

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: Сортировка слиянием. Метод декомпозиции

Выполнила:
Жмачинская Д.С.
К3141

Проверила:
Ромакина О.М.

Санкт-Петербург
2024 г.

Содержание отчета

Содержание отчета	2
Задачи	
Задача №1. Сортировка слиянием	3
Задача №3. Число инверсий	5
Задача №4. Бинарный поиск	8
Задача №5. Представитель большинства	10
Задача №7. Поиск максимального подмассива за линейное время	12
Вывод	15

1 задача. 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 .

```
import time, tracemalloc
def merge_sort(lst):
    ln = len(lst)//2
    mas1 = lst[:ln]
    mas2 = lst[ln:]
    if len(mas1) > 1:
        mas1 = merge_sort(mas1)
    if len(mas2) > 1:
        mas2 = merge_sort(mas2)
    return merge(mas1, mas2)
def merge(a, b):
    mas = []
    i = 0
    j = 0
```

```

while i < len(a) and j < len(b):
    if a[i] <= b[j]:
        mas.append(a[i])
        i += 1
    else:
        mas.append(b[j])
        j += 1
    mas += a[i:] + b[j:]
return mas
tracemalloc.start()
t_start = time.perf_counter()
f = open("input.txt")
n = int(f.readline())
mas = [int(el) for el in f.readline().split()]
f.close()
str_lst = list(map(str, merge_sort(mas)))
res = " ".join(str_lst)
w = open("output.txt", 'w')
w.write(res)
w.close()
print("Время работы: %s секунд " % (time.perf_counter() - t_start))
print("Max memory ", tracemalloc.get_traced_memory()[1] / 2 ** 20, "mb")
tracemalloc.stop()

```

Текстовое объяснение решения.

Функция `merge_sort`: рекурсивная функция, которая разделяет массивы до тех пор, пока длинны не будут равны 1, далее функция `merge` которая занимается слиянием массивов. В цикле пока можно итерироваться по первому или второму массиву идет проверка и меньший элемент записывается в итоговый массив, прибавляем индекс проверки (i, j) в зависимости от результата.

Результат работы кода на максимальных и минимальных значениях:

1.py	3.py	4.py	input	output	5.py
1	1		✓	1	6738762
2	6738762				

1.py	3.py	4.py	input	output	5.py	6.py	7.py
1	200		✓	1	1000123	1012345	1090123 1100123 1123456
2	1234567 2345678 3456789 4567890 5678901 6789012 7890123						

	Время выполнения, с	Затраты памяти, Мб
Нижняя граница диапазона значений входных данных из текста задачи	0.0006214000168256462	0.017258644104003906
Верхняя граница диапазона значений входных данных из текста задачи	0.010489000007510185	0.03083038330078125

Вывод по задаче: Сортировка Merge-sort работает быстрее, чем сортировки вставкой, выбором и пузырьковая, но и памяти затрачивает больше.

3 задача. 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 отсортированном массиве число инверсий равно 0, а в массиве, отсортированном 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.

```
import time, tracemalloc
def merge_and_count(arr, temp_arr, left, right):
    if left == right:
        return 0
    mid = (left + right) // 2
    inv_count = 0
    inv_count += merge_and_count(arr, temp_arr, left, mid)
    inv_count += merge_and_count(arr, temp_arr, mid + 1, right)
```

```

    inv_count += merge(arr, temp_arr, left, mid, right)
    return inv_count
def merge(arr, temp_arr, left, mid, right):
    i = left
    j = mid + 1
    k = left
    inv_count = 0
    while i <= mid and j <= right:
        if arr[i] <= arr[j]:
            temp_arr[k] = arr[i]
            i += 1
        else:
            temp_arr[k] = arr[j]
            inv_count += (mid - i + 1)
            j += 1
        k += 1
    while i <= mid:
        temp_arr[k] = arr[i]
        i += 1
        k += 1
    while j <= right:
        temp_arr[k] = arr[j]
        j += 1
        k += 1
    for i in range(left, right + 1):
        arr[i] = temp_arr[i]
    return inv_count

def count_inversions(arr, n):
    temp_arr = [0] * n
    return merge_and_count(arr, temp_arr, 0, n - 1)

if __name__ == "__main__":
    with open("input", "r") as file:
        n = int(file.readline())
        arr = list(map(int, file.readline().split()))

    result = count_inversions(arr, n)

    with open("output", "w") as file:
        file.write(str(result) + "\n")

```

```

tracemalloc.start()
t_start = time.perf_counter()
print("Время работы: %s секунд " % (time.perf_counter() - t_start))
print("Max memory ", tracemalloc.get_traced_memory()[1] / 2 ** 20, "mb")
tracemalloc.stop()

```

Текстовое объяснение решения

merge_and_count: функция сливает два уже отсортированных подмассива исходного массива (arr) и подсчитывает количество инверсий между ними. merge_sort_and_count рекурсивно делит массив arr на две части, сортирует их с помощью себя же, а затем сливает обратно с помощью merge_and_count, подсчитывая инверсии на каждом шаге. Функция count_inversions инициализирует временный массив temp_arr и запускает рекурсивную сортировку слиянием для всего массива arr с помощью merge_sort_and_count.

Результат работы кода на примерах из текста задачи:

1.py	3.py	4.py	input ×	:	output ×	5.py	6.py
1	10		✓		1	17	
2	1 8 2 1 4 7 3 2 3 6				2		

Результат работы кода на максимальных и минимальных значениях :

1.py	3.py	4.py	input ×	:	output ×	5.py	6.py	7.py
1	1		✓		1	0		
2	346				2			

1.py	3.py	4.py	input ×	:	output ×	5.py	6.py	7.py
1	200		✓		1	1000123 1012345 1090123 1100123 1123456		
2	1234567 2345678 3456789 4567890 5678901 6789012 7890123							

	Время выполнения, с	Затраты памяти, Мб
Нижняя граница диапазона значений входных данных из текста задачи	0.001157199963927269	0.009146690368652344
Пример из задачи	0.0017660000594332814	0.009146690368652344

Верхняя граница диапазона значений входных данных из текста задачи	0.002080800011754036	0.009177207946777344
---	----------------------	----------------------

Вывод по задаче: Код эффективно подсчитывает инверсии в массиве с помощью алгоритма сортировки слиянием.

4 задача. 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.

- **Формат входного файла (input.txt).** В первой строке входного файла содержится число n ($1 \leq n \leq 10^5$) — число элементов в массиве, и последовательность $a_0 < a_1 < \dots < a_{n-1}$ из n **различных** положительных целых чисел в порядке возрастания, $1 \leq a_i \leq 10^9$ для всех $0 \leq i < n$. Следующая строка содержит число k , $1 \leq k \leq 10^5$ и k положительных целых чисел b_0, \dots, b_{k-1} , $1 \leq b_j \leq 10^9$ для всех $0 \leq j < k$.
- **Формат выходного файла (output.txt).** Для всех i от 0 до $k - 1$ вывести индекс $0 \leq j \leq n - 1$, такой что $a_i = b_j$ или -1, если такого числа в массиве нет.
- Ограничение по времени. 2сек.
- Ограничение по памяти. 256 мб.
- Пример:

input.txt	output.txt
5	2 0 -1 0 -1
1 5 8 12 13	
5	
8 1 23 1 11	

В этом примере есть возрастающая последовательность из $a_0 = 1, a_1 = 5, a_2 = 8, a_3 = 12$ и $a_4 = 13$ длиной в $n = 5$ и пять чисел для поиска: 8 1 23 1 11. Видно, что $a_2 = 8$ и $a_0 = 1$, но чисел 23 и 11 нет в последовательности a , поэтому они имеют индекс -1. В итоге ответ: 2 0 -1 0 -1.


```

import time, tracemalloc
def binary_search(a, s):
    lw = 0
    hg = len(a) - 1
    while lw <= hg:
        mid = (lw + hg) // 2
        if s == a[mid]:
            return mid
        elif s > a[mid]:
            lw = mid + 1
        else:
            hg = mid - 1
    return -1
def main():
    tracemalloc.start()
    t_start = time.perf_counter()
    with open("input") as f:
        n = int(f.readline())
        a = list(map(int, f.readline().split()))
        k = int(f.readline())
        b = list(map(int, f.readline().split()))
    lst = []
    for i in range(k):
        if time.perf_counter() - t_start > 2:
            lst = ["-1"] * k
            break
        lst.append(binary_search(a, b[i]))
    res = " ".join(map(str, lst))
    with open("output", 'w') as w:
        w.write(res)
    print("Время работы: %s секунд" % (time.perf_counter() - t_start))
    print("Max memory ", tracemalloc.get_traced_memory()[1] / 2 ** 20, "mb")
    tracemalloc.stop()
if __name__ == "__main__":
    main()

```

Текстовое объяснение решения.

Функция `binary_search`: обозначаем нижнюю, верхнюю границы и середину. Если искомое число совпадает с серединой, возвращаем индекс середины, если нет, то выполняем проверки больше или меньше наше искомое число середины, если больше, то нижняя граница становится середины + 1 (отрезаем половину чисел). Если меньше, то верхняя граница становится середина - 1. И так в цикле пока нижняя граница меньше или равна верхней. Если ничего в итоге не находим возвращаем -1

Результат работы кода на примерах из текста задачи:

```

1.py 3.py 4.py input x
1 5 ✓
2 1 5 8 12 13
3 5
4 8 1 23 1 11
output x 5.py
1 2 0 -1 0 -1

```

Результат работы кода на максимальных и минимальных значениях:

```

1.py 3.py 4.py input x
1 1 ✓
2 346
3 1
4 546
output x
1 -1

1.py 3.py 4.py input x
1 20 ✓
2 5921 8463 3715 1294 7086 9532 4178 6805 2349 1000 9987 3512
3 50
4 8576 2359 1279 4783 6501 9127 3265 5793 7211 9865 1438 6794
output x 5.py 6.py 7.py
1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

```

	Время выполнения, с	Затраты памяти, Мб
Нижняя граница диапазона значений входных данных из текста задачи	0.0013775000115856528	0.017258644104003906
Пример из задачи	0.008687100023962557	0.01728534698486328
Верхняя граница диапазона значений входных данных из текста задачи	0.007964499993249774	0.01848125457763672

Вывод по задаче: Бинарный поиск работает быстрее линейного, важно только не забыть отсортировать массив.

5 задача. Представитель большинства

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 $O(n \log n)$.

- **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 положительных целых чисел, по модулю не превосходящих 10^9 , $0 \leq a_i \leq 10^9$.
- **Формат выходного файла (output.txt).** Выведите 1, если во входной последовательности есть элемент, который встречается строго больше половины раз; в противном случае - 0.
- Ограничение по времени. 2сек.
- Ограничение по памяти. 256 мб.

```
import time, tracemalloc
def count_occurrences(arr, left, right, element):
    count = 0
    for i in range(left, right + 1):
        if arr[i] == element:
            count += 1
    return count
def majority_element_rec(arr, left, right):
    if left == right:
        return arr[left]

    mid = (left + right) // 2
    left_majority = majority_element_rec(arr, left, mid)
    right_majority = majority_element_rec(arr, mid + 1, right)

    if left_majority == right_majority:
        return left_majority

    left_count = count_occurrences(arr, left, right, left_majority)
    right_count = count_occurrences(arr, left, right, right_majority)

    return left_majority if left_count > right_count else right_majority
```

```

def has_majority_element(arr):
    n = len(arr)
    candidate = majority_element_rec(arr, 0, n - 1)
    count = count_occurrences(arr, 0, n - 1, candidate)
    if count > n // 2:
        return 1
    return 0

if __name__ == "__main__":
    with open("input", "r") as file:
        n = int(file.readline())
        arr = list(map(int, file.readline().split()))

    result = has_majority_element(arr)

    with open("output", "w") as file:
        file.write(str(result) + "\n")
tracemalloc.start()
t_start = time.perf_counter()
print("Время работы: %s секунд " % (time.perf_counter() - t_start))
print("Max memory ", tracemalloc.get_traced_memory()[1] / 2 ** 20, "mb")
tracemalloc.stop()

```

Текстовое объяснение решения:

majority_element_rec рекурсивно ищет кандидата на элемент большинства в массиве, деля его на половины и сравнивая результаты, далее count_occurrences подсчитывает количество вхождений элемента в массиве. Функция has_majority_element проверяет, является ли найденный кандидат действительно элементом большинства, сравнивая его количество вхождений с половиной размера массива.

Результат работы кода на примерах из текста задачи:

1.py	3.py	4.py	input ×	:	output ×	5.py	7.py
1	4		✓		1	0	
2	1 2 3 4				2		

1.py	3.py	4.py	input ×	:	output ×	5.py	
1	5		✓		1	1	
2	2 3 9 2 2				2		

	Время выполнения, с	Затраты памяти, Мб
Пример из задачи	6.700051017105579e-06	0.000396728515625
Пример из задачи	1.1399970389902592e-05	0.0003986358642578125

Вывод по задаче: Алгоритм “разделяй и властвуй” – это как когда ты пытаешься съесть огромный торт: сначала разрезаешь его на маленькие кусочки, а потом с наслаждением поглощаешь их по одному.

7 задача. Поиск максимального подмассива за линейное время

Можно найти максимальный подмассив за линейное время, воспользовавшись следующими идеями. Начните с левого конца массива и двигайтесь вправо, отслеживая найденный к данному моменту максимальный подмассив. Зная максимальный подмассив массива $A[1..j]$, распространите ответ на поиск максимального подмассива, заканчивающегося индексом $j + 1$, воспользовавшись следующим наблюдением: максимальный подмассив массива $A[1..j + 1]$ представляет собой либо максимальный подмассив массива $A[1..j]$, либо подмассив $A[i..j + 1]$ для некоторого $1 \leq i \leq j + 1$. Определите максимальный подмассив вида $A[i..j + 1]$ за константное время, зная максимальный подмассив, заканчивающийся индексом j .

В этом случае у вас возможны 2 варианта тестирования: первый предполагает создание случайного массива чисел, аналогично задаче №1 (в этом случае формат входного и выходного файла смотрите там). Второй вариант - взять любые данные по акциям какой-либо компании, аналогично задаче №6.

```
import time, tracemalloc
def search_max_subarray(lst):
    frst_ind = 0
    lst_ind = 0
    sm = 0
    mx_sum = -10**10
    bl = False
    for i in range(len(lst)):
        if sm > 0 and bl:
            frst_ind = i-1
            bl = False
            sm += lst[i]
        if mx_sum < sm:
            mx_sum = sm
            lst_ind = i
        if sm < 0:
```

```

    bl = True
    sm = 0
    if frst_ind > lst_ind:
        frst_ind = lst_ind
    return lst[frst_ind:lst_ind+1]
tracemalloc.start()
t_start = time.perf_counter()
f = open("input")
n = int(f.readline())
mas = [int(el) for el in f.readline().split()]
f.close()
str_lst = list(map(str, search_max_subarray(mas)))
res = " ".join(str_lst)
w = open("output", 'w')
w.write(res)
w.close()
print("Время работы: %s секунд " % (time.perf_counter() - t_start))
print("Max memory ", tracemalloc.get_traced_memory()[1] / 2 ** 20, "mb")
tracemalloc.stop()

```

Текстовое объяснение решения.

Функция `search_max_subarray`: переменная `frst_ind` отвечает за начальный индекс максимальной subarray, аналогично `lst_ind` – за конечный индекс. В каждой итерации обновляем переменную `sum` и проверяем, если она больше, чем максимальная сумма, то обновляем `mx_sum` и обновляем конечный индекс. В случае если переменная `sum` становится меньше нуля, обнуляем `sum` и итерируемся дальше, и когда `sum` становится снова больше 0, то обновляем значение `fst_ind`. Если в конце выполняется условие, что `frst_ind > lst_ind`, следовательно, subarray состоит из одного элемента.

Результат работы кода на максимальных и минимальных значениях:

1.py	3.py	4.py	input ×	:	output ×	5.py	7.py
1	1		✓		1	3596876	
2	-3596876						

.py ×	3.py	4.py	input ×	⌵	:	output ×	5.py	7.py
1	10		✓		1	10 7 -3 -12 15 18		
2	-10 10 7 -3 -12 15 18 -17 1 -2							

	Время выполнения, с	Затраты памяти, Мб
Нижняя граница диапазона значений входных данных из текста задачи	0.002014199970290065	0.01726055145263672
Верхняя граница диапазона значений входных данных из текста задачи	0.13007949996972457	2.064253807067871

Вывод по задаче: Научилась искать подмассив за линейное время. **Вывод:** Во время выполнения лабораторной работы я научилась писать merge sort, вспомнила алгоритм бинарного поиска. Сравнила алгоритм merge sort с другими алгоритмами сортировки.

Вывод

Во время выполнения лабораторной работы я научилась писать merge sort, вспомнила алгоритм бинарного поиска. Сравнила алгоритм merge sort с другими алгоритмами сортировки.