# Competitive Programming

Bjarki Ágúst Guðmundsson     Trausti Sæmundsson     Ingólfur Eðvarðsson

November 3, 2012

# Contents

# 1 Data Structures

## 1.1 Union-Find

```cpp
class union_find {
private:
        int* parent;
        int cnt;

public:
        union_find(int n) {
                parent = new int[cnt = n];
                for (int i = 0; i < cnt; i++)
                        parent[i] = i;
        }

        union_find(const union_find& other) {
                parent = new int[cnt = other.cnt];
                for (int i = 0; i < cnt; i++)
                        parent[i] = other.parent[i];
        }

        ~union_find() {
                if (parent) {
                        delete[] parent;
                        parent = NULL;
                }
        }

        int find(int i) {
                assert(parent != NULL);
                return parent[i] == i ? i : (parent[i] = find(parent[i]));
        }

        bool unite(int i, int j) {
                assert(parent != NULL);

                int ip = find(i),
                        jp = find(j);

                parent[ip] = jp;
                return ip != jp;
        }
};
```

## 1.2 Segment Tree

## 1.3 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.

```cpp
class fenwick_tree {
private:
        int* arr;
        int cnt;

        inline int lsone(int n) {
                return n & -n;
        }

public:
        fenwick_tree(int n) {
                arr = new int[(cnt = n) + 1];
                memset(arr, 0, (cnt + 1) << 2);
        }

        ~fenwick_tree() {
                if (arr) {
                        delete[] arr;
                        arr = NULL;
                }
        }

        fenwick_tree(const fenwick_tree& other) {
                arr = new int[(cnt = other.cnt) + 1];
                for (int i = 0; i <= cnt; i++)
                        arr[i] = other.arr[i];
        }

        void adjust(int i, int v) {
                assert(arr != NULL);
                assert(i >= 0 && i < cnt);

                i++;
                while (i <= cnt) {
                        arr[i] += v;
                        i += lsone(i);
                }
        }

        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  };
```

### 1.4  Interval Tree

## 2  Graphs

### 2.1  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  }
```

## 2.2   Depth-First Search

## 2.3   Single Source Shortest Path

### 2.3.1   Dijkstra's algorithm

An implementation of Dijkstra's algorithm that returns the length of the shortest path from the starting vertex to the ending vertex. It runs in $\Theta(|E| \log |V|)$ time.

```
1   int dijkstra(int start, int end, vvii& adj_list) {
2           set<int> done;
3           priority_queue<ii, vii, greater<ii> > pq;
4           pq.push(ii(0, start));
5
6           while (!pq.empty()) {
7                   ii current = pq.top(); pq.pop();
8
```

```
 9                  if (done.find(current.second) != done.end())
10                          continue;
11
12                  if (current.second == end)
13                          return current.first;
14
15                  done.insert(current.second);
16
17                  vii &vtmp = adj_list[current.second];
18                  for (vii::iterator it = vtmp.begin(); it != vtmp.end();
                       it++)
19                          if (done.find(it->second) == done.end())
20                                  pq.push(ii(current.first + it->first,
                                          it->second));
21          }
22
23          return -1;
24  }
```

Another implementation 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    }
```

### 2.3.2 Bellman-Ford algorithm

## 2.4 All Pairs Shortest Path

### 2.4.1 Floyd-Warshall algorithm

## 2.5 Connected Components

### 2.5.1 Modified Breadth-First Search

## 2.6 Strongly Connected Components

### 2.6.1 Kosaraju's algorithm

### 2.6.2 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  }
```

## 2.7   Topological Sort

### 2.7.1   Modified Breadth-First Search

## 2.8   Articulation Points/Bridges

### 2.8.1   Modified Depth-First Search

# 3   Mathematics

## 3.1   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   }
```

## 3.2 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   }
```

### 3.3 Modular Multiplicative Inverse

```cpp
#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;
}
```

### 3.4 Modular Exponentiation

```cpp
int mod_pow(int b, int e, int m) {
        assert(e >= 0);
        assert(m > 0);
        int res = 1;
        while (e) {
                if (e & 1) res = mod(res * b, m);
                b = mod(b * b, m);
                e >>= 1;
        }

        return res;
}
```

### 3.5 Chinese Remainder Theorem

```cpp
#include "gcd.cpp"
#include "egcd.cpp"

int crt(const vi& as, const vi& ns) {
        assert(size(as) == size(ns));
        int cnt = size(as), N = 1, x = 0, r, s, l;
        for (int i = 0; i < cnt; i++) {
                N *= ns[i];
                assert(ns[i] > 0);
                for (int j = 0; j < cnt; j++)
                        if (i != j)
                                assert(gcd(ns[i], ns[j]) == 1);
        }

        for (int i = 0; i < cnt; i++) {
                egcd(ns[i], l = N/ns[i], r, s);
                x += as[i] * s * l;
        }

        return mod(x, N);
```

21   `}`

# 4   Useful Information

## 4.1   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?

## 4.2   Worst Time Complexity

| $n$ | Worst AC Algorithm | Comment |
|:---:|:---:|:---:|
| $\leq 10$ | $O(n!), O(n^6)$ | e.g. Enumerating a permutation |
| $\leq 15$ | $O(2^n \times n^2)$ | e.g. DP TSP |
| $\leq 20$ | $O(2^n), O(n^5)$ | e.g. DP + bitmask technique |
| $\leq 50$ | $O(n^4)$ | 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) |

## 4.3 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 }
```