A. A+B Problem

题意

有八个独立的数位显示器，每个显示器的每个二极管被点亮的概率为 p_i，管与管之间互相独立，显示器之间也相互独立，求分别显示出两个四位合法数字，且数字之和等于输入的常数 C 的概率。

关键词：

状态压缩、枚举、概率论、逆元

题解

首先考虑，显示器之间相互独立，所以我们其实可以分别求出显示器显示出 $0-9$ 每个数字的概率。

假如我们有了这个概率，我们就能求出四个显示器显示出某个四位数的概率，这个概率显然是这四个显示器分别显示其数位的概率乘积。（因为是独立事件）

例如如果显示器显示 "1" 的概率是 $\frac{1}{3}$，则显然四个显示器显示出 "1111" 的概率就是四个 $\frac{1}{3}$ 乘起来，即 $\frac{1}{81}$。

因此我们就可以枚举 A $(0 \leqq A \leqq C)$，计算出 $B = C - A$。

则对于当前枚举的 A，就可以计算出显示出 A, B 这两个四位数字的概率，那么对于当前的枚举，这两组显示器显示出 C 的概率就等于：$P(\text{显示出 } A) \times P(\text{显示出 } B)$。

那么接下来就只需要解决：如何方便快捷地求出一个显示器显示出 $0-9$ 每个数字的概率。

这里就需要用到 "状态压缩" 的技巧，我们把每个数位在显示器中要被点亮的部分当成二进制的 1，不能被点亮的部分当成二进制的 0。

例如数字 "0"，其中第 $1, 2, 3, 5, 6, 7$ 号灯管需要被点亮，而第 4 号灯管不能被点亮，则 "0" 可以状态压缩为：1110111。

这个东西显然可以提前手写出来，因此我们可以准备一张表，把 $0-9$ 每个数字的二进制状态存起来。

```
const int N = 7;
int inv100 = ksm(100LL, MOD - 2, MOD);
int S[N];
void init() {
    S[0] = (1 << 0) | (1 << 1) | (1 << 2) | (1 << 4) | (1 << 5) | (1 << 6);
    S[1] = (1 << 2) | (1 << 5);
    S[2] = (1 << 0) | (1 << 2) | (1 << 3) | (1 << 4) | (1 << 6);
    S[3] = (1 << 0) | (1 << 2) | (1 << 3) | (1 << 5) | (1 << 6);
    S[4] = (1 << 1) | (1 << 2) | (1 << 3) | (1 << 5);
    S[5] = (1 << 0) | (1 << 1) | (1 << 3) | (1 << 5) | (1 << 6);
    S[6] = (1 << 0) | (1 << 1) | (1 << 3) | (1 << 4) | (1 << 5) | (1 << 6);
    S[7] = (1 << 0) | (1 << 2) | (1 << 5);
    S[8] = (1 << 0) | (1 << 1) | (1 << 2) | (1 << 3) | (1 << 4) | (1 << 5) |
(1 << 6);
    S[9] = (1 << 0) | (1 << 1) | (1 << 2) | (1 << 3) | (1 << 5) | (1 << 6);
}
```

有了 S 表，我们就可以快速求出 $digit_i$ $(0 \leqq i \leqq 9)$，表示每个数位被表示出的概率。

接着按上述的，我们枚举 A，分别计算表示出 A 和 $C-A$ 的概率，相乘加入答案即可。

需要注意的是，整个过程中我们都需要取模，除法也需要用乘逆元代替。

```cpp
void solve() {
    int C;
    cin >> C;
    vector<int> p(N);
    for(int i = 0; i < N; i++) {
        cin >> p[i];
        p[i] = (1LL * p[i] * inv100) % MOD;
    }
    vector<int> digit(10, 1); // 一个显示器表示出 i (0-9) 的概率
    for(int i = 0; i < 10; i++) {
        for(int j = 0; j < N; j++) {
            if(S[i] >> j & 1) {
                digit[i] *= p[j];
            } else {
                digit[i] *= (1 + MOD - p[j]);
            }
            digit[i] %= MOD;
        }
    }

    auto calc = [&](int x) -> int { // 求四个显示器表示出四位数 x 的概率
        if(x == 0) {
            return digit[0] * digit[0] % MOD * digit[0] % MOD * digit[0] % MOD;
        }
        int ans = 1;
        int len = 0;
        while(x > 0) {
            ans *= digit[x % 10];
            ans %= MOD;
            x /= 10;
            len++;
        }
        for(int i = 0; i < 4 - len; i++) {
            ans = (ans * digit[0]) % MOD;
        }

        return ans;
    };

    int ans = 0;
    for(int A = 0; A <= C; A++) {
        int B = C - A;
        ans += 1LL * calc(A) * calc(B) % MOD;
        ans %= MOD;
    }

    cout << ans << endl;
}
```

时间复杂度：单组询问 $O(C)$，预处理可以认为是 $O(1)$，可以忽略。

B. Card Game

题意：

一个长度为 $2 \times n$ 的排列被恰好分成了两行 n 个数字（a, b 两数组），现在不停进行比较操作：

每次比较 a_1 和 b_1，将大的数字删除，同一行其余数字前移。如果是 a_1 被删除则得一分。

直到 a 或 b 被删空停止。

游戏过程如上，现在你可以在游戏开始前任意重排 a 以得到最大的得分，问有多少种重排的方式可以使得得分取到可能的最大值。

关键词：

贪心、思维、数学

题解

首先我们要求出最大得分是多少，也就是尽量删除尽可能多的 a，那么 a 靠前的数字就要大一些，因此我们不难想到最优情况实际上就是把 a 降序排列。

此时我们模拟一遍就会发现，对于 b 中的最小值 min_b，所有 a 中大于 min_b 的数字一定都会被删除（不管是因为和 min_b 比较，还是和别的数比较），而反之小于 min_b 的数字是不可能被删除的。

因此最大的得分实际上就是 a 中大于 min_b 的数字个数。

此时我们考虑如何求方案数，也就是说还有哪些情况也能使得答案取到上述的个数。

那么再次模拟一遍，我们会发现实际上只需要把 a 中大于 min_b 的所有数字都放在开头的部分，而剩下的部分放在后面，即将 a 分成两段，第一段全是大于 min_b 的数，第二段全是小于的。

我们发现此时必然也能取到答案，而如果不是这样的情况，则必然取不到，因为会把 min_b 删掉，导致后面的 a 更难被删除。

因此前一段随意排列，后一段随意排列，答案就是两段长度的阶乘之积。（乘法原理）

```
void solve() {
    int n;
    cin >> n;
    vector<int> a(n + 1);
    for(int i = 1; i <= n; i++) {
        cin >> a[i];
    }
    int mn = 2e9;
    for(int i = 1, x; i <= n; i++) {
        cin >> x;
        mn = min(mn, x);
    }
    int x = 0, y = 0;
    for(int i = 1; i <= n; i++) {
        if(a[i] > mn) {
            x++;
        } else {
            y++;
        }
```

```
    }

    int ans = 1;
    for(int i = 1; i <= x; i++) {
        ans = (ans * i) % MOD;
    }
    for(int i = 1; i <= y; i++) {
        ans = (ans * i) % MOD;
    }

    cout << ans << endl;
}
```

时间复杂度：$O(n)$。

C. Array Covering

题意：

给定一个序列，可以任意次操作，每次选择一个区间，把区间中（不包含端点）的数字都变为 $\max(a_l, a_r)$，问序列所有数字的总和最大值可以达到多少。

关键词：

贪心、脑筋急转弯

题解：

假设序列中某个最大值下标为 idx（有多个最大值的情况下，随便选一个即可），则：

我们永远可以通过最多两次操作，选中 $[1, idx]$ 和 $[idx, n]$ 把数组中，除了第一个位置，和最后一个位置以外的所有数字都变成数组的最大值。

因此答案就是：最大值 $\times(n-2)$，再加上 a_1 和 a_n 即可。

```
void solve() {
    int n;
    cin >> n;
    int mx = 0;
    vector<int> a(n + 1);
    for(int i = 1; i <= n; i++) {
        cin >> a[i];
        mx = max(mx, a[i]);
    }
    cout << mx * (n - 2) + a[1] + a[n] << endl;
}
```

时间复杂度：$O(n)$。

D. Sequence Coloring

题意：

给定一个序列，其中有白色数组也有黑色数字，再给定参数 k 表示一开始可以选择 k 个白数字染红，接下来每秒都会发生：所有红色数字都会把其右侧 a_i 个数字里的白色数字染红。

求：以最优策略染红最初的 k 个白色数字的话，所有白色数字都会被染红的最短时间。

（注意：黑色数字不会、也无需被染红。）

关键词：

二分、贪心

题解：

不难注意到答案具有单调性，即如果存在某种初始染色方式，能使得经过 x 秒可以染红所有白色数字，那么经过 $x+1$ 秒，局面一定不会发生改变，也就是说依然合法。

因此一定存在最小的 x 满足，x 秒可以染红所有白色数字，而 $x-1$ 秒不能全部染红。我们二分找这个 x 即可。

则问题转化为：模拟 x 秒，判断是否存在某个最优的染色方式，使得初始染色的点不超过 k 个的情况下，可以把整个序列的白色数字们染红。

则我们来实现 $check(mid)$ 函数，显然贪心地，我们从 1 开始，从左到右一个个染色。

具体的实现上，我们可以维护：ans, tim 分别表示目前已经主动染红了 ans 个，且当前最靠右的一个红色目前染了 tim 秒。

则我们肯定要贪心地使得 ans 尽可能小，最终只要 $ans \leqq k$ 就说明合法。

每次我们用当前的红色数字染红其能覆盖到的最远位置即可，但这里需要注意的是，有可能更靠左的红色其能覆盖的区间更大，例如 $\{10, 1\}$ 这样的，10 可以覆盖到更远的位置，因此对于这样的情况，我们完全可以维护出每个位置能跳到的最远位置后，我们再对这个值做一遍前缀 max，即：$nxt_i = \max(nxt_{i-1}, a_i + i)$。

这样一来，我们每次直接从 i 跳到 nxt_i，就是能到的最远位置。

每次跳到 nxt，都会花费 1 秒，当 tim 加到 mid，我们就需要新开一个手动染红的，这时需要 $ans := ans + 1$，同时我们还要找下一个要主动染红的位置，这里不能直接取 $i+1$，因为其有可能是黑色的，因此要写个 $while$ 避开所有黑色位置。（这里注意，一开始也不一定从 $i = 1$ 开始，而也要 $while$ 避开黑色位置。）

这样是最贪心的，只要最终手动染红的位置个数不超过 k，就说明可行。

需要注意的是：我们这样的写法不能处理 $mid = 0$ 时的情况，即如果答案为 0 我们需要一开始就特判掉。

以及最终答案显然不会超过 n 秒，如果 n 秒都不行，则说明无解输出 -1。

```
void solve() {
    int n, k;
    cin >> n >> k;
    vector<int> a(n + 1);
    vector<int> nxt(n + 1);
    int cnt = 0;
    for(int i = 1; i <= n; i++) {
```

```
        cin >> a[i];
        if(a[i] > 0) {
            cnt++;
        }
        nxt[i] = max(nxt[i - 1], a[i] + i);
    }
    if(cnt <= k) {
        cout << 0 << endl;
        return ;
    }

    auto check = [&](int mid) -> bool {
        int i = 1;
        while(i <= n && a[i] == 0) {
            i++;
        }
        int ans = 1;
        int tim = 0;
        while(i <= n) {
            tim++;
            i = nxt[i];
            if(i >= n) break;
            if(i == nxt[i]) {
                while(i <= n && i == nxt[i]) {
                    i++;
                }
                if(i <= n) {
                    tim = 0;
                    ans++;
                }
            } else {
                if(tim == mid) {
                    i++;
                    while(i <= n && a[i] == 0) {
                        i++;
                    }
                    if(i <= n) {
                        tim = 0;
                        ans++;
                    }
                }
            }
        }
        return ans <= k;
    };

    int l = 1, r = n;
    while(l < r) {
        int mid = (l + r) / 2;
        if(check(mid)) r = mid;
        else l = mid + 1;
    }
    if(!check(l)) {
        l = -1;
    }

```

```
    cout << l << endl;
}
```

时间复杂度：$O(n \times log(n))$。

E. Block Game

题意：

有 n 个小方块排成一排，其中第 i 个小方块上的数字是 a_i，另外还有个万能方块上面数字是 k。可以任意次把万能方块从方块序列的最左侧插入，其余方块后移，同时最后一个方块变成新的万能方块。

最大化：第一个方块上的数字 + 万能方块上的数字。

关键词：

枚举、贪心

题解：

实际上不用管万能不万能的，手玩一下就会发现，这实际上等价于一个长度为 $n+1$ 的序列可以不停循环右移，即 k 实际上就是 a_{n+1}。

而由于是循环的序列，因此是首尾相接的，也就是说 a_1, k 是相邻的关系，因此题目所求其实就是任意两个相邻数字的和，我们枚举一遍找最大的即可。

因此我们直接把 k 插在 $n+1$ 的位置，枚举找相邻数字和的最大值即可。

需要注意的是，为了方便，下面的代码数组下标是 $[0, n]$，而非 $[1, n+1]$。

```
void solve() {
    int n, k;
    cin >> n >> k;
    n++;
    vector<int> a(n, k);
    for(int i = 0; i < n - 1; i++) {
        cin >> a[i];
    }

    int ans = -2e9;
    for(int i = 0; i < n; i++) { // 枚举找相邻数字和的最大值
        ans = max(ans, a[i] + a[(i + 1) % n]);
    }

    cout << ans << endl;
}
```

时间复杂度：$O(n)$。

F. Permutation Counting

题意：

给定 m 条信息 $[l_i, r_i, k_i]$，表示一个排列在 $[l_i, r_i]$ 中的最大值为 k_i，求长度为 n 且符合所有信息的排列总个数。

关键词：

组合数学、链式并查集

题解：

首先我们在值域上考虑这件事情，如果说题目声称有若干个区间的最大值都是 k，则我们会发现：k 必然要存在于这些区间的交集，以及这些区间的并集必然只能填不超过 k，即 $\leqq k$ 的数字。

因此我们首先对输入 l, r, k 的所有信息的 k，都维护出上述的区间交集和并集。此时就可以特判，如果交集为空则说明排列不存在。（例如一定不存在一个排列满足他 $[1, 3]$ 区间的最大值是 5，且 $[4, 5]$ 区间的最大值也为 5。）

处理好上述信息后，我们**称对于当前最大值 k，其对应的交集区间为 $[\mathrm{L1}, \mathrm{R1}]$，而并集区间为 $[\mathrm{L2}, \mathrm{R2}]$**。

则不难发现由于并集区间填入的都是 $\leqq k$ 的数，因此启发我们从小到大枚举值域处理问题。

此时我们会发现，事实上有些数字并未在输入当中提及，因此我们需要一次性求若干个数字的填入方案数，因此我们将这类数字跳过（$R_2 = -1$ 的情况。）

接着我们会发现，我们的目的实际上是对于当前枚举的值 k，在 $[L1_k, R1_k]$（交集）中找一个位置，将 k 填入。而可用位置的个数并不一定是 $R1_k - L1_k + 1$，因为这个区间中有一些位置可能在之前更小的 k 的部分已经被填入数字，从而被占用掉了。因此这里我们可以借助区间和的思想，给已经占用的位置打上标记当做 0，没有占用的位置当做 1，那么可用位置实际上相当于 $[L1_k, R1_k]$ 的区间和，因此问题实际上变成了一个区间和的查询。

当我们查好交集的区间和，则答案乘上这个区间和，就确定好了 k 这个最大值的填入方案数。接着我们考虑并集，也就是小于等于 k 的其他数字。

对于并集 $[L2_k, R2_k]$，不难发现其实是类似的操作，只不过此时的区间变成了 $[L2_k, R2_k]$ 的区间和，我们仍然可以对这个区间查询一下区间和，得到的数值也就是我们本次填数可以用于的空位个数，**记作** pos，那么我们具体需要填哪些数呢？这里就需要我们在全局维护一个变量 cnt 表示还没填的数字个数。

实际上我们发现，如果上一个枚举的值再 $+1$ 的值为 lst，则我们本次填数就需要将 $[lst, k]$ 的所有数字填好，因此我们先给 cnt 加上 $k - lst$（这里不是 $k - lst + 1$ 是因为 k 这一个数字已经在上述的 $[L1_k, R1_k]$ 被填好了）。

因此问题实际上变成了，我们现在要从 cnt 个未填入数字中选择 pos 个位置填入，且顺序重要，不难发现这就是个经典的组合数学问题中的排列数 $A(cnt, pos)$。

因此我们给答案乘上这个排列数，同时将整个并集区间 $[L2_k, R2_k]$ 置为全 0，这样一来更后面的枚举就不会填入已经填好的这些位置。

填好后我们记得维护好 cnt，将它减去 pos（因为填好了 pos 个数字），同时更新 $lst = k + 1$，以执行下一个 k 值的枚举。

分析上述过程我们会发现：

1. 首先我们肯定要预处理组合数；

2. 更重要的是，我们似乎需要一个：能求区间和、能执行区间赋值，的数据结构。

这点很容易启发我们想到线段树，但我们会发现 n, m 的数据范围来到了 2×10^6，以线段树的大常数显然行不通。

我们再次分析上述维护过程，会发现实际上线段树是多余的，因为上述的过程中有一件很重要的事情，即每个位置最多只会被修改一次，换句话说整个过程中，只会发生：可用位置变成不可用位置，而不可用位置不可能变回可用位置。

因此我们将线段树换成序列上维护并查集即可，即链式并查集。

和传统并查集的不同实际上只有：我们在并查集的 $merge$ 中，保证往编号更大的节点合并。

这样一来，我们每次所谓的区间查询，实际上就变成了暴力遍历这个区间，但并非直接暴力，而是从 cur 跳到 $find(cur)$，每次合并也是合并 $(cur, cur+1)$。（cur 指当前位置，初始时为区间左端点。）

而区间中还剩余的空位个数 pos，实际上就是我们执行合并的次数，在跳区间的时候暴力统计一下即可。

上述并查集在实现按秩合并的情况下，复杂度是接近 $O(n)$ 的，同时常数十分优秀，且实测不进行按秩合并，只实现路径压缩的并查集仍然可以通过。

唯一需要注意的一点是，题目中输入的 k 并不一定包含了排列的最大值 n，因此最后枚举完所有 k 后，还得检查 cnt 是否有剩余，有剩余说明还剩 cnt 个数字还没填，因此我们还需要给答案乘上 cnt 的阶乘。

别忘记过程中不停地取模，以及组合数阶乘需要 $O(n)$ 预处理。

```
const int N = 2e6 + 10;
const int M = 2e6;
int fac[N], inv[N];
void init() {
    fac[0] = inv[0] = 1;
    for(int i = 1; i <= 2e6 + 1; i++) {
        fac[i] = 1LL * fac[i - 1] * i % MOD;
    }
    inv[M + 1] = ksm(fac[M + 1], MOD - 2, MOD);
    for(int i = M; i >= 1; i--) {
        inv[i] = 1LL * inv[i + 1] * (i + 1) % MOD;
    }
}

int A(int a, int b) {
    if(a < b) return 0;
    return 1LL * fac[a] * inv[a - b] % MOD;
}



void solve() {
    int n, m;
    cin >> n >> m;

    vector<int> fa(n + 2);
```

```cpp
    iota(fa.begin(), fa.end(), 0LL);
    auto find = [&](int x) -> int {
        while (x != fa[x]) x = fa[x] = fa[fa[x]];
        return x;
    };
    auto merge = [&](int a, int b) -> void {
        a = find(a);
        b = find(b);
        if(a > b) {
            swap(a, b);
        }
        fa[a] = b;
    };

    i64 ans = 1;
    vector<int> L1(n + 1), R1(n + 1, n + 1), L2(n + 1, n + 1), R2(n + 1,
-1);
    for(int i = 1, l, r, k; i <= m; i++) {
        cin >> l >> r >> k;
        if(R1[k] < l || L1[k] > r) {
            ans = 0;
        } else {
            L1[k] = max(L1[k], l);
            R1[k] = min(R1[k], r);
            L2[k] = min(L2[k], l);
            R2[k] = max(R2[k], r);
        }
    }

    int lst = 1;
    int cnt = 0;
    for(int k = 1; k <= n; k++) {
        if(R2[k] == -1) continue;
        int L = L1[k], R = R1[k];
        int cur = L;
        int mx_pos = 0;
        while(cur <= R) {
            int nxt = find(cur) + 1;
            if(cur + 1 == nxt) {
                mx_pos++;
                merge(cur, nxt);
            }
            cur = find(cur);
        }
        ans = (ans * mx_pos) % MOD;

        L = L2[k], R = R2[k];
        int pos = mx_pos - 1; // 除去k本身的可用位置的个数
        cur = L;
        while(cur <= R) {
            int nxt = find(cur) + 1;
            if(cur + 1 == nxt) {
                pos++;
                merge(cur, nxt); // merge(i, i+1)
            }
            cur = find(cur);
```

```
        }
        cnt += k - lst; // 需要填的数字个数

        ans = (ans * A(cnt, pos)) % MOD; // 从 cnt 个数字里选出 pos 个，随意打乱填入pos个位置
        lst = k + 1;
        cnt -= pos;
    }

    cnt += n - lst + 1;
    ans = (ans * fac[cnt]) % MOD;

    cout << ans << endl;
}
```

最终时间复杂度为：$O(n + m \times log(n))$ 小常数。

G. Digital Folding

题意：

多测给定区间 $[l, r], (1 \leqq l \leqq r \leqq 10^{15})$，定义 $f(x)$ 为把数字 x 的十进制翻转后去除前导 0 的值。

求区间中所有数字 i $(l \leqq i \leqq r)$ 的 $f(i)$ 最大值。

关键词：

分类讨论、枚举、贪心、（数位DP）

题解

首先，数位DP 可做，但不需要数位 DP。

方便处理起见，我们用 $string$ 存数字，而不是 $long\ long$。

实际上我们分类讨论即可，注意到非常多的情况下，我们都能让答案的开头是 9，具体的，我们可以做如下讨论：

1. $\mathrm{R} = 10000...00$
2. 其它

对于第一种情况，比较的特殊，因为这时答案一定不会和 R 的位数相同，具体来说如果 $L = R$，则答案只能取 1，反之答案必然等于 $9999...999$（长度恰好是 R 的长度减一），这是显然的，因为答案的位数一定是 R 的位数减一，而这个位数里最大的数字就是全 9，这个数字是取得到的。

第二种情况，这样的情况下，我们会发现答案总是可以取到和 R 相同的位数，因为哪怕此时 $R = 1000...0001$ 这样的数字，答案也能取到 $f(R) = R$，**长度为 $length(R)$ 的数字总是大于长度为 $length(R) - 1$ 的数字的。**

因此长度小于 $length(R)$ 的数字我们直接不考虑了，也就是说，我们可以直接把 L 变成和 R 相同位数的最小数字，且末尾不为 0，也就是：$1000...0001$。

此时 L, R 的位数已经相同，处理起来会方便很多很多。

此时就是一个经典的按位贪心了，我们直接从高位到低位找到第一个 L, R 不相同的位 k，把 k 后面的位全取 9 即可。因为这时 $R_k \neq L_k$，我们只需要将 R_k 减去 1，就可以把 $R[k+1...]$ 后面的位全变成 9。当然也有可能并不需要，或许 R 从第 k 位开始后面就全是 9，这时 R_k 甚至都不需要减去 1。

当然还有一种情况是，没有找到 L, R 不同的位，此时也就是说 $L = R$，则答案显然直接取 $f(L)$ 即可。

```cpp
void solve() {
    string L, R;
    cin >> L >> R;
    int l = stoll(L), r = stoll(R);
    int n = R.size();
    string t(1, '1');
    for(int i = 0; i < n - 1; i++) {
        t += '0';
    }
    if(R == t) { // R = 1000...
        if(L == R) {
            cout << 1 << endl;
        } else {
            int x = stoll(R);
            cout << x - 1 << endl;
        }
        return ;
    }

    if(L.size() < R.size()) { // 使得 L = 1000...0001，和 R 位数相同，方便处理
        L = t;
        L.back() += 1;
        assert(L.size() == R.size());
    }

    string ans;
    int k = -1;
    for(int i = 0; i < n; i++) {
        if(L[i] != R[i]) {
            k = i;
            break;
        }
    }
    if(k == -1) {
        ans = L;
        while(ans.size() > 1 && ans.back() == '0') {
            ans.pop_back();
        }
        reverse(ans.begin(), ans.end());
    } else {
        bool ok = 1;
        for(int i = k + 1; i < n; i++) {
            ans += '9';
            ok &= (R[i] == '9');
        }
        ans += (R[k] - !ok);
        for(int i = k - 1; i >= 0; i--) {
            ans += L[i];
```

```
        }
    }

    cout << ans << endl;
}
```

时间复杂度：单组 $O(log(R))$。

H. Blackboard

题意：

给定一个序列 a，表示运算式：$a_1 + a_2 + a_3 \ldots + a_n$，问有多少种把 $+$ 替换成 $|$（位运算按位或）的方式，使得不改变运算式的值。（不替换也是一种方案，且特别的：本题中认为 $|$ 运算优先级大于 $+$ 。）

关键词：

位 or 运算、前缀和优化DP、计数

题解：

很经典的线性 DP 模型，注意到 $[1, n-1]$ 这个子数组的答案不受 a_n 的影响，因此我们可以从前往后 DP。

我们定义 f_i 表示考虑 $[1, i]$ 这个子数组中，替换的合法方案数。

则转移我们向前找所有 j，满足：$[j+1, i]$ 这一段数字和等于或，对于这样的 j，我们就可以执行转移：

$$f_i := f_i + f_j$$

根据位运算的性质，显然形如这样的 j 是位于以 i 结尾的一段**连续区间**中的，即必然存在一个最靠左的 L，满足 $[L, i]$ 这一段区间的所有下标都可以作为上述的 j，而 $[1, L-1]$ 的所有下标都不能作为。

因此我们想办法找出这个 L 即可，这里我们可以根据 or 运算的性质，如果要 $sum = or$，则必须满足二进制没有交集（即 $\&$ 运算结果为 0），因此我们向前找出二进制没有交集的最远位置即可，这一步显然我们只需要跳 log 次（不跳 0），我们记录一个 pre_i 表示 i 前面离得最近的非 0 数字。

我们 $i = pre_i$ 这样跳最多 $log(a_i)$ 次必然能找到这样的位置。

则转移我们就把 $[L, i]$ 这一整段区间的 f_j 都加给 f_i 即可。

因此我们实际上执行的是求区间和的操作，因此我们再用一个 s 数组维护 f 的前缀和，就可以 $O(1)$ 求区间和转移了。

```
void solve() {
    int n;
    cin >> n;
    vector<int> a(n + 1);
    int lst = 0;
    vector<int> pre(n + 1);
    for(int i = 1; i <= n; i++) {
        cin >> a[i];
```

```
        pre[i] = lst;
        if(a[i] > 0) {
            lst = i;
        }
    }

    vector<int> dp(n + 2);
    vector<int> s(n + 2);
    dp[1] = 1;
    s[1] = 1;
    for(int i = 1; i <= n; i++) {
        int j = i;
        int val = 0;
        while(j > 0 && (val & a[j]) == 0) {
            val |= a[j];
            j = pre[j];
        }
        dp[i + 1] = (s[i] - s[j] + MOD) % MOD;
        s[i + 1] = (s[i] + dp[i + 1]) % MOD;
    }

    cout << dp[n + 1] << endl;
}
```

这样转移，时间复杂度为：$O(n \times log(A))$。

当然，上述找 L 的操作也可以用双指针维护，这样可以做到 $O(n)$ 的优秀复杂度。

或者可以借助 ST 表等数据结构，配合二分查找去找 L，也可以做到 $O(n \times log(n))$ 之类的复杂度。

I. AND vs MEX

题意：

多测给定区间 $[l, r]$，可以任意次操作从区间中选若干个数字，把 and 加入集合 S，求 S 的 MEX 最大值。

$T \leqq 10^5$, $0 \leqq l \leqq r < 2^{30}$。

关键词：

贪心、位运算、构造

题解：

我们做一些简单的观察，不难发现，由于数字是 AND 出来的，因此能 AND 出来的最大数字也不过就是 R，因此答案必然不超过 $\mathrm{R}+1$，也就是说这就是答案的上界。

情况 0：首先特判 $\mathrm{L}=0$，输出 $\mathrm{R}+1$，显然正确。

情况 1：此时 $\mathrm{L}>0$，我们考虑一个简单情况：L, R 的最高位 1 相同，此时我们发现，区间中所有数字也会和 L, R 一样最高位保持相同，则无论如何选择，AND 出来的结果也一定存在这一个 1。因此答案必然为 0。

上述的简单情况，启发我们从 L,R 的最高位的情况来出发考虑，为方便讨论，我们定义 $highbit(x)$ 为 x 的最高位 1（类似树状数组的 $lowbit$。）

我们令 $b_1 = highbit(L), b_2 = highbit(R)$，那么上一段的情况实际上就是 $b_1 = b_2$。

情况 2：接着我们考虑 $b_2 \geq b_1 + 2$ 的情况，即 R 的最高位比 L 的最高位多至少两位。

为了便于理解，我们以具体的例子来看。$\mathrm{L} = 0010, \mathrm{R} = 1001$。

此时我们会发现，区间中必然存在 0011 这个数字，而同时必然存在 $[0100, 0111]$ 这段区间中的所有数字。

此时我们使用这段区间中的数字全部、分别 AND 上 0011，我们会发现，实际上我们会得到 $[0000, 0011]$ 的所有数字。（其本质就是去掉了上段区间中所有数字的最高位 1。）

而 $[0000, 0011]$ 实际上包含了 $[0, \mathrm{L}]$，因此在这种情况下我们不知不觉间已经拿到了 $[0, \mathrm{R}]$ 的所有数字，因此答案必然为 $\mathrm{R} + 1$。

情况 3：此时就只剩下最后一种情况，也即 $\mathrm{b}_1 + 1 = \mathrm{b}_2$：$\mathrm{R}$ 的最高位 1 恰好是 L 最高位 1 的两倍。

我们仍然举例子：$\mathrm{L} = 0101, \mathrm{R} = 1011$。

此时，其实我们可以类比上述的情况 2，发现区间中必然存在：0111 这个数字，以及 $[1000, \mathrm{R}]$ 的所有数字，此时我们用后者的所有数字 AND 上前者这个 0111，我们会发现，我们可以得到：$[0, \mathrm{R} - 2^{\mathrm{b}_2}]$ 的所有数字。

即我们可以得到 0 到 R 去掉最高位 1 的所有数字。

因此我们的答案**至少可以达到** $\mathrm{R} - 2^{\mathrm{b}_2} + 1$，而这时我们发现，如果说这个值还大于等于 L，相当于 $[0, \mathrm{R} - 2^{\mathrm{b}_2}]$ 和 $[\mathrm{L}, \mathrm{R}]$ 产生了交集，因此答案又可以取到上界 $\mathrm{R} + 1$ 了。

上一句的分析正确，但并不完全正确，同时这也是本题最细节的部分。

事实上，我们输入得到的区间 $[\mathrm{L}, \mathrm{R}]$ 是可以天然的向下扩展的，也就是说 L 自己是可以构造出更小的数字的，同样的我们举一个具体的例子：$\mathrm{L} = 1101100$。

那么我们会发现，区间中必然存在：1101111 这个数字，以及 $[1110000, 1111111]$ 这段区间的数字。而同样的用后者区间中的所有数字全部、分别和 1101111 做 AND 总能至少得到 $[1100000, \mathrm{L}]$ 的数字，因此这一段区间又存在了和 $[\mathrm{L}, \mathrm{R}]$ 的交集，因此我们的 L 实际上可以自行构造出：L 去掉二进制中第一个有效 0 右侧的所有位。也即：L 本身能构造出的下限是 L 除去从 b_1 开始连续的一段 1 右侧的部分，下面给出两个例子便于理解：

对于 $\mathrm{L} = 111011111$，其下限就是 111000000；

对于 $\mathrm{L} = 10011111$，其下限就是 10000000。

因此我们要用这个自身构造的下限，和上述情况 3 中的 $\mathrm{R} - 2^{\mathrm{b}_2}$ 判断是否有交，这样才是最优解。

```
void solve() {
    auto highbit = [&](int x) -> int {
        for(int j = 32; j >= 0; j--) {
            if(x >> j & 1) {
                return j;
            }
        }
        return -1;
```

```
    };

    int L, R;
    cin >> L >> R;
    int b1 = highbit(L), b2 = highbit(R);
    if(b1 == -1) {
        cout << R + 1 << endl;
    } else if(b1 == b2) {
        cout << 0 << endl;
    } else if(b2 > (b1 + 1)) {
        cout << R + 1 << endl;
    } else {
        int ans = R - (1LL << b2) + 1;
        int l = 0; // L 能自己构造的下限
        for(int j = b1; j >= 0; j--) {
            if((L >> j & 1) == 0) {
                break;
            } else {
                l |= (1LL << j);
            }
        }

        if(l <= ans) {
            ans = R + 1;
        }
        cout << ans << endl;
    }
}
```

时间复杂度：单组 $O(log)$。

J. MST Problem

题意：

n 点的无向完全图，每个点有数字 a_i，其中 u, v 之间连边的权值为 $a_u + a_v$，删除图中给定的 m 条边后，求图的最小生成树权重，无解 -1。

关键词：

完全图，最小生成树，$\mathrm{prim}, \mathrm{boruvka}$，线段树

题解

先来个开幕雷击（内测提交情况）：

9/173

做法很多，这里介绍一种有教育意义的：**数据结构优化** Prim。

实际上在完全图最小生成树中，不仅有 $boruvka$ 这种算法，很多时候也可以使用数据结构（甚至 set）优化的 Prim 算法。

前置知识：Prim 算法求最小生成树。

【这里简要介绍 Prim 的思想】：

首先我们要明白 Prim 的原理，仅维护**一个**连通块，每次从当前连通块的**所有点**向外伸出去的Prim里选择一条权值最小的（称为 min），把 min 边以及其所连接的点加入连通块，这就是我们的生成树中的一条边，接着再用新加入的这个点它向外伸出去的所有边更新下一次选择的 min。

观察这个过程，会发现实际上类似于 $dijkstra$ 求最短路的过程，每次取最小边权 min，再把它伸出去的所有边加入堆中，用堆就能保证每次取出来的是 min。因此对于常见的 n, m 均为 10^5 一般稀疏图，Prim 算法的代码和 $dijkstra$ 写起来十分相像，时间复杂度也是 $nlogn$（n, m 同阶）。

但在本题中，图变成了类完全图，即在完全图的基础上做了一些小修改。

那么上述介绍的 Prim 算法，就不能用堆来维护了，而这时我们通常需要借助一些别的数据结构，例如 set 或线段树，来维护这一过程，而在本题中就是用线段树维护。**（set 维护的题，可以参考2025CCPC网络赛C题）**

那么首先还是要明确 Prim 的过程，我们仅维护一个连通块，每次要找伸出去的所有边里的 min，这一点实际上就是线段树直接查询**全局最小**值，我们求出这一最小值，同时还要在线段树中维护出这一最小值所对应的点，我们查询出这一信息后，就直接将其边权加入 ans，同时为了避免在后续的全局最小值查询中受到这个已经加入连通块的点的影响，因此我们需要直接将这个点的信息删除掉，这里最好的办法就是直接将其边权置为正无穷，这样一来后续的全局 min 查询总是不可能查询到正无穷的。

接着我们还要考虑，将这个点加入连通块后，这个点伸出去的所有边我们还需要进行更新，以便我们之后的全局 min 是正确的，因此我们要更新这个点伸出去的所有边

我们会发现由于图是类完全图，因此这个边的数量是 $O(n)$ 级别的，因此我们就需要一次性更新若干边，而不能一个个更新，那么这实际上就是我们的**区间修改**操作。

具体来讲，我们对所有输入的删除记录建一个图（记得去重），对每个点的邻点进行排序，准备好这样一个删除数组。

接着对于每次我们全局查出来的 min，以及其对应的点 u，我们都枚举 u 的所有 "删除邻点"，把相邻两个删除邻点之间的所有点一次性修改掉**（区间修改）**。

同时别忘记删除掉 u 点本身，即将其点权置为正无穷。

实现上，由于第一步时连通块是空的，此时我们随意取一个点作为初始点即可，因此第一次加边是不需要进行全局查询的，所以实现上我们是先修改，后查询（当然这里查出来的信息就是下一次要用的信息）。

按上述过程维护 $n-1$ 次，就得到了最终的生成树，因为每次必然加入了一条新边。

不过记得特判无解的情况，无解即在枚举的过程中，查询到的全局 min 等于正无穷，则说明此图必然不连通，也就不存在生成树了。

```
const int N = 3e5 + 10;
i64 inf = 1e18;
int n, m, a[N];

struct Node {
    int l, r;
```

```
    i64 lz; // lz : 区间中所有点，能取到的连通块里的点的最小点权
    pair<i64, int> vertix, edge;
}tr[N * 4]; // 对每个点 i 维护目前最小的邻接边权(edge.first)，以及邻接边权对应的下一个点(edge.second)

void push_up(int u) {
    tr[u].edge = min(tr[u * 2].edge, tr[u * 2 + 1].edge);
    tr[u].vertix = min(tr[u * 2].vertix, tr[u * 2 + 1].vertix);
}

void build(int u, int l, int r) {
    tr[u] = {l, r, inf, {a[l], l}, {a[l] + inf, l}};
    if(l == r) {
        return ;
    }
    int mid = (l + r) / 2;
    build(u * 2, l, mid);
    build(u * 2 + 1, mid + 1, r);
    push_up(u);
}

void push_down(int u) {
    for(auto son : {u * 2, u * 2 + 1}) {
        if(tr[son].lz > tr[u].lz) {
            tr[son].lz = tr[u].lz;
            if(tr[son].vertix.first + tr[u].lz <= tr[son].edge.first) {
                tr[son].edge = {tr[son].vertix.first + tr[u].lz, tr[son].vertix.second};
            }
        }
    }
}

void update(int u, int l, int r, int v) { // 从编号位于[l,r]的点到目前连通块的可选边，多了一个边权为a[x]+v的，x为区间内的点
    if(tr[u].l >= l && tr[u].r <= r) {
        if(tr[u].lz <= v) return ;
        // cerr << "u : " << u << ", l : " << l << ", r : " << r << endl;
        tr[u].lz = v;
        if(tr[u].vertix.first + v < tr[u].edge.first) {
            tr[u].edge = {tr[u].vertix.first + v, tr[u].vertix.second};
        }
        return ;
    }
    push_down(u);
    int mid = (tr[u].l + tr[u].r) / 2;
    if(l <= mid) {
        update(u * 2, l, r, v);
    }
    if(r > mid) {
        update(u * 2 + 1, l, r, v);
    }
    push_up(u);
}

void del(int u, int k) { // 把已经加入连通块的点删掉
```

```
    if(tr[u].l == tr[u].r) {
        tr[u].vertix = {inf, k};
        tr[u].edge = {inf + a[k], k};
        return ;
    }
    push_down(u);
    int mid = (tr[u].l + tr[u].r) / 2;
    if(k <= mid) {
        del(u * 2, k);
    } else {
        del(u * 2 + 1, k);
    }
    push_up(u);
}

void solve() {
    cin >> n >> m;
    int idx = -1, mn = 2e9;
    for(int i = 1; i <= n; i++) {
        cin >> a[i];
        if(a[i] < mn) {
            idx = i;
            mn = a[i];
        }
    }

    vector<set<int>> fbd(n + 1);
    for(int i = 0, u, v; i < m; i++) {
        cin >> u >> v;
        fbd[u].emplace(v);
        fbd[v].emplace(u);
    }
    for(int i = 1; i <= n; i++) {
        fbd[i].emplace(i);
    }

    vector<vector<int>> bad(n + 1);
    for(int i = 1; i <= n; i++) {
        for(auto & u : fbd[i]) {
            bad[i].emplace_back(u);
        }
    }

    build(1, 1, n);
    i64 ans = 0;
    for(int i = 0; i < n - 1; i++) {
        del(1, idx); // 去掉当前连通块里最小的点，表示这个点已经加入了当前连通块

        int l = 0;
        for(auto r : bad[idx]) {
            if(l + 1 <= r - 1) {
                update(1, l + 1, r - 1, a[idx]);
            }
            l = r;
        }
        if(l + 1 <= n) {
```

```
            update(1, l + 1, n, a[idx]);
        }

        ans += tr[1].edge.first; // 取全局最小值的边权
        idx = tr[1].edge.second; // 最小边对应的点
        if(ans >= inf) {
            break;
        }
    }

    if(ans >= inf) {
        ans = -1;
    }
    cout << ans << endl;
}
```

时间复杂度：$O(n \times log(n))$。（n, m 同阶）

Fun Fact：

1. 内测中，有 ≥ 2 位验题人在本题中整整交了两页 $submissions$。
2. 由于本场寒假营出题时间较早，本题大约在2025的暑假期间就已经出好了，因此出题人在九月份的**2025CCPC网络赛**现场看到同款类似题的 C 题，就很快做出了。（所以想必是个蛮有教育意义的题）

K. Constructive

题意：

构造字典序最小的，长度为 n 的数组满足：

1. 所有数字均为正整数
2. 所有数字互不相同
3. 所有数字的和等于所有数字的乘积

关键词：

签到、构造

题解：

注意到 $n = 1, n = 3$ 有解，其余均无解。

很好想到的是，当 $n = 4$ 时，满足所有数均为正整数且数字互不相同的字典序最小数组为 $\{1, 2, 3, 4\}$，而这个数组的乘积已经远远大于和了，当 $n > 4$ 时，乘积和总和的差距只会越来越大，因此不可能有解。

而 $n = 2$ 手玩就会发现和 $n > 4$ 是类似的，不可能存在解。

```cpp
void solve() {
    int n;
    cin >> n;
    if(n == 1 || n == 3) {
        cout << "YES" << endl;
        for(int i = 1; i <= n; i++) {
            cout << i << " \n"[i == n];
        }
    } else {
        cout << "NO" << endl;
    }
}
```

时间复杂度：单组 $O(1)$。

L. Need Zero

题意：

给定正整数 n，要选一个正整数 x 使得 $n \times x$ 的末尾是 0，问 x 最小值。

关键词：

签到、语法题

题解：

注意到末尾是 0 相当于变成 10 的倍数，那么 $10 \times n$ 一定是合法的答案，因此答案不超过 10，直接枚举 x 从 1 到 10 即可。

当然，也可以分类讨论，想变成 10 的倍数无非就是缺 $2, 5, 10$ 中的一种。

需要注意的是，如果本身就是 10 的倍数，直接输出 1。

```cpp
void solve() {
    int n;
    cin >> n;
    if(n % 10 == 0) {
        cout << 1 << endl;
    } else if(n % 5 == 0) {
        cout << 2 << endl;
    } else if(n % 2 == 0) {
        cout << 5 << endl;
    } else {
        cout << 10 << endl;
    }
}
```

时间复杂度：$O(1)$。