Федеральное государственное автономное образовательное учреждение высшего образования Национальный исследовательский университет ИТМО

Факультет программной инженерии и компьютерной техники

Алгоритмы и структуры данных

Задачи I, J, K, L (Яндекс.Контест)

Выполнил: студент группы P3208,
Васильев Н. А.

Преподаватель: Косяков М. С.

Санкт-Петербург 2025

Задача I. Машинки

Для каждой машины я храню все индексы её появлений в `carsOrder`. Это позволяет быстро получать информацию о следующем появлении конкретной машины.

В `active` хранится множество машин, которые в данный момент находятся у ребенка. Проверка и удаление работают за O(1), поэтому `unordered_set` подходит идеально.

`pq` это очередь с приоритетом, где я храню пары: <время следующего использования, номер машины>. Таким образом, я всегда могу быстро найти ту машину, которая будет использована позже всего и удалить её из активной зоны.

Алгоритм работает за **O(p log k)**, так как основная нагрузка идёт на `pq`, которая содержит максимум `k` элементов.

Код:

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

using namespace std;

int main() {
  int n, k, p = 0;
  cin >> n >> k >> p;
  vector<int> carsOrder(p);
  unordered_map<int, deque<int>> cars(p);
  for (int i = 0; i < p; i++) {
    int value;
    cin >> value;
    carsOrder[i] = value;
    cars[value].push_back(i);
  }

  int operations = 0;
  unordered_set<int> active;
  priority_queue<pair<int, int>> pq;

  for (int i = 0; i < p; i++) {
    int currentCar = carsOrder[i];
    cars[currentCar].pop_front();
    if (active.count(currentCar)) {
      int nextUsage = cars[currentCar].empty() ? p + 1 :
cars[currentCar].front();
      pq.emplace(nextUsage, currentCar);
      continue;
    }
    operations++;
    if ((int)active.size() >= k) {
```

```
      while (!pq.empty()) {
        auto [nextUsage, car] = pq.top();
        pq.pop();
        if (active.find(car) != active.end()) {
          active.erase(car);
          break;
        }
      }
    }
    active.insert(currentCar);
    int nextUsage = cars[currentCar].empty() ? p + 1 : cars[currentCar].front();
    pq.emplace(nextUsage, currentCar);
  }
  cout << operations << endl;
  return 0;
}
```

Задача J. Гоблины и очереди

Нашу память удобно разделить пополам для выполнения операции `*`. Двунаправленные очереди подходят для этого лучше всего, потому что позволяют быстро вставлять в начало и конец за O(1) и быстро удалять из начала и с конца за O(1). В результате получаем, что каждая операция (`+ i`, `* i`, `-`) выполняется за **O(1)** благодаря `deque`. В каждой итерации максимум 1 перенос элемента из одной очереди в другую, соответственно, O(1). Так как m — средняя длина строки команды, s — средняя длина значений, которые добавляются, то `v` занимает O(n * m) памяти, `result` O(n * s). В итоге получаем **O(n * max(m, s))**.

Код:

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

using namespace std;

int main() {
  int n;
  cin >> n;
  cin.ignore();
  deque<string> left, right;
  vector<string> v(n);
  vector<string> result;

  for (int i = 0; i < n; i++) {
    getline(cin, v[i]);
    v[i].erase(remove(v[i].begin(), v[i].end(), ' '), v[i].end());
  }

  for (const string& s : v) {
    switch (s[0]) {
```

```
      case '+': {
        string x = s.substr(1, s.length());
        right.push_back(x);
        break;
      }
      case '*': {
        string x = s.substr(1, s.length());
        right.push_front(x);
        break;
      }
      case '-': {
        result.push_back(left.front());
        left.pop_front();
        break;
      }
      default:
        break;
    }
    while (left.size() < right.size()) {
      left.push_back(right.front());
      right.pop_front();
    }
    while (left.size() > right.size() + 1) {
      right.push_front(left.back());
      left.pop_back();
    }
  }
  for (size_t i = 0; i < result.size(); i++) {
    cout << result[i] << endl;
  }
  return 0;
}
```

Задача K. Менеджер памяти-1

Для удобства используем структуру `Block`, где будем хранить начало и длину блока. Структуры будем хранить в векторе, чтобы быстро находить место для вставки и проверять соседние блоки. Также создадим вектор историй, который хранит историю выделений, то есть размер и адрес начала блока для каждого запроса, чтобы точно знать, какой блок нужно освободить по индексу. Вставка и изменение блока при таком подходе занимает O(1), поиск позиции вставки – O(log f), где f – количество свободных блоков. Итого получаем **O(m log f)**. Вектор `free_blocks` хранит максимум `n` блоков, `history` и `results` содержат максимум `m` значений, в итоге получаем **O(n + m)** памяти.

Код:

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

using namespace std;
```

```cpp
struct Block {
  int start, size;
  bool operator<(const Block& other) const {
    return start < other.start;
  }
};

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

  vector<Block> free_blocks = {
      {1, n}
  };
  vector<pair<int, int>> history(m);
  vector<int> result;

  for (int i = 0; i < m; ++i) {
    int req;
    cin >> req;

    if (req > 0) {
      int idx = -1;
      for (size_t j = 0; j < free_blocks.size(); ++j) {
        if (free_blocks[j].size >= req) {
          idx = free_blocks[j].start;
          history[i] = {req, idx};

          if (free_blocks[j].size == req) {
            free_blocks.erase(free_blocks.begin() + j);
          } else {
            free_blocks[j].start += req;
            free_blocks[j].size -= req;
          }
          break;
        }
      }
      result.push_back(idx);
    } else {
      int prev_idx = -req - 1;
      int size = history[prev_idx].first;
      int start = history[prev_idx].second;

      if (start == -1)
        continue;

      Block new_block = {start, size};

      auto it = lower_bound(free_blocks.begin(), free_blocks.end(), new_block);
```

```
      if (it != free_blocks.end() && start + size == it->start) {
        new_block.size += it->size;
        free_blocks.erase(it);
      }

      if (it != free_blocks.begin()) {
        auto prev = std::prev(it);
        if (prev->start + prev->size == new_block.start) {
          new_block.start = prev->start;
          new_block.size += prev->size;
          free_blocks.erase(prev);
        }
      }
      free_blocks.insert(lower_bound(free_blocks.begin(), free_blocks.end(),
new_block), new_block);
    }
  }

  for (int x : result) {
    cout << x << endl;
  }

  return 0;
}
```

Задача L. Минимум на отрезке

В двунаправленной очереди `q` храним индексы элементов массива `consistency`, которые могут быть минимумом в текущем и следующих окнах, при этом минимум всегда находится в начале. Каждый элемент массива один раз вставляется в очередь и один раз удаляется из неё. Получаем сложность **O(n)**. Вектор `consistency` хранит входной массив, следовательно O(n), а очередь максимум `k` индексов одновременно, то есть O(k). Общая память: **O(n)**.

Код:

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

using namespace std;

int main() {
  int n, k;
  cin >> n >> k;
  vector<int> consistency(n);
  for (int i = 0; i < n; i++) {
    cin >> consistency[i];
  }
  deque<int> q;
  vector<int> result;
  for (int i = 0; i < n; i++) {
```

```
    while (!q.empty() && q.front() <= i - k) {
      q.pop_front();
    }
    while (!q.empty() && consistency[q.back()] >= consistency[i]) {
      q.pop_back();
    }
    q.push_back(i);
    if (i >= k - 1) {
      result.push_back(consistency[q.front()]);
    }
  }
  for (int i : result) {
    cout << i << " ";
  }
  cout << endl;
  return 0;
}
```