UNIVERSITÀ
DEGLI STUDI
FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Laboratorio di Algoritmi e Strutture Dati (String match)

Autore:
Junliang Zheng

N° Matricola:
7046609

Corso principale:
Algoritmi e Strutture Dati

Docente corso:
Simone Marinai

Indice

Elenco delle figure

1 Introduzione generale

1.1 Scelta degli algoritmi

In questa relazione, esploreremo due importanti algoritmi di string matching:

- L'algoritmo di ricerca ingenua (string matching ingenuo)
- L'algoritmo di Knuth-Morris-Pratt (KMP)

La scelta di analizzare questi due algoritmi è basata sulla loro importanza nella risoluzione di problemi legati al confronto di stringhe. Entrambi offrono approcci diversi per affrontare il problema del rilevamento di sottostringhe all'interno di una stringa più grande.

1.2 Obiettivi della relazione

Lo scopo di questa relazione è fornire una panoramica dettagliata di come funzionano l'algoritmo di ricerca ingenua e l'algoritmo KMP. Esamineremo la loro spiegazione teorica, la documentazione del codice, gli esperimenti condotti per valutarne le prestazioni e l'analisi dei risultati sperimentali.

1.3 Struttura della relazione

La relazione è suddivisa in quattro parti fondamentali:

- **Spiegazione teorica dell'algoritmo**: in questa sezione, forniremo una descrizione teorica dell'algoritmo, compresi i principi di funzionamento e gli aspetti teorici.
- **Documentazione del codice**: qui esamineremo come l'algoritmo è stato implementato nel codice, discutendo le scelte di progettazione e l'approccio adottato.
- **Descrizione degli esperimenti condotti**: presenteremo gli esperimenti che abbiamo condotto per valutare le prestazioni degli algoritmi, inclusi i dettagli delle configurazioni e delle misurazioni.
- **Analisi dei risultati sperimentali**: alla luce dei dati raccolti dagli esperimenti, discuteremo le prestazioni degli algoritmi, ne valuteremo l'efficienza e trarranno conclusioni sulla loro efficacia in scenari di string matching reali.

1.4 Ambiente di sviluppo

Il codice per la realizzazione degli esperimenti è stato scritto in Python e l'IDE utilizzato è **PyCharm 2023.2.1**. Questa relazione è stata redatta con l'editor online **Overleaf**.

Parte I
Confronto tra gli Algoritmi di String Matching Ingenuo e KMP

2 Spiegazione teorica del problema

2.1 Introduzione

Nell'ambito dell'informatica, il problema del "string matching" è essenziale. Consiste nel trovare tutte le occorrenze di una stringa (pattern) all'interno di un'altra stringa più grande (testo). Due algoritmi comunemente utilizzati per risolvere questo problema sono l'algoritmo ingenuo e l'algoritmo KMP (Knuth-Morris-Pratt). Questa relazione si propone di confrontare questi due algoritmi in termini di efficienza e complessità computazionale attraverso una serie di test.

2.2 Teoria

- Algoritmo Ingenuo
  - Caratteristiche Teoriche: L'algoritmo Ingenuo, noto anche come approccio "Brute Force", è un algoritmo di ricerca di stringhe semplice e intuitivo.Esamina tutte le posizioni della stringa di test alla ricerca di una corrispondenza esatta del pattern.L'algoritmo è direttamente implementato secondo la definizione della ricerca, confrontando carattere per carattere tra il pattern e la porzione corrispondente della stringa di test.
  - Struttura Dati: L'algoritmo Ingenuo utilizza una struttura dati semplice, in quanto non richiede una struttura di dati complessa come il trie o il prefisso dell'albero.
- Algoritmo KMP (Knuth-Morris-Pratt)
  - Caratteristiche Teoriche: L'algoritmo KMP è un algoritmo di ricerca di stringhe efficiente che sfrutta la conoscenza del pattern per evitare confronti inutili.La caratteristica chiave del KMP è l'uso della funzione di prefisso, che calcola il più lungo prefisso proprio (proper prefix) di un sotto-pattern che è anche un suffisso (proper suffix) del sotto-pattern. Queste informazioni vengono utilizzate per saltare direttamente alle posizioni potenzialmente corrispondenti nella stringa di test, evitando di confrontare caratteri inutili. L'uso della funzione di prefisso rende l'algoritmo KMP più efficiente rispetto all'Ingenuo in termini di tempo di esecuzione, soprattutto per pattern e testo più lunghi.
  - Struttura Dati: L'algoritmo KMP utilizza principalmente una struttura dati nota come "funzione di prefisso" (o "prefix function"). La funzione di prefisso è un array che contiene le lunghezze dei prefissi più lunghi che sono anche suffissi per ciascun prefisso del pattern. Questa struttura dati è fondamentale per il funzionamento efficiente dell'algoritmo KMP.

In sintesi, mentre l'algoritmo Ingenuo è semplice e confronta direttamente i caratteri, l'algoritmo KMP utilizza la conoscenza del pattern per evitare confronti inutili e sfrutta la funzione di prefisso per rendere la ricerca di stringhe più efficiente. La struttura dati chiave per il KMP è la funzione di prefisso, che è una parte essenziale dell'algoritmo per migliorarne le prestazioni.

2.3 Assunti ed ipotesi

Una valutazione a priori delle prestazioni attese degli algoritmi "Ingenuo" e "KMP" (Knuth-Morris-Pratt) nella ricerca di stringhe dipende da vari fattori, tra cui la lunghezza del testo e del pattern e la presenza o meno di corrispondenze tra il pattern e il testo. L'algoritmo Ingenuo ha una complessità temporale di $O((n-m+1)m)$, dove "n" è la lunghezza del testo e "m" è la lunghezza del pattern. In generale, è un algoritmo lento, soprattutto quando il pattern è lungo o non ci sono corrispondenze nel testo. L'algoritmo Ingenuo funziona bene quando il pattern è molto breve o quando il pattern appare in molte posizioni nel testo. Tuttavia, è inefficiente quando il pattern è lungo e non ci sono corrispondenze frequenti.L'algoritmo KMP ha una complessità temporale di $O(n+m)$. L'algoritmo KMP è notevolmente più efficiente rispetto all'Ingenuo per pattern e testo di dimensioni significative. L'algoritmo KMP è efficiente quando il pattern è lungo e le corrispondenze nel testo sono scarse o

quando il pattern è lungo ma esistono solo alcune corrispondenze nel testo. Le prestazioni del KMP migliorano notevolmente con pattern più lunghi. Descriviamo le varie complessità degli algoritmi in figura 1.

Algoritmo	Preprocessing time	Matching time
Ingenuo	0	$O((n-m+1)m)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

Figura 1: Complessità degli algoritmi ingenuo e KMP

In sintesi, le prestazioni dell'algoritmo KMP sono generalmente superiori all'algoritmo Ingenuo, specialmente con pattern e testi più lunghi. L'Ingenuo può ancora funzionare bene con pattern brevi o frequenti corrispondenze nel testo, ma diventa inefficace quando il pattern è lungo e le corrispondenze sono rare. La scelta dell'algoritmo dipenderà dalle caratteristiche specifiche dei dati con cui stai lavorando e dalla necessità di prestazioni ottimali.

3 Documentazione del codice

3.1 Schema del contenuto

Per svolgere gli esperimenti, ho, prima di tutto, scritto il codice delle funzioni di ricerca di stringhe **ingenuo**, **kmp**, **calcolareprefisso**.Queste funzioni sono progettate per eseguire ricerche di stringhe efficienti, sia tramite il metodo ingenuo che attraverso l'algoritmo KMP. Inoltre, la funzione Calcolareprefisso è stata creata per calcolare la funzione di prefisso richiesta dall'algoritmo KMP.

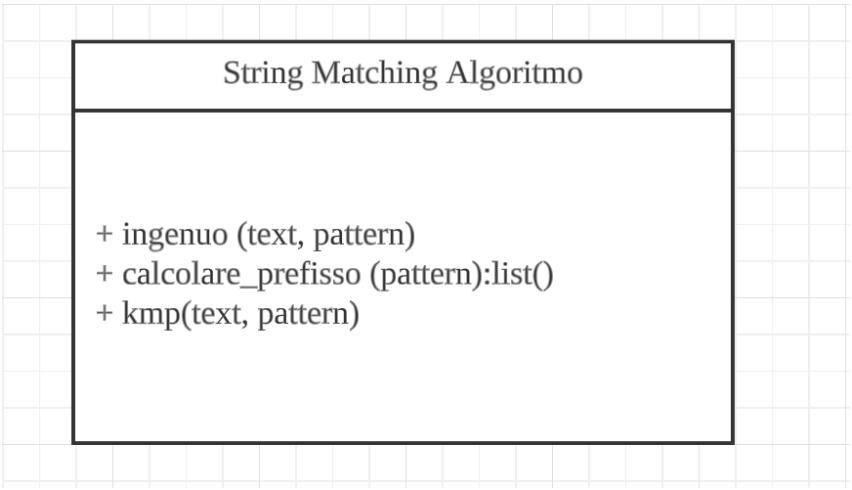


Figura 2: Diagramma degli algoritmi di String Match

In più ho scritto delle funzioni semplicemente al fine di svolgere degli test. **TestGenerator** è utile per la generazione di infografiche utili per vedere la funzione che si evolve nel tempo dei due metodi che andrò ad analizzare.

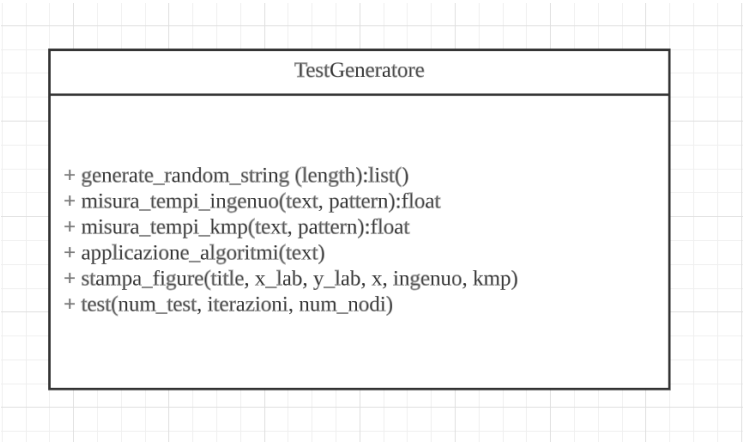


Figura 3: Diagramma di TestGenerator

3.2 Analisi delle scelte implementative

Partendo dalla funzione **Ingenuo**, qui la scelta implementativa è di utilizzare un generatore **yield**, che un normale return, per restituire le posizioni in cui il pattern è stato trovato. Questa scelta consente di risparmiare memoria, poiché le posizioni non vengono generate tutte contemporaneamente ma solo quando necessario. Analogamente anche la funzione kmp usa **yield**, questo approccio consente una gestione efficiente delle posizioni in cui il pattern è stato trovato. In più questo algoritmo sfrutta la funzione di prefisso calcolata dalla funzione **calcolareprefisso** per saltare direttamente alle posizioni potenzialmente corrispondenti nella stringa di testo.Per calcolare la funzione di prefisso, la funzione utilizza un array **prefix** di lunghezza uguale a quella del pattern. Ogni elemento in questo array rappresenta la lunghezza del prefisso più lungo che è anche un suffisso per la sottostringa del pattern fino a quella posizione.

3.3 Descrizione dei metodi implementati

In questa parte descriverò le funzionalità di ogni metodo di cui finora abbiamo parlato.

- **String Match Algoritmo**
  - **ingenuo(text, pattern)**:L'algoritmo ingenuo esamina tutte le posizioni del testo alla ricerca di una corrispondenza esatta del pattern. Ogni volta che viene trovata una corrispondenza, l'indice in cui inizia la corrispondenza viene restituito tramite un generatore.
  - **calcolarePrefisso(pattern)**: La funzione calcolarePrefisso calcola la funzione di prefisso per il pattern fornito. Questa funzione calcola il più lungo prefisso proprio che è anche un suffisso per ogni sottostringa del pattern. I risultati vengono restituiti come una lista di interi.
  - **kmp(text, pattern)**:L'algoritmo kmp utilizza la funzione di prefisso calcolata dalla funzione calcolareprefisso per eseguire una ricerca efficiente del pattern nel testo. Ogni volta che viene trovata una corrispondenza, l'indice in cui inizia la corrispondenza viene restituito tramite un generatore.
- **TestGenerator**
  - **generateRandomString(length)**:Questo metodo genera una stringa casuale di una data lunghezza utilizzando lettere maiuscole, minuscole e cifre. Restituisce la stringa generata.
  - **misuraTempiIngenuo(text, pattern)**:Questo metodo misura il tempo impiegato per eseguire l'algoritmo di ricerca ingenuo sul testo e restituisce il tempo trascorso.
  - **misuraTempiKmp(text, pattern)**:Questo metodo misura il tempo impiegato per eseguire l'algoritmo KMP sul testo e restituisce il tempo trascorso.
  - **applicazioneAlgoritmo(text, pattern)**:Questo metodo chiama misuraTempiIngenuo e misuraTempiKmp per misurare i tempi impiegati dagli algoritmi ingenuo e KMP e restituisce i risultati in una tupla.
  - **stampaFigure(title, xlab, ylab, x, naive, kmp)**:Questo metodo crea un grafico con il titolo e le etichette specificate e lo restituisce come oggetto figura Matplotlib.
  - **test(numtest, iterazioni, lunghezzatesto, lunghezzapattern)**:Questo metodo esegue una serie di test nei quali genera stringhe casuali di lunghezze variabili, quindi applica gli algoritmi ingenuo e KMP per misurare i tempi impiegati. I risultati vengono raccolti e analizzati per determinare quale algoritmo è più efficiente in base ai tempi medi di esecuzione. Infine, crea un grafico dei tempi di esecuzione.

4 Descrizione degli esperimenti condotti e analisi dei risultati sperimentali

4.1 Dati utilizzati

L'obiettivo di questo esperimento era misurare i tempi di esecuzione degli algoritmi di ricerca di stringhe **Ingenuo** e **KMP** su stringhe casuali di lunghezza variabile. Sono stati condotti test in cui sono state variate la lunghezza del testo e del pattern per valutare le prestazioni relative di questi algoritmi.

Abbiamo suddiviso l'esperimento in parti, ciascuna con configurazioni specifiche per la lunghezza del testo e del pattern:

- Testo di lunghezza **10** e Pattern di lunghezza **2**
- Testo di lunghezza maggiore di **100** e Pattern di lunghezza **2**

In ogni parte, abbiamo generato automaticamente stringhe casuali per il testo e il pattern, vedi in figura 4. Le stringhe casuali sono state create con caratteri alfanumerici casuali. Il testo ha avuto una variazione nella lunghezza, mentre il pattern è rimasto costante a 2 caratteri.

```python
def generazione_random_string(lunghezza):
    carattere = string.ascii_letters + string.digits
    return ''.join(random.choice(carattere) for _ in range(lunghezza))
```

Figura 4: Come vengono generati le stringhe

Abbiamo utilizzato le funzioni **Ingenuo** e **KMP** per cercare il pattern all'interno del testo. In ciascun test, abbiamo registrato il tempo impiegato da ciascun algoritmo per completare la ricerca.

4.2 Misurazione

I metodi di calcolo dei tempi sono nei metodi **misuraTempiIngenuo** e **misuraTempiKmp**. Innanzitutto dobbiamo prendere il tempo prima dell'esecuzione del metodo e quello dopo la sua esecuzione (esempio in figura 5).

```python
1 usage
def misuraTempiIngenuo(frase, patter):
    start = timer()
    ingenuo(frase, patter)
    stop = timer()
    return stop - start


1 usage
def misuraTempiKmp(frase, patter):
    start = timer()
    kmp(frase, patter)
    stop = timer()
    return stop - start
```

Figura 5: Come vengono presi i tempi nel codice

Questo fa si che per calcolare i tempi si possano usare:

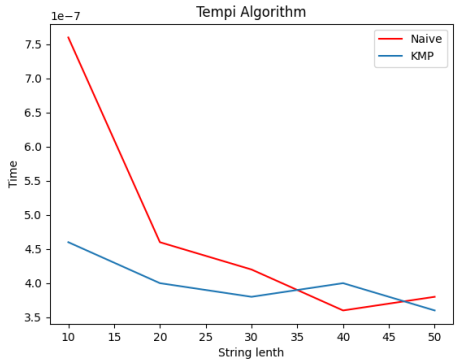
$$timeArray[x+1] = (end - start) + timeArray[x]; \tag{1}$$

che calcola sempre il tempo che ci vuole ogni qualvolta si compie un iterazione che aumenta la lunghezza delle stringhe.

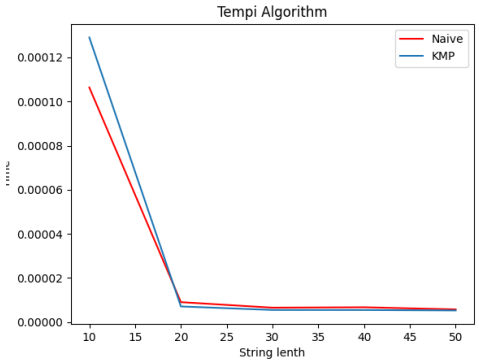
4.3 Risultati sperimentali e commenti analitici

4.4 Testo di lunghezza 10 e Pattern di lunghezza 2

Durante gli esperimenti, abbiamo osservato risultati interessanti e variazioni nelle prestazioni degli algoritmi. Notiamo come in figura 6b in alcune circostanze, l'algoritmo **Ingenuo** ha dimostrato di essere più veloce dell'algoritmo **KMP**.Questo è stato particolarmente evidente quando sia il testo che il pattern avevano lunghezze relativamente brevi. In questo caso nei testi con lunghezze da 10 fino a 20. Anche nella figura 6a l'algoritmo **Ingenuo** ha mostrato la sua efficienza rispetta all'algoritmo **KMP** per testi con lunghezza 40.



(a) Testo di lunghezza 10

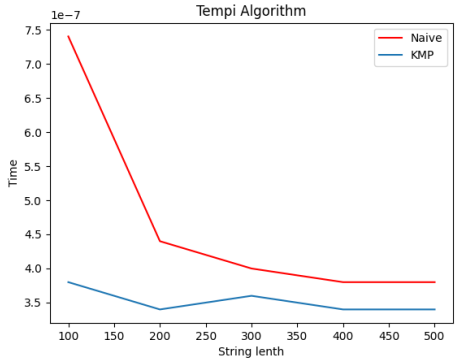


(b) Testo di lunghezza 10

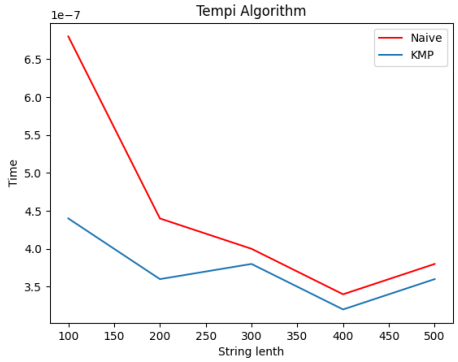
Figura 6: Testo di lunghezza 10 e pattern 2

4.5 Testo di lunghezza maggiore di 100 e pattern 2

In questi esperimenti, abbiamo condotto ricerche di pattern di 2 caratteri all'interno di un testo lungo maggiore di 100 caratteri. In ogni test, abbiamo registrato i tempi di esecuzione degli algoritmi **Ingenuo** e **KMP**. In tutti i casi, l'algoritmo **KMP** è emerso come la scelta più efficiente rispetto all'approccio **Ingenuo**. In tutte le prove effettuate l'algoritmo **KMP** ha dimostrato una performance costantemente superiore rispetto all'algoritmo **Ingenuo**. Questa costanza nella superiorità di **KMP** sottolinea la sua efficienza intrinseca nella ricerca di pattern all'interno di stringhe.La costanza dei risultati suggerisce che la lunghezza del testo non ha un impatto significativo sulle prestazioni relative degli algoritmi in questo particolare contesto, come in figura 9. Anche con un testo relativamente lungo, **KMP** riesce a mantenere un vantaggio distintivo.Questi risultati riflettono la complessità computazionale dell'algoritmo **KMP**. Grazie alla tavola di fallimento e all'approccio basato su pre-elaborazione, **KMP** è in grado di evitare ricerche inutili e ridurre notevolmente il tempo di esecuzione, anche con pattern di piccole dimensioni, in figura 7.

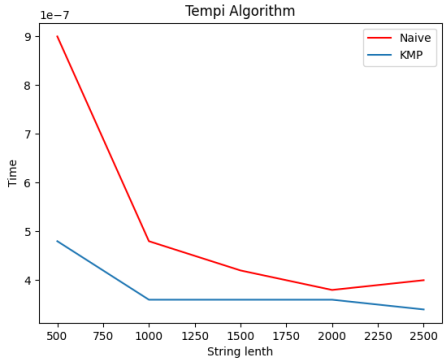


(a) Testo lungo 100

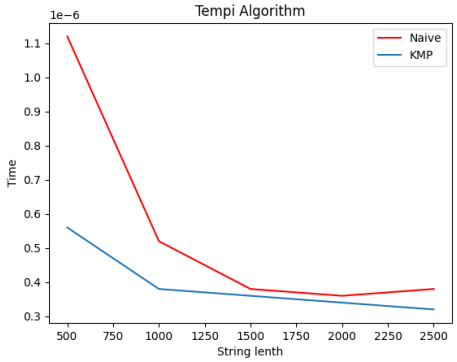


(b) Testo lungo 100

Figura 7: Test svolti con testo lungo 100 e pattern 2

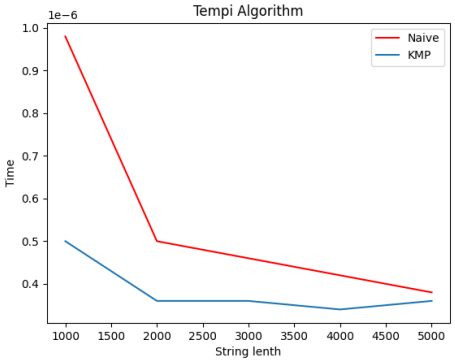


(a) Testo lungo 500

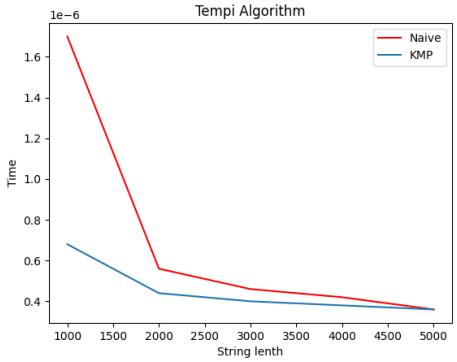


(b) Testo lungo 500

Figura 8: Test svolti con testo lungo 500 e pattern 2



(a) Testo lungo 1000



(b) Testo lungo 1000

Figura 9: Test svolti con testo lungo 1000 e pattern 2

4.6 Conclusione Generale

In base all'analisi approfondita degli esperimenti condotti, è evidente che gli algoritmi di ricerca di stringhe **Ingenuo** e **KMP** presentano differenze significative nelle loro prestazioni e complessità computazionale. Dai risultati ottenuti, è chiaro che **KMP** si è dimostrato costantemente più efficiente dell'approccio **Ingenuo** nella maggior parte delle condizioni.

L'efficienza di **KMP** deriva dalla sua complessità computazionale ottimizzata, che gli consente di evitare ricerche inutili nel testo. Questa efficienza è particolarmente evidente quando il pattern è relativamente breve rispetto al testo o quando entrambi sono di grandi dimensioni.

La scelta tra gli algoritmi **Ingenuo** e **KMP** dipende dalle specifiche esigenze del problema. Se la ricerca di stringhe è un'operazione frequente o coinvolge dati di grandi dimensioni, **KMP** si presenta come la scelta più logica grazie alla sua efficienza intrinseca.

In conclusione, questo studio ha evidenziato l'importanza di selezionare l'algoritmo di ricerca di stringhe più appropriato in base alle dimensioni dei dati e alla complessità del pattern. La comprensione delle complessità computazionali e delle prestazioni relative di questi algoritmi è essenziale per prendere decisioni informate in situazioni reali.

Riferimenti bibliografici

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (2009) Introduzione agli algoritmi e strutture dati Terza edizione, McGraw Hill.