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

СОДЕРЖАНИЕ

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)	$T_т=f(C+M)$	$T_п=C_ф+M_ф$
100	0,186 мс	100	92
1000	0,225 мс	1000	486
10000	0,556 мс	10000	4950
100000	1,963 мс	100000	9276

Таблица 2

n	T(n)	$T_т=f(C+M)$	$T_п=C_ф+M_ф$
100	0,191 мс	100	38
1000	0,206 мс	1000	221
10000	0,357 мс	10000	4551
100000	2,323 мс	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).

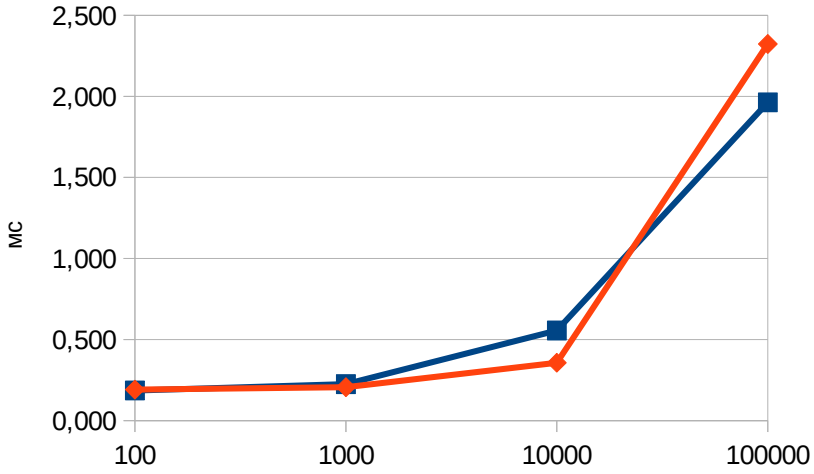

Рис.1 Графики алгоритмов линейного поиска

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)	$T_т$=f(C+M)	$T_п$=$C_ф$+$M_ф$
100	0,224 мс	6,644	17
1000	0,248 мс	9,966	279
10000	0,259 мс	13,288	4237
100000	0,267 мс	16,61	9276

Прежде чем предоставить результаты для алгоритма поиска Фибоначчи в худшем и лучшем случае, нужно понять какие случаи являются худшими и лучшими.

Очевидно, худшим случаем является отсутствие элемента в массиве. Тогда алгоритм должен пройтись по всем элементам массива.

Лучшим же случаем является ситуация, когда первый проверяемый элемент массива оказывается тем, который нужно было найти.

Ниже представлена таблица результатов для алгоритма поиска Фибоначчи в худшем случае (табл.4).

Таблица 4

n	T(n)	$T_т$=f(C+M)	$T_п$=$C_ф$+$M_ф$
100	0,226 мс	6,644	10
1000	0,245 мс	9,966	15
10000	0,263 мс	13,288	19
100000	0,276 мс	16,61	24

Ниже представлена таблица результатов для алгоритма поиска Фибоначчи в лучшем случае (табл.5).

Таблица 5

n	T(n)	$T_т$=f(C+M)	$T_п$=$C_ф$+$M_ф$
100	0,228 мс	6,644	1

Продолжение таблицы 5

n	T(n)	$T_т=f(C+M)$	$T_п=C_ф+M_ф$
1000	0,241 мс	9,966	1
10000	0,257 мс	13,288	1
100000	0,267 мс	16,61	1

5. Графики

Ниже представлены графики (рис.2) зависимости практической сложности алгоритма от количества элементов в массиве для массива, заполненного случайно (синий цвет), худшего случая (оранжевый цвет), лучшего случая (жёлтый цвет).

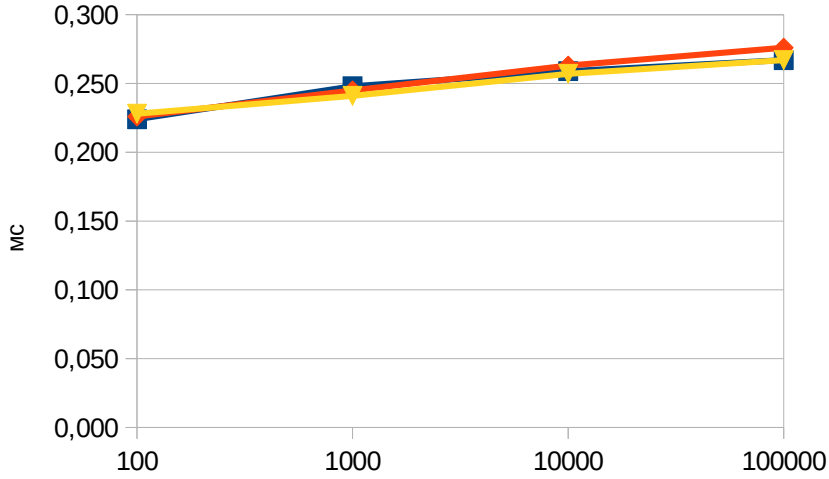

Рис.2 График для алгоритма Фибоначчи поиска

6. Выводы по эффективности

Из результатов, представленных в таблице 3, можно сделать вывод: рост времени поиска Фибоначчи небольшой. Поэтому на больших объёмах данных данный алгоритм справляется лучше простого линейного поиска и линейного поиска с барьером.

Следовательно, поиск Фибоначчи можно назвать самым эффективным алгоритмом среди рассматриваемых в задании.

7. Сравнительный анализ алгоритмов

В задании рассматриваются три алгоритма поиска:

1) Простой линейный поиск;
2) Линейный поиск с барьером;
3) Поиск Фибоначчи.

Из таблиц 1, 2, 3 можно заметить, что при *n* от 100 до 1000 первые два алгоритма работают быстрее, чем поиск Фибоначчи. Однако при *n > 1000* поиск

Фибоначчи работает быстрее, причём рост его времени выполнения меньше, чем рост времени выполнения алгоритмов линейного поиска. Поэтому при больших объёмах данных время выполнения будет сильно различаться, причём в пользу поиска Фибоначчи. Следовательно, поиск Фибоначчи эффективнее алгоритмов линейного поиска.

С другой стороны поиск Фибоначчи возможен, только в отсортированной последовательности. Поэтому предварительно нужно ещё отсортировать входные данные, что занимает немало времени. Из-за этого общее время выполнения поиска Фибоначчи складывается из времени сортировки входных данных и времени самого поиска.

В итоге поиск Фибоначчи хоть и эффективный, но также зависит от входных данных. В то время как алгоритмы линейного поиска не имеют такого недостатка и могут работать с неотсортированной последовательностью.

Если отбросить все ограничения, то, безусловно, алгоритм поиска Фибоначчи является самым эффективным, так как при больших объёмах данных он работает гораздо быстрее, чем алгоритмы линейного поиска.