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. $\square$

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

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

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

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

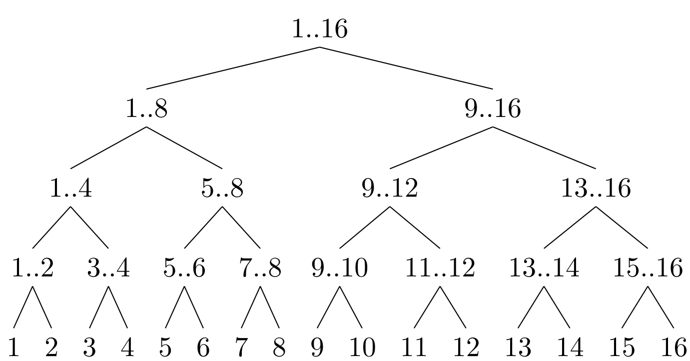
$$T(n) = 2\sum_{i=1}^{n-1} T(i) + n$$

This says we want to prove that $T(n) = o(4^n/n^{3/2})$. Since $\lim_{n\to\infty} \frac{4^n/n^{3/2}}{3.5^n} = \infty$, 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, $T(n) = o(4^n/n^{3/2})$. $\square$

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 $A_i A_{i+1} ... A_j$.
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 $p_0 p_1 p_3 = 1000$ 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 $1 \le i, j \le n$.

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