Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №4

по «Алгоритмам и структурам данных»

Базовые задачи / Timus

Выполнил:

Студент группы P3211

Болорболд А.

Преподаватели:

Косяков М. С.

Сосновцев Г. А.

Санкт-Петербург

2024

Задача №13 «Цивилизация»

1. *Краткое описание алгоритма:*

Классический пример использования алгоритмы Дейкстры, который находит кратчайший путь из возможных на взвешенном графе. Для дальнейшей оптимизации здесь используется приоритетная очередь.

Исходный код:

```
#include <algorithm>
#include <climits>
#include <iostream>
#include <queue>
#include <string>
#include <vector>

int main() {
    int n, m, x, y, dx, dy, xPath, yPath, xInit, yInit;
    std::string result;
    std::cin >> n >> m >> x >> y >> dx >> dy;
    --x, --y, --dx, --dy;
    xPath = dx, yPath = dy, xInit = x, yInit = y;
    if(x == dx && y == dy) {
        std::cout << 0 << std::endl;
    }
    std::vector<std::string> seed(n);
    for(int i = 0; i < n; ++i) {
        std::cin >>seed[i];
    }
    std::vector<std::vector<int>> path(n, std::vector<int>(m, INT_MAX));
    std::vector<std::vector<std::pair<int, int>>>
parent(n,std::vector<std::pair<int, int>>(m, {-1, -1}));

    auto axisComparator = [&](const std::pair<int, int>& s1, const
std::pair<int, int>& s2) {
        return path[s1.first][s1.second] > path[s2.first][s2.second];
    };

    std::priority_queue<std::pair<int, int>, std::vector<std::pair<int, int>>,
decltype(axisComparator)> vacant(axisComparator);
    path[x][y] = 0;

    vacant.push({x, y});
    int abscissa[] = {1, -1, 0, 0};
    int ordinate[] = {0, 0, 1, -1};
    char compass[] = {'N', 'S', 'W', 'E'};

    while(!vacant.empty()) {
        x = vacant.top().first;
        y = vacant.top().second;
        vacant.pop();
        if(x == dx && y == dy) {
            break;
        }
        for(int i = 0; i < 4; ++i) {
            int xNext = x + abscissa[i];
            int yNext = y + ordinate[i];
```

```
            if(xNext >= 0 && yNext >= 0 && xNext < n && yNext < m &&
seed[xNext][yNext] != '#') {
                int currentCost = path[x][y] + (seed[xNext][yNext] == '.' ? 1 :
(seed[xNext][yNext] == 'W' ? 2 : 0));
                if (currentCost < path[xNext][yNext]) {
                    path[xNext][yNext] = currentCost;
                    parent[xNext][yNext] = {x, y};
                    vacant.push({xNext, yNext});
                }
            }
        }
    }
    if(path[dx][dy] == INT_MAX) {
        std::cout << -1 << std::endl;
        return 0;
    }

    while(xPath != xInit || yPath != yInit) {
        int xPathAfter = parent[xPath][yPath].first;
        int yPathAfter = parent[xPath][yPath].second;
        for(int i = 0; i < 4; ++i) {
            if(xPathAfter == xPath + abscissa[i] && yPathAfter == yPath +
ordinate[i]) {
                result.push_back(compass[i]);
            }
        }
        xPath = xPathAfter;
        yPath = yPathAfter;
    }
    std::reverse(result.begin(), result.end());
    std::cout << path[dx][dy] << "\n" << result;
}
```

2. *Корректность применяемого алгоритма:*

Если точки отправления и прибытия совпадают, то ответ будет 0. Иначе создаем матрицу стоимости достижения каждой клетки из начальной точки *vector<vector> path*. Изначально все клетки имеют стоимость *INT_MAX*, кроме начальной, стоимость которой равна 0. Кроме того, для восстановления пути поселенца в виде последовательности символов сохраняем информацию о том, откуда поселенец пришел в каждую клетку – будем использовать *vector<vector<pair>> parent* чтобы для каждой клетки хранить «родителя».

Для обхода соседей в алгоритме используем приоритетную очередь с компаратором по возрастанию стоимости клеток. Каждая клетка в очереди представляет собой объект, содержащий текущую стоимость пути до нее, а также ее координаты. Вершина клетки отправления имеет цену 0, добавляем её в очередь.

В основном цикле алгоритма Дейкстры (пока очередь не станет пустым) удаляем верхнюю вершину. Если она совпадает с вершиной назначения, то мы выходим из цикла – мы достигли точки назначения. Иначе проверяем соседние вершины: если клетка в пределах карты и не вода, то считаем её стоимость, учитывая местность перехода (если ‘.’, то поле: 1, если ‘W’, то лес: 2). Если текущий путь более

«дешевый» — обновляем стоимость и «родителя», добавляем новую вершину в очередь. Готово, алгоритм Дейкстры выполнен.

Если стоимость вершины назначения осталась бесконечной, значит дойти из начальной клетки в конечную невозможно — выводим –1. Иначе выводим стоимость этой клетки (то есть искомое количество единиц времени) и формируем строку, задающую маршрут поселенца. Начиная от клетки назначения, переходим по соответствующим клеткам из *parent*, после чего инвертируем строку и получаем нужную последовательность переходов поселенца — задача решена.

3. *Оценка сложности по времени:*

Каждая клетка карты — вершина графа. Всего вершин *n * m*. Каждая вершина может иметь до 4 рёбер, поэтому общее число рёбер приблизительно равно *4 * n * m*.

Рассмотрим операции вставки и извлечения из очереди с приоритетами. В наихудшем случае каждая вершина (клетка карты) может быть добавлена в очередь при каждом рассмотрении её соседей:

- добавление в очередь (*push*) имеет сложность *O(log(n * m))*;
- извлечение из очереди (*pop*) также имеет сложность *O(log(n * m))*. В худшем случае, каждая клетка может быть обработана до четырёх раз (один раз за каждого из четырёх соседей), что приводит к частым обновлениям приоритетов.

В итоге, каждая *push* или *pop* операции стоят O*(log(n * m))*, и если учесть, что каждая вершина может быть рассмотрена несколько раз, то в худшем случае сложность алгоритма по времени должна составлять **O(n * m * log(n * m))**.

4. *Оценка сложности по памяти:*

Программа хранит некоторое постоянное число переменных, а также:

- карту размера *n * m* в виде вектора строк, что требует *O(n * m)* памяти;
- матрицу стоимостей, тоже требующую *O(n * m)* памяти;
- матрицу родителей, которая также требует *O(n * m)* памяти;
- очередь с приоритетами, которая в худшем может содержать все клетки карты, т.е. *O(n * m)*.

Значит, сложность алгоритма по памяти в худшем случае должна составлять

O(n * m).

Задача №14 «Свинки-копилки»

1. *Краткое описание алгоритма:*

Задачу можно сформулировать как задачу о поиске количества компонент связности в графе, где узлы графа — это копилки, а ребра указывают, какой ключ где

лежит. Каждая компонента связности обозначает группу копилок, в которой можно достать все ключи, начиная с разбивания одной копилки в этой компоненте.

Исходный код:

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

void dfs(int v, bool* broken, std::vector<std::vector<int>>* graph) {
    broken[v] = true;
    for(int piggy : (*graph)[v]) {
        if(!broken[piggy]) {
            dfs(piggy, broken, graph);
        }
    }
}

int main() {
    int n, piggy, result = 0;
    std::cin >> n;
    std::vector<std::vector<int>> pigs(n);
    bool broken[n];
    for(int i = 0; i < n; ++i) {
        std::cin >> piggy;
        pigs[piggy - 1].push_back(i);
        pigs[i].push_back(piggy - 1);
        broken[i] = false;
    }

    for(int i = 0; i < n; ++i) {
        if(!broken[i]) {
            ++result;
            dfs(i, broken, &pigs);
        }
    }
    std::cout << result << "\n";
}
```

2. *Корректность применяемого алгоритма:*

По условию задачи копилки с ключами образуют некоторые группы, в которых, разбив одну копилку, можно получить ключ и, открывая всё новые и новые копилки, получить содержимое всей группы. Очевидно, что таких групп может быть несколько. Тогда минимальное число свинок-копилок, которые нужно разбить, чтобы достать ключи из всех копилок — это количество таких групп копилок, связи (то есть ключи от других копилок) между которыми есть только в этой группе.

Итак, минимальное количество копилок, которые необходимо разбить, равно количеству компонент связности в графе, потому что в каждой компоненте связности достаточно «разбить» одну копилку, чтобы получить ключи для всех остальных копилок в той же компоненте.

Сначала мы считываем общее количество копилок *n* и создаем граф в виде списка смежности. Для каждой копилки создаётся вектор, где *pigs[i]* содержит индексы копилок, с которыми копилка *i* связана ключами. То есть, если ключ от

копилки *i* лежит в копилке *j*, то в граф добавляются два ребра: $i \to j$ *и* $j \to i$, указывающие взаимосвязь между копилками. Также используем массив *broken*, индексируемый номерами копилок, который показывает, была ли копилка уже «разбита» (посещена).

Далее для каждой копилки (если она ещё не была «разбита»), мы будем обходить все связанные с ней копилки в глубину и разбивать их. Все разбитые за эту итерацию копилки – это копилки одной группы, то есть компонента связности графа. Таким образом за каждую итерацию по неразбитым копилкам мы находим еще одну компоненту связности графа. Очевидно, что, посчитав количество таких итераций, мы узнаем количество компонент связности в графе — это и есть искомое минимальное количество копилок, которые необходимо разбить.

3. *Оценка сложности по времени:*

Во время ввода данных мы проходим через *n* копилок и для каждой устанавливает двунаправленное ребро между копилками. Это занимает константное время *amortized O(1)* на каждую итерацию (добавление элемента в вектор), и общее время для всех копилок будет *O(n)*.

Во время обхода графа и поиска компонент связности, каждая вершина (копилка) и каждое ребро графа будут рассмотрены ровно один раз в процессе поиска в глубину. Так как у каждой копилки максимум два связанных ребра (отношение ключей), общее количество рёбер в графе будет порядка *O(n)*.

Таким образом, общее время, затрачиваемое на поиск в глубину для обхода всех вершин и рёбер, составит *O(v + e)*, где *v* — количество вершин (копилок), а *e* — количество рёбер. Поскольку $v = n$ *и* $e \approx n$, в среднем и худшем случаях сложность по времени должна составлять **O(n)**.

4. *Оценка сложности по памяти:*

Программа хранит граф в виде вектора векторов, где каждый элемент верхнего уровня соответствует копилке. Поскольку каждая копилка связывается с двумя другими копилками, общее количество элементов в структуре данных составляет примерно *2 * n*, но основное хранилище ограничивается *n* векторами. Также хранится массив broken, используемый для отметки посещенных копилок, который содержит *n* элементов. Кроме того, программа хранит некоторое постоянное число переменных. Таким образом, в среднем и худшем случаях сложность по памяти должна составлять **O(n)**.

Задача №15 «Долой списывание!»

1. *Краткое описание алгоритма:*

По условию задачи у нас есть граф, где вершина – это лкшонок, а ребро – это обмен запиской. Требуется проверить, что можно разделить вершины на группы таким образом, чтобы между вершинами каждой группы не было ребер. Таким

образом, задачу можно смоделировать как задачу о проверке графа на двудольность.

Исходный код:

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

bool bipartite_dfs(std::vector<std::vector<int>>& classroom, int table,
std::vector<int>& depth) {
    for(auto itr : classroom[table]) {
        if(depth[itr] == -1) {
            depth[itr] = 1 - depth[table];
            if(!bipartite_dfs(classroom, itr, depth)) {
                return false;
            }
        } else if(depth[itr] == depth[table]) {
            return false;
        }
    }
    return true;
}

int main() {
    int n, m, copier, provider;
    bool result = true;
    std::cin >> n >> m;
    std::vector<std::vector<int>> students(n);
    for(int i = 0; i < m; ++i) {
        std::cin >> copier >> provider;
        students[copier - 1].push_back(provider - 1);
        students[provider - 1].push_back(copier - 1);
    }
    std::vector<int> depth(n, -1);
    for(int i = 0; i < n; ++i) {
        if(depth[i] == -1) {
            depth[i] = 1;
            if(!bipartite_dfs(students, i, depth)) {
                result = false;
            }
        }
    }
    (result) ? std::cout << "YES" : std::cout << "NO";
}
```

2. *Корректность применяемого алгоритма:*

Двудольный граф — это граф, вершины которого можно разделить на два множества таким образом, чтобы рёбра соединяли только вершины из разных групп. В контексте задачи это означает, что ученики могут быть разделены на две группы — одна группа «списывает», а другая «даёт списывать».

Часто в контексте двудольных графов используется понятие цвета вершины. Хроматическим числом графа называется минимальное количество цветов, в которые можно покрасить его вершины так, чтобы каждое ребро соединяло вершины различного цвета. Хроматическое число двудольных графов равно 2. Поэтому для

проверки графа на двудольность и разбития его на доли будем использовать покраску его вершин в два различных цвета. Так, предположим, есть 4 ученика и 3 пары обменов записками (1–2, 2–3, 3–4). Граф будет двудольным, так как ученики могут быть разделены на два цвета: {1, 3} и {2, 4}, где обмены происходят между этими группами. Если добавить еще одну связь (4–1), граф станет недвудольным, так как образуется цикл с нечётным числом вершин, и код выведет "NO".

Представим граф списком смежности, где *classroom[i]* содержит список учеников, с которыми ученик *i* обменивается записками. Введём вектор цветов по количеству учеников (вершин графа), изначально вершины не покрашены. Чтобы проверить, что граф двудольный, пройдемся по всем вершинам. Если вершина не покрашена, красим её в первый цвет и раскашиваем связанные с ней вершины поиском в глубину. В процессе поиска в глубину, при прохождении по каждому ребру красим следующую вершину в противоположный цвет. Если при переборе соседних вершин мы нашли вершину, уже покрашенную в тот же цвет, что и текущая, то он не является двудольным (так как образуется цикл с нечётным числом вершин), и функция возвращает false, то есть выводится "NO". Если таких «конфликтов» в раскрашивании не возникло, значит граф двудольный – учеников можно разделить на две группы так, чтобы любой обмен записками осуществлялся от лкшонка одной группы лкшонку другой группы – выводим "YES".

3. *Оценка сложности по времени:*

Для каждой из *m* пар происходит добавление в список смежности двух направлений (оба направления, так как граф неориентированный). Добавление каждой связи в вектор выполняется за *amortized O(1)*, поэтому общая сложность этого шага будет O(m). Для проверки на двудольность используется обход в глубину (DFS). Поскольку каждая вершина и каждое ребро посещаются ровно один раз, сложность DFS зависит от количества вершин *n* и рёбер *m*, где *n* — количество учеников (вершин), а *m* — количество пар обмена записками (рёбер). В итоге, сложность данного алгоритма по времени должна составлять **O(n + m)**.

4. *Оценка сложности по памяти:*

Программа хранит некоторое постоянное число переменных, а также массив и граф, представленный списком смежности, максимальное количество элементов которого составляет *2 * m* для каждого из n векторов, т.е. пространственная сложность графа *O(n + 2m) ~ O(n + m)*. Значит, сложность алгоритма по памяти должна составлять **O(n + m)**.

Задача №16 «Авиаперелёты»

1. *Краткое описание алгоритма*:

Сначала считаем количество городов n и инициализируем двумерный вектор pulkovo. Для того, чтобы определить минимальный размер бака, будем использовать бинарный поиск.

Исходный код:

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

void dfs(int v, std::vector<std::vector<int>>& pulkovo, std::vector<bool>& used,
int x) {
    used[v] = true;
    for(int i = 0; i < pulkovo.size(); ++i) {
        if(!used[i] && pulkovo[v][i] <= x) {
            dfs(i, pulkovo, used, x);
        }
    }
}

void dfs_inverse(int v, std::vector<std::vector<int>>& pulkovo,
std::vector<bool>& used, int x) {
    used[v] = true;
    for(int i = 0; i < pulkovo.size(); ++i) {
        if(!used[i] && pulkovo[i][v] <= x) {
            dfs_inverse(i, pulkovo, used, x);
        }
    }
}

bool check_graph(int x, std::vector<std::vector<int>> &pulkovo) {
    std::vector<bool> used(pulkovo.size(), false);
    dfs(0, pulkovo, used, x);

    for(int i = 0; i < pulkovo.size(); ++i) {
        if(!used[i]) {
            return false;
        }
    }
    std::vector<bool> used_inverse(pulkovo.size(), false);
    dfs_inverse(0, pulkovo, used_inverse, x);
    for(int i = 0; i < pulkovo.size(); ++i) {
        if(!used_inverse[i]) {
            return false;
        }
    }
    return true;
}

int binary_search_graph(int left, int right, std::vector<std::vector<int>>&
pulkovo) {
    while(right - left > 1) {
        int mid = (left + right) / 2;
        if(check_graph(mid, pulkovo)) {
            right = mid;
        } else {
            left = mid;
        }
    }
    return right;
}

int main() {
```

```
    int n, max, left = -1, right;
    std::cin >> n;
    std::vector<std::vector<int>> pulkovo(n, std::vector<int>(n, 0));
    for(auto &row : pulkovo) {
        for(auto &cell : row) {
            std::cin >> cell;
            max = std::max(cell, max);
        }
    }
    right = max + 1;
    std::cout << binary_search_graph(left, right, pulkovo);
    return 0;
}
```

2. *Корректность применяемого алгоритма:*

Сначала считаем количество городов n и инициализируем двумерный вектор *pulkovo* (для хранения расхода топлива между городами). Вектор *used* инициализируется в функции *check_graph* (для флагов, показывающих, возможен ли перелет при текущем топливе). Для того, чтобы определить минимальный размер бака, будем использовать бинарный поиск — минимумом в нем будет 0, а максимумом – максимальное расстояние между парой городов (очевидно, что в этом случае самолет может попасть из любого города в другой с дозаправкой, так как может преодолеть расстояние между любыми двумя связанными городами). Необходимо только написать функцию проверки достижимости всех городов при заданном уровне топлива.

Реализуем её так: заполним матрицу *used*, где указывается, возможен ли перелет между каждой парой городов при заданном объеме топлива. Учитывая это, проверим достижимость всех городов из одной точки (в функции *check_graph*), сначала в одном направлении, а затем в другом, чтобы убедиться, что любой город может быть достигнут из любого другого. Чтобы это проверить, используем поиск в глубину начиная с первого города, рекурсивно переходя к новому городу только если перелет возможен по матрице *used*. Если все города посещены, то данное количество топлива подходит и можно попытаться уменьшить его в бинарном поиске, если не посещены – то нужно увеличить количество топлива.

Результатом бинарного поиска является *минимальный необходимый объем топлива*. Таким образом, этот подход обеспечивает эффективную проверку минимально необходимого объема топлива для перелетов между городами, учитывая возможность дозаправок в пути.

3. *Оценка сложности по времени:*

Заполнение матрицы pulkovo выполняется за $O(n^2)$. Функции запускает DFS дважды (для проверки связности в прямом и обратном направлениях), каждый вызов DFS потенциально затрагивает все вершины и ребра, что занимает $O(n^2)$, так как в худшем случае нужно проверить все соединения. Бинарный поиск выполняется в диапазоне от 0 до m, где m — максимальное расстояние между двумя связанными

городами, то есть имеет сложность *O(log(m))*. Поскольку в худшем случае значение от binary_search_graph могло быть равно максимальному значению в матрице pulkovo, количество шагов бинарного поиска может быть $\log(10^9) \approx 30$. Таким образом, в среднем и худшем случаях сложность алгоритма по времени должна составлять **O(log(m) * n^2)**.

4. *Оценка сложности по памяти:*

Программа использует фиксированное количество переменных, а также двумерный вектор pulkovo, в которых n^2 элементов и массив visited длинной *n* символов, поэтому сложность по памяти должна составлять **O(n^2)**.

Задача №1160 «Сеть»

1. *Краткое описание алгоритма:*

Так как в задаче запрещается пересечение путей, здесь разумно использовать такую структуру данных, как система непересекающихся множеств.

Исходный код:

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

std::vector<int> dsu(1488);

int disjoint_set(int v) {
    if(v == dsu[v]) {
        return v;
    } else {
        return (dsu[v] = disjoint_set(dsu[v]));
    }
}

void dsu_merge(int a, int b) {
    a = disjoint_set(a), b = disjoint_set(b);
    if(a != b) {
        dsu[a] = b;
    }
}

int main() {
    int n, m, max = 0;
    std::cin >> n >> m;
    std::vector<std::pair<int, std::pair<int, int>>> network(m);
    std::vector<std::pair<int, int>> result;
    for(int i = 0; i < m; ++i) {
        std::cin >> network[i].second.first >> network[i].second.second >>
network[i].first;
    }
    std::sort(network.begin(), network.end());
    dsu.resize(n);
    for(int i = 0; i < n; ++i) {
        dsu[i] = i;
    }
```

```
    for(int i = 0; i < m; ++i) {
        int a = network[i].second.first, b = network[i].second.second, l =
network[i].first;
        if(disjoint_set(a) != disjoint_set(b)) {
            max = std::max(max, l);
            result.push_back(network[i].second);
            dsu_merge(a, b);
        }
    }
    std::cout << max << "\n" << result.size() << "\n";
    for(auto & i : result) {
        std::cout << i.first << " " << i.second << "\n";
    }
    return 0;
}
```

2. *Корректность применяемого алгоритма:*

Задача заключается в нахождении минимального покрывающего дерева, точнее его длины. Для этого здесь я решил использовать такую структуру данных, как система непересекающихся множеств.

Сначала идёт сортировка — это делается для того, чтобы уже получить минимальную возможную длину в нашем планировании, и для того, чтобы пройтись по всем возможным узлам сети в корректном порядке (в порядке возрастания длины кабеля). Потом заполняется наш вектор возрастающими значениями, чтобы провести операцию создания непересекающихся множеств над ним и над вектором пар (точнее, где содержатся возможные порты).

Решение проблемы с помощью СНМ работает именно благодаря тому, что это подходит к условию задач. В самом задаче есть такие требовании, как:

- Не каждый узел может быть связан с каждым другим из-за проблемы с совместимостью и геометрических ограничении; (то есть не происходит повторных пересоединении).
- Узел не может быть связан с самим собой как уроборос;
- Всегда будет хотя бы один граф, который покрывает все узлы.

Что на самом деле очень тяжко намекают на использование СНМ. Можно ещё использовать алгоритм Прима или Краскала, но лично решил научиться чему-то новому. СНМ позволяет нам найти самые большие непересекающиеся наборы в довольно незначительное количество времени, что в нашем случае является наборами узел. Это способствует правильному решению данной задачи.

3. *Оценка сложности по времени:*

Здесь присутствует интросорт (который составляет $O(n*logn)$), но внимания будем обращать на сам алгоритм СНМ. При необходимых оптимизации она составляет $O(\alpha(n))$, где α — обратная функция Аккермана. Она возрастает очень медленно, на

самом деле она не превышает 4 при $n < 10^{600}$. Сама реализация ничем не отличается, поэтому сложность алгоритма по времени должна составлять **O(n*logn)**.

4. *Оценка сложности по памяти:*

Самое значимое что есть (то есть мы не будем даже рассматривать переменные или малые вектора) — вектор сети, которая содержит в себе пару. В общем случае, сложность алгоритма по памяти должна составлять **O(2 * n) ~ *O(n)***.

Задача №1329 «Галактическая история»

1. *Краткое объяснение алгоритма:*

На каждом запросе подниматься вверх по родителям — TL, хранить у каждой вершины список её родителей — ML, поэтому я решил использовать LCA (Наименьший общий предок). Сам алгоритм составляется из несколько шагов.

Исходный код:

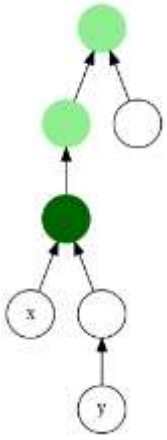
```
#include <algorithm>
#include <iostream>
#include <vector>
#include <unordered_map>

std::vector<int> lca_vert, lca_dfs_list, lca_position, lca_tree;
std::vector<char> lca_dfs_used;
std::unordered_map<int, int> indices;

void lca_dfs(const std::vector<std::vector<int>> &history, int v, int h = 1) {
    lca_dfs_used[v] = true;
    lca_vert[v] = h;
    lca_dfs_list.push_back(v);
    for (auto i = history[v].begin(); i != history[v].end(); ++i) {
        if (!lca_dfs_used[*i]) {
            lca_dfs(history, *i, h + 1);
            lca_dfs_list.push_back(v);
        }
    }
}

void lca_build_tree(int i, int l, int r) {
    if (l == r) {
        lca_tree[i] = lca_dfs_list[l];
    } else {
        int mid = (l + r) / 2;
        lca_build_tree(i + i, l, mid);
        lca_build_tree(i + i + 1, mid + 1, r);
        (lca_vert[lca_tree[2 * i]] < lca_vert[lca_tree[2 * i + 1]]) ?
lca_tree[i] = lca_tree[2 * i] : lca_tree[i] = lca_tree[i + i + 1];
    }
}

void lca_preface(const std::vector<std::vector<int>> &history, int root) {
    int n = (int) history.size();
    lca_vert.resize(n);
    lca_dfs_list.reserve(n * 2);
    lca_dfs_used.assign(n, 0);
```

```
    lca_dfs(history, root);

    int m = (int) lca_dfs_list.size();
    lca_tree.assign(lca_dfs_list.size() * 4 + 1, -1);
    lca_build_tree(1, 0, m - 1);

    lca_position.assign(n, -1);
    for (int i = 0; i < m; ++i) {
        int v = lca_dfs_list[i];
        if (lca_position[v] == -1) {
            lca_position[v] = i;
        }
    }
}

int lca_tree_min(int i, int l, int r, int sl, int sr) {
    if (sl == l && sr == r) {
        return lca_tree[i];
    }
    int s_mid = (sl + sr) / 2;
    if (r <= s_mid) {
        return lca_tree_min(2 * i, l, r, sl, s_mid);
    }
    if (l > s_mid) {
        return lca_tree_min(2 * i + 1, l, r, s_mid + 1, sr);
    }
    int ans1 = lca_tree_min(2 * i, l, s_mid, sl, s_mid);
    int ans2 = lca_tree_min(2 * i + 1, s_mid + 1, r, s_mid + 1, sr);
    if(lca_vert[ans1] < lca_vert[ans2]){
        return ans1;
    } else {
        return ans2;
    }
}

int lca(int a, int b) {
    int left = lca_position[a], right = lca_position[b];
    if (left > right) {
        std::swap(left, right);
    }
    return lca_tree_min(1, left, right, 0, (int) lca_dfs_list.size() - 1);
}

int main() {
    int n, l;
    std::cin >> n;
    int maxValue = 0;
    std::unordered_multimap<int, int> milestones;
    for (int i = 0; i < n; i++) {
        int a, b;
        std::cin >> a >> b;
        maxValue = std::max(b, std::max(maxValue, a));
        milestones.emplace(b, a);
    }
    std::vector<std::vector<int>> history(maxValue + 2);
    int root = maxValue + 1;
    auto itr = milestones.begin();
    while (!milestones.empty()) {
```

```
        std::vector<int> children;
        int parent = (*itr).first;
        if (parent == -1) {
            parent = maxValue + 1;
        }
        while (itr != milestones.end()) {
            int child = (*itr).second;
            children.push_back(child);
            milestones.erase(itr);
            if (parent == -1) {
                parent = maxValue + 1;
            }
            itr = milestones.find(parent);
            if (parent == maxValue + 1){
                parent = -1;
            }
        }
        if (parent == -1) {
            parent = maxValue + 1;
        }
        history[parent] = children;
        indices.emplace(parent, history.size() - 1);
        itr = milestones.begin();
    }
    lca_preface(history, root);
    std::cin >> l;
    for (int i = 0; i < l; i++) {
        int a, b, v;
        std::cin >> a >> b;
        v = lca(a, b);
        if (v == a) {
            std::cout << 1 << std::endl;
        } else if (v == b) {
            std::cout << 2 << std::endl;
        } else {
            std::cout << 0 << std::endl;
        }
    }
    return 0;
}
```

2. *Корректность применяемого алгоритма:*

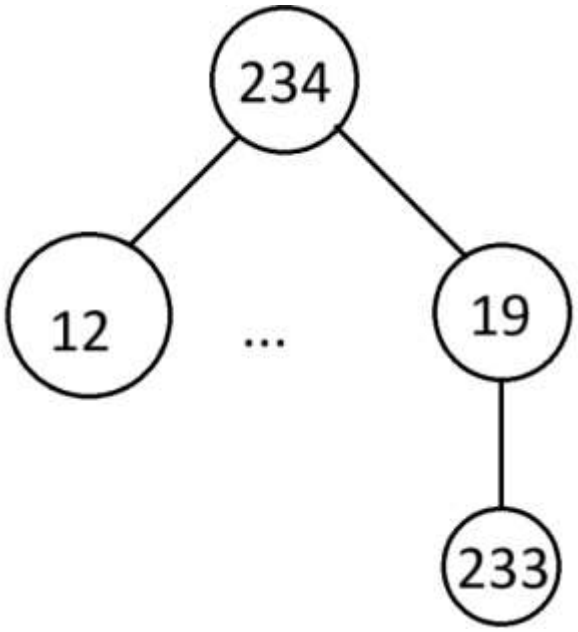
Мы можем представить историю как дерево, каждый узлом которого является веха.



Разберём пример из самого тимуса.

исходные данные	результат
10 234 -1 12 234 13 234 14 234 15 234 16 234 17 234 18 234 19 234 233 19 5 234 233 233 12 233 13 233 15 233 19	1 0 0 0 2

Построенное дерево по условию:



Чтобы найти, является ли, например, узел 234 родителем 233, надо пройти по самому дереву с самого узла. Но чтобы гарантировать корректность порядка обхода, сначала осуществляется препроцессинг. Сам препроцессинг заключается в обход в глубину, заходя в каждую вершину записывается значение её глубины в отдельно выделенный вектор. Вершины заносится в вектор по порядку обхода, а выходя из рекурсии с дочерних узлов они также заносится в лист.

Так же записывается позицию каждой вершины в массив с позициями, так что для каждой i-ой вершины *list[pos[i]] = i*, причём гарантировано, что *pos[i]* указывает на первое включение вершины на пути обхода, и строится дерево отрезков из листа с порядком обхода. Далее на каждый запрос алгоритм проходит рекурсивно по дереву отрезков, пока не найдётся общий предок. Таким образом, алгоритм гарантированно находит

3. *Оценка сложности по времени:*

Для гарантированного нахождения наименьшего общего предка требуется пройти через весь дерево. Это подготовка, который составляет *O(n)*. Для обработок самих запросов требуется *O(log(n)) * l*, где *l* — количество запросов, В итоге сложность алгоритма по времени в худшем случае должна составлять **O(n)**.

4. *Оценка сложности по памяти:*

Как было сказано заранее, алгоритм проходит через весь дерево. Обычно это означает, что сложность должна составлять *O(h)*, где *h* — глубина дерева. Но в нашем случае здесь происходит через все элементы, поэтому в принципе сложность алгоритма с НОП по памяти должна составлять O(n).

Но сама история галактики сохраняет вектор векторов, так что сложность алгоритма по памяти должна составлять $\mathbf{O(n^2)}$.