



МИНОБР НАУКИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА — Российский технологический университет»

РТУ МИРЭА

Отчёт по выполнению практического задания 8
Тема: Алгоритмы поиска в таблице (массиве)
Дисциплина Структуры и алгоритмы обработки данных

Выполнил студент	Пак С.А.
группа	ИКБО-05-20

Москва 2021

Оглавление

ОТЧЁТ ПО ЗАДАНИЮ 1.....	3
1. Постановка задачи.....	3
2. Описание подхода к решению.....	3
3. Коды функций, реализующих алгоритмы.....	4
4. Таблица.....	5
5. Графики.....	5
6. Выводы по эффективности.....	6
ОТЧЁТ ПО ЗАДАНИЮ 2.....	7
1. Постановка задачи.....	7
2. Алгоритм.....	7
3. Код функции.....	8
4. Таблица.....	9
5. Графики.....	9
6. Выводы по эффективности.....	9
7. Сравнительный анализ алгоритмов.....	10

ОТЧЁТ ПО ЗАДАНИЮ 1

Вариант: 4

1. Постановка задачи

Разработать программу поиска записи по ключу в таблице записей с применением двух алгоритмов линейного поиска: простой линейный поиск; линейный поиск с барьером.

2. Описание подхода к решению

Для начала нужно определиться со структурой таблицы записей. Каждая строка таблицы будет представлена объектом класса *TableRow*, которая будет храниться в массиве.

Класс *TableRow* обладает следующими полями (свойствами): *_number* (строковый тип) — номер машины; *_brand* (строковый тип) — марка машины; *_owner* (строковый тип) — информация о владельце. В принципе это всё, что есть в данном классе.

Теперь необходимо описать алгоритмы поиска. Первым опишем алгоритм простого линейного поиска.

Алгоритм линейного поиска: в цикле нужно проверить каждый элемент массива, и если у текущего элемента свойство, которое является ключом, совпадает с заданным ключом, то нужно вернуть этот элемент. Если цикл закончился, а нужный элемент так и не был найден, то вернуть специальное значение *undefined*.

Алгоритм линейного поиска с барьером немного отличается от простого линейного поиска. По сравнению с простым линейным поиском в поиске с барьером на одну проверку меньше.

Алгоритм линейного поиска с барьером:

- 1) Запоминаем в отдельной переменной последний элемент массива;
- 2) Последнему ключу присваиваем значение заданного ключа;
- 3) Создаём переменную индекс с начальным значением 0;
- 4) В цикле пока текущий ключ не равен заданному ключу, увеличиваем значение индекса;
- 5) Восстанавливаем последний элемент;
- 6) Если текущее значение индекса меньше значения последнего индекса, или последний ключ был равен заданному, то возвращаем элемент под текущему индексу;
- 7) Иначе, вернуть специальное значение *undefined*.

В языке программирования *JavaScript* тип данных *string* занимает 2 байта. Соответственно, в *TypeScript*, который является надмножеством *JavaScript*, тип данных *string* тоже должен занимать 2 байта.

Структура записи таблицы содержит в себе 3 поля строковых типа. Следовательно, размер одной записи будет составлять $3 * 2 = 6$ байт.

3. Коды функций, реализующих алгоритмы

Для алгоритма простого линейного поиска была создана функция под названием *linearSearch(table, value)*, где *table* — таблица, в которой происходит поиск, *value* — значение, которое нужно найти. Предусловием: функция вернёт специальное значение *undefined* если таблица будет пустой. Постусловие: по завершении вызова функции массив *table* не изменится.

Для алгоритма линейного поиска с барьером была создана функция под названием *barrierSearch(table, value)*, где *table* — таблица, в которой происходит поиск, *value* — значение, которое нужно найти. Предусловием: функция вернёт специальное значение *undefined* если таблица будет пустой. Постусловие: по завершении вызова функции массив *table* не изменится.

Коды функций представлены ниже:

```
/**
 * Выполняет простой линейный поиск значения value в "таблице" table
 * @param table      "таблица", в которой происходит поиск
 * @param value      значение, которое нужно найти
 */
export const linearSearch = (table: Array<TableRow>, value: string): TableRow |
undefined => {
  if (table.length === 0) {    // если массив пустой, то искать нечего
    return undefined;
  }

  return table.find((val: TableRow) => val.number === value);
}

/**
 * Выполняет линейный поиск с барьером
 * @param table      "таблица", в которой происходит поиск
 * @param value      значение, которое нужно найти
 */
export const barrierSearch = (table: Array<TableRow>, value: string): TableRow |
undefined => {
  const length: number = table.length;

  if (length === 0) {    // если массив пустой, то искать нечего
    return undefined;
  }

  let last: TableRow = cloneDeep(table[length - 1]);    // запоминаем последний
элемент
  let index: number = 0;

  table[length - 1].number = value;    // гарантирует, что ключ найдётся
  for (; table[index].number !== value; ++index);
}
```

```

table[length - 1] = last;          // восстанавливаем последний элемент

if (index < length - 1 || last.number === value) {
    return table[index];
}

return undefined;
}

```

4. Таблица

Ниже представлена сводная таблица результатов для алгоритмов простого линейного поиска (табл.1) и линейного поиска с барьером (табл.2):

Таблица 1

n	T(n)	$T_r=f(C+M)$	$T_n=C_\phi+M_\phi$
100	0,66 с	100	92
1000	0,65 с	1000	486
10000	0,67 с	10000	4950
100000	0,68 с	100000	9276

Таблица 2

n	T(n)	$T_r=f(C+M)$	$T_n=C_\phi+M_\phi$
100	0,65 с	100	38
1000	0,65 с	1000	221
10000	0,66 с	10000	4551
100000	0,68 с	100000	11546

5. Графики

Ниже представлены графики (рис.1) для алгоритмов простого линейного поиска (синий цвет на графике) и линейного поиска с барьером (оранжевый цвет на графике).

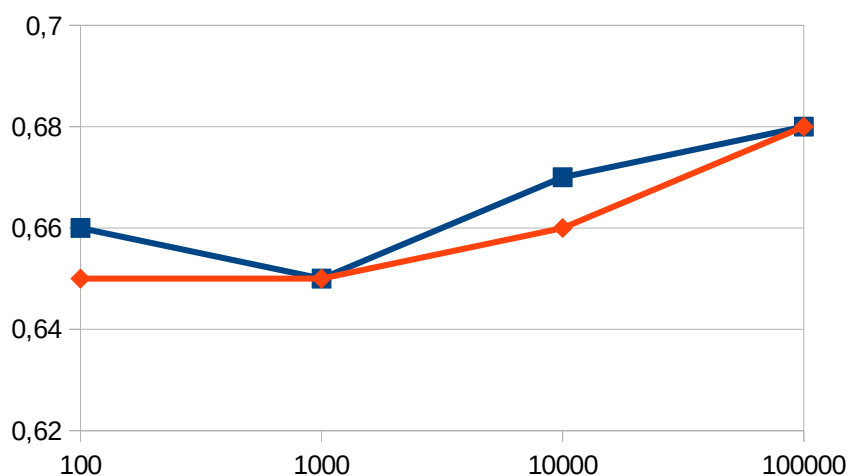


Рис.1 Графики алгоритмов линейного поиска

6. Выводы по эффективности

Исходя из таблицы 1 можно сделать вывод, что алгоритм линейного поиска с барьером лишь немного эффективнее алгоритма простого линейного поиска.

ОТЧЁТ ПО ЗАДАНИЮ 2

1. Постановка задачи

Разработать программу поиска записи по ключу в таблице записей с применением алгоритма поиска Фибоначчи.

2. Алгоритм

Для начала нужно определиться со структурой таблицы записей. Каждая строка таблицы будет представлена объектом класса *TableRow*, которая будет храниться в массиве.

Класс *TableRow* обладает следующими полями (свойствами): *_number* (строковый тип) — номер машины; *_brand* (строковый тип) — марка машины; *_owner* (строковый тип) — информация о владельце. В принципе это всё, что есть в данном классе.

Идея алгоритма Фибоначчи поиска состоит в том, чтобы сначала найти наименьшее число Фибоначчи, которое больше или равно длине данного массива. Пусть искомый элемент будет x , fib — найденное число Фибоначчи (m -е число Фибоначчи). Мы используем $(m-2)$ число Фибоначчи в качестве индекса (если это действительный индекс). Пусть $(m-2)$ число Фибоначчи будет i , мы сравним $array[i]$ с x , если они равны, то вернём i . Иначе, если x больше, мы возвращаемся для подмассива после i , иначе мы возвращаемся для подмассива до i .

Код алгоритма на псевдокоде (n — длина массива, $array$ — массив):

```
fib1 = 0;
fib2 = 1;
fib = fib1 + fib2;

while (fib < n) do
    fib2 = fib1;
    fib1 = fib;
    fib = fib1 + fib2;
od

offset = -1;

while (fib > 1) do
    i = min(offset + fib2, n - 1);
    if (array[i] < value) then
        fib = fib1;
        fib1 = fib2;
        fib2 = fib - fib1;
        offset = i;
    else if (array[i] > value) then
        fib = fib2;
        fib1 -= fib2;
        fib2 = fib - fib1;
    else
        return i;
```

```

    endif
od

if (fib1 == 1 and array[offset + 1] == value) then
    return offset + 1;
endif

return undefined;

```

3. Код функции

Функция называется *fibonacciSearch(array, value)*, где *array* — массив, в котором происходит поиск, *value* — значение, которое нужно найти. Предусловием: функция вернёт специальное значение *undefined* если таблица будет пустой. Постусловие: по завершении вызова функции массив *table* не изменится.

```

/**
 * Выполняет Фибоначчи поиск в заданном массиве array
 * @param array      массив, в котором происходит поиск
 * @param value      элемент, который нужно найти
 */
export const fibonacciSearch = (array: Array<number>, value: number): number |
undefined => {
    const length: number = array.length;

    if (length === 0)
        return undefined;

    let fib1: number = 0;
    let fib2: number = 1;
    let fib: number = fib1 + fib2;

    while (fib < length) {
        fib2 = fib1;
        fib1 = fib;
        fib = fib1 + fib2;
    }

    let offset: number = -1;

    while (fib > 1) {
        let i: number = Math.min(offset + fib2, length - 1);

        if (array[i] < value) {
            fib = fib1;
            fib1 = fib2;
            fib2 = fib - fib1;

            offset = i;
        }
        else if (array[i] > value) {
            fib = fib2;
            fib1 -= fib2;
            fib2 = fib - fib1;
        }
    }
}

```



```

    }
    else {
        return i;
    }
}

if (fib1 === 1 && array[offset + 1] === value) {
    return offset + 1;
}

return undefined;
}

```

4. Таблица

Ниже представлена сводная таблица результатов для алгоритма Фибоначчи поиска (табл.3):

Таблица 3

n	T(n)	$T_r=f(C+M)$	$T_n=C_\phi+M_\phi$
100	0,66 с	6,644	17
1000	0,82 с	9,966	279
10000	15,97 с	13,288	4237
100000	20,05 с	16,61	9276

5. Графики

Ниже представлен график (рис.2) для алгоритма Фибоначчи поиска (синий цвет на графике).

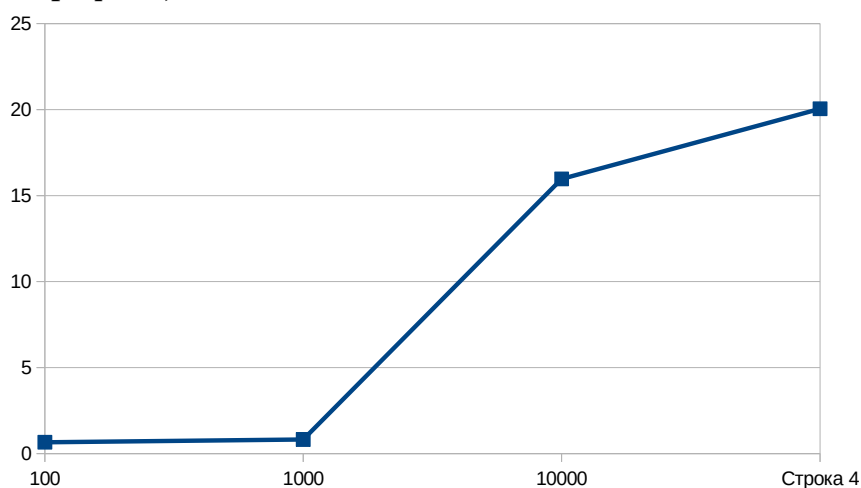


Рис.2 График для алгоритма Фибоначчи поиска

6. Выводы по эффективности

На первый взгляд алгоритм Фибоначчи поиска кажется крайне неэффективным. Однако нужно учитывать, что большая часть времени тратится на сортировку массива. Поэтому фактическое время поиска становится гораздо меньше.

Теоретическая сложность данного алгоритма составляет $O(\log n)$, что делает данный алгоритм самым быстрым среди всех рассматриваемых в данном задании.

7. Сравнительный анализ алгоритмов

В предыдущем задании было утверждено, что алгоритм линейного поиска с барьером быстрее работает, чем алгоритм простого линейного поиска. Теоретическая сложность обоих алгоритмов составляет $O(n)$.

Быстрее всех работает алгоритм поиска Фибоначчи, теоретическая сложность которого составляет $O(\log n)$. Но такое возможно только в том случае, если массив отсортирован по возрастанию.

С другой стороны первые два алгоритма менее зависимы от каких-либо условий, что, безусловно, усложняет анализ.

Однако, если не учитывать время сортировки массива в алгоритме Фибоначчи поиска, то без сомнений Фибоначчи поиск является самым быстрым алгоритмом.

В этом можно убедиться, если посмотреть на графики зависимости теоретической сложности алгоритма от объёма входных данных (рис.3). Синим цветом обозначена теоретическая сложность алгоритмов линейного поиска, оранжевым — алгоритма поиска Фибоначчи.

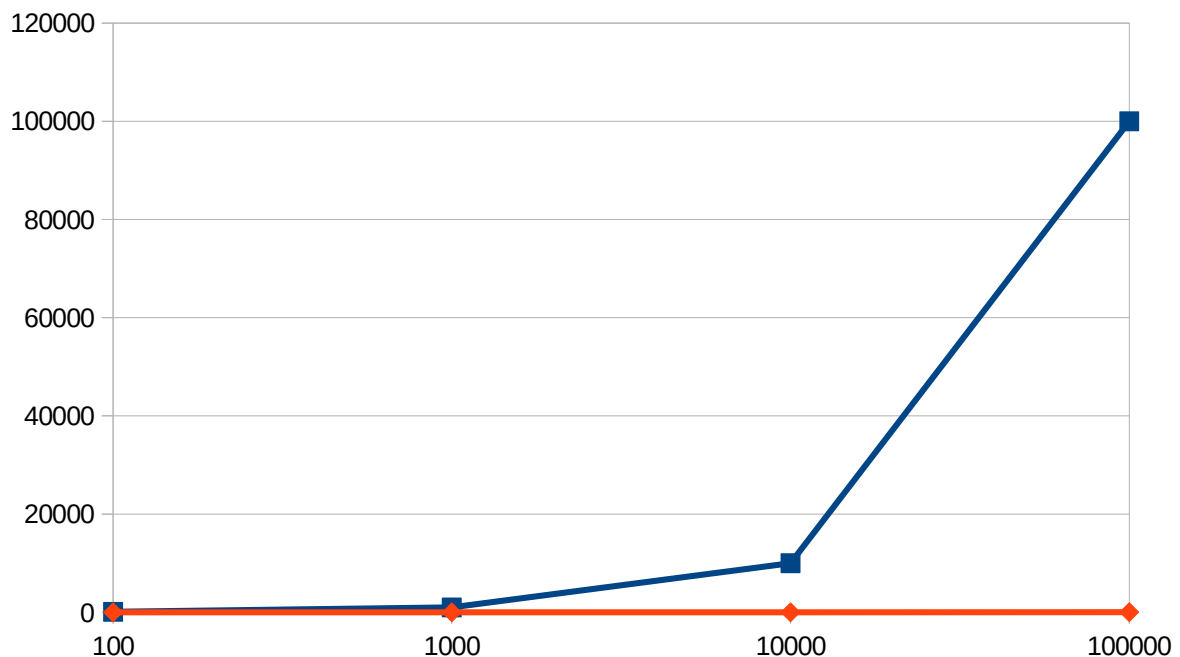


Рис.3 График зависимости теоретической сложности от объёма входных данных

Разница в сложностях, конечно же, очевидна.