CS208 Lab3 Practice

12312110 李轩然

DDL: Apr.6

Problem 1

Description

Lanran has tasks to do, while the tasks are not independent. That means to do some task Lanran needs to finish other task first. Now given the dependency of the tasks, Lanran wants you to tell him one possible order to finish all these tasks. To make the output answer unique, the answer should have the **least lexicographical order** among all possible answers. If there is no possible answer, please output 'impossible' (without quotes).

Sample Input

```
5 4
1 4
1 3
4 3
1 5
```

Sample Output

```
1 2 4 3 5
```

Analysis

First, there is a possible answer if and only if the tasks construct a DAG. If it is a DAG, basically we just need to find the least lexicographical order of topological orderings. Just make sure each time the nodes which have no indegree will be pushed into the queue from the smallest to the largeest. In order to do this, we can use **priority-queue**

to contruct a **min-heap** (which has been covered in CS203, so we use STL directly), thus we can delete the smallest element in the min-heap each time easily.

C++ Code

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

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

    vector<vector<int>> g(n + 1);
    vector<int> in(n + 1, 0);

    for (int i = 0; i < m; i++)
    {
        int u, v;
        cin >> u >> v;
        g[u].push_back(v);
        in[v] ++;
    }

    priority_queue<int, vector<int>, greater<int>> pq;

    for (int i = 1; i <= n; i++)
    {
        if (in[i] == 0) pq.push(i);
    }

    vector<int> result;
    while (!pq.empty())
    {
        int u = pq.top();
        pq.pop();
        result.push_back(u);

        for (int v : g[u])
        {
            in[v] --;
            if (in[v] == 0) pq.push(v);
        }
    }
```

```
    if (result.size() != n) cout << "impossible" << endl;
    else {
        for (int i = 0; i < n; i++)
            cout << result[i];
    }

    return 0;
}
```

Complexity Analysis

- Time complexity: For processing neighbors, we have m edges in total, and for each edge, the neighbor node will be push into the min-heap which will take $O(logn)$ time, thus processing neighbors will take $O(mlogn)$ time. For deleting nodes in the min-heap, each node will take $O(logn)$ time, thus the total taken time will be $O(nlogn)$. So, the time complexity is $O((n+m)logn)$.
- Space complexity: The space of the graph g is $O(n+m)$, and min-heap's and other arrays' is all $O(n)$, thus the space complexity is $O(n+m)$.

Problem 2 Interval scheduling

Description

Interval scheduling:

- Job j starts at s_j and finishes at f_j.
- Two jobs **compatible** if they don't overlap.
- Goal: find **maximum cardinality subset** of mutually compatible jobs.

Sample Input

```
a 0 6
b 1 4
c 3 5
d 3 8
e 4 7
f 5 9
g 6 10
h 8 11
```

Sample Output

```
b e h
```

Analysis

We use **greedy algorithm**. If we have choosen the same amount of jobs, the best way is to let the maximum finish time as early as possible, so that we can choose more jobs that after this finish time. Thus, we should **sort the jobs by their finish time** from small to big, and then just choose the non-overlap jobs and we are done.

C++ Code

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

struct job {
    char name;
    int left;
    int right;
};

int main()
{
    vector<job> jobs;
    vector<char> ans;
    char ch;
    int a, b;

    while (cin >> ch >> a >> b) jobs.push_back({ch, a, b});

    // sort by finish time
    sort(jobs.begin(), jobs.end(), [](const auto& j1, const auto& j2) {
        return j1.right < j2.right;
    });

    if (jobs.empty()) return 0;

    ans.push_back(jobs[0].name);
    int last = jobs[0].right;
```

```
    for (size_t i = 1; i < jobs.size(); i++)
    {
        if (jobs[i].left >= last) {
            ans.push_back(jobs[i].name);
            last = jobs[i].right;
        }
    }

    for (size_t i = 0; i < ans.size(); i++)
        cout << ans[i] << " ";

    return 0;
}
```

Complexity Analysis

- Time complexity: the sorting takes $O(nlogn)$ time, and the choosing by greedy takes $O(n)$ time, so the total time complexity is $O(nlogn)$.
- Space complexity: Obviously $O(n)$.