Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №3

по «Алгоритмам и структурам данных»

Базовые задачи / Timus

Выполнил:

Студент группы P3219

Билобрам Д. А.

Преподаватели:

Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2023

Задача №I «Машинки»

Пояснение к примененному алгоритму:

Алгоритм на каждой итерации находит среди машинок на полу ту, что в следующий раз понадобится как можно позднее, именно эту машинку алгоритм заменяет на нужную сейчас машинку чтобы минимизировать количество операций.

Если удалять машинку, которая понадобится раньше других, то снова заменять машинку чтобы поставить эту придется раньше, что и увеличит количество операций. Алгоритм основана на жадном методе, мы пытаемся отсрочить возможную следующую замену как можно дальше в будущее. Если мы всегда убираем машинку, которая не будет нужна дольше других, мы максимизируем время, в течение которого оставшиеся машинки на полу будут удовлетворять потребности Пети без дополнительных операций замены.

Оценка сложности по времени: O(p * log k)

p итераций в каждой из который добавление/удаление элемента priority_queue за log k

Оценка сложности по памяти: O(p + k)

Код алгоритма:

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

using namespace std;

int main()
{
    int n, k, p;
    cin >> n >> k >> p;

    vector<queue<int>> next_use(n + 1);
    vector<int> requests(p);

    for (int i = 0; i < p; i++)
    {
        cin >> requests[i];
        next_use[requests[i]].push(i);
    }
```

```
    unordered_set<int> floor;
    priority_queue<pair<int, int>> farthest_next;

    int res = 0;

    for (int i = 0; i < p; i++)
    {
        int car = requests[i];

        next_use[car].pop();

        if (floor.find(car) != floor.end())
        {
            if (!next_use[car].empty())
            {
                int next_time = next_use[car].front();
                farthest_next.push({next_time, car});
            }
            else
            {
                farthest_next.push({p, car});
            }
            continue;
        }

        res++;
        if (floor.size() == k)
        {
            auto [next_time, to_remove] = farthest_next.top();
            farthest_next.pop();
            floor.erase(to_remove);
        }

        floor.insert(car);
        if (!next_use[car].empty())
        {
            int next_time = next_use[car].front();
            farthest_next.push({next_time, car});
        }
        else
        {
            farthest_next.push({p, car});
        }
    }

    cout << res;
}
```

Задача №J «Гоблины и очереди»

Пояснение к примененному алгоритму:

Алгоритм заключается в создании двух deque, одной для левой части очереди второй для правой части очереди, тогда приоритетных гоблинов можно удобно добавлять либо в конец первой очереди, либо в начало второй очереди в

зависимости от четности общего количества гоблинов в очереди. При каждом добавлении или удалении гоблинов происходит балансировка левой и правой части очереди.

Алгоритм будет работать верно так как он моделирует процесс добавления и удаления гоблинов по заданным правилам.

Оценка сложности по времени: O(n)

n итераций в каждой из которых добавление/удаление элементов deque за O(1)

Оценка сложности по памяти: O(n)

Код алгоритма:

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

using namespace std;

int main()
{
    string n;
    getline(cin, n);

    deque<int> first, second;
    int lim = stoi(n);
    for (int i = 0; i < lim; i++)
    {
        string command;
        getline(cin, command);

        if (command[0] == '+')
        {
            int num = stoi(command.substr(2));
            second.push_back(num);
        }
        else if (command[0] == '*')
        {
            int num = stoi(command.substr(2));
            int sz = first.size() + second.size();
            if (sz % 2 == 0)
            {
                first.push_back(num);
            }
            else
            {
                second.push_front(num);
            }
        }
```

```
        else if (command[0] == '-')
        {
            if (!first.empty())
            {
                cout << first.front() << "\n";
                first.pop_front();
            }
            else
            {
                cout << second.front() << "\n";
                second.pop_front();
            }
        }

        if (first.size() > second.size() + 1)
        {
            second.push_front(first.back());
            first.pop_back();
        }
        else if (second.size() > first.size())
        {
            first.push_back(second.front());
            second.pop_front();
        }
    }
}
```

Задача №K «Менеджер памяти»

Пояснение к примененному алгоритму:

Алгоритм заключается в манипулировании массивом free в котором содержатся блоки свободной памяти в виде {начало, конец}. При выделении памяти алгоритм находит первый подходящий блок и выделяет его путем обрезания его размера или удалением. При освобождении памяти создается новый блок в free а затем запускается рекурсивная функция по объединению соседних блоков в один большой.

Алгоритм работает корректно так как всегда поддерживает максимально возможную величину блоков свободной памяти, а следовательно, корректно выделит память если это возможно.

Оценка сложности по времени: O(m*f) при f < m

m итераций и объединение f блоков на некоторых из них

Оценка сложности по памяти: O(n + m)

n сохраненных запросов и m блоков памяти

Код алгоритма:

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

using namespace std;

void mergeBlocks(vector<pair<int, int>> &free, int index)
{

    if (index > 0 && free[index - 1].second + 1 == free[index].first)
    {
        free[index - 1].second = free[index].second;
        free.erase(free.begin() + index);
        index--;
        mergeBlocks(free, index);
    }

    if (index < free.size() - 1 && free[index].second + 1 == free[index +
1].first)
    {
        free[index].second = free[index + 1].second;
        free.erase(free.begin() + index + 1);
        mergeBlocks(free, index);
    }
}

int main()
{

    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);

    int n, m;
    cin >> n >> m;
    int rej = -1;

    map<int, pair<int, int>> requests;
    vector<pair<int, int>> free;
    free.push_back({1, n});

    int req;
    for (int i = 1; i < m + 1; i++)
    {
        cin >> req;
        if (req > 0)
        {
            bool alloc = false;
            for (int j = 0; j < free.size(); j++)
            {
                if (free[j].second - free[j].first + 1 >= req)
                {
                    int start = free[j].first;
                    int end = free[j].first + req - 1;
```

```
                requests[i] = {start, end};
                if (free[j].second != end)
                {
                    free[j].first = end + 1;
                }
                else
                {
                    free.erase(free.begin() + j);
                }
                alloc = true;
                cout << start << "\n";
                break;
            }
        }

        if (!alloc)
        {
            cout << rej << "\n";
        }
    }
    else
    {
        if (requests.find(abs(req)) == requests.end())
        {
            continue;
        }
        auto bounds = requests[abs(req)];
        int last_i = -1;
        for (int i = 0; i < free.size(); i++)
        {
            if (free[i].first > bounds.second)
            {
                last_i = i;
                break;
            }
        }
        if (last_i == -1)
        {
            if (!free.empty() && free.back().second == bounds.first -
1)
            {
                free.back().second = bounds.second;
                mergeBlocks(free, free.size() - 1);
            }
            else
            {
                free.push_back(bounds);
                last_i = free.size() - 1;
                mergeBlocks(free, last_i);
            }
        }
        else
        {
            free.insert(free.begin() + last_i, bounds);
            mergeBlocks(free, last_i);
        }
    }
```

```
    }
```

Задача №L «Минимум на отрезке»

Пояснение к примененному алгоритму:

Алгоритм заключается в поддержании монотонной очереди на отрезке окна, на каждой итерации удаляется элемент, который выходит за окно и удаляются элементы, которые больше нового элемента так как они точно не будут минимальными на этой итерации и всех последующих итерациях, на которых будет этот новый элемент.

Алгоритм удаляет все элементы окна, которые больше нового элемента тем самым минимальный элемент окна всегда будет самым первым в очереди.

Оценка сложности по времени: O(n*k)

n итераций и удаление от 1 до k элементов из deque

Оценка сложности по памяти: O(n)

Код алгоритма:

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

using namespace std;

int main()
{

    int n, k;
    cin >> n >> k;

    deque<int> deq;
    vector<int> nums(n);

    for (int i = 0; i < n; i++)
    {
        cin >> nums[i];
    }

    for (int i = 0; i < n; i++)
    {
        if (!deq.empty() && deq.front() == i - k)
        {
            deq.pop_front();
        }
```

```
        while (!deq.empty() && nums[deq.back()] >= nums[i])
        {
            deq.pop_back();
        }
        deq.push_back(i);
        if (i >= k - 1)
        {
            cout << nums[deq.front()] << " ";
        }
    }
}
```

Задача №1521 «Военные учения 2»

Пояснение к примененному алгоритму:

Алгоритм заключается в простой симуляции процесса исключения каждого k-ого солдата.

Оценка сложности по времени: O(n)

Оценка сложности по памяти: O(n)

Код алгоритма:

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

using namespace std;

int main()
{

    int n, k;
    cin >> n >> k;

    vector<int> soldiers(n);

    for (int i = 0; i < n; i++)
    {
        soldiers[i] = i + 1;
    }

    int to_toilet = k - 1;
    while (!soldiers.empty())
    {

        cout << soldiers[to_toilet] << " ";

        soldiers.erase(soldiers.begin() + to_toilet);
```

```
        if (!soldiers.empty())
        {
            to_toilet = (to_toilet - 1 + k) % soldiers.size();
        }
    }
}
```

Задача №1494 «Монобильярд»

Пояснение к примененному алгоритму:

Алгоритм заключается в попытке моделировать процесс закатывания и вытаскивания n шаров с проверкой возможности вытаскивания шаров в заданном порядке, при этом в качестве лунки выступает stack так как он идеально подходит для моделирования этого процесса. В цикле для каждого шара проверяется может ли он быть закатан согласно правилам при заданном порядке вытаскивания шаров.

Алгоритм строго следует порядку закатывания шаров от 1 до N. Любое нарушение порядка (когда верхний шар в стеке не совпадает с текущим извлекаемым шаром) немедленно приводит к выводу "Cheater", что гарантирует точность проверки. В начале стек пуст, и первый шар, который должен быть закатан, имеет номер 1. Для каждого номера шара b из списка, который ревизор извлекал: если шар с номером b ещё не закатан (т.е., next_ball ≤ b), алгоритм последовательно закатывает все шары от next_ball до b, помещая каждый из них в стек. Проверяется, что верхний шар в стеке (тот, который должен быть извлечен следующим) совпадает с b. Если это так, шар извлекается из стека. Если нет, то очевидно, что без жульничества такая ситуация невозможна.

Оценка сложности по времени: O(n)

Оценка сложности по памяти: O(n)

Код алгоритма:

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

using namespace std;

int main()
{
    int n;
    cin >> n;
```

```
    vector<int> balls(n);
    for (int i = 0; i < n; i++)
    {
        cin >> balls[i];
    }

    stack<int> lunka;
    int next_ball = 1;
    for (int i = 0; i < n; i++)
    {

        while (next_ball <= balls[i])
        {
            lunka.push(next_ball);
            next_ball++;
        }

        if (lunka.top() != balls[i])
        {
            cout << "Cheater";
            return 0;
        }
        else
        {
            lunka.pop();
        }
    }
    cout << "Not a proof"
}
```