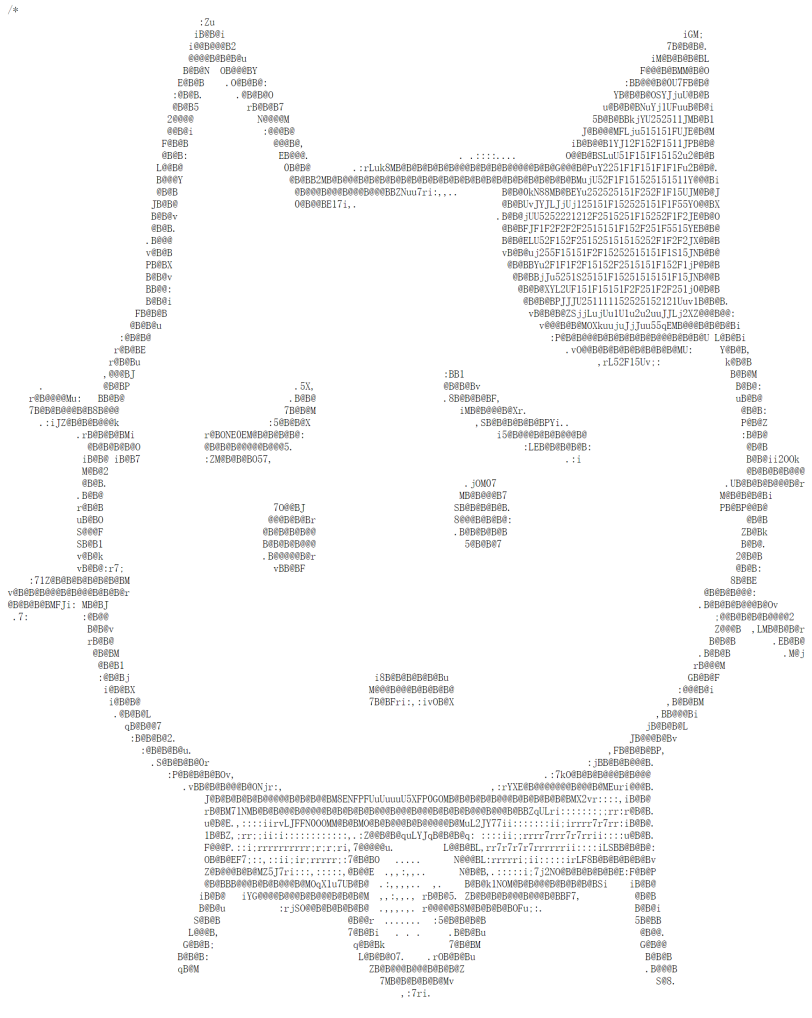
省选基础算法

雷宇辰

2017 年 3 月 22 日



目录

1 图论

1.1 有向图强连通分量的 Tarjan 算法

定义 在**有向图** G 中，如果两个顶点 u,v 间存在一条路径 u 到 v 的路径且也存在一条 v 到 u 的路径，则称这两个顶点 u,v 是**强连通的 (strongly connected)**。如果有向图 G 的每两个顶点都强连通，称 G 是一个**强连通图**。有向非强连通图的极大强连通子图，称为**强连通分量 (strongly connected components)**。若将有向图中的强连通分量都缩为一个点，则原图会形成一个 DAG（有向无环图）。

极大强连通子图 G 是一个极大强连通子图当且仅当 G 是一个强连通子图且不存在另一个强连通子图 G' 使得 G 是 G' 的真子集。

Tarjan 算法 定义 $dfn(u)$ 为结点 u 搜索的次序编号，给出函数 $low(u)$ 使得

$low(u) = min$
{
$dfn(u)$,
$low(v)$, (u,v) 为树枝边，u 为 v 的父结点
$dfn(v)$ (u,v) 为后向边或指向栈中结点的横叉边
}

当结点 u 的搜索过程结束后，若 $dfn(u) = low(u)$，则以 u 为根的搜索子树上所有还在栈中的结点是一个强连通分量。

代码

tarjan - SCC

```
void tarjan(int u)
{
    dfn[u] = low[u] = ++idx;
    st[top++] = u;
    for (Edge cur : G[u])
        if (!dfn[cur.to])
            tarjan(cur.to),
            low[u] = min(low[u], low[cur.to]);
        else if (!scc[cur.to])
            low[u] = min(low[u], dfn[cur.to]);
    if (dfn[u] == low[u] && ++cnt)
        do scc[st[--top]] = cnt;
        while (st[top] != u);
}
```

练习题

POJ2186/BZOJ1051 - Popular Cows 双倍的快乐

Popular Cows

```
#include <cstdio>
inline int min(int a, int b) { return a < b ? a : b; }
```

```
int head[10010], next[50010], to[50010], ecnt;
int dfn[10010], low[10010], stk[10010], scc[10010], top, idx, scccnt;
bool instk[10010];
int deg[10010];
inline void addEdge(int f, int t)
{
    ecnt++;
    next[ecnt] = head[f];
    head[f] = ecnt;
    to[ecnt] = t;
}
void tarjan(int x)
{
    dfn[x] = low[x] = ++idx;
    instk[stk[top++] = x] = true;
    for (int cur = head[x]; cur; cur = next[cur])
        if (!dfn[to[cur]])
            tarjan(to[cur]), low[x] = min(low[x], low[to[cur]]);
        else if (instk[to[cur]])
            low[x] = min(low[x], dfn[to[cur]]);
    if (dfn[x] == low[x])
    {
        scccnt++;
        do
        {
            top--;
            scc[stk[top]] = scccnt;
            instk[stk[top]] = false;
        } while (stk[top] != x);
    }
}
int main()
{
    int n, m;
    scanf("%d%d", &n, &m);
    for (int i = 0, x, y; i < m; i++)
    {
        scanf("%d%d", &x, &y);
        addEdge(x, y);
    }
    for (int i = 1; i <= n; i++)
        if (!dfn[i])
            tarjan(i);
    for (int i = 1; i <= n; i++)
        for (int cur = head[i]; cur; cur = next[cur])
            if (scc[i] != scc[to[cur]])
                deg[scc[i]]++;
    int zcnt = 0, id = 0;
    for (int i = 1; i <= scccnt; i++)
        if (deg[i] == 0)
            zcnt++, id = i;
    if (zcnt != 1)
```

```
        putchar('0');
    else
    {
        int ans = 0;
        for (int i = 1; i <= n; i++)
            if (scc[i] == id)
                ans++;
        printf("%d", ans);
    }
    return 0;
}
```

POJ3180 - The Cow Prom The $N(2 <= N <= 10,000)$ cows are so excited.

The Cow Prom

```
#include <cstdio>
inline int min(int a, int b) { return a < b ? a : b; }
const int maxn = 100010;
int head[maxn], next[maxn << 1], to[maxn << 1], ecnt, n, m;
int dfn[maxn], scc[maxn], cnt[maxn], scccnt, stk[maxn], low[maxn], idx, top;
inline void addEdge(int f, int t)
{
    ecnt++;
    next[ecnt] = head[f];
    head[f] = ecnt;
    to[ecnt] = t;
}
void tarjan(int x)
{
    dfn[x] = low[x] = ++idx;
    stk[top++] = x;
    for (int i = head[x]; i; i = next[i])
        if (!dfn[to[i]])
            tarjan(to[i]), low[x] = min(low[x], low[to[i]]);
        else if (!scc[to[i]])
            low[x] = min(low[x], dfn[to[i]]);
    if (dfn[x] == low[x])
    {
        scccnt++;
        do
            scc[stk[--top]] = scccnt;
        while (stk[top] != x);
    }
}
int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 0, x, y; i < m; i++)
    {
        scanf("%d%d", &x, &y);
        addEdge(x, y);
```

```
    }
    for (int i = 1; i <= n; i++)
        if (!dfn[i]) tarjan(i);
    int ans = 0;
    for (int i = 1; i <= n; i++) cnt[scc[i]]++;
    for (int i = 1; i <= scccnt; i++)
        if (cnt[i] > 1) ans++;
    printf("%d", ans);
    return 0;
}
```

POJ1236 - Network of Schools 强连通分量缩点求出度为 0 的和入度为 0 的分量个数

Network of Schools

```
#include <cstdio>
inline int min(int a, int b) { return a < b ? a : b; }
const int maxn = 110, maxm = 10100;
int head[maxn], next[maxm], to[maxm], ecnt, f[maxn], g[maxn];
inline void addEdge(int f, int t)
{
    ecnt++;
    next[ecnt] = head[f];
    head[f] = ecnt;
    to[ecnt] = t;
}
int dfn[maxn], low[maxn], stk[maxn], scc[maxn], scccnt, top, idx;
void tarjan(int x)
{
    dfn[x] = low[x] = ++idx;
    stk[top++] = x;
    for (int i = head[x]; i; i = next[i])
        if (!dfn[to[i]])
            tarjan(to[i]), low[x] = min(low[x], low[to[i]]);
        else if (!scc[to[i]])
            low[x] = min(low[x], dfn[to[i]]);
    if (dfn[x] == low[x])
    {
        scccnt++;
        do
            scc[stk[--top]] = scccnt;
        while (stk[top] != x);
    }
}
int main()
{
    int n;
    scanf("%d", &n);
    for (int i = 1, x; i <= n; i++)
        for (scanf("%d", &x); x; scanf("%d", &x))
            addEdge(i, x);
    for (int i = 1; i <= n; i++)
```

```
        if (!dfn[i]) tarjan(i);
    for (int i = 1; i <= n; i++)
        for (int j = head[i]; j; j = next[j])
            if (scc[i] != scc[to[j]])
                f[scc[i]]++, g[scc[to[j]]]++;
    int ans1 = 0, ans2 = 0;
    if (scccnt == 1)
        printf("1\n0");
    else
    {
        for (int i = 1; i <= scccnt; i++)
            ans1 += f[i] == 0, ans2 += g[i] == 0;
        printf("%d\n%d", ans2, ans1 > ans2 ? ans1 : ans2);
    }
    return 0;
}
```

1.2 图的割点、桥与双连通分量

定义

点连通度与边连通度 在一个**无向连通图**中，如果有一个顶点集合 V，删除顶点集合 V，以及与 V 中顶点相连（至少有一端在 V 中）的所有边后，原图**不连通**，就称这个点集 V 为**割点集合**。
一个图的**点连通度**的定义为：最小割点集合中的顶点数。
类似的，如果有一个边集合，删除这个边集合以后，原图不连通，就称这个点集为**割边集合**。

双连通图、割点与桥 如果一个无向连通图的**点连通度大于** 1，则称该图是**点双连通的 (point biconnected)**，简称双连通或重连通。一个图有**割点**，当且仅当这个图的点连通度为 1，则割点集合的唯一元素被称为**割点 (cut point)**，又叫关节点 (articulation point)。一个图可能有多个割点。
如果一个无向连通图的**边连通度大于** 1，则称该图是**边双连通的 (edge biconnected)**，简称双连通或重连通。一个图有**桥**，当且仅当这个图的边连通度为 1，则割边集合的唯一元素被称为**桥 (bridge)**，又叫关节边 (articulation edge)。一个图可能有多个桥。
可以看出，点双连通与边双连通都可以简称为双连通，它们之间是有着某种联系的，下文中提到的双连通，均既可指点双连通，又可指边双连通。（但这并不意味着它们等价）
双连通分量（分支）：在图 G 的所有子图 G' 中，如果 G' 是双连通的，则称 G' 为双连通子图。如果一个双连通子图 G' 它不是任何一个双连通子图的真子集，则 G' 为极大双连通子图。双连通分量 (biconnected component)，或重连通分量，就是图的极大双连通子图。特殊的，点双连通分量又叫做块。

Tarjan 算法 给出函数 $low(u)$ 使得
$low(u) = min$
{
$dfn(u)$,
$low(v)$, (u, v) 为树枝边 (父子边)
$dfn(v)$ (u, v) 为后向边 (返祖边) 等价于 $dfn(v) < dfn(u)$ 且 v 不为 u 的父亲结点
}

代码

tarjan - BCC

```
void tarjan(int u, int p)
{
    dfn[u] = low[u] = ++idx;
    for (int e = head[u]; e; e = next[e])
        if (!dfn[to[e]])
            tarjan(to[e], u), low[u] = min(low[u], low[to[e]]);
        else if (to[e] != p)
            low[u] = min(low[u], dfn[to[e]]);
}
```

练习题

POJ3177 - Redundant Paths 将一张有桥图通过加边变成边双连通图，至少要加 $\frac{leaf+1}{2}$ 条边。

Redundant Paths

```
#include <cstdio>
inline int min(int a, int b) { return a < b ? a : b; }
int head[5010], to[20010], next[20010], ecnt, map[5010][5010];
int dfn[5010], low[5010], idx, cnt[5010];
void addEdge(int f, int t)
{
    ecnt++;
    next[ecnt] = head[f];
    head[f] = ecnt;
    to[ecnt] = t;
}
void tarjan(int u, int p)
{
    dfn[u] = low[u] = ++idx;
    for (int e = head[u]; e; e = next[e])
        if (!dfn[to[e]])
            tarjan(to[e], u), low[u] = min(low[u], low[to[e]]);
        else if (to[e] != p)
            low[u] = min(low[u], dfn[to[e]]);
}
int main()
{
    int n, m;
    scanf("%d%d", &n, &m);
    for (int i = 1, x, y; i <= m; i++)
    {
        scanf("%d%d", &x, &y);
        if (!map[x][y])
        {
            addEdge(x, y);
            addEdge(y, x);
            map[x][y] = map[y][x] = true;
```

```
        }
    }
    tarjan(1, 0);
    for (int i = 1; i <= n; i++)
        for (int e = head[i]; e; e = next[e])
            if (low[to[e]] != low[i])
                cnt[low[i]]++;
    int ans = 0;
    for (int i = 1; i <= n; i++)
        ans += cnt[i] == 1;
    printf("%d", (ans + 1) >> 1);
    return 0;
}
```

POJ1523 - SPF 求割点与删除这个点之后有多少个连通分量

Redundant Paths

```
#include <cstdio>
#include <cctype>
#include <cstring>
#define clz(X) memset(X, 0, sizeof(X))
inline int max(int a, int b) { return a > b ? a : b; }
inline int min(int a, int b) { return a < b ? a : b; }
inline void read(int &x)
{
    int ch = x = 0;
    while (!isdigit(ch = getchar()));
    for (; isdigit(ch); ch = getchar())
        x = x * 10 + ch - '0';
}
int map[1010][1010], range;
int dfn[1010], low[1010], idx;
int son, subnet[1010];
void tarjan(int u)
{
    dfn[u] = low[u] = ++idx;
    for (int v = 1; v <= range; v++)
        if (map[u][v])
            if (!dfn[v])
            {
                tarjan(v);
                low[u] = min(low[u], low[v]);
                if (low[v] >= dfn[u])
                    (u == 1 ? son : subnet[u])++;
            }
            else
                low[u] = min(low[u], dfn[v]);
}
int main()
{
    int x, y, T = 0;
```

```
    while (read(x), x)
    {
        clz(map), clz(dfn), clz(low), clz(subnet), son = idx = 0;
        read(y);
        map[x][y] = map[y][x] = 1;
        range = max(x, y);
        while (read(x), x)
        {
            read(y);
            map[x][y] = map[y][x] = 1;
            range = max(range, max(x, y));
        }
        printf("Network #%d\n", ++T);
        tarjan(1);
        bool flag = false;
        if (son > 1) subnet[1] = son - 1;
        for (int i = 1; i <= range; i++)
            if (subnet[i])
                printf("  SPF node %d leaves %d subnets\n", i, subnet[i] + 1),
                flag = true;
        if (!flag)
            puts("  No SPF nodes");
        putchar('\n');
    }
    return 0;
}
```

POJ2942 - Knights of the Round Table 这题过于复杂，我来先给个别人的题解。然后是我自己的实现（仿佛还是没看懂。
//实现被狗吃了

1.3 2-SAT

定义 给定一个布尔方程，判断是否存在一组布尔变量的取值方案，使得整个方程值为真的问题，被称为布尔方程的可满足性问题 (SAT)。SAT 问题是 NP 完全的，但对于一些特殊形式的 SAT 问题我们可以有效求解。
我们将下面这种布尔方程称为合取范式：

$$(a \vee b \vee c \vee \cdots) \wedge (d \vee e \vee f \vee \cdots) \wedge \cdots$$

其中 $a, b, c, \cdots$ 称为文字，它是一个布尔变量或其否定。像 $(a \vee b \vee c \vee \cdots)$ 这样用 $\vee$ 连接的部分称为子句。如果合取范式的每个子句中的文字个数都不超过两个，那么对应的 SAT 问题又称为 **2-SAT** 问题。

解法 对于给定的 **2-SAT** 问题，首先利用 $\Rightarrow$ 将每个子句 $(a \vee b)$ 改写成等价形式 $(\neg a \Rightarrow b \wedge a \Rightarrow \neg b)$. 这样原布尔公式就变成了把 $a \Rightarrow b$ 形式的布尔公式用 $\wedge$ 连接起来的形式。
对每个布尔变量 x 构造两个顶点分别代表 x 与 $\neg x$。以 $\Rightarrow$ 关系为边建立有向图。若在此图中 a 点能到达 b 点，就表示 a 为真时 b 也一定为真。因此该图中同一个强连通分量中所含的所有变量的布尔值均相同。
若存在某个变量 x, 代表 x 与 $\neg x$ 的两个顶点在同一个强连通分量中，则原布尔表达式的值无法为真。
反之若不存在这样的变量，那么我们先将原图中所有的强连通分量缩为一个点，构出一个新图，新图显然

是一个拓扑图，我们求出它的一个拓扑序。那么对于每个变量 x，

$$x\text{所在的强连通分量（新图中的点）的拓扑序在}\neg x\text{所在的强连通分量之后} \Leftrightarrow x\text{为真}$$

就是一组合适布尔变量赋值。注意到 Tarjan 算法所求的强连通分量就是按拓扑序的逆序得出的，因此不需要真的缩点建新图求拓扑序，直接利用强连通分量的编号来当做顺序即可。

练习题

POJ3648 - Wedding Additionally, there are several pairs of people conducting adulterous relationships (both different-sex and same-sex relationships are possible)

adulterous relationships

```
#include <cstdio>
#include <cstring>
inline int min(int a, int b) { return a < b ? a : b; }
const int maxn = 2010, maxm = 500010;
int head[maxn], next[maxm], to[maxm], ecnt;
inline void addEdge(int f, int t)
{
    next[ecnt] = head[f];
    head[f] = ecnt;
    to[ecnt] = t;
    ecnt++;
}
int dfn[maxn], low[maxn], stk[maxn], scc[maxn], top, idx, scccnt;
void tarjan(int x)
{
    dfn[x] = low[x] = ++idx;
    stk[top++] = x;
    for (int i = head[x]; ~i; i = next[i])
        if (!dfn[to[i]])
            tarjan(to[i]), low[x] = min(low[x], low[to[i]]);
        else if (!scc[to[i]])
            low[x] = min(low[x], dfn[to[i]]);
    if (dfn[x] == low[x])
    {
        scccnt++;
        do
            scc[stk[--top]] = scccnt;
        while (stk[top] != x);
    }
}
int main()
{
    int m, n;
    while (scanf("%d%d", &n, &m) && (m + n))
    {
        memset(head, -1, sizeof(head));
        memset(dfn, 0, sizeof(dfn));
        memset(low, 0, sizeof(low));
        memset(scc, 0, sizeof(scc));
```

```
        idx = top = ecnt = scccnt = 0;
        int a1, a2;
        char c1, c2;
        for (int i = 0; i < m; i++)
        {
            scanf("%d%c %d%c", &a1, &c1, &a2, &c2);
            a1 = a1 << 1 | (c1 == 'h'), a2 = a2 << 1 | (c2 == 'h');
            addEdge(a1, a2 ^ 1), addEdge(a2, a1 ^ 1);
        }
        addEdge(0, 1);
        for (int i = 0; i < (n << 1); i++)
            if (!dfn[i]) tarjan(i);
        bool flag = true;
        for (int i = 0; i < n && flag; i++)
            if (scc[i << 1] == scc[i << 1 | 1])
                flag = false;
        if (!flag)
            puts("bad luck");
        else if (n < 1)
            putchar('\n');
        else
            for (int i = 1; i < n; i++)
                printf("%d%c%c", i, (scc[i << 1] > scc[i << 1 | 1]) ? 'w' : 'h', " \n"[i
                    == n - 1]);
    }
    return 0;
}
```

POJ3678 - Katu Puzzle 我什么时候做过这个题？

Katu Puzzle

```
#include <cstdio>
#include <cstring>
inline int min(int a, int b) { return a < b ? a : b; }
const int maxn = 10010, maxm = 4000010;
int head[maxn], next[maxm], to[maxm], ecnt;
inline void addEdge(int f, int t)
{
    next[ecnt] = head[f];
    head[f] = ecnt;
    to[ecnt] = t;
    ecnt++;
}
int dfn[maxn], low[maxn], stk[maxn], scc[maxn], top, idx, scccnt;
void tarjan(int x)
{
    dfn[x] = low[x] = ++idx;
    stk[top++] = x;
    for (int i = head[x]; ~i; i = next[i])
        if (!dfn[to[i]])
            tarjan(to[i]), low[x] = min(low[x], low[to[i]]);
```

```
        else if (!scc[to[i]])
            low[x] = min(low[x], dfn[to[i]]);
    if (dfn[x] == low[x])
    {
        scccnt++;
        do
            scc[stk[--top]] = scccnt;
        while (stk[top] != x);
    }
}
int main()
{
    int n, m;
    while (~scanf("%d%d", &n, &m))
    {
        memset(dfn, 0, sizeof(dfn));
        memset(low, 0, sizeof(low));
        memset(scc, 0, sizeof(scc));
        memset(head, -1, sizeof(head));
        ecnt = top = idx = scccnt = 0;
        for (int i = 0, u, v, w; i < m; ++i)
        {
            char op[5];
            scanf("%d%d%d%s", &u, &v, &w, op);
            if (op[0] == 'A')
                if (w)
                {
                    addEdge(u << 1, v << 1 | 1), addEdge(v << 1, u << 1 | 1);
                    addEdge(u << 1, v << 1), addEdge(v << 1 | 1, u << 1 | 1);
                    addEdge(u << 1 | 1, v << 1 | 1), addEdge(v << 1, u << 1);
                }
                else
                {
                    addEdge(u << 1 | 1, v << 1), addEdge(v << 1 | 1, u << 1);
                }
            if (op[0] == 'O')
                if (w)
                {
                    addEdge(u << 1, v << 1 | 1), addEdge(v << 1, u << 1 | 1);
                }
                else
                {
                    addEdge(u << 1, v << 1), addEdge(v << 1 | 1, u << 1 | 1);
                    addEdge(u << 1 | 1, v << 1 | 1), addEdge(v << 1, u << 1);
                    addEdge(u << 1 | 1, v << 1), addEdge(v << 1 | 1, u << 1);
                }
            if (op[0] == 'X')
                if (w)
                {
                    addEdge(u << 1, v << 1 | 1), addEdge(v << 1, u << 1 | 1);
                    addEdge(u << 1 | 1, v << 1), addEdge(v << 1 | 1, u << 1);
                }
```

```
                else
                {
                    addEdge(u << 1, v << 1), addEdge(v << 1 | 1, u << 1 | 1);
                    addEdge(u << 1 | 1, v << 1 | 1), addEdge(v << 1, u << 1);
                }
        }
        for (int i = 0; i < (n << 1); i++)
            if (!dfn[i]) tarjan(i);
        bool flag = true;
        for (int i = 0; i < n && flag; i++)
            if (scc[i << 1] == scc[i << 1 | 1])
                flag = false;
        puts(flag ? "YES" : "NO");
    }
    return 0;
}
```

POJ2749 - Building roads 杀光奶牛问题就会得到解决

Building roads

```
#include <cstdio>
#include <cstring>
inline int abs(int x) { return x >= 0 ? x : -x; }
inline int min(int a, int b) { return a < b ? a : b; }
const int inf = 0x3f3f3f3f, maxn = 10010, maxm = 1200010;
int head[maxn], next[maxm], to[maxm], ecnt, n, A, B;
int dfn[maxn], low[maxn], stk[maxn], scc[maxn], top, idx, scccnt;
int sx1, sy1, sx2, sy2, sLen, X[maxn], Y[maxn], hate[maxn][2], like[maxn][2],
d[maxn];
inline void addEdge(int f, int t)
{
    next[ecnt] = head[f];
    head[f] = ecnt;
    to[ecnt] = t;
    ecnt++;
}
void tarjan(int x)
{
    dfn[x] = low[x] = ++idx;
    stk[top++] = x;
    for (int i = head[x]; ~i; i = next[i])
        if (!dfn[to[i]])
            tarjan(to[i]), low[x] = min(low[x], low[to[i]]);
        else if (!scc[to[i]])
            low[x] = min(low[x], dfn[to[i]]);
    if (dfn[x] == low[x])
    {
        scccnt++;
        do
            scc[stk[--top]] = scccnt;
        while (stk[top] != x);
```

```
    }
}
bool check(int x)
{
    memset(dfn, 0, sizeof(dfn));
    memset(low, 0, sizeof(low));
    memset(scc, 0, sizeof(scc));
    memset(head, -1, sizeof(head));
    ecnt = top = idx = scccnt = 0;
    for (int i = 1; i <= n; i++)
        for (int j = i + 1; j <= n; j++)
        {
            int l1 = d[i << 1], l2 = d[i << 1 | 1];
            int r1 = d[j << 1], r2 = d[j << 1 | 1];
            if (l1 + r1 > x)
                addEdge(i << 1, j << 1 | 1), addEdge(j << 1, i << 1 | 1);
            if (l1 + r2 + sLen > x)
                addEdge(i << 1, j << 1), addEdge(j << 1 | 1, i << 1 | 1);
            if (l2 + r1 + sLen > x)
                addEdge(i << 1 | 1, j << 1 | 1), addEdge(j << 1, i << 1);
            if (l2 + r2 > x)
                addEdge(i << 1 | 1, j << 1), addEdge(j << 1 | 1, i << 1);
        }
    for (int i = 1, a, b; i <= A; i++)
    {
        a = hate[i][0], b = hate[i][1];
        addEdge(a << 1, b << 1 | 1);
        addEdge(a << 1 | 1, b << 1);
        addEdge(b << 1, a << 1 | 1);
        addEdge(b << 1 | 1, a << 1);
    }
    for (int i = 1, a, b; i <= B; i++)
    {
        a = like[i][0], b = like[i][1];
        addEdge(a << 1, b << 1);
        addEdge(a << 1 | 1, b << 1 | 1);
        addEdge(b << 1, a << 1);
        addEdge(b << 1 | 1, a << 1 | 1);
    }
    for (int i = 1; i <= (n << 1); i++)
        if (!dfn[i])
            tarjan(i);
    for (int i = 1; i <= n; i++)
        if (scc[i << 1] == scc[i << 1 | 1])
            return false;
    return true;
}
int main()
{
    scanf("%d%d%d%d%d%d%d", &n, &A, &B, &sx1, &sy1, &sx2, &sy2);
    sLen = abs(sx1 - sx2) + abs(sy1 - sy2);
    for (int i = 1; i <= n; i++)
```

```
        scanf("%d%d", X + i, Y + i);
    for (int i = 1; i <= n; i++)
        d[i << 1] = abs(X[i] − sx1) + abs(Y[i] − sy1),
        d[i << 1 | 1] = abs(X[i] − sx2) + abs(Y[i] − sy2);
    for (int i = 1; i <= A; i++)
        scanf("%d%d", &hate[i][0], &hate[i][1]);
    for (int i = 1; i <= B; i++)
        scanf("%d%d", &like[i][0], &like[i][1]);
    int l = 0, r = 8000000, m, ans = −1;
    while (l <= r)
        check(m = (l + r) >> 1) ? r = (ans = m) − 1 : l = m + 1;
    printf("%d\n", ans);
    return 0;
}
```

1.4 欧拉回路

定义 设 $G = (V, E)$ 是一个图。

欧拉回路 图 G 中经过**每条边一次**并且**仅一次**的回路称作欧拉回路。

欧拉路径 图 G 中经过**每条边一次**并且**仅一次**的路径称作欧拉路径。

欧拉图 存在**欧拉回路**的图称为欧拉图。

半欧拉图 存在欧拉路径但不存在欧拉回路的图称为半欧拉图。

性质与定理 以下不加证明的给出一些定理 ~~（因为我懒得抄讲义子~~

定理 1 无向图 G 为欧拉图，当且仅当 G 为连通图且所有顶点的度为偶数。

推论 1 无向图 G 为半欧拉图，当且仅当 G 为连通图且除了两个顶点的度为奇数之外，其它所有顶点的度为偶数。

定理 2 有向图 G 为欧拉图，当且仅当 G 的基图[1]连通，且所有顶点的入度等于出度。

推论 2 有向图 G 为半欧拉图，当且仅当 G 的基图连通，且存在顶点 u 的入度比出度大 1、v 的入度比出度小 1，其它所有顶点的入度等于出度。

解法 由此可以得到以下求欧拉图 G 的欧拉回路的算法：

1. 在图 G 中任意找一个回路 C。
2. 将图 G 中属于回路 C 的边删除
3. 在残留图的各极大连通子图中分别寻找欧拉回路。
4. 将各极大连通子图的欧拉回路合并到 C 中得到图 G 的欧拉回路。

[1] 忽略有向图所有边的方向，得到的无向图称为该有向图的基图。

该算法的伪代码如下：

```
void dfs(u)
{
    for (edge e : edges[u])
        if (!flag[e])
        {
            flag[e] = true;
            flag[rev(e)] = true; //如果图 G 是有向图则删去本行
            dfs(e.to);
            S.push(v);
        }
}
```

最后依次取出栈 S 每一条边而得到图 G 的欧拉回路 (也就是边出栈序的逆序)。由于该算法执行过程中每条边最多访问两次，因此该算法的时间复杂度为 $O(|E|)$。

练习题

UOJ117 - 欧拉回路 混合两个子任务使代码风格变得鬼畜起来。

Building roads

```
#include <cstdio>
#include <cctype>
inline void read(int &x)
{
    int ch = x = 0;
    while (!isdigit(ch = getchar()));
    for (; isdigit(ch); ch = getchar()) x = x * 10 + ch - '0';
}
const int N = 100000 + 10, E = N << 2;
int adj[N], to[E], nxt[E], ecnt = 1;
int out[N], in[N], t, n, m;
bool flag[E];
int ans[E], tail;
inline void addEdge(int u, int v)
{
    ecnt++;
    nxt[ecnt] = adj[u];
    adj[u] = ecnt;
    to[ecnt] = v;
}
void dfs(int u)
{
    for (int &i = adj[u]; i; i = nxt[i])
    {
        int c = t == 1 ? i >> 1 : i - 1;
        bool sig = i & 1;
        if (!flag[c])
        {
```

```
            flag[c] = true;
            dfs(to[i]);
            ans[tail++] = (t == 1 && sig) ? -c : c;
        }
    }
}
int main()
{
    read(t), read(n), read(m);
    for (int i = 0, u, v; i < m; i++)
    {
        read(u), read(v);
        addEdge(u, v);
        out[u]++, in[v]++;
        if (t == 1) addEdge(v, u);
    }
    for (int i = 1; i <= n; i++)
        if (t == 1 ? ((in[i] + out[i]) & 1) : (in[i] != out[i]))
            return puts("NO"), 0;
    for (int i = 1; i <= n; i++)
        if (adj[i])
        {
            dfs(i);
            break;
        }
    if (tail != m) return puts("NO"), 0;
    puts("YES");
    for (int i = m - 1; i >= 0; i--) printf("%d ", ans[i]);
    return 0;
}
```

2 字符串（一）

2.1 KMP

算法介绍 用来在线性时间内匹配字符串

算法流程 我觉得鑝鑝鑝在 WC 上讲的比较清楚，于是我开始抄讲义。

字符串： $s[1\dots n], |s| = n$。

子串： $s[i\dots j] = s[i]s[i+1]\cdots[j]$。

前缀： $pre(s,x) = s[1\dots x]$, 后缀：$suf(s,x) = s[n-x+1\dots n]$

若 $0 \le r \le |s|, pre(s,r) = suf(s,r)$, 就称 $pre(s,r)$ 是 s 的 **border**。

KMP 算法的第一步主要做这么一件事：在 $O(n)$ 时间求出数组 $next[1\dots n]$, 其中 $next[i]$ 表示前缀 $s[1\dots i]$ 的最大 border 长度。于是可以知道 s 的所有 border 长度为 $next[n], next[next[n]], \cdots$，我想这是显然的，于是不加证明的在这里给出。

第二步就是匹配，如果失配了就把模式串的当前位置指针 i 跳到 $next[i]$ 处然后继续匹配，然后就好了。

算法实现

genNext

```
for (int i = 1, j = -1; i < len; i++)
{
    while (~j && str[j + 1] != str[i]) j = next[j];
    if (str[j + 1] == str[i]) j++;
    next[i] = j;
}
```

Find

```
for (int i = 0, j = -1; i < len; i++)
{
    while (~j && t[j + 1] != s[i]) j = next[j];
    if (t[j + 1] == s[i]) j++;
    if (j == len - 1) ans++, j = next[j];
}
```

练习题

POJ3461 - Oulipo 求出所有匹配位置

Oulipo

```
#include <cstdio>
#include <cstring>
char a[1 << 20 | 1], b[1 << 14 | 1];
int la, lb;
int next[1 << 14 | 1];
int main()
{
    next[0] = -1;
    int n;
    scanf("%d", &n);
    while (n--)
    {
        scanf("%s%s", b, a);
        la = strlen(a), lb = strlen(b);
        for (int i = 1, j = -1; i < lb; i++)
        {
            while (~j && b[j + 1] != b[i]) j = next[j];
            if (b[j + 1] == b[i]) j++;
            next[i] = j;
        }
        int ans = 0;
        for (int i = 0, j = -1; i < la; i++)
        {
            while (~j && b[j + 1] != a[i]) j = next[j];
            if (b[j + 1] == a[i]) j++;
            if (j == lb - 1) ans++, j = next[j];
        }
        printf("%d\n", ans);
    }
```

```
    return 0;
}
```

POJ2406 - Power Strings *next* 数组的奇妙性质

Power Strings

```
#include <cstdio>
#include <cstring>
char str[1 << 20 | 1];
int next[1 << 20 | 1];
int len;
int main()
{
    next[0] = -1;
    while (scanf("%s", str))
    {
        if (str[0] == '.') break;
        len = strlen(str);
        for (int i = 0, j = -1; i < len;)
            (~j && str[i] != str[j]) ? j = next[j] : next[++i] = ++j;
        printf("%d\n", len % (len - next[len]) == 0 ? len / (len - next[len]) : 1);
    }
    return 0;
}
```

CF526D - Om Nom and Necklace 啥？

Om Nom and Necklace

```
#include <cstdio>
#include <cstring>
char s[1000010];
int next[1000010], n, k, len;
int main()
{
    scanf("%d%d%s", &n, &k, s);
    next[0] = -1;
    len = strlen(s);
    for (int i = 1; i < len; ++i)
    {
        int j;
        for (j = next[i - 1]; j != -1 && s[j + 1] != s[i]; j = next[j]);
        if (s[j + 1] == s[i]) j++;
        next[i] = j;
    }
    for (int i = 0; i < len; ++i)
    {
        int p = i + 1, q = p / (i - next[i]);
        putchar(((p % (i - next[i]) == 0) ? (q / k >= q % k ? '1' : '0') : (q / k > q % k
            ? '1' : '0')) );
    }
    return 0;
```

```
}
```

讲道理我 KMP 真的学的不是很明白，望各位 dalao 给予指导。

2.2 Trie

简介 字典树，也称 Trie、字母树，指的是某个字符串集合对应的形如下图的有根树。树的每条边上对应有恰好一个字符，每个顶点代表从根到该节点的路径所对应的字符串（将所有经过的边上的字符按顺序连接起来）。

实现 水

Trie - impl

```
struct node
{
    node *trans[26];
    int cnt;
};
void insert(node *n, char *str)
{
    for (; *str; n = n->trans[*str - '0'])
        if (n->trans[*str - '0'] == 0)
            n->trans[*str - '0'] = new_node();
    n->cnt++;
}
```

练习题

POJ3630 - Phone List 若插入过程中，有某个经过的节点带有串结尾标记，则之前插入的某个串是当前串的前缀。

Oulipo

```
#include <cstdio>
#include <cstring>
struct node
{
    node *trans[10];
    bool is_end;
} nodes[100010];
node *root;
int cnt;
node *new_node() { return &nodes[cnt++]; }
char buf[11];
bool try_insert(node *n, char *str)
{
    if (n->is_end) return false;
    if (*str == '\0')
    {
        for (int i = 0; i < 10; i++)
            if (n->trans[i])
                return false;
```

```
        n->is_end = true;
        return true;
    }
    if (n->trans[*str - '0'] == 0) n->trans[*str - '0'] = new_node();
    return try_insert(n->trans[*str - '0'], str + 1);
}
int main()
{
    int t, n;
    scanf("%d", &t);
    while (t--)
    {
        memset(nodes, 0, sizeof(nodes));
        cnt = 0;
        root = new_node();
        scanf("%d", &n);
        bool flag = true;
        while (n--)
        {
            scanf("%s", buf);
            if (flag) flag = try_insert(root, buf);
        }
        puts(flag ? "YES" : "NO");
    }
    return 0;
}
```

POJ2945 - Find the Clones n 个基因片段，每个长度为 m，输出 n 行表示重复出现 i 次 $(1 \le i \le n)$ 的基因片段的个数

Find the Clones

```
#include <cstdio>
#include <cstring>
struct node
{
    node *trans[4];
    int cnt;
} nodes[400010];
int tot;
node *root;
inline node *new_node() { return &nodes[tot++]; }
void try_insert(node *n, char *str)
{
    if (*str == '\0')
        n->cnt++;
    else
    {
        if (n->trans[*str - '0'] == 0) n->trans[*str - '0'] = new_node();
        try_insert(n->trans[*str - '0'], str + 1);
    }
}
```

```
char f[1 << 8 | 1];
int ans[20010];
int main()
{
    f['A'] = '0', f['C'] = '1', f['G'] = '2', f['T'] = '3';
    int n, m;
    char buf[22];
    while (~scanf("%d%d", &n, &m) && (n + m))
    {
        memset(ans, 0, sizeof(ans));
        memset(nodes, 0, sizeof(nodes));
        tot = 0;
        root = new_node();
        for (int i = 0; i < n; i++)
        {
            scanf("%s", buf);
            for (int j = 0; j < m; j++)
                buf[j] = f[buf[j]];
            try_insert(root, buf);
        }
        for (int i = 0; i < tot; i++) ans[nodes[i].cnt]++;
        for (int i = 1; i <= n; i++)
            printf("%d\n", ans[i]);
    }
    return 0;
}
```

2.3 Aho-Corasick Automaton

简介 多模式串字符串匹配，Trie 上的 KMP，其中 $next$ 数组变成了 $fail$ 指针，功能相同。

实现 Trie 的实现上文已经出现，所以此处不再重复。

buildFail

```
void buildFail()
{
    int h = 0, t = 0;
    root->fail = &virt;
    que[t++] = root;
    while (h < t)
    {
        node *cur = que[h++];
        for (int i = 0; i < 26; i++)
        {
            node *f = cur->fail;
            while (f->trans[i] == 0) f = f->fail;
            f = f->trans[i];
            if (cur->trans[i])
                (que[t++] = cur->trans[i])->fail = f;
            else
                cur->trans[i] = f;
```

```
        }
    }
}
```

练习题

HDU2222 - Keywords Search AC 自动机模板题，注意统计答案时，每个节点只能统计一次不要重复统计。

Keywords Search

```
#include <cstdio>
#include <cstring>
struct node
{
    node *trans[26], *fail;
    int cnt;
} nodes[500010], virt;
node *que[500010];
int tot;
node *root;
node *new_node() { return &nodes[tot++]; }
void insert(char *str)
{
    node *cur = root;
    for (; *str; str++)
    {
        if (cur->trans[*str - 'a'] == 0)
            cur->trans[*str - 'a'] = new_node();
        cur = cur->trans[*str - 'a'];
    }
    cur->cnt++;
}
void buildFail()
{
    int h = 0, t = 0;
    root->fail = &virt;
    que[t++] = root;
    while (h < t)
    {
        node *cur = que[h++];
        for (int i = 0; i < 26; i++)
        {
            node *f = cur->fail;
            while (f->trans[i] == 0) f = f->fail;
            f = f->trans[i];
            if (cur->trans[i])
                (que[t++] = cur->trans[i])->fail = f;
            else
                cur->trans[i] = f;
        }
    }
```

```
}
char buf[1000010];
int vis[500010];
int main()
{
    memset(vis, -1, sizeof(vis));
    int T, n;
    scanf("%d", &T);
    while (T--)
    {
        memset(nodes, 0, sizeof(nodes));
        tot = 0, root = new_node();
        for (int i = 0; i < 26; i++) virt.trans[i] = root;
        scanf("%d", &n);
        for (int i = 0; i < n; i++)
        {
            scanf("%s", buf);
            insert(buf);
        }
        buildFail();
        scanf("%s", buf);
        node *cur = root, *tmp;
        int ans = 0;
        for (char *ch = buf; *ch; ch++)
        {
            tmp = cur = cur->trans[*ch - 'a'];
            while (tmp != &virt && vis[tmp - nodes] != T)
            {
                vis[tmp - nodes] = T;
                ans += tmp->cnt;
                tmp = tmp->fail;
            }
        }
        printf("%d\n", ans);
    }
    return 0;
}
```

2.4 Manacher

求出字符串每一位的回文半径，算法流程奥妙重重，不易让常人理解，然后我就抄了份板子改了改，然后就比讲义上的标程快了 20%。

练习题

POJ3974 - Palindrome Manacher 模板题

Palindrome

```
#include <cstdio>
#include <cstring>
inline int min(int a, int b) { return a < b ? a : b; }
```

```
inline int max(int a, int b) { return a > b ? a : b; }
char buf[1000100], str[2000200];
int R[2000200], T;
int main()
{
    while (scanf("%s", buf), buf[0] != 'E')
    {
        int m = int(strlen(buf)), n = 0;
        str[n++] = '!';
        str[n++] = '#';
        for (int i = 0; i < m; i++)
            str[n++] = buf[i], str[n++] = '#';
        str[n++] = '#';
        str[n++] = '?';
        int p = 0, mx = 0, ans = 0;
        for (int i = 1; i < n; i++)
        {
            R[i] = mx > i ? min(R[2 * p - i], mx - i) : 1;
            while (str[i + R[i]] == str[i - R[i]]) R[i]++;
            if (R[i] + i > mx)
                mx = i + R[p = i];
            ans = max(ans, R[i]);
        }
        printf("Case %d: %d\n", ++T, ans - 1);
    }
    return 0;
}
```

3 简单数学

说是简单数学其实我后半部分也没看懂

3.1 整除及剩余

整除定义 设 a, b 是两个整数，且 $b \neq 0$. 如果存在整数 c，使得 $a = bc$，则称 a 被 b 整除，或 b 整除 a，记作 $b|a$。此时，又称 a 是 b 的倍数，b 是 a 的因子。

整除的基本性质

1. $a|b \wedge a|c \Rightarrow a|(b+c)$
2. $a|b \Rightarrow \forall c \in \mathbb{Z},\ a|bc$
3. $a|b \wedge b|c \Rightarrow a|c$

同余基本定义和定理

定义 1：带余除法

$$\forall a \in \mathbb{Z}, b \in \mathbb{Z}^* \rightarrow \exists q, r \in \mathbb{Z}, a = qb + r, r \in [0, |b|)$$

定义 2：同余

$$a \bmod m = b \bmod m \iff a \equiv b \pmod m$$

定义 3：剩余类

$$\mathbf{A_i} = \{x | x \in \mathbb{Z} \wedge x \bmod m = i\} \to \forall a, b \in \mathbf{A_i}, a \equiv b \pmod m$$

定义 4：完系

$$\{a_1 \bmod m, a_2 \bmod m, \ldots, a_n \bmod m\} = \{0, 1, 2, \ldots, m-1\}$$

定理 1

$$a \equiv b \pmod m \iff \exists k \in \mathbb{Z}, a = b + km \iff m | (a - b)$$

定理 2 同余关系是等价关系

1. $a \equiv a \pmod m$
2. $a \equiv b \pmod m \Rightarrow b \equiv a \pmod m$
3. $a \equiv b \pmod m \wedge b \equiv c \pmod m \Rightarrow a \equiv c \pmod m$

定理 3 同余的三则运算

$a, b, c \in \mathbb{R}, m \in \mathbb{N}^*, a \equiv b \pmod m \Rightarrow$

1. $a + c \equiv b + c \pmod m$
2. $a - c \equiv b - c \pmod m$
3. $ac \equiv bc \pmod m$

定理 4 同余式的三则运算

$a, b, c \in \mathbb{R}, m \in \mathbb{N}^*, a \equiv b \pmod m \wedge c \equiv d \pmod m \Rightarrow$

1. $ax + cy \equiv bx + dy \pmod m \qquad x, y \in \mathbb{Z}$
2. $ac \equiv bd \pmod m$
3. $a^n \equiv b^n \pmod m \qquad n \in \mathbb{N}^*$
4. $f(a) \equiv f(b) \pmod m \qquad f(x)$ 为任一整系数多项式

定理 5

1. $a \equiv b \pmod m \wedge d | m \Rightarrow a \equiv b \pmod d$
2. $a \equiv b \pmod m \Rightarrow gcd(a, m) = gcd(b, m)$
3. $\forall i \in [1, n]\ , a \equiv b \pmod{m_i} \iff a \equiv b \pmod{lcm(m_1, m_2, \ldots, m_n)}$

3.2 素数

定义 素数（质数）是大于 1 的正整数，并且除了 1 和它本身不能被其他正整数整除。大于 1 的非素数的正整数称为合数。

分布 素数有无穷多个. 如果使用 $\pi(x)$ 表示小于一个正实数 x 的素数有多少个，那么有

$$\lim_{x\to\infty}\frac{\pi(x)}{x}=\ln x$$

算术基本定理/惟一分解定理

$$n=\prod_{i=1}^{\infty}p_i^{a_i}\qquad p_i\in\mathbb{P},\ a_i\in\mathbb{N}$$

判定

$$n\in\mathbb{P}\iff\forall i\in[2,\sqrt{n}],\ i\nmid n$$

Eratosthenes 筛法

Eratosthenes Sieve

```
fill(isprime, true);
for (int i = 2; i < n; i++)
    if (isprime[i])
        for (int j = i * i; j < n; j += i)
            isprime[i] = false;
```

欧拉函数 欧拉函数 $\varphi(n)$ 指不超过 n 且与 n 互素的正整数的个数，其中 n 是一个正整数。

欧拉函数的性质 $n=\prod_{i=1}^{m}p_i^{a_i}\to\varphi(n)=\prod_{i=1}^{m}\varphi(p_i^{a_i})$

定理 1 $p\in\mathbb{P}\iff\varphi(p)=p-1$

定理 2 $p\in\mathbb{P},\ a\in\mathbb{N}_+\Rightarrow\varphi(p^a)=p^a-p^{a-1}$

定理 3 $m,n\in\mathbb{N}_+\wedge gcd(m,n)=1\Rightarrow\varphi(mn)=\varphi(m)\varphi(n)$

定理 4 $n=\prod_{i=1}^{m}p_i^{a_i}\to\varphi(n)=n\prod_{i=1}^{m}(1-\frac{1}{p_i})$

推论 $n\equiv 1\pmod 2\to\varphi(2n)=\varphi(n)$

定理 5 $n\in(2,+\infty)\bigcap\mathbb{Z}\Rightarrow\varphi(n)\equiv 0\pmod 2$

定理 6 $n\in\mathbb{N}_+\Rightarrow\sum_{d|n}\varphi(n)=n$

欧拉定理 $gcd(a,m)=1,a\in\mathbb{N}_+,m\in[2,+\infty)\bigcap\mathbb{Z}\Rightarrow a^{\varphi(m)}\equiv 1\pmod m$

费马小定理 $m\in\mathbb{P}\Rightarrow a^{m-1}\equiv 1\pmod m$

练习题

POJ2689 - Prime Distance 暴力筛掉合数

Prime Distance

```
#include <cmath>
#include <cstdio>
#include <cstring>
int prime_count;
int prime[5140];
bool f[1000010];
bool notprime[50010];
int main()
{
    for (int i = 2; i < 50010; i++)
        if (!notprime[i])
            for (long long j = 1ll * i * i; j < 50010; j += i)
                notprime[j] = true;
    for (int i = 2; i < 50010; i++)
        if (!notprime[i])
            prime[prime_count++] = i;
    int l, r;
    while (~scanf("%d%d", &l, &r))
    {
        l = l == 1 ? 2 : l;
        memset(f, 0, sizeof(f));
        for (int i = 0, a, b; i < prime_count; i++)
        {
            a = (l - 1) / prime[i] + 1;
            b = r / prime[i];
            for (int j = a; j <= b; j++)
                if (j > 1)
                    f[j * prime[i] - l] = true;
        }
        int mx = -1, mn = 0x3f3f3f3f, x1 = 0, x2 = 0, y1 = 0, y2 = 0;
        for (int i = 0, p = -1; i <= r - l; i++)
            if (!f[i])
            {
                if (~p)
                {
                    if (mx < i - p)
                        mx = i - p, x1 = p + l, y1 = i + l;
                    if (mn > i - p)
                        mn = i - p, x2 = p + l, y2 = i + l;
                }
                p = i;
            }
        if (mx == -1)
            puts("There are no adjacent primes.");
        else
            printf("%d,%d are closest, %d,%d are most distant.\n", x2, y2, x1, y1);
    }
    return 0;
}
```

POJ3421 - X-factor Chains 质因子的排列组合

X-factor Chains

```
#include <cstdio>
int f[1 << 12 | 1];
long long _f(int x)
{
    long long ans = 1;
    for (int i = x; i; i--) ans *= i;
    return ans;
}
int main()
{
    int x;
    while (~scanf("%d", &x))
    {
        int _x = x, t = 0;
        for (int i = 2; i * i <= _x; i++)
        {
            f[t] = 0;
            while (_x % i == 0)
                _x /= i, f[t]++;
            t++;
        }
        if (_x != 1) f[t++] = 1;
        int sum = 0;
        for (int i = 0; i < t; i++) sum += f[i];
        long long fac = _f(sum);
        for (int i = 0; i < t; i++) fac /= _f(f[i]);
        printf("%d %lld\n", sum, fac);
    }
    return 0;
}
```

POJ3090 - Visible Lattice Points 欧拉函数

Visible Lattice Points

```
#include <cstdio>
const int maxn = 1010;
int phi[maxn], sum[maxn];
int main()
{
    phi[1] = 1;
    for (int i = 2; i <= 1005; i++) if (!phi[i])
        for (int j = i; j <= 1005; j += i)
        {
            if (!phi[j]) phi[j] = j;
            phi[j] = phi[j] / i * (i - 1);
        }
    for (int i = 1; i <= 1005; i++) sum[i] = sum[i - 1] + phi[i];
    int T, x;
```

```
    scanf("%d", &T);
    for (int i = 1; i <= T; i++)
    {
        scanf("%d", &x);
        printf("%d %d %d\n", i, x, sum[x] << 1 | 1);
    }
    return 0;
}
```

3.3 欧几里得算法

最大公约数与最小公倍数

定义 1 设 a 和 b 是两个整数，如果 $d|a$ 且 $d|b$，则称 d 是 a 与 b 的公因子

定义 2 设 a 和 b 是两个不全为 0 的整数，称 a 与 b 的公因子中最大的为 a 与 b 的最大公因子，或最大公约数，记作 $gcd(a,b)$

定义 3 设 a 和 b 是两个非零整数，称 a 与 b 最小的正公倍数为 a 与 b 的最小公倍数，记作 $lcm(a,b)$

最大公约数与最小公倍数的性质

1. $a|m \wedge b|m \Rightarrow lcm(a,b)|m$
2. $d|a \wedge d|b \Rightarrow d|gcd(a,b)$
3. $lcm(a,b) = \frac{ab}{gcd(a,b)}$
4. $m,a,b \in \mathbb{N}_+ \rightarrow lcm(ma,mb) = m \times lcm(a,b)\ ,\ \ gcd(ma,mb) = m \times gcd(a,b)$

计算方法

素因子分解法 令

$$a = \prod_{i=1}^{m} p_i^{r_i}\ , \quad b = \prod_{i=1}^{m} p_i^{s_i}$$

于是

$$gcd(a,b) = \prod_{i=1}^{m} p_i^{min(r_i,s_i)}\ , \quad lcm(a,b) = \prod_{i=1}^{m} p_i^{max(r_i,s_i)}$$

欧几里得算法 1

$$gcd(a,b) = gcd(b, a \bmod b)$$

欧几里得算法 2

$$gcd(a,b) = gcd(a, a-b)$$

拓展欧几里得算法 不理解就记下来

exgcd

```
void exgcd(int64 a, int64 b, int64 &x, int64 &y)
{ b == 0 ? (x = 1, y = 0) : (exgcd(b, a % b, y, x), y -= x * (a / b)); }
```

3.4 线性同余方程

二元一次不定方程

定义 1 $a,b,c \in \mathbb{Z}, a \neq 0, b \neq 0$，那形如 $ax+by=c$ 的方程称为二元一次不定方程。

定理 1 设 $a,b \in \mathbb{Z}$ 且 $d=gcd(a,b)$，如果 $d|c$，那么方程存在无穷多个整数解，否则方程不存在整数解。

定理 2 如果不定方程有解且特解为 $x=x_0,y=y_0$ 那么方程的解可以表示为

$$x=x_0+\frac{b}{d}t, y=y_0-\frac{a}{d}t, \text{其中} t \in \mathbb{Z}$$

同余方程与不定方程 在 $a>0$ 且 $b>0$ 的条件下，求二元一次方程 $ax+by=c$ 的整数解等价于求一元线性同余方程 $ax \equiv c \bmod b$ 的整数解

求一元线性同余方程 要求 $ax \equiv c \bmod b$，即为求 $ax+my=b$ 的解。记 $d=gcd(a,m)$，先使用拓展欧几里得求 $ax+my=b$，如果 $d \nmid b$ 则无解，否则 $\bmod m$ 意义下的解有 d 个，可以通过对其中某个解不断地加 $\frac{m}{d}$ 得到。(这 d 个解的形式为 $x_0+\frac{m}{d}t,\ t \in \mathbb{Z}$, 其中 x_0 是已知的一个解)

中国剩余定理 若 $m_1,m_2,m_3,\ldots,m_r$ 是两两互素的正整数，则同余方程组

$$\begin{cases} x \equiv a_1 \pmod{m_1} \\ x \equiv a_2 \pmod{m_2} \\ x \equiv a_3 \pmod{m_3} \\ \cdots \\ x \equiv a_r \pmod{m_r} \end{cases} \tag{3.1}$$

有 $\bmod M=\prod_1^r m$ 的唯一解，即为中国剩余定理。

若 $n=pq$ 且 $gcd(p,q)=1$，那么 $x \bmod p, x \bmod q$ 的值确定后，$x \bmod n$ 的值也会随之确定。

算法说明

$$M_i=\prod_{j \neq i} m_j \rightarrow gcd(M_i,m_i)=1 \Rightarrow \exists p_i,q_i,\ M_ip_i+m_iq_i=1$$

$$(e_i=M_ip_i) \equiv int(j==i) \pmod{m_i} \rightarrow \sum_1^r e_ia_i \bmod \sum_1^r m_i \text{是方程的最小非负整数解}$$

练习题

POJ1061 - 蛤蛤的约会 +1s

蛤蛤的约会

```
#include <cstdio>
typedef long long int64;
int64 gcd(int64 a, int64 b) { return b == 0 ? a : gcd(b, a % b); }
void exgcd(int64 a, int64 b, int64 &x, int64 &y) { b == 0 ? (x = 1, y = 0) : (exgcd(b, a
    % b, y, x), y -= x * (a / b)); }
```

```
int main()
{
    int64 s, t, p, q, L;
    scanf("%lld%lld%lld%lld%lld", &s, &t, &p, &q, &L);
    int64 a = (p − q + L) % L, b = L, c = (t − s + L) % L;
    int64 x = 0, y = 0, g = gcd(a, b);
    if (c % g)
        puts("Impossible");
    else
    {
        a /= g, b /= g, c /= g;
        exgcd(a, b, x, y);
        printf("%lld", (((x % b + b) % b) * c) % b);
    }
    return 0;
}
```

POJ2142 - The Balance 求 $ax+by=c$ 的一组解，使得 $|x|+|y|$ 尽量小，在前者尽量小时 $|ax|+|by|$ 尽量小

The Balance

```
#include <cstdio>
inline int abs(int x) { return x >= 0 ? x : −x; }
void exgcd(int a, int b, int &d, int &x, int &y) { !b ? (x = 1, y = 0, d = a) : (exgcd(b,
     a % b, d, y, x), y −= x * (a / b)); }
int main()
{
    int a, b, c, x, y, g, u1, v1, u2, v2;
    while (~scanf("%d%d%d", &a, &b, &c) && a + b + c)
    {
        exgcd(a, b, g, x, y);
        a /= g, b /= g, c /= g;
        u1 = (x % b * c % b + b) % b;
        v1 = abs((c − u1 * a) / b);
        v2 = (y % a * c % a + a) % a;
        u2 = abs((c − v2 * b) / a);
        if (u1 + v1 > u2 + v2 || (u1 + v1 == u2 + v2 && a * u1 + b * v1 > a * u2 + b * v2
            )) u1 = u2, v1 = v2;
        printf("%d %d\n", u1, v1);
    }
    return 0;
}
```

3.5 逆元

解一元线性同余方程

$$gcd(a,m)=1 \Rightarrow \exists x, ax \equiv 1 \pmod{m}$$

$$ax \equiv 1 \pmod{m} \iff \exists k \in \mathbb{Z},\ ax - km = 1$$

```
int inv(int a, int m)
{
    int x, y;
    exgcd(a, m, x, y);
    return (x % m + m) % m;
}
```

费马小定理

$$\forall p \in \mathbb{P} \to x^p \equiv x \pmod{p}$$

被称为费马小定理，若 $p \nmid x$，有

$$x^{p-1} \equiv 1 \pmod{p}$$

于是有

$$\forall p \in \mathbb{P}, x \in \mathbb{Z} \to x^{-1} \equiv x^{p-2} \pmod{p}$$

使用快速幂即可计算。

3.6 离散对数问题

解这鬼东西：

$$A^x \equiv B \pmod{C}$$

这玩意有个性质，$A^x \mod C$ 有周期性，最大周期不超过 C，我想这是显然的（除非你没上过小学）。

$C \in \mathbb{P}$ 普通的 BSGS

POJ2417 - Discrete Logging 模板题

Primitive Roots

```
#include <cmath>
#include <cstdio>
#include <cstring>
typedef long long i64;
void exgcd(i64 a, i64 b, i64 &x, i64 &y)
{
    b == 0 ? (x = 1, y = 0) : (exgcd(b, a % b, y, x), y -= (a / b) * x);
}
struct HashTable
{
    static const size_t sz = 500009;
    i64 idx[sz], val[sz];
    void init()
    {
        memset(idx, -1, sizeof(idx));
        memset(val, -1, sizeof(val));
    }
    void insert(i64 i, i64 v)
    {
        i64 j = i % sz;
```

```
        while (idx[j] != -1 && idx[j] != i)
        {
            j++;
            if (j == sz)
                j = 0;
        }
        if (val[j] == -1)
        {
            idx[j] = i;
            val[j] = v;
        }
    }
    i64 find(i64 i)
    {
        i64 j = i % sz;
        while (idx[j] != -1 && idx[j] != i)
        {
            j++;
            if (j == sz)
                j = 0;
        }
        return val[j];
    }
} H;
int main()
{
    for (i64 a, b, c; ~scanf("%lld%lld%lld", &c, &a, &b) && a | b | c;)
    {
        H.init();
        i64 m = i64(ceil(sqrt(c))), d = 1;
        for (int i = 0; i < m; i++, d = d * a % c)
            H.insert(d, i);
        i64 res = 1, x, y;
        bool flag = false;
        for (i64 i = 0; i < m && !flag; i++, res = res * d % c)
        {
            exgcd(res, c, x, y);
            x = (x * b % c + c) % c;
            i64 j = H.find(x);
            if (j != -1)
                printf("%lld\n", i * m + j), flag = true;
        }
        if (!flag)
            puts("no solution");
    }
    return 0;
}
```

$C \in \mathbb{N}_+$ 待续

3.7 原根

阶

$$n > 1, a \in \mathbb{Z}, gcd(a, n) = 1 \rightarrow \exists r \in [1, n], a^r \equiv 1 \pmod{n}$$

r 的最小整数值称为 a 模 n 的阶，记为 $Ord_n(a)$

阶的性质

$$gcd(a, n) = 1, r = Ord_n(a), \forall N \in \{x | a^N \equiv 1 \pmod{n}\} \Rightarrow r | N$$

$$gcd(a, n) = 1 \Rightarrow Ord_n(a) | \varphi(n)$$

特别的，$p \in \mathbb{P}, gcd(a, p) = 1 \Rightarrow Ord_p(a) | p - 1$，显然$\varphi(p) = p - 1$

原根 $n \in \mathbb{N}_+, a \in \mathbb{Z}, Ord_n(a) = \varphi(n)$，则称 a 为模 n 的一个原根，由阶的定义可知原根和 n 必然互质。

求质数 p 的原根算法 暴力从小到大枚举 $g \in \mathbb{N}_+$，$\forall a \in \{x \mid x | p - 1, x \in \mathbb{P}\}, g^{\frac{p-1}{a}} \not\equiv 1 \pmod{p}$

Primitive Root

```cpp
#include <cstdio>
typedef unsigned long long u64;
const size_t MAXN = 1 << 16 | 1;
bool notprime[MAXN];
int prime[MAXN], pcnt, x;
inline u64 fast_pow(u64 a, u64 b, u64 m)
{
    u64 ret = 1;
    for (; b; a = a * a % m, b >>= 1)
        if (b & 1)
            ret = ret * a % m;
    return ret;
}
int main()
{
    for (size_t i = 2; i < MAXN; i++)
        if (!notprime[i] && (prime[pcnt++] = int(i)))
            for (size_t j = i * i; j < MAXN; j += i)
                notprime[j] = true;
    while (~scanf("%d", &x))
        for (int a = 2, flag = 0; flag == 0; a++)
        {
            flag = 1;
            for (int i = 0, k = x - 1; i < pcnt && k > 1 && flag; i++)
                if (k % prime[i] == 0)
                {
                    flag = fast_pow(a, (x - 1) / prime[i], x) != 1;
                    while (k % prime[i] == 0)
                        k /= prime[i];
                }
            if (flag)
                printf("%d\n", a);
        }
```

```
    return 0;
}
```

练习题

POJ1284 - Primitive Roots 如果 $n \in \mathbb{N}_+$ 有一个原根，那么 n 一共有 $\varphi(\varphi(n))$ 个不同余的原根

Primitive Roots

```
#include <cstdio>
const int N = 1 << 16 | 1;
int phi[N];
int main()
{
    for (int i = 2; i < N; i++) if (!phi[i])
        for (int j = i; j < N; j += i)
        {
            if (!phi[j]) phi[j] = j;
            phi[j] = phi[j] / i * (i - 1);
        }
    for (int p; ~scanf("%d", &p);)
        printf("%d\n", phi[p - 1]);
    return 0;
}
```

4 数据结构 I

4.1 树状数组

在 $\lg n$ 的时间内更新或查询 $\sum_{i=1}^{n} a_i$

lowbit `lowbit(x) = x & -x`

两个操作 $\lg n$ 更新或查询

Fenwick tree

```
void add(int x, int v)
{
    for (; x <= n; x += lowbit(x))
        A[x] += v;
}

void sum(int x)
{
    int res = 0;
    for (; x; x -= lowbit(x))
        res += A[x];
    return res;
}
```

练习题

POJ2352 - Stars 。。。

Stars

```
#include <cstdio>
#include <cstring>
#define lowbit(x) ((x) & -(x))
const int maxn = 1 << 15 | 1;
int a[maxn], n, m, level[maxn];
void update(int pos, int x)
{
    for (; pos < maxn; pos += lowbit(pos)) a[pos] += x;
}
int query(int pos)
{
    int ans = 0;
    for (; pos; pos -= lowbit(pos)) ans += a[pos];
    return ans;
}
int main()
{
    scanf("%d", &n);
    memset(level, 0, sizeof(level));
    memset(a, 0, sizeof(a));
    for (int i = 0, x, y; i < n; ++i)
    {
        scanf("%d%d", &x, &y);
        level[query(++x)]++;
        update(x, 1);
    }
    for (int i = 0; i < n; ++i)
        printf("%d\n", level[i]);
    return 0;
}
```

POJ2299 - Ultra-QuickSort 求逆序对数量，不过配图是啥玩意？马桶橛子？

Ultra-QuickSort

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <stdint.h>
#define lowbit(x) ((x) & -(x))
const int N = 500005;
int sum[N];
int query(int x)
{
    int ans = 0;
    for (; x; x -= lowbit(x)) ans += sum[x];
    return ans;
```

```
}
void update(int x, int y)
{
    for (; x <= N; x += lowbit(x)) sum[x] += y;
}
struct abcd
{
    int val, pos;
    bool operator<(const abcd &rhs) const { return val < rhs.val; }
} nodes[N];
int map[N];
int main()
{
    int n;
    while (~scanf("%d", &n) && n)
    {
        memset(sum, 0, sizeof(sum));
        for (int i = 1; i <= n; i++) scanf("%d", &nodes[i].val), nodes[i].pos = i;
        std::sort(nodes + 1, nodes + n + 1);
        for (int i = 1; i <= n; i++) map[nodes[i].pos] = i;
        int64_t ans = 0;
        for (int i = 1; i <= n; i++)
        {
            update(map[i], 1);
            ans += i - query(map[i]);
        }
        printf("%lld\n", ans);
    }
    return 0;
}
```

POJ1990 - MooFest 杀光奶牛问题就会得到解决

MooFest

```
#include <algorithm>
#include <cstdio>
#define lowbit(x) ((x) & -(x))
const int N = 20005;
void add(int *arr, int pos, int val)
{
    for (; pos < N; pos += lowbit(pos))
        arr[pos] += val;
}
int query(int *arr, int pos)
{
    int ans = 0;
    for (; pos; pos -= lowbit(pos))
        ans += arr[pos];
    return ans;
}
int sum[2][N];
struct cow
{
    int pos, vol;
    bool operator<(const cow &rhs) const { return vol < rhs.vol; }
} cows[N];
int main()
{
    int n;
    scanf("%d", &n);
    for (int i = 1; i <= n; i++)
        scanf("%d%d", &cows[i].vol, &cows[i].pos);
    std::sort(cows + 1, cows + n + 1);
    long long ans = 0;
    for (int i = 1; i <= n; i++)
    {
        long long a = query(sum[0], cows[i].pos), b = query(sum[1], cows[i].pos);
        ans += (cows[i].pos * a - b + query(sum[1], 20000) - b -
                (i - 1 - a) * cows[i].pos) *
               cows[i].vol;
        add(sum[0], cows[i].pos, 1);
        add(sum[1], cows[i].pos, cows[i].pos);
    }
    printf("%lld", ans);
    return 0;
}
```

4.2 Sparse Table

处理区间最值，即 RMQ（Range Minimum Query）问题。

预处理 计算一个数组 f，使 $f[i][j]=min[i,i+2^j)$，然后你就可以开始倍增了。

$$f[i][j]=\begin{cases} a[i] & j=0 \\ min(f[i][j-1],f[i+2^{j-1}][j-1]) & j>0 \end{cases} \tag{4.1}$$

询问 考虑一个询问区间 $[x,y)$ 的最小值的询问操作。

我们可以求出满足 $2^i \le y-x < 2^{i+1}$ 的 i，即 $\lfloor \log_2 y-x \rfloor$，这样我们可以用两个长度为 2^i 的小区间覆盖询问的大区间。

而长度为 2^i 的小区间的最小值在预处理时已经求出，于是区间 $[x,y)$ 的最小值为 $min\{f[x][i],f[y-2^i][i]\}$，由于是求最值，区间有交集也没关系。

练习题

POJ3264 - Balanced Lineup 给你一个长度为 n 的序列 $a[N]$，询问 Q 次，每次输出 $[L,R]$ 区间最大值与最小值的差是多少，果真这种简化了的题面真是清晰易懂。

Balanced Lineup

```
#include <cstdio>
inline int min(int a, int b) { return a < b ? a : b; }
inline int max(int a, int b) { return a > b ? a : b; }
const int N = 50010;
const int LogN = 16;
int minT[LogN][N], maxT[LogN][N], Log[N];
int main()
{
    Log[0] = -1;
    for (int i = 1; i < N; i++) Log[i] = Log[i >> 1] + 1;
    int m, n;
    scanf("%d%d", &n, &m);
    for (int i = 1, x; i <= n; i++)
    {
        scanf("%d", &x);
        minT[0][i] = maxT[0][i] = x;
    }
    for (int j = 1; j < LogN; j++)
        for (int i = 1; i + (1 << j) - 1 <= n; i++)
            minT[j][i] = min(minT[j - 1][i], minT[j - 1][i + (1 << (j - 1))]),
            maxT[j][i] = max(maxT[j - 1][i], maxT[j - 1][i + (1 << (j - 1))]);
    for (int i = 1, x, y, z; i <= m; i++)
    {
        scanf("%d%d", &x, &y);
        z = Log[y - x + 1];
        printf("%d\n", max(maxT[z][x], maxT[z][y - (1 << z) + 1]) - min(minT[z][x], minT[
            z][y - (1 << z) + 1]));
    }
    return 0;
}
```

CF359D - Pair of Numbers 首先要知道 gcd 有“最值的性质”，然后二分暴力。

Pair of Numbers

```
#include <cstdio>
#include <cctype>
inline int min(int a, int b) { return a < b ? a : b; }
inline int gcd(int a, int b) { return b == 0 ? a : gcd(b, a%b); }
inline void read(int &x)
{
    int ch = x = 0;
    while (!isdigit(ch = getchar()));
    for (; isdigit(ch); ch = getchar()) x = (x << 3) + (x << 1) + ch - '0';
}
int n, a[1 << 20 | 1], Log[1 << 20 | 1];
int M[1 << 20 | 1][20], G[1 << 20 | 1][20];
struct
{
    int c, a[1 << 20 | 1], r;
}ans;
inline bool can(int l, int r)
{
    int k = Log[r - l + 1];
    return min(M[l][k], M[r - (1 << k) + 1][k]) == gcd(G[l][k], G[r - (1 << k) + 1][k]);
}
bool check(int len)
{
    int cnt = 0;
    for (int i = 0; i + len < n; i++)
        if (can(i, i + len))
            ans.a[cnt++] = i;
    if (cnt) ans.r = len, ans.c = cnt;
    return cnt;
}
int main()
{
    read(n);
    Log[0] = -1;
    for (int i = 1; i <= n; i++) Log[i] = Log[i >> 1] + 1;
    for (int i = 0; i < n; i++) read(a[i]);
    for (int i = 0; i < n; i++) M[i][0] = G[i][0] = a[i];
    for (int j = 1; j <= Log[n]; j++)
        for (int i = 0; i + (1 << j) - 1 < n; i++)
            M[i][j] = min(M[i][j - 1], M[i + (1 << (j - 1))][j - 1]),
            G[i][j] = gcd(G[i][j - 1], G[i + (1 << (j - 1))][j - 1]);
    int L = 0, R = n, m;
    while (L < R)
        check(m = (L + R) >> 1) ? L = m + 1 : R = m;
    printf("%d %d\n", ans.c, ans.r);
    for (int i = 0; i < ans.c; i++)
        printf("%d ", ans.a[i] + 1);
    return 0;
}
```

4.3 左偏树

左偏树是一种可以合并的堆，所以它可以支持堆的几种操作，并且也都是在相同的时间复杂度下完成的，并且它能额外支持合并两棵左偏树的这种操作。

性质 我们先定义节点 i 的距离为节点 i 到它的后代中，最近的外节点所经过的边数，其中左子树或右子树为空的节点被称为外节点，这种节点的距离为 0。

性质 1：堆有序性质 节点的值不小于其子节点的值（假设这是一个大根左偏树）。

性质 2：左偏性质 节点的左子节点的距离不小于右子节点的距离。

性质 3 节点的距离等于它的右子节点的距离加一。

左偏树的操作

合并 现在我们有两棵左偏树 A 和 B，欲将其合并。我们假设 A 的根节点大于 B 的根节点，不然交换一下就行了。然后把 A 的右子节点与树 B 合并作为 A 的新右子节点，检查 A 现在的左子节点和右子节点的距离，如果右子节点的距离大于左子节点，那么交换这两棵子树，然后更新 A 的距离，如此这般递归下去，因为只有 $\log_2 n$ 的深度，所以铁定不会爆栈，除非你的脸黑到一定境界了。

合并的简单实现

```
struct node
{
    int val, dis;
    node *ch[2];
};
#define dis(x) ((x) ? (x)->dis : -1)
node *merge(node *a, node *b)
{
    if (a && b) return (node*)(uintptr_t(a) | uintptr_t(b));
    if (a->val < b->val) swap(a, b);
    a->ch[1] = merge(a->ch[1], b);
    if (dis(a->ch[0]) < dis(a->ch[1])) swap(a->ch[0], a->ch[1]);
    a->dis = dis(a->ch[1]) + 1;
    return a;
}
```

插入 把新节点当作一棵只有一个节点的树与原树合并。

删除极值 合并根节点下的两个子节点。

删除指定元素 将指定节点的两个子树合并作为这个节点。然后维护其父节点的数值，如果父节点被改变，就继续向上维护，直到节点数据不改变。

练习题

BZOJ2809 - [Apio2012]dispatching 考虑对每个点维护一个大根堆，堆内存储的是这个节点子树内所有点的 C_i。当堆内的和大于 M 时就要弹出堆顶元素；每个点的堆可以通过将所有儿子的堆合并起来再插入自己节点的 C_i 来得到

Apio2012 dispatching

```
#include <cctype>
#include <cstdio>
template <typename T> inline void swap(T &x, T &y) {
    T t = x;
    x = y;
    y = t;
}
template <typename T> inline T max(T x, T y) { return x > y ? x : y; }
inline void read(int &x)
{
    int ch = x = 0;
    while (!isdigit(ch = getchar()));
    for (; isdigit(ch); ch = getchar()) x = x * 10 + ch - '0';
}
const int maxn = 100010;
int n, m, fa[maxn], C[maxn], L[maxn], cnt;
typedef struct Node
{
    Node *lc, *rc;
    int val, dis, sz;
    long long sum;
} *lpNode;
lpNode merge(lpNode x, lpNode y)
{
    if (x == 0) return y;
    if (y == 0) return x;
    if (x->val < y->val) swap(x, y);
    x->rc = merge(x->rc, y);
    if (x->lc == 0 || x->lc->dis < x->rc->dis) swap(x->lc, x->rc);
    x->dis = x->rc ? x->rc->dis + 1 : 0;
    x->sz = (x->lc ? x->lc->sz : 0) + (x->rc ? x->rc->sz : 0) + 1;
    x->sum = (x->lc ? x->lc->sum : 0) + (x->rc ? x->rc->sum : 0) + x->val;
    return x;
}
Node mem[maxn];
lpNode nodes[maxn];
int head[maxn], to[maxn], next[maxn], ecnt, arr[maxn], aend;
inline void addEdge(int f, int t)
{
    ecnt++;
    next[ecnt] = head[f];
    head[f] = ecnt;
    to[ecnt] = t;
}
int main()
{
    read(n), read(m);
```

```
    for (int i = 1; i <= n; i++)
        read(fa[i]), read(C[i]), read(L[i]), addEdge(fa[i], i),
        (nodes[i] = mem + i)->sz = 1, nodes[i]->sum = nodes[i]->val = C[i];
    for (int i = 0; i <= n; i++)
        for (int cur = head[i]; cur; cur = next[cur])
            arr[++aend] = to[cur];
    long long ans = 0;
    for (int i = aend; i; i--)
    {
        int x = arr[i], y = fa[x];
        ans = max(ans, 1ll * nodes[x]->sz * L[x]);
        nodes[y] = merge(nodes[y], nodes[x]);
        while (y && nodes[y]->sum > m)
            nodes[y] = merge(nodes[y]->lc, nodes[y]->rc);
    }
    printf("%lld", ans);
    return 0;
}
```

4.4 线段树

既可以 $\log(n)$ 修改，也可以 $\log(n)$ 查询最值，也可以 $\log(n)$ 查询和，总之只要是能合并的数据都能在 $\log(n)$ 的时间内用其维护。

普通的线段树：单点修改，区间查询 直接搞

稍屌的线段树：区间修改，单点查询 普通的 Lazy-tag

通用的线段树：区间修改，区间查询 这东西包括以上两者，有两种实现方法，标记下传和标记永久化，其中标记永久化在可持久化数据结构中是必须的。现在给出一个大模板题，并且下面将会用两种方法加以实现。
这是题：POJ3468 - A Simple Problem with Integers

标记下传 这玩意儿是这么个搞法，还是照例的打标记，当我们要从某个节点递归下去时，将当前节点的 add 值下传，更新两个子节点的 add 与 sum 值，并将当前节点的 add 值清零。

标记下传例程：2.3s

```
#include <cstdio>
#include <cctype>
#define C(x) (x = getchar())
#define L(x) ((x) << 1)
#define R(x) ((x) << 1 | 1)
#define avg(x, y) (((x) + (y)) >> 1)
inline void read(int &x)
{
    int ch = x = 0, flag = 1;
    while (!isdigit(C(ch))) if (ch == '-') flag = -1;
    for (; isdigit(ch); C(ch))
        x = x * 10 + ch - '0';
    x *= flag;
```

```
}
const int N = int(1e5 + 5);
typedef long long i64;
int a[N];
i64 A[N << 2], S[N << 2];
inline void add(int k, int l, int r, i64 v)
{
    A[k] += v;
    S[k] += (r - l) * v;
}
inline void pushdown(int k, int l, int r)
{
    if (A[k] == 0) return;
    int m = avg(l, r);
    add(L(k), l, m, A[k]);
    add(R(k), m, r, A[k]);
    A[k] = 0;
}
void build(int k, int l, int r)
{
    if (l == r - 1) return void(S[k] = a[l]);
    int m = avg(l, r);
    build(L(k), l, m);
    build(R(k), m, r);
    S[k] = S[L(k)] + S[R(k)];
}
i64 query(int k, int l, int r, int x, int y)
{
    if (x <= l && r <= y) return S[k];
    pushdown(k, l, r);
    int m = avg(l, r);
    i64 res = 0;
    if (x < m) res += query(L(k), l, m, x, y);
    if (y > m) res += query(R(k), m, r, x, y);
    return res;
}
void modify(int k, int l, int r, int x, int y, int v)
{
    if (x <= l && r <= y) return add(k, l, r, v);
    int m = avg(l, r);
    pushdown(k, l, r);
    if (x < m) modify(L(k), l, m, x, y, v);
    if (y > m) modify(R(k), m, r, x, y, v);
    S[k] = S[L(k)] + S[R(k)];
}
int main()
{
    int n, m;
    read(n), read(m);
    for (int i = 0; i < n; i++) read(a[i]);
    build(1, 0, n);
    while (m--)
```

```
    {
        int op = 0, x, y, z;
        while (op != 'Q' && op != 'C') C(op);
        if (op == 'Q')
        {
            read(x), read(y);
            printf("%lld\n", query(1, 0, n, x − 1, y));
        }
        else
        {
            read(x), read(y), read(z);
            modify(1, 0, n, x − 1, y, z);
        }
    }
    return 0;
}
```

标记永久化 这种情况下 add 标记将不会被下传，子节点的影响在修改时便已计算，递归时要累加祖先节点上的标记。

标记永久化例程：2.0s

```
#include <cctype>
#include <cstdio>
#define C(x) ((x) = getchar())
#define L(x) ((x) << 1)
#define R(x) ((x) << 1 | 1)
#define avg(x, y) (((x) + (y)) >> 1)
typedef long long i64;
inline int max(int a, int b) { return a > b ? a : b; }
inline int min(int a, int b) { return a < b ? a : b; }
inline void read(int &x)
{
    int ch = x = 0, sign = 1;
    while (!isdigit(C(ch))) if (ch == '−') sign = −1;
    for (; isdigit(ch); C(ch))
        x = x * 10 + ch − '0';
    x *= sign;
}
const int N = int(1e5 + 5);
i64 add[N << 2], sum[N << 2];
int a[N];
void build(int k, int l, int r)
{
    if (l == r − 1) return void(sum[k] = a[l]);
    int m = avg(l, r);
    build(L(k), l, m);
    build(R(k), m, r);
    sum[k] = sum[L(k)] + sum[R(k)];
}
void modify(int k, int l, int r, int x, int y, int v)
{
```

```
    if (x <= l && r <= y)
        return void(add[k] += v);
    sum[k] += 1ll * (min(r, y) - max(l, x)) * v;
    int m = avg(l, r);
    if (x < m) modify(L(k), l, m, x, y, v);
    if (y > m) modify(R(k), m, r, x, y, v);
}
i64 query(int k, int l, int r, int x, int y)
{
    if (x <= l && r <= y)
        return sum[k] + (r - l) * add[k];
    int m = avg(l, r);
    i64 res = add[k] * (min(r, y) - max(l, x));
    if (x < m) res += query(L(k), l, m, x, y);
    if (y > m) res += query(R(k), m, r, x, y);
    return res;
}
int main()
{
    int n, m;
    read(n), read(m);
    for (int i = 0; i < n; i++) read(a[i]);
    build(1, 0, n);
    while (m--)
    {
        int op = 0, x, y, z;
        while (op != 'C' && op != 'Q') C(op);
        if (op == 'Q')
        {
            read(x), read(y);
            printf("%lld\n", query(1, 0, n, x - 1, y));
        }
        else
        {
            read(x), read(y), read(z);
            modify(1, 0, n, x - 1, y, z);
        }
    }
    return  0;
}
```

经过测试，发现标记永久化的写法快一些，可能是因为没有那么多的 pushdown 导致的。

4.5 树链剖分

轻重链剖分,把重链连续的铺在一棵线段树上,然后就解决了!是不是很容易?233333333333333333333333

练习题

BZOJ1036 - [ZJOI2008] 树的统计 Count 模板题

ZJOI2008 树的统计 Count

```
#include <cstdio>
#include <cctype>
#define L(x) ((x) << 1)
#define R(x) ((x) << 1 | 1)
#define avg(x, y) (((x) + (y)) >> 1)
inline int max(int a, int b) { return a > b ? a : b; }
inline int min(int a, int b) { return a < b ? a : b; }
template <typename T> inline void swap(T &a, T &b)
{
    T t = a;
    a = b;
    b = t;
}
inline void read(int &x)
{
    int ch = x = 0, sign = 1;
    while (!isdigit(ch = getchar())) if (ch == '-') sign = -1;
    for (; isdigit(ch); ch = getchar()) x = x * 10 + ch - '0';
    x *= sign;
}
const int N = int(3e4 + 5), inf = 0x3f3f3f3f;
int adj[N], nxt[N << 1], to[N << 1], ecnt;
int sum[N << 2], mx[N << 2];
int n, sz[N], son[N], dep[N], fa[N], top[N], idx[N], pos[N], val[N];
inline void addEdge(int f, int t)
{
    ecnt++;
    nxt[ecnt] = adj[f];
    adj[f] = ecnt;
    to[ecnt] = t;
}
void build(int k, int l, int r)
{
    if (l == r - 1) return void(sum[k] = mx[k] = val[idx[l]]);
    int m = avg(l, r);
    build(L(k), l, m);
    build(R(k), m, r);
    sum[k] = sum[L(k)] + sum[R(k)];
    mx[k] = max(mx[L(k)], mx[R(k)]);
}
void modify(int k, int l, int r, int p, int v)
{
    if (l == r - 1) return void(sum[k] = mx[k] = v);
    int m = avg(l, r);
    (p < m) ? modify(L(k), l, m, p, v) : modify(R(k), m, r, p, v);
    sum[k] = sum[L(k)] + sum[R(k)];
    mx[k] = max(mx[L(k)], mx[R(k)]);
}
int qsum(int k, int l, int r, int x, int y)
{
    if (x <= l && r <= y) return sum[k];
```

```
    int m = avg(l, r), res = 0;
    if (x < m) res += qsum(L(k), l, m, x, y);
    if (y > m) res += qsum(R(k), m, r, x, y);
    return res;
}
int qmax(int k, int l, int r, int x, int y)
{
    if (x <= l && r <= y) return mx[k];
    int m = avg(l, r), res = -inf;
    if (x < m) res = max(res, qmax(L(k), l, m, x, y));
    if (y > m) res = max(res, qmax(R(k), m, r, x, y));
    return res;
}
int psum(int u, int v)
{
    int res = 0;
    while (top[u] != top[v])
    {
        if (dep[top[u]] < dep[top[v]]) swap(u, v);
        res += qsum(1, 0, n, pos[top[u]], pos[u] + 1);
        u = fa[top[u]];
    }
    if (dep[u] > dep[v]) swap(u, v);
    return res + qsum(1, 0, n, pos[u], pos[v] + 1);
}
int pmax(int u, int v)
{
    int res = -inf;
    while (top[u] != top[v])
    {
        if (dep[top[u]] < dep[top[v]]) swap(u, v);
        res = max(res, qmax(1, 0, n, pos[top[u]], pos[u] + 1));
        u = fa[top[u]];
    }
    if (dep[u] > dep[v]) swap(u, v);
    return max(res, qmax(1, 0, n, pos[u], pos[v] + 1));
}
void init()
{
    static int que[N];
    int len = 0, u, v;
    que[len++] = 1;
    for (int i = 0; i < len; i++)
    {
        sz[u = que[i]] = 1;
        for (int e = adj[u]; e; e = nxt[e])
            if ((v = to[e]) != fa[u])
                fa[v] = u, dep[v] = dep[u] + 1, que[len++] = v;
    }
    for (int i = len - 1; i; i--)
    {
        u = que[i], v = fa[u];
```

```
        sz[v] += sz[u];
        if (sz[u] > sz[son[v]]) son[v] = u;
    }
    for (int i = 0, tot = 0; i < len; i++)
        if (top[u = que[i]] == 0)
            for (v = u; v; v = son[v])
                idx[pos[v] = tot++] = v, top[v] = u;
}
int main()
{
    read(n);
    for (int i = 1, x, y; i < n; i++)
    {
        read(x), read(y);
        addEdge(x, y), addEdge(y, x);
    }
    for (int i = 1; i <= n; i++) read(val[i]);
    init();
    build(1, 0, n);
    int m, u, v;
    read(m);
    static char op[10];
    while (m--)
    {
        scanf("%s", op);
        read(u), read(v);
        if (op[1] == 'H') modify(1, 0, n, pos[u], v);
        if (op[1] == 'M') printf("%d\n", pmax(u, v));
        if (op[1] == 'S') printf("%d\n", psum(u, v));
    }
    return 0;
}
```

5 计算几何

5.1 基础概念

点 $P(x,y)$

线段 用两个端点或一个端点加一条向量表示，这两种方式等价。其中线段 AB 上的点 C 满足：

$$\forall C \in AB, \exists p \in [0,1], C = pA + (1-p)B$$

直线 使用直线方程 $ax+by+c=0$ 或 $y=kx+b$ 或直线上两个不同点或一个点加上一个向量表示。

多边形 若干条线段首尾顺次相连所形成的平面图形

圆 使用圆心坐标和半径表示

向量 几何向量是线性空间中有大小与方向的量。

向量的表示 $a=\overrightarrow{OA}=(x,y),|a|=\sqrt{x^2+y^2}$

向量的计算 $a=(x_1,y_1),b=(x_2,y_2),a\pm b=(x_1+x_2\pm y_1+y_2),ak=(kx_1,ky_1)$

向量的夹角 $\cos<a,b>=\frac{ab}{|a||b|}$

正弦定理 对于任意三角形 ABC，a,b,c 分别为 $\angle A,\angle B,\angle C$ 的对边，R 为三角形 ABC 外接圆半径，则有

$$\frac{a}{\sin A}=\frac{b}{\sin B}=\frac{c}{\sin C}=2R$$

余弦定理 对于任意三角形 ABC，a,b,c 分别为 $\angle A,\angle B,\angle C$ 的对边，则有

$$\begin{cases}a^2=b^2+c^2-2bc\cos A\\ b^2=a^2+c^2-2ac\cos B\\ c^2=a^2+b^2-2ab\cos C\end{cases}$$

向量的点积 $a=(x_1,y_1),b=(x_2,y_2),ab=(x_1x_2,y_1y_2)=|a||b|\cos<a,b>$

向量的叉积

三维叉积：两个向量 $a=(x_1,y_1,z_1),b=(x_2,y_2,z_2)$ 的叉积的结果是一个向量 c，记作

$$c=a\times b=\begin{vmatrix}i & j & k\\ x_1 & y_1 & z_1\\ x_2 & y_2 & z_2\end{vmatrix}$$

c 的模长等于以 a,b 为两条邻边所作成的平行四边形的面积，c 的方向遵循右手定则。

二维叉积：若 $a=(x_1,y_1),b=(x_2,y_2)$，则我们可以将它们看做是 z 轴为 0 的两个三维向量，根据定义我们可知叉积结果为 $(0,0,x_1y_2-x_2y_1)$。因此我们定义二维叉积为 $a\times b=x_1y_2-x_2y_1$，它的几何意义是，叉积结果大小等于以 a,b 为两条邻边所作成的平行四边形的有向面积，即 $a\times b=|a|\times|b|\times\sin<a,b>$

根据叉积结果，我们能判断 a,b 的方向。当 $a\times b>0$ 时，b 在 a 的逆时针方向，当 $a\times b<0$ 时，b 在 a 的顺时针方向。

$a\times b=-b\times a$　　$a\times b=0\iff a,b$共线

向量的旋转 向量 $a=(x,y)$ 逆时针旋转 θ，得到的向量 b 的坐标 $b=(x\cos\theta-y\sin\theta,x\sin\theta+y\cos\theta)$。我们可以借助两个复数相乘，积的模等于两复数模的积，积的辐角等于两复数辐角的和这个性质，将旋转看做是两个复数相乘，即 $(x+yi)$ 与 $(\cos\theta+i\sin\theta)$ 相乘，再利用复数乘法式子 $(a+bi)\times(c+di)=(ac-bd)+(ad+bc)i$ 得出上面向量旋转的结果。

5.2 基础问题

1. $a=(x_1,y_1),b=(x_2,y_2)$ 两点距离：$\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$

2.

(1) 叉积： $2S_{\triangle}ABC=|\overrightarrow{AB}\times\overrightarrow{AC}|=|\overrightarrow{BA}\times\overrightarrow{BC}|=|\overrightarrow{CA}\times\overrightarrow{CB}|$

(2) 海伦公式： $S=\sqrt{p(p-a)(p-b)(p-c)},p=\frac{a+b+c}{2}$

3. 判断点 P 是否在线段 AB 上：判断是否 $|PA|+|PB|=|AB|$

4. 判断点 P 是否在直线 AB 上（三点共线）：判断是否 $\overrightarrow{PA}\times\overrightarrow{PB}=0$

5. 判断连续两条线段 AB,BC 的拐向：叉积判断，如 $\overrightarrow{AB}\times\overrightarrow{AC}>0 \iff$ 逆时针拐弯

6. 求 $\angle AOB$ 的种类：$s=\overrightarrow{OA}\cdot\overrightarrow{OB}, s=0 \iff$ 直角, $s>0 \iff$ 锐角, $s<0 \iff$ 钝角

7. 求点 P 到线段 AB 的最短距离：

(1) 若 $\angle BAP,\angle ABP$ 有一个是钝角，则最短距离为 $min(|PA|,|PB|)$

(2) 否则最短距离为 $\triangle PAB$ 底边 AB 的高：$\frac{|\overrightarrow{PA}\times\overrightarrow{PB}|}{|\overrightarrow{AB}|}$

8. 判断两线段 AB,CD 是否相交：判断 A,B 是否在 CD 两边，以及 C,D 是否在 AB 两边, 注意共线的特殊情况。若 AB,CD 不共线，则判断 $(\overrightarrow{CD}\times\overrightarrow{CA})\times(\overrightarrow{CD}\times\overrightarrow{CB})\le 0$ 且 $(\overrightarrow{AB}\times\overrightarrow{AC})\times(\overrightarrow{AB}\times\overrightarrow{AD})\le 0$；否则判断是否 $\exists P\in AB, P\in CD$。

9. 求两直线（线段）交点，假设交点存在：

(1) 代数方法： 求出两条直线的方程，列方程组求解

(2) 几何方法： $s_1=\overrightarrow{AC}\times\overrightarrow{AD}, s_2=\overrightarrow{BD}\times\overrightarrow{BC}, P=A+\frac{\overrightarrow{AB}\times s_1}{s_1+s_2}$

10. 求点 E 到直线 AB 的垂足 D

(1) 将 $\overrightarrow{AB}$ 旋转 $\frac{\pi}{2}$，直线求交

(2)

$$D=A+\overrightarrow{AB}\times\frac{\overrightarrow{AE}\times\overrightarrow{AB}}{\overrightarrow{AB}\times\overrightarrow{AB}}$$

11. 求 $\angle E$ 的角平分线：做两点 A,B 使得 $EA=EB$，则此时将 $\overrightarrow{AB}$ 旋转即可。

12. 求一个点 P 是否在多边形 Q 内部：射线法，过 P 做一条平行于 x 轴的射线 L，若与多边形有奇数个交点，则点在多边形内部，这可以转化为求 L 与 Q 的每一条边是否有交点，但要注意一些特殊情况，比如交点时多边形的顶点，这时我们需要规定交在边的下端点统计进答案或是交在边的上端点统计进答案（也就是保证一个点要么都被统计要么都不被统计）。

13. 多边形面积：利用叉积的几何意义（有向面积）求解。

$$S=\frac{1}{2}\left|\sum_{i=0}^{n-1} v_i\times v_{(i+1)\ mod\ n}\right|$$

14. 圆与直线求交点：求圆心到直线的距离 d，根据半径 R 以及 d 求 $\angle AOE$，然后再以 O 为中心顺时针、逆时针分别旋转 $\overrightarrow{OE}$，再利用缩放得出 $\overrightarrow{OA},\overrightarrow{OB}$，进而求出 A,B

15. 圆与圆求交点：已知大小半径 R, r，并且可以利用圆心坐标求出圆心距 d，利用圆心距与半径的关系可以先判断两圆关系，若有交点则我们可以利用余弦定理求出 $\angle BAP$，再以 A 为中心旋转、缩放 $\overrightarrow{AB}$ 得到 $\overrightarrow{AQ}, \overrightarrow{AP}$，进而求出 P, Q。

16. 求两圆公切线：考虑求出两个切点，分两种情况讨论：

(1) AC 长度为两圆半径差，$\angle ACB$ 是直角，进而可知 $\angle BAC$ 大小，将 $\overrightarrow{AB}$ 旋转至 $\overrightarrow{AC}$ 再进行缩放得到切点坐标。另一个切点同理可求。

(2) 由两圆半径比例可得 AO 长度，$\angle ACO$ 是直角且 AC 是半径，因此可求 $\angle OAC$，将 $\overrightarrow{AO}$ 旋转缩放得到 $\overrightarrow{AC}$，求出切点坐标。另一个切点同理可求。

练习题

ZOJ1081 - Points Within 以上第 12 点

Points Within

```
#include <cstdio>
#define C const
#define I inline
struct P
{
    int x, y;
    P() {}
    P(int _x, int _y) { x = _x, y = _y; }
} a[105];
I P operator+(C P &L, C P &R) { return P(L.x + R.x, L.y + R.y); }
I P operator-(C P &L, C P &R) { return P(L.x - R.x, L.y - R.y); }
I int cross(C P &L, C P &R) { return L.x * R.y - L.y * R.x; }
I int dot(C P &L, C P &R) { return L.x * R.x + L.y * R.y; }
I bool check(C P &p, C P &u, C P &v)
{
    return cross(u - p, v - p) == 0 && dot(u - p, v - p) <= 0;
}
int main()
{
    for (int n, m, T = 1; (scanf("%d", &n), n) && scanf("%d", &m); T++)
    {
        if (T > 1) putchar('\n');
        printf("Problem %d:\n", T);
        for (int i = 0; i < n; i++)
            scanf("%d%d", &a[i].x, &a[i].y);
        a[n] = a[0];
        while (m--)
        {
            P p;
            scanf("%d%d", &p.x, &p.y);
            bool flag = false;
            int cnt = 0;
            for (int i = 0; i < n; i++)
```

```
            {
                if (flag = check(p, a[i], a[i + 1])) break;
                int d1 = a[i].y - p.y, d2 = a[i + 1].y - p.y,
                    d = cross(a[i] - p, a[i + 1] - p);
                cnt += (d >= 0 && d1 < 0 && d2 >= 0) || (d <= 0 && d1 >= 0 && d2 < 0);
            }
            puts(flag || (cnt & 1) ? "Within" : "Outside");
        }
    }
    return 0;
}
```

5.3 凸包

平面上有 N 个点，有一个周长最小（面积最小）的凸多边形包含这 N 个点，这个凸多边形就叫做这个点集的凸包。

做法 常用 Graham 扫描法：

(1) 选出最左下角的点（x 坐标最小，如果相同就 y 坐标最小）作为极点，这个点显然在凸包上。

(2) 对其余点进行极角排序，相同时比较到极点的距离（使用叉积实现，免得调用 CRT 的数学库）

(3) 用一个栈来储存凸包上的点，先将极点和极角序最小的两个点入栈。

(4) 按序扫描每个点，检查栈顶的两个点和这个点“是否拐向右”

(5) 如果是的，弹出栈顶元素，回到第四步

(6) 该点入栈，继续循环这个过程。

(7) 没了

练习题

POJ3348 - Cows 模板题，注意公式要背对

Cows

```
#include <cstdio>
#include <algorithm>
const int N = int(1e4 + 5);
struct point
{
    int x, y;
    point() {}
    point(int _x, int _y) : x(_x), y(_y) {}
}a[N], stk[N];
inline point operator+(const point &L, const point &R)
{ return point(L.x + R.x, L.y + R.y); }
```

```
inline point operator−(const point &L, const point &R)
{ return point(L.x − R.x, L.y − R.y); }
inline int cross(const point &L, const point &R) { return L.x * R.y − L.y * R.x; }
inline int dot(const point &L, const point &R) { return L.x * R.x + L.y * R.y; }
inline int len2(const point &u, const point &v)
{
    point p = u − v;
    return p.x * p.x + p.y * p.y;
}
inline bool cmp1(const point &L, const point &R)
{ return L.x < R.x || (L.x == R.x && L.y < R.y); }
inline bool cmp2(const point &L, const point &R)
{
    int det = cross(L − a[0], R − a[0]);
    return det != 0 ? det > 0: len2(L, a[0]) < len2(R, a[0]);
}
int main()
{
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
        scanf("%d%d", &a[i].x, &a[i].y);
    std::swap(*std::min_element(a, a + n, cmp1), a[0]);
    std::sort(a + 1, a + n, cmp2);
    int top = 0;
    stk[top++] = a[0];
    for (int i = 1; i < n; i++)
    {
        while (top >= 2 && cross(a[i] − stk[top − 1], stk[top − 1] − stk[top − 2]) >= 0)
            top−−;
        stk[top++] = a[i];
    }
    stk[top] = stk[0];
    int res = 0;
    for (int i = 0; i < top; i++)
        res += cross(stk[i], stk[i + 1]);
    printf("%d", res / 100);
    return 0;
}
```

5.4 旋转卡壳

你知道吗？旋转卡壳有 16 种读法。

首先有个很直观的结论：最远点对的两点一定在凸包上。我们的思路就是枚举凸包上的所有边，对每一条边找出凸包上离该边最远的顶点，计算这个顶点到该边两个端点的距离，并记录最大的值。当我们逆时针枚举边的时候，最远点的变化也是逆时针的，这样就可以不用从头计算最远点，而可以紧接着上一次的最远点继续计算，于是我们得到了 $O(n)$ 的算法。

练习题

POJ2187 - Beauty Contest 模板题，注意符号要写对

Beauty Contest

```
#include <cstdio>
#include <algorithm>
inline int max(int a, int b) { return a > b ? a : b; }
const int N = 50005;
struct point
{
    int x, y;
    point() {}
    point(int _x, int _y) : x(_x), y(_y) {}
}a[N], stk[N];
#define pArg(x) const point &x
inline point operator+(pArg(l), pArg(r)) { return point(l.x + r.x, l.y + r.y); }
inline point operator-(pArg(l), pArg(r)) { return point(l.x - r.x, l.y - r.y); }
inline int cross(pArg(l), pArg(r)) { return l.x * r.y - l.y * r.x; }
inline int dot(pArg(l), pArg(r)) { return l.x * r.x + l.y * r.y; }
inline int len2(pArg(p)) { return p.x * p.x + p.y * p.y; }
inline bool cmp1(pArg(l), pArg(r)) { return l.x < r.x || (l.x == r.x && l.y < r.y); }
inline bool cmp2(pArg(l), pArg(r))
{
    int det = cross(l - a[0], r - a[0]);
    return det != 0 ? det > 0 : len2(l - a[0]) < len2(r - a[0]);
}
#define nxt(x) ((x) == n - 1 ? 0 : (x) + 1)
#define area(p, u, v) cross(u - p, v - p)
int main()
{
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
        scanf("%d%d", &a[i].x, &a[i].y);
    std::swap(a[0], *std::min_element(a, a + n, cmp1));
    std::sort(a + 1, a + n, cmp2);
    int top = 0;
    for (int i = 0; i < n; i++)
    {
        while (top >= 2 && cross(a[i] - stk[top - 1], stk[top - 1] - stk[top - 2]) >= 0)
            top--;
        stk[top++] = a[i];
    }
    stk[top] = stk[0];
    int res = 0;
    if (top == 2)
        res = len2(stk[1] - stk[0]);
    else for (int i = 0, j = 2; i < n; i++)
    {
        while (nxt(j) != i &&
               area(stk[j], stk[i], stk[i + 1]) <=
               area(stk[j + 1], stk[i], stk[i + 1]))
            j = nxt(j);
```

```
        res = max(res, max(len2(stk[j] - stk[i]), len2(stk[j] - stk[i + 1])));
    }
    printf("%d", res);
    return 0;
}
```

6 网络流

6.1 最大流/最小割

我不想抄讲义了，就通俗易懂的说一下。每次找一条从源点到汇点的可行流，增广，直到源点和汇点不联通。就这样。Dinic 和 ISAP 都是最短增广路算法，不同的就是维护增广路的方式。

Dinic 虽然我并不使用 Dinic，不过，我认为研究这种算法也是有其必要性的。比如可以加深对费用流的理解（滑稽）。

POJ1273 - Drainage Ditches (Dinic)

```
#include <cstdio>
#include <cstring>
inline int min(int a, int b) { return a < b ? a : b; }
const int inf = 0x3f3f3f3f;
int adj[205], nxt[405], to[405], cap[405], cur[205], dis[405], ecnt;
inline void addEdge_impl_(int f, int t, int c)
{
    nxt[ecnt] = adj[f];
    adj[f] = ecnt;
    to[ecnt] = t;
    cap[ecnt] = c;
    ecnt++;
}
inline void addEdge(int f, int t, int c)
{
    addEdge_impl_(f, t, c);
    addEdge_impl_(t, f, 0);
}
int n, m;
bool bfs()
{
    static int que[205];
    for (int i = 1; i <= n; i++)
        cur[i] = adj[i], dis[i] = inf;
    int len = dis[1] = 0;
    que[len++] = 1;
    for (int i = 0; i < len; i++)
        for (int e = adj[que[i]]; ~e; e = nxt[e])
            if (cap[e] && dis[to[e]] > dis[que[i]] + 1)
                dis[que[len++] = to[e]] = dis[que[i]] + 1;
    return dis[n] < inf;
}
int dinic(const int u, int rest)
```

```
{
    if (u == n) return rest;
    int flow = 0;
    for (int e = cur[u], curFlow; rest && ~e; e = nxt[e])
        if (cur[u] = e, cap[e] && dis[u] == dis[to[e]] - 1)
            if ((curFlow = dinic(to[e], min(cap[e], rest))))
                rest -= curFlow, flow += curFlow,
                cap[e] -= curFlow, cap[e ^ 1] += curFlow;
    if (rest) dis[u] = inf;
    return flow;
}
int main()
{
    while (~scanf("%d%d", &m, &n))
    {
        ecnt = 0;
        memset(adj, -1, sizeof(adj));
        for (int i = 0, x, y, z; i < m; i++)
            scanf("%d%d%d", &x, &y, &z), addEdge(x, y, z);
        int ans = 0;
        while (bfs()) ans += dinic(1, inf);
        printf("%d\n", ans);
    }
    return 0;
}
```

对这段代码简单的说两句。值得注意的有这么几点，一是`cur[u]`的使用，在单次增广时可以避免无谓的探查；二是 40，41 行在没有把以上配额用光之前没有返回可以造成多路增广的情况来增加效率；三是第 42，43 行的判断，如果所有子节点的可用流之和都不能满足当前点的配额，这个点就没有访问的必要了。我想大概就这么些。

ISAP 我经常使用 ISAP，因为据说这玩意更稳定，由于我的脸黑，我怕碰上一些恶心的数据卡掉 Dinic，并且目前没有一道普通网络流的题能卡掉 ISAP。以下还是上面那道题。

POJ1273 - Drainage Ditches (ISAP)

```
#include <cstdio>
#include <cctype>
#include <cstring>
inline int min(int a, int b) { return a < b ? a : b; }
inline void read(int &x)
{
    int ch = x = 0;
    while (!isdigit(ch = getchar()));
    for (; isdigit(ch); ch = getchar()) x = x * 10 + ch - '0';
}
const int N = 205, inf = 0x3f3f3f3f;
int adj[N], nxt[N << 1], to[N << 1], cap[N << 1], cur[N], cnt[N], dis[N], fa[N], ecnt;
inline void addEdge_impl_(int f, int t, int c)
{
    nxt[ecnt] = adj[f];
    adj[f] = ecnt;
    to[ecnt] = t;
```

```
    cap[ecnt] = c;
    ecnt++;
}
inline void addEdge(int f, int t, int c)
{
    addEdge_impl_(f, t, c);
    addEdge_impl_(t, f, 0);
}
int ISAP(int S, int T)
{
    int flow = 0;
    for (int i = 0; i < N; i++) dis[i] = N - 1;
    int len = 0, x;
    static int que[N];
    dis[que[len++] = T] = 0;
    for (int i = 0; i < len; i++)
        for (int e = adj[x = que[i]]; ~e; e = nxt[e])
            if (cap[e ^ 1] && dis[to[e]] > dis[x] + 1)
                dis[que[len++] = to[e]] = dis[x] + 1;
    memset(cnt, 0, sizeof(cnt));
    for (int i = 0; i < N; i++) cur[i] = adj[i], cnt[dis[i]]++;
    x = S;
    while (dis[S] < N - 1)
    {
        if (x == T)
        {
            int curFlow = inf;
            for (x = T; x != S; x = to[fa[x] ^ 1]) curFlow = min(curFlow, cap[fa[x]]);
            for (x = T; x != S; x = to[fa[x] ^ 1]) cap[fa[x]] -= curFlow, cap[fa[x] ^ 1]
                += curFlow;
            flow += curFlow, x = S;
        }
        bool needRetreat = true;
        for (int e = cur[x]; needRetreat && ~e; e = nxt[e])
            if (cur[x] = e, cap[e] && dis[x] == dis[to[e]] + 1)
                needRetreat = false, fa[x = to[e]] = e;
        if (needRetreat)
        {
            int nd = N - 2;
            for (int e = adj[x]; ~e; e = nxt[e])
                if (cap[e]) nd = min(nd, dis[to[e]]);
            if (--cnt[dis[x]] == 0) break;
            ++cnt[dis[x] = nd + 1];
            cur[x] = adj[x];
            if (x != S) x = to[fa[x] ^ 1];
        }
    }
    return flow;
}
int main()
{
    int n, m;
```

```
    while (~scanf("%d%d", &m, &n))
    {
        memset(adj, -1, sizeof(adj)), ecnt = 0;
        for (int i = 0, x, y, z; i < m; i++)
            scanf("%d%d%d", &x, &y, &z), addEdge(x, y, z);
        printf("%d\n", ISAP(1, n));
    }
}
```

重新写了一遍，1A，我很高兴。ISAP 比 Dinic 要注意的点真是多多了，注意第 58 行的 break，这可以大大加速，因为此时源点和汇点已经不会联通了，继续做下去无益。反正我觉得 ISAP 要是能一遍写出来的话是比 Dinic 好的。

6.2 最小费用最大流

每次找到一条最短的路增广，大体上与 Dinic 完全相同，不过总感觉很奇怪，包括没有了 cur，多出了 vis 等一些不同，并且不加就会爆炸。

POJ2135 - Farm Tour

```
#include <cstdio>
#include <cstring>
inline int min(int a, int b) { return a < b ? a : b; }
const int inf = 0x3f3f3f3f, N = 1005, M = 50005;
int adj[N], nxt[M], to[M], cap[M], cost[M], dis[N], ecnt, idx;
inline void addEdge_impl_(int f, int t, int c, int w)
{
    nxt[ecnt] = adj[f];
    adj[f] = ecnt;
    to[ecnt] = t;
    cap[ecnt] = c;
    cost[ecnt] = w;
    ecnt++;
}
inline void addEdge(int f, int t, int c, int w)
{
    addEdge_impl_(f, t, c, w);
    addEdge_impl_(t, f, 0, -w);
}
int T;
bool bfs(int S)
{
    memset(dis, inf, sizeof(dis)), idx++;
    static int que[N << 1];
    static bool inq[N];
    int len = 0;
    dis[que[len++] = S] = 0;
    for (int i = 0; i < len; inq[que[i++]] = false)
        for (int e = adj[que[i]]; ~e; e = nxt[e])
            if (cap[e] && dis[to[e]] > dis[que[i]] + cost[e])
            {
                dis[to[e]] = dis[que[i]] + cost[e];
                if (!inq[to[e]]) que[len++] = to[e];
```

```
            }
    return dis[T] < inf;
}
int dfs(int x, int rest)
{
    static int vis[N];
    if (x == T) return rest;
    vis[x] = idx;
    int flow = 0;
    for (int e = adj[x], curFlow; rest && ~e; e = nxt[e])
        if (vis[to[e]] != idx && cap[e] && dis[to[e]] == dis[x] + cost[e])
            if (bool(curFlow = dfs(to[e], min(cap[e], rest))))
                rest -= curFlow, flow += curFlow, cap[e] -= curFlow,
                cap[e ^ 1] += curFlow;
    return flow;
}
int main()
{
    memset(adj, -1, sizeof(adj));
    int n, m;
    scanf("%d%d", &n, &m);
    T = n + 1;
    for (int i = 0, x, y, z; i < m; i++)
        scanf("%d%d%d", &x, &y, &z), addEdge(x, y, 1, z), addEdge(y, x, 1, z);
    addEdge(0, 1, 2, 0);
    addEdge(n, T, 2, 0);
    int ans = 0;
    while (bfs(0)) ans += dis[T] * dfs(0, inf);
    printf("%d", ans);
    return 0;
}
```

后记 我多次尝试直接移植 dinic 的模板来解决最小费用最大流问题，不过最后都只是收获了一个个 RE，看来直接照抄 dinic 的代码是不行的，仍然要加上 vis 数组之类的东西来保证不会爆栈（虽然我觉得我的移植代码也不会爆栈），如果哪名巨佬在 Review 我的代码时发现了方法，请不吝告知，蒟蒻万分感谢。

7 XJB 算法合集 I

XJB 算法的时间复杂度均为 O(跑得过)，空间复杂度均为)(开的下)。

7.1 二分与三分

二分 玄学算法之一，什么都可以往上套，会使时间复杂度乘上 $\log n$。但是二分法仅适用于求解具有单调性的问题，所以做题前要大胆猜想，小（bu）心（yong）求证。

二分的写法

整数二分的写法 我被 STL 荼毒了

```
while (L < R) check(m = (L + R) >> 1) ? L = m + 1 : R = m;
```

注意答案的取值是 L 还是 R 还是其他什么的，这是一个大问题，因题而异，不可描述。

实数二分的写法

```
while (R - L > eps) check(m = (L + R) / 2) ? L = m : R = m;
```

二分法常见模型

二分答案 最小值最大（或是最大值最小）问题，这类双最值问题常常使用二分法求解，也就是确定答案后，配合贪心或 DP 等其他算法来检验这个答案是否合法，将最优化问题转变为判定性问题。例如：将长度为 n 的序列 a_i 分成最多 m 个连续段，求所有分法中每段和的最大值最小能是多少。

二分查找 具有单调性的布尔表达式求解分界点，比如在有序数列中求解数字 x 的排名。

二分导数代替三分 有时对于一些单峰函数，我们可以通过二分导函数的方法求解函数极值，这样使用时通常定义域为整数域比较方便，因为此时 dx 可以直接取整数 1。

二分的例题 我讨厌奶牛。

POJ2456 - Aggressive cows

```
#include <cstdio>
#include <algorithm>
int n, m, a[100005];
bool check(int d)
{
    int cow = 0, nxt = 0;
    for (int i = 0; i < n; i++)
        if (a[i] >= nxt)
            cow++, nxt = a[i] + d;
    return cow >= m;
}
int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 0; i < n; i++) scanf("%d", a + i);
    std::sort(a, a + n);
    int L = 0, R = 0x3f3f3f3f, mid;
    while (L < R)
        check(mid = (L + R) >> 1) ? L = mid + 1 : R = mid;
    printf("%d", L - 1);
    return 0;
}
```

你看，这次的答案就是`L - 1`。

三分 三分法适用于求解单峰函数的极值问题。二次函数就是一个典型的单峰函数。三分法与二分法一样，它会不断缩小答案所在的求解区间。二分法缩短区间利用的原理是函数的单调性，而三分法利用的则是函数的单峰性。

设当前求解的区间为 $[l,r]$，令 $m_1 = l + \frac{r-l}{3}, m_2 = r - \frac{r-l}{3}$，接着我们计算这两个点的函数值 $f(m_1), f(m_2)$ 之后我们将两点中函数值更优的那个点称为好点（若计算最大值，则 f 更大的那个点就为好点，计算最小

值同理），而函数值较差的那个点称为坏点。并且最优点与好点会在坏点同侧，即我们的求解区间由 $[l,r]$ 变为了 $[l,m_2]$ 或 $[m_1,r]$。因此根据这个结论我们可以不停缩短求解区间，直至可以得出近似解。

三分的写法 以求上凸单峰函数的最大值为例，三分的代码：

```
while (R - L > eps) f(m1 = L + (R - L) / 3) < f(m2 = R - (R - L) / 3) ? L = m1 : R = m2;
```

三分的例题

$$S=\pi rl+\pi r^2, h=\sqrt{l^2-r^2}, V=\frac{1}{3}\pi r^2 h$$

若确定了底面半径，则其他值都可以依次计算出来，同时根据上面几个有关圆锥的公式也可以看出，V 是关于 r 的一个单峰函数，因此三分底面半径求解

POJ3737 - UmBasketella

```cpp
#include <cmath>
#include <cstdio>
const double pi = acos(-1.0), eps = 1e-6;
double S;
inline double calc(double r)
{
    double l = (S - r * r) / r;
    double h = sqrt(l * l - r * r);
    return pi * r * r * h / 3;
}
int main()
{
    while (~scanf("%lf", &S))
    {
        S /= pi;
        double l = 0, r = sqrt(S), m1, m2, d;
        while ((d = r - l) > eps)
            calc(m1 = l + d / 3) < calc(m2 = r - d / 3) ? l = m1 : r = m2;
        l = (S - r * r) / r;
        double h = sqrt(l * l - r * r);
        double V = pi * r * r * h / 3;
        printf("%.2f\n%.2f\n%.2f\n", V, h, r);
    }
    return 0;
}
```

7.2 Hash

人品算法，没有什么意思，附一个 UOJ 模数：$998244353 = 7 * 17 * 2^{23} + 1$

7.3 二分图匹配

给定一个二分图 G，在 G 的一个子图 M 中，M 的边集 E 中的任意两条边都不依附于同一个顶点，则称 M 是一个匹配。在二分图 G 中所有的匹配 M 中，边数最多的匹配，称为二分图的最大匹配。

求解二分图最大匹配

使用网络流算法 实际上，可以将二分图最大匹配问题看成是最大流问题的一种特殊情况。首先：建立源点 s 和汇点 t，从 s 向 X 集合的所有顶点引一条边，容量为 1，从 Y 集合的所有顶点向 t 引一条边，容量为 1。然后将二分图的所有边看成是从 X_i 到 Y_j 的一条有向边，容量为 1。求最大匹配就是求 s 到 t 的最大流。

使用匈牙利算法： 利用所有边的容量都是 1 以及二分图的性质，我们还可以将二分图最大匹配算法更简单地实现：
假设 M 是二分图 G 的一个匹配，则称 M 中边所依附的顶点是已匹配的顶点，若 P 是图 G 中一条连通两个未匹配顶点的路径，并且不属于 M 的边和属于 M 的边（即待匹配的边和已匹配边）在 P 上交替出现，则称 P 为相对于 M 的一条增广路径。如果我们将 P 的路径看着边的集合，我们令 $M' = M \; xor \; P$，则 M' 是比 M 多一条边的匹配，即更大的匹配 M' 包含了或者属于 M，或者属于 P，但不同时属于 M 和 P 的边。当找不到增广路的时候，则这时候的 M 是最大匹配。

例题 把光束当作图的顶点，小行星当作连接对应光束的边。

POJ3041 - Asteroids

```cpp
#include <cstdio>
int adj[505], nxt[10005], to[10005], ecnt, mate[505], vis[505], idx, n, m;
inline void addEdge(int f, int t)
{
    nxt[++ecnt] = adj[f], adj[f] = ecnt, to[ecnt] = t;
}
bool hungry(int u)
{
    for (int e = adj[u]; e; e = nxt[e])
        if (!mate[to[e]])
            return mate[to[e]] = u, true;
    for (int e = adj[u]; e; e = nxt[e])
        if (vis[to[e]] != idx)
        {
            vis[to[e]] = idx;
            if (hungry(mate[to[e]])) return mate[to[e]] = u, true;
        }
    return false;
}
int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 0, x, y; i < m; i++)
        scanf("%d%d", &x, &y), addEdge(x, y);
    int ans = 0;
    for (int i = 1; i <= n; i++)
    {
        idx++;
        if (hungry(i)) ans++;
    }
    printf("%d", ans);
    return 0;
}
```

7.4 迭代加深搜索

通过搜索的办法来判断是否有不超过某个 x 的解。把 x 从 0 开始每次增加 1，那么首次找到解时的 x 便是最优解。这种搜索就被称为迭代加深搜索 (IDDFS)。如果加上状态 v 到目标状态的距离下界的估价函数 $h^*(v)$ 进行提前剪枝优化后的算法则称为 IDA*。但要保证 $h^*(v)$ 的值小于真实的值，否则会导致错误的剪枝。

例题 估价函数：还没有走到正确位置上的棋子的个数

BZOJ1085 - 骑士精神

```
#include <cstdio>
template <typename T>
inline void swap(T &a, T &b)
{
    T t = a;
    a = b;
    b = t;
}
const char des[5][5] = { {'1', '1', '1', '1', '1'},
                         {'0', '1', '1', '1', '1'},
                         {'0', '0', '*', '1', '1'},
                         {'0', '0', '0', '0', '1'},
                         {'0', '0', '0', '0', '0'} };
const int dx[] = { 1, 2, 2, 1, -1, -2, -2, -1 };
const int dy[] = { -2, -1, 1, 2, 2, 1, -1, -2 };
const int Move[8][2] = { {1, -2}, {2, -1}, {2, 1}, {1, 2}, {-1, 2}, {-2, 1}, {-2, -1},
    {-1, -2} };
char cur[5][5];
#define H()                                                 \
    ((cur[0][0] != des[0][0]) + (cur[0][1] != des[0][1]) + \
     (cur[0][2] != des[0][2]) + (cur[0][3] != des[0][3]) + \
     (cur[0][4] != des[0][4]) + (cur[1][0] != des[1][0]) + \
     (cur[1][1] != des[1][1]) + (cur[1][2] != des[1][2]) + \
     (cur[1][3] != des[1][3]) + (cur[1][4] != des[1][4]) + \
     (cur[2][0] != des[2][0]) + (cur[2][1] != des[2][1]) + \
     (cur[2][2] != des[2][2]) + (cur[2][3] != des[2][3]) + \
     (cur[2][4] != des[2][4]) + (cur[3][0] != des[3][0]) + \
     (cur[3][1] != des[3][1]) + (cur[3][2] != des[3][2]) + \
     (cur[3][3] != des[3][3]) + (cur[3][4] != des[3][4]) + \
     (cur[4][0] != des[4][0]) + (cur[4][1] != des[4][1]) + \
     (cur[4][2] != des[4][2]) + (cur[4][3] != des[4][3]) + \
     (cur[4][4] != des[4][4]) - 1)
inline bool can(int x, int y)
{
    return 0 <= x && x <= 4 && 0 <= y && y <= 4;
}
int DEP;
bool dfs(int x, int y, int step)
{
    if (step == DEP) return H() == -1;
    if (H() + step > DEP) return false;
    for (int i = 0, nx, ny; i < 8; i++)
```

```
        if (can(nx = x + dx[i], ny = y + dy[i]))
        {
            swap(cur[x][y], cur[nx][ny]);
            if (dfs(nx, ny, step + 1)) return true;
            swap(cur[x][y], cur[nx][ny]);
        }
    return false;
}
int main()
{
    int T;
    scanf("%d", &T);
    while (T--)
    {
        for (int i = 0; i < 5; i++)
            scanf("%s", cur[i]);
        int si = -1, sj = -1;
        for (int i = 0; i < 5; i++)
            for (int j = 0; j < 5; j++)
                if (cur[i][j] == '*')
                    si = i, sj = j;
        for (DEP = 0; DEP <= 15; DEP++)
            if (dfs(si, sj, 0))
                break;
        printf("%d\n", DEP > 15 ? -1 : DEP);
    }
    return 0;
}
```

7.5 折半搜索/MITM

从初始状态与目标状态同时出发进行搜索

有向图模型 考虑这样一个问题：给定一张有向图 G, 图中所有边的长度均为 1，现在求从图中的点 A 到点 B 之间有多少条长度为 L 的不同的路径。两条路径不同当且仅当某一步它们所走的边不相同。我们设图中点的最大出度为常数 D。

朴素的想法是我们从 A 开始 DFS 走 L 步，若最后停在了 B 点则我们可以记录进答案。这个做法的复杂度是 $O(D^L)$。

我们换个角度考虑，从点 A 到点 B 长度为 L 的路径，可以看做是点 A 到某点 u 长度为 $\frac{L}{2}$ 的路径加上点 u 到点 B 长度为 $\frac{L}{2}$ 的路径。

从点 A 正向 DFS$\frac{L}{2}$ 步，对每个点记 P_u 表示点 A 正向走到它的不同路径数。

从点 B 反向 DFS$\frac{L}{2}$ 步，对每个点记 Q_u 表示点 B 反向走到它的不同路径数。

根据加法原理与乘法原理以及上面的分析我们可知最后的答案为：$\sum_u P_u \cdot Q_u$

容易发现，这个做法的复杂度就降为了 $O(D^{\frac{L}{2}})$。

这就是常用折半搜索的第一种模型，即有向图模型，从这个模型的做法我们也能看出，折半搜索是一种双向搜索。

方程模型 考虑这样一个问题,给定一个整数的集合 S,求有多少有序六元组 (a,b,c,d,e,f) 满足 $a,b,c,d,e,f \in S, d \neq 0$ 且 $\frac{ab \pm c}{d} - e = f, |S| \leq 100$。

朴素的算法是 $O(|S|^6)$ 枚举，并按上述式子计算，判定是否符合条件。
我们将原式移项为 $ab+c=(e+f)\times d$，此时方程两边都有 3 个元素，因此我们考虑先 $O(|S|^3)$ 的时间枚举左边的三个元素，算出结果，再用 $O(|S|^3)$ 的时间枚举右边的三个元素，也算出对应的结果，那么我们现在的问题是给出两个多重数集，问有多少个数同时在两个多重数集中出现了。这个问题可以使用哈希表来解决。这里我们就简单地认为哈希表插入与查询的复杂度均为 $O(1)$。那么这个算法的总时空复杂度均为 $O(|S|^3)$。
这就是折半搜索第二种常用的模型，方程模型，其核心就是通过移项，将方程两边的变量数进行平衡，之后再暴力搜索每一边，所得到的的结果利用数据结构进行存储查询，或是利用其他的一些算法进行合并，从而优化复杂度。

算法总结 通常折半搜索能将时间复杂度从 $O(D^L)$ 变为 $O(D^{\frac{L}{2}}\times$ 合并复杂度$)$。尽管时间复杂度仍然是指数级别，但指数减小一半，在一些问题中就能使得我们在时限内出解。并且该算法实现简单，算法使用的搜索过程与朴素算法无异，合并时也只需要用到哈希表等简单的数据结构。

8 分治

把问题分割成更小的子问题递归求解，再处理不同子问题之间的部分，这种算法设计方法就是分治法。关于分治类算法的时间复杂度，算法导论上详细的介绍了专有结论：主定理。在此不再赘述。

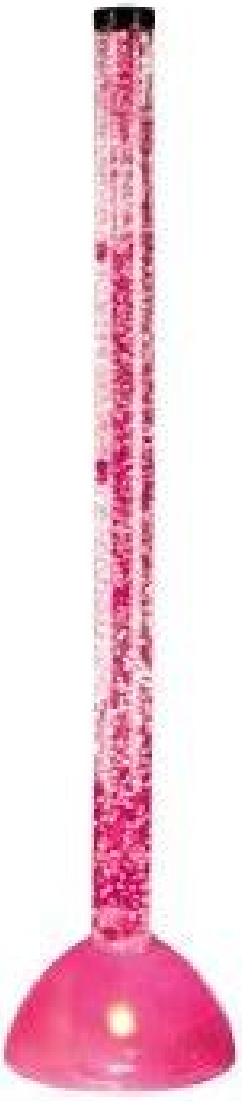
8.1 序列上的分治

数列的逆序对数除了使用数据结构外，也可以在归并排序的基础上统计额外信息来求得。
假设我们现在要统计数列 A 中逆序对的个数。如图所示，我们可以将数列 A 平均分成两半得到数列 B 和数列 C，于是，数列 A 中的逆序对 (i,j) 必然是下面三种之一：
(1)i,j 都属于数列 B 的逆序对 (i,j)
(2)i,j 都属于数列 C 的逆序对 (i,j)
(3)i 属于数列 B 而 j 属于数列 C 的逆序对 (i,j)
对于 (1) 和 (2)，它们的总数即为数列 B 与数列 C 的逆序对总数，这是一个子问题，我们递归就可以解决。而对于 (3)，我们可以对数列 C 中的每个数字 x，统计在数列 B 中比 x 大的数字的个数，再把所有的这样的个数加起来，得到的结果就是 (3) 的逆序对数。

POJ2299 - Ultra-QuickSort

```
#include <cstdio>
#include <cstring>
typedef long long i64;
const int N = int(5e5 + 5);
int n, a[N];
i64 merge(int l, int r)
{
    if (l == r - 1) return 0;
    int m = (l + r) >> 1, bn = m - l, cn = r - m;
    i64 cnt = merge(l, m) + merge(m, r);
    static int b[N], c[N];
    memcpy(b, a + l, bn << 2), memcpy(c, a + m, cn << 2);
    for (int i = l, bi = 0, ci = 0; i < r; i++)
        if (bi < bn && (ci == cn || b[bi] <= c[ci]))
            a[i] = b[bi++];
        else
```

```
            a[i] = c[ci++], cnt += bn - bi;
    return cnt;
}
int main()
{
    while (scanf("%d", &n), n)
    {
        for (int i = 0; i < n; i++) scanf("%d", a + i);
        printf("%lld\n", merge(0, n));
    }
    return 0;
}
```



8.2 平面上的分治

给定平面上的 n 个点，求距离最近的两个点的距离。假设我们把所有点按 x 坐标平均分成了左右两个部分，那么最近点对 p,q 的距离就是下面二者的最小值：

(1) 两点 p,q 同属于左半边或者右半边时的最近点对距离

(2) 两点 p,q 属于不同区域时的最近点对距离

递归计算，完成。

HDU1007 - Quoit Design

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstring>
template <typename T>
inline T min(T a, T b) { return a < b ? a : b; }
const int N = int(1e5 + 3);
struct Point
{
    double x, y;
} p[N];
double len(const Point &P) { return sqrt(P.x * P.x + P.y * P.y); }
inline bool operator<(const Point &l, const Point &r) { return l.x < r.x; }
inline Point operator-(const Point &l, const Point &r)
{
    Point ret;
    ret.x = l.x - r.x;
    ret.y = l.y - r.y;
    return ret;
}
double solve(int l, int r)
{
    if (l == r - 1) return 1e20;
    int m = (l + r) >> 1;
    double ans = min(solve(l, m), solve(m, r));
    double x0 = (p[m - 1].x + p[m].x) / 2;
    static Point a[N], b[N], c[N];
    int bn = 0, cn = 0;
    for (int i = l, li = l, ri = m; i < r; i++)
        if (li < m && (ri == r || p[li].y < p[ri].y))
        {
            a[i] = p[li++];
            if (x0 - a[i].x < ans) b[bn++] = a[i];
        }
        else
        {
            a[i] = p[ri++];
            if (a[i].x - x0 < ans) c[cn++] = a[i];
        }
    memcpy(p + l, a + l, (r - l) << 4);
    for (int i = 0, j = 0, k; i < bn || j < cn;)
        if (i < bn && (j == cn || b[i].y < c[j].y))
            for (k = j - 1; (k >= 0 && b[i].y - ans < c[k].y) || i++ > INT_MAX; k--)
```

```
                ans = min(ans, len(c[k] - b[i]));
        else
            for (k = i - 1; (k >= 0 && c[j].y - ans < b[k].y) || j++ > INT_MAX; k--)
                ans = min(ans, len(b[k] - c[j]));
    return ans;
}
int main()
{
    int n;
    while (scanf("%d", &n), n)
    {
        for (int i = 0; i < n; i++) scanf("%lf%lf", &p[i].x, &p[i].y);
        std::sort(p, p + n);
        printf("%.2f\n", solve(0, n) / 2);
    }
    return 0;
}
```

8.3 树分治

暂时跳过

8.4 cdq 分治

暂时跳过

9 数据结构 II

9.1 平衡树

9.1.1 前导知识

二叉搜索树 假定我们都知道什么是二叉搜索树。

二叉搜索树的旋转 假定我们都知道如何左旋和右旋。

算法时间复杂度分析初步 没有必要的知识。

9.1.2 Splay

简介 一种通过 Splay 操作使**均摊**时间复杂度为 $\log n$ 的二叉搜索树。

Splay 操作 将一个点移动到它上方的指定位置。

节点 x 的父节点 y 是根节点 将 x 通过左旋或右旋转到根。结束。

节点 x 的父节点 y 不是根节点，y 的父节点为 z，且 x 与 y 同时是各自父节点的左孩子或者同时是各自父节点的右孩子 先将 y 转上去，再将 x 转上去。

节点 x 的父节点 y 不是根节点，y 的父节点为 z，且 x 与 y 一个是其父节点的左孩子而另一个是其父节点的右孩子 把 x 转上去两次。

示例代码 模板题

BZOJ3224 - 普通平衡树

```
#include <cstdio>
#include <cstdlib>
#define sz(x) (x == 0 ? 0 : x->size)
#define update(x) \
    if (x) x->size = sz(x->child[0]) + sz(x->child[1]) + 1
#define getVal(x) (x == 0 ? 0 : x->val)
typedef int valType;
const int arrSize = 100005;
struct Node
{
    valType val;
    int size;
    Node *child[2], *father;
    Node *Init(int x, Node *f)
    {
        val = x, size = 1, father = f, child[0] = child[1] = 0;
        return this;
    }
};
typedef Node *lpNode;
template <typename _ty, typename _typ = _ty *>
struct allocator
{
private:
    int i;
    _ty *data;
    _typ *pt;

public:
    allocator(int size)
    {
        data = new _ty[size];
        pt = new _typ[size];
        this->i = 0;
        for (int j = 0; j < size; j++) pt[j] = &data[j];
    }
    _typ alloc()
    {
        return pt[i++];
    }
    void free(_typ ptr)
    {
        pt[--i] = ptr;
    }
};
struct SplayTree
```

```
{
private:
    lpNode root = 0;
    allocator<Node> *mem;
    void rotate(lpNode x)
    {
        if (x == 0 || x->father == 0) return;
        int d = (x == x->father->child[0]);
        lpNode y = x->father;
        y->child[d ^ 1] = x->child[d];
        if (x->child[d]) x->child[d]->father = y;
        x->father = y->father;
        if (y->father) y->father->child[y == y->father->child[1]] = x;
        y->father = x;
        x->child[d] = y;
        update(y);
        update(x);
    }
    void Splay(lpNode x, lpNode &target)
    {
        lpNode targetFather = target->father;
        while (x->father != targetFather)
            if (x->father == target)
                rotate(x);
            else if ((x->father->father->child[0] == x->father) == (x->father->child[0]
                == x))
                rotate(x->father), rotate(x);
            else
                rotate(x), rotate(x);
        target = x;
    }
    lpNode find(int x)
    {
        lpNode s = root;
        while (true)
            if (s == 0)
                return 0;
            else if (s->val == x)
            {
                Splay(s, root);
                return s;
            }
            else
                s = s->child[x > s->val];
    }
    lpNode join(lpNode x, lpNode y)
    {
        if (x == 0)
            return y;
        else
        {
            lpNode m = max_min(x, 1);
```

```
            Splay(m, x);
            m->child[1] = y;
            if (y) y->father = m;
            update(m);
            return x;
        }
    }

public:
    SplayTree(int _)
    {
        mem = new allocator<Node>(_);
    }
    lpNode max_min(lpNode x, int i)
    {
        while (x && x->child[i]) x = x->child[i];
        return x;
    }
    void insert(int x)
    {
        if (root == 0)
            root = mem->alloc()->Init(x, 0);
        else
        {
            lpNode s = root, p = 0;
            while (s)
                p = s, s = s->child[x > s->val];
            s = mem->alloc()->Init(x, p);
            p->child[x > p->val] = s;
            Splay(s, root);
        }
    }
    void remove(int x)
    {
        lpNode p = find(x);
        if (p)
        {
            root = join(p->child[0], p->child[1]);
            root->father = 0;
            mem->free(p);
        }
    }
    int rank(int x)
    {
        lpNode s = root;
        int ans = 0;
        while (s)
            if (x <= s->val)
                s = s->child[0];
            else
                ans = ans + 1 + sz(s->child[0]), s = s->child[1];
        return ans + 1;
```

```
150     }
151     lpNode select(int x)
152     {
153         lpNode s = root;
154         while (s)
155             if (x == sz(s->child[0]) + 1)
156                 return s;
157             else if (x <= sz(s->child[0]))
158                 s = s->child[0];
159             else
160                 x = x - 1 - sz(s->child[0]), s = s->child[1];
161     }
162     lpNode near(int x, int d)
163     {
164         lpNode s = root, ans = 0;
165         while (s)
166             if (d ? s->val > x : s->val < x)
167                 ans = s, s = s->child[d ^ 1];
168             else
169                 s = s->child[d];
170         return ans;
171     }
172 };
173 int main()
174 {
175     SplayTree st(arrSize);
176     int n, op, num;
177     scanf("%d", &n);
178     for (int i = 0; i < n; i++)
179     {
180         scanf("%d%d", &op, &num);
181         if (op == 1)
182             st.insert(num);
183         else if (op == 2)
184             st.remove(num);
185         else if (op == 3)
186             printf("%d\n", st.rank(num));
187         else if (op == 4)
188             printf("%d\n", getVal(st.select(num)));
189         else if (op == 5)
190             printf("%d\n", getVal(st.near(num, 0)));
191         else if (op == 6)
192             printf("%d\n", getVal(st.near(num, 1)));
193     }
194     return 0;
195 }
```

9.1.3 Treap

简介 权值满足堆性质，**期望**时间复杂度为 $\log n$ 的二叉搜索树。

做法 插入之后维护一下堆性质，乱搞搞就好了，递归的方法写起来很容易，并且不易写错。删除的时候就把那个节点转到一条链上，然后删掉它。

示例代码 模板题

BZOJ3224 - 普通平衡树

```
#include <cstdio>
#include <cstdlib>
#define sz(x) ((x) == 0 ? 0 : (x)->sz)
template <typename T>
struct Allocator
{
    int idx;
    T *data;
    T **ptr;
    Allocator(int size)
    {
        idx = 0;
        data = new T[size];
        ptr = new T *[size];
        for (int i = 0; i < size; i++) ptr[i] = &data[i];
    }
    ~Allocator()
    {
        delete[] data;
        delete[] ptr;
    }
    T *alloc() { return ptr[idx++]; }
    void free(T *pt) { ptr[--idx] = pt; }
};
struct Treap
{
    typedef struct Node
    {
        int val, pri, sz;
        Node *ch[2];
        void Init(int x = 0)
        {
            pri = rand();
            ch[0] = ch[1] = 0;
            val = x;
            sz = 1;
        }
        void update()
        {
            sz = 1 + sz(ch[0]) + sz(ch[1]);
        }
    } * lpNode;
    Treap(int max_size, int _seed)
    {
        srand(_seed);
        alloc = new Allocator<Node>(max_size);
```

```
        root = 0;
    }
    void ins(int x)
    {
        insert(root, x);
    }
    void del(int x)
    {
        remove(root, x);
    }
    int rnk(int x)
    {
        int ans = 1;
        lpNode cur = root;
        while (cur)
            if (x <= cur->val)
                cur = cur->ch[0];
            else
                ans += sz(cur->ch[0]) + 1, cur = cur->ch[1];
        return ans;
    }
    int kth(int x)
    {
        lpNode cur = root;
        while (cur)
            if (x == sz(cur->ch[0]) + 1)
                return cur->val;
            else if (x <= sz(cur->ch[0]))
                cur = cur->ch[0];
            else
                x -= sz(cur->ch[0]) + 1,
                    cur = cur->ch[1];
        return 0;
    }

private:
    Allocator<Node> *alloc;
    lpNode root;
    void rotate(lpNode &node, int d)
    {
        if (node == 0) return;
        lpNode ch = node->ch[d ^ 1];
        if (ch)
        {
            node->ch[d ^ 1] = ch->ch[d];
            ch->ch[d] = node;
            node = ch;
            node->ch[d]->update();
            node->update();
        }
    }
    void insert(lpNode &node, int x)
```

```
    {
        if (node == 0)
            (node = alloc->alloc())->Init(x);
        else
        {
            int cmp = x < node->val;
            insert(node->ch[cmp ^ 1], x);
            node->update();
            if (node->ch[cmp ^ 1]->pri < node->pri) rotate(node, cmp);
        }
    }
    void remove(lpNode &node, int x)
    {
        if (node == 0)
            return;
        else
        {
            if (x < node->val)
                remove(node->ch[0], x);
            else if (x > node->val)
                remove(node->ch[1], x);
            else if (node->ch[0] == 0)
                node = node->ch[1];
            else if (node->ch[1] == 0)
                node = node->ch[0];
            else
            {
                int d = node->ch[0]->pri < node->ch[1]->pri;
                rotate(node, d);
                remove(node->ch[d], x);
            }
            if (node) node->update();
        }
    }
};
int main()
{
    Treap T(100005, 3224);
    int n;
    scanf("%d", &n);
    for (int i = 0, op, x; i < n; i++)
    {
        scanf("%d%d", &op, &x);
        if (op == 1) T.ins(x);
        if (op == 2) T.del(x);
        if (op == 3) printf("%d\n", T.rnk(x));
        if (op == 4) printf("%d\n", T.kth(x));
        if (op == 5) printf("%d\n", T.kth(T.rnk(x) - 1));
        if (op == 6) printf("%d\n", T.kth(T.rnk(x + 1)));
    }
    return 0;
}
```

9.1.4 练习题

BZOJ1588 营业额统计 其实可以用`std::set`水过去。

BZOJ1588 - 营业额统计

```
#include <cstdio>
#include <cstdlib>
static const int arrSize = 40000;
typedef int valType;
struct Node
{
    valType val;
    Node *child[2], *father;
};
typedef Node *lpNode;
struct allocator
{
private:
    int i;
    Node data[arrSize];
    lpNode pt[arrSize];

public:
    allocator()
    {
        this->i = 0;
        for (int j = 0; j < arrSize; j++) pt[j] = data + j;
    }
    lpNode alloc() { return pt[i++]; }
    void free(lpNode ptr) { pt[--i] = ptr; }
} mem;
void rotate(lpNode x, int direction)
{
    lpNode p = x->father;
    p->child[!direction] = x->child[direction];
    if (x->child[direction]) x->child[direction]->father = p;
    x->father = p->father;
    if (p->father) p->father->child[p != p->father->child[0]] = x;
    p->father = x;
    x->child[direction] = p;
}
void Splay(lpNode x, lpNode target)
{
    lpNode p;
    target = target->father;
    while (x->father != target)
    {
        p = x->father;
        if (p->father == target)
        {
            rotate(x, x == p->child[0]);
            break;
```

```
        }
        else if (x == p->child[0])
            if (p == p->father->child[0]) {
                rotate(p, 1);
                rotate(x, 1);
            }
            else
            {
                rotate(x, 1);
                rotate(x, 0);
            }
        else if (p == p->father->child[1])
        {
            rotate(p, 0);
            rotate(x, 0);
        }
        else
        {
            rotate(x, 0);
            rotate(x, 1);
        }
    }
}
lpNode max(lpNode x)
{
    while (x && x->child[1]) x = x->child[1];
    return x;
}
lpNode min(lpNode x)
{
    while (x && x->child[0]) x = x->child[0];
    return x;
}
lpNode find(valType x, lpNode s)
{
    lpNode p = s;
    while (true)
        if (p == 0)
            return p;
        else if (p->val == x)
        {
            Splay(p, s);
            return p;
        }
        else
            p = p->child[x > p->val];
}
lpNode pred(valType x, lpNode s) { return max(find(x, s)->child[0]); }
lpNode succ(valType x, lpNode s) { return min(find(x, s)->child[1]); }
void ins(valType x, lpNode &s)
{
    if (s == 0)
```

```
    {
        s = mem.alloc();
        s->val = x;
    }
    else
    {
        lpNode p = 0, _s = s;
        while (s)
        {
            p = s;
            s = s->child[x > s->val];
        }
        if (p == 0)
            s->val = x;
        else
        {
            lpNode n = mem.alloc();
            n->val = x;
            n->father = p;
            p->child[x > p->val] = n;
            s = _s;
            Splay(n, s);
            s = n;
        }
    }
}
valType _min(valType a, valType b) { return (a < b ? a : b); }
lpNode root = 0;
int main()
{
    int n, tmp, ans = 0;
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &tmp);
        ins(tmp, root);
        if (i == 0)
            ans += tmp;
        else
        {
            lpNode p = pred(tmp, root), s = succ(tmp, root);
            if (p == 0)
                ans += abs(tmp - s->val);
            else if (s == 0)
                ans += abs(tmp - p->val);
            else
                ans += _min(abs(tmp - s->val), abs(tmp - p->val));
        }
    }
    printf("%d", ans);
    return 0;
}
```

9.2 主席树

前导知识 只要会线段树就行

简介 主席树是一系列差分后的线段树。

示例代码 模板题

POJ2104 - K-th Number

```
#include <algorithm>
#include <cstdio>
#include <cstring>
const int N = int(1e5 + 5);
int n, a[N], b[N], idx, rt[N];
struct node
{
    int ch[2], sz;
} T[N * 20];
void insert(int y, int &x, int l, int r, int p)
{
    T[x = ++idx] = T[y];
    T[x].sz++;
    if (l == r - 1) return;
    int m = (l + r) >> 1;
    p < m ?
        insert(T[y].ch[0], T[x].ch[0], l, m, p) :
        insert(T[y].ch[1], T[x].ch[1], m, r, p);
}
int query(int nl, int nr, int l, int r, int k)
{
    if (l == r - 1) return l;
    int delta = T[T[nr].ch[0]].sz - T[T[nl].ch[0]].sz, m = (l + r) >> 1;
    return delta >= k ?
        query(T[nl].ch[0], T[nr].ch[0], l, m, k) :
        query(T[nl].ch[1], T[nr].ch[1], m, r, k - delta);
}
int main()
{
    int q;
    scanf("%d%d", &n, &q);
    for (int i = 1; i <= n; i++) scanf("%d", a + i);
    memcpy(b, a + 1, n << 2);
    std::sort(b, b + n);
    int m = int(std::unique(b, b + n) - b);
    for (int i = 1; i <= n; i++)
        a[i] = int(std::lower_bound(b, b + m, a[i]) - b);
    for (int i = 1; i <= n; i++) insert(rt[i - 1], rt[i], 0, m, a[i]);
    while (q--)
    {
        int l, r, k;
        scanf("%d%d%d", &l, &r, &k);
        printf("%d\n", b[query(rt[l - 1], rt[r], 0, m, k)]);
```

```
    }
    return 0;
}
```

分析 首先，主席树里面的线段树是一堆“权值线段树”，所谓权值线段树，就是维护各个值的信息。与一般的线段树维护的东西不同，一般的线段树维护的是位置上的信息，例如这个位置上值为多少，这段区间的最大值/和是多少一类的；而权值线段树维护的是这个值出现了几次之类的信息，如果看的更长远一些，我们可以认为普通线段树是维护的时域上的信息，而权值线段树是维护的频域上的信息，当然都是离散的。这个题给出了一个序列，第一步将其离散化，第二步将每一个数依次插入一棵与之前的线段树有公用节点的新线段树的这个点的大小所代表的位置上，这样就可以看作将时域信息转成频域信息了。然后在指定区间的线段树之差上“二分”，其实就是在线段树上按照大小移动，然后输出就行了。

10 根号类算法

一类时间复杂度中带有 $\sqrt{n}$ 的算法。

10.1 分块算法

一种序列的通用维护方法。通过将序列分成 $\sqrt{n}$ 块进行暴力维护解决。一般的线段树等数据结构能够使用的前提是区间的信息可以快速合并，而对于更加通用的问题并不能很好的解决，这里描述的分块算法本质上就是暴力，每块内暴力维护信息，块之间同样暴力维护信息，所以可以不用受信息性质的影响，在一个稳定（却较大）的时间复杂度下解决问题。

示例代码 显然区间众数是不能快速合并的。

BZOJ2724 - 蒲公英

```
#include <algorithm>
#include <cctype>
#include <cmath>
#include <cstdio>
#include <cstring>
inline void read(int &x)
{
    int ch = x = 0;
    while (!isdigit(ch)) ch = getchar();
    for (; isdigit(ch); ch = getchar()) x = x * 10 + ch - '0';
}
const int B = 205, N = B * B, inf = 0x3f3f3f3f;
int n, q, S, bcnt, a[N], b[N], cnt[N][B], f[B][B], cnt2[N], vis[N];
bool isR[N];
int main()
{
    memset(vis, -1, sizeof(vis));
    read(n), read(q), S = (int)sqrt(n);
    for (int i = 0; i < n; i++) read(a[i]);
    memcpy(b, a, n << 2);
    std::sort(b, b + n);
    int m = int(std::unique(b, b + n) - b);
    for (int i = 0; i < n; i++) a[i] = int(std::lower_bound(b, b + m, a[i]) - b);
```

```
    for (int i = 0, bidx = 0; i < n; i++, bidx++)
    {
        if (bidx == S) bidx = 0, bcnt++, isR[i - 1] = true;
        cnt[a[i]][bcnt]++;
    }
    isR[n - 1] = true;
    bcnt = n / S + (n % S > 0);
    for (int i = 0; i < n; i++)
        for (int j = 1; j < bcnt; j++)
            cnt[i][j] += cnt[i][j - 1];
    for (int i = 0; i < bcnt; i++)
    {
        memset(cnt2, 0, sizeof(cnt2));
        for (int j = i * S, tans = 0; j < n; j++)
        {
            cnt2[a[j]]++;
            if (cnt2[a[j]] > cnt2[tans] || (cnt2[a[j]] == cnt2[tans] && a[j] < tans))
                tans = a[j];
            if (isR[j]) f[i][j / S] = tans;
        }
    }
    int ans = 0, l, r;
    while (q--)
    {
        read(l), read(r);
        l = (l + ans - 1) % n, r = (r + ans - 1) % n;
        if (l > r) std::swap(l, r);
        if (r - l < S * 2)
        {
            int tans = N - 1, tcnt = -1;
            for (int i = l; i <= r; i++)
            {
                if (vis[a[i]] != q) vis[a[i]] = q, cnt2[a[i]] = 0;
                cnt2[a[i]]++;
                if (cnt2[a[i]] > tcnt || (cnt2[a[i]] == tcnt && a[i] < tans))
                    tcnt = cnt2[tans = a[i]];
            }
            printf("%d\n", ans = b[tans]);
        }
        else
        {
            int bl = l / S, br = r / S;
            int bll = bl * S, brr = (br + 1) * S - 1;
            if (brr >= n) brr = n - 1;
            if (l > bll) bll = (++bl) * S;
            if (r < brr) brr = (--br + 1) * S - 1;
            int curAns = f[bl][br];
            int curAnsCnt = cnt[curAns][br];
            if (bl > 0) curAnsCnt -= cnt[curAns][bl - 1];
            for (int i = l; i < bll; i++)
            {
                if (vis[a[i]] != q)
```

```
                {
                    vis[a[i]] = q, cnt2[a[i]] = cnt[a[i]][br];
                    if (bl > 0) cnt2[a[i]] -= cnt[a[i]][bl - 1];
                }
                cnt2[a[i]]++;
                if (cnt2[a[i]] > curAnsCnt || (cnt2[a[i]] == curAnsCnt && a[i] < curAns))
                    curAnsCnt = cnt2[curAns = a[i]];
            }
            for (int i = r; i > brr; i--)
            {
                if (vis[a[i]] != q)
                {
                    vis[a[i]] = q, cnt2[a[i]] = cnt[a[i]][br];
                    if (bl > 0) cnt2[a[i]] -= cnt[a[i]][bl - 1];
                }
                cnt2[a[i]]++;
                if (cnt2[a[i]] > curAnsCnt || (cnt2[a[i]] == curAnsCnt && a[i] < curAns))
                    curAnsCnt = cnt2[curAns = a[i]];
            }
            printf("%d\n", ans = b[curAns]);
        }
    }
    return 0;
}
```

这个题目描述不完全，于是我就调试了两天，真是浪费生命。

10.2 莫队算法

将询问分块，依次暴力处理。

示例代码 暴力维护区间内不同点的个数，经过对询问的恰当排序，可以保证时间复杂度为 $n\sqrt{n}$

BZOJ1878 - HH 的项链

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstring>
int a[50001], cnt[1000005], ans[200005];
struct query
{
    int id, l, r, b;
} Q[200005];
inline bool operator<(const query &x, const query &y) { return x.b < y.b || (x.b == y.b
    && x.r < y.r); }
int main()
{
    int n, m, S;
    scanf("%d", &n), S = int(sqrt(n));
    for (int i = 1; i <= n; i++) scanf("%d", &a[i]);
    scanf("%d", &m);
    for (int i = 0; i < m; i++)
```

```
    {
        query &q = Q[i];
        scanf("%d%d", &q.l, &q.r);
        q.id = i, q.b = Q[i].l / S;
    }
    std::sort(Q, Q + m);
    for (int i = 0, L = 0, R = 0, tot = 0; i < m; i++)
    {
        int l = Q[i].l, r = Q[i].r;
        while (L < l)
            if (!--cnt[a[L++]]) --tot;
        while (L > l)
            if (!cnt[a[--L]]++) ++tot;
        while (R < r)
            if (!cnt[a[++R]]++) ++tot;
        while (R > r)
            if (!--cnt[a[R--]]) --tot;
        ans[Q[i].id] = tot;
    }
    for (int i = 0; i < m; i++)
        printf("%d\n", ans[i]);
    return 0;
}
```

11 FFT

这里介绍的是库利－图基快速傅里叶变换算法/Cooley–Tukey FFT algorithm。它能将离散傅里叶变换或其逆变换在 $O(n \log n)$ 的时间内计算出来（按照离散傅里叶变换的原始定义进行计算则需要 $O(n^2)$ 的时间）。

11.1 离散傅里叶变换

定义 对于 N 点序列 $\{x[n]\}_{0 \le n < N}$，它的离散傅里叶变换（DFT）为

$$\hat{x}[k] = \sum_{n=0}^{N-1} e^{-i\frac{2\pi}{N}nk} x[n] \qquad k = 0, 1, \ldots, N-1.$$

其中 e 是自然对数的底数，i 是虚数单位。通常以符号 $\mathcal{F}$ 表示这一变换，即

$$\hat{x} = \mathcal{F}x$$

离散傅里叶变换的逆变换（IDFT）为：

$$x\,[n] = \frac{1}{N} \sum_{k=0}^{N-1} e^{i\frac{2\pi}{N}nk} \hat{x}[k] \qquad n = 0, 1, \ldots, N-1.$$

可以记为：

$$x = \mathcal{F}^{-1}\hat{x}$$

显然如果按照定义进行计算的话，时间复杂度将会是 $O(n^2)$。

示例代码 实现中用到了 $e^{i\theta} = \cos\theta + i\sin\theta$

naïve DFT/IDFT

```
#include <cmath>
#include <cstdio>
#include <cstring>
const int N = 1000;
const double pi = acos(-1);
struct complex
{
    double r, i;
} a[N], b[N], tmp[N];
#define op(T, O) inline T operator O(const T &x, const T &y)
char s1[N], s2[N];
int ans[N];
op(complex, +) { return { x.r + y.r, x.i + y.i }; }
op(complex, -) { return { x.r - y.r, x.i - y.i }; }
op(complex, *) { return { x.r * y.r - x.i * y.i, x.r * y.i + x.i * y.r }; }
void naive_DFT(complex *arr, const int len)
{
    memset(tmp, 0, sizeof(tmp));
    for (int k = 0; k < len; k++)
        for (int n = 0; n < len; n++)
            tmp[k] = tmp[k] + arr[n] * complex{ cos(-2 * pi / len * k * n), sin(-2 * pi /
                 len * k * n) };
    memcpy(arr, tmp, sizeof(tmp));
}
void naive_IDFT(complex *arr, const int len)
{
    memset(tmp, 0, sizeof(tmp));
    for (int k = 0; k < len; k++)
        for (int n = 0; n < len; n++)
            tmp[k] = tmp[k] + arr[n] * complex{ cos(2 * pi / len * k * n), sin(2 * pi /
                len * k * n) };
    memcpy(arr, tmp, sizeof(tmp));
    for (int i = 0; i < len; i++)
        arr[i].r /= len;
}
int main()
{
    while (~scanf("%s%s", s1, s2))
    {
        int len1, len2, len;
        memset(a, 0, sizeof(a));
        memset(b, 0, sizeof(b));
        memset(ans, 0, sizeof(ans));
        len1 = (int)strlen(s1), len2 = (int)strlen(s2), len = len1 + len2;
        for (int i = 0; i < len1; i++)
            a[i].r = s1[len1 - 1 - i] - '0';
        for (int i = 0; i < len2; i++)
            b[i].r = s2[len2 - 1 - i] - '0';
        naive_DFT(a, len), naive_DFT(b, len);
```

```
        for (int i = 0; i < len; i++)
            a[i] = a[i] * b[i];
        naive_IDFT(a, len);
        for (int i = 0; i < len; i++)
            ans[i] = int(a[i].r + 0.5);
        for (int i = 0; i < len; i++)
            ans[i + 1] += ans[i] / 10, ans[i] %= 10;
        int high = 0;
        for (int i = len - 1; high == 0 && i >= 0; i--)
            if (ans[i])
                high = i;
        for (int i = high; i >= 0; i--)
            putchar(ans[i] + '0');
        putchar('\n');
    }
    return 0;
}
```

11.2 练习题

示例代码 两个大整数相乘

HDU1402 - A * B Problem Plus

```
#include <cmath>
#include <cstdio>
#include <cstring>
const int N = 500005;
const double pi = acos(-1);
struct complex
{
    double r, i;
} a[N], b[N];
inline complex operator+(const complex &x, const complex &y)
{
    return { x.r + y.r, x.i + y.i };
}
inline complex operator-(const complex &x, const complex &y)
{
    return { x.r - y.r, x.i - y.i };
}
inline complex operator*(const complex &x, const complex &y)
{
    return { x.r * y.r - x.i * y.i, x.r * y.i + x.i * y.r };
}
char s1[N], s2[N];
int ans[N];
void Rader(complex *F, int len)
{
    complex t;
    int j = len >> 1;
    for (int i = 1; i < len - 1; i++)
```

```
    {
        if (i < j)
            t = F[i], F[i] = F[j], F[j] = t;
        int k = len >> 1;
        while (j >= k)
            j -= k, k >>= 1;
        if (j < k)
            j += k;
    }
}
void fft(complex *F, int len, int flag)
{
    Rader(F, len);
    for (int h = 2; h <= len; h <<= 1)
    {
        const complex wn = { cos(-flag * 2 * pi / h), sin(-flag * 2 * pi / h) };
        for (int j = 0; j < len; j += h)
        {
            complex w = { 1, 0 };
            int m = h >> 1;
            for (int k = j; k < j + m; k++, w = w * wn)
            {
                complex u = F[k], t = w * F[k + m];
                F[k] = u + t, F[k + m] = u - t;
            }
        }
    }
}
int main()
{
    while (~scanf("%s%s", s1, s2))
    {
        int len1, len2, len;
        memset(a, 0, sizeof(a));
        memset(b, 0, sizeof(b));
        memset(ans, 0, sizeof(ans));
        len1 = (int)strlen(s1) << 1, len2 = (int)strlen(s2) << 1;
        for (len = 1; len < len1 || len < len2;)
            len <<= 1;
        len1 >>= 1, len2 >>= 1;
        for (int i = 0; i < len1; i++)
            a[i].r = s1[len1 - 1 - i] - '0';
        for (int i = 0; i < len2; i++)
            b[i].r = s2[len2 - 1 - i] - '0';
        fft(a, len, 1);
        fft(b, len, 1);
        for (int i = 0; i < len; i++)
            a[i] = a[i] * b[i];
        fft(a, len, -1);
        for (int i = 0; i < len; i++)
            ans[i] = int(a[i].r / len + 0.5);
        for (int i = 0; i < len; i++)
```

```
            ans[i + 1] += ans[i] / 10, ans[i] %= 10;
        int high = 0;
        for (int i = len - 1; high == 0 && i >= 0; i--)
            if (ans[i])
                high = i;
        for (int i = high; i >= 0; i--)
            putchar(ans[i] + '0');
        putchar('\n');
    }
    return 0;
}
```

12 弃坑了

胡写的板子 DA☆ZE

一些字符串处理/三个自动机

后缀自动机（SAM）

- 关于广义后缀自动机：
  - 给定n个字符串，询问每个字符串有多少子串（不包括空串）是所有n个字符串中至少k个字符串的子串（包括本身）。
  - 对于多串问题，普通SAM已经无法胜任。有各种应对这类多串问题的方法：

(1) 直接建SAM，每次插入新串时将lst设为1即可，不会证明正确性。

$ps:$ 这个必须每个串不相同，否则可能会炸

(2) 和SA一样在多个串接在一起中间加入分隔符'#'，然后直接建SAM。

(3) 离线做法：对所有串建出Trie，然后BFS遍历Trie，每个点在父亲的基础上$extend$即可（参考楼下代码）。时空复杂度$O(|A||T|)$。

(4) 在线做法：模板如下，时间复杂度$O(|A||T|+G(T))$，空间复杂度$O(|A||T|)$，其中$|T|$是Trie的大小，$|A|$是字符集大小，$G(T)$是Trie上所有叶子深度之和。

$ $

- 结论：如果题目是以多个模板串形式给出的，建议使用在线做法（离线做法虽然复杂度优秀但需要先建出Trie所以大部分情况常数会较大）。如果是直接以Trie形式给出的，那么直接用离线算法就好了。

```
struct SAM{
    struct node{
        int ch[26];
        int len,fa;
        node(){memset(ch,0,sizeof(ch));len=0;}
    }dian[maxn<<1];

    int tot=1;
    /*------(3) trie 上跑 or 多个字符串不相同，开始用pos=add(c,1),之后pos=add(c,pos)------*/
    int add(int c,int last_p){
        int p=last_p;int np=++tot;
        //（1）直接加可以改成int p=las;int np=las=++tot;
        // 外界加一个int las=1;返回值改void，删掉return，int last_p

        dian[np].len=dian[p].len+1;
        for(;p&&!dian[p].ch[c];p=dian[p].fa)dian[p].ch[c]=np;
        if(!p)dian[np].fa=1;
        else
        {
            int q=dian[p].ch[c];
            if(dian[q].len==dian[p].len+1)dian[np].fa=q;
            else
            {
                int nq=++tot;dian[nq]=dian[q];
                dian[nq].len=dian[p].len+1;
                dian[q].fa=dian[np].fa=nq;
                for(;p&&dian[p].ch[c]==q;p=dian[p].fa)dian[p].ch[c]=nq;
            }
        }
        return np;
    }
```

```
    long long count_diff(){
        long long res=0;
        for(int i=1;i<=tot;i++)res+=dian[i].len-dian[dian[i].fa].len;
        return res;
    }//本质不同字符串数量

    //传奇操作，O(n)后缀bfs序--后缀链接fail的bfs序排名
    void gos(){
        for(int i=1;i<=tot;i++)bfnsum[dian[i].len]++;//len长有多少
        for(int i=1;i<=tot;i++)bfnsum[i]+=bfnsum[i-1];//前缀和
        for(int i=1;i<=tot;i++)bfn[bfnsum[dian[i].len]--]=i;//一个一个放回去
    }

}S_A_M;

//-------endpos集合算数-----
long long anslist[maxn];//长度为i的最大endpos集合数
int n;
struct SAM{
    struct node{
        int ch[26];
        int len,fa;
        long long cnttime;//endpos集合大小计数
        node(){memset(ch,0,sizeof(ch));len=0;cnttime=1;}
    }dian[maxn<<1];

    int tot=1;
    int las=1;
    /*------(3) trie 上跑 or 多个字符串不相同，开始用pos=add(c,1),之后pos=add(c,pos)------*/
    void add(int c){
        int p=las;int np=las=++tot;
        //（1）直接加可以改成int p=las;int np=las=++tot;
        // 外界加一个int las=1;返回值改void，删掉return，int last_p
        dian[np].len=dian[p].len+1;
        dian[np].cnttime=1;//正常开点就是1
        for(;p&&!dian[p].ch[c];p=dian[p].fa)dian[p].ch[c]=np;
        if(!p)dian[np].fa=1;
        else
        {
            int q=dian[p].ch[c];
            if(dian[q].len==dian[p].len+1)dian[np].fa=q;
            else
            {
                int nq=++tot;dian[nq]=dian[q];
                dian[nq].len=dian[p].len+1;
                dian[q].fa=dian[np].fa=nq;
                dian[nq].cnttime=0;//克隆的点不能重复计数
                for(;p&&dian[p].ch[c]==q;p=dian[p].fa)dian[p].ch[c]=nq;
            }
        }
    }

    long long count_diff(){
        long long res=0;
        for(int i=1;i<=tot;i++)res+=dian[i].len-dian[dian[i].fa].len;
        return res;
    }//本质不同字符串数量

    int bfn[maxn<<1],bfnsum[maxn<<1];

    void gos(){
        for(int i=1;i<=tot;i++)bfnsum[dian[i].len]++;
        for(int i=1;i<=tot;i++)bfnsum[i]+=bfnsum[i-1];
```

```
        for(int i=1;i<=tot;i++)bfn[bfnsum[dian[i].len]--]=i;//后缀树bfs序

        for(int i=tot;i>=1;i--)dian[dian[bfn[i]].fa].cnttime+=dian[bfn[i]].cnttime;//反向
倒推
    }

    void solve(){
        for(int i=1;i<=tot;i++){
            int rlen=dian[i].len;
            anslist[rlen]=max(anslist[rlen],dian[i].cnttime);
        }//性质，单调不升。一个显然的例子是长度为4的endpos集合大小肯定不会比3要大，否则3可以和4一样大
        for(int i=n;i>=1;i--)anslist[i]=max(anslist[i],anslist[i+1]);
        //反向更新，i被置为i+1~n的anslist的max
        for(int i=1;i<=n;i++)printf("%lld\n",anslist[i]);
    }

}S_A_M;

//--------------------(4)网上dalao代码----------------------//
int work(int p,int c){
    int nq=++nd,q=son[p][c]; mx[nq]=mx[p]+1;
    fa[nq]=fa[q]; fa[q]=nq;
    memcpy(son[nq],son[q],sizeof(son[q]));
    while (p && son[p][c]==q) son[p][c]=nq,p=fa[p];
    return nq;
}

int ext(int p,int c){
    if (son[p][c]){
        int q=son[p][c];
        if (mx[q]==mx[p]+1) return q; else return work(p,c);
    }else{
        int np=++nd; mx[np]=mx[p]+1;
        while (p && !son[p][c]) son[p][c]=np,p=fa[p];
        if (!p) fa[np]=1;
        else{
            int q=son[p][c];
            if (mx[q]==mx[p]+1) fa[np]=q; else fa[np]=work(p,c);
        }
        return np;
    }
}

//给定n个字符串，询问每个字符串有多少子串（不包括空串）是所有n个字符串中至少k个字符串的子串（包括本
身）。//
void solve(){
    int u; ll ans;
    rep(i,1,n){
        u=1;
        rep(j,0,len[i]){
            u=son[u][s[i][j]-'a']; int p=u;
            while (p && vis[p]!=i) tot[p]++,vis[p]=i,p=fa[p];
        }
    }
    radix();
    rep(i,2,nd) u=q[i],f[u]=f[fa[u]]+(tot[u]>=k ? mx[u]-mx[fa[u]] : 0);
    rep(i,1,n){
        u=1; ans=0;
        rep(j,0,len[i]) u=son[u][s[i][j]-'a'],ans+=f[u];
        printf("%lld ",ans);
    }
}
```

AC自动机

```
struct AC{
    struct acnode{
        acnode *next[26];
        acnode *fail;
        int sum;
        acnode(){
            sum=0;
            for(int i=0;i<26;i++)next[i]=NULL;
            fail=NULL;
        }
    };
    acnode *root=new acnode;
    void insert(string s){
        int len=s.length();
        acnode *t=root;
        for(int i=0;i<len;i++){
            int cut=s[i]-'a';
            if(t->next[cut]==NULL){t->next[cut]=new acnode;}
            t=t->next[cut];
        }
        t->sum++;
    }

    void getfail(){
        queue <acnode*> q;
        acnode *t=root;
        for(int i=0;i<26;i++){
            if(t->next[i]!=NULL){
                t->next[i]->fail=root;
                q.push(t->next[i]);
            }
        }
        while(!q.empty()){
            t=q.front();
            q.pop();
            for(int i=0;i<26;i++){
                if(t->next[i]!=NULL){
                    acnode *p=t->fail;
                    while(p!=NULL){
                        if(p->next[i]!=NULL){
                            t->next[i]->fail=p->next[i];
                            break;
                        }
                        p=p->fail;
                    }
                    if(p==NULL){
                        t->next[i]->fail=root;
                    }
                    q.push(t->next[i]);
                }
            }
        }
    }
    int ans=0;
    void acfun(string s){
        acnode *p=root;
        int len=s.length();
        for(int i=0;i<len;i++){
            int cut=s[i]-'a';
```

```
            while(p->next[cut]==NULL && p!=NULL)p=p->fail;
            p=p->next[cut];
            if(p==NULL)p=root;

            acnode *t=p;
            while(t!=root){
                if(t->sum>=0){
                    ans+=t->sum;
                    t->sum=-1;
                }
                else break;
                t=t->fail;
            }
        }
    }

}A_C;
```

- lrj的AC自动机（感觉比我优美多了）：

```
待补充qwq，P215蓝book
```

回文树（PAM）/ 回文自动机

- len[i]表示编号为i的节点表示的回文串的长度（一个节点表一个回文串）
- next[i][c]表示编号为i的节点表示的回文串在两边添加字符以后变成的回文串的编号（和字典树类似）。
- fail[i]表示节点i失配以后跳转不等于自身的节点i表示的回串的最长后缀回文串（和AC自动机类似）。
- cnt[i]表示节点i表示的本质不同的串的个数（建树时求出的不是完全的，最后count()函数跑一遍以后才是正确的）
- num[i]表示以节点i表示的最长回文串的最右端点为回文串结的回文串个数。
- last指向新添加一个字母后所形成的最长回文串表示的节点。
- S[i]表示第i次添加的字符（一开始设S[0] = -1（可以是任一个在串S中不会出现的字符））。
- p表示添加的节点个数。
- n表示添加的字符个数。

```
struct Palindromic_Tree {
    int next[MAXN][N] ;//next指针，next指针和字典树类似，指向的串为当前串两端加上同一个字符构成
    int fail[MAXN] ;//fail指针，失配后跳转到fail指针指向的节点
    int cnt[MAXN] ;
    int num[MAXN] ;
    int len[MAXN] ;//len[i]表示节点i表示的回文串的长度
    int S[MAXN] ;//存放添加的字符
    int last ;//指向上一个字符所在的节点，方便下一次add
    int n ;//字符数组指针
    int p ;//节点指针,p代表了节点个数，正好是（0 ~ p-1）

    int newnode ( int l ) {//新建节点
        for ( int i = 0 ; i < N ; ++ i ) next[p][i] = 0 ;
        cnt[p] = 0 ;
        num[p] = 0 ;
        len[p] = l ;
        return p ++ ;
    }

    void init () {//初始化
        p = 0 ;
        newnode (  0 ) ;
        newnode ( -1 ) ;//顺序别变啊...
        last = 0 ;
        n = 0 ;
```

```
        S[n] = -1 ;//开头放一个字符集中没有的字符，减少特判
        fail[0] = 1 ;
    }

    int get_fail ( int x ) {//和KMP一样，失配后找一个尽量最长的
        while ( S[n - len[x] - 1] != S[n] ) x = fail[x] ;
        return x ;
    }

    void add ( int c ) {
        c -= 'a' ;
        S[++ n] = c ;
        int cur = get_fail ( last ) ;//通过上一个回文串找这个回文串的匹配位置
        if ( !next[cur][c] ) {//如果这个回文串没有出现过，说明出现了一个新的本质不同的回文串
            int now = newnode ( len[cur] + 2 ) ;//新建节点
            fail[now] = next[get_fail ( fail[cur] )][c] ;//和AC自动机一样建立fail指针，以便失
配后跳转
            next[cur][c] = now ;
            num[now] = num[fail[now]] + 1 ;//看上去是本质不同的num
        }
        last = next[cur][c] ;
        cnt[last] ++ ;
    }

    void count () {
        for ( int i = p - 1 ; i >= 0 ; -- i ) cnt[fail[i]] += cnt[i] ;
        //父亲累加儿子的cnt，因为如果fail[v]=u，则u一定是v的子回文串！
    }

//你已经可以熟练操作这棵树了！
/*-------------------------------------------------------------------------------*/
//将串中本质不同的回文子串组成集合S，任取S中的两个元素(a，b)，问a是b的子串的对数
    bool vis[maxn];
    long long nxtr[maxn],fatr[maxn];
    void dfs(int x){
        if(x!=0&&x!=1)nxtr[x]=1;
        vector <int> A;
        for(int i=x;i>1;i=fail[i]){
            if(vis[i])break;
            vis[i]=1;
            A.push_back(i);
            fatr[x]++;
        }
        for(int i=0;i<N;i++){
            int y=next[x][i];
            if(y){
                dfs(y);
                nxtr[x]+=nxtr[y];
            }
        }
        for(int i=A.size()-1;i>=0;i--){
            vis[A[i]]=0;
        }//纯vis回调可能交叉线路爆炸...导致河流袭夺一样的事件
    }

    long long solve(){
        long long ans=0;
        memset(nxtr,0,sizeof(nxtr));
        memset(fatr,0,sizeof(fatr));
        dfs(0);dfs(1);
        for(int i=2;i<p;i++)ans+=(nxtr[i]*fatr[i]-1);
        return ans;
    }
```

```
/*--------------------------------------------------------------------------------*/
}P_A_M;
```

kmp（mp） 和 exkmp

kmp（mp）相关

- mp算法=kmp算法低配版，但是时间复杂度已经到了下限，并且足够使用
- lrj的kmp（实际上是mp，感觉我学了大几年都是mp算法）：

```
void getFail(char* P,int* f) {
    int m=strlen(P);
    f[0]=0;
    f[1]=0;
    for(int i=1; i<m; i++) {
        int j=f[i];
        while(j&&P[i]!=P[j])j=f[j];
        f[i+1]= P[i]==P[j] ? j+1 : 0;
    }
}
//这个fail是跳到哪里哦，举个例子：
//pos(i): 0 1 2 3 4 5 6 7 8 9 10 11
//p:      A B R A C A D A B R A  #
//fail:   0 0 0 0 1 0 1 0 1 2 3  4

void find(char* T,char* P,int* f) {
    int n=strlen(T),m=strlen(P);
    getFail(P,f);
    int j=0;
    for(int i=0; i<n; i++) {
        while(j&&P[j]!=T[i])j=f[j];
        if(P[j]==T[i])j++;
        if(j==m)printf("%d\n",i-m+1);//找到了!
    }
}
```

传奇exkmp

- exkmp,用来求一个串S和另一个串T的**所有后缀**的最长公共前缀~
- 俩数组，一个next（和上边那个不一样！这是开头长度，fail是结尾长度）是去匹配串（S）的，另一个是extend就是要的结果啦，是被匹配的串（T）的
- 直接贴代码，能看懂的...

```
int Next[maxn],extend[maxn];
void getNext(char str[]){//[0,len)
    int i=0,len=strlen(str);
    Next[0]=len;
    while(str[i]==str[i+1] && i+1<len)i++;
    Next[1]=i;
    int po=1;
    for(i=2;i<len;i++){
        if(Next[i-po]+i<Next[po]+po)Next[i]=Next[i-po];
        else {
            int j=Next[po]+po-i;
            if(j<0)j=0;
            while(i+j<len&&str[j]==str[j+i])j++;
            Next[i]=j;po=i;
        }
    }
}
```

```
void exkmp(char s1[],char s2[]){
    int i=0,len=strlen(s1),l2=strlen(s2);
    getNext(s2);
    while(s1[i]==s2[i] && i<len)i++;
    extend[0]=i;
    int po=0;
    for(i=1;i<len;i++){
        if(Next[i-po]+i<extend[po]+po)extend[i]=Next[i-po];
        else {
            int j=extend[po]+po-i;
            if(j<0)j=0;
            while(i+j<len&&s1[j+i]==s2[j])j++;
            extend[i]=j;po=i;
        }
    }
}
```

SA!（后缀数组）

- 我终于看懂了倍增构造+简单使用...
- 但是代码莫得看懂咋写的...主要玩仨数组,sa,rank,**height**
- 简单来讲：rank数组rank[i]就是从i开始那个后缀（i~len）的字典序排名，sa数组sa[i]是排名为i的后缀开头位置是哪个
- pdf：后缀数组是“排第几的是谁？”，名次数组是“你排第几？”。容易看出，后缀数组和名次数组为互逆运算。
- 一份O(nlogn)的倍增构造代码模板：

```
int wa[maxn],wb[maxn],wv[maxn],ws[maxn];
int cmp(int *r,int a,int b,int l){
    return r[a]==r[b]&&r[a+l]==r[b+l];
}
void da(int *r,int *sa,int n,int m){
    int i,j,p,*x=wa,*y=wb,*t;
    for(i=0;i<m;i++)ws[i]=0;
    for(i=0;i<n;i++)ws[x[i]=r[i]]++;
    for(i=1;i<m;i++)ws[i]+=ws[i-1];
    for(i=n-1;i>=0;i--)sa[--ws[x[i]]]=i;
    for(j=1,p=1;p<n;j*=2,m=p){
        for(p=0,i=n-j;i<n;i++)y[p++]=i;
        for(i=0;i<n;i++)if(sa[i]>=j)y[p++]=sa[i]-j;
        for(i=0;i<n;i++)wv[i]=x[y[i]];
        for(i=0;i<m;i++)ws[i]=0;
        for(i=0;i<n;i++)ws[wv[i]]++;
        for(i=1;i<m;i++)ws[i]+=ws[i-1];
        for(i=n-1;i>=0;i--)sa[--ws[wv[i]]]=y[i];
        for(t=x,x=y,y=t,p=1,x[sa[0]]=0,i=1;i<n;i++)
            x[sa[i]]=cmp(y,sa[i-1],sa[i],j) ? p-1 : p++;
    }
    return;
}
```

- important：height数组
- 定义 height[i]=suffix(sa[i-1])和 suffix(sa[i])的最长公共前缀
- **所以是height[2]开始的...**

```
int rank[maxn],height[maxn];
void calheight(int *r,int *sa,int n){
    int i,j,k=0;
    for(i=1;i<=n;i++)rank[sa[i]]=i;
    for(i=0;i<n;height[rank[i++]]=k)
        for(k?k--:0,j=sa[rank[i]-1];r[i+k]==r[j+k];k++);
    return;
}
```

- 简单应用：（时刻记得height是rank相邻两个suffix（后缀）的值，height分组判定特别好用！）
  - 1.最长公共前缀：给定一个字符串，询问某两个后缀的最长公共前缀。
    - 直接RMQ对height找区间最小值。
  - 2.可重叠最长重复子串：给定一个字符串，求最长重复子串，这两个子串可以重叠。
    - 就是height最大值...
  - 3.不可重叠最长重复子串（pku1743）：给定一个字符串，求最长重复子串，这两个子串不能重叠。
    - 设k为长然后二分k按height分组（height >= k放一块，height < k的分开）判定：对于每组后缀，只须判断每个后缀的 sa 值的最大值和最小值之差是否不小于k。如果有一组满足，则说明存在，否则不存在。整个做法的时间复杂度为O(nlogn)。

一些数学

逆元 与 exgcd

- (如果a,n互质可以直接费马小定理$fast_pow(a,n-2)$)
- 费马小定理
  $A^{P-1} \equiv 1 \equiv A^0 (mod\ P)$
  $A^{P-2} \equiv \frac{1}{A} (mod\ P)$
- 欧拉定理：$a^{\phi(x)} \equiv 1 \pmod{x}$ **(x,a为正整数，且x,a互质)**
- 裴蜀定理：对任意两个整数a、b设d是它们的最大公约数。那么关于未知数x和y的线性丢番图方程（称为裴蜀等式）：$ax + by = m$
  有整数解（x,y）当且仅当m是d的倍数。裴蜀等式有解时必然有无穷多个解。

```
//ax + by = d，且|x|+|y|最小，其中d=gcd(a,b)
//即使a，b在int范围内，x和y也有可能超过int范围
void exgcd(ll a, ll b, ll &d, ll &x, ll &y)
{
    if (!b){ d = a; x = 1; y = 0;}
    else{ exgcd(b, a % b, d, y, x); y -= x * (a / b);}
}

//计算模n下a的逆。如果不存在逆，返回-1
//ax=1(mod n)
ll inv(ll a, ll n)
{
    ll d, x, y;
    exgcd(a, n, d, x, y);
    return d == 1 ? (x + n) % n : -1;
}
```

$BSGS$ 拔山盖世算法! ~ $A^x \equiv B\ (mod\ P)$

- 求解x $A^x \equiv B\ (mod\ P)$
- 小规模请直接暴力扫$0 \sim P-1$ 否则错！
- （$0 \sim P-1$）原因：
- 费马小定理：$A^x \equiv A (mod\ P) \rightarrow A^{x-1} \equiv 1 \equiv A^0 (mod\ P)$
- ps：原因好像不对

- $O(\sqrt{n}+logn)$(分块 + map)

```
typedef long long ll;
map <ll,ll> mp;
ll qp(ll a,ll k,ll p){
    ll res=1;
    ll base=a;
    while(k){
        if(k&1)res=res*base%p;
        base=base*base%p;
        k>>=1;
    }
    return res;
}

//-1 -> no solution
//小规模直接暴力 0 ~ P-1（费马小定理）
ll bsgs(ll A,ll B,ll P){
    if(A%P==0)return -1;
    mp.clear();

    ll m=ceil(sqrt(P));
    //这里最初是sqrt（p），但是如下文所述可以自己改...
    // int m = ceil(pow(p, 1.0 / 3)); 可修改，同思想可以换m大小范围（应该是），（处理大步找小步）
or （处理小步找大步）
    // 大步是pow(2/3)，所以小步的查找范围更大，但是大步枚举的次数更少

    ll tmp=B%P;
    for(ll i=0;i<=m;i++){
        mp[tmp]=i;//自己手打的hash可能不知道快多少!
        tmp=tmp*A%P;
    }
    ll t=qp(A,m,P);
    tmp=1;
    for(ll i=1;i<=m;i++){
        tmp=tmp*t%P;
        if(mp.count(tmp)){
            return ((i*m%P-mp[tmp])%P+P)%P;
        }
    }
    return -1;
}
```

comet oj瞎推$O(n)$计算所有差平方和

$$
\begin{aligned}
&\sum(w_i-w_x)^2\\
=&\sum(w_i^2-2w_iw_x+w_x^2)\\
=&\sum(w_i^2)-2w_x\sum w_i+n*w_x^2
\end{aligned}
$$

FFT（快速傅里叶变换）

- **FFT只能处理n为2的整数次幂的多项式**，不够整数次幂的全补上$0\rightarrow 0*x^{m-1}+0*x^m+...$,长度要是$2^n$
- 加速俩多项式相乘(**卷积/大数乘法算**$a*10^0+b*10^1+c*10^2+...$,(数组也就默认了10^n，乘完进位输出就好了))
  $O(n^2)\rightarrow O(nlogn)$

DFT（离散傅里叶变换）

- 复数（complex）一个特性
- $(a_1,\theta_1)*(a_2,\theta_2)=(a_1a_2,\theta_1+\theta_2)$ 模长相乘，极角相加
- 复数域坐标系下单位圆上的点平分：

$$\omega_n^k = \cos\frac{k}{n}2\pi + i\sin\frac{k}{n}2\pi$$

$$(\omega_n^1)^k = \omega_n^k$$

- 为了不去算x_i^n这里的$O(n)$我们选ω_n^i加速DFT进行分治成为了FFT...
- ω叫**单位根**了,特定性质:

1.$\omega_n^k = \omega_{2n}^{2k}$ $(\omega_n^k = \cos\frac{k}{n}2\pi + i\sin\frac{k}{n}2\pi = \cos\frac{2k}{2n}2\pi + i\sin\frac{2k}{2n}2\pi = \omega_{2n}^{2k})$

2.$\omega_n^{k+\frac{n}{2}} = -\omega_n^k$ $(\omega_n^{\frac{n}{2}} = \cos\frac{\frac{n}{2}}{n}2\pi + i\sin\frac{\frac{n}{2}}{n}2\pi = cos\pi + isin\pi = -1, e^{ix} = cosx + isinx, e^{i\pi} + 1 = 0)$

3.$\omega_n^0 = \omega_n^n = 1 = 1 + 0i$

FFT推导

- 将原本的DFT**递归分治**处理，推导如下：

$$A(x) = \sum_{i=0}^{n-1} a_i x^i = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1}x^{n-1}$$

按下标的奇偶性分割A(x)

$$A(x) = (a_0 + a_2 x^2 + \ldots + a_{n-2}x^{n-2}) + (a_1 x + a_3 x^3 + \ldots + a_{n-1}x^{n-1})$$

$$= (a_0 + a_2 x^2 + \ldots + a_{n-2}x^{n-2}) + x(a_1 + a_3 x^2 + \ldots + a_{n-2}x^{n-1})$$

由两者相似设：

$$A_1(x) = a_0 + a_2 x + a_4 x^2 \ldots + a_{n-2}x^{\frac{n}{2}-1}$$

$$A_2(x) = a_1 + a_3 x + a_5 x^2 \ldots + a_{n-1}x^{\frac{n}{2}-1}$$

$$A(x) = A_1(x^2) + xA_2(x^2)$$

再设$k < \frac{n}{2}$,将ω_n^k带入$A(x)$

$$A(\omega_n^k) = A_1((\omega_n^k)^2) + \omega_n^k A_2((\omega_n^k)^2)$$

$$= A_1(\omega_n^{2k}) + \omega_n^k A_2(\omega_n^{2k})$$

$$= A_1(\omega_{\frac{n}{2}}^k) + \omega_n^k A_2(\omega_{\frac{n}{2}}^k) \quad (\rightarrow \omega_n^k = \omega_{2n}^{2k})$$

再考虑带入$\omega_n^{k+\frac{n}{2}}$

$$A(\omega_n^{k+\frac{n}{2}}) = A_1((\omega_n^{k+\frac{n}{2}})^2) + \omega_n^{k+\frac{n}{2}} A_2((\omega_n^{k+\frac{n}{2}})^2)$$

$$= A_1(\omega_n^{2k+n}) + \omega_n^{k+\frac{n}{2}} A_2(\omega_n^{2k+n})$$

$$= A_1(\omega_n^{2k}\omega_n^n) - \omega_n^k A_2(\omega_n^{2k}\omega_n^n) \quad (\rightarrow \omega_n^{k+\frac{n}{2}} = -\omega_n^k)$$

$$= A_1(\omega_n^{2k}) - \omega_n^k A_2(\omega_n^{2k}) \quad (\rightarrow \omega_n^0 = \omega_n^n = 1 = 1 + 0i)$$

$$= A_1(\omega_{\frac{n}{2}}^k) - \omega_n^k A_2(\omega_{\frac{n}{2}}^k) \quad (\rightarrow \omega_n^k = \omega_{2n}^{2k})$$

发现仅有符号为+/-的差距，因此通过$A_1(\omega_{\frac{n}{2}}^k)$和$A_2(\omega_{\frac{n}{2}}^k)$的值可以得到$A(\omega_n^k)$和$A(\omega_n^{k+\frac{n}{2}})$的值（看好是A函数不是A1A2...$n/2 \rightarrow mid$)

IFFT（快速傅里叶逆变换）

- **一个多项式在分治的过程中乘上单位根的共轭复数，分治完的每一项除以n即为原多项式的每一项系数**
- 就是还是FFT函数就打个标记flag表示是该搞共轭（IFFT）还是原版的（FFT）

all代码

```
/*超朴素的递归版本FFT，容易爆。。*/
#include<complex>
#include<cmath>
#define cp complex<double>

const int maxn=6e4+50;
const double pi=acos(-1);
void FFT(cp *a,int n,int inv){//inv -> 取共轭 or not,a[]=>[0,n)
    if(n==1)return;
    int mid=n/2;
    static cp b[maxn];
    for(int i=0;i<=mid-1;i++)b[i]=a[i*2],b[i+mid]=a[i*2+1];//真实的拆分
    for(int i=0;i<=n-1;i++)a[i]=b[i];//真实再还原
    FFT(a,mid,inv);
    FFT(a+mid,mid,inv);
    for(int i=0;i<=mid-1;i++){
        cp x(cos(2*pi*i/n),inv*sin(2*pi*i/n));
        b[i]=a[i]+x*a[i+mid];
        b[i+mid]=a[i]-x*a[i+mid];
    }
    for(int i=0;i<=n-1;i++)a[i]=b[i];
}
cp ai[maxn],bi[maxn];

/*全新加入蝴蝶变化预处理的迭代版本FFT，求高精乘法*/
#include<iostream>
#include<cstdio>
#include<algorithm>
#include<cstring>
#include<cmath>
using namespace std;
const int maxn=150000+50;
const double pi=acos(-1);

struct node{
    double x,y;
    node(double xx=0,double yy=0){x=xx;y=yy;}
    node operator * (const node &r)const{return node(x*r.x-y*r.y,x*r.y+y*r.x);}
    node operator + (const node &r)const{return node(x+r.x,y+r.y);}
    node operator - (const node &r)const{return node(x-r.x,y-r.y);}
}ai[maxn],bi[maxn];

int R[maxn];

void FFT(node *J,int len,double flag){
    for(int i=0;i<len;i++){
        if(i<R[i])swap(J[i],J[R[i]]);
    }
    for(int slen=1;slen<len;slen<<=1){
        node T(cos(pi/slen),flag*sin(pi/slen));
        for(int lcur=0;lcur<len;lcur+=(slen<<1)){
            node t(1,0);
            for(int i=0;i<slen;i++,t=t*T){
                node Nx=J[lcur+i],Ny=t*J[lcur+slen+i];
                J[lcur+i]=Nx+Ny;
                J[lcur+slen+i]=Nx-Ny;
            }
        }
    }
}
```

```
char s1[maxn],s2[maxn];
int ans[maxn];
int main()
{
    int n;
    scanf("%d",&n);
    scanf("%s",s1);
    scanf("%s",s2);
    int t=0;
    for(int i=n-1;i>=0;i--)ai[t++].x=s1[i]-'0';t=0;
    for(int i=n-1;i>=0;i--)bi[t++].x=s2[i]-'0';

    int lim=1;
    while(lim<n+n)lim<<=1;
    int bit=0;
    while((1<<bit)<lim)bit++;
    for(int i=0;i<lim;i++)R[i]=(R[i>>1]>>1)|((i&1)<<(bit-1));

    FFT(ai,lim,1);
    FFT(bi,lim,1);
    for(int i=0;i<lim;i++)ai[i]=ai[i]*bi[i];
    FFT(ai,lim,-1);

    for(int i=0;i<=lim;i++){
        ans[i]+=(int)(ai[i].x/lim+0.5);
        if(ans[i]>=10){
            ans[i+1]+=ans[i]/10;
            ans[i]%=10;
            if(i==lim)lim++;
        }
    }

    while(ans[lim]==0&&lim>=1)lim--;
    for(int i=lim;i>=0;i--)printf("%d",ans[i]);
    puts("");
    return 0;
}
```

NTT前置——群+数论の知识

群

- 实际上群出现好多次了。。。

定义（群）设G为某种元素组成的一个非空集合，若在G内定义一个称为乘法的运算“·”，满足以下条件：
（1）（封闭性）$\forall a,b\in G$有$a\cdot b\in G$；

（2）（结合性）$\forall a,b,c\in G$，有$a\cdot(b\cdot c)=(a\cdot b)\cdot c$；

（3）在G中有一个元素e，对G中任意元素g，有e·g=g·e=g，元素e称为**单位元**(好像也叫**幺元/零元**来着，$-g$叫**负元**)；

（4）对G中任一元素g都存在G中的一个元素g'，使得g·g'=g'·g=e，g称为可逆元，g'称为g的**逆元**，记作g^{-1}；

则称**G关于“·”形成一个群（Group）**，记作（G，·），通常在不混淆的情况下省略“·”，用G来表示一个群，a·b也简记为ab

- **WARNING**：运算均是抽象运算，并不只是我们熟知的数之间的乘法和加法。非空集合G中的元素形式多样，可以是数、集合、矩阵等研究对象。
- 额外扩展定义：
  - **Abel群或交换群**：群中的运算满足交换律的群

- **半群**：非空集合G满足条件（1）和（2），则称G为半群
- **含幺半群**：非空集合G满足条件（1），（2）和（3），则称G为含幺半群
- **阶**：群G中元素的个数称为G的阶，记作$|G|$，若$|G|<+\infty$，则称群G为有限群；若$|G|>+\infty$，则称群G为无限群。(对群大小的*阶*)
- **子群**扩展包：
  - 设H为群G的**一**个非空子集，若H对G的运算"·"也构成群，**则称H为群G的子群**，记作H≤G
  - **平凡子群**：由子群的定义知，H={e}和H=G都是群G的子群，称之为群G的平凡子群。
  - **真子群**：如果H不是群G的平凡子群，即$e \subset H \subset G$，则称H为群G的真子群
  - **循环群**：群G由**一**个元素a生成时，称为循环群（Cyclic Group），记为G=<a>。
    - **循环群特性**：（1）循环群是Abel群，（2）若G=<a>,则|G|=ord(a)。
  - **生成元**：群中元素可以由最小数目个群元的**乘积生成**，这组群元称为该群的生成元，生成元的数目为有限群的秩。
- **定义/ord/阶（元素）**：设G为群，$a \in G$，则使$a^n = e$成立的最小正整数n称为元素a的阶，记作**ord(a)**。如果不存在这样的正整数n，则称元素a为无限阶元。（类似数论中的阶）

数论部分

- **原根**：
  - 在gcd(a,m)=1时，定义 a对模m的指数 δm(a)为使 $a^d \equiv 1(\mod m)$ 成立的最小的正整数 d。可知 δm(a) **一**定小于等于 φ (m)，若δm(a) = φ(m)，则称**a是模m的原根。**
  - **原根**另**一**种定义：假设**一**个数g对于P来说是原根，那么$g^i \mod P$的结果两两不同,且有 $1<g<P, 0<i<P$，那么g可以称为是P的**一**个原根
    - 原根例： 由于 $3^1 \equiv 3(\mod 7)$，$3^2 \equiv 2(\mod 7)$，$3^3 \equiv 6(\mod 7)$，$3^4 \equiv 4(\mod 7)$，$3^5 \equiv 5(\mod 7)$，$3^6 \equiv 1(\mod 7)$ ，所以 3 是模 7 的**一**个原根。
  - 性质：如果g是P的原根，**那么g的（1...P-1）次幂mod P的结果一定互不相同。**
  - 【求原根方法】
    对数P−1分解质因数得到不同的质因子$p_1, p_2, p_3, \ldots, p_n$，对于任何$2 \le a \le P-1$，判定a是否为P的原根，只需要检验$a^{\frac{P-1}{p_1}}, a^{\frac{P-1}{p_2}}, \ldots, a^{\frac{P-1}{p_n}}$ 这n个数中，**是否存在一个数mod P为1**，若存在，则a不是P的原根，否则a是P的原根。
  - 原根**一**般都是很小的数(100以内)。

同余式的性质

- (1)同余式可以**逐项相加。**
  若$a_1 \equiv b_1 \pmod m, a_2 \equiv b_2 \pmod m, \ldots, a_n \equiv b_n \pmod m$
  则$a_1 + a_2 + \ldots + a_n \equiv b_1 + b_2 + \ldots + b_n \pmod m$
- (2)同余式**一**边的数可以**移到另一边，只要改变符号**就可以了。
  若$a + b \equiv c \pmod m$，则$a \equiv c - b \pmod m$
- (3)同余式的**每一边都可以增加或减去模的任意倍数。**
  若$a \equiv b \pmod m$，则$a \mp km \equiv b \pmod m$
- (4)同余式可以**逐项相乘。**
  若$a_1 \equiv b_1 \pmod m, a_2 \equiv b_2 \pmod m, \ldots, a_n \equiv b_n \pmod m$
  则$a_1 a_2 \ldots a_n \equiv b_1 b_2 \ldots b_n \pmod m$
- (5)我们可以将**性质(1)(4)推广**成以下的情形。
  $(1) \rightarrow$ 若$A \equiv B \pmod m, x_1 \equiv y_1 \pmod m, x_2 \equiv y_2 \pmod m, \ldots, x_k \equiv y_k \pmod m$
  则$\sum A x_1^{a_1} x_2^{a_2} \ldots x_k^{a_k} \equiv \sum B y_1^{a_1} y_2^{a_2} \ldots y_k^{a_k} \pmod m$
  $(4) \rightarrow$ 若$a_0 \equiv b_0 \pmod m, a_1 \equiv b_1 \pmod m, \ldots, a_n \equiv b_n \pmod m$
  则$a_0 x^n + a_1 x^{n-1} + \ldots + a_n \equiv (b_0 x^n + b_1 x^{n-1} + \ldots + b_n) \pmod m$

- (6)同余式**两边的数如有公约数**，此**公约数又和模互素**，那么就**可以把两边的数除以这个公约数。**
  若$a \equiv b \pmod{m}$，且$a = a_1 d, b = b_1 d, gcd(d, m) = 1$
  则$a_1 \equiv b_1 \pmod{m}$
- (7)同余式**两边的数和模**可以**同时乘上一个整数。**
  若$a \equiv b \pmod{m}$，
  则$ak \equiv bk \pmod{km}$
- (8)同余式**两边的数和模**可以**同时被它们任一公约数除。**
  若$a \equiv b \pmod{m}$，且$a = a_1 d, b = b_1 d, m = m_1 d$
  则$a_1 \equiv b_1 \pmod{m_1}$
- (9)如果同余式**对于模m成立**，那么它**对于m的任意约数相等的模d也成立。**
  若$a \equiv b \pmod{m}$，$m = m_1 d$
  则$a \equiv b \pmod{d}$
- (10)如果同余式**一边上的数和模能被某个数除尽**，则同余式的**另一边的数也能被这个数除尽。**
  若$a \equiv b \pmod{m}$，$k|a, k|m$
  则$k|b$
- (11)同余式**一边上的数与模的最大公约数，等于另一边上的数与模的最大公约数。**
  若$a \equiv b \pmod{m}$，
  则$gcd(a, m) = gcd(b, m)$
- 不大相同的简略版：

1.**反身性**：a≡a (mod m)；
2.**对称性**：若a≡b(mod m)，则b≡a (mod m)；
3.**传递性**：若a≡b(mod m)，b≡c(mod m)，则a≡c(mod m)；
4.**同余式相加**：若a≡b(mod m)，c≡d(mod m)，则a c≡b d(mod m)；
5.**同余式相乘**：若a≡b(mod m)，c≡d(mod m)，则ac≡bd(mod m)。
6.**线性运算**：如果a ≡ b (mod m)，c ≡ d (mod m)，那么
(1)a ± c ≡ b ± d (mod m)；
(2)a * c ≡ b * d (mod m)。
7.**除法**：若$ac \equiv bc \pmod{m}$，$c \neq 0$,则$a \equiv b \pmod{m/gcd(c, m)}$，其中gcd(c,m)表示c和m的最大公约数，特殊地$gcd(c, m) = 1$则$a \equiv b \pmod{m}$；
8.**幂运算**：如果$a \equiv b \pmod{m}$，那么$a^n \equiv b^n \pmod{m}$；
9.若$a \equiv b \pmod{m}$，$n = m$，则$a \equiv b \pmod{n}$；
10.若$a \equiv b \pmod{m_i}$，$(i = 1, 2...n)$则，$a \equiv b \pmod{[m_1, m_2, \ldots, m_n]}$)其中$[m_1, m_2, \ldots m_n]$表示$m_1, m_2, \ldots m_n$的最小公倍数(lcm)。

NTT

- 迷之信息：其实这些模数的原根通常都是3
- 通常模数常见的有998244353,1004535809,469762049，这几个的原根都是3
- 大概就是...原根换单位根
- 要找到整数中模意义下的ω_n^k来加速DFT，这样没有精度误差、复数之类的东西了
  - 我们可以发现原根g拥有所有FFT所需的ω的性质，于是如果我们用$g^{\frac{P-1}{N}} \pmod{P}$来代替$\omega_n = e^{\frac{2\pi i}{N}}$，就能把复数对应成一个整数，在$\pmod{P}$意义下做快速变换了。
  - 可以证明：$\omega_n^k = g^{\frac{k \times (P-1)}{n}}$
- 单位根性质：
  1.$\omega_n^0, \ldots \omega_n^{n-1}$互不相同

  2.$\omega_n^0 = \omega_n^n = 1$

  3.$\omega_n^{k+\frac{n}{2}} = -\omega_n^k$，$\omega_{2n}^{2k} = \omega_n^k$

  4.$\omega_n^a \omega_n^b = \omega_n^{a+b}$

  5.$\sum_{i=1}^{n-1} \omega_n^i = 0 \quad (IDFT/IFFT$变化证明$)$
  - 证明原根具有相同性质：
    1.当$n \leq P - 1$时，根据定义可知$g_n^0, \ldots, g_n^{n-1}$互不相同，条件1成立。

2.根据欧拉定理：$g_n^0=1, g_n^n=g^{P-1}, g^{P-1}\equiv 1 \pmod P$。P为质数，这里的$\phi(P)=P-1$，条件2成立。

3.由上$g_n^n\equiv 1 \pmod P$，由**同余幂运算**，$g_n^{\frac{n}{2}} \pmod P$只能为1或-1，又因为$g_n^0,\dots,g_n^{n-1}$互不相同，得$g_n^{\frac{n}{2}}\equiv -1 \pmod P$，条件3成立。

4.带入显然幂运算指数相加+乘法分配律就是这里俩g_n乘一块...条件4成立。

5.由定义，$\sum_{i=1}^{n-1} g_n^i=\frac{g_n^1(1-g_n^1)}{1-g_n^1}=\frac{1-g_n^n}{1-g_n^1}=0$

- **所以只要替换$\omega_n^k=g^{\frac{k\times(P-1)}{n}}$就是NTT了！**
- 要求P是素数且N必须是$P-1$的因子，因为N是2^n，一般构造出P是$P=k*2^m+1$使$N|(P-1)$
- $1004535809=479*2^{21}+1 \quad g=3$
  $998244353=119*2^{23}+1 \quad g=3$

INTT（应该是逆变换吧...）

- 前边找的是$\omega_n^{-1},\omega_n^{-2},\dots,(\omega_n^{-1})^j$，是共轭复数。
- 那么这里就是$g_n^{-1},g_n^{-2}\dots$所以这是逆元喔
- **最后那里记得除以n变成乘以逆元**，第一份代码在NTT内完成了乘以逆元，注意n是长度（2^n）那个，不是简单的n...

修改的楼上FFT变NTT代码

```
#include<iostream>
#include<cstdio>
#include<algorithm>
#include<cstring>
#include<cmath>
using namespace std;
const int maxn=150000+50;
const long long P=998244353;
const long long g=3;
//这个变换只是借用P，g这样的数完成的多项式乘法，所以P，g构造使用就好没啥特殊，三模数NTT见
long long qp(long long a,long long k){
    long long res=1,base=a;
    while(k){
        if(k&1)res=res*base%P;
        base=base*base%P;
        k>>=1;
    }
    return res;
}

long long ai[maxn],bi[maxn];
int R[maxn];

void NTT(long long *J,int len,int flag){
    for(int i=0;i<len;i++){
        if(i<R[i])swap(J[i],J[R[i]]);
    }
    for(int slen=1;slen<len;slen<<=1){
        //node T(cos(pi/slen),flag*sin(pi/slen));
        long long T=qp(g,(P-1)/(slen<<1));
        for(int lcur=0;lcur<len;lcur+=(slen<<1)){
            long long t=1;
            for(int i=0;i<slen;i++,t=t*T%P){
                long long Nx=J[lcur+i],Ny=t*J[lcur+slen+i]%P;
                J[lcur+i]=(Nx+Ny)%P;
                J[lcur+slen+i]=(Nx-Ny+P)%P;
            }
        }
    }
```

```
    if(flag==1)return;
    long long inv=qp(len,P-2);reverse(J+1,J+len);
    for(int i=0;i<len;i++)J[i]=J[i]*inv%P;//不是很懂reverse
}

char s1[maxn],s2[maxn];
int ans[maxn];
int main()
{
    int n;
    scanf("%d",&n);
    scanf("%s",s1);
    scanf("%s",s2);
    int t=0;
    for(int i=n-1;i>=0;i--)ai[t++]=s1[i]-'0';t=0;
    for(int i=n-1;i>=0;i--)bi[t++]=s2[i]-'0';

    int lim=1;
    while(lim<n+n)lim<<=1;
    int bit=0;
    while((1<<bit)<lim)bit++;
    for(int i=0;i<lim;i++)R[i]=(R[i>>1]>>1)|((i&1)<<(bit-1));

    NTT(ai,lim,1);
    NTT(bi,lim,1);
    for(int i=0;i<lim;i++)ai[i]=ai[i]*bi[i];
    NTT(ai,lim,-1);

    for(int i=0;i<=lim;i++){
        ans[i]+=ai[i];
        if(ans[i]>=10){
            ans[i+1]+=ans[i]/10;
            ans[i]%=10;
            if(i==lim)lim++;
        }
    }

    while(ans[lim]==0&&lim>=1)lim--;
    for(int i=lim;i>=0;i--)printf("%d",ans[i]);
    puts("");
    return 0;
}

//我看的懂的自己写懂得NTT+INTT
#include<iostream>
#include<cstdio>
#include<algorithm>
#include<cstring>
#include<cmath>
using namespace std;
const int maxn=150000+50;
const long long P=998244353;
const long long g=3;

long long qp(long long a,long long k){
    long long res=1,base=a;
    while(k){
        if(k&1)res=res*base%P;
        base=base*base%P;
        k>>=1;
    }
    return res;
}
```

```
long long ai[maxn],bi[maxn];
int R[maxn];

void NTT(long long *J,int len,int flag){
    for(int i=0;i<len;i++){
        if(i<R[i])swap(J[i],J[R[i]]);
    }
    for(int slen=1;slen<len;slen<<=1){
        //node T(cos(pi/slen),flag*sin(pi/slen));
        long long T=qp(flag==1 ? g : qp(g,P-2),(P-1)/(slen<<1));
        for(int lcur=0;lcur<len;lcur+=(slen<<1)){
            long long t=1;
            for(int i=0;i<slen;i++,t=t*T%P){
                long long Nx=J[lcur+i],Ny=t*J[lcur+slen+i]%P;
                J[lcur+i]=(Nx+Ny)%P;
                J[lcur+slen+i]=(Nx-Ny+P)%P;
            }
        }
    }
}

char s1[maxn],s2[maxn];
int ans[maxn];
int main()
{
    int n;
    scanf("%d",&n);
    scanf("%s",s1);
    scanf("%s",s2);
    int t=0;
    for(int i=n-1;i>=0;i--)ai[t++]=s1[i]-'0';t=0;
    for(int i=n-1;i>=0;i--)bi[t++]=s2[i]-'0';

    int lim=1;
    while(lim<n+n)lim<<=1;
    int bit=0;
    while((1<<bit)<lim)bit++;
    for(int i=0;i<lim;i++)R[i]=(R[i>>1]>>1)|((i&1)<<(bit-1));

    NTT(ai,lim,1);
    NTT(bi,lim,1);
    for(int i=0;i<lim;i++)ai[i]=ai[i]*bi[i];
    NTT(ai,lim,-1);

    long long invlim=qp(lim,P-2);
    for(int i=0;i<=lim;i++){
        ans[i]+=(ai[i]*invlim)%P;//除以n变乘逆元
        if(ans[i]>=10){
            ans[i+1]+=ans[i]/10;
            ans[i]%=10;
            if(i==lim)lim++;
        }
    }

    while(ans[lim]==0&&lim>=1)lim--;
    for(int i=lim;i>=0;i--)printf("%d",ans[i]);
    puts("");
    return 0;
}
```

多项式全套操作（加减乘除+求逆+牛顿迭代+带余除法+Ln+Exp+快速幂）

- '+'，'-'自己瞎写吧....
- '*'，在上边那个NTT/FFT都行...

多项式除法 & 求逆

- 类比同余系下数字的逆元,只要求多项式的逆元,就可以把除法转换成乘法了。
- 倍增求逆：...我放弃...

MTT & 分治FFT

BM

FWT

莫比乌斯反演/狄利克雷卷积

杜教筛

二次剩余

Lucas定理 & EXLucas

CRT & EXCRT

- 处理同余方程**一片**，求x

$$x \equiv a_1 \pmod{m_1}$$
$$x \equiv a_2 \pmod{m_2}$$
.
.
.$$x \equiv a_n \pmod{m_n}$$

- CRT(中国剩余定理)：处理楼上那个问题，前置是那堆**m俩俩互质**
- 处理方法是搞**所有m的lcm**，
  然后对**一个式子算**$\frac{lcm}{m_i}t_i \equiv 1 \pmod{m_i}$这里$t_i$用**exgcd**算得
  这个式子两边同时乘上a_i，左边就是**一个**x**的解值**啦，类似：$x_i \equiv a_i \frac{lcm}{m_i}t_i \equiv a_i \pmod{m_i}$
  然后把所有这样得到的x**加和**就好啦!(因为其他的lcm同余全是0)
- 也就是：$x = \sum_{i=1}^{k} a_i \frac{lcm}{m_i}t_i, (t_i \rightarrow exgcd)$
- 板子crt：**看清是哪个余数哪个模数**

```
#include<cstdio>
#include<algorithm>
#include<iostream>
using namespace std;
typedef long long ll;
const int maxn=10+10;
ll a[maxn],b[maxn];
ll exgcd(ll a,ll b,ll &x,ll &y){
    if(b==0){
        x=1,y=0;
        return a;
    }
    ll ret=exgcd(b,a%b,y,x);
    y-=(a/b)*x;
    return ret;
}
ll mul(ll a,ll b,ll mod)
{
    ll res=0;
    while(b>0)
```

```
    {
        if(b&1) res=(res+a)%mod;
        a=(a+a)%mod;
        b>>=1;
    }
    return res;
}
int k;
ll crt(){
    ll x,y,res=0,lcm=1,tmp;
    for(int i=1;i<=k;i++)lcm*=b[i];
    for(int i=1;i<=k;i++){
        tmp=lcm/b[i];
        exgcd(b[i],tmp,x,y);
        y=(y%b[i]+b[i])%b[i];
        res=(res+mul(mul(y,tmp,lcm),(a[i]%lcm+lcm)%lcm,lcm))%lcm;
    }//a可能负数
    return (lcm+res)%lcm;
}
int main()
{
    scanf("%d",&k);
    for(int i=1;i<=k;i++)scanf("%lld",&a[i]);//余数a
    for(int i=1;i<=k;i++)scanf("%lld",&b[i]);//模数b
    printf("%lld\n",crt());
    return 0;
}
```

- exCRT(扩展版本)：这次**不用m俩俩互质**啦！（指不定啥关系）
- 板子excrt：
  处理那堆m不互质的，return -1就是无解，一个一个exgcd往上怼就好了hhhhhh
- 解决ing：**只考虑两个**怎么处理？
  可以得到：$x = a_1 + k_1 * m_1; x = a_2 + k_2 * m_2$
  所以 $a_1 + k_1 * m_1 = a_2 + k_2 * m_2$
  $k_2 * m_2 - k_1 * m_1 = a_1 - a_2 \quad \rightarrow exgcd_{解}\, k,\ _{反推}\, x$
  注意细节上**爆longlong和模负数、(+mod)%mod**之类的事情，二合一再继续上边的操作
  看清是哪个余数哪个模数

```
#include<cstdio>
#include<algorithm>
#include<iostream>
using namespace std;
typedef long long ll;
//太沙雕的出题人可以考虑上边用下边
//using ll=__int128;
const int maxn=100+10;
ll a[maxn],r[maxn];
ll exgcd(ll a,ll b,ll &x,ll &y){
    if(b==0){
        x=1,y=0;
        return a;
    }
    ll ret=exgcd(b,a%b,y,x);
    y-=(a/b)*x;
    return ret;
}
int n;
ll mul(ll a,ll b,ll mod)
{
    ll res=0;
    while(b>0)
```

```
    {
        if(b&1) res=(res+a)%mod;
        a=(a+a)%mod;
        b>>=1;
    }
    return res;
}
ll excrt(){
    ll M=a[1],R=r[1],x,y,d;//模a,余数r
    for(int i=2;i<=n;i++){
        d=exgcd(M,a[i],x,y);
        ll tmp=((r[i]-R%a[i]+a[i])%a[i]);
        if(tmp%d) return -1;
        //x=(R-r[i])/d*x%a[i];
        tmp/=d;
        x=mul(x,tmp,a[i]);
        R+=M*x;
        M=a[i]/d*M;
        R=(R%M+M)%M;
    }
    return (R%M+M)%M;
}
int main()
{
    ll m;
    scanf("%d%lld",&n,&m);
    for(int i=1;i<=n;i++)scanf("%lld%lld",&a[i],&r[i]);//模a,余数r
    long long ans=excrt();
    if(ans==-1)puts("he was definitely lying");
    else if(ans>m)puts("he was probably lying");
    else printf("%lld\n",ans);
    return 0;
}
```

上边那个**容易炸long long*long long**
所以写了**快速乘取模mul**
但是日常还是炸，建议__**int128**
或者下边的python3/java不会qwq

```
def egcd(a, b):
    """扩展欧几里得"""
    if 0 == b:
        return 1, 0, a
    x, y, q = egcd(b, a % b)
    x, y = y, (x - a // b * y)
    return x, y, q


s = input()
n, m = int(s.split()[0]), int(s.split()[1])
# print(n, m)
M = 0
R = 0
x = 0
y = 0
d = 0
ok = 1
for i in range(0, n):
    if i == 0:
        s = input()
        M, R = int(s.split()[0]), int(s.split()[1])
    else:
```

```
        s = input()
        ui, ri = int(s.split()[0]), int(s.split()[1])
        x, y, d = egcd(M, ui)
        if (R - ri) % d != 0:
            ok = 0
        x = (R - ri) // d * x % ui
        R -= M * x
        M = ui // d * M
        R = R % M

if ok == 1:
    if R > m:
        print("he was probably lying")
    else:
        print(R)
else:
    print("he was definitely lying")
```

拉格朗日插值法vs快速插值法

因式定理 & 余数定理

- **因式定理**:
  “设f(x)为一多项式，则x-a为f(x)的因式”等价于f(a)=0。
  (大概就是感觉能拆个(x-a)出来变成(x-a)*f'(x)=f(x)然后(x-a)=0)
- 推广:
  “ax-b为f(x)的因式”等价于$f(\frac{b}{a})=0$。
- **余数定理**:
  当一个多项式f(x)除以(x – a)时，所得的余数等于 f(a)
  整系数多项式f(x)除以(x-a)商为q(x)，余式为r，则$f(x)=(x-a)q(x)+r$.
  如果多项式r=0，那么多项式f(x)必定含有因式(x-a)。反过来，如果f(x)含有因式(x-a)，那么，r=0。
- 余式定理的推论
  当一个多项式 f(x) 除以 (mx – n) 时，所得的余数等于 f(n/m)

拉格朗日插值法

- 解决的问题:给你 n 个不同的点值(x_i,y_i),让你求出一个低于 n 次的多项式满足经过上述 n 个点。【就是点值转系数表示...】
- FFT/NTT用了单位根/原根，这里可不是lxxx...（逆xxx..）
- 拉格朗日的想法:
  1.对每个(x_i,y_i)构造$f_i(x_i)=y_i,f_i(x_k)=0\ [k\neq i]$,然后把他们加起来就好啦~
  2.先构造第二块都是零的：$G=\prod_{k=1}^{n}[k\neq i](x-x_k)$
  3.再来构造一波第一个$f(x_i)=y_i$：把x_i带入G可得到$G(x_i)=\prod_{k=1}^{n}[k\neq i](x_i-x_k)$就是常数了
  4.综合2&3得到可以设$f_i(x)=\frac{y_iG}{G(x_i)},(满足\ f_i(x_i)=y_i,f_i(x_k)=0[k\neq i])$
  5.再通过4来综合$O(n^2)$时间转换：$F=\sum_{i=1}^{n}(y_i\prod_{k=1,k\neq i}^{n}\frac{x-x_k}{x_i-x_k})$

```
//给定n个点，请你确定这个多项式，并将k代入求值
//求出的值对998244353取模
#include<iostream>
#include<cstdio>
#include<algorithm>
using namespace std;
const int maxn=2000+5;
const long long mod=998244353;
struct node{
    long long x,y;
}nds[maxn];
long long qp(long long a,long long k){
```

```
    long long res=1,base=a;
    while(k){
        if(k&1)res=res*base%mod;
        base=base*base%mod;
        k>>=1;
    }
    return res;
}
int main()
{
    int n;
    long long k;
    scanf("%d%lld",&n,&k);
    long long ans=0;
    for(int i=1;i<=n;i++){
        scanf("%lld%lld",&nds[i].x,&nds[i].y);
    }
    for(int i=1;i<=n;i++){
        long long tans=nds[i].y;
        for(int j=1;j<=n;j++){
            if(i==j)continue;
            tans=((tans*(k-nds[j].x+mod)%mod)*qp(nds[i].x-nds[j].x,mod-2)+mod)%mod;
        }
        ans=(ans+tans+mod)%mod;
    }
    printf("%lld\n",ans);
    return 0;
}
```

简单的高斯消元板板

- 复杂度$O(n^3)$

```
const double eps=1e-8;
double A[maxn][maxn];
double B[maxn];
double ans[maxn];
//A*ans=B
void Gauss(){
    int num;
    for(int i=1;i<n;i++){//i<n!!!! 没有 =
        num=i;
        for(int j=i+1;j<=n;j++){
            if(abs(A[j][i])>abs(A[num][i]))num=j;
        }//暴力找最小

        for(int j=1;j<=n;j++)swap(A[i][j],A[num][j]);
        swap(B[i],B[num]);//暴力交换

        for(int j=i+1;j<=n;j++){
            if(abs(A[j][i])<=eps)continue;
            double t=A[j][i]/A[i][i];// A[j][i] !!!
            for(int k=1;k<=n;k++){
                A[j][k]-=A[i][k]*t;
            }
            B[j]-=B[i]*t;
        }//暴力消到0/上三角矩阵
    }

    for(int i=n;i>=1;i--){
        ans[i]=B[i]/A[i][i];
        for(int j=i;j>=1;j--){
```

```
            B[j]-=A[j][i]*ans[i];
        }
    }//反向向上得到解
    return;
}
```

模意义下的高斯消元板板（和最前边的逆元联动）

- 注意得有逆元可取...

```
const long long md=1e9+7;
long long A[maxn][maxn];
long long B[maxn];
//double ans[maxn];
long long p[maxn];//这个p就是ans啦
//A*ans=B
int k;
using ll=long long;
void exgcd(ll a, ll b, ll &d, ll &x, ll &y)
{
    if (!b){ d = a; x = 1; y = 0;}
    else{ exgcd(b, a % b, d, y, x); y -= x * (a / b);}
}

//ax=1(mod n)
ll inv(ll a, ll n)
{
    ll d, x, y;
    exgcd(a, n, d, x, y);
    return d == 1 ? (x + n) % n : -1;
}//前方联动inv
void Gauss(){
    int num;
    int n=k;
    for(int i=1;i<n;i++){//i<n!!!! 没有 =
        num=i;
        for(int j=i+1;j<=n;j++){
            if(abs(A[j][i])>abs(A[num][i]))num=j;
        }//暴力找最小

        for(int j=1;j<=n;j++)swap(A[i][j],A[num][j]);
        swap(B[i],B[num]);//暴力交换

        for(int j=i+1;j<=n;j++){
            //if(abs(A[j][i])<=eps)continue;
            if(A[j][i]==0)continue;
            //double t=A[j][i]/A[i][i];// A[j][i] !!!
            long long t=(A[j][i]*inv(A[i][i],md)%md+md)%md;
            for(int k=1;k<=n;k++){
                A[j][k]=(A[j][k]-A[i][k]*t%md+md)%md;
            }
            B[j]=(B[j]-B[i]*t%md+md)%md;
        }//暴力消到0/上三角矩阵
    }

    for(int i=n;i>=1;i--){
        //ans[i]=B[i]/A[i][i];
        p[i]=(B[i]*inv(A[i][i],md)%md+md)%md;
        for(int j=i;j>=1;j--){
            B[j]=(B[j]-A[j][i]*p[i]%md+md)%md;
        }
    }//反向向上得到解
```

```
    return;
}
```

一些数据结构

线段树各种瞎写

```
//这里是最大/最小连续字段和，查询返回node
struct ST{
    #define ls i<<1
    #define rs i<<1|1

    struct node{
        int l,r;
        long long sum;
        long long lmxs,rmxs;//,maxsum;
        long long lmis,rmis;//,minsum;
    }T[maxn<<2],ans;

    void pushup(int i){
        T[i].sum=T[ls].sum+T[rs].sum;

        //T[i].maxsum=max(T[ls].rmxs+T[rs].lmxs,max(T[ls].maxsum,T[rs].maxsum));
        T[i].lmxs=max(T[ls].sum+T[rs].lmxs,T[ls].lmxs);
        T[i].rmxs=max(T[rs].sum+T[ls].rmxs,T[rs].rmxs);

        //T[i].minsum=min(T[ls].rmis+T[rs].lmis,min(T[ls].minsum,T[rs].minsum));
        T[i].lmis=min(T[ls].sum+T[rs].lmis,T[ls].lmis);
        T[i].rmis=min(T[rs].sum+T[ls].rmis,T[rs].rmis);
    }

    void build(int i,int l,int r){
        T[i].l=l;T[i].r=r;
        if(l==r){
            T[i].lmis=T[i].rmis=b[l];//T[i].minsum=b[l];
            T[i].lmxs=T[i].rmxs=b[l];//T[i].maxsum=b[l];
            T[i].sum=b[l];
            return;
        }
        int mid=(l+r)>>1;
        build(ls,l,mid);
        build(rs,mid+1,r);
        pushup(i);
    }

    node query_max(int i,int nl,int nr){
        if(nl<=T[i].l&&T[i].r<=nr)return T[i];
        int mid=(T[i].l+T[i].r)>>1;
        if(nr<=mid)return query_max(ls,nl,nr);
        if(mid<nl)return query_max(rs,nl,nr);
        node lo=query_max(ls,nl,nr),ro=query_max(rs,nl,nr),ans;

        ans.sum=lo.sum+ro.sum;
        //ans.maxsum=max(lo.rmxs+ro.lmxs,max(lo.maxsum,ro.maxsum));
        ans.lmxs=max(lo.sum+ro.lmxs,lo.lmxs);
        ans.rmxs=max(ro.sum+lo.rmxs,ro.rmxs);
        return ans;
    }

    node query_min(int i,int nl,int nr){
        if(nl<=T[i].l&&T[i].r<=nr)return T[i];
```

```
        int mid=(T[i].l+T[i].r)>>1;
        if(nr<=mid)return query_min(ls,nl,nr);
        if(mid<nl)return query_min(rs,nl,nr);
        node lo=query_min(ls,nl,nr),ro=query_min(rs,nl,nr),ans;

        ans.sum=lo.sum+ro.sum;
        //ans.minsum=min(lo.rmis+ro.lmis,min(lo.minsum,ro.minsum));
        ans.lmis=min(lo.sum+ro.lmis,lo.lmis);
        ans.rmis=min(ro.sum+lo.rmis,ro.rmis);
        return ans;
    }
}seg;
```

一些日常的dp

LIS (最长上升子序列) $O(nlogn)$

```
//最简谱的LIS
int d[maxn];
int lis(int c[],int lenc){
    d[1]=c[1];
    int len=1;
    for(int i=2;i<=lenc;i++){
        if(c[i]>d[len]){
            d[++len]=c[i];
        }
        else {
            int pos=lower_bound(d+1,d+1+len,c[i])-d;
            d[pos]=c[i];
        }
    }
    return len;
}

//能够记录pre/father/from前驱的，vis顺手还标记了一下
//这里的d记录了下标而不是数值，所以二分自己手写吧。。。
//里边的二分是[ll,rr)这样的，正好ll+1是要修改的，注意ll=0是起始
int d[maxn];
bool vis[maxn];
int from[maxn];
int lis_where(int c[],int lenc){
    memset(vis,0,sizeof(vis));
    memset(from,0,sizeof(from));

    int u=1;
    while(c[u]==-1)u++;
    d[1]=u;
    //可能的修改d[1]=1;
    int len=1;

    for(int i=u+1;i<=lenc;i++){
        if(c[i]==-1)continue;
        //这个是那道题会删除数组的数，所以我置换成-1表示删掉了。
        //如果没有删除而且还有可能有负数存在的话，请务必修改这里!
        //（上边那个while和u的存在也是这个原因）

        if(c[i]>c[d[len]]){
            d[++len]=i;
            from[i]=d[len-1];
        }
        else {
```

```
            //int pos=lower_bound(d+1,d+1+len,c[i])-d;
            int ll=0,rr=len+1,mid;
            while(ll+1<rr){//[ll,rr)
                mid=(ll+rr)>>1;
                if(c[d[mid]]<c[i])ll=mid;
                else rr=mid;
            }
            d[ll+1]=i;
            from[i]=d[ll];
        }
    }

    int cur=d[len];
    while(cur>0)vis[cur]=1,cur=from[cur];
    //如果不用标记似乎可以去掉哟
    return len;
}
```

没思路的时候看看这些沙雕想法

- 不管出题人怎么神仙也不能跳过的步骤应该就不会超时
- 如果我是出题人，那么我想考的点应该是？
- 打表找规律猜公式瞎搞
- 如果数据莫名要不要试试打表啊~可能答案没数据范围那么大
- 没事打个暴力吧，搞不好发现能奇怪技巧（拉格朗日插值啥的）加速暴力时间就是正解哇
- 如果二分的话，考虑一下左边搞一搞，右边检测可能就复杂度降下来了呢$O(2^n) \rightarrow O(2*2^{\frac{n}{2}})$
- 没法做？经过若干次尝试和总结后...会发现哒！
- 热爱生活，别怕烦躁哟da☆ze~

卡---------卡----------

卡常的方法分为以下几类：

0.硬件优化：浮点加速器，CPU等等。

1.编译优化：编译开关，inline，register 之类的。

2.封装优化：把代码的封装拆开/循环展开，封装得过死会使效率降低(stl)。

3.编写优化：细节优化，手写栈之类。

4.时空互易： 时间换空间以达到时空均衡(常数层面上)。

胡写的板子part2 —— 三倍Ice☆Creeeeeeeeeeeam！！！~

更厉害的数据结构

仍然字符串 —— SA后缀数组

- 和后缀自动机一起玩耍——
- 基数排序比较神奇，具体内容见板子1...
- 除了长度外**字符集大小**有影响，有个地方开桶的大小要考虑这个的！所以离散化~！

李超线段树

- 动态维护一个平面直角坐标系，可以在中间插入一条线段，回答询问在与x=x0这条直线相交的所有线段中，交点的y轴坐标的最大(小)值。
- 就像一个..凸.包..？（不准确）
- 开一棵线段树，标记下放
- 时间复杂度：
  修改部分：我们每次把一条线段的值域分隔到O(logn)个区间，每个区间最多把标记下传O(logn)层，因此修改的时间复杂度为O(log2n)。（但实测常数很小）
  查询部分：每次在线段树上从上到下扫一遍，时间复杂度为O(logn)

```
#include<iostream>
#include<cstdio>
#include<cstring>
#include<algorithm>
#include<cmath>
using namespace std;
const int maxn=1e6+5;
/*inline void read(int &n){
    char c='+';int x=0;bool flag=0;
    while(c<'0'||c>'9'){c=getchar();if(c=='-')flag=1;}
    while(c>='0'&&c<='9'){x=x*10+c-'0';c=getchar();}
    n=flag==1?-x:x;
}*/
struct f{
    double k,b;
}a[maxn];//直线...
int tree[maxn];//线段树...
int tot=0;//线段树计数...

double ans=0;
bool pd(int now,int pre,int day){
    day--;
    return a[now].k*day+a[now].b>a[pre].k*day+a[pre].b;
}//day这一天对应的y值（y=k*x+b），第1天是b，第二天是1*k+b，所以day要减一

#define ls id<<1
#define rs id<<1|1

void insert(int id,int l,int r,int now){
```

```
    if(l==r){
        if(pd(now,tree[id],l))tree[id]=now;//如果叶子节点，当前直线在之前直线的“上边”就直接修改
        return;
    }
    int mid=(l+r)>>1;
    //下边都是下放，如果当前斜率更大，那么在交点左侧（负半轴方向）的直线是原来的直线在“上边”，反之同理
    //下放那个在当前不占优势的直线
    if(a[now].k>a[tree[id]].k){
        if((pd(now,tree[id],mid)))insert(ls,l,mid,tree[id]),tree[id]=now;
        else insert(rs,mid+1,r,now);
    }
    else {
        if((pd(now,tree[id],mid)))insert(rs,mid+1,r,tree[id]),tree[id]=now;
        else insert(ls,l,mid,now);
    }
}

void query(int id,int l,int r,int now){//now就是：直线里的x
    ans=max(ans,a[tree[id]].k*(now-1)+a[tree[id]].b);
    if(l==r)return;
    int mid=(l+r)>>1;
    if(now<=mid)query(ls,l,mid,now);//可能直线都来一遍，上边在上的不一定比下放的更优
    else query(rs,mid+1,r,now);
}

int main()
{
    ios::sync_with_stdio(false);
    int n;
    cin>>n;
    for(int i=1;i<=n;i++){
        string opt;
        cin>>opt;
        if(opt[0]=='P'){
            ++tot;
            cin>>a[tot].b>>a[tot].k;
            insert(1,1,n,tot);
        }
        else {
            int p;
            cin>>p;
            ans=0;
            query(1,1,n,p);
            printf("%d\n",(int)(ans/100));//那个题规定到百元才÷100..
        }
    }
    return 0;
}
```

主席树+动态主席树

主席树

- 别管名字多花里胡哨
- 能支持查询区间l~r的第k大！思想是前缀和！

- 考虑每次插入一个数就造一棵线段树，线段树上点不是存代表线段l~r区间信息，而是数值在那个l_num~r_num区间的**数字出现的个数**
  这里的数值要离散化哦！
- 如果这样的话，询问数组区间l~r就是两颗线段树（l-1和r两颗）对应点的前缀和做差会得到数值在l_num~r_num在区间l~r出现的次数
- 然后两边分开走就好啦（左边个数比k多走左边k=k，左边个数比k少走右边k=k-左个数），下边是板子
- 这里主要要解决问题是如果真的都分开开线段树会炸空间，解决方案是重用之前的点
  （多加一个数最多修改一条链logn个点，只要新开logn个点）

```
#include<iostream>
#include<cstdio>
#include<cstring>
#include<algorithm>
using namespace std;
const int maxn=2e5+5;
struct node{
    int l,r,sum;
}T[maxn*20];//总每个树节点记录
int cnt=0;//节点数计数
int a[maxn];//原数组
int h[maxn];//离散化用
int rot[maxn];//第i个树的根记录

void build(int &rt,int l,int r){
    rt=++cnt;
    T[rt].sum=0;
    if(l==r)return;
    int mid=(l+r)>>1;
    build(T[rt].l,l,mid);
    build(T[rt].r,mid+1,r);
}//先造个第0棵树--

void update(int &rt,int prert,int l,int r,int num,int val){//这里写的就是rt从prert
继承的东西
    rt=++cnt;
    T[rt].sum=T[prert].sum+val;
    T[rt].l=T[prert].l;
    T[rt].r=T[prert].r;//记着左右子树也继承过来，如果改会在下边改的
    if(l==r)return;
    int mid=(l+r)>>1;//左改 or 右改
    if(num<=mid)update(T[rt].l,T[prert].l,l,mid,num,val);
    else update(T[rt].r,T[prert].r,mid+1,r,num,val);
}

int query(int rtl,int rtr,int l,int r,int k){//rtl，rtr传根；l，r就是值...
    if(l==r)return l;
    int cutcut=T[T[rtr].l].sum-T[T[rtl].l].sum;//判左值
    int mid=(l+r)>>1;
    if(cutcut>=k)return query(T[rtl].l,T[rtr].l,l,mid,k);
    else return query(T[rtl].r,T[rtr].r,mid+1,r,k-cutcut);//记着写k-cutcut
}

int main()
{
    int n,m;
```

```
    scanf("%d%d",&n,&m);

    for(int i=1;i<=n;i++)scanf("%d",&a[i]),h[i]=a[i];
    sort(h+1,h+1+n);
    int num=unique(h+1,h+1+n)-(h+1);//离散化

    build(rot[0],1,num);
    for(int i=1;i<=n;i++){
        a[i]=lower_bound(h+1,h+1+num,a[i])-h;//离散化
    //  cout<<a[i]<<endl;
        update(rot[i],rot[i-1],1,num,a[i],1);//更新造新树
    }

    for(int i=1;i<=m;i++){
        int l,r,k;
        scanf("%d%d%d",&l,&r,&k);
        printf("%d\n",h[query(rot[l-1],rot[r],1,num,k)]);//查询输出原数，query返回
序号，h[]还原原数字
    }

    return 0;
}
```

动态主席树

- 差不多可以叫树状数组套线段树了ovo，实际上已经和主席树略偏离了，继承前一个消失，前缀思想还在。
- 如果我们修改一个点（位置是pos）的话，就相当于第pos棵树**到**最后一棵树都统一修改那个数值所在链，暴力修改大概O(nlogn)是不行的
- 如果说修改是从（位置还是pos）数字a改变成b，就是pos到最后树都减a（修改链）然后加b（修改链），
- 那我们把这些操作放到树状数组里就是O(logn*logn)!
- 仅log个修改，同时与之对应，前缀和的O(1)查询就会变成O(logn)，也就是用了log棵树做差
- 不用想太复杂的，就是一个数组单点修改区间查询，只不过数组每个空存的是线段树，而这个数组又恰好是树状数组哦！
- 存一个数字修改就改个数字，那是因为他的修改方式就是改字。但是线段树，就需要按修改线段树的方法修改啦
- 看板子叭！

```
#include<iostream>
#include<cstdio>
#include<cstring>
#include<algorithm>
using namespace std;
const int maxn=2e5+5;
struct node {
    int l,r,sum;
} T[maxn*400];//蜜汁400倍...小了就re，这计算...
int cnt=0;
int a[maxn];
int h[maxn];
int rot[maxn];

struct opts {
    int a,b,c;
    bool is_q;
```

```
} op[maxn];
char opop[5];
int num;
int many[2];
int log_go[2][30];

void update(int &rt,int l,int r,int pos,int val) {
    if(!rt)rt=++cnt;//判点
    T[rt].sum+=val;
    if(l==r)return;
    int mid=(l+r)>>1;
    if(pos<=mid)update(T[rt].l,l,mid,pos,val);
    else update(T[rt].r,mid+1,r,pos,val);
}//削弱继承机制，因为是树套树，现场造点

int query(int l,int r,int k){
    if(l==r)return l;
    int mid=(l+r)>>1;
    int sum=0;
    for(int i=1;i<=many[1];i++)sum+=T[T[log_go[1][i]].l].sum;
    for(int i=1;i<=many[0];i++)sum-=T[T[log_go[0][i]].l].sum;
    //上边是按log棵树计算sum，作比较，还是左儿子的sum和k，然后下边就是log棵树同时跳然后递归
    if(k<=sum){
        for(int i=1;i<=many[1];i++)log_go[1][i]=T[log_go[1][i]].l;
        for(int i=1;i<=many[0];i++)log_go[0][i]=T[log_go[0][i]].l;
        return query(l,mid,k);
    }
    else {
        for(int i=1;i<=many[1];i++)log_go[1][i]=T[log_go[1][i]].r;
        for(int i=1;i<=many[0];i++)log_go[0][i]=T[log_go[0][i]].r;
        return query(mid+1,r,k-sum);
    }
}

int c[maxn];
int lowbit(int x){return x&(-x);}

void log_add(int x,int val){//按树状数组来，因此继承机制退去
    int k=lower_bound(h+1,h+1+num,a[x])-h;//这里按a[x]写的，所以下边的修改要记得改a
    for(int i=x;i<=num;i+=lowbit(i))update(rot[i],1,num,k,val);//log更新
}

int log_query(int l,int r,int k){
    memset(log_go,0,sizeof(log_go));//感觉可以不写...亲测也可以不写...
    many[0]=many[1]=0;//赋值0个元素
    for(int i=r;i;i-=lowbit(i))log_go[1][++many[1]]=rot[i];//先加，后边都是1~many，
跳log个树
    for(int i=l-1;i;i-=lowbit(i))log_go[0][++many[0]]=rot[i];
    return query(1,num,k);
}

int main() {
    int n,m;
    scanf("%d%d",&n,&m);

    for(int i=1; i<=n; i++)scanf("%d",&a[i]),h[i]=a[i];
    num=n;
    for(int i=1; i<=m; i++) {
```

```
        scanf("%s",opop);
        if(opop[0]=='Q') {
            scanf("%d%d%d",&op[i].a,&op[i].b,&op[i].c);
            op[i].is_q=1;//判断是不是query操作
        }
        else {
            scanf("%d%d",&op[i].a,&op[i].b);
            op[i].is_q=0;
            h[++num]=op[i].b;//改的数也要离散化
        }
    }

    sort(h+1,h+1+num);
    num=unique(h+1,h+1+num)-(h+1);

    for(int i=1; i<=n; i++) log_add(i,1);
    for(int i=1;i<=m;i++){
        if(op[i].is_q)printf("%d\n",h[log_query(op[i].a,op[i].b,op[i].c)]);
        else {
            log_add(op[i].a,-1);//改-1，实际上就是直接暴力插点改值
            a[op[i].a]=op[i].b;//改a，为下一步搞事铺垫
            log_add(op[i].a,1);//改1，again
        }
    }
    return 0;
}
```

- 感觉就是名次树+前缀和+线段树=主席树

名次树

- 就是BST二叉平衡树上加个size表示子树大小（包括自己）
- 然后感觉就是主席树的感觉了
- Treap写！set搞不定了。。

BST

- O(logn)查询+删除+插入啦
- Treap/Splay
- 一般来讲set够用/红黑树
- 蓝书瞎写吧...代码有空补上...

更多数学

基尔霍夫矩阵/拉普拉斯矩阵 & 矩阵树定理

- L=D-W这是那个矩阵的一个求法~（度数矩阵（只有对角线不是0）-邻接矩阵）
  - (这里介绍下度数矩阵，度数矩阵只有对角线上不是0，第i行第i列的值是点i的度数，也就是邻接矩阵这一行的和)
  - (这里的邻接矩阵**允许重边**，邻接矩阵第i行第j列的值是i和j之间有几条边)
- 矩阵树定理解决一个很简单的问题，一个无向图到底有多少个生成树？
  - **1.Kirchhoff矩阵同时去掉任意一行一列，剩下的矩阵行列式绝对值相等**
  - **2.Kirchhoff矩阵同时去掉任意一行一列，剩下的矩阵的行列式绝对值，就是这个无向图的生成树个数**

    - 知乎上有场论+拉普拉斯搞过来的对图梯度求散度的解释...
- 大概来说就是：拉普拉斯矩阵是n*n的对角线是点i的度数（无向图），然后其他的表示邻接矩阵...
- 行列式计算就是削成上/下三角矩阵然后对角线相乘就是答案!
    - 生成树的多少就是直接高斯消元走啦...行列式变换好像要换符号的...
- 一个特例:完全图的生成树个数是n^{n-2}
- 代码补上（SHOI2016 黑暗前的幻想乡）

```
#include<iostream>
#include<cstdio>
using namespace std;
const int maxn=20;
const int mod=1e9+7;
using ll=long long;
int n;
int m[maxn],u[maxn][maxn*maxn],v[maxn][maxn*maxn];//存边...
int siz[300000];//容斥原理的size大小用... 2^maxn大小至少...

ll qp(ll a,ll p){
    ll res=1,base=a;
    while(p){
        if(p&1)res=res*base%mod;
        base=base*base%mod;
        p>>=1;
    }
    return res;
}

ll koff[maxn][maxn];//Kirchhoff矩阵!
inline ll det(){//行列式板板ovo
    ll res=1;
    int tr=0;//+，-反号次数
    for(int i=1;i<=n-1;i++){
        if(koff[i][i]==0){
            for(int j=i+1;j<=n-1;j++){
                if(koff[j][i]==0){continue;}
                tr^=1;
                for(int k=1;k<=n-1;k++){swap(koff[i][k],koff[j][k]);}
            }
        }//先找个[i][i]位有数的换到i行

        for(int j=i;j<=n-1;j++){
            if(koff[j][i]==0){continue;}//后边行的第i个是0就不用消除了
            ll div=qp(koff[j][i],mod-2);
            //模意义下的除变乘逆元/费马小定理，那么这一行和第i行的相减就是模意义下的0啦
            res=(res*koff[j][i])%mod;
            //对于一方阵其一行/一列乘以k，行列式乘k（这里的k就是这个koff）
            for(int k=i;k<=n-1;k++)koff[j][k]=(koff[j][k]*div)%mod;//当前行全体乘
div
        }

        for(int j=i+1;j<=n-1;j++){//做差消0
            if(koff[j][i]==0){continue;}
            for(int k=i;k<=n-1;k++){
                koff[j][k]=(koff[j][k]+mod-koff[i][k])%mod;
            }
        }
```

```
    }
    for(int i=1;i<=n-1;i++)res*=koff[i][i];
    return tr ? (mod-res)%mod : res;
}

int main()
{
    scanf("%d",&n);
    int upnum=(1<<(n-1))-1;
    for(int i=1;i<n;i++){
        scanf("%d",&m[i]);
        for(int j=1;j<=m[i];j++){
            scanf("%d%d",&u[i][j],&v[i][j]);
        }
    }

    for(int i=1;i<=upnum;i++)siz[i]=siz[i>>1]+(i&1);

    long long ans=0;
    for(int i=1;i<=upnum;i++){
        for(int j=1;j<=n;j++)
            for(int k=1;k<=n;k++)koff[j][k]=0;

        for(int j=1,p=i;p;p>>=1,j++){
            if((p&1)==0)continue;//p看有没有当前位...
            for(int k=1;k<=m[j];k++){
                int bg=u[j][k],ed=v[j][k];
                koff[bg][bg]++,koff[ed][ed]++;
                koff[bg][ed]=(koff[bg][ed]+mod-1)%mod;
                koff[ed][bg]=(koff[ed][bg]+mod-1)%mod;
                //-1是负的邻接矩阵
            }
        }

        ans=(ans+mod+((n-siz[i])%2 ? det():-det()))%mod;//看好括号...
    }
    printf("%lld\n",ans);
    return 0;
}
```

承接banzi1没有的数学——

莫比乌斯反演/狄利克雷卷积

- 终于我还是要看这个哈哈哈！

先来点欧拉函数

- 欧拉函数，符号记作φ(n)，其值为小于n且与n互质的数的个数
  - 性质：
  - 1.对于质数n：φ(n)=n−1
  - 2.对于
    $n = p^k, \phi(n) = (p-1) * p^{k-1}$
  - 3.积性函数，对于gcd(n,m)=1
    $\phi(n*m) = \phi(n) * \phi(m)$

- 4.计算式，对于 $n=\prod p_i^{k_i}, \phi(n)=n*\prod(1-\frac{1}{p_i})$
- 5.欧拉定理，对于互质的a，m有 $a^{\phi(m)} \equiv 1 \pmod{m}$
- 6.小于n且与n互质的数的和：$S=n*\frac{\phi(n)}{2}$
- 7.对于质数p，若: $n\ mod\ p=0, \phi(n*p)=\phi(n)*p$ $n\ mod\ p\neq 0, \phi(n*p)=\phi(n)*(p-1)$
- 8.$\sum_{d|n}\phi(d)=n, \phi(n)=\sum_{d|n}\mu(d)*\frac{n}{d}$

- 线性筛欧拉函数φ：

```
void getphi() {
    phi[1]=1;
    for(int i=2; i<=N; i++) {
        if(!vis[i])  {
            prime[++tot]=i;
            phi[i]=i-1;//欧拉函数素数直接i-1
        }
        for(int j=1; j<=tot; j++) {
            if(i*prime[j]>N)  break;
            vis[i*prime[j]]=1;
            if(i%prime[j]==0) {
                phi[i*prime[j]]=phi[i]*prime[j];//欧拉函数性质7
                break;
            } else phi[i*prime[j]]=phi[i]*(prime[j]-1);//欧拉函数性质7
        }
    }
}
```

莫比乌斯反演

就这玩意：

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

- μ就是莫比乌斯函数啦，定义如下：
  - (1)若d=1则μ(d)=1；
  - (2)若 $d=p_1*p_2*...*p_k, p_i$为互异素数 那么μ(d)=(-1)^k;说直白点，就是d分解质因数后，没有幂次大于平方的质因子，此时函数值根据分解的个数决定
  - (3)其他情况下μ(d)=0，只要当d含有任何质因子的幂次大于等于2，则函数值为0
- 作用就是f不好算但是F好算（约数和/倍数和）也是好算就可以反演嘿嘿嘿
- 注意到这玩意那个**互异素数**的个数，这和容斥原理的奇偶挺相似的，所以一些题考虑一波容斥原理！

莫比乌斯函数的性质

- 1.对于任意正整数n$\sum_{d|n}\mu(d)=[n=1]$[n=1]表示只有当n=1成立时，返回值为1；否则，值为0。
  - 证明方法是二次项系数展开x=1,y=-1；这个就是说只有n=1这个和才是1，否则都是0。(这一条性质是莫比乌斯反演中最常用的)

- 上边这个把n换成gcd(i,j);(ps:一般不都是gcd(i,j)=1？);就可以得到.....
  $\sum_{d|gcd(i,j)}\mu(d)=[gcd(i,j)=1]$大多数直接带入...
- 2.对于任意正整数n$\sum_{d|n}\frac{\mu(d)}{d}=\frac{\phi(n)}{n}$
  - 非常神奇，μ和φ结合了...这就是为啥开头搞点欧拉函数
- 线性筛莫比乌斯函数μ：

```
void get_mu(int n)
{
    mu[1]=1;//在n=1的定义就是mu=1
    for(int i=2;i<=n;i++)
    {
        if(!vis[i]){prim[++cnt]=i;mu[i]=-1;}//奇数个素数mu=-1
        for(int j=1;j<=cnt&&prim[j]*i<=n;j++)
        {
            vis[prim[j]*i]=1;
            if(i%prim[j]==0)break;//这里阻挡了素数大于二次方的mu(prim[j]有多个)，其不改变均为mu=0
            else mu[i*prim[j]]=-mu[i];//因为多一个素数prim[j]，所以变号就行 i -> i*prim[j]
        }
    }
 }
```

实战前置数论技能——整除分块

- 考虑这个问题$\sum_{i=1}^{n}\lfloor\frac{n}{i}\rfloor$，n<1e12，O(n)是不可以的哦！(这经常出现)
- 整除分块出现O(√n)——
  - 经过一番冷静推理暴力打表，我们发现以下性质
  - 1.[N/i]最多只有2√N种取值
  - 2.设 [N/i'] 与 [N/i] 相等，则 i' 的最大值为 [N/[N/i]]
  - 所以：设两个指针 L 和 R， L 的初始值为 1，每次令 R=[N/[N/L]]，将 (R−L+1)*[N/L] 累加至答案中，再令 L=R+1，由于 [N/L] 只有 2√N 种取值，且单调递减，则最多只有 2√N 个取值不同的段，时间复杂度为 O(√N)

```
for(int l=1,r;l<=n;l=r+1)
{
    r=n/(n/l);
    ans+=(r-l+1)*(n/l);
}
```

假装实战

- 发现是自己数学公式推不来——累加...
- 求$\sum_{i=1}^{n}\sum_{j=1}^{n}[gcd(i,j)=1](n<m)$
- 直接上边带入$\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{d|gcd(i,j)}\mu(d)$
- 然后要枚举d，变形公式(不会的就这个...)= $\sum_{d=1}^{n}\mu(d)*\lfloor\frac{n}{d}\rfloor*\lfloor\frac{m}{d}\rfloor$
- 然后上边的后半部分是整除分块
- 此题如果gcd(i,j)=k时，全体除以k变成：$\sum_{i=1}^{\lfloor\frac{n}{k}\rfloor}\sum_{j=1}^{\lfloor\frac{m}{k}\rfloor}[gcd(i,j)=1]$

杜教筛

- 讲道理这玩意是接着上边可以搞的

- 看了看实在太难了...
- 先不补了---
- 任务是小于O(n)筛$\sum_{i=1}^{n} f(n)$，f(n)是积性函数，就是f的前缀和

BM线性递推

- BM算法用于求解常系数线性递推式。
- 它可以在 O(n^2) 的时间复杂度内解决问题。
- 增量法什么的再补啦，先贴个板板

题目描述
给出一个数列P从0开始的前n项，求序列P在mod 998244353下的最短线性递推式，并在mod 998244353 下输出P_m

输入格式
第一行共两个数n,m,表示将会给出序列P的前n项，要求P_m
第二行n个数，表示P_0,P_1,P_2...P_{n-1}

输出格式
第一行输出该最短线性递推式。
第二行输出P_m的值。

```
//某谷多项式板子dalao全套，NTT都有了...以后再补自己的....
#include<bits/stdc++.h>
using namespace std;
typedef int ll;
typedef long long int li;
const ll MAXN=3e5+51,MOD=998244353,G=3,INVG=332748118;
ll exponent,fd,cnt=1,limit=-1,rres,ptr;
ll rev[MAXN],f[MAXN],g[MAXN],tmp[MAXN],tmp2[MAXN],tmp3[MAXN],tbm[MAXN];
ll res[MAXN],base[MAXN],fail[MAXN];
li delta[MAXN];
inline ll read()
{
    register ll num=0,neg=1;
    register char ch=getchar();
    while(!isdigit(ch)&&ch!='-')
    {
        ch=getchar();
    }
    if(ch=='-')
    {
        neg=-1;
        ch=getchar();
    }
    while(isdigit(ch))
    {
        num=(num<<3)+(num<<1)+(ch-'0');
        ch=getchar();
    }
    return num*neg;
}
inline ll qpow(ll base,ll exponent)
{
    li res=1;
    while(exponent)
    {
```

```
        if(exponent&1)
        {
            res=1ll*res*base%MOD;
        }
        base=1ll*base*base%MOD,exponent>>=1;
    }
    return res;
}
inline void NTT(ll *cp,ll cnt,ll inv)
{
    ll cur=0,res=0,omg=0;
    for(register int i=0;i<cnt;i++)
    {
        if(i<rev[i])
        {
            swap(cp[i],cp[rev[i]]);
        }
    }
    for(register int i=2;i<=cnt;i<<=1)
    {
        cur=i>>1,res=qpow(inv==1?G:INVG,(MOD-1)/i);
        for(register ll *p=cp;p!=cp+cnt;p+=i)
        {
            omg=1;
            for(register int j=0;j<cur;j++)
            {
                ll t=1ll*omg*p[j+cur]%MOD,t2=p[j];
                p[j+cur]=(t2-t+MOD)%MOD,p[j]=(t2+t)%MOD;
                omg=1ll*omg*res%MOD;
            }
        }
    }
    if(inv==-1)
    {
        ll invl=qpow(cnt,MOD-2);
        for(register int i=0;i<=cnt;i++)
        {
            cp[i]=1ll*cp[i]*invl%MOD;
        }
    }
}
inline void inv(ll fd,ll *f,ll *res)
{
    static ll tmp[MAXN];
    if(fd==1)
    {
        res[0]=qpow(f[0],MOD-2);
        return;
    }
    inv((fd+1)>>1,f,res);
    ll cnt=1,limit=-1;
    while(cnt<(fd<<1))
    {
        cnt<<=1,limit++;
    }
    for(register int i=0;i<cnt;i++)
    {
        tmp[i]=i<fd?f[i]:0;
```

```
        rev[i]=(rev[i>>1]>>1)|((i&1)<<limit);
    }
    NTT(tmp,cnt,1),NTT(res,cnt,1);
    for(register int i=0;i<cnt;i++)
    {
        res[i]=1ll*(2-1ll*tmp[i]*res[i]%MOD+MOD)%MOD*res[i]%MOD;
    }
    NTT(res,cnt,-1);
    for(register int i=fd;i<cnt;i++)
    {
        res[i]=0;
    }
}
inline void mod(ll *f)
{
    static ll tmp[MAXN],q[MAXN];
    ll deg=fd<<1;
    while(!f[--deg]);
    if(deg<fd)
    {
    return;
    }
    for(register int i=0;i<cnt;i++)
    {
        tmp[i]=i<=deg?f[i]:0;
    }
    reverse(tmp,tmp+1+deg);
    for(register int i=deg+1-fd;i<=deg;i++)
    {
        tmp[i]=0;
    }
    NTT(tmp,cnt,1);
    for(register int i=0;i<cnt;i++)
    {
        q[i]=(li)tmp[i]*tmp3[i]%MOD;
    }
    NTT(q,cnt,-1);
    for(register int i=0;i<cnt;i++)
    {
        tmp[i]=0;
        q[i]=i<=deg-fd?q[i]:0;
    }
    reverse(q,q+1+deg-fd),NTT(q,cnt,1);
    for(register int i=0;i<cnt;i++)
    {
        tmp[i]=(li)q[i]*g[i]%MOD;
    }
    NTT(tmp,cnt,-1);
    for(register int i=0;i<fd;i++)
    {
        f[i]=(f[i]-tmp[i]+MOD)%MOD;
    }
    for(register int i=0;i<cnt;i++)
    {
        q[i]=tmp[i]=0,f[i]=i<fd?f[i]:0;
    }
}
vector<li>bmf[MAXN];
```

```cpp
inline void BerlekampMassey(ll length,ll *base,ll *res)
{
    ll cur=0;
    for(register int i=1;i<=length;i++)
    {
        li curr=base[i];
        for(register int j=0;j<bmf[cur].size();j++)
        {
            curr=(curr-(li)base[i-j-1]*bmf[cur][j]%MOD)%MOD;
        }
        delta[i]=curr;
        if(!delta[i])
        {
            continue;
        }
        fail[cur]=i;
        if(!cur)
        {
            bmf[++cur].resize(i),delta[i]=base[i];
            continue;
        }
        ll id=cur-1,x=bmf[id].size()-fail[id]+i;
        for(register int j=0;j<cur;j++)
        {
            if(i-fail[j]+bmf[j].size()<x)
            {
                id=j,x=i-fail[j]+bmf[j].size();
            }
        }
        bmf[cur+1]=bmf[cur],cur++;
        while(bmf[cur].size()<x)
        {
            bmf[cur].push_back(0);
        }
        li mul=(li)delta[i]*qpow(delta[fail[id]],MOD-2)%MOD;
        bmf[cur][i-fail[id]-1]=(li)(bmf[cur][i-fail[id]-1]+mul)%MOD;
        for(register int j=0;j<bmf[id].size();j++)
        {
            ll t=(li)mul*bmf[id][j]%MOD;
            bmf[cur][i-fail[id]+j]=(bmf[cur][i-fail[id]+j]-t+MOD)%MOD;
        }
    }
    ptr=cur;
    for(register int i=0;i<bmf[cur].size();i++)
    {
        res[i+1]=(bmf[cur][i]%MOD+MOD)%MOD;
    }
}
int main()
{
    fd=read(),exponent=read();
    for(register int i=0;i<fd;i++)
    {
        tbm[i+1]=f[i]=(read()+MOD)%MOD;
    }
    BerlekampMassey(fd,tbm,tmp);
    for(register int i=1;i<=bmf[ptr].size();i++)
    {
```

```
        printf("%d ",tmp[i]);
    }
    puts("");
    for(register int i=1;i<=fd;i++)
    {
        g[fd-i]=MOD-tmp[i];
    }
    g[fd]=1;
    for(register int i=0;i<=fd;i++)
    {
        tmp2[i]=g[i];
    }
    reverse(tmp2,tmp2+1+fd),inv(fd<<1,tmp2,tmp3);
    for(register int i=0;i<=fd;i++)
    {
        tmp2[i]=0;
    }
    while(cnt<(fd<<2))
    {
        cnt<<=1,limit++;
    }
    for(register int i=0;i<cnt;i++)
    {
        rev[i]=(rev[i>>1]>>1)|((i&1)<<limit);
    }
    NTT(g,cnt,1),NTT(tmp3,cnt,1),base[1]=res[0]=1;
    while(exponent)
    {
        if(exponent&1)
        {
            NTT(res,cnt,1),NTT(base,cnt,1);
            for(register int i=0;i<cnt;i++)
            {
                res[i]=(li)res[i]*base[i]%MOD;
            }
            NTT(res,cnt,-1),NTT(base,cnt,-1),mod(res);
        }
        NTT(base,cnt,1);
        for(register int i=0;i<cnt;i++)
        {
            base[i]=(li)base[i]*base[i]%MOD;
        }
        NTT(base,cnt,-1),mod(base),exponent>>=1;
    }
    for(register int i=0;i<fd;i++)
    {
        rres=(rres+(li)res[i]*f[i]%MOD)%MOD;
    }
    printf("%d\n",rres);
}
```

MTT & 分治FFT

FWT

二次剩余

Lucas定理 & EXLucas

蓝书Incredible——DA★ZE~

第一章

1.特殊点中位数（P6）

- 如果猜测数值可以试试中位数的特性ovo

2.汉诺塔专项（P27）

- 搬盘子操作可逆...（对称性）
- 所以步数差就很厉害

3.最小值最大 或者反过来 = 二分（P28）

- 因为容易忘记二分，就写到这里啦~
- 写二分对一些问题 等于 **加个限制条件**变成判断问题

4.无根树转有根树 & 深度表（P35）

- 对树的问题还是很有用吧?
- ~~我写的树太少了~~

5.动态维护[维护信息，而不是重新计算]（P42）

- 对某些题目思考一波，可能有些东西可以动态维护减去了一层时间复杂度哦
- 这个题的差值最大也挺有用的/模板qwq

6.Floyd判圈算法（P44）

- 俩人，一个跑的快（一次走两步），另一个跑的慢（一次走一步）
- 如果有环（链表环，循环节.....各种形式的环），那么慢的肯定会被快的追上（速度差1，追及问题）
- lyj大大的代码节选

```
int ans=k;
int k1=k,k2=k;
do{
    k1=next(n,k1);//小孩1
    k2=next(n,k2);if(k2>ans)ans=k2;//小孩2，第一步
    k2=next(n,k2);if(k2>ans)ans=k2;//小孩2，第二步
    //这里的俩if是针对那个题的
}while(k1!=k2);
```

7.1扫描线 & 细节思考（P46）

- 扫描线操作很厉害，转换模型使用很强的!
- 注意所有题目的**恰好**的地方，那些**关键的临界点**!

7.2扫描线 & 悬线法（P51）

- 最大子矩阵...O(nm)复杂度哦
- 单调栈写法qwq

8.枚举所有 -?> 枚举部分（P53）

- 打不了写的暴力，然后考虑优化容易偏到这里23333
- 有些东西不用枚举，可以预处理+靠枚举别的限制O(1)算出来啦

9.降维打击 & 多维前缀和问题（P55）

- **解决高维问题的常见思路是降维**
- P55题目题解1：综合了一些(8)和(7.2)，枚举上下推一维是一些题目的解决技巧
- 多维前缀和，自己瞎写吧233
- P56代码给了个高维容易扩展的版本（but我没看懂）

10.折半查找/Meet-in-the-Middle（P58）

- 本来是2^n的复杂度，能降低到$2^{\frac{n}{2}} * logn$
- 集合拆开对半分，两边分开算数然后枚举一边在另一边查找
- 好多东西都可以对半拆这么玩哈哈哈哈

11.DAG dp（P60）

- 这个感觉能控制点了，方向那种的感觉
- 来到这一条主要靠抽象数学模型哦

12.预定加和/提前算数（P64）

- 就像递推，因为前一个的影响，后一堆都会如何如何。
- 但是后边被影响的那些元素贡献可能分开算不好计算，这时可以提前加他们的被影响贡献到答案，等于他们都不用算那些影响啦！
- 影响合并 vs 影响分开

13.递推约瑟夫环最后留下的人（P65）

- 直接上公式：
  假设编号0~n-1的n个数排成一圈，从0开始每k个数删除一个，最后留下的数字编号记为f(n)
  则f(1)=0,f(n)=(f(n-1)+k)%n
- 看准上边的n是每一个递推的那个n，不是每次都是模最终n
- 原因详见蓝书，简略就是能重新编号
- ex：第一个删除m的话，答案是(m-k+1+f[n])%n，这个数字要改成1~n之间的
  解释一下的话感觉就是相对偏移，还是能重新编号然后算差值

14.LCS（最长公共子序列O(n^2)）转换成LIS（最长上升子序列O(nlogn)）（P66）

- 要求俩序列分别各自序列里都没有相同元素
- 给A序列重新编号1~lenA+1
- 对应的按着A序列对应给B序列编号

- example：
  A={1,7,5,4,8,3,9}
  B={1,4,3,5,6,2,8,9}
  重新编号后的：
  A={1,2,3,4,5,6,7}
  B={1,4,6,3,0,0,5,7}
- 然后在B里求LIS就好啦~（n^2 -> nlogn）

15.在dp里顺手递推计算优化时间（P69）

- 类似（5）

16.枚举一个集合S的子集S0（P70）

- 这太强了！

```
f[0]=0;
int ALL=(1<<n)-1;
for(int S=1;S<(1<<n);S++){
    f[S]=0;
    for(int S0=S;S0;S0=(S0-1)&S)
        if(cover[S0]==ALL)f[S]=max(f[S],f[S^S0]+1);
}
printf("Case %d: %d\n",++kase,f[ALL]);
```

- 简单来讲就是，(S0-1)&S，后边全1正好&之后搞掉一个，加上最后S\^S0正好枚举了所有不一样的0\^1组成的子集
- 太强了太强了，wsl
- 这个时间复杂度有点迷qwq

17.两个优化量的dp & dp状态设计改善方案（P71）

- 两个优化量v1、v2，要求首先满足v1最小，在v1相同的情况下v2最小。
  - 可以把二者组合成一个量M*(v1)+(v2)，其中M是一个比“v2的理论最大值和理论最小值之差”还要大的数。
- dp状态设计影响抉择的话，就都放到状态里试试叭！
- 当前感觉：dp状态后效？>> 状态修正/决策考虑

18.dp方程——合并与增减变换（P73）

- 给人感觉就像是求 累加$\sum$ 或者是求 累乘$\prod$式子变换一样，无关的搞出来，有关的可以组合——可能新结合量有单调性之类的考虑哦

19.集合dp的状态再压缩（P75）

- 可能有些状态是重复的表示！想想有什么能统一的方法ovo

第三章

##