基础算法

归并求逆序对

```
/*
快速排序接口:
数组头， 左端点，右端点
调用(q, 0, n - 1);
*/
void quick_sort(int q[], int l, int r)
{
    if (l >= r)
        return;
    int x = q[l + r >> 1], i = l - 1, j = r + 1;
    while (i < j)
    {
        do i++; while (q[i] < x);
        do j--; while (q[j] > x);
        if (i < j)
            swap(q[i], q[j]);
    }
    quick_sort(q, l, j);
    quick_sort(q, j + 1, r);
}
/*
归并排序接口:
```

```
数组头， 左端点，右端点
调用(q, 0, n - 1);
*/
int temp[maxn];
int con = 0;//逆序对Count Inversions
void merge_sort(int q[], int l ,int r)
{
    if (l >= r)
        return;
    int mid = l + r >> 1;
    //递归
    merge_sort(q, l, mid);
    merge_sort(q, mid + 1, r);

//    merge(q + l, q + mid + 1, q + mid + 1, q + r + 1, temp);
    //合并
    int k = 0, i = l, j = mid + 1;
    while (i <= mid && j <= r)
        if (q[i] <= q[j])
            temp[k++] = q[i++];
        else//逆序对Count Inversions
        {
            con += mid - i + 1;
            temp[k++] = q[j++];
        }

    while (i <= mid)
    {
        temp[k++] = q[i++];
    }

    while (j <= r)
    {
        temp[k++] = q[j++];
    }


    //改变原数组
    for (int i = l, j = 0; i <= r; i++, j++)
        q[i] = temp[j];
}


int a[1000010];
int main()
{
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &a[i]);
    }
    merge_sort(a, 0, n - 1);
    cout << con << endl;
//    sort(a, a + n);
//    quick_sort(a, 0, n - 1);
//    for (int i = 0; i < n; i++)
//    {
//        printf("%d ", a[i]);
```

```
//    }
//    freopen("F:/Overflow/in.txt","r",stdin);
//    ios::sync_with_stdio(false);
    return 0;
}
```

整数二分

```
bool check(int x) {/* ... */} // 检查x是否满足某种性质

// 区间[l, r]被划分成[l, mid]和[mid + 1, r]时使用：
int bsearch_1(int l, int r)
{
    while (l < r)
    {
        int mid = l + r >> 1;
        if (check(mid)) r = mid;    // check()判断mid是否满足性质
        else l = mid + 1;
    }
    return l;
}
// 区间[l, r]被划分成[l, mid - 1]和[mid, r]时使用：
int bsearch_2(int l, int r)
{
    while (l < r)
    {
        int mid = l + r + 1 >> 1;
        if (check(mid)) l = mid;
        else r = mid - 1;
    }
    return l;
}
```

高精度

```
// 输入输出都要倒序！！！！！！
bool cmp(vector<int> &A, vector<int> &B)//A > B 返回1
{
    if (A.size() != B.size())
        return A.size() > B.size();
    for (int i = A.size() - 1; i >= 0; i--)
    {
        if (A[i] != B[i])
            return A[i] > B[i];
    }
    return 1;
}

vector<int> add(vector<int> &A, vector<int> &B)
{
    vector<int> C;
```

```cpp

    int t = 0;
    for (int i = 0; i < A.size() || i < B.size(); i++)
    {
        if (i < A.size()) t += A[i];
        if (i < B.size()) t += B[i];
        C.push_back(t % 10);
        t /= 10;
    }
    if (t) C.push_back(1);
    return C;
}

vector<int> sub(vector<int> &A, vector<int> &B)
{
    vector<int> C;
    for (int i = 0, t = 0; i < A.size(); i++)
    {
        t = A[i] - t;
        if (i < B.size())
            t -= B[i];
        C.push_back((t + 10) % 10);
        if (t < 0)
            t = 1;
        else
            t = 0;
    }

    //删除前导零
    while (C.size() > 1 && C.back() == 0)
        C.pop_back();
    return C;
}

vector<int> mul(vector<int> &A, int B)
{
    vector<int> C;
    int t = 0;
    for (int i = 0; i < A.size() || t; i++)
    {
        if (i < A.size()) t += A[i] * B;
        C.push_back(t % 10);
        t /= 10;
    }
    return C;
}
//A / B ,余数r
vector<int> div(vector<int> &A, int b, int &r)
{
    vector<int> C;
    r = 0;
    for (int i = A.size() - 1; i >= 0; i--)
    {
        r = r * 10 + A[i];
        C.push_back(r / b);
        r %= b;
    }

```

```
    reverse(C.begin(), C.end());
    while (C.size() > 1 && C.back() == 0)
        C.pop_back();
    return C;
}

int main()
{
    string a;
    int B;
    vector<int> A;
    cin >> a >> B;
    for (int i = a.size() - 1; i >= 0; i--) A.push_back(a[i] - '0');
    vector <int> C;
    int r = 0;
    C = div(A, B, r);
    for (int i = C.size() - 1; i >= 0; i--) printf("%d", C[i]);
    cout << endl<< r << endl;
    return 0;
}
```

离散化

```
vector<int> alls;
vector<PII> add, query;

int a[maxn];
int s[maxn];

//二分
int find(int x)
{
    int l = 0, r = alls.size() - 1;
    while (l < r)
    {
        int mid = l + r >> 1;
        if (alls[mid] >= x) r = mid;
        else l = mid + 1;
    }
    return r + 1;
}


void solve()
{
    cin >> n >> m;
    for (int i = 0; i < n; i++)
    {
        int x, c;
        cin >> x >> c;
        add.push_back({x, c});
        alls.push_back(x);
    }
    for (int i  = 0; i < m; i++)
    {
        int l, r;
```

```
        cin >> l >> r;
        query.push_back({l, r});

        alls.push_back(l);
        alls.push_back(r);
    }
    //去重
    sort(alls.begin(), alls.end());
    alls.erase(unique(alls.begin(), alls.end()), alls.end());

    //
    for (auto item : add)
    {
        int temp = find(item.first);
        a[temp] += item.second;
    }

    for (int i  = 1; i <= alls.size(); i++) s[i] = s[i - 1] + a[i];
    for (auto item : query)
    {
        int l = find(item.first);
        int r = find(item.second);
        cout << s[r] - s[l - 1] << endl;
    }
}
```

数据结构

单调队列

```
    int hh = 0, tt = -1;
    for (int i = 0; i < n; i ++ )
    {
        if (hh <= tt && i - k + 1 > q[hh]) hh ++ ;

        while (hh <= tt && a[q[tt]] >= a[i]) tt -- ;
        q[ ++ tt] = i;

        if (i >= k - 1) printf("%d ", a[q[hh]]);
    }
```

KMP

```
// s[]是长文本，p[]是模式串，n是s的长度，m是p的长度
求模式串的Next数组：
for (int i = 2, j = 0; i <= m; i ++ )
{
    while (j && p[i] != p[j + 1]) j = ne[j];
    if (p[i] == p[j + 1]) j ++ ;
```

```
    ne[i] = j;
}

// 匹配
for (int i = 1, j = 0; i <= n; i ++ )
{
    while (j && s[i] != p[j + 1]) j = ne[j];
    if (s[i] == p[j + 1]) j ++ ;
    if (j == m)
    {
        j = ne[j];
        // 匹配成功后的逻辑
    }
}
```

Trie树

```
int son[N][26], cnt[N], idx;
// 0号点既是根节点，又是空节点
// son[][]存储树中每个节点的子节点
// cnt[]存储以每个节点结尾的单词数量

// 插入一个字符串
void insert(char *str)
{
    int p = 0;
    for (int i = 0; str[i]; i ++ )
    {
        int u = str[i] - 'a';
        if (!son[p][u]) son[p][u] = ++ idx;
        p = son[p][u];
    }
    cnt[p] ++ ;
}

// 查询字符串出现的次数
int query(char *str)
{
    int p = 0;
    for (int i = 0; str[i]; i ++ )
    {
        int u = str[i] - 'a';
        if (!son[p][u]) return 0;
        p = son[p][u];
    }
    return cnt[p];
}
```

并查集

```
(1)朴素并查集:

    int p[N]; //存储每个点的祖宗节点

```

```
    // 返回x的祖宗节点
    int find(int x)
    {
        if (p[x] != x) p[x] = find(p[x]);
        return p[x];
    }

    // 初始化，假定节点编号是1~n
    for (int i = 1; i <= n; i ++ ) p[i] = i;

    // 合并a和b所在的两个集合：
    p[find(a)] = find(b);
```

(2)维护size的并查集：

```
    int p[N], size[N];
    //p[]存储每个点的祖宗节点，size[]只有祖宗节点的有意义，表示祖宗节点所在集合中的点的数量

    // 返回x的祖宗节点
    int find(int x)
    {
        if (p[x] != x) p[x] = find(p[x]);
        return p[x];
    }

    // 初始化，假定节点编号是1~n
    for (int i = 1; i <= n; i ++ )
    {
        p[i] = i;
        size[i] = 1;
    }

    // 合并a和b所在的两个集合：
    size[find(b)] += size[find(a)];
    p[find(a)] = find(b);
```

(3)维护到祖宗节点距离的并查集：

```
    int p[N], d[N];
    //p[]存储每个点的祖宗节点，d[x]存储x到p[x]的距离

    // 返回x的祖宗节点
    int find(int x)
    {
        if (p[x] != x)
        {
            int u = find(p[x]);
            d[x] += d[p[x]];
            p[x] = u;
        }
        return p[x];
    }

    // 初始化，假定节点编号是1~n
    for (int i = 1; i <= n; i ++ )
    {
```

```
            p[i] = i;
            d[i] = 0;
        }

        // 合并a和b所在的两个集合：
        p[find(a)] = find(b);
        d[find(a)] = distance; // 根据具体问题，初始化find(a)的偏移量
```

字符串哈希

```
核心思想：将字符串看成P进制数，P的经验值是131或13331，取这两个值的冲突概率低
小技巧：取模的数用2^64，这样直接用unsigned long long存储，溢出的结果就是取模的结果

typedef unsigned long long ULL;
ULL h[N], p[N]; // h[k]存储字符串前k个字母的哈希值，p[k]存储 P^k mod 2^64

// 初始化
p[0] = 1;
for (int i = 1; i <= n; i ++ )
{
    h[i] = h[i - 1] * P + str[i];
    p[i] = p[i - 1] * P;
}

// 计算子串 str[l ~ r] 的哈希值
ULL get(int l, int r)
{
    return h[r] - h[l - 1] * p[r - l + 1];
}
```

搜索+图论

邻接表

```
// 对于每个点k，开一个单链表，存储k所有可以走到的点。h[k]存储这个单链表的头结点
int h[N], e[N], ne[N], idx;

// 添加一条边a->b
void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx ++ ;
}

// 初始化
idx = 0;
memset(h, -1, sizeof h);
```

康托展开

```
/**************************************************************************
* 简述
康托展开是一个全排列到一个自然数的双射，常用于构建hash表时的空间压缩。
设有n个数（1，2，3，4,…,n），可以有组成不同(n!种)的排列组合，康托展开
表示的就是是当前排列组合在n个不同元素的全排列中的名次。

* 原理
X=a[n]*(n-1)!+a[n-1]*(n-2)!+...+a[i]*(i-1)!+...+a[1]*0!

https://www.cnblogs.com/sky-stars/p/11216035.html
 *************************************************************************/
int FAC[10] =
{
    1,
    1,
    2,
    6,
    24,
    120,
    720,
    5040,
    40320,
    362880,
};//阶乘打表

int cantor(int *a, int n)
{
    int x = 0;
    for (int i = 0; i < n; ++i)
    {
        int smaller = 0;  // 在当前位之后小于其的个数
        for (int j = i + 1; j < n; ++j)
        {
            if (a[j] < a[i])
                smaller++;
        }
        x += FAC[n - i - 1] * smaller; // 康托展开累加
    }
    return x;  // 康托展开值
}
```

拓扑排序

$O(n+m)$

```cpp
bool topsort()
{
    int hh = 0, tt = -1;

    // d[i] 存储点i的入度
    for (int i = 1; i <= n; i ++ )
        if (!d[i])
            q[ ++ tt] = i;

    while (hh <= tt)
    {
        int t = q[hh ++ ];

        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (-- d[j] == 0)
                q[ ++ tt] = j;
        }
    }

    // 如果所有点都入队了，说明存在拓扑序列；否则不存在拓扑序列。
    return tt == n - 1;
}
```

朴素dijkstra

时间复杂是$O\left(n^2+m\right)$ n表示点数，m表示边数

```cpp
int g[N][N];  // 存储每条边
int dist[N];  // 存储1号点到每个点的最短距离
bool st[N];   // 存储每个点的最短路是否已经确定

// 求1号点到n号点的最短路，如果不存在则返回-1
int dijkstra()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

    for (int i = 0; i < n - 1; i ++ )
    {
        int t = -1;     // 在还未确定最短路的点中，寻找距离最小的点
        for (int j = 1; j <= n; j ++ )
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
                t = j;

        // 用t更新其他点的距离
        for (int j = 1; j <= n; j ++ )
            dist[j] = min(dist[j], dist[t] + g[t][j]);

        st[t] = true;
    }

    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
```

```
}
```

堆优化版dijkstra

时间复杂度$O(\text{mlogn})$, n 表示点数，m 表示边数

```
typedef pair<int, int> PII;

int n;      // 点的数量
int h[N], w[N], e[N], ne[N], idx;       // 邻接表存储所有边
int dist[N];        // 存储所有点到1号点的距离
bool st[N];     // 存储每个点的最短距离是否已确定

// 求1号点到n号点的最短距离，如果不存在，则返回-1
int dijkstra()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;
    priority_queue<PII, vector<PII>, greater<PII>> heap;
    heap.push({0, 1});      // first存储距离，second存储节点编号

    while (heap.size())
    {
        auto t = heap.top();
        heap.pop();

        int ver = t.second, distance = t.first;

        if (st[ver]) continue;
        st[ver] = true;

        for (int i = h[ver]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > distance + w[i])
            {
                dist[j] = distance + w[i];
                heap.push({dist[j], j});
            }
        }
    }

    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

spfa 算法（队列优化的Bellman-Ford算法）

时间复杂度 平均情况下 O(m)，最坏情况下 O(nm)

```
int n;      // 总点数
int h[N], w[N], e[N], ne[N], idx;       // 邻接表存储所有边
int dist[N];        // 存储每个点到1号点的最短距离
bool st[N];     // 存储每个点是否在队列中

// 求1号点到n号点的最短路距离，如果从1号点无法走到n号点则返回-1
int spfa()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

    queue<int> q;
    q.push(1);
    st[1] = true;

    while (q.size())
    {
        auto t = q.front();
        q.pop();

        st[t] = false;

        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > dist[t] + w[i])
            {
                dist[j] = dist[t] + w[i];
                if (!st[j])     // 如果队列中已存在j，则不需要将j重复插入
                {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }

    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

spfa判断图中是否存在负环

```
int n;      // 总点数
int h[N], w[N], e[N], ne[N], idx;       // 邻接表存储所有边
int dist[N], cnt[N];        // dist[x]存储1号点到x的最短距离，cnt[x]存储1到x的最短路中经过的点数
bool st[N];     // 存储每个点是否在队列中

// 如果存在负环，则返回true，否则返回false。
bool spfa()
{
    // 不需要初始化dist数组
    // 原理：如果某条最短路径上有n个点（除了自己），那么加上自己之后一共有n+1个点，由抽屉原理一定有两个点相同，所以存在环。
```

```

    queue<int> q;
    for (int i = 1; i <= n; i ++ )
    {
        q.push(i);
        st[i] = true;
    }

    while (q.size())
    {
        auto t = q.front();
        q.pop();

        st[t] = false;

        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > dist[t] + w[i])
            {
                dist[j] = dist[t] + w[i];
                cnt[j] = cnt[t] + 1;
                if (cnt[j] >= n) return true;       // 如果从1号点到x的最短路中包含至少n个点（不包括自己），则说明存在环
                if (!st[j])
                {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }

    return false;
}
```

floyd算法

$O\left(n^{3}\right)$

```
初始化:
    for (int i = 1; i <= n; i ++ )
        for (int j = 1; j <= n; j ++ )
            if (i == j) d[i][j] = 0;
            else d[i][j] = INF;

// 算法结束后，d[a][b]表示a到b的最短距离
void floyd()
{
    for (int k = 1; k <= n; k ++ )
        for (int i = 1; i <= n; i ++ )
            for (int j = 1; j <= n; j ++ )
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}
```

朴素版prim算法

$\mathcal{O}\left(n^2+m\right)$

```
int n;      // n表示点数
int g[N][N];        // 邻接矩阵，存储所有边
int dist[N];        // 存储其他点到当前最小生成树的距离
bool st[N];     // 存储每个点是否已经在生成树中


// 如果图不连通，则返回INF(值是0x3f3f3f3f)，否则返回最小生成树的树边权重之和
int prim()
{
    memset(dist, 0x3f, sizeof dist);

    int res = 0;
    for (int i = 0; i < n; i ++ )
    {
        int t = -1;
        for (int j = 1; j <= n; j ++ )
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
                t = j;

        if (i && dist[t] == INF) return INF;

        if (i) res += dist[t];
        st[t] = true;

        for (int j = 1; j <= n; j ++ ) dist[j] = min(dist[j], g[t][j]);
    }

    return res;
}
```

Kruskal算法

$O(mlogm)$

```
int n, m;       // n是点数，m是边数
int p[N];       // 并查集的父节点数组

struct Edge     // 存储边
{
    int a, b, w;

    bool operator< (const Edge &W)const
    {
        return w < W.w;
    }
}edges[M];

int find(int x)     // 并查集核心操作
{
```

```cpp
    if (p[x] != x) p[x] = find(p[x]);
    return p[x];
}

int kruskal()
{
    sort(edges, edges + m);

    for (int i = 1; i <= n; i ++ ) p[i] = i;    // 初始化并查集

    int res = 0, cnt = 0;
    for (int i = 0; i < m; i ++ )
    {
        int a = edges[i].a, b = edges[i].b, w = edges[i].w;

        a = find(a), b = find(b);
        if (a != b)     // 如果两个连通块不连通，则将这两个连通块合并
        {
            p[a] = b;
            res += w;
            cnt ++ ;
        }
    }

    if (cnt < n - 1) return INF;
    return res;
}
```

染色法判别二分图

$O(n+m)$

```cpp
int n;      // n表示点数
int h[N], e[M], ne[M], idx;     // 邻接表存储图
int color[N];       // 表示每个点的颜色，-1表示未染色，0表示白色，1表示黑色

// 参数：u表示当前节点，c表示当前点的颜色
bool dfs(int u, int c)
{
    color[u] = c;
    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (color[j] == -1)
        {
            if (!dfs(j, !c)) return false;
        }
        else if (color[j] == c) return false;
    }

    return true;
}

bool check()
{
```

```cpp
    memset(color, -1, sizeof color);
    bool flag = true;
    for (int i = 1; i <= n; i ++ )
        if (color[i] == -1)
            if (!dfs(i, 0))
            {
                flag = false;
                break;
            }
    return flag;
}
```

匈牙利算法

$O(nm)$

```cpp
int n1, n2;     // n1表示第一个集合中的点数，n2表示第二个集合中的点数
int h[N], e[M], ne[M], idx;     // 邻接表存储所有边，匈牙利算法中只会用到从第一个集合指向第二个集合的边，所以这里只用存一个方向的边
int match[N];       // 存储第二个集合中的每个点当前匹配的第一个集合中的点是哪个
bool st[N];     // 表示第二个集合中的每个点是否已经被遍历过

bool find(int x)
{
    for (int i = h[x]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (!st[j])
        {
            st[j] = true;
            if (match[j] == 0 || find(match[j]))
            {
                match[j] = x;
                return true;
            }
        }
    }

    return false;
}

// 求最大匹配数，依次枚举第一个集合中的每个点能否匹配第二个集合中的点
int res = 0;
for (int i = 1; i <= n1; i ++ )
{
    memset(st, false, sizeof st);
    if (find(i)) res ++ ;
}
```

网络流

```cpp
int t, n, m, S, T;
//==============================================
```

```cpp
struct edge {
    int to; ll cap; int rev;
};

vector<edge> G[maxn];
int level[maxn];
int iter[maxn];

void add_edge(int from, int to, ll cap) {
    G[from].push_back((edge){to, cap, G[to].size()});
    G[to].push_back((edge){from, 0, G[from].size() - 1});
}

void bfs(int s) {
    memset(level, -1, sizeof level);
    queue<int> que;
    level[s] = 0;
    que.push(s);
    while (!que.empty()) {
        int v = que.front();
        que.pop();
        for (int i = 0; i < G[v].size(); i++) {
            edge &e = G[v][i];
            if (e.cap > 0 && level[e.to] < 0) {
                level[e.to] = level[v] + 1;
                que.push(e.to);
            }
        }
    }
}

ll dfs(int v, int t, int f) {
    if (v == t) return f;
    for (int &i = iter[v]; i < G[v].size(); i++) { // 当前弧优化
        edge &e = G[v][i];
        if (e.cap > 0 && level[v] < level[e.to]) {
            ll d = dfs(e.to, t, min(f * 1ll, e.cap));
            if (d > 0) {
                e.cap -= d;
                G[e.to][e.rev].cap += d;
                return d;
            }
        }
    }
    return 0;
}

ll max_flow(int s, int t) {
    ll flow = 0;
    for (;;) {
        bfs(s);
        if (level[t] < 0) return flow;
        memset(iter, 0, sizeof iter);
        ll f;
        while ((f = dfs(s, t, INF)) > 0) {
            flow += f;
        }
    }
```

```
}


//=============================================
void solve()
{
    while(~scanf("%d%d", &n, &m)){
        S = 0, T = 2 * n + 1;
        int fd, a, b, sm = 0;
        // !!!!!!!!!!!!!n * 2 + 1
        for (int i = 0; i < n * 2 + 1; i++) G[i].clear();
        for (int i = 1; i <= n; i++) {
            scanf("%d", &fd);
            add_edge(S, i, fd);
            add_edge(i + n, T, fd);
            sm += fd;
        }

        for (int i = 1; i <= m; i++) {
            scanf("%d%d", &a, &b);
            add_edge(a, b + n, 1);
            add_edge(b, a + n, 1);
        }

        if (sm & 1) {
            puts("No");
        } else if (max_flow(S, T) == sm) {
            puts("Yes");
        } else {
            puts("No");
        }
}
}

int main()
{

//    freopen("F:/Overflow/in.txt","r",stdin);
//    ios::sync_with_stdio(false);
    solve();
    return 0;
}
```

最小费用流

```
ll sum[maxn];
// =================================================================
int t, n, m, s, q;
// first->shortest dis, second->node
typedef pair<int, int> PII;

struct edge {
    int to, cap, cost, rev;
```

```cpp
};

vector<edge> G[maxn];
int h[maxn];                        // 顶点的势
int dis[maxn];                      // 最短距离
int prevv[maxn], preve[maxn];       // 最短路中的前驱节点和对应的边

vector<int> res;
// 加一条从from到to容量为cap费用为cost的边
void add_edge(int from, int to, int cap, int cost) {
    G[from].push_back((edge){to, cap, cost, G[to].size()});
    G[to].push_back((edge){from, 0, -cost, G[from].size() - 1});
}

ll flow;
ll min_cost_flow(int s, int t, ll f = INF) {
    ll ans = 0;
    flow = 0;
    fill(h, h + t + 1, 0);
    fill(prevv, prevv + t + 1, 0);
    fill(preve, preve + t + 1, 0);
    while (f > 0) {
        priority_queue<PII, vector<PII>, greater<PII> > que;
        fill(dis,dis + t + 1, INF);
        dis[s] = 0;
        que.push(PII(0, s));
        while (!que.empty()) {
            PII p = que.top();
            que.pop();
            int v = p.second;
            if (dis[v] < p.first) continue;
            for (int i = 0; i < G[v].size(); i++) {
                edge &e = G[v][i];
                if (e.cap > 0 && dis[e.to] > dis[v] + e.cost + h[v] - h[e.to]) {
                    dis[e.to] = dis[v] + e.cost + h[v] - h[e.to];
                    prevv[e.to] = v;
                    preve[e.to] = i;
                    que.push(PII(dis[e.to], e.to));
                }
            }
        }
        if (dis[t] == INF) {
            return ans;
        }
        for (int v = 0; v <= t + 1; v++) h[v] += dis[v];
        int d = f;
        for (int v = t; v != s; v = prevv[v]) {
            d = min(d, G[prevv[v]][preve[v]].cap);
        }
        f -= d;
        flow += d;
        ans += d * h[t];
        res.push_back(h[t] * d);
        for (int v = t; v != s; v = prevv[v]) {
            G[prevv[v]][preve[v]].cap -= d;
            G[v][G[prevv[v]][preve[v]].rev].cap += d;
        }
    }
```

```
    return ans;
}

void init() {
    res.clear();
    for (int i = 0; i < maxn; i++) {
        G[i].clear();
    }
}
// ========================================================
void solve()
{
    while (~scanf("%d%d", &n, &m)) {
        init();
        for (int i = 1; i <= m; i++) {
            int a, b, c;
            scanf("%d%d%d", &a, &b, &c);
            add_edge(a, b, 1, c);
        }

        res.push_back(0);
        min_cost_flow(1, n);
        sort(res.begin(), res.end());
        for (int i = 1; i < res.size(); i++) {
            sum[i] = sum[i - 1] + res[i];
        }
        scanf("%d", &q);
        while (q--) {
            ll u, v;
            scanf("%lld%lld", &u, &v);
            if (u * (res.size() - 1) < v) {
                puts("NaN");
            } else {
                int pos = v / u;
                ll ans = sum[pos] * u;
                if (u * pos < v) {
                    ans += res[pos + 1] * (v - u * pos);
                }
                ll gc = __gcd(ans, v);
                printf("%lld/%lld\n", ans / gc, v / gc);
            }
        }
    }
}

int main()
{

//    freopen("F:/Overflow/in.txt","r",stdin);
//    ios::sync_with_stdio(false);
    solve();
    return 0;
}

```

数学知识

试除法分解质因数

```
void divide(int x)
{
    for (int i = 2; i <= x / i; i ++ )
        if (x % i == 0)
        {
            int s = 0;
            while (x % i == 0) x /= i, s ++ ;
            cout << i << ' ' << s << endl;
        }
    if (x > 1) cout << x << ' ' << 1 << endl;
    cout << endl;
}
```

线性筛法求素数

```
int primes[N], cnt;     // primes[]存储所有素数
bool st[N];         // st[x]存储x是否被筛掉

void get_primes(int n)
{
    for (int i = 2; i <= n; i ++ )
    {
        if (!st[i]) primes[cnt ++ ] = i;
        for (int j = 0; primes[j] <= n / i; j ++ )
        {
            st[primes[j] * i] = true;
            if (i % primes[j] == 0) break;
        }
    }
}
```

试除法求所有约数

```
vector<int> get_divisors(int x)
{
    vector<int> res;
    for (int i = 1; i <= x / i; i ++ )
        if (x % i == 0)
        {
            res.push_back(i);
            if (i != x / i) res.push_back(x / i);
        }
    sort(res.begin(), res.end());
    return res;
}
```

求欧拉函数

```
int phi(int x)
{
    int res = x;
    for (int i = 2; i <= x / i; i ++ )
        if (x % i == 0)
        {
            res = res / i * (i - 1);
            while (x % i == 0) x /= i;
        }
    if (x > 1) res = res / x * (x - 1);

    return res;
}
```

筛法求欧拉函数

```
int primes[N], cnt;     // primes[]存储所有素数
int euler[N];           // 存储每个数的欧拉函数
bool st[N];         // st[x]存储x是否被筛掉


void get_eulers(int n)
{
    euler[1] = 1;
    for (int i = 2; i <= n; i ++ )
    {
        if (!st[i])
        {
            primes[cnt ++ ] = i;
            euler[i] = i - 1;
        }
        for (int j = 0; primes[j] <= n / i; j ++ )
```

```
        {
            int t = primes[j] * i;
            st[t] = true;
            if (i % primes[j] == 0)
            {
                euler[t] = euler[i] * primes[j];
                break;
            }
            euler[t] = euler[i] * (primes[j] - 1);
        }
    }
}
```

扩展欧几里得算法

```
// 求x, y，使得ax + by = gcd(a, b)
int exgcd(int a, int b, int &x, int &y)
{
    if (!b)
    {
        x = 1; y = 0;
        return a;
    }
    int d = exgcd(b, a % b, y, x);
    y -= (a/b) * x;
    return d;
}
```

高斯消元

```
// a[N][N]是增广矩阵
int gauss()
{
    int c, r;
    for (c = 0, r = 0; c < n; c ++ )
    {
        int t = r;
        for (int i = r; i < n; i ++ )   // 找到绝对值最大的行
            if (fabs(a[i][c]) > fabs(a[t][c]))
                t = i;

        if (fabs(a[t][c]) < eps) continue;

        for (int i = c; i <= n; i ++ ) swap(a[t][i], a[r][i]);      // 将绝对值最大的行换到最顶端
        for (int i = n; i >= c; i -- ) a[r][i] /= a[r][c];      // 将当前上的首位变成1
        for (int i = r + 1; i < n; i ++ )       // 用当前行将下面所有的列消成0
            if (fabs(a[i][c]) > eps)
```

```
                for (int j = n; j >= c; j -- )
                    a[i][j] -= a[r][j] * a[i][c];

        r ++ ;
    }

    if (r < n)
    {
        for (int i = r; i < n; i ++ )
            if (fabs(a[i][n]) > eps)
                return 2; // 无解
        return 1; // 有无穷多组解
    }

    for (int i = n - 1; i >= 0; i -- )
        for (int j = i + 1; j < n; j ++ )
            a[i][n] -= a[i][j] * a[j][n];

    return 0; // 有唯一解
}
```

递归法求组合数

```
// c[a][b] 表示从a个苹果中选b个的方案数
for (int i = 0; i < N; i ++ )
    for (int j = 0; j <= i; j ++ )
        if (!j) c[i][j] = 1;
        else c[i][j] = (c[i - 1][j] + c[i - 1][j - 1]) % mod;
```

通过预处理逆元的方式求组合数

```
首先预处理出所有阶乘取模的余数fact[N]，以及所有阶乘取模的逆元infact[N]
如果取模的数是质数，可以用费马小定理求逆元
int qmi(int a, int k, int p)    // 快速幂模板
{
    int res = 1;
    while (k)
    {
        if (k & 1) res = (LL)res * a % p;
        a = (LL)a * a % p;
        k >>= 1;
    }
    return res;
}

// 预处理阶乘的余数和阶乘逆元的余数
fact[0] = infact[0] = 1;
for (int i = 1; i < N; i ++ )
```

```
{
    fact[i] = (LL)fact[i - 1] * i % mod;
    infact[i] = (LL)infact[i - 1] * qmi(i, mod - 2, mod) % mod;
}
```

Lucas定理

```
若p是质数，则对于任意整数 1 <= m <= n，有：
    C(n, m) = C(n % p, m % p) * C(n / p, m / p) (mod p)

int qmi(int a, int k)       // 快速幂模板
{
    int res = 1;
    while (k)
    {
        if (k & 1) res = (LL)res * a % p;
        a = (LL)a * a % p;
        k >>= 1;
    }
    return res;
}


int C(int a, int b)     // 通过定理求组合数C(a, b)
{
    int res = 1;
    for (int i = 1, j = a; i <= b; i ++, j -- )
    {
        res = (LL)res * j % p;
        res = (LL)res * qmi(i, p - 2) % p;
    }
    return res;
}


int lucas(LL a, LL b)
{
    if (a < p && b < p) return C(a, b);
    return (LL)C(a % p, b % p) * lucas(a / p, b / p) % p;
}

```

分解质因数法求组合数

```
当我们需要求出组合数的真实值，而非对某个数的余数时，分解质因数的方式比较好用：
    1. 筛法求出范围内的所有质数
    2. 通过 C(a, b) = a! / b! / (a - b)! 这个公式求出每个质因子的次数。 n! 中p的次数是 n / p + n / p^2 + n / p^3 + ...
```

```
    3. 用高精度乘法将所有质因子相乘

int primes[N], cnt;     // 存储所有质数
int sum[N];     // 存储每个质数的次数
bool st[N];     // 存储每个数是否已被筛掉


void get_primes(int n)      // 线性筛法求素数
{
    for (int i = 2; i <= n; i ++ )
    {
        if (!st[i]) primes[cnt ++ ] = i;
        for (int j = 0; primes[j] <= n / i; j ++ )
        {
            st[primes[j] * i] = true;
            if (i % primes[j] == 0) break;
        }
    }
}


int get(int n, int p)       // 求n! 中的次数
{
    int res = 0;
    while (n)
    {
        res += n / p;
        n /= p;
    }
    return res;
}


vector<int> mul(vector<int> a, int b)       // 高精度乘低精度模板
{
    vector<int> c;
    int t = 0;
    for (int i = 0; i < a.size(); i ++ )
    {
        t += a[i] * b;
        c.push_back(t % 10);
        t /= 10;
    }

    while (t)
    {
        c.push_back(t % 10);
        t /= 10;
    }

    return c;
}

get_primes(a);  // 预处理范围内的所有质数

for (int i = 0; i < cnt; i ++ )     // 求每个质因数的次数
{
    int p = primes[i];
```

```
    sum[i] = get(a, p) - get(b, p) - get(a - b, p);
}

vector<int> res;
res.push_back(1);

for (int i = 0; i < cnt; i ++ )     // 用高精度乘法将所有质因子相乘
    for (int j = 0; j < sum[i]; j ++ )
        res = mul(res, primes[i]);
```

卡特兰数

```
给定n个0和n个1，它们按照某种顺序排成长度为2n的序列，满足任意前缀中0的个数都不少于1的个数的序列的数量为： Cat(n) = C(2n, n) / (n + 1)
```

动态规划

背包问题

01

```
    for (int i = 1; i <= n; i++)
    {
        for (int j = 0; j <= V; j++)
        {
            dp[i][j] = dp[i - 1][j];
            if (j >= v[i])
            {
                dp[i][j] = max(dp[i][j], dp[i - 1][j - v[i]] + w[i]);
            }
        }
    }
    cout << dp[n][V] << endl;
```

多重

```
    int con = 0;
    for (int i = 1; i <= N; i++)
    {
```

```
        int a, b, s;
        cin >> a >> b >> s;
        int k = 1;
        while (k <= s)
        {
            con++;
            v[con] = a * k;
            w[con] = b * k;
            s -= k;
            k *= 2;
        }

        if (s > 0)
        {
            con++;
            v[con] = a * s;
            w[con] = b * s;
        }

    }

    for (int i = 1; i <= con; i++)
    {
        for (int j = V; j >= v[i]; j--)
        {
            dp[j] = max(dp[j], dp[j - v[i]] + w[i]);
        }
    }
    cout << dp[V] << endl;
```

分组

```
    cin >> N >> V;
    for (int i = 1; i <= N; i++)
    {
        cin >> s[i];
        for (int j = 1; j <= s[i]; j++)
            cin >> v[i][j] >> w[i][j];
    }

    for (int i = 1; i <= N; i++)
        for (int j = 0; j <= V; j++)
        {
            dp[i][j] = dp[i - 1][j];
            for (int k = 0; k <= s[i]; k++)
            {

                if (v[i][k] <= j)
                    dp[i][j] = max(dp[i][j], dp[i - 1][j - v[i][k]] + w[i][k]);
            }
        }

    cout << dp[N][V] << endl;
```

完全

```
cin >> n >> V;
for (int i = 1; i <= n; i++)
{
    cin >> v[i] >> w[i];
}

for (int i = 1; i <= n; i++)
{
    for (int j = v[i]; j <= V; j++)
    {
        dp[j] = max(dp[j], dp[j - v[i]] + w[i]);

    }
}
cout << dp[V] << endl;
```

线性DP

最长上升子序列

```
cin >> t;
for (int i = 0; i < t; i++)
{
    cin >> num[i];
    dp[i] = 1;
}
for (int i = 0; i < t; i++)
{
    for (int j = 0; j < i; j++)
    {
        if (num[i] > num[j])
            dp[i] = max(dp[i], dp[j] + 1);
    }
}
cout << *max_element(dp, dp + t) << endl;
```

最长上升子序列2

```
scanf("%d", &n);
for (int i = 0; i < n; i ++ ) scanf("%d", &a[i]);

int len = 0;
for (int i = 0; i < n; i ++ )
{
    int l = 0, r = len;
    while (l < r)
    {
        int mid = l + r + 1 >> 1;
```

```
            if (q[mid] < a[i]) l = mid;
            else r = mid - 1;
        }
        len = max(len, r + 1);
        q[r + 1] = a[i];
    }
    printf("%d\n", len);
```

最长公共子序列

```
    cin >> n >> m;
    cin >> s1 + 1 >> s2 + 1;
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= m; j++)
        {
            dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            if (s1[i] == s2[j]) dp[i][j] = max(dp[i][j], dp[i - 1][j - 1] + 1);
        }
    }
    cout << dp[n][m] << endl;
```

最小编辑距离

```
void solve()
{
    cin >> n >> str1 + 1;
    cin >> m >> str2 + 1;

    for (int i = 0; i <= m; i++) dp[0][i] = i;//增加
    for (int i = 0; i <= n; i++) dp[i][0] = i;//删

    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= m; j++)
        {
            dp[i][j] = min(dp[i - 1][j] + 1, dp[i][j - 1] + 1);
            if (str1[i] == str2[j]) dp[i][j] = min(dp[i][j], dp[i - 1][j - 1]);
            else dp[i][j] = min(dp[i][j], dp[i - 1][j - 1] + 1);
        }
    }
    cout << dp[n][m] << endl;
}
```

区间DP

```
    cin >> t;
    for (int i = 1; i <= t; i++)
        cin >> num[i];
    for (int i = 1; i <= t; i++)
        sum[i] += sum[i - 1] + num[i];
```

```
    for (int i = 2; i <= t; i++)//长度1的时候不需要代价，长度从小到大开始枚举
    {
        for (int j = 1; j + i - 1<= t; j++)//起始位置
        {
            int l = j, r = j + i - 1;
            dp[l][r] = 1e8;
            for (int k = l; k <= r - 1; k++)
            {
                dp[l][r] = min(dp[l][r], dp[l][k] + dp[k + 1][r] + sum[r] - sum[l - 1]);
            }
        }
    }
    cout << dp[1][t] << endl;
```

博弈论

NIM游戏

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

    int res = 0;
    while (n -- )
    {
        int x;
        scanf("%d", &x);
        res ^= x;
    }

    if (res) puts("Yes");
    else puts("No");
```

集合nim

给定nn堆石子以及一个由kk个不同正整数构成的数字集合SS。

现在有两位玩家轮流操作，每次操作可以从任意一堆石子中拿取石子，每次拿取的石子数量必须包含于集合SS，最后无法进行操作的人视为失败。

问如果两人都采用最优策略，先手是否必胜。

```
//SG函数
int SG(int x)
{
    if (f[x] != -1) return f[x];

    unordered_set<int> S;
    for (int i = 0; i < m; i++)
    {
```

```
        int sum = s[i];
        if (x >= sum) S.insert(SG(x - sum));
    }
    for (int i = 0; ; i++)
    {
        if (!S.count(i))
            return f[x] = i;
    }
}


//====================================================
void solve()
{
    memset(f, -1, sizeof f);
    cin >> m;
    for (int i = 0; i < m; i++)
        cin >> s[i];
    cin >> n;
    int res = 0;
    for (int i = 0; i < n; i++)
    {
        int x;
        cin >> x;
        res ^= SG(x);
    }

    if (res) puts("Yes");
    else puts("No");
}
```

高级数据结构

树状数组

```
int lowbit(int x) {
    return x & -x;
}

void add(int x, int k) {
    for (int i = x; i <= n; i += lowbit(i)) e[i] += k;
}

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

线段树

最大数

```
struct node {
    int l, r;
    ll MAX;
}tr[maxn << 2];
int n, m, p;
void pushup(int p) {
    tr[p].MAX = max(tr[p << 1].MAX, tr[p << 1 | 1].MAX);
}

void build(int p, int l, int r) {
    tr[p] = {l, r};
    if (l == r) {
//      tr[p].MAX = a[l];
        return ;
    }
    int mid = l + r >> 1;
    build(p << 1, l, mid);
    build(p << 1 | 1, mid + 1, r);
//  pushup(p);
}

void modify(int p, int x, int k) {
    if (tr[p].l == tr[p].r) {
        tr[p].MAX = k;
        return ;
    }

    int mid = tr[p].l + tr[p].r >> 1;

    if (x <= mid) {
        modify(p << 1, x, k);
    }
    if (x > mid) {
        modify(p << 1 | 1, x, k);
    }
    pushup(p);
}

ll query(int p, int l, int r) {
    if (l <= tr[p].l && tr[p].r <= r) {
        return tr[p].MAX ;
    }
    int mid = tr[p].l + tr[p].r >> 1;
    if (r <= mid) return query(p << 1, l, r);
    if (l > mid) return query(p << 1 | 1, l, r);
    return max(query(p << 1, l,  mid), query(p << 1 | 1, mid + 1, r));
}
```

区间修改

```
struct node{
```

```
    int l, r;
    ll sum, add;
}tr[MAXN * 4];
ll a[MAXN];
void pushup(int p) {
    tr[p].sum = tr[p << 1].sum + tr[p << 1 | 1].sum;
}

void pushdown(int p) {
    node &root = tr[p], &left = tr[p << 1], &right = tr[p << 1 | 1];
    if (root.add) {
        left.sum += (ll)(left.r - left.l + 1) * (root.add), left.add += root.add;
        right.sum += (ll)(right.r - right.l + 1) * (root.add), right.add += root.add;
        root.add = 0;
    }
}

void build(int p, int l, int r) {
    tr[p] = {l, r};
    if (l == r) {
        tr[p] = {l, r, a[l], 0};
        return ;
    }
    int mid = tr[p].l + tr[p].r >> 1;
    build(p << 1, l, mid), build(p << 1 | 1, mid + 1, r);
    pushup(p);
}

void modify(int p, int l, int r, int k) {
    if (tr[p].l == tr[p].r) {
        tr[p].sum += (ll)(tr[p].r - tr[p].l + 1) * k;
        tr[p].add += k;
        return ;
    }
    pushdown(p);
    int mid = tr[p].l + tr[p].r >> 1;
    if (l <= mid) modify(p << 1, l, r, k);
    if (r >  mid) modify(p << 1 | 1, l, r, k);
    pushup(p);
}

//ll query(int p, int l, int r) {
//  if (l <= tr[p].l && tr[p].r <= r) {
//      return tr[p].sum;
//  }
//
//  pushdown(p);
//  int mid = tr[p].l + tr[p].r >> 1;
//  if (r <= mid) return query(p << 1, l, r);
//  if (l >  mid) return query(p << 1 | 1, l, r);
//  return query(p << 1 ,l, mid) + query(p << 1 | 1, mid + 1, r);
//
//}
ll query(int p, int l, int r) {
    if (l <= tr[p].l && tr[p].r <= r) return tr[p].sum;
    pushdown(p);
    int mid = tr[p].l + tr[p].r >> 1;
```

```
    ll res = 0;
    if (l <= mid) res += query(p << 1, l, r);
    if (r >  mid) res += query(p << 1 | 1, l, r);
    return res;
}
```

最大公约数

```
struct node{
    int l, r;
    ll d, sum;
}tr[MAXN * 4];

ll a[MAXN];
ll gcd(ll a, ll b) {
    return b == 0 ? a : gcd(b, a % b);
}

void pushup(node &p, node &l, node &r) {
    p.sum = l.sum + r.sum;
    p.d = gcd(l.d, r.d);
}

void pushup(int p) {
    pushup(tr[p], tr[p << 1], tr[p << 1 | 1]);
}

void build(int p, int l, int r) {
    tr[p] = {l, r};
    if (l == r) {
        ll b = a[r] - a[r - 1];
        tr[p] = {l, r, b, b};
        return;
    }
    int mid = tr[p].l + tr[p].r >> 1;
    build(p << 1, l, mid), build(p << 1 | 1, mid + 1, r);
    pushup(p);
}

void modify(int p, int x, ll v) {
    if (tr[p].l == x && tr[p].r == x) {
        ll b = tr[p].sum + v;
        tr[p] = {x, x, b, b};
        return ;
    }
    int mid = tr[p].l + tr[p].r >> 1;
    if (x <= mid) modify(p << 1, x, v);
    else modify(p << 1 | 1, x, v);
    pushup(p);
}

node query(int p, int l, int r) {
    if (tr[p].l >= l && tr[p].r <= r) return tr[p];
    int mid = tr[p].l + tr[p].r >> 1;
```

```
    if (r <= mid) return query(p << 1, l, r);
    if (l >  mid) return query(p << 1 | 1, l, r);
    else {
        node left = query(p << 1, l, r);
        node right = query(p << 1 | 1, l, r);
        node res;
        pushup(res, left, right);
        return res;
    }
}
```

AC自动机

给定 n个长度不超过 5050 的由小写英文字母组成的单词，以及一篇长为 m 的文章。

请问，有多少个单词在文章中出现了。

```
const int N = 10010, S = 55, M = 1000010;

int n;
int tr[N * S][26], cnt[N * S], idx;
char str[M];
int q[N * S], ne[N * S];

void insert()
{
    int p = 0;
    for (int i = 0; str[i]; i ++ )
    {
        int t = str[i] - 'a';
        if (!tr[p][t]) tr[p][t] = ++ idx;
        p = tr[p][t];
    }
    cnt[p] ++ ;
}

void build()
{
    int hh = 0, tt = -1;
    for (int i = 0; i < 26; i ++ )
        if (tr[0][i])
            q[ ++ tt] = tr[0][i];

    while (hh <= tt)
    {
        int t = q[hh ++ ];
        for (int i = 0; i < 26; i ++ )
        {
            int p = tr[t][i];
            if (!p) tr[t][i] = tr[ne[t]][i];
```

```
            else
            {
                ne[p] = tr[ne[t]][i];
                q[ ++ tt] = p;
            }
        }
    }
}

int main()
{
    int T;
    scanf("%d", &T);
    while (T -- )
    {
        memset(tr, 0, sizeof tr);
        memset(cnt, 0, sizeof cnt);
        memset(ne, 0, sizeof ne);
        idx = 0;

        scanf("%d", &n);
        for (int i = 0; i < n; i ++ )
        {
            scanf("%s", str);
            insert();
        }
        build();
        scanf("%s", str);
        int res = 0;
        for (int i = 0, j = 0; str[i]; i ++ )
        {
            int t = str[i] - 'a';
            j = tr[j][t];

            int p = j;
            while (p)
            {
                res += cnt[p];
                cnt[p] = 0;
                p = ne[p];
            }
        }

        printf("%d\n", res);
    }

    return 0;
}
```