University of Chinese Academy of Sciences

计算机算法设计与分析

083500M01001H

Chap 5 课程作业解答

2022 年 10 月 22 号

Professor: 刘玉贵

学生: 周胤昌
学号: 202228018670052
学院: 网络空间安全学院
所属专业: 网络空间安全
方向: 安全协议理论与技术

Problem 1

最大子段和问题: 给定整数序列 $a_1, a_2, \cdots, a_n$, 求该序列形如 $\sum_{k=i}^{j} a_k$ 的子段和的最大值:

$$\max\left\{0, \max_{1\le i\le i\le n}\sum_{k=i}^{j} a_k\right\}$$

(1). 已知一个简单算法如下:

```cpp
int Maxsum(int n, vector<int> a, int& besti, int& bestj) {
    int sum = 0;
    for(int i = 1; i <= n; i++) {
        int suma = 0;
        for(int j = i; j <= n; j++) {
            suma += a[j];
            if(suma > sum) {
                sum = suma;
                besti = i;
                bestj = j;
            }
        }
    }
    return sum;
}
```

试分析该算法的时间复杂性;

(2). 试用分治算法解最大子段和问题, 并分析算法的时间复杂性;

(3). 试说明最大子段和问题具有最优子结构性质, 并设计一个动态规划算法求解最大子段和问题, 分析算法的时间复杂度. (提示: 可令 $b(j) = \max_{1\le i\le j\le n}\sum_{k=i}^{j} a_k, j = 1, 2, \cdots, n$)

Solution: (1). 显然第 2 层 for 循环里面的操作都是常数次的 (记为 C), 所以算法总的关键操作数为

$$\sum_{i=1}^{n}\sum_{j=i}^{n} C = C\sum_{i=1}^{n}(n-i+1) = \frac{1}{2}C\left(n^2+n\right)$$

故显然时间复杂度为 $T(n) = O(n^2)$.

(2). 采用分治算法, 则考虑: 先将数组从中间 mid 切开. 此时, 最大和的子段可能出现在左半边, 也可能出现在右半边, 也有可能横跨左右两个子数组. 所以需要返回这三种情况下所分别对应的子问题解的最大值.

当最大和的子段出现在左半边 (右半边同理) 时, 继续分中点递归直至分解到只有一个数为止;

当最大和的子段横跨 mid 左右时, 只需分别求解左子数组的最优后缀和以及右子数组的最优前缀和. 这三种情形下的最大值即为整个数组的最大子段和. 具体分治算法的 C++ 代码见后页. 从 C++ 代码可以看出最坏情形下的时间复杂度的递推式和结果分别为

$$T(n) = 2T\left(\frac{n}{2}\right) + O\left(\frac{n}{2}\right) + O\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2}\right) + O(n)$$

故根据主定理 ($\log_b(a) = \log_2(2) = 1 = d$) 可知 $T(n) = O(n\log n)$.

```cpp
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        return maxSum(nums, 0, nums.size() - 1);
    }

    int maxSum(vector<int> nums, int left, int right) {
        if(left == right) {
            return nums[left];
        }
        int mid = (left + right) >> 1;
        int LeftRes = maxSum(nums, left, mid);  //可能情况 1
        int RightRes = maxSum(nums, mid + 1, right);  //可能情况 2
        /* 可能情况 3: 最大 (和) 子段出现横跨 mid(切分点) 两边 */
        int lbs = INT_MIN, sum1 = 0;
        for(int i = mid; i >= left; i--) { //一次向前扫描来求得最大后缀和
            sum1 += nums[i];
            if(sum1 > lbs) {
                lbs = sum1;
            }
        }
        int rbs = INT_MIN, sum2 = 0;
        for(int i = mid + 1; i <= right; i++) { //一次向后扫描来求得最大前缀和
            sum2 += nums[i];
            if(sum2 > rbs) {
                rbs = sum2;
            }
        }
        return max(LeftRes, max(RightRes, lbs + rbs)); //返回三种情形下的最大值
    }
};
```

(3). 先证明此问题具有最优子结构性质: 依次考虑 $(1 \leq i \leq n)$ 以 $a[i]$ **为结尾**的最大子段和 $C[i]$, 然后在这 n 个值当中取最大值即为原问题答案. 假设以 $a[i]$ 为结尾的最大 (和) 子段为 $\{a[k], \cdots, a[i]\}$, 那么 $\{a[k], \cdots, a[i-1]\}$ 一定是以 $a[i-1]$ 为结尾的最大 (和) 子段. 否则若 $\{a[m], \cdots, a[i-1]\}$ 为以 $a[i-1]$ 为结尾的最大 (和) 子段, 那么 $\{a[m], \cdots, a[i-1], a[i]\}$ 就是以 $a[i]$ 为结尾的最大 (和) 子段, 这显然与假设相矛盾, 也就是说该优化函数是满足优化原则的 (即此问题具有最优子结构性质).

现在来推导 $C[i]$ 的递推表达式: 当 $C[i-1] \leq 0$, 说明 $C[i-1]$ 对应的子段对于整体的贡献是没有的, 所以 $C[i] \leftarrow a[i]$; 当 $C[i-1] > 0$, 说明 $C[i-1]$ 对应的子段对于整体的是有贡献的, 于是 $C[i] \leftarrow a[i] + C[i-1]$. 两种可能情况 (对应两种决策) 取最大值即可:

$$\begin{cases} C\left[i\right] = \max\left\{a\left[i\right], C\left[i-1\right] + a\left[i\right]\right\}, 2 \leq i \leq n \\ C\left[1\right] = a\left[1\right] \end{cases}$$

最后返回数组 C 中的最大值 ($\max\limits_{1 \leq i \leq n} C\left[i\right]$) 即可. 计算 $C[i]$ 的过程需要消耗 $O(n)$ 的时间, 找出数组最大值也需要 $O(n)$ 的时间 (一次遍历), 所以算法的总时间复杂度为 $T(n) = O(n)$. 并且我们可以给出后页的 C++ 代码:

```cpp
#include <algorithm>
#include <iostream>
#include <vector>
#include <limits.h>
using namespace std;

int mostvalue(vector<int>& a) { //时间复杂度为 O(n), 空间复杂度为 O(n)
    int n = a.size();
    vector<int> dp(n);
    dp[0] = a[0];
    for(int i = 1; i < n; i++) {
        dp[i] = max(dp[i - 1] + a[i], a[i]);
    }
    int index = max_element(dp.begin(), dp.end()) - dp.begin();
    return dp[index];
}

int mostvalue2(vector<int>& a) {  //时间复杂度为 O(n), 空间复杂度为 O(1)
    int n = a.size();
    int dp = a[0];
    int res = INT_MIN;
    for(int i = 1; i < n; i++) {
        dp = max(dp + a[i], a[i]);  //使用滚动数组思想来优化空间
        res = max(res, dp); //迭代式更新求得最大值
    }
    return res;
}

int main() {
    int N;
    scanf("%d", &N);
    vector<int> a(N);
    for(int i = 0; i < N; i++){
        scanf("%d", &a[i]);
    }
    //int res = mostvalue(a); //不优化空间
    int res = mostvalue2(a); //优化空间
    cout << res;

}
```

Problem 2

设 $A=\{x_1,x_2,\cdots,x_n\}$ 是 n 个不等的整数构成的序列, A 的一个单调递增子序列是指序列 $\{x_{i_1},x_{i_2},\cdots,x_{i_k}\}, i_1<i_2<\cdots i_k$ 且 $x_{i_1}<x_{i_2}<\cdots<x_{i_k}$.(子序列包含 k 个整数). 例如, $A=\{1,5,3,8,10,6,4,9\}$, 他的长度为 4 的递增子序列是: $\{1,5,8,10\},\{1,5,8,9\},\cdots$. 设计一个算法, 求 A 的最长的单调递增子序列, 分析算法的时间复杂度. 对于输入实例 $A=\{2,8,4,-4,5,9,11\}$, 给出算法的计算过程和最后的解.

Solution: 定义 $dp[i]$ 是以 nums[i] **为结尾** (且考虑前 i 个元素) 的最长单增子序列的长度.

- 如果在索引范围 $[0,i-1]$ 当中能够找到 (若干个) 下标 j 使得 nums[j] < nums[i] 成立. 根据**大小关系的传递性**可知: 以 nums[j] 为结尾的最长递增子序列中的所有元素都 < nums[i], 也就是说我们找到了符合条件的位置. 那么在这些位置 j 中找到 $dp[j]$ 的最大值, 在此基础上加 1 即可使得以 nums[i] 为结尾的单增子序列长度最大 (即将 nums[i] 接到符合条件的、且最长的 nums[j] 后边).
- 如果在索引范围 $[0,i-1]$ 当中找不到下标 j 使得 nums[j] < nums[i] 成立 (即 nums$[0,\cdots,i-1]$ 都比 nums[i] 大), 那么以 nums[i] 为结尾的单增子序列长度就只能为 1(即 nums[i] 独立作为单增子序列).

故综合上述两种情况, 我们可以写出如下转移方程 ($i\geq 1$) 和初始条件:

$$dp\left[i\right]=\begin{cases}\max\limits_{0\leq j\leq i-1} dp\left[j\right]+1, & \exists j\in[0,i-1], s.t.\,\text{nums}\left[j\right]<\text{nums}\left[i\right]\\ 1 & \forall j\in[0,i-1], s.t.\,\text{nums}\left[j\right]>\text{nums}\left[i\right]\end{cases}, dp\left[0\right]=1$$

但是上述递推方程只能给出长度, 并不能给出具体的最优解, 所以我们需要借助一个数组 m 来对解进行回溯[1](即 $m[i]$ 记录 $dp[i]$ 是由哪个下标的状态转移而来的). 而要想算出**整个数组**的最长单增子序列长度, 则需要算好所有的 $dp[i]$ 值, 再对 dp 数组进行遍历, 由此得到最长单增子序列长度和对应下标[2]. 最后使用数组 m 进行回溯以取得答案. 可以看出, 算法的空间复杂度为 $O(n)$, 而时间复杂度显然为

$$T\left(n\right)=O\left(\sum_{i=0}^{n-1}\sum_{j=0}^{i-1}1\right)=O\left(\sum_{i=0}^{n-1}i\right)=O\left(\frac{n\left(n-1\right)}{2}\right)=O\left(n^2\right)$$

对于具体实例, 计算过程如下:

$$C\left[1\right]=1; C\left[2\right]=2, k\left[2\right]=1; C\left[3\right]=2, k\left[3\right]=1; C\left[4\right]=1, k\left[4\right]=0;$$
$$C\left[5\right]=3, k\left[5\right]=3; C\left[6\right]=4, k\left[6\right]=5; C\left[7\right]=5, k\left[7\right]=6$$

显然在数组 C 中的最大值为 $C[7]=5$, 即最长递增子序列长度为 5 且追踪过程为:

$$x_7, k\left[7\right]=6\Rightarrow x_6;\ k\left[6\right]=5\Rightarrow x_5;\ k\left[5\right]=3\Rightarrow x_3;\ k\left[3\right]=1\Rightarrow x_1$$

故 $A=\{2,8,4,-4,5,9,11\}$ 的最长单调递增子序列为 $\{x_1,x_3,x_5,x_6,x_7\}=\{2,4,5,9,11\}$.

我们将上述的最优值求解过程和解的回溯过程写成 C++ 代码, 并且已完全通过**LeetCode-T300**的所有测试样例, 具体如后页所示:

[1] 对于求**具体方案**的动态规划题目, 多开一个数组来记录状态的转移情况是**最常见**的手段.

[2] 因为整个数组的最长单增子序列**不一定**以 nums[$n-1$] 为结尾! 故才需要求得 dp 数组的最大值.

```cpp
#include <algorithm>
#include <ctime>
#include <iostream>
#include <vector>
using namespace std;
vector<int> LIS(vector<int>& nums) {
    int n = nums.size();
    vector<int> dp(n, 0);
    dp[0] = 1;
    vector<int> m(n, 0); // m[0] = 0
    for (int i = 1; i <= n - 1; i++) {
        int prev = i, len = 1; //至少包含自身一个数, 因此起始长度为 1, 由自身转移而来
        for (int j = 0; j <= i - 1; j++) {
            if (nums[j] < nums[i]) { //找到满足条件的位置 j
                if (dp[j] + 1 > len) {
                    len = dp[j] + 1;
                    prev = j; //迭代式更新求得满足条件的 d[j]+1 的最大值和 dp[i] 的来源 (即 prev)
                }
            }
        }
        dp[i] = len, m[i] = prev; // m[i] 就是 nums[i] 的来源位置 (即 prev)!
    }
    int index = max_element(dp.begin(), dp.end()) - dp.begin(); //最大值对应的索引
    int MaxLen = dp[index]; //递增子序列的最大长度值
    vector<int> res; //注意整个数组的最长递增子序列是以 nums[index] 为结尾的!
    while (res.size() != MaxLen) {
        res.push_back(nums[index]);
        index = m[index]; //逐步地从后往前回溯索引, 并将对应元素写进 res 数组
    }
    reverse(res.begin(), res.end()); //reverse 一下更好看一些, 此步可以选择注释掉
    return res;
}
int main() {
    int n;
    cin >> n;             //输入数组的长度
    vector<int> nums(n); //定义数组
    cout << " 数组 nums 为:" << endl;
    for (int i = 0; i < n; i++) { //对数组初始化
        nums[i] = -1 * (n + 1) +
                  rand() % (2 * n + 2); //在 [-n - 1, n + 1] 随机产生长度为 n 的数组
        cout << nums[i] << " "; //打印该数组
    }
    cout << endl;
    clock_t startTime = clock(); //计时开始
    vector<int> res = LIS(nums);
    clock_t endTime = clock(); //计时结束
    cout << " 数组 nums 的最长单增子序列为:" << endl;
    for (int i = 0; i < res.size(); i++) {
        cout << res[i] << " ";
    }
    cout << endl;
    cout << " 算法耗时为: " << (double)(endTime - startTime) << "ms" << endl;
}
```

在解决这个问题之后, 我们需要再给出另一个解决思路与此非常类似的问题以便进行类比学习: **最大整除子集问题**: 给定一组互异的正整数集合, 找到最大子集: 使得子集元素中的每一对 (S_i, S_j) 都满足 $S_i\%S_j = 0$ 或 $S_j\%S_i = 0$. 请返回最大子集的阶数 (即子集中的元素个数). **注意**: $S_i\%S_j = 0$ 意味着 S_i 可以被 S_j 整除.

由于整除子集中的每一对值都是倍数**或**约数关系, 因此为了后续解决问题的方便, 我们不妨先将数组 nums 进行排序 (耗时为 $O(n\log n)$, (后续可知) 不影响整个算法的复杂度), 以免还要具体考虑一对值**到底是**倍数关系还是约数关系 (思考起来会很乱), 并且对 nums 排好序有助于我们利用**整除关系的传递性**来进行动态规划!

于是我们定义: $dp[i]$ 是以 nums[i] **为结尾** (且考虑前 i 个元素) 的最大整除子集的阶数 (即该子集合的元素个数).

- 如果在索引范围 $[0, i-1]$ 当中能够找到 (若干个) 下标 j 使得 nums[i]%nums[j]==0 成立. 由于 nums[j] | nums[i] 且根据**整除关系的传递性**可知: 以 nums[j] 为结尾的最大整除子集中的所有元素都能整除 nums[i], 也就是说我们找到了符合条件的位置. 那么在这些位置 j 中找到 $dp[j]$ 的最大值, 在此基础上加 1 即可使得以 nums[i] 为结尾的整除子集的长度最大 (即将 nums[i] 接到符合条件的、且最长的 nums[j] 后边).
- 如果在索引范围 $[0, i-1]$ 当中找不到下标 j 使得 nums[i]%nums[j] $\neq 0$ 成立 (即 nums[i] 不能接在位置 i 之前的任何数的后面!), 那么以 nums[i] 为结尾的最大整除子集的长度就只能为 1(即 nums[i] 独立作为最大整除子集).

故综合上述两种情况, 我们可以写出如下转移方程 ($i \geq 1$) 和初始条件:

$$dp[i] = \begin{cases} \max\limits_{0\leq j\leq i-1} dp[j]+1, & \exists j \in [0,i-1], s.t.\,\text{nums}[i]\,\%\text{nums}[j] = 0 \\ 1 & \forall j \in [0,i-1], s.t.\,\text{nums}[i]\,\%\text{nums}[j] \neq 0 \end{cases}, dp[0] = 1$$

此时我们需要借助一个数组 m 来对解进行回溯 (即 $m[i]$ 记录 $dp[i]$ 是由哪个下标的状态转移而来的). 而要想算出**整个数组**的整除子集的最大长度, 则需要算好所有的 $dp[i]$ 值, 再对 dp 数组进行遍历, 由此得到最大的整除子集长度和对应下标. 最后使用数组 m 进行回溯以取得答案. 于是我们可以给出最大整除子集算法的 C++ 代码:

```cpp
class Solution {
public:
    vector<int> largestDivisibleSubset(vector<int>& nums) {
        sort(nums.begin(), nums.end());  //先对数组进行排序, 方便后续考虑递推表达式
        int n = nums.size();
        vector<int> dp(n, 0);
        dp[0] = 1;
        vector<int> m(n, 0); // m[0] = 0
        for (int i = 1; i <= n - 1; i++) {
            int prev = i, len = 1; //至少包含自身一个数, 因此起始长度为 1, 由自身转移而来
            for (int j = 0; j <= i - 1; j++) {
                if (nums[i] % nums[j] == 0) {
                    if (dp[j] + 1 > len) {
                        len = dp[j] + 1;
                        prev = j; //迭代式更新求得满足条件的 d[j]+1 的最大值和 dp[i] 的来源
↪   (即 prev)
                    }
                }
            }
            dp[i] = len, m[i] = prev; // m[i] 就是 nums[i] 的来源位置 (即 prev)!
        }
```

```
        int index = max_element(dp.begin(), dp.end()) - dp.begin(); //最大值对应的索引
        int MaxLen = dp[index]; //整除子集的最大长度值
        vector<int> res; //注意整个数组的最大整除子集是以 nums[index] 为结尾的!
        while (res.size() != MaxLen) {
            res.push_back(nums[index]);
            index = m[index]; //逐步地从后往前回溯索引, 并将对应元素写进 res 数组
        }
        reverse(res.begin(), res.end()); //reverse 一下更好看一些, 此步可以选择注释掉
        return res;
    }
};
```

上述 C++ 代码已完全通过**LeetCode-T368**的所有测试样例. 再来分析一下该算法的时空复杂度: 排序需要 $O(n\log n)$ 的时间, 算法主体显然需要消耗 $O(n^2)$ 的时间, 而其余操作 (如 reverse 操作、求 dp 最大值以及结果写入) 均只需要 $O(n)$ 的时间, 所以综合可得算法的时间复杂度为 $O(n^2)$. 由于借助了两个长度为 n 的中间数组, 所以算法的空间复杂度为 $O(n)$. 上述两个算法之所以正确, 本质上是利用了**二元关系的可传递性**. 比如最长递增子序列是利用了**大小关系“>”的可传递性**, 而最大整除子集利用了**整除关系“|”的可传递性**. 这两段代码基本一样, 所以可以将此记住作为解题模板, 以便应付其他具有传递性的二元关系的 dp 算法题.

Problem 3

考虑下面特殊的整数线性规划问题

$$\max \sum_{i=1}^{n} c_i x_i$$
$$\sum_{i=1}^{n} a_i x_i \le b, x_i \in \{0,1,2\}, 1 \le i \le n$$

试设计一个解决此问题的动态规划算法, 并分析算法的时间复杂度.

Solution: 方法 1: 设 $y_i \in \{0,1\}, 1 \le i \le 2n$, 令 $x_i = y_i + y_{i+n}, 1 \le i \le n$, 则上述规划问题转化为

$$\max \sum_{i=1}^{2n} c_i y_i$$
$$\sum_{i=1}^{2n} a_i y_i \le b, y_i \in \{0,1\}, 1 \le i \le 2n$$

其中 $c_{i+n} = c_i, a_{i+n} = a_i, 1 \le i \le n$. 将 c_i 看作价值, a_i 看作重量, b 看作背包容量. 于是就将问题转化为了 0-1 背包问题 (物品数为 $2n$). 由于 n 件物品的 0-1 背包问题的动态规划算法的时间复杂度为 $O(2^n)$, 故此方法的时间复杂度为 $O\left(2^{2n}\right) = O(4^n)$.

方法 2: 可以看成是另一种背包问题. 即 b 为背包容量, $x_i \in \{0,1,2\}$ 为背包中可以装 0,1 或者 2 件物品, x_i 对应的价值为 c_i, 求在容量 b 一定的前提下, 背包所容纳物品的最大价值. 也就是参数完全相同的两个 0-1 背包问题, 它们同时制约于背包容量为 C 这个条件.

在设计算法时可以优先考虑 m_i, 也就是先判断背包剩下的容量能不能放进去 c_i, 若可以就再判断能否使 $p_i = 1$, 若可以那就再放入一个 c_i, 这样就间接地满足了 $x_i = m_i + p_i = 2$ 的条件. 根据单参数的

0-1 背包问题的动态规划算法, 于是可以类比写出该问题的递推公式

$$m(k,x)=\begin{cases}-\infty, & x<0\\ m(k-1,x), & 0\le x<w_k\\ \max\{m(k-1,x),m(k-1,x-w_k)+p_k\}, & w_k\le x<2w_k\\ \max\{m(k-1,x),m(k-1,x-w_k)+p_k,m(k-1,x-2w_k)+2p_k\}, & x\ge 2w_k\end{cases}$$

类似讲义中的推导过程可以得出该方法的时间复杂度为 $O(3^n)$.

Problem 4

可靠性设计: 一个系统由 n 级设备串联而成, 为了增强可靠性, 每级都可能并联了不止一台同样的设备. 假设第 i 级设备 D_i 用了 m_i 台, 该级设备的可靠性 $g_i(m_i)$, 则这个系统的可靠性是 $\Pi g_i(m_i)$. 一般来说 $g_i(m_i)$ 都是递增函数, 所以每级用的设备越多系统的可靠性越高. 但是设备都是有成本的, 假定设备 D_i 的成本是 c_i, 设计该系统允许的投资不超过 c. 那么, 该如何设计该系统 (即各级采用多少设备) 使得这个系统的可靠性最高. 试设计一个动态规划算法求解可靠性设计问题.

Solution: 问题描述为

$$\begin{aligned}&\max\prod_{i=1}^{n}g_i(m_i)\\ &\text{s.t.}\sum_{i=1}^{n}m_ic_i\le c, 1\le m_i\le 1+\left\lfloor\frac{c-\sum_{i=1}^{n}c_i}{c_n}\right\rfloor\end{aligned}$$

记 $G[k](x)$ 为第 k 级设备在可用投资为 x 时的系统可靠性最大值则有如下关系式

$$G[k](x)=\max_{1\le m_k\le\left\lfloor\frac{x}{c_k}\right\rfloor}\{g_k(m_k)\cdot G[k-1](x-c_km_k)\}$$

定义下列函数

$$G[k](x)=\begin{cases}-\infty, & x<\sum_{i=k}^{n}c_i\\ \max\limits_{1\le m_k\le\min\left\{\left\lfloor\frac{x}{c_1}\right\rfloor,\left\lfloor\frac{c}{c_k}\right\rfloor\right\}}\{g_k(m_k)\cdot G[k-1](x-c_km_k)\}, & x\ge\sum_{i=k}^{n}c_i\end{cases}$$

$$G[0](x)=\begin{cases}1, & \sum_{i=1}^{n}c_i\le x\le c\\ -\infty, & else\end{cases},\ G[1](x)=\begin{cases}-\infty, & x<\sum_{i=1}^{n}c_i\\ \max\limits_{1\le m_1\le\left\lfloor\frac{x}{c_1}\right\rfloor}\{g_1(m_1)\cdot G[0](x-c_1m_1)\}, & \sum_{i=1}^{n}c_i\le x\le c\\ \max\limits_{1\le m_1\le\left\lfloor\frac{c}{c_1}\right\rfloor}\{g_1(m_1)\cdot G[0](x-c_1m_1)\}, & x>c\end{cases}$$

初始计算 $G[0](c)$, 依次向后求解, 即可得到策略集.

Problem 5

(双机调度问题) 用两台处理机 A 和 B 处理 n 个作业. 设第 i 个作业交给机器 A 处理时所需要的时间是 a_i, 若由机器 B 来处理, 则所需要的时间是 b_i. 现在要求每个作业只能由一台机器处理, 每台机器都不能同时处理两个作业. 设计一个动态规划算法, 使得这两台机器处理完这 n 个作业的时间最短 (从任何一台机器开工到最后一台机器停工的总时间). 以下面的例子说明你的算法:

$$n = 6, (a_1, a_2, a_3, a_4, a_5, a_6) = (2, 5, 7, 10, 5, 2), (b_1, b_2, b_3, b_4, b_5, b_6) = (3, 8, 4, 11, 3, 4)$$

在完成前 k 个作业时, 设机器 A 工作了 x 时间 (注意 x 限制为正整数), 则机器 B 此时最小的工作时间是 x 的函数. 设 $F[k](x)$ 表示完成前 k 个作业时, 机器 B 最小的工作时间, 则有

$$F[k](x) = \min\{F[k-1](x) + b_k, F[k-1](x - a_k)\}$$

其中 $F[k-1](x) + b_k$ 对应的是第 k 个作业由机器 B 来处理, 此时完成前 $k-1$ 个作业时机器 A 的工作时间仍是 x, 则 B 在 $k-1$ 阶段用时为 $F[k-1](x)$; 而 $F[k-1](x-a_k)$ 对应第 k 个作业由机器 A 处理 (完成 $k-1$ 个作业, 机器 A 工作时间时 $x - a[k]$, 而 B 完成 k 阶段与完成 $k-1$ 阶段用时都为 $F[k-1](x-a_k)$). 于是完成前 k 个作业所需要的时间为 $T = \max\{x, F[k](x)\}$. 用题中的例子演示上述算法:

- 初始化第 1 个作业, 下标从 1 开始. 当处理第 1 个作业时, $a[1] = 2, b[1] = 3$, 机器 A 所用时间的可能值范围是 $0 \le x \le a[1]$, 于是会出现以下几种情况:
  - 当 $x < 0$ 时, 设 $F[1](x) = \infty$, 则 $\max\{x, \infty\} = \infty$;
  - 当 $0 \le x < 2$ 时, $F[1](x) = 3$, 则 $\max\{x, 3\} = 3$;
  - 当 $x \ge 2$ 时, $F[1](x) = 0$, 则 $\max\{1, x\} = 2$;

  从上面的 3 种情况可以看出, 当 $x = 2$ 时, 完成第一个作业两台机器花费的最少时间为 2, 此时机器 A 花费时间 2, 机器 B 花费 0 时间. 即前 1 个作业的安排为 (A)
- 再来看第 2 个作业: 首先 x 的取值范围是 $0 \le x \le a[1] + a[2] = 7$.
  - 当 $x < 0$ 时, 设 $F[2](x) = \infty$, 则 $\max\{x, \infty\} = \infty$;
  - 当 $0 \le x < 2$ 时, $F[2](x) = \min\{F[1](x) + b_2, F[1](x - a_2)\} = \min\{3 + 8, \infty\} = 11$, 则 $\max\{x, 11\} = 11$;
  - 当 $2 \le x < 5$ 时, $F[2](x) = \min\{F[1](x) + b_2, F[1](x - a_2)\} = \min\{0 + 8, \infty\} = 8$, 则 $\max\{x, 8\} = 8$;
  - 当 $5 \le x < 7$ 时, $F[2](x) = \min\{F[1](x) + b_2, F[1](x - a_2)\} = \min\{0 + 8, 3\} = 3$, 则 $\max\{x, 3\} = x$(包含 5, 6);
  - 当 $x \ge 7$ 时, $F[2](x) = \min\{F[1](x) + b_2, F[1](x - a_2)\} = \min\{0 + 8, 0\} = 0$, 则 $\max\{x, 0\} = x$(包含 7);

  于是可以看出当 $x = 5$ 时, 完成前两个作业的两台机器所花费时间最少为 5, 此时机器 A 花费 5 时间, 机器 B 花费 3 时间, 即前 2 个作业的安排为 (B, A).
- 再来看第 3 个作业: 首先 x 的取值范围是 $0 \le x \le a[1] + a[2] + a[3] = 14$.
  - 当 $x < 0$ 时, 设 $F[3](x) = \infty$, 则 $\max\{x, \infty\} = \infty$;
  - 当 $0 \le x < 2$ 时, $F[3](x) = \min\{F[2](x) + b_3, F[2](x - a_3)\} = \min\{11 + 4, \infty\} = 15$, 则 $\max\{x, 15\} = 15$;

- 当 $2 \leq x < 5$ 时,$F[3](x) = \min\{F[2](x) + b_3, F[2](x - a_3)\} = \min\{8 + 4, \infty\} = 12$, 则 $\max\{x, 12\} = 12$;
- 当 $5 \leq x < 7$ 时, $F[3](x) = \min\{F[2](x) + b_3, F[2](x - a_3)\} = \min\{3 + 4, \infty\} = 7$, 则 $\max\{x, 7\} = 7$;
- 当 $7 \leq x < 9$ 时, $F[3](x) = \min\{F[2](x) + b_3, F[2](x - a_3)\} = \min\{0 + 4, 11\} = 4$, 则 $\max\{x, 4\} = x$(包含 7, 8);
- 当 $9 \leq x < 12$ 时, $F[3](x) = \min\{F[2](x) + b_3, F[2](x - a_3)\} = \min\{0 + 4, 8\} = 4$, 则 $\max\{x, 4\} = x$(包含 9, 10, 11);
- 当 $12 \leq x < 14$ 时, $F[3](x) = \min\{F[2](x) + b_3, F[2](x - a_3)\} = \min\{0 + 4, 3\} = 3$, 则 $\max\{x, 3\} = x$(包含 12, 13);
- 当 $x \geq 14$ 时, $F[3](x) = \min\{F[2](x) + b_3, F[2](x - a_3)\} = \min\{0 + 4, 0\} = 0$, 则 $\max\{x, 0\} = x$(包含 14);

于是可以看出当 $x = 7$ 时, 完成前两个作业的两台机器所花费时间最少为 7, 此时机器 A 花费 7 时间, 机器 B 花费 7 时间 (也可以花费 4 时间). 即完成前 3 个作业有两种安排: (B, A, B)(对应 A 花费 7, B 花费 7) 和 (A, A, B)(对应 A 花费 7, B 花费 4).

- 再来看第 4 个作业: 首先 x 的取值范围是 $0 \leq x \leq a[1] + a[2] + a[3] + a[4] = 24$.
  - 当 $x < 0$ 时, 设 $F[4](x) = \infty$, 则 $\max\{x, \infty\} = \infty$;
  - 当 $0 \leq x < 2$ 时, $F[4](x) = \min\{F[3](x) + b_4, F[3](x - a_4)\} = \min\{15 + 11, \infty\} = 26$, 则 $\max\{x, 26\} = 26$;
  - 当 $2 \leq x < 5$ 时, $F[4](x) = \min\{F[3](x) + b_4, F[3](x - a_4)\} = \min\{12 + 11, \infty\} = 23$, 则 $\max\{x, 23\} = 23$;
  - 当 $5 \leq x < 7$ 时, $F[4](x) = \min\{F[3](x) + b_4, F[3](x - a_4)\} = \min\{7 + 11, \infty\} = 18$, 则 $\max\{x, 18\} = 18$;
  - 当 $7 \leq x < 9$ 时, $F[4](x) = \min\{F[3](x) + b_4, F[3](x - a_4)\} = \min\{4 + 11, \infty\} = 15$, 则 $\max\{x, 15\} = 15$;
  - 当 $9 \leq x < 10$ 时, $F[4](x) = \min\{F[3](x) + b_4, F[3](x - a_4)\} = \min\{4 + 11, \infty\} = 15$, 则 $\max\{x, 15\} = 15$;
  - 当 $10 \leq x < 12$ 时, $F[4](x) = \min\{F[3](x) + b_4, F[3](x - a_4)\} = \min\{4 + 11, 15\} = 15$, 则 $\max\{x, 15\} = 15$;
  - 当 $12 \leq x < 14$ 时, $F[4](x) = \min\{F[3](x) + b_4, F[3](x - a_4)\} = \min\{3 + 11, 12\} = 12$, 则 $\max\{x, 12\} = x$(包含 12, 13);
  - 当 $14 \leq x < 15$ 时, $F[4](x) = \min\{F[3](x) + b_4, F[3](x - a_4)\} = \min\{0 + 11, 12\} = 11$, 则 $\max\{x, 11\} = x$(包含 14);
  - 当 $15 \leq x < 17$ 时, $F[4](x) = \min\{F[3](x) + b_4, F[3](x - a_4)\} = \min\{0 + 11, 7\} = 7$, 则 $\max\{x, 7\} = x$(包含 15, 16);
  - 当 $17 \leq x < 19$ 时, $F[4](x) = \min\{F[3](x) + b_4, F[3](x - a_4)\} = \min\{0 + 11, 4\} = 4$, 则 $\max\{x, 4\} = x$(包含 17, 18);
  - 当 $19 \leq x < 22$ 时, $F[4](x) = \min\{F[3](x) + b_4, F[3](x - a_4)\} = \min\{0 + 11, 4\} = 4$, 则 $\max\{x, 4\} = x$(包含 19, 20, 21);
  - 当 $22 \leq x < 24$ 时, $F[4](x) = \min\{F[3](x) + b_4, F[3](x - a_4)\} = \min\{0 + 11, 3\} = 3$, 则 $\max\{x, 3\} = x$(包含 22, 23);

  - 当 $24 \leq x$ 时, $F[4](x) = \min\{F[3](x) + b_4, F[3](x - a_4)\} = \min\{0 + 11, 0\} = 0$, 则 $\max\{x, 0\} = x$(包含 24);

  于是可以看出当 $x = 12$ 时, 完成前两个作业的两台机器所花费时间最少为 12, 此时机器 A 花费 12 时间, 机器 B 花费 12 时间. 即完成前 3 个作业的最优安排为 (A, B, B, A).

- 再来看第 5 个作业: 首先 x 的取值范围是 $0 \leq x \leq a[1] + a[2] + a[3] + a[4] + a[5] = 29$.
  - 当 $x < 0$ 时, 设 $F[5](x) = \infty$, 则 $\max\{x, \infty\} = \infty$;
  - 当 $0 \leq x < 2$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{26 + 3, \infty\} = 29$, 则 $\max\{x, 29\} = 29$;
  - 当 $2 \leq x < 5$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{23 + 3, \infty\} = 26$, 则 $\max\{x, 26\} = 26$;
  - 当 $5 \leq x < 7$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{18 + 3, 26\} = 21$, 则 $\max\{x, 21\} = 21$;
  - 当 $7 \leq x < 9$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{15 + 3, 23\} = 18$, 则 $\max\{x, 18\} = 18$;
  - 当 $9 \leq x < 10$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{15 + 3, 23\} = 18$, 则 $\max\{x, 18\} = 18$;
  - 当 $10 \leq x < 12$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{15 + 3, 18\} = 18$, 则 $\max\{x, 18\} = 18$;
  - 当 $12 \leq x < 14$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{12 + 3, 15\} = 15$, 则 $\max\{x, 15\} = 15$;
  - 当 $14 \leq x < 15$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{11 + 3, 15\} = 14$, 则 $\max\{x, 14\} = x$(包含 14);
  - 当 $15 \leq x < 17$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{7 + 3, 15\} = 10$, 则 $\max\{x, 10\} = x$(包含 15, 16);
  - 当 $17 \leq x < 19$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{4 + 3, 12\} = 7$, 则 $\max\{x, 7\} = x$(包含 17, 18);
  - 当 $19 \leq x < 20$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{4 + 3, 11\} = 7$, 则 $\max\{x, 7\} = x$(包含 19);
  - 当 $20 \leq x < 22$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{4 + 3, 7\} = 7$, 则 $\max\{x, 7\} = x$(包含 20, 21);
  - 当 $22 \leq x < 24$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{3 + 3, 4\} = 4$, 则 $\max\{x, 4\} = x$(包含 22, 23);
  - 当 $24 \leq x < 27$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{0 + 3, 4\} = 3$, 则 $\max\{x, 3\} = x$(包含 24, 25, 26);
  - 当 $27 \leq x < 29$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{0 + 3, 3\} = 3$, 则 $\max\{x, 3\} = x$(包含 27, 28);
  - 当 $x \geq 29$ 时, $F[5](x) = \min\{F[4](x) + b_5, F[4](x - a_5)\} = \min\{0 + 3, 0\} = 0$, 则 $\max\{x, 0\} = x$(包含 29);

  于是可以看出当 $x = 14$ 时, 完成前两个作业的两台机器所花费时间最少为 14, 此时机器 A 花费 14 时间, 机器 B 花费 14 时间. 即完成前 5 个作业的最优安排为 (A, A, A, B, B).

- 再来看第 6 个作业: 首先 x 的取值范围是 $0 \le x \le a[1]+a[2]+a[3]+a[4]+a[5]+a[6] = 31$.
  - 当 $x < 0$ 时, 设 $F[6](x) = \infty$, 则 $\max\{x, \infty\} = \infty$;
  - 当 $0 \le x < 2$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{29 + 4, \infty\} = 33$, 则 $\max\{x, 33\} = 33$;
  - 当 $2 \le x < 4$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{26 + 4, 29\} = 29$, 则 $\max\{x, 29\} = 29$;
  - 当 $4 \le x < 5$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{26 + 4, 26\} = 26$, 则 $\max\{x, 26\} = 26$;
  - 当 $5 \le x < 7$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{21 + 4, 26\} = 25$, 则 $\max\{x, 25\} = 25$;
  - 当 $7 \le x < 9$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{18 + 4, 21\} = 21$, 则 $\max\{x, 21\} = 21$;
  - 当 $9 \le x < 10$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{18 + 4, 18\} = 18$, 则 $\max\{x, 18\} = 18$;
  - 当 $10 \le x < 11$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{18 + 4, 18\} = 18$, 则 $\max\{x, 18\} = 18$;
  - 当 $11 \le x < 12$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{18 + 4, 18\} = 18$, 则 $\max\{x, 18\} = 18$;
  - 当 $12 \le x < 14$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{15 + 4, 18\} = 18$, 则 $\max\{x, 18\} = 18$;
  - 当 $14 \le x < 15$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{14 + 4, 18\} = 18$, 则 $\max\{x, 18\} = 18$;
  - 当 $15 \le x < 16$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{10 + 4, 15\} = 14$, 则 $\max\{x, 14\} = x$(包含 15);
  - 当 $16 \le x < 17$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{10 + 4, 14\} = 14$, 则 $\max\{x, 14\} = x$(包含 16);
  - 当 $17 \le x < 19$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{7 + 4, 10\} = 10$, 则 $\max\{x, 10\} = x$(包含 17, 18);
  - 当 $19 \le x < 20$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{7 + 4, 7\} = 7$, 则 $\max\{x, 7\} = x$(包含 19);
  - 当 $20 \le x < 21$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{7 + 4, 7\} = 7$, 则 $\max\{x, 7\} = x$(包含 20);
  - 当 $21 \le x < 22$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{7 + 4, 7\} = 7$, 则 $\max\{x, 7\} = x$(包含 21);
  - 当 $22 \le x < 24$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{4 + 4, 7\} = 7$, 则 $\max\{x, 7\} = x$(包含 22, 23);
  - 当 $24 \le x < 26$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{3 + 4, 4\} = 4$, 则 $\max\{x, 4\} = x$(包含 24, 25);
  - 当 $26 \le x < 27$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{3 + 4, 3\} = 3$, 则 $\max\{x, 3\} = x$(包含 26);

- 当 $27 \leq x < 29$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{3 + 4, 3\} = 3$, 则 $\max\{x, 3\} = x$(包含 27, 28);
- 当 $29 \leq x < 31$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{0 + 4, 3\} = 3$, 则 $\max\{x, 3\} = x$(包含 29, 30);
- 当 $31 \leq x$ 时, $F[6](x) = \min\{F[5](x) + b_6, F[5](x - a_6)\} = \min\{0 + 4, 0\} = 0$, 则 $\max\{x, 0\} = x$(包含 31);

于是可以看出当 $x = 15$ 时, 完成前两个作业的两台机器所花费时间最少为 15, 此时机器 A 花费 15 时间, 机器 B 花费 14 时间. 即完成前 6 个作业的最优安排 (只是其中一种安排方案) 为 (B, A, B, A, B, B), 并且算法的时间复杂度为 $O(n)$.

Problem 6

有 n 项作业的集合 $J = \{1, 2, \cdots, n\}$, 每项作业 i 有加工时间 $t(i) \in \mathbb{Z}^+, t(1) \leq t(2) \leq \cdots \leq t(n)$, 效益值 $v(i)$, 任务的结束时间 $D \in \mathbb{Z}^+$, 其中 $\mathbb{Z}^+$ 表示正整数集合. 一个可行调度是对 J 的子集 A 中任务的一种安排, 对于 $i \in A$, $f(i)$ 是开始时间, 且满足下述条件:

$$\begin{cases} f(i) + t(i) \leq f(j) \text{或} f(j) + t(j) \leq f(i), \text{其中} j \neq i \text{且} i, j \in A \\ \sum_{k \in A} t(k) \leq D \end{cases}$$

设机器从 0 时刻开始启动, 只要有作业就不闲置, 求具有最大总效益的调度. 给出算法并分析其时间复杂度.

Solution: 与 0-1 背包问题相类似, 使用 DP 算法, 令 $N_j(d)$ 表示考虑作业集 $\{1, 2, \cdots, j\}$、结束时间为 d 的最优调度的效益, 那么有递推方程

$$N_j(d) = \begin{cases} \max\{N_{j-1}(d), N_{j-1}(d - t(j)) + v_j\}, & d \geq t(j) \\ N_{j-1}(d), & d < t(j) \end{cases}$$

并且边界 (初始) 条件为

$$N_1(d) = \begin{cases} v_1, & d \geq t(1) \\ 0, & d < t(1) \end{cases}, \quad N_j(0) = 0, \quad N_j(d) = -\infty \left(\text{其中} d < 0\right)$$

自底向上计算, 存储使用备忘录 (以存代算), 可以使用标记函数 $B(j)$ 记录使得 $N_j(d)$ 达到最大时是否

$$N_{j-1}(d - t(j)) + v_j > N_{j-1}(d)$$

如果是, 则 $B(j) = j$; 否则 $B(j) = B(j - 1)$.(换句话说, 如果装了作业 j, 那么就追踪其下标; 否则就不追踪更新)

伪代码如后页算法1中所示, 由此我们可以分析出时间复杂度: 得到最大效益 $N[n, D]$ 后, 通过对 $B[n, D]$ 的追踪就可以得到问题的解, 算法的主要工作在于第 7 行到第 16 行的 for 循环, 需要执行 $O(nD)$ 次, 循环体内的工作量是常数时间, 因此算法的总时间复杂度为 $O(nD)$. 显然该算法是**伪多项式时间**的算法[3].

[3]问题就在于如果 D 过大, 即 D 的 2 进制表示会很长, 且 $O(n \cdot D) = O\left(n \cdot 2^{\log(D)}\right) = O\left(n \cdot 2^{\text{输入长度}}\right)$, 是与输入长度相关的指数表达式, 这种复杂度形式的算法称之为**伪多项式时间算法**.

Algorithm 1 Homework 算法

Input: 加工时间 $t[1,\cdots,n]$, 效益 $v[1,\cdots,n]$, 结束时间 D
Output: 最优效益 $N[i,j]$, 标记函数 $B[i,j], i=1,2,\cdots,n, j=1,2,\cdots,D$

```
for d = 1; d ≤ t[1] − 1; d++ do
    N[1, d] ← 0, B[1] ← 0;
end for
for d = t[1]; d ≤ D; d++ do
    N[1, d] ← v[1], B[1] ← 1;
end for
for j = 2; j ≤ n; j++ do
    for d = 1; d ≤ D; d++ do
        N[j, d] ← N[j − 1, d];
        B[j, d] ← B[j − 1, d];
        if d ≥ t[j] && N[j − 1, d − t[j]] + v[j] > N[j − 1, d] then
            N[j, d] ← N[j − 1, d − t[j]] + v[j];
            B[j, d] ← j;
        end if
    end for
end for
end {Homework}
```

Problem 7

设 A 是顶点为 $1,2,\cdots,n$ 的凸多边形, 可以用不在内部相交的 $n-3$ 条对角线将 A 划分成三角形, 下图1中就是 5 边形的所有划分方案. 假设凸 n 边形的边及对角线的长度 d_{ij} 都是给定的正整数, 其中 $1 \le i < j \le n$. 划分后三角形 ijk 的权值等于其周长, 求具有最小权值的划分方案. 设计一个动态规划算法求解该问题, 并说明其时间复杂度 (提示: 参考矩阵连乘问题).

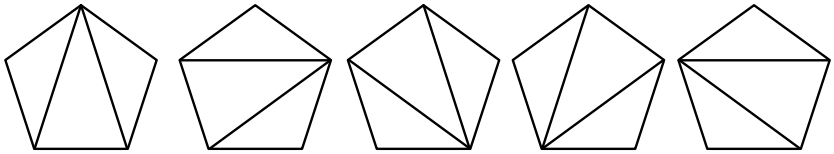

图 1: 5 边形的划分方案

如下图2所示, n 边形的顶点是 $1,2,\cdots,n$. 顶点 $i-1,i,\cdots,j$ 构成的凸多边形记作 $A[i,j]$, 于是原始问题就是 $A[2,n]$.

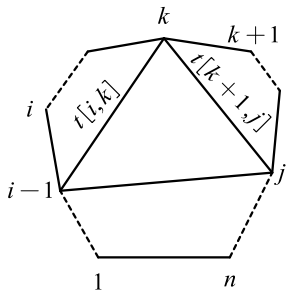

图 2: 子问题归约图

考虑子问题 $A[i,j]$ 的划分, 假设它的所有划分方案中最小权值为 $t[i,j]$. 从 $i, i+1, \cdots, j-1$ 中任选顶点 k, 它与底边 $(i-1)j$ 构成一个三角形 (图2中的三角形). 这个三角形将 $A[i,j]$ 划分成两个凸多边形: $A[i,k]$ 和 $A[k+1,j]$, 从而产生了两个子问题. 这两个凸多边形的划分方案的最小权值分别为 $t[i,k]$ 和 $t[k+1,j]$. 根据 DP 思想, $A[i,j]$ 相对于这个顶点 k 的划分方案的最小权值是

$$t[i,k] + t[k+1,j] + d_{(i-1)k} + d_{kj} + d_{(i-1)j}$$

其中 $d_{(i-1)k} + d_{kj} + d_{(i-1)j}$ 是三角形 $(i-1)kj$ 的周长, 于是得到递推关系

$$t[i,j] = \begin{cases} 0, & i = j \\ \min\limits_{i \le k \le j-1} \left\{ t[i,k] + t[k+1,j] + d_{(i-1)k} + d_{kj} + d_{(i-1)j} \right\}, & i < j \end{cases}$$

显然这个递推关系与矩阵链乘积算法的递推式十分相似, 可以通过标记函数来得到最小权值对应顶点 k 的位置, 并且类比矩阵链乘积算法, 可知该划分算法最坏情况下的时间复杂度为 $O\left(n^3\right)$.

至此, Chap 5 的作业解答完毕.