Алгоритмы поиска

- Что такое поиск
- Критерии поиска
- Виды поиска

Определение поиска

- Нахождение заданного элемента(ов) в множестве (в нашем случае это будет массив), причем искомые элементы должны обладать определенным свойством.
- Свойство может быть как абсолютным так и относительным.
- Относительное - максимум или минимум во множестве
- Абсолютное - эквивалентное искомому значение.
- Иногда задача может звучать как найти первое или последнее вхождение заданного числа

Критерии поиска

- Рефлексивность (A~A)
- Симметричность (A~B <=> B~A)
- Транзитивность (A~B, B~C <=> A~C)

Виды поиска

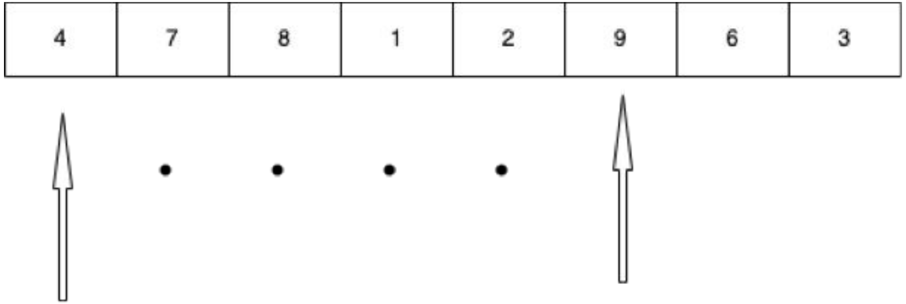
- Линейный поиск
- Бинарный поиск
- Разновидности бинарного поиска
  - Левый и правый бинарные поиски
  - Бинарный поиск по ответу
  - Тернарный поиск
- Экспоненциальный поиск

Линейный (последовательный) поиск

- Сложность по времени в наихудшем случае O(n)
- Затраты памяти O(1)
- Brute-force метод или метод полного перебора

Линейный поиск

- Начиная с первого элемента, последовательно просматриваем весь массив и сравниваем каждое значение с заданным.
- Если значения равны, то возвращаем его номер.
- Недостатки следуют из самого описания - нам необходимо пройтись по всему массиву



Для нахождения искомого элемента проверить надо будет каждый

Когда применим алгоритм

- Если данные не отсортированные, то найти элемент можно только путем последовательного перебора всех элементов.
- Если речь идет о поиске максимума или минимума в массиве

```
function lineSearch(arr []int, target int) int {
    lastIndex = len(arr) - 1
    for i=0; i<lastIndex; i++ {
        if (arr[i] == target) {
            return i
        }
    }
    return -1
}
```

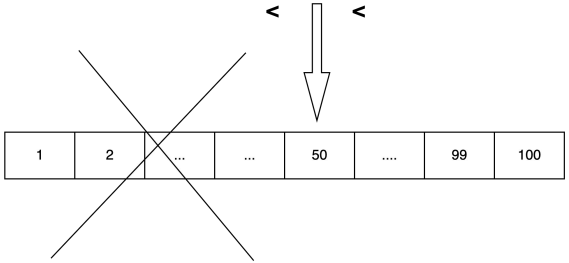
Бинарный поиск

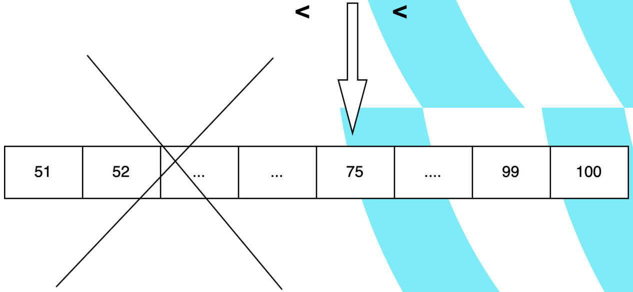
- Сложность по времени в наихудшем случае O(log(n))
- Затраты памяти O(1)

Аналогия из жизни

Игра в угадай число

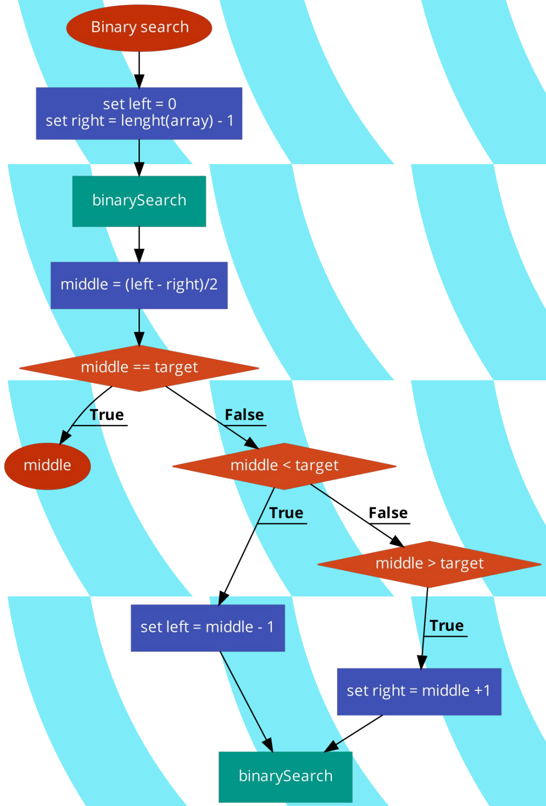
1	2	3	4	5	....	99	100





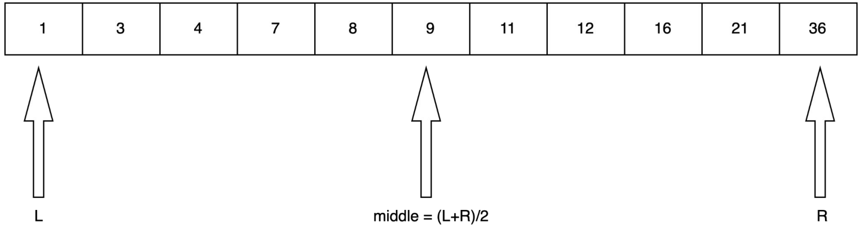
Алгоритм работы поиска

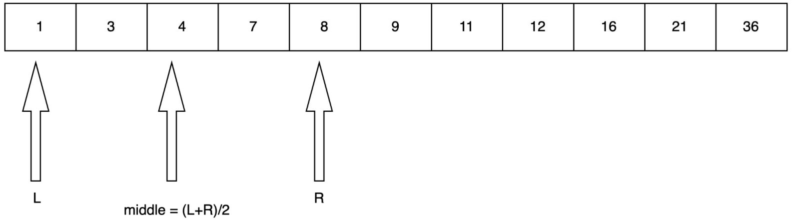
- Массив должен быть отсортирован
- Определяем левую границу в качестве первого элемента массива и правую в качестве последнего элемента
- Делим всю последовательность пополам и находим элемент, находящийся в середине. Сравниваем его с искомым значением
- Если значения равны, то возвращаем индекс элемента
- В случае, если элемент стоящий в середине больше искомого, то обрабатываем левую сторону. В противном случае наоборот.
- Повторяем алгоритм начиная со второго пункта, пока не найдем необходимый элемент или не удостоверимся, что он отсутствует.



Алгоритм работы поиска

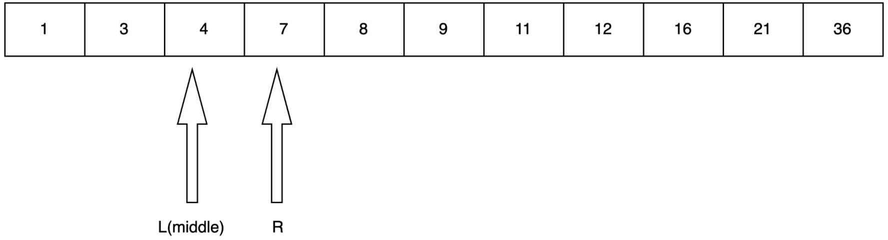
- Допустим, в заданном массиве, нам нужно найти число 7
- Устанавливаем границы отрезка и вычисляем середину
  L=0; R=10; m=(0+10)/2=5;

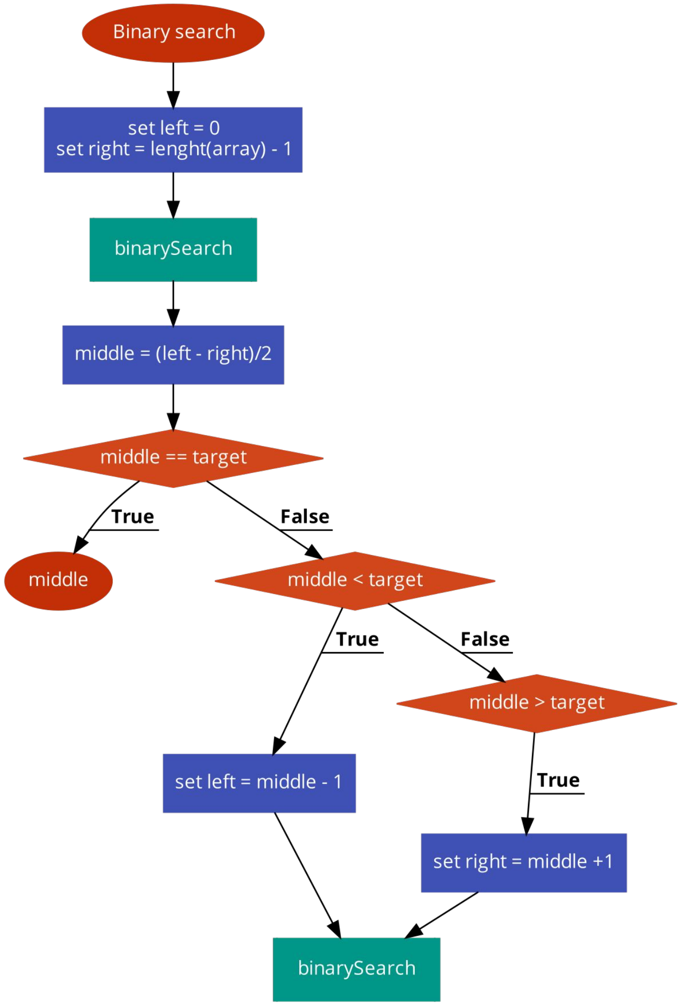




Устанавливаем правую границу
в значении R=middle-1
Почему -1?
Значение правой границы мы уже проверили
Сдвиг вправо даст +1

4<7 Сдвигаемся в право левую границу
Остается последний элемент
Сравниваем его
Возвращаем его индекс если он эквивалентен искомому




Binary search
set left = 0
set right = lenght(array) - 1
binarySearch
middle = (left - right)/2
middle == target
True
False
middle
middle < target
True
False
middle > target
set left = middle - 1
True
set right = middle +1
binarySearch

Рекурсивный подход

- Преимущества: код становится более компактным и читабельным
- Недостатки: требует больше памяти, возможно переполнение стека. При каждом рекурсивном вызове функция добавляется в стек.

```
function binarySearch(data, l, r, target) {
    if l>r {
        return -1
    }
    middle = (l+r)/2
    if (data[middle] == target) {
        return middle
    }
    if (data[middle] > target) {
        # ищем в левой стороне
        # правая граница смещается до middle-1 включительно
        return binarySearch(data, l, middle-1, target)
    } else {
        # ищем в правой стороне
        # левая граница смещается до middle+1 включительно
        return binarySearch(data, middle+1, r, target)
    }
}
```

Итеративный подход

- При каждом рекурсивном вызове мы меняем только левый и правый индексы.
- Это значит, что в зависимости от расположения элемента относительно середины, нам надо в рамках текущей итерации обновить либо левый либо правый индексы.

```
func binarySearch(data, target) int {
    l := 0
    r := len(data) - 1

    if r == 0 || target < data[0] || target > data[r-1] {
        return -1
    }
    for l <= r {
        middle := l + (r-l)/2
        if target == data[middle] {
            return middle
        }
        if target < data[middle] {
            # ищем в левой стороне
            # правая граница смещается на middle-1
            r = middle - 1
        } else {
            # ищем в правой стороне
            # левая граница смещается на middle+1
            l = middle + 1
        }
    }

    return -1 // Not found
}
```

Сложность

Количество элементов перед выполнением поиска = n
После первой итерации = n / 2
После второй = n / 4
...
Итого на i-ом проходе получаем: n / 2^i
На последнем проходе: 1
В итоге получаем формулу
1 = n / 2^i или n = 2^i
Это равносильно записи
i = log(n)
Где n это размер массива

Сравнение количества итераций

- Возьмем для простоты массив из 64 элементов
- 64 → 32 → 16 → 8 → 4 → 2 → 1 — итого 6 раз, что как раз эквивалентно логарифму 64

количество элементов в массиве	Линейный поиск	Бинарный поиск
100	100	7
10000	10000	14
1 000 000	1 000 000	20
1 000 000 000	1 000 000 000	30

Значения в наихудшем случае

Одно НО! Для бинарного поиска необходима сортировка, поэтому необходимо подсчитать когда двоичный поиск точно будет выгоден

- n - количество элементов
- k - количество операций поиска
- n*log(n) - сложность сортировки
- log(n) - сложность поиска

Получаем выражение

n*k >= n*log(n) + k * log(n)

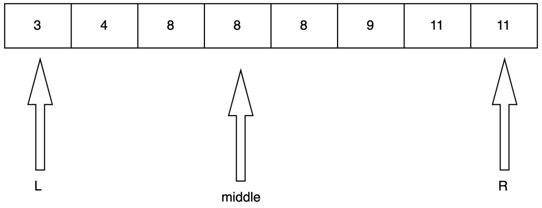
Левый бинарный поиск (поиск первого вхождения)

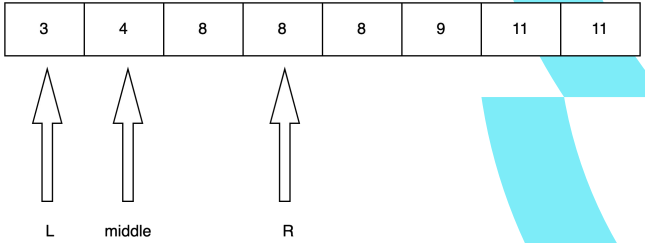
Правила написания

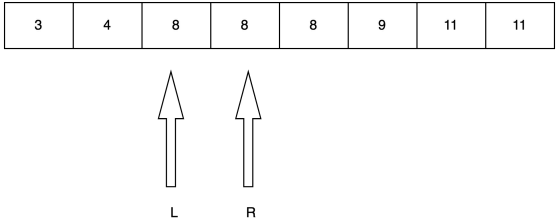
- Цикл продолжается пока не останется два элемента (вместо одного как раньше), то есть for (l + 1 < r) - это нас убережет от бесконечного цикла
- Двигаем правую и левую границы строго на середину без плюс минус единицы
- Если мы ищем первое вхождение (левый бинарный поиск), то есть искомый элемент находится слева, то вначале проверяем левый индекс, а только потом правый
- Если мы ищем последнее вхождение (правый бинарный поиск), то вначале проверяем правый индекс и только потом левый
- Вариантов реализации бинарного поиска может быть множество. Важно придерживаться единого стиля.

Алгоритм работы

- Допустим, нам задан массив и в нем необходимо найти число 8, а точнее самую первую восьмерку
- Находим середину и в случае если найденное значение эквивалентно искомому, то сдвигаем правую границу, строго на индекс middle!
- Почему двигаем в этой ситуации именно правую? Чтобы не потерять 8, которая может оказаться левее.
- Продолжаем поиск пока не останется только левая и правая границы





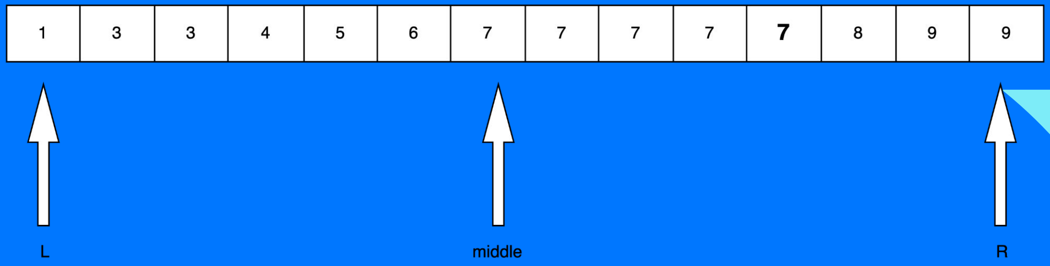


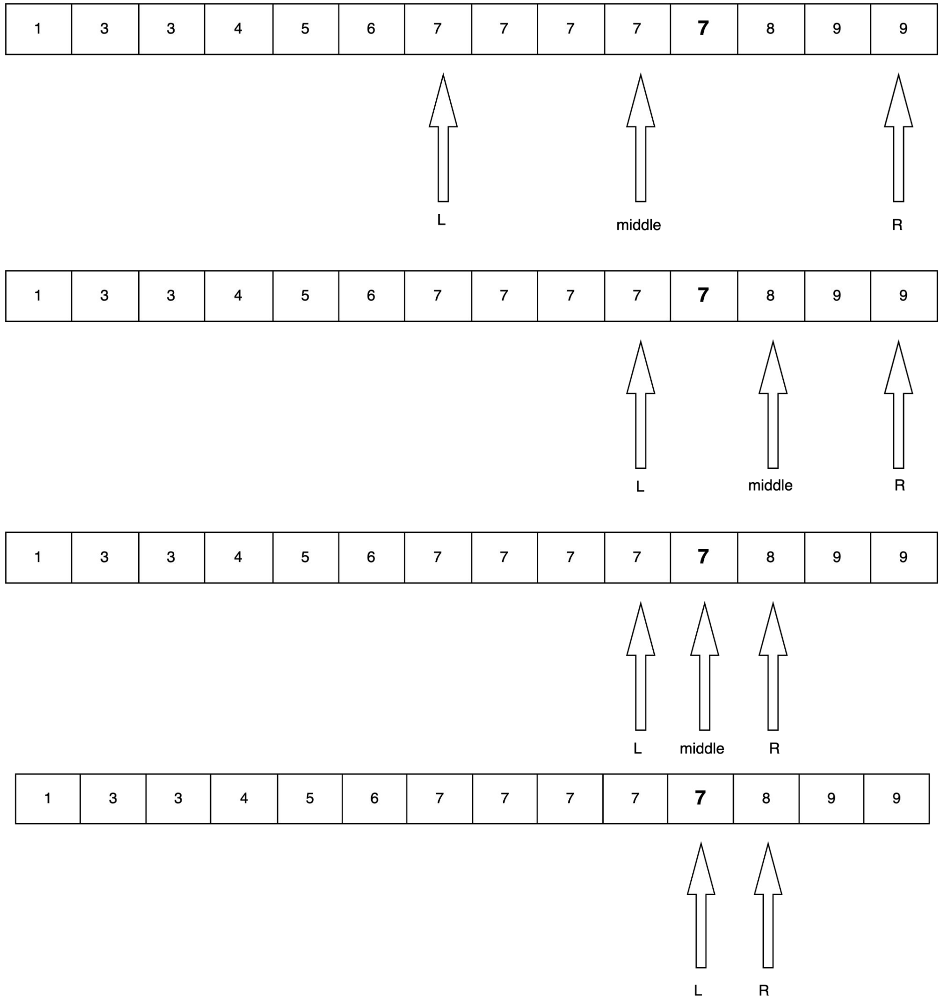
Реализация

```
function leftBinarySearch(data []int, target int) int {
    l := 0
    r := len(data) - 1

    # итерируемся пока не останется два элемента
    for l + 1 < r {
        middle := (l + r) / 2
        if data[middle] < target {
            l = middle
        } else {
            r = middle
        }
    }

    # начинаем проверку с левой границе
    if data[l] == target {
        return l
    }
    if data[r] == target {
        return r
    }
    return -1
}
```

Правый бинарный поиск

- Дан массив и нам необходимо найти число 7, а точнее последнее вхождение
- Здесь, в случае если data[middle] == 7, то мы не заканчиваем поиск, а лишь сдвигаем левую границу (l = middle в коде)



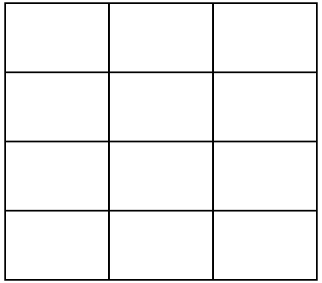

1 3 3 4 5 6 7 7 7 7 7 8 9 9
L
middle
R
1 3 3 4 5 6 7 7 7 7 7 8 9 9
L
middle
R
1 3 3 4 5 6 7 7 7 7 7 8 9 9
L
middle
R
1 3 3 4 5 6 7 7 7 7 7 8 9 9
L
R

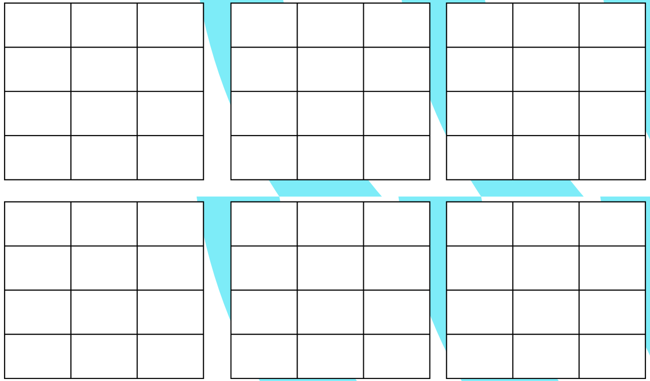
Реализация

```
func rightBinarySearch(data []int, target int) int {
    l = 0
    r = len(data) - 1

    # по-прежнему ищем пока не останется два элемента
    for l+1 < r {
        middle = (l + r) / 2
        if data[middle] <= target {
            l = middle
        } else {
            r = middle
        }
    }

    # теперь в начале проверяем правую границу
    if data[r] == target {
        return r
    }
    if data[l] == target {
        return l
    }

    return -1
}
```

Бинарный поиск по ответу

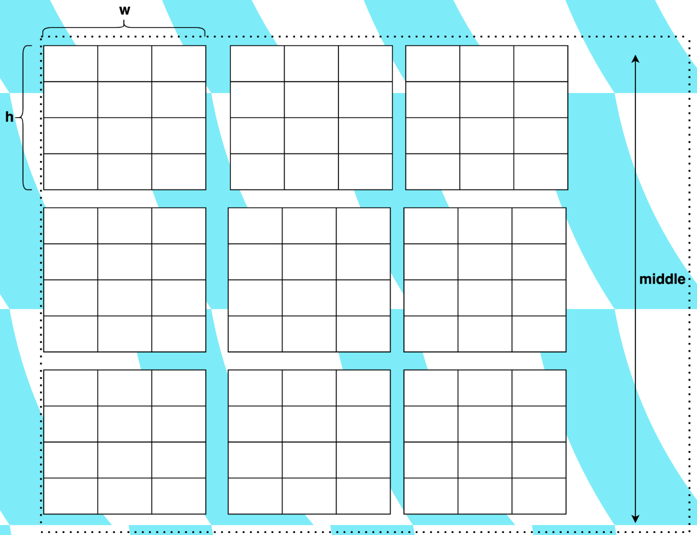
- И так, наступает кульминация нашего изучения бинарного поиска и мы с вами подходим к ключевой главе - бинарный поиск по ответу
- Разберем его на примере культовой задачи про дипломы.
- Есть еще одна не менее культовая задача про коров и стойла, но ее мы разберем на семинаре
- Задача: Петя активный малый и участвует во всех олимпиадах по математике и физике. Накопил кучу дипломов, которые лежали в столе и он не знал что с ними делать. И вот он решил, чтобы они перестали пылиться в столе, лучше чтобы они пылились на стене. Для этого ему хотелось их разместить на квадратной доске. Итак, есть 9 прямоугольных дипломов (3X4), которые надо разместить на квадратной поверхности. Необходимо найти минимальную сторону квадрата для размещения всех дипломов





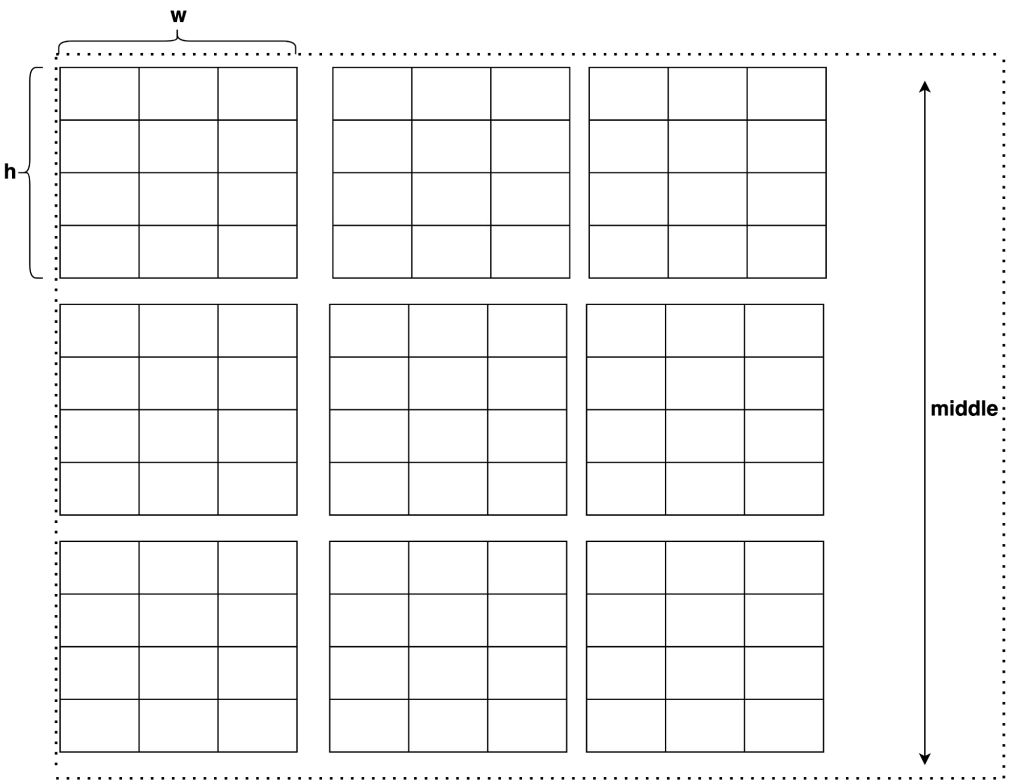
Алгоритм решения

- определяем минимальное значение как самую длинную сторону одного диплома (h). Определяем минимально возможное значение.
- очевидно, что квадрат, сторона которого равна меньшей стороне диплома, точно не подойдет
- определяем максимальное значение
- в цикле сужаем поиск
- как и раньше определяем середину
- на каждой итерации мы будем подсчитывать:
    - количество дипломов в высоту, назовем это строки
    - количество дипломов в ширину, назовем это столбцами
- (middle // h) - количество возможных строк
- (middle // w) - количество возможных столбцов
- перемножая эти два значения мы получим возможное количество дипломов
- возвращаем максимальную сторону



Реализация

```
function binarySearch(w, h, n) {
    # минимальная сторона квадрата
    # равна самой большой стороне диплома
    l = max(w, h)
    # максимально возможно значение
    # это квадрат такой длины, что туда поместятся
    # все дипломы в один ряд
    r = l * n
    # в цикле сужаем поиск
    # l при таких условиях будет последним
    # неподходящим элементом
    # r всегда будет первым подходящим элементом
    while l + 1 < r {
        # ищем среднее значение
        middle = (r + l) // 2

        # высчитываем кол-во дипломов для данного middle
        # умножаем кол-во строк на кол-во столбцов
        res = (middle // w) * (middle // h)
        if res < n {
            l = middle
        } else {
            r = middle
        }
    }

    return r
}
```

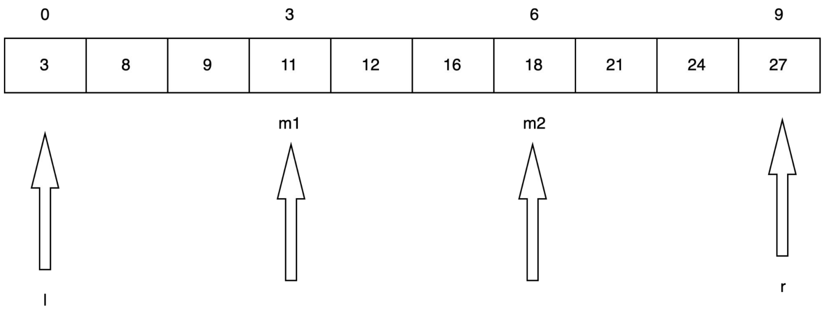


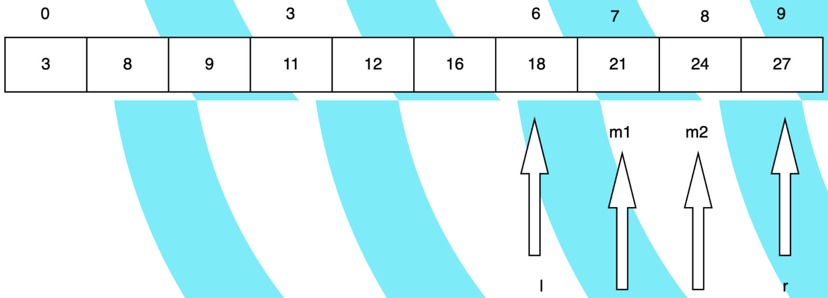
Тернарный поиск

Разбиение последовательности на три части

Описание алгоритма

- так же как и в бинарном поиске разбиваем нашу отсортированную последовательность, но не на 2 а на 3 части следующим образом
- l - левая граница (начинаем с нулевого элемента), r - правая граница (последний элемент массива), m1 = l + (r-l)/3; m2 = r - (r-l)/3
- проверяем m1 и m2 на равенство искомому значению
- если l > r прекращаем поиск
- если m1 меньше искомого значения, а m2 больше, то сдвигаем левые и правую границы на l=m1+1 и r=m2-1
- вычисляем m1 и m2 по уже известным формулам
- повторяем все с 3-го пункта





Рекурсивный подход

```
function ternarySearch(data, target, l, r) {
    if (r <= l) {
       return -1;
    }
    # вычисляем m1 и m2
    m1 = l + (r - l) / 3;
    m2 = r - (r - l) / 3;

    # проверяем на равенство искомому числу
    if (data[m1] == target) {
        return m1;
    }
    if (data[m2] == target) {
        return m2;
    }

    # рекурсивно повторяем поиск сужая границы
    if (target < data[m1]) {
        # target между l и m1
        return ternarySearch(data, target, l, m1 - 1);
    } else if (needle > data[m2]) {
        # target между m2 и r
        return ternarySearch(data, target, m2 + 1, r);
    } else {
        # target между m1 и m2
        return ternarySearch(data, target, m1 + 1, m2 - 1);
    }
}
```

Итерационный подход

```
function ternarySearch(data, target, l, r) {
    while (r >= l) {
        # вычисляем m1 и m2
        m1 = l + (r - l) / 3;
        m2 = r - (r - l) / 3;

        if (data[m1] == target) {
            return m1;
        }
        if (data[m2] == target) {
            return m2;
        }

        # так же как и в рекурсивном подходе,
        # на каждой итерации сужаем диапазон поиска
        if (target < data[m1]) {
            r = m1 - 1;
        } else if (target > data[m2]) {
            l = m2 + 1;
        } else {
            # target находится между m1 и m2
            l = m1 + 1;
            r = m2 - 1;
        }
    }

    # не удалось найти
    return -1;
}
```

Резюме

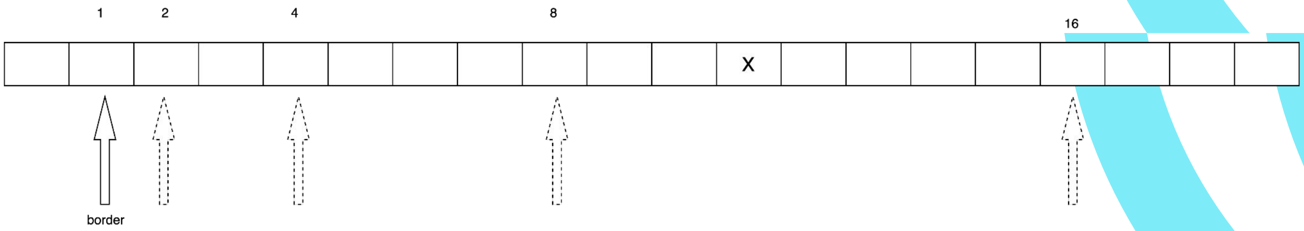
- Частный случай бинарного поиска
- Поиск также выполняется по отсортированной последовательности
- Так как мы на каждой итерации делим массив на 3 это немного уменьшает временную сложность
- Диапазон поиска будет равен n=3^i где i - количество итераций
- Сложность в худшем случае O(log(n)) Но, по основанию 3

Экспоненциальный поиск

(поиск с удвоением)

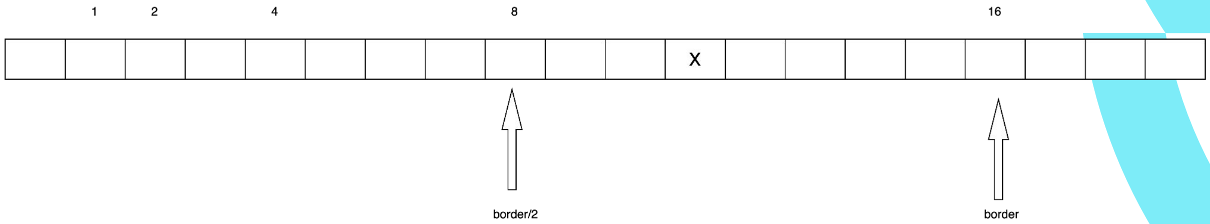
Особенности алгоритма

- Можно использовать только по отношению к отсортированным массивам
- Уточнение диапазона поиска.
- В итоговом диапазоне поиск осуществляется с помощью бинарного поиска
- Сложность такого поиска напрямую зависит от расположения искомого элемента и является O(log(i)) где i - индекс элемента, который нужно найти. Это отличает экспоненциальный поиск от бинарного.

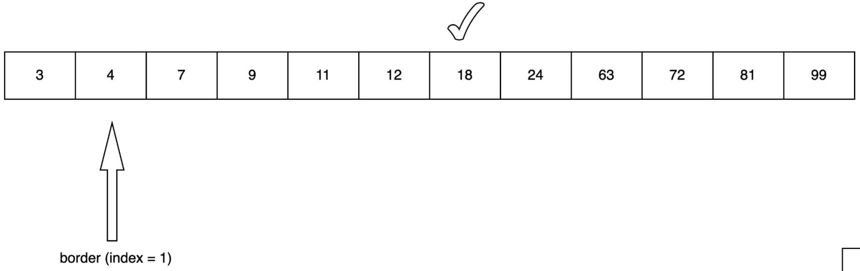


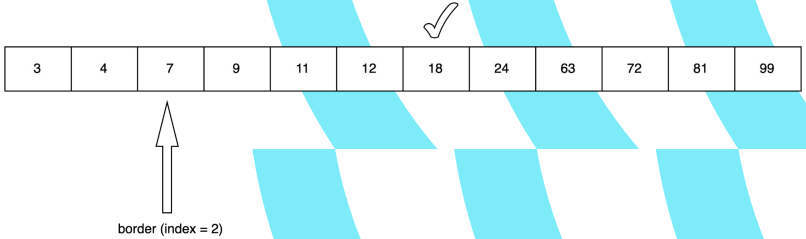
Описание алгоритма

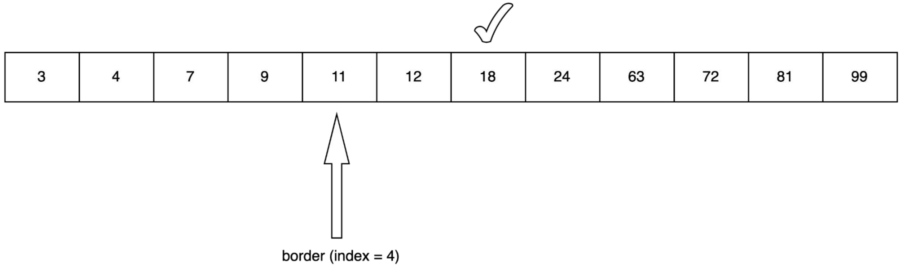
- Для уточнения диапазона введем переменную border, которая на первой итерации будет равна 1
- Сравниваем значение, стоящее под индексом border с искомым.
- Если значение под индексом border меньше того значения что мы ищем, то увеличиваем отрезок в 2 раза.
- Как только border > длины массива, то применяем бинарный поиск к отрезку border/2 - последний элемент массива (длина массива - 1)
- К итоговому диапазону применяем бинарный поиск

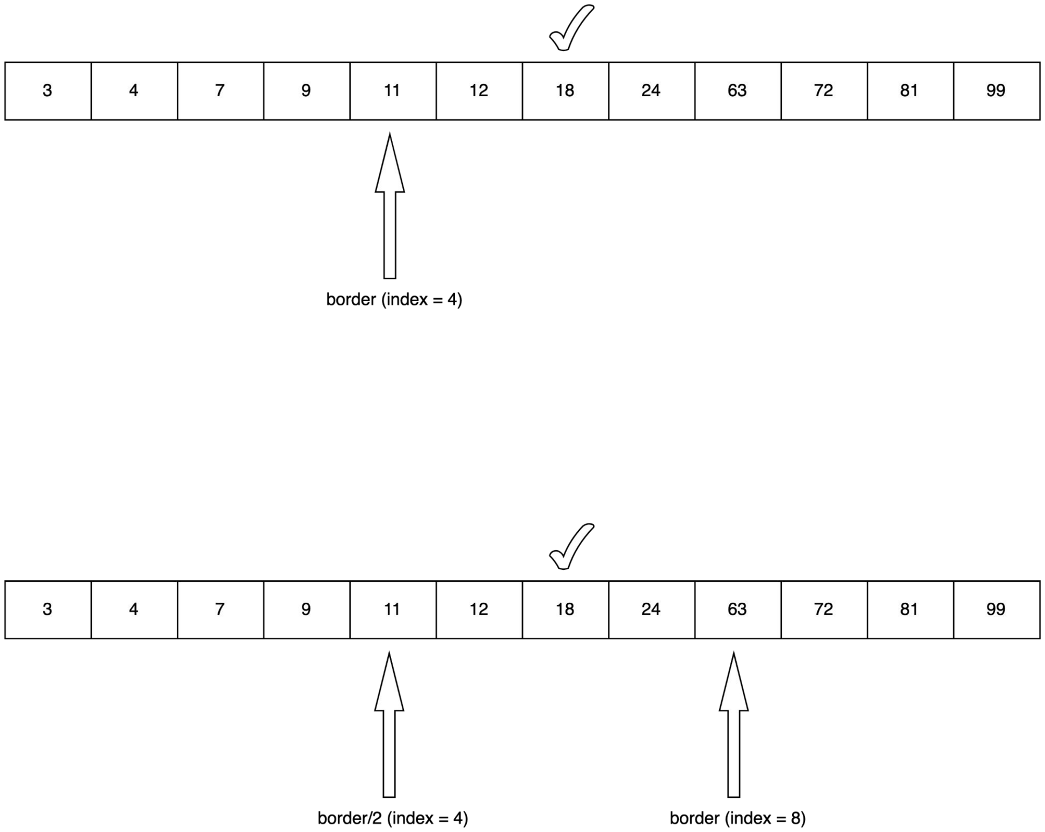


Работа алгоритма







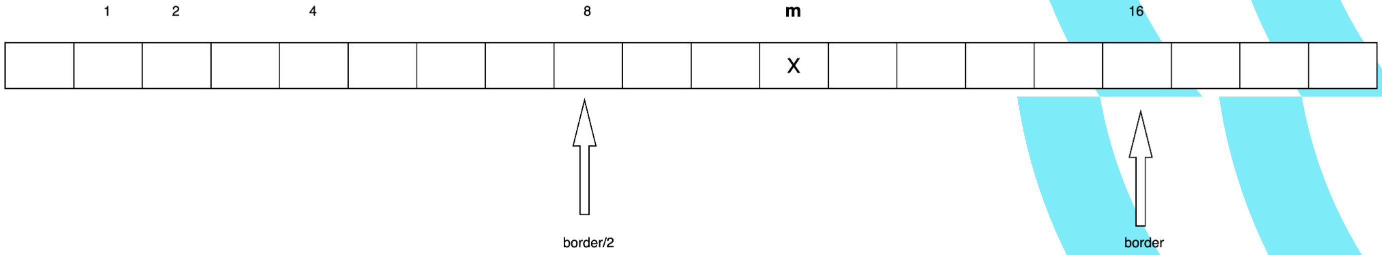

3
4
7
9
11
12
18
24
63
72
81
99
border (index = 4)
3
4
7
9
11
12
18
24
63
72
81
99
border/2 (index = 4)
border (index = 8)

Реализация

```
function exponentialSearch(data, target){
    border = 1
    lastElement = len(data)-1
    while border < lastElement and data[border] < target {
        border = border * 2
        if data[border] == target {
            return border
        }
        if border > lastElement {
            border = lastElement
        }
    }

    return binarySearch(data, target, border/2, border)
}
```

Сложность алгоритма

- Сначала мы проверяем target, сравниваем со значением под первым индексом
- Далее со значением по вторым индексом, 4, 8, 16 и так далее
- Если m это индекс x, который будет находиться в результирующем массиве, значение границ которого это степени двойки, то временная сложность первой фазы будет O(log(m))
- Поиск на втором этапе будет проходить по подмассиву, имеющему размер 2^(1+log(m)) - 2^log(m) = 2*2^log(m) - 2^log(m) = 2^log(m) где m это индекс искомого элемента
- Зная временную сложность бинарного поиска O(log(n)) где n это размер массива, то есть в нашем случае 2^log(m) получаем O(log(2^log(m))) = O(log(m))
- Итоговая сложность с учетом двух этапов поиска O(log(m)) + O(log(m)) что эквивалентно O(log(m))



Резюме

- Экспоненциальный поиск в силу быстрого уточнения финального отрезка полезен для неограниченно больших массивов
- В то же время, исходя из сложности O(log(m)) где m это индекс элемента он быстро ищет элементы находящиеся ближе к началу массива