**Київський національний університет імені Тараса Шевченка**

**Факультет комп’ютерних наук та кібернетики**

**Алгоритми та складність**

**Завдання № 8**

**Звіт**

**Виконав:**

студент групи К-29

Грищенко Юрій Анатолійович

**Київ-2019**

**Умова задачі**

Реалізуйте алгоритми пошуку зразка в текстовому рядку: наївний, КПМ, Хорспула, Боєра-Мура та Рабіна-Карпа та порівняйте їх ефективність. Виконайте пошук зразків різної довжини у випадковому бінарному тексті та випадкового слова у природньому тексті цієї мови.

**Опис алгоритмів**

Нехай треба знайти зразок pattern довжиною m у рядку string довжиною n.

Наївний алгоритм пошуку полягає у тому, що ми розглядаємо кожний символ у string, і перевіряємо m наступних символів на співпадіння з pattern. Зрозуміло, що складність такого алгоритму буде O(nm). Дуже часто одні й ті самі символи порівнюються кілька разів.

**Алгоритм Рабіна-Карпа**

Суть алгоритму Рабіна-Карпа полягає у тому, що ми використовуємо певну hash-функцію, яка певним чином характеризує послідовність символів.[1] Порівнявши hash-значення для pattern і для певного під-рядка string, можемо перевірити, чи є вони приблизно схожими одна на одну. Точна перевірка на рівність рядків відбудеться лише тоді, коли hash-функції співпадають.

Ефективність цього алгоритму порівняно зі звичайним залежатиме від вибору hash-функції. Для цієї задачі використовують так звані кільцеві хеш-функції[2] – в нашому випадку це функції, які мають таку властивість:

1. Якщо відоме значення хеш-функції для певного підрядка довжиною n (наприклад, abcdefgh) то значення функції для «наступного» підрядка (наприклад, dcdefghi) повинно обчислюватися за O(1).

Також бажано, щоб функція видавала якомога менше колізій[3] – тобто для різних послідовностей символів бажано видавати різні значення. Це робиться для того, щоб запобігти повних перевірок на співпадіння, що займає O(k) часу.

Таким чином, на практиці можна отримати час виконання, близький до O(n). Нам доведеться повністю шукати значення хеш-функції лише один раз, а для всіх наступних n під-рядків знаходження відбуватиметься за O(1).

Досить зручну та ефективну хеш-функцію наводить сам Рабін – це так званий хеш Рабіна[1]:

Для рядка a0a1a2…an хеш Рабіна має значення:

f(x) = bna­­0 + bn-1a1 + … + ban-1 + an mod p,

де a – ASCII значення символа, p – якесь просте число, b – будь-яке натуральне число (бажано 256).

Алгоритм обчислення цієї функції займає O(n) часу:

int rabin\_hash(const char\* c, int length, int mod)

{

int hash = c[0];

for(size\_t i = 1; i < length; i++)

{

hash = (hash \* 256) % mod;

hash = (hash + c[i]) % mod;

}

return hash;

}

Щоб обчислити хеш-функцію наступного підрядка a1a2…anan+1, знаючи попередній хеш і перший символ попереднього підрядка a0, достатньо:

1. Відняти від попереднього хеша bnao
2. Домножити результат на b
3. Додати an+1

Ця операція займає O(1) часу.

Ми маємо всього n-m+1 можливих під-рядків, отже, якщо не враховувати можливість кореляції хешів, середня складність алгоритму складатиме O(n).

Спробуємо врахувати можливість кореляції, тоді для кожного підрядка матимемо деякий шанс 1/p того, що доведеться просканувати весь підрядок (m символів). Тоді для кожного підрядка виконаємо O(1) + O(m/p) операцій. Тому бажано, щоб p мав дуже велике значення, тоді O(1) + O(m/p) = O(1), а загальна складність все одно буде O(n).

В гіршому випадку кореляція буде відбуватися дуже часто (шанс приблизно дорівнює 1), тоді ефективнсть буде O(nm).

**Алгоритм Боєра-Мура[4]**

Алгоритм заснований на трьох ідеях.

1. Сканування зліва направо, порівняння справа наліво. Поєднується початок тексту (рядки) і шаблону, перевірка починається з останнього символу шаблону. Якщо символи збігаються, проводиться порівняння передостаннього символу шаблону і т. д. Якщо всі символи шаблону збіглися з накладеними символами рядка, значить, підрядок знайдений, і пошук закінчено.

Якщо ж якийсь символ шаблону не збігається з відповідним символом рядка, шаблон зсувається на кілька символів вправо, і перевірка знову починається з останнього символу.

Ці «декілька», згадані в попередньому абзаці, обчислюються за двома евристиками.

2. Евристика стоп-символу. Припустимо, що ми проводимо пошук слова «колокол». Перша ж буква не збіглася — «к» (назвемо цю букву *стоп-символом*). Тоді можна зсунути шаблон вправо до останньої його букви «к».

Стрічка: \* \* \* \* \* \* к \* \* \* \* \*\*

Шаблон: к о л о к о л

Наступний крок: к о л о к о л

Якщо стоп-символу в шаблоні взагалі немає, шаблон зміщується за цей стоп-символ.

Стрічка: \* \* \* \* \* а л \* \* \* \* \* \* \* \*

Шаблон: к о л о к о л

Наступний крок: к о л о к о л

В даному випадку стоп-символ — «а», і шаблон зсувається так, щоб він виявився прямо за цією буквою. В алгоритмі Бойера-Мура евристика стоп-символу взагалі не дивиться на співпавший суфікс (див. Нижче), так що перша буква шаблону («к») опиниться під «л», і буде проведена одна завідома холоста перевірка.

Якщо стоп-символ «к» опинився за іншою буквою «к», евристика стоп-символу не працює.

Стрічка: \* \* \* \* к к о л \* \* \* \* \*

Шаблон: к о л о к о л

Наступний крок: к о л о к о л ?????

У таких ситуаціях виручає третя ідея АБМ — евристика співпавшого суфікса.

3. Евристика співпавшого суфікса. Якщо при порівнянні рядка і шаблону збіглося один або більше символів, шаблон зсувається в залежності від того, який суфікс збігся.

Стрічка: \* \* т о к о л \* \* \* \* \*

Шаблон: к о л о к о л

Наступний крок: к о л о к о л

В даному випадку збігся суфікс «ок», і шаблон зсувається вправо до найближчого «окол». Якщо підрядка «окол» в шаблоні більше немає, але він починається на «кол», зрушується до «кол», і т. д.

Обидві евристики вимагають попередніх обчислень — залежно від шаблону пошуку заповнюються дві таблиці. Таблиця стоп-символів за розміром відповідає алфавіту (наприклад, якщо алфавіт складається з 256 символів, то її довжина 256); таблиця суфіксів — шуканому шаблону. Саме через це алгоритм Бойера-Мура не враховує співпавший суфікс і неспівпавший символ одночасно — це вимагало б занадто багато попередніх обчислень.

У таблиці стоп-символів вказується остання позиція в *pattern* (**виключаючи останню букву**) кожного з символів алфавіту. Для всіх символів, що не увійшли вpattern, пишемо 0 (для нумерації з 0 — відповідно, −1). Наприклад, якщо pattern=«abcdadcd», таблиця стоп-символів буде виглядати так.

Символ a b c d [всі інші]

Остання позиція 5 2 7 6 0

Зверніть увагу, для стоп-символу «d» остання позиція буде 6, а не 8 — остання буква не враховується. Це відома помилка, що приводить до неоптимальності. Для АБМ вона не фатальна («витягує» евристика суфікса), але фатальна для спрощеної версії АБМ — алгоритму Хорспула.

Якщо розбіжність сталася на позиції i, а стоп-символ c, то зсув буде i-StopTable[c].

**Таблиця суфіксів**

Для кожного можливого суфікса  S шаблону *pattern* вказати найменшу величину, на яку потрібно зрушити вправо шаблон, щоб він знову збігся з *S*. Якщо такий зсув неможливий, ставиться |*pattern*| (в обох системах нумерації). Наприклад, для того жpattern=«abcdadcd» буде:

Суфікс [пустий] d cd dcd ... abcdadcd

Зсув 1 2 4 8 ... 8

Ілюстрація

було ? ?d ?cd ?dcd ... abcdadcd

стало abcdadcd abcdadcd abcdadcd abcdadcd ... abcdadcd

Якщо шаблон починається і закінчується однією і тією ж комбінацією букв, |*pattern*| взагалі не з'явиться в таблиці. Наприклад, для pattern=«колокол» для всіх суфіксів (крім, звичайно, порожнього) зсув буде дорівнювати 4.

Суфікс [пустий] л ол ... олокол колокол

Зсув 1 4 4 ... 4 4

Ілюстрація

було ? ?л ?ол ... ?олокол колокол

стало колокол колокол колокол ... колокол колокол

Існує швидкий алгоритм обчислення таблиці суфіксів. Цей алгоритм використовує префікс-функцію рядка.

m = length(pattern)

pi[] = префікс-функція(pattern)

pi1[] = префікс-функція(звернення(pattern))

for j=0..m

suffshift[j] = m - pi[m]

for i=1..m

j = m - pi1[i]

suffshift[j] = min(suffshift[j], i - pi1[i])

Тут suffshift[0] відповідає всій стрічці яка збіглася; suffshift[m] — пустому суфіксу. Оскільки префікс-функція обчислюється за *O*(|*pattern*|) операцій, обчислювальна складність цього кроку також дорівнює *O*(|*pattern*|).

**Приклад виконання**:

Шуканий шаблон — «abbad». Таблиця стоп-символів буде виглядати як

Символ a b [інші]

Позиція 4 3 0

Таблиця суфіксів для всіх можливих суфіксів (крім порожнього) дає максимальний зсув — 5.

abeccaabadbabbad

abbad

Накладаємо зразок на рядок. Збіги суфікса немає — таблиця суфіксів дає зсув на одну позицію. Для символу вихідної стрічки, що не збігся, «с» (5-а позиція) у таблиці стоп-символів записаний 0. Зсуваємо зразок вправо на 5-0=5 позицій.

abeccaabadbabbad

abbad

Символи 3-5 збіглися, а другий — ні. Евристика стоп-символу для «а» не працює (2-4=-2). Але оскільки частина символів збіглася, у справу включається евристика суфікса, що збігся, який робить зсув шаблон відразу на п'ять позицій!

abeccaabadbabbad

abbad

І знову збігу суфікса немає. Відповідно до таблиці стоп-символів зсуваємо зразок на 1 позицію і отримуємо шукане входження зразка:

abeccaabadbabbad

abbad

За рахунок цих оптимізацій ефективність алгоритма сягає O(n+m)[5].

**Алгоритм Боєра-Мура-Хорспула (або просто «алгоритм Хорспула»)**

* це срощений варіант алгоритму Боєра-Мура, який не враховує евристику співпавшого суфіксу.

На практиці, такий алгоритм в багатьох випадках швидший, оскільки менше часу витрачається на ініціалізацію та на початок ітерації. З іншого боку, в найгіршому випадку, складність алгоритму Хорспула може сягати O(nm) у порівнянні з набагто більш ефективним O(n + m) (наприклад, для зразка «baaaaaaaaa» у рядку «аааааааааааааааааа» виконаємо m(n-m) операцій).

**Алгоритм Кнута-Морріса-Прата**

Для пояснення подробиць алгоритму опрацюємо (порівняно штучний) перебіг алгоритму. У будь-який момент, алгоритм перебуває в стані визначеному двома цілими:

* m — позиція в S, початок сподіваного *збігу* для W,
* i — *індекс* поточного символу у W.

На кожному кроці алгоритм порівнює S[m+i] з W[i] і збільшує i якщо вони однакові. Це можна побачити на початку наступного пошуку

1 2

m: 01234567890123456789012

S: ABC ABCDAB ABCDABCDABDE

W: ABCDABD

i: 0123456

Ми рухаємось порівнюючи наступні символи W із відповідними символами з S, просуваючись від поточного до наступного в випадку збігу. Однак, на четвертому кроці, ми отримуємо S[3] як пробіл, а W[3] = 'D', розбіжність. Радше ніж починати знов з S[1], ми зауважимо, що 'A' не зустрічається між позиціями 0 і 3 в S окрім як в 0; відповідно, перевіривши всі ці символи до цього, ми занотували, що неможливо знайти початок збігу, якщо ми пробіжимо їх знов. Отже ми пересуваємось на наступний символ, встановлюючи m = 4 і i = 0. (m напочатку приймає значення 3 бо m + i - T[i] = 0 + 3 - 0 = 3 і тоді стає 4 бо T[0] = -1, де T — це таблиця «часткових збігів»)

1 2

m: 01234567890123456789012

S: ABC ABCDAB ABCDABCDABDE

W: ABCDABD

i: 0123456

Ми швидко отримуємо майже повний збіг "ABCDAB", аж як у W[6] (S[10]), знов маємо невідповідність. Однак, лиш саме перед завершенням поточного часткового збігу, ми проминули "AB", що може бути початком нового збігу, отже маємо взяти це до уваги. Раз ми вже знаємо, що ці два символи те, що нам потрібно, ми не маємо перевіряти їх знов; ми просто встановлюємо m = 8, i = 2 і продовжуємо перевіряти поточний символ. Таким чином, ми не тільки опустили символи з S, що попередньо збіглись, але й символи з W.

1 2

m: 01234567890123456789012

S: ABC ABCDAB ABCDABCDABDE

W: ABCDABD

i: 0123456

Цей пошук зазнає невдачі одразу, бо наш зразок не містить пробілу, як і за першої спроби, ми повертаємось до початку W і починаємо пошук на наступному символі S: m = 11, встановивши i = 0.

1 2

m: 01234567890123456789012

S: ABC ABCDAB ABCDABCDABDE

W: ABCDABD

i: 0123456

І знов ми відразу ж наштовхуємось на відповідність "ABCDAB", але наступний символ, 'C', не збігається з кінцевим символом 'D' слова W. Як і раніше, ми встановлюємо m = 15, щоб почати з двосимвольного рядку "AB", що веде до поточної позиції, встановлюємо i = 2, і продовжуємо зіставляння з поточної позиції.

1 2

m: 01234567890123456789012

S: ABC ABCDAB ABCDABCDABDE

W: ABCDABD

i: 0123456

Цього разу ми можемо завершити зіставляння, першим символом якого є S[15].

**Приклад роботи алгоритму побудови таблиці[6]**

Спочатку ми розглянемо приклад W = "ABCDABD". Ми побачимо, що він подібний до головного пошуку й ефективний з тих самих причин. Встановлюємо T[0] = -1. Для знаходження T[1], ми маємо виявити суфікс "A", який також є префіксом для W. Але такого суфіксу не існує, отже T[1] = 0. Так само, T[2] = 0.

Продовжуючи з T[3], ми зауважуємо наявність короткого шляху перевірки *всіх* суфіксів: припустимо ми знайшли суфікс, що є префіксом і завершується в W[2] з довжиною 2 (найбільша можлива); тоді його перший символ є префіксом префіксу W, тобто префіксом сам по собі, і завершується в W[1], а ми в випадку T[2] вже визначили, що це неможливо. Отже на кожному кроці, треба перевіряти суфікси довжини m+1 лише якщо було знайдено суфікс довжини m на попередньому кроці(тобто T[x]=m).

Звідси ми навіть не повинні розглядати підрядки довжини 2, і як і на попередньому кроці єдина спроба з довжиною 1 зазнає невдачі, тому T[3] = 0.

Ми переходимо до наступного W[4], 'A'. Та сама логіка показує, що найдовший підрядок на який треба зважати має довжину 1, і хоча тут 'A' *спрацьовує*, згадуємо, що ми шукаємо на сегмент, що завершується *до* поточного символу; звідси T[4] = 0 також.

Просуваємось до W[5], який є 'B', ми застосовуємо наступну логіку: якщо ми перед цим знайшли підзразок, що починається перед попереднім символом W[4], що триває до поточного W[5], тоді зокрема він мав би мати вірний початковий відтинок, що завершується в W[4], що заперечується фактом, що ми вже виявили 'A' як єдиний вірний відтинок, що завершується в W[4]. З цього випливає, що ми не маємо дивитись до W[4]. Отже T[5] = 1.

Нарешті, ми розглядаємо наступний символ у відтинку з початком у W[4] = 'A', це буде 'B', він збігається з W[5]. Далі більше, ті ж доводи, що й раніше кажуть, що ми не повинні дивитись перед W[4] для знаходження відтинку для W[6], і ми встановлюємо T[6] = 2.

Отже ми укладаємо таку таблицю:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **i** | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| **W[i]** | A | B | C | D | A | B | D |
| **T[i]** | −1 | 0 | 0 | 0 | 0 | 1 | 2 |

Через те, що дві складові алгоритму (а саме побудова таблиці та власне пошук) мають складності, відповідно, O(m) і O(n), складність всього алгоритму становить O(n + m).

**Інтерфейс користувача:**

Програма виконує пошук у бінарному та природньому рядку 10,000 разів і виводить загальний час пошуку:

======== Human text search: ========

Naive: 693 ms

Boyer-Moore: 1014 ms

KMP: 662 ms

Horspool: 446 ms

Rabin-Karp: 1870 ms

======== Binary search: ========

Naive: 1181 ms

Boyer-Moore: 2250 ms

KMP: 2027 ms

Horspool: 1386 ms

Rabin-Karp: 2937 ms

Як видно, на моєму компютері «людський» текст всі алгоритми шукають з одкаковою ефективністю, проте найефективніший – алгоритм Хорспула. Як не дивно, бінарний текст найшвидше обшукує наївний алгоритм – можливо, це повязано з такими явищами як локальність кешу, оптимізаціями на рівні компілятора, і т.д.[7]

**Список використаних джерел:**

1. <https://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_algorithm>
2. <https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BB%D1%8C%D1%86%D0%B5%D0%B2%D0%BE%D0%B9_%D1%85%D0%B5%D1%88>
3. <https://en.wikipedia.org/wiki/Hash_function#Uniformity>
4. <https://uk.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%91%D0%BE%D0%B9%D0%B5%D1%80%D0%B0_%D0%9C%D1%83%D1%80%D0%B0#%D0%A2%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D1%8F_%D1%81%D1%82%D0%BE%D0%BF-%D1%81%D0%B8%D0%BC%D0%B2%D0%BE%D0%BB%D1%96%D0%B2>
5. <https://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_string-search_algorithm#Performance>
6. <https://uk.wikipedia.org/wiki/Алгоритм_Кнута_—_Морріса_—_Пратта>
7. <https://stackoverflow.com/questions/11227809/why-is-processing-a-sorted-array-faster-than-processing-an-unsorted-array>