# 字符串基础

2351871 计科 郎若谷

# 前置知识

- 字符集 $\sum$，字符串长度 $|S|$
- 前缀，后缀，真前缀，真后缀
- 子串，子序列
- 字典序
- 回文串

# C++ 中的字符串

- 读入字符串时不推荐 gets() 和 getline()，一般用 cin 和 scanf()，或者用 getchar() 手动实现一个字符串读入就够了

- C++中提供了一个字符串 STL：std::string
- 方便，库函数多，但速度慢，不能直接和 C 标准库里的函数兼容

# 一些比较好用的库函数

- find(ch, start = 0) 查找并返回从 start 开始的字符 ch 的位置（皆从 0 开始）（如果查找不到，返回 $-1$）$O(n)$
- rfind(ch) 即从后面往前找 ch 字符
- substr(start, len) 可以从字符串的 start（从 0 开始）截取一个长度为 len 的字符串。$O(n)$
- push_back(ch) 将字符 ch 添加道字符串的末尾，同理 pop_back()。$O(1)$
- append(s) 将 s 添加到字符串末尾。$O(|s|)$
- erase(pos, n) 删除从 pos 开始的 n 个字符。$O(n)$
- insert(pos, s) 在 pos 位置插入字符串 s。$O(n)$
- 此外，string也重载了小于号，按照字典序比较。$O(n)$

# 字符串匹配

KMP

# 字符串匹配

- 给出两个字符串 $S$ 和 $T$，长度分别是 $n$ 和 $m$，问字符串 $T$ 在字符串 $S$ 中出现了几次
- （字符串 $T$ 又可称为 模式串）
- $n, m \leq 10^6$

# 暴力做法

- 直接枚举 $S$ 串的每一个位置作为匹配的起始位置，然后从该位置开始，往后一位一位地和模式串 $T$ 比较是否一样
- 如果比较发现该位置往后 $|T|$ 个字符都和模式串 $T$ 完全一致，那么就说明 $T$ 在 $S$ 中出现了一次，答案+1

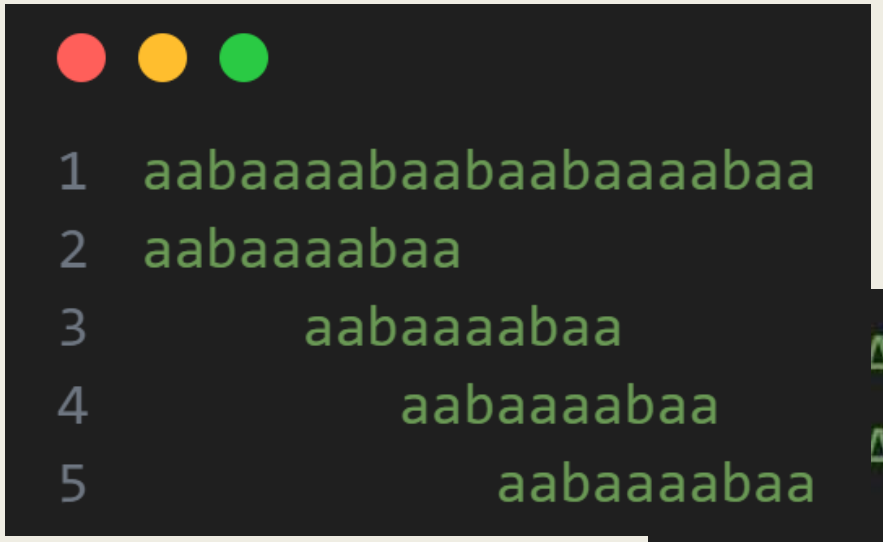
- $O(nm)$

# KMP算法

- 名字来源：1977年 Knuth, Morris, Pratt 三个人发明的

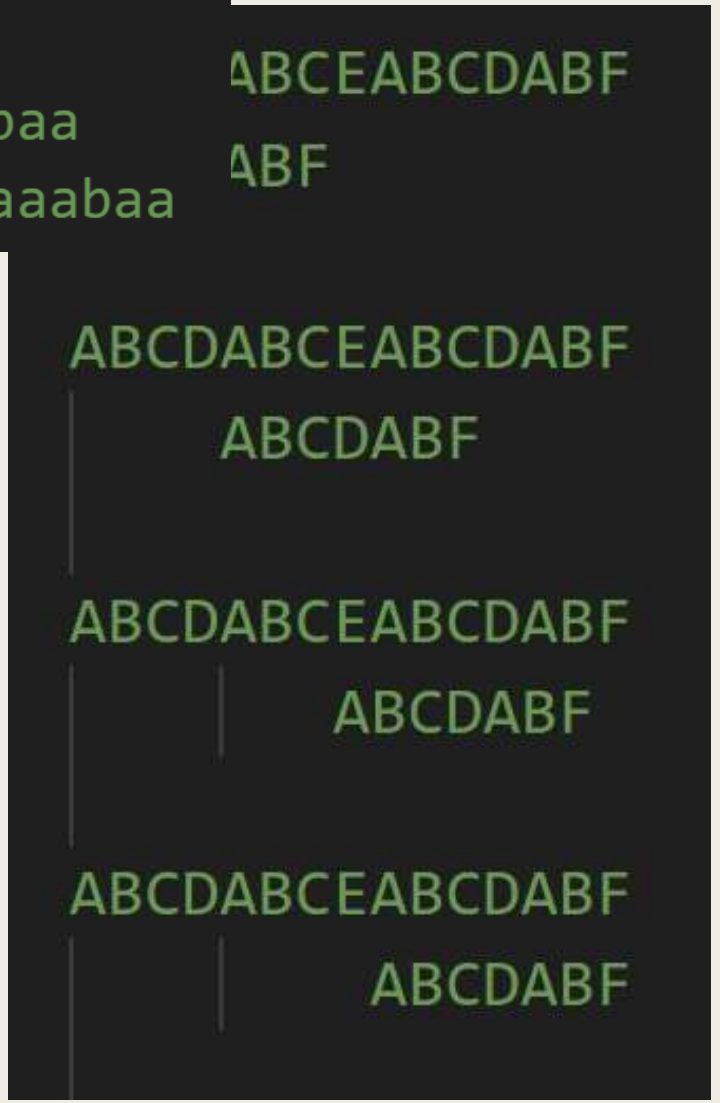
- 考虑优化上述暴力算法，注意到对于每一个新的起始位置，我们都要从头开始循环匹配模式串，把很多之前已经进行过的计算都浪费掉了，每个字符都被进行了大量重复的比较运算。因此考虑优化这个过程
- 考虑能否在某次失配之后，根据之前计算过的东西直接跳到后面的某个位置继续匹配，而非在这个位置的下一个位置从头匹配

# KMP算法




- 我们期望得到这样的一个过程：

- 假设我们匹配到 $S$ 串的 $j$ 位置，模式串 $T$ 串的 $i$ 位置时失配了，那么找到模式串 $[1 \ldots i-1]$ 这个前缀串中一样且最长的一对真前缀和真后缀，假设这个长度是 $l$，把匹配起始位置移动到这个真后缀的开头继续匹配。也就是说让 $S$ 串的 $j$ 位置和 T 串的 $l+1$ 位置继续匹配，如果这两个位置还是失配，那就在 $[1 \ldots (l+1)-1]$ 这个前缀串中重复这个过程



# KMP算法

- 因此对于一个模式串 $T$，我们需要对于每一个位置 $i$，求出一个长度 $l$，表示 $[1 \ldots i]$ 这个 $S$ 串的前缀子串中，长度最长、且完全一样的真前缀和前后缀的长度
- 这些 $l$ 形成一个数组 $nxt_i$，叫做前缀函数数组，或 border 数组，或 next 数组
- 求出这个数组以后，当我们在匹配中失配时，就可以通过这个数组直接跳过若干显然不是答案的位置

# 前缀函数数组

- 那么接下来的问题就是如何求出这个前缀函数数组
- 考虑最暴力的做法，我们可以对于每个前缀子串$[1 \dots i]$，都暴力的枚举每一个真前缀和真后缀，并$O(n)$的判断是否相同
- 复杂度$O(n^3)$

# 前缀函数数组

- 考虑如何优化，注意到如果 $nxt_i$ 如果比 $nxt_{i-1}$ 大，那至多只能比 $nxt_{i-1}$ 大 1
- 因此把枚举范围改成从 $nxt_{i-1}+1 \Rightarrow 0$
- 什么复杂度?
- 注意到 $nxt$ 的提升每次移动至多提升1，而一旦 $nxt$ 下降后续的枚举范围就会变小
- 因此总复杂度$O(n^2)$

# 前缀函数数组

- 继续考虑优化
- 考虑如何优化 $nxt$ 下降的过程，当前的真前缀和真后缀失配后，我们无非是想要找到最长的、相等的两个子串，这两个子串分别是当前真前缀的一个真前缀，当前真后缀的一个真后缀
- 而又因为当前这个真前缀和真后缀是完全一样的，所以问题变成了要找到最长的、相等的两个子串，这俩子串是当前真前缀的真前缀和真后缀
- 而这个东西就是我们已经求出来的 $nxt$
- 因此可以通过 $nxt$ 来转移 $nxt$

# KMP

一个比较完善的代码，为了方便，代码的字符串采用从 0 开始存储。

也是为了方便，代码中的 $nxt_i$ 实际是 $i-1$ 位置的前缀函数数组

```cpp
std::string s, t;
int nxt[inf];
std::vector <int> ans;

void getnxt () {
    int k = 0;
    nxt[1] = nxt[0] = 0;
    for(int i = 1; i < t.size(); i++) {
        while(t[i] != t[k] && k) k = nxt[k];
        nxt[i + 1] = (t[i] == t[k] ? ++k : 0);
    }
    return;
}

void kmp () {
    int k = 0;
    for(int i = 0; i < s.size(); i++) {
        while(s[i] != t[k] && k) k = nxt[k];
        k += (s[i] == t[k]);
        printf("i = %d, K = %d\n", i, k);
        if(k == t.size()) ans.push_back(i - t.size() + 2);
    }
}
```

# 复杂度

- 注意到求 next 数组和进行字符串匹配的过程本质上一样的，都是在失配的时候不断地跳 next 数组
- 每次跳 next 数组的时候，如果 next 比上一个位置大，那至多也只能多 1
- 而如果 next 数组下降，那么跳 next 数组时候的初始值就会减少
- 因此均摊下来的复杂度是 $O(n+m)$

# 字符串哈希

Hash

# 字符串哈希

- 字符串哈希的思想简而言之就是：把任意一个字符串映射成一个范围内的数字，这样我们就可以用一个简单的数字区分出不同的字符串。
- 但是因为数字有范围所以数字个数是有限的，而字符串的数量是无限的，所以很显然一定会有一些字符串会映射成同一个数字，此时哈希就会出问题，我们称这种情况为哈希碰撞
- 所以我们要关注的问题就是如何设计哈希映射的方法以降低哈希碰撞的概率

# 字符串哈希

- 一种最普遍且好写的哈希映射方法是把字符串的每一位看成一个 k 进制数的一位
- 就好比在 16 进制下 A 代表 10，F 代表 15 一样
- 又因为这个数字要在一定的范围内，所以这个 k 进制数要对一个数 P 取模
- 对于一个长为 $L$ 的字符串 $S$ 看成一个 k 进制数：（本质就是一个多项式）

$$Hash(S) = \sum_{i=1}^{L} S_i \times k^{L-i} \pmod{P}$$

# 字符串哈希

$$Hash(S) = \sum_{i=1}^{L} S_i \times k^{L-i} \pmod{P}$$

```cpp
const int K = 103;
const int P = 998244353;

int get_hash(const std::string& s) {
    int res = 0;
    for (int i = 0; i < s.size(); ++i) {
        res = (1ll * res * K + s[i]) % P;
    }
    return res;
}
```

# 字符串哈希

- 考虑哈希碰撞的概率
- 具体计算一个哈希函数碰撞的概率需要一些离散数学的知识，但是我们可以简单地认为在随机情况下，如果模数是 $P$，那么一个字符串和其他 m 个已经出现的字符串碰撞的概率是 $\frac{m}{P}$
- 但是如果若干字符串间存在一些规律，那么多项式算出来的结果很可能是一个等差数列，如果这个公差和模数 P 之间存在一个最大公约数 $u > 1$，那么本质上就等于说模数小了 $u$ 倍，哈希碰撞的概率就大了 $u$ 倍
- 因此一般地，我们都会把模数 $P$ 和进制 $k$ 取质数

# 字符串哈希

- 考虑有 $n$ 个不同的字符串，这些字符串在模 P 意义下在值域上随机分布
- 那么这 $n$ 个字符串完全不存在哈希碰撞的概率就是

$$\prod_{i=0}^{n-1} \frac{P-i}{P}$$

- 考虑到取模的效率问题，P 一般取值在 int 范围内
- 当 n 达到 1e5 级别的时候，P 取 $1e9+7$ 时，完全不碰撞的概率已经趋近于0了（计算得到应该是 0.7%）

- 这样的错误率显然我们无法接受，那么应该怎么办?

# 字符串哈希

- 所以一般情况下，我们都会选择同时选取两个大质数作为模数，对一个字符串同时做两次哈希，如果两个哈希结果都相同才会认为两个字符串相同
- 经验检验这种方法哈希碰撞的概率几乎是 0%
- 但是对于两个大质数取模实际上是非常慢的，所以一般情况下我们选择计算两次哈希的时候一次对大质数取模，另一次对 $2^{32}$ 取模
- 对 $2^{32}$ 取模不需要写取模运算，只需要把数用 unsigned int 类型存起来即可
- 如果数超过 $2^{32}$ 会整型溢出，自然地达到了取模的目的
- 这种方法叫自然溢出
- 经验检验一般情况下对大质数取模+自然溢出可以解决大多数情况

# 子串哈希

- 如果我们想要多次询问一个串 S 的子串的哈希值，如果每次询问都暴力地 $O(n)$ 算一遍将会非常低效
- 我们考虑存下来 S 每一位哈希的结果，也就是令

$$f(x) = \sum_{i=1}^{x} S_i \times k^{x-i} \pmod P$$

- 如果我们想求一个子串 $[l,r]$ 的哈希结果，可以发现其实就是

$$f(r) - f(l-1) \times k^{r-l+1} \pmod P$$

- 我们可以预处理出来所有的 $k^x$，这样对于每一个子串都可以 $O(1)$ 地计算答案了

# 例题

- 字符集为全体小写字母
- $|S| \leq 2^{20}$

小 C 学习完了字符串匹配的相关内容，现在他正在做一道习题。

对于一个字符串 $S$，题目要求他找到 $S$ 的所有具有下列形式的拆分方案数：

$S = ABC$，$S = ABABC$，$S = ABAB \ldots ABC$，其中 $A$，$B$，$C$ 均是非空字符串，且 $A$ 中出现奇数次的字符数量不超过 $C$ 中出现奇数次的字符数量。

更具体地，我们可以定义 $AB$ 表示两个字符串 $A$，$B$ 相连接，例如 $A = \texttt{aab}$，$B = \texttt{ab}$，则 $AB = \texttt{aabab}$。

并递归地定义 $A^1 = A$，$A^n = A^{n-1}A$（$n \geq 2$ 且为正整数）。例如 $A = \texttt{abb}$，则 $A^3 = \texttt{abbabbabb}$。

则小 C 的习题是求 $S = (AB)^i C$ 的方案数，其中 $F(A) \leq F(C)$，$F(S)$ 表示字符串 $S$ 中出现奇数次的字符的数量。两种方案不同当且仅当拆分出的 $A$、$B$、$C$ 中有至少一个字符串不同。

小 C 并不会做这道题，只好向你求助，请你帮帮他。

**输入 #1** 复制

```
3
nnrnnr
zzzaab
mmlmmlo
```

**输出 #1**

```
8
9
16
```

# 例题

- AB 一定是 S 的一个前缀，考虑枚举 $|AB|$，假设其为 L
- 考虑再枚举 AB 的重复次数，对于每一个重复次数都有一个确定的 C
- 可以提前把所有后缀中出现奇数次的字母数量处理出来，那么我们现在只需要知道长度为 L 的前缀中应该如何划分 A 和 B，使得其中 A 中出现奇数次字母的个数小于 C 中出现奇数次的字母个数
- 当我们假设 $|AB| = L$ 时，可以处理出现这 L 个前缀每个前缀中出现奇数次字母的个数，然后以个数为下标存在一个桶里面，当我们确定 C 中的字母个数是 p 时，我们只需要求出来桶中 $[0, p]$ 这个前缀和即可，这个数目就是合法的 A 的个数

# 例题

- 对于一个重复次数 m，我们需要判断 AB 能否重复 m 次，这里直接用子串哈希判断即可
- 至于复杂度，注意到对于一个长度 L，重复次数至多为 $[\frac{n}{L}]$，L 的取值是从 1 到 n
- 可以用调和级数算出来所有的 L 下合法重复次数至多为 $n\ln n$ 个
- 总复杂度 $O(26n\ln n)$，26 是字符集大小，想要通过可能比较极限，但应该没问题
- 其实可以用树状数组进一步优化成 $O(n\ln n\log 26)$

- NOIP2020 提高 T2
- P7114 字符串匹配

# 字符串匹配

- 给出两个字符串 $S$ 和 $T$，长度分别是 $n$ 和 $m$，问字符串 $T$ 在字符串 $S$ 中出现了几次

- $n, m \leq 10^6$

# 字符串匹配

- 直接求出每一个长度为 $|T|$ 的 $S$ 中的子串的哈希即可
- 效率上不及 KMP，但是复杂度也是 $O(n+m)$

- 哈希非常强大，很多其他算法的经典问题都可以用哈希解决
- 但是这不意味着只需要搞懂哈希就够了
- 一方面是双模哈希大多数情况效率是不及其他算法的
- 另一方面是有些比较复杂的问题哈希还是无法解决的，但是可以套用其他算法的思想加以解决

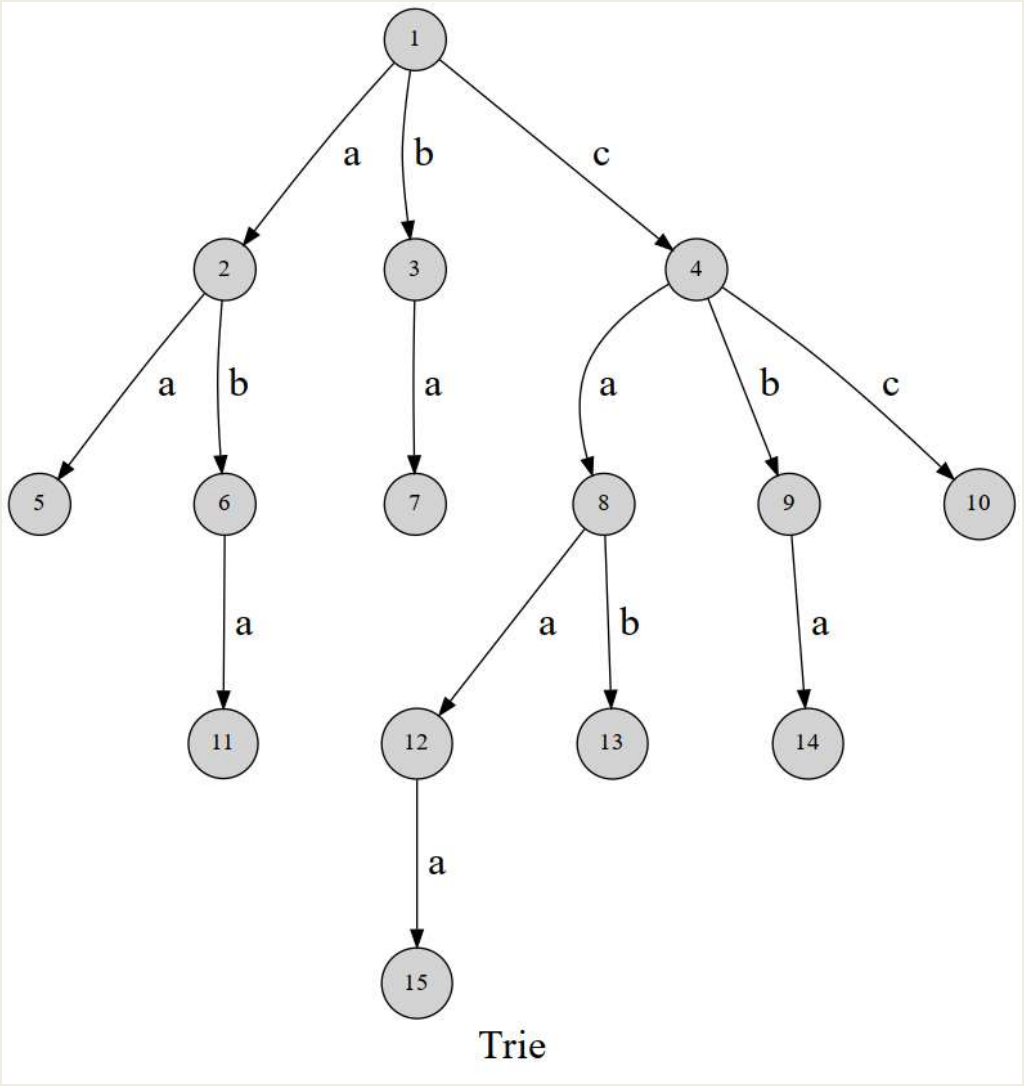
# 字典树

Trie

# 字典树

- 把若干字符串放到树上，一个边代表一个字母
- 以此建立一棵树。比如字符串
- aa,aba,ba,caaa,cab,cba,cc
- 这几个字符串建的字典树就形如这样

- 根到某个节点的路径就代表一个出现过的前缀



# 例题

给定 $n$ 个模式串 $s_1, s_2, \ldots, s_n$ 和 $q$ 次询问，每次询问给定一个文本串 $t_i$，请回答 $s_1 \sim s_n$ 中有多少个字符串 $s_j$ 满足 $t_i$ 是 $s_j$ 的**前缀**。

一个字符串 $t$ 是 $s$ 的前缀当且仅当从 $s$ 的末尾删去若干个（可以为 0 个）连续的字符后与 $t$ 相同。

输入的字符串大小敏感。例如，字符串 `Fusu` 和字符串 `fusu` 不同。

## 输入格式

**本题单测试点内有多组测试数据。**

输入的第一行是一个整数，表示数据组数 $T$。

对于每组数据，格式如下：
第一行是两个整数，分别表示模式串的个数 $n$ 和询问的个数 $q$。
接下来 $n$ 行，每行一个字符串，表示一个模式串。
接下来 $q$ 行，每行一个字符串，表示一次询问。

**输入 #1**

```
3
3 3
fusufusu
fusu
anguei
fusu
anguei
kkksc
5 2
fusu
Fusu
AFakeFusu
afakefusu
fusuisnotfake
Fusu
fusu
1 1
998244353
9
```

**输出 #1**

```
2
1
0
1
2
1
```

- $T, n, q \leq 10^5, \sum |S| \leq 10^6$

# 例题

- 每个节点都存一个 cnt，表示这个节点到根所代表的前缀出现的次数
- 字典树上插入字符串的时候把每个节点的 cnt +1
- 查询的时候按照字典树走到最终的节点，并输出那个节点的 cnt 即可
- 注意清空时候的时间复杂度即可

- Luogu P8306

# 例题

```cpp
int conchar(char c) {
    if(std::isdigit(c)) return c - '0' + 52;
    if(c >= 'A' && c <= 'Z') return c - 'A';
    return c - 'a' + 26;
}

void insert(int x, int now) {
    cnt[now]++;
    if(x == s.size()) return;
    int c = conchar(s[x]);
    if(trie[c][now] == 0) trie[c][now] = ++ndx;
    insert(x + 1, trie[c][now]);
}

int find(int x, int now) {
    if(x == s.size()) return cnt[now];
    int c = conchar(s[x]);
    if(trie[c][now] == 0) return 0;
    return find(x + 1, trie[c][now]);
}
```

```cpp
void solve() {
    clear();
    std::cin >> n >> q;
    for(int i = 1; i <= n; i++) {
        std::cin >> s;
        insert(0, 0);
    }
    while(q--) {
        std::cin >> s;
        std::cout << find(0, 0) << '\n';
    }
}
```

# 例题

- 给你一棵带边权的树，求 $(u,v)$ 使得 $u$ 到 $v$ 的路径上的边权异或和最大，输出这个最大值。这里的异或和指的是所有边权的异或。
- 点数 $n \leq 10^5$，边权 $w < 2^{31}$

# 例题

- 注意到路径异或和可以转化为两个到根路径的异或和的异或
- 因此维护每一个点到根节点的异或和即可
- 我们考虑每次依次把每个异或和插入到一个 01 Trie 上
- 01 Trie 就是对只有 01 的二进制数建 Trie，每个二进制都看成只有0和1的字符串
- 我们考虑从 Trie 的根开始往下走，如果当前位是0就尽量走1边，当前位是1就尽量走0边，如果想走的边不存在那就走存在的边
- 这样就尽可能地能异或出来大的数了
- $O(n \log w)$
- luogu P4551

# THANKS