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 слиянием. Метод декомпозиции

Вариант 18

Выполнила:

Петрова М. В

К3139

Санкт-Петербург

2024 г.

**Содержание отчета**

[**Задачи по варианту 3**](#_30j0zll)

[Задача №1. Сортировка слиянием 3](#_g1ywmykqlm9v)

[Задача №4. Бинарный поиск 6](#_8aj44myebt7b)

[Задача №9. Метод Штрассена для умножения матриц 8](#_tuthsv7arq2z)

[**Дополнительные задачи 11**](#_d6vwrp6jcjiz)

[Задача №3. Число Инверсий 11](#_7818wqlbnley)

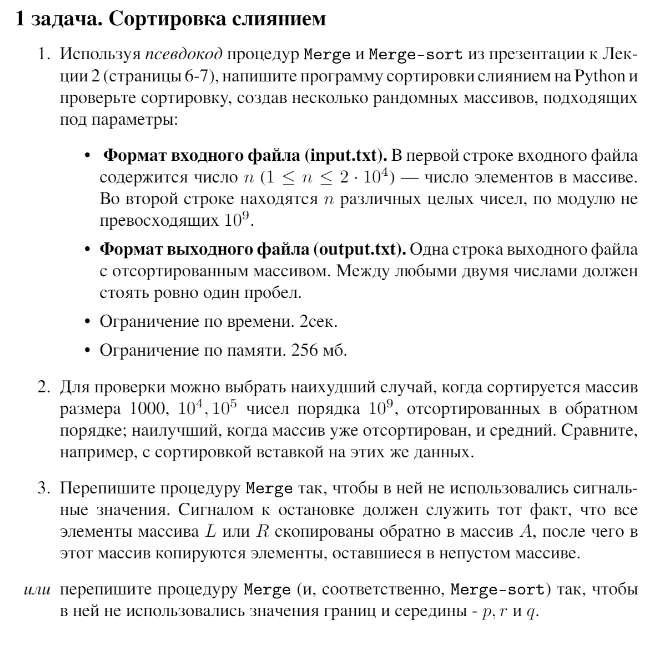
[Задача №5. Представитель большинства 13](#_eh1hailc2nus)

[Задача №7. Поиск максимального подмассива 16](#_rk8l5rh9j0l4)

[**Вывод по задаче 18**](#_81qwwlptezf7)

# **Задачи по варианту**

## Задача №1. Сортировка слиянием



import time

start = time.perf\_counter()

def merge(arr, l, m, r):

L = arr[l:m + 1]

R = arr[m + 1:r + 1]

i, j, k = 0, 0, l

while i < len(L) and j < len(R):

if L[i] <= R[j]:

arr[k] = L[i]

i += 1

else:

arr[k] = R[j]

j += 1

k += 1

while i < len(L):

arr[k] = L[i]

i += 1

k += 1

while j < len(R):

arr[k] = R[j]

j += 1

k += 1

def merge\_sort(arr, p, r):

if p < r:

q = (p + r) // 2

merge\_sort(arr, p, q)

merge\_sort(arr, q + 1, r)

merge(arr, p, q, r)

Текстовое описание решения

1. Функция merge(arr, l, m, r):

1.1 Выполняет слияние двух отсортированных подмассивов: Первый подмассив от индекса l до m. Второй подмассив от индекса m + 1 до r.

1.2 Создаются временные массивы L и R для хранения значений из этих подмассивов.

1.3 Элементы из массивов L и R сливаются в исходный массив arr в правильном порядке.

1. Функцияmerge\_sort(arr, p, r):

2.1 Рекурсивно делит массив на две части:

2.2 После рекурсивной сортировки левой и правой частей массивы сливаются с помощью функции merge.

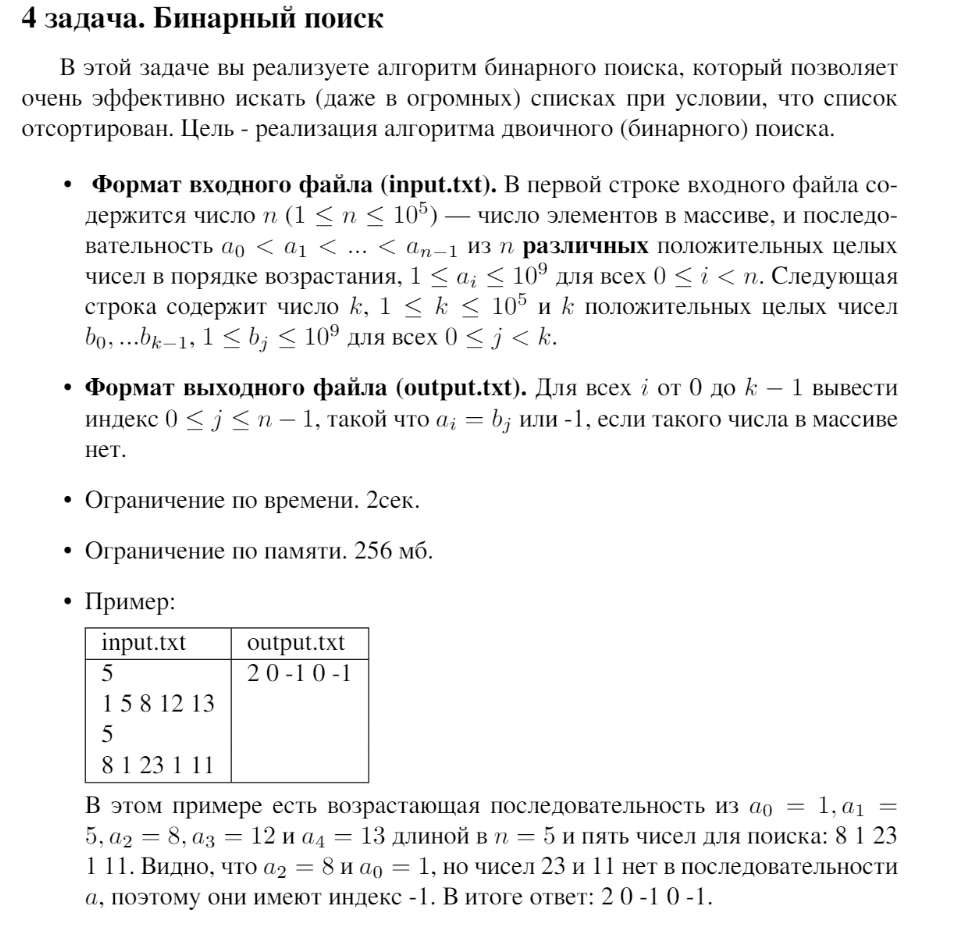
1. Данные считываются из файла input и записываются в файл output

| Тест | Время выполнения микросек |
| --- | --- |
| Массив размером 1000 | 0.002960400000000002 ms |
| Массив размером 10\*\*4 | 0.029551999999999995 ms |
| Массив размером 10\*\*5 | 0.4328012 ms |

Вывод по задаче:

Этот код реализует алгоритм сортировки слиянием (Merge Sort). Алгоритм делит массив на две части, рекурсивно сортирует каждую из них, а затем сливает две отсортированные части в один отсортированный массив.

## Задача №4. Бинарный поиск



def bin(arr, l, r, v):

if l > r:

return -1

m = (l + r) // 2

if v == arr[m]:

return m

elif v < arr[m]:

return bin(arr, l, m - 1, v)

return bin(arr, m + 1, r, v)

with open('../tests/input.txt') as f:

n = int(f.readline())

sort\_arr = list(map(int, f.readline().split()))

k = int(f.readline())

choice\_arr = list(map(int, f.readline().split()))

ans = []

for el in choice\_arr:

index = bin(sort\_arr, 0, len(sort\_arr) - 1, el)

ans.append(index)

with open('../tests/output.txt', 'w') as f:

f.write(' '.join(map(str, ans)))

Текстовое описание решения

1. Функция бинарного поиска bin:

1.1 Функция реализована рекурсивно. Она ищет элемент v в отсортированном массиве arr между индексами l и r.

1.2 Если левая граница l больше правой r, то возвращаем -1

1.3 Вычисляем середину массива m.

1.4 Если элемент в середине равен искомому значению v, возвращаем его индекс m.

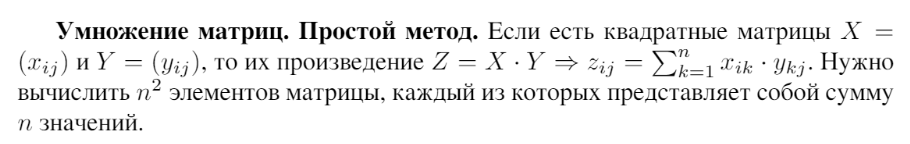
1.5 Если искомый элемент меньше среднего значения, рекурсивно продолжаем поиск в левой части массива. Иначе рекурсивно продолжаем поиск в правой части массива.

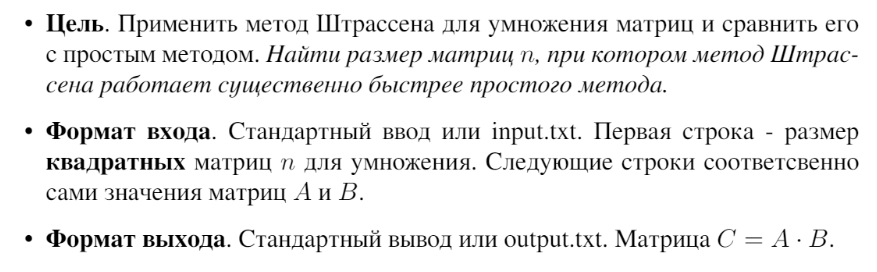
1. Считываем данные из файла
2. вычисляем результат выполнения программы
3. Записываем данные в файл output

**Вывод по задаче:**

Задача выполняет бинарный поиск для нахождения индексов элементов из одного массива в другом отсортированном массиве. Для каждого элемента из массива запросов осуществляется поиск его позиции в отсортированном массиве с использованием рекурсивной функции бинарного поиска. Если элемент найден — возвращается его индекс, если нет — возвращается -1. Результаты записываются в файл.

## Задача №9. Метод Штрассена для умножения матриц





import time

start = time.perf\_counter()

def split(matrix):

n = len(matrix)

mid = n // 2

A11 = [[matrix[i][j] for j in range(mid)] for i in range(mid)]

A12 = [[matrix[i][j] for j in range(mid, n)] for i in range(mid)]

A21 = [[matrix[i][j] for j in range(mid)] for i in range(mid, n)]

A22 = [[matrix[i][j] for j in range(mid, n)] for i in range(mid, n)]

return A11, A12, A21, A22

def add(A, B):

n = len(A)

return [[A[i][j] + B[i][j] for j in range(n)] for i in range(n)]

def sub(A, B):

n = len(A)

return [[A[i][j] - B[i][j] for j in range(n)] for i in range(n)]

def strassen(A, B):

n = len(A)

if n == 1:

return [[A[0][0] \* B[0][0]]]

A11, A12, A21, A22 = split(A)

B11, B12, B21, B22 = split(B)

P1 = strassen(A11, sub(B12, B22))

P2 = strassen(add(A11, A12), B22)

P3 = strassen(add(A21, A22), B11)

P4 = strassen(A22, sub(B21, B11))

P5 = strassen(add(A11, A22), add(B11, B22))

P6 = strassen(sub(A12, A22), add(B21, B22))

P7 = strassen(sub(A11, A21), add(B11, B12))

C11 = add(sub(add(P5, P4), P2), P6)

C12 = add(P1, P2)

C21 = add(P3, P4)

C22 = sub(sub(add(P5, P1), P3), P7)

new\_matrix = [[0 for \_ in range(n)] for \_ in range(n)]

for i in range(n // 2):

for j in range(n // 2):

new\_matrix[i][j] = C11[i][j]

new\_matrix[i][j + n // 2] = C12[i][j]

new\_matrix[i + n // 2][j] = C21[i][j]

new\_matrix[i + n // 2][j + n // 2] = C22[i][j]

return new\_matrix

Текстовое описание решения

1. Функция split Разбивает матрицу на четыре подматрицы (A11, A12, A21, A22) размером n/2.
2. Функции add и sub: Выполняют поэлементное сложение и вычитание двух матриц.
3. Функция strassen:

3.1 Рекурсивно умножает две матрицы по алгоритму

Штрассена.

3.2 Если матрицы размером 1×11 \times 11×1, просто умножает их элементы.

3.3 В остальных случаях:

Разделяет матрицы A и B на подматрицы. Далее вычисляет семь вспомогательных матриц P1,P2,...,P7 согласно формулам Штрассена. И комбинирует эти вспомогательные матрицы для получения результата.

3. 4 Собирает конечную матрицу C из четырех подматриц C11,C12,C21,C22, которые являются результатом умножения.

1. Считываем данные из файла
2. Вычисляем результат программы и записываем данные в файл output

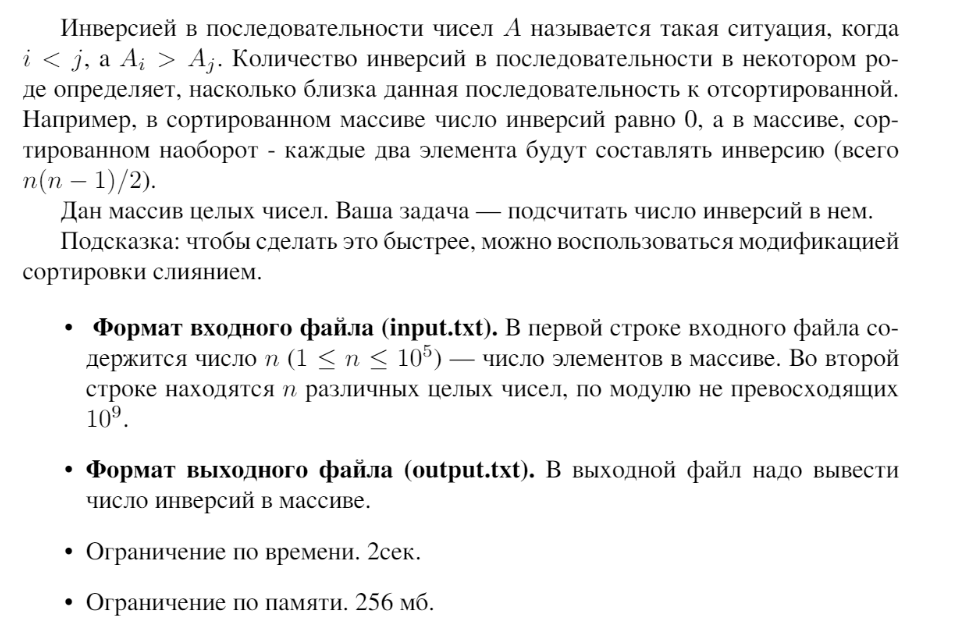
Вывод по задаче:

В данной задаче реализован алгоритм умножения квадратных матриц с использованием метода Штрассена. Этот алгоритм оптимизирует процесс умножения, снижая вычислительную сложность за счёт рекурсивного разбиения матриц на подматрицы и использования семи вспомогательных произведений.

| Тест | Время выполнения микросек |
| --- | --- |
| Размер матрицы 2\*2 | 0.0014596000000000053 |
| Размер матрицы 4\*4 | 0.00163859999999999 |

# **Дополнительные задачи**

## Задача №3. Число Инверсий



def count\_and\_merge(arr, p, q, r):

n1 = q - p + 1

n2 = r - q

L = arr[p:q+1]

R = arr[q + 1: r + 1]

res = 0

i, j, k = 0, 0, p

while i < n1 and j < n2:

if L[i] <= R[j]:

arr[k] = L[i]

i += 1

else:

arr[k] = R[j]

j += 1

res += n1 - i

k += 1

while i < n1:

arr[k] = L[i]

i += 1

k += 1

while j < n2:

arr[k] = R[j]

j += 1

k += 1

return res

def count\_inv(arr, l, r):

res = 0

if l < r:

q = (r + l) // 2

res += count\_inv(arr, l, q)

res += count\_inv(arr, q + 1, r)

res += count\_and\_merge(arr, l, q, r)

return res

1. Функция count\_and\_merge:

1.1 Делит массив на два временных подмассива L и R Использует два указателя для сравнения элементов подмассивов:

1.2 Если элемент из левого подмассива меньше либо равен элементу из правого, он копируется в исходный массив.

1.3 Если элемент из правого подмассива меньше, это инверсия, и она подсчитывается.

1.4 Оставшиеся элементы из обоих подмассивов добавляются в исходный массив.

1.5 Возвращает количество найденных инверсий.

1. Функция count\_inv

2.1 Это рекурсивная часть алгоритма, аналогичная сортировке слиянием.

2.2 Разбивает массив на две части до тех пор, пока в каждой части не останется по одному элементу.

2.3 После этого рекурсивно сливает подмассивы и подсчитывает инверсии с помощью функции count\_and\_merge.

2.4 Результат — общее количество инверсий в массиве.

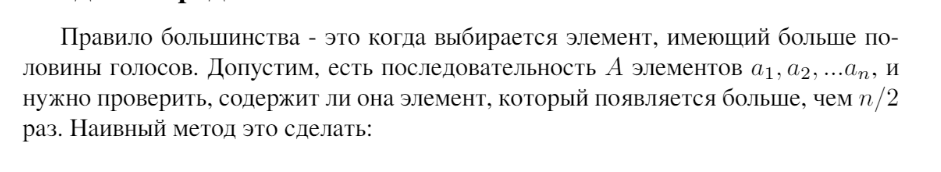
1. считываем данные из файлов
2. вычисляем результат и записываем в файл output

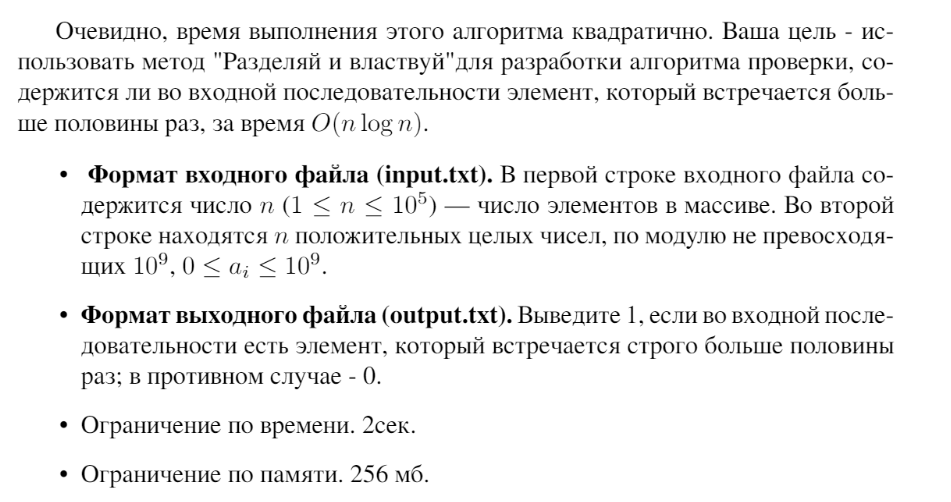
| Тест | Время выполнения микросек |
| --- | --- |
| 5  1 2 3 4 5 | 2.369999999998762e-05 ms |
| 5  5 4 3 2 1 | 2.140000000000475e-05 ms |

Вывод по задаче:

Этот код реализует алгоритм подсчёта инверсий в массиве с помощью модифицированной сортировки слиянием.

## Задача №5. Представитель большинства





def find\_majority(arr, l, r):

if l == r:

return arr[l]

m = (l+ r) // 2

l\_majority = find\_majority(arr, l, m)

r\_majority = find\_majority(arr, m + 1, r)

if l\_majority == r\_majority:

return l\_majority

l\_count = sum(1 for i in range(l, r + 1) if arr[i] == l\_majority)

r\_count = sum(1 for i in range(l, r + 1) if arr[i] == r\_majority)

return l\_majority if l\_count > r\_count else r\_majority

def majority\_element(arr):

cand = find\_majority(arr, 0, len(arr) - 1)

count = sum(1 for num in arr if num == cand)

if count > len(arr) // 2:

return 1

else:

return 0

Текстовое описание решения

1. Функция find\_majority:

1.1 Это рекурсивная функция, которая применяет метод "разделяй и властвуй" для поиска кандидата на элемент большинства в массиве.

1.2 Базовый случай: Если рассматриваемый подмассив состоит из одного элемента, этот элемент возвращается как кандидат.

1.3 Рекурсивное деление: Массив делится на две части, и для каждой части рекурсивно ищется кандидат на элемент большинства.

1.4 Сравнение кандидатов: Если кандидаты для левой и правой частей совпадают, возвращается этот кандидат. Иначе подсчитывается количество вхождений каждого кандидата в текущем диапазоне и возвращается тот, который встречается чаще.

1. Функция majority\_element:

2.1 Вызывает find\_majority для поиска кандидата на элемент большинства.

2. 2 Подсчитывает количество вхождений найденного кандидата в массиве.

2.3 Если кандидат встречается больше чем n/2 раз, возвращается 1 (элемент большинства найден), иначе 0.

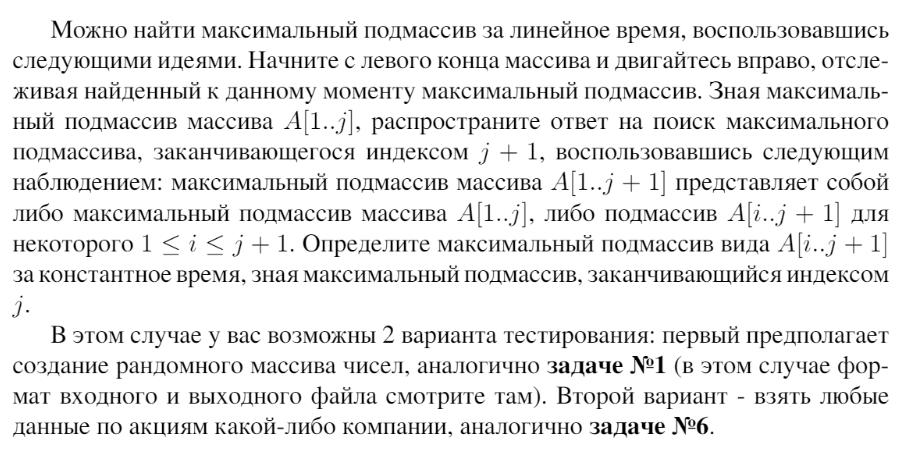
1. Считываются данные из файлв
2. Вычисляется результат выполнения программы и записывается в файл output

| Тест | Время выполнения микросек |
| --- | --- |
| 5  2 3 9 2 2 | 1.180000000000625e-05 ms |
| 4  1 2 3 4 | 1.0999999999997123e-05 ms |

Вывод по задаче:

В данной задаче реализован алгоритм поиска элемента большинства с использованием метода "разделяй и властвуй". Алгоритм эффективно находит кандидата на элемент большинства, а затем проверяет, встречается ли он более чем в половине массива.

## Задача №7. Поиск максимального подмассива



import time

start = time.perf\_counter()

def max\_subarray(arr):

max\_current = maxx = arr[0]

for i in range(1, len(arr)):

max\_current = max(arr[i], max\_current + arr[i])

if max\_current > maxx:

maxx = max\_current

return maxx

with open('../tests/input.txt') as f:

arr = list(map(int, f.readline().split()))

ans = max\_subarray(arr)

with open('../tests/output.txt', 'w') as f:

f.write(str(ans))

stop = time.perf\_counter()

print("time: %s ms" % (stop - start))

**текстовое объяснение**

1. Функция max\_subarray:

1.1 Реализуется алгоритм Кадане для поиска максимальной суммы подмассива в массиве.

**1.2** Для каждого элемента массива вычисляется максимальная сумма подмассива, который включает текущий элемент: либо текущий элемент сам по себе, либо текущий элемент плюс предыдущая максимальная сумма.

1.3 Если текущая сумма подмассива (max\_current) больше глобальной максимальной суммы (maxx), обновляем глобальную максимальную сумму.

1. Считываем данные из файла
2. Вычисляем результат работы программы и записываем в файл output
3. Выводим время выполнения программы

| Тест | Время выполнения микросек |
| --- | --- |
| 10000  -1, -1 … -1, 10000 | 0.0011578000000000005 ms |
| 10000  1, 2, … 10000 | 0.002555799999999997 ms |
|  |  |

## Вывод по задаче

В данной задаче реализован алгоритм Кадане для нахождения максимальной суммы непрерывного подмассива. Программа считывает массив из файла, вычисляет максимальную сумму подмассива и записывает результат в выходной файл.

Вывод по лабораторной

Все задачи были решены с использованием эффективных алгоритмов, что позволило оптимизировать время выполнения даже для больших входных данных. Каждая из решённых задач способствует пониманию и применению различных алгоритмов сортировки и поиска.