Competitive Programming Algorithms

Extracted from CP3 and December Algorithmics IOI Training Resources

January 2015

1 Data Structures

1.1 Stack

```
stack<int> s;
s.push(1); s.push(2); s.push(3);
while (!s.empty()) {
    cout << s.top() << endl;
    s.pop();
} // prints 3 2 1
```

1.2 Sets

A set will contain only distinct elements. O(log n) insert, delete, search.

```
set<int> s;
s.insert(1); s.insert(2); s.insert(3); s.insert(3); // set contains 1 2 3
s.erase(1);                                          // set contains 2 3
if (s.find(1) == s.end()) cout << "Can't find 1\n";
if (s.find(2) != s.end()) cout << "Can find 2\n";
```

1.3 Maps

Associative maps: get a value by a unique key. Like a set (unique key) with data attached. O(log n) insert, delete, search

```
map<int, int> m;
m[5] = 34; m[33] = 1234;
m.size(); // 2: only contains the elements you added
map<string, int> ms;
ms["hello world"] = 42;
ms["leeroy jenkins"] = 12345;
// iterating through elements
map<int,int>::iterator it;
for (it = tour.begin(); it != tour.end(); ++it){
    cout << it->first << "->" << it->second << "\n";
}
```

1.4 Bitmasks

1. To set/turn on the j-th item (0-based indexing) of the set, use the bitwise OR operation $S \mid= (1 << j)$.
2. To check if the j-th item of the set is on, use the bitwise AND operation $T = S \ \& \ (1 << j)$.
   If T = 0, then the j-th item of the set is off.
   If T != 0 (to be precise, $T = (1 << J)$), then the j-th item of the set is on.

3. To clear/turn off the j-th item of the set, use the bitwise AND operation.

   ```
   S &= ~(1 << j) // ~ is the bitwise NOT operation
   ```

4. To toggle the j-th item of the set, use the bitwise XOR operation

   ```
   S ^= (1 << j)
   ```

5. To get the value of the least significant bit that is on (first from the right), use T = (S & (-S)).
6. To turn on all bits in a set of size n, use $S = (1 << n) - 1$.

1.5 Union-Find Disjoint Sets

```
class UnionFind{
    private: vi p, rank;
    public:
        UnionFind(int N) {
            rank.assign(N,0); p.assign(N,0);
            for(int i = 0; i < N; i++) p[i] = i;
        }
        int findSet(int i){
            return (p[i] == i) ? i : (p[i] = findSet(p[i]));
        }
        bool isSameSet(int i, int j){
            return findSet(i) == findSet(j);
        }
        void unionSet(int i, int j){
            if(!isSameSet(i,j)){
                int x = findSet(i), y = findSet(j);
                if(rank[x] > rank[y]) p[y] = x;
                else{
                    p[x] = y;
                    if(rank[x] == rank[y]) rank[y]++;
}}}};
```

1.6 Fenwick Trees

1.6.1 Implementation theory

Querying To query the range from 1 to i, add the buckets at position:
$p_0 = i$,
$p_1 = p_0 -$ size of bucket p_0,
$p_2 = p_1 -$ size of bucket p_1, etc
Subtract size of bucket until 0

Updating To update, the ranges that contain i are:
$p_0 = i$,
$p_1 = p_0 +$ size of bucket p_0
$p_2 = p_1 +$ size of bucket p_1, etc

```
long long ft[N + 1]; // note: Fenwick tree must be 1-indexed.
int ls(int x) { return x & (-x); }

void fenwick_update(int p, long long v){
    for (; p <= N; p += ls(p)) ft[p] += v;
}

long long fenwick_query(int p){
    long long sum = 0;
    for (; p; p -= ls(p)) sum += ft[p];
    return sum;
}
```

2 Sorts

sort, O(n log n) - sorts entire array
stable_sort, O(n log n) - keeps original order between equal elements
partial_sort, O(n log k) - sorts the k smallest entries

3 Conversions

```
string stlstr = "hello";
printf("%s", stlstr.c_str());

char cstr[] = "world";
cout << string(cstr) << endl;
```

4 Dynamic Programming

4.1 2D-Maxsum

For every pair of rows (eg. x1, x2):

- Sum each column between them (inclusive) into an 1D- array
  - Use W columns of 1D static sum
  - Or 2D static sum works too
- Perform 1D-Maxsum on this array

Complexity: O(H^2W)

```
int G[H+1][W+1], S[H+1][W+1], ans; /* 1-indexed */
/* W rows of 1D Static Sum */
for (int i = 1; i <= H; i++)
    for (int j = 1; j <= W; j++)
        S[i][j] = S[i-1][j] + G[i][j];
for (int x1 = 1; x1 <= H; x1++) {
    for (int x2 = x1; x2 <= H; x2++) {
        int cursum = S[x2][1] - S[x1-1][1];
        for (int y = 2; y <= W; y++) {
            cursum += max(cursum, 0) + S[x2][y] - S[x1-1][y];
            ans = max(cursum, ans);
        }
    }
}
```

4.2 Lowest Common Substring

To recover LCS: start from lcs[N][M] and work backwards.

```
int lcs[1001][1001], A[1001], B[1001]; // A and B are 1-indexed here
for (int i = 0; i <= N; ++i) {
    for (int j = 0; j <= M; ++j) {
        if (i == 0 || j == 0) lcs[i][j] = 0;
        if (A[i] == B[j]) lcs[i][j] = 1 + lcs[i - 1][j - 1];
        else lcs[i][j] = max(lcs[i][j - 1], lcs[i - 1][j]);
    }
}
```

5 Graphs

5.1 Topological Sort

```
void dfs(int vertex_id) {
    if (visited[vertex_id]) return;
    visited[vertex_id] = true;
    for (auto i: adjList[vertex_id]) {
        dfs(i);
    }
    topo.push_back(vertex_id);
}

for (int i = 0; i < V; ++i)
    if (!visited[i]) dfs(i);

reverse(topo.begin(), topo.end());
```

5.2 DFS

Adjacency list: $O(V+E)$. Adjacency matrix: $O(V^2)$.

```
int VISITED = 1, UNVISITED = -1;
vi dfs_num; // global variable, initially all values are set to UNVISITED

void dfs(int u){
    dfs_num[u] = VISITED;
    for(int j = 0; j < (int)AdjList[u].size(); j++){
        ii v = AdjList[u][j];    // v is a (neighbor, weight) pair
        if(dfs_num[v.first] == UNVISITED)    // important check to avoid cycle
            dfs(v.first);
    }   // for simple graph traversal, we ignore the weight stored at v.second
}
```

5.2.1 DFS Backtracking

```
void backtrack(state){
    if (hit end state or invalid state) // we need terminating of pruning conditions
        return; // to avoid cycling and to speed up search
    for each neighbor of this state // try all permutations
        backtrack(neighbor);
}
```

5.3 Kruskal's

```
vector<pair<int,ii> > EdgeList; // (weight, two vertices) of the edge
for(int i = 0; i < E; i++){
    scanf("%d %d %d", &u, &v, &w);
    EdgeList.push_back(make_pair(w, ii(u, v)));
}
sort(EdgeList.begin(), EdgeList.end());

int mst_cost = 0;
UnionFind UF(V);
for(int i = 0; i < E; i++){
    pair<int, ii> front = EdgeList[i];
    if(!UF.isSameSet(front.second.first, front.second.second)){
        mst_cost += front.first;
        UF.unionSet(front.second.first, front.second.second);
    }
} // note: number of disjoint sets must eventually be 1 for a valid MST
printf("MST cost = %d", mst_cost);
```

5.4 Dijkstra's

O((V+E)logV), best for weighted graphs, works for negative weights (slower), unable to detect negative cycle.

```
vi dist(V, INF); dist[s] = 0;          // INF = 1B to avoid overflow
priority_queue<ii, vector<ii>, greater<ii> > pq; pq.push(ii(0,s));
while(!pq.empty()){
    ii front = pq.top(); pq.pop();
    int d = front.first, u = front.second;
    if(d > dist[u]) continue;
    for(int j = 0; j < (int)AdjList[u].size(); j++){
        ii v = AdjList[u][j];
        if(dist[u] + v.second < dist[v.first]){
            dist[v.first] = dist[u] + v.second;
            pq.push(ii(dist[v.first],v.first));
        }
    }
}
```

5.5 Bellman Ford's

O(VE), works for negative weight.

```
vi dist(V, INF); dist[s] = 0;
for(int i = 0; i < V - 1; i++){       // relax all E edges V-1 times
    for(int u = 0; u < V; u++){
        for(int j = 0; j < (int)AdjList[u].size(); j++){
            ii v = AdjList[u][j];
            dist[v.first] = min(dist[v.first], dist[u] + v.second);
        } //relax
    }
}
```

Checking for negative cycles

```
// after running Bellman Ford's algorithm shown above
bool hasNegativeCycle = false;
for(int u = 0; u < V; u++){
    for(int j = 0; j < (int)AdjList[u].size(); j++){
        ii v = AdjList[u][j];
        if(dist[v.first] > dist[u] + v.second)
            hasNegativeCycle = true;
    }
}
printf("Negative Cycle Exist? %s\n", hasNegativeCycle ? "Yes" : "No");
```

5.6 Floyd Warshall's

V $\leq$ 400. O(V^3)

```
// inside int main()
// precondition: AdjMat [i] [j] contains the weight of edge (i,j)
// or INF (1B) if there is no such edge
// AdjMat is a 32-bit signed integer array
for (int k = 0; k < V; k++) // remember that loop order is k->i->j
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            AdjMat [i][j] = min(AdjMat [i][j] , AdjMat [i][k] + AdjMat[k][j]);
```

6 Mathematics

6.1 Modular Arithmetic

$$a \equiv b(modm) \leftrightarrow a - b \equiv 0(modm),$$

$$k * m \equiv 0(modm),$$

$$a * c \equiv b * c(modm) \leftrightarrow a \equiv b(modm)$$

if m and c are coprime because of Euclid's lemma: If a prime divides the product of two numbers, it must divide at least one of those numbers.

6.2 Fast Exponentiation

```
int fastExp(ll base, ll p){
    if(p==0) return 1;
    else if(p==1) return base;
    else{
        ll res = fastExp(base,p/2);
        res *= res;
        res %= MOD;
        if(p%2==1) res *= base%MOD;
        return res%MOD;
        }
}
```

6.3 GCD and LCM

```cpp
//Euclid's algorithm, O(log N) time.
int gcd(int a, int b) { return b == 0 ? a : gcd(b, a%b);}
int lcm(int a, int b) { return a * (b / gcd(a, b)); }
```

$$lcm(m, n) = \frac{|m \cdot n|}{gcd(m, n)}$$

LCM can also be found by merging the prime decompositions of both m and n.

6.4 Sieve of Eratosthenes

```cpp
ll sieveSize;
bitset<10000010> bs;
vi primes; // list of primes

void sieve(ll upperbound){  // create list of primes in [0..upperbound]
    sieveSize = upperbound + 1; // add 1 to include upperbound
    bs.set();   // set all bits to 1
    bs[0] = bs[1] = 0;  // except index 0 and 1
    for(ll i = 2; i <= sieveSize; i++) if (bs[i]){
        // cross out multiples of i starting from i * i
        for(ll j = i * i; j <= sieveSize; j += i) bs[j] = 0;
        primes.push_back(i);
}}

bool isPrime(ll N){
    if(N <= sieveSize) return bs[N];
    for(int i = 0; i < (int)primes.size(); i++)
        if(N % primes[i] == 0) return false; // a good enough deterministic prime tester
    return true;
}   // only works for N <= (last prime in vi "primes")^2

int main(){
    sieve(10000000);
    printf("%d\n", isPrime(2147483647));    // return true
    printf("%d\n", isPrime(136117223861));  // return false
}
```