Лабораторная работа №1
Оценка сложности алгоритмов поиска

1 Цель работы

1.1 Научиться реализовывать и оценивать сложность алгоритмов поиска элементов массива.

2 Литература

2.1 Фленов, М. Е. Библия C#. – 3 изд. – Санкт-Петербург: БХВ-Петербург, 2016. – URL: https://ibooks.ru/bookshelf/353561/reading. – Режим доступа: для зарегистрир. пользователей. – Текст : электронный.

3 Подготовка к работе

3.1 Повторить теоретический материал (см. п.2).

3.2 Изучить описание лабораторной работы.

3.3 Создать в папке C:\temp папку с названием группы isppNN (NN – номер группы) для хранения создаваемых приложений.

4 Основное оборудование

4.1 Персональный компьютер.

5 Задание

5.1 Реализовать и проверить алгоритм линейного поиска элемента в массиве. Если элемент не найден, возвращать -1, иначе – индекс найденного элемента.

Для разработанного алгоритма:

- составить набор тестов, используя Debug.Assert,
- составить схему алгоритма по ГОСТ 19.701-90.

Линейный поиск последовательно проверяет каждый элемент списка на целевое значение до тех пор, пока не будет найдено совпадение или пока не будет выполнен поиск по всем элементам.

Временная сложность: O(n), так как в худшем случае проверяется каждый элемент ровно один раз.

5.2 Реализовать и проверить алгоритм двоичного поиска элемента в отсортированном массиве. Если элемент не найден, возвращать -1, иначе – индекс найденного элемента. Схема алгоритма – на рисунке 1.

Для разработанного алгоритма:

- составить набор тестов, используя Debug.Assert,
- составить схему алгоритма по ГОСТ 19.701-90.

Бинарный поиск сравнивает целевое значение со средним элементом массива, если они неравны, то устраняется половина, в которой целевое значение не может лежать, и поиск продолжается на оставшейся половине до тех пор, пока не будет найден целевой элемент.

Временная сложность: O(log(n)), так как область поиска делится пополам для каждой следующей итерации.

5.3 Реализовать и проверить алгоритм поиска прыжками элемента в отсортированном массиве. Если элемент не найден, возвращать -1, иначе – индекс найденного элемента.

Для разработанного алгоритма:

- составить набор тестов, используя Debug.Assert

Поиск прыжками – алгоритм поиска в отсортированном массиве. Основная идея состоит в том, чтобы проверить меньшее количество элементов (чем линейный поиск), перепрыгивая вперед на фиксированные шаги или пропуская некоторые элементы вместо поиска всех элементов. Лучший размер шага m = √n. После нахождения требуемого отрезка поиск в нем выполняется линейно.

Временная сложность: O(√n), так как поиск по блокам размером √n.

6 Порядок выполнения работы

6.1 Запустить MS Visual Studio и создать консольное приложение C# (Console Application) с названием LabWork1.

6.2 Выполнить все задания из п.5. Каждое задание должно быть в своем методе с соответствующим названием: ТипПоискаSearch(int[] array, int target)

При выполнении заданий использовать минимально возможное количество команд и переменных и выполнять форматирование и рефакторинг кода.

6.3 Ответить на контрольные вопросы.

7 Содержание отчета

7.1 Титульный лист

7.2 Цель работы

7.3 Ответы на контрольные вопросы

7.4 Вывод

8 Контрольные вопросы

8.1 Что такое «алгоритм сортировки»?

8.2 Какие виды поиска элементов массивов существуют?

8.3 В чем особенность алгоритма линейного поиска и какова его временная сложность?

8.4 В чем особенность алгоритма двоичного поиска и какова его временная сложность?

8.5 В чем особенность алгоритма поиска прыжками и какова его временная сложность?

9 Приложение

9.1 Массивы в C#

Стандартный способ сортировки (вызов у класса Array метода Sort)

```
int[] numbers = [97, 45, 32, 65, 83, 23, 15];
Array.Sort(numbers);
```

9.2 Поиск

Поиск – это задача нахождения индекса, по которому в массиве располагается некоторый заданный элемент.

Алгоритм поиска – это алгоритм нахождения индекса заданного значения в массиве.

Примеры алгоритмов поиска:

- **Линейный поиск** или последовательный поиск — это метод нахождения целевого значения в массиве. Он последовательно проверяет каждый элемент списка на целевое значение до тех пор, пока не будет найдено совпадение или пока не будет выполнен поиск по всем элементам. Линейный поиск выполняется в худшем линейном времени и делает не более n сравнений, где n — длина массива.
- **Двоичный поиск** – заключается в том, что на каждом шаге множество объектов делится на две части и в работе остаётся та часть множества, где находится искомый объект. Или же, в зависимости от постановки задачи, можно остановить процесс, когда получен первый или же последний индекс вхождения элемента. Последнее условие – это левосторонний/правосторонний двоичный поиск. Такой поиск требует отсортированной коллекции.

Алгоритм делит входную коллекцию на равные половины и с каждой итерацией сравнивает целевой элемент с элементом в середине (middle). Поиск заканчивается при нахождении элемента. Иначе продолжается поиск элемента, разделяя и выбирая соответствующий раздел массива. Целевой элемент сравнивается со средним. Поиск заканчивается, когда firstIndex или left (указатель) достигает lastIndex или right (последнего элемента). Значит, проверен весь массив и не найден элемент.

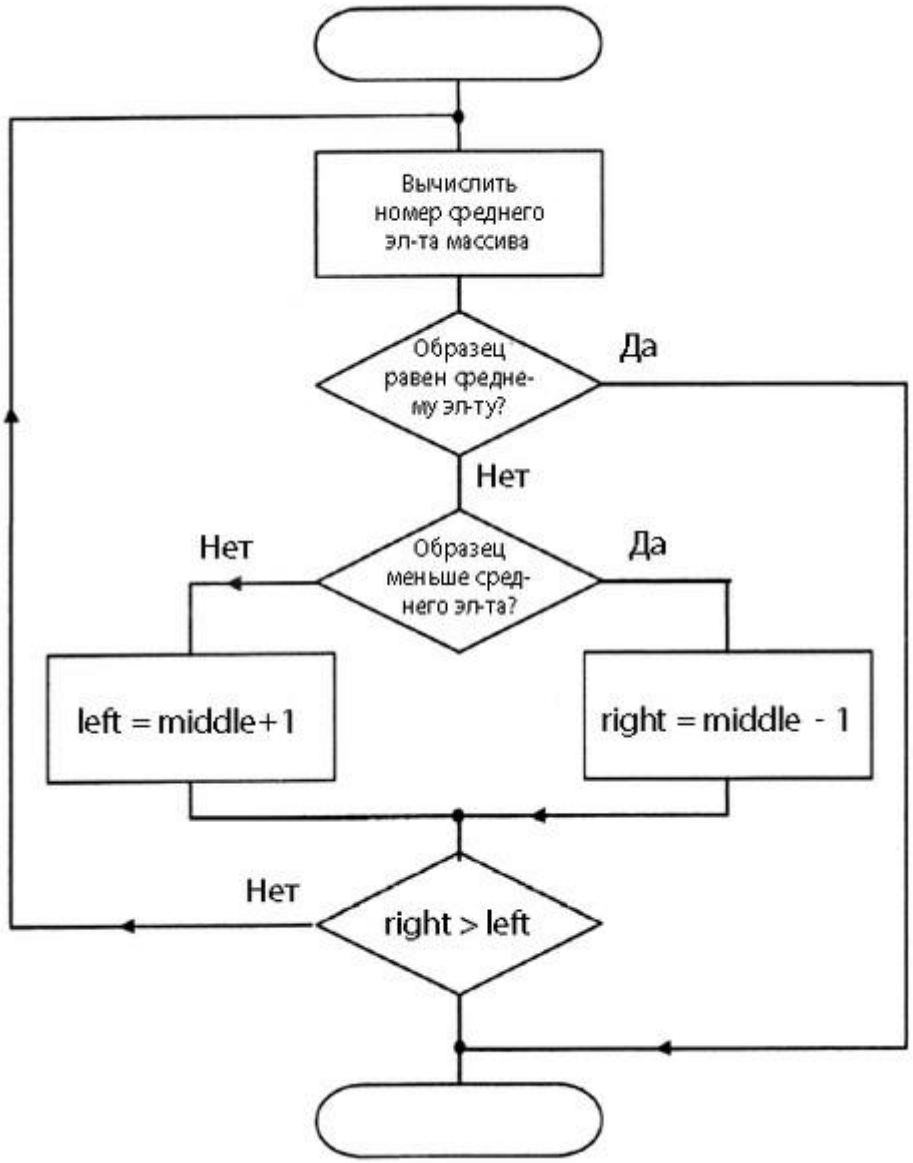

Рисунок 1 – Схема алгоритма двоичного поиска

- **Алгоритм поиска прыжками** – похож на двоичный поиск, но движение только вперёд. Такой поиск требует отсортированной коллекции.

Основная идея состоит в том, чтобы проверить меньшее количество элементов (чем линейный поиск), перепрыгивая вперед на фиксированные шаги или пропуская некоторые элементы вместо поиска всех элементов.

Например, есть массив arr[] размера n и блок размера m для прыжка. Затем поиск по индексам arr, arr[m], arr[2 * m], ..., arr[k * m] и так далее. После того, как найден интервал arr[k * m] < x < arr[(k+1) * m], выполняется операция линейного поиска по индексу k * m, чтобы найти элемент x.

Какой оптимальный размер блока для пропуска? В худшем случае приходится делать n/m прыжков и если последнее проверяемое значение больше искомого элемента, то для линейного поиска мы выполняем на m - 1 сравнения больше. Следовательно, общее число сравнений в худшем случае будет ((n/m) + m - 1). Значение функции ((n/m) + m - 1) будет минимальным при m = √n. Поэтому лучший размер шага m = √n.