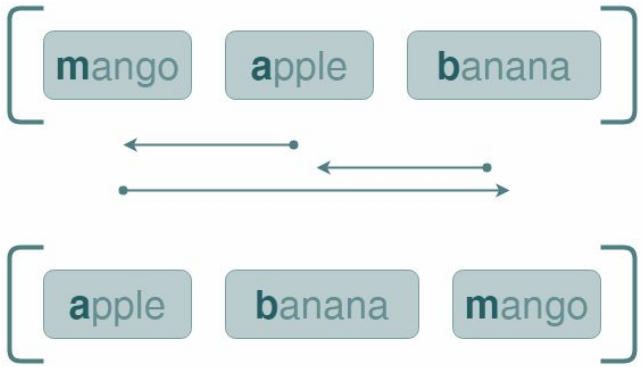
МЕТОДЫ СОРТИРОВКИ



АЛГОРИТМ СОРТИРОВКИ ЗАДАЕТ СПОСОБ РАСПОЛОЖЕНИЯ ДАННЫХ В ОПРЕДЕЛЕННОМ ПОРЯДКЕ. НАИБОЛЕЕ РАСПРОСТРАНЕННЫЕ ПОРЯДКИ ВЫПОЛНЯЮТСЯ В ЦИФРОВОМ ИЛИ ЛЕКСИКОГРАФИЧЕСКОМ ПОРЯДКЕ.

ВАЖНОСТЬ СОРТИРОВКИ ЗАКЛЮЧАЕТСЯ В ТОМ, ЧТО ПОИСК ДАННЫХ МОЖЕТ БЫТЬ ОПТИМИЗИРОВАН ДО ОЧЕНЬ ВЫСОКОГО УРОВНЯ, ЕСЛИ ДАННЫЕ ХРАНЯТСЯ СОРТИРОВАННЫМ ОБРАЗОМ. СОРТИРОВКА ТАКЖЕ ИСПОЛЬЗУЕТСЯ ДЛЯ ПРЕДСТАВЛЕНИЯ ДАННЫХ В БОЛЕЕ ЧИТАЕМОМ ФОРМАТЕ.

НЕКОТОРЫЕ ПРИМЕРЫ СОРТИРОВКИ В РЕАЛЬНЫХ СЦЕНАРИЯХ —

- ТЕЛЕФОННЫЙ СПРАВОЧНИК — ТЕЛЕФОННЫЙ СПРАВОЧНИК ХРАНИТ НОМЕРА ТЕЛЕФОНОВ ЛЮДЕЙ, СОРТИРОВКУ ПО ИМЕНАМ, ЧТОБЫ ИМЕНА МОЖНО ЛЕГКО НАЙТИ.
- СЛОВАРЬ — СЛОВАРЬ ХРАНИТ СЛОВА В АЛФАВИТНОМ ПОРЯДКЕ ТАК, ЧТО ПОИСК ЛЮБОГО СЛОВА СТАНОВИТСЯ ЛЕГКИМ.

ЗАЧЕМ ИЗУЧАТЬ СОРТИРОВКУ?

- КОГДА ВЫ ВЫПОЛНЯЕТЕ СОРТИРОВКУ НА МАССИВЕ/ЭЛЕМЕНТАХ, МНОГИЕ ЗАДАЧИ СТАНОВЯТСЯ ЛЕГКИМ (К ПРИМЕРУ, МИН/МАКС)

- ВЫПОЛНЕНИЕ СОРТИРОВКИ ТАКЖЕ ДАЕТ НЕСКОЛЬКО АЛГОРИТМИЧЕСКИХ РЕШЕНИЙ, СОДЕРЖАЩИХ МНОГИЕ ДРУГИЕ ИДЕИ, ТАКИЕ КАК:

  - ИТЕРАТИВНЫЙ
  - РАЗДЕЛЯЙ И ВЛАСТВУЙ
  - СРАВНЕНИЕ И ОТСУТСТВИЕ СРАВНЕНИЯ
  - РЕКУРСИВНЫЙ

КАТЕГОРИИ СОРТИРОВКИ

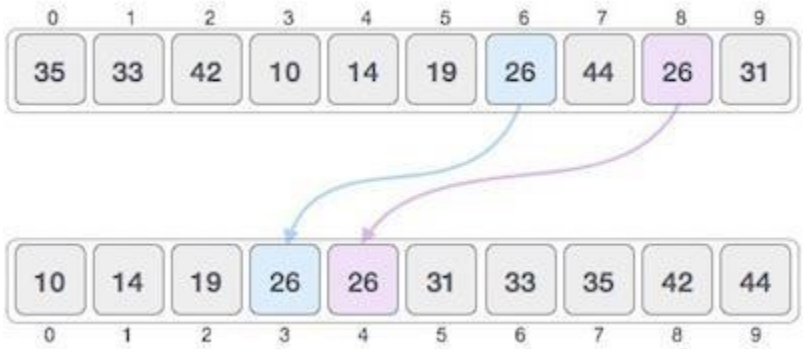
МЕТОДЫ СОРТИРОВКИ МОЖНО РАЗДЕЛИТЬ НА ДВЕ КАТЕГОРИИ:

- **ВНУТРЕННЯЯ СОРТИРОВКА**: ЕСЛИ ВСЕ ДАННЫЕ, ПОДЛЕЖАЩИЕ СОРТИРОВКЕ, МОГУТ ОДНОВРЕМЕННО ОТРЕГУЛИРОВАТЬСЯ В ОСНОВНОЙ ПАМЯТИ, ВЫПОЛНЯЕТСЯ МЕТОД ВНУТРЕННЕЙ СОРТИРОВКИ.

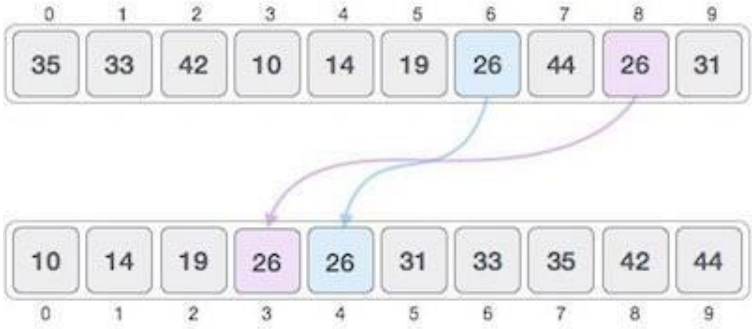
- **ВНЕШНЯЯ СОРТИРОВКА**: ЕСЛИ ДАННЫЕ, ПОДЛЕЖАЩИЕ СОРТИРОВКЕ, НЕ МОГУТ БЫТЬ РАЗМЕЩЕНЫ В ПАМЯТИ ОДНОВРЕМЕННО, И НЕКОТОРЫЕ ДОЛЖНЫ СОХРАНЯТЬСЯ В ДОПОЛНИТЕЛЬНОЙ ПАМЯТИ, ТАКОЙ КАК ЖЕСТКИЙ ДИСК, ДИСК, МАГНИТНЫЕ ЛЕНТЫ И Т. Д., ТОГДА ВЫПОЛНЯЮТСЯ ВНЕШНИЕ МЕТОДЫ СОРТИРОВКИ.

СТАБИЛЬНАЯ И НЕСТАБИЛЬНАЯ СОРТИРОВКА

- ЕСЛИ АЛГОРИТМ СОРТИРОВКИ ПОСЛЕ СОРТИРОВКИ СОДЕРЖИМОГО **НЕ ИЗМЕНЯЕТ ПОСЛЕДОВАТЕЛЬНОСТЬ ПОХОЖИХ СОДЕРЖИМЫХ**, В КОТОРОЙ ОНИ ПОЯВЛЯЮТСЯ, ЭТО НАЗЫВАЕТ СТАБИЛЬНОЙ СОРТИРОВКОЙ.



- ЕСЛИ АЛГОРИТМ СОРТИРОВКИ ПОСЛЕ СОРТИРОВКИ СОДЕРЖИМОГО **ИЗМЕНЯЕТ ПОСЛЕДОВАТЕЛЬНОСТЬ ПОХОЖИХ СОДЕРЖИМЫХ**, В КОТОРОЙ ОНИ ПОЯВЛЯЮТСЯ, ЭТО НАЗЫВАЕТСЯ НЕСТАБИЛЬНОЙ СОРТИРОВКОЙ.



СТАБИЛЬНОСТЬ АЛГОРИТМА ИМЕЕТ ЗНАЧЕНИЕ, КОГДА МЫ ХОТИМ СОХРАНИТЬ ПОСЛЕДОВАТЕЛЬНОСТЬ ИСХОДНЫХ ЭЛЕМЕНТОВ, НАПРИМЕР, В КОРТЕЖЕ.

АДАПТИВНЫЙ И НЕАДАПТИВНЫЙ АЛГОРИТМ СОРТИРОВКИ

- Алгоритм сортировки называется **адаптивным**, если он использует преимущества уже «отсортированных» элементов в списке, который должен быть отсортирован. То есть при сортировке, если в исходном списке уже есть какой-то элемент, адаптивные алгоритмы примут это во внимание и постараются не переупорядочивать их.

- **Неадаптивный** алгоритм – это алгоритм, который не учитывает элементы, которые уже отсортированы. Они пытаются принудительно переупорядочить каждый элемент, чтобы подтвердить их сортировку.

НЕКОТОРЫЕ ТЕРМИНЫ, КАК ПРАВИЛО, ПРИДУМАНЫ ПРИ ОБСУЖДЕНИИ МЕТОДОВ СОРТИРОВКИ, ВОТ КРАТКОЕ ВВЕДЕНИЕ К НИМ

- ПО ВОЗРАСТАНИЮ. Считается, что последовательность значений находится в **возрастающем порядке** , если последующий элемент больше предыдущего. Например, 1, 3, 4, 6, 8, 9

- ПО УБЫВАНИЮ. Последовательность значений называется в **порядке убывания** , если последующий элемент меньше текущего. Например, 9, 8, 6, 4, 3, 1

- НЕВОЗРАСТАЮЩИЙ. Считается, что последовательность значений находится в **не возрастающем порядке**, если последующий элемент меньше или равен своему предыдущему элементу в последовательности. Этот порядок происходит, когда последовательность содержит повторяющиеся значения. Например, 9, 8, 6, 3, 3, 1

- НЕУБЫВАЮЩИЙ. Считается, что последовательность значений находится в **неубывающем порядке** , если последующий элемент больше или равен своему предыдущему элементу в последовательности. Этот порядок происходит, когда последовательность содержит повторяющиеся значения. Например, 1, 3, 3, 6, 8, 9

АЛГОРИТМЫ СОРТИРОВКИ

СУЩЕСТВУЕТ МНОГО РАЗЛИЧНЫХ АЛГОРИТМОВ СОРТИРОВКИ С РАЗЛИЧНЫМИ ПЛЮСАМИ И МИНУСАМИ. ДЛЯ ВЫБОРА АЛГОРИТМА СОРТИРОВКИ ДЛЯ КОНКРЕТНОЙ ЗАДАЧИ УЧИТЫВАЙТЕ ВРЕМЯ ВЫПОЛНЕНИЯ, СЛОЖНОСТЬ ПРОСТРАНСТВА И ОЖИДАЕМЫЙ ФОРМАТ СПИСКА ВВОДОВ.

Алгоритм сортировки	Стабильность
Сортировка пузырьком	✓
Сортировка выбором	✘
Быстрая сортировка	✘
Сортировка вставками	✓
Сортировка слиянием	✓

Эффективность алгоритмов сортировки

Критериями оценки эффективности алгоритма сортировки

Пространственная сложность

Означает количество памяти, затраченной на выполнение алгоритма. Пространственная сложность включает вспомогательную память и память для хранения входных данных.

Временная сложность

Означает время, за которое алгоритм выполняет поставленную задачу с учетом входных данных. Может быть выражена с использованием следующих нотаций:

«Омега» (Ω)
«О» большое (O)
«Тета» (θ)

ВЫБОР АЛГОРИТМА СОРТИРОВКИ

В таблице представлена оценка сложности алгоритмов, упомянутых ранее:

Алгоритм сортировки	Время работы в худшем случае	Время работы в среднем случае	Время работы в лучшем случае	Пространственная сложность
Сортировка пузырьком	n^2	n^2	n	1
Сортировка выбором	n^2	n^2	n^2	1
Быстрая сортировка	n^2	nlog n	nlog n	nlog n
Сортировка кучей	nlog n	nlog n	nlog n	1
Сортировка вставками	n^2	n^2	n	1
Сортировка слиянием	nlog n	nlog n	nlog n	n

ВЫБОР АЛГОРИТМА СОРТИРОВКИ

HTTPS://WWW.TOPTAL.COM/DEVELOPERS/SORTING-ALGORITHMS

ПРИ ВЫБОРЕ АЛГОРИТМА СОРТИРОВКИ ВЗВЕШИВАЙТЕ ЭТИ ФАКТОРЫ.
К ПРИМЕРУ, БЫСТРАЯ СОРТИРОВКА — ОЧЕНЬ БЫСТРЫЙ АЛГОРИТМ, НО МОЖЕТ БЫТЬ СЛОЖНЫМ В РЕАЛИЗАЦИИ;

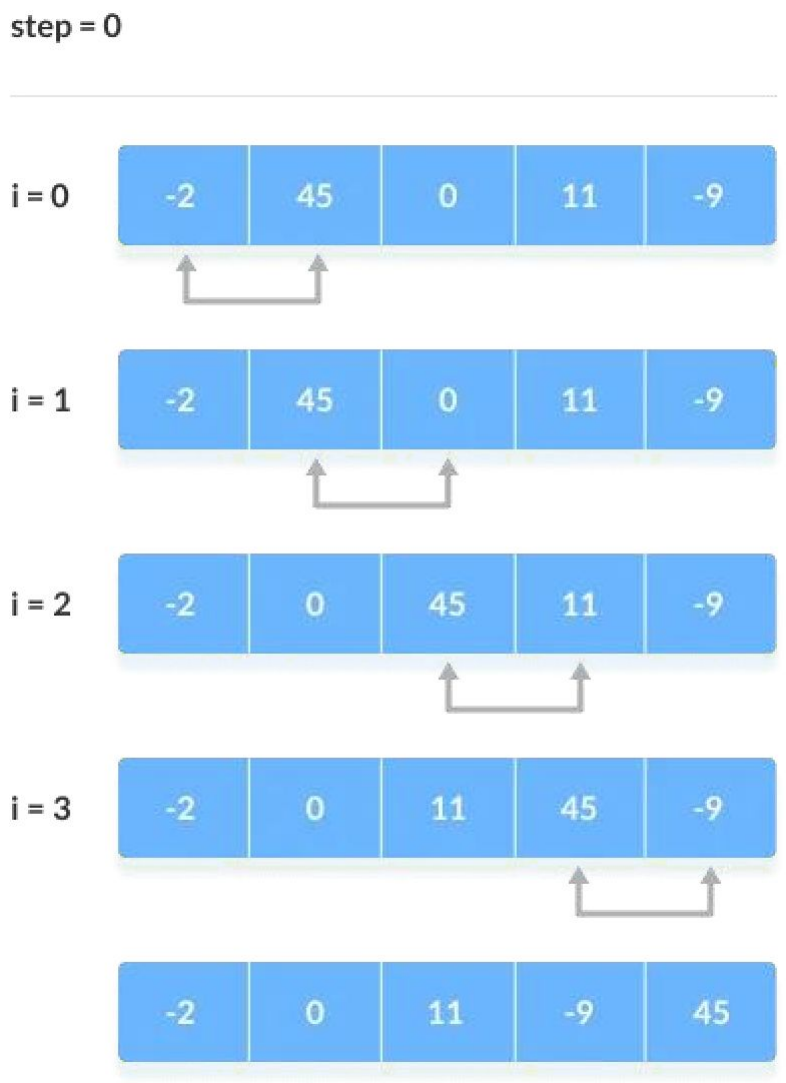
ПУЗЫРЬКОВАЯ СОРТИРОВКА

- АЛГОРИТМ СОРТИРОВКИ, КОТОРЫЙ СРАВНИВАЕТ ДВА СМЕЖНЫХ ЭЛЕМЕНТА И МЕНЯЕТ ИХ МЕСТАМИ, ПОКА ОНИ НЕ ОСТАЮТСЯ В ПРЕДУСМОТРЕННОМ ПОРЯДКЕ.
- КАК ДВИЖЕНИЕ ПУЗЫРЬКОВ ВОЗДУХА В ВОДЕ, ПОДНИМАЮЩИХСЯ НА ПОВЕРХНОСТЬ, КАЖДЫЙ ЭЛЕМЕНТ МАССИВА ДВИГАЕТСЯ ДО КОНЦА В КАЖДОЙ ИТЕРАЦИИ, ПОЭТОМУ ЕГО НАЗЫВАЮТ ПУЗЫРЬКОВОЙ СОРТИРОВКОЙ.

РАБОТА ПУЗЫРЬКОВОЙ СОРТИРОВКИ

Предположим, мы пытаемся отсортировать элементы в порядке возрастания.

1. ПЕРВАЯ ИТЕРАЦИЯ (СРАВНЕНИЕ И ЗАМЕНА)
   - НАЧИНАЯ С НУЛЕВОГО ИНДЕКСА, СРАВНИТЕ ПЕРВЫЙ И ВТОРОЙ ЭЛЕМЕНТЫ.
   - ЕСЛИ ПЕРВЫЙ ЭЛЕМЕНТ БОЛЬШЕ ВТОРОГО ЭЛЕМЕНТА, ОНИ МЕНЯЮТСЯ.
   - ТЕПЕРЬ СРАВНИТЕ ВТОРОЙ И ТРЕТИЙ ЭЛЕМЕНТЫ. ПОМЕНЯЙТЕ ИХ, ЕСЛИ ОНИ НЕ В ПОРЯДКЕ.
   - ПРОЦЕСС ПРОДОЛЖАЕТСЯ ДО ПОСЛЕДНЕГО ЭЛЕМЕНТА.



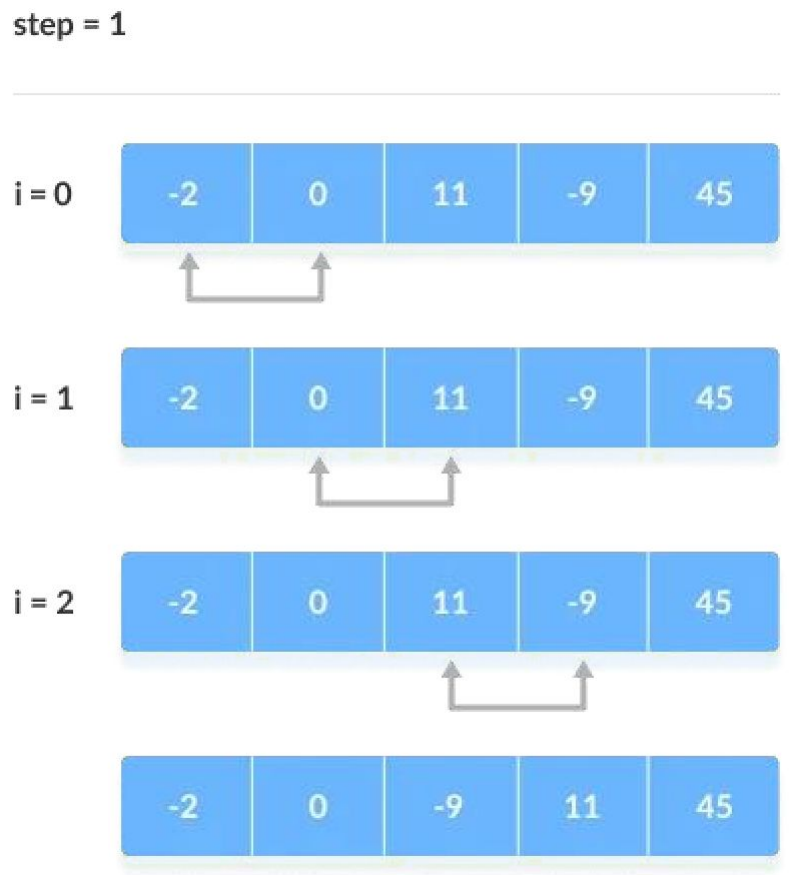
Сравнение соседних элементов

РАБОТА ПУЗЫРЬКОВОЙ СОРТИРОВКИ

2. ОСТАВШАЯСЯ ИТЕРАЦИЯ

ПРОЦЕСС ПРОДОЛЖАЕТСЯ ДЛЯ ОСТАВШИХСЯ ИТЕРАЦИЙ.

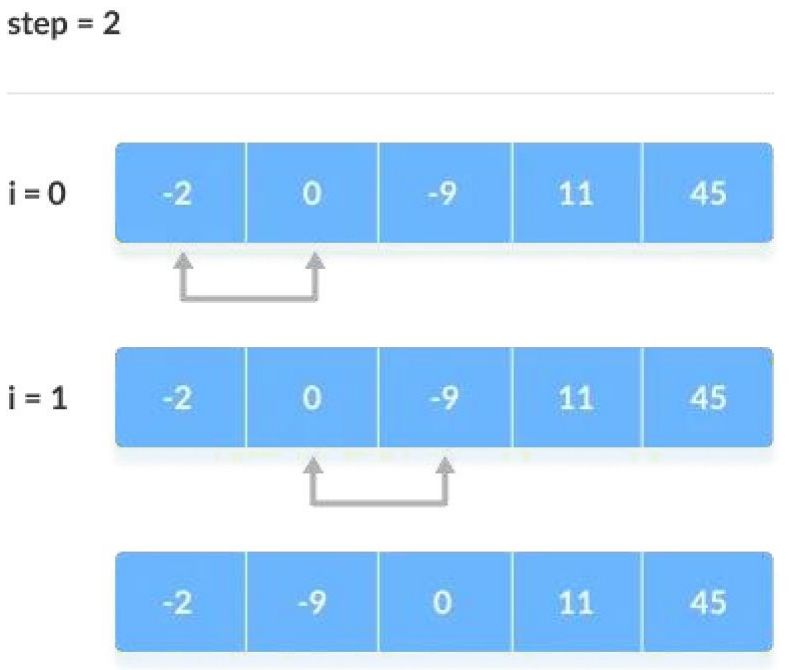
ПОСЛЕ КАЖДОЙ ИТЕРАЦИИ **БОЛЬШИЙ ЭЛЕМЕНТ** СРЕДИ НЕСОРТИРОВАННЫХ ЭЛЕМЕНТОВ СТАВИТСЯ В КОНЕЦ.



самый большой элемент в конце

РАБОТА ПУЗЫРЬКОВОЙ СОРТИРОВКИ

НА КАЖДОЙ ИТЕРАЦИИ СРАВНЕНИЕ ПРОИСХОДИТ **ДО ПОСЛЕДНЕГО НЕСОРТИРОВАННОГО ЭЛЕМЕНТА**.



Сравните соседние элементы

РАБОТА ПУЗЫРЬКОВОЙ СОРТИРОВКИ

МАССИВ ОТСОРТИРОВАН, КОГДА ВСЕ НЕСОРТИРОВАННЫЕ ЭЛЕМЕНТЫ РАЗМЕЩЕНЫ НА ПРАВИЛЬНЫХ ПОЗИЦИЯХ.

Массив отсортирован, если все элементы сохранены в правильном порядке.

ЭФФЕКТИВНОСТЬ РАБОТЫ

ЭТОТ АЛГОРИТМ СЧИТАЕТСЯ УЧЕБНЫМ И В ЧИСТОМ ВИДЕ НА ПРАКТИКЕ ПОЧТИ НЕ ПРИМЕНЯЕТСЯ. ДЕЛО В ТОМ, ЧТО СРЕДНЕЕ КОЛИЧЕСТВО ПРОВЕРОК И ПЕРЕСТАНОВОК В МАССИВЕ — ЭТО КОЛИЧЕСТВО ЭЛЕМЕНТОВ В КВАДРАТЕ.

Эффективность работы пузырьковой сортировки равна **O (n²)**

Например, для массива из 10 элементов потребуется 100 проверок,
а для массива из 100 элементов — уже в 100 раз >, 10 000 проверок.

Реализация на Python

```
from random import randint
N = 15
def Buble_sort(arr):
    n = len(arr)# число элементов
    for i in range(0,n-1):# внешний цикл, кол-во проходов n-1
        for j in range(0, n-i-1):# внутренний цикл, проход по оставшимся
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
arr = [randint(1, 99) for item in range(N)]
print(arr)
Buble_sort(arr)
print(arr)
```

Практика. ПУЗЫРЬКОВАЯ СОРТИРОВКА

Задача: Сортировка большого списка
Сгенерируйте список случайных чисел большой длины (например, 10^4 элементов) в диапазоне от 1 до 10000.
Используя алгоритм **пузырьковой сортировки** отсортируйте этот список в обратном порядке.
Замерьте время выполнения сортировки и выведите его.
Вывести отсортированный список.

Практика. ПУЗЫРЬКОВАЯ СОРТИРОВКА

1

Сгенерируйте список случайных чисел большой длины (например, 10^4 элементов) в диапазоне от 1 до 10000.

```
import random
import time

# Генерация большого списка случайных чисел
random.seed(42)  # Для воспроизводимости результатов
big_list = [random.randint(1, 10000) for _ in range(10**4)]
```

Практика. ПУЗЫРЬКОВАЯ СОРТИРОВКА

2

Используя алгоритм сортировки пузырьком отсортируйте этот список (от большего к меньшему).

```
# Функция для сортировки списка в обратном
#порядке с помощью сортировки пузырьком
def bubble_sort_reverse(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] < arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

Практика. ПУЗЫРЬКОВАЯ СОРТИРОВКА

3

Замерьте время выполнения сортировки и выведите отсортированный список:

```
# Замер времени выполнения сортировки пузырьком в обратном порядке
start_time = time.time()
bubble_sort_reverse(big_list)
end_time = time.time()
print(big_list)
# Вывод времени выполнения
print(f"Время выполнения сорт-ки пузырьком в обр.п.: {end_time - start_time} секунд")
```

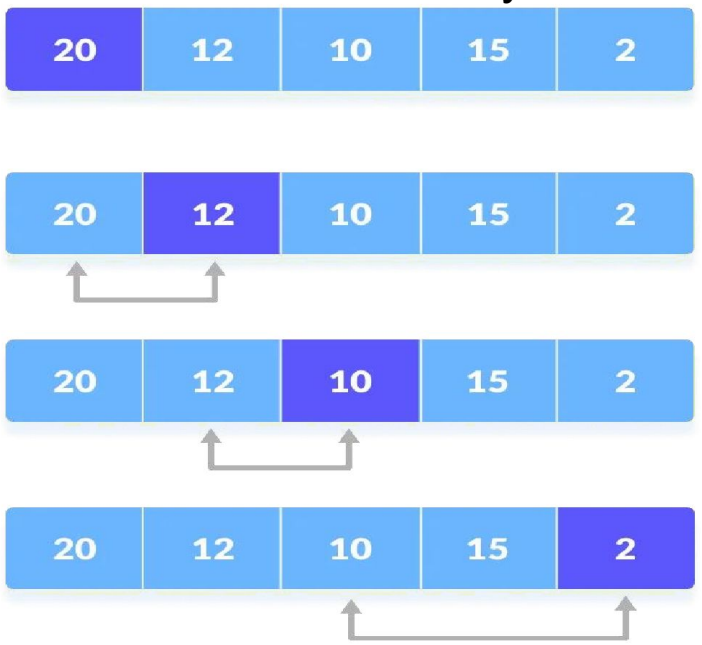
АЛГОРИТМ СОРТИРОВКИ ВЫБОРОМ

ВЫБИРАЕТ НАИМЕНЬШИЙ ЭЛЕМЕНТ ИЗ НЕСОРТИРОВАННОГО СПИСКА НА КАЖДОЙ ИТЕРАЦИИ И РАЗМЕЩАЕТ ЭТОТ ЭЛЕМЕНТ В НАЧАЛЕ НЕСОРТИРОВАННОГО СПИСКА.

СОРТИРОВКА ВЫБОРОМ

1. УСТАНОВИТЕ ПЕРВЫЙ ЭЛЕМЕНТ КАК МИНИМУМ
2. СРАВНИТЕ МИНИМУМ СО ВТОРЫМ ЭЛЕМЕНТОМ. ЕСЛИ ВТОРОЙ ЭЛЕМЕНТ МЕНЬШЕ МИНИМУМА, НАЗНАЧИТЕ ВТОРОЙ ЭЛЕМЕНТ КАК МИНИМУМ.
3. СРАВНИТЕ МИНИМУМ С ТРЕТИМ ЭЛЕМЕНТОМ. ОПЯТЬ, ЕСЛИ ТРЕТИЙ ЭЛЕМЕНТ МЕНЬШЕ, ТО НАЗНАЧИТЕ МИНИМУМ ТРЕТЬЕМУ ЭЛЕМЕНТУ, ИНАЧЕ НИЧЕГО НЕ ДЕЛАТЬ. ПРОЦЕСС ПРОДОЛЖАЕТСЯ ДО ПОСЛЕДНЕГО ЭЛЕМЕНТА.
4. ПОСЛЕ КАЖДОЙ ИТЕРАЦИИ МИНИМУМ РАЗМЕЩАЕТСЯ В НАЧАЛЕ НЕСОРТИРОВАННОГО СПИСКА.

Выберите первый элемент как минимум

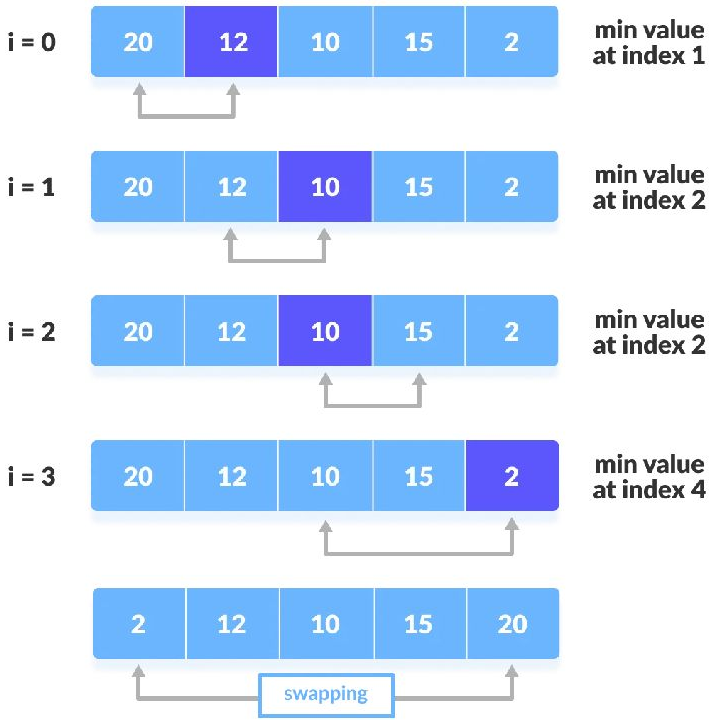


Сравните минимум с остальными элементами

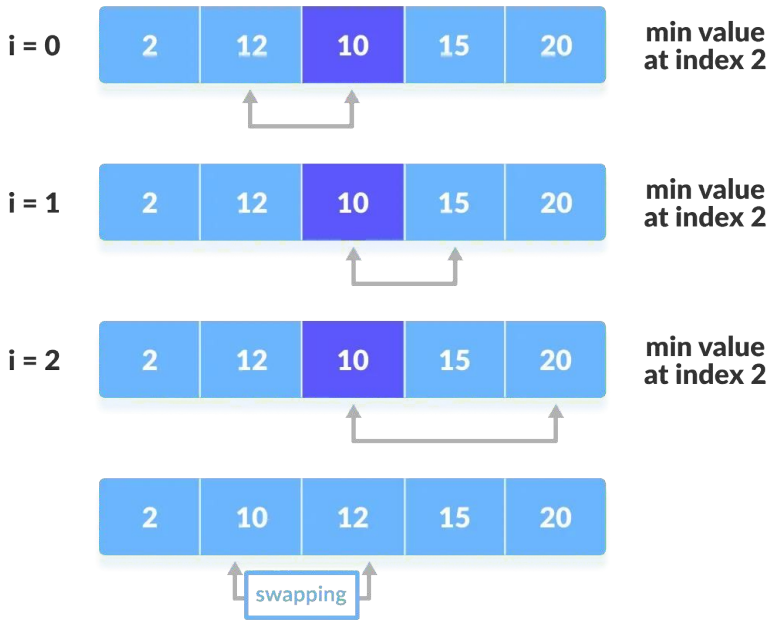
Поменяйте местами первый с минимумом

4. ДЛЯ КАЖДОЙ ИТЕРАЦИИ ИНДЕКСИРОВАНИЕ НАЧИНАЕТСЯ С ПЕРВОГО НЕСОРТИРОВАННОГО ЭЛЕМЕНТА. ШАГИ 1-3 ПОВТОРЯЮТСЯ, ПОКА ВСЕ ЭЛЕМЕНТЫ НЕ БУДУТ РАЗМЕЩЕНЫ В ПРАВИЛЬНОМ ПОЛОЖЕНИИ.

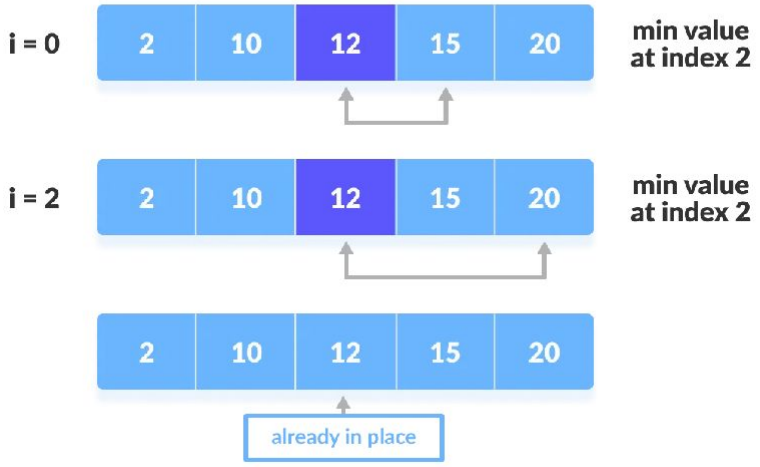
Первая итерация



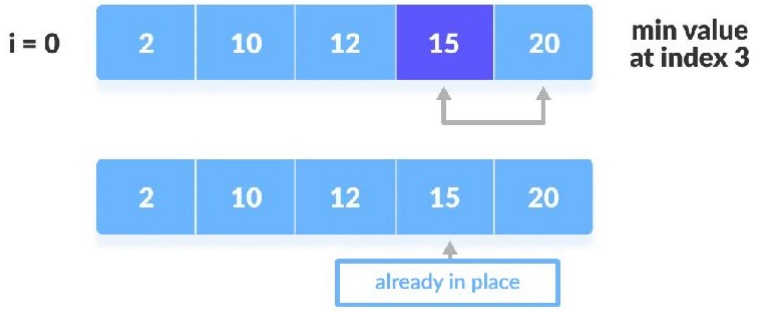
Вторая итерация



Третья итерация



Четвертая итерация



АЛГОРИТМ СОРТИРОВКИ ВЫБОРОМ

```
from random import randint
N = 15
def sel_sort(row):
    n = len(row)
    for i in range(n-1):
        m = i
        for j in range(i+1, n):
            if row[j] < row[m]:
                m = j
        row[i], row[m] = row[m], row[i]
arr = [randint(1, 99) for item in range(N)]
print(arr)
sel_sort(arr)
print(arr)
```

Сортировка по выбору имеет сложность **O(n^2)** и используется:

- Сортировка небольших массивов
- Требуется меньше подкачки

Практика. АЛГОРИТМ СОРТИРОВКИ ВЫБОРОМ

Задача: Сортировка большого списка
Сгенерируйте список случайных чисел большой длины (например, 10^4 элементов) в диапазоне от 1 до 10000.
Используя алгоритм сортировки выбором отсортируйте этот список (от меньшего к большему).
Замерьте время выполнения сортировки и выведите его.
Вывести отсортированный список.

Практика. АЛГОРИТМ СОРТИРОВКИ ВЫБОРОМ

1

Сгенерируйте список случайных чисел большой длины (например, 10^4 элементов) в диапазоне от 1 до 10000.

```
import random
from time import time
random.seed(42)
big_list = [random.randint(1,10000) for _ in range(10**4)]
print(big_list)
```

Практика. АЛГОРИТМ СОРТИРОВКИ ВЫБОРОМ

2

Используя алгоритм сортировки выбором отсортируйте этот список (от меньшего к большему).

```
def sel_sort(row):
    n = len(row)
    for i in range(n-1):
        m = i
        for j in range(i+1, n):
            if row[j] < row[m]:
                m = j
        row[i], row[m] = row[m], row[i]
```

Практика. АЛГОРИТМ СОРТИРОВКИ ВЫБОРОМ

3

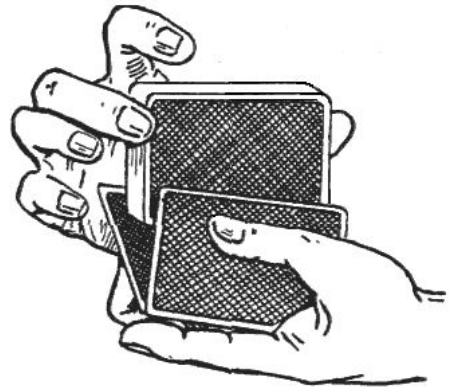
Замерьте время выполнения сортировки и выведите его.

```
start_time = time()
sel_sort(big_list)
end_time =  time()
print(f'Время выполнения сортировки выбором: {end_time-start_time}')
```

АЛГОРИТМ СОРТИРОВКИ ВСТАВКОЙ

АЛГОРИТМ СОРТИРОВКИ ВСТАВКОЙ

- АЛГОРИТМ СОРТИРОВКИ, КОТОРЫЙ РАЗМЕЩАЕТ НЕСОРТИРОВАННЫЙ ЭЛЕМЕНТ НА ЕГО ПОДХОДЯЩЕМ МЕСТЕ В КАЖДОЙ ИТЕРАЦИИ.
- СОРТИРОВКА ВСТАВКОЙ РАБОТАЕТ ТАК КАК СОРТИРОВКА КАРТ В РУКАХ В КАРТОЧНОЙ ИГРЕ.
- СЧИТАЕМ, ЧТО ПЕРВАЯ КАРТА УЖЕ ОТОРТАРИРОВАНА, ТОГДА ВЫБИРАЕМ НЕОТСОРТИРОВАННУЮ КАРТУ. ЕСЛИ НЕСОРТИРОВАННАЯ КАРТА БОЛЬШЕ, ЧЕМ КАРТА В РУКАХ, ОНА КЛАДЕТСЯ СПРАВА, ИНАЧЕ, СЛЕВА. ТАКИМ ЖЕ СПОСОБОМ БЕРУТСЯ ДРУГИЕ НЕОТСОРТИРОВАННЫЕ КАРТЫ И КЛАДУТСЯ НА ИХ ПРАВИЛЬНОЕ МЕСТО.
- АНАЛОГИЧНЫЙ ПОДХОД ИСПОЛЬЗУЕТСЯ СОРТИРОВКОЙ ВСТАВКОЙ.

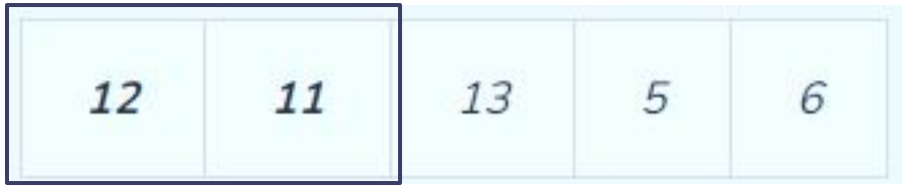


РАБОТА СОРТИРОВКИ ВСТАВКОЙ

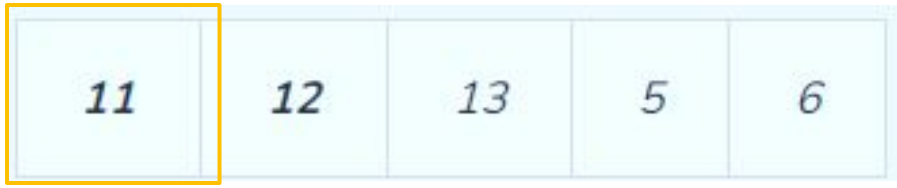
Предположим, нам нужно отсортировать следующий массив

Первый проход:

Первоначально первые два элемента массива сравниваются при сортировке вставками.



Здесь 12 больше 11, следовательно, они не в порядке возрастания, поменяем местами 11 и 12. Итак, на данный момент 11 хранится в отсортированном подмассиве.

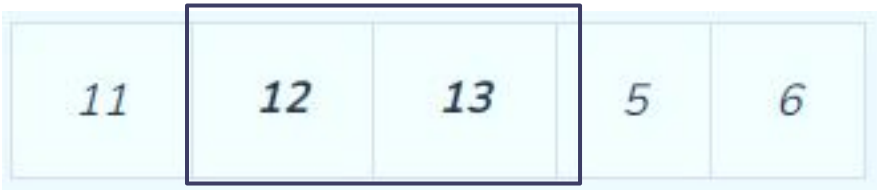


РАБОТА СОРТИРОВКИ ВСТАВКОЙ

Предположим, нам нужно отсортировать следующий массив

Второй проход:

Теперь перейдите к следующим двум элементам и сравните их.

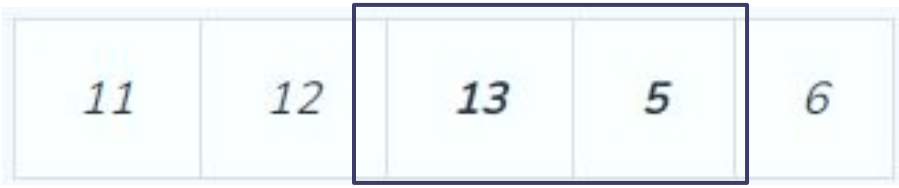


Здесь 13 больше 12, поэтому оба элемента расположены в порядке возрастания, следовательно, замены не произойдет. 12 также хранятся в отсортированном подмассиве вместе с 11

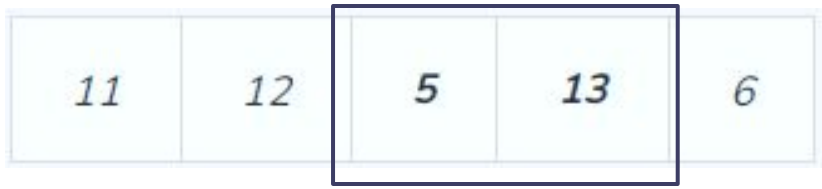
РАБОТА СОРТИРОВКИ ВСТАВКОЙ

Третий проход:

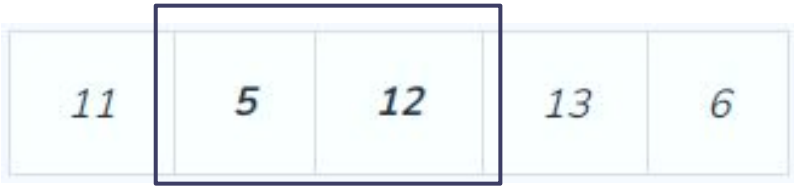
Переходим к следующим двум элементам — 13 и 5.



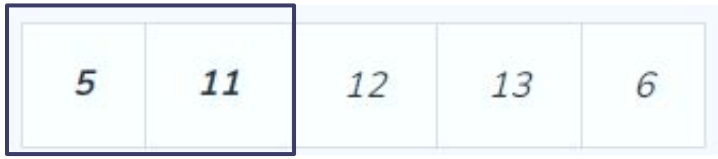
И 5, и 13 не находятся на своих местах, поэтому поменяйте их местами.



После замены элементы 12 и 5 не сортируются, поэтому поменяйте местами еще раз.



Здесь снова 11 и 5 не отсортированы, следовательно, снова поменяйте местами



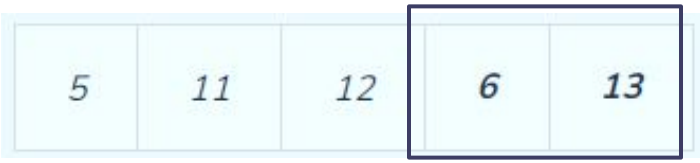
Здесь цифра 5 находится в правильном положении.

РАБОТА СОРТИРОВКИ ВСТАВКОЙ

Четвертый проход:

Теперь в отсортированном подмассиве присутствуют элементы 5, 11 и 12. Переходим к следующим двум элементам 13 и 6.

Очевидно, что они не отсортированы, поэтому выполните обмен между ними.



Теперь 6 меньше 12, следовательно, поменяйте местами еще раз.

Здесь также замена делает 11 и 6 неотсортированными, поэтому поменяйте местами еще раз.

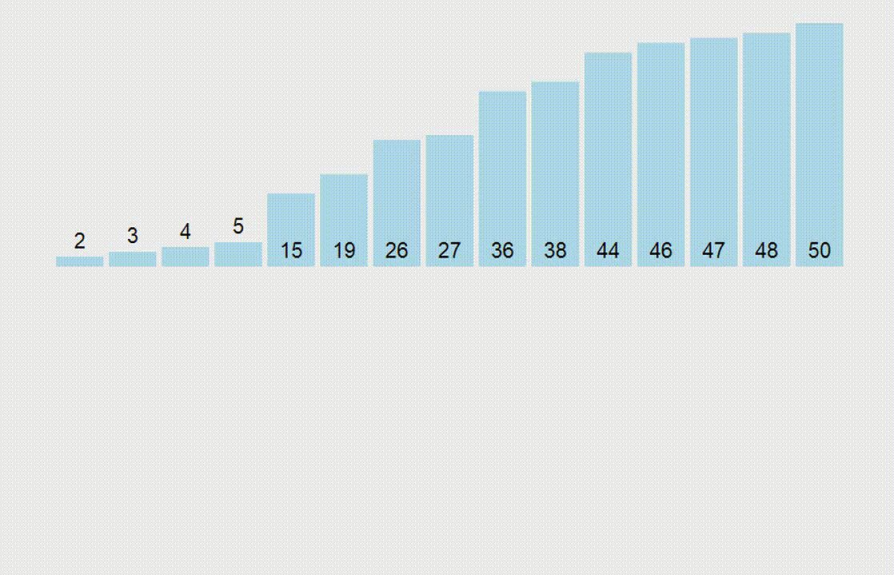
Наконец, массив полностью отсортирован.

АЛГОРИТМ СОРТИРОВКИ ВСТАВКОЙ

```
def insertionSort(array):
    for i in range(1, len(array)):
        key = array[i]
        j = i-1
        while array[j] > key and j >= 0:
            array[j+1] = array[j]
            j -= 1
        array[j+1] = key
    return array
```

Сортировка вставкой имеет сложность $O(n^2)$ и используется, когда:

- Осталось отсортировать несколько элементов;
- Массив небольшой.



Практика. АЛГОРИТМ СОРТИРОВКИ ВСТАВКОЙ

Задача: Сортировка большого списка
Сгенерируйте список случайных чисел большой длины (например, 10^4 элементов) в диапазоне от 1 до 10000.
Используя алгоритм сортировки вставкой отсортируйте этот список в обратном порядке.
Замерьте время выполнения сортировки и выведите его.
Вывести отсортированный список.

Практика. АЛГОРИТМ СОРТИРОВКИ ВСТАВКОЙ

1

Сгенерируйте список случайных чисел большой длины (например, 10^4 элементов) в диапазоне от 1 до 10000.

```
import random
from time import time
random.seed(42)
big_list = [random.randint(1,10000) for _ in range(10**4)]
print(big_list)
```

Практика. АЛГОРИТМ СОРТИРОВКИ ВСТАВКОЙ

2

Используя алгоритм сортировки вставкой отсортируйте этот список в обратном порядке:

```
def insertion_sort_reverse(arr):
    n = len(arr)
    for i in range(1, n):
        key = arr[i]
        j = i - 1
        while j >= 0 and key > arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

Практика. АЛГОРИТМ СОРТИРОВКИ ВСТАВКОЙ

3

Замерьте время выполнения сортировки и выведите его.

```
# Замер времени выполнения сортировки вставкой в обратном порядке
start_time = time.time()
insertion_sort_reverse(big_list)
end_time = time.time()
```

4

Вывести отсортированный список

```
# Вывод времени выполнения
print(f"Время вып-ния сорт. вставкой в обр. п.: {end_time - start_time} секунд")
```

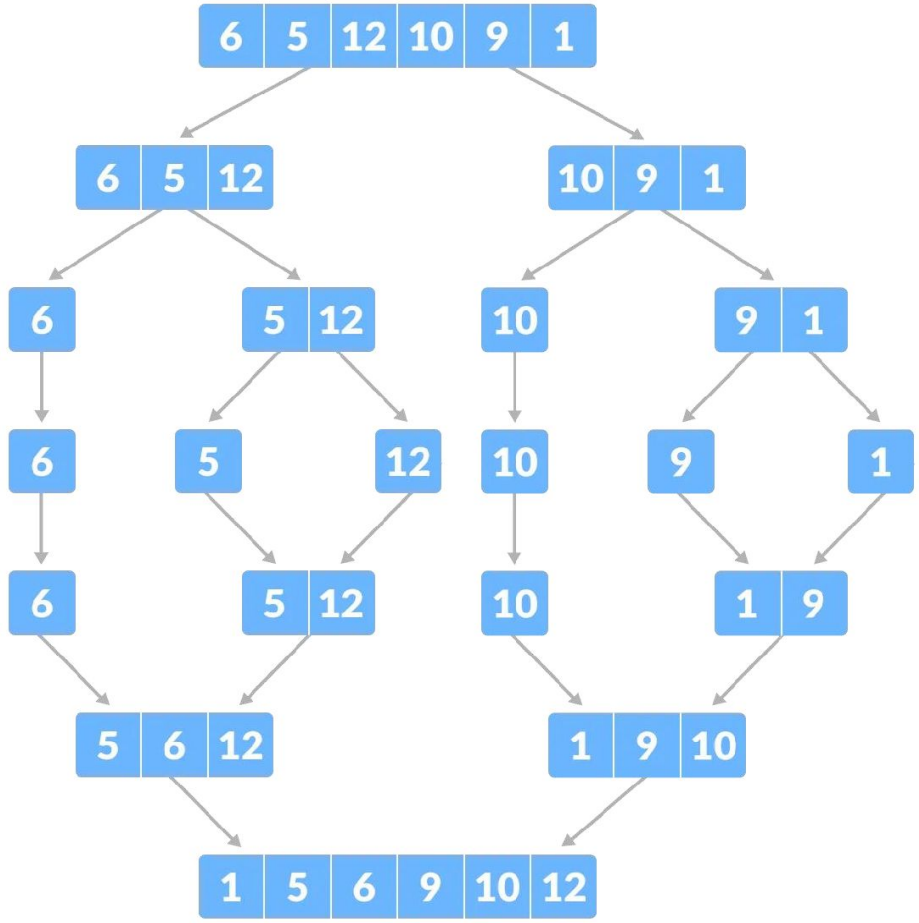
АЛГОРИТМ СОРТИРОВКИ СЛИЯНИЕМ

СОРТИРОВКИ СЛИЯНИЕМ

Учитывая сложность времени в худшем случае O (n log n), это один из наиболее быстрым алгоритмов.

Сортировка начинается с разбиения набора данных на отдельные части и сортировки частей. Затем он объединяет части таким образом, чтобы гарантировать, что она отсортировала объединенную часть.

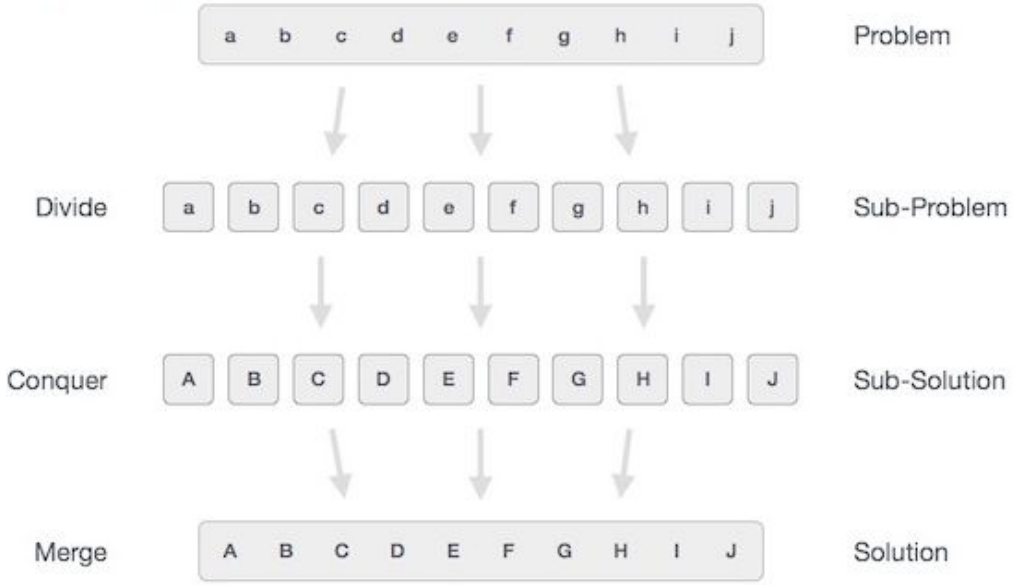
Сортировка и объединение продолжаются до тех пор, пока весь набор данных снова не станет единым целым.



Стратегия разделяй и властвуй

ПРИ ПОДХОДЕ «**РАЗДЕЛЯЙ И ВЛАСТВУЙ**» ПРОБЛЕМА РАЗБИВАЕТСЯ НА БОЛЕЕ МЕЛКИЕ ПОДЗАДАЧИ, ЗАТЕМ КАЖДАЯ ПРОБЛЕМА РЕШАЕТСЯ НЕЗАВИСИМО.

КОГДА МЫ ПРОДОЛЖАЕМ ДЕЛИТЬ ПОДЗАДАЧИ НА ЕЩЕ БОЛЕЕ МЕЛКИЕ ПОДЗАДАЧИ, МЫ МОЖЕМ В КОНЕЧНОМ ИТОГЕ ДОСТИЧЬ СТАДИИ, КОГДА БОЛЬШЕ ДЕЛЕНИЕ НЕВОЗМОЖНО. ЭТИ «АТОМАРНЫЕ» - ОПЕРАЦИИ, КОТОРЫЕ ЛИБО ВЫПОЛНЯЕТСЯ ЦЕЛИКОМ, ЛИБО НЕ ВЫПОЛНЯЕТСЯ ВОВСЕ - НАИМЕНЬШИЕ ВОЗМОЖНЫЕ ПОДЗАДАЧИ (ДРОБИ) РЕШЕНЫ. РЕШЕНИЕ ВСЕХ ПОДЗАДАЧ В КОНЕЧНОМ ИТОГЕ ОБЪЕДИНЯЕТСЯ, ЧТОБЫ ПОЛУЧИТЬ РЕШЕНИЕ ИСХОДНОЙ ЗАДАЧИ.



Стратегия разделяй и властвуй в 3 этапа

РАЗДЕЛИТЬ

РАЗБИЕНИЕ ПРОБЛЕМЫ НА БОЛЕЕ МЕЛКИЕ ПОДЗАДАЧИ. ПОДЗАДАЧИ ДОЛЖНЫ ПРЕДСТАВЛЯТЬ ЧАСТЬ ПЕРВОНАЧАЛЬНОЙ ПРОБЛЕМЫ. ЭТОТ ШАГ ОБЫЧНО ИСПОЛЬЗУЕТ **РЕКУРСИВНЫЙ ПОДХОД** ДЛЯ РАЗДЕЛЕНИЯ ПРОБЛЕМЫ ДО ТЕХ ПОР, ПОКА НИКАКАЯ ПОДЗАДАЧА НЕ СТАНЕТ БОЛЕЕ ДЕЛИМОЙ. НА ЭТОМ ЭТАПЕ ПОДЗАДАЧИ ПРИОБРЕТАЮТ АТОМАРНЫЙ ХАРАКТЕР, НО ВСЕ ЖЕ ПРЕДСТАВЛЯЮТ НЕКОТОРУЮ ЧАСТЬ АКТУАЛЬНОЙ ПРОБЛЕМЫ.

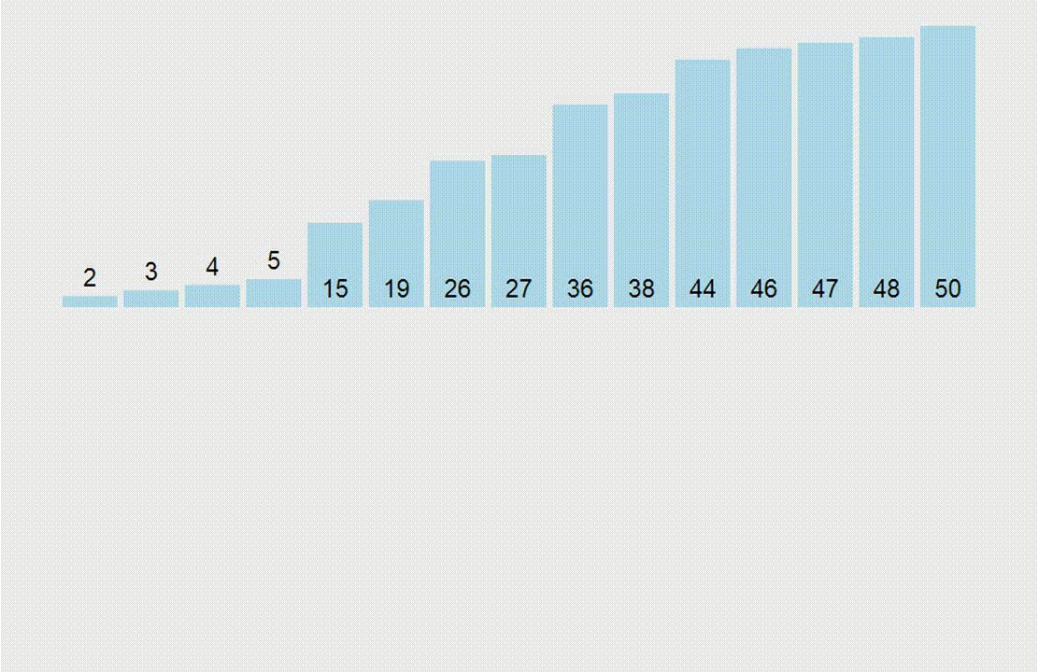
РЕШИТЬ

ЭТОТ ШАГ ПОЛУЧАЕТ МНОЖЕСТВО МЕЛКИХ ПОДЗАДАЧ, КОТОРЫЕ НЕОБХОДИМО РЕШИТЬ. КАК ПРАВИЛО, НА ЭТОМ УРОВНЕ ПРОБЛЕМЫ СЧИТАЮТСЯ «РЕШЕННЫМИ» САМИ ПО СЕБЕ.

СЛИЯНИЕ

КОГДА МЕНЬШИЕ ПОДЗАДАЧИ РЕШЕНЫ, ЭТОТ ЭТАП РЕКУРСИВНО ОБЪЕДИНЯЕТ ИХ, ПОКА ОНИ НЕ СФОРМУЛИРУЮТ РЕШЕНИЕ ИСХОДНОЙ ЗАДАЧИ. ЭТОТ АЛГОРИТМИЧЕСКИЙ ПОДХОД РАБОТАЕТ РЕКУРСИВНО, А ЭТАПЫ ЗАВОЕВАНИЯ И ОБЪЕДИНЕНИЯ РАБОТАЮТ ТАК БЛИЗКО, ЧТО ОНИ ВЫГЛЯДЯТ КАК ЕДИНОЕ ЦЕЛОЕ.

СОРТИРОВКА СЛИЯНИЕМ




```
def mergeSort(nums):
    if len(nums)==1:
        return nums
    mid = (len(nums)-1) // 2
    lst1 = mergeSort(nums[:mid+1])
    lst2 = mergeSort(nums[mid+1:])
    result = merge(lst1, lst2)
    return result
def merge(lst1, lst2):
    lst = []
    i = 0
    j = 0
    while(i<=len(lst1)-1 and j<=len(lst2)-1):
        if lst1[i]<lst2[j]:
            lst.append(lst1[i])
            i+=1
        else:
            lst.append(lst2[j])
            j+=1
    if i>len(lst1)-1:
        while(j<=len(lst2)-1):
            lst.append(lst2[j])
            j+=1
    else:
        while(i<=len(lst1)-1):
            lst.append(lst1[i])
            i+=1
    return lst
```

+/- СОРТИРОВКИ СЛИЯНИЕМ

Преимущества :

- **Стабильность** .
- **Гарантированная производительность в наихудшем случае:** временная сложность в наихудшем случае равна O(N logN), что означает, что она хорошо работает даже на больших наборах данных.
- **Параллелизуемый** : легко распараллелить, чтобы использовать преимущества нескольких процессоров или потоков.

Недостатки :

- **Пространственная сложность**: требует дополнительной памяти для хранения объединенных подмассивов во время процесса сортировки.
- **Не всегда оптимально для небольших наборов данных**: для небольших наборов данных сортировка слиянием имеет более высокую временную сложность, чем некоторые другие алгоритмы сортировки, такие как сортировка вставками. Это может привести к снижению производительности для очень маленьких наборов данных.

АЛГОРИТМ БЫСТРОЙ СОРТИРОВКИ

БЫСТРАЯ СОРТИРОВКА

БЫСТРАЯ СОРТИРОВКА - В ЦЕЛОМ ЭТО ОДИН ИЗ САМЫХ БЫСТРЫХ АЛГОРИТМОВ СОРТИРОВКИ МАССИВОВ, ОДНАКО НА ПРАКТИКЕ ОН ЧАЩЕ ВСЕГО ПРИМЕНЯЕТСЯ С РАЗНОГО РОДА МОДИФИКАЦИЯМИ. ЯВЛЯЕТСЯ ПРИМЕРОМ ПРИНЦИПА «РАЗДЕЛЯЙ И ВЛАСТВУЙ».

ИДЕЯ АЛГОРИТМА ЗАКЛЮЧАЕТСЯ В ТОМ, ЧТО ВЫБИРАЕТСЯ **ОПОРНЫЙ ЭЛЕМЕНТ**, ОТНОСИТЕЛЬНО КОТОРОГО БУДЕТ ПРОИСХОДИТ СОРТИРОВКА. РАВНЫЕ И БОЛЬШИЕ ЭЛЕМЕНТЫ ПОМЕЩАЮТСЯ СПРАВА, МЕНЬШИЕ – СЛЕВА. ЗАТЕМ К ПОЛУЧЕННЫМ ПОДМАССИВАМ РЕКУРСИВНО ПРИМЕНЯЮТСЯ ДВА ПЕРВЫХ ПУНКТА.

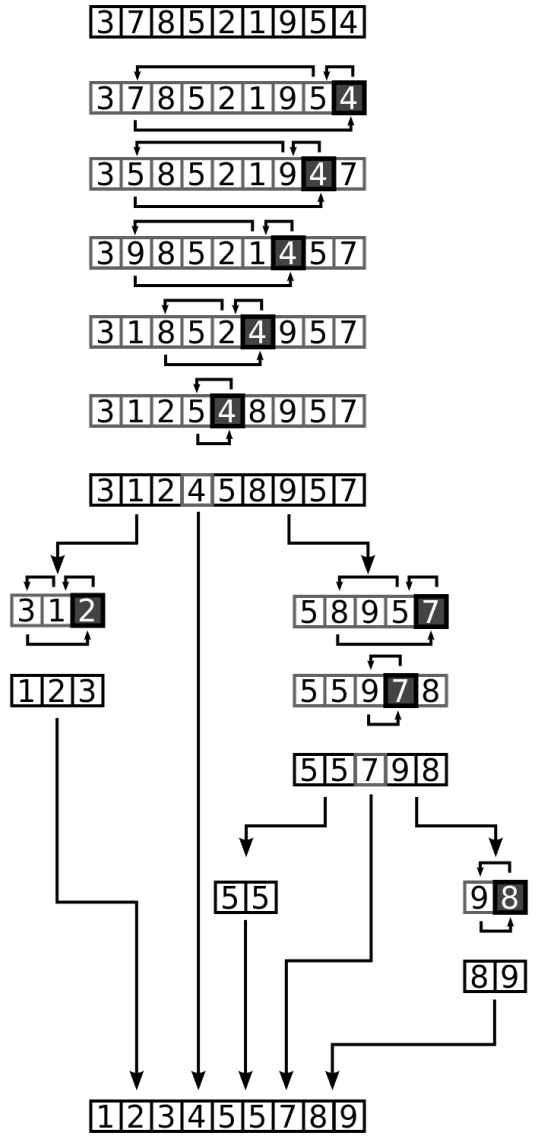
БЫСТРАЯ СОРТИРОВКА

АЛГОРИТМ СОСТОИТ ИЗ ТРЁХ ШАГОВ:

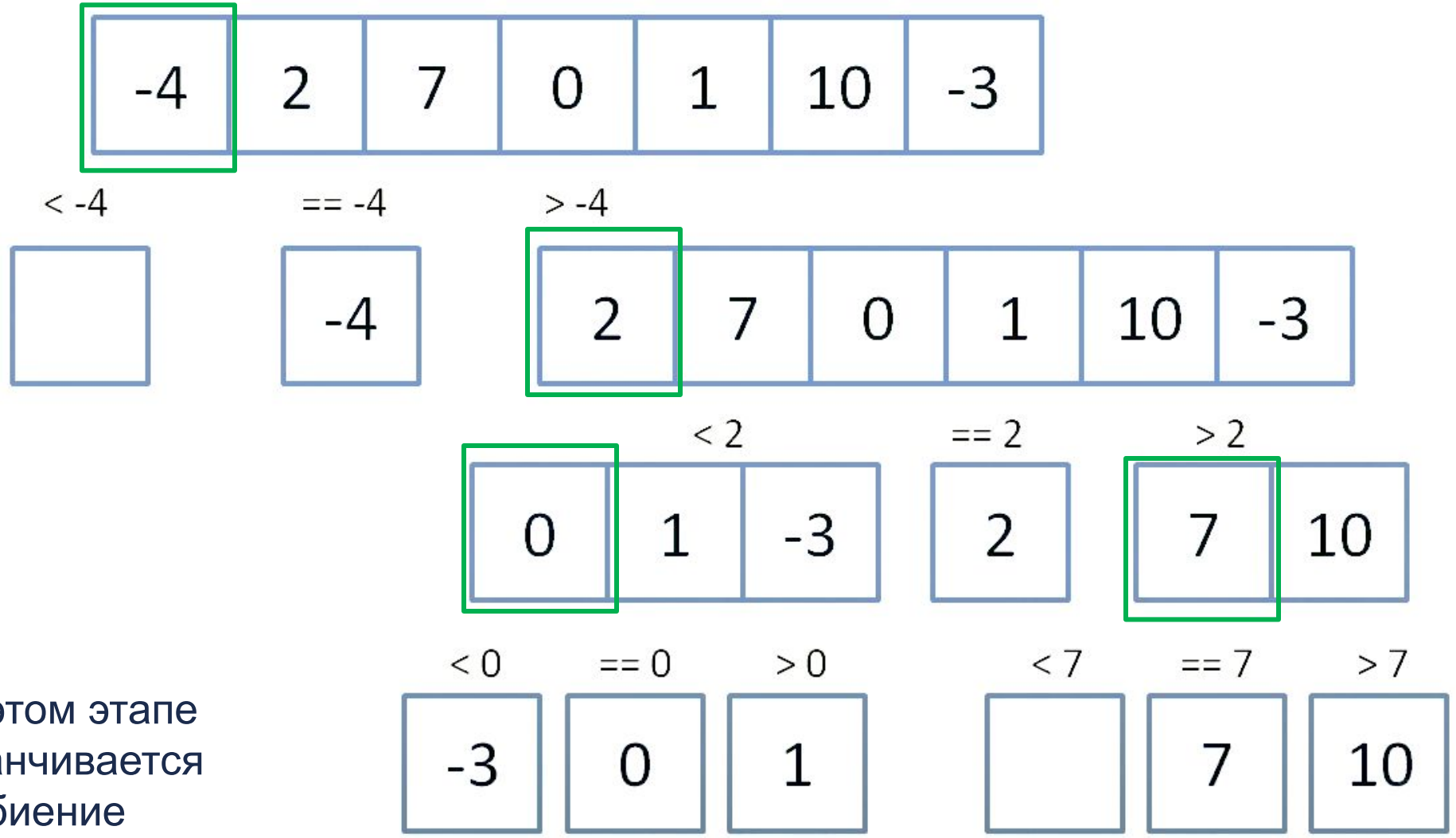
1. ВЫБРАТЬ ЭЛЕМЕНТ ИЗ МАССИВА - ОПОРНЫЙ.

2. РАЗБИЕНИЕ: ПЕРЕРАСПРЕДЕЛЕНИЕ ЭЛЕМЕНТОВ В МАССИВЕ ТАКИМ ОБРАЗОМ, ЧТО ЭЛЕМЕНТЫ, МЕНЬШИЕ ОПОРНОГО, ПОМЕЩАЮТСЯ ПЕРЕД НИМ, А БОЛЬШИЕ ИЛИ РАВНЫЕ - ПОСЛЕ.

3. РЕКУРСИВНО ПРИМЕНИТЬ ПЕРВЫЕ ДВА ШАГА К ДВУМ ПОДМАССИВАМ СЛЕВА И СПРАВА ОТ ОПОРНОГО ЭЛЕМЕНТА. РЕКУРСИЯ НЕ ПРИМЕНЯЕТСЯ К МАССИВУ, В КОТОРОМ ТОЛЬКО ОДИН ЭЛЕМЕНТ ИЛИ ОТСУТСТВУЮТ ЭЛЕМЕНТЫ.



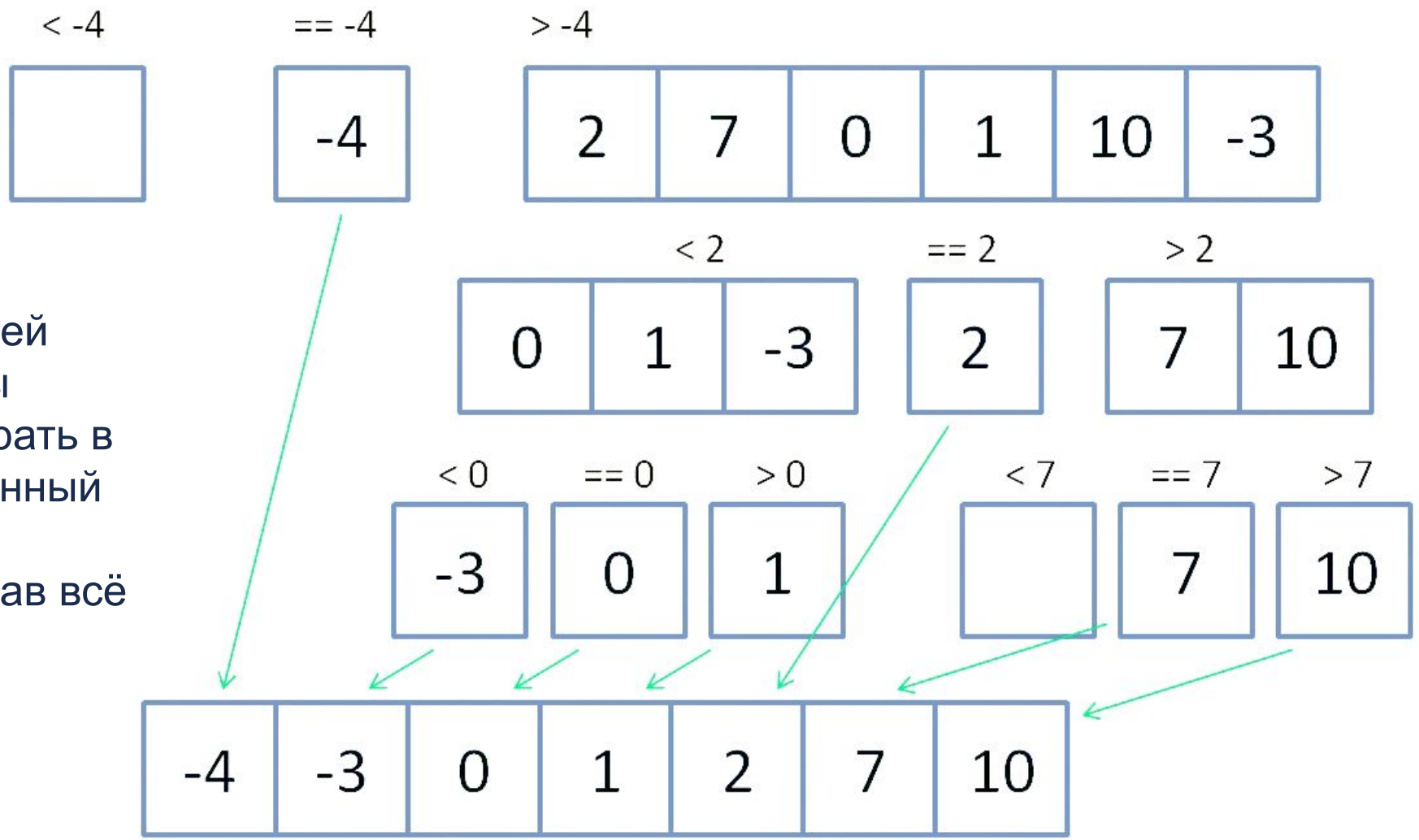
БЫСТРАЯ СОРТИРОВКА



На этом этапе заканчивается разбиение

БЫСТРАЯ СОРТИРОВКА

На следующей итерации мы должны собрать в отсортированный массив.
Просто собрав всё по порядку.



ВЫБОР ОПОРНОГО ЭЛЕМЕНТА

Правильный выбор опорного элемента может сильно повысить эффективность быстрой сортировки. В зависимости от реализации алгоритма есть разные способы выбора:

- Первый элемент — в первых версиях быстрой сортировки Хоар выбирал опорным первый элемент массива. Именно так он смог обрабатывать всю магнитную ленту за один проход. Однако, если значения выставлены изначально по возрастанию, то это при разбиении мы будем получать каждый раз новый массив, который уменьшается только на один элемент – это наихудший работы данного алгоритма.

- Средний элемент — тот, который физически стоит посередине массива.

- Медианный элемент — элемент, значение которого находится посередине между всеми значениями в массиве

- Выбор случайного – приводит к ускорению работы данного алгоритма.

Есть ещё много других техник выбора — они применяются, когда точно известно с какими массивами придётся работать: немного упорядоченными или когда всё вразнобой.

Варианты использования и применения Quicksort

- в различных организациях с целью сортировки различных данных, таких как сортировка файлов по имени/дате/цене, сортировка студентов по номеру их списка, сортировка профиля учетной записи по заданному идентификатору и т. д.

- алгоритм сортировки используется для **поиска информации , а поскольку быстрая сортировка является самым быстрым алгоритмом**, он широко используется как лучший способ поиска.

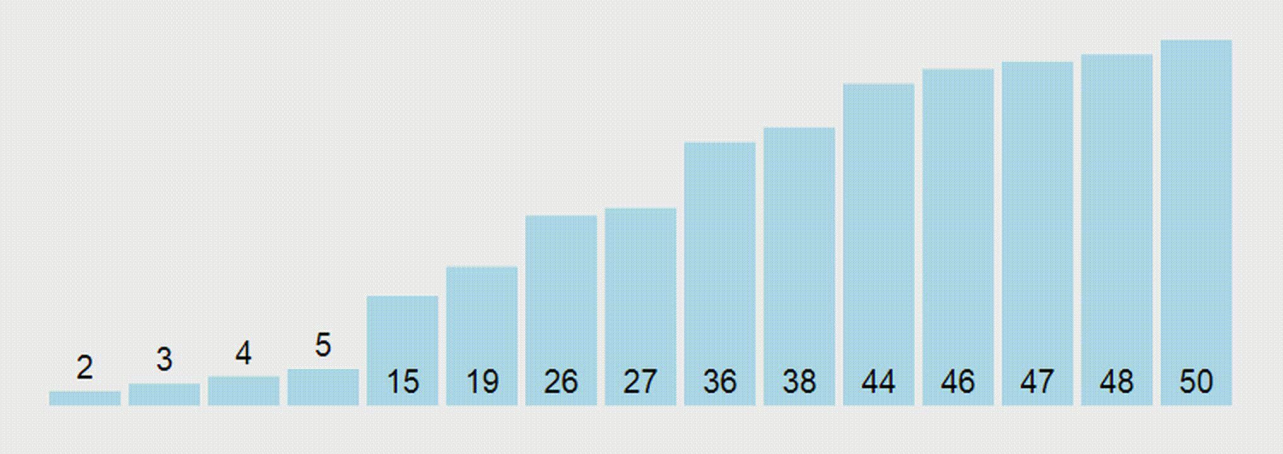
- используется везде, где не требуется **стабильная сортировка** .

- это сортировка на месте, которая не требует дополнительной **памяти**.

- используется в операционных исследованиях и событийно-ориентированном моделировании.

- если данные отсортированы, то поиск информации становится простым и эффективным.

БЫСТРАЯ СОРТИРОВКА



Сложность сортировки по времени:
Худшая O(n^2)
Средняя O(n * $\log_2 n$)
Лучшая O(n * 2$\log_2 n$)

```
def quicksort(alist, start, end):
    '''Sorts the list from indexes start to end - 1 inclusive.'''
    if end - start > 1:
        p = partition(alist, start, end)
        quicksort(alist, start, p)
        quicksort(alist, p + 1, end)


def partition(alist, start, end):
    pivot = alist[start]
    i = start + 1
    j = end - 1

    while True:
        while (i <= j and alist[i] <= pivot):
            i = i + 1
        while (i <= j and alist[j] >= pivot):
            j = j - 1

        if i <= j:
            alist[i], alist[j] = alist[j], alist[i]
        else:
            alist[start], alist[j] = alist[j], alist[start]
            return j


alist = input('Enter the list of numbers: ').split()
alist = [int(x) for x in alist]
quicksort(alist, 0, len(alist))
print('Sorted list: ', end='')
print(alist)
```

Практика. Быстрая сортировка

Задача: Сортировка большого списка

Сгенерируйте список случайных чисел большой длины (например, 10^4 элементов) в диапазоне от 1 до 10000.

Используя алгоритм быстрой сортировки отсортируйте этот список в обратном порядке.

Замерьте время выполнения сортировки и выведите его.

Вывести отсортированный список.

Практика. БЫСТРАЯ СОРТИРОВКА

1

Сгенерируйте список случайных чисел большой длины (например, 10^4 элементов) в диапазоне от 1 до 10000.

```
import random
from time import time
random.seed(42)
big_list = [random.randint(1,10000) for _ in range(10**4)]
print(big_list)
```

Практика. БЫСТРАЯ СОРТИРОВКА

2

Используя алгоритм быстрой сортировкой отсортируйте этот список в обратном порядке:

```
# Ф-я для сорт. в обр. порядке с исп. быстрой сортировки
def quick_sort_reverse(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        less = [x for x in arr[1:] if x <= pivot]
        greater = [x for x in arr[1:] if x > pivot]
        return quick_sort_reverse(greater) + [pivot] + quick_sort_reverse(less)
```

Практика. БЫСТРАЯ СОРТИРОВКА

3

Замерьте время выполнения сортировки и выведите его.

```
# Замер времени вып-я сор-ки быстрым поиском в обр. порядке
start_time = time.time()
sorted_list = quick_sort_reverse(big_list)
end_time = time.time()
```

4

Вывести отсортированный список

```
# Вывод времени выполнения
print(f"Время вып-я быстрой сор-ки в обр. п.: {end_time - start_time} секунд")
print(quick_sort_reverse(big_list))
```

Практика. Сортировка встроенной функцией

Задача: Сортировка большого списка

Сгенерируйте список случайных чисел большой длины (например, 10^4 элементов) в диапазоне от 1 до 10000.

Используя встроенную **функцию sorted()** отсортируйте этот список в обратном порядке.

Замерьте время выполнения сортировки и выведите его.

Вывести отсортированный список.

Практика. Функция sorted()

```
import random
import time
# Генерация большого списка случайных чисел
random.seed(42)  # Для воспроизводимости результатов
big_list = [random.randint(1, 10000) for _ in range(10**4)]
# Замер времени выполнения сортировки в обратном порядке
start_time = time.time()
sorted_list = sorted(big_list, reverse=True)
end_time = time.time()
# Вывод времени выполнения
print(f"Время выполнения сортировки: {end_time - start_time} секунд")
print(sorted_list)
```

Практика. Сортировка встроенной функцией

Задача: Сортировка большого списка

Сгенерируйте список случайных чисел большой длины (например, 10^4 элементов) в диапазоне от 1 до 10000.

Используя встроенную **list.sort()** с параметром **reverse=True** отсортируйте этот список в обратном порядке.

Замерьте время выполнения сортировки и выведите его.

Вывести отсортированный список.

Практика. Sort()

```
import random
import time

# Генерация большого списка случайных чисел
random.seed(42)  # Для воспроизводимости результатов
big_list = [random.randint(1, 10000) for _ in range(10**4)]

# Замер времени выполнения сортировки
start_time = time.time()
big_list.sort(reverse=True)
end_time = time.time()

# Вывод времени выполнения
print(f"Время выполнения сортировки: {end_time - start_time} секунд")
```

Практика. Итоги

Сортировка:	10**4
Пузырьком (reserve)	19,59
Выбором	8,25
Вставкой(reserve)	10,42
Быстрой сортировкой (reserve)	0,312
Sorted() (reserve)	0,0
Sort()(reserve)	0,0

Зачем изучать алгоритмы сортировки, если они работают медленнее встроенных функций?

Изучение алгоритмов сортировки важно по нескольким причинам, даже если встроенные функции сортировки в языке программирования работают быстрее в большинстве случаев:

Понимание основных принципов: Изучение алгоритмов сортировки позволяет понять основные принципы сортировки данных, как они работают и какие компоненты в них входят. Это фундаментальные знания, которые могут быть полезными при решении разнообразных задач в программировании.

Разработка и оптимизация: Понимание работы алгоритмов сортировки может помочь в разработке собственных алгоритмов для конкретных задач или оптимизации существующего кода. В некоторых случаях, если вы знаете характер данных и их распределение, вы можете разработать более эффективный алгоритм сортировки, чем встроенные функции.

Адаптация к специфическим условиям: Некоторые алгоритмы сортировки работают лучше в определенных условиях или при определенных ограничениях (например, сортировка подсчетом для сортировки целых чисел). Знание разных алгоритмов позволяет выбирать наиболее подходящий вариант для конкретной ситуации.

Собеседования и технические интервью: Вопросы о сортировке часто встречаются на собеседованиях и технических интервью для разработчиков. Знание различных алгоритмов сортировки может помочь вам успешно ответить на такие вопросы и продемонстрировать свои навыки программирования.

Учебные исследования: Если вы учите программирование в учебных целях или занимаетесь исследованиями, изучение алгоритмов сортировки поможет вам углубить свои знания в области алгоритмов и структур данных.