UNIVERSITÀ
DEGLI STUDI
FIRENZE

Laboratorio di Algoritmi e Strutture Dati

Ferritti Andrea

N° Matricola: 7113347

Indice

1 Introduzione

1.1 Progetto assegnato

Il progetto che mi è stato assegnato consiste nel confronto dei seguenti algoritmi di string matching:

- Algoritmo "ingenuo" (Naive)
- Algoritmo Knuth-Morris-Pratt (KMP)

1.2 Richieste e finalità

Le attività richieste per lo svolgimento del progetto sono:

1. **Implementazione degli algoritmi:** Sviluppo di un programma Python che soddisfi i requisiti specificati, garantendo un'implementazione corretta ed efficiente degli algoritmi.
2. **Esecuzione di esperimenti:** Progettazione ed esecuzione di un insieme di esperimenti mirati a comprendere le prestazioni, i punti di forza e le eventuali limitazioni delle diverse implementazioni.
3. **Analisi dei risultati:** Raccogliere e analizzare i dati ottenuti dagli esperimenti, per trarre conclusioni sulle prestazioni degli algoritmi implementati e confrontare le diverse soluzioni.

L'obiettivo finale è quindi quello di valutare in maniera quantitativa le prestazioni delle diverse implementazioni e confrontare i risultati per identificare le soluzioni più efficienti in specifici contesti applicativi.

1.3 Convenzioni

Nella presente relazione, l'algoritmo "ingenuo" sarà indicato con il termine inglese **Naive**, per coerenza con la letteratura scientifica internazionale. Analogamente, l'algoritmo di Knuth-Morris-Pratt sarà abbreviato come **KMP**, al fine di semplificare le future referenze e allinearsi alla notazione comunemente utilizzata.

1.4 Specifiche piattaforma di test

Il computer su cui sono stati fatti i test è un MacBook Pro 2021 il cui hardaware è così composto:

- **CPU:** Apple M1 pro
- **RAM:** 16GB
- **Disco:** Macintosh HD

Il sistema operativo installato è **macOS Sonoma 14.6.1**, il linguaggio di programmazione è **Python** e l'IDE utilizzato è **PyCharm 2024.2.1**

2 String matching

2.1 Descrizione problema generale

Lo **String matching** (o Pattern matching) riguarda la ricerca di tutte le occorenze di una sottostringa (**pattern**) all'interno di una stringa più lunga (**testo**).

È un problema fondamentale in informatica e ha molte applicazioni pratiche in contesti diversi, tra queste ci sono:

- **Bioinformatica:** Identificare sequenze di DNA o proteine in grandi basi di dati biologici.
- **Motori di ricerca e database:** Trovare rapidamente documenti o record che contengono una query o un pattern specifico
- **Controllo Ortografico e Correttori Automatici:** Confrontare parole digitate con un dizionario, identificando errori e suggerendo correzioni.

2.2 Descrizione e implementazione algoritmi

2.2.1 Naive

L'algoritmo **Naive** per il pattern matching è un metodo semplice e intuitivo per la ricerca di un pattern all'interno di un testo.

L'algoritmo scorre il testo confrontando il pattern con ogni possibile sottostringa di lunghezza equivalente. In particolare, per ogni posizione iniziale nel **testo**, viene effettuato un confronto carattere per carattere tra il **pattern** e la sottostringa corrente. Se il pattern corrisponde esattamente alla sottostringa, viene registrata *un'occorrenza*. Il processo si ripete spostando l'indice di una posizione verso destra fino a esaurire tutte le possibili sottostringhe nel testo.

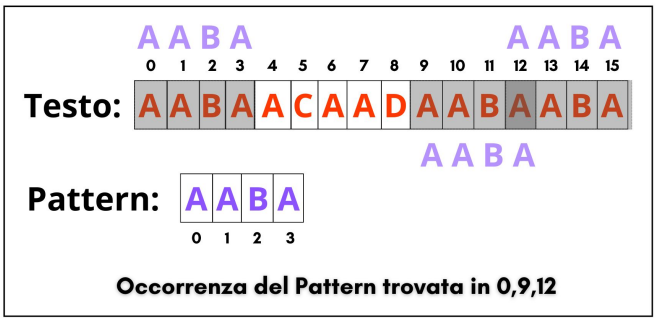


Figura 1: Esempio Naive

In questo esempio, mostrato in **Figura 1**, l'algoritmo Naive viene utilizzato per trovare il pattern `"AABA"` all'interno del testo `"AABAACAADAABAABA"` e procede in questo modo:

- **Confronto iniziale (indice 0)**: L'algoritmo inizia confrontando il pattern con i primi quattro caratteri del testo, ovvero la sottostringa `"AABA"`. Poiché il pattern corrisponde esattamente alla sottostringa:

  `Occorrenza del pattern trovata in posizione 0.`

- **Confronti successivi**: L'algoritmo continua spostandosi di una posizione a destra nel testo e confronta nuovamente il pattern con le successive sottostringhe:
  - All'indice **1**, confronta `"ABAA"`, che non corrisponde al pattern.
  - All'indice **2**, confronta `"BAAC"`, che non corrisponde al pattern.

  Questo processo continua per ogni sottostringa del testo fino all'indice **9**.

- **Nuova occorrenza (indice 9)**: All'indice **9**, viene trovata una nuova occorrenza del pattern nella sottostringa `"AABA"`. Il pattern corrisponde esattamente:

  `Occorrenza del pattern trovata in posizione 9.`

- **Ultima occorrenza (indice 12)**: Infine, all'indice **12**, viene trovata un'altra corrispondenza con la sottostringa `"AABA"`:

  `Occorrenza del pattern trovata in posizione 12.`

Per **l'implementazione** dell'algoritmo Naive, ho creato una funzione chiamata `naive_string_matcher(T, P)`, dove `T` rappresenta il testo e `P` il pattern da cercare. La funzione scorre tutte le possibili posizioni iniziali nel testo, da 0 fino a `len(T) - len(P)`. Per ogni posizione, viene confrontata la sottostringa di `T`, che parte dalla posizione corrente e ha lunghezza pari a quella di `P`, con il pattern `P`. Se corrispondono, l'algoritmo segnala un'occorrenza del pattern stampando un messaggio che indica lo spostamento `s` dove è stato trovato il pattern.

Il codice è il seguente:

```
def naive_string_matcher(T, P):
    for s in range(len(T) - len(P) + 1):
        if T[s:s + len(P)] == P:
            print(f"Occorrenza del pattern con spostamento {
                s}")
```

Questa implementazione esegue un **confronto diretto** tra ogni sottostringa del testo e il pattern, utilizzando *slicing* per estrarre la sottostringa corrente e confrontandola con `P`. Se il confronto risulta positivo, viene **indicata la posizione dello spostamento**.

2.2.2 KMP

L'algoritmo **KMP** (Knuth-Morris-Pratt) è un metodo efficiente per risolvere il problema della ricerca di un *pattern* all'interno di un testo.

A differenza dell'algoritmo **Naive**, KMP riduce il numero di confronti necessari grazie all'uso di una tabella di fallimento **LPC** (*Longest Prefix which is also Suffix*), che memorizza le lunghezze delle più lunghe sottostringhe prefisso-suffisso del *pattern*.

Questa tabella consente all'algoritmo di saltare direttamente alle posizioni rilevanti nel *pattern* quando un confronto fallisce, evitando confronti ridondanti e migliorando l'efficienza complessiva.

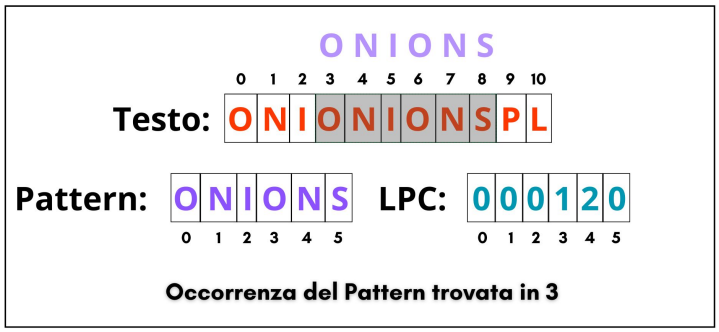


Figura 2: Esempio KMP

In questo esempio, mostrato in **Figura 2**, l'algoritmo KMP viene utilizzato per trovare il pattern `"ONIONS"` all'interno del testo `"ONIONIONSPL"` e procede in questo modo:

1. **Creazione della Tabella LPC:**
   La tabella *LPC* per il pattern "ONIONS" è:

   ```
   LPC: 0 0 0 1 2 0
   ```

   Questa tabella indica che:

   (a) Alla posizione 0, nessun prefisso è uguale a un suffisso.

   (b) Alla posizione 1, nessun prefisso è uguale a un suffisso.

(c) Alla posizione 2, nessun prefisso è uguale a un suffisso.

(d) Alla posizione 3, il prefisso "O" è uguale al suffisso "O".

(e) Alla posizione 4, il prefisso "ON" è uguale al suffisso "ON".

(f) Alla posizione 5, nessun prefisso è uguale a un suffisso.

2. **Esecuzione dell'Algoritmo KMP:**
   L'algoritmo inizia il confronto dalla posizione 0 del testo:

   - Alla posizione 0 del testo, si confronta il pattern "ONIONS" con la sottostringa corrispondente del testo "ONIONI". Il confronto fallisce alla posizione 5 (poiché il testo contiene "I" mentre il pattern richiede "S"). Grazie alla tabella *LPC*, l'algoritmo evita di ricominciare dall'inizio e può proseguire il confronto dalla posizione 3 del pattern, utilizzando l'informazione della tabella *LPC*.
   - Alla posizione 3 del testo, il confronto viene eseguito nuovamente. Questa volta il pattern "ONIONS" corrisponde esattamente alla sottostringa "ONIONS" nel testo.

     `Occorrenza del pattern trovata in posizione 3.`

L'implementazione dell'algoritmo KMP si basa su due funzioni principali: la funzione di confronto del pattern con il testo `kmp_matcher` e la funzione di calcolo della tabella *LPC* (*Longest Prefix which is also Suffix*) `compute_prefix_function`. Questa struttura consente di eseguire la ricerca del pattern nel testo in maniera efficiente, riducendo il numero di confronti ridondanti.

1. **Funzione** `kmp_matcher`**:**

   La funzione `kmp_matcher(T, P)` riceve in input il testo `T` e il pattern `P`, e si occupa di trovare tutte le occorrenze del pattern nel testo. Ecco i passaggi principali:

   - Viene calcolata la tabella *LPC* tramite la funzione `compute_prefix_function(P, len(P))`, che serve a ottimizzare i confronti successivi.
   - Si inizializza una variabile `q` a 0, che tiene traccia del numero di caratteri consecutivi corrispondenti tra il pattern e il testo.
   - Si scorre il testo `T` utilizzando un ciclo `for`, iterando su ogni carattere del testo:
     - Se il carattere corrente del pattern `P[q]` non corrisponde a quello del testo `T[i]`, l'algoritmo utilizza la tabella *LPC* per aggiornare il valore di `q` (ovvero l'indice del pattern) e saltare confronti ridondanti.
     - Se il carattere corrisponde, `q` viene incrementato.
     - Quando `q` raggiunge `len(P)`, significa che è stata trovata un'occorrenza completa del pattern nel testo, e l'algoritmo registra lo spostamento `i - len(P) + 1`.

– Dopo aver registrato l'occorrenza, `q` viene aggiornato utilizzando il valore della tabella LPC per proseguire con il confronto.

La funzione `kmp_matcher` è implementata come segue:

```
def kmp_matcher(T, P):
   lpc = compute_prefix_function(P, len(P))
    q = 0
    for i in range(len(T)):
        while q > 0 and P[q] != T[i]:
            q = lpc[q - 1]
        if P[q] == T[i]:
            q += 1
        if q == len(P):
            print(f"Occorrenza del pattern con
                spostamento {i - len(P) + 1}")
            q = lpc[q - 1]
```

2. **Funzione** `compute_prefix_function`**:**

   La funzione `compute_prefix_function(P, m)` si occupa di costruire la tabella LPC, che memorizza per ogni posizione del pattern la lunghezza del più lungo prefisso che è anche suffisso. Questa tabella è essenziale per ottimizzare i confronti durante l'esecuzione dell'algoritmo KMP.

   I passaggi principali sono i seguenti:

   - Si inizializza un array `pi` di lunghezza `m`, che rappresenta la tabella LPC, e una variabile `k`, inizialmente pari a 0, che tiene traccia della lunghezza del prefisso-suffisso più lungo trovato finora.
   - Si scorre il pattern `P` dalla posizione 1 alla posizione `m-1`:
     – Se il carattere corrente del pattern `P[q]` non corrisponde al carattere del prefisso più lungo `P[k]`, si aggiorna `k` utilizzando il valore precedente della tabella `pi`, evitando confronti inutili.
     – Se i caratteri corrispondono, `k` viene incrementato e il suo valore viene memorizzato nella tabella `pi` per la posizione corrente `q`.
   - Al termine, la funzione restituisce la tabella `pi`, che contiene le lunghezze dei prefissi-suffissi per ogni posizione del pattern.

   La funzione `compute_prefix_function` è implementata come segue:

```
def compute_prefix_function(P, m):
    pi = [0] * m
    k = 0
    for q in range(1, m):
        while k > 0 and P[k] != P[q]:
            k = pi[k - 1]
        if P[k] == P[q]:
            k += 1
        pi[q] = k
    return pi
```

2.3 Confronto algoritmi

2.3.1 Analisi Naive

La complessità temporale dell'algoritmo Naive è $O((n - m + 1) \cdot m)$ dove n è la lunghezza del testo e m è la lunghezza del pattern, in particolare possiamo analizzare i seguenti casi:

- **Caso Peggiore**: La complessità temporale dell'algoritmo Naive nel caso peggiore è $O((n - m + 1) \cdot m)$. Si verifica quando il pattern e il testo sono simili ma non identici, portando a un numero massimo di confronti. Un esempio è dato da un testo formato da caratteri ripetuti seguiti da una differenza nell'ultimo carattere (ad esempio, "aaaaa" per il pattern "aaaa").
- **Caso Migliore**: Nel caso migliore la complessità è $O(m)$ e si verifica quando il primo carattere del pattern non è presente affatto nel testo. In questo caso, l'algoritmo controlla solo il primo carattere del pattern per ogni posizione del testo e si rende subito conto che non c'è corrispondenza, evitando di fare ulteriori confronti per ogni posizione.

2.3.2 Analisi KMP

La complessità computazionale dell'algoritmo KMP è $O(n + m)$ dove $O(m)$ è dovuto alla costruzione dell'array LPC e $O(n)$ è dovuto alla ricerca del pattern nel testo poiché lo si scorre una volta, in particolare possiamo analizzare i seguenti casi:

- **Caso Peggiore**: La complessità temporale dell'algoritmo KMP nel caso peggiore è $O(n + m)$. Questo include il tempo necessario per costruire la tabella di prefissi e il tempo necessario per eseguire la ricerca del pattern nel testo. Anche nel caso peggiore, KMP esegue un numero limitato di confronti e utilizza l'informazione preelaborata per evitare confronti ridondanti.

- **Caso Migliore**: La complessità temporale nel caso migliore è anch'essa $O(n + m)$, poiché il tempo di preelaborazione e la ricerca sono sempre linearmente proporzionali alla somma delle lunghezze del testo e del pattern, indipendentemente dalle corrispondenze trovate.

2.3.3 Aspettative esperimenti

Quando si confrontano gli algoritmi Naive e KMP in esperimenti pratici, le aspettative sono:

- **Prestazioni**: L'algoritmo KMP generalmente mostrerà prestazioni significativamente migliori rispetto al Naive, specialmente con testi e pattern di grande dimensione. L'algoritmo Naive può soffrire di lentezza a causa della sua complessità $O((n - m + 1) \cdot m)$, mentre KMP mantiene la complessità $O(n + m)$ anche nel caso peggiore.

- **Efficienza**: Anche se l'algoritmo Naive può funzionare bene per pattern e testi di piccole dimensioni o con poche somiglianze, l'algoritmo KMP è generalmente più efficiente e adatto per situazioni in cui è necessario eseguire una ricerca su grandi quantità di dati. La differenza di prestazioni diventa più pronunciata con l'aumento della lunghezza del testo e del pattern.

- **Overhead della Preelaborazione**: L'algoritmo KMP comporta un overhead iniziale per la costruzione della tabella di prefissi, ma questo costo è generalmente compensato da una riduzione significativa nel numero di confronti durante la ricerca. In esperimenti pratici, il tempo di preelaborazione potrebbe essere insignificante rispetto ai benefici della riduzione dei confronti ridondanti.

- **Pattern ripetuti nel testo**: Quando il pattern contiene sezioni ripetute, come prefissi e suffissi identici, ci si aspetta che l'algoritmo KMP dimostri un vantaggio ancora maggiore rispetto al Naive. Questo perché KMP sfrutta le ripetizioni nel pattern per ridurre i confronti, mentre l'algoritmo Naive continua a scorrere il testo senza fare uso di questa informazione. In queste situazioni, la riduzione dei confronti superflui da parte di KMP dovrebbe diventare particolarmente evidente nei tempi di esecuzione.

In sintesi, gli esperimenti di confronto tra gli algoritmi Naive e KMP dovrebbero evidenziare il vantaggio dell'algoritmo KMP in termini di efficienza e scalabilità, specialmente quando si lavora con testi e pattern di grandi dimensioni o complessi.

3 Esperimenti di confronto

3.1 Descrizione degli esperimenti

Lo scopo dei test è confrontare i tempi di esecuzione degli algoritmi *Naive* e *KMP* su diverse lunghezze di testo e pattern, generati sia in maniera casuale che con ripetizioni frequenti del pattern all'interno del testo in modo da poterne valutare le loro prestazioni.

3.1.1 Setup dei test

Per ogni configurazione di test, sono stati definiti i seguenti parametri:

- **Lunghezza del testo**: Tre diverse categorie sono state considerate: *piccolo*, *medio* e *grande*. Ogni categoria contiene un insieme di lunghezze di testo che varia tra un minimo di 100 caratteri e un massimo di 500000 caratteri.
- **Lunghezza del pattern**: Anche per il pattern sono state definite tre categorie (*piccolo*, *medio*, *grande*) con lunghezze che vanno da un minimo di 50 caratteri fino a 20000 caratteri.
- **Numero di ripetizioni**: Per ogni test, l'algoritmo viene eseguito su una specifica configurazione di lunghezza del testo e del pattern per 50 ripetizioni, al fine di calcolare il tempo medio di esecuzione e ridurre l'effetto di eventuali variazioni casuali nelle prestazioni.

```
#Set di lunghezze del testo e del pattern da utilizzare
    text_lengths_sets = {
        "small": [100, 500, 1000, 2500, 5000],
        "medium": [10000, 25000, 50000, 75000, 100000],
        "large": [150000, 250000, 350000, 450000, 500000]
    }

    pattern_lengths_sets = {
        "small": [50, 100],
        "medium": [1000, 1500],
        "large": [10000, 20000]
    }

    #Esegue i test con i set di lunghezze specificati
    main(text_lengths_sets, pattern_lengths_sets,50)
```

3.1.2 Generazione del testo e del pattern

Due diversi metodi di generazione del testo e del pattern sono stati utilizzati nei test:

- **Testo e pattern casuali**: Le stringhe di testo e pattern vengono generate utilizzando caratteri alfabetici minuscoli casuali. Questa modalità simula scenari in cui il testo e il pattern non hanno alcuna struttura particolare.
- **Pattern con prefisso e suffisso uguali ripetuto nel testo**: Il pattern viene progettato con un prefisso e un suffisso identici, separati da una sezione centrale diversa. Questo pattern viene poi ripetuto all'interno del testo, creando una struttura ideale per dimostrare l'efficienza dell'algoritmo KMP, che sfrutta le somiglianze tra prefisso e suffisso per ridurre i confronti superflui rispetto all'approccio Naive.

3.1.3 Misurazione del tempo di esecuzione

Il tempo di esecuzione di ciascun algoritmo è stato misurato utilizzando il modulo `timeit` di Python, che permette di eseguire una funzione un numero predefinito di volte e misurarne il tempo totale. Il tempo medio di esecuzione per ciascuna configurazione è calcolato come segue:

$$\text{Tempo medio} = \frac{\text{tempo totale di esecuzione}}{\text{numero di ripetizioni}}$$

Dove il numero di ripetizioni è fissato a 50 per garantire l'affidabilità del risultato.

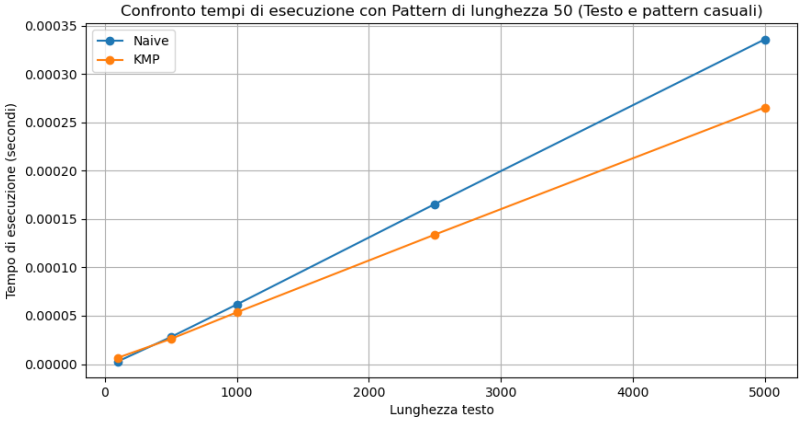
3.1.4 Confronto delle prestazioni

I test confrontano i tempi di esecuzione degli algoritmi *Naive* e *KMP* su tutte le configurazioni di lunghezze del testo e del pattern. I risultati sono poi visualizzati graficamente per ogni lunghezza del pattern, mostrando il tempo di esecuzione in funzione della lunghezza del testo.

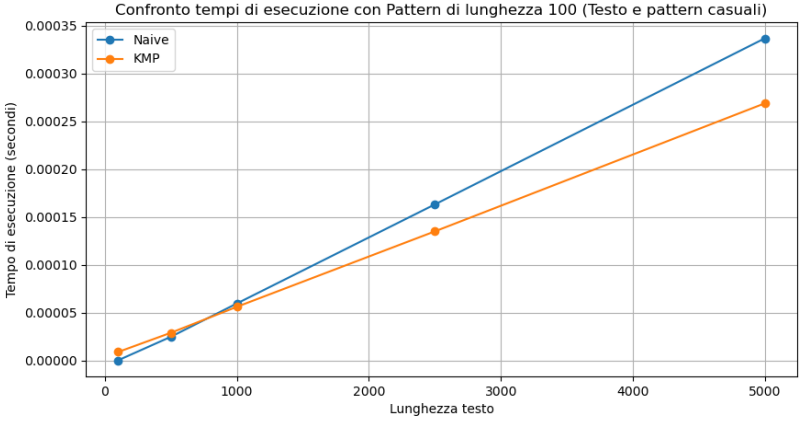
In generale, ci si aspetta che l'algoritmo *Naive* abbia un tempo di esecuzione che cresce rapidamente all'aumentare della lunghezza del testo e del pattern, mentre l'algoritmo *KMP* dovrebbe mostrare migliori prestazioni soprattutto nei casi in cui il pattern è ripetuto frequentemente, grazie all'ottimizzazione fornita dalla sua funzione di prefisso.

3.2 Testo e pattern casuali

3.2.1 Pattern di lunghezza 50 e 100 (piccolo)



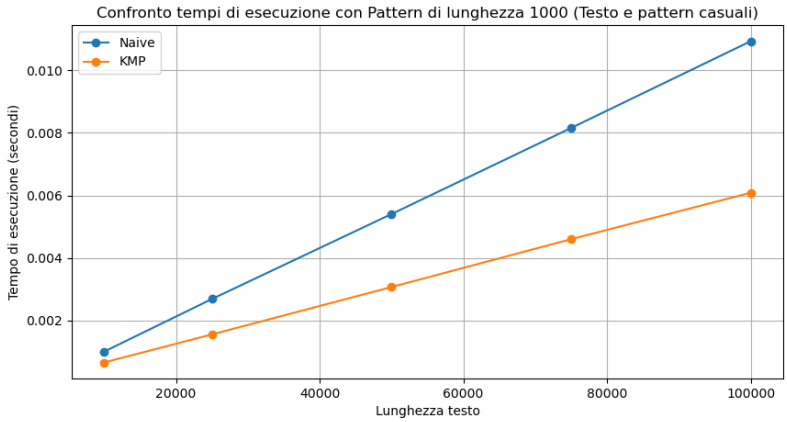
(a) Pattern di lunghezza 50



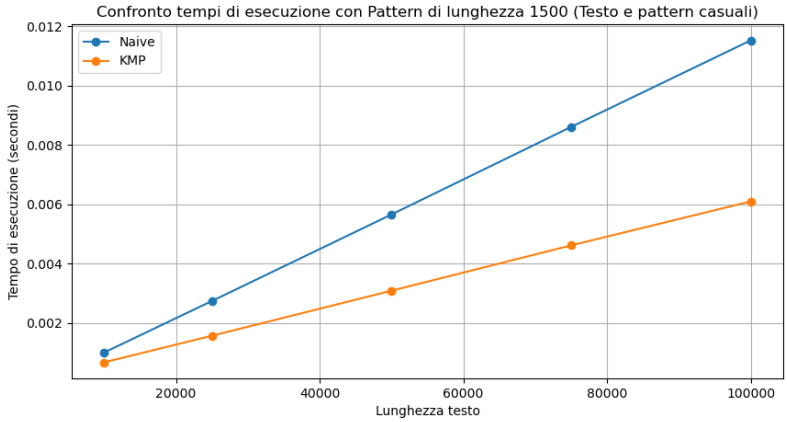
(b) Pattern di lunghezza 100

In questi grafici, i tempi di esecuzione degli algoritmi Naive e KMP vengono confrontati su pattern e testi piccoli, in particolare vengono generati testo e pattern casuali. Possiamo notare come KMP e Naive non differiscano molto in termini di velocità, ma al crescere delle lunghezze del testo c'è un leggero vantaggio di KMP.

3.2.2 Pattern di lunghezza 1000 e 1500 (medio)



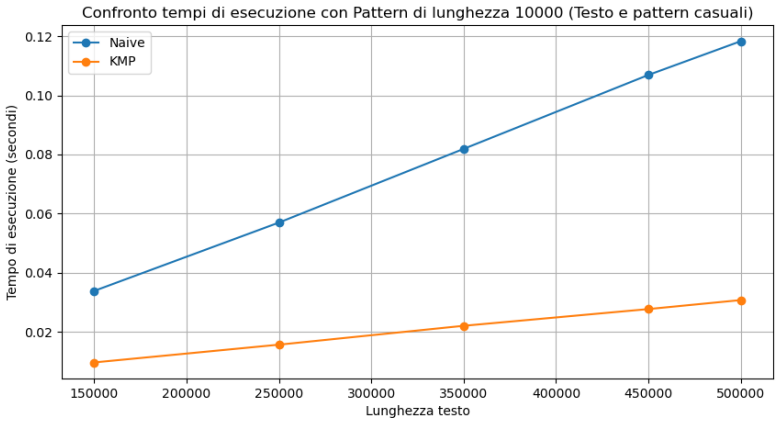
(a) Pattern di lunghezza 1000



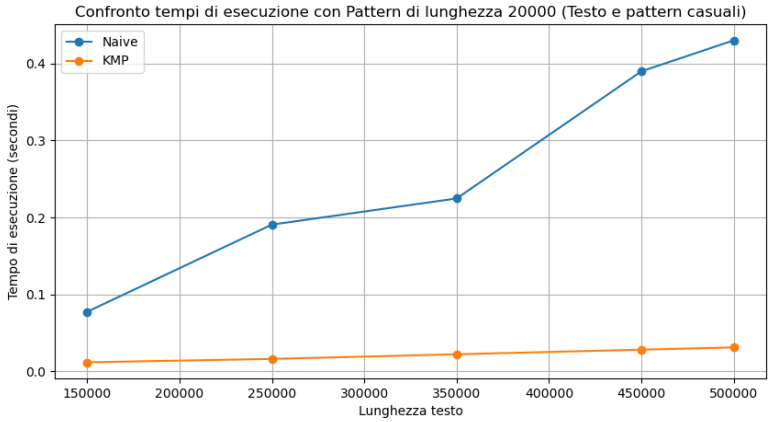
(b) Pattern di lunghezza 1500

In questi grafici, i tempi di esecuzione degli algoritmi Naive e KMP vengono confrontati su pattern e testi medi, in particolare vengono generati testo e pattern casuali. Anche qui si può notare come, all'aumentare delle lunghezze di testo, KMP si porti in vantaggio.

3.2.3 Pattern di lunghezza 10000 e 20000 (grande)



(a) Pattern di lunghezza 10000

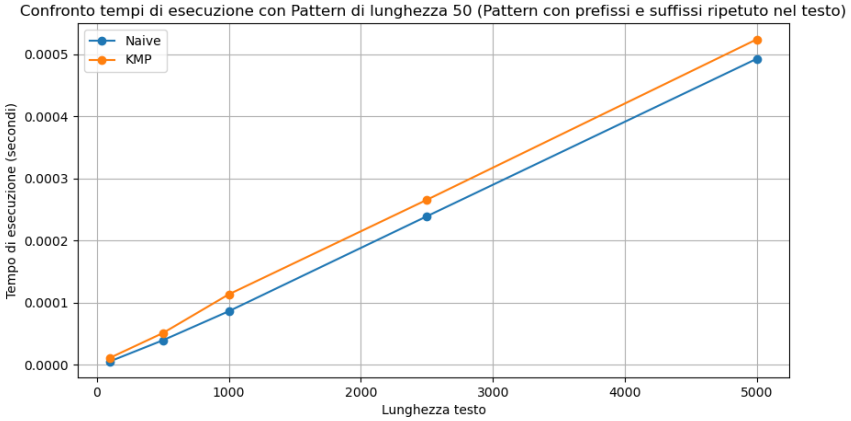


(b) Pattern di lunghezza 20000

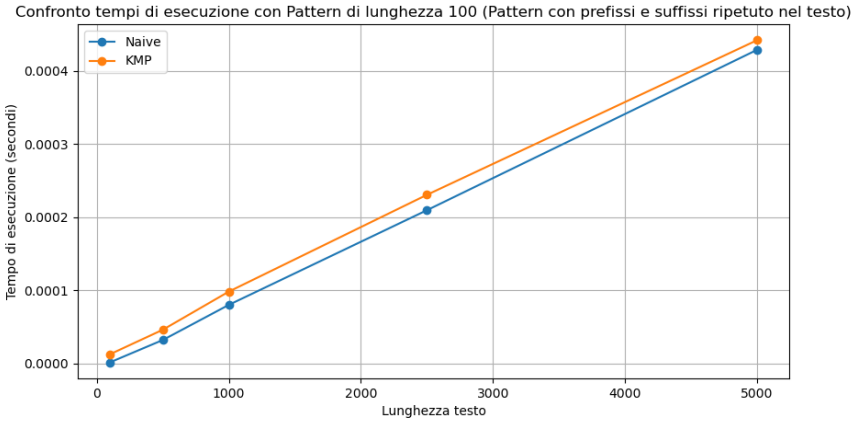
In questi grafici, i tempi di esecuzione degli algoritmi Naive e KMP vengono confrontati su pattern e testi grandi, in particolare vengono generati testo e pattern casuali. Naturalmente qui risulta ancora più accentuato il distacco tra i due tempi di esecuzione mostrando come Naive faccia sempre più "fatica" con l'incremento delle lunghezze e come KMP invece si adatti perfettamente.

3.3 Pattern con prefisso e suffisso uguali ripetuto nel testo

3.3.1 Pattern di lunghezza 50 e 100 (piccolo)



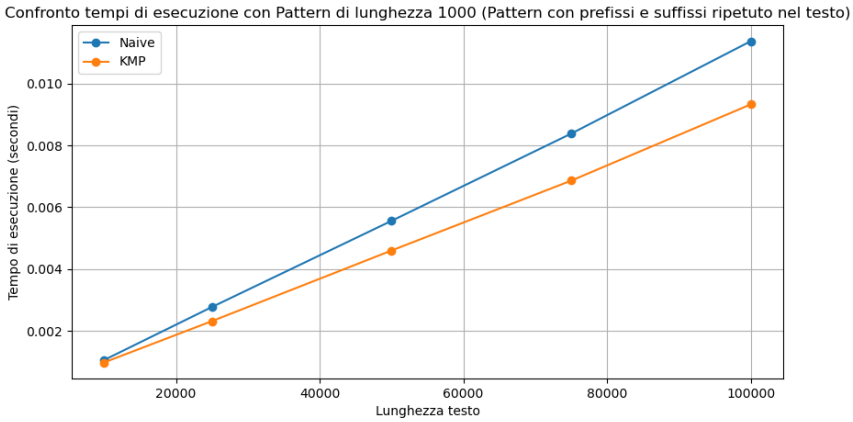
(a) Pattern di lunghezza 50



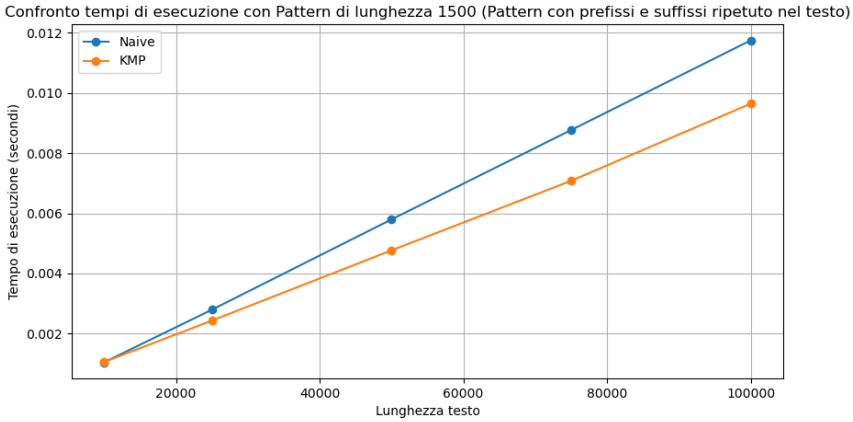
(b) Pattern di lunghezza 100

In questi grafici, i tempi di esecuzione degli algoritmi Naive e KMP vengono confrontati su pattern e testi piccoli, in particolare viene generato un pattern con prefisso e suffiso uguale che poi si ripete nel testo. KMP e Naive hanno un comportamento molto simile con un leggero vantaggio di Naive dovuto anche al fatto che KMP utilizza una parte del tempo per creare la tabella dei prefissi.

3.3.2 Pattern di lunghezza 1000 e 1500 (medio)



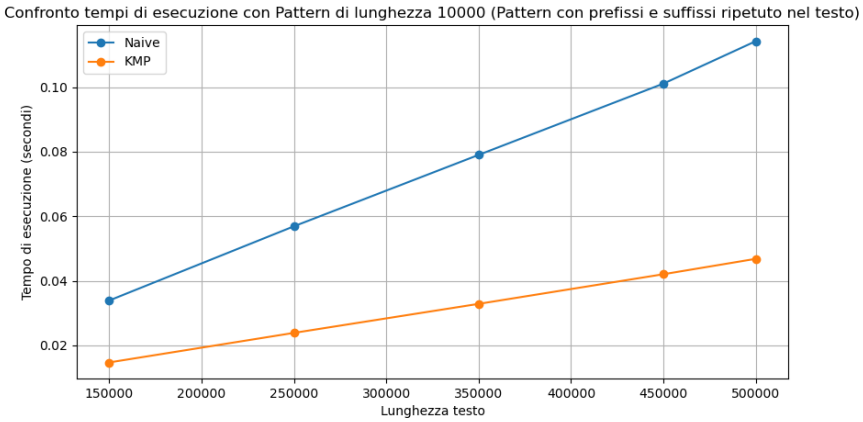
(a) Pattern di lunghezza 1000



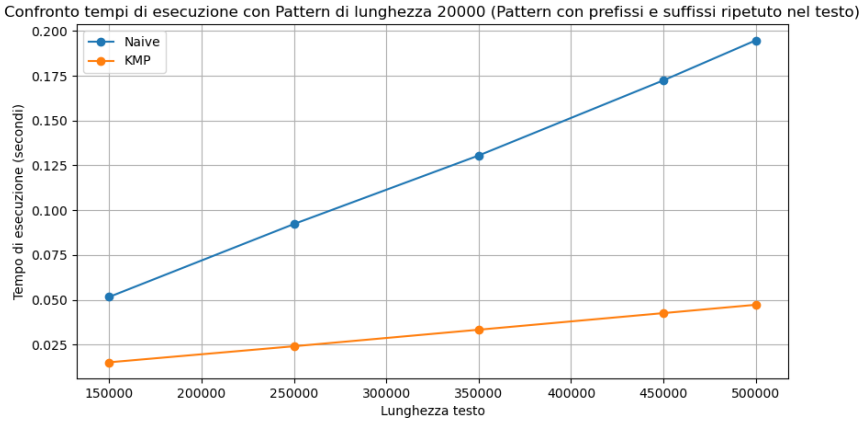
(b) Pattern di lunghezza 1500

In questi grafici, i tempi di esecuzione degli algoritmi Naive e KMP vengono confrontati su pattern e testi medi, in particolare viene generato un pattern con prefisso e suffisso uguale che poi si ripete nel testo. KMP e Naive non differiscono molto inizialmente ma, si inizia a intravedere, all'aumentare delle lunghezze, l'efficienza di KMP in questo scenario.

3.3.3 Pattern di lunghezza 10000 e 20000 (grande)



(a) Pattern di lunghezza 10000



(b) Pattern di lunghezza 20000

In questi grafici, i tempi di esecuzione degli algoritmi Naive e KMP vengono confrontati su pattern e testi grandi, in particolare viene generato un pattern con prefisso e suffiso uguale che poi si ripete nel testo. Qui si nota in modo sostanziale il vantaggio di KMP nei confronti di Naive dimostrando come la parte di memorizzazione inziale di KMP sia giustificata.

4 Conclusioni

4.1 Confronto con le aspettative

Dopo aver eseguito i test su testi e pattern generati casualmente, così come su pattern ripetuti con prefisso e suffisso uguali, i risultati dei grafici confermano ampiamente le aspettative teoriche riguardo alle prestazioni degli algoritmi KMP e Naive.

- **Prestazioni**: L'analisi mostra che l'algoritmo KMP è significativamente più efficiente rispetto al Naive, soprattutto con testi e pattern di grandi dimensioni. Nei grafici, KMP evidenzia una curva più piatta, mentre il Naive registra un notevole aumento del tempo di esecuzione all'aumentare delle dimensioni del testo e del pattern. Questo conferma la complessità $O(n + m)$ di KMP, decisamente più vantaggiosa rispetto a quella $O((n - m + 1) \cdot m)$ del Naive.
- **Efficienza e overhead di preelaborazione**: KMP si dimostra superiore in entrambi i test, sia con pattern ripetuti contenenti prefissi che coincidono con i suffissi, sia con pattern e testi casuali. In questi casi, KMP richiede meno tempo rispetto al Naive, specialmente per pattern lunghi e ripetuti, dove il Naive subisce un incremento di confronti ridondanti. Sebbene KMP necessiti di un overhead iniziale per costruire la tabella dei prefissi, questo costo è ampiamente compensato nella fase di matching, risultando trascurabile rispetto al guadagno in efficienza complessiva. Tuttavia, il Naive ha mostrato prestazioni accettabili solo su testi e pattern di piccole dimensioni.
- **Pattern ripetuti e dati casuali**: Nei test su pattern ripetuti e su dati casuali, i risultati mostrano che il Naive si avvicina alle prestazioni di KMP solo con dataset di piccole dimensioni. Quando si passa a dataset di dimensioni medie e grandi, KMP dimostra chiaramente la sua superiorità, riducendo notevolmente i tempi di esecuzione rispetto a Naive, che continua a eseguire confronti ridondanti. I risultati confermano che l'efficienza di KMP diventa particolarmente evidente con pattern e testi di dimensioni maggiori, rendendolo la scelta più vantaggiosa in questi scenari.

4.2 Riflessioni finali

Gli esperimenti condotti hanno confermato la superiorità dell'algoritmo KMP rispetto al Naive in termini di efficienza e scalabilità. Ecco alcune considerazioni chiave:

1. **Superiore efficienza di KMP**: L'algoritmo KMP ha dimostrato di gestire meglio testi e pattern di grandi dimensioni o contenenti strutture ripetitive, evidenziando la sua capacità di ridurre i confronti ridondanti.

2. **Utilizzo di Naive in scenari semplici**: Nonostante le sue limitazioni, l'algoritmo Naive rimane una scelta valida in situazioni più semplici, come con testi e pattern di piccole dimensioni o dove la semplicità di implementazione è prioritaria.

3. **Scelta in base alla complessità**: Consiglio di optare per KMP quando ci si trova a gestire grandi dataset o pattern ripetitivi. La fase di preelaborazione di KMP giustifica il suo utilizzo in tali casi.

In sintesi, la scelta dell'algoritmo dovrebbe riflettere la natura e la scala del problema: Naive per contesti più piccoli e KMP per situazioni più complesse ed esigenti in termini di efficienza.

Riferimenti bibliografici

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