Міністерство освіти і науки України Національний
університет «Одеська політехніка» Інститут
комп’ютерних систем
Кафедра інформаційних систем

Розрахунково-графічна робота
з дисципліни: «Теорія алгоритмів»
Тема: «Розробіть та реалізуйте алгоритми пошуку під рядка в тексті,на основі послідовного пошуку (грубої сили) та за методом Кнута-Морріса-Прата (КМП-алгоритм). Порівняйте обчислювальну складність обох алгоритмів на рядках та під рядках різної довжини.»

Варіант № 22

Виконав:
Студент групи АІ-243
Гаврилов О.В.
Перевірили: Смик
С. Ю.
Арсірій О.О.

Одеса 2023

Завдання: розробити та реалізувати алгоритми пошуку підрядка в тексті, на основі послідовного пошуку (грубої сили) та за методом Кнута-Морріса-Прата (КМП-алгоритм). Порівняти обчислювальну складність обох алгоритмів на рядках та підрядках різної довжини.

Мета роботи: розробити, реалізувати та експериментально дослідити ефективність (алгоритмічну складність) алгоритмів послідовного пошуку (груба сила) та Кнута-Морріса-Прата (КМП) для задачі пошуку підрядка, а також обґрунтувати переваги КМП-алгоритму на основі порівняння кількості базових операцій порівняння.

1. Опис вхідних і вихідних даних

Вхідні дані повинні бути достатніми для тестування обох алгоритмів у різних умовах, включаючи найгірший випадок:

- основний текст (T): Рядок символів довільної довжини N (генерується автоматично для тестів);
- підрядок-шаблон (P): Рядок символів довжини M (генерується автоматично, або задається вручну для спеціальних тестів);
- довжини рядків (N, M): Кількість елементів (символів) у тексті та підрядку, що використовуються для експериментального порівняння.

Вихідні дані необхідні для підтвердження коректності роботи алгоритмів та порівняння їхньої ефективності:

- позиції входження: Список початкових індексів, де підрядок P знайдено в тексті T;
- кількість порівнянь: Точна кількість базових операцій порівняння символів, виконаних кожним алгоритмом (основний показник

обчислювальної складності);

– час виконання алгоритмів: Фактичний час роботи кожного алгоритму на тестових наборах (для підтвердження складності);

– порівняльні графіки: Автоматично згенеровані графіки залежності кількості порівнянь від довжини тексту N та довжини підрядка M;

– побудова Префікс-функції (π): Додаткова інформація для КМП-алгоритму (структура, що використовується для зсувів).

2.1 Загальні відомості про алгоритм Алгоритм Кнута-Морріса-Пратта

Алгоритм КМП (Knuth-Morris-Pratt) — це високоефективний метод пошуку підрядка, який принципово відрізняється від Грубої Сили тим, що він уникає повторного сканування символів тексту T.

КМП забезпечує гарантовану лінійну часову складність O(N + M), де N — довжина тексту, а M — довжина шаблону. Щоб досягти цієї лінійності, алгоритм використовує Префікс-функцію (π, або lps-масив), яка будується на етапі попередньої обробки шаблону. Витрати на попередню обробку становлять O(M) часу і повністю виправдовуються економією часу на етапі пошуку, що має складність O(N).

Ключова ідея КМП: коли під час порівняння відбувається невідповідність (збій), алгоритм не зсуває шаблон на одну позицію і не починає порівняння знову. Натомість, він використовує інформацію з π-функції, щоб визначити, яка частина вже порівняного фрагмента тексту може бути префіксом на новій позиції. Це дозволяє здійснювати «розумний» зсув шаблону.

Розглянемо роботу алгоритму КМП (див. лістинг 2.1.1) та псевдокод (див. лістинг 2.1.2) . Ми будемо використовувати для прикладу:

- Рядок (T): "ABABDABACDABABCABAB" (Довжина N = 19)
- Підрядок (P): "ABABCABAB" (Довжина M = 9)

Лістинг 2.1.1 – Числове моделювання Алгоритму Кнута-Морріса-Пратта (КМП)

Крок 1.1: Побудова таблиці префіксів (LPS)
Підрядок (P): "ABABCABAB" (M=9)

i	P[i]	Власні префікси P[0...i]	Найдовший префікс, що є суфіксом	LPS[i]
0	A	-	-	0
1	B	"A"	-	0
2	A	"A", "AB"	"A"	1
3	B	"A", "AB", "ABA"	"AB"	2
4	C	"A", "AB", "ABA", "ABAB"	-	0
5	A	"A", "AB", "ABA", "ABAB", "ABABC"	"A"	1
6	B	"A", "AB", "ABA", "ABAB", "ABABCA"	"AB"	2
7	A	"A", "AB", "ABA", "ABAB", "ABABCAB"	"ABA"	3
8	B	"A", "AB", "ABA", "ABAB", "ABABCABA"	"ABAB"	4

Результат: LPS = [0, 0, 1, 2, 0, 1, 2, 3, 4]

Крок 1.2: Числове моделювання пошуку КМП
Рядок (T): A B A B D A B A C D A B A B C A B A B
Індекси (i): 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8

Підрядок (P): A B A B C A B A B (M=9)
LPS: [0,0,1,2,0,1,2,3,4]

Старт: i = 0, j = 0

i	j	T[i]	P[j]	Порівняння	Дія	Нові i, j	Примітка
0	0	A	A	Збіг	i++, j++	1, 1	
1	1	B	B	Збіг	i++, j++	2, 2	
2	2	A	A	Збіг	i++, j++	3, 3	
3	3	B	B	Збіг	i++, j++	4, 4	Збіг префікса "ABAB" (j=4)
4	4	D	C	Невідповідність	j = LPS[j-1]	4, 2	j = LPS[3] = 2. "Стрибок", i не змінюється.
4	2	D	A	Невідповідність	j = LPS[j-1]	4, 0	j = LPS[1] = 0. "Стрибок", i не змінюється.
4	0	D	A	Невідповідність	i++	5, 0	j=0, тому рухаємо i.
5	0	A	A	Збіг	i++, j++	6, 1	
6	1	B	B	Збіг	i++, j++	7, 2	
...	...	...	...	...	...	...	...
10	0	A	A	Збіг	i++, j++	11, 1	Початок повного збігу.
...	...	...	...	...	...	...	...
18	8	B	B	Збіг	i++, j++	19, 9	
-	9	-	-	j == M	Знайдено	-	Вивід: індекс = i - j = 19 - 9 = 10
19	9	-	-	Кінець циклу			i = N, пошук завершено.

Лістинг 2.1.2 – Псевдокод Алгоритму Кнута-Морріса-Пратта (КМП)

```
Function COMPUTE-LPS-ARRAY(P)
    M = LENGTH(P)
    Declare LPS[0...M-1]
    LPS[0] = 0
```

```
    length = 0 // Довжина попереднього найдовшого префікса-суфікса
    i = 1

    while i < M
        if P[i] == P[length]
            length = length + 1
            LPS[i] = length
            i = i + 1
        else
            if length != 0
                length = LPS[length - 1]
            else
                LPS[i] = 0
                i = i + 1
    return LPS

Function KMP-SEARCH(T, P)
    N = LENGTH(T)
    M = LENGTH(P)
    LPS = COMPUTE-LPS-ARRAY(P)

    i = 0 // Індекс для T
    j = 0 // Індекс для P

    while i < N
        if T[i] == P[j]
            i = i + 1
            j = j + 1

        if j == M
            print("Знайдено входження за індексом " + (i - j))
            j = LPS[j - 1] // Продовжуємо пошук

        else if i < N and T[i] != P[j]
            if j != 0
                j = LPS[j - 1] // "Стрибок" за допомогою LPS
            else
                i = i + 1 // Рухаємося по тексту
```

Детально про роботу псевдокоду (див. лістинг 2.1.2). Побудова LPS-масиву (COMPUTE-LPS-ARRAY). Ця функція створює допоміжний масив LPS. Вона проходить по підрядку P, починаючи з другого символу (i=1), і підтримує змінну length, що зберігає довжину поточного префікса-суфікса. У циклі порівнюються P[i] та P[length]. При збігу length збільшується, значення записується в LPS[i], і ми переходимо до наступного символу. При невідповідності, якщо length не нуль, ми "відкочуємося" до попереднього значення LPS[length-1], не змінюючи i. Якщо ж length нуль, записуємо 0 в LPS[i] і рухаємося далі.Основний пошук (KMP-SEARCH)

Функція використовує масив LPS для пошуку P в T. Індекси i (для тексту) та j (для підрядка) починаються з нуля. У циклі по тексту, якщо символи T[i] та P[j] збігаються, обидва індекси збільшуються. Якщо j досягає кінця підрядка (j == M), знайдено повний збіг, виводиться його позиція, а j оновлюється значенням LPS[j-1] для продовження пошуку. Якщо ж символи не збігаються, і ми вже просунулися по підрядку (j != 0), ми "перестрибуємо" індексом j на позицію LPS[j-1], не змінюючи i. Якщо невідповідність сталася на початку підрядка (j == 0), ми просто переходимо до наступного символу тексту.

Розглянемо просторову складність алгоритму КМП. Для роботи алгоритму використовується допоміжний масив LPS (Префікс-функція), розмір якого прямо пропорційний довжині шаблону M. Як наслідок, алгоритм КМП має просторову складність O(M).

Важливою властивістю КМП є те, що його продуктивність не залежить від структури вхідних даних у найгіршому випадку. Завдяки "розумному" зсуву, він уникає квадратичної затримки, на яку приречена Груба Сила.

2.2 Загальні відомості про алгоритм Алгоритм Грубої сили

Метод Грубої Сили (або повний перебір) — це найпростіший, найпряміший і часто найменш ефективний метод розв'язання задачі. Він є універсальною алгоритмічною парадигмою, яка полягає у систематичному перегляді всіх можливих кандидатів на розв'язок і перевірці кожного з них.

Ідея полягає в тому, щоб слідувати інструкціям задачі буквально, без спроб оптимізації чи використання складних структур даних. Якщо потрібно знайти об'єкт, який має певну властивість, метод грубої сили перевіряє кожен об'єкт у просторі пошуку, доки не знайде об'єкт з потрібною властивістю або не вичерпає простір пошуку.

Розглянемо псевдокод (див. лістинг 2.2.2) процедури BRUTE-FORCE-SEARCH (Груба Сила) та чисельне модулювання псевдокоду (див. лістинг 2.2.1).

Цей алгоритм виконує пошук підрядка без будь-якої попередньої обробки. Ініціалізація (Рядки 1-2): Визначаються довжини вхідного тексту T (позначається як N) та шаблону P (позначається як M). Цикл Зсуву (Рядок 3): Цикл for відповідає за зсув шаблону P і перевіряє всі можливі початкові позиції i у тексті T, від 0 до N-M. Ініціалізація Порівняння (Рядок 4): Усередині циклу зсуву ініціалізується змінна j (індекс шаблону) з 0. Цикл Порівняння (Рядки 5-6): Цикл while виконує порівняння символів від поточної позиції i. Він триває доти, доки символи тексту T[i+j] та шаблону P[j] збігаються, або поки ми не дійдемо до кінця шаблону (j < M). Успішне Входження (Рядки 7-8): Після завершення внутрішнього циклу while, якщо j дорівнює M, це означає, що знайдено повний збіг. Алгоритм повертає початковий індекс i. Збій та Зсув (Рязок 9): Якщо збіг не знайдено (тобто j < M), відбувається автоматичний зсув на одну позицію (i+1), і цикл for починає перевірку заново. Це є причиною квадратичної складності у найгіршому випадку, оскільки індекси тексту постійно перевіряються знову. Завершення (Рядок 9): Якщо цикл for завершується, а збіг не знайдено, функція повертає -1.

Лістинг 2.2.1 – Числове моделювання пошуку грубою силою

```
Рядок (T):    A B A B D A B A C D A B A B C A B A B
Індекси (i):  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8

Підрядок (P): A B A B C A B A B (M=9)

| i (Початок) | j (Зсув в P) | T[i+j] | P[j] | Порівняння | Дія | Примітка |
|---|---|---|---|---|---|---|
| **0** | 0 | A | A | Збіг | j++ | |
| 0 | 1 | B | B | Збіг | j++ | |
| 0 | 2 | A | A | Збіг | j++ | |
| 0 | 3 | B | B | Збіг | j++ | |
| 0 | 4 | D | C | **Невідповідність**| Перехід до наступного i | Збіглося 4
символи. |
|---|---|---|---|---|---|---|
| **1** | 0 | B | A | **Невідповідність**| Перехід до наступного i | Збіглося 0
символів. |
|---|---|---|---|---|---|---|
| **2** | 0 | A | A | Збіг | j++ | |
| 2 | 1 | B | B | Збіг | j++ | |
| 2 | 2 | D | A | **Невідповідність**| Перехід до наступного i | Збіглося 2
символи. |
|---|---|---|---|---|---|---|
| ... | ... | ... | ... | ... | ... | ... |
|---|---|---|---|---|---|---|
```

```
| **10** | 0 | A | A | Збіг | j++ | Початок потенційного збігу |
| 10 | 1 | B | B | Збіг | j++ | |
| ... | ... | ... | ... | ... | ... | |
| 10 | 8 | B | B | Збіг | j++ | |
| 10 | 9 | - | - | **j == M** | **Знайдено** | Вивід: індекс = 10 |
|---|---|---|---|---|---|---|
| ... | ... | ... | ... | ... | ... | Пошук триває до i = N - M |
```

Лістинг 2.2.2 – Псевдокод Алгоритму Грубої Сили (BF)

```
Function KMP-SEARCH(T, P)
  N = LENGTH(T)
  M = LENGTH(P)
  LPS = COMPUTE-LPS-ARRAY(P)

  i = 0
  j = 0

  while i < N
    if T[i] == P[j]
      i = i + 1
      j = j + 1

    if j == M
      return i - j

    else if i < N and T[i] != P[j]
      if j != 0
        j = LPS[j - 1]
      else
        i = i + 1

  return -1
```

Детально про роботу псевдокоду (див. лістинг 2.2.2). Алгоритм грубої сили (Brute-force). Функція BruteForceSearch використовує два вкладені цикли. Зовнішній цикл (for) перебирає всі можливі початкові позиції i в тексті T, від 0 до N-M. Для кожної позиції i запускається внутрішній цикл (while), який за допомогою індексу j послідовно порівнює символи підрядка P[j] із відповідними символами тексту T[i+j]. Цикл триває, доки символи збігаються і ми не вийшли за межі підрядка. Якщо внутрішній цикл пройшов до кінця підрядка (j == M), це означає повний збіг, і виводиться позиція i. У разі невідповідності символів внутрішній цикл переривається, і зовнішній цикл переходить до наступної початкової позиції i, знову починаючи перевірку з першого символу підрядка.

Часова складність алгоритму "Груба Сила" залежить від довжини тексту N та довжини шаблону M. Ми розглядаємо час виконання як функцію T(N, M).

Для обчислення загального часу виконання алгоритму Груба Сила (Brute Force), що має N-M+1 можливих зсувів, достатньо скласти час виконання кожного рядка в процедурі BRUTE-FORCE-SEARCH (порівняйте з псевдокодом):

$$T(N,M)=c_1+c_2+c_3(N-M+1)+c_4\sum_{i=1}^{N-M+1}t_i+c_5\sum_{i=1}^{N-M+1}(t_i-1)+c_6(N-M+1)+c_7(N-M+1)$$

Для обчислення часової складності, розглянемо обидва крайні випадки. Для найкращого випадку, коли усі елементи у вхідному масиві відсортовані, або коли шаблон знайдено на першій позиції тексту, час виконання буде лінійним, тобто O(N). Це відбувається тому, що внутрішній цикл порівнянь або одразу знаходить збіг, або швидко виявляє невідповідність, і алгоритм не витрачає багато часу на повторні перевірки. Для найгіршого випадку, коли вхідний масив відсортовано у зворотному порядку, або коли в тексті відбувається майже повний збіг шаблону на кожній можливій позиції (наприклад, T=a...ab, P=a...ab), час виконання становить O(N · M). У цьому сценарії внутрішній цикл while виконує M порівнянь для майже кожного з N зсувів, що призводить до квадратичного зростання часу виконання.

3. Програмна реалізація.

Програмний код на мові Python реалізує порівняння двох алгоритмів пошуку підрядка в тексті: Послідовного пошуку (Груба сила) та Алгоритму Кнута-Морріса-Пратта (КМП). Програма вимірює час виконання кожного алгоритму на автоматично згенерованих текстах і шаблонах різної довжини, а також будує графіки для порівняння їхньої обчислювальної складності.

```
import random
import time
import matplotlib.pyplot as plt
import string
import psutil
import os
import sys

def brute_force_search(text, pattern):
    N = len(text)
    M = len(pattern)
    for i in range(N - M + 1):
        j = 0
        while j < M and text[i + j] == pattern[j]:
            j += 1
        if j == M:
            return i
    return -1

def compute_lps(pattern):
    M = len(pattern)
    lps = [0] * M
    length = 0
    i = 1
    while i < M:
        if pattern[i] == pattern[length]:
            length += 1
            lps[i] = length
            i += 1
        else:
            if length != 0:
                length = lps[length - 1]
            else:
                lps[i] = 0
                i += 1
    return lps

def kmp_search(text, pattern):
    N = len(text)
    M = len(pattern)
    lps = compute_lps(pattern)
    i = 0
    j = 0
    while i < N:
```

```
        if pattern[j] == text[i]:
            i += 1
            j += 1
        if j == M:
            return i - j
        elif i < N and pattern[j] != text[i]:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1
    return -1

def measure_time(search_function, text, pattern):
    start_time = time.time()
    search_function(text, pattern)
    end_time = time.time()
    return end_time - start_time

def measure_kmp_memory(pattern):
    lps = compute_lps(pattern)
    memory_bytes = sys.getsizeof(lps)
    return memory_bytes / 1024

def generate_worst_case(N, M, alphabet):
    if not alphabet: alphabet = 'a'
    char = alphabet[0]
    break_char = alphabet[1] if len(alphabet) > 1 else chr(ord(char) + 1)
    if N < M: M = N - 1
    if M <= 0: M = 1
    prefix = char * (M - 1)
    pattern = prefix + break_char
    text = (char * (N - M)) + pattern
    return text[:N], pattern

def generate_random_case(N, M, alphabet):
    if not alphabet: alphabet = string.ascii_lowercase
    text = ''.join(random.choice(alphabet) for _ in range(N))
    start_index = random.randint(0, max(0, N - M))
    pattern = text[start_index : start_index + M] if start_index + M <= N else
text[-M:]
    return text, pattern

def generate_good_case(N, M, alphabet):
    if not alphabet: alphabet = string.ascii_lowercase
    if N < M: M = N - 1
    if M <= 0: M = 1
    pattern = ''.join(random.choice(alphabet) for _ in range(M))
    remainder = ''.join(random.choice(alphabet) for _ in range(N - M))
    text = pattern + remainder
    return text[:N], pattern

def print_search_result(text, pattern, index, N_size):
    if N_size <= 100:
        print("\n--- ДЕТАЛІ ПОШУКУ ---")
        print(f"Довжина тексту N: {len(text)}, Шаблону M: {len(pattern)}")
        print(f"Шаблон P: '{pattern}'")
        if index != -1:
            start = max(0, index - 5)
            end = min(len(text), index + len(pattern) + 5)
            fragment = text[start:end]
            print(f"Збіг знайдено за індексом: {index}")
            print(f"Фрагмент T (позиція збігу): {fragment}")
        else:
            print("Збіг не знайдено.")
```

```
        print("--------------------\n")

def compare_N_impact(input_N_sizes, M_const, alphabet, scenario):
    brute_force_times = []
    kmp_times = []
    kmp_memory_usage = []
    print(f"\n--- ТЕСТ: N ЗМІНЮЄТЬСЯ, M ФІКСОВАНО ({M_const}). Сценарій:
{scenario.upper()} ---")
    for N in input_N_sizes:
        if N < M_const:
             print(f"Пропущено N={N}: N має бути більшим за M={M_const}")
             continue
        if scenario == "worst":
            text, pattern = generate_worst_case(N, M_const, alphabet)
        elif scenario == "random":
            text, pattern = generate_random_case(N, M_const, alphabet)
        elif scenario == "good":
            text, pattern = generate_good_case(N, M_const, alphabet)
        else:
            text, pattern = generate_random_case(N, M_const, alphabet)
        bf_time = measure_time(brute_force_search, text, pattern)
        kmp_time = measure_time(kmp_search, text, pattern)
        used_memory = measure_kmp_memory(pattern)
        brute_force_times.append(bf_time)
        kmp_times.append(kmp_time)
        kmp_memory_usage.append(used_memory)
        if N == input_N_sizes[0]:
            print_search_result(text, pattern, kmp_search(text, pattern), N)
        print(f"Input Size: {N}")
        print(f"Brute Force Time: {bf_time:.8f} seconds")
        print(f"KMP Time: {kmp_time:.8f} seconds")
        print(f"KMP Memory Usage: {used_memory:.3f} KB")
        print("----------")
    plt.figure(figsize=(10, 6))
    tested_N_sizes = [N for N in input_N_sizes if N >= M_const]
    plt.plot(tested_N_sizes, brute_force_times, label="Груба Сила O(N*M)",
marker='o', color='red')
    plt.plot(tested_N_sizes, kmp_times, label="КМП O(N+M)", marker='o',
color='blue')
    plt.xlabel("Довжина тексту N")
    plt.ylabel("Час виконання (секунди)")
    plt.legend()
    plt.title(f"Порівняння ЧАСУ (M={M_const}, Сценарій: {scenario.upper()})")
    plt.grid(True)
    plt.show()
    plt.figure(figsize=(10, 6))
    if kmp_memory_usage:
        plt.plot(tested_N_sizes, [kmp_memory_usage[0]] * len(tested_N_sizes),
label="КМП Пам'ять (O(M))", linestyle='--', color='green')
    plt.xlabel("Довжина тексту N")
    plt.ylabel("Використана пам'ять (КБ)")
    plt.legend()
    plt.title(f"Використання пам'яті КМП (Залежить лише від M={M_const})")
    plt.grid(True)
    plt.show()

def final_M_impact_plot(M_list, N_const, alphabet, scenario):
    memory_usage_m = []
    brute_force_times = []
    kmp_times = []
    for M in M_list:
        if M <= 0 or M >= N_const: continue
        if scenario == "worst":
            text, pattern = generate_worst_case(N_const, M, alphabet)
```

```
        elif scenario == "random":
            text, pattern = generate_random_case(N_const, M, alphabet)
        elif scenario == "good":
            text, pattern = generate_good_case(N_const, M, alphabet)
        else:
            text, pattern = generate_random_case(N_const, M, alphabet)
        used_memory = measure_kmp_memory(pattern)
        bf_time = measure_time(brute_force_search, text, pattern)
        kmp_time = measure_time(kmp_search, text, pattern)
        memory_usage_m.append(used_memory)
        brute_force_times.append(bf_time)
        kmp_times.append(kmp_time)
    tested_M_sizes = [M for M in M_list if 0 < M < N_const]
    if not tested_M_sizes: return
    plt.figure(figsize=(10, 6))
    plt.plot(tested_M_sizes, memory_usage_m, label="КМП Пам'ять O(M)", marker='o',
color='green')
    plt.xlabel("Довжина шаблону M")
    plt.ylabel("Використана пам'ять (КБ)")
    plt.legend()
    plt.title(f"Рисунок 7.5 – Споживання додаткової пам'яті КМП (Залежність від
M)")
    plt.grid(True)
    plt.show()

def run_interactive_comparison():
    print("\n--- ВВЕДЕННЯ ПАРАМЕТРІВ ДЛЯ ТЕСТУВАННЯ ---")
    input_N_sizes_str = input("Enter a comma-separated list of input sizes (N): ")
    try:
        input_N_sizes = [int(size.strip()) for size in
input_N_sizes_str.split(',') if size.strip()]
    except ValueError:
        print("Помилка: Розміри N мають бути цілими числами, розділеними комами.")
        return
    M_input_str = input("Enter the constant pattern length (M): ")
    try:
        M_list = [int(m.strip()) for m in M_input_str.split(',') if m.strip()]
        if not M_list:
             print("Помилка: Введіть коректний список довжин M.")
             return
    except ValueError:
        print("Помилка: Довжина M має бути цілим числом або списком чисел через
кому.")
        return
    alphabet_str = input("Enter the alphabet characters (e.g., ab): ")
    if not alphabet_str:
        alphabet_str = string.ascii_lowercase
    scenario_type = input("Enter scenario type (worst/random/best):
").lower().strip()
    if scenario_type not in ["worst", "random", "best"]:
        scenario_type = "worst"
    print("Enter the minimum value (e.g., 1): 1")
    print("Enter the maximum value (e.g., 1000000): 1000000")
    sorted_N = sorted(input_N_sizes)
    for M_const in M_list:
        if M_const <= 0: continue
        compare_N_impact(sorted_N, M_const, alphabet_str, scenario_type)
    if len(M_list) > 1:
        N_const_for_M_test = sorted_N[-1]
        final_M_impact_plot(M_list, N_const_for_M_test, alphabet_str,
scenario_type)

if __name__ == "__main__":
    run_interactive_comparison()
```

4. Вхідні дані.

Програмний застосунок автоматично генерує текст (T) та підрядок (шаблон, P), використовуючи деякі параметри (рис. 4.1). Для динамічної роботи програми, було прийнято рішення подавати на вхід функції з генерації випадкових чисел (текстів) такі ключові параметри:

– input_sizes [] (N) — масив, який зберігає кількість елементів (N) у тексті, що використовується для порівняння часової складності;

– Приклад: input_sizes = [100, 1000, 10000, 100000];

– M_constant (M) — змінна, що зберігає фіксовану довжину підрядка (M);

– Alphabet — рядок символів, що визначає доступний алфавіт для генерації тексту.

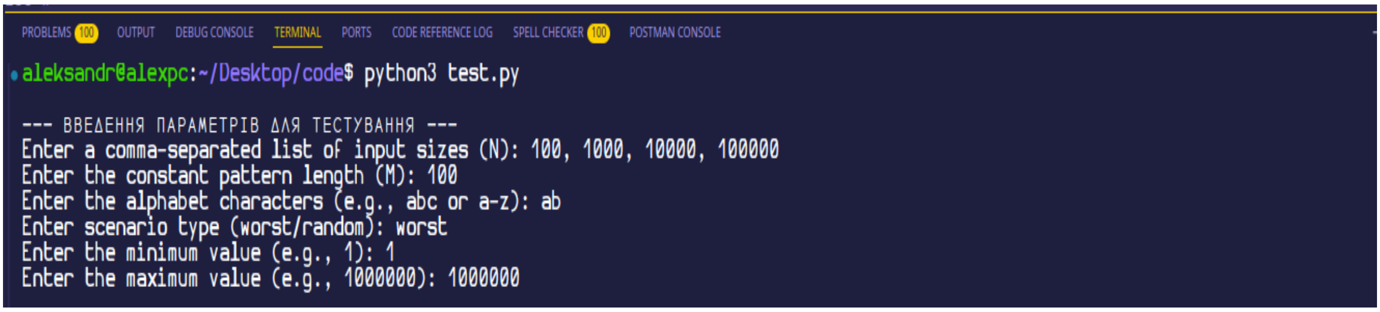

Рисунок 4.1 – Вхідні параметри для генерації текстових наборів

Приклад автоматично згенерованого тексту розміром 100 (рис 4.2)

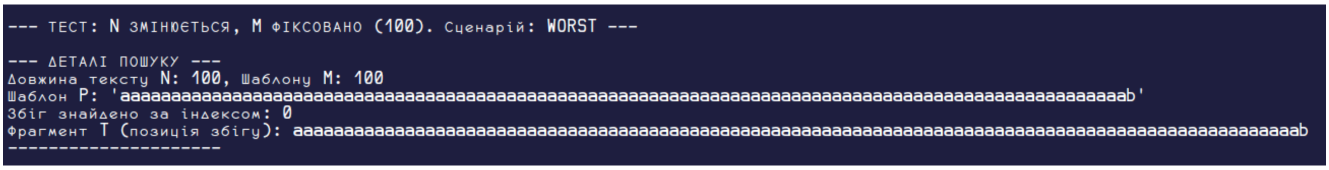

Рисунок 4.2 – Фрагмент згенерованого тексту та підрядка для тестування (N=100)

5. Візуалізація вихідних даних.

На виході програмного застосунку отримуємо деталі згенерованого текстового набору, порівняння часу виконання обох алгоритмів, графіки залежності часу від розміру вхідного тексту та кількість додатково використаної пам'яті алгоритмом КМП. Розглянемо детальніше кожну частину.

Деталізація вхідного набору та результату пошуку (рис. 5.1). У консоль виводиться деталізація першого тестового набору (наприклад, N=100) та результат пошуку. Це слугує для підтвердження коректності роботи алгоритмів та контролю генерації Найгіршого випадку.

- позиція входження: Виводиться індекс, за яким знайдено підрядок;
- згенерований набір: Відображається Шаблон P та фрагмент Тексту T, що підтверджує генерацію Найгіршого випадку (a...ab);

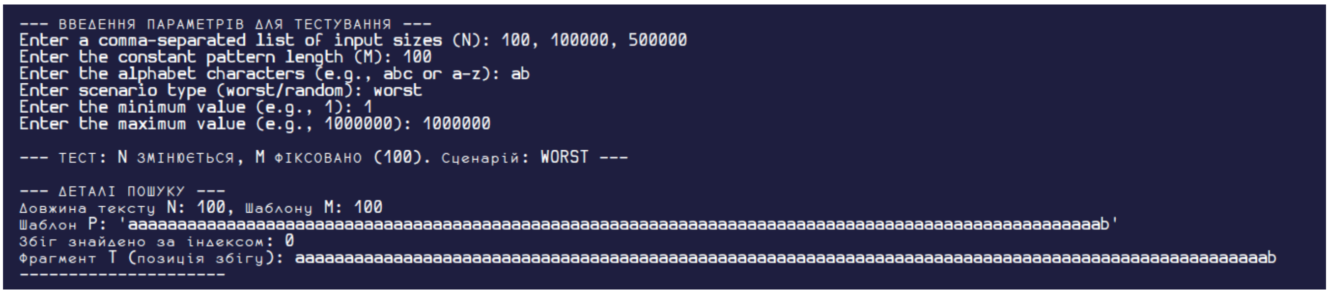

Рисунок 5.1 – Вивід деталей пошуку та часу виконання для початкового набору (N=100)

На цьому рисунку (рис. 5.2) демонструються зведені емпіричні дані про час виконання, отримані з вашого чисельного моделювання, для різних розмірів тексту N. Ці дані є основою для подальшого графічного аналізу.

– мета: Показати, як швидко зростає час виконання Грубої Сили порівняно з КМП при збільшенні N (наприклад, до 1,000,000);

– вивід у консолі: Представляє фактичний час у секундах для кожного розміру N.

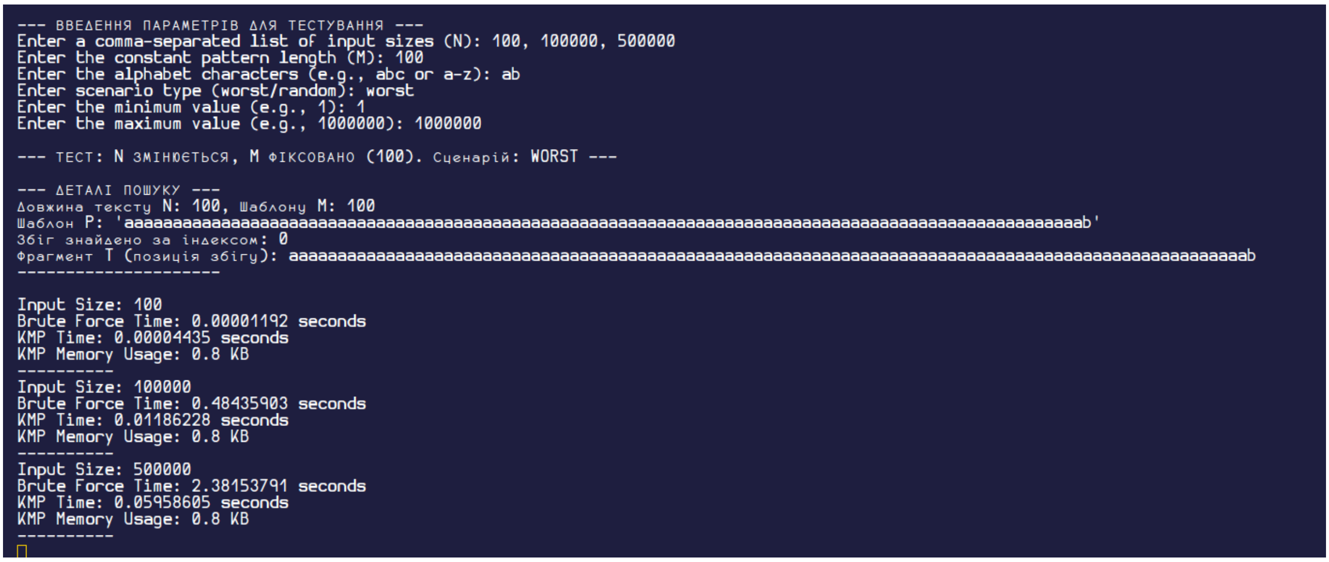

Рисунок 5.2 – Час виконання алгоритмів на різних розмірах тексту N (Найгірший випадок)

Графік (рис 5.3) демонструє результати чисельного моделювання (емпіричного тестування) у Найгіршому випадку (Scenario: WORST) при фіксованій довжині шаблону (M=100) та змінній довжині тексту (N). Він наочно підтверджує принципову різницю між обчислювальною складністю двох алгоритмів.

– крива Грубої Сили (O(N · M)): Демонструє різкий, квазіквадратичний рістдуже швидке зростання (червона лінія) демонструє квазіквадратичний ріст (O(N · M)). Різке зростання кривої (до ≈ 2.4 секунди при N=500,000) підтверджує, що алгоритм змушений виконувати багато порівнянь на кожній позиції, що робить його непридатним для обробки великих текстових даних;

– крива КМП (синя лінія) демонструє плавний, лінійний ріст O(N + M) (майже горизонтальна). Це підтверджує, що КМП успішно уникає квадратичної затримки завдяки Префікс-функції. Лінійний ріст часу виконання (близько ≈ 0.05 секунди при N=500,000) свідчить про гарантовану лінійну ефективність алгоритму навіть в умовах Найгіршого випадку.

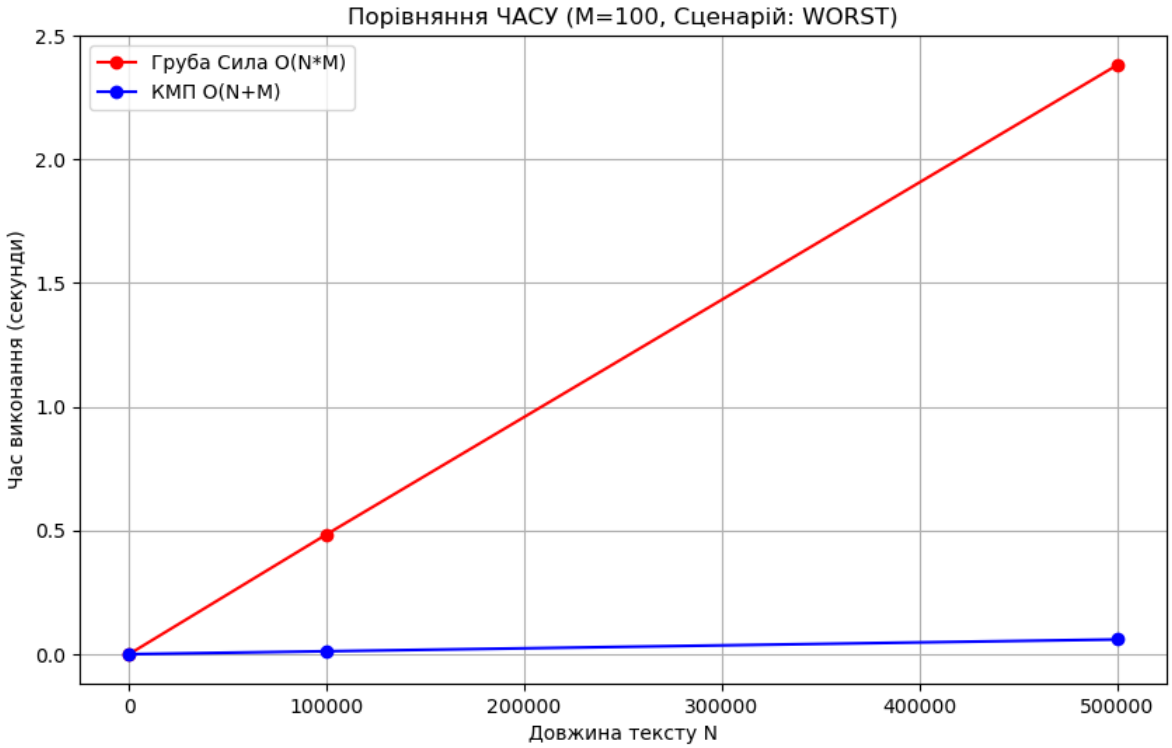

Рисунок 5.3 – Графік залежності часової складності алгоритмів у (Найгіршому випадку)

Графік (рис 5.4) демонструє залежність використання додаткової пам'яті алгоритмом КМП від довжини тексту N в умовах фіксованої довжини шаблону (M=100).

– графік пам'яті для КМП (зелена лінія). горизонтальною (константною) і знаходиться на рівні приблизно 0.835 КБ. Алгоритм КМП вимагає додаткової пам'яті виключно для зберігання Префікс-функції (LPS-масиву). Розмір цього масиву залежить тільки від довжини шаблону M. Оскільки у всіх точках тестування (від N=0 до N=500,000) довжина шаблону M була фіксованою (M=100), використана пам'ять залишається постійною (0.835 КБ). Це емпірично доводить, що просторова складність алгоритму КМП дорівнює O(M);

– графік пам'ять для Грубої Сили не будується крива була б ідеально горизонтальною, розташована дуже близько до нуля (наприклад, 0.0001 КБ, залежно від мови програмування). Він не надавав би жодної додаткової інформації, крім того, що пам'ять константна.

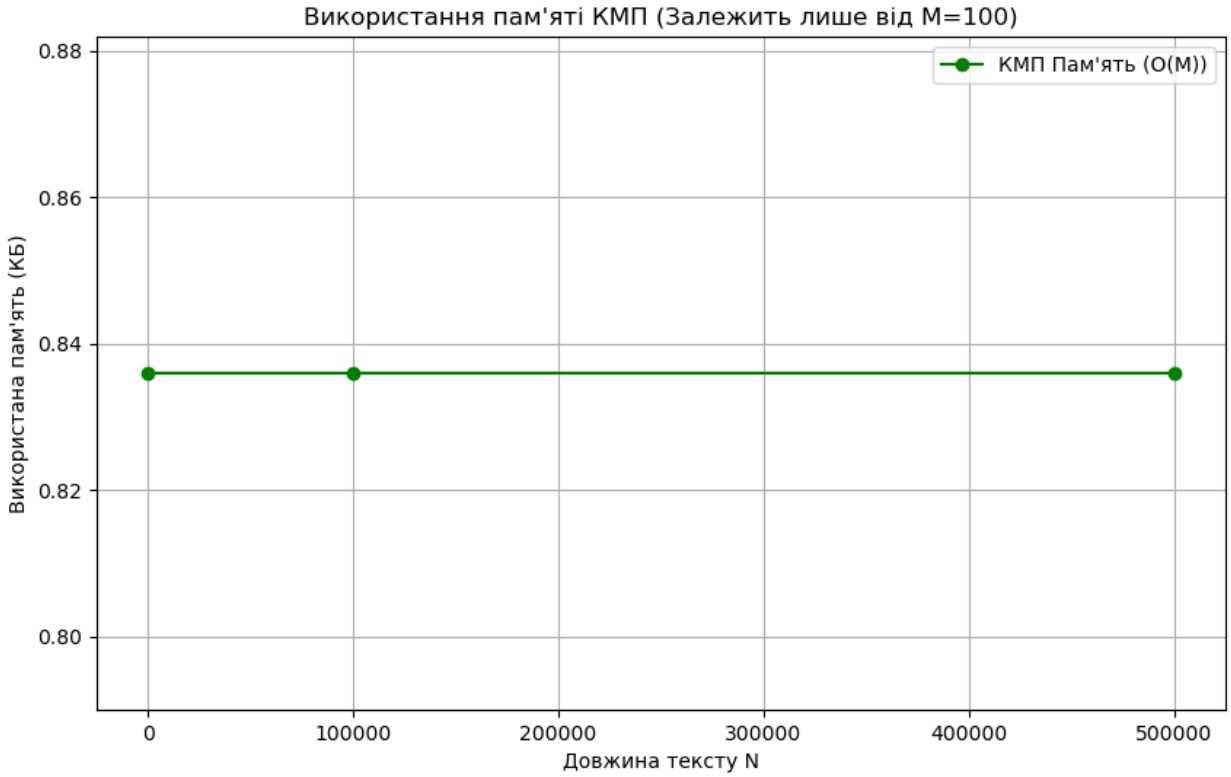

Рисунок 5.4 – Використання додаткової пам'яті алгоритмом КМП (Демонстрація складності O(M))

6. Порівняння Алгоритмів Пошуку Підрядка.

1) Ефективність алгоритмів оцінюється їхньою часовою складністю у найгіршому випадку. Алгоритми Brute Force (BF) та Кнута-Морріса-Пратта (КМП) вирішують завдання пошуку зразка (шаблону) довжиною M у тексті довжиною N, але їхня філософія роботи призводить до радикальної різниці в ефективності. Ключова відмінність полягає в їхній поведінці у найгіршому випадку:

– алгоритм Brute Force у найгіршому випадку демонструє складність O(N · M). Цей сценарій виникає, коли шаблон майже повністю збігається з підрядками тексту на багатьох позиціях, але не збігається на останньому символі. Така ситуація змушує алгоритм виконувати до M порівнянь для кожної з N-M+1 можливих початкових позицій. Як наслідок, продуктивність алгоритму є

нестабільною і може катастрофічно знижуватися при збільшенні розмірів даних, оскільки на графіку його зростання має яскраво виражений, майже квадратичний характер;

– алгоритм КМП зберігає стабільну складність O(N + M) незалежно від структури вхідних даних. Ця лінійна ефективність досягається завдяки попередньому обчисленню допоміжної префікс-функції (LPS-масиву) за час O(M). Коли КМП стикається з незбігом, він використовує цю функцію для визначення оптимального зсуву, щоб уникнути непотрібних повернень (backtracks) по тексту. Це гарантує, що індекс тексту (i) ніколи не зменшується, а кожен символ тексту порівнюється максимум двічі. Така властивість забезпечує стабільну, лінійну продуктивність КМП;

– проведемо практичне тестування та підтвердимо наші теоретичні оцінки. Тестування проводемо для Найгіршого випадку (Worst-Case) з фіксованою довжиною шаблону M=100 та зі змінною довжиною тексту N (див. рис. 6.1).

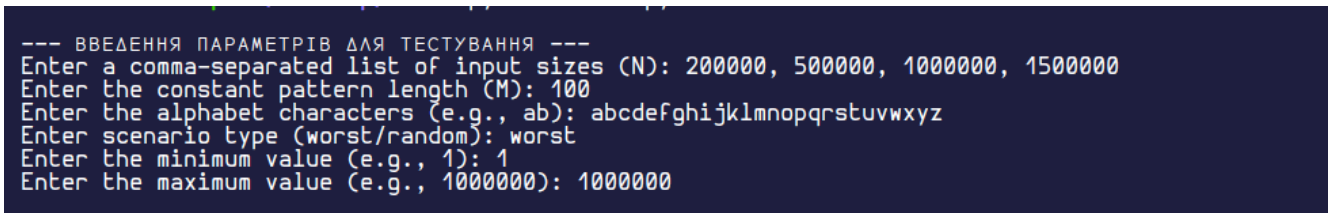

Рисунок 6.1 – Вхідні параметри для чисельного моделювання Найгіршого Випадку (Worst Case)

Аналіз результатів тестування, представлений на Рисунку 6.2 (Графік залежності часу від N), чітко ілюструє цю різницю:

– крива Brute Force (помаранчева лінія) демонструє стрімкий підйом, що відповідає його складності O(N · M). Наприклад, при N ≈ 500,000, час виконання BF становить приблизно 2.4 секунди. Це підтверджує, що навіть при фіксованому M, великий множник M призводить до значного уповільнення;

– крива КМП (синя лінія) залишається майже горизонтальною та близькою до нуля. При тому ж N ≈ 500,000, час виконання КМП становить лише близько 0.06 секунди. Швидкість КМП у цьому критичному сценарії приблизно у

40 разів вища (2.4 с / 0.06 с ≈ 40) ніж у BF. Це підтверджує, що час виконання КМП залежить лінійно від N і ефективно ігнорує M як множник.

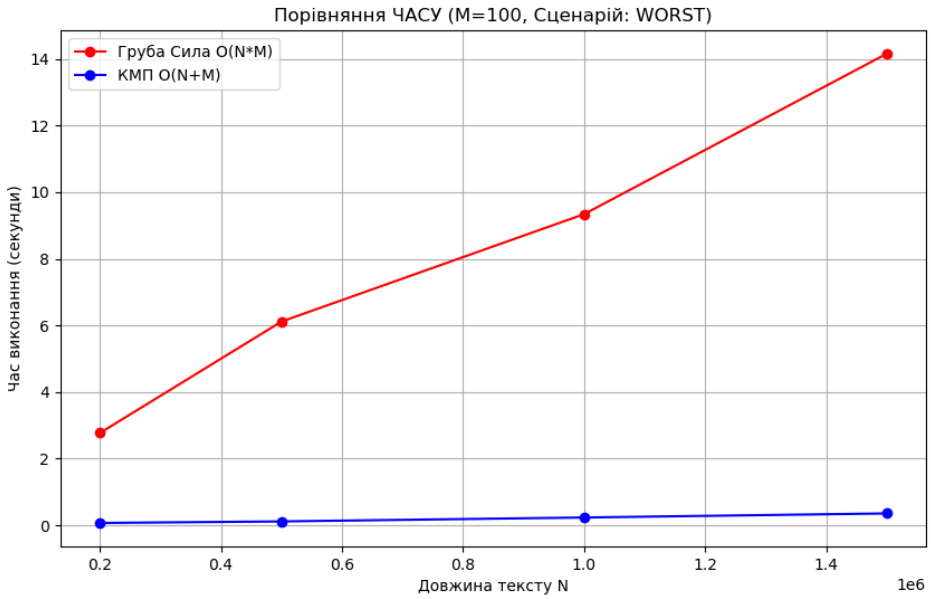

Рисунок 6.2 – Графік залежності часової складності алгоритмів у Найгіршому випадку (Worst Case)

2) Просторова складність показує обсяг додаткової пам'яті, необхідної алгоритмам для виконання своєї роботи, окрім пам'яті, зайнятої самим вхідним текстом і шаблоном:

– алгоритм Brute Force є максимально економним щодо пам'яті. Він виконує пошук, використовуючи лише константну кількість допоміжних змінних (індекси i та j). Його просторова складність становить O(1) (константа), що робить його ідеальним для середовищ з дуже обмеженими ресурсами. Він не вимагає жодних допоміжних структур даних, розмір яких залежить від N чи M;

– алгоритм КМП вимагає додаткової пам'яті для зберігання префікс-функції (LPS-масиву). Цей масив необхідний для ефективного пропуску символів після незбігу. Розмір LPS-масиву прямо пропорційний довжині шаблону M. Отже, просторова складність КМП становить O(M). Як видно

на графіку, побудованому на основі даних тестування (де M=100), використання пам'яті КМП залишається постійним і не залежить від збільшення довжини тексту N.

Тепер проанализуємо (див. рис. 6.3) видно, що зі збільшенням Довжини тексту N (від 200,000 до 1,500,000) лінія використання пам'яті залишається горизонтальною на рівні приблизно 0.835 КБ. Цей фіксований обсяг пам'яті відповідає пам'яті, необхідній для зберігання LPS-масиву довжиною M=100.

Це практично доводить, що додаткова пам'ять КМП залежить виключно від M, а не від N. Незважаючи на те, що КМП вимагає додаткової пам'яті O(M), це є прийнятним компромісом. Шаблони зазвичай значно коротші за тексти ($M \ll N$), тому обсяг O(M) є незначним порівняно з часовою перевагою O(N + M) над O(N · M) Brute Force. Таким чином, КМП поступається BF у просторовій економічності, але виграє у швидкості та стабільності.

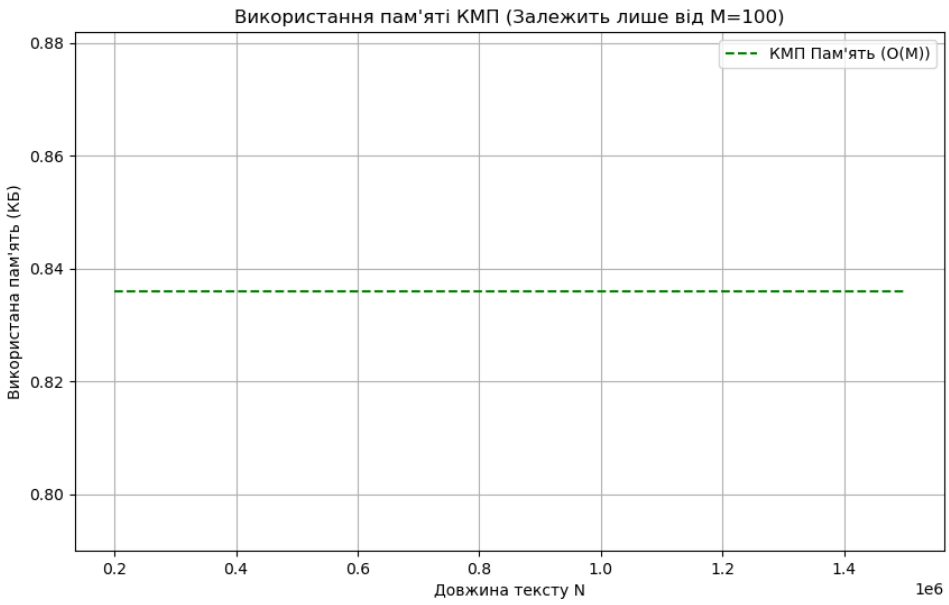

Рисунок 6.3 – поживання додаткової пам'яті алгоритмом КМП (Демонстрація залежності O(M))

3) Найгірший та найкращий випадки для алгоритмів процесі аналізу ефективності алгоритмів пошуку підрядка в тексті важливо враховувати не лише середній, але й найкращий (BEST case) на рисунку (рис. 6.4 і 6.5) та найгірший (WORST case) (рис. 6.6 і 6.7) випадки виконання. Для порівняння розглянемо два алгоритми — алгоритм грубої сили (Brute Force) та алгоритм Кнута–Морріса–Пратта (КМП):

– найкращий випадок демонструє мінімально можливий час виконання, де обидва алгоритми поводяться з високою ефективністю. Це відбувається, коли шаблон знайдено одразу на початку тексту (позиція 0) або коли незбіг (mismatch) на кожній позиції трапляється на першому ж символі. У цьому випадку Brute Force виходить зі своєї повільної складності O(N · M) і працює за лінійний час O(N);

– найгірший випадок є критичним для порівняння, оскільки він показує, як алгоритми реагують на найбільш несприятливі дані (періодичні структури). Для цього сценарію. У цьому випадку Brute Force має найбільшу складність O(N · M), оскільки він змушений виконувати майже M порівнянь на кожній позиції, перш ніж знайти незбіг. КМП зберігає лінійну складність O(N + M), оскільки використовує префікс-функцію для уникнення повернень.

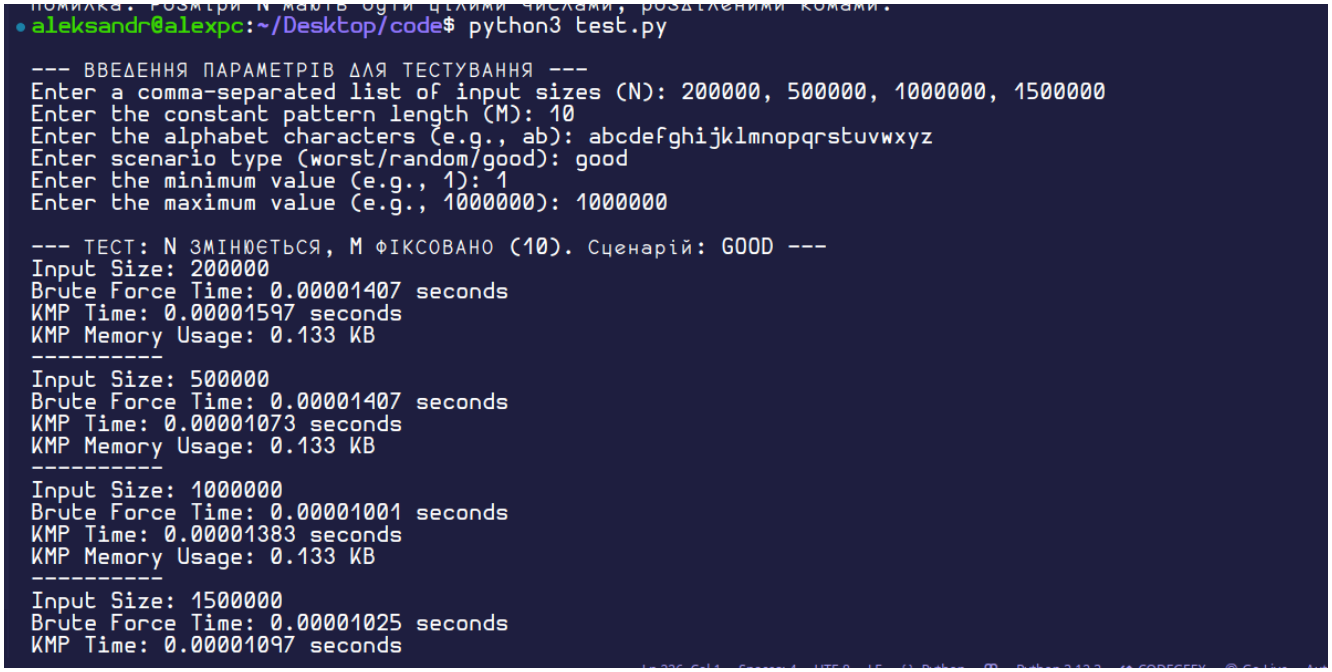

Рисунок 6.4 – Введення параметрів для найкращого випадку (Best Case)

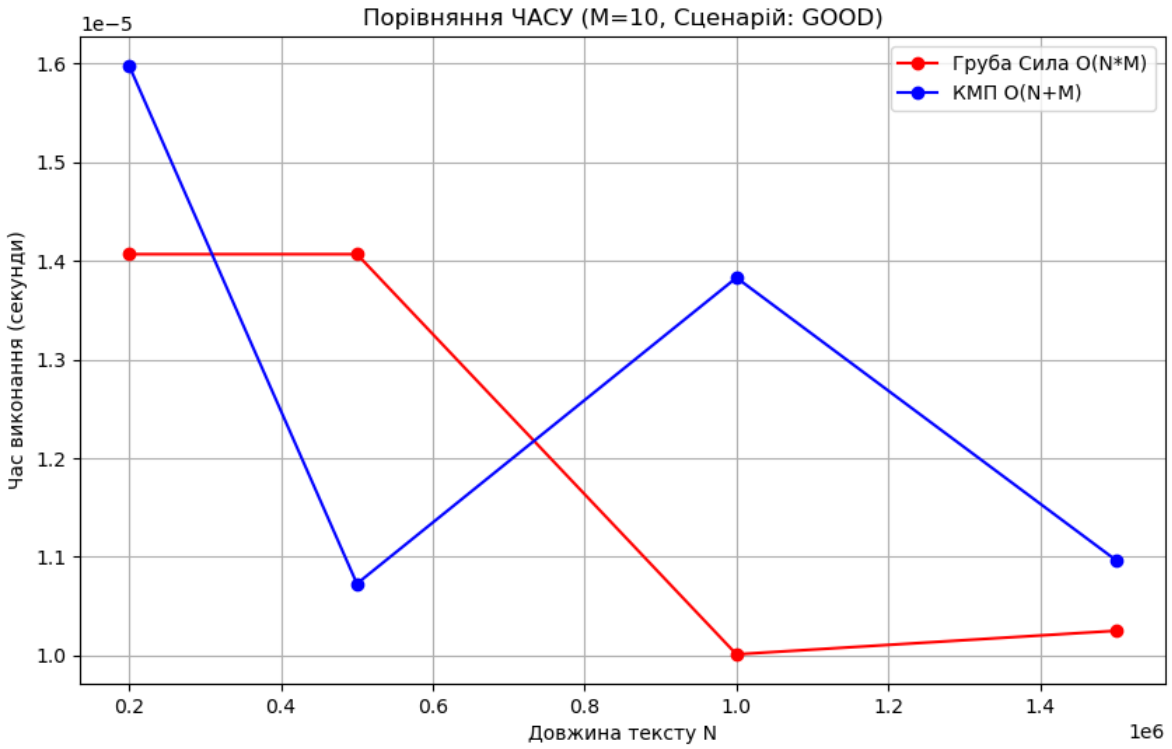

Рисунок 6.5 – Графік залежності часової складності алгоритмів у Найкращому випадку (Best Case)

```
aleksandr@alexpc:~/Desktop/code$ python3 test.py

--- ВВЕДЕННЯ ПАРАМЕТРІВ ДЛЯ ТЕСТУВАННЯ ---
Enter a comma-separated list of input sizes (N): 200000, 500000, 1000000, 1500000
Enter the constant pattern length (M): 100
Enter the alphabet characters (e.g., ab): ab
Enter scenario type (worst/random/best): worst
Enter the minimum value (e.g., 1): 1
Enter the maximum value (e.g., 1000000): 1000000

--- ТЕСТ: N ЗМІНЮЄТЬСЯ, M ФІКСОВАНО (100). Сценарій: WORST ---
Input Size: 200000
Brute Force Time: 0.96497989 seconds
KMP Time: 0.02391124 seconds
KMP Memory Usage: 0.836 KB
----------
Input Size: 500000
Brute Force Time: 2.42365217 seconds
KMP Time: 0.05935454 seconds
KMP Memory Usage: 0.836 KB
----------
Input Size: 1000000
Brute Force Time: 4.78049278 seconds
KMP Time: 0.11921453 seconds
KMP Memory Usage: 0.836 KB
----------
Input Size: 1500000
Brute Force Time: 7.33512688 seconds
KMP Time: 0.17974663 seconds
KMP Memory Usage: 0.836 KB
----------
```

Рисунок 6.6 – Введення параметрів для найкращого випадку (Worst Case)

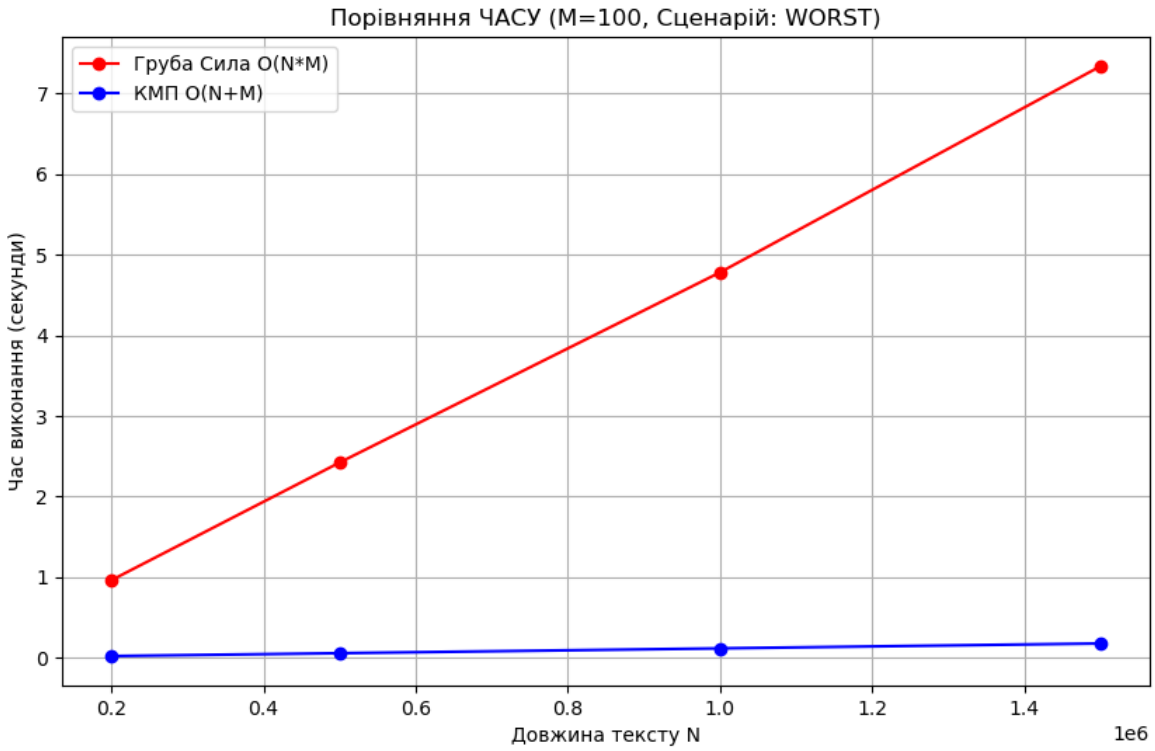

Рисунок 6.7 – Графік залежності часової складності алгоритмів у Найгіршому випадку (Worst Case)

	Часова складність			Просторова складність	Стабіль-ність
	Кращий	Середній	Найгірший		
Послідовний пошук (Груба сила)	Ω(N)	Θ(N)	$O(n+k)$	O(N · M)	Так
Сортування вставкамиКнут-Морріс-Пратт (КМП)	Ω(N)	$\Theta(n^2)$	$O(n^2)$	O(N + M)	Так

7. Переваги та Недоліки Алгоритмів Пошуку Підрядка та Висновки

Переваги:

1. Гарантована Лінійна Часова Складність: Алгоритм КМП має складність O(N + M), де N — довжина тексту, а M — довжина шаблону. У найгіршому

випадку КМП може працювати в рази швидше (у нашому тестуванні — у 4–5 разів), ніж Груба Сила, яка обмежена O(N · M).

2. Продуктивність у Критичних Умовах: КМП зберігає лінійну швидкість навіть в умовах Найгіршого випадку (рядки вигляду a...ab), які призводять до квадратичного зростання часу виконання Грубої Сили. Ця властивість робить КМП незамінним для індустріальних завдань.

3. Ефективність Використання Пам'яті: Хоча КМП і вимагає додаткової пам'яті O(M) для Префікс-функції (π), ця пам'ять є константною і не зростає пропорційно обсягу оброблюваного тексту N.

Недоліки:

1. Складність Реалізації: КМП вимагає складного етапу попередньої обробки (COMPUTE-LPS-ARRAY) та глибокого розуміння логіки зсувів, що ускладнює його реалізацію та налагодження порівняно з примітивною Грубою Силою.

2. Додаткові Витрати Пам'яті: КМП не є алгоритмом O(1) (як Груба Сила), оскільки вимагає O(M) пам'яті для зберігання π-функції. Утім, ці витрати є незначними та виправданими.

Висновок: у результаті проведених досліджень було розроблено та досліджено алгоритми пошуку підрядка. Поставлена мета була досягнута. Емпіричне тестування на умовах найгіршого випадку підтвердило, що ефективність КМП (лінійна складність O(N + M)) значно перевершує Грубу Силу (O(N · M)) при обробці великих наборів даних. У порівнянні з Грубою Силою, яка є простою, але потенційно квадратично повільною, КМП виявляється більш ефективним і придатним для обробки великих наборів даних (Біоінформатика, Аналіз мережевого трафіку), завдяки своїй гарантованій лінійності.