Algorithms and Data Structures for Competitive Programming

July 9, 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

```
struct union_find {
    vi parent;
    int cnt;
    union_find(int n) {
        parent.resize(cnt = n);
        for (int i = 0; i < cnt; i++) parent[i] = i;
    }
    int find(int i) { return parent[i] == i ? i : (parent[i] = find(parent[i])); }
    bool unite(int i, int j) {
        int ip = find(i), jp = find(j);
        parent[ip] = jp;
        return ip != jp;
    }
};
```

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.

```
struct fenwick_tree {
    vi dataMul;
    vi dataAdd;

    fenwick_tree(int n) {
        dataMul.resize(n);
        dataAdd.resize(n);
        fill(all(dataMul), 0);
        fill(all(dataAdd), 0);
    }

    void adjust(int i, int v) {
        adjust(i, i, v);
    }

    inline int rsq(int i, int j) {
        return rsq(j) - rsq(i - 1);
    }

    inline int get(int i) {
        return rsq(i) - rsq(i - 1);
    }

    void adjust(int left, int right, int by) {
        internalUpdate(left, by, -by * (left - 1));
        internalUpdate(right, -by, by * right);
    }

    void internalUpdate(int at, int mul, int add) {
```

```
        while (at < size(dataMul)) {
            dataMul[at] += mul;
            dataAdd[at] += add;
            at |= (at + 1);
        }
    }

    int rsq(int at) {
        int mul = 0, add = 0, start = at;
        while (at >= 0) {
            mul += dataMul[at];
            add += dataAdd[at];
            at = (at & (at + 1)) - 1;
        }
        return mul * start + add;
    }
};
```

2.3 Matrix

```
template <class T>
class matrix {
private:
    vector<T> data;
    int cnt;
    inline T& at(int i, int j) {
        return data[i * cols + j];
    }

public:
    int rows, cols;

    matrix(int r, int c) {
        rows = r; cols = c;
        data = vector<T>(cnt = rows * cols);
        for (int i = 0; i < cnt; i++)
            data[i] = T(0);
    }

    matrix(const matrix& other) {
        rows = other.rows; cols = other.cols;
        data = vector<T>(cnt = rows * cols);
        for (int i = 0; i < cnt; i++)
            data[i] = other.data[i];
    }

    T& operator()(int i, int j) {
        return at(i, j);
    }

    void operator +=(const matrix& other) {
        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;
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  };
```

2.5 AVL Tree

A fast, easily augmentable, balanced binary search tree.

```
1   #define AVL_MULTISET 0
2
3   template <class T>
4   class avl_tree {
5   public:
6
7       struct node {
8           T item;
9           node *p, *l, *r;
10          int size, height;
11          node(const T &item, node *p = NULL) : item(item), p(p), l(NULL), r(NULL),
                size(1), height(0) { }
12      };
13
14      avl_tree() : root(NULL) { }
15
16      node *root;
17
18      node* find(const T &item) const {
19          node *cur = root;
20          while (cur) {
21              if (cur->item < item) cur = cur->r;
22              else if (item < cur->item) cur = cur->l;
23              else break;
24          }
25
26          return cur;
27      }
28
```

```
29      node* insert(const T &item) {
30          node *prev = NULL, **cur = &root;
31          while (*cur) {
32              prev = *cur;
33              if ((*cur)->item < item) cur = &((*cur)->r);
34  #if AVL_MULTISET
35              else cur = &((*cur)->l);
36  #else
37              else if (item < (*cur)->item) cur = &((*cur)->l);
38              else return *cur;
39  #endif
40          }
41
42          node *n = new node(item, prev);
43          *cur = n;
44          fix(n);
45          return n;
46      }
47
48      void erase(const T &item) { erase(find(item)); }
49      void erase(node *n, bool free = true) {
50          if (!n) return;
51          if (!n->l && n->r) {
52              parent_leg(n) = n->r;
53              n->r->p = n->p;
54          } else if (n->l && !n->r) {
55              parent_leg(n) = n->l;
56              n->l->p = n->p;
57          } else if (n->l && n->r) {
58              node *s = successor(n);
59              erase(s, false);
60              s->p = n->p;
61              s->l = n->l;
62              s->r = n->r;
63              if (n->l) n->l->p = s;
64              if (n->r) n->r->p = s;
65              parent_leg(n) = s;
66              fix(s);
67              return;
68          } else parent_leg(n) = NULL;
69          fix(n->p);
70          n->p = n->l = n->r = NULL;
71          if (free) delete n;
72      }
73
74      node* successor(node *n) const {
75          if (!n) return NULL;
76          if (n->r) return nth(0, n->r);
77          node *p = n->p;
78          while (p && p->r == n) n = p, p = p->p;
79          return p;
80      }
81
```

```
82      node* predecessor(node *n) const {
83          if (!n) return NULL;
84          if (n->l) return nth(n->l->size-1, n->l);
85          node *p = n->p;
86          while (p && p->l == n) n = p, p = p->p;
87          return p;
88      }
89
90      inline int size() const { return sz(root); }
91
92      void clear() {
93          delete_tree(root);
94          root = NULL;
95      }
96
97      node* nth(int n, node *cur = NULL) const {
98          if (!cur) cur = root;
99          while (cur) {
100             if (n < sz(cur->l)) cur = cur->l;
101             else if (n > sz(cur->l)) n -= sz(cur->l) + 1, cur = cur->r;
102             else break;
103         }
104
105         return cur;
106     }
107
108 private:
109
110     inline int sz(node *n) const { return n ? n->size : 0; }
111     inline int height(node *n) const { return n ? n->height : -1; }
112     inline bool left_heavy(node *n) const { return n && height(n->l) >
            height(n->r); }
113     inline bool right_heavy(node *n) const { return n && height(n->r) >
            height(n->l); }
114     inline bool too_heavy(node *n) const { return n && abs(height(n->l) -
            height(n->r)) > 1; }
115
116     void delete_tree(node *n) {
117         if (!n) return;
118         delete_tree(n->l);
119         delete_tree(n->r);
120         delete n;
121     }
122
123     node*& parent_leg(node *n) {
124         if (!n->p) return root;
125         if (n->p->l == n) return n->p->l;
126         if (n->p->r == n) return n->p->r;
127         assert(false);
128     }
129
130     void augment(node *n) {
131         if (!n) return;
```

```
132         n->size = 1 + sz(n->l) + sz(n->r);
133         n->height = 1 + max(height(n->l), height(n->r));
134     }
135
136     void left_rotate(node *n) {
137         // assert(n->r);
138         node *r = n->r;
139         r->p = n->p;
140         parent_leg(n) = r;
141         n->r = r->l;
142         if (r->l) r->l->p = n;
143         r->l = n;
144         n->p = r;
145         augment(n);
146         augment(r);
147     }
148
149     void right_rotate(node *n) {
150         // assert(n->l);
151         node *l = n->l;
152         l->p = n->p;
153         parent_leg(n) = l;
154         n->l = l->r;
155         if (l->r) l->r->p = n;
156         l->r = n;
157         n->p = l;
158         augment(n);
159         augment(l);
160     }
161
162     void fix(node *n) {
163         while (n) {
164             augment(n);
165             if (too_heavy(n)) {
166                 if (left_heavy(n) && right_heavy(n->l)) left_rotate(n->l);
167                 else if (right_heavy(n) && left_heavy(n->r)) right_rotate(n->r);
168                 if (left_heavy(n)) right_rotate(n);
169                 else left_rotate(n);
170                 n = n->p;
171             }
172
173             n = n->p;
174         }
175     }
176 };
```

Also a very simple wrapper over the AVL tree that implements a map interface.

```
1 #include "avl_tree.cpp"
2
3 template <class K, class V>
4 class avl_map {
5 public:
6     struct node {
```

```
        K key;
        V value;
        node(K k, V v) : key(k), value(v) { }

        bool operator <(const node &other) const {
            return key < other.key;
        }
    };

    avl_tree<node> tree;

    V& operator [](K key) {
        typename avl_tree<node>::node *n = tree.find(node(key, V(0)));
        if (!n) n = tree.insert(node(key, V(0)));
        return n->item.value;
    }
};
```

2.6 Heap

An implementation of a binary heap.

```
#define RESIZE
#define SWP(x,y) tmp = x, x = y, y = tmp

struct default_int_cmp {
    default_int_cmp() { }
    bool operator ()(const int &a, const int &b) {
        return a < b;
    }
};

template <class Compare = default_int_cmp>
class heap {
private:
    int len, count, *q, *loc, tmp;
    Compare _cmp;

    inline bool cmp(int i, int j) { return _cmp(q[i], q[j]); }
    inline void swp(int i, int j) { SWP(q[i], q[j]), SWP(loc[q[i]], loc[q[j]]); }

    void swim(int i) {
        while (i > 0) {
            int p = (i - 1) / 2;
            if (!cmp(i, p)) break;
            swp(i, p), i = p;
        }
    }

    void sink(int i) {
        while (true) {
            int l = 2*i + 1, r = l + 1;
            if (l >= count) break;
```

```
32              int m = r >= count || cmp(l, r) ? l : r;
33              if (!cmp(m, i)) break;
34              swp(m, i), i = m;
35          }
36      }
37
38  public:
39      heap(int init_len = 128) : count(0), len(init_len), _cmp(Compare()) {
40          q = new int[len];
41          loc = new int[len];
42          memset(loc, 255, len << 2);
43      }
44
45      ~heap() {
46          delete[] q;
47          delete[] loc;
48      }
49
50      void push(int n, bool fix = true) {
51          if (len == count || n >= len) {
52  #ifdef RESIZE
53              int newlen = 2 * len;
54              while (n >= newlen) newlen *= 2;
55              int *newq = new int[newlen],
56                  *newloc = new int[newlen];
57
58              for (int i = 0; i < len; i++) {
59                  newq[i] = q[i];
60                  newloc[i] = loc[i];
61              }
62
63              memset(newloc + len, 255, (newlen - len) << 2);
64              delete[] q;
65              delete[] loc;
66              loc = newloc, q = newq, len = newlen;
67  #else
68              assert(false);
69  #endif
70          }
71
72          assert(loc[n] == -1);
73          loc[n] = count, q[count++] = n;
74          if (fix) swim(count-1);
75      }
76
77      void pop(bool fix = true) {
78          assert(count > 0);
79          loc[q[0]] = -1;
80          q[0] = q[--count];
81          loc[q[0]] = 0;
82          if (fix) sink(0);
83      }
84
```

```
85      int top() { assert(count > 0); return q[0]; }
86      void heapify() { for (int i = count - 1; i > 0; i--) if (cmp(i, (i - 1) / 2))
            swp(i, (i - 1) / 2); }
87      void update_key(int n) { assert(loc[n] != -1), swim(loc[n]), sink(loc[n]); }
88      bool empty() { return count == 0; }
89      int size() { return count; }
90      void clear() { count = 0, memset(loc, 255, len << 2); }
91  };
```

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. It runs in $\Theta(|E| \log |V|)$ time.

```
1   int *dist, *dad;
2   struct cmp {
3       bool operator()(int a, int b) { return dist[a] != dist[b] ? dist[a] < dist[b] :
            a < b; }
4   };
5
6   pair<int*, int*> dijkstra(int n, int s, vector<ii> *adj) {
7       dist = new int[n];
8       dad = new int[n];
9       for (int i = 0; i < n; i++) dist[i] = INF, dad[i] = -1;
10      set<int, cmp> pq;
11      dist[s] = 0, pq.insert(s);
12      while (!pq.empty()) {
13          int cur = *pq.begin(); pq.erase(pq.begin());
14          for (int i = 0, cnt = size(adj[cur]); i < cnt; i++) {
15              int nxt = adj[cur][i].first, ndist = dist[cur] + adj[cur][i].second;
16              if (ndist < dist[nxt]) dist[nxt] = ndist, dad[nxt] = cur,
                    pq.insert(nxt);
17          }
18      }
19
20      return pair<int*, int*>(dist, dad);
21  }
```

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++) dist[i] = i == s ? 0 : INF;
5       for (int i = 0; i < n - 1; i++)
6           for (int j = 0; j < n; j++)
7               if (dist[j] != INF)
8                   for (int k = 0, len = size(adj[j]); k < len; k++)
9                       dist[adj[j][k].first] = min(dist[adj[j][k].first], dist[j] +
                            adj[j][k].second);
10      for (int j = 0; j < n; j++)
11          for (int k = 0, len = size(adj[j]); k < len; k++)
12              if (dist[j] + adj[j][k].second < dist[adj[j][k].first])
```

```
13                 has_negative_cycle = true;
14     return dist;
15 }
```

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] + arr[k][j]);
7 }
```

3.4 Connected Components

3.5 Strongly Connected Components

3.5.1 Kosaraju's algorithm

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

```
1  #include "../data-structures/union_find.cpp"
2
3  vector<bool> visited;
4  vi order;
5
6  void scc_dfs(const vvi &adj, int u) {
7      int v; visited[u] = true;
8      for (int i = 0; i < size(adj[u]); i++) if (!visited[v = adj[u][i]])
           scc_dfs(adj, v);
9      order.push_back(u);
10 }
11
12 pair<union_find, vi> scc(const vvi &adj) {
13     int n = size(adj), u, v, w;
14     order.clear();
15     union_find uf(n);
16     vi dag;
17     vvi rev(n);
18     for (int i = 0; i < n; i++) for (int j = 0; j < size(adj[i]); j++)
           rev[adj[i][j]].push_back(i);
19     visited.resize(n), fill(all(visited), false);
20     for (int i = 0; i < n; i++) if (!visited[i]) scc_dfs(rev, i);
21     fill(all(visited), false);
22     stack<int> S;
23     for (int i = n-1; i >= 0; i--) {
24         if (visited[order[i]]) continue;
25         S.push(order[i]), dag.push_back(order[i]);
26         while (!S.empty()) {
```

```
27              visited[u = S.top()] = true, S.pop(), uf.unite(u, order[i]);
28              for (int i = 0; i < size(adj[u]); i++) if (!visited[v = adj[u][i]])
                    S.push(v);
29          }
30      }
31      return pair<union_find, vi>(uf, dag);
32  }
```

3.6 Minimum Spanning Tree

3.6.1 Kruskal's algorithm

```
1   #include "../data-structures/union_find.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];
```

```
    memset(color, 0, n);
    for (int i = 0; i < n; i++) {
        if (!color[i]) {
            tsort_dfs(i, color, adj, S, has_cycle);
            if (has_cycle) return res;
        }
    }
    while (!S.empty()) res.push_back(S.top()), S.pop();
    return res;
}
```

3.8 Bipartite Matching

The alternating paths algorithm solves bipartite matching in $O(mn^2)$ time, where m, n are the number of vertices on the left and right side of the bipartite graph, respectively.

```
vi* adj;
bool* done;
int* owner;

int alternating_path(int left) {
    if (done[left]) return 0;
    done[left] = true;
    for (int i = 0; i < size(adj[left]); i++) {
        int right = adj[left][i];
        if (owner[right] == -1 || alternating_path(owner[right])) {
            owner[right] = left;
            return 1;
        }
    }

    return 0;
}
```

3.9 Maximum Flow

3.9.1 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.

```
struct mf_edge {
    int u, v, w;
    mf_edge* rev;
    mf_edge(int _u, int _v, int _w, mf_edge* _rev = NULL) {
        u = _u; v = _v; w = _w; rev = _rev;
    }
};

int max_flow(int n, int s, int t, vii* adj) {
    vector<mf_edge*>* g = new vector<mf_edge*>[n];
    for (int i = 0; i < n; i++) {
        for (int j = 0, len = size(adj[i]); j < len; j++) {
            mf_edge *cur = new mf_edge(i, adj[i][j].first, adj[i][j].second),
                    *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  }
```

3.10 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.

```
struct mcmf_edge {
    int u, v, w, c;
    mcmf_edge* rev;
    mcmf_edge(int _u, int _v, int _w, int _c, mcmf_edge* _rev = NULL) {
        u = _u; v = _v; w = _w; c = _c; rev = _rev;
    }
};

ii min_cost_max_flow(int n, int s, int t, vector<pair<int, ii> >* adj) {
    vector<mcmf_edge*>* g = new vector<mcmf_edge*>[n];
    for (int i = 0; i < n; i++) {
        for (int j = 0, len = size(adj[i]); j < len; j++) {
            mcmf_edge *cur = new mcmf_edge(i, adj[i][j].first,
                adj[i][j].second.first, adj[i][j].second.second),
                      *rev = new mcmf_edge(adj[i][j].first, i, 0,
                          -adj[i][j].second.second, cur);
            cur->rev = rev;
            g[i].push_back(cur);
            g[adj[i][j].first].push_back(rev);
        }
    }
    int flow = 0, cost = 0;
    mcmf_edge** back = new mcmf_edge*[n];
    int* dist = new int[n];
    while (true) {
        for (int i = 0; i < n; i++) back[i] = NULL, dist[i] = INF;
        dist[s] = 0;
        for (int i = 0; i < n - 1; i++)
            for (int j = 0; j < n; j++)
                if (dist[j] != INF)
                    for (int k = 0, len = size(g[j]); k < len; k++)
                        if (g[j][k]->w > 0 && dist[j] + g[j][k]->c <
                            dist[g[j][k]->v]) {
                            dist[g[j][k]->v] = dist[j] + g[j][k]->c;
                            back[g[j][k]->v] = g[j][k];
                        }
        mcmf_edge* cure = back[t];
        if (cure == NULL) break;
        int cap = INF;
        while (true) {
            cap = min(cap, cure->w);
            if (cure->u == s) break;
            cure = back[cure->u];
        }
        assert(cap > 0 && cap < INF);
        cure = back[t];
        while (true) {
            cost += cap * cure->c;
            cure->w -= cap;
            cure->rev->w += cap;
            if (cure->u == s) break;
```

```
            cure = back[cure->u];
        }
        flow += cap;
    }
    // instead of deleting g, we could also
    // use it to get info about the actual flow
    for (int i = 0; i < n; i++)
        for (int j = 0, len = size(g[i]); j < len; j++)
            delete g[i][j];
    delete[] g;
    delete[] back;
    delete[] dist;
    return ii(flow, cost);
}
```

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 $Z_i \geq |T|$.

```
int* z_values(string s) {
    int n = size(s);
    int* z = new int[n];
    int l = 0, r = 0;
    z[0] = n;
    for (int i = 1; i < n; i++) {
        z[i] = 0;
        if (i > r) {
            l = r = i;
            while (r < n && s[r - l] == s[r]) r++;
            z[i] = r - l; r--;
        } else {
            if (z[i - l] < r - i + 1) z[i] = z[i - l];
            else {
                l = i;
                while (r < n && s[r - l] == s[r]) r++;
                z[i] = r - l; r--;
            }
        }
    }

    return z;
}
```

5 Mathematics

5.1 Fraction

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

```
template <class T>
class fraction {
private:
    T gcd(T a, T b) {
        return b == T(0) ? a : gcd(b, a % b);
    }

public:
    T n, d;

    fraction(T n_, T d_) {
        assert(d_ != 0);
        n = n_;
        d = d_;

        if (d < T(0)) {
            n = -n;
            d = -d;
        }

        T g = gcd(n, d);
        n /= g;
        d /= g;
    }

    fraction(T n_) {
        n = n_;
        d = 1;
    }

    fraction(const fraction<T>& other) {
        n = other.n;
        d = other.d;
    }

    fraction<T> operator +(const fraction<T>& other) const {
        return fraction<T>(n * other.d + other.n * d, d * other.d);
    }

    fraction<T> operator -(const fraction<T>& other) const {
        return fraction<T>(n * other.d - other.n * d, d * other.d);
    }

    fraction<T> operator *(const fraction<T>& other) const {
        return fraction<T>(n * other.n, d * other.d);
    }

    fraction<T> operator /(const fraction<T>& other) const {
```

```
        return fraction<T>(n * other.d, d * other.n);
    }

    bool operator <(const fraction<T>& other) const {
        return n * other.d < other.n * d;
    }

    bool operator <=(const fraction<T>& other) const {
        return !(other < *this);
    }

    bool operator >(const fraction<T>& other) const {
        return other < *this;
    }

    bool operator >=(const fraction<T>& other) const {
        return !(*this < other);
    }

    bool operator ==(const fraction<T>& other) const {
        return n == other.n && d == other.d;
    }

    bool operator !=(const fraction<T>& other) const {
        return !(*this == other);
    }
};
```

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.

```
int nck(int n, int k) {
    if (n - k < k)
        k = n - k;

    int res = 1;
    for (int i = 1; i <= k; i++)
        res = res * (n - (k - i)) / i;

    return res;
}
```

```
void nck_precompute(int** arr, int n) {
    for (int i = 0; i < n; i++)
        arr[i][0] = arr[i][i] = 1;

    for (int i = 1; i < n; i++)
        for (int j = 1; j < i; j++)
            arr[i][j] = arr[i - 1][j - 1] + arr[i - 1][j];
}
```

5.3 Euclidean algorithm

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

```
int _gcd_(int a, int b) {
    return b == 0 ? a : _gcd_(b, a % b);
}

int gcd(int a, int b) {
    return (a < 0 && b < 0 ? -1 : 1) * _gcd_(abs(a), abs(b));
}
```

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

```
int _egcd_(int a, int b, int& x, int& y) {
    if (b == 0) {
        x = 1; y = 0;
        return a;
    } else {
        int d = _egcd_(b, a % b, x, y);
        x -= a / b * y;
        swap(x, y);
        return d;
    }
}

int egcd(int a, int b, int& x, int& y) {
    int d = _egcd_(abs(a), abs(b), x, y);
    if (a < 0) x = -x;
    if (b < 0) y = -y;
    return a < 0 && b < 0 ? -d : d;
}
```

5.4 Trial Division Primality Testing

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

```
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.

```
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

```
#include "egcd.cpp"

int mod_inv(int a, int m) {
    assert(m > 1);
    int x, y, d = egcd(a, m, x, y);
    if (d != 1) return -1;
    return x < 0 ? x + m : x;
}
```

5.7 Modular Exponentiation

```
template <class T>
T mod_pow(T b, T e, T m) {
    assert(e >= T(0));
    assert(m > T(0));
    T res = T(1);
    while (e) {
        if (e & T(1)) res = mod(res * b, m);
        b = mod(b * b, m);
```

```
9           e >>= T(1);
10      }
11
12      return res;
13  }
```

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, ..., 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

```
#include <complex>
#define P(p) const point &p
#define L(p0, p1) P(p0), P(p1)

typedef complex<double> point;
typedef vector<point> polygon;
double dot(P(a), P(b)) { return real(conj(a) * b); }
double cross(P(a), P(b)) { return imag(conj(a) * b); }
point rotate(P(p), P(about), double radians) { return (p - about) * exp(point(0,
    radians)) + about; }
point reflect(P(p), L(about1, about2)) {
    point z = p - about1, w = about2 - about1;
    return conj(z / w) * w + about1; }
bool parallel(L(a, b), L(p, q)) { return abs(cross(b - a, q - p)) < EPS; }
double ccw(P(a), P(b), P(c)) { return cross(b - a, c - b); }
bool collinear(P(a), P(b), P(c)) { return abs(ccw(a, b, c)) < EPS; }
bool collinear(L(a, b), L(p, q)) { return abs(ccw(a, b, p)) < EPS && abs(ccw(a, b,
    q)) < EPS;  }
double angle(P(a), P(b), P(c)) { return acos(dot(b - a, c - b)) / abs(b - a) /
    abs(c - b); }
double signed_angle(P(a), P(b), P(c)) { return asin(cross(b - a, c - b)) / abs(b -
    a) / abs(c - b); }
bool intersect(L(a, b), L(p, q), point &res, bool segment = false) {
    // NOTE: check for parallel/collinear lines before calling this function
    point r = b - a, s = q - p;
    double c = cross(r, s), t = cross(p - a, s) / c, u = cross(p - a, r) / c;
    if (segment && (t < 0-EPS || t > 1+EPS || u < 0-EPS || u > 1+EPS)) return false;
    res = a + t * r;
    return true;
}
point closest_point(L(a, b), P(c), bool segment = false) {
    if (segment) {
```

```
29          if (dot(b - a, c - b) > 0) return b;
30          if (dot(a - b, c - a) > 0) return a;
31      }
32      double t = dot(c - a, b - a) / norm(b - a);
33      return a + t * (b - a);
34  }
35  double polygon_area_signed(polygon p) {
36      double area = 0; int cnt = size(p);
37      for (int i = 1; i + 1 < cnt; i++) area += cross(p[i] - p[0], p[i + 1] - p[0]);
38      return area / 2;
39  }
40  double polygon_area(polygon p) { return abs(polygon_area_signed(p)); }
```

6.0.3 Convex Hull

```
1   #include "primitives.cpp"
2
3   point ch_main;
4   bool ch_cmp(P(a), P(b)) {
5       if (collinear(ch_main, a, b)) return abs(a - ch_main) < abs(b - ch_main);
6       return atan2(imag(a) - imag(ch_main), real(a) - real(ch_main)) < atan2(imag(b)
            - imag(ch_main), real(b) - real(ch_main));
7   }
8
9   polygon convex_hull(polygon pts, bool add_collinear = false) {
10      int cnt = size(pts), main = 0, i = 1;
11      if (cnt <= 3) return pts;
12      for (int i = 1; i < cnt; i++)
13          if (imag(pts[i]) < imag(pts[main]) || abs(imag(pts[i]) - imag(pts[main]) <
                EPS && imag(pts[i]) > imag(pts[main])))
14              main = i;
15      swap(pts[0], pts[main]);
16      ch_main = pts[0];
17      sort(++pts.begin(), pts.end(), ch_cmp);
18      point prev, now;
19      stack<point> S; S.push(pts[cnt - 1]); S.push(pts[0]);
20      while (i < cnt) {
21          now = S.top(); S.pop();
22          if (S.empty()) {
23              S.push(now);
24              S.push(pts[i++]);
25          } else {
26              prev = S.top();
27              S.push(now);
28              if (ccw(prev, now, pts[i]) > 0 || (add_collinear && abs(ccw(prev, now,
                    pts[i])) < EPS)) S.push(pts[i++]);
29              else S.pop();
30          }
31      }
32      vector<point> res;
33      while (!S.empty()) res.push_back(S.top()), S.pop();
34      return res;
35  }
```

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.

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

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

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

7.2 2SAT

```
#include "../graph/scc.cpp"

bool two_sat(int n, const vii& clauses, vi& all_truthy) {
    all_truthy.clear();
    vvi adj(2*n+1);
    for (int i = 0, cnt = size(clauses); i < cnt; i++) {
        adj[-clauses[i].first + n].push_back(clauses[i].second + n);
        if (clauses[i].first != clauses[i].second)
            adj[-clauses[i].second + n].push_back(clauses[i].first + n);
    }
    pair<union_find, vi> res = scc(adj);
```

```
12      union_find scc = res.first;
13      vi dag = res.second;
14      vi truth(2*n+1, -1);
15      for (int i = 2*n; i >= 0; i--) {
16          int cur = order[i] - n, p = scc.find(cur + n), o = scc.find(-cur + n);
17          if (cur == 0) continue;
18          if (p == o) return false;
19          if (truth[p] == -1) truth[p] = 1;
20          truth[cur + n] = truth[p];
21          truth[o] = 1 - truth[p];
22          if (truth[p] == 1) all_truthy.push_back(cur);
23      }
24  }
```

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 Fast Input Reading

If input or output is huge, sometimes it is beneficial to optimize the input reading/output writing. This can be achieved by reading all input in at once (using fread), and then parsing it manually. Output can also be stored in an output buffer and then dumped once in the end (using fwrite). A simpler, but still effective, way to achieve speed is to use the following input reading method.

```
1  void readn(register int *n) {
2      int sign = 1;
3      register char c;
4      *n = 0;
5      while((c = getc_unlocked(stdin)) != '\n') {
6          switch(c) {
7              case '-': sign = -1; break;
8              case ' ': goto hell;
9              case '\n': goto hell;
10             default: *n *= 10; *n += c - '0'; break;
11         }
12     }
13 hell:
14     *n *= sign;
15 }
```

8.3 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)

8.4 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 }
```