**1. Сортировка вставками (Insertion Sort)**

Этот алгоритм прост в реализации и хорошо работает на маленьких массивах или когда массив почти отсортирован. Он устойчив, что означает сохранение порядка одинаковых элементов. Однако он становится неэффективным для больших массивов из-за квадратичной временной сложности O(n²).

**Преимущества:**

* Простота реализации.
* Эффективна для маленьких массивов или когда массив почти отсортирован.
* Устойчивая сортировка (не меняет порядок равных элементов).

**Недостатки:**

* Неэффективна для больших массивов (квадратичная сложность(On^2)).

**Когда использовать:**

Для небольших наборов данных.

Когда нужно дополнить уже отсортированный массив несколькими новыми элементами.

Пример: Сортируем массив [5, 2, 9, 1]. Начинаем со второго элемента (2). Сравниваем его с первым (5) и вставляем перед ним, потому что 2 < 5. Продолжаем для 9 (уже на месте), затем вставляем 1 в начало массива.

**Пример 1**

def insertion\_sort(arr):  
 for i in range(1, len(arr)):  
 key = arr[i]  
 j = i - 1  
 while j >= 0 and key < arr[j]:  
 arr[j + 1] = arr[j]  
 j -= 1  
 arr[j + 1] = key  
 print(f"Шаг {i}: {arr}")   
  
# Массив для сортировки  
arr = [12, 11, 13, 5, 6]  
insertion\_sort(arr)

**Пошаговое объяснение:**

1. На каждом шаге мы начинаем с i-го элемента и сравниваем его с предыдущими элементами.
2. Если i-й элемент меньше, мы смещаем предыдущие элементы вперёд, чтобы освободить место.
3. Вставляем i-й элемент на правильное место.

**Пример 2**

def insertion\_sort\_verbose(arr):  
 for i in range(1, len(arr)):  
 current\_value = arr[i]  
 position = i  
  
 # Перемещаем элементы arr[0..i-1], которые больше, чем current\_value,  
 # на одну позицию впереди текущей позиции  
 while position > 0 and arr[position - 1] > current\_value:  
 print(f"Перемещаем {arr[position - 1]} в позицию {position}")  
 arr[position] = arr[position - 1]  
 position -= 1  
  
 # Вставляем текущее значение на правильную позицию  
 print(f"Вставляем {current\_value} на позицию {position}")  
 arr[position] = current\_value  
 print(f"Результат: {arr}\n")  
  
  
# Массив для сортировки  
arr = [22, 27, 16, 18, 13]  
insertion\_sort\_verbose(arr)

**Пример 3**

def insertion\_sort\_steps(arr):  
 for i in range(1, len(arr)):  
 key = arr[i]  
 j = i - 1  
 print(f"Вставляем {key}:")  
 while j >= 0 and arr[j] > key:  
 arr[j + 1] = arr[j]  
 j -= 1  
 print(f" {arr} (сдвигаем {arr[j + 1]})")  
 arr[j + 1] = key  
 print(f"После вставки: {arr}\n")  
  
# Массив для сортировки  
arr = [29, 10, 14, 37, 13]  
insertion\_sort\_steps(arr)

**Задания для домашнего задания**

**Задание 1**

Мы отсортировали массив [3, 7, 4, 9, 5, 2, 6, 1] с помощью сортировки вставками. Вот как менялся массив после каждого шага сортировки:

Шаг: Начальное состояние массива, элемент 7 уже на своём месте, так как он больше предыдущего элемента 3.

Шаг: Элемент 4 вставляется между 3 и 7.

Шаг: Элемент 9 уже на своём месте, так как он больше предыдущего элемента 7.

Шаг: Элемент 5 вставляется между 4 и 7.

Шаг: Элемент 2 вставляется в начало массива.

Шаг: Элемент 6 вставляется между 5 и 7.

Шаг: Элемент 1 вставляется в начало массива.

Конечный отсортированный массив: [1, 2, 3, 4, 5, 6, 7, 9].

**Задание 2**

Функция сортировки вставками, которая печатает массив после каждой вставки элемента, была вызвана с тем же массивом. Результаты идентичны заданию 1, так как алгоритм одинаков.

**Задание 3**

Была выполнена сортировка массива строк ["банан", "апельсин", "яблоко", "груша"] по алфавиту:

Шаг: Строка апельсин вставляется перед банан.

Шаг: Строка яблоко остаётся на своём месте, так как она больше предыдущей банан.

Шаг: Строка груша вставляется между банан и яблоко.

Конечный отсортированный массив строк: ["апельсин", "банан", "груша", "яблоко"].

**2. Пирамидальная сортировка (Heap Sort)**

**Преимущества:**

* Сложность всегда O(n log n).
* Не требует дополнительной памяти.

**Недостатки:**

* Неустойчивая сортировка.
* На практике часто медленнее, чем быстрая сортировка.

**Когда использовать:**

Когда нужна гарантированная сложность O(n log n), независимо от предварительного упорядочения данных.

Пример: Сортируем массив [3, 1, 4, 1, 5, 9, 2]. Создаём из него двоичную кучу, затем извлекаем максимальные элементы и формируем отсортированный массив.

**Пример 1: Сортировка простого списка**

def heapify(arr, n, i):  
 largest = i  
 l = 2 \* i + 1  
 r = 2 \* i + 2  
  
 if l < n and arr[i] < arr[l]:  
 largest = l  
  
 if r < n and arr[largest] < arr[r]:  
 largest = r  
  
 if largest != i:  
 arr[i], arr[largest] = arr[largest], arr[i]  
 heapify(arr, n, largest)  
  
def heapSort(arr):  
 n = len(arr)  
  
 for i in range(n // 2 - 1, -1, -1):  
 heapify(arr, n, i)  
  
 for i in range(n-1, 0, -1):  
 arr[i], arr[0] = arr[0], arr[i]  
 heapify(arr, i, 0)  
  
arr = [12, 11, 13, 5, 6, 7]  
heapSort(arr)  
print("Отсортированный массив:", arr)

Объяснение:

Этот код сортирует массив чисел с использованием пирамидальной сортировки. Функция heapify преобразует поддерево в двоичную кучу. Затем heapSort применяет heapify к каждому элементу, создавая кучу, после чего извлекает максимальные элементы, формируя отсортированный массив.

**Пример 2: Сортировка списка строк**

def heapify(arr, n, i):  
 largest = i  
 l = 2 \* i + 1  
 r = 2 \* i + 2  
  
 if l < n and arr[l] > arr[largest]:  
 largest = l  
  
 if r < n and arr[r] > arr[largest]:  
 largest = r  
  
 if largest != i:  
 arr[i], arr[largest] = arr[largest], arr[i]  
 heapify(arr, n, largest)  
  
def heapSort(arr):  
 n = len(arr)  
  
 for i in range(n, -1, -1):  
 heapify(arr, n, i)  
  
 for i in range(n-1, 0, -1):  
 arr[i], arr[0] = arr[0], arr[i]  
 heapify(arr, i, 0)  
  
arr = ["banana", "apple", "cherry", "date"]  
heapSort(arr)  
print("Отсортированный массив:", arr)

Объяснение:

Этот пример показывает, как сортировать список строк. Методика аналогична сортировке чисел, но здесь сравниваются строки. Пирамидальная сортировка эффективно упорядочивает строки по алфавиту.

**Пример 3: Сортировка списка объектов по определенному атрибуту**

class Student:  
 def \_\_init\_\_(self, name, grade):  
 self.name = name  
 self.grade = grade  
  
 def \_\_repr\_\_(self):  
 return f"{self.name}: {self.grade}"  
  
def heapify(arr, n, i):  
 largest = i  
 l = 2 \* i + 1  
 r = 2 \* i + 2  
  
 if l < n and arr[l].grade > arr[largest].grade:  
 largest = l  
  
 if r < n and arr[r].grade > arr[largest].grade:  
 largest = r  
  
 if largest != i:  
 arr[i], arr[largest] = arr[largest], arr[i]  
 heapify(arr, n, largest)  
  
def heapSort(arr):  
 n = len(arr)  
  
 for i in range(n, -1, -1):  
 heapify(arr, n, i)  
  
 for i in range(n-1, 0, -1):  
 arr[i], arr[0] = arr[0], arr[i]  
 heapify(arr, i, 0)  
  
students = [Student("John", 86), Student("Jane", 91), Student("Dave", 88)]  
heapSort(students)  
print("Студенты, отсортированные по оценкам:", students)

Объяснение:

Этот пример демонстрирует пирамидальную сортировку списка объектов. Здесь мы создаем класс Student с атрибутами name и grade. Сортировка выполняется по оценкам студентов.

**Задания для домашнего задания**

**Задание 1:**

Используйте приведенный выше код для сортировки массива [3, 1, 4, 1, 5, 9, 2]. Выведите отсортированный массив.

**Задание 2:**

Используйте приведенный выше код для сортировки списка названий фруктов. Попробуйте отсортировать этот список: ["orange", "apple", "grape", "banana"]. Выведите отсортированный массив.

**Задание 3:**

Создайте список из объектов с различными атрибутами (например, продукты с ценами) и используйте пирамидальную сортировку для упорядочивания их по одному из атрибутов. Выведите отсортированный список.

**3. Сортировка выбором (Selection Sort)**

**Преимущества:**

* Простота реализации.
* Минимальное количество обменов элементов.

**Недостатки:**

* Квадратичная сложность, медленная для больших массивов.

**Когда использовать:**

Когда важно количество обменов элементов, например, если обмены затратны.

Пример: Сортируем массив [64, 25, 12, 22, 11]. Находим минимальный элемент (11) и меняем его с первым. Повторяем для оставшихся элементов.

def selection\_sort(arr):  
 n = len(arr)  
 for i in range(n):  
 # Находим индекс минимального элемента в неотсортированной части  
 min\_index = i  
 for j in range(i + 1, n):  
 if arr[j] < arr[min\_index]:  
 min\_index = j  
  
 # Меняем местами найденный минимальный элемент с первым в неотсортированной части  
 arr[i], arr[min\_index] = arr[min\_index], arr[i]  
 return arr  
  
  
# Пример использования  
array = [64, 25, 12, 22, 11]  
sorted\_array = selection\_sort(array)  
print("Отсортированный массив:", sorted\_array)

Описание Алгоритма:

На каждом шаге алгоритма ищем минимальный элемент в неотсортированной части массива.

Меняем местами найденный минимальный элемент с первым элементом в неотсортированной части.

Повторяем до тех пор, пока не отсортируем весь массив.

**Задания для домашнего задания**

**Задание 1:** Отсортируйте массив [3, 1, 41, 59, 26, 53, 59] с помощью алгоритма сортировки выбором. Выведите каждый шаг алгоритма.

**Задание 2:** Модифицируйте алгоритм сортировки выбором так, чтобы он сортировал массив в обратном порядке. Примените его к массиву [8, 4, 10, 9, 5].

**Задание 3**: Напишите функцию, которая использует алгоритм сортировки выбором для нахождения k-го наименьшего элемента в массиве. Примените эту функцию для нахождения 3-го наименьшего элемента в массиве [7, 10, 4, 3, 20, 15].

**4. Сортировка слиянием (Merge Sort)**

**Преимущества:**

* Сложность всегда O(n log n).
* Устойчивая сортировка.

**Недостатки:**

* Требует дополнительной памяти для массивов.

**Когда использовать:**

Для больших данных, особенно когда работа с памятью не является ограничивающим фактором.

Пример: Сортируем массив [38, 27, 43, 3, 9, 82, 10]. Разделяем массив пополам до одного элемента в подмассиве, затем сливаем и

def merge\_sort(arr):  
 if len(arr) > 1:  
 mid = len(arr) // 2  
 L = arr[:mid]  
 R = arr[mid:]  
  
 merge\_sort(L)  
 merge\_sort(R)  
  
 i = j = k = 0  
  
 # Слияние двух половин  
 while i < len(L) and j < len(R):  
 if L[i] < R[j]:  
 arr[k] = L[i]  
 i += 1  
 else:  
 arr[k] = R[j]  
 j += 1  
 k += 1  
  
 # Проверка оставшихся элементов  
 while i < len(L):  
 arr[k] = L[i]  
 i += 1  
 k += 1  
  
 while j < len(R):  
 arr[k] = R[j]  
 j += 1  
 k += 1  
  
 return arr  
  
# Пример использования  
array = [38, 27, 43, 3, 9, 82, 10]  
sorted\_array = merge\_sort(array)  
print("Отсортированный массив:", sorted\_array)

Описание Алгоритма:

Разделить массив на две половины.

Рекурсивно сортировать каждую половину.

Слить две отсортированные половины в один отсортированный массив.

**Задания для домашнего задания**

**Задание 1:** Отсортируйте массив [29, 10, 14, 37, 13] с помощью сортировки слиянием. Выведите массив на каждом этапе слияния.

**Задание 2:** Модифицируйте алгоритм сортировки слиянием, чтобы он сортировал элементы в обратном порядке. Примените его к массиву [5, 2, 8, 9, 4, 7, 6].

**Задание 3:** Напишите функцию, которая использует алгоритм сортировки слиянием для сортировки только четных чисел в массиве, оставляя нечетные числа на своих местах. Примените эту функцию к массиву [7, 2, 9, 4, 3, 6, 8, 5].

**5. Быстрая сортировка (Quick Sort)**

**Преимущества:**

* Одна из самых быстрых на практике сортировок.
* Эффективна для больших данных.

**Недостатки:**

* Сложность в худшем случае O(n^2), но это редко встречается на практике с хорошим выбором опорного элемента.
* Неустойчивая сортировка.

**Когда использовать:**

Для больших массивов, когда не требуется устойчивость сортировки.

Пример: Сортируем массив [6, 5, 3, 1, 8, 7, 2, 4]. Выбираем опорный элемент (4), распределяем элементы вокруг него; меньшие слева, большие справа. Рекурсивно повторяем для подмассивов.

def quick\_sort(arr):  
 if len(arr) <= 1:  
 return arr  
 else:  
 pivot = arr[len(arr) // 2]  
 left = [x for x in arr if x < pivot]  
 middle = [x for x in arr if x == pivot]  
 right = [x for x in arr if x > pivot]  
 return quick\_sort(left) + middle + quick\_sort(right)  
  
# Пример использования  
array = [6, 5, 3, 1, 8, 7, 2, 4]  
sorted\_array = quick\_sort(array)  
print("Отсортированный массив:", sorted\_array)

Описание Алгоритма:

Выбрать опорный элемент из массива.

Разделить массив так, чтобы элементы меньше опорного оказались слева от него, а больше – справа.

Рекурсивно применить те же шаги к левому и правому подмассивам.

**Задания для домашнего задания**

**Задание 1:** Отсортируйте массив [34, 7, 23, 32, 5, 62] с помощью быстрой сортировки. Выведите массив после каждого разделения.

**Задание 2:** Модифицируйте алгоритм быстрой сортировки, чтобы опорный элемент выбирался случайным образом. Примените этот метод к массиву [12, 35, 99, 18, 76].

**Задание 3:** Напишите функцию, которая использует быструю сортировку для сортировки массива по убыванию. Примените её к массиву [5, 1, 9, 3, 7, 6, 8, 2, 4].

**6. Сортировка подсчётом (Counting Sort)**

**Преимущества:**

* Линейное время сортировки при условии, что диапазон чисел ограничен.
* Устойчивая сортировка.

**Недостатки:**

* Неэффективна для массивов с большим диапазоном чисел.
* Требует дополнительной памяти.

**Когда использовать:**

Для сортировки целых чисел, когда диапазон чисел не очень велик по сравнению с количеством элементов.

Пример: Сортируем массив [1, 4, 1, 2, 7, 5, 2]. Подсчитываем количество каждого числа и затем воссоздаём отсортированный массив, используя эти подсчёты.

def counting\_sort(arr):  
 max\_val = max(arr)  
 count\_arr = [0] \* (max\_val + 1)  
  
 # Подсчёт элементов  
 for num in arr:  
 count\_arr[num] += 1  
  
 # Воссоздание отсортированного массива  
 sorted\_arr = []  
 for num, count in enumerate(count\_arr):  
 sorted\_arr.extend([num] \* count)  
  
 return sorted\_arr  
  
# Пример использования  
array = [1, 4, 1, 2, 7, 5, 2]  
sorted\_array = counting\_sort(array)  
print("Отсортированный массив:", sorted\_array)

Описание Алгоритма:

Определить диапазон чисел в массиве.

Подсчитать, сколько раз каждое число встречается в массиве.

Воссоздать отсортированный массив, используя подсчёты.

**Задания для домашнего задания**

**Задание 1:** Отсортируйте массив [4, 2, 2, 8, 3, 3, 1] с помощью сортировки подсчётом. Выведите количество каждого числа в массиве.

**Задание 2:** Модифицируйте алгоритм сортировки подсчётом, чтобы он сортировал числа в обратном порядке. Примените его к массиву [5, 3, 5, 2, 8, 7, 9, 4, 6, 1].

**Задание 3:** Напишите функцию, которая использует сортировку подсчётом для сортировки символов в строке. Примените эту функцию к строке "countingsort".