Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ **ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ**

КАФЕДРА **«ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ЭВМ И ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ» (ИУ7)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.03.01 Информатика и вычислительная техника**

О Т Ч Е Т

по лабораторной работе № 6

Название: Решение задачи коммивояжёра

Дисциплина: Анализ алгоритмов

Студент ИУ7-52Б _____ В.А. Иванов
(Группа) (Подпись, дата) (И.О. Фамилия)

Преподаватель _____ _____
(Подпись, дата) (И.О. Фамилия)

Москва, 2020

Оглавление

Введение

В данной лабораторной реализуются и оцениваются алгоритмы решения задачи коммивояжёра.

Задача коммивояжёра является одним из самых известных примеров NP-полной задачи. Она заключается, в поиске наиболее выгодного маршрута, проходящего однократно через все вершины графа, кроме начальной вершины, которая должна оказаться и конечной.

Интерес к данной задачи обусловлен тем, что на данный момент не существует алгоритма, способного находить её решение за полиномиальное время в зависимости от количества вершин. При этом, известны алгоритмы, которые способны найти маршрут, который по длине будет достаточно приближён к по наилучшему решению. Этот факт позволяет принимать подобные решения в практике, когда "почти идеальное"решение более чем удволетворяет требованиям решаемой проблемы. Примером подобной задачи является поиск маршрута пайки печатной платы, при котором манипулятор-пайщик проделает наименьший путь между контактами. В данном случае возможно использование достаточно короткого, но не лучшего маршрута.

В данной лабораторной работе в качестве алгоритмов поиска решения будут рассмотрены:

- поиск полным перебором;
- поиск муравьиным алгоритмом.

В первом случае будет измерения длины всех возможных маршрутов. Это является достаточно затратным решением, но гарантированно будет получено наилучшее решение.

Второй алгоритм является воплощением механизма, созданого самой природой – поведением колонией муравьёв. Суть поведения

каждого муравья заключается в использовании опыта ранее ходивших муравьёв в принятии решения о выборе следующей вершины. Опыт задаётся при помощи откладывания на рёбрах графа феромона, который тем больше, чем оптимальнее маршрут проходящий через данное ребро. Таким образом, спустя множество поколений муравьёв, пользующихся знаниями своих предков, можно выявить наиболее оптимальный вариант прохождения.

1. Аналитическая часть

1.1. Цель и задачи работы

Целью лабораторной работы является проведение сравнительного анализ метода полного перебора и эвристического метода на базе муравьиного алгоритма.

Выделены следующие задачи лабораторной работы:

- описание задачи коммивояжёра;
- описание и реализация метода полного перебора и метода на базе муравьиного алгоритма для решения задачи коммивояжёра;
- проведение параметризации муравьиного метода (определение параметров, для которых метод даёт наилучшие результаты на выбранных классах задач).

1.2. Описание задачи коммивояжёра

Задача заключается в поиске гамильтонова цикла (т.е. замкнутый путь, проходящий через каждую вершину ровно один раз) на неориентированном графе G, с количеством вершин N[1]. Вес рёбер можно задать с помощью квадратной матрицы D размером N, где D_{ij} равняется стоимости перехода из вершины i в j. Сами маршруты можно представить как массив M длиной $N + 1$, где M_i - вершина, посещённая в i-ю очередь.

1.3. Поиск полным перебором

Данный алгоритм составляет все возможные маршруты, начинающиеся из нулевой вершины, и измеряет длину каждого из вариантов. Маршрут с минимальной длиной гарантированно будет являться решением поставленной задачи.

Каждый маршрут начинается и заканчивается в нулевой вершине, потому что каждый путь обязательно будет содержать эту вершину, а так как это цикл, то любая последовательность посещения вершин может быть преобразована в маршрут из нулевой вершины, обладающий той же длиной. Поэтому, не имеет смысла рассмотрение иных начальных вершин.

1.4. Поиск муравьиным алгоритмом

Алгоритм симулирует поведение N муравьёв, которые вместе называются колонией. Колония существует max_t дней. В начале t-го дня по всем вершинам выставляется по муравью. Каждый муравей совершает попытку построить гамельтонов цикл. В случае удачного построения цикла на пройденых рёбрах им выставляется определённое количество феромона. После этого наступает t-я ночь, в которой часть феромона улетучивается, после чего начинается следующий день. Количество феромона на ребре $i - j$ в момент времени t обозначается как $\tau_{ij}(t)$

После симуляции всех дней алгоритм выдаёт в качестве решения наикратчайший маршрут среди всех пройденых. Стоит оговорить то, далеко не всегда этот маршрут будет являться правильным решением поставленной задачи, так как вероятнее всего, алгоритм проверит все возможные пути в графе.

Рассмотрим принцип формирования маршрута. Оказываясь в очередной вершине i (кроме заключительной), муравей совершает выбор одной из доступных для перехода вершин, которые ещё не были посещены. Выбор основывается на величине, определяемой формулой 1.1

$$P_{ij}(t) = [\tau_{ij}(t)]^{\alpha}/[D_{ij}]^{\beta} \tag{1.1}$$

где α, β - коэффициенты стадности и жадности.

Полученные значения можно пронормировать, поделив их на сумму всех величин, в таком случае получится вероятность перехода в j-ю вершину. Основываясь на этом муравей делает случайный выбор следующей вершины с заданными вероятностями.

Каждый муравей прошедший полный маршрут увеличивает значение феромона в посещённых рёбрах на величину, указанной в формуле 1.2

$$\Delta\tau_{ijk}(t) = Q/L_k(t) \tag{1.2}$$

где k - номер муравья, Q - параметр, по порядку приближенный к ожидаемой минимальной длине пути, $L_k(t)$ - путь пройденый k-м муравьём в день t, $\tau_{ijk}(t)$ - приращение феромона от k-го муравья в день t

Для акцентирования феромонов на лучшем пути также используется EL элитных муравьёв, которые каждый проходят по наикратчайшему маршруту на данный момент и, как и обычные муравьи, оставляют феромоны по формуле1.2

После учёта всех приращений происходит испарение феромона, т.е. значение феромона в следующий день вычисляется как 1.3

$$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \sum_{k=1}^{N} \Delta\tau_{ijk}(t) \tag{1.3}$$

где ρ - коэффициент испарения феромона ($\rho \in [0, 1]$).

Вывод

Результатом аналитического раздела стало определение цели и задач работы, описана задача коммивояжёра и алгоритмы поиска.

2. Конструкторская часть

Рассмотрим описанные алгоритмы поиска ключей в словаре. Пусть производится поиск ключа key в словаре a длиной len.

2.1. Поиск полным перебором

Алгоритм проходит от 1-го элемента до len-го в поиске полного совпадения ключа. В случае, если ключ был найден, производится досрочный выход из поиска записи. Если после прохода по всем элементам словаря совпадение не было обнаружено, сообщается о том, что ключ не был найден.

Схема алгоритма приведена на рисунке 2.1

2.2. Поиск половинным разбиением

Данный метод может применяться только для упорядоченного массива. Описание работы приведено для случая, когда алгоритм повторяет операцию поиска для интервала индексов массива, изначально содержащим все индексы от 0 до $len - 1$. Эти два значения называются левой (l) и правой (r) границей массива соответственно. Алгоритм выбирает элемент с индексом $mid = round((l + r)/2)$ и сравнивает его ключ с искомым. В случае, если искомый ключ больше ключа mid, левая граница ставится на mid+1, если меньше, то праваой границе присваивается mid-1. Если произошло совпадение, то ключ найден и происходит выход из функции. Если правая граница оказалась меньше левой, то это говорит о том, что искомого ключа в словаре нет.

Схема алгоритма приведена на рисунке 2.2

2.3. Поиск с сегментами

Для использования данного поиска требуется предварительно провести сегментацию словаря (схема алгоритма приведена на рисунке 2.3.). В данной работе сегментация производится по последней цифре номера карты, так как в этом случае элементы словаря будут равномерно распределены между сегментами.

Структура сегмента состоит из двух полей:

1. ключ - последняя цифра для всех ключей словаря;

2. массив пар - словарь из значений соответсвующих ключу.

Алгоритм использует вышеописанный полный поиск сначала для нахождения нужного сегмента, а после ищет нужный ключ среди элементов сегмента.

Схема алгоритма приведена на рисунке 2.4

2.4. Требования к программному обеспечению

Для полноценной проверки и оценки алгоритмов необходимо выполнить следующее.

1. Предоставить возможность ввода искомого ключа и проверяемого алгоритма.

2. Реализовать функцию замера процессорного времени, затраченного функциями и подсчёта статистических данных.

2.5. Заготовки тестов

При проверке алгоритма необходимо будет использовать следующие классы тестов:

- поиск отсутствующего номера карты;

- поиск первого и последнего номера в словаре.

Вывод

Результатом конструторской части стало схематическое описание алгоритмов поиска, сформулированны тесты и требования к программному обеспечению.

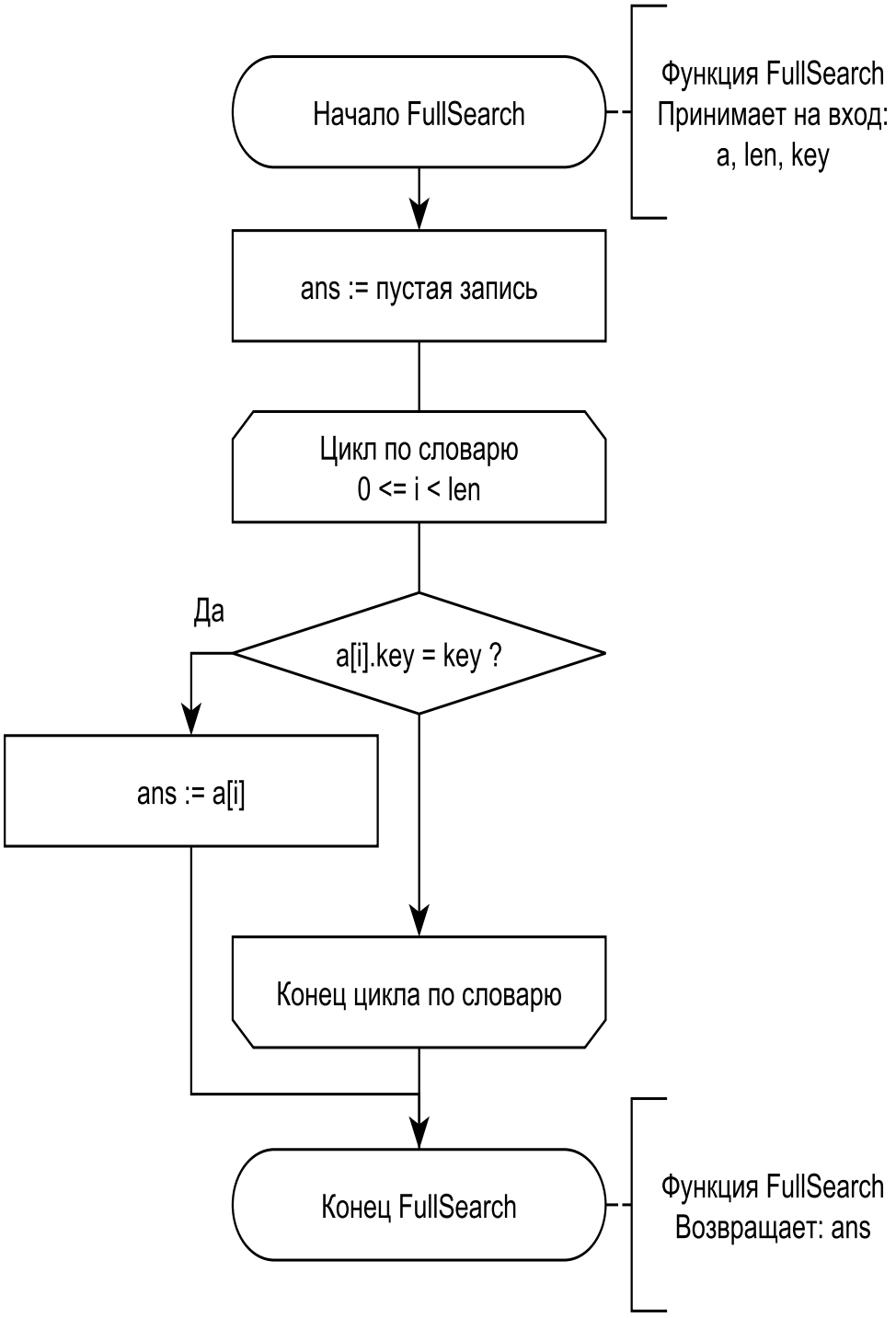

Рис. 2.1 — Поиск полным перебором

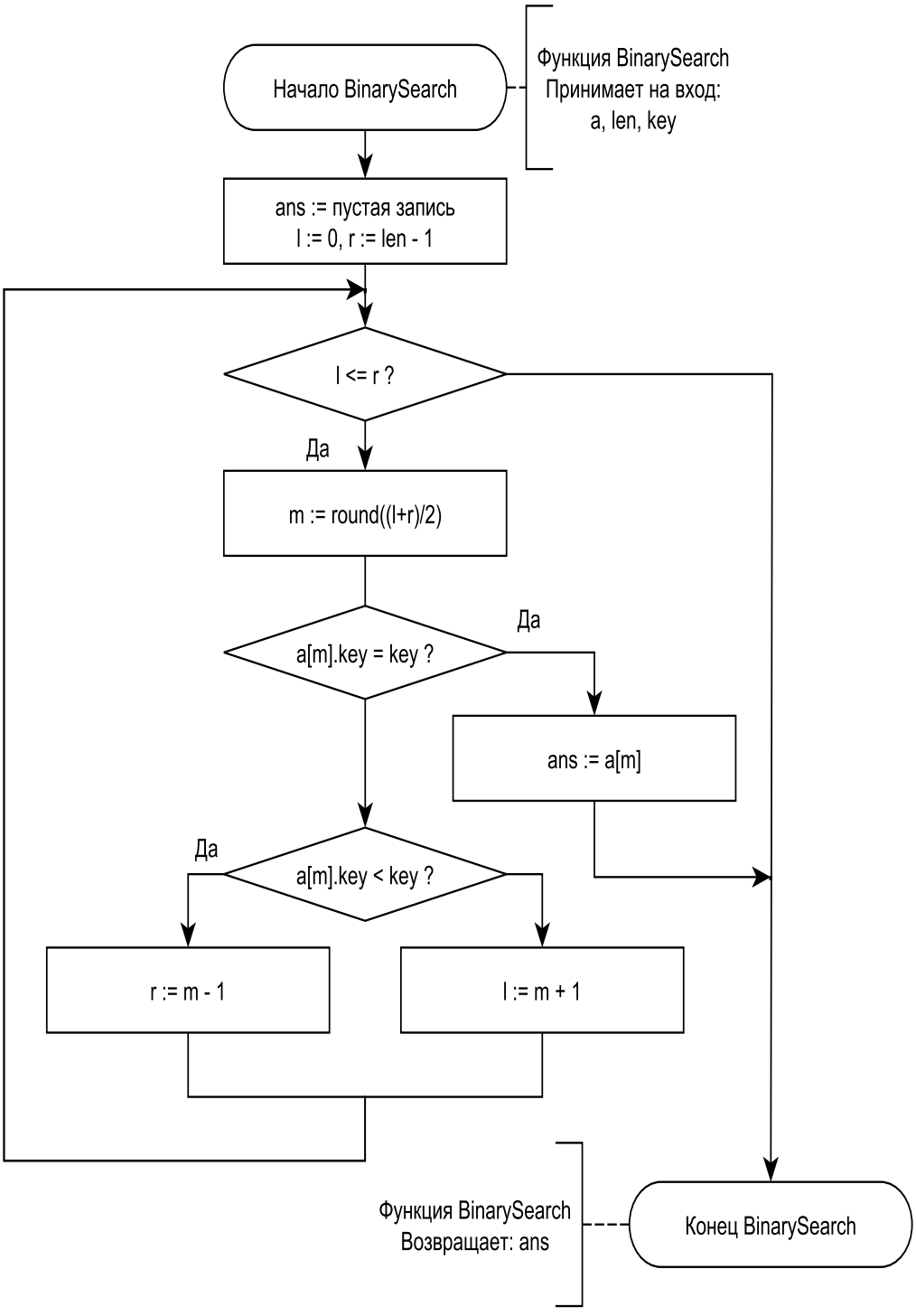

Рис. 2.2 — Поиск половинным разбиением

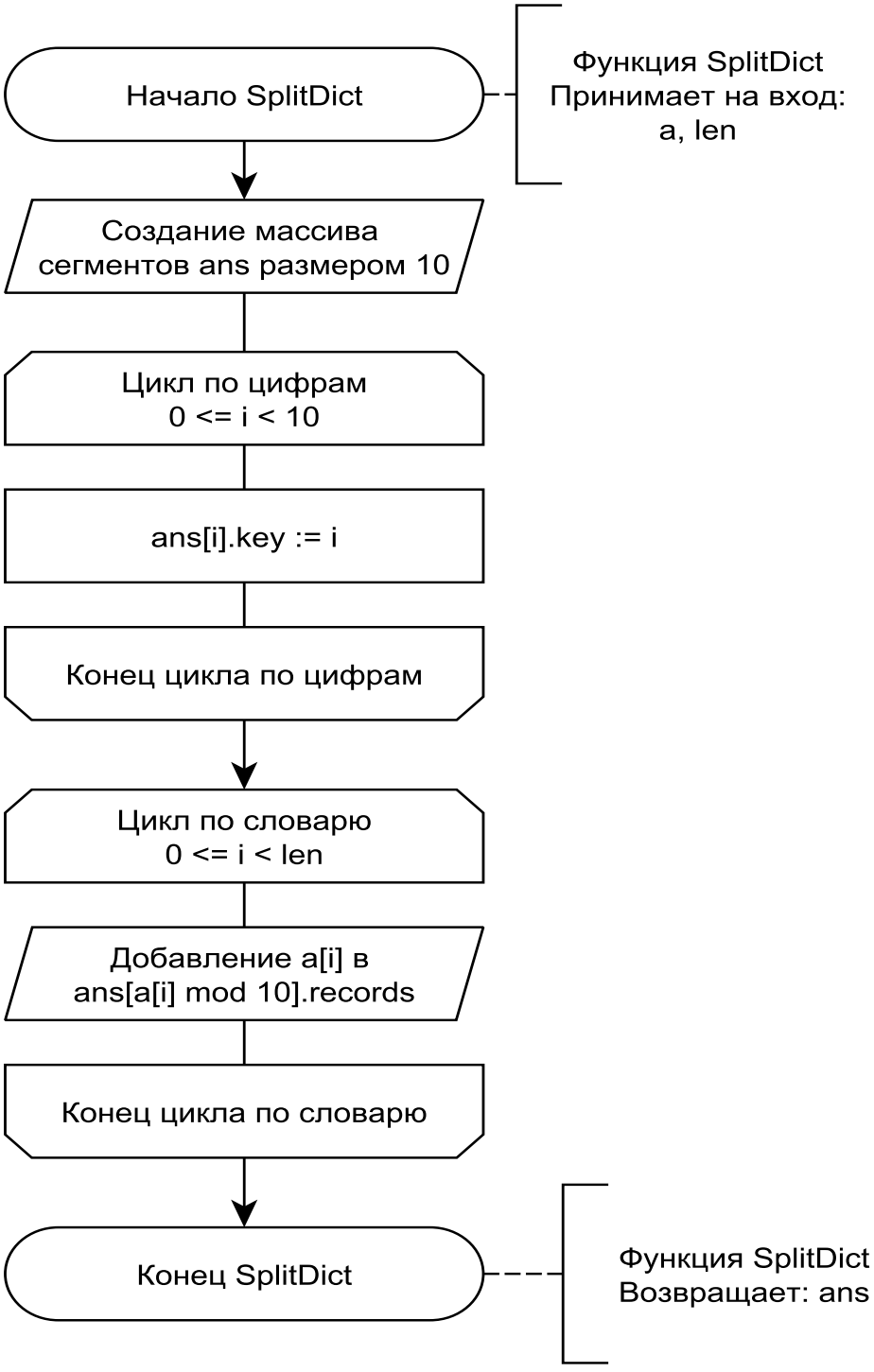

Рис. 2.3 — Разбиения словаря по сегментам

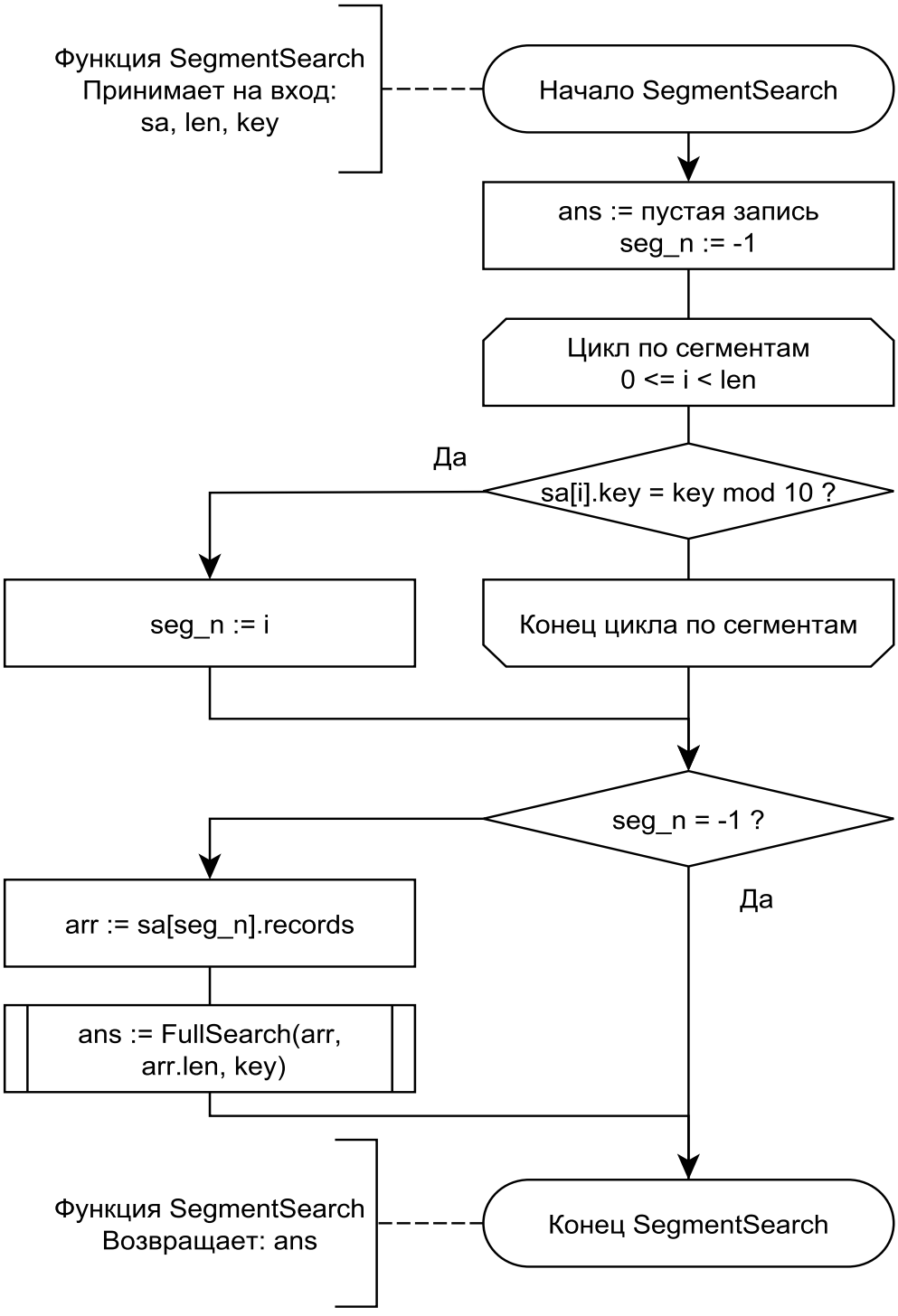

Рис. 2.4 — Поиск с сегментами

3. Технологическая часть

3.1. Выбор языка программирования

В качестве языка программирования был выбран C++[2], так как имеется опыт работы с ним, и с библиотеками, позволяющими провести исследование и тестирование программы. Разработка проводилась в среде Visual Studio 2019[3].

3.2. Листинги кода

Реализация алгоритмов поиска представлена на листингах 3.1-3.3. На листинге 3.4 представлена реализация разбиения словаря по сегментам

Листинг 3.1 — Поиск полным перебором

```
rec_t full_search(const rec_arr& arr, size_t key)
{
  for (int i = 0; i < arr.size(); i++)
    if (arr[i].key == key)
      return arr[i];
  return null_rec();
}
```

Листинг 3.2 — Поиск половинным разбиением

```
rec_t binary_search(const rec_arr& arr, size_t key)
{
  int left = 0;
  int right = arr.size() - 1;
  while (left <= right)
  {
    int mid = (left + right) / 2;
    if (key == arr[mid].key)
      return arr[mid];
    if (key < arr[mid].key)
      right = mid - 1;
```

```
12      else
13        left = mid + 1;
14    }
15    return null_rec();
16  }
```

Листинг 3.3 — Поиск с сегментами

```
1  rec_t segment_search(const seg_arr& segments, size_t key)
2  {
3    int seg_n = -1;
4    for (int i=0; i<segments.size(); i++)
5    if (segments[i].key == key % 10)
6    {
7      seg_n = i;
8        break;
9    }
10
11   if (seg_n == -1)
12     return null_rec();
13
14   const rec_arr& arr = segments[seg_n].records;
15   return full_search(arr, key);
16 }
```

Листинг 3.4 — Разбиение словаря по сегментам

```
1  seg_arr split_arr(rec_arr& arr)
2  {
3    seg_arr segments;
4    for (int i = 0; i < 10; i++)
5    {
6      rec_seg temp_seg;
7      temp_seg.key = i;
8
```

```
9       segments.push_back(temp_seg);
10    }
11
12    for (int i = 0; i < arr.size(); i++)
13      segments[arr[i].key % 10].records.push_back(arr[i]);
14    return segments;
15  }
```

3.3. Результаты тестирования

Для тестирования написанных функций был создан отдельный файл с ранее описанными классами тестов. Тестирование функций проводилось за счёт сравнения результов функций друг с другом.

Состав тестов приведён в листинге 3.5.

Листинг 3.5 — Модульные тесты

```
1 #include "tests.h"
2
3 using namespace std;
4
5 rec_t test1(rec_arr& arr, size_t key)
6 {
7   return full_search(arr, key);
8 }
9 rec_t test2(rec_arr& arr, size_t key)
10 {
11   sort_arr(arr);
12   return binary_search(arr, key);
13 }
14 rec_t test3(rec_arr& arr, size_t key)
15 {
16   seg_arr sarr = split_arr(arr);
```

```
17    return segment_search(sarr, key);
18  }
19
20
21  bool _cmp_rec(const rec_t& r1, const rec_t& r2)
22  {
23    return (r1.key == r2.key) && (r1.val == r2.val);
24  }
25
26  bool _test_all(rec_arr& arr, size_t key, rec_t res)
27  {
28    test_f test_f_arr[3] = { test1, test2, test3 };
29    rec_t test_out;
30
31    for (int i = 0; i < 3; i++)
32    {
33      test_out = test_f_arr[i](arr, key);
34      if (!_cmp_rec(test_out, res))
35      return false;
36    }
37    return true;
38  }
39
40  void _find_missing(rec_arr& arr)
41  {
42    if (_test_all(arr, 1012, null_rec()))
43    cout << __FUNCTION__ << " - OK\n";
44    else
45    cout << __FUNCTION__ << " - FAILED\n";
46  }
47  void _find_first(rec_arr& arr)
48  {
49    if (_test_all(arr, arr[0].key, arr[0]))
50    cout << __FUNCTION__ << " - OK\n";
```

```
51    else
52    cout << __FUNCTION__ << " - FAILED\n";
53  }
54  void _find_last(rec_arr& arr)
55  {
56    size_t last = arr.size() - 1;
57    if (_test_all(arr, arr[last].key, arr[last]))
58    cout << __FUNCTION__ << " - OK\n";
59    else
60    cout << __FUNCTION__ << " - FAILED\n";
61  }
62
63  void run_tests(rec_arr& arr)
64  {
65    cout << "Running tests:" << endl;
66    _find_missing(arr);
67    _find_first(arr);
68    _find_last(arr);
69    cout << endl;
70  }
```

3.4. Оценка времени

Для замера процессорного времени исполнения функции используется функция QueryPerformanceCounter библиотеки windows.h[4]. Код функций замера времени приведёны в листинге 3.6.

Листинг 3.6 — Функции замера процессорного времени работы функции

```
1  double PCFreq = 0.0;
2  __int64 CounterStart = 0;
3
4  void start_counter()
```

```
5  {
6    LARGE_INTEGER li;
7    QueryPerformanceFrequency(&li);
8
9    PCFreq = double(li.QuadPart) / 1000.0;
10
11   QueryPerformanceCounter(&li);
12   CounterStart = li.QuadPart;
13 }
14
15 double get_counter()
16 {
17   LARGE_INTEGER li;
18   QueryPerformanceCounter(&li);
19   return double(li.QuadPart - CounterStart) / PCFreq;
20 }
```

Вывод

Результатом технологической части стал выбор используемых технических средств реализации и реализация алгоритмов, системы тестов и замера времени работы на языке C++.

4. Исследовательская часть

4.1. Описание эксперимента

Измерения процессорного времени проводятся на словаре размером 1500. Содержание сгенерировано случайным образом. Вычисляются и демонстрируются минимальное, максимальное и среднее время поиска ключа, а также время поиска несуществующиего ключа.

Для повышения точности, каждый замер производится пять раз, за результат берётся среднее арифметическое.

4.2. Результат эксперимента

По результатам измерений процессорного времени можно составить таблицу 4.1

Таблица 4.1 — Результат измерений процессорного времени (в микросекундах)

Алгоритм Время	Минимальное	Максимальное	Среднее	♯
Полный перебор	0.021	1.11	0.38	1.42
Половинное деление	0.023	0.089	0.048	0.074
По сегментам	0.023	0.12	0.067	0.14

4.3. Характеристики ПК

Эксперименты проводились на компьютере с характеристиками:

- ОС - Windows 10, 64 бит;
- Процессор - Intel Core i7 8550U (1800 МГц, 4 ядра, 8 логических процессоров);
- Объем ОЗУ: 8 ГБ.

Вывод

По результатам экспериментов можно заключить следующее.

- Наименьшее минимальное время показывает функция полного перебора. Это объясняется тем, что в других алгоритмах трудоёмкость лучшего случая сделана больше из-за стремления уменьшить трудоёмкость худшего и среднего случая.
- Также полный перебор показывает и наибольшее максимальное время поиска на порядок превышающее значения у других алгоритмов.
- Наиболее быстродейственным как в худшем, так и в среднем случае оказался алгоритм половинного деления.
- Алгоритм поиска по сегментам продемонстрировал время в среднем и худшем случае медленне лишь на 50% по сравнению с методом половинного деления, что говорит о том, что этот способ оптимизации также является достаточно эффективным средством ускорения процедуры поиска.
- Во всех случаях время поиска несуществующего ключа примерно равно максимальному времени поиска.

Заключение

В ходе лабораторной работы достигнута поставленная цель: разработаны и исследованны алгоритма поиска ключей в словаре банковских карт. Решены все задачи работы.

Были изучены и описаны понятия словаря. Также были реализованы алгоритмы поиска ключей в словаре. Проведены замеры процессорного времени поиска ключей разными алгоритмами, собраны статистические данные. На основании экспериментов проведён сравнительный анализ.

Из проведённых экспериментов было выявлено,что наиболее быстродейственным является алгоритм половинного деления. Алгоритм выполняющий поиск по сегментам словаря показывает сравнимые результаты с предыдущим алгоритмом, что также говорит о его применимости для задачи поиска ключей. Алгоритм полного перебора является более быстродейственным только при лучшем случае. Уже при изученном размере словаря в 1500 записей он демонстрирует скорость на порядок хуже других алгоритмов, что ограничивает его использование при значительных размерах словарей. При этом во всех алгоритмах операция поиска несуществующего ключа фактически является худшим случаем по времени.

Список литературы

1. Алгоритмы. Построение и анализ : пер. с анг. / Кормен Т., Лейзерсон Ч., Ривест Р. [и др.]. - 3-е изд. - М. : Вильямс, 2018. - 1323 с. : ил.

2. Документация языка C++ 98 [Электронный ресурс]. Режим доступа: http://www.open-std.org/JTC1/SC22/WG21/, свободный (дата обращения: 14.11.2020)

3. Документация среды разработки Visual Studio 2019 [Электронный ресурс]. Режим доступа: https://docs.microsoft.com/ru-ru/visualstudio/windows/?view=vs-2019, свободный (дата обращения: 14.11.2020)

4. QueryPerformanceCounter function [Электронный ресурс]. Режим доступа: https://docs.microsoft.com/en-us/windows/win32/api/profileapi/nf-profileapi-queryperformancecounter, свободный (дата обращения: 29.10.2020).