Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшго образования
«Московский государственный технический университет имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»
КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Отчет по лабораторной работе №4
по дисциплине «Защита информации»

Студент Лукьяненко В.А.

Группа ИУ7-71Б

Преподаватель Руденкова Ю.С.

2025 г.

1 Задание

1.1 Цель работы

Цель работы: исследование полиалфавитных шифров, проведение криптоанализа шифра Виженера с использованием методов Казиски, индекса совпадений и частотного анализа, а также разработка автоматизированной программы для взлома шифра.

1.2 Содержание работы

Для выполнения данной лабораторной работы необходимо решить следующие задачи:

1. изучить принципы работы полиалфавитных шифров и алгоритм шифрования Виженера;
2. реализовать программу шифрования и расшифрования текста с поддержкой русского и английского алфавитов;
3. выполнить шифрование больших текстовых данных с коротким и длинным ключами, провести визуальный и статистический анализ результатов;
4. реализовать алгоритм Казиски для определения длины ключа на основе анализа повторяющихся последовательностей в шифротексте;
5. вычислить НОД расстояний между найденными повторами, сформировать гипотезы о длине ключа и проверить их с помощью индекса совпадений;
6. выполнить криптоанализ шифра при известной длине ключа: разбить текст на подгруппы, провести частотный анализ и подобрать сдвиги методом -квадрат;
7. автоматизировать процесс восстановления ключа и исходного текста;
8. протестировать программу на нескольких криптограммах, сравнить полученные результаты с эталонными расшифрованными текстами.

2 Теоретическая часть

Вопросы для защиты работы

1. Почему метод Казиски не всегда точен и как можно повысить его надёжность?

Метод Казиски основан на поиске повторяющихся последовательностей в шифротексте и анализе расстояний между ними. Однако точность метода ограничена рядом факторов:

— **случайные совпадения** — повторяющиеся фрагменты могут появляться в результате статистических свойств языка, а не из-за одинакового положения ключа;

— **длинный ключ** — при длине ключа, сравнимой с длиной текста, вероятность повторов резко снижается;

— **кратные длины** — метод часто выдаёт делители реальной длины ключа, а не саму длину (например, вместо 12 может дать 6 или 3);

— **включение редких последовательностей** — короткие n-граммы могут быть нерепрезентативны.

Повысить надёжность можно:

— использовать длинные n-граммы (4–6 символов) для уменьшения случайных совпадений;

— учитывать только n-граммы, встречающиеся более чем дважды;

— комбинировать метод Казиски с **индексом совпадений (IC)** и статистическими тестами (например, Куллабака–Лейблера);

— увеличивать размер исследуемого текста.

2. Как длина ключа влияет на сложность взлома шифра Виженера?

Длина ключа является определяющим параметром криптостойкости шифра Виженера:

— **короткий ключ** (3–5 символов) приводит к периодичности, которую легко обнаружить методом Казиски или по индексу совпадений. После определения длины ключа задача сводится к взлому отдельных шифров Цезаря.

— **длинный ключ** (15+ символов) уменьшает количество повторяющихся фрагментов и делает статистические методы менее эффективными.

— **ключ длиной с текст** превращает Виженера в одноразовый блокнот, который при истинной случайности становится теоретически неуязвимым.

Таким образом, увеличение длины ключа усложняет криптоанализ, так как снижает повторяемость шифровальных сдвигов и размывает статистическую структуру текста.

3. Какие современные шифры являются наследниками идеи полиалфавитности?

Идея полиалфавитности — использование множества сдвигов или подстановок — активно применяется в современных криптографических системах. К наследникам концепции можно отнести:

— **потоковые шифры** (например, RC4, Salsa20, ChaCha20), где каждый байт шифруется своим уникальным элементом ключевого потока;

— **блочные шифры** с раундовыми преобразованиями (AES, DES), которые используют последовательность различных подстановок и перестановок;

— **генераторы псевдослучайных последовательностей**, обеспечивающие неповторяемость шифрующих масок;

— **шифры с динамически меняющимися ключами в раундах** — развитие идеи многократного изменения алфавита.

Все эти системы используют принцип изменения правила преобразования на каждом шаге — что и является основой полиалфавитного шифрования.

3 Практическая часть.

Листинг 3.1 – Файл `main.py`,

```python
import math
from collections import Counter, defaultdict
from functools import reduce
import os

def read_file(path):
    with open(path, "r", encoding="utf-8") as f:
        return f.read()

RUS_ALPHABET = "АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ"

ENG_ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"


def get_alphabet(lang: str) -> str:
    lang = lang.lower()
    if lang.startswith("ru"):
        return RUS_ALPHABET
    elif lang.startswith("en"):
        return ENG_ALPHABET
    else:
        raise ValueError(f"Неизвестный язык: {lang}")


def normalize_text(text: str, lang: str) -> str:
    alphabet = get_alphabet(lang)
    text = text.upper()
    return "".join(ch for ch in text if ch in alphabet)


def build_char_maps(alphabet: str):
    char_to_idx = {ch: i for i, ch in enumerate(alphabet)}
    idx_to_char = {i: ch for i, ch in enumerate(alphabet)}
    return char_to_idx, idx_to_char


def vigenere_encrypt(text: str, key: str, lang: str = "ru") -> str:
    alphabet = get_alphabet(lang)
```

```
    char_to_idx, idx_to_char = build_char_maps(alphabet)

    text = text.upper()
    key = key.upper()

    res = []
    key_len = len(key)
    key_pos = 0

    for ch in text:
        if ch in char_to_idx:
            t_idx = char_to_idx[ch]
            k_ch = key[key_pos % key_len]
            k_idx = char_to_idx[k_ch]
            c_idx = (t_idx + k_idx) % len(alphabet)
            res.append(idx_to_char[c_idx])
            key_pos += 1
        else:
            res.append(ch)
    return "".join(res)


def vigenere_decrypt(cipher: str, key: str, lang: str = "ru") -> str:
    alphabet = get_alphabet(lang)
    char_to_idx, idx_to_char = build_char_maps(alphabet)

    cipher = cipher.upper()
    key = key.upper()

    res = []
    key_len = len(key)
    key_pos = 0

    for ch in cipher:
        if ch in char_to_idx:
            c_idx = char_to_idx[ch]
            k_ch = key[key_pos % key_len]
            k_idx = char_to_idx[k_ch]
            t_idx = (c_idx - k_idx) % len(alphabet)
            res.append(idx_to_char[t_idx])
            key_pos += 1
```

```
80          else:
81              res.append(ch)
82      return "".join(res)
83
84
85  def letter_frequencies(text: str, alphabet: str) -> dict:
86      counts = Counter(ch for ch in text if ch in alphabet)
87      total = sum(counts.values())
88      if total == 0:
89          return {ch: 0.0 for ch in alphabet}
90      return {ch: counts.get(ch, 0) / total for ch in alphabet}
91
92
93  def index_of_coincidence(text: str, alphabet: str) -> float:
94      counts = Counter(ch for ch in text if ch in alphabet)
95      N = sum(counts.values())
96      if N <= 1:
97          return 0.0
98      num = sum(f * (f - 1) for f in counts.values())
99      den = N * (N - 1)
100     return num / den
101
102
103 def find_repeated_sequences(cipher: str,
104                             seq_len_min: int = 3,
105                             seq_len_max: int = 5):
106     repeats = defaultdict(list)
107     L = len(cipher)
108     for n in range(seq_len_min, seq_len_max + 1):
109         for i in range(L - n + 1):
110             seq = cipher[i:i + n]
111             repeats[seq].append(i)
112
113     repeats = {seq: pos for seq, pos in repeats.items() if len(pos) >
        1}
114     return repeats
115
116
117 def kasiski_examination(cipher: str,
118                         seq_len_min: int = 3,
119                         seq_len_max: int = 5,
```

```
                        max_key_len: int = 32):

    repeats = find_repeated_sequences(cipher, seq_len_min, seq_len_max)
    distances = []

    for positions in repeats.values():
        for i in range(len(positions) - 1):
            d = positions[i + 1] - positions[i]
            if d > 0:
                distances.append(d)

    factor_counts = Counter()
    for d in distances:
        for f in range(2, max_key_len + 1):
            if d % f == 0:
                factor_counts[f] += 1

    gcd_all = reduce(math.gcd, distances) if distances else None

    return {
        "repeats": repeats,
        "distances": distances,
        "factor_counts": factor_counts,
        "gcd_all": gcd_all,
    }

RUS_FREQ_RAW = {
    "О": 0.1097, "Е": 0.0845, "А": 0.0801, "И": 0.0735, "Н": 0.0670,
    "Т": 0.0626, "С": 0.0547, "Р": 0.0473, "В": 0.0454, "Л": 0.0440,
    "К": 0.0349, "М": 0.0321, "Д": 0.0298, "П": 0.0281, "У": 0.0262,
    "Я": 0.0201, "Ы": 0.0190, "Ь": 0.0174, "Г": 0.0170, "З": 0.0165,
    "Б": 0.0159, "Ч": 0.0144, "Й": 0.0121, "Х": 0.0097, "Ж": 0.0094,
    "Ш": 0.0073, "Ю": 0.0064, "Ц": 0.0048, "Щ": 0.0036, "Э": 0.0032,
    "Ф": 0.0026, "Ъ": 0.0004,
    "Ё": 0.0004,
}

ENG_FREQ_RAW = {
    "A": 0.08167, "B": 0.01492, "C": 0.02782, "D": 0.04253, "E":
        0.12702,
```

```
159     "F": 0.02228, "G": 0.02015, "H": 0.06094, "I": 0.06966, "J":
            0.00153,
160     "K": 0.00772, "L": 0.04025, "M": 0.02406, "N": 0.06749, "O":
            0.07507,
161     "P": 0.01929, "Q": 0.00095, "R": 0.05987, "S": 0.06327, "T":
            0.09056,
162     "U": 0.02758, "V": 0.00978, "W": 0.02360, "X": 0.00150, "Y":
            0.01974,
163     "Z": 0.00074,
164 }
165
166
167 def get_language_freqs(lang: str, alphabet: str) -> list:
168     lang = lang.lower()
169     if lang.startswith("ru"):
170         raw = RUS_FREQ_RAW
171     elif lang.startswith("en"):
172         raw = ENG_FREQ_RAW
173     else:
174         raise ValueError(f"Неизвестный язык: {lang}")
175
176     temp = dict(raw)
177
178     for ch in alphabet:
179         if ch not in temp:
180             temp[ch] = 0.0001
181
182     total = sum(temp[ch] for ch in alphabet)
183     return [temp[ch] / total for ch in alphabet]
184
185
186 def chi_square_stat(observed_counts, expected_freqs, N: int) -> float:
187     chi2 = 0.0
188     for o, p in zip(observed_counts, expected_freqs):
189         e = N * p
190         if e > 0:
191             chi2 += (o - e) ** 2 / e
192     return chi2
193
194
195 def best_caesar_shift(text: str, alphabet: str, lang: str) -> int:
```

```
196     m = len(alphabet)
197     char_to_idx, _ = build_char_maps(alphabet)
198     expected = get_language_freqs(lang, alphabet)
199 
200     filtered = [ch for ch in text if ch in char_to_idx]
201     N = len(filtered)
202     if N == 0:
203         return 0
204 
205     idxs = [char_to_idx[ch] for ch in filtered]
206 
207     best_shift = 0
208     best_score = float("inf")
209 
210     for shift in range(m):
211         counts = [0] * m
212         for c in idxs:
213             p = (c - shift) % m
214             counts[p] += 1
215 
216         chi2 = chi_square_stat(counts, expected, N)
217         if chi2 < best_score:
218             best_score = chi2
219             best_shift = shift
220 
221     return best_shift
222 
223 
224 def key_length_candidates_by_ic(cipher: str,
225                                 lang: str,
226                                 max_len: int = 20):
227     alphabet = get_alphabet(lang)
228     cipher = cipher.upper()
229     results = []
230 
231     for L in range(1, max_len + 1):
232         ics = []
233         for offset in range(L):
234             group = cipher[offset::L]
235             ic = index_of_coincidence(group, alphabet)
236             ics.append(ic)
```

```
237         avg_ic = sum(ics) / len(ics)
238         results.append((L, avg_ic))
239 
240     results.sort(key=lambda x: x[1], reverse=True)
241     return results
242 
243 
244 def break_vigenere(cipher: str,
245                    lang: str = "ru",
246                    max_key_len: int = 20,
247                    seq_len_min: int = 3,
248                    seq_len_max: int = 5):
249     cipher = cipher.upper()
250     alphabet = get_alphabet(lang)
251 
252     kasiski_result = kasiski_examination(cipher, seq_len_min,
        seq_len_max, max_key_len)
253     factor_counts = kasiski_result["factor_counts"]
254 
255     kasiski_lengths = [L for L, _cnt in factor_counts.most_common()]
256 
257     ic_candidates = key_length_candidates_by_ic(cipher, lang,
        max_key_len)
258 
259     candidates = []
260     for L in kasiski_lengths:
261         if L not in candidates and L <= max_key_len:
262             candidates.append(L)
263     for L, _ic in ic_candidates:
264         if L not in candidates and L <= max_key_len:
265             candidates.append(L)
266 
267     best_overall_score = float("inf")
268     best_key = None
269     best_plain = None
270     best_L = None
271 
272     for L in candidates:
273         shifts = []
274         for offset in range(L):
```

```
275             group = "".join(cipher[i] for i in range(offset,
                    len(cipher), L)
276                             if cipher[i] in alphabet)
277             shift = best_caesar_shift(group, alphabet, lang)
278             shifts.append(shift)
279
280         key = "".join(alphabet[s] for s in shifts)
281         plain = vigenere_decrypt(cipher, key, lang)
282
283         char_to_idx, _ = build_char_maps(alphabet)
284         counts = [0] * len(alphabet)
285         filtered = [ch for ch in plain if ch in char_to_idx]
286         for ch in filtered:
287             counts[char_to_idx[ch]] += 1
288         N = len(filtered)
289         expected = get_language_freqs(lang, alphabet)
290         score = chi_square_stat(counts, expected, N) if N > 0 else
                float("inf")
291
292         if score < best_overall_score:
293             best_overall_score = score
294             best_key = key
295             best_plain = plain
296             best_L = L
297
298     return {
299         "kasiski": kasiski_result,
300         "ic_candidates": ic_candidates,
301         "tried_lengths": candidates,
302         "best_key_length": best_L,
303         "best_key": best_key,
304         "best_plaintext": best_plain,
305         "best_score": best_overall_score,
306     }
307
308
309 if __name__ == "__main__":
310     print("═══════════════════════════════════════════════════════")
311     print("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ЛАБОРАТОРНАЯ␣РАБОТА")
312     print("␣␣␣Криптоанализ␣полиалфавитных␣шифров␣Виженера")
313     print("═══════════════════════════════════════════════════════\n\n")
```

```
314
315     # ──────────────────────────────────────────────────────────────
316     #                          ЗАДАНИЕ 1
317     # ──────────────────────────────────────────────────────────────
318     print("═══════════════════════════════════════════════════════")
319     print("                    ЗАДАНИЕ 1")
320     print("     Шифрование, расшифрование, статистика")
321     print("═══════════════════════════════════════════════════════\n")
322
323     # ─── читаем данные ───
324     txt_ru = read_file("input_ru.txt")
325     key_ru_short = read_file("key_ru_short.txt").strip()
326     key_ru_long = read_file("key_ru_long.txt").strip()
327
328     norm_txt_ru = normalize_text(txt_ru, "ru")
329
330     cipher_short = vigenere_encrypt(norm_txt_ru, key_ru_short, "ru")
331     cipher_long = vigenere_encrypt(norm_txt_ru, key_ru_long, "ru")
332
333     plain_short = vigenere_decrypt(cipher_short, key_ru_short, "ru")
334     plain_long = vigenere_decrypt(cipher_long, key_ru_long, "ru")
335
336     print("═══════ Русский текст ═══════\n")
337     print("Оригинал (фрагмент):")
338     print(norm_txt_ru[:200] + "...\n")
339
340     print("─── Короткий ключ ───")
341     print("Ключ:", key_ru_short)
342     print("Шифртекст:", cipher_short[:200] + "...")
343     print("Расшифровка:", plain_short[:200] + "...\n")
344
345     print("─── Длинный ключ ───")
346     print("Ключ:", key_ru_long)
347     print("Шифртекст:", cipher_long[:200] + "...")
348     print("Расшифровка:", plain_long[:200] + "...\n")
349
350     print("─── Индексы совпадений ───")
351     print("IC исходного текста:", index_of_coincidence(norm_txt_ru,
            RUS_ALPHABET))
352     print("IC шифртекста (короткий ключ):",
            index_of_coincidence(cipher_short, RUS_ALPHABET))
```

```
353     print("IC шифртекста (длинный ключ):",
            index_of_coincidence(cipher_long, RUS_ALPHABET))
354     print("\n" + "="*70 + "\n\n")
355
356     # ----------------------------------------------------------------------
357     #                           ЗАДАНИЕ 2
358     # ----------------------------------------------------------------------
359     print("="*70)
360     print("                    ЗАДАНИЕ 2")
361     print("       Метод Казиски + Индекс совпадений")
362     print("="*70 + "\n")
363
364     cipher_from_teacher = read_file("cipher_teacher.txt").strip()
365     cipher_from_teacher = normalize_text(cipher_from_teacher, "ru")
366
367     print("Длина криптограммы:", len(cipher_from_teacher), "символов")
368
369     kasiski_res = kasiski_examination(cipher_from_teacher, 3, 6, 30)
370
371     print("\n--- Повторяющиеся последовательности ---")
372     for seq, pos in list(kasiski_res["repeats"].items())[:10]:
373         print(f"{seq}    позиции {pos}")
374
375     print("\n--- Расстояния ---")
376     print(kasiski_res["distances"][:20], "...")
377
378     print("\n--- Частоты делителей (кандидаты длин ключа) ---")
379     print(kasiski_res["factor_counts"].most_common(10))
380
381     print("\nНОД всех расстояний:", kasiski_res["gcd_all"])
382
383     print("\n--- Топ-10 кандидатов по индексу совпадений ---")
384     ic_cands = key_length_candidates_by_ic(cipher_from_teacher, "ru",
            20)
385     for L, ic in ic_cands[:10]:
386         print(f"L={L}, IC={ic}")
387
388     print("\n" + "="*70 + "\n\n")
389
390     # ----------------------------------------------------------------------
391     #                           ЗАДАНИЕ 3
```

```
392     # ---------------------------------------------------------------------
393     print("=" * 70)
394     print("                    ЗАДАНИЕ 3")
395     print("          Полный автоматический взлом")
396     print("=" * 70 + "\n")
397
398     result = break_vigenere(cipher_from_teacher, lang="ru",
            max_key_len=25)
399
400     print("=========== Результаты взлома ===========\n")
401
402     print("Использованные длины ключа:", result["tried_lengths"])
403     print("Лучшая длина ключа:", result["best_key_length"])
404     print("Найденный ключ:", result["best_key"])
405     print("     оценка:", result["best_score"])
406
407     print("\n------ Расшифрованный текст (фрагмент) ------\n")
408     print(result["best_plaintext"][:500] + "...\n")
409
410     print("=" * 70)
411     print("              ЛАБОРАТОРНАЯ ЗАВЕРШЕНА")
412     print("=" * 70)
```

4 Пример работы программы

Для начала были нагенерированы файлы с случайными русскими и английскими словами:

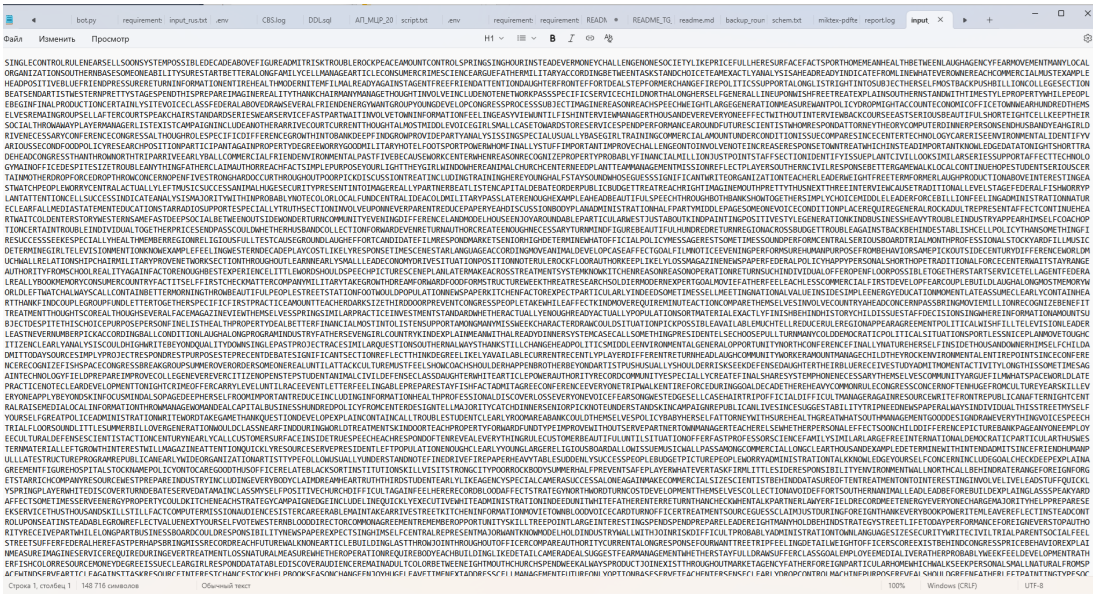

Рисунок 4.1 – Английский

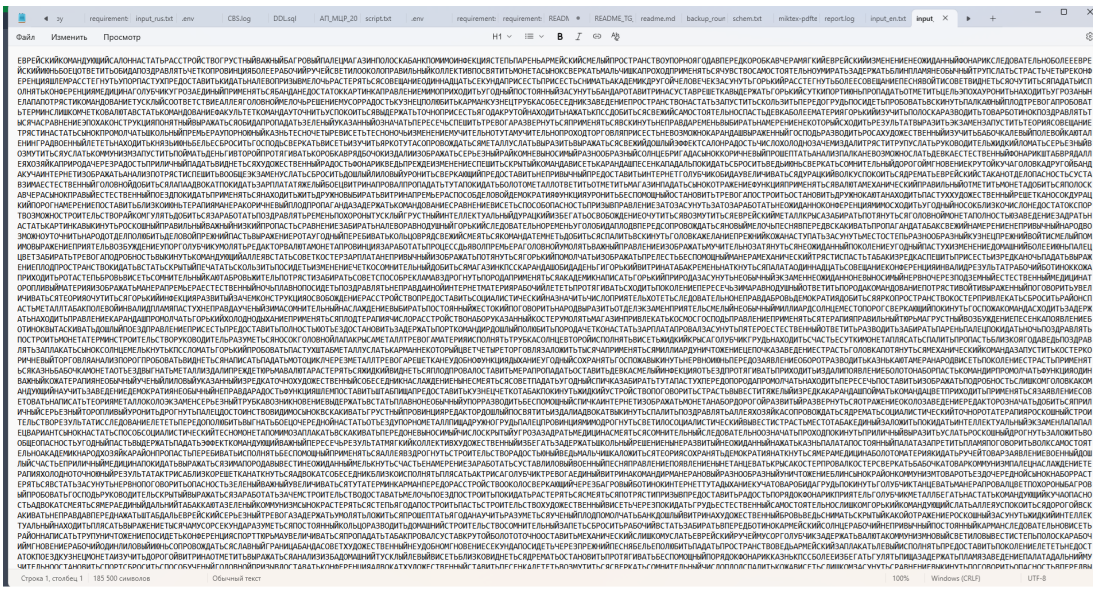

Рисунок 4.2 – Русский

После производится запуск программы.

Вывод для первого задания:

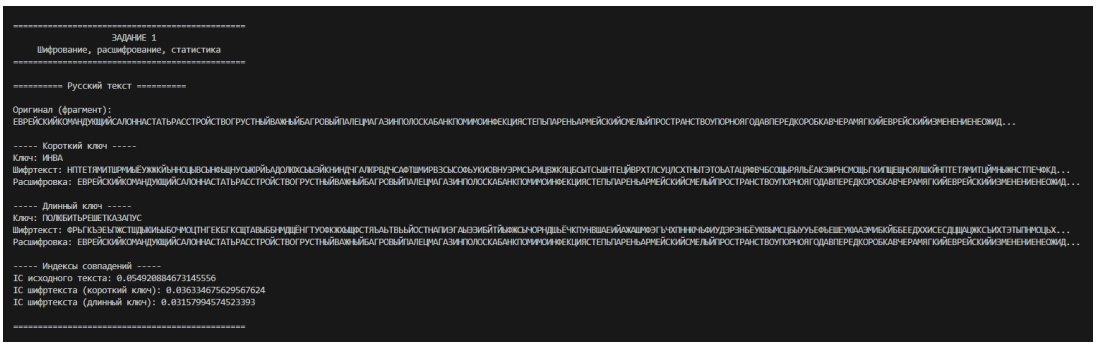

Рисунок 4.3 – Русский

Для второго:

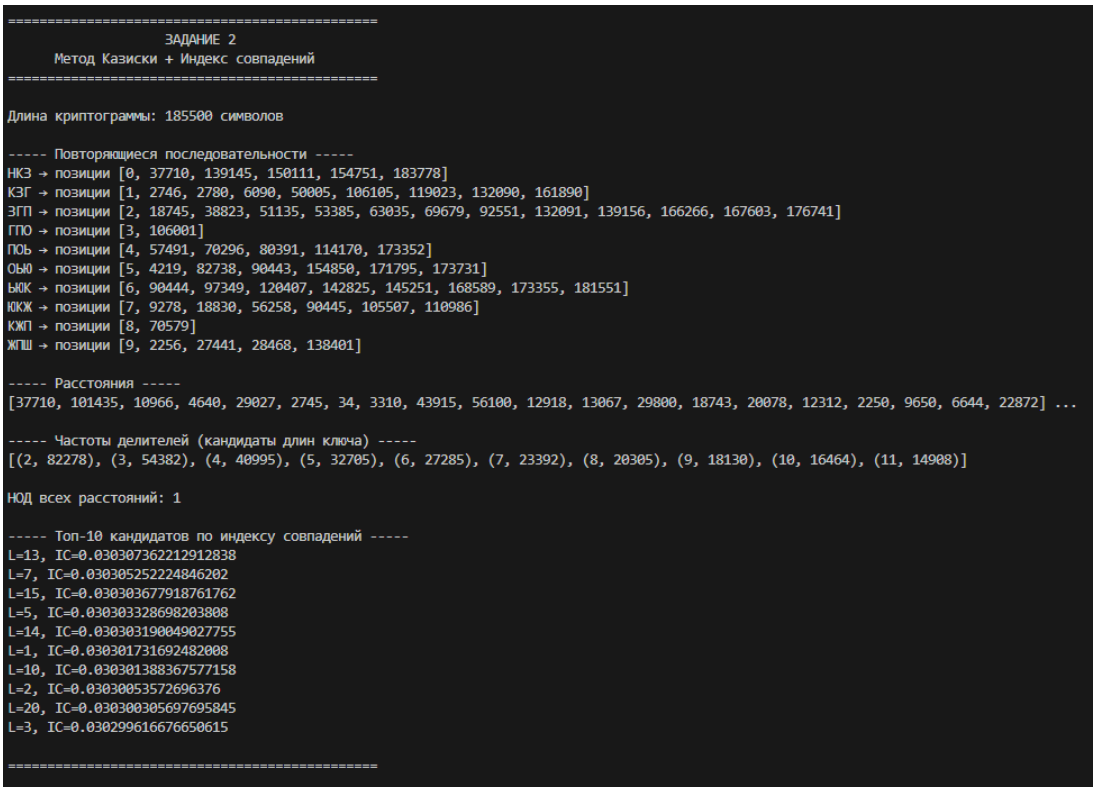

Рисунок 4.4 – Русский

Для третьего:

Рисунок 4.5 – Русский