

Министерство образования и науки РФ  
Федеральное государственное автономное образовательное учреждение высшего образования  
«Омский государственный технический университет»

Факультет (институт) Информационных технологий и компьютерных систем  
Кафедра Прикладная математика и фундаментальная информатика

### Расчетно-графическая работа

по дисциплине Алгоритмы и анализ сложности  
на тему Анализ сложности алгоритмов

Пояснительная записка

Шифр проекта 020-РГР-02.03.02-№ 4-ПЗ


Студента Добрянского Андрея Сергеевича  
фамилия, имя, отчество полностью

Курс 3 Группа ФИТ-212

Направление (специальность) 02.03.02  
Фундаментальная информатика и информационные технологии  
код, наименование

Руководитель ст. преподаватель  
ученая степень, звание

Федотова И.В.  
фамилия, инициалы

Выполнил 13.06.2024,   
дата, подпись студента

13.06.2024,   
дата, подпись руководителя

Омск 2024

## СОДЕРЖАНИЕ

СОДЕРЖАНИЕ .....	2
ЛАБОРАТОРНАЯ РАБОТА 1 (Гномья и пузырьковая сортировка) .....	3
ЛАБОРАТОРНАЯ РАБОТА 2 (Фибоначчи) .....	7
ЛАБОРАТОРНАЯ РАБОТА 3 (Ход коня).....	9
ЛАБОРАТОРНАЯ РАБОТА 4 (Генерация перестановок) .....	12
ЛАБОРАТОРНАЯ РАБОТА 5 (Генерация перестановок) .....	18
ЛАБОРАТОРНАЯ РАБОТА 6 (Алгоритмы работы со строками).....	15
ЛАБОРАТОРНАЯ РАБОТА 7 (Жадные алгоритмы).....	20
ЛАБОРАТОРНАЯ РАБОТА 8 (Жадные алгоритмы).....	24
ЛАБОРАТОРНАЯ РАБОТА 9 (Жадные алгоритмы).....	27
ЛАБОРАТОРНАЯ РАБОТА 10 (Раскраска графа).....	30
ЛАБОРАТОРНАЯ РАБОТА 11 (Фано) .....	33
ЛАБОРАТОРНАЯ РАБОТА 12 (Оператор мобильной связи).....	36
СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ.....	38

# ЛАБОРАТОРНАЯ РАБОТА 1 (Гномья и пузырьковая сортировка)

## 1.1. Постановка задачи

Выполнить сравнительный анализ временной и количественной зависимости от входных данных работы 2-х алгоритмов: гномья сортировка и пузырьковая.

## 1.2. Код реализации

Программная реализация алгоритма гномьей сортировки будет иметь вид:

```
def gnome_sort(nums):
    i = 0
    id = 1
    count = 2
    while i < len(nums):
        count += 2
        if i == 0 or nums[i] >= nums[i - 1]:
            i = id
            id += 1
        else:
            nums[i], nums[i - 1] = nums[i - 1], nums[i]
            i -= 1
        count += 2
    return nums, count
```

Программная реализация алгоритма пузырьковой сортировки будет иметь вид:

```
def bubble_sort(nums):
    s = len(nums)
    count = 1
    for i in range(s):
        count += 1
        for j in range(s - 1):
            count += 2
            if nums[j] > nums[j + 1]:
                nums[j], nums[j + 1] = nums[j + 1], nums[j]
                count += 1
    return nums, count
```

Программная реализация вычисления времени выполнения и количества операций каждого из алгоритмов имеет вид:

```
import random
import time

def measure_sorting_time(arr1, arr2):
    start_time = time.time()
    G, G_count = gnome_sort(arr1)
    G_sort_time = time.time() - start_time

    start_time = time.time()
    B, B_count = bubble_sort(arr2)
    B_sort_time = time.time() - start_time

    return G_sort_time, G_count, B_sort_time, B_count

sizes = [_ for _ in range(100, 10001, 100)]

G_sort_times = []
B_sort_times = []
G_num_operations = []
B_num_operations = []
for size in sizes:
    arr1 = [random.randint(0, 99) for _ in range(size)]
    arr2 = list(arr1)
    G_sort_time, G_count, B_sort_time, B_count = measure_sorting_time(arr1,
arr2)
    G_sort_times.append(G_sort_time)
    B_sort_times.append(B_sort_time)
    G_num_operations.append(G_count)
    B_num_operations.append(B_count)
    print(f'{size} элементов')
    print(f'Гномья сортировка: {G_sort_time} сек, кол-во операций:
{G_count}')
    print(f'Пузырьковая сортировка: {B_sort_time} сек, кол-во операций:
{B_count}\n')
```

Программная реализация построения графиков временного и количественного анализов имеет вид:

```
plt.figure(figsize=(12, 6))
```

```
plt.subplot(1, 2, 1)
plt.plot(sizes, G_sort_times, marker='o', linestyle='-', color='g',
label='Гномья сортировка')
plt.plot(sizes, B_sort_times, marker='s', linestyle='--', color='r',
label='Пузырьковая сортировка')
plt.title('Временной анализ')
plt.xlabel('Размер входных данных')
plt.ylabel('Время выполнения(в секундах)')
plt.legend()
```

```
plt.subplot(1, 2, 2)
```

```
plt.plot(sizes, G_num_operations, marker='o', linestyle='-', color='g',
label='Гномья сортировка')
plt.plot(sizes, B_num_operations, marker='s', linestyle='--', color='r',
label='Пузырьковая сортировка')
plt.title('Количественный анализ')
plt.xlabel('Размер входных данных')
plt.ylabel('Количество операций')
plt.legend()
```

```
plt.tight_layout()
plt.show()
```

### 1.3. Оценка сложности реализации

Временной и количественный анализ для алгоритмов сортировки имеет вид, изображенный на рисунке 1:

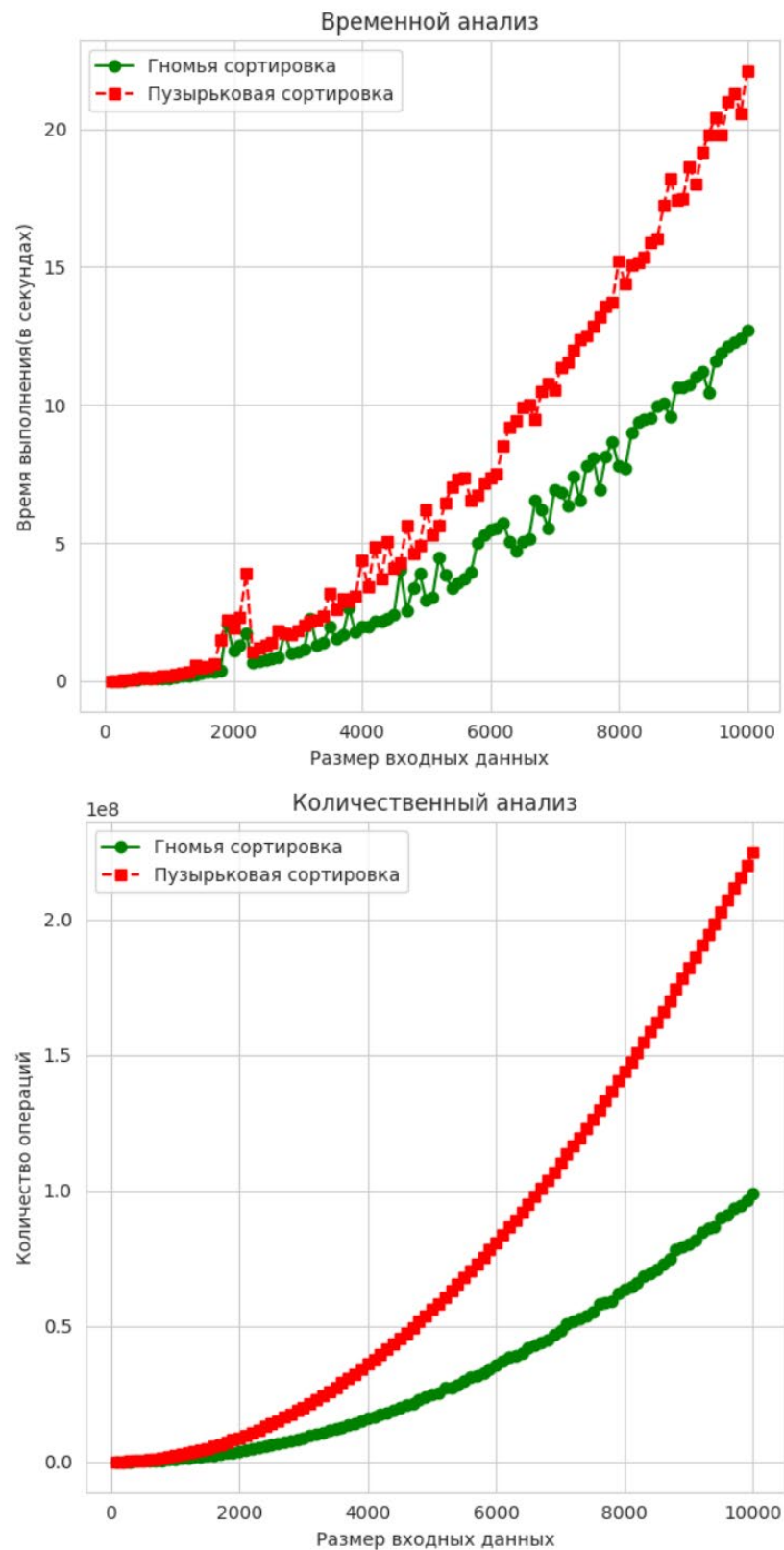


Рисунок 1 – Временной и количественный анализ

## ЛАБОРАТОРНАЯ РАБОТА 2 (Фибоначчи)

### 2.1. Постановка задачи

Выполнить анализ временной зависимости определения n-го элемента последовательности Фибоначчи на основе рекурсивного и итерационного процессов.

### 2.2. Код реализации

Программная реализация алгоритма рекурсии Фибоначчи будет иметь вид:

```
def Fibonacci(n):  
    if n == 1 or n == 2:  
        return 1  
    else:  
        return Fibonacci(n - 1) + Fibonacci(n - 2)
```

Программная реализация алгоритма итерации Фибоначчи будет иметь вид:

```
def Iter_Fibonacci(a):  
    f1, f2 = 1, 1  
    for i in range(2, a):  
        f1, f2 = f2, f2 + f1  
    return f2
```

Программная реализация вычисления времени выполнения каждого из алгоритмов имеет вид:

```
import time  
  
Times_Rec = []  
Times_Iter = []  
  
for i in range(1, 40):  
    Start_Rec = time.time()  
    Fib_Rec = Fibonacci(i)  
    Time_Rec = time.time() - Start_Rec  
    Times_Rec.append(Time_Rec)  
  
    Start_Iter = time.time()  
    Fib_Iter = Iter_Fibonacci(i)  
    Time_Iter = time.time() - Start_Iter  
    Times_Iter.append(Time_Iter)  
  
print(f'{i} - Рек: {Times_Rec} Итер: {Times_Iter}')
```

Программная реализация построения графика временного анализа имеет

ВИД:

```
import matplotlib.pyplot as plt

i = range(1, 40)

plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.plot(i, Times_Rec, label='Рекурсия', color='b')
plt.plot(i, Times_Iter, label='Итерация', color='r')
plt.xlabel('Число Фибоначчи')
plt.ylabel('Время в секундах')
plt.title('Временная зависимость')
plt.legend()

plt.tight_layout()
plt.show()
```

### 2.3. Оценка сложности реализации

Временной анализ для алгоритмов имеет вид, изображенный на рисунке 1:

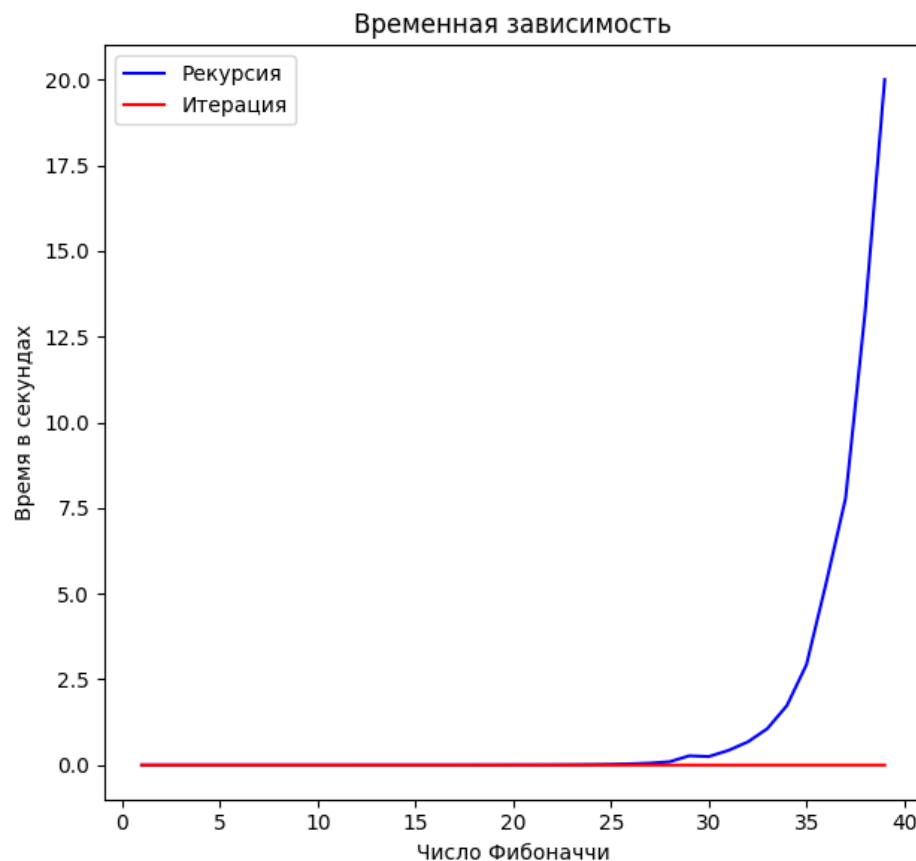


Рисунок 2 – Анализ времени получения чисел Фибоначчи



## ЛАБОРАТОРНАЯ РАБОТА 3 (Ход коня)

### 3.1. Постановка задачи

Выполнить временной анализ от размерности игрового поля алгоритма заполнения шахматной доски размерностью  $n \times n$  значениями от 1 до  $n^2$  с использованием хода коня. Если не выполняется (при определенных  $n$ ), то выдать сообщение (до 100-200).

### 3.2. Код реализации

Программная реализация алгоритма заполнения шахматной доски размерностью  $n \times n$  значениями от 1 до  $n^2$  с использованием хода коня будет иметь вид:

```
import time

def is_valid(doska, pos, n):
    return (0 <= pos[0] < n and 0 <= pos[1] < n and doska[pos[0]][pos[1]] == -1)

def knight_tour(n):
    doska = [[-1 for _ in range(n)] for _ in range(n)]
    moves = [(2, 1), (1, 2), (-1, 2), (-2, 1), (-2, -1), (-1, -2), (1, -2), (2, -1)]
    doska[0][0] = 0
    if not tour(doska, 0, 0, 1, moves, n):
        print(f"Решение не существует для доски размером {n}x{n}")
    else:
        for row in doska:
            print(row)
        print(f"Решение найдено для доски размером {n}x{n}")

def tour(doska, x, y, i, moves, n):
    if i == n * n:
        return True
    for move in moves:
        nx, ny = x + move[0], y + move[1]
        if is_valid(doska, (nx, ny), n):
            doska[nx][ny] = i
            if tour(doska, nx, ny, i + 1, moves, n):
                return True
            doska[nx][ny] = -1
    return False

Times = []
for n in range(1, 9):
    start_time = time.time()
```

```

knight_tour(n)
end_time = time.time()
Time = end_time - start_time
Times.append(Time)
print(f"Время выполнения для доски размером {n}x{n}: {Time} секунд\n")

```

Программная реализация вычисления времени выполнения алгоритма имеет вид:

```

import matplotlib.pyplot as plt

i = range(1, 9)

plt.figure(figsize=(7, 4))

plt.plot(i, Times, color='b')
plt.xlabel('n размер доски')
plt.ylabel('Время в секундах')
plt.title('Временная зависимость')
plt.legend()

plt.tight_layout()
plt.show()

```

Пример вывода:

Решение не существует для доски размером 4x4

Время выполнения для доски размером 4x4: 0.012833356857299805 секунд

```

[0, 59, 38, 33, 30, 17, 8, 63]
[37, 34, 31, 60, 9, 62, 29, 16]
[58, 1, 36, 39, 32, 27, 18, 7]
[35, 48, 41, 26, 61, 10, 15, 28]
[42, 57, 2, 49, 40, 23, 6, 19]
[47, 50, 45, 54, 25, 20, 11, 14]
[56, 43, 52, 3, 22, 13, 24, 5]
[51, 46, 55, 44, 53, 4, 21, 12]

```

Решение найдено для доски размером 8x8

Время выполнения для доски размером 8x8: 33.29193711280823 секунд

### 3.3. Оценка сложности реализации

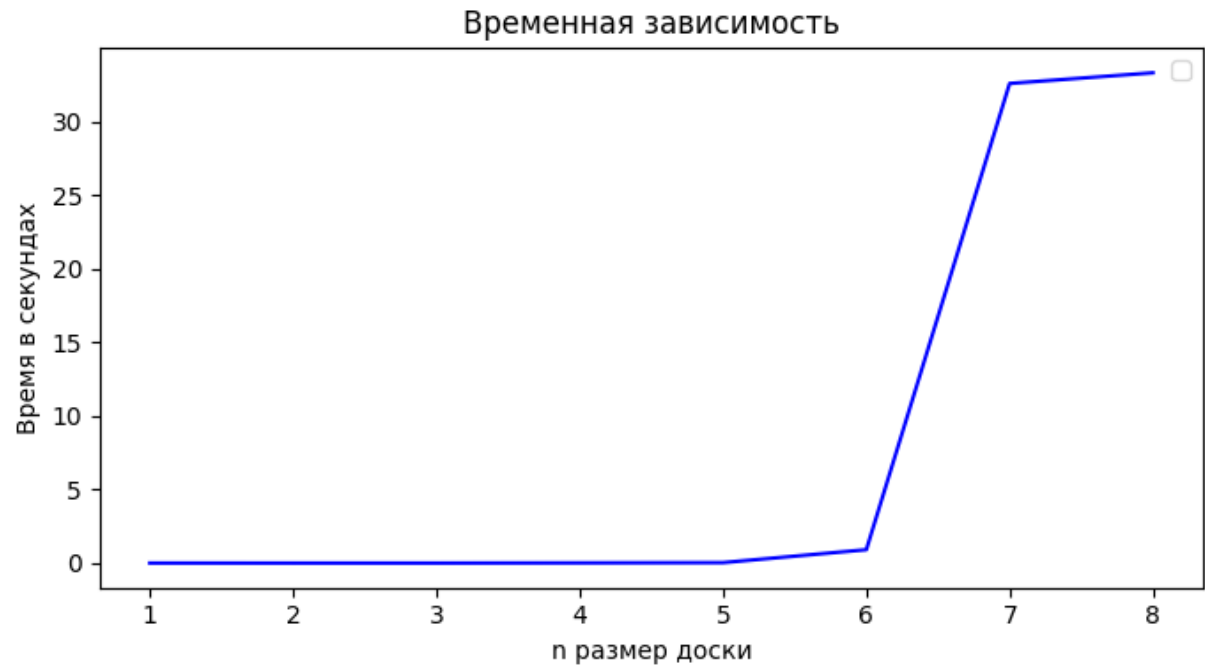


Рисунок 3 – Анализ времени получения заполнения шахматной доски.

## ЛАБОРАТОРНАЯ РАБОТА 4 (Генерация перестановок)

### 4.1.1 Постановка задачи

Сгенерировать все возможные перестановки элементов  $1, \dots, n$ .  
Использовать алгоритм Нарайана, Джонсона-Троттера, вектора инверсий.  
Выполнить сравнительный анализ времени работы алгоритмов для  $n$  в интервале от 1 до 10000 с шагом 50.

### 4.1.2 Код реализации

Алгоритмы Нарайана, Джонсона-Троттера, вектора инверсий для генерации все возможных перестановок элементов  $1, \dots, n$  будет иметь вид:

```
import time
import matplotlib.pyplot as plt

# Алгоритм Нарайана
def narayana_perm(n):
    perm = list(range(1, n + 1))
    while True:
        yield perm
        i = n - 2
        while i >= 0 and perm[i] > perm[i + 1]:
            i -= 1
        if i == -1:
            return
        j = n - 1
        while perm[j] < perm[i]:
            j -= 1
        perm[i], perm[j] = perm[j], perm[i]
        perm[i + 1:] = reversed(perm[i + 1:])

# Алгоритм Джонсона-Троттера
def johnson_trotter(n):
    perm = list(range(1, n + 1))
    dirs = [False] * n

    yield perm

    while True:
        mobile = -1
        for i in range(n):
            if ((dirs[i] and i < n - 1 and perm[i] > perm[i + 1]) or
                (not dirs[i] and i > 0 and perm[i] > perm[i - 1])):
                if mobile == -1 or perm[i] > perm[mobile]:
                    mobile = i
```

```

    if mobile == -1:
        return

    swap = mobile + 1 if dirs[mobile] else mobile - 1
    perm[mobile], perm[swap] = perm[swap], perm[mobile]
    dirs[mobile], dirs[swap] = dirs[swap], dirs[mobile]
    mobile = swap

    for i in range(n):
        if perm[i] > perm[mobile]:
            dirs[i] = not dirs[i]

    yield perm

# Вектор инверсий
def inversion_vector(n):
    perm = list(range(1, n + 1))
    while True:
        yield perm
        i = n - 2
        while i >= 0 and perm[i] > perm[i + 1]:
            i -= 1
        if i == -1:
            return
        j = n - 1
        while perm[j] < perm[i]:
            j -= 1
        perm[i], perm[j] = perm[j], perm[i]
        perm[i + 1:] = reversed(perm[i + 1:])

def measure_execution_time(func, n):
    start_time = time.time()
    for _ in func(n):
        pass
    end_time = time.time()
    return end_time - start_time

narayana_times = []
johnson_trotter_times = []
inversion_vector_times = []

n_values = range(1, 12, 1)
for n in n_values:
    narayana_time = measure_execution_time(narayana_perm, n)
    johnson_trotter_time = measure_execution_time(johnson_trotter, n)
    inversion_vector_time = measure_execution_time(inversion_vector, n)

    narayana_times.append(narayana_time)
    johnson_trotter_times.append(johnson_trotter_time)
    inversion_vector_times.append(inversion_vector_time)

```

Программная реализация сравнительного анализа времени работы алгоритмов для  $n$  в интервале от 1 до 10000 с шагом 50 имеет вид:

```
plt.figure(figsize=(10, 6))
plt.plot(n_values, narayana_times, label="Нарайана", color='r')
plt.plot(n_values, johnson_trotter_times, label="Джонсон-Троттер", color='g')
plt.plot(n_values, inversion_vector_times, label="Вектор инверсий",
color='b')
plt.xlabel("n")
plt.ylabel("Время выполнения (в секундах)")
plt.title("Время выполнения алгоритмов перестановки")
plt.legend()
plt.show()
```

#### 4.1.3 Оценка сложности реализации

Временной анализ для алгоритмов перестановки имеет вид, изображенный на рисунке 4:

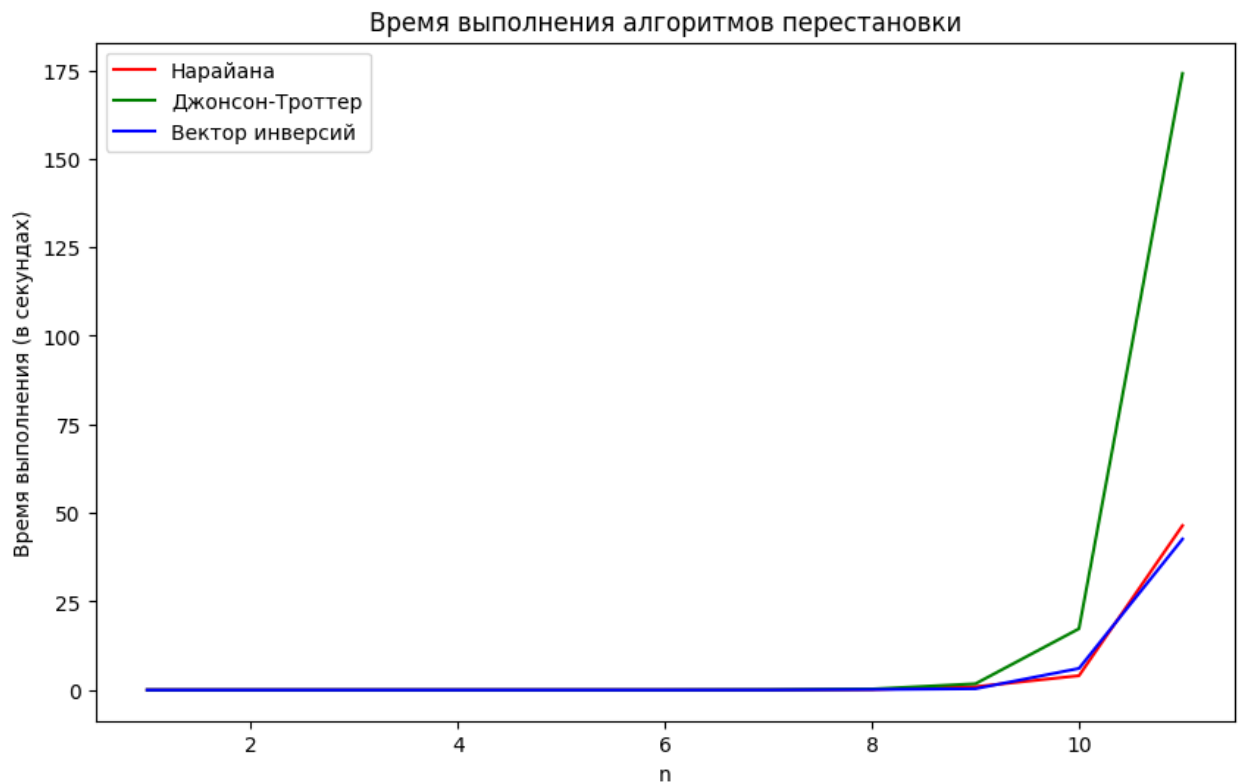


Рисунок 4 – Анализ времени выполнения алгоритмов перестановки

### 4.2.1 Постановка задачи

Дано множество элементов целого типа, сгенерировать все возможные комбинации перестановок элементов (элементы могут повторяться, каждая сгенерированная подпоследовательность должна быть уникальной).

#### Пример:

1. Последовательность: 121

Сгенерированные перестановки:

112

121

211

2. Последовательность: 123

Сгенерированные перестановки:

123

132

213

231

312

321

### 4.2.2 Код реализации

```
def generate_permutations(elements):  
    def backtrack(start_index):  
        if start_index == len(elements):  
            yield tuple(elements)  
        for i in range(start_index, len(elements)):  
            elements[start_index], elements[i] = elements[i],  
elements[start_index]  
            yield from backtrack(start_index + 1)  
            elements[start_index], elements[i] = elements[i],  
elements[start_index]  
  
    unique_permutations = set(backtrack(0))  
    return unique_permutations  
  
my_set = [1, 2, 3]  
result = generate_permutations(my_set)  
for perm in result:  
    print(*perm)
```

```
1 2 1  
2 1 1  
1 1 2
```

### 4.3.1 Постановка задачи

Пусть задано  $n$ -элементов. Необходимо перебрать все возможные варианты выборки из этих элементов (выборки по одному элементу, по два, ..., из  $n$  элементов). Пример элементов: стол, стул, шкаф.

### 4.3.2 Код реализации

```
elements = ['стол', 'стул', 'шкаф']
```

```
index = len(elements)
```

```
N = 2**index
```

```
x = '0' + str(index) + 'b'
```

```
for i in range(N):
    s = list()
    i = format(i, x)
    for j in range(index):
        if int(i[j]):
            s.append(elements[j])
    if s:
        print(*s)
```

```
стол
стул
стол стул
шкаф
стол шкаф
стул шкаф
стол стул шкаф
```

### 4.4.1. Постановка задачи

Студент Петров П.П. имеет некоторую сумму денег для покупки канцелярских принадлежностей. Он составил список необходимых канцелярских принадлежностей с указанием их названия и количества. Пусть имеется прейскурант цен магазина, в котором указаны наименования товаров с указанием цены. Определить, какие канцелярские принадлежности и в каком количестве может купить студент в пределах выделенной суммы при условии, что количество наименований в списке покупок должно быть максимально.



#### 4.4.2 Код реализации

```
def new_buy_max(money, shop_list, price_list):
    buy_list = dict()
    while shop_list:
        sorted_price_list = sorted(price_list.items(), key=lambda x: x[1])
        if not sorted_price_list or money < sorted_price_list[0][1]:
            break
        for item, price in sorted_price_list:
            if money >= price:
                money -= price
                buy_list[item] = buy_list.get(item, 0) + 1
                shop_list[item] -= 1
                if shop_list[item] == 0:
                    del shop_list[item]
                    del price_list[item]

    print(f'Мин сумма для покупки: {min_money}\n{buy_list}\nОстаток: {money}
    p.')

shop_list = {'Ручка': 2, 'Карандаш': 1, 'Линейка': 1, 'Точилка': 1,
             'Циркуль': 1}
price_list = {'Ручка': 90, 'Карандаш': 30, 'Линейка': 70, 'Точилка': 40,
              'Циркуль': 90}

min_money = 0
for key in shop_list.keys():
    min_money += shop_list[key] * price_list[key]

money = int(input('Сумма: '))
new_buy_max(money, shop_list, price_list)

Сумма: 300
Мин сумма для покупки: 410
{'Карандаш': 1, 'Точилка': 1, 'Линейка': 1, 'Ручка': 1}
Остаток: 70 p.
```

## ЛАБОРАТОРНАЯ РАБОТА 5 (Генерация перестановок)

### 5.1.1. Постановка задачи

Сгенерировать перестановки значений  $1, \dots, n$ , используя код Грэя.

### 5.1.2. Код реализации

```
def generate_permutations(n):
    if n == 1:
        return [[1]]

    result = []
    perms = generate_permutations(n - 1)
    for perm in perms:
        for i in range(n):
            current = perm[:i] + [n] + perm[i:]
            result.append(current)

    return result

n = 4
all_permutations = generate_permutations(n)
for perm in all_permutations:
    print(perm)
```

```
[4, 3, 2, 1]
[3, 4, 2, 1]
[3, 2, 4, 1]
[3, 2, 1, 4]
[4, 2, 3, 1]
[2, 4, 3, 1]
[2, 3, 4, 1]
[2, 3, 1, 4]
[4, 2, 1, 3]
[2, 4, 1, 3]
[2, 1, 4, 3]
[2, 1, 3, 4]
[4, 3, 1, 2]
[3, 4, 1, 2]
[3, 1, 4, 2]
[3, 1, 2, 4]
[4, 1, 3, 2]
[1, 4, 3, 2]
[1, 3, 4, 2]
[1, 3, 2, 4]
[4, 1, 2, 3]
[1, 4, 2, 3]
[1, 2, 4, 3]
[1, 2, 3, 4]
```

### 5.2.1. Постановка задачи

Выполнить сравнительный анализ алгоритмов генерации случайных перестановок.

### 5.2.2. Код реализации

```
import random
import time

def gray_code(n):
    def gray_code_recursive(n):
        if n == 0:
            return ['']
        first_half = gray_code_recursive(n - 1)
        second_half = first_half.copy()

        first_half = ['0' + code for code in first_half]
        second_half = ['1' + code for code in reversed(second_half)]

        return first_half + second_half

    return [int(code, 2) + 1 for code in gray_code_recursive(n)]

def random_permutation(n):
    permutation = list(range(1, n + 1))
    random.shuffle(permutation)
    return permutation

def fisher_yates_shuffle(n):
    permutation = list(range(1, n + 1))
    for i in range(n - 1, 0, -1):
        j = random.randint(0, i)
        permutation[i], permutation[j] = permutation[j], permutation[i]
    return permutation

import matplotlib.pyplot as plt

# Сохраняем результаты в списке
gray_code_times = []
random_permutation_times = []
fisher_yates_shuffle_times = []

for n in range(1, 22):
    start_time = time.time()
    gray_code(n)
    gray_code_times.append(time.time() - start_time)

    start_time = time.time()
    random_permutation(n)
    random_permutation_times.append(time.time() - start_time)
```

```

start_time = time.time()
fisher_yates_shuffle(n)
fisher_yates_shuffle_times.append(time.time() - start_time)

```

Программная реализация анализа времени выполнения алгоритмов перестановки:

```

plt.figure(figsize=(10, 6))
plt.plot(range(1, 22), gray_code_times, label='gray_code')
plt.plot(range(1, 22), random_permutation_times, label='random_permutation')
plt.plot(range(1, 22), fisher_yates_shuffle_times,
label='fisher_yates_shuffle')
plt.xlabel('n')
plt.ylabel('Время выполнения (сек)')
plt.legend()
plt.show()

```

### 6.1.3 Оценка сложности реализации

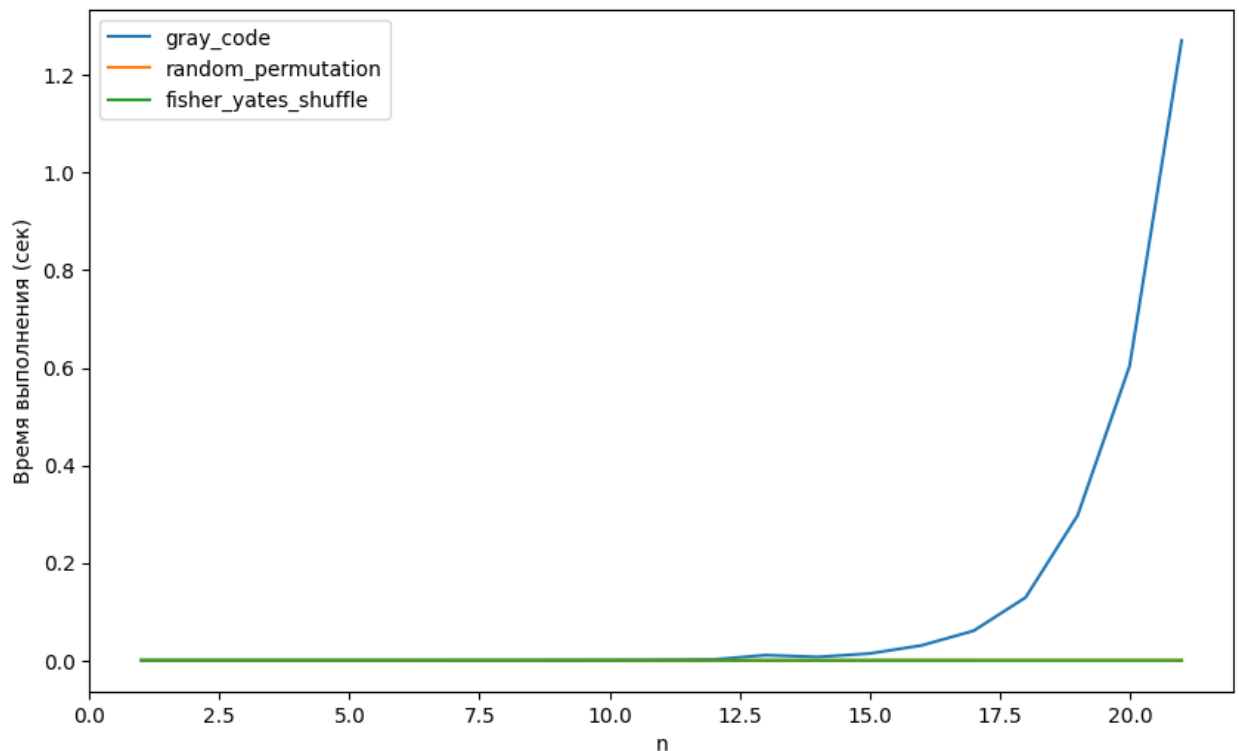


Рисунок 5 – Анализ времени выполнения алгоритмов перестановки

## ЛАБОРАТОРНАЯ РАБОТА 6 (Алгоритмы работы со строками)

### 6.1.1. Постановка задачи

Выполнить сравнительный анализ работы алгоритмов по поиску подстроки в строке (Наивный метод, Алгоритм Бойера-Мура, Алгоритм Рабина-Карпа, Алгоритм Кнута-Морриса-Пратта).

### 6.1.2. Код реализации

Наивный метод

```
def naive_search(text, pattern):
    n = len(text)
    m = len(pattern)
    matches = []

    for i in range(n - m + 1):
        j = 0
        while j < m and text[i + j] == pattern[j]:
            j += 1
        if j == m:
            matches.append(i)

    return matches
```

Алгоритм Бойера-Мура

```
def boyer_moore(text, pattern):
    def create_bad_table(pattern):
        bad_table = {}
        for i in range(len(pattern) - 1):
            bad_table[pattern[i]] = len(pattern) - i - 1
        return bad_table

    def find_suffix_position(bad_table, suffix, full_pattern):
        for i in range(len(suffix) - 1, -1, -1):
            if suffix[i:] == full_pattern[:len(suffix[i:]):]:
                return len(full_pattern) - len(suffix[i:])
        return len(full_pattern)

    if len(pattern) == 0:
        return []

    bad_table = create_bad_table(pattern)
    matches = []

    i = 0
    while i <= len(text) - len(pattern):
        j = len(pattern) - 1
```

```

    while j >= 0 and pattern[j] == text[i + j]:
        j -= 1
    if j == -1:
        matches.append(i)
        i += 1
    else:
        bad_char_skip = bad_table.get(text[i + j], len(pattern))
        good_suffix_skip = find_suffix_position(bad_table, pattern[j +
1:], pattern)
        i += max(bad_char_skip, good_suffix_skip)

    return matches

```

## Алгоритм Рабина-Карпа

```

def rabin_karp(text, pattern):
    if len(pattern) == 0:
        return []

    prime = 101
    matches = []
    n = len(text)
    m = len(pattern)
    pattern_hash = 0
    text_hash = 0
    h = 1

    for i in range(m - 1):
        h = (h * 256) % prime

    for i in range(m):
        pattern_hash = (256 * pattern_hash + ord(pattern[i])) % prime
        text_hash = (256 * text_hash + ord(text[i])) % prime

    for i in range(n - m + 1):
        if pattern_hash == text_hash:
            match = True
            for j in range(m):
                if text[i + j] != pattern[j]:
                    match = False
                    break
            if match:
                matches.append(i)

        if i < n - m:
            text_hash = (256 * (text_hash - ord(text[i]) * h) + ord(text[i +
m])) % prime
            if text_hash < 0:
                text_hash += prime

    return matches

```

## Алгоритм Кнута-Морриса-Пратта

```
def compute_prefix(pattern):
    m = len(pattern)
    prefix = [0] * m
    length = 0
    i = 1

    while i < m:
        if pattern[i] == pattern[length]:
            length += 1
            prefix[i] = length
            i += 1
        else:
            if length != 0:
                length = prefix[length - 1]
            else:
                prefix[i] = 0
                i += 1

    return prefix

def kmp(text, pattern):
    n = len(text)
    m = len(pattern)
    prefix = compute_prefix(pattern)
    matches = []

    i = 0
    j = 0

    while i < n:
        if pattern[j] == text[i]:
            i += 1
            j += 1

            if j == m:
                matches.append(i - j)
                j = prefix[j - 1]
        else:
            if j != 0:
                j = prefix[j - 1]
            else:
                i += 1

    return matches
```

## Программная реализация алгоритмов по поиску подстроки в строке:

```
import time
import random
import matplotlib.pyplot as plt

def generate_random_text(length):
    return ''.join(random.choices('abcdefghijklmnopqrstuvwxyz', k=length))

def generate_random_pattern(length):
    return ''.join(random.choices('abcdefghijklmnopqrstuvwxyz', k=length))

def time_search_algorithm(search_algorithm, text, pattern, num_trials=10):
    total_time = 0
    for _ in range(num_trials):
        start_time = time.time()
        search_algorithm(text, pattern)
        end_time = time.time()
        total_time += (end_time - start_time)
    return total_time / num_trials

text_lengths = list(range(1000, 10001, 1000))
naive_times = []
boyer_moore_times = []
rabin_karp_times = []
kmp_times = []

for length in text_lengths:
    text = generate_random_text(length)
    pattern = generate_random_pattern(10)

    naive_times.append(time_search_algorithm(naive_search, text, pattern))
    boyer_moore_times.append(time_search_algorithm(boyer_moore, text,
pattern))
    rabin_karp_times.append(time_search_algorithm(rabin_karp, text, pattern))
    kmp_times.append(time_search_algorithm(kmp, text, pattern))

plt.plot(text_lengths, naive_times, label='Наивный метод')
plt.plot(text_lengths, boyer_moore_times, label='Алгоритм Бойера-Мура')
plt.plot(text_lengths, rabin_karp_times, label='Алгоритм Рабина-Карпа')
plt.plot(text_lengths, kmp_times, label='Алгоритм Кнута-Морриса-Пратта')

plt.xlabel('Длина текста')
plt.ylabel('Среднее время выполнения (секунды)')
plt.title('Сравнение производительности алгоритмов поиска подстроки')
plt.legend()
plt.show()
```



### 6.1.3. Оценка сложности реализации

Анализ времени выполнения алгоритмов по поиску подстроки в строке имеет вид, изображенный на рисунке 6:

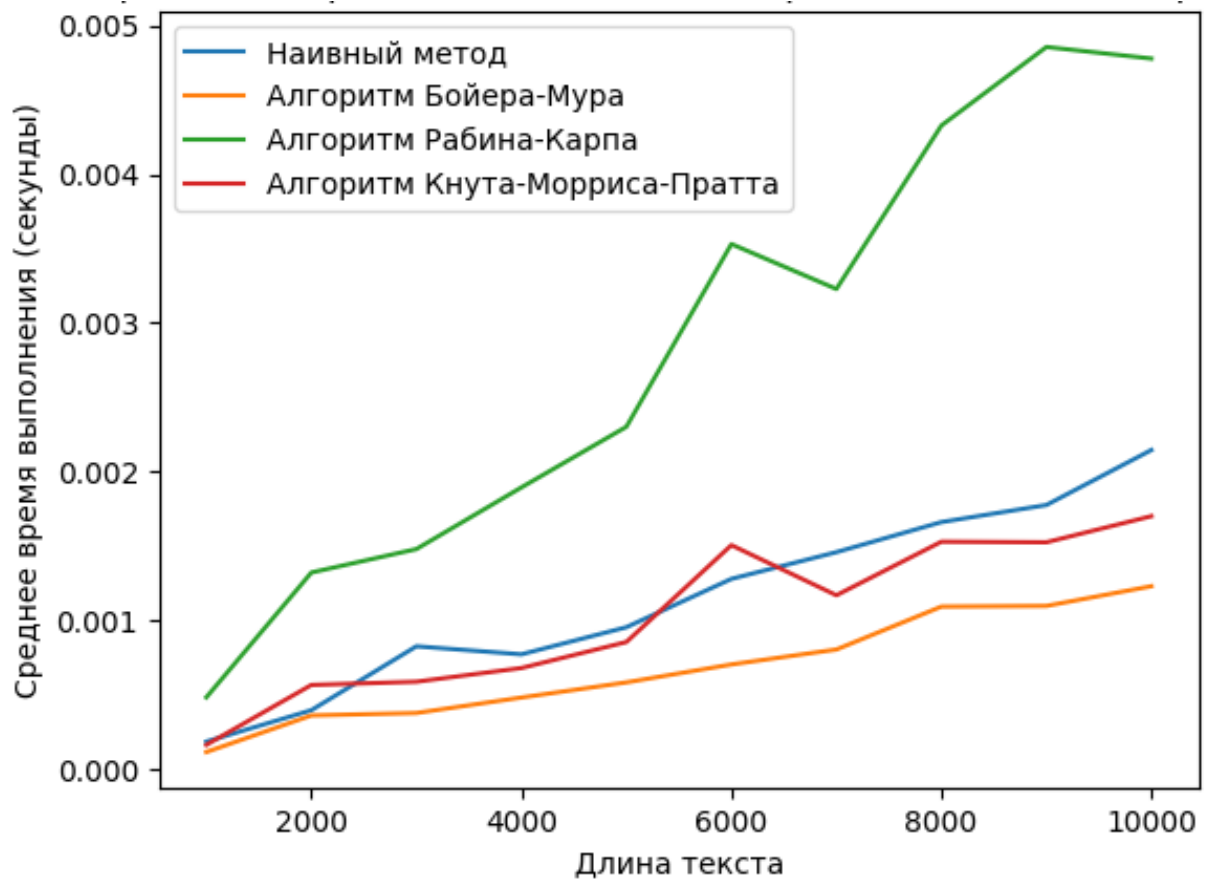


Рисунок 6 – Анализ времени выполнения алгоритмов по поиску подстроки в строке

## ЛАБОРАТОРНАЯ РАБОТА 7 (Жадные алгоритмы)

### 7.1.1. Постановка задачи

**Задача 1:** На прямой даны  $n$  отрезков, нужно выбрать максимальное по размеру подмножество непересекающихся.

### 7.1.2. Код реализации

```
segments = [(1, 3), (1, 6), (2, 9), (4, 7), (8, 13), (10, 11), (12, 14), (14, 16), (15, 20), (17, 21)]
```

```
def max_non_overlapping_segments(segments):  
    segments.sort(key=lambda x: x[1])  
    chosen_segments = []  
    end_point = float('-inf')
```

```
    for segment in segments:  
        if segment[0] > end_point:  
            chosen_segments.append(segment)  
            end_point = segment[1]
```

```
    return chosen_segments
```

```
selected_segments = max_non_overlapping_segments(segments)  
print(f'Отрезки: {segments}\nМаксимальное по размеру подмножество  
непересекающихся отрезков: {selected_segments}')
```

```
fig, ax = plt.subplots(figsize=(8, 3))
```

```
height_non_overlap = 1  
height_overlap = 1.05
```

```
overlap_segments = list(set(segments).difference(set(selected_segments)))
```

```
for segment in selected_segments:  
    ax.plot([segment[0], segment[1]], [height_non_overlap,  
height_non_overlap], color='red', linewidth=2)
```

```
for segment in overlap_segments:  
    if any(s[0] < segment[1] < s[1] or s[0] < segment[0] < s[1] for s in  
overlap_segments if s != segment):  
        ax.plot([segment[0], segment[1]], [height_overlap, height_overlap],  
color='black', linewidth=2)  
        height_overlap += 0.05  
    else:  
        ax.plot([segment[0], segment[1]], [height_non_overlap + 0.05,  
height_non_overlap + 0.05], color='black', linewidth=2)
```

```
plt.title('Множество отрезков')  
plt.xlabel('Отрезки')
```

```

red_patch = mpatches.Patch(color='red', label='Непересекающиеся отрезки')
black_patch = mpatches.Patch(color='black', label='Пересекающиеся отрезки')
plt.legend(handles=[red_patch, black_patch])

plt.show()

```

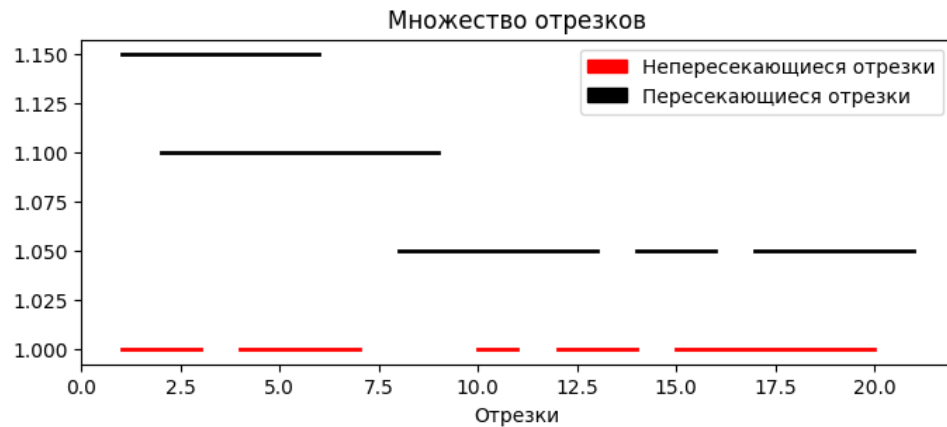


Рисунок 7 – Множество отрезков

### 7.2.1 Постановка задачи

**Задача 2:** Для конкретной аудитории есть ряд заявок с временем начала и окончания работы. Необходимо вернуть максимальное количество допустимых заявок.

### 7.2.2. Код реализации

```

requests = [(1, 3), (5, 7), (11, 13), (12, 14), (16, 20), (17, 21)]

def max_requests(requests):
    requests.sort(key=lambda x: x[1])
    chosen_requests = []
    end_point = float('-inf')

    for request in requests:
        if request[0] > end_point:
            chosen_requests.append(request)
            end_point = request[1]

    return chosen_requests

selected_requests = max_requests(requests)
print(f'Заявки: {requests}\nМаксимальное количество допустимых заявок: {len(selected_requests)} - {selected_requests}')

import matplotlib.pyplot as plt
import matplotlib.patches as mpatches

```

```

fig, ax = plt.subplots(figsize=(8, 3))

height_non_overlap = 1
height_overlap = 1.05

overlap_requests = list(set(requests).difference(set(selected_requests)))

for request in selected_requests:
    ax.plot([request[0], request[1]], [height_non_overlap,
height_non_overlap], color='red', linewidth=2)

for request in overlap_requests:
    if any(s[0] < request[1] < s[1] or s[0] < request[0] < s[1] for s in
overlap_requests if s != request):
        ax.plot([request[0], request[1]], [height_overlap, height_overlap],
color='black', linewidth=2)
        height_overlap += 0.05
    else:
        ax.plot([request[0], request[1]], [height_non_overlap + 0.05,
height_non_overlap + 0.05], color='black', linewidth=2)

plt.title('Множество заявок')
plt.xlabel('Заявки')

red_patch = mpatches.Patch(color='red', label='Недопустимые заявки')
black_patch = mpatches.Patch(color='black', label='Допустимые заявки')
plt.legend(handles=[red_patch, black_patch])

plt.show()

```

Заявки: [(1, 3), (5, 7), (11, 13), (12, 14), (16, 20), (17, 21)]  
Максимальное количество допустимых заявок: 4 – [(1, 3), (5, 7), (11, 13), (16, 20)]

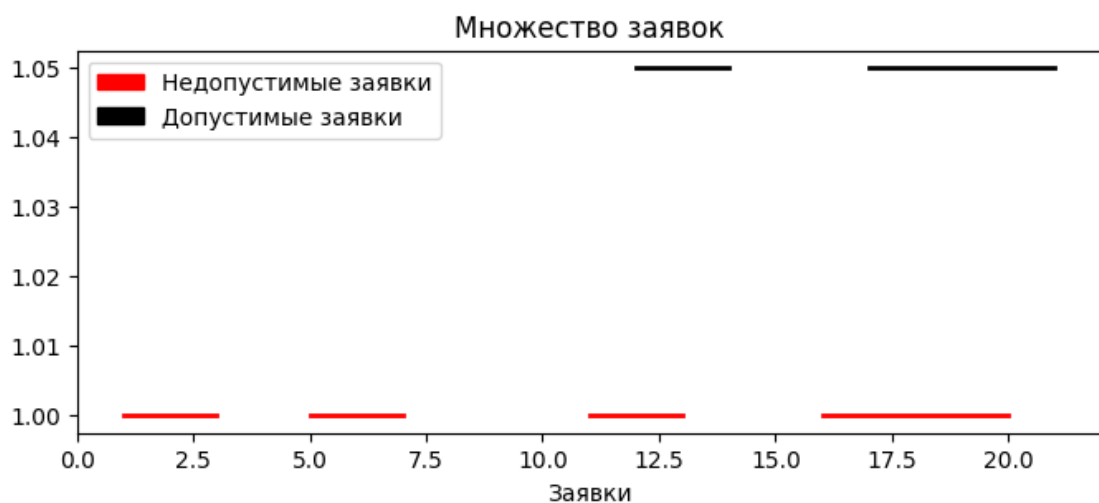


Рисунок 8 – Множество заявок

### 7.3.1 Постановка задачи

**Задача 3:** На прямой расположены отрезки  $[l_i, r_i]$ , которые полностью или даже с избытком покрывают интервал  $[L, R]$ . необходимо выбрать наименьшее множество отрезков так, чтобы они всё ещё покрывали интервал.

### 7.3.2. Код реализации

```
intervals = [(1, 4), (2, 5), (3, 6), (5, 7), (6, 8)]
L, R = 1, 8

def min_covering_intervals(segments, L, R):
    segments.sort(key=lambda x: x[0])

    result = []
    i = 0
    while L < R:
        covering_segment = None
        while i < len(segments) and segments[i][0] <= L:
            if covering_segment is None or segments[i][1] >
covering_segment[1]:
                covering_segment = segments[i]
            i += 1
        if covering_segment is None:
            return None
        result.append(covering_segment)
        L = covering_segment[1]
    return result

selected_segments = min_covering_intervals(intervals,L,R)
print(f"Отрезки: {intervals}\nНаименьшее множество отрезков, покрывающее
интервал [{L}, {R}]: {selected_segments}")
```

```
Отрезки: [(1, 4), (2, 5), (3, 6), (5, 7), (6, 8)]
Наименьшее множество отрезков, покрывающее интервал [1, 8]: [(1, 4), (3,
6), (6, 8)]
```

## ЛАБОРАТОРНАЯ РАБОТА 8 (Жадные алгоритмы)

### 8.1.1 Постановка задачи

**Задача 1:** Имеется  $m$  дней и  $n$  заказов, каждый из которых займёт один день. У каждого заказа есть номер, стоимость и дедлайн. Каждый заказ можно начинать в любой день до дедлайна, одновременно можно делать только один заказ. Определить наилучшее по суммарной стоимости число заказов.

### 8.1.2. Код реализации

```
import heapq
import pandas as pd

def max_profit(orders):
    sort_orders = orders.copy()
    sort_orders.sort(key=lambda x: x[1])
    h = []
    for order in sort_orders:
        cost = order[2]
        if len(h) < order[1]:
            heapq.heappush(h, (cost, order[0]))
        elif h and cost > h[0][0]:
            heapq.heapreplace(h, (cost, order[0]))
    return sum(order[0] for order in h), h

orders = [(1, 2, 40), (2, 1, 25), (3, 2, 30), (4, 1, 15), (5, 3, 20)]
total, order_list = max_profit(orders)

df = pd.DataFrame(orders, columns=['Номер заказа', 'Дедлайн', 'Стоимость'])
df['Выбран'] = df['Стоимость'].apply(lambda x: 'Да' if x in [order[0] for
order in order_list] else 'Нет')
df.set_index('Номер заказа', inplace=True)

display(df)
print(f"Наилучшее по суммарной стоимости число заказов:
{len(order_list)}\nМаксимальная прибыль : {total} ---> Стоимости и номера
заказов: {dict(order_list)}")
```

	Дедлайн	Стоимость	Выбран
Номер заказа			
1	2	40	Да
2	1	25	Нет
3	2	30	Да
4	1	15	Нет
5	3	20	Да

Наилучшее по суммарной стоимости число заказов: 3  
 Максимальная прибыль : 90 ---> Стоимости и номера заказов: {20: 5, 40: 1, 30: 3}

Рисунок 9 – Вывод программы

### 8.2.1 Постановка задачи

**Задача 2:** Необходимо организовать детский утренник на который пришло  $n$  детей. У каждого ребёнка есть номер и возраст. Определить минимальное количество групп, которые будут организованы, но при этом возраст детей в каждой группе не должен отличаться больше, чем на 2 года.

### 8.2.2. Код реализации

```
import random

def generate_children(n):
    children = []
    for i in range(1, n+1):
        age = random.randint(3, 10)
        children.append((i, age))
    return children

def organize_groups(children):
    sort_children = children.copy()
    sort_children.sort(key=lambda x: x[1])

    groups = []
    current_group = [sort_children[0]]

    for i in range(1, len(sort_children)):
        if sort_children[i][1] - current_group[0][1] <= 2:
            current_group.append(sort_children[i])
        else:
            groups.append(current_group)
            current_group = [sort_children[i]]

    groups.append(current_group)

    return groups
```

```

n = 8
children = generate_children(n)

groups = organize_groups(children)

df = pd.DataFrame(columns=['Номер ребенка', 'Возраст', 'Группа'])
for i, group in enumerate([sorted(group, key=lambda x: x[0]) for group in
groups], start=1):
    for child in group:
        df.loc[len(df)] = {'Номер ребенка': child[0], 'Возраст': child[1],
'Группа': i}
df.set_index('Номер ребенка', inplace=True)

print('Дети:')
for child in children:
    print(f'Ребенок №{child[0]} - {child[1]} лет')

display(df)
print(f'Минимальное количество групп: {len(groups)}')

```

Дети:

Ребенок №1 - 3 лет  
Ребенок №2 - 7 лет  
Ребенок №3 - 10 лет  
Ребенок №4 - 6 лет  
Ребенок №5 - 4 лет  
Ребенок №6 - 6 лет  
Ребенок №7 - 6 лет  
Ребенок №8 - 9 лет

	Возраст	Группа
Номер ребенка		
1	3	1
5	4	1
2	7	2
4	6	2
6	6	2
7	6	2
3	10	3
8	9	3

Минимальное количество групп: 3

Рисунок 10 – Вывод программы



## ЛАБОРАТОРНАЯ РАБОТА 9 (Жадные алгоритмы)

### 9.1.1. Постановка задачи

Дана последовательность элементов с заданным весом и стоимостью, каждый элемент может встречаться только один раз. Необходимо заполнить рюкзак предметами, суммарная стоимость которых была бы как можно большей, грузоподъемность рюкзака ограничивается  $W$  килограммами, где  $W$  – целая величина. Определить предметы, которые будут помещены в рюкзак. Решение выполнить с использованием полного перебора и жадным алгоритмом. Оценить сложность работы алгоритмов.

### 9.1.2. Код реализации

Программная реализация алгоритма полного перебора:

```
from itertools import combinations
import time

def knapsack_brute_force(weights, values, W):
    max_value = 0
    best_combination = None

    for r in range(1, len(weights) + 1):
        for items in combinations(range(len(weights)), r):
            weight = sum(weights[i] for i in items)
            value = sum(values[i] for i in items)

            if value > max_value and weight <= W:
                max_value = value
                best_combination = items

    return best_combination, max_value
```

Программная реализация жадного алгоритма:

```
def knapsack_greedy(weights, values, W):
    items = sorted([(w, v, v/w) for w, v in zip(weights, values)],
                    key=lambda x: x[2], reverse=True)

    max_value = 0
```

```

best_combination = []

for item in items:
    if W >= item[0]:
        best_combination.append(item)
        max_value += item[1]
        W -= item[0]

return best_combination, max_value

weights = [12, 1, 21, 30, 15, 40]
values = [4, 2, 10, 40, 8, 50]
W = 50

```

### Программа для сравнительного анализа алгоритмов:

```

start = time.time()
items, max_value = knapsack_brute_force(weights, values, W)
end = time.time()
print(f"Полный перебор: {round((end - start), 7)} секунд")
print("Предметы:", [(weights[i], values[i]) for i in items]) # выводим вес
и стоимость
print("Максимальная стоимость:", max_value)

start = time.time()
items, max_value = knapsack_greedy(weights, values, W)
end = time.time()
print(f"\nЖадный алгоритм: {round((end - start), 7)} секунд")
print("Предметы:", [item[:2] for item in items]) # выводим вес и стоимость
print("Максимальная стоимость:", max_value)

```

### 9.1.3. Оценка сложности реализации

Сравнительный анализ работы алгоритмов по определению предметов, которые будут помещены в рюкзак представлен на рисунке 11.

⇒ Полный перебор: 0.0002356 секунд  
Предметы: [(1, 2), (40, 50)]  
Максимальная стоимость: 52

Жадный алгоритм: 0.0001426 секунд  
Предметы: [(1, 2), (30, 40), (15, 8)]  
Максимальная стоимость: 50

Рисунок 11 – Сравнительный анализ работы алгоритмов

## ЛАБОРАТОРНАЯ РАБОТА 10 (Раскраска графа)

### 10.1.1. Постановка задачи

Задача о распределении работ между механизмами. Заданы множества  $V = \{v_1, v_2, \dots, v_n\}$  и  $S = \{s_1, s_2, \dots, s_m\}$  работ и механизмов соответственно. Для выполнения каждой из работ требуется некоторое время, одинаковое для всех работ, и некоторые механизмы. При этом никакой из механизмов не может быть одновременно занят в нескольких работах. Нужно распределить механизмы так, чтобы общее время выполнения всех работ было минимальным. Использовать жадный алгоритм. Оценить сложность работы алгоритма.

### 10.1.2. Код реализации

Программная реализация жадного алгоритма для задачи о распределении работ между механизмами:

```
import time

def assign_jobs(V, S):
    machines = [{'id': s, 'jobs': []} for s in S]

    for i, job in enumerate(V):
        machines[i % len(S)]['jobs'].append(job)

    return machines

V = ['v1', 'v2', 'v3', 'v4', 'v5']
S = ['s1', 's2', 's3']

start = time.time()
machines = assign_jobs(V, S)
end = time.time()

print(f"Сложность O(n); Время выполнения {round((end - start), 8)} секунды")

for machine in machines:
    print(f"Механизм {machine['id']} назначены работы: {'',
        '.join(machine['jobs'])}")
```

### 10.1.3. Оценка сложности реализации

Анализ оценки сложности работы алгоритма представлен на рисунке 12.

⇒ Сложность  $O(n)$ ; Время выполнения  $5.794e-05$  секунды  
Механизм s1 назначены работы: v1, v4  
Механизм s2 назначены работы: v2, v5  
Механизм s3 назначены работы: v3

Рисунок 12 – Анализ оценки сложности работы алгоритма

### 10.2.1. Постановка задачи

Задача о размещении грузов. Пусть имеется  $n$  грузов, которые необходимо разместить по контейнерам для перевозки, причем некоторые грузы нельзя помещать в один контейнер. Определить наименьшее количество контейнеров, которые потребуются для перевозки всех грузов, определить компоновку грузов по контейнерам. Оценить сложность работы алгоритма.

### 10.2.2. Код реализации

Программная реализация решения задачи о размещении грузов:

```
def assign_containers(cargo):  
    containers = []  
  
    for c in cargo:  
        for container in containers:  
            if not any(conflict in container for conflict in c['conflicts']):  
                container.append(c['id'])  
                break  
        else:  
            containers.append([c['id']])  
  
    return containers  
  
cargo = [{'id': 'c1', 'conflicts': ['c2', 'c3']}, {'id': 'c2', 'conflicts':  
['c1']}, {'id': 'c3', 'conflicts': ['c1']}, {'id': 'c4', 'conflicts': []}]
```

```

start = time.time()
containers = assign_containers(cargo)
end = time.time()

print(f"Сложность O(n^2); Время выполнения {round((end - start), 7)}
секунды")

for i, container in enumerate(containers, start=1):
    print(f"Контейнер {i} содержит грузы: {' '.join(container)}")

```

### 10.2.3. Оценка сложности реализации

Анализ оценки сложности работы алгоритма представлен на рисунке 13.

⇒ Сложность  $O(n^2)$ ; Время выполнения  $9.94e-05$  секунды  
 Контейнер 1 содержит грузы: c1, c4  
 Контейнер 2 содержит грузы: c2, c3

---

Рисунок 13 – Анализ оценки сложности работы алгоритма

## ЛАБОРАТОРНАЯ РАБОТА 11 (Фано)

### 11.1.1. Постановка задачи

На вход программы подается текст, выполнить кодировку текста. Выдать словарь, в котором каждому символу соответствует код, удовлетворяющий условию Фано. Текст должен быть закодирован минимально возможной последовательностью нулей и единиц. Выполнить восстановление закодированного текста. Оценить сложность работы алгоритма.

### 11.1.2. Код реализации

Программная реализация кодировки текста:

```
from collections import Counter
from queue import PriorityQueue
import time

class Node:
    def __init__(self, char, freq, left=None, right=None):
        self.char = char
        self.freq = freq
        self.left = left
        self.right = right

    def __lt__(self, other):
        return self.freq < other.freq

def build_tree(text):
    counter = Counter(text)
    queue = PriorityQueue()
    for char, freq in counter.items():
        queue.put(Node(char, freq))

    while queue.qsize() > 1:
        left = queue.get()
        right = queue.get()
```

```

        queue.put(Node(None, left.freq + right.freq, left, right))

    return queue.get()

def build_codes(root, code, codes):
    if root is None:
        return

    if root.char is not None:
        codes[root.char] = code
        return

    build_codes(root.left, code + '0', codes)
    build_codes(root.right, code + '1', codes)

def encode(text):
    root = build_tree(text)
    codes = {}
    build_codes(root, '', codes)
    return ''.join(codes[char] for char in text), codes

def decode(encoded, codes):
    char_to_code = {code: char for char, code in codes.items()}
    decoded = ''
    code = ''

    for bit in encoded:
        code += bit

        if code in char_to_code:
            decoded += char_to_code[code]
            code = ''

    return decoded

text = """
Я помню чудное мгновенье:
Передо мной явилась ты,
Как мимолетное виденье,

```



```

Как гений чистой красоты
"""

start = time.time()
encoded, codes = encode(text)
end = time.time()

print(f"Время кодирования {round((end - start), 8)}")
print(f"Закодированный текст: {encoded}")
print(f"Словарь кодов: {codes}")

start = time.time()
decoded = decode(encoded, codes)
end = time.time()

print(f"\nВремя декодирования {round((end - start), 8)}")
print(f"Декодированный текст: {decoded}")

```

### 11.1.3. Оценка сложности реализации

Анализ оценки сложности работы алгоритма представлен на рисунке 20.

```

Время кодирования 0.00077534
Закодированный текст:
0001111011001111101111110010110010110000111101100000001001111001111010011
00101001011100111110000010110000001010101100100011001000101101110101001111
1101100101100111110101011000001100000011011111101010001000010111110010110
11011100001111010110101010001100100011001011111011110101110011001111010011
10000001110011010110000001010101110000111101011010101000111001010101100001
11010101111011000111000111100111110101011101001101111101010001111111100101
1010001
Словарь кодов: {'у': '000000', 'я': '000001', 'ь': '00001', '\n': '0001',
'м': '0010', 'и': '0011', 'е': '010', ' ': '011', 'в': '10000', 'с':
'10001', 'п': '100100', 'т': '100101', 'д': '10011', 'к': '10100', 'й':
'10101', 'ю': '1011000', ':': '1011001', 'ы': '101101', ',': '101110',
'л': '101111', 'н': '1100', 'а': '11010', 'ч': '110110', 'р': '110111',
'т': '11100', 'к': '111010', 'я': '1110110', 'п': '1110111', 'о': '1111'}

Время декодирования 0.00023818
Декодированный текст:
Я помню чудное мгновенье:
Передо мной явилась ты,
Как мимолетное виденье,
Как гений чистой красоты

```

Рисунок 21 – Анализ сложности работы алгоритма

## ЛАБОРАТОРНАЯ РАБОТА 12 (Оператор мобильной связи)

### 12.1.1. Постановка задачи

Оператор мобильной связи организовал базу данных абонентов, содержащую сведения о телефонах, их владельцах и используемых тарифах, в виде бинарного дерева.

Разработать программу, которая:

- обеспечивает начальное формирование базы данных в виде бинарного дерева;
- производит вывод всей базы данных;
- производит поиск владельца по номеру телефона;
- выводит наиболее востребованный тариф (по наибольшему числу абонентов).

### 12.1.2. Код реализации

Программа, выполняющая все заданные условия задачи, будет выглядеть следующим образом:

```
class Node:
    def __init__(self, phone, owner, tariff):
        self.phone = phone
        self.owner = owner
        self.tariff = tariff
        self.left = None
        self.right = None

class BinaryTree:
    def __init__(self):
        self.root = None
        self.tariffs = {}

    def insert(self, phone, owner, tariff):
        if not self.root:
            self.root = Node(phone, owner, tariff)
        else:
            self._insert(self.root, phone, owner, tariff)
        self.tariffs[tariff] = self.tariffs.get(tariff, 0) + 1

    def _insert(self, node, phone, owner, tariff):
        if phone < node.phone:
            if node.left is None:
                node.left = Node(phone, owner, tariff)
            else:
                self._insert(node.left, phone, owner, tariff)
        else:
            if node.right is None:
```

```

        node.right = Node(phone, owner, tariff)
    else:
        self._insert(node.right, phone, owner, tariff)

def find_owner(self, phone):
    return self._find_owner(self.root, phone)

def _find_owner(self, node, phone):
    if node is None:
        return None
    if node.phone == phone:
        return node.owner
    if phone < node.phone:
        return self._find_owner(node.left, phone)
    else:
        return self._find_owner(node.right, phone)

def print_tree(self):
    self._print_tree(self.root)

def _print_tree(self, node):
    if node is not None:
        self._print_tree(node.left)
        print(f"Phone: {node.phone}, Owner: {node.owner}, Tariff: {node.tariff}")
        self._print_tree(node.right)

def most_popular_tariff(self):
    return max(self.tariffs, key=self.tariffs.get)

tree = BinaryTree()
tree.insert('79001234567', 'Иван Иванов', 'Тариф 1')
tree.insert('79007654321', 'Петр Петров', 'Тариф 2')
tree.insert('79001112233', 'Сергей Сергеев', 'Тариф 1')
tree.insert('79003322110', 'Анна Аннова', 'Тариф 3')
tree.insert('79005566777', 'Мария Мариева', 'Тариф 2')

tree.print_tree()

owner = tree.find_owner('79001112233')
print(f'Владелец номера 79001112233: {owner}')

popular_tariff = tree.most_popular_tariff()
print(f'Наиболее популярный тариф: {popular_tariff}')

```

Вывод работы программы представлен на рисунке 22.

⇒ Phone: 79001112233, Owner: Сергей Сергеев, Tariff: Тариф 1  
 Phone: 79001234567, Owner: Иван Иванов, Tariff: Тариф 1  
 Phone: 79003322110, Owner: Анна Аннова, Tariff: Тариф 3  
 Phone: 79005566777, Owner: Мария Мариева, Tariff: Тариф 2  
 Phone: 79007654321, Owner: Петр Петров, Tariff: Тариф 2  
 Владелец номера 79001112233: Сергей Сергеев  
 Наиболее популярный тариф: Тариф 1

Рисунок 22 – Вывод всей базы данных и наиболее востребованного тарифа

## СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Pyplot [Электронный ресурс]. URL: <https://matplotlib.org/stable/tutorials/pyplot.html> (дата обращения: 13.05.2024).
2. Habr [Электронный ресурс]. URL: <https://habr.com/ru/articles/> (дата обращения: 13.05.2024).
3. Stackoverflow [Электронный ресурс]. URL: <https://ru.stackoverflow.com/?tags=python> (дата обращения: 13.05.2024).
4. Лекции [Электронный ресурс]. URL: <https://up.omgtu.ru/index.php?r=remote/read/taskList&discipline=80000000000003EB&time=0> (дата обращения: 13.05.2024).