Линейный поиск

Линейный или последовательный поиск – простейший алгоритм поиска. Он редко используется из-за своей неэффективности. По сути, это метод полного перебора, и он уступает другим алгоритмам.

У линейного поиска нет предварительных условий к состоянию структуры данных.

Объяснение

Алгоритм ищет элемент в заданной структуре данных, пока не достигнет конца структуры.

При нахождении элемента возвращается его позиция в структуре данных. Если элемент не найден, возвращаем -1.

Теперь посмотрим, как реализовать линейный поиск в Java:

```
public static int linearSearch(int arr[], int elementToSearch) {

    for (int index = 0; index < arr.length; index++) {
        if (arr[index] == elementToSearch)
            return index;
    }
    return -1;
}
```

Для проверки используем целочисленный массив:

```
int index = linearSearch(new int[]{89, 57, 91, 47, 95, 3, 27, 22, 67, 99}, 67);
print(67, index);
```

Простой метод для вывода результата:

```
public static void print(int elementToSearch, int index) {
    if (index == -1){
        System.out.println(elementToSearch + " not found.");
    }
    else {
        System.out.println(elementToSearch + " found at index: " + index);
    }
}
```

Вывод:

```
67 found at index: 8
```

Временная сложность

Для получения позиции искомого элемента перебирается набор из N элементов. В худшем сценарии для этого алгоритма искомый элемент оказывается последним в массиве.

В этом случае потребуется N итераций для нахождения элемента.

Следовательно, временная сложность линейного поиска равна O(N).

Пространственная сложность

Этот поиск требует всего одну единицу памяти для хранения искомого элемента. Это не относится к размеру входного массива.

Следовательно, пространственная сложность линейного поиска равна O(1).

Применение

Линейный поиск можно использовать для малого, несортированного набора данных, который не увеличивается в размерах.

Несмотря на простоту, алгоритм не находит применения в проектах из-за линейного увеличения временной сложности.

Двоичный поиск

Двоичный или логарифмический поиск часто используется из-за быстрого времени поиска.

Объяснение

Этот вид поиска использует подход «Разделяй и властвуй», требует предварительной сортировки набора данных.

Алгоритм делит входную коллекцию на равные половины, и с каждой итерацией сравнивает целевой элемент с элементом в середине. Поиск заканчивается при нахождении элемента. Иначе продолжаем искать элемент, разделяя и выбирая соответствующий раздел массива. Целевой элемент сравнивается со средним.

Вот почему важно иметь отсортированную коллекцию при использовании двоичного поиска.

Поиск заканчивается, когда firstIndex (указатель) достигает lastIndex (последнего элемента). Значит мы проверили весь массив Java и не нашли элемента.

Есть два способа реализации этого алгоритма: итеративный и рекурсивный.

Временная и пространственная сложности одинаковы для обоих способов в реализации на Java.

Реализация

Итеративный подход

Посмотрим:

```
public static int binarySearch(int arr[], int elementToSearch) {

    int firstIndex = 0;
    int lastIndex = arr.length - 1;

    // условие прекращения (элемент не представлен)
    while(firstIndex <= lastIndex) {
        int middleIndex = (firstIndex + lastIndex) / 2;
        // если средний элемент - целевой элемент, вернуть его индекс
        if (arr[middleIndex] == elementToSearch) {
            return middleIndex;
        }

        // если средний элемент меньше
        // направляем наш индекс в middle+1, убирая первую часть из рассмотрения
        else if (arr[middleIndex] < elementToSearch)
            firstIndex = middleIndex + 1;

        // если средний элемент больше
        // направляем наш индекс в middle-1, убирая вторую часть из рассмотрения
        else if (arr[middleIndex] > elementToSearch)
            lastIndex = middleIndex - 1;

    }
    return -1;
}
```

Используем алгоритм:

```
int index = binarySearch(new int[]{89, 57, 91, 47, 95, 3, 27, 22, 67, 99}, 67);
print(67, index);
```

Вывод:

```
67 found at index: 5
```

Рекурсивный подход

Теперь посмотрим на рекурсивную реализацию:

```
public static int recursiveBinarySearch(int arr[], int firstElement, int lastElement, int
elementToSearch) {
```

```
    // условие прекращения
    if (lastElement >= firstElement) {
        int mid = firstElement + (lastElement - firstElement) / 2;

        // если средний элемент - целевой элемент, вернуть его индекс
        if (arr[mid] == elementToSearch)
            return mid;

        // если средний элемент больше целевого
        // вызываем метод рекурсивно по суженным данным
        if (arr[mid] > elementToSearch)
            return recursiveBinarySearch(arr, firstElement, mid - 1, elementToSearch);

        // также, вызываем метод рекурсивно по суженным данным
        return recursiveBinarySearch(arr, mid + 1, lastElement, elementToSearch);
    }

    return -1;
}
```

Рекурсивный подход отличается вызовом самого метода при получении нового раздела. В итеративном подходе всякий раз, когда мы определяли новый раздел, мы изменяли первый и последний элементы, повторяя процесс в том же цикле.

Другое отличие – рекурсивные вызовы помещаются в стек и занимают одну единицу пространства за вызов.

Используем алгоритм следующим способом:

```
int index = binarySearch(new int[]{3, 22, 27, 47, 57, 67, 89, 91, 95, 99}, 0, 10, 67);
print(67, index);
```

Вывод:

```
67 found at index: 5
```

Временная сложность

Временная сложность алгоритма двоичного поиска равна O(log (N)) из-за деления массива пополам. Она превосходит O(N) линейного алгоритма.

Пространственная сложность

Одна единица пространства требуется для хранения искомого элемента. Следовательно, пространственная сложность равна O(1).

Рекурсивный двоичный поиск хранит вызов метода в стеке. В худшем случае пространственная сложность потребует O(log (N)).

Применение

Этот алгоритм используется в большинстве библиотек и используется с отсортированными структурами данных.

Двоичный поиск реализован в методе Arrays.binarySearch Java API.

Алгоритм Кнута – Морриса – Пратта

Алгоритм КМП осуществляет поиск текста по заданному шаблону. Он разработан Дональдом Кнутом, Воном Праттом и Джеймсом Моррисом: отсюда и название.

Объяснение

В этом поиске сначала компилируется заданный шаблон. Компилируя шаблон, мы пытаемся найти префикс и суффикс строки шаблона. Это поможет в случае несоответствия – не придётся искать следующее совпадение с начального индекса.

Вместо этого мы пропускаем часть текстовой строки, которую уже сравнили, и начинаем сравнивать следующую. Необходимая часть определяется по префиксу и суффиксу, поэтому известно, какая часть уже прошла проверку и может быть безопасно пропущена.

КМП работает быстрее алгоритма перебора благодаря пропускам.

Реализация

Итак, пишем метод compilePatternArray(), который позже будет использоваться алгоритмом поиска КМП:

```
public static int[] compilePatternArray(String pattern) {
   int patternLength = pattern.length();
   int len = 0;
   int i = 1;
   int[] compliedPatternArray = new int[patternLength];
   compliedPatternArray[0] = 0;

   while (i < patternLength) {
      if (pattern.charAt(i) == pattern.charAt(len)) {
         len++;
         compliedPatternArray[i] = len;
         i++;
      } else {
         if (len != 0) {
            len = compliedPatternArray[len - 1];
         } else {
            compliedPatternArray[i] = len;
            i++;
         }
```

```
        }
    }
    System.out.println("Compiled Pattern Array " + Arrays.toString(compliedPatternArray));
    return compliedPatternArray;
}
```

Скомпилированный массив Java можно рассматривать как массив, хранящий шаблон символов. Цель – найти префикс и суффикс в шаблоне. Зная эти элементы, можно избежать сравнения с начала текста после несоответствия и приступать к сравнению следующего символа.

Скомпилированный массив сохраняет позицию предыдущего местонахождения текущего символа в массив шаблонов.

Давайте реализуем сам алгоритм:

```
public static List<Integer> performKMPSearch(String text, String pattern) {
    int[] compliedPatternArray = compilePatternArray(pattern);

    int textIndex = 0;
    int patternIndex = 0;

    List<Integer> foundIndexes = new ArrayList<>();

    while (textIndex < text.length()) {
        if (pattern.charAt(patternIndex) == text.charAt(textIndex)) {
            patternIndex++;
            textIndex++;
        }
        if (patternIndex == pattern.length()) {
            foundIndexes.add(textIndex - patternIndex);
            patternIndex = compliedPatternArray[patternIndex - 1];
        }

        else if (textIndex < text.length() && pattern.charAt(patternIndex) !=
text.charAt(textIndex)) {
            if (patternIndex != 0)
                patternIndex = compliedPatternArray[patternIndex - 1];
            else
                textIndex = textIndex + 1;
        }
    }
    return foundIndexes;
}
```

Здесь мы последовательно сравниваем символы в шаблоне и текстовом массиве. Мы продолжаем двигаться вперёд, пока не получим совпадение. Достижение конца массива при сопоставлении означает нахождение шаблона в тексте.

Но! Есть один момент.

Если обнаружено несоответствие при сравнении двух массивов, индекс символьного массива перемещается в значение compiledPatternArray(). Затем мы переходим к следующему символу в текстовом массиве. КМП превосходит метод грубой силы однократным сравнением текстовых символов при несоответствии.

Запустите алгоритм:

```
String pattern = "AAABAAA";
String text = "ASBNSAAAAAABAAAAABAAAAAGAHUHDJKDDKSHAAJF";

List<Integer> foundIndexes = KnuthMorrisPrathPatternSearch.performKMPSearch(text, pattern);

if (foundIndexes.isEmpty()) {
   System.out.println("Pattern not found in the given text String");
} else {
   System.out.println("Pattern found in the given text String at positions: " +
.stream().map(Object::toString).collect(Collectors.joining(", ")));
}
```

В текстовом шаблоне AAABAAA наблюдается и кодируется в массив шаблонов следующий шаблон:

- Шаблон A (Одиночная A) повторяется в 1 и 4 индексах.
- Паттерн AA (Двойная A) повторяется во 2 и 5 индексах.
- Шаблон AAA (Тройная A) повторяется в индексе 6.

В подтверждение наших расчётов:

```
Compiled Pattern Array [0, 1, 2, 0, 1, 2, 3]
Pattern found in the given text String at positions: 8, 14
```

Описанный выше шаблон ясно показан в скомпилированном массиве.

С помощью этого массива КМП ищет заданный шаблон в тексте, не возвращаясь в начало текстового массива.

Временная сложность

Для поиска шаблона алгоритму нужно сравнить все элементы в заданном тексте. Необходимое для этого время составляет O(N). Для составления строки шаблона нам нужно проверить каждый символ в шаблоне – это еще одна итерация O(M).

O (M + N) – общее время алгоритма.

Пространственная сложность

O(M) пространства необходимо для хранения скомпилированного шаблона для заданного шаблона размера M.