Lezione 15 29/11/2023

Ricerca esatta con BYG

Il confronto tra simboli del pattern e del testo non è esplicito

Vengono effettuate in parallelo operazioni bit a bit su *word* di bit (algoritmo *bit parallel*)

Segue il paradigma SHIFT-AND, cioè compie fondamentalmente le due operazioni :

- shift dei bit
- AND logico tra bit

Svolge azioni bit a bit in parallelo, non in maniera sequenziale. Questo algoritmo segue il paradigma SHIFT-AND perchè compie due operazioni: uno shift di queste parole di bit e l'and logico bit a bit.

1. Preprocessing del pattern P in tempo $\theta(|\Sigma|+m)$ → calcolo di $|\Sigma|$ *words* di m bit
2. Scansione del testo T in tempo $\theta(n)$ per cercare le occorrenze esatte di P

Il preprocessing prende il pattern e calcola delle parole lunghe come il pattern.

La seconda fase è la scansione del testo, dove vengono trovate le occorrenze di P nel testo.

Word di bit

Word di bit: gruppo di bit trattato come un'unità la cui dimensione può variare e che rappresenta un valore di un certo tipo (numero, un carattere o altro).

Ad esempio,
11010011
è una word di 8 bit

Bit più a destra = Bit meno significativo (posizione 8)
Bit più a sinistra = Bit più significativo (posizione 1)

Sono gruppi di bit trattati come un'unità. Non è una stringa di bit ma un gruppo di bit che rappresenta un dato di un certo tipo (intero, decimale, …).

Quando si implementa quest'algoritmo bisogna fare in modo che usi parole di bit e non stringhe di bit, perché sarebbe un errore perché non sfrutterebbe le potenzialità dell'algoritmo, sarebbe meno efficiente.

Diremo che il bit meno significativo è quello di posizione maggiore (posizione 8 in questo caso) mentre quello più significativo è quello nella posizione più

bassa. Teniamo quindi la stessa indicizzazione che con le stringhe.

Operazioni bit a bit su words di bit

OPERAZIONE BIT A BIT SU *WORD* DI BIT

Esecuzione di un'operazione tra i bit corrispondenti di due o più *words* di bit della stessa lunghezza.

Valore restituito → nuova *word* in cui ogni bit è il risultato dell'operazione tra i bit corrispondenti nelle *words* in input.

Un'operazione bit a bit è l'esecuzione di un'operazione tra i bit corrispondenti di due word della stessa lunghezza.

1. Congiunzione logica → **AND**
   C, Java → operatore `&`
2. Disgiunzione logica (inclusiva) → **OR**
   C, Java → operatore `|`
3. Shift dei bit di una posizione a destra con bit più significativo a 0 → **RSHIFT**
   C, Java → operatore `>>1`
4. Shift dei bit di una posizione a destra con bit più significativo a 1 → **RSHIFT1**
   RSHIFT seguito da **OR**

AND logico

$w = w_1$ AND w_2

restituisce una *word* w tale che:

- $w[j]=1 \Leftrightarrow w_1[j]=1$ and $w_2[j]=1$
- $w[j]=0$, altrimenti

```
0010 = 1010 AND 0110
```

Il j-esimo bit della parola in output è 1 solo se il j-esimo bit di entrambe parole è 1, altrimenti è 0.

L'operazione è fatta in parallelo su ogni bit.

OR logico (inclusivo)

$w = w_1$ OR w_2

restituisce una *word* w tale che:

- $w[j]=1 \Leftrightarrow w_1[j]=1$ oppure $w_2[j]=1$
- $w[j]=0$, altrimenti

```
1110 = 1010 OR 0110
```

Uguale a prima ma come or.

Shift di una posizione a destra con bit più significativo uguale a 0

$w = \text{RSHIFT}(w_1)$

restituisce una *word* w tale che:

- $w[1]=0$
- $w[j]=w_1[j-1]$, se $j \geq 2$

$0101 = \text{RSHIFT}(1010)$

Il risultato ha il primo bit a 0, dal secondo in poi invece il b j-esimo è preso dal bit (j-1)-esimo della parola originale.

Shift di una posizione a destra con bit più significativo uguale a 1

$w = \text{RSHIFT1}(w_1)$

restituisce una *word* w tale che:

- $w[1]=1$
- $w[j]=w_1[j-1]$, se $j \geq 2$

$1101 = \text{RSHIFT1}(1010)$

$1101 = \text{RSHIFT}(1010)\ \text{OR}\ 1000$

Uguale a prima ma con il primo bit a 1.

Questo si può fare anche con un RSHIFT a 0, messo in OR con una maschera che ha il primo bit a 1.

Word B_σ (preprocessing, fase 1)

Dato un pattern di lunghezza m e σ in Σ, B_σ è una *word* di m bit tale che:

$B_\sigma[j] = 1 \Leftrightarrow P[j] = \sigma$

B → Tabella delle *word* B_σ (una *word* per ogni simbolo in Σ)

Il preprocessing consiste nel calcolare delle parole che chiamiamo B sigma. Si fa una parola per ogni simbolo dell'alfabeto.

Il bit j-esmio è 1 sse il simbolo del pattern in posizione j è uguale a sigma.

Esempio tabella B

P

a	b	c	a	b	a

Σ={a,b,c,d}

B

B_a	100101
B_b	010010
B_c	001000
B_d	000000

P

a	b	c	a	b	a

Σ={a,b,c,d}

B

B_a	100101
B_b	010010
B_c	001000
B_d	000000

DOMANDA: il simbolo in posizione j di P è uguale a un certo simbolo σ?

RISPOSTA: YES se $B_\sigma[j]$ [illegible], altrimenti NO

Il preprocessing ci serve a capire proprio questo, se in una posizione del pattern c'è un simbolo sigma.

L'1 è associato al concetto di vero quindi lo cancello.

Calcolo della tabella B

STEP1: tutte le *words* B_σ vengono inizializzate a m bit a 0

STEP2: viene creata una maschera M di m bit tutti uguali a 0 tranne il più significativo che è uguale a 1

STEP3: si esegue una scansione di P da sinistra a destra, e per ogni posizione j vengono eseguite le due operazioni bit a bit

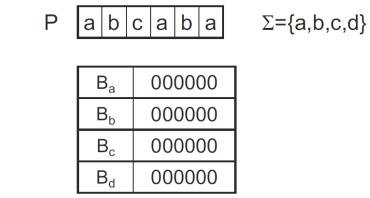
$B_{P[j]}$ = M **OR** $B_{P[j]}$
M = **RSHIFT**(M)

(più significativo → a sinistra)

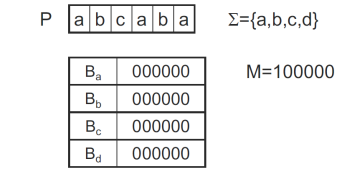
STEP3: prendiamo la maschera e la mettiamo in OR logico alla parola del simbolo che abbiamo letto nella posizione P[j]. Successivamente facciamo un RSHIFT della maschera M e ripetiamo.

Esempio

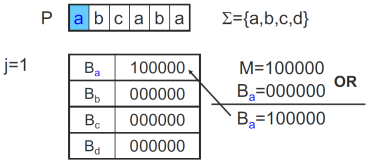
STEP1 → Inizializzazione delle *words* B_σ



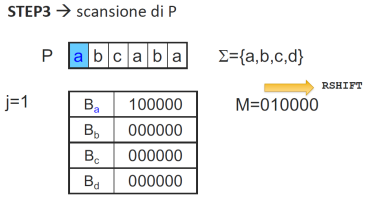
STEP2 → Creazione della maschera M



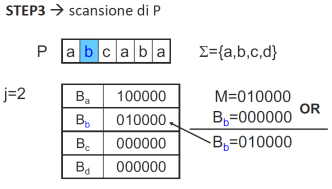
STEP3 → scansione di P



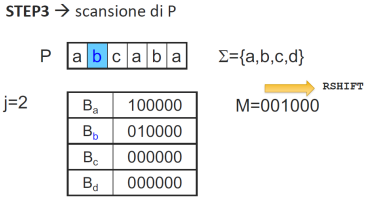
STEP3 → scansione di P



STEP3 → scansione di P



STEP3 → scansione di P



Algoritmo

```
Procedura Compute-table-B(P)
 begin
    m ← |P|
    B ← empty table of |Σ| words Bσ
    foreach σ in Σ do
      Bσ ← 00…0
    M ← 10…0
    for j ← 1 to m do
      σ ← P[j]
      Bσ ← M OR Bσ
      M = RSHIFT(M)
    return B
 end
```

Tempo → θ(|Σ|+m)

Scansione del testo (fase 2)

1. Il testo T viene scandito dalla prima all'ultima posizione
2. Per ogni posizione i del testo T viene calcolata una *word* D_i di m bit
3. Ogni volta che in D_i il bit meno significativo è uguale a 1, allora i-m+1 è occorrenza esatta di P in T

j scorre il pattern, i scorre il testo.

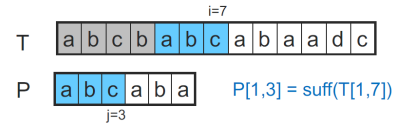
Per ogni posizione del testo mi calcolo una parola D_i lunga come il testo.

Ogni volta che la parola D_i ha il bit meno significativo (più a destra) uguale a 1, allora quello è un match esatto del pattern.

P[1,j] = suff(T[1,i])
significa
"P[1,j] è uguale a un suffisso di T[1,i]"

Introduciamo la notazione di suff:

Il prefisso lungo 1,j del pattern ha un'occorrenza esatta nel testo che finisce in posizione i.



Dati P lungo m e T lungo n, D_i ($0 \le i \le n$) è una *word* di m bit tale che:

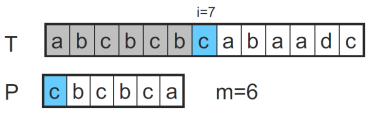
$D_i[j] = 1 \Leftrightarrow$ P[1,j] = suff(T[1,i])

Comprendiamo l'indice 0, quindi per un testo lungo n avremo n+1 parole.

Il bit j-esimo della parola D_i è 1 sse il prefisso lungo j del pattern è suffisso del prefisso lungo i del testo.

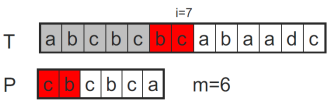
Esempio

D_7 = 1<u>0</u>1010



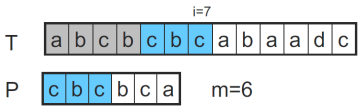
Il primo bit è a 1 perchè il prefisso lungo 1 del pattern occorre come suffisso del prefisso lungo 7 del testo.

D_7 = 10<u>1</u>010

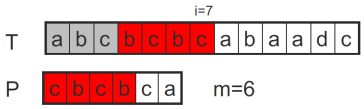


Il secondo bit è 0 perché il prefisso lungo 2 del pattern non occorre come suffisso del prefisso lungo 7 del testo.

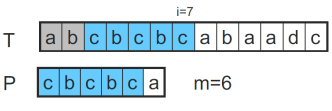
D_7 = 101<u>0</u>10



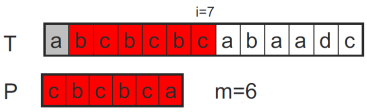
D_7 = 1010<u>1</u>0



D_7 = 10101<u>0</u>



D_7 = 10101<u>0</u>



Dati P lungo m e T lungo n, D_i ($0 \le i \le n$) è una *word* di m bit tale che:

$D_i[j] = 1 \Leftrightarrow P[1,j] = suff(T[1,i])$

D_0 = 000...0 per definizione. Infatti:

$P[1,j] \neq suff(T[1,0])\ \forall\ j \quad \Rightarrow \quad D_0[j] = 0 \quad \forall\ j$

Inoltre:

$D_i[m] = 1 \Leftrightarrow P[1,m] = T[i-m+1, i]$

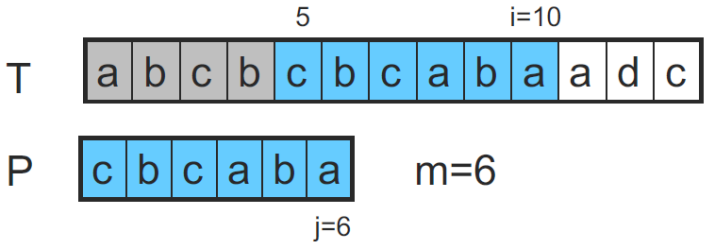
La parola D_0 si riferisce alla posizione i=0 del testo, ma con i=0 mi sto riferendo ad un prefisso nullo del testo, quindi qualsiasi prefisso del pattern non occorrera mai come suffisso di un prefisso nullo. Quindi è tutto 0.

Inoltre per la generica parola, se il bit meno significativo (più a destra) a 1, allora tutto il pattern occorre esattamente come suffisso del prefisso del testo che finisce in i.

$D_i[m] = 1 \Leftrightarrow$ i-m+1 è occorrenza esatta di P in T

Esempio

D_{10} = 000001→ i-m+1 = 5 è occorrenza esatta



Lo step di scansione del testo T:

- inizia dalla *word* D_0 = 00...0
- per ogni i da 1 a n calcola la *word* D_i a partire dalla *word* D_{i-1}
- ogni volta che D_i ha il bit meno significativo uguale a 1, viene prodotta in output l'occorrenza i-m+1

QUESTIONE: calcolare D_i a partire da D_{i-1}

Si inizia dalla parola D_0 e la si mette tutta a 0.

Si fa partire la scansione, per ogni posizione di i da 1 a n, calcoli la parola D_i usando la parola che abbiamo calcolato alla posizione i-1.

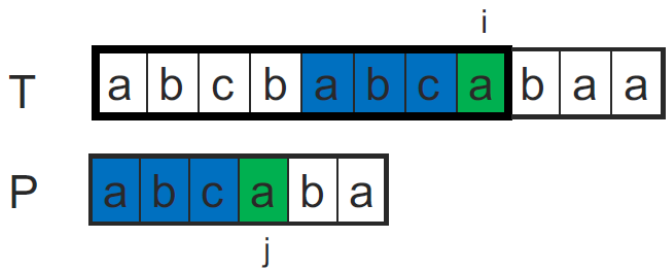
...

Calcolo di $D_i[j]$ con j>1

Per definizione

$D_i[j] = 1 \Leftrightarrow P[1,j] = suff(T[1,i])$

$D_i[j] = 1 \Leftrightarrow P[1,j] = suff(T[1,i]) \Leftrightarrow P[1,j-1] = suff(T[1,i-1])$ **AND** $P[j] = T[i]$



Scomponiamo in un AND logico.

La prima parte dice che è vera se il prefisso lungo j-1 occorre come suffisso del prefisso lungo i-1 del testo.

La seconda parte considera l'ultima posizione di entrambe.

$D_i[j] = 1 \Leftrightarrow D_{i-1}[j-1] = 1$ **AND** $B_{T[i]}[j] = 1$

Sostituiamo il primo pezzo applicando la definizione di parola D_i, usiamo quindi la parola precedente.

Il secondo pezzo lo sostituisco usando le parole della tabella B del preprocessing, con il simbolo sigma dato da T[i] in posizione j.

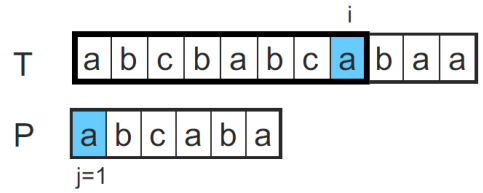
$D_i[j] = D_{i-1}[j-1]$ **AND** $B_{T[i]}[j]$

Dato che i bit hanno il concetto di vero e falso, togliamo gli 1 e la lasciamo come formula logica.

Calcolo di $D_i[j]$ con j=1

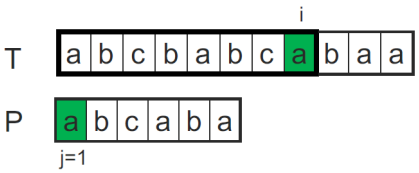
$D_i[1] = 1 \Leftrightarrow P[1,1] = suff(T[1,i])$

Come prima, applico la definizione.



$D_i[1] = 1 \Leftrightarrow P[1] = T[i]$

Qui non possiamo scomporlo quindi scriviamo così.



$D_i[1] = B_{T[i]}[1]$

Come prima recuperiamo le parole della fase di preprocessing. Come prima togliamo l'=1 e trasformiamo il sse in =.

$D_i[1] = 1$ **AND** $B_{T[i]}[1]$

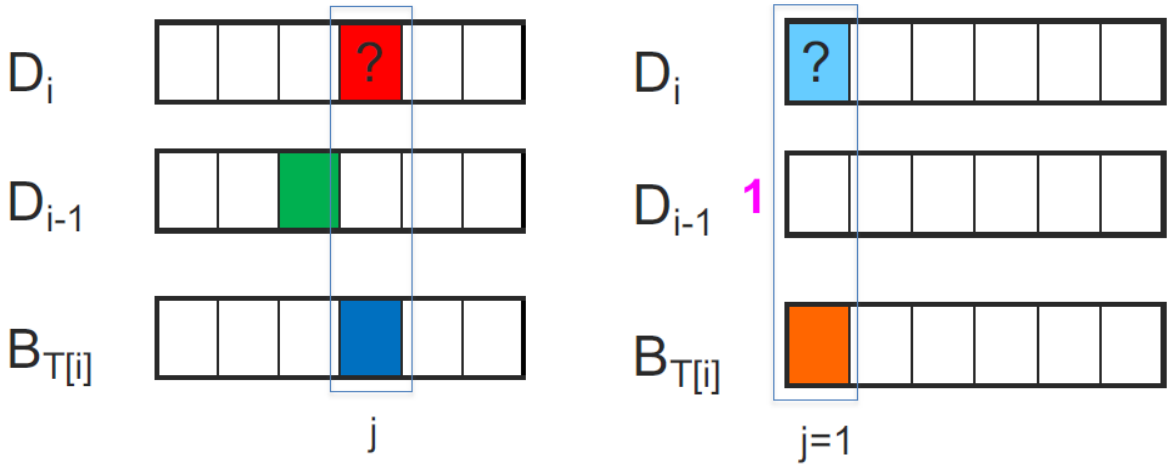
Calcolo di $D_i[j]$ con j>1

Tornando al caso j>1

Calcolo di $D_i[j]$ con j=1

Mentre nel caseo j=1

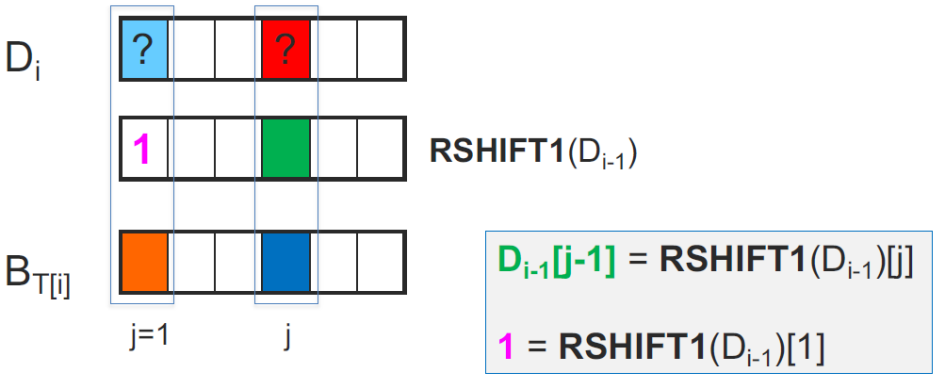
$D_i[j] = D_{i-1}[j-1]$ AND $B_{T[i]}[j]$ $D_i[1] = 1$ AND $B_{T[i]}[1]$



Per il calcolo utilizzo quindi la parola D precedente e la parola B del preprocessing.

Quindi il bit verde e l'1 sono disallineati.

$D_i[1] = 1$ AND $B_{T[i]}[1]$ $D_i[j] = D_{i-1}[j-1]$ AND $B_{T[i]}[j]$



Quindi prendiamo la parola di mezzo e facciamo un RSHIFT1.

In questo modo gli indici sono allineati.

Calcolo di $D_i[j]$ con j≥0

$$D_i[j] = \textbf{RSHIFT1}(D_{i-1})[j] \textbf{ AND } B_{T[i]}[j]$$

Quindi possiamo unire i due casi in questo modo.

$$D_i = \textbf{RSHIFT1}(D_{i-1}) \textbf{ AND } B_{T[i]}$$

Ma siccome lavoriamo su word di bit in parallelo, cancelliamo gli indici.

Scansione del testo definitiva

La fase di scansione di T:

- Inizializza la *word* $D_0 = 00...0$
- Per i compreso tra 1 e n, calcola la *word* D_i
  $D_i = \textbf{RSHIFT1}(D_{i-1}) \textbf{ AND } B_{T[i]}$
- Ogni volta che D_i ha il bit meno significativo uguale a 1, viene prodotta in output l'occorrenza esatta i-m+1

Calcoliamo tutte le D_i in questo modo.

```
Procedura BYG(B,T)
begin
    n ← |T|
    D ← 00…00
    M ← 00…01
    for i ← 1 to n do
      σ ← T[i]
      D ← RSHIFT1(D) AND Bσ
      if (D AND M) = 00…01 then
        output i-m+1
end
```

Tempo → θ(n)

Per vedere se l'ultimo bit è 1 (per controllar eoccorrenza del pattern) facciamo un AND con una maschera che ha 0…01.