Laboratorio di Algoritmi e Strutture Dati I

Esercitazione 4

Martedì 7 aprile


Donald
Knuth '60
50th Reunion

Lezione 4: Il pattern matching

- Una stringa è definita come un array di caratteri $S = s_0, \ldots, s_{n-1}$
- **Reminder:** in C avete bisogno di n+1 caratteri per memorizzare dinamicamente una stringa (perché?)
- Il problema dell'esercitazione odierna riguarda il pattern matching, vale a dire la ricerca di un certo pattern (stringa) all'interno di una stringa
- A lezione avete visto l'algoritmo "ingenuo" che, sostanzialmente, prevede un controllo di ogni singolo carattere del pattern con ogni singolo carattere della stringa principale. La sua complessità è $O(n \times m)$
- Tra i differenti algoritmi che affrontano questo problema, l'algoritmo di **Knuth, Morris and Pratt** migliora la complessità portandola a $O(n+m)$
- Come funziona?

Lezione 4: Knuth, Morris and Pratt

- L'idea è quella di evitare confronti inutili:

- Se non viene trovata un'occorrenza di un pattern nella stringa al passo *i*, si sfruttano comunque i confronti fatti fino a quel momento;

Esempio:

Stringa: ABCABCDABD

Pattern: ABCDAB

Quando si arriva ad analizzare “D” (pat) con “A” (str), ci si rende conto che:

1. Sicuramente il matching non inizia nei primi 3 caratteri della stringa.

2. Non ha senso riprendere il confronto a partire da B, nella stringa, perché è già noto che "BC" (str) != “AB” (pat) dal confronto precedente.

Lezione 4: Knuth, Morris and Pratt

- Supponiamo di avere la seguente stringa:

A C A C A C A C A C A G T

- E di volere verificare se in essa è contenuto il seguente pattern:

A C A C A G T

- Abbiamo bisogno di una funzione chiamata funzione **insuccesso** come ausilio per il pattern matching tramite KMP.

Lezione 4: KMP, funzione insuccesso

$$f(j) = \begin{cases} \text{massimo valore di } i \ (i < j) \text{ tale che } p_0 p_1 \ldots p_i = p_{j-i} p_{j-i+1} \ldots p_j \\ -1 \end{cases}$$

- Si calcolano tutti i possibili **prefissi** del pattern
- Per ogni prefisso, si cerca il più lungo prefisso che sia anche suffisso
- Si memorizza, per ogni prefisso, la posizione dell'ultima lettera del prefisso valido.

Prefissi del pattern
A C A C A G T

A	-
A C	-
A C A	A
A C A C	A C
A C A C A	A C A
A C A C A G	-
A C A C A G T	-

Prefissi che siano anche suffissi

posizione	0	1	2	3	4	5	6
pattern	A	C	A	C	A	G	T
insuccesso	-1	-1	0	1	2	-1	-1

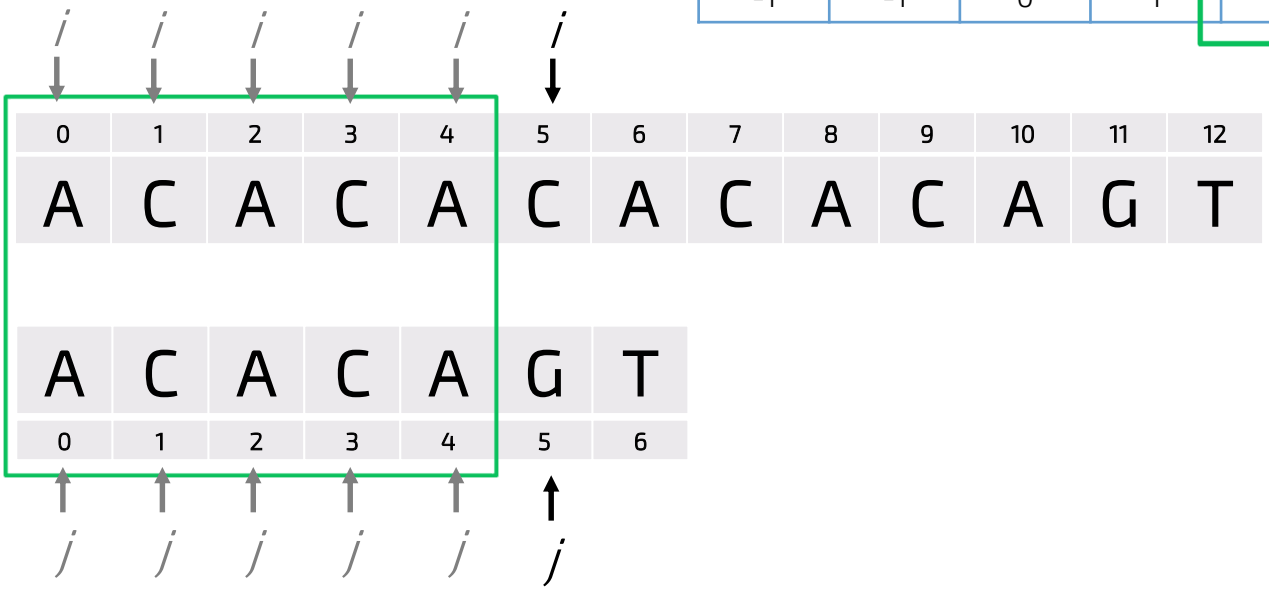
Lezione 4: KMP, funzionamento logico

Effettuiamo il pattern matching con la funzione **KMP_match**, avvalendoci dell'array prodotto con la funzione insuccesso per capire da quale posizione è conveniente ripartire in caso di mismatch.

Utilizziamo l'indice ***i*** per scorrere la stringa e l'indice ***j*** per scorrere il pattern.

- Se ***str[i] = pat[j]*** incrementiamo **i** (indice stringa) e **j** (indice pattern)
- Se ***str[i] != pat[j]*** allora **j ← insuccesso[j-1]+1**

0	1	2	3	4	5	6
A	C	A	C	A	G	T
-1	-1	0	1	2	-1	-1



Mismatch*: j ← insuccesso[j-1]+1* quindi j riparte dalla posizione 3 del pattern

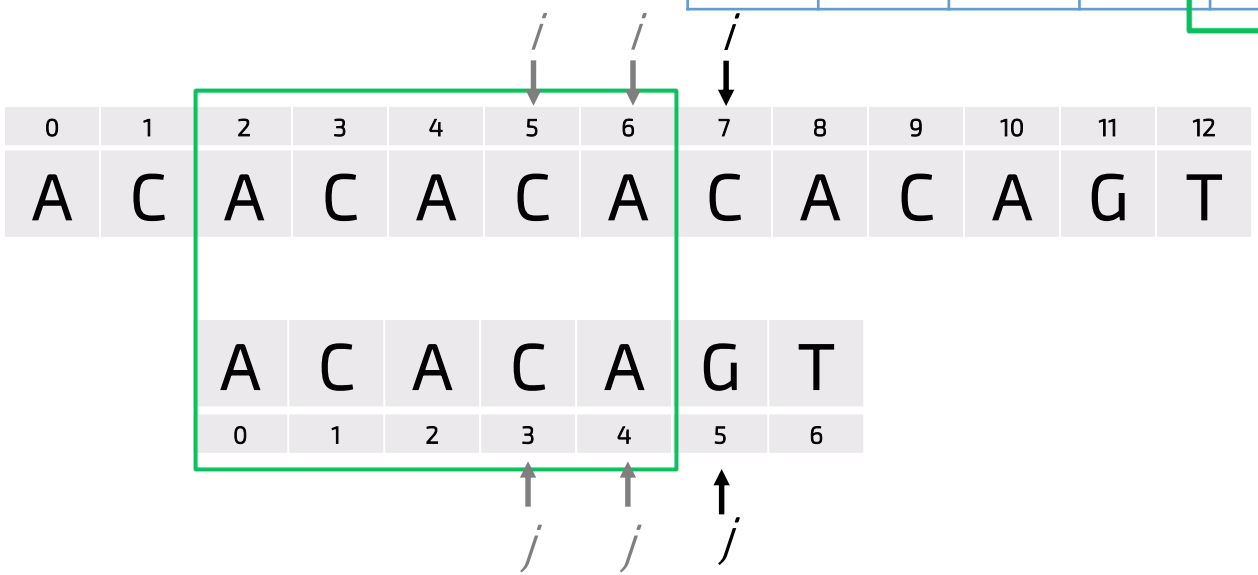
Lezione 4: KMP, funzionamento logico

Effettuiamo il pattern matching con la funzione KMP_match, avvalendoci dell'array prodotto con la funzione insuccesso per capire da quale posizione è conveniente ripartire in caso di mismatch.

Utilizziamo l'indice i per scorrere la stringa e l'indice j per scorrere il pattern.

- Se *str[i] = pat[j]* incrementiamo **i** e **j**
- Se *str[i] != pat[j]* allora **j** ← **insuccesso[j-1]+1**

0	1	2	3	4	5	6
A	C	A	C	A	G	T
-1	-1	0	1	2	-1	-1



- Mismatch: *j* ← *insuccesso[j-1]+1* quindi ***j*** riparte dalla posizione 3 del pattern

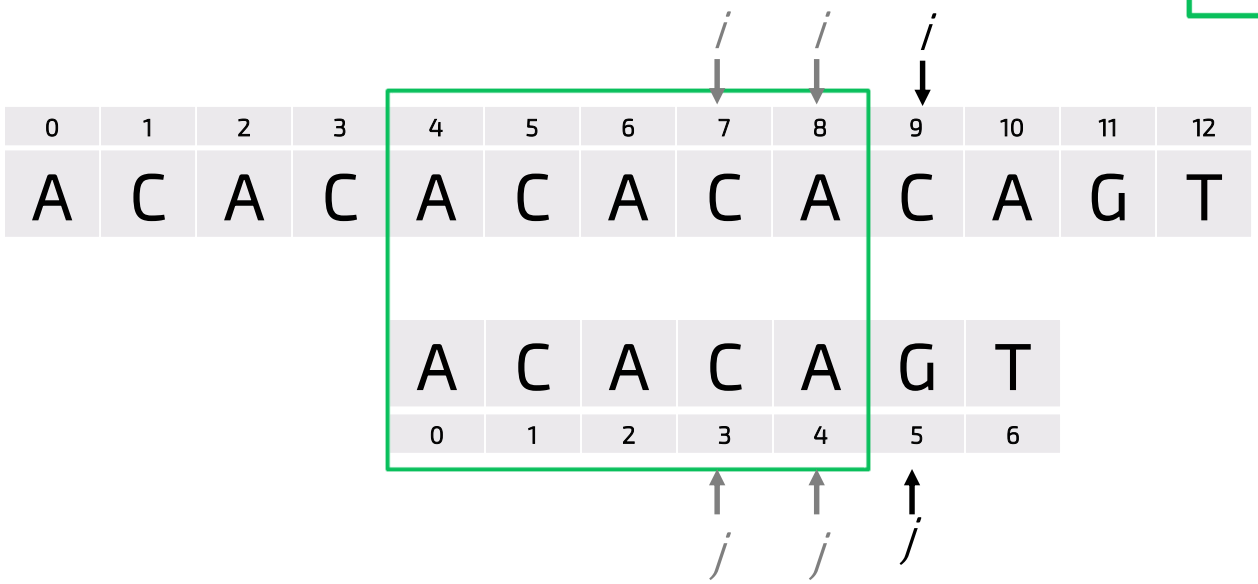
Lezione 4: KMP, funzionamento logico

Effettuiamo il pattern matching con la funzione KMP_match, avvalendoci dell'array prodotto con la funzione insuccesso per capire da quale posizione è conveniente ripartire in caso di mismatch.

Utilizziamo l'indice i per scorrere la stringa e l'indice j per scorrere il pattern.

- Se *str[i] = pat[j]* incrementiamo **i** e **j**
- Se *str[i] != pat[j]* allora **j** ← **insuccesso[j-1]+1**

0	1	2	3	4	5	6
A	C	A	C	A	G	T
-1	-1	0	1	2	-1	-1



- Mismatch: *j* ← *insuccesso[j-1]+1* quindi ***j*** riparte dalla posizione 3 del pattern

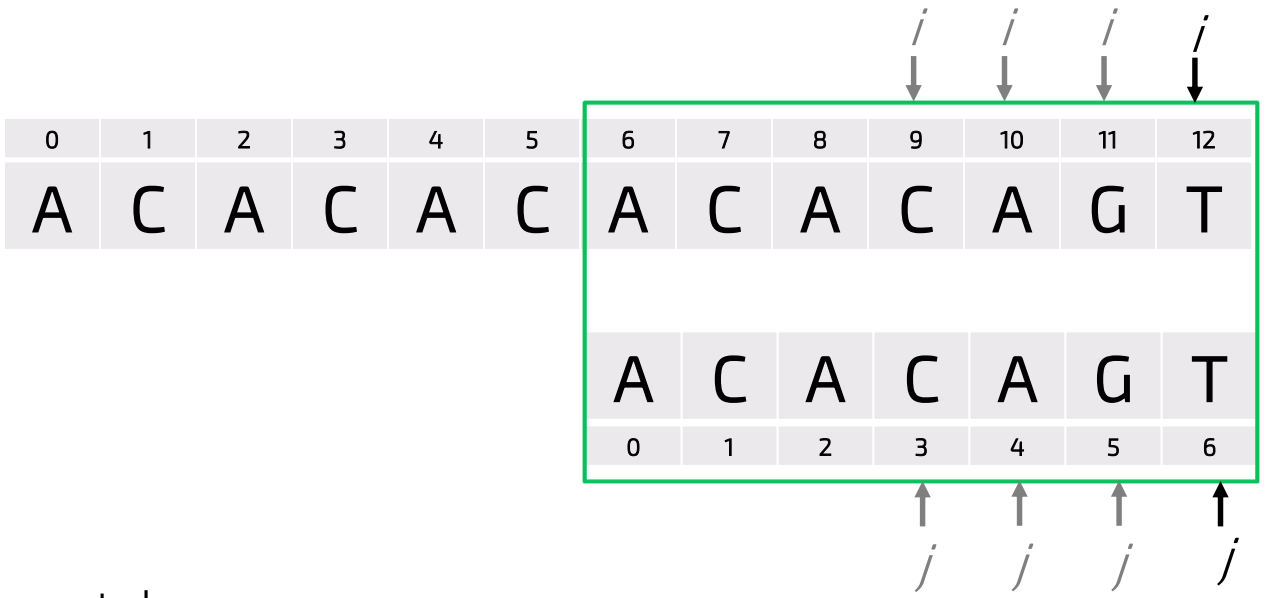
Lezione 4: KMP, funzionamento logico

Effettuiamo il pattern matching con la funzione KMP_match, avvalendoci dell'array prodotto con la funzione insuccesso per capire da quale posizione è conveniente ripartire in caso di mismatch.

Utilizziamo l'indice i per scorrere la stringa e l'indice j per scorrere il pattern.

- Se *str[i] = pat[j]* incrementiamo **i** e **j**
- Se *str[i] != pat[j]* allora **j** ← **insuccesso[j-1]+1**

0	1	2	3	4	5	6
A	C	A	C	A	G	T
-1	-1	0	1	2	-1	-1



- Match trovato!

Esercizio 4_1: Funzione insuccesso

Implementare la funzione insuccesso (che utilizzeremo per creare l'array di appoggio per il KMP).

```
algoritmo insuccesso (Stringa pat) → intero *

n ← lunghezza di pat
alloca dinamicamente il vettore insuccesso (insucc)

insucc[0] ← -1

// analisi dei prefissi
for j ← 1  to  n-1  do
    i ← insucc[j-1]
    while (pat[j] ≠ pat[i+1] and i≥0) do
          i ← insucc[i]

    if (pat[j] == pat[i+1]) then
          insucc[j] ← i+1
    else
          insucc[j] ← -1

return  insucc
```

Esercizio 4_2: Knuth, Morris and Pratt

Implementare l'algoritmo di Knuth, Morris e Pratt per risolvere il problema del pattern matching (utilizzando l'array creato con la funzione insuccesso).

```
algoritmo KMP_match (Stringa stringa, Stringa pat) → intero

insucc ← insuccesso(pat)// array  insuccesso
i ← 0
j ← 0
lenS ←  lunghezza di stringa
lenP ←  lunghezza di pat

while (i < lenS and j < lenP) do
        if (stringa[i] == pat[j]) then
                    i ← i+1
                    j ← j+1
        else  if (j == 0) then
                    i ← i+1
        else
                    j ← insucc[j-1]+1

if  (j == lenP)  then
        return  i-lenP
else return  -1
```

Esercizio 4_3: KMP, rotazioni cicliche

Utilizzare tale metodo per creare un nuovo algoritmo, con tempo lineare, per determinare se un pattern P è una rotazione ciclica di un'altra stringa S.

Per esempio, data la stringa **abcde**, le stringhe **deabc** e **eabcd** sono sue rotazioni cicliche.

NB#1: stringa e pattern devono avere la stessa lunghezza
NB#2: l'utilizzo dell'array insuccesso rimane tale e quale!
NB#3: non potete duplicare la stringa
NB#4: basta modificare KMP_match

Lezione 4: Ricapitolando

Implementare:

- Es 4_1: Funzione Insuccesso
- Es 4_2: Funzione KMP_match
- Es 4_3: Funzione KMP_match modificata (rotazione ciclica)

end().