Министерство науки и высшего образования РФ ФГБОУ ВО “Удмуртский государственный университет” институт математики, информационных технологий и физики кафедра теоретических основ информатики

Отчет по теме

«Анализ сложности алгоритмов поиска»

Выполнил: студент
группы ОБ-02.03.02.01-41
Полянских Сергей Владимирович

Ижевск, 2023

Задание

1. При помощи датчика случайных чисел сгенерировать массив данных (по данным практической работы №2) размером 30, 200 элементов.
2. Выполнить поиск определенного заданного значения в неупорядоченном массиве для двух наборов данных (30 и 200) методом полного перебора.
Вывести на экран индекс найденного элемента.
3. Выполнить сортировку массива данных (для двух наборов данных 30 и 200 элементов) по возрастанию применяя метод из практической работы №2.
4. Выполнить поиск определенного заданного значения в упорядоченном массиве для двух наборов данных (30 и 200) двумя методами: методом прямого перебора и выбранным методом.
Вывести на экран индекс найденного элемента и количество итераций по выбранному методу поиска.
5. Проанализировать полученные результаты.

1 Описание алгоритма бинарного поиска

Алгоритм двоичного поиска является одним из основных алгоритмов поиска в отсортированных массивах данных. Он основан на идее разделения области поиска пополам и последующем сужении этой области до нахождения искомого элемента.

Основной принцип работы алгоритма двоичного поиска заключается в следующем: в отсортированном массиве данных ищется элемент, который нужно найти. Сначала определяется начальная и конечная позиции в массиве. Затем определяется средний элемент между этими позициями. Если искомый элемент равен среднему элементу, то поиск завершается. Если искомый элемент меньше среднего элемента, то область поиска сужается до левой половины массива. Если искомый элемент больше среднего элемента, то область поиска сужается до правой половины массива. Этот процесс повторяется, пока не будет найден искомый элемент или пока область поиска не станет пустой.

Временная сложность алгоритма двоичного поиска составляет O(log n), где n - количество элементов в массиве. Это свойство делает алгоритм двоичного поиска очень эффективным для больших отсортированных массивов данных, поскольку время выполнения алгоритма увеличивается медленно с ростом размера массива.

Рассмотрим пример использования алгоритма двоичного поиска. Допустим, у нас есть отсортированный массив целых чисел *[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]*. Мы хотим найти элемент 11 в этом массиве. Используя алгоритм двоичного поиска, мы начинаем с определения начальной позиции (индекс 0) и конечной позиции (индекс 9) в массиве. Затем вычисляем средний элемент, который равен 9. Так как искомый элемент

(11) больше среднего элемента (9), мы сужаем область поиска до правой половины массива *[11, 13, 15, 17, 19]*. Далее снова вычисляем средний элемент, который равен 15. Так как искомый элемент (11) меньше среднего элемента (15), мы сужаем область поиска до левой половины массива [11, 13]. В результате получаем искомый элемент (11) на индексе 0 в суженной области.

2 Реализация алгоритма

Учитывая всё вышесказанное, реализуем алгоритм бинарного поиска на языке F#.

Листинг 1. Бинарный поиск

```
let mutable iteration = 0
let binarySearch (value: int) (array: array<int>) =
    iteration <- 0
    let rec loop lo hi =
        if lo <= hi then
            iteration <- iteration + 1
            let mid = lo + ((hi - lo) >>> 1)
            match array[mid] with
            | x when x = value -> Some mid, iteration
            | x when x < value -> loop (mid + 1) hi
            | _ -> loop lo (mid - 1)
        else None, iteration
    loop 0 (array.Length - 1)
```

Подробно рассмотрим, что делает каждая строка кода:

1. Создается изменчивая переменная *iteration*, которая будет считать количество итераций, сделанных функцией *binarySearch*.
2. В функции *binarySearch* используемые параметры - это *value* (значение, которое мы ищем) и *array* (массив, в котором мы ищем).
3. В начале функции счетчик *iteration* обнуляется.
4. Функция loop является внутренней, рекурсивной функцией, которая выполняет большую часть работы. Она получает два значения, *lo* и hi, которые представляют нижнюю и верхнюю границы диапазона поиска в массиве.
5. Если *lo* меньше или равно *hi*, то есть еще где искать, и мы продолжаем выполнение алгоритма.
6. На каждой итерации поиска счетчик *iteration* увеличивается на 1.
7. Следующий шаг - это вычисление средней точки диапазона поиска (mid).
8. Затем мы проверяем значение элемента *mid* в массиве. Если это искомое значение, мы возвращаем его индекс и количество итераций. Если значение mid меньше искомого, мы продолжаем поиск в правой половине массива. Если больше - ищем в левой половине.
9. Если lo становится больше, чем *hi*, поиск завершается, возвращается *None* (что означает, что искомое значение не найдено) и количество выполненных итераций.
10. На последнем этапе функция *loop* вызывается с начальными значениями *lo* и *hi*, равными 0 и длине массива минус 1 соответственно.

Таким образом, алгоритм определяет, есть ли заданное значение в отсортированном массиве, и возвращает его индекс и количество выполненных итераций.

Здесь же приведем работу алгоритма поиска полным перебором.

Листинг 2. Полный перебор

```
let simpleSearch (value:int) (array: 'T[]) =
    iteration <- 0
    let rec loop i =
        iteration <- iteration + 1
        if i >= array.Length then
            None, iteration-1
        else if value = array[i] then
            Some i, iteration-1
        else
            loop (i + 1)
    loop 0
```

3 Результаты работы

Проведём запуск программы, которая будет выполнять поставленной задачи.

Листинг 3. Сравнение работы алгоритмов поиска

```
let printResult (value:option<int>*int) =
    match value with
    | Some index, iteration -> printfn $"Value was found at {index} place, iterations: {iteration}"
    | None, iteration -> printfn $"Value was not found, iterations: {iteration}"

let arr30 = generateList 10 |> List.toArray
let arr200 = generateList 200 |> List.toArray
```

```
printfn "Simple search, size: 30"
printResult <| simpleSearch 1 arr30
printfn "Simple search, size: 200"
printResult <| simpleSearch 1 arr200

let sortedArr30 = quicksort (arr30 |> Array.toList) |> List.toArray
let sortedArr200 = quicksort (arr200 |> Array.toList) |> List.toArray

printfn "Simple search, size: 30"
printResult <| simpleSearch 1 sortedArr30
printfn "Binary search, size: 30"
printResult <| binarySearch 1 sortedArr30
printfn "Simple search, size: 200"
printResult <| simpleSearch 1 sortedArr200
printfn "Binary search, size: 200"
printResult <| binarySearch 1 sortedArr200
```

В качестве результата выполнения получаем интересующие значения:

```
Simple search, size: 30
Value was found at 7 place, iterations: 7
Simple search, size: 200
Value was found at 72 place, iterations: 72
Simple search, size: 30
Value was found at 4 place, iterations: 4
Binary search, size: 30
Value was found at 4 place, iterations: 0
Simple search, size: 200
Value was found at 105 place, iterations: 105
Binary search, size: 200
Value was found at 105 place, iterations: 4
```

Рис 1. Результаты работы

Вывод

В данном отчете было проведено сравнение двух важных алгоритмов поиска: полного перебора и бинарного поиска. Оба алгоритма представляют собой методы поиска элемента в наборе данных, но они имеют существенные различия в эффективности и времени выполнения.

Алгоритм полного перебора является наивным методом поиска, при котором каждый элемент в наборе данных последовательно сравнивается с целевым значением. Этот метод прост в реализации, но его основной недостаток заключается в его медленной скорости выполнения, особенно при больших объемах данных. Полный перебор имеет временную сложность O(n), где n - количество элементов в наборе данных.

В отличие от полного перебора, бинарный поиск работает на отсортированных данных и использует деление пополам для уменьшения области поиска с каждой итерацией. Этот метод обладает значительно большей эффективностью по сравнению с полным перебором, особенно при больших объемах данных. Бинарный поиск имеет временную сложность O(log n), где n - количество элементов в наборе данных. Однако, перед использованием бинарного поиска, необходимо отсортировать данные, что может потребовать времени O(n log n) для больших наборов данных.

Сравнив эти два алгоритма, можно сделать следующие выводы:

Полный перебор - прост в реализации, но неэффективен для больших наборов данных из-за своей линейной временной сложности.

Бинарный поиск - гораздо более эффективен, особенно на больших объемах данных, но требует предварительной сортировки данных, что может занять некоторое время для больших наборов данных.

Таким образом, выбор между этими двумя алгоритмами зависит от конкретной задачи и особенностей данных. Если данные уже отсортированы и требуется многократный поиск, бинарный поиск является

оптимальным выбором. В противном случае, если данных немного или они не отсортированы, полный перебор может быть более подходящим вариантом из-за своей простоты и отсутствия необходимости в предварительной сортировке.