УДК 004

Применение методов троичного поиска для реализации алгоритма сортировки массива целочисленных значений

Фатеенков Данила Витальевич
Приамурский государственный университет имени Шолом-Алейхема
Студент

Аннотация
В статье рассмотрен алгоритм троичного поиска. Описана его реализация на языке C++ и сравнена с другими алгоритмами поиска. Представлена реализация сортировки вставками числового массива с использованием реализованного алгоритма поиска применён для сортировки числового массива.
Ключевые слова: троичный поиск, сортировка, алгоритмизация, алгоритмы поиска, C++

Using methods of ternary search to implement an array-sorting algorithm

Fateenkov Danila Vitalievich
Sholom-Aleichem Priamursky State University
Student

Abstract
The article deals with the ternary search algorithm. Its implementation in C++ is described and compared to other search algorithms. The implementation of inset sorting of a numeric array using the implemented search algorithm is presented.
Keywords: ternary search, sorting, algorithmization, search algorithms, C++

1. Введение

1.1 Актуальность

Поиск информации является одной из основных задач в алгоритмизации. Алгоритмы поиска распространены во многих областях IT-сферы, а также в других сферах жизни человека. Реализация оптимизированного алгоритма поиска позволяет упростить жизнь человеку, который работает с большим количеством информации, где поиск может быть затруднительным без автоматизации процесса.

В настоящее время существует большое количество алгоритмов поиска (как для числовых данных, так и для текстовых) и новые алгоритмы и их модификации продолжают разрабатываться. Многие из новых алгоритмов опираются на математические алгоритмы и являются эффективными по времени, что является одним из важнейших показателей эффективности алгоритма. Один из таких алгоритмов – троичный (или тернарный) поиск.

1.2 Обзор исследований

О.Б. Попова и Н.В. Богацкий описали способы реализации двоичного дерева поиска и длинной арифметики на языке программирования C#, а результаты представлены в виде теоретических выводов, которые основаны на практических исследованиях [1]. А.М. Глухов представил сравнение алгоритмов поиска [2]. В статье рассматривались линейный поиск, бинарный поиск и интерполирующий поиск. Т.С. Бадасян и С.К. Авагян рассмотрели красно-чёрное дерево поиска, его свойства, реализацию и балансировку [3]. Я.Е. Ромм и Д.А. Чабанюк описали и реализовали параллельное построение двоичного дерева на основе сортировки [4]. А.Р. Агабек и А.Е. Дюсембаев представили вариант оптимизации алгоритма поиска для пространственных объектов за счёт использования двоичных деревьев разбиения пространства k-d дерева [5].

1.3 Цель исследования

Цель – реализовать алгоритм троичного поиска и применить его для сортировки массива.

2. Материалы и методы

Для реализации алгоритма будет использован язык программирования C++.

3. Результаты и обсуждения

В настоящее время чаще всего используются 2 алгоритма поиска: линейный и бинарный. Идея линейного алгоритма заключается в последовательном проходе по элементам списка с последующим сравнением с ключом (значение, которое необходимо найти). Алгоритм прост в реализации, но является не эффективным (особенно при работе с большими наборами данных). Данный алгоритм поиска на C++ можно представить следующим образом:

```
for (int i=0;i<arr.size();i++) {
     if (arr[i] == X) return i;
return -1;
```

Данный код ищет X среди всех элементов массива arr и возвращает его позицию (либо возвращает -1, если элемент не найден). В лучшем случае искомый элемент будет находиться на нулевой позиции и поиск завершится после первого сравнения, а в худшем случае сравнений будет n соответственно (где n – длина массива).

Алгоритм подходит для работы с небольшими наборами данных, но существует более совершенный вариант поиска – бинарный.

Идея алгоритма заключается в следующем:

1. Выбирается элемент части массива (на первом шаге выбирается весь массив) и сравнивается с искомым элементом.

2. Если элемент меньше X, то выбирается правая часть оставшегося списка и повторяется первый шаг.

3. Если элемент больше X, то выбирается левая часть оставшегося списка и также повторяется первый шаг.

Таким образом количество сравнений при поиске элемента значительно сокращается, а также повышается эффективность работы алгоритма. Поиск продолжается до тех пор, пока не будет найден нужный элемент или пока интервал для поиска не станет пустым. Время работы алгоритма составляется log2(n), что является хорошим показателем.

Главная проблема алгоритма двоичного поиска – его можно осуществлять только в массивах, элементы которого заранее упорядочены (иначе появляется риск пропуска необходимого значения).

Одним из улучшений бинарного поиска считается троичный (или тернарный) поиск – метод нахождения экстремумов функции. Суть метода состоит в делении графика функции на 3 части, предварительно вычисляя значения точек a и b, которые содержат значения функции в этих точках, с последующим поиском экстремума в двух из трёх частей.

Точки a и b вычисляются по заранее определённым формулам (например, a= $l + \frac{r-l}{3}$, b=$l + 2 * \frac{r-l}{3}$ [6]). После нахождения значений в полученных точках, они сравниваются и выбираются 2 части для продолжения поиска:

1. Если f(a) > f(b), то выбираются средняя и крайняя правая части.

2. Если f(a) < f(b), то выбираются крайняя левая и средняя части.

3. Если f(a) = f(b), то выбирается одна из частей (чаще всего выбирается середина).

Также вводится мера точности e, которая характеризует, насколько глубоко производится поиск (чем меньше значение меры, тем меньше будет конечный отрезок, на котором выбирается минимальный элемент).

Алгоритм завершает свою работу при достижении минимального значения на выбранном отрезке ли прекращения выполнения неравенства “r - l > e”.

Алгоритм троичного поиска применяется часто в математике, но в программировании он задействуется намного реже бинарного поиска. При этом эффективная реализация троичного поиска может по времени работы превзойти бинарный поиск (также троичный поиск не предполагает работу с уже отсортированным массивом).

Пример: на вход поступает последовательность чисел [2, 12, 5, 36, 63, 4, 1, 75, 60, 8, 54, 11] и необходимо найти минимум на промежутке от 3 до 10 (отсчёт начинается с нуля). Мера точности e равна 2.

1. Промежуток от 3 до 10 содержит следующие элементы: 36, 63, 4, 1, 75, 60, 8, 54. Точка A, исходя из приведённой выше формулы, расположена на 5 элементе, а точка B на 7 элементе. Значение элемента массива в точке A равно 4, а значение в точке B 75. Сравниваются 4 и 75 и выбираются

соответствующие части для следующего шага. Остаётся следующий промежуток значений: 36, 63, 4, 1, 75.

2. l не меняет своего значения и равно 3 (так как сдвинулась только правая граница), а r = 7. По соответствующим формулам точки A и B равны 4 и 5, соответственно. Значения элементов в точках A и B равны 63 и 4. Значение в точке A больше, значит выбирается средняя и крайняя правая части: 63, 4, 1, 75.

3. l = 4 и r = 7 (всё ещё выполняется неравенство “r - l > e”). A = 5 и B = 6. Значения в этих точках 4 и 1, соответственно. Значение в точке A больше, выбираются средняя и правая части: 4, 1, 75.

4. l = 5 и r = 7. Неравенство “r - l > e” более не выполняется и на оставшейся части исходного массива выбирается минимальный элемент. Минимальный элемент промежутка [36, 63, 4, 1, 75, 6, 8, 54] будет 1.

Данный алгоритм поиска применим для сортировки массивов и может быть использован в паре с сортировкой вставками, например. Функция поиска выполняется рекурсивно и имеет следующие входные параметры:

1. array – массив, в котором производится поиск.
2. l – левая граница массива.
3. r – правая граница массива.
4. e – мера точности, которой необходимо достигнуть во время поиска.

Первым действием производятся проверки на граничные случаи (длина массива равна одному или двум элементам) и если массив не соответствует условиям, то производится нахождение a и b:

```
void TernarySearchSort(vector<int>& arr, int l, int r, int
e) {
     if (main_arr.size() == 0) {
          return;
     }
     if (arr.size() == 1) {
          res_arr.push_back(arr[0]);
          return;
     }

     int a=l+floor((r-l)/3),b=l+2*floor(((r-l)/3));
}
```

Для работы используются заранее объявленные main_arr – вектор, содержащий входные значения и res_arr – результирующий вектор. Данные векторы объявлены до начала выполнения функции main и являются глобальными:

```
vector<int> res_arr;
vector<int> main_arr;
```

Следующий шаг после нахождения a и b – проверка выполнения неравенства: “r – l > e”. Если данное неравенство выполняется, то осуществляется сравнение элементов массива по индексам a и b с последующим формированием нового массива и запуска функции TernarySearchSort с теми двумя частями массива, в которых продолжается поиск:

```
if (r-l > e) {
     vector<int> new_arr;
     if (arr[a] > arr[b]) {
          for (int i=a;i<r+1;i++) new_arr.push_back(arr[i]);
          TernarySearchSort(new_arr,1,new_arr.size(),e);
     }
     else if (arr[a] < arr[b]) {
          for (int i=l;i<a+1;i++) new_arr.push_back(arr[i]);
          TernarySearchSort(new_arr,1,new_arr.size(),e);
     }
     else {
          for (int i=a;i<b+1;i++) new_arr.push_back(arr[i]);
          TernarySearchSort(new_arr,1,new_arr.size(),e);
     }
}
```

Не трудно определить, что мера точности e является количеством элементов в массиве, в котором производится поиск. Таким образом, оптимальными значениями будут 1 или 2 (ниже будет описан случай, когда e = 2).

Если неравенство не является истинным, то запускается процесс внесения найденного значения в результирующий массив. Сравниваются 2 оставшихся числа и выбирается минимальное из пары, после чего оно заносится в массив. Необходимо учитывать, что в массиве могут встречаться повторяющиеся значения. Чтобы избежать поиска и добавления уже ранее найденных элементов, достаточно узнать – сколько раз встречается элемент с таким же значением в массиве, занести такое же количество элементов в результирующий массив, а из исходного удалить все элементы с такими значениями. После внесения нового значения в результирующий массив, исходный меняется (за счёт удаления уже добавленных значений) и поиск начинается сначала. Поиск повторяющихся значений можно реализовать с помощью алгоритмов линейного или бинарного поиска:

```
else {
     int min;
     if (arr[0] < arr[1]) {
          min = arr[0];
     }
     else {
          min = arr[1];
     }
```

```
        int q=0;
        for (int j=0;j<main_arr.size();j++) {
             if (main_arr[j] == min) {
                  q++;
                  main_arr.erase(main_arr.begin() + j);
             }
        }
        insertElement(min,q);

        TernarySearchSort(main_arr,1,3,e);
   }
```

Функция insertElement добавляет новый элемент и сортирует его в результирующем массиве. Также необходимо проверить, является ли массив пустым, чтобы избежать ошибок выхода за границы массива. На вход функция получает 2 параметра:

1. num – число, которое необходимо занести в массив.
2. quantity – количество появлений данного числа в исходном массиве.

```
void insertElement(int num, int quantity) {
     if (res_arr.size() == 0) {
          res_arr.push_back(num);
          quantity--;
     }
     for (int i=1;i<=quantity;i++) {
          res_arr.push_back(num);
          int cur = res_arr.size()-1;
          while (res_arr[cur] < res_arr[cur-1]) {
               swap(res_arr[cur],res_arr[cur-1]);
               cur--;
          }
     }

     return;
}
```

Поиск чувствителен к расположению элементов в списке и 2 массива с одинаковыми, но по-разному расположенными элементами, будут на выходе иметь разные последовательности значений перед сортировкой (см. рис. 1).

Но на граничных случаях, то есть, когда массив отсортирован, алгоритм работает без использования дополнительного алгоритма сортировки (независимо от того, как отсортирован массив – согласно условию или обратно заданному условию).

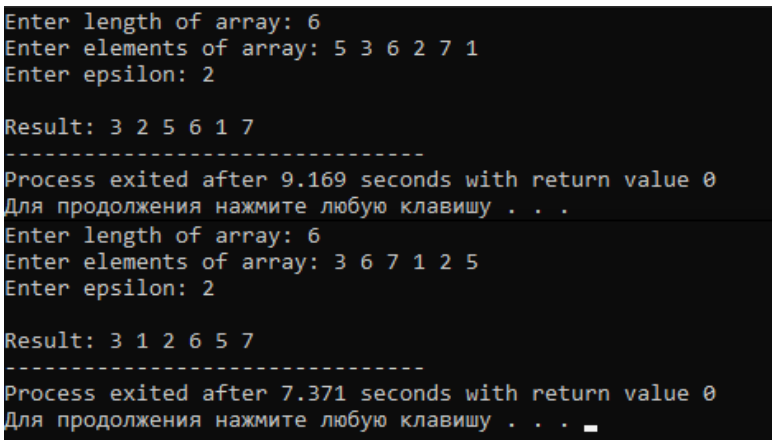

Рисунок 1. Результат работы алгоритма для двух массивов с одинаковыми значениями, но расположенными на различных позициях.

Данную проблему нельзя исправить, так как алгоритм заранее не может знать, в какой из частей находится минимум, основываясь на значениях только двух элементов массива. Но нет необходимости выбирать произвольный промежуток, на котором будет производиться поиск – стоит выбирать весь массив, то есть l равен 0 и r равен длине поступающего в функцию массива (в задаче сортировки массива это правильный подход, если не подразумевается деление исходного массива на составные части). Также можно изменить процесс выбора минимального значения и сделать его для произвольного значения меры точности e. Решив проблему выбора промежутка, на котором производится поиск, всё равно нельзя отказаться от дополнительной сортировки, так как алгоритм не будет гарантированно находить минимум среди всех элементов массива (работает только в случае, когда выбраны границы l = 0 и r = array.size): “TernarySearchSort(new_arr,0,new_arr.size()-1,e)”. Для доработки процесса нахождения минимума после достижения значения меры точности будет внесены следующие изменения:

```
int min=arr[0];
for (int i=0;i<arr.size();i++) {
     if (arr[i] < min)
          min = arr[i];
}
int q=0;
for (int j=0;j<main_arr.size();j++) {
     if (main_arr[j] == min) {
          q++;
          main_arr.erase(main_arr.begin() + j);
     }
}
insertElement(min,q);

TernarySearchSort(main_arr,0,main_arr.size()-1,e);
```

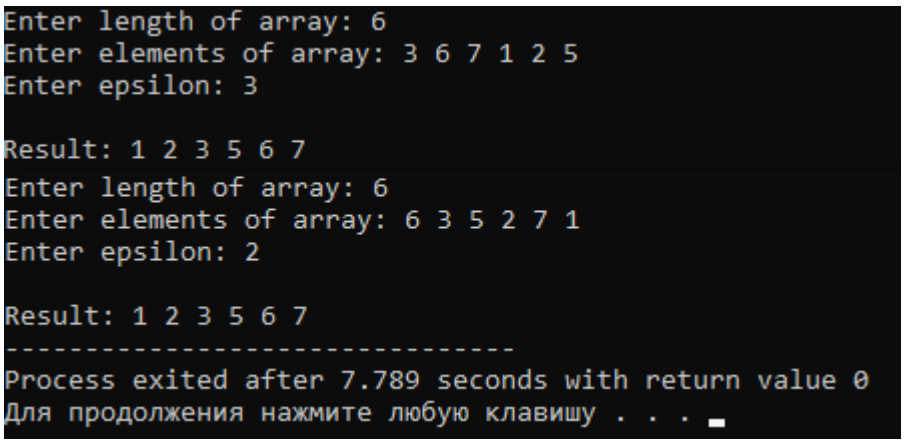

Рисунок 2. Результат работы алгоритма поиска и сортировки.

Проблема точного нахождения минимума заключается в мере точности: если она меньше длины массива, то отсутствует гарантия того, что будет найден минимум среди всех элементов массива. Это связано с тем, что часть, в которой расположен минимум, может быть пропущена алгоритмом, так как не выполняется соответствующее условие выбора частей для продолжения поиска. Данная проблема исправима, если мера точности всегда будет равна длине массива, но в таком случае алгоритм сводится к линейному поиску с последующей сортировкой вставками, что не соответствует поставленной задаче.

Таким образом, оптимальным вариантом решения такой проблемы будет реализация алгоритма сортировки (можно использовать любой, но был выбран алгоритм сортировки вставками).

Алгоритм устойчив к повторяющимся значениям за счёт того, что после нахождения минимума, вычисляется количество появлений данного минимума в массиве. Все элементы добавляются в результирующий массив сортировкой вставками.

Данный процесс можно немного оптимизировать, если запоминать расположение первого добавленного элемента и повторы значений добавить после сохранённого индекса первого элемента.

Время работы для нахождения одного элемента составит $2log_{\frac{3}{2}}(\frac{r-l}{e})$, потому что необходимо вычислить 2 значения функции и на каждой итерации область поиска уменьшается в полтора раза, поиск продолжается, пока не будет выполнено условие “r - l <= e”. Данный логарифм умножается на количество уникальных элементов в массиве, так как перед новой итерацией алгоритма повторяющиеся элементы удаляются после добавления в результирующий массив. Также необходимо учесть алгоритм сортировки, который в среднем случае выполняется за O(n^2) (можно выбрать более оптимальный по времени алгоритм, данная сортировка выбрана как пример работы общей идеи сортировки с применением троичного поиска). Также учитывается нахождение минимума среди оставшихся значений после выполнения условия меры точности – всегда время работы данного алгоритма будет равно e, так как используется линейный поиск. Проблема с

подсчётом времени работы возникает на шаге поиска повторяющихся значений. Проблема связана с меняющейся длиной массива из-за удаления повторов. Данный процесс можно пропустить, но тогда алгоритм троичного поиска будет находить одинаковые значения, что увеличит время выполнения работы алгоритма.

Также можно оптимизировать использование памяти, отказавшись от использования глобальных переменных, в которых хранятся массивы.

Сам алгоритм троичного поиска можно оптимизировать, если реализовать поиск с использованием золотого сечения. В процессе выполнения троичного поиска вычисляется и сравниваются 2 значения массива, но поиск с применением золотого сечения не предполагает этого – используется только одно значение (за исключением первой итерации).

При использовании поиска с помощью золотого сечения, вводится новое значение φ (характеризующее золотое сечение) – некоторое отношение, в котором делится исходный массив. После достижения значения данной переменной, одна из точек фиксируется, что позволяет улучшить работу алгоритма поиска. Вторая точка будет смещаться до тех пор, пока не будет достигнуто необходимое отношение между частями массива. Таким образом, время выполнения работы алгоритма составит $log_{\varphi}(\frac{r-l}{e})$. Принято задавать φ следующее значение: $\frac{1+\sqrt{5}}{2}$, что приблизительно 1.618.

Таким образом, с применением языка программирования C++, был реализован алгоритм троичного поиска минимума в целочисленном массиве и адаптирован для задачи сортировки массива. Также была описана эффективность алгоритма, от чего она зависит и как можно улучшить троичный поиск.

В статье был описан алгоритм троичного поиска, программная реализация на C++.

Библиографический список

1. Попова О.Б., Богацкий Н.В. Способы организации двоичного дерева поиска и длинной арифметики на языке программирования C#. - Петрозаводск : Международный центр научного партнерства «Новая Наука» (ИП Ивановская И.И.), 2020. - 116-128 с.

2. Глухов А.М. Сравнительный анализ трудоемкости алгоритмов поиска в программировании // Академия педагогических идей новация. Серия: студенческий научный вестник. - 2018. - № 5. - С. 232-242.

3. Бадасян Т.С., Авагян С.К. Красно-чёрное дерево: балансирование и сложность // Наука, техника и образование. - 2020. - № 3 (67). - С. 26-31.

4. Ромм Я.Е., Чабанюк Д.А. Параллельное построение двоичного дерева на основе сортировки // Вестник Донского государственного технического университета. - 2018. - Т. 18. - № 4. - С. 449-454.

5. Агабек А.Р., Дюсембаев А.Е. Об одном подходе по построению k-d дерева для пространственного поиска при малых k // Актуальные научные исследования в современном мире. - 2020. - № 4-1 (60). - С. 31-36.

6. Троичный поиск – Викиконспекты. URL: https://neerc.ifmo.ru/wiki/index.php?title=Троичный_поиск. - Дата обращения: 06.06.2022.

7. Поиск с помощью золотого сечения – Викиконспекты. URL: https://neerc.ifmo.ru/wiki/index.php?title=Поиск_с_помощью_золотого_сечения . – Дата обращения: 07.06.2022.