Competitive Programming

Bjarki Ágúst Guðmundsson Trausti Sæmundsson Ingólfur Eðvarðsson

November 15, 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 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  };
```

1.3 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 * rows + 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[other.rows * k + 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  };
```

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 Single-Source Shortest Paths

2.2.1 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  }
```

2.2.2 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], dist[j] +
                                        adj[j][k].second);
11          for (int j = 0; j < n; j++)
12                  for (int k = 0, len = size(adj[j]); k < len; k++)
13                          if (dist[j] + adj[j][k].second <
                                dist[adj[j][k].first])
14                                  has_negative_cycle = true;
15          return dist;
16  }
```

2.3 All-Pairs Shortest Paths

2.3.1 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]);
```

```
}
```

2.4 Connected Components

2.5 Strongly Connected Components

2.5.1 Tarjan's algorithm

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

```
stack<int> t_s;
set<int> t_z;
int t_index;
int* t_idx;
int* t_ll;

void t_strongconnect(int v, vvi& adj) {
        t_idx[v] = t_ll[v] = t_index++;
        t_s.push(v);
        t_z.insert(v);

        int cnt = size(adj[v]);
        for (int i = 0; i < cnt; i++) {
                int w = adj[v][i];
                if (t_idx[w] == -1) {
                        t_strongconnect(w, adj);
                        t_ll[v] = min(t_ll[v], t_ll[w]);
                } else if (t_z.find(w) != t_z.end())
                        t_ll[v] = min(t_ll[v], t_idx[w]);
        }

        if (t_ll[v] == t_idx[v])
                while (true) {
                        // Vertices from top of stack down to v form a SCC
                        int w = t_s.top(); t_s.pop();
                           t_z.erase(t_z.find(w));
                        if (w == v) break;
                }
}

int* tarjan_scc(vvi adj) {
        int n = size(adj);
        t_idx = new int[n];
        t_ll = new int[n];
        t_index = 0;

        for (int i = 0; i < n; i++)
                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  }
```

2.6 Minimum Spanning Tree

2.6.1 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  }
```

2.7 Topological Sort

2.7.1 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  }
```

2.8 Articulation Points/Bridges

3 Strings

3.1 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      }
```

4 Mathematics

4.1 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  };
```

4.2 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) {
```

```
        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;

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

        return res;
}
```

4.3 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 {
```

```
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 }
```

4.4 Trial Division Primality Testing

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

```
1  bool is_prime(int n) {
2         if (n < 2) return false;
3         if (n < 4) return true;
4         if (n % 2 == 0 || n % 3 == 0) return false;
5         if (n < 25) return true;
6
7         int s = static_cast<int>(sqrt(static_cast<double>(n)));
8         for (int i = 5; i <= s; i += 6) {
9                 if (n % i == 0) return false;
10                if (n % (i + 2) == 0) return false;
11        }
12
13        return true;
14 }
```

4.5 Sieve of Eratosthenes

An optimized implementation of Eratosthenes' Sieve.

```
1  vector<int> prime_sieve(int n) {
2         int mx = (n - 3) >> 1;
3         vector<int> primes;
4         bool* prime = new bool[mx + 1];
5         memset(prime, 1, mx + 1);
6
7         if (n >= 2)
8                 primes.push_back(2);
9
10        int i = 0;
11        for ( ; i <= mx; i++)
```

```
12                  if (prime[i]) {
13                          int v = (i << 1) + 3;
14                          primes.push_back(v);
15                          int sq = i * ((i << 1) + 6) + 3;
16                          if (sq > mx) break;
17                          for (int j = sq; j <= mx; j += v)
18                                  prime[j] = false;
19                  }
20
21          for (i++; i <= mx; i++)
22                  if (prime[i])
23                          primes.push_back((i << 1) + 3);
24
25          delete[] prime; // can be used for O(1) lookup
26          return primes;
27  }
```

4.6 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  }
```

4.7 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 }
```

4.8 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 Other Algorithms

5.1 Binary Search

An implementation of binary search that finds a real value x on the real interval $[a, b]$ such that $|f(x) - t| < \epsilon$, where t is the target value and ϵ is the maximum error allowed.

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

Another implementation that finds an integer value x on the integer interval $[a, b]$ such that $f(x) = t \wedge f(x-1) < t$, where t is the target value.

```
1  int binary_search_low(int low, int high, int target, int (*f)(int)) {
2          while (low <= high) {
3                  int mid = (low + high) / 2, cur = f(mid);
4                  if (target <= cur) high = mid - 1;
5                  else low = mid + 1;
6          }
7          return low;
8  }
```

Another implementation that finds an integer value x on the integer interval $[a, b]$ such that $f(x) = t \wedge f(x+1) > t$, where t is the target value.

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

6 Useful Information

6.1 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?

6.2 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)

6.3 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 }
```