Team Notebook

Rubick n Warwick n Friends

March 23, 2024

Contents

1 Advanced **2**
- 1.1 Bipartite Matching . . . 2
- 1.2 Convex Hull . . . 2
- 1.3 FFT . . . 3
- 1.4 Maximum Flow . . . 3

2 Graphs **4**
- 2.1 BFS . . . 4
- 2.2 Bellman-Ford . . . 4
- 2.3 DFS . . . 4
- 2.4 Dijkstra . . . 4
- 2.5 Floyd-Warshall . . . 5
- 2.6 Kosaraju . . . 5
- 2.7 Kruskal's Algorithm . . . 5
- 2.8 Prim's Algorithm . . . 6
- 2.9 Toplogical Sort . . . 6

3 Math **7**
- 3.1 Chinese Remainder Theorem . . . 7
- 3.2 Gauss-Jordan . . . 7
- 3.3 Linear Diophantine . . . 7
- 3.4 Modulo . . . 7
- 3.5 Primes . . . 7

4 Segment Trees **8**
- 4.1 Coordinate Compressed . . . 8
- 4.2 Range Update (Add and Set) . . . 8

5 template **9**

1 Advanced

1.1 Bipartite Matching

```
int b, g, m, n;
struct edge {
  size_t i; // index at edges
  int v, c, f; // directed to v, capacity, flow
  int residue() { return c - f; }
};
struct flow_network {
  int n, s, t;
  vector<edge> edges; // even indeces are forward flows, e_i
      +1 are reverse flows.
  vector<vector<int>> adj; // stores index pointing in edges
  vector<int> parent;
  flow_network(int n, int s, int t) : n(n), s(s), t(t) {
    adj.resize(n);
    parent.resize(n);
}
  void add_edge(int u, int v, int cap) {
    edges.push_back({edges.size(), v, cap, 0});
    adj[u].push_back((int)edges.size()-1);
    edges.push_back({edges.size(), u, 0, 0}); // reverse
    adj[v].push_back((int)edges.size()-1);
}
  bool aug_path() {
    for (int i=0; i<n; i++) parent[i] = -1;
    parent[s] = s;
    queue<int> q;
    q.push(s);
    while (!q.empty()) {
      int u = q.front(); q.pop();
      // cout << "Pop: " << u << endl;
      if (u == t) break;
      for (auto ind : adj[u]){
        edge& e = edges[ind];
        // cout << e.i << " " << e.v << endl;
        if (e.residue() > 0 && parent[e.v] == -1) {
          parent[e.v] = e.i;
          q.push(e.v);
} }
}
    return parent[t] != -1;
  };
  int augment() {
    int bottleneck = numeric_limits<int>::max();
    for (int v = t; v != s; v = edges[parent[v] ^ 1].v) {
      bottleneck = min(bottleneck, edges[parent[v]].residue()
          );
    }
    for (int v = t; v != s; v = edges[parent[v] ^ 1].v) {
      edges[parent[v]].f += bottleneck;
      edges[parent[v] ^ 1].f -= bottleneck;
}
    return bottleneck;
  }
  int calc_max_flow() {
    int flow = 0;
    while (aug_path()){
      flow += augment();
    }
    return flow;
  }
  void get_matching(){
    // only call after running calc_max_flow()
    vector<int> has_match;
    for (auto ind: adj[0]){
      edge& e = edges[ind];
      if (e.residue() == 0) has_match.push_back(e.v);
    }
    for (int a: has_match){
      for (auto ind: adj[a]){
        edge& e = edges[ind];
        if (e.residue() == 0){
          printf("%d %d\n", a, (e.v)-b);
          break;
} }
} }
};
int main(){
  cin >> b >> g >> m;
  n = b+g+2;
  flow_network fn(n, 0, n-1);
  for (int i=0; i<m; i++){
    int u, v;
    cin >> u >> v;
    fn.add_edge(u,b+v,1);
}
  for (int i=1; i<=b; i++){
    fn.add_edge(0, i, 1);
  }
  for (int i=b+1; i<(n-1); i++){
    fn.add_edge(i, n-1, 1);
  }
  cout << fn.calc_max_flow() << endl;
  fn.calc_max_flow();
  fn.get_matching();
}
```

1.2 Convex Hull

```
struct pt {
    double x, y;
    bool operator == (pt const& t) const {
        return x == t.x && y == t.y;
    }
};

int orientation(pt a, pt b, pt c) {
    double v = a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y);
    if (v < 0) return -1; // clockwise
    if (v > 0) return +1; // counter-clockwise
    return 0;
}

bool cw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o < 0 || (include_collinear && o == 0);
}
bool collinear(pt a, pt b, pt c) { return orientation(a, b,
     c) == 0; }

void convex_hull(vector<pt>& a, bool include_collinear =
     false) {
    pt p0 = *min_element(a.begin(), a.end(), [](pt a, pt b) {
        return make_pair(a.y, a.x) < make_pair(b.y, b.x);
    });
    sort(a.begin(), a.end(), [&p0](const pt& a, const pt& b)
         {
        int o = orientation(p0, a, b);
        if (o == 0)
            return (p0.x-a.x)*(p0.x-a.x) + (p0.y-a.y)*(p0.y-a
                .y)
              < (p0.x-b.x)*(p0.x-b.x) + (p0.y-b.y)*(p0.y-b.y
                  );
        return o < 0;
    });
    if (include_collinear) {
        int i = (int)a.size()-1;
        while (i >= 0 && collinear(p0, a[i], a.back())) i--;
        reverse(a.begin()+i+1, a.end());
    }

    vector<pt> st;
    for (int i = 0; i < (int)a.size(); i++) {
        while (st.size() > 1 && !cw(st[st.size()-2], st.back
            (), a[i], include_collinear))
            st.pop_back();
        st.push_back(a[i]);
```

```
    }

    if (include_collinear == false && st.size() == 2 && st[0]
         == st[1])
        st.pop_back();

    a = st;
}
```

1.3 FFT

```
using cd = complex<double>;
const double PI = acos(-1);
void fft(vector<cd> & a, bool invert) {
  int n = a.size();
  if (n == 1)
return;
  vector<cd> a0(n / 2), a1(n / 2);
  for (int i = 0; 2 * i < n; i++) {
    a0[i] = a[2*i];
    a1[i] = a[2*i+1];
  }
  fft(a0, invert);
  fft(a1, invert);
  double ang = 2 * PI / n * (invert ? -1 : 1);
  cd w(1), wn(cos(ang), sin(ang));
  for (int i = 0; 2 * i < n; i++) {
    a[i] = a0[i] + w * a1[i];
    a[i + n/2] = a0[i] - w * a1[i];
    if (invert) {
a[i] /= 2;
      a[i + n/2] /= 2;
    }
w *= wn; }
}
vector<int> multiply(vector<int> const& a, vector<int> const
     & b) {
  vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end());
  int n = 1;
  while (n < a.size() + b.size())
    n <<= 1;
  fa.resize(n);
  fb.resize(n);
  fft(fa, false);
  fft(fb, false);
  for (int i = 0; i < n; i++)
    fa[i] *= fb[i];
  fft(fa, true);
  vector<int> result(n);
  for (int i = 0; i < n; i++){
    result[i] = round(fa[i].real());
  }
  return result;
}
int n, m;
int main(){
  cin >> n >> m;
  vector<int> a(n);
  vector<int> b(m);
  vector<int> r;
  for (int i=0; i<n; i++){cin >> a[i];}
  for (int i=0; i<m; i++){cin >> b[i];}
  r = multiply(a, b);
  for (int i=0; i<r.size(); i++){
    cout << r[i] << " ";
  }
cout << endl;
return 0;}
```

1.4 Maximum Flow

```
struct edge {
  size_t i; // index at edges
  int v, c, f; // directed to v, capacity, flow
  int residue() { return c - f; }
};
struct flow_network {
  int n, s, t;
  vector<edge> edges; // even indeces are forward flows, e_i
       +1 are reverse flows.
  vector<vector<int>> adj; // stores index pointing in edges
  vector<int> parent;
  flow_network(int n, int s, int t) : n(n), s(s), t(t) {
    adj.resize(n);
    parent.resize(n);
}
  void add_edge(int u, int v, int cap) {
    edges.push_back({edges.size(), v, cap, 0});
    adj[u].push_back((int)edges.size()-1);
    edges.push_back({edges.size(), u, 0, 0}); // reverse
    adj[v].push_back((int)edges.size()-1);
}
  bool aug_path() {
    for (int i=0; i<n; i++) parent[i] = -1;
    parent[s] = s;
    queue<int> q;
    q.push(s);
    while (!q.empty()) {
      int u = q.front(); q.pop();
      // cout << "Pop: " << u << endl;
      if (u == t) break;
      for (auto ind : adj[u]){
        edge& e = edges[ind];
        // cout << e.i << " " << e.v << endl;
        if (e.residue() > 0 && parent[e.v] == -1) {
          parent[e.v] = e.i;
          q.push(e.v);
} }
}
    return parent[t] != -1;
  };
  int augment() {
    int bottleneck = numeric_limits<int>::max();
    for (int v = t; v != s; v = edges[parent[v] ^ 1].v) {
      bottleneck = min(bottleneck, edges[parent[v]].residue()
           );
    }
    for (int v = t; v != s; v = edges[parent[v] ^ 1].v) {
      edges[parent[v]].f += bottleneck;
      edges[parent[v] ^ 1].f -= bottleneck;
}
    return bottleneck;
  }
  int calc_max_flow() {
    int flow = 0;
    while (aug_path()){
      flow += augment();
    }
    return flow;
  }
  void get_min_cut(){
    // only call after running calc_max_flow()
    queue<int> q;
    vector<int> vis(n+1, 0);
    vector<vector<int>> pted (51, vector<int>(51, 0));
    q.push(s);
    while (!q.empty()){
      int u = q.front(); q.pop();
      for (auto ind: adj[u]){
        edge& e = edges[ind];
        if (e.residue() > 0 && !vis[e.v]) {
          vis[e.v] = 1;
          q.push(e.v);
} }
    }
    for (int i=0; i<n+1; i++){
      if (vis[i]){
        for (auto ind: adj[i]){
```

```
        edge& e = edges[ind];
        if ( (!vis[e.v]) and (!pted[i][e.v]) ) {
          printf("%d %d\n", i+1, e.v+1);
          pted[i][e.v] = 1;
        }
} }
} }
};
int main(){
  int n, m;
  cin >> n >> m;
  flow_network fn(n, 0, 1);
  for (int i=0; i<m; i++){
    int u, v, cap;
    cin >> u >> v >> cap;
    fn.add_edge(u-1, v-1, cap);
    fn.add_edge(v-1, u-1, cap);
}
  // cout << fn.calc_max_flow() << endl;
  fn.calc_max_flow();
  fn.get_min_cut();
}
```

2 Graphs

2.1 BFS

```
// BFS
void bfs(int n, int s){ // number of nodes, source vertex
    vector<vector<int>> adj; // adjacency list representation
    queue<int> q;
    vector<bool> used(n);
    vector<int> d(n), p(n);

    q.push(s);
    used[s] = true;
    p[s] = -1;
    while (!q.empty()) {
        int v = q.front();
        q.pop();
        for (int u : adj[v]) {
            if (!used[u]) {
                used[u] = true;
                q.push(u);
                d[u] = d[v] + 1;
                p[u] = v;
}}}}
```

```
// python pseudocode(not tested)
distance = [-1 for _ in range(n + 1)]
distance[source] = 0
shortest_path_tree_parent = [-1 for _ in range(n + 1)]
queue = [source]
for u in queue:
  for v in adj[u]:
    if distance[v] == -1:
      distance[v] = distance[u] + 1
      shortest_path_tree_parent[v] = u
      queue.append(v)
```

2.2 Bellman-Ford

```
// Bellman-Ford

struct Edge {
    int a, b, cost;
};

int n, m, v;
vector<Edge> edges;
const int INF = 1000000000;

void solve(){
    vector<int> d(n, INF);
    d[v] = 0;
    for (int i = 0; i < n - 1; ++i) {
        bool any = false;

        for (Edge e : edges)
            if (d[e.a] < INF)
                if (d[e.b] > d[e.a] + e.cost) {
                    d[e.b] = d[e.a] + e.cost;
                    any = true;
                }

        if (!any)
            break;
    }
    // display d, for example, on the screen
}

// python pseudodcode (not tested)
distance = [float('inf') for _ in range(n + 1)]
for k in range(1, n):
  for u in range(1, n + 1):
    for v, w in adj[u]:
      distance[v] = min(distance[v], distance[u] + w)
for u in range(1, n + 1):
  for v, w in adj[u]:
    if distance[v] > distance[u] + w:
      report_negative_cycle()
```

2.3 DFS

```
// DFS
void dfs_comp(int n){
    vector<vector<int>> adj; // assumed nodes are 0 indexed

    vector<bool> visited;
    void dfs(int v) {
        visited[v] = true;
        for (int u : adj[v]) {
            if (!visited[u])
                dfs(u);
        }
    }
    for (int i=0; i<n; i++){
        if (not visited[i]) dfs(i);
    }
}

// python pseudocode (not tested)
component_index = [-1 for _ in range(n + 1)]
def dfs(u, c):
  component_index[u] = c
  for v in adj[u]:
    if component_index[v] == -1:
      dfs(v, c)
num_components = 0
for u in range(1, n + 1):
  if component_index[u] == -1:
    dfs(u, num_components)
    num_components += 1
```

2.4 Dijkstra

```
// Dijkstra

// For non-sparse graphs O(n^2 + m)

const int INF = INT_MAX;
vector<vector<pair<int, int>>> adj;

void dijkstra(int s, vector<int> & d, vector<int> & p) {
```

```
    int n = adj.size();
    d.assign(n, INF);
    p.assign(n, -1);

    d[s] = 0;
    // can be implemented using a priority queue with
        negative values but this works as well
    set<pair<int, int>> q;
    q.insert({0, s});
    while (!q.empty()) {
        int v = q.begin()->second;
        q.erase(q.begin());

        for (auto edge : adj[v]) {
            int to = edge.first;
            int len = edge.second;

            if (d[v] + len < d[to]) {
                q.erase({d[to], to});
                d[to] = d[v] + len;
                p[to] = v;
                q.insert({d[to], to});
}}}}

// python pseudocode(not tested)
import heapq
distance = [float('inf') for _ in range(n + 1)]
distance[source] = 0
shortest_path_tree_parent = [-1 for _ in range(n + 1)]
queue = [(distance[source], source)]
done = [False for _ in range(n + 1)]
while len(queue) > 0:
  _, u = heapq.heappop(queue)
  if not done[u]:
    for v, w in adj[u]:
      if distance[v] > distance[u] + w:
        distance[v] = distance[u] + w
        shortest_path_tree_parent[v] = u
        heapq.heappush((distance[v], v))
    done[u] = True
```

2.5 Floyd-Warshall

```
for (int k = 0; k < n; ++k) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}}}
// with negative edge weights
for (int k = 0; k < n; ++k) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (d[i][k] < INF && d[k][j] < INF)
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}}}
```

2.6 Kosaraju

```
vector<vector<int>> adj, adj_rev;
vector<bool> used;
vector<int> order, component;

void dfs1(int v) {
    used[v] = true;
    for (auto u : adj[v])
        if (!used[u])
            dfs1(u);
    order.push_back(v);
}

void dfs2(int v) {
    used[v] = true;
    component.push_back(v);
    for (auto u : adj_rev[v])
        if (!used[u])
            dfs2(u);
}

int main() {
    int n;
    for (;;) {
        int a, b;
        // ... read next directed edge (a,b) ...
        adj[a].push_back(b);
        adj_rev[b].push_back(a);
    }
    used.assign(n, false);

    for (int i = 0; i < n; i++)
        if (!used[i])
            dfs1(i);
    used.assign(n, false);
    reverse(order.begin(), order.end());

    for (auto v : order)
        if (!used[v]) {
            dfs2 (v);
            // ... processing next component ...
            component.clear();
    }
    vector<int> roots(n, 0);
    vector<int> root_nodes;
    vector<vector<int>> adj_scc(n);

    for (auto v : order)
        if (!used[v]) {
            dfs2(v);

            int root = component.front();
            for (auto u : component) roots[u] = root;
            root_nodes.push_back(root);

            component.clear();
        }


    for (int v = 0; v < n; v++){
        for (auto u : adj[v]) {
            int root_v = roots[v],
                root_u = roots[u];

            if (root_u != root_v)
                adj_scc[root_v].push_back(root_u);
    }}}
```

2.7 Kruskal's Algorithm

```
struct Edge {
    int u, v, weight;
    bool operator<(Edge const& other) {
        return weight < other.weight;
    }
};

int n;
vector<Edge> edges;

int cost = 0;
vector<int> tree_id(n);
vector<Edge> result;
for (int i = 0; i < n; i++)
    tree_id[i] = i;

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

for (Edge e : edges) {
```

```
    if (tree_id[e.u] != tree_id[e.v]) {
        cost += e.weight;
        result.push_back(e);

        int old_id = tree_id[e.u], new_id = tree_id[e.v];
        for (int i = 0; i < n; i++) {
            if (tree_id[i] == old_id)
                tree_id[i] = new_id;
}}}
```

2.8 Prim's Algorithm

```
// dense n^2
int n;
vector<vector<int>> adj; // adjacency matrix of graph
const int INF = 1000000000; // weight INF means there is no
     edge

struct Edge {
    int w = INF, to = -1;
};

void prim() {
    int total_weight = 0;
    vector<bool> selected(n, false);
    vector<Edge> min_e(n);
    min_e[0].w = 0;

    for (int i=0; i<n; ++i) {
        int v = -1;
        for (int j = 0; j < n; ++j) {
            if (!selected[j] && (v == -1 || min_e[j].w <
                 min_e[v].w))
                v = j;
        }

        if (min_e[v].w == INF) {
            cout << "No MST!" << endl;
            exit(0);
        }

        selected[v] = true;
        total_weight += min_e[v].w;
        if (min_e[v].to != -1)
            cout << v << " " << min_e[v].to << endl;

        for (int to = 0; to < n; ++to) {
            if (adj[v][to] < min_e[to].w)
                min_e[to] = {adj[v][to], v};
        }
    }

    cout << total_weight << endl;
}


// sparse mlogn
const int INF = 1000000000;

struct Edge {
    int w = INF, to = -1;
    bool operator<(Edge const& other) const {
        return make_pair(w, to) < make_pair(other.w, other.to
            );
    }
};

int n;
vector<vector<Edge>> adj;

void prim() {
    int total_weight = 0;
    vector<Edge> min_e(n);
    min_e[0].w = 0;
    set<Edge> q;
    q.insert({0, 0});
    vector<bool> selected(n, false);
    for (int i = 0; i < n; ++i) {
        if (q.empty()) {
            cout << "No MST!" << endl;
            exit(0);
        }

        int v = q.begin()->to;
        selected[v] = true;
        total_weight += q.begin()->w;
        q.erase(q.begin());

        if (min_e[v].to != -1)
            cout << v << " " << min_e[v].to << endl;

        for (Edge e : adj[v]) {
            if (!selected[e.to] && e.w < min_e[e.to].w) {
                q.erase({min_e[e.to].w, e.to});
                min_e[e.to] = {e.w, v};
                q.insert({e.w, e.to});
            }
        }
    }

    cout << total_weight << endl;
}
```

2.9 Toplogical Sort

```
int n; // number of vertices
vector<vector<int>> adj; // adjacency list of graph
vector<bool> visited;
vector<int> ans;

void dfs(int v) {
    visited[v] = true;
    for (int u : adj[v]) {
        if (!visited[u])dfs(u);}
    ans.push_back(v);
}

void topological_sort() {
    visited.assign(n, false);
    ans.clear();
    for (int i = 0; i < n; ++i) {
        if (!visited[i]) {
            dfs(i);
    }}
    reverse(ans.begin(), ans.end());
}

// python pseudocode
visited = [False for _ in range(n + 1)]
in_dfs_stack = [False for _ in range(n + 1)]
topologically_sorted = []
def toposort(u):
  in_dfs_stack[u] = True
  visited[u] = True
  for v in rev_adj[u]:
    if in_dfs_stack[v]:
      report_cycle()
    elif not visited[v]:
      toposort(v)
  topologically_sorted.append(u)
  in_dfs_stack[u] = False
for u in range(1, n + 1):
  if not visited[u]:
toposort(u)
```

3 Math

3.1 Chinese Remainder Theorem

```
struct Congruence {
    long long a, m;
};

long long chinese_remainder_theorem(vector<Congruence> const
     & congruences) {
    long long M = 1;
    for (auto const& congruence : congruences) {
        M *= congruence.m;
    }

    long long solution = 0;
    for (auto const& congruence : congruences) {
        long long a_i = congruence.a;
        long long M_i = M / congruence.m;
        long long N_i = mod_inv(M_i, congruence.m);
        solution = (solution + a_i * M_i % M * N_i) % M;
    }
    return solution;
}
```

3.2 Gauss-Jordan

```
const double EPS = 1e-9;
const int INF = 2; // it doesn't actually have to be
     infinity or a big number

int gauss (vector < vector<double> > a, vector<double> & ans
     ){
    int n = (int) a.size();
    int m = (int) a[0].size() - 1;

    vector<int> where (m, -1);
    for (int col=0, row=0; col<m && row<n; ++col) {
        int sel = row;
        for (int i=row; i<n; ++i)
            if (abs (a[i][col]) > abs (a[sel][col]))
                sel = i;
        if (abs (a[sel][col]) < EPS)
            continue;
        for (int i=col; i<=m; ++i)
            swap (a[sel][i], a[row][i]);
        where[col] = row;

        for (int i=0; i<n; ++i)
            if (i != row) {
                double c = a[i][col] / a[row][col];
                for (int j=col; j<=m; ++j)
                    a[i][j] -= a[row][j] * c;
            }
        ++row;
    }

    ans.assign (m, 0);
    for (int i=0; i<m; ++i)
        if (where[i] != -1)
            ans[i] = a[where[i]][m] / a[where[i]][i];
    for (int i=0; i<n; ++i) {
        double sum = 0;
        for (int j=0; j<m; ++j)
            sum += ans[j] * a[i][j];
        if (abs (sum - a[i][m]) > EPS)
            return 0;
    }

    for (int i=0; i<m; ++i)
        if (where[i] == -1)
            return INF;
    return 1;
}
```

3.3 Linear Diophantine

```
int gcd(int a, int b, int& x, int& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    int x1, y1;
    int d = gcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - y1 * (a / b);
    return d;
}

bool find_any_solution(int a, int b, int c, int &x0, int &y0
     , int &g) {
    g = gcd(abs(a), abs(b), x0, y0);
    if (c % g) {
        return false;
    }

    x0 *= c / g;
    y0 *= c / g;
    if (a < 0) x0 = -x0;
    if (b < 0) y0 = -y0;
    return true;
}
```

3.4 Modulo

```
ll mod(ll x, ll m){
    if (m == 0) return 0;
    if (m < 0) m*=-1;
    return (x%m + m)%m;
}

ll EEA(ll a, ll b, ll &x, ll &y){
    if (b==0){x = 1; y = 0; return a;}
    ll g = eea(b, a%b, x, y);
    ll z = x - a/b*y;
    x = y; y = z; return g;
}

ll modinv(ll a, ll m) {
    ll x, y; ll g = extended_euclid(a, m, x, y);
    if (g == 1 || g == -1) return mod(x * g, m);
    return 0; // 0 if invalid
}

pair<ll, ll> modsolve(ll a, ll b, ll m){
    ll x, y; ll g = eea(a, m, x, y);
    if (b % g != 0) return {-1, -1};
    return {mod(x*b/g, m/g), abs(m/g)};
}

ll fast_exp(ll b, ll e, ll m){
    ll res = 1;
    while (e > 0){
        if (e & 1) res = mod(res*b, m);
        b = mod(b*b, m);
        e >>= 1;
    }
    return res;
}
```

3.5 Primes

```
void primeSieve(){
```

```
    vector<bool> isPrime;
    vector<int> numFact;
    ll n;

    isPrime.resize(n, true);
    numFact.resize(n);

    isPrime[0] = isPrime[1] = false;
    for (ll i=2; i<n; i++){
        if (isPrime[i]){
            numFact[i] = 1;
            for (ll j=2*i; j<n; j+=i){
                isPrime[j] = false;
                numFact[j]++;
}}}}
```

4 Segment Trees

4.1 Coordinate Compressed

```
struct CompressedST {
  int n;
  vector<ll> st, lazy;

  // compressed information
  vector<pair<ll,ll>> lr;
  map<ll, int> compress;
  CompressedST(vector<ll> &c) {
    int sz = c.size();
    for (int i = 0; i < sz-1; i++) {
      compress[c[i]] = lr.size();
      lr.push_back({c[i], c[i]});
      if (c[i]+1 <= c[i+1]-1)
        lr.push_back({c[i]+1, c[i+1]-1});
    }
    compress[c[sz-1]] = lr.size();
    lr.push_back({c[sz-1], c[sz-1]});
    n = lr.size();

    st.assign(4*n, 0);
    lazy.assign(4*n, 0);
  }

  void pull(int p) {
    st[p] = st[p<<1] + st[p<<1|1];
  }

  void push(int p, int i, int j) {
    if (lazy[p]) {
      st[p] += (lr[j].second-lr[i].first+1)*lazy[p];
      if (i != j) {
        lazy[p<<1] += lazy[p];
        lazy[p<<1|1] += lazy[p];
      }
      lazy[p] = 0;
    }
  }

  void update(int l, int r, ll v, int p, int i, int j) {
    push(p, i, j);
    if (l <= i && j <= r) {
      lazy[p] += v;
      push(p, i, j);
    }
    else if (j < l || r < i);
    else {
      int k = (i+j)/2;
      update(l, r, v, p<<1, i, k);
      update(l, r, v, p<<1|1, k+1, j);
      pull(p);
    }
  }

  ll query(int l, int r, int p, int i, int j) {
    push(p, i, j);
    if (l <= i && j <= r) return st[p];
    else if (j < l || r < i) return 0;
    else {
      int k = (i+j)/2;
      return query(l, r, p<<1, i, k)
        + query(l, r, p<<1|1, k+1, j);
    }
  }

  ll query(ll l, ll r) {
    return query(compress[l], compress[r], 1, 0, n-1);
  }
  void update(ll l, ll r, ll v) {
    update(compress[l], compress[r], v, 1, 0, n-1);
  }
};
```

4.2 Range Update (Add and Set)

```
#define pb push_back
typedef long long ll;
typedef vector<int> vi;
struct segtree {
ll n, *vals, *adeltas, *sdeltas;
segtree(vi &ar) {
  n = ar.size();
  vals = new ll[4*n];
  adeltas = new ll[4*n];
  sdeltas = new ll[4*n];
  build(ar, 1, 0, n-1); }
// constructs segtree
void build(vi &ar, int p, int i, int j) {
  adeltas[p] = 0;
  sdeltas[p] = -1;
  if (i == j) vals[p] = ar[i];
  else {
  int k = (i + j) / 2;
  build(ar, p<<1, i, k);
  build(ar, p<<1|1, k+1, j);
  pull(p); } }
void pull(int p) { vals[p] = vals[p<<1] + vals[p<<1|1]; }
void push(int p, int i, int j) {
  if (sdeltas[p] != -1) {
    // cout << "went here" << endl;
    vals[p] = (j - i + 1) * sdeltas[p];
    if (i != j) {
      sdeltas[p<<1] = sdeltas[p];
      adeltas[p<<1] = 0;
      adeltas[p<<1|1] = 0;
      sdeltas[p<<1|1] = sdeltas[p]; }
    sdeltas[p] = -1; }
  if (adeltas[p]) {
    vals[p] += (j - i + 1) * adeltas[p];
    if (i != j) {
      adeltas[p<<1] += adeltas[p];
      adeltas[p<<1|1] += adeltas[p]; }
    adeltas[p] = 0; } }
void add_update(int _i, int _j, int v, int p, int i, int j)
     {
  push(p, i, j);
  if (_i <= i && j <= _j) {
     adeltas[p] += v;
    push(p, i, j);
  } else if (_j < i || j < _i) {
  // do nothing
  } else {
    int k = (i + j) / 2;
    add_update(_i, _j, v, p<<1, i, k);
    add_update(_i, _j, v, p<<1|1, k+1, j);
    pull(p); } }
void set_update(int _i, int _j, int v, int p, int i, int j)
     {
```

```
  push(p, i, j);
  if (_i <= i && j <= _j) {
    sdeltas[p] = v;
    adeltas[p] = 0;
    push(p, i, j);
  } else if (_j < i || j < _i) {
  // do nothing
  } else {
    int k = (i + j) / 2;
    set_update(_i, _j, v, p<<1, i, k);
    set_update(_i, _j, v, p<<1|1, k+1, j);
    pull(p); } }
ll query(int _i, int _j, int p, int i, int j) {
  push(p, i, j);
  if (_i <= i and j <= _j)
  return vals[p];
  else if (_j < i || j < _i)
  return 0;
  else {
  int k = (i + j) / 2;
  return query(_i, _j, p<<1, i, k) +
      query(_i, _j, p<<1|1, k+1, j); } } };
```

```
int n, q;
vi a;
int main(){
  cin >> n >> q;
  for (int i=0; i<n; i++){int x; cin >> x; a.pb(x);}
  segtree st(a);
  int t, b, c, d;
  for (int i=0; i<q; i++){
cin >> t;
    // add to segment
    if (t == 1){
      cin >> b >> c >> d;
      st.add_update(b-1, c-1, d, 1, 0, n-1);
      // cout << st.query(b-1, c-1, 1, 0, n-1) << endl;
}
    // // set each segment
    else if (t == 2){
      cin >> b >> c >> d;
      st.set_update(b-1, c-1, d, 1, 0, n-1);
      // cout << st.query(b-1, c-1, 1, 0, n-1) << endl;
}
    // // compute segment sum
    else if (t == 3){
      cin >> b >> c;;
      cout << st.query(b-1, c-1, 1, 0, n-1) << endl;
} }
return 0; }
```

5 template

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long int
using ld = long double
using ull = unsigned long long int

int main(){
    cin.tie(0) -> sync_with_stdio(0);

    return 0;
}
```