Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №3

по «Алгоритмам и структурам данных»

Яндекс.Контест

Выполнил:

Студент группы P3211

Кривоносов Е.Д.

Преподаватели:

Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2021

Задача №I «Машинки»

Пояснение к примененному алгоритму:

Для решения данной задачи нам нужно знать когда последний раз Петя взял определенную машинку (на каком шаге), которая была на полу уже для того, чтобы расставить на позиции машинки, которые он захочет следующими. Таким образом мы получим массив с шагами, на которых Петя захочет определенную машинку, которую уже брал.
Дальше мы пробегаем данный массив и считаем, сколько раз нам придётся достать новую машинку, которой в данный момент нет на полу. Если шаг, на котором понадобится, машинка совпадает с шагами, которые находятся в данный момент на полу машинок, то мы удаляем его и добавляем следующий, не считая изменение. Если у нас уже на полу максимальное количество машинок (шагов), то мы удаляем наибольший и добавляем следующий учтя изменение машинок на полу.

Сложность алгоритма:
O(n)

Код:

```
#include <iostream>

#include <string.h>
#include <set>
#include <vector>

using namespace std;

void solve(){
    int n, k, p, car;
    int i = 0, count = 0;
    cin >> n >> k >> p;

    vector<int> cars;       // Содержит все машинки
    vector<int> last_car;   // Указатели, когда мы брали в последний раз машинку
    vector<int> next_car;   // На каком шаге машинка понадобится, которые уже мы брали

    while (cin >> car){
        cars.push_back(car);
        next_car.push_back(100000);
        i++;
    }

    for (int i = 0; i < n; i++){
        last_car.push_back(-100000);
    }

    // Запоминаем когда мы в последний раз брали машинку (шаг)
    // Заполняем массив, шагами, машинок, с которыми мы уже играли
    for (int i = 0; i < p; i++){
        if (last_car[cars[i] - 1] != -100000){
            next_car[last_car[cars[i] - 1]] = i;
        }
        last_car[cars[i] - 1] = i;
    }
```

```
    set<int> cur_cars;  // Храним шаги текущих машинок на полу, когда они
                        // понадобятся в следующий раз, учитывая вместимость

    for (int i = 0; i < p; i++){
        // Проверяем, если у нас машинка, которая у нас на полу
        // (шаг на котором она понадобится с текущим шагом)
        if (!cur_cars.count(i)){
            count++;
            if (cur_cars.size() == k){
                // Удаляем последний добавленный шаг машинки и добавляем
следующий
                auto id_car = cur_cars.end();
                cur_cars.erase(--id_car);
            }
        } else {
            cur_cars.erase(i);
        }
        cur_cars.insert(next_car[i]);
    }
    cout << count;
}

int main(){
    ios_base::sync_with_stdio(0);
    cin.tie(0); cout.tie(0);

    solve();
    return 0;
}
```

Задача №J «Гоблины»

Пояснение к примененному алгоритму:

Для решения этой задачи я использовал две двухсторонние очереди, хотя позже узнал, что list имеет вставку за O(1). Суть двух очередей заключалась в том, что ими я мог контролировать середину очереди гоблинов и делать туже самую вставку за O(1). И если количество в правой очереди было больше чем в левой, то мы просто 1 гоблина перемещали налево и снова возобновляли баланс середины.

Сложность алгоритма:
O(n)

Код:

```
#include <iostream>
#include <string.h>
#include <deque>

using namespace std;

void solve(){
    int n;
    string line;
```

```
    cin >> n;
    deque<string> q1, q2;
    while(cin >> line){
        // добавляет в конец очереди
        if (line == "+"){
            cin >> line;
            q2.push_back(line);
            // cout << "+ " << line << endl;
        // добавляет в середину очереди
        } else if (line == "*"){
            cin >> line;
            q2.push_front(line);
        } else if (line == "-"){
            cout << q1.front() << endl;
            q1.pop_front();
        }
        // отвечает за середину очереди (полной), часть которая перед VIP мы
храним
        // в q1, а всех остальных в q2 и как раз в начале очереди q2 у нас
находится середина
        if (q1.size() < q2.size()){
            q1.push_back(q2.front());
            q2.pop_front();
        }
    }
}

int main(){
    ios_base::sync_with_stdio(0);
    cin.tie(0); cout.tie(0);

    solve();
    return 0;
}
```

Задача №K «Менеджер памяти-1»

Пояснение к примененному алгоритму:

Одна из самых сложных задач на всем контесте. Для этой задачи я воспользовался двухсвязным списком, кучей и массивом запросов, где мы хранили под значением индекса номер запроса, а значение у нас отвечало за статус этого запроса.

Обрабатывали мы запроса таким образом: рассматривали самый длинный свободный отрезок (куча позволяла нам его найти за O(1)). Если нам нахватало длины для вмещения, то мы выдавали -1, иначе мы могли разместить данный блок, вставляла отрезок в список и укорачивали длину отрезка.
При обработке запросов на освобождение памяти мы должны были рассмотреть 4 случая:
1) Слева и справа от нашего длинного отрезка нет свободного отрезка. Тогда мы делаем наш отрезок свободным и добавляем его в кучу.

2) Слева от нашего длинного отрезка нет свободного отрезка, а справа есть. Тогда мы удаляем наш длинный отрезок и отрезок, который справа.

3) Аналогично пункту 2, только в этот раз справа нет, а слева есть.

4) Ну и, если у нас с двух сторон нашего длинного отрезка, есть два других отрезка. Тогда мы удаляем наш длинный отрезок и отрезок справа из списка, а отрезок слева удлиняем.

Сложность алгоритма:
O(n*log(n))

Код:

```
#include <iostream>

using namespace std;

struct Segment{
    Segment *prev, *next;
    bool free_m;
    int l, r, p;

    Segment(Segment *prev, Segment *next, bool free_m, int l, int r, int p){
        this->prev = prev;
        this->next = next;
        this->free_m = free_m;
        this->l = l;
        this->r = r;
        this->p = p;
        if (prev){
            prev->next = this;
        }
        if (next){
            next->prev = this;
        }
    };

    void remove(){
        if (prev){
            prev->next = next;
        }
        if (next){
            next->prev = prev;
        }
    };
};

// Куча из указателей на свободные отрезки и двухсвязный список свободных
отрезков
Segment *heap[100001], *r[100001];

bool better(int a, int b){
    return (heap[a]->r - heap[a]->l) > (heap[b]->r - heap[b]->l);
}

void solve(){
    int n, m, k, l;
    int rs[100001]; // номер запроса (статус)
    cin >> n >> m;
    l = 1;
    heap[0] = new Segment(0, 0, true, 0, n, 0);
```

```
    for (int i = 0; i < m; i++){
        int t;
        cin >> t;
        if (t > 0){
            Segment *prefix = heap[0];

            // Запрос отклоняется, если нет свободных ячеек
            if (prefix->r - prefix->l < t){
                rs[k++] = 0;
                cout << "-1" << '\n';
                continue;
            }

            rs[k++] = 1;
            r[k - 1] = new Segment(prefix->prev, prefix, false, prefix->l,
prefix->l+t, -1);
            cout << 1 + prefix->l << '\n';

            prefix->l += t;
            if (prefix->l < prefix->r){
                int a = prefix->p;
                while (true){
                    int q = a;
                    if ((a << 1) + 1 < l && better((a << 1) + 1, q)){
                        q = (a << 1) + 1;
                    }
                    if ((a << 1) + 2 < l && better((a << 1) + 2, q)){
                        q = (a << 1) + 2;
                    }
                    if (a == q){
                        break;
                    }
                    Segment *t = heap[a];
                    heap[a] = heap[q];
                    heap[q] = t;
                    heap[a]->p = a;
                    heap[q]->p = q;
                    a = q;
                }
            } else {
                prefix->remove();
                l--;
                if (l){
                    Segment *t = heap[0];
                    heap[0] = heap[l];
                    heap[l] = t;
                    heap[0]->p = 0;
                    heap[l]->p = l;

                    int a = 0;
                    while (true){
                        int q = a;
                        if ((a << 1) + 1 < l && better((a << 1) + 1, q)){
                            q = (a << 1) + 1;
                        }
                        if ((a << 1) + 2 < l && better((a << 1) + 2, q)){
                            q = (a << 1) + 2;
                        }
                        if (a == q){
                            break;
                        }
```

```
                    Segment *t = heap[a];
                    heap[a] = heap[q];
                    heap[q] = t;
                    heap[a]->p = a;
                    heap[q]->p = q;
                    a = q;
                }
            }
            delete(prefix);
        }
    } else {
        int t_n = -t;
        t_n--;

        rs[k++] = 2;
        if (!rs[t_n]){
            continue;
        }
        rs[t_n] = 2;

        Segment *prefix = r[t_n], *sp = prefix->prev, *sn = prefix->next;
        bool bp = sp && sp->free_m;
        bool bn = sn && sn->free_m;

        // Создаем новый отрезок в кучу, если справа и слева от нашего
отрезка
        // не располагается нового отрезка
        if (!bp && !bn){
            prefix->free_m = true;

            prefix->p = l;
            heap[l] = prefix;
            int d = l++;
            while (d && better(d, (d - 1) >> 1)){
                Segment *t = heap[d];
                heap[d] = heap[(d - 1) >> 1];
                heap[(d - 1) >> 1] = t;
                heap[d]->p = d;
                heap[(d - 1) >> 1]->p = (d - 1) >> 1;
                d = (d-1) >> 1;
            }
            continue;
        }

        // Увеличиваем следующий отрезок, если слева от нашего отрезка
        // не располагается нового отрезка, а справа располагается
        if (!bp){
            sn->l = prefix->l;
            int d = sn->p;
            while (d && better(d, (d - 1) >> 1)){
                Segment *t = heap[d];
                heap[d] = heap[(d - 1) >> 1];
                heap[(d - 1) >> 1] = t;
                heap[d]->p = d;
                heap[(d - 1) >> 1]->p = (d - 1) >> 1;
                d = (d-1) >> 1;
            }
            prefix->remove();
            delete(prefix);
            continue;
        }
```

```
            // Увеличиваем предыдущий отрезок, если справа от нашего отрезка
            // не располагается нового отрезка, а слева располагается
            if (!bn){
                sp->r = prefix->r;
                int d = sp->p;
                while (d && better(d, (d - 1) >> 1)){
                    Segment *t = heap[d];
                    heap[d] = heap[(d - 1) >> 1];
                    heap[(d - 1) >> 1] = t;
                    heap[d]->p = d;
                    heap[(d - 1) >> 1]->p = (d - 1) >> 1;
                    d = (d-1) >> 1;
                }
                prefix->remove();
                delete(prefix);
                continue;
            }

            // Если слева от нашего отрезка располагается свободный отрезок, а
            // справа другой свободный отрезок, то мы удаляем наш отрезок и
отрезок справа,
            // а отрезок слева увеличиваем
            sp->r = sn->r;
            int d = sp->p;
            while (d && better(d, (d - 1) >> 1)){
                Segment *t = heap[d];
                heap[d] = heap[(d - 1) >> 1];
                heap[(d - 1) >> 1] = t;
                heap[d]->p = d;
                heap[(d - 1) >> 1]->p = (d - 1) >> 1;
                d = (d-1) >> 1;
            }
            prefix->remove();
            delete(prefix);

            int a = sn->p;
            Segment *t = heap[a];
            heap[a] = heap[l-1];
            heap[l-1] = t;
            heap[a]->p = a;
            heap[l-1]->p = l-1;
            l--;
            if (!(a >= l)){
                int d = a;
                while (d && better(d, (d - 1) >> 1)){
                    Segment *t = heap[d];
                    heap[d] = heap[(d - 1) >> 1];
                    heap[(d - 1) >> 1] = t;
                    heap[d]->p = d;
                    heap[(d - 1) >> 1]->p = (d - 1) >> 1;
                    d = (d-1) >> 1;
                }
                while (true){
                    int q = a;
                    if ((a << 1) + 1 < l && better((a << 1) + 1, q)){
                        q = (a << 1) + 1;
                    }
                    if ((a << 1) + 2 < l && better((a << 1) + 2, q)){
                        q = (a << 1) + 2;
                    }
                    if (a == q){
                        break;
```

```
                    }
                    Segment *t = heap[a];
                    heap[a] = heap[q];
                    heap[q] = t;
                    heap[a]->p = a;
                    heap[q]->p = q;
                    a = q;
                }
            }
            sn->remove();
            delete(sn);
        }
    }
}

int main(){
    ios_base::sync_with_stdio(0);
    cin.tie(0); cout.tie(0);

    solve();
    return 0;
}
```

Задача №L «Минимум на отрезке»

Пояснение к примененному алгоритму:

Для решения этой задачи я использовал двухстороннюю очередь со своей реализацией add_back и del_front.

В чем логика функций add_back и del_front? Представим вот такую последовательность, которую занимает наше окно k = 5: {1, 5, 7, 9, 12}. Минимальный здесь будет 1. Следующее число, на которое мы сдвинемся, будет, например 3. С помощью функции del_front мы удаляем из очереди наше самое первое число т.к. мы уходим с него и получаем {5, 7, 9, 12}, если бы у нас минимум был 2 числом, то мы удалили бы его на 2 шаге т.к мы следим за этим. А с помощью функции add_back, мы сразу ищем минимум для его окна и добавляем в конец наше число. Т.к. мы сдвинулись на число 3, все остальные числа в очереди можно удалить. Потому что они больше, чем наше число, а по нашей логике, минимальное число у нас всегда будет находится первым, получая с помощью deque.front() за O(1); В итоге мы будем иметь очередь {3}. Нам уже не важно какие числа, стояли до этого так как мы нашли уже минимальное для новой позиции окна.

За O(k) - мы найдем в начальном положении.

В худшем случае Алгоритм будет работать за O(n) если нам придётся пробегать каждый раз все окна.

Сложность алгоритма:
O(n)

Код:

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

using namespace std;

void add_back(deque<int> &deq, int val){
    // Если в наше число, на которое сдвигаемся, будет меньше чем все остальные
в очереди
    // то мы должны их будем удалить, так как наше число будет считаться
минимальным,
    // в другом случае перед ним сохранится минимальное число для позции 'окна'.
    // Пример логики ниже кода.
    while (!deq.empty() && deq.back() > val){
        deq.pop_back();
    }
    // Добавляем в конец очереди наше число, на которое мы передвинулись.
    deq.push_back(val);
}

    // Данная функция нужна для того, чтобы удалять число, с которого мы
здвинулись.
void del_front(deque<int> &deq, int val){
    if (deq.front() == val){
        deq.pop_front();
    }
}

void solve(){
    int n, k;
    cin >> n >> k;
    vector<int> arr(n);
    for (int i = 0; i < n; i++){
        cin >> arr[i];
    }

    // Работа программы:
    deque<int> deq;

    // Ищем для 'окна' в начальном положении длины k, минимальное число в нем.
    int i = 0;
    for (; i < k; i++){
        add_back(deq, arr[i]);
    }
    cout << deq.front() << ' ';

    // Двигаясь вправо ищем для всех остальных расположений 'окна' длиной k
минимальное число в нем
    // Двигаться мы будем N - K раз.
    for (; i < n; i++){
        del_front(deq, arr[i - k]);
        add_back(deq, arr[i]);
        cout << deq.front() << ' ';
    }
}

int main(){
    ios_base::sync_with_stdio(0);
    cin.tie(0); cout.tie(0);

    solve();
    return 0;
```

```
}
```