Лабораторная работа 1. Методы сортировки.

Выполнил: Щитов В.М.
Группа: БВТ1903
Вариант: 22

Задание.

Реализовать метод сортировки Шелла строк числовой матрицы в соответствии с индивидуальным заданием. Добавить реализацию быстрой сортировки (quicksort). Оценить время работы каждого алгоритма сортировки и сравнить его со временем стандартной функции сортировки, используемой в Python.

Необходимые библиотеки.

Для реализации двумерного массива (матрицы) можно использовать библиотеку **NumPy**. Для замеров времени выполнения алгоритмов используется модуль **time** стандартной библиотеки языка.

In [1]:
```
# Импортирование библиотеки
import numpy as np
import time
```

In [2]:
```
# Реализация матрицы с помощью двумерного массива NumPy
matrix = np.array([ [1, 3, 2, 28, 19, 32, 7],
                    [4, 18, 6, 5, 2, 22, 11],
                    [13, 8, 24, 48, 15, 10, 9],
                    [76, 62, 89, 30, 60, 93, 57],
                    [83, 75, 26, 10, 76, 42, 10],
                    [69, 88, 24, 38, 71, 92, 9] ], float)
print(matrix)
```

```
[[ 1.  3.  2. 28. 19. 32.  7.]
 [ 4. 18.  6.  5.  2. 22. 11.]
 [13.  8. 24. 48. 15. 10.  9.]
 [76. 62. 89. 30. 60. 93. 57.]
 [83. 75. 26. 10. 76. 42. 10.]
 [69. 88. 24. 38. 71. 92.  9.]]
```

In [3]:
```
# Функция сортировки Шелла для каждой строки матрицы
def shell_sort(row):
    last_index = len(row) - 1
    step = len(row) // 2
    sort_row = row.copy()
    while step > 0:
        for i in range(step, last_index + 1, 1):
            j = i
            delta = j - step
            while delta >= 0 and sort_row[delta] > sort_row[j]:
                sort_row[delta], sort_row[j] = sort_row[j], sort_row[delta]
                j = delta
                delta = j - step
        step //= 2
    return sort_row
```

In [4]:
```
# Функция возвращает индекс опорного элемента
def partition(row, low, high):
    x = row[high]
    i = low - 1
    j = low
    tmp = 0
    while j < high:
        if row[j] <= x:
            i = i + 1
            tmp = row[i]
            row[i] = row[j]
            row[j] = tmp
        j = j + 1
    tmp = row[high]
    row[high] = row[i + 1]
    row[i + 1] = tmp
    return i + 1

# Функция быстрой сортировки
def quicksort(row, low, high):
    if low < high:
        p = partition(row, low, high)
        quicksort(row, low, p - 1)
        quicksort(row, p + 1, high)
```

In [5]:
```
# Функция, формирующая отсортированную матрицу
def using_sort(matrix, sort_type = 'none'):
    sort_matrix = np.zeros(matrix.shape)
    algo_time = 0;

    # Если выбрана сортировка Шелла
    if sort_type == 'shell':
        tic = time.perf_counter_ns()
        for i in range (0, len(matrix)):
            sort_matrix[i] = shell_sort(matrix[i,:])
        toc = time.perf_counter_ns()
        algo_time = toc - tic

    # Если выбрана быстрая сортировка
    elif sort_type == 'quick':
        sort_matrix = matrix.copy()
        tic = time.perf_counter_ns()
        for i in range (0, len(matrix)):
            quicksort(sort_matrix[i,:], 0, sort_matrix.shape[1] - 1)
        toc = time.perf_counter_ns()
        algo_time = toc - tic

    # В противном случае выбирается стандартная функция сортировки NumPy
    else:
        tic = time.perf_counter_ns()
        for i in range (0, len(matrix)):
            sort_matrix[i] = np.sort(matrix[i,:])
        toc = time.perf_counter_ns()
        algo_time = toc - tic

    # Вывод результатов на экран и их возвращение функцией
    print('Время выполнения сортировки (в милисекундах): ', algo_time / (pow(10, 6)))
    print('Результат:\n', sort_matrix, "\n")
    return sort_matrix
```

In [6]:
```
# Вызов каждого метода сортировки с выводом времени выполнения и отсортированной матрицы

# Сортировка Шелла
print('Сортировка Шелла')
a = using_sort(matrix, 'shell')

# Быстрая сортировка
print('Быстрая сортировка')
a = using_sort(matrix, 'quick')

# Стандартная функция сортировки
print('Стандартная функция сортировки NumPy')
a = using_sort(matrix)
```

```
Сортировка Шелла
Время выполнения сортировки (в милисекундах):  0.190784
Результат:
 [[ 1.  2.  3.  7. 19. 28. 32.]
 [ 2.  4.  5.  6. 11. 18. 22.]
 [ 8.  9. 10. 13. 15. 24. 48.]
 [30. 57. 60. 62. 76. 89. 93.]
 [10. 10. 26. 42. 75. 76. 83.]
 [ 9. 24. 38. 69. 71. 88. 92.]]

Быстрая сортировка
Время выполнения сортировки (в милисекундах):  0.122561
Результат:
 [[ 1.  2.  3.  7. 19. 28. 32.]
 [ 2.  4.  5.  6. 11. 18. 22.]
 [ 8.  9. 10. 13. 15. 24. 48.]
 [30. 57. 60. 62. 76. 89. 93.]
 [10. 10. 26. 42. 75. 76. 83.]
 [ 9. 24. 38. 69. 71. 88. 92.]]

Стандартная функция сортировки NumPy
Время выполнения сортировки (в милисекундах):  0.187162
Результат:
 [[ 1.  2.  3.  7. 19. 28. 32.]
 [ 2.  4.  5.  6. 11. 18. 22.]
 [ 8.  9. 10. 13. 15. 24. 48.]
 [30. 57. 60. 62. 76. 89. 93.]
 [10. 10. 26. 42. 75. 76. 83.]
 [ 9. 24. 38. 69. 71. 88. 92.]]
```