[UVic][Vicky Nguyen] ICPC 2023 Notes

Fast I/O

```cpp
ios_base::sync_with_stdio(false);
cin.tie(NULL);
```

Union Find

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef vector<int> vi;

class UF {
private:
  vi parent, rank;

public:
  UF(int N) {
    parent.assign(N, 0);
    for (int i = 0; i < N; i++)
      parent[i] = i;
    rank.assign(N, 0);
  }

  int find(int v) {
    if (parent[v] != v)
      parent[v] = find(parent[v]);
    return parent[v];
  }

  bool same_set(int a, int b) { return find(a) == find(b); }

  void union_set(int a, int b) {
    int ra = find(a), rb = find(b);
    if (ra == rb)
      return;
    if (rank[ra] < rank[rb])
      swap(ra, rb);
    parent[rb] = ra;
    if (rank[ra] == rank[rb])
      rank[ra]++;
  }
};

int main() {
  UF uf(5);
  uf.union_set(1, 2);
  cout << "same_set(1, 2): " << uf.same_set(1, 2) << '\n';
  cout << "same_set(1, 3): " << uf.same_set(1, 3) << '\n';
  for (int i = 0; i < 5; i++)
    cout << "root of " << i << ": " << uf.find(i) << '\n';

  return 0;
}
```

MST

```cpp
#include <bits/stdc++.h>
using namespace std;
using vi = vector<int>;
using iii = tuple<int, int, int>;

// Union Find class here ...

// Kruskal's algorihm
class MST {
public:
  int N;
  vector<iii> EL;

  MST(const int n) : N(n), uf(n) {}

  void add_edge(const int u, const int v, const int w) {
    EL.push_back({w, u, v});
  }

  void run_mst() {
    sort(EL.begin(), EL.end());

    int edge_count = 0;
    for (auto [w, u, v] : EL) {
      if (edge_count == (N - 1))
        break;
      if (!uf.is_same_set(u, v)) {
        uf.union_set(u, v);
        ++edge_count;
        mst_cost += w;
        mst_edges.push_back({u, v, w}); // order: u, v, w
      }
    }
  }

  int get_mst_cost() { return mst_cost; }

  vector<iii> get_mst_edges() { return mst_edges; }

private:
  UF uf;
  vector<iii> mst_edges;
  int mst_cost = 0;
};

/*
input: n, m; n: number of nodes, m: number of edges. Then follow m lines, each
line consisting of u, v and w indicating that there is an edge between u and v
in the graph with weight w
4 4
0 1 1
1 2 2
1 3 3
2 3 0
```

```
output: minimum spanning tree cost, and edges in lexicographic order
3
0 1
1 2
2 3
*/

int main() {
  MST mst(4);
  mst.add_edge(0, 1, 1);
  mst.add_edge(1, 2, 2);
  mst.add_edge(1, 3, 3);
  mst.add_edge(2, 3, 0);

  mst.run_mst();
  cout << mst.get_mst_cost() << '\n';

  vector<iii> mst_edges = mst.get_mst_edges();

  // order u, v
  for (auto &[u, v, w] : mst_edges) {
    if (u > v)
      swap(u, v);
  }

  sort(mst_edges.begin(), mst_edges.end());

  for (auto &[u, v, w] : mst_edges) {
    cout << u << " " << v << '\n';
  }

  return 0;
}
```

Max Flow

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef vector<int> vi;
typedef pair<int, int> ii;
typedef pair<ll, ll> pll;
#define all(x) (x).begin(), (x).end()
#define pb push_back

// Reference:
// https://github.com/stevenhalim/cpbook-code/blob/master/ch8/maxflow.cpp
typedef tuple<int, ll, ll> edge;
const ll INF = 1e18;

class max_flow {
private:
  int V;
  vector<edge> EL;
  vector<vi> AL;
  vi d, last;
  vector<ii> p;
  unordered_map<ll, int> AL_EL_map;

  bool BFS(int s, int t) { // find augmenting path
    d.assign(V, -1);
    d[s] = 0;
    queue<int> q({s});
    p.assign(V, {-1, -1}); // record BFS sp tree
    while (!q.empty()) {
      int u = q.front();
      q.pop();
      if (u == t)
        break;                                          // stop as sink t
      reached
      for (auto &idx : AL[u]) {                         // explore
      neighbors of u
        auto &[v, cap, flow] = EL[idx];                 // stored in EL[
      idx]
        if ((cap - flow > 0) && (d[v] == -1))           // positive
      residual edge
          d[v] = d[u] + 1, q.push(v), p[v] = {u, idx}; // 3 lines in one!
      }
    }
    return d[t] != -1; // has an augmenting path
  }

  ll DFS(int u, int t, ll f = INF) { // traverse from s->t
    if ((u == t) || (f == 0))
      return f;
    for (int &i = last[u]; i < (int)AL[u].size(); ++i) { // from last
    edge
      auto &[v, cap, flow] = EL[AL[u][i]];
      if (d[v] != d[u] + 1)
        continue; // not part of layer graph
      if (ll pushed = DFS(v, t, min(f, cap - flow))) {
        flow += pushed;
        auto &rflow = get<2>(EL[AL[u][i] ^ 1]); // back edge
        rflow -= pushed;
        return pushed;
      }
    }
    return 0;
  }

public:
  max_flow(int initialV) : V(initialV) {
    EL.clear();
    AL.assign(V, vi());
  }

  // if you are adding a bidirectional edge u<->v with weight w into your
  // flow graph, set directed = false (default value is directed = true)
  void add_edge(int u, int v, ll w, bool directed = true) {
    if (u == v)
      return;                          // safeguard: no self loop
    EL.emplace_back(v, w, 0);          // u->v, cap w, flow 0
    AL[u].push_back(EL.size() - 1); // remember this index
    AL_EL_map[u * V + v] = EL.size() - 1;
    EL.emplace_back(u, directed ? 0 : w, 0); // back edge
    AL[v].push_back(EL.size() - 1);          // remember this index
```

```
    AL_EL_map[v * V + u] = EL.size() - 1;
  }

  ll dinic(int s, int t) {
    ll mf = 0;                  // mf stands for max_flow
    while (BFS(s, t)) {         // an O(V^2*E) algorithm
      last.assign(V, 0);        // important speedup
      while (ll f = DFS(s, t)) // exhaust blocking flow
        mf += f;
    }
    return mf;
  }

  ll get_flow_edge(int u, int v) { return get<2>(EL[AL_EL_map[u * V + v
    ]]); }
};
```

Dijkstra

```
#include <bits/stdc++.h>
using namespace std;
using ii = pair<int, int>;
constexpr int INF = 1e9;

/*
 * Graph:
 * 4 5
 * 0 1 5
 * 0 2 2
 * 0 3 10
 * 2 3 3
 * 1 3 1
 * src=0
 */

int main() {
  // create a graph
  int n = 4, s = 0;
  vector<vector<pair<int, int>>> graph(4);
  graph[0].push_back({1, 5});
  graph[1].push_back({0, 5});
  graph[0].push_back({2, 2});
  graph[2].push_back({0, 2});
  graph[0].push_back({3, 10});
  graph[3].push_back({0, 10});
  graph[1].push_back({3, 1});
  graph[3].push_back({1, 1});
  graph[2].push_back({3, 3});
  graph[3].push_back({2, 3});

  vector<int> dist(n, INF);
  dist[s] = 0;
  priority_queue<ii, vector<ii>, greater<ii>> pq;
  pq.push({0, s});

  // (Modified) Dijkstra's algorithm
  while (!pq.empty()) {
    auto [d, u] = pq.top();
    pq.pop();
    if (d > dist[u]) continue;
    for (auto &[v, w]: graph[u]) {
      if (dist[u] + w >= dist[v]) continue;
      dist[v] = dist[u] + w; // relax
      pq.push({dist[v], v});
    }
  }

  for (int i = 0; i < n; ++i) {
    cout << "Dist. from " << s << " to " << i << ": " << dist[i] << '\n';
  }

  return 0;
}
```

Topological Sort

```
#include <bits/stdc++.h>
using namespace std;

/* Graph with directed edges
 * 0 1
 * 1 2
 * 1 3
 * 2 4
 * 3 4
 */

vector<vector<int>> graph;
vector<int> indegree;

vector<int> topological_sort(int n) {
  priority_queue<int, vector<int>, greater<int>> pq;
  for (int i = 0; i < n; ++i) {
    if (indegree[i] == 0) {
      pq.push(i);
    }
  }

  vector<int> result;
  result.reserve(n);

  while (!pq.empty()) {
    int u = pq.top();
    pq.pop();

    result.push_back(u);

    for (auto &v: graph[u]) {
      --indegree[v];
      if (indegree[v] == 0) pq.push(v);
    }
  }

  return result;
}

```

```cpp
int main() {
  // create a graph
  int n = 5;
  graph.resize(n);
  graph[0].push_back(1);
  graph[1].push_back(2);
  graph[1].push_back(3);
  graph[2].push_back(4);
  graph[3].push_back(4);

  // Topological sort
  // Take edges with 0 in-degree
  // Need to implement the in-degree vector when solving actual problem
  indegree.resize(n);
  indegree[0] = 0;
  indegree[1] = 1;
  indegree[2] = 1;
  indegree[3] = 1;
  indegree[4] = 2;

  vector<int> result = topological_sort(n);
  for (auto x: result) cout << x << " ";
  return 0;
}
```

Cycle Check

```cpp
#include <bits/stdc++.h>
using namespace std;

enum {UNVISITED = -1, EXPLORED = -2, VISITED = -3};
vector<vector<int>> graph;
vector<int> status;
vector<int> parents;

bool cyclecheck(int u) {
  status[u] = EXPLORED;
  bool hascycle = false;
  for (auto v: graph[u]) {
    if (status[v] == UNVISITED) {
      parents[v] = u;
      hascycle |= cyclecheck(v);
    } else if (status[v] == EXPLORED) {
      if (v == parents[u]) // bidirectional
        continue;
      else // back edge (cycle)
        hascycle = true;
    } else if (status[v] == VISITED) {
        continue;
    }
  }
  status[u] = VISITED;
  return hascycle;
}

/* Graph with bidirectional edges
 * 0 4
 * 4 5
 * 1 2
 * 1 3
 * 2 3
 */

int main() {
  // create a graph
  int n = 6;
  graph.resize(n);
  graph[0].push_back(4);
  graph[4].push_back(0);
  graph[4].push_back(5);
  graph[5].push_back(4);
  graph[1].push_back(2);
  graph[2].push_back(1);
  graph[1].push_back(3);
  graph[3].push_back(1);
  graph[2].push_back(3);
  graph[3].push_back(2);

  status.assign(n, UNVISITED);
  parents.assign(n, -1);
  for (int i = 0; i < n; ++i) {
    if (status[i] == UNVISITED) {
      if (cyclecheck(i)) cout << i << ": cycle\n";
      else cout << i << ": no cycle\n";
    }
  }

  return 0;
}
```

Fenwick

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef vector<int> vi;
typedef vector<ll> vll;

// Reference:
// https://github.com/mission-peace/interview/blob/master/src/com/
    interview/tree/FenwickTree.java
class FenwickTree {
private:
  vll ft;

  /**
   * To get next
   * 1) 2's complement of get minus of index
   * 2) AND this with index
   * 3) Add it to index
   */
  int getNext(int idx) { return idx + (idx & -idx); }

  /**
   * To get parent
   * 1) 2's complement to get minus of index
```

```
   * 2) AND this with index
   * 3) Subtract that from index
   */
  int getParent(int idx) { return idx - (idx & -idx); }

public:
  FenwickTree(int n) { ft.assign(n + 1, 0); }
  FenwickTree(const vll &input) { create(input); }

  void update(ll val, int idx) {
    idx++;
    while (idx < ft.size()) {
      ft[idx] += val;
      idx = getNext(idx);
    }
  }

  void create(const vll &input) {
    int n = input.size();
    ft.assign(n + 1, 0);
    for (int i = 0; i < n; i++) {
      update(input[i], i);
    }
  }

  // get sum from [0, idx-1]
  ll getSum(int idx) {
    ll sum = 0;
    while (idx > 0) {
      sum += ft[idx];
      idx = getParent(idx);
    }
    return sum;
  }
};

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(NULL);

  FenwickTree ft(5);
  ft.update(2, 0);
  ft.update(1, 1);
  ft.update(-5, 2);
  ft.update(20, 3);
  ft.update(1, 4);
  cout << "Fenwick Tree after update: [2,1,-5,20,1] \n";
  cout << "sum [0, 4] = " << ft.getSum(5) << '\n';
  cout << "sum [0, 2] = " << ft.getSum(3) << '\n';

  cout << "Fenwick Tree with initial vector [1,2,-5,3,4]\n";
  vll v = {1, 2, -5, 3, 4};
  FenwickTree t(v);
  cout << "sum [0, 3] = " << t.getSum(4) << '\n';
  cout << "sum [1, 4] = " << (t.getSum(5) - t.getSum(1)) << '\n';

  return 0;
}
```

Find All Shortest Paths

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
using namespace std;

void find_paths(vector<vector<int>>& paths, vector<int>& path, vector<
    vector<int>>& parents, const int n, const int u) {
  if (u == -1) {
    paths.push_back(path);
    return;
  }

  for (int p: parents[u]) {
    path.push_back(u);
    find_paths(paths, path, parents, n, p);
    path.pop_back();
  }
}

vector<vector<int>> get_paths(vector<vector<int>>& graph, const int n,
    const int src, const int dest) {
  vector<vector<int>> paths;
  vector<int> path;
  vector<vector<int>> parents(n);

  vector<int> dist(n, INT_MAX);

  // bfs
  queue<int> q;
  q.push(src);
  dist[src] = 0;
  parents[src].push_back(-1);

  while (!q.empty()) {
    int u = q.front();
    q.pop();

    for (int v: graph[u]) {
      if (dist[v] > dist[u] + 1) {
        dist[v] = dist[u] + 1;
        q.push(v);
        parents[v].clear();
        parents[v].push_back(u);
      } else if (dist[v] == dist[u] + 1) {
        parents[v].push_back(u);
      }
    }
  }

  // recursively find paths
  find_paths(paths, path, parents, n, dest);

  return paths;
}

```

```cpp
int main() {
  const int n = 6;

  vector<vector<int>> graph(n);
  vector<pair<int, int>> edges = {{0,1}, {0,2}, {1,3}, {1,4}, {2,3},
    {3,5}, {4,5}};

  for (auto& [u, v]: edges) {
    graph[u].push_back(v);
    graph[v].push_back(u);
  }

  // get all shortest paths from 0 to 5
  vector<vector<int>> paths = get_paths(graph, n, 0, 5);

  // output
  for (auto& path: paths) {
    for (int i = path.size() - 1; i >= 0; --i) cout << path[i] << " ";
    cout << '\n';
  }

  return 0;
}
```

Prime Factors

```cpp
#include <iostream>
#include <unordered_map>
using namespace std;

unordered_map<int, int> prime_factors(int n) {
  unordered_map<int, int> power;
  while (n % 2 == 0) {
    ++power[2];
    n >>= 1;
  }

  for (int i = 3; i * i <= n; i += 2) {
    while (n % i == 0) {
      ++power[i];
      n /= i;
    }
  }

  if (n != 1) power[n] = 1;
  return power;
}

int main() {
  int n = 4800;
  unordered_map<int, int> power = prime_factors(n);
  for (auto& [k, v]: power) cout << k << ":" << v << '\n';
  return 0;
}
```