CSC3100 Data Structures Fall 2023 Assignment Solutions Writeup

December 28, 2023

Assignment 1

Queue Disorder

Written and developed by: **Xuanhao Pan**

Editorial by: **Yohandi**

Description In a queue, people are supposed to stand in ascending order of their heights. However, due to some confusion, the order of the queue gets disrupted. Your task is to determine the number of disorder pairs in the queue.

A disorder pair is defined as a pair of people (p_i, p_j) such that $i < j$ and p_i is taller than p_j in the queue, which means their order is reversed.

For example, consider a queue with 5 people: $[180, 160, 175, 170, 170]$. In this case, there are 6 disorder pairs: $(180, 160), (180, 175), (180, 170), (180, 170), (175, 170)$ and $(175, 170)$. Please note that $(170, 170)$ is not considered as a disorder pair. Write a program to calculate the number of disorder pairs in a given queue with N people.

Constraints

- $1 \leq N \leq 10^6$
- $1 \leq p_i \leq 10^9, \forall i \in \{1, \ldots, N\}$

Solution This problem is a straightforward inversion count problem. We can use merge sort to maintain the number of inversions between pairs. Suppose a range $[L, R)$ accounts for the number of inversions of all pairs in the range. This then recurses into sub-ranges $[L, \text{mid})$ and $[\text{mid}, R)$. Any indices i and j such that $i \in [L, \text{mid})$ and $j \in [\text{mid}, R)$ can be processed concurrently with the merge procedure for the two half arrays using the sorted property.

Alternatively, we can use a Binary Indexed Tree to count the number of elements within a specific index range. To count only those elements smaller than a certain number, updates can be made following the sequence of the sorted elements. Each element, however, should be mapped into a pair of itself and its index (descendingly) to handle elements with the same value.

Both solutions work in $\mathcal{O}(n \log n)$.

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

class BIT {
private:
  vector<long long> bit;
  int size;

public:
  BIT(int n) {
    size = n;
    bit.resize(n + 1, 0LL);
  }

  void update(int idx, int delta) {
```

```
    idx++;
    while (idx <= size) {
      bit[idx] += delta;
      idx += idx & -idx;
    }
  }

  long long query(int idx) {
    idx++;
    long long sum = 0;
    while (idx > 0) {
      sum += bit[idx];
      idx -= idx & -idx;
    }
    return sum;
  }
};

int main() {
  int n;
  cin >> n;

  int pos = 0;
  vector<pair<int, int>> a(n);
  for (auto &e : a) {
    cin >> e.first;
    e.first *= -1;
    e.second = -(++pos);
  }

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

  BIT tree(n);
  long long ans = 0;
  for (auto e : a) {
    ans += tree.query(-e.second - 1);
    tree.update(-e.second, 1);
  }

  cout << ans << endl;
  return 0;
}
```

Star Sequence

Written and developed by: **Tianci Hou**

Editorial by: **Yohandi**

Description Renko Usami observes space through a telescope when she notices a fantastic phenomenon – the number of stars in the fields follows a mathematical pattern.

Specifically, let's denote the number of stars in the N-th field by f_N, then f_N satisfies the following expression, and a, b are given positive integers.

$$f_N = af_{N-1} + bf_{N-2} \ (N \geq 2)$$

Now Renko Usami is curious about how many stars in the n-th field, but the n-th field is too far away to be observed through her cheap astronomical telescope. Since there are so many stars, she only cares about the value of the number of stars modulo m. In other words, she want to know $f_n \mod m$.

Fortunately, Renko Usami is able to observe the two closest star fields to her, and the numbers of stars in fields are f_0 and f_1. Unfortunately, she is going to be late again for her appointment with Merry. Can you write a program for her to calculate $f_n \mod m$?

Constraints

- $1 \leq n \leq 10^{18}$
- $1 \leq a, b, f_0, f_1 \leq 100$
- $1 \leq m \leq 2 \cdot 10^9$

Solution To calculate $f_n \mod m$, we use matrix exponentiation to efficiently compute the required term in the sequence. We can construct a vector consisting of f_N and f_{N-1} using the following matrix power multiplied by the initial vector, which consists of f_1 and f_0. The formulation is derived as follows:

$$\begin{pmatrix} f_N \\ f_{N-1} \end{pmatrix} = \begin{pmatrix} a & b \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f_{N-1} \\ f_{N-2} \end{pmatrix} = \ldots = \begin{pmatrix} a & b \\ 1 & 0 \end{pmatrix}^{N-1} \begin{pmatrix} f_1 \\ f_0 \end{pmatrix}$$

The matrix power can be computed using the divide and conquer technique, which works in $\mathcal{O}(\log n)$.

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

long long MOD;

struct Matrix {
  vector<vector<long long>> val;

  void set() { val.resize(2, vector<long long>(2, 0LL)); }
} fs, I;

Matrix mult(Matrix A, Matrix B) {
  Matrix res;
  res.set();

  for (int k = 0; k < 2; ++k) {
    for (int i = 0; i < 2; ++i) {
      for (int j = 0; j < 2; ++j) {
        res.val[i][j] += A.val[i][k] * B.val[k][j];
        res.val[i][j] %= MOD;
      }
    }
  }

  return res;
}

Matrix pow(Matrix a, long long b) {
  if (b == 0)
    return I;

  Matrix tmp = pow(a, b / 2);

  tmp = mult(tmp, tmp);
  if (b % 2 == 1)
    tmp = mult(tmp, fs);

  return tmp;
}

int main() {
  long long n, a, b, f0, f1;
```

```
  cin >> n >> a >> b >> f0 >> f1 >> MOD;

  I.set();
  I.val[0][0] = 1;
  I.val[0][1] = 0;
  I.val[1][0] = 0;
  I.val[1][1] = 1;

  fs.set();
  fs.val[0][0] = a;
  fs.val[0][1] = b;
  fs.val[1][0] = 1;
  fs.val[1][1] = 0;

  Matrix res = pow(fs, n - 1);
  cout << (res.val[0][0] * f1 % MOD + res.val[0][1] * f0 % MOD) % MOD << endl;

  return 0;
}
```

Assignment 2

Battle Game

Written and developed by: **Bingwei Zhang**

Editorial by: **Yohandi**

Description Imagine a group of n players that are located in different floors. The i-th player (for each i such that $1 \leq i \leq n$, where i is an integer) is characterized by the following: - p_i, a unique starting floor level - h_i, a starting *HP* (Health Points) value - d_i, a direction of movement, either `U` indicating that the i-th player is going upward or `D` indicating that the i-th player is going downward

All players move simultaneously at the same speed in their respective directions. When two players meet, they engage in a battle. The player with the lower health is eliminated instantaneously, while the other continue to move in the same direction at the same speed but with a reduced HP by 1. In a case where two players have identical HP, they are both eliminated simultaneously after the battle.

Your task is to determine the final HP of the surviving players

Constraints

- $1 \leq n \leq 10^6$
- $1 \leq p_i \leq 10^8, \forall i \in \{1, \ldots, n\}$
- $1 \leq h_i \leq 100, \forall i \in \{1, \ldots, n\}$
- $d_i \in \{\texttt{U}, \texttt{D}\}, \forall i \in \{1, \ldots, n\}$

Solution Note that downward-moving players survive only if they win every battle with all players below them that are moving upward. Moreover, there won't be any fight between players that are moving in the same direction.

To solve this, we first sort the players by their starting floor. We then simulate from the lowest floor. If a player is moving up, we add them to a stack. Otherwise, this player battles against every player in the current state of the stack, starting from the one on top.

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

```
int main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  cout.tie(0);

  struct Player {
    int id, p, h;
    char d;
  };
  auto battle = [](Player A, Player B) -> Player {
    if (A.h == B.h)
      return {-1, -1, -1, 'X'};
    if (A.h < B.h)
      std::swap(A, B);
    A.h -= 1;
    return A;
  };

  int n;
  cin >> n;

  vector<Player> players(n);
  for (int i = 0; i < n; ++i) {
    cin >> players[i].p >> players[i].h >> players[i].d;
    players[i].id = i;
  }

  sort(players.begin(), players.end(),
       [](const Player &A, const Player &B) { return A.p < B.p; });

  vector<int> surviving_players(n, -1);
  stack<Player> st;
  for (int i = 0; i < n; ++i) {
    if (players[i].d == 'U') {
      st.push(players[i]);
      continue;
    }
    Player current_player = players[i];
    while (!st.empty() && current_player.d == 'D') {
      current_player = battle(st.top(), current_player);
      st.pop();
    }
    if (current_player.d == 'U') {
      st.push(current_player);
    } else if (current_player.d == 'D') {
      assert(st.empty());
      surviving_players[current_player.id] = current_player.h;
    }
  }
  while (!st.empty()) {
    surviving_players[st.top().id] = st.top().h;
    st.pop();
  }

  for (int i = 0; i < n; ++i) {
    if (surviving_players[i] != -1) {
      cout << surviving_players[i] << "\n";
    }
  }
}
```

Personal comment: the overall solution works in $\mathcal{O}(n \log n)$; however, the constant itself was made strict as stack, the main solution, works in $\mathcal{O}(n)$. The sorting algorithm dominates the runtime of the overall algorithm. The constraint of $p_i, \forall i \in \{1, \ldots, n\}$ should have been lower than or equal to 10^6 instead of 10^8 so that we can use the counting sort algorithm, which can be well-applied in $\mathcal{O}(n)$.

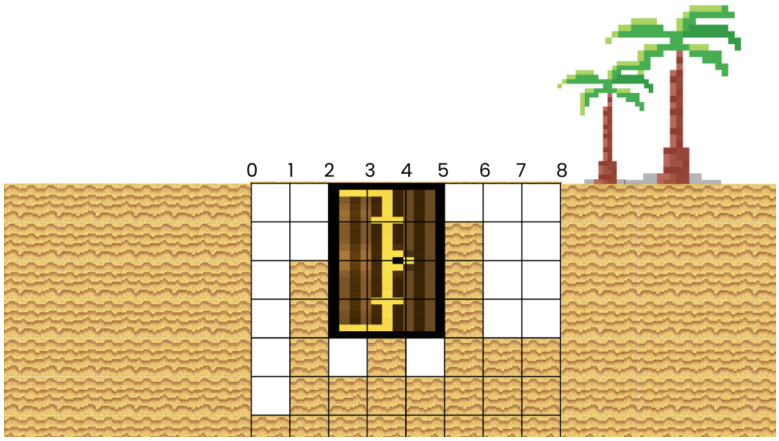
Buried Treasure

Written and developed by: **Yohandi**

Editorial by: **Yohandi**

Description Jack Sparrow, the legendary pirate of the Seven Seas, sets sail to an inhabited island in search of a buried treasure. Guided by his map, he believes the treasure lies within a weird-looking trench. The trench stretches across a width of n. For any given point from $i1$ to i on the trench's x-coordinate, the depth is represented by d_i (for each i such that $1 \leq i \leq n$, where i is an integer).

Jack is wondering about the size of the largest treasure that could possibly fit inside the trench. Wanting to maximize his haul, he turns to you, a trusted member of his crew, to make the calculations. By largest, Jack means the maximum area – the product of width and height – of the rectangular treasure chest that can be buried within the trench's confines. For example, the following figure shows the largest possible treasure that can fit in a trench with $n = 8$ and $d = [6, 2, 5, 4, 5, 1, 4, 4]$.



Could you give these calculations a look for our legendary pirate?

Constraints

- $1 \leq T \leq 10$
- $1 \leq n \leq 10^6$
- $1 \leq d_i \leq 10^6, \forall i \in \{1, \ldots, n\}$
- The sum of n over all queries, denoted as N, does not exceed 10^6 in each test case.

Solution This problem is a classic "Maximum Rectangle Area in Histogram" problem with a rephrased and modified statement. However, the proof of optimal chest rotation angles is only between $0°, 90°, 180°, 270°$ is still needed to correctly solve this problem.

The approach is to have a stack that maintains the indices of depths in ascending order so that we can identify the potential of extension (starting points) for rectangles. Each of those rectangles is computed as $A_i = d_i \times (r_i - l_i + 1)$, where r_i is the rightmost index to which the rectangle containing the i-th index can extend (same goes for l_i but on the other direction). The main task is to find $\max_i A_i$.

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

long long largest_chest(int n, vector<int> &v) {
  long long ret = 0;

  vector<int> left_most(n, -1), right_most(n, -1);
  stack<int> st;
```

```
  for (int i = 0; i < n; ++i) {
    if (st.empty() || v[st.top()] <= v[i]) {
      st.push(i);
      continue;
    }

    while (!st.empty() && v[st.top()] > v[i]) {
      right_most[st.top()] = i - 1;
      st.pop();
    }
    st.push(i);
  }
  while (!st.empty()) {
    right_most[st.top()] = n - 1;
    st.pop();
  }

  for (int i = n - 1; i >= 0; --i) {
    if (st.empty() || v[st.top()] <= v[i]) {
      st.push(i);
      continue;
    }

    while (!st.empty() && v[st.top()] > v[i]) {
      left_most[st.top()] = i + 1;
      st.pop();
    }
    st.push(i);
  }
  while (!st.empty()) {
    left_most[st.top()] = 0;
    st.pop();
  }

  for (int i = 0; i < n; ++i) {
    ret = max(ret, (long long)v[i] * (right_most[i] - left_most[i] + 1));
  }

  return ret;
}

int main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);

  int T;
  cin >> T;

  for (int tc = 0; tc < T; ++tc) {
    int n;
    cin >> n;

    vector<int> v(n);
    for (int i = 0; i < n; ++i) {
      cin >> v[i];
    }

    cout << largest_chest(n, v) << "\n";
  }
}
```

Assignment 3

Node Distance

Written and developed by: **Yige Jiang**

Editorial by: **Yohandi**

Description You are given a tree with n nodes, where each edge in the tree has a corresponding weight denoting the length of each edge. The nodes in the tree are colored either black or white. Your task is to calculate the sum of distances between every pair of black nodes in the tree. Let $B = \{b_1, b_2, \ldots\}$ a set of black nodes, then the answer is formulated as:

$$\sum_{i=1}^{|B|1} \sum_{j=i+1}^{|B|} \text{dist}(b_i, b_j)$$

where $|B|$ denotes the number of the black nodes in the tree, and $\text{dist}(b_i, b_j)$ is the length of the simple path from the i-th to j-th black node.

Write a program to calculate the formulation above.

Constraints

- $1 \leq n \leq 10^5$
- $c_i \in \{0, 1\}, \forall i \in \{1, \ldots, n\}$
- $1 \leq q_{p-1} < p, \forall p \in \{1, \ldots, n-1\}$
- $1 \leq w_p \leq 1\,000, \forall p \in \{1, \ldots, n-1\}$

Solution Suppose we focus on the contribution of each edge to the total distance. An edge connecting nodes u and v, when removed, splits the tree into two subtrees, say they are T_1 and T_2. The key insight here is that this edge contributes to the total distance exactly the product of the number of black nodes in T_1 and the number of black nodes in T_2.

The implementation is done by performing a DFS from any fixed root. During DFS, we count the number of black nodes in each subtree. For a subtree rooted at u, the count of black nodes in T_1 is the number of black nodes within this subtree. The count for T_2 is determined by subtracting the count of black nodes in T_1 from the total number of black nodes in the tree.

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

int main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);

  int n;
  cin >> n;
  vector<int> a(n + 1, 0), subtree_size(n + 1, 0);
  for (int i = 1; i <= n; ++i) {
    cin >> a[i];
    subtree_size[i] = a[i];
  }

  vector<vector<pair<int, long long>>> edges(n + 1);
  for (int i = 1, q, w; i < n; ++i) {
    cin >> q >> w;
    edges[q].emplace_back(i + 1, w);
  }

  long long ans = 0, black_nodes = accumulate(a.begin(), a.end(), 0);

  function<void(int, long long)> DFS = [&](int u, long long weight) {
```

```
    for (auto [v, w] : edges[u]) {
      DFS(v, w);
      subtree_size[u] += subtree_size[v];
    }
    ans += subtree_size[u] * (black_nodes - subtree_size[u]) * weight;
  };

  DFS(1, 0);
  cout << ans << endl;

  return 0;
}
```

Price Sequence

Written and developed by: **Ruiying Liu**

Editorial by: **Yohandi**

Description Mario bought n math books and he recorded their prices. The prices are all integers, and the price sequence is $a = \{a_0, a_1, ..., a_i, ..., a_{n1}\}$ of length n. Please help him to manage this price sequence. There are three types of operations: - `BUY x`: buy a new book with price x, thus x is added at the end of a. - `CLOSEST_ADJ_PRICE`: output the minimum absolute difference between adjacent prices. - `CLOSEST_PRICE`: output the absolute difference between the two closest prices in the entire sequence.

A total of m operations are performed. Each operation is one of the three mentioned types. You need to write a program to perform given operations. For operations `CLOSEST_ADJ_PRICE` and `CLOSEST_PRICE` you need to output the corresponding answers.

Constraints

- $2 \leq n, m \leq 100\,000$
- $0 \leq a_i \leq 10^{12}, \forall i \in \{0, \ldots, n-1\}$
- $0 \leq x \leq 10^{12}$

Solution Both the `CLOSEST_ADJ_PRICE` and `CLOSEST_PRICE` operations are independent to each other; hence, for each update, i.e., the `BUY x` operation, we can perform a separate update on two different data structures. - For the `CLOSEST_ADJ_PRICE` operation, we maintain two variables: the price of the last book bought and the minimum absolute difference between adjacent prices. When a `BUY x` operation occurs, update the last book's price to `x` and, if necessary, update the minimum absolute difference using the absolute difference between `x` and the previously last book's price. - For the `CLOSEST_PRICE` operation, we use a balanced binary search tree to avoid the linear time complexity in the insertion worst case. Upon inserting a new price `x`, we compare it with its immediate predecessor and successor in the tree to update the minimum absolute difference between any two prices in the sequence.

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

int main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  cout.tie(0);

  int n, m;
  cin >> n >> m;

  long long closest_adj_price = LLONG_MAX, closest_price = LLONG_MAX;
  vector<long long> a(n);
  multiset<long long> a_rb;
  for (int i = 0; i < n; ++i) {
    cin >> a[i];
    a_rb.insert(a[i]);
```

```
    if (i > 0)
      closest_adj_price = min(closest_adj_price, abs(a[i] - a[i - 1]));
  }

  for (auto it = next(a_rb.begin()); it != a_rb.end(); ++it) {
    auto prev_it = prev(it);
    closest_price = min(closest_price, abs(*it - *prev_it));
  }

  function<void(long long)> update = [&](long long x) {
    closest_adj_price = min(closest_adj_price, abs(x - a.back()));
    a.push_back(x);

    auto it = a_rb.insert(x);
    if (it != a_rb.begin())
      closest_price = min(closest_price, abs(x - *prev(it)));
    if (next(it) != a_rb.end())
      closest_price = min(closest_price, abs(*next(it) - x));

    return;
  };

  for (int i = 0; i < m; ++i) {
    string op;
    cin >> op;

    if (op == "BUY") {
      long long x;
      cin >> x;

      update(x);
    } else if (op == "CLOSEST_ADJ_PRICE") {
      cout << closest_adj_price << "\n";
    } else if (op == "CLOSEST_PRICE") {
      cout << closest_price << "\n";
    }
  }

  return 0;
}
```

Assignment 4

Divine Ingenuity

Written and developed by: **Shu Wang**

Editorial by: **Yohandi**

Description If you have ever played Genshin Impact, you probably know about the "Divine Ingenuity: Collector's Chapter" event. In this event, players can create custom domains by arranging components, including props and traps, between the given starting point and exit point.

Paimon does not want to design a difficult domain; she pursues the ultimate "automatic map". In the given domain with a size of $m \times n$, she only placed *Airflow* and *Spikes*. Specifically, Spikes will eliminate the player (represented by `x`), while the Airflow will blow the player to the next position according to the wind direction (up, left, down, right represented by `w`, `a`, `s`, `d`, respectively).

The starting point and exit point are denoted by `i` and `j`, respectively. Ideally, in Paimon's domain, the player selects a direction and advances one position initially; afterward, the Airflow propels the player to the endpoint without falling into the Spikes. The player will achieve automatic clearance in such a domain.

However, Paimon, in her slight oversight, failed to create a domain that allows players to achieve automatic clearance. Please assist Paimon by making the minimum adjustments to her design to achieve automatic clearance.

Given that the positions of the starting point and exit point are fixed, you can only adjust components at other locations. You have the option to remove existing component at any position; then, place a new direction of Airflow, or position a Spikes.

Constraints

- $3 \leq m, n \leq 2\,000$

Solution This problem can be modeled as a graph problem, where: - For nodes corresponding to *Airflow*, an edge is created to the node in the direction of the Airflow with weight 0, as no adjustment is needed. For nodes in directions other than the Airflow, edges are created to the adjacent nodes in their respective directions with weight 1, indicating that one adjustment (changing the direction of the Airflow) is required. - For *Spikes* nodes are connected to all adjacent nodes with weight 1, as a Spike can be replaced with an Airflow in any direction with one adjustment.

We run Dijkstra's algorithm to find the shortest path from a node with the character `i` to a node with the character `j`.

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

int main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);

  int m, n;
  cin >> m >> n;

  function<int(int, int)> hash = [&](int x, int y) { return x * (n + 2) + y; };

  int start, end;
  vector<vector<char>> grid(m, vector<char>(n));
  for (int i = 1; i <= m; ++i) {
    for (int j = 1; j <= n; ++j) {
      cin >> grid[i - 1][j - 1];

      if (grid[i - 1][j - 1] == 'i') {
        start = hash(i, j);
      } else if (grid[i - 1][j - 1] == 'j') {
        end = hash(i, j);
      }
    }
  }

  Dijkstra<int> graph((m + 2) * (n + 2));
  graph.setZero(0);
  graph.setInfinite(INT_MAX);

  for (auto [dx, dy] :
       vector<pair<int, int>>{{-1, 0}, {1, 0}, {0, -1}, {0, 1}}) {
    graph.addEdge(start, hash(start / (n + 2) + dx, start % (n + 2) + dy), 0);
  }

  for (int i = 1; i <= m; ++i) {
    for (int j = 1; j <= n; ++j) {
      graph.addEdge(hash(i, j), hash(i - 1, j),
                    (int)(grid[i - 1][j - 1] != 'w'));
```

```
      graph.addEdge(hash(i, j), hash(i + 1, j),
                    (int)(grid[i - 1][j - 1] != 's'));
      graph.addEdge(hash(i, j), hash(i, j - 1),
                    (int)(grid[i - 1][j - 1] != 'a'));
      graph.addEdge(hash(i, j), hash(i, j + 1),
                    (int)(grid[i - 1][j - 1] != 'd'));
    }
  }

  cout << graph.distance(start)[end] << endl;

  return 0;
}
```

Edge Changing

Written and developed by: **Ziyi Zhao**

Editorial by: **Yohandi**

Description Give you a graph with n vertices and m edges. No two edges connect the same two vertices. For vertex ID from 1 to n, we do the following operation: If any two neighbors of a vertex have a $k\times$ relationship in terms of their IDs, we add a new edge between them. In other words, for any vertex $i = 1$ to n, if $u = kv$ or $v = ku$, we add an edge (u, v), where $u, v\text{Neighbor}(i)$. Besides, if there is already an edge between u and v, no operation is taken.

After the operation, we want you to output the BFS order starting from vertex s. Please traverse all neighbors in ascending order of their IDs when expanding a vertex.

Constraints

- $1 \leq n, m, \leq 10^5$
- $2 \leq k \leq 10^5$

Solution This problem is a straightforward BFS implementation problem with few modifications on the main graph. However, the inserted nodes need to be handled carefully as the problem requires the traversal in ascending order. One of the ways to handle this is to use a balanced binary search tree.

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

int main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  cout.tie(0);

  int n, m, k, s;
  cin >> n >> m >> k >> s;

  vector<set<int>> edges(n + 1);
  for (int i = 0; i < m; ++i) {
    int u, v;
    cin >> u >> v;
    edges[u].insert(v);
    edges[v].insert(u);
  }

  vector<pair<int, int>> pending;
  for (int u = 1; u <= n; ++u) {
    for (auto v : edges[u]) {
      if (v * k <= n && edges[u].find(v * k) != edges[u].end()) {
        pending.push_back({v, v * k});
```

```
      }
    }
  }

  for (auto [u, v] : pending) {
    edges[u].insert(v);
    edges[v].insert(u);
  }

  vector<bool> visited(n + 1, false);
  vector<int> ans;
  queue<int> BFS;

  BFS.push(s);
  visited[s] = true;
  while (!BFS.empty()) {
    int u = BFS.front();
    BFS.pop();
    ans.push_back(u);

    for (auto v : edges[u]) {
      if (!visited[v]) {
        BFS.push(v);
        visited[v] = true;
      }
    }
  }

  for (int i = 0; i < (int)ans.size(); ++i) {
    cout << ans[i] << " \n"[i + 1 == (int)ans.size()];
  }

  return 0;
}
```