



СРСІ №2

WIN-1-22

Алгоритмы обработки массивов

Выполнил:

Бактыбеков Н.Б.

Проверил:

Картанова А.Д.

April 6, 2023

Contents

Индивидуальное задание	2
1 Задача 1	2
1.1 Условие задачи	2
1.2 Постановка задачи	2
1.3 Описание алгоритма	3
1.4 Контрольные примеры	3
1.5 Реализация решения задачи	4
1.6 Исходный код программы	4
1.7 Результаты работы программы	5
2 Задача 2	5
2.1 Условие задачи	5
2.2 Постановка задачи	5
2.3 Описание алгоритма	6
2.4 Контрольные примеры	6
2.5 Реализация решения задачи	6
2.6 Исходный код программы	6
2.7 Результаты работы программы	8
3 Задача 3	8
3.1 Условие задачи	8
3.2 Постановка задачи	8
3.3 Описание алгоритмов	9
3.4 Контрольные примеры	15
3.5 Реализация решения задачи	15
3.6 Построение таблицы	16
3.7 Скорость работы алгоритмов	22
3.8 Потребление памяти алгоритмов	32
4 Выводы по трем алгоритмам	42
5 Ссылки	42

Индивидуальное задание

Решить 3 задачи

Задача 1

Удалить из массива элемент, расположенный после максимального элемента. Если удаление элемента невозможно, выдать об этом сообщение.

Задача 2

Вставить заданное значение после каждого элемента массива, расположенного до первого нулевого элемента. Если вставка элементов невозможна, выдать об этом сообщение.

Задача 3

Реализовать алгоритм трех методов сортировки:

1. Сортировка Бэтчера
2. Сортировка на основе приоритетных очередей
3. Сортировка Пирамидой

Сравнить эти три сортировки по критериям:

1. Скорость выполнения
2. Потребление памяти

Решить каждую задачу в отдельной программе. Реализовать программу средствами языка программирования Python.

1 Задача 1

1.1 Условие задачи

Удалить из массива элемент, расположенный после максимального элемента. Если удаление элемента невозможно, выдать об этом сообщение.

1.2 Постановка задачи

Входные данные:

arr - массив из N элементов.

Выходные данные:

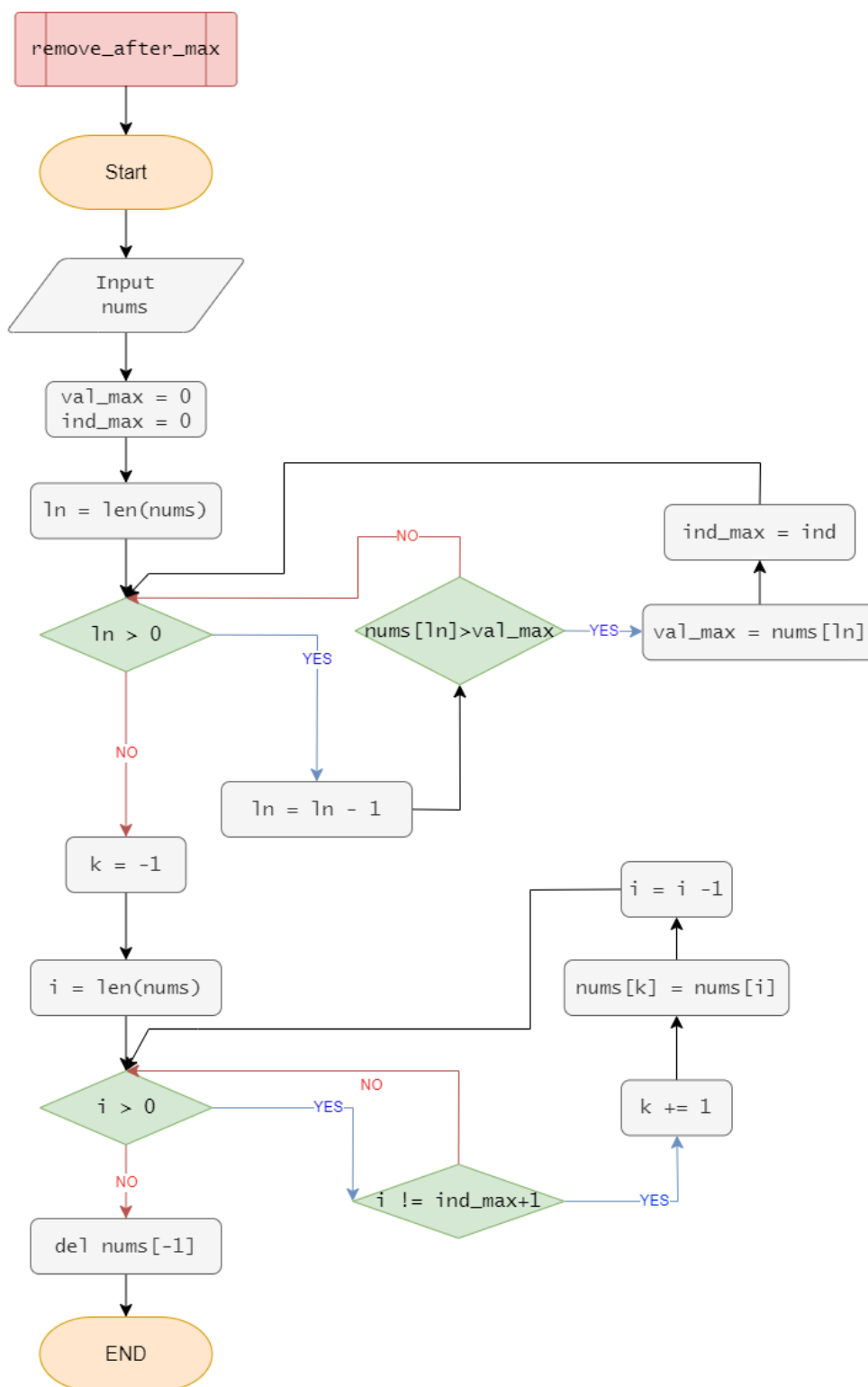
arr - результирующий массив из N-1 элементов.

Условия и ограничения:

Задание не может быть выполнено, если в массиве последний элемент является минимальным.

1.3 Описание алгоритма

Алгоритм задачи 1 описан в следующей блок-схеме:



Picture. 1: Блок схема Задание 1

1.4 Контрольные примеры

1. $a = [1, 5, 6, 2, 3, 4, 5, 9, 6, 7, 8]$

new $a = [1, 5, 6, 2, 3, 4, 5, 9, 7, 8]$

```
2. b=[8, 3, 6, 2, 3, 4, 5, 9, 3, 7, 2]
   new b=[8, 3, 6, 2, 3, 4, 5, 9, 7, 2]
3. Максимальный последний
   c=[3, 8, 2, 7, 3, 5, 4, 1, 2, 0, 9]
   new c=[3, 8, 2, 7, 3, 5, 4, 1, 2, 0, 9]
```

1.5 Реализация решения задачи

Реализация задачи 1 оформлена в виде функции в программе first-task.py

1.6 Исходный код программы

```
1  """Удалить
2  из массива элемент, расположенный
3  после максимального элемента. Если
4  удаление элемента невозможно, выдать об этом сообщение.
5  """
6
7
8  def remove_after_max(nums: list[int]):
9      # finding max value
10
11     val_max = 0
12     ind_max = 0
13     for ind, num in enumerate(nums):
14         if num > val_max:
15             val_max = num
16             ind_max = ind
17
18     if ind_max == len(nums) - 1:
19         print("max is last element, impossible to remove")
20         return -1
21
22     k = -1
23
24     for i in range(0, len(nums)):
25         if i != ind_max + 1: # elem after max
26             k += 1
27             nums[k] = nums[i]
28     del nums[-1]
29     return len(nums)
30
```

```

31
32 a = [1, 5, 6, 2, 3, 4, 5, 9, 6, 7, 8]
33 b = [8, 3, 6, 2, 3, 4, 5, 9, 3, 7, 2]
34 c = [3, 8, 2, 7, 3, 5, 4, 1, 2, 0, 9]
35 print(f"{a}")
36 remove_after_max(a)
37 print(f"new a={a}")
38 print(f"{b}")
39 remove_after_max(b)
40 print(f"new b={b}")
41 print(f"{c}")
42 remove_after_max(c)
43 print(f"new c={c}")

```

1.7 Результаты работы программы

```

> py .\two_tasks\first.py
a=[1, 5, 6, 2, 3, 4, 5, 9, 6, 7, 8]
new a=[1, 5, 6, 2, 3, 4, 5, 9, 7, 8]
b=[8, 3, 6, 2, 3, 4, 5, 9, 3, 7, 2]
new b=[8, 3, 6, 2, 3, 4, 5, 9, 7, 2]
c=[3, 8, 2, 7, 3, 5, 4, 1, 2, 0, 9]
max is last element, impossible to remove
new c=[3, 8, 2, 7, 3, 5, 4, 1, 2, 0, 9]

```

Picture. 2: Блок схема Задание 1

2 Задача 2

2.1 Условие задачи

Вставить заданное значение после каждого элемента массива, расположенного до первого нулевого элемента. Если вставка элементов невозможна, выдать об этом сообщение.

2.2 Постановка задачи

Входные данные:

arr - массив из N элементов. val - число которое нужно вставить

Выходные данные:

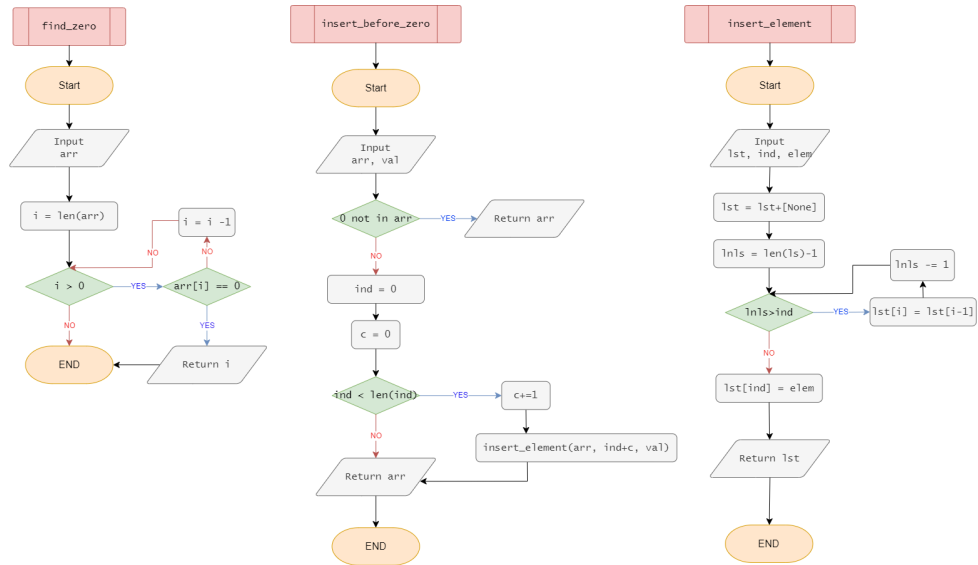
arr - результирующий массив из N+индекс-нуля элементов.

Условия и ограничения:

Задание не может быть выполнено, если в массиве первый элемент является нулевым.

2.3 Описание алгоритма

Алгоритм задачи 2 описан в следующей блок-схеме:



Picture. 3: Блок схема Задание 2

2.4 Контрольные примеры

1. $a = [1, 5, 6, 0, 3, 4, 5, 9, 6, 7, 8]$
 $\text{new } a = [1, 10, 5, 10, 6, 10, 0, 3, 4, 5, 9, 6, 7, 8]$
2. $b = [8, 3, 6, 2, 3, 4, 5, 0, 3, 7, 2]$
 $\text{new } b = [8, 10, 3, 10, 6, 10, 2, 10, 3, 10, 4, 10, 5, 10, 0, 3, 7, 2]$
3. Первый элемент = 0
 $c = [0, 8, 2, 7, 3, 5, 4, 1, 2, 0, 9]$
 $\text{new } c = [0, 8, 2, 7, 3, 5, 4, 1, 2, 0, 9]$

2.5 Реализация решения задачи

Реализация задачи 2 оформлена в виде функции в программе second-task.py

2.6 Исходный код программы

```
1 """Вставить
2 заданное значение после каждого элемента массива, расположенного
3 до первого нулевого элемента. Если
4 вставка элементов невозможна, выдать об этом сообщение.
5 """
```

```

6
7
8 def find_zero(arr):
9     for i, e in enumerate(arr):
10         if e == 0:
11             return i
12
13
14 def insert_before_zero(val, arr: list):
15     if 0 not in arr or arr[0] == 0:
16         print("Error: no zero element found")
17         return arr
18
19     c = 0
20
21     for ind, elem in enumerate(arr[: find_zero(arr)]):
22         c += 1
23         insert_element(arr, ind + c, val)
24
25     return arr
26
27
28 def insert_element(lst, index, element):
29     lst.append(None)
30     for i in range(len(lst) - 1, index, -1):
31         lst[i] = lst[i - 1]
32     lst[index] = element
33     return lst
34
35
36 val = 10
37 a = [1, 5, 6, 0, 3, 4, 5, 9, 6, 7, 8]
38
39 print(f"{a}")
40 a = insert_before_zero(val, a)
41 print(f"new a={a}")
42
43 b = [8, 3, 6, 2, 3, 4, 5, 0, 3, 7, 2]
44
45 print(f"{b}")
46 b = insert_before_zero(val, b)
47 print(f"new b={b}")

```



```

48
49 c = [0, 8, 2, 7, 3, 5, 4, 1, 2, 0, 9]
50
51 print(f"{c}")
52 c = insert_before_zero(val, c)
53 print(f"new c={c}")

```

2.7 Результаты работы программы

```

> py .\two_tasks\second.py
a=[1, 5, 6, 0, 3, 4, 5, 9, 6, 7, 8]
new a=[1, 10, 5, 10, 6, 10, 0, 3, 4, 5, 9, 6, 7, 8]
b=[8, 3, 6, 2, 3, 4, 5, 0, 3, 7, 2]
new b=[8, 10, 3, 10, 6, 10, 2, 10, 3, 10, 4, 10, 5, 10, 0, 3, 7, 2]
c=[0, 8, 2, 7, 3, 5, 4, 1, 2, 0, 9]
Error: no zero element found
new c=[0, 8, 2, 7, 3, 5, 4, 1, 2, 0, 9]

< orenv on Sunday at 5:39 PM  P main  24 ~5 -11  0.089s  CPU: 48%  RAM: 8/16GB
> { home -> Repos -> Algorhythms_and_DataStructures -> second_colloq -> Lab2 }

```

Picture. 4: Запуск программы Задание 2

3 Задача 3

3.1 Условие задачи

Реализовать алгоритм трех методов сортировки:

1. Сортировка Бэтчера
2. Сортировка на основе приоритетных очередей
3. Сортировка Пирамидой

Сравнить эти три сортировки по критериям:

1. Скорость выполнения
2. Потребление памяти

Решить каждую задачу в отдельной программе. Реализовать программу средствами языка программирования Python.

3.2 Постановка задачи

Входные данные:

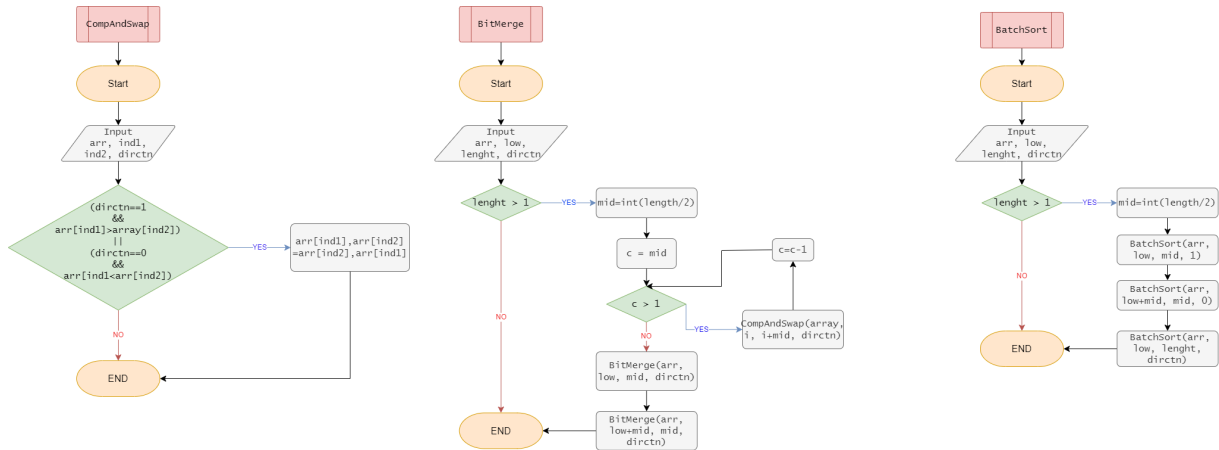
arr – список типа list, заполненный случайным образом

Выходные данные: V – Скорость выполнения M – Максимум расходуемой памяти arr –

отсортированный список

3.3 Описание алгоритмов

Блок-схема подпрограммы сортировки Бэтчера:



Picture. 5: Сортировка Бэтчера

Космлексность данного алгоритма составляет $O(n * \log n)$

Код программы с сортировкой Бэтчера

```
1 """
2 Python program for Bitonic Sort.
3 Note that this program works only when size of input is a power of 2.
4 """
5 from __future__ import annotations
6
7 from .wrappers_memory import profile
8 from .wrappers_speed import speedometer
9
10
11 def comp_and_swap(array: list[int], index1: int, index2: int, direction:
12     int) -> None:
13     """Compare the value at given index1 and index2 of the array and swap
14     them as per
15     the given direction.
16     The parameter direction indicates the sorting direction, ASCENDING(1)
17     or
18     DESCENDING(0); if (a[i] > a[j]) agrees with the direction, then a[i]
19     and a[j] are
20     interchanged.
21     """
22     if (direction == 1 and array[index1] > array[index2]) or (
```

```

19         direction == 0 and array[index1] < array[index2]
20     ):
21         array[index1], array[index2] = array[index2], array[index1]
22
23
24 def bitonic_merge(array: list[int], low: int, length: int, direction: int)
    -> None:
25     """
26     It recursively sorts a bitonic sequence in ascending order, if
27     direction = 1, and in
28     descending if direction = 0.
29     The sequence to be sorted starts at index position low, the parameter
30     length is the
31     number of elements to be sorted.
32     """
33     if length > 1:
34         middle = int(length / 2)
35         for i in range(low, low + middle):
36             print(array)
37             comp_and_swap(array, i, i + middle, direction)
38             bitonic_merge(array, low, middle, direction)
39             bitonic_merge(array, low + middle, middle, direction)
40
41 @speedometer
42 def bitonic_sort(array: list[int], low: int, length: int, direction: int)
    -> None:
43     """
44     This function first produces a bitonic sequence by recursively sorting
45     its two
46     halves in opposite sorting orders, and then calls bitonic_merge to make
47     them in the
48     same order.
49     """
50     if length > 1:
51         middle = int(length / 2)
52         bitonic_sort(array, low, middle, 1)
53         bitonic_sort(array, low + middle, middle, 0)
54         bitonic_merge(array, low, length, direction)
55
56 @profile

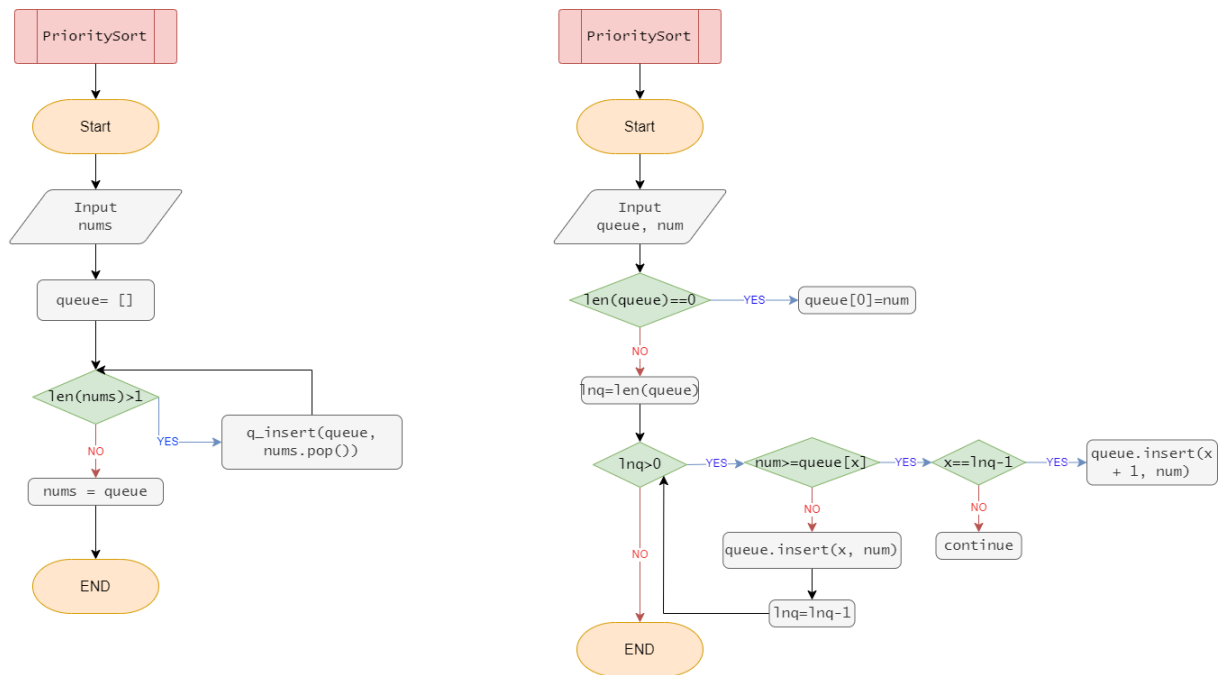
```

```

55 def bitonic_sortMem(array: list[int], low: int, length: int, direction: int
    ) -> None:
56     if length > 1:
57         middle = int(length / 2)
58         bitonic_sort(array, low, middle, 1)
59         bitonic_sort(array, low + middle, middle, 0)
60         bitonic_merge(array, low, length, direction)

```

Блок-схема подпрограммы сортировки Приоритетных очередей:



Picture. 6: Сортировка на основе Приоритетных очередей

Комплексность данного алгоритма составляет $O(\log^2 n)$

Код программы с сортировкой на основе Приоритетных очередей

```

1 from .wrappers_memory import profile
2 from .wrappers_speed import speedometer
3
4
5 @speedometer
6 def priority_sort(nums: list[int]):
7     queue = []
8     for num in nums:
9         q_insert(queue, num)
10        print(queue)
11    nums = queue
12
13
14 @profile

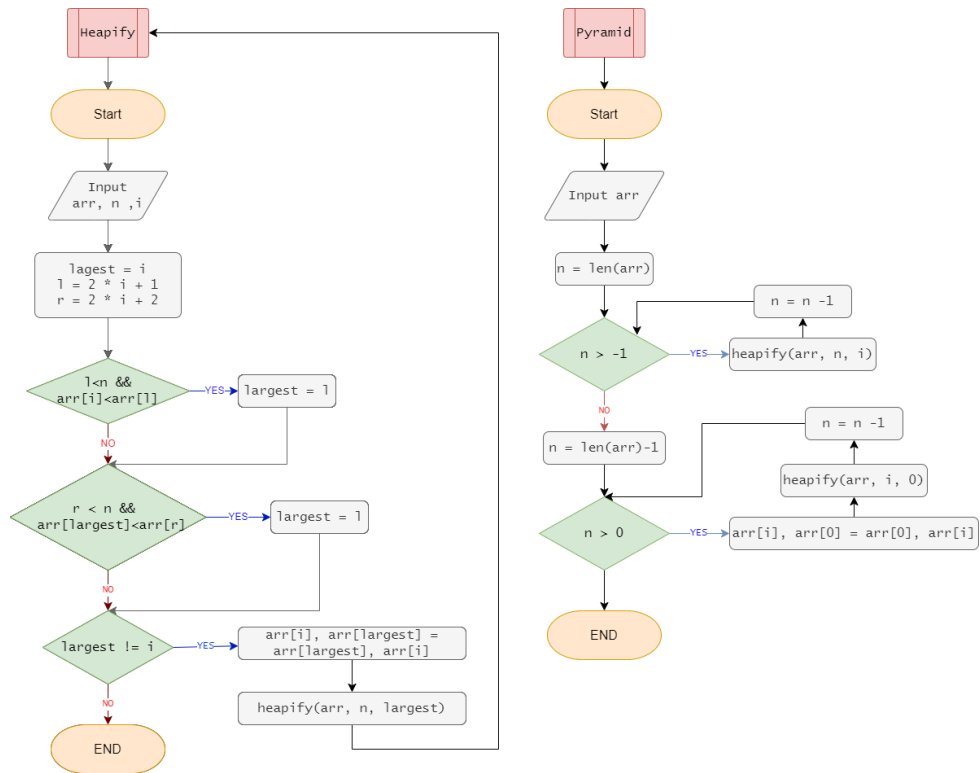
```

```

15 def priority_sortMem(nums: list[int]):
16     queue = []
17     for num in nums:
18         q_insert(queue, num)
19     nums = queue
20
21
22 def q_insert(queue: list, num):
23     # if queue is empty
24     if len(queue) == 0:
25         # add the new num
26         queue.append(num)
27     else:
28         # traverse the queue to find the right place for new num
29         for x in range(0, len(queue)):
30             # if the of new num is greater
31             if num >= queue[x]:
32                 # if we have traversed the complete queue
33                 if x == (len(queue) - 1):
34                     # add new num at the end
35                     queue.insert(x + 1, num)
36             else:
37                 continue
38         else:
39             queue.insert(x, num)
40     return True

```

Блок-схема подпрограммы сортировки Пирамидой:



Picture. 7: Запуск программы Задание 2

Комплексность данного алгоритма составляет $O(n^2)$

Код программы с сортировкой Пирамидой

```

1 from .wrappers_memory import profile
2 from .wrappers_speed import speedometer
3
4 # Implementation of heapsort in Python
5
6 # Procedure to convert to a binary heap a subtree with root node i, which
  is an index in arr[]. n - heap size
7
8
9 def heapify(arr, n, i):
10     largest = i # Initialize largest as root
11     l = 2 * i + 1 # left = 2*i + 1
12     r = 2 * i + 2 # right = 2*i + 2
13
14     # check if exists left child elem > root
15
16     if l < n and arr[i] < arr[l]:
17         # print(arr)
18         largest = l
19
20     # check if exists right child elem > root

```

```

21
22     if r < n and arr[largest] < arr[r]:
23         # print(arr)
24         largest = r
25
26     # replace root if needed
27     if largest != i:
28         print(arr)
29         arr[i], arr[largest] = arr[largest], arr[i] # swap
30
31         # heapify to root.
32         heapify(arr, n, largest)
33
34
35 # main func
36 @speedometer
37 def heapSort(arr):
38     n = len(arr)
39
40     # building max-heap.
41     for i in range(n, -1, -1):
42         heapify(arr, n, i)
43
44     # one after one taking out elements
45     for i in range(n - 1, 0, -1):
46         arr[i], arr[0] = arr[0], arr[i] # swap
47         heapify(arr, i, 0)
48
49
50 @profile
51 def heapSortMem(arr):
52     n = len(arr)
53
54     # building max-heap.
55     for i in range(n, -1, -1):
56         heapify(arr, n, i)
57
58     # step by step taking out elements
59     for i in range(n - 1, 0, -1):
60         arr[i], arr[0] = arr[0], arr[i] # swap
61         heapify(arr, i, 0)

```

3.4 Контрольные примеры

Входные данные:

ls: list[list[int]] список содержащий списки длиной 8 содержащие числа

Выходные данные

пошаговый показ процесса сортировки, отсортированный массив, скорость выполнения, потребление памяти

Сортировка массива длиной 8 тремя алгоритмами по шагам

```
> py .\main_steps.py
[array('i', [0, 4, 5, 7, 1, 3, 6, 2]), array('i', [0, 4, 5, 7, 1, 3, 6, 2]), array('i', [0, 4, 5, 7, 1, 3, 6, 2])]
*****bitonic*****
array('i', [0, 4, 5, 7, 1, 3, 6, 2])
array('i', [0, 4, 5, 7, 1, 3, 6, 2])
array('i', [0, 4, 7, 5, 1, 3, 6, 2])
array('i', [0, 4, 7, 5, 1, 3, 6, 2])
array('i', [0, 4, 7, 5, 1, 3, 6, 2])
array('i', [0, 4, 7, 5, 1, 3, 6, 2])
array('i', [0, 4, 5, 7, 1, 3, 6, 2])
array('i', [0, 4, 5, 7, 1, 3, 6, 2])
array('i', [0, 4, 5, 7, 1, 3, 6, 2])
array('i', [0, 4, 5, 7, 6, 3, 1, 2])
array('i', [0, 4, 5, 7, 6, 3, 1, 2])
array('i', [0, 4, 5, 7, 6, 3, 1, 2])
array('i', [0, 4, 5, 7, 6, 3, 2, 1])
array('i', [0, 4, 5, 7, 6, 3, 2, 1])
array('i', [0, 3, 5, 7, 6, 4, 2, 1])
array('i', [0, 3, 2, 7, 6, 4, 5, 1])
array('i', [0, 3, 2, 1, 6, 4, 5, 7])
array('i', [0, 3, 2, 1, 6, 4, 5, 7])
array('i', [0, 1, 2, 3, 6, 4, 5, 7])
array('i', [0, 1, 2, 3, 6, 4, 5, 7])
array('i', [0, 1, 2, 3, 6, 4, 5, 7])
array('i', [0, 1, 2, 3, 5, 4, 6, 7])
array('i', [0, 1, 2, 3, 5, 4, 6, 7])
array('i', [0, 1, 2, 3, 4, 5, 6, 7])
*****bitonic*****
*****priority*****
[0]
[0, 4]
[0, 4, 5]
[0, 4, 5, 7]
[0, 1, 4, 5, 7]
[0, 1, 3, 4, 5, 7]
[0, 1, 3, 4, 5, 6, 7]
[0, 1, 2, 3, 4, 5, 6, 7]
*****priority*****
*****pyramid*****
array('i', [0, 4, 5, 7, 1, 3, 6, 2])
array('i', [0, 4, 6, 7, 1, 3, 5, 2])
array('i', [0, 7, 6, 4, 1, 3, 5, 2])
array('i', [7, 0, 6, 4, 1, 3, 5, 2])
array('i', [7, 4, 6, 0, 1, 3, 5, 2])
array('i', [0, 4, 6, 2, 1, 3, 5, 7])
array('i', [6, 4, 0, 2, 1, 3, 5, 7])
array('i', [0, 4, 5, 2, 1, 3, 6, 7])
array('i', [5, 4, 0, 2, 1, 3, 6, 7])
array('i', [0, 4, 3, 2, 1, 5, 6, 7])
array('i', [4, 0, 3, 2, 1, 5, 6, 7])
array('i', [1, 2, 3, 0, 4, 5, 6, 7])
array('i', [0, 2, 1, 3, 4, 5, 6, 7])
*****pyramid*****
[array('i', [0, 1, 2, 3, 4, 5, 6, 7]), array('i', [0, 4, 5, 7, 1, 3, 6, 2]), array('i', [0, 1, 2, 3, 4, 5, 6, 7])]
< orenv on Wednesday at 11:28 PM  o P main  71 ~5
> { home - Repos - Algorithms_and_DataStructures - second_colloq - Lab2 } * 0.127s CPU: 25% RAM: 4/16GB
```

Picture. 8: Запуск программы Задание 2

3.5 Реализация решения задачи

Решение задачи 3 оформлено в виде трех программ: table.py, main.py, memory-usege.py. Программы используют три подпрограммы для выполнения сортировки: batcher-sort, priority-sort, pyramid-sort.

3.6 Построение таблицы

Программа 1 импортирует 3 подпрограммы сортировки и строит таблицу значений скорости и потребления памяти для трех алгоритмов сортировки

Исходный код Программы 1

```
1 # for randomizing
2 import random
3 # for fast arrays
4 from array import array
5
6 from prettytable import PrettyTable
7 # for table
8 from tabletexifier import Table
9
10 # importing sorts
11 from methods.batcher import bitonic_sort, bitonic_sortMem
12 from methods.priority_func import priority_sort, priority_sortMem
13 from methods.pyramid import heapSort, heapSortMem
14
15 # for beautifull print
16 # from pprint import pprint
17
18 # for graphs and plots
19 # import numpy as np
20 # from matplotlib import pyplot as plt
21
22
23 arr = list(range(128))
24 arr = array("i", arr) # 'i' - signed int
25
26 ls = []
27
28 for i in range(600):
29     random.shuffle(arr)
30     ls.append(arr)
31
32
33 def save_to_tex(table: Table, path: str):
34     with open(path, "w", encoding="utf-8") as file:
35         pass
36     txt = table.build_latex()
37     with open(path, "w", encoding="utf-8") as file:
```

```

38         file.writelines(txt.split("\n"))
39
40     # Read in the file
41     with open(path, "r", encoding="utf-8") as file:
42         filedata = file.read()
43     # Replace the target string
44     filedata = filedata.replace("begin{table}", "begin{table}[H]")
45     filedata = filedata.replace(
46         "{\\label{Tab:}}", ("{" + path.split("/")[2][:-10] + " sort")
47     )
48     # Write the file out again
49     with open(path, "w", encoding="utf-8") as file:
50         file.write(filedata)
51
52
53     ##### Batcher Sort
54     #####
55     exec_speeds_bitonic = [
56         [i + 1, bitonic_sort(ls[i], 0, 128, 1), bitonic_sortMem(ls[i + 100], 0,
57             128, 1)]
58         for i in range(40)
59     ] # works
60
61     bitonic_table = Table(
62         [
63             "Запуск программы",
64             "Скорость выполнения",
65             "Потребление памяти",
66         ],
67         table_style="A",
68     )
69     # bitonic_table.title = "Batcher sort"
70
71     for row in exec_speeds_bitonic:
72         bitonic_table.add_row(row)
73
74     save_to_tex(bitonic_table, "research_doc/tables/batcher_table.tex")
75
76     ##### Batcher Sort
77     #####
78
79     ##### Pyramid Sort

```

```

#####
77 exec_speeds_heap = [
78     [i + 1, heapSort(ls[i + 200]), heapSortMem(ls[i + 300])] for i in range
79     (40)
80 ] # works
81
82 heap_table = Table(
83     [
84         "Запуск программы",
85         "Скорость выполнения",
86         "Потребление памяти",
87     ],
88     table_style="A",
89 )
90 # heap_table.title = "Pyramid sort"
91
92 for row in exec_speeds_heap:
93     heap_table.add_row(row)
94
95 save_to_tex(heap_table, "research_doc/tables/pyramid_table.tex")
96
97 ##### Pyramid Sort
98 #####
99
100 ##### Priority Sort
101 #####
102
103 exec_speeds_priority = [
104     [i + 1, priority_sort(ls[i + 400]), priority_sortMem(ls[i + 500])]
105     for i in range(40)
106 ] # works
107
108 priority_table = Table(
109     [
110         "Запуск программы",
111         "Скорость выполнения",
112         "Потребление памяти",
113     ],
114     table_style="A",
115 )
116 # priority_table.title = "Pyramid sort"
117
118

```

```

115 for row in exec_speeds_priority:
116     priority_table.add_row(row)
117
118 save_to_tex(priority_table, "research_doc/tables/priority_table.tex")
119 ##### Priority Sort
    #####

```

Таким образом были построены таблицы:

Table 1: batcher sort

Запуск программы	Скорость выполнения	Потребление памяти
1	0.0008890000026440248	20086784
2	0.0008505000005243346	20086784
3	0.0008888000011211261	20086784
4	0.0008426999993389472	20086784
5	0.0008421999955317006	20086784
6	0.0008430000016232952	20086784
7	0.0008448000007774681	20086784
8	0.0008440000092377886	20086784
9	0.0008452999900327995	20086784
10	0.0008441999962087721	20086784
11	0.0008420999947702512	20086784
12	0.0008587999909650534	20086784
13	0.000836799998069182	20086784
14	0.0008390000002691522	20086784
15	0.0008393000025535002	20086784
16	0.0008424000116065145	20086784
17	0.0008411999879172072	20086784
18	0.0008405999979004264	20090880
19	0.0008417999924859032	20090880
20	0.0008437000069534406	20090880
21	0.0008435000054305419	20090880
22	0.0008436999924015254	20090880
23	0.0008428000001003966	20090880
24	0.0008422000100836158	20090880
25	0.0008414000039920211	20090880
26	0.0008418999932473525	20090880
27	0.0008421999955317006	20090880
28	0.0008429000008618459	20090880
29	0.0008436000061919913	20090880
30	0.0009122999908868223	20094976
31	0.0008470000029774383	20094976
32	0.0008422999962931499	20094976
33	0.0008432000031461939	20094976
34	0.0008437000069534406	20094976
35	0.0008432000031461939	20094976
36	0.0008431000023847446	20094976
37	0.0008440999954473227	20094976
38	0.0008423999970545992	20094976
39	0.0008410000009462237	20094976
40	0.0008455000061076134	20094976

Table 2: priority sort

Запуск программы	Скорость выполнения	Потребление памяти
1	0.0006151999987196177	20180992
2	0.0006086999928811565	20180992
3	0.0006117000011727214	20180992
4	0.0006113000126788393	20180992
5	0.0007803000044077635	20180992
6	0.0005686999938916415	20180992
7	0.0005701000045519322	20180992
8	0.0005651999963447452	20180992
9	0.0005688999954145402	20180992
10	0.0006706999993184581	20180992
11	0.000640199999907054	20180992
12	0.0005913000059081241	20180992
13	0.0006655000033788383	20180992
14	0.0005650000093737617	20180992
15	0.0005659000016748905	20180992
16	0.0005753000004915521	20180992
17	0.000701400000252761	20180992
18	0.0005936999950790778	20180992
19	0.0005670999962603673	20180992
20	0.0006739999953424558	20180992
21	0.0006854999955976382	20180992
22	0.0005665999924531206	20180992
23	0.0005696000007446855	20180992
24	0.0005664000054821372	20180992
25	0.000636800003121607	20180992
26	0.0005664999916916713	20180992
27	0.0006527000077767298	20180992
28	0.0005699000030290335	20180992
29	0.000698099989676848	20180992
30	0.000582000007852912	20180992
31	0.0005678999878000468	20180992
32	0.0005677000008290634	20180992
33	0.0005855000053998083	20180992
34	0.0005704999930458143	20180992
35	0.00056820000463631	20185088
36	0.0005673999985447153	20185088
37	0.000567600000067614	20185088
38	0.0005667999939760193	20185088
39	0.000566999995498918	20185088
40	0.0006833000079495832	20185088

Table 3: pyramid sort

Запуск программы	Скорость выполнения	Потребление памяти
1	0.0003741999971680343	20172800
2	0.0003708000003825873	20172800
3	0.00036299999919719994	20172800
4	0.0003657999914139509	20172800
5	0.00036419999378267676	20172800
6	0.00036530000215861946	20172800
7	0.00036869999894406646	20172800
8	0.00036580000596586615	20172800
9	0.0003635000030044466	20172800
10	0.00036920000275131315	20172800
11	0.0003634000022429973	20172800
12	0.00036639999598264694	20172800
13	0.00036639999598264694	20172800
14	0.0003656000044429675	20172800
15	0.00036439999530557543	20172800
16	0.0003657999914139509	20172800
17	0.00036360000376589596	20172800
18	0.0003699000080814585	20172800
19	0.00036419999378267676	20172800
20	0.0003657000052044168	20172800
21	0.00036809999437537044	20172800
22	0.0003642999945441261	20172800
23	0.0003643000090960413	20172800
24	0.00037259999953676015	20172800
25	0.00036720000207424164	20172800
26	0.00036669999826699495	20172800
27	0.0003704999980982393	20172800
28	0.000367600005120039	20172800
29	0.0003642999945441261	20172800
30	0.0003727000002982095	20172800
31	0.00036779999209102243	20172800
32	0.00036839999665971845	20172800
33	0.0003676999913295731	20176896
34	0.00036949999048374593	20180992
35	0.00036639999598264694	20180992
36	0.0003681999951368198	20180992
37	0.0003677000058814883	20180992
38	0.00037290000182110816	20180992
39	0.00036689999978989363	20180992
40	0.00036890000046696514	20180992

Для их построения каждая программа сортировки отсортировала массив длиной 128 заполненный случайными числами от 0 до 127, 40 раз

3.7 Скорость работы алгоритмов

Стабильность скорости работы

Программа 2 импортирует 3 подпрограммы сортировки и строит графики для трех алгоритмов сортировки.

По оси x Запуски программ

По оси y Скорость выполнения

Исходный код Программы 2

```
1 # for randomizing
2 import random
3 # for fast arrays
4 from array import array
5
6 # for graphs and research_doc/plots
7 import numpy as np
8 from matplotlib import pyplot as plt
9
10 # importing sorts
11 from methods.batcher import bitonic_sort
12 from methods.priority_func import priority_sort
13 from methods.pyramid import heapSort
14
15 arr = list(range(128))
16 arr = array("i", arr) # 'i' - signed int
17
18 ls = []
19
20 for i in range(300):
21     random.shuffle(arr)
22     ls.append(arr)
23
24
25 colors = ["batcher:blue", "priority:orange", "pyramid:green"]
26 ##### Batcher Sort
27 #####
28 exec_speeds_bitonic = [[i, bitonic_sort(ls[i], 0, 128, 1)] for i in range
29 (100)] # works
30
31 data = np.array(exec_speeds_bitonic)
32
33 x, y = data.T
34 plt.scatter(
```



```

35     x,
36     y,
37     c=colors[0].split(":")[1],
38     label=colors[0],
39     alpha=0.3,
40     edgecolors="none",
41 )
42 plt.ylabel("Скорость выполнения в секундах")
43 plt.xlabel("Запуски программы сортировки")
44 plt.legend()
45 # plt.show()
46 plt.savefig("research_doc/plots/batcher_speed.png", dpi=400) # savefig,
    don't show
47 ##### Batcher Sort
    #####
48
49 ##### Priority Sort
    #####
50 exec_speeds_priority = [
51     [i, priority_sort(ls[i + 100])] for i in range(100)
52 ] # for heapSort only
53
54 data = np.array(exec_speeds_priority)
55
56 x, y = data.T
57 plt.scatter(
58     x,
59     y,
60     c=colors[1].split(":")[1],
61     label=colors[1],
62     alpha=0.3,
63     edgecolors="none",
64 )
65 plt.ylabel("Скорость выполнения в секундах")
66 plt.xlabel("Запуски программы сортировки")
67 plt.legend()
68 # plt.show()
69 plt.savefig("research_doc/plots/priority_speed.png", dpi=400) # savefig,
    don't show
70 ##### Priority Sort
    #####
71

```

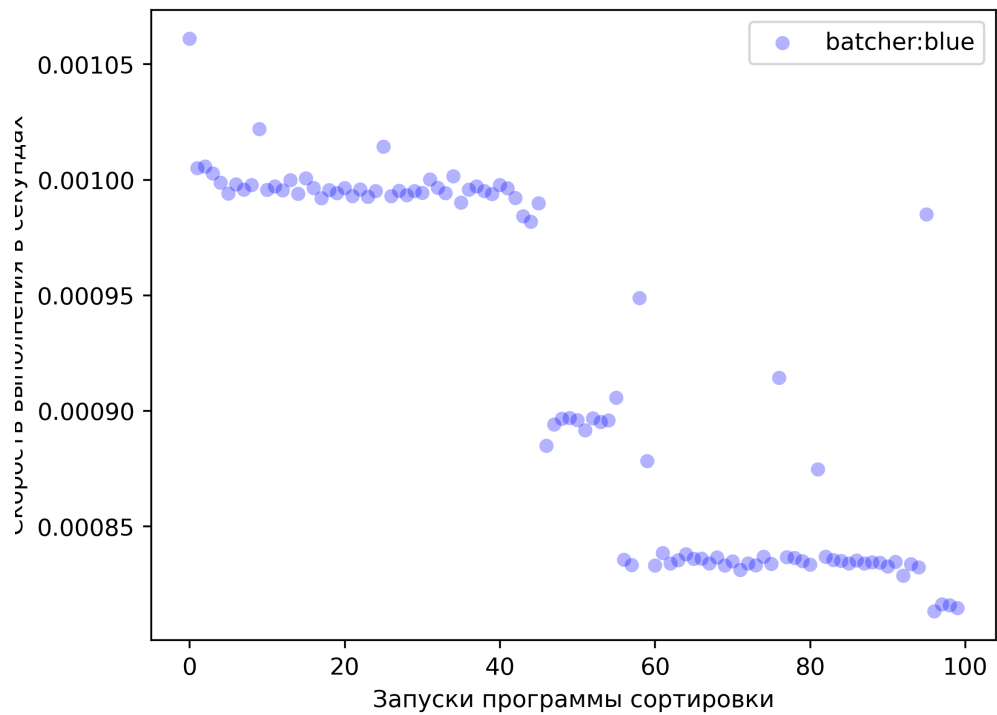
```

72 ##### Pyramid Sort
   #####
73 exec_speeds_heap = [[i, heapSort(ls[i + 200])] for i in range(100)] #
   works
74
75 data = np.array(exec_speeds_heap)
76
77 x, y = data.T
78 plt.scatter(
79     x,
80     y,
81     c=colors[2].split(":")[1],
82     label=colors[2],
83     alpha=0.3,
84     edgecolors="none",
85 )
86 plt.ylabel("Скорость выполнения в секундах")
87 plt.xlabel("Запуски программы сортировки")
88 plt.legend()
89 # plt.show()
90 plt.savefig("research_doc/plots/bitonic_speed.png", dpi=400) # savefig,
   don't show
91 ##### Pyramid Sort
   #####

```

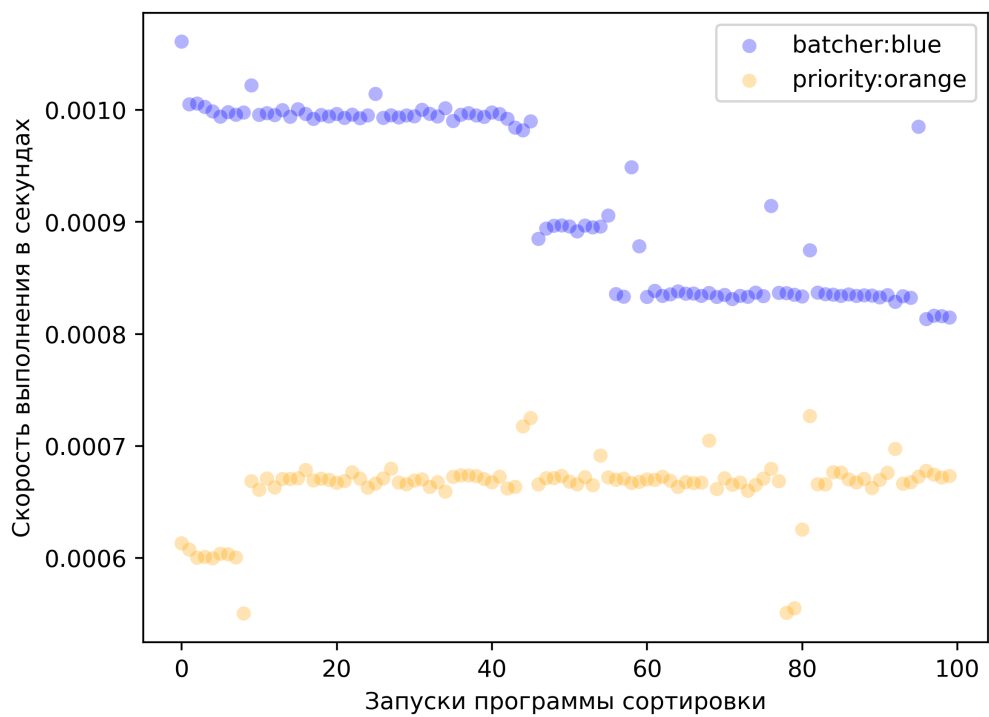
Таким образом были построены графики:

Скорость выполнения сортировки Бэтчера

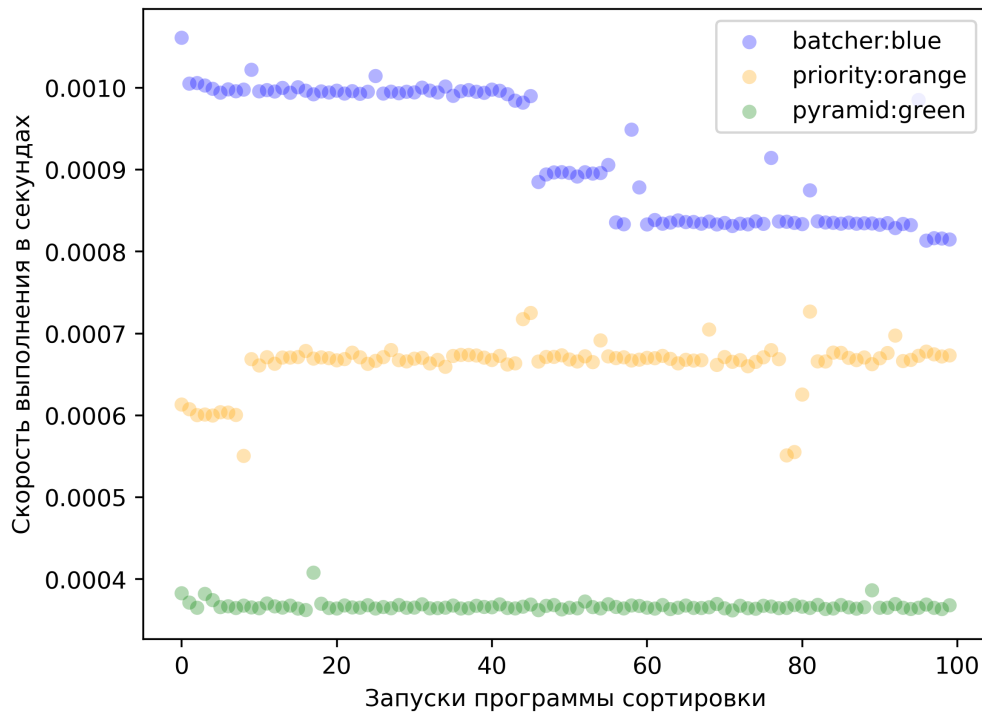


Picture. 9: График сортировки Бэтчера

Скорость выполнения сортировки на основе приоритетных очередей в сравнении с сортировкой Бэтчера



Picture. 10: График сортировки Приоритетных очередей



Picture. 11: График сортировки Пирамидой

Скорость работы по отношению к длине

Программа 2 импортирует 3 подпрограммы сортировки и строит графики для трех алгоритмов сортировки.

По оси x Запуски программ

По оси y Скорость выполнения

Исходный код Программы 2

```
1 # for randomizing
2 import random
3 # for fast arrays
4 from array import array
5
6 # for graphs and research_doc/plots
7 import numpy as np
8 from matplotlib import pyplot as plt
9
10 # importing sorts
11 from methods.batcher import bitonic_sort
12 from methods.priority_func import priority_sort
13 from methods.pyramid import heapSort
14
```

```

15 arr = list(range(4))
16 arr = array("i", arr) # 'i' - signed int
17
18 ls = []
19
20 for i in range(10):
21     random.shuffle(arr)
22     ls.append(arr)
23     ln_arr = len(arr)
24     arr.extend(list(range(ln_arr, ln_arr * 2)))
25
26
27 colors = ["batcher:blue", "priority:orange", "pyramid:green"]
28 ##### Batcher Sort
29 #####
29 exec_speeds_bitonic = [
30     [2 ** (i + 2), bitonic_sort(ls[i], 0, len(ls[i]), 1)] for i in range
31     (10)
32 ] # works
33
34 data = np.array(exec_speeds_bitonic)
35
36 x, y = data.T
37
38 plt.scatter(
39     x,
40     y,
41     c=colors[0].split(":")[1],
42     label=colors[0],
43     alpha=0.3,
44     edgecolors="none",
45 )
46 plt.ylabel("Скорость выполнения в секундах")
47 plt.xlabel("Длина сортируемого массива")
48 plt.legend()
49 # plt.show()
50 plt.savefig(
51     "research_doc/plots/batcher_speed_delta.png", dpi=400
52 ) # savefig, don't show
53 ##### Batcher Sort
54 #####

```

```

54
55 arr = list(range(4))
56 arr = array("i", arr) # 'i' - signed int
57
58 ls = []
59
60 for i in range(10):
61     random.shuffle(arr)
62     ls.append(arr)
63     ln_arr = len(arr)
64     arr.extend(list(range(ln_arr, ln_arr * 2)))
65
66 ##### Priority Sort
67 #####
68 exec_speeds_priority = [
69     [2 * (i + 2), priority_sort(ls[i])] for i in range(10)
70 ] # for heapSort only
71
72 data = np.array(exec_speeds_priority)
73
74 x, y = data.T
75 plt.scatter(
76     x,
77     y,
78     c=colors[1].split(":")[1],
79     label=colors[1],
80     alpha=0.3,
81     edgecolors="none",
82 )
83 plt.ylabel("Скорость выполнения в секундах")
84 plt.xlabel("Длина сортируемого массива")
85 plt.legend()
86 # plt.show()
87 plt.savefig(
88     "research_doc/plots/priority_speed_delta.png", dpi=400
89 ) # savefig, don't show
90
91 ##### Priority Sort
92 #####
93
94 arr = list(range(4))
95 arr = array("i", arr) # 'i' - signed int

```

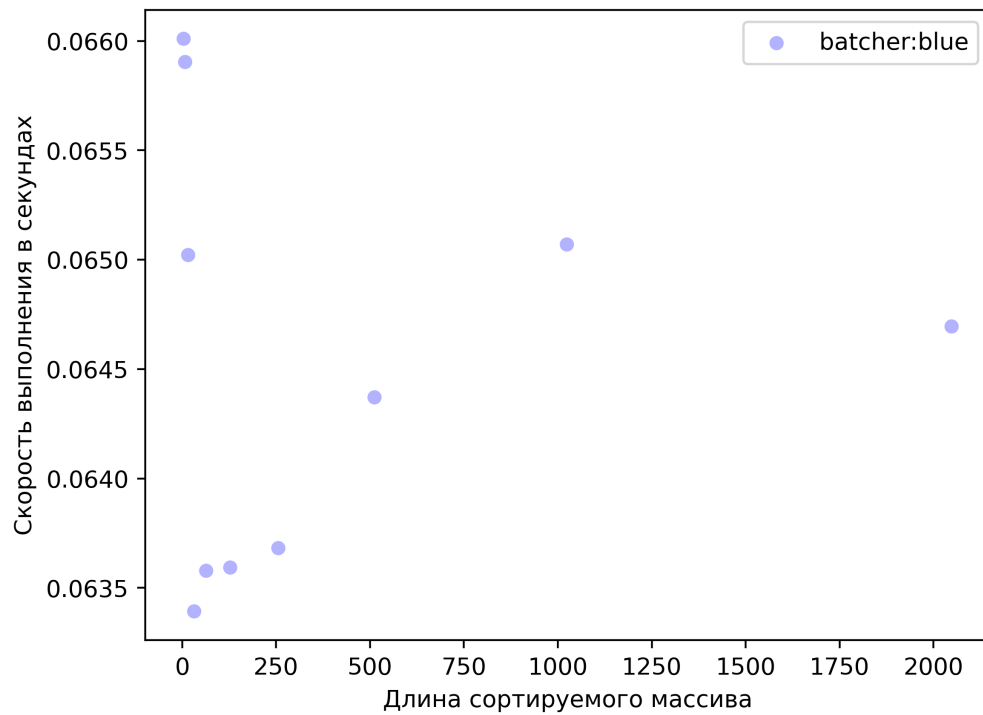
```

94 ls = []
95
96 for i in range(10):
97     random.shuffle(arr)
98     ls.append(arr)
99     ln_arr = len(arr)
100     arr.extend(list(range(ln_arr, ln_arr * 2)))
101
102 ##### Pyramid Sort
103 #####
104 exec_speeds_heap = [[2 ** (i + 2), heapSort(ls[i])] for i in range(10)] #
105 works
106
107 data = np.array(exec_speeds_heap)
108
109 x, y = data.T
110 plt.scatter(
111     x,
112     y,
113     c=colors[2].split(":")[1],
114     label=colors[2],
115     alpha=0.3,
116     edgecolors="none",
117 )
118 plt.ylabel("Скорость выполнения в секундах")
119 plt.xlabel("Длина сортируемого массива")
120 plt.legend()
121 # plt.show()
122 plt.savefig(
123     "research_doc/plots/bitonic_speed_delta.png", dpi=400
124 ) # savefig, don't show
125 ##### Pyramid Sort
126 #####

```

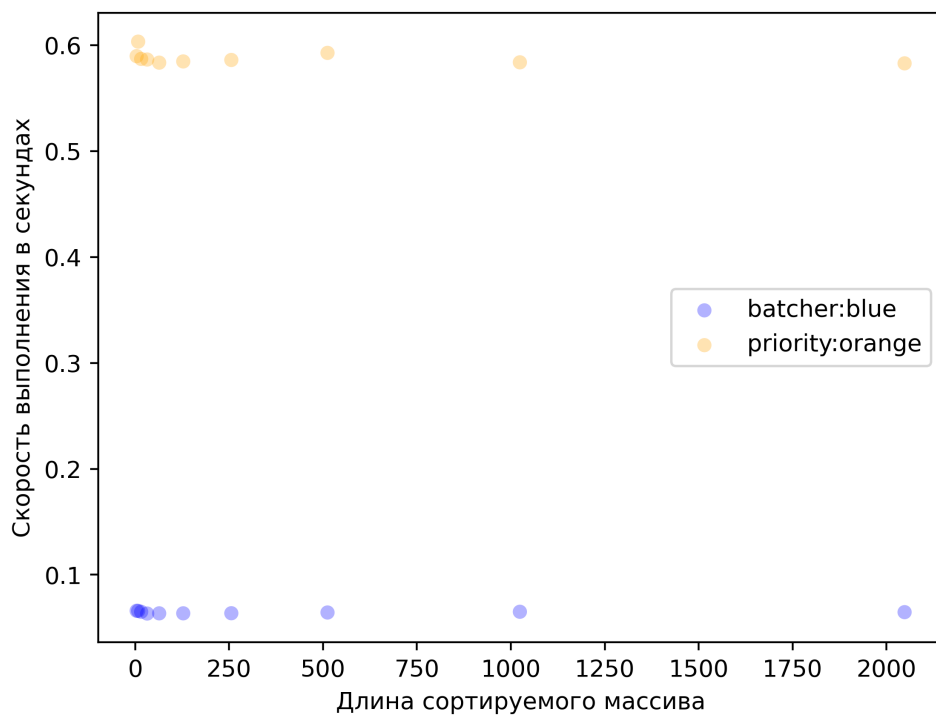
Таким образом были построены графики:

Скорость выполнения сортировки Бэтчера

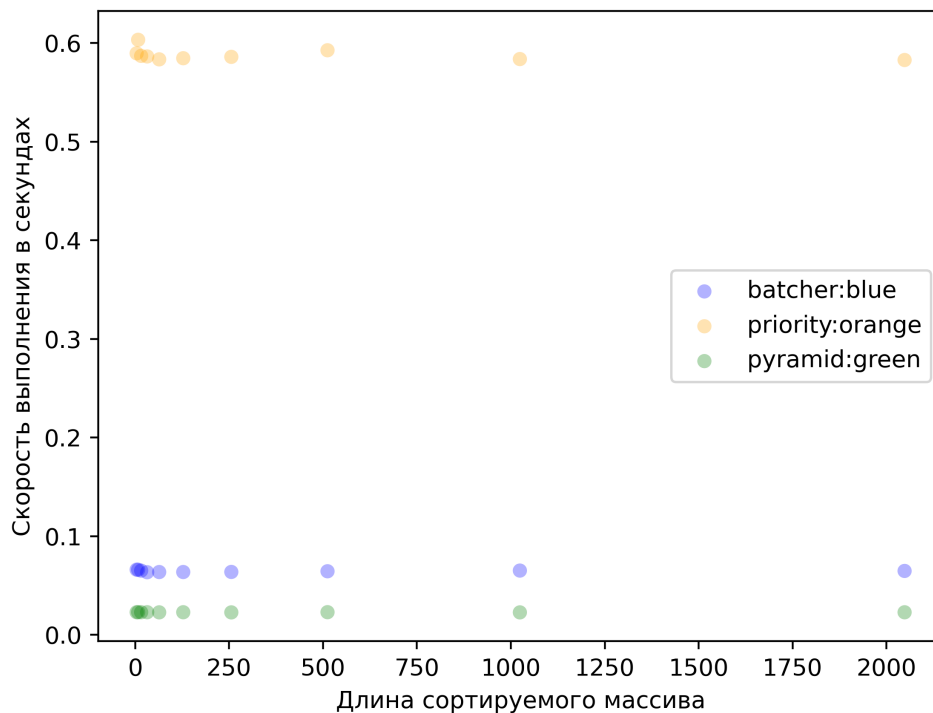


Picture. 12: График сортировки Бэтчера

Скорость выполнения сортировки на основе приоритетных очередей в сравнении с сортировкой Бэтчера



Picture. 13: График сортировки Приоритетных очередей



Picture. 14: График сортировки Пирамидой

3.8 Потребление памяти алгоритмов

Среднее потребление памяти алгоритмов

Программа 3 импортирует 3 подпрограммы сортировки и строит графики для трех алгоритмов сортировки.

По оси x Запуски программ

По оси y Потребление памяти

Исходный код Программы 3

```
1 # for randomizing
2 import random
3
4 # for fast arrays
5 from array import array
6
7 # for beautifull print
8 from pprint import pprint
9
10 # for graphs and plots
11 import numpy as np
12 from matplotlib import pyplot as plt
```

```

13
14 # importing sorts
15 from methods.batcher import bitonic_sortMem
16 from methods.priority_func import priority_sortMem
17 from methods.pyramid import heapSortMem
18
19 arr = list(range(128))
20 arr = array("i", arr) # 'i' - signed int
21
22 ls = []
23
24 for i in range(300):
25     random.shuffle(arr)
26     ls.append(arr)
27
28
29 colors = ["batcher:blue", "priority:orange", "pyramid:green"]
30 ##### Batcher Sort
31 #####
32 exec_memory_bitonic = [
33     [i, bitonic_sortMem(ls[i], 0, 128, 1)] for i in range(100)
34 ] # works
35
36 data = np.array(exec_memory_bitonic)
37
38 x, y = data.T
39 plt.scatter(
40     x,
41     y,
42     c=colors[0].split(":")[1],
43     label=colors[0],
44     alpha=0.3,
45     edgecolors="none",
46 )
47 plt.ylabel("Потребление памяти в байтах")
48 plt.xlabel("Запуски программы сортировки")
49 plt.legend()
50 # plt.show()
51 plt.savefig("research_doc/plots/batcher_memory.png", dpi=300) # savefig,
52     don't show
53 ##### Batcher Sort

```

```

#####
53
54 ##### Priority Sort
#####
55 exec_memory_priority = [
56     [i, priority_sortMem(ls[i + 100])] for i in range(100)
57 ] # for heapSort only
58
59 data = np.array(exec_memory_priority)
60
61 x, y = data.T
62 plt.scatter(
63     x,
64     y,
65     c=colors[1].split(":")[1],
66     label=colors[1],
67     alpha=0.3,
68     edgecolors="none",
69 )
70 plt.ylabel("Потребление памяти в байтах")
71 plt.xlabel("Запуски программы сортировки")
72 plt.legend()
73 # plt.show()
74 plt.savefig("research_doc/plots/priority_memory.png", dpi=300) # savefig,
    don't show
75 ##### Priority Sort
#####
76
77 ##### Pyramid Sort
#####
78 exec_memory_heap = [[i, heapSortMem(ls[i + 200])] for i in range(100)] #
    works
79
80 data = np.array(exec_memory_heap)
81
82 x, y = data.T
83 plt.scatter(
84     x,
85     y,
86     c=colors[2].split(":")[1],
87     label=colors[2],
88     alpha=0.3,

```

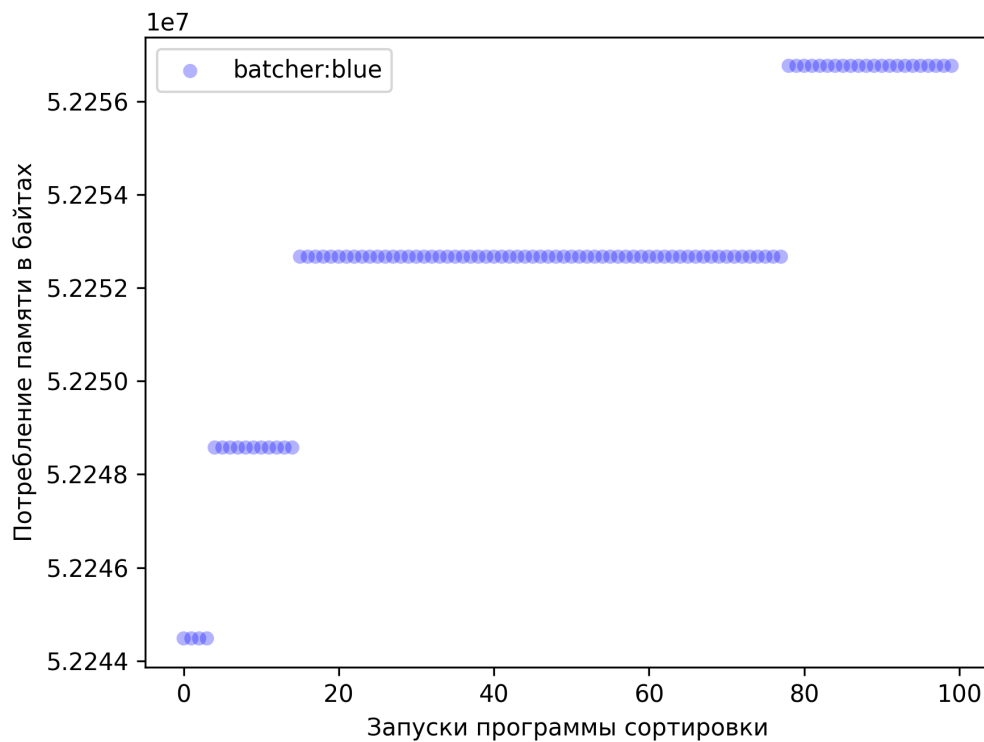
```

89     edgecolors="none",
90 )
91 plt.ylabel("Потребление памяти в байтах")
92 plt.xlabel("Запуски программы сортировки")
93 plt.legend()
94 # plt.show()
95 plt.savefig("research_doc/plots/bitonic_memory.png", dpi=300) # savefig,
    don't show
96 ##### Pyramid Sort
    #####

```

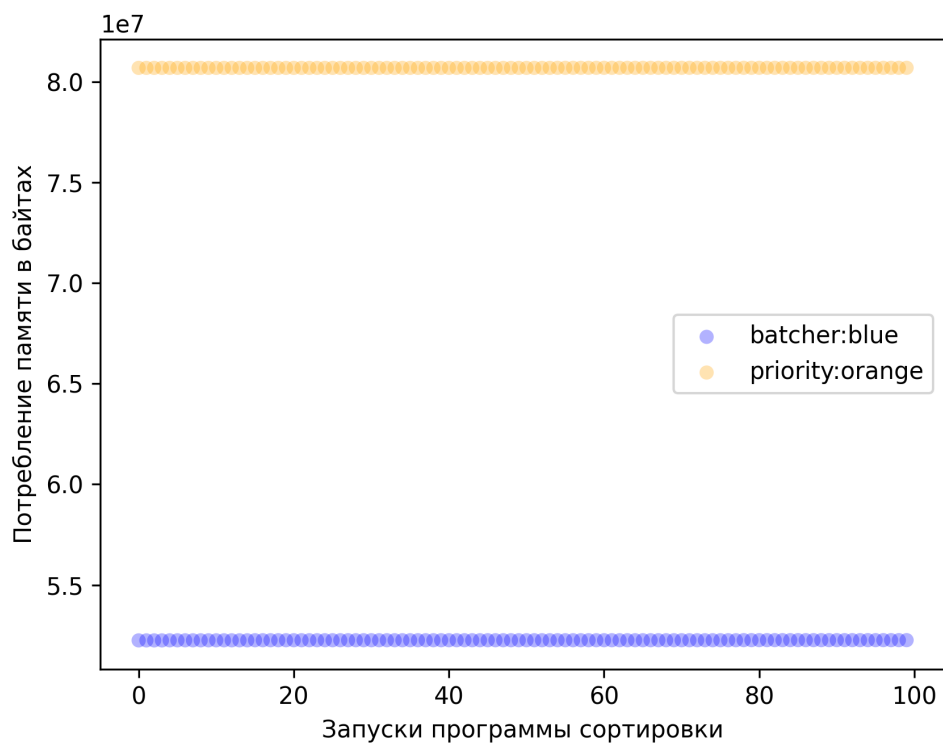
Таким образом были построены графики:

Потребление памяти сортировки Бэтчера



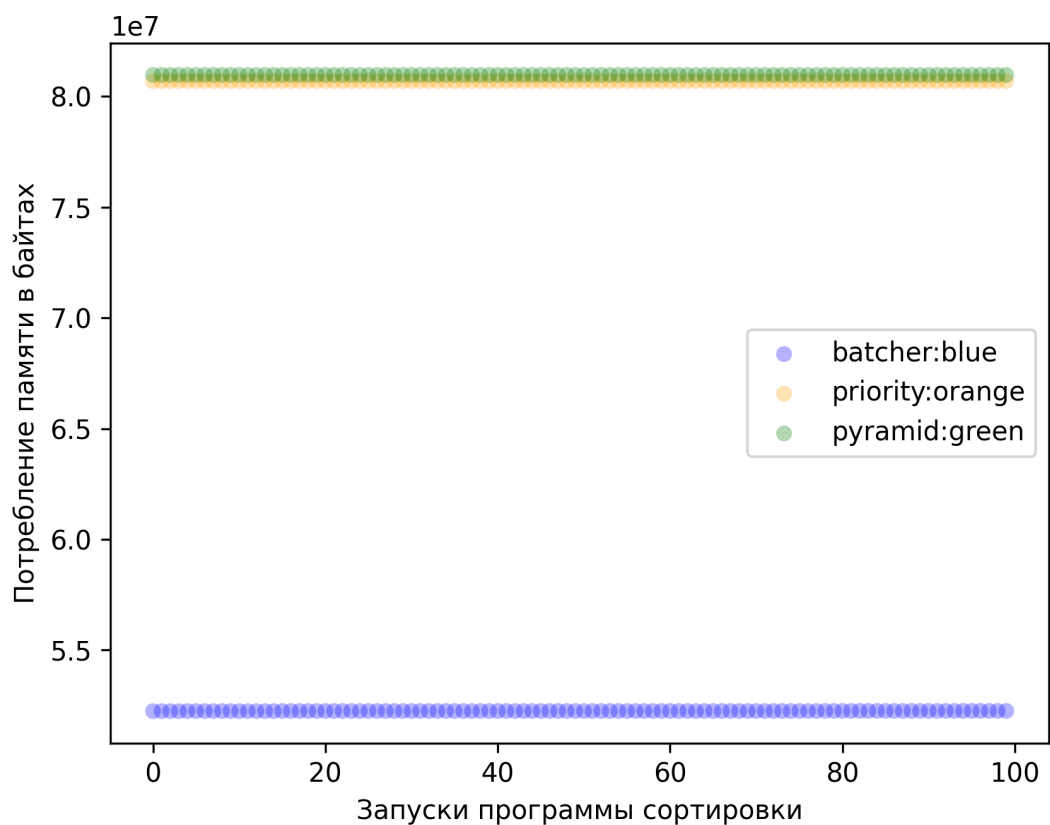
Picture. 15: График сортировки Бэтчера

Потребление памяти сортировки на основе приоритетных очередей в сравнении с сортировкой Бэтчера



Picture. 16: График сортировки Приоритетных очередей

Потребление памяти сортировки Пирамидой в сравнении с остальными двумя сортировками



Picture. 17: График сортировки Пирамидой

Потребление памяти по отношению к длине массива

Программа 3 импортирует 3 подпрограммы сортировки и строит графики для трех алгоритмов сортировки.

По оси x Запуски программ

По оси y Потребление памяти

Исходный код Программы 3

```
1 # for randomizing
2 import random
3 # for fast arrays
4 from array import array
5
6 # for graphs and research_doc/plots
7 import numpy as np
8 from matplotlib import pyplot as plt
9
10 # importing sorts
11 from methods.batcher import bitonic_sortMem
12 from methods.priority_func import priority_sortMem
13 from methods.pyramid import heapSortMem
14
15 arr = list(range(4))
16 arr = array("i", arr) # 'i' - signed int
17
18 ls = []
19
20 for i in range(10):
21     random.shuffle(arr)
22     ls.append(arr)
23     ln_arr = len(arr)
24     arr.extend(list(range(ln_arr, ln_arr * 2)))
25
26
27 colors = ["batcher:blue", "priority:orange", "pyramid:green"]
28 ##### Batcher Sort
29 #####
30 exec_speeds_bitonic = [
31     [2 ** (i + 2), bitonic_sortMem(ls[i], 0, len(ls[i]), 1)] for i in range
32     (10)
33 ] # works
```

```

34 data = np.array(exec_speeds_bitonic)
35
36 x, y = data.T
37
38 plt.scatter(
39     x,
40     y,
41     c=colors[0].split(":")[1],
42     label=colors[0],
43     alpha=0.3,
44     edgecolors="none",
45 )
46 plt.ylabel("Скорость выполнения в секундах")
47 plt.xlabel("Потребление памяти")
48 plt.legend()
49 # plt.show()
50 plt.savefig(
51     "research_doc/plots/batcher_memory_delta.png", dpi=400
52 ) # savefig, don't show
53 ##### Batcher Sort
54 #####
55
56 arr = list(range(4))
57 arr = array("i", arr) # 'i' - signed int
58
59 ls = []
60
61 for i in range(10):
62     random.shuffle(arr)
63     ls.append(arr)
64     ln_arr = len(arr)
65     arr.extend(list(range(ln_arr, ln_arr * 2)))
66
67 ##### Priority Sort
68 #####
69 exec_speeds_priority = [
70     [2 * (i + 2), priority_sortMem(ls[i])] for i in range(10)
71 ] # for heapSort only
72
73 data = np.array(exec_speeds_priority)
74
75 x, y = data.T

```

```

74 plt.scatter(
75     x,
76     y,
77     c=colors[1].split(":")[1],
78     label=colors[1],
79     alpha=0.3,
80     edgecolors="none",
81 )
82 plt.ylabel("Скорость выполнения в секундах")
83 plt.xlabel("Потребление памяти")
84 plt.legend()
85 # plt.show()
86 plt.savefig(
87     "research_doc/plots/priority_memory_delta.png", dpi=400
88 ) # savefig, don't show
89 ##### Priority Sort
90 #####
91 arr = list(range(4))
92 arr = array("i", arr) # 'i' - signed int
93
94 ls = []
95
96 for i in range(10):
97     random.shuffle(arr)
98     ls.append(arr)
99     ln_arr = len(arr)
100     arr.extend(list(range(ln_arr, ln_arr * 2)))
101
102 ##### Pyramid Sort
103 #####
104 exec_speeds_heap = [[2 ** (i + 2), heapSortMem(ls[i])] for i in range(10)]
105 # works
106
107 data = np.array(exec_speeds_heap)
108
109 x, y = data.T
110 plt.scatter(
111     x,
112     y,
113     c=colors[2].split(":")[1],
114     label=colors[2],

```



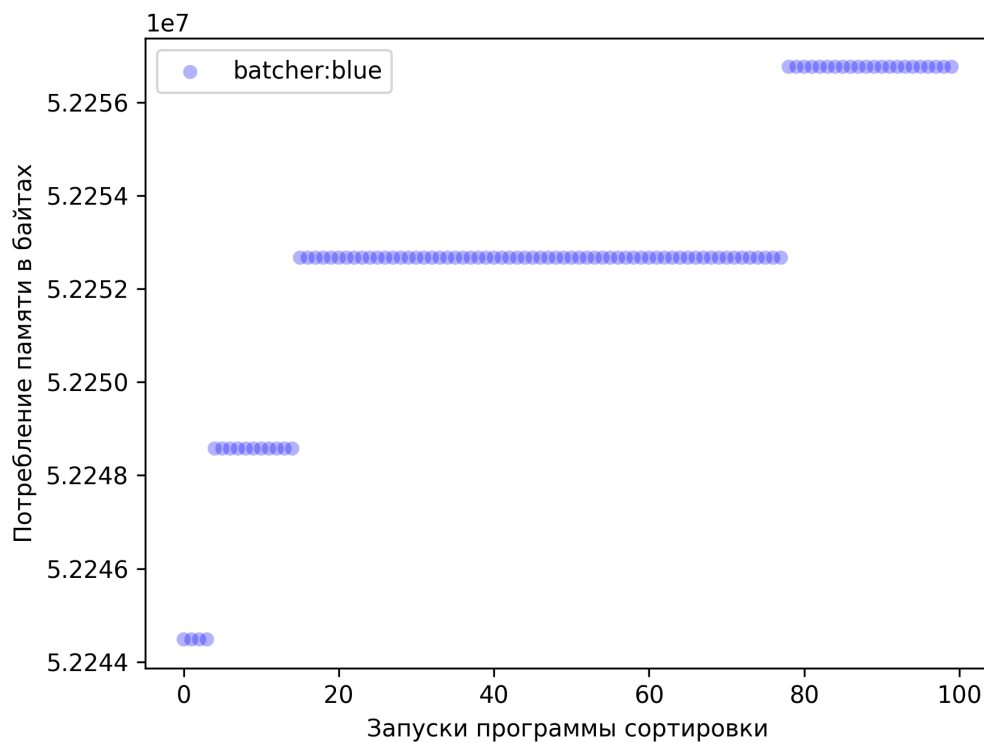
```

113     alpha=0.3,
114     edgecolors="none",
115 )
116 plt.ylabel("Скорость выполнения в секундах")
117 plt.xlabel("Потребление памяти")
118 plt.legend()
119 # plt.show()
120 plt.savefig(
121     "research_doc/plots/bitonic_memory_delta.png", dpi=400
122 ) # savefig, don't show
123 ##### Pyramid Sort
    #####

```

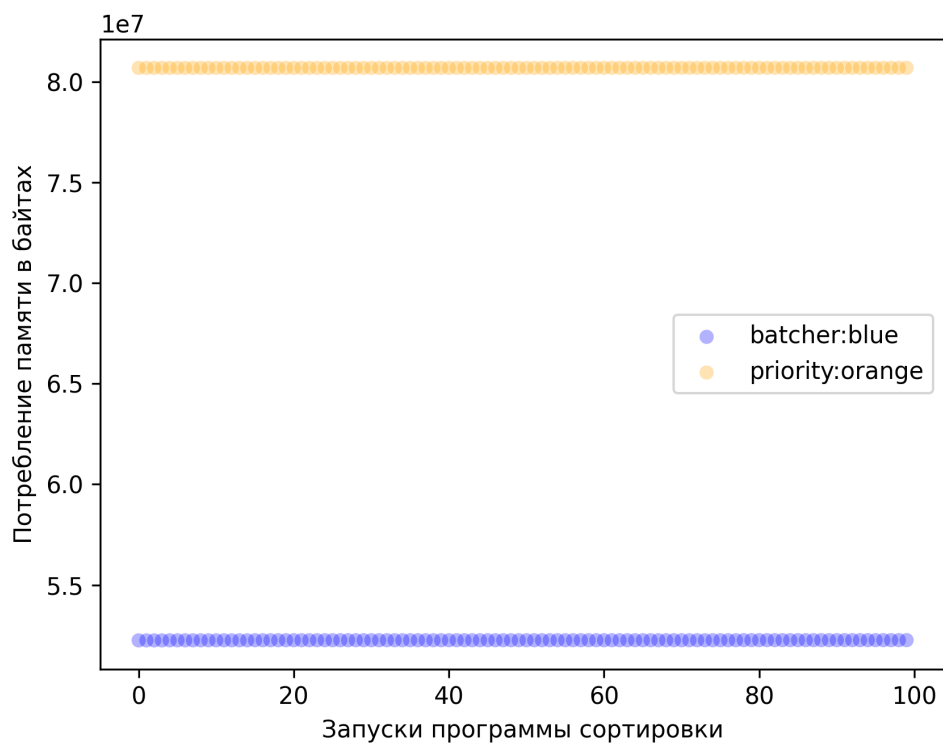
Таким образом были построены графики:

Потребление памяти сортировки Бэтчера



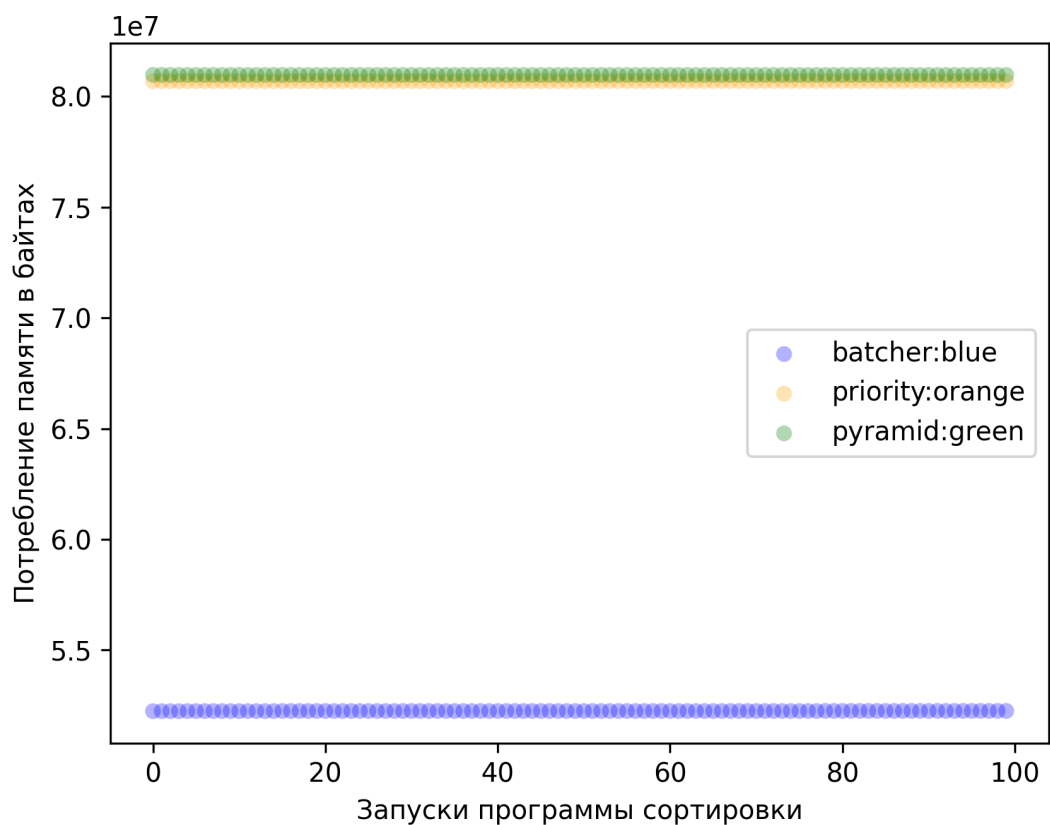
Picture. 18: График сортировки Бэтчера

Потребление памяти сортировки на основе приоритетных очередей в сравнении с сортировкой Бэтчера



Picture. 19: График сортировки Приоритетных очередей

Потребление памяти сортировки Пирамидой в сравнении с остальными двумя сортировками



Picture. 20: График сортировки Пирамидой

4 Выводы по трем алгоритмам

Учитывая полученные данные и графики можно прийти к нескольким выводам, а именно:

1. При длине массива не более 10000 скорость сортировки не особо меняется
2. Все три алгоритма относительно стабильны и получая на входе одинаковые массивы, они сортируют их с практически равной скоростью
3. Потребление памяти также особо не меняется, возможно из-за языка Python, который сам по себе потребляет достаточно много памяти
4. Тоже самое можно сказать и о стабильности потребления памяти
5. Самым быстрым оказался Алгоритм сортировки Пирамидой
6. Самым экономным по памяти Оказался Алгоритм сортировки Бэтчера,
7. Если вам нужна скорость сортировки и у вас в доступе достаточное количество памяти, следует использовать Алгоритм сортировки Пирамидой
8. Если же у вас недостаточно памяти, то следует использовать Алгоритм сортировки Бэтчера

5 Ссылки

Ссылка на весь код представленный в этом документе, а также исходный код самого документа вы можете найти по этой ссылке

<https://github.com/orenvadi/LAB2>