Programming exercises

John Doe

April 19, 2023

Contents

1 Browser history

You have a browser of one tab where you start on the homepage and you can visit another url, get back in the history number of steps or move forward in the history number of steps.

1.1 Implement the `BrowserHistory` class:

`BrowserHistory(string homepage)` Initializes the object with the homepage of the browser.

`void visit(string url)` Visits `url` from the current page. It clears up all the forward history.

`string back(int steps)` Move `steps` back in history. If you can only return `x` steps in the history and `steps > x`, you will return only `x` steps. Return the current `url` after moving back in history at most `steps`.

`string forward(int steps)` Move `steps` forward in history. If you can only forward `x steps` in the history and `steps > x`, you will forward only `x steps`. Return the current `url` after forwarding in history at most `steps`.

1.2 Example

```
Input:
["BrowserHistory","visit","visit","visit","back","back","forward","visit","forward","ba
[["mig.mk"],["google.com"],["facebook.com"],["youtube.com"],[1],[1],[1],["linkedin.com"

Output:
[null,null,null,null,"facebook.com","google.com","facebook.com",null,"linkedin.com","go

Explanation:
BrowserHistory browserHistory = new BrowserHistory("mig.mk");
browserHistory.visit("google.com");       // You are in "mig.mk". Visit "google.com"
browserHistory.visit("facebook.com");     // You are in "google.com". Visit "facebook.c
browserHistory.visit("youtube.com");      // You are in "facebook.com". Visit "youtube
browserHistory.back(1);                   // You are in "youtube.com", move back to "fa
browserHistory.back(1);                   // You are in "facebook.com", move back to "g
browserHistory.forward(1);                // You are in "google.com", move forward to "
browserHistory.visit("linkedin.com");     // You are in "facebook.com". Visit "linkedin
browserHistory.forward(2);                // You are in "linkedin.com", you cannot move
```

```
browserHistory.back(2);                  // You are in "linkedin.com", move back two s
browserHistory.back(7);                  // You are in "google.com", you can move back
```

1.3 Constraints:

- 1 <= homepage.length <= 20
- 1 <= url.length <= 20
- 1 <= steps <= 100
- homepage and url consist of '.' or lower case English letters.
- At most 5000 calls will be made to visit, back, and forward.

1.4 Analysis

One approach is to use two stacks, one for `history`, one to save the `future` pages.

1.5 Test scenario

```
#include <string>

using std::string;

int main(){
BrowserHistory history = new BrowserHistory("https://www.google.com");

// User visits a few pages
history.visit("https://www.google.com/search?q=java");
history.visit("https://www.wikipedia.org/");
history.visit("https://www.amazon.com/");

// User clicks back button once
string previousPage = history.back(1);
cout << "User is now on page: " << previousPage;

// User clicks forward button twice
string nextPage = history.forward(2);
cout << "User is now on page: ",  nextPage;
}
```

Figure 1: Test scenario

1.6 Starter code

```
#include <string>

class BrowserHistory {
public:
    BrowserHistory(string homepage){

    }
    void visit(string url){

    }
    string back(int steps){

    }
    string next(int steps){

    }
}
```

Figure 2: Starter code

1.7 Solution

```
class BrowserHistory {
    stack<string> history, future;
    string current;
public:
    BrowserHistory(string homepage) {
        // 'homepage' is the first visited URL.
        current = homepage;
    }

    void visit(string url) {
        // Push 'current' in 'history' stack and mark 'url' as 'current'.
        history.push(current);
        current = url;
        // We need to delete all entries from 'future' stack.
```

```
    future = stack<string>();
}

string back(int steps) {
    // Pop elements from 'history' stack, and push elements in 'future' stack.
    while(steps > 0 && !history.empty()) {
        future.push(current);
        current = history.top();
        history.pop();
        steps--;
    }
    return current;
}

string forward(int steps) {
    // Pop elements from 'future' stack, and push elements in 'history' stack.
    while(steps > 0 && !future.empty()) {
        history.push(current);
        current = future.top();
        future.pop();
        steps--;
    }
    return current;
}
```

Figure 3: Example solution

2 Bipartite Graph

There is an undirected graph with `n` nodes, where each node is numbered between `0` and `n-1`. You are given a 2D array graph, where `graph[u]` is an array of nodes that node `u` is adjacent to. More formally, for each `v` in `graph[u]`, there is an undirected edge between node `u` and node `v`. The graph has the following properties:

- There are no self-edges (`graph[u]` does not contain `u`).
- There are no parallel edges (`graph[u]` does not contain duplicate values).

- If `v` is in `graph[u]`, then `u` is in `graph[v]` (the graph is undirected).
- The graph may not be connected, meaning there may be two nodes `u` and `v` such that there is no path between them.

A graph is bipartite if the nodes can be partitioned into two independent sets `A` and `B` such that every edge in the graph connects a node in set `A` and a node in set `B`.

Return `true` if and only if it is bipartite.

2.1 Examples:

2.1.1 Example 1:

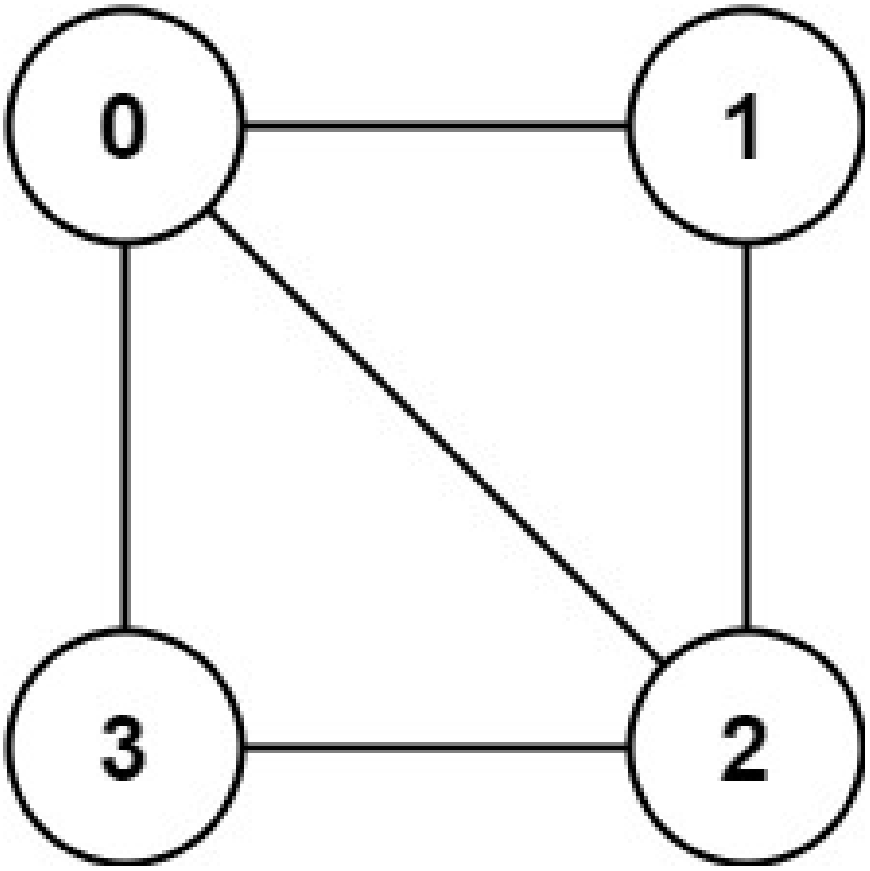

Figure 4: Example graph 1

```
Input: graph = [[1,2,3],[0,2],[0,1,3],[0,2]]
Output: false
```

Explanation There is no way to partition the nodes into two independent sets such that every edge connects a node in one and a node in the other.

2.1.2 Example 2:

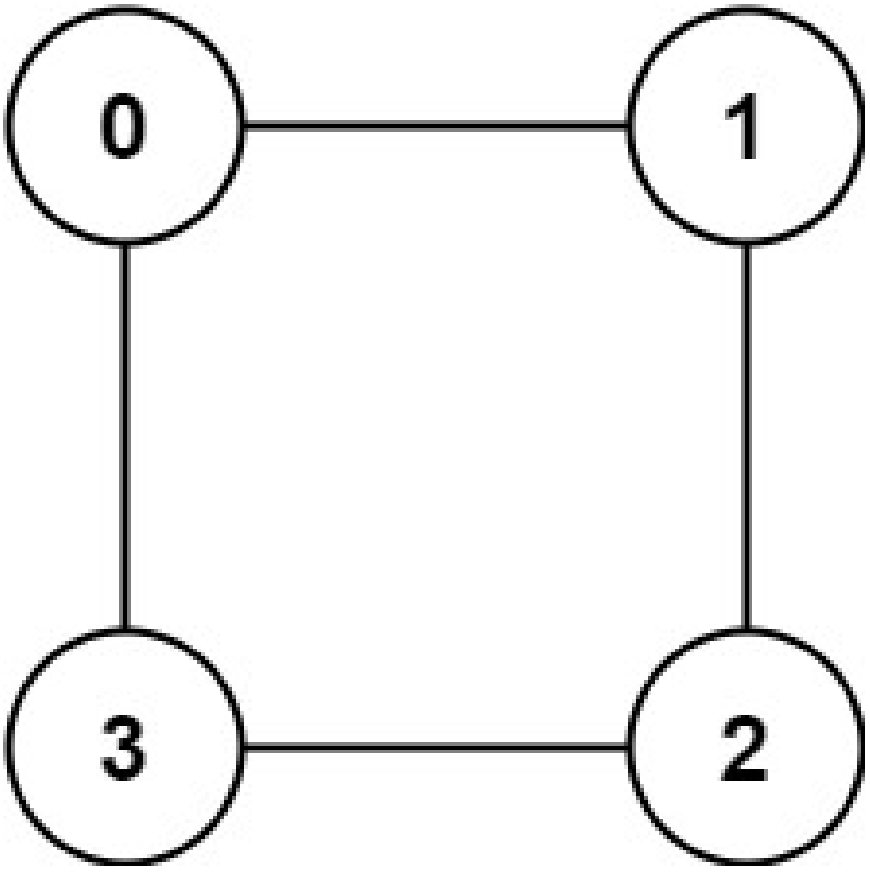

Figure 5: Example graph 2

```
Input: graph = [[1,3],[0,2],[1,3],[0,2]]
Output: true
```

Explanation: We can partition the nodes into two sets: {0, 2} and {1, 3}.

2.2 Constraints:

- `graph.length == n`
- `1 <= n <= 100`
- `0 <= graph[u].length < n`
- `0 <= graph[u][i] <= n - 1`
- `graph[u]` does not contain `u`.
- All the values of `graph[u]` are unique.
- If `graph[u]` contains `v`, then `graph[v]` contains `u`.

2.3 Analysis

The obvious approach is to use BFS or DFS to traverse the graph.

2.4 Solutions

2.4.1 Bipartite Graph solution using BFS

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

using std::vector;
using std::queue;

bool bfs(int node, int color, vector<vector<int>>& graph, vector <int>& visited) {
    queue <int> q;
    q.push(node);
    visited[node] = color;
    while (!q.empty()) {
        int curr = q.front();
        q.pop();
        for (auto adjNode: graph[curr]) {
            if (visited[adjNode] == -1) {
                q.push(adjNode);
                visited[adjNode] = !visited[curr];
            } else if (visited[adjNode] == visited[curr]) {
```

```
                return false;
            }
        }
    }
    return true;
}
bool isBipartite(vector<vector<int>>& graph) {
    int n = graph.size(); // Get the node number
    vector <int> visited(n, -1); // Create visited vector for every node
    for (int i = 0; i < n; i++) {
        if (visited[i] == -1) {
            if (bfs(i, 0, graph, visited) == false)
                return false;
        }
    }
    return true;
}
```

Similatrly, we can use DFS instead of BFS

2.4.2 Bipartite Graph solution using DFS

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

using std::vector;

bool dfs(int node, int color, vector<vector<int>>& graph, vector <int>& visited) {
    visited[node] = color;
    for (auto adjNode: graph[node]) {
        if (visited[adjNode] == -1) {
            if (dfs(adjNode, !color, graph, visited) == false)
                return false;
        } else if (visited[adjNode] == color) {
            return false;
        }
    }
    return true;
}
```

```
bool isBipartite(vector<vector<int>>& graph) {
    int n = graph.size();
    vector <int> visited(n, -1);
    for (int i = 0; i < n; i++) {
        if (visited[i] == -1) {
            if (dfs(i, 0, graph, visited) == false)
                return false;
        }
    }
    return true;
}
```

But this is a primer on how we can utilize `Union-Find`.

2.4.3 Bipartite Graph solution using Union-Find

```
#include <iostream>
#include <vector>
#include <utility> // For std::pair

using std::vector;
using std::pair;


int find(vector<pair<int,int>> &parent, int k){
    if(parent[k].first == -1 || parent[k].first==k){
        parent[k].first = k;
        return k;
    }
    parent[k].first=find(parent,parent[k].first);
    return parent[k].first;
}

void unionFind(vector<pair<int,int>> &parent, int x, int y){
   int a = find(parent,x);
    int b = find(parent,y);
    if(a != b){
       if(parent[a].second < parent[b].second){
   parent[a].first=b;
   }else if(parent[a].second > parent[b].second){
```

```
parent[b].first=a;
}else{
parent[b].first=a;
parent[a].second += 1;
}
    }
}
bool isBipartite(vector<vector<int>>& graph) {
    int n =graph.size();
    vector<pair<int,int>> parent(n,{-1,0});
    for(int i=0;i<n;i++){
        for(int j=0;j<graph[i].size();j++){
 // node and neighbour shouldn't belong to same set in bipartite
            if(find(parent,i) == find(parent,graph[i][j])){
                return false;
            }
// all node neighbours should belong to one set
            unionFind(parent,graph[i][j],graph[i][0]);
        }
    }
    return true;
}
```

3 Reverse Pairs

Given an integer array `nums`, return the number of *reverse pairs* in the array.

A *reverse pair* is a pair `(i, j)` where:

- `0 <= i < j < nums.length` and
- `nums[i] > 2 * nums[j]`.

3.1 Examples:

3.1.1 Example 1:

```
Input: nums = [1,3,2,3,1]
Output: 2
```

Explanation The reverse pairs are:

```
(1, 4) --> nums[1] = 3, nums[4] = 1, 3 > 2 * 1
(3, 4) --> nums[3] = 3, nums[4] = 1, 3 > 2 * 1
```

3.1.2 Example 2:

```
Input: nums = [2,4,3,5,1]
Output: 3
```

Explanation The reverse pairs are:

```
(1, 4) --> nums[1] = 4, nums[4] = 1, 4 > 2 * 1
(2, 4) --> nums[2] = 3, nums[4] = 1, 3 > 2 * 1
(3, 4) --> nums[3] = 5, nums[4] = 1, 5 > 2 * 1
```

3.2 Constraints:

- $1 <= \ nums.length \ <= 5 * 10^4$
- $-2^{31} <= \ nums[i] \ <= 2^{31} - 1$

3.3 Solution

3.3.1 Using Divide and Conquer and Merge sort

```
class Solution {
public:
    int inversions=0;
    int find_index_where_just_greater(int left,int right,vector<int>&arr,int target)
    {
        while(left<=right)
        {
            int mid=(left+right)/2;
            long long x=arr[mid];
            x*=2;
            if(x>=target)
            {
                right=mid-1;
            }
            else
            {
                left=mid+1;
            }
```

```
    }
    return (left-1);
}
void merge(vector<int>&arr,int l,int m,int r)
{
    vector<int>Arr(r-l+1);
    int ptr1=l,ptr2=m+1,ptr3=0;
    while(ptr3<=(r-l))
    {
        if(ptr1==(m+1))
        {
            Arr[ptr3]=arr[ptr2];
            ptr2++;
            ptr3++;
            continue;
        }
        if(ptr2==(r+1))
        {
            int idx=find_index_where_just_greater(m+1,r,arr,arr[ptr1]);
            inversions+=idx-m;
            Arr[ptr3]=arr[ptr1];
            ptr1++;
            ptr3++;
            continue;
        }
        if(arr[ptr1]<=arr[ptr2])
        {
            int idx=find_index_where_just_greater(m+1,r,arr,arr[ptr1]);
            inversions+=idx-m;
            Arr[ptr3]=arr[ptr1];
            ptr1++;
            ptr3++;
        }
        else
        {
            Arr[ptr3]=arr[ptr2];
            ptr2++;
            ptr3++;
        }
    }
```

```
        for(int i=l;i<=r;i++)
        {
            arr[i]=Arr[i-l];
        }
        return;
    }
    void mergeSort(vector<int>&arr,int l,int r)
    {
        if(l==r)
        {
            return;
        }
        int mid=(l+r)/2;
        mergeSort(arr,l,mid);
        mergeSort(arr,mid+1,r);
        merge(arr,l,mid,r);
        return;
    }
    int reversePairs(vector<int>&nums)
    {
        mergeSort(nums,0,nums.size()-1);
        return inversions;
    }
};
```

3.3.2 Using segment tree

```
int seg[1000006];
void update(int ind, int st, int en, int index) {
    if(st == en) {
        seg[ind]++;
        return;
    }
    int mid = (st + en) >> 1;
    if(index <= mid) {
        update(2 * ind + 1, st, mid, index);
    }
    else {
        update(2 * ind + 2, mid + 1, en, index);
    }
```

```
    seg[ind] = seg[2 * ind + 1] + seg[2 * ind + 2];
}
int query(int ind, int st, int en, int l, int r) {
    if(en < l || st > r || st > en) {
        return 0;
    }
    if(l <= st && en <= r) {
        return seg[ind];
    }
    int mid = (st + en) >> 1;
    return query(2 * ind + 1, st, mid, l, r) + query(2 * ind + 2, mid + 1, en, l, r);
}
int reversePairs(vector<int>& nums) {
    int n = nums.size();
    set<long long> st(nums.begin(), nums.end());
    int cnt = 0;
    unordered_map<int, int> mp;
    for(int i : st) {
        mp[i] = cnt++;
    }
    int ans = 0;
    for(int i = 0; i < n; i++) {
        long long rq = nums[i] * 2ll + 1;
        if(st.find(rq) != st.end()) {
            int l = mp[rq], r = n - 1;
            ans += query(0, 0, n - 1, l, r);
        }
        else {
            auto it = st.lower_bound(rq);
            if(it != st.end()) {
                int l = mp[*it], r = n - 1;
                ans += query(0, 0, n - 1, l, r);
            }
        }
        update(0, 0, n - 1, mp[nums[i]]);
    }
    return ans;
}
```