# Algoritmo di Boyer–Moore

L'algoritmo di pattern matching su stringhe di Boyer–Moore è un algoritmo di pattern matching efficiente che rappresenta il principale punto di riferimento nel suo ambiente. È stato sviluppato da Robert S. Boyer e J Strother Moore nel 1977. L'algoritmo preprocessa la stringa da cercare (il pattern), ma non la stringa esaminata (il testo). È quindi particolarmente adatto per applicazioni in cui il pattern è molto più breve del testo o persiste su più ricerche. L'algoritmo di Boyer–Moore utilizza informazioni raccolte durante la fase di preprocessamento per poter saltare sezioni del testo, risultando in un fattore costante più basso rispetto a molti altri algoritmi su stringhe. In generale, l'algoritmo è eseguito più velocemente con l'aumentare della lunghezza del pattern. La caratteristica fondamentale dell'algoritmo è il confronto in coda del pattern piuttosto che in testa, e la possibilità di muoversi lungo il testo saltando interi gruppi di caratteri piuttosto che verificare ogni singolo carattere presente nel testo.

## Definizioni

- ***S[i]*** indica il carattere in posizione ***i*** in una stringa ***S***, partendo da 1;
- ***S[i..j]*** indica una sottostringa di una stringa ***S*** che inizia in posizione ***i*** e finisce in posizione ***j***, estremi inclusi;
- Un **prefisso** di ***S*** è una sottostringa ***S[1..i]*** con ***i*** appartenente all'intervallo ***[1, n]***, dove ***n*** è la lunghezza di ***S***;
- Un **suffisso** di ***S*** è una sottostringa ***S[i..n]*** con ***i*** appartenente all'intervallo ***[1, n]***, dove ***n*** è la lunghezza di ***S***;
- La stringa da cercare viene chiamata **pattern** e si indica con il simbolo ***P***;
- La stringa in cui si effettua la ricerca viene chiamata **testo** e si indica con il simbolo ***T***;
- La lunghezza di ***P*** è ***n***;
- La lunghezza di ***T*** è ***m***;
- Un **allineamento** di ***P*** in ***T*** è un indice ***k*** in ***T*** tale che l'ultimo carattere di ***P*** è in linea con l'indice ***k*** di ***T***;
- Un'**occorrenza** di ***P*** si verifica per un allineamento se ***P*** è uguale a ***T[(k-n+1)..k]***.

## Descrizione

L'algoritmo di Boyer–Moore cerca occorrenze di ***P*** in ***T*** eseguendo un confronto diretto dei caratteri per diversi allineamenti. Invece di utilizzare un metodo forza bruta di tutti gli allineamenti (che sono in numero ***m - n + 1***), Boyer–Moore sfrutta le informazioni raccolte preprocessando ***P*** per saltare più allineamenti possibile.

Prima dell'introduzione di tale algoritmo, il tipico metodo per cercare qualcosa all'interno di un testo era esaminare ogni carattere del testo con il carattere iniziale del pattern. Una volta trovato un riscontro si procedeva con un confronto tra i successivi caratteri del testo con il resto del

pattern. Se non si verificava un'occorrenza si continuava a controllare il testo carattere per carattere nel tentativo di ottenere un altro riscontro. In questo modo bisognava esaminare quasi ogni carattere del testo. L'intuizione fondamentale in questo algoritmo è che se si confronta la fine del pattern con il testo allora possono essere compiuti dei salti all'interno del testo piuttosto che confrontare ogni singolo carattere. Il motivo è che allineando il pattern al testo, l'ultimo carattere del pattern viene confrontato col carattere nel testo. Se i caratteri non corrispondono non c'è bisogno di effettuare confronti con i caratteri precedenti. Se il carattere nel testo non corrisponde ad alcun carattere del pattern, allora il prossimo carattere del testo da confrontare si trova a ***n*** caratteri di distanza lungo il testo, dove ***n*** è la lunghezza del pattern. Se il carattere nel testo è presente nel pattern, allora viene compiuto uno spostamento parziale lungo il testo per allineare i caratteri corrispondenti, dopodiché viene ripetuto l'intero processo. Lo scorrimento a salti lungo il testo per effettuare confronti piuttosto che controllare ogni singolo carattere riduce il numero di confronti da compiere, che è la chiave per l'aumento dell'efficienza dell'algoritmo.

Più formalmente, l'algoritmo incomincia con un allineamento ***k = n***, cioè l'inizio di ***P*** corrisponde all'inizio di ***T***. I caratteri in ***P*** e ***T*** vengono confrontati a partire dalla posizione ***n*** in ***P*** e ***k*** in ***T***, e spostandosi all'indietro: le stringhe vengono verificate dalla fine di ***P*** all'inizio di ***P***. Il confronto continua finché non viene raggiunto l'inizio di ***P*** (significa che si è verificata un'occorrenza) o si ha un mismatch che genera uno spostamento a destra dell'allineamento del massimo valore consentito da alcune regole. I confronti vengono nuovamente eseguiti col nuovo allineamento, e il processo si ripete finché l'allineamento non viene allineato oltre la fine di ***T***, che significa che non verranno trovate altre corrispondenze.

Le regole di spostamento sono implementate come ricerche su tabelle a tempo costante, utilizzando le tabelle generate nel preprocessamento di ***P***.

# Regole di Spostamento

---

Uno spostamento viene calcolato applicando entrambe le seguenti regole: la regola "del cattivo carattere" e la regola "del buon suffisso". L'attuale offset di spostamento è il massimo degli spostamenti calcolati da queste regole.

## Regola del Cattivo Carattere

### Descrizione

La regola del cattivo carattere prende in considerazione il carattere in ***T*** per cui il processo di confronto è fallito (assumendo che tale fallimento si sia verificato). Viene trovata la successiva occorrenza di tale carattere in ***P*** verso sinistra, e viene proposto uno spostamento tale che porti quella occorrenza in linea con la relativa occorrenza in ***T***. Se il carattere in ***T*** che ha causato il mismatch non presenta occorrenze in ***P*** verso sinistra, viene allora suggerito uno spostamento che porti interamente ***P*** oltre la posizione in cui il confronto era fallito.

### Preprocessamento

I metodi variano riguardo l'esatta forma che dovrebbe assumere la tabella per la regola del cattivo carattere, ma una semplice soluzione per una ricerca a tempo costante è la seguente: si crea una tabella bidimensionale che viene indicizzata prima con l'indice del carattere ***c*** nell'alfabeto e poi con l'indice ***i*** nel pattern. Questa ricerca restituirà l'occorrenza di ***c*** in ***P*** con il successivo più alto indice ***j < i*** oppure ***-1*** se non esiste tale occorrenza. Lo spostamento proposto sarà allora ***i - j***, con ***O(1)*** tempo di ricerca e ***O(kn)*** spazio, assumendo un alfabeto finito di dimensione ***k***.

## Regola del Buon Suffisso

### Descrizione

La regola del buon suffisso è decisamente più complessa sia come concetto che come implementazione rispetto alla regola del cattivo carattere. È il motivo per cui i confronti avvengono alla fine del pattern e non all'inizio, ed è formalmente indicato così:

> Si supponga che, per un certo allineamento di ***P*** e ***T***, una sottostringa ***t*** di ***T*** corrisponda a un suffisso di ***P***, ma si verifichi un mismatch al successivo confronto verso sinistra. Quindi si trovi, se esiste, la copia più a destra ***t'*** di ***t*** in ***P*** tale che ***t'*** non è un suffisso di ***P*** e il carattere a sinistra di ***t'*** in ***P*** è diverso dal carattere a sinistra di ***t*** in ***P***. Si sposti ***P*** verso destra così che la sottostringa ***t'*** in ***P*** si allinei con la sottostringa ***t*** in ***T***. Se ***t'*** non esiste, allora si sposti l'estremità sinistra di ***P*** oltre l'estremo sinistro di ***t*** in ***T***. Se non è possibile nessuno di questi spostamenti, allora si sposti ***P*** di ***n*** posizioni verso destra. Se viene trovata un'occorrenza di ***P***, si sposti ***P*** del minor numero di posti possibile cosicché un *corretto* prefisso del pattern ***P*** spostato corrisponda a un suffisso dell'occorrenza di ***P*** in ***T***. Se tale spostamento non è possibile, si sposti ***P*** di ***n*** posizioni, che significa spostare ***P*** oltre ***t***.

### Preprocessamento

La regola del buon suffisso richiede due tabelle: una da usare nel caso generale, e un'altra da usare o quando il caso generale non restituisce risultati utili o avviene una corrispondenza. Indichiamo queste tabelle rispettivamente con ***L*** e ***H***, e le definiamo come segue:

> Per ogni ***i***, ***L[i]*** è la posizione più grande a meno di ***n*** tale che la stringa ***P[i..n]*** corrisponda a un suffisso di ***P[1..L[i]]*** e tale che il carattere precedente tale suffisso sia diverso da ***P[i-1]***. ***L[i]*** vale zero se non esiste una posizione che soddisfi la condizione.
>
> ***H[i]*** denoti la lunghezza del più grande suffisso di ***P[i..n]*** che sia anche prefisso di ***P***, se ne esiste uno. Se non ne esistono, ***H[i]*** vale zero.

Entrambe le tabelle sono costruibili in tempo ***O(n)*** e utilizzano spazio ***O(n)***. Lo spostamento di allineamento per l'indice ***i*** in ***P*** è dato da ***n - L[i]*** oppure ***n - H[i]***. ***H*** dovrebbe essere usato soltanto se ***L[i]*** è zero o se si è verificata una corrispondenza.

## Regola di Galil

Una semplice ma importante ottimizzazione del Boyer–Moore fu avanzata da Zvi Galil nel 1979. In contrasto con lo spostamento, la regola di Galil si occupa di accelerare i confronti effettivamente compiuti per ogni allineamento saltando quelle sezioni che si sa che corrisponderanno. Supponiamo che per un allineamento $k_1$, ***P*** venga confrontato con ***T*** fino al carattere ***c*** di ***T***. Allora se ***P*** viene spostato in un $k_2$ tale che la sua estremità sinistra sia compresa tra ***c*** e $k_1$, nella prossima fase di confronto un prefisso di ***P*** deve corrispondere alla sottostringa $T[(k_2 - n)..k_1]$. Così se i confronti arrivano fino alla posizione $k_1$ of ***T***, un'occorrenza di ***P*** può essere individuata senza effettuare espressamente confronti dopo $k_1$. Oltre ad aumentare l'efficienza del Boyer–Moore, la regola di Galil è necessaria per dimostrare l'esecuzione a tempo lineare nel caso peggiore.

## Performance

L'algoritmo di Boyer–Moore per com'è stato presentato nel documento originale ha nel caso peggiore un tempo di esecuzione $O(n+m)$ solo se il pattern *non* appare all'interno del testo. Questo venne dimostrato per la prima volta da Knuth, Morris e Pratt nel 1977, seguiti da Guibas e Odlyzko nel 1980 con un upper bound di $5m$ di confronti nel caso peggiore. Richard Cole riuscì a fornire una dimostrazione con un upper bound di $3m$ di contronti nel caso peggiore nel 1991.

Quando il pattern *è* presente all'interno del testo, il tempo di esecuzione dell'algoritmo originale è $O(nm)$ nel caso peggiore. È facilmente da vedere quando sia il pattern che il testo consistono unicamente dello stesso carattere ripetuto. Tuttavia, l'inclusione della regola di Galil risulta in tempo di esecuzione lineare in tutti i casi.