Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное
учреждение высшего образования
«Национальный исследовательский университет ИТМО»

Факультет Программной инженерии и компьютерной техники

Решения задач блока 3

Алгоритмам и структурам данных

Работу выполнил:

Гаврилин О.С.

Группа:

P3230

Санкт-Петербург,

2025

1. I

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

using namespace std;
void solve() {
  int totalCars, maxOnFloor, totalRequests;
  cin >> totalCars >> maxOnFloor >> totalRequests;

  vector<int> requests(totalRequests);
  for (int i = 0; i < totalRequests; ++i) {
    cin >> requests[i];
  }

  const int notUsed = totalRequests + 1;
  vector<int> nextUsage(totalRequests);
  vector<int> futurePosition(totalCars + 1, notUsed);

  for (int i = totalRequests - 1; i >= 0; --i) {
    nextUsage[i] = futurePosition[requests[i]];
    futurePosition[requests[i]] = i;
  }

  unordered_set<int> floorSet;
  floorSet.reserve(maxOnFloor * 2);
  unordered_map<int, int> nextIndex;
  nextIndex.reserve(maxOnFloor * 2);
  set<pair<int, int>> usageOrder;

  int actionCount = 0;

  for (int i = 0; i < totalRequests; ++i) {
    int carId = requests[i];
    int upcoming = nextUsage[i];

    if (floorSet.count(carId)) {
      usageOrder.erase({nextIndex[carId], carId});
      nextIndex[carId] = upcoming;
      usageOrder.insert({upcoming, carId});
    } else {
      ++actionCount;
```

```
      if (floorSet.size() ==
static_cast<size_t>(maxOnFloor)) {
        auto it = prev(usageOrder.end());
        int toRemove = it->second;
        usageOrder.erase(it);
        floorSet.erase(toRemove);
        nextIndex.erase(toRemove);
      }
      floorSet.insert(carId);
      nextIndex[carId] = upcoming;
      usageOrder.insert({upcoming, carId});
    }
  }

  cout << actionCount;
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);
  solve();
  return 0;
}
```

Описание: Использую set для быстрого поиска машинки, которую можно заменить, текущий набор машинок на полу храню через unordered_set. В цикле обновляем информацию о следующем использовании текущей машинки в очереди, обновляем данные в зависимости от того на полу она или нет.

Сложность по времени: O(P log K)

Сложность по памяти: O (N + P + K)

2. **J**

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

using namespace std;

void solve() {
    int q;
    cin >> q;
    deque<int> front_half, back_half;
```

```
    while (q--) {
        string action;
        cin >> action;

        if (action[0] == '+') {
            int val;
            cin >> val;
            back_half.emplace_back(val);
        } else if (action[0] == '*') {
            int val;
            cin >> val;
            front_half.emplace_back(val);
        } else {
            if (!front_half.empty()) {
                cout << front_half.front() << '\n';
                front_half.pop_front();
            } else {
                cout << back_half.front() << '\n';
                back_half.pop_front();
            }
        }

        if (front_half.size() < back_half.size()) {
            front_half.push_back(back_half.front());
            back_half.pop_front();
        } else if (front_half.size() > back_half.size() + 1)
{
            back_half.push_front(front_half.back());
            front_half.pop_back();
        }
    }
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    solve();
}
```

Описание: Реализуем решение с помощью двух деков. Обычные гоблины добавляются в конец очереди, привелигерованный в середину (в данном случае конец первой половины). При вызове

“-” удаляется первый гоблин из начала очереди и проверяется front_half, затем back_half если первый пустой

Сложность по времени: O(q)

Сложность по памяти: O(q)

3. **K**

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

using namespace std;

void solve() {
  multimap<int, int> blocks_by_size;
  map<int, int> blocks;

  auto remove = [&](const multimap<int, int>::iterator& it)
{
    blocks.erase(it->second);
    blocks_by_size.erase(it);
  };

  auto remove_by_size = [&](const map<int, int>::iterator&
it) {
    auto it_d = blocks_by_size.find(it->second);
    while (it_d->second != it->first)
      ++it_d;
    blocks_by_size.erase(it_d);
    blocks.erase(it);
  };

  auto insert = [&](const pair<int, int>& pair) {
    blocks.insert(pair);
    blocks_by_size.insert({pair.second, pair.first});
  };

  int n, m;
  cin >> n >> m;
  vector<pair<int, int>> history(m);

  insert({1, n});

  for (int i = 0; i < m; ++i) {
```

```
    int k, index = 0, size = 0;
    cin >> k;

    if (k > 0) {
      auto it = blocks_by_size.lower_bound(k);
      if (it == blocks_by_size.end()) {
        index = -1;
      } else {
        index = it->second;
        size = it->first - k;
        remove(it);
        if (size > 0)
          insert({index + k, size});
      }
      cout << index << '\n';
      history[i] = {k, index};
    } else {
      history[i] = {k, 0};
      int idx = abs(k) - 1;
      int index_x = history[idx].second;
      int size_x = history[idx].first;

      if (index_x == -1)
        continue;

      auto it_r = blocks.lower_bound(index_x);
      auto it_l = (it_r != blocks.begin()) ? prev(it_r) :
blocks.end();

      if (it_r != blocks.end() && it_r->first == index_x +
size_x) {
        if (it_l != blocks.end() && it_l->first + it_l-
>second == index_x) {
          index = it_l->first;
          size = it_l->second + it_r->second;
          remove_by_size(it_l);
          remove_by_size(it_r);
          insert({index, size + size_x});
        } else {
          size = it_r->second;
          remove_by_size(it_r);
          insert({index_x, size + size_x});
        }
      } else {
```

```
        if (it_l != blocks.end() && it_l->first + it_l-
>second == index_x) {
          index = it_l->first;
          size = it_l->second;
          remove_by_size(it_l);
          insert({index, size + size_x});
        } else {
          insert({index_x, size_x});
        }
      }
    }
  }
}

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(nullptr);
  solve();
}
```

Описание: Использую map для отображения начала свободного блока и его размера, что позволяет быстро искать соседей, multimap хранит те же блоки, но позволяет быстро находить минимально подходящий по размеру блок. Запрос на выделение памяти – ищем самый левый свободный блок с размером >= k, если находим такой то удаляем блок из обоих контейнеров и возвращаем начало выделенного блока. Запрос на освобождение – ищем history[|t| - 1] пару выделенного блока и его индекса, если находим то пытаемся объединить с соседними свободными блоками слева и справа, обновляем структуры.

Сложность по времени: O (log n)

Сложность по памяти: O (m + n)

4. **L**

```
#include <deque>
#include <iostream>
#include <vector>
using namespace std;

void solve() {
    int n, k;
    cin >> n >> k;
    vector<int> a(n);
```

```
    for (int& x : a)
        cin >> x;

    deque<int> dq;

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

        while (!dq.empty() && a[dq.back()] >= a[i]) {
            dq.pop_back();
        }

        dq.push_back(i);

        if (i >= k - 1) {
            cout << a[dq.front()] << ' ';
        }
    }
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    solve();
    return 0;
}
```

Описание: Задача сводится к тому, чтобы эффективно анализировать новое окно и заранее знать минимальные элементы в нем. При нахождении нового окна и обладании данными о старом, нам необходимо проанализировать ушедший и добавленный элементы. Дек в данном случае позволяет хранить элементы в текущем окне в порядке возрастания, что позволяет за O (n) находить следующий минимум в окне, если предыдущей был убран.

Сложность по времени: O (n)

Сложность по памяти: O (n)