## Последовательный поиск

Последовательный поиск (sequential search) - это метод поиска элемента в массиве, при котором каждый элемент массива последовательно сравнивается с искомым элементом до тех пор, пока не будет найден или пока не будут пройдены все элементы массива. Этот метод прост в реализации, но его скорость поиска зависит от количества элементов в массиве и может быть довольно медленным при большом размере массива.

public static int SequentialSearch(int[] arr, int target)

{

for (int i = 0; i < arr.Length; i++)

{

if (arr[i] == target)

{

return i; // элемент найден, возвращаем его индекс

}

}

return -1; // элемент не найден, возвращаем -1

}

Допустим, у нас есть массив целых чисел, и мы хотим найти индекс элемента, равного заданному числу. Мы можем реализовать поиск с помощью последовательного перебора элементов массива и сравнения их со значением, которое мы ищем.

Пример кода на языке C#:

int[] arr = { 1, 3, 5, 7, 9 };

int target = 7;

int index = -1; // индекс, где мы найдем искомый элемент

for (int i = 0; i < arr.Length; i++)

{

if (arr[i] == target)

{

index = i;

break;

}

}

if (index == -1)

{

Console.WriteLine("Элемент не найден");

}

else

{

Console.WriteLine("Искомый элемент находится в массиве под индексом: " + index);

}

В данном примере мы ищем число 7 в массиве arr. Последовательно перебирая все элементы массива, мы сравниваем каждый элемент с target. Если мы находим искомый элемент, мы сохраняем его индекс в переменную index и выходим из цикла с помощью оператора break. Если после перебора всего массива мы не находим искомый элемент, то index остается равным -1, и мы выводим сообщение "Элемент не найден". Если же index отличается от -1, то мы выводим сообщение о том, что искомый элемент находится в массиве под указанным индексом.

Сложность последовательного поиска в худшем случае составляет O(n), где n - количество элементов в массиве. Это означает, что если элемент, который мы ищем, находится в конце массива или отсутствует в массиве, нам придется проверить каждый элемент, что может занять много времени. Однако, если искомый элемент находится в начале массива, то поиск может быть выполнен быстро.

## Бинарный поиск (binary search)

Бинарный поиск (binary search) - это метод поиска элемента в упорядоченном массиве, при котором массив последовательно делится пополам, и искомый элемент сравнивается с элементом в середине каждой части массива. Если элемент совпадает с искомым, поиск завершается. Если элемент меньше искомого, поиск продолжается в правой половине массива, иначе - в левой. Этот метод более быстрый, чем последовательный поиск, но его можно использовать только с упорядоченными массивами.

public static int BinarySearch(int[] arr, int target)

{

int left = 0;

int right = arr.Length - 1;

while (left <= right)

{

int mid = (left + right) / 2;

if (arr[mid] == target)

{

return mid; // элемент найден, возвращаем его индекс

}

else if (arr[mid] < target)

{

left = mid + 1; // искомый элемент находится в правой половине массива

}

else

{

right = mid - 1; // искомый элемент находится в левой половине массива

}

}

return -1; // элемент не найден, возвращаем -1

}

Здесь мы передаем методу BinarySearch упорядоченный массив arr и искомый элемент target. Затем мы устанавливаем левую и правую границы поиска: left равен 0 (индекс первого элемента), а right равен индексу последнего элемента (arr.Length - 1). Мы повторяем цикл while до тех пор, пока left не станет больше right. На каждой итерации мы вычисляем средний индекс mid, и сравниваем его значение с искомым элементом. Если они совпадают, мы возвращаем индекс mid. Если значение arr[mid] меньше искомого элемента, мы продолжаем поиск в правой половине массива, устанавливая left = mid + 1. В противном случае мы продолжаем поиск в левой половине массива, устанавливая right = mid - 1.

Сложность алгоритма бинарного поиска - O(log n), где n - количество элементов в массиве. Это означает, что время выполнения алгоритма будет логарифмически расти с увеличением размера массива. Если размер массива очень большой, то бинарный поиск может быть значительно более быстрым, чем последовательный поиск. Однако, бинарный поиск можно использовать только с упорядоченными массивами.

Бинарный поиск используется для поиска элемента в упорядоченном массиве путем деления массива пополам и поиска искомого элемента только в той половине, где элемент может находиться. Рассмотрим пример поиска элемента в упорядоченном массиве с помощью бинарного поиска.

Пример кода на языке C#:

int[] arr = { 2, 5, 7, 10, 15, 17, 20 };

int target = 15;

int left = 0;

int right = arr.Length - 1;

int mid = 0;

int index = -1; // индекс, где мы найдем искомый элемент

while (left <= right)

{

mid = (left + right) / 2;

if (arr[mid] == target)

{

index = mid;

break;

}

else if (arr[mid] < target)

{

left = mid + 1;

}

else

{

right = mid - 1;

}

}

if (index == -1)

{

Console.WriteLine("Элемент не найден");

}

else

{

Console.WriteLine("Искомый элемент находится в массиве под индексом: " + index);

}

В данном примере мы ищем число 15 в упорядоченном массиве arr. Начиная с середины массива, мы проверяем, равен ли средний элемент target. Если да, то мы сохраняем индекс mid в переменную index и выходим из цикла. Если arr[mid] < target, то мы знаем, что искомый элемент находится в правой половине массива, поэтому мы изменяем левую границу left на mid + 1. Иначе, если arr[mid] > target, то искомый элемент находится в левой половине массива, поэтому мы изменяем правую границу right на mid - 1. Мы продолжаем делать это до тех пор, пока не находим искомый элемент или не дойдем до конца массива.

Сложность бинарного поиска составляет O(log n), где n - количество элементов в массиве. Это означает, что время поиска уменьшается в два раза после каждой итерации цикла, что делает бинарный поиск значительно быстрее, чем последовательный поиск, особенно для больших массивов. Однако, перед использованием бинарного поиска необходимо упорядочить массив, что может занять дополнительное время.

## Распределенный поиск

Распределенный поиск (distributed search) - это метод поиска, при котором поиск элемента в массиве происходит параллельно на нескольких узлах вычислительной системы. Каждый узел получает часть массива и ищет в нем элемент. После того как каждый узел закончил поиск, результаты объединяются, и искомый элемент либо находится, либо не находится в массиве. Этот метод может быть эффективным для поиска в больших массивах на вычислительных системах с большим количеством узлов, но требует наличия специального программного обеспечения и аппаратного обеспечения.

public static int InterpolationSearch(int[] arr, int target)

{

int left = 0;

int right = arr.Length - 1;

while (left <= right && target >= arr[left] && target <= arr[right])

{

// используем формулу линейной интерполяции для оценки позиции искомого элемента

int pos = left + (((target - arr[left]) \* (right - left)) / (arr[right] - arr[left]));

if (arr[pos] == target)

{

return pos; // элемент найден, возвращаем его индекс

}

else if (arr[pos] < target)

{

left = pos + 1; // искомый элемент находится в правой части массива

}

else

{

right = pos - 1; // искомый элемент находится в левой части массива

}

}

return -1; // элемент не найден, возвращаем -1

}

Здесь мы передаем методу InterpolationSearch упорядоченный массив arr и искомый элемент target. Затем мы устанавливаем левую и правую границы поиска: left равен 0 (индекс первого элемента), а right равен индексу последнего элемента (arr.Length - 1). Мы повторяем цикл while до тех пор, пока left не станет больше right, или если target находится вне границ массива. На каждой итерации мы используем формулу линейной интерполяции, чтобы оценить позицию искомого элемента. Затем мы сравниваем значение arr[pos] с искомым элементом. Если они совпадают, мы возвращаем индекс pos. Если значение arr[pos] меньше искомого элемента, мы продолжаем поиск в правой части массива, устанавливая left = pos + 1. В противном случае мы продолжаем поиск в левой части массива, устанавливая right = pos - 1.

Сложность алгоритма распределительного поиска - в худшем случае O(n), но в большинстве случаев O(log log n), где n - количество элементов в массиве. Это означает, что время выполнения алгоритма может быть значительно быстрее, чем бинарный поиск, в лучшем случае, когда элементы в массиве равномерно распределены и элемент, который мы ищем, находится близко к середине массива. Однако, если элементы в массиве не равномерно распределены, или если элемент, который мы ищем, находится ближе к краю массива, то распределительный поиск может работать медленнее, чем бинарный поиск. Также стоит учесть, что распределительный поиск требует, чтобы массив был упорядочен, что может занять дополнительное время. В целом, распределительный поиск подходит для поиска в больших упорядоченных массивах, если распределение элементов в массиве известно или можно предположить, что оно близко к равномерному.

Рассмотрим пример использования алгоритма распределительного поиска для поиска элемента в большом отсортированном массиве. Предположим, что мы хотим найти элемент со значением 23 в массиве размером 10 миллионов элементов.

Пример кода на языке C#:

using System;

class Program

{

static int DistributiveSearch(int[] arr, int target)

{

int blockSize = (int)Math.Sqrt(arr.Length);

int blockIndex = 0;

// ищем блок, в котором может быть искомый элемент

while (blockIndex < arr.Length && arr[blockIndex] < target)

{

blockIndex += blockSize;

}

// если мы вышли за границы массива, возвращаем -1

if (blockIndex >= arr.Length)

{

return -1;

}

// перебираем элементы в блоке, пока не найдем искомый элемент

for (int i = blockIndex - blockSize; i < Math.Min(blockIndex, arr.Length); i++)

{

if (arr[i] == target)

{

return i;

}

}

return -1;

}

static void Main(string[] args)

{

// создаем отсортированный массив размером 10 миллионов элементов

int[] arr = new int[10000000];

for (int i = 0; i < arr.Length; i++)

{

arr[i] = i;

}

// ищем элемент со значением 23 в массиве

int target = 23;

int index = DistributiveSearch(arr, target);

// выводим результат поиска

if (index == -1)

{

Console.WriteLine($"Элемент со значением {target} не найден в массиве");

}

else

{

Console.WriteLine($"Элемент со значением {target} найден в массиве по индексу {index}");

}

}

}

В этом примере мы используем алгоритм распределительного поиска для поиска элемента со значением 23 в массиве размером 10 миллионов элементов. Сначала мы вычисляем размер блока как корень из длины массива. Затем мы ищем блок, в котором может находиться искомый элемент, путем перебора элементов с шагом blockSize, пока не найдем блок, в котором arr[blockIndex] >= target. Затем мы перебираем элементы в найденном блоке и ищем искомый элемент. Если элемент найден, мы возвращаем его индекс, в противном случае возвращаем -1.

Сложность алгоритма распределительного поиска в худшем случае составляет O(n), но в большинстве случаев он значительно быстрее, чем простой последовательный поиск, и имеет сложность O(log log n), где n - количество элементов в массиве.

Задание практики:

шаблонная функцию поиска по ключу в массиве, содержащих целые, вещественные числа.

проверить работу функций на

1. массивах размером 500 элементов заданных рандомом

2. замерить время тестирования каждой функции и выведение результата тестирования в текстовый файл

сделать линейный поиск ееще

моя программа:

public static class Search

{

public static int BinarySearch<T>(T[] array, T key) where T : IComparable<T>

// название метода, где <T> - это обобщенный тип, который позволяет использовать метод с различными типами данных.

//where T : IComparable<T> - ограничение типа, которое гарантирует, что тип T будет реализовывать интерфейс IComparable<T>, который позволяет сравнивать объекты типа T.

{

int left = 0;

int right = array.Length - 1;

while (left <= right) // цикл продолжается до тех пор, пока левая граница не станет больше правой.

{

int middle = (left + right) / 2; // - определяет средний индекс в массиве, округленный в меньшую сторону, путем сложения левой и правой границ и деления на 2.

int comparison = key.CompareTo(array[middle]); // сравнивает ключ с элементом массива по среднему индексу и сохраняет результат в переменную comparison.

if (comparison == 0) // - если результат сравнения равен 0, значит, ключ найден в массиве, и метод возвращает индекс этого элемента в массиве.

{

return middle;

}

else if (comparison < 0) // - если результат сравнения меньше 0, значит, искомый элемент находится в левой половине массива, и правая граница поиска становится равной среднему индексу минус 1.

{

right = middle - 1;

}

else // если результат сравнения больше 0, значит, искомый элемент находится в правой половине массива, и левая граница поиска становится равной среднему индекс

{

left = middle + 1;

}

}

return -1;

}

}

public static class MainF

{

public static void Main(string[] args)

{

Random rand = new Random();

int[] intArray = new int[500];

double[] doubleArray = new double[500];

for (int i = 0; i < 500; i++)

{

intArray[i] = rand.Next(1000);

doubleArray[i] = rand.NextDouble() \* 1000;

}

Stopwatch stopwatch = new Stopwatch(); // слздаем таймер

stopwatch.Start(); // запускаем таймер

int intIndex = Search.BinarySearch(intArray, 42); // выполняем бинарный поиск по интовому массиву­­

stopwatch.Stop(); // остановили

TimeSpan intTime = stopwatch.Elapsed; // сняли время

stopwatch.Reset(); // обнулили и по новой для double массива

stopwatch.Start();

int doubleIndex = Search.BinarySearch(doubleArray, 42.0);

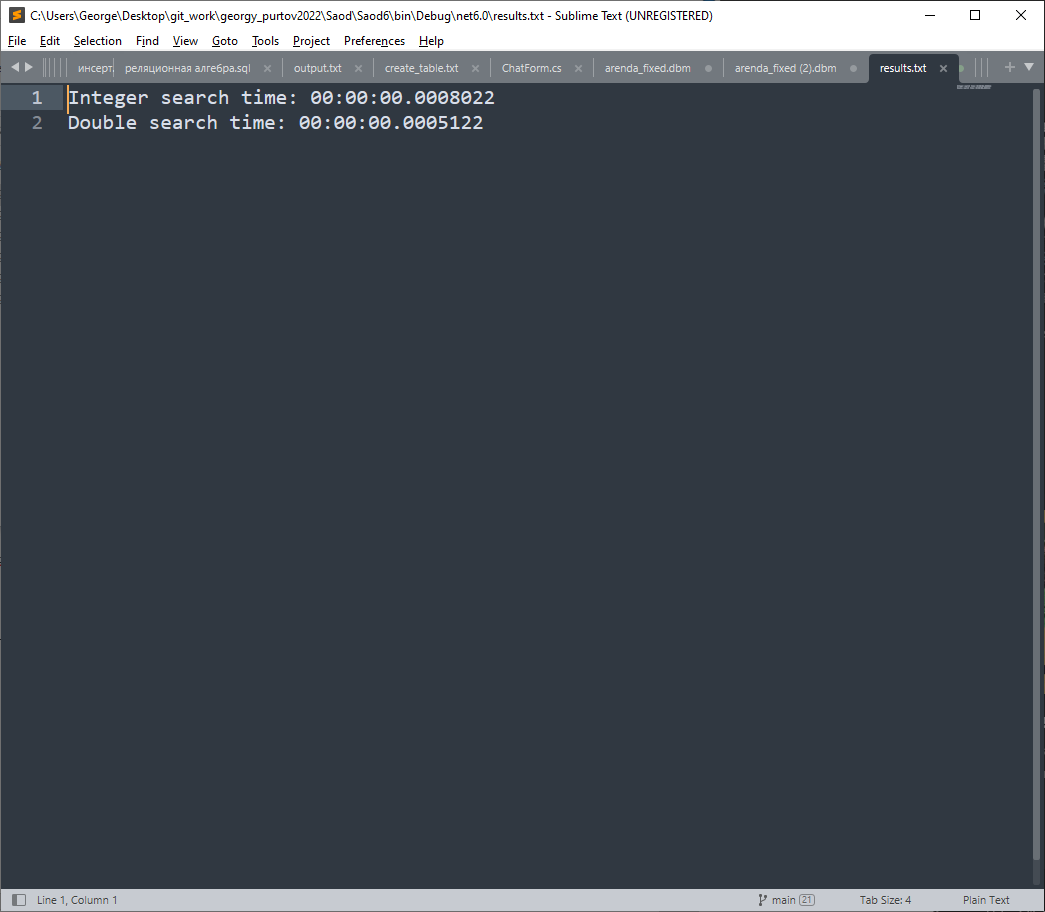
stopwatch.Stop();

TimeSpan doubleTime = stopwatch.Elapsed;

File.WriteAllText("results.txt", $"Integer search time: {intTime}\nDouble search time: {doubleTime}"); // вывод в текстовый файл

}

}

Результат тестирования: 

источник: chat.openai.com

https://proglib.io/p/10-samyh-populyarnyh-algoritmov-sortirovki-na-c-2019-11-14