Pattern matching: Alg. di Boyer-Moore

- Qual è la motivazione fondamentale per l'euristica looking-glass?
  - Serve a rendere massima l'efficacia dei "salti lunghi", quando, durante la verifica di una posizione iniziale del pattern nel testo, si trova una coppia di caratteri non corrispondenti e il carattere del testo NON è presente nel pattern
- Se, nella porzione di testo di lunghezza $|P|$ a partire da una particolare posizione *i*, **sono presenti più caratteri diversi da quelli corrispondenti in *P*** e questi **NON** sono presenti in *P*, allora, se la prima coppia diversa che trovo è quella più a sinistra, il salto sarà più piccolo di quello che posso fare se la prima coppia diversa che trovo è quella più a destra, perché il salto porta la nuova posizione iniziale ad essere immediatamente a destra del carattere diverso trovato
- Quindi è meglio fare la verifica da destra a sinistra, come prevede looking-glass

Pattern matching

- Tra le varianti di Boyer-Moore, ne esistono di più complicate, che riescono ad avere prestazioni $O(n + m + |\Sigma|)$, cioè $O(n + m)$
- Non le vediamo perché analizziamo, invece, un diverso algoritmo che ha prestazioni $O(n + m)$ e si basa su strategie diverse
- **Algoritmo KMP**
  - Dagli inventori: Knuth, Morris, Pratt
  - **Prestazioni $O(n + m)$ nel caso peggiore, cioè ottimo**, perché gli algoritmi di pattern matching sono $\Omega(n + m)$
  - Richiede, come Boyer-Moore, una "pre-compilazione" del pattern P, con occupazione di spazio $\Theta(m)$

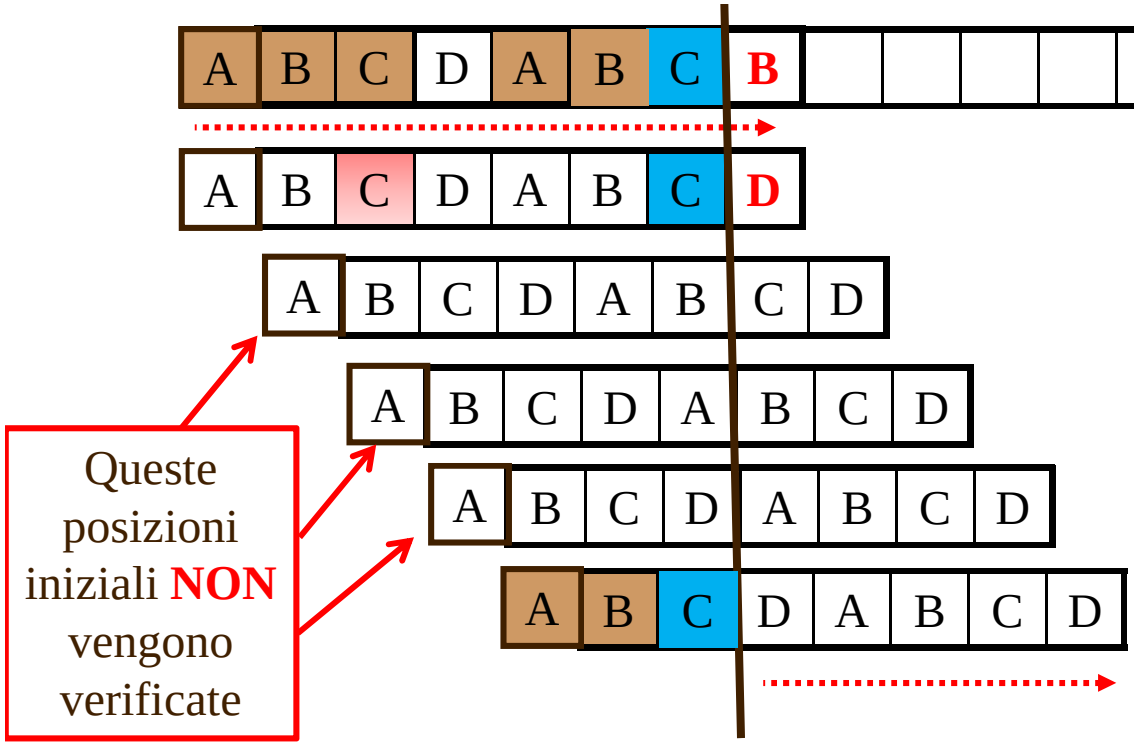
Pattern matching: Algoritmo KMP

- L'algoritmo KMP si basa sempre sullo schema di funzionamento dell'algoritmo a forza bruta
  - Diversamente da BM, **non usa la strategia looking-glass**: ogni possibile posizionamento viene verificato procedendo da sinistra a destra
  - Rispetto a *brute force*, è in grado di "saltare" alcuni posizionamenti, in modo più efficace di quanto fa BM, perché **sfrutta le informazioni derivanti dai confronti già fatti** fino al punto in cui ha trovato una "mancata corrispondenza" che provoca un nuovo posizionamento di *P*
    - Boyer-Moore, quando trova una "mancata corrispondenza", trae informazioni soltanto dal carattere di *T* che ha provocato la mancata corrispondenza, NON da tutti i confronti (andati a buon fine) fatti fino a quel punto… uno spreco…

Non è esattamente così, più avanti è descritto con precisione

Pattern matching: Algoritmo KMP

- Vediamo l'idea di base con un esempio

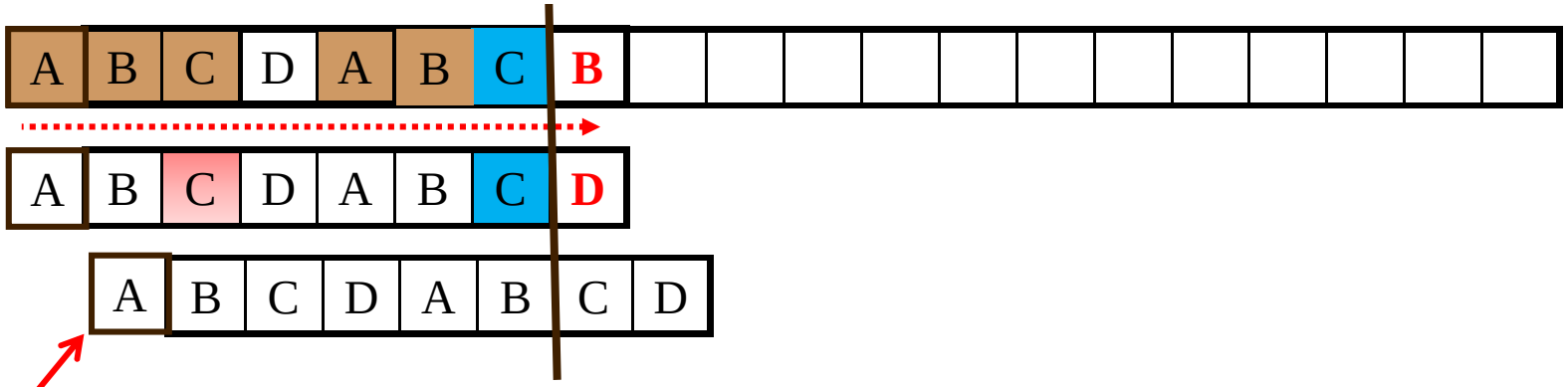


Trovata una discordanza, per decidere il nuovo posizionamento guardo il **carattere PRECEDENTE** a **quello che ha dato errore** e sposto il pattern fin quando trovo un carattere che coincide (se c'è). Sono sicuro che gli altri posizionamenti non andrebbero a buon fine (è simile a quello che fa Boyer-Moore, ma un po' diverso)

- La novità di KMP è che, a questo punto, trovato il nuovo posizionamento da verificare, "si ricorda" di alcuni confronti già andati a buon fine e evita di fare di nuovo tali confronti con la parte iniziale: come definire questa "parte iniziale"? **È il più lungo prefisso di *P* che è suffisso della porzione di *P* che precede il punto di mancata corrispondenza (vedi dettagli in seguito)**

Pattern matching: Algoritmo KMP

- Vediamo l'idea di base con un esempio



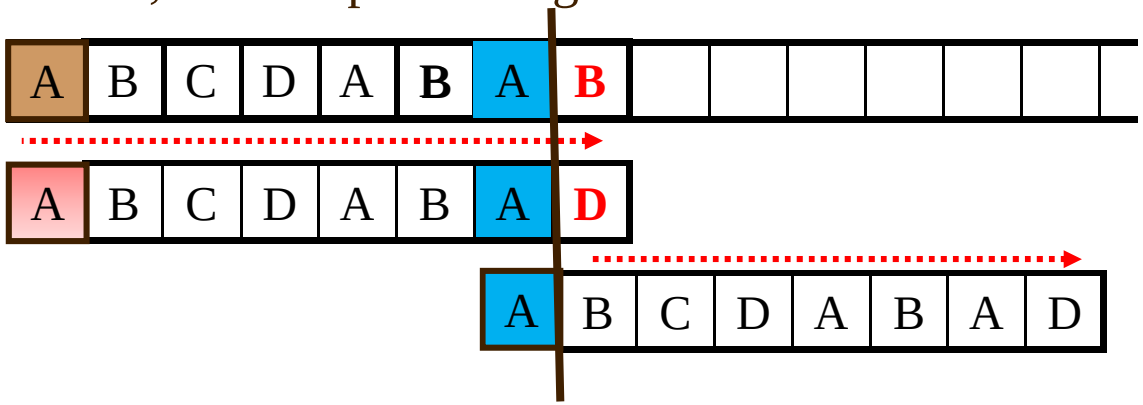
Questa posizione iniziale **NON** viene verificata

Perché il primo dei posizionamenti "scartati" va scartato?
Potrebbe essere, invece, un possibile match?
Se fosse un match, sarebbe $P[0\ldots5] = T[1\ldots6]$, ma sappiamo già che $T[1\ldots6] = P[1\ldots6]$ (dai confronti fatti in precedenza), quindi sarebbe $P[0\ldots5] = P[1\ldots6]$, cioè il più lungo prefisso di P che sia anche suffisso della porzione di P che precede il punto di mancata corrispondenza (che è $P[7]$) avrebbe lunghezza 6 e non, come in realtà è, 3. **Assurdo**.

- **Analogamente per gli altri posizionamenti scartati.**
- L'informazione "il più lungo prefisso…" serve sia per "spostare in avanti" il pattern (eventualmente saltando alcune posizioni), sia per non rifare alcuni confronti già fatti

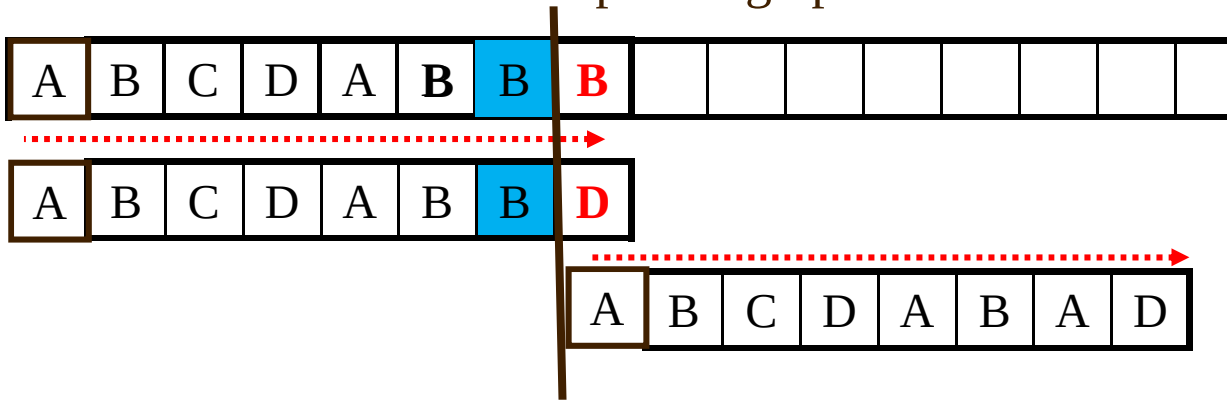
Pattern matching: Algoritmo KMP

- Naturalmente, "il più lungo prefisso che sia suffisso…" può anche essere "corto", ad esempio di lunghezza unitaria



Anche in questo caso, dopo lo "scorrimento" del pattern, i caratteri che costituiscono il prefisso appena utilizzato NON vengono confrontati di nuovo, perché si sa già che il confronto andrebbe a buon fine

- Può anche succedere che "il più lungo prefisso…" abbia lunghezza zero!



In questo caso, dopo lo "scorrimento" del pattern, il confronto ricomincia dall'inizio del pattern: non ci sono informazioni da riutilizzare (ma si fa un salto "lungo"…)

Pattern matching: Algoritmo KMP

- In pratica, KMP sfrutta la presenza di sottostringhe ripetute nel pattern
  - Di conseguenza, **KMP è più efficiente quando l'alfabeto è di piccole dimensioni (al contrario di Boyer-Moore)**, perché aumenta la probabilità che un pattern contenga sottostringhe ripetute
- Come vedremo dalla più dettagliata presentazione che segue, **nel testo *T* non si torna MAI indietro** a confrontare caratteri già confrontati
  - Al massimo si sottopone nuovamente a confronto il carattere appena confrontato
  - Questo NON è vero in Boyer-Moore, né, ovviamente, in *brute force*
- La **pre-compilazione** di cui ha bisogno KMP è la seguente
  - Per ogni posizione *j* nel pattern *P*, che dimensione ha il più lungo prefisso di *P* che sia anche suffisso di *P*[**1**…*j*] ?
  - Attenzione, suffisso di *P*[**1**…*j*], non di *P*[**0**…*j*], altrimenti la risposta sarebbe sempre *j* + 1… e non sarebbe utile

Pattern matching: Algoritmo KMP

- La funzione pre-compilata si chiama *failure function*, la indichiamo con f ed è una funzione della posizione j del carattere che precede quello di mancata corrispondenza (oltre che, ovviamente, del pattern P)
  - $f(j) = \max_{P[0\ldots k]=P[j-k\ldots j],\ k<j} |P[0\ldots k]|$
  - $f(0) = 0$ (comodo per descrivere l'algoritmo)
  - Proprietà: $f(j) \leq j$
- Esempio: pattern

a	*b*	*a*	*a*	*b*	*a*

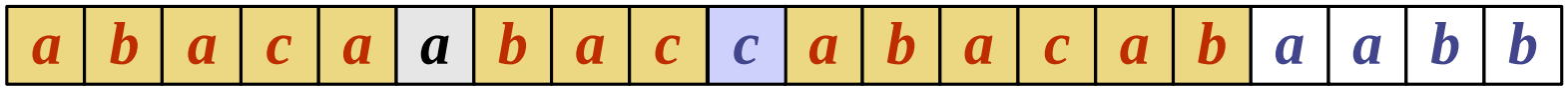
- Failure function:

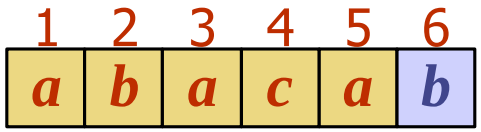
j	0	1	2	3	4	5
P[j]	*a*	*b*	*a*	*a*	*b*	*a*
f(j)	0	0	1	1	2	3

Pattern matching: Algoritmo KMP

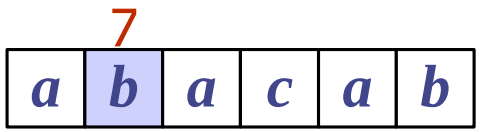
```
Algorithm KMPMatch(T, P)
    f ← KMPFailureFunction(P) // costruisce la failure function
    i ← 0            // i: indice nel testo,T = T[0...n-1]
    j ← 0            // j: indice nel pattern, P = P[0...m-1]
    while i < n
          if P[j] = T[i]
                if j = m - 1
                      return i - m - 1 // match con inizio in T[i-m-1]
                else i ← i + 1 // trovata corrispondenza, procedo
                      j ← j + 1
          else if j > 0          // no match, ma si è avanzati in P
                j ← f(j-1) // j pos. subito dopo il matching prefix in P
                                 // osservare che i non cambia
          else    // j = 0, il primo carattere di P non corrisponde a T[i]
                i ← i + 1 // nessun "trucco", j rimane 0
    return P ⊄ T
```

Algoritmo KMP: esempio

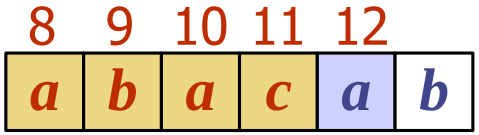




j = f(4) = 1 shift=4

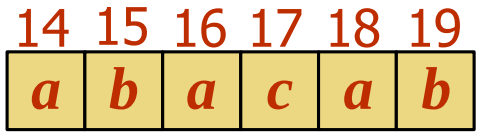


j = f(0) = 0 shift=1



j = f(3) = 0 shift=4

j=0



j	0	1	2	3	4	5
P[j]	a	b	a	c	a	b
f(j)	0	0	1	0	1	2
shift		1	2	2	4	4

shift(j) = j - f(j-1)

Algoritmo KMP: complessità

- Definiamo, ad ogni passo dell'algoritmo, la variabile $\boldsymbol{k = i - j}$
  - Rappresenta lo scorrimento del pattern rispetto al testo
  - Ovviamente $i \geq j$, quindi $k \geq 0$; inoltre, $\boldsymbol{k \leq n}$ **e** $\boldsymbol{i \leq n}$
- Ad ogni iterazione del ciclo principale, se non si trova un match, succede una e una sola di queste quattro situazioni
  - $T[i] = P[j] \Rightarrow i$ aumenta di 1, j aumenta di 1, quindi k non cambia (infatti, il pattern non scorre)
  - $T[i] \neq P[j]$ e $j > 0 \Rightarrow i$ non cambia, j diventa uguale a $f(j-1) \leq j - 1$, quindi k aumenta almeno di 1
  - $T[i] \neq P[j]$ e $j = 0 \Rightarrow i$ aumenta di 1, j non cambia, quindi k aumenta di 1
- Ad ogni iterazione, i aumenta di 1 oppure k aumenta di (almeno) 1 (oppure entrambe le cose), **quindi** il numero di iterazioni è $\leq 2n$

Algoritmo KMP: complessità

- **Il numero di iterazioni del ciclo di KMP è $\leq 2n$**
- La valutazione della funzione *f* è $O(1)$, perché questa viene pre-compilata in una tabella, quindi
  - ogni iterazione del ciclo è $\boldsymbol{O(1)}$
- L'algoritmo è, quindi, $O(n + F)$, essendo F il tempo necessario alla pre-compilazione della failure function
  - Le prestazioni di KMP non dipendono dalla lunghezza del pattern, se non nel calcolo della failure function
- Vedremo che $F = O(m)$, quindi **l'algoritmo KMP** è $\boldsymbol{O(n + m)}$, cioè **è un algoritmo di pattern matching ottimo**

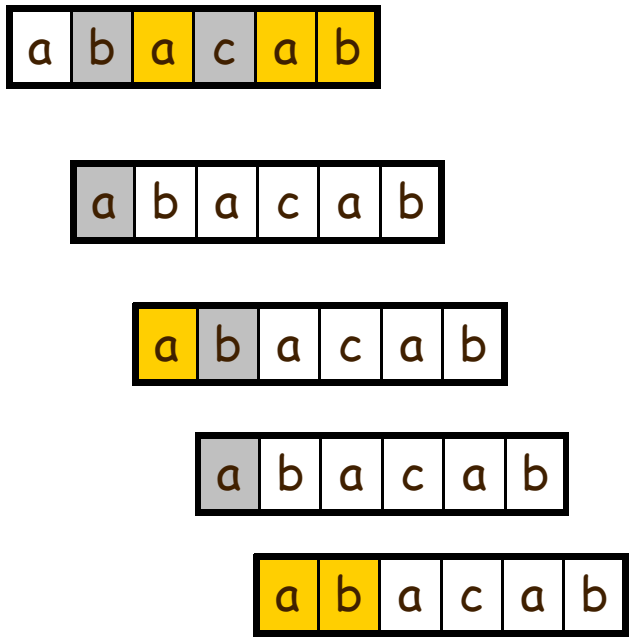
KMP: calcolo della failure function

- Per calcolare la *failure function* si usa un algoritmo assai simile a KMP stesso: si cerca il pattern *P* all'interno del pattern *P* !
- L'algoritmo è un po' complicato da capire…
- Mentre costruisce la funzione *f*, **usa** la funzione *f* che sta costruendo!
  - Naturalmente, usa sempre valori di *f* che ha già calcolato
  - Ha le stesse prestazioni di KMP su un testo di lunghezza *m*, quindi è $O(m)$

KMP: calcolo della failure function

```
Algorithm KMPFailureFunction(P)
    f(0) ← 0
    i ← 1
    j ← 0
    while i < m
        if P[j] = P[i]
            f(i) ← j + 1
            i ← i + 1
            j ← j + 1
        else if j > 0
            j ← f(j-1)    // usa la funzione che sta costruendo
                          // ma è sempre j ≤ i, quindi j – 1 < i, cioè
                          // per calcolare f(i) uso f(j – 1 < i)
        else
            f(i) = 0
            i ← i + 1
```

KMP: calcolo della failure function



i=1 j=0 f(i)=0 ➔f(1)=0

i=2 j=0 f(i)=j+1 ➔f(2)=1

i=3 j=1 j = f(j-1) = 0

i=3 j=0 f(i)=0 ➔f(3)=0

i=4 j=0 f(i)=j+1 ➔f(4)=1

i=5 j=1 f(i)=j+1 ➔f(5)=2

j	0	1	2	3	4	5
P[j]	*a*	*b*	*a*	*c*	*a*	*b*
f(j)	0	0	1	0	1	2