# Team Notebook

**IUT_Serenity**

## Contents

# 1 MATH

## 1.1 FFT

```cpp
const double PI = acos(-1);
void fft(vector<complex<double>> & a, bool invert)
{
    int n = a.size();
    for(int i = 1, j = 0; i < n; i++)
    {
        int bit = n >> 1;
        for(; j & bit; bit >>= 1) j ^= bit;
        j ^= bit;
        if(i < j) swap(a[i], a[j]);
    }
    for(int len = 2; len <= n; len <<= 1)
    {
        double ang = 2 * PI / len * (invert ? -1 : 1);
        complex<double> wlen(cos(ang), sin(ang));
        for(int i = 0; i < n; i += len)
        {
            complex<double> w(1);
            for(int j = 0; j < len / 2; j++)
            {
                complex<double> u = a[i+j], v = a[i+j+len/2] * w;
                a[i+j] = u + v;
                a[i+j+len/2] = u - v;
                w *= wlen;
            }
        }
    }
    if(invert)
    {
        for(complex<double> & x : a) x /= n;
    }
}

vector<int> multiply(vector<int> const& a, vector<int> const& b)
{
    vector<complex<double>> fa(a.begin(), a.end()), fb(b.begin(), b.end());
    int n = 1;
    while (n < a.size() + b.size()) n <<= 1;
    fa.resize(n);
    fb.resize(n);
    fft(fa, false);
    fft(fb, false);
    for (int i = 0; i < n; i++) fa[i] *= fb[i];
    fft(fa, true);
    vector<int> result(n);
    for (int i = 0; i < n; i++) result[i] = round(fa[i].real());
    return result;
}
```

## 1.2 Pollard Rho and Miller Rabin

```cpp
unsigned LL mult(unsigned LL a, unsigned LL b, unsigned LL mod){
    return (__int128)a * b % mod;
}

unsigned LL f(unsigned LL x, unsigned LL c, unsigned LL mod) {
    return (mult(x, x, mod) + c) % mod;
}

unsigned LL rho(unsigned LL n, unsigned LL x0=2, unsigned LL c=1) {
    LL x = x0;
    LL y = x0;
    unsigned LL g = 1;
    while (g == 1) {
        x = f(x, c, n);
        y = f(y, c, n);
        y = f(y, c, n);
        g = __gcd(abs(x - y), (LL) n);
    }
    return g;
}

using u64 = uint64_t;
using u128 = __uint128_t;

u64 binpower(u64 base, u64 e, u64 mod) {
    u64 result = 1;
    base %= mod;
    while (e) {
        if (e & 1)
            result = (u128)result * base % mod;
        base = (u128)base * base % mod;
        e >>= 1;
    }
    return result;
}

bool check_composite(u64 n, u64 a, u64 d, int s) {
    u64 x = binpower(a, d, n);
    if (x == 1 || x == n - 1)
        return false;
    for (int r = 1; r < s; r++) {
        x = (u128)x * x % n;
        if (x == n - 1)
            return false;
    }
    return true;
};

bool MillerRabin(u64 n) { // returns true if n is prime, else returns false.
    if (n < 2)
        return false;

    int r = 0;
    u64 d = n - 1;
    while ((d & 1) == 0) {
        d >>= 1;
        r++;
    }

    for (int a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {
        if (n == a)
            return true;
        if (check_composite(n, a, d, r))
            return false;
    }
    return true;
}
```

## 1.3 Euler's Totient

```cpp
int phi[1000002];
bool mark[1000002];

void seive_phi(int n) {
  iota(phi, phi + n + 1, 0);
  mark[1] = 1;
  for(int i = 2; i <= n; i++) {
    if(not mark[i]) {
      for(int j = i; j <= n; j += i) {
        mark[j] = 1;
        phi[j] = (phi[j] / i) * (i - 1);
      }
    }
```

```
  }
}
```

### 1.4 Extended GCD

```
int egcd(int a, int b, int &x, int &y) {
  if(a == 0) {
    x = 0; y = 1;
    return b;
  }

  int x1, y1;
  int d = egcd(b % a, a, x1, y1);
  x = y1 - (b / a)*x1;
  y = x1;

  return d;
}
```

### 1.5 Linear Sieve

```
const int N = 10000000;
vector<int> lp(N+1);
vector<int> pr;

for (int i=2; i <= N; ++i) {
    if (lp[i] == 0) {
        lp[i] = i;
        pr.push_back(i);
    }
    for (int j=0; j < (int)pr.size() && pr[j]
    <= lp[i] && i*pr[j] <= N; ++j) {
        lp[i * pr[j]] = pr[j];
    }
}
```

## 2 GRAPH

### 2.1 LCA

```
struct LCA {
    vector<int> height, euler, first, segtree;
    vector<bool> visited;
    int n;

    LCA(vector<vector<int>> &adj, int root = 0)
     {
        n = adj.size();
        height.resize(n);
        first.resize(n);
        euler.reserve(n * 2);
        visited.assign(n, false);
        dfs(adj, root);
        int m = euler.size();
        segtree.resize(m * 4);
        build(1, 0, m - 1);
    }

    void dfs(vector<vector<int>> &adj, int node
    , int h = 0) {
        visited[node] = true;
        height[node] = h;
        first[node] = euler.size();
        euler.push_back(node);
        for (auto to : adj[node]) {
            if (!visited[to]) {
                dfs(adj, to, h + 1);
                euler.push_back(node);
            }
        }
    }

    void build(int node, int b, int e) {
        if (b == e) {
            segtree[node] = euler[b];
        } else {
            int mid = (b + e) / 2;
            build(node << 1, b, mid);
            build(node << 1 | 1, mid + 1, e);
            int l = segtree[node << 1], r =
    segtree[node << 1 | 1];
            segtree[node] = (height[l] < height
    [r]) ? l : r;
        }
    }

    int query(int node, int b, int e, int L,
    int R) {
        if (b > R || e < L)
            return -1;
        if (b >= L && e <= R)
            return segtree[node];
        int mid = (b + e) >> 1;

        int left = query(node << 1, b, mid, L,
    R);
        int right = query(node << 1 | 1, mid +
    1, e, L, R);
        if (left == -1) return right;
        if (right == -1) return left;
        return height[left] < height[right] ?
    left : right;
    }

    int lca(int u, int v) {
        int left = first[u], right = first[v];
        if (left > right)
            swap(left, right);
        return query(1, 0, euler.size() - 1,
    left, right);
    }
};
```

### 2.2 Articulation Point

```
// node number starts from 0
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of
    graph

vector<bool> visited;
vector<int> tin, low;
int timer;

void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    int children=0;
    for (int to : adj[v]) {
        if (to == p) continue;
        if (visited[to]) {
            low[v] = min(low[v], tin[to]);
        } else {
            dfs(to, v);
            low[v] = min(low[v], low[to]);
            if (low[to] >= tin[v] && p!=-1)
                IS_CUTPOINT(v);
            ++children;
        }
    }
    if(p == -1 && children > 1)
        IS_CUTPOINT(v);
}

void find_cutpoints() {
    timer = 0;
```

```
    visited.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs (i);
    }
}
```

## 2.3 Bellman Ford

```
struct edge {
  int u, v, w;
};

vector<edge> edges;
int dist[102];
int n = 100;

void bellman_ford(int s) {
  fill(dist + 1, dist + n + 1, 1e9);
  dist[s] = 0;
  for(int i = 1; i < n; i++) {
    for(auto e : edges) {
      if(dist[e.v] > dist[e.u] + e.w) {
        dist[e.v] = dist[e.u] + e.w;
      }
    }
  }
}
```

## 2.4 Bridge

```
// node number starts from 0
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of
    graph

vector<bool> visited;
vector<int> tin, low;
int timer;

void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    for (int to : adj[v]) {
        if (to == p) continue;
        if (visited[to]) {
            low[v] = min(low[v], tin[to]);
        } else {
            dfs(to, v);
            low[v] = min(low[v], low[to]);
            if (low[to] > tin[v])
                IS_BRIDGE(v, to);
        }
    }
}

void find_bridges() {
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs(i);
    }
}
```

## 2.5 Dijkstra

```
const int INF = 1000000000;
vector<vector<pair<int, int>>> adj;

void dijkstra(int s, vector<int> & d, vector<
    int> & p) {
    int n = adj.size();
    d.assign(n, INF);
    p.assign(n, -1);

    d[s] = 0;
    using pii = pair<int, int>;
    priority_queue<pii, vector<pii>, greater<
    pii>> q;
    q.push({0, s});
    while (!q.empty()) {
        int v = q.top().second;
        int d_v = q.top().first;
        q.pop();
        if (d_v != d[v])
            continue;

        for (auto edge : adj[v]) {
            int to = edge.first;
            int len = edge.second;

            if (d[v] + len < d[to]) {
                d[to] = d[v] + len;
                p[to] = v;
                q.push({d[to], to});
            }
        }
    }
}
```

## 2.6 Finding Cycle

```
int n;
vector<vector<int>> adj;
vector<char> color;
vector<int> parent;
int cycle_start, cycle_end;

bool dfs(int v) {
    color[v] = 1;
    for (int u : adj[v]) {
        if (color[u] == 0) {
            parent[u] = v;
            if (dfs(u))
                return true;
        } else if (color[u] == 1) {
            cycle_end = v;
            cycle_start = u;
            return true;
        }
    }
    color[v] = 2;
    return false;
}

void find_cycle() {
    color.assign(n, 0);
    parent.assign(n, -1);
    cycle_start = -1;

    for (int v = 0; v < n; v++) {
        if (color[v] == 0 && dfs(v))
            break;
    }

    if (cycle_start == -1) {
        cout << "Acyclic" << endl;
    } else {
        vector<int> cycle;
        cycle.push_back(cycle_start);
        for (int v = cycle_end; v !=
    cycle_start; v = parent[v])
            cycle.push_back(v);
```

```
        cycle.push_back(cycle_start);
        reverse(cycle.begin(), cycle.end());

        cout << "Cycle found: ";
        for (int v : cycle)
            cout << v << " ";
        cout << endl;
    }
}
```

## 2.7 Floyd Warshall

```
//d[i][j] = min dist between i and j. Initially d[i][j] = w[i][j]
for (int k = 0; k < n; ++k) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (d[i][k] < INF && d[k][j] < INF)
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
        }
    }
}
```

## 2.8 Prim's MST

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

bool vis[10002];
vector<PII> adj[10002];

LL MST(int i) {
  LL ans = 0;
  PII e;
  priority_queue<PII, vector<PII>, greater<PII >> pq;
  pq.emplace(0, i);
  while(pq.size()) {
    e = pq.top();
    pq.pop();
    if(vis[e.second]) continue;

    vis[e.second] = true;
    ans += e.first;
    for(auto [w, v] : adj[e.second]) {
      if(not vis[v])
        pq.emplace(w, v);
    }
  }

  return ans;
}
```

## 2.9 Kruskal's MST

```
LL MST(int n, vector<array<int, 3>> &edges) {
  struct DSU {
    vector<int> parent;
    vector<int> Size;

    DSU(int n):parent(vector<int>(n)), Size(vector<int>(n, 1)) { iota(all(parent), 0); }

    int root(int i) { return parent[i] == i ? i : parent[i] = root(parent[i]); }

    void merge(int u, int v) {
      u = root(u); v = root(v);
      if(u == v) return;
      if(Size[u] < Size[v]) swap(u, v);
      parent[v] = u;
      Size[u] += Size[v];
    }
  };
  DSU dsu(n);
  LL ans = 0;
  priority_queue<array<int, 3>, vector<array<int, 3>>, greater<array<int, 3>>> pq;
  for(auto &i : edges) pq.emplace(i);

  for(array<int, 3> a; --n and pq.size(); ) {
    a = pq.top();
    pq.pop();
    if(dsu.root(a[1]) == dsu.root(a[2])) {
      n++;
      continue;
    }
    ans += a[0];
    dsu.merge(a[1], a[2]);
  }

  return ans;
}
```

## 2.10 SCC

```
vector<vector<int>> adj, adj_rev;
vector<bool> used;
vector<int> order, component;

void dfs1(int v) {
    used[v] = true;

    for (auto u : adj[v])
        if (!used[u])
            dfs1(u);

    order.push_back(v);
}

void dfs2(int v) {
    used[v] = true;
    component.push_back(v);

    for (auto u : adj_rev[v])
        if (!used[u])
            dfs2(u);
}

int main() {
    int n;
    // ... read n ...

    for (;;) {
        int a, b;
        // ... read next directed edge (a,b) ...
        adj[a].push_back(b);
        adj_rev[b].push_back(a);
    }

    used.assign(n, false);

    for (int i = 0; i < n; i++)
        if (!used[i])
            dfs1(i);

    used.assign(n, false);
    reverse(order.begin(), order.end());

    for (auto v : order)
        if (!used[v]) {
            dfs2 (v);

            // ... processing next component ...

```

```
            component.clear();
        }
}
```

### 2.11 Topsort with DFS

```
int n; // number of vertices
vector<vector<int>> adj; // adjacency list of graph
vector<bool> visited;
vector<int> ans;

void dfs(int v) {
    visited[v] = true;
    for (int u : adj[v]) {
        if (!visited[u])
            dfs(u);
    }
    ans.push_back(v);
}

void topological_sort() {
    visited.assign(n, false);
    ans.clear();
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs(i);
    }
    reverse(ans.begin(), ans.end());
}
```

### 2.12 Topsort with Indegree

## 3 DATA STRUCTURE

### 3.1 Seg Tree

```
int n, t[4*MAXN];
void build(int a[], int v, int tl, int tr) {
    if (tl == tr) {
        t[v] = a[tl];
    } else {
        int tm = (tl + tr) / 2;
        build(a, v*2, tl, tm);
        build(a, v*2+1, tm+1, tr);
        t[v] = t[v*2] + t[v*2+1];
    }
}
int sum(int v, int tl, int tr, int l, int r) {
    if (l > r)
        return 0;
    if (l == tl && r == tr) {
        return t[v];
    }
    int tm = (tl + tr) / 2;
    return sum(v*2, tl, tm, l, min(r, tm))
           + sum(v*2+1, tm+1, tr, max(l, tm+1), r);
}
void update(int v, int tl, int tr, int pos, int new_val) {
    if (tl == tr) {
        t[v] = new_val;
    } else {
        int tm = (tl + tr) / 2;
        if (pos <= tm)
            update(v*2, tl, tm, pos, new_val);
        else
            update(v*2+1, tm+1, tr, pos, new_val);
        t[v] = t[v*2] + t[v*2+1];
    }
}
```

### 3.2 Lazy Seg

```
int a[MAXN],tre[4*MAXN];
void build(int a[], int v, int tl, int tr) {
    if (tl == tr) {
        tre[v] = a[tl];
    } else {
        int tm = (tl + tr) / 2;
        build(a, v*2, tl, tm);
        build(a, v*2+1, tm+1, tr);
        tre[v] = 0;
    }
}

void update(int v, int tl, int tr, int l, int r, int add) {
    if (l > r)
        return;
    if (l == tl && r == tr) {
        tre[v] += add;
    } else {
        int tm = (tl + tr) / 2;
        update(v*2, tl, tm, l, min(r, tm), add);
        update(v*2+1, tm+1, tr, max(l, tm+1), r, add);
    }
}

int get(int v, int tl, int tr, int pos) {
    if (tl == tr)
        return tre[v];
    int tm = (tl + tr) / 2;
    if (pos <= tm)
        return tre[v] + get(v*2, tl, tm, pos);
    else
        return tre[v] + get(v*2+1, tm+1, tr, pos);
}
```

### 3.3 DSU

```
vector<int> lst[MAXN];
int parent[MAXN];

void make_set(int v) {
    lst[v] = vector<int>(1, v);
    parent[v] = v;
}

int find_set(int v) {
    return parent[v];
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (lst[a].size() < lst[b].size())
            swap(a, b);
        while (!lst[b].empty()) {
            int v = lst[b].back();
            lst[b].pop_back();
            parent[v] = a;
            lst[a].push_back (v);
        }
    }
}
```

### 3.4 BIT 2D

```cpp
// 0-indexed
struct FenwickTree2D {
    vector<vector<int>> bit;
    int n, m;

    FenwickTree2D(int row, int col) : n(row), m(col)
    {
        bit.assign(row, vector<int> (col,0));
    }
    int sum(int x, int y) {
        int ret = 0;
        for (int i = x; i >= 0; i = (i & (i + 1)) - 1)
            for (int j = y; j >= 0; j = (j & (j + 1)) - 1)
                ret += bit[i][j];
        return ret;
    }

    void add(int x, int y, int delta) {
        for (int i = x; i < n; i = i | (i + 1))
            for (int j = y; j < m; j = j | (j + 1))
                bit[i][j] += delta;
    }
};
```

## 3.5 BIT

```cpp
// 1-indexing
template<typename T> struct BIT {
  int n;
  vector<T> Tree;

  BIT() {}
  BIT(int n): n(n) { Tree.assign(n, 0); }
  // Pass a 1-indexed vector
  BIT(vector<T> &a): n(a.size()) {
    Tree.assign(n, 0);
    for(int i = 1; i < n; i++)
      update(i, a[i]);
  }

  void update(int i, int val) {
    for(; i < n; i += (i & -i))
      Tree[i] += val;
  }

  T query(int i) {
    T ret = 0;
    for(; i; i -= (i & -i))
      ret += Tree[i];
    return ret;
  }

  // [l, r]
  T query(int l, int r) { return query(r) - query(l - 1); }
};
```

## 3.6 Sparse Table

```cpp
#define __lg(x) (31 - __builtin_clz(x))
// 0-based indexing, query finds in range [first, last)
template<typename T> struct sparse_table {
  int n;
  vector<T> a;
  vector<vector<T>> table;

  sparse_table(vector<T> &a) : n(a.size()), table(n, vector<T>(__lg(n) + 1)) { this->a=a; build(); }
  T query(int l, int r) {
    int d = r - l;
    T ret = INT_MAX;
    int lg = __lg(d);

    // overlapping queries
    ret = f(table[l][lg], table[r - (1<<lg)][lg]);

    // Non-overlapping queries
    for(int i = 0; i <= lg; l += ((d>>i)&1)*(1<<i), i++)
      if((d >> i) & 1)
        ret = f(ret, table[l][i]);

    return ret;
  }

private:
  T f(T p1, T p2) { return min(p1, p2); }
  void build() {
    for(int i = 0; i < n; i++) table[i][0] = a[i];
    int lg = __lg(n) + 1;
    for(int j = 1; j < lg; j++) {
      for(int i = 0; i + (1<<j) <= n; i++)
        table[i][j] = f(table[i][j - 1], table[i + (1<<(j - 1))][j - 1]);
    }
  }
};
```

# 4 DP

## 4.1 Sibling DP rearrangement

```cpp
vector<int> adj[MAXN];
int dg[MAXN][2]; // directed graph
void rearrange(int curr, int par)
{
    if (adj[curr].size() == 1) return;
    for (int i = 0; i < adj[curr].size(); i++)
    {
        if (adj[curr][i] == par)
        {
            swap(adj[curr][i], adj[curr][0]);
            break;
        }
    }
    dg[curr][0] = adj[curr][1]; // [0] is child
    rearrange(adj[curr][1], curr);
    for (int i = 2; i < adj[curr].size(); i++)
    {
        int u = adj[curr][i], v = adj[curr][i - 1];
        rearrange(u, curr);
        dg[v][1] = u; // [1] is sibling
    }
}
```

## 4.2 SOS DP Iterative

## 4.3 SOS DP Recursive

# 5 Flow and Matchings

## 5.1 Kuhn

```cpp
int n, k;
vector<vector<int>> g;
vector<int> mt;
```

```cpp
vector<bool> used;
bool try_kuhn(int v) {
    if (used[v])
        return false;
    used[v] = true;
    for (int to : g[v]) {
        if (mt[to] == -1 || try_kuhn(mt[to])) {
            mt[to] = v;
            return true;
        }
    }
    return false;
}
int main() {
    // ... reading the graph ...

    mt.assign(k, -1);
    vector<bool> used1(n, false);
    for (int v = 0; v < n; ++v) {
        for (int to : g[v]) {
            if (mt[to] == -1) {
                mt[to] = v;
                used1[v] = true;
                break;
            }
        }
    }
    for (int v = 0; v < n; ++v) {
        if (used1[v])
            continue;
        used.assign(n, false);
        try_kuhn(v);
    }

    for (int i = 0; i < k; ++i)
        if (mt[i] != -1)
            printf("%d %d\n", mt[i] + 1, i + 1)
    ;
}
```

## 5.2 Dinic

```cpp
int n, m;
ll adj[501][501], oadj[501][501];
ll flow[501];
bool V[501];
int pa[501];
bool reachable() {
    memset(V, false, sizeof(V));
    queue<int> Q; Q.push(1); V[1]=1;
    while(!Q.empty()) {
        int i=Q.front(); Q.pop();
        for(int j = 1; i <= n; i++)
            if (adj[i][j] && !V[j])
                V[j]=1, pa[j]=i, Q.push(j);
    }
    return V[n];
}
int main() {
    cin >> n >> m;
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= n; j++)
            adj[i][j] = 0;
    for(int i = 0; i < m; i++) {
        ll a,b,c; cin >> a >> b >> c;
        adj[a][b] += c;
    }
    int v, u;
    ll maxflow = 0;
    while(reachable()) {
        ll flow = 1e18;
        for (v=n; v!=1; v=pa[v]) {
            u = pa[v];
            flow = min(flow, adj[u][v]);
        }
        maxflow += flow;
        for (v=n; v!=1; v=pa[v]) {
            u = pa[v];
            adj[u][v] -= flow;
            adj[v][u] += flow;
        }
    }
    cout << maxflow << '\n';
}
```

# 6 Geometry

## 6.1 Convex Hull

```cpp
struct point
{
    double x, y;
};
bool operator<(point a, point b)
{
    return ((a.x < b.x )|| (a.x == b.x && a.y < b.y));
}
point reff;
double dist(point a, point b)
{
    return ((a.x-b.x) * (a.x-b.x) + (a.y-b.y)*(a.y-b.y));
}
double area(point a, point b,  point c)
{
    return ((b.x - a.x) * (c.y - b.y) - (c.x - b.x) * (b.y - a.y));
}
bool cmp(point a, point b)
{
    if(area(reff, a, b) != area(reff, b,a))
    return area(reff, a, b) > 0;
    return dist(reff, a) < dist(reff, b);
}
vector<point> convex_hull(const vector<point> & given)
{
    set<point> st;
    vector<point> v;
    for(auto p : given) st.insert(p); /// selecting unique points
    for(auto p: st) v.push_back(p);
    st.clear();
    reff = {1e9,1e9};
    for(auto p: v)
        reff = (p.y < reff.y || p.y == reff.y && p.x < reff.x ) ? p : reff;
    sort(all(v), cmp);
    vector<point> hull;
    if(v.size()) hull.push_back(v[0]);
    for(int i = 1; i < v.size(); i++)
    {
        while(hull.size() > 1)
            if(area(hull[hull.size()-2], hull.back(), v[i]) <= 0) /// Counter Clockwise Convex hull
                hull.pop_back();
            else break;
        hull.push_back(v[i]);
    }
    return hull;
}
```

## 6.2 2D Geo

## 6.3 Closest Pair

```cpp
struct Point
```

```
{
    int x, y;
};
int compareX(const void* a, const void* b)
{
    Point *p1 = (Point *)a, *p2 = (Point *)b;
    return (p1->x != p2->x) ? (p1->x - p2->x) :
     (p1->y - p2->y);
}
int compareY(const void* a, const void* b)
{
    Point *p1 = (Point *)a, *p2 = (Point *)b;
    return (p1->y != p2->y) ? (p1->y - p2->y) :
     (p1->x - p2->x);
}
float dist(Point p1, Point p2)
{
    return sqrt( (p1.x - p2.x)*(p1.x - p2.x) +
                 (p1.y - p2.y)*(p1.y - p2.y)
           );
}
float bruteForce(Point P[], int n)
{
    float min = FLT_MAX;
    for (int i = 0; i < n; ++i)
        for (int j = i+1; j < n; ++j)
            if (dist(P[i], P[j]) < min)
                min = dist(P[i], P[j]);
    return min;
}
float min(float x, float y)
{
    return (x < y)? x : y;
}
float stripClosest(Point strip[], int size,
    float d)
{
    float min = d;
    for (int i = 0; i < size; ++i)
        for (int j = i+1; j < size && (strip[j
    ].y - strip[i].y) < min; ++j)
            if (dist(strip[i],strip[j]) < min)
                min = dist(strip[i], strip[j]);

    return min;
}
float closestUtil(Point Px[], Point Py[], int n
    )
{
    if (n <= 3)
        return bruteForce(Px, n);
    int mid = n/2;
    Point midPoint = Px[mid];.
    Point Pyl[mid];
    Point Pyr[n-mid];
    int li = 0, ri = 0;
    for (int i = 0; i < n; i++)
    {
    if ((Py[i].x < midPoint.x || (Py[i].x ==
    midPoint.x && Py[i].y < midPoint.y)) && li<
    mid)
        Pyl[li++] = Py[i];
    else
        Pyr[ri++] = Py[i];
    }
    float dl = closestUtil(Px, Pyl, mid);
    float dr = closestUtil(Px + mid, Pyr, n-mid
    );
    float d = min(dl, dr);
    Point strip[n];
    int j = 0;
    for (int i = 0; i < n; i++)
        if (abs(Py[i].x - midPoint.x) < d)
            strip[j] = Py[i], j++;
    return stripClosest(strip, j, d);
}

float closest(Point P[], int n)
{
    Point Px[n];
    Point Py[n];
    for (int i = 0; i < n; i++)
    {
        Px[i] = P[i];
        Py[i] = P[i];
    }

    qsort(Px, n, sizeof(Point), compareX);
    qsort(Py, n, sizeof(Point), compareY);
    return closestUtil(Px, Py, n);
}
int main()
{
    Point P[] = {{2, 3}, {12, 30}, {40, 50},
    {5, 1}, {12, 10}, {3, 4}};
    int n = sizeof(P) / sizeof(P[0]);
    cout << "The smallest distance is " <<
    closest(P, n);
    return 0;
}
```

## 6.4 Line Intersection

```
struct Point
{
    int x;
    int y;
};
bool onSegment(Point p, Point q, Point r)
{
    if (q.x <= max(p.x, r.x) && q.x >= min(p.x,
     r.x) && q.y <= max(p.y, r.y) && q.y >= min
    (p.y, r.y))
        return true;
    return false;
}
int orientation(Point p, Point q, Point r)
{
    int val = (q.y - p.y) * (r.x - q.x) - (q.x
    - p.x) * (r.y - q.y);
    if (val == 0) return 0; // collinear
    return (val > 0)? 1: 2; // clock or
    counterclock wise
}
bool doIntersect(Point p1, Point q1, Point p2,
    Point q2)
{
    int o1 = orientation(p1, q1, p2);
    int o2 = orientation(p1, q1, q2);
    int o3 = orientation(p2, q2, p1);
    int o4 = orientation(p2, q2, q1);
    if (o1 != o2 && o3 != o4) return true;
    if (o1 == 0 && onSegment(p1, p2, q1))
    return true;
    if (o2 == 0 && onSegment(p1, q2, q1))
    return true;
    if (o3 == 0 && onSegment(p2, p1, q2))
    return true;
    if (o4 == 0 && onSegment(p2, q1, q2))
    return true;
    return false;
}
```

# 7 String

## 7.1 Hashing

```
namespace Hashing {
    #define ff first
```

```
    #define ss second
    const PLL M = {1e9+7, 1e9+9};          ///
    Should be large primes
    const LL base = 1259;                  ///
    Should be larger than alphabet size
    const int N = 1e6+7;                   ///
    Highest length of string
    PLL operator+ (const PLL& a, LL x)     {
    return {a.ff + x, a.ss + x};}
    PLL operator- (const PLL& a, LL x)     {
    return {a.ff - x, a.ss - x};}
    PLL operator* (const PLL& a, LL x)     {
    return {a.ff * x, a.ss * x};}
    PLL operator+ (const PLL& a, PLL x)    {
    return {a.ff + x.ff, a.ss + x.ss};}
    PLL operator- (const PLL& a, PLL x)    {
    return {a.ff - x.ff, a.ss - x.ss};}
    PLL operator* (const PLL& a, PLL x)    {
    return {a.ff * x.ff, a.ss * x.ss};}
    PLL operator% (const PLL& a, PLL m)    {
    return {a.ff % m.ff, a.ss % m.ss};}
    ostream& operator<<(ostream& os, PLL hash)
    {
        return os<<"("<<hash.ff<<", "<<hash.ss
    <<")";
    }
    PLL pb[N];      ///powers of base mod M
    ///Call pre before everything
    void hashPre() {
        pb[0] = {1,1};
        for (int i=1; i<N; i++)     pb[i] = (pb
    [i-1] * base)%M;
    }
    ///Calculates hashes of all prefixes of s
    including empty prefix
    vector<PLL> hashList(string s) {
        int n = s.size();
        vector<PLL> ans(n+1);
        ans[0] = {0,0};
        for (int i=1; i<=n; i++)    ans[i] = (
    ans[i-1] * base + s[i-1])%M;
        return ans;
    }
    ///Calculates hash of substring s[l..r] (1
    indexed)
    PLL substringHash(const vector<PLL> &
    hashlist, int l, int r) {
        return (hashlist[r]+(M-hashlist[l-1])*
    pb[r-l+1])%M;
    }
    ///Calculates Hash of a string
    PLL Hash (string s) {
        PLL ans = {0,0};
        for (int i=0; i<s.size(); i++)  ans=(
    ans*base + s[i])%M;
        return ans;
    }
    ///appends c to string
    PLL append(PLL cur, char c) {
        return (cur*base + c)%M;
    }
    ///prepends c to string with size k
    PLL prepend(PLL cur, int k, char c) {
        return (pb[k]*c + cur)%M;
    }
    ///replaces the i-th (0-indexed) character
    from right from a to b;
    PLL replace(PLL cur, int i, char a, char b)
     {
        return cur + pb[i] * (M+b-a)%M;
    }
    ///Erases c from front of the string with
    size len
    PLL pop_front(PLL hash, int len, char c) {
        return (hash + pb[len-1]*(M-c))%M;
    }
    ///concatenates two strings where length of
     the right is k
    PLL concat(PLL left, PLL right, int k) {
        return (left*pb[k] + right)%M;
    }
    PLL power (const PLL& a, LL p) {
        if (p==0)   return {1,1};
        PLL ans = power(a, p/2);
        ans = (ans * ans)%M;
        if (p%2)    ans = (ans*a)%M;
        return ans;
    }
    PLL inverse(PLL a)  {
        if (M.ss == 1)  return power(a, M.ff-2)
    ;
        return power(a, (M.ff-1)*(M.ss-1)-1);
    }
    ///Erases c from the back of the string
    PLL invb = inverse({base, base});
    PLL pop_back(PLL hash, char c) {
        return ((hash-c+M)*invb)%M;
    }
    ///Calculates hash of string with size len
    repeated cnt times
    ///This is O(log n). For O(1), pre-
    calculate inverses
    PLL repeat(PLL hash, int len, LL cnt) {
        PLL mul = ((pb[len*cnt]-1+M) * inverse(
    pb[len]-1+M))%M;
        PLL ans = (hash*mul);
        if (pb[len].ff == 1)    ans.ff = hash.
    ff*cnt;
        if (pb[len].ss == 1)    ans.ss = hash.
    ss*cnt;
        return ans%M;
    }
}
```

## 7.2 KMP

```
struct KMP {
    string s;
    int n;
    vector<int> fail;
    KMP(const string &ss) {
        s = ss;
        n = s.size();
        fail.assign(n+1, 0);

        fail[0] = fail[1] = 0;

        for (int i=2; i<=n; i++) {
            fail[i] = (s[i-1] == s[0]);
            for (int j = fail[i-1]; j>0; j =
    fail[j])
                if (s[j] == s[i-1]) {
                    fail[i] = j+1;
                    break;
                }
        }
    }
    int match(string t) { ///No of matches
        int cur = 0, ans = 0;
        for (int i=0; i<t.size();) {
            if (t[i] == s[cur]) cur++, i++;
            else if (cur==0) i++;
            else cur = fail[cur];
            if (cur==n) ans++, cur = fail[cur];
        }
        return ans;
    }
    vector<vector<int>> prefixAutomaton() {
        vector<vector<int>> automaton(n+1,
    vector<int> (26, 0));
        automaton[0][s[0]-'a'] = 1;
        for (int i=1, k=0; i<=n; i++) {
```

```
            automaton[i] = automaton[k];
            if (i < n) {
                automaton[i][s[i]-'a'] = i+1;
                k = automaton[k][s[i]-'a'];
            }
        }
        return automaton;
    }
};
```

## 7.3 Z Algo

```
vector<int> z_function(string s) {
    int n = s.size();
    vector<int> z(n);
    int l = 0, r = 0;
    for (int i=1; i<n; i++) {
        if (i<=r)   z[i] = min(r-i+1, z[i-l]);
        while (i+z[i]<n && s[i+z[i]] == s[z[i
    ]])    z[i]++;
        if (i+z[i]-1>r)   l = i, r = i+z[i]-1;
    }
    z[0] = s.size();
    return z;
}
```

## 7.4 Aho Corasick

```
namespace Aho {
    const int N = 1e6+7;        ///Number of
    characters in dictionary
    const int K = 26;           ///Alphabet
    size
    int nxt[N][K];              ///Children
    int go[N][K];               ///automaton
    int link[N];                ///Suffix link
    bool leaf[N];               ///isLeaf
    int par[N];                 ///Parent
    char ch[N];                 ///character of
     incoming edge
    int ex[N];                  ///exit link
    int sz;
    void init() {
        memset(nxt, -1, sizeof nxt);
        memset(go, -1, sizeof go);
        memset(link, -1, sizeof link);
        memset(leaf, 0, sizeof leaf);
        memset(ex, -1, sizeof ex);
        sz = 0;
    }
    void addString(const string &s) {
        int cur = 0;
        for (char c: s) {
            int cc = c-'a';
            if (nxt[cur][cc] == -1) {
                nxt[cur][cc] = ++sz;
                ch[sz] = c;
                par[sz] = cur;
            }
            cur = nxt[cur][cc];
        }
        leaf[cur] = 1;
    }
    int Go(int v, char ch);
    ///Amortized O(1)
    int getlink(int v) {
        if (link[v] != -1)  return link[v];
        if (v==0 || par[v] == 0)    return link
    [v] = 0;
        else return link[v] = Go(getlink(par[v
    ]), ch[v]);
    }
    ///Amortized O(1)
    int Go (int v, char c) {
        int cc = c- 'a';
        if (go[v][cc] != -1)     return go[v][
    cc];
        if (nxt[v][cc] != -1)    return go[v][
    cc] = nxt[v][cc];
        else return go[v][cc] = (v ? Go(getlink
    (v), c) : 0);
    }
    ///Amortized O(1)
    int exitlink(int v) {
        if (ex[v] != -1)             return ex[
    v];
        int nxt = getlink(v);
        if (nxt==0 || leaf[nxt])     return ex[
    v] = nxt;
        return ex[v] = exitlink(nxt);
    }
    ///returns number of matches (including
    multiple matches)
    ///O(no of matches + length of s)
    int match(string s) {
        int cur = 0;
        int ans = 0;
        for (auto c: s) {
            cur = Go(cur, c);
            int e = (leaf[cur] ? cur : exitlink
    (cur));
            while (e)
                ans++,
                e = exitlink(e);
        }
        return ans;
    }
}
int main() {
    Aho::init();
    Aho::addString("banana");
    Aho::addString("ban");
    Aho::addString("nana");
    Aho::addString("anachor");
    Aho::addString("ana");

    cout<<Aho::match("ban")<<endl;           //
    /1
    cout<<Aho::match("banana")<<endl;        //
    /5
    cout<<Aho::match("bananachor")<<endl;    //
    /6
    cout<<Aho::match("ananana")<<endl;       //
    /5
    cout<<Aho::match("ba")<<endl;            //
    /0
    cout<<Aho::match("anachor")<<endl;       //
    /2
}
```

## 7.5 Trie

# 8 MISC

## 8.1 Ordered Set

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

template<typename T>
using ordered_set =  tree<T, null_type,less<T>,
     rb_tree_tag,
    tree_order_statistics_node_update>;
```