СРСП №2

WIN-1-22

Алгоритмы обработки массивов

Выполнил:

Бактыбеков Н.Б.

Проверил:

Картанова А.Д.

April 6, 2023

Contents

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:

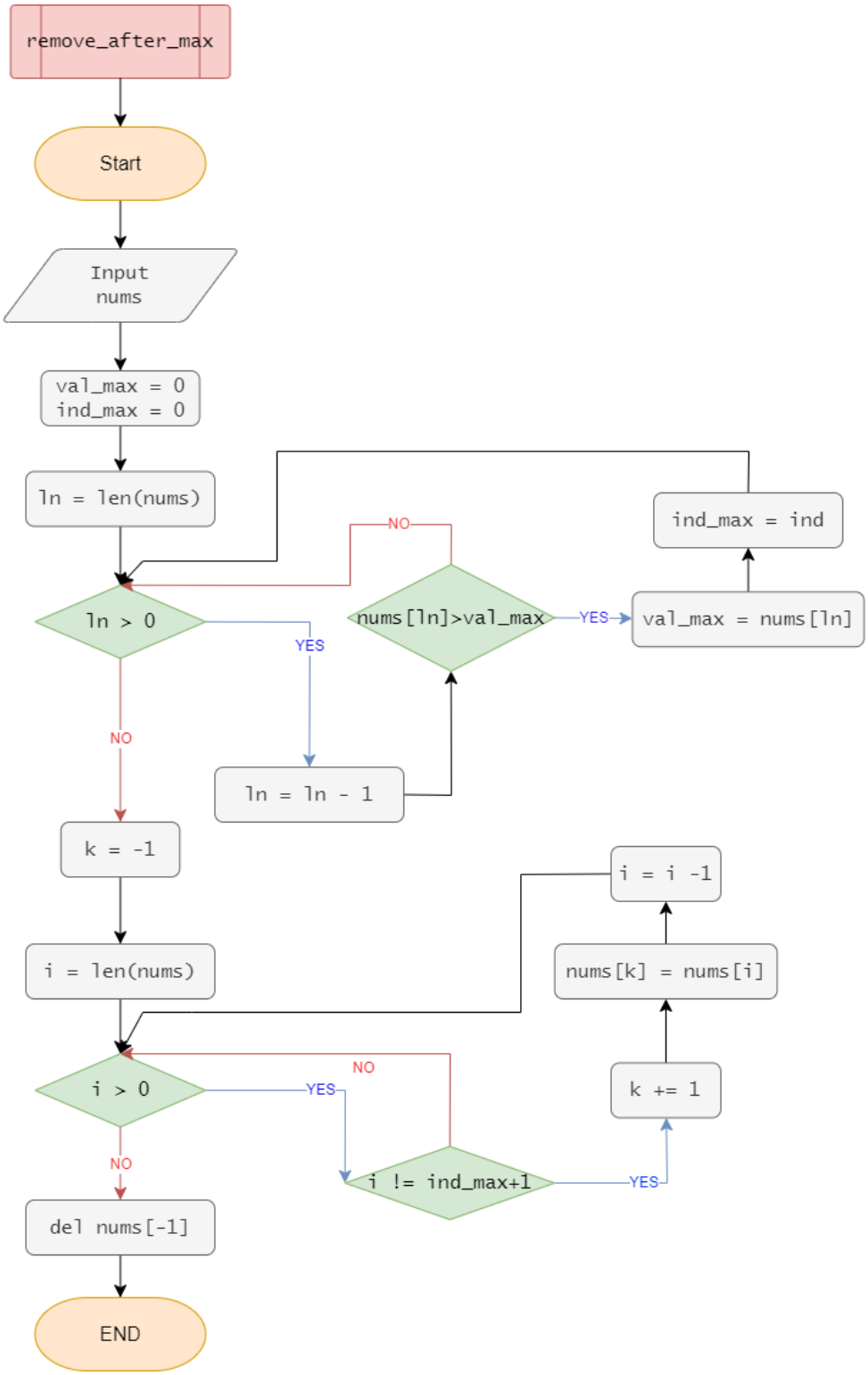
arr - результирующий массив из N-1 элементов.

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

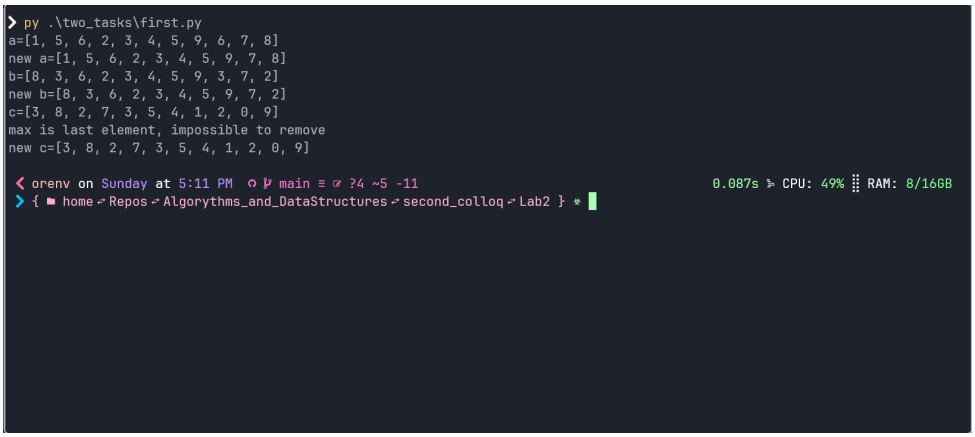
Реализация задачи 1 оформлена ввиде функции в программе first-task.py

1.6 Исходный код программы

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


def remove_after_max(nums: list[int]):
    # finding max value

    val_max = 0
    ind_max = 0
    for ind, num in enumerate(nums):
        if num > val_max:
            val_max = num
            ind_max = ind

    if ind_max == len(nums) - 1:
        print("max is last element, impossible to remove")
        return -1

    k = -1

    for i in range(0, len(nums)):
        if i != ind_max + 1:  # elem after max
            k += 1
            nums[k] = nums[i]
    del nums[-1]
    return len(nums)

```

```

a = [1, 5, 6, 2, 3, 4, 5, 9, 6, 7, 8]
b = [8, 3, 6, 2, 3, 4, 5, 9, 3, 7, 2]
c = [3, 8, 2, 7, 3, 5, 4, 1, 2, 0, 9]
print(f"{a=}")
remove_after_max(a)
print(f"new a={a}")
print(f"{b=}")
remove_after_max(b)
print(f"new b={b}")
print(f"{c=}")
remove_after_max(c)
print(f"new c={c}")
```

1.7 Результаты работы программы



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:

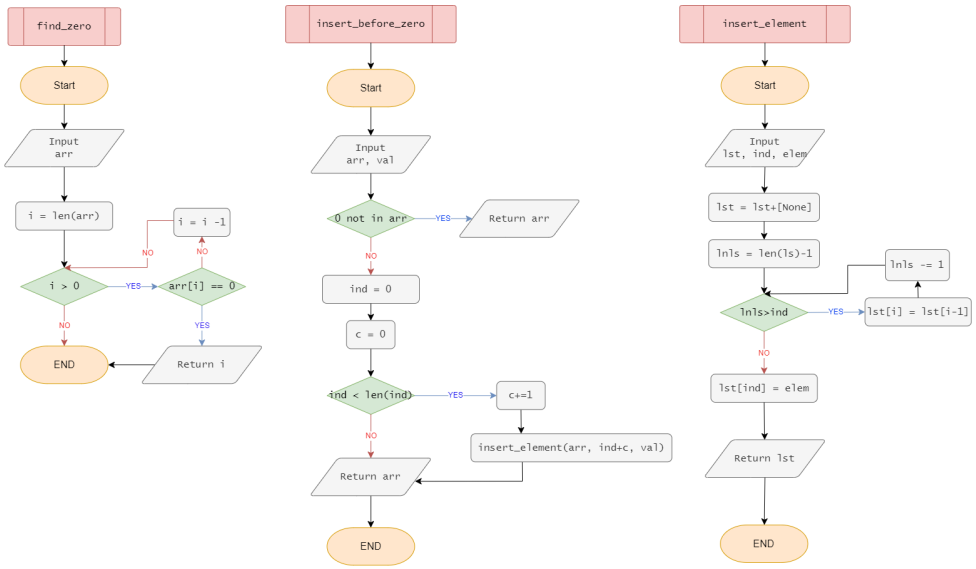
arr - результирующий массив из N+индекс-нуля элементов.

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]

   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]

   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]

   new c=[0, 8, 2, 7, 3, 5, 4, 1, 2, 0, 9]

2.5 Реализация решения задачи

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

2.6 Исходный код программы

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

```python


def find_zero(arr):
    for i, e in enumerate(arr):
        if e == 0:
            return i


def insert_before_zero(val, arr: list):
    if 0 not in arr or arr[0] == 0:
        print("Error: no zero element found")
        return arr

    c = 0

    for ind, elem in enumerate(arr[: find_zero(arr)]):
        c += 1
        insert_element(arr, ind + c, val)

    return arr


def insert_element(lst, index, element):
    lst.append(None)
    for i in range(len(lst) - 1, index, -1):
        lst[i] = lst[i - 1]
    lst[index] = element
    return lst


val = 10
a = [1, 5, 6, 0, 3, 4, 5, 9, 6, 7, 8]

print(f"{a=}")
a = insert_before_zero(val, a)
print(f"new a={a}")

b = [8, 3, 6, 2, 3, 4, 5, 0, 3, 7, 2]

print(f"{b=}")
b = insert_before_zero(val, b)
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

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:

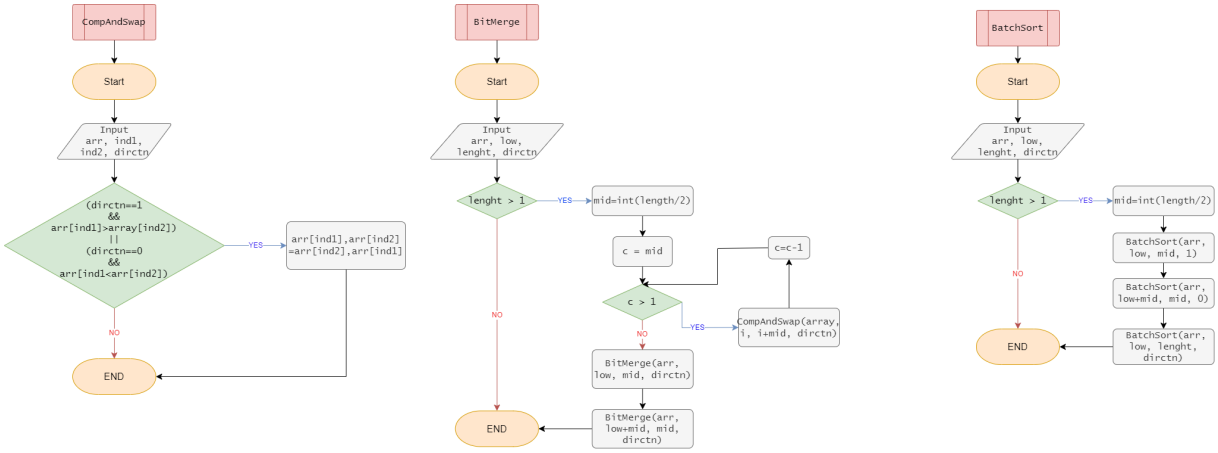
arr – список типа list, заполненный случайным образом

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 $O(n * logn)$

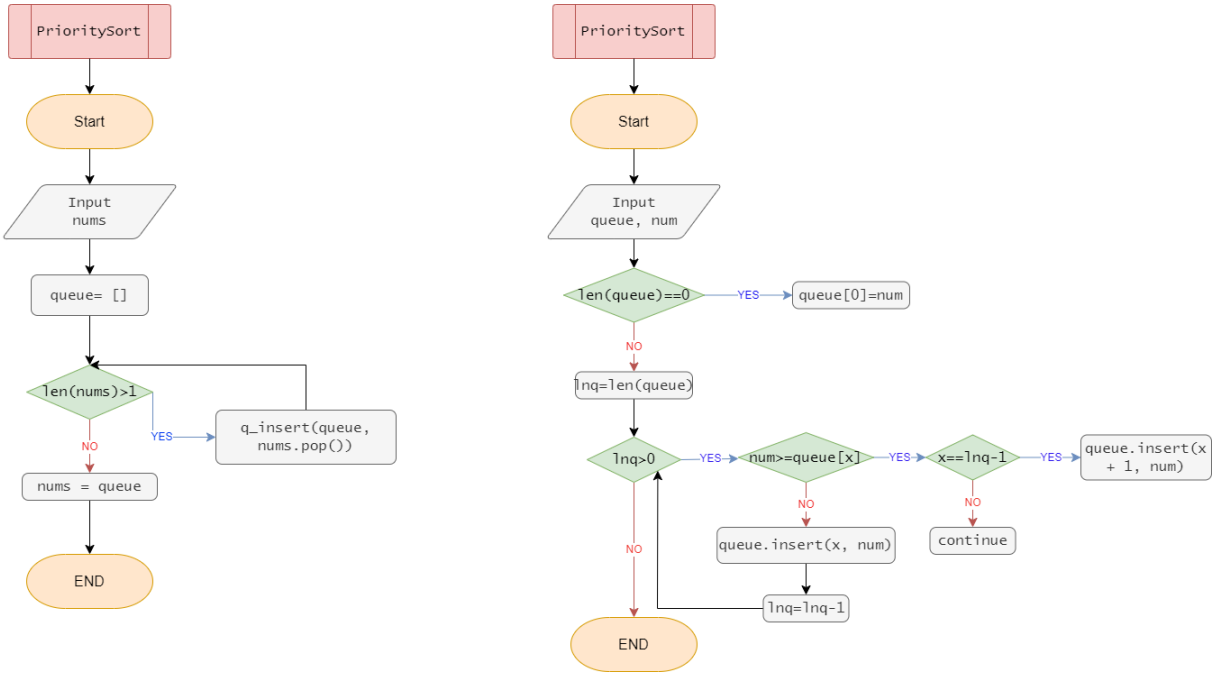
Код программы с сортировкой Бэтчера

```
"""
Python program for Bitonic Sort.
Note that this program works only when size of input is a power of 2.
"""
from __future__ import annotations

from .wrappers_memory import profile
from .wrappers_speed import speedometer


def comp_and_swap(array: list[int], index1: int, index2: int, direction:
    int) -> None:
    """Compare the value at given index1 and index2 of the array and swap
    them as per
    the given direction.
    The parameter direction indicates the sorting direction, ASCENDING(1)
    or
    DESCENDING(0); if (a[i] > a[j]) agrees with the direction, then a[i]
    and a[j] are
    interchanged.
    """
    if (direction == 1 and array[index1] > array[index2]) or (
```

```
        direction == 0 and array[index1] < array[index2]
    ):
        array[index1], array[index2] = array[index2], array[index1]


def bitonic_merge(array: list[int], low: int, length: int, direction: int)
    -> None:
    """
    It recursively sorts a bitonic sequence in ascending order, if
    direction = 1, and in
    descending if direction = 0.
    The sequence to be sorted starts at index position low, the parameter
    length is the
    number of elements to be sorted.
    """
    if length > 1:
        middle = int(length / 2)
        for i in range(low, low + middle):
            print(array)
            comp_and_swap(array, i, i + middle, direction)
        bitonic_merge(array, low, middle, direction)
        bitonic_merge(array, low + middle, middle, direction)


@speedometer
def bitonic_sort(array: list[int], low: int, length: int, direction: int)
    -> None:
    """
    This function first produces a bitonic sequence by recursively sorting
    its two
    halves in opposite sorting orders, and then calls bitonic_merge to make
     them in the
    same order.
    """
    if length > 1:
        middle = int(length / 2)
        bitonic_sort(array, low, middle, 1)
        bitonic_sort(array, low + middle, middle, 0)
        bitonic_merge(array, low, length, direction)


@profile
```

```
def bitonic_sortMem(array: list[int], low: int, length: int, direction: int
    ) -> None:
    if length > 1:
        middle = int(length / 2)
        bitonic_sort(array, low, middle, 1)
        bitonic_sort(array, low + middle, middle, 0)
        bitonic_merge(array, low, length, direction)
```

Блок-схема подпрограммы сортировки Приоритетных очередей:



Picture. 6: Сортировка на основе Приоритетных очередей

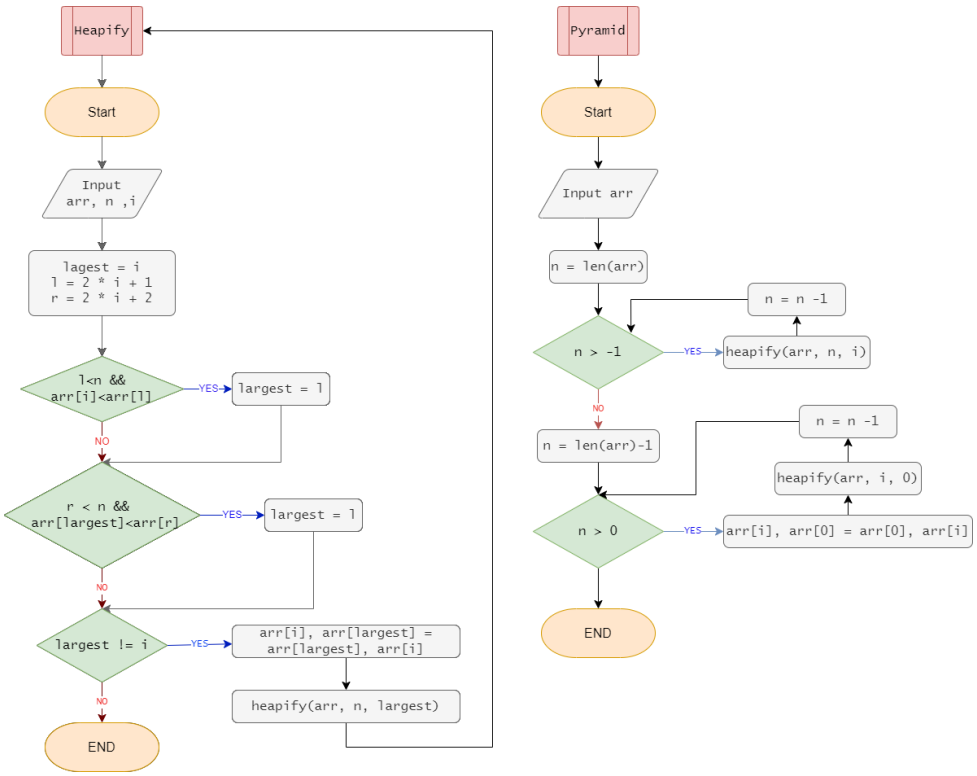
Космлексность данного алгоритма составляет $O(log^2 n)$

Код программы с сортировкой на основе Приоритетных очередей

```
from .wrappers_memory import profile
from .wrappers_speed import speedometer


@speedometer
def priority_sort(nums: list[int]):
    queue = []
    for num in nums:
        q_insert(queue, num)
        print(queue)
    nums = queue


@profile
```

```python
def priority_sortMem(nums: list[int]):
    queue = []
    for num in nums:
        q_insert(queue, num)
    nums = queue


def q_insert(queue: list, num):
    # if queue is empty
    if len(queue) == 0:
        # add the new num
        queue.append(num)
    else:
        # traverse the queue to find the right place for new num
        for x in range(0, len(queue)):
            # if the of new num is greater
            if num >= queue[x]:
                # if we have traversed the complete queue
                if x == (len(queue) - 1):
                    # add new num at the end
                    queue.insert(x + 1, num)
                else:
                    continue
            else:
                queue.insert(x, num)
                return True
```

Блок-схема подпрограммы сортировки Пирамидой:



Picture. 7: Запуск программы Задание 2

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

Код программы с сортировкой Пирамидой

```
from .wrappers_memory import profile
from .wrappers_speed import speedometer

# Implementation of heapsort in Python

# Procedure to convert to a binary heap a subtree with root node i, which
    is an index in arr[]. n - heap size


def heapify(arr, n, i):
    largest = i  # Initialize largest as root
    l = 2 * i + 1  # left = 2*i + 1
    r = 2 * i + 2  # right = 2*i + 2

    # check if exists left child elem > root

    if l < n and arr[i] < arr[l]:
        # print(arr)
        largest = l

    # 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]  # свап
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]  # свап
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]  # свап
61          heapify(arr, i, 0)
```

3.4 Контрольные примеры

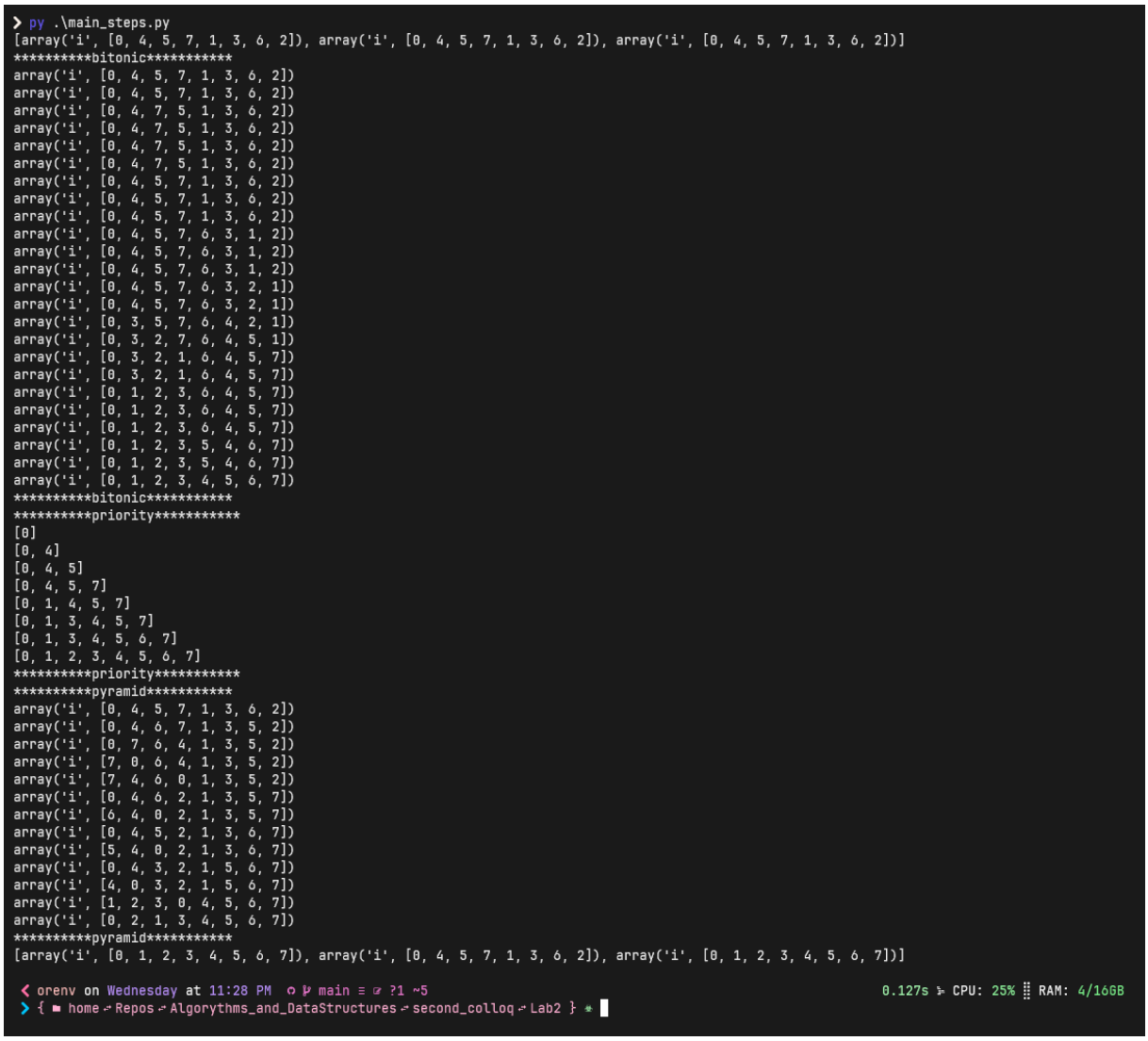
Входные данные:

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



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

```
# for randomizing
import random
# for fast arrays
from array import array

from prettytable import PrettyTable
# for table
from tabletexifier import Table

# importing sorts
from methods.batcher import bitonic_sort, bitonic_sortMem
from methods.priority_func import priority_sort, priority_sortMem
from methods.pyramid import heapSort, heapSortMem

# for beautifull print
# from pprint import pprint

# for graphs and plots
# import numpy as np
# from matplotlib import pyplot as plt


arr = list(range(128))
arr = array("i", arr)  # 'i' - signed int

ls = []

for i in range(600):
    random.shuffle(arr)
    ls.append(arr)


def save_to_tex(table: Table, path: str):
    with open(path, "w", encoding="utf-8") as file:
        pass
    txt = table.build_latex()
    with open(path, "w", encoding="utf-8") as file:
```

```
        file.writelines(txt.split("\n"))

    # Read in the file
    with open(path, "r", encoding="utf-8") as file:
        filedata = file.read()
    # Replace the target string
    filedata = filedata.replace("begin{table}", "begin{table}[H]")
    filedata = filedata.replace(
        "{\\label{Tab:}", ("{" + path.split("/")[2][:-10] + " sort")
    )
    # Write the file out again
    with open(path, "w", encoding="utf-8") as file:
        file.write(filedata)


###################################### Batcher Sort
   ############################################
exec_speeds_bitonic = [
    [i + 1, bitonic_sort(ls[i], 0, 128, 1), bitonic_sortMem(ls[i + 100], 0,
    128, 1)]
    for i in range(40)
]  # works

bitonic_table = Table(
    [
        "Запуск программы",
        "Скорость выполнения",
        "Потребление памяти",
    ],
    table_style="A",
)
# bitonic_table.title = "Batcher sort"

for row in exec_speeds_bitonic:
    bitonic_table.add_row(row)

save_to_tex(bitonic_table, "research_doc/tables/batcher_table.tex")
###################################### Batcher Sort
   ############################################


###################################### Pyramid Sort
```

```
    ##########################################
exec_speeds_heap = [
    [i + 1, heapSort(ls[i + 200]), heapSortMem(ls[i + 300])] for i in range
    (40)
]  # works

heap_table = Table(
    [
        "Запуск программы",
        "Скорость выполнения",
        "Потребление памяти",
    ],
    table_style="A",
)
# heap_table.title = "Pyramid sort"

for row in exec_speeds_heap:
    heap_table.add_row(row)

save_to_tex(heap_table, "research_doc/tables/pyramid_table.tex")

###################################### Pyramid Sort
    ##########################################


###################################### Priority Sort
    ##########################################
exec_speeds_priority = [
    [i + 1, priority_sort(ls[i + 400]), priority_sortMem(ls[i + 500])]
    for i in range(40)
]  # works

priority_table = Table(
    [
        "Запуск программы",
        "Скорость выполнения",
        "Потребление памяти",
    ],
    table_style="A",
)
# priority_table.title = "Pyramid sort"

```

```
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

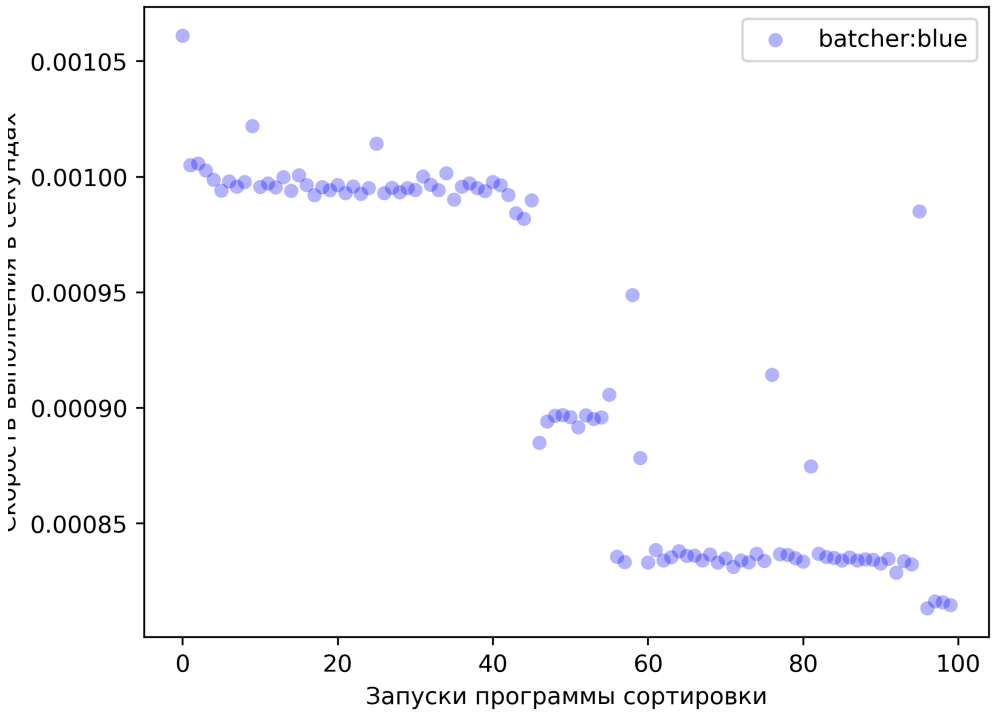
```
# for randomizing
import random
# for fast arrays
from array import array

# for graphs and research_doc/plots
import numpy as np
from matplotlib import pyplot as plt

# importing sorts
from methods.batcher import bitonic_sort
from methods.priority_func import priority_sort
from methods.pyramid import heapSort

arr = list(range(128))
arr = array("i", arr)  # 'i' - signed int

ls = []

for i in range(300):
    random.shuffle(arr)
    ls.append(arr)


colors = ["batcher:blue", "priority:orange", "pyramid:green"]
######################################## Batcher Sort
   ##############################################
exec_speeds_bitonic = [[i, bitonic_sort(ls[i], 0, 128, 1)] for i in range
   (100)]  # works


data = np.array(exec_speeds_bitonic)

x, y = data.T

plt.scatter(
```

```
    x,
    y,
    c=colors[0].split(":")[1],
    label=colors[0],
    alpha=0.3,
    edgecolors="none",
)
plt.ylabel("Скорость выполнения в секундах")
plt.xlabel("Запуски программы сортировки")
plt.legend()
# plt.show()
plt.savefig("research_doc/plots/batcher_speed.png", dpi=400)  # savefig, don't show
######################################## Batcher Sort ##############################################

######################################## Priority Sort ##############################################
exec_speeds_priority = [
    [i, priority_sort(ls[i + 100])] for i in range(100)
]  # for heapSort only

data = np.array(exec_speeds_priority)

x, y = data.T
plt.scatter(
    x,
    y,
    c=colors[1].split(":")[1],
    label=colors[1],
    alpha=0.3,
    edgecolors="none",
)
plt.ylabel("Скорость выполнения в секундах")
plt.xlabel("Запуски программы сортировки")
plt.legend()
# plt.show()
plt.savefig("research_doc/plots/priority_speed.png", dpi=400)  # savefig, don't show
######################################## Priority Sort ##############################################

```

```
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
       ##########################################
```

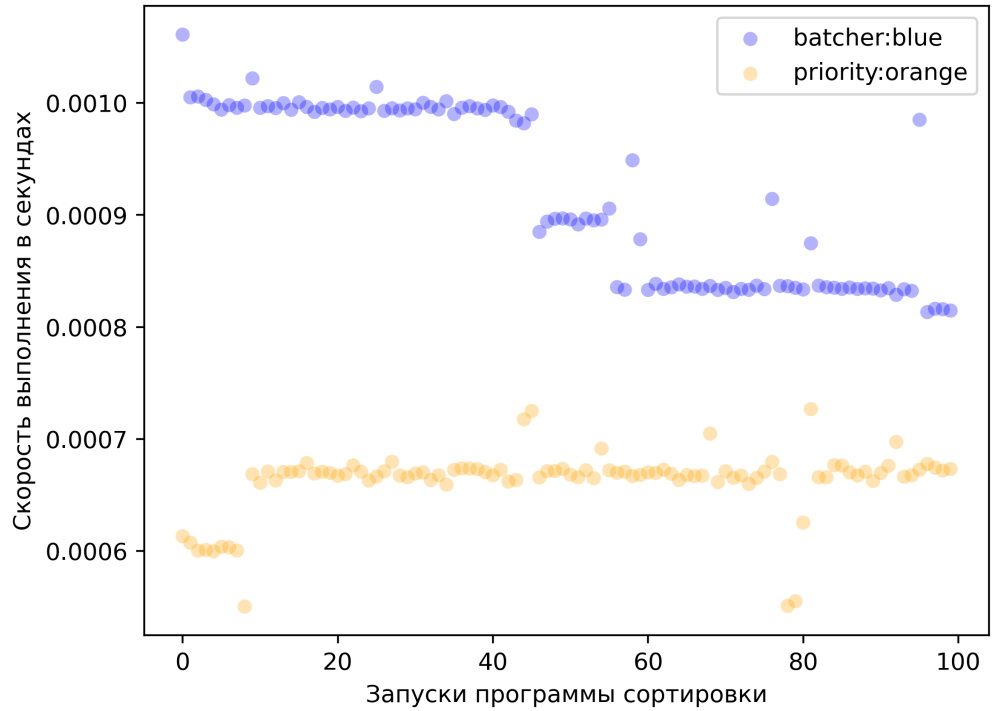
Таким образом были построены графики:

Скорость выполнения сортировки Бэтчера

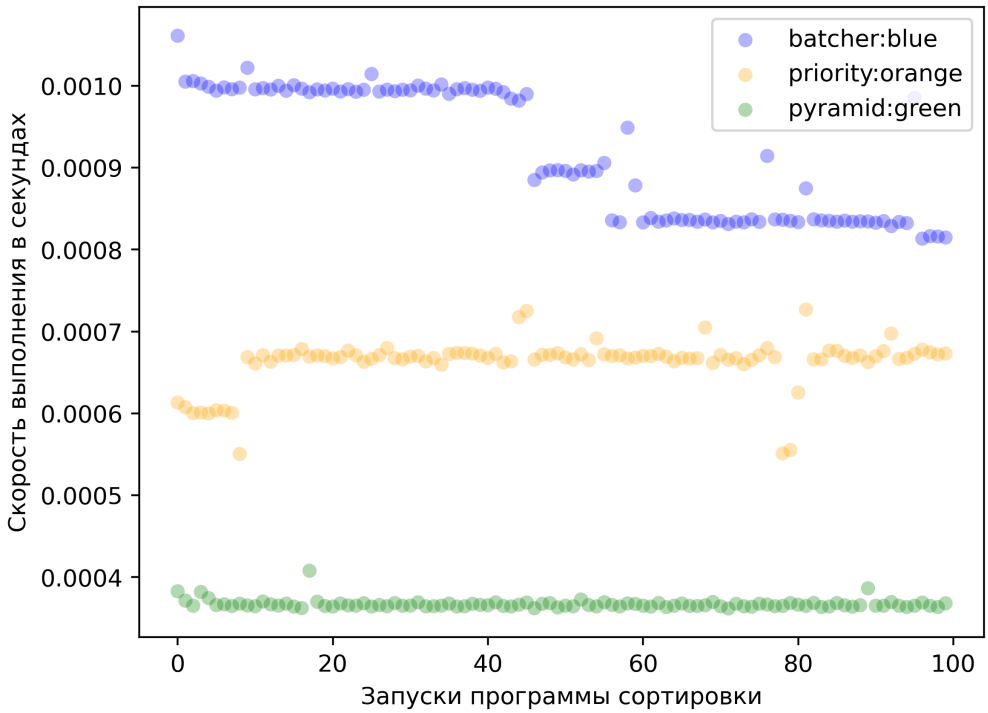


Picture. 9: График сортировки Бэтчера

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

```
# for randomizing
import random
# for fast arrays
from array import array

# for graphs and research_doc/plots
import numpy as np
from matplotlib import pyplot as plt

# importing sorts
from methods.batcher import bitonic_sort
from methods.priority_func import priority_sort
from methods.pyramid import heapSort

```

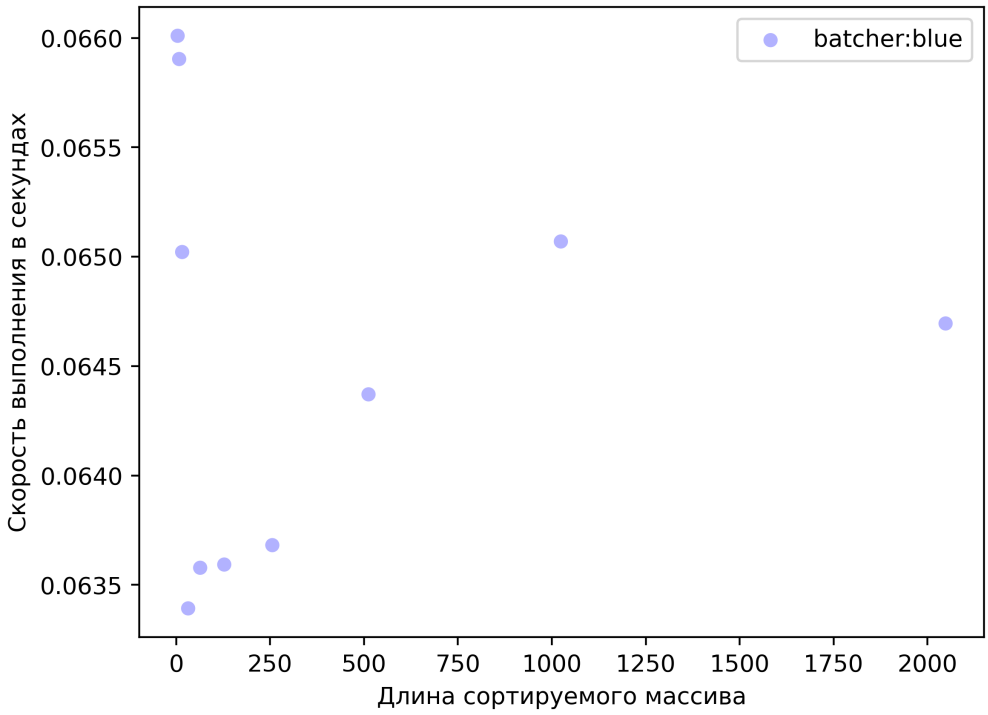
```
arr = list(range(4))
arr = array("i", arr)  # 'i' - signed int

ls = []

for i in range(10):
    random.shuffle(arr)
    ls.append(arr)
    ln_arr = len(arr)
    arr.extend(list(range(ln_arr, ln_arr * 2)))


colors = ["batcher:blue", "priority:orange", "pyramid:green"]
######################################## Batcher Sort
    ##########################################
exec_speeds_bitonic = [
    [2 ** (i + 2), bitonic_sort(ls[i], 0, len(ls[i]), 1)] for i in range
    (10)
]  # works


data = np.array(exec_speeds_bitonic)

x, y = data.T

plt.scatter(
    x,
    y,
    c=colors[0].split(":")[1],
    label=colors[0],
    alpha=0.3,
    edgecolors="none",
)
plt.ylabel("Скорость выполнения в секундах")
plt.xlabel("Длина сортируемого массива")
plt.legend()
# plt.show()
plt.savefig(
    "research_doc/plots/batcher_speed_delta.png", dpi=400
)  # savefig, don't show
######################################## Batcher Sort
    ##########################################
```

```
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 exec_speeds_priority = [
68     [2 ** (i + 2), priority_sort(ls[i])] for i in range(10)
69 ]  # for heapSort only
70
71 data = np.array(exec_speeds_priority)
72
73 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("Длина сортируемого массива")
84 plt.legend()
85 # plt.show()
86 plt.savefig(
87     "research_doc/plots/priority_speed_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
```

```
ls = []

for i in range(10):
    random.shuffle(arr)
    ls.append(arr)
    ln_arr = len(arr)
    arr.extend(list(range(ln_arr, ln_arr * 2)))

######################################## Pyramid Sort
    ##############################################
exec_speeds_heap = [[2 ** (i + 2), heapSort(ls[i])] for i in range(10)]  #
    works

data = np.array(exec_speeds_heap)

x, y = data.T
plt.scatter(
    x,
    y,
    c=colors[2].split(":")[1],
    label=colors[2],
    alpha=0.3,
    edgecolors="none",
)
plt.ylabel("Скорость выполнения в секундах")
plt.xlabel("Длина сортируемого массива")
plt.legend()
# plt.show()
plt.savefig(
    "research_doc/plots/bitonic_speed_delta.png", dpi=400
)  # savefig, don't show
######################################## Pyramid Sort
    ##############################################
```

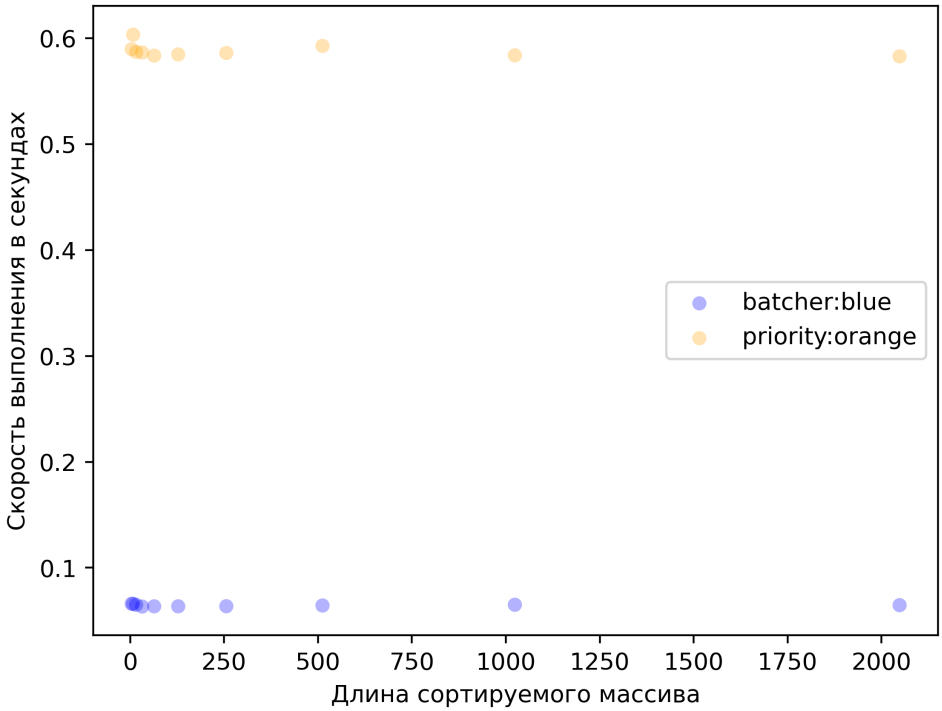
Таким образом были построены графики:

Скорость выполнения сортировки Бэтчера

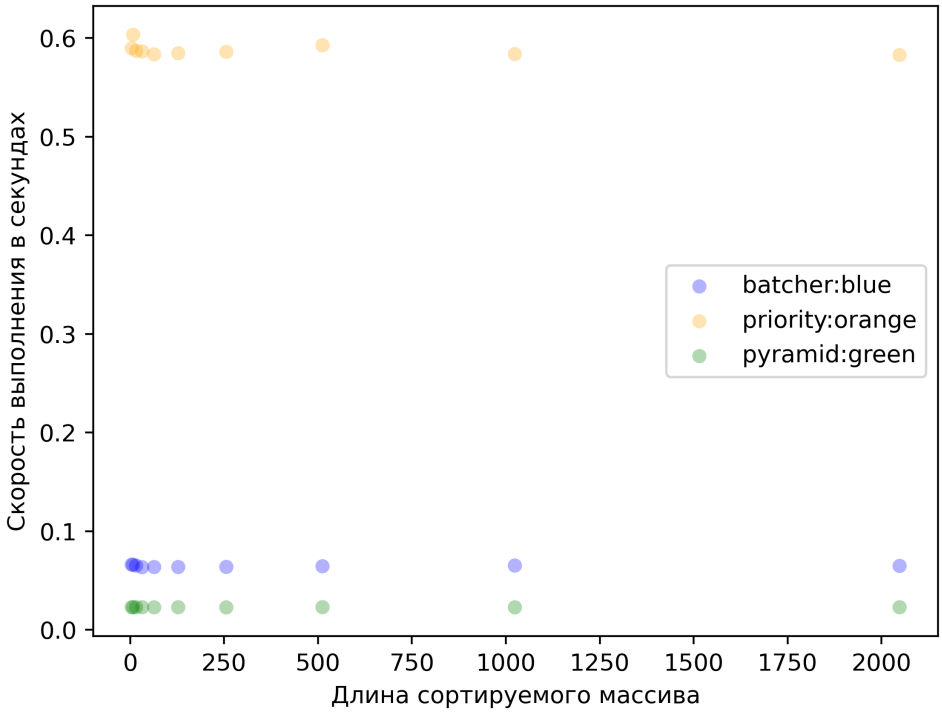


Picture. 12: График сортировки Бэтчера

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

```
# for randomizing
import random

# for fast arrays
from array import array

# for beautifull print
from pprint import pprint

# for graphs and plots
import numpy as np
from matplotlib import pyplot as plt
```

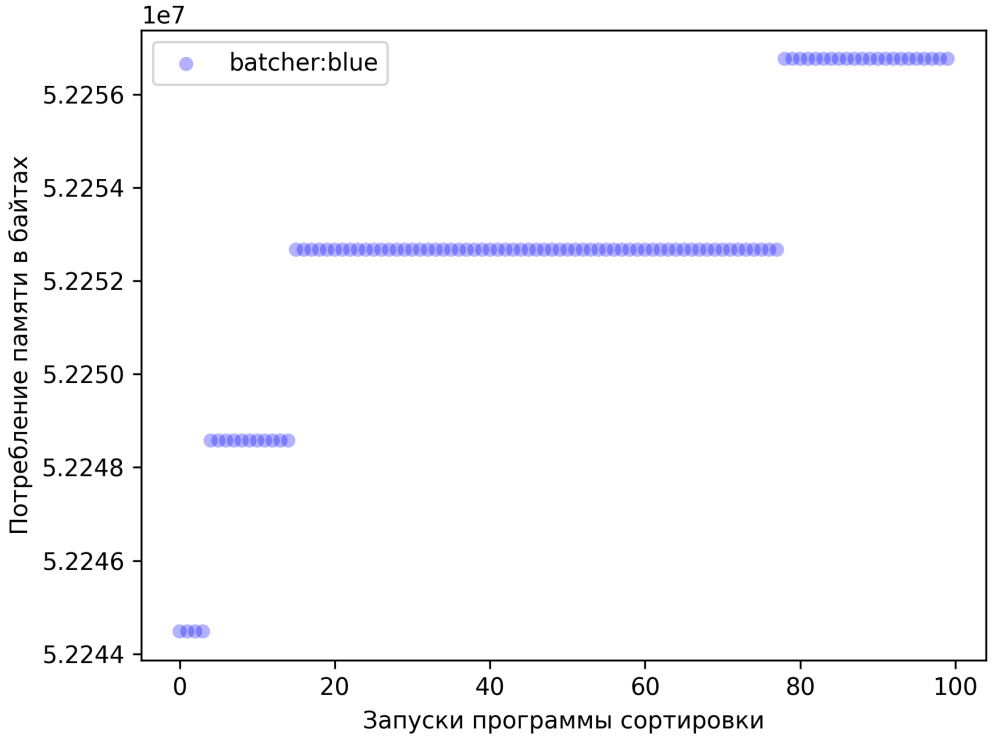
```python

# importing sorts
from methods.batcher import bitonic_sortMem
from methods.priority_func import priority_sortMem
from methods.pyramid import heapSortMem

arr = list(range(128))
arr = array("i", arr)  # 'i' - signed int

ls = []

for i in range(300):
    random.shuffle(arr)
    ls.append(arr)


colors = ["batcher:blue", "priority:orange", "pyramid:green"]
######################################## Batcher Sort
    ##############################################
exec_memory_bitonic = [
    [i, bitonic_sortMem(ls[i], 0, 128, 1)] for i in range(100)
]  # works


data = np.array(exec_memory_bitonic)

x, y = data.T
plt.scatter(
    x,
    y,
    c=colors[0].split(":")[1],
    label=colors[0],
    alpha=0.3,
    edgecolors="none",
)
plt.ylabel("Потребление памяти в байтах")
plt.xlabel("Запуски программы сортировки")
plt.legend()
# plt.show()
plt.savefig("research_doc/plots/batcher_memory.png", dpi=300)  # savefig,
    don't show
######################################## Batcher Sort
```

```
    ##########################################

##################################### Priority Sort
    ##########################################
exec_memory_priority = [
    [i, priority_sortMem(ls[i + 100])] for i in range(100)
]  # for heapSort only

data = np.array(exec_memory_priority)

x, y = data.T
plt.scatter(
    x,
    y,
    c=colors[1].split(":")[1],
    label=colors[1],
    alpha=0.3,
    edgecolors="none",
)
plt.ylabel("Потребление памяти в байтах")
plt.xlabel("Запуски программы сортировки")
plt.legend()
# plt.show()
plt.savefig("research_doc/plots/priority_memory.png", dpi=300)  # savefig,
    don't show
##################################### Priority Sort
    ##########################################

##################################### Pyramid Sort
    ##########################################
exec_memory_heap = [[i, heapSortMem(ls[i + 200])] for i in range(100)]  #
    works

data = np.array(exec_memory_heap)

x, y = data.T
plt.scatter(
    x,
    y,
    c=colors[2].split(":")[1],
    label=colors[2],
    alpha=0.3,
```

```
    edgecolors="none",
)
plt.ylabel("Потребление памяти в байтах")
plt.xlabel("Запуски программы сортировки")
plt.legend()
# plt.show()
plt.savefig("research_doc/plots/bitonic_memory.png", dpi=300)  # savefig, don't show
######################################## Pyramid Sort ##############################################
```

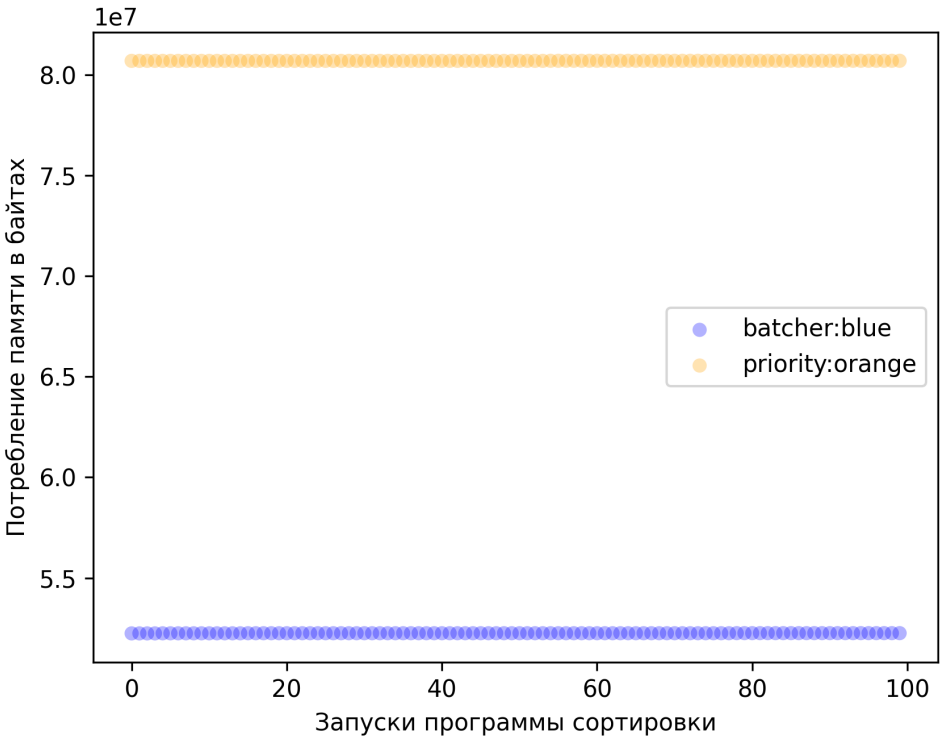
Таким образом были построены графики:

Потребление памяти сортировки Бэтчера



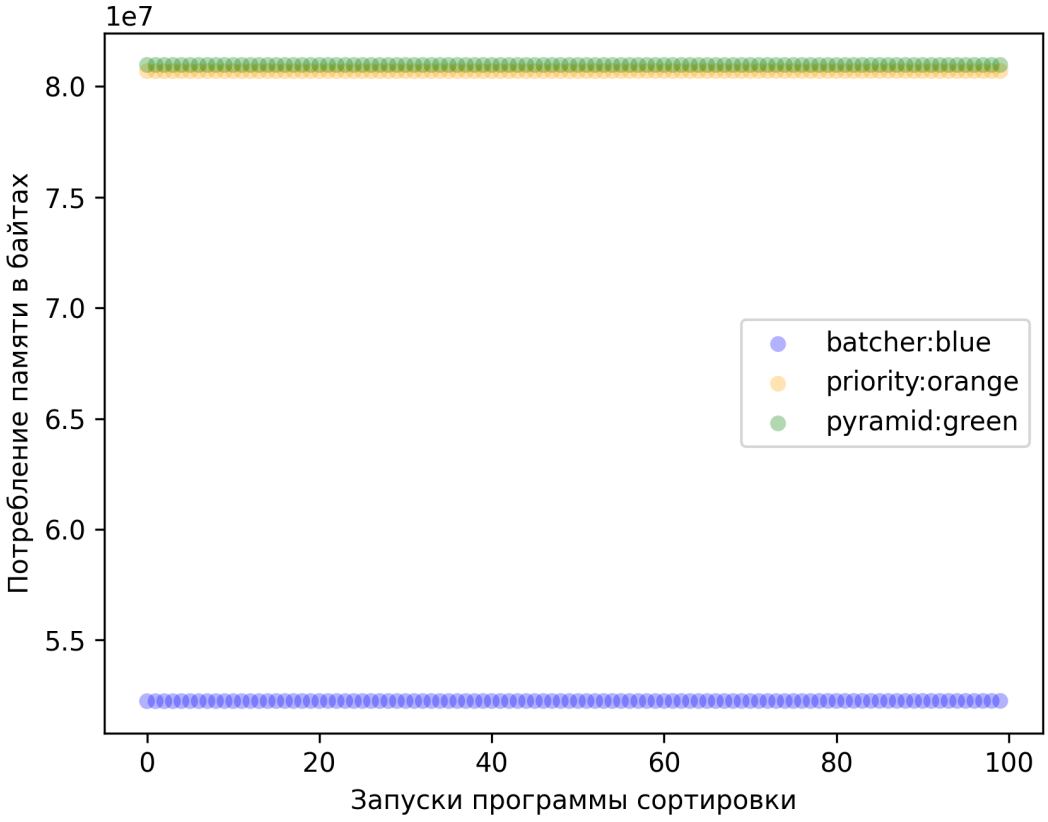
Picture. 15: График сортировки Бэтчера

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

```
# for randomizing
import random
# for fast arrays
from array import array

# for graphs and research_doc/plots
import numpy as np
from matplotlib import pyplot as plt

# importing sorts
from methods.batcher import bitonic_sortMem
from methods.priority_func import priority_sortMem
from methods.pyramid import heapSortMem

arr = list(range(4))
arr = array("i", arr)  # 'i' - signed int

ls = []

for i in range(10):
    random.shuffle(arr)
    ls.append(arr)
    ln_arr = len(arr)
    arr.extend(list(range(ln_arr, ln_arr * 2)))


colors = ["batcher:blue", "priority:orange", "pyramid:green"]
######################################## Batcher Sort
   ##############################################
exec_speeds_bitonic = [
    [2 ** (i + 2), bitonic_sortMem(ls[i], 0, len(ls[i]), 1)] for i in range
   (10)
]  # works


```

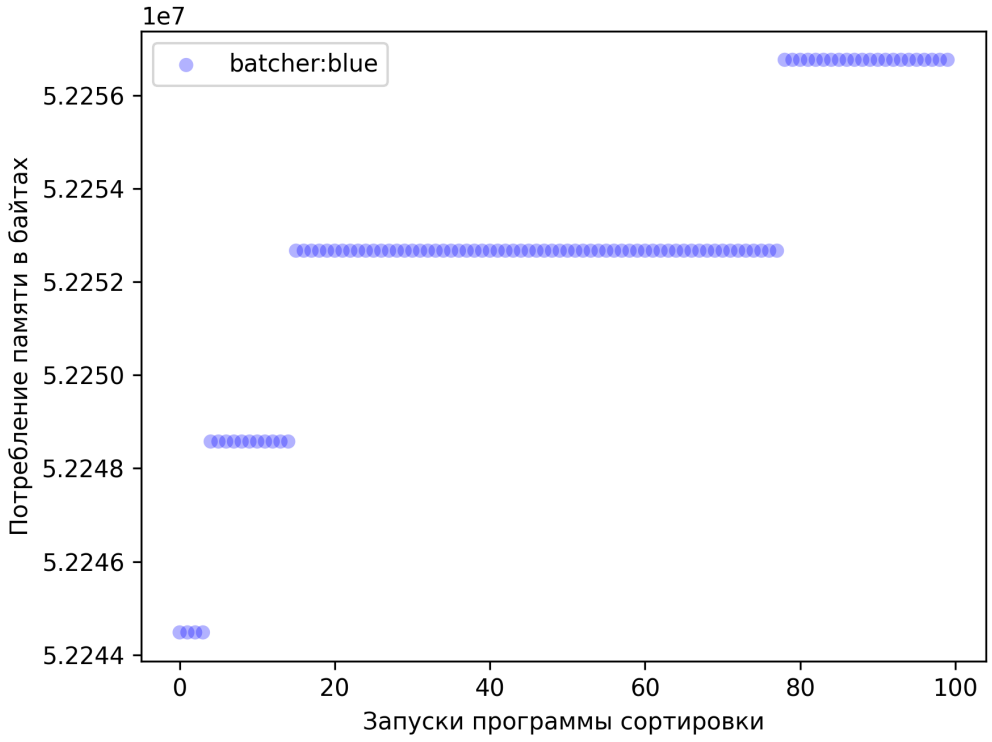
```
data = np.array(exec_speeds_bitonic)

x, y = data.T

plt.scatter(
    x,
    y,
    c=colors[0].split(":")[1],
    label=colors[0],
    alpha=0.3,
    edgecolors="none",
)
plt.ylabel("Скорость выполнения в секундах")
plt.xlabel("Потребление памяти")
plt.legend()
# plt.show()
plt.savefig(
    "research_doc/plots/batcher_memory_delta.png", dpi=400
)  # savefig, don't show
######################################## Batcher Sort
    ############################################

arr = list(range(4))
arr = array("i", arr)  # 'i' - signed int

ls = []

for i in range(10):
    random.shuffle(arr)
    ls.append(arr)
    ln_arr = len(arr)
    arr.extend(list(range(ln_arr, ln_arr * 2)))

######################################## Priority Sort
    ############################################
exec_speeds_priority = [
    [2 ** (i + 2), priority_sortMem(ls[i])] for i in range(10)
]  # for heapSort only

data = np.array(exec_speeds_priority)

x, y = data.T
```

```
plt.scatter(
    x,
    y,
    c=colors[1].split(":")[1],
    label=colors[1],
    alpha=0.3,
    edgecolors="none",
)
plt.ylabel("Скорость выполнения в секундах")
plt.xlabel("Потребление памяти")
plt.legend()
# plt.show()
plt.savefig(
    "research_doc/plots/priority_memory_delta.png", dpi=400
)  # savefig, don't show
######################################## Priority Sort
    ##############################################

arr = list(range(4))
arr = array("i", arr)  # 'i' - signed int

ls = []

for i in range(10):
    random.shuffle(arr)
    ls.append(arr)
    ln_arr = len(arr)
    arr.extend(list(range(ln_arr, ln_arr * 2)))

######################################## Pyramid Sort
    ##############################################
exec_speeds_heap = [[2 ** (i + 2), heapSortMem(ls[i])] for i in range(10)]
    # works

data = np.array(exec_speeds_heap)

x, y = data.T
plt.scatter(
    x,
    y,
    c=colors[2].split(":")[1],
    label=colors[2],
```

```
    alpha=0.3,
    edgecolors="none",
)
plt.ylabel("Скорость выполнения в секундах")
plt.xlabel("Потребление памяти")
plt.legend()
# plt.show()
plt.savefig(
    "research_doc/plots/bitonic_memory_delta.png", dpi=400
)  # savefig, don't show
######################################## Pyramid Sort
   ############################################
```

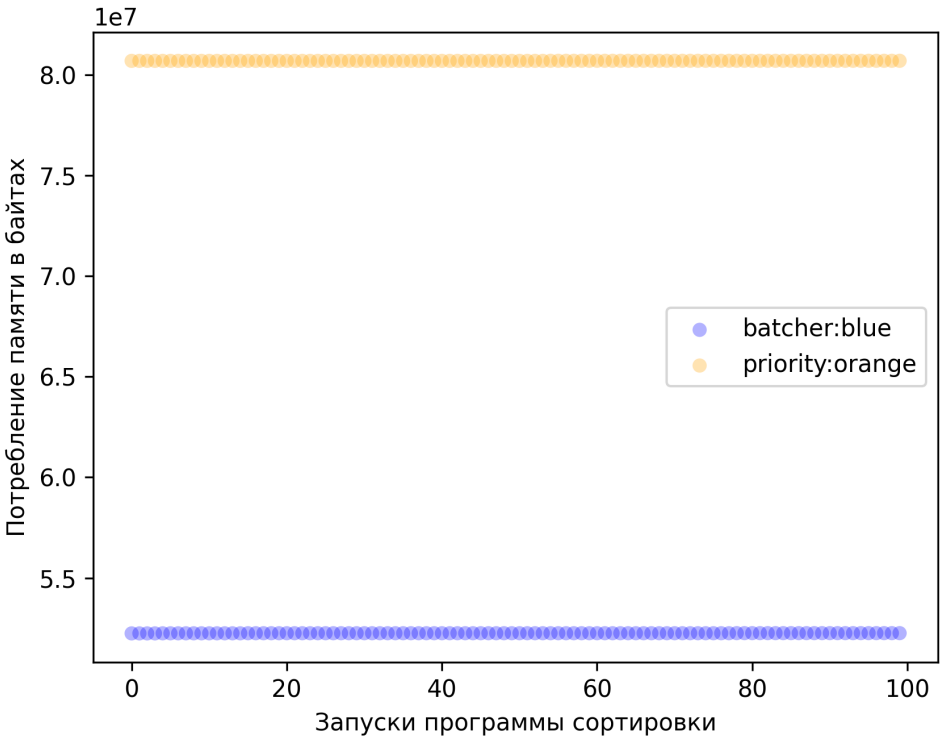
Таким образом были построены графики:

Потребление памяти сортировки Бэтчера



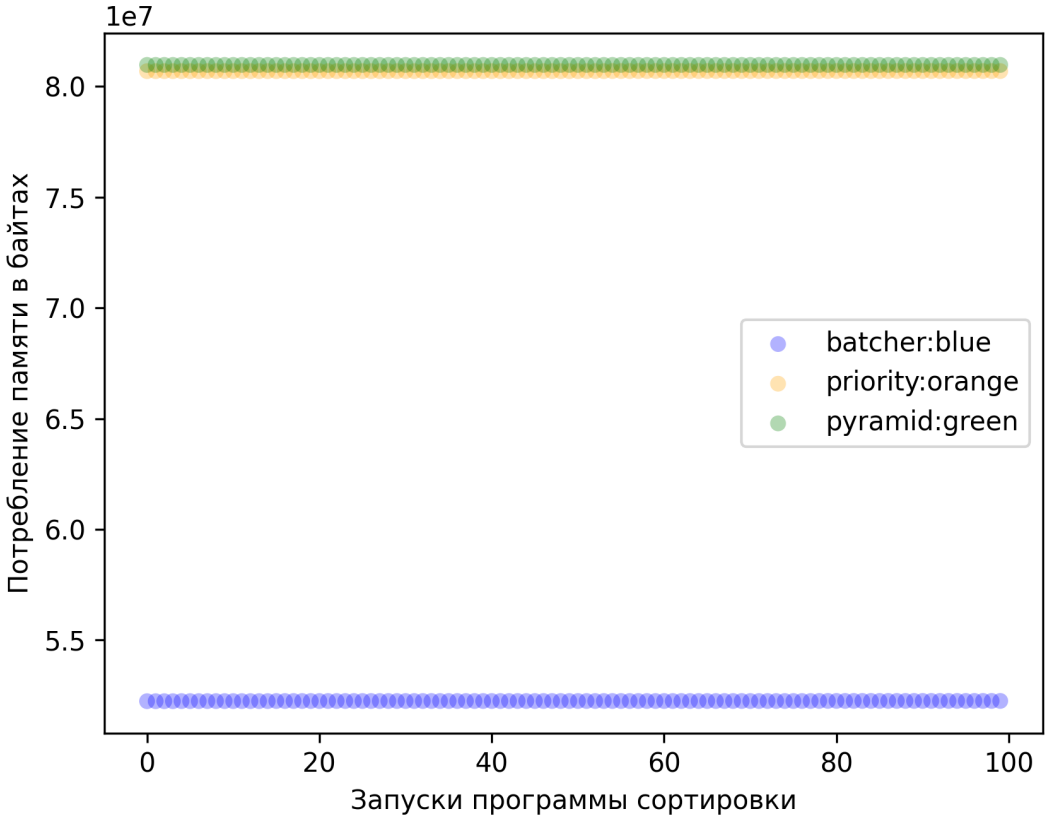
Picture. 18: График сортировки Бэтчера

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 придти к нескольким выводам, а именно:

1. При длине массива не более 10000 скорость сортивровки не особо меняется
2. Все три алгоритма носительно стабильны и получая на входе одинаковые массивы, они сорируют их с практически равной скоростью
3. Потребление памяти также особо не меняется, возможно из-за языка Python, который сам по себе потребляет достадочно 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 достадочное количество памяти, следует использовать Алгоритм сортировки Пирамидой
8. Если же у вас недостадочно 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