ACM-ICPC-REFERENCE

serchgabriel97

December 2017

Contents

1 NumberTheory

1.1 extendedEuclidean

```cpp
#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;
}
```

1.2 divisibilityCriterion

```python
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))
```

1.3 gcd

```cpp
#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";
}
```

2 strings

2.1 trie

```cpp
#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;
}
```

2.2 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;
}
```

2.3 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;
}
```

3 graphs

3.1 dijkstra

```
#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;

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


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() {
    auto *g = new Graph<int>(0);
    return 0;
}
```

3.2 connectedComponents

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

using namespace std;
typedef int T;
class Graph {
public:
    //node -> value , neighbors
    unordered_map<T, unordered_set<T> > nodes;
    unordered_map<T, T> tree;
    unordered_map<T, int> treeSize;
    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);
        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];
        }
```

```
    }

private:
    void dfs(unordered_set<T> &vertexesInComponent, unordered_set<T>
    ↪  &visited, vector<T> &connectedComponents) {
        for (auto vertex : vertexesInComponent) {
            if (!visited.count(vertex)) {
                connectedComponents.push_back(vertex);
                visited.insert(vertex);
                dfs(this->nodes[vertex], visited,
                ↪  connectedComponents);
            }
        }
    }

public:
    vector< vector<T> > getConnectedComponents() {
        unordered_set<T> visited;
        vector< vector<T> > connectedComponents;
        for (auto edge : this->nodes) {
            if (!visited.count(edge.first)) {
                vector<T> vertexesInComponent;
                vertexesInComponent.push_back(edge.first);
                visited.insert(edge.first);
                dfs(edge.second, visited, vertexesInComponent);
                connectedComponents.push_back(vertexesInComponent);
            }
        }
        return connectedComponents;
    }

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

    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]) {
            this->tree[v] = this->tree[this->tree[v]];
            v = this->tree[v];
        }
        return v;
    }

};


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() {
    Graph g;
    T a, b;
    int i;
    cin >> i;
    while (i--) {
        cin >> a >> b;
        g.addEdge(a, b);
    }

    for (auto v : g.getConnectedComponents()) {
        printv(v);
    }
    cin >> i;
    while (i--) {
        cin >> a >> b;
        cout << g.areVertexesConnected(a, b) << '\n';
    }
    return 0;
```

```
}
```

3.3 graphAPI

```
#include <bits/stdc++.h>
#include "unionFind.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;
    UnionFind<T> uf;

    Graph(bool isDirectedGraph = false) {
        this->isDirectedGraph = isDirectedGraph;
    }
    void addEdge(T v, T w, double cost = 0) {
        this->nodes[v][v] = cost;
        this->nodes[w][w] = cost;
        this->nodes[v][w] = cost;
        if (isDirectedGraph)
            return;
        this->nodes[w][v] = cost;
        uf.addEdge(v, w);
    }
public:
    bool isEdgeInGraph(T v, T w) {
        return this->nodes.count(v) ? this->nodes[v].count(w) :
        ↪  false;
    }

private:
    void dfsConnectedComponents(unordered_map<T, double>
    ↪  &vertexesInComponent, unordered_set<T> &visited, vector<T>
    ↪  &connectedComponents) {
        for (auto vertex : vertexesInComponent) {
            if (!visited.count(vertex.first)) {
                connectedComponents.push_back(vertex.first);
                visited.insert(vertex.first);
                dfsConnectedComponents(this->nodes[vertex.first],
                ↪  visited, connectedComponents);
            }
        }
    }

public:
    vector< vector<T> > getConnectedComponents() {
        unordered_set<T> visited;
        vector< vector<T> > connectedComponents;
        if (this->isDirectedGraph)
            return connectedComponents;
        for (auto edge : this->nodes) {
            if (!visited.count(edge.first)) {
                vector<T> vertexesInComponent;
                vertexesInComponent.push_back(edge.first);
                visited.insert(edge.first);
                dfsConnectedComponents(edge.second, visited,
                ↪  vertexesInComponent);
                connectedComponents.push_back(vertexesInComponent);
            }
        }
        return connectedComponents;
    }
public:
    bool areVertexesConnected(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;
    }

};


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() {
    while (true) {
        int n, m;
        int a, b;
        auto *g = new Graph<int>(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;
}*/
int main() {
    Graph<int> g(0);
    int a, b;
    int i;
    cin >> i;
    while (i--) {
        cin >> a >> b;
        g.addEdge(a, b);
    }
    for (auto v : g.getConnectedComponents()) {
        printv(v);
    }
    cin >> i;
    while (i--) {
        cin >> a >> b;
        cout << g.areVertexesConnected(a, b) << '\n';
    }
    return 0;
}
```

3.4 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;
}
```

3.5 kruskal

```
#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] = cost;
        this->nodes[w][w] = cost;
        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:
    vector<Edge> kruskalMST() {
        //mst = minimum spanning tree
        vector<Edge> mst;
        int 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;
            }
        }
        cout << minCost << endl;
        return mst;
    }
};


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);
        }
        g->kruskalMST();
        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);
        }
        g->kruskalMST();
        t++;
    }
    return 0;
}
```

3.6 unionFind

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

using namespace std;
template <class T> class UnionFind {
public:
    unordered_map<T, T> 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]) {
            this->tree[v] = this->tree[this->tree[v]];
            v = this->tree[v];
        }
        return v;
    }

};

/*int main() {
    UnionFind<int> g;
    int a, b;
    int i;
    cin >> i;
    while (i--) {
        cin >> a >> b;
        g.addEdge(a, b);
    }
    cin >> i;
    while (i--) {
        cin >> a >> b;
```

```
        cout << g.areVertexesConnected(a, b) << '\n';
    }
    return 0;
}*/
```

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;
}


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

4.8 myPrimesSieve

```
#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);
}
```