ALGORITMI E STRUTTURE DATI

Prof. Manuela Montangero

A.A. 2021/22

Algoritmi su stringhe: String matching

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Stringhe

Alfabeto: insieme di caratteri finito $\Sigma = \{c_1, c_2, \ldots, c_m\}$

Stringa su alfabeto Σ: sequenza lineare di elementi di Σ,

$\alpha = s_0 s_1 s_2 \ldots s_n$, tale che $\forall i = 0,\ldots,n \;\; s_i \in \Sigma$

Insieme di stringhe definite su Σ: Σ^* (stella di Kleene o monodie libero)

insieme di tutte le stringhe di lunghezza finita sull'alfabeto Σ

Esempi:

- $\Sigma = \{0,1\}$, un stringa $\alpha = 1001010010$ è un numero binario,
  Σ^* = insieme di tutti i numeri binari (di qualunque lunghezza)
- $\Sigma = \{\texttt{a}, \texttt{c}, \texttt{g}, \texttt{t}\}$, un stringa $\alpha = \texttt{acctggtgca}$ è una porzione di DNA
- $\Sigma = \{a, b, c, d, e, f, g, h, i, l, m, n, o, p, q, r, s, t, u, v, z\}$, una stringa
  $\alpha = algortimi$ è una parola contenente lettere dell'alfabeto italiano

Stringhe

DEFINIZIONI:

- Stringa vuota (non ha caratteri): ϵ
- Accesso carattere i-esimo di una stringa α: $\alpha[i]$ o α_i (iniziamo a contare da indice zero)

α = ALGORITMI $\alpha[3] = \text{O}$

- Lunghezza di una stringa α: $|\alpha|$ = numero di caratteri di α ($|\epsilon| = 0$)

α = ALGORITMI $|\alpha| = 9$

- β è una sottostringa di α se esistono due stringhe γ e β (eventualemente vuote) tale che $\alpha = \gamma\beta\delta$
($\alpha[i\,..\,j] = \alpha[i]\alpha[i+1]\ldots\alpha[j] = \alpha_{i..j}$ sottostringa di α con i caratteri dall'i-esimo al j-esimo)

α = ALGORITMI, β = ORI —> γ = ALG e δ = TMI

- β è un suffisso di α se esiste una stringa γ (eventualemente vuota) tale che $\alpha = \gamma\beta$

α = ALGORITMI, β = RITMI —> γ = ALGO

- β è un prefisso di α se esiste una stringa δ (eventualemente vuota) tale che $\alpha = \beta\delta$

α = ALGORITMI, β = AL —> β = GORITMI

Stringhe

OPERAZIONI SULLE STRINGHE:

- **Concatenazione**:
  date su stringhe α e β sullo stesso alfabeto Σ,
  la concatenazione è $\gamma = \alpha\beta$ data dalla giustapposizione di α e β ($|\gamma| = |\alpha| + |\beta|$)

α = IPPO

β = POTAMO

γ = IPPOPOTAMO

- **Potenza**:
  data una stringa α su un alfabeto Σ,
  la potenza $k-$esima è la concatenazione di α con se stessa k volte

α = PIPPO

α^3 = PIPPOPIPPOPIPPO

String-matching

OCCORRENZA:

- se β è una sottostringa di α, allora α contiene (almeno) un'occorrenza di

α = ALGORITMI, β = ORI —> γ = ALG e δ = TMI

occorrenza di β in α

- se $\alpha = \gamma\beta\delta$ e $|\gamma| = k$, β occorre (ha un match) alla posizione k di α

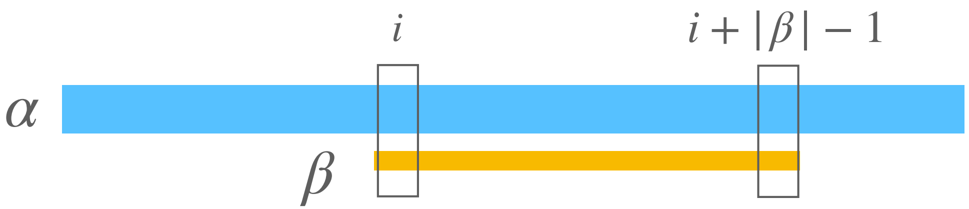
α = ALGORITMI, β = ORI —> γ = ALG e δ = TMI

occorrenza di β in α alla posizione 3

String-matching

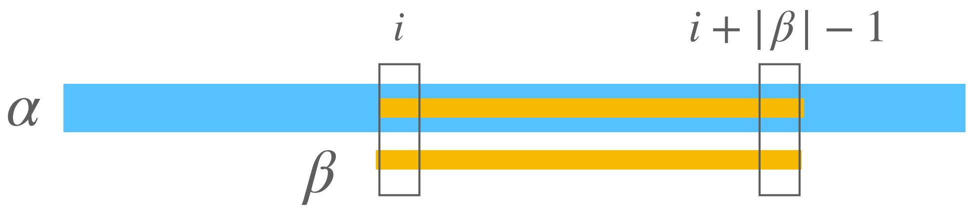
ALLINEAMENTO:

L'allineamento di una stringa β alla posizione i della stringa α è la sovrapposizione (ideale) di β con la sottostringa $\alpha[i\,..\,i+|\beta|-1]$



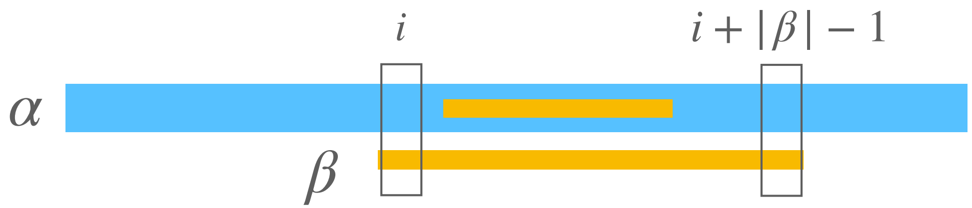
ALLINEAMENTO con SUCCESSO (matching):

Esiste un'occorrenza della stringa β alla posizione i della stringa α



ALLINEAMENTO FALLITO

Non esiste un'occorrenza della stringa β alla posizione i della stringa α



String-matching

PROBLEMA:

INPUT: Stringa $T \in \Sigma^*$ di n caratteri —> TESTO

Stringa $P \in \Sigma^*$ di $m \leq n$ caratteri —> PATTERN

OUTPUT: le (eventuali) posizioni delle occorrenze di P in T

ESEMPIO:

```
T = perdindirindina

P = din
```

Occorrenze di P in T alle posizioni 3 e 11

perdindirindina

String-matching

PROBLEMA:

INPUT: Stringa $T \in \Sigma^*$ di n caratteri —> TESTO

Stringa $P \in \Sigma^*$ di $m \leq n$ caratteri —> PATTERN

OUTPUT: le (eventuali) posizioni delle occorrenze di P in T

Lower bound sul numero di confronti tra TESTO e PATTERN:

Ogni carattere del testo DEVE essere coinvolto in ALEMNO un confronto

—> Lower bound $\Omega(|T|)$

`T = 1`^{n}

`P = 1`

“saltando" anche solo un carattere di T si “perde" un’occorrenza di P in T

String-matching

PROBLEMA:

INPUT: Stringa $T \in \Sigma^*$ di n caratteri —> TESTO

Stringa $P \in \Sigma^*$ di $m \leq n$ caratteri —> PATTERN

OUTPUT: le (eventuali) posizioni delle occorrenze di P in T

ALGORITMO a FORZA BRUTA

IDEA:

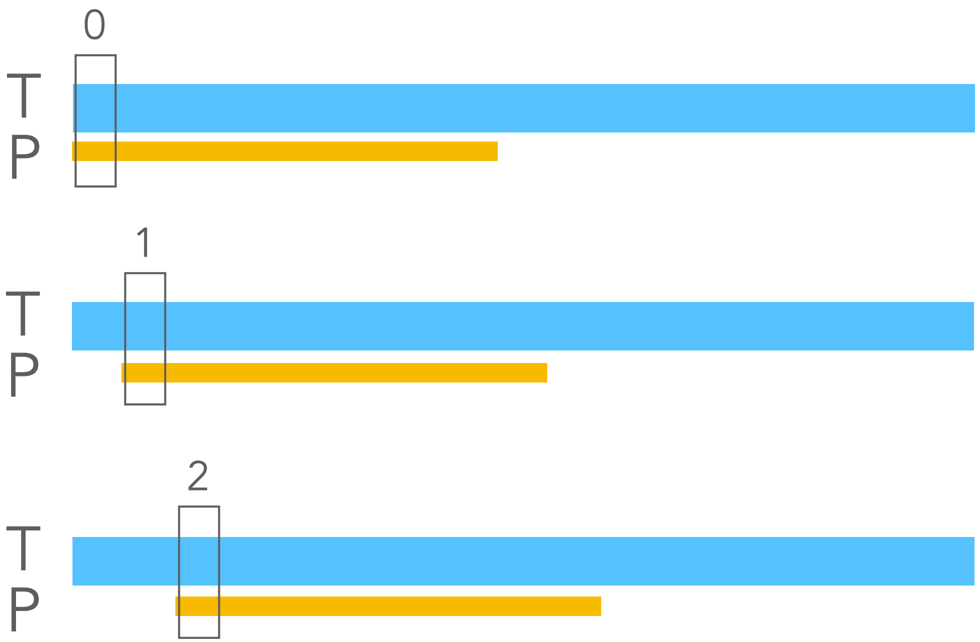
proviamo tutti i possibili allineamenti del pattern nel testo e verifichiamo danno un successo o un fallimento

String-matching

ALGORITMO a FORZA BRUTA

IDEA:

proviamo tutti i possibili allineamenti del pattern nel testo e verifichiamo danno un successo o un fallimento



1a possibilità —> indice 0 di T

2a possibilità —> indice 1 di T

3a possibilità —> indice 2 di T

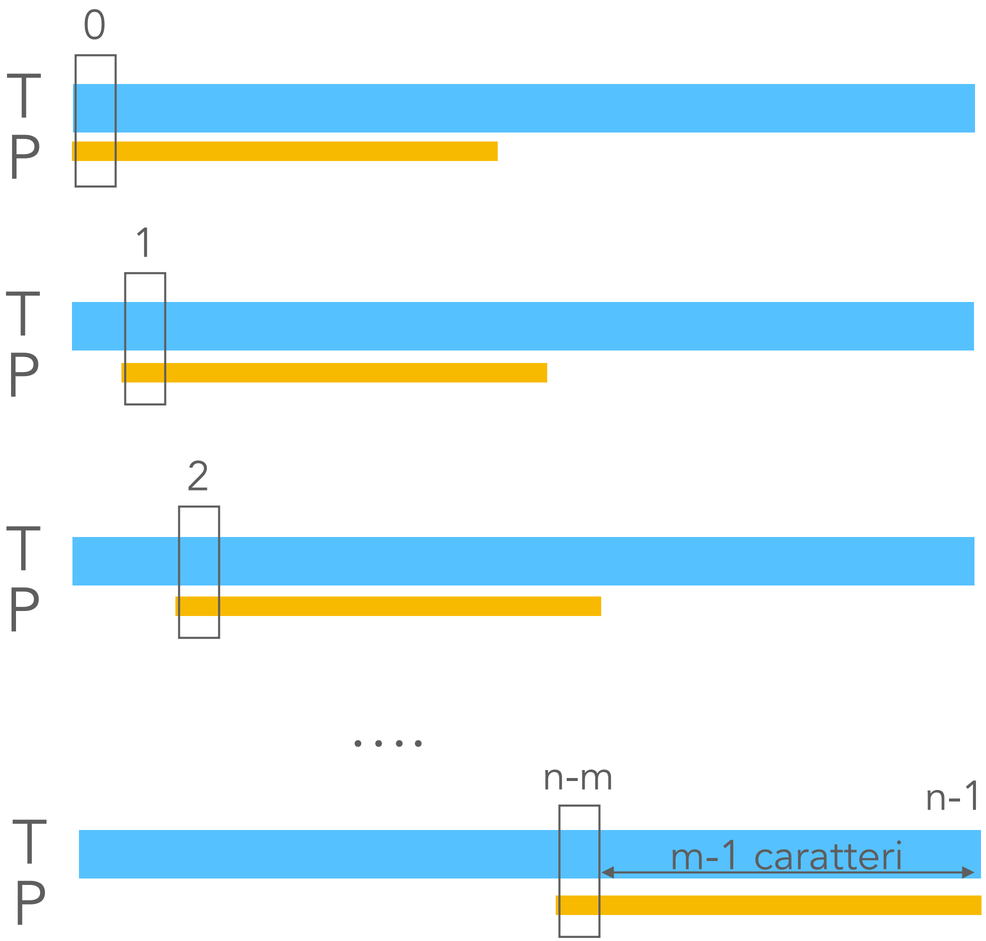
....

String-matching

ALGORITMO a FORZA BRUTA

IDEA:

proviamo tutte le possibili collocazioni del pattern nel testo e verifichiamo se determinano un’occorrenza o no



1a possibilità —> indice 0 di T

2a possibilità —> indice 1 di T

3a possibilità —> indice 2 di T

ultima possibilità —> indice n-m di T

String-matching

ALGORITMO a FORZA BRUTA

IDEA:

proviamo tutte le possibili collocazioni del pattern nel testo e verifichiamo se determinano un'occorrenza o no

```
Brute-Force(T,P)
 n := |T|
 m := |P|
 for i = 0 to n-m do
  j := 0
  while (j < m AND T[i+j] = P[j]) do
   j := j + 1
  if j = m
   then print i
 print -1
```

Controlla se $P = T[i\,..\,i+m-1]$

String-matching

ALGORITMO a FORZA BRUTA

IDEA:

proviamo tutte le possibili collocazioni del pattern nel testo e verifichiamo se determinano un'occorrenza o no

```
Brute-Force(T,P)
 n := |T|
 m := |P|
 for i = 0 to n-m do
  j := 0
  while (j < m AND T[i+j] = P[j]) do
   j := j + 1                          O(m)
  if j = m
   then print i
 print -1
```

Costo computazionale

contiamo solo i confronti
tra caratteri delle due stringhe

Il for viene eseguito `n-m+1` volte

$$(n-m+1)\cdot m = nm - m^2 + m$$

se $m \approx n \Rightarrow O(n)$

se $m = n/c, c \in O(1) \Rightarrow O(n^2)$

se $m \in O(1) \Rightarrow O(n)$

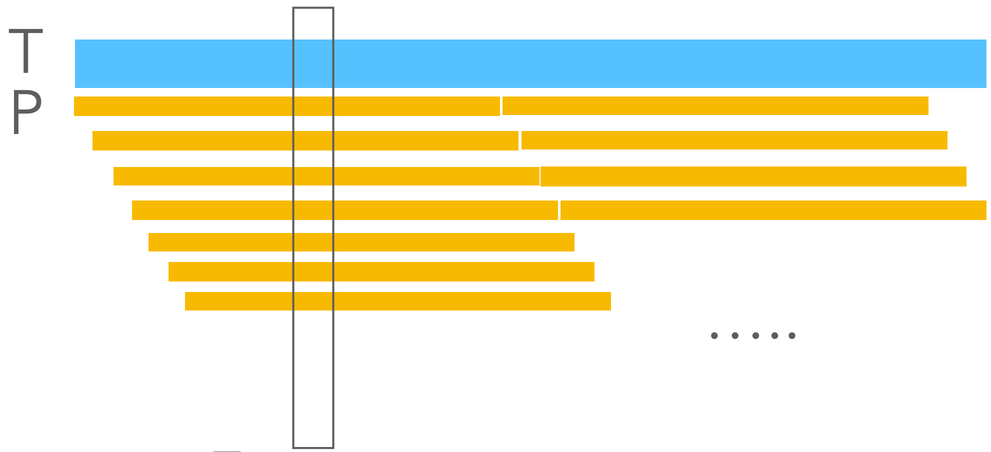
String-matching

ALGORITMO a FORZA BRUTA

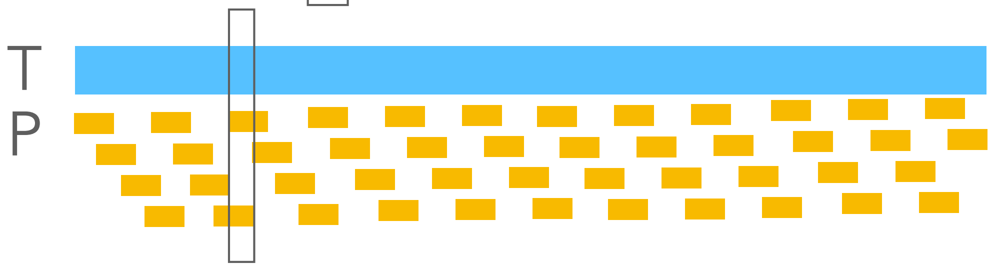
Costo computazionale

se $m \approx n \Rightarrow O(n)$

se $m = n/c, c \in O(1) \Rightarrow O(n^2)$



se $m \in O(1) \Rightarrow O(n)$



String-matching

L'algoritmo Brute-Force testa anche posizioni in cui è impossibile che inizi una nuova occorrenza (e lo sa!)

i

T .... A A A $\boxed{\text{B}}$ x x x x x .....

P A A A $\boxed{\text{A}}$ x x x x ...

è possibile che ci sia un'occorrenza del pattern nel testo in posizione i+1?

i $i+1$

T .... A A A $\boxed{\text{B}}$ x x x x x .....

P A A $\boxed{\text{A}}$ A x x x x ...

NO

String-matching

L’algoritmo Brute-Force testa anche posizioni in cui è impossibile che inizi una nuova occorrenza (e lo sa!)

	i									
T	.... A	A	A	$\boxed{B}$	x	x	x	x	x	.....
P	A	A	A	$\boxed{A}$	x	x	x	x	...	

qual è il primo indice (dopo i) in cui è possibile che ci sia un’occorrenza del pattern nel testo?

	i				$i+4$					
T	.... A	A	A	B	$\boxed{x}$	x	x	x	x	.....
P					$\boxed{A}$	A	A	A	x	x x x ...

String-matching

L’algoritmo Brute-Force testa anche posizioni in cui è impossibile che inizi una nuova occorrenza (e lo sa!)

i

T .... A B A B x x x x x .....

P A B A A x x x x ...

qual è il primo indice (dopo i) in cui è possibile che ci sia un’occorrenza del pattern nel testo?

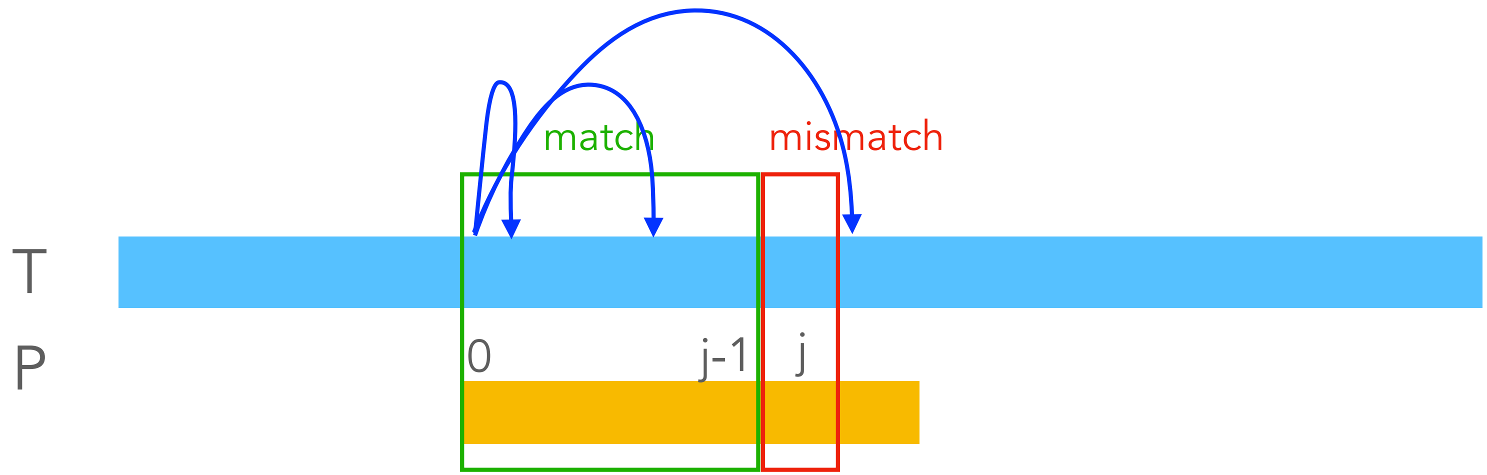
i *i+2*

T .... A B A B x x x x x .....

P A B A A x x x x ...

String-matching

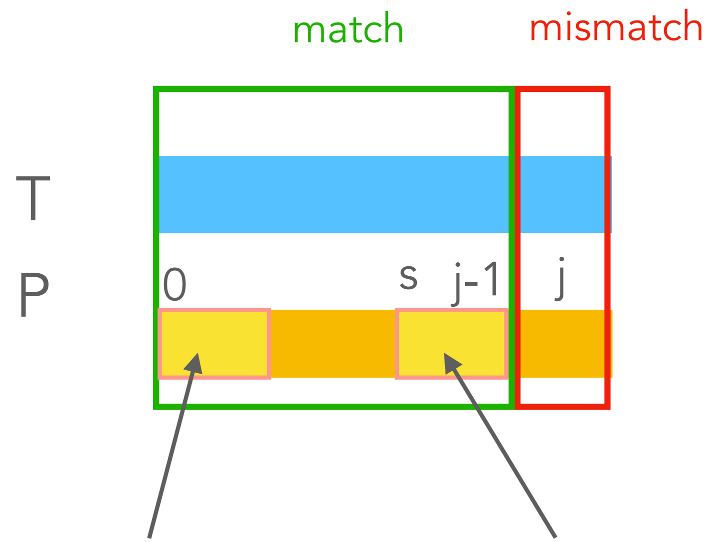
Come determinare il prossimo allineamento?



shift: #posizioni di cui spostarsi a destra nel testo per provare un nuovo allineamento

String-matching

Come determinare il prossimo allineamento?



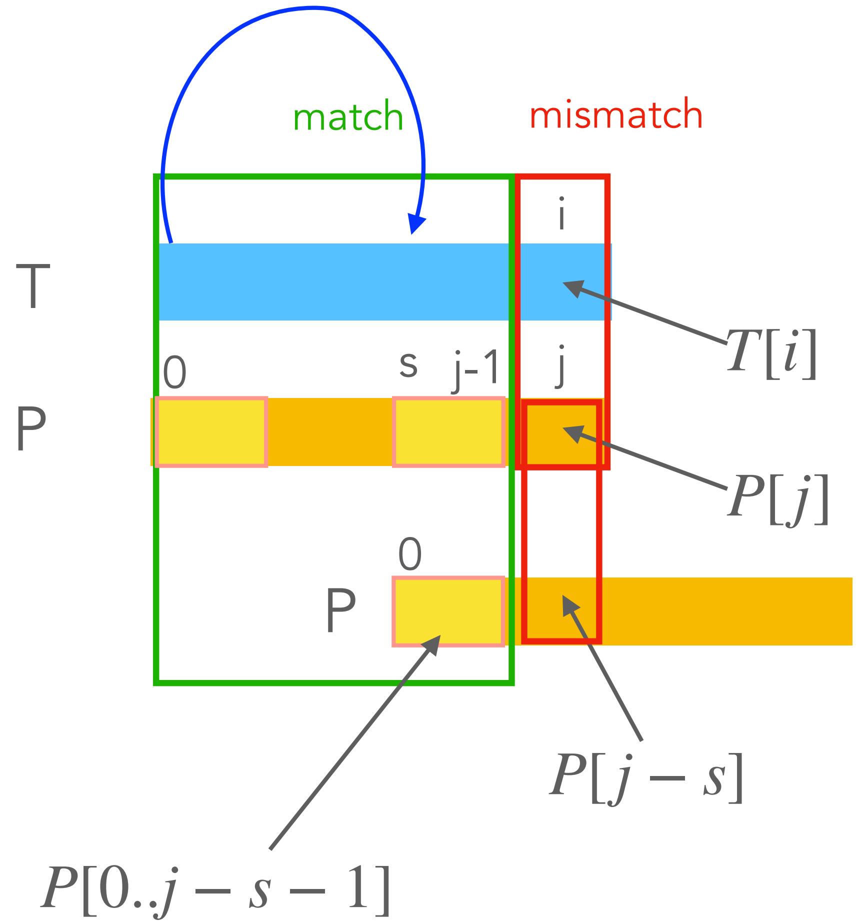
$P[0..j-s-1]$ $\quad$ $P[s..j-1]$

se $P[0..j-s-1] = P[s..j-1]$?

UNIMORE

String-matching

Come determinare il prossimo allineamento?



se $P[0..j-s-1] = P[s\,..\,j-1]$?

Vale la pena riprovare con uno **shift di s** posizioni?

dipende da $P[j-s]$ e $P[j]$

se $P[j] = P[j-s]$ non è possibile perché $P[j-s] \neq T[i]$

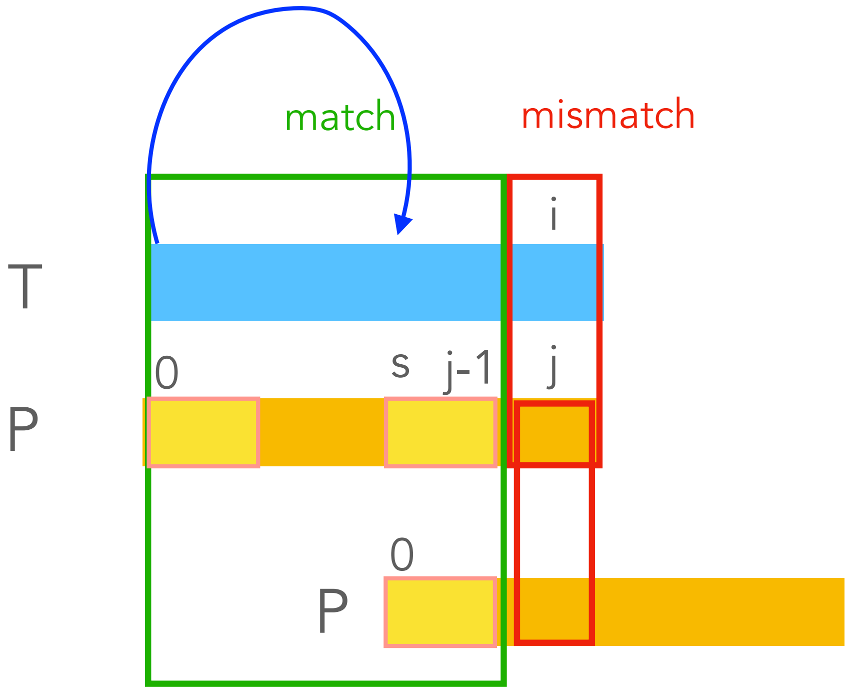
se $P[j] \neq P[j-s]$ vale la pena provare

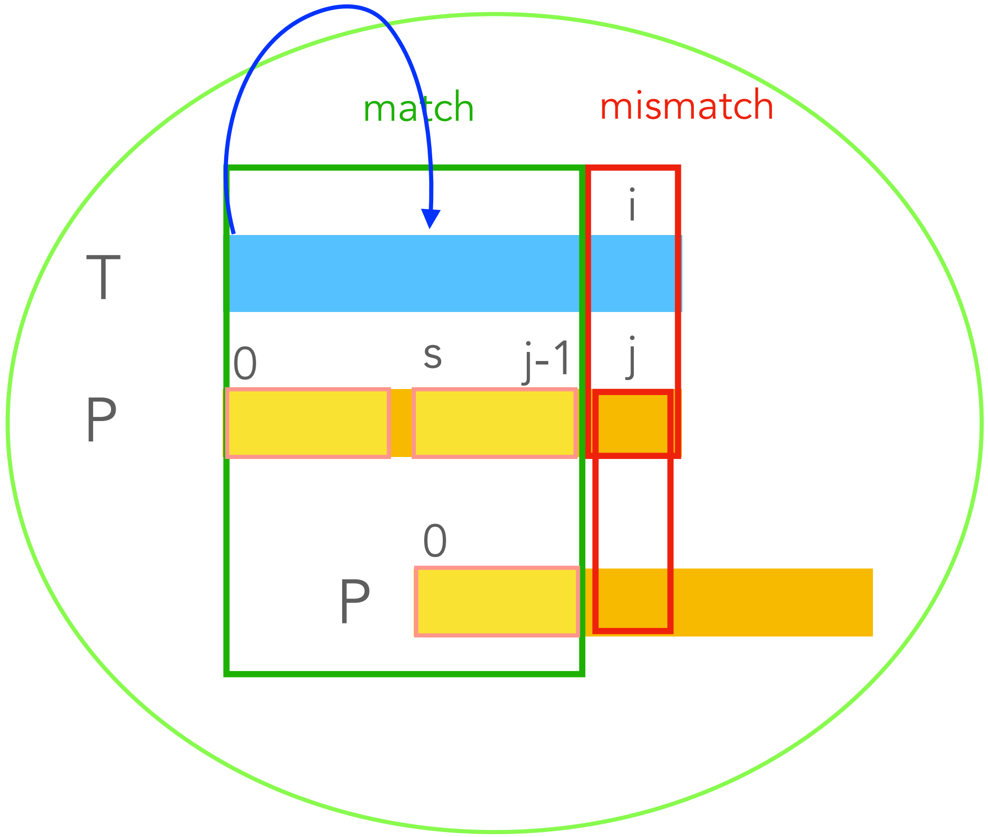
String-matching

Come determinare il prossimo allineamento?

e se esiste più di un s per cui vale

$P[0..j-s-1] = P[s..j-1]$ e $P[j] \neq P[j-s]$?



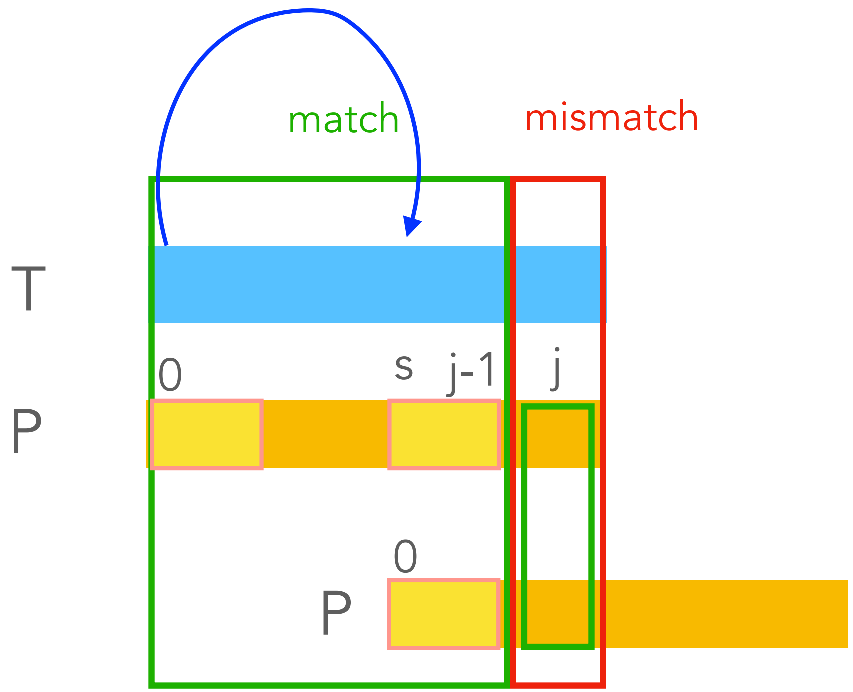


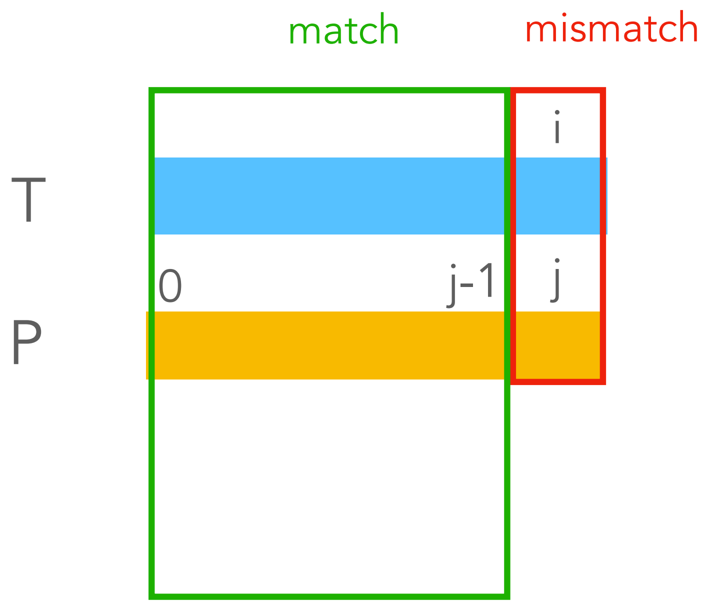
scegliamo l's più piccolo

String-matching

Come determinare il prossimo allineamento?

e se non esiste tale s?



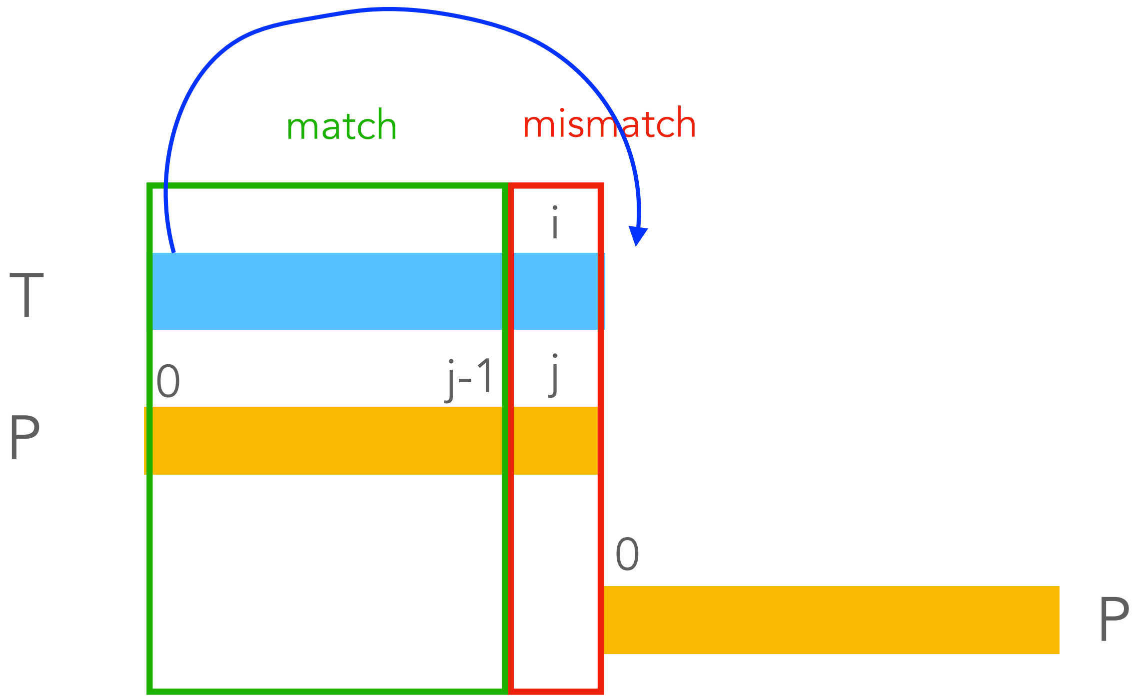


UNIMORE

String-matching

Come determinare il prossimo allineamento?

e se non esiste tale s?

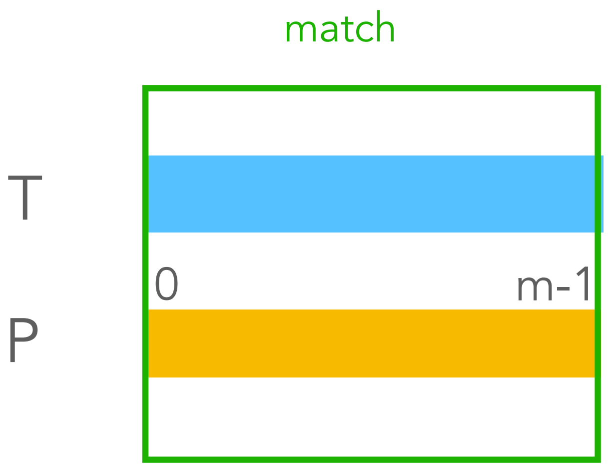


il prossimo tentativo va fatto con uno **shift di $j+1$** posizioni

String-matching

Come determinare il prossimo allineamento?

e se abbiamo trovato un’occorrenza
(quindi non c’è mismatch)?



Si ragiona in modo analogo,
senza dover controllare cosa
succede nella posizione m-esima

String-matching

Come determinare il prossimo allineamento?

Analizzando il pattern (e basta!)

Per ogni $0 \leq j \leq |P|$ per ogni possibile posizione del mismatch, o se è stata trovata un'occorrenza del pattern

cerchiamo il più piccolo s tale che $0 \leq s \leq j+1$ e $A_P(j,s) = \texttt{True}$

minima lunghezza dello shift

$$A_P(j,s) = \begin{cases} \texttt{True} & \text{se } P[0..j-s-1] = P[s\,..\,j-1] \texttt{ AND } (P[j] \neq P[j-s] \texttt{ OR } j = |P|) \\ \texttt{True} & \text{se } s = j+1 \\ \texttt{False} & \text{altrimenti} \end{cases}$$

UNIMORE

String-matching

Come determinare il prossimo allineamento?

Analizzando il pattern (e basta!)

Non sono coinvolti confronti con il testo

```
Best-Prefix(P)
 m := |P|
 shift := new array[0..m]
 for j = 0 to m do
  s := 1
  while NOT AP(j,s) do
   s := s+1
  shift[j] := s
 return shift[]
```

per ogni possibile posizione del mismatch (`j < m`), o se è stata trovata un'occorrenza del pattern (`j = m`)

ricerca dello shift di minima lunghezza

il **`while`** termina al più tardi quando **`s = j+1`**

quando non inizia una nuova iterazione del **`while`**, il valore di **`s`** è il primo per cui **`AP(j,s) = True`**

String-matching

ESEMPIO

```
Best-Prefix(P)
 m := |P|
 shift := new array[0..m]
 for j = 0 to m do
  s := 1
  while NOT AP(j,s) do
   s := s+1
  shift[j] := s
 return shift[]
```

P = dindina   m=|P|=7

j	P[0..j-1]	shift[j]
0		1
1	d	
2	di	
3	din	
4	dind	
5	dindi	
6	dindin	
7	dindina	

String-matching

ESEMPIO

```
Best-Prefix(P)
 m := |P|
 shift := new array[0..m]
 for j = 0 to m do
  s := 1
  while NOT A_P(j,s) do
   s := s+1
  shift[j] := s
 return shift[]
```

P = dindina   m=|P|=7

j	P[0..j-1]	shift[j]
0		1
1	d	1
2	di	
3	din	
4	dind	
5	dindi	
6	dindin	
7	dindina	

T   d X

P   d i n d i n a

0 1
=
j

String-matching

ESEMPIO

```
Best-Prefix(P)
 m := |P|
 shift := new array[0..m]
 for j = 0 to m do
  s := 1
  while NOT AP(j,s) do
   s := s+1
  shift[j] := s
 return shift[]
```

P = dindina   m=|P|=7

j	P[0..j-1]	shift[j]
0		1
1	d	1
2	di	2
3	din	
4	dind	
5	dindi	
6	dindin	
7	dindina	

T   d i X

P   d i n d i n a

0 2 = j

String-matching

ESEMPIO

```
Best-Prefix(P)
 m := |P|
 shift := new array[0..m]
 for j = 0 to m do
  s := 1
  while NOT A_P(j,s) do
   s := s+1
  shift[j] := s
 return shift[]
```

P = dindina   m=|P|=7

j	P[0..j-1]	shift[j]
0		1
1	d	1
2	di	2
3	din	4
4	dind	
5	dindi	
6	dindin	
7	dindina	

T   d i n X

P   d i n d i n a

0   3

=

j

String-matching

ESEMPIO

```
Best-Prefix(P)
 m := |P|
 shift := new array[0..m]
 for j = 0 to m do
  s := 1
  while NOT A_P(j,s) do
   s := s+1
  shift[j] := s
 return shift[]
```

P = dindina   m=|P|=7

j	P[0..j-1]	shift[j]
0		1
1	d	1
2	di	2
3	din	4
4	dind	4
5	dindi	
6	dindin	
7	dindina	

```
T     d i n d X
P     d i n d i n a
      0       4
              =
              j
```

String-matching

ESEMPIO

```
Best-Prefix(P)
 m := |P|
 shift := new array[0..m]
 for j = 0 to m do
  s := 1
  while NOT A_P(j,s) do
   s := s+1
  shift[j] := s
 return shift[]
```

P = dindina   m=|P|=7

j	P[0..j-1]	shift[j]
0		1
1	d	1
2	di	2
3	din	4
4	dind	4
5	dindi	5
6	dindin	
7	dindina	

```
T   d i n d i X
P   d i n d i n a
    0         5
              =
              j
```

String-matching

ESEMPIO

```
Best-Prefix(P)
 m := |P|
 shift := new array[0..m]
 for j = 0 to m do
  s := 1
  while NOT AP(j,s) do
   s := s+1
  shift[j] := s
 return shift[]
```

P = dindina m=|P|=7

j	P[0..j-1]	shift[j]
0		1
1	d	1
2	di	2
3	din	4
4	dind	4
5	dindi	5
6	dindin	3
7	dindina	

```
T    d i n d i n X
P    d i n d i n a
     0           6
                 =
                 j
```

String-matching

ESEMPIO

```
Best-Prefix(P)
 m := |P|
 shift := new array[0..m]
 for j = 0 to m do
  s := 1
  while NOT A_P(j,s) do
   s := s+1
  shift[j] := s
 return shift[]
```

P = dindina   m=|P|=7

j	P[0..j-1]	shift[j]
0		1
1	d	1
2	di	2
3	din	4
4	dind	4
5	dindi	5
6	dindin	3
7	dindina	7

T   d i n d i n a

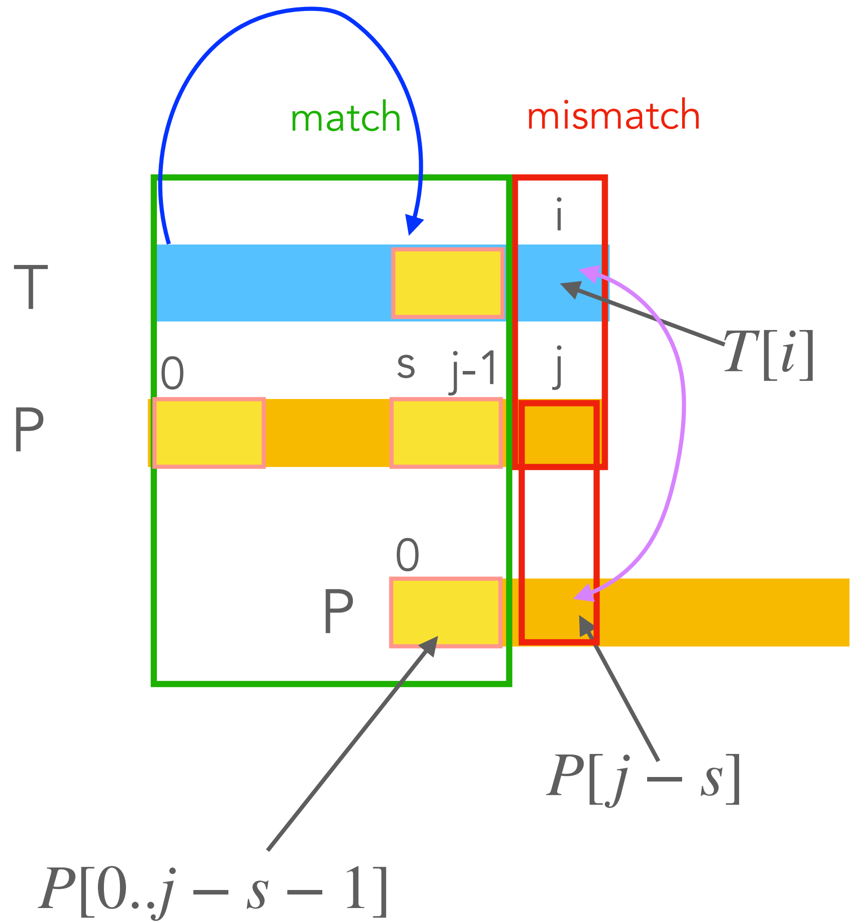
P   d i n d i n a

0   7 = j

String-matching

Sappiamo come determinare il prossimo tentativo di allineamento, qual’è il prossimo confronto da fare?

Evitiamo di ripetere confronti tra il testo e il pattern di cui conosciamo già l'esito



shift $\leq j$

Dall’allenamento precedente sappiamo che

$$P[0..j-s-1] = T[i-j+s\,..\,i-1]$$

$$\Downarrow$$

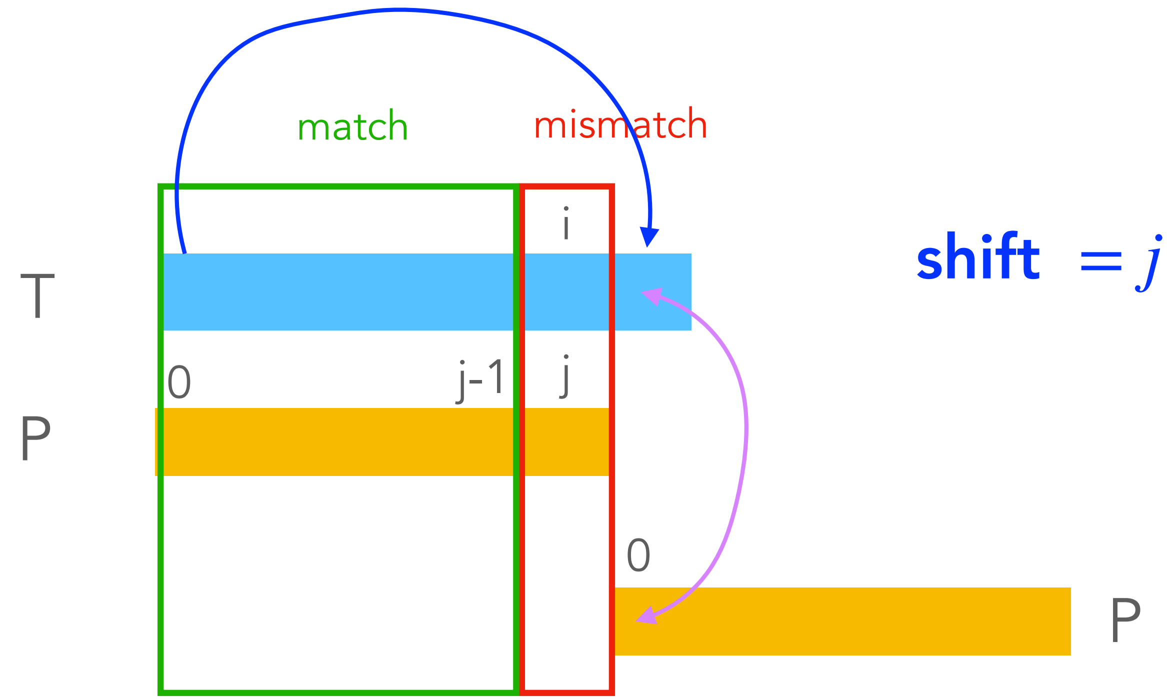
Il **prossimo** confronto interessante è tra

$P[j-s]$ e $T[i]$

String-matching

Sappiamo come determinare il prossimo tentativo di allineamento

Evitiamo di ripetere confronti tra il testo e il pattern di cui conosciamo già l'esito



shift $= j + 1$

Il **prossimo** confronto interessante è tra $P[0]$ e $T[i+1]$

String-matching

Sappiamo come determinare il prossimo tentativo di allineamento

Evitiamo di ripetere confronti tra il testo e il pattern di cui conosciamo già l'esito

shift = s $\leq j$ $\quad P[j-s]$ e $T[i]$ $\quad$ prox indice di `P` $\Rightarrow$ $j-s$

shift = s $= j+1$ $\quad P[0]$ e $T[i+1]$ $\quad$ prox indice di `P` $\Rightarrow$ 0

prox indice di `P` $\quad$ **next =** $\max\{0, j-s\}$

String-matching

ALGORTIMO di KNUTH-MORRIS-PRATT (1975)

```
Knuth-Morris-Pratt(T,P)
 n := |T|
 m := |P|
 shift := Best-Prefix(P)
 for j = 0 to m do
  next[j] := max{0,j - sift[j]}
 i := 0 //indice che scorre T
 j := 0 //indice che scorre P
 while i ≤ m-n do
  while (j < m AND P[j] = T[i+j]) do
   j := j+1
  if j = m then print i
  i := i + shift[j]
  j := next[j]
 print -1
```

String-matching

ALGORTIMO di KNUTH-MORRIS-PRATT

```
Knuth-Morris-Pratt(T,P)
 n := |T|
 m := |P|
 shift := Best-Prefix(P)
 for j = 0 to m do
  next[j] := max{0,j - sift[j]}
 i := 0 //indice che scorre T
 j := 0 //indice che scorre P
 while i ≤ m-n do
  while (j < m AND P[j] = T[i+j]) do
   j := j+1
  if j = m then print i
  i := i + shift[j]
  j := next[j]
 print -1
```

non coinvolge caratteri del testo!

Costo computazionale

contiamo solo i confronti tra caratteri delle due stringhe

$$O(|T|)$$

ogni carattere del testo viene associato ad AL PIÙ DUE confronti con caratteri del pattern:

- un match
- il mismatch che occorre quando il pattern è allineato con il carattere

… nel mondo reale…

Variazioni sul tema:

- String matching **esatto** VS string matching **approssimato**
- **Single pattern** string matching VS **multiple pattern** string matching

Alcune applicazioni (… solo alcune di quelle più famose…):

- Correttore ortografico e operazione "cerca"
- Filtri anti-SPAM
- Antivirus
- Motori di ricerca
- Software anti-plagio
- Bioinformatica e sequenziamento DNA
- Indagini forensi digitali
- Information Retrieval
- ….