Esercizi di Algoritmi e Strutture Dati

Moreno Marzolla
marzolla@cs.unibo.it

Ultimo aggiornamento: 3 novembre 2010

1 Trova la somma/1

Scrivere un algoritmo che dati in input un array $A[1 \ldots n]$ di n interi positivi (ossia tutti strettamente maggiori di zero) e un intero positivo v, restituisce *true* se e solo se esistono due valori in A la cui somma sia esattamente uguale a v. Analizzare il costo computazionale dell'algoritmo proposto.

Soluzione L'algoritmo controlla tutte le coppie $A[i]$, $A[j]$, con $1 \leq i < j \leq n$ per verificare se $A[i] + A[j] = v$ per qualche valore di i e j. Lo pseudocodice è il seguente:

```
algoritmo trova_somma(array A[1..n] di interi, intero v) -> bool
  for i:=1 to n-1 do
    for j:=i+1 to n do
      if (A[i] + A[j] == v) then
        return true;
      endif
    endfor
  endfor
  return false;
```

Analizziamo il costo computazionale dell'algoritmo proposto. Il corpo del ciclo "for" più interno ha costo $O(1)$, e viene eseguito $n - i$ volte, per $i = 1, 2, \ldots n - 1$. Pertanto, il costo complessivo si ottiene come:

$$\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

Da cui il costo complessivo dell'algoritmo nel caso generale è $O(n^2)$ (*non* $\Theta(n^2)$, perché l'algoritmo potrebbe terminare in qualsiasi momento se l'`if` risulta vero). Il caso peggiore si ha quando non esiste alcuna coppia di indici $i < j$ per cui $A[i] + A[j] = v$, e in tal caso il costo dell'algoritmo è $\Theta(n^2)$. Il caso migliore si verifica quando $A[1] + A[2] = v$, e in tal caso il costo dell'algoritmo è $O(1)$ in quanto l'algoritmo termina alla prima iterazione.

2 Trova la somma/2

Scrivere un algoritmo che dati in input un array $A[1 \dots n]$ di n interi positivi (ossia tutti strettamente maggiori di zero), restituisce *true* se e solo se esistono due valori in A la cui somma sia esattamente uguale a 17. L'algoritmo deve avere costo $O(n)$. (*Suggerimento: utilizzare un vettore di valori booleani* $B[1 \dots 16]$ *tale che...*).

Soluzione L'idea è la seguente: tutti gli elementi del vettore B sono inizialmente settato a *false*. Si effettua quindi una scansione lineare dell'array A. Per ogni intero k presente in A tale che $1 \leq k \leq 16$, si pone $B[k] = true$. Al termine si effettua una scansione di B, controllando se esiste un indice i per cui $B[i] = B[17 - i] = true$. In caso affermativo l'algoritmo ritorna *true* (esistono due numeri in A la cui somma è 17, e tali numeri sono i e $17 - i$), altrimenti ritorna *false*. Lo pseudocodice è il seguente:

```
algoritmo esiste_coppia_17( array A[1..n] di interi ) -> bool
  Sia B[1..16] un array di bool

  // blocco (1)
  for i:=1 to 16 do
    B[i] := false;
  endfor

  // blocco (2)
  for i:=1 to n do
    if ( A[i] < 17 ) then
      B[A[i]] := true;
    endif
  endfor

  // blocco (3)
  for i:=1 to 8 do
    if (B[i] == true && B[17-i] == true)
      return true;
    endif
  endfor
  return false;
```

Analizziamo il costo computazionale dell'algoritmo proposto. Il ciclo "for" indicato con *blocco (1)* ha costo $O(1)$: si noti infatti che effettua un numero *costante* di iterazioni, e ogni iterazione ha costo $O(1)$. Il ciclo indicato con *blocco (2)* ha costo $\Theta(n)$. Infine, il ciclo indicato con *blocco (3)* ha costo $O(1)$ percé anche qui viene eseguito al più 8 volte, e ogni iterazione costa $O(1)$.

Notiamo che se dobbiamo cercare una coppia di valori la cui somma sia 16 (oppure un qualsiasi numero pari) la soluzione sopra *non* funziona: questo si

puo' verificare applicando l'algoritmo al vettore $A = [8, 5]$. Il problema è che in questo caso non è sufficiente sapere se un certo valore compare in A, ma anche *quante* volte compare. La soluzione corretta è la seguente:

```
algoritmo esiste_coppia_16( array A[1..n] di interi ) -> bool
  Sia B[1..15] un array di interi

  // blocco (1)
  for i:=1 to 15 do
    B[i] := 0;
  endfor

  // blocco (2)
  for i:=1 to n do
    if ( A[i] < 16 ) then
      B[A[i]] := B[A[i]]+1;
    endif
  endfor

  // blocco (3)
  for i:=1 to 8 do
    if ( (i==8 && B[i]>1) || (B[i]>0 && B[16-i]>0) ) then
      return true;
    endif
  endfor
  return false;
```

che ha sempre costo $O(n)$.

3 Profondità di un albero binario

Dato un albero binario T, scrivere un algoritmo che calcola la profondità di T. Analizzare il costo computazionale dell'algoritmo proposto. Quando si verifica il caso ottimo? Quando si verifica il caso pessimo?

Soluzione Una possibile soluzione è la seguente:

```
algoritmo profondita(albero binario T) -> intero
  if ( T == null ) then
    errore: albero vuoto!
  endif
  int p_left := 0;
  if ( T.left() != null ) then
    p_left := 1+profondita(T.left());
  endif
  int p_right := 0;
```

```
  if ( T.right() != null ) then
    p_right := 1+profondita(T.right());
  endif
  return max(p_left, p_right);
```

ove la funzione `max(a,b)` restituisce il massimo valore tra a e b. Il costo dell'algoritmo è $\Theta(n)$ (infatti sia nel caso ottimo che nel caso pessimo visita tutti i nodi dell'albero una e una sola volta). L'analisi può essere fatta risolvendo l'equazione di ricorrenza per il tempo $T(n)$, che ha la stessa forma di quella degli algoritmi di visita analizzati a lezione.

4 Conta foglie

Dato un albero binario T, scrivere un algoritmo che calcola il numero di foglie di T (le foglie sono i nodi che non hanno figli). Analizzare il costo computazionale dell'algoritmo proposto. Quando si verifica il caso ottimo? Quando si verifica il caso pessimo?

Soluzione Una possibile soluzione è la seguente:

```
algoritmo contafoglie(albero binario T) -> intero
  if ( T == null ) then
    return 0;
  else
    if ( T.left() == null && T.right() == null ) then
      return 1; // il nodo T e' una foglia
    else
      return contafoglie(T.left()) + contafoglie(T.right());
    endif
  endif
```

Il costo dell'algoritmo è $\Theta(n)$ (sia nel caso ottimo che nel caso pessimo visita tutti i nodi dell'albero una e una sola volta). L'analisi è identica a quella degli algoritmi di visita vista a lezione.

5 Conta nodi

Dato un albero binario T e un intero non negativo k, scrivere un algoritmo che calcola il numero di nodi che si trovano esattamente a profondità k (ricordiamo che la radice si trova a profondità zero). Analizzare il costo computazionale dell'algoritmo proposto. Quando si verifica il caso ottimo? Quando si verifica il caso pessimo?

Soluzione Una possibile soluzione è la seguente:

```
algoritmo contanodi(albero binario T, int k) -> intero
  if ( T == null ) then
    return 0;
  else
    if (k == 0) then
      return 1;
    else
      return contanodi(T.left(),k-1) + contanodi(T.right(),k-1);
    endif
  endif
```

Osserviamo che in questo caso il costo computazionale dell'algoritmo dipende dalla profonditá h dell'albero e dal valore di k. Infatti l'algoritmo opera una visita dell'albero *fermandosi al livello k*.

- Se $k \leq h$, significa che ci possono essere dei nodi a livello maggiore di k che non vengono mai esplorati. Il caso migliore si verifica quando ci sono $k+1$ nodi fino al livello k (disposti secondo una catena), e il costo dell'algoritmo è quindi $\Theta(k)$. Il caso peggiore invece si verifica quando l'albero binario è *completo* fino a livello k, il che significa che tutti i nodi fino al livello k (escluso) hanno esattamente due figli. Considerando che un albero binario completo di altezza k ha $2^{k+1}-1$ nodi, il costo nel caso peggiore è $O(2^{k+1}-1) = O(2^k)$.
- se $k > h$, l'algoritmo visita tutti gli n nodi dell'albero e il costo è $\Theta(n)$ (quindi se $k > h$, caso ottimo e caso pessimo coincidono).

6 Estensione di array dinamici

Supponiamo che uno stack venga implementato mediante un array dinamico S, nel modo seguente:

- Inizialmente l'array S ha dimensione 0;
- Se vogliamo inserire un nuovo elemento (tramite una operazione `push()`) in uno stack che già contiene n elementi, e l'array è pieno ($S.length = n$), allora viene allocato un nuovo array di $n+d$ elementi;
- Se vogliamo rimuovere un elemento (tramite una operazione `pop()`) da uno stack che ne contiene n, e otteniamo uno stack con $n-1$ elementi tale che $n-1 = S.length - d$, allora viene allocato un nuovo array avente $n-1$ elementi.

Il costo delle riallocazioni è dato anche qui dal costo di copiare gli elementi dal vecchio al nuovo array. In sostanza, la dimensione anziché raddoppiare o dimezzarsi, cresce o cala di una quantità d costante.

1. Partendo da uno stack vuoto, quale è il costo complessivo di una sequenza di n operazioni `push()`?

2. Partendo da uno stack con n elementi, quale è il costo complessivo di una sequenza di n operazioni `pop()`?

Soluzione L'analisi è la stessa che abbiamo visto a lezione nel caso dell'algoritmo convenzionale di raddoppiamento-dimezzamento. Supponiamo di inserire n elementi nello stack, a partire dallo stack vuoto. La prima espansione del vettore costa d (vengono copiati d elementi dal vecchio al nuovo vettore). La seconda espansione costa $2d$, la terza $3d$ e così via. Il costo complessivo è quindi:

$$d + 2d + 3d + \ldots + i\ timesd$$

Le espansioni terminano in corrispondenza dell'ultimo indice i per cui risulta $id \leq n$, ossia $i \leq n/d$. Il costo complessivo è quindi

$$\sum_{i=1}^{n/d} i \times d = d\frac{\frac{n}{d}\left(\frac{n}{d}+1\right)}{2} = \frac{1}{2}\left(\frac{n^2}{d} + n\right)$$

che è $\Theta(n^2)$.