**
* running time: O(n)
* n: n-th fibonacci number (must greater than 0)
*/
int FibonacciNumber(int n)
{
    int *f, i, result;
    f = new int[n + 1];
    f[0] = 0;
    f[1] = 1;
    for (i = 2; i <= n; ++i)
        f[i] = f[i - 1] + f[i - 2];
    result = f[n];
    delete[] f;
    return result;
}
```

15.2

15.2-1

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

Minimum cost: 2010

15.2-2

```
/**
* s: table (2d) storing index of k achieved the optimal cost
*       (index start by 1)
* caller is responisble to deallocate the return value
*/
Matrix* MatrixChainMultiply
    (const std::vector<Matrix*>& matrices, const Table* s, int i, int j)
{
    Matrix *matrix_a, *matrix_b, *matrix_c;
    if (i == j)
    {
        return matrices[i - 1];
    }
    matrix_a = MatrixChainMultiply(matrices, s, i, (*s)[i][j]);
    matrix_b = MatrixChainMultiply(matrices, s, (*s)[i][j] + 1, j);
    matrix_c = MatrixMultiply(matrix_a, matrix_b);
    if (i != (*s)[i][j]) delete matrix_a;
    if ((*s)[i][j] + 1 != j) delete matrix_b;
    return matrix_c;
}
```

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:

$$\sum_{i=1}^{n}\sum_{j=i}^{n}(j-i)=\sum_{i=1}^{n}(\sum_{j=i}^{n}(j)-\sum_{j=i}^{n}(i))=\sum_{i=1}^{n}(\sum_{j=1}^{n}(j)-\sum_{j=1}^{i-1}(j)-(n-i+1)i)$$
$$=\sum_{i=1}^{n}(\tfrac{n(n+1)}{2}-\tfrac{(i-1)i}{2}-(n-i+1)i)=\tfrac{n^2(n+1)}{2}-\sum_{i=1}^{n}(\tfrac{i-i^2+2ni}{2})$$
$$=\tfrac{n^2(n+1)}{2}+\tfrac{1}{2}\sum_{i=1}^{n}(i^2)-\tfrac{1}{2}(1+2n)\sum_{i=1}^{n}(i)=\tfrac{n^2(n+1)}{2}+\tfrac{n(n+1)(2n+1)}{12}-\tfrac{n(n+1)(2n+1)}{4}$$
$$=\tfrac{n^2(n+1)}{2}-\tfrac{n(n+1)(2n+1)}{6}=\tfrac{(n-1)n(n+1)}{6}$$

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

$$\sum_{i=1}^{n}\sum_{j=i}^{n}R(i,j)=\sum_{l=2}^{n}\sum_{i=1}^{n-l+1}\sum_{k=i}^{i+l-2}2=2\sum_{l=2}^{n}(n-l+1)(l-1)=2((n+2)\sum_{l=2}^{n}l-\sum_{l=2}^{n}l^2-(n-1)(n+1))$$
$$=2((n+2)(\tfrac{n(n+1)}{2}-1)-(\tfrac{n(n+1)(2n+1)}{6}-1))=\tfrac{n^3-n}{3}$$ □

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: $A_1A_2...A_nA_{n+1}$. 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. □

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

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

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

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

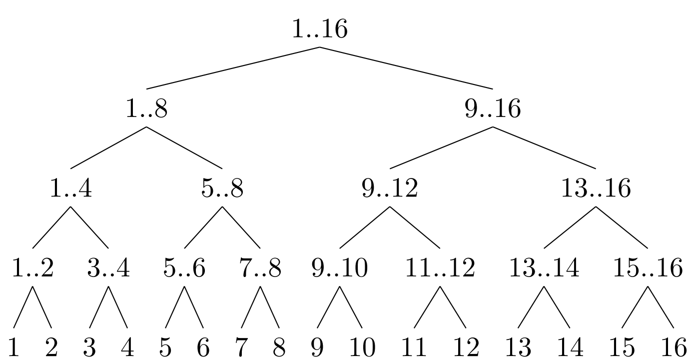
$$T(n) = 2\sum_{i=1}^{n-1} T(i) + n$$

This says we want to prove that $T(n) = o(4^n/n^{3/2})$. Since $\lim_{n\to\infty} \frac{4^n/n^{3/2}}{3.5^n} = \infty$, 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 subsitution method. Let c be some constant. Assume $T(n) \leq c \cdot 3.5^n$.

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

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, $T(n) = o(4^n/n^{3/2})$. □

15.3-2



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

15.3-3

Yes.

Proof. Let $A_{i...j}$ denotes sequece of matrices $A_iA_{i+1}...A_j$.
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. □

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 $p_0p_1p_3 = 1000$ and $p_0p_2p_3 = 2000$. Hence, the solution of greedy algorithm is $(A_1(A_2A_3))$.
However, $((A_1A_2)A_3)$ $(k = 2)$ is the optimal solution, which takes $p_0p_1p_2 + p_0p_2p_3 = 200 + 2000 = 2200$ multiplications. Greedy solution $(A_1(A_2A_3))$ $(k = 1)$ takes $p_1p_2p_3 + p_0p_1p_3 = 20000 + 1000 = 21000$ 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.

$$m[1,3] = \begin{cases} m[1,1] + m[2,3] + p_0p_1p_3 & k = 1 \\ m[1,2] + m[3,3] + p_0p_2p_3 & k = 2 \end{cases} = \begin{cases} p_1p_2p_3 + p_0p_1p_3 & k = 1 \\ p_0p_1p_2 + p_0p_2p_3 & k = 2 \end{cases}$$

We try to make the greedy algorithm choose $k = 1$. This says we want $p_0p_1p_3 < p_0p_2p_3 \iff p_1 < p_2$.
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 $p_1p_2p_3 + p_0p_1p_3 > p_0p_1p_2 + p_0p_2p_3$.
Hence, our goal is to find p_0, p_1, p_2, p_3 such that $p_1 < p_2$ and $p_1p_2p_3 + p_0p_1p_3 > p_0p_1p_2 + p_0p_2p_3$. 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 $k = 1, 2..., n$, 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 $k_1, k_2, k_3, ..., k_{n-1}, k_n$ be the best sequence to exchange from currency from k_1 to currency k_n, which means $r_{k_1k_2}r_{k_2k_3}...r_{k_{n-1}k_n}$ is maximized. We show that the sequence of currencies $k_i, k_{i+1}, ..., k_{j-1}, k_j$ is the best sequence to exchange from currency from k_i to currency k_j by contradiction. Assume $k_i, q_{i+1}, ..., q_{j-1}, k_j$ is the best sequence to exchange from currency from k_i to currency k_j. By using the "cut-and-paste" technique to replace $k_i, k_{i+1}, ..., k_{j-1}, k_j$ with $k_i, q_{i+1}, ..., q_{j-1}, k_j$, we get a better sequence of currencies to exchange from currency from k_1 to currency k_n (e.g. $k_1, k_2, k_3, ...k_i, q_{i+1}, ..., q_{j-1}, k_j, ..., k_{n-1}, k_n$), which contradicts to $k_1, k_2, k_3, ..., k_{n-1}, k_n$ is the best sequence. □

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.

$r_{k_1k_2}$	$r_{k_2k_3}$	$r_{k_3k_4}$	$r_{k_1k_3}$	$r_{k_2k_4}$	$r_{k_1k_4}$
6	2	5	2	8	10

c_1	c_2	c_3
4	5	20

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

k_1k_4: $r_{k_1k_4} - c_1 = 10 - 4 = 6$

$k_1k_2k_4$: $r_{k_1k_2}r_{k_2k_4} - c_2 = 6 \cdot 8 - 5 = 43$

$k_1k_3k_4$: $r_{k_1k_3}r_{k_3k_4} - c_2 = 2 \cdot 5 - 5 = 5$

$k_1k_2k_3k_4$: $r_{k_1k_2}r_{k_2k_3}r_{k_3k_4} - c_3 = 6 \cdot 2 \cdot 5 - 20 = 40$

$k_1k_3k_2k_4$: $r_{k_1k_3}r_{k_3k_2}r_{k_2k_4} - c_3 = 2 \cdot \frac{1}{2} \cdot 8 - 20 = -12$

The optimal sequence from k_1 to k_4 is $k_1k_2k_4$.

Now, we try to show that k_2k_4 is not the optimal sequence from k_2 to k_4 by list all possible sequences:

k_2k_4: $r_{k_2k_4} - c_1 = 8 - 4 = 4$

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

$k_2k_1k_4$: (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

```
template <typename T>
std::list<T> OutputLCS(LengthTable length_table,
    const std::vector<T>& sequence_x, const std::vector<T>& sequence_y)
{
    size_t it_x, it_y;
    std::list<T> common_sequence;
    it_x = sequence_x.size();
    it_y = sequence_y.size();
    while (it_x > 0 && it_y > 0)
    {
        const T& element_x = sequence_x[it_x - 1];
        const T& element_y = sequence_y[it_y - 1];
        if (element_x == element_y)
        {
            common_sequence.push_front(element_y);
            --it_x;
            --it_y;
        }
        else if (length_table[it_x - 1][it_y] >= length_table[it_x][it_y - 1])
        {
            --it_x;
        }
```

```
            else
            {
                --it_y;
            }
        }
        return common_sequence;
    }
```

15.4-3

```
    template <typename T>
    int RecursiveLCSLengthMemoized
        (const std::vector<T>& sequence_x, const std::vector<T>& sequence_y,
        size_t it_x, size_t it_y,
        LengthTable& length_table, PointerTable& pointer_table)
    {
        int sequence_length_x_prev, sequence_length_y_prev;
        if (length_table[it_x][it_y] >= 0)
            return length_table[it_x][it_y];
        if (sequence_x[it_x - 1] == sequence_y[it_y - 1])
        {
            length_table[it_x][it_y] = RecursiveLCSLengthMemoized(sequence_x, sequence_y,
                it_x - 1, it_y - 1, length_table, pointer_table) + 1;
            pointer_table[it_x][it_y] = PointerType::X_Y_PREV;
        }
        else
        {
            sequence_length_x_prev = RecursiveLCSLengthMemoized(sequence_x, sequence_y,
                it_x - 1, it_y, length_table, pointer_table);
            sequence_length_y_prev = RecursiveLCSLengthMemoized(sequence_x, sequence_y,
                it_x, it_y - 1, length_table, pointer_table);
            if (sequence_length_x_prev >= sequence_length_y_prev)
            {
                length_table[it_x][it_y] = sequence_length_x_prev;
                pointer_table[it_x][it_y] = PointerType::X_PREV;
            }
            else
            {
                length_table[it_x][it_y] = sequence_length_y_prev;
                pointer_table[it_x][it_y] = PointerType::Y_PREV;
```

```
        }
    }
    return length_table[it_x][it_y];
}

template <typename T>
std::pair<LengthTable, PointerTable> LCSLengthMemoized
    (const std::vector<T>& sequence_x, const std::vector<T>& sequence_y)
{
    size_t size_x, size_y, it_x, it_y,
        sequence_length_x_prev, sequence_length_y_prev;
    size_x = sequence_x.size();
    size_y = sequence_y.size();
    // note this line init all elements in length_table with -1
    LengthTable length_table(size_x + 1, LengthTableRow(size_y + 1, -1));
    PointerTable pointer_table(size_x + 1, PointerTableRow(size_y + 1, NIL));
    for (it_x = 0; it_x <= size_x; ++it_x)
        length_table[it_x][0] = 0;
    for (it_y = 1; it_y <= size_y; ++it_y)
        length_table[0][it_y] = 0;
    RecursiveLCSLengthMemoized(sequence_x, sequence_y, size_x, size_y,
        length_table, pointer_table);
    return std::make_pair(std::move(length_table), std::move(pointer_table));
}
```

15.4-4

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

```
template <typename T>
std::pair<int, PointerTable> LCSLengthByTwoRowLengthTable
    (const std::vector<T>& sequence_x, const std::vector<T>& sequence_y)
{
    size_t size_x, size_y, it_x, it_y;
    int sequence_length_x_prev, sequence_length_y_prev;
    size_x = sequence_x.size();
    size_y = sequence_y.size();
    if (size_x < size_y)
        return LCSLengthByTwoRowLengthTable(sequence_y, sequence_x);
    LengthTableRow length_table_row_1(size_y + 1), length_table_row_2(size_y + 1);
    PointerTable pointer_table(size_x + 1, PointerTableRow(size_y + 1));
```

```
    for (it_y = 0; it_y <= size_y; ++it_y)
        length_table_row_1[it_y] = 0;
    for (it_x = 1; it_x <= size_x; ++it_x)
    {
        for (it_y = 1; it_y <= size_y; ++it_y)
        {
            if (sequence_x[it_x - 1] == sequence_y[it_y - 1])
            {
                length_table_row_2[it_y] =
                    length_table_row_1[it_y - 1] + 1;
                pointer_table[it_x][it_y] = PointerType::X_Y_PREV;
            }
            else
            {
                sequence_length_x_prev = length_table_row_1[it_y];
                sequence_length_y_prev = length_table_row_2[it_y - 1];
                if (sequence_length_x_prev >= sequence_length_y_prev)
                {
                    length_table_row_2[it_y] = sequence_length_x_prev;
                    pointer_table[it_x][it_y] = PointerType::X_PREV;
                }
                else
                {
                    length_table_row_2[it_y] = sequence_length_y_prev;
                    pointer_table[it_x][it_y] = PointerType::Y_PREV;
                }
            }
        }
        length_table_row_1.swap(length_table_row_2);
    }
    return std::make_pair(length_table_row_2[size_y], std::move(pointer_table));
}
```

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

```
template <typename T>
std::pair<int, PointerTable> LCSLengthByOneRowLengthTable
    (const std::vector<T>& sequence_x, const std::vector<T>& sequence_y)
{
    size_t size_x, size_y, it_x, it_y;
    int sequence_length_x_y_prev, sequence_length_x_y_now;
```

```
        size_x = sequence_x.size();
        size_y = sequence_y.size();
        if (size_x < size_y)
            return LCSLengthByOneRowLengthTable(sequence_y, sequence_x);
        LengthTableRow length_table_row(size_y + 1);
        PointerTable pointer_table(size_x + 1, PointerTableRow(size_y + 1));
        for (it_y = 0; it_y <= size_y; ++it_y)
            length_table_row[it_y] = 0;
        for (it_x = 1; it_x <= size_x; ++it_x)
        {
            sequence_length_x_y_prev = 0;
            for (it_y = 1; it_y <= size_y; ++it_y)
            {
                sequence_length_x_y_now = length_table_row[it_y];
                if (sequence_x[it_x - 1] == sequence_y[it_y - 1])
                {
                    length_table_row[it_y] =
                        sequence_length_x_y_prev + 1;
                    pointer_table[it_x][it_y] = PointerType::X_Y_PREV;
                }
                else
                {
                    if (length_table_row[it_y] >= length_table_row[it_y - 1])
                    {
                        // length_table_row[it_y] = length_table_row[it_y];
                        pointer_table[it_x][it_y] = PointerType::X_PREV;
                    }
                    else
                    {
                        length_table_row[it_y] = length_table_row[it_y - 1];
                        pointer_table[it_x][it_y] = PointerType::Y_PREV;
                    }
                }
                sequence_length_x_y_prev = sequence_length_x_y_now;
            }
        }
        return std::make_pair(length_table_row[size_y], std::move(pointer_table));
    }
```

15.4-5

```cpp
// O(n^2)
std::vector<int> LongestIncreasingSubsequence
    (const std::vector<int>& sequence)
{
    size_t i, j, size, sequence_length_it, max_length_index;
    size = sequence.size();
    std::vector<size_t> length(size), prev(size);
    // compute
    max_length_index = 0;
    for (i = 0; i < size; ++i)
    {
        length[i] = 1;
        for (j = 0; j < i; ++j)
        {
            if (sequence[j] < sequence[i])
            {
                sequence_length_it = length[j] + 1;
                if (sequence_length_it > length[i])
                {
                    length[i] = sequence_length_it;
                    prev[i] = j;
                }
            }
        }
        if (length[i] > length[max_length_index])
            max_length_index = i;
    }
    // output
    std::vector<int> result(length[max_length_index]);
    for (i = length[max_length_index] - 1; i < length[max_length_index]; --i)
    {
        result[i] = sequence[max_length_index];
        max_length_index = prev[max_length_index];
    }
    return result;
}
```

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

15.4-6

```
// O(nlgn)
std::vector<int> LongestIncreasingSubsequenceBinarySerach
    (const std::vector<int>& sequence)
{
    size_t i, size, seq_index;
    int lower, upper, middle;
    size = sequence.size();
    /**
    * prev
    * index is the element index of sequence
    * value is the element index of sequence
    *       such that the element is the prev element
    *       in the increasing subsequence
    * ---------------------------------------------------
    * sub_seq
    * index is length of the increasing subsequence
    * value is the element index of sequence
    */
    std::vector<size_t> prev(size), sub_seq;
    sub_seq.reserve(size);
    // compute
    for (i = 0; i < size; ++i)
    {
        lower = 0;
        upper = sub_seq.size() - 1;
        while (lower <= upper)
        {
            middle = lower + ((upper - lower) >> 1);
            if (sequence[sub_seq[middle]] > sequence[i])
            {
                upper = middle - 1;
            }
            else if (sequence[sub_seq[middle]] < sequence[i])
            {
                lower = middle + 1;
            }
            else
            {
```

```
                lower = middle;
                break;
            }
        }
        // sequence[i] <= sequence[sub_seq[lower]] must be true
        if (lower >= sub_seq.size())
            sub_seq.push_back(i);
        else
            sub_seq[lower] = i;
        if (lower > 0)
            prev[i] = sub_seq[lower - 1];
    }
    // output
    size = sub_seq.size();
    std::vector<int> result(size);
    seq_index = sub_seq[size - 1];
    for (i = size - 1; i < size; --i)
    {
        result[i] = sequence[seq_index];
        seq_index = prev[seq_index];
    }
    return result;
}
```

15.5

15.5-1

```
void ConstructOptimalBST(std::ostream& os, const Table& root,
    const std::string& suffix, size_t i, size_t j)
{
    if (j == i - 1)
    {
        os << "d_" << j << " is the " << suffix << "\n";
    }
    else
    {
        size_t r = root[i][j];
        os << "k_" << r << " is the " << suffix << "\n";
        ConstructOptimalBST(os, root,
```

```
                "left child of k_" + std::to_string(r), i, r - 1);
            ConstructOptimalBST(os, root,
                "right child of k_" + std::to_string(r), r + 1, j);
        }
    }

    void ConstructOptimalBST(std::ostream& os, const Table& root)
    {
        ConstructOptimalBST(os, root, "root", 1, root.size() - 1);
    }
```

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 computer $w(i,j)$ 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

```
// O(n^2)
std::pair<Table, Table> OptimalBST(const Row& p, const Row& q)
{
    size_t q_size, i, j, l, r;
    int t;
    q_size = q.size();// n + 1
    Table e(q_size + 1, Row(q_size)), w(q_size + 1, Row(q_size)),
        root(q_size, Row(q_size));
    for (i = 1; i <= q_size; ++i)// 1 to (n + 1)
    {
        e[i][i - 1] = q[i - 1];
        // w[i][i - 1] = q[i - 1];
    }
    // l = 1
    for (i = 1; i < q_size; ++i)// 1 to n
    {
        // w[i][i] = w[i][i - 1] + p[i] + q[i];
        w[i][i] = q[i - 1] + p[i] + q[i];
        // e[i][i] = e[i][i - 1] + e[i + 1][i] + w[i][i];
        e[i][i] = q[i - 1] + q[i] + w[i][i];
        root[i][i] = i;
    }
    // l = 2 ... n
    for (l = 2; l < q_size; ++l)// 2 to n
    {
        for (i = 1; i <= q_size - l; ++i)// 1 to (n - l + 1)
        {
```

```
                j = i + l - 1;
                e[i][j] = INT_MAX;
                // root[i][j - 1] to root[i + 1][j]
                for (r = root[i][j - 1]; r <= root[i + 1][j]; ++r)
                {
                    t = e[i][r - 1] + e[r + 1][j];
                    if (t < e[i][j])
                    {
                        e[i][j] = t;
                        root[i][j] = r;
                    }
                }
                w[i][j] = w[i][j - 1] + p[j] + q[j];
                e[i][j] += w[i][j];
            }
        }
        return std::make_pair(std::move(e), std::move(root));
    }
```

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 varible 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 $k \leq h$.

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

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

□

Chapter 15 Problems

15-1

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

```
// O(E)
int RecursiveLongestSimplePath(const Graph& graph, int start, int dist,
    Row& length, Row& next)
{
    int curr_length;
    if (length[start] >= 0)
        return length[start];
    for (const AdjListElement& adj_element : graph[start])
    {
        curr_length = adj_element.weight + RecursiveLongestSimplePath(graph,
            adj_element.vertex, dist, length, next);
        if (curr_length > length[start])
        {
            length[start] = curr_length;
            next[start] = adj_element.vertex;
        }
    }
    return length[start];
}

// O(V + E)
std::list<int> LongestSimplePath(const Graph& graph, int start, int dist)
{
    int vertex;
    size_t size;
```

```
    std::list<int> result;
    size = graph.size();
    Row length(size, INT_MIN), next(size);
    length[dist] = 0;
    RecursiveLongestSimplePath(graph, start, dist, length, next);
    if (length[start] < 0)
        return result;
    vertex = start;
    while (vertex != dist)
    {
        result.push_back(vertex);
        vertex = next[vertex];
    }
    result.push_back(dist);
    return result;
}
```

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

15-2

Denote $X_{ij} = \langle x_i, x_{i+1}, \cdots, x_{j-1}, x_j \rangle$ and $Z_{kt} = \langle z_k, z_{k+1}, \cdots, z_{t-1}, z_t \rangle$. 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 $\Theta(n^2)$:

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

```
    for (i = 0; i < size - 1; ++i)
    {
        lengths[i][i] = 1;
        j = i + 1;
        if (str[i] == str[j])
        {
            lengths[i][j] = 2;
            pointers[i][j] = COMMON;
        }
        else
        {
            lengths[i][j] = 1;
            pointers[i][j] = I_SUCC;
        }
    }
    lengths[i][i] = 1;// i == size - 1
    for (i = size - 3; i < size; --i)
    {
        for (j = i + 2; j < size; ++j)
        {
            if (str[i] == str[j])
            {
                lengths[i][j] = lengths[i + 1][j - 1] + 2;
                pointers[i][j] = COMMON;
            }
            else if (lengths[i + 1][j] >= lengths[i][j - 1])
            {
                lengths[i][j] = lengths[i + 1][j];
                pointers[i][j] = I_SUCC;
            }
            else
            {
                lengths[i][j] = lengths[i][j - 1];
                pointers[i][j] = J_PREV;
            }
        }
    }
    // output
    i = 0;
    j = size - 1;
```

```
        lps_left = 0;
        lps_right = lengths[i][j] - 1;
        result.resize(lengths[i][j]);
        while (i < j)
        {
            if (pointers[i][j] == I_SUCC)
            {
                ++i;
            }
            else if (pointers[i][j] == J_PREV)
            {
                --j;
            }
            else
            {
                result[lps_left] = str[i];
                result[lps_right] = str[j];
                ++i;
                --j;
                ++lps_left;
                --lps_right;
            }
        }
        if (i == j)
        {
            result[lps_left] = str[i];
        }
        return result;
    }
```

15-3

Sort points by x-coordinate first, and put the sorted points in a sequence $\langle p_1, p_2, \cdots, p_n \rangle$. 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 $1 \le a \le i$ and $1 \le b \le j$. 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:

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

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

```
std::vector<size_t> BitonicTour(std::vector<Point>& points)
{
    int l_length;
    size_t size, i, j, b, result_left_p, result_right_p;
    size = points.size();
    Table t_length(size, Row(size, INT_MAX)),
        t_left_neighbour(size, Row(size));
    // r_neighbour_distance[j] == Distance(points[j - 1], points[j])
    Row r_neighbour_distance(size);
    std::vector<size_t> result(size);
    // sort points by x-coordinate
    QuickSort(points, 0, (int)size - 1);
    // compute
    for (j = 2; j < size - 1; ++j)
    {
        r_neighbour_distance[j] = Distance(points[j - 1], points[j]);
    }
    t_length[0][1] = Distance(points[0], points[1]);
    t_left_neighbour[0][1] = 0;
    for (i = 0; i < size - 1; ++i)
    {
        j = i + 1;
        for (b = 0; b < i; ++b)
        {
            l_length = t_length[b][i] + Distance(points[b], points[j]);
            if (l_length < t_length[i][j])
            {
                t_length[i][j] = l_length;
                t_left_neighbour[i][j] = b;
            }
        }
```

```
            for (j = i + 2; j < size - 1; ++j)
            {
                t_length[i][j] = t_length[i][j - 1] + r_neighbour_distance[j];
                t_left_neighbour[i][j] = j - 1;
            }
        }
        // output
        result_left_p = 0;
        result_right_p = size - 1;
        i = size - 2;
        j = size - 1;
        while (i != j)
        {
            if (i < j)
            {
                result[result_left_p] = j;
                ++result_left_p;
                j = t_left_neighbour[i][j];
            }
            else
            {
                result[result_right_p] = i;
                --result_right_p;
                i = t_left_neighbour[j][i];
            }
        }
        result[result_left_p] = 0;
        return result;
    }
```

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 $e(i,j) = M - j + i - \sum\limits_{k=1}^{j} l_k$. 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:

$$c[1,j] = \begin{cases} e(1,j) & \text{if } t = 1. \\ \min\limits_{t \le k \le j} (c[1,k-1] + e(k,j)) & \text{if } t \ge 2. \end{cases}$$

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

```
std::list<int> PrintingNeatly(int line_max_char,
    const std::vector<int>& lengths)
{
    int size, j, k, extra;
    size = (int)(lengths.size());
    /**
    * min_sum[k + 1] is the minimum extra char
    * from the first word to the word which the length is lengths[k]
    */
    std::vector<int> min_sum(size + 1);
    std::vector<int> line_start_index(size);
    std::list<int> result;
    // compute
    min_sum[0] = 0;
    for (j = 0; j < size - 1; ++j)
    {
        min_sum[j + 1] = INT_MAX;
        extra = line_max_char + 1;
        for (k = j; k >= 0; --k)
        {
            extra = extra - 1 - lengths[k];
            if (extra < 0)
                break;
            if (min_sum[k] + extra < min_sum[j + 1])
            {
                min_sum[j + 1] = min_sum[k] + extra;
                line_start_index[j] = k;
            }
        }
    }
    extra = line_max_char + 1;
    min_sum[size] = INT_MAX;
    for (k = size - 1; k >= 0; --k)
    {
        extra = extra - 1 - lengths[k];
```

```
            if (extra < 0)
                break;
            if (min_sum[k] < min_sum[size])
            {
                min_sum[size] = min_sum[k];
                line_start_index[size - 1] = k;
            }
        }
        // output
        k = size;
        while (k != 0)
        {
            k = line_start_index[k - 1];
            result.push_front(k);
        }
        return result;
    }
```

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

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

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

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

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

```
enum Operation { COPY, REPLACE, DELETE, INSERT, TWIDDLE, KILL };

// i and j are index of edit_distance and operations,
```

```
// instead of source and target
inline void CheckCopyReplaceInsertDelete(const std::string& source,
    const std::string& target, const std::vector<int>& cost,
    std::vector< std::vector<int> >& edit_distance,
    std::vector< std::vector<Operation> >& operations,
    int i, int j)
{
    int l_cost;
    if (source[i - 1] == target[j - 1])
    {
        l_cost = edit_distance[i - 1][j - 1] + cost[COPY];
        if (l_cost < edit_distance[i][j])
        {
            edit_distance[i][j] = l_cost;
            operations[i][j] = COPY;
        }
    }
    if (source[i - 1] != target[j - 1])
    {
        l_cost = edit_distance[i - 1][j - 1] + cost[REPLACE];
        if (l_cost < edit_distance[i][j])
        {
            edit_distance[i][j] = l_cost;
            operations[i][j] = REPLACE;
        }
    }
    l_cost = edit_distance[i][j - 1] + cost[INSERT];
    if (l_cost < edit_distance[i][j])
    {
        edit_distance[i][j] = l_cost;
        operations[i][j] = INSERT;
    }
    l_cost = edit_distance[i - 1][j] + cost[DELETE];
    if (l_cost < edit_distance[i][j])
    {
        edit_distance[i][j] = l_cost;
        operations[i][j] = DELETE;
    }
}
```

```
// O(mn)
std::list<Operation> EditDistance(const std::string& source,
    const std::string& target, const std::vector<int>& cost)
{
    int m, n, i, j, kill_i_pos;
    int l_cost;
    m = (int)(source.size());
    n = (int)(target.size());
    std::list<Operation> result;
    /**
     * edit_distance[i + 1][j + 1] is the edit distance
     * from source[0 ... i] to target[0 ... j]
     */
    std::vector< std::vector<int> > edit_distance(m + 1,
        std::vector<int>(n + 1));
    /**
     * operations[i + 1][j + 1] is the operation
     * which incremented offset of source and target to i and j
     */
    std::vector< std::vector<Operation> > operations(m + 1,
        std::vector<Operation>(n + 1));
    // compute
    edit_distance[0][0] = 0;
    for (i = 1; i <= m; ++i)// j = 0
    {
        edit_distance[i][0] = edit_distance[i - 1][0] + cost[DELETE];
        operations[i][0] = DELETE;
    }
    for (j = 1; j <= n; ++j)// i = 0
    {
        edit_distance[0][j] = edit_distance[0][j - 1] + cost[INSERT];
        operations[0][j] = INSERT;
    }
    for (i = 1; i <= m; ++i)// j = 1
    {
        edit_distance[i][1] = INT_MAX;
        CheckCopyReplaceInsertDelete(source, target, cost,
            edit_distance, operations, i, 1);
    }
    for (j = 2; j <= n; ++j)// i = 1
```

```
    {
        edit_distance[1][j] = INT_MAX;
        CheckCopyReplaceInsertDelete(source, target, cost,
            edit_distance, operations, 1, j);
    }
    for (i = 2; i <= m; ++i)
    {
        for (j = 2; j <= n; ++j)
        {
            edit_distance[i][j] = INT_MAX;
            CheckCopyReplaceInsertDelete(source, target, cost,
                edit_distance, operations, i, j);
            if (source[i - 2] == target[j - 1] &&
                source[i - 1] == target[j - 2])
            {
                l_cost = edit_distance[i - 2][j - 2] + cost[TWIDDLE];
                if (l_cost < edit_distance[i][j])
                {
                    edit_distance[i][j] = l_cost;
                    operations[i][j] = TWIDDLE;
                }
            }
        }
    }
    for (i = 0; i < m; ++i)
    {
        l_cost = edit_distance[i][n] + cost[KILL];
        if ( l_cost < edit_distance[m][n])
        {
            edit_distance[m][n] = l_cost;
            operations[m][n] = KILL;
            kill_i_pos = i;
        }
    }
    // output
    i = m;
    j = n;
    while (i != 0 || j != 0)
    {
        result.push_front(operations[i][j]);
```

```
        switch (operations[i][j])
        {
        case COPY:
            --i;
            --j;
            break;
        case REPLACE:
            --i;
            --j;
            break;
        case DELETE:
            --i;
            break;
        case INSERT:
            --j;
            break;
        case TWIDDLE:
            i -= 2;
            j -= 2;
            break;
        case KILL:
            i = kill_i_pos;
            break;
        }
    }
    return result;
}
```

(b)

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

cost(copy) $= -1$

cost(replace) $= +1$

cost(delete) $= +2$

cost(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 $\Theta(n)$:

```
/* declarations of function are omitted */

struct Node
{
    Node *left_child = nullptr;
    Node *right_sibling = nullptr;
    std::string name;
    int conviviality_rating;
    int dp_attend_subtree_r_sum = INT_MIN;
    int dp_not_attend_subtree_r_sum = INT_MIN;
    bool dp_attend_if_parent_not_attend;
};

void DpAuxRootAttend(Node *root)
{
    Node *child;
    root->dp_attend_subtree_r_sum = root->conviviality_rating;
    for (child = root->left_child; child; child = child->right_sibling)
    {
        if (child->dp_not_attend_subtree_r_sum < 0)
            DpAuxRootNotAttend(child);
        root->dp_attend_subtree_r_sum += child->dp_not_attend_subtree_r_sum;
    }
}

void DpAuxRootNotAttend(Node *root)
{
    Node *child;
    root->dp_not_attend_subtree_r_sum = 0;
    for (child = root->left_child; child; child = child->right_sibling)
    {
        if (child->dp_attend_subtree_r_sum < 0)
            DpAuxRootAttend(child);
```

```
        if (child->dp_not_attend_subtree_r_sum < 0)
            DpAuxRootNotAttend(child);
        if (child->dp_attend_subtree_r_sum > child->dp_not_attend_subtree_r_sum)
        {
            root->dp_not_attend_subtree_r_sum += child->dp_attend_subtree_r_sum;
            child->dp_attend_if_parent_not_attend = true;
        }
        else
        {
            root->dp_not_attend_subtree_r_sum += child->dp_not_attend_subtree_r_sum;
            child->dp_attend_if_parent_not_attend = false;
        }
    }
}

void OutputListRootAttend(Node *root, std::list<Node*>& result)
{
    Node *child;
    result.push_back(root);
    for (child = root->left_child; child; child = child->right_sibling)
    {
        OutputListRootNotAttend(child, result);
    }
}

void OutputListRootNotAttend(Node *root, std::list<Node*>& result)
{
    Node *child;
    for (child = root->left_child; child; child = child->right_sibling)
    {
        if (child->dp_attend_if_parent_not_attend)
            OutputListRootAttend(child, result);
        else
            OutputListRootNotAttend(child, result);
    }
}

std::list<Node*> PlanningCompanyParty(Node *root)
{
    std::list<Node*> result;
```

```
    DpAuxRootAttend(root);
    DpAuxRootNotAttend(root);
    if (root->dp_attend_subtree_r_sum > root->dp_not_attend_subtree_r_sum)
        OutputListRootAttend(root, result);
    else
        OutputListRootNotAttend(root, result);
    return result;
}
```

15-7

Note that path in this book is refered "walk" by some other authors. Path is not defaulted to simple path. This says there might exists 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 $\langle \sigma_1, \sigma_2, \cdots, \sigma_i \rangle$ 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:

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

In another way, we denote $d[u, i] = 1$ where $0 \leq i \leq k$ if there exists path that has $\langle \sigma_{i+1}, \cdots, \sigma_k \rangle$ 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:

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

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

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

```
        return true;
    if (dp_checked[v][sigma_index])// dp
        return false;
    dp_checked[v][sigma_index] = true;
    for (const AdjListElement& edge : graph[v].adj_list)
    {
        if (sigma[sigma_index] == edge.sound)
        {
            if (ViterbiAux(graph, edge.vertex_index, sigma,
                sigma_index + 1, result, dp_checked))
            {
                result.push_front(edge.vertex_index);
                return true;
            }
        }
    }
    return false;
}

// index of sigma start from 0
// return empty list if NO-SUCH-PATH
std::list<int> Viterbi(Graph& graph, int v0,
    const std::vector<int>& sigma)
{
    std::list<int> result;
    std::vector< std::vector<bool> > dp_checked(graph.size(),
        std::vector<bool>(sigma.size(), false));
    if (ViterbiAux(graph, v0, sigma, 0, result, dp_checked))
        result.push_front(v0);
    return result;
}
```

(b)

Denote $c[v, i]$ where $0 \leq i \leq k$ as the maximum probability of the path that has $\langle \sigma_1, \sigma_2, \cdots, \sigma_i \rangle$ 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:

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

In another way, we denote $d[u,i]$ where $0 \leq i \leq k$ as the maximum probability of the path that has $\langle \sigma_{i+1}, \cdots, \sigma_k \rangle$ 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:

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

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

```
double ViterbiProbabilityAux(Graph& graph, int v,
    const std::vector<Sound>& sigma,
    int sigma_index, std::vector< std::vector<int> >& dp_path,
    std::vector< std::vector<double> >& dp_probability)
{
    double l_probability;
    if (sigma_index >= sigma.size())
        return 1;
    if (dp_probability[v][sigma_index] >= 0)// dp
        return dp_probability[v][sigma_index];
    dp_probability[v][sigma_index] = 0;
    for (const AdjListElement& edge : graph[v].adj_list)
    {
        if (sigma[sigma_index] == edge.sound)
        {
            l_probability = ViterbiProbabilityAux(graph, edge.vertex_index,
                sigma, sigma_index + 1, dp_path, dp_probability)
                * edge.probability;
            if (l_probability > dp_probability[v][sigma_index])
            {
                dp_probability[v][sigma_index] = l_probability;
                dp_path[v][sigma_index] = edge.vertex_index;
            }
        }
    }
```

```
    return dp_probability[v][sigma_index];
}

// index of sigma start from 0
// return empty list if NO-SUCH-PATH
std::list<int> ViterbiProbability(Graph& graph, int v0,
    const std::vector<int>& sigma)
{
    int l_vertex;
    size_t vertex_size, sigma_size, i;
    std::list<int> result;
    vertex_size = graph.size();
    sigma_size = sigma.size();
    std::vector< std::vector<int> > dp_path(vertex_size,
        std::vector<int>(sigma_size));
    std::vector< std::vector<double> > dp_probability(vertex_size,
        std::vector<double>(sigma_size, -1));
    if (ViterbiProbabilityAux(graph, v0, sigma, 0, dp_path,
        dp_probability) > 0)
    {
        l_vertex = v0;
        result.push_back(l_vertex);
        for (i = 0; i < sigma_size; ++i)
        {
            l_vertex = dp_path[l_vertex][i];
            result.push_back(l_vertex);
        }
    }
    return result;
}
```

15-8

(a)

We form the seam from the top row to the bottom row. Assuming that $n > 1$, there are at least two chioces 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

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

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

```
std::vector<int> LowestDisruptionSeam(const std::vector< std::vector<int> >& disruption)
{
    int m, n, i, j;
    m = disruption.size();
    n = disruption[0].size();
    std::vector< std::vector<int> > dp_disruption_sum(m, std::vector<int>(n)),
        dp_col_of_next_row(m - 1, std::vector<int>(n));
    std::vector<int> result;
    result.reserve(m);
    // compute
    for (j = 0; j < n; ++j)
    {
        dp_disruption_sum[m - 1][j] = disruption[m - 1][n];
    }
    for (i = m - 2; i >= 0; --i)
    {
        if (dp_disruption_sum[i + 1][0] < dp_disruption_sum[i + 1][1])
        {
            dp_disruption_sum[i][0] = dp_disruption_sum[i + 1][0] + disruption[i][0];
            dp_col_of_next_row[i][0] = 0;
        }
        else
        {
            dp_disruption_sum[i][0] = dp_disruption_sum[i + 1][1] + disruption[i][0];
            dp_col_of_next_row[i][0] = 1;
        }
        for (j = 1; j < n - 1; ++j)
        {
            if (dp_disruption_sum[i + 1][j - 1] < dp_disruption_sum[i + 1][j])
```

```
            {
                dp_disruption_sum[i][j] = dp_disruption_sum[i + 1][j - 1];
                dp_col_of_next_row[i][j] = j - 1;
            }
            else
            {
                dp_disruption_sum[i][j] = dp_disruption_sum[i + 1][j];
                dp_col_of_next_row[i][j] = j;
            }
            if (dp_disruption_sum[i + 1][j + 1] < dp_disruption_sum[i][j])
            {
                dp_disruption_sum[i][j] = dp_disruption_sum[i + 1][j + 1];
                dp_col_of_next_row[i][j] = j + 1;
            }
            dp_disruption_sum[i][j] += disruption[i][j];
        }
        if (dp_disruption_sum[i + 1][n - 2] < dp_disruption_sum[i + 1][n - 1])
        {
            dp_disruption_sum[i][n - 1] = dp_disruption_sum[i + 1][n - 2] +
                disruption[i][n - 1];
            dp_col_of_next_row[i][n - 1] = n - 2;
        }
        else
        {
            dp_disruption_sum[i][n - 1] = dp_disruption_sum[i + 1][n - 1] +
                disruption[i][n - 1];
            dp_col_of_next_row[i][n - 1] = n - 1;
        }
    }
    // output
    j = std::max_element(dp_disruption_sum.begin(), dp_disruption_sum.end()) -
        dp_disruption_sum.begin();
    result.push_back(j);
    for (i = 0; i < m - 1; ++i)
    {
        j = dp_col_of_next_row[0][j];
        result.push_back(j);
    }
    return result;
}
```

15-9

Denote $c[i,j]$ where $1 \le i \le j \le n$ 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

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

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 $0 \le i < j \le m+1$ 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

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

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

```
void OutputSequence(const std::vector<int>& breaking_point,
    const std::vector< std::vector<int> >& dp_break,
    int i, int j, std::vector<int>& result)
{
    int x;
    if (j - i == 1)
        return;
    x = dp_break[i][j];
    result.push_back(breaking_point[x]);
    OutputSequence(breaking_point, dp_break, i, x, result);
    OutputSequence(breaking_point, dp_break, x, j, result);
}

std::pair<int, std::vector<int> > BreakingString(std::vector<int>& breaking_point, int n)
{
    int m, l, i, j, x, l_cost;
    m = (int)(breaking_point.size());
    std::vector< std::vector<int> > dp_cost(m + 2, std::vector<int>(m + 2)),
        dp_break(m + 2, std::vector<int>(m + 2));
    std::vector<int> result;
    result.reserve(m);
    breaking_point.push_back(breaking_point[0]);
    breaking_point[0] = 0;
    breaking_point.push_back(n);
```

```
    Quicksort(breaking_point, 1, m);
    for (i = 0; i <= m; ++i)// l = 1
    {
        dp_cost[i][i + 1] = 0;
    }
    for (l = 2; l <= m + 2; ++l)
    {
        for (i = 0; i <= m + 1 - l; ++i)
        {
            j = i + l;
            dp_cost[i][j] = INT_MAX;
            for (x = i + 1; x < j; ++x)
            {
                l_cost = dp_cost[i][x] + dp_cost[x][j];
                if (l_cost < dp_cost[i][j])
                {
                    dp_cost[i][j] = l_cost;
                    dp_break[i][j] = x;
                }
            }
            dp_cost[i][j] += breaking_point[j] - breaking_point[i];
        }
    }
    // output
    OutputSequence(breaking_point, dp_break, 0, m + 1, result);
    return std::make_pair(dp_cost[0][m + 1], result);
}
```

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 exists 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 inverstment.

For each year such that we shift money into k inverstments 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 $g_1, g_2, \cdots, g_k$ into

$i_1, i_2, \cdots, i_k$ 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. $r_{i_m j} = \max(r_{i_1 j}, r_{i_2 j}, \cdots, r_{i_n j})$. 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 $u \leq v$. Contradition $\square$

(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 $s_{10} = \langle i_1, i_2, \cdots, i_9, i_{10} \rangle$; i.e. year k put all money in single investment i_k. Consider a year t such that $1 < t < 10$. Let strategy $s_t = \langle i_1, i_2, \cdots, i_{t-1}, i_t \rangle$. 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: $s'_t = \langle i'_1, i'_2, \cdots, i'_{t-1}, i_t \rangle$. We can utilize "cut-and-paste" technique to replace $i_1, i_2, \cdots, i_{t-1}$ with $i'_1, i'_2, \cdots, i'_{t-1}$, and let $s'_{10} = \langle i'_1, i'_2, \cdots, i'_{t-1}, i_t \cdots, i_{10} \rangle$. The strategy s'_{10} result a greater amount of money after 10 years than the strategy s_{10}. Contradition. $\square$

(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:

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

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

```
std::pair<double, std::vector<int> > InvestmentStrategy
    (const std::vector< std::vector<double> >& rate, int f_1, int f_2)
{
    int i, j, k, n;
    double l_money;
    n = (int)(rate.size());
    std::vector< std::vector<double> > dp_money(n, std::vector<double>(10));
    std::vector< std::vector<int> > dp_investment(n, std::vector<int>(10));
    std::vector<int> result(10);
    // compute
    for (i = 0; i < n; ++i)// j == 0
    {
        dp_money[i][0] = rate[i][0] * 10000;
    }
```

```
    for (j = 1; j < 10; ++j)
    {
        for (i = 0; i < n; ++i)
        {
            dp_money[i][j] = (dp_money[i][j - 1] - f_1) * rate[i][j];
            dp_investment[i][j] = i;
            for (k = 0; k < n; ++k)
            {
                l_money = (dp_money[k][j - 1] - f_2) * rate[i][j];
                if (l_money > dp_money[i][j])
                {
                    dp_money[i][j] = l_money;
                    dp_investment[i][j] = k;
                }
            }
        }
    }
    // output
    l_money = dp_money[0][9];
    result[9] = 0;
    for (i = 1; i < n; ++i)
    {
        if (dp_money[i][9] > l_money)
        {
            l_money = dp_money[i][9];
            result[9] = i;
        }
    }
    for (j = 9; j > 0; --j)
    {
        result[j - 1] = dp_investment[result[j]][j];
    }
    return std::make_pair(l_money, std::move(result));
}
```

(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:

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

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

```
std::pair<int, std::vector<int> > InventoryPlanning
    (const std::vector<int>& demands, int m, int c,
    const std::vector<int>& holding_cost)
{
    int j, k, l_cost;
    size_t n, i;
    n = demands.size();
    std::vector<int> sum_demands(n), result(n);
    // compute
    sum_demands[0] = demands[0];
    for (i = 1; i < n; ++i)
    {
        sum_demands[i] += sum_demands[i - 1] + demands[i];
    }
    std::vector< std::vector<int> > dp_cost(n, std::vector<int>(sum_demands[n - 1])),
        dp_plan(n, std::vector<int>(sum_demands[n - 1]));
    for (j = sum_demands[0]; j <= sum_demands[n - 1]; ++j)
    {
        dp_cost[0][j] = holding_cost[j - sum_demands[0]] +
            c * std::max(j - m, 0);
    }
    for (i = 1; i < n; ++i)
    {
        for (j = sum_demands[i]; j <= sum_demands[n - 1]; ++j)
        {
            dp_cost[i][j] = INT_MAX;
            for (k = sum_demands[i - 1]; k <= j; ++k)
            {
                l_cost = dp_cost[i - 1][k] + c * std::max(j - k - m, 0);
                if (l_cost < dp_cost[i][j])
                {
```

```
                    dp_cost[i][j] = l_cost;
                    dp_plan[i][j] = k;
                }
            }
            dp_cost[i][j] += holding_cost[j - sum_demands[i]];
        }
    }
    // output
    result[n - 1] = sum_demands[n - 1];
    for (i = n - 2; i < n; --i)
    {
        result[i] = dp_plan[i + 1][result[i + 1]];
        result[i + 1] -= result[i];
    }
    return std::make_pair(dp_cost[n - 1][sum_demands[n - 1]], std::move(result));
}
```