Competitive Programming

Bjarki Ágúst Guðmundsson Trausti Sæmundsson Ingólfur Eðvarðsson

October 29, 2012

Contents

1 Data Structures

1.1 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 };
```

1.2 Segment Tree

1.3 Fenwick Tree

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.

```
int bfs(int start, int end, vvi& adj_list) {
        queue<ii> Q;
        Q.push(ii(start, 0));

        while (true) {
                ii cur = Q.front(); Q.pop();

                if (cur.first == end)
                        return cur.second;

                vi& adj = adj_list[cur.first];
                for (vi::iterator it = adj.begin(); it != adj.end(); it++)
                        Q.push(ii(*it, cur.second + 1));
        }
}
```

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.

```
int bfs(int start, int end, vvi& adj_list) {
        set<int> visited;
        queue<ii> Q;
        Q.push(ii(start, 0));
        visited.insert(start);

        while (!Q.empty()) {
                ii cur = Q.front(); Q.pop();

                if (cur.first == end)
                        return cur.second;

                vi& adj = adj_list[cur.first];
                for (vi::iterator it = adj.begin(); it != adj.end(); it++)
                        if (visited.find(*it) == visited.end()) {
                                Q.push(ii(*it, cur.second + 1));
                                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.

```
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 }
```

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.

```
int factorial(int n) {
        int res = 1;
        while (n) res *= n--;
        return res;
}

int nck(int n, int k) {
        return factorial(n) / factorial(k) / factorial(n - k);
}
```

```
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];
}
```

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