算法设计与分析

夏锦熠

2023 年 12 月 29 日

目录

1 算法概述 **3**

1.1 算法与程序 3
1.2 算法复杂性分析 3
1.3 NP 完全性理论（略） 4

2 递归与分治策略 **5**

2.1 递归的概念 5
2.2 分治法的基本思想 7
2.3 二分搜索 7
2.4 大整数乘法 7
2.5 Strassen 矩阵乘法 8
2.6 棋盘覆盖 9
2.7 归并排序 10
2.8 快速排序 11
2.9 线性时间选择 12
2.10 最接近点对问题（略） 14
2.11 循环赛日程表 14

3 动态规划 **16**

3.1 矩阵连乘问题 16
3.2 动态规划算法的基本要素 18
3.3 最长公共子序列 19
3.4 最大子段和 21
3.5 凸多边形最优三角剖分 22
3.6 多边形游戏 24
3.7 图像压缩 26
3.8 电路布线 26
3.9 流水作业调度（略） 26
3.10 0—1 背包问题 26

3.11 最优二叉搜索树（略） 28

4 贪心算法 **29**

4.1 活动安排问题 29
4.2 贪心算法的基本要素 29
4.3 最优装载 30
4.4 哈夫曼编码 30
4.5 单源最短路径 30
4.6 最小生成树 30
4.7 多机调度问题（略） 30

Chapter 1

算法概述

Contents

1.1 算法与程序 3

1.2 算法复杂性分析 3

1.3 NP 完全性理论（略） 4

1.1 算法与程序

定义 1.1 (算法的定义). *算法是若干指令组成的有穷序列，满足以下条件：*

1. *输入 零个或多个外部提供的量；*
2. *输出 产生至少一个量；*
3. *确定性 每条指令清晰无歧义；*
4. *有穷性 指令数量有限，每条指令执行时间有限。*

注意：算法的输入可以为 0 个。

1.2 算法复杂性分析

定义 1.2 (渐进上界 O). *若函数 $f(N)$ 充分大时有上界 $g(N)$ ($f(N)$ 的阶不高于 $g(N)$ 的阶)，则记为 $f(N) = O(g(N))$。*

例 1.1 (O 的计算). $f(N) = 2N^2 + 3N\log N + 1 \leq 3N^2$, 则 $f(N) = O(N^2)$。

符号 O 运算规则如下：

- $f = O(f)$;
- $O(f) + O(g) = O(f + g) = O(\max\{f, g\})$;

- $O(f) \cdot O(g) = O(f \cdot g)$;
- 若 $f(N) = O(g(N))$，则 $O(f) + O(g) = O(g)$;
- 若 C 为正常数，则 $O(C \cdot f(N)) = O(f(N))$。

渐进确界（同阶）记作符号 Θ，渐进下界记作符号 Ω。

1.3 NP 完全性理论（略）

Chapter 2

递归与分治策略

Contents

2.1 递归的概念 5
2.2 分治法的基本思想 7
2.3 二分搜索 7
2.4 大整数乘法 7
2.5 Strassen 矩阵乘法 8
2.6 棋盘覆盖 9
2.7 归并排序 10
2.8 快速排序 11
2.9 线性时间选择 12
2.10 最接近点对问题（略） 14
2.11 循环赛日程表 14

重难点

分治法的基本思想（时间复杂度）；归并排序；快速排序。

2.1 递归的概念

函数的定义包含自身。

例 2.1 (阶乘函数)**.**

$$n! = \begin{cases} 1 & n = 0, \\ n \cdot (n-1)! & n > 0. \end{cases}$$

例 2.2 (斐波那契数列).

$$F_n = \begin{cases} 1 & n = 0, 1, \\ F_{n-1} + F_{n-2} & n > 1. \end{cases}$$

双递归函数：函数的定义以及函数的一个参数中包含函数自身。

例 2.3 (Ackermann 函数).

$$\begin{cases} A(1,0) = 2, & \\ A(0,m) = 1 & m \geq 0, \\ A(n,0) = n+2 & n \geq 2, \\ A(n,m) = A(A(n-1,m), m-1) & n, m \geq 1. \end{cases}$$

单变量 *Ackermann* 函数：$A(n) = A(n,n)$。

例 2.4 (全排列). 递归思路：对于待排列的元素集合 $R = \{r_1, r_2, \cdots r_n\}$，

1. 从 R 中取出一个元素 r_i；
2. 对剩余元素 $R - \{r_i\}$ 进行全排列。

整数划分：将正整数 n 表示为一系列正整数 $n_1, n_2, \cdots n_k (n \geq n_1 \geq \cdots \geq n_k \geq 1)$ 之和，例如 $4 = 4 = 3+1 = 2+2 = 2+1+1 = 1+1+1+1$，有 5 种不同的划分，划分数 $p(4) = 5$。

例 2.5 (划分数). 将最大加数 n_1 不大于 $m (m \geq 1)$ 的划分数记为 $q(n,m)$，则 $p(n) = q(n,n)$。递归思路如下：

1. $m = 1$ 时，$q(n,1) = 1$，任何整数只能划分为 n 个 *1* 相加；
2. $m = n$ 时，$q(n,n) = q(n,n-1) + 1$，划分为仅自身或小于自身的数相加；
3. $m \geq n$ 时，$q(n,m) = q(n,n)$，划分中实际上不可能出现比自身大的数；
4. $1 < m < n$ 时，$q(n,m) = q(n,m-1) + q(n-m,m)$，其中 $q(n,m-1)$ 是最大加数小于 m 的情况，$q(n-m,m)$ 是最大加数等于 m 的情况。

提示：$q(n-m,m)$ 从 n 中减去一个最大加数，剩余部分 $n-m$ 的划分数即为 n 的划分中最大加数等于 $m-1$ 的情况。

$$q(n,m) = \begin{cases} 1, & m = 1 \\ q(n,n-1) + 1, & m = n, \\ q(n,n), & m \geq n, \\ q(n,m-1) + q(n-m,m), & 1 < m < n. \end{cases}$$

例 2.6 (汉诺塔). 递归思路：

1. 将 $n-1$ 个盘子从 A 经 C 移到 B；
2. 将第 n 个盘子从 A 移到 C；
3. 将 $n-1$ 个盘子从 B 经 A 移到 C。

2.2 分治法的基本思想

将大问题分成几个子问题，递归求解子问题，再将子问题的解合成原问题的解。

分治法求解时间的递归方程为

$$T(n) = \begin{cases} O(1) & n = 1, \\ k \cdot T\left(\frac{n}{m}\right) + f(n) & n > 1, \end{cases} \tag{2.1}$$

其中 k 为子问题个数，$\frac{n}{m}$ 为单个子问题规模，$f(n)$ 为分解与合并子问题的时间。求解这个递归方程得到

$$T(n) = n^{\log_m k} + \sum_{j=0}^{\log_m n - 1} k^j f\left(\frac{n}{m^j}\right). \tag{2.2}$$

一般情况下有

$$T(n) = O(n^{\log_m k}), \tag{2.3}$$

但是当 $f(n)$ 的阶高于 $n^{\log_m k}$ 时，$f(n)$ 的影响就不能忽略。

2.3 二分搜索

最坏情况 $O(\log n)$。

```
1  // 在有序数组 arr[0...n-1] 中二分查找 x
2  int binary_search(int arr[], int n, int x) {
3      int l = 0, r = n - 1;
4      while (l <= r) {
5          int mid = (l + r) / 2;
6          if (arr[mid] == x)
7              return mid;
8          else if (arr[mid] < x)
9              l = mid + 1;
10         else
11             r = mid - 1;
12     }
13     return -1;
14 }
```

2.4 大整数乘法

将乘数的位数 n 视为数据规模，则乘法的时间复杂度为 $O(n^2)$。

对于 n 位大整数 X 和 Y，将其分成两部分

$$X = 2^{\frac{n}{2}}A + B,$$

$$Y = 2^{\frac{n}{2}}C + D,$$

则

$$\begin{aligned} XY &= (2^{\frac{n}{2}}A + B)(2^{\frac{n}{2}}C + D) \\ &= 2^n AC + 2^{\frac{n}{2}}(AD + BC) + BD \\ &= 2^n AC + 2^{\frac{n}{2}}((A-B)(D-C) + AC + BD) + BD. \end{aligned}$$

其中 AC, BD 都是 $\frac{n}{2}$ 位的大整数，因此可以递归求解。

表达式中共有 3 次乘法，每次递归乘数长度减为一半，故执行时间为

$$T(n) = \begin{cases} O(1) & n = 1, \\ 3T\left(\frac{n}{2}\right) + O(n) & n > 1. \end{cases} \tag{2.4}$$

求解得到 $T(n) = O(n^{\log_2 3})$。

2.5 Strassen 矩阵乘法

将 $\boldsymbol{C} = \boldsymbol{AB}$ 分解为

$$\begin{pmatrix} \boldsymbol{C}_{11} & \boldsymbol{C}_{12} \\ \boldsymbol{C}_{21} & \boldsymbol{C}_{22} \end{pmatrix} = \begin{pmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} \\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22} \end{pmatrix} \begin{pmatrix} \boldsymbol{B}_{11} & \boldsymbol{B}_{12} \\ \boldsymbol{B}_{21} & \boldsymbol{B}_{22} \end{pmatrix}.$$

正常计算

$$\begin{aligned} \boldsymbol{C}_{11} &= \boldsymbol{A}_{11}\boldsymbol{B}_{11} + \boldsymbol{A}_{12}\boldsymbol{B}_{21}, \\ \boldsymbol{C}_{12} &= \boldsymbol{A}_{11}\boldsymbol{B}_{12} + \boldsymbol{A}_{12}\boldsymbol{B}_{22}, \\ \boldsymbol{C}_{21} &= \boldsymbol{A}_{21}\boldsymbol{B}_{11} + \boldsymbol{A}_{22}\boldsymbol{B}_{21}, \\ \boldsymbol{C}_{22} &= \boldsymbol{A}_{21}\boldsymbol{B}_{12} + \boldsymbol{A}_{22}\boldsymbol{B}_{22}, \end{aligned}$$

需要 8 次乘法。

但是可以通过计算

$$\begin{aligned} \boldsymbol{M}_1 &= (\boldsymbol{A}_{11} + \boldsymbol{A}_{22})(\boldsymbol{B}_{11} + \boldsymbol{B}_{22}), \\ \boldsymbol{M}_2 &= (\boldsymbol{A}_{21} + \boldsymbol{A}_{22})\boldsymbol{B}_{11}, \\ \boldsymbol{M}_3 &= \boldsymbol{A}_{11}(\boldsymbol{B}_{12} - \boldsymbol{B}_{22}), \\ \boldsymbol{M}_4 &= \boldsymbol{A}_{22}(\boldsymbol{B}_{21} - \boldsymbol{B}_{11}), \\ \boldsymbol{M}_5 &= (\boldsymbol{A}_{11} + \boldsymbol{A}_{12})\boldsymbol{B}_{22}, \\ \boldsymbol{M}_6 &= (\boldsymbol{A}_{21} - \boldsymbol{A}_{11})(\boldsymbol{B}_{11} + \boldsymbol{B}_{12}), \\ \boldsymbol{M}_7 &= (\boldsymbol{A}_{12} - \boldsymbol{A}_{22})(\boldsymbol{B}_{21} + \boldsymbol{B}_{22}), \end{aligned}$$

Strassen 矩阵乘法思想与大整数乘法类似，都是通过减少乘法运算的次数降低复杂度，但同时会增加加法等运算的次数。

得到

$$\begin{aligned}
\boldsymbol{C}_{11} &= \boldsymbol{M}_1 + \boldsymbol{M}_4 - \boldsymbol{M}_5 + \boldsymbol{M}_7, \\
\boldsymbol{C}_{12} &= \boldsymbol{M}_3 + \boldsymbol{M}_5, \\
\boldsymbol{C}_{21} &= \boldsymbol{M}_2 + \boldsymbol{M}_4, \\
\boldsymbol{C}_{22} &= \boldsymbol{M}_1 - \boldsymbol{M}_2 + \boldsymbol{M}_3 + \boldsymbol{M}_6,
\end{aligned}$$

只需要 7 次乘法。

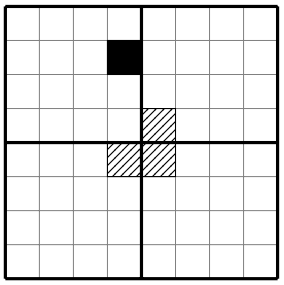
每次递归需要 7 次乘法，每次乘法矩阵维数减半，故时间复杂度为

$$T(n) = \begin{cases} O(1) & n = 1, \\ 7T\left(\frac{n}{2}\right) + O(n^2) & n > 1. \end{cases} \tag{2.5}$$

求解得到 $T(n) = O(n^{\log_2 7})$。

2.6 棋盘覆盖

特殊棋盘指 $2^k \times 2^k$ 的棋盘缺少一个方格。需要设计一种覆盖方法，用 L 型骨牌恰好覆盖特殊棋盘。



$k = 3$ 时的棋盘覆盖

用分治策略，将特殊棋盘分成 4 个 $2^{k-1} \times 2^{k-1}$ 的子区域，缺少的方格必位于其中一个子区域，因此可以用一个 L 型骨牌覆盖另外三个子区域，使这三块子区域也变成特殊棋盘，然后递归地使用这种分割。

计算该算法时间复杂度：

1. 以 $2^k \times 2^k$ 计算棋盘大小，以 k 衡量数据规模，由于每次要遍历 4 个子区域，每个子区域大小为 $2^{k-1} \times 2^{k-1}$，数据规模为 k，求解时间为

   $$T(k) = \begin{cases} O(1) & k = 1, \\ 4T(k-1) + O(1) & k > 1, \end{cases}$$

   求解得到 $T(k) = O(4^k)$；

2. 以 $n \times n$ 计算棋盘大小，以 n 衡量数据规模，由于每次要遍历 4 个子区域，每个子区域大小为 $\frac{n}{2} \times \frac{n}{2}$，数据规模为 n，求解时间为

   $$T(n) = \begin{cases} O(1) & n = 1, \\ 4T\left(\frac{n}{2}\right) + O(n^2) & n > 1, \end{cases}$$

   求解得到 $T(n) = O(n^2)$。

两种复杂度计算方法数据规模的关系为 $n = 2^k$，所以两种方法计算出的时间复杂度等价。

2.7 归并排序

将待排序数组分成两半，递归排序两个子数组，然后将两个子数组合并成一个有序数组。

```
1  // 将 a[l...mid] 和 a[mid+1...r] 两部分进行归并
2  void merge(int a[], int l, int mid, int r) {
3      int i = l;           // 左边部分的下标
4      int j = mid + 1;     // 右边部分的下标
5      int k = 0;           // tmp 的下标
6      int tmp[r - l + 1];
7      while (i <= mid && j <= r) {
8          if (a[i] <= a[j]) tmp[k++] = a[i++];
9          else tmp[k++] = a[j++];
10     }
11     while (i <= mid)
12         tmp[k++] = a[i++];
13     while (j <= r)
14         tmp[k++] = a[j++];
15     // 将 tmp 全部拷贝到原数组中
16     for (i = l, k = 0; i <= r; i++, k++)
17         a[i] = tmp[k];
18 }
19
20 // 对 a[l...r] 的范围进行排序
21 void merge_sort(int a[], int l, int r) {
22     if (l >= r)
23         return;
24     int mid = (l + r) / 2;
25     merge_sort(a, l, mid);
26     merge_sort(a, mid + 1, r);
27     merge(a, l, mid, r);
28 }
```

merge 函数中共有 3 个 while 循环。其中，第一个循环依次比较左右两个子数组当前下标位置的元素，将较小的一个加入到结果数组中并移动那一侧的下标，直到其中一个子数组遍历完毕；第二个循环将左边子数组中的剩余元素加入到结果数组中；第三个循环将右边子数组中的剩余元素加入到结果数组中。在一次 merge 操作中，后两个 while 循环一定只有一个会被执行。

$$T(n) = \begin{cases} O(1) & n = 1, \\ 2T\left(\frac{n}{2}\right) + O(n) & n > 1, \end{cases}$$

求解得到 $T(n) = O(n \log n)$。

该算法也可以改写成非递归形式，只需要将数组元素两两配对后分别

归并，多次迭代后就能完成排序。

2.8 快速排序

先从当前数组中选取一个基准元素 flag，小于其的元素都放在左边，大于其的元素都放在右边，然后递归地对左右两个子数组进行快速排序。

```
1  void swap(int *a, int *b) {
2      int t = *a;
3      *a = *b;
4      *b = t;
5  }
6
7  // 根据基准元素 a[p] 将数组 a[p...r]
       分为两部分，返回分界点下标
8  int partition(int a[], int p, int r) {
9      int i = p;       // 左指针
10     int j = r + 1;   // 右指针
11     int x = a[p];    // 基准元素
12     while (1) {
13         // 从左向右找第一个大于等于 x 的元素
14         while (a[++i] < x && i < r);
15         // 从右向左找第一个小于等于 x 的元素
16         while (a[--j] > x);
17         if (i >= j)
18             break;
19         swap(a + i, a + j);
20     }
21     a[p] = a[j];
22     a[j] = x;
23     return j;
24 }
25
26 // 对数组 a[p...r] 快速排序
27 void quick_sort(int a[], int p, int r) {
28     if (p < r) {
29         int q = partition(a, p, r);      // 获取分界点下标
30         quick_sort(a, p, q - 1);
31         quick_sort(a, q + 1, r);
32     }
33 }
```

最好情况下，每次选取的 flag 都是当前数组中的中位数，

$$T(n) = \begin{cases} O(1) & n = 1, \\ 2T\left(\frac{n}{2}\right) + O(n) & n > 1, \end{cases}$$

求解得到 $T(n) = O(n \log n)$。

最坏情况下，每次选取的 flag 都是当前数组中最小的元素，此时快速排序退化为冒泡排序，

$$T(n) = \begin{cases} O(1) & n = 1, \\ T(n-1) + O(n) & n > 1, \end{cases}$$

求解得到 $T(n) = O(n^2)$。

平均情况下快速排序的时间复杂度为 $O(n \log n)$，可以通过修改 partition 函数，随机选择 flag 来产生随机的划分。

快速排序平均情况下的时间复杂度与归并排序相同，但是快速排序的常数项更小（可以原地排序，没有复制开销），因此实际运行速度通常更快。

```
1  int randomized_partition(int a[], int p, int r) {
2      int i = rand() % (r - p + 1) + p; // 随机选一个作为
           flag
3      swap(a + p, a + i);
4      return partition(a, p, r);
5  }
6
7  int randomized_quick_sort(int a[], int p, int r) {
8      if (p < r) {
9          int q = randomized_partition(a, p, r);
10         randomized_quick_sort(a, p, q - 1);
11         randomized_quick_sort(a, q + 1, r);
12     }
13 }
```

2.9 线性时间选择

在含有 n 个元素的线性序列中找到第 k 小的元素。

先介绍 randomized-select 算法，该算法思想与快速排序类似。先随机选取一个元素作为 flag，然后将小于 flag 的元素放在左边，大于 flag 的元素放在右边。如果 flag 的下标为 i，则左边有 i 个元素，右边有 $n-i-1$ 个元素。如果 $i = k-1$，则 flag 即为第 k 小的元素；如果 $i > k-1$，则在左边的 i 个元素中继续寻找第 k 小的元素；如果 $i < k-1$，则在右边的 $n-i-1$ 个元素中继续寻找第 $k-i-1$ 小的元素。

```cpp
int randomized_select(int a[], int p, int r, int i) {
    if (p == r)
        return a[p];
    int q = randomized_partition(a, p, r);  // 获取分界点下标
    int k = q - p + 1;                      // 左子数组元素个数
    if (i == k)
        return a[q];
    else if (i < k)
        return randomized_select(a, p, q - 1, i);
    else
        return randomized_select(a, q + 1, r, i - k);
}
```

易知 randomized-select 算法最坏情况下时间复杂度为 $O(n^2)$，平均情况下时间复杂度为 $O(n)$。

下面讨论最坏情况下时间复杂度为 $O(n)$ 的算法。由递归问题的结构知，若能够在线性时间内找到划分基准，使得划分出的两个子数组大小至少为原数组的 ε 倍，则可以在 $O(n)$ 的时间内解决问题。

这是分治法中“平衡子问题”的重要思想。

按以下步骤找到划分基准：

1. 将 n 个元素划分为 $\frac{n}{5}$ 组，每组 5 个元素，剩余元素组成一组，共 $\left\lceil \frac{n}{5} \right\rceil$ 组；
2. 对每组 5 个元素进行排序，找到每组的中位数 (如果有偶数个元素，找较大的一个中位数)，共 $\left\lceil \frac{n}{5} \right\rceil$ 个中位数；
3. 对这 $\left\lceil \frac{n}{5} \right\rceil$ 个中位数递归地调用 select 函数，找到它们的中位数 x；
4. 以 x 为基准，对原数组进行划分，得到左右两个子数组，其中左边的元素个数至少为 $\frac{3}{10}(n-5)$。

开始时选出的中位数不小于组内的 3 个元素，随后选出的数不小于所有中位数中 $\frac{1}{2}(\lceil n/5 \rceil - 1) = \lceil (n-5)/10 \rceil$ 个元素，最终选出的数不小于组内 $\frac{3}{10}(n-5)$ 个元素。

当 $n \geq 75$ 时，$\frac{3}{10}(n-5) \geq \frac{n}{4}$，$\varepsilon = \frac{3}{4}$，与此前预期相符。

```cpp
// 在 a[p...r] 中查找第 k 小元素
int select(int a[], int p, int r, int k) {
    // 当元素个数小于 75 时，使用插入排序
    if (r - p < 75) {
        sort(a + p, r - p + 1);
        return a[p + k - 1];
    }
```

```
    // 每 5 个元素一组找出中位数，并将中位数与 a[p+i] 交换位置
    for (int i = 0; i <= (r - p - 4) / 5; i++) {
        sort(a + p + 5 * i, 5);
        swap(a + p + 5 * i, a + p + i);
    }
    // 递归调用 select 函数求出 a[p...r] 的中位数 x
    int x = select(a, p, p + (r - p - 4) / 5, (r - p - 4) / 10 + 1);
    // 以 x 为基准元素，调用 partition 函数划分数组，并获取分界点下标 q
    int q = partition(a, p, r, x);
    // 求出分界点下标 q 相对于数组 a[p...r] 的位置 i
    int i = q - p + 1;
    // 如果 k 等于所求的位置 i，则返回 a[q]
    if (k == i)
        return a[q];
    // 如果 k 小于 i，则返回 a[p...q-1] 中的第 k 小元素
    else if (k < i)
        return select(a, p, q - 1, k);
    // 如果 k 大于 i，则返回 a[q+1...r] 中的第 k-i 小元素
    else
        return select(a, q + 1, r, k - i);
}
```

分析 $n \geq 75$ 时的复杂度。设求解时间为 $T(n)$，则找中位数的复杂度为 $O(n)$，找中位数的中位数时间为 $T(n/5)$，partition 的复杂度为 $O(n)$，对划分后的子数组递归调用的时间为 $T(3n/4)$，

线性时间复杂度的关键之处在于 $n/5 + 3n/4 = 19n/20 < n$。

$$T(n) = \begin{cases} O(1) & n < 75, \\ O(n) + T\left(\frac{n}{5}\right) + T\left(\frac{3n}{4}\right) & n \geq 75, \end{cases}$$

求解得到 $T(n) = O(n)$。

2.10 最接近点对问题（略）

2.11 循环赛日程表

有 $n = 2^k$ 个运动员参加比赛，要求每个运动员与其他 $n-1$ 个运动员各比赛一次，每个运动员每天只能比赛一场，要求 $n-1$ 天内完成所有

比赛。按此要求设计比赛日程表。

可将所有选手分成两半，将子问题规模减小至 $\frac{n}{2}$，然后递归地设计比赛日程表。

	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7
1	2	3	4	5	6	7	8
2	1	4	3	6	5	8	7
3	4	1	2	7	8	5	6
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	1	4	3
7	8	5	6	3	4	1	2
8	7	6	5	4	3	2	1

图 2.1: $n = 8$ 时的循环赛日程表

图中横轴为比赛日，纵轴为选手每日的对手，相同颜色的矩阵可以通过复制得到。

Chapter 3

动态规划

Contents

3.1 矩阵连乘问题 16

3.2 动态规划算法的基本要素 18

3.3 最长公共子序列 19

3.4 最大子段和 21

3.5 凸多边形最优三角剖分 22

3.6 多边形游戏 24

3.7 图像压缩 26

3.8 电路布线 26

3.9 流水作业调度（略） 26

3.10 0—1 背包问题 26

3.11 最优二叉搜索树（略） 28

重难点

最优子结构的证明、矩阵连乘问题、最长公共子序列、最大子段和、凸多边形最优三角剖分、0—1 背包问题。

3.1 矩阵连乘问题

给定 n 个矩阵 $A_1, A_2, \ldots, A_n$，其中 A_i 的规模为 $p_{i-1} \times p_i$。求完全括号化方案，使得计算乘积 $A_1 A_2 \ldots A_n$ 所需标量乘法次数最少。

分析最优子结构

若计算 $A_1 \cdots A_n$ 的次序是最优的，则由其加括号方式 $((A_1 \cdots A_k) \cdot (A_{k+1} \cdots A_n))$ 可知，$A_1 \cdots A_k$ 和 $A_{k+1} \cdots A_n$ 两个矩阵子链的计算次序也是最优的。

使用反证法证明最优子结构。

证明. 假设 $A_1 \cdots A_n$ 计算次序最优，记其乘法次数为 m。将其加括号方式 $((A_1 \cdots A_k) \cdot (A_{k+1} \cdots A_n))$ 产生的两个矩阵子链 $A_1 \cdots A_k$ 与 $A_{k+1} \cdots A_n$ 的乘法次数分别记为 m_1 和 m_2。易知 $m = m_1 + m_2 + c$，其中 $c = p_{k-1}p_k p_n$ 为两个矩阵相乘的乘法次数。若子链 $A_1 \cdots A_k$ 存在更优的计算次序，则该次序下乘法次数 m_1' 必然小于 m_1。用新的计算次序替换原来的计算次序，则 $A_1 \cdots A_n = ((A_1 \cdots A_k) \cdot (A_{k+1} \cdots A_n))$ 的乘法次数变为 $m_1' + m_2 + c < m$，与 $A_1 \cdots A_n$ 原计算次序最优的假设矛盾。 □

因此，矩阵连乘问题具有最优子结构性质。

建立递归关系

计算 $A_i \cdots A_j$ 时，可以在 $A_i \cdots A_k$ 和 $A_{k+1} \cdots A_j$ $(i \le k < j)$ 之间断开，找到最优的断开位置 k，使得乘法次数最少。设 $m(i,j)$ 为计算 $A_i \cdots A_j$ 所需的最少乘法次数，则有

$$m(i,j) = \begin{cases} 0, & i = j \\ \min\limits_{i \le k < j} \{m(i,k) + m(k+1,j) + p_{i-1}p_k p_j\}, & i < j \end{cases}$$

其中 $p_{i-1}p_k p_j$ 为两个子矩阵相乘所需的乘法次数。

计算最优值

根据矩阵链的长度自底向上计算，过程中保存问题答案。

$m(i,j)$ 的计算需要用到 $m(i,k)$ 和 $m(k+1,j)$，后两者的矩阵链长度小于前者，因此采用矩阵链长度从 1 递增的方式迭代计算。

```
1  // p[0...n] 数组中存放矩阵链的行数和列数
2  // n 为矩阵链的长度
3  // m[i][j] 用于存放 A_i ... A_j 的最优值
4  // s[i][j] 用于存放 A_i ... A_j 的最优断开位置
5  void mat_chain(int *p, int n, int **m, int **s) {
6      // 边界值
7      for (int i = 1; i <= n; i++)
8          m[i][i] = 0;
9      // 迭代计算，r 为链长
10     for (int r = 2; r <= n; r++) {
11         // i 为链的起点
```

```
12          for (int i = 1; i <= n - r + 1; i++) {
13              // j 为链的终点
14              int j = i + r - 1;
15              // 计算左链仅有一个矩阵的情况
16              m[i][j] = m[i + 1][j] + p[i - 1] * p[i] *
                    p[j];
17              s[i][j] = i;
18              // 依次计算其他断开位置的情况，找到最优解
19              for (int k = i + 1; k < j; k++) {
20                  int t = m[i][k] + m[k + 1][j] + p[i - 1]
                        * p[k] * p[j];
21                  if (t < m[i][j]) {
22                      m[i][j] = t;
23                      s[i][j] = k;
24                  }
25              }
26          }
27      }
28  }
```

构造最优解

由于 $s(i,j)$ 存储了计算 $A_i \cdots A_j$ 时的最优断开位置 k，因此可以根据 $s(i,j)$ 的计算结果递归构造出最优解。

```
1  void traceback(int **s, int i, int j) {
2      if (i == j)
3          return;
4      traceback(s, i, s[i][j]);
5      traceback(s, s[i][j] + 1, j);
6      printf("Multiply A[%d...%d] and A[%d...%d]\n",
7             i, s[i][j], s[i][j] + 1, j);
8  }
```

3.2 动态规划算法的基本要素

动态规划依赖于问题的最优子结构性质和子问题重叠性质。

最优子结构

子结构的最优解一定是全局最优解。

使用递归与动态规划求解的问题都具有最优子结构。

重叠子问题

自顶向下求解问题时可能有子问题重叠。动态规划利用了子问题的重叠性质，对每个子问题只求解一次。通常只需要多项式时间。

递归算法反复求解相同的子问题，动态规划算法将子问题的解保存在数组中，避免重复计算。

备忘录方法

备忘录方法时动态规划的变形，用于自顶向下求解问题。备忘录方法维护一个表，记录每个子问题的答案。在求解子问题前先检查是否已经求解过，若已求解则直接返回答案，否则求解并保存答案。

3.3 最长公共子序列

给定两个序列 $X = \{x_1, x_2, \ldots, x_m\}$ 和 $Y = \{y_1, y_2, \ldots, y_n\}$，求 X 和 Y 的最长公共子序列 $Z = \{z_1, z_2, \cdots z_k\}$。

最长公共子序列的结构

设序列 $X = \{x_1, x_2, \ldots, x_m\}$ 和 $Y = \{y_1, y_2, \ldots, y_n\}$ 的最长公共子序列为 $Z = \{z_1, z_2, \cdots z_k\}$，则

1. 若 $x_m = y_n$，则 $z_k = x_m = y_n$ 且 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的最长公共子序列；
2. 若 $x_m \neq y_n$ 且 $z_k \neq x_m$，则 Z 是 X_{m-1} 和 Y 的最长公共子序列；
3. 若 $x_m \neq y_n$ 且 $z_k \neq y_n$，则 Z 是 X 和 Y_{n-1} 的最长公共子序列。

其中，$X_{m-1} = \{x_1, \ldots, x_{m-1}\}$，$Y_{n-1} = \{y_1, \ldots, y_{n-1}\}$，$Z_{k-1} = \{z_1, \cdots z_{k-1}\}$。

子问题的递归结构

要找出 $X = \{x_1, x_2, \ldots, x_m\}$ 和 $Y = \{y_1, y_2, \ldots, y_n\}$ 的最长公共子序列，考虑

- 若 $x_m = y_n$，找出 X_{m-1} 和 Y_{n-1} 的最长公共子序列，然后在末尾添加 x_m 或 y_n；

- 若 $x_m \neq y_n$，找出 X_{m-1} 和 Y 的最长公共子序列，以及 X 和 Y_{n-1} 的最长公共子序列，取两者中较长的一个。

将 X_i 和 Y_j 的最长公共子序列长度记为 $c(i,j)$，则有

$$c(i,j) = \begin{cases} 0, & i = 0 \text{ 或 } j = 0 \\ c(i-1,j-1)+1, & i,j > 0 \text{ 且 } x_i = y_j \\ \max\{c(i-1,j), c(i,j-1)\}, & i,j > 0 \text{ 且 } x_i \neq y_j \end{cases}$$

计算最优值

```
1  // x[1...m], y[1...n] 为两个序列
2  // c[i][j] 为最长公共子序列的长度
3  // b[][] 用于构造最长公共子序列
4  void lcs_length(char *x, char *y, int m, int n, int **c,
       int **b) {
5      // 边界值
6      for (int i = 0; i <= m; i++)
7          c[i][0] = 0;
8      for (int j = 0; j <= n; j++)
9          c[0][j] = 0;
10     // 迭代计算
11     for (int i = 1; i <= m; i++)
12         for (int j = 1; j <= n; j++) {
13             // x[i] 和 y[j] 相等时, c[i][j] 为
                   c[i-1][j-1] + 1
14             if (x[i] == y[j]) {
15                 c[i][j] = c[i - 1][j - 1] + 1;
16                 b[i][j] = 1;
17             }
18             // x[i] 和 y[j] 不相等时, c[i][j] 为
                   c[i-1][j] 和 c[i][j-1] 的最大值
19             else if (c[i - 1][j] >= c[i][j - 1]) {
20                 c[i][j] = c[i - 1][j];
21                 b[i][j] = 2;
22             }
23             else {
24                 c[i][j] = c[i][j - 1];
25                 b[i][j] = 3;
26             }
```

```
27          }
28  }
```

构造最优解

代码中

- $b(i,j)=1$ 表示 X_i 和 Y_j 的最长公共子序列是由 X_{i-1} 和 Y_{j-1} 的最长公共子序列加上 x_i 或 y_j 得到的；
- $b(i,j)=2$ 表示 X_i 和 Y_j 的最长公共子序列是由 X_{i-1} 和 Y_j 的最长公共子序列得到的；
- $b(i,j)=3$ 表示 X_i 和 Y_j 的最长公共子序列是由 X_i 和 Y_{j-1} 的最长公共子序列得到的。

可以根据 b 中的值递归构造最优解。

```
1  void lcs(char *x, int **b, int i, int j) {
2      if (i == 0 || j == 0)
3          return;
4      if (b[i][j] == 1) {
5          lcs(x, b, i - 1, j - 1);
6          printf("%c", x[i]);
7      }
8      else if (b[i][j] == 2)
9          lcs(x, b, i - 1, j);
10     else
11         lcs(x, b, i, j - 1);
12 }
```

算法的改进

将 $c(i,j)$ 与 $c(i-1,j-1)$, $c(i-1,j)$, $c(i,j-1)$ 的值比较，可以直接判断该步骤最长公共子序列的构造方式，无需使用 b 数组。

另外，求解 $c(i,j)$ 时只需要用到 $c(i-1,j-1)$, $c(i-1,j)$, $c(i,j-1)$ 的值，因此可以只使用一个两行数组来保存 c 的值。

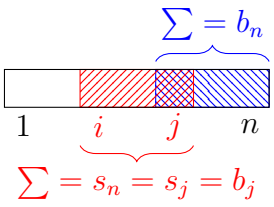
动态规划中，这种压缩数组行数的存储方式称为滚动数组。这种优化空间的思想经常被用到。

3.4 最大子段和

给定 n 个整数组成的序列 $a_1,\cdots,a_n$，求该序列形如 $\sum_{k=i}^{j} a_k$ 的子段和的最大值。(如果所有整数均为负数，则最大子段和为 0。)

最大字段和问题的动态规划算法

对于整数序列 $a_1, \cdots, a_n$，记其最大子段和为 s_n，记包含其右端点的连续区间和的最大值为 $b_n = \max\limits_{l=1}^{n} \sum\limits_{k=l}^{n} a_k$。假设 $s_n = \sum\limits_{k=i}^{j} a_k$ 为其最大子段和，易知：



最大子段和图示

- $a_1, \cdots, a_j$ 的最大子段和 s_j 也为 $\sum_{k=i}^{j} a_k$；
- $a_1, \cdots, a_j$ 包含右端点的连续区间和的最大值 b_j 也为 $\sum_{k=i}^{j} a_k$。

也就是说，s_n 的值要么为此前某个范围内包含右端点的连续区间和的最大值 $b_j (j < n)$ (注意到 $b_j = s_{n-1}$)，要么是自身包含右端点的连续区间和的最大值 b_n，即

$$s_i = \max\{b_i, s_{i-1}\},$$

其中 $b_i = \max\{b_{i-1} + a_i, a_i\}$。

b_i 的求法可以从另一个角度来看，若 $b_{i-1} \leq 0$，$b_i = a_i \geq b_{i-1} + a_i$；若 $b_{i-1} > 0$，$b_i = b_{i-1} + a_i > a_i$。

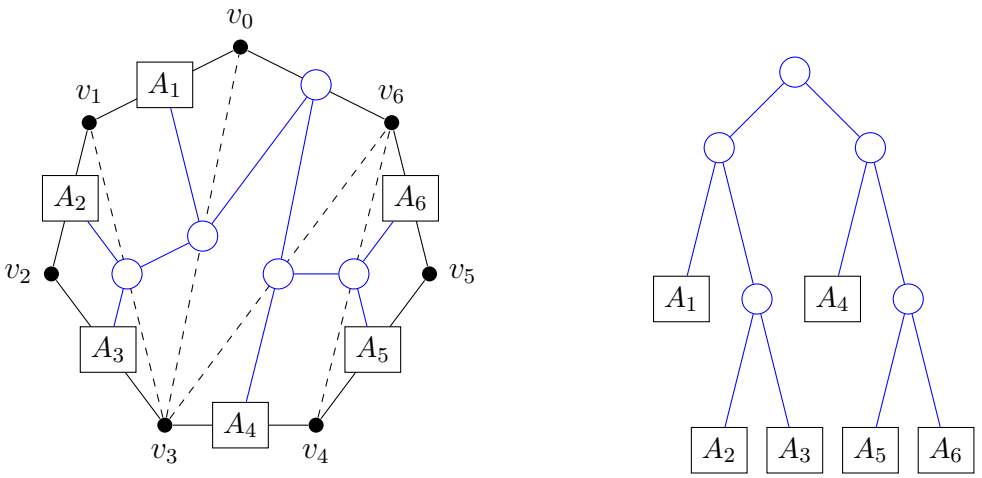
```
1  // 求解 a[1...n] 的最大子段和
2  int max_sum(int n, int a[]) {
3      int s = 0; // 当前的最大子段和
4      int b = 0; // 当前右区间的最大和
5      for (int i = 1; i <= n; i++) {
6          if (b > 0) {
7              b += a[i];
8          } else {
9              b = a[i];
10         }
11         if (b > s) {
12             s = b;
13         }
14     }
15     return s;
16 }
```

最大子段和求解算法的时间复杂度为 $O(n)$。

最大子段和问题与动态规划算法的推广（略）

3.5 凸多边形最优三角剖分

凸多边形最优三角剖分问题：给定凸多边形 $P = \{v_0, v_1, \cdots v_{n-1}\}$，以及定义在由凸多边形的边和弦组成的三角形上的权函数 w，求 P 的三角剖分 T，使得 $w(T) = \sum_{t \in T} w(t)$ 最小。



(a) 三角剖分　　(b) 对应的树

图 3.1: 图示的三角剖分与矩阵连乘 $((A_1(A_2A_3))(A_4(A_5A_6)))$ 的加括号方式等价。

凸多边形的三角剖分方式等价于一棵完全二叉树。对于图 3.1a 中的三角剖分，其对应的语法树如图 3.1b 所示。一般选择边 v_0v_6 为语法树的根结点。易知选择任何一条边为根结点都是等价的。多边形的其余边为叶结点。

矩阵连乘可以看成一种特殊的三角剖分，其权函数为两个矩阵相乘需要的乘法次数。

最优子结构性质（略）

三角剖分的最优子结构与矩阵连乘类似，不加赘述。

最优三角剖分的递归结构

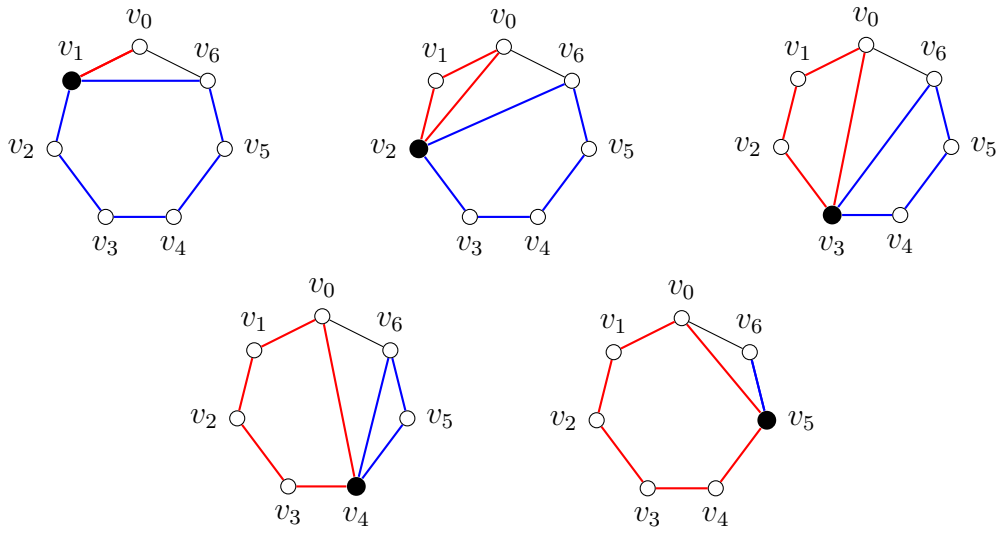


图 3.2: $t(1,6)$ 的子问题划分情况

定义 $t(i,j)$ 为凸子多边形 $\{v_{i-1}, v_i, \cdots, v_j\}$ 最优三角剖分对应的权函数值。方便起见，设 $t(i-1,i)=0$。则有

$$t(i,j)=\begin{cases}0, & i=j\\ \min\limits_{i\le k\le j}\{t(i,k)+t(k+1,j)+w(v_{i-1}v_kv_j)\}, & i<j\end{cases}$$

注意：此处的下标 i 与 j 表示的是凸子多边形的边的编号，而非顶点编号。例如，$t(1,3)$ 表示边 $A_1=v_0v_1$，$A_2=v_1v_2$，$A_3=v_2v_3$ 与分割线围成的凸子多边形 $\{v_0,v_1,v_2,v_3\}$ 的计算结果。

计算最优值

```
// t[i][j] 存储子多边形最优三角剖分的权函数值
// s[i][j] 存储子多边形最优三角剖分的分界点编号
void min_weight_triangulation(int n, double **t, int
    **s) {
    // 初始化仅有一条边的情况
    for (int i = 1; i <= n; i++)
        t[i][i] = 0;
    // 迭代计算结果，r 为子多边形的顶点数 - 1
    for (int r = 2; r <= n; r++) {
        for (int i = 1; i <= n - r + 1; i++) {
            int j = i + r - 1;  // 子多边形的最后一个顶点
            t[i][j] = t[i + 1][j] + w(i - 1, i, j);
            s[i][j] = i;
            for (int k = i + 1; k <= j - 1; k++) {
                double u = t[i][k] + t[k + 1][j] + w(i -
                    1, k, j);
                if (u < t[i][j]) {
                    t[i][j] = u;
                    s[i][j] = k;
                }
            }
        }
    }
}
```

上述算法的时间复杂度为 $O(n^3)$。

3.6 多边形游戏

游戏开始时有一个 n 边形，每个顶点上有一个整数值（可为负数），每条边用 0 到 $n-1$ 编号，每条边上有一个运算符号 $+$ 或 $*$。

游戏第一步删除一条边，随后的 $n-1$ 步每步按以下方式操作：

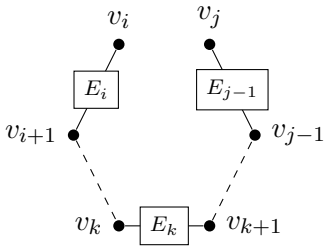
1. 选择一条边 E_i 以及其两端的顶点 v_i 和 v_{i+1}；

2. 用 E_i 上的运算符计算 v_i 和 v_{i+1} 的运算结果 r；

3. 将计算结果 r 作为新顶点，替换掉 E_i 及其两端的顶点 v_i 和 v_{i+1}。

直到所有边都被删除，游戏结束。游戏得分为最后剩余顶点的值。

对于给定的多边形，设计一个算法，计算游戏的最高得分。



最优子结构性质（略）

递归求解

对于一条链 $\{v_i, \cdots, v_j\}$，其中包含边 $E_i, \cdots E_{j-1}$。由于存在负数的乘法运算，因此我们要同时考虑链的最大值和最小值。设 $M(i,j)$ 和 $m(i,j)$ 分别为链 $\{v_i, \cdots, v_j\}$ 计算结果的最大值和最小值。

此处 i, j 的含义与书上不同，思路也与课本有小差异，阅读时请留意。

假设链从边 E_k 处断开，则两个子链的最优解分别为 $M(i,k)$, $m(i,k)$ 和 $M(k+1,j)$, $m(k+1,j)$。为了简化表示，令

$$\begin{cases} a_{\max} = M(i,k), \\ a_{\min} = m(i,k), \\ b_{\max} = M(k+1,j), \\ b_{\min} = m(k+1,j). \end{cases}$$

下面考虑 E_k 上的运算符号：

1. 若 E_k 上的运算符号为 $+$，
   $M_k(i,j) = a_{\max} + b_{\max}$,
   $m_k(i,j) = a_{\min} + b_{\min}$;

2. 若 E_k 上的运算符号为 $*$，由于可能有负数存在，
   $M_k(i,j) = \max\{a_{\max}b_{\max}, a_{\min}b_{\min}, a_{\max}b_{\min}, a_{\min}b_{\max}\}$,
   $m_k(i,j) = \min\{a_{\max}b_{\max}, a_{\min}b_{\min}, a_{\max}b_{\min}, a_{\min}b_{\max}\}$.

此处的记号 M_k 和 m_k 表明只考虑从 k 处断开的情况。

由于断开位置 k 的不确定性，因此需要枚举所有可能的断开位置，即

$$\begin{cases} M(i,j) = \max\limits_{i \le k < j} M_k(i,j), \\ m(i,j) = \min\limits_{i \le k < j} m_k(i,j). \end{cases}$$

由此得到了对于链的递归求解方法。对于 $i = j$ 的边界情况，$M(i,j) = m(i,j) = v_i$。

由于多边形是封闭的，因此需要考虑删除的第一条边的所有可能位置。对于多边形 $\{v_0,\cdots,v_{n-1}\}$，设断开位置为 E_i (即 $v_iv_{i+1}, 0\le i\le n-1$)。我们可以记断开后的链为 $\{v_{i+1},\cdots,v_{n-1},v_n,\cdots,v_{n+i}\}$。由于多边形可以看成一个环，故下标 $n,n+1,\cdots,n+i$ 分别与 $0,1,\cdots i$ 模 n 同余，即顶点 $v_n,v_{n+1},\cdots v_{n+i}$ 分别与顶点 $v_0,v_1,\cdots v_i$ 相同。利用同余的性质扩展顶点编号为 $0,1,\cdots,2n-1$，则这种情况下的最大值为 $M(i,n+i)$。遍历所有可能的断开位置，即可得到最终的结果。

算法描述

代码较长，此处不展示，可以参考书本和课件给出的代码。求解复杂度为 $O(n^3)$。

上述思路理解起来可能较为容易，但是书上思路的代码实现可能更为简洁。

3.7 图像压缩

待补充。

3.8 电路布线

待补充。

3.9 流水作业调度（略）

3.10 0—1 背包问题

给定 n 种物品和一个容量为 c 的背包，物品 i 的重量为 w_i，价值为 v_i，求解将哪些物品装入背包可使价值总和最大。在选择装入背包的物品时，对每种物品 i 只有两种选择：装入背包或不装入背包。

此问题的形式化描述是，给定 $c>0, w_i>0, v_i>0\ (1\le i\le n)$，求解 n 元组 $(x_1,\cdots,x_n), x_i\in\{0,1\}$，使得 $\sum_{i=1}^{n} w_ix_i\le c$，且 $\sum_{i=1}^{n} v_ix_i$ 最大。

最优子结构性质

设 $(y_1,y_2,\cdots y_n)$ 是所给定的 n 个物品的一个最优解，则 $(y_2,y_3,\cdots y_n)$ 是下面相应子问题的一个最优解：

$$\max\sum_{i=2}^{n} v_ix_i,\ \text{s.t.}\begin{cases}\sum\limits_{i=2}^{n} w_ix_i\le c-w_1x_1\\ x_i\in\{0,1\} \qquad\qquad 2\le i\le n\end{cases}$$

递归关系

设所给问题的子问题

$$\max \sum_{k=i}^{n} v_k x_k, \text{ s.t.} \begin{cases} \sum_{k=i}^{n} w_k x_k \leq j \\ x_i \in \{0,1\} \qquad i \leq k \leq n \end{cases}$$

最优值为 $m(i,j)$。即 $m(i,j)$ 表示在背包容量为 j 时，从第 i 个物品开始选择，所能获得的最大价值。计算的递归表达式为

$$m(i,j) = \begin{cases} m(i+1,j), & 0 \leq j < w_i \\ \max\{m(i+1,j), m(i+1,j-w_i)+v_i\}, & j \geq w_i \end{cases}$$

$0 \leq j < w_i$ 时，背包容量不足以装入物品 i，因此只能选择不装入。

考虑边界情况 $(i = n)$。只放一个物品时，要么能装下，要么装不下。故边界条件为

$$m(n,j) = \begin{cases} 0, & 0 \leq j < w_n \\ v_n, & j \geq w_n \end{cases}$$

算法描述

给出求解 0—1 背包问题的 Knapsack 算法。

```
1  #define min(a, b) ((a) < (b) ? (a) : (b))
2  #define max(a, b) ((a) < (b) ? (b) : (a))
3
4  void knapsack(int v[], int w[], int c, int n, int **m) {
5      int j_max = min(w[n] - 1, c);
6      // 初始化不能容纳物品 n 的情况
7      for (int j = 0; j <= j_max; j++) {
8          m[n][j] = 0;
9      }
10     // 初始化能容纳物品 n 的情况
11     for (int j = w[n]; j <= c; j++) {
12         m[n][j] = v[n];
13     }
14     // 迭代计算
15     for (int i = n-1; i > 1; i--) {
16         j_max = min(w[i] - 1, c);
17         // 不能容纳物品 i 的情况
18         for (int j = 0; j <= j_max; j++) {
19             m[i][j] = m[i+1][j];
20         }
```

```
21          // 能容纳物品 i 的情况
22          for (int j = w[i]; j <= c; j++) {
23              m[i][j] = max(m[i+1][j], m[i+1][j-w[i]] +
                    v[i]);
24          }
25      }
26      // 计算最优值存放在 m[1][c] 中
27      m[1][c] = m[2][c];
28      if (c >= w[1]) {
29          m[1][c] = max(m[1][c], m[2][c-w[1]] + v[1]);
30      }
31  }
32
33  void traceback(int **m, int w[], int c, int n, int x[]) {
34      for (int i = 1; i < n; i++) {
35          if (m[i][c] == m[i + 1][c]) { // 第 i
                个物品不放入背包
36              x[i] = 0;
37          } else { // 第 i 个物品放入背包
38              x[i] = 1;
39              c -= w[i];
40          }
41      }
42      x[n] = (m[n][c]) ? 1 : 0;
43  }
```

计算复杂性分析

Knapsack 算法的时间复杂度为 $O(nc)$，Traceback 算法的时间复杂度为 $O(n)$。

3.11 最优二叉搜索树（略）

Chapter 4

贪心算法

Contents

4.1 活动安排问题 29
4.2 贪心算法的基本要素 29
4.3 最优装载 30
4.4 哈夫曼编码 30
4.5 单源最短路径 30
4.6 最小生成树 30
4.7 多机调度问题（略） 30

4.1 活动安排问题

待补充。

4.2 贪心算法的基本要素

贪心选择性质

使用数学归纳法证明贪心选择性质。

所求问题的整体最优解可以通过一系列局部最优的选择，即贪心选择来达到。

最优子结构性质

同动态规划。

4.3 最优装载

待补充。

4.4 哈夫曼编码

4.5 单源最短路径

4.6 最小生成树

4.7 多机调度问题（略）