

## Contents

1	Graph Theory	
1.1	Adjacency List	
1.2	DFS	
1.3	BFS	
1.4	Disjoint Set and Kruskal	
1.5	Floyd-Warshall	
1.6	Dijkstra	
2	Number Theory	
2.1	Modulo	
2.2	Linear Sieve	
2.3	Prime Factorization	
2.4	Exponentiating by Squaring	

## 1 Graph Theory

### 1.1 Adjacency List

```

1 vector<int> list[5];
2
3 void Adjacency_List(){
4
5     // initial
6     for (int i = 0; i < 5; i++)
7         list[i].clear();
8
9     int a, b;    // start & end of an edge
10
11     while (cin >> a >> b)
12         list[a].push_back(b);
13         // list[b].push_back(a);
14 }

```

### 1.2 DFS

```

1 vector<int> G[N];
2 bitset<N> vis;
3 void dfs(int s) {
4     vis[s] = 1;
5     for (int t : G[s]) {
6         if (!vis[t])
7             dfs(t);
8     }
9 }

```

### 1.3 BFS

```

1 vector<int> G[N];
2 bitset<N> vis;
3 void bfs(int s) {
4     queue<int> q;
5     q.push(s);
6     vis[s] = 1;
7     while (!q.empty()) {
8         int v = q.front();
9         q.pop();
10        for (int t : G[v]) {
11            if (!vis[t]) {
12                q.push(t);
13                vis[t] = 1;
14            }
15        }
16    }
17 }

```

## 1.4 Disjoint Set and Kruskal

```

1 struct Edge{
2     int u, v, w;
3     // bool operator < (const Edge &rhs) const {
4         // return w < rhs.w; }
5 };
6 vector<int> parent;
7 vector<Edge> E;
8
9 bool cmp(Edge edge1, Edge edge2){
10     return edge2.w > edge1.w;
11 }
12
13 int find(int x){
14     if(parent[x] < 0){
15         return x;
16     }
17     return parent[x] = find(parent[x]);
18 }
19
20 bool Uni(int a, int b){
21     a = find(a);
22     b = find(b);
23     if(a == b){
24         return false;
25     }
26     if(parent[a] > parent[b]){
27         swap(a, b);
28     }
29     parent[a] = parent[a] + parent[b];
30     parent[b] = a;
31     return true;
32 }
33
34 void Kruskal() {
35
36     int cost = 0;
37
38     sort(E.begin(), E.end()); // sort by w
39     // sort(E.begin(), E.end(), cmp);
40
41     // two edge in the same tree or not
42     for (auto it: E){
43         it.s = Find(it.s);
44         it.t = Find(it.t);
45         if (Uni(it.s, it.t)){
46             cost = cost + it.w;
47         }
48     }
49 }
50
51 int main(){
52
53     // create N space and initial -1
54     parent = vector<int> (N, -1);
55
56     for(i = 0; i < M; i++){
57         cin >> u >> v >> w;
58         E.push_back({u, v, w});
59     }
60
61     Kruskal();
62
63     return 0;
64 }

```

## 1.5 Floyd-Warshall

```

1 for (k = 0; k < n; k++){
2     for (i = 0; i < n; i++){
3         for (j = 0; j < n; j++){
4             w[i][j] = min(w[i][j],
5                             max(w[i][k], w[k][j]));

```

```

5 |     }
6 | }
7 | }

```

## 1.6 Dijkstra

```

1 | struct edge {
2 |     int s, t;
3 |     LL d;
4 |     edge(){};
5 |     edge(int s, int t, LL d) : s(s), t(t), d(d) {}
6 | };
7 |
8 | struct heap {
9 |     LL d;
10 |    int p; // point
11 |    heap(){};
12 |    heap(LL d, int p) : d(d), p(p) {}
13 |    bool operator<(const heap &b) const { return d >
14 |        b.d; }
15 | };
16 | int d[N], p[N];
17 | vector<edge> edges;
18 | vector<int> G[N];
19 | bitset<N> vis;
20 |
21 | void Dijkstra(int ss){
22 |     priority_queue<heap> Q;
23 |     for (int i = 0; i < V; i++){
24 |         d[i] = INF;
25 |     }
26 |     d[ss] = 0;
27 |     p[ss] = -1;
28 |     vis.reset() : Q.push(heap(0, ss));
29 |     heap x;
30 |     while (!Q.empty()){
31 |         x = Q.top();
32 |         Q.pop();
33 |         int p = x.p;
34 |         if (vis[p])
35 |             continue;
36 |         vis[p] = 1;
37 |         for (int i = 0; i < G[p].size(); i++){
38 |             edge &e = edges[G[p][i]];
39 |             if (d[e.t] > d[p] + e.d){
40 |                 d[e.t] = d[p] + e.d;
41 |                 p[e.t] = G[p][i];
42 |                 Q.push(heap(d[e.t], e.t));
43 |             }
44 |         }
45 |     }
46 | }

```

## 2 Number Theory

### 2.1 Modulo

- $(a + b) \bmod p = (a \bmod p + b \bmod p) \bmod p$
- $(a - b) \bmod p = (a \bmod p - b \bmod p + p) \bmod p$
- $(a * b) \bmod p = (a \bmod p * b \bmod p) \bmod p$
- $(a^b) \bmod p = ((a \bmod p)^b) \bmod p$
- $((a + b) \bmod p + c) \bmod p = (a + (b + c)) \bmod p$
- $((a * b) \bmod p * c) \bmod p = (a * (b * c)) \bmod p$
- $(a + b) \bmod p = (b + a) \bmod p$
- $(a * b) \bmod p = (b * a) \bmod p$
- $((a + b) \bmod p * c) = ((a * c) \bmod p + (b * c) \bmod p) \bmod p$

- $a \equiv b \pmod{m} \Rightarrow c \cdot m = a - b, c \in \mathbb{Z}$   
 $\Rightarrow a \equiv b \pmod{m} \Rightarrow m \mid a - b$
- $a \equiv b \pmod{c}, b \equiv d \pmod{c}$   
 則  $a \equiv d \pmod{c}$
- $\begin{cases} a \equiv b \pmod{m} \\ c \equiv d \pmod{m} \end{cases} \Rightarrow \begin{cases} a \pm c \equiv b \pm d \pmod{m} \\ a \cdot c \equiv b \cdot d \pmod{m} \end{cases}$

### 2.2 Linear Sieve

```

1 | vector<int> p;
2 | bitset<MAXN> is_notp;
3 | void PrimeTable(int n) {
4 |     is_notp.reset();
5 |     is_notp[0] = is_notp[1] = 1;
6 |     for (int i = 2; i <= n; ++i) {
7 |         if (!is_notp[i]){
8 |             p.push_back(i);
9 |         }
10 |        for (int j = 0; j < (int)p.size(); ++j) {
11 |            if (i * p[j] > n){
12 |                break;
13 |            }
14 |            is_notp[i * p[j]] = 1;
15 |            if (i % p[j] == 0){
16 |                break;
17 |            }
18 |        }
19 |    }
20 | }

```

### 2.3 Prime Factorization

```

1 | void primeFactorization(int n){
2 |     for(int i = 0; i < (int)p.size(); i++){
3 |         if(p[i] * p[i] > n){
4 |             break;
5 |         }
6 |         if(n % p[i]){
7 |             continue;
8 |         }
9 |         cout << p[i] << ' ';
10 |        while(n % p[i] == 0){
11 |            n /= p[i];
12 |        }
13 |    }
14 |    if(n != 1){
15 |        cout << n << ' ';
16 |    }
17 |    cout << '\n';
18 | }

```

### 2.4 Exponentiating by Squaring

```

1 | T pow(int a, int b, int c){ // calculate a ^ b % c
2 |     T ans = 1, tmp = a;
3 |     for (; b >= 1) {
4 |         if (b & 1){ // b is odd
5 |             ans = ans * tmp % c;
6 |         }
7 |         tmp = tmp * tmp % c;
8 |     }
9 |     return ans;
10 | }

```