Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №2

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 №5 «Коровы»

1. **Пояснение к примененному алгоритму:**

Довольно известная задача, основанная на применение бинарного поиска. После ввода работает обычный бинарный поиск, в результате которого выводится максимальное расстояние между стойлами.

Исходный код:

```
#include<iostream>

int main() {
    int n, k, curDist = 1, maxDist;
    std::cin >> n >> k;
    int stall[n];
    for(int i = 0; i < n; i++){
        std::cin >> stall[i];
    }
    maxDist = stall[n - 1] - stall[0] + 1;
    while(maxDist - 1 > curDist){
        int mid = (curDist + maxDist) / 2;
        int last = stall[0];
        int cows = 1;
        for(int i = 0; i < n; i++) {
            if(mid <= stall[i] - last) {
                last = stall[i];
                cows++;
            }
        }
        (cows >= k) ? curDist = mid : maxDist = mid;
    }
    std::cout << curDist;
    return 0;
}
```

2. **Оценка сложности по времени:**

Так как здесь применяется алгоритм бинарного поиска, сложность по времени должна составлять **O(logn)**. (Так как для нахождения элемента на n/2 требуется 1 сравнение, для n/4 и 3n/4 требуется 2 сравнения и т.д.)

3. **Оценка сложности по памяти:**

Для самого алгоритма — O(1). Но если учитывать сам массив, то в итоге получается **O(n)**.

Задача №6 «Число»

1. **Пояснение к примененному алгоритму:**

Здесь имплементирована собственный сравнительный метод. С помощью его логики получается так, что всегда получается большее значение сравниваемых строк.

Исходный код:

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

```
bool comparator(const std::string &s1, const std::string &s2) {
    return s1 + s2 > s2 + s1;
}

int main(){
    std::vector<std::string> number;
    std::string num;

    while(std::cin >> num){
        if(num.empty()) break;
        number.push_back(num);
    }

    std::sort(number.begin(), number.end(), comparator);
    for(const auto & i : number){
        std::cout << i;
    }
    return 0;
}
```

2. **Оценка сложности по времени:**

Здесь с первого взгляда — O(n). Но с учётом самой сортировки итоговая сложность по времени должна составлять **O(nlogn)**.

3. **Оценка сложности по памяти:**

С применением вектора получается **O(n)**, даже несмотря на саму сортировку, которая в теории должна составлять O(logn).

Задача №7 «Кошмар в замке»

1. **Пояснение к примененному алгоритму:**

Здесь пришлось наряду с собственным сравнительным методом имплементировать ещё и структуру, которая представляет из себя саму букву и заданный вес. После сортировки (в результате которого точно получается так, чтобы между буквами были максимальный вес) идёт конкатенация, соответствующая заданию.

Исходный код:

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

struct element {
    char letter;
    int weight;
};

bool comparator(const element a, const element b) {
    if(a.weight == b.weight) {
        return a.letter < b.letter;
    } else {
        return a.weight < b.weight;
    }
}
```

```
int main() {
    const std::string alphabet = "abcdefghijklmnopqrstuvwxyz";
    std::string str, answer;
    std::cin >> str;
    std::unordered_map<char, int> weights;
    for (int i = 0; i < 26; ++i) {
        int x;
        std::cin >> x;
        weights[alphabet[i]] = x;
    }
    std::map<char, int> taken, all;
    for (char &i : str) {
        all[i]++;
    }
    std::vector<element> word;
    for (char &i : str) {
        if (all[i] > 1 && taken[i] < 2) {
            word.push_back({i, weights[i]});
            taken[i]++;
        } else {
            answer += i;
        }
    }
    std::sort(word.begin(), word.end(), comparator);
    for (int j = 0; answer.length() != str.length(); j += 2) {
        answer = word[j].letter + answer;
        answer += word[j + 1].letter;
    }
    std::cout << answer;
    return 0;
}
```

2. **Оценка сложности по времени:**

С учётом самой сортировки итоговая сложность по времени должна составлять **O(nlogn)**.

3. **Оценка сложности по памяти:**

С применением вектора и мапов должна получаться **O(n)**.

Задача №8 «Магазин»

1. Пояснение к примененному алгоритму:

Опять же, используется собственный сравнительный метод, после которого используется дефолтная сортировка. Единственная особенность — здесь проверяется чётность товаров.

Исходный код:

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

bool comparator(int const a, int const b){
    return a > b;
}
int main(){
    int n, k, total = 0;
    std::cin >> n >> k;
    int products[n];
    for(int i = 0; i < n; i++) {
```

```
        std::cin >> products[i];
    }
    std::sort(products, products + sizeof products / sizeof products[0],
comparator);
    for(int i = 1; i <= n; i++) {
        if(i % k != 0) {
            total += products[i - 1];
        }
    }
    std::cout << total;
    return 0;
}
```

2. **Оценка сложности по времени:**

С учётом самой сортировки итоговая сложность по времени должна составлять **O(nlogn)**.

3. **Оценка сложности по памяти:**

С применением массива должна получаться **O(n)**.

Задача №1444 «Накормить элефпотама»

1. **Пояснение к примененному алгоритму:**

Здесь уже поинтереснее. Кроме выражения тыквы (или точки) здесь более интересный сравнительный метод: сравнивается конкретно тангенсы точек. Если точки равны, то сравнивается сумма квадратов координат. После сортировки находится начало элефпотама.

Исходный код:

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

struct pumpkin{
    int x;
    int y;
    int i;
};
bool comparator(const pumpkin a, const pumpkin b) {
    if(atan2(a.y, a.x) != atan2(b.y, b.x)) {
        return atan2(b.y, b.x) > atan2(a.y, a.x);
    }
    return pow(b.x, 2) + pow(b.y, 2) > pow(a.x, 2) + pow(a.y, 2);
}
int main() {
    int n, x_init, y_init, begin = 0;
    std::vector<pumpkin> lawn;
    std::cin >> n;
    std::cin >> x_init >> y_init;
    for(int i = 1; i < n; ++i) {
        int x, y;
        std::cin >> x >> y;
        lawn.push_back({x - x_init, y - y_init, i + 1});
    }
    sort(lawn.begin(), lawn.end(), comparator);
    for(int i = 0; i < n - 2; ++i) {
```

```
        if(lawn[i].x * lawn[i + 1].y <= lawn[i].y * lawn[i + 1].x
        && lawn[i].x * lawn[i + 1].x < -(lawn[i].y * lawn[i + 1].y)) {
            begin = i;
            break;
        }
    }
    std::cout << n << "\n" << 1 << "\n";
    for(int i = 0; i < n - 1; i++) {
        std::cout << (lawn[(begin + i) % (n - 1)].i) << std::endl;
    }
    return 0;
}
```

2. **Оценка сложности по времени:**

С учётом дефолтной сортировки (quicksort) итоговая сложность по времени должна составлять **O(nlogn)**.

3. **Оценка сложности по памяти:**

С применением вектора должна получаться **O(n)**.

Задача №1604 «В стране дураков»

1. **Пояснение к примененному алгоритму:**

Здесь пришлось использовать помощь priority_queue. Это такая структура данных, который работает почти как стек (здесь LIFO), но ещё и автосортируемый (но требуется здесь класс). После этого здесь происходит поочередное вытаскивание знаков из вершины очереди до его опустошения.

Исходный код:

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

class comparable {
public:
    bool operator()(const std::pair<int, int> a, const std::pair<int, int> b) {
        if(a.first != b.first) {
            return a.first < b.first;
        }
        return a.second > b.second;
    }
};
int main() {
    std::priority_queue<std::pair<int, int>, std::vector<std::pair<int, int>>,
comparable> p_queue;
    int k;
    std::cin >> k;
    for(int i = 1; i <= k; ++i) {
        int x;
        std::cin >> x;
        p_queue.emplace(x, i);
    }
    while(!p_queue.empty()) {
        std::pair<int, int> top = p_queue.top();
        p_queue.pop();
        std::cout << top.second << " ";
```

```
        if(!p_queue.empty()) {
            std::pair<int, int> second_top = p_queue.top();
            p_queue.pop();
            std::cout << second_top.second << " ";
            if(second_top.first > 1) {
                p_queue.emplace(second_top.first - 1, second_top.second);
            }
        }
        if(top.first > 1) {
            p_queue.emplace(top.first - 1, top.second);
        }
    }
    return 0;
}
```

2. **Оценка сложности по времени:**

С учётом дефолтной сортировки (quicksort), которая используется для сортировки очереди, итоговая сложность по времени должна составлять **O(nlogn)**.

3. **Оценка сложности по памяти:**

С применением очереди должна получаться **O(n)**.