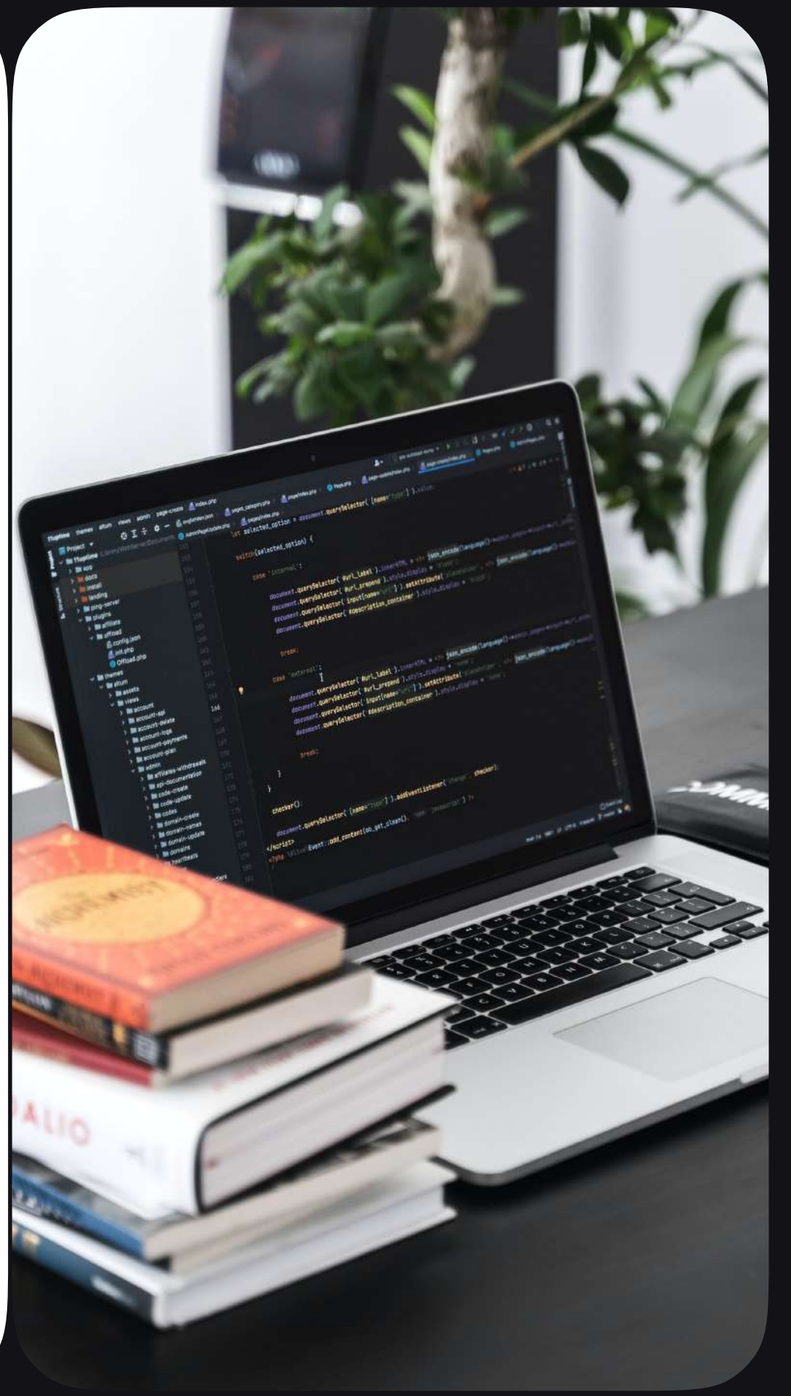
Модуль 2 Занятие 11

Алгоритмы поиска и сортировки



Сортировки

Сортировка пузырьком

Сортировка выбором

Сортировка вставками

Сортировка слиянием

Быстрая сортировка

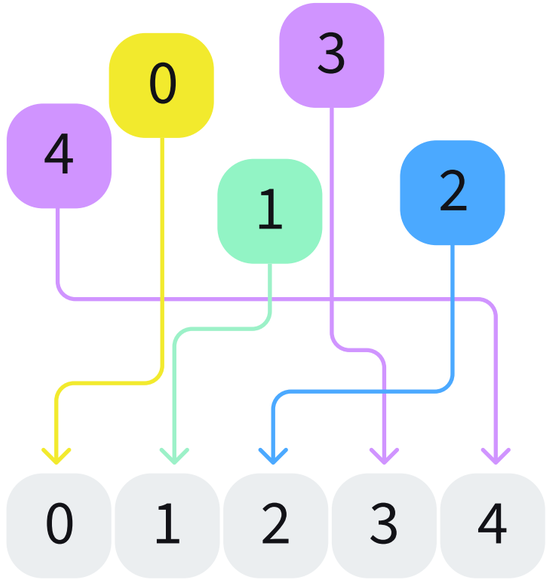
Линейный поиск

Двоичный поиск

Сортировка

Сортировка массива — это упорядочивание элементов массива в определенном порядке.

Порядок сортировки может быть любым, для чисел обычно говорят о сортировке по возрастанию (неубыванию, если массив содержит повторяющиеся элементы) или убыванию (невозрастанию).



Виды сортировок

Существует множество алгоритмов сортировок, они отличаются временем выполнения и эффективностью использования памяти:

Медленные сортировки:

- Сортировка пузырьком (bubble sort)
- Сортировка выбором (selection sort)
- Сортировка вставками (insertion sort)

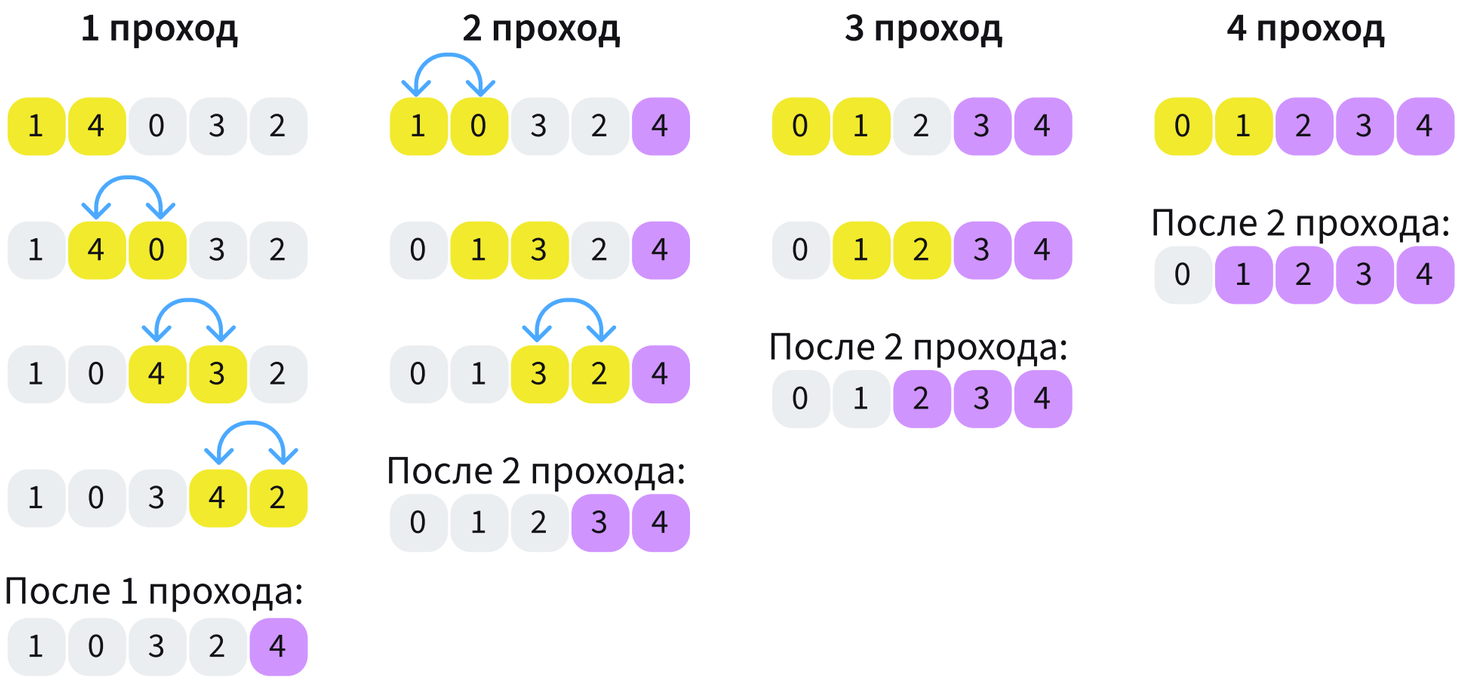
Быстрые сортировки:

- Сортировка слиянием (merge sort)
- Быстрая сортировка (quick sort)

Разные алгоритмы также могут использовать дополнительную память. Алгоритм сортировки не использующий дополнительную память, называют in-place сортировка (сортировка на месте).

Сортировка пузырьком

Сортировка пузырьком (bubble sort) — основана на повторяющихся проходах по массиву и попарном сравнении двух рядом стоящих элементов. Наибольший элемент за один такой проход «всплывает» в конец массива, как пузырек воздуха, — отсюда и произошло название алгоритма.



Рассмотрим сортировку массива [1, 4, 0, 3, 2] по возрастанию значений.

Желтым цветом выделена пара сравниваемых элементов. Фиолетовым цветом выделены отсортированные элементы.

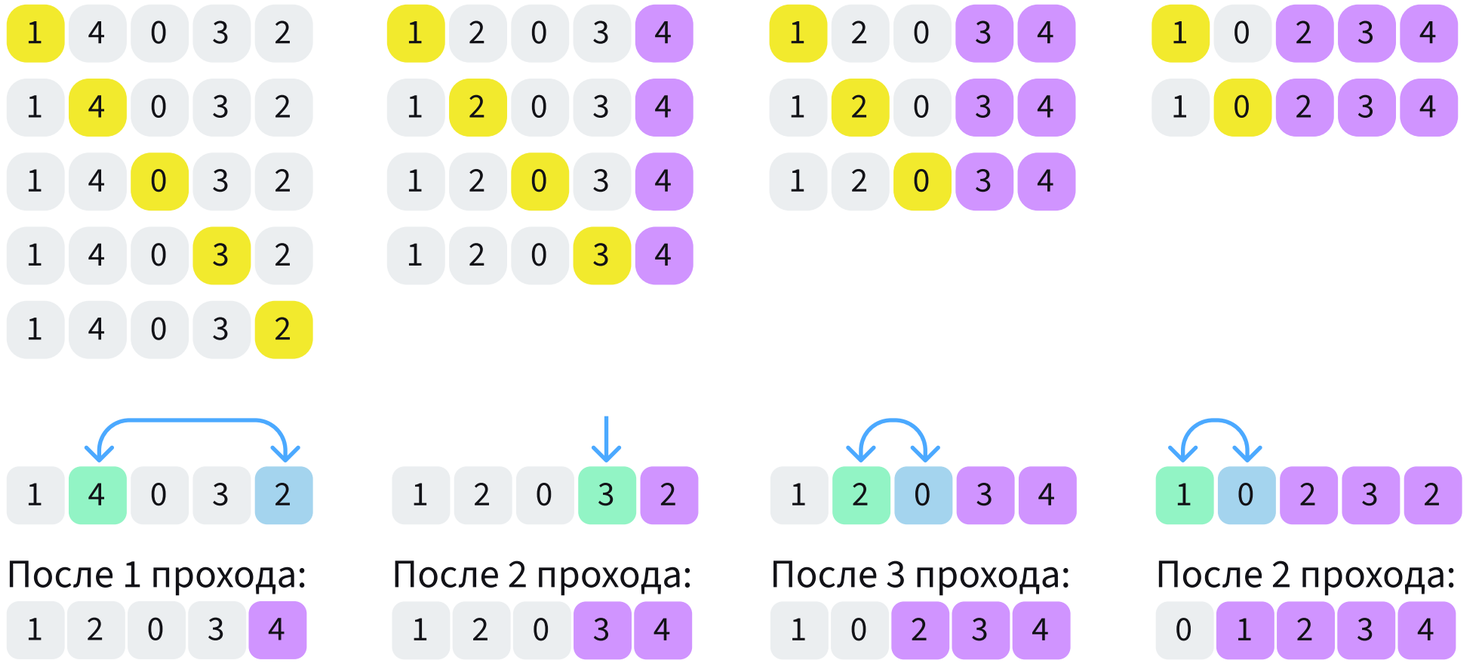
Программа сортировки пузырьком

```
array = [1, 4, 0, 3, 2]
n = len(array)

for i in range(n - 1):
    for j in range(n - i - 1):
        if array[j] > array[j + 1]:
            array[j], array[j + 1] = array[j + 1], array[j]

print(array)
```

Сортировку пузырьком можно ускорить. Если за один проход, ни одна пара элементов не поменялась местами, то это значит, что элементы уже отсортированы и можно досрочно завершить цикл, воспользовавшись командой break.

Сортировка выбором

Сортировка выбором (selection sort) — также основана на повторяющихся проходах по массиву, но в отличие от сортировки пузырьком на каждом проходе происходит всего один обмен значений. За каждый проход по массиву ищется максимальный элемент и помещается в нужную позицию.



Также рассмотрим сортировку массива [1, 4, 0, 3, 2] по возрастанию значений.

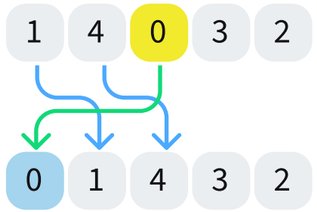
Желтым цветом выделен текущий рассматриваемый элемент, зеленым цветом — максимальный, синим цветом показана позиция, с которой происходит обмен. Фиолетовым цветом выделены отсортированные элементы.

Сортировка вставками

Сортировка вставками (insertions sort) — основана на том, что элементы рассматриваются по порядку, по одному, и если в процессе находится неупорядоченный элемент, то он перемещается в подходящее место слева, сдвигая следующие элементы вправо.

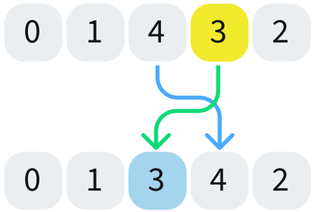
1 проход

0 < 4



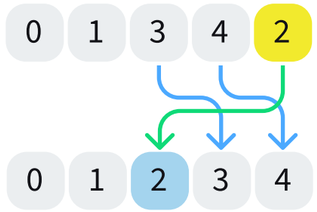
Ищем позицию для 0 и сдвигаем элементы правее этой позиции вправо.

3 < 4



Ищем позицию для 3 и сдвигаем элементы правее этой позиции вправо.

2 < 4



Ищем позицию для 2 и сдвигаем элементы правее этой позиции вправо.

Также рассмотрим сортировку массива [1, 4, 0, 3, 2] по возрастанию значений.

Желтым цветом выделен текущий рассматриваемый элемент, синим цветом показана подходящая позиция для вставки.

Программа сортировки вставками

```
array = [1, 4, 0, 3, 2]
n = len(array)

for i in range(1, n):
    x = array[i]
    j = i
    while j > 0 and x < array[j - 1]:
        array[j] = array[j - 1]
        j -= 1
array[j] = x

print(array)
```

Сортировка вставками эффективна, если список уже частично отсортирован и элементов массива немного.

Сортировка слиянием

Сортировка слиянием (merge sort) — основана на принципе «разделяй и властвуй», это значит, что исходная задача разбивается на подзадачи меньшего размера, которые в свою очередь еще разбиваются на подзадачи, пока не станут элементарными, а затем решения этих подзадач комбинируются, и получается решение исходной задачи.

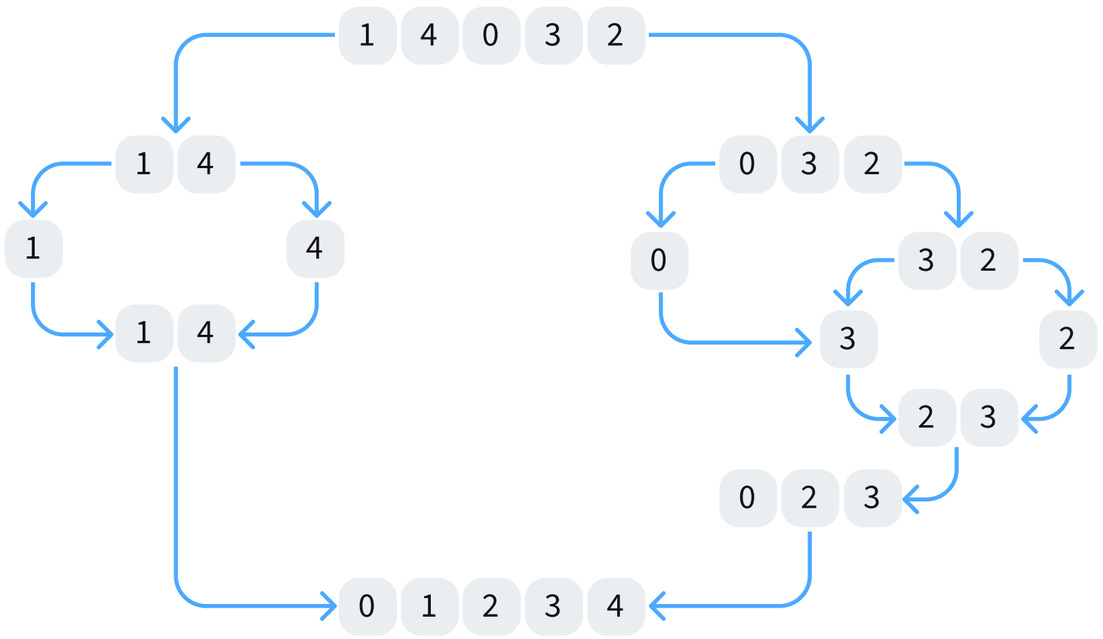
Сортировку слиянием можно описать следующим образом:

1. Массив разбивается на две части.
2. Если в получившейся части массива только один элемент, то он отсортирован, а иначе для него рекурсивно запускается сортировка по этому же алгоритму, начиная с пункта 1.
3. После сортировки двух частей массива они объединяются (сливаются) в один отсортированный массив.

Объединение отсортированных массивов

На каждом шаге выбираем из какого массива забрать элемент и добавить в новый массив. Желтым цветом показаны сравниваемые элементы.

Сортировка слиянием



Сортировка массива [1, 4, 0, 3, 2]
по возрастанию значений сортировкой слиянием.

Массив разбивается на две части.
Каждая часть рекурсивно сортируется.
После этого отсортированные части объединяются в один отсортированный массив.

Программа сортировки слиянием

```
def merge_sort(array):
    print(array)
    if len(array) == 1: # если в массиве 1 элемент, то он отсортирован
        return array
    left = merge_sort(array[: len(array) // 2]) # запускаем сортировку левой части
    right = merge_sort(array[len(array) // 2:]) # запускаем сортировку правой части
    result = []
    l, r = 0, 0
    while l < len(left) and r < len(right): # объединяем массивы в один
        if left[l] <= right[r]: # выбираем из какой части забрать элемент
            result.append(left[l])
            l += 1
        else:
            result.append(right[r])
            r += 1
    if l >= len(left): # прицепляем оставшуюся часть оставшегося массива
        result += right[r:]
    else:
        result += left[l:]
    return result


array = [1, 4, 0, 3, 2]
print(merge_sort(array))
```

Быстрая сортировка

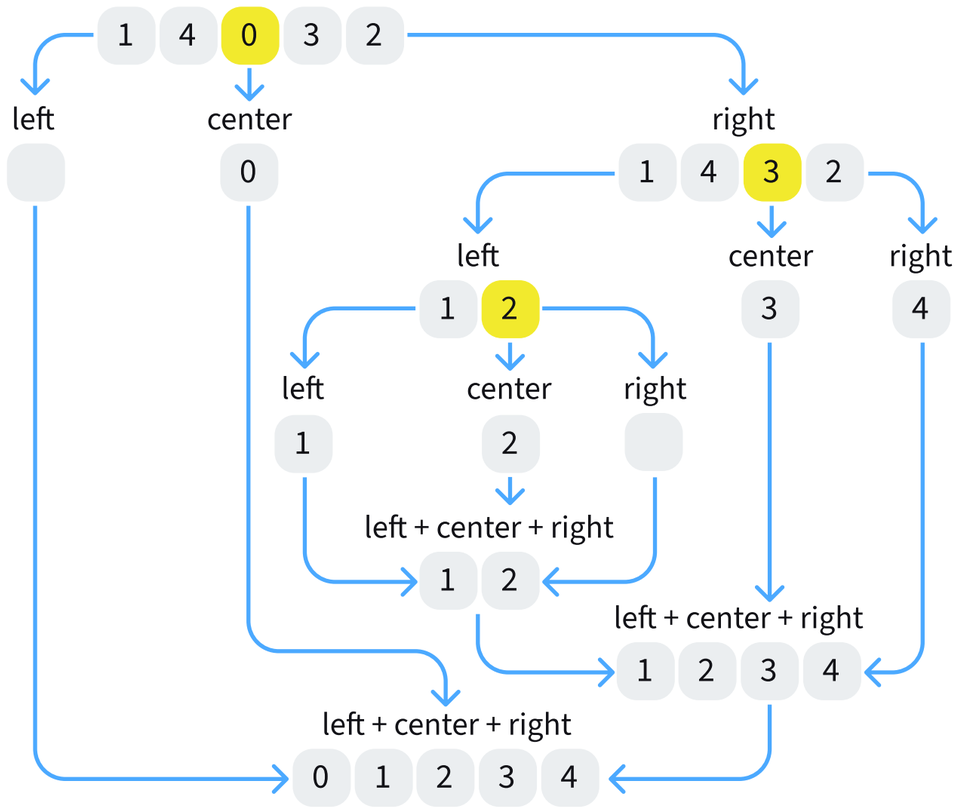
Быстрая сортировка (quick sort) — один из самых быстрых и используемых алгоритмов сортировки. Быстрая сортировка также основана на принципе «разделяй и властвуй».

Быструю сортировку можно описать следующим образом:

1. Выбрать некоторый опорный элемент массива.
2. Делим массив на две части, левую и правую: в левый массив переносим элементы меньше опорного, в правый массив — больше опорного, посередине остаются элементы — равные опорному.
3. Если в левой или правой части массива один или нет элементов, то сортировка не требуется, а иначе для каждой части рекурсивно запускается сортировка, начиная с шага 1.
4. В результате получим отсортированные массивы, достаточно объединить их в один.

Для выбора опорного элемента существуют разные подходы, влияющие на производительность алгоритма.

Быстрая сортировка



В качестве опорного элемента будем выбирать средний элемент массива. Делим массив на две части: левую и правую. В левый массив переносим элементы меньше опорного, в правый массив — больше опорного, посередине остаются элементы равные опорному, и для каждой части рекурсивно запускаем быструю сортировку.

Программа быстрой сортировки

```
def quick_sort(array):
    if len(array) < 1: # если в массиве 1 элемент или нет элементов
        return array
    pivot = array[(0 + len(array)) // 2] # находим опорный элемент
    left, center, right = [], [], []
    for el in array: # заполняем левую и правую часть
        if el < pivot:
            left.append(el)
        elif el > pivot:
            right.append(el)
        else:
            center.append(el)
    return quick_sort(left) + center + quick_sort(right) # сортируем левую и правую части и объединяем результат


array = [1, 4, 0, 3, 2]
print(quick_sort(array))
```

У нашей реализации есть существенный недостаток — программа использует дополнительную память и на практике такой алгоритм не используется.

Использования дополнительной памяти можно избежать, для этого надо изменить код программы так, чтобы выполнялась in-place сортировка.

Линейный поиск

Пусть нам дан массив целых чисел размером n.
И нам необходимо найти индекс элемента равного числу x,
а если такого числа нет, то вывести –1.

Наивное решение этой задачи: будем рассматривать каждый элемент и сравнивать его с x, если элемент равен x, то выводим его индекс. Если мы рассмотрели все элементы, а x не нашли то выводим –1 — искомого элемента нет в массиве.

Программа линейного поиска

```
array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
n = len(array)
x = 8

for i in range(n):
    if x == array[i]:
        print(i)
        break
else:
    print(-1)
```

Нетрудно догадаться, что в худшем случае (когда искомого элемента нет в списке или искомый элемент стоит где-то в конце) наше решение выполнит столько сравнений, сколько элементов в списке.

Говорят,что время работы линейно зависит от размера входных данных, поэтому такой поиск называется линейным поиском.

Двоичный поиск

Рассмотрим другой вариант решения задачи поиска элемента равного числу x в **отсортированном** списке размером n с помощью **двоичного поиска**.

Двоичный поиск (binary search) — основан на том, что на каждом шаге исходный массив разделяется на две половины и затем выбирается та половина, в которой может находиться искомый элемент x. Затем такая половина опять разбивается на две половины и поиск продолжается до тех пор, пока искомый элемент не будет найден. Или не найден.

Важным условием использования двоичного поиска является то, что данные должны быть отсортированы. Если линейный поиск можно использовать на неупорядоченных данных, то для использования двоичного поиска данные, либо должны быть отсортированы изначально, либо их придется предварительно отсортировать. Зато двоичный поиск работает намного быстрее.

Двоичный поиск

Будем искать x = 8.
array[mid] < x,
продолжаем поиск
в правой половине:
left = mid + 1

array[mid] > x,
продолжаем поиск
в левой половине:
right = mid

array[mid] = x,
элемент найден

Переменные left и right ограничивают область поиска искомого элемента x. Изначально left = 0, а right = n, такой выбор сделан для того, чтобы array[left] был всегда меньше или равен x, а array[right] всегда больше x.

Индекс среднего элемента вычисляется так:

mid = (left + right) // 2

Программа двоичного поиска

```
array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
n = len(array)
x = 8

left = 0
right = n
while left < right:
    mid = (left + right) // 2
    if array[mid] == x:
        print(mid)
        break
    elif array[mid] < x:
        left = mid + 1
    else:
        right = mid
else:
    print(-1)
```

Левый и правый двоичный поиск

В нашей реализации двоичного поиска есть проблема: что если в массиве будут содержаться повторяющиеся элементы равные **x**? Наш поиск вернет любой из индексов элемента равного **x**. В случае, если необходимо проверить просто факт наличия такого элемента, то это решение нас устроит.

Но если нужно найти первый такой индекс или последний, то в этом случае говорят про левый двоичный поиск — найти самый левый, т.е. с наименьшим индексом элемент равный **x** и правый двоичный поиск — найти самый правый, т.е. с наибольшим индексом элемент равный **x**.