Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №3

по «Алгоритмам и структурам данных»

Базовые задачи / Timus

Выполнил:

Студент группы P3211

Болорболд А.

Преподаватели:

Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2024

Задача №9 «Машинки»

1. Пояснение к применённому алгоритму:

Чтобы решать эту задачу, надо использовать некоторые свойства структур данных. Очередь операции составляется с помощью таких действий, как pop_front(), front() и top(). Число всего операции определяется тем, находится ли машинка (которым хочет играть Петя) в конце набора.

Исходный код:

```
#include <iostream>
#include <list>
#include <queue>
#include <set>
#include <unordered_set>

int main() {
    const int max_p = 500000;
    int n, k, p, tact = 0;
    std::unordered_set<int> on_the_floor;
    std::priority_queue<std::pair<int, int>, std::vector<std::pair<int,
int>>, std::less<>> cars;

    std::cin >> n >> k >> p;

    int input[p];
    std::list<int> nodes[n];

    for (int i = 0; i < p; ++i) {
        std::cin >> input[i];
        nodes[--input[i]].push_back(i);
    }

    for (int i = 0; i < p; ++i) {
        nodes[input[i]].pop_front();
        if(on_the_floor.find(input[i]) == on_the_floor.end()) {
            tact++;
//            std::cout << i << " снять машинку " << input[i] + 1 << " ";
            if(on_the_floor.size() >= k) {
//                std::cout << "поднять машинку " << cars.top().second + 1;
                on_the_floor.erase(cars.top().second);
                cars.pop();
            }
            on_the_floor.insert(input[i]);
//            std::cout << std::endl;
        }
        int priority;
        if(nodes[input[i]].empty()) {
            priority = max_p;
        } else {
            priority = nodes[input[i]].front();
        }
        cars.emplace(priority, input[i]);
    }
    std::cout << tact;
    return 0;
}
```

2. Оценка сложности по времени:

Здесь по идее сложность по времени должна составлять O(n), но здесь надо учитывать некоторые нюансы, связанные с операциями структурами данных.

Рассмотрим первую структуру: неупорядоченный набор. Все операции (вставка, нахождение, размер) занимают постоянное время (O(1)). Разве что операции удаления занимают O(<число удаляемых элементов>), но в любом случае они не превосходит O(n). Та же самая история со списком.

Потом рассмотрим вторую структуру: приоритетная очередь. Если бы очередь построилась из n элементов сразу, то операция занял бы O(n). Если бы очередь построилась из n элементов по одному, то операция занял бы O(n*logn). В нашем алгоритме наблюдается второй сценарий, поэтому в итоге сложности по времени должна составлять **O(n*logn)**.

3. **Оценка сложности по памяти:**

Здесь нет ничего особого на самом деле, все структуры занимают **O(n)**.

Задача №10 «Гоблины и очереди»

1. **Пояснение к применённому алгоритму:**

Исходный код:

```
#include <deque>
#include <iostream>

int main() {

    int n;
    std::cin >> n;

    std::deque<std::string> q1;
    std::deque<std::string> q2;

    for (int i = 0; i < n; i++) {
        char action;
        std::cin >> action;
        if(action == '-') {
            std::cout << q1.front() << std::endl;
            q1.pop_front();
        } else {
            std::string number;
            std::cin >> number;
            if(action == '+') {
                q2.push_back(number);
            } else {
                q2.push_front(number);
            }
        }
        int diff = q1.size() - q2.size();
        if(diff == 2) {
```

```
            q2.push_front(q1.back());
            q1.pop_back();
        } else if(diff == -1) {
            q1.push_back(q2.front());
            q2.pop_front();
        }
    }
    return 0;
}
```

2. **Оценка сложности по времени:**

Опять преследуем тому же исследованию. Здесь используется очередь с двойным концом (͡° ͜ʖ ͡°), где всякие операций обычно занимает O(1). Раз только операции вставки (в этом задаче ключевая) занимает O(n), но здесь операции вставки только в концах, благодаря которому занимает O(1). В итоге, сложность алгоритма по времени должна составлять **O(n)**.

3. **Оценка сложности по памяти:**

Сложность по памяти составляет **O(n)** для хранения самих гоблинов. Для операции в алгоритме, связанных с очередями, не требуется вспомогательной памяти.

Задача №11 «Менеджер памяти-I»

1. **Пояснение к применённому алгоритму:**

Что-то вроде задачи №3 «Конфигурационный файл». Здесь пошла реализация собственных функции, где я просто совместил операций двух мапов в одну. Потом я сохранил изначальное начало, размер и конец, после чего произошли операции непосредственно с самим «запросом». Не использовать мапы для решения таких задач — это то, чего я делаю делать в далёком будущем.

Исходный код:

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

std::map<int,int> blocks_ends;
std::multimap<int,int> blocks_sz_start;

void remove_by_sz(const std::map<int, int>::iterator& it){
    auto it_sizes = blocks_sz_start.find(it -> second - it -> first);
    while (it_sizes -> second != it -> first) it_sizes++;
    blocks_sz_start.erase(it_sizes);
    blocks_ends.erase(it);
}
void remove(const std::multimap<int, int>::iterator& it){
    blocks_ends.erase(it -> second);
    blocks_sz_start.erase(it);
}
```

```cpp
void insert(const std::pair<int, int>& p){
    blocks_ends.insert(p);
    blocks_sz_start.insert({p.second - p.first, p.first});
}


int main() {

    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::cout.tie(nullptr);

    int n, m;
    std::cin >> n >> m;
    n++;
    std::vector<std::pair<int,int>> memory_table(m,{0,0});
    insert({1,n});
    for (int i = 0; i < m; ++i) {
        int query;
        std::cin >> query;
        if(query < 0){
            query *= -1;
            query--;
            if(memory_table[query].first < 0) continue;
            int start_orig = memory_table[query].first,
            sz_orig = memory_table[query].second,
            end_orig = start_orig + sz_orig;
            auto after_block = blocks_ends.lower_bound(end_orig),
            before_block = (after_block != blocks_ends.begin()) ?
prev(after_block) : blocks_ends.end();
            if(after_block != blocks_ends.end() && after_block -> first ==
end_orig) {
                if(before_block != blocks_ends.end() && before_block ->
second == start_orig) {
                    int new_start = before_block -> first;
                    int new_end = after_block -> second;
                    remove_by_sz(before_block);
                    remove_by_sz(after_block);
                    insert({new_start,new_end});
                } else {
                    int new_end = after_block -> second;
                    remove_by_sz(after_block);
                    insert({start_orig,new_end});
                }
            } else {
                if(before_block != blocks_ends.end() && before_block ->
second ==start_orig) {
                    int new_start = before_block -> first;
                    remove_by_sz(before_block);
                    insert({new_start,end_orig});
                } else {
                    insert({start_orig,end_orig});
                }
            }
        } else {
            auto it = blocks_sz_start.lower_bound(query);
            if(it == blocks_sz_start.end()) {
                memory_table[i] = std::make_pair(-1,query);
                std::cout << -1 << std::endl;
            } else {
                std::cout << it -> second << std::endl;
```

```
                int new_start = it -> second + query;
                int new_end = it -> first + it -> second;
                memory_table[i] = std::make_pair(it -> second, query);
                remove(it);
                insert({new_start,new_end});
            }
        }
    }
    return 0;
}
```

2. **Оценка сложности по времени:**

Операции с мапами стабильно составляют O(logn) из-за красно-чёрного дерева, операции с мультимапами (тот же мап, но здесь нет ограничения уникальности ключа) составляют O(n). В общем, сложность алгоритма по времени должна составлять **O(n)**.

3. **Оценка сложности по памяти:**

Для хранения самих блоков сложность алгоритма по памяти должна составлять **O(n)**. Почти все операции, связанные с мапами, требуют константной вспомогательной памяти.

Задача №12 «Минимум на отрезке»

1. **Пояснение к применённому алгоритму:**

Здесь применяется техника «скользящего окна». Оттуда просто находится минимум в каждом отрезке. В принципе задачу можно решать просто через массивы, но тогда нет смысла просто не использовать структуры данных.

Из-за того, что у меня возникли ряд проблемы с предыдущими решениями, я решил перейти к секретному оружию — использование deque для решения данной задачи.

Здесь я создал двустороннюю очередь с емкостью k, который хранит только полезные элементы текущего окна из k элементов. Элемент полезен, если он находится в текущем окне и больше, чем все остальные элементы справа от него в текущем окне. Потом постепенно обрабатываются все элементы массива один за другим и сохраняются в двустороннюю очередь, чтобы он содержал полезные элементы текущего окна, и эти полезные элементы сохраняются в отсортированном порядке. Элемент, стоящий в самом начале двусторонней очереди, является самым большим, а элемент, стоящий позади двусторонней очереди, является **самым маленьким** в текущем окне.

Исходный код:

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

```cpp
void sliding_window_min(std::vector<int> vec, int N, int K) {
    std::deque<int> window(K);
    int i;
    for (i = 0; i < K; ++i) {
        while ((!window.empty()) && vec[i] <= vec[window.back()])
            window.pop_back();
        window.push_back(i);
    }
    for (; i < N; ++i) {
        std::cout << vec[window.front()] << " ";
        while ((!window.empty()) && window.front() <= i - K)
            window.pop_front();
        while ((!window.empty()) && vec[i] <= vec[window.back()])
            window.pop_back();
        window.push_back(i);
    }
    std::cout << vec[window.front()];
}
int main() {
    int n, k;
    std::cin >> n >> k;

    std::vector<int> numbers;
    for (int i = 0; i < n; i++) {
        int num;
        std::cin >> num;
        numbers.push_back(num);
    }
    sliding_window_min(numbers, n, k);
    return 0;
}
```

2. **Оценка сложности по времени:**

По ощущению кажется, что здесь квадратичная сложность. Но можно наблюдать, что каждый элемент вектора добавляется и вытаскивается только 1 раз. Поэтому здесь на самом деле 2*n операций, и в итоге сложность алгоритма по времени должна составлять **O(n)**.

3. **Оценка сложности по памяти:**

Из-за самих значений в векторе сложность алгоритма по памяти должна составлять **O(n)**. Важно здесь ещё тот факт, что при работе двусторонней очереди ещё и используется вспомогательная память **O(k)**.

В итоге: **O(n+k)**.

Задача №1494 «Монобильярд»

1. **Пояснение к применённому алгоритму:**

Довольно простая задача, здесь используются свойства стека. Так как Чичиков забивает бильярдные шарики в лузу по порядку, уместнее всего использовать стек. Этого должен проверять ревизор: если шарик не соответствует тому, что должно было быть, то сразу же оборвётся цикл проверки, из-за которого из стека прекратится процесс вытаскивания шариков из стека, вследствие которого можно выявить, жульничал ли Чичиков или нет.

Исходный код:

```
#include <array>
#include <iostream>
#include <set>
#include <stack>
#include <unordered_set>

int main() {
    int n, p = 0;
    std::cin >> n;
    int balls[n];
    for(int i = 0; i < n; ++i) {
        std::cin >> balls[i];
    }
    std::stack<int> st;
    for(int i = 1; i <= n; i++) {
        st.push(i);
        while(!st.empty() && balls[p] == st.top()) {
            p++;
            st.pop();
        }
    }
    while(p < n) {
        if(balls[p] == st.top()) {
            p++;
            st.pop();
        } else {
            break;
        }
    }
    (st.empty()) ? std::cout << "Not a proof" : std::cout << "Cheater";
    return 0;
}
```

2. **Оценка сложности по времени:**

Все операции со стеком в алгоритме занимает O(1). Так как здесь нет какого-то нюанса с структурами данных, который может мешать нормальному оцениванию сложности по времени, алгоритм должен занимать **O(n)** времени.

3. **Оценка сложности по памяти:**

Стек — линейный тип данных, поэтому для хранения шариков сложность по памяти должна составлять **O(n)**.

Задача №1628 «Белые полосы»

1. **Пояснение к применённому алгоритму:**

Читаем координаты занятых ячеек и записываем их в вектор в виде пары координат, потом добавляем в вектор ячейки, формирующие границы календаря. Сортируем вектор так, чтобы ячейки шли по порядку своего расположения в строках (слева-направо, сверху-вниз). Проходим по порядку вектор, считая все свободные "серии" длиной > 1, и сохраняя в неупорядоченный набор все одиночные свободные дни. Далее сортируем вектор по порядку расположения в столбцах (сверху-вниз, слева-направо). Опять проходим по нему, считая все свободные "серии" длиной > 1, а, встречая одиночный свободный день, проверяем, был ли он одиночным при прошлом проходе, если да — это изолированный свободный день, и его считаем как "серии" длиной 1.

Исходный код:

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

static int m, n, k;

bool horizontal_comparator(std::pair<int, int> p1, std::pair<int, int> p2)
{
    if (p1.first < p2.first) {
        return true;
    } else if (p1.first > p2.first) {
        return false;
    } else {
        return p1.second < p2.second;
    }
}

bool vertical_comparator(std::pair<int, int> p1, std::pair<int, int> p2) {
    if(p1.second < p2.second) {
        return true;
    } else if(p1.second > p2.second) {
        return false;
    } else {
        return p1.first < p2.first;
    }
}

int main() {
    std::cin >> m >> n >> k;
    std::vector<std::pair<int, int>> occupied_days;
    for (int i = 0; i < m + 2; i++) {
        occupied_days.emplace_back(i, 0);
        occupied_days.emplace_back(i, n + 1);
    }
    for (int i = 1; i < n + 1; i++) {
        occupied_days.emplace_back(0, i);
        occupied_days.emplace_back(m + 1, i);
    }
    for (int i = 0; i < k; i++) {
        int x, y;
        std::cin >> x >> y;
        occupied_days.emplace_back((x), (y));
    }
    std::unordered_set<int> one_days;
    int weeks = 0;
    std::sort(occupied_days.begin(), occupied_days.end(),
horizontal_comparator);
```

```
    for (int i = 1; i < occupied_days.size(); i++) {
        std::pair<int, int> prev = occupied_days[i - 1];
        std::pair<int, int> curr = occupied_days[i];
        if (curr.first == prev.first) {
            int dist = curr.second - prev.second - 1;
            if (dist == 1) {
                one_days.emplace(curr.first * n + curr.second - 1);
            } else if (dist > 1) {
                weeks++;
            }
        }
    }
    std::sort(occupied_days.begin(), occupied_days.end(),
vertical_comparator);
    for (int i = 1; i < occupied_days.size(); i++) {
        std::pair<int, int> prev = occupied_days[i - 1];
        std::pair<int, int> curr = occupied_days[i];
        if (curr.second == prev.second) {
            int dist = curr.first - prev.first - 1;
            if (dist == 1) {
                auto item = one_days.find((curr.first - 1) * n +
curr.second);
                if (item != one_days.end()) {
                    weeks++;
                }
            } else if (dist > 1) {
                weeks++;
            }
        }
    }
    std::cout << weeks;
    return 0;
}
```

2. **Оценка сложности по времени:**

Опять же, посмотрим сложности операции, связанные с структурами данных.

S = n + m (измерения календаря) + k (неудачные дни).

O(S) — чтение

O(S*logS) — сортировка (introsort)

O(S) — проход по вектору

В итоге сложность алгоритма по времени должна составлять **O(S*logS)**.

3. **Оценка сложности по памяти:**

Здесь используется вектор, поэтому сложности алгоритма по памяти должна составлять **O(n)**.