

## Contents

1	Basic
1.1	Run . . . . .
1.2	Default . . . . .
1.3	Black Magic . . . . .
1.4	Binary Search . . . . .
2	Data Structure
2.1	Disjoint Set . . . . .
2.2	BIT RARSQ . . . . .
2.3	zkw RMQ . . . . .
3	Graph
3.1	Dijkstra . . . . .
3.2	SPFA(negative cycle) . . . . .
3.3	Floyd Warshall . . . . .
3.4	Topological Sort . . . . .
3.5	Kosaraju SCC . . . . .
3.6	Tree Diameter . . . . .
3.7	Directed MST . . . . .
4	Flow & Matching
4.1	Bipartite Matching . . . . .
4.2	KM . . . . .
4.3	Dinic . . . . .
4.4	MCMF . . . . .
5	String
5.1	Manacher . . . . .
6	DP
6.1	LIS . . . . .
6.2	LCS . . . . .
7	Math
7.1	Extended GCD . . . . .
7.2	Gaussian Elimination . . . . .
7.3	Prime Table . . . . .
8	Geometry
8.1	Point . . . . .
8.2	Line . . . . .
8.3	Area . . . . .
8.4	Convex Hull . . . . .

## 1 Basic

### 1.1 Run

```

1 | #use -> sh run.sh {name}
2 | g++ -O2 -std=c++14 -Wall -Wextra -Wshadow -o $1 $1.cpp
3 | ./ $1 < t.in > t.out

```

### 1.2 Default

```

1 | #include <bits/stdc++.h>
2 | using namespace std;
3 | using LL = long long;
4 | #define IOS ios_base::sync_with_stdio(0); cin.tie(0);
5 | #define pb push_back
6 | #define eb emplace_back
7 | const int INF = 1e9;
8 | const int MOD = 1e9 + 7;
9 | const double EPS = 1e-6;
10 | const int MAXN = 0;
11 |
12 | int main() {
13 |
14 | }

```

## 1.3 Black Magic

```

1 | #include <bits/stdc++.h>
2 | #include <ext/pb_ds/assoc_container.hpp>
3 | #include <ext/pb_ds/tree_policy.hpp>
4 | #include <ext/pb_ds/priority_queue.hpp>
5 | using namespace std;
6 | using namespace __gnu_pbds;
7 | using set_t =
8 |     tree<int, null_type, less<int>, rb_tree_tag,
9 |         tree_order_statistics_node_update>;
10 | using map_t =
11 |     tree<int, int, less<int>, rb_tree_tag,
12 |         tree_order_statistics_node_update>;
13 | using heap_t =
14 |     __gnu_pbds::priority_queue<int>;
15 | using ht_t =
16 |     gp_hash_table<int, int>;
17 | int main() {
18 |     //set-----
19 |     set_t st;
20 |     st.insert(5); st.insert(6);
21 |     st.insert(3); st.insert(1);
22 |
23 |     // the smallest is (0), biggest is (n-1), kth small
24 |     // is (k-1)
25 |     int num = *st.find_by_order(0);
26 |     cout << num << '\n'; // print 1
27 |
28 |     num = *st.find_by_order(st.size() - 1);
29 |     cout << num << '\n'; // print 6
30 |
31 |     // find the index
32 |     int index = st.order_of_key(6);
33 |     cout << index << '\n'; // print 3
34 |
35 |     // check if there exists x
36 |     int x = 5;
37 |     int check = st.erase(x);
38 |     if (check == 0) printf("st not contain 5\n");
39 |     else if (check == 1) printf("st contain 5\n");
40 |
41 |     //tree policy like set
42 |     st.insert(5); st.insert(5);
43 |     cout << st.size() << '\n'; // print 4
44 |
45 |     //map-----
46 |     map_t mp;
47 |     mp[1] = 2;
48 |     cout << mp[1] << '\n';
49 |     auto tmp = *mp.find_by_order(0); // pair
50 |     cout << tmp.first << " " << tmp.second << '\n';
51 |
52 |     //heap-----
53 |     heap_t h1, h2;
54 |     h1.push(1); h1.push(3);
55 |     h2.push(2); h2.push(4);
56 |     h1.join(h2);
57 |     cout << h1.size() << h2.size() << h1.top() << '\n';
58 |     // 404
59 |
60 |     //hash-table-----
61 |     ht_t ht;
62 |     ht[85] = 5;
63 |     ht[89975] = 234;
64 |     for (auto i : ht) {
65 |         cout << i.first << " " << i.second << '\n';
66 |     }

```

## 1.4 Binary Search

```

1 | lower_bound(a, a + n, k); //最左邊 ≥ k 的位置
2 | upper_bound(a, a + n, k); //最左邊 > k 的位置
3 | upper_bound(a, a + n, k) - 1; //最右邊 ≤ k 的位置

```

```

4 lower_bound(a, a + n, k) - 1; //最右邊 < k 的位置
5 [lower_bound, upper_bound) //等於 k 的範圍
6 equal_range(a, a + n, k);

```

## 2 Data Structure

### 2.1 Disjoint Set

```

1 // 0-base
2 const int MAXN = 1000;
3 int boss[MAXN];
4 void init(int n) {
5     for (int i = 0; i < n; i++) {
6         boss[i] = -1;
7     }
8 }
9 int find(int x) {
10     if (boss[x] < 0) {
11         return x;
12     }
13     return boss[x] = find(boss[x]);
14 }
15 bool uni(int a, int b) {
16     a = find(a);
17     b = find(b);
18     if (a == b) {
19         return false;
20     }
21     if (boss[a] > boss[b]) {
22         swap(a, b);
23     }
24     boss[a] += boss[b];
25     boss[b] = a;
26     return true;
27 }

```

### 2.2 BIT RARSQ

```

1 // 1-base
2 #define lowbit(k) (k & -k)
3
4 int n;
5 vector<int> B1, B2;
6
7 void add(vector<int> &tr, int id, int val) {
8     for (; id <= n; id += lowbit(id)) {
9         tr[id] += val;
10    }
11 }
12 void range_add(int l, int r, int val) {
13     add(B1, l, val);
14     add(B1, r + 1, -val);
15     add(B2, l, val * (1 - 1));
16     add(B2, r + 1, -val * r);
17 }
18 int sum(vector<int> &tr, int id) {
19     int ret = 0;
20     for (; id >= 1; id -= lowbit(id)) {
21         ret += tr[id];
22     }
23     return ret;
24 }
25 int prefix_sum(int id) {
26     return sum(B1, id) * id - sum(B2, id);
27 }
28 int range_sum(int l, int r) {
29     return prefix_sum(r) - prefix_sum(l - 1);
30 }

```

### 2.3 zkw RMQ

```

1 // 0-base
2 const int INF = 1e9;
3 const int MAXN = ;
4
5 int n;
6 int a[MAXN], tr[MAXN << 1];
7
8 // !!! remember to call this function
9 void build() {
10     for (int i = 0; i < n; i++) {
11         tr[i + n] = a[i];
12     }
13     for (int i = n - 1; i > 0; i--) {
14         tr[i] = max(tr[i << 1], tr[i << 1 | 1]);
15     }
16 }
17 void update(int id, int val) {
18     for (tr[id += n] = val; id > 1; id >>= 1) {
19         tr[id >> 1] = max(tr[id], tr[id ^ 1]);
20     }
21 }
22 int query(int l, int r) { // [l, r)
23     int ret = -INF;
24     for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
25         if (l & 1) {
26             ret = max(ret, tr[l++]);
27         }
28         if (r & 1) {
29             ret = max(ret, tr[--r]);
30         }
31     }
32     return ret;
33 }

```

## 3 Graph

### 3.1 Dijkstra

```

1 // 0-base
2 const LL INF = 1e18;
3 const int MAXN = ;
4 struct Edge {
5     int to;
6     LL cost;
7     Edge(int v, LL c) : to(v), cost(c) {}
8     bool operator < (const Edge &other) const {
9         return cost > other.cost;
10    }
11 };
12
13 int n;
14 LL dis[MAXN];
15 vector<Edge> G[MAXN];
16
17 void init() {
18     for (int i = 0; i < n; i++) {
19         G[i].clear();
20         dis[i] = INF;
21     }
22 }
23 void Dijkstra(int st, int ed = -1) {
24     priority_queue<Edge> pq;
25     pq.emplace(st, 0);
26     dis[st] = 0;
27     while (!pq.empty()) {
28         auto now = pq.top();
29         pq.pop();
30         if (now.to == ed) {
31             return;
32         }
33         if (now.cost > dis[now.to]) {
34             continue;
35         }
36     }
37 }

```

```

35     }
36     for (auto &e : G[now.to]) {
37         if (dis[e.to] > now.cost + e.cost) {
38             dis[e.to] = now.cost + e.cost;
39             pq.emplace(e.to, dis[e.to]);
40         }
41     }
42 }
43 }

```

### 3.2 SPFA(negative cycle)

```

1 // 0-base
2 const LL INF = 1e18;
3 const int MAXN = ;
4 struct Edge {
5     int to;
6     LL cost;
7     Edge(int v, LL c) : to(v), cost(c) {}
8 };
9
10 int n;
11 LL dis[MAXN];
12 vector<Edge> G[MAXN];
13
14 void init() {
15     for (int i = 0; i < n; i++) {
16         G[i].clear();
17         dis[i] = INF;
18     }
19 }
20 bool SPFA(int st) {
21     vector<int> cnt(n, 0);
22     vector<bool> inq(n, false);
23     queue<int> q;
24
25     q.push(st);
26     dis[st] = 0;
27     inq[st] = true;
28     while (!q.empty()) {
29         int now = q.front();
30         q.pop();
31         inq[now] = false;
32         for (auto &e : G[now]) {
33             if (dis[e.to] > dis[now] + e.cost) {
34                 dis[e.to] = dis[now] + e.cost;
35                 if (!inq[e.to]) {
36                     cnt[e.to]++;
37                     if (cnt[e.to] > n) {
38                         // negative cycle
39                         return false;
40                     }
41                     inq[e.to] = true;
42                     q.push(e.to);
43                 }
44             }
45         }
46     }
47     return true;
48 }

```

### 3.3 Floyd Warshall

```

1 // 0-base
2 // G[i][i] < 0 -> negative cycle
3 const LL INF = 1e18;
4 const int MAXN = ;
5
6 int n;
7 LL G[MAXN][MAXN];
8
9 void init() {
10     for (int i = 0; i < n; i++) {

```

```

11         for (int j = 0; j < n; j++) {
12             G[i][j] = INF;
13         }
14         G[i][i] = 0;
15     }
16 }
17 void floyd() {
18     for (int k = 0; k < n; k++) {
19         for (int i = 0; i < n; i++) {
20             for (int j = 0; j < n; j++) {
21                 if (G[i][k] != INF && G[k][j] != INF) {
22                     G[i][j] = min(G[i][j], G[i][k] + G[k][j]);
23                 }
24             }
25         }
26     }
27 }

```

### 3.4 Topological Sort

```

1 // 0-base
2 // if ret.size < n -> cycle
3 int n;
4 vector<vector<int>> G;
5
6 vector<int> topoSort() {
7     vector<int> indeg(n), ret;
8     for (auto &li : G) {
9         for (int x : li) {
10             ++indeg[x];
11         }
12     }
13     // use priority queue for lexic. largest ans
14     queue<int> q;
15     for (int i = 0; i < n; i++) {
16         if (!indeg[i]) {
17             q.push(i);
18         }
19     }
20     while (!q.empty()) {
21         int u = q.front();
22         q.pop();
23         ret.pb(u);
24         for (int v : G[u]) {
25             if (--indeg[v] == 0) {
26                 q.push(v);
27             }
28         }
29     }
30     return ret;
31 }

```

### 3.5 Kosaraju SCC

```

1 // 0-base
2 int n;
3 vector<vector<int>> G, G2; // G2 = G rev
4 vector<bool> vis;
5 vector<int> s, color;
6 int sccCnt;
7 void dfs1(int u) {
8     vis[u] = true;
9     for (int v : G[u]) {
10         if (!vis[v]) {
11             dfs1(v);
12         }
13     }
14     s.pb(u);
15 }
16 void dfs2(int u) {
17     color[u] = sccCnt;
18     for (int v : G2[u]) {
19         if (!color[v]) {

```

```

20     dfs2(v);
21 }
22 }
23 }
24 void Kosaraju() {
25     sccCnt = 0;
26     for (int i = 0; i < n; i++) {
27         if (!vis[i]) {
28             dfs1(i);
29         }
30     }
31     for (int i = n - 1; i >= 0; i--) {
32         if (!color[s[i]]) {
33             ++sccCnt;
34             dfs2(s[i]);
35         }
36     }
37 }

```

### 3.6 Tree Diameter

```

1 // 0-base;
2 const int MAXN = ;
3
4 struct Edge {
5     int to;
6     int cost;
7     Edge(int v, int c) : to(v), cost(c) {}
8 };
9
10 int n, d = 0;
11 int d1[MAXN], d2[MAXN];
12 vector<Edge> G[MAXN];
13 // dfs(0, -1);
14 void dfs(int u, int from) {
15     d1[u] = d2[u] = 0;
16     for (auto e : G[u]) {
17         if (e.to == from) {
18             continue;
19         }
20         dfs(e.to, u);
21         int t = d1[e.to] + e.cost;
22         if (t > d1[u]) {
23             d2[u] = d1[u];
24             d1[u] = t;
25         } else if (t > d2[u]) {
26             d2[u] = t;
27         }
28     }
29     d = max(d, d1[u] + d2[u]);
30 }

```

### 3.7 Directed MST

```

1 // 0-base
2 const LL INF = 1e18;
3 const int MAXN = ;
4
5 struct Edge {
6     int from;
7     int to;
8     LL cost;
9     Edge(int u, int v, LL c) : from(u), to(v), cost(c) {}
10 };
11
12 struct DMST {
13     int n;
14     int vis[MAXN], pre[MAXN], id[MAXN];
15     LL in[MAXN];
16     vector<Edge> edges;
17     void init(int _n) {
18         n = _n;

```

```

19     edges.clear();
20 }
21 void add_edge(int from, int to, LL cost) {
22     edges.eb(from, to, cost);
23 }
24 LL run(int root) {
25     LL ret = 0;
26     while (true) {
27         for (int i = 0; i < n; i++) {
28             in[i] = INF;
29         }
30
31         // find in edge
32         for (auto &e : edges) {
33             if (e.cost < in[e.to] && e.from != e.to) {
34                 pre[e.to] = e.from;
35                 in[e.to] = e.cost;
36             }
37         }
38
39         // check in edge
40         for (int i = 0; i < n; i++) {
41             if (i == root) {
42                 continue;
43             }
44             if (in[i] == INF) {
45                 return -1;
46             }
47         }
48
49         int nodelist = 0;
50         memset(id, -1, sizeof(id));
51         memset(vis, -1, sizeof(vis));
52         in[root] = 0;
53
54         // find cycles
55         for (int i = 0; i < n; i++) {
56             ret += in[i];
57             int v = i;
58             while (vis[v] != i && id[v] == -1 && v != root) {
59                 vis[v] = i;
60                 v = pre[v];
61             }
62             if (id[v] == -1 && v != root) {
63                 for (int j = pre[v]; j != v; j = pre[j]) {
64                     id[j] = nodelist;
65                 }
66                 id[v] = nodelist++;
67             }
68         }
69
70         // no cycle
71         if (nodelist == 0) {
72             break;
73         }
74
75         for (int i = 0; i < n; i++) {
76             if (id[i] == -1) {
77                 id[i] = nodelist++;
78             }
79         }
80
81         // grouping the vertices
82         for (auto &e : edges) {
83             int to = e.to;
84             e.from = id[e.from];
85             e.to = id[e.to];
86             if (e.from != e.to) {
87                 e.cost -= in[to]; //!!!
88             }
89         }
90
91         n = nodelist;
92         root = id[root];
93     }
94     return ret;

```

```
95 }
96 };
```

## 4 Flow & Matching

### 4.1 Bipartite Matching

```
1 // 0-base
2 const int MAXN = ;
3 int n;
4 vector<int> G[MAXN];
5 int vy[MAXN], my[MAXN];
6
7 bool match(int u) {
8     for (int v : G[u]) {
9         if (vy[v]) {
10             continue;
11         }
12         vy[v] = true;
13         if (my[v] == -1 || match(my[v])) {
14             my[v] = u;
15             return true;
16         }
17     }
18     return false;
19 }
20 int sol() {
21     int cnt = 0;
22     memset(my, -1, sizeof(my));
23     for (int i = 0; i < n; i++) {
24         memset(vy, 0, sizeof(vy));
25         if (match(i)) {
26             cnt++;
27         }
28     }
29     return cnt;
30 }
```

### 4.2 KM

```
1 const int INF = 1e9;
2 const int MAXN = ;
3 struct KM { //1-base
4     int n, G[MAXN][MAXN];
5     int lx[MAXN], ly[MAXN], my[MAXN];
6     bool vx[MAXN], vy[MAXN];
7     void init(int _n) {
8         n = _n;
9         for (int i = 1; i <= n; i++) {
10             for (int j = 1; j <= n; j++) {
11                 G[i][j] = 0;
12             }
13         }
14     }
15     bool match(int i) {
16         vx[i] = true;
17         for (int j = 1; j <= n; j++) {
18             if (lx[i] + ly[j] == G[i][j] && !vy[j]) {
19                 vy[j] = true;
20                 if (!my[j] || match(my[j])) {
21                     my[j] = i;
22                     return true;
23                 }
24             }
25         }
26         return false;
27     }
28     void update() {
29         int delta = INF;
30         for (int i = 1; i <= n; i++) {
31             if (vx[i]) {
32                 for (int j = 1; j <= n; j++) {
```

```
33         if (!vy[j]) {
34             delta = min(delta, lx[i] + ly[j] -
35                         G[i][j]);
36         }
37     }
38 }
39 for (int i = 1; i <= n; i++) {
40     if (vx[i]) {
41         lx[i] -= delta;
42     }
43     if (vy[i]) {
44         ly[i] += delta;
45     }
46 }
47 }
48 int run() {
49     for (int i = 1; i <= n; i++) {
50         lx[i] = ly[i] = my[i] = 0;
51         for (int j = 1; j <= n; j++) {
52             lx[i] = max(lx[i], G[i][j]);
53         }
54     }
55     for (int i = 1; i <= n; i++) {
56         while (true) {
57             for (int i = 1; i <= n; i++) {
58                 vx[i] = vy[i] = 0;
59             }
60             if (match(i)) {
61                 break;
62             } else {
63                 update();
64             }
65         }
66     }
67     int ans = 0;
68     for (int i = 1; i <= n; i++) {
69         ans += lx[i] + ly[i];
70     }
71     return ans;
72 }
73 };
```

### 4.3 Dinic

```
1 #define eb emplace_back
2 const LL INF = 1e18;
3 const int MAXN = ;
4 struct Edge {
5     int to;
6     LL cap;
7     int rev;
8     Edge(int v, LL c, int r) : to(v), cap(c), rev(r) {}
9 };
10 struct Dinic {
11     int n;
12     int level[MAXN], now[MAXN];
13     vector<Edge> G[MAXN];
14     void init(int _n) {
15         n = _n;
16         for (int i = 0; i <= n; i++) {
17             G[i].clear();
18         }
19     }
20     void add_edge(int u, int v, LL c) {
21         G[u].eb(v, c, G[v].size());
22         // directed graph
23         G[v].eb(u, 0, G[u].size() - 1);
24         // undirected graph
25         // G[v].eb(u, c, G[u].size() - 1);
26     }
27     bool bfs(int st, int ed) {
28         fill(level, level + n + 1, -1);
29         queue<int> q;
30         q.push(st);
31         level[st] = 0;
```

```

32 while (!q.empty()) {
33     int u = q.front();
34     q.pop();
35     for (const auto &e : G[u]) {
36         if (e.cap > 0 && level[e.to] == -1) {
37             level[e.to] = level[u] + 1;
38             q.push(e.to);
39         }
40     }
41 }
42 return level[ed] != -1;
43 }
44 LL dfs(int u, int ed, LL limit) {
45     if (u == ed) {
46         return limit;
47     }
48     LL ret = 0;
49     for (int &i = now[u]; i < G[u].size(); i++) {
50         auto &e = G[u][i];
51         if (e.cap > 0 && level[e.to] == level[u] + 1) {
52             LL f = dfs(e.to, ed, min(limit, e.cap));
53             ret += f;
54             limit -= f;
55             e.cap -= f;
56             G[e.to][e.rev].cap += f;
57             if (!limit) {
58                 return ret;
59             }
60         }
61     }
62     if (!ret) {
63         level[u] = -1;
64     }
65     return ret;
66 }
67 LL flow(int st, int ed) {
68     LL ret = 0;
69     while (bfs(st, ed)) {
70         fill(now, now + n + 1, 0);
71         ret += dfs(st, ed, INF);
72     }
73     return ret;
74 }
75 };

```

#### 4.4 MCMF

```

1 // 0-base
2 const LL INF = 1e18;
3 const int MAXN = ;
4 struct Edge {
5     int u, v;
6     LL cost;
7     LL cap;
8     Edge(int _u, int _v, LL _c, LL _cap) : u(_u),
9         v(_v), cost(_c), cap(_cap) {}
10 };
11 struct MCMF { // inq times
12     int n, pre[MAXN], cnt[MAXN];
13     LL ans_flow, ans_cost, dis[MAXN];
14     bool inq[MAXN];
15     vector<int> G[MAXN];
16     vector<Edge> edges;
17     void init(int _n) {
18         n = _n;
19         edges.clear();
20         for (int i = 0; i < n; i++) {
21             G[i].clear();
22         }
23     }
24     void add_edge(int u, int v, LL c, LL cap) {
25         // directed
26         G[u].pb(edges.size());
27         edges.pb(u, v, c, cap);
28         G[v].pb(edges.size());
29         edges.pb(v, u, -c, 0);
30     }

```

```

29 }
30 bool SPFA(int st, int ed) {
31     for (int i = 0; i < n; i++) {
32         pre[i] = -1;
33         dis[i] = INF;
34         cnt[i] = 0;
35         inq[i] = false;
36     }
37     queue<int> q;
38     bool negcycle = false;
39
40     dis[st] = 0;
41     cnt[st] = 1;
42     inq[st] = true;
43     q.push(st);
44
45     while (!q.empty() && !negcycle) {
46         int u = q.front();
47         q.pop();
48         inq[u] = false;
49         for (int i : G[u]) {
50             int v = edges[i].v;
51             LL cost = edges[i].cost;
52             LL cap = edges[i].cap;
53
54             if (dis[v] > dis[u] + cost && cap > 0) {
55                 dis[v] = dis[u] + cost;
56                 pre[v] = i;
57                 if (!inq[v]) {
58                     q.push(v);
59                     cnt[v]++;
60                     inq[v] = true;
61
62                     if (cnt[v] == n + 2) {
63                         negcycle = true;
64                         break;
65                     }
66                 }
67             }
68         }
69     }
70
71     return dis[ed] != INF;
72 }
73 LL sendFlow(int v, LL curFlow) {
74     if (pre[v] == -1) {
75         return curFlow;
76     }
77     int i = pre[v];
78     int u = edges[i].u;
79     LL cost = edges[i].cost;
80
81     LL f = sendFlow(u, min(curFlow, edges[i].cap));
82
83     ans_cost += f * cost;
84     edges[i].cap -= f;
85     edges[i ^ 1].cap += f;
86     return f;
87 }
88 pair<LL, LL> run(int st, int ed) {
89     ans_flow = ans_cost = 0;
90     while (SPFA(st, ed)) {
91         ans_flow += sendFlow(ed, INF);
92     }
93     return make_pair(ans_flow, ans_cost);
94 }
95 };

```

## 5 String

### 5.1 Manacher

```

1 int p[2 * MAXN];
2 int Manacher(const string &s) {

```

```

3 | string st = "@#";
4 | for (char c : s) {
5 |     st += c;
6 |     st += '#';
7 | }
8 | st += '$';
9 | int id = 0, mx = 0, ans = 0;
10 | for (int i = 1; i < st.length() - 1; i++) {
11 |     p[i] = (mx > i ? min(p[2 * id - i], mx - i) : 1);
12 |     for (; st[i - p[i]] == st[i + p[i]]; p[i]++);
13 |     if (mx < i + p[i]) {
14 |         mx = i + p[i];
15 |         id = i;
16 |     }
17 |     ans = max(ans, p[i] - 1);
18 | }
19 | return ans;
20 | }

```

## 6 DP

### 6.1 LIS

```

1 | int LIS(vector<int> &a) {
2 |     vector<int> s;
3 |     for (int i = 0; i < a.size(); i++) {
4 |         if (s.empty() || s.back() < a[i]) {
5 |             s.push_back(a[i]);
6 |         } else {
7 |             *lower_bound(s.begin(), s.end(), a[i],
8 |                 [](int x, int y) {return x < y;}) = a[i];
9 |         }
10 |    }
11 |    return s.size();
12 | }

```

### 6.2 LCS

```

1 | int LCS(string s1, string s2) {
2 |     int n1 = s1.size(), n2 = s2.size();
3 |     vector<vector<int>> dp(n1 + 1, vector<int>(n2 + 1,
4 |         0));
5 |     for (int i = 1; i <= n1; i++) {
6 |         for (int j = 1; j <= n2; j++) {
7 |             if (s1[i - 1] == s2[j - 1]) {
8 |                 dp[i][j] = dp[i - 1][j - 1] + 1;
9 |             } else {
10 |                 dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
11 |             }
12 |         }
13 |     }
14 |     return dp[n1][n2];
15 | }

```

## 7 Math

### 7.1 Extended GCD

```

1 | // ax + by = c
2 | int extgcd(int a, int b, int c, int &x, int &y) {
3 |     if (b == 0) {
4 |         x = c / a;
5 |         y = 0;
6 |         return a;
7 |     }
8 |     int d = extgcd(b, a % b, c, y, x);
9 |     y -= (a / b) * x;
10 |    return d;
11 | }

```

## 7.2 Gaussian Elimination

```

1 | const double EPS = 1e-6;
2 | double Gauss(vector<vector<double>> &d) {
3 |     int n = d.size(), m = d[0].size();
4 |     double det = 1;
5 |     for (int i = 0; i < m; i++) {
6 |         int p = -1;
7 |         for (int j = i; j < n; j++) {
8 |             if (fabs(d[j][i]) < EPS) {
9 |                 continue;
10 |            }
11 |            if (p == -1 || fabs(d[j][i]) > fabs(d[p][i])) {
12 |                p = j;
13 |            }
14 |        }
15 |        if (p == -1) {
16 |            continue;
17 |        }
18 |        if (p != i) {
19 |            det *= -1;
20 |        }
21 |        for (int j = 0; j < m; j++) {
22 |            swap(d[p][j], d[i][j]);
23 |        }
24 |        for (int j = 0; j < n; j++) {
25 |            if (i == j) {
26 |                continue;
27 |            }
28 |            double z = d[j][i] / d[i][i];
29 |            for (int k = 0; k < m; k++) {
30 |                d[j][k] -= z * d[i][k];
31 |            }
32 |        }
33 |    }
34 |    for (int i = 0; i < n; i++) {
35 |        det *= d[i][i];
36 |    }
37 |    return det;
38 | }

```

## 7.3 Prime Table

```

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 |         for (int j = 0; j < (int)p.size(); ++j) {
10 |             if (i * p[j] > n)
11 |                 break;
12 |             is_notp[i * p[j]] = 1;
13 |             if (i % p[j] == 0)
14 |                 break;
15 |         }
16 |     }
17 | }

```

## 8 Geometry

### 8.1 Point

```

1 | // notice point type!!!
2 | using dvt = int;
3 | const double EPS = 1e-6;
4 | const double PI = acos(-1);
5 |
6 | struct Pt {
7 |     dvt x;

```

```

8   dvt y;
9   };
10  bool operator < (const Pt &a, const Pt &b) {
11      return a.x == b.x ? a.y < b.y : a.x < b.x;
12  }
13  bool operator == (const Pt &a, const Pt &b) {
14      return a.x == b.x && a.y == b.y;
15  }
16  Pt operator + (const Pt &a, const Pt &b) {
17      return {a.x + b.x, a.y + b.y};
18  }
19  Pt operator - (const Pt &a, const Pt &b) {
20      return {a.x - b.x, a.y - b.y};
21  }
22  // multiply constant
23  Pt operator * (const Pt &a, const dvt c) {
24      return {a.x * c, a.y * c};
25  }
26  Pt operator / (const Pt &a, const dvt c) {
27      return {a.x / c, a.y / c};
28  }
29  // |a| x |b| x cos(x)
30  dvt iproduct(const Pt &a, const Pt &b) {
31      return a.x * b.x + a.y * b.y;
32  }
33  // |a| x |b| x sin(x)
34  dvt cross(const Pt &a, const Pt &b) {
35      return a.x * b.y - a.y * b.x;
36  }
37  dvt dis_pp(const Pt &a, const Pt &b) {
38      dvt dx = a.x - b.x;
39      dvt dy = a.y - b.y;
40      return sqrt(dx * dx + dy * dy);
41  }

```

## 8.2 Line

```

1  struct Line {
2      Pt st;
3      Pt ed;
4  };
5  // return point side
6  // left, on line, right -> 1, 0, -1
7  int side(Line l, Pt a) {
8      dvt cross_val = cross(a - l.st, l.ed - l.st);
9      if (cross_val > EPS) {
10         return 1;
11     } else if (cross_val < -EPS) {
12         return -1;
13     } else {
14         return 0;
15     }
16 }
17 // AB infinity, CD segment
18 bool has_intersection(Line AB, Line CD) {
19     int c = side(AB, CD.st);
20     int d = side(AB, CD.ed);
21     if (c == 0 || d == 0) {
22         return true;
23     } else {
24         // different side
25         return c == -d;
26     }
27 }
28 // find intersection point, two line, not seg
29 pair<int, Pt> intersection(Line a, Line b) {
30     Pt A = a.ed - a.st;
31     Pt B = b.ed - b.st;
32     Pt C = b.st - a.st;
33     dvt mom = cross(A, B);
34     dvt son = cross(C, B);
35     if (std::abs(mom) <= EPS) {
36         if (std::abs(son) <= EPS) {
37             return {1, {}}; // same line
38         } else {
39             return {2, {}}; // parallel

```

```

40     }
41 } else {
42     // ok
43     return {0, a.st + A * (son / mom)};
44 }
45 // line to point distance
46 dvt dis_lp(Line l, Pt a) {
47     return area3x2(l.st, l.ed, a) / dis_pp(l.st, l.ed);
48 }

```

$$d(P, L) = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}$$

## 8.3 Area

```

1  // triangle
2  dvt area3(Pt a, Pt b, Pt c) {
3      return std::abs(cross(b - a, c - a) / 2);
4  }
5  dvt area3x2(Pt a, Pt b, Pt c) { // for integer
6      return std::abs(cross(b - a, c - a));
7  }
8  // simple convex area(can in)
9  dvt area(vector<Pt> &a) {
10     dvt ret = 0;
11     for (int i = 0, sz = a.size(); i < sz; i++) {
12         ret += cross(a[i], a[(i + 1) % sz]);
13     }
14     return std::abs(ret) / 2;
15 }
16 // check point in/out a convex
17 int io_convex(vector<Pt> convex, Pt q) {
18     // convex is Counterclockwise
19     for (int i = 0, sz = convex.size(); i < sz; i++) {
20         Pt cur = convex[i] - q;
21         Pt nex = convex[(i + 1) % sz] - q;
22         dvt cross_val = cross(cur, nex);
23         if (std::abs(cross_val) <= EPS) {
24             return 0; // on edge
25         }
26         if (cross_val < 0) {
27             return -1; // outside
28         }
29     }
30     return 1; // inside
31 }

```

## 8.4 Convex Hull

```

1  vector<Pt> convex_hull(vector<Pt> &a) {
2      sort(a.begin(), a.end());
3      a.erase(unique(a.begin(), a.end()), a.end());
4      int sz = a.size(), m = 0;
5      vector<Pt> ret(sz + 5); // safe 1 up
6      for (int i = 0; i < sz; i++) {
7          while (m > 1 &&
8              cross(ret[m - 1] - ret[m - 2], a[i] - ret[m - 2]) <= EPS) {
9              m--;
10         }
11         ret[m++] = a[i];
12     }
13     int k = m;
14     for (int i = sz - 2; i >= 0; i--) {
15         while (m > k &&
16             cross(ret[m - 1] - ret[m - 2], a[i] - ret[m - 2]) <= EPS) {
17             m--;
18         }
19         ret[m++] = a[i];
20     }
21     if (sz > 1) {
22         m--;
23     }

```



---

```
24 |     ret.resize(m);  
25 |     return ret;  
26 | }
```