哈希

考虑一种字符串的简单映射方法：将字符串看成一个 $|\Sigma|$ 进制的数，将其在 $(\mathrm{mod}\ p)$ 的意义下转化成十进制。这种方法可以近似的看做是随机的映射，可以用以快速判断两个字符串是否相等。这种映射方法是一种字符串哈希。

如果维护字符串所有前缀的哈希值，那么我们可以 $O(1)$ 地计算出其任意一个字串的哈希值，参考接下来的例题。

例题一

给出两个字符串 s, t，每次查询两个字符串的子串，问是否相等。

```
#include <bits/stdc++.h>

#define maxn 1000005
#define base 2333
#define ull unsigned long long

using namespace std;

int q;
ull pw[maxn];

struct str{
    int n;
    char s[maxn];
    ull hs[maxn];

    void get() {
        scanf("%s", s + 1);
        n = strlen(s + 1);
```

```
        pw[0] = 1;
        for (int i = 1; i <= n; i++) {
            hs[i] = hs[i - 1] * base + s[i];
            pw[i] = pw[i - 1] * base;
        }
    }

    ull Hs(int l, int r) {
        return hs[r] - hs[l - 1] * pw[r - l + 1];
    }

}s, t;

int main() {
    s.get();
    t.get();
    scanf("%d", &q);
    while (q--) {
        int l, r, x, y;
        scanf("%d%d%d%d", &l, &r, &x, &y);
        puts(r - l == y - x && s.Hs(l, r) == t.Hs(x, y) ? "Yes" :
"No");
    }
    return 0;
}
```

注意， `r-l==y-x` 是**不能删去**的，删去的话哈希值极容易发生冲突。

例题二

给定两个字符串 s,t，询问 t 在 s 中的出现次数。

哈希即可。

KMP

设 $next_i$ 表示 $s_{1...i}$ 的非平凡的最长的相等的前后缀。

当 s_j 匹配到 t_i 时，如果下一个位置不匹配，那就令 $j = next_j$ 直到下一个位置能匹配或者 $j = 0$ 为止。

注意到 i 的移动是单调的，j 的移动可以均摊，所以求解的复杂度为 $O(n)$。

预处理 $next$ 的时候其实就是拿自己跟自己做匹配，因此也是 $O(n)$ 的。

代码如下：

```
#include <bits/stdc++.h>

#define maxn 1000005

using namespace std;

int n, m;
char s[maxn];
char t[maxn];
int nxt[maxn];

void iNit() {
    for (int i = 2, j = 0; i <= n; i++) {
        while (j && s[i] != s[j + 1]) j = nxt[j];
        nxt[i] = (j += (s[i] == s[j + 1]));
    }
}

void Kmp() {
    for (int i = 1, j = 0; i <= m; i++) {
        while (j && t[i] != s[j + 1]) j = nxt[j];
        if (t[i] == s[j + 1]) ++j;
        if (j == n) printf("%d\n", i - n + 1), j = nxt[j];
    }
}

int main() {
    scanf("%s%s", t + 1, s + 1);
```

```
    n = strlen(s + 1);
    m = strlen(t + 1);
    iNit();
    Kmp();
    for (int i = 1; i <= n; i++) printf("%d ", nxt[i]);
    return 0;
}
```

失配树

一个字符串所有相等的前后缀称为他的 border。

对于 KMP 中的 $next_i$，$s_{1\dots next_i}$ 就是 $s_{1\dots i}$ 的最大 border。

注意到一个串的 border 的 border 仍然是他的 border。

要确定不同 border 之间的关系，就可以使用失配树。

每个前缀朝他的最大 border 连边就得到了失配树。

P5829 【模板】失配树

多次询问 s 的两个前缀的最长公共 border。

$n \leq 10^6$

Sol

失配树上的 lca。

拓展 KMP

对于每个后缀 $s_{i\dots n}$，记 $z_i = \operatorname{lcp}(s, s_{i\dots n})$。

设 $[l, r]$ 表示 $s_{1\dots r-l+1} = s_{l\dots r}$ 的一个区间。

对于 $l \le i \le r$，一定有 $z_i \ge \min(z_{i-l}, r-i+1)$。

- 如果 $z_{i-l} < r-i+1$，那么 $z_i = z_{i-l}$。
- 如果 $z_{i-l} \le r-i+1$，那么继续暴力向后匹配直到不匹配为止得到 z_i。

我们初始令 $l = r = 0$，当 $i > r$ 时暴力求出 z_i，当 $i \le r$ 时利用上述做法求出 z_i，并不断尝试用 $[i, i+z_i-1]$ 更新 $[l, r]$，我们就能在 $O(n)$ 时间复杂度内求出 z 数组。

```
void zinit() {
    int l = 0, r = 0;
    for (int i = 2; i <= n; i++) {
        if (i <= r && z[i - l] < r - i + 1) z[i] = z[i - l];
        else {
            z[i] = max(0, r - i + 1);
            while (i + z[i] <= n && s[z[i]] == s[i + z[i]]) ++z[i];
        }
        if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
    }
}
```

AC自动机

相信大家都会，但是再讲一下。

将所有模式串的 trie 树建出来，再类似于 kmp 一样求一下 next 指针就可以了。

注意此处用到了一个优化，不需要暴力跳 next，可以预处理出沿着这条边跳 next 的最终结果。

```
#include <bits/stdc++.h>

#define maxn 1000005

using namespace std;

int n, m;
char s[maxn];

int tot;
int t[maxn][27];
int nxt[maxn];
int cnt[maxn];

void insert(string s, int n) {
    int p = 0;
    for (int i = 1; i <= n; i++) {
        auto &x = t[p][s[i] - 'a'];
        if (!x) x = ++tot;
        p = x;
    }
    ++cnt[p];
}

queue<int> q;

void bfs() {
    for (int i = 0; i < 26; i++) if (t[0][i]) q.push(t[0][i]);
    while (!q.empty()) {
        int now = q.front(); q.pop();
        for (int i = 0; i < 26; i++) {
            if (!t[now][i]) t[now][i] = t[nxt[now]][i];
```

```
            else {
                q.push(t[now][i]);
                nxt[t[now][i]] = t[nxt[now]][i];
            }
        }
    }
}

long long qry(char s[], int n) {
    long long ans = 0;
    int p = 0;
    for (int i = 1; i <= n; i++) {
        p = t[p][s[i] - 'a'];
        ans += cnt[p];
        cnt[p] = 0;
    }
    return ans;
}

void clear() {
    for (int i = 0; i <= tot; i++) memset(t[i], 0, sizeof t[i]),
cnt[i] = nxt[i] = 0;
    tot = 0;
}

int main() {
    while (1) {
        scanf("%d", &n);
        if (!n) break;
        clear();
        for (int i = 1; i <= n; i++) {
            scanf("%s", s + 1);
            insert(s, strlen(s + 1));
        }
        bfs();
        scanf("%s", s + 1);
        printf("%lld\n", qry(s, strlen(s + 1)));

    }
    return 0;
}
```

例题

给定 n 个字符串 $s_1, s_2 .. s_n$，查询其他串中在其中出现次数之和最大的串。

Sol

建出来每个串在上面都跑一跑就好了。

Border Theory

border 定义如前文所述。

周期定义：如果 $\forall i \leq |s| - T, s_i = s_{i+T}$，则称 T 为 s 的周期。

性质

s 存在一个长为 p 的 border 则存在一个周期 $|s| - p$。

证明

长为 p 的 border 意味着 $\forall i \leq p, s_i = s_{|s|-p+i}$，也就是 $\forall i \leq |s| - (|s| - p), s_i = s_{i+(|s|-p)}$ 也就是 $|s| - p$ 是 s 的周期。

弱周期定理 WLP

若 p, q 均为 s 的周期，且 $p + q \leq |s|$，则 $gcd(p, q)$ 也是 s 的周期。

证明

不妨设 $p > q$。

当 $i \le q$ 时，$s_i = s_{i+p} = s_{i+p-q}$ 即 $s_i = s_{i+(p-q)}$。

当 $i > q$ 时，$s_i = s_{i-q} = s_{i-q+p}$ 也即 $s_i = s_{i+(p-q)}$。

故 $p-q$ 也是 s 的周期。根据辗转相减法可知 $gcd(p,q)$ 也为 s 的周期。

强周期定理 SLP

若 p, q 均为 s 的周期，且 $p + q - gcd(p,q) \le |s|$，则 $gcd(p,q)$ 也是 s 的周期。

证明

略。（不重要）

定理

s 所有长度大于 $\lfloor \frac{|s|}{2} \rfloor$ 的 border 与 |s| 构成等差数列。

证明

设 s 的最大 border 为 $|s| - p$，则 p 为 s 的周期，因此 p 的倍数都是 s 的周期，因此 $|s| - kp$ 都是 s 的border。

设有一个 border 为 $|s| - q > \lfloor \frac{|s|}{2} \rfloor$，则 q 也为 s 的周期，且 $p + q \le |s|$，因此 $\gcd(p,q)$ 也为 s 的周期。由于 $|s| - p$ 已经是最大的 border 了，所以一定有 $gcd(p,q) = p$，也就是 $q = kp$，也即上段所述的 border 即包含了所有长度大于等于 $\lfloor \frac{|s|}{2} \rfloor$ 的 border 了。

定理

s 的所有 border 可以划分为不超过 $\log |s|$ 个等差数列。

证明

上一个定理已经处理了长度在 $(\lfloor \frac{|s|}{2} \rfloor, |s|]$ 中的所有 border。

对于长度在 $(\lfloor \frac{w}{2} \rfloor, w]$ 中的所有 border，我们先取其中最长的一个 p，那么所有 border 都是 p 的 border 了，我们对 $s_{1\ldots p}$ 使用上一个定理就可以得知这一段中所有 border 也是等差数列。

不断地往下分即可将所有 border 划分为 $\log |s|$ 个等差数列。

MaNaChEr

我们可以在每个字符间插入一个特殊字符使得所有回文串都转化为奇回文串。

设 f_i 表示以 i 为回文中心的最大回文半径。

与拓展 KMP 类似，我们仍然考虑一段回文子串，用回文中心 p 表示。

如果 $p < i \le p + f_p - 1$，那么一定有 $f_i \ge min(f_{2p-i}, p + f_p - i)$。

- 如果 $f_{2p-i} < p + f_p - i$，那么 $f_i = f_{2p-i}$。
- 如果 $f_{2p-1} \ge p + f_p - i$，那么继续往后暴力匹配直到不匹配为止得到 f_i。

我们初始令 $p = 0$，当 $i > p + f_p - 1$ 时暴力求出 f_i，当 $i \le p + f_p - 1$ 时利用上述做法求出 f_i，并不断尝试用 i 更新 p，我们就能在 $O(n)$ 时间复杂度内求出 f 数组。

最小表示法

一个长度为 n 的字符串有 n 个循环同构串，其中字典序最小的那个称为这个字符串的最小表示法。

显然先倍长 s，考虑暴力：目前可能成为答案的两个起始点分别为 i,j。那么用 k 暴力往后跳，直到 $s_{i+k}\neq s_{j+k}$，不妨设 $s_{i+k}<s_{j+k}$，那么此时原本应该 `j++`，但是注意到 $[j,j+k]$ 中所有点实际上都不可能成为答案了，所以可以直接 `j+=k+1`。容易发现每次暴力跳 k 都会使得 i 或者 j 增加跳的步数，而 i,j 最多跳 $O(n)$ 步，所以时间复杂度 $O(n)$。

例题

给定两个字符串 s,t，判断两个字符串是否循环同构。

Sol

直接求出 s 和 t 的最小表示法是否相同即可。

```
#include <bits/stdc++.h>

#define maxn 2000005

using namespace std;

int n, m;
char s[maxn];
char t[maxn];

int zx(char s[], int n) {
    for (int i = 1; i <= n; i++) s[i + n] = s[i];
    int i = 1, j = 2, k = 0;
    while (i <= n && j <= n) {
        k = 0;
        while (i + k <= n * 2 && j + k <= n * 2 && s[i + k] == s[j +
k]) ++k;
        if (s[i + k] < s[j + k]) j += k + 1;
        else i += k + 1;
        if (i == j) ++j;
    }
```

```
    return i <= n ? i : j;
}

int main() {
    scanf("%s%s", s + 1, t + 1);
    n = strlen(s + 1);
    m = strlen(t + 1);
    if (n == m) {
        int x = zx(s, n);
        int y = zx(t, m);
        for (int i = 1; i <= n; i++) {
            if (s[x + i - 1] != t[y + i - 1]) {
                puts("No");
                return 0;
            }
        }
        puts("Yes");
    }
    else puts("No");
    return 0;
}
```

后缀数组

注意到 AC 自动机只能离线多模匹配，如果强制在线，则只能做到 $O(n|S| + \sum |T|)$ 的复杂度，显然是不能接受的。

考虑对母串进行一些处理。

如果将母串所有后缀插入一颗字典树，那么模式串只需要在上面查询就可以做到 $O(|S|^2 + \sum |T|)$ 的复杂度，显然还是非常不优秀。

不妨考虑缩减我们所需要的信息：我们只需要所有后缀在 trie 上的 dfs 序，也就是所有后缀的字典序排名，我们记这个数组为 rk_i，那么满足 $sa_{rk_i} = i$ 的数组 sa 就是我们所说的后缀数组。

显然最暴力的求法就是直接 sort + 硬字符串比较，时间复杂度 $O(n^2 \log n)$，非常劣。

注意到我们要比较同一字符串的两个字串，故我们可以使用二分哈希求出两个字串第一个不同的位置，就能 $O(\log n)$ 比较字典序，总复杂度 $O(n \log^2 n)$。

考虑另外一种算法。倍增，每次对所有后缀的长为 2^k 的前缀排序，不足的用极小字符补足，那么比较两个 2^k 的字符串比较就可以先比较前 2^{k-1} 的部分，如果相同再比较后 2^{k-1} 的部分即可。倍增 k 之后内部使用 sort 进行双关键字排序，时间复杂度是 $O(n \log^2 n)$。

但是我们发现两个关键字的范围都是不超过 n 的，所以使用桶排序或计数排序就可以做到 $O(n \log n)$ 的复杂度。

但同时还有另外一种做法，2^k 的 sa 实际上是 2^{k-1} 按第一关键字排序之后的结果，所以如果将第一关键字相同的每一段按第二关键字分别排序，那么据 JuanZhang 的高妙理论，直接 sort 就是 $O(n \log n)$ 的。

实测下来最后三个算法加上线性算法跑 1e6 大概速度是 1000ms，200ms，300ms，130ms。

所以实际上大多数时候采用二三做法就好。

算法二代码：

```
#include <bits/stdc++.h>

#define maxn 1000005

using namespace std;

int n;
int y[maxn];
int c[maxn];
int sa[maxn];
char s[maxn];
int rk[maxn];
```

```
inline void Sort(int m) {
    for (int i = 1; i <= m; i++) c[i] = 0;
    for (int i = 1; i <= n; i++) ++c[rk[i]];
    for (int i = 1; i <= m; i++) c[i] += c[i - 1];
    for (int i = n; i >= 1; i--) sa[c[rk[y[i]]]--] = y[i];
}

inline bool cmp(int i, int j, int k) {
    return y[i] != y[j] || y[i + k] != y[j + k];
}

int *suffix_array(char s[], int m) {
    for (int i = 1; i <= n; i++) rk[i] = s[i], y[i] = i;
    Sort(m);
    for (int k = 1; k <= n; k <<= 1) {
        int cnt = 0;
        for (int i = n - k + 1; i <= n; i++) y[++cnt] = i;
        for (int i = 1; i <= n; i++) if (sa[i] > k) y[++cnt] = sa[i]
- k;
        Sort(m);
        swap(y, rk);
        rk[sa[1]] = m = 1;
        for (int i = 2; i <= n; i++) rk[sa[i]] = (m += cmp(sa[i],
sa[i - 1], k));
        if (m == n) break;
    }
    return sa;
}

int main() {
    scanf("%s", s + 1);
    n = strlen(s + 1);
    auto res = suffix_array(s, 127);
    for (int i = 1; i <= n; i++) printf("%d ", res[i]);
    return 0;
}
```

算法三代码

```
#include <bits/stdc++.h>

#define maxn 2000005

using namespace std;

int n;
char s[maxn];
int sa[maxn];
int rk[maxn];
int tmp[maxn];

int *suffix_array(char s[]) {
    for (int i = 1; i <= n; i++) sa[i] = i, rk[i] = s[i];
    sort(sa + 1, sa + n + 1, [](int x, int y) {return rk[x] <
rk[y];});
    for (int k = 1; ; k <<= 1) {
        int m = 0;
        for (int l = 1, r; l <= n; l = r + 1) {
            r = l;
            while (r < n && rk[sa[r + 1]] == rk[sa[l]]) ++r;
            sort(sa + l, sa + r + 1, [&](int x, int y) {return rk[x
+ k] < rk[y + k];});
            tmp[sa[l]] = ++m;
            for (int i = l + 1; i <= r; i++) tmp[sa[i]] = (m +=
(rk[sa[i - 1] + k] != rk[sa[i] + k]));
        }
        swap(tmp, rk);
        if (n == m) break;
    }
    return sa;
}

int main() {
    scanf("%s", s + 1);
    n = strlen(s + 1);
    auto sa = suffix_array(s);
    for (auto i = 1; i <= n; i++) printf("%d ", sa[i]);
    return 0;
}
```

例题

多次强制在线每次给定 t，询问其在 s 中的出现次数。

$|s|, \sum |t| \leq 10^6$

Sol

先求出 s 的后缀数组，然后 t 在后缀数组上二分即可找到第一次出现以及最后一次出现，单次查询复杂度为 $O(|t| \log |s|)$。

Height 数组

> height 数组可以说是后缀数组的灵魂了。

注意到我们光是拿着个后缀数组还是做不了一些后缀树可以干的事情，比如询问一个字符串两个后缀的最长公共前缀，这个在后缀树上就是一次查询 lca，非常的方便。~~(五块五包邮)~~

我们不妨在后缀数组上维护一个 $height_i = \text{lcp}(suffix_{sa_{i-1}}, suffix_{sa_i})$，即 sa 上相邻两个后缀的 lcp 的长度。

注意到一个关键性质：$height_{rk_i} \geq height_{rk_{i-1}} - 1$

显然捏qwq

然后就可以线性求出 $height$ 啦：

```
int h[maxn];

void height_array() {
    for (int i = 1, k = 0; i <= n; i++) {
        if (rk[i] == 0) continue;
        if (k) --k;
        while (s[i + k] == s[sa[rk[i] -1] + k]) ++k;
        h[i] = k;
    }
}
```

又注意到另一个关键性质 $\text{lcp}(suffix_i, suffix_j) = \min\limits_{rk_i<k\le rk_j} height_k$。

显然捏qwq

然后就可以用 $O(n\log n)\sim O(1)$ 的 RMQ 查询任意两个后缀的 lcp 了。

例题

给定字符串 s，询问 $\sum\limits_{1\le i<j\le n}\text{lcp}(suffix_i, suffix_j)$。

Sol

相当于求出 $height$ 之后询问所有区间最小值之和，显然单调栈两边跑一下就完了。

例题

给定字符串 s，询问其本质不同的子串个数。

Sol

按照 sa 的顺序依次考虑所有后缀所新产生的与 [前面所有后缀产生的字串] 不同的子串。

发现就是 $\text{len}(suffix_{sa_i}) - height_i$，那么答案就是 $\binom{n+1}{2} - \sum_{i=2}^{n} height_i$。

广义后缀数组

如果是关于多个字符串的子串或后缀问题，那么就需要广义后缀数组。

只需要将所有字符串连接起来，中间用一个极小符号做分割，并且认为极小符号与极小符号不同，直接求 sa 即可。

例题

给定 n 个字符串 $s_1, s_2 \dots s_n$，求所有串的本质不同的字串数。

Sol

直接在广义后缀数组上跑上一题的做法即可。

分组 trick

广为人知，直接看题。

例题

给定一个字符串 s，询问最长的出现次数超过 k 的子串。

Sol

答案有单调性，可以直接二分。

将 $height$ 超过 mid 的连续一段划分为同一组，如果存在一组大小超过 k，那么就是合法的。

时间复杂度 $O(n \log n)$。

例题

给定 n 个字符串 $s_1, s_2 \dots s_n$，求其最长公共子串。

Sol

广义后缀数组上二分分组，然后如果存在一组包含了 1~n 的后缀，那么就是合法的。

并查集 trick

同样广为人知的另一个 trick。

模拟分组的过程。

从大到小枚举分组的界限，那么每次一个 $height$ 从不可行变成可行，就是两个组的合并操作，如果信息支持合并，那么就可以那并查集+可并数据结构进行维护了。

注意到链上并查集的复杂度是 $O(n)$ 的，所以使用并查集并不会成为大部分 sa 题目的瓶颈。

例题 P3804 【模板】后缀自动机 (SAM)

给定一个只包含小写字母的字符串 S。

请你求出 S 的所有出现次数不为 1 的子串的出现次数乘上该子串长度的最大值。

Sol

显然此时就不具有可二分性了。

我们用并查集模拟分组过程。

显然每次之后合并一组的时候，新的到的 $height \times size$ 有可能更新答案。

因为其他所有组的 $size$ 没变，而对应的 $height$ 变小了，所以一定更新不了答案。

所以总复杂度 $O(n \log n)$ 或 $O(n)$。

例题 自缚川 ~ Self Tying River

记一个字符串 t 在另一个字符串 s 中匹配的最大次数为 $f_s(t)$。

例如 aba 在 abababa 中匹配的最大次数为 3。

称若干个字符串 $s_1, s_2 \dots s_m$ 是 BL - 捆绑的，当且仅当存在一个字符串 t，使得 $|t| = L$ 且 $\forall 1 \le i \le m, f_{s_i}(t) \ge B$。

小柯希望能够理（rou）性（ti）愉悦，所以希望你从他发现的 n 个字符串中，找到 m 个 BL - 捆绑的字符串 $s_1, s_2 \dots s_m$，使得 $m \times B \times L$ 尽量大，并输出最大值。

$n \times S \le 10^5$。

Sol

首先考虑分组。

枚举长度 L，然后扫一遍进行分组，得到每一组内每种颜色的出现次数，然后 mB 就可以排一个序贪心选就可以了。

枚举 L 是 $O(S)$，每一组做一遍是 $O(nS)$，所以这种做法时间复杂度为 $O(nS^2)$。

再考虑并查集。

每次合并的时候暴力合并两个组的颜色桶，合并之后更新一次答案，更新方式如上。

总共要合并 $O(nS)$ 次，每次合并是 $O(n)$ 的，所以这种做法时间复杂度为 $O(n^2S)$。

那么综合两种做法，谁小就让谁带平方，那么总时间复杂度就是 $O(nS\sqrt{nS})$，可以通过此题。

例题 P5576 [CmdOI2019]口头禅

蒟蒻出题人收集了某位大佬的 n 条语录，并按时间为序编号为 $1...n$ 。

他发现这位大佬的口头禅是随着时间而变化的，而且里面有些看不懂的内容。

在请教了群 DS 带师之后，他得到了某种 hash 方法，把这些语录都变成了 01 串，这样似乎好懂一些。

为了研究水群的奥秘，他进行了多次询问 : $[l,r]$ **之间的所有语录，最长公共子串的长度是多少**?

出题人知道这并不是一个简单的问题，所以他并不急于即时得知每个询问的答案。

$n \le 2 \times 10^4, m \le 10^5, \sum |s| \le 4 \times 10^5$。