

SICNU ACM Template

renli

2018 年 7 月 23 日

目录

1 头文件

```

1 #include <bits/stdc++.h>
2 using namespace std;
3 #define LL long long
4 #define INF 0x3f3f3f3f
5 #define clr(a, x) memset(a, x, sizeof(a))
6 const double eps = 1e-6;
7 const LL MOD = 1e9+7;
8 const int MAXN = 1e5 + 5;
9
10 int main()
11 {
12 #ifndef ONLINE_JUDGE
13     freopen("in.txt", "r", stdin);
14 #endif
15
16     return 0;
17 }
18

```

2 数学

2.1 素数

2.1.1 素数筛

```

1 /*
2  * 素数筛选, 判断小于MAXN的数是不是素数。
3  * O(nloglogn)
4  * notprime为false表示是素数, true表示不是素数
5  */
6 bool notprime[MAXN]; // 值为false表示素数, 值为true表示非
   素数
7 void init()
8 {
9     memset(notprime, false, sizeof(notprime));
10     notprime[0] = notprime[1] = true;
11     for(int i = 2; i < MAXN; i++)
12         if(!notprime[i])
13         {
14             if(i > MAXN / i) continue; // 防止后面i*i溢出(
               或者i,j用long long)
15             // 直接从i*i开始就可以, 小于i倍的已经筛选过了, 注意j+=i
16             for(int j = i * i; j < MAXN; j += i)
17                 notprime[j] = true;
18         }
19 }

```

2.1.2 区间筛

```

1 /*
2  * 对区间[a,b)内的整数执行筛法。
3  * 函数返回区间内素数个数
4  * isPrime[i-a]=true表示i是素数
5  * $a < b <= 10^12, b - a < 10^6
6  */
7 bool isPrime_small[MAXN], isPrime[MAXN];
8 int prime[MAXN];
9 int segment_sieve(LL a, LL b)
10 {

```

```

11     int tot = 0;
12     for (LL i = 0; i * i < b; ++i)
13         isPrime_small[i] = true;
14     for (LL i = 0; i < b - a; ++i)
15         isPrime[i] = true;
16     for (LL i = 2; i * i < b; ++i)
17         if (isPrime_small[i])
18         {
19             for (LL j = 2 * i; j * j < b; j += i)
20                 isPrime_small[j] = false;
21             for (LL j = max(2LL, (a + i - 1) / i) * i;
22                 j < b; j += i)
23                 isPrime[j - a] = false;
24         }
25     for (LL i = 0; i < b - a; ++i)
26         if (isPrime[i]) prime[tot++] = i + a;
27     return tot;
28 }

```

2.1.3 Miller Rabin 素数判断

```

1 /*
2  * O(slogn) 内判定 2^63 内的数是不是素数, s 为测定次数
3  */
4 LL qmul(LL x, LL y, LL mod) // 乘法防止溢出, 如果p *
   p不爆LL的话可以直接乘; O(1)乘法或者转化成二进制加法
5 {
6     return (x * y - (long long)(x / (long double)mod *
7         y + 1e-3) * mod + mod) % mod;
8 }
9 LL qpow(LL a, LL b, LL n) // 快速幂取模 a^b%n
10 {
11     LL ans = 1;
12     while (b)
13     {
14         if (b & 1)
15             ans = qmul(ans, a, n);
16         a = qmul(a, a, n);
17         b >>= 1;
18     }
19     return ans;
20 }
21 bool Miller_Rabin(LL n, int s)
22 {
23     if (n == 2)
24         return 1;
25     if (n < 2 || !(n & 1))
26         return 0;
27     int t = 0;
28     LL x, y, u = n - 1;
29     while ((u & 1) == 0)
30         t++, u >>= 1;
31     for (int i = 0; i < s; i++)
32     {
33         LL a = rand() % (n - 1) + 1;
34         LL x = qpow(a, u, n);
35         for (int j = 0; j < t; j++)
36         {
37             LL y = qmul(x, x, n);
38             if (y == 1 && x != 1 && x != n - 1)
39                 return
40
41                     x = y;

```

```

42     }
43     if (x != 1)
44         return 0;
45     }
46     return 1;
47 }

```

2.2 高斯消元

$$A^{-1} * |A| = (A^*)$$

2.2.1 求逆矩阵

```

1  int a[MAXN][MAXN];
2  int ni[MAXN][MAXN]; // 逆矩阵
3
4  void solve(int n)
5  {
6      for(int i = 1; i <= n; i++)
7          for(int j = 1; j <= n; j++)
8              ni[i][j] = (i == j); // 初始化
9      int det = 1; // |A|
10     for(int i = 1; i <= n; i++)
11     {
12         int t = i;
13         for(int k = i; k <= n; k++)
14             if(a[k][i]) t = k;
15         if(t != i) det *= -1;
16         for(int j = 1; j <= n; j++)
17         {
18             swap(a[i][j], a[t][j]);
19             swap(ni[i][j], ni[t][j]);
20         }
21         det = 1ll * a[i][i] * det % MOD;
22         int inv = qpow(a[i][i], MOD - 2);
23         for(int j = 1; j <= n; j++)
24         {
25             a[i][j] = 1ll * inv * a[i][j] % MOD;
26             ni[i][j] = 1ll * inv * ni[i][j] % MOD;
27         }
28         for(int k = 1; k <= n; k++)
29         {
30             if(k == i) continue;
31             int tmp = a[k][i];
32             for(int j = 1; j <= n; j++)
33             {
34                 a[k][j] = (a[k][j] - 1ll * a[i][j] * tmp
35                     % MOD + MOD) % MOD;
36                 ni[k][j] = (ni[k][j] - 1ll * ni[i][j] *
37                     tmp % MOD + MOD) % MOD;
38             }
39         }
40         det = (det + MOD) % MOD; // |A|
41         for(int j = 1; j <= n; j++)
42             ni[i][j] = 1ll * det * ni[i][j] % MOD; //
43     }
44 }

```

2.2.2 解 01 方程组

```

1  //有equ个方程, var个变元。增广矩阵行数为equ, 列数为var+1,
   分别为0到var
2  int a[MAXN][MAXN];
3  int b[MAXN][MAXN];
4  //返回值为-1表示无解, 为0是唯一解, 否则返回自由变元个数
5  int Gauss(int equ, int var)
6  {
7      int max_r, col, k;
8      for(k = 0, col = 0; k < equ && col < var; k++,
9          col++)
10     {
11         max_r = k;
12         for(int i = k + 1; i < equ; i++)
13         {
14             if(abs(a[i][col]) > abs(a[max_r][col]))
15                 max_r = i;
16         }
17         if(a[max_r][col] == 0)
18         {
19             k--;
20             continue;
21         }
22         if(max_r != k)
23         {
24             for(int j = col; j < var + 1; j++)
25                 swap(a[k][j], a[max_r][j]);
26         }
27         for(int i = k + 1; i < equ; i++)
28         {
29             if(a[i][col] != 0)
30             {
31                 for(int j = col; j < var + 1; j++)
32                     a[i][j] ^= a[k][j];
33             }
34         }
35         return k;
36     }
37 }

```

2.3 矩阵快速幂

$$F(i) = \begin{cases} F(i-1) + F(i-2) + i^3 + i^2 + i + 1 & i > 1 \\ 0 & i = 0 \\ 1 & i = 1 \end{cases} \text{ 求 } F(i)$$

$$\begin{pmatrix} F(i) \\ F(i-1) \\ i^3 \\ i^2 \\ i \\ 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 1 & 1 & 4 & 6 & 4 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 3 & 3 & 1 \\ 0 & 0 & 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} F(i-1) \\ F(i-2) \\ (i-1)^3 \\ (i-1)^2 \\ (i-1) \\ 1 \end{pmatrix}$$

```

1  LL aaa[MAXN] =
2  {
3      1, 1, 1, 4, 6, 4,
4      1, 0, 0, 0, 0, 0,
5      0, 0, 1, 3, 3, 1,
6      0, 0, 0, 1, 2, 1,
7      0, 0, 0, 0, 1, 1,
8      0, 0, 0, 0, 0, 1
9  };
10 struct matrix
11 {
12     LL a[6][6];
13 };
14

```

```

15 matrix mat_mul(matrix x, matrix y)
16 {
17     matrix res;
18     memset(res.a, 0, sizeof(res.a));
19     for(int i = 0; i < 6; i++)
20         for(int j = 0; j < 6; j++)
21             for(int k = 0; k < 6; k++)
22                 res.a[i][j] = (res.a[i][j] + x.a[i][k] *
23                     y.a[k][j]) % MOD;
24     return res;
25 }
26 matrix mat_pow(LL n)
27 {
28     matrix c, res; // res = c ^ n
29     int ind = 0;
30     for(int i = 0; i < 6; i++)
31         for(int j = 0; j < 6; j++)
32             c.a[i][j] = aaa[ind++];
33     memset(res.a, 0, sizeof(res.a));
34     for(int i = 0; i < 6; i++)
35         res.a[i][i] = 1;
36     while(n)
37     {
38         if(n & 1)
39             res = mat_mul(res, c);
40         c = mat_mul(c, c);
41         n = n >> 1;
42     }
43     return res;
44 }

```

3 字符串

3.1 KMP

```

1  /*
2  * next[] 的含义: x[i-next[i]...i-1]=x[0...next[i]-1]
3  * next[i] 为满足 x[i-z...i-1]=x[0...z-1] 的最大 z 值 (就是 x
4  * 的自身匹配)
5  */
6  void kmp_pre(char x[], int m, int next[])
7  {
8      int i = 0, j = next[0] = -1;
9      while(i < m)
10     {
11         while(-1 != j && x[i] != x[j]) j = next[j];
12         next[++i] = ++j;
13     }
14 }
15 /*
16 * kmpNext[] 的意思: next'[i]=next[next[...[next[i]]]] (
17 * 直到 next'[i]<0 或者 x[next'[i]]!=x[i])
18 * 这样的预处理可以快一些
19 */
20 void preKMP(char x[], int m, int kmpNext[])
21 {
22     int i = 0, j = kmpNext[0] = -1;
23     while(i < m)
24     {
25         while(-1 != j && x[i] != x[j]) j = kmpNext[j];
26         if(x[++i] == x[++j]) kmpNext[i] = kmpNext[j];
27         else kmpNext[i] = j;
28     }
29 }

```

```

27 }
28 /*
29 * 返回 x 在 y 中出现的次数, 可以重叠
30 */
31 int next[10010];
32 int KMP_Count(char x[], int m, char y[], int n)
33 {
34     //x 是模式串, y 是主串
35     int i, j, ans = 0;
36     //preKMP(x, m, next);
37     kmp_pre(x, m, next);
38     i = j = 0;
39     while(i < n)
40     {
41         while(-1 != j && y[i] != x[j]) j = next[j];
42         i++, j++;
43         if(j >= m)
44         {
45             ans++;
46             j = next[j];
47         }
48     }
49     return ans;
50 }

```

3.2 扩展 KMP

```

1  //next[i]:x[i...m-1]与x[0...m-1]的最长公共前缀
2  //extend[i]:y[i...n-1]与x[0...m-1]的最长公共前缀
3  void pre_EKMP(char x[], int m, int next[])
4  {
5      next[0] = m;
6      int j = 0;
7      while(j + 1 < m && x[j] == x[j + 1]) j++;
8      next[1] = j;
9      int k = 1;
10     for(int i = 2; i < m; i++)
11     {
12         int p = next[k] + k - 1;
13         int L = next[i - k];
14         if(i + L < p + 1) next[i] = L;
15         else
16         {
17             j = max(0, p - i + 1);
18             while(i + j < m && x[i + j] == x[j]) j++;
19             next[i] = j;
20             k = i;
21         }
22     }
23 }
24 void EKMP(char x[], int m, char y[], int n, int next
25     [], int extend[])
26 {
27     pre_EKMP(x, m, next);
28     int j = 0;
29     while(j < n && j < m && x[j] == y[j]) j++;
30     extend[0] = j;
31     int k = 0;
32     for(int i = 1; i < n; i++)
33     {
34         int p = extend[k] + k - 1;
35         int L = next[i - k];
36         if(i + L < p + 1) extend[i] = L;
37         else

```

```

37     {
38         j = max(0, p - i + 1);
39         while(i + j < n && j < m && y[i + j] == x[j])j++;
40         extend[i] = j;
41         k = i;
42     }
43 }
44 }

```

3.3 Manacher 最长回文子串

```

1  /*
2  * O(n)求最长回文子串
3  */
4  char Ma[MAXN * 2];
5  int Mp[MAXN * 2];
6  void Manacher(char s[], int len)
7  {
8      int l = 0;
9      Ma[l++] = '$';
10     Ma[l++] = '#';
11     for(int i = 0; i < len; i++)
12     {
13         Ma[l++] = s[i];
14         Ma[l++] = '#';
15     }
16     Ma[l] = 0;
17     int mx = 0, id = 0;
18     for(int i = 0; i < l; i++)
19     {
20         Mp[i] = mx > i ? min(Mp[2 * id - i], mx - i) : 1;
21         while(Ma[i + Mp[i]] == Ma[i - Mp[i]])Mp[i]++;
22         if(i + Mp[i] > mx)
23         {
24             mx = i + Mp[i];
25             id = i;
26         }
27     }
28 }
29 /*
30 * abaaba
31 * i: 0 1 2 3 4 5 6 7 8 9 10 11 12 13
32 * Ma[i]: $ # a # b # a # a $ b # a #
33 * Mp[i]: 1 1 2 1 4 1 2 7 2 1 4 1 2 1
34 */
35 char s[MAXN];
36 int main()
37 {
38     while(scanf("%s", s) == 1)
39     {
40         int len = strlen(s);
41         Manacher(s, len);
42         int ans = 0;
43         for(int i = 0; i < 2 * len + 2; i++)
44             ans = max(ans, Mp[i] - 1);
45         printf("%d\n", ans);
46     }
47     return 0;
48 }

```

3.4 AC 自动机

```

1  /*
2  * 求目标串中出现了几个模式串
3  */
4  struct Trie
5  {
6      int next[500010][26], fail[500010], end[500010];
7      int root, L;
8      int newnode()
9      {
10         for(int i = 0; i < 26; i++)
11             next[L][i] = -1;
12         end[L++] = 0;
13         return L - 1;
14     }
15     void init()
16     {
17         L = 0;
18         root = newnode();
19     }
20     void insert(char buf[])
21     {
22         int len = strlen(buf);
23         int now = root;
24         for(int i = 0; i < len; i++)
25         {
26             if(next[now][buf[i] - 'a'] == -1)
27                 next[now][buf[i] - 'a'] = newnode();
28             now = next[now][buf[i] - 'a'];
29         }
30         end[now]++;
31     }
32     void build()
33     {
34         queue<int>Q;
35         fail[root] = root;
36         for(int i = 0; i < 26; i++)
37             if(next[root][i] == -1)
38                 next[root][i] = root;
39         else
40         {
41             fail[next[root][i]] = root;
42             Q.push(next[root][i]);
43         }
44         while( !Q.empty() )
45         {
46             int now = Q.front();
47             Q.pop();
48             for(int i = 0; i < 26; i++)
49                 if(next[now][i] == -1)
50                     next[now][i] = next[fail[now]][i];
51             else
52             {
53                 fail[next[now][i]] = next[fail[now]][i];
54                 Q.push(next[now][i]);
55             }
56         }
57     }
58     int query(char buf[])
59     {
60         int len = strlen(buf);
61         int now = root;
62         int res = 0;

```

```

63     for(int i = 0; i < len; i++)
64     {
65         now = next[now][buf[i] - 'a'];
66         int temp = now;
67         while( temp != root )
68         {
69             res += end[temp];
70             end[temp] = 0;
71             temp = fail[temp];
72         }
73     }
74     return res;
75 }
76 };
77 char buf[1000010];
78 Trie ac;
79
80 int solve(int n)
81 {
82     scanf("%d", &n);
83     ac.init();
84     for(int i = 0; i < n; i++)
85     {
86         scanf("%s", buf);
87         ac.insert(buf);
88     }
89     ac.build();
90     scanf("%s", buf);
91     printf("%d\n", ac.query(buf));
92 }

```

3.5 后缀数组

```

1 //倍增算法构造后缀数组,复杂度O(nlogn)
2 char s[MAXN];
3 int sa[MAXN], t[MAXN], t2[MAXN], c[MAXN], rank[MAXN],
   height[MAXN];
4 //n为字符串的长度,字符集的值为0~m-1
5 void build_sa(int m, int n)
6 {
7     n++;
8     int *x = t, *y = t2;
9     //基数排序
10    for (int i = 0; i < m; i++) c[i] = 0;
11    for (int i = 0; i < n; i++) c[x[i] = s[i]]++;
12    for (int i = 1; i < m; i++) c[i] += c[i - 1];
13    for (int i = n - 1; ~i; i--) sa[--c[x[i]]] = i;
14    for (int k = 1; k <= n; k <= 1)
15    {
16        //直接利用sa数组排序第二关键字
17        int p = 0;
18        for (int i = n - k; i < n; i++) y[p++] = i;
19        for (int i = 0; i < n; i++)
20            if (sa[i] >= k) y[p++] = sa[i] - k;
21        //基数排序第一关键字
22        for (int i = 0; i < m; i++) c[i] = 0;
23        for (int i = 0; i < n; i++) c[x[y[i]]]++;
24        for (int i = 1; i < m; i++) c[i] += c[i - 1];
25        for (int i = n - 1; ~i; i--) sa[--c[x[y[i]]]]
26            = y[i];
27        //根据sa和y数组计算新的x数组
28        swap(x, y);
29        p = 1;
30        x[sa[0]] = 0;

```

```

30     for (int i = 1; i < n; i++)
31         x[sa[i]] = y[sa[i - 1]] == y[sa[i]] && y[sa[
32             i - 1] + k] == y[sa[i] + k] ? p - 1 :
33             p++;
34         if (p >= n) break; //以后即使继续倍增,sa也不会改
35         变,推出
36         m = p; //下次基数排序的最大值
37     }
38     n--;
39     int k = 0;
40     for (int i = 0; i <= n; i++) rank[sa[i]] = i;
41     for (int i = 0; i < n; i++)
42     {
43         if (k) k--;
44         int j = sa[rank[i] - 1];
45         while (s[i + k] == s[j + k]) k++;
46         height[rank[i]] = k;
47     }
48 }
49 int dp[MAXN][30];
50 void initrmq(int n)
51 {
52     for (int i = 1; i <= n; i++)
53         dp[i][0] = height[i];
54     for (int j = 1; (1 << j) <= n; j++)
55         for (int i = 1; i + (1 << j) - 1 <= n; i++)
56             dp[i][j] = min(dp[i][j - 1], dp[i + (1 <<
57                 j - 1)][j - 1]);
58 }
59 int rmq(int l, int r)
60 {
61     int k = 31 - __builtin_clz(r - l + 1);
62     return min(dp[l][k], dp[r - (1 << k) + 1][k]);
63 }
64 int lcp(int a, int b)
65 {
66     // 求两个后缀的最长公共前缀
67     a = rank[a], b = rank[b];
68     if (a > b) swap(a, b);
69     return rmq(a + 1, b);
70 }

```

3.6 后缀自动机

```

1 struct SAM
2 {
3     int len[MAXN << 1], link[MAXN << 1], ch[MAXN <<
4         1][26];
5     int sz, rt, last;
6     int newnode(int x = 0)
7     {
8         len[sz] = x;
9         link[sz] = -1;
10        clr(ch[sz], -1);
11        return sz++;
12    }
13    void init() { sz = last = 0, rt = newnode(); }
14    void extend(int c)
15    {
16        int np = newnode(len[last] + 1);
17        int p;
18        for (p = last; ~p && ch[p][c] == -1; p = link[
19            p]) ch[p][c] = np;
20        if (p == -1)

```

```

19     link[np] = rt;
20     else
21     {
22         int q = ch[p][c];
23         if (len[p] + 1 == len[q])
24             link[np] = q;
25         else
26         {
27             int nq = newnode(len[p] + 1);
28             memcpy(ch[nq], ch[q], sizeof(ch[q]));
29             link[nq] = link[q], link[q] = link[np] =
30                 nq;
31             for (; ~p && ch[p][c] == q; p = link[p])
32                 ch[p][c] = nq;
33         }
34     }
35     last = np;
36 }
37 int topcnt[MAXN], topsam[MAXN << 1];
38 void sort()
39 { // 加入串后拓扑排序
40     clr(topcnt, 0);
41     for (int i = 0; i < sz; i++) topcnt[len[i]]++;
42     for (int i = 0; i < MAXN - 1; i++) topcnt[i +
43         1] += topcnt[i];
44     for (int i = 0; i < sz; i++) topsam[--topcnt[
45         len[i]]] = i;
46 }
47 };

```

4 数据结构

4.1 RMQ

4.1.1 一维 RMQ

```

1  /*
2  * 求区间最大最小值，数组下标从 1 开始。
3  * 预处理复杂度 O(nlogn)，查询O(1)
4  */
5  int mmax[MAXN][30], mmin[MAXN][30];
6  int a[MAXN], n;
7  void init()
8  {
9      for (int i = 1; i <= n; i++) mmax[i][0] = mmin[i]
10         [0] = a[i];
11      for (int j = 1; (1 << j) <= n; j++)
12          for (int i = 1; i + (1 << j) - 1 <= n; i++)
13              {
14                  mmax[i][j] = max(mmax[i][j - 1], mmax[i +
15                      (1 << (j - 1))][j - 1]);
16                  mmin[i][j] = min(mmin[i][j - 1], mmin[i +
17                      (1 << (j - 1))][j - 1]);
18              }
19      }
20      int rmqMax(int l, int r)
21      {
22          int k = 31 - __builtin_clz(r - l + 1);
23          return max(mmax[l][k], mmax[r - (1 << k) + 1][k]);
24      }
25      int rmqMin(int l, int r)
26      {

```

```

26         int k = 31 - __builtin_clz(r - l + 1);
27         return min(mmin[l][k], mmin[r - (1 << k) + 1][k]);
28     }

```

4.1.2 二维 RMQ

```

1  int dp[310][310][9][9]; // 最大值
2  int n, m;
3  void init()
4  {
5      for (int i = 0; (1 << i) <= n; i++)
6          for (int j = 0; (1 << j) <= m; j++)
7              {
8                  if (i == 0 && j == 0) continue;
9                  for (int row = 1; row + (1 << i) - 1 <= n;
10                     row++)
11                      for (int col = 1; col + (1 << j) - 1 <=
12                         m; col++)
13                          if (i)
14                              dp[row][col][i][j] =
15                                  max(dp[row][col][i - 1][j], dp[row + (1 << (i - 1))][
16                                      col][i - 1][j]);
17                          else
18                              dp[row][col][i][j] =
19                                  max(dp[row][col][i][j - 1], dp[row][col + (1 << (j -
20                                      1))][i][j - 1]);
21                      }
22      }
23      int rmq(int x1, int y1, int x2, int y2)
24      {
25          int kx = 31 - __builtin_clz(x2 - x1 + 1);
26          int ky = 31 - __builtin_clz(y2 - y1 + 1);
27          int m1 = dp[x1][y1][kx][ky];
28          int m2 = dp[x2 - (1 << kx) + 1][y1][kx][ky];
29          int m3 = dp[x1][y2 - (1 << ky) + 1][kx][ky];
30          int m4 = dp[x2 - (1 << kx) + 1][y2 - (1 << ky) +
31              1][kx][ky];
32          return max(max(m1, m2), max(m3, m4));
33      }

```

4.2 线段树

4.2.1 宏定义

```

1  #define lson rt << 1 // 左儿子
2  #define rson rt << 1 | 1 // 右儿子
3  #define Lson l, m, lson // 左子树
4  #define Rson m + 1, r, rson // 右子树

```

4.2.2 单点修改

```

1  int sum[MAXN << 2]; // sum[rt]用于维护区间和
2  void PushUp(int rt)
3  {
4      sum[rt] = sum[lson] + sum[rson];
5  }
6  void build(int l, int r, int rt)
7  {
8      if (l == r)
9      {
10         scanf("%d", &sum[rt]); // 建立的时候直接输入叶节点

```



```

11     return;
12 }
13 int m = (l + r) >> 1;
14 build(Lson);
15 build(Rson);
16 PushUp(rt);
17 }
18 void update(int p, int add, int l, int r, int rt)
19 {
20     if (l == r)
21     {
22         sum[rt] += add;
23         return;
24     }
25     int m = (l + r) >> 1;
26     if (p <= m) update(p, add, Lson);
27     else update(p, add, Rson);
28     PushUp(rt);
29 }
30 int query(int L, int R, int l, int r, int rt)
31 {
32     if (L <= l && r <= R)
33         return sum[rt];
34     int m = (l + r) >> 1;
35     int s = 0;
36     if (L <= m) s += query(L, R, Lson);
37     if (m < R) s += query(L, R, Rson);
38     return s;
39 }

```

4.2.3 区间修改

```

1 int lazy[MAXN << 2], sum[MAXN << 2];
2 void PushUp(int rt)
3 {
4     sum[rt] = sum[lson] + sum[rson];
5 }
6 void PushDown(int rt, int m)
7 {
8     if (lazy[rt] == 0)
9         return;
10    lazy[lson] += lazy[rt];
11    lazy[rson] += lazy[rt];
12    sum[lson] += lazy[rt] * (m - (m >> 1));
13    sum[rson] += lazy[rt] * (m >> 1);
14    lazy[rt] = 0;
15 }
16 void build(int l, int r, int rt)
17 {
18     lazy[rt] = 0;
19     if (l == r)
20     {
21         scanf("%lld", &sum[rt]);
22         return;
23     }
24     int m = (l + r) >> 1;
25     build(Lson);
26     build(Rson);
27     PushUp(rt);
28 }
29 void update(int L, int R, int add, int l, int r, int
    rt)
30 {
31     if (L <= l && r <= R)

```

```

32     {
33         lazy[rt] += add;
34         sum[rt] += add * (r - l + 1);
35         return;
36     }
37     PushDown(rt, r - l + 1);
38     int m = (l + r) >> 1;
39     if (L <= m) update(L, R, add, Lson);
40     if (m < R) update(L, R, add, Rson);
41     PushUp(rt);
42 }
43 int query(int L, int R, int l, int r, int rt)
44 {
45     if (L <= l && r <= R)
46         return sum[rt];
47     PushDown(rt, r - l + 1);
48     int m = (l + r) >> 1, ret = 0;
49     if (L <= m) ret += query(L, R, Lson);
50     if (m < R) ret += query(L, R, Rson);
51     return ret;
52 }

```

4.3 分块

```

1 int n, block, num, l[MAXN], r[MAXN], belong[MAXN];
2
3 void build()
4 {
5     block = sqrt(n);
6     num = n / block;
7     if (n % block) //除不尽, 多出一块
8         num++;
9     for (int i = 1; i <= num; i++)
10    {
11        l[i] = (i - 1) * block + 1;
12        r[i] = i * block;
13    }
14    r[num] = n; //制定最后一块的右端点为n
15    for (int i; i <= n; i++)
16        belong[i] = (i - 1) / block + 1;
17 }

```

5 图论

5.1 最小生成树

5.1.1 并查集

```

1 int fa[MAXN], ra[MAXN];
2 void init(int n)
3 {
4     clr(ra, 0);
5     for (int i = 1; i <= n; i++)
6         fa[i] = i;
7 }
8 int find(int x)
9 {
10    return fa[x] != x ? fa[x] = find(fa[x]) : x;
11 }
12 void Union(int x, int y)
13 {
14     x = find(x), y = find(y);

```

```

15     if (x == y) return;
16     if (ra[x] < ra[y])
17         fa[x] = y;
18     else
19     {
20         fa[y] = x;
21         if (ra[x] == ra[y]) ra[x]++;
22     }
23 }

```

5.1.2 Kruskal

```

1  const int MAXM = 10000; //最大边数
2
3  struct Edge
4  {
5      int u, v, w;
6  } edge[MAXM]; //存储边的信息, 包括起点/终点/权值
7  int tol; //边数, 加边前赋值为0
8  void addedge(int u, int v, int w)
9  {
10     edge[tol].u = u;
11     edge[tol].v = v;
12     edge[tol++].w = w;
13 }
14 bool cmp(Edge a, Edge b)
15 {
16     //排序函数, 讲边按照权值从小到大排序
17     return a.w < b.w;
18 }
19
20 int Kruskal(int n) //传入点数, 返回最小生成树的权值, 如果
    不连通返回-1
21 {
22     init(n);
23     sort(edge, edge + tol, cmp);
24     int cnt = 0; //计算加入的边数
25     int ans = 0;
26     for(int i = 0; i < tol; i++)
27     {
28         int u = edge[i].u;
29         int v = edge[i].v;
30         int w = edge[i].w;
31         if(find(u) != find(v))
32         {
33             Union(u, v);
34             ans += w;
35             cnt++;
36         }
37         if(cnt == n - 1) break;
38     }
39     if(cnt < n - 1) return -1; //不连通
40     else return ans;
41 }

```

5.1.3 Prim

```

1  /*
2  * Prim求MST
3  * 耗费矩阵cost[], 标号从0开始, 0~n-1
4  * 返回最小生成树的权值, 返回-1表示原图不连通
5  */
6  bool vis[MAXN];

```

```

7  int lowc[MAXN];
8  int Prim(int cost[][MAXN], int n) //点是0~n-1
9  {
10     int ans = 0;
11     memset(vis, false, sizeof(vis));
12     vis[0] = true;
13     for(int i = 1; i < n; i++)
14         lowc[i] = cost[0][i];
15     for(int i = 1; i < n; i++)
16     {
17         int minc = INF;
18         int p = -1;
19         for(int j = 0; j < n; j++)
20             if(!vis[j] && minc > lowc[j])
21             {
22                 minc = lowc[j];
23                 p = j;
24             }
25         if(minc == INF) return -1; //原图不连通
26         ans += minc;
27         vis[p] = true;
28         for(int j = 0; j < n; j++)
29             if(!vis[j] && lowc[j] > cost[p][j])
30                 lowc[j] = cost[p][j];
31     }
32     return ans;
33 }

```

5.2 最短路

5.2.1 Dijkstra

```

1  /*
2  * 复杂度O(ElogE)
3  * 注意对vector<Edge>E[MAXN]进行初始化后加边
4  */
5  struct qnode
6  {
7      int v, c;
8      qnode(int _v = 0, int _c = 0): v(_v), c(_c) {}
9      bool operator <(const qnode &r) const
10     {
11         return c > r.c;
12     }
13 };
14 struct Edge
15 {
16     int v, cost;
17     Edge(int _v = 0, int _cost = 0): v(_v), cost(_cost) {}
18 };
19 vector<Edge>E[MAXN];
20 bool vis[MAXN];
21 int dist[MAXN];
22 void Dijkstra(int n, int start) //点的编号从1开始
23 {
24     memset(vis, false, sizeof(vis));
25     for(int i = 1; i <= n; i++) dist[i] = INF;
26     priority_queue<qnode>que;
27     while(!que.empty()) que.pop();
28     dist[start] = 0;
29     que.push(qnode(start, 0));
30     qnode tmp;
31     while(!que.empty())

```

```

32 {
33     tmp = que.top();
34     que.pop();
35     int u = tmp.v;
36     if(vis[u])continue;
37     vis[u] = true;
38     for(int i = 0; i < E[u].size(); i++)
39     {
40         int v = E[tmp.v][i].v;
41         int cost = E[u][i].cost;
42         if(!vis[v] && dist[v] > dist[u] + cost)
43         {
44             dist[v] = dist[u] + cost;
45             que.push(qnode(v, dist[v]));
46         }
47     }
48 }
49 }
50 void addedge(int u, int v, int w)
51 {
52     E[u].push_back(Edge(v, w));
53 }

```

5.2.2 SPFA

```

1  /*
2  *单源最短路SPFA
3  *时间复杂度 O(kE)
4  *这个是队列实现，有时候改成栈实现会更加快，很容易修改
5  *这个复杂度是不定的
6  */
7  const int MAXN = 1010;
8  const int INF = 0x3f3f3f3f;
9  struct Edge
10 {
11     int v;
12     int cost;
13     Edge(int _v = 0, int _cost = 0): v(_v), cost(_cost) {}
14 };
15 vector<Edge>E[MAXN];
16 void addedge(int u, int v, int w)
17 {
18     E[u].push_back(Edge(v, w));
19 }
20 bool vis[MAXN];//在队列标志
21 int cnt[MAXN];//每个点的入队列次数
22 int dist[MAXN];
23 bool SPFA(int start, int n)
24 {
25     memset(vis, false, sizeof(vis));
26     for(int i = 1; i <= n; i++)dist[i] = INF;
27     vis[start] = true;
28     dist[start] = 0;
29     queue<int>que;
30     que.push(start);
31     memset(cnt, 0, sizeof(cnt));
32     cnt[start] = 1;
33     while(!que.empty())
34     {
35         int u = que.front();
36         que.pop();
37         vis[u] = false;
38         for(int i = 0; i < E[u].size(); i++)

```

```

39     {
40         int v = E[u][i].v;
41         if(dist[v] > dist[u] + E[u][i].cost)
42         {
43             dist[v] = dist[u] + E[u][i].cost;
44             if(!vis[v])
45             {
46                 vis[v] = true;
47                 que.push(v);
48                 if(++cnt[v] > n)return false;
49                 //cnt[i]为入队列次数，用来判定是否存在
                    负环回路
50             }
51         }
52     }
53 }
54 return true;
55 }

```

5.2.3 Floyd

```

1  // ---
2  // O(n^3)求出任意两点间最短路
3  // 邻接矩阵存图需注意判断重边
4  // ---
5  int G[MAXN][MAXN];
6  void init(int n)
7  {
8      clr(G, 0x3f);
9      for (int i = 0; i < n; i++) G[i][i] = 0;
10 }
11 void add_edge(int u, int v, int w) { G[u][v] = min(G[u][v], w); }
12 void Floyd(int n)
13 {
14     for (int k = 0; k < n; k++)
15         for (int i = 0; i < n; i++)
16             for (int j = 0; j < n; j++)
17                 G[i][j] = min(G[i][j], G[i][k] + G[k][j]);
18 }

```

5.3 LCA

5.3.1 离线 Tarjan

```

1  /**
2  * Tarjan离线算法
3  * 时间复杂度O(n+q)
4  */
5  int par[MAXN]; //并查集
6  int ans[MAXN]; //存储答案
7  vector<int> G[MAXN]; //邻接表
8  vector<int> query[MAXN], num[MAXN]; //存储查询信息
9  bool vis[MAXN]; //是否被遍历
10 inline void init(int n)
11 {
12     for (int i = 1; i <= n; i++)
13     {
14         G[i].clear();
15         query[i].clear();
16         num[i].clear();
17         par[i] = i;

```

```

18     vis[i] = 0;
19 }
20 }
21 int find(int x)
22 {
23     return par[x] != x ? par[x] = find(par[x]) : x;
24 }
25 void Union(int x, int y)
26 {
27     x = find(x), y = find(y);
28     if (x == y) return;
29     par[y] = x;
30 }
31
32 inline void addEdge(int u, int v)
33 {
34     G[u].push_back(v);
35 }
36 inline void addQuery(int id, int u, int v)
37 {
38     query[u].push_back(v), query[v].push_back(u);
39     num[u].push_back(id), num[v].push_back(id);
40 }
41 void tarjan(int u)
42 {
43     vis[u] = 1;
44     for(int i = 0; i < G[u].size(); i++)
45     {
46         int v = G[u][i];
47         if (vis[v]) continue;
48         tarjan(v);
49         Union(u, v);
50     }
51     for(int i = 0; i < query[u].size(); i++)
52     {
53         int v = query[u][i];
54         if (!vis[v]) continue;
55         ans[num[u][i]] = find(v);
56     }
57 }

```

5.3.2 LCA 倍增法

```

1  /*
2  * LCA 在线算法
3  */
4  const int DEG = 20;
5  struct Edge
6  {
7      int to, next;
8      int val;
9  } edge[MAXN * 2];
10 int head[MAXN], tot;
11 void addedge(int u, int v, int k)
12 {
13     edge[tot].to = v;
14     edge[tot].next = head[u];
15     edge[tot].val = k;
16     head[u] = tot++;
17 }
18 void init()
19 {
20     tot = 0;
21     memset(head, -1, sizeof(head));

```

```

22 }
23 int fa[MAXN][DEG]; // fa[i][j] 表示结点 i 的第 2^j 个祖先
24 int deg[MAXN]; // 深度数组
25 int dis[MAXN]; // 距离数组
26 void BFS(int root)
27 {
28     queue<int> que;
29     deg[root] = 0;
30     fa[root][0] = root;
31     que.push(root);
32     while(!que.empty())
33     {
34         int tmp = que.front();
35         que.pop();
36         for(int i = 1; i < DEG; i++)
37             fa[tmp][i] = fa[fa[tmp][i - 1]][i - 1];
38         for(int i = head[tmp]; i != -1; i = edge[i].next)
39         {
40             int v = edge[i].to;
41             if(v == fa[tmp][0]) continue;
42             deg[v] = deg[tmp] + 1;
43             dis[v] = dis[tmp] + edge[i].val;
44             fa[v][0] = tmp;
45             que.push(v);
46         }
47     }
48 }
49 int LCA(int u, int v)
50 {
51     if(deg[u] > deg[v]) swap(u, v);
52     int hu = deg[u], hv = deg[v];
53     int tu = u, tv = v;
54     for(int det = hv - hu, i = 0; det; det >>= 1, i++)
55         if(det & 1)
56             tv = fa[tv][i];
57     if(tu == tv) return tu;
58     for(int i = DEG - 1; i >= 0; i--)
59     {
60         if(fa[tu][i] == fa[tv][i])
61             continue;
62         tu = fa[tu][i];
63         tv = fa[tv][i];
64     }
65     return fa[tu][0];
66 }
67 bool flag[MAXN];
68
69 void solve()
70 {
71     int n, q;
72     scanf("%d%d", &n, &q);
73     init();
74     clr(flag, 0);
75     for(int i = 1; i < n; i++)
76     {
77         int a, b, k;
78         scanf("%d%d%d", &a, &b, &k);
79         addedge(a, b, k);
80         addedge(b, a, k);
81     }
82     BFS(1);
83     for(int i = 1; i <= q; i++)
84     {

```

```

85     int u, v;
86     scanf("%d%d", &u, &v);
87     printf("%d\n", dis[u]+dis[v]-2*dis[LCA(u, v)])
88     ;
89 }

```

5.4 拓扑排序

```

1  // ---
2  // 存图前记得初始化
3  // Ans排序结果, G邻接表, deg入度, map用于判断重边
4  // 排序成功返回1, 存在环返回0
5  // ---
6  int Ans[MAXN];
7  vector<int> G[MAXN];
8  int deg[MAXN];
9  map<pair<int, int>, bool> S;
10 void init(int n)
11 {
12     S.clear();
13     for (int i = 0; i < n; i++) G[i].clear();
14     clr(deg, 0), clr(Ans, 0);
15 }
16 void add_edge(int u, int v)
17 {
18     if (S[make_pair(u, v)]) return;
19     G[u].push_back(v), S[make_pair(u, v)] = 1, deg[v]++;
20 }
21 bool Toposort(int n)
22 {
23     int tot = 0;
24     queue<int> q;
25     for (int i = 0; i < n; ++i)
26         if (deg[i] == 0) q.push(i);
27     while (!q.empty())
28     {
29         int u = q.front();
30         q.pop();
31         Ans[tot++] = u;
32         for(int i = 0; i < G[u].size(); i++)
33         {
34             int v = G[u][i];
35             if(--deg[v] == 0)
36                 q.push(v);
37         }
38     }
39     if (tot < n - 1) return false;
40     return true;
41 }

```

5.5 网络流

5.5.1 建模技巧

建模技巧

二分图带权最大独立集。给出一个二分图，每个结点上有一个正权值。要求选出一些点，使得这些点之间没有边相连，且权值和最大。

解：在二分图的基础上添加源点 S 和汇点 T ，然后从 S 向所有 X 集合中的点连一条边，所有 Y 集合中的点向 T 连一条边，容量

均为该点的权值。 X 结点与 Y 结点之间的边的容量均为无穷大。这样，对于图中的任意一个割，将割中的边对应的结点删掉就是一个符合要求的解，权和为所有权减去割的容量。因此，只需要求出最小割，就能求出最大权和。

公平分配问题。把 m 个任务分配给 n 个处理器。其中每个任务有两个候选处理器，可以任选一个分配。要求所有处理器中，任务数最多的那个处理器所分配的任务数尽量少。不同任务的候选处理器集 $\{p_1, p_2\}$ 保证不同。

解：本题有一个比较明显的二分图模型，即 X 结点是任务， Y 结点是处理器。二分答案 x ，然后构图，首先从源点 S 出发向所有的任务结点引一条边，容量等于 1，然后从每个任务结点出发引两条边，分别到达它能分配到的两个处理器结点，容量为 1，最后从每个处理器结点出发引一条边到汇点 T ，容量为 x ，表示选择该处理器的任务不能超过 x 。这样网络中的每个单位流量都是从 S 流到一个任务结点，再到处理器结点，最后到汇点 T 。只有当网络中的总流量等于 m 时才意味着所有任务都选择了一个处理器。这样，我们通过 $O(\log m)$ 次最大流便算出了答案。

区间 k 覆盖问题。数轴上有一些带权值的左闭右开区间。选出权和尽量大的一些区间，使得任意一个数最多被 k 个区间覆盖。

解：本题可以用最小费用流解决，构图方法是把每个数作为一个结点，然后对于权值为 w 的区间 $[u, v)$ 加边 $u \rightarrow v$ ，容量为 1，费用为 $-w$ 。再对所有相邻的点加边 $i \rightarrow i+1$ ，容量为 k ，费用为 0。最后，求最左点到最右点的最小费用最大流即可，其中每个流量对应一组互不相交的区间。如果数值范围太大，可以先进行离散化。

最大闭合子图。给定带权图 G （权值可正可负），求一个权和最大的点集，使得起点在该点集中的任意弧，终点也在该点集中。

解：新增附加源 s 和附加汇 t ，从 s 向所有正权点引一条边，容量为权值；从所有负权点向汇点引一条边，容量为权值的相反数。求出最小割以后， $S - \{s\}$ 就是最大闭合子图。

5.5.2 Edge

```

1  // ---
2  // 最大流
3  // ---
4  struct Edge
5  {
6      int from, to, cap, flow;
7      Edge(int u, int v, int c, int f)
8          : from(u), to(v), cap(c), flow(f) {}
9  };
10 // ---
11 // 最小费用流
12 // ---
13 struct Edge
14 {
15     int from, to, cap, flow, cost;
16     Edge(int u, int v, int c, int f, int w)
17         : from(u), to(v), cap(c), flow(f), cost(w) {}
18 };

```

5.5.3 Dinic

```

1 struct Dinic
2 {
3     int n, m, s, t; //结点数, 边数 (包括反向弧), 源点编号和汇点编号

```

```

4  vector<Edge> edges; //边表。edge[e]和edge[e^1]互为
   反向弧
5  vector<int> G[MAXN]; //邻接表，G[i][j]表示节点i的第
   j条边在e数组中的序号
6  bool vis[MAXN]; //BFS使用
7  int d[MAXN]; //从起点到i的距离
8  int cur[MAXN]; //当前弧下标
9  void init(int n)
10 {
11     this->n = n;
12     for (int i = 0; i < n; i++) G[i].clear();
13     edges.clear();
14 }
15 void AddEdge(int from, int to, int cap)
16 {
17     edges.push_back(Edge(from, to, cap, 0));
18     edges.push_back(Edge(to, from, 0, 0));
19     m = edges.size();
20     G[from].push_back(m - 2);
21     G[to].push_back(m - 1);
22 }
23 bool BFS()
24 {
25     clr(vis, 0);
26     clr(d, 0);
27     queue<int> q;
28     q.push(s);
29     d[s] = 0;
30     vis[s] = 1;
31     while (!q.empty())
32     {
33         int x = q.front();
34         q.pop();
35         for (int i = 0; i < G[x].size(); i++)
36         {
37             Edge& e = edges[G[x][i]];
38             if (!vis[e.to] && e.cap > e.flow)
39             {
40                 vis[e.to] = 1;
41                 d[e.to] = d[x] + 1;
42                 q.push(e.to);
43             }
44         }
45     }
46     return vis[t];
47 }
48 int DFS(int x, int a)
49 {
50     if (x == t || a == 0) return a;
51     int flow = 0, f;
52     for (int& i = cur[x]; i < G[x].size(); i++)
53     {
54         //从上次考虑的弧
55         Edge& e = edges[G[x][i]];
56         if (d[x] + 1 == d[e.to] && (f = DFS(e.to,
57             min(a, e.cap - e.flow))) > 0)
58         {
59             e.flow += f;
60             edges[G[x][i] ^ 1].flow -= f;
61             flow += f;
62             a -= f;
63             if (a == 0) break;
64         }
65     }
66     return flow;

```

```

66     }
67     int Maxflow(int s, int t)
68     {
69         this->s = s;
70         this->t = t;
71         int flow = 0;
72         while (BFS())
73         {
74             clr(cur, 0);
75             flow += DFS(s, INF);
76         }
77         return flow;
78     }
79 };

```

5.5.4 ISAP

```

1  struct ISAP
2  {
3      int n, m, s, t; //结点数，边数（包括反向弧），源点编
   号和汇点编号
4      vector<Edge> edges; //边表。edges[e]和edges[e^1]互
   为反向弧
5      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
6      bool vis[MAXN]; //BFS使用
7      int d[MAXN]; //起点到i的距离
8      int cur[MAXN]; //当前弧下标
9      int p[MAXN]; //可增广路上的一条弧
10     int num[MAXN]; //距离标号计数
11     void init(int n)
12     {
13         this->n = n;
14         for (int i = 0; i < n; i++) G[i].clear();
15         edges.clear();
16     }
17     void AddEdge(int from, int to, int cap)
18     {
19         edges.push_back(Edge(from, to, cap, 0));
20         edges.push_back(Edge(to, from, 0, 0));
21         int m = edges.size();
22         G[from].push_back(m - 2);
23         G[to].push_back(m - 1);
24     }
25     int Augument()
26     {
27         int x = t, a = INF;
28         while (x != s)
29         {
30             Edge& e = edges[p[x]];
31             a = min(a, e.cap - e.flow);
32             x = edges[p[x]].from;
33         }
34         x = t;
35         while (x != s)
36         {
37             edges[p[x]].flow += a;
38             edges[p[x] ^ 1].flow -= a;
39             x = edges[p[x]].from;
40         }
41         return a;
42     }
43     void BFS()
44     {

```

```

45   clr(vis, 0);
46   clr(d, 0);
47   queue<int> q;
48   q.push(t);
49   d[t] = 0;
50   vis[t] = 1;
51   while (!q.empty())
52   {
53       int x = q.front();
54       q.pop();
55       int len = G[x].size();
56       for (int i = 0; i < len; i++)
57       {
58           Edge& e = edges[G[x][i]];
59           if (!vis[e.from] && e.cap > e.flow)
60           {
61               vis[e.from] = 1;
62               d[e.from] = d[x] + 1;
63               q.push(e.from);
64           }
65       }
66   }
67 }
68 int Maxflow(int s, int t)
69 {
70     this->s = s;
71     this->t = t;
72     int flow = 0;
73     BFS();
74     clr(num, 0);
75     for (int i = 0; i < n; i++)
76         if (d[i] < INF) num[d[i]]++;
77     int x = s;
78     clr(cur, 0);
79     while (d[s] < n)
80     {
81         if (x == t)
82         {
83             flow += Augument();
84             x = s;
85         }
86         int ok = 0;
87         for (int i = cur[x]; i < G[x].size(); i++)
88         {
89             Edge& e = edges[G[x][i]];
90             if (e.cap > e.flow && d[x] == d[e.to] + 1)
91             {
92                 ok = 1;
93                 p[e.to] = G[x][i];
94                 cur[x] = i;
95                 x = e.to;
96                 break;
97             }
98         }
99         if (!ok) //Retreat
100         {
101             int m = n - 1;
102             for (int i = 0; i < G[x].size(); i++)
103             {
104                 Edge& e = edges[G[x][i]];
105                 if (e.cap > e.flow) m = min(m, d[e.to]);
106             }
107             if (--num[d[x]] == 0) break; //gap优化

```

```

108         num[d[x] = m + 1]++;
109         cur[x] = 0;
110         if (x != s) x = edges[p[x]].from;
111     }
112 }
113 return flow;
114 }
115 };

```

5.5.5 MCMF

```

1 struct MCMF
2 {
3     int n, m;
4     vector<Edge> edges;
5     vector<int> G[MAXN];
6     int inq[MAXN]; //是否在队列中
7     int d[MAXN]; //bellmanford
8     int p[MAXN]; //上一条弧
9     int a[MAXN]; //可改进量
10    void init(int n)
11    {
12        this->n = n;
13        for (int i = 0; i < n; i++) G[i].clear();
14        edges.clear();
15    }
16    void AddEdge(int from, int to, int cap, int cost)
17    {
18        edges.push_back(Edge(from, to, cap, 0, cost));
19        edges.push_back(Edge(to, from, 0, 0, -cost));
20        m = edges.size();
21        G[from].push_back(m - 2);
22        G[to].push_back(m - 1);
23    }
24    bool BellmanFord(int s, int t, int& flow, LL& cost)
25    {
26        for (int i = 0; i < n; i++) d[i] = INF;
27        clr(inq, 0);
28        d[s] = 0;
29        inq[s] = 1;
30        p[s] = 0;
31        a[s] = INF;
32        queue<int> q;
33        q.push(s);
34        while (!q.empty())
35        {
36            int u = q.front();
37            q.pop();
38            inq[u] = 0;
39            for (int i = 0; i < G[u].size(); i++)
40            {
41                Edge& e = edges[G[u][i]];
42                if (e.cap > e.flow && d[e.to] > d[u] + e.cost)
43                {
44                    d[e.to] = d[u] + e.cost;
45                    p[e.to] = G[u][i];
46                    a[e.to] = min(a[u], e.cap - e.flow);
47                    if (!inq[e.to])
48                    {
49                        q.push(e.to);
50                        inq[e.to] = 1;
51                    }

```



```

52     }
53 }
54 }
55 if (d[t] == INF) return false; // 当没有可增广的
    路时退出
56 flow += a[t];
57 cost += (LL)d[t] * (LL)a[t];
58 for (int u = t; u != s; u = edges[p[u]].from)
59 {
60     edges[p[u]].flow += a[t];
61     edges[p[u] ^ 1].flow -= a[t];
62 }
63 return true;
64 }
65 int MincostMaxflow(int s, int t, LL& cost)
66 {
67     int flow = 0;
68     cost = 0;
69     while (BellmanFord(s, t, flow, cost));
70     return flow;
71 }
72 };

```

6 计算几何

6.1 基本函数

6.1.1 定义点和线

```

1  const double PI = acos(-1.0);
2  #define zero(x) ((fabs(x) < eps ? 1 : 0))
3  int sgn(double x)
4  {
5      if(fabs(x) < eps) return 0;
6      if(x < 0) return -1;
7      else return 1;
8  }
9
10 struct point
11 {
12     double x, y;
13     point(double a = 0, double b = 0)
14     {
15         x = a, y = b;
16     }
17     point operator-(const point &b) const
18     {
19         return point(x - b.x, y - b.y);
20     }
21     point operator+(const point &b) const
22     {
23         return point(x + b.x, y + b.y);
24     }
25     // 两点是否重合
26     bool operator==(point &b)
27     {
28         return zero(x - b.x) && zero(y - b.y);
29     }
30     // 点积(以原点为基准)
31     double operator*(const point &b) const
32     {
33         return x * b.x + y * b.y;
34     }
35     // 叉积(以原点为基准)

```

```

36 double operator^(const point &b) const
37 {
38     return x * b.y - y * b.x;
39 }
40 // 绕P点逆时针旋转a弧度后的点
41 point rotate(point b, double a)
42 {
43     double dx, dy;
44     (*this - b).split(dx, dy);
45     double tx = dx * cos(a) - dy * sin(a);
46     double ty = dx * sin(a) + dy * cos(a);
47     return point(tx, ty) + b;
48 }
49 // 点坐标分别赋值到a和b
50 void split(double &a, double &b)
51 {
52     a = x, b = y;
53 }
54 };
55 struct line
56 {
57     point s, e;
58     line() {}
59     line(point _s, point _e)
60     {
61         s = _s;
62         e = _e;
63     }
64     // 两直线相交求交点
65     // 第一个值为0表示直线重合, 为1表示平行, 为2表示相交, 为
        2是相交
66     // 只有第一个值为2时, 交点才有意义
67     pair<int, point> operator &(const line &b) const
68     {
69         point res = s;
70         if(sgn((s - e) ^ (b.s - b.e)) == 0)
71         {
72             if(sgn((s - b.e) ^ (b.s - b.e)) == 0)
73                 return make_pair(0, res); // 重合
74             else return make_pair(1, res); // 平行
75         }
76         double t = ((s - b.s) ^ (b.s - b.e)) / ((s - e)
77             ^ (b.s - b.e));
78         res.x += (e.x - s.x) * t;
79         res.y += (e.y - s.y) * t;
80         return make_pair(2, res);
81     }
82 };

```

6.1.2 两点间距离

```

1  // 两点间距离
2  double dist(point a, point b)
3  {
4      return sqrt((a - b) * (a - b));
5  }

```

6.1.3 线段相交

```

1  // 判断线段相交
2  bool inter(line l1, line l2)
3  {
4      return

```



```

5     max(l1.s.x, l1.e.x) >= min(l2.s.x, l2.e.x) &&
6     max(l2.s.x, l2.e.x) >= min(l1.s.x, l1.e.x) &&
7     max(l1.s.y, l1.e.y) >= min(l2.s.y, l2.e.y) &&
8     max(l2.s.y, l2.e.y) >= min(l1.s.y, l1.e.y) &&
9     sgn((l2.s - l1.e) ^ (l1.s - l1.e)) * sgn((l2.e
10    -l1.e) ^ (l1.s - l1.e)) <= 0 &&
11    sgn((l1.s - l2.e) ^ (l2.s - l2.e)) * sgn((l1.e
    -l2.e) ^ (l2.s - l2.e)) <= 0;
}

```

6.1.4 直线和线段相交

```

1 //判断直线和线段相交
2 bool Seg_inter_line(line l1, line l2) //判断直线l1和线
   段l2是否相交
3 {
4     return sgn((l2.s - l1.e) ^ (l1.s - l1.e)) * sgn((
       l2.e-l1.e) ^ (l1.s - l1.e)) <= 0;
5 }

```

6.1.5 点到直线距离

```

1 //点到直线距离
2 //返回为result,是点到直线最近的点
3 point PointToLine(point P, line L)
4 {
5     point result;
6     double t = ((P - L.s) * (L.e-L.s)) / ((L.e-L.s) *
       (L.e-L.s));
7     result.x = L.s.x + (L.e.x - L.s.x) * t;
8     result.y = L.s.y + (L.e.y - L.s.y) * t;
9     return result;
10 }

```

6.1.6 点到线段距离

```

1 //点到线段的距离
2 //返回点到线段最近的点
3 point NearestPointToLineSeg(point P, line L)
4 {
5     point result;
6     double t = ((P - L.s) * (L.e-L.s)) / ((L.e-L.s) *
       (L.e-L.s));
7     if(t >= 0 && t <= 1)
8     {
9         result.x = L.s.x + (L.e.x - L.s.x) * t;
10        result.y = L.s.y + (L.e.y - L.s.y) * t;
11    }
12    else
13    {
14        if(dist(P, L.s) < dist(P, L.e))
15            result = L.s;
16        else result = L.e;
17    }
18    return result;
19 }

```

6.1.7 判断点在线段上

```

1 /**判断点在线段上
2 bool OnSeg(point P, line L)
3 {
4     return
5         sgn((L.s - P) ^ (L.e-P)) == 0 &&
6         sgn((P.x - L.s.x) * (P.x - L.e.x)) <= 0 &&
7         sgn((P.y - L.s.y) * (P.y - L.e.y)) <= 0;
8 }

```

6.2 多边形

6.2.1 计算多边形面积

```

1 //计算多边形面积
2 //点的编号从0~n-1
3 double CalcArea(point p[], int n)
4 {
5     double res = 0;
6     for(int i = 0; i < n; i++)
7         res += (p[i] ^ p[(i + 1) % n]) / 2;
8     return fabs(res);
9 }

```

6.2.2 判断点在凸多边形内

```

1 /**判断点在凸多边形内
2 //点形成一个凸包, 而且按逆时针排序 (如果是顺时针把里面的<0
   改为>0)
3 //点的编号:0~n-1
4 //返回值:
5 //-1:点在凸多边形外
6 //0:点在凸多边形边界上
7 //1:点在凸多边形内
8 int inConvexPoly(point a, point p[], int n)
9 {
10    for(int i = 0; i < n; i++)
11    {
12        if(sgn((p[i] - a) ^ (p[(i + 1) % n] - a)) < 0)
13            return -1;
14        else if(OnSeg(a, line(p[i], p[(i + 1) % n])))
15            return 0;
16    }
17    return 1;
18 }

```

6.2.3 判断点在任意多边形内

```

1 /**判断点在任意多边形内
2 //射线法, poly[]的顶点数要大于等于3,点的编号0~n-1
3 //返回值
4 //-1:点在凸多边形外
5 //0:点在凸多边形边界上
6 //1:点在凸多边形内
7 int inPoly(point p, point poly[], int n)
8 {
9     int cnt;
10    line ray, side;
11    cnt = 0;
12    ray.s = p;

```

```

13 ray.e.y = p.y;
14 ray.e.x = -10000000000.0; //-INF, 注意取值防止越界
15 for(int i = 0; i < n; i++)
16 {
17     side.s = poly[i];
18     side.e = poly[(i + 1) % n];
19     if(OnSeg(p, side))return 0;
20     //如果平行轴则不考虑
21     if(sgn(side.s.y - side.e.y) == 0)
22         continue;
23     if(OnSeg(side.s, ray))
24     {
25         if(sgn(side.s.y - side.e.y) > 0)cnt++;
26     }
27     else if(OnSeg(side.e, ray))
28     {
29         if(sgn(side.e.y - side.s.y) > 0)cnt++;
30     }
31     else if(inter(ray, side))
32         cnt++;
33 }
34 if(cnt % 2 == 1)return 1;
35 else return -1;
36 }

```

6.2.4 判断凸多边形

```

1 //判断凸多边形
2 //允许共线边
3 //点可以是顺时针给出也可以是逆时针给出
4 //点的编号1~n-1
5 bool isconvex(point poly[], int n)
6 {
7     bool s[3];
8     memset(s, false, sizeof(s));
9     for(int i = 0; i < n; i++)
10     {
11         s[sgn( (poly[(i + 1) % n] - poly[i]) ^ (poly[(i + 2) % n] - poly[i]) ) + 1] = true;
12         if(s[0] && s[2])return false;
13     }
14     return true;
15 }

```

6.2.5 凸包

```

1 /*
2  * 求凸包, Graham算法
3  * 点的编号0~n-1
4  * 返回凸包结果Stack[0~top-1]为凸包的编号
5  */
6 point lst[MAXN];
7 int Stack[MAXN], top;
8 //相对于lst[0]的极角排序
9 bool _cmp(point p1, point p2)
10 {
11     double tmp = (p1 - lst[0]) ^ (p2 - lst[0]);
12     if(sgn(tmp) > 0)return true;
13     else if(sgn(tmp) == 0 && sgn(dist(p1, lst[0]) - dist(p2, lst[0])) <= 0)
14         return true;
15     else return false;
16 }

```

```

17 void Graham(int n)
18 {
19     point p0;
20     int k = 0;
21     p0 = lst[0];
22     //找最下边的一个点
23     for(int i = 1; i < n; i++)
24     {
25         if( (p0.y > lst[i].y) || (p0.y == lst[i].y && p0.x > lst[i].x) )
26         {
27             p0 = lst[i];
28             k = i;
29         }
30     }
31     swap(lst[k], lst[0]);
32     sort(lst + 1, lst + n, _cmp);
33     if(n == 1)
34     {
35         top = 1;
36         Stack[0] = 0;
37         return;
38     }
39     if(n == 2)
40     {
41         top = 2;
42         Stack[0] = 0;
43         Stack[1] = 1;
44         return;
45     }
46     Stack[0] = 0;
47     Stack[1] = 1;
48     top = 2;
49     for(int i = 2; i < n; i++)
50     {
51         while(top > 1 && sgn((lst[Stack[top - 1]] - lst[Stack[top - 2]]) ^ (lst[i] - lst[Stack[top - 2]])) <= 0)
52             top--;
53         Stack[top++] = i;
54     }
55 }

```

6.3 圆

6.3.1 外心

```

1 //过三点求圆心坐标
2 point waixin(point a, point b, point c)
3 {
4     double a1 = b.x - a.x, b1 = b.y - a.y, c1 = (a1 * a1 + b1 * b1) / 2;
5     double a2 = c.x - a.x, b2 = c.y - a.y, c2 = (a2 * a2 + b2 * b2) / 2;
6     double d = a1 * b2 - a2 * b1;
7     return point(a.x + (c1 * b2 - c2 * b1) / d, a.y + (a1 * c2 - a2 * c1) / d);
8 }

```

6.3.2 两圆相交的面积

```

1 //两个圆的公共部分面积

```

```

2 double Area_of_overlap(point c1, double r1, point c2,
3 double r2)
4 {
5     double d = dist(c1, c2);
6     if(r1 + r2 < d + eps) return 0;
7     if(d < fabs(r1 - r2) + eps)
8     {
9         double r = min(r1, r2);
10        return PI * r * r;
11    }
12    double x = (d * d + r1 * r1 - r2 * r2) / (2 * d);
13    double t1 = acos(x / r1);
14    double t2 = acos((d - x) / r2);
15    return r1 * r1 * t1 + r2 * r2 * t2 - d * r1 * sin(
        t1);
}

```

7 动态规划

7.1 最长上升子序列

```

1 // 序列下标从1开始, LIS()返回长度, 序列存在lis[]中
2 int len, a[MAXN], b[MAXN], f[MAXN];
3 int Find(int p, int l, int r)
4 {
5     while (l <= r)
6     {
7         int mid = (l + r) >> 1;
8         if (a[p] > b[mid])
9             l = mid + 1;
10        else
11            r = mid - 1;
12    }
13    return f[p] = l;
14 }
15 int LIS1(int lis[], int n)
16 {
17     int len = 1;
18     f[1] = 1, b[1] = a[1];
19     for (int i = 2; i <= n; i++)
20     {
21         if (a[i] > b[len])
22             b[++len] = a[i], f[i] = len;
23         else
24             b[Find(i, 1, len)] = a[i];
25     }
26     for (int i = n, t = len; i >= 1 && t >= 1; i--)
27         if (f[i] == t) lis[--t] = a[i];
28     return len;
29 }
30
31 // 简单写法(下标从0开始,只返回长度)
32 int dp[MAXN];
33 int LIS(int a[], int n)
34 {
35     clr(dp, 0x3f);
36     for (int i = 0; i < n; i++) *lower_bound(dp, dp +
37         n, a[i]) = a[i];
38     return lower_bound(dp, dp + n, INF) - dp;
}

```

7.2 数位 dp

```

1 int a[20];
2 ll dp[20][state]; //不同题目状态不同
3 ll dfs(int pos, /*state变量*/, bool lead /*前导零*/,
4 bool limit /*数位上界变量*/) //不是每个题都要判断前导
5 零
6 {
7     //递归边界, 既然是按位枚举, 最低位是0, 那么pos== -1说明
8     这个数我枚举完了
9     if(pos == -1) return 1; /*这里一般返回1, 表示你枚举
10    的这个数是合法的, 那么这里就需要你在枚举时必须每一
11    位都要满足题目条件, 也就是说当前枚举到pos位, 一定
12    要保证前面已经枚举的数位是合法的. 不过具体题目不同
13    或者写法不同的话不一定要返回1 */
14    //第二个就是记忆化(在此前可能不同题目还能有一些剪枝)
15    if(!limit && !lead && dp[pos][state] != -1) return
16    dp[pos][state];
17    //常规写法都是在没有限制的条件记忆化, 这里与下面记录状
18    态是对应, 具体为什么是有条件的记忆化后面会讲*/
19    int up = limit ? a[pos] : 9; //根据limit判断枚举的
20    上界up;这个的例子前面用213讲过了
21    ll ans = 0;
22    //开始计数
23    for(int i = 0; i <= up; i++) //枚举, 然后把不同情况
24    的个数加到ans就可以了
25    {
26        if() ...
27        else if() ...
28            ans += dfs(pos - 1, /*状态转移*/, lead &&
29                i == 0, limit && i == a[pos]) //最后
30            两个变量传参都是这样写的
31            /*这里还算比较灵活, 不过做几个题就觉
32            得这里也是套路了
33            大概就是说, 我当前数位枚举的数是i, 然
34            后根据题目的约束条件分类讨论
35            去计算不同情况下的个数, 还要要根据
36            state变量来保证i的合法性, 比如题
37            目
38            要求数位上不能有62连续出现, 那么就是
39            state就是要保存前一位pre, 然后分
40            类,
41            前一位如果是6那么这意味就不能是2, 这
42            里一定要保存枚举的这个数是合法*/
43    }
44    //计算完, 记录状态
45    if(!limit && !lead) dp[pos][state] = ans;
46    /*这里对应上面的记忆化, 在一定条件下时记录, 保证一致
47    性, 当然如果约束条件不需要考虑lead, 这里就是lead
48    就完全不用考虑了*/
49    return ans;
50 }
51 ll solve(ll x)
52 {
53     int pos = 0;
54     while(x) //把数位都分解出来
55     {
56         a[pos++] = x % 10; //个人老是喜欢编号为[0, pos),
57         看不惯的就按自己习惯来, 反正注意数位边界就行
58         x /= 10;
59     }
60     return dfs(pos - 1 /*从最高位开始枚举*/, /*一系列状态
61     */, true, true); //刚开始最高位都是有限制并且有
62     前导零的, 显然比最高位还要高的一位视为0嘛
63 }

```

8 其他

8.1 Java

```

1 import java.io.BufferedReader;
2 import java.math.BigInteger;
3 import java.util.Scanner;
4
5 public class Main {
6     public static void main(String[] args) {
7         Scanner in = new Scanner(System.in);
8         Scanner cin = new Scanner(new BufferedInputStream(
9             System.in));
10        // 使用cin进行输入的时候可能会比in快一些。
11        while (in.hasNext()) { // 多组输入
12            int x = in.nextInt();
13            int n = in.nextInt();
14            int k = in.nextInt();
15            // 大数BigInteger
16            BigInteger b = new BigInteger(x + "");
17            BigInteger ans = BigInteger.valueOf(k);
18            b = b.add(b); // b = b + b
19            b = b.subtract(b); // b = b - b
20            b = b.multiply(b); // b = b * b
21            b = b.divide(b); // b = b / b
22            b = b.remainder(b); // b = b % b
23            b = b.pow(n); // b ^ n
24            if (ans.compareTo(b) == 0) { // 比较两个大数
25                System.out.println("equ");
26            }
27            // 二维
28            BigInteger c[][] = new BigInteger[110][110];
29            for (int i = 0; i <= 100; i++)
30                for (int j = 0; j <= 100; j++)
31                    c[i][j] = BigInteger.valueOf(0); // 别忘了初始化
32            String s = b.toString(k); // 大数转化成k进制的字符串
33            System.out.println(s.charAt(0)); // s[0]
34            char[] ch = s.toCharArray(); // String转化成char数组
35            for (int i = 0; i < ch.length; i++) {
36                System.out.println(ch[i]);
37            }
38        }
39    }
40 }

```

8.2 STL

8.2.1 优先队列

```

1 // empty() 如果队列为空返回真
2 // pop() 删除对顶元素
3 // push() 加入一个元素
4 // size() 返回优先队列中拥有的元素个数
5 // top() 返回优先队列队顶元素
6 // 在默认的优先队列中，优先级高的先出队。在默认的 int 型
7 // 中先出队的为较大的数。
8 priority_queue<int> q1; // 大的先出队
9 priority_queue<int, vector<int>, greater<int>> q2;
10 // 小的先出队
11 // 自定义比较函数:

```

```

10 struct cmp
11 {
12     bool operator()(int x, int y)
13     {
14         return x > y; // x小的优先级高
15         //也可以写成其他方式, 如: return p[x] > p[y];表示p[i]小的优先级高
16     }
17 };
18 priority_queue<int, vector<int>, cmp> q; // 定义方法
19 // 其中, 第二个参数为容器类型。第三个参数为比较函数。
20 // 结构体排序:
21 struct node
22 {
23     int x, y;
24     friend bool operator < (node a, node b)
25     {
26         return a.x > b.x; // 结构体中, x小的优先级高
27     }
28 };
29 priority_queue<node> q; // 定义方法
30 // 在该结构中, y为值, x为优先级。
31 // 通过自定义operator<操作符来比较元素中的优先级。
32 // 在重载" <" 时, 最好不要重载" >" , 可能会发生编译错误

```

8.3 SG 函数

8.3.1 解题模型

1. 把原游戏分解成多个独立的子游戏, 则原游戏的 SG 函数值是它的所有子游戏的 SG 函数值的异或。
即 $sg(G) = sg(G1) \oplus sg(G2) \oplus \dots \oplus sg(Gn)$ 。

2. 分别考虑没一个子游戏, 计算其 SG 值。SG 值的计算方法: (重点)

1. 可选步数为 $1 - m$ 的连续整数, 直接取模即可, $SG(x) = x \% (m + 1)$;
2. 可选步数为任意步, $SG(x) = x$;
3. 可选步数为一系列不连续的数, 用模板计算。

一般 DFS 只在打表解决不了的情况下用, 首选打表预处理。

8.3.2 打表

```

1 //f[]: 可以取走的石子个数
2 //sg[]: 0~n的SG函数值
3 //hash[]: mex{}
4 int f[MAXN], sg[MAXN], hash[MAXN];
5 void getSG(int n)
6 {
7     int i, j;
8     memset(sg, 0, sizeof(sg));
9     for(i = 1; i <= n; i++)
10     {
11         memset(hash, 0, sizeof(hash));
12         for(j = 1; f[j] <= i; j++)
13             hash[sg[i - f[j]]] = 1;
14         for(j = 0; j <= n; j++) //求mex{}中未出现的最小
15             //的非负整数
16         {
17             if(hash[j] == 0)
18             {
19                 sg[i] = j;
20                 break;
21             }
22         }
23     }
24 }

```

```

21     }
22   }
23 }

```

8.3.3 dfs

```

1  //注意 S数组要按从小到大排序 SG函数要初始化为-1 对于每个
   集合只需初始化1遍
2  //n是集合s的大小 S[i]是定义的特殊取法规则的数组
3  int s[110], sg[10010], n;
4  int SG_dfs(int x)
5  {
6      int i;
7      if(sg[x] != -1)
8          return sg[x];
9      bool vis[110];
10     memset(vis, 0, sizeof(vis));
11     for(i = 0; i < n; i++)
12     {
13         if(x >= s[i])
14         {
15             SG_dfs(x - s[i]);
16             vis[sg[x - s[i]]] = 1;
17         }
18     }
19     int e;
20     for(i = 0;; i++)
21         if(!vis[i])
22         {
23             e = i;
24             break;
25         }
26     return sg[x] = e;
27 }

```

- 找积性
- 点阵打表
- 相除
- 找循环节
- 凑量纲
- 猜想满足 $P(n)f(n) = Q(n)f(n-2) + R(n)f(n-1) + C$ ，其中 P, Q, R 都是关于 n 的二次多项式

8.4 战术研究

- 读新题的优先级高于一切
- 读完题之后必须看一遍 clarification
- 交题之前必须看一遍 clarification
- 可能有 SPJ 的题目提交前也应该尽量做到与样例输出完全一致
- A 时需要检查 INF 是否设小
- 构造题不可开场做
- 每道题需至少有两个人确认题意
- 上机之前做法需得到队友确认
- 带有猜想性质的算法应放后面写
- 当发现题目不会做但是过了一片时应冲一发暴力
- 将待写的题按所需时间放入小根堆中，每次选堆顶的题目写
- 交完题目后立马打印随后让出机器
- 写题超过半小时应考虑是否弃题
- 细节、公式等在上机前应在草稿纸上准备好，防止上机后越写越乱
- 提交题目之前应检查 $solve(n, m)$ 是否等于 $solve(m, n)$
- 检查是否所有东西都已经清空
- 对于中后期题应该考虑一人写题，另一人在一旁辅助，及时发现手误
- 最后半小时不能慌张
- 对于取模的题，在输出之前一定要再取模一次进行保险

8.5 打表找规律方法

- 直接找规律
- 差分后找规律