УДК 004

Алгоритм экспоненциального поиска на языке программирования C++

Фатеенков Данила Витальевич
Приамурский государственный университет имени Шолом-Алейхема
Студент

Аннотация
В статье рассмотрен алгоритм экспоненциального поиска и его реализация на языке программирования C++. Так как алгоритм является дополнением к бинарному поиску, то также рассмотрена реализация алгоритма бинарного поиска в линейной и рекурсивной формах. Также описана производительность алгоритма экспоненциального поиска и насколько эффективно его использовать для нахождения элементов в одномерных массивах.
Ключевые слова: C++, алгоритмизация, алгоритм поиска, бинарный поиск, экспоненциальный поиск

Algorithm of exponential search in C++ programming language

Fateenkov Danila Vitalievich
Sholom-Aleichem Priamursky State University
Student

Abstract
This article considers the exponential search algorithm and its implementation in the C++ programming language. Since the algorithm is a complement to binary search, the implementation of the binary search algorithm in linear and recursive forms is also considered. The performance of the exponential search algorithm is also described, and how effective it is for finding elements in one-dimensional arrays.
Keywords: C++, algorithmization, search algorithm, binary search, exponential search

1. Введение

1.1 Актуальность

Алгоритмы поиска используются во многих IT-сферах. При работе с большим количеством данных очень важно оптимизировать и использовать определённый алгоритм поиска, чтобы достичь наилучшей производительности работы системы.

Зачастую стандартные реализации алгоритмов поиска не подходят для решения тех или иных задач (особенно могут не подойти для работы с большими данными). Из-за этого задача оптимизации алгоритмов остаётся актуальной в настоящее время. Для бинарного поиска создано уже большое

количество методов оптимизации, но одним из основных является метод экспоненциального поиска. Данный алгоритм способен улучшить работу бинарного поиска, но он не является наиболее оптимальным.

1.2 Обзор исследований

З.С. Сейдаметова описала особенности изучения сложных структур данных, в частности, BST, RBT и AVL, определены тематики изучения сбалансированных BST, а также выявлены ожидаемые результаты обучения в соответствии с уровнями образовательных целей [1].

В.И. Шелехов в своей статье описал построение и верификацию предикатной программы бинарного поиска, идентичной программе bsearch на языке Си из библиотеки ОС Linux [2].

И.В. Панкратов рассмотрел задачу нечёткого поиска булевых векторов в потоке данных. Под нечётким вхождением искомого вектора понимается вхождение вектора, близкого к искомому в смысле расстояния Хемминга [3].

А.А. Ефимчик рассмотрел алгоритмы, которые нужны для осуществления поиска тех или иных элементов в больших массивах данных [4]. В статье рассматривались следующие алгоритмы: линейный поиск и бинарный поиск.

А.В. Параничев и В.А. Филипович описали модификацию бинарного дерева на основе классического красно-черного дерева [5]. В статье представлена программная реализация бинарного дерева для выполнения специализированных операций.

1.3 Цель исследования

Цель – реализовать алгоритм экспоненциального поиска.

2. Материалы и методы

Для реализации поставленной цели используется язык программирования C++.

3. Результаты и обсуждения

Экспоненциальный поиск является дополнением к бинарному поиску и наличие двоичного поиска в коде программы обязательно. Поэтому сначала стоит описать алгоритм двоичного поиска, так как он является основой алгоритма.

Двоичный поиск является одним из наиболее известных алгоритмов поиска в информатике и программировании за счёт своей простоты реализации и оптимальной производительности при средних и худших случаях. Алгоритм работает за логарифмическое время и только с отсортированными по определённому условию массивами данных (чаще всего бинарный поиск используется для нахождения целочисленных значений). При попытке выполнить алгоритм на неотсортированном массиве приводит к получению неправильных результатов и на данный момент не

существует модификации поиска для работы с массивами, в которых элементы расположены случайным образом.

Алгоритм двоичного поиска заключается в выполнении следующих шагов:

1. Вычисляется расположение серединного элемента массива. Данный шаг делит массив на 2 части. Длины частей могут различаться на 1 элемент при нечётной длине исходного массива.

2. Алгоритм сверяет серединный элемент с ключом (элемент, который необходимо найти) и если элемент не равен ключу, то на его основе определяется часть массива, с которой необходимо работать дальше:

2.1. Серединный элемент больше того, что нужно найти: в таком случае необходимо выбрать часть, которая находится слева.

2.2. Серединный элемент меньше того, что нужно найти: в таком случае алгоритм выбирает часть, которая находится справа.

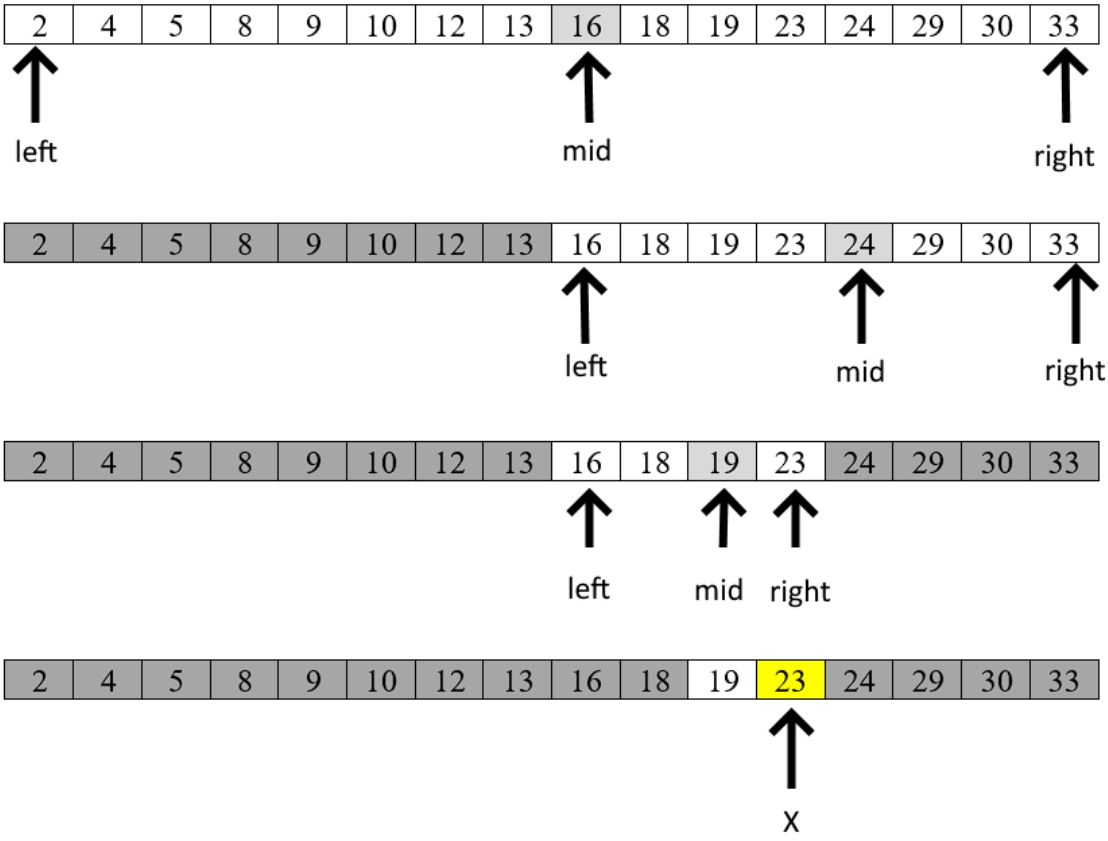

Рисунок 1. Пример нахождения числа 23 с помощью бинарного поиска

Второй шаг повторяется до тех пор, пока не будет найден элемент или не будет подтверждено его отсутствие. Нетрудно заметить, что алгоритм является рекурсивным, но также существует его линейная реализация.

В случае с рекурсивной реализацией можно формировать подмассив исходного массива, в котором осуществляется поиск и снова вызывать функцию поиска, не выходя из неё же.

```
bool binary_search(int arr[], int x, int size) {
     if (size == 0) {
          return false;
     }
     int middle = size / 2;
     if (x > arr[middle]) {
          return binary_search(arr + middle + 1, x, size -
middle - 1);
     }
     else if (x < arr[middle]) {
          return binary_search(arr, x, middle);
     }

     return true;
}
```

В случае с линейной реализацией учитываются дополнительные переменные, которые содержат в себе левую и правую границы. В зависимости от выполнения условия необходимо сдвигать ту или иную границу до тех пор, пока не будет найден элемент или не доказано его отсутствие в массиве.

```
void binary_search(vector<int>& arr, int x, int l, int r) {
     int res = -1;
     while (l < r - 1) {
          int m = (r + l) / 2;
          if (arr[m] == x) {
               res = m;
               break;
          }
          else if (arr[m] < x) l = m;
          else r = m;
     }
     cout << res;
}
```

Время работы алгоритма в худшем и среднем случаях: O(logn), где n – длина исходного массива. Под худшим случаем стоит понимать тот случай, когда элемент находится на первой или поздней позиции массива. Под средним случаем предполагается, что элемент, который необходимо найти, находится на случайной позиции.

Близким по идее реализации поиска в массиве данных является бинарное дерево поиска, суть которого заключается в разделении массива и составлении из его элементов древовидной структуры, слева от родителя которой расположены элементы, меньшие родительского и справа от родителя расположены большие элементы. Поиск осуществляется посредством передвижения по дереву и сравнения узлов с искомым элементом (см. рис. 2).

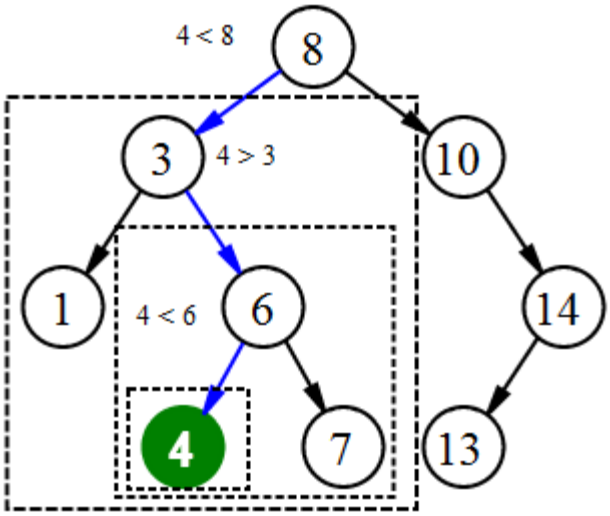

Рисунок 2. Пример нахождения числа 4 с помощью бинарного дерева поиска

Экспоненциальный поиск может превзойти бинарный поиск в плане производительности и в среднем он выполняется за O(logi), где i – позиция искомого элемента в массиве. Данный параметр является основным отличием от бинарного поиска (в лучшую сторону, конечно). Экспоненциальный поиск предназначен в первую очередь для уточнения диапазона поиска необходимого элемента массива с последующим нахождением элемента с применением бинарного поиска. Алгоритм экспоненциального поиска заключается в следующем:

1. Для определения границы необходимо ввести дополнительную переменную. Её значение устанавливается равной единице.
2. Если значение переменной границы больше длины массива, то необходимо выполнить двоичный поиск в промежутке от половины значения ранее введённой переменной до крайнего правого элемента массива. Иначе нужно перейти к третьему шагу.
3. Если значение на индексе дополнительной переменной больше искомого элемента, то бинарный поиск необходимо осуществить в промежутке от половины значения дополнительной переменной до значения дополнительной переменной. Иначе перейти к четвёртому шагу.
4. Увеличить значение переменной границы в 2 раза.

-5	-3	-2	0	1	2	5	6	9	11	13	15
0	1	2	3	4	5	6	7	8	9	10	11

x = 5

1. border = 1

-5	-3	-2	0	1	2	5	6	9	11	13	15
0	1	2	3	4	5	6	7	8	9	10	11

2. border = 2

-5	-3	-2	0	1	2	5	6	9	11	13	15
0	1	2	3	4	5	6	7	8	9	10	11

3. border = 4

-5	-3	-2	0	1	2	5	6	9	11	13	15
0	1	2	3	4	5	6	7	8	9	10	11

4. border = 8

-5	-3	-2	0	1	2	5	6	9	11	13	15
0	1	2	3	4	5	6	7	8	9	10	11

$$9 > 5 \Rightarrow \begin{cases} L = 4 \\ r = 8 \end{cases} \Rightarrow [1, 2, 5, 6, 9]$$

Рисунок 3. Пример работы алгоритма экспоненциального поиска

Данный алгоритм не требует рекурсивной реализации и прост в линейном исполнении:

```
void exponential_search(vector<int>& arr, int x) {
     int border = 1;
     while (border < arr.size()-1 && arr[border] < x) {
          border *= 2;
     }
     if (border > arr.size()-1) {
          border = arr.size()-1;
     }
     binary_search(arr, x, border / 2, border);
}
```

Вначале алгоритма задаётся новая переменная border, которая увеличивается в 2 раза до тех пор, пока элемент, находящийся на позиции значения переменной не станет больше искомого элемента или значение переменной не выйдет за границы массива. В случае выхода за границы происходит поиск в промежутке (border / 2, border), где border это крайний элемент массива справа. После вызывается функция бинарного поиска для необходимого промежутка.

Таким образом производительность алгоритма экспоненциального поиска составляет O(logi), как и было описано ранее. Данная производительность может улучшить работу алгоритма бинарного поиска, но зачастую прирост производительности незначителен и время отличается на десятки миллисекунд. Такой прирост актуален при работе с большими данными.

Также стоит учитывать рост области поиска, в котором может находиться необходимый элемент, на каждом шаге в 2 раза. Поиск на промежутке, в который входят 256 элементов массива значительно лучше поиска среди всех элементов, которых может быть больше тысячи. Но при этом нужно учитывать, что область поиска может быть также слишком большой и для поиска на ней также может требоваться оптимизация алгоритма.

Выводы

В данной статье был описан алгоритм экспоненциального поиска, а также рассмотрено его влияние на производительность алгоритма бинарного поиска.

Библиографический список

1. Сейдаметова З.С. Особенности изучения сбалансированных бинарных деревьев поиска // Информационно-компьютерные технологии в экономике, образовании и социальной сфере. 2019. № 3 (25). С. 83-93.
2. Шелехов В.И. Верификация предикатной программы бинарного поиска объекта произвольного типа // Системная информатика. 2019. № 15. С. 45-64.
3. Панкратов И.В. Применение конечных автоматов для нечёткого бинарного поиска // Прикладная дискретная математика. Приложение. 2018. № 11. С. 117-122.
4. Ефимчик А.А. Анализ алгоритмов поиска в больших массивах данных // Вестник современных исследований. 2018. № 12.1 (27). С. 537-541.
5. Параничев А.В., Филипович В.А. Программная реализация бинарного дерева на основе модификации красно-черного дерева // Информационные системы и технологии в моделировании и управлении. Сборник трудов V Международной научно-практической конференции. 2020. С. 75-79.
6. Binary Search - Geeksforgeeks URL: https://www.geeksforgeeks.org/binary-search/. (Дата обращения: 11.01.2023).
7. Exponential Search - Geeksforgeeks URL: https://www.geeksforgeeks.org/exponential-search/. (Дата обращения: 11.01.2023).
8. Целочисленный двоичный поиск – Викиконспекты URL: https://neerc.ifmo.ru/wiki/index.php?title=Целочисленный_двоичный_поиск. (Дата обращения: 22.12.2022).