Strutture Dati

Lezione 5
La stringa

Oggi parleremo di …

- Tipo di dati astratto *Stringa*
  - specifica
  - rappresentazione
- Ricerca di stringhe
  - algoritmo banale
  - algoritmo di Knuth, Morris e Pratt (1977)

Il tipo di dati astratto *Stringa*

- Una **stringa** è definita come una sequenza $S=s_0,\ldots,s_{n-1}$, dove s_i sono i caratteri che appartengono all'insieme dei caratteri del linguaggio di programmazione.

  Se $n=0$, allora S è una stringa nulla o vuota.

Il tipo di dati astratto *Stringa*

```
Struttura Stringa
   oggetti: un insieme finito di zero o più caratteri

   funzioni: per ogni s,t ∈ Stringa, i, j, m ∈ interi non negativi

   Stringa Null(m)        ::=  ritorna una stringa la cui lunghezza
                               massima è pari a m caratteri, inizialmente
                               impostata a NULL. La stringa NULL è
                               identificata da "".
   Intero Confronta(s,t)  ::=  if (s è uguale a t) return 0, else if (s
                               precede t) return -1, else return +1
   Booleano E_Null(s)     ::=  if (Confronta(s,NULL)) return FALSE, else
                               return TRUE
   Intero Lunghezza(s)    ::=  if (Confronta(s,NULL)) return il numero di
                               caratteri di s, else return 0
   Stringa Concat(s,t)    ::=  if (Confronta(t,NULL)) return una stringa
                               i cui elementi sono quelli di s seguiti da
                               quelli di t, else return s
   Stringa Substr(s,i,j)  ::=  if ((j>0)&&(i+j-1)<Lunghezza(s)) return la
                               stringa contenente i caratteri di s nelle
                               posizioni i, i+1, …, i+j-1, else return
                               NULL
end Stringa
```

Il tipo di dati *Stringa*

- Le stringhe vengono rappresentate come *array* di caratteri che terminano con il carattere nullo \0.
- **Esempio**:
  - #define MAX_LUN 30
  - char s[MAX_LUNG] = {"casale"};
  - char p[MAX_LUNG] = {"libro"};
  - char stringa1[MAX_LUNG], *s = stringa1;
  - char stringa2[MAX_LUNG], *p = stringa2;

Il pattern matching

- Ricercare l'occorrenza di una particolare sottostringa, *pat*, in un'altra stringa, *stringa*.
- Il problema può essere risolto con un algoritmo banale che richiede, nel caso pessimo, un numero di operazioni proporzionale a $n \times m$.
- Il problema può essere risolto in modo molto semplice anche in tempo $n+m$.

Algoritmo banale

- Si esaminano in sequenza i singoli caratteri di una stringa finché non viene trovata la serie di caratteri ricercata o finché non viene raggiunta la fine della stringa.
- Se *pat* non si trova nella *stringa*, tale metodo ha un tempo di esecuzione pari a $O(n \times m)$, dove n è la lunghezza di *pat* ed m è la lunghezza della *stringa*.

```
int ntrova(char *stringa, char *pat)
{
  int i,j;
  int lens = strlen(stringa);
  int lenp = strlen(pat);

  for(i=0, j=0; i<lens && j<lenp; i++, j++)
      while (stringa[i]!=pat[j]) {i = i -(j-1); j=0; }

  if(j == lenp) return i-lenp;
  else  return -1;
}
```

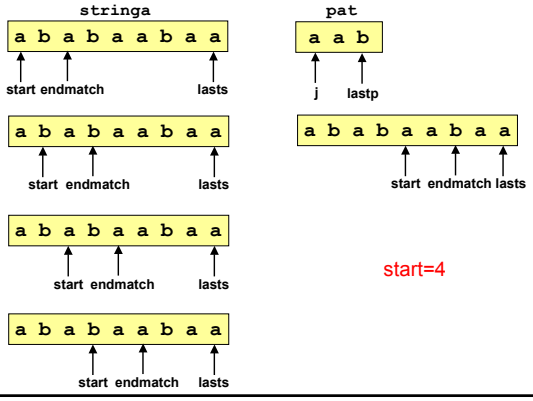
Algoritmo banale "migliorato"

- Possiamo interrompere le operazioni quando la lunghezza di *pat* è maggiore del numero di caratteri rimasti nella *stringa*.
- Possiamo controllare il primo e l'ultimo carattere di *pat* e di *stringa* prima di verificare i caratteri restanti.

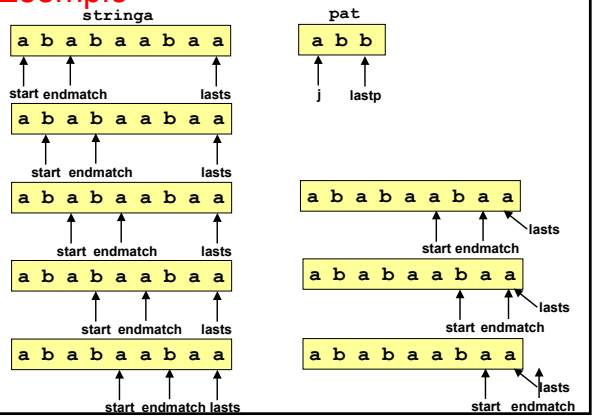
```
int ntrova(char *stringa, char *pat)
{
  int i,j, start=0;
  int lasts = strlen(stringa) - 1;
  int lastp = strlen(pat) - 1;
  int endmatch = lastp;

  for(i=0; endmatch<=lasts; endmatch++, start++)
    {
      if(stringa[endmatch] == pat[lastp])
           for(j=0, i=start; j<lastp && stringa[i]==pat[j]; i++, j++) ;
      if(j == lastp) return start;
     }
  return -1;
}
```

$O(n \times m)$

Esempio



Esempio



Algoritmo di Knuth, Morris e Pratt

- Perché spostarci all'indietro ogni volta che si verifica una diversità?
- Perché non usare la conoscenza dei caratteri di *pat* e della posizione in cui si è verificata la diversità per determinare il punto in cui continuare la ricerca?
- Dopo aver riconosciuto j -1 caratteri di *pat* a partire da una certa posizione i nella *stringa* ed aver fallito al j-esimo, perché tornare indietro di j - 2 posizioni nella *stringa*?
- I j -1 caratteri già riconosciuti fanno parte di *pat* e sono noti addirittura prima di iniziare la ricerca nella *stringa*!
- Perché non trarre vantaggio da questa informazione nota in anticipo?

Algoritmo di Knuth, Morris e Pratt

Esempio: Siano

S = 10110001010110101011011

P = 10110110

I primi 5 caratteri di *P* sono uguali ai primi 5 caratteri di *S*, ma il sesto è diverso.

Gli indici sono *start*=5 e *j*=5.

Dovremmo traslare a destra di una posizione *P* rispetto a *S* cioè ripartire con *start*=1 e *j*=0:

S = 10110001010110101011011

P = 10110110

Ma nella sequenza 10110 di *P* che è stata riconosciuta

- i primi 4 caratteri non coincidono con gli ultimi 4
- i primi 3 caratteri non coincidono con gli ultimi 3
- I primi 2 caratteri coincidono con gli ultimi 2: 10110!

E' inutile ripartire con *start*=1 o *start*=2, ma conviene ripartire direttamente con *start*=3.

Poiché i successivi due caratteri di *P* coincidono con quelli di *S*, tanto vale ripartire con *start*=5 e *j*=2.

S = 10110001010110101011011

P = 10110110

Algoritmo di Knuth, Morris e Pratt

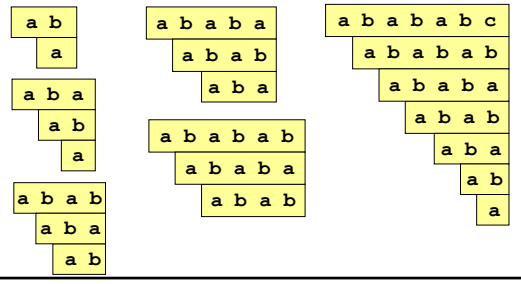
- Dato un pattern $P = p_0p_1\ldots p_{n-1}$ si definisce la seguente funzione insuccesso:

$$f(j) = \begin{cases} \text{massimo valore di } i\ (i < j) \text{ tale che } p_0p_1\ldots p_i = p_{j-i}p_{j-i+1}\ldots p_j \\ -1 \end{cases}$$

- Se si verifica una corrispondenza parziale tale che $s_{i-j}\ldots s_{i-1} = p_0p_1\ldots p_{j-1}$ e $s_i \neq p_j$, allora si riprende la ricerca confrontando s_i con $p_{f(j-1)+1}$ se $j \neq 0$.
- Se $j = 0$, si continua la ricerca confrontando s_{i+1} con p_0.

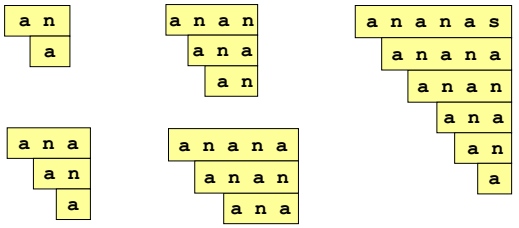
Funzione insuccesso

0	1	2	3	4	5	6	7	
a	b	a	b	a	b	c	b	*pat*
-1	-1	0	1	2	3	-1	-1	*f*



Funzione insuccesso

0	1	2	3	4	5	
a	n	a	n	a	s	*pat*
-1	-1	0	1	2	-1	*f*



Algoritmo di Knuth, Morris e Pratt

```
int pmatch(char *stringa, char *pat)
{
 int i=0, j=0;
 int lens = strlen(stringa);
 int lenp = strlen(pat);

 while(i<lens && j<lenp)
 {
  if(stringa[i] == pat[j]) { i++; j++; }
  else  if (j==0) i++;
        else  j=insuccesso[j-1]+1;
 }
 return ( (j==lenp) ? (i-lenp) : -1);
}
```

$O(m) = O(length(stringa))$

```
void insucc(char *pat)
{
 /* Calcola la funzione insuccesso per
    un pattern (pat) */

 int i,j;
 int n = strlen(pat);

 insuccesso[0] = -1;
 for(j=1; j<n; j++)
  {
   i=insuccesso[j-1];
   while((pat[j]!=pat[i+1]) && (i>=0))
         i = insuccesso[i];
   if (pat[j] == pat[i+1])
           insuccesso[j]=i+1;
   else    insuccesso[j] = -1;
   }
}
```

$O(n) = O(length(pat))$

Il tempo di esecuzione dell'intero processo è $O(n + m)$