

Team Notebook

IUT_Serenity

Contents

1 MATH	2	4 DP	7
1.1 FFT	2	4.1 Sibling DP rearrangement	7
1.2 Pollard Rho and Miller Rabin	2	4.2 SOS DP Iterative	7
1.3 Euler's Totient	2	4.3 SOS DP Recursive	7
1.4 Extended GCD	3	5 Flow and Matchings	7
1.5 Linear Sieve	3	5.1 Kuhn	7
2 GRAPH	3	5.2 Dinic	8
2.1 LCA	3	6 Geometry	8
2.2 Articulation Point	3	6.1 Convex Hull	8
2.3 Bellman Ford	4	6.2 2D Geo	8
2.4 Bridge	4	6.3 Closest Pair	8
2.5 Dijkstra	4	6.4 Line Intersection	9
2.6 Finding Cycle	4	7 String	9
2.7 Floyd Warshall	5	7.1 Hashing	9
2.8 Prim's MST	5	7.2 KMP	10
2.9 Kruskal's MST	5	7.3 Z Algo	11
2.10 SCC	5	7.4 Aho Corasick	11
2.11 Topsort with DFS	6	7.5 Trie	11
2.12 Topsort with Indegree	6	8 MISC	11
3 DATA STRUCTURE	6	8.1 Ordered Set	11
3.1 Seg Tree	6		
3.2 Lazy Seg	6		
3.3 DSU	6		
3.4 BIT 2D	6		
3.5 BIT	7		
3.6 Sparse Table	7		

1 MATH

1.1 FFT

```

1  const double PI = acos(-1);
2  void fft(vector<complex<double>> & a, bool
   invert)
3  {
4      int n = a.size();
5      for(int i = 1, j = 0; i < n; i++)
6      {
7          int bit = n >> 1;
8          for(; j & bit; bit >>= 1) j ^= bit;
9          j ^= bit;
10         if(i < j) swap(a[i], a[j]);
11     }
12     for(int len = 2; len <= n; len <= 1)
13     {
14         double ang = 2 * PI / len * (invert ?
-1 : 1);
15         complex<double> wlen(cos(ang), sin(ang)
);
16         for(int i = 0; i < n; i += len)
17         {
18             complex<double> w(1);
19             for(int j = 0; j < len / 2; j++)
20             {
21                 complex<double> u = a[i+j], v =
a[i+j+len/2] * w;
22                 a[i+j] = u + v;
23                 a[i+j+len/2] = u - v;
24                 w *= wlen;
25             }
26         }
27     }
28     if(invert)
29     {
30         for(complex<double> & x : a) x /= n;
31     }
32 }
33
34 vector<int> multiply(vector<int> const& a,
   vector<int> const& b)
35 {
36     vector<complex<double>> fa(a.begin(), a.end
()), fb(b.begin(), b.end());
37     int n = 1;
38     while (n < a.size() + b.size()) n <= 1;
39     fa.resize(n);
40     fb.resize(n);
41     fft(fa, false);

```

```

42     fft(fb, false);
43     for (int i = 0; i < n; i++) fa[i] *= fb[i];
44     fft(fa, true);
45     vector<int> result(n);
46     for (int i = 0; i < n; i++) result[i] =
round(fa[i].real());
47     return result;
48 }

```

1.2 Pollard Rho and Miller Rabin

```

1  unsigned LL mult(unsigned LL a, unsigned LL b,
   unsigned LL mod){
2      return (__int128)a * b % mod;
3  }
4
5  unsigned LL f(unsigned LL x, unsigned LL c,
   unsigned LL mod) {
6      return (mult(x, x, mod) + c) % mod;
7  }
8
9  unsigned LL rho(unsigned LL n, unsigned LL x0
=2, unsigned LL c=1) {
10     LL x = x0;
11     LL y = x0;
12     unsigned LL g = 1;
13     while (g == 1) {
14         x = f(x, c, n);
15         y = f(y, c, n);
16         y = f(y, c, n);
17         g = __gcd(abs(x - y), (LL) n);
18     }
19     return g;
20 }
21
22 using u64 = uint64_t;
23 using u128 = __uint128_t;
24
25 u64 binpower(u64 base, u64 e, u64 mod) {
26     u64 result = 1;
27     base %= mod;
28     while (e) {
29         if (e & 1)
30             result = (u128)result * base % mod;
31         base = (u128)base * base % mod;
32         e >>= 1;
33     }
34     return result;
35 }
36

```

```

37 bool check_composite(u64 n, u64 a, u64 d, int s
) {
38     u64 x = binpower(a, d, n);
39     if (x == 1 || x == n - 1)
40         return false;
41     for (int r = 1; r < s; r++) {
42         x = (u128)x * x % n;
43         if (x == n - 1)
44             return false;
45     }
46     return true;
47 };
48
49 bool MillerRabin(u64 n) { // returns true if n
   is prime, else returns false.
50     if (n < 2)
51         return false;
52
53     int r = 0;
54     u64 d = n - 1;
55     while ((d & 1) == 0) {
56         d >>= 1;
57         r++;
58     }
59
60     for (int a : {2, 3, 5, 7, 11, 13, 17, 19,
23, 29, 31, 37}) {
61         if (n == a)
62             return true;
63         if (check_composite(n, a, d, r))
64             return false;
65     }
66     return true;
67 }

```

1.3 Euler's Totient

```

1  int phi[1000002];
2  bool mark[1000002];
3
4  void seive_phi(int n) {
5      iota(phi, phi + n + 1, 0);
6      mark[1] = 1;
7      for(int i = 2; i <= n; i++) {
8          if(not mark[i]) {
9              for(int j = i; j <= n; j += i) {
10                 mark[j] = 1;
11                 phi[j] = (phi[j] / i) * (i - 1);
12             }
13         }
14     }

```

```

14 }
15 }

```

1.4 Extended GCD

```

1 int egcd(int a, int b, int &x, int &y) {
2     if(a == 0) {
3         x = 0; y = 1;
4         return b;
5     }
6
7     int x1, y1;
8     int d = egcd(b % a, a, x1, y1);
9     x = y1 - (b / a)*x1;
10    y = x1;
11
12    return d;
13 }

```

1.5 Linear Sieve

```

1 const int N = 10000000;
2 vector<int> lp(N+1);
3 vector<int> pr;
4
5 for (int i=2; i <= N; ++i) {
6     if (lp[i] == 0) {
7         lp[i] = i;
8         pr.push_back(i);
9     }
10    for (int j=0; j < (int)pr.size() && pr[j]
11        <= lp[i] && i*pr[j] <= N; ++j) {
12        lp[i * pr[j]] = pr[j];
13    }

```

2 GRAPH

2.1 LCA

```

1 struct LCA {
2     vector<int> height, euler, first, segtree;
3     vector<bool> visited;
4     int n;
5

```

```

6     LCA(vector<vector<int>> &adj, int root = 0)
7     {
8         n = adj.size();
9         height.resize(n);
10        first.resize(n);
11        euler.reserve(n * 2);
12        visited.assign(n, false);
13        dfs(adj, root);
14        int m = euler.size();
15        segtree.resize(m * 4);
16        build(1, 0, m - 1);
17    }
18
19    void dfs(vector<vector<int>> &adj, int node
20        , int h = 0) {
21        visited[node] = true;
22        height[node] = h;
23        first[node] = euler.size();
24        euler.push_back(node);
25        for (auto to : adj[node]) {
26            if (!visited[to]) {
27                dfs(adj, to, h + 1);
28                euler.push_back(node);
29            }
30        }
31
32        void build(int node, int b, int e) {
33            if (b == e) {
34                segtree[node] = euler[b];
35            } else {
36                int mid = (b + e) / 2;
37                build(node << 1, b, mid);
38                build(node << 1 | 1, mid + 1, e);
39                int l = segtree[node << 1], r =
40                    segtree[node << 1 | 1];
41                segtree[node] = (height[l] < height
42                    [r]) ? l : r;
43            }
44        }
45
46        int query(int node, int b, int e, int L,
47            int R) {
48            if (b > R || e < L)
49                return -1;
50            if (b >= L && e <= R)
51                return segtree[node];
52            int mid = (b + e) >> 1;
53
54            int left = query(node << 1, b, mid, L,
55                R);

```

```

56            int right = query(node << 1 | 1, mid +
57                1, e, L, R);
58            if (left == -1) return right;
59            if (right == -1) return left;
60            return height[left] < height[right] ?
61                left : right;
62        }
63
64        int lca(int u, int v) {
65            int left = first[u], right = first[v];
66            if (left > right)
67                swap(left, right);
68            return query(1, 0, euler.size() - 1,
69                left, right);
70        }
71    };

```

2.2 Articulation Point

```

1 // node number starts from 0
2 int n; // number of nodes
3 vector<vector<int>> adj; // adjacency list of
4     graph
5
6 vector<bool> visited;
7 vector<int> tin, low;
8 int timer;
9
10 void dfs(int v, int p = -1) {
11     visited[v] = true;
12     tin[v] = low[v] = timer++;
13     int children=0;
14     for (int to : adj[v]) {
15         if (to == p) continue;
16         if (visited[to]) {
17             low[v] = min(low[v], tin[to]);
18         } else {
19             dfs(to, v);
20             low[v] = min(low[v], low[to]);
21             if (low[to] >= tin[v] && p!=-1)
22                 IS_CUTPOINT(v);
23             ++children;
24         }
25     }
26     if(p == -1 && children > 1)
27         IS_CUTPOINT(v);
28 }
29
30 void find_cutpoints() {
31     timer = 0;

```

```

31     visited.assign(n, false);
32     tin.assign(n, -1);
33     low.assign(n, -1);
34     for (int i = 0; i < n; ++i) {
35         if (!visited[i])
36             dfs(i);
37     }
38 }

```

2.3 Bellman Ford

```

1 struct edge {
2     int u, v, w;
3 };
4
5 vector<edge> edges;
6 int dist[102];
7 int n = 100;
8
9 void bellman_ford(int s) {
10     fill(dist + 1, dist + n + 1, 1e9);
11     dist[s] = 0;
12     for(int i = 1; i < n; i++) {
13         for(auto e : edges) {
14             if(dist[e.v] > dist[e.u] + e.w) {
15                 dist[e.v] = dist[e.u] + e.w;
16             }
17         }
18     }
19 }

```

2.4 Bridge

```

1 // node number starts from 0
2 int n; // number of nodes
3 vector<vector<int>> adj; // adjacency list of
4     graph
5
6 vector<bool> visited;
7 vector<int> tin, low;
8 int timer;
9
10 void dfs(int v, int p = -1) {
11     visited[v] = true;
12     tin[v] = low[v] = timer++;
13     for (int to : adj[v]) {
14         if (to == p) continue;
15         if (visited[to]) {

```

```

15         low[v] = min(low[v], tin[to]);
16     } else {
17         dfs(to, v);
18         low[v] = min(low[v], low[to]);
19         if (low[to] > tin[v])
20             IS_BRIDGE(v, to);
21     }
22 }
23 }
24
25 void find_bridges() {
26     timer = 0;
27     visited.assign(n, false);
28     tin.assign(n, -1);
29     low.assign(n, -1);
30     for (int i = 0; i < n; ++i) {
31         if (!visited[i])
32             dfs(i);
33     }
34 }

```

2.5 Dijkstra

```

1 const int INF = 1000000000;
2 vector<vector<pair<int, int>>> adj;
3
4 void dijkstra(int s, vector<int> & d, vector<
5     int> & p) {
6     int n = adj.size();
7     d.assign(n, INF);
8     p.assign(n, -1);
9
10    d[s] = 0;
11    using pii = pair<int, int>;
12    priority_queue<pii, vector<pii>, greater<
13        pii>> q;
14    q.push({0, s});
15    while (!q.empty()) {
16        int v = q.top().second;
17        int d_v = q.top().first;
18        q.pop();
19        if (d_v != d[v])
20            continue;
21
22        for (auto edge : adj[v]) {
23            int to = edge.first;
24            int len = edge.second;
25
26            if (d[v] + len < d[to]) {
27                d[to] = d[v] + len;

```

```

26         p[to] = v;
27         q.push({d[to], to});
28     }
29 }
30 }
31 }

```

2.6 Finding Cycle

```

1 int n;
2 vector<vector<int>> adj;
3 vector<char> color;
4 vector<int> parent;
5 int cycle_start, cycle_end;
6
7 bool dfs(int v) {
8     color[v] = 1;
9     for (int u : adj[v]) {
10         if (color[u] == 0) {
11             parent[u] = v;
12             if (dfs(u))
13                 return true;
14         } else if (color[u] == 1) {
15             cycle_end = v;
16             cycle_start = u;
17             return true;
18         }
19     }
20     color[v] = 2;
21     return false;
22 }
23
24 void find_cycle() {
25     color.assign(n, 0);
26     parent.assign(n, -1);
27     cycle_start = -1;
28
29     for (int v = 0; v < n; v++) {
30         if (color[v] == 0 && dfs(v))
31             break;
32     }
33
34     if (cycle_start == -1) {
35         cout << "Acyclic" << endl;
36     } else {
37         vector<int> cycle;
38         cycle.push_back(cycle_start);
39         for (int v = cycle_end; v !=
40             cycle_start; v = parent[v])
41             cycle.push_back(v);

```

```

41     cycle.push_back(cycle_start);
42     reverse(cycle.begin(), cycle.end());
43
44     cout << "Cycle found: ";
45     for (int v : cycle)
46         cout << v << " ";
47     cout << endl;
48 }
49 }

```

2.7 Floyd Warshall

```

1 //d[i][j] = min dist between i and j. Initially
  d[i][j] = w[i][j]
2 for (int k = 0; k < n; ++k) {
3     for (int i = 0; i < n; ++i) {
4         for (int j = 0; j < n; ++j) {
5             if (d[i][k] < INF && d[k][j] < INF)
6                 d[i][j] = min(d[i][j], d[i][k]
7                     + d[k][j]);
8         }
9     }

```

2.8 Prim's MST

```

1 typedef pair<int, int> PII;
2
3 bool vis[10002];
4 vector<PII> adj[10002];
5
6 LL MST(int i) {
7     LL ans = 0;
8     PII e;
9     priority_queue<PII, vector<PII>, greater<PII>
10         >> pq;
11     pq.emplace(0, i);
12     while(pq.size()) {
13         e = pq.top();
14         pq.pop();
15         if(vis[e.second]) continue;
16
17         vis[e.second] = true;
18         ans += e.first;
19         for(auto [w, v] : adj[e.second]) {
20             if(not vis[v])
21                 pq.emplace(w, v);
22         }
23     }
24 }

```

```

22 }
23
24 return ans;
25 }

```

2.9 Kruskal's MST

```

1 LL MST(int n, vector<array<int, 3>> &edges) {
2     struct DSU {
3         vector<int> parent;
4         vector<int> Size;
5
6         DSU(int n):parent(vector<int>(n)), Size(
7             vector<int>(n, 1)) { iota(all(parent), 0);
8         }
9
10        int root(int i) { return parent[i] == i ? i
11            : parent[i] = root(parent[i]); }
12
13        void merge(int u, int v) {
14            u = root(u); v = root(v);
15            if(u == v) return;
16            if(Size[u] < Size[v]) swap(u, v);
17            parent[v] = u;
18            Size[u] += Size[v];
19        }
20    };
21    DSU dsu(n);
22    LL ans = 0;
23    priority_queue<array<int, 3>, vector<array<
24        int, 3>>, greater<array<int, 3>>> pq;
25    for(auto &i : edges) pq.emplace(i);
26
27    for(array<int, 3> a; --n and pq.size(); ) {
28        a = pq.top();
29        pq.pop();
30        if(dsu.root(a[1]) == dsu.root(a[2])) {
31            n++;
32            continue;
33        }
34        ans += a[0];
35        dsu.merge(a[1], a[2]);
36    }
37
38    return ans;
39 }

```

2.10 SCC

```

1 vector<vector<int>> adj, adj_rev;
2 vector<bool> used;
3 vector<int> order, component;
4
5 void dfs1(int v) {
6     used[v] = true;
7
8     for (auto u : adj[v])
9         if (!used[u])
10             dfs1(u);
11
12     order.push_back(v);
13 }
14
15 void dfs2(int v) {
16     used[v] = true;
17     component.push_back(v);
18
19     for (auto u : adj_rev[v])
20         if (!used[u])
21             dfs2(u);
22 }
23
24 int main() {
25     int n;
26     // ... read n ...
27
28     for (;;) {
29         int a, b;
30         // ... read next directed edge (a,b)
31         ...
32         adj[a].push_back(b);
33         adj_rev[b].push_back(a);
34     }
35
36     used.assign(n, false);
37
38     for (int i = 0; i < n; i++)
39         if (!used[i])
40             dfs1(i);
41
42     used.assign(n, false);
43     reverse(order.begin(), order.end());
44
45     for (auto v : order)
46         if (!used[v]) {
47             dfs2(v);
48             // ... processing next component
49             ...
50         }
51 }

```

```

50         component.clear();
51     }
52 }

```

2.11 Topsort with DFS

```

1  int n; // number of vertices
2  vector<vector<int>> adj; // adjacency list of
   graph
3  vector<bool> visited;
4  vector<int> ans;
5
6  void dfs(int v) {
7      visited[v] = true;
8      for (int u : adj[v]) {
9          if (!visited[u])
10             dfs(u);
11     }
12     ans.push_back(v);
13 }
14
15 void topological_sort() {
16     visited.assign(n, false);
17     ans.clear();
18     for (int i = 0; i < n; ++i) {
19         if (!visited[i])
20             dfs(i);
21     }
22     reverse(ans.begin(), ans.end());
23 }

```

2.12 Topsort with Indegree

3 DATA STRUCTURE

3.1 Seg Tree

```

1  int n, t[4*MAXN];
2  void build(int a[], int v, int tl, int tr) {
3      if (tl == tr) {
4          t[v] = a[tl];
5      } else {
6          int tm = (tl + tr) / 2;
7          build(a, v*2, tl, tm);
8          build(a, v*2+1, tm+1, tr);
9          t[v] = t[v*2] + t[v*2+1];
10     }

```

```

11 }
12 int sum(int v, int tl, int tr, int l, int r) {
13     if (l > r)
14         return 0;
15     if (l == tl && r == tr) {
16         return t[v];
17     }
18     int tm = (tl + tr) / 2;
19     return sum(v*2, tl, tm, l, min(r, tm))
20         + sum(v*2+1, tm+1, tr, max(l, tm+1),
21             r);
22 }
23 void update(int v, int tl, int tr, int pos, int
24     new_val) {
25     if (tl == tr) {
26         t[v] = new_val;
27     } else {
28         int tm = (tl + tr) / 2;
29         if (pos <= tm)
30             update(v*2, tl, tm, pos, new_val);
31         else
32             update(v*2+1, tm+1, tr, pos,
33                 new_val);
34         t[v] = t[v*2] + t[v*2+1];
35     }
36 }

```

3.2 Lazy Seg

```

1  int a[MAXN], tre[4*MAXN];
2  void build(int a[], int v, int tl, int tr) {
3      if (tl == tr) {
4          tre[v] = a[tl];
5      } else {
6          int tm = (tl + tr) / 2;
7          build(a, v*2, tl, tm);
8          build(a, v*2+1, tm+1, tr);
9          tre[v] = 0;
10     }
11 }
12
13 void update(int v, int tl, int tr, int l, int r
14     , int add) {
15     if (l > r)
16         return;
17     if (l == tl && r == tr) {
18         tre[v] += add;
19     } else {
20         int tm = (tl + tr) / 2;

```

```

21         update(v*2, tl, tm, l, min(r, tm), add)
22         ;
23         update(v*2+1, tm+1, tr, max(l, tm+1), r
24             , add);
25     }
26 }
27
28 int get(int v, int tl, int tr, int pos) {
29     if (tl == tr)
30         return tre[v];
31     int tm = (tl + tr) / 2;
32     if (pos <= tm)
33         return tre[v] + get(v*2, tl, tm, pos);
34     else
35         return tre[v] + get(v*2+1, tm+1, tr,
36             pos);
37 }

```

3.3 DSU

```

1  vector<int> lst[MAXN];
2  int parent[MAXN];
3
4  void make_set(int v) {
5      lst[v] = vector<int>(1, v);
6      parent[v] = v;
7  }
8
9  int find_set(int v) {
10     return parent[v];
11 }
12
13 void union_sets(int a, int b) {
14     a = find_set(a);
15     b = find_set(b);
16     if (a != b) {
17         if (lst[a].size() < lst[b].size())
18             swap(a, b);
19         while (!lst[b].empty()) {
20             int v = lst[b].back();
21             lst[b].pop_back();
22             parent[v] = a;
23             lst[a].push_back(v);
24         }
25     }
26 }

```

3.4 BIT 2D

```

1 // 0-indexed
2 struct FenwickTree2D {
3     vector<vector<int>>> bit;
4     int n, m;
5
6     FenwickTree2D(int row, int col) : n(row), m(col)
7     {
8         bit.assign(row, vector<int> (col, 0));
9     }
10    int sum(int x, int y) {
11        int ret = 0;
12        for (int i = x; i >= 0; i = (i & (i + 1)) - 1)
13            for (int j = y; j >= 0; j = (j & (j + 1)) - 1)
14                ret += bit[i][j];
15        return ret;
16    }
17
18    void add(int x, int y, int delta) {
19        for (int i = x; i < n; i = i | (i + 1))
20            for (int j = y; j < m; j = j | (j + 1))
21                bit[i][j] += delta;
22    }
23 };

```

3.5 BIT

```

1 // 1-indexing
2 template<typename T> struct BIT {
3     int n;
4     vector<T> Tree;
5
6     BIT() {}
7     BIT(int n): n(n) { Tree.assign(n, 0); }
8     // Pass a 1-indexed vector
9     BIT(vector<T> &a): n(a.size()) {
10        Tree.assign(n, 0);
11        for(int i = 1; i < n; i++)
12            update(i, a[i]);
13    }
14
15    void update(int i, int val) {
16        for(; i < n; i += (i & -i))
17            Tree[i] += val;
18    }
19
20    T query(int i) {

```

```

21    T ret = 0;
22    for(; i; i -= (i & -i))
23        ret += Tree[i];
24    return ret;
25    }
26
27    // [l, r]
28    T query(int l, int r) { return query(r) - query(l - 1); }
29 };

```

3.6 Sparse Table

```

1 #define __lg(x) (31 - __builtin_clz(x))
2 // 0-based indexing, query finds in range [first, last)
3 template<typename T> struct sparse_table {
4     int n;
5     vector<T> a;
6     vector<vector<T>>> table;
7
8     sparse_table(vector<T> &a) : n(a.size()),
9         table(n, vector<T>(__lg(n) + 1)) { this->a=a; build(); }
10
11    T query(int l, int r) {
12        int d = r - l;
13        T ret = INT_MAX;
14        int lg = __lg(d);
15
16        // overlapping queries
17        ret = f(table[l][lg], table[r - (1<<lg)][lg]);
18
19        // Non-overlapping queries
20        for(int i = 0; i <= lg; i += ((d>>i)&1) * (1<<i), i++)
21            if((d >> i) & 1)
22                ret = f(ret, table[l][i]);
23    }
24
25    return ret;
26    }
27
28 private:
29    T f(T p1, T p2) { return min(p1, p2); }
30    void build() {
31        for(int i = 0; i < n; i++) table[i][0] = a[i];
32        int lg = __lg(n) + 1;
33        for(int j = 1; j < lg; j++) {
34            for(int i = 0; i + (1<<j) <= n; i++)

```

```

35        table[i][j] = f(table[i][j - 1], table[i + (1<<(j - 1))][j - 1]);
36    }
37 }
38 };

```

4 DP

4.1 Sibling DP rearrangement

```

1 vector<int> adj[MAXN];
2 int dg[MAXN][2]; // directed graph
3 void rearrange(int curr, int par)
4 {
5     if (adj[curr].size() == 1) return;
6     for (int i = 0; i < adj[curr].size(); i++)
7     {
8         if (adj[curr][i] == par)
9             continue;
10        swap(adj[curr][i], adj[curr][0]);
11        break;
12    }
13    dg[curr][0] = adj[curr][1]; // [0] is child
14    rearrange(adj[curr][1], curr);
15    for (int i = 2; i < adj[curr].size(); i++)
16    {
17        int u = adj[curr][i], v = adj[curr][i - 1];
18        rearrange(u, curr);
19        dg[v][1] = u; // [1] is sibling
20    }
21 }

```

4.2 SOS DP Iterative

4.3 SOS DP Recursive

5 Flow and Matchings

5.1 Kuhn

```

1 int n, k;
2 vector<vector<int>>> g;
3 vector<int> mt;

```

```

4 vector<bool> used;
5 bool try_kuhn(int v) {
6     if (used[v])
7         return false;
8     used[v] = true;
9     for (int to : g[v]) {
10         if (mt[to] == -1 || try_kuhn(mt[to])) {
11             mt[to] = v;
12             return true;
13         }
14     }
15     return false;
16 }
17 int main() {
18     // ... reading the graph ...
19
20     mt.assign(k, -1);
21     vector<bool> used1(n, false);
22     for (int v = 0; v < n; ++v) {
23         for (int to : g[v]) {
24             if (mt[to] == -1) {
25                 mt[to] = v;
26                 used1[v] = true;
27                 break;
28             }
29         }
30     }
31     for (int v = 0; v < n; ++v) {
32         if (used1[v])
33             continue;
34         used.assign(n, false);
35         try_kuhn(v);
36     }
37
38     for (int i = 0; i < k; ++i)
39         if (mt[i] != -1)
40             printf("%d %d\n", mt[i] + 1, i + 1);
41 }

```

5.2 Dinic

```

1 int n, m;
2 ll adj[501][501], oadj[501][501];
3 ll flow[501];
4 bool V[501];
5 int pa[501];
6 bool reachable() {
7     memset(V, false, sizeof(V));
8     queue<int> Q; Q.push(1); V[1]=1;

```

```

9     while(!Q.empty()) {
10         int i=Q.front(); Q.pop();
11         for(int j = 1; i <= n; i++)
12             if (adj[i][j] && !V[j])
13                 V[j]=1, pa[j]=i, Q.push(j);
14     }
15     return V[n];
16 }
17 int main() {
18     cin >> n >> m;
19     for(int i = 1; i <= n; i++)
20         for(int j = 1; j <= n; j++)
21             adj[i][j] = 0;
22     for(int i = 0; i < m; i++) {
23         ll a,b,c; cin >> a >> b >> c;
24         adj[a][b] += c;
25     }
26     int v, u;
27     ll maxflow = 0;
28     while(reachable()) {
29         ll flow = 1e18;
30         for (v=n; v!=1; v=pa[v]) {
31             u = pa[v];
32             flow = min(flow, adj[u][v]);
33         }
34         maxflow += flow;
35         for (v=n; v!=1; v=pa[v]) {
36             u = pa[v];
37             adj[u][v] -= flow;
38             adj[v][u] += flow;
39         }
40     }
41     cout << maxflow << '\n';
42 }

```

6 Geometry

6.1 Convex Hull

```

1 struct point
2 {
3     double x, y;
4 };
5 bool operator<(point a, point b)
6 {
7     return ((a.x < b.x) || (a.x == b.x && a.y <
8         b.y));
9 }
10 point reff;

```

```

10 double dist(point a, point b)
11 {
12     return ((a.x-b.x) * (a.x-b.x) + (a.y-b.y)*(
13         a.y-b.y));
14 }
15 double area(point a, point b, point c)
16 {
17     return ((b.x - a.x) * (c.y - b.y) - (c.x -
18         b.x) * (b.y - a.y));
19 }
20 bool cmp(point a, point b)
21 {
22     if(area(reff, a, b) != area(reff, b,a))
23         return area(reff, a, b) > 0;
24     return dist(reff, a) < dist(reff, b);
25 }
26 vector<point> convex_hull(const vector<point> &
27     given)
28 {
29     set<point> st;
30     vector<point> v;
31     for(auto p : given) st.insert(p); ///
32     selecting unique points
33     for(auto p: st) v.push_back(p);
34     st.clear();
35     reff = {1e9,1e9};
36     for(auto p: v)
37         reff = (p.y < reff.y || p.y == reff.y
38             && p.x < reff.x) ? p : reff;
39     sort(all(v), cmp);
40     vector<point> hull;
41     if(v.size()) hull.push_back(v[0]);
42     for(int i = 1; i < v.size(); i++)
43     {
44         while(hull.size() > 1)
45             if(area(hull[hull.size()-2], hull.
46                 back(), v[i]) <= 0) /// Counter Clockwise
47                 Convex hull
48                 hull.pop_back();
49             else break;
50         hull.push_back(v[i]);
51     }
52     return hull;
53 }

```

6.2 2D Geo

6.3 Closest Pair

```

1 struct Point

```



```

2 {
3     int x, y;
4 };
5 int compareX(const void* a, const void* b)
6 {
7     Point *p1 = (Point *)a, *p2 = (Point *)b;
8     return (p1->x != p2->x) ? (p1->x - p2->x) :
9         (p1->y - p2->y);
10 }
11 int compareY(const void* a, const void* b)
12 {
13     Point *p1 = (Point *)a, *p2 = (Point *)b;
14     return (p1->y != p2->y) ? (p1->y - p2->y) :
15         (p1->x - p2->x);
16 }
17 float dist(Point p1, Point p2)
18 {
19     return sqrt((p1.x - p2.x)*(p1.x - p2.x) +
20                (p1.y - p2.y)*(p1.y - p2.y));
21 }
22 float bruteForce(Point P[], int n)
23 {
24     float min = FLT_MAX;
25     for (int i = 0; i < n; ++i)
26         for (int j = i+1; j < n; ++j)
27             if (dist(P[i], P[j]) < min)
28                 min = dist(P[i], P[j]);
29 }
30 float min(float x, float y)
31 {
32     return (x < y) ? x : y;
33 }
34 float stripClosest(Point strip[], int size,
35                    float d)
36 {
37     float min = d;
38     for (int i = 0; i < size; ++i)
39         for (int j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)
40             if (dist(strip[i], strip[j]) < min)
41                 min = dist(strip[i], strip[j]);
42     return min;
43 }
44 float closestUtil(Point Px[], Point Py[], int n)
45 {
46     if (n <= 3)
47         return bruteForce(Px, n);

```

```

48     int mid = n/2;
49     Point midPoint = Px[mid];
50     Point Pyl[mid];
51     Point Pyr[n-mid];
52     int li = 0, ri = 0;
53     for (int i = 0; i < n; i++)
54     {
55         if ((Py[i].x < midPoint.x || (Py[i].x ==
56             midPoint.x && Py[i].y < midPoint.y)) && li <
57             mid)
58             Pyl[li++] = Py[i];
59         else
60             Pyr[ri++] = Py[i];
61     }
62     float dl = closestUtil(Px, Pyl, mid);
63     float dr = closestUtil(Px + mid, Pyr, n-mid);
64     float d = min(dl, dr);
65     Point strip[n];
66     int j = 0;
67     for (int i = 0; i < n; i++)
68         if (abs(Py[i].x - midPoint.x) < d)
69             strip[j++] = Py[i];
70     return stripClosest(strip, j, d);
71 }
72 float closest(Point P[], int n)
73 {
74     Point Px[n];
75     Point Py[n];
76     for (int i = 0; i < n; i++)
77     {
78         Px[i] = P[i];
79         Py[i] = P[i];
80     }
81     qsort(Px, n, sizeof(Point), compareX);
82     qsort(Py, n, sizeof(Point), compareY);
83     return closestUtil(Px, Py, n);
84 }
85 int main()
86 {
87     Point P[] = {{2, 3}, {12, 30}, {40, 50},
88                 {5, 1}, {12, 10}, {3, 4}};
89     int n = sizeof(P) / sizeof(P[0]);
90     cout << "The smallest distance is " <<
91         closest(P, n);
92 }

```

6.4 Line Intersection

```

1 struct Point
2 {
3     int x;
4     int y;
5 };
6 bool onSegment(Point p, Point q, Point r)
7 {
8     if (q.x <= max(p.x, r.x) && q.x >= min(p.x,
9         r.x) && q.y <= max(p.y, r.y) && q.y >= min
10         (p.y, r.y))
11         return true;
12     return false;
13 }
14 int orientation(Point p, Point q, Point r)
15 {
16     int val = (q.y - p.y) * (r.x - q.x) - (q.x
17         - p.x) * (r.y - q.y);
18     if (val == 0) return 0; // collinear
19     return (val > 0) ? 1 : 2; // clockwise or
20     counterclockwise
21 }
22 bool doIntersect(Point p1, Point q1, Point p2,
23                 Point q2)
24 {
25     int o1 = orientation(p1, q1, p2);
26     int o2 = orientation(p1, q1, q2);
27     int o3 = orientation(p2, q2, p1);
28     int o4 = orientation(p2, q2, q1);
29     if (o1 != o2 && o3 != o4) return true;
30     if (o1 == 0 && onSegment(p1, p2, q1))
31         return true;
32     if (o2 == 0 && onSegment(p1, q2, q1))
33         return true;
34     if (o3 == 0 && onSegment(p2, p1, q2))
35         return true;
36     if (o4 == 0 && onSegment(p2, q1, q2))
37         return true;
38     return false;
39 }

```

7 String

7.1 Hashing

```

1 namespace Hashing {
2     #define ff first

```

```

3  #define ss second
4  const PLL M = {1e9+7, 1e9+9};          ///
5  Should be large primes
6  const LL base = 1259;                  ///
7  Should be larger than alphabet size
8  const int N = 1e6+7;                   ///
9  Highest length of string
10 PLL operator+ (const PLL& a, LL x)    {
11   return {a.ff + x, a.ss + x};}
12 PLL operator- (const PLL& a, LL x)    {
13   return {a.ff - x, a.ss - x};}
14 PLL operator* (const PLL& a, LL x)    {
15   return {a.ff * x, a.ss * x};}
16 PLL operator+ (const PLL& a, PLL x)   {
17   return {a.ff + x.ff, a.ss + x.ss};}
18 PLL operator- (const PLL& a, PLL x)   {
19   return {a.ff - x.ff, a.ss - x.ss};}
20 PLL operator* (const PLL& a, PLL x)   {
21   return {a.ff * x.ff, a.ss * x.ss};}
22 PLL operator% (const PLL& a, PLL m)   {
23   return {a.ff % m.ff, a.ss % m.ss};}
24 ostream& operator<<(ostream& os, PLL hash)
25 {
26   return os<<"("<<hash.ff<<" ", "<<hash.ss
27   <<")";
28 }
29 PLL pb[N];          ///powers of base mod M
30 ///Call pre before everything
31 void hashPre() {
32   pb[0] = {1,1};
33   for (int i=1; i<N; i++)    pb[i] = (pb
34   [i-1] * base)%M;
35 }
36 ///Calculates hashes of all prefixes of s
37 including empty prefix
38 vector<PLL> hashList(string s) {
39   int n = s.size();
40   vector<PLL> ans(n+1);
41   ans[0] = {0,0};
42   for (int i=1; i<=n; i++)    ans[i] = (
43   ans[i-1] * base + s[i-1])%M;
44   return ans;
45 }
46 ///Calculates hash of substring s[l..r] (1
47 indexed)
48 PLL substringHash(const vector<PLL> &
49 hashlist, int l, int r) {
50   return (hashlist[r]+(M-hashlist[l-1])*
51   pb[r-l+1])%M;
52 }
53 ///Calculates Hash of a string

```

```

36 PLL Hash (string s) {
37   PLL ans = {0,0};
38   for (int i=0; i<s.size(); i++)    ans=(
39   ans*base + s[i])%M;
40   return ans;
41 }
42 ///appends c to string
43 PLL append(PLL cur, char c) {
44   return (cur*base + c)%M;
45 }
46 ///prepends c to string with size k
47 PLL prepend(PLL cur, int k, char c) {
48   return (pb[k]*c + cur)%M;
49 }
50 ///replaces the i-th (0-indexed) character
51 from right from a to b;
52 PLL replace(PLL cur, int i, char a, char b)
53 {
54   return cur + pb[i] * (M+b-a)%M;
55 }
56 ///Erases c from front of the string with
57 size len
58 PLL pop_front(PLL hash, int len, char c) {
59   return (hash + pb[len-1]*(M-c))%M;
60 }
61 ///concatenates two strings where length of
62 the right is k
63 PLL concat(PLL left, PLL right, int k) {
64   return (left*pb[k] + right)%M;
65 }
66 PLL power (const PLL& a, LL p) {
67   if (p==0)    return {1,1};
68   PLL ans = power(a, p/2);
69   ans = (ans * ans)%M;
70   if (p%2)    ans = (ans*a)%M;
71   return ans;
72 }
73 PLL inverse(PLL a) {
74   if (M.ss == 1)    return power(a, M.ff-2)
75   ;
76   return power(a, (M.ff-1)*(M.ss-1)-1);
77 }
78 ///Erases c from the back of the string
79 PLL invb = inverse({base, base});
80 PLL pop_back(PLL hash, char c) {
81   return ((hash-c+M)*invb)%M;
82 }
83 ///Calculates hash of string with size len
84 repeated cnt times
85 ///This is O(log n). For O(1), pre-
86 calculate inverses

```

```

79 PLL repeat(PLL hash, int len, LL cnt) {
80   PLL mul = ((pb[len*cnt]-1+M) * inverse(
81   pb[len]-1+M))%M;
82   PLL ans = (hash*mul);
83   if (pb[len].ff == 1)    ans.ff = hash.
84   ff*cnt;
85   if (pb[len].ss == 1)    ans.ss = hash.
86   ss*cnt;
87   return ans%M;
88 }

```

7.2 KMP

```

1 struct KMP {
2   string s;
3   int n;
4   vector<int> fail;
5   KMP(const string &ss) {
6     s = ss;
7     n = s.size();
8     fail.assign(n+1, 0);
9
10    fail[0] = fail[1] = 0;
11
12    for (int i=2; i<=n; i++) {
13      fail[i] = (s[i-1] == s[0]);
14      for (int j = fail[i-1]; j>0; j =
15      fail[j])
16        if (s[j] == s[i-1]) {
17          fail[i] = j+1;
18          break;
19        }
20    }
21    int match(string t) { ///No of matches
22      int cur = 0, ans = 0;
23      for (int i=0; i<t.size(); i++) {
24        if (t[i] == s[cur]) cur++, i++;
25        else if (cur==0) i++;
26        else cur = fail[cur];
27        if (cur==n) ans++, cur = fail[cur];
28      }
29      return ans;
30    }
31    vector<vector<int>> prefixAutomaton() {
32      vector<vector<int>> automaton(n+1,
33      vector<int> (26, 0));
34      automaton[0][s[0]-'a'] = 1;
35      for (int i=1, k=0; i<=n; i++) {

```

```

35     automaton[i] = automaton[k];
36     if (i < n) {
37         automaton[i][s[i]-'a'] = i+1;
38         k = automaton[k][s[i]-'a'];
39     }
40 }
41 return automaton;
42 }
43 };

```

7.3 Z Algo

```

1 vector<int> z_function(string s) {
2     int n = s.size();
3     vector<int> z(n);
4     int l = 0, r = 0;
5     for (int i=1; i<n; i++) {
6         if (i<=r) z[i] = min(r-i+1, z[i-l]);
7         while (i+z[i]<n && s[i+z[i]] == s[z[i]
8             ]) z[i]++;
9         if (i+z[i]-1>r) l = i, r = i+z[i]-1;
10    }
11    z[0] = s.size();
12    return z;
13 }

```

7.4 Aho Corasick

```

1 namespace Aho {
2     const int N = 1e6+7;          ///Number of
3     const int K = 26;             ///Alphabet
4     const int size;               ///size
5     int nxt[N][K];                ///Children
6     int go[N][K];                 ///automaton
7     int link[N];                  ///Suffix link
8     bool leaf[N];                 ///isLeaf
9     int par[N];                   ///Parent
10    char ch[N];                    ///character of
11    incoming edge
12    int ex[N];                      ///exit link
13    int sz;
14    void init() {
15        memset(nxt, -1, sizeof nxt);
16        memset(go, -1, sizeof go);
17        memset(link, -1, sizeof link);
18        memset(leaf, 0, sizeof leaf);
19    }
20 }

```

```

17     memset(ex, -1, sizeof ex);
18     sz = 0;
19 }
20 void addString(const string &s) {
21     int cur = 0;
22     for (char c: s) {
23         int cc = c-'a';
24         if (nxt[cur][cc] == -1) {
25             nxt[cur][cc] = ++sz;
26             ch[sz] = c;
27             par[sz] = cur;
28         }
29         cur = nxt[cur][cc];
30     }
31     leaf[cur] = 1;
32 }
33 int Go(int v, char ch);
34 ///Amortized O(1)
35 int getlink(int v) {
36     if (link[v] != -1) return link[v];
37     if (v==0 || par[v] == 0) return link
38     [v] = 0;
39     else return link[v] = Go(getlink(par[v]
40     ), ch[v]);
41 }
42 ///Amortized O(1)
43 int Go (int v, char c) {
44     int cc = c-'a';
45     if (go[v][cc] != -1) return go[v][cc];
46     if (nxt[v][cc] != -1) return go[v][cc]
47     = nxt[v][cc];
48     else return go[v][cc] = (v ? Go(getlink
49     (v), c) : 0);
50 }
51 ///Amortized O(1)
52 int exitlink(int v) {
53     if (ex[v] != -1) return ex[v];
54     if (nxt[v][cc] != -1) return ex[v] =
55     getlink(v);
56     if (nxt==0 || leaf[nxt]) return ex[v]
57     = nxt;
58     return ex[v] = exitlink(nxt);
59 }
60 ///returns number of matches (including
61 multiple matches)
62 ///O(no of matches + length of s)
63 int match(string s) {
64     int cur = 0;
65     int ans = 0;
66     for (auto c: s) {
67         cur = Go(cur, c);
68         int e = (leaf[cur] ? cur : exitlink
69         (cur));
70         while (e)
71             ans++,
72             e = exitlink(e);
73     }
74     return ans;
75 }
76 }
77 int main() {
78     Aho::init();
79     Aho::addString("banana");
80     Aho::addString("ban");
81     Aho::addString("nana");
82     Aho::addString("anachor");
83     Aho::addString("ana");
84     cout<<Aho::match("ban")<<endl; //
85     /1
86     cout<<Aho::match("banana")<<endl; //
87     /5
88     cout<<Aho::match("bananachor")<<endl; //
89     /6
90     cout<<Aho::match("ananana")<<endl; //
91     /5
92     cout<<Aho::match("ba")<<endl; //
93     /0
94     cout<<Aho::match("anachor")<<endl; //
95     /2
96 }

```

```

59     for (auto c: s) {
60         cur = Go(cur, c);
61         int e = (leaf[cur] ? cur : exitlink
62         (cur));
63         while (e)
64             ans++,
65             e = exitlink(e);
66     }
67     return ans;
68 }
69 }
70 int main() {
71     Aho::init();
72     Aho::addString("banana");
73     Aho::addString("ban");
74     Aho::addString("nana");
75     Aho::addString("anachor");
76     Aho::addString("ana");
77     cout<<Aho::match("ban")<<endl; //
78     /1
79     cout<<Aho::match("banana")<<endl; //
80     /5
81     cout<<Aho::match("bananachor")<<endl; //
82     /6
83     cout<<Aho::match("ananana")<<endl; //
84     /5
85     cout<<Aho::match("ba")<<endl; //
86     /0
87     cout<<Aho::match("anachor")<<endl; //
88     /2
89 }

```

7.5 Trie

8 MISC

8.1 Ordered Set

```

1 #include <ext/pb_ds/assoc_container.hpp>
2 #include <ext/pb_ds/tree_policy.hpp>
3 using namespace __gnu_pbds;
4
5 template<typename T>
6 using ordered_set = tree<T, null_type, less<T>,
7     rb_tree_tag,
8     tree_order_statistics_node_update>;

```