Team Notebook

December 9, 2021

Contents

1 ds-fenwick

```
// Fenwick Tree
//
// Dynamic data structure used to solve problems such as
// RSQ (Range Sum Queries) given a set of M integer keys
    that range
// from [1 ... N]. RangeFT allows range updates and queries
    in
// log N time.
//
// Functions:
// Class FenwickTree
// - rsq(a, b) returns the RSQ from a to b (inclusive).
// - adjust(N, v) add v to the Nth element.
// - q(N) returns Nth element.
// Class RangeFT
// - rsq(a, b) returns the RSQ from a to b (inclusive).
// - adjust(l, r, v) add v to elements from a to b (
    inclusive);
//
// Time complexities:
// - rsq: O(log N)
// - adjust: O(k log N)

#include <iostream>
#include <vector>

using namespace std;

template <class T>
class FenwickTree
{
    vector<T> ft;

public:
    FenwickTree(int n) : ft(n + 1)
    {
    }
    T rsq(int b)
    {
        T sum = 0;
        for (; b; b -= (b & (-b)))
            sum += ft[b];
        return sum;
    }
    T rsq(int a, int b)
    {
        return rsq(b) - rsq(a - 1);
    }
    T q(int x)
    {
        return rsq(x) - rsq(x - 1);
    }
    void adjust(int k, T v)
    {
        for (; k < ft.size(); k += (k & (-k)))
            ft[k] += v;
    }
};

template <class T>
class RangeFT
{
private:
    FenwickTree<T> mul, add;

public:
    RangeFT(int n) : mul(n), add(n)
    {
    }
    T rsq(int a)
    {
        return mul.rsq(a) * a + add.rsq(a);
    }
    T rsq(int a, int b)
    {
        return rsq(b) - rsq(a - 1);
    }
    void adjust(int l, int r, T v)
    {
        mul.adjust(l, v);
        mul.adjust(r, -v);
        add.adjust(l, -v * (l - 1));
        add.adjust(r, v * r);
    }
};

int main()
{
    RangeFT<int> f(100);

    f.adjust(1, 50, 10);

    cout << f.rsq(10, 20) << endl; // 110

    return 0;
}
```

2 ds-segtree-basic

```
// Segment Tree
//
// Dynamic data structure used to answer dynamic range
    queries such as
// RSQ (Range Sum Query), RMQ (Range Minimum Query) and
    others.
// This implementation solves the RMQ problem given a set of
     N integers.
//
// Functions:
// - query(i, j) returns the RMQ on [i, j].
// - update(i, v) updates the i-th element to the value v.
//
// Time complexities:
// - build: O(N)
// - query: O(log N)
// - update: O(log N)

#include <iostream>

using namespace std;

const int MAXN = 1 << 18; // 2.6e5
int N = 200000;
int seg[2 * MAXN]; // root is seg[1]

void build()
{
    for (int i = N - 1; i >= 0; i--)
        seg[i] = seg[i << 1] + seg[i << 1 | 1];
}

void update(int p, int val)
{
    for (seg[p += N] = val; p > 0; p >>= 1)
        seg[p >> 1] = seg[p] + seg[p ^ 1];
}

int query(int l, int r) // [l, r]
{
    int res = 0;
    for (l += N, r += N; l <= r; l >>= 1, r >>= 1)
    {
        if (l & 1)
            res += seg[l++];
        if (!(r & 1))
            res += seg[r--];
    }
```

```
    return res;
}

int main()
{
    for (int i = 1; i <= 100; ++i)
        seg[i + N] = i;
    build();

    cout << query(2, 5) << endl;

    return 0;
}
```

3 ds-segtree-lazy

```
#include <iostream>
#include <vector>
#include <cmath>
#include <cstring>
using namespace std;

class SegTree
{
    vector<int> tree;
    vector<int> lazy;
    int n;

    void build_tree(const vector<int> &v, int node, int a,
         int b)
    {
        if (a > b)
            return;

        if (a == b)
        {
            tree[node] = v[a];
            return;
        }

        build_tree(v, node * 2, a, (a + b) / 2);
        build_tree(v, node * 2 + 1, 1 + (a + b) / 2, b);
        tree[node] = min(tree[node * 2], tree[node * 2 + 1]);
    }

    void update_lazy(int node, int a, int b)
    {
        tree[node] += lazy[node];

        if (a != b)
        {
            lazy[node * 2] += lazy[node];
            lazy[node * 2 + 1] += lazy[node];
        }

        lazy[node] = 0;
    }

    void update_tree(int node, int a, int b, int i, int j,
         int value)
    {
        if (lazy[node] != 0)
            update_lazy(node, a, b);

        if (a > b || a > j || b < i)
            return;

        if (a >= i && b <= j)
        {
            tree[node] += value;
            if (a != b)
            {
                lazy[node * 2] += value;
                lazy[node * 2 + 1] += value;
            }
            return;
        }

        update_tree(node * 2, a, (a + b) / 2, i, j, value);
        update_tree(1 + node * 2, 1 + (a + b) / 2, b, i, j,
             value);

        tree[node] = min(tree[node * 2], tree[node * 2 + 1]);
    }

    int query_tree(int node, int a, int b, int i, int j)
    {
        if (a > b || a > j || b < i)
            return 2100000000;

        if (lazy[node] != 0)
            update_lazy(node, a, b);

        if (a >= i && b <= j)
            return tree[node];

        int q1 = query_tree(node * 2, a, (a + b) / 2, i, j);
        int q2 = query_tree(1 + node * 2, 1 + (a + b) / 2, b,
             i, j);

        return min(q1, q2);
    }

public:
    SegTree(const vector<int> &v)
    {
        n = v.size();

        int s = 2 * pow(2, ceil(log2(v.size())));

        tree.resize(s);
        lazy.resize(s);

        build_tree(v, 1, 0, n - 1);
    }

    void update(int idx1, int idx2, int add)
    {
        update_tree(1, 0, n - 1, idx1, idx2, add);
    }

    int query(int idx1, int idx2)
    {
        return query_tree(1, 0, n - 1, idx1, idx2);
    }
};

int main()
{
    vector<int> a = {1, 2, 3, 4, 5};
    SegTree s(a);

    s.update(1, 3, 10);

    for (int i = 0; i < a.size(); ++i)
        cout << s.query(i, i) << " ";
    cout << endl;

    return 0;
}
```

4 graphs-bellman-ford

```
// Bellman Ford's algorithm
//
```

```
// Calculates SSSP (single source shortest path) on a
     weighted graph and
// detects if a graph contains a negative cycle.
//
// Variables:
// - G is the edge list of the graph.
// - s is the source.
// - dist is the vector of distances from the source to
     other vertices.
// - V, E are the number of vertices and edges of the graph.
//
// Functions:
// - BellmanFord() stores lengths of shortest paths in dist
     and returns true
// if there exists a negative weight cycle.
//
// Time complexity: O(VE)

#include <iostream>
#include <vector>
#include <tuple>

using namespace std;

#define inf 1e9

typedef tuple<int, int, int> iii;
typedef vector<iii> viii;
typedef vector<int> vi;

int s, V, E;
viii G;
vi dist;

bool BellmanFord()
{
    dist.assign(V, inf);
    dist[s] = 0;
    int u, v, w;
    for (int i = 0; i < V - 1; ++i)
        for (iii e : G)
            tie(u, v, w) = e, dist[v] = min(dist[v], dist[u]
                 + w);

    for (iii e : G)
    {
        tie(u, v, w) = e;
        if (dist[v] > dist[u] + w)
            return true;
    }
    return false;
}

int main()
{
    G.clear();
    cin >> V >> E;
    for (int i = 0; i < E; ++i)
    {
        int u, v, w;
        cin >> u >> v >> w;
        G.emplace_back(u, v, w);
    }

    BellmanFord();

    return 0;
}
```

5 graphs-bfs

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

int main()
{
    vector<vector<int>> adj = {
        // adjacency list representation
        {1},
        {2},
        {0},
    };
    int n = 3;       // number of nodes
    int s = 0, u = 2; // source vertex, end vertex

    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;
            }
        }
    }

    if (!used[u])
    {
        cout << "No path!";
    }
    else
    {
        vector<int> path;
        for (int v = u; v != -1; v = p[v])
            path.push_back(v);
        reverse(path.begin(), path.end());
        cout << "Path: ";
        for (int v : path)
            cout << v << " ";
    }

    return 0;
}
```

6 graphs-dfs

```
#include <iostream>
#include <vector>

using namespace std;

vector<vector<int>> adj = {
    // adjacency list representation
    {1},
    {2},
    {0},
};
int n = 3; // number of vertices

vector<bool> visited(n);
```

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

// vector<int> color(n);

// vector<int> time_in(n), time_out(n);
// int dfs_timer = 0;

// void dfs(int v)
// {
//     time_in[v] = dfs_timer++;
//     color[v] = 1;
//     for (int u : adj[v])
//         if (color[u] == 0)
//             dfs(u);
//     color[v] = 2;
//     time_out[v] = dfs_timer++;
// }

int main()
{
    dfs(0);

    return 0;
}
```

7 graphs-dijkstra

```
// Dijkstra's algorithm
//
// Calculates SSSP (single source shortest path) on a
    weighted graph.
//
// Variables:
// - V, E are the number of vertices and edges of the graph
// - s is the source
// - AdjList is the adjacency list of the graph
// - dist is the vector of distances from the vertices to
    the source
//
// Time complexity: O((V + E)logV)

#include <functional>
#include <iostream>
#include <queue>
#include <tuple>
#include <vector>
using namespace std;

typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
typedef vector<vii> vvii;
#define INF 1000000000

void dijkstra(vvii &AdjList, vi &dist, int s)
{
    priority_queue<ii, vii, greater<ii>> pq;
    pq.emplace(0, s);

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

        if (d > dist[u])
            continue;

        for (ii v : AdjList[u])
        {
            if (dist[u] + v.second < dist[v.first])
            {
                dist[v.first] = dist[u] + v.second;
                pq.emplace(dist[v.first], v.first);
            }
        }
    }
}

int main()
{
    int V, E, s, u, v, w;
    vvii AdjList;
    cin >> V >> E >> s;

    AdjList.assign(V, vii());
    for (int i = 0; i < E; ++i)
    {
        cin >> u >> v >> w;
        AdjList[u].emplace_back(v, w); // directed graph
        // AdjList[v].emplace_back(u, w);
    }

    vi dist(V, INF);
    dist[s] = 0;
    dijkstra(AdjList, dist, s);

    // use dist

    return 0;
}
```

8 graphs-floyd-warshall

```
// Floyd Warshall's algorithm
//
// Calculates APSP (all-pairs shortest path) on a weighted
    graph.
//
// Variables:
// - G is the adjecency matrix of the graph.
// - N is the number of vertices of the graph.
// - MAX is the maximum numbers of vertices (needs to be
    configurated).
//
// Note: G[i][j] represents the shortest path from vertex i
    to vertex j.
// If G[i][j] = inf then vertex j is unreachable from vertex
    i.

#include <iostream>

using namespace std;

const int inf = 1e9;

const int MAX = 10000;

int G[MAX][MAX] = {
    {0, inf, 3},
    {1, 0, inf},
    {2, 2, 0},
},
    N = 3;

int main()
{
```

```
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            G[i][j] = (i == j ? 0 : G[i][j]); // : inf);
                 usually

    for (int k = 0; k < N; ++k)
        for (int i = 0; i < N; ++i)
            for (int j = 0; j < N; ++j)
                G[i][j] = min(G[i][j], G[i][k] + G[k][j]);

    for (int i = 0; i < N; ++i)
    {
        for (int j = 0; j < N; ++j)
            cout << G[i][j] << " ";
        cout << endl;
    }

    return 0;
}
```

9 graphs-kruskal-mst

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

struct Edge
{
    int u, v, weight;
} typedef Edge;

bool operator<(const Edge &a, const Edge &b)
{
    return a.weight < b.weight;
}

int main()
{
    int n = 3;
    vector<Edge> edges = {{1, 2, 1}, {1, 2, 3}, {2, 3, 1}};

    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;
            }
        }
    }

    for (auto i : result)
        cout << i.u << " " << i.v << endl;

    return 0;
}
```

10 graphs-lca-misc

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

using ll = long long;

const int MAX = 100000;
const int LG = 17;
ll length[MAX];
vector<int> adj[MAX] = {
    {1, 2},
    {3},
    {4},
    {},
    {},
};

int jump[MAX][LG];
int depth[MAX];
ll dist[MAX];

int tick;
int discovery[MAX];
int finish[MAX];

void dfs(int x, int p)
{
    jump[x][0] = (p == -1 ? x : p);
    for (int i = 1; i < LG; ++i)
        jump[x][i] = jump[jump[x][i - 1]][i - 1];

    discovery[x] = tick++;
    for (int y : adj[x])
    {
        if (y == p)
            continue;
        depth[y] = depth[x] + 1;
        dist[y] = dist[x] + length[y];
        dfs(y, x);
    }
    finish[x] = tick++;
}

bool is_ancestor(int x, int y)
{
    return discovery[x] <= discovery[y] && finish[y] <=
         finish[x];
}

int lca(int x, int y)
{
    if (is_ancestor(x, y))
        return x;
    for (int i = LG - 1; i >= 0; --i)
        if (!is_ancestor(jump[x][i], y))
            x = jump[x][i];
    return jump[x][0];
}

ll distance(int x, int y)
{
    return dist[x] + dist[y] - 2 * dist[lca(x, y)];
}

int num_edges(int x, int y)
{
    return depth[x] + depth[y] - 2 * depth[lca(x, y)];
}

int main()
{
    dfs(0, -1);
```

```
    cout << lca(3, 4) << endl;    // 0
    cout << distance(3, 4) << endl; // 0
    cout << num_edges(3, 4) << endl; // 4

    return 0;
}
```

11 graphs-tarjan

```
// Tarjan Algorithm for SCC
//
// Finds SCC(Strongly Connected Components)
//
// Variables:
// - G is the adjecency list of the graph.
// - V is the number of vertices in the graph.
// - discovery[x] is true if x has been visited.
// - SCC is the number of strongly connected components.
// - S is a temporary "stack" (vector) that holds the
     current SCC.
//
// Time complexity: O(V + E)

#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

typedef vector<int> vi;
typedef vector<vi> vvi;

vvi G;
vi discovery, num, low, S;
int SCC, tick, V;

void tarjan(int u)
{
    low[u] = num[u] = tick++;
    S.push_back(u);
    discovery[u] = 1;
    for (int v : G[u])
    {
        if (num[v] == -1)
            tarjan(v);

        if (discovery[v])
            low[u] = min(low[u], low[v]);
    }

    if (low[u] == num[u])
    {
        SCC++;
        while (true)
        {
            int v = S.back();
            S.pop_back();
            discovery[v] = 0;
            // cout << v << '\n';
            if (u == v)
                break;
        }
    }
}

int main()
{
    G.assign(V, vi());
    SCC = tick = 0;
    num.assign(V, -1);
    low.assign(V, 0);
    discovery.assign(V, 0);

    for (int i = 0; i < V; ++i)
        if (num[i] == -1)
            tarjan(i);

    return 0;
}
```

12 graphs-topsort

```
// Topological Sort on DAG
//
// Prints out one topological sort for a given DAG (Directed
      Acyclic Graph).
//
// Variables:
// - G is the adjecency list of the graph.
// - P[x] is true if vertex x has been visited.
// - Sorted is the deque of topologicaly ordered vertices.
// - V, E are the number of vertices and edges in the graph.
//
// Functions:
// - dfs(D, u) does a dfs from vertex u.
//
// Time complexity: O(E)

#include <iostream>
#include <vector>
#include <deque>

using namespace std;

typedef vector<int> vi;
typedef vector<vi> vvi;

vvi G;
vi P;

int dfs(deque<int> &D, int u)
{
    P[u] = 1;
    for (int v : G[u])
        if (!P[v])
            dfs(D, v);

    D.push_front(u);
}

int main()
{
    int V, E;
    G.assign(V, vi());
    cin >> V >> E;
    for (int i = 0; i < E; ++i)
    {
        int u, v;
        cin >> u >> v;
        G[u].push_back(v);
    }

    deque<int> Sorted;
    P.assign(V, 0);
    for (int i = 0; i < V; ++i)
        if (!P[i])
            dfs(Sorted, i);

    for (int i : Sorted)
        cout << i << ' ';

    return 0;
}
```

13 graphs-unionfind

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

const int maxn = 200005;
int p[maxn];

void make_set()
{
    for (int i = 0; i < maxn; ++i)
        p[i] = i;
}

int find_set(int v)
{
    if (v == p[v])
        return v;
    return p[v] = find_set(p[v]);
}

void union_sets(int a, int b)
{
    a = find_set(a);
    b = find_set(b);
    if (a != b)
        p[b] = a;
}

int main()
{
    make_set();
    union_sets(1, 2);
    union_sets(3, 4);

    cout << find_set(1) << " " << find_set(3) << endl;

    union_sets(2, 4);

    cout << find_set(1) << " " << find_set(3) << endl;

    return 0;
}
```

14 math-binary

```
#include <iostream>
#include <vector>

using namespace std;

int bsrch_it(vector<int> &A, int p, int q, int x)
{
    while (p < q)
    {
        int mid = (p + q) / 2;
        if (A[mid] == x)
            return mid;
        else if (x < A[mid])
            q = mid - 1;
        else
            p = mid + 1;
    }
    return -1;
}

int a[] = {1, 1, 1, 0, 0, 0};

int f(int x)
{
    return a[x];
}

//          o
// 0 0 0 0 0 1 1 1 1 1
template <typename func>
int bsrch_a(func f, int lo, int hi)
{
    while (lo < hi)
    {
        int mid = (lo + hi) / 2; // pod
        if (f(mid))
            hi = mid;
        else
            lo = mid + 1;
    }
    return lo;
}

//         o
// 1 1 1 1 1 0 0 0 0 0 0
template <typename func>
int bsrch_b(func f, int lo, int hi)
{
    while (lo < hi)
    {
        int mid = (lo + hi + 1) / 2; // strop
        if (f(mid))
            lo = mid;
        else
            hi = mid - 1;
    }
    return lo;
}

int main()
{
    cout << bsrch_b(f, 0, 5) << endl;
    return 0;
}
```

15 math-crt

```
# Python 3.6
from functools import reduce


def chinese_remainder(m, a):
    sum = 0
    prod = reduce(lambda acc, b: acc*b, m)
    for m_i, a_i in zip(m, a):
        p = prod // m_i
        sum += a_i * mul_inv(p, m_i) * p
    return sum % prod


def mul_inv(a, b):
    b0 = b
    x0, x1 = 0, 1
    if b == 1:
        return 1
    while a > 1:
        q = a // b
        a, b = b, a % b
        x0, x1 = x1 - q * x0, x0
    if x1 < 0:
        x1 += b0
    return x1


# x     1 (mod 3)
# x     1 (mod 4)
# x     0 (mod 7)
if __name__ == '__main__':
    m = [3, 4, 7]
```

```
    a = [1, 1, 0]
    print(chinese_remainder(m, a)) # 49
```

16 math-discrete-log

```cpp
#include <iostream>
#include <cmath>
#include <unordered_map>

using namespace std;

int gcd(int a, int b) // basic gcd
{
    return (b == 0 ? a : gcd(b, a % b));
}

// Returns minimum x for which a ^ x % m = b % m.
int discrete_log(int a, int b, int m)
{
    a %= m, b %= m;

    int k = 1, add = 0, g;
    while ((g = gcd(a, m)) > 1)
    {
        if (b == k)
            return add;
        if (b % g)
            return -1;
        b /= g, m /= g, ++add;
        k = (k * 1ll * a / g) % m;
    }

    int n = sqrt(m) + 1;
    int an = 1;
    for (int i = 0; i < n; ++i)
        an = (an * 1ll * a) % m;

    unordered_map<int, int> vals;
    for (int q = 0, cur = b; q <= n; ++q)
    {
        vals[cur] = q;
        cur = (cur * 1ll * a) % m;
    }

    for (int p = 1, cur = k; p <= n; ++p)
    {
        cur = (cur * 1ll * an) % m;
        if (vals.count(cur))
        {
            int ans = n * p - vals[cur] + add;
            return ans;
        }
    }
    return -1;
}

int main()
{
    cout << discrete_log(2, 7, 13) << endl; // 11, 2^11 = 7
        mod 13
    return 0;
}
```

17 math-euclid-gcd-and-extended

```cpp
// Extended Euclid Algorithm
//
// Solves the Linear Diophantine Equation a * x + b * y = d
     where
// d = gcd(a, b).
//
// Time complexity: O(log(min(a, b)))

#include <iostream>

using namespace std;

typedef long long ll;

ll gcd(ll a, ll b) // basic gcd
{
    return (b == 0 ? a : gcd(b, a % b));
}

int x, y, d;

void Euclid(ll a, ll b)
{
    if (b == 0)
        x = 1, y = 0, d = a;
    else
    {
        Euclid(b, a % b);
        int x1 = y, y1 = x - (a / b) * y;
        x = x1, y = y1;
    }
}

int main()
{
    Euclid(10, 11);
    cout << x << " " << y << endl;

    return 0;
}
```

18 math-factor

```cpp
#include <iostream>
#include <vector>
#include <array>
using namespace std;

vector<long long> trial_division1(long long n)
{
    vector<long long> factorization;
    for (long long d = 2; d * d <= n; d++)
    {
        while (n % d == 0)
        {
            factorization.push_back(d);
            n /= d;
        }
    }
    if (n > 1)
        factorization.push_back(n);
    return factorization;
}

vector<long long> trial_division3(long long n) // 1/3
     constant
{
    vector<long long> factorization;
    for (int d : {2, 3, 5})
    {
        while (n % d == 0)
        {
            factorization.push_back(d);
            n /= d;
        }
    }
    static array<int, 8> increments = {4, 2, 4, 2, 4, 6, 2,
         6};
    int i = 0;
```

```
    for (long long d = 7; d * d <= n; d += increments[i++])
    {
        while (n % d == 0)
        {
            factorization.push_back(d);
            n /= d;
        }
        if (i == 8)
            i = 0;
    }
    if (n > 1)
        factorization.push_back(n);
    return factorization;
}

int main()
{
    for (int i : trial_division3(341768312))
        cout << i << " ";
    cout << endl;
    return 0;
}
```

19 math-gauss

```
// Gaussian Elimination
//
// Solves system of linear equations Ax = b.
//
// Variables:
// - N dimension of the system.
// - A is the augmented matrix (A = [A, b]).
//
// Time complexity: O(N ^ 3)

#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

typedef vector<double> vd;
typedef vector<vd> vvd;

vd GaussianElimination(int N, vvd &A)
{
    double t;
    vd X(N);
    for (int j = 0; j < N - 1; ++j)
    {
        int l = j;
        for (int i = j + 1; i < N; ++i)
            if (fabs(A[i][j]) > fabs(A[l][j]))
                l = i;

        for (int k = j; k <= N; ++k)
            t = A[j][k], A[j][k] = A[l][k], A[l][k] = t;

        for (int i = j + 1; i < N; ++i)
            for (int k = N; k >= j; --k)
                A[i][k] -= A[j][k] * A[i][j] / A[j][j];
    }

    for (int j = N - 1; j >= 0; --j)
    {
        t = 0;
        for (int k = j + 1; k < N; ++k)
            t += A[j][k] * X[k];

        X[j] = (A[j][N] - t) / A[j][j];
    }

    return X;
}

int main()
{
    int n;
    cin >> n;
    vvd A(n, vd(n + 1));
    for (int i = 0; i < n; ++i)
        for (int j = 0; j <= n; ++j)
            cin >> A[i][j];

    vd X = GaussianElimination(n, A);
    for (int i = 0; i < n; ++i)
        cout << X[i] << endl;

    return 0;
}
```

20 math-matrix

```
// Matrix multiplication and exponentiation
//
// Fast matrix exponentiation and multiplication.
//
// Attributes:
// - dim is the dimension of the matrix.
// - A is the vector representing the matrix.
//
// Time complexities:
// - operator *: O(N ^ 3)
// - operator ^: O(N ^ 3 * log k)

#include <vector>
#include <iostream>
#include <numeric>
using namespace std;

typedef long long ll;
typedef vector<int> vi;
typedef vector<vi> vvi;

const int mod = 1000000007;
int add(int a, int b) { return (a += b) < mod ? a : a - mod;
    }
int mul(int a, int b) { return 1LL * a * b % mod; }
void adds(int &a, int b) { a = add(a, b); }

int pwr(int a, ll p)
{
    if (p == 0)
        return 1;
    if (p & 1)
        return mul(a, pwr(a, p - 1));
    return pwr(mul(a, a), p / 2);
}

int inv(int a) { return pwr(a, mod - 2); }

struct Matrix
{
    vvi A;
    Matrix() {}
    Matrix(int n) { A.assign(n, vi(n)); }
    vi &operator[](int n) { return A[n]; }
};

int operator*(vi &a, vi &b)
{
    return inner_product(a.begin(), a.end(), b.begin(), 0,
        add, mul);
}

Matrix operator*(Matrix A, Matrix B)
```

```
{
    int n = A.A.size();
    Matrix C(n);
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            for (int k = 0; k < n; ++k)
                adds(C[i][j], mul(A[i][k], B[k][j]));

    return C;
}

Matrix operator^(Matrix A, ll k)
{
    int n = A.A.size();
    Matrix R(n);
    for (int i = 0; i < n; ++i)
        R[i][i] = 1;

    while (k > 0)
    {
        if (k % 2)
            R = R * A;

        A = A * A;
        k /= 2;
    }
    return R;
}

int main()
{
    Matrix a;
    a.A = {
        {0, 1},
        {1, 1},
    };

    vector<int> v = {0, 1};

    a = (a ^ 10);

    cout << a[0][1] * v[1] + a[1][1] * v[1] << endl; // 144,
        fibonacci

    return 0;
}
```

21 math-mod

```
// Binomial coefficient
//
// Calculating ncr (n choose r).
//
// Variables:
// - mod is the modulo
// - maxn is the largest possible n
// - f[n] = n!
// - fi[n] = inverse of n! modulo mod
//
// Functions:
// - pwr(a, p) calculates a ^ k % mod
// - inv(a) calculates inverse of a modulo mod
// - precompute() precomputes factorials and inverse
    factorials up to maxn
// - ncr_single(n, r) calculates n choose r
//
// Time complexities:
// - pwr: O(log(p))
// - inv: O(log(mod))
// - ncr: O(1)
// - precompute: O(maxn)
//
// Note: before using ncr call precompute() to precompute
    factorials

#include <iostream>
#include <cstring>
using namespace std;
typedef long long ll;

const int mod = 1000000009;
const int maxn = 1000000;

int f[maxn + 1], fi[maxn + 1];

int add(int a, int b) { return (a += b) < mod ? a : a - mod;
    }
int sub(int a, int b) { return (a -= b) >= 0 ? a : a + mod;
    }
int mul(int a, int b) { return 1LL * a * b % mod; }
void adds(int &a, int b) { a = add(a, b); }
void subs(int &a, int b) { a = sub(a, b); }
void muls(int &a, int b) { a = mul(a, b); }
void maxs(int &a, int b) { a = max(a, b); }
void mins(int &a, int b) { a = min(a, b); }

int pwr(int a, ll p)
{
    if (p == 0)
        return 1;
    if (p & 1)
        return mul(a, pwr(a, p - 1));
    return pwr(mul(a, a), p / 2);
}

int inv(int a) { return pwr(a, mod - 2); }
int ncr_single(int n, int r) { return mul(f[n], mul(fi[r],
     fi[n - r])); }

void precompute()
{
    f[0] = 1;
    for (int i = 1; i <= maxn; ++i)
        f[i] = mul(f[i - 1], i);

    fi[maxn] = inv(f[maxn]);
    for (int i = maxn - 1; i >= 0; --i)
        fi[i] = mul(fi[i + 1], i + 1);
}

int ncr_dp[2500][2500];

int ncr(int n, int r)
{
    if (r == 0)
        return 1;
    if (n == r)
        return 1;
    if (ncr_dp[n][r] != -1)
        return ncr_dp[n][r];
    return ncr_dp[n][r] = ncr(n - 1, r) + ncr(n - 1, r - 1);
}

int main()
{
    precompute();

    cout << inv(1231241) << endl; // 603580458
    // 603580458 * 1231241 = 743153006688378, mod = 1

    memset(ncr_dp, -1, sizeof ncr_dp);

    cout << ncr(5, 2) << endl; // 10

    return 0;
}
```

22 math-newton

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

// x_n = x_{n-1} - f(x_{n-1}) / f'(x_{n-1})

double sqrt_newton(double n)
{
    const double eps = 1E-15;
    double x = 1;
    for (;;)
    {
        double nx = (x + n / x) / 2;
        if (abs(x - nx) < eps)
            break;
        x = nx;
    }
    return x;
}

int isqrt_newton(int n)
{
    int x = 1;
    bool decreased = false;
    for (;;)
    {
        int nx = (x + n / x) >> 1;
        if (x == nx || nx > x && decreased)
            break;
        decreased = nx < x;
        x = nx;
    }
    return x;
}

int main()
{
    cout << sqrt_newton(17) << endl;
    return 0;
}
```

23 math-prim-test

```
#include <iostream>
using namespace std;
using u64 = long long;
using u128 = __uint128_t;

u64 binpower(u64 base, u64 e, u64 mod)
{
    u64 result = 1;
    base %= mod;
    while (e)
    {
        if (e & 1)
            result = (u128)result * base % mod;
        base = (u128)base * base % mod;
        e >>= 1;
    }
    return result;
}

bool isPrime(int x)
{
    for (int d = 2; d * d <= x; d++)
    {
        if (x % d == 0)
            return false;
    }
    return true;
}

bool probablyPrimeFermat(int n, int iter = 5)
{
    if (n < 4)
        return n == 2 || n == 3;

    for (int i = 0; i < iter; i++)
    {
        int a = 2 + rand() % (n - 3);
        if (binpower(a, n - 1, n) != 1)
            return false;
    }
    return true;
}

bool check_composite(u64 n, u64 a, u64 d, int s)
{
    u64 x = binpower(a, d, n);
    if (x == 1 || x == n - 1)
        return false;
    for (int r = 1; r < s; r++)
    {
        x = (u128)x * x % n;
        if (x == n - 1)
            return false;
    }
    return true;
};

bool MillerRabin(u64 n) // deterministic
{                   // returns true if n is prime, else
     returns false.
    if (n < 2)
        return false;

    int r = 0;
    u64 d = n - 1;
    while ((d & 1) == 0)
    {
        d >>= 1;
        r++;
    }

    for (int a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
         37})
    {
        if (n == a)
            return true;
        if (check_composite(n, a, d, r))
            return false;
    }
    return true;
}

int main()
{
    cout << MillerRabin(1000000000091) << endl;
    return 0;
}
```

24 math-sieve

```
// Sieve of Eratosthenes
//
// Computes all primes less than a given bound
//
// Variables:
// - P is a list of all primes less than a given bound
// - maxn is the upper bound of primes
// - n is the number of primes less than the upper bound
// - maxN ~ number of primes less than maxn
//
```

```
// Functions:
// - SimpleSieve() stores all primes less than sqrt(maxn)
// - SegSieve() stores all primes less than maxn
//
// Time complexity: O(N(log(log N)))
// Space complexity:
// - O(N / log(N)) - if all primes are stored
// - O(sqrt(N)) - if only first sqrt(N) primes are stored

#include <iostream>
#include <cstring>
#include <cmath>
#include <vector>

using namespace std;

typedef long long ll;

const ll maxN = 60000000;
const ll maxn = 100000000;
constexpr ll fn = 10002; //(ll)(sqrtl(maxn)) + 2;

ll Prime[fn],
    IsNotPrime[fn], p, n, P[maxN];

void SimpleSieve()
{
    for (ll i = 2; i < fn; ++i)
    {
        if (!IsNotPrime[i])
        {
            for (ll j = i * i; j < fn; j += i)
                IsNotPrime[j] = 1;

            P[n++] = Prime[p++] = i;
        }
    }
}

void SegSieve() // upto n
{
    SimpleSieve();
    ll lo = fn, hi = 2 * fn;
    while (lo < maxn)
    {
        if (hi >= maxn)
            hi = maxn;

        memset(IsNotPrime, 0, sizeof(IsNotPrime));
        for (ll i = 0; i < p; ++i)
        {
            int lolim = ll(lo / Prime[i]) * Prime[i];
            if (lolim < lo)
                lolim += Prime[i];

            for (ll j = lolim; j < hi; j += Prime[i])
                IsNotPrime[j - lo] = 1;
        }

        for (ll i = lo; i < hi; ++i)
            if (!IsNotPrime[i - lo])
                P[n++] = i;

        lo += fn, hi += fn;
    }
}

vector<char> segmentedSieve(long long L, long long R)
{
    // generate all primes up to sqrt(R)
    long long lim = sqrt(R);
    vector<char> mark(lim + 1, false);
    vector<long long> primes;
    for (long long i = 2; i <= lim; ++i)
    {
        if (!mark[i])
        {
            primes.emplace_back(i);
            for (long long j = i * i; j <= lim; j += i)
                mark[j] = true;
        }
    }

    vector<char> isPrime(R - L + 1, true);
    for (long long i : primes)
        for (long long j = max(i * i, (L + i - 1) / i * i); j
            <= R; j += i)
            isPrime[j - L] = false;
    if (L == 1)
        isPrime[0] = false;
    return isPrime;
}

int main()
{
    auto a = segmentedSieve(1e12, 1e12 + 100);

    for (ll i = 1e12; i < 1e12 + 100; ++i)
        if (a[i - 1e12])
            cout << i << " ";
    cout << endl;

    return 0;
}
```

25 math-ternary

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

double f(double x)
{
    return -((x - 2.3) * (x - 2.3)) + 5;
}

double ternary_search(double l, double r)
{
    double eps = 1e-9; // set the error limit here
    while (r - l > eps)
    {
        double m1 = l + (r - l) / 3;
        double m2 = r - (r - l) / 3;
        double f1 = f(m1); // evaluates the function at m1
        double f2 = f(m2); // evaluates the function at m2
        if (f1 < f2)
            l = m1;
        else
            r = m2;
    }
    return f(l); // return the maximum of f(x) in [l, r]
}

int main()
{
    cout << ternary_search(0, 5) << endl;
}
```

26 string-hash

```
#include <iostream>
#include <vector>
using namespace std;
using ll = long long;

const int mod = 1e9 + 9;
```

```
int add(int a, int b) { return (a += b) < mod ? a : a - mod;
    }
int sub(int a, int b) { return (a -= b) >= 0 ? a : a + mod;
    }
int mul(int a, int b) { return 1LL * a * b % mod; }
int pwr(int a, int p)
{
    if (p == 0)
        return 1;
    if (p & 1)
        return mul(a, pwr(a, p - 1));
    return pwr(mul(a, a), p / 2);
}
int inv(int a) { return pwr(a, mod - 2); }

const int maxn = 100000;
int invp[maxn], ppow[maxn], pref[maxn + 1];

void pre_hash(const string &s)
{
    for (int i = 0; i < s.size(); ++i)
        pref[i + 1] = add(pref[i], mul(s[i] - 'a' + 1, ppow[i
            ]));
}

int substr_hash(int i, int j)
{
    return mul(invp[i], sub(pref[j + 1], pref[i]));
}

int hash_f(const string &s)
{
    ll p = 1, h = 0;
    for (int i = 0; i < s.size(); ++i)
        h = add(h, mul(s[i] - 'a' + 1, ppow[i]));
    return h;
}

vector<int> rabin_karp(string const &s, string const &t)
{
    pre_hash(t);
    ll h_s = hash_f(s);
    vector<int> occurences;
    for (int i = 0; i + s.size() - 1 < t.size(); i++)
        if (substr_hash(i, i + s.size() - 1) == h_s)
            occurences.push_back(i);
    return occurences;
}

int LCP(const string &s, int x0, int y0, int x1, int y1)
{
    if (s[x0] != s[x1])
        return 0;

    int lo = 1, hi = min(y0 - x0 + 1, y1 - x1 + 1);
    while (lo < hi)
    {
        int mid = (lo + hi + 1) / 2;
        if (substr_hash(x0, x0 + mid - 1) == substr_hash(x1,
             x1 + mid - 1))
            lo = mid;
        else
            hi = mid - 1;
    }
    return lo;
}

bool cmp(const string &s, int x0, int y0, int x1, int y1)
{
    int L = LCP(s, x0, y0, x1, y1);
    if (L == y0 - x0 + 1)
        return true;
    if (L == y1 - x1 + 1)
        return false;
    return s[x0 + L] < s[x1 + L];
}

int main()
{
    invp[0] = ppow[0] = 1;
    for (int i = 1; i < maxn; ++i)
    {
        ppow[i] = mul(ppow[i - 1], 31);
        invp[i] = inv(ppow[i]);
    }

    string s = "mislav";
    string z = "isla";
    pre_hash(s);
    cout << hash_f(z) << ' ' << substr_hash(1, 4) << endl;
    for (int x : rabin_karp("la", "lahhlllallarla"))
        cout << x << ' ';

    return 0;
}
```