第 1 章 基础算法（下）

8. 贪心

贪心算法是**两极分化很严重的算法**。简单的问题会让你觉得理所应当，难一点的问题会让你怀疑人生。

1. 什么是贪心算法？

> 贪心算法，或者说是贪心策略：企图用局部最优找出全局最优。
>
> 1. 把解决问题的过程分成若干步；
> 2. 解决每一步时，都选择"当前看起来最优的"解法；
> 3. "希望"得到全局的最优解。

2. 贪心算法的特点

> 1. 对于大多数题目，贪心策略的提出并不是很难，难的是证明它是正确的。因为贪心算法相较于暴力枚举，每一步并不是把所有情况都考虑进去，而是只考虑当前看起来最优的情况。但是，局部最优并不等于全局最优，所以我们必须要能严谨的证明我们的贪心策略是正确的。
>
>   一般证明策略有：反证法，数学归纳法，交换论证法等等。
> 2. 当问题的场景不同时，贪心的策略也会不同。因此，贪心策略的提出是没有固定的套路和模板的。我们后面讲的题目虽然分类，但是大家会发现具体的策略还是相差很大。
>
>   因此，不要妄想做几道贪心题目就能遇到一个会一个。有可能做完 50 道贪心题目之后，第 51 道还是没有任何思路。

3. 如何学习贪心？

> 先有一个认知：做了几十道贪心的题目，遇到一个新的又没有思路，这时很正常的现象，把心态放平。
>
> 1. 前期学习的时候，重点放在各种各样的策略上，把各种策略当成经验来吸收；
> 2. 在平常学习的时候，我们尽可能的证明一下这个贪心策略是否正确，这样有利于培养我们严谨的思维。但是在比赛中，能想出来一个策略就已经不错了，如果再花费大量的时间去证明，有点得不偿失。这个时候，如果根据贪心策略想出来的若干个边界情况都能过的话，就可以尝试去写代码了。

8.1 简单贪心

8.1.1 货仓选址

> **题目来源：** 洛谷
>
> **题目链接：** P10452 货仓选址
>
> **难度系数：** ★

【题目描述】

> 在一条数轴上有 n 家商店，它们的坐标分别为 $a_1, a_2, ..., a_n$ 。
>
> 现在需要在数轴上建立一家货仓，每天清晨，从货仓到每家商店都要运送一车商品。
>
> 为了提高效率，求把货仓建在何处，可以使得货仓到每家商店的距离之和最小。

【输入描述】

> 第一行一个整数 n ，第二行 n 个整数 $a_1, a_2, ..., a_n$
>
> 对于 100% 的数据: $N \leq 100000, a_i \leq 1000000$

【输出描述】

> 一个整数，表示距离之和的最小值。

【示例一】

> 输入：
>
> ```
> 4
> 6 2 9 1
> ```
>
> 输出：
>
> ```
> 12
> ```

【解法】

将所有的商店按照「从小到大」的顺序「排序」，把货仓建在中位数处，可以使得货仓到每家商店的距离之和最小：

- 如果 n 是奇数，货仓建在 $a[(n+1)/2]$ 位置处；
- 如果 n 是偶数，货仓建在 $a[n/2] \sim a[n/2+1]$ 之间都是可以的

拓展：从这道题中我们可以得出一个结论（往后做题的时候可能会用到）：

> 形如： $sum = \sum_{i=1}^{n} |a[i] - x| = |a[1] - x| + |a[2] - x| + ... + |a[n] - x|$ 这样一个式子：
>
> - 当 x 取到 n 个数的中位数时，和最小；
> - 最小和为： $(a[n] - a[1]) + (a[n-1] - a[2]) + ... + (a[n+1-n/2] - a[n/2])$ 。

【参考代码】

代码块

```cpp
#include <iostream>
#include <algorithm>

using namespace std;

typedef long long LL;

const int N = 1e5 + 10;

int n;
LL a[N];

int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> a[i];
    sort(a + 1, a + 1 + n);

    LL ret = 0;
    // 利用中间值来计算
    // for(int i = 1; i <= n; i++)
    // {
    //     ret += abs(a[i] - a[n / 2]);
    // }

    // 用结论计算
    for(int i = 1; i <= n / 2; i++)
    {
        ret += a[n - i + 1] - a[i];
    }

    cout << ret << endl;

    return 0;
}
```

8.1.2 最大子段和

> **题目来源：** 洛谷
>
> **题目链接：** P1115 最大子段和

> **难度系数：** ★★

【题目描述】

> 给出一个长度为 n 的序列 a ，选出其中连续且非空的一段使得这段和最大。

【输入描述】

> 第一行是一个整数，表示序列的长度 n 。
>
> 第二行有 n 个整数，第 i 个整数表示序列的第 i 个数字 a_i 。
>
> - 对于 40% 的数据，保证 $n \leq 1 \times 10^3$ 。
> - 对于 100% 的数据，保证 $1 \leq n \leq 2 \times 10^5, -10^4 \leq a_i \leq 10^4$ 。

【输出描述】

> 输出一行一个整数表示答案。

【示例一】

> 输入：
>
> 7
>
> 2 -4 3 -1 2 -4 3
>
> 输出：
>
> 4

【解法】

有没有似曾相识的感觉？这是我们第二次遇见它了，但还不是最后一次~

贪心想法：从前往后累加，我们会遇到下面两种情况：

- 目前的累加和 ≥ 0 ：那么当前累加和还会对后续的累加和做出贡献，那我们就继续向后累加，然后更新结果；
- 目前的累加和 < 0 ：对后续的累加和做不了一点贡献，直接大胆舍弃计算过的这一段，把累加和重置为 0 ，然后继续向后累加。

这样我们在扫描整个数组一遍之后，就能更新出最大子段和。

如果是善于思考的同学此时就会有「一些」大大的疑惑了， $why?\ why?\ why?$ 为什么可以得到「最优解」？怎么感觉这个策略是「错」的啊？感觉「好多情况」都没考虑进去，为什么就得到一个正确的结果？为什么可以「大胆舍去」这一段累加和？

如果你有大大的疑惑，这就对了。我刚开始做这道题，看到别人的题解是这样写的时候，也有如此疑惑。（我觉得这就是贪心算法的魅力吧，看似很简单，很玄学，其实有很多值的我们思考的地方）别

着急，我们接下来证明一下这个贪心策略是正确的。

其实只需要证明我们在累加的过程中，出现负数时，为什么可以大胆的舍去这一段区间，然后重新开始。证明以下三点，就可以「大胆舍弃」了：

在累加的过程中算出一段区间和 $sum[a,b]<0$，如果不舍弃这一段，那么 $[a,b]$ 段之间就会存在一点，「以某个位置为起点」就会「更优」，分为下面两种情况：

1. 在 ab 段存在一个点 c，从这个位置开始，「越过 b 」的累加和比从 a 开始的累加和更优：

> 用「反证法」证明这种情况不存在。
>
> 如果存在这一点，那么：$sum[c,b]>sum[a,b]$，这样才能保证向后加的时候更优。
>
> 但这是「不可能」的。如果 $sum[c,b]>sum[a,b]$，那么 $sum[a,c-1]<0$，这与我们的贪心策略矛盾。
>
> 因为我们贪心策略向后加的时候，只要不小于 0，就会一直加下去。如果 $[a,c-1]$ 段小于 0，就会在 c 点之前停止，不会累加到 b。
>
> 因此区间内不存在一点，在计算子数组和时，在「越过 b 」的情况下，能比从 a 开始更优。

2. 在 ab 段存在一个点 c，从这个位置开始，「不越过 b 」的累加和比从 a 开始的累加和更优：

> 也可以用「反证法」证明这种情况不存在。
>
> 如果存在这一点，那么：$sum[c,k]>sum[a,k]$。
>
> 但这是不可能的。如果 $sum[c,k]>sum[a,k]$，那么 $sum[a,c-1]<0$，这与我们的贪心策略矛盾。
>
> 因此区间内不存在一点，在计算子数组和时，在「不越过 b 」的情况下，能比从 a 开始更优。

综上所述，我们可以大胆舍弃这一段，重新开始。

【参考代码】

代码块

```
#include <iostream>

using namespace std;

typedef long long LL;
```

```cpp

const int N = 2e5 + 10;

int n;
LL a[N];

int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> a[i];

    LL sum = 0, ret = -1e6;
    for(int i = 1; i <= n; i++)
    {
        sum += a[i];
        ret = max(ret, sum);
        if(sum < 0) sum = 0;
    }

    cout << ret << endl;

    return 0;
}
```

8.1.3 纪念品分组

题目来源： 洛谷

题目链接： P1094 [NOIP2007 普及组] 纪念品分组

难度系数： ★

【题目描述】

元旦快到了，校学生会让乐乐负责新年晚会的纪念品发放工作。为使得参加晚会的同学所获得 的纪念品价值相对均衡，他要把购来的纪念品根据价格进行分组，但每组最多只能包括两件纪念品， 并且每组纪念品的价格之和不能超过一个给定的整数。为了保证在尽量短的时间内发完所有纪念品，乐乐希望分组的数目最少。

你的任务是写一个程序，找出所有分组方案中分组数最少的一种，输出最少的分组数目。

【输入描述】

共 $n+2$ 行：

第一行包括一个整数 w，为每组纪念品价格之和的上限。

第二行为一个整数 n，表示购来的纪念品的总件数 G。

> 第 $3 \sim n+2$ 行每行包含一个正整数 p_i 表示所对应纪念品的价格。
>
> 100% 的数据满足： $1 \leq n \leq 3 \times 10^4, 80 \leq w \leq 200, 5 \leq p_i \leq w$ 。

【输出描述】

> 一个整数，即最少的分组数目。

【示例一】

> 输入：
>
> 100
> 9
> 90
> 20
> 20
> 30
> 50
> 60
> 70
> 80
> 90
>
> 输出：
>
> 6

【解法】

先将所有的纪念品排序，每次拿出当前的最小值 x 与最大值 y ：

- 如果 $x+y \leq w$ ：就把这两个放在一起；
- 如果 $x+y>w$ ：说明此时最大的和谁都凑不到一起， y 单独分组， x 继续留下在进行下一次判断。

直到所有的物品都按照上述规则分配之后，得到的组数就是最优解。

可以用「交换论证法」证明贪心解就是最优解：

> 对于区间 $[a_i.....a_j]$ ，如果存在最优解，但是 a_i 与 a_j 的分配方式与我们贪心解的分配方式不一样，那么就会有以下几种情况：
>
> 1. 当 $a[i]+a[j]>w$ 时：

- 贪心解会把 $a[j]$ 单独分组， $a[i]$ 留待下次考虑；
- 最优解也必定会把 $a[j]$ 单独分组，因为没有更小的值与 $a[j]$ 组合。

此时贪心解与最优解一致。

2. 当 $a[i]+a[j]\le w$ 时：

- 贪心解会把两者组合分在一个组里面；
- 最优解可能有以下几种情况：
  - $a[j]$ 单独一组：
    - 如果 $a[i]$ 也单独一组的话，最优解还不如贪心解分的组少，矛盾；
    - 如果 $a[i]$ 和另一个 $a[k]$ 一组的话，我们可以把 $a[k]$ 与 $a[j]$ 交换，此时并不影响结果，和贪心解一致。
  - $a[j]$ 和 $a[k]$ 一组：
    - 如果 $a[i]$ 单独一组的话，交换 $a[i]$ 和 $a[k]$，此时并不影响最终结果，和贪心解一致；
    - 如果 $a[i]$ 和 $a[l]$ 一组的话，交换 $a[i]$ 和 $a[k]$，此时变成 $(a[i]+a[j]),(a[l]+a[k])$，其中 $a[l]+a[k]\le a[j]+a[k]\le w$，不影响最终结果，和贪心解一致。

综上所述，我们可以通过不断的「调整」，使的最优解在「不改变其最优性」的前提下，变得和贪心解一致。那我们的贪心策略就等价于最优策略。

【参考代码】

代码块

```cpp
#include <iostream>
#include <algorithm>

using namespace std;

const int N = 3e4 + 10;

int w, n;
int a[N];

int main()
{
    cin >> w >> n;
    for(int i = 1; i <= n; i++) cin >> a[i];
    sort(a + 1, a + 1 + n);

    int l = 1, r = n, ret = 0;
```

```
18      while(l <= r)
19      {
20          if(a[l] + a[r] <= w) l++, r--;
21          else r--;
22          ret++;
23      }
24      cout << ret << endl;
25
26      return 0;
27  }
```

8.1.4 排座椅

题目来源： 洛谷

题目链接： P1056 [NOIP2008 普及组] 排座椅

难度系数： ★★

【题目描述】

上课的时候总有一些同学和前后左右的人交头接耳，这是令小学班主任十分头疼的一件事情。不过，班主任小雪发现了一些有趣的现象，当同学们的座次确定下来之后，只有有限的D对同学上课时会交头接耳。

同学们在教室中坐成了 M 行 N 列，坐在第 i 行第 j 列的同学的位置是 (i,j)，为了方便同学们进出，在教室中设置了 K 条横向的通道， L 条纵向的通道。

于是，聪明的小雪想到了一个办法，或许可以减少上课时学生交头接耳的问题：她打算重新摆放桌椅，改变同学们桌椅间通道的位置，因为如果一条通道隔开了两个会交头接耳的同学，那么他们就不会交头接耳了。

请你帮忙给小雪编写一个程序，给出最好的通道划分方案。在该方案下，上课时交头接耳的学生对数最少。

【输入描述】

第一行，有 5 各用空格隔开的整数，分别是
$M,N,K,L,D(2 \le N,M \le 1000, 0 \le K < M, 0 \le L < N, D \le 2000)$

接下来 D 行，每行有 4 个用空格隔开的整数，第 i 行的 4 个整数 X_i,Y_i,P_i,Q_i，表示坐在位置 (X_i,Y_i) 与 (P_i,Q_i) 的两个同学会交头接耳（输入保证他们前后相邻或者左右相邻）。

输入数据保证最优方案的唯一性。

【输出描述】

共两行

第一行包含 K 个整数， $a_1, a_2, ..., a_K$ ，表示第 a_1 行和 a_1+1 行之间、第 a_2 行和第 a_2+1 行之间、…、第 a_K 行和第 a_K+1 行之间要开辟通道，其中 $a_i < a_{i+1}$ ，每两个整数之间用空格隔开（行尾没有空格）。

第二行包含 L 个整数， $b_1, b_2, ..., b_L$ ，表示第 b_1 行和 b_1+1 行之间、第 b_2 行和第 b_2+1 行之间、…、第 b_K 行和第 b_L+1 行之间要开辟通道，其中 $b_i < b_{i+1}$ ，每两个整数之间用空格隔开（行尾没有空格）。

【示例一】

输入：

```
5 1 2 3
4 2 4 3
2 3 3 3
2 5 2 4
```

输出：

```
2
2 4
```

【解法】

由题意可得，我们会发现一些性质：

- 设置横向通道的时候，并「不影响」左右相邻的同学；
- 设置纵向通道的时候，并「不影响」上下相邻的同学。

因此我们可以「分开」处理横向通道和纵向通道。

处理横向通道（纵向同理，就不多赘述）：

- 收集每一行如果放上通道之后，会解决多少个交头接耳的同学；
- 对收集的信息「从大到小」排序，选最大的 k 行就是最优结果。

不需要证明了吧……我们都把最大的 k 个拿出来了，如果还不是最优的话找谁说理去。

【参考代码】

代码块

```cpp
#include <iostream>
#include <algorithm>

```

```cpp
using namespace std;

const int N = 1010;

struct node
{
    int index;
    int cnt;
}row[N], col[N];

int m, n, k, l, d;

// 按照 cnt 从大到小排序
bool cmp1(node& x, node& y)
{
    return x.cnt > y.cnt;
}

// 按照 index 从小到大排序
bool cmp2(node& x, node& y)
{
    return x.index < y.index;
}

int main()
{
    cin >> m >> n >> k >> l >> d;
    // 初始化结构体数组
    for(int i = 1; i <= m; i++) row[i].index = i;
    for(int i = 1; i <= n; i++) col[i].index = i;

    while(d--)
    {
        int x, y, p, q; cin >> x >> y >> p >> q;
        if(x == p) col[min(y, q)].cnt++;
        else row[min(x, p)].cnt++;
    }

    // 对两个数组按照 cnt 从大到小排序
    sort(row + 1, row + 1 + m, cmp1);
    sort(col + 1, col + 1 + n, cmp1);

    // 对 row 数组，前 k 个元素，按照下标从小到大排序
    sort(row + 1, row + 1 + k, cmp2);
    // 对 col 数组，前 l 个元素，按照下标从小到大排序
    sort(col + 1, col + 1 + l, cmp2);

```

```
51      for(int i = 1; i <= k; i++)
52      {
53          cout << row[i].index << " ";
54      }
55      cout << endl;
56      for(int i = 1; i <= l; i++)
57      {
58          cout << col[i].index << " ";
59      }
60      cout << endl;
61
62      return 0;
63  }
```

8.1.5 矩阵消除游戏

> **题目来源：** 牛客网
>
> **题目链接：** 矩阵消除游戏
>
> **难度系数：** ★★★

【题目描述】

> 牛妹在玩一个名为矩阵消除的游戏，矩阵的大小是 n 行 m 列，第 i 行第 j 列的单元格的权值为 $a_{i,j}$ ，牛妹可以进行 k 个回合的游戏，在每个回合，牛妹可以选择一行或者选择一列，然后将这一行或者这一列的所有单元格中的权值变为 0 ，同时牛妹的分数会加上这一行或者这一列中的所有单元格的权值的和。
>
> 牛妹想最大化她的得分，球球你帮帮她吧！

【输入描述】

> 第一行三个整数 n, m, k
> 接下来 n 行每行 m 个整数表示矩阵中各个单元格的权值
>
> $1 \leq n, m \leq 15$
>
> $1 \leq a_{i,j} \leq 10^6$
>
> $1 \leq k \leq n * m$

【输出描述】

> 输出一个整数表示牛妹能获得的最大分数。

【示例一】

> 输入：

> 3 3 2
>
> 101 1 102
>
> 1 202 1
>
> 100 8 100
>
> 输出：
>
> 414

【解法】

错误的贪心：每次都选当前看起来最好的一行或者一列，然后选完之后，修改原数组，继续重复刚刚的操作。

> 反例，当 $k = 2$ 时：
>
> ```
> 100  9   10
> ```
>
> ```
> 0    0    0
> ```
>
> ```
> 100  0   10
> ```
>
> 如果是贪心的话，会选第一列和最后一列。但是最优情况下我们可以把所有的数都选上，因此直接上来就贪心不是最优解。

直接贪心的错误在于，我们每次选完「一行」或者「一列」之后，会对接下来的「列」或者「行」的选择造成影响。比如上面的反例：

- 我们先贪心的选了第一列。站在「行」的视角上，在「列」没选之前，我分明能贪心的选「第一行」，但是你把「第一列」拿走哦之后，就会对「第一行」造成影响。

既然行的选择会影响列，那我们干脆直接把「所有行的选法」枚举出来，然后针对「每一种行的选法」再处理列，这样就会把「所有情况」都考虑进去。

因此，最优解是先暴力枚举所有行的选法，在行的选择都确定之后，再去贪心的处理列。这个思想我们是不陌生的，参考二进制枚举里面《费解的开关》。

【参考代码】

代码块

```cpp
#include <iostream>
#include <algorithm>
#include <cstring>

using namespace std;

const int N = 20;
```

```cpp

int n, m, k;
int a[N][N];
int col[N]; // 统计列和

// 统计 x 的二进制表示中 1 的个数
int calc(int x)
{
    int ret = 0;
    while(x)
    {
        ret++;
        x -= x & -x;
    }
    return ret;
}

// 按照值从大到小排序
bool cmp(int a, int b)
{
    return a > b;
}

int main()
{
    cin >> n >> m >> k;
    for(int i = 0; i < n; i++)
        for(int j = 0; j < m; j++)
            cin >> a[i][j];

    int ret = 0;
    // 暴力枚举出行的所有选法
    for(int st = 0; st < (1 << n); st++)
    {
        int cnt = calc(st);
        if(cnt > k) continue; // 不合法的状态

        memset(col, 0, sizeof col);
        int sum = 0; // 记录当前选法中的和
        for(int i = 0; i < n; i++)
        {
            for(int j = 0; j < m; j++)
            {
                if((st >> i) & 1) sum += a[i][j];
                else col[j] += a[i][j];
            }
        }
```

```
55
56          // 处理列
57          sort(col, col + m, cmp);
58          // 选 k - cnt 列
59          for(int i = 0; i < k - cnt; i++) sum += col[i];
60          ret = max(ret, sum);
61      }
62
63      cout << ret << endl;
64
65      return 0;
66  }
```

8.2 推公式

如果细说的话，这个专题应该叫**推公式+排序**。其中推公式就是**寻找排序规则**，排序就是在该排序规则下**对整个对象排序**。

在解决某些问题的时，当我们发现**最终结果需要调整每个对象的先后顺序**，也就是对整个对象排序时，那么我们就可以用推公式的方式，得出我们的排序规则，进而对整个对象排序。

正确性证明：

> 利用排序解决问题，最重要的就是需要证明"**在新的排序规则下，整个集合可以排序**"。这需要用到**离散数学中"全序关系"的知识**。我会在第一道题中证明该题的排序规则下，整个集合是可以排序的。
>
> 但是证明过程很麻烦，后续题目中我们**只要发现该题最终结果需要排序，并且交换相邻两个元素的时候，对其余元素不会产生影响**，那么我们就可以推导出排序的规则，然后直接去排序，就不去证明了。

8.2.1 拼数

> **题目来源：** 洛谷
>
> **题目链接：** P1012 [NOIP1998 提高组] 拼数
>
> **难度系数：** ★★

【题目描述】

> 设有 n 个正整数 $a_1, a_2, ..., a_n$ ，将它们联接成一排，相邻数字首尾相接，组成一个最大的整数。

【输入描述】

> 第一行有一个整数，表示数字个数 n 。
>
> 第二行有 n 个整数，表示给出的 n 个整数 a_i 。

> 对于全部的测试点，保证 $1 \leq n \leq 20$，$1 \leq a_i \leq 10^9$

【输出描述】

> 一个正整数，表示最大的整数

【示例一】

> 输入：
>
> 3
>
> 13 312 343
>
> 输出：
>
> 34331213

【解法】

我们发现，任取序列里面相邻的两项 $a[i], a[i+1]$，交换他们的顺序，并不影响 $[1, i-1]$ 与 $[i+1, n]$ 之间每一位的权值。因此我们可以找一种比较方式，对整个数组排序，最终的结果就是最优序列。

设两个相邻的数对应的字符串形式为 x, y，因为要的是最大值，所以自定义比较方式：

- $x+y > y+x$： x 放在前面， y 放在后面；
- $x+y < y+x$： y 放在前面， x 放在后面；
- $x+y = y+x$：谁前谁后无所谓。

【参考代码】

代码块

```
#include <iostream>
#include <algorithm>

using namespace std;

const int N = 25;

int n;
string a[N];

bool cmp(string& x, string& y)
{
    return x + y > y + x;
```

```cpp
}

int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> a[i];

    // 排序
    sort(a + 1, a + 1 + n, cmp);

    for(int i = 1; i <= n; i++) cout << a[i];

    return 0;
}
```

8.2.2 保卫花园

> **题目来源：** 洛谷
>
> **题目链接：** [USACO07JAN] Protecting the Flowers S
>
> **难度系数：** ★★★

【题目描述】

> 有 n 头奶牛跑到 FJ 的花园里去吃花儿了，它们分别在距离牛圈 T_i （这里指 FJ 到那里需要 T_i 分钟) 处吃花，每分钟会吃掉 D_i 朵花，FJ 现在要将它们给弄回牛圈，但是他每次只能弄一头回去，来回用时总共为 $2 \times T_i$ 分钟，在这段时间内，其它的奶牛会继续吃 FJ 的花，速度保持不变，当然正在被赶回牛圈的奶牛不能继续吃了。现在求在最好的方案下奶牛吃掉花的最小朵数。

【输入描述】

> 第一行：单个整数 n
>
> 后面 n 行：每行包含两个空格分隔的整数 T_i 和 D_i ，它们描述了一头奶牛的特征

【输出描述】

> 单独一行: 单个整数，表示被破坏的花的最小数量

【示例一】

> 输入：
>
> 6
>
> 3 1
>
> 2 5
>
> 2 3

```
3 2
4 1
1 6
```

输出：

```
86
```

【解法】

我们发现，在在一个序列中，任意交换相邻两头牛 $a[i], a[i+1]$ 的顺序之后，区间 $[1, i-1]$ 以及 $[i+1, n]$ 内所有牛吃草的总量不变。因此我们可以找一种比较方式，对整个数组排序，最终的结果就是最优序列。

设交换的相邻两头牛的下标为 i, j $(i+1=j)$，到达 i 位置时，经过的时间为 T。由此可得：

- 交换前：

  第 i 头牛吃草量： $T \times d_i$；

  第 j 头牛的吃草量： $(T + 2 \times t_i) \times d_j$

  消耗的总时长： $sum_1 = Td_i + Td_j + 2t_id_j$

- 交换后：

  第 j 头牛吃草量： $T \times d_j$；

  第 i 头牛的吃草量： $(T + 2 \times t_j) \times d_i$

  消耗的总时长： $sum_2 = Td_j + Td_i + 2t_jd_i$

如果 i 在 j 前比较好，对应的 sum_1 应该小于 sum_2，也就是 $t_id_j < t_jd_i$。

综上，我们自定义的比较方式为：

- 当 $t_id_j < t_jd_i$ 时： i 在 j 前；
- 当 $t_id_j > t_jd_i$ 时： i 在 j 后；
- 当 $t_id_j = t_jd_i$ 时：谁前谁后无所谓。

【参考代码】

代码块

```cpp
#include <iostream>
#include <algorithm>

using namespace std;
```

```cpp

typedef long long LL;

const int N = 1e5 + 10;

int n;
struct node
{
    int t;
    int d;
}a[N];

bool cmp(node& x, node& y)
{
    return x.t * y.d < y.t * x.d;
}

int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++)
    {
        cin >> a[i].t >> a[i].d;
    }

    sort(a + 1, a + 1 + n, cmp);

    LL ret = 0, t = 0;
    for(int i = 1; i <= n; i++)
    {
        ret += a[i].d * t;
        t += 2 * a[i].t;
    }

    cout << ret << endl;

    return 0;
}
```

8.2.3 奶牛玩杂技

> **题目来源：** 洛谷
>
> **题目链接：** [USACO05NOV] 奶牛玩杂技

> **难度系数：** ★★★

【题目描述】

> Farmer John 养了 N 头牛，她们已经按 $1 \sim N$ 依次编上了号。FJ 所不知道的是，他的所有牛都梦想着从农场逃走，去参加马戏团的演出。可奶牛们很快发现她们那笨拙的蹄子根本无法在钢丝或晃动的的秋千上站稳（她们还尝试过把自己装在大炮里发射出去，但可想而知，结果是悲惨的）。最终，她们决定练习一种最简单的杂技：把所有牛都摞在一起， 比如说， 第一头牛站在第二头的身上， 同时第二头牛又站在第三头牛的身上...最底下的是第 N 头牛。
>
> 每头牛都有自己的体重以及力量，编号为 i 的奶牛的体重为 W_i ，力量为 S_i 。
>
> 当某头牛身上站着另一些牛时它就会在一定程度上被压扁，我们不妨把它被压扁的程度叫做它的压扁指数。对于任意的牛，她的压扁指数等于摞在她上面的所有奶牛的总重（当然不包括她自己）减去它的力量。奶牛们按照一定的顺序摞在一起后， 她们的总压扁指数就是被压得最扁的那头奶牛的压扁指数。
>
> 你的任务就是帮助奶牛们找出一个摞在一起的顺序，使得总压扁指数最小。

【输入描述】

> 第一行一个整数 N 。
>
> 接下来 N 行，每行两个整数 W_i 和 S_i 。
>
> 对于 100% 的数据， $1 \leq N \leq 5 \times 10^4, 1 \leq W_i \leq 10^4, 1 \leq S_i \leq 10^9$ 。

【输出描述】

> 一行一个整数表示最小总压扁指数。

【示例一】

> 输入：
>
> ```
> 3
> 10 3
> 2 5
> 3 3
> ```
>
> 输出：
>
> ```
> 2
> ```

【解法】

我们发现，在在一个序列中，任意交换相邻两头牛 $a[i], a[i+1]$ 的顺序之后，区间 $[1, i-1]$ 以及 $[i+1, n]$ 内每一头牛的压扁指数都是不变。因此我们可以找一种比较方式，对整个数组排序，最终的结果就是最优序列。

设交换的相邻两头牛的下标为 i, j $(i+1=j)$，$[1, i-1]$ 区间内所有奶牛的重量为 W。由此可得：

- 交换前：

  第 i 头牛的压扁指数为：$W - s_i$；

  第 j 头牛的压扁指数为：$W + w_i - s_j$；

  被压得最扁的那头奶牛的压扁指数：$k_1 = max(W - s_i,\ W + w_i - s_j)$

- 交换后：

  第 j 头牛的压扁指数为：$W - s_j$；

  第 i 头牛的压扁指数为：$W + w_j - s_i$；

  被压得最扁的那头奶牛的压扁指数：$k_2 = max(W - s_j,\ W + w_j - s_i)$

由于 $k1, k2$ 里面都包含 W，可以统一删除。如果 i 在比 j 在前比较好，也就是总体的压扁指数要尽可能的小。那么得要求 $k1 < k2$，也就是：$max(-s_i, w_i - s_j) < max(-s_j, w_j - s_i)$。

【参考代码】

代码块

```
#include <iostream>
#include <algorithm>

using namespace std;

typedef long long LL;

const int N = 5e4 + 10;

int n;
struct node
{
    int w, s;
}a[N];

bool cmp(node& i, node& j)
{
    return i.w + i.s < j.w + j.s;
}

int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> a[i].w >> a[i].s;
```

```
25
26      sort(a + 1, a + 1 + n, cmp);
27
28      LL ret = -1e9 - 10, w = 0;
29      for(int i = 1; i <= n; i++)
30      {
31          ret = max(ret, w - a[i].s);
32          w += a[i].w;
33      }
34      cout << ret << endl;
35
36      return 0;
37  }
```

8.3 哈夫曼编码

> 主要记一下流程。

8.3.1 哈夫曼编码

> **题目来源：** 牛客网
>
> **题目链接：** 【模板】哈夫曼编码
>
> **难度系数：** ★★

【题目描述】

> 给出一个有 n 种字符组成的字符串，其中第 i 种字符出现的次数为 a_i。请你对该字符串应用哈夫曼编码，使得该字符串的长度尽可能短，求编码后的字符串的最短长度。

【输入描述】

> 第一行输入一个整数 $n(1 \le n \le 2 \times 10^5)$，表示字符种数。
>
> 第二行输入 n 个整数 $a_i(1 \le a_i \le 10^9)$，表示每种字符的出现次数。

【输出描述】

> 输出一行一个整数，表示编码后字符串的最短长度。

【示例一】

> 输入：
>
> 3
>
> 1 2 3

> 输出：
>
> 9

【解法】

每次拿出权值最小的两颗树合并，然后将合并后的树继续放回集合中，直到集合中只剩下一棵树。

【参考代码】

代码块

```cpp
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

typedef long long LL;

const int N = 2e5 + 10;

int n;
priority_queue<LL, vector<LL>, greater<LL>> heap; // 小根堆

int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++)
    {
        LL x; cin >> x;
        heap.push(x);
    }

    LL len = 0;
    while(heap.size() > 1)
    {
        // 每次拿出权值最小的两棵树合并
        LL x = heap.top(); heap.pop();
        LL y = heap.top(); heap.pop();
        LL t = x + y;
        len += t;
        heap.push(t);
    }
    cout << len << endl;

    return 0;
}
```

8.3.2 合并果子

> **题目来源：** 洛谷
>
> **题目链接：** P1090 [NOIP2004 提高组] 合并果子
>
> **难度系数：** ★★

【题目描述】

> 在一个果园里，多多已经将所有的果子打了下来，而且按果子的不同种类分成了不同的堆。多多决定把所有的果子合成一堆。
>
> 每一次合并，多多可以把两堆果子合并到一起，消耗的体力等于两堆果子的重量之和。可以看出，所有的果子经过 n-1 次合并之后，就只剩下一堆了。多多在合并果子时总共消耗的体力等于每次合并所耗体力之和。
>
> 因为还要花大力气把这些果子搬回家，所以多多在合并果子时要尽可能地节省体力。假定每个果子重量都为 1，并且已知果子的种类数和每种果子的数目，你的任务是设计出合并的次序方案，使多多耗费的体力最少，并输出这个最小的体力耗费值。 例如有 3 种果子，数目依次为 1，2，9。可以先将 1、2 堆合并，新堆数目为 3，耗费体力为 3。接着，将新堆与原先的第三堆合并，又得到新的堆，数目为 12，耗费体力为 12。所以多多总共耗费体力 = 3+12=15。可以证明 15 为最小的体力耗费值。

【输入描述】

> 输入包括两行，第一行是一个整数 $n(1 \leq n \leq 10000)$，表示果子的种类数。第二行包含 n 个整数，用空格分隔，第 i 个整数 $a_i(1 \leq a_i \leq 20000)$ 是第 i 种果子的数目。

【输出描述】

> 输出包括一行，这一行只包含一个整数，也就是最小的体力耗费值。输入数据保证这个值小于 231。

【示例一】

> 输入：
>
> ```
> 3
> 1 2 9
> ```
>
> 输出：
>
> ```
> 15
> ```

【解法】

每次拿出最小的两堆合并。

【参考代码】

代码块

```cpp
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

typedef long long LL;

int n;
priority_queue<LL, vector<LL>, greater<LL>> heap;

int main()
{
    cin >> n;
    for(int i = 0; i < n; i++)
    {
        LL x;
        cin >> x;
        heap.push(x);
    }

    LL sum = 0;
    while(heap.size() > 1)
    {
        // 取出最小的两堆合并
        LL a = heap.top(); heap.pop();
        LL b = heap.top(); heap.pop();
        heap.push(a + b);
        sum += a + b;
    }

    cout << sum << endl;

    return 0;
}
```

8.4 区间问题

8.4.1 线段覆盖

> **题目来源：** 洛谷
>
> **题目链接：** P1803 凌乱的yyy / 线段覆盖

> **难度系数：** ★★

【题目描述】

> 现在各大 oj 上有 n 个比赛，每个比赛的开始、结束的时间点是知道的。
>
> yyy 认为，参加越多的比赛，noip 就能考的越好（假的）。
>
> 所以，他想知道他最多能参加几个比赛。
>
> 由于 yyy 是蒟蒻，如果要参加一个比赛必须善始善终，而且不能同时参加 2 个及以上的比赛。

【输入描述】

> 第一行是一个整数 n ，接下来 n 行每行是 2 个整数 $a_i, b_i\ (a_i < b_i)$ ，表示比赛开始、结束的时间。
>
> 对于 100% 的数据，$1 \le n \le 10^6, 0 \le a_i < b_i \le 10^6$

【输出描述】

> 一个整数，表示最多参加的比赛数目。

【示例一】

> 输入：
>
> ```
> 3
> 0 2
> 2 4
> 1 3
> ```
>
> 输出：
>
> ```
> 2
> ```

【解法】

按照区间左端点从小到大排序，当两个区间「重叠」的时候，我们必须要舍弃一个。为了能够「在移除某个区间后，保留更多的区间」，我们应该把「区间范围较大」的区间移除。

因此以第一个区间为基准，遍历所有的区间：

- 如果重叠，选择「最小的右端点」作为新的基准；
- 如果不重叠，那么我们就能多选一个区间，以「新区间为基准」继续向后遍历。

可以用「交换论证法」证明我们的贪心策略是最优解：

> 在从前往后扫描的过程中，当贪心解和最优解第一次出现不同决策时，关于两个区间 a, b （其中 a 在左，b 在右）的取舍，有下面两种情况：
>
> 1. a, b 两个区间不重叠：

- 贪心解会将 a 区间保留，然后以 b 区间为基准，继续向后对比别的区间；
- 最优解的选择有下面几种情况：
  a. 舍弃一个区间，那么必定不如贪心解，矛盾。
  b. 以 a 区间为基准，向后对比。那更夸张了，我们已经按照区间左端点从小到大排好序了，如果 a,b 不重叠，那么 a 与后面的所有区间都不重叠，用 a 作为基准没有一点意义。还会选出与 b 重叠的区间。

综上，如果「不重叠」的话，贪心解和最优解的决策应该是「一致」的。

2. a,b 两个区间重叠，那么无论什么解，都需要「舍弃」一个区间：

- 贪心解会保留两者「右端点较小」的区间，舍弃「右端点较大」的区间；
- 最优解的选择就是，保留「右端点较大」的区间，舍弃「右端点较小」的区间。

  如果第二种决策能在此基础上得到最优解，那么我们把「右端点较大」区间换成「右端点较小」的区间是不受影响的。

  因为「较大区间」都和后续选择的区间「没有重叠」，这个较小的区间也必定「没有重叠」。

因此，最优解可以调整成贪心解。

【参考代码】

代码块

```cpp
#include <iostream>
#include <algorithm>

using namespace std;

typedef pair<int, int> PII;

const int N = 1e6 + 10;

int n;
PII a[N];

int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> a[i].first >> a[i].second;
    sort(a + 1, a + 1 + n); // 默然按照首元素比较

    int r = a[1].second, ret = 1;
    for(int i = 2; i <= n; i++)
```

```
21      {
22          int x = a[i].first, y = a[i].second;
23          if(r <= x) // 没有重叠
24          {
25              ret++;
26              r = y;
27          }
28          else // 有重叠，就选右边界最小的那一个
29          {
30              r = min(r, y);
31          }
32      }
33
34      cout << ret << endl;
35
36      return 0;
37  }
```

8.4.2 Radar Installation

题目来源： 洛谷

题目链接： Radar Installation

难度系数： ★★

【题目描述】

假设海岸线是一条无限长的直线，陆地位于海岸线的一边，大海位于海岸线的另一边。大海中有许多小岛。某安全部门为了监视这些岛上是否有敌人入侵，打算在海岸线上安装若干个雷达来检测岛屿的情况。每个雷达的覆盖范围是以雷达中心为圆心，半径为 d 的圆形区域。

我们用平面之间坐标系来表示整个区域，海岸线为 x 轴，大海位于 x 轴上方，陆地位于 x 轴下方。为了节约成本，安全部门想使用最少的雷达覆盖所有的岛屿。现在已知每个岛屿的坐标 (x, y) 和雷达的覆盖半径 d ，你的任务就是计算出能够覆盖所有岛屿的最少雷达数量。

【输入描述】

输入包含若干组数据。

每组数据的第一行有两个整数 $n(1 \leq n \leq 1000)$ 和 d ，分别表示岛屿的数量和雷达的覆盖半径，之后的 n 行，每行有两个整数，表示第 i 个岛屿的坐标 (x_i, y_i) 。

相邻的两组数据使用一个空行隔开。输入 $0\ 0$ 表示输入结束。

【输出描述】

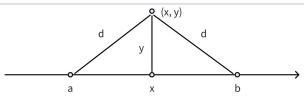
> 对于每一组数据的输出格式为 $Case\ i:\ x$，表示第 i 组数据中最少需要 x 个雷达来覆盖所有的岛屿； $x=-1$ 表示这组数据无解（无法覆盖所有的岛屿）。

【示例一】

> 输入：
>
> 3 2
>
> 1 2
>
> -3 1
>
> 2 1
>
> 1 2
>
> 0 2
>
> 0 0
>
> 输出：
>
> Case 1: 2
>
> Case 2: 1

【解法】

如图所示，当一个岛屿的「坐标」已知，其实可以计算出：当雷达放在 x 轴的「哪段区间」时，可以覆盖到这个岛屿。



根据「勾股定理」得： ax 的长度为 $l = \sqrt{d^2 - y^2}$ ，那么雷达所处的范围就是 $[x - l, x + l]$ 。因此，针对每一个岛屿，我们都可以算出一个「能够覆盖它的区间」。

原问题就变成给定一些区间，从中选取一些区间，能够覆盖掉所有的区间，最少能够选多少个。就从「二维平面」转化成「一维线段」。

按照区间「左端点从小到大」排序，当两个区间「重叠」的时候，为了后面能够「尽可能多的选出互相重叠的区间」，我们应该把「区间范围较大」的区间移除，因为选择较大区间会造成选出来的区间「不是互相重叠」的。

因此以第一个区间为基准，遍历所有的区间：

- 如果重叠，选择「最小的右端点」作为新的基准；
- 如果不重叠，那么我们就能多选一个区间，以「新区间为基准」继续向后遍历。

可以用「反证法」证明，所有区间按照按照「左端点」排序之后，「互相重叠的区间」都是「相邻」的：

> 假设所有区间按照左端点排序之后，存在互相重叠的区间，它们是不相邻的。也就是存在 a, b, c, d 四个区间，其中 a, b, d 互相重叠，但是 a, b, c 与它们三个不是互相重叠。
>
> 设 a, b, d 区间重叠部分的范围是 $[x, y]$ ，那么 c 的位置有两种情况：
>
> - c 在 $[x, y]$ 的左侧，与实际不符：
>
> 因为如果 c 在左侧，又要与 a, b 不是互相重叠，那么 c 的右端点必须要大于 b 的左端点，那就与所有线段按照左端点排序不符；
>
> - c 在 $[x, y]$ 的右侧，也与实际不符：
>
> 因为如果 c 在右侧，又要与 a, b 不是互相重叠，那么 c 的左端点必须要大于 a, b 的右端点的最小值；又因为 d 与 a, b 互相重叠， d 的左端点就要小于 a, b 的右端点的最小值，与所有区间按照左端点排序不符。
>
> 综上所述，所有区间按照按照「左端点」排序之后，「互相重叠的区间」都是「相邻」的。

大家注意，虽然这道题和《线段覆盖》这道题处理区间的方式是一样的，但是「贪心的策略」是不一样的。

【参考代码】

代码块

```
#include <iostream>
#include <cmath>
```

```cpp
#include <algorithm>

using namespace std;

const int N = 1010;

int n, d;
struct node
{
    double l, r;
}a[N];

bool cmp(node& x, node& y)
{
    return x.l < y.l;
}

int main()
{
    int cnt = 0;
    while(cin >> n >> d, n && d)
    {
        cnt++;
        bool flag = false; // 有可能岛屿太远，无论如何也覆盖不到
        for(int i = 1; i <= n; i++)
        {
            double x, y; cin >> x >> y;
            if(y > d) flag = true;
            else
            {
                double t = sqrt(d * d - y * y);
                a[i].l = x - t, a[i].r = x + t;
            }
        }
        cout << "Case " << cnt << ": ";
        if(flag) cout << -1 << endl;
        else
        {
            sort(a + 1, a + 1 + n, cmp);
            int ret = 1;
            double r = a[1].r;
            for(int i = 2; i <= n; i++)
            {
                double x = a[i].l, y = a[i].r;
                if(x > r) // 没有重叠
                {
                    ret++;
```

```
50                      r = y;
51                  }
52                  else // 有重叠
53                  {
54                      r = min(r, y);
55                  }
56              }
57              cout << ret << endl;
58          }
59      }
60
61      return 0;
62  }
```

8.4.3 Sunscreen

题目来源： 洛谷

题目链接： P2887 [USACO07NOV] Sunscreen G

难度系数： ★★

【题目描述】

有 C 个奶牛去晒太阳 $(1 \leq C \leq 2500)$，每个奶牛各自能够忍受的阳光强度有一个最小值和一个最大值，太大就晒伤了，太小奶牛没感觉。

而刚开始的阳光的强度非常大，奶牛都承受不住，然后奶牛就得涂抹防晒霜，防晒霜的作用是让阳光照在身上的阳光强度固定为某个值。那么为了不让奶牛烫伤，又不会没有效果。给出了 L 种防晒霜。每种的数量和固定的阳光强度也给出来了

每个奶牛只能抹一瓶防晒霜，最后问能够享受晒太阳的奶牛有几个。

【输入描述】

第一行：两个整数： C, L；

接下来 C 行，每行两个整数，表示每个奶牛能忍受的阳光强度的最小值和最大值；

接下来 L 行，每行两个整数，表示每种防晒霜的固定阳光强度和数量

【输出描述】

输出一个整数，代表最多可以满足奶牛日光浴的奶牛数目。

【示例一】

输入：

```
3 2
```

3 10

2 5

1 5

6 2

4 1

输出：

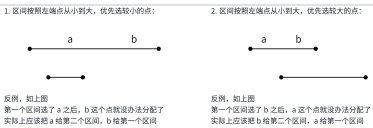
2

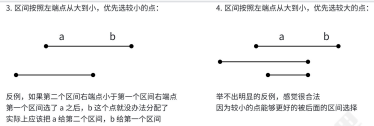
【解法】

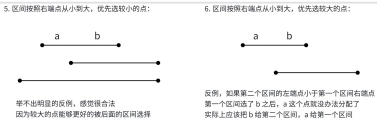
思考具体解法，从下面的情况中，筛选出没有特别明显的反例的组合：

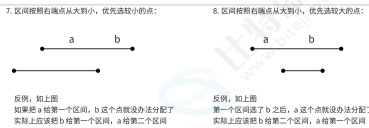
1. 区间按照左端点从小到大 + 防晒霜从小到大（优先选小）；
2. 区间按照左端点从小到大 + 防晒霜从大到小（优先选大）；
3. 区间按照左端点从大到小 + 防晒霜从小到大（优先选小）；
4. 区间按照左端点从大到小 + 防晒霜从大到小（优先选大）；
5. 区间按照右端点从小到大 + 防晒霜从小到大（优先选小）；
6. 区间按照右端点从小到大 + 防晒霜从大到小（优先选大）。
7. 区间按照右端点从大到小 + 防晒霜从小到大（优先选小）；
8. 区间按照右端点从大到小 + 防晒霜从大到小（优先选大）。

虽然看似很多，但是很容易在错误的策略中举出「反例」。









综上所述，有两种组合没有明显的反例，分别是：

1. 区间按照「左端点从大到小」排序，防晒霜从大到小排序，「优先选择较大」的防晒霜；
2. 区间按照「右端点从小到大」排序，防晒霜从小到大排序，「优先选择较小」的防晒霜。

实际上两种情况都是正确的，我们取其一证明一下，另一种证明方式类似。

可以用「交换论证法」证明方式二的正确性：

> 从前往后依次比较「贪心解」和「最优解」针对每一个区间的决策，当找到第一个区间，它们的「分配决策不同」时：设贪心解用的是 a 防晒霜，最优解用的是 b 防晒霜，因为贪心解选的是最小的，所以有 $a \leq b$。
>
> 此时关于 a 防晒霜的使用情况，可以分以下几种情况：
>
> 1. a 防晒霜在「最优解中没有使用」，那么我们可以直接用 a 防晒霜替换 b 防晒霜，此时最优解的「最优性」并没有损失，那么贪心解就和最优解决策一致；
> 2. a 防晒霜在「最优解的后续决策中使用了」，设 b 使用的区间是 $[x_b, y_b]$，a 使用的区间是 $[x_a, y_a]$：
>
>   易得：$x_b \leq b \leq y_b$，$x_a \leq a \leq y_a$
>
>   因为我们是按照右端点从小到大排序的，所以 $y_a \geq y_b$；
>
>   综上：$x_a \leq a \leq b \leq y_b \leq y_a$。
>
>   所以 b 也可以作用于「a 作用的区间」，也就是在最优解中「a, b 可以互换」，进而就转换成贪心解。
>
> 因此，针对每一个位置，我们都可以把最优解在「不失去其最优性的前提下」，转化成「贪心解」。

【参考代码】

代码块

```cpp
#include <iostream>
#include <algorithm>

using namespace std;

const int N = 2510;

int n, m;

struct node
{
    int x, y;
}a[N], b[N];

bool cmp1(node& x, node& y)
{
    return x.y < y.y;
}

bool cmp2(node& x, node& y)
```

```cpp
{
    return x.x < y.x;
}

bool cmp3(node& x, node& y)
{
    return x.x > y.x;
}

// 所有区间按照右端点从小到大排序
// 所有点按照从小到大排序
void solve1()
{
    sort(a + 1, a + 1 + n, cmp1);
    sort(b + 1, b + 1 + m, cmp2);

    int ret = 0;
    for(int i = 1; i <= n; i++)
    {
        int l = a[i].x, r = a[i].y;
        for(int j = 1; j <= m; j++)
        {
            // 选一个最小的，符合要求的点
            int &sp = b[j].x, &cnt = b[j].y;
            if(!cnt || sp < l) continue;
            if(sp > r) break;
            cnt--;
            ret++;
            break;
        }
    }
    cout << ret << endl;
}

// 所有区间按照左端点从大到小排列
// 所有点从大到小排列
void solve2()
{
    sort(a + 1, a + 1 + n, cmp3);
    sort(b + 1, b + 1 + m, cmp3);

    int ret = 0;
    for(int i = 1; i <= n; i++)
    {
        int l = a[i].x, r = a[i].y;
        for(int j = 1; j <= m; j++)
        {
```

```
            // 选一个最大的，符合要求的点
            int &sp = b[j].x, &cnt = b[j].y;
            if(!cnt || sp > r) continue;
            if(sp < l) break;
            ret++;
            cnt--;
            break;
        }
    }
    cout << ret << endl;
}

int main()
{
    cin >> n >> m;
    for(int i = 1; i <= n; i++) cin >> a[i].x >> a[i].y;
    for(int i = 1; i <= m; i++) cin >> b[i].x >> b[i].y;

    // solve1(); // 按照右端点排序
    solve2(); // 按照左端点排序
```

8.4.4 牛栏预定

题目来源： 洛谷

题目链接： P2859 [USACO06FEB] Stall Reservations S

难度系数： ★★

【题目描述】

约翰的 $N(1 \leq N \leq 50000)$ 头奶牛实在是太难伺候了，她们甚至有自己独特的产奶时段。对于某一头奶牛，她每天的产奶时段是固定的时间段 $[A, B]$（即 A 到 B，包括 A 和 B）。这使得约翰必须开发一个调控系统来决定每头奶牛应该被安排到哪个牛棚去挤奶，因为奶牛们并不希望在挤奶时被其它奶牛看见。

请帮约翰计算：如果要满足奶牛们的要求，并且每天每头奶牛都要被挤过奶，至少需要多少牛棚和每头牛应该在哪个牛棚被挤奶。如果有多种答案，输出任意一种均可。

【输入描述】

第一行为一个整数： N。

接下来 N，每行两个数字，分别代表第 i 头奶牛的产奶时段。

【输出描述】

第 1 行输出一个整数，代表需要牛棚的最少数量。

> 第 $2 \sim (N+1)$ 行，每行一个数字，第 $i+1$ 行的数字代表第 i 头奶牛将会被安排到哪个牛棚挤奶。

【示例一】

> 输入：
>
> 5
>
> 1 10
>
> 2 4
>
> 3 6
>
> 5 8
>
> 4 7
>
> 输出：
>
> 4
>
> 1
>
> 2
>
> 3
>
> 2
>
> 4

【解法】

按照「起始时间」对所有奶牛「从小到大」排序，然后「从前往后」依次安排每一头奶牛，设这头奶牛的产奶的时间区间是 $[a,b]$：

- 在已经有牛的所有牛棚里，如果「结束时间小于 a」，就可以把这头奶牛放在这个牛棚里面；如果有很多符合要求的，我们应该找一个「结束时间最大」的，这样可以将「结束时间较小」的留在后面，为后面的区间提供更多的机会。
- 如果所有已经有牛的牛棚的「结束时间都大于 a」，那么这头牛只能自己单独开一个牛棚。

这个贪心策略是比较符合我们「常识」的，尽量优的安排每一头牛。证明方法可以用「交换论证法」，前面已经叙述多次这种方式，大家可以自己尝试证明一下。我觉得重点应该是「如何实现我们这个想法」。

如何实现我们的算法：

1. 因为需要记录每一头奶牛分配的「牛棚号」，但是「排序」操作会让奶牛的「顺序」发生变化，不利于最后统计答案。因此我们可以定义一个叙述奶牛信息的「结构体」，里面存放如下信息：

- 产奶开始时间；
- 产奶结束时间；
- 排序之前的下标；
- 分配的牛棚；

这样我们对结构体排序之后，就能找到排序后的每一头奶牛所对应的「原始下标」。

2. 用什么存储每一个牛棚的「结束时间」？思考一下，我们要不断地在牛棚里面查找「小于 a 的最大值」：
   - 如果牛棚无序，我们需要遍历整个牛棚，时间复杂度吃不消；
   - 如果有序，我们可以用「二分」快速找到目标位置。

   但是用「数组」存，每次还要「排序」，效率还不如遍历，此时我们要用到 STL 提供的一个容器： set ，这个容器能够让「每次插入一个新的元素之后都是有序」的，并且「自带 $lower\_bound$ 和 $upper\_bound$ 」，整体的时间复杂度为 $O(logN)$ 。

   我们还需要标记每一个牛棚的「编号」，因此我们可以用 map 存储「双关键字」，一个标记结束时间，一个标记当前牛棚编号。但是有可能牛棚的结束时间一致，而 map 不能有重复元素，此时要用 $multimap$ 来存储每一个牛棚的信息。

【参考代码】

代码块

```
#include <iostream>
#include <map>
#include <algorithm>

using namespace std;

const int N = 5e4 + 10;

int n;
struct node
{
    // 存每一头牛的信息：开始时间，结束时间，排序之前的位置，分配的牛棚号
    int x, y, pos, num;
}a[N];
int cnt;
int ret[N];

multimap<int, int> st;

bool cmp(node& x, node& y)
```

```cpp
{
    return x.x < y.x;
}

int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++)
    {
        cin >> a[i].x >> a[i].y;
        a[i].pos = i;
    }
    sort(a + 1, a + 1 + n, cmp);

    for(int i = 1; i <= n; i++)
    {
        int l = a[i].x, r = a[i].y;
        auto it = st.upper_bound(l);

        // 不能接在任何一个区间后面，那就新开一个区间
        if(it == st.end())
        {
            st.insert({r, ++cnt});
            a[i].num = cnt;
        }
        else // 能接在某个区间后面，那就接上去
        {
            int p = it->second;
            st.erase(it); // 把之前牛删掉
            st.insert({r, p}); // 新来的牛放进去
            a[i].num = p;
        }
    }

    cout << cnt << endl;
    for(int i = 1; i <= n; i++)
    {
        ret[a[i].pos] = a[i].num;
    }

    for(int i = 1; i <= n; i++) cout << ret[i] << endl;

    return 0;
}
```

9. 倍增思想

9.1 快速幂

> **题目来源：** 洛谷
>
> **题目链接：** P1226 【模板】快速幂
>
> **难度系数：** ★

【题目描述】

> 给你三个整数 a, b, p ，求 $a^b \bmod p$

【输入描述】

> 三个用空格隔开的整数 a, b 和 p。

【输出描述】

> 输出一行一个字符串 `a^b mod p=s` ，其中 a, b, p 分别为题目给定的值， s 为运算结果。

【示例一】

> 输入：
>
> 2 10 9
>
> 输出：
>
> 2^10 mod 9=7

【解法】

利用「二进制」以及「倍增」的思想，通过一个具体的例子说明，比如 3^{11} ：

- 幂运算的运算法则： $a^{b+c} = a^b \times a^c$ ；
- 通过一个数的二进制表示，可以写成若干数相加：
  $11 = (1011)_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
- 两者结合： $3^{11} = 3^{(1011)_2} = 3^{1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0} = 3^8 \times 3^0 \times 3^2 \times 3^1$

因此，当我们知道 $3^1, 3^2, 3^4, 3^8, ..., 3^{log_2{}^n}$ 之后，只用做几次乘法就能计算出 3^{11} 。

如何快速算出 $3^1, 3^2, 3^3, ..., 3^{log_2{}^n}$ 。其实很简单，从前往后看，后一个数是前一个数的平方：

$$
\begin{aligned}
3^1 &= 3 \\
3^2 &= 3^1 \times 3^1 = 9 \\
3^4 &= 3^2 \times 3^2 = 81 \\
3^8 &= 3^4 \times 3^4 = 6561
\end{aligned}
$$

因此，计算 3^{11} ，我们只需将 11 的二进制表示中 1 所对应的幂乘起来即可。

如何实现这个算法呢，以 a^b 为例：

- 一边提取 b 的二进制表示中的每一位；
- 一边让 $a = a * a$，不断变成之前的平方（倍增的思想）；
- 在提取 b 的二进制表示时，如果这一位是 1，就乘上对应位置的权值。

【参考代码】

代码块

```cpp
#include <iostream>

using namespace std;

typedef long long LL;

// a^b % p 的值
LL quickpow(LL a, LL b, LL p)
{
    LL ret = 1;
    while(b)
    {
        if(b & 1) ret = ret * a % p;
        a = a * a % p;
        b >>= 1; // 提取 b 的二进制位
    }
    return ret;
}

int main()
{
    LL a, b, p;
    scanf("%lld%lld%lld", &a, &b, &p);
    printf("%lld^%lld mod %lld=%lld\n", a, b, p, quickpow(a, b, p));

    return 0;
}
```

9.2 大整数乘法

> **题目来源： 洛谷**

> 题目链接： P10446 64位整数乘法
>
> 难度系数： ★

【题目描述】

> 求 a 乘 b 对 p 取模的值，其中 $0 \leq a,b,p \leq 10^{18}$

【输入描述】

> 第一行 a ，第二行 b ，第三行 p 。

【输出描述】

> 一个整数，表示 $a \times b \bmod p$ 的值。

【示例一】

> 输入：
>
> 2
>
> 3
>
> 9
>
> 输出：
>
> 6

【解法】

跟「快速幂」的思想一致，我们通过一个具体的例子模拟一下算法的流程，比如 3×11 ：

- 乘法的分配率： $a \times (b + c) = a \times b + a \times c$ ；
- 通过一个数的二进制表示，可以写成若干数相加：
  $11 = (1011)_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
- 两者结合：
  $3 \times 11 = 3 \times (1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0) = 3 \times 8 + 3 \times 0 + 3 \times 2 + 3 \times 1$

因此，当我们知道 $3 \times 1, 3 \times 2, 3 \times 4, 3 \times 8, ..., 3 \times {log_2}^n$ 之后，只用做几次加法就能计算出 3×11 。

如何实现这个算法呢，以 $a \times b$ 为例：

- 一边提取 b 的二进制表示中的每一位；
- 一边让 $a = a + a$ ，不断变成之前的两倍（倍增的思想）；
- 在提取 b 的二进制表示时，如果这一位是 1 ，就加上对应位置的权值。

【参考代码】

代码块

```
#include <iostream>

using namespace std;

typedef long long LL;

// 加法比乘法快，而且防溢出
LL qmul(LL a, LL b, LL p)
{
    LL sum = 0;
    while(b) // 枚举 b 的二进制位
    {
        if(b & 1) sum = (sum + a) % p;
        a = (a + a) % p; // 计算下一个权值
        b >>= 1;
    }
    return sum % p;
}

int main()
{
    LL a, b, p;
    cin >> a >> b >> p;
    cout << qmul(a % p, b, p) << endl;

    return 0;
}
```

10. 离散化

【算法原理】

当题目中数据范围很大，但是数据的总量不是很大，并且我们需要用数据的值来映射数组的下标时，我们就可以用离散化的思想先预处理一下所有的数据，使的每一个数据都映射成一个范围较小的值。

比如：$[9, 99, 9999, 999999]$ 离散之后就变成 $[1, 2, 3, 4]$。

【离散化模板一】

代码块

```
// 离散化方式一：排序 + 去重 + 二分查找离散化后的值
#include <iostream>
```

```cpp
#include <algorithm>

using namespace std;

const int N = 1e5 + 10;

int n;
int a[N]; // 原始数据

int pos; // 记录离散化数组中元素的个数
int disc[N]; // 离散化需要的数组

// 二分查找离散化之后的值，其实就是排序之后的下标
int find(int x)
{
    int l = 1, r = pos; // 注意查找的区间
    while(l < r)
    {
        int mid = (l + r) >> 1;
        if(disc[mid] >= x) r = mid;
        else l = mid + 1;
    }
    return l;
}

int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++)
    {
        int x; cin >> x;
        a[i] = x;
        disc[++pos] = x; // 数据放进离散化数组中
    }

    // 离散化：排序 + 去重
    sort(disc + 1, disc + 1 + pos);
    pos = unique(disc + 1, disc + 1 + pos) - (disc + 1);

    // 找到离散化之后的值
    for(int i = 1; i <= n; i++)
    {
        int x = a[i];
        cout << x << "离散化之后是：  " << find(x) << endl; // 二分查找离散化之后的值
    }

    return 0;
```

```
50  }
```

【离散化模版二】

代码块

```
1   // 离散化方式二：排序 + STL
2   // 本质是和方式一一样的，只不过借助了 STL，去重以及查找更方便
3   #include <iostream>
4   #include <unordered_map>
5   #include <algorithm>
6
7   using namespace std;
8
9   const int N = 1e5 + 10;
10
11  int n;
12  int a[N]; // 原始数据
13
14  int tmp[N]; // 用来排序的数组
15  int cnt;
16  unordered_map<int, int> id; // 记录离散化之后的值
17
18  int main()
19  {
20      cin >> n;
21      for(int i = 1; i <= n; i++)
22      {
23          int x; cin >> x;
24          a[i] = x;
25          tmp[i] = x; // 数据放进离散化数组中
26      }
27
28      // 离散化：排序 + 放进哈希表中
29      sort(tmp+ 1, tmp + 1 + n);
30      for(int i = 1; i <= n; i++)
31      {
32          if(id.count(tmp[i])) continue; // 如果已经存过这个数，不做处理
33          cnt++; // 这个数映射之后的值
34          id[tmp[i]] = cnt; // 放进哈希表中
35      }
36
37      // 找到离散化之后的值
38      for(int i = 1; i <= n; i++)
39      {
```

```
40          int x = a[i];
41          cout << x << "离散化之后是:  " << id[a[i]] << endl; // 二分查找离散化之后的
    值
42      }
43
44      return 0;
45  }
```

【注意事项】

1. 离散化是一种「思想」，模版其实不用背，根据算法思想就可以实现。而且实现离散化的方式也可以在上述模板的基础上「修改」，千万「不要生搬硬套」（大家也会看到有些题解里面是借助「结构体」离散化的，但是核心的思想都是不变的）；
2. 前期学习离散化的时候可能会被「绕」进去，会把「离散前」和「离散后」的值搞混，分不清楚是用离散前的值还是离散后的值。觉得迷惑是「很正常」，一定要根据最基础的模板画图分析整个流程，搞清楚每一个变量的作用，以及达到的目的。离散化「使用的多」了，慢慢的就不会迷了。

10.1 火烧赤壁

> **题目来源：** 洛谷
>
> **题目链接：** P1496 火烧赤壁
>
> **难度系数：** ★★

【题目描述】

> 曹操平定北方以后，公元 208 年，率领大军南下，进攻刘表。他的人马还没有到荆州，刘表已经病死。他的儿子刘琮听到曹军声势浩大，吓破了胆，先派人求降了。孙权任命周瑜为都督，拨给他三万水军，叫他同刘备协力抵抗曹操。
>
> 隆冬的十一月，天气突然回暖，刮起了东南风。没想到东吴船队离开北岸大约二里距离，前面十条大船突然同时起火。火借风势，风助火威。十条火船，好比十条火龙一样，闯进曹军水寨。那里的船舰，都挤在一起，又躲不开，很快地都烧起来。一眨眼工夫，已经烧成一片火海。
>
> 曹操气急败坏的把你找来，要你钻入火海把连环线上着火的船只的长度统计出来！
>
> 给定每个起火部分的起点和终点，请你求出燃烧位置的长度之和。

【输入描述】

> 第一行一个整数，表示起火的信息条数 n 。
> 接下来 n 行，每行两个整数 a,b ，表示一个着火位置的起点和终点（注意：左闭右开）。

> 对于全部的测试点，保证 $1 \leq n \leq 2 \times 10^4$, $-2^{31} \leq a < b < 2^{31}$ ，且答案小于 2^{31} 。

【输出描述】

> 输出一行一个整数表示答案

【示例一】

> 输入：
>
> 3
>
> -1 1
>
> 5 11
>
> 2 9
>
> 输出：
>
> 11

【解法】

抛开数据范围不看，这就是一道「差分」题目：

- 给定一个区间，我们可以全部执行 $+1$ 操作；
- 最后看看整个数组中，大于 0 的位置有多少个。

因此可以创建一个原数组的「差分」数组，然后执行完「区间修改」操作之后，还原原数组，「统计大于 0 」的区间长度。

但是，这道题的「数据范围」不允许我们直接差分，因为「开不了那么大」的数组；即使能开那么大的数组，「时间」也不够用。

我们发现，区间的范围虽然很大，区间的「个数」却只有 2×10^4 级别。此时我们就可以：

1. 先将所有的「区间信息」离散化；
2. 然后在「离散化的基础」上，处理所有的「区间修改」操作；
3. 处理完之后找出「原始数组对应的区间端点」，计算相应的「长度」。

【参考代码】

代码块

```
#include <iostream>
#include <algorithm>

using namespace std;

```

```cpp
const int N = 2e4 + 10;

int n;
int l[N], r[N];

int m; // 离散数组的大小
int disc[N * 2]; // 离散之后的数组
int f[N * 2]; // 差分数组

int find(int x)
{
    int l = 1, r = m;
    while(l < r)
    {
        int mid = (l + r + 1) >> 1;
        if(disc[mid] <= x) l = mid;
        else r = mid - 1;
    }
    return l;
}

int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++)
    {
        int x, y; cin >> x >> y;
        l[i] = x, r[i] = y;
        disc[++m] = l[i], disc[++m] = r[i];
    }

    // 离散化处理
    sort(disc + 1, disc + 1 + m);
    m = unique(disc + 1, disc + 1 + m) - disc - 1;

    // 处理区间修改
    for(int i = 1; i <= n; i++)
    {
        int x = find(l[i]), y = find(r[i]);
        f[x] += 1, f[y] -= 1;
    }

    // 还原数组
    for(int i = 1; i <= m; i++) f[i] += f[i - 1];

    // 找出每一段大于0的区间，统计长度
    int sum = 0;
```

```
    for(int i = 1; i <= m; i++)
    {
        if(f[i] <= 0) continue;
        int j = i;
        while(j <= m && f[j] > 0) j++;
        // 累加长度的时候记得使用离散化之前的值
        sum += disc[j] - disc[i];
        i = j;
    }
    cout << sum << endl;

    return 0;
}
```

10.2 贴海报

题目来源： 洛谷

题目链接： P3740 [HAOI2014] 贴海报

难度系数： ★★

【题目描述】

Bytetown 城市要进行市长竞选，所有的选民可以畅所欲言地对竞选市长的候选人发表言论。为了统一管理，城市委员会为选民准备了一个张贴海报的 electoral 墙。

张贴规则如下：

1. electoral 墙是一个长度为 N 个单位的长方形，每个单位记为一个格子；
2. 所有张贴的海报的高度必须与 electoral 墙的高度一致的；
3. 每张海报以 `A B` 表示，即从第 A 个格子到第 B 个格子张贴海报；
4. 后贴的海报可以覆盖前面已贴的海报或部分海报。

现在请你判断，张贴完所有海报后，在 electoral 墙上还可以看见多少张海报。

【输入描述】

第一行，两个正整数 N, M ，分别表示 electoral 墙的长度和海报个数。

接下来 M 行，每行两个正整数 A_i, B_i ，表示每张海报张贴的位置。

数据范围：$10 \leq N \leq 10000000, 1 \leq M \leq 1000, 1 \leq A_i, B_i \leq 10000000$

【输出描述】

输出贴完所有海报后，在 electoral 墙上还可以看见的海报数。

【示例一】

输入：

100 5

1 4

2 6

8 10

3 4

7 10

输出：

4

【解法】

根据题意「模拟」即可。

由于「区间的长度」很大，暴力模拟的时候会超时。但是我们发现，虽然区间长度很大，但是「区间的个数」是很少的，所以我们可以「离散化」处理一下区间的端点值，然后在「离散化的基础上」模拟覆盖情况。

注意注意注意，重要的注意说三遍：

离散化在离散「区间问题」的时候一定要小心！因为我们离散化操作会把区间缩短，从而导致丢失一些点。在涉及「区间覆盖」问题上，离散化会导致「结果出错」。

比如我们这道题，如果有三个区间分别为：$[2,5],[2,3],[5,6]$，离散化之后为：$[1,3],[1,2],[3,4]$，区间覆盖如图所示：

为了避免出现上述情况，我们可以在离散化的区间 $[x,y]$ 时，不仅考虑 x,y 这两个值，也把「$x+1,y+1$」也考虑进去。此时「单个区间内部」就出现空隙，「区间与区间之间」也会出现空隙。就可以避免上述情况出现。

可见，离散化之后可能会导致结果错误，使用的时候还是需要「谨慎」一点。

【参考代码】

代码块

```
#include <iostream>
#include <algorithm>

using namespace std;

const int N = 1010;

int n, m;
int a[N], b[N];

int pos;
int disc[N * 4]; // 因为有两套位置

int w[N * 4];
bool mp[N];

// 找到 x 映射之后的数，也就是 x 的下标
int find(int x)
{
    int l = 1, r = pos;
    while(l < r)
    {
        int mid = (l + r) >> 1;
        if(disc[mid] >= x) r = mid;
        else l = mid + 1;
    }
    return l;
}

int main()
{
    cin >> n >> m;
    for(int i = 1; i <= m; i++)
    {
        int x, y; cin >> x >> y;
```

```
36          a[i] = x, b[i] = y;
37          // 离散化之后有可能导致区间缩小，多加一个位置
38          disc[++pos] = x, disc[++pos] = x + 1;
39          disc[++pos] = y, disc[++pos] = y + 1;
40      }
41
42      // 离散化
43      sort(disc + 1, disc + 1 + pos);
44      pos = unique(disc + 1, disc + 1 + pos) - (disc + 1);
45
46      // 用离散化之后的值覆盖区间
47      for(int i = 1; i <= m; i++)
48      {
49          int x = find(a[i]), y = find(b[i]);
50          for(int j = x; j <= y; j++) w[j] = i;
51      }
52
53      // 统计整个数组中，一共有多少个不同的数
54      int cnt = 0;
55      for(int i = 1; i <= pos; i++)
56      {
57          int x = w[i];
58          if(!x) continue; // 不要统计 0
59          if(mp[x]) continue;
60          cnt++;
61          mp[x] = true;
62      }
63
64      cout << cnt << endl;
65
66      return 0;
67  }
```

11. 递归初阶

递归：函数自己调用自己。

在本小结里，我们会通过一些简单题目了解递归的本质。先带着大家消除对递归的恐惧，为我们后续学习更高级的算法做铺垫。

11.1 汉诺塔

> **题目来源：** 信息学奥赛一本通
>
> **题目链接：** 汉诺塔问题

> **难度系数：** ★

【题目描述】

> 约 19 世纪末，在欧州的商店中出售一种智力玩具，在一块铜板上有三根杆，最左边的杆上自上而下、由小到大顺序串着由 64 个圆盘构成的塔。目的是将最左边杆上的盘全部移到中间的杆上，条件是一次只能移动一个盘，且不允许大盘放在小盘的上面。
>
> 这是一个著名的问题，几乎所有的教材上都有这个问题。由于条件是一次只能移动一个盘，且不允许大盘放在小盘上面，所以 64 个 盘的移动次数是： 18446744073709551615
>
> 这是一个天文数字，若每一微秒可能计算(并不输出)一次移动，那么也需要几乎一百万年。我们仅能找出问题的解决方法并解决较小 N 值时的汉诺塔，但很难用计算机解决 64 层的汉诺塔。
>
> 假定圆盘从小到大编号为 $1, 2, ...$

【输入描述】

> 输入为一个整数(小于 20）后面跟三个单字符字符串。
>
> 整数为盘子的数目，后三个字符表示三个杆子的编号。

【输出描述】

> 输出每一步移动盘子的记录。一次移动一行。
>
> 每次移动的记录为例如 `a->3->b` 的形式，即把编号为 3 的盘子从 a 杆移至 b 杆。

【示例一】

> 输入：
>
> ```
> 2 a b c
> ```
>
> 输出：
>
> ```
> a->1->c
> a->2->b
> c->1->b
> ```

【解法】

这是一道「递归」算法的经典题目，我们可以先从「最简单」的情况考虑：

- 假设 $n = 1$，只有一个盘子，很简单，直接把它从 a 中拿出来，移到 c 上；
- 如果 $n = 2$ 呢？这时候我们就要借助 b 了，因为小盘子必须时刻都在大盘子上面，共需要 3 步（为了方便叙述，记 a 中的盘子从上到下为 $1, 2$）：
  a. 1 号盘子放到 b 上；
  b. 2 号盘子放到 c 上；
  c. 1 号盘子放到 c 上。

  至此， c 中的盘子从上到下为 1 号， 2 号。

- 如果 $n=3$ 呢？这是我们需要用到 $n=2$ 时的策略，将 a 上面的 2 个盘子挪到 b 上，再将最大的盘子挪到 c 上，最后将 b 上的 2 个盘子挪到 c 上。其中转移 2 个盘子的策略正好是 $n=2$ 的方式。

  在处理 $n-1$ 的时候，问题又转化成了相同的子问题，此时就可以用递归帮助我们解决。
- 由此我们可以类比出 n 为任意值的处理方式：
  - 当 $n>1$ 时：将 a 上面的 $n-1$ 个盘子挪到 b 上，再将最大的盘子挪到 c 上，最后将 b 上的 $n-1$ 个盘子挪到 c 上就完成了所有步骤；

    在处理 $n-1$ 的时候，问题又转化成了相同的子问题，此时就可以用递归帮助我们解决。
  - 当 $n=1$ 时：直接放到目标柱子上。

设计递归函数（从重复子问题入手）：

- 重复子问题：将 x 个盘子，从 $begin$ 柱子转移到 end 柱子上，其中通过 tmp 柱子作为中转；
- 具体操作：
  a. 先将 $beign$ 上面 $x-1$ 个盘子借助 end 转移到 tmp 上；
  b. 再把 $begin$ 最下面一个盘子放在 end 上；
  c. 最后把 tmp 上面 $x-1$ 个盘子借助 $begin$ 转移到 end 上。

【参考代码】

代码块

```cpp
#include <iostream>

using namespace std;

int n;
char a, b, c;

// 将 x 个盘子从 begin 移动到 end
// 其中 tmp 是中转站
void move(int x, char begin, char tmp, char end)
{
    if(x == 0) return;

    move(x - 1, begin, end, tmp);
    printf("%c->%d->%c\n", begin, x, end);
    move(x - 1, tmp, begin, end);
}
```

```
18
19  int main()
20  {
21      cin >> n >> a >> b >> c;
22      move(n, a, c, b); // 把 a 上面 n 个盘子，借助 c 转移到 b 上
23
24      return 0;
25  }
```

11.2 占卜DIY

题目来源： 洛谷

题目链接： P10457 占卜DIY

难度系数： ★

【题目描述】

lyd 学会了使用扑克 DIY 占卜。方法如下：一副去掉大小王的扑克共 52 张，打乱后均分为 13 堆，编号 $1\sim13$，每堆 4 张，其中第 13 堆称作“生命牌”，也就是说你有 4 条命。这里边，4 张 k 被称作死神。

初始状态下，所有的牌背面朝上扣下。

流程如下：

- 抽取生命牌中的最上面一张（第一张）。
- 把这张牌翻开，正面朝上，放到牌上的数字所对应编号的堆的最上边。(例如抽到 2，正面朝上放到第 2 堆牌最上面，又比如抽到 J，放到第 11 堆牌最上边，注意是正面朝上放)
- 从刚放了牌的那一堆最底下（最后一张）抽取一张牌，重复第 2 步。（例如你上次抽了 2，放到了第二堆顶部，现在抽第二堆最后一张发现是 8，又放到第 8 堆顶部……）
- 在抽牌过程中如果抽到 K，则称死了一条命，就扔掉 K 再从第 1 步开始。
- 当发现四条命都死了以后，统计现在每堆牌上边正面朝上的牌的数目，只要同一数字的牌出现 4 张正面朝上的牌（比如 4 个 A），则称“开了一对”，当然 4 个 K 是不算的。
- 统计一共开了多少对，开了 0 对称作"极凶"，$1\sim2$ 对为“大凶”，3 对为“凶”，$4\sim5$ 对为“小凶”，6 对为“中庸”，$7\sim8$ 对“小吉”，9 对为“吉”，$10\sim11$ 为“大吉”，12 为“满堂开花，极吉”。

【输入描述】

一共 13 行，为每堆牌的具体牌是什么(不区分花色只区分数字)，每堆输入的顺序为从上到下。

为了便于读入，用 0 代表 10。

【输出描述】

> 输出一共开了多少对。

【示例一】

> 输入：
>
> 85AA
>
> K532
>
> 9606
>
> 3434
>
> 3445
>
> 5676
>
> 8777
>
> 9988
>
> 9000
>
> KJJJ
>
> QAQK
>
> JQ22
>
> AKQ2
>
> 输出：
>
> 9

【解法】

搞清楚题意之后，其实就是一个简单的模拟题。这道题用循环也是可以模拟出来的，但是可以锻炼一下递归解法。

整个模拟过程：

1. 抽出第 13 堆最上面的牌 x ：
2. 把第 x 堆的最后一张拿出来；
3. 拿到这张牌之后，重复 2 过程，直到拿到 13

上述三步整个一循环，一直循环 4 次，直到把 13 堆上面的牌拿完。

重复子问题：

- 拿到一张牌 x 之后，把第 x 堆最后一张拿出来。

【参考代码】

代码块

```cpp
#include <iostream>

using namespace std;

const int N = 15, M = 6;

int n = 13, m = 4;
int a[N][M];
int cnt[N]; // 记录一下每堆牌里面剩余拍张个数，方便拿到最后一张牌

void dfs(int x)
{
    // 遇到 13 就停止
    if(x == 13) return;

    // 拿到第 x 堆里面最后一张牌
    int t = a[x][cnt[x]];
    cnt[x]--; // 剩余牌数 -1
    // 继续处理拿到的牌
    dfs(t);
}

int main()
{
    // 读入数据，将对应的字符转化成数字
    for(int i = 1; i <= n; i++)
    {
        cnt[i] = 4;
        for(int j = 1; j <= m; j++)
        {
            char ch; cin >> ch;
            if(ch >= '2' && ch <= '9') a[i][j] = ch - '0';
            else if(ch == 'A') a[i][j] = 1;
            else if(ch == '0') a[i][j] = 10;
            else if(ch == 'J') a[i][j] = 11;
            else if(ch == 'Q') a[i][j] = 12;
            else a[i][j] = 13;
        }
    }

    // 从前往后依次处理每张牌
    for(int j = 1; j <= m; j++)
    {
```

```
44          dfs(a[n][j]);
45      }
46
47      // 统计结果
48      int ret = 0;
49      for(int i = 1; i <= n; i++)
50      {
51          if(cnt[i] == 0) ret++;
52      }
53      cout << ret << endl;
54
55      return 0;
56  }
```

11.3 FBI 树

题目来源： 洛谷

题目链接： P1087 [NOIP2004 普及组] FBI 树

难度系数： ★★

【题目描述】

我们可以把由 “0” 和 “1” 组成的字符串分为三类：全 “0” 串称为 B 串，全 “1” 串称为 I 串，既含 “0” 又含 “1” 的串则称为 F 串。

FBI 树是一种二叉树，它的结点类型也包括 F 结点，B 结点和 I 结点三种。由一个长度为 2^N 的 “01” 串 S 可以构造出一棵 FBI 树 T，递归的构造方法如下：

1. T 的根结点为 R，其类型与串 S 的类型相同；
2. 若串 S 的长度大于 1 ，将串 S 从中间分开，分为等长的左右子串 S_1 和 S_2 ；由左子串 S_1 构造 R 的左子树 T_1 ，由右子串 S_2 构造 R 的右子树 T_2 。

现在给定一个长度为 2^N 的 “01” 串，请用上述构造方法构造出一棵 FBI 树，并输出它的后序遍历序列。

【输入描述】

第一行是一个整数 N $0 \leq N \leq 10$

第二行是一个长度为 2^N 的 “01” 串。

【输出描述】

一个字符串，即 FBI 树的后序遍历序列。

【示例一】

> 输入：
>
> 3
>
> 10001011
>
> 输出：
>
> IBFBBBFIBFIIIFF

【解法】

重复子问题：处理每一棵子树：

1. 确定出该子树的类型；
2. 然后从中间分开，先处理左子树，再处理右子树；
3. 然后打印该子树的类型

如何快速判断出该子树的类型？因为我们要求的是一段区间内 1 的个数，我们可以利用「前缀和」数组求出这段区间和，然后在查询某段区间时，判断一下此时的区间和：

- 如果等于区间长度，说明是 I 类型；
- 如果等于 0 ，说明是 B 类型；
- 否则就是 F 类型。

【参考代码】

代码块

```
#include <iostream>
#include <string>

using namespace std;

int n;
int sum[1 << 11];
string s;

void dfs(int l, int r)
{
    if(l > r) return;

    // 判断区间类型
    char ret;
    int s = sum[r] - sum[l - 1];
    if(s == r - l + 1) ret = 'I';
```

```
18        else if(s == 0) ret = 'B';
19        else ret = 'F';
20
21        // 如果只有一个节点，不需要递归了，直接打印
22        if(l == r)
23        {
24            cout << ret;
25            return;
26        }
27
28        // 先处理左右，再打印自己
29        int mid = (r + l) / 2;
30        dfs(l, mid);
31        dfs(mid + 1, r);
32        cout << ret;
33    }
34
35    int main()
36    {
37        cin >> n >> s;
38        n = (1 << n);
39        s = " " + s;
40        for(int i = 1; i <= n; i++)
41        {
42            if(s[i] == '1') sum[i] = sum[i - 1] + 1;
43            else sum[i] = sum[i - 1];
44        }
45        dfs(1, n);
46
47        return 0;
48    }
```

12. 分治

分治，字面上的解释是「分而治之」，就是把一个复杂的问题分成两个或更多的相同或相似的子问题，直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。

12.1 逆序对

> **题目来源：** 洛谷
>
> **题目链接：** P1908 逆序对
>
> **难度系数：** ★★★

【题目描述】

> 猫猫 TOM 和小老鼠 JERRY 最近又较量上了，但是毕竟都是成年人，他们已经不喜欢再玩那种你追我赶的游戏，现在他们喜欢玩统计。
>
> 最近，TOM 老猫查阅到一个人类称之为“逆序对”的东西，这东西是这样定义的：对于给定的一段正整数序列，逆序对就是序列中 $a_i > a_j$ 且 $i < j$ 的有序对。知道这概念后，他们就比赛谁先算出给定的一段正整数序列中逆序对的数目。注意序列中可能有重复数字。

【输入描述】

> 第一行，一个数 n ，表示序列中有 n 个数。
>
> 第二行 n 个数，表示给定的序列。序列中每个数字不超过 10^9 。
>
> 对于所有数据， $n \leq 5 \times 10^5$

【输出描述】

> 输出序列中逆序对的数目。

【示例一】

> 输入：
>
> ```
> 6
> 5 4 2 6 3 1
> ```
>
> 输出：
>
> ```
> 11
> ```

【解法】

「分治」是解决「逆序对」非常经典的解法，后续我们也会学到利用「树状数组」或「线段树」解决逆序对问题。

如果把整个序列 $[l, r]$ 从中间 mid 位置分成两部分，那么逆序对个数可以分成三部分：

- $[l, mid]$ 区间内逆序对的个数 $c1$ ；
- $[mid + 1, r]$ 区间内逆序对的个数 $c2$ ；
- 从 $[l, mid]$ 以及 $[mid + 1, r]$ 各选一个数，能组成的逆序对的个数 $c3$

那么逆序对的总数就是 $c1 + c2 + c3$ 。其中求解 $c1, c2$ 的时候跟原问题是一样的，可以交给「递归」去处理（经过前面题目的铺垫，我相信大家应该能够理解），那我们重点处理「一左一右」的情况。

如果在处理一左一右的情况时，两个数组「无序」，我们的时间复杂度其实并没有优化到哪去，甚至还「不如暴力解法」。但是如果两个数组有序的话，我们就可以快速找出逆序对的个数。

先不管怎么求逆序对，我们能让左右两个数组有序嘛？当然可以，这不就是「归并排序」么。因此，我们能做到在求完 $c1, c2$ 之后，然后让「左右两个区间有序」。

那么接下来问题就变成，已知两个「有序数组」，如何求出左边选一个数，右边选一个数的情况下的逆序对的个数。核心思想就是找到右边选一个数之后，左边区间内「有多少个比我大的」。

定义两个「指针」扫描两个有序数组：此时会有下面三种情况：

1. $a[cur1] \le a[cur2]$： $a[cur1]$ 不会与 $[cur2, r]$ 区间内任何一个元素构成逆序对， $cur1++$；
2. $a[cur1] > a[cur2]$：此时 $[cur1, mid]$ 区间内所有元素都会与 $a[cur2]$ 构成逆序对，逆序对个数增加 $mid - cur1 + 1$，此时 $cur2$ 已经统计过逆序对了， $cur2++$；

重复上面两步，我们就可以在 $O(N)$ 的时间内处理完「一左一右」时，逆序对的个数。而且，我们会发现，这跟我们「归并排序的过程」是高度一致的。所以可以一边排序，一边计算逆序对的个数。

【参考代码】

代码块

```
#include <iostream>

using namespace std;
typedef long long LL;

const int N = 5e5 + 10;

int n;
int arr[N], tmp[N];

LL merge(int l, int r)
{
    if(l >= r) return 0;

    int mid = (l + r) / 2;
    // [l, mid] [mid + 1, r]
    LL ret = 0;
```

```cpp
    // 先去算算左右两部分逆序对的和，顺便给左右两边排排序
    ret += merge(l, mid); ret += merge(mid + 1, r);

    // 计算一左一右时，逆序对的个数
    int cur1 = l, cur2 = mid + 1, i = l;
    while(cur1 <= mid && cur2 <= r)
    {
        if(arr[cur1] <= arr[cur2])
        {
            tmp[i++] = arr[cur1++];
        }
        else // 右边元素比左边小时，产生逆序对
        {
            // 统计逆序对的个数
            ret += (mid - cur1 + 1);
            tmp[i++] = arr[cur2++];
        }
    }
    while(cur1 <= mid) tmp[i++] = arr[cur1++];
    while(cur2 <= r) tmp[i++] = arr[cur2++];

    for(int j = l; j <= r; j++) arr[j] = tmp[j];
    return ret;
}

int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> arr[i];

    cout << merge(1, n) << endl;

    // for(int i = 1; i <= n; i++) cout << arr[i] << " ";

    return 0;
}
```

12.2 求第 k 小的数

> **题目来源：** 洛谷
>
> **题目链接：** 求第 k 小的数
>
> **难度系数：** ★★★

【题目描述】

> 输入 n （$1 \leq n < 5000000$ 且 n 为奇数）个数字 a_i $(1 \leq a_i < 10^9)$，输出这些数字的第 k 小的数。最小的数是第 0 小。
>
> 请尽量不要使用 `nth_element` 来写本题，因为本题的重点在于练习分治算法。

【示例一】

> 输入：
>
> ```
> 5 1
> 4 3 2 1 5
> ```
>
> 输出：
>
> ```
> 2
> ```

【解法】

【参考代码】

代码块

```cpp
#include <iostream>
#include <ctime>

using namespace std;

const int N = 5e6 + 10;

int n, k;
int a[N];

// [l, r] 之间随机返回一个数
int get_random(int l, int r)
{
    return a[rand() % (r - l + 1) + l];
}

int quick_select(int l, int r, int k)
{
    if(l == r) return a[l];

    // 随机选择基准元素
    int p = get_random(l, r);
    // 以 p 为基准划分三部分
    int x = l - 1, y = r + 1, i = l;
    while(i < y)
    {
        if(a[i] == p) i++;
```

```
28          else if(a[i] < p) swap(a[++x], a[i++]);
29          else swap(a[--y], a[i]);
30      }
31
32      // [l, x] [x + 1, y - 1] [y, r]
33      int c1 = x - l + 1, c2 = y - 1 - x, c3 = r - y + 1;
34      if(k <= c1) return quick_select(l, x, k);
35      else if(k <= c1 + c2) return p;
36      else return quick_select(y, r, k - c1 - c2);
37  }
38
39  int main()
40  {
41      srand(time(0));
42
43      scanf("%d%d", &n, &k);
44      k++;
45      for(int i = 1; i <= n; i++)
46      {
47          scanf("%d", &a[i]);
48      }
49      printf("%d\n", quick_select(1, n, k));
50
51      return 0;
52  }
```

12.3 最大子段和

> **题目来源：** 洛谷
>
> **题目链接：** P1115 最大子段和
>
> **难度系数：** ★★

【题目描述】

> 给出一个长度为 n 的序列 a ，选出其中连续且非空的一段使得这段和最大。

【输入描述】

> 第一行是一个整数，表示序列的长度 n。
>
> 第二行有 n 个整数，第 i 个整数表示序列的第 i 个数字 a_i。
>
> - 对于 40% 的数据，保证 $n \leq 1 \times 10^3$ 。
> - 对于 100% 的数据，保证 $1 \leq n \leq 2 \times 10^5, -10^4 \leq a_i \leq 10^4$ 。

【输出描述】

> 输出一行一个整数表示答案。

【示例一】

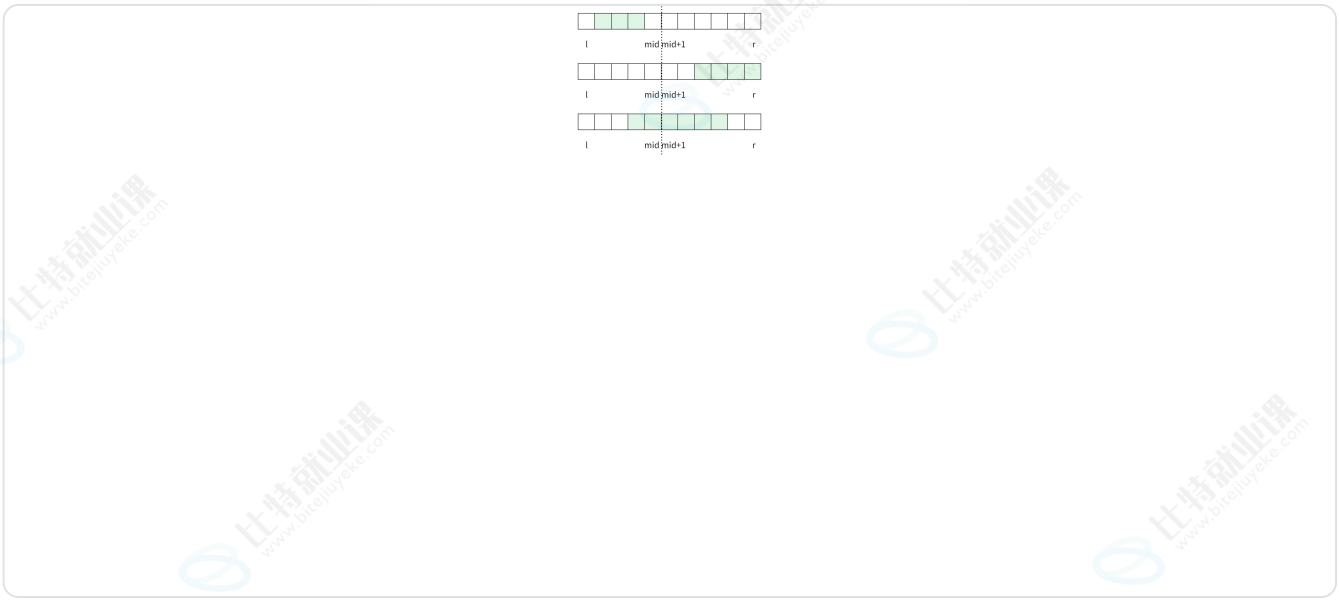
> 输入：
>
> 7
>
> 2 -4 3 -1 2 -4 3
>
> 输出：
>
> 4

【解法】

第三次遇见它了~

如果把整个序列 $[l,r]$ 从中间 mid 位置分成两部分，那么整个序列中「所有的子数组」就分成三部分：

- 子数组在区间 $[l,mid]$ 内；
- 子数组在区间 $[mid+1,r]$ 内；
- 子数组的左端点在 $[l,mid]$ 内，右端点在 $[mid+1,r]$ 内。



那么我们的「最终结果」也会在这三部分取到，要么在左边区间，要么在右边区间，要么在跨越中轴线的区间。因此，我们可以先求出左边区间的最大子段和，再求出右边区间的最大子段和，最后求出中间区间的最大子段和。其中求「左右区间」时，可以交给「递归」去解决。

那我们重点处理如何求出「中间区间」的最大子段和。可以把中间区间分成两部分：

- 左边部分是从 mid 为起点，「向左延伸」的最大子段和；

- 右边部分是从 $mid+1$ 为起点，「向右延伸」的最大子段和。

分别求出这两个值，然后相加即可。

求法也很简单，直接「固定起点」，一直把「以它为起点的所有子数组」的和都计算出来，取最大值即可。

【参考代码】

代码块

```
#include <iostream>

using namespace std;

const int N = 2e5 + 10;

int n;
int a[N];

int dfs(int l, int r)
{
    if(l == r) return a[l];

    int mid = (l + r) >> 1;
    // 先求一下左右两边的最大值
    int ret = max(dfs(l, mid), dfs(mid + 1, r));

    int suml = a[mid], sumr = a[mid + 1];
    int maxl = suml, maxr = sumr;
    // 求出以 a[mid] 开始，向左延伸的最大值
    for(int i = mid - 1; i >= l; i--)
    {
        suml += a[i];
        maxl = max(maxl, suml);
    }
    // 求出以 a[mid + 1] 开始，向右延伸的最大值
    for(int i = mid + 2; i <= r; i++)
    {
        sumr += a[i];
        maxr = max(maxr, sumr);
    }

    // 返回三种情况的最大值
    return max(maxr + maxl, ret);
}

```

```cpp
int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> a[i];
    cout << dfs(1, n) << endl;

    return 0;
}
```

12.4 地毯填补问题

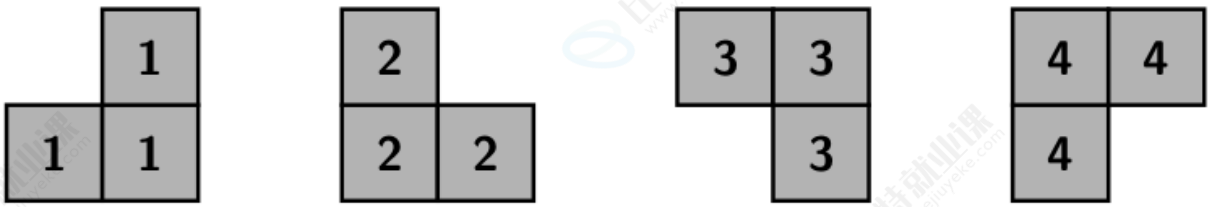
题目来源： 洛谷

题目链接： P1228 地毯填补问题

难度系数： ★★★

【题目描述】

相传在一个古老的阿拉伯国家里，有一座宫殿。宫殿里有个四四方方的格子迷宫，国王选择驸马的方法非常特殊，也非常简单：公主就站在其中一个方格子上，只要谁能用地毯将除公主站立的地方外的所有地方盖上，美丽漂亮聪慧的公主就是他的人了。公主这一个方格不能用地毯盖住，毯子的形状有所规定，只能有四种选择（如图）：



洛谷

并且每一方格只能用一层地毯，迷宫的大小为 $2^k \times 2^k$ 的方形。当然，也不能让公主无限制的在那儿等，对吧？由于你使用的是计算机，所以实现时间为 1 秒。

【输入描述】

输入文件共 2 行。

第一行一个整数 k，即给定被填补迷宫的大小为 $2^k \times 2^k (0 < k \leq 10)$；第二行两个整数 x, y，即给出公主所在方格的坐标（x 为行坐标，y 为列坐标），x 和 y 之间有一个空格隔开。

【输出描述】

将迷宫填补完整的方案：每一行为 x, y, c（x, y 为毯子拐角的行坐标和列坐标，c 为使用毯子的形状，具体见上面的图，毯子形状分别用 $1, 2, 3, 4$ 表示，x, y, c 之间用一个空格隔开）。

【示例一】

输入：

3

3 3

输出：

5 5 1

2 2 4

1 1 4

1 4 3

4 1 2

4 4 1

2 7 3

1 5 4

1 8 3

3 6 3

4 8 1

7 2 2

5 1 4

6 3 2

8 1 2

8 4 1

7 7 1

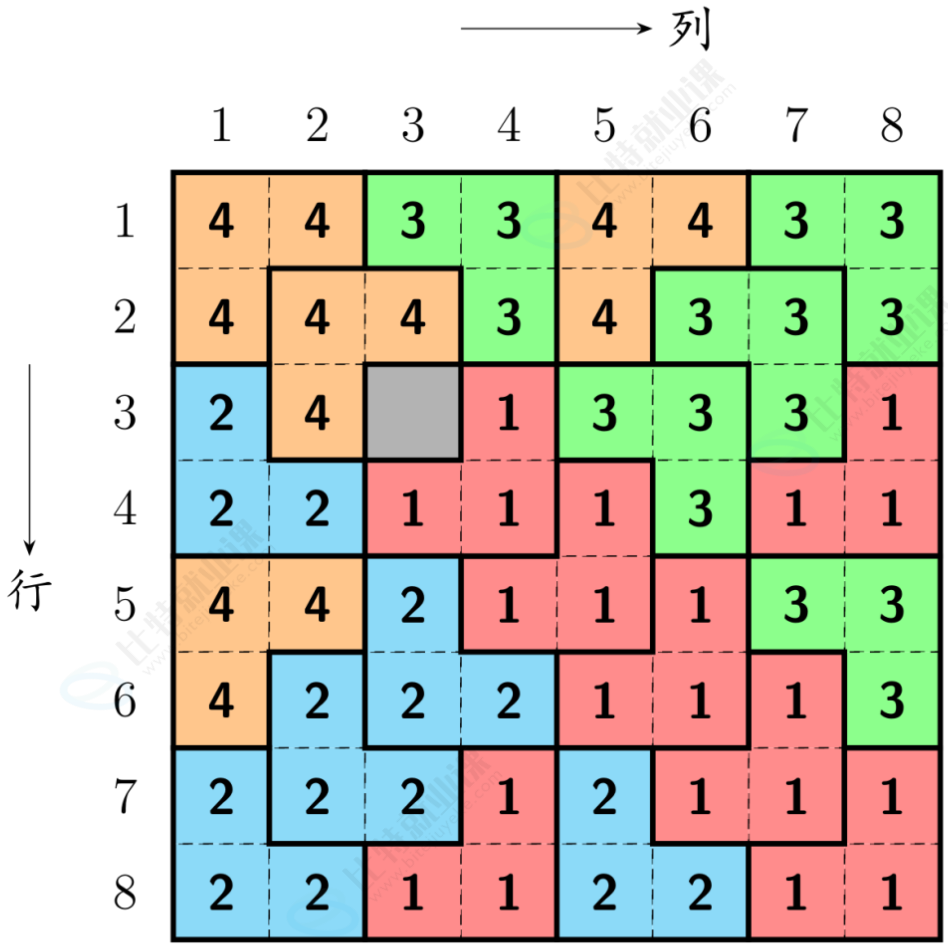
6 6 1

5 8 3

8 5 2

8 8 1

样例解释：

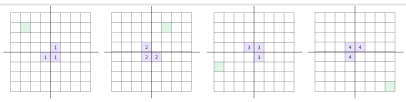


【解法】

非常经典的一道分治题目，也可以说是一道递归题目。

一维分治的时候，我们是从中间把整个区间切开，分成左右两部分（其实有时候我们可以三等分，就看具体问题是什么）。二维的时候，我们可以横着一刀竖着一刀，分成左上、右上、左下、右下四份。而这道题的矩阵长度正好是 2^k ，能够被不断平分下去。像是在暗示我们，要用分治，要用分治，要用分治.....

当我们把整个区间按照中心点一分为四后，四个区间里面必然有一个区间有缺口（就是公主的位置），那这四个区间不一样，那就没有相同子问题了。别担心，只要我们在中心位置放上一块地毯，四个区间就都有一个缺口了。如下图所示：



无论缺口在哪里，我们都可以在缺口对应的区间的角落，放上一个地毯。接下来四个区间都变成只有一个缺口的形式，就可以用递归处理子问题。

因此，我们拿到一个矩阵后的策略就是：

- 先四等分；
- 找出缺口对面的区间，放上一块地毯；
- 递归处理四个子问题

【参考代码】

代码块

```
#include <iostream>

using namespace std;

void dfs(int a, int b, int x, int y, int k)
{
    if(k < 1) return;
    int len = 1 << (k - 1);

    if(x < a + len && y < b + len) // 障碍物在左上角
    {
        cout << a + len << " " << b + len << " 1" << endl;
        dfs(a, b, x, y, k - 1);
        dfs(a, b + len, a + len - 1, b + len, k - 1);
        dfs(a + len, b, a + len, b + len - 1, k - 1);
        dfs(a + len, b + len, a + len, b + len, k - 1);
    }
    else if(x >= a + len && y >= b + len) // 障碍物在右下角
    {
        cout << a + len - 1 << " " << b + len - 1 << " 4" << endl;
        dfs(a, b, a + len - 1, b + len - 1, k - 1);
        dfs(a, b + len, a + len - 1, b + len, k - 1);
        dfs(a + len, b, a + len, b + len - 1, k - 1);
```

```cpp
            dfs(a + len, b + len, x, y, k - 1);
        }
        else if(x >= a + len) // 障碍物在左下角
        {
            cout << a + len - 1 << " " << b + len << " 3" << endl;
            dfs(a, b, a + len - 1, b + len - 1, k - 1);
            dfs(a, b + len, a + len - 1, b + len, k - 1);
            dfs(a + len, b, x, y, k - 1);
            dfs(a + len, b + len, a + len, b + len, k - 1);
        }
        else // 障碍物在右上角
        {
            cout << a + len << " " << b + len - 1 << " 2" << endl;
            dfs(a, b, a + len - 1, b + len - 1, k - 1);
            dfs(a, b + len, x, y, k - 1);
            dfs(a + len, b, a + len, b + len - 1, k - 1);
            dfs(a + len, b + len, a + len, b + len, k - 1);
        }
    }

    int main()
    {
        int k, x, y; cin >> k >> x >> y;
        dfs(1, 1, x, y, k);

        return 0;
    }
```