Федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский университет ИТМО»

Факультет программной инженерии и компьютерной техники

Отчет по блоку №3 (яндекс контест)

По Алгоритмам и структурам данных

Выполнил:

Ступин Тимур Русланович

Группа № P3108

Поток № 1.3

Преподаватель:

Тараканов Денис Сергеевич

Санкт-Петербург 2025

Содержание

Задача 1 Машинки

Решение

Начнём с того, что пока на полу есть место алгоритм вполне понятен: нужно просто класть машинки на пол. Но когда места больше нет, возникает вопрос какую машинку убрать. Идея состоит в том, чтобы убирать ту машинку, которая понадобится позже всего. Это значение можно рассчитать, так как порядок снятия машинок известен заранее. При считывании входных данных заполним `vector<queue<int>> v` составив для каждого типа машинки очередь из индексов в порядке их появления. В ходе основного прохода по массиву будем поддерживать `set<pair<int, int>, greater<>> s` в котором будут хранится пары вида: {индекс-следующего-вхождения; тип-машинки}, отсортированные по убыванию индекса вхождения. Состояние данного множества соответствует текущему состоянию пола. В ходе основного прохода будем выполнять следующее:

- Для начала проверим есть ли текущий элемент на полу. Для этого достаточно проверить наличие соответствующей пары в множестве `s`, при этом индекс следующего вхождения лежит в начале очереди рассматриваемого элемента.
- Если элемент есть на полу, то его нужно удалить, так как теперь его значение индекса следующего вхождения некорректно (мы уже прошли соответствующий индекс)
- Если же рассматриваемого элемента нет в списке, то нужно увеличить счётчик изменений, так как в любом случае придётся добавлять машинку на пол. Также нужно проверить, если ли место на полу, и если его нет, то произвести удаление первого элемента (в соответствии с параметрами сортировки он как раз будет иметь максимальный индекс следующего вхождения).
- В конце в любом случае производим обновление состояние пола, добавляя текущий элемент с индексом следующего вхождения из очереди.

Сложность алгоритма

- По времени: Основной проход по массиву выполняется за $O(N)$, при этом в цикле производятся операции поиска, удаления и добавление с `set` которые стоят $O(\log N)$, что даёт итоговую сложность $O(N \log N)$
- По памяти: Программа хранит массив входных дынных размера N, вектор очередей, в котором суммарно N простейших элементов, а также множество для хранения

содержимого пола, которое в худшем случае может иметь размер K. Итоговая сложность $O(N)$.

Итого:

- По времени: $O(N \log N)$
- По памяти: $O(N)$

Код

```cpp
#include <bits/stdc++.h>
using namespace std;

constexpr int INF = 1e9;

int main() {
  int n, p;
  size_t k;
  cin >> n >> k >> p;
  vector<int> d;
  vector<queue<int>> v(n);
  for (int i = 0; i < p; i++) {
    int a;
    cin >> a;
    a--;
    d.push_back(a);
    v[a].push(i);
  }
  int ans = 0;
  set<pair<int, int>, greater<>> s;
  for (auto a : d) {
    auto it = s.find({v[a].front(), a});
    if (it == s.end()) {
      if (s.size() == k) {
        s.erase(s.begin());
      }
      ans++;
    } else {
      s.erase(it);
    }
    v[a].pop();
    int new_next = v[a].empty() ? INF : v[a].front();
    s.insert({new_next, a});
  }
  cout << ans << endl;
  return 0;
}
```

Задача 2 Гоблины и очереди

Решение

Для решения задачи нужна очередь, которая будет помимо базовых операций поддерживать вставку в середину. Такую очередь можно реализовать с использованием двусвязного списка. При этом будем постоянно поддерживать указатель на центральный элемент, чтобы выполнять вставку за $O(1)$.

Сложность алгоритма

- По времени: операции с очередью работают за $O(1)$, сам проход по входным данным занимает $O(N)$. Итоговая сложность $O(N)$
- По памяти: двусвязный список занимает $O(N)$ по памяти

Итого:

- По времени: $O(N)$
- По памяти: $O(N)$

Код

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
  list<int> l;
  auto c = l.begin();
  int n;
  char t;
  int i;
  cin >> n;
  while (n--) {
    cin >> t;
    if (t == '-') {
      cout << l.front() << endl;
      l.pop_front();
      if (l.empty() || l.size() == 1)
        c = l.begin();
      else if (l.size() % 2 == 1)
        ++c;
      continue;
    }
    cin >> i;
    if (t == '+') {
      l.push_back(i);
      if (l.size() == 1)
        c = l.begin();
      else if (l.size() % 2 == 1)
        ++c;
    } else {
      auto it = c;
      l.insert(++it, i);
      if (l.size() == 1)
        c = l.begin();
      else if (l.size() % 2 == 1)
        ++c;
    }
  }
  return 0;
}
```

Задача 3 Менеджер памяти-1

Решение

Заведём структуру данных `block` которая будет соответствовать блоку памяти. Она будет хранить в себе адрес начала блока (`l`), его размер (`sz`), а также информацию о занятости блока (`is_free`). Сортировка данной структуры будет происходить по адресу начала `l`. Определим также двусвязный список блоков, в котором будем хранить все блоки, причём в том порядке в котором они идут в памяти. В дополнение к данному списку, определим `multimap` для свободных блоков, в котором ключом будет размер блока, а значением указатель на элемент списка, соответствующий этому блоку. Данная структура позволит эффективно находить оптимальный свободный блок по запрошенному размеру. Также для хранения истории запросов понадобится массив указателей на узлы списка. Изначально список и множество инициализированы одним свободным блоком размера N.

Принцип работы алгоритма заключается в следующем.

Когда поступает запрос на выделение памяти, при помощи метода `lower_bound` происходит выбор блока минимального размера, который удовлетворял бы условию. Если такой блок не найден, запрос игнорируется. Если блок точно совпадает по размеру с запрашиваемым, то он просто удаляется из множества, его состояние меняется на занятое, и указатель на него добавляется в историю. Иначе создаётся новый блок требуемого размера, он помечается как занятый и добавляется перед выбранным свободным блоком. Размер и адрес свободного блока изменяется соответственно. Также обновляется состояние множества свободных блоков, старый блок удаляется, новый оставшийся кусочек добавляется. Указатель на вновь созданный блок добавляется в историю.

Когда поступает запрос на освобождение памяти, для начала из истории извлекается указатель на соответствующий блок. После этого происходит рассмотрение левого и правого соседей данного блока и попытка объединения с ними, в случае если они свободны. Наконец состояние полученного блока устанавливается как свободное и он добавляется в множество. В историю же записывается нулевой указатель, чтобы сохранить индексацию.

Сложность алгоритма

- По времени: Для каждого из M запросов, происходят операции со списком за $O(1)$ и операции с множеством за $O(\log N)$. Итоговая сложность $O(M \log N)$
- По памяти: В худшем случае размеры списка и множества будут $O(N)$

Итого:

- По времени: $O(M \log N)$
- По памяти: $O(N)$

Код

```cpp
#include <bits/stdc++.h>
using namespace std;

struct block {
  long long l;
  long long sz;
  bool is_free;
  block() {
    l = sz = -1;
    is_free = false;
  }
  block(const long long l, const long long sz, const bool is_free) {
    this->l = l;
    this->sz = sz;
    this->is_free = is_free;
  }
};

struct node {
  block* data;
  node* next;
  node* prev;
  node() {
    data = nullptr;
    next = prev = nullptr;
  }
  explicit node(block* data) {
    this->data = data;
    next = prev = nullptr;
  }
};

void remove_from_multimap(const node* node, multimap<long long, struct
↪  node*>& mm) {
  auto [fst, snd] = mm.equal_range(node->data->sz);
  for (auto it = fst; it != snd; ++it) {
    if (it->second == node) {
      mm.erase(it);
      break;
    }
  }
}

int main() {
  long long n, m;
  cin >> n >> m;
  multimap<long long, node*> free_blocks_map;
  vector<node*> history;
  auto blocks_list = new node(new block(0, n, true));
```

```
49    free_blocks_map.insert(make_pair(blocks_list->data->sz, blocks_list));
50    while (m--) {
51      long long t;
52      cin >> t;
53      if (t > 0) {
54        auto it = free_blocks_map.lower_bound(t);
55        if (it == free_blocks_map.end()) {
56          cout << -1 << endl;
57          history.push_back(nullptr);
58          continue;
59        }
60        auto cur_block = it->second;
61        if (it->first == t) {
62          free_blocks_map.erase(it);
63          cur_block->data->is_free = false;
64          history.push_back(cur_block);
65          cout << cur_block->data->l + 1 << endl;
66          continue;
67        }
68        // create new filled block before current
69        auto new_block = new node(new block(cur_block->data->l, t, false));
70        new_block->prev = cur_block->prev;
71        new_block->next = cur_block;
72        if (cur_block->prev != nullptr)
73          cur_block->prev->next = new_block;
74        cur_block->prev = new_block;
75        // update current free block
76        cur_block->data->l += t;
77        cur_block->data->sz -= t;
78        // update map
79        free_blocks_map.erase(it);
80        free_blocks_map.insert(make_pair(cur_block->data->sz, cur_block));
81        history.push_back(new_block);
82        cout << new_block->data->l + 1 << endl;
83      }
84      if (t < 0) {
85        t = -t;
86        t--;
87        auto cur_block = history[t];
88        if (cur_block == nullptr) {
89          history.push_back(nullptr);
90          continue;
91        }
92        auto l_block = cur_block->prev;
93        if (l_block != nullptr && l_block->data->is_free) {
94          remove_from_multimap(l_block, free_blocks_map);
95          cur_block->prev = l_block->prev;
96          if (l_block->prev != nullptr)
97            l_block->prev->next = cur_block;
98          cur_block->data->sz += l_block->data->sz;
```

```
        cur_block->data->l = l_block->data->l;
        delete l_block->data;
        delete l_block;
      }
      auto r_block = cur_block->next;
      if (r_block != nullptr && r_block->data->is_free) {
        remove_from_multimap(r_block, free_blocks_map);
        cur_block->next = r_block->next;
        if (r_block->next != nullptr)
          r_block->next->prev = cur_block;
        cur_block->data->sz += r_block->data->sz;
        delete r_block->data;
        delete r_block;
      }
      if (blocks_list == l_block || blocks_list == r_block) {
        blocks_list = cur_block;
      }
      cur_block->data->is_free = true;
      free_blocks_map.insert(make_pair(cur_block->data->sz, cur_block));
      history.push_back(nullptr);
    }
  }
  auto ptr1 = blocks_list;
  auto ptr2 = blocks_list->prev;
  while (ptr1) {
    const node* temp = ptr1;
    ptr1 = ptr1->next;
    delete temp->data;
    delete temp;
  }
  while (ptr2) {
    const node* temp = ptr2;
    ptr2 = ptr2->prev;
    delete temp->data;
    delete temp;
  }
  return 0;
}
```

Задача 4 Минимум на отрезке

Решение

В данное задаче можно использовать способ модификации очереди, который позволяет за $O(1)$ находить минимум в очереди, осуществлять добавление новых элементов и взятие элементов из очереди. Единственной особенностью данной модификации очереди является то, что при попытке взять элемент из начала очереди нужно заранее знать каким этот элемент должен быть, так как очередь хранит не все добавленные в неё элементы. Однако для данной задачи это не является проблемой, ведь мы всегда знаем какой элемент хотим извлечь.

Принцип работы данной модификации очереди основан на идее хранить в ней только те значения, которые нужны для определения минимума. Будем поддерживать неубывающий порядок элементов в очереди. Таким образом голова очереди и будет минимальным элементом. При добавлении элемента в конец очереди, чтобы сохранить состояние не убывания, нужно последовательно удалять элементы с конца, пока последней элемент не будет меньше либо равен добавляемому. Извлечение же первого элемента также тривиально, единственно что нужно проверить, это не пустота очереди и равенство значения первого элемента ожидаемому, так как на момент извлечения он может быть уже удалён.

Так как алгоритм предполагает обращение как к началу так и к концу очереди то в решении был использован дек.

Сам алгоритм заключается в следующем. Вначале происходит добавление первых `k` элементов. Далее в цикле последовательно производится вывод минимума, удаление `i-k`-го элемента (которые теперь уходит из окна) и добавление нового элемента.

Сложность алгоритма

- По времени: Каждый из элементов массива будет один раз добавлен в очередь и один раз извлечён из неё. Тем самым получаем сложность $O(N)$
- В памяти хранится входной массив а также очередь, размер которой не превышает N. Сложность $O(N)$

Итого:

- По времени: $O(N)$
- По памяти: $O(N)$

Код

```cpp
#include <bits/stdc++.h>
using namespace std;

void add(deque<int>& q, const int a) {
  while (!q.empty() && q.back() > a) {
    q.pop_back();
  }
  q.push_back(a);
}

void remove(deque<int>& q, const int a) {
  if (!q.empty() && q.front() == a) {
    q.pop_front();
  }
}

int get_min(const deque<int>& q) {
  return q.front();
}

int main() {
  deque<int> q;
  int n, k;
  cin >> n >> k;
  vector<int> v(n);
  for (int i = 0; i < n; i++) {
    cin >> v[i];
  }
  for (int i = 0; i < k; i++) {
    add(q, v[i]);
  }
  for (int i = k; i < n; i++) {
    cout << get_min(q) << " ";
    remove(q, v[i - k]);
    add(q, v[i]);
  }
  cout << get_min(q) << endl;
  return 0;
}
```