ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«Национальный исследовательский университет ИТМО»

Факультет программной инженерии и компьютерной техники

«Алгоритмы и структуры данных»

отчет по блоку задач №3 (Яндекс.Контест)

Выполнил:

Студент группы P3215

Барсуков М.А.

Преподаватель:

Косяков М.С.

Санкт-Петербург, 2024

Задачи из 3-го блока Яндекс.Контеста

Задача №9 «I. Машинки»

Описание:

У Пети всего *N* машинок, но на полу может находиться только *K*. Перемещать машинки между полом и полкой может только Мама Пети, которой известно о том, в каком порядке Петя будет использовать машинки. Мама Пети выполняет любую операцию замещения одной машинки на другую (или просто снятия машинки) за одну операцию. Нужно узнать минимальное количество операций, которое надо совершить Петиной маме.

Доказательство:

Задача похожа на оптимальный алгоритм замещения страниц в виртуальной памяти OPT (оптимальный, потому что мы, в отличии от ОС, знаем, к каким машинкам/страницам будут следующие обращения, а также это глобальный алгоритм замещения, так как машинки использует только один Петя). Алгоритм прост: *замещай машину (страницу), которая не будет использоваться в течение самого длительного периода времени*.

Тогда для решения задачи нам нужно для каждого шага знать, когда в следующий раз Петя захочет использовать машинку, которую он использует на этом шаге – тривиальная задача. Для шагов, после которых используемая машинка не потребуется, используем как можно большее число, которое не может представлять номер шага (это потребуется в дальнейшем для определения порядка замещения).

Так, для использований машинок `[1, 2, 3, 1, 3, 1, 2]` из примера, этот массив следующих шагов использования будет:

- `[3, 6, 4, 5, M, M, M]`, где `M = INT_MAX` (последние три шага `M`, так как это последние шаги использований 3, 1 и 2 машинок соответственно)

Очевидно, что машинка, которая не будет использоваться в течение самого длительного периода времени – это машинка, шаг использования которой максимально далек от текущего шага, то есть та, следующий шаг использования которой максимален.

Теперь будем хранит текущие следующие шаги машинок в `set`, чтобы автоматически сортировать шаги и легко удалять последние элементы (то есть максимальные элементы, «следующие» шаги, которые не будут использоваться в течении самого длительного периода времени).

Итак, для каждого шагов (1… P):

- если набор **содержит текущий шаг**, это значит, что Петя снова использует ту же машинку, которую задействовал на шаге, для которого текущий шаг являлся следующим, использующим ту же машинку. Значит, мы удаляем текущий шаг из набора – замещение машинки не требуется, так как этот шаг был в наборе и не убирался из него.
- если набор **НЕ содержит текущий шаг**, это значит, что нужно заменить машинку по оптимальному критерию (т.е. **инкрементируем счетчик** операций Мамы Пети). Если в наборе меньше K шагов, значит его еще не наполнили, иначе мы удаляем максимальный следующий шаг – последний элемент.
- в конце добавляем в набор *следующий* шаг для *текущего* шага.

Таким образом, в счетчике мы получаем минимальное количество операций, которое надо совершить Петиной маме, то есть ответ.

Алгоритмическая сложность:

1. **По времени:** нам требуется по *P* выполнений цикла чтобы получить введенные машинки, установить следующие использования и узнать количество операций (там же мы вставляем и удаляем данные из набора максимальной длины *K*, `insert` и `erase` для `set` требуют O(log(K)) что в среднем и худшем случаях даёт временную сложность O(P*log(K)).
2. **По памяти:** Программа хранит только массив, набор и вектор из *P* шагов, вектор из *N* машинок, и некоторое постоянное число переменных (n, k, p, i, operations_count), поэтому сложность по памяти O(P).

Итого:

- По времени: O(P*log(K)) (линейная логарифмическая сложность).
- По памяти: O(P) (линейная сложность).

Код:

```cpp
#include <bits/stdc++.h>

using namespace std;

#define repeat(times, i) for (int (i) = 0; (i) < (times); (i)++)
#define NOT_SET_YET -1

int get_car_id(int* cars, int step) {
    return cars[step] - 1;
}

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

    int played_cars[p];
    repeat(p, i) cin >> played_cars[i];

    vector<int> last_step_for_car(n, NOT_SET_YET);

    // для каждого шага информация о следующем шаге, в котором будет использована машинка с этого шага
    vector<int> next_step_to_use_car(p, INT_MAX);

    repeat(p, i) {
        int car_id = get_car_id(played_cars, i);
        int current_last_step_for_i = last_step_for_car[car_id];

        if (current_last_step_for_i != NOT_SET_YET) {
            next_step_to_use_car[current_last_step_for_i] = i;
        }

        last_step_for_car[car_id] = i;
    }
```

```
34.
35.     int operations_count = 0;
36.     set<int> current_steps_for_cars_on_floor;
37.
38.     repeat(p, i) {
39.         if (current_steps_for_cars_on_floor.count(i)) {
40.             current_steps_for_cars_on_floor.erase(i);
41.         } else {
42.             operations_count++;
43.
44.             if (current_steps_for_cars_on_floor.size() == k) {
45.                 current_steps_for_cars_on_floor.erase(--
   current_steps_for_cars_on_floor.end());
46.             }
47.         }
48.
49.         current_steps_for_cars_on_floor.insert(next_step_to_use_car[i]);
50.     }
51.
52.     cout << operations_count << endl;
53.     return 0;
54. }
```

Задача №10 «J. Гоблины и очереди»

Описание:

Есть очередь, в которую можно вставать в середину, в конец, и из которой можно выходить из начала. Для каждого запроса выхода нужно найти элемент, который стоит в начале.

Доказательство:

Существует очевидная наивная реализация, где мы храним очередь просто как `deque` (так как мы очень часто изменяем нашу очередь и редко читаем) и спокойно изменяем очередь по условиям задачи при каждом запросе. Однако такая реализация не проходит по времени, поэтому нам нужно немного её модифицировать.

Для `deque` проанализируем используемые в наивной реализации методы:

`push_back` (для «+»)	amortized O(1)
`pop_front` (для «-»)	amortized O(1)
`insert` (для «*»)	O(N)

Амортизация константы в `push/pop_back/front` малосущественна, так как длина очереди изменяется не слишком значительно. Однако вставка привилегированного гоблина в середину это O(N). Значит, вставку в середину и нужно исправить.

Для того, чтобы вставка в середину также была ~ O(1), напишем структуру данных, состоящую из двух `deque` – первой и второй половин очереди. Реализуем нужные для задачи методы:

- **Добавление обычного гоблина «+ i»**: добавляем гоблина в конец последней половины и *уравниваем половины (см. далее)
- **Добавление привилегированного гоблина «* i»**: если длина очереди четная, добавляем гоблина в конец первой половины, иначе в начало второй.
- **Выход из очереди «–»**: Убираем гоблина из начала первой очереди и *уравниваем половины.

* Чтобы поддерживать очередь в удобном для нас состоянии (то есть очереди равны друг другу и разделены серединой), нам после выхода из очереди или добавления обычного гоблина нужно **уравнивать половины** (то есть сдвигать по необходимости): если длина очереди четная, перемещать гоблинов из начала второй очереди в конец первой. Сложность уравнивания – amortized O(1).

Таким образом, вне зависимости от метода, очередь поддерживается в состоянии, *при котором длина первой половины либо равна длине второй половины* (когда длина всей очереди чётна), *либо превосходит длину второй на 1* (когда длина всей очереди нечётна). Это помогает нам легко добавлять привилегированного гоблина со сложностью amortized O(1).

Благодаря этой модификации правильность алгоритма сохраняется, а время вставки привилегированного гоблина значительно уменьшается.

Алгоритмическая сложность:

1. **По времени:** Сложность каждого отдельного запроса: **«+ i»**, **«* i»** и **«–»** – amortized O(1). Значит, для N запросов в лучшем и среднем (так как в худшем на каждое изменение `deque` выделяется память) случаях будет временная сложность – O(N).

2. **По памяти:** Программа в общем случае хранит в очереди до *N* гоблинов (каждый гоблин занимает постоянное место) и некоторое постоянное число переменных (size, n, new_goblin, i, command, value), что в худшем случае требует O(N) памяти.

Итого:

- По времени: O(N) (линейная сложность).
- По памяти: O(N) (линейная сложность).

Код:

```
#include <bits/stdc++.h>

using namespace std;

#define repeat(times, i) for (int (i) = 0; (i) < (times); (i)++)

struct halved_list {
    size_t size = 0;
    deque<int> first_half;
    deque<int> last_half;
};

struct goblin {
    int id;
    bool is_privileged;
};


void move_between_halves(struct halved_list* goblins) {
    if (!goblins->last_half.empty()) {
        goblins->first_half.push_back(goblins->last_half.front());
        goblins->last_half.pop_front();
    }
}

void equalize_halves(struct halved_list* goblins) {
    if (goblins->size % 2 == 0) move_between_halves(goblins);
}

void push_goblin(struct halved_list* goblins, int new_goblin) {
    goblins->last_half.push_back(new_goblin);

    equalize_halves(goblins);
}

void push_privileged_goblin(struct halved_list* goblins, int new_goblin) {
    if (goblins->size % 2 == 0) {
        goblins->first_half.push_back(new_goblin);
    } else {
        goblins->last_half.push_front(new_goblin);
    }
```

```
}


int pop(struct halved_list* goblins) {
    int value = goblins->first_half.front();
    goblins->first_half.pop_front();

    equalize_halves(goblins);

    goblins->size--;
    return value;
}

void push(struct halved_list* goblins, goblin new_goblin) {
    if (new_goblin.is_privileged) {
        push_privileged_goblin(goblins, new_goblin.id);
    } else {
        push_goblin(goblins, new_goblin.id);
    }
    goblins->size++;
}


int main() {
    int n, new_goblin;
    cin >> n;

    halved_list goblins;

    char command;
    repeat(n, i) {
        cin >> command;

        if (command == '-') {
            cout << pop(&goblins) << endl;
        } else {
            cin >> new_goblin;
            push(&goblins, { .id = new_goblin, .is_privileged = (command == '*')
    });
        }
    }

    return 0;
}
```

Задача №11 «K. Менеджер памяти-1»

Описание:

В распоряжении у менеджера памяти находится массив из *N* последовательных ячеек памяти, пронумерованных от *1* до *N*. Менеджер памяти должен уметь выполнять команды аллокации памяти какого-то размера и освобождения памяти для предыдущих запросов аллокации. Если места для аллокации нет, запрос отклоняется. Требуется написать менеджер памяти, удовлетворяющий приведенным критериям.

Доказательство:

Разобьем всю память на блоки, каждый из которых свободен или занят (то есть эта часть памяти выделена в ответ на запрос). Свободные блоки не могут следовать друг за другом, так как аллоцированный блок памяти может находиться только либо в начале, либо после занятого блока. Для реализации менеджера памяти воспользуемся **двоичной кучей max-heap**, содержащей блоки и строящейся по их длине, а также двусвязным списком для хранения свободных блоков. Для введенных запросов будем хранить выделенный для запроса блок по индексу запроса в векторе.

Реализуем требуемые методы следующий образом:

- **Выделение памяти**: Возьмём корень кучи (то есть самый длинный свободный участок). Если его длина меньше требуемой памяти, то, очевидно, отклоняем запрос, а иначе укорачиваем корень на требуемое количество памяти и добавляем занятый блок.
- **Освобождение памяти**: Из вектора выделенных по запросу блоков берем нужный блок. Если запрос был отклонен, ничего не делаем, иначе помечаем блок удаленным и смотрим на соседние блоки. Если по обе стороны находятся занятые блоки, то найденный блок становится свободным и возвращается в кучу. Если же по обоим сторонам от занятого найденного блока находятся свободные, то мы удаляем текущий и правый блок и удлиняем левый блок на размеры текущего и соседнего. Если один из соседних блоков занят, а другой свободен, то мы удаляем текущий блок и удлиняем соседний свободный блок на его длину. Готово, память освобождена.

Алгоритмическая сложность:

- **По времени:** Сложность методов `heapify` (упорядочение двоичной кучи) и `lift` (а значит и `pop` и `remove`) O(log(N)), а нахождение наиболее длинного свободного блока по свойству кучи O(1). Значит, аллокация и освобождение памяти так же будут иметь сложность O(log(N)). Таким образом, для *M* запросов в среднем и худшем случаях временная сложность – O(M*log(N)).

- **По памяти:** Каждый блок занимает постоянное место, а в куче и в векторе выделений на запросы находится не более *M* блоков, а также есть некоторое постоянное число переменных (n, m, request, heap_size, current_request, i и так далее). В таком случае сложность по памяти будет O(M).

Итого:

- По времени: O(M*log(N)) (линейная логарифмическая сложность).
- По памяти: O(M) (линейная сложность).

Код:

```
#include <bits/stdc++.h>

using namespace std;

#define repeat(times, i) for (int(i) = 0; (i) < (times); (i)++)
#define loop for (;;)
#define unless(cond) if (!(cond))

typedef int block_index;

enum class Status {
    Declined = 0,
    Allocated = 1,
    Removed = 2
};


class Block {
public:
    bool is_free;
    int start, end;

    block_index index;

    Block *prev, *next;

    Block(Block *prev, Block *next, bool free, int start, int end, block_index index) {
        this->is_free = free;
        this->start = start;
        this->end = end;

        this->index = index;
        this->prev = prev;
        this->next = next;

        if (prev) prev->next = this;
        if (next) next->prev = this;
    }

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

    int size() {
        return this->end - this->start;
    }
};


```

```
class Heap {
public:
    int current_request;
    int heap_size;

    vector<Status> request_status;
    vector<Block*> heap;
    vector<Block*> blocks_for_requests;

    Heap(int n, int m) {
        current_request = 0;

        request_status.resize(m);
        blocks_for_requests.resize(m);

        heap.resize(m);
        heap_size = 1;
        heap[0] = new Block(nullptr, nullptr, true, 0, n, 0);
    }

    void allocate(int request_size) {
        Block *root = heap[0];

        if (heap_size == 0 || (root->size() < request_size)) {
            request_status[current_request++] = Status::Declined;
            cout << "-1" << endl;
            return;
        }

        request_status[current_request++] = Status::Allocated;
        blocks_for_requests[current_request - 1] = new Block(root-
>prev, root, false, root->start, root->start + request_size, -1);

        cout << root->start + 1 << endl;

        root->start += request_size;
        if (root->start < root->end) {
            heapify(root->index);
        } else {
            root->remove();
            pop();
            delete(root);
        }
    }

    void free(int request_index) {
        request_index--;

        request_status[current_request++] = Status::Removed;

            if (request_status[request_index] == Status::Declined) return;

            request_status[request_index] = Status::Removed;
```

```
103.
104.             Block *block = blocks_for_requests[request_index];
105.             Block *prev_block = block->prev;
106.             Block *next_block = block->next;
107.
108.             unless ((prev_block && prev_block-
    >is_free) || (next_block && next_block->is_free)) {
109.                 block->is_free = true;
110.                 block->index = heap_size;
111.                 heap[heap_size] = block;
112.                 lift(heap_size++);
113.                 return;
114.             }
115.             unless (prev_block && prev_block->is_free) {
116.                 next_block->start = block->start;
117.                 lift(next_block->index);
118.                 block->remove();
119.                 delete(block);
120.                 return;
121.             }
122.             unless (next_block && next_block->is_free) {
123.                 prev_block->end = block->end;
124.                 lift(prev_block->index);
125.                 block->remove();
126.                 delete(block);
127.                 return;
128.             }
129.
130.             prev_block->end = next_block->end;
131.             lift(prev_block->index);
132.
133.             block->remove();
134.             delete(block);
135.
136.             remove(next_block->index);
137.             next_block->remove();
138.             delete(next_block);
139.         }
140.
141.         void dispatch(int request) {
142.             if (request > 0) {
143.                 this->allocate(abs(request));
144.             } else {
145.                 this->free(abs(request));
146.             }
147.         }
148.
149.     private:
150.         block_index get_parent_index(block_index index) {
151.             return (index - 1) / 2;
152.         }
153.
154.         block_index get_left_child_index(block_index index) {
```

```
155.             return 2 * index + 1;
156.         }
157.
158.         block_index get_right_child_index(block_index index) {
159.             return 2 * index + 2;
160.         }
161.
162.         void swap(block_index index1, block_index index2) {
163.             std::swap(heap[index1], heap[index2]);
164.             heap[index1]->index = index1;
165.             heap[index2]->index = index2;
166.         }
167.
168.         bool better(block_index index1, block_index index2) {
169.             return heap[index1]->size() > heap[index2]->size();
170.         }
171.
172.         void heapify(block_index index) {
173.             loop {
174.                 block_index largest = index;
175.                 block_index left_child = get_left_child_index(index);
176.                 block_index right_child = get_right_child_index(index);
177.
178.                 if ((left_child < heap_size) && better(left_child, largest))
179.                     largest = left_child;
180.                 if ((right_child < heap_size) && better(right_child, largest))
181.                     largest = right_child;
182.                 if (index == largest) return;
183.
184.                 swap(index, largest);
185.                 heapify(largest);
186.             }
187.         }
188.
189.         void pop() {
190.             heap_size--;
191.             unless (heap_size == 0) {
192.                 swap(0, heap_size);
193.                 heapify(0);
194.             }
195.         }
196.
197.         void lift(block_index index) {
198.             while (index && better(index, get_parent_index(index))) {
199.                 swap(index, get_parent_index(index));
200.                 index = get_parent_index(index);
201.             }
202.         }
203.
204.         void remove(block_index index) {
205.             swap(index, heap_size - 1);
206.             heap_size--;
207.             if (index < heap_size) {
```

```
                lift(index);
                heapify(index);
            }
        }
    };


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

        int n, m, request;
        cin >> n >> m;

        Heap* heap = new Heap(n, m);
        repeat(m, i) {
            cin >> request;
            heap->dispatch(request);
        }

        delete(heap);
        return 0;
    }
```

Задача №12 «L. Минимум на отрезке»

Описание:

По массиву из *N* чисел слева направо с шагом 1 движется окно размером *K*. Для каждого положения окна найти минимум.

Доказательство:

Сначала считаем все введённые числа. Наивная реализация (когда мы ищем минимум среди *K* чисел *N – K + 1* раз) имеет сложность O(N*K) и не подходит по времени.

Поэтому для нахождения минимума в окне будем использовать двустороннюю очередь deque, в которой будем хранить индексы чисел в окне (мы могли бы хранить и сами числа, но с индексами удобнее проверять, что мы прошли первые K элементов и окно достигло полной длины).

В цикле по всем введённым числам, после того как прошли первые *K* элементов (размер текущего окна), каждый новый элемент добавляется в дек, а старые элементы удаляются, чтобы поддерживать размер окна *K*.

Затем мы удаляем слева от текущего элемента очереди все элементы, числа (не элементы-индексы!) которых ≥ текущего (по текущему индексу цикла) числа (т.е. до тех пор, пока не встретится число меньше текущего). То есть мы убеждаемся, что все элементы, которые меньше нового элемента и находятся левее его, удаляются из дека. Таким образом мы удаляем элементы, у которых уже нет шансов стать минимумом в окне.

Это гарантирует, что в деке всегда будут храниться только индексы элементов, которые могут быть потенциальным минимумом в текущем и следующих окне. Значит, **минимальным значением** в текущем окне в таком случае будет **первый элемент в деке** (так как слева от него точно нет чисел меньше, потому что мы бы их удалили, а справа, если есть число меньше, то для него мы бы должны были удалить наш первый элемент в цикле с удалением (так как он меньше) но он есть в деке, поэтому справа чисел меньше нет).

Алгоритмическая сложность:

1. **По времени:** Мы считываем *N* чисел. В основном цикле работы с окном происходит *N* итераций. Удаление старых элементом, добавление нового и вывод минимума происходят за константное время, а цикл удаления меньших элементов за все разы суммарно выполнится не больше *N* раз (больше чем есть он удалить, очевидно, не сможет), а значит получаем всего не более *N* + *N* операций. Таким образом, в среднем и худшем случаях временная сложность – O(N).
2. **По памяти:** Программа использует фиксированное количество переменных (n, k, i), а также массив из *N* введенных чисел и deque из не более чем *K* < *N* элементов, поэтому сложность по памяти составляет O(N).

Итого:

- По времени: O(N) (линейная сложность).
- По памяти: O(N) (линейная сложность).

Код:

```cpp
#include <bits/stdc++.h>

using namespace std;

#define repeat(times, i) for (int (i) = 0; (i) < (times); (i)++)

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

    int numbers[n];
    repeat(n, i) cin >> numbers[i];

    deque<int> chunk;
    repeat(n, i) {
        if (!chunk.empty() && (i - chunk.front() >= k)) {
            chunk.pop_front();
        }
        while (!chunk.empty() && numbers[chunk.back()] >= numbers[i]) {
            chunk.pop_back();
        }

        chunk.push_back(i);

        if (i + 1 >= k) cout << numbers[chunk.front()] << " ";
    }

    return 0;
}
```