Algoritmi di Pattern Matching

Laboratorio di Algoritmi e Strutture Dati - Lezione 7

Giuditta Franco

Dipartimento di Informatica,
Università di Verona

4 Marzo 2008

Talk Outline

1. Stringhe (testi) e sottostringhe (“pattern”) in Java.
2. Problema del pattern matching - tre algoritmi di “corrispondenza tra stringhe”.
3. Algoritmo Naive
4. Algoritmo di Boyer-Moore
5. Algoritmo di Knuth-Morris-Pratt

Richiami di Terminologia

Data una stringa P di m caratteri, si dice **sottostringa** $P[i \ldots j]$ la stringa della forma $P[i]P[i+1] \ldots P[j]$, con $0 \leq i \leq j \leq m-1$.

Se $i = 0$, $P[i \ldots j]$ è un **prefisso** di P, se $j = m-1$, $P[i \ldots j]$ è un **suffisso** di P, se $i > j$, $P[i \ldots j]$ è la **stringa nulla** λ, di lunghezza 0, che è sia un prefisso che un suffisso.

L'insieme A dei caratteri si dice **alfabeto**, che di solito si assume finito e di cardinalità fissata.

Java definisce una classe **String** per rappresentare stringhe immutabili (che non possono essere modificate), e la classe **StringBuffer** per rappresentare le stringhe mutabili (che possono essere modificate).

Classe String

String estende **Object** e implementa **Comparable**.

Operazioni tipiche su una stringa S:

- (int) **length()** e (char) **charAt(int index)** - lunghezza di S e carattere con indice index.
- (boolean) **startsWith(String prefix)** e **endsWith(String suffix)** - testano se prefix o suffix sono un prefisso o un suffisso di S, rispettivamente.
- (String) **substring(int beginIndex, int endIndex)** - restituisce S[beginIndex, endIndex].

Classe String

- (String) **concat(String str)** - restituisce la concatenazione di S e *str* (senza alterare *S*).
- (boolean) **equals(Object anObject)** e **equalsIgnoreCase(String anotherString)** - testano se anObject o anotherString sono uguali a S, nel secondo caso ignorando le differenze di maiuscola/minuscola.
- (int) **indexOf(String str)** - restituisce l'indice in *S* della prima occorrenza della sottostringa *str*, se esiste, altrimenti restituisce -1.

Questi metodi, tranne l'ultimo (lez di oggi), si implementano facilmente rappresentando le stringhe come un array di caratteri.

Classe StringBuffer

StringBuffer estende **Object**. Rappresenta una “mutabile” sequenza di caratteri - una StringBuffer è come una String, ma può essere modificata tramite chiamate di metodi.

Metodi tipici sono su una stringa StringBuffer *S*:

- (StringBuffer) **append(Type Q)** - sostituisce S con la concatenazione di S e Q. Type può essere di tipo int, double, float, Object, String, StringBuffer, CharSequence - in ogni caso si concatena la rappresentazione come stringa del parametro *Q*.

- (StringBuffer) **insert(int offset, String str)** aggiorna (e poi restituisce) *S* inserendo la stringa *str* a partire dall’indice *i*.

Interface CharSequence

```
public interface CharSequence{

char charAt(int index);
```
Returns the char value at the specified index.

```
int length();
```
Returns the length of this character sequence.

```
CharSequence subSequence(int start, int
end);
```
Returns a new CharSequence that is a subsequence of this sequence.

```
String toString(); }
```
Returns a string containing the characters in this sequence in the same order as this sequence.

Classe StringBuffer

Altri metodi tipici sono su una stringa StringBuffer *S*:

- (StringBuffer) **reverse()** rovescia (e restituisce) *S*.
- void **setCharAt(int index, char ch)** Imposta a *ch* il carattere corrispondente all'indice index in *S*.
- (char) **charAt(int index)** restituisce il carattere di *S* all'indice *i*.

Fatta eccezione per questo ultimo, i metodi di String non si possono applicare ad un oggetto StringBuffer - senza prima avergli applicato il metodo toString() che restituisce una versione String di *S*.

Per rendere una String modificabile si può usare il costruttore **StringBuffer(String str)**.

Problema del pattern matching

Data una stringa di testo T di lunghezza n e una stringa modello (pattern) P di lunghezza m, è richiesto di verificare se P sia una sottostriga di T.

La corrispondenza ("match") consiste nel verificare che esiste un certo indice i tale che $P = T[i \ldots i + m]$.

L'output di un algoritmo di pattern matching consiste in una segnalazione dell'inesistenza del modello P in T oppure nell'indice in cui ha inizio l'uguaglianza di una sottostringa di T con il modello P: un intero o un insieme di interi.

Tre algoritmi di pattern matching

1. *Algoritmo di forza bruta (Naive), con complessità $O(nm)$ (tempo quadratico)*.
   Ha una finestra che scorre col pattern, e quando passa al controllo successivo si dimentica le informazioni acquisite precedentemente.

2. *Algoritmo di Boyer-Moore (BM), con complessità $O(n + m + |A|)$.*
   Rispetto all'algoritmo Naive, evita di fare dei confronti tra P e una buona parte di T.

3. *Algoritmo di Knuth-Morris-Pratt(KMP), con complessità $O(n + m)$*.
   Migliora ulteriormente le prestazioni mantenendo le informazioni ottenute precedentemente nei dislocamenti falliti.

Algoritmo Naive

L'algoritmo può lavorare anche con alfabeti infiniti.

Si controllano tutte le possibili disposizioni di P (length m) relative a T (length n), con due cicli innestati: quello più esterno scorre tutti i possibili indici di partenza del pattern nel testo, mentre quello più interno scorre i caratteri del pattern confrontandoli con quelli corrispondenti nel testo.

```
for (int i=0; i <= n-m; i++) {
int j=0;
while(j<m &&
text.charAt(i+j)==pattern.charAt(j)){
   j++;
   if(j==m) { return i;}
  }
}
return -1;
```

Algoritmo BM

L'algoritmo assume un alfabeto finito e fissato, ed è molto veloce con un alfabeto piccolo e un pattern breve (ideale per cercare parole in un testo), è molto efficiente.

Rispetto al Naive, aggiunge due euristiche (dette "del carattere discordante" e "del buon suffisso") per risparmiare tempo:

- Si inizia il confronto dalla fine di P e ci si sposta verso l'inizio.

- Se $T[i] = c$ è il primo disaccoppiamento (e.g., su P[j]) durante il controllo di una possibile disposizione di P, si calcola l'ultima occorrenza l di c in P: se si trova a destra di j, si salta di 1, altrimenti di $j - l$ unità.

Caso peggiore per l'algoritmo: $T = a^n$ e $P = ba^{m-1}$.

Un’implementazione dell’algoritmo BM

```
public static int BMmatch (String text,
String pattern) {

int[] last = buildLastFunction(pattern);
int n = text.length();
int m= pattern.length();

int i = m-1;
if (i>n-1)
return -1;  //nessuna corrispondenza se il
//pattern è più lungo del testo

int j = m-1;
```

Implementazione algoritmo BM

```
do{

if(pattern.charAt(j) == text.charAt(i))

if(j == 0)
return i; //corrispondenza
else {//l'euristica procede da destra verso sinistra
i - - ;
j - - ;}

else{//l'euristica del buon suffisso
i=i+m-Math.min(j, 1+last[text.charAt(i)]);
j = m-1;}}while (i<=n-1);

return -1; }//nessuna corrispondenza
```

Metodo buildLastFunction (String pattern)

//Si crea un array indicizzato dal cod ASCII del carattere.

```
public static int[] buildLastFunction
(String pattern) {

int[] last = new int[128]; //si inizializza con tutti
i caratteri ASCII

for (int i = 0; i<128; i++) {
last[i] = -1; //inizializzazione dell'array
}

for (int i = 0; i < pattern.length();
i++){
last[pattern.charAt(i)] = i; //un cast implicito
per il codice ASCII intero
}

return last;
}
```

Algoritmo KMP

L'algoritmo KMP conserva le informazioni ottenute dai confronti precedenti dei matching falliti, e raggiunge la complessità lineare, ottimale nel caso peggiore.

Trucco: Preelabora il pattern P con una funzione fallimento f, che indica uno spostamento proprio di P che sia il più esteso possibile e che riutilizza i confronti effettuati precedentemente, definita come segue:

per j>0, f(j) è la lunghezza del più lungo prefisso di P che sia un suffisso proprio di P[1. . . j], e f(0)=0.

Un'implementazione dell'algoritmo KMP

```
public static int KMPmatch (String text,
String pattern) {

int n = text.length();
int m = pattern.length();
int[] fail = computeFailFunction(pattern);
int i = 0;
int j = 0;

while (i<n){
if (pattern.charAt(j) == text.charAt(i)){
if (j == m-1) return i-m +1; //abbinamento
i++;
j++;}
else if (j>0)
j = fail[j-1];
else i++;
}
return -1; //nessun abbinamento
```

Implementazione della funzione fallimento

```
public static int[] computeFailFunction
(String pattern){

int [] fail = new int[pattern.length()];
fail[0]=0;
int m = pattern.length();

int j = 0;
int i = 1;
while (i<m){
```

Implementazione della funzione fallimento

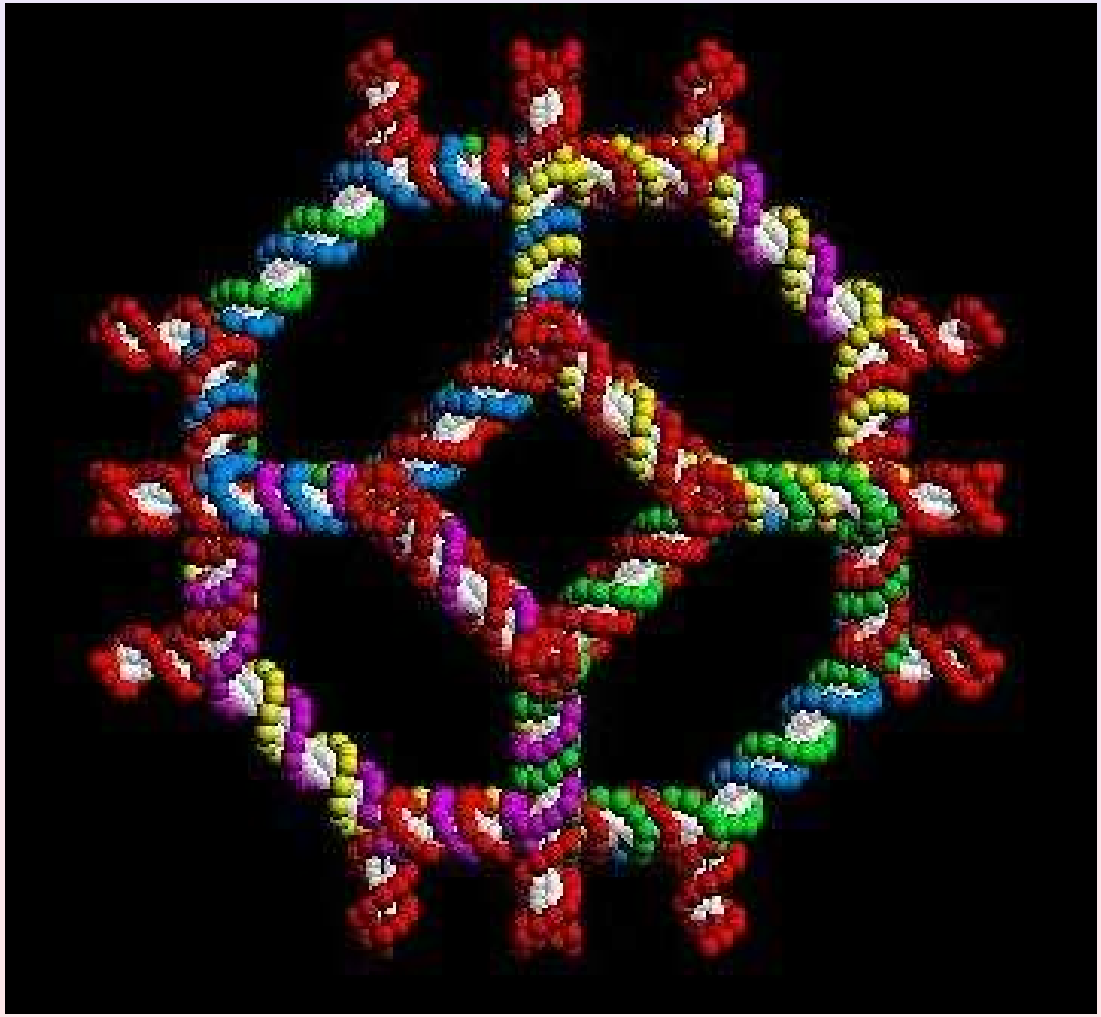
```
if (pattern.charAt(j)==pattern.charAt(i)){
// j + 1 caratteri corrispondenti
fail[i]=j+1;
i++;
j++;}
else

if (j>0) // j segue un prefisso corrispondente
j = fail[j-1];
else{ //nessuna corrispondenza
fail[i]=0;
i++;}
}

return fail;
}
```

Applicazioni su stringhe biologiche

Individuazione di geni sul genoma, ricerca di primers, e DNA design di strutture tridimensionali.



Compito dell'esercitazione

Implementare i tre algoritmi di pattern matching visti a lezione, con un Test per verificarne il funzionamento e le prestazioni.

Javadoc

`@author` va scritto una sola volta, all'inizio, di tutti i metodi vanno commentati con `@param` *tutti* i parametri, nell'ordine, e il `@return`, se c'e'.

Le variabili di classe (non quelle interne ai metodi) vengono listate automaticamente dal javadoc, non vengono commentate con @param, ma vengono precedute nella loro dichiarazione da un commento standard.

Alberi

Per cancellare un nodo da un albero *non* va bene metterne il contenuto a null! Generalmente questo comporta solo l'eliminazione del dato, o peggio, di tutto un sottoalbero.

Rivedere il metodo `deleteNode(BinaryNode daCancellare)` usato dal metodo delete(Object item) spiegato nella quinta lezione, che cancella il nodo contenente un dato oggetto item in un albero binario di ricerca.

Ricorsione

Per richiedere al sottoalbero sinistro e al sottoalbero destro di visualizzare il proprio contenuto si può realizzare, per esempio, l'esecuzione ricorsiva del seguente metodo:

```
public void visualizza(){
if (a !=null) {
a.sx.visualizza();
System.out.println(a.dato);
a.dx.visualizza();
}
}
```