数据结构与算法

第 13 讲：基础算法

韩文弢

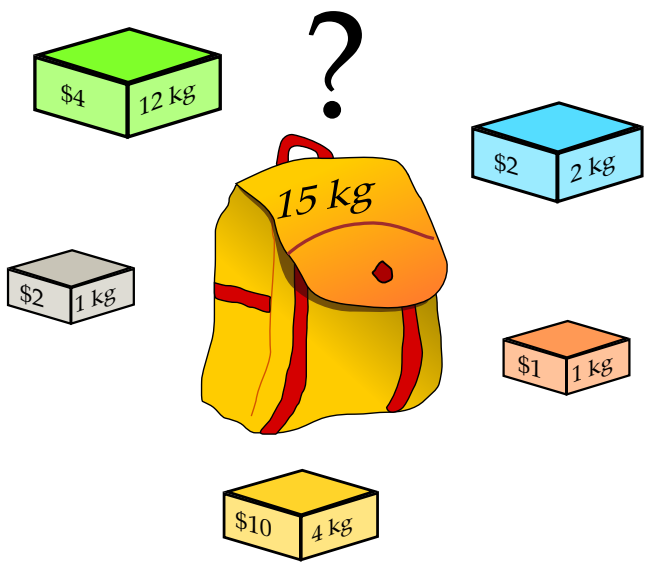
清华大学

2025—2026 学年度春季学期

本讲内容

13.1 问题引入

13.1.1 背包问题



有一个背包，最大承重是 t。有 n 个物品，其中第 i 个物品（$1 \leq i \leq n$）的重量为 w_i，价值为 v_i。

问：如何选取物品，在不超过背包承重的情况下，使得装入的物品价值之和最大。

背包问题还有一些条件的变化，例如在选取物品时是要整个物品都选取，还是可以选取一个物品的一部分。

13.1.2 最优化问题

定义 13.1 数学、工程学、计算机科学和经济学领域中，**最优化问题**（也称优化问题，optimization problem）是指从所有可行解中找到最优答案的问题。

根据变量是连续的或离散的，可将最优化问题分为两类：

- 具有连续变量的最优化问题称为**连续优化**，其中必须找到连续函数的最优值。
- 具有离散变量的最优化问题称为**离散优化**，其中必须找到可数集合中的整数、排列或图等对象。

13.2 贪心法

13.2.1 部分背包问题

在背包问题中，如果物品选取时允许选取一个物品的一部分，称为部分背包问题。部分背包问题是典型的连续优化。

例. 已知背包的最大承重 $t = 50$，有 3 件物品，物品的重量和价值如下表所示。

物品 i	重量 w_i	价值 v_i
1	20	100
2	10	60
3	30	120

最优解：物品 1 和 2 全取，物品 3 中取 20，最大价值为 240。

13.2.2 部分背包问题的求解思路

思路：由于可以只取一部分，因此关键是尽量取单位重量价值高的物品。以上一页的条件为例，先计算单位重量的价值：

物品 i	重量 w_i	价值 v_i	单位重量的价值 u_i
1	20	100	5
2	10	60	6
3	30	120	4

由上表可知，应该按物品 $2, 1, 3$ 的顺序来取，直到达到最大承重。

这种每次按当前阶段（局部）的最优策略进行决策的方法称为**贪心法**（greedy algorithm）。

13.2.3 部分背包问题算法

伪码 13.1: 部分背包问题的贪心算法

FractionalKnapsack(n, t, w, v)

将 w 和 v 按 $\frac{v_i}{w_i}$ 降序排序

$s \leftarrow 0$

\# 按单位重量价值的降序进行贪心选择

for $i \leftarrow 0$ **to** $n-1$

  $x \leftarrow \min\{t, w_i\}$

  if $x = 0$

    break

  $s \leftarrow s + \frac{v_i}{w_i} \times x$

  $t \leftarrow t - x$

return s

13.2.4 部分背包问题算法的分析

正确性：用反证法，设最优解的方案不是按单位重量价值降序选取的，则存在单位重量价值更高的物品部分，用它替代假设的最优解方案中同等重量但单位重量价值较低的部分后，所得到的方案比假设的最优解要好。

时间复杂度：排序为 $O(n \log n)$，贪心选择过程为 $O(n)$，总体为 $O(n \log n)$。

13.2.5 部分背包问题的实现

C++

```cpp
#include <algorithm>
#include <iostream>
#include <tuple>
#include <vector>
using namespace std;

// 部分背包问题
// 输入：
// 物品数量 n，背包容量 t
// 每件物品的重量 w_i 和价值 v_i，共 n 行
// 输出：
// 最大价值，获得最大价值时取几件物品
// 取的物品编号和重量
int main() {
```

13.2.5 部分背包问题的实现

```
15    // 输入数据
16    int n, t;
17    vector<tuple<int, int, int>> items;
18    cin >> n >> t;
19    for (int i = 0; i < n; i++) {
20      int w, v;
21      cin >> w >> v;
22      items.emplace_back(i, w, v);
23    }
24    // 使用贪心法求解
25    sort(items.begin(), items.end(),
26         [](const auto& lhs, const auto& rhs) {
27           const auto& [i1, w1, v1] = lhs;
28           const auto& [i2, w2, v2] = rhs;
```

13.2.5 部分背包问题的实现

```
29            return v1 * w2 > v2 * w1;
30        });
31    double s = 0;
32    vector<tuple<int, int>> select;
33    for (const auto& [i, w, v] : items) {
34      int x = min(w, t);
35      if (x == 0) break;
36      t -= x;
37      s += static_cast<double>(v) / w * x;
38      select.emplace_back(i, x);
39    }
40    // 输出结果
41    sort(select.begin(), select.end(),
42         [](const auto& lhs, const auto& rhs) {
```

13.2.5 部分背包问题的实现

```
43            const auto& [i1, x1] = lhs;
44            const auto& [i2, x2] = rhs;
45            return i1 < i2;
46          });
47    cout << s << ' ' << select.size() << endl;
48    for (auto [i, w] : select) {
49      cout << i << ' ' << w << endl;
50    }
51  }
```

13.2.6 贪心法的设计思想

贪心法的设计思想：分步骤解决问题，每一步都采取当前（局部）最优策略。两个关键点：

1. **划分步骤**：将原问题划分为①执行当前一步指令，②递归解决剩下的结构相同（但是规模更小）的子问题。
2. **“贪”**：也就是定义当前的最优策略。①这个策略必须能形成一个可行解，也就是满足问题的约束条件；②这个策略必须是当前所能达到的最优状态。

13.2.7 贪心法存在的问题

贪心法并不总能获得全局最优解。

例. 砝码称重问题：如果有 100 克、50 克和 10 克三种砝码称 160 克的重量，用贪心策略每次加入不超过余下重量的最大砝码，得到用一个 100 克、一个 50 克和一个 10 克，一共用 3 个砝码，是最优解。

然而，如果是用 120 克、50 克和 10 克三种砝码称 160 克的重量，用前面的贪心策略，会得到用一个 120 克和四个 10 克，一共 5 个砝码，而最优解是三个 50 克和一个 10 克，一共 4 个砝码。

13.2.8 活动安排问题

某人要去参加 n 个活动，给出每个活动 i（$1 \leq i \leq n$）的开始时间 s_i 和结束时间 t_i（$s_i < t_i$），规定对于每个活动，要么全程参加，要么不参加。问最多能参加其中的几个活动?

可能的贪心策略：在不冲突的前提下，

1. 每次参加**最早开始**的一项活动
2. 每次参加**时长最短**的一项活动
3. 每次参加**冲突最少**的一项活动
4. 每次参加**最早结束**的一项活动

只有第 4 种策略能保证全局最优。

13.2.9 活动安排问题示例

例. 有 6 个活动，时间如下表所示。

活动 i	开始 s_i	结束 t_i	时长	冲突数	选择
1	0	3	3	2	✓
2	1	4	3	3	
3	3	5	2	2	✓
4	0	6	6	4	
5	5	7	2	1	✓
6	7	9	2	0	✓

按“最早结束”策略：选活动 $1(t=3)$，再选活动 $3(t=5)$，再选活动 $5(t=7)$，再选活动 $6(t=9)$，共参加 4 个活动，为最优解。

而“最早开始”会先选活动 1 或 4（$s=0$），若先选活动 4，则只能再选活动 6，共参加 2 个，不是最优解。

13.3 动态规划

13.3.1 0-1 背包问题

在背包问题中，如果对于一件物品，要么整个都要，要么整个都不要，称为 **0-1 背包问题**。0-1 背包问题是典型的离散优化。与上一节的部分背包问题相比，约束条件从“可以部分选取”变为“只能整个选取”。

13.3.2 0-1 背包问题示例

例. 已知背包的最大承重 $t = 50$，有 3 件物品，物品的重量和价值如下表所示。

物品 i	重量 w_i	价值 v_i
1	20	100
2	10	60
3	30	120

13.3.3 0-1 背包问题的分析

考虑贪心法是否能正确求解 0-1 背包问题，贪心策略为每次取单位重量价值最高的物品。

物品 i	重量 w_i	价值 v_i	单位重量的价值 u_i
1	20	100	5
2	10	60	6
3	30	120	4

使用以上贪心策略的结果是选取物品 2 和 1，此时因承重限制无法再选取物品 3，总价值为 160。

然而，如果选取物品 1 和 3，总价值为 220，才是最优解。

13.3.4 0-1 背包问题的求解思路

由于问题的约束条件从可以部分选取变成了只能整个选取，约束条件会限制贪心策略的发挥，因此贪心法不能得到最优解。

考虑 0-1 背包问题如何拆成子问题，同时还要保留最优结构。在贪心法中，只考虑了物品，而没有将已选取物品的重量考虑在对子问题的刻画中。已选取物品的重量会影响后续能否选取其他物品。

设 $f_{i,j}$ 表示对于前 i 件物品，从中选取一些（具体是哪些物品不重要）的重量之和为 j 时所能获得的最大价值之和。

13.3.5 状态转移方程

基于前面 $f_{i,j}$ 的设置，有

$$f_{i,j} = \max \begin{cases} f_{i-1,j} \\ f_{i-1,j-w_i} + v_i \text{，如果 } j \geq w_i \end{cases}$$

解释：对于前 i 件物品，从中选取一些重量之和为 j 时可以获得的最大价值是如下二者中的较大者。

1. 对于前 $i-1$ 件物品，从中选取一些重量之和为 j 时可以获得的最大价值（不选取物品 i）
2. 对于前 $i-1$ 件物品，从中选取一些重量之和为 $j-w_i$ 时（$j \geq w_i$）可以获得的最大价值，加上物品 i 的价值（选取物品 i）

13.3.6 计算过程示例

物品 i	重量 w_i	价值 v_i
1	2	10
2	1	6
3	3	12

简便起见，物品的重量和价值都缩小 10 倍。

$f_{i,j}$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	10	10	10	10
2	0	6	10	16	16	16
3	0	6	10	16	18	22

13.3.7 0-1 背包问题算法

伪码 13.2: 0-1 背包问题的算法

Knapsack(n, t, w, v)

- 分配 $f[0..n, 0..t]$，元素都初始化为 0
- **for** $i \leftarrow 1$ **to** n
  - **for** $j \leftarrow 0$ **to** t
    - **if** $j \geq w[i]$
      - $f[i, j] \leftarrow \max\{f[i-1, j], f[i-1, j-w[i]] + v[i]\}$
    - **else**
      - $f[i, j] \leftarrow f[i-1, j]$
- **return** $f[n, t]$

13.3.8 0-1 背包问题算法的分析

上述方法称为**动态规划**（dynamic programming）。

正确性：使用归纳法，从初始状态出发，经过状态转移方程能算出更多状态的最优解，直到规模达到原问题。

时间复杂度：$O(nt)$，注意该复杂度被称为伪多项式复杂度，不仅与问题规模 n 有关，还与取值范围 t 有关。

空间复杂度：$O(nt)$，注意到计算时只跟上一行有关，可以优化到 $O(t)$。

13.3.9 0-1 背包问题的实现

C++

```cpp
#include <algorithm>
#include <iostream>
#include <ranges>
#include <tuple>
#include <vector>
using namespace std;

// 01 背包问题
// 输入：
// 物品数量 n，背包容量 t
// 每件物品的重量 w_i 和价值 v_i，共 n 行
// 输出：
// 最大价值，获得最大价值时取几件物品
// 取的物品编号
```

13.3.9 0-1 背包问题的实现

```
15  int main() {
16    // 输入数据
17    int n, t;
18    vector<tuple<int, int>> items;
19    cin >> n >> t;
20    for (int i = 0; i < n; i++) {
21      int w, v;
22      cin >> w >> v;
23      items.emplace_back(w, v);
24    }
25    // 使用动态规划进行求解
26    vector<vector<int>> f(n + 1, vector<int>(t + 1));
27    for (int i = 0; i < n; i++) {
28      auto [w, v] = items[i];
```

13.3.9 0-1 背包问题的实现

```
29      for (int j = 0; j <= t; j++) {
30        f[i + 1][j] = f[i][j];
31        if (j >= w) {
32          f[i + 1][j] = max(f[i + 1][j], f[i][j - w] + v);
33        }
34      }
35    }
36    // 输出结果
37    vector<int> s;
38    int i = n;
39    int j = t;
40    while (i > 0 && j > 0) {
41      if (f[i][j] == f[i - 1][j]) {
42        i--;
```

13.3.9 0-1 背包问题的实现

```
43      } else {
44        i--;
45        s.push_back(i);
46        auto [w, _] = items[i];
47        j -= w;
48      }
49    }
50    cout << f[n][t] << ' ' << s.size() << endl;
51    for (auto i : s | views::reverse) {
52      cout << i << ' ';
53    }
54    cout << endl;
55  }
```

13.3.10 动态规划方法

可以使用动态规划求解的优化问题都具有**最优子结构**，也就是最优解中，子问题的解也同样是该子问题的最优解。

使用动态规划求解的步骤：

1. 找出问题的最优子结构
2. 推导子问题最优解之间的递推关系（状态转移方程）
3. 按照正确的顺序计算子问题的最优解（从小到大，保证满足依赖关系）
4. 根据记录的值构造最优解方案

13.3.11 矩阵乘法问题

有 n 个矩阵 $A_1, A_2, A_3, ..., A_n$，矩阵 A_i 的大小为 $r_i \times r_{i+1}$。计算它们的乘积 $\prod_{i=1}^{n} A_i$，要求标量乘法次数最少。

例. A_1 为 1×10，A_2 为 10×5，A_3 为 5×2，若按 $(A_1A_2)A_3$ 来算乘法次数为 $1 \times 10 \times 5 + 1 \times 5 \times 2 = 60$，若按 $A_1(A_2A_3)$ 来算乘法次数为 $10 \times 5 \times 2 + 1 \times 10 \times 2 = 120$。

设状态：令 $f_{i,j}$ 表示从 A_i 到 A_j 相乘所需的最少标量乘法次数，则

$$f_{i,j} = \begin{cases} 0\text{，若 } i = j \\ \min_{i \le k < j}\{f_{i,k} + f_{k+1,j} + r_i \times r_{k+1} \times r_{j+1}\}\text{，若 } i < j \end{cases}$$

13.3.11 矩阵乘法问题

例. 对于上面的例子（$A_1 : 1 \times 10$，$A_2 : 10 \times 5$，$A_3 : 5 \times 2$），设 $r = (1, 10, 5, 2)$。

$f_{i,j}$	$j = 1$	$j = 2$	$j = 3$
$i = 1$	0	$1 \times 10 \times 5 = 50$	60
$i = 2$		0	$10 \times 5 \times 2 = 100$
$i = 3$			0

最少乘法次数为 $f_{1,3} = 60$，即按 $(A_1 A_2) A_3$ 的顺序。

13.4 小结

13.4.1 本讲小结

- 最优化问题：离散、连续
- 贪心法：贪心选择性质 + 最优子结构
- 动态规划：最优子结构 + 重叠子问题

对比	贪心法	动态规划	分治法
适用条件	贪心选择性质+最优子结构	最优子结构+重叠子问题	最优子结构+独立子问题
策略	每步取局部最优	枚举子问题取最优	分解→求解→合并
全局最优	不一定	保证	保证
典型问题	部分背包活动安排	0-1 背包矩阵乘法	归并排序快速排序

13.5 课程总结

13.5.1 主要知识点

- 线性结构：线性表、栈、队列、字符串
- 树状结构（层次结构）：树与二叉树、堆
- 排序与查找：排序、查找、二叉查找树、散列表
- 图状结构（网络结构）：图、最短路径
- 基础算法：贪心法、动态规划

13.5.2 常用容器

容器	底层实现	有序性	元素访问	查找	头部增删	尾部增删	中间增删
vector	顺序表	无序	随机访问	$O(n)$	$O(n)$	$O(1)$	$O(k)$
deque	分块数组	无序	随机访问	$O(n)$	$O(1)$	$O(1)$	$O(k)$
list	双向链表	无序	顺序访问	$O(n)$	$O(1)$	$O(1)$	$O(1)$
priority_queue	堆	有序	堆顶访问	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
map 等	平衡二叉树	有序	关键字访问	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
unordered_map 等	散列表	无序	关键字访问	$O(1)$	$O(1)$	$O(1)$	$O(1)$

其中，n 为容器中元素的数量，k 为增删位置与头部或尾部的距离。