



历年 CSP 题目解析

仅为参考练习所用

作者: Ionlyn

组织: Shanxi University Algorithm Group

时间: December 30, 2021

版本: 1.0

特别声明

该书仅供内部学习使用，如果有侵权请联系作者。

信息学竞赛的发展，吸引越来越多的人加入了“卷”的行列。CCF CSP 的历年题解在网上也是随处可见，但题解质量参差不齐。很多题解只有标准答案，缺少题目分析；更有甚者无法通过答案，充满了分号大小写问题等错误。

本书的目的是为了实现以下几点：

- 提供规范的代码程序。这里的规范，既要具有程序的可读性，也要具备考场的简易性。
- 提供多样的解题思路。有些时候，网上的大佬往往一语道破问题求解的思路，但怎么想到的却往往不提。这里力求从部分分开始，逐渐深入，汇集众人智慧，逐步解决难题。
- 提供筛选的额外补充。做一道题的目的不是只做一道题，而是可以做到举一反三，但我们常常忽略这一点。考虑到 CSP 考试的形式也在不断变化，本书章节编排从新到旧。

感谢 [Elegant \$\text{\LaTeX}\$](#) 提供如此精美的模板，希望这本书能够给大家带来帮助。

lonlyn

December 30, 2021

目录

1	CCF CSP 认证总览	1
2	第 24 次认证 (2021 年 12 月)	2
2.1	题目及涉及知识点	2
2.2	202112-1 序列查询	3
2.2.1	50% 数据——模拟	3
2.2.1.1	思路	3
2.2.2	100% 数据——利用 $f(x)$ 单调性	3
2.2.2.1	思路	3
2.2.2.2	C++ 实现	3
2.2.3	100% 数据——阶段求和	4
2.2.3.1	思路	4
2.2.3.2	C++ 实现	4
2.3	202112-2 序列查询新解	6
2.3.1	与上一题的比较	6
2.3.2	70% 数据——计算出每个 $f(x), g(x)$ 的值	6
2.3.2.1	思路	6
2.3.3	100% 数据——对 $f(x), g(x)$ 都相同的区间进行求和处理	6
2.3.3.1	思路	6
2.3.3.2	C++ 实现	6
2.3.4	100% 思路——以 $f(x)$ 为单位, 讨论内部 $g(x)$ 求和	7
2.3.4.1	思路	7
2.3.4.2	C++ 实现	8
2.4	202112-3 登机牌号码	11
2.4.1	40% 数据——直接模拟	11
2.4.1.1	思路	11
2.4.1.2	C++ 实现	11
2.4.2	100% 数据——模拟 + 多项式除法	12
2.4.2.1	思路	12
2.4.2.2	C++ 实现	14
2.5	202112-4 磁盘文件操作	18
2.5.1	25% 数据——直接模拟	18
2.5.1.1	思路	18
2.5.2	100% 数据——离散化 + 线段树	18
2.5.2.1	思路	18
2.5.2.2	C++ 实现	19
2.5.3	100% 数据——动态开点线段树	26
2.5.3.1	思路	26
2.5.3.2	C++ 实现	26
2.6	202112-5 极差路径	32
2.6.1	$n \leq 5000$ 数据——枚举两点	32

2.6.1.1	思路	32
2.6.1.2	C++ 实现	32
3	第 23 次认证 (2021 年 09 月)	34
3.1	题目及设计知识点	34
3.2	202109-1 数组推导	35
3.2.1	100% 数据——模拟	35
3.2.1.1	思路	35
3.3	202109-2 非零段划分	36
3.3.1	70% 数据——模拟	36
3.3.1.1	思路	36
3.3.2	100% 数据——避免不必要更新	36
3.3.2.1	思路	36
3.3.2.2	C++ 实现	37
3.4	202109-3 脉冲神经网络	39
3.5	202109-4 收集卡牌	40
3.6	202109-5 箱根山岳险天下	41
4	第 22 次认证 (2021 年 04 月)	42
4.1	题目及设计知识点	42
4.2	202104-1 灰度直方图	43
4.3	202104-2 邻域均值	44
4.4	202104-3 DHCP 服务器	45
4.5	202104-4 校门外的树	46
4.6	202104-5 疫苗运输	47
5	第 21 次认证 (2020 年 12 月)	48
5.1	题目及设计知识点	48
5.2	202012-1 期末预测之安全指数	49
5.3	202012-2 期末预测之最佳阈值	50
5.4	202012-3 带配额的文件系统	51
5.5	202012-4 食材运输	52
5.6	202012-5 星际旅行	53

第 1 章 CCF CSP 认证总览

待补充。

第 2 章 第 24 次认证（2021 年 12 月）

2.1 题目及涉及知识点

题目编号	题目名称	知识点
202112-1	序列查询	数学
202112-2	序列查询新解	数学
202112-3	登机牌条码	模拟，多项式除法
202112-4	磁盘文件操作	线段树
202112-5	极差路径	树分治

2.2 202112-1 序列查询

2.2.1 50% 数据——模拟

2.2.1.1 思路

模拟一下这个过程，计算出每一个 $f(i)$ 后加起来即可。

考虑针对确定的 x ，如何求解 $f(x)$ 。我们可以从小到大枚举 A 中的数，枚举到第一个大于等于 x 的数即可。注意末尾的判断。

枚举 x 时间复杂度 $O(N)$ ，计算 $f(x)$ 时间复杂度 $O(n)$ ，整体时间复杂度 $O(nN)$ 。

2.2.2 100% 数据——利用 $f(x)$ 单调性

2.2.2.1 思路

为了方便，设 $f(n+1) = \infty$ 。

通过模拟，可以得到一个显然的结论：

定理 2.1 ($f(x)$ 的单调性)

对于 $x, y \in [0, N)$ ，若 $x \leq y$ ，则 $f(x) \leq f(y)$ 。



那么，我们可以从小到大枚举 x ，同时记录目前 $f(x)$ 的值，设为 y ，那么 A_{y+1} 是第一个大于 x 的数。当需要计算 $f(x+1)$ 的时候，我们从小到大依次判断 A_{y+1}, A_{y+2}, \dots 是否满足条件，直到遇到第一个大于 $f(x+1)$ 的数 A_z ，那么 $f(x+1) = z - 1$ 。之后，在 $f(x+1)$ 的基础上以同样的步骤求 $f(x+2)$ ，直到求完所有的值。

考虑该算法的时间复杂度，枚举 x 的复杂度是 $O(N)$ ，而 A 数组中每个数对多被枚举一次，枚举所有 x 的整体复杂度 $O(n)$ ，可以得到整体复杂度 $O(N+n)$ 。

2.2.2.2 C++ 实现

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstring>
#include <iostream>
using namespace std;
#define ll long long
#define il inline
const int maxn = 210;
int n, N;
int a[maxn];
ll ans = 0;
int main() {
    scanf("%d%d", &n, &N);
    for (int i = 1; i <= n; ++i) {
        scanf("%d", &a[i]);
    }
    int cur = 0;
    for (int i = 0; i < N; ++i) {
        while (cur < n && a[cur + 1] <= i)
```

```

        ++cur;
        ans += cur;
    }
    printf("%lld\n", ans);
    return 0;
}

```

2.2.3 100% 数据——阶段求和

2.2.3.1 思路

在提示中，指出了可以将 $f(x)$ 相同的值一起计算。现在需要解决的问题是如何快速确定 $f(x)$ 值相等的区间。

通过观察和模拟可以发现，随着 x 增大， $f(x)$ 只会在等于某个 A 数组的值时发生变化。更具体的说，对于某个属于 A 数组的值 A_i 来说， $[A_i, A_{i+1} - 1]$ 间的 $f(x)$ 值是相同的，这样的数共有 $A_{i+1} - A_i$ 个。

也可以以另一种方式理解：对于一个值 y ，考虑有多少 x 满足 $f(x) = y$ 。当 $x < A_y$ 时， $f(x) < y$ ，当 $x \geq A_{y+1}$ 时， $f(x) > y$ 。只有 $x \in [A_y, A_{y+1}]$ 时才能得到 $f(x) = y$ 。

得到范围后，我们就可以根据 A 数组来进行求和计算。

考虑 $f(x) = n$ 的处理：我们可以得知满足 $f(x) = n$ 的 x 共有 $N - A_n$ 个，根据上文推算，我们可以将 A_{n+1} 设置为 $A_n + (N - A_n) = N$ 即可等效替代。

时间复杂度 $O(n)$ 。

2.2.3.2 C++ 实现

```

#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstring>
#include <iostream>
using namespace std;
#define ll long long
#define il inline
const int maxn = 210;
int n, N;
int a[maxn];
ll ans = 0;
int main() {
    scanf("%d%d", &n, &N);
    for (int i = 1; i <= n; ++i) {
        scanf("%d", &a[i]);
    }
    a[n + 1] = N;
    for (int i = 1; i <= n + 1; ++i) {
        // 处理区间 [A(i-1), A(i)] 的 f(x) 值的和
        ans += 1ll * (a[i] - a[i - 1]) * (i - 1);
    }
    printf("%lld\n", ans);
    return 0;
}

```



```
}
```

2.3 202112-2 序列查询新解

2.3.1 与上一题的比较

1. 上一题是求和，而本题要求求绝对值的和，无法转化为两者求差的形式。
2. $f(x), g(x)$ 的变化是各自独立的，当 $f(x)$ 改变时， $g(x)$ 可能不变，也可能改变； $g(x)$ 对 $f(x)$ 也是如此。
3. 对于所有数据点， n 和 N 都增大了许多。如果复杂度涉及到 n ，则最多预计为 $O(n \log n)$ 级别；如果涉及到 N ，则必须是亚线性级别。

2.3.2 70% 数据——计算出每个 $f(x), g(x)$ 的值

2.3.2.1 思路

由于 1,2 条限制，我们无法直接对 $f(x), g(x)$ 分别进行处理。但我们可以求出每个 $f(x), g(x)$ 的值，再计算求和即可。

$f(x)$ 的计算同第一问，任意方法皆可。单个 $g(x)$ 的值可以直接 $O(1)$ 求得。


2.3.3 100% 数据——对 $f(x), g(x)$ 都相同的区间进行求和处理

2.3.3.1 思路

注：为了防止混淆，将题目中的 r 改为 $ratio$ 。

假设 $f(x)$ 一共有 x 种取值， $g(x)$ 一共有 y 种取值。直接来看 $f(x), g(x)$ 的组合一共有 xy 种，但注意到 $f(x), g(x)$ 都是单调不递减函数，所以真正的组合只有 $x + y$ 种。

在第一题中已经说明 $f(x)$ 的取值范围为 $[0, n]$ ，在 $O(n)$ 级别。考虑 $g(x)$ 的取值情况，将 $ratio$ 的公式带入可以得到 $g(x) = \lfloor \frac{x}{ratio} \rfloor = \lfloor \frac{x}{\frac{N}{n+1}} \rfloor$ 。由于 x 取值有 N 种，所以 $g(x)$ 的取值是 $O(\frac{N}{\frac{N}{n+1}}) = O(n)$ 级别的。所以，整体复杂度为 $O(n + n) = O(n)$ 。

 **提示** 有些时候，题目给出的某些量的值会比较特殊（如本题 $ratio = \lfloor \frac{N}{n+1} \rfloor$ ），代表着出题人可能想要隐藏某些做法，但不得不为了让时间复杂度正确而妥协。在没有思路的时候，可以作为突破口。

考虑范围问题：假设当前左端点为 l ，如何找到右端点 r ，满足 $f(l) = f(l+1) = \dots = f(r), g(l) = g(l+1) = \dots = g(r)$ 且 $f(l) \neq f(r+1)$ or $g(l) \neq g(r+1)$ 。我们可以对 $f(x), g(x)$ 分别考虑：

1. 对于 $f(x)$ 而言，第一个满足 $f(x) = f(l) + 1$ 的 x 值为 A_{f_l+1} 。
2. 对于 $g(x)$ 而言，因为分母 $ratio$ 是固定的，所以值相同的区间长度也是固定为 $ratio$ 。我们不妨将 $g(x)$ 值相同的数字为一组，则可以得到 $[0, ratio - 1], [ratio, 2 \cdot ratio - 1], \dots, [n \cdot ratio, (n+1) \cdot ratio - 1], \dots$ 这样的分组序列，每组的 $g(x)$ 取值为 $0, 1, \dots, n, \dots$ 。可以发现，对于一个数 l ，其所属的分组是 $\lfloor \frac{l}{ratio} \rfloor$ ，也即 $g(l)$ ；而下一组开始的第一个数为 $ratio \cdot (g(l) + 1)$ ，从而可以得到右端点 $r = ratio \cdot (g(l) + 1) - 1$ 。
3. 在 $f(x), g(x)$ 计算得到的右端点中，选择较小的一个作为计算的右端点。

计算完一段后，设 $l = r + 1$ 继续计算下一段，直到结束。时间复杂度 $O(n)$ 。

2.3.3.2 C++ 实现

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

```

#define ll long long
const int maxn = 100010;
int n, N;
int a[maxn];
int rat, f, g;
ll ans = 0;
int main() {
    scanf("%d%d", &n, &N);
    for (int i = 1; i <= n; ++i) {
        scanf("%d", &a[i]);
    }
    rat = N / (n + 1); // 为了防止冲突，题目中 r 改为 rat
    int cur = 0; // 用来计算 f(x)
    bool flag = false; // 如果需要更新 f(x) 值，则 flag = true
    for (int l = 0, r; l < N; l = r + 1) {
        flag = false;
        // 利用 f(x) 的值确定 r 的范围
        if (cur < n)
            r = a[cur + 1] - 1;
        else
            r = N - 1;
        // 判断 f(x), g(x) 谁先变化，选择较小的区间
        if ((l / rat + 1) * rat - 1 < r) {
            // 如果 g(x) 先变化，则改为选择 g(x)
            r = (l / rat + 1) * rat - 1;
        } else {
            // 如果 f(x) 先变化，则确定选择 f(x)，计算后更新 f(x)
            flag = true;
        }
        // [l, r] 区间内的值是相等的，可以求和
        ans += 1ll * (r - l + 1) * abs(l / rat - cur);
        // 更新 f(x) 的值
        if (flag)
            ++cur;
    }
    printf("%lld\n", ans);
    return 0;
}

```

2.3.4 100% 思路——以 $f(x)$ 为单位，讨论内部 $g(x)$ 求和

感谢 [DoctorLazy](#) 提供的宝贵思路，原文可以查看 [第二题 100 分题解 by DoctorLazy.md](#)。

2.3.4.1 思路

和上文同样的思路，我们需要进行区间求和来降低复杂度。一种思路是，整体上对 $f(x)$ 进行求和，而在内部对 $g(x)$ 的情况进行分类讨论。

我们单独考虑每一个 $f(x)$ 的区间，每个区间上 $f(x)$ 的值相同。可以观察到，对于一个区间上的下标 i ，可

能存在 $g(i) \geq f(i)$ ，也可能存在 $g(i) < f(i)$ 。求绝对值时，前者用 $g(x) - f(x)$ ，后者用 $f(x) - g(x)$ 。

观察到，由于 $g(x)$ 单调不减的性质，我们可以得到：对于该区间，一定存在一个下标 p ，如同一个分界线，当 $i \geq p$ 时，有 $g(i) \geq f(i)$ ，当 $i < p$ ，有 $g(i) < f(i)$ 。这样，就把该区间分成了两个“小区间”。我们就可以用“乘法思想”来加速两个“小区间”的求解了。

更规范些，用 $contribution(i)$ 代表区间 $[A_i, A_{i+1})$ 对答案的贡献，用 $len(l, r) = r - l + 1$ 代表区间长度，用公式可以表达为：

$$\begin{aligned} contribution(i) = & len(A_i, p-1) \times f(x) - \sum_{x=A[i]}^{p-1} g(x) \\ & + \sum_{x=p}^{A_{i+1}-1} g(x) - len(p, A_{i+1}-1) \times f(x) \end{aligned}$$

上式中， $f(x)$ 是一个常数，所以乘以“小区间”的长度即可； $g(x)$ 的求和，大家可以发挥数学思维：因为 $g(x)$ 其实非常规律，它的每一块是定长的，我们可以通过除法和取余来确定相同值的数量，再利用乘法思想求和，灵活实现，在 $O(n)$ 时间内求出即可。 p 的具体值可以通过在 $g(x)$ 中二分查找， $O(\log n)$ 时间内求出， n 为区间的长度。

一个例子：

x	...	4	5	6	7	...
$f(x)$...	2	2	2	2	...
$g(x)$...	1	1	2	2	...

上面的表格截取了一个小区间， $f(x)$ 的值固定 2， $p = 6$ ，那么 p 的左边用 $f(x) - g(x)$ ， p 的右边用 $g(x) - f(x)$ 。

当然，有一个特殊的边界情况，那就是该区间上有可能所有的 $g(x)$ 都绝对大于或小于 $f(x)$ ，这时候 p 可能会在区间外。该情况大家可以对 p 设置初值，然后在写完二分后加以判断即可。

2.3.4.2 C++ 实现

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <map>
#include <queue>
#include <vector>
using namespace std;
typedef long long LL;
typedef unsigned long long uLL;
typedef pair<int, int> pii;
const int mod = 1e9 + 7;
const int maxn = 1e5 + 5;

LL N, n;
LL arr[maxn]; // 题中 A 数组
```

```

vector<LL> f; // 存储每个区间上f的值
vector<pii> pos; // 存储每个区间的边界，是左闭右闭
LL r, ans; // 题中的 r, ans为计算的答案

// 下面的函数用于计算g(x)在区间上的和
// 这一步比较细，具体可以灵活实现
// 下面的思路还是比较冗杂的
LL totG(LL be, LL ed) {
    // 右边界小于左边界，返回0
    if (ed < be) {
        return 0;
    }
    // 两边界重合，返回一个g值
    if (be == ed) {
        return be / r;
    }
    // 如果两边界g值相同，返回该值乘以区间长度
    if (be / r == ed / r) {
        return (be / r) * (ed - be + 1);
    }
    // 将区间分为三部分，分别累计
    LL tot = 0;
    // 对于左边界，其值为be/r,数目为 r - be % r
    tot += (r - (be % r)) * (be / r);
    // 对于右边界，其值为ed/r,数目为 ed % r + 1
    tot += (ed % r + 1) * (ed / r);
    // 对于不在边界上的g值，我们用等差数列求和公式
    if (ed / r - be / r > 1) {
        be = be / r + 1;
        ed = ed / r - 1;
        tot += r * ((be + ed) * (ed - be + 1) / 2);
    }
    return tot;
}

void solve() {
    // 输入
    scanf("%lld%lld", &n, &N);
    r = N / (n + 1);
    for (int i = 1; i <= n; i++) {
        scanf("%d", &arr[i]);
    }
    // 根据数组，生成f(x)的每个区间，值存入f，区间边界存入pos
    LL last = 0; // 记录上一个边界
    // 这里的逻辑参考第一题
    for (int i = 1; i <= n; i++) {
        if (arr[i] > arr[last]) {
            f.push_back(last);
            pos.push_back({arr[last], arr[i] - 1});
        }
    }
}

```

```

        last = i;
    }
}
// 单独处理下最后一个区间，即[A[n],N-1]
f.push_back(n);
pos.push_back({arr[last], N - 1});
// 对于每个f区间，将g分成两个小区间
int si = f.size();
for (int i = 0; i < si; i++) {
    LL num = f[i];
    LL be = pos[i].first;
    LL ed = pos[i].second;
    LL length = ed - be + 1;
    // 因为be和ed在二分过程其值发生变化，所以下面再存一份
    LL bbe = be, eed = ed;
    // 下面使用二分，在g中寻找分界p
    LL pin = -1;
    while (be <= ed) {
        LL mid = (be + ed) / 2;
        LL cur = mid / r;
        if (cur >= num) {
            pin = mid;
            ed = mid - 1;
        } else {
            be = mid + 1;
        }
    }
    // 如果f的值一直大于g，p值不会被二分的过程赋值，所以还是初值
    if (pin == -1) {
        ans += num * length - totG(bbe, eed);
    } else {
        // 左边的用f-g，右边用g-f。就算g的值一直大于f，即左边的部分长度为0
        ans += num * (pin - bbe) - totG(bbe, pin - 1);
        ans += totG(pin, eed) - num * (eed - pin + 1);
    }
}
printf("%lld", ans);
}

int main() {
    int t;
    t = 1;
    while (t--) {
        solve();
    }
    return 0;
}

```

2.4 202112-3 登机牌号码

2.4.1 40% 数据——直接模拟

2.4.1.1 思路

这一部分数据满足 $s = -1$ ，即校验码为空。我们按照题目要求进行对应操作即可，大体分为以下几个步骤：

1. 得到数字序列，注意不同模式的切换以及最后的补全。
2. 将得到的数字转换为码字。
3. 根据有效数据区每行能容纳的码字数 w 及目前码字个数，在末尾补充码字。注意不要忽略长度码字。
4. 输出结果。

2.4.1.2 C++ 实现

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <vector>
using namespace std;
const int maxn = 210;
const int mod = 929;
int w, lev;
char s[100010];
int n;
vector<int> numberList; // 数字序列
vector<int> codeWord; // 码字序列
int currentMode; // 目前编码器模式。0:大写模式 1:小写模式 2:数字模式
void checkmode(char c) {
    /*
        检查将要输出的下个字符与目前模式是否匹配，
        若不匹配，则输出对应更改模式步骤。
    */
    if (c >= '0' && c <= '9') {
        if (currentMode != 2) {
            numberList.push_back(28);
            currentMode = 2;
        }
    } else if (c >= 'a' && c <= 'z') {
        if (currentMode != 1) {
            numberList.push_back(27);
            currentMode = 1;
        }
    } else if (c >= 'A' && c <= 'Z') {
        if (currentMode == 1) {
            numberList.push_back(28);
            numberList.push_back(28);
        }
    }
}
```

```

        currentMode = 0;
    }
    if (currentMode == 2) {
        numberList.push_back(28);
        currentMode = 0;
    }
}
}
int main() {
    scanf("%d%d", &w, &lev); // lev 表示校验级别
    scanf("%s", s);
    n = strlen(s);
    // 步骤一：得到数字序列
    currentMode = 0; // 初始为大写模式
    for (int i = 0; i < n; ++i) {
        checkmode(s[i]);
        if (s[i] >= '0' && s[i] <= '9') {
            numberList.push_back(s[i] - '0');
        } else if (s[i] >= 'a' && s[i] <= 'z') {
            numberList.push_back(s[i] - 'a');
        } else if (s[i] >= 'A' && s[i] <= 'Z') {
            numberList.push_back(s[i] - 'A');
        }
    }
    if (numberList.size() % 2)
        numberList.push_back(29);
    // 步骤二：转换为码字
    for (int i = 0; i < numberList.size(); i += 2) {
        codeWord.push_back(30 * numberList[i] + numberList[i + 1]);
    }
    // 步骤三：补充码字
    while ((1 + codeWord.size()) % w != 0) {
        codeWord.push_back(900);
    }
    // 步骤四：输出结果
    codeWord.insert(codeWord.begin(), codeWord.size() + 1);
    for (int i = 0; i < codeWord.size(); ++i) {
        printf("%d\\n", codeWord[i]);
    }
    return 0;
}

```

2.4.2 100% 数据——模拟 + 多项式除法

2.4.2.1 思路

这部分数据要求我们对校验码进行处理，所以步骤变为：

1. 得到数字序列，注意不同模式的切换以及最后的补全。

2. 将得到的数字转换为码字。
3. 根据有效数据区每行能容纳的码字数 w 、目前码字数以及校验码的位数，在末尾补充码字。注意不要忽略长度码字。
4. 输出数据码部分结果。
5. 计算得出校验码，并输出。

校验码的位数能比较方便得出，关键在于校验码的计算。考虑关键公式：

$$x^k d(x) \equiv q(x)g(x) - r(x)$$

其中 $d(x)$ 是 $n-1$ 次多项式（已知）， $g(x)$ 是 k 次多项式（已知），未知项有 $q(x), r(x)$ ，其中 $r(x)$ 为所求。考虑消去 $q(x)$ 的影响：可以在两端同时对 $g(x)$ 取余，则 $q(x)g(x)$ 项会被直接消去，可以化所求式为：

$$x^k d(x) \equiv -r(x) \pmod{g(x)}$$

所以目前问题转化为求解 $x^k d(x) \pmod{g(x)}$ 。

定义 2.1 (多项式带余除法)

若 $f(x)$ 和 $g(x)$ 是两个多项式，且 $g(x)$ 不等于 0，则存在唯一的多项式 $q(x)$ 和 $r(x)$ ，满足：

$$f(x) = q(x)g(x) + r(x)$$

其中 $r(x)$ 的次数小于 $g(x)$ 的次数。此时 $q(x)$ 称为 $g(x)$ 除 $f(x)$ 的商式， $r(x)$ 称为余式。

定义 2.2 (多项式长除法)

求解多项式带余除法的一种方法，步骤如下：

1. 把被除式、除式按某个字母作降幂排列，并把所缺的项用零补齐；
2. 用被除式的第一项除以除式第一项，得到商式的第一项；
3. 用商式的第一项去乘除式，把积写在被除式下面（同类项对齐），消去相等项，把不相等的项结合起来；
4. 把减得的差当作新的被除式，再按照上面的方法继续演算，直到余式为零或余式的次数低于除式的次数时为止。

下面展示的是一个多项式长除法的例子：

$$\begin{array}{r}
 6x^5 + 286x^4 + 4111x^3 + 42431x^2 + 398624x + 3638751 \\
x^2 - 12x + 27) 6x^7 + 214x^6 + 841x^5 + 821x^4 + 449x^3 + 900x^2 \\
 - 6x^7 + 72x^6 - 162x^5 \\
\hline
 286x^6 + 679x^5 + 821x^4 \\
 - 286x^6 + 3432x^5 - 7722x^4 \\
\hline
 4111x^5 - 6901x^4 + 449x^3 \\
 - 4111x^5 + 49332x^4 - 110997x^3 \\
\hline
 42431x^4 - 110548x^3 + 900x^2 \\
 - 42431x^4 + 509172x^3 - 1145637x^2 \\
\hline
 398624x^3 - 1144737x^2 \\
 - 398624x^3 + 4783488x^2 - 10762848x \\
\hline
 3638751x^2 - 10762848x \\
 - 3638751x^2 + 43665012x - 98246277 \\
\hline
 32902164x - 98246277
\end{array}$$

得到求解多项式带余除法的步骤后，考虑求解 $r(x)$ 的步骤：

1. 计算 $g(x) = (x-3)(x-3^2)\cdots(x-3^k)$;
2. 计算 $x^k d(x)$;
3. 计算 $x^k d(x) \bmod g(x)$ ，得到 $-r(x)$;
4. 对得到的每一项取反即可得到 $r(x)$ 。

计算 $g(x)$ ：考虑到每一次多项式乘以的因子都是 $(x-a)$ 的格式，所以可以把 $A(x-a)$ 的多项式相乘转化为 $xA - aA$ 的格式。 xA 可以通过整体移项实现；在移项后，原本在 x^i 的系数成为 x^{i+1} 的系数，所以可以在一个数组上，从低位到高位依次计算，得到结果。

计算 $x^k d(x)$ ：这部分比较简单，将低 k 位的系数赋 0，再将已计算出的数据位放入对应位置即可。

计算 $x^k d(x) \bmod g(x)$ ：利用上文提到的多项式长除法即可。本题 $g(x)$ 的最高位系数恒为 1，简化了计算。

2.4.2.2 C++ 实现

```

#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <vector>
using namespace std;
const int maxn = 210;
const int mod = 929;
int w, lev;
char s[100010];
int n;
vector<int> numberList;
vector<int> codeWord;
int verifyCodeLen; // 校验码长度
int currentMode;
void checkmode(char c) {
    /*

```

检查将要输出的下个字符与目前模式是否匹配，
若不匹配，则输出对应更改模式步骤。

```

*/
if (c >= '0' && c <= '9') {
    if (currentMode != 2) {
        numberList.push_back(28);
        currentMode = 2;
    }
} else if (c >= 'a' && c <= 'z') {
    if (currentMode != 1) {
        numberList.push_back(27);
        currentMode = 1;
    }
} else if (c >= 'A' && c <= 'Z') {
    if (currentMode == 1) {
        numberList.push_back(28);
        numberList.push_back(28);
        currentMode = 0;
    }
    if (currentMode == 2) {
        numberList.push_back(28);
        currentMode = 0;
    }
}
}

vector<int> get_gx(int k) {
    // 根据 k 计算 g(x)
    vector<int> res;
    res.push_back(mod - 3);
    res.push_back(1);
    int a0 = 3;
    for (int i = 2; i <= k; ++i) {
        a0 = (a0 * 3) % mod;
        res.insert(res.begin(), 0); // 在最低位插入 1，即整体次数 +1
        for (int j = 0; j < i; ++j) {
            res[j] = (res[j] - (a0 * res[j + 1]) % mod + mod) % mod;
        }
    }
    return res;
}

void get_verify_code() {
    // 计算校验码并输出
    vector<int> tmp;
    vector<int> g = get_gx(verifyCodeLen);
    // 初始化  $x^{kd}(x)$ 
    for (int i = 1; i <= verifyCodeLen; ++i) {
        tmp.push_back(0);
    }
}

```

```

for (int i = codeWord.size() - 1; i >= 0; --i) {
    tmp.push_back(codeWord[i]);
}
// 多项式长除法计算结果
for (int i = tmp.size() - 1; i >= verifyCodeLen; --i) {
    int val = tmp[i];
    for (int j = 0; j < g.size(); ++j) {
        tmp[i - j] =
            (tmp[i - j] - (val * g[g.size() - 1 - j]) % mod + mod) % mod;
    }
}
// 将 -r(x) 转化为 r(x)
for (int i = 0; i < verifyCodeLen; ++i) {
    // 注意: 不能直接 mod - tmp[i], 因为 tmp[i] 可能为 0
    tmp[i] = (mod - tmp[i]) % mod;
}
// 输出结果
for (int i = verifyCodeLen - 1; i >= 0; --i) {
    printf("%d\n", tmp[i]);
}
}

int main() {
    scanf("%d%d", &w, &lev);
    scanf("%s", s);
    n = strlen(s);
    // 步骤一: 得到数字序列
    currentMode = 0;
    for (int i = 0; i < n; ++i) {
        checkmode(s[i]);
        if (s[i] >= '0' && s[i] <= '9') {
            numberList.push_back(s[i] - '0');
        } else if (s[i] >= 'a' && s[i] <= 'z') {
            numberList.push_back(s[i] - 'a');
        } else if (s[i] >= 'A' && s[i] <= 'Z') {
            numberList.push_back(s[i] - 'A');
        }
    }
    if (numberList.size() % 2)
        numberList.push_back(29);
    // 步骤二: 转换为码字
    for (int i = 0; i < numberList.size(); i += 2) {
        codeWord.push_back(30 * numberList[i] + numberList[i + 1]);
    }
    if (lev == -1)
        verifyCodeLen = 0;
    else {
        verifyCodeLen = 1;
        for (int i = 0; i <= lev; ++i) {
            verifyCodeLen *= 2;
        }
    }
}

```

```
    }  
}  
// 步骤三：补充码字  
while ((1 + verifyCodeLen + codeWord.size()) % w != 0) {  
    codeWord.push_back(900);  
}  
codeWord.insert(codeWord.begin(), codeWord.size() + 1);  
// 步骤四：输出数据码结果  
for (int i = 0; i < codeWord.size(); ++i) {  
    printf("%d\\n", codeWord[i]);  
}  
// 步骤五：如果有校验码，则计算并输出  
if (verifyCodeLen != 0) {  
    get_verify_code();  
}  
return 0;  
}
```

2.5 202112-4 磁盘文件操作

2.5.1 25% 数据——直接模拟

2.5.1.1 思路


我们按照题目要求进行对应操作即可，注意每一个要求执行的条件：

- 写入操作：从左往右依次执行，直到第一个不被自己占用的位置。除了第一个点就被其他程序占用以外，必然会写入。遇到自己占用，则覆盖。
- 删除操作：同时整体进行，要求所有位置都被目前程序占用。要么全删，要么不做任何更改。
- 恢复操作：同时整体进行，要求所有位置都不被占用，且上次占用程序为目前程序。要么全恢复，要么不做任何更改。遇到自己占用，则不做任何更改。
- 读入操作：读取占用程序和数值，若未被占用，则输出 0 0。

2.5.2 100% 数据——离散化 + 线段树

2.5.2.1 思路

通过这道题的操作要求等，我们可以大致推测出这道题可能需要使用线段树。

 **提示** 如果没什么思路，可以拿各种数据结构往上套。例如本题，因为涉及区间修改、单点查询，对于树状数组来说负担太重，我们可以尝试其他数据结构。如果使用平衡树，则一般是要求出第 k 大数，或者是序列翻转类问题，对于本题来说不太契合。其他数据结构不再一一列举。综合考虑下，线段树是比较符合要求的。

2.5.2.1.1 考虑线段树的做法 先不考虑线段树的内存空间问题，我们分析一下如何用线段树解决这道题目。

考虑我们需要维护的量，目前已知的有磁盘位置的值、目前占用程序 id、上次占用程序 id。

在这里，我们假设一个位置未被占用和被 id 为 0 的程序占用是等价的。

- 写入操作：可以划分为找到写入右边界和直接写入两个操作。
直接写入操作就是直接的线段树区间修改，而划分操作需要知道该区间被占用的位置是否属于将要写入的 id。我们不妨将这个量设为 id1。
- 删除操作：可以划分为判断是否可删和直接删除两个操作。
直接删除操作就是直接的线段树区间修改，而判断是否可删需要知道该区间所有的位置是否属于将要写入的 id。我们不妨将这个量设为 id2，注意 id1 与 id2 的区别——是否允许包含未被占用的程序。
- 恢复操作：可以划分为判断是否可恢复和直接恢复两个操作。
该操作与删除操作类似，不过需要注意的是判断时需要判断目前占用的 id 和上次被占用的 id。
- 读取操作：可以划分为查询占用程序 id 和查询值两个操作。
该操作是相对比较质朴的单点查询，当然也可以处理为区间。

通过以上分析，我们得到了需要维护的量：值、有关目前占用程序 id 的两个量、上次被占用的程序 id。我们考虑每个量针对父子之间的维护。

- 值 val：每个节点代表取值的多少，若左右子节点不同则设为一个不存在的值。因为我们是单点查询，所以不用担心查询到不存在的值的问题。
- 被占用位置程序 id1：
 - 若左右子节点都未被占用，则该节点标记为未占用；
 - 若左右子节点中存在不唯一节点，则该节点标记为不唯一。
 - 若左右子节点中一个节点未占用，则该节点标记为另一个非空节点的标记；
 - 若左右子节点都非空且相等，则该节点标记为任意一个节点；
 - 若左右子节点都非空且不等，则该节点标记为不唯一；

- 被占用位置程序 id2: 为了方便进行讨论, 将未被程序占用节点视为被 id 为 0 的程序占用。
 - 若左右子节点中存在不唯一节点, 则该节点标记为不唯一。
 - 若左右子节点相等, 则该节点标记为任意一个节点;
 - 若左右子节点不等, 则该节点标记为不唯一;
- 上一次被占用程序 lid: 与 id2 相同。
 - 若左右子节点中存在不唯一节点, 则该节点标记为不唯一。
 - 若左右子节点相等, 则该节点标记为任意一个节点;
 - 若左右子节点不等, 则该节点标记为不唯一;

2.5.2.1.2 解决空间问题 理解线段树的解法之后, 就会出现另一个问题: 空间达到了 $1e9$ 级别, 肯定会 MLE。

我们可以从另一个角度考虑: 一共有 2×10^5 次询问, 每次最多操作涉及一个区间, 可以用两个端点表示。考虑到临界处的影响, 一次操作最多会涉及 4 个点 (比如原来的区间是 $[1, 10]$, 我们更改了区间 $[3, 5]$, 那么得到的区间为 $[1, 2], [3, 5], [6, 10]$, 多出了 2, 3, 5, 6 四个点)。那么总体来看, 涉及到的点最多有 $2 \times 10^5 \times 4 = 8 \times 10^5$ 个。

我们可以维持这些点的相对大小关系, 而将其投影到一个值域较小的区域, 就可以减少空间占用了。这种方法称为离散化。

定义 2.3 (离散化)

把无限空间中有限的个体映射到有限的空间中去, 以此提高算法的时空效率。通俗的说, 离散化是在不改变数据相对大小的条件下, 对数据进行相应的缩小。离散化本质上可以看成是一种哈希, 其保证数据在哈希以后仍然保持原来的全/偏序关系。

离散化的步骤:

1. 统计所有出现过的数字, 在知道确切上界的时候可以用数组, 不清楚情况下可以用 vector;
2. 对所有的数据排序 (sort)、去重 (unique);
3. 对于每个数, 其离散化后的对应值即为其在排序去重后数组中的位置, 可以通过二分 (lower_bound) 确定。

这道题允许我们提前将所有可能出现的数记录下来 (当然不是所有的题目都允许这样), 所以这道题就解决了。线段树节点的个数与询问个数成比例, 时间复杂度 $O(k \log k)$ 。

2.5.2.2 C++ 实现

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <queue>
#include <vector>
using namespace std;
const int maxn = 200010;
const int INF = 1e9 + 10;
int n, m, k;
#define ls o << 1
#define rs ls | 1
struct treenode {
    // 当前节点的值, 若不唯一则为 INF; lazy 为 INF 表示无延迟更新
```

```

    int val, lazy_val;
    // 当前占用 id, 若存在除 0 以外两种 id 则为 -1; lazy 为 -1 表示无延迟更新
    int id1, lazy_id1;
    // 当前占用 id, 若存在两种 id 则为 -1; lazy 为 -1 表示无延迟更新
    int id2, lazy_id2;
    // 上次占用 id, 若存在两种 id 则为 -1; lazy 为 -1 表示无延迟更新
    int lid, lazy_lid;
} tree[maxn << 5];

void pushup(int o) {
    // 线段树上传操作, 合并左右子树结果
    // val 的合并
    tree[o].val = (tree[ls].val == tree[rs].val) ? tree[ls].val : INF;
    // id1 的合并
    if (tree[ls].id1 == -1 || tree[rs].id1 == -1) {
        tree[o].id1 = -1;
    } else if (tree[ls].id1 == tree[rs].id1) {
        tree[o].id1 = tree[ls].id1;
    } else if (tree[ls].id1 == 0) {
        tree[o].id1 = tree[rs].id1;
    } else if (tree[rs].id1 == 0) {
        tree[o].id1 = tree[ls].id1;
    } else {
        tree[o].id1 = -1;
    }
    // id2 的合并
    if (tree[ls].id2 == -1 || tree[rs].id2 == -1) {
        tree[o].id2 = -1;
    } else if (tree[ls].id2 == tree[rs].id2) {
        tree[o].id2 = tree[ls].id2;
    } else {
        tree[o].id2 = -1;
    }
    // lid 的合并
    if (tree[ls].lid == -1 || tree[rs].lid == -1) {
        tree[o].lid = -1;
    } else if (tree[ls].lid == tree[rs].lid) {
        tree[o].lid = tree[ls].lid;
    } else {
        tree[o].lid = -1;
    }
}

void pushdown(int o) {
    // 线段树标记下传操作
    if (tree[o].lazy_val != INF) {
        tree[ls].val = tree[rs].val = tree[o].lazy_val;
        tree[ls].lazy_val = tree[rs].lazy_val = tree[o].lazy_val;
        tree[o].lazy_val = INF;
    }
}

```



```

}
if (tree[o].lazy_id1 != -1) {
    tree[ls].id1 = tree[rs].id1 = tree[o].lazy_id1;
    tree[ls].lazy_id1 = tree[rs].lazy_id1 = tree[o].lazy_id1;
    tree[o].lazy_id1 = -1;
}
if (tree[o].lazy_id2 != -1) {
    tree[ls].id2 = tree[rs].id2 = tree[o].lazy_id2;
    tree[ls].lazy_id2 = tree[rs].lazy_id2 = tree[o].lazy_id2;
    tree[o].lazy_id2 = -1;
}
if (tree[o].lazy_lid != -1) {
    tree[ls].lid = tree[rs].lid = tree[o].lazy_lid;
    tree[ls].lazy_lid = tree[rs].lazy_lid = tree[o].lazy_lid;
    tree[o].lazy_lid = -1;
}
}

void build(int o, int l, int r) {
    // 线段树初始化建树
    if (l == r) {
        tree[o].val = 0;
        tree[o].lazy_val = INF;
        tree[o].id1 = tree[o].id2 = tree[o].lid = 0;
        tree[o].lazy_id1 = tree[o].lazy_id2 = tree[o].lazy_lid = -1;
        return;
    }
    int mid = (l + r) >> 1;
    build(ls, l, mid);
    build(rs, mid + 1, r);
    tree[o].lazy_val = INF;
    pushup(o);
}

#define ALLOK -2
int find_right(int o, int l, int r, int ql, int qid) {
    // 操作一中，固定左端点，寻找右端点可能的最大值
    // 这里没有考虑和右端点的比较，直接寻找了最大的可能值
    pushdown(o);
    if (r < ql || tree[o].id1 == qid || tree[o].id1 == 0) {
        // 全部符合条件
        return ALLOK;
    } else if (tree[o].id2 != -1) {
        // 不符合条件，返回该区域左边第一个
        return l - 1;
    } else {
        // 需要寻找确切位置
        // 先查找左区间，如果左区间全满足则再寻找右区间
        int mid = (l + r) >> 1;

```

```

        int leftPart = (ql <= mid) ? find_right(ls, l, mid, ql, qid) : ALLOK;
        return (leftPart == ALLOK) ? find_right(rs, mid + 1, r, ql, qid)
                                   : leftPart;
    }
}

#undef ALLOK

void modify_val(int o, int l, int r, int ql, int qr, int val, int id,
               bool ignoreLid = false) {
    // 若 val = INF 代表不需要对 val 进行处理
    // 若 ignoreLid = true 则不对 lid 进行更改
    if (l >= ql && r <= qr) {
        if (val != INF)
            tree[o].val = tree[o].lazy_val = val;
        tree[o].id1 = tree[o].lazy_id1 = id;
        tree[o].id2 = tree[o].lazy_id2 = id;
        if (!ignoreLid)
            tree[o].lid = tree[o].lazy_lid = id;
        return;
    }
    pushdown(o);
    int mid = (l + r) >> 1;
    if (ql <= mid) {
        modify_val(ls, l, mid, ql, qr, val, id, ignoreLid);
    }
    if (qr > mid) {
        modify_val(rs, mid + 1, r, ql, qr, val, id, ignoreLid);
    }
    pushup(o);
}

bool is_same_id(int o, int l, int r, int ql, int qr, int id,
               bool isRecover = false) {
    // 判断该区域 id 和 lid 是否满足条件
    if (l >= ql && r <= qr) {
        if (isRecover) {
            // 检查 id = 0 且 lid = id
            return (tree[o].id2 == 0 && tree[o].lid == id);
        } else {
            // 检查 id = id
            return (tree[o].id2 == id);
        }
    }
    pushdown(o);
    int mid = (l + r) >> 1;
    bool isSame = true;
    if (ql <= mid) {
        isSame = isSame && is_same_id(ls, l, mid, ql, qr, id, isRecover);
    }
}

```

```

    if (qr > mid && isSame) {
        isSame = isSame && is_same_id(rs, mid + 1, r, ql, qr, id, isRecover);
    }
    return isSame;
}

int query_val(int o, int l, int r, int p) {
    // 线段树单点求值: val
    if (p >= l && p <= r && tree[o].val != INF) {
        return tree[o].val;
    }
    pushdown(o);
    int mid = (l + r) >> 1;
    if (p <= mid)
        return query_val(ls, l, mid, p);
    else
        return query_val(rs, mid + 1, r, p);
}

int query_id(int o, int l, int r, int p) {
    // 线段树单点求值: id2
    if (p >= l && p <= r && tree[o].id2 != -1) {
        return tree[o].id2;
    }
    pushdown(o);
    int mid = (l + r) >> 1;
    if (p <= mid)
        return query_id(ls, l, mid, p);
    else
        return query_id(rs, mid + 1, r, p);
}

#undef ls
#undef rs

struct instruction {
    int opt, id, l, r, x;
} inst[maxn];
// numList 存储所有可能出现的数, totnum 表示个数
int numList[maxn << 2], totnum;
void discretization() {
    // 离散化操作
    sort(numList + 1, numList + 1 + totnum);
    totnum = unique(numList + 1, numList + 1 + totnum) - numList - 1;
    m = totnum;
    for (int i = 1; i <= k; ++i) {
        if (inst[i].opt == 0 || inst[i].opt == 1 || inst[i].opt == 2) {
            inst[i].l =
                lower_bound(numList + 1, numList + 1 + totnum, inst[i].l) -

```

```

        numList;
    inst[i].r =
        lower_bound(numList + 1, numList + 1 + totnum, inst[i].r) -
        numList;
} else {
    inst[i].x =
        lower_bound(numList + 1, numList + 1 + totnum, inst[i].x) -
        numList;
}
}
}

int main() {
    scanf("%d%d%d", &n, &m, &k);
    numList[++totnum] = 1;
    numList[++totnum] = m;
    for (int i = 1; i <= k; ++i) {
        scanf("%d", &inst[i].opt);
        if (inst[i].opt == 0) {
            scanf("%d%d%d", &inst[i].id, &inst[i].l, &inst[i].r, &inst[i].x);
            numList[++totnum] = inst[i].l;
            numList[++totnum] = inst[i].r;
            // 注意边界问题，为了方便这里把交界处两点分开了，下同
            if (inst[i].l != 1)
                numList[++totnum] = inst[i].l - 1;
            if (inst[i].r != m)
                numList[++totnum] = inst[i].r + 1;
        } else if (inst[i].opt == 1) {
            scanf("%d%d", &inst[i].id, &inst[i].l, &inst[i].r);
            numList[++totnum] = inst[i].l;
            numList[++totnum] = inst[i].r;
            if (inst[i].l != 1)
                numList[++totnum] = inst[i].l - 1;
            if (inst[i].r != m)
                numList[++totnum] = inst[i].r + 1;
        } else if (inst[i].opt == 2) {
            scanf("%d%d", &inst[i].id, &inst[i].l, &inst[i].r);
            numList[++totnum] = inst[i].l;
            numList[++totnum] = inst[i].r;
            if (inst[i].l != 1)
                numList[++totnum] = inst[i].l - 1;
            if (inst[i].r != m)
                numList[++totnum] = inst[i].r + 1;
        } else {
            scanf("%d", &inst[i].x);
            // 对于查询的数，不需要进行离散化，查找第一个比它大的数即可
        }
    }
}

```

```

// 离散化处理
discretization();

// 线段树初始化建树
build(1, 1, m);

// 进行操作
for (int i = 1; i <= k; ++i) {
    if (inst[i].opt == 0) {
        // 写入操作：先求得范围，再进行填充
        int r = find_right(1, 1, m, inst[i].l, inst[i].id);
        if (r == -2)
            // r = -2 代表全部满足
            r = inst[i].r;
        else
            r = min(r, inst[i].r);
        if (inst[i].l <= r) {
            printf("%d\n", numList[r]); // 注意返回离散化前的值
            modify_val(1, 1, m, inst[i].l, r, inst[i].x, inst[i].id);
        } else {
            printf("-1\n");
        }
    }
    else if (inst[i].opt == 1) {
        // 删除操作：先判断是否可行，之后执行
        if (is_same_id(1, 1, m, inst[i].l, inst[i].r, inst[i].id)) {
            printf("OK\n");
            modify_val(1, 1, m, inst[i].l, inst[i].r, INF, 0, true);
        } else {
            printf("FAIL\n");
        }
    }
    else if (inst[i].opt == 2) {
        // 恢复操作：先判断是否可行，之后执行
        if (is_same_id(1, 1, m, inst[i].l, inst[i].r, inst[i].id, true)) {
            printf("OK\n");
            modify_val(1, 1, m, inst[i].l, inst[i].r, INF, inst[i].id,
                        true);
        } else {
            printf("FAIL\n");
        }
    }
    else if (inst[i].opt == 3) {
        // 读取操作：分别读取 id 和 val
        int id = query_id(1, 1, m, inst[i].x);
        int val = query_val(1, 1, m, inst[i].x);
        if (id == 0) {
            printf("0_0\n");
        } else {
            printf("%d_%d\n", id, val);
        }
    }
}

```

```

    }
    return 0;
}

```

2.5.3 100% 数据——动态开点线段树

2.5.3.1 思路

在上一题的做法中，我们需要先读入所有的数据并进行离散化处理，之后再执行主要的算法过程。但不是所有的题目都可以在执行主要的算法过程前得到所有的输入数据。

定义 2.4 (离线算法)

要求在执行算法前输入数据已知的算法称为离线算法。一般而言，如果没有对输入输出做特殊处理，则可以用离线算法解决该问题。

定义 2.5 (在线算法)

不需要输入数据已知就可以执行的算法称为在线算法。一般而言，如果对输入输出做特殊处理（如本次的询问需要与上次执行的答案进行异或才能得到真正的询问），则只能用离线算法解决该问题。

对于一道能用离线和在线算法解决的题目，如果出题人对数据进行了加密处理，导致只能使用在线算法，则我们称这道题是**强制在线**的。

离散化需要事先知道所有可能出现的数，所以是**离线算法**。如果要强制在线，就需要另一种思路。

同样，从询问涉及的点有限出发，我们考虑最多能涉及线段树上点的个数。线段树的高度为 $O(\log m)$ ，假设每个涉及查询的点都到达了线段树的叶子结点，且不考虑根到任意两个结点之间重复的节点，则总共涉及的线段树节点数的个数为 $O(k \log m)$ 。所以我们只需要为用到的节点开辟空间即可。

针对一般的线段树，我们是预先建好了整棵线段树（build 函数），每个线段树节点的左右子节点编号与其本身编号都是对应的（通常一个子节点是父结点的二倍，而另一个子节点则相差 1）。而对于这种只为需要用到节点开辟空间的线段树，其左右子树只有在需要的时候才会被创建，所以编号间没有特定关系，需要记录。

考虑什么时候需要开辟新结点：在初始化的时候需要开创一个根节点；在进行修改及查询的时候，如果区间不是所要的区间，则需要开创新的节点。有一个技巧是，在修改和查询的时候往往要下传标记（pushdown），可以在此之前检查是否需要开创节点。

2.5.3.2 C++ 实现

```

#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <queue>
#include <vector>
using namespace std;
const int maxn = 200010;
const int INF = 1e9 + 10;
int n, m, k;
struct treenode {
    // 左右子节点编号

```

```

int lc, rc;
// 当前节点的值, 若不唯一则为 INF; lazy 为 INF 表示无延迟更新
int val, lazy_val;
// 当前占用 id, 若存在除 0 以外两种 id 则为 -1; lazy 为 -1 表示无延迟更新
int id1, lazy_id1;
// 当前占用 id, 若存在两种 id 则为 -1; lazy 为 -1 表示无延迟更新
int id2, lazy_id2;
// 上次占用 id, 若存在两种 id 则为 -1; lazy 为 -1 表示无延迟更新
int lid, lazy_lid;
} tree[maxn << 5];
int cnt; // 线段树节点个数
#define ls tree[o].lc
#define rs tree[o].rc
int insert_node() {
    // 向线段树中插入一个节点
    ++cnt;
    tree[cnt].lc = tree[cnt].rc = 0;
    tree[cnt].val = 0;
    tree[cnt].id1 = tree[cnt].id2 = 0;
    tree[cnt].lid = 0;
    tree[cnt].lazy_val = INF;
    tree[cnt].lazy_id1 = tree[cnt].lazy_id2 = -1;
    tree[cnt].lid = -1;
    return cnt;
}

void pushup(int o) {
    // 线段树上传操作, 合并左右子树结果
    // val 的合并
    tree[o].val = (tree[ls].val == tree[rs].val) ? tree[ls].val : INF;
    // id1 的合并
    if (tree[ls].id1 == -1 || tree[rs].id1 == -1) {
        tree[o].id1 = -1;
    } else if (tree[ls].id1 == tree[rs].id1) {
        tree[o].id1 = tree[ls].id1;
    } else if (tree[ls].id1 == 0) {
        tree[o].id1 = tree[rs].id1;
    } else if (tree[rs].id1 == 0) {
        tree[o].id1 = tree[ls].id1;
    } else {
        tree[o].id1 = -1;
    }
    // id2 的合并
    if (tree[ls].id2 == -1 || tree[rs].id2 == -1) {
        tree[o].id2 = -1;
    } else if (tree[ls].id2 == tree[rs].id2) {
        tree[o].id2 = tree[ls].id2;
    } else {
        tree[o].id2 = -1;
    }
}

```

```

}
// lid 的合并
if (tree[ls].lid == -1 || tree[rs].lid == -1) {
    tree[o].lid = -1;
} else if (tree[ls].lid == tree[rs].lid) {
    tree[o].lid = tree[ls].lid;
} else {
    tree[o].lid = -1;
}
}

void pushdown(int o) {
    // 线段树标记下传操作
    // 如果对应点未被创建, 则进行创建
    if (!ls)
        ls = insert_node();
    if (!rs)
        rs = insert_node();
    if (tree[o].lazy_val != INF) {
        tree[ls].val = tree[rs].val = tree[o].lazy_val;
        tree[ls].lazy_val = tree[rs].lazy_val = tree[o].lazy_val;
        tree[o].lazy_val = INF;
    }
    if (tree[o].lazy_id1 != -1) {
        tree[ls].id1 = tree[rs].id1 = tree[o].lazy_id1;
        tree[ls].lazy_id1 = tree[rs].lazy_id1 = tree[o].lazy_id1;
        tree[o].lazy_id1 = -1;
    }
    if (tree[o].lazy_id2 != -1) {
        tree[ls].id2 = tree[rs].id2 = tree[o].lazy_id2;
        tree[ls].lazy_id2 = tree[rs].lazy_id2 = tree[o].lazy_id2;
        tree[o].lazy_id2 = -1;
    }
    if (tree[o].lazy_lid != -1) {
        tree[ls].lid = tree[rs].lid = tree[o].lazy_lid;
        tree[ls].lazy_lid = tree[rs].lazy_lid = tree[o].lazy_lid;
        tree[o].lazy_lid = -1;
    }
}

#define ALLOK -2
int find_right(int o, int l, int r, int ql, int qid) {
    // 操作一中, 固定左端点, 寻找右端点可能的最大值
    // 这里没有考虑和右端点的比较, 直接寻找了最大的可能值
    pushdown(o);
    if (r < ql || tree[o].id1 == qid || tree[o].id1 == 0) {
        // 全部符合条件
        return ALLOK;
    } else if (tree[o].id2 != -1) {

```



```

        // 不符合条件, 返回该区域左边第一个
        return l - 1;
    } else {
        // 需要寻找确切位置
        // 先查找左区间, 如果左区间全满足则再寻找右区间
        int mid = (l + r) >> 1;
        int leftPart = (ql <= mid) ? find_right(ls, l, mid, ql, qid) : ALLOK;
        return (leftPart == ALLOK) ? find_right(rs, mid + 1, r, ql, qid)
            : leftPart;
    }
}

#undef ALLOK

void modify_val(int o, int l, int r, int ql, int qr, int val, int id,
               bool ignoreLid = false) {
    // 若 val = INF 代表不需要对 val 进行处理
    // 若 ignoreLid = true 则不对 lid 进行更改
    if (l >= ql && r <= qr) {
        if (val != INF)
            tree[o].val = tree[o].lazy_val = val;
        tree[o].id1 = tree[o].lazy_id1 = id;
        tree[o].id2 = tree[o].lazy_id2 = id;
        if (!ignoreLid)
            tree[o].lid = tree[o].lazy_lid = id;
        return;
    }
    pushdown(o);
    int mid = (l + r) >> 1;
    if (ql <= mid) {
        modify_val(ls, l, mid, ql, qr, val, id, ignoreLid);
    }
    if (qr > mid) {
        modify_val(rs, mid + 1, r, ql, qr, val, id, ignoreLid);
    }
    pushup(o);
}

bool is_same_id(int o, int l, int r, int ql, int qr, int id,
               bool isRecover = false) {
    // 判断该区域 id 和 lid 是否满足条件
    if (l >= ql && r <= qr) {
        if (isRecover) {
            // 检查 id = 0 且 lid = id
            return (tree[o].id2 == 0 && tree[o].lid == id);
        } else {
            // 检查 id = id
            return (tree[o].id2 == id);
        }
    }
}

```

```

pushdown(o);
int mid = (l + r) >> 1;
bool isSame = true;
if (ql <= mid) {
    isSame = isSame && is_same_id(ls, l, mid, ql, qr, id, isRecover);
}
if (qr > mid && isSame) {
    isSame = isSame && is_same_id(rs, mid + 1, r, ql, qr, id, isRecover);
}
return isSame;
}

int query_val(int o, int l, int r, int p) {
    // 线段树单点求值: val
    if (p >= l && p <= r && tree[o].val != INF) {
        return tree[o].val;
    }
    pushdown(o);
    int mid = (l + r) >> 1;
    if (p <= mid)
        return query_val(ls, l, mid, p);
    else
        return query_val(rs, mid + 1, r, p);
}

int query_id(int o, int l, int r, int p) {
    // 线段树单点求值: id2
    if (p >= l && p <= r && tree[o].id2 != -1) {
        return tree[o].id2;
    }
    pushdown(o);
    int mid = (l + r) >> 1;
    if (p <= mid)
        return query_id(ls, l, mid, p);
    else
        return query_id(rs, mid + 1, r, p);
}

#undef ls
#undef rs

int main() {
    scanf("%d%d%d", &n, &m, &k);
    // 创建根节点
    insert_node();
    // 进行操作
    int r_opt, r_id, r_l, r_r, r_x, r_p;
    while (k--) {
        scanf("%d", &r_opt);

```

```

if (r_opt == 0) {
    // 写入
    scanf("%d%d%d", &r_id, &r_l, &r_r, &r_x);
    int r = find_right(1, 1, m, r_l, r_id);
    if (r == -2)
        r = r_r;
    else
        r = min(r, r_r);
    if (r_l <= r) {
        printf("%d\n", r);
        modify_val(1, 1, m, r_l, r, r_x, r_id);
    } else {
        printf("-1\n");
    }
} else if (r_opt == 1) {
    // 删除
    scanf("%d%d%d", &r_id, &r_l, &r_r);
    if (is_same_id(1, 1, m, r_l, r_r, r_id)) {
        printf("OK\n");
        modify_val(1, 1, m, r_l, r_r, INF, 0, true);
    } else {
        printf("FAIL\n");
    }
} else if (r_opt == 2) {
    // 恢复
    scanf("%d%d%d", &r_id, &r_l, &r_r);
    if (is_same_id(1, 1, m, r_l, r_r, r_id, true)) {
        printf("OK\n");
        modify_val(1, 1, m, r_l, r_r, INF, r_id, true);
    } else {
        printf("FAIL\n");
    }
} else {
    // 查询
    scanf("%d", &r_p);
    int id = query_id(1, 1, m, r_p);
    int val = query_val(1, 1, m, r_p);
    if (id == 0) {
        printf("0_0\n");
    } else {
        printf("%d_%d\n", id, val);
    }
}
}
return 0;
}

```

2.6 202112-5 极差路径

2.6.1 $n \leq 5000$ 数据——枚举两点

2.6.1.1 思路

先考虑用最朴素的思路来解决问题：枚举两个端点，然后计算两个端点简单路径上的所有路径。枚举两个端点的复杂度为 $O(n^2)$ ，计算路径 $O(n)$ ，总复杂度 $O(n^3)$ ，无法通过任何测试点。

思考可以节省的时间：在计算路径的时候，我们都是从两点中一点出发开始搜索，直到找到另一个节点为止。由于是一棵树上的搜索，我们从出发点开始，搜索到的每一个节点都是一条简单路径。我们只需要从一个节点开始，遍历整棵树，就可以得到结果了，复杂度 $O(n^2)$ 。

注意到可能会有重复：若 (u, v) 是合法点对，那么 (v, u) 也会被加入。我们可以先特判 $u = v$ 的情况，对于其他情况，每种合法点对都重复出现了一次，只需要将答案减半即可。

2.6.1.2 C++ 实现

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <vector>
using namespace std;
#define ll long long
#define il inline
const int maxn = 500010;
int n;
int k1, k2;
vector<int> G[maxn];
ll ans = 0;
void dfs(int s, int u, int fa, int minval, int maxval) {
    /*
        s: 出发点
        u: 目前遍历到的点
        fa: u 从 fa 递归而来
        minval: (s,u) 路径上的最小值
        maxval: (s,u) 路径上的最大值
    */
    for (int i = 0; i < G[u].size(); ++i) {
        // 遍历每一个能达到的节点
        int v = G[u][i];
        if (v == fa)
            // 防止递归时 u -> v -> u 情况发生
            continue;
        if (min(s, v) - k1 <= min(minval, v) &&
            max(s, v) + k2 >= max(maxval, v)) {
            // 满足条件，答案增加
            ++ans;
        }
    }
}
```

```
    }
    // 更新 min,max, 继续递归
    dfs(s, v, u, min(minval, v), max(maxval, v));
}
}

int main() {
    scanf("%d%d%d", &n, &k1, &k2);
    int ru, rv;
    for (int i = 1; i < n; ++i) {
        scanf("%d%d", &ru, &rv);
        G[ru].push_back(rv);
        G[rv].push_back(ru);
    }

    for (int i = 1; i <= n; ++i) {
        // 遍历每一个起始点, 不包括 x=y 型
        dfs(i, i, -1, i, i);
    }
    ans /= 2;

    // 加上 x=y 型
    ans += n;

    printf("%lld\n", ans);
    return 0;
}
```

第 3 章 第 23 次认证（2021 年 09 月）

3.1 题目及设计知识点

题目编号	题目名称	知识点
202109-1	数组推导	模拟
202109-2	非零段划分	模拟，数学
202109-3	脉冲神经网络	模拟
202109-4	收集卡牌	状压 dp
202109-5	箱根山岳险天下	树链剖分，动态树

3.2 202109-1 数组推导

3.2.1 100% 数据——模拟

3.2.1.1 思路

B 数组是 A 数组的前缀最大值，所以 B 必定是单调不降的。

考虑 B 数组中相邻元素之间的关系，推测 A 数组中的元素值范围：

- $B_1 = A_1$;
- 若 $B_i \neq B_{i-1} (i \geq 2)$ ，说明前缀最大值在第 i 个位置发生了改变；考虑 A_i ：若 $A_i \leq B_{i-1}$ 则 B_i 不会发生变化，若 $A_i > B_{i-1}$ 且 $A_i \neq B_i$ 则 $B_i = A_i$ 矛盾，所以此时 $B_i = A_i$;
- 若 $B_i = B_{i-1} (i \geq 2)$ ，说明前缀最大值在 i 处没有改变，结合上面的分析我们可以得到 $A_i \in [0, B_{i-1}]$ 。

综合以上分析，只有在 $B_i = B_{i-1} (i \geq 2)$ 时， A_i 的值是不确定的。要求最大最小值，只要令不确定的 $A_i = 0$ 或 $A_i = B_{i-1}$ 即可。

3.3 202109-2 非零段划分

3.3.1 70% 数据——模拟

3.3.1.1 思路

题目中说明 $A_i \in [0, 10000]$ 。当 $p > 10000$ 后， A 数组则成为全 0 数组，所以我们只需考虑 $k \in [1, 10000]$ 时的非零段个数，不妨设 $m = 10000$ 。

我们可以针对每一个 p ，计算出 A 数组的情况，进而计算非零段的个数，不断更新答案。在维护 A 数组的时候，我们可以让 p 递增更新。这样在 p 更新的时候，只需要将 $A_i = p - 1$ 处更新为 0 即可。

时间复杂度 $O(nm)$ 。

3.3.2 100% 数据——避免不必要更新

3.3.2.1 思路

在上个做法中，时间主要浪费在了对 A 数组的更新与重复计算非零段上。对于每一个非零 A_i ，随着 p 逐渐增大，其最多改变一次，即变为 0。而在上面的方法中，我们忽略了这个条件，每次都对所有元素进行检查，无论其是否为 0。

考虑每个 A_i 变为 0 时对非零段个数的贡献。为了简化后续讨论，这里给出一个推论：

推论 3.1 (相邻相等元素与单个元素等价)

对于一段值相同的区间，可以把它们看做成其中任意的一个元素。

这一点很好理解：既然值相同，那么这一段在变为 0 时必然是同时改变。

通过以上推论，我们先缩小 A 数组的元素个数，直到任意相邻两个元素都不同，之后考虑每个元素对整体的贡献。只看元素本身看不出什么，一种思路是查看与之相邻的两个元素。

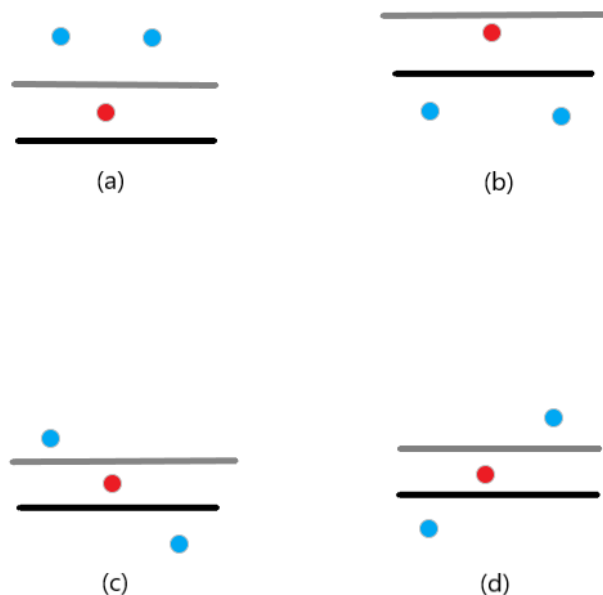


图 3.1: 相邻元素的四种情况

如图，红色点代表将要变为 0 的元素，蓝色点为相邻两个元素；黑线以下为目前变为 0 的元素。我们对以上四种情况进行分类讨论：

1. 当左右相邻元素均大于中间元素时：当中间元素变为 0 时，原本一个非零段分成了两个非零段，对非零段个数贡献 +1；
2. 当左右相邻元素均小于中间元素时：当中间元素变为 0 之前，左右元素均已变成 0，中间元素是孤立的非零段，在中间元素变为 0 后，非零段个数减少，对非零段个数贡献 -1；
3. 其他两种情况：相当于某个非零段的边界去掉了一个元素，对非零段个数无影响。

针对边界元素而言，可以将其缺失的相邻元素视为 0。

同时，考虑到我们只要求解非零段的个数，并不需要输出对应 A 数组的状态，我们可以将每个元素的贡献（当然只有在对应 $p = A_i + 1$ 时才会有贡献）累加，成为每个 p 对应的贡献。这样我们就可以先求出初始状态的非零段个数，之后随着 p 增加利用之前求出的贡献进行更新，就可以比较快速地解决。

求出每个元素的贡献、并累加到对应 p 的复杂度为 $O(n)$ ，计算每一个 p 的最后贡献的复杂度为 $O(m)$ ，整体复杂度 $O(n + m)$ 。

3.3.2.2 C++ 实现

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstring>
#include <iostream>
using namespace std;
const int maxn = 500010;
const int maxm = 10010;
int n;
int a[maxn];
int sum[maxm];
// sum[i] 表示 p = i 时的贡献
// 注意当 a[i] 发生变化时，p = a[i] + 1
int main() {
    scanf("%d", &n);
    for (int i = 1; i <= n; ++i) {
        scanf("%d", &a[i]);
    }

    // 去掉数组中相同元素段
    int tot = 0;
    a[++tot] = a[1];
    for (int i = 2; i <= n; ++i) {
        if (a[i] == a[i - 1])
            continue;
        a[++tot] = a[i];
    }
    n = tot;

    // 对非两侧元素进行处理
    for (int i = 2; i < n; ++i) {
        if (a[i] < a[i - 1] && a[i] < a[i + 1]) {
```

```
        // 中间小两侧大，贡献 +1
        ++sum[a[i] + 1];
    }
    if (a[i] > a[i - 1] && a[i] > a[i + 1]) {
        // 中间大两侧小，贡献 -1
        --sum[a[i] + 1];
    }
}

// 处理两侧元素
if (n > 1 && a[1] > a[2])
    --sum[a[1] + 1];
if (n > 1 && a[n] > a[n - 1])
    --sum[a[n] + 1];

// 计算初始情况 p = 1 时的非零段数 cur
int cur = 0;
for (int i = 1; i <= n; ++i) {
    if (a[i] == 0 && a[i - 1] != 0)
        ++cur;
}
if (a[n])
    ++cur;

// 计算并得到最大的非零段数
int ans = cur;
for (int i = 2; i <= 10001; ++i) {
    cur += sum[i];
    if (cur > ans) {
        ans = cur;
    }
}

printf("%d", ans);
return 0;
}
```

3.4 202109-3 脉冲神经网络

3.5 202109-4 收集卡牌

3.6 202109-5 箱根山岳险天下

第 4 章 第 22 次认证（2021 年 04 月）

4.1 题目及设计知识点

题目编号	题目名称	知识点
202104-1	灰度直方图	
202104-2	邻域均值	
202104-3	DHCP 服务器	
202104-4	校门外的树	
202104-5	疫苗运输	

4.2 202104-1 灰度直方图

4.3 202104-2 邻域均值

4.4 202104-3 DHCP 服务器

4.5 202104-4 校门外的树

4.6 202104-5 疫苗运输

第 5 章 第 21 次认证（2020 年 12 月）

5.1 题目及设计知识点

题目编号	题目名称	知识点
202012-1	期末预测之安全指数	
202012-2	期末预测之最佳阈值	
202012-3	带配额的文件系统	
202012-4	食材运输	
202012-5	星际旅行	

5.2 202012-1 期末预测之安全指数

5.3 202012-2 期末预测之最佳阈值

5.4 202012-3 带配额的文件系统

5.5 202012-4 食材运输

5.6 202012-5 星际旅行