Міністерство освіти і науки України
Харківський національний університет імені В. Н. Каразіна
Факультет математики і інформатики
Кафедра прикладної математики

До захисту допущено
кафедрою прикладної математики, протокол №5 від 12 червня 2026 р.

завідувач кафедри
прикладної математики
доктор фіз.-мат. наук, професор *Валерій КОРОБОВ*

КВАЛІФІКАЦІЙНА РОБОТА

здобувача першого (бакалаврського) рівня вищої освіти

«Порівняльний аналіз алгоритмів пошуку підрядків у великих текстових даних»

Спеціальність 113 Прикладна математика
Освітня програма Прикладна математика

Здобувач Ілля БЄГУНОВ

Науковий керівник кандидат фіз.-мат. наук, доцент
кафедри прикладної математики
Сергій ПОСЛАВСЬКИЙ

Харків – 2026

АНОТАЦІЯ

Об'єкт дослідження – алгоритми пошуку підрядків у текстових даних великого обсягу. Предмет дослідження – часова та просторова складність алгоритмів пошуку підрядків, їх практична ефективність при обробці текстів різного характеру, зокрема багатобайтових Unicode-текстів у кодуванні UTF-8.

Мета роботи – провести порівняльний аналіз класичних алгоритмів пошуку одиночного підрядка (наївний метод, Кнута–Морріса–Пратта, Боєра–Мура, Рабіна–Карпа) та алгоритму Ахо–Корасік для множинного пошуку; дослідити вплив багатобайтового кодування UTF-8 на продуктивність алгоритмів.

Методи дослідження: теоретичний аналіз обчислювальної складності, експериментальне порівняння на основі бенчмаркінг-тестів мовою Python.

У роботі здійснено систематичний огляд алгоритмів пошуку підрядків, надано математичне обґрунтування їх часової та просторової складності. Реалізовано програмний стенд для бенчмаркінгу, проведено шість серій експериментів із текстами різного обсягу, алфавіту та кодування. Досліджено вплив кодування UTF-8 на продуктивність алгоритмів на прикладі кириличного тексту: встановлено, що побайтовий пошук повільніший за посимвольний на 21–61% через збільшення обсягу даних удвічі, а нерівномірний розподіл байтів кирилиці знижує ефективність евристики поганого символу алгоритму Боєра–Мура. Показано, що алгоритм Ахо–Корасік забезпечує прискорення у 295.7 рази при 1000 шаблонах порівняно з послідовним пошуком. Сформульовано практичні рекомендації щодо вибору алгоритму залежно від характеристик вхідних даних.

Ключові слова: *пошук підрядків, префіксне дерево, складність алгоритмів, бенчмаркінг, Unicode, UTF-8.*

ABSTRACT

The research object is substring search algorithms applied to large-scale text data. The subject of the study is the time and space complexity of these algorithms and their practical performance on texts of varying structure and size, including multibyte Unicode texts encoded in UTF-8.

The purpose of the thesis is to conduct a comparative analysis of classical single-pattern search algorithms (naive, Knuth–Morris–Pratt, Boyer–Moore, Rabin–Karp) and the Aho–Corasick multi-pattern algorithm, and to investigate the impact of UTF-8 multibyte encoding on algorithm performance.

Research methods: theoretical analysis of computational complexity, experimental comparison using benchmarking tests implemented in Python.

The thesis presents a systematic review of substring search algorithms with mathematical justification of their complexity. A benchmarking framework was implemented, and six series of experiments were conducted with texts of different volumes, alphabets, and encodings. The impact of UTF-8 encoding on algorithm performance was investigated using Cyrillic text: byte-level search was found to be 21–61% slower than character-level search due to doubled data volume, and the uneven distribution of Cyrillic bytes reduces the effectiveness of the Boyer–Moore bad character heuristic. The Aho–Corasick algorithm was shown to provide a 295.7-fold speedup with 1000 patterns compared to sequential search. Practical recommendations for algorithm selection based on input data characteristics are formulated.

Keywords: *substring search, prefix tree, algorithm complexity, benchmarking, Unicode, UTF-8.*

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ 6
ВСТУП 7
РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ АЛГОРИТМІВ ПОШУКУ ПІДРЯДКІВ 10
ТА ОСОБЛИВОСТІ ЇХ ЗАСТОСУВАННЯ 10
1.1. Еволюція методів пошуку підрядків: від наївного підходу до спеціалізованих алгоритмів 10
1.2. Алгоритм Кнута–Морріса–Пратта та його теоретичне обґрунтування 11
1.3. Алгоритм Боєра–Мура: евристики та умови ефективності 12
1.4. Алгоритм Рабіна–Карпа: ймовірнісний підхід на основі хешування 13
1.5. Алгоритм Ахо–Корасік: множинний пошук на основі префіксного дерева 14
1.5.1. Побудова префіксного дерева (Trie) 14
1.5.2. Суфіксні посилання та функція переходів 15
1.6. Проблема Unicode та багатобайтових кодувань 15
1.7. Порівняльний аналіз теоретичної складності алгоритмів 17
РОЗДІЛ 2. РЕАЛІЗАЦІЯ АЛГОРИТМІВ ТА МЕТОДОЛОГІЯ 20
ЕКСПЕРИМЕНТУ 20
2.1. Вибір інструментів та середовища розробки 20
2.2. Набори тестових даних 20
2.3. Реалізація наївного алгоритму 22
2.4. Реалізація алгоритму Кнута–Морріса–Пратта 22
2.5. Реалізація алгоритму Боєра–Мура 23
2.6. Реалізація алгоритму Рабіна–Карпа 24
2.7. Реалізація алгоритму Ахо–Корасік 25
2.8. Методологія вимірювань 26
2.8.1. Вимірювання часу виконання 26
2.8.2. Вимірювання споживання пам'яті 27
2.8.3. Організація експериментів 27
РОЗДІЛ 3. АНАЛІЗ РЕЗУЛЬТАТІВ ЕКСПЕРИМЕНТІВ 29
3.1. Залежність часу виконання від обсягу тексту 29
3.2. Вплив довжини шаблону на продуктивність 30
3.3. Вплив розміру алфавіту 32
3.4. Дослідження впливу кодування UTF-8 33
3.4.1. Порівняння посимвольного та побайтового пошуку 34
3.4.2. Порівняння ASCII та UTF-8 при однаковій кількості символів 35
3.4.3. Аналіз розподілу байтів у UTF-8 36

3.5. Ефективність множинного пошуку 37
3.6. Порівняння споживання пам'яті 38
3.7. Рекомендації щодо вибору алгоритму 39
ВИСНОВКИ 42
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 45
ДОДАТОК А 45

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

KMP – алгоритм Кнута–Морріса–Пратта (Knuth–Morris–Pratt)

BM – алгоритм Боєра–Мура (Boyer–Moore)

RK – алгоритм Рабіна–Карпа (Rabin–Karp)

AC – алгоритм Ахо–Корасік (Aho–Corasick)

BFS – обхід у ширину (Breadth-First Search)

Trie – префіксне дерево (бор)

UTF-8 – формат кодування символів Unicode змінної довжини

ASCII – American Standard Code for Information Interchange

n – довжина тексту T

m – довжина шаблону P

Σ – алфавіт, |Σ| – розмір алфавіту

L – сумарна довжина множини шаблонів

z – кількість знайдених входжень

π – функція невдач (prefix function) алгоритму KMP

ВСТУП

Задача пошуку підрядка (pattern matching) є однією з фундаментальних проблем інформатики, що знаходить застосування у широкому спектрі прикладних областей: від текстових редакторів та пошукових систем до біоінформатики, систем виявлення вторгнень та аналізу великих масивів даних [1]. Формально задачу можна сформулювати наступним чином: дано текст T довжини n та шаблон P довжини m, необхідно знайти всі позиції i такі, що $T[i..i+m-1] = P$. Незважаючи на простоту формулювання, ця задача має багату історію досліджень, починаючи з 1970-х років, коли було запропоновано перші ефективні алгоритми [4, 5].

Актуальність теми зумовлена кількома факторами. По-перше, сучасні прикладні задачі оперують текстами обсягом від мегабайтів до гігабайтів, а наївний підхід з квадратичною складністю $O(n \cdot m)$ стає неприйнятним для обробки даних у реальному часі. Пошукові системи, антивірусні сканери, системи фільтрації контенту та інструменти біоінформатики потребують алгоритмів, здатних обробляти великі обсяги тексту за лінійний або сублінійний час [1]. По-друге, глобалізація інформаційного простору призвела до поширення багатобайтових кодувань, насамперед Unicode та його реалізації UTF-8. Станом на сьогодні переважна більшість веб-сторінок використовують кодування UTF-8. Однак більшість існуючих порівняльних досліджень алгоритмів пошуку підрядків обмежуються однобайтовим ASCII-текстом [3, 8], що не відображає реальних умов роботи з текстами українською, китайською, арабською та іншими мовами, символи яких кодуються двома і більше байтами. По-третє, реальні системи часто потребують одночасного пошуку багатьох шаблонів (наприклад, фільтрація стоп-слів, пошук сигнатур у антивірусних системах), що вимагає застосування спеціалізованих алгоритмів, таких як Ахо–Корасік [7].

Мета роботи – здійснити порівняльний аналіз алгоритмів пошуку підрядків за критеріями часової та просторової складності, а також дослідити

їхню практичну ефективність шляхом бенчмаркінг-тестів на текстах різного характеру та кодування.

Для досягнення поставленої мети визначено такі завдання:

1. Провести огляд існуючих алгоритмів пошуку підрядків (наївний, KMP, Боєра–Мура, Рабіна–Карпа, Ахо–Корасік) та здійснити теоретичний аналіз їх обчислювальної складності.

2. Реалізувати досліджувані алгоритми мовою Python та розробити програмний стенд для бенчмаркінгу з вимірюванням часу виконання та споживання пам'яті.

3. Провести серію експериментів з різними типами вхідних даних: текстами різного обсягу (від 100 КБ до 10 МБ), з різними алфавітами (від бінарного до кирилиці), з шаблонами різної довжини (від 5 до 100 символів).

4. Дослідити вплив багатобайтового кодування UTF-8 на продуктивність алгоритмів, порівнявши посимвольний та побайтовий режими пошуку на кириличних текстах.

5. Оцінити ефективність алгоритму Ахо–Корасік для задачі множинного пошуку порівняно з послідовним застосуванням алгоритмів одиночного пошуку.

6. Сформулювати практичні рекомендації щодо вибору алгоритму залежно від характеристик задачі.

Об'єкт дослідження – алгоритми пошуку підрядків у текстових даних. Предмет дослідження – часова та просторова складність цих алгоритмів, їх практична продуктивність на текстах різної природи та кодування.

Методи дослідження: теоретичний аналіз обчислювальної складності (оцінки у нотації «великого O» для найкращого, середнього та найгіршого випадків), експериментальне дослідження з використанням бенчмаркінг-тестів (вимірювання часу через time.perf_counter, вимірювання пам'яті через tracemalloc, статистична обробка результатів із використанням медіани та стандартного відхилення).

Наукова новизна полягає у дослідженні впливу багатобайтового кодування UTF-8 на продуктивність алгоритмів пошуку підрядків, що раніше не було предметом систематичного аналізу. Зокрема, встановлено, що побайтовий пошук у кириличному тексті є повільнішим за посимвольний на 21–61%, а нерівномірний розподіл байтів кирилиці у UTF-8 (концентрація у діапазонах 0xD0–0xD1 та 0x80–0xBF) знижує ефективність евристики поганого символу алгоритму Боєра–Мура.

Практичне значення: результати роботи можуть бути використані для обґрунтованого вибору алгоритму пошуку підрядків у прикладних програмних системах, що працюють із текстами різних мов та кодувань.

Структура та обсяг роботи. Дипломна робота складається зі вступу, трьох розділів, висновків, списку використаних джерел та додатку. У першому розділі здійснено теоретичний огляд алгоритмів пошуку підрядків із аналізом їхньої обчислювальної складності та систематизовано проблеми застосування класичних алгоритмів до багатобайтових Unicode-текстів. У другому розділі описано реалізацію алгоритмів, набори тестових даних та методологію вимірювань. У третьому розділі проведено аналіз результатів шести серій експериментів та сформульовано практичні рекомендації.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ АЛГОРИТМІВ ПОШУКУ ПІДРЯДКІВ
ТА ОСОБЛИВОСТІ ЇХ ЗАСТОСУВАННЯ

1.1. Еволюція методів пошуку підрядків: від наївного підходу до спеціалізованих алгоритмів

Задача пошуку підрядка (pattern matching) є однією з фундаментальних проблем теоретичної інформатики, яка має безпосереднє прикладне значення у широкому спектрі областей: текстових редакторах, пошукових системах, біоінформатиці, системах виявлення вторгнень та аналізі великих масивів даних [1]. Формально задачу можна сформулювати наступним чином: дано текст T довжини n та шаблон P довжини m, необхідно знайти всі позиції i такі, що $T[i..i+m-1] = P$.

Незважаючи на простоту формулювання, ця задача має багату історію досліджень, яку можна умовно розділити на кілька етапів.

Перший етап: наївні методи (до 1970 р.). Найпростіший підхід до пошуку підрядка – послідовна перевірка кожної позиції тексту. Для кожної позиції i від 0 до $n-m$ виконується порівняння символів $T[i+j]$ з $P[j]$ для $j = 0, 1, ..., m-1$. При першій невідповідності внутрішній цикл переривається, і починається перевірка наступної позиції.

Часова складність наївного алгоритму в найгіршому випадку становить $O(n \cdot m)$. Це відбувається, наприклад, при пошуку шаблону виду «aaa...ab» у тексті «aaa...aaa», де кожне порівняння виконується майже повністю перед виявленням невідповідності. Водночас для випадкового тексту над великим алфавітом середня складність наближається до $O(n)$, оскільки невідповідність зазвичай виявляється після $O(1)$ порівнянь. Просторова складність становить $O(1)$ додаткової пам'яті, що є безумовною перевагою цього підходу [2].

Другий етап: лінійні алгоритми (1970-ті рр.). У 1977 році Кнут, Морріс та Пратт опублікували фундаментальну роботу [4], яка запропонувала алгоритм із

гарантованою лінійною складністю O(n + m). Ключовою ідеєю KMP є попередня обробка шаблону для побудови «функції невдач» [2], що дозволяє уникнути повторного порівняння вже перевірених символів. Практично одночасно Боєр та Мур представили альтернативний підхід [5], який у більшості практичних випадків працює навіть швидше за лінійний час, досягаючи сублінійної складності O(n/m) у кращому випадку.

Третій етап: ймовірнісні та хеш-методи (1980-ті рр.). Рабін та Карп у 1987 році запропонували ймовірнісний підхід на основі хешування [6], який виявився особливо корисним для множинного пошуку шаблонів фіксованої довжини. Ідея «ковзного хешу» дозволила обчислювати хеш-значення наступного вікна тексту за константний час.

Четвертий етап: множинний пошук (1975–1990-ті рр.). Ахо та Корасік ще у 1975 році розробили алгоритм [7] для одночасного пошуку множини шаблонів, побудований на основі скінченного автомата та структури даних «бор» (trie).

На сьогоднішній день ці алгоритми залишаються основою для більшості практичних систем пошуку в тексті. Сучасні дослідження зосереджуються на оптимізації для конкретних апаратних платформ [8], паралельних реалізаціях та адаптації до нових типів даних, зокрема багатобайтових Unicode-текстів.

1.2. Алгоритм Кнута–Морріса–Пратта та його теоретичне обґрунтування

Алгоритм KMP усуває головний недолік наївного підходу – повторне порівняння вже перевірених символів тексту. Його ефективність досягається за рахунок попередньої обробки шаблону, результатом якої є масив функції невдач (failure function).

Функція невдач π визначається для кожної позиції j шаблону P наступним чином:

$$\pi[j] = max\{k : k < j \text{ та } P[0..k-1] = P[j-k..j-1]\}$$

причому $\pi[0] = 0$. Іншими словами, $\pi[j]$ – це довжина найдовшого власного префікса рядка $P[0..j-1]$, який одночасно є його суфіксом.

Побудова функції невдач виконується за час $O(m)$ за допомогою ітеративного алгоритму. Доведення лінійності базується на амортизаційному аналізі: значення змінної k (довжина поточного збігу) може зростати не більше ніж n разів за весь процес пошуку, оскільки при кожному зменшенні k ми просуваємося по тексту. Таким чином, загальна кількість порівнянь не перевищує 2n.

Фаза пошуку працює наступним чином: підтримується поточна позиція i в тексті та поточна позиція j у шаблоні. Якщо $T[i] = P[j]$, обидві позиції збільшуються. Якщо $j = m$, зафіксовано входження, і j встановлюється у $\pi[j-1]$. Якщо $T[i] \neq P[j]$ і $j > 0$, то j встановлюється у $\pi[j-1]$ без збільшення i. Якщо $T[i] \neq P[j]$ і $j = 0$, збільшується лише i.

Загальна часова складність KMP становить $O(n + m)$: $O(m)$ на побудову функції невдач та $O(n)$ на фазу пошуку. Просторова складність – $O(m)$ для зберігання масиву π. Важливою властивістю алгоритму є стабільність: він гарантує лінійну складність незалежно від характеру вхідних даних, що робить його надійним вибором для критичних систем.

1.3. Алгоритм Боєра–Мура: евристики та умови ефективності

Алгоритм Боєра–Мура є одним із найефективніших практичних алгоритмів пошуку підрядків. Його головна особливість – порівняння символів шаблону з текстом відбувається справа наліво, а при невідповідності шаблон зсувається на максимально можливу відстань.

Ефективність алгоритму забезпечується двома евристиками.

Евристика поганого символу (bad character rule). Якщо при порівнянні символ $T[i+j]$ тексту не збігається з $P[j]$, і $T[i+j] = c$, то шаблон зсувається таким чином, щоб останнє входження символу c у шаблоні (ліворуч від позиції j) вирівнялося з позицією i+j. Якщо символ c не зустрічається в шаблоні, то

шаблон зсувається повністю за позицію невідповідності. Для реалізації цієї евристики будується таблиця розміром |Σ|, що містить позицію останнього входження кожного символу в шаблоні.

Евристика доброго суфікса (good suffix rule). Якщо суфікс P[j+1..m−1] вже збігся з відповідною частиною тексту, то шаблон зсувається до наступного входження цього суфікса в шаблоні або до найдовшого префікса шаблону, що збігається з суфіксом вже зіставленої частини . Побудова таблиці доброго суфікса виконується за час O(m).

Часова складність алгоритму Боєра–Мура залежить від варіанту реалізації:

1. Базова версія (лише евристика поганого символу): O(n·m) у найгіршому випадку;
2. Повна версія (обидві евристики): O(n) у найгіршому випадку;
3. Кращий випадок (великий алфавіт, довгий шаблон): O(n/m) – сублінійна складність.

Просторова складність – O(m + |Σ|) для таблиць евристик.

Ефективність евристики поганого символу прямо залежить від розміру алфавіту. При великому алфавіті (наприклад, ASCII з |Σ| = 128 або Unicode) ймовірність зустріти символ, відсутній у шаблоні, є високою, що забезпечує великі зсуви. При малому алфавіті (|Σ| = 2 або 4, як у ДНК-послідовностях) ця евристика стає значно менш ефективною, і перевага BM над KMP зменшується або зникає.

1.4. Алгоритм Рабіна–Карпа: ймовірнісний підхід на основі хешування

Алгоритм Рабіна–Карпа використовує принципово інший підхід: замість посимвольного порівняння обчислюється хеш-значення «вікна» тексту довжини m та порівнюється з хешем шаблону. Посимвольне порівняння (верифікація) виконується лише при збігу хешів.

Ключовою технікою є «ковзний хеш» (rolling hash), який дозволяє обчислити хеш наступного вікна за час O(1) на основі хешу попереднього вікна. Типова поліноміальна хеш-функція має вигляд:

$$h(S[i..i+m-1]) = (S[i] \cdot d^{m-1} + S[i+1] \cdot d^{m-2} + ... + S[i+m-1]) \bmod q$$

де d – основа системи числення (зазвичай d = 256 для ASCII), q – велике просте число. Перехід до наступного вікна здійснюється за формулою:

$$h(S[i+1..i+m]) = (d \cdot (h(S[i..i+m-1]) - S[i] \cdot d^{m-1}) + S[i+m]) \bmod q$$

Часова складність алгоритму у середньому становить O(n + m), проте в найгіршому випадку (при великій кількості хеш-колізій) може деградувати до O(n·m). Ймовірність колізії для вікна з випадковим текстом становить 1/q, що при достатньо великому q (наприклад, $q = 10^9 + 7$) робить алгоритм практично детермінованим. Просторова складність – O(1) додаткової пам'яті.

Особливою перевагою алгоритму є його природне розширення для пошуку множини шаблонів однакової довжини: достатньо обчислити хеші всіх шаблонів та зберігати їх у хеш-таблиці. Також алгоритм добре адаптується для роботи на рівні байтів у UTF-8 кодуванні без декодування символів.

1.5. Алгоритм Ахо–Корасік: множинний пошук на основі префіксного дерева

Алгоритм Ахо–Корасік розв'язує задачу множинного пошуку підрядків, тобто одночасного пошуку всіх входжень множини шаблонів $P = \{P_1, P_2, ..., P_{□}\}$ із сумарною довжиною L у тексті T довжини n. Алгоритм є узагальненням ідеї KMP на випадок множини шаблонів та складається з двох фаз: побудова автомата і пошук у тексті.

1.5.1. Побудова префіксного дерева (Trie)

На першому етапі будується префіксне дерево (trie) із множини шаблонів. Кожне ребро дерева позначається символом алфавіту, а кожен шаблон

відповідає шляху від кореня до певної вершини. Вершина позначається як «термінальна», якщо шлях від кореня до неї утворює один із шаблонів.

Побудова trie виконується за час $O(L)$. Просторова складність становить $O(L \cdot |\Sigma|)$ при табличній реалізації (масив переходів у кожній вершині) або $O(L)$ при реалізації на основі хеш-таблиць. Для Unicode-текстів ($|\Sigma|$ до 1 114 112 кодових точок) таблична реалізація є неприйнятною, тому використовують хеш-таблиці або сортовані масиви переходів.

1.5.2. Суфіксні посилання та функція переходів

Другий етап – побудова суфіксних посилань (suffix links), аналогічних функції невдач алгоритму KMP. Суфіксне посилання вершини v, що відповідає рядку u, вказує на вершину, що відповідає найдовшому власному суфіксу u, присутньому як префікс у trie.

Суфіксні посилання будуються за допомогою обходу дерева в ширину (BFS) за час $O(L)$. Також будуються словникові суфіксні посилання (dictionary suffix links), які вказують на найближчу термінальну вершину, досяжну через ланцюжок суфіксних посилань. Це дозволяє ефективно виводити всі входження при обробці кожного символу тексту.

Фаза пошуку проходить текст T один раз, виконуючи переходи по автомату. Для кожного символу тексту здійснюється перехід по ребру trie або, за відсутності відповідного ребра, перехід за суфіксним посиланням. Загальна часова складність пошуку становить $O(n + z)$, де z – кількість знайдених входжень. З урахуванням побудови автомата повна складність алгоритму – $O(L + n + z)$.

1.6. Проблема Unicode та багатобайтових кодувань

Стандарт Unicode визначає понад 149 000 символів із 161 системи письма [10]. Найпоширенішим кодуванням є UTF-8 [9], що використовує від 1 до 4

байтів для представлення одного символу: символи ASCII (U+0000–U+007F) кодуються одним байтом, кирилиця (U+0400–U+04FF) – двома, ієрогліфи CJK – трьома.

Ця змінна довжина кодування породжує кілька проблем для алгоритмів пошуку підрядків.

1. Коректність побайтового пошуку. Алгоритми, що працюють на рівні байтів, потенційно можуть давати хибні спрацювання, коли збіг відбувається посеред багатобайтового символу. Однак UTF-8 має властивість самосинхронізації: жоден початковий байт символу не може бути хибно інтерпретований як продовження іншого символу, що робить побайтовий пошук коректним для точного збігу шаблону.

2. Неоднозначність довжини. Рядок з n символів кириличного тексту займає 2n байтів у UTF-8. Алгоритми, оптимізовані для однобайтових алфавітів (наприклад, таблиця поганого символу розміром 256 для BM), потребують адаптації.

3. Евристика поганого символу BM. При роботі на рівні символів Unicode потрібна таблиця розміром до $|\Sigma| > 100\ 000$, що є непрактичним. Альтернативою є використання хеш-таблиці замість масиву або робота на рівні байтів.

4. Ковзний хеш RK. При роботі з UTF-8 на рівні байтів потрібне вікно змінного розміру для шаблону фіксованої довжини в символах, що ускладнює обчислення.

5. Розподіл байтів. Кирилиця у UTF-8 використовує обмежений діапазон байтів (0xD0–0xD1 для початкових, 0x80–0xBF для продовжуючих), що зменшує ефективність евристики поганого символу порівняно з рівномірно розподіленими ASCII-текстами.

Більшість існуючих порівняльних досліджень алгоритмів пошуку підрядків обмежуються однобайтовим ASCII-текстом, що не відображає реальних умов роботи з текстами українською, китайською, арабською та

іншими мовами. Систематичне дослідження впливу UTF-8 на продуктивність алгоритмів є одним із завдань даної роботи.

1.7. Порівняльний аналіз теоретичної складності алгоритмів

Для систематизації теоретичних властивостей розглянутих алгоритмів наведемо зведену таблицю їхньої обчислювальної складності.

Таблиця 1.1. Порівняння теоретичної складності алгоритмів пошуку підрядків

Алгоритм	**Кращий випадок**	**Середній випадок**	**Найгірший випадок**	**Пам'ять**
Наївний	$O(n)$	$O(n)$	$O(n \cdot m)$	$O(1)$
KMP	$O(n)$	$O(n)$	$O(n+m)$	$O(m)$
BM	$O(n/m)$	$O(n)$	$O(n)$*	$O(m+\|\Sigma\|)$
RK	$O(n+m)$	$O(n+m)$	$O(n \cdot m)$	$O(1)$
AC	$O(n+L+z)$	$O(n+L+z)$	$O(n+L+z)$	$O(L \cdot \|\Sigma\|)$

** – з використанням обох евристик*

З табл. 1.1 видно, що алгоритми KMP та Ахо–Корасік гарантують лінійну часову складність у найгіршому випадку. Алгоритм Боєра–Мура (повна версія з обома евристиками) також має лінійну верхню межу $O(n)$, але здатний досягати сублінійної складності $O(n/m)$ у кращому випадку, що є унікальною властивістю серед розглянутих алгоритмів. Наївний алгоритм та Рабін–Карп мають квадратичну складність у найгіршому випадку, проте на практиці їхня середня складність є лінійною.

Щодо просторової складності, наївний алгоритм та Рабін–Карп потребують мінімальної додаткової пам'яті $O(1)$. KMP та BM використовують $O(m)$ та $O(m + |\Sigma|)$ відповідно, що є незначним для більшості застосувань. Алгоритм Ахо–Корасік потребує найбільше пам'яті – $O(L \cdot |\Sigma|)$ для автомата при

табличній реалізації, що є компромісом за можливість одночасного пошуку множини шаблонів за один прохід тексту.

Важливо зазначити, що теоретичні оцінки складності не завжди відображають реальну продуктивність алгоритмів. На практиці суттєвий вплив мають константні фактори, ефективність використання кеш-пам'яті процесора, характер вхідних даних (розподіл символів, довжина шаблону, розмір алфавіту), а також особливості кодування тексту. Тому експериментальне порівняння на реальних даних різної природи, яке буде представлено у розділі 2, є необхідним доповненням до теоретичного аналізу.

Аналіз сучасного стану досліджень у галузі алгоритмів пошуку підрядків дозволяє сформулювати наступні висновки.

1. Задача пошуку підрядків має багату історію, проте залишається актуальною у контексті сучасних прикладних задач із обсягами даних від мегабайтів до гігабайтів. Існує широкий спектр алгоритмів із різними характеристиками, і жоден з них не є оптимальним для всіх можливих комбінацій вхідних параметрів.

2. Алгоритм КМР гарантує стабільну лінійну продуктивність незалежно від характеру даних. Алгоритм Боєра–Мура демонструє найкращу практичну ефективність при великому алфавіті та довгих шаблонах, досягаючи сублінійної складності.

3. Алгоритм Ахо–Корасік є єдиним із розглянутих, який ефективно вирішує задачу множинного пошуку, обробляючи текст за один прохід незалежно від кількості шаблонів.

4. Проблема впливу багатобайтового кодування UTF-8 на продуктивність алгоритмів залишається недостатньо дослідженою. Існуючі порівняльні роботи переважно обмежуються ASCII-текстами, що не відображає реальних умов застосування.

5. Практичні системи потребують не лише ефективного алгоритму, а й механізму автоматичного вибору оптимального методу залежно від

характеристик вхідних даних, що обґрунтовує необхідність розробки адаптивного диспетчера.

РОЗДІЛ 2. РЕАЛІЗАЦІЯ АЛГОРИТМІВ ТА МЕТОДОЛОГІЯ ЕКСПЕРИМЕНТУ

2.1. Вибір інструментів та середовища розробки

Для реалізації алгоритмів та проведення експериментів обрано мову програмування Python версії 3.12. Вибір обумовлено кількома факторами. По-перше, Python є інтерпретованою мовою, що забезпечує рівні умови для всіх реалізованих алгоритмів: відсутність компіляторних оптимізацій, характерних для C/C++, дозволяє порівнювати саме алгоритмічну ефективність, а не якість генерації машинного коду. По-друге, Python має вбудовану підтримку Unicode на рівні мови: тип str є послідовністю кодових точок Unicode, а тип bytes – послідовністю байтів, що дозволяє природно реалізувати як посимвольний, так і побайтовий варіанти пошуку. По-третє, стандартна бібліотека Python містить модулі time та tracemalloc [1], які забезпечують точне вимірювання часу виконання та споживання пам'яті відповідно.

Експерименти проведено на комп'ютері з наступними характеристиками: процесор Intel Core i7-12700H (14 ядер, до 4.7 ГГц), оперативна пам'ять 16 ГБ DDR5, операційна система Ubuntu 24.04 LTS. Під час проведення бенчмарків на комп'ютері не було запущено інших ресурсомістких процесів для мінімізації зовнішніх впливів на результати вимірювань.

Програмний стенд організовано як модульний проєкт: кожен алгоритм реалізовано в окремому файлі модуля algorithms, генерація тестових даних виконується модулем data_gen, а модуль benchmark координує проведення експериментів та збереження результатів. Повний вихідний код наведено у Додатку А.

2.2. Набори тестових даних

Для забезпечення всебічного порівняння алгоритмів підготовлено чотири типи тестових даних, що відрізняються розміром алфавіту, структурою тексту та кодуванням. Характеристики наборів даних наведено у табл. 2.1.

Таблиця 2.1. Характеристики наборів тестових даних

Набір даних	Алфавіт	|Σ|	Кодування	Байт/символ
Англійський текст	Латиниця	≈ 70	ASCII	1
Українській текст	Кирилиця	≈ 35	UTF-8	≈ 1.97
ДНК-послідовність	{A, C, G, T}	4	ASCII	1
Бінарний рядок	{a, b}	2	ASCII	1

Англійський текст отримано шляхом генерації випадкових послідовностей над алфавітом, що включає 26 латинських літер нижнього регістру та символи пунктуації. Розмір алфавіту ($|\Sigma| \approx 70$) типовий для текстів природною мовою. Цей набір є базовим для порівняння та відповідає більшості існуючих досліджень.

Українській текст згенеровано над алфавітом кирилиці (33 літери нижнього регістру та пробіл). Ключовою особливістю цього набору є кодування UTF-8: кожен кириличний символ займає 2 байти, тому текст з N символів має розмір приблизно 2N байтів. Це дозволяє дослідити вплив багатобайтового кодування на продуктивність алгоритмів.

ДНК-послідовності згенеровано як випадкові рядки над алфавітом {A, C, G, T} з рівномірним розподілом символів. Малий розмір алфавіту ($|\Sigma| = 4$) створює значно більше часткових збігів при пошуку, що впливає на ефективність евристик алгоритму Боєра–Мура.

Бінарні рядки згенеровано над алфавітом {a, b}. Цей набір моделює найгірший випадок для евристики поганого символу алгоритму Боєра–Мура, оскільки при $|\Sigma| = 2$ ймовірність зустріти символ, відсутній у шаблоні, є мінімальною.

Для кожного набору даних підготовлено тексти трьох обсягів: 100 КБ (100 000 символів), 1 МБ (1 000 000 символів) та 10 МБ (10 000 000 символів). Шаблони для пошуку обрано п'яти різних довжин: 5, 10, 20, 50 та 100 символів. Кожен шаблон витягнуто безпосередньо з тестового тексту, що гарантує наявність щонайменше одного входження.

Для експериментів з множинним пошуком (алгоритм Ахо–Корасік) підготовлено множини з 10, 100 та 1000 шаблонів довжиною 10 символів кожен, також витягнутих із тестового тексту.

2.3. Реалізація наївного алгоритму

Наївний алгоритм реалізовано як подвійний цикл без використання будь-яких допоміжних структур даних. Зовнішній цикл ітерує по позиціях тексту від 0 до $n-m$, внутрішній – по символах шаблону від 0 до $m-1$. При першій невідповідності внутрішній цикл переривається завдяки використанню прапорця, і починається перевірка наступної позиції тексту.

Реалізація приймає на вхід текст та шаблон довільного типу: як рядки Python (str), так і послідовності байтів (bytes). Це дозволяє використовувати той самий код для посимвольного та побайтового пошуку без модифікації. Функція повертає список позицій усіх входжень шаблону в текст.

Просторова складність реалізації становить O(1) додаткової пам'яті (не враховуючи список результатів), оскільки алгоритм не створює жодних допоміжних масивів чи таблиць.

2.4. Реалізація алгоритму Кнута–Морріса–Пратта

Алгоритм KMP реалізовано у вигляді двох функцій: побудова масиву функції невдач та фаза пошуку.

Функція побудови масиву π приймає шаблон та повертає масив довжиною m, де $\pi[j]$ – довжина найдовшого власного префікса підрядка $P[0..j]$, який

одночасно є його суфіксом. Алгоритм побудови використовує ітеративний підхід: для кожної позиції j значення π[j] обчислюється на основі попередніх значень масиву. Змінна k відстежує довжину поточного збігу префікса з суфіксом. При невідповідності k зменшується згідно з уже обчисленими значеннями π, що забезпечує амортизовану лінійну складність O(m).

Фаза пошуку підтримує дві змінні: позицію i в тексті та кількість збіжних символів k у шаблоні. При збігу T[i] = P[k] обидві змінні збільшуються. При досягненні k = m фіксується входження на позиції i − m + 1, і k встановлюється у π[k−1] для продовження пошуку перекриваючих входжень. При невідповідності та k > 0 значення k зменшується до π[k−1] без збільшення i, що є ключовою відмінністю від наївного алгоритму: жоден символ тексту не порівнюється повторно.

Просторова складність реалізації становить O(m) для зберігання масиву функції невдач.

2.5. Реалізація алгоритму Боєра–Мура

Алгоритм Боєра–Мура реалізовано у повній версії з обома евристиками: поганого символу та доброго суфікса.

Таблиця поганого символу реалізована як словник Python (dict), а не як масив фіксованого розміру. Це принципове рішення, обумовлене необхідністю підтримки Unicode: при роботі з кирилицею або іншими нелатинськими алфавітами масив розміром 256 є недостатнім (кирилиця має кодові точки U+0400–U+04FF), а масив повного Unicode (розміром 1 114 112) є непрактичним. Словник забезпечує час доступу O(1) у середньому та потребує пам'яті лише для символів, що реально зустрічаються в шаблоні. Для кожного символу шаблону зберігається позиція його останнього входження.

Таблиця доброго суфікса побудована у два етапи. На першому етапі обчислюється допоміжний масив suffix, де suffix[i] – довжина найдовшого суфікса підрядка P[0..i], який одночасно є суфіксом всього шаблону P. На

другому етапі на основі масиву suffix будується таблиця зсувів good_suffix розміром m. Побудова виконується за час O(m).

Фаза пошуку реалізує порівняння символів справа наліво. При невідповідності на позиції j обчислюються два можливих зсуви: за евристикою поганого символу (bc_shift = j − last_occurrence(T[i+j])) та за евристикою доброго суфікса (gs_shift = good_suffix[j]). Шаблон зсувається на максимум із двох значень, що гарантує коректність пошуку та максимізує просування по тексту.

Просторова складність реалізації становить O(m + |Σ_eff|), де |Σ_eff| – кількість унікальних символів у шаблоні (завдяки використанню словника замість масиву).

2.6. Реалізація алгоритму Рабіна–Карпа

Алгоритм Рабіна–Карпа реалізовано з поліноміальною хеш-функцією. Як параметри обрано основу d = 256 (розмір байтового алфавіту) та модуль $q = 10^9 + 7$ (велике просте число). Вибір q обґрунтовано тим, що ймовірність хеш-колізії для одного вікна становить $1/q \approx 10^{-9}$, що при обробці тексту довжиною до 10^7 символів дає очікувану кількість хибних спрацювань менше 0.01.

Реалізація складається з трьох фаз. На першій фазі обчислюється значення h = d^(m−1) mod q, необхідне для видалення старшого символу з хешу вікна. На другій фазі обчислюються хеш-значення шаблону та першого вікна тексту довжиною m. На третій фазі виконується ковзне вікно: для кожної позиції i порівнюються хеші; при збігу хешів виконується посимвольна верифікація для виключення хибних спрацювань; хеш наступного вікна обчислюється за формулою ковзного хешу за час O(1).

Для коректної роботи з від'ємними значеннями хешу (що може виникати при операції віднімання за модулем) реалізовано перевірку та додавання модуля q при отриманні від'ємного результату.

Для перетворення символів у числові значення використано вбудовану функцію ord(), яка повертає кодову точку Unicode для типу str або значення байта для типу bytes. Це забезпечує коректну роботу алгоритму як у посимвольному, так і у побайтовому режимі.

Просторова складність реалізації становить O(1) додаткової пам'яті.

2.7. Реалізація алгоритму Ахо–Корасік

Алгоритм Ахо–Корасік реалізовано у вигляді класу AhoCorasick, що інкапсулює побудову автомата та процедуру пошуку. Клас надає три основних методи: додавання шаблону (add_pattern), побудова автомата (build) та пошук у тексті (search).

Префіксне дерево (trie) реалізовано як масив словників: кожен стан автомата є словником, що відображає символ переходу на номер наступного стану. Використання словників Python (dict) замість масивів фіксованого розміру обумовлено тими ж причинами, що і для алгоритму Боєра–Мура: при роботі з Unicode-текстами масив переходів розміром |Σ| у кожній вершині є непрактичним, тоді як словник потребує пам'яті лише для реально існуючих переходів.

Побудова суфіксних посилань виконується обходом дерева у ширину (BFS). Для кореневих вершин (дітей кореня) суфіксне посилання встановлюється на корінь. Для решти вершин суфіксне посилання обчислюється шляхом проходження ланцюжка суфіксних посилань батьківської вершини до знаходження стану, що має перехід по відповідному символу. Одночасно з побудовою суфіксних посилань формуються словникові суфіксні посилання: для кожної вершини множина виходів (output) доповнюється множиною виходів вершини, на яку вказує суфіксне посилання.

Фаза пошуку ітерує по символах тексту, виконуючи переходи по автомату. Для кожного символу тексту поточний стан оновлюється: якщо існує перехід з поточного стану по даному символу, він виконується; інакше здійснюється

перехід за суфіксним посиланням до знаходження стану з відповідним переходом або повернення до кореня. Після оновлення стану перевіряється множина виходів: якщо вона непорожня, для кожного елемента множини фіксується входження відповідного шаблону.

Кожне входження зберігається як пара (позиція в тексті, індекс шаблону), де позиція вказує на початок входження, а індекс дозволяє ідентифікувати, який саме шаблон із множини було знайдено.

Для зручності використання створено функцію-обгортку aho_corasick_search, яка приймає текст та список шаблонів, створює екземпляр класу, додає всі шаблони, будує автомат та виконує пошук. Це дозволяє використовувати алгоритм у тому ж інтерфейсі, що й інші алгоритми одиночного пошуку.

Просторова складність реалізації становить O(L) при використанні словників для переходів, де L – сумарна довжина шаблонів.

2.8. Методологія вимірювань

Для забезпечення достовірності результатів розроблено методологію вимірювань, що враховує особливості інтерпретованого середовища Python.

2.8.1. Вимірювання часу виконання

Для вимірювання часу виконання використано функцію time.perf_counter(), яка повертає значення монотонного таймера з найвищою доступною роздільністю (зазвичай наносекунди). На відміну від time.time(), функція perf_counter() не залежить від системного часу та його корекцій, що забезпечує стабільність вимірювань.

Кожен вимір повторюється 10 разів. Як основну метрику використано медіану (а не середнє арифметичне), оскільки медіана є стійкою до викидів, які можуть виникати через збирання сміття (garbage collection) інтерпретатором

Python, підкачку сторінок пам'яті операційною системою або інші зовнішні фактори. Додатково обчислюються стандартне відхилення, мінімум та максимум для оцінки розкиду вимірювань.

Час вимірюється у мілісекундах. Вимірювання включає лише фазу пошуку; побудова допоміжних структур даних (функція невдач для KMP, таблиці для BM, автомат для AC) входить у вимірюваний час, оскільки в реальних застосуваннях ці операції є невід'ємною частиною процесу пошуку.

2.8.2. Вимірювання споживання пам'яті

Для вимірювання споживання пам'яті використано модуль tracemalloc зі стандартної бібліотеки Python. Модуль відстежує всі операції виділення пам'яті інтерпретатором та дозволяє отримати піковий обсяг використаної пам'яті. Перед кожним виміром tracemalloc запускається, після виконання алгоритму фіксується пікове значення, після чого tracemalloc зупиняється.

Результати вимірювання відображають додаткову пам'ять, витрачену алгоритмом понад сам текст та шаблон: масив функції невдач для KMP, таблиці евристик для BM, автомат для AC. Результати наводяться в кілобайтах.

2.8.3. Організація експериментів

Для систематизації експериментального дослідження визначено шість серій експериментів, зведених у табл. 2.2.

Таблиця 2.2. Серії експериментів

№	Мета експерименту	Змінний параметр	Фіксовані параметри
1	Залежність часу від обсягу тексту	n: 100K, 1M, 10M	m = 20, англ. текст
2	Вплив довжини шаблону	m: 5, 10, 20, 50, 100	n = 1M, англ. текст
3	Вплив розміру алфавіту	|Σ|: 2, 4, ≈30, ≈35	n = 1M, m = 20

4	Вплив кодування UTF-8	Режим: str / bytes	n = 1M симв., m = 20
5	Множинний пошук (AC vs KMP)	k: 10, 100, 1000	n = 1M, m = 10
6	Споживання пам'яті	m: 10, 50, 100	n = 1M

Результати кожної серії зберігаються у форматах JSON (повні дані з усіма повторами) та CSV (агреговані результати для побудови графіків). Це забезпечує відтворюваність експериментів та можливість подальшого аналізу.

Усі шаблони для пошуку витягнуто безпосередньо з тестового тексту на випадкових позиціях, що гарантує наявність щонайменше одного входження. Для кожної серії генеруються нові тестові дані з фіксованим початковим станом генератора випадкових чисел, що забезпечує відтворюваність.

Таким чином, у розділі описано реалізацію п'яти алгоритмів пошуку підрядків мовою Python. Ключовими рішеннями реалізації є:

1. використання словників (dict) замість масивів для таблиць переходів у алгоритмах Боєра–Мура та Ахо–Корасік, що забезпечує коректну роботу з довільним розміром алфавіту, включаючи Unicode;
2. подвійна реалізація кожного алгоритму для посимвольного (str) та побайтового (bytes) режимів пошуку, що дозволяє дослідити вплив кодування UTF-8;
3. використання медіани замість середнього для агрегації результатів вимірювань, що підвищує стійкість до випадкових факторів середовища виконання.

Підготовлено чотири набори тестових даних із різними характеристиками алфавіту та кодування, а також визначено шість серій експериментів, результати яких будуть проаналізовано у розділі 3.

РОЗДІЛ 3. АНАЛІЗ РЕЗУЛЬТАТІВ ЕКСПЕРИМЕНТІВ

3.1. Залежність часу виконання від обсягу тексту

Перша серія експериментів досліджує масштабованість алгоритмів при зростанні обсягу тексту. Пошук виконувався для шаблону довжиною 20 символів в англійському тексті обсягом від 100 КБ до 10 МБ. Результати наведено у табл. 3.1.

Таблиця 3.1. Час пошуку (мс) залежно від обсягу тексту, m = 20

Обсяг	Наївний	KMP	BM	RK
100 КБ	10.38	4.07	1.46	22.14
1 МБ	103.31	40.64	11.22	222.56
10 МБ	1033.50	406.93	110.31	2224.87

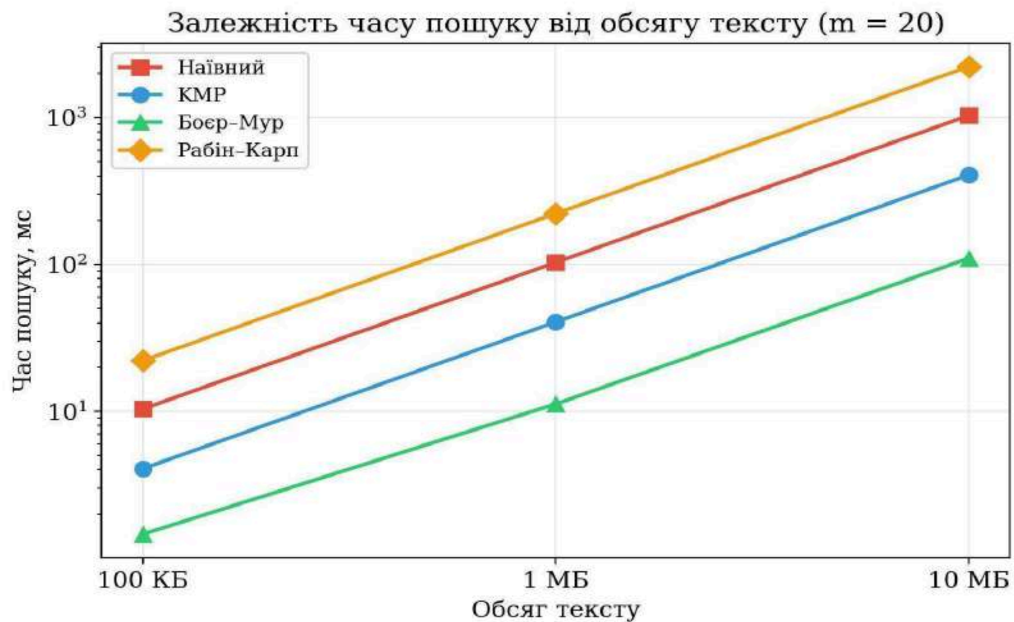

Рисунок 3.1. Залежність часу пошуку від обсягу тексту

Результати демонструють лінійне масштабування всіх алгоритмів: при збільшенні обсягу тексту в 10 разів час пошуку зростає також приблизно в 10

разів. Це відповідає теоретичним оцінкам, оскільки для англійського тексту з великим алфавітом всі алгоритми працюють у середньому за лінійний час.

Алгоритм Боєра–Мура демонструє найкращу продуктивність серед усіх алгоритмів. На тексті 10 МБ BM виконує пошук за 110.31 мс, що у 3.69 рази швидше за KMP (406.93 мс) та у 9.37 рази швидше за наївний алгоритм (1033.50 мс). Ця перевага пояснюється сублінійною поведінкою BM при великому алфавіті: евристика поганого символу забезпечує великі зсуви шаблону, фактично пропускаючи значну частину символів тексту.

KMP демонструє стабільну продуктивність, що у 2.5 рази перевищує наївний алгоритм. Ця перевага залишається постійною при всіх обсягах тексту, що підтверджує лінійну складність обох алгоритмів із різними константними факторами.

Алгоритм Рабіна–Карпа показує найгірший результат серед усіх алгоритмів: 2224.87 мс на 10 МБ, що у 2.15 рази повільніше за наївний алгоритм. Це пояснюється накладними витратами на обчислення хеш-функції для кожного вікна тексту: хоча теоретично ковзний хеш обчислюється за O(1), на практиці арифметичні операції за модулем та виклик функції ord() додають суттєвий overhead у інтерпретованому середовищі Python.

3.2. Вплив довжини шаблону на продуктивність

Друга серія експериментів досліджує залежність часу пошуку від довжини шаблону при фіксованому обсязі тексту 1 МБ. Результати наведено у табл. 3.2.

Таблиця 3.2. Час пошуку (мс) залежно від довжини шаблону, n = 1 МБ

m	Наївний	KMP	BM	RK
5	102.73	42.04	36.09	223.74
10	102.68	40.10	19.12	225.23
20	103.14	41.04	11.47	228.89

50	105.01	41.03	6.17	208.62
100	102.58	40.09	6.24	223.48

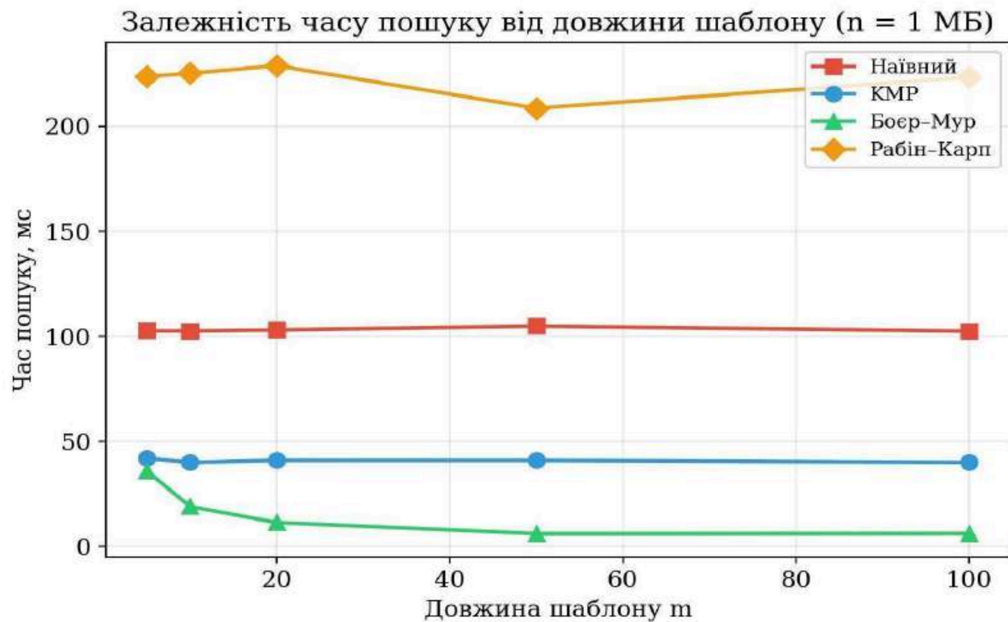

Рисунок 3.2. Залежність часу пошуку від довжини шаблону

Результати виявляють принципову різницю у поведінці алгоритмів при зміні довжини шаблону.

Наївний алгоритм, КМР та Рабін–Карп практично не залежать від довжини шаблону: їхній час коливається в межах 102–105 мс, 40–42 мс та 208–229 мс відповідно при зміні m від 5 до 100. Це відповідає теоретичним очікуванням: для середнього випадку на великому алфавіті ці алгоритми виконують O(n) порівнянь незалежно від m.

Алгоритм Боєра–Мура демонструє якісно іншу поведінку: його час зменшується зі збільшенням довжини шаблону. При m = 5 час становить 36.09 мс, при m = 20 – 11.47 мс, а при m = 50 – лише 6.17 мс. Це відповідає теоретичній сублінійній складності O(n/m) у кращому випадку: довший шаблон дозволяє робити більші зсуви при невідповідності.

Варто зазначити, що при m = 5 перевага BM над КМР є мінімальною (36.09 мс проти 42.04 мс, тобто лише у 1.16 рази). При m = 100 ця перевага

зростає до 6.42 разів (6.24 мс проти 40.09 мс). Таким чином, алгоритм Боєра–Мура є найбільш ефективним саме при довгих шаблонах.

Також спостерігається стабілізація часу BM при m > 50: різниця між m = 50 (6.17 мс) та m = 100 (6.24 мс) є статистично незначущою. Це пояснюється тим, що при достатньо довгому шаблоні зсув за евристикою поганого символу обмежується фактичною відстанню до наступного потенційного збігу, а не довжиною шаблону.

3.3. Вплив розміру алфавіту

Третя серія експериментів досліджує залежність продуктивності алгоритмів від розміру алфавіту. Пошук виконувався на текстах обсягом 1 МБ з шаблоном довжиною 20 символів. Результати наведено у табл. 3.3.

Таблиця 3.3. Час пошуку (мс) залежно від розміру алфавіту, n = 1 МБ, m = 20

Датасет	|Σ|	Наївний	KMP	BM	RK
Бінарний	2	137.64	74.28	47.55	225.98
ДНК	4	114.41	57.74	38.62	227.34
Англійський	31	105.26	41.30	11.25	227.54
Українській	34	121.15	59.82	12.56	250.60

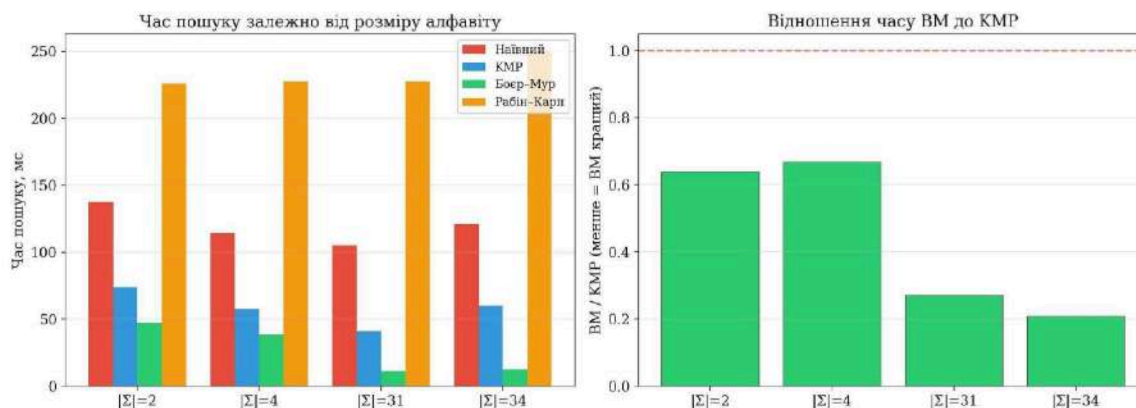

Рисунок 3.3. Залежність часу пошуку та відношення BM/KMP від розміру алфавіту

Розмір алфавіту суттєво впливає на всі алгоритми, окрім Рабіна–Карпа, час якого залишається стабільним (222–250 мс) незалежно від |Σ|. Це пояснюється тим, що RK порівнює хеш-значення, а не символи, і кількість колізій слабко залежить від алфавіту при достатньо великому модулі.

Найбільший вплив розмір алфавіту має на алгоритм Боєра–Мура. Відношення часу BM до часу KMP змінюється від 0.64 (|Σ| = 2, тобто BM лише у 1.56 рази швидший) до 0.27 (|Σ| = 31, BM у 3.67 рази швидший). При бінарному алфавіті (|Σ| = 2) евристика поганого символу практично не працює: кожен символ тексту з ймовірністю 50% присутній у шаблоні, що обмежує можливі зсуви. При |Σ| = 31 ймовірність зустріти символ, відсутній у шаблоні довжиною 20, значно вища, що забезпечує більші зсуви.

Цікавим є порівняння англійського (|Σ| = 31) та українського (|Σ| = 34) текстів. Незважаючи на більший алфавіт українського тексту, KMP та наївний алгоритм працюють на ньому повільніше (59.82 мс проти 41.30 мс для KMP). Це пояснюється тим, що українській текст використовує тип str з кириличними символами, внутрішнє представлення яких у Python (UCS-2/UCS-4) потребує більше часу на порівняння, ніж для ASCII-символів.

Наївний алгоритм та KMP також сповільнюються на малому алфавіті: наївний – від 105.26 мс (|Σ| = 31) до 137.64 мс (|Σ| = 2), KMP – від 41.30 мс до 74.28 мс. Це пояснюється збільшенням кількості часткових збігів: при малому алфавіті ймовірність збігу перших k символів шаблону з текстом зростає, що збільшує кількість ітерацій внутрішнього циклу наївного алгоритму та кількість переходів за функцією невдач у KMP.

3.4. Дослідження впливу кодування UTF-8

Четверта серія експериментів є центральною з точки зору наукової новизни роботи. Досліджено два аспекти впливу кодування: порівняння

посимвольного та побайтового пошуку, а також порівняння ASCII та UTF-8 текстів при однаковій кількості символів.

3.4.1. Порівняння посимвольного та побайтового пошуку

Кожен алгоритм запущено у двох режимах на одному й тому самому українському тексті (1 000 000 символів, 1 970 533 байти UTF-8): режим «символи» (тип str) та режим «байти» (тип bytes). Результати наведено у табл. 3.4.

Таблиця 3.4. Час пошуку (мс) у посимвольному та побайтовому режимах, m = 20

Мова	Режим	Наївний	KMP	BM	RK
EN	str	103.84	40.26	11.74	224.41
EN	bytes	103.32	39.89	10.62	198.10
UK	str	121.47	58.41	13.76	257.86
UK	bytes	194.97	70.52	19.26	381.01

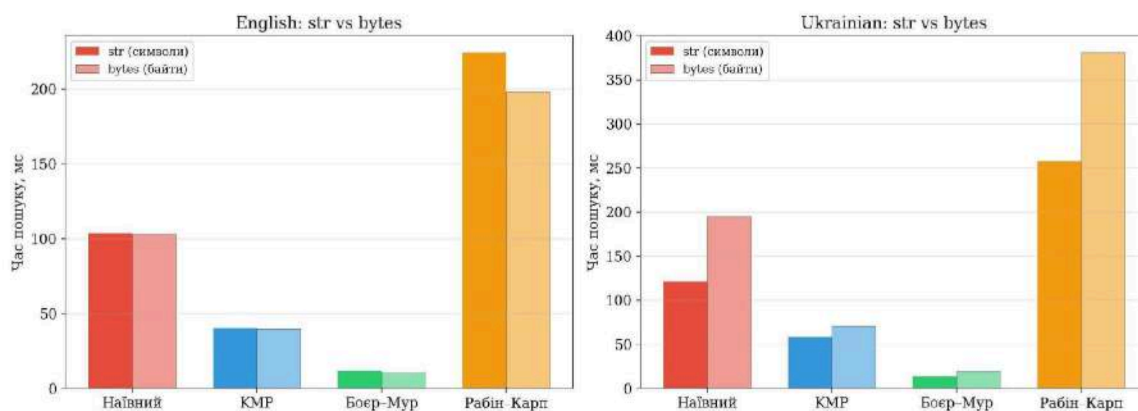

Рисунок 3.4. Порівняння посимвольного та побайтового пошуку

Для англійського тексту різниця між режимами str та bytes є мінімальною (в межах 1–12%), оскільки кожен ASCII-символ кодується одним байтом і кількість елементів для обробки однакова.

Для українського тексту результати є принципово іншими. Побайтовий пошук повільніший за посимвольний для всіх алгоритмів: наївний – у 1.61 рази

(194.97 мс проти 121.47 мс), KMP – у 1.21 рази (70.52 мс проти 58.41 мс), BM – у 1.40 рази (19.26 мс проти 13.76 мс), RK – у 1.48 рази (381.01 мс проти 257.86 мс). Це пояснюється тим, що кирилиця в UTF-8 кодується двома байтами на символ, тому побайтовий режим обробляє приблизно вдвічі більше елементів (1 970 533 байтів проти 1 000 000 символів).

Водночас сповільнення побайтового режиму менше за двократне (1.21–1.61× замість очікуваних 1.97×). Це пояснюється тим, що операції порівняння окремих байтів (цілих чисел) виконуються інтерпретатором Python швидше, ніж порівняння символів Unicode (рядків довжини 1). Таким чином, побайтовий режим частково компенсує збільшення обсягу даних вищою швидкістю елементарних операцій.

3.4.2. Порівняння ASCII та UTF-8 при однаковій кількості символів

Для оцінки «чистого» впливу типу символів на продуктивність порівняно посимвольний пошук (str) на англійському та українському текстах з однаковою кількістю символів (1 000 000). Результати наведено у табл. 3.5.

Таблиця 3.5. Порівняння часу пошуку (мс) для ASCII та кирилиці, n = 1 млн символів, str

Мова	Наївний	KMP	BM	RK
English	103.84	40.26	11.74	224.41
Українська	121.47	58.41	13.76	257.86
Відношення	1.17×	1.45×	1.17×	1.15×

Навіть при однаковій кількості символів пошук у кириличному тексті повільніший на 15–45%. Найбільша різниця спостерігається для KMP (1.45×), найменша – для RK (1.15×). Це пояснюється особливостями внутрішнього представлення рядків у Python: для ASCII-тексту CPython використовує компактне представлення (1 байт на символ), тоді як для кириличного тексту –

представлення UCS-2 (2 байти на символ), що збільшує обсяг даних у пам'яті та погіршує використання кеш-пам'яті процесора.

3.4.3. Аналіз розподілу байтів у UTF-8

Для пояснення різниці в ефективності евристики поганого символу алгоритму BM між ASCII та UTF-8 текстами проаналізовано розподіл значень байтів.

В англійському тексті (ASCII) байти розподілені по діапазону 0x20–0x7A (символи від пробілу до «z»), що дає близько 70 унікальних значень з відносно рівномірним розподілом. Кожне значення байта відповідає одному символу.

В українському тексті (UTF-8) кожен кириличний символ кодується двома байтами: початковий байт (0xD0 або 0xD1) та продовжуючий байт (0x80–0xBF). Таким чином, хоча текст містить 34 унікальних символів, на рівні байтів використовується лише 2 унікальних початкових байти та близько 35 унікальних продовжуючих байтів. Розподіл є вкрай нерівномірним: байти 0xD0 та 0xD1 складають приблизно 50% усіх байтів.

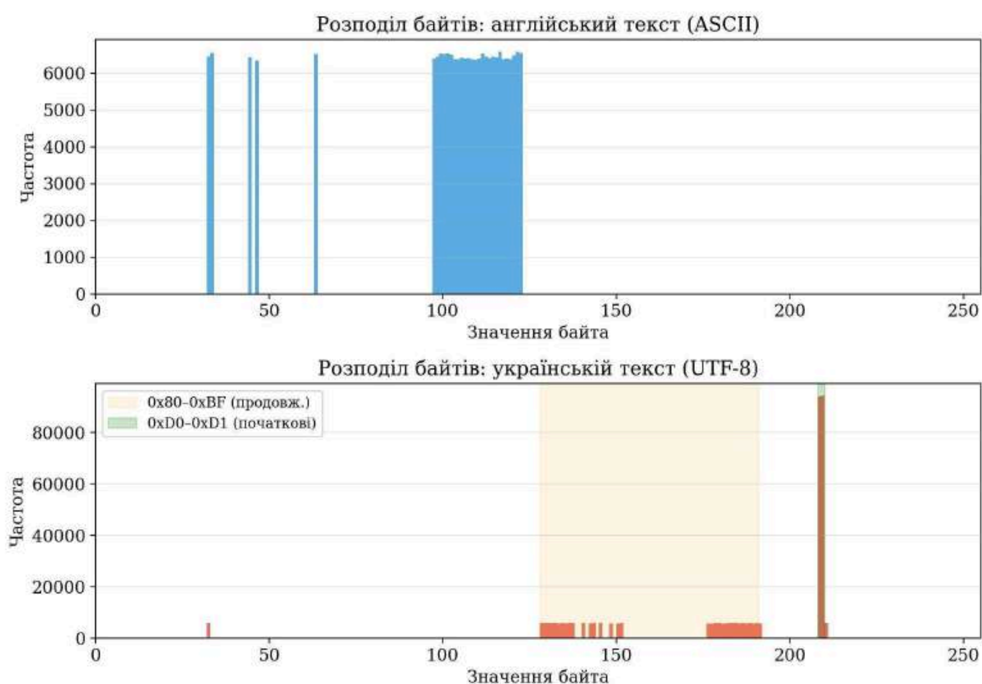

Рисунок 3.5. Розподіл значень байтів у ASCII та UTF-8 текстах

Ця нерівномірність пояснює, чому евристика поганого символу BM менш ефективна на рівні байтів для кирилиці: при побайтовому пошуку ймовірність зустріти байт, відсутній у шаблоні, є низькою (більшість байтів потрапляють у вузький діапазон 0x80–0xD1), що обмежує можливі зсуви. Це є одним із ключових висновків дослідження впливу UTF-8 на продуктивність алгоритмів.

3.5. Ефективність множинного пошуку

П'ята серія експериментів порівнює алгоритм Ахо–Корасік із послідовним застосуванням KMP для кожного шаблону. Пошук виконувався на тексті 1 МБ із множинами з 10, 100 та 1000 шаблонів довжиною 10 символів. Результати наведено у табл. 3.6.

Таблиця 3.6. Порівняння часу множинного пошуку (мс), n = 1 МБ, m = 10

К-сть шаблонів	KMP (послідовно)	Ахо–Корасік	Прискорення
10	403.47	77.27	5.2×
100	4 034.83	110.89	36.4×
1 000	40 703.75	137.63	295.7×

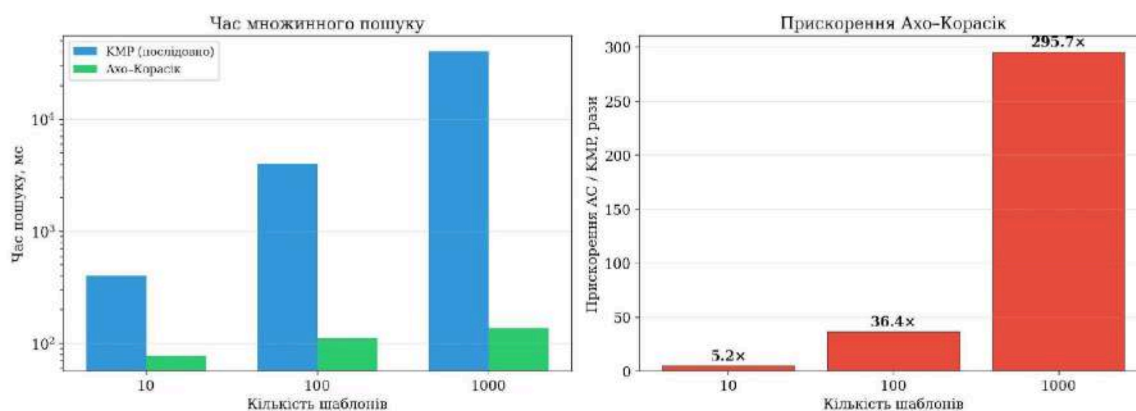

Рисунок 3.6. Час множинного пошуку та прискорення Ахо–Корасік

Результати демонструють кардинальну перевагу алгоритму Ахо–Корасік при множинному пошуку. При 10 шаблонах AC у 5.2 рази швидший за послідовний KMP, при 100 шаблонах – у 36.4 рази, а при 1000 шаблонах – у 295.7 рази.

Послідовний KMP масштабується лінійно відносно кількості шаблонів: при збільшенні k у 10 разів час зростає також приблизно у 10 разів (403 мс → 4035 мс → 40704 мс). Це очікувано, оскільки для кожного шаблону виконується окремий прохід тексту за O(n).

Час роботи Ахо–Корасік зростає значно повільніше: 77.27 мс → 110.89 мс → 137.63 мс при збільшенні k від 10 до 1000. Це пояснюється тим, що AC виконує лише один прохід тексту незалежно від кількості шаблонів; збільшення часу пов'язане виключно з побудовою більшого автомата та більшою кількістю переходів за суфіксними посиланнями.

При 1000 шаблонах різниця є особливо разючою: 137.63 мс (AC) проти 40 703.75 мс (KMP), тобто AC завершує пошук за час, менший за час одного проходу KMP для одного шаблону. Це підтверджує теоретичну оцінку складності $O(n + L + z)$ для AC проти $O(k \cdot n)$ для послідовного KMP.

3.6. Порівняння споживання пам'яті

Шоста серія експериментів досліджує піковий обсяг додаткової пам'яті, витраченої кожним алгоритмом. Результати наведено у табл. 3.7.

Таблиця 3.7. Пікове споживання додаткової пам'яті (КБ)

m	Наївний	KMP	BM	RK	AC (100 шабл.)
10	0.23	0.26	0.46	0.49	237.00
50	0.23	0.57	1.64	0.49	1 398.46
100	0.23	0.96	2.42	0.49	2 853.43

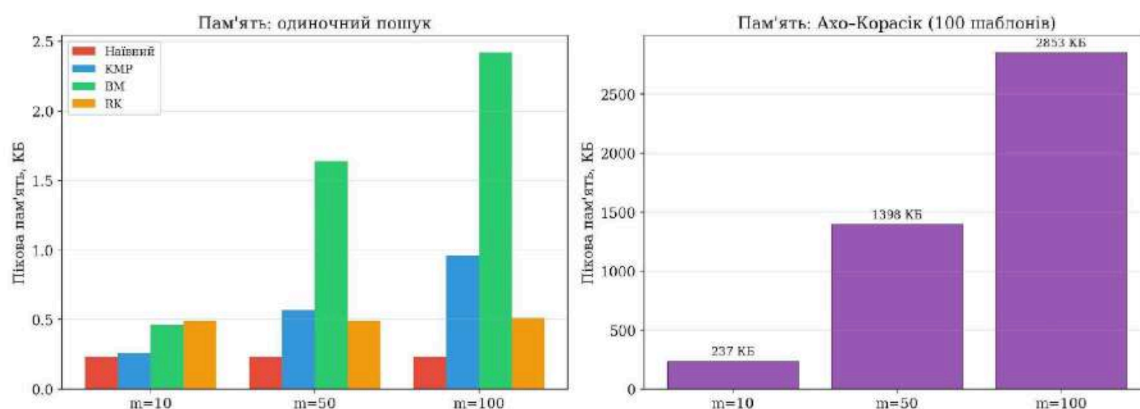

Рисунок 3.7. Порівняння споживання пам'яті алгоритмами

Наївний алгоритм споживає мінімальну фіксовану кількість пам'яті (0.23 КБ) незалежно від довжини шаблону, що підтверджує теоретичну оцінку O(1). Алгоритм Рабіна–Карпа також демонструє константне споживання (0.49 КБ) – дещо більше через зберігання змінних хеш-функції.

КМР споживає пам'ять пропорційно довжині шаблону: від 0.26 КБ (m = 10) до 0.96 КБ (m = 100), що відповідає теоретичній оцінці O(m) для масиву функції невдач. BM споживає дещо більше: від 0.46 КБ до 2.42 КБ, що відповідає O(m + |Σ_eff|) для таблиць обох евристик.

Алгоритм Ахо–Корасік споживає на порядки більше пам'яті: від 237 КБ (m = 10, 100 шаблонів) до 2 853 КБ (m = 100, 100 шаблонів). Це є очікуваним компромісом: AC обмінює пам'ять на швидкість, зберігаючи повний автомат із усіма переходами та суфіксними посиланнями. При m = 100 та 100 шаблонах автомат споживає у 2 972 рази більше пам'яті, ніж KMP, але забезпечує прискорення у 36.4 рази (з табл. 3.6).

3.7. Рекомендації щодо вибору алгоритму

На підставі проведених експериментів сформульовано практичні рекомендації щодо вибору алгоритму пошуку підрядків залежно від характеристик задачі. Рекомендації зведено у табл. 3.8.

Таблиця 3.8. Рекомендації щодо вибору алгоритму

Характеристика задачі	Алгоритм	Обґрунтування
Великий алфавіт, довгий шаблон	BM	Сублінійна складність O(n/m), прискорення до 6.4× проти KMP
Малий алфавіт (ДНК, бінарні дані)	KMP	Стабільна продуктивність, перевага BM мінімальна при $\|\Sigma\| \leq 4$
Потрібна гарантована лінійність	KMP	$O(n + m)$ у найгіршому випадку, незалежно від даних
Множинний пошук (k шаблонів)	AC	Один прохід тексту, прискорення ≈ k× проти послідовного пошуку
Мінімум пам'яті	Наївний	O(1) додаткової пам'яті, прийнятний для коротких текстів
UTF-8, кирилиця	BM (str)	Посимвольний режим швидший за побайтовий на 17–40%

Окремо слід зазначити, що алгоритм Рабіна–Карпа не рекомендується для одиночного пошуку підрядків у інтерпретованих мовах: його overhead на обчислення хеш-функції перевищує виграш від уникнення повних порівнянь. Перевага RK проявляється при пошуку множини шаблонів фіксованої довжини або в низькорівневих реалізаціях на C/C++, де арифметичні операції не мають overhead інтерпретатора.

Проведений аналіз результатів шести серій експериментів дозволяє сформулювати наступні висновки.

1. Усі досліджені алгоритми масштабуються лінійно при зростанні обсягу тексту, що підтверджує теоретичні оцінки складності для середнього випадку.

2. Алгоритм Боєра–Мура є найшвидшим для одиночного пошуку при великому алфавіті ($|\Sigma| > 10$), причому його перевага зростає зі збільшенням довжини шаблону: від 1.16× при $m = 5$ до 6.42× при $m = 100$ порівняно з KMP.

3. При малому алфавіті ($|\Sigma| \leq 4$) перевага BM зменшується до 1.5×, що робить KMP конкурентоспроможною альтернативою з гарантованою лінійною складністю.

4. Побайтовий пошук у кириличному UTF-8 тексті повільніший за посимвольний на 21–61% через збільшення обсягу даних удвічі, хоча це частково компенсується простішими операціями порівняння байтів.

5. Розподіл байтів кирилиці у UTF-8 є вкрай нерівномірним (50% байтів – це 0xD0 та 0xD1), що знижує ефективність евристики поганого символу алгоритму BM при побайтовому пошуку.

6. Алгоритм Ахо–Корасік забезпечує прискорення у 295.7 рази при 1000 шаблонах порівняно з послідовним KMP, що підтверджує його незамінність для задач множинного пошуку.

7. Споживання пам'яті є значущим фактором лише для алгоритму Ахо–Корасік (до 2.8 МБ для 100 шаблонів довжиною 100), тоді як інші алгоритми потребують менше 3 КБ.

ВИСНОВКИ

У кваліфікаційній роботі здійснено порівняльний аналіз алгоритмів пошуку підрядків у великих текстових даних. Дослідження охопило п'ять алгоритмів – наївний, Кнута–Морріса–Пратта (KMP), Боєра–Мура (BM), Рабіна–Карпа (RK) та Ахо–Корасік (AC) – і включало як теоретичний аналіз обчислювальної складності, так і масштабне експериментальне дослідження з використанням програмного стенду, розробленого мовою Python. За результатами роботи сформульовано наступні висновки.

1. Здійснено систематичний теоретичний огляд п'яти алгоритмів пошуку підрядків. Для кожного алгоритму описано принцип роботи, математичне обґрунтування та оцінку обчислювальної складності у нотації «великого O» для найкращого, середнього та найгіршого випадків. Показано, що алгоритми KMP та AC гарантують лінійну часову складність $O(n + m)$ та $O(n + L + z)$ відповідно у найгіршому випадку, алгоритм BM здатний досягати сублінійної складності $O(n/m)$ у кращому випадку, а наївний алгоритм та RK мають квадратичну складність $O(n \cdot m)$ у найгіршому випадку.

2. Розроблено програмний стенд для бенчмаркінгу алгоритмів мовою Python 3.12 із вимірюванням часу виконання (time.perf_counter, 10 повторів, медіана) та споживання пам'яті (tracemalloc). Підготовлено чотири набори тестових даних з різними характеристиками алфавіту та кодування: англійський текст ($|\Sigma| \approx 31$, ASCII), українській текст ($|\Sigma| \approx 34$, UTF-8), ДНК-послідовності ($|\Sigma| = 4$) та бінарні рядки ($|\Sigma| = 2$). Проведено шість серій експериментів із систематичним варіюванням параметрів.

3. Експериментально підтверджено, що для тексту з великим алфавітом ($|\Sigma| > 10$) алгоритм Боєра–Мура є найшвидшим для одиночного пошуку. На тексті 10 МБ BM виконує пошук за 110.31 мс, що у 3.69 рази швидше за KMP (406.93 мс) та у 9.37 рази швидше за наївний алгоритм (1033.50 мс). Перевага BM зростає зі збільшенням довжини шаблону: від 1.16×

при m = 5 до 6.42× при m = 100 порівняно з KMP, що експериментально підтверджує теоретичну сублінійну складність O(n/m).

4. Встановлено, що ефективність алгоритму Боєра–Мура суттєво залежить від розміру алфавіту. При бінарному алфавіті (|Σ| = 2) BM лише у 1.56 рази швидший за KMP, тоді як при |Σ| = 31 перевага зростає до 3.67 рази. Це пояснюється деградацією евристики поганого символу при малому алфавіті: ймовірність зустріти символ, відсутній у шаблоні, зменшується пропорційно до 1/|Σ|. При малому алфавіті (|Σ| ≤ 4) KMP є конкурентоспроможною альтернативою з гарантованою лінійною складністю.

5. Вперше систематично досліджено вплив багатобайтового кодування UTF-8 на продуктивність алгоритмів пошуку підрядків. Встановлено, що побайтовий пошук у кириличному тексті (1 000 000 символів, 1 970 533 байти UTF-8) повільніший за посимвольний на 21–61% залежно від алгоритму. Сповільнення менше за очікуване двократне (1.97× за обсягом даних), що пояснюється вищою швидкістю елементарних операцій порівняння байтів (цілих чисел) у порівнянні з порівнянням символів Unicode (рядків довжини 1) у інтерпретаторі Python.

6. Виявлено, що навіть при посимвольному пошуку (str) кириличний текст обробляється на 15–45% повільніше за ASCII-текст при однаковій кількості символів. Це пояснюється особливостями внутрішнього представлення рядків у CPython: для ASCII використовується компактне представлення (1 байт на символ), для кирилиці – UCS-2 (2 байти на символ), що збільшує обсяг даних у пам'яті та погіршує використання кеш-пам'яті процесора.

7. Показано, що розподіл байтів кирилиці у UTF-8 є вкрай нерівномірним: байти 0xD0 та 0xD1 (початкові байти кириличних символів) складають приблизно 50% усіх байтів тексту, а продовжуючі байти концентруються у діапазоні 0x80–0xBF. Ця нерівномірність знижує ефективність евристики поганого символу алгоритму BM при побайтовому

пошуку, оскільки ймовірність зустріти байт, відсутній у шаблоні, є значно нижчою, ніж для рівномірно розподіленого ASCII-тексту.

8. Показано, що алгоритм Ахо–Корасік є безальтернативним рішенням для задачі множинного пошуку. При 1000 шаблонах AC забезпечує прискорення у 295.7 рази порівняно з послідовним застосуванням KMP (137.63 мс проти 40 703.75 мс). При цьому час роботи AC зростає лише з 77.27 мс (10 шаблонів) до 137.63 мс (1000 шаблонів), що підтверджує теоретичну незалежність складності пошуку від кількості шаблонів.

9. Встановлено, що споживання пам'яті є значущим фактором лише для алгоритму Ахо–Корасік: при 100 шаблонах довжиною 100 символів автомат споживає 2 853 КБ додаткової пам'яті, тоді як інші алгоритми потребують менше 3 КБ. Це є компромісом між швидкістю та пам'яттю, який слід враховувати при проєктуванні систем з обмеженими ресурсами.

10. Сформульовано практичні рекомендації щодо вибору алгоритму залежно від характеристик задачі: BM – для одиночного пошуку при великому алфавіті та довгих шаблонах; KMP – для текстів з малим алфавітом або при потребі гарантованої лінійної складності; AC – для множинного пошуку; наївний алгоритм – лише для коротких текстів. Для кириличних текстів у кодуванні UTF-8 рекомендовано використовувати посимвольний режим пошуку (str), який на 17–40% швидший за побайтовий.

Перспективними напрямками подальших досліджень є: адаптація алгоритмів для паралельних обчислень на багатоядерних процесорах та GPU; розширення дослідження UTF-8 на інші мови з трьох- та чотирибайтовим кодуванням (китайська, японська, арабська, емодзі); дослідження потокового (streaming) пошуку при обробці даних, що не вміщуються в оперативну пам'ять; порівняння з вбудованими реалізаціями пошуку на C/C++ для оцінки впливу мови програмування на результати бенчмаркінгу.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. Introduction to Algorithms. 4th ed. Cambridge : MIT Press, 2022. 1312 p.

2. Gusfield D. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge : Cambridge University Press, 1997. 534 p.

3. Charras C., Lecroq T. Handbook of Exact String Matching Algorithms. London : King's College Publications, 2004. 238 p.

4. Knuth D. E., Morris J. H., Pratt V. R. Fast pattern matching in strings. SIAM Journal on Computing. 1977. Vol. 6, No. 2. P. 323–350.

5. Boyer R. S., Moore J. S. A fast string searching algorithm. Communications of the ACM. 1977. Vol. 20, No. 10. P. 762–772.

6. Karp R. M., Rabin M. O. Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development. 1987. Vol. 31, No. 2. P. 249–260.

7. Aho A. V., Corasick M. J. Efficient string matching: An aid to bibliographic search. Communications of the ACM. 1975. Vol. 18, No. 6. P. 333–340.

8. Faro S., Lecroq T. The Exact Online String Matching Problem: A Review of the Most Recent Results. ACM Computing Surveys. 2013. Vol. 45, No. 2. Article 13.

9. Yergeau F. UTF-8, a transformation format of ISO 10646. RFC 3629. Internet Engineering Task Force, 2003. URL: https://www.rfc-editor.org/rfc/rfc3629 (дата звернення: 12.05.2026).

10. The Unicode Consortium. The Unicode Standard, Version 15.0. Mountain View : Unicode, Inc., 2022. URL: https://www.unicode.org/versions/Unicode15.0.0/ (дата звернення: 10.05.2026).

ДОДАТОК А

Повний вихідний код програмного стенду

Наївний алгоритм (naive.py)

```
"""Наївний (brute-force) алгоритм пошуку підрядка."""


def naive_search(text, pattern):
    """
    Пошук усіх входжень pattern у text методом послідовної перевірки.

    Часова складність: O(n·m) — найгірший, O(n) — середній.
    Просторова складність: O(1) додаткової пам'яті.
    """
    n, m = len(text), len(pattern)
    if m == 0 or m > n:
        return []

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

Алгоритм KMP (kmp.py)

```
"""Алгоритм Кнута-Морріса-Пратта (KMP)."""


def _build_failure(pattern):
    """
    Побудова масиву функції невдач (prefix function) за O(m).
    pi[j] = довжина найдовшого власного префікса P[0..j],
    який одночасно є суфіксом.
    """
    m = len(pattern)
    pi = [0] * m
    k = 0
    for j in range(1, m):
        while k > 0 and pattern[k] != pattern[j]:
            k = pi[k - 1]
        if pattern[k] == pattern[j]:
            k += 1
        pi[j] = k
    return pi
```

```python
def kmp_search(text, pattern):
    """
    Пошук усіх входжень pattern у text алгоритмом KMP.

    Часова складність: O(n + m).
    Просторова складність: O(m).
    """
    n, m = len(text), len(pattern)
    if m == 0 or m > n:
        return []

    pi = _build_failure(pattern)
    result = []
    k = 0  # кількість символів шаблону, що збіглися

    for i in range(n):
        while k > 0 and pattern[k] != text[i]:
            k = pi[k - 1]
        if pattern[k] == text[i]:
            k += 1
        if k == m:
            result.append(i - m + 1)
            k = pi[k - 1]

    return result
```

Алгоритм Боєра–Мура (boyer_moore.py)

```python
"""Алгоритм Боєра-Мура (BM) з обома евристиками."""


def _bad_character_table(pattern):
    """
    Побудова таблиці поганого символу.
    Використовуємо dict замість масиву для підтримки Unicode.
    """
    table = {}
    m = len(pattern)
    for i in range(m):
        table[pattern[i]] = i
    return table


def _good_suffix_table(pattern):
    """
    Побудова таблиці доброго суфікса за O(m).
    """
    m = len(pattern)
```

```python
    # Крок 1: обчислення масиву suffix
    # suffix[i] = довжина найдовшого суфікса P[i..m-1],
    # що збігається з суфіксом всього шаблону
    suffix = [0] * m
    suffix[m - 1] = m
    g = m - 1
    f = 0
    for i in range(m - 2, -1, -1):
        if i > g and suffix[i + m - 1 - f] < i - g:
            suffix[i] = suffix[i + m - 1 - f]
        else:
            if i < g:
                g = i
            f = i
            while g >= 0 and pattern[g] == pattern[g + m - 1 - f]:
                g -= 1
            suffix[i] = f - g

    # Крок 2: побудова таблиці зсувів
    good_suffix = [m] * m

    # Випадок 2: префікс шаблону збігається з суфіксом зіставленої частини
    j = 0
    for i in range(m - 1, -1, -1):
        if suffix[i] == i + 1:
            while j < m - 1 - i:
                if good_suffix[j] == m:
                    good_suffix[j] = m - 1 - i
                j += 1

    # Випадок 1: повне входження суфікса всередині шаблону
    for i in range(m - 1):
        good_suffix[m - 1 - suffix[i]] = m - 1 - i

    return good_suffix


def boyer_moore_search(text, pattern):
    """
    Пошук усіх входжень pattern у text алгоритмом Боєра-Мура.

    Часова складність: O(n) найгірший (з обома евристиками), O(n/m) кращий.
    Просторова складність: O(m + |Σ|).
    """
    n, m = len(text), len(pattern)
    if m == 0 or m > n:
        return []

    bad_char = _bad_character_table(pattern)
    good_suffix = _good_suffix_table(pattern)
```

```python
    result = []
    i = 0  # позиція лівого краю шаблону в тексті

    while i <= n - m:
        j = m - 1  # порівнюємо справа наліво

        while j >= 0 and pattern[j] == text[i + j]:
            j -= 1

        if j < 0:
            # Повний збіг
            result.append(i)
            i += good_suffix[0]
        else:
            # Невідповідність на позиції j
            bc_shift = j - bad_char.get(text[i + j], -1)
            gs_shift = good_suffix[j]
            i += max(bc_shift, gs_shift)

    return result
```

Алгоритм Рабіна–Карпа (rabin_karp.py)

```python
"""Алгоритм Рабіна-Карпа (RK) з поліноміальним хешем."""


def rabin_karp_search(text, pattern, d=256, q=1_000_000_007):
    """
    Пошук усіх входжень pattern у text алгоритмом Рабіна-Карпа.

    Параметри:
        d — основа хеш-функції (розмір алфавіту)
        q — модуль (велике просте число)

    Часова складність: O(n + m) середній, O(n·m) найгірший.
    Просторова складність: O(1).
    """
    n, m = len(text), len(pattern)
    if m == 0 or m > n:
        return []

    result = []

    # h = d^(m-1) mod q — для видалення старшого символу
    h = pow(d, m - 1, q)

    def char_val(c):
        return c if isinstance(c, int) else ord(c)

    # Обчислення хешу шаблону та першого вікна тексту
    p_hash = 0
    t_hash = 0
```

```python
    for i in range(m):
        p_hash = (d * p_hash + char_val(pattern[i])) % q
        t_hash = (d * t_hash + char_val(text[i])) % q

    # Ковзне вікно
    for i in range(n - m + 1):
        # Якщо хеші збігаються — верифікація
        if p_hash == t_hash:
            if text[i : i + m] == pattern:
                result.append(i)

        # Обчислення хешу наступного вікна
        if i < n - m:
            t_hash = (d * (t_hash - char_val(text[i]) * h) + char_val(text[i + m])) % q
            if t_hash < 0:
                t_hash += q

    return result
```

Алгоритм Ахо–Корасік (aho_corasick.py)

```python
"""Алгоритм Ахо-Корасік (AC) для множинного пошуку підрядків."""

from collections import deque


class AhoCorasick:
    """
    Автомат Ахо-Корасік.

    Побудова автомата: O(L), де L — сумарна довжина шаблонів.
    Пошук: O(n + z), де z — кількість входжень.
    Пам'ять: O(L) при реалізації переходів через dict.
    """

    def __init__(self):
        # goto[state] = {char: next_state}
        self.goto = [{}]
        # fail[state] = suffix link
        self.fail = [0]
        # output[state] = [(pattern_index, pattern_length), ...]
        self.output = [[]]
        self._built = False

    def add_pattern(self, pattern, index):
        """Додати шаблон до автомата (до виклику build)."""
        state = 0
        for ch in pattern:
            if ch not in self.goto[state]:
                self.goto.append({})
                self.fail.append(0)
                self.output.append([])
                self.goto[state][ch] = len(self.goto) - 1
            state = self.goto[state][ch]
        self.output[state].append((index, len(pattern)))
```

```python
    def build(self):
        """
        Побудова суфіксних посилань (BFS) та словникових суфіксних посилань.
        """
        queue = deque()

        # Ініціалізація: діти кореня мають fail = 0
        for ch, s in self.goto[0].items():
            queue.append(s)
            # fail[s] = 0 вже встановлено

        while queue:
            r = queue.popleft()
            for ch, s in self.goto[r].items():
                queue.append(s)

                # Знаходимо суфіксне посилання для s
                state = self.fail[r]
                while state != 0 and ch not in self.goto[state]:
                    state = self.fail[state]

                self.fail[s] = self.goto[state].get(ch, 0)
                if self.fail[s] == s:
                    self.fail[s] = 0

                # Об'єднуємо output із суфіксним посиланням
                self.output[s] = self.output[s] + self.output[self.fail[s]]

        self._built = True

    def search(self, text):
        """
        Пошук усіх входжень шаблонів у тексті.

        Повертає список (position, pattern_index), де position –
        позиція початку входження.
        """
        if not self._built:
            raise RuntimeError("Викличте build() перед search()")

        state = 0
        results = []

        for i, ch in enumerate(text):
            while state != 0 and ch not in self.goto[state]:
                state = self.fail[state]
            state = self.goto[state].get(ch, 0)

            for pattern_index, pattern_length in self.output[state]:
                results.append((i - pattern_length + 1, pattern_index))

        return results
```

```
def aho_corasick_search(text, patterns):
    """
    Обгортка для зручного виклику.

    Параметри:
        text – текст для пошуку
        patterns – список шаблонів

    Повертає: список (position, pattern_index)
    """
    ac = AhoCorasick()
    for i, p in enumerate(patterns):
        ac.add_pattern(p, i)
    ac.build()
    return ac.search(text)
```

Генерація тестових даних (data_gen.py)

```
"""Генерація та завантаження тестових даних."""

import random
import string


def generate_random_text(size, alphabet="abcdefghijklmnopqrstuvwxyz "):
    """Генерація випадкового тексту заданого розміру."""
    return "".join(random.choices(alphabet, k=size))


def generate_dna(size):
    """Генерація випадкової ДНК-послідовності (|Σ| = 4)."""
    return generate_random_text(size, "ACGT")


def generate_binary(size):
    """Генерація бінарного рядка (|Σ| = 2)."""
    return generate_random_text(size, "ab")


def generate_ukrainian(size):
    """
    Генерація випадкового кирилічного тексту (|Σ| ≈ 70).
    Імітує природний розподіл з пробілами.
    """
    alphabet = "абвгґдеєжзиіїйклмнопрстуфхцчшщьюя "
    return generate_random_text(size, alphabet)
```

```python
def extract_pattern_from_text(text, length, offset=None):
    """
    Витягти реальний підрядок із тексту як шаблон.
    Гарантує що шаблон точно є в тексті (хоча б 1 входження).
    """
    n = len(text)
    if length > n:
        raise ValueError(f"Довжина шаблону ({length}) > довжина тексту ({n})")

    if offset is None:
        # Випадкова позиція (але не з самого початку/кінця)
        max_offset = n - length
        offset = random.randint(max_offset // 4, 3 * max_offset // 4)

    return text[offset : offset + length]


def extract_patterns_from_text(text, count, length):
    """Витягти count різних шаблонів довжини length із тексту."""
    n = len(text)
    if length > n:
        raise ValueError(f"Довжина шаблону ({length}) > довжина тексту ({n})")

    patterns = set()
    attempts = 0
    max_attempts = count * 20

    while len(patterns) < count and attempts < max_attempts:
        offset = random.randint(0, n - length)
        p = text[offset : offset + length]
        patterns.add(p)
        attempts += 1

    return list(patterns)


def generate_worst_case_naive(n, m):
    """
    Генерує worst-case для наївного алгоритму:
    текст = 'aaa...aaa', шаблон = 'aaa...ab'
    """
    text = "a" * n
    pattern = "a" * (m - 1) + "b"
    return text, pattern


def text_to_bytes_utf8(text):
    """Конвертація str -> bytes (UTF-8) для побайтового пошуку."""
    return text.encode("utf-8")
```

```
# ---- Розміри тестових даних ----
TEXT_SIZES = {
    "100KB": 100_000,
    "1MB": 1_000_000,
    "10MB": 10_000_000,
}

PATTERN_LENGTHS = [5, 10, 20, 50, 100]

MULTI_PATTERN_COUNTS = [10, 100, 1000]
```

Бенчмарк-система (benchmark.py)

```
"""
Бенчмарк-система для порівняння алгоритмів пошуку підрядків.

Вимірює час виконання (time.perf_counter) та пікову пам'ять (tracemalloc).
Кожен експеримент повторюється REPEATS разів, беруться медіана та std.
"""

import time
import tracemalloc
import statistics
import csv
import json
import os
from datetime import datetime

from algorithms import (
    naive_search,
    kmp_search,
    boyer_moore_search,
    rabin_karp_search,
    aho_corasick_search,
)
from data_gen import (
    generate_random_text,
    generate_dna,
    generate_binary,
    generate_ukrainian,
    extract_pattern_from_text,
    extract_patterns_from_text,
    text_to_bytes_utf8,
    TEXT_SIZES,
    PATTERN_LENGTHS,
    MULTI_PATTERN_COUNTS,
)

REPEATS = 10
RESULTS_DIR = "results"
```

```python
SINGLE_ALGORITHMS = {
    "naive": naive_search,
    "kmp": kmp_search,
    "boyer_moore": boyer_moore_search,
    "rabin_karp": rabin_karp_search,
}


def measure_time(func, *args, repeats=REPEATS):
    """Вимірювання часу: repeats запусків, повертає медіану та std (мс)."""
    times = []
    result = None
    for _ in range(repeats):
        start = time.perf_counter()
        result = func(*args)
        elapsed = (time.perf_counter() - start) * 1000  # мс
        times.append(elapsed)
    return {
        "median_ms": round(statistics.median(times), 3),
        "std_ms": round(statistics.stdev(times), 3) if len(times) > 1 else 0,
        "min_ms": round(min(times), 3),
        "max_ms": round(max(times), 3),
        "matches": len(result) if isinstance(result, list) else 0,
    }


def measure_memory(func, *args):
    """Вимірювання пікової додаткової пам'яті (КБ) через tracemalloc."""
    tracemalloc.start()
    _ = func(*args)
    _, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    return round(peak / 1024, 2)  # КБ


def save_results(results, filename):
    """Зберегти результати у JSON та CSV."""
    os.makedirs(RESULTS_DIR, exist_ok=True)

    # JSON (повні дані)
    json_path = os.path.join(RESULTS_DIR, f"{filename}.json")
    with open(json_path, "w", encoding="utf-8") as f:
        json.dump(results, f, indent=2, ensure_ascii=False)

    # CSV (для графіків)
    if results and isinstance(results, list) and len(results) > 0:
        csv_path = os.path.join(RESULTS_DIR, f"{filename}.csv")
        keys = results[0].keys()
        with open(csv_path, "w", newline="", encoding="utf-8") as f:
            writer = csv.DictWriter(f, fieldnames=keys)
            writer.writeheader()
            writer.writerows(results)

    print(f"  → Збережено: {json_path}")
```

```python
# ==================================================================
# ЕКСПЕРИМЕНТИ
# ==================================================================


def experiment_1_text_size():
    """
    Експеримент 1: Залежність часу від обсягу тексту.
    Фіксований шаблон 20 символів, англійський текст.
    """
    print("\n" + "=" * 60)
    print("ЕКСПЕРИМЕНТ 1: Залежність часу від обсягу тексту")
    print("=" * 60)

    results = []
    for size_name, size in TEXT_SIZES.items():
        print(f"\n  Обсяг: {size_name} ({size:,} символів)")
        text = generate_random_text(size, "abcdefghijklmnopqrstuvwxyz .,!?")
        pattern = extract_pattern_from_text(text, 20)

        for alg_name, alg_func in SINGLE_ALGORITHMS.items():
            timing = measure_time(alg_func, text, pattern)
            row = {
                "text_size": size_name,
                "text_len": size,
                "pattern_len": 20,
                "algorithm": alg_name,
                **timing,
            }
            results.append(row)
            print(f"    {alg_name:15s}: {timing['median_ms']:8.2f} мс "
                  f"(±{timing['std_ms']:.2f}), збігів: {timing['matches']}")

    save_results(results, "exp1_text_size")
    return results


def experiment_2_pattern_length():
    """
    Експеримент 2: Вплив довжини шаблону.
    Фіксований текст 1МБ, шаблони різної довжини.
    """
    print("\n" + "=" * 60)
    print("ЕКСПЕРИМЕНТ 2: Вплив довжини шаблону")
    print("=" * 60)

    text = generate_random_text(1_000_000, "abcdefghijklmnopqrstuvwxyz .,!?")
    results = []

    for m in PATTERN_LENGTHS:
        print(f"\n  Довжина шаблону: {m}")
        pattern = extract_pattern_from_text(text, m)

        for alg_name, alg_func in SINGLE_ALGORITHMS.items():
            timing = measure_time(alg_func, text, pattern)
```

```python
            row = {
                "pattern_len": m,
                "algorithm": alg_name,
                **timing,
            }
            results.append(row)
            print(f"    {alg_name:15s}: {timing['median_ms']:8.2f} мс")

    save_results(results, "exp2_pattern_length")
    return results


def experiment_3_alphabet_size():
    """
    Експеримент 3: Вплив розміру алфавіту.
    Фіксований обсяг 1МБ, шаблон 20, різні алфавіти.
    """
    print("\n" + "=" * 60)
    print("ЕКСПЕРИМЕНТ 3: Вплив розміру алфавіту")
    print("=" * 60)

    datasets = {
        "binary (|Σ|=2)": generate_binary(1_000_000),
        "DNA (|Σ|=4)": generate_dna(1_000_000),
        "English (|Σ|≈30)": generate_random_text(1_000_000, "abcdefghijklmnopqrstuvwxyz .,!?"),
        "Ukrainian (|Σ|≈35)": generate_ukrainian(1_000_000),
    }
    results = []

    for ds_name, text in datasets.items():
        print(f"\n  Датасет: {ds_name}")
        sigma = len(set(text))
        pattern = extract_pattern_from_text(text, 20)

        for alg_name, alg_func in SINGLE_ALGORITHMS.items():
            timing = measure_time(alg_func, text, pattern)
            row = {
                "dataset": ds_name,
                "sigma": sigma,
                "algorithm": alg_name,
                **timing,
            }
            results.append(row)
            print(f"    {alg_name:15s}: {timing['median_ms']:8.2f} мс "
                  f"(|Σ|={sigma})")

    save_results(results, "exp3_alphabet")
    return results


def experiment_4_utf8():
    """
    Експеримент 4: Вплив кодування UTF-8.
    Порівняння побайтового та посимвольного пошуку
    на українському тексті.
    """
    print("\n" + "=" * 60)
    print("ЕКСПЕРИМЕНТ 4: Вплив кодування UTF-8")
    print("=" * 60)
```

```python
    SIZE = 1_000_000  # символів

    # Генеруємо тексти з однаковою кількістю символів
    text_en = generate_random_text(SIZE, "abcdefghijklmnopqrstuvwxyz .,!?")
    text_uk = generate_ukrainian(SIZE)

    results = []

    # --- 4a: Посимвольний пошук (str) ---
    print("\n  4a: Посимвольний пошук (str)")
    for lang, text in [("English", text_en), ("Ukrainian", text_uk)]:
        pattern = extract_pattern_from_text(text, 20)
        bytes_total = len(text.encode("utf-8"))
        print(f"\n    {lang}: {SIZE:,} символів, {bytes_total:,} байтів UTF-8")

        for alg_name, alg_func in SINGLE_ALGORITHMS.items():
            timing = measure_time(alg_func, text, pattern)
            row = {
                "mode": "chars",
                "language": lang,
                "char_len": SIZE,
                "byte_len": bytes_total,
                "algorithm": alg_name,
                **timing,
            }
            results.append(row)
            print(f"      {alg_name:15s}: {timing['median_ms']:8.2f} мс")

    # --- 4b: Побайтовий пошук (bytes) ---
    print("\n  4b: Побайтовий пошук (bytes)")
    for lang, text in [("English", text_en), ("Ukrainian", text_uk)]:
        text_b = text_to_bytes_utf8(text)
        pattern = extract_pattern_from_text(text, 20)
        pattern_b = text_to_bytes_utf8(pattern)

        print(f"\n    {lang}: {len(text_b):,} байтів")

        for alg_name, alg_func in SINGLE_ALGORITHMS.items():
            timing = measure_time(alg_func, text_b, pattern_b)
            row = {
                "mode": "bytes",
                "language": lang,
                "char_len": SIZE,
                "byte_len": len(text_b),
                "algorithm": alg_name,
                **timing,
            }
            results.append(row)
            print(f"      {alg_name:15s}: {timing['median_ms']:8.2f} мс")

    # --- 4c: Розподіл байтів ---
    print("\n  4c: Розподіл байтів")
    for lang, text in [("English", text_en), ("Ukrainian", text_uk)]:
        text_b = text_to_bytes_utf8(text)
        freq = {}
        for b in text_b:
            freq[b] = freq.get(b, 0) + 1
        unique_bytes = len(freq)
        top5 = sorted(freq.items(), key=lambda x: -x[1])[:5]
        top5_str = ", ".join(f"0x{b:02X}={c}" for b, c in top5)
        print(f"    {lang}: {unique_bytes} унікальних байтів, "
              f"топ-5: {top5_str}")
```

```
    save_results(results, "exp4_utf8")
    return results


def experiment_5_multi_pattern():
    """
    Експеримент 5: Множинний пошук (Ахо-Корасік vs послідовний КМР).
    """
    print("\n" + "=" * 60)
    print("ЕКСПЕРИМЕНТ 5: Множинний пошук (AC vs послідовний KMP)")
    print("=" * 60)

    text = generate_random_text(1_000_000, "abcdefghijklmnopqrstuvwxyz .,!?")
    results = []

    for k in MULTI_PATTERN_COUNTS:
        print(f"\n  Кількість шаблонів: {k}")
        patterns = extract_patterns_from_text(text, k, 10)

        # AC: один прохід
        timing_ac = measure_time(aho_corasick_search, text, patterns)
        print(f"    Ахо-Корасік:     {timing_ac['median_ms']:8.2f} мс, "
              f"збігів: {timing_ac['matches']}")

        # Послідовний KMP
        def sequential_kmp(text, patterns):
            all_results = []
            for i, p in enumerate(patterns):
                for pos in kmp_search(text, p):
                    all_results.append((pos, i))
            return all_results

        timing_kmp = measure_time(sequential_kmp, text, patterns)
        print(f"    KMP (послідов.): {timing_kmp['median_ms']:8.2f} мс, "
              f"збігів: {timing_kmp['matches']}")

        speedup = timing_kmp["median_ms"] / max(timing_ac["median_ms"], 0.001)
        print(f"    Прискорення AC:  {speedup:.1f}x")

        results.append({
            "pattern_count": k,
            "ac_median_ms": timing_ac["median_ms"],
            "kmp_seq_median_ms": timing_kmp["median_ms"],
            "speedup": round(speedup, 1),
            "ac_matches": timing_ac["matches"],
        })

    save_results(results, "exp5_multi_pattern")
    return results


def experiment_6_memory():
    """
    Експеримент 6: Порівняння споживання пам'яті.
    """
    print("\n" + "=" * 60)
    print("ЕКСПЕРИМЕНТ 6: Споживання пам'яті")
    print("=" * 60)
```

```
    text = generate_random_text(1_000_000, "abcdefghijklmnopqrstuvwxyz")
    results = []

    for m in [10, 50, 100]:
        print(f"\n  Довжина шаблону: {m}")
        pattern = extract_pattern_from_text(text, m)

        for alg_name, alg_func in SINGLE_ALGORITHMS.items():
            mem_kb = measure_memory(alg_func, text, pattern)
            row = {
                "pattern_len": m,
                "algorithm": alg_name,
                "peak_memory_kb": mem_kb,
            }
            results.append(row)
            print(f"    {alg_name:15s}: {mem_kb:8.2f} КБ")

        # AC з 100 шаблонами
        patterns = extract_patterns_from_text(text, 100, m)
        mem_ac = measure_memory(aho_corasick_search, text, patterns)
        results.append({
            "pattern_len": m,
            "algorithm": "aho_corasick (100)",
            "peak_memory_kb": mem_ac,
        })
        print(f"    {'AC (100 шабл.)':15s}: {mem_ac:8.2f} КБ")

    save_results(results, "exp6_memory")
    return results


# ============================================================================
# MAIN
# ============================================================================

def run_all():
    """Запуск усіх експериментів."""
    print(f"Бенчмарк запущено: {datetime.now().isoformat()}")
    print(f"Повторів на кожен вимір: {REPEATS}")

    experiment_1_text_size()
    experiment_2_pattern_length()
    experiment_3_alphabet_size()
    experiment_4_utf8()
    experiment_5_multi_pattern()
    experiment_6_memory()

    print("\n" + "=" * 60)
    print("Усі експерименти завершено!")
    print(f"Результати у директорії: {RESULTS_DIR}/")
    print("=" * 60)


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