

# CP Algorithms Handbook

for the recently deceased

Stanisław Fiedler

November 13, 2025

## Contents

<b>1 Sorting and Searching</b>	<b>2</b>
1.1 Quick Sort . . . . .	2
1.2 Merge Sort . . . . .	2
<b>2 Dynamic Programming</b>	<b>2</b>
2.1 State representation . . . . .	2
2.1.1 All subsets . . . . .	2
<b>3 Graph Algorithms</b>	<b>3</b>
3.1 Shortest Path . . . . .	3
3.1.1 Djikstra . . . . .	3
3.1.2 Belman-Ford . . . . .	3
3.1.3 Floyd-Warshal . . . . .	3
3.2 Union Find . . . . .	4
3.3 Path through all edges . . . . .	4
3.4 Topo Sort . . . . .	5
3.5 Flows and Cuts . . . . .	6
3.5.1 Maximum Flow . . . . .	6
3.5.2 Minimum Cut . . . . .	7
3.5.3 Maximum Matching . . . . .	7
3.6 2SAT . . . . .	7
<b>4 Range Queries</b>	<b>9</b>
4.1 Segment trees . . . . .	9
4.1.1 Point-Range Trees . . . . .	9
4.1.2 Range-Range Trees . . . . .	10
4.2 Max Subarray Sum . . . . .	11
<b>5 Tree Algorithms</b>	<b>11</b>
5.1 Binary Liftng . . . . .	11
5.2 Tree Traversal . . . . .	12
5.3 LCA . . . . .	12
<b>6 String Algorithms</b>	<b>12</b>
6.1 Trie . . . . .	12
6.2 Pattern Finding . . . . .	12
6.2.1 KMP . . . . .	12
6.2.2 Hashing . . . . .	13
6.3 Palindromes . . . . .	14
<b>7 Mathematics</b>	<b>14</b>
7.1 Fermat's Theorem . . . . .	14
7.2 Fast Fibonacci . . . . .	14
<b>8 Geometry</b>	<b>15</b>
8.1 Point location . . . . .	15
8.2 Point in polygon . . . . .	15
8.3 Convex hull . . . . .	16

# 1 Sorting and Searching

## 1.1 Quick Sort

```
1 void qwik_sort(int p, int q, int tab[]){
2     int a = p, b = q;
3     int m = tab[(p+q)/2];
4
5     while(a <= b){
6         while(m > tab[a])
7             a++;
8         while(m < tab[b])
9             b--;
10        if(a <= b){
11            swap(tab[a],tab[b]);
12            a++;
13            b--;
14        }
15    }
16    if(b > p)
17        qwik_sort(p, b, tab);
18    if(a < q)
19        qwik_sort(a, q, tab);
20 }
```

## 1.2 Merge Sort

```
1 void merge(int left, int mid, int right, int t[]){
2     int a = left, b = mid;
3     int i = left;
4     while(a < mid && b < right)
5         tab[i++] = (t[a] < t[b] ? t[a++] : t[b++]);
6     while(a < mid)
7         tab[i++] = t[a++];
8     while(b < right)
9         tab[i++] = t[b++];
10    for(int k = left; k < right; k++)
11        t[k] = tab[k];
12 }
13
14 void mergesort(int left, int right, int t[]){
15     if(left+1 != right){
16         int m = (left+right)/2;
17         mergesort(left, m, t);
18         mergesort(m, right, t);
19         merge(left, m, right, t);
20     }
21 }
```

# 2 Dynamic Programming

## 2.1 State representation

### 2.1.1 All subsets

This way you can generate all subsets in order ready for DP.

```
1 int sub = whole_set;
2 while(sub > 0){
3
4     sub = (sub-1)&whole_set;
5 }
```

### 3 Graph Algorithms

#### 3.1 Shortest Path

##### 3.1.1 Djikstra

```
1 priority_queue<pair<long long int ,int>> q;
2 dist[1] = 0;
3 q.push({0, 1});
4 while(!q.empty()){
5     int a = q.top().second;
6     q.pop();
7     if(vis[a]) continue;
8     vis[a] = true;
9     for(auto v : adj[a]){
10         int b = v.first;
11         int w = v.second;
12         if(dist[b] > dist[a] + w){
13             dist[b] = dist[a] + w;
14             q.push({-dist[b], b});
15         }
16     }
17 }
```

##### 3.1.2 Belman-Ford

```
1 dist[1] = 0;
2 for(int i = 1; i <= n-1; i++){
3     for(auto e : edges){
4         int a, b, w;
5         tie(a, b, w) = e;
6         if(dist[b] > dist[a]+w){
7             par[b] = a;
8             dist[b] = dist[a]+w;
9         }
10    }
11 }
12 for(auto e : edges){
13     int a, b, w;
14     tie(a, b, w) = e;
15     if(dist[b] > dist[a]+w){
16         found = true;
17         node = b;
18         par[b] = a;
19         dist[b] = dist[a]+w;
20     }
}
```

##### 3.1.3 Floyd-Warshal

```
1 long long int dist[SIZE][SIZE] {};
2 for(int i = 0; i <= n; i++){
3     for(int k = 0; k <= n; k++){
4         dist[i][k] = __INT64_MAX__/2;
5     }
6 }
7 for(int i = 1; i <= n; i++){
8     dist[i][i] = 0;
9     for(auto v : adj[i]){
10         int b = v.first;
11         long long int w = v.second;
12         dist[i][b] = min(w, dist[i][b]);
13         dist[b][i] = min(w, dist[i][b]);
14     }
15 }
16
17 for(int k = 1; k <= n; k++){
18     for(int i = 1; i <= n; i++){
19         for(int j = 1; j <= n; j++){
20             dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
21         }
22     }
23 }
```

## 3.2 Union Find

```
1 int s_size[MX];
2 int link[MX];
3
4 int max_size = 0;
5 int num;
6
7 int find(int x){
8     while(x != link[x])
9         x = link[x];
10    return x;
11 }
12
13 bool same(int a, int b){
14     return find(a) == find(b);
15 }
16
17 void unite(int a, int b){
18     a = find(a);
19     b = find(b);
20     if(s_size[a] < s_size[b])
21         swap(a, b);
22     s_size[a] += s_size[b];
23     max_size = max(max_size, s_size[a]);
24     link[b] = a;
25     num--;
26 }
27
28 int main(){
29     for(int i = 0; i < MX; i++){
30         s_size[i] = 1;
31         link[i] = i;
32     }
33
34     int n, m;
35     cin >> n >> m;
36     num = n;
37     for(int i = 0; i < m; i++){
38         int a, b;
39         cin >> a >> b;
40         if(!same(a, b))
41             unite(a, b);
42         cout << num << " " << max_size << endl;
43     }
44     return 0;
45 }
```

## 3.3 Path through all edges

```
1 set <int> adj[MX];
2
3 bool vis[MX];
4
5 void zero(){
6     for(int i = 0; i < MX; i++)
7         vis[i] = false;
8 }
9
10 int dfs1(int a){
11     int ret = 0;
12     vis[a] = true;
13     for(int b : adj[a]){
14         ret += 1;
15         if(!vis[b])
16             ret += dfs1(b);
17     }
18     return ret;
19 }
20
21 int main(){
22     int n, m;
23     cin >> n >> m;
```

```

24     for(int i = 0; i < m; i++){
25         int a, b;
26         cin >> a >> b;
27         adj[a].insert(b);
28         adj[b].insert(a);
29     }
30
31     zero();
32     if(dfs1(1)/2 != m){
33         cout << "IMPOSSIBLE";
34         return 0;
35     }
36
37     for(int i = 1; i <= n; i++){
38         if(adj[i].size() & 1){
39             cout << "IMPOSSIBLE";
40             return 0;
41         }
42     }
43
44     vector<int> ans;
45
46     stack<int> s;
47     s.push(1);
48     while(!s.empty()){
49         int a = s.top();
50         if(adj[a].size() == 0){
51             ans.push_back(a);
52             s.pop();
53         }
54         else{
55             int b = *adj[a].begin();
56             adj[a].erase(adj[a].begin());
57             adj[b].erase(a);
58             s.push(b);
59         }
60     }
61
62     for(int a : ans)
63         cout << a << " ";
64
65
66     return 0;
67 }
```

### 3.4 Topo Sort

```

1 int n, m;
2 stack <int> res;
3 vector<int> g[100007];
4 int vis[100007];
5 bool imp = false;
6
7 void dfs(int n){
8     vis[n] = 1;
9     for(int v : g[n]){
10         if(vis[v] == 0)
11             dfs(v);
12         else if(vis[v] == 1)
13             imp = true;
14     }
15     vis[n] = 2;
16     res.push(n);
17 }
18
19 int main(){
20     cin >> n >> m;
21     for(int i = 0 ; i < m; i++){
22         int a, b;
23         cin >> a >> b;
24         g[a].push_back(b);
25     }
26 }
```

```

27     for(int i = 1; i <= n; i++){
28         if(vis[i] == 0){
29             dfs(i);
30         }
31     }
32
33     if(!imp){
34         while(!res.empty()){
35             cout << res.top() << " ";
36             res.pop();
37         }
38     }
39     else{
40         cout << "IMPOSSIBLE";
41     }
42
43     return 0;
44 }
```

## 3.5 Flows and Cuts

All problems in this section can be solved using the same basic algorithm defined in 3.5.1.

### 3.5.1 Maximum Flow

This implementation of Ford-Fulkerson algorithm (known as Edmonds-Karp algorithm) uses BFS to check if it's possible to expand the flow through the graph and assign depths from the source. Later it uses DFS to expand te flow.

```

1  ll flow[MX][MX];
2  ll used[MX][MX];
3
4  vector<int> adj[MX];
5  vector<int> radj[MX];
6
7  bool vis[MX];
8
9  bool bfs(int s, int lvl[], const int n){
10    queue<int> q;
11    q.push(s);
12    lvl[s] = 1;
13    while(!q.empty()){
14      int p = q.front();
15      q.pop();
16      for(int v : adj[p]){
17        if(lvl[v] == 0 and flow[p][v] - used[p][v] > 0){
18          lvl[v] = lvl[p]+1;
19          q.push(v);
20        }
21      }
22      for(int v : radj[p]){
23        if(lvl[v] == 0 and used[v][p] > 0){
24          lvl[v] = lvl[p]+1;
25          q.push(v);
26        }
27      }
28    }
29    return lvl[n] > 0;
30  }
31
32  int dfs(int x, ll val, const int lvl[], const int n){
33    vis[x] = true;
34    if(x == n){
35      return val;
36    }
37    for(int v : adj[x]){
38      if(lvl[v] == lvl[x]+1 and flow[x][v] - used[x][v] > 0 and !vis[v]){
39        int r = dfs(v, min(val, flow[x][v]-used[x][v]), lvl, n);
40        if(r > 0){
41          used[x][v] += r;
42          return r;
43        }
44      }
45    }
46  }
```

```

45     }
46     for(int v : radj[x]){
47         if(lvl[v] == lvl[x]+1 and used[v][x] > 0 and !vis[v]){
48             int r = dfs(v, min(val, used[v][x]), lvl, n);
49             if(r > 0){
50                 used[v][x] -= r;
51                 return r;
52             }
53         }
54     }
55     return 0;
56 }

```

To find the flow:

```

1 long long int res = 0;
2
3 while(bfs(1, lvl, n)){
4     res += dfs(1, __INT_MAX__, lvl, n);
5     for(int i = 0 ; i < MX; i++){
6         vis[i] = false;
7         lvl[i] = 0;
8     }
9 }

```

### 3.5.2 Minimum Cut

To find Minimum Cut in a graph we need to find the Maximum Flow and check which edges connect two created disjoint sets of nodes. This modification to finds them:

```

1 long long int res = 0;
2
3 while(bfs(1, lvl, n)){
4     res += dfs(1, __INT_MAX__, lvl, n);
5     for(int i = 0 ; i < MX; i++){
6         vis[i] = false;
7         lvl[i] = 0;
8     }
9 }
10 cout << res << endl;
11 for(int i = 0 ; i < MX; i++){
12     vis[i] = false;
13     lvl[i] = 0;
14 }
15
16 bfs(1, lvl, n);
17 for(int a = 1; a <= n; a++){
18     if(lvl[a] > 0){
19         for(int b : adj[a]){
20             if(lvl[b] == 0){
21                 cout << a << " " << b << endl;
22             }
23         }
24     }
25 }

```

### 3.5.3 Maximum Matching

## 3.6 2SAT

Given logical formula in the conjunctive normal form:

$$(a_1 \vee b_1) \wedge (a_2 \vee b_2) \wedge \dots \wedge (a_n \vee b_n)$$

we can eliminate disjunction by replacing each  $(a_i \vee b_i)$  element with pair:

$$\neg a_i \rightarrow b_i \wedge \neg b_i \rightarrow a_i$$

```

1 vector<int> adj[MX];
2 vector<int> Tadj[MX];
3 stack<int> scc_stack;
4
5 int scc[MX];
6
7 bool vis[MX];
8 int scc_val[MX];
9
10 int not_node(int a){
11     if(!(a&1))
12         return a+1;
13     else
14         return a-1;
15 }
16
17 void dfs1(int a){
18     vis[a] = true;
19     for(int b : adj[a]){
20         if(!vis[b])
21             dfs1(b);
22     }
23     scc_stack.push(a);
24 }
25
26 void dfs2(int a, int scc_n){
27     vis[a] = true;
28     for(int b : Tadj[a]){
29         if(!vis[b])
30             dfs2(b, scc_n);
31     }
32     scc[a] = scc_n;
33 }
34
35 void zero(){
36     for(int i = 0 ; i < MX; i++)
37         vis[i] = false;
38 }
39
40 int main(){
41     int n, m;
42     cin >> n >> m;
43     for(int i = 0 ; i < n; i++){
44         char op;
45         int a, b;
46         cin >> op >> a;
47         if(op == '+'){
48             a *= 2;
49         }
50         else{
51             a *= 2;
52             a++;
53         }
54         cin >> op >> b;
55         if(op == '+'){
56             b *= 2;
57         }
58         else{
59             b *= 2;
60             b++;
61         }
62         Tadj[a].push_back(not_node(b));
63         Tadj[b].push_back(not_node(a));
64         adj[not_node(b)].push_back(a);
65         adj[not_node(a)].push_back(b);
66     }
67
68     for(int i = 1 ; i <= 2*m+1; i++){
69         if(!vis[i])
70             dfs1(i);
71     }
72
73     zero();

```

```

74     int scc_n = 1;
75     while(!scc_stack.empty()){
76         int a = scc_stack.top();
77         scc_stack.pop();
78         if(scc[a] == 0){
79             dfs2(a, scc_n);
80             scc_n++;
81         }
82     }
83
84     for(int i = 2; i <= m*2; i+=2){
85         if(scc[i] == scc[i+1]){
86             cout << "IMPOSSIBLE" << endl;
87             return 0;
88         }
89     }
90     // 1 - false; 2 - true;
91     for(int i = 2; i <= 2*m; i+=2){
92         int a = scc[i];
93         int b = scc[i+1];
94         if(scc_val[a] != 0){
95             scc_val[b] = (scc_val[a] == 1 ? 2 : 1);
96         }
97         if(scc_val[b] != 0){
98             scc_val[a] = (scc_val[b] == 1 ? 2 : 1);
99         }
100        if(a < b){
101            scc_val[a] = 1;
102            scc_val[b] = 2;
103        }
104        else{
105            scc_val[a] = 2;
106            scc_val[b] = 1;
107        }
108    }
109
110    for(int i = 2; i <= m*2; i+=2)
111        cout << (scc_val[scc[i]] == 2 ? "+" : "-") << " ";
112
113    return 0;
114 }

```

## 4 Range Queries

### 4.1 Segment trees

BASE size table:

a	$2 \cdot 10^5$	$5 \cdot 10^5$	$10^6$
$\log_2 a$	18	19	20

#### 4.1.1 Point-Range Trees

```

1 int sal[2*BASE];
2
3 void add(int p, int val){
4     p += BASE;
5     sal[p] += val;
6     while(p > 0){
7         p >>= 1;
8         sal[p] = sal[2*p] + sal[2*p+1];
9     }
10 }
11
12 int query(int p, int q){
13     int ret = 0 ;
14     p += BASE -1;
15     q += BASE +1;
16     while(p>>1 != q>>1){
17         if(not(p&1)){

```

```

18         ret += sal[p+1];
19     }
20     if(q&1){
21         ret += sal[q-1];
22     }
23     p >>= 1;
24     q >>= 1;
25 }
26 return ret;
27 }

```

#### 4.1.2 Range-Range Trees

- range ADD insert, range MAX value query

```

1 void max_add(int a, int b, int val, int k=1, int x=0, int y=BASE-1){
2     if(a <= x and y <= b){
3         max_tree[k][1] += val;
4         return;
5     }
6     max_tree[2*k][1] += max_tree[k][1];
7     max_tree[2*k+1][1] += max_tree[k][1];
8     max_tree[k][0] += max_tree[k][1];
9     max_tree[k][1] = 0;
10
11    int d = (x+y)/2;
12    if(a <= d){
13        max_add(a, b, val, 2*k, x, d);
14    }
15    if(b > d){
16        max_add(a, b, val, 2*k+1, d+1, y);
17    }
18    max_tree[k][0] = max(max_tree[2*k+1][0] + max_tree[2*k+1][1], max_tree[2*k][0]
19    + max_tree[2*k][1]);
20 }

11 max_query(int a, int b, int k=1, int x=0, int y=BASE-1){
2 if(a <= x and y <= b){
3     return max_tree[k][0] + max_tree[k][1];
4 }
5 max_tree[2*k][1] += max_tree[k][1];
6 max_tree[2*k+1][1] += max_tree[k][1];
7 max_tree[k][0] += max_tree[k][1];
8 max_tree[k][1] = 0;
9 int d = (x+y)/2;
10 ll ret = -INF;
11 if(a <= d){
12     ret = max(ret, max_query(a, b, 2*k, x, d));
13 }
14 if(b > d){
15     ret = max(ret, max_query(a, b, 2*k+1, d+1, y));
16 }
17 return ret;
18 }

```

- range ADD insert, range SUM query

```

1 void sum_add(int a, int b, int val, int k=1, int x=0, int y=BASE-1){
2     if(a <= x and y <= b){
3         sum_tree[k][1] += val;
4         return;
5     }
6     sum_tree[2*k][1] += sum_tree[k][1];
7     sum_tree[2*k+1][1] += sum_tree[k][1];
8     sum_tree[k][0] += sum_tree[k][1]*(y-x+1);
9     sum_tree[k][1] = 0;
10    int d = (x+y)/2;
11    if(a <= d){
12        int w = (d <= b ? d : b)+1;
13        w -= (x <= a ? a : x);
14        sum_tree[k][0] += val*(w);
15        sum_add(a, b, val, 2*k, x, d);
16    }

```

```

17     if(b > d){
18         int w = (y <= b ? y : b)+1;
19         w -= (d+1 <= a ? a : d+1);
20         sum_tree[k][0] += val*(w);
21         sum_add(a, b, val, 2*k+1, d+1, y);
22     }
23 }

1 11 sum_query(int a, int b, int k=1, int x=0, int y=BASE-1){
2     if(a <= x and y <= b){
3         return sum_tree[k][0] + sum_tree[k][1]*(y-x+1);
4     }
5     sum_tree[2*k][1] += sum_tree[k][1];
6     sum_tree[2*k+1][1] += sum_tree[k][1];
7     sum_tree[k][0] += sum_tree[k][1]*(y-x+1);
8     sum_tree[k][1] = 0;
9     int d = (x+y)/2;
10    ll ret = 0;
11    if(a <= d){
12        ret += sum_query(a, b, 2*k, x, d);
13    }
14    if(b > d){
15        ret += sum_query(a, b, 2*k+1, d+1, y);
16    }
17    return ret;
18 }
19 int sal[2*BASE];

```

## 4.2 Max Subarray Sum

```

1 11 ssum[2*BASE];
2 11 prfx[2*BASE];
3 11 sufpx[2*BASE];
4 11 tsum[2*BASE];
5
6 void insert(int val, int p){
7     p+=BASE;
8     ssum[p] = val;
9     prfx[p] = val;
10    sufpx[p] = val;
11    tsum[p] = val;
12    int l, r;
13    while(p){
14        p >>= 1;
15        l = 2*p;
16        r = 2*p+1;
17        tsum[p] = tsum[l] + tsum[r];
18        prfx[p] = max(prfx[l], tsum[l] + prfx[r]);
19        sufpx[p] = max(sufpx[l] + tsum[r], sufpx[r]);
20        ssum[p] = max(sufpx[l] + prfx[r], max(ssum[l], ssum[r]));
21    }
22 }

```

## 5 Tree Algorithms

### 5.1 Binary Lifting

```

1 int main(){
2     int n, q;
3     cin >> n >> q;
4     boss[1][0] = 0;
5     for(int i = 2; i <= n; i++){
6         cin >> boss[i][0];
7     }
8     for(int i = 1; i < LOG_MX; i++){
9         for(int emp = 1; emp <= MX; emp++){
10             boss[emp][i] = boss[boss[emp][i-1]][i-1];
11         }
12     }

```

## 5.2 Tree Traversal

```
1 vector <int> adj [MX];
2 int subtree[MX];
3 int flat[MX];
4 int vals[MX];
5
6 int idx = 1;
7 int dfs(int a, int p){
8     flat[a] = idx;
9     idx++;
10    for(int b : adj[a]){
11        if(b == p)
12            continue;
13        subtree[a] += dfs(b, a)+1;
14    }
15    return subtree[a];
16 }
```

## 5.3 LCA

LCA can be found using binary lifting or by building a segment tree build on pre- and postorder.

# 6 String Algorithms

## 6.1 Trie

```
1 int trie[TMX][30];
2 bool stop[TMX];
3 int next_node = 1;
4 int dp[MX];
5
6 void insert(string s){
7     int idx = 0;
8     for(char c : s){
9         if(trie[idx][c-'a'] == 0){
10             trie[idx][c-'a'] = next_node;
11             next_node++;
12         }
13         idx = trie[idx][c-'a'];
14     }
15     stop[idx] = true;
16 }
17
18 int main(){
19     string text;
20     int n;
21     cin >> text >> n;
22     for(int i = 0; i < n; i++){
23         string s;
24         cin >> s;
25         insert(s);
26     }
```

## 6.2 Pattern Finding

### 6.2.1 KMP

```
1 int pi[MX];
2
3 int main(){
4     string text, pattern;
5     cin >> text >> pattern;
6     string s = pattern + "+" + text;
7     for(int i = 1; i < s.size(); i++){
8         int j = pi[i-1];
9         while(j > 0 and s[i] != s[j])
10             j = pi[j-1];
11         if(s[i] == s[j])
12             j++;
```

```

13     pi[i] = j;
14 }
15
16 int res = 0;
17 for(int i = 0; i < s.size(); i++){
18     if(pi[i] == pattern.size())
19         res++;
20 }
21
22 cout << res;
23 return 0;

```

## 6.2.2 Hashing

```

1 const int MOD = 1e9+9;
2 const int P = 9973;
3 const int MX = 1e6+7;
4
5 ll ppow[MX];
6 ll pfx_hash[MX];
7
8 int res = 0;
9
10 int main(){
11     string s, pattern;
12     cin >> s >> pattern;
13     ppow[0] = 1;
14     for(int i = 1 ; i < MX; i++){
15         ppow[i] = (ppow[i-1]*P)%MOD;
16     }
17
18     ll pattern_hash = 0;
19     for(int i = 0; i < pattern.size(); i++){
20         pattern_hash = (pattern_hash + (pattern[i]-'a'+1)*ppow[i])%MOD;
21     }
22
23     for(int i = 0; i < s.size(); i++){
24         pfx_hash[i+1] = (pfx_hash[i] + (s[i]-'a'+1)*ppow[i])%MOD;
25     }
26
27     for(int i = 0; i+pattern.size()-1 < s.size(); i++){
28         int a = i, b = i+pattern.size();
29         ll sub_hash = pfx_hash[b] - pfx_hash[a]+MOD;
30         sub_hash %= MOD;
31         if(sub_hash == (pattern_hash*ppow[a])%MOD){
32             res++;
33         }
34     }
35     cout << res << endl;
36     return 0;

```

### 6.3 Palindromes

Fiding longest palindromic substring.

```
1 int p[MX];
2
3 int main(){
4     string txt1, txt = "#";
5     cin >> txt1;
6     for(char c : txt1){
7         string s{c};
8         txt += s + "#";
9     }
10    txt = "^" + txt + "$";
11    int l = 1, r = 1;
12    for(int i = 1; i < txt.size(); i++){
13        p[i] = max(0, min(r-i, p[l+r-i]));
14        while(txt[i-p[i]] == txt[i+p[i]])
15            p[i]++;
16        if(i+p[i] > r){
17            l = i-p[i];
18            r = i+p[i];
19        }
20    }
21    int maxi = 0;
22    int imaxi;
23    for(int i = 0; i < txt.size(); i++){
24        if(p[i] > maxi){
25            maxi = p[i];
26            imaxi = i;
27        }
28    }
29    string res = txt.substr(imaxi-maxi+1,maxi*2-1);
30    for(auto c : res){
31        if(c != '#')
32            cout << c;
33    }
```

## 7 Mathematics

### 7.1 Fermat's Theorem

Modular inverse:

If m is prime, then:  $x^{-1} \bmod m = x^{m-2} \bmod m$

$$x^{\varphi(m)} \bmod m = x^{k \bmod (m-1)} \bmod m$$

```
1 int exp(int a, int b, int MOD){
2     if(b == 0)
3         return 1;
4     if(b & 1){
5         return ((11)a * exp(a, b-1, MOD))%MOD;
6     }
7     ll tmp = exp(a, b/2, MOD);
8     return (tmp*tmp)%MOD;
9 }
```

### 7.2 Fast Fibonacci

```
1 void mul(ll a[][2], ll b[][2]){
2     ll res[2][2];
3     res[0][0] = a[0][0]*b[0][0] + a[0][1]*b[1][0];
4     res[0][1] = a[0][0]*b[0][1] + a[0][1]*b[1][1];
5     res[1][0] = a[1][0]*b[0][0] + a[1][1]*b[1][0];
6     res[1][1] = a[1][0]*b[0][1] + a[1][1]*b[1][1];
```

```

7     a[0][0] = res[0][0] % MOD;
8     a[0][1] = res[0][1] % MOD;
9     a[1][0] = res[1][0] % MOD;
10    a[1][1] = res[1][1] % MOD;
11 }
12
13
14 int main(){
15     ll n;
16     cin >> n;
17     ll m_pow[2][2] =
18         {{0, 1},
19          {1, 1}};
20     ll m[2][2] =
21         {{1, 0},
22          {0, 1}};
23
24     auto pow = bitset<64> (n);
25
26     for(int i = 0 ; i < 64; i++){
27         if(pow[i]){
28             mul(m, m_pow);
29         }
30         mul(m_pow, m_pow);
31     }
32
33     cout << m[0][1];
34
35     return 0;
36 }
```

## 8 Geometry

### 8.1 Point location

```

1 ll cros = cross(a, b, p);
2 if(cros > 0)
3     cout << "LEFT\n";
4 else if(cros == 0)
5     cout << "TOUCH\n";
6 else
7     cout << "RIGHT\n";
```

### 8.2 Point in polygon

```

1 int inter(pii a, pii b, pii c, pii d){
2     pii p1 = {min(a.first, b.first), max(a.second, b.second)};
3     pii p2 = {max(a.first, b.first), min(a.second, b.second)};
4     pii p3 = {min(c.first, d.first), max(c.second, d.second)};
5     pii p4 = {max(c.first, d.first), min(c.second, d.second)};
6
7     if (p3.first > p2.first || p4.first < p1.first || p3.second < p2.second || p4.second
8         > p1.second)
9         return 0;
10
11     if (sign(cross(a,b,c)) * sign(cross(a,b,d)) <= 0 && sign(cross(c,d,a)) * sign(cross(
12         c,d,b)) <= 0) {
13         return 1;
14     }
15
16     return 0;
17 }
18
19 bool point_on_line(pii a, pii b, pii p){
20     int c = cross(a, b, p);
21     pii ld = {min(a.first, b.first), min(a.second, b.second)};
22     pii pg = {max(a.first, b.first), max(a.second, b.second)};
23     if(c == 0){
24         if(!(p.first < ld.first || p.second < ld.second || p.first > pg.first || p.
second > pg.second))
25             return true;
26     }
27 }
```

```

25     return false;
26 }
27
28 int count = 0;
29 bool boundary = false;
30 for(int i = 0; i < n; i++){
31     if (point_on_line(polygon[i], polygon[(i == n-1 ? 0 : i+1)], p)){
32         boundary = true;
33     }
34     if (point_on_line(p, out, polygon[i])){
35         if(cross(p, out, polygon[(i == 0 ? n-1 : i-1)]) * cross(p, out,polygon[(i == n-1 ? 0 : i+1)]) < 0){
36             count--;
37         }
38     }
39     count += inter(p, out, polygon[i], polygon[(i == n-1 ? 0 : i+1)]);
40 }
41
42 if (boundary){
43     cout << "BOUNDARY" << endl;
44 }
45 else if (count & 1){
46     cout << "INSIDE" << endl;
47 }
48 else {
49     cout << "OUTSIDE" << endl;
50 }

```

### 8.3 Convex hull

Only finds the smallest convex hull. Points on the hull are omitted.

```

1 11 cross2(pii a, pii b) {
2     return (ll)a.first * b.second - (ll)a.second * b.first;
3 }
4
5 int cross3(pii a, pii b, pii c){
6     pii va, vb;
7     va.first = b.first-a.first;
8     va.second = b.second-a.second;
9     vb.first = c.first-b.first;
10    vb.second = c.second-b.second;
11    return sign((ll)va.first*vb.second - (ll)vb.first*va.second);
12 }
13
14 sort(points.begin(), points.end(), [] (pii l, pii r){
15     int o = cross3(l, r, p0);
16     if (o == 0) {
17         return norm(sub(p0,l)) < norm(sub(p0,r));
18     }
19     return o < 0;
20 });
21
22 vector<pii> hull;
23
24 for (int i = 0; i < n; i++){
25     while(hull.size() > 1 && cross3(hull[hull.size()-2], hull[hull.size()-1], points[i])
26           >= 0){
27         hull.pop_back();
28     }
29     hull.push_back(points[i]);
30 }

```