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 4](#_Toc179456014)

[1 Сортировка вставкой 4](#_Toc179456015)

[Задание 4](#_Toc179456016)

[Код 4](#_Toc179456017)

[Анализ кода 5](#_Toc179456018)

[Вывод 5](#_Toc179456019)

[2 Сортировка вставкой c запоминанием индексов 6](#_Toc179456020)

[Задание 6](#_Toc179456021)

[Код 6](#_Toc179456022)

[Анализ кода 7](#_Toc179456023)

[3 Сортировка вставкой по убыванию 8](#_Toc179456024)

[Задание 8](#_Toc179456025)

[Код 8](#_Toc179456026)

[Анализ кода 8](#_Toc179456027)

[Вывод 9](#_Toc179456028)

[4 Линейный поиск 10](#_Toc179456029)

[Задание 10](#_Toc179456030)

[Код 10](#_Toc179456031)

[Анализ кода 11](#_Toc179456032)

[Вывод 11](#_Toc179456033)

[5 Сортировка выбором 11](#_Toc179456034)

[Задание 11](#_Toc179456035)

[Код 11](#_Toc179456036)

[Анализ кода 11](#_Toc179456037)

[Вывод 12](#_Toc179456038)

[6 Сортировка пузырьком 13](#_Toc179456039)

[Задание 13](#_Toc179456040)

[Код 13](#_Toc179456041)

[Анализ кода 13](#_Toc179456042)

[Вывод 15](#_Toc179456043)

[7 Сортировка пузырьком с индексами 15](#_Toc179456044)

[Задание 15](#_Toc179456045)

[Код 15](#_Toc179456046)

[Анализ кода 16](#_Toc179456047)

[8 Сортировка выбором + 17](#_Toc179456048)

[Задание 17](#_Toc179456049)

[Код 17](#_Toc179456050)

[Анализ кода 18](#_Toc179456051)

[9 Сложение бинарных чисел 19](#_Toc179456052)

[Задание 19](#_Toc179456053)

[Код 19](#_Toc179456054)

[Анализ кода 20](#_Toc179456055)

[Вывод 20](#_Toc179456056)

[10 Построение палиндромов 21](#_Toc179456057)

[Задание 21](#_Toc179456058)

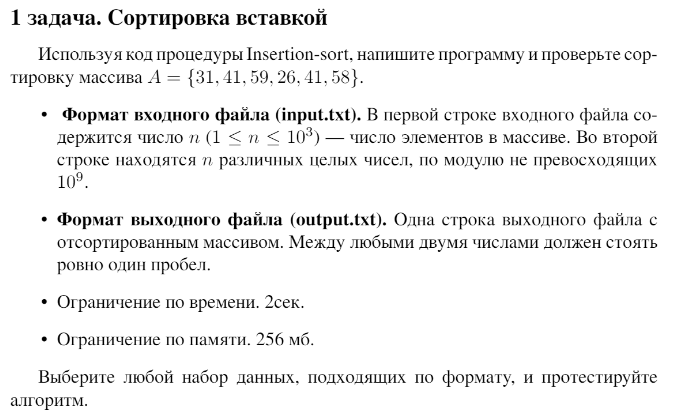
[Код 21](#_Toc179456059)

[Анализ кода 22](#_Toc179456060)

# ЗАДАЧИ ПО ВАРИАНТУ

## Сортировка вставкой

### Задание



### Код

*InsertationSort.py*

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

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

key = lst[i]

left = 0

right = i

**while** left < right:

mid = left + (right - left) // 2

**if** key < lst[mid]:

right = mid

**else**:

left = mid + 1

**for** j **in** range(i, right, -1):

lst[j] = lst[j - 1]

lst[right] = key

**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()

### Анализ кода

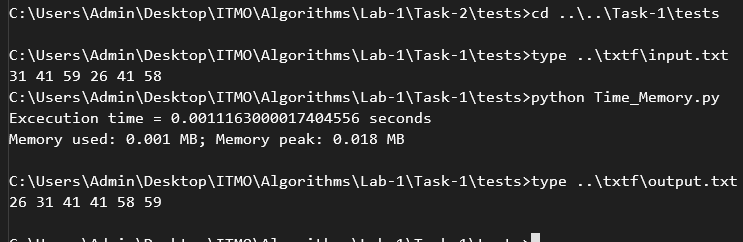
Код представляет собой две реализации алгоритма сортировки вставкой: *insertion\_sort\_bin\_pow* и *insertion\_sort*. Функции принимает список *lst* в качестве входного параметра и сортирует его на месте.

Первая функция итерирует по списку от начала до конца, для каждого элемента ищет его позицию в отсортированной части списка с помощью бинарного поиска, затем сдвигает элементы в отсортированной части списка, чтобы освободить место для вставки элемента.

Вторая функция итерирует по списку от начала до конца, для каждого элемента сравнивает его с предыдущим элементом, и меняет их местами, если предыдущий элемент больше. Так происходит до тех пор, пока предыдущий элемент не окажется меньше.

*insertion\_sort* в лучшем случае, когда список практически отсортирован, будет работать за O(n), а в худшем случае – за O(n^2).

*insertion\_sort\_bin\_pow* независимо от исходных данных выполняет бинарный поиск за O(log\_n) и сдвигает элементы, и время работы всегда будет около O(n\*(log\_n + n)), что чуть быстрее, чем сортировка при помощи предыдущей реализации, но время работы все равно O(n^2).

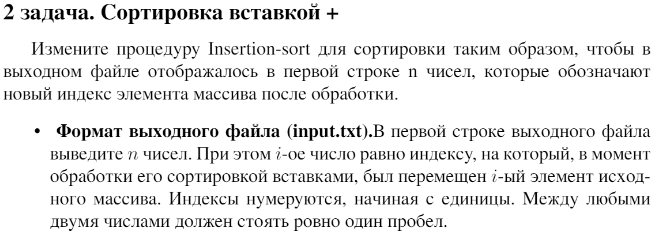


### Вывод

В целом, *insertion\_sort\_bin\_pow* является более эффективной для больших списков, а *insertion\_sort* - для маленьких или почти отсортированных списков.

## Сортировка вставкой c запоминанием индексов

### Задание



### Код

*InsertationSort.py*

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

output\_lst = [1]

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

key = lst[i]

left = 0

right = i

**while** left < right:

mid = left + (right - left) // 2

**if** key < lst[mid]:

right = mid

**else**:

left = mid + 1

**for** j **in** range(i, right, -1):

lst[j] = lst[j - 1]

lst[right] = key

output\_lst.append(right + 1)

**return** output\_lst

**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*

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

**def** main():

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

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

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

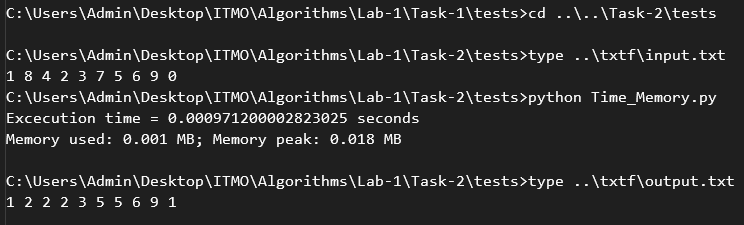
**print**(\*insertion\_sort\_indexes(lst), file=file)

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

main()

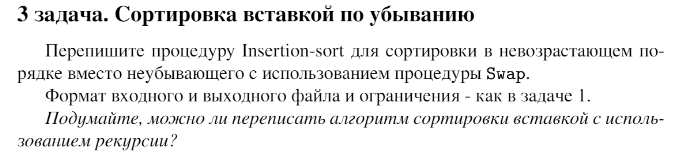
### Анализ кода

Код аналогичен коду из [задачи 1](#_Анализ_кода_1), различие состоит в том, что параллельно сортировке, выполняется запись индексов в список, который потом возвращается функцией.



## Сортировка вставкой по убыванию

### Задание



### Код

*InsertationSort.py*

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

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

key = lst[i]

left = 0

right = i

**while** left < right:

mid = left + (right - left) // 2

**if** key > lst[mid]:

right = mid

**else**:

left = mid + 1

**for** j **in** range(i, right, -1):

lst[j] = lst[j - 1]

lst[right] = key

**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\_reversed

**def** main():

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

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

insertion\_sort\_reversed(lst)

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

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

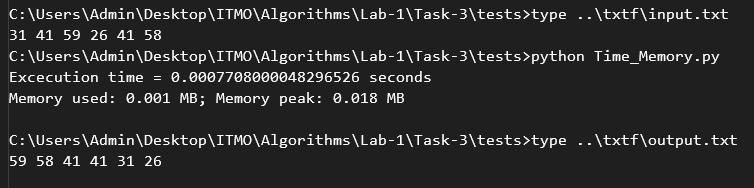
**if** \_\_name\_\_ == "\_\_main\_\_":

main()

### Анализ кода

Код аналогичен коду из [задачи 1](#_Анализ_кода_1), различие состоит в том, что в первой реализации бинарный поиск идет по отсортированному в обратном порядке массиву, поэтому поменяно условие *if key < lst[mid]:* на *if key > lst[mid]:*

Во второй реализации мы меняем элементы не тогда, когда предыдущий больше текущего, а наоборот.

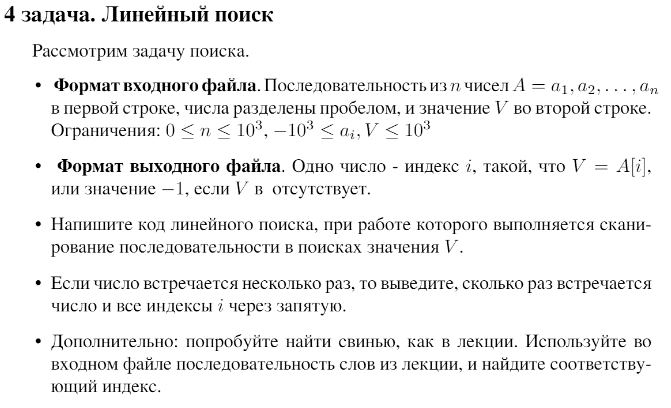


### Вывод

Чтобы сделать сортировку в обратном порядке, достаточно поменять в нужном месте знак > на знак <. Очень удобно.

## Линейный поиск

### Задание



### Код

*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:

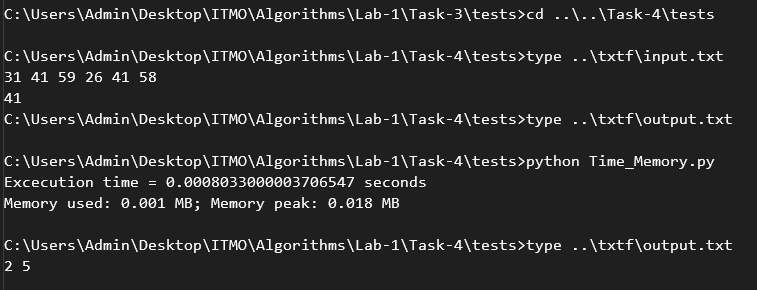
**print**(\*find(lst, value), file=file)

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

main()

### Анализ кода

Идем по списку, если элемент списка совпадает с искомым значением, запоминаем его индекс. Если список с индексами окажется в итоге пустым, добавим в него -1, чтобы было понятно, что ничего не нашлось.

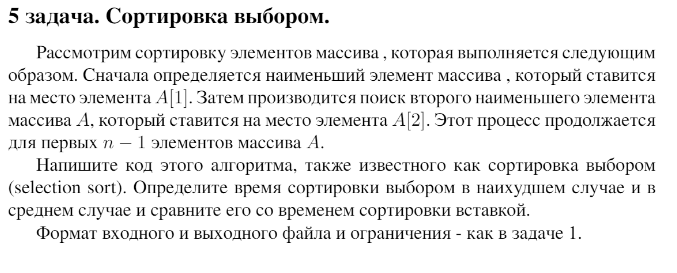


### Вывод

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

## Сортировка выбором

### Задание



### Код

*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\_\_":

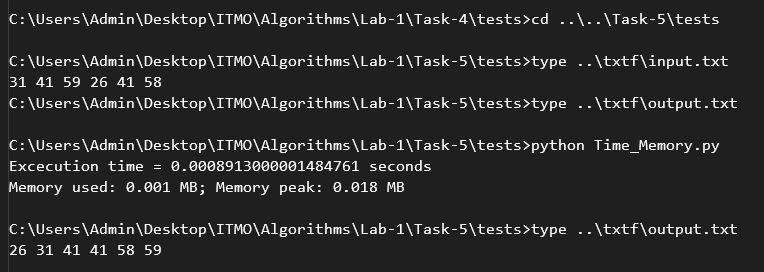
main()

### Анализ кода

Функция *selection\_sort* принимает список *lst* в качестве входного параметра и сортирует его на месте.

Функция использует два цикла для итерации по списку. Внешний цикл итерирует по списку от начала до конца, а внутренний цикл итерирует от текущей позиции внешнего цикла до конца списка в поисках минимального элемента в неотсортированной части. Найденный элемент меняется с элементов в текущей позиции, т.е. становится последним в отсортированной части.

Время работы такой сортировки в любом случае будет O(n), тес не менее, можно оптимизировать поиск минимального элемента, например, храня исходные данные в куче (это позволяет искать минимум за O(log\_n), т.е. общее время работы алгоритма улучшится до O(n log\_n))

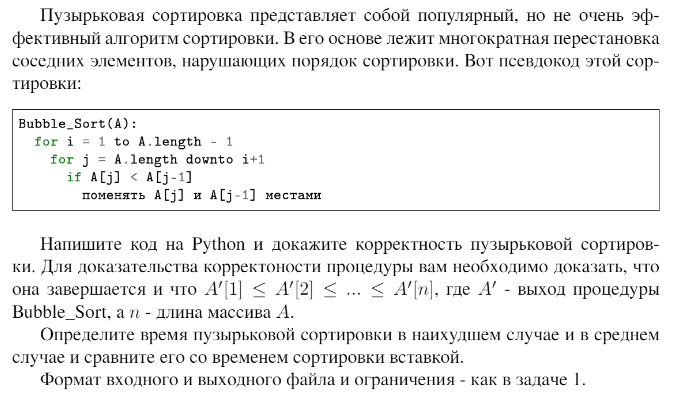


### Вывод

Сортировка вставкой пока выглядит более презентабельно, т.к. она хотя бы для почти отсортированных списков хорошо работает. К тому же сортировку выбором сложнее оптимизировать и есть более легкие в написании алгоритмы, время выполнения которых лучше оптимизированной сортировки выбором.

## Сортировка пузырьком

### Задание



### Код

*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()

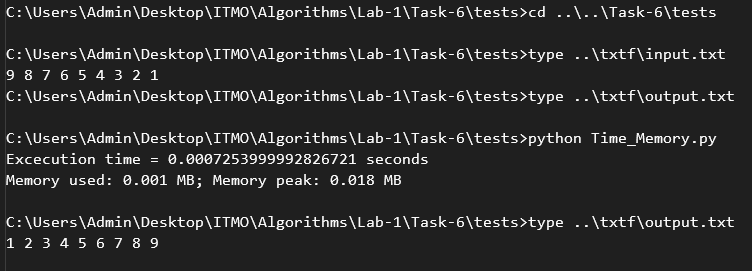
### Анализ кода

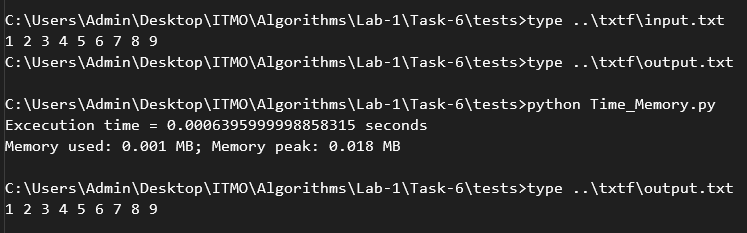
Функция *bubble\_sort* принимает список *lst* в качестве входного параметра и сортирует его на месте.

Внешний цикл итерирует по списку от начала до конца. Во внутреннем цикле, начиная с конца списка и заканчивая первой парой после уже отсортированной части, сравнивается каждый элемент с его соседом справа и, если элемент больше, чем его сосед, обменивает их местами. Таким образом, на каждой итерации внешнего цикла неотсортированная часть списка изменяется таким образом, что минимальный элемент этой части оказывается первым.

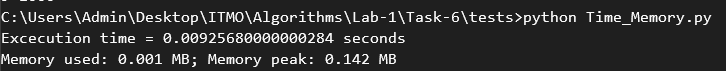
Функция оптимизирована с помощью флага: если при выполнении внутреннего цикла не поменялись никакие два соседних элемента, значит, неотсортированная часть списка уже отсортирована и можно закончить внешний цикл.

Благодаря оптимизации в лучшем случае алгоритм отработает за O(n), но в худшем время работы остается O(n^2). В этом плане эта сортировка похожа на сортировку вставкой.



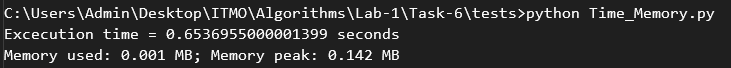


Запишем в файл input.txt последовательность от 0 до 1000 и посмотрим время выполнения программы  

А теперь запишем туда последовательность от 1000 до 0



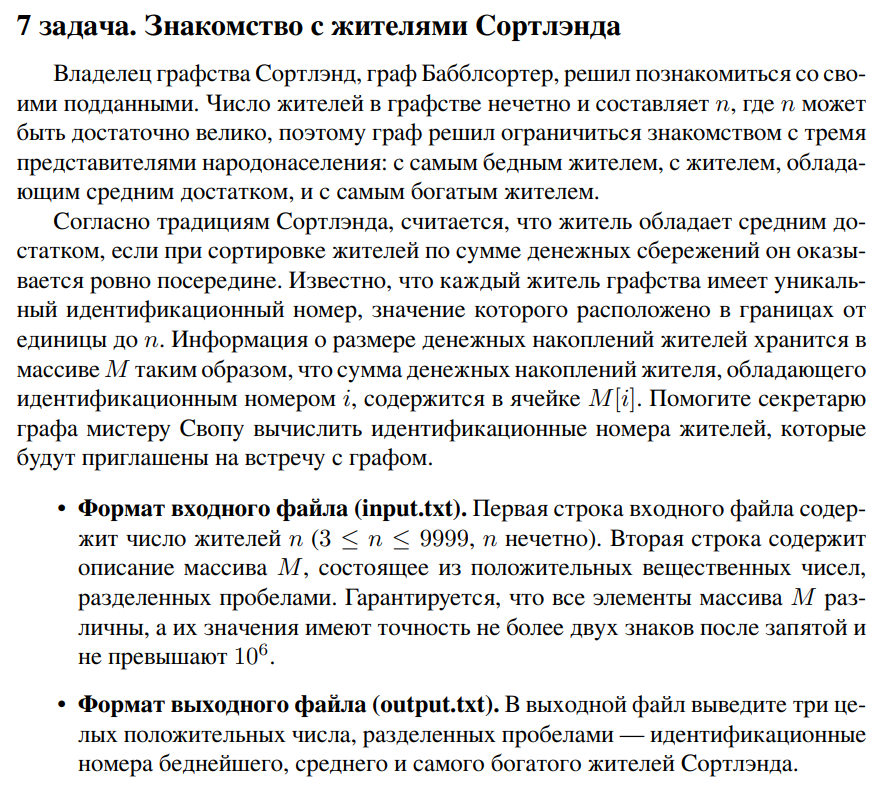


Время в наилучшем и наихудшем случае сильно отличается.

### Вывод

## Сортировка пузырьком с индексами

### Задание



### Код

*BubbleSort.py*

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

indexes = list(range(1, len(lst) + 1))

**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]

indexes[i - 1], indexes[i] = indexes[i], indexes[i - 1]

flag = True

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

**return** indexes

*main.py*

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

**from** decimal **import** Decimal

**def** main():

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

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

**if** len(lst) % 2 == 0:

**raise** ValueError("The list must contain an odd number of elements")

indexes = bubble\_sort\_indexes(lst)

**print**(\*lst)

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

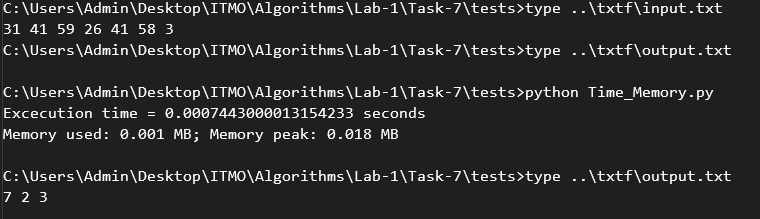
**print**(indexes[0], indexes[len(indexes) // 2], indexes[-1], sep=" ", file=file)

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

main()

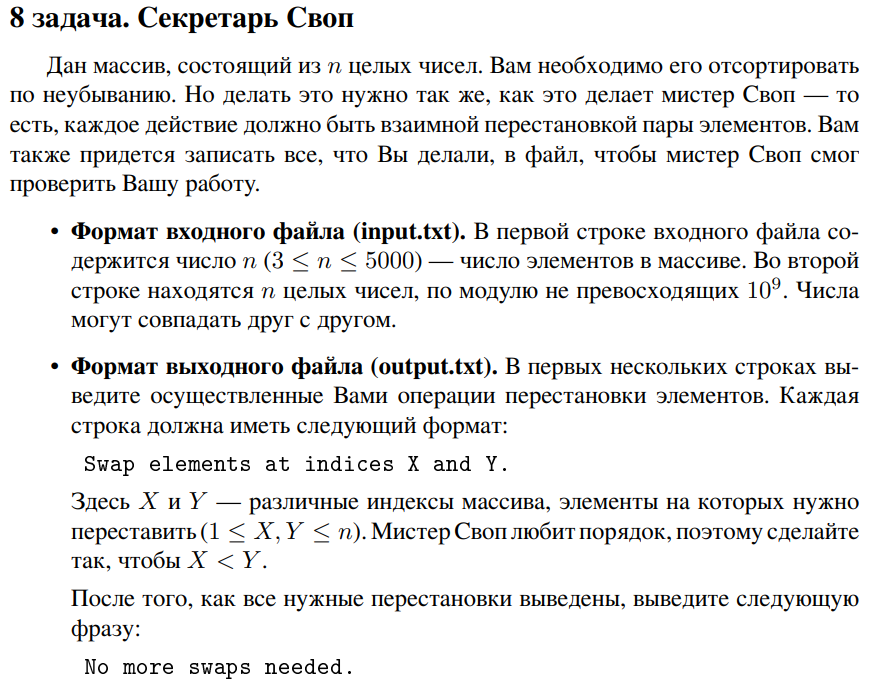
### Анализ кода

Код аналогичен коду из [задачи 6](#_Анализ_кода), различие состоит в том, что параллельно сортировке, выполняется запись индексов в список, который потом возвращается функцией.



## Сортировка выбором +

### Задание



### Код

*SelectionSort.py*

**def** selection\_sort(lst: list):

**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

**if** start != mn[1]:

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

**yield** f"Swap elements at indices {start + 1} and {mn[1] + 1}."

**yield** "No more swaps needed."

*main.py*

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

**def** main():

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

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

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

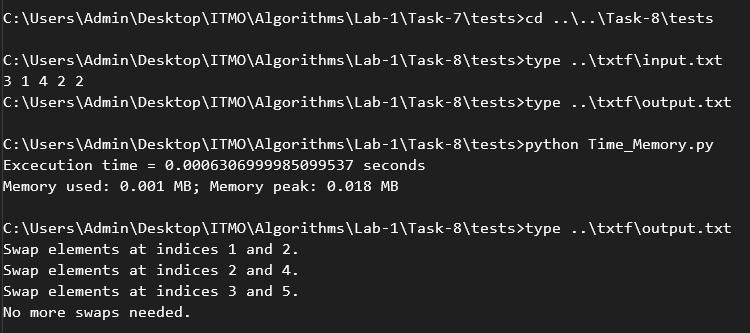
**print**(\*selection\_sort(lst), sep="\n", file=file)

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

main()

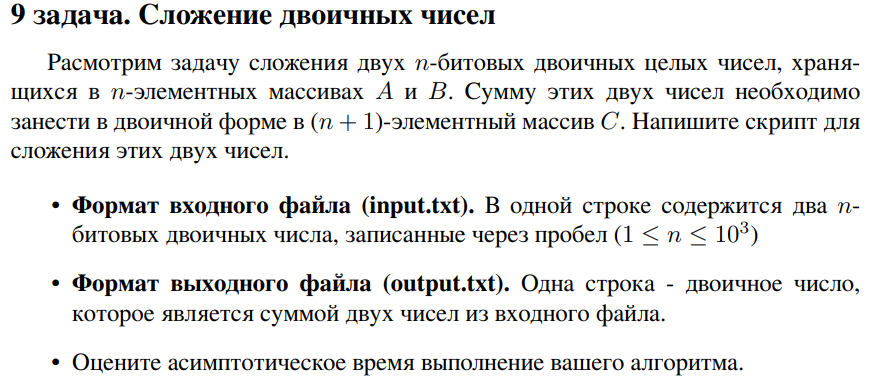
### Анализ кода

Код аналогичен коду из [задачи 5](#_Анализ_кода_2), различие состоит в том, что параллельно сортировке, выполняется генерация сообщений с информацией о том, на какой позиции был найден минимум. Также генерируется сообщение о том, что сортировка завершена.



## Сложение бинарных чисел

### Задание



### Код

*BinAdd.py*

**def** bin\_add(first: list, second: list) -> list:

**if** len(first) < len(second):

first = [0] \* (len(second) - len(first)) + first

**elif** len(first) > len(second):

second = [0] \* (len(first) - len(second)) + second

n = len(first)

result = [0] \* (n + 1)

**for** i **in** range(n - 1, -1, -1):

**if** **not**(first[i] == 0 **or** first[i] == 1) **or** **not**(second[i] == 0

**or** second[i] == 1):

**raise** ValueError("Incorrect number")

result[i + 1] += first[i] + second[i]

**if** result[i + 1] > 1:

result[i] += 1

result[i + 1] -= 2

**return** result

*main.py*

**from** BinAdd **import** bin\_add

**def** main():

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

first, second = map(list, file.readline().split())

first = list(map(int, first))

second = list(map(int, second))

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

**print**(\*bin\_add(first, second), sep="", file=file)

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

main()

### Анализ кода

Функция *bin\_add* принимает два списка *first* и *second* в качестве входных параметров и возвращает результат сложения в виде списка.

Cначала сравниваются длины входных списков и дополняет короче список нулями до длины более длинного списка.

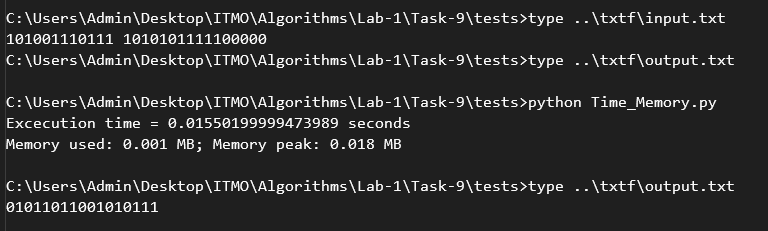
Затем создается список *result* длиной на 1 больше длины входных списков для хранения результата.

Функция также проверяет, что элементы входных списков являются бинарными (0 или 1), и если это не так, генерирует исключение *ValueError*.

Функция итерирует по спискам в обратном порядке, начиная с конца, и для каждого элемента суммирует соответствующие элементы из обоих списков.

Если сумма элементов превышает 1, функция добавляет 1 к предыдущему элементу результата и вычитает 2 из текущего элемента результата. Достаточно просто вычитать 2, т.к. результат сложения не превышает 3 (максимум 2 после сложения соотв. разрядов, максимум 1 от переноса единицы после сложения предыдущих разрядов).

Алгоритм работает за O(n).

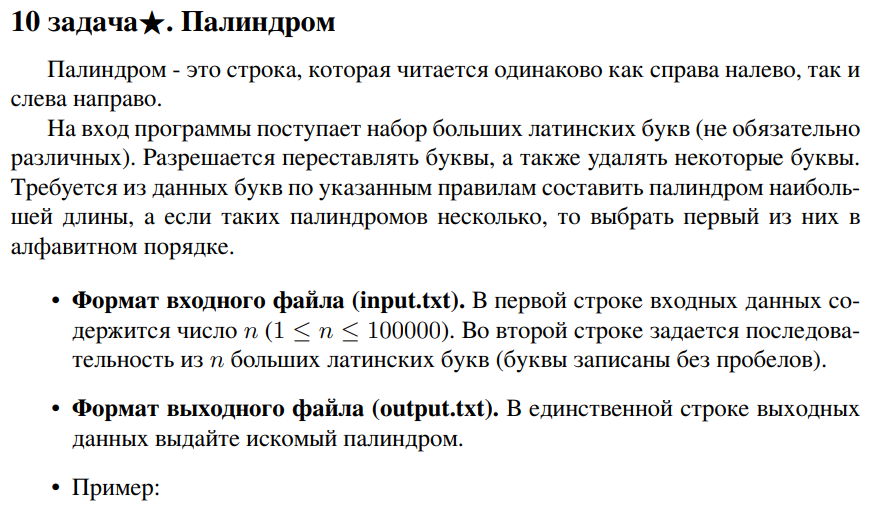


### Вывод

В целом, функция *bin\_add* является эффективным алгоритмом для сложения двоичных чисел, представленных в виде списков.

## Построение палиндромов

### Задание



### Код

*MakePalindrome.py*

**from** string **import** ascii\_uppercase

**def** make\_palindrome(letters: str) -> str:

letters\_cnt = [0] \* len(ascii\_uppercase)

**for** sym **in** letters:

**if** sym **not** **in** ascii\_uppercase:

**raise** ValueError("Unknown symbol")

letters\_cnt[ord(sym) - ord("A")] += 1

center = str()

palindrome\_half = []

**for** i **in** range(len(letters\_cnt)):

**if** letters\_cnt[i] % 2 == 1 **and** center == "":

center = chr(ord("A") + i)

palindrome\_half.extend([chr(ord("A") + i)] \* (letters\_cnt[i] // 2))

**return** "".join(palindrome\_half + [center] + list(reversed(palindrome\_half)))

*main.py*

**from** MakePalindrome **import** make\_palindrome

**def** main():

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

letters = file.readline()

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

**print**(make\_palindrome(letters), file=file)

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

main()

### Анализ кода

Функция *make\_palindrome* принимает строку *letters* в качестве входного параметра и возвращает палиндром в виде строки.

Функция создает список *letters\_cnt* длиной 26, где каждый элемент представляет собой количество вхождений соответствующей буквы в строке *letters*. Заполняя его, она итерирует по строке *letters* и для каждого символа увеличивает соответствующий элемент в списке *letters\_cnt*. Если символ не является буквой латинского алфавита, генерируется исключение *ValueError*.

Функция затем итерирует по списку *letters\_cnt* и для каждой буквы, которая встречается нечетное количество раз, добавляет ее в центр палиндрома.

Для каждой буквы, которая встречается четное количество раз, функция добавляет половину этого количества в начало и конец палиндрома. Первая буква, которая встречается нечетное количество раз, становится центральной.

Функция возвращает палиндром в виде строки.

