5. 双指针

分为两大类:

(1)两个指针指向两个序列,如归并排序中合并的过程.

(2)两个指针指向同一序列,维护区间信息.

模板:

```
for (int i = 0, j = 0; i < n; i++) {
    while (j < i && check(i, j)) j++;
  }
```

虽有两重循环,但两指针移动的距离之和不超过n,故时间复杂度$O(n)$.朴素做法的两重循环时间复杂度$O(n^2)$.

实际应用时先写朴素做法,后用单调性优化.

5.1 双指针

5.1.1 最长连续不重复子列

题意

给定一字符串为一个英语句子,其中有若干个用1个空格分隔的单词,且句子开头无空格,句子无标点符号.输出句子中的每个单词.

思路

i指向每个单词的开头,j从i开始往后移动至每个单词后面的空格.

代码

```
int main() {
  char s[100]; fgets(s, 100, stdin);

  int len = strlen(s);
  for (int i = 0, j = 0; i < len; i = j + 1) {  // i指针每次更新到下一个单词的开头
    j = i;
    while (j < len && s[j] != ' ') j++;
    for (int k = i; k < j; k++) cout << s[k];
    cout << endl;
  }
}
```

5.1.2 最长连续不重复子列

题意

给定一长度为n $(1 \leq n \leq 1e5)$的整数序列(元素范围为$[0, 1e5]$), 找出其中最长的不包含重复数的连续区间, 输出其长度.

思路

朴素做法: i指向区间的右端点, j指向区间的左端点, 时间复杂度$O(n^2)$, 代码:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j <= i; j++) {
      if (check(i, j)) ans = max(ans, i - j + 1);
    }
  }
```

考虑优化. 同朴素做法, 考虑每个以i为右端点的区间, 左端点j最远能右移到何位置. 显然i右移时, j不左移, 即有单调性.

i指针右移, 每次检查j指针是否可右移, 这样两指针移动距离之和不超过$2n$, 故时间复杂度$O(n)$.

因最多有$1e5$个数, 则可开一个$cnt[]$数组记录当前区间内每个数的出现次数. i每次右移时加入一个数$cnt[i]$, 则$cnt[nums[i]] ++$. 若此时区间内出现重复的数字, 则重复的数字是$nums[i]$; j每次右移时删去一个数$nums[j]$, 则$cnt[nums[j]] --$.

代码

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

    int ans = 0;
    vector<int> cnt(1e5 + 1);
    for (int l = 1, r = 1; r <= n; r++) {
        cnt[a[r]]++;
        while (l < r && cnt[a[r]] > 1)
            cnt[a[l++]]--;
        ans = max(ans, r - l + 1);
    }
    cout << ans << endl;
}

int main() {
    solve();
}
```

5.1.3 数组元素的目标和

题意

给定两长度分别为n、m $(1 \le n, m \le 1e5)$的升序数组A、B和目标值x, 数组下标从0开始, 数组元素的范围为$[1, 1e9]$. 求满足$A[i] + B[j] = x$的数对(i, j), 数据保证有唯一解.

思路

朴素做法, 时间复杂度$O(nm)$:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      if (A[i] + B[j] == x) {
        cout << i << ' ' << j;
        exit(0);
      }
    }
  }
```

注意到A和B都是升序序列, 考虑对每个i, 找到最小的j使得$A[i] + B[j] > x$. 指针i从1开始, 指针j从m开始, 因和x不变, 则i右移时, j不右移, 即j的移动有单调性. i至多移动n次, j至多移动m次, 时间复杂度$O(n + m)$.

代码

```
const int MAXN = 1e5 + 5;
int A[MAXN], B[MAXN];

int main() {
  for (int i = 0; i < n; i++) cin >> A[i];
  for (int i = 0; i < m; i++) cin >> B[i];

  for (int i = 0, j = m - 1; i < n; i++) {
    while (j >= 0 && A[i] + B[j] > x) j--;
    if (j >= 0 && A[i] + B[j] == x) {
      cout << i << ' ' << j;
      exit(0);
    }
  }
}
```

5.1.4 判断子列

题意

给定长度分别为n、m $(1 \le n, m \le 1e5)$的整数序列$a[]$、$b[]$, 元素范围为$[-1e9, 1e9]$. 判断$a[]$是否为$b[]$的子列, 若是则输出"Yes", 否则输出"No". 子列是序列的一部分项按原次序排列得到的序列, 如$\{a_1, a_3, a_5\}$是序列$\{a_1, a_2, a_3, a_4, a_5\}$的子列.

思路

称$a[]$是$b[]$的子列为顺次匹配. 显然双指针可找到一种匹配. 还需说明若存在一种匹配, 可用双指针找到,这样问题才等价.

假设双指针算法已找到一种匹配A, 考虑另一种匹配B与A的第一个不同对应, 不妨设为$a[i] \to b[j]$和$a[i] \to b[k]$. 显然$b[k]$在$b[j]$之间, 即将$b[k]$换为$b[j]$不改变后续的对应关系, 则每个匹配B都可通过将不同对应换为A中的对应得到匹配A, 故每一种匹配都可用双指针找到.

代码

```cpp
void solve() {
    int n, m; cin >> n >> m;
    vector<int> a(n), b(m);
    for (auto& ai : a) cin >> ai;
    for (auto& bi : b) cin >> bi;

    for (int i = 0, j = 0; i < n && j < m; j++) {
        if (a[i] == b[j]) i++;

        if (i == n) {
            cout << "Yes" << endl;
            return;
        }
    }
    cout << "No" << endl;
}

int main() {
    solve();
}
```

5.1.5 unique()函数的实现

题意

实现STL中的unique()函数.假定所给的数组$nums[]$是升序的.

思路

不重复的元素有两种情况:①它是数组的首元素;②$nums[i] \neq nums[i-1]$.

用双指针,将所有不重复的元素放到前j个数.

代码

```cpp
vector<int>::iterator unique(vector<int>& a) {
  for (int i = 0, j = 0; i < a.size(); i++) {
    if (!i || a[i] != a[i - 1])
      a[j++] = a[i];  // 将不重复的元素放在前j个数
  }
  return a.begin() + 1;
}
```

5.1.6 调整数组顺序使得奇数在偶数前面

题意

给定一个整数数组(长度≤ 100),调整其元素的顺序使得奇数在偶数前面.

思路

用类似快排的思路:两指针i和j分别从首尾往中间移动,i移动至偶数停下,j移动至奇数停下,若两指针未相遇,则交换两指针对应的数.这样保证i的左边都是奇数,j的右边都是偶数.时间复杂度$O(n)$.

代码

```cpp
class Solution {
public:
    void reOrderArray(vector<int> &array) {
         int i = 0, j = array.size() - 1;
         while(i < j){
             while(i < j && array[i] & 1) i++;
             while(i < j && !(array[j] & 1)) j--;

             if(i < j) swap(array[i], array[j]);
         }
    }
};
```

5.1.7 King of String Comparison

题意

给定两长度为n $(1 \leq n \leq 2e5)$的字符串s和t,求使得子串$s[l \cdots r]$的字典序<子串$t[l \cdots r]$的字典序的(l, r) $(1 \leq l \leq r \leq n)$的对数.

思路

从左往右扫一遍,用双指针维护子串的左右端点.

代码 -> 2021ICPC欧洲东南部区域赛-A(双指针)

```cpp
const int MAXN = 2e5 + 5;
int n;
string s, t;

void solve() {
  cin >> n >> s >> t;
  ll ans = 0;
  int l = 0, r = 0;
  while (max(l, r) < n) {
    r = max(r, l);

    if (s[r] == t[r]) r++;
    else if (s[r] < t[r]) ans += n - r, l++;
    else l = r + 1, r = l;
  }
  cout << ans;
```

```
}

int main() {
  solve();
}
```

5.1.8 Jason ABC

题意

给定一个长度为$3n$,且只包含'A'、'B'、'C'的字符串s.现有操作:选择s一个连续子串,将其中的全部字符替换为'A'、'B'或'C'.求将s变为恰含'A'、'B'、'C'各n个所需的最小操作次数,并输出任一方案.

第一行输入一个整数n $(1 \leq n \leq 3e5)$.第二行输入一个长度为$3n$,且只包含'A'、'B'、'C'的字符串s.

第一行输出一个整数k表示最小操作次数.接下来k行每行输出两个整数l, r $(1 \leq l \leq r \leq 3n)$和一个字符$c \in \{A, B, C\}$,表示将子串$s[l \cdots r]$中的全部字符换为$c$.

思路

首先一定有解,最坏可每次改变一个字符.其次最小操作次数不超过3次,最坏可先将整个串变为'B',然后将前n个字符变为'A',再将后n个字符变为'C'.

进一步地,最小操作次数不超过2次.证明:不妨设$s[1 \cdots p]$是s最短的、恰包含n个相同字符(不妨设为'A')的前缀,设其中字符'B'、'C'的数量分别为$cntb, cntc$ $(cntb, cntc < n)$.只需将子串$s[(p+1) \cdots (p+n-cntb)]$替换为'B',将子串$s[(p+n-cntb+1), 3n]$替换为'C'即可.

显然最小操作次数为0当且仅当初始串已满足条件.下面考察何时1次操作即可满足条件.

设s中'B'、'C'的个数分别为b、c.考察是否存在一个区间$[l, r]$ $s.t.$ 子串$s[1 \cdots (l-1)] + s[(r+1) \cdots 3n]$中恰有$n$个'B'和$n$个'C',进而可将子串$s[l \cdots r]$替换为'A'即可满足条件.注意到此时子串$s[l \cdots r]$中恰有$(b-n)$个'B'和$(c-n)$个'C',可预处理出每个前缀$s[1 \cdots i]$中'B'和'C'的个数$b_i, c_i$,用双指针检查是否存在一个区间$[l, r]$ $s.t.$ $b_r - b_{l-1} = b-n, c_r - c_{l-1} = c-n$.总时间复杂度$O(n)$.

代码

```
const int MAXN = 9e5 + 5;  // 注意开3倍空间
int n;
string s;
int cnt[MAXN][3];  // cnt[i][]表示子串s[1...i]中字符'A'(0)、'B'(1)、'C'(2)出现的次数

bool check(char ch) {  // 检查是否存在一个区间[l,r] s.t. b[r]-b[l-1]=b-n,c[r]-c[l-1]=c-n
  int x = (ch - 'A' + 1) % 3, y = (ch - 'A' + 2) % 3;  // 另外两字符
  int l = 1, r = 1;
  while (l <= r) {
    while ((cnt[r][x] - cnt[l - 1][x] < cnt[n][x] - n / 3) || (cnt[r][y] - cnt[l - 1][y] < cnt[n][y] - n / 3)) r++;
    if (cnt[r][x] - cnt[l - 1][x] == cnt[n][x] - n / 3 && cnt[r][y] - cnt[l - 1][y] == cnt[n][y] - n / 3) {
      cout << 1 << endl;
      cout << l << ' ' << r << ' ' << ch;
      return true;
    }
    else l++;
  }
  return false;
```

```
}

void complete(int p, char ch) {  // 用两次操作完成
  int x = (ch - 'A' + 1) % 3, y = (ch - 'A' + 2) % 3;  // 另外两字符
  char X = x + 'A', Y = y + 'A';
  cout << 2 << endl;
  cout << p + 1 << ' ' << p + n/3 - cnt[p][x] << ' ' << X << endl;
  cout << p + n / 3 - cnt[p][x] + 1 << ' ' << n << ' ' << Y;
}

void solve() {
  cin >> n;
  n *= 3;
  cin >> s;
  s = " " + s;

  int p = -1; char ch;  // s[1...p]是s最短的、恰包含n个相同字符ch的前缀
  for (int i = 1; i <= n; i++) {
    for (int j = 0; j < 3; j++) cnt[i][j] = cnt[i - 1][j];
    cnt[i][s[i] - 'A']++;

    if (p == -1) {
      if (cnt[i][0] == n / 3) p = i, ch = 'A';
      else if (cnt[i][1] == n / 3) p = i, ch = 'B';
      else if (cnt[i][2] == n / 3) p = i, ch = 'C';
    }
  }

  if (cnt[n][0] == n / 3 && cnt[n][1] == n / 3 && cnt[n][2] == n / 3) {
    cout << 0;
    return;
  }

  if (check('A') || check('B') || check('C')) return;

  complete(p, ch);
}

int main() {
  solve();
}
```

5.1.9 Quiz Master

原题指路:https://codeforces.com/contest/1777/problem/C

题意 (2 s)

给定一个整数m和一个长度为n的序列$a_1,\cdots,a_n$,在$a[]$中选取若干个元素$a_{p_1},\cdots,a_{p_k}$,使得对$\forall T\in[1,m],\exists a_i\ s.t.\ a_{p_i}\bmod T=0$.若有解,输出选出的元素的极差的最小值;否则输出$-1$.

有t $(1\le t\le 1e4)$组测试数据.每组测试数据第一行输入两个整数n,m $(1\le n,m\le 1e5)$.第二行输入n个整数$a_1,\cdots,a_n$ $(1\le a_i\le 1e5)$.数据保证所有测试数据的n之和、m之和都不超过1e5.

思路I

显然若极差为$diff$时存在合法方案,则极差$> diff$时也存在合法方案(至少可在极差为$diff$的合法方案上加入元素),故极差有二段性,考虑二分极差.注意右边界取$\max\limits_{1\le i\le n} a_i$以判断无解的情况.

考虑如何check.对二分出的每个极差$diff$,先将$a[]$非降序排列,枚举左端点$i\in[1,n]$,则对每个i,可在$a[]$上二分出最靠右的$s.t.\ a_j-a_i\le diff$的$a_j\ \ (i\le j\le n)$,移项得$a_j\le diff+a_i$,则合法的范围在第一个$> diff+a_i$的a_j之前.

暴力检查二分出区间是否合法时间复杂度$O(n^2)$,会TLE.考虑优化,注意到指针i右移时,指针j不左移,即两指针的移动都是单调的,则可用类似于莫队中将当前区间移动到下一个询问区间的方式移动指针,同时维护记录$1\sim m$中每个数的出现次数的桶$cnt[]$.

考察每个区间内的a_i对该区间的贡献.注意到$a_{p_i}\bmod T=0$表明a_{p_i}是T的倍数,则每个a_i会对其$\le m$的约数产生贡献,预处理出$a[]$中每个元素的约数$factors[]$即可.

一个区间合法的充要条件是:该区间内$1\sim m$都出现,这可用类似于莫队维护区间元素种类的方式维护.

时间复杂度$O\left(n\log n+n\sqrt{a}+(n\log n)\cdot a^{\frac{1}{3}}\cdot\log a\right)$.

代码I

```cpp
void solve() {
  int n, m; cin >> n >> m;
  vector<int> a(n);
  for (auto& ai : a) cin >> ai;
  sort(all(a));

  vector<vector<int>> factors(n);  // factors[i]存a[i]的约数
  for (int i = 0; i < n; i++) {
    for (int j = 1; (ll)j * j <= a[i]; j++) {
      if (a[i] % j == 0) {
        factors[i].push_back(j);
        if (a[i] / j != j)
          factors[i].push_back(a[i] / j);
      }
    }
  }

  vector<int> cnt(m + 1);  // 记录1~m中每个数出现的次数
  int ans = 0;  // cnt[]中出现的元素种类

  auto insert = [&](int i)->void {
    for (auto x : factors[i]) {
      if (x <= m) {
        if (!cnt[x]) ans++;
        cnt[x]++;
      }
    }
  };

  auto remove = [&](int i)->void {
    for (auto x : factors[i]) {
      if (x <= m) {
        cnt[x]--;
        if (!cnt[x]) ans--;
      }
    }
```

```
  };

  auto check = [&](int diff)->bool {
    cnt = vector<int>(m + 1, 0);
    ans = 0;
    insert(0);

    for (int i = 0, j = 0; i < n; i++) {
      int tmpj = upper_bound(all(a), diff + a[i]) - a.begin();  // 注意是upper_bound
      while (j + 1 < n && j + 1 < tmpj) insert(++j);

      if (ans == m) return true;
      else remove(i);
    }
    return false;
  };

  int l = 0, r = a[n - 1];
  while (l < r) {
    int mid = l + r >> 1;
    if (check(mid)) r = mid;
    else l = mid + 1;
  }
  cout << (l == a[n - 1] ? -1 : l) << endl;
}

int main() {
  CaseT
    solve();
}
```

思路II

在思路I的基础上撤去二分,枚举左端点,移动右端点,遇到合法方案时更新极差的最小值即可.

时间复杂度$O\left(n\log n+n\sqrt{a}+n\cdot a^{\frac{1}{3}}\right)$.

代码II

```
void solve() {
  int n, m; cin >> n >> m;
  vector<int> a(n);
  for (auto& ai : a) cin >> ai;
  sort(all(a));

  vector<vector<int>> factors(n);  // factors[i]存a[i]的约数
  for (int i = 0; i < n; i++) {
    for (int j = 1; (ll)j * j <= a[i]; j++) {
      if (a[i] % j == 0) {
        factors[i].push_back(j);
        if (a[i] / j != j)
          factors[i].push_back(a[i] / j);
      }
    }
  }

```

```cpp
  vector<int> cnt(m + 1);  // 记录1~m中每个数出现的次数
  int kind = 0;  // cnt[]中出现的元素种类
  int ans = INF;

  auto insert = [&](int i)->void {
    for (auto x : factors[i]) {
      if (x <= m) {
        if (!cnt[x]) kind++;
        cnt[x]++;
      }
    }
  };

  auto remove = [&](int i)->void {
    for (auto x : factors[i]) {
      if (x <= m) {
        cnt[x]--;
        if (!cnt[x]) kind--;
      }
    }
  };

  for (int l = 0, r = -1; l < n; remove(l++)) {  // 枚举左端点
    while (kind != m) {
      if (r == n - 1) break;
      else insert(++r);
    }

    if (kind == m) ans = min(ans, a[r] - a[l]);
  }
  cout << (ans == INF ? -1 : ans) << endl;
}

int main() {
  CaseT
    solve();
}
```

5.1.10 Flexible String

原题指路:https://codeforces.com/contest/1778/problem/C

题意 (2 s)

给定两个长度为n的、只包含小写英文字母的字符串a和b,其中a至多包含10种不同的字符,两字符串的下标都从1开始.给定一个初始时为空的集合Q.现有操作:选择一个下标i $(1 \le i \le n)$和一个小写英文字母c,令$a_i = c$.现进行若干次操作,使得最后Q最多包含k种不同的字符,同时最大化 $s.t.\ a[l \cdots r] = b[l \cdots r]$的数对$(l, r)$ $(1 \le l \le r \le n)$的个数,输出该个数.

有t $(1 \le t \le 1e4)$组测试数据.每组测试数据第一行输入两个整数n, k $(1 \le n \le 1e5, 0 \le k \le 10)$.第二行输入一个长度为$n$的、只包含小写英文字母的字符串$a$,数据保证$a$至多包含10种不同的字符.第三行输入一个长度为$n$的、只包含小写英文字母的字符串$b$.

思路

注意到随k的增加, $s.t.\ a[l\cdots r]=b[l\cdots r]$的数对$(l,r)\ \ (1\le l\le r\le n)$的个数不减,故只需考虑$|Q|$取得最大值$\min\{k,cnt\}$的情况,其中$cnt$为$a$包含的字符种数.

注意到$|Q|=\min\{k,cnt\}\le 10$,考虑二进制枚举保留的字符,对枚举的每个二进制表示中恰包含$|Q|$个1的状态$mask$,用双指针求出所有a与b匹配的区间,则长度为len的区间对答案的贡献为$len+(len-1)+\cdots+1=\dfrac{len\cdot(len+1)}{2}$.对所有状态的答案取max即可.

时间复杂度$<O\left(n\cdot 2^{\min\{k,cnt\}}\right)$,这是因为不是所有的$2^{\min\{k,cnt\}}$个状态都需要算一遍贡献.事实上,时间复杂度$O\left(n\cdot C_{cnt}^{\min\{k,cnt\}}\right)$.

代码

```cpp
int popcount(int x) {
  x = x - ((x >> 1) & 0x55555555);
  x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
  return ((x + (x >> 4) & 0xf0f0f0f) * 0x1010101) >> 24;
}

void solve() {
  int n, k; cin >> n >> k;
  string a, b; cin >> a >> b;

  vector<int> frequency(26);
  for (auto ch : a) frequency[ch - 'a']++;

  int cnt = 0;  // a包含的字符种数
  for (int i = 0; i < 26; i++)
    frequency[i] = frequency[i] ? cnt++ : -1;
  k = min(k, cnt);

  ll ans = 0;
  for (int mask = 0; mask < (1 << cnt); mask++) {
    if (popcount(mask) != k) continue;

    ll res = 0;
    for (int i = 0, j = -1; i <= n; i++) {  // 注意i<=n
      if (i == n || (a[i] != b[i] && (~mask >> frequency[a[i] - 'a'] & 1))) {
        int len = i - j - 1;
        res += (ll)len * (len + 1) / 2;
        j = i;
      }
    }
    ans = max(ans, res);
  }
  cout << ans << endl;
}

int main() {
  CaseT
    solve();
}
```

5.1.11 Hard Process

原题指路: https://codeforces.com/problemset/problem/660/C

题意

给定一个长度为n $(1 \le n \le 3e5)$的整数序列$a = [a_1, \cdots, a_n]$ $(a_i \in \{0, 1\})$和一个整数k $(0 \le k \le n)$. 现有操作: 将$a[]$中的至多k个0变为1. 求至多进行k次操作后$a[]$中的最长连续1的个数, 输出任一方案.

思路I

显然应将尽可能多的0变为1. 固定区间左端点l, 右移区间右端点r, 维护区间$[l, r]$中0的个数cnt, 更新最长连续1的个数和当前的区间端点. 若$cnt > k$, 则右移l直至$cnt = k$. 总时间复杂度$O(n)$.

代码I

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

    int ans = 0, ansl, ansr;
    int cnt = 0;  // 0的个数
    for (int l = 1, r = 1; r <= n; r++) {
        cnt += !a[r];
        if (cnt > k) cnt -= !a[l++];

        if (r - l + 1 > ans) {
            ans = r - l + 1;
            ansl = l, ansr = r;
        }
    }

    for (int i = ansl; i <= ansr; i++) a[i] = 1;

    cout << ans << endl;
    for (int i = 1; i <= n; i++)
        cout << a[i] << " \n"[i == n];
}

int main() {
    solve();
}
```

思路II

设序列中元素0的下标依次为$pos_1, \cdots, pos_m$ $(m \ge 1)$. 枚举i $(1 \le i \le \max\{m - k, 1\})$, 则可将下标为$pos_i, \cdots, pos_{i+k-1}$的$k$个0都变为1, 此时得到的连续1的长度
$len = (pos_{i+k} - 1) - (pos_{i-1} + 1) + 1 = pos_{i+k} - pos_{i-1} - 1$, 更新答案并记录方案即可.

注意到连续1可能出现在序列首尾, 在序列首位加入0作为哨兵即可. 总时间复杂度$O(n)$.

代码II

```cpp
void solve() {
    int n, k; cin >> n >> k;
    vector<int> a(n + 5);
    vector<int> pos = { 0 };  // 0的下标, 序列首尾加入0作为哨兵
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
        if (!a[i]) pos.push_back(i);
    }
    pos.push_back(n + 1);

    int ans = 0, l, r;
    int cnt = (int)pos.size() - 1;  // 0的个数
    if (cnt <= k) {
        ans = n;
        l = 0, r = n + 1;
    }
    else {
        for (int i = 1; i + k <= cnt; i++) {
            int len = (pos[i + k] - 1) - (pos[i - 1] + 1) + 1;
            if (len > ans) {
                ans = len;
                l = pos[i - 1], r = pos[i + k];
            }
        }
    }

    for (int i = l + 1; i < r; i++) a[i] = 1;

    cout << ans << endl;
    for (int i = 1; i <= n; i++)
        cout << a[i] << " \n"[i == n];
}

int main() {
    solve();
}
```

思路III

显然最长连续1的个数越小越易达到, 则最长连续1的个数有二段性. 预处理$a[]$中1的个数的前缀和$pre[]$, 二分最长连续1的个数len, 枚举最长连续1的起点下标$i \in [1, n - len + 1]$, 检查$pre_{i+len-1} - pre_{i-1} + k \geq len$是否成立, 若成立, 记录最长连续1的起点下标$pos$. 总时间复杂度$O(n \log n)$.

代码III

```cpp
void solve() {
    int n, k; cin >> n >> k;
    vector<int> a(n + 1);
    vector<int> pre(n + 1);  // a[]中1的个数的前缀和
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
        pre[i] = pre[i - 1] + a[i];
    }
```

```

    int pos = 1;  // 最长连续1的起点
    auto check = [&](int len) {  // 检查能否有len个连续1
        for (int i = 1; i <= n - len + 1; i++) {
            if (pre[i + len - 1] - pre[i - 1] + k >= len) {
                pos = i;
                return true;
            }
        }
        return false;
    };

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

    if (n == k) pos = 1, l = n;
    for (int i = pos; i <= pos + l - 1; i++) a[i] = 1;

    cout << l << endl;
    for (int i = 1; i <= n; i++)
        cout << a[i] << " \n"[i == n];
}

int main() {
    solve();
}
```

思路IV

二分最长连续1的个数len, 用长度为len的滑动窗口check. 总时间复杂度$O(n \log n)$.

代码IV

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

    int pos = 1;  // 最长连续1的起点
    auto check = [&](int len) {  // 检查能否有len个连续1
        vector<int> cnt(2);
        for (int i = 1; i < len; i++) cnt[a[i]]++;

        for (int i = len; i <= n; i++) {
            if (i - len) cnt[a[i - len]]--;
            cnt[a[i]]++;

            if (cnt[0] <= k) {
                pos = i - len + 1;
                return true;
            }
        }
```

```
        return false;
    };

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

    if (n == k) pos = 1, l = n;
    for (int i = pos; i <= pos + l - 1; i++) a[i] = 1;

    cout << l << endl;
    for (int i = 1; i <= n; i++)
        cout << a[i] << " \n"[i == n];
}

int main() {
    solve();
}
```

思路V

设区间$[l,r]$为连续1. 对每个区间左端点l, 考察如何快速找到能与之匹配的最靠右的r. 预处理$a[]$中0的个数的前缀和$pre[]$, $last[i]$表示 $s.t.\ pre[j]=i$的最靠右的下标j. 对每个区间左端点l, 能与之匹配的最靠右的r为$last[pre[l-1]+k]$.

注意序列中0的个数$<k$时, $last[pre[l-1]+k]=0$, 对 $s.t.\ last[i]=0$的下标i, 令$last[i]=n$即可. 总时间复杂度$O(n)$.

代码V

```
void solve() {
    int n, k; cin >> n >> k;
    vector<int> a(n + 1);
    vector<int> pre(n + 1);  // a[]中0的个数的前缀和
    vector<int> last(n + k + 1);  // last[i]表示 s.t. pre[j] = i的最靠右的下标j, 注意数组大小
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
        last[pre[i] = pre[i - 1] + !a[i]] = i;
    }

    for (int i = 1; i <= n; i++)
        if (!last[i]) last[i] = n;

    int ans = 0;
    int pos = 1;  // 最长连续1的起点
    for (int i = 1; i <= n; i++) {
        int len = last[pre[i - 1] + k] - i + 1;
        if (len > ans) {
            ans = len;
            pos = i;
        }
    }

```

```
    if (n == k) pos = 1, ans = n;
    for (int i = pos; i <= pos + ans - 1; i++) a[i] = 1;

    cout << ans << endl;
    for (int i = 1; i <= n; i++)
        cout << a[i] << " \n"[i == n];
}

int main() {
    solve();
}
```

思路VI

$dp[i][j]$表示将j个0变为1的以$a[i]$结尾的最长连续1的个数, 最终答案$\max\limits_{1\le i\le n} dp[i][k]$.

按$a[i]=0/1$分类, 状态转移方程$dp[i]][j]=\begin{cases}dp[i-1][j-1]+1, a[i]=0\\dp[i-1][j]+1, a[i]=1\end{cases}$, 对$a[i]=0$的情况, 还需清空非法状态, 即令$dp[i][0]=0$.

上述DP的时间复杂度为$O(n^2)$, 会TLE. 考虑优化, 显然可以滚掉DP的第一维. 考察第二维$dp[j]$, 注意到状态转移对$dp[]$的修改只有两种: ①$dp[j]=dp[j-1]+1$, 即整体右移一位后$+1$; ②$dp[j]=dp[j]+1$, 即整体$+1$. 考虑用队列优化状态转移, 具体地, 维护一个队首为$dp[k]$、队尾为$dp[0]$的队列, 此时: ①对整体右移的操作, 弹出队首元素即可, 此时原来的$dp[j-1]$变为现在的$dp[j]$; ②对整体$+1$的操作, 用变量$delta$记录$+1$的次数即可. 注意队列中的元素值为真实值$-delta$, 对初始条件$dp[0]=\cdots=dp[k]=0$, 初始时将$-delta$入队即可. 初始时$delta=0$, 则初始队列为$(k+1)$个0.

每个元素至多进出各一次队列, 总时间复杂度$O(n)$.

代码VI

```
void solve() {
    int n, k; cin >> n >> k;

    queue<int> que;
    // 初始条件dp[0] = ... = dp[k] = 0
    while (k--) que.push(0);
    que.push(0);

    vector<int> a(n + 1);
    int ans = 0;
    int delta = 1;  // 整体 + 1的次数
    int pos = n;  // 最长连续1的右端点
    for (int i = 1; i <= n; i++, delta++) {  // 整体 + 1
        cin >> a[i];

        if (!a[i]) {
            que.pop();  // dp[j] = dp[j - 1] + 1
            que.push(-delta);  // dp[0] = 0
        }

        int real = que.front() + delta;  // 真实值 = 存储值 + delta
        if (real > ans) {
            ans = real;
            pos = i;
        }
```

```
    }

    for (int i = pos - ans + 1; i <= pos; i++) a[i] = 1;

    cout << ans << endl;
    for (int i = 1; i <= n; i++)
        cout << a[i] << " \n"[i == n];
}

int main() {
    solve();
}
```

5.1.12 Lecture Sleep

原题指路: https://codeforces.com/problemset/problem/961/B

题意

有$1 \sim n$的n $(1 \leq n \leq 1e5)$个时刻, 在i $(1 \leq i \leq n)$时刻老师会讲解a_i $(1 \leq a_i \leq 1e4)$个定理, 学生处于t_i $(t_i \in \{0, 1\})$状态. 若某时刻学生处于1状态, 他会记下此时老师所讲的所有定理; 否则无事发生. 现有操作: 将$t[]$中一段长度为k $(1 \leq k \leq n)$的区间置为1. 求恰用一次操作后学生记下的定理数的最大值.

思路I

用一个长度为k的滑动窗口暴力求出所有长度为k的区间的答案并更新答案.

时间复杂度$O(n)$.

代码I

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

    int sum = 0;
    for (int i = 1; i < k; i++) sum += a[i];
    for (int i = k; i <= n; i++) sum += a[i] * t[i];

    int ans = 0;
    for (int r = k; r <= n; r++) {
        sum += a[r] * !t[r];
        if (r - k) sum -= a[r - k] * !t[r - k];
        ans = max(ans, sum);
    }
    cout << ans << endl;
}

int main() {
    solve();
}
```

思路II

将$t_i = 1$的a_i与$t_i = 0$的a_i分开统计, 对后者求前缀和$pre[]$后枚举长度为k的区间右端点并更新答案.

时间复杂度$O(n)$.

代码II

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

    int sum = 0; // t[i] = 1的a[i]之和
    vector<int> pre(n + 1);  // t[i] = 0的a[i]的前缀和
    for (int i = 1; i <= n; i++) {
        pre[i] = pre[i - 1];
        if (!t[i]) pre[i] += a[i];
        else sum += a[i];
    }

    int res = 0;
    for (int r = k; r <= n; r++)
        res = max(res, pre[r] - pre[r - k]);
    cout << sum + res << endl;
}

int main() {
    solve();
}
```

5.1.13 Longest k-Good Segment

原题指路: https://codeforces.com/problemset/problem/616/D

题意

给定一个正整数k, 称序列的一个区间是好的, 如果它不包含超过k种元素. 给定一个长度为n $(1 \le k \le n \le 5e5)$的序列$a = [a_1, \cdots, a_n]$ $(0 \le a_i \le 1e6, 1 \le i \le n)$. 求$a[]$的任一最长的好的区间.

思路I

因$k \ge 1$, 则必有解. 为使得好的区间最长, 应使得区间中包含的元素的种数在不超过k的前提下尽量大.

为用桶记录每种元素出现的次数, 用双指针维护每个出现了k种元素(若存在)的区间, 更新答案即可.

时间复杂度$O(n)$.

代码I

```cpp
const int MAXA = 1e6 + 5;

void solve() {
    int n, m; cin >> n >> m;  // 序列长度、区间最大元素种数
    vector<int> a(n + 1);
```

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

    int ansl = 1, ansr = 1;
    vector<int> cnt(MAXA);
    for (int l = 1, r = 1, cur = 0; r <= n; r++) {  // cur表示当前区间中的元素种数
        if (++cnt[a[r]] == 1) cur++;
        while (cur > m) {
            if (--cnt[a[l]] == 0) cur--;
            l++;
        }

        if (r - l + 1 > ansr - ansl + 1) ansl = l, ansr = r;
    }
    cout << ansl << ' ' << ansr << endl;
}

int main() {
    solve();
}
```

思路II

线段树维护数组$cnt=[cnt_1,\cdots,cnt_n,cnt_{n+1}]$, 其中$cnt_i=1$表示下标$i$是序列中某元素第一次出现的位置.

枚举好的区间的左端点$l\in[1,n]$, 对每个l, 用线段树求 $s.t.$ 前缀$a[1...r']$中元素个数$>k$的最靠左的下标r', 则能与l匹配的最靠右的好的区间的右端点$r=r'-1$.

对每个好的区间的左端点l, 显然后缀$a[l...n]$中只有每个元素第一次出现的位置对区间的元素种数有贡献. $fir[i]$表示值i第一次出现的下标, 若值i未出现, 则定义$fir[i]=n+1$. $nxt[i]$表示值$a[i]$下一次出现的下标, 若值$a[i]$无下一次出现, 则定义$nxt[i]=n+1$. 这两个数组可从后往前递推求得, 即先将$fir[]$都初始化为$(n+1)$, 再枚举下标i从n到1, 令$nxt[i]=fir[a[i]], fir[a[i]]=i$.

初始时, 将序列的所有元素的第一次出现的下标的$cnt[]$值在线段树中$+1$.

区间左端点l右移时, 先令$cnt[l]=0$, 即若下标l原本是某元素第一次出现的位置, 则将其$cnt[]$值清空; 再令$cnt[nxt[l]]=1$, 即若元素$a[l]$有下一次出现的位置, 则将其$cnt[]$值置为1.

代码II

```
const int MAXA = 1e6 + 5;

struct SegmentTree {
    int n;
    struct Node {
        int sum;

        Node(int _sum = 0) :sum(_sum) {}

        friend Node operator+(const Node A, const Node B) {
            Node res;
            res.sum = A.sum + B.sum;
            return res;
        }
    };
    vector<Node> SegT;

    SegmentTree(int _n) :n(_n) {
```

```
        n++;  // 注意序列长度 + 1以区分无解的情况
        SegT.resize(n + 5 << 2);
    }

    void pushUp(int u) {
        SegT[u] = SegT[u << 1] + SegT[u << 1 | 1];
    }

    void modify(int u, int l, int r, int pos, int val) {  // cnt[pos] += val
        if (l == r) {
            SegT[u].sum += val;
            return;
        }

        int mid = l + r >> 1;
        if (pos <= mid) modify(u << 1, l, mid, pos, val);
        else modify(u << 1 | 1, mid + 1, r, pos, val);
        pushUp(u);
    }

    void modify(int pos, int val) {
        modify(1, 1, n, pos, val);
    }

    int find(int u, int l, int r, int k) {  // 求 s.t. 前缀a[1...r']中元素种数 > k的最靠左的下标r'
        if (l == r) return l;

        int mid = l + r >> 1;
        if (SegT[u << 1].sum > k) return find(u << 1, l, mid, k);
        else return find(u << 1 | 1, mid + 1, r, k - SegT[u << 1].sum);
    }

    int find(int k) {
        return find(1, 1, n, k);
    }
};

void solve() {
    int n, m; cin >> n >> m;  // 序列长度、区间最大元素种数
    vector<int> a(n + 1);
    for (int i = 1; i <= n; i++) cin >> a[i];

    vector<int> fir(MAXA, n + 1);  // fir[i]表示值i第一次出现的下标
    vector<int> nxt(n + 1);  // nxt[i]表示值a[i]下一次出现的下标
    for (int i = n; i; i--) {
        nxt[i] = fir[a[i]];
        fir[a[i]] = i;
    }

    SegmentTree st(n);
    for (int i = 0; i < MAXA; i++)  // 注意值域从0开始枚举
        if (fir[i] <= n) st.modify(fir[i], 1);  // cnt[fir[i]]++

    int ansl = 1, ansr = 1;
    for (int l = 1; l <= n; l++) {
        int r = st.find(m) - 1;  // - 1后为当前元素种数 <= k的最靠右的下标
        if (r - l + 1 > ansr - ansl + 1)
```

```
            ansl = l, ansr = r;
        if (r == n) break;

        st.modify(l, -1);  // cnt[l]--
        if (nxt[l] <= n) st.modify(nxt[l], 1);  // cnt[nxt[l]]++
    }
    cout << ansl << ' ' << ansr << endl;
}

int main() {
    solve();
}
```

5.1.14 日志统计

题意

现收集到n $(1 \le n \le 1e5)$行日志, 每行的格式为"ts id", 表示ts $(0 \le ts \le 1e5)$时刻编号id $(0 \le id \le 1e5)$的帖子收到一个赞. 给定两个整数d, k $(1 \le d \le 1e4, 1 \le k \le 1e5)$, 若某帖子在任一长度为$d$的时间区间内收到至少$k$个赞, 则称该帖子是好的. 升序输出所有好的帖子的编号.

思路

将所有日志按时刻非降序排列, 依次枚举每个日志, 双指针维护长度不超过d的时间区间, 同时维护每个帖子获赞的个数.

代码

```
void solve() {
    int n, d, k; cin >> n >> d >> k;
    vector<pair<int, int>> logs(n);
    for (auto& [ts, id] : logs) cin >> ts >> id;
    sort(all(logs));

    vector<int> cnt(1e5 + 1);
    vector<bool> good(1e5 + 1);
    for (int l = 0, r = 0; r < n; r++) {
        int id = logs[r].second;
        cnt[id]++;

        while (logs[r].first - logs[l].first >= d)  // 注意是 >= d
            cnt[logs[l++].second]--;

        if (cnt[id] >= k) good[id] = true;
    }

    for (int i = 0; i <= 1e5; i++)  // 注意i从0开始
        if (good[i]) cout << i << endl;
}

int main() {
    solve();
}
```

5.1.15 完全二叉树的权值

题意

给定一棵包含n $(1 \leq n \leq 1e5)$个节点的完全二叉树, 节点从上到下、从左到右依次编号$1 \sim n$, 节点i的权值为a_i $(-1e5 \leq a_i \leq 1e5)$. 设根节点的深度为1. 现将每个深度的节点的权值累加, 求节点权值之和最大的深度, 若有多组解, 输出最小的深度.

思路I

DFS预处理出各节点的深度后, 对每个深度累加权值.

代码I

```cpp
struct BinaryTree {
    int n;
    struct Node {
        int l, r;
        int w;
    };
    vector<Node> BTree;
    vector<int> depth;
    int maxDepth = 0;
    vector<ll> sums;

    BinaryTree(int _n, const vector<int>& a) :n(_n) {
        BTree.resize(n + 1);
        depth.resize(n + 1), sums.resize(n + 1);
        for (int i = 1; i <= n; i++)
            BTree[i] = { i << 1, i << 1 | 1, a[i] };

        dfs(1, 0);
    }

    void dfs(int u, int fa) {
        maxDepth = max(maxDepth, depth[u] = depth[fa] + 1);
        if (BTree[u].l <= n) dfs(BTree[u].l, u);
        if (BTree[u].r <= n) dfs(BTree[u].r, u);
    }

    int get() {
        for (int i = 1; i <= n; i++)
            sums[depth[i]] += BTree[i].w;

        int res = 1;
        for (int i = 2; i <= maxDepth; i++)  // 注意枚举到maxDepth而不是n
            if (sums[i] > sums[res]) res = i;
        return res;
    }
};

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

```

```
    BinaryTree bt(n, a);
    cout << bt.get() << endl;
}

int main() {
    solve();
}
```

思路II

注意完全二叉树的每个深度对应序列中的一个区间, 则无需建树, 可用双指针找到每个区间.

代码II

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

    ll maxSum = -0x3f3f3f3f3f3f3f3f;
    int ans;
    for (int depth = 1, l = 1; l <= n; l *= 2, depth++) {
        ll sum = 0;
        for (int r = l; r <= min(l + (1 << depth - 1) - 1, n); r++)
            sum += a[r];

        if (sum > maxSum) {
            maxSum = sum;
            ans = depth;
        }
    }
    cout << ans << endl;
}

int main() {
    solve();
}
```

5.1.16 Vasya and Arrays

原题指路: https://codeforces.com/problemset/problem/1036/D

题意

给定一个长度为$n\ \ (1 \leq n \leq 3e5)$的序列$a = [a_1, \cdots, a_n]\ \ (1 \leq a_i \leq 1e9)$和长度为$m\ \ (1 \leq m \leq 3e5)$的序列$b = [b_1, \cdots, b_m]$. 现有操作: 取$a[]$或$b[]$的一个区间, 将其替换为该区间的元素之和. 问若干次(可能为零)操作后能否使得序列$a[]$与序列$b[]$相同, 若不能, 则输出-1; 否则输出操作后$a[]$和$b[]$的长度的最大值.

思路I

问题等价于: 将$a[]$和$b[]$分别划分为若干个不交的区间, 使得每个区间中元素之和对应相等. 求划分区间数的最大值.

显然有解 iff $a[]$的元素之和等于$b[]$的元素之和, 最坏可将整个序列划分为一个区间.

贪心策略: 每次将$a[]$与$b[]$的元素之和相等且长度最小的前缀划分为一个区间, 删去该前缀后重复该过程, 直至将$a[]$和$b[]$的所有元素都划分到某个区间中.

[证] 设最优解中的某个划分与贪心策略的划分不同.

因贪心策略的划分每次取最短的前缀, 则最优解的划分取更长的前缀.

因$a_i, b_j > 0 \ \ (1 \le i \le n, 1 \le j \le m)$, 则最优解中的划分的元素之和更大,

进而最优解中包含一个长度更小的元素之和相等的前缀, 可将它们划分以增大区间数, 与是最优解矛盾.

实现时, 判断是否无解后, 每次用双指针取一个元素之和相等且长度最小的前缀即可.

总时间复杂度$O(n)$.

代码I

```cpp
void solve() {
    int n; cin >> n;
    vector<int> a(n);
    ll sum = 0;
    for (auto& ai : a) {
        cin >> ai;
        sum += ai;
    }
    int m; cin >> m;
    vector<int> b(m);
    for (auto& bi : b) {
        cin >> bi;
        sum -= bi;
    }

    // if (accumulate(all(a), 0) != accumulate(all(b), 0)) {  // 爆int
    if (sum) {
        cout << -1 << endl;
        return;
    }

    int ans = 0;
    int idx1 = 0, idx2 = 0;
    while (idx1 < n) {
        ans++;

        ll suma = a[idx1++], sumb = b[idx2++];
        while (suma != sumb) {
            if (suma < sumb) suma += a[idx1++];
            else sumb += b[idx2++];
        }
    }
    cout << ans << endl;
```

```
}

int main() {
    solve();
}
```

思路II

同**思路I**, 但划分区间时直接比较$a[]$和$b[]$的前缀和$preA[]$和$preB[]$的大小. 显然元素之和相等的前缀对应的$preA[]$和$preB[]$中的元素相等, 进而同时删去或加入不改变两者之差.

总时间复杂度$O(n)$.

代码II

```
void solve() {
    int n; cin >> n;
    vector<ll> preA(n + 1);
    for (int i = 1; i <= n; i++) {
        cin >> preA[i];
        preA[i] += preA[i - 1];
    }
    int m; cin >> m;
    vector<ll> preB(m + 1);
    for (int i = 1; i <= m; i++) {
        cin >> preB[i];
        preB[i] += preB[i - 1];
    }

    if (preA[n] != preB[m]) {
        cout << -1 << endl;
        return;
    }

    int ans = 0;
    int idxA = 1, idxB = 1;
    while (idxA <= n && idxB <= m) {
        if (preA[idxA] == preB[idxB]) {
            ans++;
            idxA++, idxB++;
        }
        else if (preA[idxA] > preB[idxB]) idxB++;
        else idxA++;
    }
    cout << ans << endl;
}

int main() {
    solve();
}
```

思路III

由**思路II**: 答案即两个前缀和数组$preA[]$和$preB[]$中相等元素的个数, 用map统计即可.

时间复杂度$O(n\log n)$.

代码III

```cpp
void solve() {
    map<ll, int> cnt;
    int n; cin >> n;
    ll preA = 0;
    for (int i = 0; i < n; i++) {
        int a; cin >> a;
        cnt[preA += a]++;
    }

    int ans = 0;
    int m; cin >> m;
    ll preB = 0;
    for (int i = 0; i < m; i++) {
        int b; cin >> b;
        ans += cnt[preB += b];
    }
    cout << (preA == preB ? ans : -1) << endl;
}

int main() {
    solve();
}
```

思路IV

显然$a[]$的前缀和$preA[]$单调增. 用变量$last$记录上一个区间结尾的下标. 枚举下标$j\in[1,m]$, 对每个b_j, 将其加入当前的前缀和$preB$中, 并二分出 $s.t.\ preA[r]-preA[last]=preB$的下标$r\in[last+1,n]$, 若存在这样的下标$r$则更新答案和$last$, 并清空$preB$, 否则继续枚举下一个$b_j$. 由**思路I**中贪心策略的证明知这样是最优的.

时间复杂度$O(n\log n)$.

代码IV

```cpp
void solve() {
    int n; cin >> n;
    vector<ll> preA(n + 1);
    for (int i = 1; i <= n; i++) {
        cin >> preA[i];
        preA[i] += preA[i - 1];
    }

    int m; cin >> m;
    vector<int> b(m + 1);
    ll preB = 0;
    for (int i = 1; i <= m; i++) {
        cin >> b[i];
        preB += b[i];
    }
```

```cpp

    if (preA[n] != preB) {
        cout << -1 << endl;
        return;
    }

    int ans = 0;
    preB = 0;
    int last = 0;  // 上个区间的右端点
    for (int i = 1; i <= m; i++) {
        preB += b[i];

        // 二分出 s.t. preA[r] - preA[last] = preB的下标r ∈ [last + 1, n]
        int l = last, r = n;
        while (l < r) {
            int mid = l + r + 1 >> 1;
            if (preA[mid] - preA[last] <= preB) l = mid;
            else r = mid - 1;
        }

        if (preA[r] - preA[last] == preB) {  // 有解才更新
            ans++;
            preB = 0;
            last = r;
        }
    }
    cout << ans << endl;
}

int main() {
    solve();
}
```

5.1.17 Shuffle Hashing

原题指路: https://codeforces.com/problemset/problem/1278/A

题意 (2 s)

有 t $(1 \le t \le 100)$ 组测试数据. 每组测试数据给定两个非空的、只包含小写英文字母的、长度不超过 100 的字符串 p 和 h . 问 h 中是否存在一个子串是 p 的一个排列.

思路I

将 p 非降序排列后, 枚举 h 的子串, 检查子串排序后是否与 p 相等, 时间复杂度 $O(n^3 \log n)$, 在CF上可通过.

考虑优化, 注意只需枚举 h 的长度为 $|p|$ 的子串, 时间复杂度 $O(n^2 \log n)$.

代码I

```cpp
void solve() {
    string p, h; cin >> p >> h;
    sort(all(p));

    int lenP = p.length(), lenH = h.length();
```

```
    for (int i = 0; i + lenP - 1 < lenH; i++) {
        string tmp = h.substr(i, lenP);
        sort(all(tmp));
        if (tmp == p) {
            cout << "YES" << endl;
            return;
        }
    }
    cout << "NO" << endl;
}

int main() {
    CaseT
    solve();
}
```

思路II

h 的子串 tmp 是 p 的一个排列 iff 其中每种字符的数量对应相等.

预处理 p 中每种字符出现的次数 $cntP[]$ 后, 枚举 h 的长度为 $|p|$ 的子串, 求其中每种字符出现的次数 $cntH[]$, 检查其与 $cntP[]$ 是否相等即可.

时间复杂度 $O(n^2 \cdot |\Sigma|)$.

代码II

```
const int Sigma = 26;

void solve() {
    string p, h; cin >> p >> h;

    vector<int> cntP(Sigma);
    for (auto ch : p) cntP[ch - 'a']++;

    int lenP = p.length(), lenH = h.length();
    for (int i = 0; i + lenP - 1 < lenH; i++) {
        vector<int> cntH(Sigma);
        for (int j = i; j <= i + lenP - 1; j++)
            cntH[h[j] - 'a']++;

        if (cntP == cntH) {
            cout << "YES" << endl;
            return;
        }
    }
    cout << "NO" << endl;
}

int main() {
    CaseT
    solve();
}
```

思路III

同**思路II**, 统计 $tmpH[]$ 时改为维护一个长度为 $|p|$ 的滑动窗口.

时间复杂度 $O(n \cdot |\Sigma|)$.

代码III

```cpp
const int Sigma = 26;

void solve() {
    string p, h; cin >> p >> h;

    vector<int> cntP(Sigma);
    for (auto ch : p) cntP[ch - 'a']++;

    vector<int> cntH(Sigma);
    int lenP = p.length(), lenH = h.length();
    for (int i = 0; i < lenH; i++) {
        cntH[h[i] - 'a']++;
        if (i - lenP >= 0) cntH[h[i - lenP] - 'a']--;

        if (cntP == cntH) {
            cout << "YES" << endl;
            return;
        }
    }
    cout << "NO" << endl;
}

int main() {
    CaseT
    solve();
}
```

思路IV

对每个字符 c , 维护一个bool数组 $equal[c]$ 表示当前是否有 $cntP[c] = cntH[c]$, 维护 $equal[]$ 中为 $true$ 的元素的个数 tot , 则 $tot = |\Sigma|$ 时为一组解.

时间复杂度 $O(n + |\Sigma|)$.

代码IV

```cpp
const int Sigma = 26;

void solve() {
    string p, h; cin >> p >> h;

    vector<int> cntP(Sigma);
    for (auto ch : p) cntP[ch - 'a']++;

    vector<int> cntH(Sigma);
    vector<bool> equal(Sigma);
    int tot = 0;  // equal[]中为true的元素个数
    for (int c = 0; c < Sigma; c++) {
```

```
        equal[c] = (cntP[c] == cntH[c]);
        tot += equal[c];
    }

    auto modify = [&](char ch, int op) {
        int c = ch - 'a';
        tot -= equal[c];
        cntH[c] += op;
        equal[c] = (cntP[c] == cntH[c]);
        tot += equal[c];
    };

    int lenP = p.length(), lenH = h.length();
    for (int i = 0; i < lenH; i++) {
        modify(h[i], 1);
        if (i - lenP >= 0) modify(h[i - lenP], -1);

        if (tot == Sigma) {
            cout << "YES" << endl;
            return;
        }
    }
    cout << "NO" << endl;
}

int main() {
    CaseT
    solve();
}
```