Algorithms and Data Structures for Competitive Programming

May 12, 2013

Contents

1 Code Templates

1.1 C++ Header

```cpp
#include <algorithm>
#include <cassert>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <ctime>
#include <iomanip>
#include <iostream>
#include <map>
#include <queue>
#include <set>
#include <sstream>
#include <stack>
#include <string>
#include <utility>
#include <vector>
using namespace std;

#define all(o) (o).begin(), (o).end()
#define allr(o) (o).rbegin(), (o).rend()
#define pb push_back
const int INF = 2147483647;
const double EPS = 1e-9;
const double pi = acos(-1);
typedef long long ll;
typedef unsigned long long ull;
typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
typedef vector<vi> vvi;
typedef vector<vii> vvii;
template <class T> T mod(T a, T b) { return (a % b + b) % b; }
template <class T> int size(T x) { return x.size(); }
```

1.2 Java Template

```java
import java.util.*;
import java.math.*;
import java.io.*;

public class Main {
        public static void main(String[] args) throws Exception {
                Scanner in = new Scanner(System.in);
                PrintWriter out = new PrintWriter(System.out, false);
                // code
                out.flush();
        }
}
```

2 Data Structures

2.1 Union-Find

```
1  class union_find {
2  private:
3          int* parent;
4          int cnt;
5
6  public:
7          union_find(int n) {
8                  parent = new int[cnt = n];
9                  for (int i = 0; i < cnt; i++)
10                         parent[i] = i;
11         }
12
13         union_find(const union_find& other) {
14                 parent = new int[cnt = other.cnt];
15                 for (int i = 0; i < cnt; i++)
16                         parent[i] = other.parent[i];
17         }
18
19         ~union_find() {
20                 if (parent) {
21                         delete[] parent;
22                         parent = NULL;
23                 }
24         }
25
26         int find(int i) {
27                 assert(parent != NULL);
28                 return parent[i] == i ? i : (parent[i] = find(parent[i]));
29         }
30
31         bool unite(int i, int j) {
32                 assert(parent != NULL);
33
34                 int ip = find(i),
35                     jp = find(j);
36
37                 parent[ip] = jp;
38                 return ip != jp;
39         }
40 };
```

2.2 Fenwick Tree

A Fenwick Tree is a data structure that represents an array of n numbers. It supports adjusting the i-th element in $O(\log n)$ time, and computing the sum of numbers in the range $i..j$ in $O(\log n)$ time. It only needs $O(n)$ space.

```
1  class fenwick_tree {
2  private:
3          int* arr;
```

```
 4          int cnt;
 5
 6          inline int lsone(int n) {
 7                  return n & -n;
 8          }
 9
10  public:
11          fenwick_tree(int n) {
12                  arr = new int[(cnt = n) + 1];
13                  memset(arr, 0, (cnt + 1) << 2);
14          }
15
16          ~fenwick_tree() {
17                  if (arr) {
18                          delete[] arr;
19                          arr = NULL;
20                  }
21          }
22
23          fenwick_tree(const fenwick_tree& other) {
24                  arr = new int[(cnt = other.cnt) + 1];
25                  for (int i = 0; i <= cnt; i++)
26                          arr[i] = other.arr[i];
27          }
28
29          void adjust(int i, int v) {
30                  assert(arr != NULL);
31                  assert(i >= 0 && i < cnt);
32
33                  i++;
34                  while (i <= cnt) {
35                          arr[i] += v;
36                          i += lsone(i);
37                  }
38          }
39
40          int rsq(int i) {
41                  assert(arr != NULL);
42
43                  if (i < 0)
44                          return 0;
45                  if (i >= cnt)
46                          return rsq(cnt - 1);
47
48                  i++;
49                  int sum = 0;
50                  while (i) {
51                          sum += arr[i];
52                          i -= lsone(i);
53                  }
54
55                  return sum;
56          }
```

```
57
58          inline int rsq(int i, int j) {
59                  return rsq(j) - rsq(i - 1);
60          }
61
62          inline int get(int i) {
63                  return rsq(i) - rsq(i - 1);
64          }
65  };
```

2.3   Matrix

```
 1  template <class T>
 2  class matrix {
 3  private:
 4          vector<T> data;
 5          int cnt;
 6          inline T& at(int i, int j) {
 7                  return data[i * cols + j];
 8          }
 9
10  public:
11          int rows, cols;
12
13          matrix(int r, int c) {
14                  rows = r; cols = c;
15                  data = vector<T>(cnt = rows * cols);
16                  for (int i = 0; i < cnt; i++)
17                          data[i] = T(0);
18          }
19
20          matrix(const matrix& other) {
21                  rows = other.rows; cols = other.cols;
22                  data = vector<T>(cnt = rows * cols);
23                  for (int i = 0; i < cnt; i++)
24                          data[i] = other.data[i];
25          }
26
27          T& operator()(int i, int j) {
28                  return at(i, j);
29          }
30
31          void operator +=(const matrix& other) {
32                  assert(rows == other.rows && cols == other.cols);
33                  for (int i = 0; i < cnt; i++)
34                          data[i] += other.data[i];
35          }
36
37          void operator -=(const matrix& other) {
38                  assert(rows == other.rows && cols == other.cols);
39                  for (int i = 0; i < cnt; i++)
40                          data[i] -= other.data[i];
```

```
41              }
42
43              void operator *=(T other) {
44                      for (int i = 0; i < cnt; i++)
45                              data[i] *= other;
46              }
47
48              matrix<T> operator +(const matrix& other) {
49                      matrix<T> res(*this);
50                      res += other;
51                      return res;
52              }
53
54              matrix<T> operator -(const matrix& other) {
55                      matrix<T> res(*this);
56                      res -= other;
57                      return res;
58              }
59
60              matrix<T> operator *(T other) {
61                      matrix<T> res(*this);
62                      res *= other;
63                      return res;
64              }
65
66              matrix<T> operator *(const matrix& other) {
67                      assert(cols == other.rows);
68                      matrix<T> res(rows, other.cols);
69                      for (int i = 0; i < rows; i++)
70                              for (int j = 0; j < other.cols; j++)
71                                      for (int k = 0; k < cols; k++)
72                                              res(i, j) += at(i, k) * other.data[k *
                                                      other.cols + j];
73                      return res;
74              }
75
76              matrix<T> transpose() {
77                      matrix<T> res(cols, rows);
78                      for (int i = 0; i < rows; i++)
79                              for (int j = 0; j < cols; j++)
80                                      res(j, i) = at(i, j);
81                      return res;
82              }
83
84              matrix<T> pow(int p) {
85                      assert(rows == cols);
86                      matrix<T> res(rows, cols), sq(*this);
87                      for (int i = 0; i < rows; i++)
88                              res(i, i) = 1;
89
90                      while (p) {
91                              if (p & 1) res = res * sq;
92                              p >>= 1;
```

```
 93                                 if (p) sq = sq * sq;
 94                         }
 95
 96                         return res;
 97                 }
 98
 99         matrix<T> triangular_form() {
100                 matrix<T> res(*this);
101                 for (int k = 0; k < res.rows; k++) {
102                         int i_max = k;
103                         for (int i = k + 1; i < res.rows; i++) {
104                                 if (res.at(i_max, k) < res.at(i, k)) {
105                                         i_max = i;
106                                 }
107                         }
108
109                         assert(res.at(i_max, k) != T(0));
110
111                         for (int j = 0; j < res.cols; j++) {
112                                 swap(res.at(k, j), res.at(i_max, j));
113                                 //T tmp = res.at(k, j);
114                                 //res.at(k, j) = res.at(i_max, j);
115                                 //res.at(i_max, j) = tmp;
116                         }
117
118                         for (int i = k + 1; i < res.rows; i++) {
119                                 for (int j = k + 1; j < res.cols; j++) {
120                                         res.at(i, j) -= res.at(k, j) * res.at(i, k)
                                                / res.at(k, k);
121                                 }
122
123                                 res.at(i, k) = T(0);
124                         }
125
126                         for (int j = res.cols - 1; j >= k; j--)
127                         {
128                                 res.at(k, j) /= res.at(k, k);
129                         }
130                 }
131
132                 return res;
133         }
134 };
```

2.4 Trie

```
 1 template <class T>
 2 class trie {
 3 private:
 4         struct node {
 5                 map<T, node*> children;
 6                 int prefixes;
 7                 int words;
```

```cpp
 8
 9                    node() {
10                            prefixes = words = 0;
11                    }
12            };
13
14            node* root;
15
16  public:
17            trie() {
18                    root = new node();
19            }
20
21            template <class I>
22            void insert(I begin, I end) {
23                    node* cur = root;
24                    while (true) {
25                            cur->prefixes++;
26                            if (begin == end) {
27                                    cur->words++;
28                                    break;
29                            } else {
30                                    T head = *begin;
31                                    typename map<T, node*>::const_iterator it =
                                            cur->children.find(head);
32                                    if (it == cur->children.end()) it =
                                            cur->children.insert(pair<T, node*>(head, new
                                            node())).first;
33                                    begin++;
34                                    cur = it->second;
35                            }
36                    }
37            }
38
39            template<class I>
40            int countMatches(I begin, I end) {
41                    node* cur = root;
42                    while (true) {
43                            if (begin == end) {
44                                    return cur->words;
45                            } else {
46                                    T head = *begin;
47                                    typename map<T, node*>::const_iterator it =
                                            cur->children.find(head);
48                                    if (it == cur->children.end()) return 0;
49                                    begin++;
50                                    cur = it->second;
51                            }
52                    }
53            }
54
55            template<class I>
56            int countPrefixes(I begin, I end) {
```

```
57              node* cur = root;
58              while (true) {
59                      if (begin == end) {
60                              return cur->prefixes;
61                      } else {
62                              T head = *begin;
63                              typename map<T, node*>::const_iterator it =
                                      cur->children.find(head);
64                              if (it == cur->children.end()) return 0;
65                              begin++;
66                              cur = it->second;
67                      }
68              }
69      }
70 };
```

3   Graphs

3.1   Breadth-First Search

An implementation of a breadth-first search that counts the number of edges on the shortest path from the starting vertex to the ending vertex in the specified unweighted graph (which is represented with an adjacency list). Note that it assumes that the two vertices are connected. It runs in $O(|V| + |E|)$ time.

```
1 int bfs(int start, int end, vvi& adj_list) {
2      queue<ii> Q;
3      Q.push(ii(start, 0));
4
5      while (true) {
6              ii cur = Q.front(); Q.pop();
7
8              if (cur.first == end)
9                      return cur.second;
10
11             vi& adj = adj_list[cur.first];
12             for (vi::iterator it = adj.begin(); it != adj.end(); it++)
13                     Q.push(ii(*it, cur.second + 1));
14     }
15 }
```

Another implementation that doesn't assume the two vertices are connected. If there is no path from the starting vertex to the ending vertex, a **-1** is returned.

```
1 int bfs(int start, int end, vvi& adj_list) {
2      set<int> visited;
3      queue<ii> Q;
4      Q.push(ii(start, 0));
5      visited.insert(start);
6
7      while (!Q.empty()) {
8              ii cur = Q.front(); Q.pop();
9
10             if (cur.first == end)
11                     return cur.second;
12
```

```
13                       vi& adj = adj_list[cur.first];
14                       for (vi::iterator it = adj.begin(); it != adj.end(); it++)
15                               if (visited.find(*it) == visited.end()) {
16                                       Q.push(ii(*it, cur.second + 1));
17                                       visited.insert(*it);
18                               }
19               }
20
21           return -1;
22   }
```

3.2   Single-Source Shortest Paths

3.2.1   Dijkstra's algorithm

An implementation of Dijkstra's algorithm that returns a map, where each key of the map is a vertex reachable from the starting vertex, and the value is a tuple of the length from the starting vertex to the current vertex and the vertex that preceeds the current vertex on the shortest path from the starting vertex to the current vertex.

```
1    map<int, ii> dijkstra_path(int start, vvii& adj_list) {
2            map<int, ii> parent;
3            priority_queue<ii, vii, greater<ii> > pq;
4            pq.push(ii(0, start));
5            parent[start] = ii(0, start);
6
7            while (!pq.empty()) {
8                    ii cur = pq.top(); pq.pop();
9
10                   if (cur.first > parent[cur.second].first)
11                           continue;
12
13                   vii &vtmp = adj_list[cur.second];
14                   for (vii::iterator it = vtmp.begin(); it != vtmp.end(); it++) {
15                           if (parent.find(it->second) == parent.end() ||
                                 parent[it->second].first > cur.first + it->first) {
16                                   parent[it->second] = ii(cur.first + it->first,
                                         cur.second);
17                                   pq.push(ii(cur.first + it->first, it->second));
18                           }
19                   }
20           }
21
22           return parent;
23   }
```

A much faster implementation of Dijkstra's algorithm. It runs in $\Theta(|E| \log |V|)$ time.

```
1    #define TSWP(x, y) tmp = x, x = y, y = tmp
2    #define SWP(a, b) TSWP(loc[heap[a]], loc[heap[b]]), TSWP(heap[a], heap[b])
3
4    pair<int*, int*> dijkstra_fast(int n, int s, vector<ii> *adj) {
5        int *heap = new int[n],
6            *loc = new int[n],
7            *dist = new int[n],
```

```
 8              *parent = new int[n],
 9               tmp;
10
11          for (int i = 0; i < n; i++) {
12              heap[i] = i;
13              loc[i] = i;
14              dist[i] = INF;
15              parent[i] = -1;
16          }
17
18          SWP(0, s), dist[s] = 0;
19          for (int left = n; left > 0; ) {
20              int cur = heap[0];
21              if (dist[cur] == INF) break;
22              SWP(0, left-1), left--;
23              int at = 0;
24              while (at < left) {
25                  int l = 2*at + 1, r = l + 1, nxt = -1, ad = dist[heap[at]];
26                  if (r >= left) {
27                      if (l < left && ad > dist[heap[l]]) nxt = l;
28                  } else {
29                      int ld = dist[heap[l]], rd = dist[heap[r]];
30                      if (ad > ld && ad > rd) nxt = ld <= rd ? l : r;
31                      else if (ad > ld) nxt = l;
32                      else if (ad > rd) nxt = r;
33                  }
34                  if (nxt != -1) SWP(at, nxt), at = nxt;
35                  else break;
36              }
37
38              for (int i = 0, cnt = size(adj[cur]); i < cnt; i++) {
39                  int nxt = adj[cur][i].first,
40                      ndist = dist[cur] + adj[cur][i].second;
41                  if (ndist < dist[nxt]) {
42                      dist[nxt] = ndist;
43                      parent[nxt] = cur;
44                      int at = loc[nxt];
45                      while (at > 0) {
46                          int p = at/2;
47                          if (dist[nxt] < dist[heap[p]]) SWP(at, p), at = p;
48                          else break;
49                      }
50                  }
51              }
52          }
53
54          delete[] heap;
55          delete[] loc;
56          return pair<int*, int*>(dist, parent);
57      }
```

3.2.2   Bellman-Ford algorithm

The Bellman-Ford algorithm solves the single-source shortest paths problem in $O(|V||E|)$ time. It is slower than Dijkstra's algorithm, but it works on graphs with negative edges and has the ability to detect negative cycles, neither of which Dijkstra's algorithm can do.

```
1  int* bellman_ford(int n, int s, vii* adj, bool& has_negative_cycle) {
2          has_negative_cycle = false;
3          int* dist = new int[n];
4          for (int i = 0; i < n; i++)
5                  dist[i] = INF;
6          dist[s] = 0;
7          for (int i = 0; i < n - 1; i++)
8                  for (int j = 0; j < n; j++)
9                          for (int k = 0, len = size(adj[j]); k < len; k++)
10                                 dist[adj[j][k].first] = min(dist[adj[j][k].first],
11                                         dist[j] + adj[j][k].second);
12          for (int j = 0; j < n; j++)
13                  for (int k = 0, len = size(adj[j]); k < len; k++)
14                          if (dist[j] + adj[j][k].second < dist[adj[j][k].first])
15                                  has_negative_cycle = true;
16          return dist;
17  }
```

3.3   All-Pairs Shortest Paths

3.3.1   Floyd-Warshall algorithm

The Floyd-Warshall algorithm solves the all-pairs shortest paths problem in $O(|V|^3)$ time.

```
1  void floyd_warshall(int** arr, int n) {
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 (arr[i][k] != INF && arr[k][j] != INF)
6                                          arr[i][j] = min(arr[i][j], arr[i][k] +
7                                                  arr[k][j]);
8  }
```

3.4   Connected Components

3.5   Strongly Connected Components

3.5.1   Tarjan's algorithm

Tarjan's algorithm finds strongly connected components of a directed graph in $O(|V| + |E|)$ time.

```
1  stack<int> t_s;
2  set<int> t_z;
3  int t_index;
4  int* t_idx;
5  int* t_ll;
6
7  void t_strongconnect(int v, vvi& adj) {
8          t_idx[v] = t_ll[v] = t_index++;
```

```
 9              t_s.push(v);
10              t_z.insert(v);
11
12              int cnt = size(adj[v]);
13              for (int i = 0; i < cnt; i++) {
14                      int w = adj[v][i];
15                      if (t_idx[w] == -1) {
16                              t_strongconnect(w, adj);
17                              t_ll[v] = min(t_ll[v], t_ll[w]);
18                      } else if (t_z.find(w) != t_z.end())
19                              t_ll[v] = min(t_ll[v], t_idx[w]);
20              }
21
22              if (t_ll[v] == t_idx[v])
23                      while (true) {
24                              // Vertices from top of stack down to v form a SCC
25                              int w = t_s.top(); t_s.pop(); t_z.erase(t_z.find(w));
26                              if (w == v) break;
27                      }
28 }
29
30 int* tarjan_scc(vvi adj) {
31         int n = size(adj);
32         t_idx = new int[n];
33         t_ll = new int[n];
34         t_index = 0;
35
36         for (int i = 0; i < n; i++)
37                 t_idx[i] = t_ll[i] = -1;
38
39         for (int i = 0; i < n; i++)
40                 if (t_idx[i] == -1)
41                         t_strongconnect(i, adj);
42
43         delete[] t_idx;
44         return t_ll;
45 }
```

3.6   Minimum Spanning Tree

3.6.1   Kruskal's algorithm

```
 1 #include "../data-structures/unionfind.cpp"
 2
 3 // n is the number of vertices
 4 // edges is a list of edges of the form (weight, (a, b))
 5 // the edges in the minimum spanning tree are returned on the same form
 6 vector<pair<int, ii> > mst(int n, vector<pair<int, ii> > edges) {
 7         union_find uf(n);
 8         sort(all(edges));
 9         vector<pair<int, ii> > res;
10         for (int i = 0, cnt = size(edges); i < cnt; i++)
```

```
11                        if (uf.find(edges[i].second.first) !=
                              uf.find(edges[i].second.second)) {
12                                res.push_back(edges[i]);
13                                uf.unite(edges[i].second.first, edges[i].second.second);
14                        }
15                return res;
16      }
```

3.7 Topological Sort

3.7.1 Modified Depth-First Search

```
1  void tsort_dfs(int cur, char* color, const vvi& adj, stack<int>& res, bool&
       has_cycle) {
2          color[cur] = 1;
3          for (int i = 0, cnt = size(adj[cur]); i < cnt; i++) {
4                  int nxt = adj[cur][i];
5                  if (color[nxt] == 0)
6                          tsort_dfs(nxt, color, adj, res, has_cycle);
7                  else if (color[nxt] == 1)
8                          has_cycle = true;
9                  if (has_cycle) return;
10         }
11         color[cur] = 2;
12         res.push(cur);
13     }
14
15     vi tsort(int n, vvi adj, bool& has_cycle) {
16         has_cycle = false;
17         stack<int> S;
18         vi res;
19         char* color = new char[n];
20         memset(color, 0, n);
21         for (int i = 0; i < n; i++) {
22                 if (!color[i]) {
23                         tsort_dfs(i, color, adj, S, has_cycle);
24                         if (has_cycle) return res;
25                 }
26         }
27         while (!S.empty()) res.push_back(S.top()), S.pop();
28         return res;
29     }
```

3.8 Maximum Flow

3.8.1 Edmonds Karp's algorithm

An implementation of Edmonds Karp's algorithm that runs in $O(|V||E|^2)$. It computes the maximum flow of a flow network.

```
1  struct mf_edge {
2          int u, v, w;
3          mf_edge* rev;
4          mf_edge(int _u, int _v, int _w, mf_edge* _rev = NULL) {
5                  u = _u; v = _v; w = _w; rev = _rev;
```

```
 6              }
 7    };
 8
 9    int max_flow(int n, int s, int t, vii* adj) {
10            vector<mf_edge*>* g = new vector<mf_edge*>[n];
11            for (int i = 0; i < n; i++) {
12                    for (int j = 0, len = size(adj[i]); j < len; j++) {
13                            mf_edge *cur = new mf_edge(i, adj[i][j].first,
                                    adj[i][j].second),
14                                                     *rev = new mf_edge(adj[i][j].first, i, 0,
                                                             cur);
15                            cur->rev = rev;
16                            g[i].push_back(cur);
17                            g[adj[i][j].first].push_back(rev);
18                    }
19            }
20            int flow = 0;
21            mf_edge** back = new mf_edge*[n];
22            while (true) {
23                    for (int i = 0; i < n; i++) back[i] = NULL;
24                    queue<int> Q; Q.push(s);
25                    while (!Q.empty()) {
26                            int cur = Q.front(); Q.pop();
27                            if (cur == t) break;
28                            for (int i = 0, len = size(g[cur]); i < len; i++) {
29                                    mf_edge* nxt = g[cur][i];
30                                    if (nxt->v != s && nxt->w > 0 && back[nxt->v] ==
                                            NULL) {
31                                            back[nxt->v] = nxt;
32                                            Q.push(nxt->v);
33                                    }
34                            }
35                    }
36                    mf_edge* cure = back[t];
37                    if (cure == NULL) break;
38                    int cap = INF;
39                    while (true) {
40                            cap = min(cap, cure->w);
41                            if (cure->u == s) break;
42                            cure = back[cure->u];
43                    }
44                    assert(cap > 0 && cap < INF);
45                    cure = back[t];
46                    while (true) {
47                            cure->w -= cap;
48                            cure->rev->w += cap;
49                            if (cure->u == s) break;
50                            cure = back[cure->u];
51                    }
52                    flow += cap;
53            }
54            // instead of deleting g, we could also
55            // use it to get info about the actual flow
```

```
56              for (int i = 0; i < n; i++)
57                      for (int j = 0, len = size(g[i]); j < len; j++)
58                              delete g[i][j];
59              delete[] g;
60              delete[] back;
61              return flow;
62      }
```

3.9   Minimum Cost Maximum Flow

An implementation of Edmonds Karp's algorithm, modified to find shortest path to augment each time (instead of just any path). It computes the maximum flow of a flow network, and when there are multiple maximum flows, finds the maximum flow with minimum cost.

```
1       struct mcmf_edge {
2               int u, v, w, c;
3               mcmf_edge* rev;
4               mcmf_edge(int _u, int _v, int _w, int _c, mcmf_edge* _rev = NULL) {
5                       u = _u; v = _v; w = _w; c = _c; rev = _rev;
6               }
7       };
8
9       ii min_cost_max_flow(int n, int s, int t, vector<pair<int, ii> >* adj) {
10              vector<mcmf_edge*>* g = new vector<mcmf_edge*>[n];
11              for (int i = 0; i < n; i++) {
12                      for (int j = 0, len = size(adj[i]); j < len; j++) {
13                              mcmf_edge *cur = new mcmf_edge(i, adj[i][j].first,
                                        adj[i][j].second.first, adj[i][j].second.second),
14                                              *rev = new mcmf_edge(adj[i][j].first, i, 0,
                                                        -adj[i][j].second.second, cur);
15                              cur->rev = rev;
16                              g[i].push_back(cur);
17                              g[adj[i][j].first].push_back(rev);
18                      }
19              }
20              int flow = 0, cost = 0;
21              mcmf_edge** back = new mcmf_edge*[n];
22              int* dist = new int[n];
23              while (true) {
24                      for (int i = 0; i < n; i++) back[i] = NULL, dist[i] = INF;
25                      dist[s] = 0;
26                      for (int i = 0; i < n - 1; i++)
27                              for (int j = 0; j < n; j++)
28                                      if (dist[j] != INF)
29                                              for (int k = 0, len = size(g[j]); k < len;
                                                        k++)
30                                                      if (g[j][k]->w > 0 && dist[j] +
                                                                g[j][k]->c < dist[g[j][k]->v])
31                                                      {
32                                                              dist[g[j][k]->v] = dist[j]
                                                                        + g[j][k]->c;
33                                                              back[g[j][k]->v] = g[j][k];
34                                                      }
```

```
35                    mcmf_edge* cure = back[t];
36                    if (cure == NULL) break;
37                    int cap = INF;
38                    while (true) {
39                            cap = min(cap, cure->w);
40                            if (cure->u == s) break;
41                            cure = back[cure->u];
42                    }
43                    assert(cap > 0 && cap < INF);
44                    cure = back[t];
45                    while (true) {
46                            cost += cap * cure->c;
47                            cure->w -= cap;
48                            cure->rev->w += cap;
49                            if (cure->u == s) break;
50                            cure = back[cure->u];
51                    }
52                    flow += cap;
53            }
54        // instead of deleting g, we could also
55        // use it to get info about the actual flow
56        for (int i = 0; i < n; i++)
57                for (int j = 0, len = size(g[i]); j < len; j++)
58                        delete g[i][j];
59        delete[] g;
60        delete[] back;
61        delete[] dist;
62        return ii(flow, cost);
63 }
```

4   Strings

4.1   The Z algorithm

Given a string S, $Z_i(S)$ is the longest substring of S starting at i that is also a prefix of S. The Z algorithm computes these Z values in $O(n)$ time, where $n = |S|$. Z values can, for example, be used to find all occurrences of a pattern P in a string T in linear time. This is accomplished by computing Z values of $S = TP$, and looking for all i such that $Z_i \geq |T|$.

```
1 int* z_values(string s) {
2        int n = size(s);
3        int* z = new int[n];
4        int l = 0, r = 0;
5        z[0] = n;
6        for (int i = 1; i < n; i++) {
7                z[i] = 0;
8                if (i > r) {
9                        l = r = i;
10                       while (r < n && s[r - l] == s[r]) r++;
11                       z[i] = r - l; r--;
12               } else {
13                       if (z[i - l] < r - i + 1) {
14                               z[i] = z[i - l];
```

```
15                        } else {
16                                l = i;
17                                while (r < n && s[r - l] == s[r]) r++;
18                                z[i] = r - l; r--;
19                        }
20                }
21        }
22
23        return z;
24 }
```

5 Mathematics

5.1 Fraction

A fraction (rational number) class. Note that numbers are stored in lowest common terms.

```
1 template <class T>
2 class fraction {
3 private:
4        T gcd(T a, T b) {
5                return b == T(0) ? a : gcd(b, a % b);
6        }
7
8 public:
9        T n, d;
10
11        fraction(T n_, T d_) {
12                assert(d_ != 0);
13                n = n_;
14                d = d_;
15
16                if (d < T(0)) {
17                        n = -n;
18                        d = -d;
19                }
20
21                T g = gcd(n, d);
22                n /= g;
23                d /= g;
24        }
25
26        fraction(T n_) {
27                n = n_;
28                d = 1;
29        }
30
31        fraction(const fraction<T>& other) {
32                n = other.n;
33                d = other.d;
34        }
35
36        fraction<T> operator +(const fraction<T>& other) const {
```

```
37                    return fraction<T>(n * other.d + other.n * d, d * other.d);
38           }
39
40         fraction<T> operator -(const fraction<T>& other) const {
41                    return fraction<T>(n * other.d - other.n * d, d * other.d);
42           }
43
44         fraction<T> operator *(const fraction<T>& other) const {
45                    return fraction<T>(n * other.n, d * other.d);
46           }
47
48         fraction<T> operator /(const fraction<T>& other) const {
49                    return fraction<T>(n * other.d, d * other.n);
50           }
51
52         bool operator <(const fraction<T>& other) const {
53                    return n * other.d < other.n * d;
54           }
55
56         bool operator <=(const fraction<T>& other) const {
57                    return !(other < *this);
58           }
59
60         bool operator >(const fraction<T>& other) const {
61                    return other < *this;
62           }
63
64         bool operator >=(const fraction<T>& other) const {
65                    return !(*this < other);
66           }
67
68         bool operator ==(const fraction<T>& other) const {
69                    return n == other.n && d == other.d;
70           }
71
72         bool operator !=(const fraction<T>& other) const {
73                    return !(*this == other);
74           }
75  };
```

5.2   Binomial Coefficients

The binomial coefficient $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ is the number of ways to choose k items out of a total of n items.

```
1  int factorial(int n) {
2         int res = 1;
3         while (n) res *= n--;
4         return res;
5  }
6
7  int nck(int n, int k) {
8         return factorial(n) / factorial(k) / factorial(n - k);
9  }
```

```
1  void nck_precompute(int** arr, int n) {
2         for (int i = 0; i < n; i++)
3                 arr[i][0] = arr[i][i] = 1;
4
5         for (int i = 1; i < n; i++)
6                 for (int j = 1; j < i; j++)
7                         arr[i][j] = arr[i - 1][j - 1] + arr[i - 1][j];
8  }
```

```
1  int nck(int n, int k) {
2         if (n - k < k)
3                 k = n - k;
4
5         int res = 1;
6         for (int i = 1; i <= k; i++)
7                 res = res * (n - (k - i)) / i;
8
9         return res;
10 }
```

5.3   Euclidean algorithm

The Euclidean algorithm computes the greatest common divisor of two integers a, b.

```
1  int _gcd_(int a, int b) {
2         return b == 0 ? a : _gcd_(b, a % b);
3  }
4
5  int gcd(int a, int b) {
6         return (a < 0 && b < 0 ? -1 : 1) * _gcd_(abs(a), abs(b));
7  }
```

The extended Euclidean algorithm computes the greatest common divisor d of two integers a, b and also finds two integers x, y such that $a \times x + b \times y = d$.

```
1  int _egcd_(int a, int b, int& x, int& y) {
2         if (b == 0) {
3                 x = 1; y = 0;
4                 return a;
5         } else {
6                 int d = _egcd_(b, a % b, x, y);
7                 x -= a / b * y;
8                 swap(x, y);
9                 return d;
10        }
11 }
12
13 int egcd(int a, int b, int& x, int& y) {
14        int d = _egcd_(abs(a), abs(b), x, y);
15        if (a < 0) x = -x;
16        if (b < 0) y = -y;
17        return a < 0 && b < 0 ? -d : d;
18 }
```

5.4   Trial Division Primality Testing

An optimized trial division to check whether an integer is prime.

```cpp
bool is_prime(int n) {
        if (n < 2) return false;
        if (n < 4) return true;
        if (n % 2 == 0 || n % 3 == 0) return false;
        if (n < 25) return true;

        int s = static_cast<int>(sqrt(static_cast<double>(n)));
        for (int i = 5; i <= s; i += 6) {
                if (n % i == 0) return false;
                if (n % (i + 2) == 0) return false;
        }

        return true;
}
```

5.5   Sieve of Eratosthenes

An optimized implementation of Eratosthenes' Sieve.

```cpp
vector<int> prime_sieve(int n) {
        int mx = (n - 3) >> 1;
        vector<int> primes;
        bool* prime = new bool[mx + 1];
        memset(prime, 1, mx + 1);

        if (n >= 2)
                primes.push_back(2);

        int i = 0;
        for ( ; i <= mx; i++)
                if (prime[i]) {
                        int v = (i << 1) + 3;
                        primes.push_back(v);
                        int sq = i * ((i << 1) + 6) + 3;
                        if (sq > mx) break;
                        for (int j = sq; j <= mx; j += v)
                                prime[j] = false;
                }

        for (i++; i <= mx; i++)
                if (prime[i])
                        primes.push_back((i << 1) + 3);

        delete[] prime; // can be used for O(1) lookup
        return primes;
}
```

5.6   Modular Multiplicative Inverse

```
1  #include "egcd.cpp"
2
3  int mod_inv(int a, int m) {
4          assert(m > 1);
5          int x, y, d = egcd(a, m, x, y);
6          if (d != 1) return -1;
7          return x < 0 ? x + m : x;
8  }
```

5.7   Modular Exponentiation

```
1  int mod_pow(int b, int e, int m) {
2          assert(e >= 0);
3          assert(m > 0);
4          int res = 1;
5          while (e) {
6                  if (e & 1) res = mod(res * b, m);
7                  b = mod(b * b, m);
8                  e >>= 1;
9          }
10
11         return res;
12 }
```

5.8   Chinese Remainder Theorem

```
1  #include "gcd.cpp"
2  #include "egcd.cpp"
3
4  int crt(const vi& as, const vi& ns) {
5          assert(size(as) == size(ns));
6          int cnt = size(as), N = 1, x = 0, r, s, l;
7          for (int i = 0; i < cnt; i++) {
8                  N *= ns[i];
9                  assert(ns[i] > 0);
10                 for (int j = 0; j < cnt; j++)
11                         if (i != j)
12                                 assert(gcd(ns[i], ns[j]) == 1);
13         }
14
15         for (int i = 0; i < cnt; i++) {
16                 egcd(ns[i], l = N/ns[i], r, s);
17                 x += as[i] * s * l;
18         }
19
20         return mod(x, N);
21 }
```

5.9   Formulas

5.9.1   Combinatorics

- Number of ways to choose k objects from a total of n objects where order matters and each item can only be chosen once: $P_k^n = \frac{n!}{(n-k)!}$

- Number of ways to choose k objects from a total of n objects where order matters and each item can be chosen multiple times: n^k

- Number of permutations of n objects, where there are n_1 objects of type 1, n_2 objects of type 2, $\ldots$, n_k objects of type k: $\binom{n}{n_1,n_2,\ldots,n_k} = \frac{n!}{n_1! \times n_2! \times \cdots \times n_k!}$

- Number of ways to choose k objects from a total of n objects where order does not matter and each item can only be chosen once:
$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} = \binom{n}{n-k} = \prod_{i=1}^{k} \frac{n-(k-i)}{i} = \frac{n!}{k!(n-k)!}, \binom{n}{0} = 1, \binom{0}{k} = 0$

- Number of ways to choose k objects from a total of n objects where order does not matter and each item can be chosen multiple times: $f_k^n = \binom{n+k-1}{k} = \frac{(n+k-1)!}{k!(n-1)!}$

- Number of integer solutions to $x_1 + x_2 + \cdots + x_n = k$ where $x_i \geq 0$: f_k^n

- Number of subsets of a set with n elements: 2^n

- $|A \cup B| = |A| + |B| - |A \cap B|$

- $|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$

- Number of ways to walk from the lower-left corner to the upper-right corner of an $n \times m$ grid by walking only up and to the right: $\binom{n+m}{m}$

- Number of strings with n sets of brackets such that the brackets are balanced:
$C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k} = \frac{1}{n+1}\binom{2n}{n}$

- Number of triangulations of a convex polygon with n points, number of rooted binary trees with $n+1$ vertices, number of paths across an $n \times n$ lattice which do not rise above the main diagonal: C_n

- Number of permutations of n objects with exactly k ascending sequences or *runs*:
$\left\langle {n \atop k} \right\rangle = \left\langle {n \atop n-k-1} \right\rangle = k\left\langle {n-1 \atop k} \right\rangle + (n-k+1)\left\langle {n-1 \atop k-1} \right\rangle = \sum_{i=0}^{k}(-1)^i\binom{n+1}{i}(k+1-i)^n, \left\langle {n \atop 0} \right\rangle = \left\langle {n \atop n-1} \right\rangle = 1$

- Number of permutations of n objects with exactly k cycles: $\left[{n \atop k} \right] = \left[{n-1 \atop k-1} \right] + (n-1)\left[{n-1 \atop k} \right]$

- Number of ways to partition n objects into k sets: $\left\{ {n \atop k} \right\} = k\left\{ {n-1 \atop k} \right\} + \left\{ {n-1 \atop k-1} \right\}, \left\{ {n \atop 0} \right\} = \left\{ {n \atop n} \right\} = 1$

6   Geometry

6.0.2   Primitives

```
1  struct vector2d;
2  struct point {
3          double x, y;
4          point(double _x = 0, double _y = 0) { x = _x; y = _y; }
5          bool operator <(point other) {
6                  if (abs(x - other.x) > EPS) return x < other.x;
7                  if (abs(y - other.y) > EPS) return y < other.y;
8                  return false;
9          }
10         bool operator ==(point other) { return abs(x - other.x) < EPS && abs(y -
                   other.y) < EPS; }
11         vector2d vectorTo(point other);
12         vector2d toVector();
```

```
13  };
14  struct vector2d {
15          double x, y;
16          vector2d(double _x = 0, double _y = 0) { x = _x; y = _y; }
17          bool operator <(vector2d other) {
18                  if (abs(x - other.x) > EPS) return x < other.x;
19                  if (abs(y - other.y) > EPS) return y < other.y;
20                  return false;
21          }
22          bool operator ==(vector2d other) { return abs(x - other.x) < EPS && abs(y -
                  other.y) < EPS; }
23          vector2d operator +(vector2d other) { return vector2d(x + other.x, y +
                  other.y); }
24          vector2d operator -(vector2d other) { return vector2d(x - other.x, y -
                  other.y); }
25          vector2d operator *(double c) { return vector2d(x * c, y * c); }
26          double dot(vector2d other) { return x * other.x + y * other.y; }
27          double cross(vector2d other) { return x * other.y - y * other.x; }
28          double magnitudeSquared() { return dot(*this); }
29          double magnitude() { return sqrt(magnitudeSquared()); }
30          double angleTo(vector2d other) { return acos(dot(other) / magnitude() /
                  other.magnitude()); }
31          double signedAngleTo(vector2d other) { return asin(cross(other) /
                  magnitude() / other.magnitude()); }
32          vector2d rotate(double theta) { return vector2d(x * cos(theta) - y *
                  sin(theta), x * sin(theta) + y * cos(theta)); }
33  };
34  vector2d point::vectorTo(point other) { return vector2d(other.x - x, other.y - y); }
35  vector2d point::toVector() { return vector2d(x, y); }
36  double ccw(point a, point b, point c) { return a.vectorTo(b).cross(a.vectorTo(c)); }
37  bool collinearPoints(point a, point b, point c) {
38          return abs(a.vectorTo(b).signedAngleTo(a.vectorTo(c))) < EPS;
39  }
40  point closestPointOnLine(point a, point b, point c, bool is_segment = false) {
41          if (is_segment) {
42                  if (a.vectorTo(b).dot(b.vectorTo(c)) > 0) return b;
43                  if (b.vectorTo(a).dot(a.vectorTo(c)) > 0) return a;
44          }
45          vector2d ac = a.vectorTo(c), ab = a.vectorTo(b);
46          double t = ac.dot(ab) / ab.magnitudeSquared();
47          return point(a.x + ab.x * t, a.y + ab.y * t);
48  }
49  struct line {
50          // ax + by + c = 0
51          double a, b, c;
52
53          line(point p1, point p2) {
54                  if (abs(p1.x - p2.x) < EPS) { a = 1; b = 0; c = -p1.y; }
55                  else {
56                          a = -(p1.y - p2.y) / (p1.x - p2.y);
57                          b = 1;
58                          c = -a * p1.x - b * p1.y;
59                  }
```

```
60              }
61
62              bool parallelTo(line other) { return abs(a - other.a) < EPS && abs(b -
                    other.b) < EPS; }
63              bool operator ==(line other) { return parallelTo(other) && abs(c - other.c)
                    < EPS; }
64          point intersection(line other) {
65                  assert(!parallelTo(other));
66                  double x = (other.b * c - b * other.c) / (other.a * b - a *
                        other.b), y;
67                  if (abs(b) > EPS) y = -(a * x + c) / b;
68                  else y = -(other.a * x + other.c) / other.b;
69                  return point(x, y);
70          }
71  };
72  double polygon_area_signed(vector<point>& points) {
73          double area = 0; int cnt = size(points);
74          for (int i = 1; i + 1 < cnt; i++) {
75                  area +=
                        points[0].vectorTo(points[i]).cross(points[0].vectorTo(points[i
                        + 1]));
76          }
77          return area / 2;
78  }
79  double polygon_area(vector<point>& points) {
80          return abs(polygon_area_signed(points));
81  }
```

6.0.3   Convex Hull

```
 1  #include "primitives.cpp"
 2
 3  point ch_main;
 4  bool ch_cmp(const point& a, const point& b) {
 5          if (collinearPoints(ch_main, a, b))
 6                  return ch_main.vectorTo(a).magnitude() <
                        ch_main.vectorTo(b).magnitude();
 7          return atan2(a.y - ch_main.y, a.x - ch_main.x) < atan2(b.y - ch_main.y, b.x
                - ch_main.x);
 8  }
 9
10  vector<point> convex_hull(vector<point> pts, bool add_collinear = false) {
11          int cnt = size(pts), main = 0, i = 1;
12          if (cnt <= 3) return pts;
13          for (int i = 1; i < cnt; i++)
14                  if (pts[i].y < pts[main].y || (abs(pts[i].y - pts[main].y) < EPS &&
                        pts[i].y > pts[main].y))
15                          main = i;
16          swap(pts[0], pts[main]);
17          ch_main = pts[0];
18          sort(++pts.begin(), pts.end(), ch_cmp);
19          point prev, now;
20          stack<point> S; S.push(pts[cnt - 1]); S.push(pts[0]);
```

```
21          while (i < cnt) {
22                  now = S.top(); S.pop();
23                  if (S.empty()) {
24                          S.push(now);
25                          S.push(pts[i++]);
26                  } else {
27                          prev = S.top();
28                          S.push(now);
29                          if (ccw(prev, now, pts[i]) > 0 || (add_collinear &&
                                abs(ccw(prev, now, pts[i])) < EPS)) S.push(pts[i++]);
30                          else S.pop();
31                  }
32          }
33          vector<point> res;
34          while (!S.empty()) res.push_back(S.top()), S.pop();
35          return res;
36 }
```

6.0.4 Formulas

Let $a = (a_x, a_y)$ and $b = (b_x, b_y)$ be two-dimensional vectors.

- $a \cdot b = |a||b| \cos \theta$, where θ is the angle between a and b.

- $a \times b = |a||b| \sin \theta$, where θ is the signed angle between a and b.

- $a \times b$ is equal to the area of the parallelogram with two of its sides formed by a and b. Half of that is the area of the triangle formed by a and b.

7 Other Algorithms

7.1 Binary Search

An implementation of binary search that finds a real valued root of the continous function f on the interval $[a, b]$, with a maximum error of ε.

```
1 double binary_search_continuous(double low, double high, double eps, double
      (*f)(double)) {
2          while (true) {
3                  double mid = (low + high) / 2, cur = f(mid);
4                  if (abs(cur) < eps) return mid;
5                  else if (0 < cur) high = mid;
6                  else low = mid;
7          }
8 }
```

Another implementation that takes a binary predicate f, and finds an integer value x on the integer interval $[a, b]$ such that $f(x) \land \neg f(x - 1)$.

```
1 int binary_search_discrete(int low, int high, bool (*f)(int)) {
2          assert(low <= high);
3          while (low < high) {
4                  int mid = low + (high - low) / 2;
5                  if (f(mid)) high = mid;
6                  else low = mid + 1;
7          }
```

```
8            assert(f(low));
9            return low;
10  }
```

7.2   2SAT

```
1   /* TODO: implement linear version using Tarjan's SCC algorithm */
2   bool two_sat(int n, const vii& clauses, vi& all_truthy) {
3           all_truthy.clear();
4           vi* adj = new vi[2 * n + 1];
5           for (int i = 0, cnt = size(clauses); i < cnt; i++) {
6                   adj[-clauses[i].first + n].push_back(clauses[i].second);
7                   if (clauses[i].first != clauses[i].second)
8                           adj[-clauses[i].second + n].push_back(clauses[i].first);
9           }
10
11          char* ass = new char[2 * n + 1];
12          bool* negpath = new bool[2 * n + 1];
13          for (int i = -n; i <= n; i++) {
14                  if (i == 0) continue;
15                  memset(ass, 0, 2 * n + 1);
16                  negpath[i + n] = false;
17                  ass[i + n] = true;
18                  queue<int> Q;
19                  Q.push(i);
20                  while (!Q.empty()) {
21                          int cur = Q.front(); Q.pop();
22                          for (int j = 0, cnt = size(adj[cur + n]); j < cnt; j++) {
23                                  int nxt = adj[cur + n][j];
24                                  if (nxt == -i) {
25                                          negpath[i + n] = true;
26                                          break;
27                                  }
28                                  if (!ass[nxt + n]) {
29                                          ass[nxt + n] = true;
30                                          Q.push(nxt);
31                                  }
32                          }
33                  }
34          }
35
36          for (int i = 1; i <= n; i++)
37                  if (negpath[i + n] && negpath[-i + n]) {
38                          delete[] ass;
39                          delete[] negpath;
40                          delete[] adj;
41                          return false;
42                  }
43
44          memset(ass, 0, 2 * n + 1);
45          for (int i = -n; i <= n; i++) {
46                  if (i == 0 || ass[i + n] != 0 || negpath[i + n]) continue;
```

```
47                    ass[i + n] = 1;
48                    assert(ass[-i + n] != 1);
49                    ass[-i + n] = 2;
50                    queue<int> Q;
51                    Q.push(i);
52                    while (!Q.empty()) {
53                            int cur = Q.front(); Q.pop();
54                            for (int j = 0, cnt2 = size(adj[cur + n]); j < cnt2; j++) {
55                                    int nxt = adj[cur + n][j];
56                                    if (ass[nxt + n] != 0) { assert(ass[nxt + n] == 1);
                                            continue; }
57                                    ass[nxt + n] = 1;
58                                    assert(ass[-nxt + n] != 1);
59                                    ass[-nxt + n] = 2;
60                                    Q.push(nxt);
61                            }
62                    }
63            }
64
65            for (int i = -n; i <= n; i++) {
66                    if (i == 0 || ass[i + n] != 1) continue;
67                    all_truthy.push_back(i);
68            }
69
70            delete[] ass;
71            delete[] negpath;
72            delete[] adj;
73            return true;
74  }
```

7.3 nth Permutation

A very fast algorithm for computing the nth permutation of the list $\{0, 1, \ldots, k-1\}$.

```
1  vector<int> nth_permutation(int cnt, int n) {
2          vector<int> idx(cnt), per(cnt), fac(cnt);
3          for (int i = 0; i < cnt; i++) idx[i] = i;
4          for (int i = 1; i <= cnt; i++) fac[i - 1] = n % i, n /= i;
5          for (int i = cnt - 1; i >= 0; i--) per[cnt - i - 1] = idx[fac[i]],
                  idx.erase(idx.begin() + fac[i]);
6          return per;
7  }
```

7.4   Cycle-Finding

7.4.1   Floyd's Cycle-Finding algorithm

```
1  ii find_cycle(int x0, int (*f)(int)) {
2          int t = f(x0), h = f(t), mu = 0, lam = 1;
3          while (t != h) t = f(t), h = f(f(h));
4          h = x0;
5          while (t != h) t = f(t), h = f(h), mu++;
6          h = f(t);
7          while (t != h) h = f(h), lam++;
```

```
8            return ii(mu, lam);
9    }
```

8 Useful Information

8.1 Tips & Tricks

- How fast does our algorithm have to be? Can we use brute-force?

- Does order matter?

- Is it better to look at the problem in another way? Maybe backwards?

- Are there subproblems that are recomputed? Can we cache them?

- Do we need to remember everything we compute, or just the last few iterations of computation?

- Does it help to sort the data?

- Can we speed up lookup by using a map (tree or hash) or an array?

- Can we binary search the answer?

- Can we add vertices/edges to the graph to make the problem easier? Can we turn the graph into some other kind of a graph (perhaps a DAG, or a flow network)?

- Make sure integers are not overflowing.

- Is it better to compute the answer modulo n? Perhaps we can compute the answer modulo $m_1, m_2, \ldots, m_k$, where $m_1, m_2, \ldots, m_k$ are pairwise coprime integers, and find the real answer using CRT?

- Are there any edge cases? When $n = 0, n = -1, n = 1, n = 2^{31} - 1$ or $n = -2^{31}$? When the list is empty, or contains a single element? When the graph is empty, or contains a single vertex? When the graph contains self-loops? When the polygon is concave or non-simple?

- Can we use exponentiation by squaring?

8.2 Worst Time Complexity

n	Worst AC Algorithm	Comment
≤ 10	$O(n!), O(n^6)$	e.g. Enumerating a permutation
≤ 15	$O(2^n \times n^2)$	e.g. DP TSP
≤ 20	$O(2^n), O(n^5)$	e.g. DP + bitmask technique
≤ 50	$O(n^4)$	e.g. DP with 3 dimensions + $O(n)$ loop, choosing $_nC_k = 4$
$\leq 10^2$	$O(n^3)$	e.g. Floyd Warshall's
$\leq 10^3$	$O(n^2)$	e.g. Bubble/Selection/Insertion sort
$\leq 10^5$	$O(n \log_2 n)$	e.g. Merge sort, building a Segment tree
$\leq 10^6$	$O(n), O(\log_2 n), O(1)$	Usually, contest problems have $n \leq 10^6$ (e.g. to read input)

8.3 Bit Hacks

- `n & -n` returns the first set bit in n.

- `n & (n - 1)` is 0 only if n is a power of two.

- `snoob(x)` returns the next integer that has the same amount of bits set as `x`. Useful for iterating through subsets of some specified size.

```
1  int snoob(int x) {
2          int y = x & -x, z = x + y;
3          return z | ((x ^ z) >> 2) / y;
4  }
```