(САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №1
по курсу «Алгоритмы и структуры данных»
Тема: Сортировка вставками, выбором, пузырьковая
Вариант 12

Выполнил:
Колпаков А.С.
К3139

Проверил:
Афанасьев А.В

Санкт-Петербург
2024 г.

Содержание отчета

Задачи по варианту

Задача №1. Сортировка вставкой

Используя код процедуры Insertion-sort, напишите программу и проверьте сортировку массива $A = \{31, 41, 59, 26, 41, 58\}$.

- **Формат входного файла (input.txt).** В первой строке входного файла содержится число n ($1 \leq n \leq 10^3$) — число элементов в массиве. Во второй строке находятся n различных целых чисел, по модулю не превосходящих 10^9.
- **Формат выходного файла (output.txt).** Одна строка выходного файла с отсортированным массивом. Между любыми двумя числами должен стоять ровно один пробел.
- Ограничение по времени. 2сек.
- Ограничение по памяти. 256 мб.

Выберите любой набор данных, подходящих по формату, и протестируйте алгоритм.

Листинг кода:

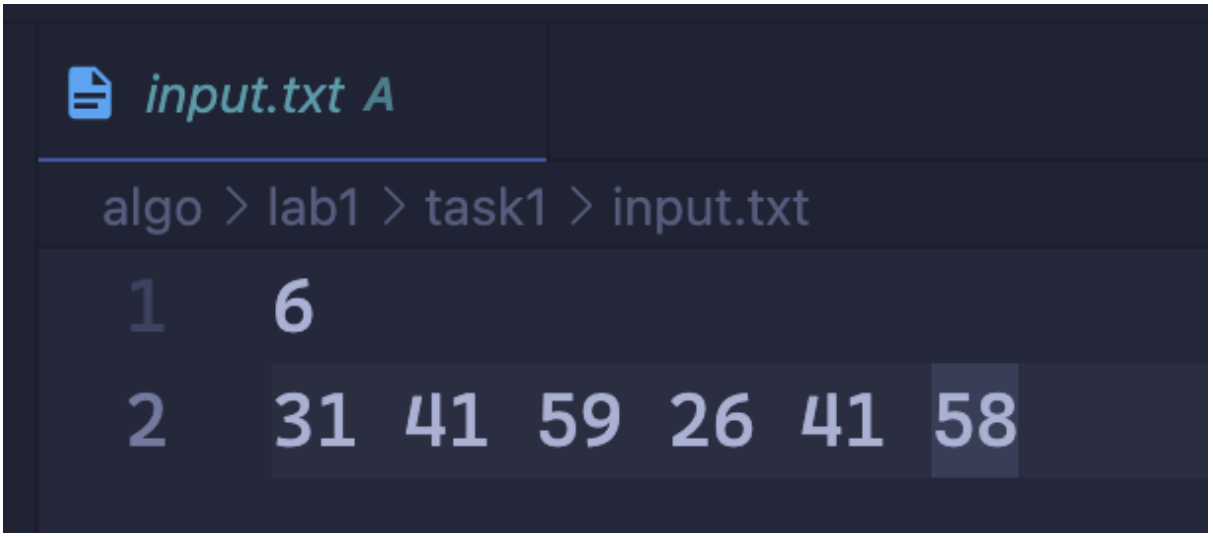
```
import time
import tracemalloc
tracemalloc.start()
t_start = time.perf_counter()

f = open('./input.txt')
n = int(f.readline())
arr = list(map(int, f.readline().split()))
f.close()

for i in range(1, n):
    key = arr[i]
    j = i - 1
    while j >= 0 and key < arr[j]:
```

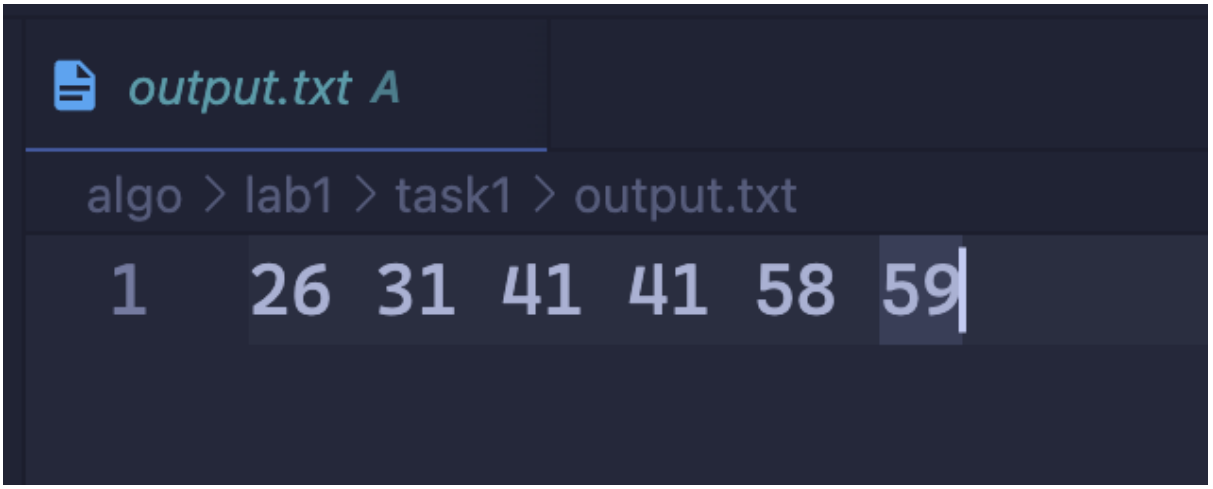
```
        arr[j + 1] = arr[j]
        j -= 1
    arr[j + 1] = key

f2 = open("output.txt", 'w')
f2.write((" ").join(list(map(str, arr))))
f2.close()

print('Время работы: %s секунд' %
(time.perf_counter() - t_start))
print("Память:", tracemalloc.get_traced_memory()[1]
/ (1024 ** 2), "МБ")
tracemalloc.stop()
```

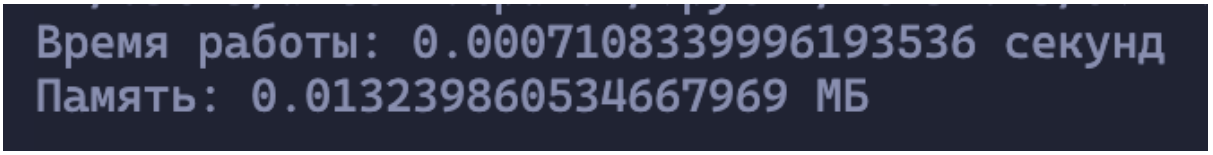
Текстовое объяснение решения:

1. Открываем файл и читаем оттуда массив
2. Мы запоминаем arr[i] в переменную key и с помощью цикла проверяем каждый элемент от i - 1 до 0, если этот элемент больше key, тогда записываем этот элемент по индексу j + 1.
3. После цикла while каждый раз записываем сам key в j + 1.

Результат работы кода на примерах из текста задачи:







	Время выполнения	Затраты памяти
Пример из задачи	0.0007108339996193536 секунд	0.013239860534667969 МБ

Вывод по задаче:
Программа успешно реализует алгоритм сортировки вставками. Тестирование показало правильную работу алгоритма. Несмотря на квадратичную временную сложность в худшем случае, сортировка вставками может быть эффективна для небольших массивов или уже частично отсортированных данных.

Задача №2. Сортировка вставкой +

Измените процедуру Insertion-sort для сортировки таким образом, чтобы в выходном файле отображалось в первой строке n чисел, которые обозначают новый индекс элемента массива после обработки.

- **Формат выходного файла (input.txt).**В первой строке выходного файла выведите n чисел. При этом i-ое число равно индексу, на который, в момент обработки его сортировкой вставками, был перемещен i-ый элемент исходного массива. Индексы нумеруются, начиная с единицы. Между любыми двумя числами должен стоять ровно один пробел.

Пример.

input.txt	output.txt
10 1 8 4 2 3 7 5 6 9 0	1 2 2 2 3 5 5 6 9 1 0 1 2 3 4 5 6 7 8 9

Листинг кода:

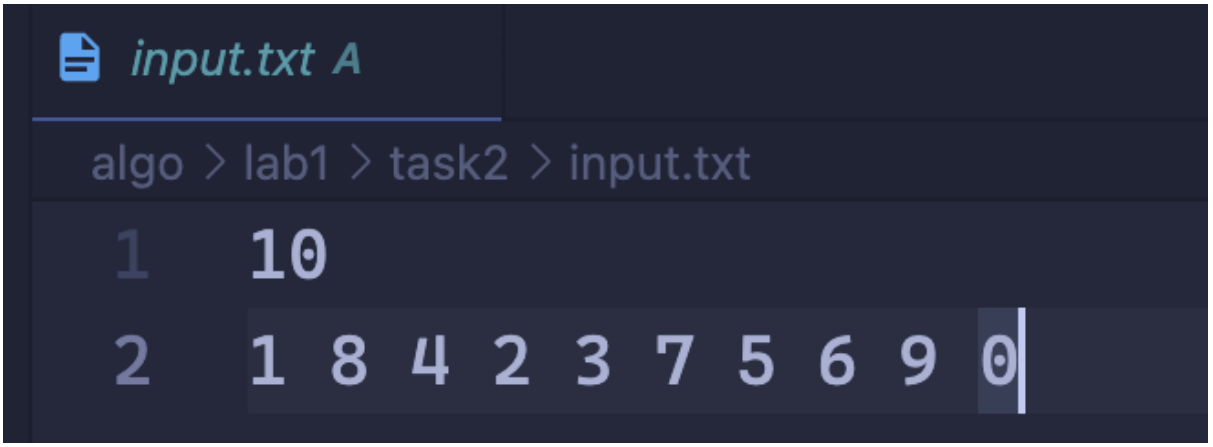
```
import time
import tracemalloc
tracemalloc.start()
t_start = time.perf_counter()


f = open('./input.txt')
n = int(f.readline())
arr = list(map(int, f.readline().split()))


indxs = []


for i in range(n):
    key = arr[i]
    j = i - 1
    while j >= 0 and key < arr[j]:
        arr[j + 1] = arr[j]
        j -= 1
    indxs.append(j+2)
```

```
    arr[j + 1] = key

f2 = open("output.txt", 'w')
f2.write((" ").join(list(map(str, indxs))) + "\n")
f2.write((" ").join(list(map(str, arr))))
f2.close()

print('Время работы: %s секунд' %
(time.perf_counter() - t_start))
print("Память:", tracemalloc.get_traced_memory()[1]
/ (1024 ** 2), "МБ")
tracemalloc.stop()
```

Текстовое объяснение решения:

1. Алгоритм аналогичен алгоритму из задачи 1, только теперь мы запоминаем индексы значений из изначального массива, записывая их как j+2.
2. Далее записываем в первой строке через join массив индексов, а следующей строкой записываем сами значения.

Результат работы кода на примерах из текста задачи:



	Время выполнения	Затраты памяти
Пример из задачи	0.0007853330025682226 секунд	0.01324462890625 МБ

Вывод по задаче:
Программа успешно реализует модифицированный алгоритм сортировки вставками, отслеживающий новое положение элементов в массиве.

Задача №7. Знакомство с жителями Сортлэнда

Владелец графства Сортлэнд, граф Бабблсортер, решил познакомиться со своими подданными. Число жителей в графстве нечетно и составляет n, где n может быть достаточно велико, поэтому граф решил ограничиться знакомством с тремя представителями народонаселения: с самым бедным жителем, с жителем, обладающим средним достатком, и с самым богатым жителем.

Согласно традициям Сортлэнда, считается, что житель обладает средним достатком, если при сортировке жителей по сумме денежных сбережений он оказывается ровно посередине. Известно, что каждый житель графства имеет уникальный идентификационный номер, значение которого расположено в границах от единицы до n. Информация о размере денежных накоплений жителей хранится в массиве M таким образом, что сумма денежных накоплений жителя, обладающего идентификационным номером i, содержится в ячейке $M[i]$. Помогите секретарю графа мистеру Свопу вычислить идентификационные номера жителей, которые будут приглашены на встречу с графом.

- **Формат входного файла (input.txt).** Первая строка входного файла содержит число жителей n ($3 \leq n \leq 9999$, n нечетно). Вторая строка содержит описание массива M, состоящее из положительных вещественных чисел, разделенных пробелами. Гарантируется, что все элементы массива M различны, а их значения имеют точность не более двух знаков после запятой и не превышают 10^6.
- **Формат выходного файла (output.txt).** В выходной файл выведите три целых положительных числа, разделенных пробелами — идентификационные номера беднейшего, среднего и самого богатого жителей Сортлэнда.

Листинг кода:

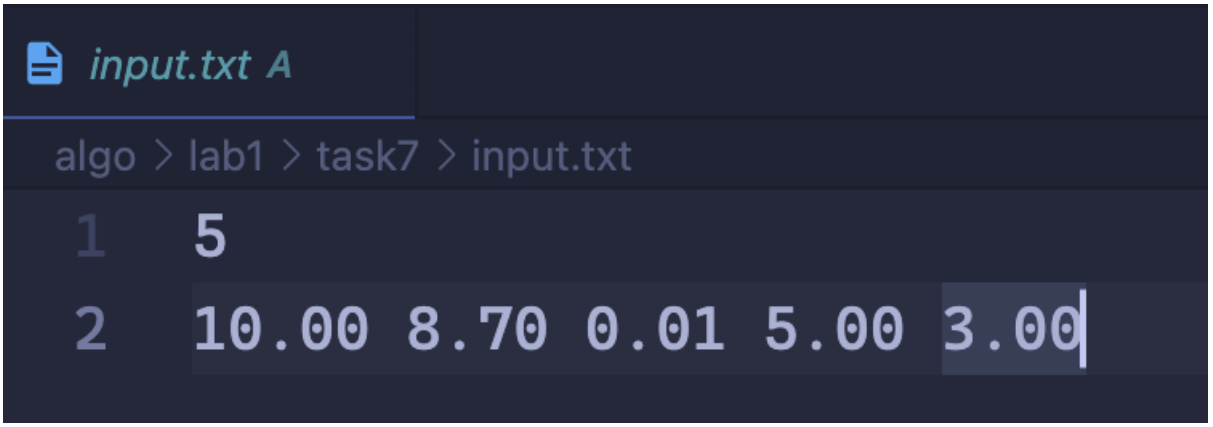
```
import time
import tracemalloc
tracemalloc.start()
t_start = time.perf_counter()


f = open('./input.txt')
n = int(f.readline())
M = list(map(float, f.readline().split()))


arr_ind = [[M[x], x + 1] for x in range(len(M))]
arr_ind.sort()
```

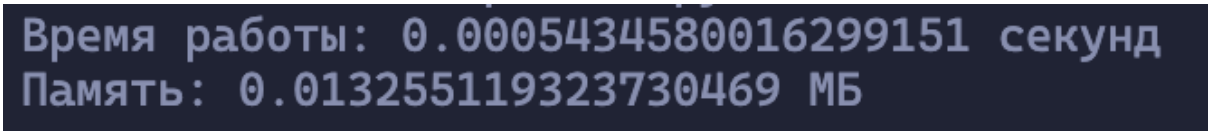
```
res = [arr_ind[0][1], arr_ind[len(M) // 2][1]
,arr_ind[-1][1]]

f2 = open("output.txt", 'w')
f2.write((" ").join(list(map(str, res))))
f2.close()

print('Время работы: %s секунд' %
(time.perf_counter() - t_start))
print("Память:", tracemalloc.get_traced_memory()[1]
/ (1024 ** 2), "МБ")
tracemalloc.stop()
```

Текстовое объяснение решения:

1. Считываем значения из файла input.txt в массив M.
2. Далее создаем двумерный массив arr_ind, в который записываем каждое значение из массива M с его индексом.
3. Далее сортируем arr_ind по первым значениям и записываем самого бедного жителя, со средним доходом и самого богатого в массив res.
4. Записываем значения из массива res в output.txt.

Результат работы кода на примерах из текста задачи:





	Время выполнения	Затраты памяти
Пример из задачи	0.0005434580016299151 секунд	0.013255119323730469 МБ

Вывод по задаче:
Программа эффективно определяет идентификационные номера всех трех жителей, которых граф пригласит на встречу, используя алгоритмы поиска минимума и максимума, а также алгоритм сортировки.

Дополнительные задачи

Задача №3. Сортировка вставкой по убыванию

Перепишите процедуру Insertion-sort для сортировки в невозрастающем порядке вместо неубывающего с использованием процедуры Swap.

Формат входного и выходного файла и ограничения - как в задаче 1.

Подумайте, можно ли переписать алгоритм сортировки вставкой с использованием рекурсии?

Листинг кода:

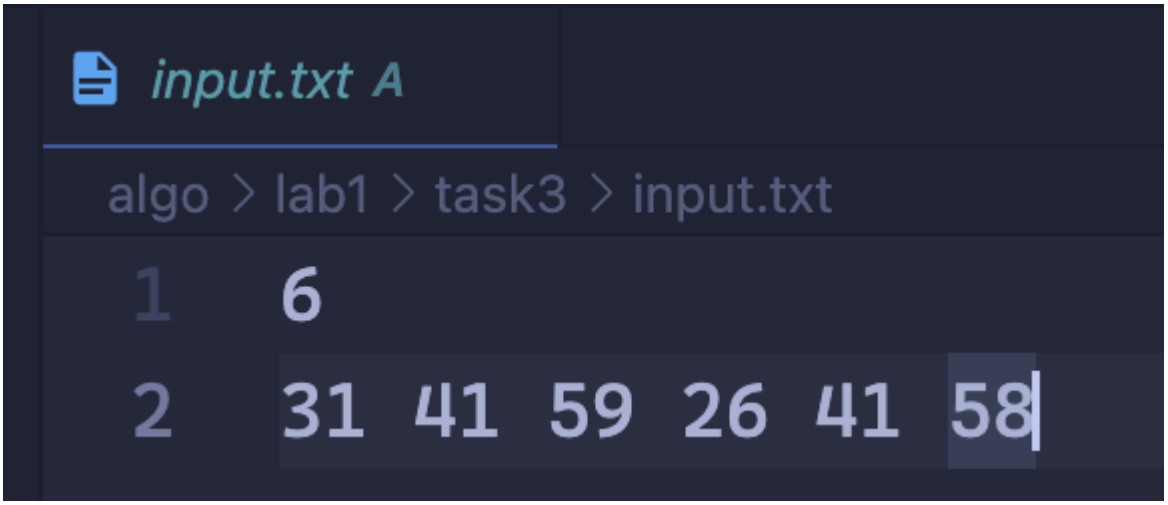
```
import time
import tracemalloc
tracemalloc.start()
t_start = time.perf_counter()

f = open('./input.txt')
n = int(f.readline())
arr = list(map(int, f.readline().split()))

def swap(i, j, ar):
   ar[i], ar[j] = ar[j], ar[i]

for i in range(1, n):
   key = arr[i]
   j = i - 1
   while j >= 0 and key > arr[j]:
       swap(j, j+1, arr)
       j -= 1

f2 = open("output.txt", 'w')
f2.write((" ").join(list(map(str, arr))))
```

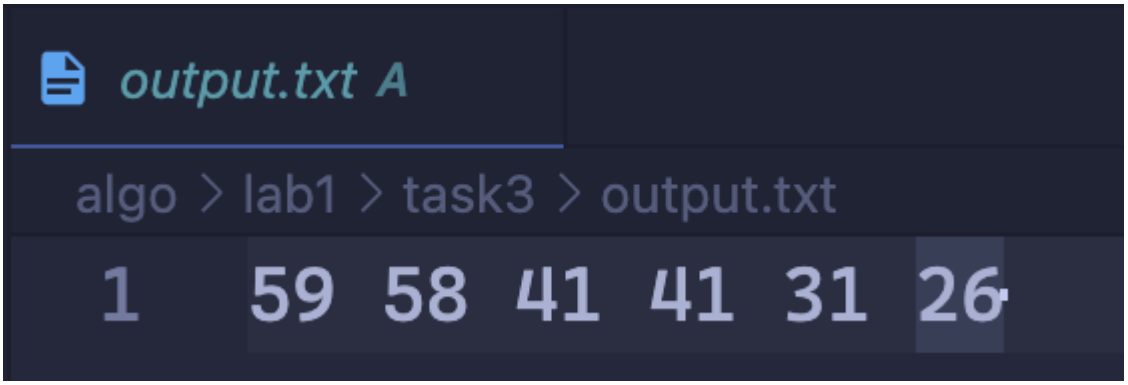
```
f2.close()

print('Время работы: %s секунд' %
(time.perf_counter() - t_start))
print("Память:", tracemalloc.get_traced_memory()[1]
/ (1024 ** 2), "МБ")
tracemalloc.stop()
```

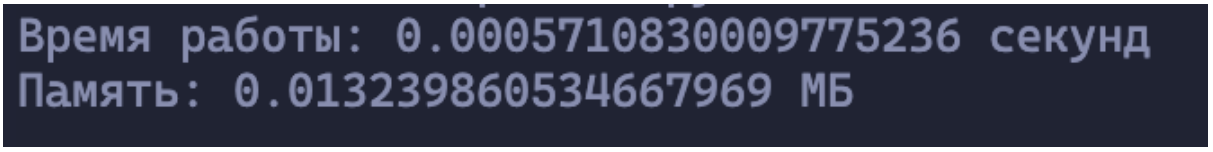
Текстовое объяснение решения:

1. Алгоритм аналогичен алгоритму из задания 2, только теперь условие в цикле while таково, что key должен быть больше arr[j].
2. Также добавил функцию swap, которая просто заменяет элементы массива по их индексам.

Результат работы кода на примерах из текста задачи:







	Время выполнения	Затраты памяти
Пример из задачи	0.0005710830009775236 секунд	0.013239860534667969 МБ

Вывод по задаче:
Модифицированная рекурсивная процедура `Insertion-sort` эффективно сортирует массив в убывающем порядке. Изменения в процедуре сравнения и применение процедуры `Swap` позволяют получить желаемый результат.

Задача №6. Пузырьковая сортировка

Пузырьковая сортировка представляет собой популярный, но не очень эффективный алгоритм сортировки. В его основе лежит многократная перестановка соседних элементов, нарушающих порядок сортировки. Вот псевдокод этой сортировки:

```
Bubble_Sort(A):
  for i = 1 to A.length - 1
    for j = A.length downto i+1
      if A[j] < A[j-1]
        поменять A[j] и A[j-1] местами
```

Напишите код на Python и докажите корректность пузырьковой сортировки. Для доказательства корректоности процедуры вам необходимо доказать, что она завершается и что $A'[1] \leq A'[2] \leq ... \leq A'[n]$, где A' - выход процедуры Bubble_Sort, а n - длина массива A.

Определите время пузырьковой сортировки в наихудшем случае и в среднем случае и сравните его со временем сортировки вставкой.

Формат входного и выходного файла и ограничения - как в задаче 1.

Листинг кода:

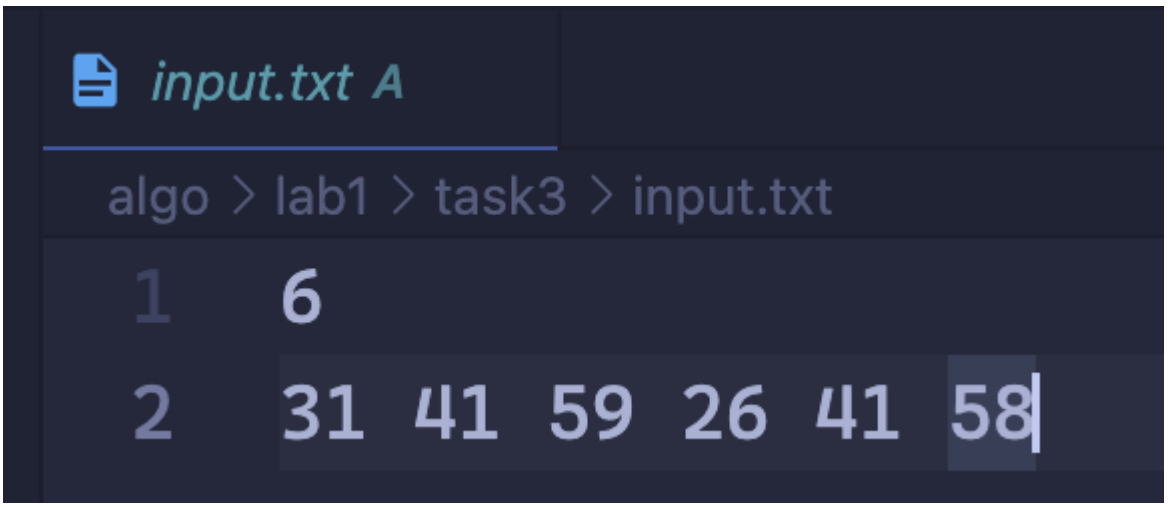
```python
import time
import tracemalloc
tracemalloc.start()
t_start = time.perf_counter()


f = open('./input.txt')
n = int(f.readline())
arr = list(map(int, f.readline().split()))


for i in range(n):
 for j in range(0, n-i-1):
   if arr[j] > arr[j+1]:
     arr[j], arr[j+1] = arr[j+1], arr[j]


f2 = open("output.txt", 'w')
f2.write((" ").join(list(map(str, arr))))
```

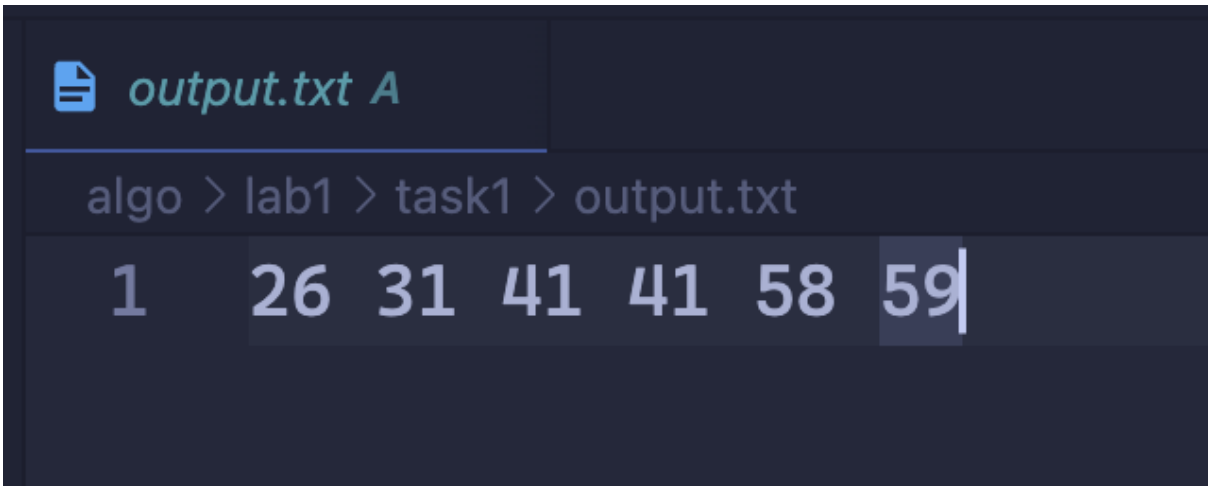
```
f2.close()


print('Время работы: %s секунд' %
(time.perf_counter() - t_start))
print("Память:", tracemalloc.get_traced_memory()[1]
/ (1024 ** 2), "МБ")
tracemalloc.stop()
```

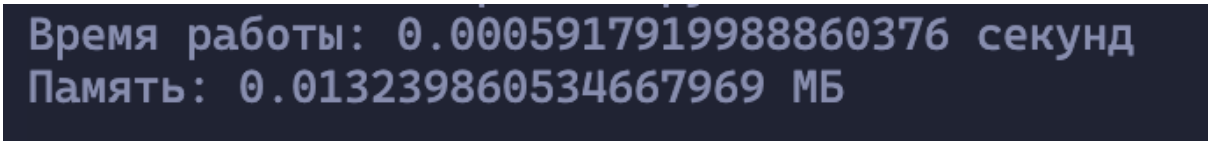
Текстовое объяснение решения:

1. Считываем данные из файла в массив.
2. Далее с помощью вложенного цикла проходимся по всем значениям массива. причем во вложенном цикле проходимся до n-i-1
3. Если текущее число больше следующего, то меняем их местами, таким образом, большие значения будут постепенно передвигаться к краю списка.

Результат работы кода на примерах из текста задачи:







	Время выполнения	Затраты памяти
Пример из задачи	0.0005917919988860376 секунд	0.013239860534667969 МБ

Вывод по задаче:
Алгоритм пузырьковой сортировки корректен: он успешно завершается и на выходе дает отсортированный массив значений, который и требуется получить. При худшем и среднем случаях - скорость равна O(n^2), как и в алгоритме сортировки вставкой.

Задача №10. Палиндром

Палиндром - это строка, которая читается одинаково как справа налево, так и слева направо.

На вход программы поступает набор больших латинских букв (не обязательно различных). Разрешается переставлять буквы, а также удалять некоторые буквы. Требуется из данных букв по указанным правилам составить палиндром наибольшей длины, а если таких палиндромов несколько, то выбрать первый из них в алфавитном порядке.

- **Формат входного файла (input.txt).** В первой строке входных данных содержится число n ($1 \leq n \leq 100000$). Во второй строке задается последовательность из n больших латинских букв (буквы записаны без пробелов).
- **Формат выходного файла (output.txt).** В единственной строке выходных данных выдайте искомый палиндром.
- Пример:

Листинг кода:

```
import time
import tracemalloc
tracemalloc.start()
t_start = time.perf_counter()


f = open('./input.txt')
n = int(f.readline())
chars = f.readline()


chars_count = {}
for c in chars:
 if c not in chars_count:
   chars_count[c] = 1
 else:
   chars_count[c] += 1
```

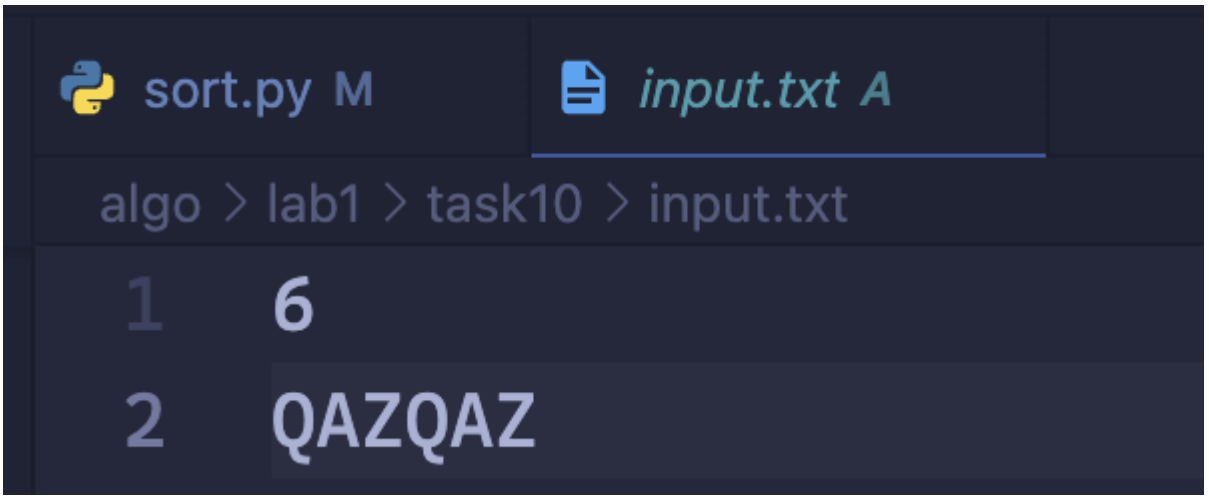
```
chars_count_sort = sorted(chars_count.items(),
key=lambda x: (-x[1], x[0]), reverse=True)

res = ""
for key, value in chars_count_sort:
 if value % 2 != 0 and res == "":
   res += key * value
 elif value % 2 == 0:
   res = key * (value // 2) + res + key * (value //
2)
 f2 = open("output.txt", 'w')
f2.write(res)
f2.close()

print('Время работы: %s секунд' %
(time.perf_counter() - t_start))
print("Память:", tracemalloc.get_traced_memory()[1]
/ (1024 ** 2), "МБ")
tracemalloc.stop()
```

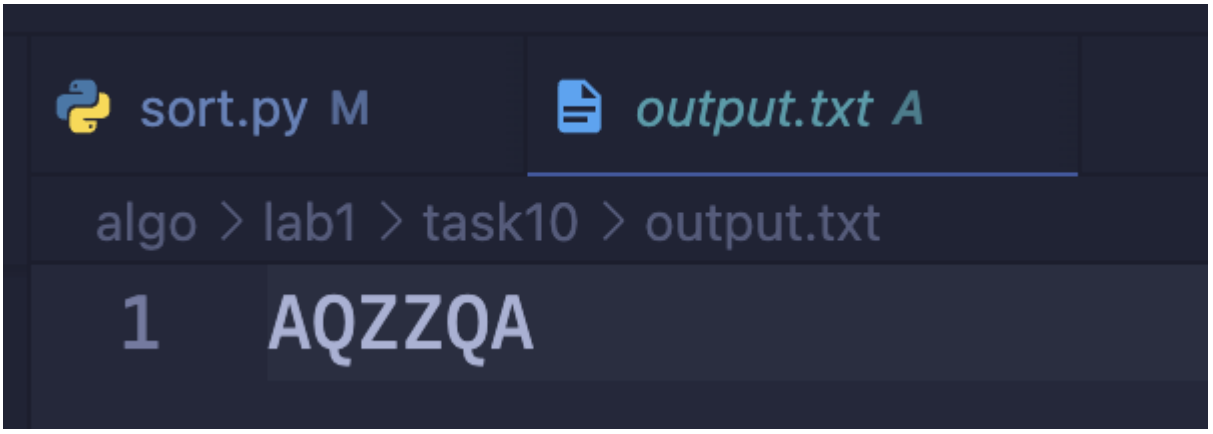
Текстовое объяснение решения:

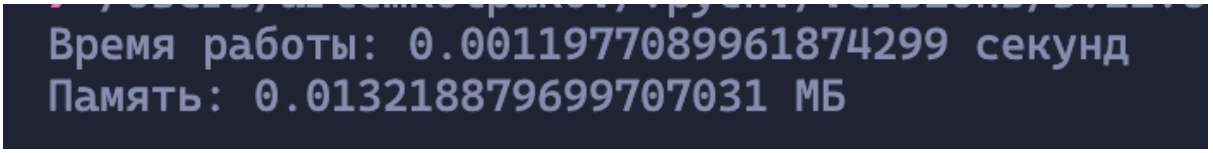
1. Считываем строку из файла и задаем словарь.
2. Далее с помощью цикла считаем количество каждого символа в строке.
3. Потом сортируем по количеству и алфавиту значения словаря.
4. После чего, если количество символов в строке не делится на 2 и при этом строка пустая, то добавляем в нее это значение, умноженное на его количество в строке.

5. Если же количество символа в строке делится на 2, то с каждой стороны строки добавляем по одинаковому количеству этого символа, чтобы в итоге получился палиндром.

Результат работы кода на примерах из текста задачи:







	Время выполнения	Затраты памяти
Пример из задачи	0.00119770899618742 99 секунд	0.013218879699707031 МБ

Вывод по задаче:
Данная программа позволяет эффективно находить наибольший палиндром, который можно составить из заданного набора букв. Он

использует сортировку для оптимизации процесса построения палиндрома и обеспечивает выбор наибольшего палиндрома в алфавитном порядке.

Вывод

Рассмотренные в ходе лабораторной работы алгоритмы: Вставкой, Пузырьковый и Выбором представляют собой довольно простые и часто используемые, а также легко реализуемые алгоритмы, которые, в свою очередь, имеют и недостатки. Например, неэффективность (временная сложность этих алгоритмов оценивается O(n^2)).
Таким образом, хоть данные алгоритмы и являются довольно простыми в реализации, но для больших объемов данных подходят не лучшим образом, поэтому для реальных задач лучше воспользоваться более продвинутыми алгоритмами сортировки.