5. Пошук рядків. Алгоритми KMP і RK

Задача пошуку підрядка у рядку

- **Постановка задачі:** знайти патерн (підрядок) довжини `M` у тексті (рядку) довжини `N`
- Зазвичай `N >> M`
- **Приклад** (збіг позначено червоним):

Патерн	L	O	W							
Текст	H	E	L	L	O	W	O	R	L	D

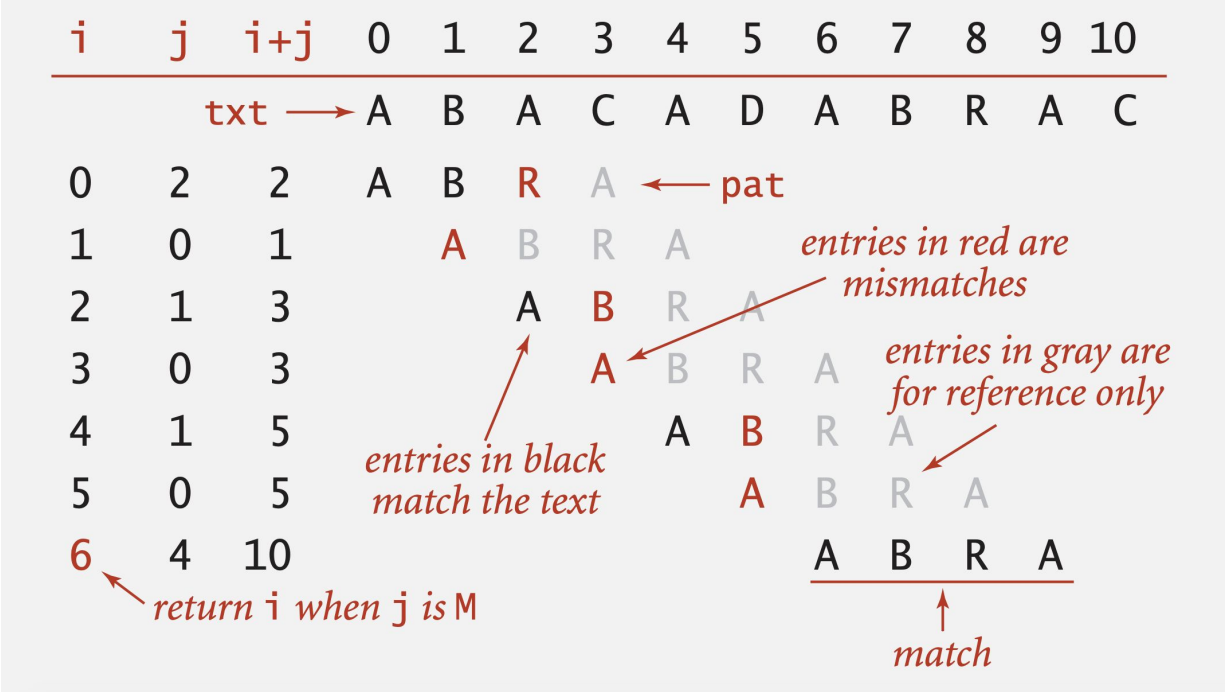
Застосування. Вимоги

- Пошук документів/веб-сторінок/файлів за ключовим словом
- Фільтрація спаму
- Модерація контенту
- Системи електронного спостереження

- Пошук зазвичай ділять на **чіткий (exact search)** – пошук точного збігу, а також **нечіткий (fuzzy search)** – пошук релевантних документів за неповним збігом (для врахування одруківок тощо)
- Основні **вимоги** до алгоритмів пошуку – **висока швидкість** та **мінімізація використання обчислювальних ресурсів** (для уможливлення пошуку по мільярдах документів великого розміру)

Brute force пошук

- Найпростіший алгоритм пошуку підрядків – **метод brute force** (грубої сили, перебору)
- **Ідея:** шукати підрядок (патерн) в тексті, починаючи з кожної позиції в тексті
- **Приклад:**



Реалізація brute force пошуку

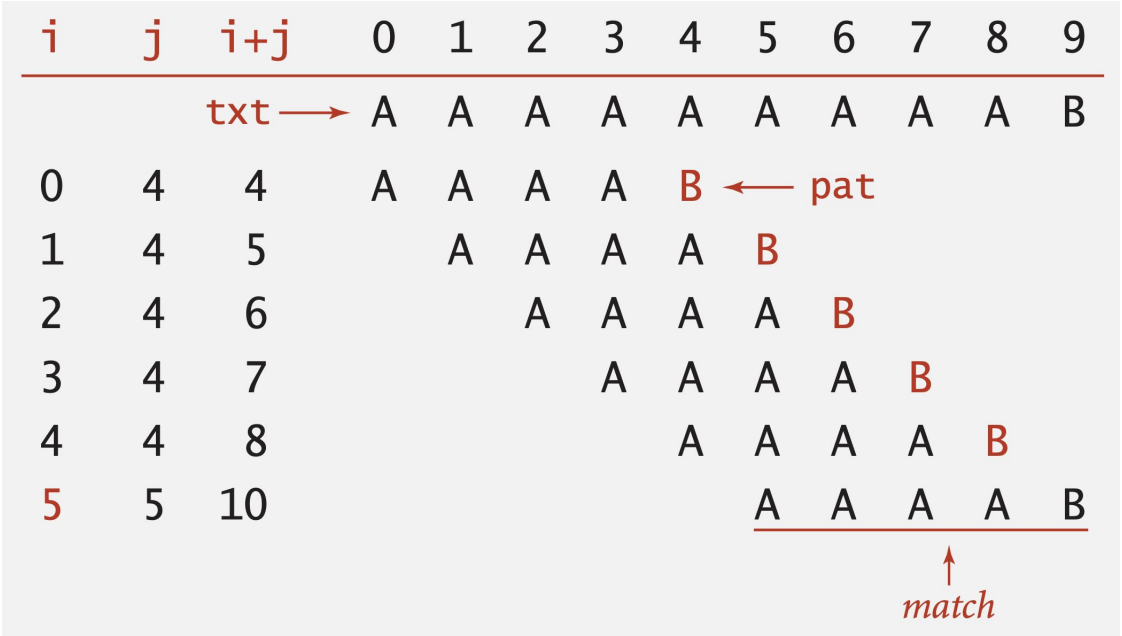
- Змінна `i` позначає індекс початку фрагменту тексту, який перевіряється на збіг з патерном
- Змінна `j` – поточна позиція в патерні
- Функція повертається позицію (індекс) початку збігу (`i`), якщо патерн знайдено, або `-1`, якщо патерну в тексті немає

```
def search(pattern: str, text: str) -> int:
    m = len(pattern)
    n = len(text)
    i = 0
    while i <= n - m:
        j = 0
        while j < m:
            if text[i + j] != pattern[j]:
                break
            j += 1
        if j == m:
            return i
        i += 1
    return -1
```

i	j	i + j	0	1	2	3	4	5	6	7	8	9	10
			A	B	A	C	A	D	A	B	R	A	C
4	3	7					A	D	A	C	R		
5	0	5						A	D	A	C	R	

Аналіз складності алгоритму brute force

- Алгоритм brute force досить повільний, особливо у випадках, коли текст і патерн містять символи, що повторюються
- Складність у найгіршому випадку: ~M*N порівнянь символів



Backup

- У алгоритмі brute force застосовується backup – повторне зчитування та перевірка вже прочитаних символів
- На практиці бажано уникнути backup – це потрібно при аналізі мережевого трафіку, для прискорення зчитування з диску тощо
- Уникнути backup можна:
  - Зберігаючи в пам’яті буфер із M останніх символів
  - Використовуючи кращі алгоритми (Кнута-Морріса-Пратта, Бойєра-Мура, Рабіна-Карпа тощо)

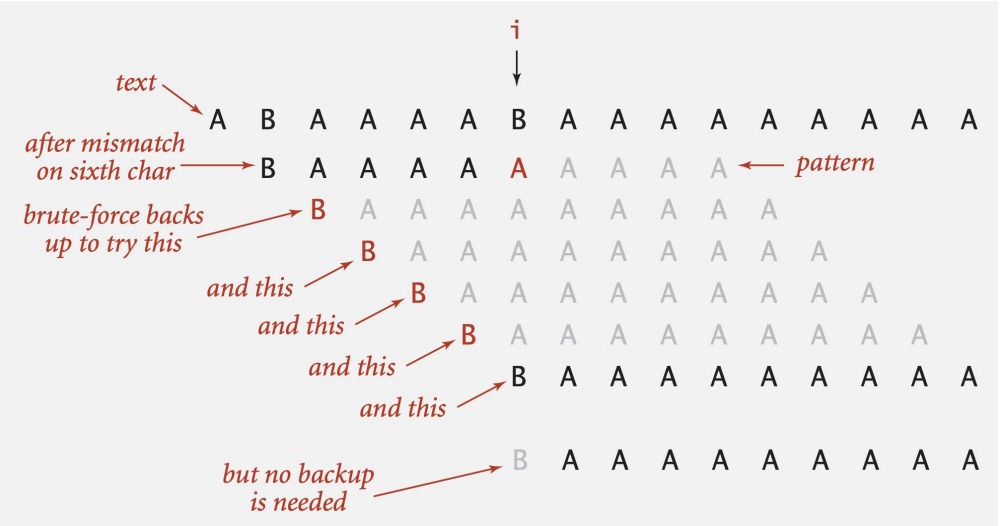
Реалізація алгоритму brute force із явним backup

- Індекс `i` вказує на кінець послідовності символів, які збігаються з патерном, у тексті
- Змінна `j` містить кількість символів, які збіглися з патерном (індекс кінця послідовності у патерні)
- Ця реалізація порівнює символи у такій самій послідовності, і має таку саму складність, як і попередня

```
def search_bfb(pattern, text):
    m = len(pattern)
    n = len(text)
    i, j = 0, 0
    while i < n and j < m:
        if text[i] == pattern[j]:
            j += 1
        else:  # backup
            i -= j
            j = 0
        i += 1
    if j == m:
        return i - m
    else:
        return -1
```

Алгоритм Кнута-Морріса-Пратта (KMP)

- Припустимо, що ми шукаємо патерн в тексті (використовуючи алфавіт {A,B}):
  - Текст: ABAAAABAAAAAAAAA
  - Патерн: BAAAAAAAAA
- Також припустимо, що ми знайшли 5 перших символів у патерні, але на 6 символі збігу нема
- Оскільки ми вже зчитали 7 перших символів тексту, **не потрібно робити повторне зчитування (backup)**



1. Детермінований скінченний автомат (DFA)

- **Детермінований скінченний автомат** (ДСА; deterministic finite automaton – DFA) – це це скінченна машина, яка приймає або відкидає певний рядок із символів, виконуючи послідовність станів, які визначаються рядком
- ДСА задається такими компонентами:
  - Скінченна множина станів
  - Скінченна множина вхідних символів (абетка, R)
  - Функція переходу
  - Початковий стан
  - Множина приймаючих (кінцевих) станів
- ДСА часто використовуються для пошуку в рядках

2. Детермінований скінченний автомат (DFA)

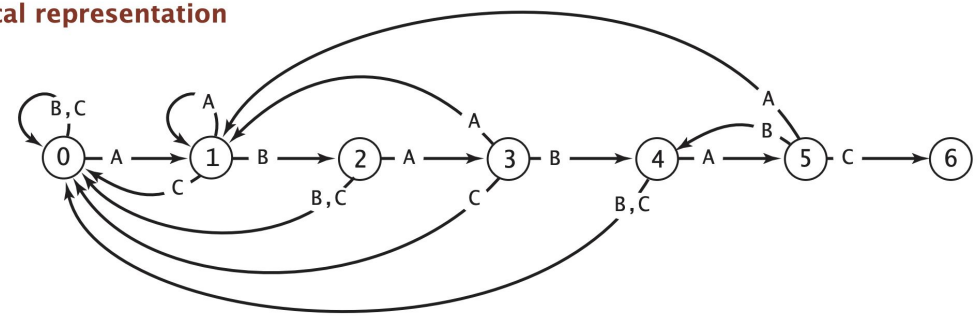
- У розрізі застосування DFA для алгоритму KMP:
  - Один початковий і один кінцевий стан
  - Рівно один перехід для кожного символу в абетці
  - Пошук завершується, якщо відбувається перехід у кінцевий стан

internal representation

	j	0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

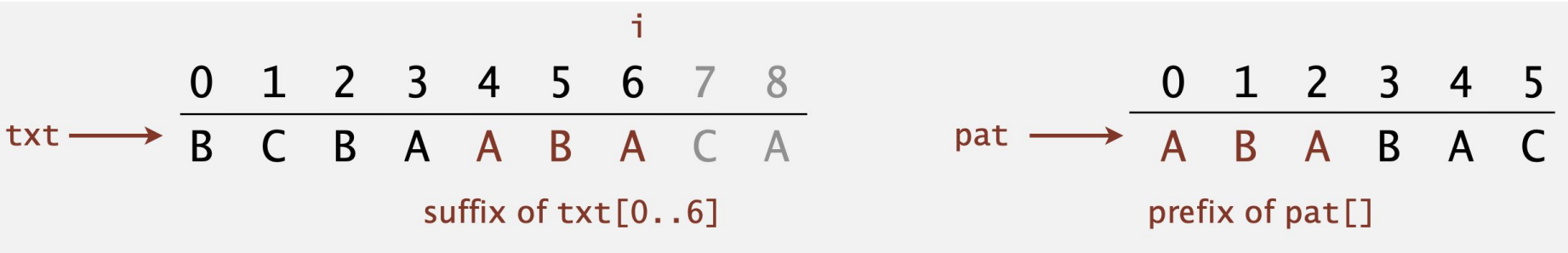
If in state `j` reading char `c`:
if `j` is 6 halt and accept
else move to state `dfa[c][j]`

graphical representation

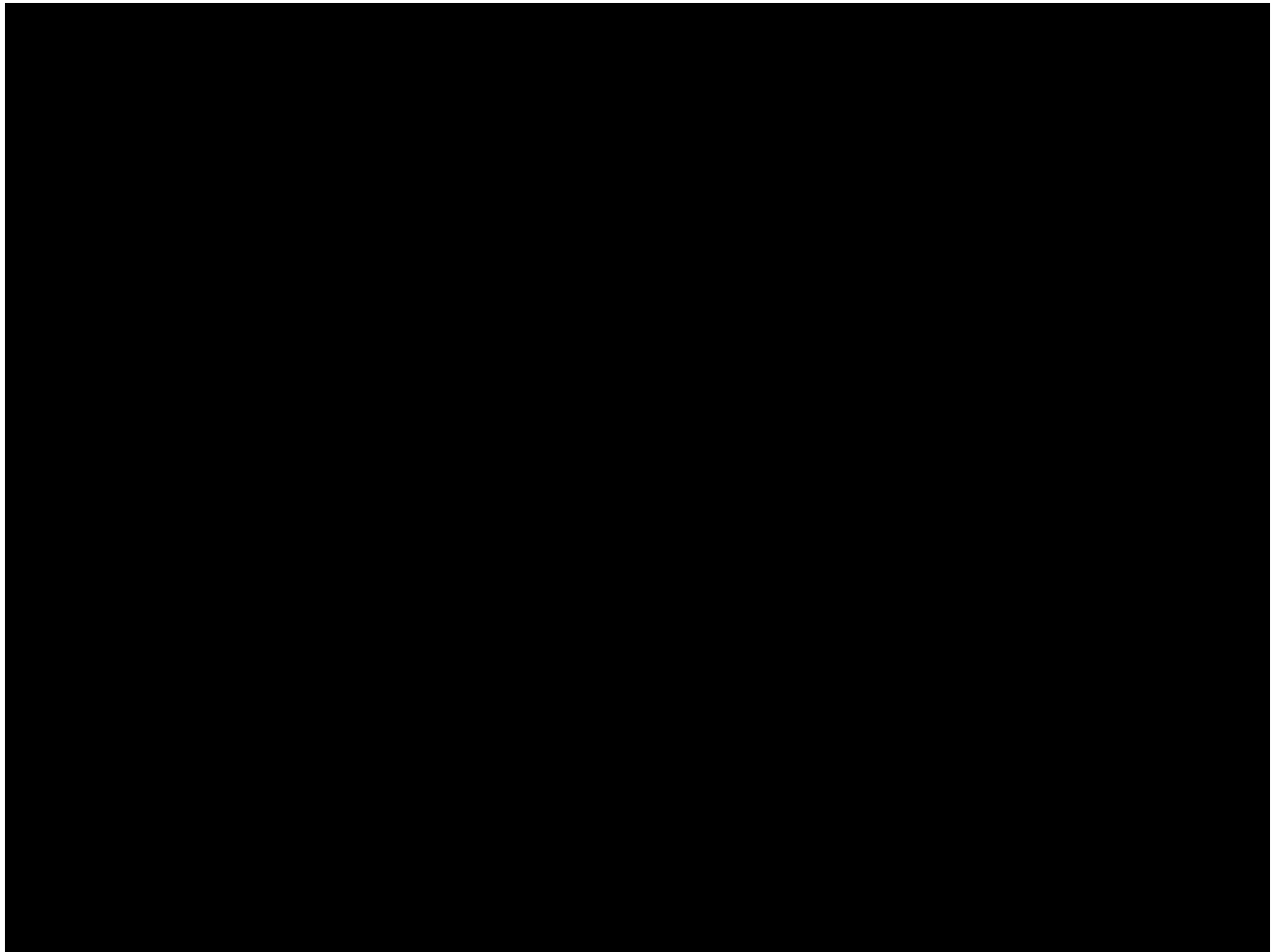


Інтерпретація DFA у алгоритмі KMP

- **Стан** – це кількість символів у патерні, які було знайдено в тексті
- Інше формулювання: це довжина найдовшого префіксу патерну, який є суфіксом `text[0..i]`
- **Приклад.** DFA у стані 3 після прочитання `text[0..6]`:



Демонстрація DFA у алгоритмі KMP



Реалізація пошуку за алгоритмом KMP

- Наведена функція є фрагментом реалізації – перед застосування необхідно побудувати DFA (двовимірний масив `dfa`)
- Індекс `i` – позиція в тексті, `j` – поточний стан
- Досягнення стану, що дорівнює довжині патерну, сигналізує, що підрядок знайдено

```
def search(self, text):
    i, j = 0, 0
    while i < len(text) and j < self.pattern_size:
        j = self.dfa[ord(text[i])][j]
        i += 1
    if j == self.pattern_size:
        return i - self.pattern_size
    else:
        return -1
```

Відмінності від brute force алгоритму

- Перед пошуком потрібно побудувати двовимірний масив `dfa`
- Вказівник на позицію в тексті ніколи не декрементується, тобто **немає повторного зчитування (backup)**
- Алгоритм потребує N доступів до символів у тексті, **працює за лінійний час**

Демонстрація побудови DFA у алгоритмі KMP

Knuth-Morris-Pratt demo: DFA construction

Include one state for each character in pattern (plus accept state).

	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
dfa[][j] A						
dfa[][j] B						
dfa[][j] C						

Constructing the DFA for KMP substring search for A B A B A C

1. Побудова DFA у алгоритмі KMP

- **Перехід у разі збігу символів (match transition).** Якщо DFA знаходиться у стані `j` (знайшли перші `j` символів з патерну) і наступний символ `ord(c) == ord(pattern[j])` (наступний символ збігається), переходимо до наступного стану `j+1` (після цього `j+1` символів з патерну збіглися)
- **Перехід у разі, якщо символи не збігаються (mismatch transition).** Якщо DFA знаходиться у стані `j` і `ord(c) != ord(pattern[j])`, потрібно залишитися у поточному стані, або ж перейти в один із попередніх станів

Демонстрація побудови DFA у алгоритмі KMP за лінійний час

Knuth-Morris-Pratt demo: DFA construction in linear time

Include one state for each character in pattern (plus accept state).

	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
dfa[][j] A						
dfa[][j] B						
dfa[][j] C						

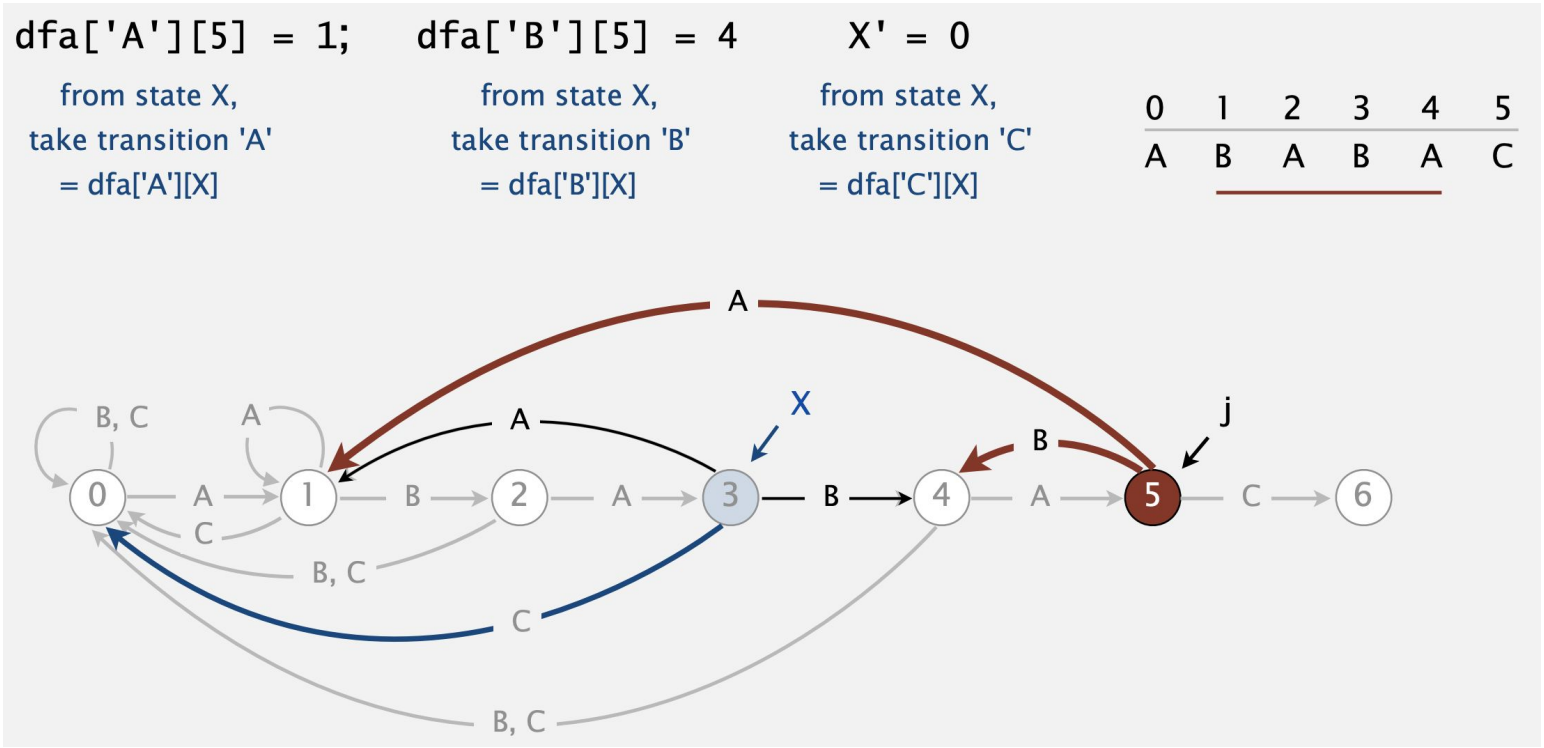
Constructing the DFA for KMP substring search for A B A B A C

3. Побудова DFA у алгоритмі KMP

- **Перехід у разі, якщо символи не збігаються (mismatch transition).** Якщо DFA у стані `j` і `ord(c) != ord(pattern[j])`, тоді останні `j-1` символів – це `pattern[1..j-1]`, за якими слідує c
- Обчислити `dfa[c][j]` можна, симулювавши `pattern[1..j-1]` у DFA, і виконавши перехід c
- Це потребує `j` кроків. Але якщо зберігати попередній стан X (після симуляції `pattern[1..j-1]`), то це потребуватиме константного часу

4. Побудова DFA у алгоритмі KMP

- Наприклад, щоб побудувати переходи із стану 5, подивимося на переходи із стану X:



1. Реалізація алгоритму KMP

```
class KMPSearcher:
    def __init__(self, pattern, alphabet_size=256):
        self.pattern = pattern
        self.pattern_size = len(pattern)
        self.alphabet_size = alphabet_size
        self.dfa = self.build_dfa()
```

- Спочатку створимо клас. Він міститиме інформацію про патерн, яку потрібно обчислити заздалегідь
- Цей клас у конструкторі приймає патерн (рядок) і розмір абетки (256 відповідає кількості символів у таблиці ASCII)
- Функція build_dfa будує DFA відповідно до заданого патерну та розміру абетки

2. Реалізація алгоритму KMP

- Спочатку створюємо двовимірний масив dfa і заповнюємо його нулями
- Код у циклі відповідає за побудову DFA:
  - Рядок з коментарем (1): копіює переходи для випадків, коли символи не збігаються (mismatch transition)
  - (2): обробляє випадок, коли символи збігаються (match transition)
  - (3): оновлює стан X, з якого починається “симуляція”

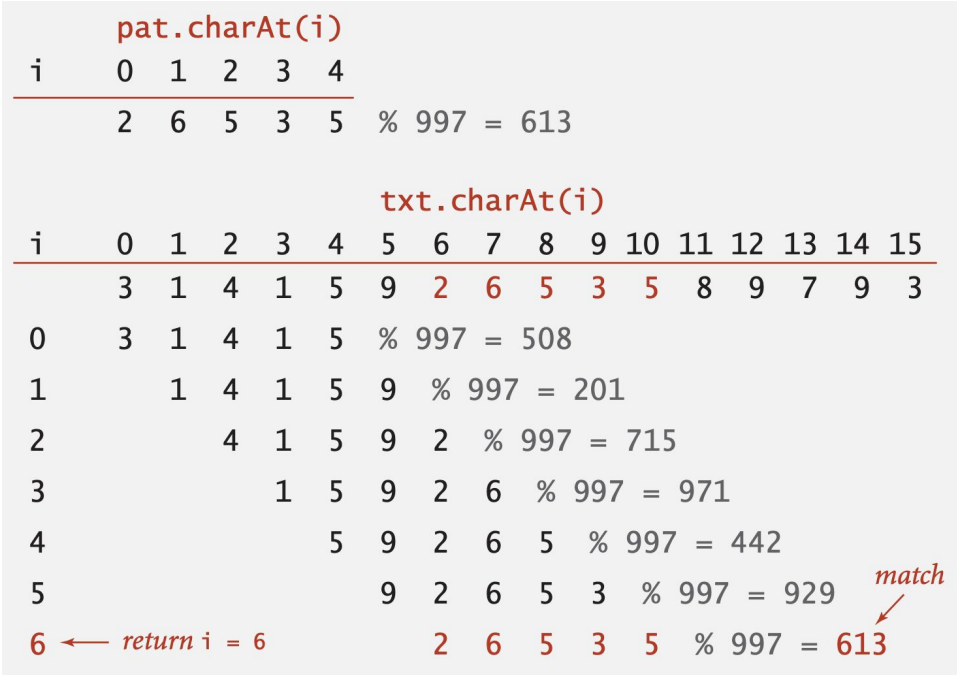
```
def build_dfa(self):
    dfa = [None] * self.alphabet_size
    for i in range(self.alphabet_size):
        dfa[i] = [0] * self.pattern_size
    x, j = 0, 0
    while j < self.pattern_size:
        for c in range(self.alphabet_size):
            dfa[c][j] = dfa[c][x]  # 1
        dfa[ord(self.pattern[j])][j] = j + 1  # 2
        x = dfa[ord(self.pattern[j])][x]  # 3
        j += 1
    return dfa
```

Складність алгоритму KMP

- Під час побудови DFA робиться `M` зчитувань символів з патерну (лише по одному разу), а алгоритм побудови DFA працює за час, пропорційний `R * M` (де `R` – розмір абетки, `M` – довжина патерну)
- Зберігання DFA потребує обсяг пам'яті, пропорційний `R * M`
- Алгоритм пошуку робить `N` зчитувань символів з тексту (лише по одному разу) у найгіршому випадку
- Відповідно, алгоритм KMP потребує `O(N + R*M)` часу у середньому та найгіршому випадках, та `O(R*M)` пам'яті
- Алгоритм KMP можна ефективно застосовувати у випадках, коли є багато текстів і потрібно шукати один і той самий патерн, оскільки у цьому випадку DFA потрібно будувати лише один раз

Алгоритм Рабіна-Карпа (RK)

- **Основна ідея:** хешування за модулем
- **Алгоритм:**
  - Обчислити хеш патерна (`pattern[0..M-1]`)
  - Для кожного `i` (початок підрядка), обчислити хеш підрядка тексту (`text[i..M+i-1]`)
  - Якщо хеш патерна дорівнює хешу підрядка тексту, перевірити рівність цих двох рядків
- **Приклад.** Основа системи числення `R = 10`; хеш рахується по модулю `Q = 997`, тобто `hash(s) = s (mod 997)`.



1. Ефективне обчислення хеш-функції

- Модульна хеш-функція:
  $h_i = c_i * R^{M-1} + c_{i+1} * R^{M-2} + \ldots + c_{i+M-1} * R^0 \pmod{Q}$, де:
  - h_i – хеш підрядка, починаючи з `i` позиції (`text[i..M+i-1]`)
  - c_i – код символа на `i` позиції в тексті (`ord(text[i])`)
  - `M` – довжина патерна
  - `Q` – модуль
  - `R` – основа системи числення (`R = 10` для десяткової) або розмір алфавіту
- Цю хеш-функцію (поліном) можна обчислити за лінійний час за допомогою **схеми Горнера**

2. Ефективне обчислення хеш-функції

```
      pat.charAt()
 i    0  1  2  3  4
      2  6  5  3  5
 0    2  % 997 = 2
 1    2  6  % 997 = (2*10 + 6) % 997 = 26
 2    2  6  5  % 997 = (26*10 + 5) % 997 = 265
 3    2  6  5  3  % 997 = (265*10 + 3) % 997 = 659
 4    2  6  5  3  5  % 997 = (659*10 + 5) % 997 = 613
```

R

Q

```
def mod_hash(string: str, m: int,
             q: int = 997, r: int = 10):
    h = 0
    for i in range(m):
        h = (h * r + ord(string[i])) % q
    return h
```

3. Ефективне обчислення хеш-функції

- Якщо обчислювати хеш-функцію для кожного підрядка у тексті, це не дасть прискорення порівняно з алгоритмом brute-force, а сповільнить його
- Тому основа алгоритму Рабіна-Карпа – обчислення хеш-функції за алгоритмом рухомого вікна
- Маємо:
  - $h_i = c_i * R^{M-1} + c_{i+1} * R^{M-2} + \ldots + c_{i+M-1} * R^0$
  - $h_{i+1} = c_{i+1} * R^{M-1} + c_{i+2} * R^{M-2} + \ldots + c_{i+M} * R^0$
- Оскільки у hi вже враховано символи від c_{i+1} до c_{i+M-1}, потрібно лише “прибрати” c_{i+M} і “видалити” c_i
- Це можна зробити так:
  - $h_{i+1} = (h_i - c_i * R^{M-1}) * R + c_{i+M}$

Алгоритм Рабіна-Карпа

- Обчислити хеш патерна (`pattern[0..M-1]`) за **схемою Горнера**
- Для кожного i (початок підрядка), обчислити хеш підрядка тексту (`text[i..M+i-1]`):
  - Для перших M символів (першого підрядка) – за **схемою Горнера**
  - Для всіх інших підрядків – використовувати **модульне хешування за алгоритмом рухомого вікна**
- Якщо хеш патерна дорівнює хешу підрядка тексту, перевірити рівність цих двох рядків

1. Реалізація алгоритму Рабіна-Карпа

- Створимо клас та обчислимо те, що можна зробити наперед: хеш патерну, а також множник $R^{M-1} \ \% \ Q$

```
class RKSearcher:
    def __init__(self, pattern, alphabet_size=256):
        self.pattern = pattern
        self.pattern_size = len(pattern)
        self.alphabet_size = alphabet_size
        self.pattern_hash = RKSearcher.mod_hash(pattern,
m=self.pattern_size, q=Q, r=alphabet_size)
        self.rm = RKSearcher.precompute_r(self.pattern_size,
alphabet_size, Q)  # R^(M-1) % Q

    @staticmethod
    def precompute_r(pattern_size, alphabet_size, q):
        rm = 1
        for i in range(pattern_size - 1):
            rm = (alphabet_size * rm) % q
        return rm

    @staticmethod
    def mod_hash(string: str, m: int, q: int, r: int):
        h = 0
        for i in range(m):
            h = (h * r + ord(string[i])) % q
        return h
```

2. Реалізація алгоритму Рабіна-Карпа

```
def search(self, text):
    text_hash = RKSearcher.mod_hash(text, m=self.pattern_size, q=Q, r=self.alphabet_size)
    if self.pattern_hash == text_hash and self.pattern == text[0:self.pattern_size]:
        return 0
    n = len(text)
    i = self.pattern_size
    while i < n:
        text_hash = (text_hash - self.rm * ord(text[i - self.pattern_size]) % Q) % Q
        text_hash = (text_hash * self.alphabet_size + ord(text[i])) % Q
        if self.pattern_hash == text_hash and
           self.pattern == text[i - self.pattern_size + 1:i + 1]:
            return i - self.pattern_size + 1
        i += 1
    return -1
```

- Тепер реалізуємо алгоритму пошуку всередині цього класу:
  - Спочатку перевіримо, чи міститься патерн у першому підрядку
  - Потім із застосування модульного хешування за алгоритмом модульного вікна перевірятимемо наступні підрядки

Аналіз складності алгоритму Рабіна-Карпа

- У теорії, якщо `Q` – достатньо велике просте число (приблизно рівне `M * N`^{2}), імовірність колізії приблизно дорівнює `1 / N`
- На практиці, вибравши `Q` як велике просте число (яке вміщається у використовувану розрядність цілих чисел), зазвичай можна припустити, що імовірність колізії приблизно `1 / Q`
- Існують **два варіанти** алгоритму Рабіна-Карпа:
  - **Точний.** При збігу хешів завжди перевіряється рівність підрядків.
    - Дуже велика ймовірність, що працює за лінійний час (але найгірший випадок – складність пропорційна `M * N`)
    - Потрібен backup (при перевірці рівності рядків)
    - Завжди повертає точний результат
  - **Приблизний.** Достатньо перевірити лише рівність хешів.
    - Завжди працює за лінійний час
    - Дуже велика ймовірність отримання коректного результату

Алгоритм Рабіна-Карпа для пошуку декількох патернів

- Часто виникає необхідність знайти в певному тексті декілька патернів (наприклад, якщо пошуковий запит задано набором синонімів)
- Алгоритм Рабіна-Карпа можна узагальнити для цієї задачі:
  - Спочатку захешуємо всі патерни, і додамо їх у множину (set). Для точного алгоритму додамо також рядки-патерни у окремий set
  - Далі будемо хешувати підрядки у тексті за алгоритмом рухомого вікна
  - Для кожного рядка перевірятимемо, чи є хеш підрядка тексту у множині хешів патернів. Якщо так:
    - У точному алгоритмі: перевіримо наявність підрядка тексту у множині рядків-патернів
    - У приблизному алгоритмі: одразу повернемо результат, що патерн знайдено

Аналіз складності алгоритму Рабіна-Карпа для пошуку декількох патернів

- Brute-force пошук патерну довжиною M у тексті довжиною N працює за $O(N*M)$ у найгіршому випадку
- Пошук K патернів однакової довжини M алгоритмом brute-force працює за $O(K*N*M)$ у найгіршому випадку
- Точний алгоритм Рабіна-Карпа працює за $O(N + M)$ (середній випадок; M операцій на перевірку рівності рядків), приблизний – за $O(N)$ (найгірший випадок)
- Пошук декількох патернів алгоритмом Рабіна-Карпа:
  - Точний алгоритм: $O(N + K*M)$ (середній випадок, оскільки перевірка наявності елементу у хеш-множині потребує $O(1)$ операцій)
  - Приблизний алгоритм: $O(N)$ (найгірший випадок)

Складності розглянутих алгоритмів

Алгоритм	Варіація	Обчислювальна складність		Пам'ять	Backup	Завжди точний результат
		Середній випадок	Найгірший випадок			
Brute force	–	O(N)	O(M*N)	O(1)	✓	✓
KMP	–	O(N + M*R)	O(N + M*R)	O(M*R)	✘	✓
RK	точний	O(N + M)	O(M*N)	O(1)	✓	✓
	приблизний	O(N)	O(N)	O(1)	✘	✘
Brute force (M)	–	O(K*N)	O(K*M*N)	O(1)	✓	✓
RK (M)	точний	O(N + K * M)	O(K*M*N)	O(K)	✓	✓
	приблизний	O(N)	O(N)	O(K)	✘	✘

Додаткові матеріали

Посилання на код:

1. Реалізації алгоритму brute force: https://gist.github.com/DmitriyTkachenko/aebf4af5057caa086a0fe3165946b738
2. Реалізація алгоритму Кнута-Морріса-Пратта (KMP): https://gist.github.com/DmitriyTkachenko/2ec037eecedc98f7603878d20eb59372
3. Реалізація алгоритму Рабіна-Карпа (RK): https://gist.github.com/DmitriyTkachenko/66e68f2fe57d7ef8ee9faf4162bec421

Додаткові матеріали (лекція/слайди):

- Лекція 19 – Substring Search: https://algs4.cs.princeton.edu/lectures/