贪心

本页面将简要介绍贪心算法。

引入

贪心算法（英语：greedy algorithm），是用计算机来模拟一个「贪心」的人做出决策的过程。这个人十分贪婪，每一步行动总是按某种指标选取最优的操作。而且他目光短浅，总是只看眼前，并不考虑以后可能造成的影响。

可想而知，并不是所有的时候贪心法都能获得最优解，所以一般使用贪心法的时候，都要确保自己能证明其正确性。

解释

适用范围

贪心算法在有最优子结构的问题中尤为有效。最优子结构的意思是问题能够分解成子问题来解决，子问题的最优解能递推到最终问题的最优解。[1]

证明

贪心算法有两种证明方法：反证法和归纳法。一般情况下，一道题只会用到其中的一种方法来证明。

1. 反证法：如果交换方案中任意两个元素/相邻的两个元素后，答案不会变得更好，那么可以推定目前的解已经是最优解了。
2. 归纳法：先算得出边界情况（例如 $n = 1$）的最优解 F_1，然后再证明：对于每个 n，F_{n+1} 都可以由 F_n 推导出结果。

要点

常见题型

在提高组难度以下的题目中，最常见的贪心有两种。

- 「我们将 XXX 按照某某顺序排序，然后按某种顺序（例如从小到大）选择。」。
- 「我们每次都取 XXX 中最大/小的东西，并更新 XXX。」（有时「XXX 中最大/小的东西」可以优化，比如用优先队列维护）

二者的区别在于一种是离线的，先处理后选择；一种是在线的，边处理边选择。

排序解法

用排序法常见的情况是输入一个包含几个（一般一到两个）权值的数组，通过排序然后遍历模拟计算的方法求出最优值。

后悔解法

思路是无论当前的选项是否最优都接受，然后进行比较，如果选择之后不是最优了，则反悔，舍弃掉这个选项；否则，正式接受。如此往复。

区别

与动态规划的区别

贪心算法与动态规划的不同在于它对每个子问题的解决方案都做出选择，不能回退。动态规划则会保存以前的运算结果，并根据以前的结果对当前进行选择，有回退功能。

例题详解

邻项交换法的例题

NOIP 2012 国王游戏

恰逢 H 国国庆，国王邀请 n 位大臣来玩一个有奖游戏。首先，他让每个大臣在左、右手上面分别写下一个整数，国王自己也在左、右手上各写一个整数。然后，让这 n 位大臣排成一排，国王站在队伍的最前面。排好队后，所有的大臣都会获得国王奖赏的若干金币，每位大臣获得的金币数分别是：排在该大臣前面的所有人的左手上的数的乘积除以他自己右手上的数，然后向下取整得到的结果。

国王不希望某一个大臣获得特别多的奖赏，所以他想请你帮他重新安排一下队伍的顺序，使得获得奖赏最多的大臣，所获奖赏尽可能的少。注意，国王的位置始终在队伍的最前面。

解题思路

设排序后第 i 个大臣左右手上的数分别为 a_i, b_i。考虑通过邻项交换法推导贪心策略。

用 s 表示第 i 个大臣前面所有人的 a_i 的乘积，那么第 i 个大臣得到的奖赏就是 $\frac{s}{b_i}$，第 $i+1$ 个大臣得到的奖赏就是 $\frac{s \cdot a_i}{b_{i+1}}$。

如果我们交换第 i 个大臣与第 $i+1$ 个大臣，那么此时的第 i 个大臣得到的奖赏就是 $\frac{s}{b_{i+1}}$，第 $i+1$ 个大臣得到的奖赏就是 $\frac{s \cdot a_{i+1}}{b_i}$。

如果交换前更优当且仅当

$$\max\left(\frac{s}{b_i}, \frac{s \cdot a_i}{b_{i+1}}\right) < \max\left(\frac{s}{b_{i+1}}, \frac{s \cdot a_{i+1}}{b_i}\right)$$

提取出相同的 s 并约分得到

$$\max\left(\frac{1}{b_i}, \frac{a_i}{b_{i+1}}\right) < \max\left(\frac{1}{b_{i+1}}, \frac{a_{i+1}}{b_i}\right)$$

然后分式化成整式得到

$$\max(b_{i+1}, a_i \cdot b_i) < \max(b_i, a_{i+1} \cdot b_{i+1})$$

实现的时候我们将输入的两个数用一个结构体来保存并重载运算符：

```
struct uv {
  int a, b;

  bool operator<(const uv &x) const {
    return max(x.b, a * b) < max(b, x.a * x.b);
  }
};
```

后悔法的例题

「USACO09OPEN」工作调度 Work Scheduling

约翰的工作日从 0 时刻开始，有 10^9 个单位时间。在任一单位时间，他都可以选择编号 1 到 N 的 $N(1 \le N \le 10^5)$ 项工作中的任意一项工作来完成。工作 i 的截止时间是 $D_i(1 \le D_i \le 10^9)$，完成后获利是 $P_i(1 \le P_i \le 10^9)$。在给定的工作利润和截止时间下，求约翰能够获得的利润最大为多少。

解题思路

1. 先假设每一项工作都做，将各项工作按截止时间排序后入队；
2. 在判断第 `i` 项工作做与不做时，若其截至时间符合条件，则将其与队中报酬最小的元素比较，若第 `i` 项工作报酬较高（后悔），则 `ans += a[i].p - q.top()`。
   用优先队列（小根堆）来维护队首元素最小。
3. 当 `a[i].d<=q.size()` 可以这么理解从 0 开始到 `a[i].d` 这个时间段只能做 `a[i].d` 个任务，而若 `q.size()>=a[i].d` 说明完成 `q.size()` 个任务时间大于等于 `a[i].d` 的时间，所以当第 `i` 个任务获利比较大的时候应该把最小的任务从优先级队列中换出。

参考代码

C++

```cpp
#include <algorithm>
#include <cmath>
#include <cstring>
#include <iostream>
#include <queue>
using namespace std;

struct f {
  long long d;
  long long p;
} a[100005];

bool cmp(f A, f B) { return A.d < B.d; }

// 小根堆维护最小值
priority_queue<long long, vector<long long>, greater<long long>>
q;

int main() {
  long long n, i;
  cin >> n;
  for (i = 1; i <= n; i++) {
    cin >> a[i].d >> a[i].p;
  }
  sort(a + 1, a + n + 1, cmp);
  long long ans = 0;
  for (i = 1; i <= n; i++) {
    if (a[i].d <= (int)q.size()) {  // 超过截止时间
      if (q.top() < a[i].p) {       // 后悔
        ans += a[i].p - q.top();
        q.pop();
        q.push(a[i].p);
      }
    } else {  // 直接加入队列
      ans += a[i].p;
      q.push(a[i].p);
    }
  }
  cout << ans << endl;
  return 0;
}
```

Python

```python
from collections import defaultdict
from heapq import heappush, heappop

```

```python
a = defaultdict(list)
for _ in range(int(input())):
    d, p = map(int, input().split())
    a[d].append(p)  # 存放对应时间的收益

ans = 0  # 记录总收益
q = []  # 小根堆维护最小值
l = sorted(a.keys(), reverse=True)
for i, j in zip(l, l[1:] + [0]):
    for k in a.pop(i):
        heappush(q, ~k)
    for _ in range(i - j):
        if q:  # 从堆中取出收益最多的工作
            ans += ~heappop(q)
        else:  # 堆为空时退出循环
            break
print(ans)
```

复杂度分析

- 空间复杂度：当输入 n 个任务时使用 n 个 a 数组元素，优先队列中最差情况下会储存 n 个元素，则空间复杂度为 $O(n)$。
- 时间复杂度：`std::sort` 的时间复杂度为 $O(n \log n)$，维护优先队列的时间复杂度为 $O(n \log n)$，综上所述，时间复杂度为 $O(n \log n)$。

习题

- P1209[USACO1.3] 修理牛棚 Barn Repair - 洛谷
- P2123 皇后游戏 - 洛谷
- LeetCode 上标签为贪心算法的题目

参考资料与注释

1. 贪心算法 - 维基百科，自由的百科全书 ↩