УДК 004

Оптимизация алгоритма бинарного поиска на языке программирования C++

Фатеенков Данила Витальевич
Приамурский государственный университет имени Шолом-Алейхема
Студент

Аннотация
В данной статье описан улучшенный и оптимизированный алгоритм бинарного поиска. Рассматривается метод sqrt-декомпозиции и его применение при реализации бинарного поиска. Также в статье представлен код реализации оптимизированного бинарного поиска с применением декомпозиции массива на отрезки равной длины.
Ключевые слова: C++, алгоритмизация, поиск, бинарный поиск, sqrt-декомпозиция

Binary search algorithm optimization in C++ programming language

Fateenkov Danila Vitalievich
Sholom-Aleichem Priamursky State University
Student

Abstract
This article presents an improved and optimized binary search algorithm. The method of sqrt-decomposition and its application to binary search implementation are considered. It also presents code for the implementation of an optimized binary search using array decomposition into equal-length segments.
Keywords: C++, algorithmization, search, binary search, sqrt-decomposition

1. Введение

1.1 Актуальность

Алгоритмы часто нуждаются в оптимизации из-за постоянно изменяющихся условий работы с ними. Часто классические реализации алгоритмов не только поиска, но и сортировок, не подходят для решения поставленной задачи.

По этой причине методы оптимизации методов и алгоритмов поиска остаются актуальными и востребованными в IT-сфере. Алгоритмы трансформируют или создают гибриды между различными вариациями, чтобы значительно сократить время поиска тех или иных данных в различных структурах (классах, базах данных, сложных структурах).

Одним из таких методов оптимизации бинарного поиска является внедрение метода sqrt-декомпозиции в работу бинарного поиска. Данное

улучшение может позволить значительно улучшить и ускорить работу алгоритма.

1.2 Обзор исследований

А.М. Глухов представил сравнение линейного, бинарного и интерполирующих алгоритмов поиска [1]. О.Б. Попова и Н.В. Богацкий представили различные способы реализации двоичного дерева поиска и длинной арифметики на языке программирования C# [2]. Я.Е. Ромм и С.С. Белоконова описали метод разрядного распараллеливания операций сравнения, который предназначен для применения в системах информационного поиска [3]. З.С. Сейдаметова рассмотрела особенности изучения сбалансированных бинарных деревьев поиска [4].

1.3 Цель исследования

Цель – оптимизировать алгоритм бинарного поиска, применяя метод sqrt-декомпозиции массива чисел.

2. Материалы и методы

Для реализации поставленной цели используется язык программирования C++.

3. Результаты и обсуждения

Перед разбором способа оптимизации алгоритма двоичного поиска, необходимо описать его классическую реализацию. Алгоритм является одним из основных алгоритмов поиска, который применяется не только в программировании, но и также в математике (метод бисекции для нахождения приближённого решения уравнений).

Алгоритм прост и в своём принципе, и в реализации: некоторое множество чисел сортируется по возрастанию или убыванию и делится на 2 равные части. Если предположить, что во множестве необходимо найти число X, то первым шагом необходимо проверить – превосходит ли число в середине множества искомое.

Если X больше, то половина множества справа также делится на 2 части и середина подмножества сверяется с X. Так продолжается до тех пор, пока не будет найден X (либо алгоритм не дойдет до крайнего элемента). Данный шаг также актуален, если X меньше серединного числа исходного множества, но тогда поиск будет производиться в левой части множества по тому же принципу.

Для примера задан массив целочисленных значений: [1, 3, 4, 7, 8, 10, 13, 15, 14, 17, 20] и необходимо найти 8.

1. Если разделить исходный массив на две примерно равные части (если массив содержит чётное количество элементов, то получится разделить на 2 равные части, иначе длины будут отличаться на 1). В середине стоит чисто 10, которое и сравнивается с 8-ю. Искомое число меньше, а значит поиск сдвигается в левую часть массива.

2. Получился подмассив [1, 3, 4, 7, 8, 10], длина которого 6. При делении на 2 части не получится однозначно выделить серединный элемент, поэтому можно взять любой близлежащий к середине. В данном случае 8 больше, чем 4 и больше, чем 7, поэтому поиск сдвигается в правую часть подмассива.

3. Подмассив [7, 8, 10] также делится на 2 части. В левой части стоит элемент, равный 7, а в правой части остаются 8 и 10. На данном шаге достаточно проверить правую часть и сравнить 2 оставшихся числа с искомым значением. Число 8 стоит на пятой позиции в исходном массиве.

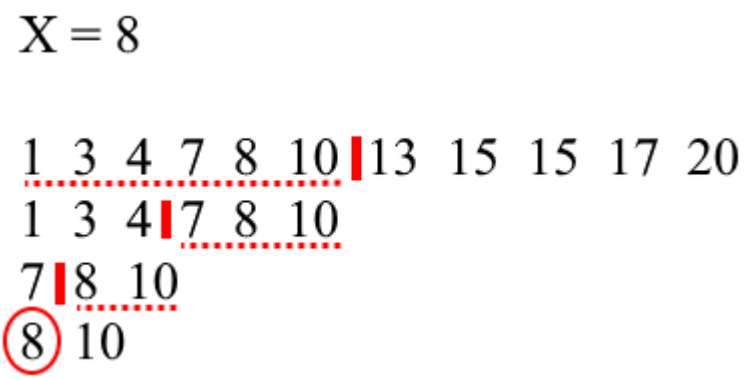

Рисунок 1. Пример работы бинарного поиска

Алгоритм бинарного поиска удобен при работе с небольшим количеством чисел, но если массив состоит из нескольких сотен тысяч элементов, то поиск может затянуться на десять и больше секунд. Сложность алгоритма в среднем составляет O(log(n)) с округлением до большего числа, что является хорошим показателем, но не наилучшим. Можно заметить, что лучшим случаем для бинарного поиска будет, когда искомое число будет находиться в середине исходного массива. Тогда сложность алгоритма составит O(1), что является наилучшим показателем в алгоритмах (но, можно заметить, что линейный алгоритм также будет выполняться за O(1), если искомый элемент будет стоять с левой границы множества чисел).

Классический (итерационный) метод реализации алгоритма можно представить следующим образом:

```
int binary_search(vector<int>& arr, int x) {
    int l = 0;
    int r = arr.size();
    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;
        }
        return res;
    }
```

```
6
1 2 3 5 6 7
6
Position of 6 in array is 4
6
1 2 3 5 6 7
0
There's no any 0 in array
--------------------------------
Process exited after 15.95 seconds with return value 0
Для продолжения нажмите любую клавишу . . .
```

Рисунок 2. Пример работы алгоритма бинарного поиска

Можно определить следующие проблемы алгоритма:

1. Бинарный поиск не предназначен для нахождения элементов в массивах, которые не отсортированы. Эту проблему не получится решить из-за специфики алгоритма.

2. Проблема выбора позиции искомого элемента, если он встречается в исходном массиве несколько раз.

Если вернуться к массиву [1, 3, 4, 7, 8, 10, 13, 15, 15, 17, 20] и произвести поиск числа 15, то это число будет найдено на позиции 9, хотя 15 встречается также на позиции 8.

Данную неоднозначность можно решить проверкой соседних элементов на равенство с искомым числом.

3. Алгоритм становится неэффективным при работе с большим количеством данных из-за постоянного процесса деления массива на части.

Данную проблему можно решить модификацией алгоритма поиска.

При работе с большими массивами нет необходимости делить их на 2 части постоянно: можно построить отрезки из элементов подмассива исходного множества и искать нужный отрезок по первому элементу каждого отрезка.

Возникает проблема нахождения длин отрезков. Можно опираться на длину массива, который необходимо разделить на части. Оптимальнее всего взять за длину отрезка корень из длины массива (например, можно поделить массив из сотни элементов на 10 отрезков равной длины).

В качестве примера дан массив из 20-и элементов [1, 5, 6, 8, 9, 10, 13, 14, 16, 18, 20, 24, 31, 33, 35, 38, 41, 46, 49, 50] и необходимо найти число 46. Необходимо искать в правой половине массива, то есть в подмассиве, который состоит из элементов [20, 24, 31, 33, 35, 38, 41, 46, 49, 50].

Данный подмассив можно разделить на отрезки, длина которых будет равна трём элементам: [20, 24, 31], [33, 35, 38], [41, 46, 49, 50] (в последнем

множестве на один элемент больше, так как количество элементов подмассива не делится на 3). По крайним элементам можно определить, что 46 находится в третьем отрезке.

Далее бинарным поиском можно найти необходимый элемент массива (или убедиться в его отсутствии).

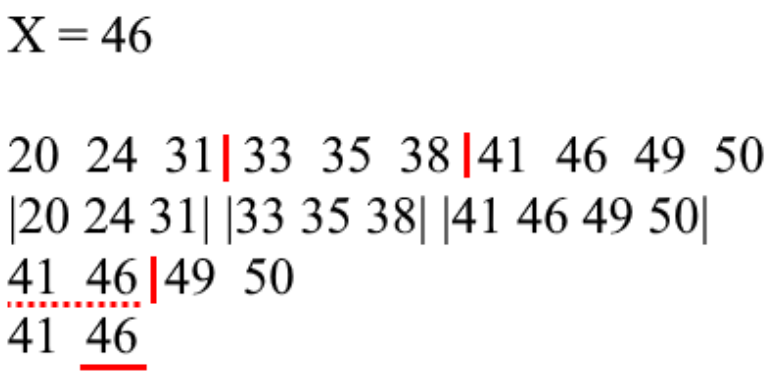

Рисунок 3. Пример деления массива на отрезки и поиска необходимого элемента

Можно сохранять индексы, которые будут указывать на первые элементы отрезков. Таким образом можно также ввести простую структуру, которая будет содержать в себе 2 элемента: начало и конец отрезка (индексы в массиве):

```
struct section {
    int begin;
    int end;
};
```

Но в этом нет необходимости, так как нам изначально известна длина отрезков и по найденным началам каждого отрезка можно без трудностей найти и его конец.

```
if (x < arr[m]) {
    int dist = round(pow(m,0.5f));
    int start = 0;
    while (start < m) {
        ind.push_back(start);
        start += dist;
    }
    vector<int> res_arr;
    for (int i=1;i<ind.size();i++) {
        if (ind[i] == x) return ind[i];
        else if (arr[ind[i]] > x) {
            for (int j=ind[i-1];j<ind[i-
1]+dist;j++) {
```

```
                    res_arr.push_back(arr[j]);
                }
                return binary_search(res_arr,x,ind[i-
1]);
            }
        }
        for (int j=ind[ind.size()-1];j<ind[ind.size()-
1]+dist;j++) {
            res_arr.push_back(arr[j]);
        }
        return binary_search(res_arr,x,m);
    }
```

Данная часть функции определяет в какой части (выше представлен только случай, когда элемент находится слева, для элемента справа алгоритм практически такой же, за исключением изменённых значений в переменных) массива находится искомый элемент и после формируется массив индексов начала каждого отрезка. Опираясь на полученные индексы, можно найти необходимый отрезок через один цикл. Так как массив изначально отсортирован, то гарантировано расположение начал отрезков в порядке возрастания (или убывания), что позволяет без затруднений найти необходимый отрезок.

Эту часть алгоритма можно оптимизировать и не формировать массив индексов начал отрезков. Вместо этого достаточно проходить по массиву с шагом, который будет равен корню из n (n – количество элементов в массиве). В случае если все отрезки начинаются с элементов, меньших искомому, то достаточно проверить последний отрезок, чтобы определить наличие необходимого элемента.

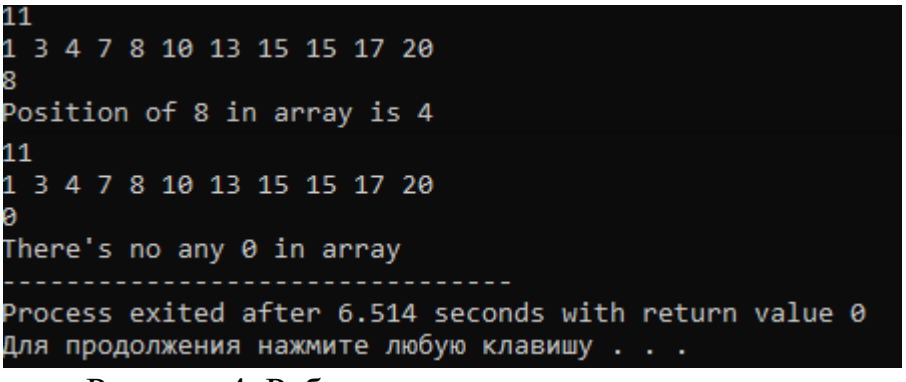

Рисунок 4. Работа оптимизированного поиска

Такой метод деления массива на отрезки называется “Sqrt-декомпозиция”

Sqrt-декомпозиция — это метод, или структура данных, которая позволяет выполнять некоторые типичные операции (суммирование

элементов подмассива, нахождение минимума/максимума и т.д.) за O(sqrt(n)), что значительно быстрее, чем O(n) для тривиального алгоритма.

Простейшая реализация декомпозиции, которая делит массив на части, длина которых равна корню из n выглядит следующим образом:

```
    sq = round(pow(n,0.5f));
    vector<vector<int>> sections; vector<int>
sections_temp;
    for (int i=0;i<n;) {
        sections_temp.clear();
        while (sections_temp.size() != sq) {
            sections_temp.push_back(array[i]);
            i++;
            if (i == n) break;
        }
        sections.push_back(sections_temp);
    }
```

n – количество элементов в массиве.
sq – шаг в декомпозиции.

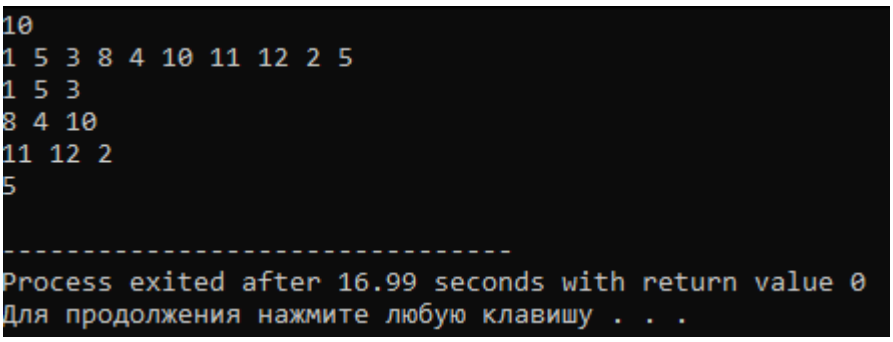

Рисунок 5. Пример работы sqrt-декомпозиции

Таким образом, скорость выполнения бинарного поиска может составить O(log(sqrt(n))) при должной реализации алгоритма.

В статье была описана оптимизация алгоритма бинарного поиска на языке программирования C++. Также описан метод sqrt-декомпозиции и использован при реализации алгоритма бинарного поиска.

Библиографический список

1. Глухов А.М. Сравнительный анализ трудоемкости алгоритмов поиска в программировании // Академия педагогических идей новация. Серия: студенческий научный вестник. 2018. № 5. С. 232-242.
2. 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. С. 116-128 с.
3. Ромм Я.Е., Белоконова С.С. Метод точного информационного поиска на основе разрядного распараллеливания // Современные наукоемкие технологии. 2019. № 7. С. 90-98.
4. Сейдаметова З.С. Особенности изучения сбалансированных бинарных деревьев поиска // Информационно-компьютерные технологии в экономике, образовании и социальной сфере. 2019. № 3 (25). С. 83-93.
5. MAXimal:: algo :: Sqrt-декомпозиция URL: https://e-maxx.ru/algo/sqrt_decomposition. (дата обращения: 22.12.2022).
6. Целочисленный двоичный поиск – Викиконспекты URL: https://neerc.ifmo.ru/wiki/index.php?title=Целочисленный_двоичный_поиск. (дата обращения: 22.12.2022).