

Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт цифрового развития
Кафедра инфокоммуникаций

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ №10
дисциплины «Алгоритмизация»

Выполнил:
Болуров Ислам Расулович
2 курс, группа ИВТ-б-о-22-1,
09.03.01 «Информатика и
вычислительная техника»,
направленность (профиль)
«Программное обеспечение средств
вычислительной техники и
автоматизированных систем», очная
форма обучения

(подпись)

Руководитель практики:
Воронкин Р.А., доцент кафедры
инфокоммуникаций

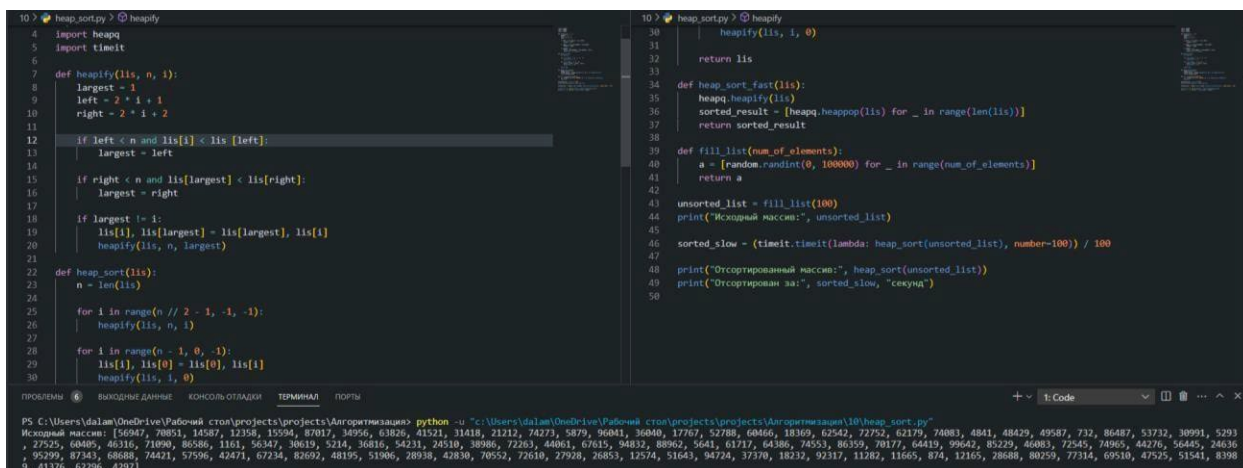
(подпись)

Отчет защищен с оценкой _____ Дата защиты _____

Ставрополь, 2024 г.

Порядок выполнения работы:

1. Написал программу поиска элемента в массиве, автоматического заполнения массива, расчёта тысячи точек, показывающих время поиска элемента в массиве в худшем и среднем случае, вывода графиков, составленных из этих точек, и подсчета корреляции:



```
10 > heap.sort.py ? heapify
4 import heapq
5 import timeit
6
7 def heapify(lis, n, i):
8     largest = i
9     left = 2 * i + 1
10    right = 2 * i + 2
11
12    if left < n and lis[i] < lis[left]:
13        largest = left
14
15    if right < n and lis[largest] < lis[right]:
16        largest = right
17
18    if largest != i:
19        lis[i], lis[largest] = lis[largest], lis[i]
20        heapify(lis, n, largest)
21
22 def heap_sort(lis):
23     n = len(lis)
24
25     for i in range(n // 2 - 1, -1, -1):
26         heapify(lis, n, i)
27
28     for i in range(n - 1, 0, -1):
29         lis[i], lis[0] = lis[0], lis[i]
30         heapify(lis, i, 0)
31
32     return lis
33
34 def heap_sort_fast(lis):
35     heapq.heapify(lis)
36     sorted_result = [heapq.heappop(lis) for _ in range(len(lis))]
37     return sorted_result
38
39 def fill_list(num_of_elements):
40     a = [random.randint(0, 100000) for _ in range(num_of_elements)]
41     return a
42
43 unsorted_list = fill_list(100)
44 print("Исходный массив:", unsorted_list)
45
46 sorted_slow = (timeit.timeit(lambda: heap_sort(unsorted_list), number=100)) / 100
47
48 print("Отсортированный массив:", heap_sort(unsorted_list))
49 print("Отсортирован на:", sorted_slow, "секунд")
50
```

PS C:\Users\dalame\OneDrive\Рабочий стол\projects\projects\Алгоритмы\10\heap_sort.py

Исходный массив: [56947, 78051, 14587, 12358, 15594, 87017, 34956, 63826, 41521, 31418, 21212, 74273, 5879, 96041, 36040, 17767, 52788, 60466, 18369, 62542, 72752, 62179, 74083, 4841, 48429, 49587, 732, 86487, 53732, 30991, 5293, 27525, 60405, 46316, 71090, 86586, 1161, 56347, 30619, 5214, 36816, 54231, 24510, 38986, 72263, 44061, 67615, 94832, 88962, 5641, 61717, 64386, 74553, 86359, 70177, 64419, 99642, 85229, 46083, 72545, 74965, 44276, 56445, 24636, 95299, 87343, 68688, 74421, 57596, 42471, 67234, 62692, 48195, 51906, 28938, 42830, 70552, 72618, 27928, 26853, 12574, 51643, 94724, 37370, 18232, 92317, 11282, 11665, 874, 12165, 28688, 80259, 77314, 69510, 47525, 51541, 8398, 41376, 62296, 4297]

Рисунок 1. Код и результат неоптимизированного алгоритма heapsort

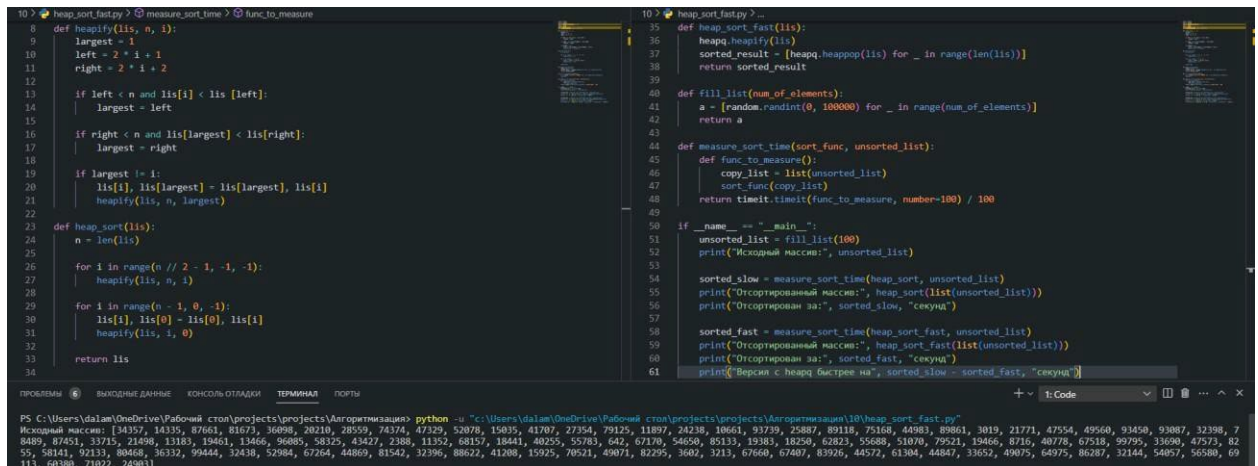
Таблица 1. Сравнение алгоритма Heap Sort с Quick Sort и Merge Sort

| Характеристика | Heap Sort | Quick Sort | Merge Sort |
|---------------------|------------------------|---|---------------|
| Сложность времени | $O(n \log n)$ | $O(n^2)$ в худшем случае, $O(n \log n)$ в среднем | $O(n \log n)$ |
| Сложность по памяти | $O(1)$ или $O(\log n)$ | $O(\log n)$ в среднем | $O(n)$ |
| Лучший случай | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ |
| Худший случай | $O(n \log n)$ | $O(n^2)$ | $O(n \log n)$ |
| Средний случай | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ |

Heapsort не требует доп. память, занимает меньше всех места, но считается нестабильным. Quick Sort медленный в худшем случае, занимает больше места, требует доп. память и является нестабильным. Merge Sort стабилен, быстр, не требует доп памяти, но имеет наибольшую сложность по памяти.

2. Произвел оптимизацию алгоритма при помощи встроенной

библиотеки heapq:



```
10 > heap_sort_fast.py > measure_sort_time > func_to_measure
8 def heapify(lis, n, i):
9     largest = i
10    left = 2 * i + 1
11    right = 2 * i + 2
12
13    if left < n and lis[i] < lis[left]:
14        largest = left
15
16    if right < n and lis[largest] < lis[right]:
17        largest = right
18
19    if largest != i:
20        lis[i], lis[largest] = lis[largest], lis[i]
21        heapify(lis, n, largest)
22
23 def heap_sort(lis):
24     n = len(lis)
25
26     for i in range(n // 2 - 1, -1, -1):
27         heapify(lis, n, i)
28
29     for i in range(n - 1, 0, -1):
30         lis[i], lis[0] = lis[0], lis[i]
31         heapify(lis, i, 0)
32
33     return lis
34
35
36 heap_sort_fast.py ~
37 def heap_sort_fast(lis):
38     sorted_result = [heapq.heappop(lis) for _ in range(len(lis))]
39     return sorted_result
40
41 def fill_list(num_of_elements):
42     a = [random.randint(0, 100000) for _ in range(num_of_elements)]
43     return a
44
45 def measure_sort_time(sort_func, unsorted_list):
46     def func_to_measure():
47         copy_list = list(unsorted_list)
48         sort_func(copy_list)
49     return timeit.timeit(func_to_measure, number=100) / 100
50
51 if __name__ == "__main__":
52     unsorted_list = fill_list(1000)
53     print("Исходный массив:", unsorted_list)
54
55     sorted_slow = measure_sort_time(heap_sort, unsorted_list)
56     print("Отсортированный массив:", heap_sort(list(unsorted_list)))
57     print("Отсортирован за:", sorted_slow, "секунд")
58
59     sorted_fast = measure_sort_time(heap_sort_fast, unsorted_list)
60     print("Отсортированный массив:", heap_sort_fast(list(unsorted_list)))
61     print("Отсортирован за:", sorted_fast, "секунд")
62     print("Версия с heapq быстрее на:", sorted_slow - sorted_fast, "секунд")
63
64
65 PS C:\Users\dalam\OneDrive\Рабочий стол\projects\projects\Алгоритмизация> python -u "C:\Users\dalam\OneDrive\Рабочий стол\projects\projects\Алгоритмизация\10\heap_sort_fast.py"
Исходный массив: [14157, 14335, 87661, 81673, 36898, 20210, 28559, 74374, 47329, 52078, 15035, 41707, 27354, 79125, 11897, 24238, 18661, 93739, 25887, 89118, 75168, 44983, 89861, 3019, 21771, 47554, 49560, 93408, 93087, 32398, 7
8489, 87451, 31715, 21498, 13183, 19461, 13466, 96085, 58325, 43427, 2288, 11352, 68157, 18441, 40255, 55783, 642, 67110, 54659, 83133, 18385, 18289, 62823, 55688, 51070, 78221, 19466, 8716, 48778, 67518, 29795, 33698, 45713, 82
55, 58141, 92133, 80468, 36332, 99444, 32438, 52984, 67264, 44869, 81542, 32396, 88622, 41208, 15925, 70521, 49071, 82295, 3602, 3213, 67668, 67407, 83926, 44572, 61304, 44847, 33652, 49075, 64975, 86287, 32144, 54857, 56580, 69
113, 60380, 71022, 24903]
```

Рисунок 2. Оптимизированный алгоритм heapsort

3. Применение в реальной жизни:

Системы с ограниченной памятью: Эффективен там, где важно минимизировать использование дополнительной памяти.

Приоритетные очереди: Используется для эффективного управления данными с приоритетом, например в обработке событий.

Потоковая обработка: Подходит для сортировки данных в реальном времени, когда данные постоянно поступают.

Базы данных: Помогает в сортировке больших объемов данных вне основной памяти, через внешнюю сортировку.

Вычислительные задачи с таймингом: Хорош для задач в реальном времени, где важна предсказуемость времени выполнения операций.

Heap Sort выбирают из-за надёжности и предсказуемости, когда стоит избегать риска существенного замедления из-за худшего случая выполнения, как, например, у Quick Sort. Также он полезен, если требуется сортировать данные без дополнительного пространства, например, при выполнении сортировки прямо на физических носителях или в ситуациях с ограниченной доступной памятью.

Анализ сложности:

Добавил рисование графиков зависимости времени сортировки с помощью оптимизированного и не оптимизированного heapsort от количества

элементов в массиве.

Поскольку Heap Sort может выполняться in-place, для его работы теоретически не требуется дополнительное пространство кроме самого массива, который сортируется. При этом методе сортировки:

- Не требуется выделять дополнительный массив для разделения данных.
- Манипуляции с элементами осуществляются в пределах того же массива.
- Требуется ограниченное количество переменных для хранения индексов и временных значений в процессе выполнения алгоритма.

Heap Sort возможно реализовать без рекурсии, при этом алгоритм получается с чистой пространственной сложностью $O(1)$, то есть он будет занимать константное дополнительное пространство, не зависимо от размеров данных. Эта модификация использует только циклы и не требует дополнительной памяти для рекурсивного стека, что делает пространственную сложность чисто константной.

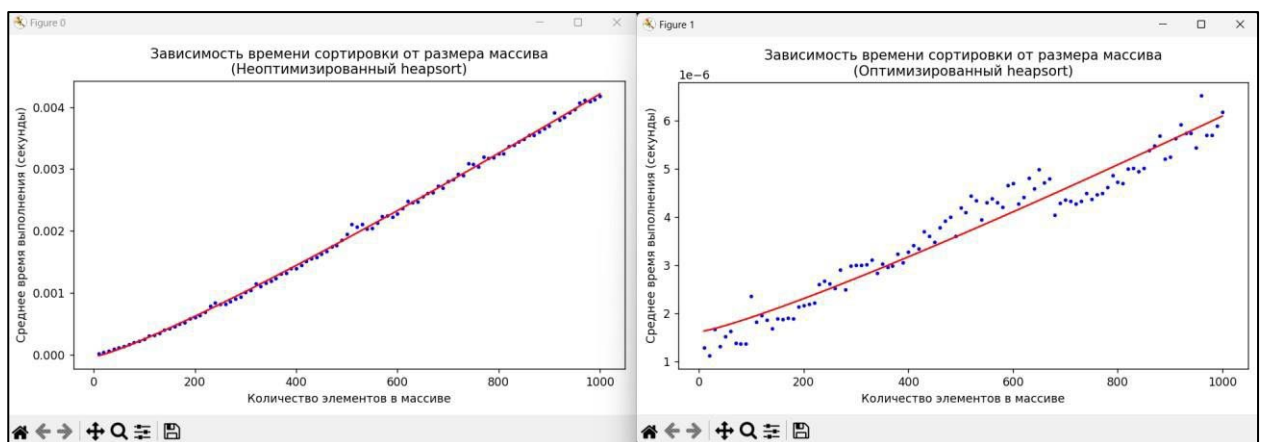


Рисунок 3. Графики зависимости времени сортировки с помощью оптимизированного и не оптимизированного heapsort от количества элементов в массиве

```
10 > task_6.py > ...
11 amount_of_data = 100 # Количество точек
12 and = (amount_of_data + 1) * 10
13 median_time = 0
14 graph_stuff = [1 for i in range(10, and, 10)]
15 xlabel = "Количество элементов в массиве"
16 ylabel = "Среднее время выполнения (секунды)"
17
18 def heapsort(lis, n, i):
19     largest = i
20     left = 2 * i + 1
21     right = 2 * i + 2
22
23     if left < n and lis[left] < lis[largest]:
24         largest = left
25
26     if right < n and lis[right] < lis[largest]:
27         largest = right
28
29     if largest != i:
30         lis[i], lis[largest] = lis[largest], lis[i]
31         heapsort(lis, n, largest)
32
33 def heap_sort(lis):
34     n = len(lis)
35
36     for i in range(n // 2 - 1, -1, -1):
37         heapsort(lis, n, i)
38
39     for i in range(n - 1, 0, -1):
40         lis[i], lis[0] = lis[0], lis[i]
41         heapsort(lis, i, 0)
42
43     return lis
44
45 def heap_sort_fast(lis):
46     heapsort(lis)
47     sorted_result = [heapq.heappop(lis) for _ in range(len(lis))]
48     return sorted_result
49
50 def fill_list(num_of_elements):
51     a = [random.randint(0, 100000) for _ in range(num_of_elements)]
52     return a
53
54 def measure_sort_time(sort_func, unsorted_list):
55     def func_to_measure():
56         # Сортируем массив, чтобы не влиять на оригинальный список
57         copy_list = list(unsorted_list)
58         sort_func(copy_list)
59         return timeit.timeit(func_to_measure, number=100) / 100
60
61     n_log_n_model(x, n, b):
62         return a * x * np.log(x) + b
63
64     def line(name, time, graph_index):
65         plt.figure(graph_index).set_figsize(8)
66         plt.title(f"Зависимость времени сортировки от размера массива n({name})")
67         plt.xlabel(xlabel)
68         plt.ylabel(ylabel)
69         plt.plot(time.values())
70         plt.tight_layout()
71         plt.grid(False)
72
73     x_data = np.array(graph_stuff)
74     y_data = np.array(list(time.values()))
75
76     params, _ = curve_fit(n_log_n_model, x_data, y_data)
77
78     a_fit, b_fit = params
79     print(f"Коэффициенты уравнения: {name}: a = {a_fit}, b = {b_fit}")
80
81     x_fit = np.linspace(min(x_data), max(x_data), 100)
82     y_fit = n_log_n_model(x_fit, *params)
83
84     plt.plot(x_fit, y_fit, "r-", label=f"n log n Fit")
85
86     plt.scatter(graph_stuff, time.values(), s=5, c="blue")
87     plt.tight_layout()
88
89 def results(name, func, graph_index):
90     for i in range(10, and, 10):
91         a = fill_list(i)
92         median_time[i] = timeit.timeit(lambda: func(a),
93                                     number=100) / 100
94
95     line(name, median_time, graph_index)
96
97 if __name__ == "__main__":
98     unsorted_list = fill_list(200)
99
100 sorted_time = measure_sort_time(heap_sort, unsorted_list)
101 print("Среднее время сортировки heap_sort: ", sorted_time, "сек")
102 sorted_fast = measure_sort_time(heap_sort_fast, unsorted_list)
103 print("Среднее время сортировки c heapq: ", sorted_fast, "сек")
104 print("Разница c heapq: ", sorted_time - sorted_fast, "сек")
105
106 PROBLEMY 6 ВЫХОДНЫЕ ДАННЫЕ КОНСОЛЬ ОТЛАДКИ ТЕРМИНАЛ ПОРТЫ
107
108 Сортирован heapsort с heapq за: 6.833400000000000e-05 сек
109 Разрешение с heapq быстрее на 0.000000000000000e+00 сек
110 Коэффициенты уравнения (Потенциальный heapsort): a = 6.144304091306450e-07, b = -3.296334370577808e-05
111 Коэффициенты уравнения (Отсортированный heapsort): a = 6.464182701956654e-10, b = 1.622077980115460e-06
```

Рисунок 4. Полный код с двумя алгоритмами heapsort и выводом графиков скорости их сортировки

4. Решение задания:

```
10 > task_6.py > ...
1 import heapq
2 import random
3
4
5 def print_ascending_sums(A, B):
6
7     # Сортируем оба массива
8     A.sort()
9     B.sort()
10
11     # min-heap хранит выражения вида (сумма, индекс в A, индекс в B)
12     heap = [(A[i] + B[0], i, 0) for i in range(len(A))]
13     heapq.heapify(heap)
14
15     # Вывод суммы в возрастающем порядке
16     for _ in range(len(A)**2):
17         sum, i, j = heapq.heappop(heap)
18         print(sum, end=' ')
19         if j + 1 < len(B):
20             heapq.heappush(heap, (A[i] + B[j+1], i, j+1))
21
22
23 def fill_list(num_of_elements):
24     a = [random.randint(1, 10) for _ in range(num_of_elements)]
25     return a
26
27
28 n = 5
29 A = fill_list(n)
30 B = fill_list(n)
31 print(sorted(A), "\n", sorted(B))
32 print_ascending_sums(A, B)
33
34 PROBLEMY 6 ВЫХОДНЫЕ ДАННЫЕ КОНСОЛЬ ОТЛАДКИ ТЕРМИНАЛ ПОРТЫ
35
36 PS C:\Users\dalam\OneDrive\Рабочий стол\projects\projects\Алгоритмизация> python -u "c:\Users\dalam\OneDrive\Рабочий
37 [4, 5, 5, 9, 10]
38 [2, 3, 5, 8, 10]
39 6 7 7 7 8 8 9 10 11 12 12 12 13 13 13 14 14 15 15 15 17 18 19 20
40 PS C:\Users\dalam\OneDrive\Рабочий стол\projects\projects\Алгоритмизация>
```

Рисунок 3. Код решения задания 6 и результат выполнения

Вывод: в результате выполнения лабораторной работы был изучен алгоритм heap sort и проведено исследование зависимости времени поиска от количества элементов в массиве, показавшее что зависимость время поиска линейно увеличивается с добавлением элементов в массив.