Lezione 14 24/11/2023

Ricerca esatta con KMP

1. Preprocessing del pattern P di lunghezza m

   Calcolo in tempo O(m) della *prefix function* φ (funzione di fallimento)

2. Scansione del testo T di lunghezza n in tempo O(n) per cercare tutte le occorrenze esatte di P

Prefix function ϕ

Prefix Function del pattern P di lunghezza m:

$$\varphi : \{0,1,\dots,m\} \rightarrow \{-1, 0,1,\dots,m\}$$

$$\varphi(j) = |B(P[1, j])| \qquad \text{se } 1 \le j \le m$$

$$\varphi(j) = -1 \qquad \text{se } j = 0$$

Esempio

P		a	b	c	a	b	a	a	b	c	a	b	a	b
j	0	1	2	3	4	5	6	7	8	9	10	11	12	13
φ	-1	0												

m=13

$\varphi(1) = |B(P[1,1])| = 0$

P		a	b	c	a	b	a	a	b	c	a	b	a	b
j	0	1	2	3	4	5	6	7	8	9	10	11	12	13
φ	-1	0	0											

$\varphi(2) = |B(P[1,2])| = 0$

P		a	b	c	a	b	a	a	b	c	a	b	a	b
j	0	1	2	3	4	5	6	7	8	9	10	11	12	13
φ	-1	0	0	0										

m=13

$\varphi(3) = |B(P[1,3])| = 0$

P		a	b	c	a	b	a	a	b	c	a	b	a	b
j	0	1	2	3	4	5	6	7	8	9	10	11	12	13
φ	-1	0	0	0	1									

$\varphi(4) = |B(P[1,4])| = 1$

P		a	b	c	a	b	a	a	b	c	a	b	a	b
j	0	1	2	3	4	5	6	7	8	9	10	11	12	13
φ	-1	0	0	0	1	2								

$\varphi(5) = |B(P[1,5])| = 2$

P		a	b	c	a	b	a	a	b	c	a	b	a	b
j	0	1	2	3	4	5	6	7	8	9	10	11	12	13
φ	-1	0	0	0	1	2	1							

$\varphi(6) = |B(P[1,6])| = 1$

P		a	b	c	a	b	a	a	b	c	a	b	a	b
j	0	1	2	3	4	5	6	7	8	9	10	11	12	13
φ	-1	0	0	0	1	2	1	1						

φ(7) =|B(P[1,7])| = 1

P		a	b	c	a	b	a	a	b	c	a	b	a	b
j	0	1	2	3	4	5	6	7	8	9	10	11	12	13
φ	-1	0	0	0	1	2	1	1	2					

φ(8) =|B(P[1,8])| = 2

P		a	b	c	a	b	a	a	b	c	a	b	a	b
j	0	1	2	3	4	5	6	7	8	9	10	11	12	13
φ	-1	0	0	0	1	2	1	1	2	3	4	5	6	2

m=13

φ(13) =|B(P[1,13])| = 2

Algoritmo banale

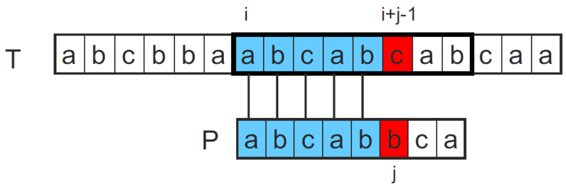
1. Viene usata una finestra *W* lunga m che scorre lungo T da sinistra a destra con posizione iniziale i = 1
2. Si confrontano i simboli di P con i corrispondenti simboli di T all'interno di *W* andando da sinistra a destra e partendo dal primo simbolo di P
3. Non appena si incontra un *mismatch* oppure ogni simbolo di P ha un *match* con il corrispondente simbolo in *W* (i è occorrenza esatta), *W* viene spostata di una posizione verso destra e il confronto viene ripetuto
4. Ultima posizione di *W* → i = |T| - |P| + 1 = n – m + 1

Algoritmo KMP

1. Viene usata una finestra *W* lunga m che scorre lungo T da sinistra a destra con posizione iniziale i = 1
2. Si confrontano i simboli di P con i corrispondenti simboli di T all'interno di *W* andando da sinistra a destra e partendo dal primo simbolo di P
3. Non appena si incontra un *mismatch* oppure ogni simbolo di P ha un *match* con il corrispondente simbolo in *W* (i è occorrenza esatta), *W* viene spostata di una posizione verso destra e il confronto viene ripetuto
4. Ultima posizione di *W* → i = |T| - |P| + 1 = n – m + 1

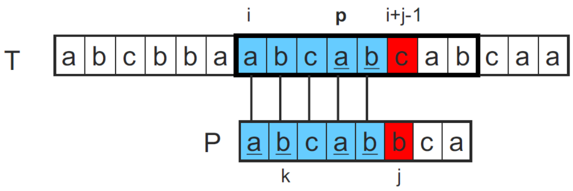
Spostamento di W (j>1)

- i, posizione di *W*
- j, posizione di *mismatch* su P
- P[1,j-1], prefisso di *match*
- i+j-1, posizione di *mismatch* su T



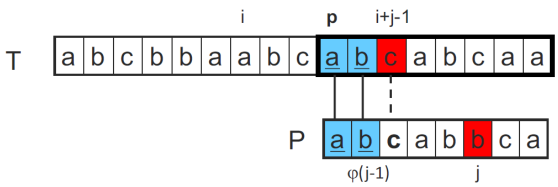
k = φ(j-1) → lunghezza del bordo del prefisso P[1,j-1]

p = i + j - k - 1 → occorrenza di P[1,k]



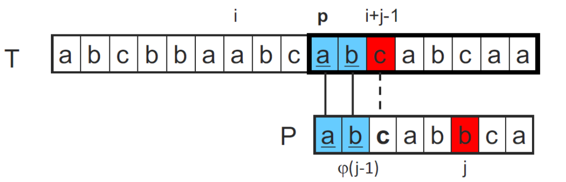
k = φ(j-1) → lunghezza del bordo del prefisso P[1,j-1]

p = i + j - φ(j-1) - 1 → nuova posizione di *W*



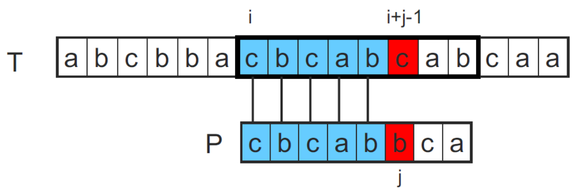
Riassumendo…

- *W* viene spostata dalla posizione i alla posizione p = i + j - φ(j-1) - 1, con j posizione *mismatch* su P per *W* in posizione i
- Il confronto riparte dai simboli in posizione i+j-1 di T e in posizione φ(j-1) + 1 di P



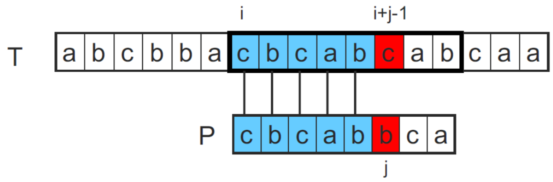
Se $\varphi(j-1)$ è uguale a 0

- *W* viene spostata alla posizione $p = i + j - \varphi(j-1) - 1$



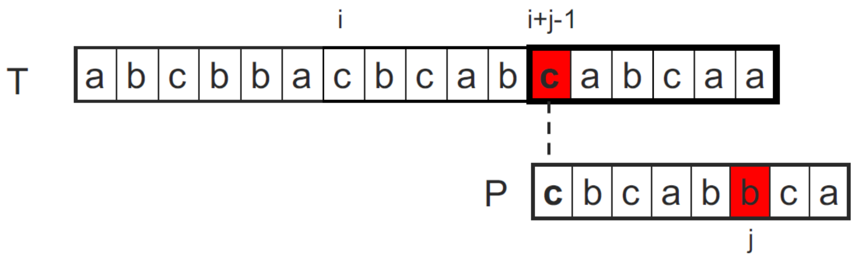
Se $\varphi(j-1)$ è uguale a 0

- *W* viene spostata alla posizione $p = i + j - 1$
- Il confronto riparte dai simboli in posizione i+j-1 di T e in posizione $\varphi(j-1) + 1$ di P



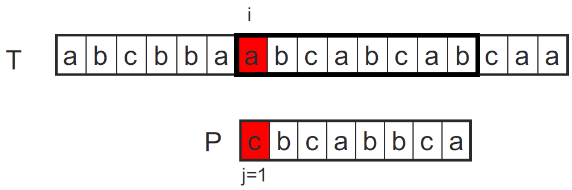
Se $\varphi(j-1)$ è uguale a 0

- *W* viene spostata alla posizione $p = i + j - 1$
- Il confronto riparte dai simboli in posizione i+j-1 di T e in posizione 1 di P

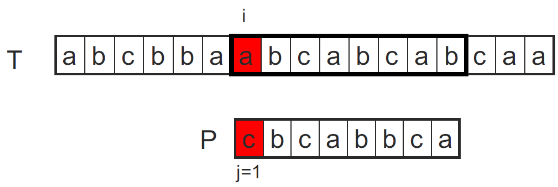


Spostamento di W (j=1)

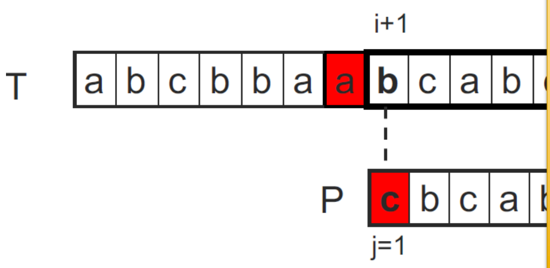
- *W* viene spostata alla posizione $p = i + j - \varphi(j-1) - 1$



- *W* viene spostata alla posizione $p = i + 1$
- Il confronto riparte dai simboli in posizione i+j-1 di T e in posizione $\varphi(j-1) + 1$ di P



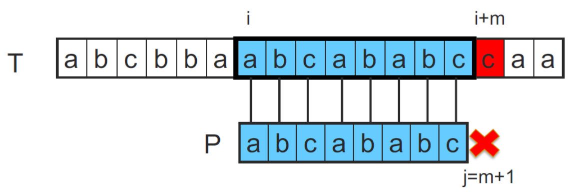
- *W* viene spostata alla posizione p = i + 1
- Il confronto riparte dai simboli in posizione i di T e in posizione 0 di P



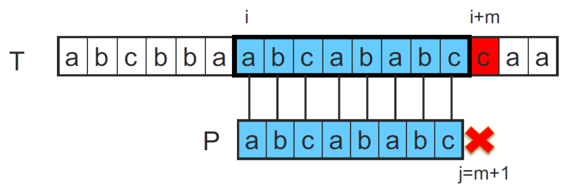
NB: chiaramente il confronto riparte dalle posizioni i+1 su T e 1 su P, ma dire che riparte dalla posizione i su T e dalla posizione 0 su P implicitamente fa riferimento ad un confronto iniziale fittizio tra T[i] e P[0] (simbolo inesistente) che di default viene considerato un *match*

Spostamento di W (j=m+1)

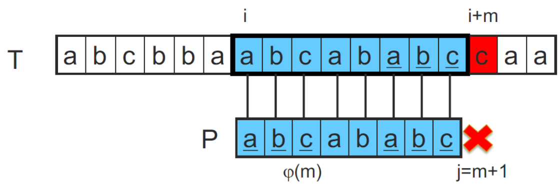
- *W* viene spostata alla posizione p = i + j - φ(j-1) - 1



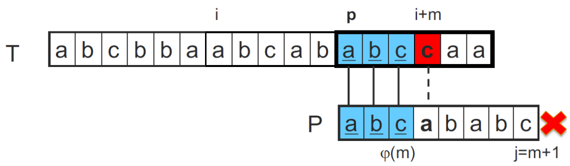
- *W* viene spostata alla posizione p = i + m - φ(m)
- Il confronto riparte dai simboli in posizione i+j-1 di T e in posizione φ(j-1) + 1 di P



- *W* viene spostata alla posizione p = i + m - φ(m)
- Il confronto riparte dai simboli in posizione i+m di T e in posizione φ(m) + 1 di P



- *W* viene spostata alla posizione p = i + m - φ(m)
- Il confronto riparte dai simboli in posizione i+m di T e in posizione φ(m) + 1 di P



Algoritmo di scansione di T

```
KMP(P,T,φ)
 begin
    m ← |P|
    n ← |T|
    j ← 0
    for q ← 1 to n do
       while j >= 0 and P[j+1] != T[q] then
            j ← φ(j)
       j ← j+1
       if j = m then
            output q-m+1
 end
```

Automa vs KMP

Memoria occupata:	$O(m\|\Sigma\|)$	**VS**	$O(m)$
Tempo:			
Preprocessing di P:	$O(m\|\Sigma\|)$	**VS**	$O(m)$
Scansione di T:	$O(n)$	**VS**	$O(n)$

Automa:

👍 efficiente per pattern piccoli
👎 richiede più tempo e memoria per pattern grandi
👍 ricerca di P in testi diversi

KMP:

👍 efficiente per pattern grandi
👎 richiede più tempo per pattern piccoli