**Дисциплина «Алгоритмы решения прикладных задач»**

**Рабочая тетрадь 6.**

**Алгоритмы сортировки массивов**

| **Теоретический материал** | |
| --- | --- |
| **Алгоритм сортировки** — это алгоритм для упорядочивания элементов в массиве.  По некоторым источникам, именно программа сортировки стала первой программой для вычислительных машин. Некоторые конструкторы ЭВМ, в частности разработчики EDVAC, называли задачу сортировки данных наиболее характерной нечисловой задачей для вычислительных машин. В 1945 году Джон фон Нейман для тестирования ряда команд для EDVAC разработал программы сортировки методом слияния. В том же году немецкий инженер Конрад Цузе разработал программу для сортировки методом простой вставки.  В 50-х – 60-х гг XX века было предложено множество различных алгоритмов сортировки: слияние с вставкой, обменная поразрядная сортировка, каскадное слияние и метод Шелла в 1959 году, многофазное слияние и вставки в дерево в 1960 году, осциллирующая сортировка и быстрая сортировка Хоара в 1962 году, пирамидальная сортировка Уильямса и обменная сортировка со слиянием Бэтчера в 1964 году. Появившиеся позже алгоритмы во многом являлись вариациями уже известных методов. Получили распространение адаптивные методы сортировки, ориентированные на более быстрое выполнение в случаях, когда входная последовательность удовлетворяет заранее установленным критериям.  Для реализации алгоритмов сортировки необходимо располагать инструментами, позволяющими произвести обмен значениями двум различным ячейкам массива. *Пример 1 демонстрирует, как можно организовать такой обмен (swap) с помощью средств стандартной библиотеки и с помощью собственной функции.*  Некоторые алгоритмы сортировки быстрее производят сортировку небольших массивов. Некоторые алгоритмы сортировки напротив более эффективны с большими массивами.  **Сортировка простыми обменами, сортировка пузырьком (англ. bubble sort)** — простой алгоритм сортировки. Алгоритм состоит из повторяющихся проходов по сортируемому массиву. За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется перестановка элементов. Проходы по массиву повторяются N-1 раз или до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает — массив отсортирован. При каждом проходе алгоритма по внутреннему циклу очередной наибольший элемент массива ставится на своё место в конце массива рядом с предыдущим «наибольшим элементом», а наименьший элемент перемещается на одну позицию к началу массива («всплывает» до нужной позиции, как пузырёк в воде — отсюда и название алгоритма).  **Сортировка перемешиванием, или Шейкерная сортировка, или двунаправленная (англ. Cocktail sort)** — разновидность пузырьковой сортировки. Анализируя метод пузырьковой сортировки, можно отметить два обстоятельства. Во-первых, если при движении по части массива перестановки не происходят, то эта часть массива уже отсортирована и, следовательно, её можно исключить из рассмотрения. Во-вторых, при движении от конца массива к началу минимальный элемент «всплывает» на первую позицию, а максимальный элемент сдвигается только на одну позицию вправо. Эти две идеи приводят к следующим модификациям в методе пузырьковой сортировки. Границы рабочей части массива (то есть части массива, где происходит движение) устанавливаются в месте последнего обмена на каждой итерации. Массив просматривается поочередно справа налево и слева направо.  **Сортировка вставками (англ. Insertion sort)** — алгоритм сортировки, в котором элементы входной последовательности просматриваются по одному, и каждый новый поступивший элемент размещается в подходящее место среди ранее упорядоченных элементов.    **Сортировка выбором**  Шаги алгоритма:   1. находим номер минимального значения в текущем списке 2. производим обмен этого значения со значением первой неотсортированной позиции (обмен не нужен, если минимальный элемент уже находится на данной позиции) 3. теперь сортируем хвост списка, исключив из рассмотрения уже отсортированные элементы   **Сортировка Шелла (англ. Shell sort)** — алгоритм сортировки, являющийся усовершенствованным вариантом сортировки вставками. Идея метода Шелла состоит в сравнении элементов, стоящих не только рядом, но и на определённом расстоянии друг от друга. Иными словами — это сортировка вставками с предварительными «грубыми» проходами.  При сортировке Шелла сначала сравниваются и сортируются между собой значения, стоящие один от другого на некотором расстоянии d. После этого процедура повторяется для некоторых меньших значений d, а завершается сортировка Шелла упорядочиванием элементов при d=1 (то есть обычной сортировкой вставками). Эффективность сортировки Шелла в определённых случаях обеспечивается тем, что элементы «быстрее» встают на свои места (в простых методах сортировки, например, пузырьковой, каждая перестановка двух элементов уменьшает количество инверсий в списке максимум на 1, а при сортировке Шелла это число может быть больше).  Среднее время работы алгоритма зависит от длин промежутков d, на которых будут находиться сортируемые элементы исходного массива ёмкостью N на каждом шаге алгоритма. Существует несколько подходов к выбору этих значений, простейший из которых – это первоначально используемая Шеллом последовательность длин промежутков: d=N/2, di=di-1/2, dk=1. Про другие способы выбора длин промежутков можно прочитать, например, на <https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%A8%D0%B5%D0%BB%D0%BB%D0%B0>  **Быстрая сортировка, сортировка Хоара (англ. quicksort)**, часто называемая qsort (по имени в стандартной библиотеке языка Си)  QuickSort является существенно улучшенным вариантом алгоритма сортировки с помощью прямого обмена, известного в том числе своей низкой эффективностью. Принципиальное отличие состоит в том, что в первую очередь производятся перестановки на наибольшем возможном расстоянии и после каждого прохода элементы делятся на две независимые группы (таким образом улучшение самого неэффективного прямого метода сортировки дало в результате один из наиболее эффективных улучшенных методов).  Общая идея алгоритма состоит в следующем:   * Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность (см. ниже). * Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующих друг за другом: «элементы меньшие опорного», «равные» и «большие»[[2]](https://ru.wikipedia.org/wiki/%D0%91%D1%8B%D1%81%D1%82%D1%80%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0#cite_note-2). * Для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.   На практике массив обычно делят не на три, а на две части: например, «меньшие опорного» и «равные и большие»; такой подход в общем случае эффективнее, так как упрощает алгоритм разделения | |
| **Пример 1** | |
| ***Задача:*** | |
|  | Написать на языке C++ программу, в которой для динамически заданного массива производится перестановка местами двух его произвольных элементов. Реализовать перестановку функцией стандартной библиотеки std::swap, а также написать собственную функцию. Сделать замеры времени выполнения функций swap для обеих случаев |
| ***Решение:*** | |
|  |  |
| ***Ответ:*** | |
|  |  |
| **Задание 1** | |
| ***Задача:*** | |
|  | Написать программу для сортировки массива пузырьковым алгоритмом. Здесь и далее во всех задачах массив сортируется по возрастанию. |
| ***Решение:*** | |
|  | def bubble\_sort(arr):  n = len(arr)  for i in range(n):  # Последние i элементов уже отсортированы, поэтому можно их пропустить  for j in range(0, n-i-1):  # Сравниваем соседние элементы  if arr[j] > arr[j+1]:  # Меняем местами, если нужно  arr[j], arr[j+1] = arr[j+1], arr[j]  # Пример использования  my\_array = [64, 25, 12, 22, 11]  bubble\_sort(my\_array)  print("Отсортированный массив:", my\_array) |
| ***Ответ:*** | |
|  |  |
| **Задание 2\*** | |
| ***Задача:*** | |
|  | Написать программу для сортировки массива шейкерным алгоритмом |
| ***Решение:*** | |
|  | def cocktail\_sort(arr):  n = len(arr)  swapped = True  start = 0  end = n-1  while (swapped == True):  # Сбросим флаг после каждого прохода  swapped = False  # Проход слева направо  for i in range(start, end):  if (arr[i] > arr[i + 1]):  arr[i], arr[i + 1] = arr[i + 1], arr[i]  swapped = True  # Если не было обменов, то массив уже отсортирован  if (swapped == False):  break  # Уменьшаем "край" массива, так как самый большой элемент уже находится в конце  end = end-1  # Сбросим флаг перед каждым проходом  swapped = False  # Проход справа налево  for i in range(end-1, start-1, -1):  if (arr[i] > arr[i + 1]):  arr[i], arr[i + 1] = arr[i + 1], arr[i]  swapped = True  # Увеличиваем "край" массива, так как самый маленький элемент уже находится в начале  start = start + 1  # Пример использования  my\_array = [64, 25, 12, 22, 11]  cocktail\_sort(my\_array)  print("Отсортированный массив:", my\_array) |
| ***Ответ:*** | |
|  |  |
| **Задание 3** | |
| ***Задача:*** | |
|  | Написать программу для сортировки массива алгоритмом сортировки вставками |
| ***Решение:*** | |
|  | 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  # Пример использования  my\_array = [64, 25, 12, 22, 11]  insertion\_sort(my\_array)  print("Отсортированный массив:", my\_array) |
| ***Ответ:*** | |
|  |  |
| **Задание 4** | |
| ***Задача:*** | |
|  | Написать программу для сортировки массива алгоритмом сортировки выбором |
| ***Решение:*** | |
|  | def selection\_sort(arr):  for i in range(len(arr)):  # Находим индекс минимального элемента в оставшейся части массива  min\_index = i  for j in range(i + 1, len(arr)):  if arr[j] < arr[min\_index]:  min\_index = j  # Меняем местами текущий элемент с минимальным элементом  arr[i], arr[min\_index] = arr[min\_index], arr[i]  # Пример использования  my\_array = [64, 25, 12, 22, 11]  selection\_sort(my\_array)  print("Отсортированный массив:", my\_array) |
| ***Ответ:*** | |
|  |  |
| **Задание 5\*** | |
| ***Задача:*** | |
|  | Написать программу для сортировки массива алгоритмом сортировки Шелла |
| ***Решение:*** | |
|  | def shell\_sort(arr):  n = len(arr)  gap = n // 2  while gap > 0:  for i in range(gap, n):  temp = arr[i]  j = i  while j >= gap and arr[j - gap] > temp:  arr[j] = arr[j - gap]  j -= gap  arr[j] = temp  gap //= 2  # Пример использования  my\_array = [64, 25, 12, 22, 11]  shell\_sort(my\_array)  print("Отсортированный массив:", my\_array) |
| ***Ответ:*** | |
|  |  |
| **Задание 6** | |
| ***Задача:*** | |
|  | Написать программу для сортировки массива алгоритмом быстрой сортировки |
| ***Решение:*** | |
|  | def quicksort(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 quicksort(left) + middle + quicksort(right)  # Пример использования  my\_array = [64, 25, 12, 22, 11]  sorted\_array = quicksort(my\_array)  print("Отсортированный массив:", sorted\_array) |
| ***Ответ:*** | |
| **Задание 7\*** | |
| ***Задача:*** | |
|  | Оформить алгоритмы сортировки из заданий 1-6 в виде функций. Сделать замеры времени выполнения этих функций для одного и того же исходного массива (маленького и большого размера). |
| ***Решение:*** | |
|  | Пузырьковая:  def bubble\_sort(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]  Шейкерная:  def cocktail\_sort(arr):  n = len(arr)  swapped = True  start = 0  end = n-1  while swapped:  swapped = False  for i in range(start, end):  if arr[i] > arr[i + 1]:  arr[i], arr[i + 1] = arr[i + 1], arr[i]  swapped = True  if not swapped:  break  swapped = False  for i in range(end-1, start-1, -1):  if arr[i] > arr[i + 1]:  arr[i], arr[i + 1] = arr[i + 1], arr[i]  swapped = True  start = start + 1  end = end - 1  Вставками:  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  Выбором:  def selection\_sort(arr):  for i in range(len(arr)):  min\_index = i  for j in range(i + 1, len(arr)):  if arr[j] < arr[min\_index]:  min\_index = j  arr[i], arr[min\_index] = arr[min\_index], arr[i]  Школа:  def shell\_sort(arr):  n = len(arr)  gap = n // 2  while gap > 0:  for i in range(gap, n):  temp = arr[i]  j = i  while j >= gap and arr[j - gap] > temp:  arr[j] = arr[j - gap]  j -= gap  arr[j] = temp  gap //= 2  Быстрая:  def quicksort(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 quicksort(left) + middle + quicksort(right)  #Замер времени  import time  import random  # Функция для замера времени выполнения  def measure\_time(func, array):  start\_time = time.time()  func(array)  end\_time = time.time()  return end\_time - start\_time  # Создадим массивы для тестирования (маленький и большой)  small\_array = random.sample(range(1, 100), 10)  large\_array = random.sample(range(1, 1000), 100)  # Замер времени выполнения для каждого алгоритма на маленьком массиве  print("Маленький массив:")  print("Bubble Sort:", measure\_time(bubble\_sort, small\_array.copy()))  print("Cocktail Sort:", measure\_time(cocktail\_sort, small\_array.copy()))  print("Insertion Sort:", measure\_time(insertion\_sort, small\_array.copy()))  print("Selection Sort:", measure\_time(selection\_sort, small\_array.copy()))  print("Shell Sort:", measure\_time(shell\_sort, small\_array.copy()))  print("Quick Sort:", measure\_time(quicksort, small\_array.copy()))  # Замер времени выполнения для каждого алгоритма на большом массиве  print("\nБольшой массив:")  print("Bubble Sort:", measure\_time(bubble\_sort, large\_array.copy()))  print("Cocktail Sort:", measure\_time(cocktail\_sort, large\_array.copy()))  print("Insertion Sort:", measure\_time(insertion\_sort, large\_array.copy()))  print("Selection Sort:", measure\_time(selection\_sort, large\_array.copy()))  print("Shell Sort:", measure\_time(shell\_sort, large\_array.copy()))  print("Quick Sort:", measure\_time(quicksort, large\_array.copy())) |
| ***Ответ:*** | |
|  |  |
| **Задание 8\*** | |
| ***Задача:*** | |
|  | Алгоритм вставки допускает простое и очень эффективное улучшение. Ключевой момент вставки – это поиск правильной позиции. Заметим, что если речь идет о поиске нового места для элемента с номером L, то, значит, массив с 1 по L-1 элемент уже упорядочен. Поиск правильной позиции можно легко выполнить методом половинного деления. Напишите реализацию алгоритма вставки, в которой простой перебор заменен методом половинного деления. Сравните время работы алгоритмов. |
| ***Решение:*** | |
|  | def binary\_insertion\_sort(arr):  for i in range(1, len(arr)):  key = arr[i]  left, right = 0, i - 1  # Используем метод половинного деления для поиска правильной позиции  while left <= right:  mid = (left + right) // 2  if key < arr[mid]:  right = mid - 1  else:  left = mid + 1  # Сдвигаем элементы вправо, чтобы вставить key на правильное место  for j in range(i, left, -1):  arr[j] = arr[j - 1]  arr[left] = key  # Пример использования  my\_array = [64, 25, 12, 22, 11]  binary\_insertion\_sort(my\_array)  print("Отсортированный массив:", my\_array)  # Сравним время работы  import time  import random  # Функция для замера времени выполнения  def measure\_time(func, array):  start\_time = time.time()  func(array)  end\_time = time.time()  return end\_time - start\_time  # Создадим массивы для тестирования (маленький и большой)  small\_array = random.sample(range(1, 100), 10)  large\_array = random.sample(range(1, 1000), 100)  # Замер времени выполнения для обычной сортировки вставками на маленьком массиве  print("Обычная сортировка вставками (маленький массив):", measure\_time(insertion\_sort, small\_array.copy()))  # Замер времени выполнения для улучшенной сортировки вставками на маленьком массиве  print("Улучшенная сортировка вставками (маленький массив):", measure\_time(binary\_insertion\_sort, small\_array.copy()))  # Замер времени выполнения для обычной сортировки вставками на большом массиве  print("Обычная сортировка вставками (большой массив):", measure\_time(insertion\_sort, large\_array.copy()))  # Замер времени выполнения для улучшенной сортировки вставками на большом массиве  print("Улучшенная сортировка вставками (большой массив):", measure\_time(binary\_insertion\_sort, large\_array.copy())) |
| ***Ответ:*** | |
|  |  |
|  |  |