Алгоритмы поиска

education

Тернарный поиск

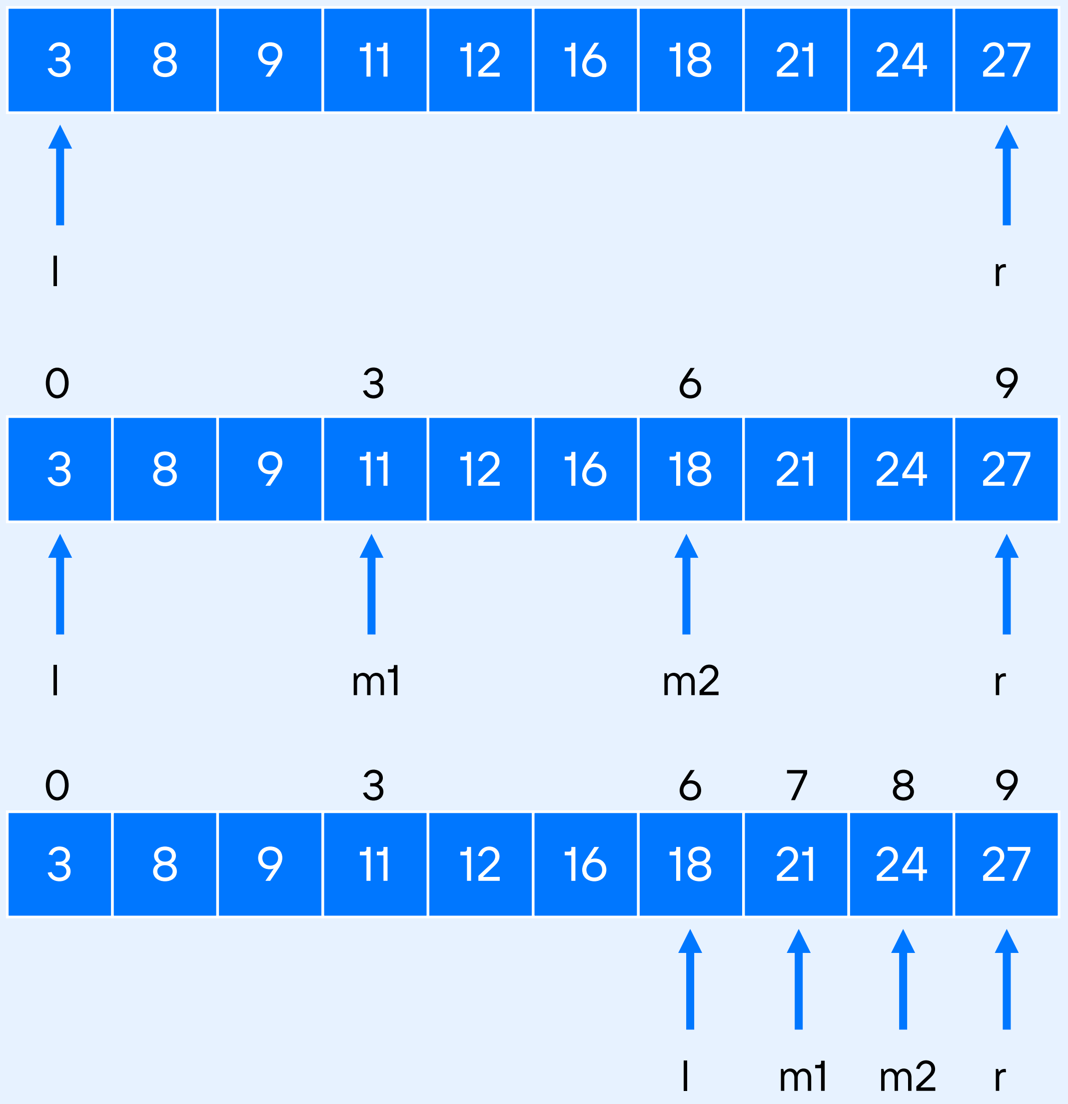
Описание алгоритма

Так же, как и в бинарном поиске, разбиваем нашу отсортированную последовательность,
но не на две, а на три части следующим образом:

- l — левая граница;
- r — правая граница.

$m1 = l + (r - l) \div 3; m2 = r - (r - l) \div 3$

- 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 всё с 3-го пункта.



Описание алгоритма

- Частный случай бинарного поиска.
- Поиск также выполняется по отсортированной последовательности.
- Так как мы на каждой итерации делим массив на три, это немного уменьшает временную сложность.

Диапазон поиска будет равен

$$n = 3 ^ K$$

K — количество итераций

- Сложность в худшем случае O(log(n)), но по основанию 3

Псевдокод

Рекурсивный подход

```
func ternarySearch(data, needle, l, r) {
  if (r >= l) {
    // вычисляем mid1 и mid2
    mid1 = l + (r - l) / 3;
    mid2 = r - (r - l) / 3;

    // проверяем на равенство искомому числу
    if (data[mid1] == needle) {
      return mid1;
    }
    if (data[mid2] == needle) {
      return mid2;
    }

    // рекурсивно повторяем поиск, сужая границы
    if (needle < data[mid1]) {
      // needle между l и mid1
      return ternarySearch(data, needle, l, mid1 - 1);
    } else if (needle > data[mid2]) {
      // needle между mid2 и r
      return ternarySearch(data, needle, mid2 + 1, r);
    } else {
      // needle между mid1 и mid2
      return ternarySearch(data, needle, mid1 + 1, mid2 - 1);
    }
  }

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

Псевдокод

Итерационный подход

```
func ternarySearch(data, needle, l, r) {
   while (r >= l) {
      // вычисляем mid1 и mid2
      int mid1 = l + (r - l) / 3;
      int mid2 = r - (r - l) / 3;

      if (data[mid1] == needle) {
         return mid1;
      }
      if (data[mid2] == needle) {
         return mid2;
      }

      // так же, как и в рекурсивном подходе, на каждой итерации сужаем
диапазон поиска
      if (key < data[mid1]) {
         r = mid1 - 1;
      } else if (key > data[mid2]) {
         l = mid2 + 1;
      } else {
         l = mid1 + 1;
         r = mid2 - 1;
      }
   }

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

education

Экспоненциальный поиск

Поиск с удвоением

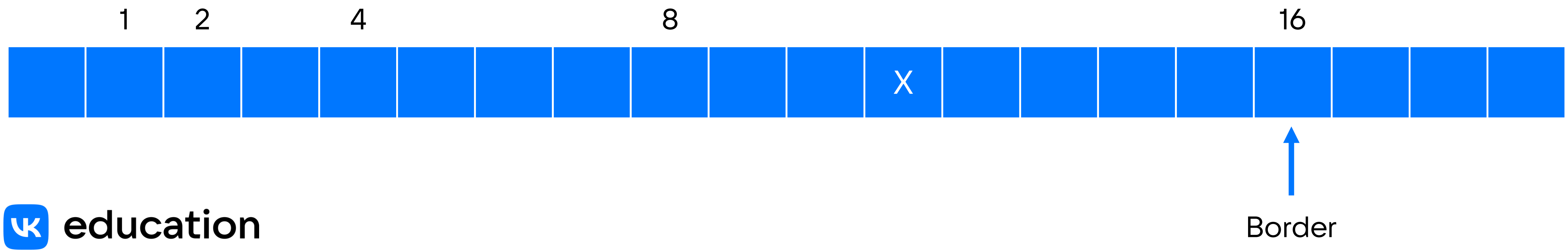
Особенности алгоритма

- 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.

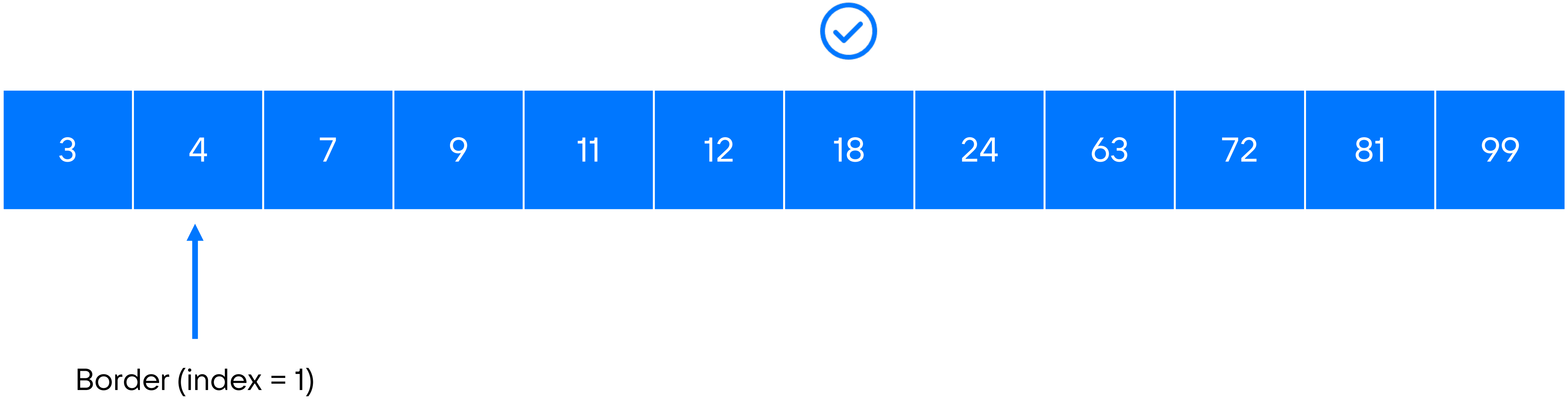
Описание алгоритма

- Для уточнения диапазона введём переменную border, которая на первой итерации будет равна 1.
- Как только border > длины массива, то применяем бинарный поиск к отрезку border/2 — последний элемент массива (длина массива — 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 (границы подмассивов равны степеням 2, отсюда и название).
- В противном случае переходим ко второму пункту и применяем бинарный поиск.



education

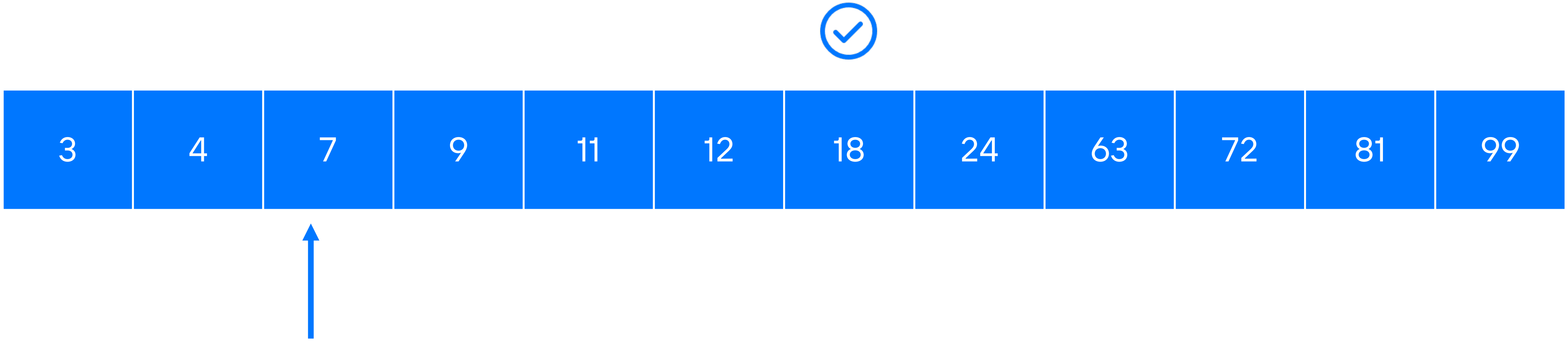
Работа алгоритма наглядно



education

Работа алгоритма наглядно

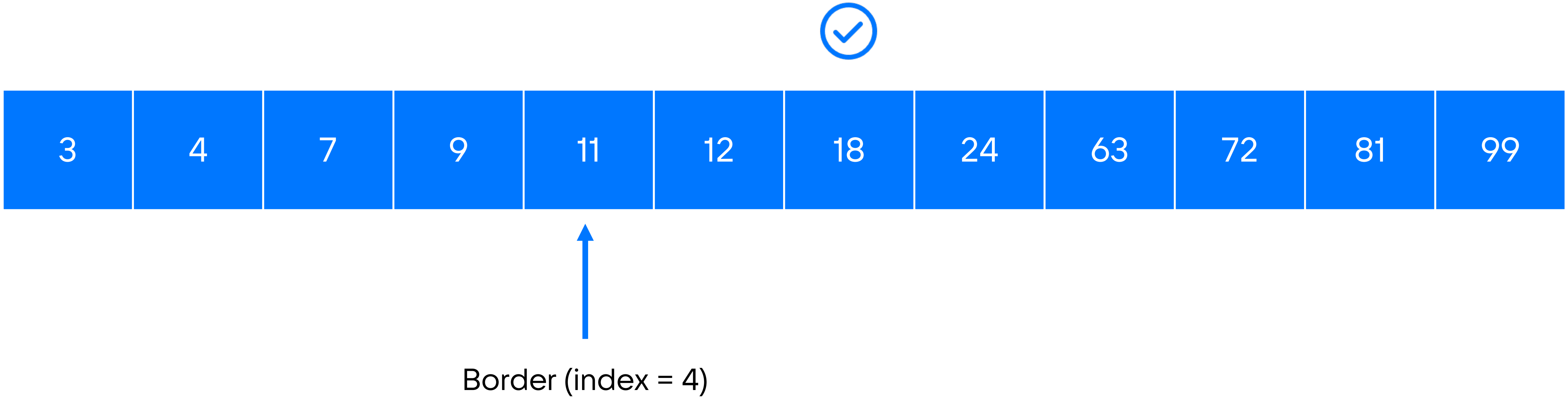
На каждой итерации двигаем border вправо на x2



Border (index = 2)

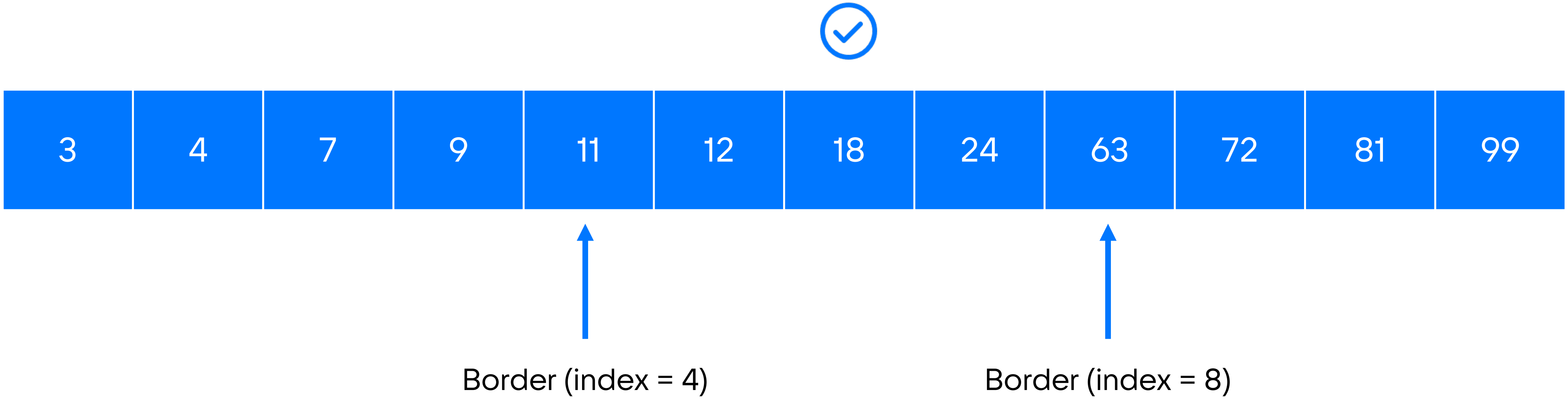
education

Работа алгоритма наглядно



Работа алгоритма наглядно

Определим конечный подмассив,
в котором начнём поиск двоичным методом



Псевдокод

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

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

Вывод сложности алгоритма

- Сначала мы проверяем x, сравниваем со значением под первым индексом.
- Далее со значением по второму индексу, 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 учётом двух этапов поиска — O(log(m)) + O(log(m)), что эквивалентно O(log(m)).

Вывод

1

Экспоненциальный поиск в силу быстрого уточнения финального отрезка полезен для неограниченно больших массивов.

2

В то же время, исходя из сложности O(log(m)), где m — это индекс элемента, он быстро ищет элементы, находящиеся ближе к началу массива.

Поиск Фибоначчи

Последовательность Фибоначчи

1

Бесконечная числовая последовательность, в которой каждое последующее число есть сумма двух предыдущих

2

По первым двум числам можно выстраивать последовательность

Последовательность Фибоначчи

$$F_0 = 0$$

Задаём первое число

$$F_1 = 1$$

Второе число

$$F_n = F_{n-1} + F_{n-2}$$

Получаем последовательность: 0, 1, 1, 2, 3, 5, 8, 13, 21

Алгоритм работы

- Допустим, у нас есть отсортированный массив: 11, 12, 16, 19, 21, 33, 44, 45, 49, 62, 64, 69.
- Находим наименьшее число Фибоначчи. Большее или равное длине массива f2 ($m \geq n$, где n — длина массива).
- Ближайшее большее число в ряду Фибоначчи — 13 (5 + 8); $13 \geq 12$.
- Тогда два предыдущих числа — f0 (m − 2) и f1 (m − 1); f0 = 5 f1 = 8.
- f2 даёт ограничение справа. Введём переменную offset для ограничения слева. Offset в самом начале задаём равным −1.

- Допустим, нам надо найти 62. Вычисляем index — предполагаемая точка; index = min(offset + f0, n − 1) = min(−1 + 5, 11); index4 = 21; 62 > 21.
- Сдвигаем крайнее правое число Фибоначчи влево; получаем f2 = 8, f1 = 5, f0 = 3; сдвигаем offset до нашего индекса: offset = 4; index= min(offset + f0, n −1) = min(4 + 3, 11) = 7; index7 = 45; 45 < 62.
- Сдвигаем крайнее правое число Фибоначчи влево; получаем f2 = 5, f1 = 3, f0 = 2; сдвигаем offset до нашего индекса: offset=7; index = min(offset + f0, n − 1) = min(7 + 2, 11) = 9; index9 = 62; 62 = 62.

Код нашего поиска

```
def fibonacci_search(data, needle):
  size = len(data)
  # задаём первые два числа и по известной формуле получаем третье
  f0 = 0
  f1 = 1
  f2 = f0 + f1

  offset = -1
  # ищем ближайшее максимальное число Ф, которое будет
   # больше либо равно размеру нашего массива
  while f2 < size:
    f0 = f1
    f1 = f2
    f2 = f1 + f0

  # сужаем наш массив поиска
  while f2 > 1:
    # ищем индекс по уже известной нам формуле
    index = min(offset + f0, size - 1)
    # если искомое число больше, чем число под нашим индексом,
     # то сдвигаем offset на место индекса
    if data[index] < needle:
      f2 = f1
      f1 = f0
      f0 = f2 - f1
      offset = index
    # если же число под индексом больше,
     # то сдвигаем наше максимальное число Ф к нулевому
    elif data[index] > needle:
      f2 = f0
      f1 = f1 - f0
      f0 = f2 - f1
    else:
      return index
  if f1 and (data[size - 1] == needle):
    return size - 1
  return None


a = [11, 12, 16, 19, 21, 33, 44, 45, 49, 62, 64, 69]
print(fibonacci_search(a, 33))
print(fibonacci_search(a, 64))
print(fibonacci_search(a, 12))
```

education

Сложность

1

Сложность по времени в наихудшем случае — $O(\log(n))$

2

При поиске Фибоначчи мы используем только сложение и вычитание, что несколько быстрее операции деления, которая используется в бинарном поиске

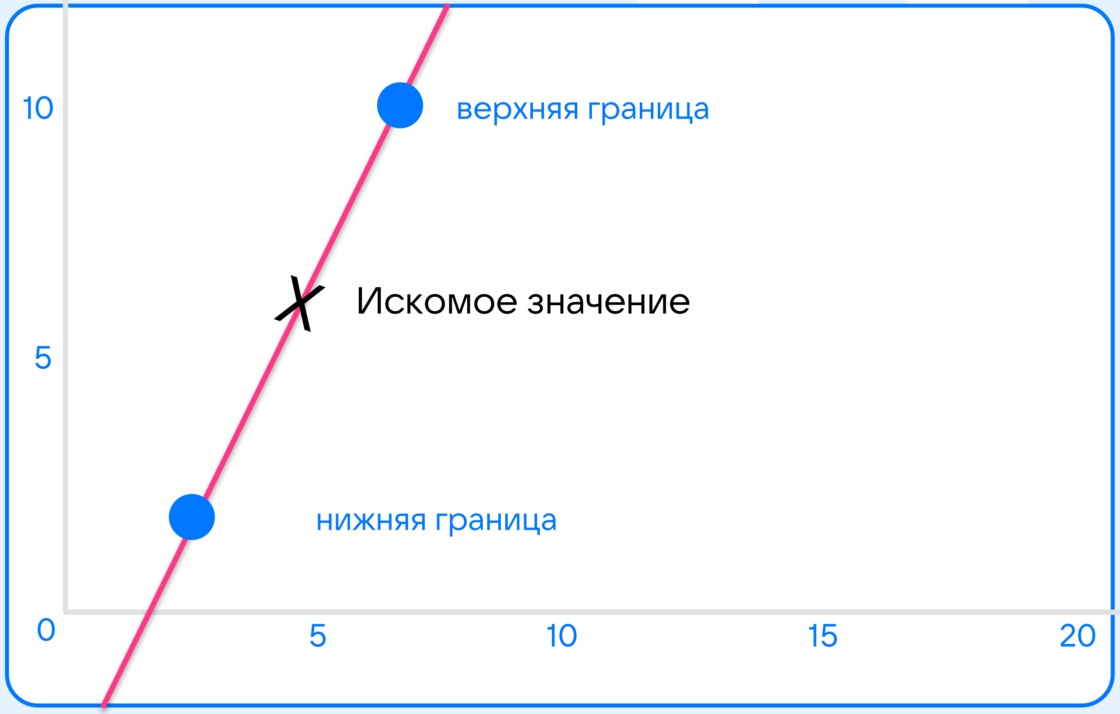
Интерполяционный поиск

О чём речь

- **Интерполяция, интерполирование** — метод нахождения промежуточных значений по имеющемуся набору известных значений.
- Алгоритм, который в качестве вычисления возможного расположения необходимого элемента использует интерполяционную функцию.
- Бинарный поиск всегда уменьшает область поиска в два раза, в то время как интерполяционный может быстрее сужать поиск.
- Сложность в наилучшем случае — **O(log(logn))**.
- Сложность в наихудшем случае — **O(n)**.

О чём речь

- Пытаемся рассчитать по имеющейся интерполяционной функции место нахождения искомого числа по двум известным значениям.
- При этом возможных функций, проходящих через эти точки, может быть сколько угодно много.
- Мы в нашем примере будем использовать **уравнение прямой**, это довольно просто и очень эффективно в таких ситуациях, когда данные **распределены равномерно**.



$y = 2x - 3$

Задача нашей функции — максимально точно предугадать расположение искомого значения

В нашем случае мы будем разбирать линейную интерполяцию

education

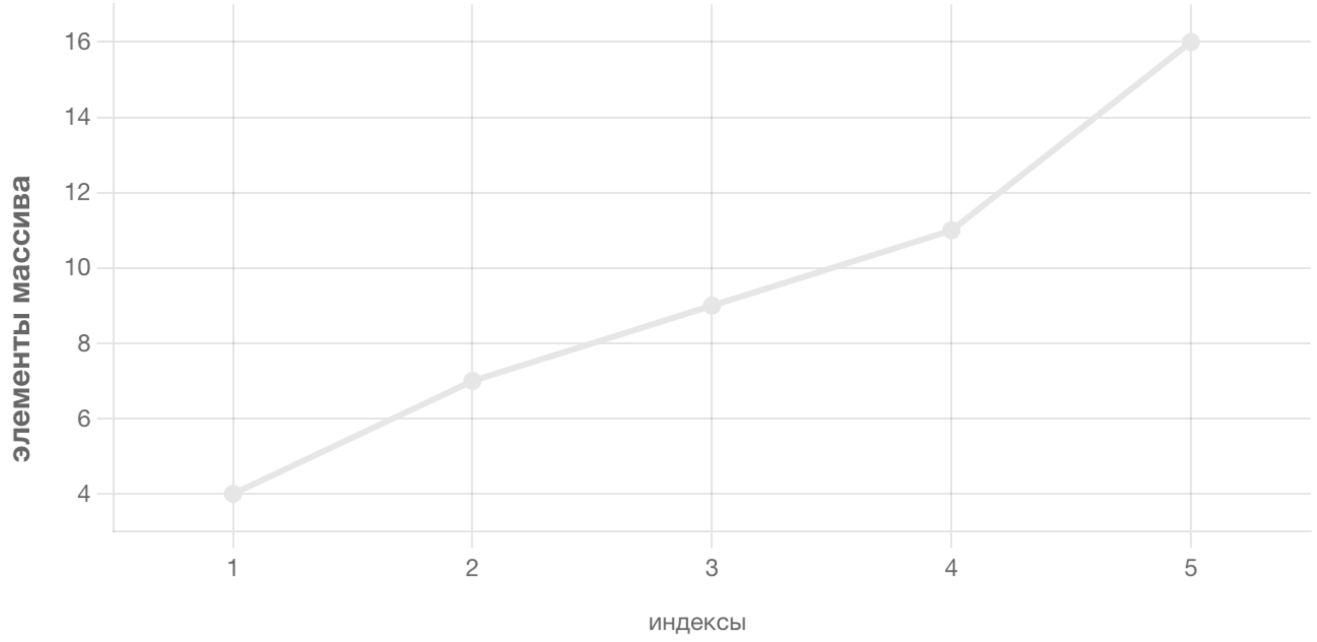
Формула линейной интерполяции

- Мы разберём уравнение прямой, которая проходит через две точки.
- $x0$, $y0$ $x1$, $y1$ — координаты наших точек, которые нам известны и по которым мы будем строить наш график.
- Уравнение прямой по двум точкам: $(x - x0) \div (x1 - x0) = (y - y0) \div (y1 - y0)$, где x и y — это наша предполагаемая точка, которую нам надо найти.
- Формула для поиска необходимой точки будет иметь вид: $y = (x - x0)$ $(y1 - y0) \div (x0 - x1) + y0$.
- Теперь вопрос! А как сопоставить график с массивом? Ведь у нас нет никаких координат! Для использования этой формулы просто подставим на ось x индексы, а на ось y значения элементов массива.

Формула в нашем случае будет иметь вид

$$lowEnd + \frac{(highEnd - lowEnd) \times (item - data[lowEnd])}{data[highEnd] - data[lowEnd]}$$

Index = left + ((right − left) × (needle − array[left]) ÷ array[right] − array[left])



Алгоритм поиска

- В качестве известных «координат» мы возьмём левое и правое значение нашего массива.
- По уже известной формуле линейной интерполяции получаем предполагаемое место нахождения нашего элемента.
- В случае, если он равен заданному, возвращаем его и заканчиваем работу.
- Если найденное значение больше, сдвигаем правую границу к нашему индексу, а точнее — **index – 1**.
- Соответственно, если меньше, то сдвигаем левую границу к **index + 1**.
- Повторяем эти операции до тех пор, пока не найдём нужное нам значение или не удостоверимся, что его нет в заданном массиве.

Код нашего поиска

```
def interpolation_search(data, needle):
  left = 0
  right = len(data) - 1
  # выполняем наш поиск, пока заданное число находится в границах
массива,
  # точнее, пока граница не сведётся к нулю
  while data[left] < needle < data[right]:
    if data[left] == data[right]:
      break
    index = left + (right - left) * (needle - data[left]) // (data[right] - data[left])
    # в зависимости от того, больше ли искомое число или меньше,
    # сдвигаем соответствующим образом наши границы
    if data[index] > needle:
      right = index - 1
    elif data[index] < needle:
      left = index + 1
    else:
      return index
  # после выхода из цикла остаётся проверить значения границ
  if data[left] == needle:
    return left
  if data[right] == needle:
    return right
  # если не нашли, то возвращаем минус единицу
  return -1
```

education

Будем
ВКонтакте!