

Algoritmi e Strutture Dati

24/04/14

Esercizio 1

La funzione `random()` può lasciare perplessi, specialmente quando viene usata per definire il numero di iterazioni del ciclo `for`. Ma visto che stiamo cercando un limite superiore, possiamo notare che comunque $k \leq n$, e quindi $2^k \leq 2^n$. Quindi la complessità di questa procedura è approssimabile a:

$$\begin{aligned} T(n) &\leq 2T(n/2) + 2^n & n > 1 \\ T(n) &= 1 & n = 1 \end{aligned}$$

Utilizzando il Master Theorem, e in particolare il terzo caso, è possibile vedere che $T(n) = O(2^n)$. Ma trattandosi del terzo caso, bisogna anche dimostrare che esiste una costante $c < 1$ tale per cui $af(n/b) \leq cf(n)$ per ogni n sufficientemente grande. Ovvero, dobbiamo dimostrare che:

$$\exists c < 1, \exists m \geq 0 : 2 \cdot 2^{n/2} \leq c \cdot 2^n, \forall n \geq m$$

Riscriviamo la disequazione in questo modo:

$$2^{n/2+1} \leq c \cdot 2^n$$

e scegliamo $c = 1/2$; otteniamo quindi

$$2^{n/2+1} \leq 2^{n-1}$$

L'ultima disequazione è vera per $n \geq 4$.

Esercizio 2

È possibile risolvere il problema con una semplice post-visita dell'albero, di costo $O(n)$.

```
(int, boolean) checkBalanceRic(TREE t)
```

```
if t == nil then
```

```
  return (0, true)
```

```
  nL, balancedL = checkBalanceRic(t.left)
```

```
  nR, balancedR = checkBalanceRic(t.right)
```

```
  return (nL + nR + 1, balancedL and balancedR and nL ≤ 2nR + 1 and nR ≤ 2nL + 1)
```

```
boolean checkBalance(TREE t)
```

```
  n, balanced = checkBalanceRic(t)
```

```
  return balanced
```

La soluzione proposta usa il trucco di ritornare una coppia di valori, dove il primo è un intero che rappresenta il numero di nodi compresi nel sottoalbero, mentre il secondo è un booleano uguale a `true` se il sottoalbero è abbastanza bilanciato. Notate che meccanismi per scrivere funzioni che restituiscono coppie di valori (e in generale n-tuple) sono presenti fra l'altro in Python, Go e Scala.

Un vostro collega ha utilizzato un metodo alternativo, basato su un singolo intero, negativo se il sottoalbero non è abbastanza bilanciato, positivo o nullo (e corrispondente al numero di nodi) se lo è:

```

int checkBalanceRic(TREE t)
    if t == nil then
        | return 0
    nL = checkBalanceRic(t.left)
    nR = checkBalanceRic(t.right)
    if nL < 0 or nR < 0 or nL > 2nR + 1 or nR > 2nL + 1 then
        | return -1                                     % Non abbastanza bilanciato
    return nL + nR + 1

```

```

boolean checkBalance(TREE t)
    return checkBalanceRic(t) ≥ 0;

```

Un ulteriore metodo, altrettanto valido ma che richiede la modifica dei nodi e l'utilizzo di una quantità di memoria pari ad $O(n)$, consiste nel memorizzare il numero di nodi in campi aggiuntivi aggiunti nei nodi dell'albero. In questo modo si fanno due visite, una per scrivere i nodi nell'albero, una per controllare se è abbastanza bilanciato.

```

int count(TREE t)
    if t == nil then
        | return 0
    t.count = count(t.left) + count(t.right) + 1
    return t.count

```

```

boolean checkBalanceRic(TREE t)
    int nL, nR = 0, 0
    int balancedL, balancedR = true, true
    if t.left ≠ nil then
        | nL = t.left.count
        | balancedL = checkBalanceRic(t.left)
    if t.right ≠ nil then
        | nR = t.right.count
        | balancedR = checkBalanceRic(t.right)
    return balancedL and balancedR and nL ≤ 2nR + 1 and nR ≤ 2nL + 1

```

```

boolean checkBalance(TREE t)
    count(t)
    return checkBalanceRic(t);

```

Tutti questi metodi hanno complessità $\Theta(n)$, in quanto si basano su una o due visite in profondità. Personalmente, sceglierei il metodo basato su un singolo intero.

Molti invece hanno seguito un approccio in cui la funzione `count()` veniva richiamata a partire da ogni nodo, senza memorizzazione dei risultati intermedi:

```
int count(TREE t)
```

```
  if t == nil then
```

```
    | return 0
```

```
  return count(t.left) + count(t.right) + 1
```

```
boolean checkBalance(TREE t)
```

```
  if t == nil then
```

```
    | return true
```

```
  int nL = count(t.left)
```

```
  int nR = count(t.right)
```

```
  boolean balancedL = checkBalance(t.left)
```

```
  boolean balancedR = checkBalance(t.right)
```

```
  return balancedL and balancedR and nL ≤ 2nR + 1 and nR ≤ 2nL + 1
```

Questo ha l'effetto di aumentare la complessità dell'algoritmo, perchè la chiamata *count()* su un nodo viene ripetuta un numero di volte pari al livello del nodo nell'albero, più uno. Ma come calcolare precisamente la complessità?

Quando *checkBalanceRic(t)* viene chiamata su un nodo *t* radice di un sottoalbero di *n* nodi, il sottoalbero viene diviso in due parti contenenti *n_L* e *n_R* nodi, con $n = n_L + n_R + 1$. Ad ogni chiamata di *checkBalance()*, vengono chiamate due visite in profondità sui sottoalberi di *n_L* e *n_R* nodi, con complessità $O(n_L)$ e $O(n_R)$. Le due visite costano quindi $\Theta(n_L + n_R)$ ovvero $\Theta(n)$.

Consideriamo ora i casi ottimo e pessimo. Nel caso ottimo, l'albero in input è ben bilanciato, ovvero e' l'albero con altezza minima. In altre parole $n_L = \lfloor (n-1)/2 \rfloor$ e $n_R = \lceil (n-1)/2 \rceil$. L'equazione di ricorrenza è quindi:

$$T(n) = \begin{cases} 1 & n = 1 \\ T(\lfloor (n-1)/2 \rfloor) + T(\lceil (n-1)/2 \rceil) + \Theta(n) & n > 1 \end{cases}$$

Abbiamo già affrontato equazioni simili a lezione, che possono essere approssimate da:

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n/2) + \Theta(n) & n > 1 \end{cases}$$

che ha limite inferiore $\Omega(n \log n)$.

Nel caso pessimo, l'albero è una linea di nodi di altezza massima. In questo caso, l'equazione di ricorrenza diventa:

$$T(n) = \begin{cases} 1 & n = 1 \\ T(n-1) + \Theta(n) & n > 1 \end{cases}$$

che ha limite superiore $O(n^2)$. In altre parole, la complessità di questo algoritmo è $\Omega(n \log n)$ e $O(n^2)$, comunque ben superiore alle versioni $\Theta(n)$ che abbiamo visto in precedenza.

Esercizio 3

Una soluzione semplice, ma poco efficiente, consiste nell'utilizzare tre cicli annidati che scorrono da 1 a *n*, testando la condizione della terna pitagorica per ognuna delle terne. Questa versione ha costo $\Theta(n^3)$, in quanto esegue esattamente n^3 confronti. Si noti che non è necessario controllare se gli indici sono in qualche modo uguali, perchè in nessun caso è possibile formare una terna pitagorica con due o tre numeri uguali: ad esempio, se $x = y$, allora $x^2 + y^2 = 2x^2$ e non esiste nessun intero *z* tale che $z^2 = 2x^2$, in quanto si ottiene $z = \sqrt{2} \cdot x$, ma $\sqrt{2} \cdot x$ è irrazionale mentre *z* è intero.

```
boolean pythagorean(int[] A, int n)
```

```
for i = 1 to n do
  for j = 1 to n do
    int sum = A[i]2 + A[j]2
    for k = 1 to n do
      if sum == A[k]2 then
        return true
  return false
```

Una seconda possibilità sfrutta la proprietà commutativa della somma, e considera tutte le coppie distinte di cateti, che vengono sommate in ordine qualsiasi, per confrontarla con tutte le possibili ipotenuse. Questa versione ha sempre costo $\Theta(n^3)$ ed esegue esattamente $\frac{n^2(n-1)}{2}$ confronti:

```
boolean pythagorean(int[] A, int n)
```

```
for i = 1 to n - 1 do
  for j = i + 1 to n do
    int sum = A[i]2 + A[j]2
    for k = 1 to n do
      if sum == A[k]2 then
        return true
  return false
```

Una terza possibilità consiste nell'ordinare i numeri all'inizio, con costo $O(n \log n)$, e poi considerare tutte le possibili terne con indici distinti. Essendo il vettore ordinato si ottiene che l'ipotenusa, specificata dal terzo indice e per definizione maggiore dei cateti, debba avere un indice maggiore di quello dei cateti. Ancora una volta, abbiamo trovato una versione con complessità $\Theta(n^3)$ ma con costanti moltiplicative migliori (salvo errori di calcolo):

$$T(n) = \frac{n(n-2)(n-7)}{6} + O(n \log n)$$

```
boolean pythagorean(int[] A, int n)
```

```
sort(A, n)
for i = 1 to n - 2 do
  for j = i + 1 to n - 1 do
    int sum = A[i]2 + A[j]2
    for k = j + 1 to n do
      if sum == A[k]2 then
        return true
  return false
```

Una quarta possibilità si ottiene notando che quando l'ipotenusa testata corrente è più grande della somma cercata, allora è inutile andare avanti. Di nuovo, questo algoritmo ha complessità $\Theta(n^3)$, mentre il calcolo esatto del numero di confronti non è possibile perchè dipende dai valori che vengono ordinati. Sicuramente è migliore o uguale a quello precedente.

```
boolean pythagorean(int[] A, int n)
```

```
sort(A, n)
for i = 1 to n - 2 do
    for j = i + 1 to n - 1 do
        int sum = A[i]2 + A[j]2
        int k = j + 1
        while k ≤ n and A[k]2 ≤ sum do
            k = k + 1
        if k ≤ n and sum = A[k]2 then
            return true
return false
```

Tanta fatica per nulla, perchè stiamo solo migliorando le costanti moltiplicative, cosa buona e giusta da fare, ma solo quando non si riesce a migliorare la classe di complessità computazionale dell'algoritmo. E invece, c'è la possibilità di migliorare la classe computazionale. Visto che abbiamo ordinato il vettore, un modo alternativo consiste nel ciclare su tutte le coppie di valori x, y , cercando il valore $z = \sqrt{x^2 + y^2}$ tramite la ricerca binaria. Questo algoritmo ha complessità $\Theta(n^2 \log n)$.

```
boolean pythagorean(int[] A, int n)
```

```
sort(A, n)
for i = 1 to n do
    for j = i + 1 to n do
        real z =  $\sqrt{A[i]^2 + A[j]^2}$ 
        if z = [z] and binarySearch(A, z, 1, n) > 0 then
            return true
return false
```

Un modo ancora più efficiente è il seguente. Ordiniamo il vettore, come prima, con costo $O(n \log n)$. Una terna pitagorica è data da tre valori $A[i], A[j], A[k]$ tali per cui $A[i]^2 + A[j]^2 = A[k]^2$. Senza perdere di generalità, assumiamo che $i < j < k$. Facciamo quindi variare l'indice k dell'ipotenusa fra 3 e n . I cateti sono compresi fra 1 e $k - 1$. Poniamo $i = 1$ e $j = k - 1$; possono darsi tre casi:

- Se $A[i]^2 + A[j]^2 = A[k]^2$, abbiamo trovato una terna e possiamo restituire **true**
- Se $A[i]^2 + A[j]^2 < A[k]^2$, la somma dei quadrati costruiti sui cateti non è sufficiente a raggiungere il quadrato costruito sull'ipotenusa, e quindi bisogna incrementare l'indice i in modo da incrementare tale somma;
- Se $A[i]^2 + A[j]^2 > A[k]^2$, la somma dei quadrati costruiti sui cateti è troppo alta rispetto al quadrato costruito sull'ipotenusa, e quindi bisogna decrementare l'indice j in modo da diminuire tale somma.

In altre parole, stiamo utilizzando l'algoritmo in tempo lineare visto a lezione per identificare una coppia di valori in un vettore ordinato la cui somma è pari ad un valore determinato. La complessità dell'algoritmo è pari a $\Theta(n^2)$.

```
boolean pythagorean(int[] A, int n)
```

```
    sort(A, n)
    for k = 3 to n do
        i = 1
        j = k - 1
        while i < j do
            if A[i]2 + A[j]2 == A[k]2 then
                return true
            else if A[i]2 + A[j]2 < A[k]2 then
                i = i + 1
            else
                j = j - 1
    return false
```

Esercizio 4

Se esiste un ciclo contenente $[x, y]$, la rimozione di $[x, y]$ lascia il grafo connesso (infatti, gli altri archi del ciclo permettono comunque il collegamento fra x e y). Se invece la rimozione $[x, y]$ rende il grafo disconnesso, allora $[x, y]$ non appartiene ad un ciclo. Senza modificare il grafo, modifichiamo quindi la visita in modo da evitare di passare per l'arco $[x, y]$ o $[y, x]$. La complessità è dunque $O(m + n)$.

```
boolean cc(Graph G, Node x, Node y)
```

```
    int[] visited = new boolean[1...G.n]
    foreach u ∈ G.V() do
        visited[u] = false
    ccdfs(G, x, visited)
    foreach u ∈ G.V() do
        if not visited[u] then
            return false
    return true
```

```
ccdfs(Graph G, Node u, int[] visited, int x, int y)
```

```
    visited[u] = true
    foreach v ∈ G.adj(u) do
        if [u, v] ≠ [x, y] and [u, v] ≠ [y, x] and not visited[v] then
            ccdfs(G, v, visited)
```

La piccola bottega degli orrori

Ho visto cose nei vostri compiti, che voi umani non potete nemmeno immaginare... inserisco una raccolta di chicche di programmazione che non vorrei mai vedere, e spero, pubblicandole, di esorcizzarle e in questo modo di non vederle più in futuro.

Mi capita di vedere cose di questo genere, dove *cond* è una variabile booleana.

```

if cond then
  | return true
else
  | return false

```

Non sarebbe meglio scrivere:

```
return cond
```

Non sarebbe un grosso problema se fosse tutto qui, ma mi capita di vedere cose anche più complicate.

```

boolean cond = cond1 and cond2
if cond then
  | return true and cond3 and cond4
else
  | return false

```

La condizione **true and cond3 and cond4** è equivalente a **cond3 and cond4** ma a questo punto, tanto varrebbe scrivere

```

boolean cond = cond1 and cond2
return cond and cond3 and cond4

```

o ancora più brevemente:

```
return cond1 and cond2 and cond3 and cond4
```

Può essere anche più complicato di così. Guardate questo.

```

boolean cond = cond1 and cond2
if not cond then
  | return false
cond = cond and cond3
if not cond then
  | return false
cond = cond and cond4
return cond

```

Qui secondo me mancano due concetti. Il primo è che le operazioni booleane sono "corto-circuitate", ovvero che se la prima condizione di un AND è valutata a **false** o la prima condizione di un OR è valutata a **true**, il resto della condizione non viene valutato. Il secondo è che se abbiamo stabilito che **not cond** è **false**, allora *cond* è **true** e quindi *cond* = *cond and cond3* potrebbe essere sostituito semplicemente da *cond* = *cond3*. In ogni caso, vale la soluzione precedente: basta scrivere

```
return cond1 and cond2 and cond3 and cond4
```

Il codice proposto in questi esempi non è scorretto; ma è difficile da capire e lungo da scrivere, quindi perchè scriverlo?

Il codice seguente, invece, è sbagliato. Supponiamo di volere valutare una condizione ricorsivamente su tutti i nodi di un albero, come da Esercizio 2. Ho visto funzioni scritte in questo modo:

```
boolean funzioneCheRitornaUnBooleano(TREE t)
```

```
if qualche condizione then
| return false
funzioneCheRitornaUnBooleano(t.left)
funzioneCheRitornaUnBooleano(t.right)
return true
```

Questo codice non ha senso: le due chiamate ricorsive testano la condizione nei sottoalberi, ma il loro valore di ritorno viene ignorato; in altre parole, questa funzione non fa altro che testare la condizione sulla radice e basta.

```
int count(TREE t)
```

```
if t = nil then
| return 0
else if t.left  $\neq$  nil and t.right  $\neq$  nil then
| return 2 + count(t.left) + count(t.right)
```

A parte il valore 2 che non si giustifica, questa funzione valuta solo il caso in cui ci siano esattamente due figli; foglie e nodi con un figlio non vengono valutati.

Evitate gli effetti collaterali, lavorate nella maniera più funzionale possibile

```
int count(TREE t)
```

```
if t = nil then
| return 0
int l = count(t.left) + 1
int r = count(t.right) + 1
if (l  $\leq$  2r + 1) and (r  $\leq$  2l + 1) then
| return l + r
else
| print "non abbastanza bilanciato"
| exit(0)
```

A parte l'errore nel calcolo della dimensione (ad esempio, un albero con un nodo ritorna 2), il meccanismo scelto per gestire il doppio parametro booleano (abbastanza bilanciato) e intero (dimensione) non è accettabile, perchè questa funzione fa uscire dal programma non appena si trova un albero non bilanciato.

L'ultima funzione la riporto come scritta dallo studente, perchè somiglia al C ma secondo me è un esempio di cattiva programmazione:

```
int function c(Nodo t) {
    (static) int count = 0;
    c_util(t);
    return count;
}

void function c_util(Nodo t) {
    if (t->left != null) {
        count++;
    }
    c_util(t->left);
}
if (t->right != null) {
```



```
        count++;  
c_util(t->right);  
    }  
}
```