

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:  
Епифанов Алексей Александрович  
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 Роман Александрович

---

(подпись)

Отчет защищен с оценкой \_\_\_\_\_ Дата защиты \_\_\_\_\_

Ставрополь, 2023 г.

## Порядок выполнения работы:

1. Написал программу сравнения функций сортировки кучей: стандартной, улучшенной и использующей мин-кучу, а не макс.

```
pyqt5 | algorithm0 | program > main.py | MySort | heapify
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3
4 import random as rnd
5 import matplotlib.pyplot as plt
6 import numpy as np
7 import timeit
8 from scipy.optimize import curve_fit
9 import heapq
10
11
12
13 class MySort:
14     @staticmethod
15     def heapify(work_list, n, i):
16         largest = i # Инициализировать наибольший элемент как корень
17         left = 2 * i + 1 # левый = 2i + 1
18         right = 2 * i + 2 # правый = 2i + 2
19
20         # Проверка, существует ли левый дочерний элемент корня,
21         # и является ли он больше, чем корень
22         if left < n and work_list[left] > work_list[largest]:
23             largest = left
24
25         # То же самое для правого дочернего элемента
26         if right < n and work_list[right] > work_list[largest]:
27             largest = right
28
29         # Изменить корень, если нужно
30         if largest != i:
31             work_list[i], work_list[largest] = work_list[largest], work_list[i]
32             # Проверка heapify в корне
33             MySort.heapify(work_list, n, largest)
34
35     @staticmethod
36     def heap_sort(work_list):
37         n = len(work_list)
38         # Построение максимальной кучи
39         for i in range(n // 2 - 1, -1, -1):
40             MySort.heapify(work_list, n, i)
41
42         # Отсортированное извлечение элементов из кучи
43         for i in range(n - 1, 0, -1):
44             work_list[i], work_list[0] = work_list[0], work_list[i] # Swap
45             MySort.heapify(work_list, i, 0)
46
47 class HeapSort:
48     @staticmethod
49     def heap_sort(iterable):
50         # Преобразование списка в кучу
51         heapq.heapify(iterable)
52
53         # Извлечение минимальных элементов из кучи для сортировки
54         return [heapq.heappop(iterable) for _ in range(len(iterable))]
55
56 class HeapSortSpeed:
57     @staticmethod
58     def find_coeffs_bin(x, time):
59         params, _ = curve_fit(n_log_p, np.array(x),
60                               np.array(time))
61         a, b = params
62         return a, b
63
64     def n_log_n(x, a, b):
65         return a * x + np.log(x) + b
66
67     def create_graph(b, c, nograph):
68         plt.scatter(b, c, s=50)
69         ax, bx = find_coeffs_bin(b, c)
70         y_line = n_log_n(np.array(b), ax, bx)
71         plt.plot(b, y_line, color='red')
72         plt.title(nograph + " cnyash")
73         plt.xlabel("Размер массива")
74         plt.ylabel("Время работы функции")
75
76     def create_list(size, max_value, option):
77         match option:
78             case 'ordered':
79                 return list(range(1, size+1))
80             case 'random':
81                 return [rnd.randint(1, max_value) for _ in range(size)]
82
83 class HeapSortSpeed:
84     @staticmethod
85     def heapify(heap, n, pos):
86         """Heapify variant of siftup"""
87         endpos = n
88         newitem = heap[pos]
89         childpos = 2*pos + 1
90
91         while childpos < endpos:
92             rightpos = childpos + 1
93
94             if rightpos < endpos and not heap[rightpos] < heap[childpos]:
95                 childpos = rightpos
96
97             if heap[pos] < heap[childpos]:
98                 heap[pos] = heap[childpos]
99                 heap[childpos] = newitem
100                 pos = childpos
101                 childpos = 2*pos + 1
102             else:
103                 break
104         heap[pos] = newitem
105
106     @staticmethod
107     def heap_sort(iterable):
108         n = len(iterable)
109         heapq._heapify_max(iterable)
110
111         # Преобразование списка в кучу
112         heapq._heapify_max(iterable)
113
114         # Извлечение максимальных элементов из кучи для сортировки
115         for i in range(n-1, 0, -1):
116             iterable[i], iterable[0] = iterable[0], iterable[i] # Swap
117             HeapSortSpeed.heapify(iterable, i, 0)
118
119     def find_coeffs_bin(x, time):
120         params, _ = curve_fit(n_log_p, np.array(x),
121                               np.array(time))
122         a, b = params
123         return a, b
124
125     def n_log_n(x, a, b):
126         return a * x + np.log(x) + b
127
128     def create_graph(b, c, nograph):
129         plt.scatter(b, c, s=50)
130         ax, bx = find_coeffs_bin(b, c)
131         y_line = n_log_n(np.array(b), ax, bx)
132         plt.plot(b, y_line, color='red')
133         plt.title(nograph + " cnyash")
134         plt.xlabel("Размер массива")
135         plt.ylabel("Время работы функции")
136
137     def create_list(size, max_value, option):
138         match option:
139             case 'ordered':
140                 return list(range(1, size+1))
141             case 'random':
142                 return [rnd.randint(1, max_value) for _ in range(size)]
143
144 def create_list(size, max_value, option):
145     match option:
146         case 'ordered':
147             return list(range(1, size+1))
148         case 'random':
149             return [rnd.randint(1, max_value) for _ in range(size)]
150         case 'reverse':
151             return list(range(size, 0, -1))
152         case _:
153             return []
154
155 def func_time(class_func, case, case_name, size):
156     time = []
157     randmax = 1000000
158     x = [i for i in range(1000, 10001, 100)]
159     repeat = 20
160     if class_func == MySort:
161         x = x[150:]
162     for i in x:
163         timer = 0
164         for _ in range(repeat):
165             list_temp = create_list(i, randmax, case[i])
166             timer += (timeit.timeit(lambda: class_func.heap_sort(list_temp),
167                                   number=1)))
168         time.append(timer/repeat)
169     plt.figure(case[i] + case_name, size)
170     plt.subplots_adjust(left=0.25)
171     # Создать график
172     create_graph(x, time, case[i])
173
174 if __name__ == '__main__':
175     # Построить график
176     dpi = 100
177     width_inches = (1000 / dpi) / 4
178     height_inches = (1000 / dpi) / 2
179     size = (width_inches, height_inches)
180     item_func_name = ("MySort", "ordered",
181                      "HeapSort", "random",
182                      "HeapSortSpeed", "reverse")
183     for case_func in item_func_name.items():
184         func_time(MySort, case_func,
185                  "MySort", size)
186         func_time(HeapSort, case_func,
187                  "HeapSort", size)
188         func_time(HeapSortSpeed, case_func,
189                  "HeapSortSpeed", size)
190
191 # Показать график
192 plt.show()
```

Рисунок 1. Код программы

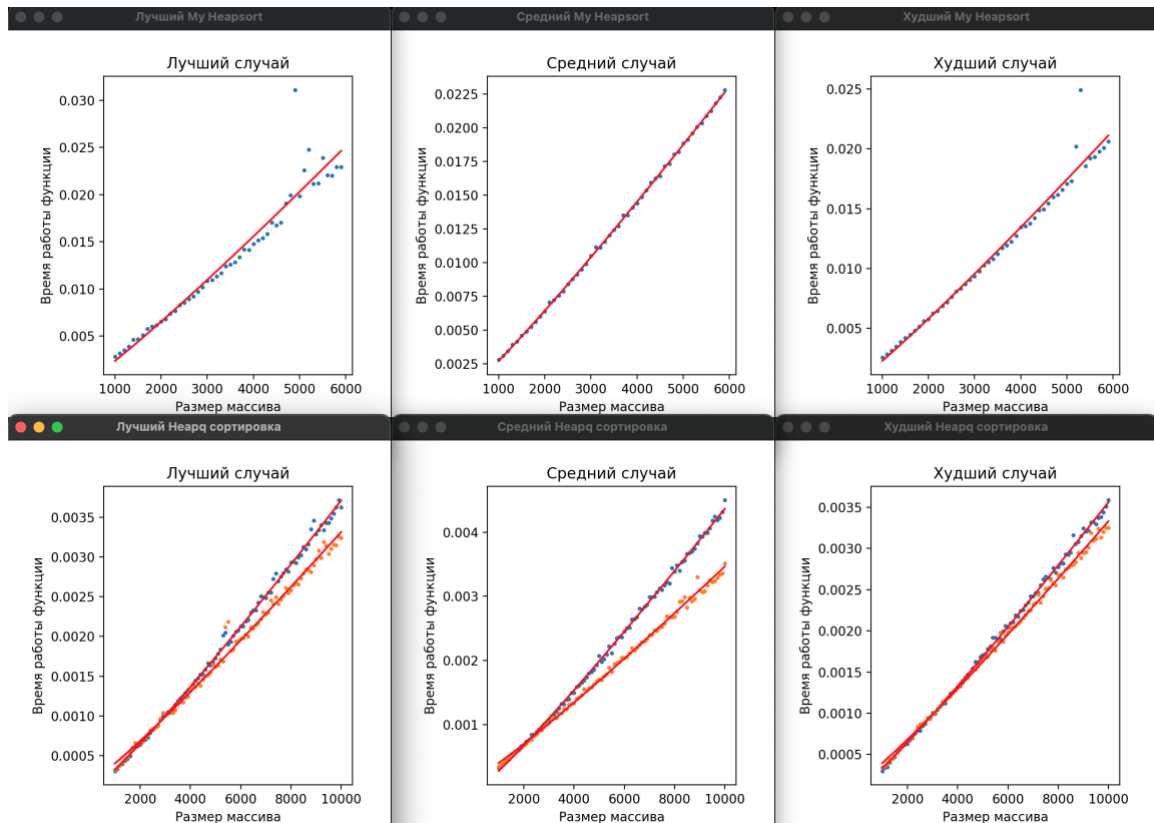


Рисунок 2. Результат работы программы

## 2. Сравнение с другими сортировками:

Алгоритм	Лучший случай	В среднем	Худший случай
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

## 3. Применение в реальной жизни:

Heap Sort может быть предпочтительным выбором, когда требуется эффективная сортировка больших массивов данных, так как его пространственная сложность  $= O(1)$ , а значит при любых размерах входных данных алгоритм будет требовать минимального выделения пространства для него, размер которого никак не зависит от размера входного массива. Также алгоритм применим когда важна стабильность процесса сортировки в плане производительности, так как он гарантирует время сортировки  $O(n \log n)$  независимо от начального порядка элементов.

## 4. Оценка времени и пространственной сложности:

Heapsort имеет временную сложность  $O(n \log n)$

Построение максимальной кучи:

```
for i in range(n // 2 - 1, -1, -1):
```

```
    heapify(work_list, n, i)
```

heapify() Выполняется за  $O(\log n)$  так как внутри содержит операции, выполняющиеся за  $O(1)$ , и может вызвать себя максимум  $\log n$  раз, потому что высота дерева  $\log n$ .

Цикл выполняется примерно  $n/2$  раз, но в сумме построение максимальной кучи выполняется за  $O(n)$ .

Отдельное извлечение элементов из кучи:

```
for i in range(n-1, 0, -1):
```

```
    work_list[i], work_list[0] = work_list[0], work_list[i] # Сwap
```

```
    heapify(work_list, i, 0)
```

heapify работает  $n$  раз, каждый раз выполняется за  $O(\log n)$ , а значит в сумме получается  $O(n \log n)$

Конечное время получается:  $O(n + n \log n)$  что равно  $O(n \log n)$ .

Пространственная сложность данного алгоритма равна  $O(1)$ , так как функция не требует дополнительной памяти и работает с данным списком (за исключением локальных переменных, которые не влияют на сложность, так как не меняются в зависимости от размера списка).

5. Написал программу по заданию: Даны массивы  $A[1 \dots n]$  и  $B[1 \dots n]$ . Мы хотим вывести все  $n^2$  сумм вида  $A[i] + B[j]$  в возрастающем порядке. Наивный способ — создать массив, содержащий все такие суммы, и отсортировать его. Соответствующий алгоритм имеет время работы  $O(n^2 \log n)$  и использует  $O(n^2)$  памяти. Приведите алгоритм с таким же временем работы, который использует линейную память.

```
pytgit > algoritm10 > program > task5.py > ...
Click here to ask Blackbox to help you code faster

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4
5  import heapq
6  from random import randint
7
8
9  def print_sorted_sums(list1, list2):
10     if not list1 or not list2:
11         return
12
13     list1.sort()
14     list2.sort()
15
16     heap = [(list1[0] + list2[0], 0, 0)]
17     pushed = {(0, 0)}
18     for _ in range(len(list1) * len(list2)):
19         sum, i, j = heapq.heappop(heap)
20         pushed.discard((i, j))
21         print(sum)
22
23         if i + 1 < len(list1) and (i + 1, j) not in pushed:
24             heapq.heappush(heap, (list1[i + 1] + list2[j], i + 1, j))
25             pushed.add((i + 1, j))
26
27         if j + 1 < len(list2) and (i, j + 1) not in pushed:
28             heapq.heappush(heap, (list1[i] + list2[j + 1], i, j + 1))
29             pushed.add((i, j + 1))
30
31
32 def generate_random_list(len_list):
33     random_list = [randint(0, 10)
34                    for _ in range(len_list)]
35     return random_list
36
37
38 def main():
39     # list1 = [1, 3, 5, 7]
40     # list2 = [6, 7, 8, 9]
41     list1 = generate_random_list(7)
42     list2 = generate_random_list(7)
43     print_sorted_sums(list1, list2)
44
45
46 if __name__ == '__main__':
47     main()
48
```

Рисунок 3. Код программы

```
aleksejepifanov@MacBook-Pro-Aleksej pytgit % /usr/local/bin/python3 /Users/aleksejepifanov/Desktop/пары_3_семе/pytgit/algorithm10/program/task5.py
23
28
48
52
53
54
54
57
59
68
79
80
82
83
83
84
85
87
93
97
99
105
108
109
109
111
112
113
113
114
115
115
119
120
124
127
139
142
143
146
150
160
164
172
175
176
176
179
180
aleksejepifanov@MacBook-Pro-Aleksej pytgit %
```

Рисунок 4. Результат работы программы

Данная программа работает за время  $O(n^2 \log n)$  и пространственную сложность  $O(n)$ , так как сначала выполняется сортировка двух списков на месте за время  $O(n \log n)$ , затем создается мин-куча из кортежей (сумма, индекс элемента из первого списка, индекс элемента из второго списка) и список для отслеживания уже помещенных в кучу сумм. В списке для отслеживания всегда столько же элементов, сколько и в куче.

Далее в кучу последовательно добавятся все  $n \cdot n$  пар элементов, каждое добавление будет выполняться за время  $O(\log n)$ , а также удалятся из кучи все эти элементы, каждое удаление также будет выполняться за время  $O(\log n)$ .

Суммарное время выполнения функции будет  $O(2n \log n + n^2 \log n)$ , что тоже самое, что и  $O(n^2 \log n)$ .

Пространственная сложность по итогу будет  $O(4(n-1))$ , что тоже самое что и  $O(n)$ .

Это из-за того, что в куче всегда хранится не более чем  $2(n-1)$  элементов, и столько же в списке, так как добавляются только те пары, сумма которых потенциально может быть больше извлеченной, то есть при извлечении из мин-кучи суммы пары  $i, j$  в кучу добавляются суммы пар  $i+1, j$  и  $i, j+1$ , если они есть и еще не помещены в кучу.

А так как  $i$  ограничена длиной первого списка, а  $j$  – второго, то размер кучи не превзойдет суммы длин обоих списков.

В ходе выполнения лабораторной работы было исследовано 3 разных алгоритма сортировки кучей. Первый, и самый худший написан сразу после прочтения механизма работы данной сортировки, второй алгоритм использует встроенную в python библиотеку `heapq`, которая реализует мин-кучу; он уже работает намного быстрее (примерно в 10 раз). Третий же алгоритм использует некоторую часть кода из библиотеки `heapq`, но немного переделанную; это привело к тому, что алгоритм при маленьких размерах входного списка (до 4000) уступает второму алгоритму, но далее в каждом случае он обгоняет его. Из полученных результатов можно сделать следующий вывод: для эффективной реализации алгоритма сортировки кучей лучше всего воспользоваться или функциями из библиотеки `heapq`, или воспользоваться кодом этой библиотеки в качестве основы для собственной реализации.