

САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ  
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ  
ИТМО  
ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №2  
по курсу «Алгоритмы и структуры данных»  
Тема: Сортировка слиянием. Метод декомпозиции  
Вариант 17

Выполнил:  
Останин А. С.  
К3140

Проверил:  
Афанасьев А. В.

Санкт-Петербург  
2024 г.

## Содержание отчета

<b>Содержание отчета</b>	<b>2</b>
<b>Задачи по варианту</b>	<b>3</b>
Задача №1. Сортировка слиянием	
Задача №4. Бинарный поиск	
Задача №5. Большинство числа	
<b>Вывод</b>	<b>5</b>

## Задачи по варианту

### Задача №1. Сортировка слиянием

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 случайных массивов, подходящих под параметры:

```
import time, tracemalloc
from utils import read_file, write_file, close_files

def Merge(A, p, q, r):
    n1 = q - p + 1
    n2 = r - q
    L = []
    R = []

    for i in range(n1):
        L.append(A[p + i])
    for j in range(n2):
        R.append(A[q + j + 1])

    i, j = 0, 0
    k = p
    while i < len(L) and j < len(R):
        if L[i] <= R[j]:
            A[k] = L[i]
            i += 1
        else:
            A[k] = R[j]
            j += 1
        k += 1

    while i < len(L):
        A[k] = L[i]
        i += 1
        k += 1

    while j < len(R):
        A[k] = R[j]
        j += 1
        k += 1

def MergeSort(A, p, r):
    if p < r:
        q = (p + r) // 2
        MergeSort(A, p, q)
        MergeSort(A, q + 1, r)
        Merge(A, p, q, r)
    return A

def main(input_file, output_file, info_file):
    t_start = time.perf_counter()
    tracemalloc.start()

    in_f = read_file(input_file)
    out_f = write_file(output_file)
    info_f = write_file(info_file)
```

```

A = [int(x) for x in in_f[1].split()]

p, r = 0, len(A) - 1

A = MergeSort(A, p, r)

info_f.write(f'time: {time.perf_counter() - t_start} s\nmemory:
{tracemalloc.get_traced_memory()[1] / 2 ** 20} Mb')
tracemalloc.stop()

res_s = ''
for i in A:
    res_s += str(i) + ' '

out_f.write(res_s)

close_files(out_f, info_f)

```

В функции Merge копируем все элементы из списков.

```

for i in range(n1):
    L.append(A[p + i])
for j in range(n2):
    R.append(A[q + j + 1])

```

Так как, по заданию, не нужно использовать сигнальные значение, то вместо них будем использовать обычные индексы

```

i, j = 0, 0
k = p

```

Теперь выполним слияние двух отсортированных списков в один

```

while i < len(L) and j < len(R):
    if L[i] <= R[j]:
        A[k] = L[i]
        i += 1
    else:
        A[k] = R[j]
        j += 1
    k += 1

```

Теперь, если есть оставшиеся элементы в списках, то копируем их

```
while i < len(L):
```

```
    A[k] = L[i]
```

```
    i += 1
```

```
    k += 1
```

```
while j < len(R):
```

```
    A[k] = R[j]
```

```
    j += 1
```

```
    k += 1
```

Перейдём к функции MergeSort, которая разделяет список на два списка, разделённых с середины изначального, при этом выполняясь рекурсивно. На выходе получим отсортированный A

```
def MergeSort(A, p, r):
```

```
    if p < r:
```

```
        q = (p + r) // 2
```

```
        MergeSort(A, p, q)
```

```
        MergeSort(A, q + 1, r)
```

```
        Merge(A, p, q, r)
```

```
    return A
```

И в конце считаем время выполнения и затраченную память, считываем файлы, проходим по заданному списку значений и записываем ответ в файл

```
def main(input_file, output_file, info_file):
    t_start = time.perf_counter()
    tracemalloc.start()

    in_f = read_file(input_file)
    out_f = write_file(output_file)
    info_f = write_file(info_file)

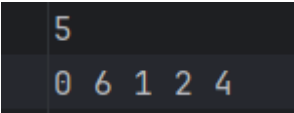
    A = [int(x) for x in in_f[1].split()]

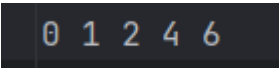
    p, r = 0, len(A) - 1

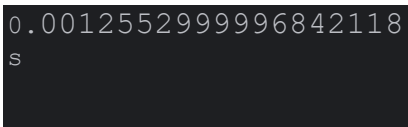
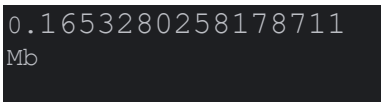
    A = MergeSort(A, p, r)

    info_f.write(f'time: {time.perf_counter() - t_start} s\nmemory:
{tracemalloc.get_traced_memory()[1] / 2 ** 20} Mb')
    tracemalloc.stop()

    res_s = ''
    for i in A:
        res_s += str(i) + ' '
    out_f.write(res_s)
    close_files(out_f, info_f)
```

Input: 

Output: 

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи		
Пример из задачи		
Верхняя граница диапазона значений входных данных из текста задачи		

Вывод по задаче: Написан алгоритм сортировки слиянием, выполняющегося за  $O(n \log n)$

## Задача №4. Бинарный поиск

В этой задаче вы реализуете алгоритм бинарного поиска, который позволяет очень эффективно искать (даже в огромных) списках при условии, что список отсортирован. Цель - реализация алгоритма двоичного (бинарного) поиска.

```
import time, tracemalloc
from utils import read_file, write_file, close_files

def BinSearch(A, n):
    if len(A) == 0:
        return -1
    if len(A) == 1:
        if A[0] == n:
            return 0
        return -1

    left = 0
    right = len(A) - 1
```

```

while left <= right:
    mid = (left + right) // 2

    if n == A[mid]:
        return mid
    elif n > A[mid]:
        left = mid + 1
    else:
        right = mid - 1

return -1

def search(input_file, output_file, info_file):
    t_start = time.perf_counter()
    tracemalloc.start()

    in_f = read_file(input_file)
    out_f = write_file(output_file)
    info_f = write_file(info_file)

    a = in_f[1].strip()
    b = in_f[3].strip()
    result_s = ''
    for i in b.split():
        result_s += str(BinSearch(a.split(), i)) + ' '
    out_f.write(result_s)

    info_f.write(f'time: {time.perf_counter() - t_start} s\nmemory:
{tracemalloc.get_traced_memory()[1] / 2 ** 20} Mb')

    tracemalloc.stop()

    close_files(info_f, out_f)

```

Напишем функцию BinSearch.

```

def BinSearch(A, n):
    if len(A) == 0:
        return -1
    if len(A) == 1:
        if A[0] == n:
            return 0
        return -1

    left = 0
    right = len(A) - 1

    while left <= right:
        mid = (left + right) // 2

        if n == A[mid]:
            return mid
        elif n > A[mid]:
            left = mid + 1
        else:
            right = mid - 1

    return -1

```

Для начала рассмотрим случаи, когда в списке элементов нет и когда только один

```
if len(A) == 0:
    return -1
if len(A) == 1:
    if A[0] == n:
        return 0
    return -1
```

Теперь напомним алгоритм бинарного поиска. Будем рекурсивно сужать предел поиска в 2 раза, путём нахождения элемента в середине и сравнения его с заданным элементом, каждый проход, пока не останется единственное значение, после чего проверим подходит ли оно нам.

```
left = 0
right = len(A) - 1
while left <= right:
    mid = (left + right) // 2

    if n == A[mid]:
        return mid
    elif n > A[mid]:
        left = mid + 1
    else:
        right = mid - 1
```



Считываем входные данные, проходим циклом по заданным значениям функцией BinSearch и записываем их результат в output.txt. Также затраченное время и память в info.txt

```
def search(input_file, output_file, info_file):
    t_start = time.perf_counter()
    tracemalloc.start()

    in_f = read_file(input_file)
    out_f = write_file(output_file)
    info_f = write_file(info_file)

    a = in_f[1].strip()
    b = in_f[3].strip()
    result_s = ''
    for i in b.split():
        result_s += str(BinSearch(a.split(), i)) + ' '
    out_f.write(result_s)

    info_f.write(f'time: {time.perf_counter() - t_start} s\nmemory:
{tracemalloc.get_traced_memory()[1] / 2 ** 20} Mb')

    tracemalloc.stop()

    close_files(info_f, out_f)
```

Input:

```
5
1 5 8 12 13
5
8 1 23 1 11
```

Output:

```
2 0 -1 0 -1
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи		
Пример из задачи	0.0013339000006453716 s	0.01873016357421875 Mb
Верхняя граница диапазона значений входных данных из		

Вывод по задаче: Написан алгоритм бинарного поиска, выполняющегося за  $O(\log n)$

## Задача №5. Большинство числа

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

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  $O(n \log n)$ .

```
import time, tracemalloc
from utils import read_file, write_file, close_files

def majority_number(A):
    if len(A) == 1:
        return A[0]

    q = len(A) // 2
    left = majority_number(A[:q])
    right = majority_number(A[q:])

    if left == right:
        return left

    left_count = A.count(left)
    right_count = A.count(right)

    if left_count > right_count:
        return left
    else:
        return right

def majority(input_file, output_file, info_file):
    t_start = time.perf_counter()
    tracemalloc.start()

    in_f = read_file(input_file)
    out_f = write_file(output_file)
    info_f = write_file(info_file)

    ar = in_f[1].split()

    if ar.count(majority_number(ar)) > len(ar) / 2:
        out_f.write('1')
    else:
        out_f.write('0')
```

```

        info_f.write(f'time: {time.perf_counter() - t_start} s\nmemory:
{tracemalloc.get_traced_memory()[1] / 2 ** 20} Mb')

    tracemalloc.stop()

    close_files(info_f, out_f)

```

Напишем функцию `majority_number`,

```

def majority_number(A):
    if len(A) == 1:
        return A[0]

    q = len(A) // 2
    left = majority_number(A[:q])
    right = majority_number(A[q:])

    if left == right:
        return left

    left_count = A.count(left)
    right_count = A.count(right)

    if left_count > right_count:
        return left
    else:
        return right

```

которая будет рекурсивно искать число, повторяющееся большее число раз. Функция разделяет список на две части, рекурсивно вычисляя элемент большинства для каждой половины и, если результаты совпадают, возвращает этот элемент, а если результаты разные, считает количество вхождений каждого элемента и возвращает тот, который встречается чаще.

Далее в функции `majority` считываем входные данные, обрабатываем их функцией и записываем результат в `output.txt`. Также затраченное время и память в `info.txt`

```

def majority(input_file, output_file, info_file):
    t_start = time.perf_counter()
    tracemalloc.start()

    in_f = read_file(input_file)
    out_f = write_file(output_file)
    info_f = write_file(info_file)

    ar = in_f[1].split()

    if ar.count(majority_number(ar)) > len(ar) / 2:
        out_f.write('1')
    else:
        out_f.write('0')

    info_f.write(f'time: {time.perf_counter() - t_start} s\nmemory:
{tracemalloc.get_traced_memory()[1] / 2 ** 20} Mb')

```

```

tracemalloc.stop()

close_files(info_f, out_f)

```

Input:

```

5
1 2 3 3 3

```

Output: 1

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи		
Пример из задачи	0.0012511999993876088 s	0.01836681365966797 Mb
Верхняя граница диапазона значений входных данных из текста задачи		

Вывод по задаче: Написан алгоритм поиска большинства числа, выполняющегося за  $O(n \log n)$

## Вывод

При выполнении этой лабораторной, мы научились реализовывать алгоритм сортировки слиянием, бинарного поиска и нахождения большинства числа