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

# Отчет по лабораторной работе №2
# по курсу «Алгоритмы и структуры данных»
# Тема: Сортировки.
# Вариант 9

Выполнила:
Гашимов И.Ф.
К3139

Проверил:
Афанасьев А. В.

Санкт-Петербург
2024 г.

## Содержание отчета

# 1. Сортировка слиянием.

1. Используя *псевдокод* процедур `Merge` и `Merge-sort` из презентации к Лекции 2 (страницы 6-7), напишите программу сортировки слиянием на Python и проверьте сортировку, создав несколько рандомных массивов, подходящих под параметры:

   - **Формат входного файла (input.txt).** В первой строке входного файла содержится число $n$ ($1 \leq n \leq 2 \cdot 10^4$) — 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.

2. Для проверки можно выбрать наихудший случай, когда сортируется массив размера 1000, $10^4, 10^5$ чисел порядка $10^9$, отсортированных в обратном порядке; наилучший, когда массив уже отсортирован, и средний. Сравните, например, с сортировкой вставкой на этих же данных.

3. Перепишите процедуру `Merge` так, чтобы в ней не использовались сигнальные значения. Сигналом к остановке должен служить тот факт, что все элементы массива $L$ или $R$ скопированы обратно в массив $A$, после чего в этот массив копируются элементы, оставшиеся в непустом массиве.

*или* перепишите процедуру `Merge` (и, соответственно, `Merge-sort`) так, чтобы в ней не использовались значения границ и середины - $p, r$ и $q$.

```
def merge(A, p, q, r):

    n1 = q - p + 1

    n2 = r - q



    L = [0] * n1

    R = [0] * n2
```

```
for i in range(n1):
    L[i] = A[p + i]


for j in range(n2):
    R[j] = A[q + 1 + j]


L.append(float('inf'))
R.append(float('inf'))


i = 0
j = 0
for k in range(p, r + 1):
    if L[i] <= R[j]:
        A[k] = L[i]
        i += 1
    else:
        A[k] = R[j]
        j += 1
```

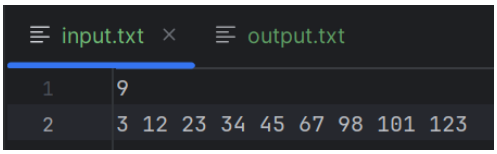
```python
def merge_sort(A, p, r):

    if p < r:

        q = (p + r) // 2

        merge_sort(A, p, q)

        merge_sort(A, q + 1, r)

        merge(A, p, q, r)

    return A
```

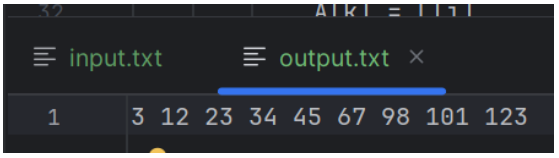
Этот код использует метод сортировки под названием "сортировка слиянием". Сначала он разбивает массив на две части, пока не останется только один элемент в каждой части. Затем он собирает эти части обратно, сравнивая элементы и помещая их в правильном порядке. В конце весь массив оказывается отсортированным.

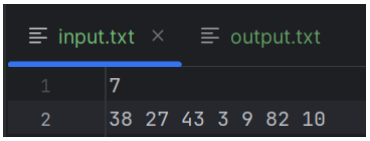
merge - это вспомогательная функция, которая объединяет два отсортированных подмассива в один отсортированный массив.

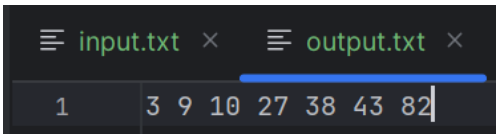
merge_sort - это основная функция, которая реализует алгоритм сортировки слиянием.

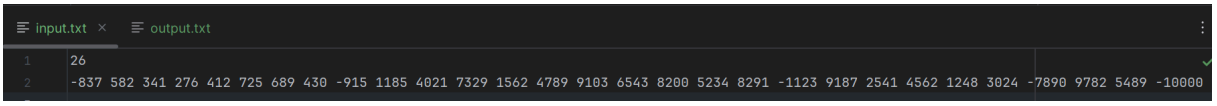
| / | time and memory |
|---|---|
| test_one(самый простой) | Время сортировки: 0.000161 секунд<br>Использование памяти: 0.00390625 МБ |
| test_two(средний) | Время сортировки: 0.000139 секунд<br>Использование памяти: 0.00390625 МБ |
| test_two(самый сложный) | Время сортировки: 0.000223 секунд<br>Использование памяти: 0.00390625 МБ |

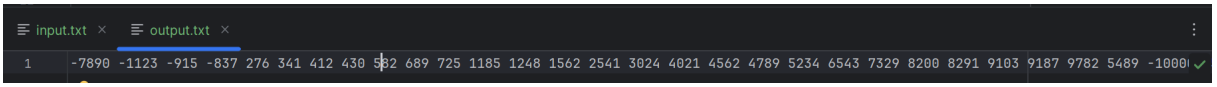



- Удалены сигнальные значения (float('inf')) в массивах L и R
- Используется цикл while, который продолжает сравнивать элементы из L и R, пока оба массива имеют элементы.
- В конце добавлены два отдельных цикла while, чтобы добавить оставшиеся элементы из L и R, если один из них был исчерпан.

Теперь merge не использует сигнальные значения

```python
def merge_nofl(A, p, q, r):
    n1 = q - p + 1
    n2 = r - q

    L = [0] * n1
    R = [0] * n2

    for i in range(n1):
        L[i] = A[p + i]

    for j in range(n2):
        R[j] = A[q + 1 + j]

    i, j, k = 0, 0, p
    while i < n1 and j < n2:
        if L[i] <= R[j]:
            A[k] = L[i]
            i += 1
        else:
            A[k] = R[j]
            j += 1
```

```
        k += 1

    while i < n1:  # Копируем оставшиеся элементы L, если есть
        A[k] = L[i]
        i += 1
        k += 1

    while j < n2:  # Копируем оставшиеся элементы R, если есть
        A[k] = R[j]
        j += 1
        k += 1


def merge_sort(A, p, r):
    if p < r:
        q = (p + r) // 2
        merge_sort(A, p, q)
        merge_sort(A, q + 1, r)
        merge_nofl(A, p, q, r)
    return A
```

# 3.Число Инверсий

Инверсией в последовательности чисел $A$ называется такая ситуация, когда $i < j$, а $A_i > A_j$. Количество инверсий в последовательности в некотором роде определяет, насколько близка данная последовательность к отсортированной. Например, в сортированном массиве число инверсий равно 0, а в массиве, сортированном наоборот - каждые два элемента будут составлять инверсию (всего $n(n-1)/2$).

Дан массив целых чисел. Ваша задача — подсчитать число инверсий в нем.

Подсказка: чтобы сделать это быстрее, можно воспользоваться модификацией сортировки слиянием.

- **Формат входного файла (input.txt).** В первой строке входного файла содержится число $n$ ($1 \leq n \leq 10^5$) — число элементов в массиве. Во второй строке находятся $n$ различных целых чисел, по модулю не превосходящих $10^9$.
- **Формат выходного файла (output.txt).** В выходной файл надо вывести число инверсий в массиве.
- Ограничение по времени. 2сек.
- Ограничение по памяти. 256 мб.

merge_sort_and_count: Рекурсивно делит массив на две половины и вызывает merge_and_count для их слияния.
merge_and_count: Сливает два отсортированных подмассива и подсчитывает количество инверсий.

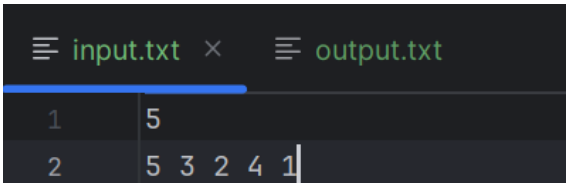
merge_sort_and_count:

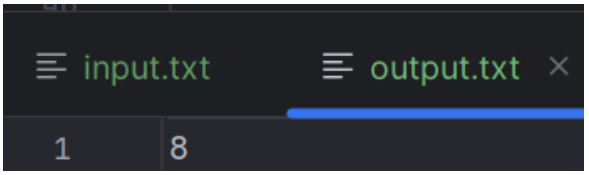
Рекурсивно делит массив на две части и подсчитывает инверсии, вызывая merge_and_count

```
def merge_and_count(arr, L, mid, R):
    left_part = arr[L:mid + 1]
    right_part = arr[mid + 1:R + 1]

    i = 0
    j = 0
    k = L
```

```
    inversions = 0

    while i < len(left_part) and j < len(right_part):
        if left_part[i] <= right_part[j]:
            arr[k] = left_part[i]
            i += 1
        else:
            arr[k] = right_part[j]
            inversions += (len(left_part) - i)
            j += 1
        k += 1

    while i < len(left_part):
        arr[k] = left_part[i]
        i += 1
        k += 1

    while j < len(right_part):
        arr[k] = right_part[j]
        j += 1
        k += 1

    return inversions

def merge_sort_and_count(arr, L, R):
    inversions = 0
    if L < R:
        mid = (L + R) // 2
        inversions += merge_sort_and_count(arr, L, mid)
        inversions += merge_sort_and_count(arr, mid + 1, R)
        inversions += merge_and_count(arr, L, mid, R)
    return inversions
```

| / | time and memory |
|---|---|
| test: [5, 4, 3, 2, 1] | Число инверсий: 8<br>Время выполнения: 0.000011 секунд<br>Использование памяти: 23.57812500 МБ |



5.Представитель Большинства

Правило большинства - это когда выбирается элемент, имеющий больше половины голосов. Допустим, есть последовательность $A$ элементов $a_1, a_2, ...a_n$, и нужно проверить, содержит ли она элемент, который появляется больше, чем $n/2$ раз. Наивный метод это сделать:



```
Majority(A):
for i from 1 to n:
        current_element = a[i]
        count = 0
        for j from 1 to n:
                 if a[j] = current_element:
                        count = count+1
        if count > n/2:
                return a[i]
return "нет элемента большинства"
```

Очевидно, время выполнения этого алгоритма квадратично. Ваша цель - использовать метод "Разделяй и властвуй"для разработки алгоритма проверки, содержится ли во входной последовательности элемент, который встречается больше половины раз, за время $O(n \log n)$.

Функция Majority(A) принимает список A и ищет элемент, который встречается более чем n/2 раз. Если такой элемент найден, функция возвращает его; в противном случае возвращает None.

```python
def Majority(A):
    n = len(A)

    for i in range(n):
        current_element = A[i]
        count = 0

        for j in range(n):
            if A[j] == current_element:
                count += 1

        if count > n // 2:
            return current_element

    return None
```

Функция Majority(A) принимает список A и подсчитывает количество вхождений каждого элемента, чтобы определить, встречается ли какой-либо элемент более чем n/2n/2n/2 раз. Если такой элемент найден, функция возвращает его, иначе возвращает None. После вызова функции в основном коде, проверяется, был ли найден элемент большинства, и в зависимости от результата записывается 1 или 0 в файл output.txt

```python
import time
import os
import psutil

def get_memory_usage():
    process = psutil.Process(os.getpid())
    return process.memory_info().rss / (1024 ** 2)

initial_memory = get_memory_usage()
start_time = time.perf_counter()

def Majority(A):
    n = len(A)

    for i in range(n):
        current_element = A[i]
        count = 0

        for j in range(n):
            if A[j] == current_element:
```

```
                count += 1

        if count > n // 2:
            return current_element

    return None

with open('input.txt', 'r') as f:
    n = int(f.readline())
    unlist = list(map(int, f.readline().split()))

if not (1 <= n <= 10**5):
    print('[Ошибка] количество элементов должно быть от 1 до
100.000!')
    exit(1)

for i in unlist:
    if abs(i) > 10 ** 9:
        print('[Ошибка] Числа должны быть не больше 10**9 по
модулю!')
        exit(1)


majority_element = Majority(unlist)

end_time = time.perf_counter()
final_memory = get_memory_usage()
time_elapsed = end_time - start_time
memory_used = final_memory - initial_memory

if majority_element is not None:
    result = 1
else:
    result = 0

print(f"Время выполнения: {time_elapsed:.6f} секунд")
print(f"Использование памяти: {memory_used:.8f} МБ")

# Записываем результат в файл
with open('output.txt', 'w') as f:
    f.write(str(result))
```