Algoritmi e Strutture Dati - Prova d'esame
02/05/11

Esercizio 1

Questo esercizio è molto semplice: è sufficiente fare una visita in post-ordine dell'albero, calcolando per ogni nodo v il valore dell'espressione radicata nel nodo v. La complessità è $O(n)$.

```
real expression(TREE t)
if t.data is an operator then
    real v1 ← expression(t.left())
    real v2 ← expression(t.right())
    if t.data = · then
        return v1 · v2
    else if t.data = + then
        return v1 + v2
    else if t.data = − then
        return v1 − v2
else
    return t.data
```

Esercizio 2

E' facile notare che $T(n) = \Omega(n)$, per via della sua componente non ricorsiva. Per provarlo, dobbiamo dimostrare che esiste $c > 0$, $m \geq 0$ tale per cui $T(n) \geq cn$ per ogni $n \geq m$. Si noti quindi che

$$T(n) = 3T(n/4) + T(n/5) + n \geq n \geq cn$$

è vera per ogni $c \leq 1$; questa stessa condizione si applica anche al caso $n = 1$.
Proviamo quindi (per tentativi) a dimostrare che esiste $c > 0$, $m \geq 0$ tale per cui $T(n) \leq cn$ per ogni $n \geq m$. Procediamo per induzione.

- Caso base: $T(1) = 1 \leq c \cdot 1$, che è vera per $c \geq 1$.
- Passo induttivo: supponiamo che $T(n') \leq cn'$ per ogni $n' < n$, e proviamo che la diseguaglianza è vera anche per n: $T(n) \leq cn$.

$$\begin{aligned} T(n) &= 3T(n/4) + T(n/5) + n && \text{Sostituzione} \\ &\leq 3cn/4 + cn/5 + n \\ &= \frac{19}{20}cn + n \\ &\leq cn && \text{Obiettivo} \end{aligned}$$

che è vera per $19/20c + 1 \leq c \Leftrightarrow c \geq 20$.

Abbiamo così ottenuto che $T(n) = \Theta(n)$.

Esercizio 3

E' sufficiente modificare l'algoritmo di BFS (erdos()) per calcolare le distanza da un nodo sorgente s; tutte le volte che un nodo v viene scoperto per la prima volta a partire da un nodo u, il numero di cammini per raggiungere v è pari al numero di cammini per raggiungere

u; se v viene scoperto altre volte a partire da un nodo w, si aggiungono i cammini minimi per raggiungere w.

```
countmin(GRAPH G, NODE s, integer[] dist, integer[] count)
    QUEUE S ← Queue()
    S.enqueue(s)
    foreach u ∈ G.V() − {s} do dist[u] ← ∞
    dist[s] ← 0
    count[s] ← 1
    while not S.isEmpty() do
        NODE u ← S.dequeue()
        foreach v ∈ G.adj(u) do                          % Esamina l'arco (u, v)
            if dist[v] = ∞ then                          % Il nodo u non è già stato scoperto
                dist[v] ← dist[u] + 1
                count[v] ← count[u]
                S.enqueue(v)
            else if dist[v] = dist[u] + 1 then           % Il nodo u è gia stato scoperto da un cammino di lunghezza uguale
                count[v] ← count[v] + count[u]
```

Sia v un nodo la cui distanza minima dalla sorgente sia $d = dist[v]$; supponiamo che esistano k nodi $u_1, \ldots, u_k$ tale per cui gli archi (u_i, v) siano l'ultimo arco di un cammino da s a v di lunghezza d; allora

$$count[v] = \sum_{i=1}^{k} count[u_i]$$

Al termine, il vettore $count$ contiene il numero di cammini minimi diversi che differiscono di almeno un arco. L'algoritmo ha complessità $O(m + n)$.

Esercizio 4

Un algoritmo banale, di costo $O(n^3)$, verifica per ogni coppia di indici i, j, con $i \leq j$, se la sottostringa $s[i \ldots j]$ è palindroma oppure no, memorizzando la stringa più lunga.

```
longestPalindrome(ITEM[] s, integer n)
    integer max ← 0
    integer m_i, m_j
    for i ← 1 to n do
        for j ← i to n do
            if j − i + 1 > max and isPalindrome(s, i, j) then
                max ← j − i + 1
                m_i ← i; m_j ← j
    print s[m_i ... m_j]
```

```
isPalindrome(ITEM[] s, integer i, integer j)
    if i ≥ j then
        return true                                      % Stringhe vuote o con un carattere sono palindrome
    else
        return (s[i] = s[j]) and isPalindrome(s, i + 1, j − 1)
```

Questo algoritmo ha un problema: risolve inutilmente sottoproblemi ripetuti. E' quindi possibile utilizzare un meccanismo di programmazione dinamica che memorizza i problemi già risolti, riducendo il costo a $O(n^2)$.

La sottostruttura ottima può essere definita dalla seguente relazione ricorsiva, che dati due indici i, j, $1 \leq i < j \leq n$, ritorna la lunghezza della sottostringa palindroma massimale contenuta in $s[i \dots j]$.

$$L[i,j] = \begin{cases} 0 & i = j+1 \\ 1 & i = j \\ j-i+1 & i < j \wedge s[i] = s[j] \wedge L[i+1, j-1] = j-i-3 \\ max\{L[i, j-1], L[i+1, j]\} & \text{altrimenti} \end{cases}$$

Affiancata a questa struttura è poi utile avere un ulteriore matrice S che contenga informazioni per identificare una stringa massimale $S[i,j]$ di lunghezza $L[i,j]$. Una bozza di codice potrebbe essere la seguente:

```
longestPalindrome(ITEM[] s, integer n, integer i, integer j, SUBSTRING[][]S)
if L[i,j] = nil then
    if i > j then return 0
    if i = j then
        L[i,j] ← 1
        S[i,j] ← s[i...j]
    else if s[i] = s[j] and longestPalindrome(s,n,i,j,L,S) = j-i-3 then
        L[i,j] ← j-i+1
        S[i,j] ← s[i...j]
    else
        L[i,j] ← max(L[i,j-1], L[i+1,j])
        S[i,j] ← iif(L[i,j] = longestPalindrome(s,n,i+1,j,L,S), longestPalindrome(s,n,i,j-1,L,S))
return L[i,j]
```

Assumiamo che la matrice L sia inizializzata a **nil** (costo $\Theta(n^2)$) e che la chiamata iniziale sia longestPalindrome$(s, n, 1, n, L, S)$; la massima stringa palindroma si troverà alla fine in $S[1, n]$. Il costo computazionale è comunque $\Theta(n^2)$ (al di là dell'inizilizzazione) perché tutte le caselle verranno riempite.

Invece di memorizzare la lunghezza e la sottostringa separatemente, è possibile memorizzare un unico oggetto di tipo substring, che contiene due campi i e j (rappresentanti l'indice di inizio e di fine di una sottostringa) e ha un metodo length() che ritorna la lunghezza $j - i + 1$ della sottostringa stessa. Attenzione ad utilizzare `String` di Java, la cui creazione (rappresentata da $s[i \dots j]$ nel programma) ha costo $O(n)$.

Un approccio differente, sempre basato sulla programmazione dinamica, considera invece una matrice booleana che contiene vero se la sottostringa rappresentata da una coppia di indici è palindroma; ovviamente lo è se i due caratteri agli estremi sono uguali e la stringa ottenuta eliminando questi è ancora palindroma. La relazione ricorsiva che definisce la sottostruttura ottima è la seguente:

$$B[i,j] = \begin{cases} (s[i] = s[j]) \wedge B[i-1, j-1] & i > j \\ \textbf{true} & i \geq j \end{cases}$$

Si utilizza quindi una tabella booleana $B[1 \dots n, 1 \dots n]$. I casi base sono dati dalle caselle (i,i) e $(i, i-1)$, $1 \leq i \leq n$, che hanno tutte valore **true**.
Data una matrice, identifichiamo con D_k la k-esima diagonale, definita in questo modo: la prima diagonale, $k = 1$, o diagonale principale, è quella data dalle caselle (i,i), con $1 \leq i \leq k$; la diagonale k è data dalle caselle $(i, i+k-1)$, con $1 \leq i \leq n-k+1$. Si noti che le caselle dell'i-esima diagonale rappresentano sottostringhe di lunghezza i.
Nonostante non fosse richiesto dal problema, gli algoritmi che seguono stampano tutte le stringhe di lunghezza massimale (e non solo una). Le coppie di indici delle stringhe massimali vengono salvate in un insieme S.
Utilizzando memoization, il costo dipende dall'inizializzazione della matrice, che deve contenere solo valori **nil**, e questo comporta un costo di $\Theta(n^2)$. Inoltre, poichè non esiste un solo problema generale $B[i,j]$ da cui hanno origine tutti i sottoproblemi, è necessario chiamare il meccanismo di memoization per diagonali, a partire dalla n-esima diagonale (costituita da un solo elemento) fino alla prima

(la principale) e fermarsi non appena si trova una diagonale con almeno un valore **true**.

```
longestPalindrome(ITEM[] s, integer n)
integer[][] B ← new integer[1...n][1...n]
foreach i, j : 1 ≤ i < j ≤ n do B[i, j] ← nil
integer S ← Set();                          % Insieme degli indici delle stringhe palindrome massimali
integer ℓ ← n                               % Lunghezza stringa
while ℓ ≥ 1 and S = ∅ do
    for i ← 1 to n − l + 1 do               % Indice iniziale
        j ← i + ℓ − 1;                      % Indice finale
        B[i, j] ← isPalindrome(s, i, j)
        if B[i, j] = true then
            S.insert(⟨i, j⟩)
    ℓ ← ℓ − 1
foreach ⟨i, j⟩ ∈ S do
    print s[i...j]
```

```
isPalindrome(ITEM[] s, integer[][] B, integer i, integer j)
if i ≥ j then
    return true
if B[i, j] = nil then
    B[i, j] ← (s[i] = s[j]) and B[i + 1, j − 1]
return B[i, j]
```

Utilizzando programmazione dinamica (e non memoization), la tabella viene riempita per diagonali, partendo dalla principale e andando verso la diagonale n. L'insieme viene resettato ogni qualvolta una stringa di lunghezza maggiore delle precedenti viene trovata; la lunghezza delle stringhe massimali trovate finora viene conservata nella variabile *max*. È possibile ottimizzare l'algoritmo notando che se non esistono caselle **true** nelle ultime due diagonali (condizione $max \geq \ell - 2$), allora è inutile andare avanti, perchè tutti i valori successivi saranno **false**.

```
longestPalindrome(ITEM[] s, integer n)
boolean[][] B ← new boolean[1...n][1...n]
integer S ← Set();                          % Insieme degli indici delle stringhe palindrome massimali
integer max ← 1;                            % Lunghezza palindroma massimale
for i ← 1 to n do                           % Inizializza casi base
    B[i, i] ← true
    B[i, i − 1] ← true
    S.insert(⟨i, i⟩)
integer ℓ ← 2                               % Lunghezza stringa
while ℓ ≤ n and max ≥ ℓ − 2 do
    for i ← 1 to n − l + 1 do               % Indice iniziale
        j ← i + ℓ − 1;                      % Indice finale
        if B[i, j] = (s[i] = s[j]) and B[i + 1, j − 1] then
            B[i, j] ← true
            if ℓ > max then
                max ← ℓ
                S ← Set()
            S.insert(⟨i, j⟩)
        ℓ ← ℓ + 1
foreach ⟨i, j⟩ ∈ S do
    print s[i...j]
```

Poichè ogni casella della tabella D viene riempita al più una volta ed ogni riempimento costa $O(1)$ (nel caso l'insieme sia rappresentato da una lista non ordinata, la cui operazione di inserimento ha costo costante), la complessità totale è $O(n^2)$. Il caso pessimo è rappresentato da una stringa interamente palindroma.
Il meccanismo di ottimizzazione ha tuttavia un effetto sulla complessità; detto k la lunghezza della stringa palindroma massimale, la complessità di questo algoritmo è limitata superiormente da $O(kn)$. Provando l'algoritmo sul primo capitolo dei Promessi sposi (senza spazi e punteggiatura), si vede che $k = 9$.
Questa soluzione richiede ancora $\Theta(n^2)$ spazio, che però non richiede inizializzazione. Si può migliorare ancora, notando che è necessario mantenere solamente le ultime due diagonali per calcolare la successiva, il che significa che è possibile ridurre lo spazio necessario ad $\Theta(n)$. La gestione degli indici è tuttavia più complicata e c'è una soluzione migliore, che non utilizza memoria aggiuntiva.
L'idea è che possibile partire da ogni singolo carattere e da ogni coppia di caratteri consecutivi uguali e utilizzarli come *semi* di una stringa palindroma. Si guarda quindi la coppia di caratteri immediatamente successiva e precedente al seme, e se sono uguali si allarga la nostra stringa palindroma; si prosegue così fino a quando non si trovano due caratteri estremi diversi o si esce dalla stringa.

```
longestPalindrome(ITEM[] s, integer n)
SET S
integer max ← 0
{ k = 0 stringhe lunghezza dispari, k = 1 stringhe lunghezza pari }
for k ← 0 to 1 do
    for i ← 1 to n − k do
        if S[i] = S[i + k] then
            ℓ ← 0
            while i − (ℓ + 1) ≥ 1 and i + k + (ℓ + 1) ≤ n ∧ s[i − (ℓ + 1)] = s[i + k + (ℓ + 1)] do
                ℓ ← ℓ + 1
            if 2ℓ + 1 + k > max then
                max ← 2ℓ + 1 + k
                S ← ∅
            if 2ℓ + 1 + k = max then
                S.insert(⟨i − ℓ, i + k + ℓ⟩)
foreach ⟨i, j⟩ ∈ S do
    print s[i...j]
```

Il costo di questo algoritmo è $O(n^2)$, che si ottiene per esempio con un insieme di caratteri tutti uguali; è anche $O(kn)$, perchè comunque per ogni carattere e per ogni coppia di caratteri consecutivi uguali non fa più di $O(k)$ operazioni.
C'è un ultimo problema, che è il seguente: questo algoritmo si comporta male nel caso di stringhe che contengono molte stringhe palindrome di lunghezza $O(n)$, come per esempio stringhe formate da soli caratteri uguali e stringhe formate da due caratteri alternanti. E' possibile risolvere questo problema cambiando l'ordine in cui vengono utilizzati i semi; di questa versione non presento lo pseudo-codice, ma direttamente codice Java. Il codice è poco commentato e di difficile spiegazione; viene discusso in classe. Come principio generale, lavora sulle *antidiagonali*, ovvero le diagonali che corrono da sinistra/basso a destra/alto.
Come il precedente, la complessità è $O(kn)$, ma per casi particolari può essere addirittura inferiore; ad esempio, stringhe totalmente palindrome, di soli caratteri uguali e casi simili vengono identificati in tempo $O(n)$; il caso pessimo avviene per stringhe in cui k è circa $n/2$, per i quali $O(kn)$ corrisponde a $O(n^2)$.

```
import java.io.*;
import java.util.*;

public class Palindrome
{

public static void main(String[] args) throws Exception
{
  BufferedReader reader = new BufferedReader(new FileReader(args[0]));
  String m = reader.readLine();
  char[] s = new char[m.length() + 1];
  m.getChars(0, m.length() - 1, s, 1);
  List<String> pal = longestPalindrome(s, m.length());
  for (String e : pal) {
    System.out.println(e);
  }
}

static public List<String> longestPalindrome(char[] s, int n)
{
  int max = 0; // Max palindrome found so far
  int i, j; // Substrings
  int ci, cj; // Pairs of chars checked for equality
  List<String> S = new ArrayList<String>();

  for (int len = n; len >= 1; len--) {

    // Main and upper anti-diagonal
    i = 1;
    j = len;

    ci = (i + j) / 2;
    cj = (i + j + 1) / 2;
    while (ci >= 1 && cj <= n && s[ci] == s[cj]) {
      ci--; cj++;
    }
    ci++; cj--;
    if (cj - ci + 1 > max) {
      max = cj - ci + 1;
      S.clear();
    }
    if (cj - ci + 1 == max) {
      S.add(String.copyValueOf(s, ci, max));
    }

    // Lower anti-diagonal
    i = n - len + 1;
    j = n;

    // Skip computing (1,n), already computed in the previous block.
    if (i == 1)
      continue;

    ci = (i + j) / 2;
    cj = (i + j + 1) / 2;
    while (ci >= 1 && cj <= n && s[ci] == s[cj]) {
      ci--; cj++;
    }
    ci++; cj--;
    if (cj - ci + 1 > max) {
      max = cj - ci + 1;
      S.clear();
    }
    if (cj - ci + 1 == max) {
      S.add(String.copyValueOf(s, ci, max));
    }
    if (max >= len) {
      return S;
    }
  }
  assert true; // Never reached
  return S;
}
}
```