Stringhe e pattern matching

(GT5 12.1, 12.3)

Stringhe

- Un stringa è una sequenza di caratteri appartenenti a un alfabeto
- Un alfabeto Σ è l'insieme dei caratteri utilizzato da una famiglia di stringhe, all'interno di un particolare problema
  - Solitamente si usano alfabeti noti a priori e di dimensione finita, $|\Sigma|$
  - Es. l'insieme dei caratteri ASCII, l'insieme dei caratteri Unicode, l'insieme delle cifre binarie {0, 1}, l'insieme {A, C, G, T} nella descrizione delle sequenze di DNA
- La lunghezza o dimensione di una stringa P è il numero di caratteri che la compongono e si indica con $|P|$
  - Le posizioni all'interno di una stringa si numerano solitamente a partire da 0, mediante un indice
    - Più raramente a partire da 1, mediante un rango
    - Il carattere in posizione i nella stringa P si indica con $P[i]$

Sottostringhe

- Data una stringa P di dimensione m si definisce **sottostringa** $S(P, i, j) = P[i]\ldots P[j] \subseteq P$ la sotto-sequenza dei suoi caratteri **consecutivi** a partire da quello in posizione i fino a quello in posizione j, **entrambi compresi**
  - Invece di $P[i]\ldots P[j]$ si può usare la notazione $P[i\ldots j]$
  - Naturalmente deve essere $0 \leq i \leq j \leq m - 1$
  - $|S(P, i, j)| = j - i \mathbf{+ 1}$
  - Se $i = 0$ e $j = m - 1$, la sottostringa coincide con P, altrimenti è una sottostringa propria
  - Se $i > j$, si assume che la notazione sia comunque valida e che rappresenti la sottostringa vuota
  - Se $i = 0$, S è un **prefisso** di P e a volte si scrive $P[\ldots j]$
  - Se $j = m - 1$, S è un **suffisso** di P e a volte si scrive $P[i\ldots]$

Stringhe in Java

- Ricordiamo che la libreria standard di Java fornisce due classi per rappresentare stringhe
  - La classe **`String`**, i cui esemplari sono immutabili
  - La classe **`StringBuffer`**, i cui esemplari sono modificabili
- Entrambe usano un indice (e non un rango) per rappresentare le posizioni dei caratteri
- Usano lo standard Unicode come alfabeto
- Nella rappresentazione di sottostringhe usano una convenzione diversa: il secondo indice rappresenta la posizione del primo carattere che NON fa parte della sottostringa (la differenza tra gli indici è la lunghezza della sottostringa)

Il problema del *pattern matching*

- Data una stringa *T* (un "testo") e una stringa *P* (un "pattern", cioè "modello" o "schema ricorrente"), il problema del **pattern matching** ("trovare una corrispondenza tra il pattern e il testo") si può enunciare in vari modi, tutti utili in problemi diversi
  - Esiste una sottostringa di *T* che sia uguale a *P* ?
    - Metodo **`contains`** di **`String`**
  - Se esiste una sottostringa di *T* che è uguale a *P*, qual è l'indice del suo primo carattere in *T* ?
    - Metodo **`indexOf`** di **`String`**
  - Trovare gli indici del primo carattere di tutte le sottostringhe di *T* che sono uguali a *P*
    - Ripetute invocazioni del metodo **`indexOf`** di **`String`**

Brute force pattern matching

- Algoritmo "a forza bruta" (*brute force algorithm*)
  - A partire da qualsiasi possibile indice i in T, verifica se $S(T, i, i + |P| - 1) = P$
    - Per verificare se due stringhe sono uguali bisogna, ovviamente, controllare tutte le loro coppie di caratteri posti in posizioni corrispondenti
  - Se si ferma alla prima corrispondenza trovata, risolve i primi due problemi; se prosegue, risolve anche il terzo problema

Brute force pattern matching

- BruteForcePatternMatch(T, P)

  $m = |P|$

  for each $i \in [0, |T| - m]$ // no match se $|T[i\ldots]| < m$

      $j = 0$

      while $j < m$ and $T[i + j] = P[j]$

          $j{+}{+}$

      if $j = m$

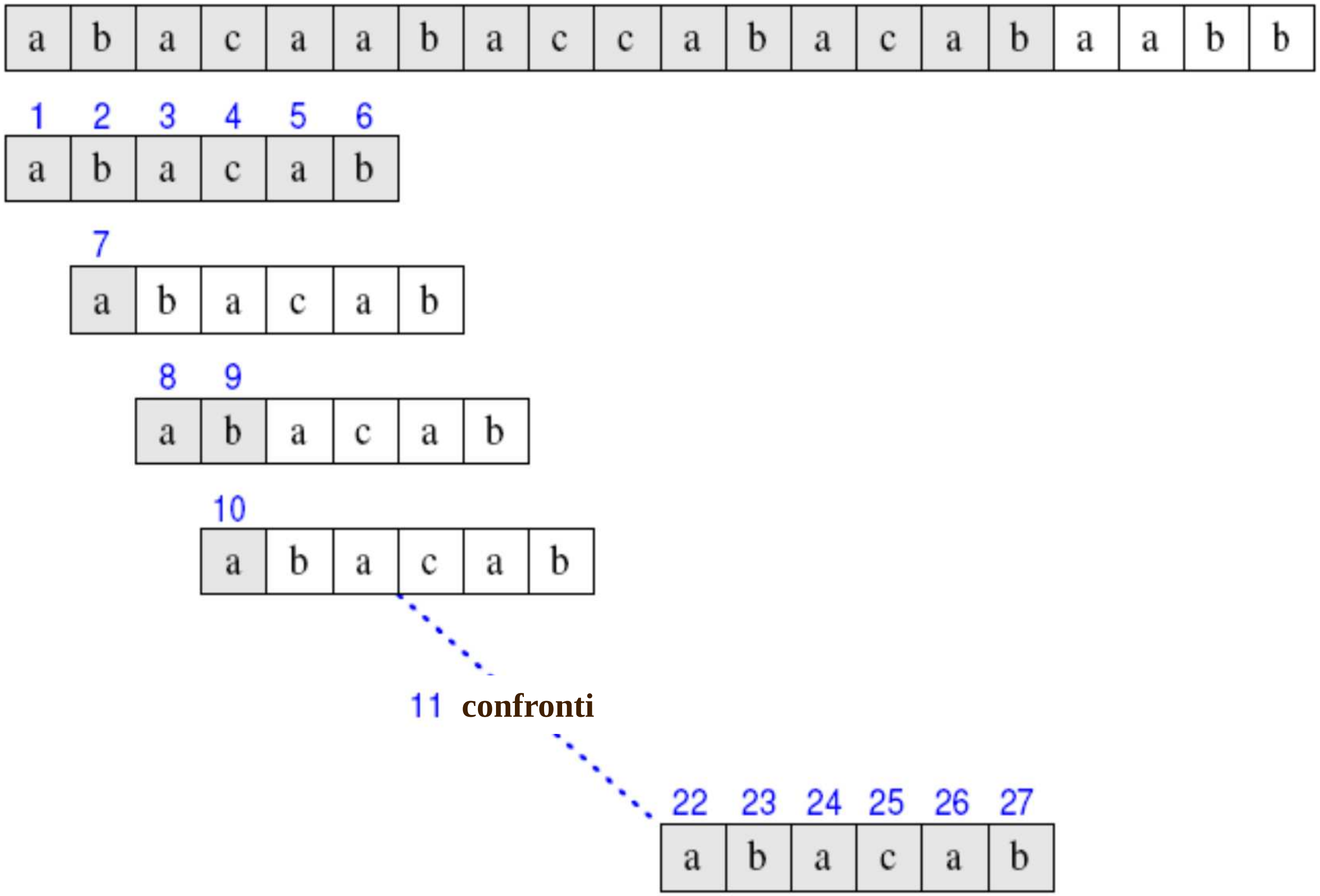
          return i // trovato un match: $P = T[i]\ldots T[i + m - 1]$

                  // oppure aggiunge i alla lista dei risultati e
                  // prosegue, per trovare tutti i match

  return -1 // no match, $P \not\subset T$

Brute force pattern matching

- Algoritmo "a forza bruta" (*brute force algorithm*)



Brute force pattern matching

- Algoritmo "a forza bruta" (*brute force algorithm*): prestazioni
  - $n = |T|$, $m = |P|$
- Caso peggiore, $P \not\subset T$ e, per ogni possibile posizione iniziale di P in T, bisogna confrontare m caratteri per scoprire che proprio l'ultima coppia è la prima ad avere due caratteri diversi
  - Numero di confronti $m(n - m + 1) = \mathbf{O(n\ m)}$
  - Esempio: T = AAAAAAAAAA, P = AAAB
- Nel caso peggiore, gli algoritmi di pattern matching sono $\Omega(n + m)$
  - Bisogna almeno ispezionare tutti i caratteri di T e di P
  - C'è ampio margine di miglioramento!
  - È un problema MOLTO studiato, ha grande interesse pratico, scientifico e industriale
  - Vediamo due algoritmi molto utilizzati

Pattern matching: ottimizzazioni

- Per migliorare le prestazioni degli algoritmi di pattern matching si usano diverse strategie, che partono comunque dalla stessa idea di base
  - Per ogni **possibile** posizione iniziale i in T, verifica se $P = P[0]\ldots P[m-1] = T[i]\ldots T[i+m-1]$
- Tali strategie hanno tutte l'obiettivo, in pratica, di **ridurre il numero di posizioni iniziali** in T per le quali occorre verificare la presenza degli m caratteri consecutivi di P in m posizioni consecutive di T
  - L'algoritmo a forza bruta non mette in atto alcuna strategia di riduzione

Pattern matching: Alg. di Boyer-Moore

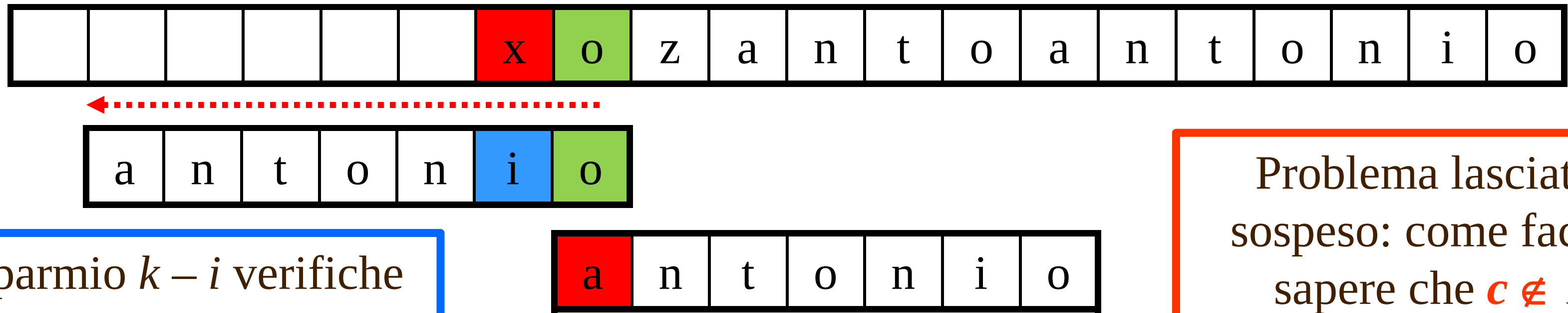
- Esiste in più varianti, vediamo la più semplice
  - Ha ancora prestazioni $O(n\ m)$ nel caso peggiore, ma ha **prestazioni mediamente molto buone in casi pratici di grande interesse**, come la ricerca di parole all'interno di testi (ad esempio in lingua inglese)
- Impiega due strategie ***euristiche*** (cioè che "spesso" migliorano le prestazioni, ma non è garantito che lo facciano sempre, per cui le prestazioni di caso peggiore non cambiano)
  - Looking-glass heuristic
    - Si guarda il testo "allo specchio", cioè a rovescio…
  - Character-jump heuristic
    - **In determinate situazioni**, si "saltano" alcune posizioni iniziali, ottenendo un miglioramento delle prestazioni
      - Se tali situazioni non si verificano, non c'è alcun miglioramento

Pattern matching: Alg. di Boyer-Moore

- Looking-glass heuristic (si guarda il testo "allo specchio", cioè a rovescio…)
  - Questa strategia non porta alcun miglioramento in sé, serve soltanto a rendere più efficace la seconda strategia
    - Si veda commento al termine della presentazione dell'algoritmo
  - Nel confrontare $P = P[0]\ldots P[m-1]$ con $T[i]\ldots T[i+m-1]$ si procede a ritroso, cioè per indici decrescenti, confrontando prima $P[m-1]$ con $T[i+m-1]$
  - Finché si trovano coppie di caratteri uguali, si procede
    - Dopo aver confrontato m coppie consecutive con successo, ovviamente si dichiara di aver trovato un match a partire dalla posizione i in T
    - Se, invece, si trova una coppia di caratteri diversi, entra in gioco la seconda euristica (**character-jump heuristic**), che cerca di "saltare" in avanti e di **non** iniziare la successiva verifica della presenza di un match a partire dalla posizione $i + 1$, bensì a partire da una posizione di indice maggiore

Pattern matching: Alg. di Boyer-Moore

- Character-jump heuristic
  Sia $c = T[k] \neq P[k - i]$ durante la verifica di $P = T[i]\ldots T[i + m - 1]$
- Ci sono tre casi da considerare
  - Se $c \notin P$, allora la prossima ricerca può iniziare da $i = k + 1$ anziché da $i + 1$, perché nessuna posizione iniziale di indice inferiore potrà dar luogo a un match



Risparmio $k - i$ verifiche di posizionamenti

Problema lasciato in sospeso: come faccio a sapere che $c \notin P$? Vedremo

  - In questo esempio, la successiva verifica prevede di posizionare P a partire dal carattere di T successivo a $T[k] = c =$ **'x'**, perché **'x'** $\notin$ $P =$ **"antonio"**, quindi nessun posizionamento che preveda il confronto tra **'x'** e un carattere di P può avere successo

Pattern matching: Alg. di Boyer-Moore

- Character-jump heuristic
  Sia $\boldsymbol{c = T[k] \neq P[k-i]}$ durante la verifica di $P = T[i]\ldots T[i+m-1]$
- Se $\boldsymbol{c \in P}$, ci sono due ulteriori casi da distinguere, in base alla posizione più a destra in P che contiene c : sia h tale posizione
  - h = lastIndexOf(c, P) // usando il nome del metodo di **`String`**
  - Se $\boldsymbol{h < k - i}$, cioè se la posizione più a destra in cui si trova c in P è a sinistra del punto in cui, in P, ho trovato un carattere diverso da c, allora la successiva verifica può partire da una posizione i che renda $T[k] = P[h]$

						o	i	o	t	i	o	i	o	n	t	o	n	i	o

a	n	t	o	n	i	o

a	n	t	o	n	i	o

Risparmio $k - i - h$ verifiche di posizionamenti

Problema in sospeso: $\boldsymbol{c \in P}$? lastIndexOf ?

- Avendo trovato una lettera **'t'** in T e sapendo che, in P, la **'t'** più a destra è quella evidenziata, sarebbe inutile verificare il posizionamento di P spostato di una sola posizione, perché, in corrispondenza della **'t'** di T, non ci sarebbe una **'t'** in P

Pattern matching: Alg. di Boyer-Moore

- Character-jump heuristic
  Sia $c = T[k] \neq P[k - i]$ durante la verifica di $P = T[i] \ldots T[i + m - 1]$
- Se $c \in P$, ci sono due ulteriori casi da distinguere, in base alla posizione più a destra in P che contiene c : sia h tale posizione
  - h = lastIndexOf(c, P) // usando il nome del metodo di **`String`**
  - Se $h > k - i$ (non può essere $h = k - i$, altrimenti i due caratteri sarebbero uguali), cioè se la posizione più a destra in cui si trova c in P è a destra del punto in cui, in P, ho trovato un carattere diverso da c, mi trovo nell'unica situazione in cui non posso risparmiare niente, posso spostare i di una sola posizione

						o	i	i	n	i	o	i	o	n	t	o	n	i	o

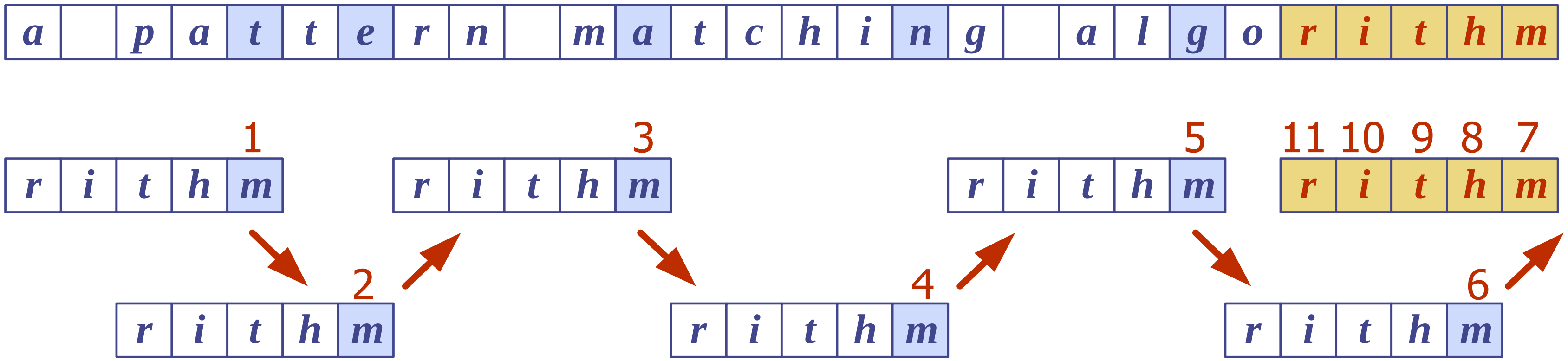
a	n	t	o	n	i	o

Nessun risparmio

a	n	t	o	n	i	o

Pattern matching: Alg. di Boyer-Moore

- Esempio completo: con Boyer-Moore si fanno **11** confronti per trovare l'unico match presente



- Con l'algoritmo "a forza bruta" sono necessari **29** confronti per trovare lo stesso risultato
- L'esempio è particolarmente favorevole, perché ci sono molti "salti lunghi"… ma, in effetti, questo succede spesso quando si cercano parole di senso compiuto all'interno di testi di senso compiuto

Pattern matching: Alg. di Boyer-Moore

- Prestazioni dell'algoritmo
- Ipotizziamo che in un tempo $O(1)$ si possa ottenere una risposta alle domande
  - Dato un carattere c, $c \in P$?
  - Se $c \in P$, quanto vale h = lastIndexOf(c, P) ?
  - Usando la convenzione lastIndexOf(c, P) = – 1 $\Leftrightarrow c \notin P$, basta conoscere il valore di lastIndexOf(c, P) . Immaginiamo di poterlo conoscere in un tempo $O(1)$ per qualsiasi c, ogni volta che ci serve
- Nel caso peggiore, l'algoritmo non trova mai "salti efficaci", quindi le sue prestazioni sono uguali a quelle dell'algoritmo a forza bruta, cioè $O(n\ m)$
  - **Sperimentalmente, però, si osserva che in molti casi pratici le prestazioni sono decisamente migliori**

Pattern matching: Alg. di Boyer-Moore

- Problema rimasto in sospeso: dato un carattere $c \in \Sigma$, come faccio a conoscere lastIndexOf(c, P) in un tempo $O(1)$?
- Si usa una strategia di "pre-compilazione" della funzione applicata al pattern P in esame
  - Si osservi cha la funzione NON dipende dal testo T, ma solo dall'alfabeto Σ caratteristico del dominio in esame e, ovviamente, dal pattern P
  - Nell'ipotesi di poter usare i caratteri come indici in un array, la soluzione è molto semplice: basta avere un array A_P con $|A_P| = |\Sigma|$ e $A_P[c]$ = lastIndexOf(c, P)
  - In questo modo, per conoscere lastIndexOf(c, P) basta un tempo $O(1)$, grazie all'accesso casuale dell'array

Pattern matching: Alg. di Boyer-Moore

- $|A_P| = |\Sigma|$ e $A_P[c]$ = lastIndexOf(c, P)
- Come si calcolano i valori da inserire nell'array A_P?

  for each $c \in \Sigma$

  $\quad A_P[c] = -1$

  for each $i \in [0, |P| - 1]$, per indici crescenti

  $\quad A_P[P[i]] = i$

- L'array viene creato in un tempo $\Theta(m + |\Sigma|)$, quindi l'intero algoritmo di Boyer-Moore è $O(n\ m + |\Sigma|)$, cioè sempre $O(n\ m)$
- Richiede, però, uno spazio di memoria $\Theta(|\Sigma|)$, che non è proporzionale né a n né a m (aspetto negativo)

Pattern matching: Alg. di Boyer-Moore

- L'algoritmo di Boyer-Moore è vantaggioso soprattutto se l'alfabeto è abbastanza vasto
  - Se l'alfabeto contiene poche lettere (e magari il pattern è lungo), è poco probabile che una lettera c **non** appartenga al pattern P, mentre questa è proprio la situazione in cui si hanno i "salti" più lunghi
  - È decisamente **poco** adatto quando, ad esempio, $\Sigma = \{0, 1\}$, cioè si fanno ricerche in dati binari, oppure $\Sigma = \{A, C, G, T\}$, cioè si fanno ricerche in sequenze di DNA
- L'algoritmo di Boyer-Moore è vantaggioso soprattutto nella ricerca di pattern abbastanza lunghi
  - La lunghezza massima di un "salto" è $m = |P|$, quindi se i pattern sono corti si risparmia poco
  - Attenzione, però, perché con l'aumento di m diventa meno probabile che una lettera c del testo non appartenga a P, cioè diminuisce la probabilità di poter fare un salto "lungo"