**Классификация алгоритмов сортировки**

Алгоритмы сортировки можно классифицировать по различным критериям:

**1. По принципу работы:**

* **Сортировка обменами:**
  + **Пузырьковая сортировка:** Проходит по массиву несколько раз, меняя местами соседние элементы, если они стоят в неправильном порядке.
  + **Шейкерная сортировка:** Улучшенная версия пузырьковой сортировки, которая чередует направления проходов по массиву.
* **Сортировка выбором:**
  + **Сортировка выбором:** На каждом проходе находит минимальный (или максимальный) элемент и помещает его на свое место.
* **Сортировка вставками:**
  + **Сортировка вставками:** Последовательно вставляет каждый элемент в отсортированную часть массива.
* **Сортировка слиянием:**
  + **Сортировка слиянием:** Разделяет массив на две части, сортирует их рекурсивно, а затем объединяет в отсортированный массив.
* **Быстрая сортировка:**
  + **Быстрая сортировка:** Выбирает опорный элемент, разделяет массив на две части относительно опорного и рекурсивно сортирует их.
* **Сортировка деревом:**
  + **Сортировка деревом:** Строит бинарное дерево поиска из элементов массива и обходит его в порядке возрастания.

**2. По сложности:**

* **Временная сложность:**
  + **O(n²):** Пузырьковая сортировка, сортировка выбором, сортировка вставками.
  + **O(n log n):** Сортировка слиянием, быстрая сортировка, сортировка деревом.
* **Пространственная сложность:**
  + **O(1):** Пузырьковая сортировка, сортировка выбором, сортировка вставками.
  + **O(n):** Сортировка слиянием, сортировка деревом.
  + **O(log n):** Быстрая сортировка.

**3. По устойчивости:**

* **Устойчивая сортировка:** Сохраняет относительный порядок равных элементов (например, сортировка вставками, сортировка слиянием).
* **Неустойчивая сортировка:** Не сохраняет относительный порядок равных элементов (например, быстрая сортировка, сортировка выбором).

**4. По адаптивности:**

* **Адаптивная сортировка:** Эффективнее работает на частично отсортированных данных (например, сортировка вставками, шейкерная сортировка).
* **Неадаптивная сортировка:** Работает одинаково эффективно на всех типах данных (например, быстрая сортировка, сортировка слиянием).

**Теоретическое описание алгоритмов сортировки**

**1. Шейкерная сортировка:**

* **Принцип работы:** Шейкерная сортировка (Cocktail Sort) является улучшенной версией пузырьковой сортировки. Она проходит по массиву в двух направлениях: слева направо и справа налево, меняя местами соседние элементы, если они стоят в неправильном порядке. Это позволяет быстрее "всплывать" большим элементам в конец массива и "тонуть" маленьким элементам в начало.
* **Временная сложность:**
  + **Худший случай:** O(n²)
  + **Лучший случай:** O(n) (если массив уже отсортирован)
* **Пространственная сложность:** O(1)
* **Устойчивость:** Да
* **Адаптивность:** Да

**2. Сортировка деревом:**

* **Принцип работы:** Сортировка деревом (Tree Sort) строит бинарное дерево поиска из элементов массива, а затем обходит его в порядке возрастания.
* **Временная сложность:**
  + **Худший случай:** O(n²) (если дерево вырождается в список)
  + **Средний случай:** O(n log n)
* **Пространственная сложность:** O(n)
* **Устойчивость:** Да
* **Адаптивность:** Нет

**2. Сортировка деревом:**

**Достоинства и недостатки алгоритмов**

**1. Шейкерная сортировка:**

* **Достоинства:**
  + **Простая реализация.**
  + **Адаптивная: работает быстрее на частично отсортированных данных.**
  + **Устойчивая.**
* **Недостатки:**
  + **В худшем случае имеет квадратичную временную сложность.**
  + **Неэффективна для больших массивов.**

**2. Сортировка деревом:**

* **Достоинства:**
  + **Имеет среднюю временную сложность O(n log n).**
  + **Устойчивая.**
* **Недостатки:**
  + **Требует дополнительной памяти для хранения дерева.**
  + **В худшем случае (если дерево вырождается в список) имеет квадратичную временную сложность.**
  + **Неадаптивная: работает одинаково эффективно на всех типах данных.**

**Реализация алгоритмов сортировки**

def cocktail\_sort(A):  
 left = 0  
 right = len(A) - 1  
 swapped = True  
 while swapped:  
 swapped = False  
 for i in range(left, right):  
 if A[i] > A[i + 1]:  
 A[i], A[i + 1] = A[i + 1], A[i]  
 swapped = True  
 if not swapped:  
 break  
 swapped = False  
 right -= 1  
 for i in range(right, left, -1):  
 if A[i] < A[i - 1]:  
 A[i], A[i - 1] = A[i - 1], A[i]  
 swapped = True  
 left += 1

**2. Сортировка деревом:**

class TreeNode:  
 def \_\_init\_\_(self, key):  
 self.left = None  
 self.right = None  
 self.val = key  
  
def insert(root, key):  
 if root is None:  
 return TreeNode(key)  
 else:  
 if root.val < key:  
 root.right = insert(root.right, key)  
 else:  
 root.left = insert(root.left, key)  
 return root  
  
def inorder\_traversal(root, result):  
 if root:  
 inorder\_traversal(root.left, result)  
 result.append(root.val)  
 inorder\_traversal(root.right, result)  
  
def tree\_sort(A):  
 root = None  
 for key in A:  
 root = insert(root, key)  
 result = []  
 inorder\_traversal(root, result)  
 return result