Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №3
по “Алгоритмам и структурам данных”
Базовые задачи

Выполнил:
Студент группы P3206
Сорокин А.Н.
Преподаватели:
Косяков М.С.
Тараканов Д.С.

Санкт-Петербург
2024

Задача №I “Машинки”

Решение за O(n log n).
~~Почему нельзя просто все машинки на пол вывалить~~

В чем идея: чтобы достичь минимального количества действий необходимо догадываться до того, какая машинка будет востребована раньше, а какая – позже. Для этого можно заранее высчитать “расстояние” между запросами на одну и ту же машинку. В ситуации, когда одну из машинок необходимо заменить, можно просто сравнить расстояния для каждой машинки, и убрать машинку с наибольшим расстоянием.

На примере из задачи:
3 2 7
1
2
3
1
3
1
2

На полу машинки 1 и 2. Петя хочет машинку 3. Машинка 2 ему понадобится не скоро – в самую последнюю очередь. Машинка 1 же, напротив, понадобится ему сразу после 3 машинки, и убирать ее было бы невыгодно. Следовательно, заменим 2 на 3 и сэкономим маме 1 действие, так как до 1 машинки Петя дотянется сам.

Теперь о структурах.
Прекальк
Необходимо по обращению к машинке сразу узнавать расстояние до следующей. Требование $cars[car_i]=distance$.
Однако, если, предположим, машинка 1 понадобится в 1, 5 и 10 очередь, у нас возникнет нарушение инъекции: $cars[1]=5$ и $cars[1]=10$. Простой map здесь не подойдет. Так как ключи будут повторяться, нам нужен multimap. Multimap позволяет хранить одинаковые ключи. Сортировка

нам в данном случае не нужна, однако быстрый доступ необходим – воспользуемся **unordered_multimap**.
Далее, при преподсчете расстояний нам необходимо всегда иметь доступ к последнему запросу машинки N. Зачем? Потому что запрос может оказаться не последним.
Предположим, машинка 1 понадобилась в 1 очередь. Мы предполагаем, что она больше не понадобится, и ставим $cars[1]=10^7$. Однако на 5 итерации она понадобилась, и мы изменяем $cars[1]=5$, и тогда предполагаем, что она после 5 итерации уже не понадобится (ставим следующий $cars[1]=10^7$). Нам необходимо по номеру машинки как можно быстрее обращаться к последнему возникновению машинки в очереди, чтобы при необходимости изменить его. Для этого подходит **unordered_map**, где ключом является номер машинки, а значением – итератор на нужную пару в unordered_multimap.
Наконец, нам необходима структура, поддерживающая изначальный порядок элементов. Такой структурой является **vector**. Он будет использоваться для быстрого доступа к элементам unordered_multimap.

Сама задача
Необходимо знать, какие машинки находятся на полу в данный момент. Нам совершенно не важен порядок машинок на полу – только их номера. Очевидно, что на полу не могут быть 2 одинаковые машинки. Мы будем часто вставлять и удалять элементы в эту структуру, поэтому данные операции должны быть быстрыми. Для данной задачи подходит **unordered_set**.
Теперь необходима структура, которая будет обеспечивать быстрый поиск максимума и самобалансироваться при вставке/удалении элементов. В этой структуре мы будем хранить пары из unordered_multimap, где максимумом будет являться пара из машинки с максимальным расстоянием до следующего запроса – по сути проекция того, какие машинки находятся на полу. Для данной задачи подходит **priority_queue**.

Итоговый набор структур:
unordered_map<int, unordered_multimap<int, int>::iterator>
unordered_multimap<int, int>
vector<unordered_multimap<int, int>::iterator>
unordered_set<int>
priority_queue<pair<int, int>>

Код решения:

```
#include <fstream>
#include <iostream>
#include <map>
#include <queue>
#include <unordered_map>
#include <unordered_set>
#include <vector>
using namespace std;

class Compare {
public:
  bool operator()(const pair<int, int> a, const pair<int, int> b) {
    return (a.second == b.second) ? a.first > b.first : a.second < b.second;
  }
};
int main() {
  ifstream infile("input.txt");
  int n, k, p;
  infile >> n >> k >> p;

  unordered_map<int, unordered_multimap<int, int>::iterator> requests;
  unordered_multimap<int, int> currentNextPair;
  vector<unordered_multimap<int, int>::iterator> requestVector;

  for (int i = 0; i < p; ++i) {
    int car;
    infile >> car;
    auto findCar = requests.find(car);
```

```
    if (findCar == requests.end()) {
      auto carNext = currentNextPair.insert({car, 1e7});
      requests[car] = carNext;
      requestVector.push_back(carNext);
    } else {
      findCar->second->second = i;
      auto carNext = currentNextPair.insert({car, 1e7});
      requests[car] = carNext;
      requestVector.push_back(carNext);
    }
  }

  int numOp = 0, currentK = 0;
  priority_queue<pair<int, int>, vector<pair<int, int>>, Compare> pq;
  unordered_set<int> carsOnFloor;
  for (int i = 0; i < requestVector.size(); ++i) {
    auto car = requestVector[i];
    if (carsOnFloor.find(car->first) == carsOnFloor.end()) {
      if (currentK >= k) {
        int carToRemove = pq.top().first;
        carsOnFloor.erase(carToRemove);
        pq.pop();
        --currentK;
      }
      carsOnFloor.insert(car->first);
      pq.push({car->first, car->second});
      ++numOp;
      ++currentK;
    } else {
      auto car = requestVector[i];
      pq.push({car->first, car->second});
    }
  }
  cout << numOp;
  return 0;
}
```

Задача №J “Гоблины и очереди”

Решение за O(n).
Просто сделать то, что требуется в задаче.
Предыдущее решение за O(n^2) полагалось на использование вектора, который, во-первых, медленно удаляет элемент из начала, во-вторых, медленно вставляет элемент в середину. Дорого по времени.

Подумаем о структурах данных, которые позволяют делать быстрые вставки в начало/конец. Такой структурой является **list**. Особенности реализации листа позволяют не беспокоиться о перемещении элементов при вставке/удалении, как это происходит в векторе.

Однако вставка в середину листа все равно происходит за линию.
Дело не в самой вставке (так как она происходит за константу), а в поиске серединного элемента, который ищется за линию (Floyd’s Tortoise and Hare Algorithm).

Тем не менее, мы можем вместе с листом хранить указатель на серединный элемент и двигать его по необходимости.
Это поможет нам совершать все операции за константу, что значительно ускорит алгоритм.

(P.S. просьба написать тесты такие, чтобы квадрат не проходил)
Код решения:

```
#include <fstream>
#include <iostream>
#include <vector>

using namespace std;

class Node {
public:
  int val;
  Node *prev;
```

```
  Node *next;
  Node(int v, Node *n, Node *p) {
    val = v;
    next = n;
    prev = p;
  }
};
class GoblinQueue {
  Node *queueBegin;
  Node *queueEnd;
  Node *queueMid;
  int size;

public:
  void enterQueue(int val) {
    Node *newNode = new Node(val, nullptr, queueEnd);
    if (queueEnd) {
      queueEnd->next = newNode;
    } else {
      queueBegin = newNode;
      queueMid = queueBegin;
    }
    if (size % 2 == 0 && queueMid->next)
      queueMid = queueMid->next;
    queueEnd = newNode;
    ++size;
  }

  void enterMid(int val) {
    if (size <= 1) {
      enterQueue(val);
      return;
    }
    if (size % 2 == 0) {
      Node *buf = queueMid->next;
      Node *prevMid = queueMid;
```

```
    Node *newNode = new Node(val, buf, prevMid);
    queueMid = newNode;
    prevMid->next = queueMid;
    buf->prev = queueMid;

  } else {
    Node *buf = queueMid->next;
    Node *newNode = new Node(val, buf, queueMid);
    queueMid->next = newNode;
    buf->prev = newNode;
  }
  ++size;
}

void leaveQueue() {
  cout << queueBegin->val << "\n";
  if (!queueBegin->next) {
    queueEnd = nullptr;
    queueMid = nullptr;
    queueBegin = nullptr;
    size = 0;
    return;
  }
  Node *newNode = queueBegin->next;
  queueBegin = newNode;
  queueBegin->prev = nullptr;
  if (size % 2 == 0 && queueMid) {
    queueMid = queueMid->next;
  }
  --size;
}
GoblinQueue() {
  queueBegin = nullptr;
  queueEnd = nullptr;
  queueMid = nullptr;
  size = 0;
```

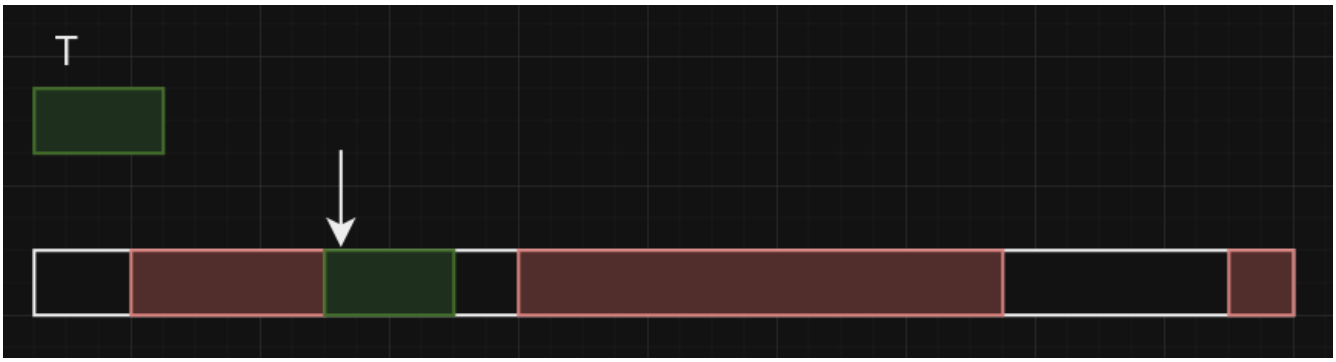
```
  }
};

int main() {
  GoblinQueue *gq = new GoblinQueue();
  ifstream infile("input.txt");
  int n;
  infile >> n;
  for (int i = 0; i < n; ++i) {
    char mode;
    infile >> mode;
    if (mode != '-') {
      int num;
      infile >> num;
      if (mode == '+') {
        gq->enterQueue(num);
      } else if (mode == '*') {
        gq->enterMid(num);
      }
    } else {
      gq->leaveQueue();
    }
  }
  return 0;
}
```

Задача №K “Менеджер памяти-1”

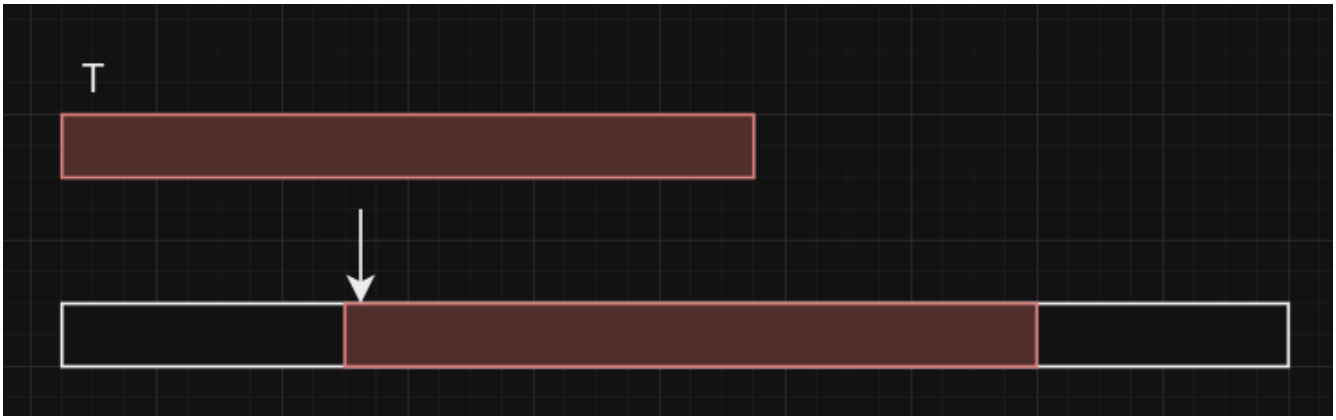
Решение за O(n log n).
Задача буквально сводится к написанию malloc() и free().
Требования к malloc():
1) вернуть указатель на первый блок в последовательности, если удалось найти последовательность свободных блоков подходящей длины
2) иначе вернуть -1
Требования к free():

1) найти и освободить последовательность блоков, начиная с указателя на блок последовательности
2) если память не была аллоцирована, ничего не делать
С malloc все ясно. Если память представить как последовательность блоков, то malloc(T) ищет в этой последовательности отрезок пустых блоков длины $len \geq T$.



Пустые участки памяти начинаются с какого-то блока. Поскольку у нас дан размер T, необходимо быстро получать индекс блока такого же или большего размера. Учитывая, что участки, находящиеся в разных местах памяти, могут иметь одинаковый размер, выберем структуру **multimap<int, int>**, где ключ – размер участка свободных блоков, а значение – номер блока, с которого начинается участок. Использование сортированой мапы дает доступ к функции lower_bound(), которая позволяет быстро находить заданный блок.
Для free() же необходимо знать, сколько памяти надо освободить по указателю на начало участка. Поскольку по одному адресу не могут находиться сразу 2 блока памяти, воспользуемся **map<int, int>.**
Наконец, воспользуемся структурой **vector<pair<int, int>>** как связующим звеном между мапами – это поможет нам по номеру запроса выяснять, был ли аллоцирован участок памяти, и если да, то с какого блока.

Проблема заключается в обработке подобного случая для free():



При освобождении участка памяти может оказаться, что вокруг него находятся свободные участки памяти. Чтобы malloc() при следующем поиске подходящего участка не “промахнулся”, участки необходимо соединить в один. Всего существует 4 случая:

- пустые участки слева и справа от занятого
- пустой участок слева от занятого
- пустой участок справа от занятого
- пустых участков нет (тривиальный случай)

Здесь поможет метод lower_bound(), имеющийся у сортированных мап и метод prev(), позволяющий получать итератор на предыдущий блок.

Во время работы программы будет происходить очень много операций вставки, удаления, поиска соседних элементов в структурах, поэтому необходимо, чтобы данные операции производились как можно быстрее.
Методы мап работают за $\log n$, что дает нам ассимптотику O(nlogn).

Код решения:

```
#include <fstream>
#include <ios>
#include <iostream>
#include <iterator>
#include <map>
#include <string>
#include <utility>
#include <vector>
```

```
#define ll long long
#define PLL pair<ll, ll>
using namespace std;
multimap<ll, ll> memoryMapSizeToPointer;
map<ll, ll> memoryMapPointerToSize;

void insertIntoMaps(const PLL p) {
  memoryMapSizeToPointer.insert({p.second, p.first});
  memoryMapPointerToSize[p.first] = p.second;
}

void removeSize(map<ll, ll>::iterator it) {
  auto itBySize = memoryMapSizeToPointer.find(it->second);
  while (itBySize->second != it->first)
    ++itBySize;
  memoryMapSizeToPointer.erase(itBySize);
  memoryMapPointerToSize.erase(it);
}

ll allocateMemory(ll requestedSize, vector<PLL> *requestVector) {
  auto optimalBlockPointer =
memoryMapSizeToPointer.lower_bound(requestedSize);
  if (optimalBlockPointer != memoryMapSizeToPointer.end()) {
    // block found
    requestVector->push_back({optimalBlockPointer->second,
requestedSize});
    ll newSize = optimalBlockPointer->first - requestedSize,
       newPointer = optimalBlockPointer->second + requestedSize;

    memoryMapSizeToPointer.erase(optimalBlockPointer);
    memoryMapPointerToSize.erase(requestVector->back().first);

    insertIntoMaps({newPointer, newSize});
    return requestVector->back().first;
  }
  // block not found
```

```
  requestVector->push_back({-1, requestedSize});
  return -1;
}

void mergeLeftRight(PLL p, map<ll, ll>::iterator l, map<ll, ll>::iterator r) {
  ll start = l->first;
  ll size = l->second + p.second + r->second;
  removeSize(l);
  removeSize(r);
  insertIntoMaps({start, size});
}

void mergeLeft(PLL p, map<ll, ll>::iterator l) {
  ll start = l->first;
  ll size = l->second + p.second;
  removeSize(l);
  insertIntoMaps({start, size});
}

void mergeRight(PLL p, map<ll, ll>::iterator r) {
  ll start = p.first;
  ll size = p.second + r->second;
  removeSize(r);
  insertIntoMaps({start, size});
}

void freeMemory(ll requestNum, vector<PLL> *requestVector) {
  PLL possibleSatisfiedRequest = requestVector->at(requestNum);
  requestVector->push_back({0, 0});
  if (possibleSatisfiedRequest.first != -1) {
    // need to find both pointed block and his neighbours

    auto rightBlock = memoryMapPointerToSize.lower_bound(
        possibleSatisfiedRequest.second + possibleSatisfiedRequest.first);
    auto leftBlock = (rightBlock == memoryMapPointerToSize.begin())
                     ? memoryMapPointerToSize.end()
```

```
                    : prev(rightBlock);

    bool predL =
        leftBlock != memoryMapPointerToSize.end() &&
        leftBlock->first + leftBlock->second == possibleSatisfiedRequest.first;
    bool predR =
        rightBlock != memoryMapPointerToSize.end() &&
        possibleSatisfiedRequest.first + possibleSatisfiedRequest.second ==
            rightBlock->first;

    if (predL && predR) {
      mergeLeftRight(possibleSatisfiedRequest, leftBlock, rightBlock);
      return;
    }
    if (predL) {
      mergeLeft(possibleSatisfiedRequest, leftBlock);
      return;
    }
    if (predR) {
      mergeRight(possibleSatisfiedRequest, rightBlock);
      return;
    }
    insertIntoMaps(
        {possibleSatisfiedRequest.first, possibleSatisfiedRequest.second});
  }
}

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(NULL);
  ifstream infile("input.txt");
  ll N, M, T;
  infile >> N >> M;
  string s;
  vector<PLL> requestVector;
  insertIntoMaps({1, N});
```

```
  for (ll i = 1; i <= M; ++i) {
    infile >> T;
    if (T > 0) {
      s += to_string(allocateMemory(T, &requestVector)) + '\n';
    } else {
      freeMemory(-T - 1, &requestVector);
    }
  }
  cout << s;
  return 0;
}
```

Задача №L “Минимум на отрезке”

Решение за O(n log n).
Для того, чтобы выводить минимум на отрезке необходимо знать минимум на каждом отрезке. Обычный поиск минимума на каждом сдвиге отрезка вылился бы в квадрат по времени, следовательно, необходим более быстрый подход, позволяющий сразу находить минимум.

В качестве симуляции самого отрезка можно использовать **queue<int>.** Это отличная структура для данной задачи – при сдвиге отрезка в конец очереди добавляется новый элемент и из начала выходит старый.

Предположим, при подсчете первых k элементов мы уже нашли минимум. Сдвинем отрезок. Может возникнуть ситуация, когда наш минимум был самым левым элементом в отрезке и потому выпал, требуя поиск нового минимума. Необходима структура, сортирующая элементы, чтобы можно было быстро находить новый минимум. Очевидно, что в этой структуре элементы могут повторяться. Берем **multiset<int>**.

В чем идея:
Всякий раз, когда из queue необходимо удалить элемент, этот элемент ищется в multiset и из него удаляется. Если таких элементов в multiset

несколько, удаляется первый из них. Затем в queue и multiset вставляется только что вошедший в отрезок элемент. Далее первый в multiset элемент выводится на экран.
Фактически, наш отрезок симулируется 2 структурами:

- структура, где элементы идут в том же порядке, в каком они находятся в отрезке
- структура, где элементы расположены в порядке возрастания

Код решения:

```
#include <ios>
#include <iostream>
#include <queue>
#include <set>
using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(NULL);
  int n, k, x;
  cin >> n >> k;
  multiset<int> ms;
  queue<int> q;
  for (int i = 0; i < k; ++i) {
    cin >> x;
    q.push(x);
    ms.insert(x);
  }
  cout << *ms.begin() << " ";
  for (int i = k; i < n; ++i) {
    cin >> x;
    int toRemove = q.front();
    q.pop();
    ms.erase(ms.find(toRemove));
    q.push(x);
    ms.insert(x);
    cout << *ms.begin() << " ";
```

```
  }
  return 0;
}
```