Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

(Университет ИТМО)

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №2

по курсу «Алгоритмы и структуры данных»

Тема: СОРТИРОВКА ВСТАВКАМИ, ВЫБОРОМ, ПУЗЫРЬКОМ

Вариант 1

СОДЕРЖАНИЕ

[ЗАДАЧИ ПО ВАРИАНТУ 3](#_Toc179208245)

[1 Сортировка вставкой 3](#_Toc179208246)

[2 Сортировка вставкой c запоминанием индексов 5](#_Toc179208247)

[3 Сортировка вставкой по убыванию 6](#_Toc179208248)

[4 Линейный поиск 7](#_Toc179208249)

[5 Сортировка выбором 8](#_Toc179208250)

[1 Сортировка выбором 10](#_Toc179208251)

# ЗАДАЧИ ПО ВАРИАНТУ

## Сортировка вставкой

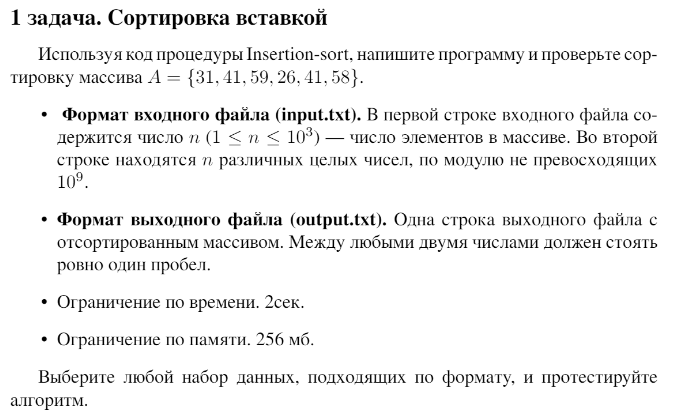


Рисунок 1 – Задача 1

InsertationSort.py

**def** insertion\_sort(lst: list) -> None:

**for** i **in** range(1, len(lst)):

cur = i

**while** cur > 0 **and** lst[cur - 1] > lst[cur]:

lst[cur - 1], lst[cur] = lst[cur], lst[cur - 1]

cur -= 1

Main.py

**from** InsertionSort **import** insertion\_sort

**def** main():

**with** open('../txtf/input.txt') **as** file:

lst = list(map(int, file.readline().split()))

insertion\_sort(lst)

**with** open('../txtf/output.txt', 'w') **as** file:

**print**(\*lst, file=file)

**if** \_\_name\_\_ == "\_\_main\_\_":

main()

Предоставленный код состоит из двух файлов: InsertionSort.py и Main.py. Файл InsertionSort.py содержит функцию insertion\_sort, которая реализует алгоритм сортировки вставками для сортировки списка целых чисел по возрастанию. Файл Main.py импортирует функцию insertion\_sort и использует ее для сортировки списка целых чисел, прочитанного из файла input.txt, и записывает отсортированный список в файл output.txt.

**Анализ кода**

InsertionSort.py

Функция insertion\_sort принимает список целых чисел в качестве входных данных и сортирует его на месте с помощью алгоритма сортировки вставками. Функция имеет сложность времени O(n^2) и сложность пространства O(1), поскольку она использует только несколько дополнительных переменных для выполнения сортировки.

Main.py

Функция main читает список целых чисел из файла input.txt, сортирует его с помощью функции insertion\_sort и записывает отсортированный список в файл output.txt. Функция использует оператор with для обеспечения правильного закрытия файлов после чтения и записи.

**Преимущества**

1. Простая реализация

Алгоритм сортировки вставками прост в реализации и понимании.

1. Эффективность для небольших списков

Сортировка вставками имеет низкий накладной расход в плане дополнительной памяти, необходимой для выполнения сортировки, что делает ее эффективной для небольших списков.

1. Стабильная сортировка

Сортировка вставками является стабильным алгоритмом сортировки, что означает, что порядок равных элементов сохраняется.

**Недостатки**

1. Медленная работа для больших списков

Сортировка вставками имеет сложность времени O(n^2), что делает ее медленной для больших списков.

1. Неподходящая для параллельной обработки

Сортировка вставками является последовательным алгоритмом, что означает, что она не может быть параллелизирована для использования нескольких процессоров.

**Вывод**

Код реализует алгоритм сортировки вставками для сортировки списка целых чисел. Хотя алгоритм прост в реализации и эффективен для небольших списков, он медлен для больших списков и не подходит для параллельной обработки.

## Сортировка вставкой c запоминанием индексов

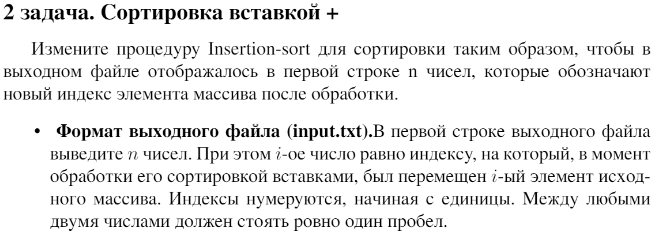


Рисунок 2 – Задача 2

InsertationSort.py

**def** insertion\_sort\_indexes(lst: list) -> list:

output\_lst = [1]

**for** i **in** range(1, len(lst)):

cur = i

**while** cur > 0 **and** lst[cur - 1] > lst[cur]:

lst[cur - 1], lst[cur] = lst[cur], lst[cur - 1]

cur -= 1

output\_lst.append(cur + 1)

**return** output\_lst

Main.py ***аналогичен коду к задаче 1***

Код практически идентичен предыдущему, за исключением запоминания индексов во время выполнения программы.

## Сортировка вставкой по убыванию

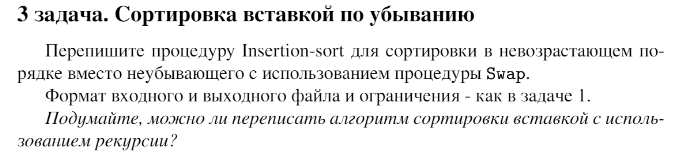


Рисунок 3 – Задача 3

InsertationSort.py

**def** insertion\_sort\_indexes(lst: list) -> list:

output\_lst = [1]

**for** i **in** range(1, len(lst)):

cur = i

**while** cur > 0 **and** lst[cur - 1] < lst[cur]:

lst[cur - 1], lst[cur] = lst[cur], lst[cur - 1]

cur -= 1

output\_lst.append(cur + 1)

**return** output\_lst

Main.py ***аналогичен коду к задаче 1***

Код практически идентичен предыдущему, единственное, знак > при сравнении символов меняется на знак <.

## Линейный поиск

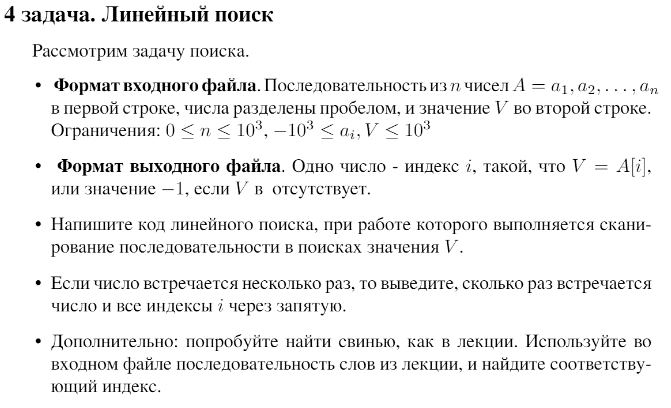


Рисунок 4 – Задача 4

Find.py

**def** find(lst: list, value) -> list:

indexes = []

**for** i **in** range(len(lst)):

**if** lst[i] == value:

indexes.append(i + 1)

**if** **not** indexes:

indexes.append(-1)

**return** indexes

Main.py

**from** Find **import** find

**def** main():

**with** open('../txtf/input.txt') **as** file:

lst = list(map(int, file.readline().split()))

value = int(file.readline())

**with** open('../txtf/output.txt', 'w') **as** file:

**print**(\*find(lst, value), file=file)

**if** \_\_name\_\_ == "\_\_main\_\_":

main()

**Анализ кода**

Find.py

Содержит функцию find, которая проходит по всему списку, ищет совпадение со значением value и записывает индексы найденных элементов в список indexes. Если значение найдено, возвращается список с индексами (индексация начинается с 1). Если значение не найдено, возвращается список с элементом -1.

Main.py

Читает список чисел и значение для поиска из файла input.txt. Вызывает функцию find для поиска всех вхождений значения в список. Результаты записываются в файл output.txt.

**Вывод**

Линейный, последовательный поиск — алгоритм нахождения заданного значения произвольной функции на некотором отрезке. Данный алгоритм является простейшим алгоритмом поиска

## Сортировка выбором

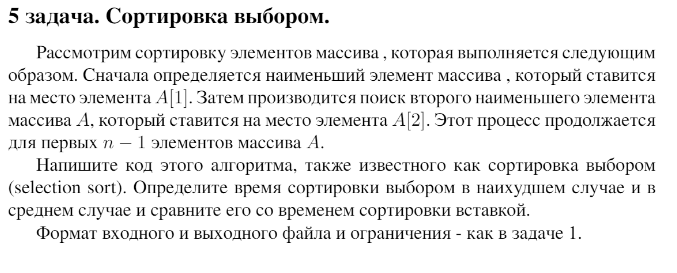


Рисунок 5 – Задача 5

SelectionSort.py

**def** selection\_sort(lst: list) -> None:

**for** start **in** range(len(lst) - 1):

mn = [lst[start], start]

**for** i **in** range(start + 1, len(lst)):

**if** lst[i] < mn[0]:

\*mn, = lst[i], i

lst[start], lst[mn[1]] = lst[mn[1]], lst[start]

Main.py

**from** SelectionSort **import** selection\_sort

**def** main():

**with** open('../txtf/input.txt') **as** file:

lst = list(map(int, file.readline().split()))

selection\_sort(lst)

**with** open('../txtf/output.txt', 'w') **as** file:

**print**(\*lst, file=file)

**if** \_\_name\_\_ == "\_\_main\_\_":

main()

**Анализ кода**

SelectionSort.py

Функция selection\_sort реализует алгоритм **сортировки выбором**. Она проходит по списку и на каждом шаге выбирает минимальный элемент среди оставшихся неотсортированных и меняет его местами с текущей позицией.

1. Внешний цикл идет по элементам списка, начиная с первой позиции.
2. На каждом шаге внешний цикл ищет минимальный элемент в оставшейся части списка:
   * Переменная mn хранит текущий минимальный элемент и его индекс.
   * Внутренний цикл проходит по оставшейся части списка, начиная со следующего элемента.
   * Если найден элемент меньше текущего минимального, обновляются значения в mn.
3. После завершения внутреннего цикла, минимальный элемент меняется местами с элементом на позиции start.

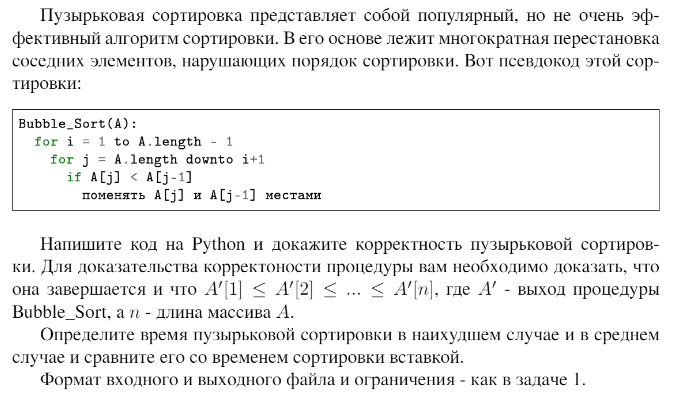
Main.py

Функция main() читает список чисел из файла input.txt и передает его в функцию selection\_sort(). После сортировки результат записывается в файл output.txt.

**Вывод**

Код программы является корректной реализацией сортировки выбором. Он прост и понятен для понимания, однако его производительность – О(n^2) – оставляет желать лучшего. Для больших списков лучше использовать более эффективные алгоритмы.

## Сортировка пузырьком



BubbleSort.py

**def** bubble\_sort(lst: list) -> None:

**for** left **in** range(len(lst) - 1):

flag = False

**for** i **in** range(len(lst) - 1, left, -1):

**if** lst[i - 1] > lst[i]:

lst[i - 1], lst[i] = lst[i], lst[i - 1]

flag = True

**if** **not** flag: **break**

Main.py

**from** BubbleSort **import** bubble\_sort

**def** main():

**with** open('../txtf/input.txt') **as** file:

lst = list(map(int, file.readline().split()))

bubble\_sort(lst)

**with** open('../txtf/output.txt', 'w') **as** file:

**print**(\*lst, file=file)

**if** \_\_name\_\_ == "\_\_main\_\_":

main()

**Анализ кода**

BubbleSort.py

Функция bubble\_sort реализует алгоритм **сортировки пузырьком**. Этот алгоритм проходит по списку несколько раз, и на каждом шаге сравнивает соседние элементы, меняя их местами, если они расположены не в правильном порядке. Процесс повторяется до тех пор, пока список не станет отсортированным.

1. Внешний цикл (for left in range(len(lst) - 1)):

* Проходит по каждому элементу списка, начиная с первого и заканчивая предпоследним. Этот цикл отвечает за число полных проходов по списку.

1. Инициализация флага (flag = False):

* Флаг flag отслеживает, происходили ли обмены элементов за текущую итерацию. Если за итерацию обменов не произошло, это означает, что массив уже отсортирован, и можно завершить выполнение алгоритма.

1. Внутренний цикл (for i in range(len(lst) - 1, left, -1)):

* Проходит по элементам от конца списка к началу, сравнивая и меняя местами соседние элементы, если они находятся в неправильном порядке.
* Если текущий элемент больше следующего (lst[i - 1] > lst[i]), они меняются местами, а флаг flag устанавливается в True, указывая на то, что произошёл хотя бы один обмен.

1. Проверка на завершение:

* Если после одной полной итерации внутреннего цикла флаг остаётся False, то массив отсортирован, и выполнение алгоритма завершается досрочно через команду break.

Main.py

1. Программа считывает список целых чисел из файла input.txt, преобразует их в список, и передает этот список в функцию bubble\_sort.
2. После выполнения сортировки результат записывается в файл output.txt.

**Корректность сортировки пузырьком**

Алгоритм **сортировки пузырьком** состоит из последовательного перемещения наибольшего элемента на своё место на каждой итерации. Рассмотрим доказательство корректности по двум критериям: **терминирование** (завершение алгоритма) и **получение отсортированного массива на выходе.**

1. Терминирование

Алгоритм сортировки пузырьком работает, начиная с последнего элемента и продвигаясь в обратном порядке. При каждой итерации количество элементов, среди которых проверяется порядок, уменьшается на 1. Алгоритм выполняет фиксированное число итераций, равное n−1, где n — длина списка. Так как каждый шаг уменьшает область проверки, и внутренний цикл гарантированно завершается, процедура завершается после выполнения n−1 проходов.

Таким образом, алгоритм **гарантированно завершится.**

1. Массив на выходе отсортирован

Для доказательства, что алгоритм сортирует массив, можно воспользоваться **индукцией**:

* **База индукции**: после первой итерации наибольший элемент "всплывает" в конец массива, занимая свою правильную позицию.
* **Переходный шаг:** на каждой следующей итерации, когда наибольший элемент на отрезке массива находится на правильной позиции, алгоритм "всплывает" следующий наибольший элемент на своё место.

После выполнения всех итераций на каждом шаге наибольший элемент на соответствующем подотрезке оказывается на своём месте. Таким образом, после выполнения всех проходов массив будет полностью отсортирован.

**Сравнение с сортировкой вставкой**

**Наихудший случай**:

* + **Сортировка пузырьком**: Время работы — **O(n²)**, так как каждый элемент может быть перемещён через весь массив. При полностью отсортированном в обратном порядке массиве каждый элемент будет перемещаться на своё место на каждой итерации.
  + **Сортировка вставкой**: Время работы — **O(n²)**, так как каждый элемент вставляется в уже отсортированную часть массива. В случае отсортированного в обратном порядке массива каждый элемент придётся перемещать на своё место через всю длину отсортированной части.

**Наилучший случай**:

* + **Сортировка пузырьком**: Время работы — **O(n)**, если массив уже отсортирован. В этом случае можно добавить флаг, который завершит алгоритм, если на очередной итерации не произошло перестановок.
  + **Сортировка вставкой**: Время работы — **O(n)**, так как если массив отсортирован, каждый элемент будет просто проверяться и оставаться на своём месте.

**Вывод**

Сортировка пузырьком — это простой алгоритм, но не самый эффективный. Время работы в наихудшем случае квадратичное, что делает его нежелательным для сортировки больших массивов. Однако он может быть эффективен для уже почти отсортированных массивов с добавлением флага завершения.

Сортировка вставкой обычно предпочтительнее из-за более естественной инкрементальной стратегии сортировки, особенно на небольших наборах данных.