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: Тема работы
# Вариант 2

Выполнил:
Беляева В.А.
К3139

Проверила:

Санкт-Петербург
2024 г.

## Содержание отчета

## Задачи по варианту

### Задача №1. Сортировка вставкой

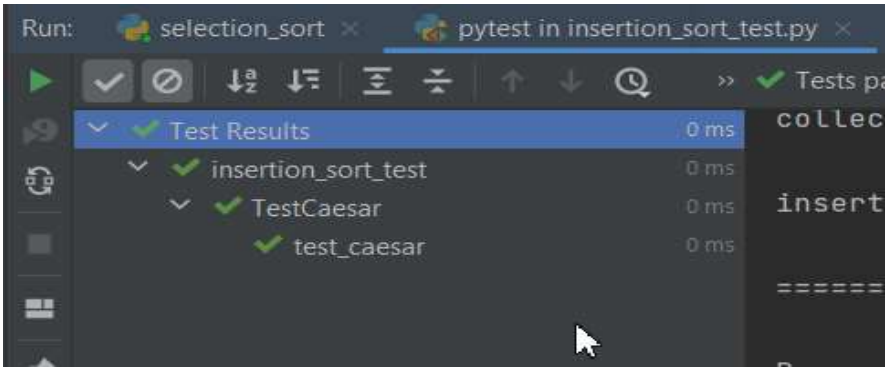
Используя код процедуры Insertion-sort, напишите программу и проверьте сортировку массива A = {31, 41, 59, 26, 41, 58}.

Листинг кода. (именно листинг, а не скрины)

```
def insertion_sort(to_sort: list) -> list:
  for i in range(1, len(to_sort)):
    key = to_sort[i]
    j = i - 1
    while j >= 0 and key < to_sort[j]:
          to_sort[j + 1] = to_sort[j]   # Shift the
element at index `j` one position to the right
      j -= 1
     to_sort[j + 1] = key   # Insert the `key` at the
correct position
  return to_sort  # Return the sorted list
```

Мы начинаем с первого элемента в списке, который считается уже отсортированным. Для каждого последующего элемента в списке: 1) Сохраняем его значение в переменной key. 2) Устанавливаем индекс j на позицию перед текущим элементом (то есть i - 1). 3) Пока индекс j не выходит за пределы отсортированной части списка и значение key меньше элемента на позиции j сдвигаем элемент на позицию j вправо, чтобы освободить место для key. Когда находим правильное место для key (то есть когда либо j становится меньше 0, либо значение на позиции j меньше или равно key), вставляем key в освободившуюся позицию.

Проверка соответствующих тестов из таблицы:




| | Время выполнения sec | Затраты памяти MB |
|---|---|---|
| [5, 4, 3, 2, -1] | 0.000031 | 0.000152 |
| [31, 41, 59, 26, 41, 58] | 0.000018 | 0.000160 |
| [1000000000, 1000000000, 50000000] | 0.000013 | 0.000160 |
| [-1000000000, -1000000000, 50000000] | 0.0056 | 0.0408 |
| [1 for i in range(1000)] | 0.0056 | 0.0408 |
| [i for i in range(1000, 0, -1)] | 01.01.99 | 0.0170 |

Вывод по задаче:Алгоритм имеет квадратичную асимптотику

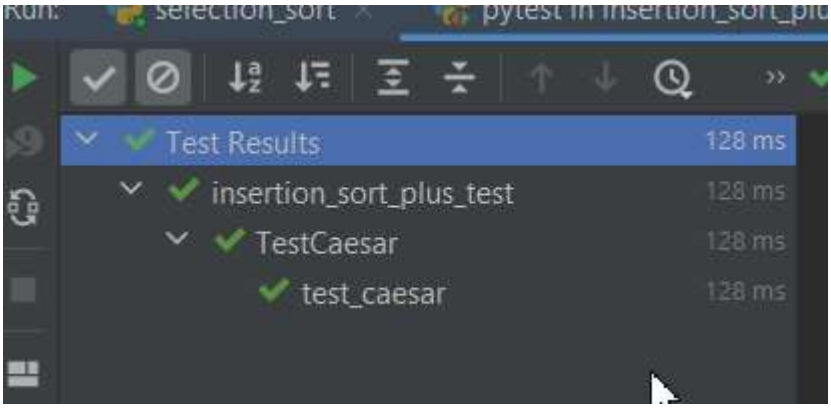
**Задача №2. Сортировка вставкой+**

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.

```
def insertion_sort(to_sort: list) -> list:
  n = len(to_sort)
swap_id = [1]
for i in range(1, n):
   key = to_sort[i]   # Store the current element as
the "key"
  j = i - 1
  while j >= 0 and key < to_sort[j]:
```

```
    to_sort[j + 1] = to_sort[j]
    j -= 1
  swap_id.append(j + 2)  # Append the new index
   to_sort[j + 1] = key  # Insert the `key` at the
correct position
return (swap_id, to_sort)
```

Мы начинаем с первого элемента в списке, который считается уже отсортированным. Для каждого последующего элемента в списке: 1) Сохраняем его значение в переменной key. 2) Устанавливаем индекс j на позицию перед текущим элементом (то есть i - 1). 3) Пока индекс j не выходит за пределы отсортированной части списка и значение key меньше элемента на позиции j сдвигаем элемент на позицию j вправо, чтобы освободить место для key. Сохраняем индекс в swap_id. Когда находим правильное место для key (то есть когда либо j становится меньше 0, либо значение на позиции j меньше или равно key), вставляем key в освободившуюся позицию.




| | Время выполнения sec | Затраты памяти MB |
|---|---|---|
| [5, 4, 3, 2, -1] | 00.000026 | 0.000152 |
| [31, 41, 59, 26, 41, 58] | 0.000012 | 0.000160 |
| [1000000000, 1000000000, 50000000] | 0.000011 | 0.000160 |
| [-1000000000, -1000000000, 50000000] | 0.000010 | 0.000160 |
| [1 for i in range(1000)] | 0.0078 | 0.04188 |
| [i for i in range(1000, 0, -1)] | 01.01.56 | 0.0195 |

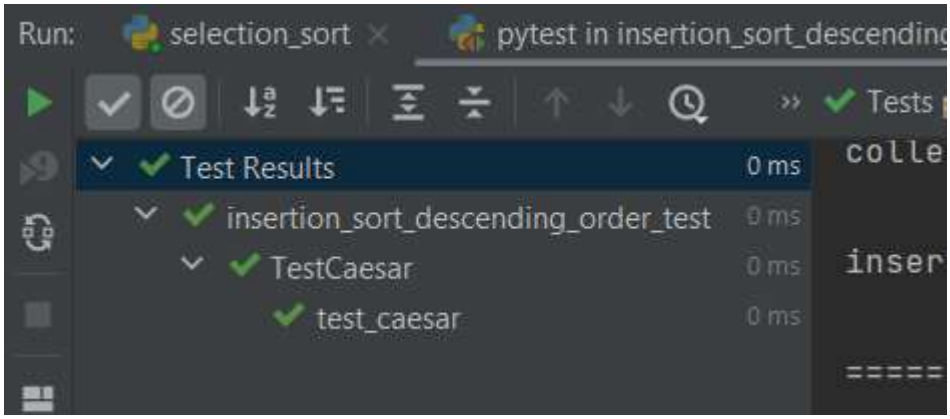
Вывод: В сортировке вставкой легко найти индек в процессе сортировки

**Задача №3. Сортировка вставкой по убыванию**

Перепишите процедуру Insertion-sort для сортировки в невозрастающем по-
рядке вместо неубывающего с использованием процедуры Swap.
Формат входного и выходного файла и ограничения - как в задаче 1.

```
def insertion_sort_reverse(to_sort: list) -> list:
  for i in range(1, len(to_sort)):
    key = to_sort[i]
    j = i - 1
    while j >= 0 and key > to_sort[j]:
          to_sort[j + 1] = to_sort[j]   # Shift the
element at index `j` one position to the right
      j -= 1
     to_sort[j + 1] = key  # Insert the `key` at the
correct position
  return to_sort  # Return the sorted list in reverse
order
```

Мы начинаем с первого элемента в списке, который считается уже отсортированным. Для каждого последующего элемента в списке: 1) Сохраняем его значение в переменной key. 2) Устанавливаем индекс j на позицию перед текущим элементом (то есть i - 1). 3) Пока индекс j не выходит за пределы отсортированной части списка и значение key больше элемента на позиции j сдвигаем элемент на позицию j вправо, чтобы освободить место для key. Когда находим правильное место для key (то есть когда либо j становится меньше 0, либо значение на позиции j меньше или равно key), вставляем key в освободившуюся позицию.




| | Время выполнения sec | Затраты памяти MB |
|---|---|---|
| [5, 4, 3, 2, -1] | 0.000024 | 0.000152 |
| [31, 41, 59, 26, 41, 58] | 0.000015 | 0.000160 |
| [1000000000, 1000000000, 50000000] | 0.000012 | 0.000160 |
| [-1000000000, -1000000000, 50000000] | 0.0000179 | 0.000160 |
| [1 for i in range(1000)] | 0.0060 | 0.0421 |
| [i for i in range(1000, 0, -1)] | 0.0060 | 0.0408 |

Вывод: В сортировке вставкой легко сделать по убыванию

**Задача №4 Линейный поиск**

Рассмотрим задачу поиска.

• Напишите код линейного поиска, при работе которого выполняется сканирование последовательности в поисках значения V .

• Если число встречается несколько раз, то выведите, сколько раз встречается

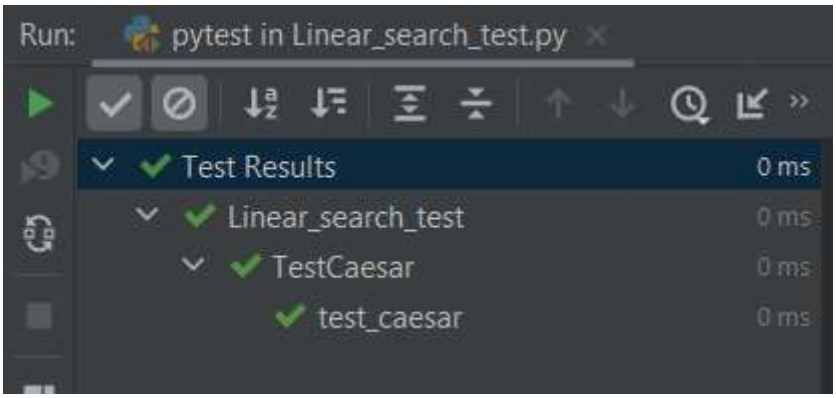
число и все индексы i через запятую.

• Дополнительно: попробуйте найти свинью, как в лекции. Используйте во входном файле последовательность слов из лекции, и найдите соответствующий индекс.

```
def linear_search(target: int, search_list: list) ->
list:
```

```
    to_return = []   # Initialize an empty list for
indexes
  for i in range(len(search_list)):
      if search_list[i] == target:   # Check if the
current element matches the target
          to_return.append(i)   # If found, append the
index to the return list
  if len(to_return) == 0:   # If the target id not in
search_list
    return [-1]
  return to_return
```

Создается пустой список to_return, который будет хранить индексы найденных совпадений. Происходит перебор всех элементов списка search_list с помощью цикла for. Для каждого элемента проверяется, равен ли он target. Если элемент совпадает с target, его индекс добавляется в список to_return. Если после завершения цикла to_return остается пустым (т.е. элемент не найден), функция возвращает список [-1]. Если совпадения найдены, возвращается список индексов найденных элементов.




(С учетом входного списка)

| тест | Время выполнения sec | Затраты памяти Bits |
|---|---|---|
| (1000, list(range(-1000, -800)) +list(range(800, 1001))) | 0.000034 | 27040 |
| (3, list(range(-1000, 4))+[3, 3, 3, 3]+list(range(600, 700))+list(range(3, 650))) | 0.000118 | 113952 |
| (250, list(range(200, 400))+list(range(100, | 0.000069 | 55776 |

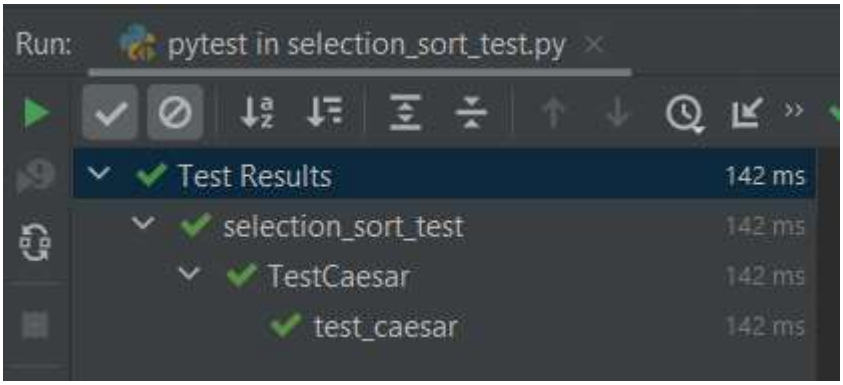
| | | |
|---|---|---|
| 300))+list(range(300, -150, -1))) | | |
| (-500, list(range(-550, -450))+list(range(-550, -450))+list(range(-550, -450))) | 0.000016 | 20576 |
| (100, [100 for i in range(100)]) | 0.000011 | 14944 |
| (0, [3, 2, 4, 0] * 5) | 0.000002 | 2912 |
| (-2, [3, 2, 4, 0] * 5) | 0.000002 | 2464 |

Вывод: Результаты показывают, что время выполнения и использование памяти увеличиваются с ростом размера списка, что ожидаемо для алгоритма линейного поиска.

**Задача №5 Сортировка выбором**

Рассмотрим сортировку элементов массива , которая выполняется следующим образом. Сначала определяется наименьший элемент массива , который ставится на место элементаA[1]. Затем производится поиск второго наименьшего элемента массива A, который ставится на место элемента A[2]. Этот процесс продолжается для первых n − 1 элементов массива A. Напишите код этого алгоритма, также известного как сортировка выбором (selection sort). Определите время сортировки выбором 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.

```
def selection_sort(to_sort: list) -> list:
    n = len(to_sort)
    for i in range(n - 1):
          # Find the index of the minimum element in
the remaining unsorted part of the to_sortay
        min_index = i
        for j in range(i + 1, n):
            if to_sort[j] < to_sort[min_index]:
                min_index = j
         # Swap the minimum element with the current
element
                    to_sort[i],  to_sort[min_index]  =
to_sort[min_index], to_sort[i]
    return to_sort
```




Переменная `n` хранит длину списка `to_sort`. Внешний цикл по индексу `i` проходит по каждому элементу списка, кроме последнего. Для каждого `i` внутренняя часть ищет индекс минимального элемента в оставшейся части списка. Переменная `min_index` хранит этот индекс. После нахождения минимального элемента происходит его обмен с текущим элементом на позиции `i`. По завершении всех итераций функция возвращает отсортированный список

(С учетом входящего списка)

| тест | Время выполнения sec | Затраты памяти Bits |
|---|---|---|
| [5, 4, 3, 2, -1] | 0.000008 | 1,6 |
| [31, 41, 59, 26, 41, 58] | 0.000004 | 1,66 |
| [1000000000, 1000000000, 50000000] | 0.000002 | 1,34 |
| [-1000000000, -1000000000, 50000000] | 0.000001 | 1,34 |
| [1 for i in range(1000)] | 0.0246 | 135,3 |
| [i for i in range(1000, 0)] | 0.2562 | 135,3 |

Вывод: Алгоритм сортировки выбором демонстрирует линейное увеличение времени выполнения с увеличением размера списка

Вывод по лабораторной: Простейшие алгоритмы поиска элемента в массиве и сортировок тривиальны, имеют линейную/квадратичную асимптотику соответственно и просты в написании.