ACM-ICPC-REFERENCE

serchgabriel97

December 2017

Contents

1 Strings

1.1 trie

```
#include<bits/stdc++.h>

using namespace std;

class Trie {
private:
    class TrieNode {
    public:
        unordered_map<char, TrieNode*> children;
        // map<char, TrieNode*> children;
        bool endOfWord;
        int numberOfWords;
        TrieNode() {
            this->numberOfWords = 0;
            this->endOfWord = false;
        }
        ~TrieNode() {
            unordered_map<char, TrieNode*> thisNodeChildren =
            ↪  this->children;
            unordered_map<char, TrieNode*>::iterator i =
            ↪  thisNodeChildren.begin();
            // map<char, TrieNode*> thisNodeChildren = this->children;
            // map<char, TrieNode*>::iterator i =
            ↪  thisNodeChildren.begin();
            while (i != thisNodeChildren.end()) {
                delete i->second;
                i++;
            }
            thisNodeChildren.clear();
        }
    };
private:
    TrieNode* root = nullptr;
public:
    Trie() {
        root = new TrieNode();
    }
    ~Trie() {
        delete root;
    }
    void insert(string &word) {
        TrieNode *current = this->root;
        current->numberOfWords++;
        for (int i = 0; i < word.size(); i++) {
            char symbol = word[i];
            if (current->children.count(symbol) == 0)
                current->children[symbol] = new TrieNode();
            current = current->children[symbol];
            current->numberOfWords++;
        }
        current->endOfWord = true;
    }

    bool find(string &word) {
        TrieNode *current = this->root;
        for (int i = 0; i < word.size(); i++) {
            char symbol = word[i];
            if (current->children.count(symbol) == 0)
                return false;
            current = current->children[symbol];
        }
        return current->endOfWord;
    }

private:
    bool deleteWord(TrieNode *current, string &word, int &index) {
        if (index == word.size()) {
            if (!current->endOfWord)
                return false;
            current->endOfWord = false;
            return current->children.size() == 0;
        }
        char symbol = word[index];
        if (current->children.count(symbol) == 0)
            return false;
        bool shouldDeleteChild = deleteWord(current->children[symbol],
        ↪  word, index += 1);

        if (shouldDeleteChild) {
            current->children.erase(symbol);
            return current->children.size() == 0;
        }
        return false;
    }

    void getWords(TrieNode* node, vector<string> &words, string &word) {
        if (node->endOfWord)
            words.push_back(word);
```

```
        for (auto i : node->children) {
            getWords(i.second, words, word += i.first);
            word.pop_back();
        }
    }
public:
    void deleteWord(string &word) {
        int index = 0;
        deleteWord(this->root, word, index);
    }
    vector<string> getWords() {
        vector<string> words;
        string word = "";
        getWords(this->root, words, word);
        return words;
    }

    vector<string> getWords(string &prefix) {
        vector<string> words;
        TrieNode *current = this->root;
        for (int i = 0; i < prefix.size(); i++) {
            if (current->children.count(prefix[i]) == 0)
                return words;
            current = current->children[prefix[i]];
        }
        bool prevState = current->endOfWord;
        current->endOfWord = false;
        getWords(current, words, prefix);
        current->endOfWord = prevState;
        return words;
    }


};

void printv(vector<string> v) {
    if (v.size() == 0) {
        cout << "[]" << endl;
        return;
    }
    cout << "[" << v[0];
    for (int i = 1; i < v.size(); i++) {
        cout << ", " << v[i];
    }
    cout << "]" << endl;
}

int main() {
    std::ios_base::sync_with_stdio(0);

    int n, k;
    Trie *tr = new Trie();
    cin >> n;
    for (int i = 0; i < n; i++) {
        string aux;
        cin >> aux;
        tr->insert(aux);
    }
    cin >> k;
    for (int j = 0; j < k; j++) {
        string str;
        cin >> str;
        cout << "Case #" << j + 1 << ":" << '\n';
        printv(tr->getWords(str));
    }
    cin >> k;
    for (int j = 0; j < k; j++) {
        string str;
        cin >> str;
        tr->deleteWord(str);
        printv(tr->getWords());
    }

    delete tr;

    return 0;
}
```

1.2 kmp

```
#include <bits/stdc++.h>

using namespace std;

vector<int> prefixArray(string& pattern) {

    vector<int> prefixArr(pattern.size());
    for (int i = 0, j = 1; j < pattern.size();)
    {
        if (pattern[i] == pattern[j])
        {
```

```
            i++;
            prefixArr[j] = i;
            j++;
        } else {
            if (i != 0)
                i = prefixArr[i - 1];
            else {
                prefixArr[j] = 0;
                j++;
            }
        }
    }
    return prefixArr;
}

vector<int> kmp(string& str, string& pattern) {
    vector<int> positions;
    if (pattern.size() == 0)
        return positions;
    vector<int> prefixArr = prefixArray(pattern);
    for (int i = 0, j = 0; j < str.size();) {
        if (pattern[i] == str[j]) {
            j++;
            i++;
        } else {
            if (i != 0) {
                i = prefixArr[i - 1];
            } else {
                j++;
            }
        }
        if (i == pattern.size()) {
            positions.push_back(j + 1 - pattern.size());
            i = prefixArr[i - 1];
        }
    }
    return positions;
}

int main() {
    int t;
    cin >> t;
    for (int i = 0; i < t; i++) {
        string str, pattern;
        cin >> str >> pattern;
        vector<int> positions = kmp(str, pattern);
        if (positions.size() == 0) {
            cout << "Not Found" << "\n\n";
            continue;
        }
        cout << positions.size() << '\n';
        for (int j = 0; j < positions.size(); j++) {
            cout << positions[j] << " ";
        }
        cout << "\n\n";
    }
    return 0;
}
```

1.3 rabinKarp

```
#include <iostream>
#include <stdio.h>
#include <vector>

using namespace std;


class RollingHash {

public:
    vector <unsigned long long int> pow;
    vector <unsigned long long int> hash;
    unsigned long long int B;
    RollingHash(const string &str) : B(257) {
        int N = str.size();
        pow.resize(N + 1);
        hash.resize(N + 1);
        pow[0] = 1;
        hash[0] = 0;
        for (int i = 1; i <= N; ++i) {
            // in c++ an unsigned long long int is automatically modulated
            ↪  by 2^64
            pow[i] = pow[i - 1] * B;
            hash[i] = hash[i - 1] * B + str[i - 1];
        }
    }

    unsigned long long int getWordHash() {
        return hash[hash.size() - 1];
    }
```

```
    unsigned long long int getSubstrHash(int begin, int end) {
        return hash[end] - hash[begin - 1] * pow[end - begin + 1];
    }

    int size() {
        return hash.size();
    }
};

vector<int> rabinKarp(RollingHash &rhStr, string &pattern) {
    vector<int> positions;
    RollingHash rhPattern(pattern);
    unsigned long long int patternHash = rhPattern.getWordHash();
    int windowSize = pattern.size(), end = windowSize;
    for (int i = 1; end < rhStr.size(); i++) {
        if (patternHash == rhStr.getSubstrHash(i, end))
            positions.push_back(i);
        end = i + windowSize;
    }
    return positions;
}

int main() {
    int t;
    cin >> t;
    for (int i = 0; i < t; i++) {
        string str, pattern;
        cin >> str;
        RollingHash rhStr(str);
        int k;
        cin >> k;
        for (int l = 0; l < k; ++l)
        {
            cin >> pattern;
            vector<int> positions = rabinKarp(rhStr, pattern);
            if (positions.size() == 0) {
                cout << "Not Found" << "\n\n";
                continue;
            }
            cout << positions.size() << '\n';
            for (int j = 0; j < positions.size(); j++) {
                cout << positions[j] << " ";
            }
            cout << "\n\n";
        }
    }
    return 0;
}
```

2 Graphs

2.1 indexedPriorityQueue

```
#include <bits/stdc++.h>

using namespace std;

template <typename T> class indexedPriorityQueue {
private:
    class Node {
    public:
        double weight;
        T key;
    };
    vector<Node> pq;
    unordered_map<T, int> nodePosition;

private:
    void updatePositionInMap(T key1, T key2, int pos1, int pos2) {
        this->nodePosition[key1] = pos1;
        this->nodePosition[key2] = pos2;
    }
public:
    bool containsKey(T key) {
        return this->nodePosition.count(key);
    }

    T top() {
        return this->pq[0].key;
    }

    bool empty() {
        return this->pq.size() == 0;
    }

    void push(T key, int weight) {
        Node node;
        node.weight = weight;
        node.key = key;
```

```
        this->pq.push_back(node);
        int current = this->pq.size() - 1;
        int parentIndex = (current - 1) / 2;
        this->nodePosition[node.key] = current;
        while (parentIndex > -1) {
            Node parentNode = this->pq[parentIndex];
            Node currentNode = this->pq[current];
            // use '>' for minHeap, use '<' for maxheap
            if (parentNode.weight > currentNode.weight) {
                swap(this->pq[parentIndex], this->pq[current]);
                updatePositionInMap(this->pq[parentIndex].key,
                ↪  this->pq[current].key, parentIndex, current);
                current = parentIndex;
                parentIndex = (current - 1) / 2;
            } else
                break;
        }
    }

    void update(T key, int newWeight) {
        if (!this->nodePosition.count(key))
            return;
        int pos = this->nodePosition[key];
        this->pq[pos].weight = newWeight;
        int parent = (pos - 1) / 2;
        while (parent > - 1) {
            // use '>' for minHeap, use '<' for maxheap
            if (this->pq[parent].weight > this->pq[pos].weight) {
                swap(this->pq[parent], this->pq[pos]);
                updatePositionInMap(this->pq[parent].key,
                ↪  this->pq[pos].key, parent, pos);
                pos = parent;
                parent = (pos - 1) / 2;
            } else
                break;
        }
    }

    double getWeight(T key) {
        if (!this->nodePosition.count(key))
            return -1;
        return this->pq[this->nodePosition[key]].weight;
    }

    pair<T, double> pop() {
        int lastPos = this->pq.size() - 1;
        Node topNode = this->pq[0];
        this->pq[0] = this->pq[lastPos];
        this->nodePosition.erase(topNode.key);
        this->nodePosition[this->pq[0].key] = 0;
        this->pq.erase(pq.begin() + lastPos);
        int currentPos = 0;
        lastPos--;
        while (true) {
            int left = 2 * currentPos + 1;
            int right = left + 1;
            if (left > lastPos)
                break;
            if (right > lastPos)
                right = left;
            // in case of using maxheap, smallerPos will haver the
            ↪  biggerPos value
            // use '>' for minHeap, use '<' for maxheap
            int smallerPos = this->pq[right].weight >
            ↪  this->pq[left].weight ? left : right;
            // use '>' for minHeap, use '<' for maxheap
            if (this->pq[currentPos].weight > this->pq[smallerPos].weight)
            ↪  {
                swap(this->pq[currentPos], this->pq[smallerPos]);
                updatePositionInMap(this->pq[currentPos].key,
                ↪  this->pq[smallerPos].key, currentPos, smallerPos);
                currentPos = smallerPos;
            } else
                break;
        }
        return {topNode.key, topNode.weight};
    }

    void printPositionInMap () {
        cout << "{ ";
        for (auto i : this->nodePosition)
            cout << i.first << "=" << i.second << ",";
        cout << " }" << endl;
    }

    void printHeap () {
        for (Node n : this->pq)
            cout << n.weight << " " << n.key << endl;
    }

};
```

```
/*int main () {
    indexedPriorityQueue<string> ipq;
    ipq.push("Tushar", 3);
    ipq.push("Ani", 4);
    ipq.push("Vijay", 8);
    ipq.push("Pramila", 10);
    ipq.push("Roy", 5);
    ipq.push("NTF", 6);
    ipq.push("AFR", 2);
    ipq.update("Pramila", 1);
    ipq.printHeap();
    ipq.printPositionInMap();
    ipq.pop();
    ipq.printHeap();
    ipq.printPositionInMap();
    ipq.pop();
    ipq.printHeap();
    ipq.printPositionInMap();
    ipq.pop();
    ipq.printHeap();
    ipq.printPositionInMap();
    ipq.pop();
    ipq.printHeap();
    ipq.printPositionInMap();
    ipq.pop();
    ipq.printHeap();
    ipq.printPositionInMap();
    ipq.pop();
    ipq.printHeap();
    ipq.printPositionInMap();
    return 0;
}*/
```

2.2 dijkstra

```
#include <bits/stdc++.h>
#include "indexedPriorityQueue.cpp"

using namespace std;

double INF = 1 << 30;

template <class T> class Graph {
public:
    //node -> value , neighbors -> value, weight
    unordered_map<T, unordered_map<T, double> > nodes;
    bool isDirectedGraph;

    // 0 -> undirected, 1 -> directed
    Graph(bool isDirectedGraph = false) {
        this->isDirectedGraph = isDirectedGraph;
    }
    void addOrUpdateEdge(T v, T w, double cost = 0) {
        this->nodes[v][v] = 0;
        this->nodes[w][w] = 0;
        this->nodes[v][w] = cost;
        if (isDirectedGraph)
            return;
        this->nodes[w][v] = cost;
    }

    bool hasEdge(T v, T w) {
        return this->nodes.count(v) ? this->nodes[v].count(w) : false;
    }

    double getEdgeWeight(T v, T w) {
        if (hasEdge(v, w))
            return this->nodes[v][w];
        return INF;
    }

    void dijkstra(unordered_map<T, double> &distances, unordered_map<T, T>
    ↪  &parents, T source) {
        indexedPriorityQueue<T> ipq;
        for (auto node : nodes) {
            ipq.push(node.first, INF);
        }
        ipq.update(source, 0);
        distances[source] = 0;
        parents[source] = source;
        while (!ipq.empty()) {
            auto current = ipq.pop();
            distances[current.first] = current.second;
            for (auto neighbor : nodes[current.first]) {
                if (!ipq.containsKey(neighbor.first))
                    continue;
                double newDistance = distances[current.first] +
                ↪  neighbor.second;
```

```
                if (ipq.getWeight(neighbor.first) > newDistance) {
                    ipq.update(neighbor.first, newDistance);
                    parents[neighbor.first] = current.first;
                }
            }
        }
    }
};

void printMap (unordered_map<int, double> m) {
    cout << "{ ";
    for (auto i : m)
        cout << i.first << "=" << i.second << ",";
    cout << " }" << endl;
}

int main() {
    int T;
    cin >> T;
    for (int l = 1; l <= T; l++) {
        int n, m, s, t, a, b;
        double w;
        cin >> n >> m >> s >> t;
        Graph<int> g(0);
        while (m--) {
            cin >> a >> b >> w;
            g.addOrUpdateEdge(a, b, w < g.getEdgeWeight(a, b) ? w :
            ↪  g.getEdgeWeight(a, b));
        }
        cout << "Case #" << l << ": ";
        unordered_map<int, double> distances;
        unordered_map<int, int> parents;
        g.dijkstra(distances, parents, s);
        if (!distances.count(t)) {
            cout << "unreachable" << endl;
            continue;
        }

        if (distances[t] == INF) {
            cout << "unreachable" << endl;
            continue;
        }

        cout << distances[t] << endl;
    }
    return 0;
}
```

2.3 connectedComponents

```
#include <bits/stdc++.h>

using namespace std;
template <class T> class CC {
public:
    //node -> id , neighbors
    unordered_map<T, unordered_set<T> > nodes;

    void addEdge(T v, T w) {
        this->nodes[v].insert(v);
        this->nodes[w].insert(w);
        this->nodes[v].insert(w);
        this->nodes[w].insert(v);
    }

private:
    void dfsCC(vector<T> &component, unordered_map<T, int>
    ↪  &nodeComponentIds, unordered_set<T> &visited, T actualNodeId, int
    ↪  &componentId) {
        visited.insert(actualNodeId);
        nodeComponentIds[actualNodeId] = componentId;
        component.push_back(actualNodeId);
        for (auto neighborId : this->nodes[actualNodeId])
            if (!visited.count(neighborId))
                dfsCC(component, nodeComponentIds, visited, neighborId,
                ↪  componentId);
    }

public:
    pair<vector<vector<T>>, unordered_map<T, int>>
    ↪  getConnectedComponents() {
        unordered_map<T, int> nodeComponentIds;
        vector<vector<T>> connectedComponents;
        unordered_set<T> visited;
        int componentId = 1;
        for (auto node : this->nodes)
            if (!visited.count(node.first)) {
                vector<T> component;
                dfsCC(component, nodeComponentIds, visited, node.first,
                ↪  componentId);
                connectedComponents.push_back(component);
```

```
                componentId++;
            }
        return {connectedComponents, nodeComponentIds};
    }
};

string input() {
    string str;
    getline(cin, str);
    return str;
}

int main() {
    int t;
    t = stoi(input());
    input();
    while (t--) {
        CC<char> g;
        string highest;
        highest = input();
        for (char i = highest[0]; i >= 'A'; i--)
            g.addEdge(i, i);
        while (true) {
            string edge = input();
            if (edge == "")
                break;
            g.addEdge(edge[0], edge[1]);
        }
        pair<vector<vector<char>>, unordered_map<char, int>>
        ↪  connectedComponents = g.getConnectedComponents();
        cout << connectedComponents.first.size() << endl;
        if (t != 0)
            cout << endl;
    }
    return 0;
}
```

2.4 graphAPI

```
#include <bits/stdc++.h>
#include "unionFind.cpp"
#include "indexedPriorityQueue.cpp"

using namespace std;

double INF = 1 << 30;

class Edge {
public:
    int v, w;
    double weight;
    Edge(int v, int w, double weight = 0) {
        this->v = v;
        this->w = w;
        this->weight = weight;
    }
};

template <class T> class Graph {
public:
    //node -> value , neighbors -> value, weight
    unordered_map<T, unordered_map<T, double> > nodes;
    bool isDirectedGraph;
    UnionFind<T> uf;
    vector<Edge> edges;

    // 0 -> undirected, 1 -> directed
    Graph(bool isDirectedGraph = false) {
        this->isDirectedGraph = isDirectedGraph;
    }

    unordered_map<T, unordered_map<T, double> > getNodes() {
        return this->nodes;
    }

    void addOrUpdateEdge(T v, T w, double cost = 0) {
        this->nodes[v][v] = 0;
        this->nodes[w][w] = 0;
        this->nodes[v][w] = cost;
        this->edges.push_back(Edge(v, w, cost));
        if (isDirectedGraph)
            return;
        this->nodes[w][v] = cost;
        this->edges.push_back(Edge(w, v, cost));
        uf.addEdge(v, w);
    }

    bool hasEdge(T v, T w) {
        return this->nodes.count(v) ? this->nodes[v].count(w) : false;
    }
```

```
    double getEdgeWeight(T v, T w) {
        if (hasEdge(v, w))
            return this->nodes[v][w];
        return 0;
    }

    void printEdges() {
        for (auto node : this->nodes) {
            T u = node.first;
            for (auto neighbor : this->nodes[u]) {
                cout << u << " " << neighbor.first << " " <<
                ↪  neighbor.second << endl;
            }
        }
    }

private:
    void dfsCC(vector<T> &component, unordered_map<T, int>
    ↪  &nodeComponentIds, unordered_set<T> &visited, T actualNodeId, int
    ↪  &componentId) {
        visited.insert(actualNodeId);
        nodeComponentIds[actualNodeId] = componentId;
        component.push_back(actualNodeId);
        for (auto neighbor : this->nodes[actualNodeId])
            if (!visited.count(neighbor.first))
                dfsCC(component, nodeComponentIds, visited,
                ↪  neighbor.first, componentId);
    }

public:
    pair<vector<vector<T>>, unordered_map<T, int>>
    ↪  getConnectedComponents() {
        unordered_map<T, int> nodeComponentIds;
        vector<vector<T>> connectedComponents;
        unordered_set<T> visited;
        int componentId = 1;
        for (auto node : this->nodes)
            if (!visited.count(node.first)) {
                vector<T> component;
                dfsCC(component, nodeComponentIds, visited, node.first,
                ↪  componentId);
                connectedComponents.push_back(component);
                componentId++;
            }
        return {connectedComponents, nodeComponentIds};
    }

    bool areNodesConnected(T v, T w) {
        return this->uf.areVertexesConnected(v, w);
    }

private:
    void dfsTopologicalSort(unordered_map<T, double> neighbors, int
    ↪  &index, unordered_set<T> &visited, vector<T>
    ↪  &topologicalSortedNodes) {
        for (auto neighbor : neighbors) {
            if (!visited.count(neighbor.first)) {
                visited.insert(neighbor.first);
                dfsTopologicalSort(this->nodes[neighbor.first], index,
                ↪  visited, topologicalSortedNodes);
                topologicalSortedNodes[index] = neighbor.first;
                index--;
            }
        }
    }

public:
    vector<T> topologicalSort() {
        if (!this->isDirectedGraph) {
            vector<T> trash;
            return trash;
        }
        unordered_set<T> visited;
        vector<T> topologicalSortedNodes(this->nodes.size());
        int index = this->nodes.size() - 1;
        for (auto edge : this->nodes) {
            if (!visited.count(edge.first)) {
                visited.insert(edge.first);
                dfsTopologicalSort(edge.second, index, visited,
                ↪  topologicalSortedNodes);
                topologicalSortedNodes[index] = edge.first;
                index--;
            }
        }
        return topologicalSortedNodes;
    }

    double kruskalMST(vector<Edge> &mst) {
        //mst = minimum spanning tree
        double minCost = 0;
        // change '<' to '>' if maximum spanning tree is needed
```

```
    auto cmp = [] (const Edge & a, const Edge & b) {return a.weight <
    ↪  b.weight;};
    sort(this->edges.begin(), this->edges.end(), cmp);
    UnionFind<T> uf;
    int limit = nodes.size() - 1;
    for (int i = 0; (i < this->edges.size()) && (mst.size() < limit);
    ↪  i++) {
        Edge e = this->edges[i];
        T v = e.v, w = e.w;
        if (!uf.areVertexesConnected(v, w)) {
            uf.addEdge(v, w);
            mst.push_back(e);
            minCost += e.weight;
        }
    }
    return minCost;
}

void dijkstra(unordered_map<T, double> &distances, unordered_map<T, T>
↪  &parents, T source) {
    indexedPriorityQueue<T> ipq;
    for (auto node : nodes) {
        ipq.push(node.first, INF);
    }
    ipq.update(source, 0);
    distances[source] = 0;
    parents[source] = source;
    while (!ipq.empty()) {
        auto current = ipq.pop();
        distances[current.first] = current.second;
        for (auto neighbor : nodes[current.first]) {
            if (!ipq.containsKey(neighbor.first))
                continue;
            double newDistance = distances[current.first] +
            ↪  neighbor.second;
            if (ipq.getWeight(neighbor.first) > newDistance) {
                ipq.update(neighbor.first, newDistance);
                parents[neighbor.first] = current.first;
            }
        }
    }
}

bool bellmanFord(unordered_map<T, double> &distances, unordered_map<T,
↪  T> &parents, T source) {
    queue<T> q;
    unordered_set<T> in_queue;
    unordered_map<T, int> ocurrenceOfNodeInQueue;
    for (auto node : nodes) {
        distances[node.first] = INF;
        parents[node.first] = node.first;
    }
    distances[source] = 0;
    q.push(source);
    int limit = nodes.size() - 1;
    in_queue.insert(source);
    ocurrenceOfNodeInQueue[source] += 1;
    while (!q.empty()) {
        T u = q.front(); q.pop(); in_queue.erase(u);
        for (auto neighbor : nodes[u]) {
            T v = neighbor.first;
            double newDistance = distances[u] + neighbor.second;
            if (newDistance < distances[v]) {
                distances[v] = newDistance;
                parents[v] = u;
                if (!in_queue.count(v)) {
                    q.push(v);
                    ocurrenceOfNodeInQueue[v] += 1;
                    if (ocurrenceOfNodeInQueue[v] > limit)
                        return false;
                }
            }
        }
    }
    return true;
}

double maxFlow(vector<vector<T>> &paths, T source, T target) {
    Graph<T> residualGraph(1);
    residualGraph.nodes = this->nodes;
    double max_flow = 0;
    unordered_map<T, T> parent;
    while (hasAugmentedPath(residualGraph, parent, source, target)) {
        vector<T> path;
        double flow = INF;
        T v = target;
        while (v != source) {
            T u = parent[v];
            if (flow > residualGraph.getEdgeWeight(u, v))
                flow = residualGraph.getEdgeWeight(u, v);
            path.push_back(v);
```

```
                v = u;
            }
            path.push_back(source);
            reverse(path.begin(), path.end());
            paths.push_back(path);
            max_flow += flow;
            v = target;
            while (v != source) {
                T u = parent[v];
                residualGraph.addOrUpdateEdge(u, v,
                ↪  residualGraph.getEdgeWeight(u, v) - flow);
                residualGraph.addOrUpdateEdge(v, u,
                ↪  residualGraph.getEdgeWeight(v, u) + flow);
                v = u;
            }
        }
        return max_flow;
    }
private:
    bool hasAugmentedPath(Graph<T> &residualGraph, unordered_map<T, T>
    ↪  &parent, T source, T target) {
        queue<T> q;
        q.push(source);
        unordered_set<T> visited;
        visited.insert(source);
        while (!q.empty()) {
            T current = q.front(); q.pop();
            for (auto neighbor : residualGraph.nodes[current]) {
                T v = neighbor.first;
                if (!visited.count(v) && neighbor.second > 0) {
                    q.push(v);
                    visited.insert(v);
                    parent[v] = current;
                    if (v == target)
                        return true;
                }
            }
        }
        return false;
    }

};

int main() {
    return 0;
}
```

2.5 bellmanFord

```
#include <bits/stdc++.h>

using namespace std;

template <class T> class Graph {

public:
    //node -> value , neighbors -> value, weight
    unordered_map<T, unordered_map<T, double> > nodes;
    bool isDirectedGraph;
    double INF = 1 << 30;

    // 0 -> undirected, 1 -> directed
    Graph(bool isDirectedGraph = false) {
        this->isDirectedGraph = isDirectedGraph;
    }
    void addEdge(T v, T w, double cost = 0) {
        this->nodes[v][v] = 0;
        this->nodes[w][w] = 0;
        // consider the smallest edge in case of duplicates
        if (!(this->nodes.count(v) && this->nodes[v].count(w)) || (cost <
        ↪  this->nodes[v][w]))
            this->nodes[v][w] = cost;
        if (isDirectedGraph)
            return;
        this->nodes[w][v] = this->nodes[v][w];
    }

    bool bellmanFord(unordered_map<T, double> &distances, unordered_map<T,
    ↪  T> &parents, T source) {
        queue<T> q;
        unordered_set<T> in_queue;
        unordered_map<T, int> ocurrenceOfNodeInQueue;
        for (auto node : nodes) {
            distances[node.first] = INF;
            parents[node.first] = node.first;
        }
        distances[source] = 0;
        q.push(source);
        int limit = nodes.size() - 1;
        in_queue.insert(source);
        ocurrenceOfNodeInQueue[source] += 1;
```

```
        while (!q.empty()) {
            T u = q.front(); q.pop(); in_queue.erase(u);
            for (auto neighbor : nodes[u]) {
                T v = neighbor.first;
                double newDistance = distances[u] + neighbor.second;
                if (newDistance < distances[v]) {
                    distances[v] = newDistance;
                    parents[v] = u;
                    if (!in_queue.count(v)) {
                        q.push(v);
                        ocurrenceOfNodeInQueue[v] += 1;
                        if (ocurrenceOfNodeInQueue[v] > limit)
                            return false;
                    }
                }
            }
        }
        return true;
    }
};

void printMap (unordered_map<int, double> m) {
    cout << "{ ";
    for (auto i : m)
        cout << i.first << "=" << i.second << ",";
    cout << " }" << endl;
}

int main() {
    int T;
    cin >> T;
    while (T--) {
        unordered_map<int, double> distances;
        unordered_map<int, int> parents;
        Graph<int> g(1);
        int n, m, x, y, t;
        cin >> n >> m;
        while (m--) {
            cin >> x >> y >> t;
            g.addEdge(x, y, t);
        }
        if (!g.bellmanFord(distances, parents, 0))
            cout << "possible" << endl;
        else
            cout << "not possible" << endl;
    }
    return 0;
}
```

2.6 kruskalMST

```
#include <bits/stdc++.h>
#include "unionFind.cpp"

using namespace std;

class Edge {
public:
    int v, w;
    double weight;
    Edge(int v, int w, double weight) {
        this->v = v;
        this->w = w;
        this->weight = weight;
    }
};

template <class T> class Graph {
public:
    //node -> value , neighbors -> value, weight
    unordered_map<T, unordered_map<T, double> > nodes;
    bool isDirectedGraph;
    vector<Edge> edges;

    Graph(bool isDirectedGraph = false) {
        this->isDirectedGraph = isDirectedGraph;
    }
    void addEdge(T v, T w, double cost = 0) {
        this->nodes[v][v] = 0;
        this->nodes[w][w] = 0;
        this->nodes[v][w] = cost;
        this->edges.push_back(Edge(v, w, cost));
        if (isDirectedGraph)
            return;
        this->nodes[w][v] = cost;
        this->edges.push_back(Edge(w, v, cost));
    }

public:
    double kruskalMST(vector<Edge> &mst) {
```

```
        //mst = minimum spanning tree
        double minCost = 0;
        // change '<' to '>' if maximum spanning tree is needed
        auto cmp = [] (const Edge & a, const Edge & b) {return a.weight <
        ↪  b.weight;};
        sort(this->edges.begin(), this->edges.end(), cmp);
        UnionFind<T> uf;
        int limit = nodes.size() - 1;
        for (int i = 0; (i < this->edges.size()) && (mst.size() < limit);
        ↪  i++) {
            Edge e = this->edges[i];
            T v = e.v, w = e.w;
            if (!uf.areVertexesConnected(v, w)) {
                uf.addEdge(v, w);
                mst.push_back(e);
                minCost += e.weight;
            }
        }
        return minCost;
    }
};


void printv(vector<Edge> v) {
    if (v.size() == 0) {
        cout << "" << endl;
        return;
    }
    for (int i = 0; i < v.size(); i++) {
        cout << v[i].v << " " << v[i].w << " " << v[i].weight << endl;
    }
}

int main() {
    int i;
    int t = 0;
    while (cin >> i) {
        if (t != 0)
            cout << endl;
        auto *g = new Graph<int>(0);
        int a, b, cost;
        i--;
        while (i--) {
            cin >> a >> b >> cost;
            g->addEdge(a, b, cost);
        }
        vector<Edge> mst1;
        cout << g->kruskalMST(mst1) << endl;
        delete g;
        g = new Graph<int>(0);
        cin >> i;
        while (i--) {
            cin >> a >> b >> cost;
            g->addEdge(a, b, cost);
        }
        cin >> i;
        while (i--) {
            cin >> a >> b >> cost;
            g->addEdge(a, b, cost);
        }
        vector<Edge> mst2;
        cout << g->kruskalMST(mst2) << endl;
        t++;
    }
    return 0;
}
```

2.7 topologicalSort

```
#include <bits/stdc++.h>

using namespace std;
typedef int T;
class Graph {
public:
    //node -> value , neighbors
    unordered_map<T, unordered_set<T> > nodes;
    bool isDirectedGraph;

    Graph(bool isDirectedGraph) {
        this->isDirectedGraph = isDirectedGraph;
    }
    void addEdge(T v, T w) {
        this->nodes[v].insert(v);
        this->nodes[w].insert(w);
        this->nodes[v].insert(w);
        if (!isDirectedGraph)
            this->nodes[w].insert(v);
    }
```

```
private:
    void dfsTopologicalSort(unordered_set<T> neighbors, int &index,
    ↪  unordered_set<T> &visited, vector<T> &topologicalSortedNodes) {
        for (auto neighbor : neighbors) {
            if (!visited.count(neighbor)) {
                visited.insert(neighbor);
                dfsTopologicalSort(this->nodes[neighbor], index, visited,
                ↪  topologicalSortedNodes);
                topologicalSortedNodes[index] = neighbor;
                index--;
            }
        }
    }

public:
    vector<T> topologicalSort() {
        unordered_set<T> visited;
        vector<T> topologicalSortedNodes(this->nodes.size());
        int index = this->nodes.size() - 1;
        for (auto edge : this->nodes) {
            if (!visited.count(edge.first)) {
                visited.insert(edge.first);
                dfsTopologicalSort(edge.second, index, visited,
                ↪  topologicalSortedNodes);
                topologicalSortedNodes[index] = edge.first;
                index--;
            }
        }
        return topologicalSortedNodes;
    }

};


void printv(vector<T> v) {
    if (v.size() == 0) {
        cout << "" << endl;
        return;
    }
    cout << "" << v[0];
    for (int i = 1; i < v.size(); i++) {
        cout << " " << v[i];
    }
    cout << "" << endl;
}
```

```
int main() {
    while (true) {
        int n, m;
        T a, b;
        Graph *g = new Graph(true);
        cin >> n >> m;
        if (n == 0 && m == 0)
            break;
        while (n) {
            g->addEdge(n, n);
            n--;
        }
        while (m) {
            cin >> a >> b;
            g->addEdge(a, b);
            m--;
        }
        printv(g->topologicalSort());
        delete g;
    }



    return 0;
}
```

2.8 maxFlow

```
#include <bits/stdc++.h>

using namespace std;

template <class T> class Graph {
public:
    //node -> value , neighbors -> value, weight
    unordered_map<T, unordered_map<T, double> > nodes;
    bool isDirectedGraph;
    double INF = 1 << 30;
    // 0 -> undirected, 1 -> directed
    Graph(bool isDirectedGraph = false) {
        this->isDirectedGraph = isDirectedGraph;
    }

    unordered_map<T, unordered_map<T, double> > getNodes() {
```

```
    return this->nodes;
}

void addOrUpdateEdge(T v, T w, double cost = 0) {
    this->nodes[v][v] = 0;
    this->nodes[w][w] = 0;
    this->nodes[v][w] = cost;
    if (isDirectedGraph)
        return;
    this->nodes[w][v] = cost;
}

bool hasEdge(T v, T w) {
    return this->nodes.count(v) ? this->nodes[v].count(w) : false;
}

double getEdgeWeight(T v, T w) {
    if (hasEdge(v, w))
        return this->nodes[v][w];
    return 0;
}

void printEdges() {
    for (auto node : this->nodes) {
        T u = node.first;
        for (auto neighbor : this->nodes[u]) {
            cout << u << " " << neighbor.first << " " <<
            ↪  neighbor.second << endl;
        }
    }
}

double maxFlow(vector<vector<T>> &paths, T source, T target) {
    Graph<T> residualGraph(1);
    residualGraph.nodes = this->nodes;
    double max_flow = 0;
    unordered_map<T, T> parent;
    while (hasAugmentedPath(residualGraph, parent, source, target)) {
        vector<T> path;
        double flow = INF;
        T v = target;
        while (v != source) {
            T u = parent[v];
            if (flow > residualGraph.getEdgeWeight(u, v))
                flow = residualGraph.getEdgeWeight(u, v);
            path.push_back(v);
            v = u;
        }
        path.push_back(source);
        reverse(path.begin(), path.end());
        paths.push_back(path);
        max_flow += flow;
        v = target;
        while (v != source) {
            T u = parent[v];
            residualGraph.addOrUpdateEdge(u, v,
            ↪  residualGraph.getEdgeWeight(u, v) - flow);
            residualGraph.addOrUpdateEdge(v, u,
            ↪  residualGraph.getEdgeWeight(v, u) + flow);
            v = u;
        }
    }
    return max_flow;
}

bool hasAugmentedPath(Graph<T> &residualGraph, unordered_map<T, T>
↪  &parent, T source, T target) {
    queue<T> q;
    q.push(source);
    unordered_set<T> visited;
    visited.insert(source);
    while (!q.empty()) {
        T current = q.front(); q.pop();
        for (auto neighbor : residualGraph.nodes[current]) {
            T v = neighbor.first;
            if (!visited.count(v) && neighbor.second > 0) {
                q.push(v);
                visited.insert(v);
                parent[v] = current;
                if (v == target)
                    return true;
            }
        }
    }
    return false;
}
};

int main() {
    int i = 1, n, s, t, c, u, v;
    double w;
```

```
    while (true) {
        Graph<int> g(0);
        cin >> n;
        if (!n)
            break;
        cin >> s >> t >> c;
        while (c--) {
            cin >> u >> v >> w;
            g.addOrUpdateEdge(u, v, g.getEdgeWeight(u, v) + w);
        }
        vector<vector<int>> parents;
        cout << "Network " << i << endl;
        cout << "The bandwidth is " << g.maxFlow(parents, s, t) << "." <<
        ↪  endl;
        cout << endl;
        i++;
    }
    return 0;
}
```

2.9 unionFind

```
#include <bits/stdc++.h>

using namespace std;
template <class T> class UnionFind {
public:
    // stores the parent of each node
    unordered_map<T, T> tree;
    // stores the size of each sub-tree
    unordered_map<T, int> treeSize;
    void addEdge(T v, T w) {
        if (!this->tree.count(v)) {
            this->tree[v] = v;
            this->treeSize[v] = 1;
        }
        if (!this->tree.count(w)) {
            this->tree[w] = w;
            this->treeSize[w] = 1;
        }
        T i = setGetRoot(v);
        T j = setGetRoot(w);
        if (i == j)
            return;
        if (treeSize[i] < treeSize[j]) {
            this->tree[i] = j;
            this->treeSize[j] += this->treeSize[i];
        } else {
            this->tree[j] = i;
            this->treeSize[i] += this->treeSize[j];
        }
    }

    bool areVertexesConnected(T v, T w) {
        if (!this->tree.count(v) || !this->tree.count(w))
            return false;
        return setGetRoot(v) == setGetRoot(w);
    }

private:
    T setGetRoot(T v) {
        while (v != this->tree[v])
            v = this->tree[v] = this->tree[this->tree[v]];
        return v;
    }

};

string input() {
    string ans;
    getline(cin, ans);
    return ans;
}

vector<string> split(string str, char token) {
    stringstream test(str);
    string segment;
    vector<std::string> seglist;

    while (std::getline(test, segment, token))
        seglist.push_back(segment);
    return seglist;
}

/*int main() {
    string str;
    int t;
    t = stoi(input());
    str = input();
    while (t--) {
        auto g = new UnionFind<int>();
```

```
        int ac = 0, wa = 0;
        int n;
        n = stoi(input());
        while (n--) {
            g->addEdge(n + 1, n + 1);
        }
        while (true) {
            vector<string> vals = split(input(), ' ');
            if (vals.size() == 0)
                break;
            if (vals[0] == "c") {
                g->addEdge(stoi(vals[1]), stoi(vals[2]));
            }
            if (vals[0] == "q") {
                if (g->areVertexesConnected(stoi(vals[1]), stoi(vals[2])))
↪   {
                    ac++;
                }
                else {
                    wa++;
                }
            }
        }
        cout << ac << "," << wa << "\n";
        if (t != 0)
            cout << "\n";
        delete g;
    }
    return 0;
}*/
```

3 NumberTheory

3.1 extendedEuclidean

```
#include <bits/stdc++.h>

using namespace std;

void printv(vector<long long int> v) {
    if (v.size() == 0) {
        cout << "[]" << endl;
        return;
    }
    cout << "[" << v[0];
    for (int i = 1; i < v.size(); i++) {
        cout << ", " << v[i];
    }
    cout << "]" << endl;
}

// gcd(a, b) = ax + by
vector<long long int> extendedGCD(long long int a, long long int b) {
    if (a > 0LL && b == 0LL) {
        return {a, 1LL, 0LL};
    }
    long long int x = 1LL, y = 0LL, prevx = 0LL, prevy = 1LL, q,
    ↪  remainder;
    while (true) {
        q = a / b;
        remainder = a - b * q;
        if (remainder == 0LL)
            break;
        a = b;
        b = remainder;
        x = x - prevx * q;
        swap(x, prevx);
        y = y - prevy * q;
        swap(y, prevy);
    }
    // gcd = b, x = prevx, y = prevy
    return {b, prevx, prevy};
}


int main() {
    long long int a, b;
    cin >> a >> b;
    printv(extendedGCD(a, b));
    printv(extendedGCD(b, a));
    return 0;
}
```

3.2 divisibilityCriterion

```
def divisorCriteria(n, lim):
    results = []
    tenElevated = 1
    for i in range(lim):
        # remainder = pow(10, i, n)
        remainder = tenElevated % n
```

```
        negremainder = remainder - n
        if(remainder <= abs(negremainder)):
            results.append(remainder)
        else:
            results.append(negremainder)
        tenElevated *= 10
    return results


def testDivisibility(dividend, divisor, divisor_criteria):
    dividend = str(dividend)
    addition = 0
    dividendSize = len(dividend)
    i = dividendSize - 1
    j = 0
    while j < dividendSize:
        addition += int(dividend[i]) * divisor_criteria[j]
        i -= 1
        j += 1
    return addition % divisor == 0


if __name__ == '__main__':
    dividend, divisor = map(int, input().split())
    divisor_criteria = divisorCriteria(divisor, len(str(dividend)))
    print(divisor_criteria)
    print(testDivisibility(dividend, divisor, divisor_criteria))
```

3.3 gcd

```
#include <bits/stdc++.h>

using namespace std;

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

int gcdI(int a, int b) {
    while (b) {
        a %= b;
        swap(a, b);
    }
    return a;
}

int main() {
    int a, b;
    cin >> a >> b;
    cout << gcd(a, b) << "\n";
    cout << gcdI(a, b) << "\n";
}
```

4 Primes

4.1 myPrimesSieve

```
# sieve of primes, use dict if you want to save memory
# however using it will make this slower
def mySieve(N=10000000):
    n = N + 1
    dic = [0] * (n)
    # dic = {0: 0, 1: 1}
    primes = []
    if N == 2:
        primes = [2]
    if N > 2:
        primes = [2, 3]
    dic[0] = -1
    dic[1] = 1
    for i in range(4, n, 2):
        dic[i] = 2
    for i in range(9, n, 6):
        dic[i] = 3
    i = 5
    w = 2
    k = i * i
    while k < n:
        # if i not in dic:
        if dic[i] == 0:
            primes.append(i)
            # skip multiples of 2
            jump = 2 * i
            for j in range(k, n, jump):
                dic[j] = i
        i += w
        w = 6 - w
        k = i * i
    # if you need primes bigger than the root of N
    while i < n:
        if dic[i] == 0:
```

```
            primes.append(i)
        i += w
        w = 6 - w
    return dic, primes


if __name__ == '__main__':
    print(mySieve(int(input()))[1])
```

4.2 primeFactorization

```
def mySieve(N=10000000):
    n = N + 1
    dic = [0] * (n)
    # dic = {0: 0, 1: 1}
    primes = []
    if N == 2:
        primes = [2]
    if N > 2:
        primes = [2, 3]
    dic[0] = -1
    dic[1] = 1
    for i in range(4, n, 2):
        dic[i] = 2
    for i in range(9, n, 6):
        dic[i] = 3
    i = 5
    w = 2
    k = i * i
    while k < n:
        # if i not in dic:
        if dic[i] == 0:
            primes.append(i)
            # skip multiples of 2
            jump = 2 * i
            for j in range(k, n, jump):
                dic[j] = i
        i += w
        w = 6 - w
        k = i * i
    # if you need primes bigger than the root of N
    while i < n:
        if dic[i] == 0:
            primes.append(i)
        i += w
        w = 6 - w
    return dic, primes


def getPrimeFactors(N, sieveToMaxN):
    n = N
    primeFactors = []
    while n != 1:
        if sieveToMaxN[n] == 0:
            primeFactors.append(n)
            break
        primeFactors.append(sieveToMaxN[n])
        n /= sieveToMaxN[n]
    return primeFactors


if __name__ == '__main__':
    n = int(input())
    sieve = mySieve(n)[0]
    print(sieve)
    print(getPrimeFactors(n, sieve))
```

4.3 isPrimeSieve

```
#include <bits/stdc++.h>

using namespace std;

pair<vector<int>, vector<int> > mySieve(int N) {
    int n = N + 1;
    vector<int> dic(n);
    vector<int> primes;
    if (N == 2)
        primes = {2};
    if (N > 2)
        primes = {2, 3};
    dic[0] = -1;
    dic[1] = 1;
    for (int i = 4; i < n; i += 2)
        dic[i] = 2;
    for (int i = 9; i < n; i += 6)
        dic[i] = 3;
    int i = 5, w = 2, k = i * i;
    while (k < n) {
        if (dic[i] == 0) {
            primes.push_back(i);
            // skip multiples of 2
```

```
            int jump = 2 * i;
            for (long long int j = k; j < n; j += jump)
                dic[j] = i;
        }
        i += w;
        w = 6 - w;
        k = i * i;
    }
    // if you need primes bigger than the root of N
    while (i < n) {
        if (dic[i] == 0)
            primes.push_back(i);
        i += w;
        w = 6 - w;
    }
    return {dic, primes};
}

bool isPrime(int N, vector<int> &sieve, vector<int> &primes) {
    if (N < sieve.size())
        return sieve[N] == 0 ? true : false;
    for (int prime : primes) {
        if (prime * prime > N)
            break;
        if (N % prime == 0)
            return false;
    }
    return true;
}

int main() {
    pair<vector<int>, vector<int> > sieve = mySieve(10000000);
    long long int n;
    cin >> n;
    cout << isPrime(n, sieve.first, sieve.second) << '\n';
    return 0;
}
```

4.4 isPrimeMillerRabin

```
from random import randrange


def is_prime(p):
    k = 100
    if p == 2 or p == 3:
        return True
    if (p & 1) == 0 or p == 1:
        return False
    phi = p - 1
    d = phi
    r = 0
    while (d & 1) == 0:
        d = int(d >> 1)
        r += 1
    for i in range(k):
        a = randrange(2, p - 2)
        exp = pow(a, d, p)
        if exp == 1 or exp == p - 1:
            continue
        flag = False
        for j in range(r - 1):
            exp = pow(exp, 2, p)
            if exp == 1:
                return False
            if exp == p - 1:
                flag = True
                break
        if flag:
            continue
        else:
            return False
    return True


if __name__ == '__main__':
    while True:
        try:
            n = int(input())
            print(n, is_prime(n))
        except EOFError:
            break
```

4.5 primesSievesComparison

```
from math import sqrt
import timeit


# sieve of primes, use dict if you want to save memory
# however using it will make this slower
def sieve(N=100000000):
```

```
    n = N + 1
    dic = [0] * (n)
    # dic = {0: 0, 1: 1}
    dic[0] = -1
    dic[1] = 1
    for i in range(4, n, 2):
        dic[i] = 2
    for i in range(9, n, 6):
        dic[i] = 3
    i = 5
    w = 2
    k = i * i
    while k < n:
        # if i not in dic:
        if dic[i] == 0:
            # skip multiples of 2
            jump = 2 * i
            for j in range(k, n, jump):
                dic[j] = i
        i += w
        w = 6 - w
        k = i * i
    return dic


def clasicSieve(N=100000000):
    criba = [0] * (N + 1)
    raiz = int(sqrt(N))
    criba[0] = -1
    criba[1] = 1
    for i in range(4, N + 1, 2):
        criba[i] = 2
    for i in range(3, raiz + 1, 2):
        if (criba[i] == 0):
            for j in range(i * i, N + 1, i):
                if (criba[j] == 0):
                    criba[j] = i
    return criba


if __name__ == '__main__':
    print(timeit.timeit(clasicSieve, number=1))
    print(timeit.timeit(sieve, number=1))
```

4.6 primesSievesComparison

```
#include <bits/stdc++.h>

using namespace std;


vector<int> sieve(int N) {
    int n = N + 1;
    vector<int> dic(n);
    dic[0] = -1;
    dic[1] = 1;
    for (int i = 4; i < n; i += 2)
        dic[i] = 2;
    for (int i = 9; i < n; i += 6)
        dic[i] = 3;
    int i = 5, w = 2, k = i * i;
    while (k < n) {
        if (dic[i] == 0) {
            int jump = 2 * i;
            for (int j = k; j < n; j += jump)
                dic[j] = i;
        }
        i += w;
        w = 6 - w;
        k = i * i;
    }
    return dic;
}

// Criba de Eratostenes de 1 a n.
vector<int> clasicSieve(int n) {
    vector<int> criba(n + 1);
    for (int i = 4; i <= n; i += 2)
        criba[i] = 2;
    for (int i = 3; i * i <= n; i += 2)
        if (!criba[i])
            for (int j = i * i; j <= n; j += i)
                if (!criba[j]) criba[j] = i;
    return criba;
}


int main() {
    int n = 10000000;
    cin >> n;
```

```
    clock_t start, stop;
    for (int i = 0; i < 4; i++) {
        start = clock();
        clasicSieve(n);
        stop = clock();
        cout << (double)(stop - start) / CLOCKS_PER_SEC << " seconds." <<
        ↪  endl;

        start = clock();
        sieve(n);
        stop = clock();
        cout << (double)(stop - start) / CLOCKS_PER_SEC << " seconds." <<
        ↪  endl;
    }
    return 0;
}
```

4.7 primeFactorization

```
#include <bits/stdc++.h>

using namespace std;

// sieve of primes, use unordered_map if you want to save memory
// however using it will make this slower
pair<vector<int>, vector<int> > mySieve(int N) {
    int n = N + 1;
    vector<int> dic(n);
    vector<int> primes;
    if (N == 2)
        primes = {2};
    if (N > 2)
        primes = {2, 3};
    dic[0] = -1;
    dic[1] = 1;
    for (int i = 4; i < n; i += 2)
        dic[i] = 2;
    for (int i = 9; i < n; i += 6)
        dic[i] = 3;
    int i = 5, w = 2, k = i * i;
    while (k < n) {
        if (dic[i] == 0) {
            primes.push_back(i);
            // skip multiples of 2
            int jump = 2 * i;
            for (long long int j = k; j < n; j += jump)
                dic[j] = i;
        }
        i += w;
        w = 6 - w;
        k = i * i;
    }
    // if you need primes bigger than the root of N
    while (i < n) {
        if (dic[i] == 0)
            primes.push_back(i);
        i += w;
        w = 6 - w;
    }
    return {dic, primes};
}


vector<int> getPrimeFactors(long long int N, vector<int> &sieveToMaxN) {
    long long int n = N;
    vector<int> primeFactors;
    while (n != 1LL) {
        if (sieveToMaxN[n] == 0) {
            primeFactors.push_back(n);
            break;
        }
        primeFactors.push_back(sieveToMaxN[n]);
        n /= sieveToMaxN[n];
    }
    return primeFactors;
}


void printv(vector<int> v) {
    if (v.size() == 0) {
        cout << "[]" << endl;
        return;
    }
    cout << "[" << v[0];
    for (int i = 1; i < v.size(); i++) {
        cout << ", " << v[i];
    }
    cout << "]" << endl;
}
```

```cpp
int main() {
    int n;
    cin >> n;
    vector<int> sieve = mySieve(n).first;
    printv(sieve);
    printv(getPrimeFactors(n, sieve));
}
```

4.8 myPrimesSieve

```cpp
#include <bits/stdc++.h>

using namespace std;

// sieve of primes, use unordered_map if you want to save memory
// however using it will make this slower
pair<vector<int>, vector<int> > mySieve(int N) {
    int n = N + 1;
    vector<int> dic(n);
    vector<int> primes;
    if (N == 2)
        primes = {2};
    if (N > 2)
        primes = {2, 3};
    dic[0] = -1;
    dic[1] = 1;
    for (int i = 4; i < n; i += 2)
        dic[i] = 2;
    for (int i = 9; i < n; i += 6)
        dic[i] = 3;
    int i = 5, w = 2, k = i * i;
    while (k < n) {
        if (dic[i] == 0) {
            primes.push_back(i);
            // skip multiples of 2
            int jump = 2 * i;
            for (long long int j = k; j < n; j += jump)
                dic[j] = i;
        }
        i += w;
        w = 6 - w;
        k = i * i;
    }
    // if you need primes bigger than the root of N
    while (i < n) {
        if (dic[i] == 0)
            primes.push_back(i);
        i += w;
        w = 6 - w;
    }
    return {dic, primes};
}

void printv(vector<int> v) {
    if (v.size() == 0) {
        cout << "[]" << endl;
        return;
    }
    cout << "[" << v[0];
    for (int i = 1; i < v.size(); i++) {
        cout << ", " << v[i];
    }
    cout << "]" << endl;
}


int main() {
    int n;
    cin >> n;
    printv(mySieve(n).second);
}
```

5 CodingResources

5.1 priorityQueueOfClass

```cpp
#include <bits/stdc++.h>

using namespace std;

struct Object {
    char first;
    int second;
};

template<typename T> void print_queue(T& q) {
    while(!q.empty()) {
        std::cout << "{" << q.top().first << " " << q.top().second << "}";
        q.pop();
    }
    std::cout << '\n';
```

```
}

int main() {
    auto cmp = [](const Object& a, const Object& b) {return a.second >
    ↪  b.second;};
    priority_queue<Object, vector<Object>, decltype(cmp)> pq(cmp);
    vector<Object> v = {{'c',3}, {'a', 1}, {'b', 2}};
    for (auto i : v)
        pq.push(i);
    sort(v.begin(), v.end(), cmp);
    print_queue(pq);
    return 0;
}
```

5.2 printVector

```
#include <bits/stdc++.h>

using namespace std;

void printv(vector<int> v) {
    if (v.size() == 0) {
        cout << "[]" << endl;
        return;
    }
    cout << "[" << v[0];
    for (int i = 1; i < v.size(); i++) {
        cout << ", " << v[i];
    }
    cout << "]" << endl;
}

int main() {
    vector<int> v = {1, 2, 3, 4, 5, 6};
    printv(v);
    return 0;
}
```

5.3 splitString

```
#include <bits/stdc++.h>

using namespace std;

vector<string> split(string str, char token) {
    stringstream test(str);
    string segment;
    vector<std::string> seglist;

    while (std::getline(test, segment, token))
        seglist.push_back(segment);
    return seglist;
}

int main () {
    string str;
    getline(cin, str);
    vector<string> segments = split(str, ' ');
    for (string segment : segments)
        cout << segment << endl;
    return 0;
}
```

5.4 intToBinary

```
#include <bits/stdc++.h>
using namespace std;
typedef long long int lli;

lli bitsInInt(lli n) {
    return floor(log2(n) + 1LL);
}

void printv(vector <int> v) {
    cout << v[0];
    for (int i = 1; i < v.size(); i++) {
        cout << " " << v[i];
    }
    cout << endl;
}

vector<int> intToBitsArray(lli n) {
    n = abs(n);
    if (!n) {
        vector<int> v;
        return v;
    }
    int length = bitsInInt(n);
    int lastPos = length - 1;
    vector<int> v(length);
    for (lli i = lastPos, j = 0; i > -1LL; i--, j++) {
        lli aux = (n >> i) & 1LL;
```

```
        v[j] = aux;
    }
    return v;
}

int main() {
    lli n;
    cin >> n;
    printv(intToBitsArray(n));
    return 0;
}
```

5.5 readLineCpp

```
#include <bits/stdc++.h>

using namespace std;
/*
string strInput() {
    string ans;
    cin >> ans;
    cin.ignore();
    return ans;
}

int intInput() {
    int ans;
    cin >> ans;
    cin.ignore();
    return ans;
}

double dInput() {
    double ans;
    cin >> ans;
    cin.ignore();
    return ans;
}
*/
string input() {
    string ans;
    cin >> ws;
    getline(cin, ans);
    return ans;
}

int main() {
    ios_base::sync_with_stdio(0);
    string ans;
    // cout << strInput() << endl;
    cin >> ans;
    cout << ans << endl;
    cout << input() << endl;
    return 0;
}
```

5.6 sortVectorOfClass

```
#include <bits/stdc++.h>

using namespace std;

struct Object {
    char first;
    int second;
};

void printv(vector<Object> v) {
    if (v.size() == 0) {
        cout << "[]" << endl;
        return;
    }
    cout << "[{" << v[0].first << ", " << v[0].second << "}";
    for (int i = 1; i < v.size(); i++) {
        cout << ", {" << v[i].first << ", " << v[i].second << "}";
    }
    cout << "]" << endl;
}

int main() {
    auto cmp = [](const Object& a, const Object& b) {return a.second >
    ↪  b.second;};
    vector<Object> v = {{'c',3}, {'a', 1}, {'b', 2}};
    sort(v.begin(), v.end(), cmp);
    printv(v);
    return 0;
}
```

5.7 sortListOfClass

```
class MyObject:
    def __init__(self, first, second):
```

```
        self.first = first
        self.second = second

l = [MyObject('c', 3), MyObject('a', 1), MyObject('b', 2)]

for myObject in l:
    print(myObject.first, myObject.second)
print()
l.sort(key=lambda x: x.first, reverse=False)

for myObject in sorted(l, key=lambda x: x.first, reverse=False):
    print(myObject.first, myObject.second)

print()

for myObject in l:
    print(myObject.first, myObject.second)
```