

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 8

Выполнил:
Буй Тхук Хуен
К3139

Проверила:
Афанасьев А.В.

Санкт-Петербург
2024 г.

Содержание отчета

Содержание отчета	2
Задачи по варианту	3
Задача №1. Сортировка вставкой	3
Задача №3. Сортировка вставкой по убыванию	5
Задача №5. Сортировка выбором	10
Дополнительные задачи	
Задача №4. Линейный поиск	7
Задача №6. Пузырьковая сортировка	12
Вывод	16

Задачи по варианту

TASK 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 tracemalloc
import time

def insertion_sort(num):
    for i in range(1, n):
        elem = num[i]
        j = i - 1

        while j >= 0 and num[j] > elem:
            num[j + 1] = num[j]
            j -= 1
        num[j + 1] = elem
    return num

if __name__ == '__main__':
    f1 = open('input.txt', 'r')
    f2 = open('output.txt', 'w')

    tracemalloc.start()
    start = time.perf_counter()

    n = int(f1.readline())
    arr = list(map(int, f1.readline().strip().split()))
    if (n < 1 or n > 10 ** 3) or not all([abs(i) <= 10 ** 9 for i in arr]):
        print('Ввод неверен')

    result = insertion_sort(arr)
```

```
f2.write(' '.join(map(str,result)))
stop = time.perf_counter()

print("time:", stop - start)
print('memory usage:', tracemalloc.get_traced_memory()[1], 'bytes')
```

- Текстовое объяснение решения

Сначала открываю файлы для чтения (*input.txt*) и записи (*output.txt*) с помощью функции *open()*, входные данные взяты из файла *input.txt*, использую алгоритм сортировки вставками. Результаты записываются в файл *output.txt*.

def insertion_sort():

- + Внешний цикл: начинается со второго элемента (индекс 1) и проходит по каждому элементу в списке *num*.
- + *elem = num[i]* сохраняет значение текущего элемента.
- + Внутренний цикл: выполняет итерацию по отсортированным элементам (элементы перед элементом). Если текущий элемент больше *elem*, он будет сдвинут вправо (*num[j + 1] = num[j]*). Уменьшите индекс *j*, чтобы проверить предыдущий элемент.
- + когда правильная позиция найдена, элемент вставляется (*num[j + 1] = elem*).

- Результат работы кода

Task1\input.txt ×	
1	6
2	31 41 59 26 41 58
output.txt ×	
1	26 31 41 41 58 59


```
time: 0.00023950007744133472
memory usage: 8708 bytes
```

Тест	Время выполнения (s)	Затраты памяти(bytes)
6 31 41 59 26 41 58	0.00023950007744133472	8708

TASK 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 tracemalloc
import time

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

def insertion_sort(arr):
    for i in range(1, len(arr)):
        elem = arr[i]
        j = i-1

        while j >= 0 and arr[j] > elem:
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = elem
    return arr

def reverse_arr(arr):
    n = len(arr)
    for i in range(n//2):
        swap(arr, i, n-i-1)

if __name__ == '__main__':
    f1 = open('input.txt', 'r')
    f2 = open('output.txt', 'w')
```

```

tracemalloc.start()
start = time.perf_counter()

n = int(f1.readline())
num = list(map(int, f1.readline().strip().split()))
if (n < 1 or n > 10 ** 3) or not all([abs(i) <= 10 ** 9 for i in num]):
    print('Ввод неверен')

insertion_sort(num)
reverse_arr(num)
f2.write(' '.join(map(str, num)))
stop = time.perf_counter()

print("time:", stop - start)
print('memory usage:', tracemalloc.get_traced_memory()[1], 'bytes')

```

- Текстовое объяснение решения.

Сначала открываю файлы для чтения (*input.txt*) и записи (*output.txt*) с помощью функции *open()*, входные данные взяты из файла *input.txt*, использую алгоритм сортировки вставками. Затем я использую функцию *swap()*, чтобы поменять местами положение *i*-го элемента и соответствующего ему элемента при итерации с конца списка. Результаты записываются в файл *output.txt*.

- Результат работы кода на примерах из текста задачи:

≡ input.txt ×	
1	6
2	31 41 59 26 41 58
≡ output.txt ×	
1	59 58 41 41 31 26

time: 0.00023399991914629936
memory usage: 8708 bytes

Тест	Время выполнения (s)	Затраты памяти (bytes)
6 31 41 59 26 41 58	0.00023399991914629936	8708

TASK 4: Линейный поиск

Рассмотрим задачу поиска.

- **Формат входного файла.** Последовательность из n чисел $A = a_1, a_2, \dots, a_n$ в первой строке, числа разделены пробелом, и значение V во второй строке. Ограничения: $0 \leq n \leq 10^3$, $-10^3 \leq a_i, V \leq 10^3$
- **Формат выходного файла.** Одно число - индекс i , такой, что $V = A[i]$, или значение -1 , если V в отсутствует.
- 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.

- Листинг кода

```
import tracemalloc
import time

def linear_search(arr, x):
    indices = []
    count = 0
    for i in range(len(arr)):
        if arr[i] == x:
            indices.append(i)
            count += 1

    if count == 0:
        return -1
    else:
        return count, indices

if __name__ == '__main__':
    f1 = open('input.txt', 'r')
    f2 = open('output.txt', 'w')
    start = time.perf_counter()
    tracemalloc.start()

    num = list(map(int, f1.readline().strip().split()))
    target = int(f1.readline())
    if (len(num) < -10**3 or len(num) > 10**3) or not all([i >= -10**3
for i in num]) or target > 10**3:
        print('Ввод неверен.')
```



```

result = linear_search(num, target)
count, indices = result

if count == 1:
    f2.write(f'{indices}\n')
else:
    f2.write(f'{count}, {" ".join(map(str, indices))}\n')

stop = time.perf_counter()
print(f'time: {stop-start:.15f} second')
print('memory usage:', tracemalloc.get_traced_memory()[1], 'bytes')

```

- Текстовое объяснение решения.

Введите входные данные, преобразую последовательность чисел в формат списка, используя алгоритм линейного поиска `linear_search()`: Цикл `for` проходит через каждое значение в массиве. Если значение любого элемента равно значению искомого элемента, добавит индекс этого элемента к списку *indices*, переменная *count* увеличится на 1. Использую переменную *count* для подсчета количества повторений, если искомый элемент встречается в массиве несколько раз.

- Результат работы кода на примерах из текста задачи:

```

0 3 5 8 10 12 15 18 20 20 50 60
12
5
time: 0.000045499997213 second
memory usage: 8814 bytes

```

```

0 3 5 8 10 12 15 18 20 20 50 60
20
2, 8 9
time: 0.000053500058129 second
memory usage: 8814 bytes

```

Тест	Время выполнения (s)	Затраты памяти (bytes)
0 3 5 8 10 12 15 18 20 20 50 60 12	0,000045499997213	8814
0 3 5 8 10 12 15 18 20 20 50 60 20	0,000053500058129	8814

TASK 5: Сортировка выбором

Рассмотрим сортировку элементов массива, которая выполняется следующим образом. Сначала определяется наименьший элемент массива, который ставится на место элемента $A[1]$. 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.

- Листинг кода

```
import tracemalloc
import time
def selection_sort(l, arr):
    for i in range(l):
        min_index = i
        for j in range(i+1, l):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]
    return arr

if __name__ == '__main__':
    f1 = open('input.txt', 'r')
    f2 = open('output.txt', 'w')
    tracemalloc.start()
    start = time.perf_counter()

    n = int(f1.readline())
    num = list(map(int, f1.readline().strip().split()))
    if (n < 1 or n > 10 ** 3) or not all([abs(i) <= 10 ** 9 for i in num]):
        print('Ввод неверен')

    f2.write(' '.join(map(str, selection_sort(n, num))))
    stop = time.perf_counter()
    print("time:", stop - start)
    print('memory usage:', tracemalloc.get_traced_memory()[1], 'bytes')
```

- Текстовое объяснение решения

Сначала открываю файлы для чтения (*input.txt*) и записи (*output.txt*) с помощью функции *open()*, входные данные взяты из файла *input.txt*, использую алгоритм сортировки выбором. Результаты записываются в файл *output.txt*.

def selection_sort():

- + Этот цикл выполняется от индекса *0* до *l-1*, представляя каждую позицию в массиве.
- + *min_index = i*, т.е. предположим, что элемент с номером *i* является наименьшим в остальном массиве
- + Этот цикл просматривает оставшийся массив (*от i + 1 до l*), чтобы найти наименьший элемент. Если элемент с индексом *j* меньше элемента с индексом *min_index*, обновите *min_index* до *j*.
- + Найдя наименьший элемент в остальной части массива, замените его элементом с номером *i*.

- Результат работы кода:

The screenshot shows a code editor with two files open: *Task5\input.txt* and *output.txt*. The *input.txt* file contains two lines: the first line has the number '6', and the second line contains the numbers '31 41 59 26 41 58'. The *output.txt* file contains one line with the numbers '26 31 41 41 58 59'. Below the files, the execution statistics are displayed: *time: 0.0002855999628081918* and *memory usage: 8708 bytes*.

Тест	Время выполнения (s)	Затраты памяти (bytes)
6 31 41 59 26 41 58	0.0002855999628081918	8708

TASK 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 корректности процедуры вам необходимо доказать, что она завершается и что $A'[1] \leq A'[2] \leq \dots \leq A'[n]$, 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

```
import time  
  
import tracemalloc  
  
def bubble_sort(num):  
    for i in range(len(num)-1):  
        for j in range(len(num)-1, i, -1):  
            if num[j] < num[j-1]:  
                num[j], num[j-1] = num[j-1], num[j]  
  
    return num  
  
if __name__ == '__main__':  
    f1 = open('input.txt', 'r')  
    f2 = open('output.txt', 'w')  
    start = time.perf_counter()  
    tracemalloc.start()  
  
    n = int(f1.readline())
```

```

arr = list(map(int, f1.readline().strip().split()))

if (n < 1 or n > 10 ** 3) or not all([abs(i) <= 10 ** 9 for i in arr]):
    print('Ввод неверен')

result = bubble_sort(arr)

f2.write(' '.join(map(str, result)))

stop = time.perf_counter()

print("time:", stop - start)

print('memory usage:', tracemalloc.get_traced_memory()[1], 'bytes')

```

Время пузырьковой сортировки в наихудшем случае и в среднем случае: $O(n^2)$

Обе сортировки имеют одинаковую временную сложность, но сортировка вставками часто работает быстрее из-за меньшего числа перемещений элементов.

- Текстовое объяснение решения

Сначала открываю файлы для чтения (*input.txt*) и записи (*output.txt*) с помощью функции *open()*, входные данные взяты из файла *input.txt*, использую алгоритм пузырьковая сортировка. Результаты записываются в файл *output.txt*.

def bubble_sort():

- + Этот цикл выполняется от 0 до $len(num) - 2$. Каждый раз при прохождении внешнего цикла самый большой элемент в неотсортированной секции будет помещен в правильное положение.
- + Этот цикл выполняется от конца массива ($len(num) - 1$) к индексу *i*. Это позволяет сравнивать каждую пару соседних элементов для их сортировки.
- + Если элемент с индексом *j* меньше элемента с индексом *j - 1*, они будут заменены местами. Это гарантирует, что больший элемент будет вниз массива на каждой итерации.

- Результат работы кода:

```
Task6\input.txt ×
1 | 6
2 | 31 41 59 26 41 58

output.txt ×
1 | 26 31 41 41 58 59

time: 0.00011999998241662979
memory usage: 8708 bytes
```

Тест	Время выполнения (s)	Затраты памяти (bytes)
6 31 41 59 26 41 58	0.00011999998241662979	8708

Вывод :

Алгоритм	Временная сложность	Пространственная сложность	Стабильность	Область применения
Сортировка вставкой	$O(n)$ - лучший случай $O(n^2)$ - средний и худший случай	$O(1)$	Стабильный	Подходит для небольших списков или почти отсортированных
Сортировка выбором	$O(n^2)$ - все случаи	$O(1)$	Нестабильный	Легко понять и реализовать, но неэффективен для больших списков
Пузырьковая сортировка	$O(n)$ - лучший случай $O(n^2)$ - средний и худший случай	$O(1)$	Нестабильный	Легко реализовать, но низкая производительность для больших списков

Все три алгоритма — сортировка вставками, выбором и пузырьком — имеют временную сложность $O(n^2)$ в худшем случае, что делает их неэффективными для больших списков. Хотя сортировка вставками может работать лучше в некоторых конкретных ситуациях, сортировка выбором и пузырьком обычно не рекомендуются для практического применения из-за низкой производительности.

- **Линейный поиск:**

- + лучший случай: $O(1)$ — когда искомый элемент является первым элементом.

- + средний и худший случай : $O(n)$ — в случае случайного массива.
- + Сложность памяти: $O(1)$ — дополнительная память, кроме временных переменных, не требуется.
- + Простота реализации.
- + Может работать с неупорядоченными массивами
- + Низкая эффективность для больших массивов.