Dati e Algoritmi I (Pietracaprina)

Esercizi sul Text Processing

Problema 1. Sia T un testo di n caratteri e P un pattern di $m \leq n$ caratteri e si supponga che i caratteri di P siano tutti distinti. Dimostrare che l'algoritmo `findBrute` esegue al più 2 confronti per ogni posizione $T[k]$ di T .

Soluzione. Consideriamo una posizione arbitraria $T[k]$ del testo T e supponiamo che il primo confronto che coinvolge $T[k]$ sia con $P[j]$, per un qualche j , con $0 \leq j < m$. Se $j = 0$ il confronto è chiaramente l'ultimo che coinvolge $T[k]$ in quanto il ciclo **while** interno al ciclo **for** toccherà $T[k]$ solo una volta, e la prossima iterazione del **for** partirà da $T[k+1]$. Se invece $j > 0$ significa che il primo confronto che coinvolge $T[k]$ avviene nell'iterazione i del **for**, con $i = k - j$. In tale iterazione $T[k]$ viene toccato solo una volta. Inoltre, dato che i caratteri di P sono tutti diversi, le successive $j - 1$ iterazioni del **for**, con $i = k - j + 1, k - j + 2, \dots, k - 1$, falliscono al primo confronto e non arrivano a toccare $T[k]$. Quindi $T[k]$ verrà toccato una seconda volta nella iterazione $i = k$, quando viene confrontato con $P[0]$. Per quanto argomentato sopra, questa sarà l'ultima iterazione in cui $T[k]$ è coinvolto in un confronto. $\square$

Problema 2. Sia T un testo di n caratteri.

- Progettare un algoritmo che, usando un unico ciclo, trova la posizione i in cui inizia la più lunga sottostringa di T fatta di caratteri uguali. (Ad esempio per $T = \text{aabaaabccbb}$ l'algoritmo deve restituire la posizione $i = 3$ relativa all'inizio della sottostringa `aaa`).
- Analizzare la complessità in tempo dell'algoritmo. (Per avere punteggio pieno l'algoritmo deve avere complessità $O(n)$.)
- Trovare un invariante per il ciclo che serva per provare la correttezza dell'algoritmo.

Soluzione.

- Un possibile algoritmo è il seguente

Algoritmo MaxSubstringEq(T)

input Testo T , con $|T|=n$

output indice $i \geq 0$ di inizio della sottostringa piu' lunga di T
di caratteri uguali

```

i ← 0; k ← 1;
max-k ← 1;
for j ← 1 to n-1 do {
    if (T[j] = T[j-1]) then {
        k ← k+1;
        if (k > max-k) then {
            i ← j-k+1;
            max-k ← k;
        }
    }
    else k ← 1;
}
return i

```

- b. Dato che fuori dal ciclo `for` e in ogni sua iterazione si esegue un numero costante di operazioni, la complessità è proporzionale al numero di iterazioni del ciclo `for`, ed è quindi $O(n)$.
- c. Un invariante che vale alla fine di ogni iterazione del ciclo, e che alla fine del ciclo garantisce la correttezza dell'algoritmo, è il seguente:
- una sottostringa di $T[0 \dots j]$ fatta di caratteri uguali e di lunghezza massima inizia alla posizione i ed è lunga max-k ;
 - la più lunga sottostringa di T fatta di caratteri uguali che termina in $T[j]$ è $T[j - k + 1 \dots j]$, ed è quindi lunga k .

□

Problema 3. Si consideri un testo T e un pattern P , con $|T| = n$, $|P| = m \leq n$ e si supponga che $P[0] \neq P[i], \forall 1 \leq i \leq m - 1$.

- a. Modificare `findBrute` per risolvere efficientemente le istanze del problema di pattern matching che soddisfano queste ipotesi. L'algoritmo modificato deve contenere un solo ciclo e deve avere complessità $O(n)$.
- b. Analizzare la complessità dell'algoritmo
- c. Identificare un opportuno invariante per il ciclo e usarlo per provare la correttezza dell'algoritmo.

Soluzione.

- a. La modifica a `findBrute` si basa sulle seguenti idee. Anzitutto, si progetta un unico ciclo che utilizza due variabili i e j per scandire testo e pattern rispettivamente. Le variabili avanzano se $T[i] = P[j]$, mentre appena si verifica un mismatch ($T[i] \neq P[j]$), la variabile j viene reimpostata a 0. Per sfruttare le ipotesi fatte sul pattern, quando si verifica il mismatch la variabile i viene lasciata inalterata. Supponendo infatti che $T[i] \neq P[j]$ ma $T[i - j \dots i - 1] = P[0 \dots j - 1]$, sarebbe inutile provare ad allineare il pattern a partire da $T[\ell]$, con $i - j < \ell < i$, in quanto quelle posizioni non possono contenere il carattere $P[0]$ e produrrebbero subito un mismatch. Lo pseudocodice dell'algoritmo è il seguente:

Algoritmo BFMPPlus(T,P)

input Testo T, e pattern P, con $|T|=n$ e $[P]=m$, e $P[0] \neq P[j] \forall 0 < j < m$

output indice i tale $P=T[i \dots i-m+1]$, se esiste, altrimenti -1

```

i ← 0; j ← 0;
while (i < n) do {
    if (P[j] = T[i]) then {
        i ← i+1
        j ← j+1
        if (j=m) then return (i-j)
    }
    else if (j > 0) then j ← 0;
    else i ← i+1
}
return -1

```

- b. Dato che in ogni iterazione del while si esegue un numero costante di operazioni, la complessità risulta proporzionale al numero di iterazioni del while. Per dare un limite superiore a tale numero, si osservi che i aumenta di 1 almeno ogni due iterazioni e non decresce mai. Di conseguenza la complessità di BFMPPlus è $O(n)$.
- c. La correttezza è basata sul seguente invariante che deve valere alla fine di ogni iterazione del while. Siano $P[j]$ e $T[i]$ i prossimi caratteri di pattern e testo da confrontare. Allora si ha che

- $P \neq T[\ell \dots \ell + m - 1], \forall 0 \leq \ell < i - j$
- $P[0 \dots j - 1] = T[i - j \dots i - 1]$

Si noti che è, nella sostanza, lo stesso invariante dell'algoritmo **findBrute**. L'invariante vale all'inizio del ciclo per vacuità. Supponiamo che valga all'inizio di una generica iterazione (ovvero alla fine della precedente) e dimostriamo che rimane vero anche alla fine dell'iterazione. Distinguiamo diversi casi

- **Caso 1** ($P[j] = T[i]$). In questo caso, l'algoritmo incrementa i e j , ed entrambe le proprietà dell'invariante rimangono vere. La prima infatti non cambia, mentre per la seconda la corrispondenza tra testo e pattern si estende di un carattere.
- **Caso 2** ($P[j] \neq T[i]$ e $j > 0$). L'assunzione fatta su P garantisce e l'invariante che valeva all'inizio della iterazione, garantiscono che non si può trovare un'occorrenza del pattern nel testo a partire da un indice $< i$. Ponendo quindi $j = 0$ e non cambiando i , l'invariante continua a valere.
- **Caso 3** ($P[j] \neq T[i]$ e $j = 0$). Chiaramente, incrementando i di 1 l'invariante continua a valere.

Consideriamo adesso la fine del while. Se l'ultima iterazione è quella in cui si è trovato il pattern (quindi si ha che $j = m$), allora l'algoritmo restituisce correttamente l'indice di inizio di tale pattern nel testo. Se invece si esce dal while con $i = n$ e $j < m$, allora la prima proprietà dell'invariante garantisce che non ci sono occorrenze di P in T e l'algoritmo restituisce correttamente -1.

□

Problema 4. Si consideri l'algoritmo `computeFailKMP` visto a lezione per il calcolo della failure function di un pattern P . Dimostrare che l'algoritmo è corretto usando il seguente invariante per il ciclo **while**:

- $f(0), f(1), \dots, f(i-1)$ sono stati calcolati correttamente
- $f(i) \leq j+1$
- $P[0 \dots j-1] = P[i-j \dots i-1]$

Soluzione. Verifichiamo anzitutto che l'invariante è vero alla fine di ciascuna iterazione del **while**. All'inizio del ciclo ($i=1, j=0$) l'invariante vale dato che $f(0)=0$ è stata già calcolata, $f(1) \leq 1$, dalla definizione di failure function, e la terza proprietà è vera per vacuità, dato che $j=0$. Supponiamo che sia vero alla fine di una certa iterazione e consideriamo l'iterazione successiva, ovvero l'iterazione in cui si confrontano $P[i]$ e $P[j]$. Distinguiamo diversi casi:

- **Caso 1** ($P[j] = P[i]$). In questo caso, le relazioni $f(i) \leq j+1$, $P[0 \dots j-1] = P[i-j \dots i-1]$, e $P[j] = P[i]$, implicano immediatamente che $f(i) = j+1$ e che $P[0 \dots j] = P[i-j \dots i]$. Inoltre, se $f(i) = j+1$ deve essere necessariamente $f(i+1) \leq j+2$. Di conseguenza, l'assegnamento $f(i) \leftarrow j+1$ e l'incremento di i e di j rendono vero l'invariante alla fine della iterazione.
- **Caso 2** ($P[j] \neq P[i]$ e $j > 0$). In questo caso possiamo dedurre che $f(i) \leq j$. Il fatto che $P[0 \dots j-1] = P[i-j \dots i-1]$ e la definizione di failure function implicano che il prefisso $P[0 \dots f(j-1)-1]$ coincide con la sottostringa $P[i-f(j-1) \dots i-1]$ e che $P[0 \dots f(j-1)-1]$ è il più lungo prefisso che, esteso di un carattere, ha una qualche speranza di diventare suffisso di $P[1 \dots i]$. Quindi, l'assegnamento $j \leftarrow f(j-1)$ mantiene verificato l'invariante.
- **Caso 3** ($P[j] \neq P[i]$ e $j = 0$). In questo caso possiamo dedurre che $f(i) = 0$ e $f(i+1) \leq 1$. Quindi, l'assegnamento $f(i) \leftarrow 0$ e l'incremento di i mantengono verificato l'invariante.

Consideriamo adesso la fine dell'ultima iterazione del **while** ($i=m$). La prima proprietà dell'invariante assicura che tutti i valori della failure function sono stati correttamente calcolati. L'algoritmo quindi risulta corretto. □

Problema 5. Dati un testo T e un pattern P , con $|T|=n$, $|P|=m \leq n$, si vogliono trovare tutte le occorrenze di P in T .

- a. Modificare l'algoritmo `findKMP` per risolvere il problema
- b. Far vedere che la complessità dell'algoritmo è $\Theta(n+m)$.

Soluzione.

- a. Stabiliamo che l'algoritmo restituisca una lista L contenente tutti gli indici i tali che $P = T[i \dots i + m - 1]$. Per determinare efficientemente L basta modificare `findKMP` in modo che continui ad andare avanti anche dopo aver trovato il pattern. In particolare, se a un certo punto confrontiamo $T[i]$ e $P[j]$, con $j = m - 1$, e troviamo corrispondenza, si incrementa i di 1, si aggiunge l'indice $i - m$ alla lista L e si imposta j a $f(m - 1)$. In questo modo, si testerà il prossimo shift del pattern sotto il testo che può portare a un matching completo. Lo pseudocodice è il seguente:

Algoritmo AllOccurrences(T, P)

input Testo T , e pattern P , con $|T|=n$ e $|P|=m$

output Lista L contenente gli indici i tali che $P=T[i \dots i+m-1]$

```
f ← computeFailKMP
i ← 0; j ← 0; L ← lista vuota
while (i < n) do {
  if (P[j] = T[i]) then {
    i ← i+1
    j ← j+1
    if (j=m) then {
      aggiungi i-j a L
      j ← f(m-1)
    }
  }
  else if (j > 0) then j ← f(j-1)
  else i ← i+1
}
return L
```

- b. Lo stesso argomento usato per analizzare `findKMP` dimostra che l'algoritmo `AllOccurrences` ha complessità $\Theta(n + m)$. Per quanto riguarda la correttezza (non richiesta dall'esercizio), essa si prova tramite il seguente invariante che deve valere alla fine di ogni iterazione del `while`, ovvero quando $P[j]$ e $T[i]$ sono i prossimi caratteri da confrontare:

- L contiene tutti gli indici ℓ tali che $P = T[\ell \dots \ell + m - 1]$, con $0 \leq \ell < i - j$.
- $P[0 \dots j - 1] = T[i - j \dots i - 1]$.

□

Problema 6. (Esercizio C-12.19 del testo [GTG14]) *Dati un testo T e un pattern P , con $|T| = n$, $|P| = m$, si vogliono trovare inizio e lunghezza del più lungo prefisso di P in T in tempo $O(n + m)$.*

- Modificare l'algoritmo `findKMP` per risolvere il problema
- Analizzare la complessità dell'algoritmo.

Soluzione.

- a. Modifichiamo l'algoritmo `findKMP` in modo che durante la sua esecuzione tenga traccia del più lungo prefisso di P osservato come sottostringa di T . A tal fine, utilizziamo due variabili k e ℓ che in ogni momento mantengano, rispettivamente, la lunghezza del più lungo prefisso osservato sino a quel momento e il suo punto di inizio. Ovviamente, se a un certo punto tutto P viene trovato in T , quello sarà il più lungo prefisso cercato e l'algoritmo può terminare. Lo pseudocodice è il seguente:

Algoritmo MaxPrefix(T,P)

input Testo T , e pattern P , con $|T|=n$ e $[P]=m$

output lunghezza (k) e inizio (ℓ) del più lungo prefisso di P sottostringa di T

```
f ← compute FailKMP
i ← 0; j ← 0; k ← 0; ℓ ← -1
while (i < n) do {
  if (P[j] = T[i]) then {
    i ← i+1
    j ← j+1
    if (j > k) then {k ← j; ℓ ← i-j}
    if (j = m) then return (k, ℓ)
  }
  else if (j > 0) then j ← f(j-1)
  else i ← i+1
}
return (k, ℓ)
```

- b. Lo stesso argomento usato per analizzare `findKMP` dimostra che l'algoritmo `MaxPrefix` ha complessità $\Theta(n + m)$. Per quanto riguarda la correttezza (non richiesta dall'esercizio), essa si prova tramite il seguente invariante che deve valere alla fine di ogni iterazione del `while`, ovvero quando $P[j]$ e $T[i]$ sono i prossimi caratteri da confrontare:
- un prefisso di P di lunghezza massima che sia sottostringa di $T[0 \dots i-1]$ a partire da una posizione $\leq i-j$ ha lunghezza k e inizia in $T[\ell]$;
 - $P[0 \dots j-1] = T[i-j \dots i-1]$.

□

Problema 7. Si vuole semplificare l'algoritmo `findKMP`, nel caso in cui i caratteri del pattern P siano ordinati alfabeticamente (supponendo che sull'alfabeto sia definito un qualche ordinamento).

- Determinare la failure function per il pattern $P = \text{aaabbcdd}$.
- Usando l'intuizione ricavata dal punto precedente, individuare le caratteristiche particolari della failure function di un qualsiasi pattern ordinato alfabeticamente.
- Dire come può essere modificato l'algoritmo `findKMP` in modo da utilizzare spazio aggiuntivo costante (oltre allo spazio richiesto da T e P), nel caso in cui P sia ordinato alfabeticamente.

Soluzione.

a. La failure function per $P = \text{aaabbcdd}$ è

$$\begin{aligned} f(j) &= j \quad \text{per } 0 \leq j \leq 2 \\ f(j) &= 0 \quad \text{per } 2 < j < 8 \end{aligned}$$

b. In generale si può facilmente verificare che se P è ordinato alfabeticamente e se k è il massimo indice tale che $P[k] = P[0]$, allora la failure function è:

$$\begin{aligned} f(j) &= j \quad \text{per } 0 \leq j \leq k \\ f(j) &= 0 \quad \text{per } k < j < m \end{aligned}$$

c. Nell'algoritmo `findKMP` il calcolo della failure function per il pattern P diventa semplicemente la determinazione del massimo indice k tale che $P[k] = P[0]$, che richiede una scansione di P e solo spazio costante per memorizzare k . Infine si sostituisce all'istruzione $j \leftarrow f(j-1)$, l'istruzione

if ($j-1 \leq k$) **then** $j \leftarrow j-1$ **else** $j \leftarrow 0$

□

Problema 8. Dati un testo T e un pattern P , con $|T| = n$, $|P| = m \leq n$, si vuole determinare se P è sottostringa circolare di T , cioè se:

- P è sottostringa (“normale”) di T ; oppure
- P si può ottenere concatenando un suffisso di T a un prefisso di T .

(Ad esempio, DBA è sottostringa circolare di ABCDDDB .) Progettare un algoritmo che risolva il problema in tempo $O(n)$.

Soluzione. È sufficiente creare un testo T' di lunghezza $n + m - 1$ concatenando T con il suo prefisso di lunghezza $m - 1$. È immediato vedere che un pattern P di lunghezza m è sottostringa circolare di T se e solo se P è sottostringa di T' . Si applica quindi l'algoritmo `findKMP` a T' e P . La creazione di T' contribuisce una complessità $O(n)$, mentre l'esecuzione di `findKMP` con T' e P contribuisce una complessità $O(n + m)$. Essendo $m \leq n$ si ha che la complessità totale è $O(n)$. □