C++ 算法讲义：贪心算法（上）

1. 引言

1.1 问题引入：如何使得资源分配最优化

在很多领域，我们频繁面对资源受限下的最优化问题。一个经典的抽象模型是：给定一组物品，每种物品都具有特定的价值和成本（例如重量、体积或时间消耗），以及一个总的成本预算（例如背包容量）。目标是在不超过预算的前提下，选取一部分或全部物品，使得总价值最大化。

这个问题，被称为**背包问题**，是运筹学和算法理论中的一个核心问题。根据物品是否可分割，它又分为两种主要变体。我们从其中更适合贪心策略的**部分背包问题**入手。

1.2 贪心算法的地位与应用

贪心算法是一种基础而强大的算法。它代表了一种简洁、高效的解决问题的世界观。其核心在于：

- **高效性**：相较于需要穷举所有可能性的暴力搜索算法，贪心算法通常具有多项式 $O(n+m)$ 时间复杂度，实现高效。在处理大规模数据时，这种效率优势尤为关键。
- **基础性**：许多图论和组合优化领域的基础算法都根植于贪心思想。例如，计算图的最小生成树的 Prim 算法和 Kruskal 算法，以及计算单源最短路径的 Dijkstra 算法，其核心都包含了一个贪心选择的过程，我们以后就将会学习二叉树与图论。
- **启发性算法**：即便在贪心策略无法保证得到全局最优解的复杂问题（NP-hard 问题）中，贪心算法也常被用作一种有效的近似算法或启发式方法，就算得不到最优解，也能够快速给出一个质量较高的可行解。

---

2. 核心思想

2.1 问题引入：部分背包问题(例12-1 OJ:B0174)

题目描述：

阿里巴巴走进了装满宝藏的藏宝洞。藏宝洞里面有 $N(N \leq 100)$ 堆金币，第 i 堆金币的总重量和总价值分别是 $m_i, v_i (1 \leq m_i, v_i \leq 100)$。阿里巴巴有一个承重量为 $T(T \leq 1000)$ 的背包，但并不一定有办法将全部的金币都装进去。他想装走尽可能多价值的金币。所有金币都可以随意分割，分割完的金币重量价值比（也就是单位价格）不变。请问阿里巴巴最多可以拿走多少价值的金币？

面对部分背包问题，即物品可被分割，我们的目标是最大化总价值。直觉上，存在多种可能的“贪心”标准：

- **价值最高优先**：优先装入总价值最高的物品。此策略的缺陷是显而易见的，一个价值极高但重量巨大的物品可能会占据所有容量，使得总价值远低于组合多个小物品的情况。
- **重量最轻优先**：优先装入重量最轻的物品，以期装入更多数量的物品。此策略同样存在问题，因为它完全忽略了物品的价值，可能会导致背包被大量低价值物品填满。

上述直觉性策略都是不成立了，于是我们应该**寻找一个正确的贪心标准。**

对于部分背包问题，一个更合理的标准应该是衡量单位重量所能提供的价值，即**价值密度(性价比)。**

2.2 贪心算法的性质

定义：贪心算法在求解问题的每一步中，都做出一个局部最优的选择。它寄希望于一系列的局部最优选择能够产生一个全局最优解。

一个问题能够使用贪心算法求解，通常需要满足以下两个关键性质：

1. **无后效性**
   即某个状态以前的过程**不会影响以后的状态，只与当前状态有关。**换句话说，我们做出的一个贪心选择，只会将原问题简化为一个规模更小的、同性质的子问题，而我们此前的选择并不会影响这个子问题的最优解。也就是**一个全局最优解可以通过一系列局部最优的选择来达到。**
2. **最优子结构**
   一个问题的最优解包含了其子问题的最优解。当我们做出一个贪心选择后，原问题就简化为了一个规模更小的、性质相同的子问题。例如，当我们将一个物品装入背包后，剩下的问题就变成了：使用剩余的背包容量，在剩余的物品中获取最大价值。这个子问题的解，结合我们已做出的选择，将构成原问题的解。

一个贪心策略的正确性必须经过严格的数学证明，通常使用“交换论证法”（也叫反证法）来证明。即假设存在一个不采用贪心选择的最优解，通过一系列交换操作，可以将其转化为一个采用贪心选择且结果不劣于原解的方案，从而证明贪心选择的正确性。

---

3. 实例：部分背包问题

3.1 案例描述

给定一个容量为 `W` 的背包和 `n` 个物品。第 `i` 个物品的重量为 `weight[i]`，价值为 `value[i]`。物品可以被分割。求解如何装包使得总价值最大。

样例输入:

- 背包容量 `W = 50`
- 物品 (weight, value):
  - Item 1: (10, 60)
  - Item 2: (20, 100)
  - Item 3: (30, 120)

3.2 思路展示与推演

1. **贪心标准**：选择价值密度 `value/weight` 最高的物品。
2. **计算并排序**：
   - Item 1: 密度 = 60 / 10 = 6.0
   - Item 2: 密度 = 100 / 20 = 5.0
   - Item 3: 密度 = 120 / 30 = 4.0
     排序后的顺序为：Item 1 > Item 2 > Item 3。
3. **决策过程**：
   - **处理 Item 1**：背包剩余容量为50。Item 1重量为10，可完全装入。
     - 装入10kg的Item 1。
     - 背包更新：容量 `50 - 10 = 40`。总价值 `+ 60`。
   - **处理 Item 2**：背包剩余容量为40。Item 2重量为20，可完全装入。

        - 装入20kg的Item 2。
        - 背包更新：容量 `40 - 20 = 20`。总价值 `+ 100`。
    - **处理 Item 3**：背包剩余容量为20。Item 3重量为30，无法完全装入。
        - 只能装入剩余容量允许的部分，即20kg的Item 3。
        - 背包更新：容量 `20 - 20 = 0`。总价值 `+ (20/30) * 120 = 80`。
4. **最终结果**：
    - 背包已满。总价值为 `60 + 100 + 80 = 240`。

---

4. 代码实现

为了实现上述逻辑，我们通常定义一个结构体来表示物品，然后使用C++标准库的 `sort` 配合自定义比较函数来完成排序。

4.1 关键代码

```
#include <bits/stdc++.h>
using namespace std;

// 定义物品结构体
struct Item {
    int m; // 重量
    int v; // 价值
    double ratio; // 价值密度 (性价比)
};

// 自定义比较函数，用于sort排序，按价值密度降序
bool cmp(const Item& a, const Item& b) {
    return a.ratio > b.ratio;
}

int main() {
    int n;
    double t; // 背包容量可能需要参与浮点数运算，用double更方便
    cin >> n >> t;

    vector<Item> items(n);
    for (int i = 0; i < n; ++i) {
        cin >> items[i].m >> items[i].v;
        // 计算价值密度，注意要进行浮点数除法
        items[i].ratio = (double)items[i].v / items[i].m;
    }

    // 按价值密度从高到低排序
    sort(items.begin(), items.end(), cmp);

    double total_value = 0.0;

    // 遍历排序后的物品
    for (int i = 0; i < n; ++i) {
        // 如果背包容量为0，提前结束
        if (t == 0) {
            break;
```

```
        }

        // 如果能装下整个物品
        if (t >= items[i].m) {
            total_value += items[i].v;
            t -= items[i].m;
        } else { // 如果只能装下部分物品
            // 装入部分物品，价值 = 剩余容量 * 价值密度
            total_value += t * items[i].ratio;
            t = 0; // 背包已满
        }
    }

    // 按格式输出，保留两位小数
    cout << fixed << setprecision(2) << total_value << endl;

    return 0;
}
```

4.2 代码解读

- `struct Item`：封装了物品的重量，价值等属性。
- `cmp` / `sort`：这是贪心策略的实现。我们通过提供一个自定义比较函数，让 `sort` 算法按照我们定义的贪心标准（价值密度降序）来排列物品。
- **循环**：循环体内的`if-else`结构实现了贪心决策。首先尝试完整地装入一个物品；如果容量不足，则计算能装入的部分，并将其价值计入总和，随后由于背包已满，循环提前终止。