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

Вариант 15

Выполнил:
Скворцов Д.А.
К3140

Проверил:
Афанасьев А.В.

Санкт-Петербург
2024 г.

Содержание отчета

Задачи по варианту

Задача №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.

Листинг кода.

```
def merge(lst, l, m, r):
    n1 = m - l + 1
    n2 = r - m
    L = [0] * (n1)
    R = [0] * (n2)

    for i in range(0, n1):
        L[i] = lst[l + i]
    for j in range(0, n2):
        R[j] = lst[m + 1 + j]

    i, j = 0, 0
    k = l
```

```
    while i < n1 and j < n2:
        if L[i] <= R[j]:
            lst[k] = L[i]
            i += 1
        else:
            lst[k] = R[j]
            j += 1
        k += 1

    while i < n1:
        lst[k] = L[i]
        i += 1
        k += 1
    while j < n2:
        lst[k] = R[j]
        j += 1
        k += 1

def merge_sort(lst, l, r):
    if l < r:
        m = l + (r - l) // 2
        merge_sort(lst, l, m)
        merge_sort(lst, m + 1, r)
        merge(lst, l, m, r)

def main():
    with open('input.txt', 'r') as inp, open('output.txt', 'w') as out:
        n = int(inp.readline())
        lst = [int(x) for x in inp.readline().split()]
        merge_sort(lst, 0, n - 1)
        print(*lst, file=out, end='')
```

Алгоритм сортировки слиянием реализован двумя функциями: *merge_sort(), merge()*. Первая отвечает за разделение массива на части для сортировки и вызов второй функции, реализующей упорядоченное слияние двух данных ей частей массива.

Также для того, чтобы избавиться от добавления конечных элементов массивов *L, R,* используем 2 цикла *while,* перебирающих оставшуюся часть большего массива до конца.

```
10
1 8 4 2 3 7 5 6 9 0
```

```
1   0 1 2 3 4 5 6 7 8 9
```

```
.Тест 100 элементов:
Время работы: 0.002619400154799223 секунд
Затрачено памяти: 880 байт
Тест 1000 элементов:
Время работы: 0.07416059984825552 секунд
Затрачено памяти: 8288 байт
Тест 2*10e4 элементов:
Время работы: 2.1670756998937577 секунд
Затрачено памяти: 160288 байт
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.002619400154799223 секунд	880 байт
Медиана диапазона значений входных данных из текста задачи	0.07416059984825552 секунд	8288 байт
Верхняя граница диапазона значений входных данных из текста задачи	2.1670756998937577 секунд	160288 байт

Вывод по задаче:
Алгоритм реализован корректно и даже на данных порядка 10^4 за время, меньшее максимально заданного. По сравнению с алгоритмом сортировки вставками, рассматриваемый алгоритм работает за время порядка *n*log(n),* а не n^2 .

Задача №4. Бинарный поиск

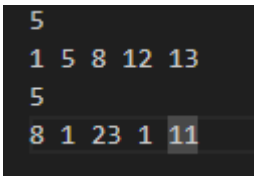
В этой задаче вы реализуете алгоритм бинарного поиска, который позволяет очень эффективно искать (даже в огромных) списках при условии, что список отсортирован. Цель - реализация алгоритма двоичного (бинарного) поиска.

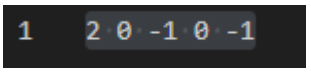
- **Формат входного файла (input.txt).** В первой строке входного файла содержится число n ($1 \le n \le 10^5$) — число элементов в массиве, и последовательность $a0 < a1 < ... < an-1$ из n различных положительных целых чисел в порядке возрастания, $1 \le ai \le 10^5$ для всех $0 \le i < n$. Следующая строка содержит число k, $1 \le k \le 10^5$ и k положительных целых чисел $b_0, ...b_{k-1}$, $1 \le b_j \le 10^9$ для всех $0 \le j < k$.
- **Формат выходного файла (output.txt).** Для всех i от 0 до k − 1 вывести индекс $0 \le j \le n - 1$, такой что ai = bj или -1, если такого числа в массиве нет.
- Ограничение по времени. 2сек.
- Ограничение по памяти. 256 мб.

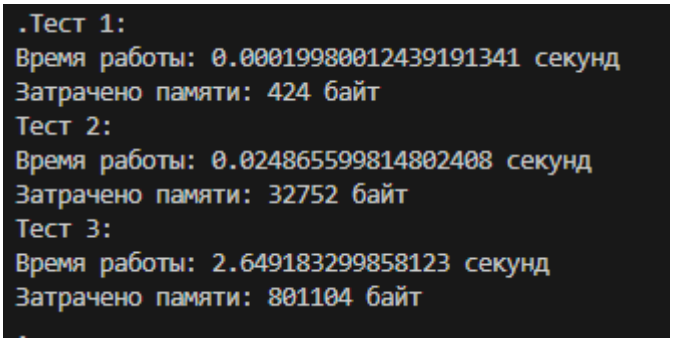
Листинг кода.

```
def bin_search(n, lst, x):
    low, mid, high = 0, 0, n - 1

    while low <= high:
        mid = (high + low) // 2
        if lst[mid] < x:
            low = mid + 1
        elif lst[mid] > x:
            high = mid - 1
        else:
            return mid
    return -1


def bin_search_loop(n, lst, values):
    res = []
    for v in values:
        res.append(bin_search(n, lst, v))
    return res

def main():
    with open('input.txt', 'r') as inp, open('output.txt', 'w') as out:
        n = int(inp.readline())
        lst = [int(x) for x in inp.readline().split()]
        _ = int(inp.readline())
        values = [int(x) for x in inp.readline().split()]
        ans = bin_search_loop(n, lst, values)
        print(*ans, file=out, end='')
```

Функция *bin_search()* возвращает индекс найденного элемента в отсортированном массиве, деля область поиска пополам, пока либо не находит искомый элемент, либо возвращает -1. Функция *bin_search_loop()* обрабатывает в цикле каждое значение, которое необходимо найти.







	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.00019980012439191341 секунд	424 байта
Медиана диапазона значений входных данных из текста задачи	0.024865599814802408 секунд	32752 байт
Верхняя граница диапазона значений входных данных из текста задачи	2.649183299858123 секунд	801104 байт

Вывод по задаче:
Алгоритм работает корректно: время выполнения в самом крайнем случае практически не выходит за рамки, но довольно велико, поскольку асимптотическая сложность алгоритма = $O(n*log(n))$. Тогда при $n = 10^5$ время выполнения слегка превышает 2 секунды.

Задача №6. Поиск максимальной прибыли

Используя псевдокод процедур Find Maximum Subarray и Find Max Crossing Subarray из презентации к Лекции 2 (страницы 25-26), напишите программу поиска максимального подмассива. Примените ваш алгоритм для ответа на следующий вопрос. Допустим, у нас есть данные по акциям какой-либо фирмы за последний месяц (год, или иной срок). 6 Проанализируйте этот срок и выдайте ответ, в какой из дней при покупке единицы акции данной фирмы, и в какой из дней продажи, вы бы получили максимальную прибыль? Выдайте дату покупки, дату продажи и максимальную прибыль. Вы можете использовать любые данные для своего анализа. Например, я набрала в Google "акции" и мне поиск выдал акции Газпрома, тут - можно скачать информацию по стоимости акций за любой период. (Перейдя по ссылке, нажмите на вкладку "Настройки"→ "Скачать") Соответственно, вам нужно только выбрать данные, посчитать изменение цены и применить алгоритм поиска максимального подмассива.

- **Формат входного файла** в данном случае на ваше усмотрение.
- **Формат выходного файла (output.txt).** Выведите название фирмы, рассматриваемый вами срок изменения акций, дату покупки и дату продажи единицы акции, чтобы получилась максимальная выгода; и сумма этой прибыли.

Листинг кода.

```
def find_max_crossing_subarray(lst, low, mid, high):
    left_sum = lst[mid]-1
    s = 0
    for i in range(mid, low-1, -1):
        s += lst[i]
        if s > left_sum:
            left_sum = s
            maxleft = i
    right_sum = lst[mid+1]-1
    s = 0
    for j in range(mid+1, high+1):
        s += lst[j]
        if s > right_sum:
            right_sum = s
            maxright = j
    return maxleft, maxright, left_sum + right_sum

def find_max_subarray(lst, low, high):
```

```
    if high == low:
        return low, high, lst[low]

    mid = (low + high) // 2
    left_low, left_high, left_sum = find_max_subarray(lst, low, mid)
      right_low, right_high, right_sum = find_max_subarray(lst, mid+1,
high)
     cross_low, cross_high, cross_sum = find_max_crossing_subarray(lst,
low, mid, high)
    if left_sum >= max(right_sum, cross_sum):
        return left_low, left_high, left_sum
    elif right_sum >= max(left_sum, cross_sum):
        return right_low, right_high, right_sum
    return cross_low, cross_high, cross_sum

def diff_array(n, lst):
    return [lst[i] - lst[i-1] for i in range(1, n)]


def solution(n, lst):
    diff_lst = diff_array(n, lst)
    low, high, profit = find_max_subarray(diff_lst, 0, (n - 1)-1)
    return low, high + 1, round(profit, 3)
```

Согласно написанным на псевдокоде алгоритмам нахождения максимального подмассива, пишем аналоги на python. При помощи рекурсивного разбиения массива задача сводится к более простой, пока не находится лучший результат и поднимается на предыдущий уровень рекурсии. Также добавляем функцию *diff_array()*, возвращающую изменения значений входного массива - относительно её результата будет работать основная функция. Функция *solution()* компонует ответ и возвращает значения для записи в выходной файл.

```
17
100 113 110 85 105 102 86 63 81 101 94 106 101 79 94 90 97
```

```
1 11 62.11
Buy: for 166.89 at 1, Sell: for 229.0 at 11. Profit: 62.11
```

```
.Тест 1:
Время работы: 0.00021919998107478023 секунд
Затрачено памяти: 424 байт
Тест 2:
Время работы: 0.02325090003432706 секунд
Затрачено памяти: 10160 байт
Тест 3:
Время работы: 0.3883563000126742 секунд
Затрачено памяти: 87056 байт
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.00021919998107478023 секунд	424 байт
Медиана диапазона значений входных данных из текста задачи	0.02325090003432706 секунд	10160 байт
Верхняя граница диапазона значений входных данных из текста задачи	0.3883563000126742 секунд	87056 байт

Вывод по задаче:
Алгоритм работает корректно: искомые номера элементов находятся, как и получаемая прибыль. Время выполнения порядка линейного. Память затрачивается на вызов рекурсии и создание большого массива.

Дополнительные задачи

Задача №7. 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.

Листинг кода.

```
def find_max_subarray(n, lst):
    prefix_sum = []
    s = 0
    for el in lst:
        s += el
        prefix_sum.append(s)

    min_i = 0
    j = 1
    max_subarr = (0, 0, lst[0])
    while j < n:
         cur_profit = prefix_sum[j] if min_i == 0 else prefix_sum[j] -
prefix_sum[min_i]
        if prefix_sum[j] < prefix_sum[min_i]:
            min_i = j
        elif cur_profit > max_subarr[2]:
                max_subarr = ((min_i+1 if min_i != 0 else min_i), j,
cur_profit)

        j += 1

    low, high, profit = max_subarr
    return low, high, profit
```

```
def diff_array(n, lst):
    return [lst[i] - lst[i-1] for i in range(1, n)]

def solution(n, lst):
    diff_lst = diff_array(n, lst)
    low, high, profit = find_max_subarray(n - 1, diff_lst)
    return low, high + 1, round(profit, 3)
```

Алгоритм запоминает в цикле текущую минимальную сумму слева, чтобы вычитать её из общей суммы и находить максимальную для *j*-го шага. Как только новая вычисляемая сумма становится больше текущей, мы запоминаем её. Остальное оформление задачи аналогично задаче №6.

```
17
100 113 110 85 105 102 86 63 81 101 94 106 101 79 94 90 97
```

```
7 11 43
```

```
178.85 166.89 181.82 197.96 181.18 169.0 178.87 183.38 193.12 221.55 207.23 229
```

```
1 11 62.11
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.0005110999918542802 секунд	419 байт
Медиана диапазона значений входных данных из текста задачи	0.0034465999924577773 секунд	41472 байт
Верхняя граница диапазона значений входных данных из текста задачи	0.053108900028746575 секунд	482112 байт

Вывод по задаче:
На тестах задачи № 6 проверена равносильность решений задачи. Этот алгоритм работает чуть быстрее, так как его время выполнения линейно,

память тратится на создания массива префиксных сумм, для того чтобы не создавать лишних переменных.

Задача №8. Умножение многочленов

Задача. Даны 2 многочлена порядка $n-1$: $a_{n-1}x^{n-1} + a_{n-2}x^{n-1} + ... + a_1x + a_0$ и $b_{n-1}x^{n-1} + b_{n-2}x^{n-1} + ... + b_1x + b_0$. Нужно получить произведение:

$$c_{2n-2}x^{2n-2} + c_{2n-3}x^{2n-3} + ... + c_1x + c_0, \text{ где:}$$

$$\begin{aligned} c_{2n-2} &= a_{n-1}b_{n-1} \\ c_{2n-3} &= a_{n-1}b_{n-2} + a_{n-2}b_{n-1} \\ ... &\quad ... \\ c_2 &= a_2b_0 + a_1b_1 + a_0b_2 \\ c_1 &= a_1b_0 + a_0b_1 \\ c_0 &= a_0b_0 \end{aligned}$$

- **Формат входного файла (input.txt).** В первой строке число n - порядок многочленов A и B. Во второй строке коэффициенты многочлена A через пробел. В третьей строке коэффициенты многочлена B через пробел.
- **Формат выходного файла (output.txt).** Ответ - одна строка, коэффициенты многочлена C(x) = A(x)B(x) через пробел.
- Нужно использовать метод "Разделяй и властвуй".

Листинг кода.

```
def sum_polynoms(A, B, shift_A=0, shift_B=0):
    len_r = max(len(A) + shift_A, len(B) + shift_B)
    R = [0] * len_r
    i = min(shift_A, shift_B)
    while i < len_r:
        if 0 <= i-shift_A < len(A):
            R[i] += A[i - shift_A]
        if 0 <= i-shift_B < len(B):
            R[i] += B[i - shift_B]
        i += 1
    return R

def multiply_polynoms(n, A, B):
```

```
    if n == 1:
        return [A[0] * B[0]]

    A0, A1 = A[:n//2], A[n//2:]
    B0, B1 = B[:n//2], B[n//2:]
    if len(A0) < len(A1):
        A0.append(0)
    if len(B0) < len(B1):
        B0.append(0)
    l = len(A0)
    U = multiply_polynoms(l, A0, B0)
    V = multiply_polynoms(l, A0, B1)
    W = multiply_polynoms(l, A1, B0)
    Z = multiply_polynoms(l, A1, B1)

    VW = sum_polynoms(V, W, n//2, n//2)
    UZ = sum_polynoms(U, Z, shift_B=2*(n//2))
    R = sum_polynoms(VW, UZ)
    return R
```

Помимо основной функции умножения, работающей по принципу разделения пополам многочлена вида $a_0 + a_1x + ... + a_n x^{n-1}$, реализована функция сложения многочленов с возможностью домножения на x^k путем сдвига массива и добавления нулей.

```
1   3
2   3 2 5
3   5 1 2
```

```
1   15 13 33 9 10
```

```
1   5
2   2 5 3 1 -1
3   1 2 2 3 6
```

```
1   2 9 17 23 34 39 19 3 -6
```

	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.0003021999727934599 секунд	443 байт
Медиана диапазона значений входных	0.13986719999229535 секунд	9248 байт

данных из текста задачи		
Верхняя граница диапазона значений входных данных из текста задачи	2.3203765000216663 секунд	46068 байт

Вывод по задаче:

Время выполнения алгоритма чуть быстрее квадратичного, память практически не затрачивается. На тестах была проверена корректность выполнения, включая крайние случаи.

Вывод

Рекурсивные алгоритмы, основанные на принципе “Разделяй и властвуй” можно использовать во многих задачах, легко задающихся правилам упрощения. Их плюсом является простота написания, при обозначенных правилах, но в случаях с большими значениями и глубокой рекурсией может потребоваться значительное количество памяти.