fshp971

ACM常用算法模板

SCUT - 2018.10.

目录

1. 数学

1.1.欧几里得算法

```
typedef long long LL;
int gcd(LL a,LL b) { return b ? gcd(b, a%b) : a; }

/*
 * 调用时需保证 a >= b >= 0, 且提供的 g, x, y 需为实际存在的变量
 * g = gcd(a,b), 计算结果满足 x*a + y*b = g = gcd(a,b)
 * 需要注意 (x, y) 并不唯一
 */
void ex_gcd(LL a,LL b,LL &g,LL &x,LL &y) {
    if(b==0){ g = a; x = 1; y = 0; }
    else{ ex_gcd(b,a%b,g,y,x); y -= (a/b)*x; }
}
```

1.2.乘法逆元

```
// 若 a*b ≡ 1(mod p), 则称 b 为 a mod p 时的乘法逆元, 记作a^(-1)
// 乘法逆元存在当且仅当 a 与 p 互质

// 用扩展欧几里得求乘法逆元
// 若逆元不存在, 返回 -1
LL inv(LL a, LL p) {
    LL g,x,y;
    ex_gcd(a,p,g,x,y);
    return g==1 ? (x+p)%p : -1;
}

// 用费马小定理求乘法逆元
// 费马小定理: (1). 若p是一个质数, 则有 a^p ≡ a(mod p)
//             (2). 当a不是p的倍数时有: a^(p-1) ≡ 1(mod p)
// 故当a不是p的倍数时, a^(p-2) 即为a的逆元
LL inv2(LL a, LL p) {
    return pow_mod(a, p-2, p);
}

// 线性求逆元
// 原理: (P/i)*i + (P%i) = 0(mod P) -> i^(-1) = -(P/i) * (P%i)^(-1) (mod P)
LL inv[maxn + 5], P;
LL GetInv(int n) {
    inv[1] = 1;
    for(int i = 1; i <= n; ++i)
        inv[i] = (P-(P/i)) * inv[P%i] % P;
}
```

1.3.中国剩余定理解同余方程组 x ≡ ai (mod mi)

```
/*
 * 中国剩余定理
 * Chinese_REmainder_Theorem
 * 考虑同余方程组 x ≡ ai (mod mi)
 * 其中, m1, m2, ... mn 两两互质, 设 M = Pai(mi), Mi = M / mi
 * 则该方程组在 mod M 的剩余系中有唯一解:
 *      x0 = sigma(ai * Mi * (Mi)^-1)
 *  其中, (Mi)^-1 为 Mi 在 mod mi 下的乘法逆元
 *  而对于一般情况, x = x0 + kM , k为整数
 */
typedef long long LL;

LL CRT(int len, LL *ai, LL *mi) {
    LL M = 1, res = 0, x,y,g;
    for(int i=0;i<len;++i) M *= mi[i];
    for(int i=0;i<len;++i) {
        LL tmp = M / mi[i];
        ex_gcd(tmp, mi[i], g, x, y);
        x = (x+mi[i]) % mi[i];
        //注意此处可能会溢出
        res = (res + ai[i] * tmp * x) % M;
    }
    return res;
}
```

1.4.解方程 a^x ≡ b(mod p)

```
/*
 * 解方程 a^x ≡ b(mod p) , 其中 p 为质数
 * 由费马小定理可知, 只需检查 x = 0, 1, 2, ... , p-1 即可
 * 本例使用 shank 大步小步算法 (Shank's Baby-Step-Giant-Step Algorithm) 求解, 无解返回-1
 */
typedef long long LL;
LL log_mod(LL a, LL b, LL p) {
    //此处使用 int 来存储 m. 事实上若 m 过大, 则必定会超时
    int m = sqrt(n)+1;
    //pow_mod() 为快速幂
    LL vv = inv(pow_mod(a, m, p), p);
    LL tmp = 1;
    map<LL, int> shank;
    for(int i=0;i<m;++i) {
        if(!shank.count(tmp)) shank[tmp] = i;
        tmp = tmp*a % p;
    }
    for(int i=0;i<m;++i) {
        if(shank.count(b)) return (LL)i*m + shank[b];
        b = b*vv % p;
    }
    return -1;
}
```

1.5.大数乘法转化为对数加法

UVa-10883：计算

$$\sum_{i=0}^{n-1} \frac{C_{n-1}^{i} * a_i}{2^{n-1}}$$

将每项转化为：

$$\log(n-1)! + \log|a_i| - \log(n-1-i)! - \log i! - (n-1)\log 2$$

```
/*
 * 当遇到大数运算，且结果可为浮点数时，可以考虑转化为对数来运算
 * 例：UVa-10883
 * 题目要求：计算 sigma( C(n-1, i) * ai / 2^(n-1) ), i from 0 to n-1
 * 将每一项转化为对数后有：log((n-1)!) + log(abs(ai)) - log((n-1-i)!) -log(i!) -
(n-1)*log(2).
 * 一定要注意 log(x) 中的 x 需严格为正，计算时要针对非正的情况另行判断.
 * 附代码
 */
#include<bits/stdc++.h>
using namespace std;

const int maxn = 50000;

int n;
double f[maxn+10], arr[maxn+10];

void run() {
    f[0] = 0;
    for(int i = 1; i <= maxn; ++i)
        f[i] = f[i-1] + log10(i);
}

double init() {
    cin>>n;
    scanf("%d", &n);
    for(int i = 0; i < n; ++i)
        scanf("%lf", arr+i);
    double res = 0;
    for(int i = 0; i <= n-1; ++i) {
        if(arr[i] == 0) continue;
        double tmp = f[n-1] + log10(fabs(arr[i])) - f[n-1-i] - f[i] - log10(2) *
(n-1);
        if(arr[i] > 0) res += pow(10, tmp);
        else res -= pow(10, tmp);
    }
    return res;
}

int main() {
    run();
```

```
40     int T;
41     scanf("%d", &T);
42     for(int t = 1; t <= T; ++t)
43         printf("Case #%d: %.3f\n",t,init());
44     return 0;
45 }
```

1.6.Berlekamp-Massey Algorithm

```
 1 // power by Metowolf
 2 // usage Sample:
 3 // >
 4 // 9
 5 // 1 2 3 4 5 6 7 8 9
 6 // <
 7 // 2 -1
 8 // which is: A_n = 2 * A_{n-1} - A_{n-2}
 9 #include <iostream>
10 #include <cstdio>
11 #include <vector>
12 #include <cmath>
13 using namespace std;
14
15 const double eps=1e-7;
16 const int maxn=1e5;
17
18 vector<double>ps[maxn+10];
19 int fail[maxn+10];
20 double x[maxn+10],delta[maxn+10];
21 int n,pn;
22
23 int main(){
24     while(~scanf("%d",&n)&&n){
25         pn=0;
26         for(int i=1;i<=n;i++)scanf("%lf",x+i);
27         for(int i=1;i<=n;i++){
28             double dt=-x[i];
29             for(int j=0;j<ps[pn].size();j++)dt+=x[i-j-1]*ps[pn][j];
30             delta[i]=dt;
31             if(fabs(dt)<=eps)continue;
32             fail[pn]=i;
33             if(!pn){ps[++pn].resize(1);continue;}
34             vector<double> &ls=ps[pn-1];
35             double k=-dt/delta[fail[pn-1]];
36             vector<double>cur;
37             cur.resize(i-fail[pn-1]-1);
38             cur.push_back(-k);
39             for(int j=0;j<ls.size();j++)cur.push_back(ls[j]*k);
40             if(cur.size()<ps[pn].size())cur.resize(ps[pn].size());
41             for(int j=0;j<ps[pn].size();j++)cur[j]+=ps[pn][j];
42             ps[++pn]=cur;
43         }
44         int len=(int)ps[pn].size();
```

```
45          for(int i=0;i<len;++i)
46              printf("%g%c",ps[pn][i]," \n"[i==len-1]);
47      }
48      return 0;
49  }
```

1.7.Meissel-Lehmer-Method

```
 1  //Meissel-Lehmer-Method
 2  //快速计算素数计数函数
 3  #include<bits/stdc++.h>
 4  using namespace std;
 5
 6  typedef long long LL;
 7
 8  const int N = 5e6+5; //N >= sqrt(n), 适当增大N可提升速度
 9  bool np[N];
10  int prime[N], pi[N];
11
12  int getprime() {
13      int cnt = 0;
14      np[0] = np[1] = true;
15      pi[0] = pi[1] = 0;
16      for(int i = 2; i < N; ++i) {
17          if(!np[i]) prime[++cnt] = i;
18          pi[i] = cnt;
19          for(int j = 1; j <= cnt && i * prime[j] < N; ++j) {
20              np[i * prime[j]] = true;
21              if(i % prime[j] == 0) break;
22          }
23      }
24      return cnt;
25  }
26  const int M = 7;
27  const int PM = 2 * 3 * 5 * 7 * 11 * 13 * 17;
28  int phi[PM + 1][M + 1], sz[M + 1];
29  void init() {
30      getprime();
31      sz[0] = 1;
32      for(int i = 0; i <= PM; ++i)  phi[i][0] = i;
33      for(int i = 1; i <= M; ++i) {
34          sz[i] = prime[i] * sz[i - 1];
35          for(int j = 1; j <= PM; ++j) {
36              phi[j][i] = phi[j][i - 1] - phi[j / prime[i]][i - 1];
37          }
38      }
39  }
40  int sqrt2(LL x) {
41      LL r = (LL)sqrt(x - 0.1);
42      while(r * r <= x)   ++r;
43      return int(r - 1);
44  }
45  int sqrt3(LL x) {
```

```
46      LL r = (LL)cbrt(x - 0.1);
47      while(r * r * r <= x)   ++r;
48      return int(r - 1);
49  }
50  LL getphi(LL x, int s) {
51      if(s == 0) return x;
52      if(s <= M) return phi[x%sz[s]][s] + (x/sz[s]) * phi[sz[s]][s];
53      if(x <= prime[s]*prime[s]) return pi[x] - s + 1;
54      if(x <= prime[s]*prime[s]*prime[s] && x < N) {
55          int s2x = pi[sqrt2(x)];
56          LL ans = pi[x] - (s2x + s - 2) * (s2x - s + 1) / 2;
57          for(int i = s + 1; i <= s2x; ++i) {
58              ans += pi[x / prime[i]];
59          }
60          return ans;
61      }
62      return getphi(x, s - 1) - getphi(x / prime[s], s - 1);
63  }
64  LL getpi(LL x) {
65      if(x < N) return pi[x];
66      LL ans = getphi(x, pi[sqrt3(x)]) + pi[sqrt3(x)] - 1;
67      for(int i = pi[sqrt3(x)] + 1, ed = pi[sqrt2(x)]; i <= ed; ++i){
68          ans -= getpi(x / prime[i]) - i + 1;
69      }
70      return ans;
71  }
72  LL lehmer_pi(LL x) {
73      if(x < N)   return pi[x];
74      int a = (int)lehmer_pi(sqrt2(sqrt2(x)));
75      int b = (int)lehmer_pi(sqrt2(x));
76      int c = (int)lehmer_pi(sqrt3(x));
77      LL sum = getphi(x, a) + (LL)(b + a - 2) * (b - a + 1) / 2;
78      for (int i = a + 1; i <= b; i++) {
79          LL w = x / prime[i];
80          sum -= lehmer_pi(w);
81          if (i > c) continue;
82          LL lim = lehmer_pi(sqrt2(w));
83          for (int j = i; j <= lim; j++) {
84              sum -= lehmer_pi(w / prime[j]) - (j - 1);
85          }
86      }
87      return sum;
88  }
89
90  int main() {
91      init();
92      LL n;
93      while(~scanf("%lld", &n)) {
94          printf("%lld\n", lehmer_pi(n));
95      }
96      return 0;
97  }
```

1.8.矩阵

```cpp
typedef long long LL;
const int maxn = 10;

struct Matrix {
    int row, col;
    LL mat[maxn+5][maxn+5];
    // 如果懒得写自定义初始化，那到时记得要手动初始化mat[][]
    Matrix(): row(0), col(0) { memset(mat, 0, sizeof(mat)); }
    void clear(int _row = 0, int _col = 0) {
        row = _row, col = _col;
        memset(mat, 0, sizeof(mat));
    }
    Matrix operator*(const Matrix &a) {
        Matrix res;
        res.row = row, res.col = a.col;
        for(int i=0; i<row; ++i) {
            for(int k=0; k<col; ++k) {
                if(mat[i][k] == 0) continue;
                for(int j=0; j<a.col; ++j)
                    //可能要取模
                    res.mat[i][j] += mat[i][k]*a.mat[k][j];
            }
        }
        return res;
    }
    void output() {
        printf("row = %d, col = %d\n", row, col);
        for(int i=0; i<row; ++i) {
            for(int k=0; k<col; ++k)
                std::cout << mat[i][k] << " ";
            printf("\n");
        }
        printf("\n------------------------\n\n");
    }
};
```

1.9.高斯消元

```cpp
/*
 * n条方程n个变量，若有解则唯一，无法判断无解情况
 * 计算完成后A[i][n] (i = 0 ~ n-1) 共n个数即为方程的解
 */
double A[maxn + 5][maxn + 5];
void Gauss(int n) {
    for(int i = 0; i < n; ++i) {
        int r = i;
        for(int j = i + 1; j < n; ++j)
            if(fabs(A[r][i]) < fabs(A[j][i])) r = j;
        if(r != i)
            for(int j = 0; j <= n; ++j) swap(A[i][j], A[r][j]);
```

```
13          for(int j = i + 1; j < n; ++j) {
14              for(int k = n; k >= i; --k)
15                  A[j][k] -= A[i][k]/A[i][i] * A[j][i];
16          }
17      }
18      for(int i = n - 1; i >= 0; --i) {
19          for(int j = i + 1; j < n; ++j)
20              A[i][n] -= A[j][n] * A[i][j];
21          A[i][n] /= A[i][i];
22      }
23  }
24
25  /*
26   * 高斯消元化阶梯型求解的个数
27   * 这一般是在剩余系中才会问这样的问题，所以这里以(mod P)的剩余系来搞
28   * 化阶梯型的一般做法可以直接照着这个板修改得到
29   */
30  int A[maxn + 5][maxm + 5], Inv[maxP + 5]; //Inv[i]是i的逆元
31  void Gauss(int n, int m) { //共有n条方程，m个变元
32      int row = 0;
33      for(int i = 0; i < m && row < n; ++i) {
34          if(A[row][i] == 0) {
35              int r = row;
36              for(int j = row + 1; j < n; ++j)
37                  if(A[j][i] != 0) { r = j; break; }
38              if(r == row) continue;
39              for(int j = i; j <= m; ++j) swap(A[row][j], A[r][j]);
40          }
41          for(int j = row + 1; j < n; ++j)
42              for(int k = m; k >= i; --k)
43                  A[j][k] = (A[j][k] - A[row][k]*Inv[A[row][i]]*A[j][i] % P + P) % P;
44          ++row;
45      }
46      //此时若A[row][m]~A[n-1][m]中有非零数，则表示方程组无解
47      //否则该方程组的秩恰为row
48      while(row < n) {
49          if(A[row][m]) { puts("No answer"); break; } //无解
50          ++row;
51      }
52  }
```

1.10.计算a*b%c

```
1  // 计算 a*b%c
2  // 方法1: 自然溢出法，时间复杂度O(1)
3  typedef long long LL;
4  const long double eps = 1e-8;
5
6  LL multi_mod(LL a,LL b,LL c) {
7      a%=c, b%=c;
8      LL ans=a*b-(LL)((long double)a*b/c+eps)*c;
9      if(ans<0) ans+=c;
10     return ans;
```

```
11 }
12
13 // 方法2: 瞎搞, 类似快速幂, 时间复杂度O(logn)
14 LL MulMod(LL a, LL b, LL c) {
15     //a %= c, b %= c;
16     LL ret = 0;
17     while(b) {
18         if(b&1) ret = (ret+a)%c;
19         a = (a+a)%c, b >>= 1;
20     }
21     return ret;
22 }
```

1.11.Miller Rabin素性检测&Pollard-Rho大数分解

```
 1 // Miller Rabin 素性检测
 2 // Pollard-Rho 大数分解
 3 // 使用前请仔细阅读每一部分的说明
 4 #include <bits/stdc++.h>
 5 using namespace std;
 6 typedef long long LL;
 7
 8 // 一系列工具函数
 9 LL Gcd(LL a, LL b) { return b ? Gcd(b,a%b) : a; }
10 // 如果为保险可以使用较慢但没有精度问题的类快速幂方法
11 const long double eps = 1e-8;
12 LL MulMod(LL a,LL b,LL c) {
13     a%=c, b%=c;
14     LL ans=a*b-(LL)((long double)a*b/c+eps)*c;
15     if(ans<0) ans+=c;
16     return ans;
17 }
18 LL PowMod(LL a, LL b, LL p) {
19     LL r = 1 % p;
20     while(b) {
21         if(b&1) r = MulMod(r,a,p);
22         a = MulMod(a,a,p);
23         b >>= 1;
24     }
25     return r;
26 }
27
28 // Miller Rabin 素性检测(原理基于二次剩余)
29 // 返回为true表示n(极可能)是素数, 返回为false表示n一定不是素数
30 // 亦即, Miller Rabin的检测方法只有可能把合数判成素数, 但一定不会把素数判成合数
31 //
32 // MaxTry表示尝试次数, 每次尝试随机一个在1~(n-1)的整数a来判断(使用了rand()函数)
33 // 若n为合数, 可以证明每次尝试判断将n判错(判为素数)的概率不超过1/2
34 // 因此尝试MaxTry次后将合数判为素数的概率不超过1/(2^MaxTry)
35 //
36 // MaxTry是可以调节的参数, 不过一般不太影响速度, 调高可以提高正确性
37 const int MaxTry = 30;
38
```

```
bool MillerRabin(LL n) {
    if(n==2) return true;
    if(n==0 || n==1 || !(n&1)) return false;
    LL u = n-1, t = 0;
    while((u&1) == 0) u >>= 1, ++t;
    for(int s = 1; s <= MaxTry; ++s) {
        LL a = rand() % (n-1) + 1;
        LL x = PowMod(a, u, n);
        for(int i = 1; i <= t; ++i) {
            LL tx = MulMod(x, x, n);
            if(tx==1 && x!=1 && x!=n-1) return false;
            x = tx;
        }
        if(x!=1) return false;
    }
    return true;
}

// Pollard-Rho 大数分解
//
// 递归将n拆成tot素因子，并将结果保存在deco[1~tot]中
// 将重复的素因子也一并拆出，且结果无序，需自行排序
// 调用前tot记得清零
//
// 本质上就是一个有一定道理的随机乱搞
// 其中的(MulMod(x,x,n)+c)%n就是一个近似的随机函数
// 此非算法导论原版Pollard-Rho，但思想是一样的
// 此改良版速度非常快(测板题：hdu-4344)
//
// 由于此算法用的rand()+while()，所以有一定机率会一直找不到解最终TLE
// 所以如果TLE，在确保代码正确的请狂喜，可以尝试多交几次
// 或者可以尝试将MulMod()替换成纯整数运算的版本，防止精度问题
// 此模板至少能保证2^63以内的正确性
int tot;
LL deco[100];

void PollardRho(LL n) {
    if(n == 1) return;
    if(MillerRabin(n)) { deco[++tot] = n; return; }
    while(1) {
        LL c = rand() % (n-1) + 1;
        LL x = rand() % n;
        LL y = (MulMod(x,x,n)+c) % n;
        while(x != y) { // 关键终止条件，不然容易陷入死循环
            LL d = Gcd(abs(x-y), n);
            if(d != 1) {
                // 找出的d仅保证d!=1且d!=n，不一定是素因子，因此还需要继续递归分解
                PollardRho(d);
                PollardRho(n/d);
                return;
            }
            x = (MulMod(x,x,n)+c) % n;
            // 相当于y每次多走1步，即Floyd判圈(所以说，变量名别打错啊)
            y = (MulMod(y,y,n)+c) % n;
```

```
 93             y = (MulMod(y,y,n)+c) % n;
 94         }
 95     }
 96 }
 97
 98 int main() {
 99     srand(time(NULL));
100     // 调用例子，对n进行因数分解
101     LL n; scanf("%lld", &n);
102     tot = 0; // tot赋值为0
103     PollardRho(n);
104     sort(deco+1, deco+1+tot); // 自行排序保证有序，其中包括重因子
105     return 0;
106 }
```

1.12.傅立叶变换

- 快速傅立叶变换（FFT）

DFT 变换公式：

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk} \qquad k = 0,\ 1,\cdots,\ N-1$$

DFT 逆变换公式：

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{\frac{2\pi i}{N} kn} \qquad n = 0,\ 1,\cdots,\ N-1$$

提高精度的方法：1）预处理单位根；2）尽可能缩短多项式长度；3）不要使用2次DFT科技（虽然一般影响很小）。

```
 1 #include<bits/stdc++.h>
 2 //#include<complex>
 3 using namespace std;
 4
 5 // 若使用自己定义的复数速度貌似会略快
 6 typedef complex<double> Comp;
 7 const double PI = acos(-1.0);
 8
 9 const int Bin = 17, maxn = 1<<Bin;
10
11 // 手写复数，比STL复数快很多
12 struct Comp {
13     double re, im;
14     Comp(): re(0), im(0) {}
15     Comp(const double _re, const double _im) { re = _re; im = _im; }
16     Comp operator+(const Comp &a) { return Comp(re+a.re, im+a.im); }
17     Comp operator-(const Comp &a) { return Comp(re-a.re, im-a.im); }
18     Comp operator*(const Comp &a) { return Comp(re*a.re-im*a.im,
re*a.im+im*a.re); }
19     Comp operator*(const double &a) { return Comp(re*a, im*a); }
20     Comp operator/(const double &a) { return Comp(re/a, im/a); }
21     void operator*=(const Comp &a) { (*this) = (*this) * a; }
```

```
22      void operator+=(const Comp &a) { re += a.re; im += a.im; }
23      void operator*=(const double &a) { re*=a; im*=a; }
24      void operator/=(const double &a) { re/=a; im/=a; }
25      Comp conj() { return Comp(re, -im); }
26  };
27
28  // 预处理单位根，可以提高精度，但速度可能会变慢
29  vector<Comp> ww[Bin + 2];
30  void Init() {
31      for(int i = 1, l = 2; l <= maxn; ++i, l <<= 1) {
32          for(int k = 0; k < l/2; ++k)
33              ww[i].push_back( Comp( cos(PI*2*k/l), sin(PI*2*k/l) ) );
34      }
35  }
36
37  // type=1为DFT运算，type=-1为DFT逆运算
38  void FFT(Comp *a, int len, int type) {
39      int i, j, k, l, tt;
40      for(i = 1, j = len>>1; i < len-1; ++i) {
41          if(i < j) swap(a[i], a[j]);
42          k = len>>1;
43          while(j >= k) j -= k, k >>= 1;
44          j += k;
45      }
46      Comp var, u, v;
47      for(l = 2, tt = 1; l <= len; l <<= 1, ++tt) {
48          for(k = 0; k < len; k += l) {
49              for(i = k; i < k+l/2; ++i) {
50                  var = ww[tt][i-k];
51                  if(type == -1) var.im = -var.im;
52                  u = a[i], v = var * a[i+l/2];
53                  a[i] = u+v, a[i+l/2] = u-v;
54              }
55          }
56      }
57      /* 不预处理单位根时的做法
58      for(l = 2; l <= len; l <<= 1) {
59          step = Comp( cos(PI*2 / l * type), sin(PI*2 / l * type) );
60          for(k = 0; k < len; k += l) {
61              var = Comp(1,0);
62              for(i = k; i < k+l/2; ++i) {
63                  u = a[i], v = var * a[i+l/2];
64                  a[i] = u+v, a[i+l/2] = u-v;
65                  var *= step;
66              }
67          }
68      }
69      */
70      if(type == -1) for(i = 0; i < len; ++i) a[i] /= len;
71  }
72
73  int main() {
74      Init(); // 切记要预处理单位根
```

```
75      // n必须是2的幂，若不足则在后面补0
76      // n指的是FFT后算出的结果的多项式长度，故 n >= len(aa) + len(bb)
77      int n = 1<<10;
78      // aa[i]表示 x^i 项的系数，bb[i]类似
79      Comp aa[n+5], bb[n+5];
80      FFT(aa, n, 1);
81      FFT(bb, n, 1);
82      for(int i=0; i<n; ++i)
83          aa[i] *= bb[i];
84      FFT(aa, n, -1);
85      return 0;
86  }
```

- 2次DFT变换卷积

求实多项式 $A(x)$ 和 $B(x)$ 的卷积时，我们可以进行如下构造：

$$P(x) = A(x) + iB(x),\ \ Q(x) = A(x) - iB(x)$$

设 $F_p[k] = P(\omega^k)$，$F_q[k] = P(\omega^k)$，其中 ω 是 n 次单位根（即进行 DFT 变换后的第 k 项）。

那么可以证明：$\begin{cases} F_q[k] = conj(F_p[L-k]) & (k > 0) \\ F_q[0] = conj(F_p[0]) \end{cases}$

又有：$\begin{cases} DFT(A[k]) = \dfrac{F_p[k] + F_q[k]}{2} \\ DFT(B[k]) = i\dfrac{F_p[k] - F_q[k]}{2} \end{cases}$

因此我们只要进行一次 DFT 变换和一次 $IDFT$ 变换即可求出卷积。
这么做的唯一问题是精度相对较低，在进行多次实数域卷积后（例如实数分治FFT）会积累很大的误差，故一般只在计算整数多项式时使用该方法。

```
1  // 计算卷积a(x)*b(x)，结果存在c(x)当中
2  // a(x)*b(x)结果的最大非零项次数不超过(len-1)
3  Comp A[maxn + 5], B[maxn + 5];
4  void Conv(LL *a, LL *b, LL *c, int len) {
5      for(int i = 0; i < len; ++i) A[i] = Comp(a[i], b[i]);
6      FFT(A, len, 1);
7      for(int i = 0; i < len; ++i) {
8          int j = (len-i) & (len-1);
9          B[i] = (A[i]*A[i] - (A[j]*A[j]).conj()) * Comp(0,-0.25);
10     }
11     FFT(B, len, -1);
12     for(int i = 0; i < len; ++i) c[i] = LL(B[i].re+0.5);
13 }
```

- FFT任意数取模

计算 $A(x) * B(x)\ mod\ M$，由于 M 可能很大（例如取 $M = 10^9 + 7$），那么若直接进行卷积后在取模，中间变量就有可能达到 M^2L 级别，超出64位整数范围，即便使用128位整数，也可能产生比较大的误差，因此我们要考虑通过其他方法计算。

设 $M_0 = \lceil \sqrt{M} \rceil$，将两个整系数多项式的系数分别表示成 $c = kM_0 + b$ 的形式，其中 $k = \lfloor \dfrac{c}{M_0} \rfloor$，$b = c\ mod\ M_0$，那么我们就得到关于 k, b 的共计4个新的多项式。对着4个多项式两两进行卷积后再乘上相

应系数相加后取模，就可以得到 $A(x)*B(x)\ mod\ M$ 的结果。在这种情况下，卷积时候的中间变量被控制在 $M*L$ 范围内，不会超出长整型变量范围。通过前面所述的优化方法可以仅通过4次 FFT 计算。

```
//计算a(x)*b(x)%md，结果存在c(x)中
//a(x)*b(x)的最大非零项次数应小于(len-1)
const int md = 1e9 + 7, mmd = sqrt(md) + 1;
Comp A[maxn + 5], B[maxn + 5], C[maxn + 5], D[maxn + 5];
void Conv(LL *a, LL *b, LL *c, int len) {
    for(int i = 0; i < len; ++i) {
        A[i] = Comp(a[i]/mmd, b[i]/mmd);
        B[i] = Comp(a[i]%mmd, b[i]%mmd);
    }
    FFT(A, len, 1); FFT(B, len, 1);
    for(int i = 0; i < len; ++i) {
        int j = (len-i) & (len-1);
        C[i] = (A[i]*A[i] - (A[j]*A[j]).conj()) * Comp(0, -0.25);
        // 下面这步也可以拆开计算，若拆开的话后面合并时也要进行相应修改
        C[i] += (A[i]*B[i] - (A[j]*B[j]).conj()) * Comp(0.5, 0);
        D[i] = (B[i]*B[i] - (B[j]*B[j]).conj()) * Comp(0, -0.25);
    }
    FFT(C, len, -1); FFT(D, len, -1);
    for(int i = 0; i < len; ++i) {
        c[i] = LL(C[i].re+0.5) % md * mmd * mmd % md;
        c[i] = (c[i] + LL(C[i].im+0.5) % md * mmd % md) % md;
        c[i] = (c[i] + LL(D[i].re+0.5) % md) % md;
     }
}
```

- 多项式求逆

给定 $A(x)$，求 $B(x)$ 满足 $A(x)*B(x)\equiv 1\pmod{x^n}$
假设我们已经计算出 $B'(x)$ 满足 $A(x)*B'(x)\equiv 1\pmod{x^{\lceil n/2\rceil}}$
易知 $A(x)*B(x)\equiv 1\pmod{x^{\lceil n/2\rceil}}$，因此有 $B(x)-B'(x)\equiv 0\pmod{x^{\lceil n/2\rceil}}$
则易证 $(B(x)-B'(x))^2\equiv (B(x))^2-2B(x)B'(x)+(B'(x))^2\equiv 0\pmod{x^n}$
两边同乘 $A(x)$ 并移项得 $B(x)=2B'(x)-A(x)*(B'(x))^2$
因此我们可以递归求解 $B(x)$，由主定理知总时间复杂度为 $O(n\log n)$。

```
// 给出多项式a(x)，求多项式b(x)使得 a(x)*b(x) = 0 (mod x^(deg))
// 因此调用前应确保已给出a(x)的第0 ~ (deg-1)项系数并填入a[]数组
// 数组空间一定要开够(2倍原长且为2的次幂)
// 若使用FFT一定要调用Init()初始化原根
const int md = 1e9+7; // 自行设置模数
const int maxn = 1 << 18;

LL temp[maxn+5];
void ConvInv(int deg, LL *a, LL *b) {
    if(deg == 1) {
        b[0] = PowMod(a[0], md-2);
        // 由此可以看出，a(x)是否存在逆元只取决于a(x)的常数项是否存在逆元
        return;
    }
    ConvInv((deg+1)>>1, a, b); // 递归求解

```

```
17      // 因在这一步时相乘涉及的多项式次数均不超过(deg-1)
18      // 故应有len >= deg*2-1
19      int len = 1;
20      while(len < deg*2-1) len <<= 1;
21
22      // 由于是mod x^(deg), 故当前只需要a(x)的第0 ~ (deg-1)项系数(其他相当于被模掉了)
23      for(int i = 0; i < deg; ++i) temp[i] = a[i];
24      for(int i = deg; i < len; ++i) temp[i] = 0;
25
26      // 要手动把b(x)的不存在项清成0, 否则怕FFT时出现精度问题(NTT大概没有这样的问题)
27      for(int i = (deg+1)>>1; i < len; ++i) b[i] = 0;
28
29      // 计算temp[]和b[]的卷积并将结果保存回temp[]
30      // 此处使用FFT模大素数模板, 每次卷积需要4次FFT, 速度较快, 但可能存在潜在的精度问题
31      // 也可以使用三模数NTT+CRT, 这样据说每个递归层需要15次NTT, 速度较慢, 但绝对没有精度问题
32      Conv(temp, b, temp, len);
33      for(int i = deg; i < len; ++i) temp[i] = 0; // 再次手动清0不存在项
34      Conv(temp, b, temp, len); // 再次计算temp[]*b[]并将结果保存回temp[]
35      for(int i = 0; i < deg; ++i)
36          b[i] = (b[i]*2 - temp[i] + md) % md; // 计算最终结果(注意次数为0 ~ deg-1)
37  }
```

- 快速数论变换（NTT）

DFT 变换公式：

$$X_k = \sum_{n=0}^{N-1} x_n g^{nk} (mod\ P) \qquad k = 0, 1, \cdots, N-1$$

DFT 逆变换公式：

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k g^{-kn} (mod\ P) \qquad n = 0, 1, \cdots, N-1$$

其中 P 为素数，g 为模 P 的原根，且 N 必须是 $P-1$ 的因子。
由于 N 经常是 2 的幂，故可构造形如 $P = c * 2^k + 1$ 的素数。

```
 1  #include<bits/stdc++.h>
 2  using namespace std;
 3
 4  /*
 5   * const LL P = (2615053605667LL << 18) + 1, G = 3;
 6   * const LL P = (190734863287LL << 18) + 1, G = 3;
 7   * const LL P = (15LL << 27) + 1, G = 31;
 8   * const LL P = ((7 * 17) << 23) + 1, G = 3;
 9   * const LL P = ((3 * 3 * 211) << 19) + 1, G = 5;
10   * const LL P = (25 << 22) + 1, G = 3;
11   */
12  const LL P = (479 << 21) + 1, G = 3;
13  const int maxn = 1<<18;
14
15  //type=1为DFT运算, type=-1为DFT逆运算
16  void NTT(LL *a, int len, int type) {
17      int i, j, k, l;
18      for(i=1, j=len>>1; i<len-1; ++i) {
19          if(i<j) swap(a[i], a[j]);
```

```
        k = len>>1;
        while(j>=k)
            j -= k, k >>= 1;
        j += k;
    }
    LL var, step, u, v;
    for(l=2; l<=len; l<<=1) {
        step = pow_mod(G, (P-1)/l);
        for(k=0; k<len; k+=l) {
            var = 1;
            for(i=k; i<k+l/2; ++i) {
                u = a[i], v = var*a[i+l/2] % P;
                a[i] = (u+v) % P;
                a[i+l/2] = (u-v+P) % P;
                var = var*step % P;
            }
        }
    }
    if(type == -1) {
        for(i=1; i<len/2; ++i) swap(a[i], a[len-i]);
        LL inv = pow_mod(len, P-2);
        for(i=0; i<len; ++i) a[i] = a[i]*inv % P;
    }
}

int main() {
    //n必须是2的幂，若不足则在后面补0
    //n指的是NTT后算出的结果的多项式长度，故 n >= len(aa) + len(bb)
    int n = 1<<10;
    //aa[i]表示 x^i 项的系数，bb[i]类似
    Comp aa[n+5], bb[n+5];
    NTT(aa, n, 1);
    NTT(bb, n, 1);
    for(int i=0; i<n; ++i)
        aa[i] = aa[i]*bb[i] % P;
    NTT(aa, n, -1);
    return 0;
}
```

- 三模数NTT的CRT模板

```
// 三模数NTT使用的CRT模板
// 具体使用时就是通过三次NTT分别计算出卷积在模m1, m2, m3下的结果
// 然后再把三个结果提供给CRT进行合并, CRT合并时会同时进行(%mod)操作
//
// 额外需要的模板: MulMod(), PowMod()
typedef long long LL;
const int maxn = 5e5 + 10 , G = 3; // 所用到的三个模数原根均为3
const int M[] = {998244353, 1004535809, 469762049};
const LL _M = (LL)M[0] * M[1]; // 这玩意儿不要漏啊(
const int mod = 1e9 + 7; // 实际要用于取模的数
const int m1 = M[0], m2 = M[1], m3 = M[2];
const int inv1 = PowMod(m1 % m2, m2-2, m2);
const int inv2 = PowMod(m2 % m1, m1-2, m1);
```

```
14 const int inv12 = PowMod(_M % m3, m3-2, m3);
15
16 LL CRT(LL a1, LL a2, LL a3){
17     LL ret = MulMod(a1 * m2 % _M, inv2, _M);
18     (ret += MulMod(a2 * m1 % _M, inv1, _M)) %= _M;
19     LL ans = ((a3 - ret) % m3 + m3) % m3 * inv12 % m3;
20     ans = (ans % mod * (_M % mod) % mod + ret % mod) % mod;
21     return ans;
22 }
```

- 快速沃尔什变换（FWT）

Fast Walsh-Hadmard Transform用于解决一类卷积问题：

$$C_i = \sum_{j \oplus k = i} A_j * B_k$$

其中 $\oplus$ 指任一逻辑二元运算。
我们可以考虑令 $A = (A_0, A_1)$ 来进行变换。

XOR
当 $\oplus$ 指异或运算时，有如下变换：

$$tf(A) = (tf(A_0) + tf(A_1),\ tf(A_0) - tf(A_1))$$
$$utf(A) = (utf(\frac{A_0 + A_1}{2}),\ utf(\frac{A_0 - A_1}{2}))$$

AND
当 $\oplus$ 指与运算时，有如下变换：

$$tf(A) = (tf(A_0) + tf(A_1),\ tf(A_1))$$
$$utf(A) = (utf(A_0) - utf(A_1),\ utf(A_1))$$

OR
当 $\oplus$ 指或运算时，有如下变换：

$$tf(A) = (tf(A_0),\ tf(A_1) + tf(A_0))$$
$$utf(A) = (utf(A_0),\ utf(A_1) - utf(A_0))$$

XNOR, *NAND*, *NOR*
当 $\oplus$ 指异或非运算、与非运算、或非运算时，我们可以将 C 直接用异或运算、与运算、或运算的方法求出来，然后将互反的两位交换即可。

以下为 *OR* 运算的实现代码：

```
 1 #include<bits/stdc++.h>
 2 using namespace std;
 3 typedef long long LL;
 4
 5 const int maxn = (1 << 18) + 100;
 6 const int md = 1e9 + 7; //要模的数自行修改
 7
 8 //type = 1为FWT运算, type = -1为FWT逆运算
 9 //以下为OR运算
10 void FWT(LL *a, int len, int type) {
11     for(int l = 1; l < len; l <<= 1) {
```

```
12          for(int i = 0; i < len; i += (l << 1))
13              for(int k = 0; k < l; ++k) {
14                  LL x = a[i + k], y = a[i + k + l];
15                  if(type == 1)
16                      a[i + k + l] = (x + y) % md;
17                  else
18                      a[i + k + l] = (y - x + md) % md;
19              }
20      }
21  }
22
23  int main() {
24      //n必须是2的幂，若不足则在后面补0
25      //n指的是FWT后算出的结果的多项式长度，故n >= len(aa) + len(bb)
26      LL n = 1 << 10;
27      //aa[i]表示x^i项的系数，bb[i]类似
28      int aa[n + 5], bb[n + 5];
29      FWT(aa, n, 1);
30      FWT(bb, n, 1);
31      for(int i = 0; i < n; ++i)
32          aa[i] = aa[i] * bb[i] % md;
33      FWT(aa, n, -1);
34      return 0;
35  }
```

1.13.SG定理

- 基本概念

对于一个Nim博弈状态 A，$SG[A]$ 表示它的赢面的“等级”，$SG[A]=0$ 表示状态 i 必输。
若一个Nim博弈局面由 $A_1, A_2, \cdots, A_n$ 共 n 个独立状态组成，那该局面的 SG 函数为：

$$SG[A_1, A_2, \cdots, A_n] = SG[A_1] \oplus SG[A_2] \oplus \cdots \oplus SG[A_n]$$

其中 $\oplus$ 指逻辑异或运算 XOR。

```
1  const int maxn = 1e3;
2  int SG[maxn+10];
3
4  void SG_cal(int x,int len) {
5      bool vis[len+10];
6      SG[0] = x;  //set the initial status of SG[0]
7      for(int i=1;i<=len;++i) {
8          memset(vis,0,sizeof(vis));
9          //travel every sub-status of i
10         for(k=head[i];k!=tail[i];k=nex[i])
11             vis[SG[k]]=true;
12         for(int k=0;;++k)
13             if(!vis[k]) {
14                 SG[i]=k;
15                 break;
16             }
```

```
17      }
18      return;
19  }
```

- 树上删边SG游戏

叶子节点的 SG 值为 0;
中间节点的 SG 为：$SG[A] = (SG[son_1] + 1) \oplus (SG[son_2] + 1) \oplus \cdots \oplus (SG[son_k] + 1)$

1.14.线性筛法

```
 1  const int max_len = 1e9;
 2  bool P[max_len+10];
 3
 4  void prime_maker() {
 5      memset(P,0,sizeof(P));
 6      int len = max_len;
 7      //p[1] = false;
 8      for(int i=2;i*i<=len;++i) if(!P[i]) {
 9          for(int j=i;i*j<=len;++j) if(!P[j])
10              for(long long k=i*j;k<=len;k*=j) P[k]=true;
11      }
12  }
```

1.15.欧拉函数

```
 1  /*
 2   * 欧拉函数
 3   * phi[1] = 1;
 4   * 若 p 为质数, 则 phi[p^k] = p^k - p^(k-1);
 5   * 若 gcd(m,n)=1, 则 phi[m*n] = phi[m] * phi[n];
 6   */
 7  const int phi_len = 1e6;
 8  int phi[phi_len+10];
 9
10  void phi_maker() {
11      memset(phi,0,sizeof(phi));
12      phi[1] = 1;
13      for(int i=2;i<=phi_len;++i) if(!phi[i]) {
14          for(int k=i;k<=phi_len;k+=i) {
15              if(!phi[k]) phi[k] = k;
16              phi[k] = phi[k] / i * (i-1);
17          }
18      }
19  }
20
21  //O(√n)
22  LL Euler(int n) {
23      LL res = n, a = n;
24      for(int i=2; i*i<=a; ++i) {
25          if(a%i == 0) {
26              res = res/i*(i-1);
```

```
27             while(a%i == 0) a /= i;
28         }
29     }
30     if(a>1) res = res/a*(a-1);
31     return res;
32 }
```

1.16.莫比乌斯函数

```
 1 //計算mobius函數
 2 #include<bits/stdc++.h>
 3 using namespace std;
 4
 5 typedef long long LL;
 6 const int max_len = 1e8; //差不多是極限長度了
 7
 8 LL N = max_len;
 9 bool vis[max_len+5];
10 int prime[max_len+5];
11 int mu[max_len+5]; //mobius函數值
12
13 void mobius() {
14     memset(vis, 0, sizeof(vis));
15     int tot = 0;
16     mu[1] = 1;
17     for(int i=2; i<=N; ++i) {
18         if(!vis[i])
19             prime[++tot] = i, mu[i] = -1;
20         for(int k=1; k<=tot; ++k) {
21             if((LL)prime[k] * i > N) break;
22             vis[prime[k] * i] = true;
23             if(i % prime[k]) mu[i * prime[k]] = -mu[i];
24             else {
25                 mu[i * prime[k]] = 0;
26                 break;
27             }
28         }
29     }
30 }
```

1.17.拉格朗日插值

给出 n 个不同的点 $(x_0,y_0),\cdots,(x_{n-1},y_{n-1})$，其拉格朗日插值公式为：$L(x)=\sum_{j=0}^{n-1}y_j\ell_j(x)$。

其中 $\ell_j(x)$ 称为插值基函数，其公式为：$\ell_j(x)=\prod_{i=0,i\neq j}^{n-1}\frac{x-x_i}{x_j-x_i}$。

一般来说，给出 n 个不同的点，可以在 $O(n^2)$ 的时间复杂度内插出一个 $n-1$ 次多项式在给定点的值。但是如果能根据公式快速求出该多项式在 $x=1,\cdots,n$ 的值，即 $(1,f(1)),\cdots,(n,f(n))$，则可以通过下面的模板在 $O(n)$ 的时间复杂度内快速求出 $f(x)$ 在给定点的值。

```cpp
// 51nod-1258 计算\sigma_{i=1}^{n}{i^k} % P, n<=1e18, k<=50000
// 线性插值模板
// 1. 必须要预处理逆元
// 2. 调用Larange()插值前要先求出f(1), f(2), ..., f(n)并保存在f[]中
// 3. 通过n个点插出来的是(n-1)次多项式
#include<bits/stdc++.h>
using namespace std;
typedef long long LL;

const int P = 1e9+7;
const int maxn = 50000+10;

inline LL PowMod(LL a, LL b) { LL r=1; while(b) { if(b&1) r=r*a%P; a=a*a%P, b>>=1; } return r; }

// f[i]保存f(i)的值
LL f[maxn+5], inv[maxn+5];
LL pfx[maxn+5], sfx[maxn+5];

// 拉格朗日插值, 通过f(x)在x=1~n共计n个点插出一个n-1次多项式
// 并计算这个n-1次多项式在x处的值
LL Lagrange(LL n, LL x) {
    x %= P;
    LL div = 1;
    for(int i = 1; i <= n-1; ++i)  div = div * LL(-i+P) % P;
    div = PowMod(div, P-2);
    pfx[0] = sfx[n+1] = 1;
    for(int i = 1; i <= n; ++i)  pfx[i] = ((x-i+P)%P) * pfx[i-1] % P;
    for(int i = n; i >= 1; --i) sfx[i] = ((x-i+P)%P) * sfx[i+1] % P;
    LL ret = 0;
    for(int i = 1; i <= n; ++i) {
        ret = (ret + pfx[i-1] * sfx[i+1] % P * div % P * f[i]) % P;
        div = div * inv[i] % P * (P-(n-i)) % P;
    }
    return ret;
}

int main() {
    // 线性预处理逆元 (预处理部分必不可少)
    inv[1] = 1;
    for(int i = 2; i <= maxn; ++i) inv[i] = LL(P-P/i) * inv[P%i] % P;

    // 题目相关处理
    int _; scanf("%d", &_);
    while(_--) {
        LL n;
        int kk;
        scanf("%lld%d", &n, &kk);
        // 因S_kk(n)是一个关于n的(kk+1)次多项式, 故需要计算1~(kk+2)的点的值
        for(int i = 1; i <= kk+2; ++i) f[i] = (PowMod(i,kk) + f[i-1]) % P;
        printf("%lld\n", Lagrange(kk+2,n));
    }
    return 0;
}
```

1.18.Java大数开方

牛顿迭代法求 $f(x)$ 零点公式：$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$。

对应到求 $\sqrt[k]{a}$ 的递推式为：$x_{n+1} = x_n - \frac{x_n^k - a}{k\,x_n^{k-1}} = \frac{k-1}{k} x_n + \frac{a}{k\,x_n^{k-1}}$。

以下代码求 $\lfloor \sqrt[m]{n} \rfloor$，其中 $m \geq 1,\ n \geq 0$。

```java
// 计算n^(1/m)像下取整
// 一坨括号仔细核对千万别打错
import java.io.*;
import java.util.*;
import java.math.*;
public class Main {
    public static BigInteger sqrt(BigInteger n, int m) {
        if (n.compareTo(BigInteger.ONE) <= 0 || m <= 1) return n;
        BigInteger now = n.shiftRight((n.bitLength() - 1) * (m - 1) / m), last;
        do {
            last = now;
            // ( last*(m-1) + n/(last^(m-1)) ) / m;
            now =
last.multiply(BigInteger.valueOf(m-1)).add(n.divide(last.pow(m-1))).divide(BigInteger.
valueOf(m));
        } while (now.compareTo(last) < 0);
        return last;
    }
    public static void main(String[] args) {
        Scanner in = new Scanner(new BufferedInputStream(System.in));
        BigInteger n = in.nextBigInteger();
        int m = in.nextInt();
        System.out.println( sqrt(n,m) );
        in.close();
        return;
    }
}
```

1.19.数学相关理论

- Polya计数

Burnside引理：设 G 是一个有限群，作用在集合 X 上。对每个 $g \in G$，令 x^g 表示 X 中在 g 作用下的不动元素。则有如下公式：

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

即等价类个数等于每个置换不动点的均值。

Polya定理：对于含 n 个对象的置换群 G，用 t 种颜色着色的不同方案数为：

$$l = \frac{1}{|G|}\sum_{g\in G} t^{c(a_g)}$$

其中$G = \{a_1, a_2, \ldots, a_g\}$，$c(a_g)$ 为置换 a_k 的循环指标数目。

- 皮克定理

给定顶点座标均是整点（或正方形格子点）的简单多边形，设其面积为 A，内部格点（整点）数目为 a，边上格点数目为 b，则有关系式：$A = a + \frac{b}{2} - 1$。

证明：数学归纳法；特殊形式：三角形。

- 勾股数

可以用下面的方法找出所有**素勾股数**。设 $n < m$，n、m 均是正整数，令：

$$\begin{cases} a = m^2 - n^2 \\ b = 2mn \\ c = m^2 + n^2 \end{cases}$$

若 n 与 m 互素，且 n 和 m 之中有（且仅有）一个是偶数，那么 (a, b, c) 就是素勾股数。

- 原根

若 a 为模 m 乘法群的原根，则事实上该模 m 乘法群是由 a 生成的循环群；
模 m 乘法群有原根的充要条件是 $m = 2,\ 4,\ p^n,\ 2p^n$，其中 p 是奇素数（即非 2），n 是任意正整数。

- Lucas定理

对于非负整数 n 和 m 和素数 p，同余式 $\binom{n}{m} \equiv \prod_{i=0}^{k}\binom{n_i}{m_i} \pmod p$ 成立。其中：

$$n = n_k p^k + n_{k-1}p^{k-1} + \cdots + n_1 p^1 + n_0 p^0$$
$$m = m_k p^k + m_{k-1}m^{k-1} + \cdots + m_1 p^1 + m_0 p^0$$

即将 n 和 m 按照 p 进制展开。另外当 $m_i > n_i$ 时，$\binom{n_i}{m_i} = 0$。

- 逆矩阵与伴随矩阵

若 A 可逆，则 $A^{-1} = \frac{A^*}{|A|}$。其中 A^* 是 A 的伴随矩阵，即 A 的代数余子式构成矩阵的**转置**。

- Freivalds算法验算矩阵乘法结果

设有三个 n 阶矩阵 A, B, C，要求验证 AB 是否等于 C。

我们可以随机化一个 $n \times 1$ 的 01 向量 v，则有结论：$P\{A*(B*v) = C*v,\ A*B \neq C\} \leq \frac{1}{2}$。

因此我们多随机化几次向量 v 并进行上述判断，就可以显著提高判断正确的概率，时间复杂度由 $O(n^3)$ 变为 $O(n^2)$。

- 欧拉函数相关公式

$$\phi(n) = n\prod_{p|n} 1 - \frac{1}{p}$$

$$\sum_{d|n} \phi(d) = n \ \Rightarrow\ \phi(n) = n - \sum_{d|n, d<n} \phi(d) \quad \text{（常用于杜教筛）}$$

$$S(n) = \sum_{i=1}^{n} \phi(i) = \frac{n(n+1)}{2} - \sum_{i=2}^{n} S(\lfloor \frac{n}{i} \rfloor)$$

$$\sum_{i=1}^{n} i[(i,n) = 1] = \frac{1}{2}([n = 1] + n\phi(n))$$

- 莫比乌斯函数相关公式

$$\sum_{d|n} \mu(d) = [n = 1] \ \Rightarrow\ \mu(n) = [n = 1] - \sum_{d|n, d<n} \mu(d) \quad \text{（常用于杜教筛）}$$

$$M(n) = \sum_{i=1}^{n} \mu(i) = 1 - \sum_{i=2}^{n} M(\lfloor \frac{n}{i} \rfloor)$$

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

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

- 欧拉降幂公式

$$A^B \mod C = A^{(B \mod \phi(c) + \phi(c))} \mod C, \quad B \geq \phi(c)$$

式中的函数为欧拉函数。

- 第二类斯特林数相关公式

$S(n,k)$ 表示将 n 个不同的物品划分成 k 个（不可分辨）的集合的方案数。则有如下公式：

$$S(n,n) = S(n,1) = 1, \quad S(n,k) = S(n-1,k-1) + kS(n-1,k)$$

$$S(n,n-1) = C(n,2) = n(n-1)/2$$

$$S(n,2) = 2^{n-1} - 1$$

$$S(n,k) = \frac{1}{k!} \sum_{j=1}^{k} (-1)^{k-j} C(k,j) j^n$$

- 杜教筛

定义两个数论函数 $f(x), g(x)$ 的Dirichlet卷积为：

$$(f * g)(n) = \sum_{d|n} f(d) g(\frac{n}{d})$$

设 $f(n)$ 为一数论函数，现需计算：

$$S(n)=\sum_{i=1}^{n}f(i)$$

通过构造 $S(n)$ 关于 $S(\lfloor\frac{n}{i}\rfloor)$ 的递推式，套上预处理和 Hash 来快速计算。例如计算 Mertens 函数可以利用公式：

$$M(n)=\sum_{i=1}^{n}\mu(i)=n[n=1]-\sum_{i=2}^{n}M(\lfloor\frac{n}{i}\rfloor)$$

通常会使用 $\phi(x),\mu(x),1$ 等数论函数来构造Dirichlet卷积，使得 $(f*g)(n)=\epsilon$ 或其他特殊的值或函数，然后再构造类似于 $\sum_{i=1}^{n}\epsilon(i)$ 这样的式子来求出杜教筛递推式。

- （第一）伯努利数与等幂求和
  - 伯努利数记为 $B_i(i=0,1,\cdots)$，其中 $B_0=1$，$B_1=\mp\frac{1}{2}$，我们称 $B_1=-\frac{1}{2}$ 的伯努利数为第一伯努利数。以下出现的 B_i 均指第一类伯努利数。
  - 母函数定义式：$\frac{x}{e^x-1}=\sum_{n=0}^{\infty}B_n\frac{x^n}{n!}$
  - 伯努利数计算方法：由公式 $\sum_{i=0}^{n}\binom{n+1}{i}B_i=0$ 可递归计算伯努利数
  - 等幂求和公式：$S_m(n)=\sum_{i=1}^{n}i^m=\frac{1}{m+1}\sum_{i=0}^{m}\binom{m+1}{i}B_i\,(n+1)^{m+1-i}$

    $$\sum_{i=1}^{n}i=\frac{n(n+1)}{2}$$

    $$\sum_{i=1}^{n}i^2=\frac{n(n+1)(2n+1)}{6}$$

    $$\sum_{i=1}^{n}i^3=\frac{n^2(n+1)^2}{4}=(\sum_{i=1}^{n}i)^2$$
  - 拉格朗日插值计算方法：可以通过归纳法证明 $S_m(n)$ 是一个关于 n 的 $(m+1)$ 次多项式，因此可以暴力计算 $S_m(n)$ 在 $1,\cdots,(m+2)$ 处的值，然后套个线性插值模板就 $O(m)$ 了。

- 卡特兰数

卡特兰数通项公式为：$C_n=\frac{1}{n+1}\binom{2n}{n}\ (n\geq 0)$。

递推关系：$C_0=1$，$C_{n+1}=\sum_{i=0}^{n}C_iC_{n-i}$。

组合意义：

- C_n 表示包含 n 对括号的合法括号匹配序列的个数（证明：$\binom{2n}{n}-\binom{2n}{n+1}=\frac{1}{n+1}\binom{2n}{n}=C_n$）；
- C_n 表示由 n 个节点组成的不同构二叉树的方案数；
- C_n 表示由 $2n+1$ 个节点组成的不同构满二叉树（**指不存在只含一个孩子的节点的树**）的方案数；
- C_n 表示通过连接顶点将有 $n+2$ 条边的凸多边形分成三角形的方案数（此时每个顶点是本质不同的）。

- 没什么卵用的组合公式

$$\binom{k}{k}+\binom{k+1}{k}+\cdots+\binom{n}{k}=\binom{n+1}{k+1}$$

$$A_n=\sum_{i=0}^{n}\binom{n}{i}B_i \Leftrightarrow B_n=\sum_{i=0}^{n}(-1)^{n-i}\binom{n}{i}A_i \quad \text{（二项式反演）}$$

- 数学分析公式

$$F(y)=\int_{a(y)}^{b(y)}f(x,y)dx$$

$$F'(y)=\int_{a(y)}^{b(y)}f_y(x,y)dx+f(b(y),y)b'(y)-f(a(y),y)a'(y)$$

- Stern–Brocot Tree（用于枚举分数）

简单来说，Stern–Brocot Tree 就是把所有正的既约分数（包括假分数）按照一定顺序塞入一个无限深度的满二叉排序树中，具体构造是通过定义数列及连分数来进行的，再此不加赘述，只叙述一个在二叉树上搜索特定分数的方法。

假定当前要找的分数是 a，初始化两个数 $L=\frac{0}{1}$，$R=\frac{1}{0}$，并重复一下步骤：

（1）设$L=\frac{p_1}{q_1}$，$R=\frac{p_2}{q_2}$，令 $M=\frac{p_1+p_2}{q_1+q_2}$（$M$ 不用约分，事实上 M 已经是既约分数了）；
（2）若 $a=M$，则此时已找到目标分数，跳出循环；
（3）若 $a<M$，则令$R=M$，重复（1）；
（4）若 $a>M$，则令$L=M$，重复（1）。

循环结束后，每次生成的中间值 M 构成的序列，就是在 Stern–Brocot Tree 上由根结点走到目标分数所在结点时所经过的一系列结点对应的值。应用时可以通过二分找拐点来加速。
应用：kattis - Probe Droids (NAIPC 2018 F)

- Pell方程

具有如下形式的方程称为 Pell 方程：

$$x^2-ny^2=1$$

其中 $n\in\mathbb{N}$ 且**非完全平方数。**

考虑 $\sqrt{n}$ 的连分数逼近序列 $\frac{h_i}{k_i}$，在其中所有使 $x=h_i,y=k_i$ 为Pell方程的解的分数中，**取 h_i 最小的一组**并记为 $x_1=h_i,y_1=k_i$，则 (x_1,y_1) 称为方程的基本解。而方程的所有正整数解可表示为：

$$x_k+y_k\sqrt{n}=(x_1+y_1\sqrt{n})^k$$

2. 字符串

2.1.KMP算法

```cpp
#include<bits/stdc++.h>
using namespace std;

const int max_len = 1e7;

void KMP(char *A, char *B) { //A爲匹配串, B爲模式串
    //計算失配函數
    int nA = strlen(A), nB = strlen(B);
    int f[max_len + 5], now = 0;
    f[1] = 0;
    for(int i=1; i<nB; ++i) {
        while(now && B[now] != B[i]) now = f[now];
        f[i+1] = B[now] == B[i] ? (++now) : 0;
    }

    //模式匹配
    now = 0;
    for(int i=0; i<nA; ++i) {
        while(now && B[now] != A[i]) now = f[now];
        if(A[i] == B[now]) ++now;
        if(now == nA) {
            printf("Well Done\n");
            break;
        }
    }
}
```

2.2.Z-Function

```cpp
// 对于字符串a[1...len]计算其每一位的Z-Function(保存在f[]中)
// 其中f[i]表示a[]从第i位开始和前缀的最长匹配, 即a[1...f[i]] == a[i...(i+f[i]-1)]
// 需特别注意, 这里的字符串a[]下标是从1开始一直到len的
void Z_Function(char *a, int len, int *f) {
    f[1] = len;
    for(int i = 2, l = 0, r = 0; i <= len; ++i) {
        if(i <= r) f[i] = min(r-i+1, f[i-l+1]);
        else f[i] = 0;
        while(i+f[i]<=len && a[i+f[i]]==a[f[i]+1]) ++f[i];
        if(i+f[i]-1 > r) l = i, r = i+f[i]-1;
    }
}
```

2.3.AC自动机

```cpp
#include<bits/stdc++.h>
using namespace std;
```

```
3  const int maxnode = 1e7, c_size = 127; //結點個數與字符集大小
4
5  struct Tire {
6      int top;
7      int nex[maxnode+5][c_size+5], val[maxnode+5];
8      int fail[maxnode+5], last[maxnode+5];
9
10     void clear() {
11         top = val[0] = 0;
12         memset(nex[0], 0, sizeof(nex[0]));
13     }
14
15     inline int c_id(char c){ return c; } //自行設定返回值
16
17     void insert(char *a, int vv) {
18         int len = strlen(a), u = 0, t;
19         for(int i=0; i<len; ++i) {
20             t = c_id(a[i]);
21             if(!nex[u][t]) {
22                 nex[u][t] = (++top), val[top] = 0;
23                 memset(nex[top], 0, sizeof(nex[top]));
24             }
25             u = nex[u][t];
26         }
27         val[u] = vv; //需根據情況設定如何保存關鍵值
28     }
29
30     void get_fail() {
31         fail[0] = last[0] = 0;
32         queue<int> q;
33         for(int i=0; i<c_size; ++i) if(int &t = nex[0][i]) {
34             fail[t] = last[t] = 0;
35             q.push(t);
36         }
37         while(!q.empty()) {
38             int x = q.front(); q.pop();
39             for(int i=0; i<c_size; ++i) {
40                 int u = nex[x][i];
41                 if(!u) { nex[x][i] = nex[fail[x]][i]; continue; }
42                 q.push(u);
43                 int v = fail[x];
44                 while(v && !nex[v][i]) v = fail[v];
45                 fail[u] = nex[v][i];
46                 //last函數，val在此處的含義爲：結點是否爲一個串的結尾
47                 last[u] = val[fail[u]] ? fail[u] : last[fail[u]];
48                 //另一種題目的常用寫法
49                 //match[u] |= match[fail[u]];
50             }
51         }
52     }
53
54     void find(int x) //查詢所有以該串結尾的字符串
55     {
56         if(x) {
```

```
57              //查詢內容
58              //...;
59              find(last[x]);
60          }
61      }
62
63      /*匹配函數略，可参考kmp的思路
64      void query(char *a) {
65          ...;
66      }*/
67  };
```

2.4.后缀数组

- O(nlogn)后缀数组（时代的眼泪）

```
 1 #include<bits/stdc++.h>
 2 using namespace std;
 3 const int maxn = 100000;
 4
 5 int sa[maxn+5], rk[maxn+5], ht[maxn+5];
 6 int ttx[maxn+5], tty[maxn+5], tc[maxn+5];
 7
 8 // len为a的长度，size为字符集大小，即size值大于最大字符的ascii码;
 9 // 这样计算出的sa数组中，sa[0] = len(即终止字符排最前面),
10 // 而sa[1] ~ sa[len]中存的即为所要求的原字符串的后缀数组值.
11 // 以上的len均指a的原始长度
12 void BuildSA(char *a, int len, int size) {
13     a[len++] = '\0'; // 手动添加末尾终止字符
14     int *tx = ttx, *ty = tty; // 必须要使用指针，方便交换数组
15     int i, k, p;
16     // 以下为直接根据原串首字母进行基数排序
17     for(i = 0; i < size; ++i) tc[i] = 0;
18     for(i = 0; i < len; ++i) ++tc[tx[i] = a[i]];
19     for(i = 1; i < size; ++i) tc[i] += tc[i-1];
20     for(i = len - 1; i >= 0; --i) sa[--tc[tx[i]]] = i;
21
22     for(k = 1; k <= len; k <<= 1) {
23         // 以下根据第二关键字(即[k,k*2]的字典序)进行排序，结果存在ty[]中
24         // 要仔细分析以下几行的意义
25         p = 0;
26         for(i = len - k; i < len; ++i) ty[p++] = i;
27         for(i = 0; i < len; ++i) if(sa[i] >= k) ty[p++] = sa[i] - k;
28
29         // 以下在保证第二关键字相同的情况下对第一关键字进行基数排序
30         for(i = 0; i < size; ++i) tc[i] = 0;
31         for(i = 0; i < len; ++i) ++tc[tx[ty[i]]];
32         for(i = 1; i < size; ++i) tc[i] += tc[i-1];
33
34         // 重构sa数组及tx
35         for(i = len - 1; i >= 0; --i) sa[--tc[tx[ty[i]]]] = ty[i];
```

```
36          swap(tx, ty);
37          p = 1, tx[sa[0]] = 0;
38          for(i = 1; i < len; ++i)
39              tx[sa[i]] = ty[sa[i]] == ty[sa[i-1]] && ty[min(sa[i]+k,len-1)] ==
ty[min(sa[i-1]+k,len-1)] ? p-1 : p++;
40          if(p >= len) break;
41          size = p;
42      }
43      --len; //回退len
44  }
45
46  // ht[i]表示后缀sa[i]与sa[i-1]的LCP值;
47  // rk[i]表示后缀i在sa数组中的下标(即后缀i的字典序);
48  // 则有性质: ht[rk[i]] >= ht[rk[i-1]] - 1;
49  // 由于所求字符串已添加终止结点, 故rk[len-1]=0, 在计算时要注意不要越界;
50  // ht的有效值为ht[1] ~ ht[len-1].
51  void GetHeight(char *a, int len) {
52      int i, j, k = 0;
53      for(i = 0; i <= len; ++i) rk[sa[i]] = i;
54      for(i = 0; i < len; ++i) { // 由于rk[len] = 0, 故无需考虑
55          if(k) --k;
56          j = sa[rk[i] - 1];
57          while(a[i+k] == a[j+k]) ++k;
58          ht[rk[i]] = k;
59      }
60  }
61
62  int rmq[maxn + 5][MaxStep + 2];
63  void InitRMQ(int len) {
64      for(int i = 1; i <= len; ++i) {
65          rmq[i][0] = ht[i]; // 这导致rmq[i][k]存的是sa[i-2^k]~sa[i]之间的串的LCP
66          for(int k = 1; (1 << k) <= i; ++k)
67              rmq[i][k] = min(rmq[i][k-1], rmq[i - (1<<(k-1))][k-1]);
68      }
69  }
70
71  // 进行LCP前应先调用InitRMQ()初始化RMQ数组;
72  // 要注意这里l != r, 因为后缀数组的RMQ为了构造上的方便,
73  // 实际上并没有存sa[i]和自身的LCP值;
74  // 如果一定要求这样的值, 则需要特判.
75  int LCP(int l, int r) {
76      l = rk[l], r = rk[r];
77      if(l > r) swap(l, r);
78      // 因为rmq[i][k]存的是排名在[i-2^k, i]之间的串的LCP
79      // 所以直接在[l,r]区间进行RMQ算出来的值其实是sa[l-1]~sa[r]的LCP值, 因此需要++l
80      int i; ++l;
81      for(i = 0; (1 << (i+1)) < (r-l+1); ++i);
82      return min(rmq[r][i], rmq[l + (1<<i) - 1][i]);
83  }
84
85  int main() {
86      int len, c_size;
87      char aa[maxn+5];
88      scanf("%s", aa);
```

```
89      len = strlen(aa);
90      BuildSA(aa, len, c_size); // c_size应不小于字符串所有字符的ascii码的最大值
91      GetHeight(aa, len);
92  }
```

- SA-IS算法O(n)构建后缀数组

```
 1  // copy from http://uoj.ac/submission/255151
 2  #include <cstdio>
 3  #include <cstring>
 4  #include <algorithm>
 5  using namespace std;
 6
 7  namespace SA{
 8
 9  const int MAXN = 1e5 + 10; // 字符串长度上限
10
11  // sa[]表示后缀数组，下标0~n，其中sa[0]表示终止字符(字典序最小)
12  // rk[i]表示suffix(i)对应的排名
13  // ht[i]表示suffix(sa[i-1])与suffix(sa[i])的LCP长度
14  int sa[MAXN],rk[MAXN],ht[MAXN],s[MAXN<<1],t[MAXN<<1];
15  int p[MAXN],b[MAXN],cur[MAXN];
16  #define pushS(x) sa[cur[s[x]]--] = x
17  #define pushL(x) sa[cur[s[x]]++] = x
18  void inducedsort(int n, int n1, int m, int *s, int *t, int *v) {
19      fill_n(b,m,0),fill_n(sa,n,-1);
20      for(int i=0;i<n;i++) b[s[i]]++;
21      for(int i=1;i<m;i++) b[i] += b[i-1];
22      for(int i=0;i<m;i++) cur[i] = b[i]-1;
23      for(int i=n1-1;i>=0;--i) pushS(v[i]);
24      for(int i=1;i<m;i++) cur[i] = b[i-1];
25
26      // 自行修改部分，为防止越界．如果出锅了再改回来
27      //for(int i=0;i<n;i++) if(sa[i]>0&&t[sa[i]-1]))pushL(sa[i]-1);
28      for(int i=0;i<n;i++) if(sa[i]>0&&(sa[i]==0||t[sa[i]-1]))pushL(sa[i]-1);
29
30      for(int i=0;i<m;i++) cur[i] = b[i]-1;
31
32      // 自行修改部分，为防止越界．如果出锅了再改回来
33      //for(int i=n-1;i>=0;--i) if(sa[i]>0&&!t[sa[i]-1])pushS(sa[i]-1);
34      for(int i=n-1;i>=0;--i) if(sa[i]>0&&(sa[i]==0||!t[sa[i]-1]))pushS(sa[i]-1);
35  }
36
37  void sais(int n,int m,int *s,int *t,int *p) {
38      int n1 = t[n] = 0,*s1 = s+n,ch = rk[0] = -1;
39      for(int i = n-1;~i;--i) t[i] = s[i]==s[i+1]?t[i+1]:s[i]>s[i+1];
40      for(int i = 0;i<n;i++) rk[i] = (!t[i]&&t[i-1])?(p[n1]=i,n1++):-1;
41      inducedsort(n,n1,m,s,t,p);
42      for(int i = 0,x,y;i<n;i++) if(~(x = rk[sa[i]])) {
43          if(ch < 1 || p[x+1] - p[x] != p[y+1] - p[y]) ch++;
44          else for(int j=p[x],k=p[y];j<=p[x+1];j++,k++)
45              if(((s[j]<<1)|t[j])!=((s[k]<<1)|t[k])){ch++;break;}
46          s1[y=x] = ch;
47      }
```

```
48      if(ch+1<n1) sais(n1,ch+1,s1,t+n,p+n1);
49      else for(int i = 0;i<n1;i++) sa[s1[i]] = i;
50      for(int i = 0;i<n1;i++) s1[i] = p[sa[i]];
51      inducedsort(n,n1,m,s,t,s1);
52  }
53
54  template<typename T>
55  int mapCharToInt(int n,const T *str) {
56      int m = *max_element(str,str+n);
57      fill_n(rk,m+1,0);
58      for(int i = 0;i<n;i++) rk[str[i]] = 1;
59      for(int i = 1;i<=m;i++) rk[i] += rk[i-1];
60      for(int i = 0;i<n;i++) s[i] = rk[str[i]]-1;
61      return rk[m];
62  }
63
64
65  // 这份GetHeight和前面倍增排序中所用的GetHeight是有区别的
66  // 具体区别在于长度n是否包含终止字符
67  void getheight(int n) {
68      int i, j, k = 0;
69      for(i = 0; i < n; ++i) rk[sa[i]] = i;
70      for(i = 0; i < n-1; ++i) { //由于rk[n-1] = 0, 故无需考虑
71          if(k) --k;
72          j = sa[rk[i] - 1];
73          while(s[i+k] == s[j+k]) ++k;
74          ht[rk[i]] = k;
75      }
76  }
77
78  template<typename T>
79  void BuildSA(int n,const T *str) { //str[n] miniuium
80      int m = mapCharToInt(++n,str);
81      sais(n,m,s,t,p);
82      getheight(n);
83  }
84
85  }
86
87  const int maxn = 1e5;
88
89  int n;
90  char str[maxn + 5];
91
92  int main(){
93      scanf("%s", str);
94      n = strlen(str); // 字符串下标从0开始, 第n位放终止字符
95      str[n] = '\0'; // 显式的放置终止字符
96      SA::BuildSA(n, str); // 调用方式
97      for(int i = 1;i<=n;i++)
98          printf("%d%c", SA::sa[i]+1, (i==n)?'\n':' ');
99      for(int i = 2;i<=n;i++)
100         printf("%d%c", SA::ht[i], (i==n)?'\n':' ');
101     return 0;
```

```
102 }
```

- 后缀数组的小结论

设 P 为一个长度为 n 的后缀数组，R 为其对应的Rank数组，同时令 $R[n+1]=0$。
现定义数组 $R_+[i]=R[P[i]+1],(1\le i\le n)$，同时定义集合 $U=\{i\,|\,i\in[1,n-1],\ R_+[i]>R_+[i+1]\}$。
最后令 $d=|U|$，设字符集大小为 σ。那么我们有如下定理：

定理1：长度为 n 且对应的后缀数组为 P 的所有字符串的个数为 $\binom{n+\sigma-d-1}{\sigma-d-1}$，若 $d>\sigma-1$，则这样的字符串不存在；

定理2：长度为 n 且对应的后缀数组为 P 的所有字符串中，满足恰由 k $(k\le\sigma)$ 种不同的字符组成的字符串的个数为 $\binom{n-d-1}{k-d-1}$，若$d>k-1$，则这样的字符串不存在。

2.5.后缀自动机（SAM）

```
#include<bits/stdc++.h>
using namespace std;
/*
 * 后缀自动机（SAM）
 * 只有parent树才是树形结构，且存在严格包含关系
 * 而对于正向情况，同一个State可能被多个不同的其他State找到
 * 对于一个State，其唯一对应于一系列子串，这些子串的长度构成一个连续闭区间
 * 设其上界为Max(s)，下界为Min(s)，则有如下公式：
 * Max(s) = s->val，Min(s) = Max(par(s))+1;
 * 这在各种计数问题中会用到
 * SAM的状态数至少为原串长度的2倍
 */
const int maxn = 1e5, c_size = 26;

struct State {
    int val;
    State *par, *go[c_size];
    void init() { val = 0, par = NULL, memset(go,0,sizeof(go)); }
}que[maxn*2+5], *root, *last;
int tot;

void Extend(int w) {
    State *p = last, *np = que + (tot++);
    np->init();
    np->val = p->val+1;
    while(p!=NULL && p->go[w]==NULL)
        p->go[w] = np, p = p->par;
    if(p == NULL) np->par = root;
    else {
        State *q = p->go[w];
        if(p->val+1 == q->val) np->par = q;
        else {
            State *nq = que + (tot++);
            *nq = *q; //要小心这里可能存在的数据覆盖问题
            nq->val = p->val+1;
```

```
36              np->par = q->par = nq;
37              while(p!=NULL && p->go[w]==q)
38                  p->go[w] = nq, p = p->par;
39          }
40      }
41      last = np;
42  }
43
44  void init() {
45      root = last = que;
46      tot = 1, que[0].init();
47  }
48
49  /*
50   * SAM计算LCS的计算过程，match是可选的，需要自己加上
51   * 最后根据题目要求可以加上基数排序更新最终的match直并计算一些信息
52   */
53  char arr[maxn+5];
54  int cont[maxn+5], S[maxn*2+5];
55  void LCS {
56      State *p = root;
57      int now = 0, len = strlen(arr);
58      for(i=0; i<len; ++i) {
59          int w = arr[i]-'a';
60          if(p->go[w] != NULL) {
61              p = p->go[w];
62              p->match = max(p->match, ++now);
63          }
64          else {
65              while(p!=NULL && p->go[w]==NULL)
66                  p = p->par; //往回找到的串会缩短
67              if(p == NULL) p = root, now = 0;
68              else {
69                  now = p->val+1; //此处要特别注意
70                  p = p->go[w];
71                  p->match = max(p->match, now);
72              }
73          }
74      }
75      //基数排序，设S_len为SAM中字符串的长度
76      for(int i=0; i<=S_len; ++i) cont[i] = 0;
77      for(int i=0; i<tot; ++i) ++cont[que[i].val];
78      for(int i=1; i<=S_len; ++i) cont[i] += cont[i-1];
79      for(int i=tot-1; i>=1; --i) S[--cont[que[i].val]] = i;
80      //排序完毕后，排序后的结果便存在S[0]~S[tot-1]
81      //即在parent树中，一个结点的孩子必定排在它后面
82
83      /*
84       * SAM统计第k小字串（第k大类似）
85       * 首先确保SAM的状态中存有当前后缀的出现次数cnt
86       * 要区分是否统计相同子串的情况（对应的cnt会不同）
87       * 然后增设一个sum值，表示以当前串为前缀的串有多少个
88       */
89      for(int i = tot - 1; i >= 0; --i) {
```

```
90          que[S[i]].sum = que[S[i]].cnt; //特别注意要按照拓扑序遍历
91          for(int k = 0; k < c_size; ++k) {
92              if(que[S[i]].go[k] != NULL)
93                  que[S[i]].sum += que[S[i]].go[k]->sum;
94          }
95      }
96      //最后只要再正着dfs一遍就可以了
97  }
```

2.6.广义后缀自动机（Trie上建SAM）

BZOJ-3926: [ZJOI2015]诸神眷顾的幻想乡

题意：给一棵子母树，计算树上本质不同字符串的个数（同一段点列，正着走和反着走可能会产生不一样的串），保证叶结点不超过 20 个。

题解：从每一个叶结点分别 dfs 一遍子母树，建立一个 $l * c * n$ 规模的广义后缀自动机（l 为叶结点个数，c 为字符集大小，n 为树的结点个数），然后统计本质不同的字符串。时间复杂度是线性的。详细构造代码如下：

```
 1  #include<bits/stdc++.h>
 2  using namespace std;
 3
 4  const int maxn = 1e5;
 5  const int maxnode = maxn * 20 * 2, c_sz = 10;
 6
 7  struct State {
 8      int val;
 9      State *par, *go[c_sz];
10      void init() { val = 0, par = NULL, memset(go, 0, sizeof(go)); }
11  } que[maxnode + 5], *root;
12  int tot;
13
14  //在线构造算法，长得跟线性SAM几乎一摸一样……
15  State* Extend(int w, State *last) {
16      //注意这一步与线性SAM不同，其实不加也可以但会浪费空间
17      if(last->go[w] != NULL && last->val + 1 == last->go[w]->val)
18          return last->go[w];
19      State *p = last, *np = que + (tot++);
20      np->init();
21      np->val = p->val + 1;
22      while(p != NULL && p->go[w] == NULL)
23          p->go[w] = np, p = p->par;
24      if(p == NULL) np->par = root;
25      else {
26          State *q = p->go[w];
27          if(p->val + 1 == q->val) np->par = q;
28          else {
29              State *nq = que + (tot++);
30              *nq = *q;
31              nq->val = p->val + 1;
32              np->par = q->par = nq;
33              while(p != NULL && p->go[w] == q)
34                  p->go[w] = nq, p = p->par;
35          }
```

```
36      }
37      return np;
38  }
39
40  int n, col[maxn + 5];
41  vector<int> G[maxn + 5];
42
43  //树形构造的关键部分
44  void dfs(int x, int pre, State *last) {
45      last = Extend(col[x], last);
46      for(int i = 0; i < G[x].size(); ++i)
47          if(G[x][i] != pre)
48              dfs(G[x][i], x, last);
49  }
50
51  int main() {
52      int u, v;
53      scanf("%d%d", &n, &u);
54      for(int i = 1; i <= n; ++i) scanf("%d", col + i);
55      for(int i = 1; i < n; ++i) {
56          scanf("%d%d", &u, &v);
57          G[u].push_back(v);
58          G[v].push_back(u);
59      }
60      root = que;
61      tot = 1, que[0].init();
62      for(int i = 1; i <= n; ++i) {
63          if(G[i].size() == 1)
64              dfs(i, 0, root);
65      }
66      long long ans = 0;
67      //统计答案时候和普通的SAM一样
68      for(int i = 1; i < tot; ++i)
69          ans += que[i].val - que[i].par->val;
70      printf("%lld\n", ans);
71      return 0;
72  }
```

2.7.回文自动机

```
1  #include<bits/stdc++.h>
2  using namespace std;
3
4  /*
5   * 在回文树（Palindromic Tree）中，其每个结点代表一个本质不同的回文串
6   * 其中0号结点表示偶数串的空串，1号结点表示奇数串的空串
7   * 0号结点的fail指针指向1号结点（fail[0] = 1）
8   * 故实际存在的不同回文串个数为 tree.tot-2
9   */
10 const int maxn = 1e5, c_size = 26;
11 struct PTree {
12     int tot, last, S_len;
13     int len[maxn+5], fail[maxn+5], S[maxn+5];
```

```
14      //nex[i][w]表示字符串i在前后加上字符w时指向的字符串
15      int nex[maxn+5][c_size];
16
17      int newnode(int l) {
18          len[tot] = l, fail[tot] = 0;
19          //此处将所有nex赋值为0，是为了先判断aa形式的回文串，再判断a形式的回文串
20          //而a形式的回文串是必定存在的
21          memset(nex[tot], 0, sizeof(nex[tot]));
22          return tot++;
23      }
24
25      void init() {
26          tot = 0;
27          newnode(0), newnode(-1);
28          fail[0] = 1, last = 0;
29          //此处为字符串的边界标记
30          //在支持两端添加字符的回文树中，可以在每次get_fail前重新标记一下两端的边界
31          S[S_len = 0] = -1;
32      }
33
34      int get_fail(int x) {
35          while(S[S_len-len[x]-1] != S[S_len])
36              x = fail[x];
37          return x;
38      }
39
40      void add(int w) {
41          S[++S_len] = w;
42          last = get_fail(last);
43          if(nex[last][w] == 0) {
44              int u = newnode(len[last]+2);
45              fail[u] = nex[get_fail(fail[last])][w];
46              //num[u] = num[fail[u]]+1; 一种十分有用的计数方法
47              //这里的num[u]表示使用了S[S_len]位置的回文串的个数
48              nex[last][w] = u;
49          }
50          last = nex[last][w];
51          //return num[last];
52      }
53  };
```

2.8.Shift - And 字符串匹配

HDOJ-6190: Matching in a Tree

题意：给一棵动态构造的字典树，又给一个字符不确定的字符串，再给一些询问查询该串的在一定长度范围内的字串是否在字典树中某个指定的前缀中作为字串出现过。

题解：注意到给定的字符串很短，因此可以用bitset加速，通过Shift-And来处理这种字符不确定的模式匹配问题。

```
1  //hdoj-6190 ICPC2017广西邀请赛 shift-and模版题
2  #include<bits/stdc++.h>
3  using namespace std;
4
```

```cpp
#define mem(a,b) memset(a,b,sizeof(a))
#define REP(i,a,b) for(int i=a; i<=b; ++i)
#define FOR(i,a,b) for(int i=a; i<b; ++i)
#define MP make_pair
typedef long long LL;
typedef pair<int,int> pii;

const int maxn = int(1e3), maxm = 1e6, md = 1e9 + 7, c_size = 26;

struct Node {
    int fa, cc;
    //dp[i] = true 表示存在一个长度为i且以当前结点为末尾的串;
    //res[i]=true 表示长度为i的字串在当前结点对应的串中出现过;
    bitset<maxn + 1> dp, res;
} tree[maxm + 5];

struct Que {
    int l, r, u, n;
    Que(): l(0), r(0), u(0), n(0) {}
    Que(int _l, int _r, int _u, int _n) {
        l = _l, r = _r, u = _u, n = _n;
    }
};

vector<Que> arr;
//f[c][i]为真表示模式串第i个位置可以放字母c;
bitset<maxn + 1> f[c_size];

int main() {
    int T, n, u, v, l, r, tot;
    char str[10], ord[10];
    LL ans;
    tree[0].dp.reset(), tree[0].dp[0] = 1;
    tree[0].res.reset();
    while(~scanf("%d", &T)) {
        n = tot = 0;
        arr.clear();
        for(int i = 0; i < c_size; ++i) f[i].reset();
        while(T--) {
            scanf("%s", ord);
            if(ord[1] == 'D') {
                scanf("%d%d%s", &u, &v, str);
                tree[v].fa = u, tree[v].cc = str[0] - 'a';
                tot = v;
            } else if(ord[1] == 'S') {
                scanf("%d%d%d", &l, &r, &u);
                arr.push_back(Que(l, r, u, n));
            } else {
                scanf("%s", str), l = str[0] - 'a';
                scanf("%s", str), r = str[0] - 'a';
                ++n;
                for(int i = l; i <= r; ++i)
                    f[i][n] = 1; //f的构建
            }
```

```
59          }
60          for(int i = 1; i <= tot; ++i) {
61              Node &x = tree[i];
62              x.dp = (tree[x.fa].dp << 1) & f[x.cc]; //shift-and关键步骤
63              x.dp[0] = 1; //注意这里的赋值
64              x.res = tree[x.fa].res | x.dp;
65          }
66          ans = 0;
67          for(auto it: arr) {
68              int cnt = (tree[it.u].res >> it.l).count() - (tree[it.u].res >> (it.r +
1)).count();
69              ans = (ans * 233 + (cnt ? 1 : 2)) % md;
70          }
71          printf("%lld\n", ans);
72      }
73      return 0;
74  }
```

3. 图论

3.1.Tarjan算法求强连通分量

```cpp
//Tarjan求强连通分量
#include<bits/stdc++.h>
using namespace std;

const int maxn = 10000, maxm = 1000000;

//n为结点个数，m为边数，使用链式前向星存储边
int n, m;
int h[maxn+5], e[maxm+5], nex[maxm+5];

/*
 * scc为强连通块个数，sta[top]为栈顶元素，Index为当前递归时间
 * DFN为时间戳，Low为当前点最早追溯到的时间，Belong为所属分块
 */
int scc, top, Index;
int sta[maxn+5], DFN[maxn+5], Low[maxn+5], Belong[maxn+5];
char Insta[maxn+5];

void init() {
    memset(h, 0, sizeof(h));
    int u, v, tot = 0;
    for(int i=1; i<=m; ++i)
        e[++tot] = v, nex[tot] = h[u], h[u] = tot;
}

void Tarjan(int u) {
    DFN[u] = Low[u] = ++Index;
    sta[++top] = u, Insta[u] = 1;
    for(int i=h[u]; i; i=nex[i]) {
        if(!DFN[e[i]]) {
            Tarjan(e[i]);
            Low[u] = min(Low[u], Low[e[i]]);
        }
        else if(Insta[e[i]])
            Low[u] = min(Low[u], Low[e[i]]);
    }
    int t;
    if(DFN[u] == Low[u]) {
        ++scc;
        do {
            //此处可以根据题目要求各种扩展
            t = sta[top--];
            Insta[t] = 0;
            Belong[t] = scc;
        } while(t != u);
    }
}

int main() {
```

```
50      init();
51      scc = top = Index = 0;
52      memset(DFN, 0, sizeof(DFN));
53      memset(Insta, 0, sizeof(Insta));
54      for(int i=1; i<=n; ++i)
55          if(!DFN[i])
56              Tarjan(i);
57      return 0;
58  }
```

3.2.最大流Dinic

```
 1  //Powered By Metowolf
 2  #include<vector>
 3  #include<queue>
 4  #include<algorithm>
 5  using namespace std;
 6
 7  const int MAXN = 100 + 10;
 8  const int INF = 1e9;
 9  struct Edge {
10      int from, to, cap, flow;
11  };
12  struct Dinic {
13      int n, m, s, t;
14      vector<Edge> edges; //边表，edges[e]和edges[e^1]互为反向弧
15      vector<int> G[MAXN]; //邻接表，G[i][j]表示结点i的第j条边在e数组中的序号
16      bool vis[MAXN]; //BFS使用
17      int d[MAXN]; //从起点到i的距离
18      void AddEdge(int from, int to, int cap) {
19          edges.push_back((Edge) {
20              from, to, cap, 0
21          });
22          edges.push_back((Edge) {
23              to, from, 0, 0
24          });
25          m = edges.size();
26          G[from].push_back(m - 2);
27          G[to].push_back(m - 1);
28      }
29      bool BFS() { //使用BFS计算出每一个点在残量网络中到t的最短距离d
30          memset(vis, 0, sizeof(vis));
31          queue<int> Q;
32          Q.push(s);
33          vis[s] = 1;
34          d[s] = 0;
35          while(!Q.empty()) {
36              int x = Q.front();
37              Q.pop();
38              for(int i = 0; i < G[x].size(); ++i) {
39                  Edge &e = edges[G[x][i]];
40                  if(!vis[e.to] && e.cap > e.flow) {
41                      vis[e.to] = 1;
```

```
                    d[e.to] = d[x] + 1;
                    Q.push(e.to);
                }
            }
        }
        return vis[t];
    }
    int DFS(int x, int a) { //使用DFS从S出发，沿着d值严格递减的顺序进行多路增广
        if(x == t || a == 0) return a;
        int flow = 0, f;
        for(int i = 0; i < G[x].size(); ++i) {
            Edge &e = edges[G[x][i]];
            if(d[x] + 1 == d[e.to] && (f = DFS(e.to, min(a, e.cap - e.flow))) > 0)
{
                e.flow += f;
                edges[G[x][i] ^ 1].flow -= f;
                flow += f;
                a -= f;
                if(a == 0) break;
            }
        }
        return flow;
    }
    int Maxflow(int s, int t) {
        this->s = s;
        this->t = t;
        int flow = 0;
        while(BFS())
            flow += DFS(s, INF);
        return flow;
    }
};
Dinic g;
```

3.3.全局最小割（Stoer-Wagner algorithm）

（我不会用orz

```
// powered by Curiosity
// write by Unknown
// implementation of Stoer-Wagner algorithm
// O(V^3)
//usage
// MinCut mc;
// mc.init(n);
// for (each edge) mc.addEdge(a,b,weight);
// mincut = mc.solve();
// mc.cut = {0,1}^n describing which side the vertex belongs to.
struct MinCutMatrix
{
    typedef int cap_t;
    int n;
    vector<vector<cap_t>> graph;
```

```
16      void init(int _n) {
17          n = _n;
18          graph = vector<vector<cap_t>>(n, vector<cap_t>(n, 0));
19      }
20      void addEdge(int a, int b, cap_t w) {
21          if (a == b) return;
22          graph[a][b] += w;
23          graph[b][a] += w;
24      }
25      pair<cap_t, pair<int, int>> stMinCut(vector<int> &active) {
26          vector<cap_t> key(n);
27          vector<int> v(n);
28          int s = -1, t = -1;
29          for (int i = 0; i < active.size(); i++) {
30              cap_t maxv = -1;
31              int cur = -1;
32              for (auto j : active) {
33                  if (v[j] == 0 && maxv < key[j]) {
34                      maxv = key[j];
35                      cur = j;
36                  }
37              }
38              t = s; s = cur;
39              v[cur] = 1;
40              for (auto j : active) key[j] += graph[cur][j];
41          }
42          return make_pair(key[s], make_pair(s, t));
43      }
44      vector<int> cut;
45      cap_t solve() {
46          cap_t res = numeric_limits <cap_t>::max();
47          vector<vector<int>> grps;
48          vector<int> active;
49          cut.resize(n);
50          for (int i = 0; i < n; i++) grps.emplace_back(1, i);
51          for (int i = 0; i < n; i++) active.push_back(i);
52          while (active.size() >= 2) {
53              auto stcut = stMinCut(active);
54              if (stcut.first < res) {
55                  res = stcut.first;
56                  fill(cut.begin(), cut.end(), 0);
57                  for (auto v : grps[stcut.second.first]) cut[v] = 1;
58              }
59              int s = stcut.second.first, t = stcut.second.second;
60              if (grps[s].size() < grps[t].size()) swap(s, t);
61              active.erase(find(active.begin(), active.end(), t));
62              grps[s].insert(grps[s].end(), grps[t].begin(), grps[t].end());
63              for (int i = 0; i < n; i++) { graph[i][s] += graph[i][t]; graph[i][t] =
0; }
64              for (int i = 0; i < n; i++) { graph[s][i] += graph[t][i]; graph[t][i] =
0; }
65              graph[s][s] = 0;
66          }
67          return res;
```

```
68     }
69 };
```

3.4.有向图k短路

```
  1 // 完全k短路，求有向图中起点ss到终点tt的第k短路
  2 // 支持自环，重边，非负边，不连通图
  3 // 可并堆是通过随机化实现的，如果被卡的话就换成左偏树(红书有板)
  4 // 这里的所有路径，是包括单点的，即若ss==tt时单点也算路径，可能需要根据题目具体修改
  5 // 同时，这里的路径上每个点是可以重复走的(不过不重复走好像也做不了)
  6 //
  7 // 若是无向图，照着稍微修改一下也能做，但要注意同一条边的编号不要维护错
  8 // (大概需要单独再开一个数组存编号，然后就不需要再分正向/反向边了)
  9 //
 10 // 如果要求k长路的话，首先要保证起点到终点的路径不会成环(不会出现无穷长路)
 11 // 这样才能保证终点tt到其他点有最长路
 12 // 然后要把可并堆改成最大堆，同时优先队列中传入的权值也不能传相反数
 13 // 以及用INF判连通性的部分，也要改(本来用INF就是个偷懒的写法)
 14 //
 15 // 代码有很多细节(各种正负号什么的，变量名还长)，小心别敲错
 16 #include<cstdio>
 17 #include<cstdlib>
 18 #include<cstring>
 19 #include<cmath>
 20 #include<ctime>
 21 #include<algorithm>
 22 #include<queue>
 23
 24 using namespace std;
 25
 26 #define MP make_pair
 27 #define fi first
 28 #define se second
 29 typedef long long LL;
 30
 31 const LL INF = 1LL<<60; // 无穷大还是有点用的...
 32 const int maxn = 1000; // 点数上界
 33 const int maxE = 100000; // (有向边) 边数上界
 34 const int maxnode = 500000 * 10; // 可并堆结点个数上界，在不影响性能的前提下尽量开大
 35
 36 struct Node {
 37     int l, r;
 38     LL val; // 最短路增加值 (用心体会)
 39     int to; // 接下来会到达的点
 40     Node() {}
 41     Node(LL _val, int _to) : l(0), r(0), val(_val), to(_to) {}
 42 } heap[maxnode + 5];
 43 int tot = 0, rt[maxn+5];
 44
 45 int Merge(int x, int y) {
 46     if(!x || !y) return x|y;
 47     if(heap[x].val > heap[y].val) swap(x,y); // 由于是求最短路，所以是最小堆
 48     int t = ++tot;
```

```
49      heap[t] = heap[x];
50      (rand()&1)? (heap[t].l= Merge(heap[t].l,y)) : (heap[t].r= Merge(heap[t].r,y));
51      return t;
52  }
53
54  int n, m; // n个结点, m条有向边
55  LL dist[maxn+5];
56  priority_queue< pair<LL,int> > q1;
57  queue<int> q2;
58
59  int top = 0;
60  int h1[maxE+5], e1[maxE+5], w1[maxE+5], nx1[maxE+5]; // 正向边
61  int h2[maxE+5], e2[maxE+5], w2[maxE+5], nx2[maxE+5]; // 反向边
62  bool flag[maxE+5]; // 用于标记对应编号的边是否属于最短路树边
63
64  // 同时添加有向图中正向和反向边
65  inline void AddEdge(int u, int v, int d) {
66      ++top, flag[top] = 0; // flag[]必须要清空
67      e1[top] = v, w1[top] = d, nx1[top] = h1[u], h1[u] = top;
68      e2[top] = u, w2[top] = d, nx2[top] = h2[v], h2[v] = top;
69  }
70
71  // 根据反向边计算x到其他所有点的最短路
72  void Dij(int x) {
73      while(!q1.empty()) q1.pop(); // 清空q1
74      for(int i = 1; i <= n; ++i)  dist[i] = INF;
75      dist[x] = 0, q1.push( MP(-0LL,x) );
76      while(!q1.empty()) {
77          LL d = -q1.top().fi;
78          int x = q1.top().se;
79          q1.pop();
80          if(dist[x] != d) continue;
81          for(int i = h2[x]; i; i = nx2[i]) {
82              if(w2[i]+dist[x] < dist[e2[i]]) {
83                  dist[e2[i]] = w2[i] + dist[x];
84                   // 因要求最短路, 所以要用相反数作为优先队列权值
85                  q1.push( MP( -dist[e2[i]], e2[i] ) );
86              }
87          }
88      }
89  }
90
91  // 求ss到tt的第kk短路, kk >= 1
92  // 无解返回-1
93  LL KthPath(int ss, int tt, int kk) {
94      // 清除树边标记, 若要做多次k短路则每次都需要清标记
95      // REP(i,1,m) flag[i] = 0;
96
97      Dij(tt); // 根据反向边计算tt到其他所有点的最短路
98
99      // 构建最短路树
100     for(int x = 1; x <= n; ++x) {
101         if(x==tt || dist[x]==INF) continue;
```

```
102         // 注意此时应使用正向边判断
103         for(int i = h1[x]; i; i = nx1[i])
104             if(e1[i]!=x && dist[e1[i]]!=INF && dist[e1[i]]+w1[i]==dist[x]) {
105                 flag[i] = 1; break;
106             }
107     }
108
109     // 根据最短路树构建可并堆
110     for(int i = 1; i <= n; ++i) rt[i] = 0; // 清堆
111     tot = 0, q2.push(tt);
112     while(!q2.empty()) {
113         int x = q2.front(); q2.pop();
114         if(dist[x] != INF) { // 只从非树边扩展
115             for(int i = h1[x]; i; i = nx1[i]) if(flag[i]==0 && dist[e1[i]]!=INF) {
116                 heap[++tot] = Node( dist[e1[i]]+w1[i] - dist[x], e1[i] );
117                 rt[x] = Merge(rt[x], tot);
118             }
119         }
120         for(int i = h2[x]; i; i = nx2[i]) if(flag[i]) {
121             rt[e2[i]] = rt[x], q2.push(e2[i]);
122         }
123     }
124
125     // 以下开始求kk短路，无解返回-1
126     if(dist[ss] == INF) return -1LL;
127     //if(ss == tt) ++kk;
128     if(kk == 1) return dist[ss]; // 最短路也是一条路径
129     --kk;
130     while(!q1.empty()) q1.pop();
131     if(rt[ss])
132         q1.push( MP( -(dist[ss]+heap[rt[ss]].val), rt[ss]) );
133     while(!q1.empty() && (kk--)) {
134         LL d = -q1.top().fi;
135         int x = q1.top().se;
136         q1.pop();
137         if(kk == 0) return d;
138         if(rt[heap[x].to])
139             q1.push( MP( -(d+heap[rt[heap[x].to]].val), rt[heap[x].to] ) );
140         d -= heap[x].val;
141         if(heap[x].l)
142             q1.push( MP( -(d+heap[heap[x].l].val), heap[x].l ) );
143         if(heap[x].r)
144             q1.push( MP( -(d+heap[heap[x].r].val), heap[x].r ) );
145     }
146     return -1LL;
147 }
148
149 int main() {
150     srand(time(NULL)); // 一定要初始化随机数，不然可并堆复杂度就炸了
151     //srand(19260817);
152     while(~scanf("%d%d", &n, &m)) {
153         // 以下为初始化部分
154         tot = top = 0; // 注意区分 tot 和 top
155         for(int i = 1; i <= n; ++i) h1[i] = h2[i] = 0;
```

```
156
157         // 读入图, AddEdge()中有清除flag[]标记
158         for(int i = 1, u,v,d; i <= m; ++i) {
159             scanf("%d%d%d", &u, &v, &d);
160             AddEdge(u,v,d);
161         }
162         int ss, tt, kk; // 起点ss; 终点tt; 第kk短路;
163         scanf("%d%d%d", &ss, &tt, &kk);
164
165         printf("%lld\n", KthPath(ss,tt,kk));
166     }
167     return 0;
168 }
```

3.5.图论相关理论

- Hall定理

考虑二分图 $G = < V_1, V_2, E >$，设 $|V_1| \le |V_2|$，则 G 存在完美匹配当且仅当 V_1 中任意 $k (k = 1,2,\ldots,|V_1|)$ 个点至少与 V_2 中 k 个（不同的）顶点相邻。

- 平面图欧拉公式

设 V 是结点个数，E 是边的数目，F 是面（即所谓的单联通区域，**包括外部无穷大平面**）的个数，C 是联通块个数，则有如下公式：

$$V - E + F = C + 1$$

4. 数据结构

4.1.树状数组

```
//树状数组
const int max_range = 10000;

//单点修改，单点查询的树状数组模板
struct SeqArray {
    int top;
    LL node[max_range+5];
    void add(int x, LL del) {
        for(; x <= top; x += x&(-x)) node[x] += del;
    }
    LL sum(int x) {
        LL res = 0;
        for(; x; x -= x&(-x)) res += node[x];
        return res;
    }
};

//区间修改，单点查询
SeqArray T1;

void update(int l, int r, LL del) {
    T1.add(l, del), T1.add(r+1, -del);
}

LL query(int x){ return T1.sum(x); }

//区间修改，区间查询（要注意指定树状数组的大小）
SeqArray T1, T2;

void update(int l, int r, LL del) {
    T1.add(l, del), T1.add(r+1, -del, T0);
    T2.add(l, del*l), T2.add(r+1, -del*(r+1));
}

LL sum(int x) {
    return T1.sum(x)*(x+1) - T2.sum(x);
}

LL query(int l, int r) {
    return sum(r) - sum(l-1);
}

//二维树状数组，单点修改，单点查询（有效下标均从1开始）
void Add(int i, int j, int del) {
    for(; i <= MaxRow; i += i&(-i)) {
        int jj = j;
        for(; jj <= MaxCol; jj += jj&(-jj))
```

```
48              cnt[i][jj] += del;
49      }
50  }
51
52  int Sum(int i, int j) {
53      int res = 0;
54      for(; i > 0; i -= i&(-i)) {
55          int jj = j;
56          for(; jj > 0; jj -= jj&(-jj))
57              res += cnt[i][jj];
58      }
59      return res;
60  }
61
62  //二维树状数组，区间修改，区间查询（要注意指定树状数组的大小）
63  struct Tree {
64      SeqArray T1, T2, T3, T4;
65
66      void _update(int x, int y, int del) {
67          T1.add(x, y, del);
68          T2.add(x, y, del*x);
69          T3.add(x, y, del*y);
70          T4.add(x, y, del*x*y);
71      }
72      void update(int x1, int y1, int x2, int y2, LL del) {
73          _update(x1, y1, del);
74          _update(x1, y2+1, -del);
75          _update(x2+1, y1, -del);
76          _update(x2+1, y2+1, del);
77      }
78      LL _query(int x, int y) {
79          return T1.sum(x,y)*x*y - T2.sum(x,y)*(y+1) - T3.sum(x,y)*(x+1) +
T4.sum(x,y);
80      }
81      LL query(int x1, int y1, int x2, int y2) {
82          return _query(x2,y2) - _query(x1-1,y2) - _query(x2,y1-1) +
_query(x1-1,y1-1);
83      }
84  };
```

4.2.RMQ

```
 1  LL rmq[maxn + 5][MaxStep + 2];
 2  void InitRMQ(LL *a, int len) { //arr[]爲原始數據，len为数组长度
 3      for(int i = 1; i <= len; ++i) {
 4          rmq[i][0] = a[i];
 5          for(int k = 1; (1 << k) <= i; ++k)
 6              rmq[i][k] = min(rmq[i][k-1], rmq[i - (1 << (k-1))][k-1]); //自行修改比较
方式
 7      }
 8  }
 9
10  LL RMQ(int l, int r) {
```

```
11      int i;
12      for(i = 0; (1 << (i+1)) < (r-l+1); ++i);
13      return min(rmq[r][i], rmq[l + (1<<i) - 1][i]); //根據需要修改比較方式
14  }
```

4.3.可持久化可并堆

```
 1  // 可持久化可并堆，注意要srand(time(NULL))
 2  struct Node {
 3      int l, r, v; // l(r)保存左(右)结点的编号，为0表示不存在左(右)孩子；v保存权值
 4  } heap[maxn + 5];
 5  int tot = 0; // 结点从1开始编号，0表示不存在
 6
 7  // 将根结点编号为x，y的两个堆合并，返回合并后的堆的根结点编号
 8  int Merge(int x, int y) {
 9      if(!x || !y) return x|y;
10      if(heap[x].v < heap[y].v) swap(x,y); // 此为最大堆
11      int t = ++tot;
12      heap[t] = heap[x];
13      (rand()&1) ? (heap[t].l = Merge(heap[t].l,y)) : (heap[t].r =
Merge(heap[t].r,y));
14      return t;
15  }
```

4.4.伸展树（Splay）

```
 1  // bzoj-3223: Tyvj 1729 文艺平衡树
 2  // splay模板题，主要操作为区间翻转
 3  //
 4  // Splay的核心思想就是要把最近访问过的结点移到根结点上
 5  // 所以无论什么样的扩展操作，只要涉及到查询某个结点x
 6  // 操作到最后都要执行Splay(x,0)
 7  #include <cstdio>
 8  #include <cassert>
 9  #include <algorithm>
10  using namespace std;
11
12  const int maxn = 1e5 + 10;
13
14  // 结点不存在时对应编号为0
15  // 需保证ch[0][0]，ch[0][1]，fa[0]这三个变量恒为0
16  int root = 0, tot = 0; // root保存根结点编号
17  int val[maxn+5], sz[maxn+5]; // 需要维护的结点信息(例子)，val为结点值，sz为子树大小
18  int ch[maxn+5][2], fa[maxn+5];
19  bool rev[maxn+5]; // 用于支持区间翻转
20
21  // 根据左右孩子更新结点x保存的值，可自行扩展
22  void PushUp(int x) {
23      sz[x] = sz[ch[x][0]] + sz[ch[x][1]] + 1;
24  }
25
26  // 下放lazy标记
27  void PushDown(int x) {
```

```
28      if(rev[x]) {
29          swap(ch[x][0], ch[x][1]), rev[x] = 0;
30          if(ch[x][0]) rev[ch[x][0]] ^= 1;
31          if(ch[x][1]) rev[ch[x][1]] ^= 1;
32      }
33  }
34
35  // 旋转操作，可以自己判断执行左旋还是右旋
36  void Rotate(int x) {
37      int y = fa[x], d = ch[y][1]==x;
38      ch[y][d] = ch[x][d^1];
39      if(ch[x][d^1]) fa[ch[x][d^1]] = y;
40      if(fa[y]) { // 防止进行非法修改
41          int z = fa[y];
42          if(ch[z][0] == y) ch[z][0] = x;
43          if(ch[z][1] == y) ch[z][1] = x;
44      }
45      fa[x] = fa[y], fa[y] = x, ch[x][d^1] = y;
46      PushUp(y); // 减少PushUp()操作次数
47  }
48
49  // Splay操作，作用是把结点x旋转到target的孩子上
50  int top, sta[maxn+5];
51  void Splay(int x, int target) {
52      // 下放lazy标记
53      sta[top = 1] = x;
54      for(int i = x; i != target; i = fa[i]) sta[++top] = fa[i];
55      while(top>0) PushDown(sta[top--]);
56      // fa[p] != target: 很关键啊，不然就会执行非法旋转了
57      while(fa[x] != target) {
58          int y = fa[x];
59          if(fa[y] != target) {
60              if((ch[fa[y]][0]==y)^(ch[y][0]==x)) Rotate(x);
61              else Rotate(y);
62          }
63          Rotate(x);
64      }
65      PushUp(x); // 用于减少PushUp()执行次数，所以最后一次PushUp()必须执执行
66      // 如果target = 0，意味着要把x旋转变成root结点，因此这里要更新root
67      if(fa[x] == 0) root = x;
68  }
69
70  // 找排序在kth的结点，并返回对应结点的编号(相当于指针)
71  int Kth(int k) {
72      int x = root;
73      while(x) {
74          PushDown(x);
75          if(k == sz[ch[x][0]]+1) return x;
76          if(k <= sz[ch[x][0]]) x = ch[x][0];
77          else k -= sz[ch[x][0]]+1, x = ch[x][1];
78      }
79      assert(false); // 用于调试
80      return 0;
81  }
```

```

int n, m, ans[maxn+5];

// 翻转区间[l,r], 原理简单易懂
void Reverse(int l, int r) {
    // 为了实现方便, 所以插入了 0th 和 (n+1)th
    // 因此 (l-1)th 和 (r+1)th 实际对应的是(l-1+1), (r+1+1)
    int x = Kth(l-1+1), y = Kth(r+1+1);
    Splay(x,0), Splay(y,root);
    rev[ch[ch[x][1]][0]] ^= 1;
}

// 建立初始树, 尽可能平衡
int Build(int p, int l, int r) {
    int x = ++tot, mid = (l+r)>>1;
    fa[x] = p, sz[x] = r-l+1, val[x] = mid;
    if(l<=mid-1) ch[x][0] = Build(x, l, mid-1);
    if(mid+1<=r) ch[x][1] = Build(x, mid+1, r);
    return x;
}

// 计算答案
void dfs(int x, int d) {
    PushDown(x);
    ans[ sz[ch[x][0]]+1+d-1 ] = val[x];
    if(ch[x][0]) dfs(ch[x][0], d);
    if(ch[x][1]) dfs(ch[x][1], d+sz[ch[x][0]]+1);
}

int main() {
    // root = 0, tot = 0; // 重置伸展树
    scanf("%d%d", &n, &m);
    root = Build(0, 0, n+1);
    while(m--) {
        int l, r; scanf("%d%d", &l, &r);
        Reverse(l,r);
    }
    dfs(root, 0);
    for(int i = 1; i <= n; ++i)
        printf("%d%c", ans[i], (i==n)?'\n':' ');
    return 0;
}
```

4.5.动态树（LCT）

```
// uoj-274 【清华集训2016】温暖会指引我们前行
// lct动态维护最大生成树
// lct核心思想是将树剖成一些链, 通过lct按深度维护这些链
#include<bits/stdc++.h>
using namespace std;
#define mem(a,b) memset(a,b,sizeof(a))
#define REP(i,a,b) for(int i=a; i<=b; ++i)
#define PER(i,a,b) for(int i=a; i>=b; --i)
```

```
 9 #define MP make_pair
10 #define PB push_back
11 #define fi first
12 #define se second
13 typedef long long LL;
14 typedef pair<int,int> pii;
15
16 const int maxn = 4e5;
17
18 int ch[maxn+5][2], fa[maxn+5];
19 int val[maxn+5], mi[maxn+5], len[maxn+5], sum[maxn+5];
20 bool rev[maxn+5]; // 区间翻转, 主要用于实现快速换根(MakeRoot)
21
22 // 判断当前结点是否是一个树链的根
23 inline bool IsRoot(int x) { return (ch[fa[x]][0]!=x && ch[fa[x]][1]!=x); }
24 inline void PushUp(int x) {
25     sum[x] = sum[ch[x][0]] + sum[ch[x][1]] + len[x];
26     mi[x] = min(val[x], min(mi[ch[x][0]], mi[ch[x][1]]));
27 }
28 inline void PushDown(int x) {
29     if(rev[x]) {
30         rev[x] = 0;
31         swap(ch[x][0], ch[x][1]);
32         if(ch[x][0]) rev[ch[x][0]] ^= 1;
33         if(ch[x][1]) rev[ch[x][1]] ^= 1;
34     }
35 }
36
37 // 快速旋转
38 void Rotate(int x) {
39     int y = fa[x], d = ch[y][1]==x;
40     ch[y][d] = ch[x][d^1];
41     if(ch[x][d^1]) fa[ch[x][d^1]] = y;
42     if(fa[y]) {
43         int z = fa[y];
44         if(ch[z][0] == y) ch[z][0] = x;
45         if(ch[z][1] == y) ch[z][1] = x;
46     }
47     fa[x] = fa[y], fa[y] = x, ch[x][d^1] = y;
48     PushUp(y);
49 }
50
51 // 一般lct中的splay只需要将结点旋转到伸展树的根即可
52 // 因此target是不必要的, 可以删掉
53 int top, sta[maxn+5];
54 void Splay(int x, int target = 0) {
55     sta[top = 1] = x;
56     for(int i = x; !IsRoot(i) && fa[x]!=target; i = fa[i]) sta[++top] = fa[i];
57     while(top>0) PushDown(sta[top--]);
58     while(!IsRoot(x) && fa[x]!=target) {
59         int y = fa[x];
60         if(!IsRoot(y) && fa[y]!=target)
61             Rotate( ((ch[fa[y]][0]==y)^(ch[y][0]==x)) ? x : y );
62         Rotate(x);
```

```
 63     }
 64     PushUp(x);
 65 }
 66
 67 // lct的核心操作是Access(x)，即为将x和它当前的根单独剖成一条链，这时x是当前链最深的结点
 68 // Split(x,y)是提出x到y的路径作为单独的一条链，这时x是根，而y是这条链最深的结点且被Splay()到
伸展树的根
 69 // 其他操作简单明了，在此不做赘述
 70 void Access(int x) { for(int y = 0; x; y = x, x = fa[x]) Splay(x), ch[x][1] = y,
PushUp(x); }
 71 void MakeRoot(int x) { Access(x); Splay(x); rev[x]^=1; }
 72 int FindRoot(int x) { Access(x); Splay(x); while(ch[x][0]) { x = ch[x][0],
PushDown(x); } return x; }
 73 void Split(int x, int y) { MakeRoot(x); Access(y); Splay(y); }
 74 void Cut(int x, int y) { Split(x,y); if(ch[y][0] == x) ch[y][0] = 0, PushUp(y),
fa[x] = 0; }
 75 void Link(int x, int y) { Split(x,y); if(ch[y][0] != x) fa[x] = y; }
 76
 77 int fp[maxn+5]; // 使用并查集判连通性(FindRoot()太慢了)
 78 int find(int x) { return fp[x]==x ? x : fp[x]=find(fp[x]); }
 79
 80 int main() {
 81     val[0] = mi[0] = 1e9+10; // 为了实现方便而设置的边界条件
 82     int n, m; scanf("%d%d", &n, &m);
 83     REP(i,1,n) val[i] = mi[i] = 1e9 + 10, fp[i] = i;
 84     char ord[10];
 85     while(m--) {
 86         scanf("%s", ord);
 87         if(ord[0] == 'f') {
 88             int id, u, v, t, l;
 89             scanf("%d%d%d%d%d", &id, &u, &v, &t, &l);
 90             id += n+1, ++u, ++v;
 91             val[id] = mi[id] = t, len[id] = sum[id] = l;
 92             if(find(u) != find(v)) {
 93                 fp[find(u)] = find(v);
 94                 // 以下操作将边id视为一个结点，并把其对应的两个端点连到自身
 95                 MakeRoot(u);
 96                 MakeRoot(v);
 97                 fa[v] = id, fa[u] = id;
 98             }
 99             else {
100                 Split(u,v);
101                 if(mi[v] >= t) continue; // Split(u,v)后v在平衡树根，因此可以进行这样的
操作
102                 int key = mi[v];
103                 int x = v;
104                 while(1) {
105                     if(val[x] == key) break;
106                     if(ch[x][0] && mi[ch[x][0]] == key) x = ch[x][0];
107                     else x = ch[x][1];
108                 }
109                 Splay(x); // 重要，x必须要是伸展树根才能进行如下查找
110                 int y = ch[x][0], z = ch[x][1];
111                 while(PushDown(y), ch[y][1]) y = ch[y][1];
```

```
112                 while(PushDown(z), ch[z][0]) z = ch[z][0];
113                 // 以下操作将边x和它对应的两个端点y,z之间的边断掉(即断掉了<x,y>, <x,z>)
114                 MakeRoot(x);
115                 Access(y), Splay(y), fa[x] = 0, ch[y][0] = 0, PushUp(y);
116                 Access(z), Splay(z), fa[x] = 0, ch[z][0] = 0, PushUp(z);
117                 // 以下操作将u,v连到id
118                 MakeRoot(u), MakeRoot(v);
119                 fa[u] = id, fa[v] = id;
120             }
121         }
122         else if(ord[0] == 'm') {
123             int u, v;
124             scanf("%d%d", &u, &v);
125             ++u, ++v;
126             if(find(u) != find(v)) puts("-1");
127             else {
128                 Split(u,v);
129                 printf("%d\n", sum[v]);
130             }
131         }
132         else {
133             int id, l;
134             scanf("%d%d", &id, &l);
135             id += n+1;
136             // 以下操作用于修改边id的距离
137             Splay(id);
138             len[id] = l; PushUp(id);
139         }
140     }
141     return 0;
142 }
```

4.6.KD-Tree

```
1  // hdu-5992 KD-Tree模板题(其实是因为数据太弱)
2  // 对于每个询问, 找出价格不超过给定值的欧氏距离最近结点的编号
3  //
4  // 只有建树和查找部分
5  // KD-Tree查找方法的本质就是搜索剪枝+启发式乱搞
6  // 插入直接类似建树瞎模拟一遍即可
7  // 插入可能需要通过替罪羊树实现高度平衡
8  // 简单来说就是插入后找到最浅层的不平衡度超过阈值的点(即sz1>sz2*a, a一般取0.7)
9  // 然后dfs出该结点对应子树的所有结点编号并存在num[]中, 然后使用Build()重建子树
10 // 然而替罪羊树在实际应用中相当慢, 所以不到万不得已时尽量别使用
11 //
12 // 此模板仅用于比赛时回忆KD-Tree的实现细节, 应尽量做到能裸敲KD-Tree
13 #include<bits/stdc++.h>
14 using namespace std;
15
16 #define REP(i,a,b) for(int i=a; i<=b; ++i)
17 typedef long long LL;
18 const int maxn = 2e5;
19
```

```
20 struct Node {
21     int x[2], mi[2], mx[2], mic, c, id, l, r;
22 } tr[maxn+5];
23 int D, num[maxn+5]; // num[]保存将要用于建树的结点编号
24
25 // 对num[]进行比较，注意这里的大写D别打错
26 bool cmp(const int &a, const int &b) { return tr[a].x[D] < tr[b].x[D]; }
27
28 int Build(int l, int r, int d) {
29     int mid = (l+r) >> 1;
30     D = d; // 用于cmp()排序
31     nth_element(num+l, num+mid, num+r+1, cmp);
32     int x = num[mid]; // 注意num[mid]才是实际上的结点编号
33     tr[x].mi[0] = tr[x].mx[0] = tr[x].x[0];
34     tr[x].mi[1] = tr[x].mx[1] = tr[x].x[1];
35     tr[x].mic = tr[x].c;
36     tr[x].l = tr[x].r = 0;
37     if(l<mid) {
38         tr[x].l = Build(l, mid-1, d^1); // 使用(d^1)维对下一层建树
39         REP(i,0,1) {
40             tr[x].mi[i] = min(tr[x].mi[i], tr[tr[x].l].mi[i]);
41             tr[x].mx[i] = max(tr[x].mx[i], tr[tr[x].l].mx[i]);
42         }
43         tr[x].mic = min(tr[x].mic, tr[tr[x].l].mic);
44     }
45     if(mid<r) {
46         tr[x].r = Build(mid+1, r, d^1);
47         REP(i,0,1) {
48             tr[x].mi[i] = min(tr[x].mi[i], tr[tr[x].r].mi[i]);
49             tr[x].mx[i] = max(tr[x].mx[i], tr[tr[x].r].mx[i]);
50         }
51         tr[x].mic = min(tr[x].mic, tr[tr[x].r].mic);
52     }
53     return x;
54 }
55
56 LL dist;
57 int pid;
58 inline LL sqr(LL x) { return x*x; }
59 void Query(int x, Node &pt) {
60     if(tr[x].c <= pt.c) {
61         LL tmp = 0;
62         REP(i,0,1) tmp += sqr(tr[x].x[i]-pt.x[i]);
63         if(tmp < dist) dist = tmp, pid = tr[x].id;
64         else if(tmp == dist && tr[x].id < pid) pid = tr[x].id;
65     }
66     LL lc = 0, rc = 0;
67     int l = tr[x].l, r = tr[x].r;
68
69     // 搜索剪枝+启发式乱搞
70     // 因为此题求的是欧式距离，所以启发式的函数长下面这样
71     if(l) REP(i,0,1) lc += sqr( max(0, tr[l].mi[i]-pt.x[i]) + max(0, pt.x[i]-
tr[l].mx[i]) );
```

```
 72     if(r) REP(i,0,1) rc += sqr( max(0, tr[r].mi[i]-pt.x[i]) + max(0, pt.x[i]-
tr[r].mx[i]) );
 73     if(lc < rc) {
 74         // 在剪枝时一定要仔细思考，不要把可能的正确答案也剪掉
 75         if(l && tr[l].mic<=pt.c && lc<=dist) Query(l, pt);
 76         if(r && tr[r].mic<=pt.c && rc<=dist) Query(r, pt);
 77     }
 78     else {
 79         if(r && tr[r].mic<=pt.c && rc<=dist) Query(r, pt);
 80         if(l && tr[l].mic<=pt.c && lc<=dist) Query(l, pt);
 81     }
 82 }
 83
 84 int n, m, xx[maxn+5], yy[maxn+5], cc[maxn+5];
 85 int main() {
 86     int _; scanf("%d", &_);
 87     while(_--) {
 88         scanf("%d%d", &n, &m);
 89         REP(i,1,n) {
 90             scanf("%d%d%d", xx+i, yy+i, cc+i);
 91             tr[i].x[0] = xx[i], tr[i].x[1] = yy[i];
 92             tr[i].c = cc[i], tr[i].id = i;
 93             num[i] = i; // 一定要把用于建树的结点编号存入num[]中
 94         }
 95         int root = Build(1,n,0);
 96         while(m--) {
 97             int x, y, c; scanf("%d%d%d", &x, &y, &c);
 98             Node nd;
 99             nd.x[0] = x, nd.x[1] = y, nd.c = c;
100             dist = 1LL << 60, pid = 0;
101             Query(root, nd);
102             printf("%d %d %d\n", xx[pid], yy[pid], cc[pid]);
103         }
104     }
105     return 0;
106 }
```

4.7.虚树

```
 1 // 通过dfs序来建立虚树
 2 // 这种维护关键点dfs序的思想还有很多其他的应用
 3 int DFN[maxn + 5]; // 存储dfs序号
 4 int dep[maxn + 5]; // 存储结点深度，需保证虚树根结点编号最小
 5 vector<int> seg; // seg存储需要包含的点
 6 int sta[maxn + 5], top = 0;
 7 const bool cmp(const int &a, const int &b) { return DFN[a] < DFN[b]; }
 8 void Build() {
 9     sort(seg.begin(), seg.end(), cmp); // 按照dfs序排序
10     sta[top = 1] = 0; // 选取一个不在原树中的点作为虚树根结点，保证该结点深度最小
11     for(int i = 0; i < seg.size(); ++i) {
12         int lca = LCA(sta[top], seg[i]); //注意此处LCA的点可能不在原树中，要特判
13         while(top >= 2 && dep[sta[top-1]] >= dep[lca])
14             AddEdge(sta[top-1], sta[top]), --top; //AddEdge时记得清空原虚树图
```

```
        if(sta[top] != lca)
            AddEdge(lca, sta[top]), sta[top] = lca;
        sta[++top] = seg[i];
    }
    while(top >= 2)
        Addedge(sta[top-1], sta[top]), --top;
}
```

4.8.笛卡尔树

```
// O(n)构建笛卡尔树
// A[]存储1~n的一个排列，sta[]是一个栈，存储当前构造出的树的右链
// 每个结点意义为：当前结点是其所在区间的最大值，同时改把区间分成不相交的两部分
// 此模板构造的笛卡尔树存在最大堆性质，可自行修改比较方式
int n, A[maxn + 5], sta[maxn + 5];
int Build() {
    int top = 0;
    for(int i = 1; i <= n; ++i) {
        L[ A[i] ] = R[ A[i] ] = 0;
        int p = top;
        while(p && sta[p] < A[i]) --p; // 可修改比较方式
        if(p>0) R[ sta[p] ] = A[i];
        if(p<top) L[ A[i] ] = sta[p+1];
        sta[++p] = A[i]; top = p;
    }
    return sta[1];
}
```

4.9.树分治找重心

```
// 树分治找重心，结点下标从1开始
vector<int> G[maxn+5];
int que[maxn+5], fa[maxn+5], sz[maxn+5], msz[maxn+5];
bool ban[maxn+5];
int FindRoot(int x) {
    int s = 1, t = 1;
    que[1] = x, fa[x] = 0; // 结点下标从1开始
    while(s <= t) {
        x = que[s++], sz[x] = 1, msz[x] = 0;
        for(auto v : G[x])
            if(!vist[v] && v!=fa[x])
                que[++t] = v, fa[v] = x;
    }
    for(int i = t; i >= 1; --i) {
        x = que[i], msz[x] = max(msz[x], t-sz[x]);
        // 判断重心的写法比较特殊
        // 这导致找出的点不一定是严格定义上的重心，但能满足分治的要求
        if((msz[x]<<1) <= t) return x;
        sz[fa[x]] += sz[x], msz[fa[x]] = max(msz[fa[x]], sz[x]);
    }
    assert(false); // 错误检测
    return 0;
}
```

5. 计算几何

5.1.几何定理

- 正弦定理&余弦定理

设有三角形 ABC，其中 $\angle A, \angle B, \angle C$ 的弧度分别为 α, β, γ，对应的对边边长分别为 $|BC| = a, |AC| = b, |AB| = c$，最后设该三角形的外接圆半径为 R。那么我们有如下定理：

$$\frac{a}{\sin\alpha} = \frac{b}{\sin\beta} = \frac{c}{\sin\gamma} = 2R \quad \text{（正弦定理）}$$

$$\begin{cases} a^2 = b^2 + c^2 - 2bc\cos\alpha \\ b^2 = a^2 + c^2 - 2ac\cos\beta \\ c^2 = a^2 + b^2 - 2ab\cos\gamma \end{cases} \quad \text{（余弦定理）}$$

- 海伦公式

设三角形的三边长分别为a, b, c，则对三角形的面积 S 有如下公式：

$$S = \sqrt{p(p-a)(p-b)(p-c)}, \quad p = \frac{a+b+c}{2}$$

5.2.一些基础函数以及Point类

```
 7 typedef double DB; // 方便随时修改
 8 const DB eps = 1e-10;
 9
10 // -1: a<0;  0: a==0;  1: a>0;
11 int Cmp(const DB &a) {
12     if(fabs(a) < eps) return 0;
13     return a<0 ? -1 : 1;
14 }
15
16 struct Point {
17     DB x, y;
18     Point() {}
19     Point(DB _x, DB _y) { x = _x, y = _y; }
20     DB operator * (const Point &a) const { return x*a.x + y*a.y; } // 点乘
21     DB operator ^ (const Point &a) const { return x*a.y - y*a.x; } // 叉乘
22     Point operator + (const Point &a) const { return Point(x+a.x, y+a.y); }
23     Point operator - (const Point &a) const { return Point(x-a.x, y-a.y); }
24     Point operator * (const DB &a) const { return Point(x*a, y*a); }
25     Point operator / (const DB &a) const { return Point(x/a, y/a); }
26     bool operator == (const Point &a) const { return Cmp(x-a.x) == 0 && Cmp(y-a.y)
== 0; }
27     DB Mod() const { return sqrt(x*x + y*y); }
28 };
```

5.3.三角形外心&点的最小圆覆盖

```
64 // 返回三角形外心，小心别抄错板子
65 Point Circumcenter(Point &a, Point &b, Point &c) {
66     DB a1 = b.x - a.x, b1 = b.y - a.y, c1 = (a1 * a1 + b1 * b1) / 2.0;
67     DB a2 = c.x - a.x, b2 = c.y - a.y, c2 = (a2 * a2 + b2 * b2) / 2.0;
68     DB d = a1 * b2 - a2 * b1;
69     return Point(a.x + (c1*b2 - c2*b1) / d, a.y + (a1*c2 - a2*c1) / d);
70 }
71
72 DB Dist(Point &a, Point & b) { return sqrt((a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-
b.y)); }
73
74 // 点的最小圆覆盖
75 void MinCoverCircle(vector<Point> &seg, Point &O, DB &R) {
76     // 将点随机排序，这样能保证总算法期望复杂度O(n)
77     random_shuffle(seg.begin(), seg.end());
78     int i, k, j, n = seg.size();
79     for(O = seg[0], R = 0, i = 1; i < n; ++i) if(Dist(O,seg[i]) > R+eps) {
80         for(O = seg[i], R = 0, k = 0; k < i; ++k) if(Dist(O,seg[k]) > R+eps) {
81             O = (seg[i] + seg[k]) / 2, R = Dist(O, seg[i]);
82             for(j = 0; j < k; ++j) if(Dist(O,seg[j]) > R+eps)
83                 O = Circumcenter(seg[i], seg[k], seg[j]), R = Dist(O,seg[i]);
84         }
85     }
86 }
```

5.4.直线交点&线段交点

```
33 // 直线相交
34 // 二元组<a,b>，<c,d>分别表示两条线段，现在要求这两条线段的交点res
35 // 原理看代码就明白了
36 bool Line_Intersection(Point &a, Point &b, Point &c, Point &d, Point &res) {
37     DB d1 = (a-c) ^ (d-c), d2 = (d-c) ^ (b-a);
38     if(Cmp(d2) == 0) return false; // 平行无交点
39     res = a + (b - a) * (d1 / d2);
40     // 即令a点向向量ab方向移动 (d1/d2*|ab|) 的距离
41     return true;
42 }
43
44 // 线段相交
45 // 此处的交点可以是线段的端点，若不能在端点处相交则可以省去很多判断
46 // 两条线段不能完全重合（总之就是要根据具体题目具体修改）
47 bool Segment_Intersection(Point &a, Point &b, Point &c, Point &d, Point &res) {
48     Point v0 = b-a, v1 = c-a, v2 = d-a;
49     if( Cmp(v0^v1) * Cmp(v0^v2) > 0) return false;
50     v0 = d-c, v1 = a-c, v2 = b-c;
51     if( Cmp(v0^v1) * Cmp(v0^v2) > 0) return false;
52     // 以上的判断得到的两条线段可能会在同一直线上
53     // 因此接下来要判断是否在端点处有交点
54     if(Cmp((d-c) ^ (b-a)) == 0) {
55         if(a == c || a == d) { res = a; return true; }
```

```
56          if(b == c || b == d) { res = b; return true; }
57          return false;
58      }
59
60      DB d1 = (a-c) ^ (d-c), d2 = (d-c) ^ (b-a);
61      res = a + (b - a) * (d1 / d2);
62      // 即令a点向向量ab方向移动 (d1/d2*|ab|) 的距离
63      return true;
64  }
```

5.5.几何坑点

1. 在使用反三角函数acos()，asin()时一定要检查输入值是否在函数值域([-1,1])内。
2. 对于输出答案为0的数，一定要手动判0，否则可能会输出”-0”导致PE或WA。
3. 尽可能减少sqrt或除法的使用以提高精度。
4. 浮点数应typedef一个DB类型，这样可以方便在卡精度时切换long double。
5. 对于需要控制相对误差(而不是绝对误差)的题，二分/三分时候应手动限制查找次数而不是使用while(r-l>eps)，否则可能由于数值太大引起精度丢失，最终陷入死循环(cf-1059D)。事实上，若答案数值过大都应手动限制查找次数。

6. 各种科技

6.1.Vim配置

```
set si nu ts=4 sw=4 sts=4 mouse=a
syn on

func! Compile()
    if &filetype == 'c'
        :!gcc -g % -o %<.exe
    elseif &filetype == 'cpp'
        :!g++ -std=gnu++11 -g % -o %<.exe
    elseif &filetype == 'java'
        :!javac -g %
    elseif &filetype == 'php'
        :!php -l %
    endif
endfunc

func! Run()
    if &filetype == 'c'
        :!time ./%<.exe
    elseif &filetype == 'cpp'
        :!time ./%<.exe
    elseif &filetype == 'java'
        :!time java -cp %:p:h %:t:r
    elseif &filetype == 'python'
        :!time python3 %
    elseif &filetype == 'php'
        :!time php %
    endif
endfunc


map <F4> :call Compile()<cr>
map <F5> :call Run()<cr>
map <C-A> ggVG"+y
```

6.2.头文件

```
#include<cstdio>
#include<cstdlib>
#include<cstring>
#include<cmath>
#include<ctime>
#include<iostream>
#include<algorithm>
#include<vector>
#include<string>
#include<queue>
#include<utility>
#include<bitset>
#include<complex>
```

```cpp
#include<map>
#include<set>
#include<unordered_map>
#include<unordered_set>

using namespace std;

#define mem(a,b) memset(a,b,sizeof(a))
#define REP(i,a,b) for(int i = a; i <= b; ++i)
#define PER(i,a,b) for(int i = a; i >= b; --i)
#define MP make_pair
typedef long long LL;
typedef pair<int,int> pii;

int main() {
    return 0;
}
```

6.3.C++ STL

1. lower_bound(first, last, val): 返回[first, last)中第一个 >= val 的数的指针，无解返回last;

2. upper_bound(first, last, val): 返回[first, last)中第一个 > val 的数的指针，无解返回last;

3. map, set自带lower_bound/upper_bound;

4. unique(first, last): [first, last)区间去重，并返回去重后的末尾指针last'(即去重后的数据存放在first ~ last-1);

5. bitset的方法: set(pos, val), flip(pos), reset(), count(), to_string(), to_ulong();

6. atan2(y,x): 计算 $\arctan\frac{y}{x}$ 在 $(-\pi,\pi]$ 内对应的值（即计算向量 (x,y) 对应的弧度大小）;

6.4.__int128解决爆long long

```cpp
/*
 * __int128 使用示例：hdu-6004节选（注意这里是两个下划线）
 * __int128 是一个至少存在于unix中的c/c++数据类型
 * 在某些计算中可能出现中间变量超出 long long 数据类型范围，
 * 但最终结果仍在 long long 范围中的情况；
 * 这时便可以考虑用 __int128 保存中间变量，算出结果后再变回 long long
 */
#include<bits/stdc++.h>
using namespace std;

typedef __int128 Lint;
typedef long long LL;
const int maxn = 200;

```

```
struct Equ {
    LL a, b;
    Equ(): a(-1), b(-1) {}
    Equ(LL _a, LL _b) { a = _a, b = _b; }
} pre[maxn+2][maxn*(maxn+1)/2+2];

Equ Calc(Equ &l, Equ &r) {
    if(l.a == -1 || r.a == -1) return Equ(-1, -1);
    LL x, y, g;
    LL p, m_a3;
    Lint a3, b3; //用于存储中间变量
    LL &a1 = l.a, &b1 = l.b, &a2 = r.a, &b2 = r.b;
    ex_gcd(b1, b2, g, x, y), y = -y;
    if((a2-a1)%g)
        return Equ(-1, -1);
    p = (a2-a1)/g;
    Lint tx = x, ty = y;
    tx *= p, ty *= p;
    m_a3 = max(a1, a2);
    a3 = a1 + b1*tx, b3 = b1 / g * b2; //此处可能超过 long long 范围
    if(a3 < m_a3)
        a3 += ((m_a3-a3)/b3 + 1) * b3;
    if(a3 >= m_a3)
        a3 -= ((a3-m_a3)/b3) * b3;
    return Equ(a3, b3); //此处将 __int128 又转变回 long long
}
```

6.5.BUAA输入挂

```
#include<cstdio>
//BUAA输入挂，可读__int128
namespace FastIO {
    #define BUF_SIZE 100000 //缓冲区大小可修改
    bool IOError = 0; //IOError == false 时表示处理到文件结尾
    inline char NextChar() {
        static char buf[BUF_SIZE], *p1 = buf + BUF_SIZE, *pend = buf + BUF_SIZE;
        if(p1 == pend) {
            p1 = buf;
            pend = buf + fread(buf, 1, BUF_SIZE, stdin);
            if(pend == p1) {
                IOError = 1;
                return -1;
            }
        }
        return *p1++;
    }
    inline bool Blank(char c) {
        return c == ' ' || c == '\n' || c == '\r' || c == '\t';
    }

    template<class T> inline void Read(T &x) {
        char c;
        while(Blank(c = NextChar()));
```

```
25          if(!IOError) {
26              for(x = 0; '0' <= c && c <= '9'; c = NextChar())
27                  x = (x << 3) + (x << 1) + c - '0';
28          }
29      }
30  }
```

6.6.Java基本输入输出

```
 1  import java.io.*;
 2  import java.util.*;
 3  import java.math.*;
 4
 5  public class Main {
 6      //public static void main(String[] args) throws FileNotFoundException {
 7      public static void main(String[] args) {
 8          //普通读入
 9          Scanner in = new Scanner( new BufferedInputStream(System.in) );
10          in.close(); // 终止语句
11
12          //判断文件末尾
13          while(in.hasNextInt());
14
15          //文件读入
16          Scanner in = new Scanner(new File("in.txt"));
17          PrintWriter out = new PrintWriter("out.txt");
18          //读入结束后执行如下代码
19          out.flush();
20          out.close();
21
22          //stdin快速度入，需结合InputReader食用
23          InputStream inputStream = System.in;
24          InputReader in = new InputReader(inputStream);
25
26          //快速输出到stdout
27          OutputStream outputStream = System.out;
28          PrintWriter out = new PrintWriter(outputStream);
29          out.close();
30      }
31
32      //stdin快速度入
33      static class InputReader {
34          public BufferedReader reader;
35          public StringTokenizer tokenizer;
36
37          public InputReader(InputStream stream) {
38              reader = new BufferedReader(new InputStreamReader(stream), 32768);
39              tokenizer = null;
40          }
41
42          public String next() {
43              while (tokenizer == null || !tokenizer.hasMoreTokens()) {
44                  try {
```

```
45                      tokenizer = new StringTokenizer(reader.readLine());
46                  } catch (IOException e) {
47                      throw new RuntimeException(e);
48                  }
49              }
50              return tokenizer.nextToken();
51          }
52
53          public int nextInt() {
54              return Integer.parseInt(next());
55          }
56      }
57  }
```

6.7.Java高精度

- 基本用法

```
1  //BigInteger
2  BigInteger a("123"); //通过字符串构造
3  c = BigInteger.valueOf(456); //通过整数构造
4  c = BigInteger.ONE; //或者BigInteger.ZERO
5  c = a.add(b); //加法
6  c = a.subtract(b); //减法
7  c = a.multiply(b); //乘法
8  c = a.divide(b); //除法
9  c = a.gcd(b); //gcd(a,b)
10 c = a.abs();
11 c = a.mod(b); //取模，a，b必须都是非负数
12 c = a.modPow(b,p); //a^b % p
13 //c = -1: a < b;
14 //c = 0: a = b;
15 //c = 1: a > b;
16 int c = a.compareTo(b);
17 String c = a.toString(); //可以用于打印结果
18
19 //BigDecimal
20 //add(), subtract(), multiply(), abs(), compareTo()及构造函数和BigInteger类似
21 public BigDecimal divide(BigDecimal divisor, int scale, int roundingMode);
22 //scale表示计算规模（精度？长度？取大点就好了，比如30）
23 //roundingMode表示取整规则，有以下几种常用规则：
24 //ROUND_UP, ROUND_DOWN, ROUND_CEILING, ROUND_FLOOR, ROUND_HALF_UP（四舍五入），
ROUND_HALF_DOWN
```

- BigDecimal使用例子

```
1  //hdu 4624
2  import java.io.*;
3  import java.util.*;
4  import java.math.*;
5
6  public class Main {
```

```
 7     static final int maxn = 50;
 8     public static void main(String[] args) {
 9         long [][][]f = new long[maxn+5][2][maxn*maxn+5];
10         BigDecimal []E = new BigDecimal[maxn+5];
11         f[0][0][0] = 1;
12         for(int i=0; i<=maxn; ++i)
13             for(int k=0; k<=(i+1)*i/2; ++k)
14                 for(int t=i+1; t<=maxn; ++t) {
15                     f[t][0][k+(t-i)*(t-i-1)/2] += f[i][1][k];
16                     f[t][1][k+(t-i)*(t-i-1)/2] += f[i][0][k];
17                 }
18         for(int i=1; i<=maxn; ++i) {
19             E[i] = new BigDecimal("0");
20             for(int k=0; k<=i; ++k) {
21                 for(int j=0; j<=(i+1)*i/2; ++j) {
22                     if( j+(i-k+1)*(i-k)/2 == (i+1)*i/2) continue;
23                     BigDecimal p = new BigDecimal("1");
24                     BigDecimal a = new BigDecimal(j + (i-k+1)*(i-k)/2);
25                     BigDecimal b = new BigDecimal((i+1)*i/2);
26                     p = p.subtract( a.divide(b, 30, BigDecimal.ROUND_HALF_UP) );
27                     BigDecimal cont = new BigDecimal(f[k][1][j] - f[k][0][j]);
28                     E[i] = E[i].add( cont.divide(p, 30,
BigDecimal.ROUND_HALF_UP) );
29                 }
30             }
31         }
32         Scanner in = new Scanner(System.in);
33         int T = in.nextInt(), x;
34         for(int t=1; t<=T; ++t) {
35             x = in.nextInt();
36             System.out.printf("%.15f\r\n", E[x]);
37         }
38     }
39 }
```