XCPC 算法竞赛模板

Sergio Gao

2025 年 6 月 17 日

目　录

第 1 章　图论

主要记录一些比较常用的高级算法。

1.1 Tarjan 相关图论算法

叫 Tarjan 的图论算法其实有很多。所以 Tarjan 实际上可以说是一类思想。

它可以说是洞察了 dfs 树的性质，考虑了返祖边，跨越边（对于无向图，实际上并不可能存在）等。

1.1.1 求强连通分量

这是对于有向图，求出若干极大两两可互相到达的点集。

一些没有写上去但可能会用到的 trick：

对于强连通分量，存在容易算法判断是否存在非零环。（提示：如果存在两个权值不一样的环，那么一定存在非零环。）

```
#include<bits/stdc++.h>
using namespace std;
class TarjanSCC
{
private:
    int n;
    vector<vector<int> > e;
    vector<int> dfn, low;
    int dfncnt;
    stack<int> st;
    vector<bool> instack;
    void dfs(int u)
    {
        if(dfn[u]) return;
        low[u] = dfn[u] = ++dfncnt;
        st.push(u);
        instack[u] = 1;
        for(auto v: e[u])
        {
            if(!dfn[v])
            {
                dfs(v);
                low[u] = min(low[u], low[v]);
            }
            else if(instack[v])
            {
                low[u] = min(low[u], dfn[v]);
            }
        }
        if(dfn[u] == low[u])
        {
            ++sc;
            while(!st.empty() && st.top()!=u)
            {
                scc[st.top()] = sc;

                instack[st.top()] = 0;
                st.pop();
            }
            assert(!st.empty() && st.top() == u);
            scc[st.top()] = sc;
            instack[st.top()] = 0;
            st.pop();
        }
        return;
    }
    int sc;
public:
    vector<int> scc;
    TarjanSCC(int _n)
```

```
    {
        n = _n;
        e = vector<vector<int> >(n);
        dfn = vector<int>(n);
        low = vector<int>(n);
        st = stack<int>();
        instack = vector<bool>(n);
        scc = vector<int>(n);
        dfncnt = sc = 0;
    }
    void addEdge(int u, int v)
    {
        assert(u<n && v<n);
        e[u].push_back(v);
        return;
    } // point id should strictly less than n
    void get()
    {
        for(int i=0; i<n; i++) dfs(i);
        return;
    }
};
void solve()
{
    int n, m;
    cin>>n>>m;
    TarjanSCC _graph(n+1);
    for(int i=1; i<=m; i++)
    {
        int u, v;
        cin>>u>>v;
        _graph.addEdge(u, v);
    }
    _graph.get();
    return;
}
int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);
    int T = 1;
    // cin>>T;
    while(T--)
    {
        solve();
    }
    return 0;
}
```

1.2 网络流

网络流在 XCPC 中常考的两类：

第一类是最大流或者最小割。这类一般建完图用 Dinic 算法解决即可。

特别地，最经典的一类题型：等价于求二分图的最大匹配。而最大匹配用最大流来跑又快又好，并且还能顺便证明二分图最大匹配 = 最小点覆盖。

二分图中：最大匹配（最大流） = 最小点覆盖（最小割） = 点数 - 最大独立集

第二类是最小费用最大流。这类一般建完图用 Dij 费用流解决即可。大多数费用流算法都和其流 f 相关。构造时一般是那最大流作为可以控制的约束。特别地，最经典的一类题型：等价于求二分图的最大权匹配。

1.2.1 Dinic 最大流

一般的时间复杂度会估计到 $O(n^2m)$，但很多题目的建图有特殊性，跑不满。特别的，如果拿来处理二分图最大匹配，复杂度可以估到 $O(n^3)$。

```
#include<bits/stdc++.h>
using namespace std;
template<class T> class Dinic
{
private:
    struct _Edge
    {
        int to;
        T cap;
    };
    int n;
    vector<_Edge> pool;
    vector<vector<int> > g;
    vector<int> cur, h;
    bool bfs(int s, int t)
    {
        for(auto &i : h) i = -1;
        h[s] = 0;
        queue<int> q;
        q.push(s);
        while(!q.empty())
        {
            int u = q.front(); q.pop();
            for(int i=0; i<(int)g[u].size(); i++)
            {
                auto _to = pool[g[u][i]].to;
                auto _cap = pool[g[u][i]].cap;
                if( _cap == 0 || h[_to] !=-1 ) continue;
                h[_to] = h[u] + 1;
                if(_to == t) return true;
                q.push(_to);
            }
        }

        return false;
    }
    T dfs(int u, int t, T f)
    {
        if(u==t) return f;
        auto r = f;
        for(auto &i = cur[u]; i<(int)g[u].size(); i++)
        {
            auto _to = pool[g[u][i]].to;
            auto _cap = pool[g[u][i]].cap;
            if(h[_to] != h[u] + 1) continue;
            auto a = dfs(_to, t, min(r, _cap));
            pool[g[u][i]] .cap -= a;
            pool[g[u][i]^1] .cap += a;
            r -= a;
            if(r==0) break;
        }
        return f-r;
    }
```

```
public:
    Dinic(int _n)
    {
        n = _n;
        cur.resize(n);
        h.resize(n);
        g.resize(n);
    }
    void addEdge(int u, int v, T c = numeric_limits<T>::max())
    {
        g[u].push_back(pool.size());
        pool.push_back({v, c});

        g[v].push_back(pool.size());
        pool.push_back({u, 0});

        return;
    }
    T flow(int s, int t)
    {
        T ans = 0;
        while(bfs(s, t))
        {
            for(auto &i : cur) i = 0;
            ans += dfs(s, t, numeric_limits<T>::max());
        }
        return ans;
    }
};
void solve()
{
    int n, m, k;
    cin>>n>>m>>k;
    vector<int> p(m), invid(m);
    vector<pair<int, int> > pp(m);
    for(int i=0; i<m; i++) cin>>p[i], pp[i] = {p[i], i};
    sort(pp.begin(), pp.end());
    sort(p.begin(), p.end());
    for(int i=0; i<m; i++) invid[pp[i].second] = i;
    vector<int> s[n];
    for(int i=0; i<n; i++)
    {
        s[i] = vector<int>(m);
        for(int j=0; j<m; j++)
        {
            cin>>s[i][invid[j]];
        }
    }
    Dinic<long long> d(n*m + 1 + 1);
    for(int i=0; i<n; i++)
    {
        d.addEdge(0, i*m+0+1, s[i][0]);
        for(int j=0; j<m-1; j++)
        {
            // i*m+j+1 [i,j]
            d.addEdge(i*m+j+1, i*m+ j+1 +1, s[i][j+1]);
        }
        d.addEdge(i*m+(m-1)+1, n*m+1);
```

```
    }
    for(int i=1; i<=k; i++)
    {
        int x, y, dis;
        cin>>x>>y>>dis;
        x--;
        y--;
        if(x==y) continue;
        for(int j=0, r=0; j<m; j++)
        {
            while(r<m && p[r] - p[j] <= dis) r++;
            if(r>=m) break;
            d.addEdge(x*m+ (r-1) +1, y*m+ j +1);
            d.addEdge(y*m+ (r-1) +1, x*m+ j +1);
        }
    }
    cout<<d.flow(0, n*m+1)<<"\n";
    return;
}
int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0), cout.tie(0);

    int T;
    cin>>T;
    while(T--)
    {
        solve();
    }
        return 0;
}
```

1.2.2 Dijkstra 费用流

首先，费用流一般是要自己构造的，所以一般有意义的问题不会有真正的负环。然而负权是允许的。通过调整正负，可以把最小费用流的算法改写成最大费用流。

Dijkstra 费用流也就是在保证初始能赋出一个合理的势能函数的前提下，用 Dijkstra 来找到从源点到汇点的最短路。

最暴力的做法是初始做一遍 spfa，这个复杂度可能会达到 $O(nm)$。或者根据题目的特殊性。有一些问题它给出时就已经是让我们求如何最大化一些势能差之和。

除去这个初始化，复杂度是 $O(fm\log m)$。要注意，像是二部图最大权匹配，f 取决于较小的那一部，所以不要误认为其等于总点数。

```
#include<bits/stdc++.h>
using namespace std;
template<class T> class mcmf
{
private:
    const T INF = numeric_limits<T>::max();
    struct _edge
    {
        int to, cap;
        T cost;
    };
    int n;
    vector<_edge> e;
    vector<vector<int> > g;
    int src, sink;
    vector<T> h;
    vector<T> dis;
    vector<pair<int, int> > prev;
    bool Dij()
    {
        dis = vector<T>(n, INF);
        dis[src] = 0;
        priority_queue<pair<T, int> > q;
        q.push({-dis[src], src});
        while(!q.empty())
        {
            auto frt = q.top();
            q.pop();
            if(frt.first != -dis[frt.second]) continue;
            int u = frt.second;
            for(int i=0; i<(int)g[u].size(); i++)
            {
                auto edge = e[g[u][i]];
                if(edge.cap > 0 && dis[edge.to] > dis[u] + h[u] + edge.cost - h[edge.to])
                {
                    dis[edge.to] = dis[u] + h[u] + edge.cost - h[edge.to];
                    prev[edge.to] = {u, g[u][i]};
                    q.push({-dis[edge.to], edge.to});
                }
            }
        }
        return dis[sink] < INF;
    }
    void spfa()
```

```
    {
        queue<int> q;
        vector<int> vis(n, 0);
        h[src] = 0;
        vis[src] = 1;
        q.push(src);
        while(!q.empty())
        {
            int u = q.front();
            q.pop();
            vis[u] = 0;
            for(int i=0; i<(int)g[u].size(); i++)
            {
                auto edge = e[g[u][i]];
                if( edge.cap > 0 && h[edge.to] > h[u] + edge.cost)
                {
                    h[edge.to] = h[u] + edge.cost;
                    if( !vis[edge.to] )
                    {
                        vis[edge.to] = 1;
                        q.push(edge.to);
                    }
                }
            }
        }
        return;
    }
public:
    mcmf(int _n, int _src, int _sink)
    {
        n = _n;
        src = _src;
        sink = _sink;
        g = vector<vector<int> >(n);
        return;
    }
    void addEdge(int u, int v, int f, T c)
    {
        g[u].push_back( e.size() );
        e.push_back({v, f, c});
        g[v].push_back( e.size() );
        e.push_back({u, 0, -c});
        return;
    }
    pair<int, T> get()
    {
        prev = vector<pair<int, int> >(n);
        int maxf = 0;
        T minc = 0;
        h = vector<T>(n, 0);
        spfa();
        while( Dij() )
        {
            int flow = numeric_limits<int>::max();
            for(int i=0; i<n; i++) if(dis[i] < INF) h[i] += dis[i];
            for(int v = sink; v != src; v = prev[v].first )
            {
                flow = min(flow, e[prev[v].second].cap);
```

```
            }
            for(int v = sink; v != src; v = prev[v].first )
            {
                e[prev[v].second].cap -= flow;
                e[prev[v].second^1].cap += flow;
            }
            maxf += flow;
            minc += h[sink] * flow;
        }
        return {maxf, minc};
    }
};
void solve()
{
    int n, m, s, t;
    cin>>n>>m>>s>>t;
    s--; t--;
    auto _graph = mcmf<int>(n, s, t);
    while(m--)
    {
        int u, v, w, c;
        cin>>u>>v>>w>>c;
        u--;
        v--;
        _graph.addEdge(u, v, w, c);
    }
    auto ans = _graph.get();
    cout<<ans.first << " " << ans.second<<"\n";
    return;
}

int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);
    int T = 1;
    // cin>>T;
    while(T--)
    {
        solve();
    }
    return 0;
}
```

第 2 章　凸优化相关

目前看起来在 ICPC 区域赛几乎是每场必出。

2.1 四边形不等式相关

要讨论的是这样的一类问题

$$f(j)=\min_{i\le j} w(i,j)$$

关于某些 w 满足四边形不等式的证明：

(i) 把不等式写出来后，由于实质上只是左端点交换了一下，只与左端点有关的或者只与右端点有关的并不会改变，两边长得一样的，可以消去；常量更加可以消去了。 for example, 有时候 $w(i,j)$ 可能会被替换成 $w'(i,j)=dp[i-1]+w(i,j)$，然而 $dp[i-1]$ 这一项只与左端点有关，所以其是否满足四边形不等式并不会被改变。还有就像是 $\lambda(j-i+1)$ 这种可以被拆成左、右端点独立作用的也会被消去。

(ii) 通过一点小技巧，可以发现 $i<i+1\le j<j+1$ for $j=j_0,j_0+1$ 满足四边形不等式 $\Longrightarrow$ $i<i+1\le j<j+2$ 满足四边形不等式，同理可以如此扩展左端点。所以可以只证明左端点相差 1 与右端点相差 1 的情况。

(iii) 对于显式的、可求二阶导的下凸函数 f，$w(i,j):=f(j-i)$，证明则不需要这么麻烦。稍微推导可以发现无非是要说明对于 $a<b\le c<d$ at the same time $a+d=b+c$，which actually represent the length of the 4 mentioned intervals，we have $f(b)+f(c)\le f(a)+f(d)$。FYI, there's a special interpretation on "intersect $\le$ include", which is "equal $\le$ different"。这样的话其实就是要说明 $f(b)-f(a)\le f(d)-f(c)$，然而 $b-a=d-c$，从而就是要说明 $f(a+\Delta)-f(a)\le f(c+\Delta)-f(c)$。这几乎就是下凸性质的定义。

(iv) 决策单调性优化 dp 学习笔记 — Exber's Blog 上述性质似乎可以扩展（但没有提供证明），如果 $w(l,r)$ 满足区间包含单调性和四边形不等式，$f(x)$ 为下凸函数，则：

- $f(w(l,r))$ 满足四边形不等式
- 如果 $f(x)$ 单调不降（也就是没有触底反弹那一段），则 $f(w(l,r))$ 还满足区间包含单调性

(v) 还有一些问题，在考虑四边形不等式进行代数运算时，不妨考虑对于三个区间 $[l_1,l_2],[l_2,r_1],[r_1,r_2]$ 分别设一些变量，这样代数证明会明显一点。比如说证明

$$f(s_1+s_2)+f(s_2+s_3)\le f(s_1+s_2+s_3)+f(s_2)$$

对于区间划分型 dp，**听说**恰经过 k 条边蒙日矩阵最短路关于 k 是下凸的。

2.1.1 四边形不等式优化 DP

2157. 「POI2011 R1」避雷针 Lightning Conductor - 题目 - LibreOJ

这个情景是，下标决策点未必是要小于当前点的。这种问题只需要正向跑一边，逆向跑一边，在两者之间取更优即可。模板是解决前面标准的

- 四边形不等式指相交 $\leq$ 包含
- 等价于要决策的是最小值

注意中途运算时 sqrt 相关操作不能取整，反之则会破坏 $-\sqrt{x}$ 的下凸性（比如说相邻两项点值一样斜率就直接为 0 了）。

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e6 + 5;
int n, a[N], ans[N];
int ceilsq[N];
double f(int i, int j)
{
    return a[j] -a[i] + -sqrt(j-i);
}
void work()
{
    deque<pair<int, pair<int, int> > > q;
    for(int i=1; i<=n; i++)
    {
        if(!q.empty() && q.front().second.second < i ) q.pop_front();
        if(!q.empty() ) q.front().second.first = i;
        while( !q.empty() &&
            f(i, q.back().second.first) <= f(q.back().first, q.back().second.first) )
            q.pop_back();
        if(q.empty()) q.push_back({i, {i, n}});
        else if( f(i, n) < f(q.back().first, n) )
        {
            int l = q.back().second.first, r = n, ans = n;
            while(l<=r)
            {
                int mid = (l+r)/2;
                if( f(i, mid) < f(q.back().first, mid) )
                {
                    ans = mid;
                    r = mid-1;
                }
                else l = mid+1;
            }
            q.back().second.second = ans-1;
            q.push_back({i, {ans, n}});
        }
        ans[i] = min(ans[i], (int)floor(f( q.front().first, i)) );
    }
    return;
}
void solve()
{
    cin>>n;
    for(int i=1; i<=n; i++) cin>>a[i], ans[i] = 0;
```

```
    work();
    reverse(a+1, a+n+1);
    reverse(ans+1, ans+n+1);
    work();
    reverse(ans+1, ans+n+1);
    reverse(a+1, a+n+1);
    for(int i=1; i<=n; i++) cout<< (-ans[i]) <<"\n";
    return;
}
int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);
    int now = 0;
    for(int i=0; i<N; i++)
    {
        ceilsq[i] = now;
        if( now*now == i) now++;
    }
    int T = 1;
    // cin>>T;
    while(T--)
    {
        solve();
    }
    return 0;
}
```

2.1.2 预备节：WQS 二分

P2619 [国家集训队]Tree I - 洛谷

洛谷题解质量有点差，很多都看不出和 wqs 二分有什么关系。

这样思考，设 g_x 是恰选中 x 条白边的最小权。那么 $f(x) := g_x - \lambda x$ 就是给这 x 条白边的权值减去 λ 的惩罚代价后的权值，那么 $g_x - \lambda x$ 的最小值在什么地方取到？这等价于提前给所有白边都减去 λ，然后跑 mst。这有什么意义呢？

考虑 $(need-1, g_{need-1})$ 和 $(need, g_{need})$，当 λ 恰好等于这两个点的斜率时，如果能有下凸性，并且同取最小值的时候可以取 x 大的，这时候就会取到后面的那个作为最小值的 x 点（如果取到 $> need$ 的其实同理可以往回取）。

当 λ 大于这两个点的斜率时，$< need$ 的部分比大于等于 $\geq need$ 的部分不优。

当 λ 大于这两个点的斜率时，$< need$ 的部分比大于等于 $\geq need$ 的部分更优。

所以二分这个惩罚代价，使得 $p = \max\{pos : f(pos) = \min_x\{f(x)\}\}$ 位于 $\geq need$ 部分，那么必然通过调整使得 $f(need) = f(p)$。

貌似有道更难一点的：P5633 最小度限制生成树 - 洛谷。

P1912 [NOI2009] 诗人小G - 洛谷

题解 P1912 【[NOI2009]诗人小G】 - 洛谷专栏

```cpp
#include<bits/stdc++.h>
using namespace std;
const int N = 1e5 + 5;
int fa[N];
int father(int x)
{
    if(fa[x] == x) return x;
    return fa[x] = father(fa[x]);
}
void merge(int x, int y)
{
    x = father(x);
    y = father(y);
    fa[x] = y;
}
vector< pair<pair<int, int>, pair<int, int> > > e;
int V, E, need, val;
bool check(int d)
{
    for(int i=1; i<=V; i++) fa[i] = i;
    vector< pair<pair<int, int>, pair<int, int> > > tmpe = e;
    for(int i=1; i<=E; i++) if(tmpe[i].first.second == 0) tmpe[i].first.first += d;
    sort(tmpe.begin()+1, tmpe.end());
    int cnt = 0;
    val = 0;
    for(int i=1; i<=E; i++)
    {
        int s = tmpe[i].second.first,
            t = tmpe[i].second.second,
            c = tmpe[i].first.first,
            col = tmpe[i].first.second;
        if(father(s) == father(t)) continue;
        if(col==0) cnt++;
        val += c;
```

```
        merge(s, t);
    }
    return cnt >= need;
}
void solve()
{
    cin>>V>>E>>need;
    e = vector< pair<pair<int, int>, pair<int, int> > >(E+1);
    for(int i=1; i<=E; i++)
    {
        cin>>e[i].second.first>>e[i].second.second>>e[i].first.first>>e[i].first.second;
        e[i].second.first++;
        e[i].second.second++;
    }
    int l = -100, r = 100, ans = 666;
    while(l<=r)
    {
        int mid = (l+r)/2;
        if(check(mid))
        {
            ans = mid;
            l = mid+1;
        }
        else r = mid-1;
    }
    check(ans);
    cout<< val - need*ans <<"\n";
    return;
}
int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);
    int T = 1;
    // cin>>T;
    while(T--)
    {
        solve();
    }
    return 0;
}
```

2.1.3 WQS 二分结合决策单调性

这个属于是固定套路，区间分划问题。下面给出对应模板代码情景的题解。

考虑到

$$(\sum_{i=1}^{m}\frac{num_i}{k_i})(\sum_{i=1}^{m}k_i sum_i)\geq(\sum_{i=1}^{m}\sqrt{num_i\cdot sum_i})^2$$

因为要最小化 $\sum\limits_{i=1}^{m}\frac{num_i}{k_i}$ 所以用调整法可以说明 $(\sum\limits_{i=1}^{m}k_i sum_i)$ 会取其上界 1，这就把 $\{k\}$ 给安排了。只需要最小化右式子。再用调整法可以说明 sum 里面应该是连续的，于是可以先对 s 排序，然后按顺序做一个 dp。这实际上就转化成了一个限制区间个数的区间分拆问题

WQS 二分，引入一个惩罚参数 lambda，将问题转化为：不限制分组数，但每多一组就要付出 lambda 的代价。可以看出惩罚参数越大分组越少，所以我们大概是要求一个适当大的惩罚参数使得分组的组数不超过 m。

这里的四边形不等式貌似不能直接套结论把 $\sqrt{x}$ 剥掉直接考虑 x，因为这里就是要考虑 $w(i,j)=dp[i-1]+\sqrt{(j-i)(pre_j-pre_i)}+\lambda$ 这样的 w 取 min，而不带负号 sqrt 并非下凸而是上凸。当然这里还是只需要考虑 $\sqrt{(j-i)(pre_j-pre_i)}$ 是否满足四边形不等式。直观地理解的话，把 pre_j 替换成 j，于是 w 刚好到 $j-i$，一个凸性的临界点。然后 pre 可能会再稍微下凸一点，即考虑到 Δpre 单调不减，就让这个 w 变成下凸的了。

然后就是一个典型的区间分拆问题，这类问题**听说**只要把四边形不等式证出来了，则可推出 Monge 最短路关于经过的边数 k（即拆分组数）有下凸性。这题貌似没有说清楚能不能让 $num_i=0$，即某个区间为空，即只需要区间数 $\leq m$ 而非恰好 m。但看起来通过一点调整可以说明组数越多越好。所以关于组数应该是一个下凸的减函数。

```
#include<bits/stdc++.h>
using namespace std;
const int N = 2e5 + 5;
int n, m;
int a[N];
long double pre[N];
int cnt[N];
long double dp[N];
long double w(int i, int j, long double lamb)
{
    return dp[i-1] + sqrtl( ( pre[j] - pre[i-1] )*(j-i+1) ) + lamb;
}
bool check(long double lamb)
{
    deque<pair<int, pair<int, int> > > q;
    dp[0] = 0;
    cnt[0] = 0;
    for(int i=1; i<=n; i++)
    {
        if(!q.empty() && q.front().second.second < i) q.pop_front();
        if(!q.empty() && q.front().second.first < i) q.front().second.first = i;
        while(!q.empty() &&
            w(i, q.back().second.first, lamb)
            <= w(q.back().first, q.back().second.first, lamb) ) q.pop_back();
        if(q.empty()) q.push_back({i, {i, n}});
        else if( w(i, n, lamb) < w(q.back().first, n, lamb) )
        {
```

```
            int l = q.back().second.first, r = n, ans = n;
            while(l<=r)
            {
                int mid = (l+r)/2;
                if( w(i, mid, lamb) < w(q.back().first, mid, lamb) )
                {
                    ans = mid;
                    r = mid-1;
                }
                else l = mid+1;
            }
            q.back().second.second = ans-1;
            q.push_back({i, {ans, n}});
        }
        dp[i] = w(q.front().first, i, lamb);
        cnt[i] = cnt[q.front().first-1] +1;
    }
    return cnt[n] <= m;
}
void solve()
{
    cin>>n>>m;
    for(int i=1; i<=n; i++) cin>>a[i];
    sort(a+1, a+n+1);
    for(int i=1; i<=n; i++) pre[i] = pre[i-1] + a[i];

    long double l = 0, r = 1e14;
    for(int itr = 0; itr < 200; itr++)
    {
        long double mid = (l+r)/2;
        if( check(mid) )
        {
            r = mid;
        }
        else l = mid;
    }
    check(r);
    cout<<dp[n]-m*r<<"\n";
    return;
}
int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);
    cout<<fixed<<setprecision(15);
    int T = 1;
    // cin>>T;
    while(T--)
    {
        solve();
    }
    return 0;
}
```

第 3 章　计算几何

3.1 随机化方法

todo

3.1.1 最小公共圆

一个期望复杂度正确的随机化算法

```
#include<bits/stdc++.h>
using namespace std;
const double eps = 1e-9;
pair<double, double> getCircle(pair<double, double> p1, pair<double, double> p2, pair<double, double> p3) {
    double A = p2.first - p1.first;
    double B = p2.second - p1.second;
    double C = p3.first - p1.first;
    double D = p3.second - p1.second;
    double E = A * (p1.first + p2.first) + B * (p1.second + p2.second);
    double F = C * (p1.first + p3.first) + D * (p1.second + p3.second);
    double det = 2 * (A * D - B * C);
    if (abs(det) < eps)
    {
        return {0, 0};
    }
    double centerX = (D * E - B * F) / det;
    double centerY = (A * F - C * E) / det;
    return {centerX, centerY};
}
void solve()
{
    int n;
    cin>>n;
    vector<pair<double, double> > p(n);
    for(int i=0; i<n; i++) cin>>p[i].first>>p[i].second;
    random_shuffle(p.begin(), p.end());
    auto dis = [&](pair<double, double> x, pair<double, double> y)
    {
        return sqrt( pow(x.first - y.first, 2) + pow(x.second - y.second, 2) );
    };
    pair<double, double> O = p[0];
    double R = 0;
    for(int i=1; i<n; i++)
    {
        if(dis(p[i], O) > R + eps)
        {
            O = p[i], R = 0;
            for(int j=0; j<i; j++)
            {
                if(dis(p[j], O) > R + eps)
                {
                    O.first = (p[i].first + p[j].first)/2;
                    O.second = (p[i].second + p[j].second)/2;
                    R = dis(p[j], O);
                    for(int k=0; k<j; k++)
                    {
                        if(dis(p[k], O) > R + eps)
                        {
                            O = getCircle(p[i], p[j], p[k]);
                            R = dis(p[k], O);
                        }
                    }
                }
            }
```

```
        }
    }
    cout<<R<<"\n"<<O.first<<" "<<O.second<<"\n";
    return;
}
int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);
    cout<<fixed<<setprecision(10);
    int T = 1;
    // cin>>T;
    while(T--)
    {
        solve();
    }
    return 0;
}
```

第 4 章　数学

4.1 初等数论相关

4.1.1 找模素数 P 的原根

首先必然素数 P 必然有原根。据说可以证明最小的原根的数量级大概是 $O(P^{1/4})$，然后考虑到 Fermat 小定理给出的 $p-1$ 必然是阶的倍数，所以可以暴力枚举其因数判断阶是否小于 $p-1$。

```
#include<bits/stdc++.h>
using namespace std;
class FindPrimitiveRoot
{
private:
    int P;
    long long power_mod_P(long long x, int idx)
    {
        long long res = 1;
        for(long long now = x; idx>0; now = now*now%P, idx>>=1)
        {
            if(idx & 1) res = res*now%P;
        }
        return res;
    }
public:
    FindPrimitiveRoot(int _P)
    {
        P = _P;
        return;
    }
    int work()
    {
        if(P==2) return 1;
        vector<int> d;
        for(int i=1; i*i<=(P-1); i++)
        {
            if( (P-1)%i !=0 ) continue;
            d.push_back(i);
            if(i!=1 && i*i!=(P-1) ) d.push_back( (P-1)/i );
        } // the possible choices to be period must be the divisor of (P-1)
        for(int i=2; i<P; i++)
        {
            bool flag = 1;
            for(auto j : d)
            {
                if(power_mod_P(i, j) == 1)
                {
                    flag = 0;
                    break;
                } // P-1 isn't the least period
            }
            if(flag) return i;
        } // for prime, the least root is small, so just check from 2
        return -1; // for legal case, it's not gonna happen
    }
};
int main()
{
    return 0;
}
```

4.1.2 杜教筛求 premu

```cpp
#include<bits/stdc++.h>
using namespace std;
const int M = 2e7+5;
bool vis[M];
int mu[M], premu[M];
vector<int> prime;
map<long long, long long> S0;
long long sum(long long n)
{
    if(n<M) return premu[n];
    if(S0.count(n)) return S0[n];
    long long res = 1;
    for(long long t=2, r; t<=n; t = r+1)
    {
        r = min(n/(n/t), n);
        res -= sum(n/t)*(r-t+1);
    }
    S0[n] = res;
    return res;
}
void sieve()
{
    premu[1] = mu[1] = 1;
    for(int j=2; j<M; j++)
    {
        if(!vis[j])
        {
            prime.push_back(j);
            mu[j] = -1;
        }
        premu[j] = premu[j-1] + mu[j];
        for(auto p : prime)
        {
            if(1ll*p*j<M)
            {
                vis[p*j] = 1;
                if(j%p == 0)
                {
                    mu[p*j] = 0;
                    break;
                }
                else mu[p*j] = -mu[j];
            }
            else break;
        }
    }
    return;
}
int main()
{
    return 0;
}
```

4.2 多项式相关

4.2.1 NTT

主要的部分就是算一个特殊的矩阵乘向量得到的向量 void ntt(vector a, bool invert 因为逆矩阵几乎一样，方法通用，就写进一个函数里) 记录一下 a 的大小 n。实际上这里的 n 会在后面调用的时候提前变成 2 的幂次。

先作一个预处理，所谓蝴蝶操作。就是后面我们会把系数根据奇偶位分成两份多项式，直到拆到底层成为一个直接可以引用的下标。我们把这个下标提前放好，后面就可以迭代地自下往上了。这里先直观给出小例子

$0 \to 000 \to 000 \to 0$

$1 \to 001 \to 100 \to 4$

$2 \to 010 \to 010 \to 2$

$3 \to 011 \to 110 \to 6$

$4 \to 100 \to 001 \to 1$

$5 \to 101 \to 101 \to 5$

$6 \to 110 \to 011 \to 3$

$7 \to 111 \to 111 \to 7$

大致上是一个两两配对并 swap 的过程。我们可以只枚举小的然后去找对应大的，防止 swap 两次结果还原了。只要能做到 $O(n \log n)$ 就行了，这里的实现方法可以是：考虑我给 i 从 0 开始加实际上在干什么：从右往左找到第一个 0，变成 1，然后抹掉这个位置以右的位置。镜面地对另一个 j 从 0 开始操作，就能得到镜面的结果了。

从最短 len=2 开始往 n 合并，每次 len 倍长。假设已经计算出了 $len/2$ 的奇数项多项式和偶数项多项式。现在来计算 len 的多项式。记当前单位根是 w_{len}，实质上对于 $f(w_{len}^k)$，我们需要的是 $f(w_{len}^k) = f_1(w_{len/2}^k) + w_{len}^k f_2(w_{len/2}^k)$。对于 $k < len/2$ 我们就直接引用对应位置的点值来计算；对于 $k \geq len/2$ 的，考虑到 $w_{len/2}^k = w_{len/2}^{k-len/2}$，$w_{len}^k = -w_{len}^{k-len/2}$，可以引用对应点值，也可以在在处理 $k - len/2$ 时顺便处理了，并且这样的话就可以直接覆盖掉对应的位置而不怕后面还会被引用了。

模意义下的单位根 w_{len} 是什么意思呢？NTT 模数 p 减一，即 998244353-1 后是 $2^{23} \approx 8e6$ 的倍数，大于算法竞赛环境下 $O(n \log n)$ 复杂度允许的 n； 3 又是这个模数的原根，即第一个循环节刚好是费马小定理指出的一个循环节 $p-1$，所以我们可以指定 $w_{len} = 3^{(p-1)/len}$，并验证它符合我们需要的各种单位根性质。稍微强调一下：最主要的动机在于

$$\sum_{k=0}^{n-1} (w_n^{i-j})^k$$

应该需要在 $i = j$ 时为一个统一的常数，而在 $i \neq j$ 时为 0，即这时

$$w_n^{i-j} \neq 1 \wedge w_n^{(i-j)n} = 1$$

加入让 $w_{len} = g^{(p-1)/len}$，其中 g 不是原根的话，$g^{(i-j)(p-1)/n}$，这里 $(i-j)$ 的取值范围实际上是 $-(n-1)$ 到 $(n-1)$，就有可能撞上一个比 $p-1$ 小的循环节，让这个结果变成 1 了。这个动机实质上就是在验证这个形式的矩阵是否真的是逆矩阵。对于原根打底的，那确实是。反之，要么我们通过上面的解剖来验证伪，要么就直接考虑一下矩阵中存在完全相同的某两行，那么行列式为 0，必然不存在逆矩阵。

如果是 invert 的话实质上求逆时还会多一项公因数 $1/n$（其实就是我们先前提到的统一常数），所以每项结果再乘一下就好了。

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

```
const int MOD = 998244353, G = 3;
class NTT
{
private:
	static long long power(long long x, int idx)
	{
		if(idx==0) return 1;
		auto res = power(x, idx/2);
		res = res*res%MOD;
		if(idx&1) res = res*x%MOD;
		return res;
	}
	static void ntt(vector<int> &a, bool invert)
	{

		int len = a.size();
		for(int now = 0, i = 1; i<len-1; i++)
		{
			int hbit = len/2;
			while( (now & hbit) )
			{
				now ^= hbit;
				hbit>>=1;
			}
			now^=hbit;
			if(now > i) swap(a[now], a[i]);
		}
		for(int l = 2; l<=len; l<<=1)
		{
			int wl = power(G, (MOD-1)/l);
			if(invert) wl = power(wl, MOD-2);
			for(int i=0; i<len; i+=l)
			{
				for(int k=0, now = 1; k<l/2; k++, now = 1ll*now*wl%MOD)
				{
					int u = a[i+k], v = 1ll*a[i+k+l/2]*now%MOD;
					a[i+k] = (u+v)%MOD;
					a[i+k+l/2] = (u-v+MOD)%MOD;
				}
			}
		}

		if(invert)
		{
			int ivn = power(len, MOD-2);
			for(int i=0; i<len; i++) a[i] = 1ll*ivn*a[i]%MOD;
		}

		return;
	}

public:
	static vector<int> multiply(vector<int> vec1, vector<int> vec2)
	{
		vector<int> res;
		int len = 1;
		while(len < (int)(vec1.size() + vec2.size()-1) ) len<<=1;
```

```
                vec1.resize(len, 0);
                vec2.resize(len, 0);
                ntt(vec1, 0);
                ntt(vec2, 0);
                for(int i=0; i<len; i++)
                {
                        res.push_back(1ll*vec1[i]*vec2[i]%MOD);
                }
                ntt(res, 1);
                return res;
        }
};

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