

ACM 模板

rogeryoung

2021 年 6 月 3 日

目录

第一章 上号	1
1.1 头文件	1
1.2 预编译	1
1.3 进制转换	1
1.4 常见技巧	2
1.5 二分查找	2
1.6 矩阵乘法	2
1.7 快速幂	3
1.8 快速排序	4
1.9 第 k 大数	4
1.10 归并排序求逆序对	4
第二章 数学	5
2.1 GCD 和 LCM	5
2.2 EXGCD	5
2.3 数论函数	5
2.4 乘法逆元	6
2.5 筛法	6
2.6 素性测试	7
2.6.1 试除法	7

2.6.2 Miller Rabin	8
2.6.3 排列组合	8
2.6.4 Lucas 定理	8
2.7 约瑟夫 Josephus 问题	8
2.8 中国剩余定理	9
2.9 Mobius 反演	9
2.10 博弈	10
2.10.1 Nim 博弈	10
2.10.2 Wythoff 博弈	10
2.11 生成函数	10
2.12 容斥原理	10
2.13 组和排列	11
2.13.1 数学常数	11
2.13.2 常见公式	11
第三章 图论	13
3.1 链式前项星	13
3.2 最短路	13
3.2.1 Dijkstra	13
3.2.2 Bellman-Ford	13
3.2.3 Floyd	14
3.3 最近公共祖先 LCA	14
第四章 动态规划	15
4.1 背包	15
4.1.1 01 背包	15
4.1.2 完全背包	15
4.1.3 多重背包	15
4.2 最长公共上升序列	15
4.3 数字计数	16

第五章 数据结构	17
5.1 链表	17
5.2 滑动窗口	17
5.3 树状数组	18
5.4 ST 表	19
5.5 线段树	19
第六章 字符串	21
6.1 KMP	21
6.2 Manacher	22

第一章 上号

1.1 头文件

</> 代码 1.1: /上号/头文件.hpp

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 typedef long long ll;
4 typedef long double ld;
5 #define _fora(i,a,n) for(ll i=(a);i<=(n);i++)
6 #define _forz(i,a,n) for(ll i=(a);i>=(n);i--)
7 #define _forb(i,a) for(ll i=(a);i>0;i-=i&(-i))
8 #define _fore(i,a) for(int i=head[(a)];i=edge[i].nxt)
9 #define _in(i,min,max) ( ((i)-(min)) | ((max)-(i)) )
10 #define _dbg(...) printf(__VA_ARGS__)
11 #define LN putchar('\n')
12
13 inline ll rr() {
14     ll s = 0, w = 1; char c = getchar();
15     while(c<'0' || c>'9') { if(c=='-')w=-1; c=getchar(); }
16     while(c>='0' && c<='9') { s=s*10+c-'0'; c=getchar(); }
17     return s*w;
```

18 }

1.2 预编译

头文件引入方式改为如下, 可以把头文件放入 lab.hpp, 然后使用 clang++ lab.hpp 预编译。实际编译使用 clang++ lab.cpp -DRYLOCAL 添加条件编译参数。

```
1 #ifdef RYLOCAL
2 #include "lab.hpp"
3 #else //include <bits/stdc++.h>
4 #include <cstdio> #include <iostream> #include <functional>
5 #include <cmath> #include <cstring> #include <algorithm>
6 #endif
```

1.3 进制转换

</> 代码 1.2: /上号/进制转换.cpp

```
1 void pr(ll n, ll x) {
2     if(n >= x) pr(n / x, x);
3     putchar(n%x + (n%x > 10 ? 'A'-10 : '0'));
4 }
5 void pr(ll n) {
6     if(n >= 10) pr(n / 10);
7     putchar(n % 10 + '0');
8 }
```

1.4 常见技巧

向上取整 p/q 为 $(p-1)/q+1$ 。

预计算 \log_n ，只需 `_fora(i, n, MN) logn[i] = logn[i/n] + 1;`

字典序 `strcmp(x,y) < 0`。

1.5 二分查找

STL 二分 在 $[l, r)$ 查找 $\geq v$ 中最前的一个，找不到则返回 r 。支持 `cmp` 函数。

```
1 T* lower_bound(T* l, T* r, const T& v);
```

在 $[l, r)$ 查找 $> value$ 中最前的一个，找不到则返回 r 。支持 `cmp` 函数。

```
1 T* upper_bound(T* l, T* r, const T& v);
```

手写二分，在单增（单减）数组中查找 $\geq x$ ($\leq x$) 的数中最前的一个。

</> 代码 1.3: /上号/二分/01.cpp

```
1 while (l < r) {
2     int mid = (l + r) >> 1;
3     if (aa[mid] >= x) r = mid; // <=
4     else l = mid + 1;
5 } return l;
```

在单增（单减）数组中查找 $\leq x$ ($\geq x$) 的数中最后的一个。

</> 代码 1.4: /上号/二分/02.cpp

```
1 while (l < r) {
2     int mid = (l + r + 1) >> 1;
3     if (aa[mid] <= x) l = mid; // >=
4     else r = mid - 1;
```

```
5 } return l;
```

对于上凸（ \wedge 形）函数，可以使用三分法来查找最大值。对于下凸（ \vee 形）变号即可

</> 代码 1.5: /上号/三分法.cpp

```
1 while (r - l > eps) {
2     ld mid = (r + l) / 2;
3     if (f(mid + eps) > f(mid - eps))
4         l = mid;
5     else
6         r = mid;
7 }
```

1.6 矩阵乘法

构建一个 p 行 q 列的矩阵。

</> 代码 1.6: /上号/矩阵乘法.cpp

```
1 struct Mtx {
2     ll m[MAXM][MAXM], p, q;
3     Mtx(ll p, ll q) : p(p), q(q) {
4         memset(m, 0, sizeof(m));
5     }
6     Mtx operator * (Mtx& mtx) {
7         Mtx c(p, mtx.q);
8         _fora (i, 1, p) { _fora (k, 1, q) {
9             ll t = m[i][k];
10            _fora (j, 1, mtx.q) {
11                c.m[i][j] += t * mtx.m[k][j];
```

```

12         c.m[i][j] %= MOD;
13     }
14 } }
15 return c;
16 }
17 };

```

矩阵的输入、输出。

```

1 void read(Mtx& mtx) {
2     _fora(i, 1, mtx.p) _fora(j, 1, mtx.q)
3         mtx.m[i][j] = (MOD + rr()) % MOD;
4 }
5 void pr(Mtx mtx) {
6     _fora(i, 1, mtx.p) {
7         printf("%lld",mtx.m[i][1]);
8         _fora(j, 2, mtx.q)
9             printf(" %lld",mtx.m[i][j]);
10        putchar('\n');
11    }
12 }

```

1.7 快速幂

</> 代码 1.7: /上号/快速幂.cpp

```

1 ll qpow(ll a, ll b, ll p) {
2     ll rst = 1 % p;
3     for(; b > 0; b >>= 1, a = a * a % p)
4         if(b & 1) rst = a * rst % p;

```

```

5     return rst;
6 }

```

</> 代码 1.8: /上号/矩阵快速幂.cpp

```

1 struct QMtx {
2     ll m[5][5], p;
3     QMtx(ll p) : p(p) {
4         memset(m, 0, sizeof(m));
5     }
6     QMtx operator * (QMtx& mtx) {
7         QMtx c(p);
8         _fora(i, 1, p) { _fora(k, 1, p) {
9             ll t = m[i][k];
10            _fora(j, 1, p) {
11                c.m[i][j] += t * mtx.m[k][j];
12                c.m[i][j] %= MOD;
13            }
14        } }
15        return c;
16    }
17 };
18 QMtx base(ll p) {
19     QMtx rst(p);
20     _fora(i, 1, p)
21         rst.m[i][i] = 1;
22     return rst;
23 }
24 QMtx operator ^ (QMtx m, ll n) {
25     QMtx rst = base(3);

```

```

26     for(; n > 0; n >>= 1, m = m * m)
27         if(n & 1) rst = m * rst;
28     return rst;
29 }

```

1.8 快速排序

</> 代码 1.9: /上号/快速排序.cpp

```

1 void quick_sort(ll* nn, ll l, ll r) {
2     if(l >= r) return;
3     ll i = l, j = r;
4     ll x = nn[(l+r)/2];
5     while (i <= j) {
6         while (nn[j] > x) j--;
7         while (nn[i] < x) i++;
8         if(i <= j) swap(nn[i++], nn[j--]);
9     }
10    quick_sort(nn, l, j); quick_sort(nn, i, r);
11 }

```

1.9 第 k 大数

</> 代码 1.10: /上号/第 k 大数.cpp

```

1 ll q_sort(ll* nn, ll l, ll r) {
2     ll i = l, j = r, x = nn[(l+r)/2];
3     while (i <= j) {
4         while (nn[j] > x) j--;

```

```

5         while (nn[i] < x) i++;
6         if (i <= j) swap(nn[i++], nn[j--]);
7     } // l <= j <= i <= r
8     if (k <= j) return q_sort(l, j);
9     else if (k >= i) return q_sort(i, r);
10    else return nn[k + 1];
11 }

```

1.10 归并排序求逆序对

</> 代码 1.11: /上号/逆序对.cpp

```

1 ll aa[MAXN], tt[MAXN], ans = 0;
2 void merge_sort(int l, int r) {
3     if (r - l <= 1) return;
4     int mid = (l + r) >> 1;
5     merge_sort(l, mid); merge_sort(mid, r);
6     int p = l, q = mid, s = 1;
7     while (s < r) {
8         if (p >= mid || (q < r && aa[p] > aa[q]))
9             tt[s++] = aa[q++], ans += mid - p;
10        else
11            tt[s++] = aa[p++];
12    }
13    _fora (i, l, r - 1)
14        aa[i] = tt[i];
15 }

```

第二章 数学

2.1 GCD 和 LCM

</> 代码 2.1: /数学/gcdlcm.cpp

```
1 int gcd(int a, int b)
2     return a ? gcd(b % a, a) : b;
3 int lcm(int a, int b)
4     return a / gcd(b % a, a) * b;
5 int gcd(int a, int b)
6     while(b) { int t=a%b; a=b; b=t; } return a;
```

2.2 EXGCD

对于方程

$$ax + by = \gcd(a, b)$$

可通过 exgcd 求出一个整数解。

</> 代码 2.2: /数学/exgcd.cpp

```
1 void exgcd(int a, int b, int& x, int& y) {
2     if(!b) { y=0; x=1; return; /* gcd = a */ }
3     exgcd(b, a%b, y, x); y -= a/b*x;
```

4 }

方程 $ax + by = c$ 有解的充要条件是 $\gcd(a, b) \mid c$ 。

</> 代码 2.3: /数学/liEu.cpp

```
1 bool liEu(int a, int b, int c, int& x, int& y) {
2     exgcd(a, b, x, y);
3     if(c % gcd != 0) return false;
4     int k = c / gcd;
5     x *= k, y *= k;
6     return true;
7 }
```

2.3 数论函数

数论函数是 $\mathbb{N}^+ \rightarrow \mathbb{C}$ 的函数。

定义 常见函数 $\varepsilon(n) := [n = 1]$, $1(n) := 1$, $\text{Id}_k(n) := n^k$, $\sigma_k(n) = \sum_{d|n} d^k$ 。

设 $n > 1$ 的唯一分解式是

$$n = p_1^{k_1} p_2^{k_2} \cdots p_s^{k_s}$$

则 $\omega(n) = s$, $\Omega(n) = \sum k_i$, $\nu(n) = (-1)^{\omega(n)}$, $\lambda(n) = (-1)^{\Omega(n)}$ 。

Euler φ 函数:

$$\varphi(m) := \sum_{i=1}^m [\gcd(i, m) = 1] = m \prod_{p|n} \left(1 - \frac{1}{p}\right)$$

Mobius μ 函数:

$$\mu(n) = \begin{cases} 1, & n = 1 \\ 0, & \exists d > 1, d^2 \mid n \\ (-1)^{\omega(n)}, & \text{otherwise} \end{cases}$$

若数论函数 f 满足对 $\gcd(a, b)$ 有 $f(ab) = f(a)f(b)$, 则称为积性函数; 若有 $f(ab) = f(a) + f(b)$ 则称为加性函数。若 f, g 为积性函数, 则 $f(x^p), f^p(x), fg(x), f * g$ 都是积性函数。

积性函数有 $\varphi, \varepsilon, 1, \text{Id}_k, \sigma_k, \nu, \lambda$, 加性函数有 ω, Ω 。

性质 $\varphi(m)$ 的性质: $\varphi(p^k) = (p-1)p^{k-1}$ 。

$$\varphi(m) = m \prod_{p|n} \left(1 - \frac{1}{p}\right), \sum_{d|m} \varphi(d) = m$$

$\mu(m)$ 的性质: $\sum_{d|n} \mu(d) = [n=1] = \varepsilon$ 。

Femmat - Euler 定理: 当 $\gcd(a, m) = 1$ 时, 有

$$a^{\varphi(m)} \equiv 1 \pmod{m}$$

推广, 不要求互质, 当 $b \geq \varphi(m)$ 时有

$$a^b \equiv a^{b \bmod \varphi(m) + \varphi(m)}$$

2.4 乘法逆元

方程 $ax \equiv 1 \pmod{p}$ 有解的充要条件是 $\gcd(a, p) = 1$ 。

容易想到它与方程 $ax + py = c$ 等价, 于是可以利用 exgcd 求最小正解。

</> 代码 2.4: /数学/逆元/exgcd 法.cpp

```
1 ll inv(ll a, ll p) {
2     ll x, y;
3     exgcd(a, p, x, y);
4     return (x % p + p) % p;
5 }
```

仅当 p 为质数时, 由 Fermat 小定理知 $x \equiv a^{p-2} \pmod{p}$ 。

</> 代码 2.5: /数学/逆元/快速幂法.cpp

```
1 ll inv(ll a, ll p) {
2     return qpow(a, p - 2, p);
3 }
```

2.5 筛法

Eratosthenes 筛 复杂度 $O(n \log \log n)$ 。

</> 代码 2.6: /数学/筛法/Eratosthenes.cpp

```
1 bool notp[MAXN];
2 int prime[MAXN/10], cnt;
3 void Eratosthenes(int n) {
4     _fora (i, 2, n) { if (!notp[i]) {
5         prime[++cnt] = i;
6         int tn = n / i;
7         _fora (j, i, tn)
8             notp[i * j] = true;
9     } }
10 }
```

Euler 筛 复杂度 $O(n)$, 每个合数只会被筛一次。

</> 代码 2.7: /数学/筛法/Euler.cpp

```
1 bool notp[MAXN];
2 int prime[MAXN/10], cnt;
3 void Euler(int n){
4     _fora (i, 2, n) {
5         if (!notp[i]) prime[++cnt] = i;
```

```

6         int t = n / i;
7         _fora (j, 1, cnt) {
8             ll pj = prime[j];
9             if (pj > t) break;
10            notp[i * pj] = true;
11            if (i % pj == 0) break;
12        }
13    }
14 }

```

Euler 筛 + MobiusMu + EulerPhi 由于 Euler 筛过程中会把每一个数拆分成两个互质的因数，因此可以用来预处理积性函数和加性函数。如 Euler 筛 + $\mu(m)$ + $\phi(m)$ ，复杂度 $O(n)$ 。

</> 代码 2.8: /数学/筛法/Sieve.cpp

```

1  bool notp[MN];
2  int prime[MN/10], cnt;
3  int mu[MN], phi[MN];
4  void sieve(int n) {
5      phi[1] = mu[1] = 1;
6      _fora (i, 2, n) {
7          if (!notp[i]) {
8              prime[++cnt] = i;
9              phi[i] = i - 1, mu[i] = -1;
10         }
11         int t = n / i;
12         _fora (j, 1, cnt) {
13             int pj = prime[j], ti = i * pj;
14             if (pj > t) break;

```

```

15         notp[ti] = true;
16         if (i % pj == 0) {
17             phi[ti] = phi[i] * pj;
18             mu[ti] = 0;
19             break;
20         }
21         phi[ti] = phi[i] * (pj - 1);
22         mu[ti] = - mu[i];
23     }
24 }
25 }

```

2.6 素性测试

2.6.1 试除法

</> 代码 2.9: /数学/试除法.cpp

```

1  bool isprime(ll n) {
2      if (n < 3) return n == 2;
3      if (n & 1 == 0) return false;
4      ll sn = (ll) sqrt(n * 1.0);
5      for (ll i = 3; i <= sn; i += 2)
6          if (n % i == 0) return false;
7      return true;
8  }

```

2.6.2 Miller Rabbin

如果 $n \leq 2^{32}$, 那么 ppp 取 2, 7, 61; 如果 ppp 选择 2, 3, 7, 61, 24251, 那么 10^{16} 内只有唯一的例外。如果莫名 WA 了, 就多取点素数吧。

</> 代码 2.10: /数学/Miller_Rabbin.cpp

```
1 bool Miller_Rabbin(ll n) {
2     if (n < 3) return n == 2;
3     if (n & 1 == 0) return false;
4     int a = n - 1, b = 0;
5     while (1 - a & 1) a /= 2, ++b;
6     int ppp[10] = {2, 7, 61};
7     _fora (i, 0, 2) {
8         ll x = ppp[i], j;
9         if (n == x) return true;
10        ll v = qpow(x, a, n);
11        if (v == 1 || v == n-1) continue;
12        _fora (j, 0, b - 1) {
13            v = v * v % n;
14            if (v == n - 1) break;
15        }
16        if (j >= b) return false;
17    } return true;
18 }
```

2.6.3 排列组合

2.6.4 Lucas 定理

当 n, m 很大而 p 较小时, 有

$$\binom{n}{m} \bmod p = \binom{\lfloor n/p \rfloor}{\lfloor m/p \rfloor} \cdot \binom{n \bmod p}{m \bmod p} \bmod p$$

</> 代码 2.11: /数学/Lucas.cpp

```
1 ll Lucas(ll n, ll m, int p){
2     if (!m) return 1;
3     return Lucas(n/p, m/p, p) * comb(n%p, m%p, p) % p;
4 }
```

2.7 约瑟夫 Josephus 问题

对 n 个人进行标号 $0, \dots, n-1$, 顺时针站一圈。从 0 号开始, 每一次从当前的人继续顺时针数 k 个, 然后让这个人出局, 如此反复。

设最后剩下的人的编号为 $J(n, k)$, 有递推式

$$J(n+1, k) = (J(n, k) + k) \bmod (n+1)$$

踢出第一个人 k 后, 剩下就转化为 $J(n, k)$ 的情景, 还原编号只需增加相对位移 k 。

</> 代码 2.12: /数学/Josephus.cpp

```
1 int Josephus(int n, int k) {
2     int rst = 0;
3     _fora (i, 1, n)
4         rst = (rst + k) % i;
5     return rst;
}
```

2.8 中国剩余定理

若 n_i 中任意两个互质, 求方程组的解

$$x \equiv a_i \pmod{n_i}, \quad i \in \{1, 2, \dots, k\}$$

</> 代码 2.13: /数学/China.cpp

```
1 ll China(ll* aa, ll* nn) {
2     ll prod = 1;
3     ll rst = 0;
4     _fora (i, 1, n)
5         prod *= nn[i];
6     _fora (i, 1, n) {
7         ll m = prod / nn[i];
8         rst += aa[i] * m * inv(m, nn[i]);
9         rst %= prod;
10    } return rst;
11 }
```

2.9 Mobius 反演

对于数论函数 $f(x)$ 和 $g(x)$, 定义 $h(x)$ 为

$$h(x) = \sum_{d|x} f(d)g\left(\frac{x}{d}\right) = \sum_{ab=x} f(a)g(b)$$

为其 Dirichlet 卷积, 简记为 $h = f * g$ 。

Dirichlet 卷积满足交换律、结合律、分配律等, 其上有单位元 ε 和逆元, 换言之其与数论函数构成了一个整环。

对于数论函数 $f(n)$, 有 $F = f * 1 \Rightarrow f = F * \mu$ 。

即对于数论函数 $f(n)$, 有

$$F(n) = \sum_{d|n} f(d) \Rightarrow f(n) = \sum_{d|n} \mu(d) f\left(\frac{n}{d}\right)$$

以及

$$F(n) = \sum_{n|d} f(d) \Rightarrow f(n) = \sum_{n|d} \mu\left(\frac{d}{n}\right) F(d)$$

常见的有

$$\begin{aligned} 1 &= \varepsilon * 1, & \sigma_k &= \text{Id}_k * 1, & d &= 1 * 1 \\ \text{Id} &= \varphi * 1, & 1_{\text{Sq}} &= \lambda * 1, & (d * 1)^2 &= d^3 * 1 \end{aligned}$$

整除分块 令 $A = \{[n/d] \mid 1 \leq d \leq n\}$, 则 $|A| \leq 2\sqrt{n}$, 即整除求和时可以做到 \sqrt{n} 的速度。

</> 代码 2.14: /数学/整除分块/01.cpp

```
1 for (ll l = 1, r; l <= n; l = r + 1) {
2     r = n / (n / l);
3     ll t = (r - l + 1);
4     // ..
5 }
```

假如求和的是 $\sum_{i=1}^{\min(n,m)} \left\lfloor \frac{n}{i} \right\rfloor \left\lfloor \frac{m}{i} \right\rfloor$, 则可以用二维分块的技巧

</> 代码 2.15: /数学/整除分块/02.cpp

```
1 for (ll l = 1, r; l <= n; l = r + 1) {
2     r = min(n / (n / l), m / (m / l));
3     ll t = (r - l + 1);
4     // ..
```

2.10 博弈

下面都是石子游戏, 轮流取走物品。方便起见, 称场上 n 堆石子 a_1, \dots, a_n 为局势。先手必输的局势称为奇异局势

2.10.1 Nim 博弈

有 n 堆分别有 a_i 个物品, 两人轮流取走任意一堆的任意个物品, 不能不取, 最后取光者获胜。奇异局势判定

$$a_1 \oplus \dots \oplus a_n = 0$$

2.10.2 Wythoff 博弈

两堆分别有 a, b 各物品, 两个人轮流从某一堆或同时从两堆中取同样多的物品, 不可不取, 最后取光者获胜。

</> 代码 2.16: /数学/博弈/Wythoff.cpp

```
1  const ld phi = 1.6180339887498948482045868343656;
2  int Wythoff(ll a, ll b) { // 判先手输赢
3      if (a > b) swap(a, b);
4      ll t = (ll) (b - a) * phi;
5      if (t == a) return false;
6      return true;
7  }
```

特点: 所有自然数都出现在奇异局势中, 不重不漏。

2.11 生成函数

普通生成函数 可以把序列 $\{a_n\}$ 映射到形式幂级数

$$a(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n + \dots$$

这样就可以通过对级数的研究得到序列的性质。记作 $f_n = [x^n]F(x)$ 。

生成函数的四则运算是显然的。其中 $F(x)G(x) = \{\sum_{i=0}^n f_i g_{n-i}\}$ 为序列 $\{f_n\}$ 与 $\{g_n\}$ 的卷积。

对生成函数移位可以乘除 x^m , 也可以对生成函数逐项求导、逐项积分。求生成函数的部分和, 即是 $(1-x)^{-1}F(x)$ 。

基础函数:

$$\begin{aligned}\frac{1}{1-cx} &= \sum_{k=0}^{\infty} c^k x^k \\ \frac{x}{1-x-x^2} &= \sum_{k=0}^{\infty} F_k x^k \\ e^{cx} &= \sum_{k=0}^{\infty} \frac{c^k x^k}{k!} \\ \ln(1+x) &= \sum_{k=1}^{\infty} \frac{(-1)^{k-1}}{k} x^k \\ -\ln(1-x) &= \sum_{k=1}^{\infty} \frac{1}{k} x^k\end{aligned}$$

五边形数定理:

$$\prod_{i=1}^{\infty} (1-x^i) = \sum_{k=0}^{\infty} (-1)^k x^{k(3k\pm 1)/2}$$

2.12 容斥原理

对于有限集 S , 它的子集 A, B, C 有

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |B \cap C| - |C \cap A| + |A \cap B \cap C|$$

对于 n 个集合的情况,

容斥原理: 设有限集 S 的 n 个子集 A_i , 则有

$$\left| \bigcup_{i=1}^n A_i \right| = \sum_{\emptyset \neq J \subseteq \{1, 2, \dots, n\}} (-1)^{|J|-1} \left| \bigcap_{j \in J} A_j \right|$$

Burnside 引理 设 A 和 B 为有限集合, $X = B^A$ 表示所有从 $A \rightarrow B$ 的映射. G 是 A 上的置换群, X/G 表示 G 作用在 X 上产生的所有等价类的集合 (若 X 中的两个映射经过 G 中的置换作用后相等, 则它们在同一等价类中), 则

$$|X/G| = \frac{1}{|G|} \sum_{g \in G} |X^g|$$

其中 $|S|$ 表示集合 S 中元素的个数, 且

$$X^g = \{x \mid x \in X, g(x) = x\}$$

Polya 定理 前置条件与 Burnside 引理相同

$$|X/G| = \frac{1}{|G|} \sum_{g \in G} |B|^{c(g)}$$

其中 $c(g)$ 表示置换 g 能拆分成不相交的循环置换的数量。

2.13 组和排列

2.13.1 数学常数

排列数 从 n 个不同元素中取 m 个不同元素按顺序排成的一列称作一个排列, 则排列种数为

$$A_n^m = n(n-1) \cdots (n-m+1) = \frac{n!}{(n-m)!}$$

若元素可重复取出, 则为 n^m 。

组和数 从 n 个不同元素中取 m 个不同元素组成的一组称作一个组合, 则组合种数为

$$C_n^m = \frac{n(n-1) \cdots (n-m+1)}{m!} = \frac{n!}{m!(n-m)!}$$

也记作 $\binom{n}{m}$ 。若元素可重复取出, 则为 $\binom{n+m-1}{m}$ 。

圆排列 将 n 个元素不分首尾的排成一圈称作一个圆排列, 则圆排列种数为 $(n-1)!$ 。

隔板法 即 n 个相同的球放进 k 个盒子 (要求盒子非空) 里的种数为 $\binom{n-1}{k-1}$, 即不定方程 $\sum x_m = n$ 的正整数解数。若允许盒子为空, 则为 $\binom{n+k-1}{k-1}$ 。

错排公式 每一个元素都不在自己的位置上的 n 元排列称为一个错排, 则错排种数为

$$D_n = n! \left(1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \cdots + \frac{(-1)^n}{n!} \right) = \left\lfloor \frac{n!}{e} + 0.5 \right\rfloor$$

其递推式为 $D_n = (n-1)(D_{n-1} + D_{n-2})$ 。

2.13.2 常见公式

组和数和式

$$\binom{n+1}{k} = \binom{n}{k} + \binom{n}{k-1}, \quad \binom{n}{k} = \frac{n}{k} \binom{n-1}{k-1}$$

$$\sum_{r=0}^k \binom{n+r-1}{r} = \binom{n+k}{k}, \quad \sum_{r=0}^n \binom{n}{r}^2 = \binom{2n}{n}$$

$$\binom{n}{i} \binom{i}{m} = \binom{n}{m} \binom{n-m}{i-m}, \quad \sum_{i=m}^n \binom{i}{a} = \binom{n+1}{a+1} - \binom{m}{a+1}$$

$$\sum_{i=0}^k \binom{n}{i} \binom{m}{k-i} = \binom{n+m}{k}, \quad \sum_{j=0}^k \binom{k}{j}^2 \binom{n+2k-j}{2k} = \binom{n+k}{k}^2$$

$$\sum_{r=0}^n \binom{dn}{dr} = \frac{1}{d} \sum_{r=1}^d \left(1 + e^{\frac{2\pi r i}{d}} \right)^{dn}$$

二项式反演 注意到

$$\sum_{k=0}^n (-1)^k \binom{n}{k} = [n=0]$$

于是有

$$f(n) = \sum_{k=0}^n \binom{n}{k} g(k) \Rightarrow g(n) = \sum_{k=0}^n (-1)^{n-k} \binom{n}{k} f(k)$$

第三章 图论

3.1 链式前项星

</> 代码 3.1: /图论/链式前项星.cpp

```
1  const int MN = 10005; int head[MN];
2  struct Edge { int too,nxt,len; } edge[MN*2];
3  void add(int frm, int too, int len) {
4      static int cnt = 0;
5      edge[++cnt] = { too, head[frm], len };
6      head[frm] = cnt;
7  }
8  void dfs(int x,int fa) {
9      _fore(i,x) if(edge[i].too != fa)
10         dfs(edge[i].too,x);
11 }
```

3.2 最短路

3.2.1 Dijkstra

权值必须是非负, 复杂度 $O(E \log E)$ 。

</> 代码 3.2: /图论/最短路/Dijkstra.cpp

```
1  int dis[MN];
2  struct Dis {
3      int dis, pos;
4      bool operator < (const Dis& x) const
5          { return x.dis < dis; }
6  };
7  void Dijkstra(int ss) {
8      memset(dis, 0x3f, sizeof(dis));
9      dis[ss] = 0;
10     priority_queue<Dis> pq;  pq.push({0,ss});
11     while (!pq.empty()) {
12         Dis td = pq.top();  pq.pop();
13         int d = td.dis, x = td.pos;
14         if (d != dis[x]) continue;
15         _fore (i, x) {
16             int y = edge[i].too, z = d + edge[i].len;
17             if (dis[y] > z)
18                 dis[y] = z, pq.push({dis[y],y});
19         }
20     }
21 }
```

3.2.2 Bellman-Ford

复杂度 $O(VE)$ 。

</> 代码 3.3: /图论/最短路/Bellman-Ford.cpp

```
1  int dis[MN];
2  void Bellman_Ford(int ss) {
```



```

3     memset(dis, 0x3f, sizeof(dis)); dis[ss] = 0;
4     _fora (iia, 1, n-1) { int flag = 1;
5         _fora (x, 1, n) { _fore (i, x) {
6             int y = edge[i].too, z = dis[x] + edge[i].len;
7             if(dis[y] > z)
8                 dis[y] = z, flag = 0;
9         } } if(flag) return;
10    }
11 }

```

3.2.3 Floyd

起始条件 $f(i, j) = \text{edge}(i, j)$, $f(i, i) = 0$ 。

</> 代码 3.4: /图论/最短路/Floyd.cpp

```

1 void Floyd() {
2     _fora (k, 1, n) { _fora (i, 1, n) {
3         if (i == k || f[i][k] == 0x3f3f3f3f)
4             continue;
5         _fora (j, 1, n)
6             f[i][j] = min(f[i][j], f[i][k] + f[k][j]);
7     } }
8 }

```

3.3 最近公共祖先 LCA

需要预处理 \log_2 ; 如果数据小, 直接莽 20。

</> 代码 3.5: 图论/LCA.cpp

```

1 int pa[MN][30], lgb[MN], dep[MN]; // 预处理 lgb
2 void LCA_dfs(int u, int fa) {
3     pa[u][0] = fa, dep[u] = dep[fa] + 1;
4     _fora (i, 1, lgb[dep[u]])
5         pa[u][i] = pa[pa[u][i-1]][i-1];
6     _fore (i, u) if (edge[i].too != fa)
7         lca_dfs(edge[i].too, u);
8 }
9 int LCA(int x, int y) {
10    if (dep[x] < dep[y]) swap(x, y);
11    while (dep[x] > dep[y])
12        x = pa[x][lgb[dep[x]-dep[y]]];
13    if (x == y) return x;
14    _forz (k, lgb[dep[x]] - 1, 0)
15        if (pa[x][k] != pa[y][k])
16            x = pa[x][k], y = pa[y][k];
17    return pa[x][0];
18 }

```

第四章 动态规划

4.1 背包

4.1.1 01 背包

给定体积为 v_i , 价值 w_i 的 N 个物品, 背包容积为 M , 每个物品只能取 1 个, 求最大价值。

</> 代码 4.1: /动态规划/01 背包.cpp

```
1 _fora(i, 1, n) _forz(j, m, v[i])
2     dp[j] = max(dp[j], dp[j-v[i]]+w[i]);
3 _fora(j, 1, m) ans = max(ans, dp[j]);
```

4.1.2 完全背包

给定体积为 v_i , 价值 w_i 的 n 个物品, 背包容积为 v , 每个物品任意取, 求最大价值。

</> 代码 4.2: /动态规划/完全背包.cpp

```
1 _fora(i, 1, n) _fora(j, v[i], m)
2     dp[j] = max(dp[j], dp[j-v[i]]+w[i]);
3 _fora(j, 1, m) ans = max(ans, dp[j]);
```

4.1.3 多重背包

给定体积为 v_i , 价值 w_i 的 N 个物品, 背包容积为 M , 每个物品有 c_i 个, 求最大价值。

如各种背包组合 (如洛谷 P1833 樱花), 通常把完全背包转为 99999 个 (适当调节) 多重背包, 再按 01 背包来。

</> 代码 4.3: /动态规划/多重背包.cpp

```
1 int tm=1,vv[ ],ww[ ];
2 _fora(i, 1, n) {
3     int tc = c[i];
4     for(int b = 1;b <= p;b <= 1,tc -= b, ++tm)
5         vv[tm] = v[i] * b, ww[tm] = w[i] * b;
6     vv[tm] = v[i] * tc, ww[tm] = w[i] * tc;
7     ++tm;
8 }
9 _fora(i, 1, n) _forz(j, m, v[i])
10     dp[j] = max(dp[j], dp[j-v[i]]+w[i]);
11 _fora(j, 1, m) ans = max(ans, dp[j]);
```

4.2 最长公共上升序列

给出 $1, 2, \dots, n$ 的两个排列 a 和 b , 求它们的最长公共子序列。

</> 代码 4.4: /动态规划/最长公共上升序列.cpp

```
1 int f[MN], ma[MN], b[MN], n, len = 0;
2 memset(f, 0x3f, sizeof(f)); f[0] = 0;
3 _fora(i, 1, n) ma[rr()] = i;
4 _fora(i, 1, n) b[i] = rr();
5 _fora(i, 1, n) {
```

```

6     int l = 0, r = len;
7     if (ma[b[i]] > f[len])
8         f[++len] = ma[b[i]];
9     else { while (l < r) {
10         int mid = (l + r) / 2;
11         if (f[mid] > ma[b[i]]) r = mid;
12         else l = mid+1;
13     } }
14     f[l] = min(ma[b[i]], f[l]);
15 }

```

4.3 数字计数

试计算在区间 1 到 n 的所有整数中，数码 $x(0 \leq x \leq 9)$ 共出现了多少次？

</> 代码 4.5: /动态规划/数字计数/01.cpp

```

1  int addup(int n, int x) {
2      int ans = 0, m = 1;
3      while (m <= n){
4          int a = n / (m * 10), b = n / m % 10, c = n % m;
5          if (x > 0){
6              if (b > x) ans += m;
7              else if (b == x) ans += c + 1;
8          } else if (b == 0) {
9              ans += c + 1 - m;
10         } ans += a * m, m *= 10;
11     } return ans;
12 }

```

试计算在区间 1 到 n 的所有整数中，出现数码 $x(0 \leq x \leq 9)$ 的数字有多少？

</> 代码 4.6: /动态规划/数字计数/02.cpp

```

1  // 预计算
2  ll dp[20], m = 1;
3  _fora (i, 1, 9) {
4      dp[i] = dp[i-1] * 9 + m;
5      m *= 10;
6  }
7  ll addup(ll n, ll x) {
8      ll m = 1, i = 0;
9      while (m <= n) i++, m *= 10;
10     ll ans = 0, t;
11     while (m) {
12         t = n / m, n = n % m;
13         ans += t * dp[i];
14         if (t == x) {
15             ans += n + 1;
16             break;
17         } else if (t > x) {
18             ans += - dp[i] + m;
19             i--, m /= 10;
20         } return ans;
21     }

```

第五章 数据结构

5.1 链表

</> 代码 5.1: /数据结构/链表.cpp

```
1 struct Node { int val; Node *prev, *next; };
2 struct List {
3     Node *head, *tail; int len;
4     List() {
5         head = new Node(); tail = new Node();
6         head->next = tail; tail->prev = head;
7         len = 0;
8     } // 在节点后 p 后插入值 v
9     void insert(Node *p, int v) {
10         Node *q = new Node(); q->val = v;
11         p->next->prev = q; q->next = p->next;
12         p->next = q; q->prev = p;
13         len++;
14     } // 删除节点 p
15     void erase(Node *p) {
16         p->prev->next = p->next;
17         p->next->prev = p->prev;
```

```
18         delete p; len--;
19     } // 清空链表
20     ~List() {
21         while(head != tail) {
22             head = head->next;
23             delete head->prev;
24         } delete tail; len = 0;
25     }
26 };
```

链表的遍历。

```
1 for(Node *p=l->head->next; p!=l->tail; p=p->next)
2     //正序遍历
3 for(Node *p=l->tail->prev; p!=l->head; p=p->prev)
4     //逆序遍历
```

5.2 滑动窗口

</> 代码 5.2: /数据结构/滑动窗口.cpp

```
1 ll h = 1, t = 0;
2 _fora (i, 1, n) {
3     while (h <= t && i + 1 - k > qq[h])
4         h++;
5     while (h <= t && aa[i] <= aa[qq[t]]) // aa[i] >= aa[qq[t]]
6         t--;
7     qq[++t] = i;
8     if (i >= k) {
```

```

9         printf("%lld ", aa[qq[h]]);
10    }
11 }

```

5.3 树状数组

树状数组可以维护数组 a 实现 (1) 将某个数加上 x 。(2) 求前缀和。

</> 代码 5.3: /数据结构/树状数组/01.cpp

```

1  ll aa[MN], cc[MN], n;
2  void build() {
3      _fora (i,1,n) {
4          cc[i] += aa[i];
5          ll j = i + (i&(-i));
6          if (j <= n)
7              cc[j] += cc[i];
8      }
9  }
10 ll ask(ll *cc, ll x) {
11     ll sum = 0;
12     while (x >= 1)
13         sum += cc[x], x -= x & (-x);
14     return sum;
15 }
16 void add(ll *cc, ll x, ll k) {
17     while (x <= n)
18         cc[x] += k, x += x & (-x);
19 }

```

区间加 & 单点查询 维护数组 a 的额外差分数组 b , 那么 a 的区间加就被转化为 b 的单点增加, 且 a 单点查询就被转化为 b 的区间查询。

</> 代码 5.4: /数据结构/树状数组/02.cpp

```

1  void badd(ll l, ll r, ll k) {
2      add(bb, l, k);
3      add(bb, r+1, -k);
4  }
5  ll bask(ll x) {
6      return ask(bb, x) + aa[x];
7  }

```

区间加 & 区间求和 维护数组 a 的额外差分数组 b , 当我们对 a 的前缀 r 求和时有

$$\sum_{i=1}^r \sum_{j=1}^i b_j = \sum_{i=1}^r b_i (r - i + 1) = (r + 1) \sum_{i=1}^r b_i - \sum_{i=1}^r b_i i$$

因此还需要两个树状数组来维护 $\sum b_i$ 和 $\sum b_i i$ 。查询前缀和 cask 。

</> 代码 5.5: /数据结构/树状数组/03.cpp

```

1  ll bb1[MN], bb2[MN];
2  void cadd(ll l, ll r, ll k) {
3      add(bb1, l, k);
4      add(bb1, r+1, -k);
5      add(bb2, l, l*k);
6      add(bb2, r+1, -(r+1)*k);
7  }
8  ll cask(ll x) {
9      return (x+1) * ask(bb1, x) + ask(cc,x) - ask(bb2,x);
10 }

```

5.4 ST 表

需要预处理 \log_2 。令 $st(i, j)$ 表示区间 $[i, i + 2^j - 1]$ 的最大值，显然 $ST(i, 0) = a_i$ 。状态转移方程

$$ST(i, j + 1) = \max(f(i, j), f(i + 2^j, j))$$

</> 代码 5.6: /数据结构/ST 表/01.cpp

```
1  _fora (j, 0, lg2n - 1) {
2      ll tj = 1 << j;
3      ll ti = n - (1<<(j+1)) + 1;
4      _fora (i, 1, ti)
5          ST[i][j + 1] = max(ST[i][j], ST[i + tj][j]);
6  }
```

对于 RMQ 问题，记 $s = \lfloor \log_2(r - l + 1) \rfloor$ ，我们总是可以用两个区间 $[l, l + 2^s - 1]$ 和 $[r - 2^s + 1, r]$ 来覆盖所查询区间。

</> 代码 5.7: /数据结构/ST 表/02.cpp

```
1  ll p = lg2[y - x + 1], q = y - (1<<s) + 1;
2  return max(ST[x][p], ST[q][p]);
```

5.5 线段树

单点修改，区间查询。

</> 代码 5.8: /数据结构/线段树/01.cpp

```
1  ll aa[MAXN], pl[MAXN*4], pr[MAXN*4], val[MAXN*4];
2  void build(ll l, ll r, ll p = 1) {
3      pl[p] = l, pr[p] = r;
```

```
4      if (l == r) {
5          val[p] = aa[l]; return;
6      }
7      ll mid = (l + r) / 2;
8      build(l, mid, p * 2);
9      build(mid + 1, r, p * 2 + 1);
10     val[p] = val[p * 2] + val[p * 2 + 1];
11 }
12 void change(ll x, ll v, ll p = 1) {
13     if (pl[p] == pr[p]) {
14         val[p] = val[p] + v; return;
15     }
16     ll mid = (pl[p] + pr[p]) / 2;
17     if (x <= mid) change(x, v, p * 2);
18     else change(x, v, p * 2 + 1);
19     val[p] = val[p * 2] + val[p * 2 + 1];
20 }
21 ll ask(ll l, ll r, ll p = 1) {
22     if (l <= pl[p] && r >= pr[p]) {
23         return val[p];
24     }
25     ll mid = (pl[p] + pr[p]) / 2, v = 0;
26     if (l <= mid) v += ask(l, r, p * 2);
27     if (r > mid) v += ask(l, r, p * 2 + 1);
28     return v;
29 }
```

区间修改，区间查询。

</> 代码 5.9: /数据结构/线段树/02.cpp

```

1  ll aa[MAXN], pl[MAXN*4], pr[MAXN*4], sum[MAXN*4], add[MAXN
   *4];
2  void build(ll l, ll r, ll p = 1) {
3      pl[p] = l, pr[p] = r;
4      if (l == r) {
5          sum[p] = aa[l]; return;
6      }
7      ll mid = (l + r) / 2;
8      build(l, mid, p * 2);
9      build(mid + 1, r, p * 2 + 1);
10     sum[p] = sum[p * 2] + sum[p * 2 + 1];
11 }
12 void spread(ll p) {
13     if (add[p]) {
14         sum[p * 2] += add[p] * (pr[p * 2] - pl[p * 2] + 1);
15         sum[p * 2 + 1] += add[p] * (pr[p*2+1] - pl[p*2+1] +
16         1);
17         add[p * 2] += add[p];
18         add[p * 2 + 1] += add[p];
19         add[p] = 0;
20     }
21 void change(ll l, ll r, ll v, ll p = 1) {
22     if (l <= pl[p] && r >= pr[p]) {
23         sum[p] += v * (pr[p] - pl[p] + 1);
24         add[p] += v; return;
25     } spread(p);
26     ll mid = (pl[p] + pr[p]) / 2;
27     if (l <= mid) change(l, r, v, p * 2);

```

```

28     if (r > mid) change(l, r, v, p * 2 + 1);
29     sum[p] = sum[p * 2] + sum[p * 2 + 1];
30 }
31 ll ask(ll l, ll r, ll p = 1) {
32     if (l <= pl[p] && r >= pr[p]) {
33         return sum[p];
34     } spread(p);
35     ll mid = (pl[p] + pr[p]) / 2, v = 0;
36     if (l <= mid) v += ask(l, r, p * 2);
37     if (r > mid) v += ask(l, r, p * 2 + 1);
38     return v;
39 }

```

第六章 字符串

6.1 KMP

前缀函数 对于长为 n 的字符串 s , 定义每个位置的前缀函数 $\pi(i)$, 值为 $s(0, i)$ 的真后缀与 s 的真前缀中相等的最长的长度。

设最长的长度为 $j_1 = \pi(i)$, 如何找到其次长 j_2 ?

注意到后缀 j_1 位与前缀 j_1 位完全相同, 故 j_2 为前缀 j_1 中相等真前缀与真后缀中最长的, 即

$$j_{n+1} = \pi(j_n - 1)$$

</> 代码 6.1: /字符串/KMP/01.cpp

```
1 ll pi[MN];
2 void pre_kmp(char* s, ll lens) {
3     _fora(i, 1, lens - 1) {
4         ll j = pi[i-1];
5         while(j && s[i] != s[j])
6             j = pi[j-1];
7         pi[i] = j + (s[i] == s[j]);
8     }
9 }
```

Knuth - Morris - Pratt 给定一个文本 t 和一个字符串 s (模式串), 尝试找到 s 在 t 中所有出现。

构造字符串 $s + * + t$, 其中 $*$ 为不出现在两个字符串中的特殊字符, 此时字符串 t 的前缀恰为 s , $\pi(i)$ 的意义为 s 在此处的出现长度。

当 $\pi(i) = |s|$ 时, s 在此处完全出现。

当字符串已经合并时, 直接计算 $\pi(i)$ 函数即可, 字符串出现位置是 $i - 2|s|$ 。

</> 代码 6.2: /字符串/KMP/02.cpp

```
1 void kmp(char* s, ll lens, char* t, ll lent) {
2     pre_kmp(s, lens);
3     ll p = 0;
4     _fora(i, 0, lent-1) {
5         ll j = p;
6         while(j && t[i] != s[j])
7             j = pi[j-1];
8         p = j + (t[i] == s[j]);
9         if(p == lens)
10             // 出现起始 i-lens+2
11     }
12 }
```

EXKMP 对于长为 n 的字符串 s , 定义每个位置的后缀函数 $z(i)$, 值为 $s(i, n-1)$ 的真前缀与 s 真前缀中相等的最长的长度。

</> 代码 6.3: /字符串/KMP/03.cpp

```
1 void pre_exkmp(char* s, ll lens) {
2     ll l = 0, r = 0;
3     _fora(i, 1, lens - 1) {
4         if(i <= r) zz[i] = min(r - i + 1, zz[i - 1]);
5         ll tz = zz[i];
```



```

6         while(i + tz < lens && s[tz] == s[i+tz])
7             tz++;
8         zz[i] = tz;
9         if(i + zz[i] - 1 > r) l = i, r = i + zz[i] - 1;
10    }
11    zz[0] = lens;
12 }
13
14 void exkmp(char* s, ll lens, char* t, ll lent) {
15     pre_exkmp(t, lent);
16     ll l = -1, r = -1;
17     ext[0] = 0;
18     _fora(i, 0, lens - 1) {
19         if(i <= r) ext[i] = min(r - i + 1, zz[i - 1]);
20         ll tz = ext[i];
21         while(i + tz < lens && t[tz] == s[i+tz]) tz++;
22         ext[i] = tz;
23         if(i + ext[i] - 1 > r) l = i, r = i + ext[i] - 1;
24     }
25 }

```

6.2 Manacher

</> 代码 6.4: /字符串/Manacher/01.cpp

```

1 char str[MN], st[MN * 2];
2 ll mp[MN * 2];

```

```

3 void Manacher(char *s, int len) {
4     st[0] = st[1] = '#';
5     int l = 2;
6     _fora(i, 0, len - 1)
7         st[l++] = s[i], st[l++] = '#';
8     st[l] = 0;
9     int mx = 0, id = 0;
10    _fora(i, 0, l - 1) {
11        mp[i] = mx > i ? min(mp[2 * id - i], mx - i) : 1;
12        while (st[i + mp[i]] == st[i - mp[i]])
13            mp[i]++;
14        if (i + mp[i] > mx)
15            mx = i + mp[i], id = i;
16    }
17 }
18
19 int main() {
20     scanf("%s", str);
21     int len = strlen(str);
22     Manacher(str, len);
23     ll ans = 0;
24     _fora(i, 0, 2 * len + 1)
25         ans = max(ans, mp[i] - 1);
26     printf("%lld", ans);
27     return 0;
28 }

```
