**Вступ**

Мета курсової роботи: вивчити та проаналізувати алгоритми пошуку підрядка в рядку, і на основі них написати відповідний програмний код на мові програмування Java.

Завдання курсової роботи: вивчити завдання, пов’язані з проблемою пошуку підрядка в рядку, дати основні визначення, розглянути низку алгоритмів пошуку підрядку в рядку на практиці, написати програмний код що реалізовуватиме ці алгоритми.

Для реалізації даного програмного коду я обрав один з фреймворків Java, а саме Android. Тому всі тести моєї програми будуть проводитися на 2 видах пристроїв:

1. Телефон з операційною системою Android.
2. Вбудований емулятор AVD(Android Virtual Device) у програмі Android Studio

Результати роботи будуть порівнюватися відповідно до цих груп.

Тема курсової роботи є актуальна в сучасному світі, ті, кому доводиться часто працювати з текстовими редакторами, знають що функція знаходження потрібних слів у тексті є важливою. Вона полегшує редагування документів та пошук потрібної інформацї. Сучасні програми для обробки тексту привчили всіх користувачів до такої можливості, як пошук і заміна фрагментів тексту, відповідно якщо потрібно реалізувати подібну програму то буде логічним потурбуватися про дані функції.

Звичайно, зараз подібні функції вже присутні в багатьох мовах програмування високого рівня, тому для пошуку підрядка в рядку використовуються вбудовані функції. Але в готових підпрограмах далеко не завжди все написано кращим чином. Наприклад в стандартній функції не завжди може використовуватися найефективніший алгоритм та і область використання функції пошуку не обмежується тільки текстовим редактором. Слід відзначити використання алгоритмів пошуку при індексації сторінок пошуковим роботом, де актуальність інформації безпосередньо залежить від швидкості знаходження ключових слів у тексті html. Робота найпростішого спам-фільтра, полягає в знаходженні в тексті листа фраз таких, як «Мільйон за годину» або «Розкрутка сайту». Це все говорить про актуальність проблеми пошуку. Дана робота містить три розділи. У першому розділі буде розглянуто основні поняття що таке рядок та поняття про складність алгоритму. У другому буде розглянуто основні алгоритми пошуку підрядка в рядку, та їх модифікації. У третьому розділі буде проведено аналіз алгоритмів та їх практичне застосування. У висновку ж буде зроблено висновок про найбільш ефективний алгоритм.

**Розділ 1**

**Теоретичні відомості**

**1.1. Рядок, довжина рядка, підрядок**

Тут наведено ряд визначень, які будуть використовуватися у подальшому.

Визначення 1. Рядок (слово) – це послідовність знаків, літер та інших символів написаних чи надрукованих в одну лінію.

Визначення 2. Довжина рядка – це кількість знаків у рядку.

Визначення 3. Порожній рядок – слово яке не містить у собі жодної букви, символу.

Визначення 4. Підрядок – не порожня звязана частина рядка.

Визначення 5. Слово **X** називається префіксом слова **Y**, якщо є таке слово **Z,** що **Y = XZ.**

Визначення 6. Слово **X** називається суфіксом слова **Y**, якщо є таке слово **Z**, що **Y = ZX.** Аналогічно, слово є суфіксом самого себе.

Визначення 7. Слово **X** називається підрядком рядка **Y**, якщо знайдуться такі рядки **Z 1** і **Z 2,** що **Y = Z 1 XZ 2**. При цьому **Z 1** називають лівим, а **Z 2** - правим крилом підрядка. Підрядком може бути й саме слово. Іноді при цьому слово **X** називають входженням в слово **Y**. Серед всіх входжень слова **X** в слово **Y,** входження з найменшою довжиною свого лівого крила називають першим або головним змістом.

**1.2. Поняття про складність алгоритму**

Метою даної курсової роботи є знаходження ефективного алгоритму знаходження підрядка в рядку та слід звернути увагу і на метод оцінки даних алгоритмів.

Традиційно в програмуванні поняття складності алгоритму пов'язано з використанням ресурсів комп'ютера: наскільки багато процесорного часу вимагає програма для свого виконання, наскільки багато при цьому витрачається пам'ять машини? Облік пам'яті зазвичай ведеться за обсягом даних і не береться до уваги пам'ять, що витрачається для запису команд програми. Час розраховується у відносних одиницях так, щоб ця оцінка, по можливості, була однаковою для машин з різною тактовою частотою.   
Існують дві характеристики складності алгоритмів - тимчасова і ємнісна. Та розглянемо ми лише тимчасову, а також ми не будемо обговорювати логічну складність розробки алгоритму - скільки «людино-днів» потрібно витратити на створення програми, оскільки не представляється можливим дати об'єктивні кількісні характеристики. Тимчасову складність будемо підраховувати у виконуваних командах: кількість арифметичних операцій, кількість порівнянь, пересилань (залежно від алгоритму). Ємнісну складність визначається кількістю змінних, елементів масивів, елементів записів або просто кількістю байт. Ефективність алгоритму також буде оцінюватися за допомогою підрахунку часу виконання алгоритмом конкретно поставленої задачі, тобто з допомогою експерименту.

**Розділ 2**

**Алгоритми пошуку підрядків в рядку**

**2.1. Алгоритм послідовного (прямого) пошуку**

Цей алгоритм являється найочевиднішим. В його основі лежить просте порівняння всіх можливих підрядків **S** в рядку **X**. Нехай m i n – довжини слів **S** i **X** відповідно. Тоді можна порівняти з словом X всі підслова S, які починаються з позицій 0,1, …, m-n в слові S; у разі рівності виводиться відповідна фраза.

Це не ефективний алгоритм тому максимальна, кількість порівнянь дорівнюватиме O ((m-n) \* n), хоча більшість з них насправді зайві. Оцінимо швидкість роботи цього програмного коду. У ньому присутні два цикли (один вкладений), час роботи зовнішнього більшим ступенем залежить від n, а внутрішній у гіршому випадку робить m операцій. Таким чином, час роботи всього алгоритму є O ((n-m) \* m). Для маленьких рядків пошук пропрацює швидко, але якщо в деякому багатомегабайтному файлі буде шукатися послідовність довжиною 100 Кб, то доведеться чекати досить довго. Втім багатьом вистачає і цього.

Реалізацію алгоритму представлено в Додатку 1.

**2.2. Алгоритм Рабіна – Карпа**

Алгоритм Рабіна представляє собою модифікацію лінійного алгоритму.

У слові A, довжина якого дорівнює m, шукається зразок X довжини n. Виріжемо "віконечко" розміром n воно рухається по вхідному слову. Чи збігається слово в "віконечку" із заданим зразком? Фіксується деяка числова функція на словах довжини n, тоді завдання зводиться до порівняння чисел, що, безсумнівно, швидше. Якщо значення цієї функції на слові в "віконечку" та на зразку різні, то збігу немає. Тільки якщо значення однакові, необхідно перевіряти послідовно збіг по буквах.

Цей алгоритм виконує лінійний прохід по рядку (n кроків) і лінійний прохід по всьому тексту (m кроків), отже, загальний час роботи є O (n + m). При цьому не враховується часова складність обчислення хеш-функції, так як, суть алгоритму в тому і полягає, щоб дана функція настільки легко обчислювалася, що її робота не впливала на загальну роботу алгоритму. Тоді, час роботи алгоритму лінійно залежить від розміру рядка і тексту, відповідно програма працює швидко. Адже замість того, щоб перевіряти кожну позицію на предмет відповідності зі зразком, можна перевіряти тільки ті, які «нагадують» зразок. Отже, для того, щоб легко встановлювати явну невідповідність, буде використовуватися функція, яка повинна:

1. Легко обчислюватися.

2. Як можна краще розрізняти неспівпадаючі рядки.

3. Hash (y [i +1, i + m]) повинен легко обчислюватися за hash (y [i, i + m -1].

Під час пошуку х буде порівнюватися hash (x) з hash (y [i, i + m-1]) для i від 0 до n\*m включно. Якщо буде виявлено збіг, то перевіряється посимвольно.

Реалізація алгоритму представлена ​​у Додатку 2.

Алгоритм Рабіна і алгоритм послідовного пошуку є алгоритмами з найменшими трудовитратами, тому вони годяться для використання при вирішенні деякого класу задач. Однак ці алгоритми не є найбільш оптимальними, тому буде розглядатися наступний клас алгоритмів. Ці алгоритми з'явилися в результаті ретельного дослідження алгоритму послідовного пошуку. Дослідники хотіли знайти способи більш повно використовувати інформацію, отриману під час сканування (алгоритм прямого пошуку її просто викидає).

**2.3. Алгоритм Кнута – Морріса – Пратта (КМП)**

Метод використовує попередню обробку шуканого рядка, а саме: на її основі створюється так звана префікс-функція. Суть цієї функції в знаходженні для кожної підрядка S [1 .. i] рядка S найбільшою підрядка S [1 .. j] (j <i), присутньої одночасно і на початку, і в кінці підрядка (як префікс і як суфікс). Наприклад, для підрядка abcHelloabc такий підрядком є abc (одночасно і префіксом, і суфіксом). Сенс префікс-функції в тому, що можна відкинути свідомо невірні варіанти, тобто якщо при пошуку збігся підрядок abcHelloabc (наступний символ не збігся), то має сенс продовжувати перевірку продовжити пошук вже з четвертого символу (перші три і так співпадуть). Спочатку розглядаються деякі допоміжні твердження. Для довільного слова X розглядаються всі його початки, що одночасно є його кінцями, і вибираються з них найдовше (не враховуючи, звичайно, самого слова X). Воно позначається n (X). Така функція носить назву префікс – функції.

**Приклади.**

n (aba) = a, n (n (aba)) = n (a) = L;

n (abab) = ab, n (n (abab)) = n (ab) = L;

n (ababa) = aba, n (n (ababa)) = n (aba) = a, n (n (n (ababa))) = n (a) = L; n (abc) = L.

Справедливо наступну пропозицію:

(1) Послідовність слів n (X), n (n (X)), n (n (n (X ))),... "Обривається" (на порожньому слові L).

(2) Усі слова n (X), n (n (X)), n (n (n (X ))),..., L є началами слова X.

(3) Будь-яке слово, що одночасно є початком і кінцем слова X (крім самого X), входить в послідовність n (X), n (n (X )),...., L.

Метод КМП використовує попередню обробку шуканого рядка, а саме: на його основі створюється префікс-функція. При цьому використовується наступна ідея: якщо префікс (він же суфікс) рядка довгий i довше одного символу, то він одночасно і префікс підрядка довжиною i-1. Таким чином, перевіряється префікс попереднього підрядка, якщо ж той не підходить, то префікс його префікса, і т.д.

Наступне питання, на яке варто відповісти: чому час роботи процедури лінійний, адже в ній присутній вкладений цикл? Ну, по-перше, присвоєння префікс-функції відбувається чітко m разів, решту часу змінюється змінна k. Так як у циклі while вона зменшується (P [k] <k), але не стає менше 0, то зменшуватися вона може не частіше, ніж зростати. Змінна k зростає на 1 не більше m разів. Значить, змінна k змінюється всього не більше 2\*m разів. Виходить, що час роботи всієї процедури є O (m).

Алгоритм КМП крім знаходження самих рядків вважають, знаходить скільки символів збіглося в процесі роботи.

Реалізація алгоритму представлена ​​у Додатку 3.

**2.4. Алгоритм Бойєра – Мура та його модифікації**

**2.4.1. Алгоритм Бойєра – Мура (БМ)**

Алгоритм Бойєра-Мура, розроблений двома вченими - Бойером і Муром, вважається найбільш швидким серед алгоритмів загального призначення, призначених для пошуку підрядка в рядку.

Найпростіший варіант алгоритму Бойєра-Мура складається з наступних кроків. На першому кроці будується таблиця зміщень для шуканого зразка. Процес побудови таблиці буде описано нижче. Далі поєднується початок рядка і зразка і починається перевірка з останнього символу зразка. Якщо останній символ зразка та відповідний йому при накладенні символ рядка не збігаються, зразок зрушується щодо рядка на величину, отриману з таблиці зміщень, і знову проводиться порівняння, починаючи з останнього символу зразка. Якщо ж символи збігаються, проводиться порівняння передостаннього символу зразка і т. д. Якщо всі символи зразка збіглися з накладеними символами рядка, значить знайшовся підрядок і пошук закінчено. Якщо ж якийсь (не останній) символ зразка не співпадає з відповідним символом рядка, зсувається зразок на один символ вправо і знову починається перевірку з останнього символу. Весь алгоритм виконується до тих пір, поки або не буде знайдено входження шуканого зразка, або не буде досягнуто кінця рядку.

Величина зрушення у разі неспівпадання останнього символу обчислюється виходячи з таких міркувань: зрушення зразка повине бути мінімальним, таким, щоб не пропустити входження зразка в рядку. Якщо цей символ рядка зустрічається у зразку, зміщується зразок таким чином, щоб символ рядка збігся з найбільш правим входженням цього символу у зразку. Якщо ж зразок взагалі не містить цього символу, зсувається зразок на величину, що дорівнює його довжині, так що перший символ зразка накладається на наступний за перевірений символ рядка.

Величина зміщення для кожного символу зразка залежить тільки від порядку символів у зразку, тому зсуви зручно обчислити наперед і зберігати у вигляді одновимірного масиву, де кожному символу алфавіту відповідає зміщення відносно останнього символу зразка.

Приклад. Нехай є алфавіт з п'яти символів: a, b, c, d, e і треба знайти входження зразка "abbad" в рядку "abeccacbadbabbad". Наступні схеми ілюструють всі етапи виконання алгоритму. Таблиця зсувів буде виглядати так:

Таблиця 2.1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| a | b | c | d | e |
| 1 | 2 | 5 | 0 | 5 |

Початок пошуку.

Таблиця 2.2

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| a | b | e | c | c | a | c | b | a | d | b | a | b | b | a | b |
| a | b | b | a | d |  |  |  |  |  |  |  |  |  |  |  |

Останій символ зразка не співпадає з накладеним символом рядка. Зрушуємо зразок на 5 позицій вправо:

Таблиця 2.3

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| a | b | e | c | c | a | c | b | a | d | b | a | b | b | a | b |
|  |  |  |  |  | a | b | b | a | d |  |  |  |  |  |  |

Три символу зразка збіглися, а четвертий - ні. Зрушуємо зразок вправо на одну позицію:

Таблиця 2.4

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| a | b | e | c | c | a | c | b | a | d | b | a | b | b | a | b |
|  |  |  |  |  |  | a | b | b | a | d |  |  |  |  |  |

Останній символ знову не збігається з символом рядка. Відповідно до таблиці зміщень(Таблиця 2.1) зрушуємо зразок на 2 позиції:

Таблиця 2.5

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| a | b | e | c | c | a | c | b | a | d | b | a | b | b | a | b |
|  |  |  |  |  |  |  |  | a | b | b | a | d |  |  |  |

Ще раз зрушуємо зразок на 2 позиції:

Таблиця 2.6

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| a | b | e | c | c | a | c | b | a | d | b | a | b | b | a | b |
|  |  |  |  |  |  |  |  |  |  | a | b | b | a | d |  |

Тепер згідно таблиці вище, зрушимо зразок ще на одну позицію, і отримаємо шукане входження зразка:

Таблиця 2.7

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| a | b | e | c | c | a | c | b | a | d | b | a | b | b | a | b |
|  |  |  |  |  |  |  |  |  |  |  | a | b | b | a | d |

Отже виконавши низку зсувів зразка ми знаходим відповідне йому в рядку. Проте таблиця зсувів має певне обмеження. Для неї використовується лише 256 символів, які являються базовими для кожної електронно- обчислювальної машини (ЕОМ).

**2.4.2. Модифікації БМ**

Окрім звичайного алгоритму Бойєра – Мура (БМ) існують ще деякі його модифікації. Проаналізувавши принцип роботи деяких з них можна дійти висновку що вони використовують майже однаковий підхід, а також є досить схожими на оригінальний алгоритм. Тому у своїй роботі я розгляну тільки деякі з цих алгоритмів, які викладу нижче.

**1)Алгоритм Бойєра – Мура – Хорспула**

Цей алгоритм працює краще за Бойєра – Мура на випадкових тестах – для нього оцінка в середньому краща. Цей алгоритм використовує евристику стоп-символа, при цьому за стоп-символ береться символ вхідного рядка, який відповідає останьому символу шаблона, незалежно від того, де сталось неспівпадання. Оскільки реальні пошукові образці рідко мають рівномірне розташування, то алгоритм Бойєра – Мура – Хорспула може дати як виграш, так і програш в порівнянні з алгоритмом Бойєра – Мура.

Основна суть алгоритму дуже проста. Спочатку будується таблиця зміщень для кожного символу. Потім вихідний рядок і шаблон поєднуються з початку, порівняння ведеться за останнім символу. Якщо останні символи збігаються, то порівняння йде по передостаннього символу і так далі. Якщо ж символи не співпали, то шаблон зміщується вправо, на число позицій взяте з таблиці зсувів для символу з початкового рядка, і тоді знову порівнюються останні символи початкового рядка і шаблону. І так далі, поки не шаблон повністю не співпаде з підрядком початкового рядка, або не буде досягнуто кінець рядка.

Таблиця зсувів будується за принципом «пропускати стільки символів, скільки можливо, але не більше цього». Наприклад, якщо на якомусь етапі алгоритму останні символи не співпали, і символ, що знаходиться в заданій стрічці не присутній в шаблоні взагалі, то зрозуміло, що можна зрушити вправо на повну довжину шаблону, без будь-яких побоювань. У загальному випадку, кожному символу ставиться у відповідність величина, що дорівнює різниці довжини шаблону і порядкового номера символу (якщо символ повторюється, то береться саме праве входження). Ясно, що ця величина буде в точності дорівнює порядковому номеру символу, якщо рахувати від кінця рядка, що і дає можливість зміщатися вправо на максимально можливе число позицій.

**Приклад.**

Нехай вихідний рядок дорівнює «somestring» і шаблон дорівнює «string». Тепер будуємо таблицю зміщень, вона буде дорівнює довжині шаблона для всіх символів, які не зустрічаються в шаблоні, і порядковому номеру з кінця для інших (крім останнього, для нього теж береться довжина шаблону):

Таблиця 2.8

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| s | T | r | I | n | g |
| 5 | 4 | 3 | 2 | 1 | 6 |

Тепер поєднуємо наші рядки:

Таблиця 2.9

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| s | o | m | E | s | t | r | i | n | g |
| s | t | r | I | n | g |  |  |  |  |

Порівнюємо останні символи: бачимо, що ”t” не дорівнює “g”. Беремо значення зміщення для символу “t”, воно дорівнює 4. Зрушуємо рядок вправо на 4 позиції, і вуаля:

Таблиця 2.10

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| s | o | m | e | s | t | r | i | n | g |
|  |  |  |  | s | t | r | i | n | g |

Далі алгоритм буде порівнювати символи, від останнього і до першого в шаблоні з відповідними символами в заданій стрічці. Як тільки він порівняє останній, то буде з'ясовано, що це і є перше входження.

Реалізація алгоритму наведена в Додатку 4.

**2)Алгоритм Чжу – Такаоки**

Ця модифікація найчастіше використовується для порівняння відрізків ДНК(дезоксирибонуклеинова кислота). Де алфавіт побудований всього лиш на 4 символах: А, Т, Г, Ц. В даному алгоритму евристика стоп-символа вже не задовільняє нас навіть на достатньо малих відрізках, тому принцип побудови такого алгоритму будується на: замість одного стоп-символу ми будуємо таблицю для пари символів (того який не відповідний та попередній перед ним).

**3)Алгоритм турбо Бойєра – Мура**

Турбо-алгоритм, розроблений групою вчених очолюваних М.Крошмором, він пропонує інший підхід до малих алфавітів і одночасно вирішує другу проблему – квадратичну складність в найгіршому виді.

Окрім евристики стоп-символа і евристики співпадаючого суфікса, використовується третя евристика – евристика турбозсуву.

Нехай перший раз збігся суфікс UV (і спрацювала евристика суфіксів, забезпечивши повне перекриття цього суфікса), другий раз - більш короткий V (можливо, V = ∅).

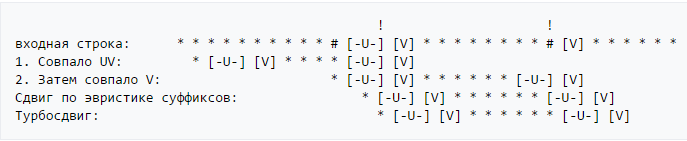


Рис. 2.1

За малюнком видно, що мінімальний можливий зсув - | U |. В іншому випадку два символу, позначені знаками оклику, у вхідному рядку різні, а в шаблоні однакові. В цьому і полягає евристика турбозсуву. Алгоритм виконує свою роботу за 2 · n порівнянь до першого збігу в гіршому випадку.

**Розділ 3**

**Експериментальний аналіз алгоритмів**

**3.1. Суть експерименту**

Отже ми розглянули кілька алгоритмів, тому час переходити до практичної реалізації та до тестування цих алгоритмів. Для порівняння швидкодії ми будем виміряти час, за який алгоритм виконує конкретно поставлене завдання.

Як я і писав на початку цієї роботи я використаю 2 види тестуючих пристроїв. Це телефон на операційній системі Android, а також емулятор AVD для персонального комп’ютера.

Перша група тестуючих пристроїв:

1. Ноутбук Aser Aspire F5-573g-31W8  
   Процесор: Intel core I3-6100U(2.3 GHz)  
   ОЗУ: 4096 Mb DDR4(2133 MHz)
2. Ноутбук Lenovo G780  
   Процесор: MobileDualCore Intel Pentium 2020M(2.4 GHZ)  
   ОЗУ: 6144 Mb DDR3(1600 MHz)

Друга група тестуючих пристроїв:

1. Телефон Samsung Galaxy Star Plus  
   Процесор: Cortex A5 (1 GHz)  
   ОЗУ: 512 Gb (533 MHz)  
   Версія Android: 4.1
2. Телефон Meizu M2 Note  
   Процесор: MediaTek MT6735 (1,3 ГГц)  
   ОЗУ: 2 Gb LPDDR3 (800 MHz)  
   Версія Android: 5.1

На кожному пристрою буде проводитися 3 типа тестів:

1. Кількість символів <100
2. Кількість символів <500
3. Кількість символів <1000

Зрозуміло, що ці тести будуть мати дещо об’єктивний результат, адже він буде залежати від характеристик і завантаження тестуючого пристрою. Тому під час тестів буде відключено всі фонові програми, які впливають на роботу програми. При запуску однієї і тієї ж задачі ми можемо отримати різний час, тому відбудеться кілька запусків, з яких вибиратиметься найкращий результат(тобто з найменшим часом роботи програми).

* 1. **Результати та аналіз експерименту**

Експеримент проводився в однакових умовах для всіх пристроїв, тому можна порівняти результати.

Результати експерименту проведених на ноутбуках наведені в таблиці:

Таблиця 3.1



Результати експерименту проведених на телефонах наведені в таблиці:

Таблиця 3.2



З таблиць видно, що алгоритм Бойєра-Мура-Хорспула впорався з експериментальним завданням швидше за інших. Слід, однак, зауважити, що його ефективність зростає лише зі збільшенням довжини рядка і, відповідно, довжини зразка. Так при довжині рядка меншим за 100 символів, він показав себе гірше, ніж послідовний пошук. Аналогічні результати показує і алгоритм КМП, як для коротких, так і для довгих слів. Його можна використовувати як універсальний, коли невідомі довжини рядка й зразка.

Алгоритм Рабіна, при його схожості з послідовним працює швидше, а його простота і малі трудовитрати на реалізацію, роблять його привабливим для використання у неспеціальзованих програмах.

Найгірший результат показав алгоритм послідовного пошуку. Як передбачалося при невеликому збільшенні довжини рядка, він працює істотно повільніше інших алгоритмів.

**Висновок**

У процесі виконання курсової роботи були розглянуті різні алгоритми пошуку підрядка в рядку та проведено їх порівняльний аналіз.

Вивчивши отримані результати можна зробити висновок, що алгоритм Бойєра-Мура-Хорспула є провідним за всіма параметрами. Але, як показує експеримент, алгоритм Кнута-Моріса-Пратта, перевершує алгоритм БМХ на невеликих довжинах зразка. Тому не можна робити висновок, котрий з алгоритмів є найоптимальнішим. Кожен алгоритм ефективно працює в певних класах завдань, про це також говорять різні вузькоспрямовані удосконалення кожного з алгоритмів. Таким чином, тип алгоритмів пошуку підрядка в рядку слід вибирати тільки після точної постановки завдання, визначення її цілей і функціональності, які вона повинна реалізувати.

Тільки після цього етапу необхідно переходити до реалізації програмного коду.

У зв'язку з глобалізацією інформації в мережі Internet був розроблений інтелектуальний пошук. Який дозволяє знайти документ за змістом що міститься в ньому, тобто документи по заданій темі. У системі реалізований алгоритм з використанням комп'ютерної обробки документа. Відповідно до гіпотези Ципфа сенс документа залежить від частоти термінів, що зустрічаються в документі. Однак набагато важливіше не сама частота слова, а те, наскільки часто в даному документі це слово зустрічається щодо інших слів.

Щодо самої реалізації курсової роботи і написання програмного коду під один з фреймворків Java, Android – цей процес майже не відрізняється від написання коду для звичайної JVM. Я зробив перевірку працездатності моєї програми на кількох пристроях, а також на кількох видах пристроїв, як на самих мобільних пристроях з операційною системою Android, так і на емуляторах які були запущені на ноутбуках різних цінових категорій. І там і там програма показала себе професійно, не було помітно помилок, так званих виключеннь, які можуть бути присутні в кожній програмі.

Отже підводячи підсумок можу сказати що на даний момент не існує оптимальних алгоритмів пошуку підрядка в рядку, адже кожен з існуючих показує себе по різному на різних довжинах рядків, і написання подібної програми на Java як для JVM так і написання програми для Android є дуже близькими за значенням речами. А за умови використання “хорошого середовища розробки” цей процес не тільки легкий, але і приємний.

**Список використаних джерел**

*Інтернет джерела*

1. Алгоритм пошуку рядка – Вікіпедія [Eлектроний ресурс]. URL: <https://uk.wikipedia.org/wiki/Алгоритм_пошуку_рядка>   
   (дата звернення 20.05.2017).
2. Примітивний алгоритм пошуку рядка – Вікіпедія [Eлектроний ресурс]. URL: <https://uk.wikipedia.org/wiki/Примітивний_алгоритм_пошуку_рядка>  
   (дата звернення 20.05.2017).
3. Алгоритм Рабина – Карпа – Вікіпедія [Eлектроний ресурс]. URL:  
   <https://ru.wikipedia.org/wiki/Алгоритм_Рабина_—_Карпа>  
   (дата звернення 21.05.2017).
4. Алгоритм Кнута – Морріса – Пратта – Вікіпедія [Eлектроний ресурс]. URL:  
   <https://uk.wikipedia.org/wiki/Алгоритм_Кнута_—_Морріса_—_Пратта>  
   (дата звернення 22.05.2017).
5. Алгоритм Бойера – Мура – Вікіпедія [Eлектроний ресурс]. URL:  
   <https://ru.wikipedia.org/wiki/Алгоритм_Бойера_—_Мура>  
   (дата звернення 22.05.2017).
6. @zelserg, Это маленькое чудо — алгоритм Кнута-Морриса-Пратта (КМП) – Хабрахабр [Eлектроний ресурс]. URL: <https://habrahabr.ru/post/307220>  
   (дата звернення 24.05.2017).
7. Тимур Ибремпашаев, Упрощений алгоритм Бойера – Мура - Хабрахабр [Eлектроний ресурс]. URL: <https://habrahabr.ru/post/116725>  
   (дата звернення 24.05.2017).
8. @DarkGenius, Поиск подстроки. Алгоритм Кнута-Морриса-Пратта - Хабрахабр [Eлектроний ресурс]. URL: <https://habrahabr.ru/post/191454>  
   (дата звернення 27.05.2017).

**Додаток 1**

Реалізація алгоритму послідовного пошуку

private String lineSearch(String subStr, String str) {

int i = 0;

while (i <= str.length() - subStr.length()) {

if (str.substring(i, i + subStr.length()).equals(subStr)) {

return "Знайдено";

} else {

i++;

}

}

return "Не знайдено";

}

**Додаток 2**

Реалізація алгоритму Рабіна – Карпа

private String rabinSearch(String subStr, String str) {

int strHCode = str.substring(0, subStr.length()).hashCode();

int subStrHCode = subStr.hashCode();

for (int i = 0; i + subStr.length() < str.length(); i++) {

if (strHCode == subStrHCode) {

if (str.substring(i, i + subStr.length()).equals(subStr)) {

return "Знайдено";

}

} else {

strHCode = str.substring(i + 1, i + 1 + subStr.length()).hashCode();

System.out.println(strHCode);

}

}

if (strHCode == subStrHCode) {

if (str.substring(str.length() - subStr.length(), str.length()).equals(subStr)) {

return "Знайдено";

}

}

return "Не знайдено";

}

**Додаток 3**

Реалізація алгоритму Кнута-Морріса-Пратта (КМП)

private int[] prefix(String s) {

int[] res = new int[s.length() + 1];

for (int i = 0; i < res.length; i++) {

res[i] = 0;

}

for (int i = 1; i < s.length() + 1; i++) {

int k = res[i - 1];

while (k > 0 && s.charAt(k) != s.charAt(i - 1)) {

k = res[k - 1];

}

if (s.charAt(k) == s.charAt(i - 1)) {

k++;

}

res[i] = k;

}

return res;

}

private String morisSearch(String subStr, String str) {

int index = -1;

int[] f = prefix(subStr);

int k = 0;

for (int i = 0; i < str.length(); i++) {

while (k > 0 && subStr.charAt(k) != str.charAt(i)) {

k = f[k - 1];

}

if (subStr.charAt(k) == str.charAt(i)) {

k = k + 1;

}

if (k == subStr.length()) {

index = i - subStr.length() + 1;

break;

}

}

if (index > -1) {

return "Знайдено";

}

return "Не знайдено";

}

**Додаток 4**

Реалізація алгоритму Бойєра-Мура-Хорспула (БМХ)

private String muraSearch(String subStr, String str) {

HashMap<Character, Integer> offsetTable = new HashMap<>();

for (int i = 0; i < 256; i++) { offsetTable.put((char) i, subStr.length()); }

for (int i = 0; i < subStr.length() - 1; i++) {

offsetTable.put(subStr.charAt(i), subStr.length() - i - 1);

}

int i = subStr.length() - 1;

int j = i;

int k = i;

while (j >= 0 && i <= str.length() - 1) {

j = subStr.length() - 1;

k = i;

while (j >= 0 && str.charAt(k) == subStr.charAt(j)) {

k--; j--;

}

i += offsetTable.get(str.charAt(i));

}

if (k >= str.length() - subStr.length()) {

return "Не знайдено";

} else {

return "Знайдено";

} }