Министерство науки и высшего образования РФ
федеральное государственное автономное образовательное учреждение высшего образования

«Омский государственный технический университет»

Факультет информационных технологий и компьютерных систем

Кафедра «Прикладная математика и фундаментальная информатика»

# ЛАБОРАТОРНАЯ РАБОТА №1
## «Вероятностные алгоритмы»

по дисциплине «Программирование на языке *Python*»
Вариант 13

Студента Колодницкого Ильи Михайловича
фамилия, имя, отчество полностью

Курс 2 Группа ФИТ-232

Направление 02.03.02 Фундаментальная информатика и информационные технологии
код, наименование

Проверил ассистент
должность

Плескунов Д. А.
фамилия, инициалы

Выполнил 19.05.25
дата, подпись студента

19.05.2025 г.
дата, подпись преподавателя

г. Омск

2025 г.

**Задание №1 «фильтр Блума»**

1. Реализовать фильтр Блума с использованием стандартной библиотеки Python (при этом реализовать собственные хеш-функции).
2. Определить процент ложноположительных срабатываний конкретной реализации.
3. Оценить зависимость ложноположительных срабатываний относительно размерности массива m и числа хеш-функций k (таблица и графики зависимостей).
4. Произвести многофакторный дисперсионный анализ для выявления степени значимости следующих факторов: размерности массива m и числа хеш-функций k. (3 курс)
5. При реализации учесть возможность пересечения и объединения фильтров Блума.

**Решение**

Реализация кода для выполнения данной задачи на языке программирования Python представлен ниже:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

def hash_mult_add(item, m):
    hash_value = 0
    for char in str(item):
        hash_value = (hash_value * 31 + ord(char))
    return hash_value % m

def hash_xor(item, m):
    hash_value = 0
    for char in str(item):
        hash_value ^= (hash_value << 5) + ord(char) + (hash_value >> 2)
    return hash_value % m

class BloomFilter:
    def __init__(self, m, k, hash_fn):
        self.m = m
        self.k = k
        self.hash_fn = hash_fn
        self.bit_array = [0] * m

    def add(self, item):
        for i in range(self.k):
            index = self.hash_fn(f"{item}_{i}", self.m)
            self.bit_array[index] = 1

    def contains(self, item):
        for i in range(self.k):
            index = self.hash_fn(f"{item}_{i}", self.m)
```

```
            if self.bit_array[index] == 0:
                return False
        return True

    def __or__(self, other):
        if self.m != other.m or self.k != other.k:
            raise ValueError("Фильтры должны иметь одинаковые m и k")
        new_filter = BloomFilter(self.m, self.k, self.hash_fn)
        new_filter.bit_array = [a | b for a, b in zip(self.bit_array,
other.bit_array)]
        return new_filter

    def __and__(self, other):
        if self.m != other.m or self.k != other.k:
            raise ValueError("Фильтры должны иметь одинаковые m и k")
        new_filter = BloomFilter(self.m, self.k, self.hash_fn)
        new_filter.bit_array = [a & b for a, b in zip(self.bit_array,
other.bit_array)]
        return new_filter

def false_positive_rate(bloom_filter, added_elements, test_elements):
    for element in added_elements:
        bloom_filter.add(element)

    false_positives = 0
    for element in test_elements:
        if bloom_filter.contains(element):
            false_positives += 1

    return false_positives / len(test_elements)

def evaluate_false_positives(n, m_values, k_values, hash_fn):
    results = []
    for m in m_values:
        for k in k_values:
            bloom_filter = BloomFilter(m, k, hash_fn)
            added_elements = np.random.randint(0, 100000, n)
            test_elements = np.random.randint(100001, 200000, n)
            fp_rate = false_positive_rate(bloom_filter, added_elements,
test_elements)
            results.append((m, k, fp_rate))
    return results

n = 1000
m_values = [1000, 5000, 10000]
k_values = [3, 5, 7]

hash_functions = {
    "hash_mult_add": hash_mult_add,
    "hash_xor": hash_xor,
}

for hash_name, hash_fn in hash_functions.items():
    print(f"Результаты для хеш-функции: {hash_name}")
    results = evaluate_false_positives(n, m_values, k_values, hash_fn)
```

```
    df = pd.DataFrame(results, columns=["m", "k", "Ложноположительные
срабатывания"])
    print(df)

    plt.figure(figsize=(10, 6))
    for k in k_values:
        m_list = [m for (m, k_val, _) in results if k_val == k]
        fp_list = [fp for (_, k_val, fp) in results if k_val == k]
        plt.plot(m_list, fp_list, marker='o', label=f"k={k}")

    plt.xlabel("Размер массива (m)")
    plt.ylabel("Ложноположительные срабатывания")
    plt.title(f"Зависимость ложноположительных срабатываний от m и k
({hash_name})")
    plt.legend()
    plt.grid()
    plt.show()
```

Пример работы программы представлен на рисунке 1 и 2.

```
Результаты для хеш-функции: hash_mult_add
      m  k  Ложноположительные срабатывания
0   1000  3                           0.878
1   1000  5                           0.991
2   1000  7                           1.000
3   5000  3                           0.246
4   5000  5                           0.363
5   5000  7                           0.490
6  10000  3                           0.130
7  10000  5                           0.156
8  10000  7                           0.215
```

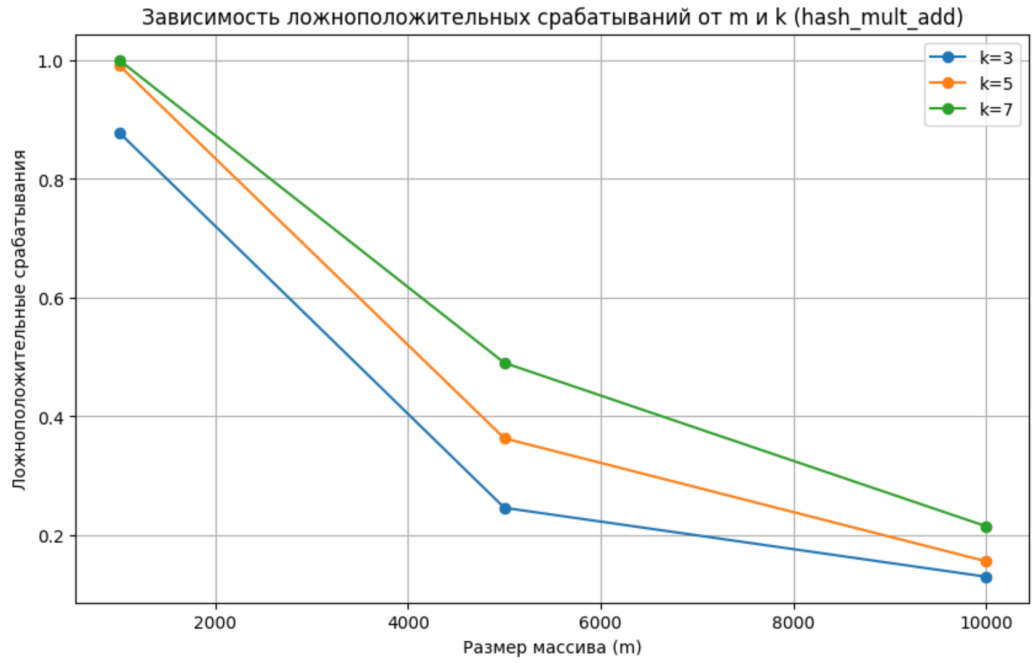



Рисунок 1 – работа программы для задания №1

```
Результаты для хеш-функции: hash_xor
       m  k  Ложноположительные срабатывания
0   1000  3                             0.891
1   1000  5                             0.986
2   1000  7                             0.994
3   5000  3                             0.189
4   5000  5                             0.255
5   5000  7                             0.387
6  10000  3                             0.081
7  10000  5                             0.099
8  10000  7                             0.153
```

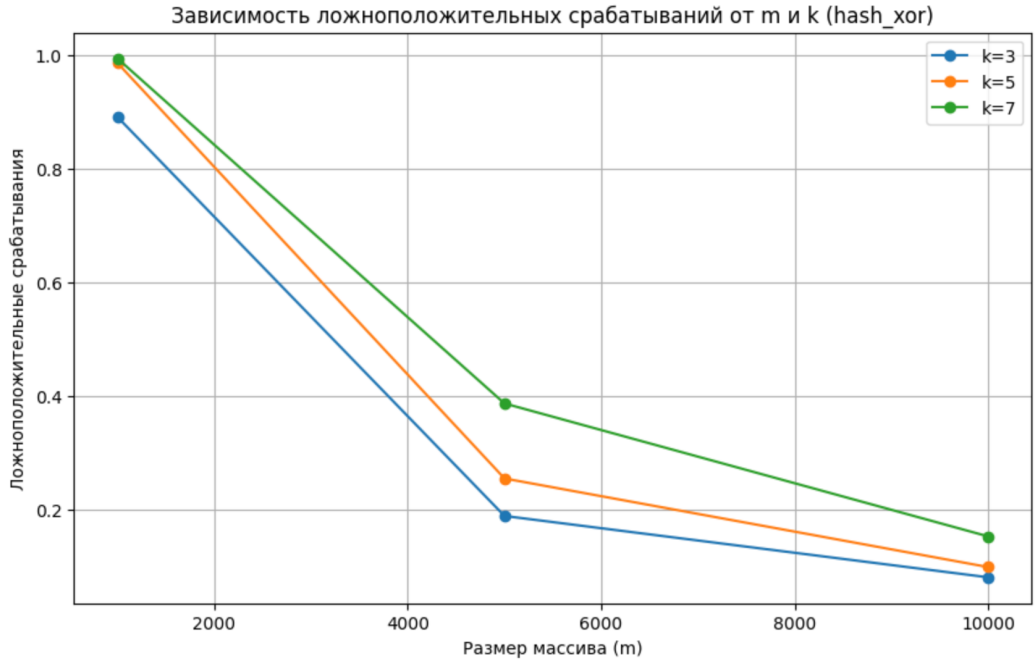



Рисунок 2 – работа программы для задания №1

```
bf1 = BloomFilter(m=1000, k=3, hash_fn=hash_xor)
bf2 = BloomFilter(m=1000, k=3, hash_fn=hash_xor)

words1 = ["apple", "banana", "cherry"]
for word in words1:
    bf1.add(word)

words2 = ["banana", "cherry", "date"]
for word in words2:
    bf2.add(word)

bf_union = bf1 | bf2

bf_intersection = bf1 & bf2

print("Объединенный фильтр:")
for word in ["apple", "banana", "cherry", "date", "elderberry"]:
    if bf_union.contains(word):
        print(f"Слово '{word}' вероятно принадлежит объединенному
множеству.")

print("\nПересечение фильтров:")
for word in ["apple", "banana", "cherry", "date", "elderberry"]:
    if bf_intersection.contains(word):
        print(f"Слово '{word}' вероятно принадлежит пересечению множеств.")
```

Пример работы программы представлен на рисунке 3.

```
Объединенный фильтр:
Слово 'apple' вероятно принадлежит объединенному множеству.
Слово 'banana' вероятно принадлежит объединенному множеству.
Слово 'cherry' вероятно принадлежит объединенному множеству.
Слово 'date' вероятно принадлежит объединенному множеству.

Пересечение фильтров:
Слово 'banana' вероятно принадлежит пересечению множеств.
Слово 'cherry' вероятно принадлежит пересечению множеств.
```

Рисунок 3 – работа программы для задания №1

**Задание №2 «фильтр Блума со счетом»**

1. Реализовать фильтр Блума со счетом
2. Определить процент ложноположительных срабатываний конкретной реализации на добавление и удаление.
3. Оценить зависимость ложноположительных срабатываний от гиперпараметров алгоритма (таблица и графики зависимостей).
4. При реализации учесть возможность пересечения и объединения фильтров Блума.

**Решение**

Реализация кода для выполнения данной задачи на языке программирования Python представлен ниже:

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

def hash_mult_add(item, m):
    hash_value = 0
    for char in str(item):
        hash_value = (hash_value * 31 + ord(char))
    return hash_value % m

def hash_xor(item, m):
    hash_value = 0
    for char in str(item):
        hash_value ^= (hash_value << 5) + ord(char) + (hash_value >> 2)
    return hash_value % m

class CountingBloomFilter:
    def __init__(self, m, k, hash_fn):
        self.m = m
        self.k = k
        self.hash_fn = hash_fn
```

```
        self.counter_array = [0] * m

    def add(self, item):
        for i in range(self.k):
            index = self.hash_fn(f"{item}_{i}", self.m)
            self.counter_array[index] += 1

    def contains(self, item):
        for i in range(self.k):
            index = self.hash_fn(f"{item}_{i}", self.m)
            if self.counter_array[index] == 0:
                return False
        return True

    def remove(self, item):
        for i in range(self.k):
            index = self.hash_fn(f"{item}_{i}", self.m)
            if self.counter_array[index] > 0:
                self.counter_array[index] -= 1

    def __or__(self, other):
        if self.m != other.m or self.k != other.k:
            raise ValueError("Фильтры должны иметь одинаковые m и k")
        new_filter = CountingBloomFilter(self.m, self.k, self.hash_fn)
        new_filter.counter_array = [a + b for a, b in zip(self.counter_array,
other.counter_array)]
        return new_filter

    def __and__(self, other):
        if self.m != other.m or self.k != other.k:
            raise ValueError("Фильтры должны иметь одинаковые m и k")
        new_filter = CountingBloomFilter(self.m, self.k, self.hash_fn)
        new_filter.counter_array = [min(a, b) for a, b in
zip(self.counter_array, other.counter_array)]
        return new_filter

def false_positive_rate(bloom_filter, added_elements, test_elements):
    for element in added_elements:
        bloom_filter.add(element)

    false_positives = 0
    for element in test_elements:
        if bloom_filter.contains(element):
            false_positives += 1

    return false_positives / len(test_elements)

def evaluate_false_positives(n, m_values, k_values, hash_fn):
    results = []
    for m in m_values:
        for k in k_values:
            bloom_filter = CountingBloomFilter(m, k, hash_fn)
            added_elements = np.random.randint(0, 100000, n)
            test_elements = np.random.randint(100001, 200000, n)
            fp_rate = false_positive_rate(bloom_filter, added_elements,
test_elements)
```

```
            results.append((m, k, fp_rate))
    return results

n = 1000
m_values = [1000, 5000, 10000]
k_values = [3, 5, 7]

hash_functions = {
    "hash_mult_add": hash_mult_add,
    "hash_xor": hash_xor,
}

for hash_name, hash_fn in hash_functions.items():
    print(f"Результаты для хеш-функции: {hash_name}")
    results = evaluate_false_positives(n, m_values, k_values, hash_fn)

    df = pd.DataFrame(results, columns=["m", "k", "Ложноположительные
срабатывания"])
    print(df)

    plt.figure(figsize=(10, 6))
    for k in k_values:
        m_list = [m for (m, k_val, _) in results if k_val == k]
        fp_list = [fp for (_, k_val, fp) in results if k_val == k]
        plt.plot(m_list, fp_list, marker='o', label=f"k={k}")

    plt.legend()
    plt.grid()
    plt.show()
```

Пример работы программы представлен на рисунке 4 и 5.

```
Результаты для хеш-функции: hash_mult_add
       m  k  Ложноположительные срабатывания
0   1000  3                           0.867
1   1000  5                           0.967
2   1000  7                           0.989
3   5000  3                           0.239
4   5000  5                           0.379
5   5000  7                           0.502
6  10000  3                           0.098
7  10000  5                           0.164
8  10000  7                           0.222
```

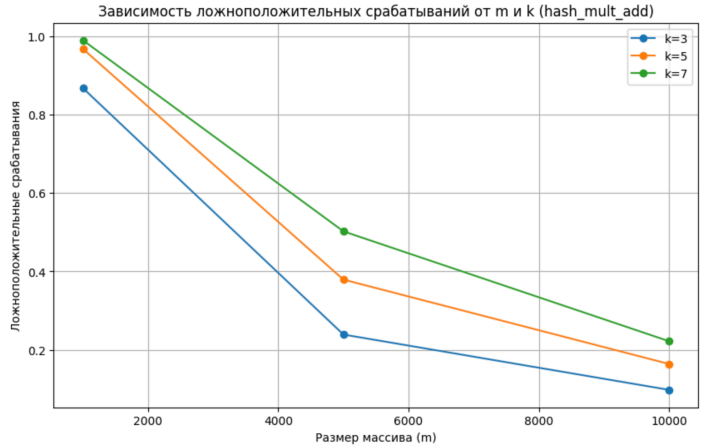



Рисунок 4 – работа программы для задания №2

```
Результаты для хеш-функции: hash_xor
      m  k  Ложноположительные срабатывания
0   1000  3                           0.884
1   1000  5                           0.978
2   1000  7                           0.995
3   5000  3                           0.197
4   5000  5                           0.283
5   5000  7                           0.388
6  10000  3                           0.075
7  10000  5                           0.099
8  10000  7                           0.142
```

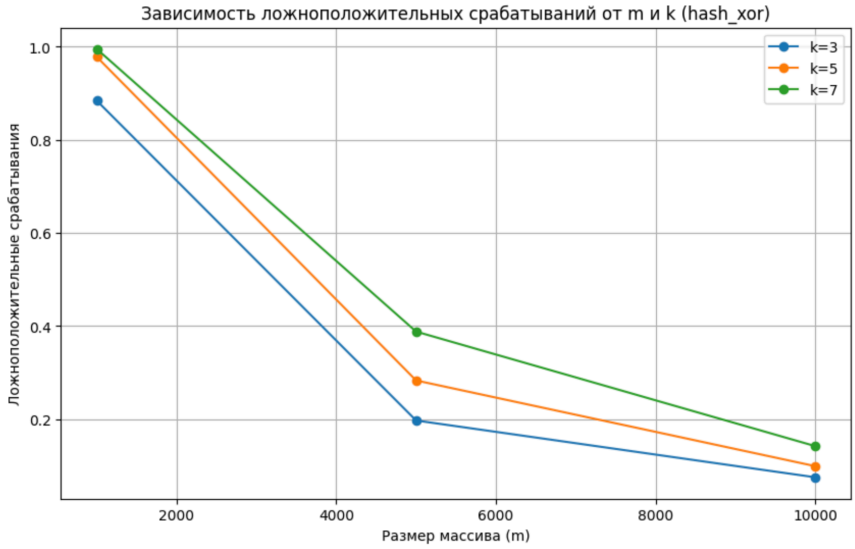



Рисунок 5 – работа программы для задания №2

```python
bf1 = CountingBloomFilter(m=1000, k=3, hash_fn=hash_xor)
bf2 = CountingBloomFilter(m=1000, k=3, hash_fn=hash_xor)

words1 = ["apple", "banana", "cherry"]
for word in words1:
    bf1.add(word)

words2 = ["banana", "cherry", "date"]
for word in words2:
    bf2.add(word)

bf_union = bf1 | bf2

bf_intersection = bf1 & bf2

print("Объединенный фильтр:")
for word in ["apple", "banana", "cherry", "date", "elderberry"]:
    if bf_union.contains(word):
        print(f"Слово '{word}' вероятно принадлежит объединенному
множеству.")

print("\nПересечение фильтров:")
for word in ["apple", "banana", "cherry", "date", "elderberry"]:
    if bf_intersection.contains(word):
        print(f"Слово '{word}' вероятно принадлежит пересечению множеств.")
```

Пример работы программы представлен на рисунке 6.

```
Объединенный фильтр:
Слово 'apple' вероятно принадлежит объединенному множеству.
Слово 'banana' вероятно принадлежит объединенному множеству.
Слово 'cherry' вероятно принадлежит объединенному множеству.
Слово 'date' вероятно принадлежит объединенному множеству.

Пересечение фильтров:
Слово 'banana' вероятно принадлежит пересечению множеств.
Слово 'cherry' вероятно принадлежит пересечению множеств.
```

Рисунок 6 – работа программы для задания №2

**Задание №3 «HyperLogLog»**

1. Реализовать HyperLogLog.
2. Определить процент ложноположительных срабатываний конкретной реализации.
3. Оценить зависимость ложноположительных срабатываний от гиперпараметров алгоритма (таблица и графики зависимостей).

**Решение**

Реализация кода для выполнения данной задачи на языке программирования Python представлен ниже:

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

def hash_mult_add(item, m):
    hash_value = 0
    for char in str(item):
        hash_value = (hash_value * 31 + ord(char))
    return hash_value % m

def hash_xor(item, m):
    hash_value = 0
    for char in str(item):
        hash_value ^= (hash_value << 5) + ord(char) + (hash_value >> 2)
    return hash_value % m

class HyperLogLog:
    def __init__(self, b=4, hash_fn=hash_mult_add):
        self.b = b
        self.m = 2 ** b
        self.registers = [0] * self.m
        self.alpha = self._get_alpha()
        self.hash_fn = hash_fn

    def _get_alpha(self):
        if self.m == 16:
            return 0.673
```

```
        elif self.m == 32:
            return 0.697
        elif self.m == 64:
            return 0.709
        else:
            return 0.7213 / (1 + 1.079 / self.m)

    def _hash(self, item):
        return self.hash_fn(item, 1 << 64)

    def add(self, item):
        hash_value = self._hash(item)
        index = hash_value & (self.m - 1)
        w = hash_value >> self.b
        self.registers[index] = max(self.registers[index],
self._count_leading_zeros(w) + 1)

    def _count_leading_zeros(self, w):
        if w == 0:
            return 64 - self.b
        return bin(w)[2:].zfill(64 - self.b).index('1')

    def count(self):
        harmonic_mean = sum(2 ** -r for r in self.registers)
        estimate = self.alpha * self.m ** 2 / harmonic_mean

        if estimate <= 2.5 * self.m:
            zeros = self.registers.count(0)
            if zeros != 0:
                return self.m * np.log(self.m / zeros)
        elif estimate > (1 << 64) / 30:
            return -(1 << 64) * np.log(1 - estimate / (1 << 64))
        return estimate

def evaluate_hyperloglog_accuracy(n, b_values, hash_fn):
    results = []
    for b in b_values:
        hll = HyperLogLog(b=b, hash_fn=hash_fn)
        elements = np.random.randint(0, 1000000, n)
        unique_elements = set(elements)
        for element in elements:
            hll.add(element)
        estimated_count = hll.count()
        true_count = len(unique_elements)
        error = abs(estimated_count - true_count) / true_count
        results.append((b, estimated_count, true_count, error))
    return results

n = 100000
b_values = [4, 6, 8, 10]

hash_functions = {
    "hash_mult_add": hash_mult_add,
    "hash_xor": hash_xor,
}
```

```
for hash_name, hash_fn in hash_functions.items():
    print(f"Результаты для хеш-функции: {hash_name}")
    results = evaluate_hyperloglog_accuracy(n, b_values, hash_fn)

    df = pd.DataFrame(results, columns=["b", "Оценка", "Истинное значение",
"Ошибка"])
    print(df)

    plt.figure(figsize=(10, 6))

    m_list = [2 ** b for b in b_values]
    error_list = [error for (b, est, true, error) in results]

    plt.plot(m_list, error_list, marker='o', label=f"{hash_name}")

    plt.xlabel("Размер массива (m = 2^b)")
    plt.ylabel("Ошибка оценки")
    plt.title(f"Зависимость ошибки оценки от размера массива (m)
({hash_name})")
    plt.legend()
    plt.grid()
    plt.show()

hll = HyperLogLog(b=4, hash_fn=hash_mult_add)

words = ["apple", "banana", "cherry", "date", "elderberry"]
for word in words:
    hll.add(word)

estimated_count = hll.count()
print(f"Оценка количества уникальных элементов: {estimated_count}")
```

Пример работы программы представлен на рисунке 7.

```
True
False
False
```

Рисунок 7 – работа программы для задания №3

**Задание №4 «Quotient filter»**

1. Реализовать Фильтр коэффициентов (Quotient filter).
2. Определить процент ложноположительных срабатываний конкретной реализации.
3. Оценить зависимость ложноположительных срабатываний от гиперпараметров алгоритма (таблица и графики зависимостей).

**Решение**

Реализация кода для выполнения данной задачи на языке программирования Python представлен ниже:

```
class QuotientFilter:
    def __init__(self, q, r):

        self.q = q
        self.r = r
        self.size = 1 << q
        self.table = [None] * self.size
        self.occupied = [False] * self.size
        self.runends = [False] * self.size


    def _hash(self, item):
        if isinstance(item, str):
            item = item.encode()
        hash_val = hash(item) & ((1 << (self.q + self.r)) - 1)
        quotient = hash_val >> self.r
        remainder = hash_val & ((1 << self.r) - 1)
        return quotient, remainder

    def insert(self, item):
        quotient, remainder = self._hash(item)

        if self.occupied[quotient]:
            start = quotient
            while start > 0 and self.occupied[start - 1]:
                start -= 1

            pos = start
            while pos < quotient:
                if self.table[pos] is None:
                    break
                pos += 1
            else:
                while pos < self.size and self.occupied[pos] and
self.table[pos] is not None:
                    pos += 1

            if pos >= self.size:
                pos = 0
                while pos < quotient and self.occupied[pos] and
self.table[pos] is not None:
                    pos += 1

            if pos < quotient:
                for i in range(pos, quotient):
                    if self.table[i] is None:
                        self.table[i] = self.table[i-1]
                        self.runends[i] = self.runends[i-1]
                        self.runends[i-1] = False
                self.table[quotient] = remainder
                self.runends[quotient] = True
            else:
                self.table[pos] = remainder
                self.runends[pos] = True
```

```
            else:
                self.table[quotient] = remainder
                self.runends[quotient] = True

            self.occupied[quotient] = True

        def lookup(self, item):
            quotient, remainder = self._hash(item)

            if not self.occupied[quotient]:
                return False

            start = quotient
            while start > 0 and self.occupied[start - 1]:
                start -= 1

            pos = start
            while pos < self.size and self.occupied[pos]:
                if self.table[pos] == remainder:
                    return True
                if self.runends[pos]:
                    break
                pos += 1

            return False

        def delete(self, item):
            quotient, remainder = self._hash(item)

            if not self.occupied[quotient]:
                return False

            start = quotient
            while start > 0 and self.occupied[start - 1]:
                start -= 1

            pos = start
            found_pos = -1
            while pos < self.size and self.occupied[pos]:
                if self.table[pos] == remainder:
                    found_pos = pos
                if self.runends[pos]:
                    break
                pos += 1

            if found_pos == -1:
                return False

            self.table[found_pos] = None

            if found_pos == quotient:
                has_other = False
                pos = start
                while pos < self.size and self.occupied[pos]:
                    if pos != found_pos and (self._get_quotient(pos) ==
quotient):
```

```
                    has_other = True
                    break
                if self.runends[pos]:
                    break
                pos += 1

            if not has_other:
                self.occupied[quotient] = False
                self.runends[quotient] = False

        return True

    def _get_quotient(self, pos):
        start = pos
        while start > 0 and self.occupied[start - 1]:
            start -= 1

        current = start
        quotient = None
        while current <= pos:
            if self.runends[current]:
                quotient = current
            current += 1

        return quotient

    def __contains__(self, item):
        return self.lookup(item)

    def __repr__(self):
        return f"QuotientFilter(q={self.q}, r={self.r}, size={self.size})"

qf = QuotientFilter(10, 4)

qf.insert("712cetbg")
qf.insert("39gyfhau")
qf.insert("ra4j5pgoipongdae")

print("712cetbg" in qf)
print("39gyfhau" in qf)
print("ra4j5pgoipongdae" in qf)

qf.delete("ra4j5pgoipongdae")
print("ra4j5pgoipongdae" in qf)
```

Пример работы программы представлен на рисунке 8.

```
True
True
True
False
```

Рисунок 8 – работа программы для задания №4

```
import random
import string

def generate_random_string(length=10):
    return ''.join(random.choices(string.ascii_letters + string.digits,
k=length))

def measure_false_positive_rate(q, r, num_inserted, num_tested):

    qf = QuotientFilter(q, r)

    inserted_elements = set()
    while len(inserted_elements) < num_inserted:
        item = generate_random_string()
        if item not in inserted_elements:
            qf.insert(item)
            inserted_elements.add(item)

    tested_elements = set()
    while len(tested_elements) < num_tested:
        item = generate_random_string()
        if item not in inserted_elements and item not in tested_elements:
            tested_elements.add(item)

    false_positives = 0
    for item in tested_elements:
        if item in qf:
            false_positives += 1

    false_positive_rate = (false_positives / num_tested) * 100
    return false_positive_rate

Q = 10
R = 6
NUM_INSERTED = 500
NUM_TESTED = 10000

fp_rate = measure_false_positive_rate(Q, R, NUM_INSERTED, NUM_TESTED)
print(f"Процент ложноположительных срабатываний: {fp_rate:.2f}%")
```

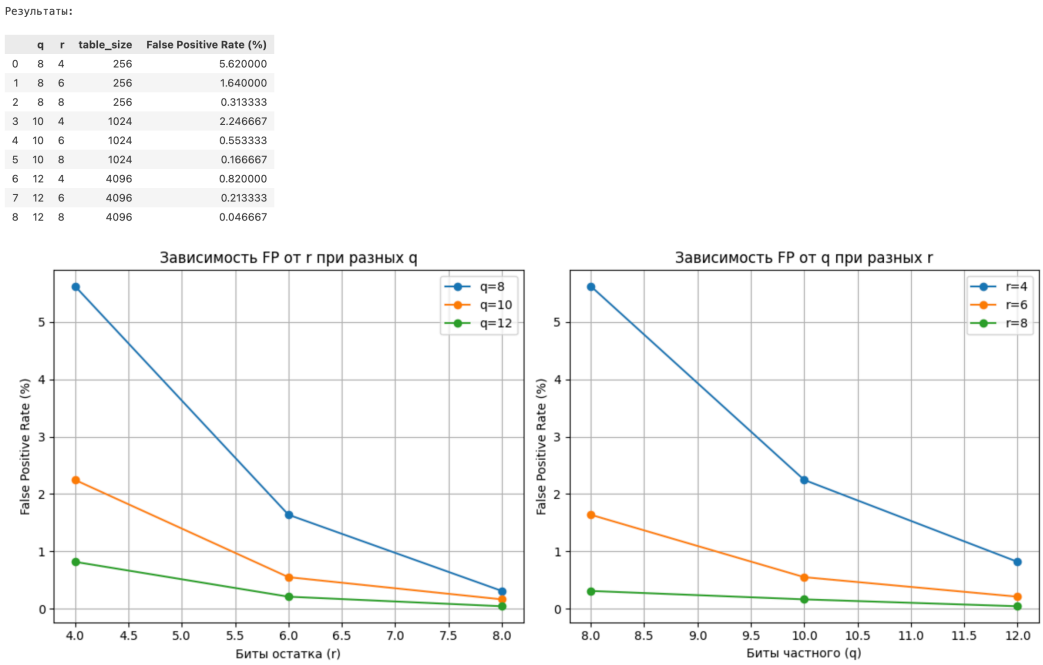
```
Процент ложноположительных срабатываний: 0.56%
```

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import random
import string
```

```
def generate_random_string(length=10):
    return ''.join(random.choices(string.ascii_letters + string.digits,
k=length))

def analyze_quotient_filter_performance():
    q_values = [8, 10, 12]
    r_values = [4, 6, 8]
    num_inserted = 500
    num_tested = 5000
    trials = 3

    results = []

    for q in q_values:
        for r in r_values:
            if q + r > 64:
                continue

            total_fp = 0

            for _ in range(trials):
                qf = QuotientFilter(q, r)

                inserted_elements = set()
                while len(inserted_elements) < num_inserted:
                    item = generate_random_string()
                    if item not in inserted_elements:
                        qf.insert(item)
                        inserted_elements.add(item)

                tested_elements = set()
                while len(tested_elements) < num_tested:
                    item = generate_random_string()
                    if item not in inserted_elements and item not in
tested_elements:
                        tested_elements.add(item)

                false_positives = 0
                for item in tested_elements:
                    if item in qf:
                        false_positives += 1

                total_fp += (false_positives / num_tested) * 100

            avg_fp = total_fp / trials

            results.append({
                'q': q,
                'r': r,
                'table_size': 2 ** q,
                'False Positive Rate (%)': avg_fp
            })

    df = pd.DataFrame(results)
```

```
    print("\nРезультаты:")
    display(df[['q', 'r', 'table_size', 'False Positive Rate (%)']]
            .sort_values(['q', 'r']))

    plt.figure(figsize=(12, 5))

    plt.subplot(1, 2, 1)
    for q_val in q_values:
        subset = df[df['q'] == q_val]
        plt.plot(subset['r'], subset['False Positive Rate (%)'],
                 marker='o', label=f'q={q_val}')
    plt.legend()
    plt.grid(True)

    plt.subplot(1, 2, 2)
    for r_val in r_values:
        subset = df[df['r'] == r_val]
        plt.plot(subset['q'], subset['False Positive Rate (%)'],
                 marker='o', label=f'r={r_val}')
    plt.legend()
    plt.grid(True)

    plt.tight_layout()
    plt.show()

    return df

results_df = analyze_quotient_filter_performance()
```

Пример работы программы представлен на рисунке 9.



Результаты:

| | q | r | table_size | False Positive Rate (%) |
|---|---|---|---|---|
| 0 | 8 | 4 | 256 | 5.620000 |
| 1 | 8 | 6 | 256 | 1.640000 |
| 2 | 8 | 8 | 256 | 0.313333 |
| 3 | 10 | 4 | 1024 | 2.246667 |
| 4 | 10 | 6 | 1024 | 0.553333 |
| 5 | 10 | 8 | 1024 | 0.166667 |
| 6 | 12 | 4 | 4096 | 0.820000 |
| 7 | 12 | 6 | 4096 | 0.213333 |
| 8 | 12 | 8 | 4096 | 0.046667 |



Рисунок 9 – работа программы для задания №4

**Задание №5 «Count-Min Sketch»**

1. Реализовать Count-Min Sketch.

2. Определить процент ложноположительных срабатываний конкретной реализации.
3. Оценить зависимость ложноположительных срабатываний от гиперпараметров алгоритма (таблица и графики зависимостей).

**Решение**

Реализация кода для выполнения данной задачи на языке программирования Python представлен ниже:

```
import matplotlib.pyplot as plt
import numpy as np
import random

class CountMinSketch:
    def __init__(self, width, depth):
        self.width = width
        self.depth = depth
        self.table = np.zeros((depth, width), dtype=np.int32)
        self.seeds = [random.randint(0, 1024) for _ in range(depth)]

    def _hash(self, item, seed):
        return (hash(str(item)) ^ seed) % self.width

    def update(self, item, count=1):
        for i in range(self.depth):
            pos = self._hash(item, self.seeds[i])
            self.table[i][pos] += count

    def estimate(self, item):
        return min(self.table[i][self._hash(item, self.seeds[i])]
                   for i in range(self.depth))

random.seed(42)
data = [random.randint(1, 100) for _ in range(10000)]

true_counts = {}
for num in data:
    true_counts[num] = true_counts.get(num, 0) + 1

widths = [50, 100, 200, 500, 1000]
depths = [3, 5, 7]

plt.figure(figsize=(14, 6))

plt.subplot(1, 2, 1)
for depth in depths:
    errors = []
    for width in widths:
        cms = CountMinSketch(width, depth)
        for num in data:
            cms.update(num)

        sample_size = min(100, len(true_counts))
```

```
            sample = random.sample(list(true_counts.keys()), sample_size)
            total_error = 0
            for num in sample:
                total_error += cms.estimate(num) - true_counts[num]
            avg_error = total_error / sample_size
            errors.append(avg_error)

        plt.plot(widths, errors, marker='o', label=f'Depth={depth}')

plt.xlabel('Width of Sketch')
plt.ylabel('Average Overestimation Error')
plt.title('Count-Min Sketch: Error vs Width')
plt.legend()
plt.grid()

plt.subplot(1, 2, 2)
cms = CountMinSketch(200, 5)
for num in data:
    cms.update(num)

sample_size = min(200, len(true_counts))
sample = random.sample(list(true_counts.keys()), sample_size)
errors = []
for num in sample:
    errors.append(cms.estimate(num) - true_counts[num])

plt.hist(errors, bins=30, alpha=0.7)
plt.grid()

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

Пример работы программы представлен на рисунке 10.

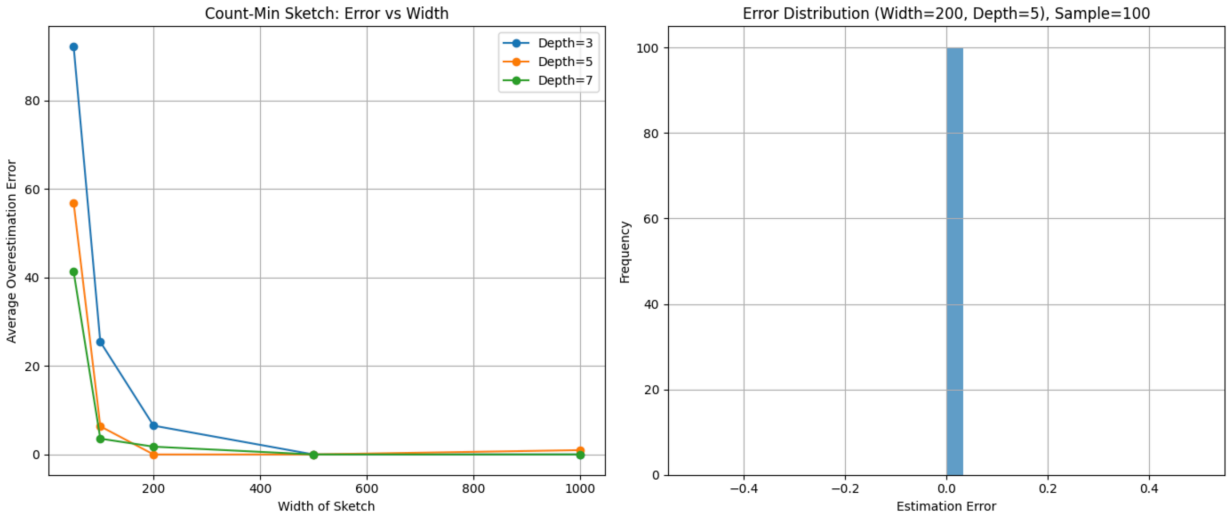



Рисунок 10 – работа программы для задания №5

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

```
words = ["apple", "banana", "cherry", "date", "elderberry", "fig", "grape"]
test_data = [random.choice(words) for _ in range(1000)]

cms = CountMinSketch(width=50, depth=3)

for word in test_data:
    cms.update(word)

exact_counts = {}
for word in test_data:
    if word not in exact_counts:
        exact_counts[word] = 0
    exact_counts[word] += 1

print("{:<10} {:<10} {:<10}".format("Word", "Estimate", "Real"))
print("-" * 30)
for word in words:
    estimated = cms.estimate(word)
    real = exact_counts.get(word, 0)
    print("{:<10} {:<10} {:<10}".format(word, estimated, real))

errors = [cms.estimate(word) - exact_counts.get(word, 0) for word in words]
plt.figure(figsize=(10, 4))
plt.bar(words, errors, color='skyblue')
plt.axhline(0, color='red', linestyle='--', linewidth=1)
plt.grid(axis='y', alpha=0.3)
```

Пример работы программы представлен на рисунке 11.

```
Word       Estimate   Real
------------------------------
apple      144        144
banana     154        154
cherry     121        121
date       160        160
elderberry 120        120
fig        171        171
grape      130        130
```

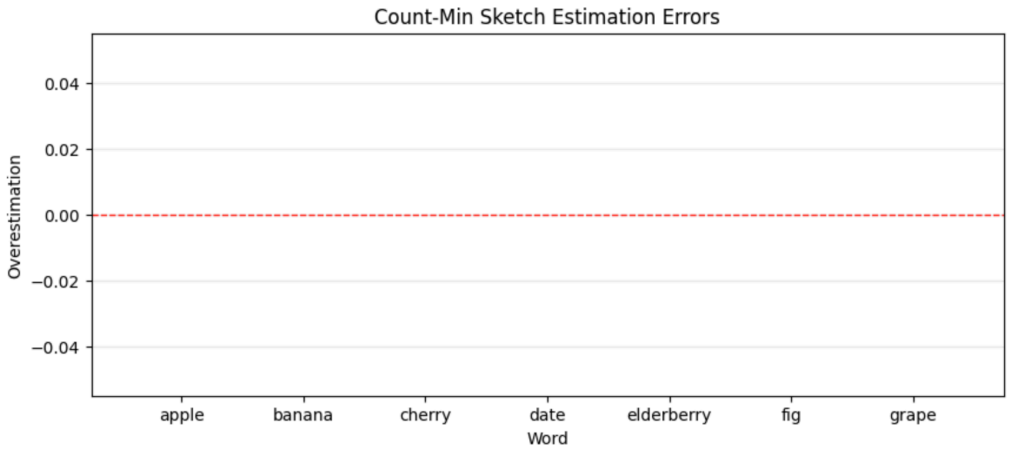



Рисунок 11 – работа программы для задания №5

**Задание №6 «MinHash»**

1. Реализовать MinHash.

2. Определить процент ложноположительных срабатываний конкретной реализации.
3. Оценить зависимость ложноположительных срабатываний от гиперпараметров алгоритма (таблица и графики зависимостей).

**Решение**

Реализация кода для выполнения данной задачи на языке программирования Python представлен ниже:

```
import matplotlib.pyplot as plt
import numpy as np
import random

class MinHash:
    def __init__(self, num_hashes):
        self.num_hashes = num_hashes
        self.a_params = [random.randint(1, 2**32) for _ in range(num_hashes)]
        self.b_params = [random.randint(1, 2**32) for _ in range(num_hashes)]
        self.min_values = np.full(num_hashes, np.inf)

    def _hash(self, item, a, b):
        item_hash = hash(str(item)) & 0xFFFFFFFF
        return (a * item_hash + b) & 0xFFFFFFFF

    def update(self, item):
        for i in range(self.num_hashes):
            h = self._hash(item, self.a_params[i], self.b_params[i])
            if h < self.min_values[i]:
                self.min_values[i] = h

    def jaccard(self, other):
        return np.mean(self.min_values == other.min_values)

random.seed(42)
data1 = set(random.sample(range(1, 10001), 1000))
data2 = set(random.sample(range(5000, 15001), 1000))

intersection = len(data1 & data2)
union = len(data1 | data2)
true_jaccard = intersection / union


num_hashes_list = [10, 50, 100, 200, 500]

plt.figure(figsize=(14, 6))

plt.subplot(1, 2, 1)
errors = []
for num_hashes in num_hashes_list:

    mh1 = MinHash(num_hashes)
    mh2 = MinHash(num_hashes)
```

```
    for item in data1:
        mh1.update(item)
    for item in data2:
        mh2.update(item)

    estimated_jaccard = mh1.jaccard(mh2)
    error = abs(estimated_jaccard - true_jaccard)
    errors.append(error)
    print(f"Хеш-функций: {num_hashes}, Оценка: {estimated_jaccard:.4f},
Истинное: {true_jaccard:.4f}, Ошибка: {error:.4f}")

plt.plot(num_hashes_list, errors, marker='o', color='blue')
plt.grid(True)

plt.subplot(1, 2, 2)
num_trials = 100
num_hashes = 100
estimates = []

for _ in range(num_trials):
    mh1 = MinHash(num_hashes)
    mh2 = MinHash(num_hashes)

    for item in data1:
        mh1.update(item)
    for item in data2:
        mh2.update(item)

    estimates.append(mh1.jaccard(mh2))
```

Пример работы программы представлен на рисунке 12.

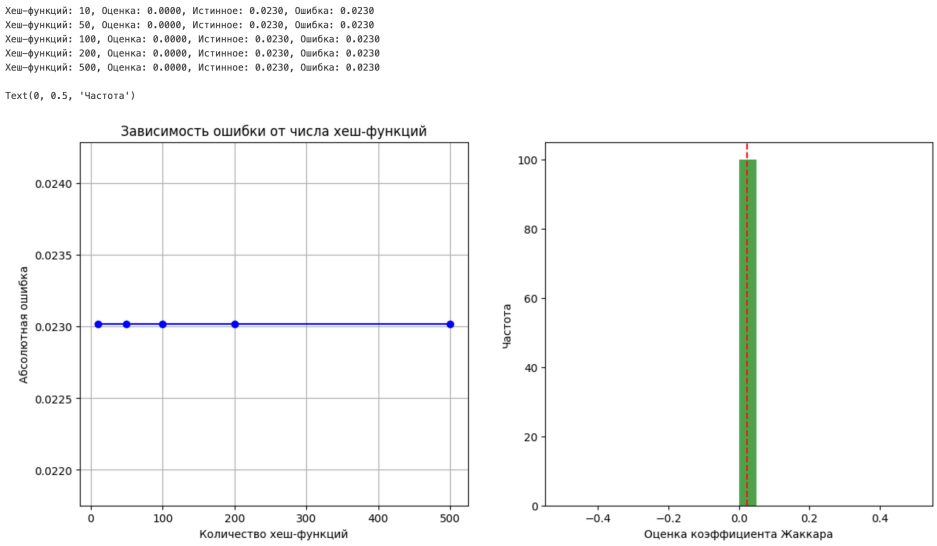


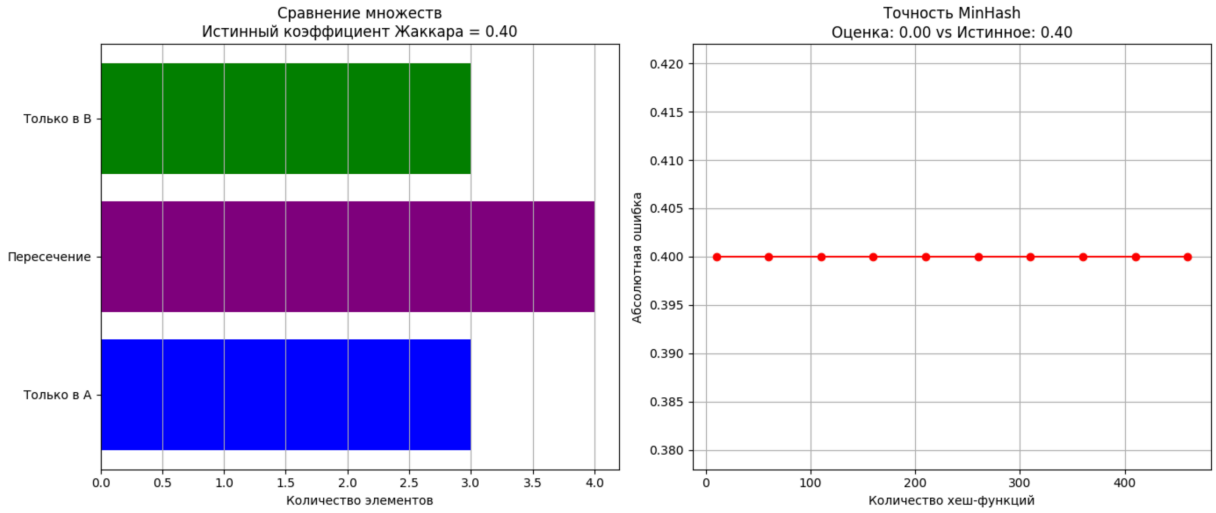

Рисунок 12 – работа программы для задания №6

```
import matplotlib.pyplot as plt
import numpy as np
import random
```

```
set_A = {'яблоко', 'банан', 'апельсин', 'груша', 'киви', 'манго', 'ананас'}
set_B = {'банан', 'апельсин', 'виноград', 'арбуз', 'киви', 'персик', 'манго'}

intersection = set_A & set_B
only_A = set_A - set_B
only_B = set_B - set_A
true_jaccard = len(intersection) / len(set_A | set_B)

mh_A = MinHash(100)
mh_B = MinHash(100)

for item in set_A:
    mh_A.update(item)
for item in set_B:
    mh_B.update(item)

estimated_jaccard = mh_A.jaccard(mh_B)

plt.figure(figsize=(14, 6))

plt.subplot(1, 2, 1)
plt.barh(['Только в A', 'Пересечение', 'Только в B'],
        [len(only_A), len(intersection), len(only_B)],
        color=['blue', 'purple', 'green'])
plt.title(f"Сравнение множеств\nИстинный коэффициент Жаккара =
{true_jaccard:.2f}")
plt.xlabel("Количество элементов")
plt.grid(axis='x')

plt.subplot(1, 2, 2)
num_hashes_range = range(10, 501, 50)
errors = []

for num_hashes in num_hashes_range:
    mh1 = MinHash(num_hashes)
    mh2 = MinHash(num_hashes)

    for item in set_A:
        mh1.update(item)
    for item in set_B:
        mh2.update(item)

    errors.append(abs(mh1.jaccard(mh2) - true_jaccard))

plt.plot(num_hashes_range, errors, marker='o', color='red')
plt.xlabel('Количество хеш-функций')
plt.ylabel('Абсолютная ошибка')
plt.title(f'Точность MinHash\nОценка: {estimated_jaccard:.2f} vs Истинное:
{true_jaccard:.2f}')
plt.grid(True)

plt.tight_layout()
plt.show()

print("Элементы только в A:", only_A)
```

```
print("Элементы только в B:", only_B)
print("Общие элементы:", intersection)
print(f"Истинный коэффициент Жаккара: {true_jaccard:.4f}")
print(f"Оценка MinHash: {estimated_jaccard:.4f}")
print(f"Абсолютная ошибка: {abs(estimated_jaccard - true_jaccard):.4f}")
```

Пример работы программы представлен на рисунке 13.




```
Элементы только в A: {'груша', 'ананас', 'яблоко'}
Элементы только в B: {'арбуз', 'персик', 'виноград'}
Общие элементы: {'манго', 'апельсин', 'банан', 'киви'}
Истинный коэффициент Жаккара: 0.4000
Оценка MinHash: 0.0000
Абсолютная ошибка: 0.4000
```

Рисунок 13 – работа программы для задания №6