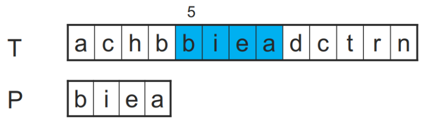
Lezione 9 15/11/2023

Introduzione al pattern matching su stringhe

Pattern matching significa cercare un motivo all’interno di un oggetto più o meno complesso. Noi ci concentriamo sul pattern matching sul testo T che sarà lungo, e cercheremo una stringa più piccola (pattern) P.



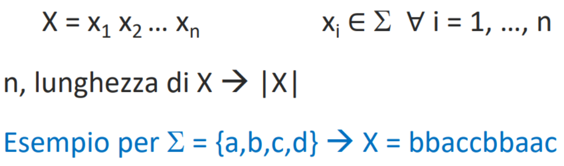
Il primo simbolo ha posizione 1, non 0.

Cercheremo anche delle sottostringhe che “assomigliano” al pattern, ma che non saranno identiche.

Stringhe

Una stringa è una giustapposizione di simboli appartenenti a un alfabeto Σ.

Una sequenza di simboli può essere anche una sequenza di interi, per esempio “1, 2, 5, 7” mentre invece nelle stringhe non c’è la virgola, è come se fosse una parola.

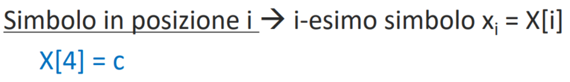


$X = x_1 x_2 \dots x_n$ $\quad x_i \in \Sigma \ \forall i = 1, \dots, n$

n, lunghezza di X → |X|

Esempio per Σ = {a,b,c,d} → X = bbaccbbaac

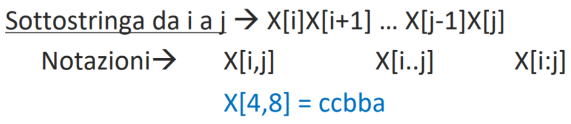
Useremo la lettera maiuscola per denotare la stringa, e le lettere minuscole per i simboli della stringa. La stringa nulla sarà rappresentata dal simbolo ϵ (epsilon). L’alfabeto di definizione può essere più ampio rispetto all’insieme stretto dei simboli che compaiono nella stringa.



Simbolo in posizione i → i-esimo simbolo x_i = X[i]

X[4] = c

Con la posizione 1-based.

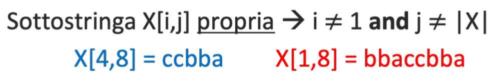


Sottostringa da i a j → X[i]X[i+1] … X[j-1]X[j]

Notazioni→ X[i,j] X[i..j] X[i:j]

X[4,8] = ccbba

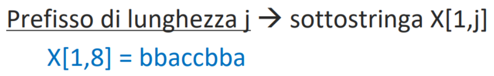
Stringa dalla posizione i alla posizione j, i e j inclusi. Useremo la notazione con la virgola.



Sottostringa X[i,j] propria → i ≠ 1 **and** j ≠ |X|

X[4,8] = ccbba X[1,8] = bbaccbba

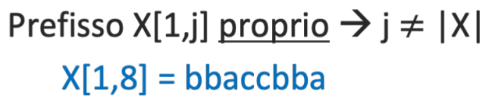
Se contemporaneamente, *i* non è la prima posizione e *j* non è l'ultima posizione. In quest'esempio quella rossa è impropria perché inizia con 1.



Prefisso di lunghezza j → sottostringa X[1,j]

X[1,8] = bbaccbba

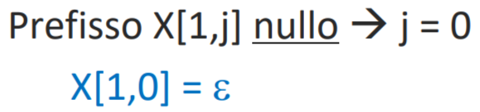
è una sottostringa impropria che parte dall'inizio della stringa (posizione 1).



Prefisso X[1,j] proprio → j ≠ |X|

X[1,8] = bbaccbba

Il prefisso è improprio quando la posizione j è l'ultima della stringa. Un prefisso è proprio quando non è lungo quanto la stringa originale.

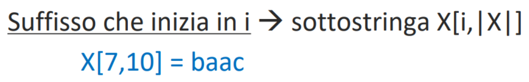


Prefisso X[1,j] nullo → j = 0

X[1,0] = ε

Se il prefisso finisce con 0, ovvero con una posizione precedente, è nullo.

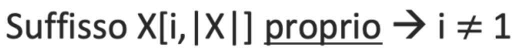
Definiamo il suffisso come quello che inizia in una certa posizione *i* e finisce nell'ultima posizione.



Suffisso che inizia in i → sottostringa X[i,|X|]

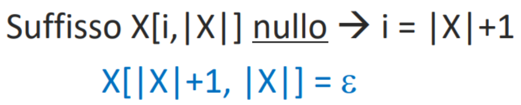
X[7,10] = baac

L'esempio in blu è anche un suffisso proprio perché non inizia dalla posizione 1.



Suffisso X[i,|X|] proprio → i ≠ 1

Suffisso nullo se l'ultima posizione è più grande della lunghezza della stringa, e quindi non esiste.



Suffisso X[i,|X|] nullo → i = |X|+1

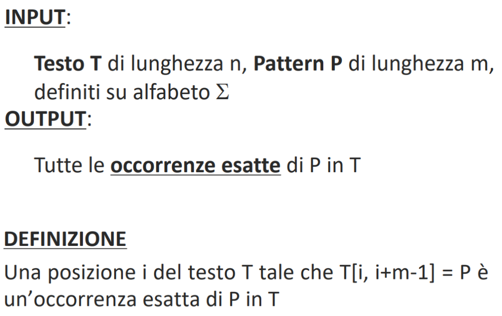
X[|X|+1, |X|] = ε

String matching

Dati un pattern P e un testo T:

- Lo **string matching esatto** è la ricerca delle occorrenze esatte di P in T;
- Lo **string matching approssimato** ricerchiamo le occorrenze approssimate di P in T e quindi ammettiamo delle differenze. Saremo noi a decidere quanto l'occorrenza del pattern del testo possa differire dal pattern.

String matching esatto



INPUT:

Testo T di lunghezza n, **Pattern P** di lunghezza m, definiti su alfabeto Σ

OUTPUT:

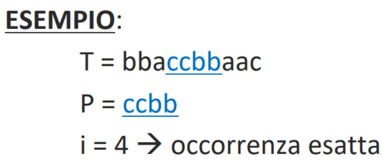
Tutte le **occorrenze esatte** di P in T

DEFINIZIONE

Una posizione i del testo T tale che T[i, i+m-1] = P è un'occorrenza esatta di P in T

Useremo sempre la *n* per denotare la cardinalità del testo T, mentre useremo *m* per la cardinalità del pattern.

Ci basta solo il primo indice per definire l'occorrenza esatta, che è quindi una posizione del testo dove inizia la sottostringa che è uguale al pattern.



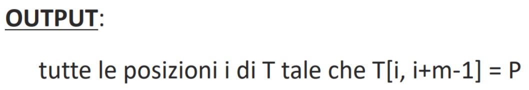
ESEMPIO:

T = bbaccbbaac

P = ccbb

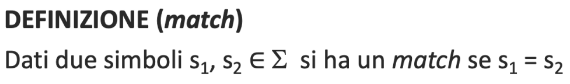
i = 4 → occorrenza esatta

Riformuliamo quindi l'output dello string matching come:



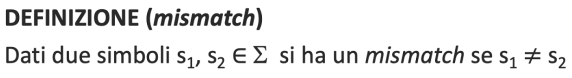
OUTPUT:

tutte le posizioni i di T tale che T[i, i+m-1] = P



DEFINIZIONE (*match*)

Dati due simboli s_1, $s_2 \in \Sigma$ si ha un *match* se $s_1 = s_2$



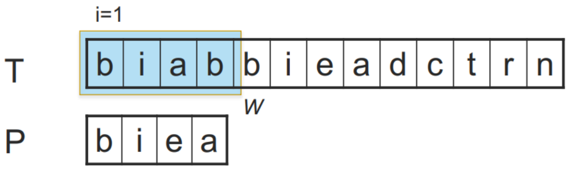
DEFINIZIONE (*mismatch*)

Dati due simboli s_1, $s_2 \in \Sigma$ si ha un *mismatch* se $s_1 \neq s_2$

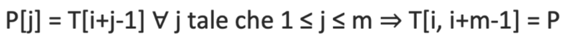
Si riferiscono ad una coppia di simboli.

Algoritmo banale per string matching esatto:

1. Uso una **finestra W** lunga m che scorre lungo T da sinistra a destra partendo dalla posizione i=1.
2. Si confronta ogni simbolo di P con il corrispondente simbolo di T all'interno di W da sinistra verso destra per vedere se c'è un match o un mismatch.
3. W viene spostata di una posizione verso destra e il confronto viene ripetuto

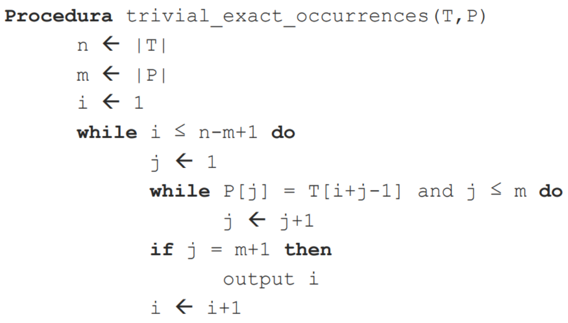


Questa è la formalizzazione di un'occorrenza esatta:



P[j] = T[i+j-1] ∀ j tale che 1 ≤ j ≤ m ⇒ T[i, i+m-1] = P

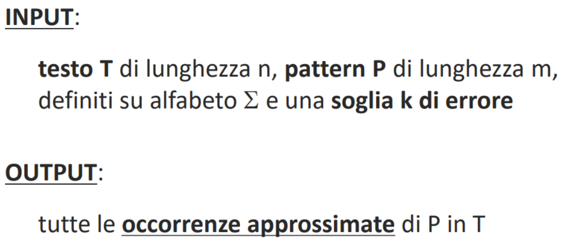
L'ultima posizione di W è: $i = |T| - |P| + 1 = n - m + 1$



```
Procedura trivial_exact_occurrences(T,P)
    n ← |T|
    m ← |P|
    i ← 1
    while i ≤ n-m+1 do
        j ← 1
        while P[j] = T[i+j-1] and j ≤ m do
            j ← j+1
        if j = m+1 then
            output i
        i ← i+1
```

La complessità sarà quindi $O(nm)$.

String matching approssimato



INPUT:

testo T di lunghezza n, **pattern P** di lunghezza m, definiti su alfabeto Σ e una **soglia k di errore**

OUTPUT:

tutte le **occorrenze approssimate** di P in T

Abbiamo un parametro in più: *k* che è la soglia di errore.

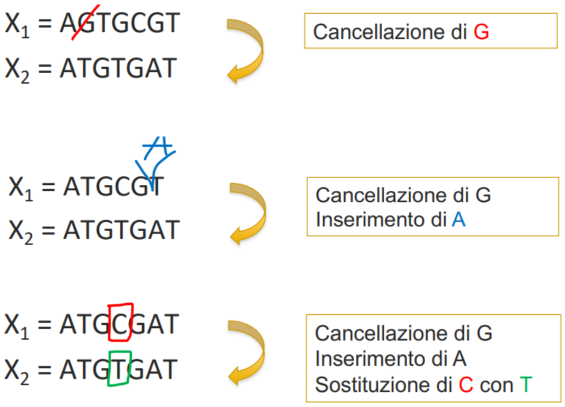
Misura dell'errore

Usiamo la distanza di Edit (Edit distance) tra due stringhe, che è il minimo numero di operazioni di:

- sostituzione
- cancellazione
- inserimento

di un simbolo che trasformano una stringa nell'altra.

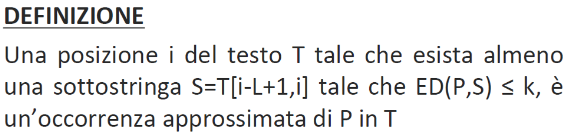
Nell'esempio a destra $ED = 3$



Ma possiamo fare di meglio? In questo caso no ma potremmo anche fare il procedimento opposto modificando X_2 per trasformala in X_1.

$ED(X1, X2) \geq abs(|X1| - |X2|)$ La distanza di edit è quantomeno il valore assoluto della sottrazione delle lunghezze delle due stringhe.

Occorrenza approssimata

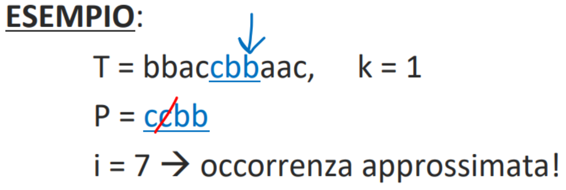


DEFINIZIONE

Una posizione i del testo T tale che esista almeno una sottostringa S=T[i-L+1,i] tale che ED(P,S) ≤ k, è un'occorrenza approssimata di P in T

è sempre una posizione i nel testo, ma qui deve esistere almeno una sottostringa s che inizia *i-L+1* e che finisce in *i*, tale che la distanza di edit tra il pattern e la sottostringa sia minore uguale a *k*, ovvero la soglia di errore. *i* è quindi la posizione finale.

Noi ammettiamo delle differenze che possono essere inserimenti o cancellazioni, quindi la lunghezza L della sottostringa che matcha il pattern può avere una lunghezza diversa dal pattern.

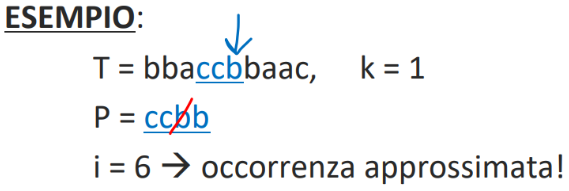


ESEMPIO:

T = bbaccbbaac,   k = 1

P = c~~c~~bb

i = 7 → occorrenza approssimata!

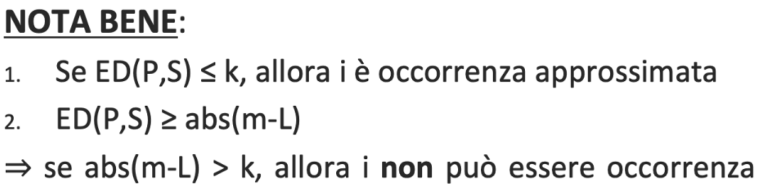


ESEMPIO:

T = bbaccbbaac,   k = 1

P = cc~~b~~b

i = 6 → occorrenza approssimata!

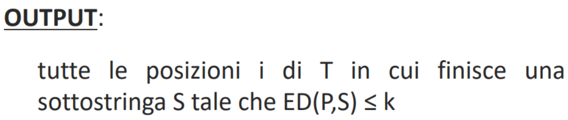


NOTA BENE:

1. Se ED(P,S) ≤ k, allora i è occorrenza approssimata
2. ED(P,S) ≥ abs(m-L)

⇒ se abs(m-L) > k, allora i **non** può essere occorrenza

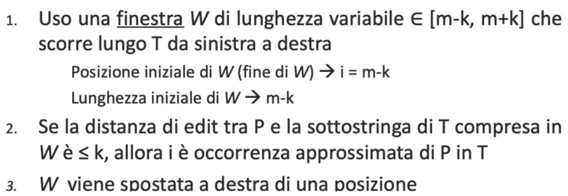
Riformuliamo l'output dello string matching approssimato:



OUTPUT:

tutte le posizioni i di T in cui finisce una sottostringa S tale che ED(P,S) ≤ k

Algoritmo banale per string matching approssimato:

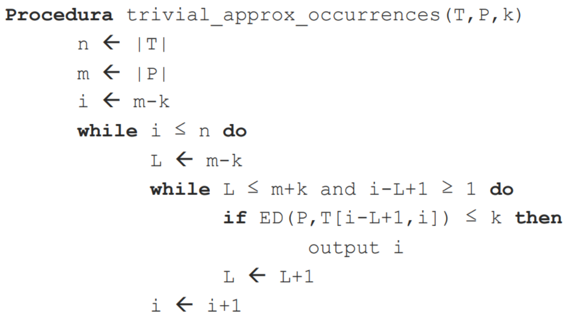


1. Uso una finestra *W* di lunghezza variabile ∈ [m-k, m+k] che scorre lungo T da sinistra a destra
    Posizione iniziale di *W* (fine di *W*) → i = m-k
    Lunghezza iniziale di *W* → m-k
2. Se la distanza di edit tra P e la sottostringa di T compresa in *W* è ≤ k, allora i è occorrenza approssimata di P in T
3. *W* viene spostata a destra di una posizione

La finestra non ha ampiezza m fissa come nello string matching esatto. L'ampiezza varia tra m-k e m+k.

Partiamo quindi con l'ampiezza minima, puntata ad *i* che è l'ultimo simbolo della stringa.

Possiamo ampliare la finestra a sinistra e calcoliamo quindi la distanza di edit.



```
Procedura trivial_approx_occurrences(T,P,k)
    n ← |T|
    m ← |P|
    i ← m-k
    while i ≤ n do
        L ← m-k
        while L ≤ m+k and i-L+1 ≥ 1 do
            if ED(P,T[i-L+1,i]) ≤ k then
                output i
            L ← L+1
        i ← i+1
```

Il primo while sposta la finestra, limitato superiormente dalla lunghezza *n* della stringa.

Parto dal minimo L e poi lo amplio verso sinistra nel secondo while.

Calcolo la distanza di edit, se è minore uguale a k allora questa è un'occorrenza approssimata.