ESERCIZIO DI ASD DEL 6 OTTOBRE 2008

Selection-Sort

Sia A un vettore contenente n numeri interi. Si consideri il problema di ordinare gli elementi di A in ordine crescente. In particolare si implementi l'algoritmo SelectionSort che effettua l'ordinamento seguendo i seguenti passi:

1.1 Si eseguono n iterazioni;
1.2 Durante l'i-esima iterazione si determina il minimo tra gli elementi che occorrono in $A[i..n]$ e si porta tale minimo in posizione $A[i]$.

a.1 Si scriva lo pseudo-codice di SelectionSort(A).
a.2 Si dimostri la correttezza di SelectionSort(A).
a.3 Si determini la complessità di SelectionSort(A).

Date: 6 Ottobre 2008.

ESERCIZIO DI ASD DEL 6 OTTOBRE 2008 - CON SUGGERIMENTI

SELECTION-SORT

Sia A un vettore contenente n numeri interi. Si consideri il problema di ordinare gli elementi di A in ordine crescente. In particolare si implementi l'algoritmo SELECTIONSORT che effettua l'ordinamento seguendo i seguenti passi:

1.1 Si eseguono n iterazioni;

1.2 Durante l'i-esima iterazione si determina il minimo tra gli elementi che occorrono in $A[i..n]$ e si porta tale minimo in posizione $A[i]$.

a.1 Si scriva lo pseudo-codice di SELECTIONSORT(A).
Suggerimento: ad ogni iterazione si determini la posizione del minimo.

a.2 Si dimostri la correttezza di SELECTIONSORT(A).
Suggerimento: determinare e dimostrare prima l'invariante per il ciclo.

a.3 Si determini la complessità di SELECTIONSORT(A).
Suggerimento: non c'è differenza tra caso migliore e peggiore.

Date: 6 Ottobre 2008.

ESERCIZIO DI ASD DEL 6 OTTOBRE 2008

Selection-Sort

Algorithm 1 SelectionSort(A)

for $i \leftarrow 1$ to $length(A) - 1$ **do**
  $minpos \leftarrow i$
  for $j \leftarrow i + 1$ to $length(A)$ **do**
    if $A[j] < A[minpos]$ **then**
      $minpos \leftarrow j$
    end if
  end for
  Scambia($A, i, minpos$)
end for

Algorithm 2 Scambia(A, p, q)

$x \leftarrow A[p]$
$A[p] \leftarrow A[q]$
$A[q] \leftarrow x$

Correttezza.

Invariante 1. *All'inizio della j-esima iterazione del ciclo for più interno $A[minpos]$ è minore o uguale di ogni elemento in $A[i..j-1]$, ovvero*

$$\forall k \in [i..j-1](A[minpos] \leq A[k])$$

Nota: nella dimostrazione useremo la notazione $A[minpos] \leq A[i..j-1]$ per indicare che $A[minpos]$ è minore o uguale di ogni elemento in $A[i..j-1]$.

Dimostrazione. Per induzione su j.
Base: $j = i + 1$.
All'inizio dell'iterazione in cui j vale $i+1$ si ha che $minpos = i$ e $A[i..j-1] = A[i]$. Vale banalmente che $A[minpos] = A[i] \leq A[i]$.
Passo:
HpInd) All'inizio della $(k-1)$-esima iterazione del ciclo for vale che $A[minpos] \leq A[i..k-2]$.

Date: 6 Ottobre 2008.

TsInd) All'inizio della k-esima iterazione del ciclo for vale che $A[minpos] \leq A[i..k-1]$.

Tra l'inizio della $(k-1)$-esima iterazione e l'inizio della k-esima iterazione viene eseguita la $(k-1)$-esima iterazione del ciclo for. Quindi dobbiamo esaminare cosa avviene durante tale iterazione. La riga di codice 4 confronta $A[k-1]$ con $A[minpos]$. Se $A[k-1]$ è minore di $A[minpos]$, la riga 5 assegna a *minpos* il valore $k-1$. Per ipotesi induttiva $A[minpos]$ era il più piccolo in $A[i..k-2]$. Se $A[k-1]$ è maggiore o uguale di $A[minpos]$, allora la riga 5 non viene eseguita e vale che $A[minpos] \leq A[i..k-1]$. Se $A[k-1]$ è minore di $A[minpos]$, allora vale che $A[k-1] < A[minpos] \leq A[i..k-2]$ ed inoltre alla riga 5 la variabile *minpos* assume come nuovo valore $k-1$. Quindi dopo l'esecuzione della riga 5 vale che $A[minpos] = A[k-1] \leq A[i..k-1]$. □

L'invariante ci dice che all'inizio dell'iterazione in cui j assume valore $length(A)+1$ vale che $A[minpos] \leq A[i..length(A)]$. Inoltre all'inizio di quest'iterazione la condizione $j \leq length(A)$ che si trova a guardia del ciclo for non è più verificata ed il ciclo termina.

Invariante 2. *All'inizio della i-esima iterazione del ciclo for più esterno vale che:*
- $A[1..i-1]$ *è ordinato;*
- $A[1..i-1] \leq A[i..length(A)]$.

Dimostrazione. Per induzione su i.
Base: $i = 1$

$A[1..0]$ è un vettore vuoto, quindi è banalmente ordinato. Vale anche banalmente che $A[1..0] \leq A[1..length(A)]$, sempre perchè il primo non contiene elementi.
Passo:

HpInd) All'inizio dell'iterazione $k-1$ del ciclo for vale che $A[1..k-2]$ è ordinato e $A[1..k-2] \leq A[k-1..length(A)]$.

TsInd) All'inizio dell'iterazione k del ciclo for vale che $A[1..k-1]$ è ordinato e $A[1..k-1] \leq A[k..length(A)]$.

Dobbiamo esaminare cosa avviene durante l'iterazione $k-1$ del ciclo for. Per quanto dimostrato nell'Invariante precedente sappiamo che quando raggiungiamo la riga 8 del codice vale che $A[minpos] \leq A[k-1..length(A)]$. A questo punto viene eseguita la riga 8 che porta il $A[minpos]$ in posizione $k-1$. Quindi dopo l'esecuzione della riga 8 abbiamo che $A[k-1] \leq A[k..length(A)]$. Inoltre per ipotesi induttiva sappiamo che $A[1..k-2] \leq A[k-1]$. Riassumendo abbiamo che per ipotesi induttiva $A[1..k-2]$ era un vettore ordinato e non è stato modificato, inoltre sempre dall'ipotesi induttiva $A[1..k-2] \leq A[k-1]$ ed abbiamo dimostrato che $A[k-1] \leq A[k..length(A)]$. Quindi otteniamo che $A[1..k-1]$ è ordinato e $A[1..k-1] \leq A[k..length(A)]$. □

L'invariante ci dice che all'inizio dell'iterazione in cui i assume valore $length(A)$ vale che $A[1..length(A)-1]$ è ordinato e $A[1..length(A)-1] \leq A[length(A)]$. Inoltre all'inizio di quest'iterazione la guardia del ciclo for non è più soddisfatta ed il ciclo for termina.

Theorem 1. SelectionSort(A) *termina sempre ed al termine il vettore A è ordinato.*

Dimostrazione. SelectionSort(A) è costituito da un ciclo for, quindi termina quando termina il ciclo for. Il ciclo for contiene al suo interno un'altro ciclo for.

Il ciclo for più interno termina sempre in quanto viene eseguito al più $length(A)$ volte. Il ciclo più esterno termina per la stessa ragione. Quindi SelectionSort(A) termina sempre.

Dall'Invariante precedente abbiamo che, al termine dell'esecuzione del ciclo for più esterno, $A[1..length(A)-1]$ è ordinato e $A[1..length(A)-1] \leq A[length(A)]$. Da questo segue immediatamente che al termine dell'esecuzione del for (e quindi di tutta la procedura) il vettore A è ordinato. □

Complessità. Indichiamo con n la lunghezza di A. Analizziamo la complessità di ogni riga di codice:

1: Ha complessità $\Theta(1)$ e viene eseguita $\Theta(n)$ volte (per la precisione n volte);
2: Ha complessità $\Theta(1)$ e viene eseguita $\Theta(n)$ volte;
3: Ha complessità $\Theta(1)$ ed viene eseguita $\Theta(n-i)$ volte (per la precisione $n-i+1$ volte) durante la i-esima iterazione del for esterno;
4: Ha complessità $\Theta(1)$ ed viene eseguita $\Theta(n-i)$ volte (per la precisione $n-i$ volte) durante la i-esima iterazione del for esterno;
5: Ha complessità $\Theta(1)$ ed viene eseguita $O(n-i)$ volte (per la precisione viene eseguita al più $n-i$ volte) durante la i-esima iterazione del for esterno;
6: Analogo alla riga 5;
7: Analogo alla riga 3;
8: Ha complessità $\Theta(1)$ e viene eseguita $\Theta(n)$ volte.

Sommando tutto ciò otteniamo

$$\begin{array}{rcl} T(n) & = & \Theta(n) + \sum_{i=1}^{n} -1\Theta(n-i) \\ & = & \Theta(n) + \sum_{i=1}^{n} -1c * (n-i) \\ & = & \Theta(n) + \sum_{i=1}^{n} -1c * n - \sum_{i=1}^{n} -1c * i \\ & = & \Theta(n) + c * n * (n-1) - c * \frac{n*(n-1)}{2} \\ & = & \Theta(n) + c * \frac{n*(n-1)}{2} \\ & = & \Theta(n^2) \end{array}$$

ESERCIZIO DI ASD DEL 13 OTTOBRE 2008

Ricerca per Bisezione

Sia A un vettore contenente n numeri interi ordinati in ordine crescente. Sia k un numero intero. Si consideri il problema di determinare se il numero k occorre in A ed in caso affermativo restituire una delle posizioni in cui occorre k (in caso negativo si restituisca -1).

- a.1 Si scriva lo pseudo-codice di un algoritmo ricorsivo per risolvere il problema sopra proposto.
- a.2 Si dimostri la correttezza dell'algoritmo descritto.
- a.3 Si determini la complessità dell'algoritmo descritto.
- a.4 Si scriva lo pseudo-codice di un algoritmo non ricorsivo per risolvere lo stesso problema. L'algoritmo deve avere la stessa complessità della versione ricorsiva.

Date: 13 Ottobre 2008.

ESERCIZIO DI ASD DEL 13 OTTOBRE 2008

Ricerca per Bisezione

Sia A un vettore contenente n numeri interi ordinati in ordine crescente. Sia k un numero intero. Si consideri il problema di determinare se il numero k occorre in A ed in caso affermativo restituire una delle posizioni in cui occorre k (in caso negativo si restituisca -1).

a.1 Si scriva lo pseudo-codice di un algoritmo ricorsivo per risolvere il problema sopra proposto.
Suggerimento: Si divida il vettore in due parti e si confronti k con l'elemento centrale. Se k è minore dell'elemento centrale si proceda con la ricerca nella metà di sinistra, altrimenti si proceda con la ricerca nella metà di destra.

a.2 Si dimostri la correttezza dell'algoritmo descritto.
Suggerimento: Si proceda per induzione sulla dimensione del vettore.

a.3 Si determini la complessità dell'algoritmo descritto.
Suggerimento: Si determini l'equazione ricorsiva di complessità e la si risolva con l'albero delle chiamate ricorsive.

a.4 Si scriva lo pseudo-codice di un algoritmo non ricorsivo per risolvere lo stesso problema. L'algoritmo deve avere la stessa complessità della versione ricorsiva.
Suggerimento: Si cerchi di mimare con un ciclo while il comportamento dell'algoritmo ricorsivo.

Date: 13 Ottobre 2008.

ESERCIZIO DI ASD DEL 13 OTTOBRE 2008

Ricerca per Bisezione in un Vettore Ordinato

BisectSearch(A, k, i, j) ricerca l'elemento k nel vettore $A[i..j]$ (A tra la posizione i e la posizione j). L'algoritmo procede ricorsivamente eliminando ogni volta almeno metà elementi. Il caso base è il caso di una porzione priva di elementi, ovvero $A[i..j]$ con $i > j$. In questo caso sicuramente k non occorre in $A[i..j]$ e quindi viene ritornato il valore -1. Altrimenti si determina l'elemento $A[h]$ in posizione centrale tra i e j e si distinguono tre casi:

- se $A[h] > k$, si analizza ricorsivamente solo la metà di sinistra, ovvero $A[i..h-1]$;
- se $A[h] = k$, possiamo restituire come risultato h;
- se $A[h] < k$, si analizza ricorsivamente solo la metà di destra.

Si noti che era possibile diminuire il numero di casi da considerare da tre facendo attenzione agli indici nel caso limite $j = i+1$. Si presti attenzione al fatto che tutti i casi (anche quelli non base) devono restituire un valore.

MainBisectSearch(A, k) ricerca l'elemento k nel vettore A richiamando la procedura BisectSearch$(A, k, 1, length(A))$.

Algorithm 1 BisectSearch(A, k, i, j)

```
if i > j then
   return -1
else
   h ← (j + i)/2
   if A[h] > k then
      return BisectSearch(A, k, i, h − 1)
   else
      if A[h] = k then
         return h
      else
         return BisectSearch(A, k, h + 1, j)
      end if
   end if
end if
```

Algorithm 2 MainBisectSearch(A, k)

```
return BisectSearch(A, k, 1, length(A))
```

Date: 13 Ottobre 2008.

Correttezza.

Lemma 1 (Correttezza di BisectSearch(A, k, i, j))**.** *BisectSearch(A, k, i, j) termina sempre ed al termine:*

- *se viene restituito p, allora $A[p] = k$;*
- *viene restituito -1 se e soltanto se k non compare in $A[i..j]$.*

Dimostrazione. Sia $m = j - i + 1$ la lunghezza della porzione $A[i..j]$. Procediamo per induzione su m.
Base: $m \leq 0$, ovvero $i > j$.

Se $i > j$ la condizione alla riga 1 è soddisfatta e quindi si entra nel primo if che non contiene chiamate ricorsive e cicli, quindi banalmente la procedura termina. Dopo aver eseguito la riga 1 si passa alla riga 2. La riga 2 impone di ritornare come valore -1. Effettivamente se $i > j$ non ci sono elementi nella porzione $A[i..j]$, quindi k non può occorrere in tale porzione.
Passo:

HpInd) Se m è minore di $q > 0$ allora BisectSearch(A, k, i, j) termina restituendo -1 sse k non occorre in $A[i..j]$ e p se $A[p] = k$.

TsInd) Se m è uguale a $q > 0$ allora BisectSearch(A, k, i, j) termina restituendo -1 sse k non occorre in $A[i..j]$ e p se $A[p] = k$.

Visto che $m = q > 0$ la condizione alla riga 1 non è soddisfatta e si passa alla riga 3. Alla riga 4 si calcola la posizione centrale in $A[i..j]$ e si memorizza la posizione nella variabile h. Alla riga 5 si controlla se $A[h] > k$. Se $A[h] > k$, allora, visto che il vettore A è ordinato, l'elemento k non si può trovare nella porzione $A[h..j]$, ma si può trovare nella porzione $A[i..h-1]$. Infatti se $A[h] > k$ si passa alla riga 6 e si ritorna lo stesso valore ritornato da BisectSearch$(A, k, i, h-1)$. Siccome $(h-1) - i + 1 < m$, per ipotesi induttiva BisectSearch$(A, k, i, h-1)$ termina sempre restituendo -1 sse k non occorre in $A[i..j]$ e p se $A[p] = k$. Quindi se $A[h] > k$ abbiamo che vale la tesi. Se non è soddisfatta la condizione alla riga 5, allora $A[h] \leq k$ e si passa direttamente alla riga 7. Alla riga 8 si controlla se $A[h] = k$ ed in questo caso si ritorna correttamente il valore h e si termina. Se neanche la condizione alla riga 8 è soddisfatta, allora $A[h] > k$ e si passa alla riga 10. A questo punto la riga 11 effettua correttamente la chiamata ricorsiva sulla porzione $A[h+1..j]$ e per ipotesi induttiva otteniamo la tesi anche in questo caso. □

Theorem 1. MainBisectSearch(A, k) *termina sempre ed al termine:*

- *se viene restituito p, allora $A[p] = k$;*
- *viene restituito -1 se e soltanto se k non compare in A.*

Dimostrazione. Segue immediatamente dal lemma precedente, visto che MainBisectSearch(A, k) esegue solamente BisectSearch$(A, k, 1, length(A))$. □

Complessità. Indichiamo con $m = j - i + 1$, ovvero la lunghezza di $A[i..j]$. Analizziamo la complessità di BisectSearch(A, k, i, j). Ogni riga di codice, a parte le righe 6 e 11, ha complessità $\Theta(1)$ e viene eseguita al più una volta. Inoltre la riga 1 viene sicuramente eseguita una volta. Quindi, escludendo le righe 6 e 11 abbiamo una complessità pari a $\Theta(1)$. La riga 6 comporta una chiamata ricorsiva su dimensione $m/2$. La riga 11 comporta una chiamata ricorsiva su dimensione $m/2$. La riga 6 e la riga 11 si trovano su due rami di un if, quindi al più una di esse viene eseguita. Nel caso peggiore una delle due chiamate sempre verrà eseguita. Per esempio questo si verifica se k occorre solo in posizione 1. Quindi otteniamo la seguente equazione ricorsiva di complessità:

$$T(m) \quad = \quad \begin{cases} \Theta(1) & m = 1 \\ \Theta(1) + T\left(\frac{m}{2}\right) & m > 1 \end{cases}$$

Risolviamo l'equazione con l'albero delle chiamate ricorsive (si veda Figure). Ricordiamo che $\Theta(1)$ va sostituito con una costante c.

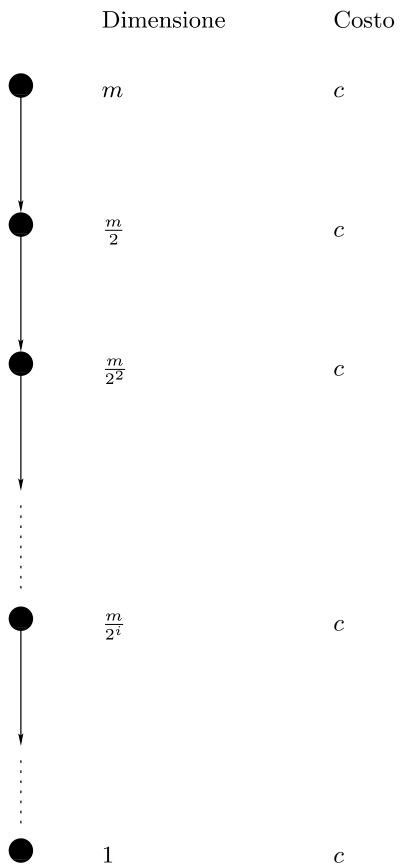

Figura 1. Albero delle chiamate ricorsive

Osserviamo che all'i-esimo livello dell'albero la dimensione su cui viene effettuata la chiamata ricorsiva è $\frac{m}{2^i}$. Quindi ci fermeremo al livello x tale che $\frac{m}{2^x} = 1$, cioè $x = \log m$. Da ciò otteniamo

$$T(m) = \sum_{i=0}^{\log m} c = c \log m = \Theta(\log m)$$

Versione Non Ricorsiva. Nella versione non ricorsiva gestiamo i due indici i e j e li aggiorniamo all'interno di un ciclo while. Nel ciclo while k viene confrontato con l'elemento $A[h]$ che si trova in posizione centrale tra la posizione i e la posizione j. Se $A[h]$ è maggiore di k, l'indice j viene spostato verso il basso in posizione $h-1$. Se $A[h]$ è uguale a k si ritorna il valore h, altrimenti l'indice i viene spostato verso l'alto in posizione $h+1$.

Algorithm 3 WhileBisectSearch(A, k)

```
i ← 1
j ← length(A)
while i ≤ j do
   h ← (j + i)/2
   if A[h] > k then
      j ← h − 1
   else
      if A[h] = k then
         return k
      else
         i ← h + 1
      end if
   end if
end while
return −1
```

ESERCIZIO DI ASD DEL 20 OTTOBRE 2008

Distanze Minime

Sia A un vettore contenente n numeri interi positivi. Definiamo la distanza tra due elementi $A[i]$ e $A[j]$ di A come $d(A[i], A[j]) = |A[i] - A[j]|$.

Si consideri il problema di determinare due distinte posizioni a e b nel vettore A che minimizzino la distanza $d(A[a], A[b])$, ovvero

$$\forall c, d(d(A[c], A[d]) \geq d(A[a], A[b])$$

a.1 Si scriva lo pseudo-codice di un algoritmo per risolvere il problema sopra proposto.

a.2 Si dimostri la correttezza dell'algoritmo descritto.

a.3 Si determini la complessità nel caso peggiore dell'algoritmo descritto.

Date: 20 Ottobre 2008.

ESERCIZIO DI ASD DEL 20 OTTOBRE 2008

Distanze Minime

Sia A un vettore contenente n numeri interi positivi. Definiamo la distanza tra due elementi $A[i]$ e $A[j]$ di A come $d(A[i], A[j]) = |A[i] - A[j]|$.

Si consideri il problema di determinare due distinte posizioni a e b nel vettore A che minimizzino la distanza $d(A[a], A[b])$, ovvero

$$\forall c, d(d(A[c], A[d]) \geq d(A[a], A[b])$$

- a.1 Si scriva lo pseudo-codice di un algoritmo per risolvere il problema sopra proposto.
  Suggerimento: Copiare il vettore ed ordinarlo. Usare la versione ordinata per trovare gli elementi cercati in $\Theta(n)$.
- a.2 Si dimostri la correttezza dell'algoritmo descritto.
- a.3 Si determini la complessità nel caso peggiore dell'algoritmo descritto.
  Suggerimento: La complessità nel caso peggiore deve risultare $\Theta(n \log n)$.

Date: 20 Ottobre 2008.

ESERCIZIO DI ASD DEL 20 OTTOBRE 2008

Distanze Minime

Si osservi che in un vettore ordinato due elementi a distanza minima sono sicuramente consecutivi. MinDist(A) sfrutta questa proprietà ed opera nel seguente modo:

- copia il vettore A in un vettore B;
- ordina B;
- cerca in B due elementi consecutivi a distanza minima;
- ricerca in A le posizioni dei due elementi.

La copia in B è necessaria perchè l'esercizio chiede di ritornare le due posizioni degli elementi a distanza minima e non i due elementi.

Algorithm 1 MinDist(A)

```
Copia(A, B)
MergeSort(B, 1, length(B))
i ← CercaMinDist(B)
a ← Cerca(A, B[i])
b ← Cerca(A, B[i + 1])
return a and b
```

Algorithm 2 Copia(A, B)

```
for i ← 1 to length(A) do
    B[i] ← A[i]
end for
```

Algorithm 3 CercaMinDist(B)

```
min ← B[2] − B[1]
minpos ← 1
for i ← 2 to length(B) − 1 do
    if B[i + 1] − B[i] < min then
        min ← B[i + 1] − B[i]
        minpos ← i
    end if
end for
return minpos
```

Date: 20 Ottobre 2008.

Algorithm 4 CERCA(A, x)

```
i ← 1
while A[i] ≠ x do
    i ← i + 1
end while
return i
```

Correttezza. La correttezza della procedura COPIA è banale. La correttezza della procedura MERGESORT è stata dimostrata a lezione.

Dimostriamo la correttezza della procedura CERCAMINDIST(B). Iniziamo dimostrando l'invariante per il ciclo for.

Invariante 1. *All'inizio della i-esima iterazione del ciclo for di* CERCAMINDIST(B) *la variabile min contiene la distanza minima tra gli elementi in $B[1..i]$ e la distanza minima è raggiunta nelle posizioni $minpos$ e $minpos + 1$, ovvero*

$$d(B[minpos], B[minpos + 1]) = min$$

Dimostrazione. Procediamo per induzione su i.
Base: $i = 2$

La variabile min contiene $B[2] - B[1] = d(B[1], B[2])$ e $minpos$ vale 1. In $B[1..2]$ ci sono solo gli elementi $B[1]$ e $B[2]$, quindi la tesi è soddisfatta.
Passo:

HpInd) All'inizio della $k-1$-esima iterazione del ciclo for la variabile min contiene la distanza minima in $B[1..k-1]$ e $d(B[minpos], B[minpos + 1]) = min$.

TsInd) All'inizio della k-esima iterazione del ciclo for la variabile min contiene la distanza minima in $B[1..k]$ e $d(B[minpos], B[minpos + 1]) = min$.

Dobbiamo analizzare ciò che accade durante l'esecuzione della $k - 1$-esima iterazione. Se $B[k] - B[k-1] = d(B[k], B[k-1]) < min$, min prende valore $d(B[k], B[k-1])$ e $minpos$ prende valore $k - 1$. Dall'ipotesi induttiva sappiamo che sicuramente non ci sono due elementi a distanza minore di min in $B[1..k-1]$. L'unica cosa da dimostrare è che non è possibile che esista $j < k - 1$ tale che $d(B[k], B[j]) < min$. Il vettore B è ordinato quindi se $j < k - 1$ abbiamo che $B[j] \leq B[k-1]$ e quindi $B[k] - B[j] \geq B[k] - B[k-1] = min$. Se $B[k] - B[k-1] = d(B[k], B[k-1]) \geq min$, allora min e $minpos$ non vengono aggiornati ed in modo analogo al caso precedente otteniamo la tesi. □

Lemma 1 (Correttezza di CERCAMINDIST(B))**.** CERCAMINDIST(B) *termina sempre ed al termine $minpos$ è tale che $B[minpos]$ e $B[minpos + 1]$ sono a distanza minima.*

Dimostrazione. CERCAMINDIST(B) termina sicuramente dopo la terminazione del ciclo for. Il ciclo for termina quando i prende valore $length(B)$. Dall'invariante abbiamo che al termine del ciclo for min contiene la distanza minima tra gli elementi in $B[1..length(B)]$ e $d(B[minpos], B[minpos + 1]) = min$ che è la tesi. □

Lemma 2 (Correttezza di CERCA(A, x))**.** *Se x è un elemento di A, allora la procedura* CERCA(A, x) *termina ed al termine $A[i] = x$.*

Dimostrazione. Indichiamo con k la prima posizione in cui l'elemento x occorre in A. L'indice i parte da 1 e viene solo incrementato all'interno del while. Quindi dopo $k - 1$ iterazioni del while l'indice i assume valore k. Alla k-esima iterazione del while viene eseguito il test $A[k] \neq x$ che fallisce. Il ciclo while termina a questo punto ed il valore che viene restituito è k. $\square$

Complessità. Sia $n = length(A) = length(B)$. $\textsc{MergeSort}(B, 1, length(B))$ ha complessità $\Theta(n \log n)$. Tutte le altre procedure utilizzate hanno complessità $\Theta(n)$. Quindi la complessità totale è $\Theta(n \log n)$.

ESERCIZIO DI ASD DEL 3 NOVEMBRE 2008

Modifica nella Heap

Sia A un vettore contenente una max-heap di interi di dimensione n. Sia k un numero intero ed i un indice nella heap, ovvero $1 \leq i \leq n$. Si consideri il problema di modificare l'i-esimo elemento della heap A aggiungendogli k, ovvero $A[i]$ viene sostituito con $A[i] + k$, e di ripristinare la max-heap.

- a.1 Si scriva lo pseudo-codice di una procedura ricorsiva per risolvere il problema proposto.
- a.2 Si scriva lo pseudo-codice di una procedura non ricorsiva per risolvere il problema proposto.
- a.3 Si dimostri la correttezza della procedura non ricorsiva proposta.
- a.4 Si determini la complessità della procedura non ricorsiva proposta.

Date: 3 Novembre 2008.

ESERCIZIO DI ASD DEL 3 NOVEMBRE 2008

Modifica nella Heap

Sia A un vettore contenente una max-heap di interi di dimensione n. Sia k un numero intero ed i un indice nella heap, ovvero $1 \leq i \leq n$. Si consideri il problema di modificare l'i-esimo elemento della heap A aggiungendogli k, ovvero $A[i]$ viene sostituito con $A[i] + k$, e di ripristinare la max-heap.

a.1 Si scriva lo pseudo-codice di una procedura ricorsiva per risolvere il problema proposto. **Suggerimento**: considerare separatamente i due casi $k \geq 0$ e $k < 0$. Se $k \geq 0$ la modifica di $A[i]$ può comportare problemi verso l'alto. È quindi necessario scrivere una procedura che, fino a quando è necessario, scambia $A[j]$ con $A[parent(j)]$. Se $k < 0$ la modifica può comportare problemi verso il basso. In questo caso occorre effettuare scambi con i figli fino a quando è necessario.

a.2 Si scriva lo pseudo-codice di una procedura non ricorsiva per risolvere il problema proposto. **Suggerimento**: Modificare il codice delle procedure ricorsive sostituendo le chiamate ricorsive con dei cicli while. Può essere utile introdurre una variabile booleana ausiliaria che serve come guardia del ciclo while e che assume valore false solo quando si arriva al caso base.

a.3 Si dimostri la correttezza della procedura non ricorsiva proposta. **Suggerimento**: Le procedure devono contenere sicuramente almeno un ciclo (while). Procedere per induzione sulle iterazioni del ciclo.

a.4 Si determini la complessità della procedura non ricorsiva proposta. **Suggerimento**: è più facile ragionare in termini di altezza della heap e poi riscrivere il risultato in termini di nodi della heap.

Date: 3 Novembre 2008.

ESERCIZIO DI ASD DEL 3 NOVEMBRE 2008

Modifica nella Heap

Versione Ricorsiva. Distinguiamo due casi:

- Se $k \geq 0$ effettuiamo una correzione verso l'alto, sfruttando la procedura Correggi vista a lezione.
- Se $k < 0$ effettuiamo una correzione verso il basso, sfruttando la procedura Heapify vista a lezione.

Algorithm 1 HeapModify(A, i, k)

$A[i] \leftarrow A[i] + k$
if $k \geq 0$ **then**
    Correggi(A, i)
else
    Heapify(A, i)
end if

Algorithm 2 Correggi(A, i)

if $i > 1$ and $A[i] > A[$parent$(i)]$ **then**
    Scambia$(A, i,$parent$(i))$
    Correggi$(A,$parent$(i))$
end if

Algorithm 3 Heapify(A, i)

$\ell \leftarrow$left(i)
$r \leftarrow$right(i)
if $\ell \leq heapsize[A]$ and $A[\ell] > A[i]$ **then**
    $max \leftarrow \ell$
else
    $max \leftarrow i$
end if
if $r \leq heapsize[A]$ and $A[r] > A[max]$ **then**
    $max \leftarrow r$
end if
if $max \neq i$ **then**
    Scambia(A, i, max)
    Heapify(A, max)
end if

Date: 3 Novembre 2008.

Versione Ricorsiva. Dobbiamo fornire una versione non ricorsiva delle due procedure Correggi ed Heapify. Entrambe le procedure sono ricorsive di coda. La ricorsione si può facilmente rimpiazzare con un ciclo while. Introduciamo una variabile *loop* che indicherà quando il ciclo deve terminare. La variabile *loop* assume valore iniziale $True$. All'interno del ciclo while troviamo tutto il codice che avevamo nella versione ricorsiva. La chiamata ricorsiva viene sostituita con un'opportuna modifica del valore della variabile i. Aggiungiamo un caso else in cui alla variabile *loop* viene assegnato valore $False$. Questo corrisponde al caso in cui nella versione ricorsiva non venivano effettuate chiamate ricorsive.

Algorithm 4 NoRicCorreggi(A, i)

```
loop ← True
while loop do
   if i > 1 and A[i] > A[PARENT(i)] then
      SCAMBIA(A, i,PARENT(i))
      i ←PARENT(i))
   else
      loop ← False
   end if
end while
```

Algorithm 5 NoRicHeapify(A, i)

```
loop ← True
while loop do
   ℓ ←LEFT(i)
   r ←RIGHT(i)
   if ℓ ≤ heapsize[A] and A[ℓ] > A[i] then
      max ← ℓ
   else
      max ← i
   end if
   if r ≤ heapsize[A] and A[r] > A[max] then
      max ← r
   end if
   if max ≠ i then
      SCAMBIA(A, i, max)
      i ← max
   else
      loop ← False
   end if
end while
```

Nel caso della procedura Correggi è facile anche immaginare una versione non ricorsiva semplificata senza la variabile ausiliaria *loop*. Chiamiamo NoRicCorreggi2 la versione semplificata.

La versione non ricorsiva di HeapModify si ottiene semplicemente sostituendo le chiamate alle procedure Correggi e Heapify con chiamate alle versioni

Algorithm 6 NoRicCorreggi2(A, i)

while $i > 1$ and $A[i] > A[\text{PARENT}(i)]$ **do**
  Scambia(A, i,PARENT(i))
  $i \leftarrow$PARENT(i))
end while

non ricorsive. In particolare nel resto della trattazione facciamo riferimento alla procedura NoRicCorreggi2 come versione non ricorsiva di Correggi.

Correttezza. Dimostriamo la correttezza di NoRicCorreggi2. Indichiamo con $brother(k)$ l'indice in cui si trova il fratello del nodo che ha indice k: se k è pari $brother(k)$ è $k+1$, se k è dispari $brother(k)$ è $k-1$.

Lemma 1 (while in NoRicCorreggi2)**.** *Se $A[k]$ e $A[brother(k)]$ sono radici di heap e $A[2k], A[2k+1], A[brother(k)] \leq A[parent(k)]$, allora dopo una esecuzione del ciclo while della procedura* NoRicCorreggi2 *con indice i uguale a k vale che $A[parent(k)]$ è radice di heap.*

Dimostrazione. Se $A[k] \leq A[parent(k)]$, allora il codice non fa niente ed infatti abbiamo che $A[parent(k)]$ è radice di heap visto che $A[k]$ ed $A[brother(k)]$ sono radici di heap e $A[k], A[brother(k)] \leq A[parent(k)]$.

Se $A[k] > A[parent(k)]$, allora siano $A[k] = y$, $A[parent(k)] = x$, $A[brother(k)] = z$. Il codice scambia x ed y. A questo punto x si trova in posizione k ed ha come figli $A[2k]$ e $A[2k+1]$ che sono radici di heap per ipotesi. Inoltre per ipotesi $A[2k], A[2k+1] \leq x$, quindi in posizione k abbiamo una heap. $A[brother(k)]$ era radice di heap e tale è rimasto, visto che non si sono toccati nodi che stanno nel suo sottoalbero. y ora si trova in posizione $parent(k)$ ed ha come figli z e x che sono due radici di heap e che soddisfano la proprietà $z \leq x < y$. Quindi in posizione $parent(k)$ abbiamo una heap. □

Invariante 1 (while in NoRicCorreggi2)**.** *Se all'inizio dell'esecuzione della procedura* NoRicCorreggi2(A, i) *A è una heap tranne al più l'errore $A[i] > A[parent(i)]$, allora all'inizio dell'iterazione del ciclo while in cui i vale k abbiamo che $A[k]$ e $A[brother(k)]$ sono radici di heap, $A[2k], A[2k+1], A[brother(k)] \leq A[parent(k)]$ e in A c'è al più l'errore $A[k] > A[parent(k)]$.*

Dimostrazione. Procediamo per induzione su k.

BASE. $k = i$.

Per ipotesi abbiamo che A è una heap tranne al più l'errore $A[i] > A[parent(i)]$, quindi $A[i]$ e $A[brother(i)]$ sono radici di heap e $A[2i], A[2i+1], A[brother(i)] \leq A[parent(i)]$.

PASSO.

HpInd) Supponiamo che all'inizio dell'iterazione in cui i vale h valga che $A[h]$ e $A[brother(h)]$ sono radici di heap, $A[2h], A[2h+1], A[brother(h)] \leq A[parent(h)]$ e in A c'è al più l'errore $A[h] > A[parent(h)]$.

Ts) All'inizio dell'iterazione in cui i vale $h/2$ abbiamo che $A[h/2]$ e $A[brother(h/2)]$ sono radici di heap, $A[2(h/2)], A[2(h/2)+1], A[brother(h/2)] \leq A[parent(h/2)]$, e in A c'è al più l'errore $A[h/2] > A[parent(h/2)]$.

Dobbiamo analizzare cosa avviene durante l'esecuzione del ciclo while con $i = h$. Dal Lemma 1 abbiamo che l'esecuzione del ciclo while rende $A[parent(h)] = A[h/2]$

radice di heap. Per ipotesi in A c'era al più l'errore $A[h] > A[parent(h)]$. Lo scambio tra $A[h]$ e $A[parent(h)]$ ha al più spostato l'errore in $A[h/2] > A[parent(h/2)]$, quindi $A[brother(h/2)]$ è radice di heap e $A[2(h/2)], A[2(h/2)+1], A[brother(h/2)] \leq A[parent(h/2)]$. □

Lemma 2 (Correttezza di NoRicCorreggi2(A, i))**.** *NoRicCorreggi2(A, i) termina sempre. Se A è una heap tranne al più l'errore $A[i] > A[parent(i)]$, allora al termine di NoRicCorreggi2(A, i) A è una heap.*

Dimostrazione. Il ciclo while che costituisce NoRicCorreggi2(A, i) termina sempre perché tra le guardie del ciclo abbiamo la condizione $i > 1$ e l'unica istruzione all'interno del ciclo che modifica la variabile i la dimezza ogni volta. Quindi dopo al più un numero finito di esecuzioni del ciclo while la variabile i raggiunge il valore 1.

Dimostriamo che se A è una heap tranne al più l'errore $A[i] > A[parent(i)]$, allora al termine di NoRicCorreggi2(A, i) A è una heap. Ci sono due possibilità:

- se il ciclo while termina perché $i \leq 1$, allora dall'invariante sappiamo che non ci sono errori ed A è una heap (l'errore sarebbe sopra alla radice);
- se il ciclo while termina perché $A[i] \leq A[parent(i)]$, allora sempre dall'invariante sappiamo che A non contiene errori ed è una heap.

□

La correttezza di NoRicHeapify si dimostra in modo simile.

La correttezza di HeapModify segue immediatamente dalla correttezza di NoRicCorreggi2 e NoRicHeapify.

Complessità. Sia $n = length[A] = heapsize[A]$.

La procedura NoRicCorreggi2 esegue un ciclo while. Tutte le operazioni all'interno del ciclo while (compresa l'intestazione del ciclo while) hanno complessità $\Theta(1)$. Quindi per determinare la complessità della procedura dobbiamo determinare quante volte viene eseguito al più il ciclo while. Ad ogni iterazione del ciclo while ci si sposta verso l'alto di un livello sulla heap A. Quindi al più il ciclo while può essere eseguito tante volte quanti sono i livelli della heap A (ovvero tante volte quanta è l'altezza di A). Per quanto visto a lezione l'altezza di A è $\Theta(\log n)$. Quindi la complessità della procedura è $O(\log n)$.

Analogamente, ma scendendo verso il basso, si dimostra che la complessità della procedura NoRicHeapify è $O(\log n)$.

Quindi la complessità della procedura HeapModify è $O(\log n)$.

ESERCIZIO DI ASD DEL 10 NOVEMBRE 2008

Intervalli

Si supponga di rappresentare mediante una coppia di interi $[l, u]$ con $l \leq u$, l'intervallo di interi compresi tra l ed u (estremi inclusi). Sia dato l'insieme di n intervalli $S = \{[l_1, u_1], [l_2, u_2], \ldots, [l_n, u_n]\}$.

1 Si scriva lo pseudocodice di un algoritmo efficiente per determinare se esistono (almeno) due intervalli disgiunti (aventi intersezione vuota) in S.
2 Si dimostri la correttezza della procedura proposta.
3 Si determini la complessità della procedura proposta.
4 Si scriva lo pseudocodice di un algoritmo efficiente per determinare se esistono (almeno) due intervalli **NON** disgiunti in S.

Date: 10 Novembre 2008.

ESERCIZIO DI ASD DEL 10 NOVEMBRE 2008

Intervalli

Si supponga di rappresentare mediante una coppia di interi $[l, u]$ con $l \leq u$, l'intervallo di interi compresi tra l ed u (estremi inclusi). Sia dato l'insieme di n intervalli $S = \{[l_1, u_1], [l_2, u_2], \ldots, [l_n, u_n]\}$.

1 Si scriva lo pseudocodice di un algoritmo efficiente per determinare se esistono (almeno) due intervalli disgiunti (aventi intersezione vuota) in S.
Suggerimento: Affinché ci siano almeno due intervalli disgiunti il più piccolo degli estremi di destra deve essere minore del più grande degli estremi di sinistra.

2 Si dimostri la correttezza della procedura proposta.
Suggerimento: dimostrare che ci sono due intervalli disgiunti se e soltanto se la condizione che viene testata nell'algoritmo è soddisfatta.

3 Si determini la complessità della procedura proposta.

4 Si scriva lo pseudocodice di un algoritmo efficiente per determinare se esistono (almeno) due intervalli **NON** disgiunti in S.
Suggerimento: In questo caso occorre che un intevallo termini dopo l'inizio dell'intervallo successivo. Quindi occorrerà ordinare gli intervalli rispetto agli estremi di sinistra.

Date: 10 Novembre 2008.

ESERCIZIO DI ASD DEL 10 NOVEMBRE 2008

Intervalli

Algoritmo Intervalli Disgiunti. Condizione necessaria e sufficiente affinché tra n intervalli della forma $[l_i, u_i]$ ce ne siano almeno due disgiunti è che quello che termina più a sinistra termini prima dell'inizio di quello che inizia più a destra. Per esempio se consideriamo $S = \{[1,5],[2,8],[4,6]\}$ abbiamo che l'intervallo che termina più a sinistra è $[1,5]$ e l'intervallo che inizia più a destra è $[4,6]$. Visto che 5 è maggiore o uguale di 4 non ci sono intervalli disgiunti. Infatti $[1,5]$ e $[2,8]$ non sono disgiunti, $[1,5]$ e $[4,6]$ non sono disgiunti, $[2,8]$ e $[4,6]$ non sono disgiunti. Se invece consideriamo $S = \{[1,5],[2,8],[7,9]\}$ abbiamo che l'intervallo che termina più a sinistra è $[1,5]$ e l'intervallo che inizia più a destra è $[7,9]$. Visto che 5 è minore di 7 abbiamo che ci sono due intervalli disgiunti. Infatti $[1,5]$ e $[7,9]$ sono disgiunti.

Descriviamo quindi un algoritmo che si basa su questa condizione necessaria e sufficiente. Supponiamo che l'insieme S sia memorizzato in un vettore V di lunghezza n i cui elementi hanno due campi: $V[i].lower = l_i$ è l'estremo di sinistra dell'i-esimo intervallo e $V[i].upper = u_i$ è l'estremo di destra. Eseguremo un unico ciclo for durante il quale determineremo il minimo degli estremi di destra ed il massimo degli estremi di sinistra. Controlleremo che il minimo degli estremi di destra sia minore del massimo degli estremi di sinistra.

Algorithm 1 DUE_INTERVALLI_DISGIUNTI(V)

```
min_u ← V[1].upper
max_l ← V[1].lower
for i ← 2 to length[V] do
    if V[i].upper < min_u then
        min_u ← V[i].upper
    end if
    if V[i].lower > max_l then
        max_l ← V[i].lower
    end if
end for
if min_u < max_l then
    return TRUE
else
    return FALSE
end if
```

Date: 10 Novembre 2008.

Correttezza. Dimostriamo che la condizione su cui abbiamo basato il nostro algoritmo è corretta.

Lemma 1. *Sia* $S = \{[l_1, u_1], [l_2, u_2], \ldots, [l_n, u_n]\}$ *un insieme di* n *intervalli chiusi (estremi inclusi) con* $l_i \leq u_i$. *Esistono* $i, j \leq n$ *tali che* $[l_i, u_i] \cap [l_j, u_j] = \emptyset$ *se e soltanto se il minimo dell'insieme* $U = \{u_1, u_2, \ldots, u_n\}$ *è minore del massimo dell'insieme* $L = \{l_1, l_2, \ldots, l_n\}$.

Dimostrazione. $\Rightarrow$)

Hp) Esistono $i, j \leq n$ tali che $[l_i, u_i] \cap [l_j, u_j] = \emptyset$.

Ts) Il minimo dell'insieme $U = \{u_1, u_2, \ldots, u_n\}$ è minore del massimo dell'insieme $L = \{l_1, l_2, \ldots, l_n\}$.

Visto che $[l_i, u_i] \cap [l_j, u_j] = \emptyset$ deve essere $u_i < l_j$. Sia min_u il minimo dell'insieme U e max_l il massimo dell'insieme L. Abbiamo che $min\_u \leq u_i < l_j \leq max\_l$, da cui otteniamo $min\_u < max\_l$.

$\Leftarrow$)

Hp) Il minimo dell'insieme $U = \{u_1, u_2, \ldots, u_n\}$ è minore del massimo dell'insieme $L = \{l_1, l_2, \ldots, l_n\}$.

Ts) Esistono $i, j \leq n$ tali che $[l_i, u_i] \cap [l_j, u_j] = \emptyset$.

Sia u_i il minimo dell'insieme U ed l_j il massimo dell'insieme L. Per ipotesi abbiamo che $u_i < l_j$. Quindi visto che $l_i \leq u_i$ abbiamo che sicuramente u_i ed l_j sono estremi di due intervalli distinti. Se consideriamo i due intervalli $[l_i, u_i]$ e $[l_j, u_j]$ abbiamo che visto che $u_i < l_j$ vale che $[l_i, u_i] \cap [l_j, u_j] = \emptyset$ □

Ora possiamo dimostrare la correttezza della procedura proposta. Come prima cosa dimostriamo l'invariante per il ciclo for.

Invariante 1. *All'inizio della* i*-esima iterazione del ciclo for abbiamo che:*

- *per ogni* j *tale che* $1 \leq j < i$ *vale che* $V[j].upper \geq min\_u$*;*
- *per ogni* j *tale che* $1 \leq j < i$ *vale che* $V[j].lower \leq max\_l$.

Dimostrazione. Procediamo per induzione su i.

BASE) $i = 2$. All'inizio dell'iterazione in cui i vale 2 abbiamo che $min\_u = V[1].upper$ e $max\_l = V[1].lower$. Quindi vale la tesi.

PASSO)

HpInd) All'inizio dell'iterazione in cui i vale $k - 1$ abbiamo che per ogni j tale che $1 \leq j < k - 1$ vale che $V[j].upper \geq min\_u$ e $V[j].lower \leq max\_l$.

Ts) All'inizio dell'iterazione in cui i vale k abbiamo che per ogni j tale che $1 \leq j < k$ vale che $V[j].upper \geq min\_u$ e $V[j].lower \leq max\_l$.

Dobbiamo esaminare ciò che accade durante l'iterazione del ciclo for in cui i vale $k - 1$. Durante tale iterazione vengono esaminati $V[k-1].upper$ e $V[k-1].lower$. Se $V[k-1].upper \geq min\_u$, allora min_u non viene modificato. In tal caso abbiamo dall'ipotesi induttiva che per ogni j tale che $1 \leq j < k - 1$ vale che $V[j].upper \geq min\_u$, inoltre vale che $V[k-1].upper \geq min\_u$, quindi otteniamo che per ogni j tale che $1 \leq j < k$ vale che $V[j].upper \geq min\_u$. Se invece $V[k-1].upper = a < min\_u = b$, allora a min_u viene assegnato valore a. Avevamo che per ipotesi induttiva valeva $V[j].upper \geq b$ per ogni j tale che $1 \leq j < k - 1$, quindi visto che $a < b$ a maggior ragione abbiamo $V[j].upper \geq a$ per ogni j tale che $1 \leq j < k - 1$. Inotre abbiamo $V[k-1].upper = a$ quindi possiamo concludere che vale $V[j].upper \geq a = min\_u$ per ogni j tale che $1 \leq j < k$.

Analogamente si dimostra che vale $V[j].lower \leq max\_l$ per ogni $1 \leq j < k$. □

Theorem 1. DUE_INTERVALLI_DISGIUNTI(V) *termina sempre e restituisce* $TRUE$ *se e soltanto se tra gli intervalli* $\{[V[1].lower, V[1].upper]$, $[V[2].lower, V[2].upper]$, $\dots$, $[V[length[V]].lower, V[length[V]].upper]\}$ *ce ne sono due disgiunti.*

Dimostrazione. DUE_INTERVALLI_DISGIUNTI(V) è costituito principalmente da un ciclo for che termina sicuramente.

Il ciclo for termina quando la variabile i assume valore $length[V]+1$. Dall'invariante abbiamo che al termine del ciclo for vale che per ogni j compreso tra 1 e $length[V]$ vale che $V[j].upper \geq min\_u$ e $V[j].lower \leq max\_l$.

Dal Lemma 1 abbiamo che tra gli intervalli dell'insieme $\{[V[1].lower, V[1].upper]$, $[V[2].lower, V[2].upper]$, $\dots$, $[V[length[V]].lower, V[length[V]].upper]\}$ ce ne sono due disgiunti se e soltanto se $min\_u < max\_l$ e l'algoritmo termina restituendo $TRUE$ solo in questo caso. □

Complessità. Sia $n = length[V]$.

La procedura DUE_INTERVALLI_DISGIUNTI esegue un ciclo for di lunghezza n. Quindi la complessità della procedura è $\Theta(n)$.

Algoritmo Intervalli Non Disgiunti. Condizione necessaria e sufficiente affinché tra n intervalli della forma $[l_i, u_i]$ ce ne siano almeno due non disgiunti è che ci sia un intervallo che termina dopo (o contemporaneamente con) l'inizio dell'intervallo successivo. Per esempio se consideriamo $S = \{[1,5],[6,8],[10,11]\}$ abbiamo che ogni intervallo termina prima dell'inizio del successivo. Quindi non ci sono due intervalli non disgiunti. Se invece consideriamo $S = \{[1,5],[4,8],[10,11]\}$ abbiamo che l'intervallo $[1,5]$ termina in 5, mentre l'intervallo successivo è $[4,8]$ che inizia in 4. Visto che $5 \geq 4$ abbiamo che i due intervalli sono non disgiunti. Va osservato che per parlare di intervallo successivo ad un dato intervallo abbiamo considerato gli intervalli ordinati rispetto agli estremi di sinistra.

Descriviamo quindi un algoritmo che si basa su questa condizione necessaria e sufficiente. Come prima supponiamo che l'insieme S sia memorizzato in un vettore V di lunghezza n i cui elementi hanno due campi: $V[i].lower = l_i$ è l'estremo di sinistra dell'i-esimo intervallo e $V[i].upper = u_i$ è l'estremo di destra. Ordineremo il vettore V rispetto ai valori contenuti nei campi *lower*. Poi eseguiremo una scansione e controlleremo se $V[i].upper \geq V[i+1].lower$. Se questa condizione è soddisfatta abbiamo trovato due intervalli non disgiunti. Se lungo tutta la lunghezza del vettore la condizione non è mai soddisfatta possiamo concludere che nel vettore non ci sono due intervalli non disgiunti.

Algorithm 2 DUE_INTERVALLI_NON_DISGIUNTI(V)

```
MERGESORT(V, lower)
for i ← 1 to length[V] − 1 do
   if V[i].upper ≥ V[i+1].lower then
      return TRUE
   end if
end for
return FALSE
```

Anche se non è richiesto dall'esercizio dimostriamo parzialmente la correttezza della soluzione proposta dimostrando la correttezza della condizione necessaria e sufficiente.

Lemma 2. *Sia Sia* $S = \{[l_1, u_1], [l_2, u_2], \ldots, [l_n, u_n]\}$ *un insieme di* n *intervalli chiusi (estremi inclusi) con* $l_i \leq u_i$ *e tale che* $l_1 \leq l_2 \leq \cdots \leq l_n$*, ovvero gli estremi di sinistra sono ordinati. Esistono* $i, j \leq n$ *tali che* $[l_i, u_i] \cap [l_j, u_j] \neq \emptyset$ *se e soltanto se esiste* k *tale che* $u_k \geq l_{k+1}$*.*

Dimostrazione. Assumiamo che gli estremi di sinistra degli intervalli siano tutti distinti (se non lo sono il lemma è immediato).

$\Rightarrow$)

Hp) Esistono $i, j \leq n$ tali che $[l_i, u_i] \cap [l_j, u_j] \neq \emptyset$.

Ts) Esiste k tale che $u_k \geq l_{k+1}$.

Non è restrittivo supporre che l_i venga prima di l_j nell'ordinamento. Quindi abbiamo che $l_i < l_{i+1} \leq l_j$. Visto che i due intervalli si intersecano e $l_i < l_j$ deve essere $u_i \geq l_j$. Ma visto che $l_{i+1} \leq l_j$ e $u_i \geq l_j$ otteniamo che $u_i \geq l_{i+1}$. Quindi vale la tesi con $k = i$.

$\Leftarrow$)

Hp) Esiste k tale che $u_k \geq l_{k+1}$.

Ts) Esistono $i, j \leq n$ tali che $[l_i, u_i] \cap [l_j, u_j] \neq \emptyset$.

Consideriamo gli intervalli $[l_k, u_k]$ e $[l_{k+1}, u_{k+1}]$. Visto che $l_k < l_{k+1}$ e $u_k \geq l_{k+1}$ abbiamo che i due intervalli si intersecano. Quindi vale la tesi con $i = k$ e $j = k + 1$. □

In questo caso la complessità della procedura è data dalla complessità dell'algoritmo di ordinamento ed è quindi $\Theta(n \log n)$, dove $n = length[V]$.

ESERCIZIO DI ASD DEL 17 NOVEMBRE 2008

Pochi da Ordinare

Sia A un vettore di lunghezza n di interi positivi contenente k elementi distinti, con k costante rispetto alla dimensione del vettore. Si consideri il problema di ordinare A.

1 Si scriva lo pseudocodice di un algoritmo di ordinamento stabile avente complessità lineare per ordinare A.
2 Si scriva lo pseudocodice di un algoritmo di ordinamento in place avente complessità lineare per ordinare A.
3 Si controlli la complessità delle procedure proposte.

Date: 17 Novembre 2008.

ESERCIZIO DI ASD DEL 17 NOVEMBRE 2008

Pochi da Ordinare

Sia A un vettore di lunghezza n di interi positivi contenente k elementi distinti, con k costante rispetto alla dimensione del vettore. Si consideri il problema di ordinare A.

2 Si scriva lo pseudocodice di un algoritmo di ordinamento stabile avente complessità lineare per ordinare A.
Suggerimento: Modificare SelectionSort oppure modificare CountingSort.

1 Si scriva lo pseudocodice di un algoritmo di ordinamento in place avente complessità lineare per ordinare A.
Suggerimento: Modificare ulteriormente SelectionSort.

3 Si controlli la complessità delle procedure proposte.

Date: 17 Novembre 2008.

ESERCIZIO DI ASD DEL 17 NOVEMBRE 2008

Pochi da Ordinare

Algoritmo di Ordinamento Stabile. Analizzando gli algoritmi di ordinamento che conosciamo ci accorgiamo che abbiamo almeno due possibilità:

- Modificare SelectionSort utilizzando un vettore ausiliario;
- Modificare CountingSort per fare in modo che operi su k interi distinti, anche se molto grandi.

Proponiamo la soluzione per entrambe le alternative.

SelectionSort ricerca ad ogni iterazione il minimo tra gli elementi rimasti da considerare e lo colloca nella posizione esatta. In generale ha complessità $\Theta(n^2)$. Tuttavia possiamo ottimizzarlo per operare su vettori con k elementi distinti nel seguente modo:

- Ricerco il minimo per k volte (ogni volta solo tra gli elementi rimasti). Per fare questo mi basta una scansione del vettore.
- Una volta trovato il minimo eseguo un'altra scansione del vettore e copio tutte le occorrenze del minimo in un vettore ausiliario che conterrà il risultato finale.

Per essere sicuri di non considerare più volte gli stessi elementi, ogni volta che un numero viene copiato in B il suo valore in A viene modificato assegnandogli un valore maggiore del massimo di A.

Algorithm 1 RiSelectionSort(A, k)

```
max ← TrovaMax(A)
h ← 1
for i ← 1 to k do
   min ← TrovaMin(A)
   for j ← 1 to length[A] do
      if A[j] = min then
         B[h] ← A[j]
         h ← h + 1
         A[j] ← max + 1
      end if
   end for
end for
```

Passiamo alla seconda soluzione, che consiste nel modificare CountingSort. Il vettore A contiene relativamente pochi elementi distinti, in quanto k è una costante e non aumenta all'aumentare della dimensione del vettore. Purtroppo non possiamo applicare immediatamente CountingSort perché gli elementi di A potrebbero essere molto grandi. Per esempio A potrebbe contenere k elementi distinti

Date: 17 Novembre 2008.

Algorithm 2 TROVAMAX(A)

$max \leftarrow A[1]$
for $i \leftarrow 2$ to $length[A]$ **do**
  if $A[i] > max$ **then**
    $max \leftarrow A[i]$
  end if
end for
return max

Algorithm 3 TROVAMIN(A)

$min \leftarrow A[1]$
for $i \leftarrow 2$ to $length[A]$ **do**
  if $A[i] < min$ **then**
    $min \leftarrow A[i]$
  end if
end for
return min

compresi tra n^{100} e n^{1000}. Tuttavia possiamo prendere spunto da COUNTINGSORT per disegnare un algoritmo di ordinamento stabile e lineare adatto al problema proposto.

Utilizziamo un vettore C di lunghezza k i cui elementi hanno due campi: $C[i].key$ che conterrà uno dei k elementi che compaiono in A e $C[i].occ$ che ci dirà quante occorrenze di $C[i].key$ si trovano in A. Con una scansione del vettore A possiamo riempire il vettore C, esattamente come si fa in COUNTINGSORT, con l'unica differenza che per ogni elemento di A che consideriamo dobbiamo trovare la sua "posizione" in C.

Una volta riempito il vettore C possiamo ordinarlo rispetto al campo *key*. Questo ci costerà poco, indipendentemente dall'algoritmo di ordinamento che useremo perché C ha dimensione k.

Ora possiamo procedere come in COUNTINGSORT, "sommando" gli elementi di C e poi scrivendo il risultato finale in B.

Algoritmo di Ordinamento in Place. Cerchiamo di sistemare una delle due soluzioni prima proposte per ottenere un algoritmo in place, con il rischio di perdere la stabilità. SELECTIONSORT usa solo un vettore ausiliario B per il risultato, quindi sembra più facile da adattare. Procediamo nel seguente modo:

- manteniamo un indice j che ci dirà quale è la porzione di A ancora da considerare;
- eseguiamo come prima la ricerca del minimo per k volte, ma ora cercheremo il minimo in $A[j..length[A]]$;
- scandiamo nuovamente $A[j..length[A]]$ per portare ogni occorrenza del minimo nella posizione corretta eseguendo degli scambi.

Quindi all'interno della i-esima iterazione di ricerca e sistemazione dei minimi ci servirà un altro indice *pos* che ci dirà dove posizionare le occorrenze del minimo. In particolare *pos* verrà inizializzato all'inizio del ciclo con valore j ed al termine

Algorithm 4 RiCountingSort(A, k)

```
for i ← 1 to k do
   C[i].key ← −1
   C[i].occ ← 0
end for
for j ← 1 to length[A] do
   i ← 1
   while C[i].key ≠ −1 and C[i].key ≠ A[j] do
      i ← i + 1
   end while
   C[i].key ← A[j]
   C[i].occ ← C[i].occ + 1
end for
InsertionSort(C)
for i ← 2 to k do
   C[i].occ ← C[i − 1].occ + C[i].occ
end for
for j ← length[A] down to 1 do
   i ← 1
   while C[i].key ≠ A[j] do
      i ← i + 1
   end while
   B[C[i].occ] ← A[j]
   C[i].occ ← C[i].occ − 1
end for
```

del ciclo verrà utilizzato per aggiornare j. È possibile evitare l'utilizzo di questo indice aggiuntivo sostituendo il ciclo for con un while.

Algorithm 5 RiRiSelectionSort(A, k)

```
j ← 1
for i ← 1 to k do
   pos ← j
   min ←RiTrovaMin(A, j, length[A])
   for h ← j to length[A] do
      if A[h] = min then
         Scambia(A, h, pos)
         pos ← pos + 1
      end if
   end for
   j ← pos
end for
```

Si noti che RiRiSelectionSort non è stabile. Trovate un vettore che dimostri questa affermazione.

Complessità. Tutti gli algoritmi presentati hanno complessità $\Theta(n)$. Infatti al massimo troviamo due cicli for innestati, di cui uno con indici che variano da 1 ad

Algorithm 6 RiTrovaMin(A, r, s)

```
min ← A[r]
for i ← r + 1 to s do
    if A[i] < min then
        min ← A[i]
    end if
end for
return min
```

n e l'altro con gli indici che variano da 1 a k. Quindi la complessità è al più $O(n*k)$ e quest'ultima coincide con $O(n)$, visto che k è costante. Inoltre la complessità è almeno $\Omega(n)$ in quanto abbiamo dei cicli for con indici che variano da 1 ad n che vengono sicuramente eseguiti.

ESERCIZIO DI ASD DEL 24 NOVEMBRE 2008

Liste cicliche

Sia L una lista concatenata i cui elementi contengono una chiave intera ed un puntatore all'elemento successivo in L. La lista L si dice ciclica se uno dei suoi elementi punta ad un suo predecessore nella lista, come mostrato in figura.

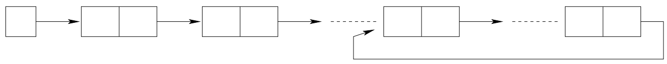

Figura 1. Lista concatenata ciclica.

1 Si scriva lo pseudocodice di un algoritmo **in place** che permette di determinare se L è ciclica, nel caso in cui L contenga solo interi maggiori di zero. Si noti che L può contenere ripetizioni. Al termine della procedura la lista L non deve essere cambiata.
2 Si scriva lo pseudocodice di un algoritmo che permette di determinare se L è ciclica nel caso in cui L contenga anche interi negativi. Al termine della procedura la lista L non deve essere cambiata.
3 Si dimostri la correttezza delle procedure proposte.
4 Si determini la complessità delle procedure proposte.

Date: 24 Novembre 2008.

ESERCIZIO DI ASD DEL 24 NOVEMBRE 2008

Liste cicliche

Sia L una lista concatenata i cui elementi contengono una chiave intera ed un puntatore all'elemento successivo in L. La lista L si dice ciclica se uno dei suoi elementi punta ad un suo predecessore nella lista, come mostrato in figura.

Figura 1. Lista concatenata ciclica.

1 Si scriva lo pseudocodice di un algoritmo **in place** che permette di determinare se L è ciclica, nel caso in cui L contenga solo interi maggiori di zero. Si noti che L può contenere ripetizioni. Al termine della procedura la lista L non deve essere cambiata.
Suggerimento: modificare gli elementi della lista, trasformando $key[x]$ in $-key[x]$. Se si raggiunge un elemento negativo allora la lista è ciclica. Al termine occorre ripristinare la lista.

2 Si scriva lo pseudocodice di un algoritmo che permette di determinare se L è ciclica nel caso in cui L contenga anche interi negativi. Al termine della procedura la lista L non deve essere cambiata.
Suggerimento: Modificare i puntatori degli elementi della lista, in modo da riconoscere gli elementi già visitati e nel frattempo copiare gli elementi in una nuova lista. Si noti che la nuova lista conterrà esattamente gli stessi elementi della lista L, ma non sarà ciclica.

3 Si dimostri la correttezza delle procedure proposte.

4 Si determini la complessità delle procedure proposte.
Suggerimento: Entrambe le procedure scandiscono la lista un numero costante di volte.

Date: 24 Novembre 2008.

ESERCIZIO DI ASD DEL 24 NOVEMBRE 2008

Liste Cicliche

Lista con Elementi Positivi. Ogni elemento x della lista L ha due campi $key[x]$ e $next[x]$. Per accedere al primo elemento della lista L si utilizza il campo $head[L]$ di L.

Scorriamo gli elementi della lista L. Trasformiamo la chiave di ogni elemento x che troviamo in $-key[x]$. Se raggiungiamo un elemento con chiave negativa, allora la lista è ciclica.

Algorithm 1 ListaPosCiclica(L)

```
if head[L] = NIL then
   ciclica ← False
else
   x ← head[L]
   while next[x] ≠ NIL and key[x] > 0 do
      key[x] ← −key[x]
      x ← next[x]
   end while
   if next[x] ≠ NIL then
      ciclica ← True
   else
      ciclica ← False
   end if
   x ← head[L]
   while next[x] ≠ NIL and key[x] < 0 do
      key[x] ← −key[x]
      x ← next[x]
   end while
end if
return ciclica
```

Lista con Elementi Interi. Una possibilità sarebbe quella di scandire la lista alla ricerca del massimo max della lista, per poi modificare gli elementi della lista trasformandoli tutti in $max + 1$. Se si raggiunge un elemento con chiave $max + 1$, allora la lista è ciclica. Purtroppo se la lista è ciclica la ricerca del massimo non termina. Non abbiamo quindi modo di modificare le chiavi degli elementi della lista in modo da essere sicuri di riconoscere gli elementi su cui si è già passati. Proviamo dunque a modificare i campi $next$ degli elementi che scandiamo, in modo

Date: 24 Novembre 2008.

da riconoscere gli elementi su cui siamo già passati. Per esempio possiamo modificare $next[x]$ ed assegnargli come valore x stesso. Se raggiungiamo un elemento il cui *next* è se stesso, allora o abbiamo raggiunto un elemento già visto oppure la lista ha un ciclo che contiene un solo elemento. In ogni caso la lista è ciclica. Nell'implementare questa procedura dobbiamo fare attenzione a due cose:

- visto che modificheremo $next[x]$, dobbiamo salvare $next[x]$ in una variabile temporanea y, altrimenti non riusciremo a procedere con la scansione
- l'esercizio richiede che al termine la lista non sia stata modificata, quindi procediamo anche a creare una copia della lista.

Usiamo il costrutto *new* per creare gli elementi della lista "copia" che creiamo

Algorithm 2 ListaCiclica(L)

```
if head[L] = NIL then
   ciclica ← False
else
   x ← head[L]
   new[z]
   key[z] ← key[x]
   head[L] ← z
   while next[x] ≠ NIL and next[x] ≠ x do
      y ← next[x]
      next[x] ← x
      new[w]
      key[w] ← key[y]
      next[z] ← w
      x ← y
      z ← w
   end while
   if next[x] ≠ NIL then
      ciclica ← True
   else
      ciclica ← False
   end if
   next[z] ← NIL
end if
return ciclica
```

Correttezza. Dimostriamo la correttezza della procedura ListaPosCiclica(L). Dimostriamo prima gli invarianti relativi ai cicli while.

Invariante 1. *Se L contiene solo elementi aventi chiavi maggiori di 0 ed L ha n elementi, allora per ogni $i \leq n$ all'inizio della i-esima iterazione del ciclo while x è l'i-esimo elemento di L, tutti gli elementi che precedono x hanno chiave negativa e tutti gli elementi che non sono ancora stati raggiunti, x compreso, hanno chiave positiva.*

Dimostrazione. Procediamo per induzione su i.
BASE $i = 1$. x è head L, tutti gli elementi hanno chiave positiva e non ci sono elementi che precedono x.
PASSO.
HPInd) Vale la tesi per $i < k$.
TS) Vale la tesi per $i = k$.

Dobbiamo analizzare cosa avviene durante la $k - 1$-esima iterazione del ciclo while. All'inizio dell'iterazione x è il $k - 1$-esimo elemento. Per ipotesi induttiva x ha chiave positiva. Durante il ciclo la chiave di x viene modificata e diventa negativa. Inoltre x viene spostato in avanti di una posizione nella lista L. Quindi vale la tesi. □

Quindi possiamo dimostrare che *ciclica* assume valore *True* se e soltanto se la lista L è ciclica.

Lemma 1. *ciclica assume valore True sse L è ciclica.*

Dimostrazione. Se *ciclica* assume valore *True*, allora abbiamo raggiunto un elemento con chiave negativa. Dall'invariante abbiamo che gli elementi con chiave negativa sono tutti e soli gli elementi che sono già stati analizzati. Quindi c'è un elemento di L che è stato analizzato due volte ed L è ciclica.

Se L è ciclica, allora nel ciclo while torneremo due volte sullo stesso elemento di L. Dall'invariante abbiamo che questo elemento avrà chiave negativa, quindi *ciclica* assume valore *True*. □

Da queste considerazioni segue la correttezza della procedura. Per motivi di tempo non riesco ad entrare maggiormente nei dettagli.

Complessità. È immediato osservare che entrambe le procedure descritte hanno complessità $\Theta(n)$, dove n è la lunghezza della lista L.

ESERCIZIO DI ASD DEL 1 DICEMBRE 2008

Fusione di Liste

Siano L_1 ed L_2 due liste concatenate ordinate le cui chiavi sono numeri interi. Si consideri il problema di fondere le due liste ottenendo un'unica lista ordinata.

1 Si scriva lo pseudocodice di una procedura per risolvere tale problema.
2 Si dimostri la correttezza della procedura proposta.
3 Si determini la complessità della procedura proposta.

Date: 1 Dicembre 2008.

ESERCIZIO DI ASD DEL 1 DICEMBRE 2008

Fusione di Liste

Siano L_1 ed L_2 due liste concatenate ordinate le cui chiavi sono numeri interi. Si consideri il problema di fondere le due liste ottenendo un'unica lista ordinata.

1 Si scriva lo pseudocodice di una procedura per risolvere tale problema.
Suggerimento: Modificare la procedura Merge, per fare in modo che lavori su liste. Evitare di scrivere un'unica procedura, ma scrivere separatamente le procedure per l'inserimento e la cancellazione nelle liste. Se necessario utilizzare anche il puntatore *tail*.

2 Si dimostri la correttezza della procedura proposta.
Suggerimento: Procedere per induzione sulla lunghezza delle liste.

3 Si determini la complessità della procedura proposta.
Suggerimento: la complessità sarà lineare. In oltre, se gli elementi sono stati opportunamente cancellati dalle liste L_1 ed L_2 ed inseriti nella nuova lista, la procedura sarà in place.

Date: 1 Dicembre 2008.

ESERCIZIO DI ASD DEL 1 DICEMBRE 2008

Fusione di Liste

Algoritmo per la Fusione di Liste. Siano L_1 ed L_2 due liste ordinate in cui ogni elemento x ha un campo $key[x]$ contenente una chiave intera ed un campo $next[x]$ che punta al successore di x. Consideriamo una lista ausiliaria M che all'inizio sarà vuota ed alla fine conterrà il risultato. Useremo anche due puntatori, $temp_1$ e $temp_2$ per scandire le due liste L_1 ed L_2. Ad ogni iterazione dovremo copiare in fondo alla lista M il più piccolo tra l'elemento puntato da $temp_1$ e quello puntato da $temp_2$, e spostare avanti il puntatore relativo all'elemento copiato. Per copiare in fondo alla lista M è opportuno che M abbia sia il campo $head[M]$ che punta all'inizio di M, che il campo $tail[M]$ che punta alla fine di M.

Algorithm 1 FONDI(L_1, L_2)

```
M ← CREA_LISTA_VUOTA()
temp1 ← head[L1]
temp2 ← head[L2]
while temp1 ≠ NIL and temp2 ≠ NIL do
   if key[temp1] < key[temp2] then
      INSERISCI_IN_FONDO(M, key[temp1])
      temp1 ← next[temp1]
   else
      INSERISCI_IN_FONDO(M, key[temp2])
      temp2 ← next[temp2]
   end if
end while
if temp1 = NIL then
   while temp2 ≠ NIL do
      INSERISCI_IN_FONDO(M, key[temp2])
      temp2 ← next[temp2]
   end while
end if
if temp2 = NIL then
   while temp1 ≠ NIL do
      INSERISCI_IN_FONDO(M, key[temp1])
      temp1 ← next[temp1]
   end while
end if
return M
```

Date: 1 Dicembre 2008.

Algorithm 2 CREA_LISTA_VUOTA()

```
new(N)
head[N] ← NIL
tail[N] ← NIL
return N
```

Algorithm 3 INSERISCI_IN_FONDO(N, i)

```
new(x)
key[x] ← i
next[x] ← NIL
next[tail[N]] ← x
tail[N] ← x
```

Correttezza.

Invariante 1. *All'inizio della i-esima iterazione del primo ciclo while la lista M contiene $i-1$ elementi, è ordinata, ed inoltre $temp_1$ e $temp_2$ puntano a due liste che contengono solo elementi maggiori o uguali rispetto agli elementi che compaiono in M.*

Dimostrazione. Procediamo per induzione su i.
BASE. $i = 1$. All'inizio della prima iterazione del ciclo while la lista M è vuota, quindi vale la tesi.
PASSO.

HpInd) all'inizio della j-esima iterazione del primo ciclo while lista M contiene $j-1$ elementi, è ordinata, ed inoltre $temp_1$ e $temp_2$ puntano a due liste che contengono solo elementi maggiori o uguali rispetto agli elementi che compaiono in M.

Ts) all'inizio della $j+1$-esima iterazione del primo ciclo while lista M contiene j elementi, è ordinata, ed inoltre $temp_1$ e $temp_2$ puntano a due liste che contengono solo elementi maggiori o uguali rispetto agli elementi che compaiono in M.

Dobbiamo esaminare ciò che accade durante la j-esima iterazione del ciclo while. Se $key[temp_1]$ è minore di $key[temp_2]$, allora $key[temp_1]$ viene copiata in fondo ad M e $temp_1$ viene spostato in avanti. Per ipotesi induttiva $key[temp_1]$ è maggiore o uguale di tutti gli elementi di M, ed M è ordinata, quindi alla fine dell'iterazione M continua ad essere ordinata e contiene j elementi. Siccome L_1 ed L_2 erano ordinate, $temp_1$ e $temp_2$ puntano ad elementi maggiori o uguali degli elementi di M. Quindi vale la tesi. Analogamente si dimostra che vale la tesi nel caso $key[temp_2] \leq key[temp_1]$. □

Invariante 2. *All'inizio della i-esima iterazione del secondo ciclo while la lista M è ordinata, ed inoltre $temp_1$ e $temp_2$ puntano a due liste che contengono solo elementi maggiori o uguali rispetto agli elementi che compaiono in M.*

Invariante 3. *All'inizio della i-esima iterazione del terzo ciclo while la lista M è ordinata, ed inoltre $temp_1$ e $temp_2$ puntano a due liste che contengono solo elementi maggiori o uguali rispetto agli elementi che compaiono in M.*

La dimostrazione di questi due invarianti è analoga a quella del primo invariante.

Theorem 1. *Se L_1 ed L_2 sono due liste ordinate, allora* FONDI(L_1, L_2) *termina sempre restituendo una lista ordinata che contiene tutti e soli gli elementi di L_1 ed L_2.*

Dimostrazione. FONDI(L_1, L_2) termina sempre in quanto ad ogni iterazione viene copiato in M un elemento di L_1 o un elemento di L_2 e l'elemento copiato non verrà più considerato.

La lista M contiene tutti e soli gli elementi di L_1 ed L_2 in quanto i puntatori $temp_1$ e $temp_2$ vengono spostati in avanti solo dopo che un elemento è stato copiato.

Infine la lista M è ordinata come segue dall'Invariante 3. □

Complessità. Dato che ogni elemento di L_1 ed ogni elemento di L_2 viene esaminato esattamente una volta, la procedura ha complessità $\Theta(|L_1| + |L_2|)$, dove $|L_1|$ è la lunghezza di L_1 e $|L_2|$ è la lunghezza di L_2.

ESERCIZIO DI ASD DEL 15 DICEMBRE 2008

Da Vettore Ordinato a Binary Search Tree

Sia A un vettore di interi ordinato di lunghezza n. Si consideri il problema di creare un BST T contenente gli elementi di A ed avente altezza $O(\log n)$.

1 Si scriva lo pseudocodice di una procedura per risolvere tale problema.

2 Si dimostri la correttezza della procedura proposta.

3 Si determini la complessità della procedura proposta.

Date: 15 Dicembre 2008.

ESERCIZIO DI ASD DEL 15 DICEMBRE 2008

Da Vettore Ordinato a Binary Search Tree

Sia A un vettore di interi ordinato di lunghezza n. Si consideri il problema di creare un BST T contenente gli elementi di A ed avente altezza $O(\log n)$.

1 Si scriva lo pseudocodice di una procedura per risolvere tale problema.
Suggerimento: Mettere l'elemento centrale del vettore nella radice e poi procedere ricorsivamente sul sottovettore di sinistra per costruire il sottoalbero sinistro e sul sottovettore di destra per costruire il sottoalbero destro.

2 Si dimostri la correttezza della procedura proposta.
Suggerimento: Procedere per induzione sulla lunghezza del vettore.

3 Si determini la complessità della procedura proposta.
Suggerimento: Determinare e risolvere l'equazione ricorsiva di complessità.

Date: 15 Dicembre 2008.

ESERCIZIO DI ASD DEL 15 DICEMBRE 2008

Da Vettore Ordinato a Binary Search Tree

Algoritmo. Per costruire un albero il più possibile bilanciato, utilizziamo l'elemento centrale del vettore come radice dell'albero e procediamo in maniera ricorsiva a sinistra e destra. Come già visto in altre procedure ricorsive su vettori, dobbiamo passare alla procedura gli indici che delimitano la porzione di vettore su cui stiamo lavorando. Inoltre, per poter sistemare i puntatori *parent* passeremo alla procedura anche il nodo che ha "generato" la chiamata ricorsiva. La procedura ritornerà sempre come risultato il nuovo nodo creato.

Algorithm 1 $\text{ARRAY\_TO\_BST}(A)$

$x \leftarrow \text{REC\_ARRAY\_TO\_BST}(A, 1, length[A], NIL)$
$root[T] \leftarrow x$
return T

Algorithm 2 $\text{REC\_ARRAY\_TO\_BST}(A, i, j, y)$

if $i \leq j$ **then**
    $k \leftarrow (i + j)/2$
    $x \leftarrow new\_node()$
    $key[x] \leftarrow A[k]$
    $parent[x] \leftarrow y$
    $left[x] \leftarrow \text{REC\_ARRAY\_TO\_BST}(A, i, k - 1, x)$
    $right[x] \leftarrow \text{REC\_ARRAY\_TO\_BST}(A, k + 1, j, x)$
else
    $x \leftarrow NIL$
end if
return x

Correttezza. Dimostriamo prima di tutto che viene costruito un binary search tree.

Lemma 1. *Se il vettore A è ordinato, la procedura* $\text{REC\_ARRAY\_TO\_BST}(A, i, j, y)$ *termina sempre restituendo un nodo x che è radice di un binary search tree contenente tutti e soli gli elementi di $A[i..j]$.*

Dimostrazione. Procediamo per induzione sul numero di elementi contenuti in $A[i..j]$, ovvero per induzione su $n = j - i + 1$.
BASE. $n = 0$. In questo caso deve essere $j < i$. Quindi ad x viene assegnato valore NIL, che è radice di un BST che non contiene elementi.
PASSO.

Date: 15 Dicembre 2008.

HpInd) Se $n < m$ ed A è ordinato, allora la procedura REC_ARRAY_TO_BST(A, i, j, y) termina sempre restituendo un nodo x che è radice di un binary search tree contenente tutti e soli gli elementi di $A[i..j]$.

Ts) Se $n = m$ ed A è ordinato, allora la procedura REC_ARRAY_TO_BST(A, i, j, y) termina sempre restituendo un nodo x che è radice di un binary search tree contenente tutti e soli gli elementi di $A[i..j]$.

Abbiamo che $(k-1) - i + 1 < n$ e $j - (k+1) + 1 < n$, quindi per ipotesi induttiva REC_ARRAY_TO_BST$(A, i, k-1, x)$ e REC_ARRAY_TO_BST$(A, i, k-1, x)$ terminano restituendo due nodi z e w radici di BST contenenti rispettivamente gli elementi di $A[i..k-1]$ e $A[k+1..i]$. Ad x viene assegnata chiave $A[k]$, che è maggiore di tutti i valori contenuti nel BST radicato in z e minore di tutti i valori nel BST radicato in w, in quanto A è ordinato. Quindi vale la tesi. □

Dobbiamo anche dimostrare che l'altezza dell'albero costruito è logaritmica.

Lemma 2. REC_ARRAY_TO_BST(A, i, j, y) *restituisce un nodo x che è radice di un albero di altezza $\theta(\log(j - i + 1))$.*

Dimostrazione. Sia $h(n)$ l'altezza dell'albero costruito a partire da $n = j - i + 1$ elementi. Dobbiamo dimostrare che esiste un $\bar{n} \geq 0$ ed una costante $c > 0$ tali che per ogni $n \geq \bar{n}$ vale che $h(n) \leq c * \log n$. Procediamo per induzione su n.

Per il caso base dobbiamo partire con $\bar{n} = 2$, altrimenti non riusciamo a dimostrare la tesi. Abbiamo che nel caso di un vettore con due elementi viene costruito un albero di altezza 1 (la radice ed un figlio). Quindi $1 = h(2) \leq c * \log 2 = c * 1$ è vera a patto che valga $c \geq 1$.

Per il passo induttivo abbiamo $h(n) = max\{h(m), h(n-m-1)\} + 1$, dove $h(m)$ è l'altezza del sottoalbero sinistro e $h(n - m - 1)$ è l'altezza del sottoalbero destro. Sappiamo che $m \leq n/2$ e $n - m - 1 \leq n/2$, quindi per ipotesi induttiva abbiamo che $h(m) \leq c * \log(n/2)$ e $h(n - m - 1) \leq c * \log(n/2)$. Da questo otteniamo $h(n) \leq c * (\log(n/2)) + 1 = c * (\log n) - c + 1$ e quest'ultimo è minore o uguale a $c * \log n$ se vale $c \geq 1$.

Quindi vale la tesi con $c \geq 1$.

Dovremmo anche dimostrare che $h(n) \geq d * \log n$ per un'opportuna costante d ed n sufficientemente grande, ma questo è sempre vero per un albero binario contenente n nodi. □

Complessità. Ponendo $n = j - i + 1$ l'equazione ricorsiva di complessità della procedura REC_ARRAY_TO_BST(A, i, j, y) è:

$$T(n) = \begin{cases} \Theta(1) & n = 0 \\ T(m) + T(n-m-1) + \Theta(1) & n > 0 \end{cases}$$

Questa equazione ricorsiva è la stessa che abbiamo già visto a lezione per le visite degli alberi. Si intuisce che la sua complessità è $\Theta(n)$, in quanto vengono eseguite operazioni aventi costo $\Theta(1)$ per ogni elemento del vettore. Si dimostra formalmente il risultato applicando il metodo per sostituzione.

ESERCIZIO DI ASD DEL 16 MARZO 2009

Binary Search Tree Colorabile

Sia T un BST (avente anche le foglie fittizie come nei RBT). Sia C un colore (Rosso oppure Nero) ed $h \geq 0$ un intero. Si consideri il problema di determinare se T può essere un RBT con radice di colore C ed altezza nera h.

1 Si scriva lo pseudocodice di una procedura per risolvere tale problema.

2 Si dimostri la correttezza della procedura proposta.

3 Si determini la complessità della procedura proposta.

Date: 16 Marzo 2009.

ESERCIZIO DI ASD DEL 16 MARZO 2009

Binary Search Tree Colorabile

Sia T un BST (avente anche le foglie fittizie come nei RBT). Sia C un colore (Rosso oppure Nero) ed $h \geq 0$ un intero. Si consideri il problema di determinare se T può essere un RBT con radice di colore C ed altezza nera h.

1 Si scriva lo pseudocodice di una procedura per risolvere tale problema.
Suggerimento: Implementare una procedura ricorsiva. Ricordare che occorre solo controllare se è colorabile, ma non occorre colorare effettivamente l'albero.

2 Si dimostri la correttezza della procedura proposta.
Suggerimento: Procedere per induzione sul numero di chiamate ricorsive o sull'altezza dell'albero.

3 Si determini la complessità della procedura proposta.
Suggerimento: Scrivere e risolvere l'equazione ricorsiva di complessità nel caso peggiore in funzione dell'altezza.

Date: 16 Marzo 2009.

ESERCIZIO DI ASD DEL 16 MARZO 2009

Binary Search Tree Colorabile

Algoritmo Ricorsivo. Procedendo ricorsivamente abbiamo che:

- Se l'albero è *vuoto*, allora contiene un'unica foglia fittizia NIL, quindi il colore deve essere NERO e l'altezza nera deve essere 0.
- Se l'albero non è vuoto ed il colore C è ROSSO, allora i due figli devono essere NERI ed avere altezza nera di uno inferiore.
- Se l'albero non è vuoto ed il colore C è NERO, allora ognuno dei due figli può essere o NERO con altezza nera inferiore di uno o ROSSO con la stessa altezza nera.

Algorithm 1 CheckRBTree(T, C, h)

return NodeCheck$(root[T], C, h)$

Algorithm 2 NodeCheck(x, C, h)

if $x = NIL$ **then**
    if $C = RED$ or $h \neq 0$ **then**
        return FALSE
    else
        return TRUE
    end if
else
    if $C = RED$ **then**
        return NodeCheck$(left[x], BLACK, h-1)$ and
                NodeCheck$(right[x], BLACK, h-1)$
    else
        return (NodeCheck$(left[x], BLACK, h-1)$ or
                NodeCheck$(left[x], RED, h)$) and
                (NodeCheck$(right[x], BLACK, h-1)$ or
                NodeCheck$(right[x], RED, h)$)
    end if
end if

Date: 16 Marzo 2009.

Correttezza.

Lemma 1. NodeCheck(x, C, h) *termina restituendo TRUE se e soltanto se il sottoalbero T_x radicato in x può essere un RB-tree con x di colore C ed altezza nera h.*

Dimostrazione. Procediamo per induzione sull'altezza di T_x.
BASE. T_x ha altezza 0.

x è NIL. Quindi x può assumere solo colore NERO e l'altezza nera di T_x è 0. Effettivamente in questo caso la procedura ritorna TRUE se e soltanto se C è NERO ed h è 0 (linee 1–6).
PASSO. T_x ha altezza n.

HpInd) Vale la tesi per tutti gli alberi di altezza minore di n.

Ts) Vale la tesi per T_x.

Se C è ROSSO, allora T_x puó essere un RB-Tree di altezza nera h con x ROSSO se e soltanto se sia $left[x]$ che $right[x]$ possono essere colorati di NERO e diventare radici di due RB-tree di altezza nera $h-1$. Nel caso C sia ROSSO le linee 8–9 effettuano questo controllo e le due chiamate ricorsive terminano correttamente per ipotesi induttiva. t Se C è NERO, allora T_x può essere un RB-Tree di altezza nera h con x NERO se e soltanto se valgono le due condizioni seguenti:

1. $left[x]$ può essere NERO e radice di un RB-Tree di altezza nera $h-1$ oppure $left[x]$ può essere ROSSO e radice di un RB-Tree di altezza nera h;
2. $right[x]$ può essere NERO e radice di un RB-Tree di altezza nera $h-1$ oppure $right[x]$ può essere ROSSO e radice di un RB-Tree di altezza nera h.

Le linee di codice 10–11 effettuano questi controlli e le chiamate ricorsive terminano correttamente per ipotesi induttiva. □

Theorem 1. CheckRBTree(T, C, h) *termina restituendo TRUE se e soltanto se l'albero T può essere un RB-tree con radice di colore C ed altezza nera h.*

Dimostrazione. La tesi segue immediatamente dal lemma. □

Complessità. Nel caso peggiore partendo da un albero di altezza k vengono effettuate 4 chiamate ricorsive su alberi di altezza $k-1$ più alcuni controlli che hanno costo $\Theta(1)$.

$$T(k) \quad \leq \quad \begin{cases} \Theta(1) & \text{if } k = 0 \\ 4 * T(k-1) + \Theta(1) & \text{if } k > 0 \end{cases}$$

Risolvendo l'equazione si ottiene $T(k) = O(4^k)$.

ESERCIZIO DI ASD DEL 30 MARZO 2009

B-Tree Join

Siano T_1 e T_2 due B-tree di grado $t \geq 2$ e sia k una chiave intera tale che tutte le chiavi di T_1 sono minori di k e tutte le chiavi di T_2 sono maggiori di k. Si consideri il problema di costruire un B-tree T di grado t che contenga tutte le chiavi di T_1, tutte le chiavi di T_2 e la chiave k.

1 Si scriva lo pseudocodice di una procedura per risolvere tale problema.

2 Si dimostri la correttezza della procedura proposta.

3 Si determini la complessità della procedura proposta.

Date: 30 Marzo 2009.

ESERCIZIO DI ASD DEL 30 MARZO 2009

B-Tree Join

Siano T_1 e T_2 due B-tree di grado $t \geq 2$ e sia k una chiave intera tale che tutte le chiavi di T_1 sono minori di k e tutte le chiavi di T_2 sono maggiori di k. Si consideri il problema di costruire un B-tree T di grado t che contenga tutte le chiavi di T_1, tutte le chiavi di T_2 e la chiave k.

1 Si scriva lo pseudocodice di una procedura per risolvere tale problema.
Suggerimento: Procedere come nel caso di unione di due RB-tree vista a lezione, facendo attenzione ai nodi pieni.
2 Si dimostri la correttezza della procedura proposta.
Suggerimento: Dimostrare che le chiavi sono posizionate correttamente e che ogni nodo contiene un numero ammissibile di chiavi.
3 Si determini la complessità della procedura proposta.
Suggerimento: La complessità sarà simile a quella già calcolata a lezione nel caso dei RB-tree.

Date: 30 Marzo 2009.

ESERCIZIO DI ASD DEL 30 MARZO 2009

B-Tree Join

Algoritmo. L'idea di base è la seguente:

- dobbiamo assicurarci di trovarci sempre su un nodo non pieno, quindi come prima cosa splittiamo le radici di T_1 e T_2 se sono piene. Ogni volta che scenderemo su un nuovo nodo lo splitteremo se è pieno;
- se T_1 e T_2 hanno la stessa altezza, allora creiamo una nuova radice che conterrà solo la chiave k ed avrà la radice di T_1 come primo figlio e la radice di T_2 come secondo figlio;
- se T_1 ha altezza maggiore di T_2, allora scendiamo in T_1 verso destra fino ad arrivare ad un nodo che ha altezza di uno superiore rispetto all'altezza di T_2. Aggiungiamo a questo nodo la chiave k come chiave più a destra e come nuovo figlio la radice di T_2;
- se T_2 ha altezza maggiore di T_1, allora procediamo analogamente al caso precedente, ma scendendo verso sinistra.

Algorithm 1 BTreeJoin(T_1, T_2, k, t)

$x_1 \leftarrow root[T_1]$
$x_2 \leftarrow root[T_2]$
if $n[x_1] = 2t - 1$ **then**
    BTreeRootSplit(T_1, x_1, t)
    $x_1 \leftarrow root[T_1]$
end if
if $n[x_2] = 2t - 1$ **then**
    BTreeRootSplit(T_2, x_2, t)
    $x_2 \leftarrow root[T_2]$
end if
$h_1 \leftarrow$ Heigh(x_1)
$h_2 \leftarrow$ Heigh(x_2)
if $h_1 = h_2$ **then**
    return BTreeJoinEqual(T_1, T_2, k, t, x_1, x_2)
else
    if $h_1 > h_2$ **then**
        return BTreeJoinRight(T_1, T_2, k, t, x_1, x_2)
    else
        return BTreeJoinLeftt(T_1, T_2, k, t, x_1, x_2)
    end if
end if

Date: 30 Marzo 2009.

Algorithm 2 BTreeJoinEqual(T_1, T_2, k, t, x_1, x_2)

$z \leftarrow$AllocateNewNode(t)
$n[z] \leftarrow 1$
$key_1[z] \leftarrow k$
$c_1[z] \leftarrow x_1$
$c_2[z] \leftarrow x_2$
if $n[x_1] = 1$ **then**
    BTreeMerge($z, x_1, 1$)
end if
if $n[x_2] = 1$ **then**
    BTreeMerge($z, x_2, n[z] + 1$)
end if
DiskWrite(z)
$root[T_1] \leftarrow z$
return T_1

Algorithm 3 BTreeJoinRight(T_1, T_2, k, t, x_1, x_2)

while $h_1 > h_2 + 1$ **do**
    $y_1 \leftarrow$ DiskRead$c_{n[x_1]+1}[x_1]$
    if $n[y_1] = 2t - 1$ **then**
        BTreeSplit($x_1, y_1, n[x_1] + 1$)
        $y_1 \leftarrow$ DiskRead$c_{n[x_1]+1}[x_1]$
    end if
    $x_1 \leftarrow y_1$
    $h_1 \leftarrow h_1 - 1$
end while
$n[x_1] \leftarrow n[x_1] + 1$
$key_{n[x_1]}[x_1] \leftarrow k$
$c_{n[x_1]+1}[x_1] \leftarrow x_2$
DiskWrite(x_1)
return T_1

Correttezza.

Theorem 1. BTreeJoin(T_1, T_2, k, t) *termina sempre restituendo un B-tree di grado t che contiene tutte le chiavi di T_1, tutte le chiavi di T_2 e la chiave k.*

Dimostrazione. L'algoritmo termina sempre perchè gli unici due cicli while che compaiono nelle procedure BTreeJoinLeft e BTreeJoinRight terminano in quando h_1 ed h_2 vengono opportunamente decrementati.

Sicuramente al termine l'albero restituito contiene tutte le chiavi richieste in quanto non vengono cancellate chiavi, viene aggiunta la chiave k e tutte le chiavi di T_2 o tutte le chiavi di T_1 (a seconda dei casi).

Dobbiamo dimostrare che l'albero restituito è un B-tree di grado t.

Se $h_1 = h_2$, allora l'albero restituito è un B-tree perchè:

- ha una radice contenente solamente la chiave k ed avente come figli x_1 ed x_2;
- x_1 è radice di un B-tree di grado t contentente solo chiavi minori di k;

Algorithm 4 BTreeJoinLeft$(T_1, T_2, k, t, x_1, x_2)$

while $h_2 > h_1 + 1$ **do**
  $y_2 \leftarrow$ DiskRead$c_1[x_2]$
  if $n[y_1] = 2t - 1$ **then**
    BTreeSplit$(x_2, y_2, 1)$
    $y_2 \leftarrow$ DiskRead$c_1[x_2]$
  end if
  $x_2 \leftarrow y_2$
  $h_2 \leftarrow h_2 - 1$
end while
$n[x_2] \leftarrow n[x_2] + 1$
for $i \leftarrow n[x_2]$ downto 2 **do**
  $key_i[x_2] \leftarrow key_{i-1}[x_2]$
end for
for $i \leftarrow n[x_2] + 1$ downto 2 **do**
  $c_i[x_2] \leftarrow c_{i-1}[x_2]$
end for
$key_1[x_2] \leftarrow k$
$c_1[x_2] \leftarrow x_1$
DiskWrite(x_2)
return T_2

Algorithm 5 NodeCheck(x, C, h)

if $x = NIL$ **then**
  if $C = RED$ or $h \neq 0$ **then**
    return FALSE
  else
    return TRUE
  end if
else
  if $C = RED$ **then**
    return NodeCheck$(left[x], BLACK, h-1)$ and
        NodeCheck$(right[x], BLACK, h-1)$
  else
    return (NodeCheck$(left[x], BLACK, h-1)$ or
        NodeCheck$(left[x], RED, h)$) and
        (NodeCheck$(right[x], BLACK, h-1)$ or
        NodeCheck$(right[x], RED, h)$)
  end if
end if

- x_2 è radice di un B-tree di grado t contentente solo chiavi maggiori di k;
- se x_1 e/o x_2 hanno solo una chiave vengono fusi con z.

Se $h_1 > h_2$, allora l'albero restituito è un B-tree perchè:

- al termine del ciclo while il nodo x_1 è un nodo non pieno di altezza $h_2 + 1$;
- il nodo x_1 si trova a destra nell'albero T_1;

- al nodo x_1 viene aggiunta a destra la chiave k, che è maggiore di tutte le altre di T_1;
- al nodo x_1 viene aggiunto a destra il figlio x_2 che contiene tutte chiavi maggiori di quelle di T_1 e di k e che ha altezza h_2.

Il caso $h1 < h_2$ è analogo. □

Complessità. Per determinare h_1 ed h_2 occorre scendere dalla radice ad una foglia in T_1 e T_2, rispettivamente. Tutte le altre istruzioni richiedono un numero di operazioni limitato superiormente dall'altezza dei due alberi. Quindi abbiamo complessità $\Theta(h(T_1) + h(T_2))$ in termini di operazioni di lettura/scrittura da disco ed una complessità $O(t * (h(T_1) + h(T_2)))$ in termini di operazioni di CPU.

ESERCIZIO DI ASD DEL 6 APRILE 2009

Massimo di Insiemi Disgiunti

Si consideri un'estensione della struttura dati "Disjoint-Sets" in cui oltre alle operazioni Make, Union, Find, vengono introdotte le operazioni:

- Find-Max(x). Restituisce l'elemento massimo dell'insieme in cui sta x
- Set(x). Stampa tutte le chiavi degli elementi che si trovano nello stesso insieme in cui si trova x.

1 Si proponga una struttura dati per supportare le operazioni sopra descritte.

2 Si calcoli la complessità di m operazioni di Make, Union, Find, Find-Max, Set, di cui n Make e p Set.

Date: 6 Aprile 2009.

ESERCIZIO DI ASD DEL 6 APRILE 2009

Massimo di Insiemi Disgiunti

Si consideri un'estensione della struttura dati "Disjoint-Sets" in cui oltre alle operazioni Make, Union, Find, vengono introdotte le operazioni:

- Find-Max(x). Restituisce l'elemento massimo dell'insieme in cui sta x
- Set(x). Stampa tutte le chiavi degli elementi che si trovano nello stesso insieme in cui si trova x.

1 Si proponga una struttura dati per supportare le operazioni sopra descritte.
Suggerimento: Non è conveniente implementare l'operazione Set(x) utilizzando alberi

2 Si calcoli la complessità di m operazioni di Make, Union, Find, Find-Max, Set, di cui n Make e p Set.
Suggerimento: Per le operazioni di Make, Union e Find si possono utilizzare i conti già visti a lezione. Find-Max può essere implementata in modo che ogni Find-Max abbia costo minimo possibile. Lo stesso discorso vale per l'operazione Set.

Date: 6 Aprile 2009.

ESERCIZIO DI ASD DEL 6 APRILE 2009

Massimo di Insiemi Disgiunti

Struttura Dati e Operazioni. Utilizziamo per implementare le operazioni richieste le liste concatenate già usate a lezione per implementare Weighted-Union. In particolare ogni elemento x avrà i seguenti campi:

- $key[x]$. La chiave dell'elemento x;
- $rap[x]$. Il puntatore al rappresentante della lista contenente x;
- $next[x]$. Il puntatore al successore di x nella lista;
- $last[x]$. Il puntatore all'ultimo elemento della lista contenente x. Questo campo viene mantenuto aggiornato solo nel rappresentante.
- $length[x]$. La lunghezza della lista contenente x. Questo campo viene mantenuto aggiornato solo nel rappresentante.

Per implementare efficientemente la richiesta del massimo facciamo in modo che il massimo si trovi sempre nel rappresentante. Per garantire questa proprietà è sufficiente che nell'unione si scambino le chiavi dei due rappresentanti in modo da mettere per prima quella maggiore.

Algorithm 1 MAKE(x)

$rap[x] \leftarrow x$
$next[x] \leftarrow NIL$
$last[x] \leftarrow x$
$length[x] \leftarrow 1$

Algorithm 2 FIND(x)

return $rap[x]$

Algorithm 3 FIND-MAX(x)

return $rap[x]$

Algorithm 4 UNION(x, y)

if $rap[x] \neq rap[y]$ **then**
    LINK($rap[x], rap[y]$)
end if

Date: 6 Aprile 2009.

Algorithm 5 Link(z, w)

```
if length[z] > length[w] then
    if key[w] > key[z] then
        ScambiaKey(z, w)
    end if
    length[z] → length[z] + length[w]
    u ← last[z]
    last[z] ← last[w]
    next[u] ← w
    while u ≠ NIL do
        rap[u] ← z
        u ← next[u]
    end while
else
    Link(w, z)
end if
```

Algorithm 6 ScambiaKey(z, w)

```
k ← key[z]
key[z] ← key[w]
key[w] ← k
```

Algorithm 7 Set(x)

```
y ← rap[x]
while y ≠ NIL do
    Print(key[y])
    y ← next[y]
end while
```

Correttezza. Mentre è immediato vedere che Set è corretta, occorre dimostrare formalmente che Find-Max è corretta. Questo equivale a dire che occorre dimostrare formalmente che il massimo è sempre il primo elemento della lista. Mostriamo una bozza di dimostrazione di correttezza.

Theorem 1. *Con le operazioni sopra descritte il massimo della lista contenente x è il rappresentante della lista contenente x.*

Dimostrazione. Procediamo per induzione sul numero di operazioni di unione che sono state applicate per ottenere la lista contenente x.

BASE. Non sono state effettuate unioni. In questo caso la lista contenente x contiene solo x che è sia il massimo che il rappresentante.

PASSO. Supponiamo per ipotesi induttiva che la tesi sia vera se la lista è stata ottenuta con al più $n-1$ unioni e dimostriamo che vale la tesi anche se la lista è stata ottenuta con n unioni.

Se la lista contentente x è stata ottenuta con n unioni, allora l'ultima unione ha unito due liste L_1 ed L_2 che erano state ottenute con al più $n-1$ unioni ciascuna. Quindi per ipotesi induttiva L_1 ha come rappresentante l'elemento y_1 con chiave maggiore ed L_2 ha come rappresentante l'elemento y_2 con chiave maggiore. Quindi la chiave maggiore nella lista ottenuta dopo n unioni è il più grande tra la chiave di y_1 e quella di y_2. Quando si uniscono L_1 ed L_2 viene messa per prima la lista con più elementi, ma se questa non ha il rappresentante con chiave maggiore, allora le chiavi dei due rappresentanti vengono scambiate. □

Complessità.

- Un'operazione di MAKE costa $\Theta(1)$;
- Un'operazione di FIND costa $\Theta(1)$;
- Un'operazione di FIND-MAX costa $\Theta(1)$;
- Un'operazione di UNION costa $\Theta(min(length(L_1), length(L_2))$, dove L_1 ed L_2 sono le due liste che vengono unite;
- Un'operazione di SET costa $\Theta(length(L))$, dove L è la lista che viene stampata.

Se consideriamo m operazioni di cui n MAKE e p SET, abbiamo che:

- m operazioni di MAKE, UNION, FIND, e FIND-MAX costano $O(m+n\log n)$ (la dimostrazione è la stessa vista a lezione nel caso di Weighted-Union);
- p operazioni di SET costano $O(p * n)$, perchè nel caso peggiore possono essere effettuate tutte su un unico insieme contenente tutti gli elementi.

Quindi complessivamente abbiamo costo $O(m + n \log n + pn)$.

ESERCIZIO DI ASD DEL 27 APRILE 2009

Diametro

Sia $G = (V, E)$ un grafo non orientato, connesso, aciclico. Il diametro di G, $d(G)$, è la massima distanza tra due nodi di G, ovvero:

$$d(G) = max\{\delta(u, v) \mid u, v \in V\}$$

dove $\delta(u, v)$, la distanza tra u e v, è la lunghezza del cammino più corto che porta da u a v.

1 Si descriva tramite pseudocodice un algoritmo che dato un grafo G non orientato, connesso ed aciclico, calcola $d(G)$.
2 Si calcoli la complessità dell'algoritmo proposto.
3 Se ne dimostri la correttezza.

Date: 27 Aprile 2009.

ESERCIZIO DI ASD DEL 27 APRILE 2009

Diametro

Sia $G = (V, E)$ un grafo non orientato, connesso, aciclico. Il diametro di G, $d(G)$, è la massima distanza tra due nodi di G, ovvero:

$$d(G) = max\{\delta(u, v) \mid u, v \in V\}$$

dove $\delta(u, v)$, la distanza tra u e v, è la lunghezza del cammino più corto che porta da u a v.

1 Si descriva tramite pseudocodice un algoritmo che dato un grafo G non orientato, connesso ed aciclico, calcola $d(G)$.
Suggerimento: È semplice proporre una procedura naïve che sfrutta $|V|$ volte la visita BFS. Si può migliorare tale soluzione riducendosi a 2 chiamate a BFS. La prima chiamata deve servire per determinare uno dei due nodi che determineranno il diametro.

2 Si calcoli la complessità dell'algoritmo proposto.
Suggerimento: La procedura più efficiente che si può ottenere opera nello stesso tempo di una visita BFS.

3 Se ne dimostri la correttezza.
Suggerimento: La dimostrazione di correttezza della procedura naïve è immediata. Nella dimostrazione di correttezza della procedura più efficiente occorre dimostrare che comunque si scelgano due nodi a e b vale che $\delta(a, b)$ è minore o uguale del valore restituito dall'algoritmo.

Date: 27 Aprile 2009.

ESERCIZIO DI ASD DEL 27 APRILE 2009

Diametro

Algoritmi. Ricordiamo che un grafo non orientato, aciclico e connesso è un albero. Un albero può essere pensato come albero radicato una volta che si sia fissato un nodo come radice. Ad esempio il grafo Gr in Figura 1 può essere pensato come albero radicato nel nodo a.

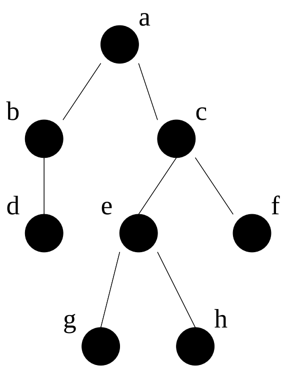

Figura 1. Grafo Gr.

A seconda del nodo che utilizziamo come radice l'albero radicato avrà altezza diversa. L'altezza di un albero radicato non è altro che la massima distanza tra la radice ed una foglia. Il grafo Gr se pensato come albero radicato in a ha altezza 3, mentre se pensato come albero radicato in d ha altezza 5. Se richiamiamo BFS(Gr, a) otteniamo che il nodo a distanza massima da a si trova a distanza 3, mentre se richiamiamo BFS(Gr, d) otteniamo che il nodo a distanza massima da d è a distanza 5.

Il diametro di un grafo G si può quindi trovare richiamando BFS(G, x) per ogni x nodo di G e memorizzando di volta in volta la distanza massima calcolata. Nel grafo Gr otteniamo che il diametro è 5.

Quindi come prima cosa modifichiamo la procedura BFS in modo che al termine ritorni la massima distanza calcolata (Algorithm 1). Si noti che non è necessario usare i colori e neanche costruire l'albero di visita.

A questo punto è sufficiente richiamare BFS_diam una volta per ogni nodo e determinare la distanza massima tra tutte le distanze massime restituite (Algorithm 2).

Possiamo utilizzare BFS per ottenere una procedura più efficiente? Dovremmo evitare di richiamare BFS su ogni nodo del grafo e richiamarla sul "pochi" nodi. Ma come? Bisognerebbe riuscire ad indovinare una radice che massimizza l'altezza. Sicuramente non basta utilizzare BFS una sola volta, perchè la prima volta che

Date: 27 Aprile 2009.

Algorithm 1 BFS_DIAM$(G = (N, E), x)$

```
max ← −1
for each v ∈ N do
  d[v] ← ∞
end for
Q ← ∅
d[x] ← 0
ENQUEUE(Q, x)
while Q ≠ ∅ do
  v ←HEAD(Q)
  for each u ∈ Adj[v] do
    if d[u] = ∞ then
      d[u] ← d[v] + 1
      ENQUEUE(Q, u)
    end if
  end for
  max ← d[v]
  DEQUEUE(Q)
end while
return max
```

Algorithm 2 DIAMETRO_NAIVE$(G = (N, E))$

```
diametro ← −1
for each x ∈ N do
  diametro ← MAX(diametro,BFS_DIAM(G, x))
end for
return diametro
```

richiamiamo la procedura non sappiamo niente del nostro grafo. Se osserviamo nuovamente il grafo Gr in Figura 1 notiamo che ci sono 3 nodi che ci consentono di calcolare il diametro: il nodo d, il nodo g, ed il nodo h. Due di questi hanno una caratteristica interessante: g ed h sono due nodi a distanza massima da a. In effetti in generale possiamo procedere in questo modo:

- richiamiamo BFS a partire da un nodo $first$ e determiniamo un nodo *second* a distanza massima da $first$;
- richiamiamo BFS a partire da *second*. La distanza massima da *second* sarà il diametro del grafo.

A questo punto ci serve una variante di BFS che restituisca un nodo a distanza massima (Algorithm 3). L'ultimo nodo ad uscire dalla coda sarà sicuramente un nodo a distanza massima.

Ora non resta che combinare BFS_DIAM e BFS_NODO per ottenere una procedura efficiente per il diametro.

Correttezza. La correttezza della procedura DIAMETRO_NAIVE segue immediatamente dalla definizione di diametro, dalla correttezza della procedura BFS e dal fatto che BFS(G, x) calcola le distanze da x in ordine crescente, quindi l'ultima distanza calcolata sarà la distanza massima di un nodo dal nodo x.

Algorithm 3 BFS_NODO$(G = (N, E), x)$

```
nodo ← NIL
for each v ∈ N do
    d[v] ← ∞
end for
Q ← ∅
d[x] ← 0
ENQUEUE(Q, x)
while Q ≠ ∅ do
    v ←HEAD(Q)
    for each u ∈ Adj[v] do
        if d[u] = ∞ then
            d[u] ← d[v] + 1
            ENQUEUE(Q, u)
        end if
    end for
    nodo ← v
    DEQUEUE(Q)
end while
return nodo
```

Algorithm 4 DIAMETRO$(G = (N, E))$

```
first ←PICK(N) //sceglie un nodo qualsiasi in N
second ←BFS_NODO(G, first)
return BFS_DIAM(G, second))
```

Dimostriamo la correttezza della procedura DIAMETRO. Diamo per buona la correttezza della procedura BFS_DIAM(G, x) che calcola la distanza massima di un nodo da x e della procedura BFS_NODO(G, x) che determina un nodo a distanza massima da x.

Theorem 1. *La procedura* DIAMETRO(G) *termina sempre ed al termine restituisce il diametro di G.*

Dimostrazione. Dalla correttezza di BFS_DIAM$(G, second)$ abbiamo che la procedura restituisce un valore d tale che esiste un nodo z tale che $\delta(second, z) = d$.

Dalla definizione di diametro sappiamo che sicuramente vale $d = \delta(second, z) \leq d(G)$.

Quindi dobbiamo solo dimostrare che $d(G) \leq \delta(second, z) = d$, ovvero che

$$\forall a, b \in N(\delta(a, b) \leq \delta(second, z))$$

Consideriamo l'albero di BFS generato a partire dal nodo $first$. In questo albero troviamo sicuramente l'unico cammino che esiste in G tra a e b. Questo cammino risalirà da a verso la radice $first$ fino ad arrivare ad un nodo t e poi riscenderà verso b. In altri termini t è il più giovane antenato comune tra a e b nell'albero BFS di $first$. Indichiamo con $p(first, t)$ il cammino tra $first$ e t e con $p(first, second)$ il cammino tra $first$ e $second$. Distinguiamo due casi.

Caso 1. $p(first, t)$ e $p(first, second)$ non hanno archi in comune. Si veda la Figura 2. In tal caso abbiamo che

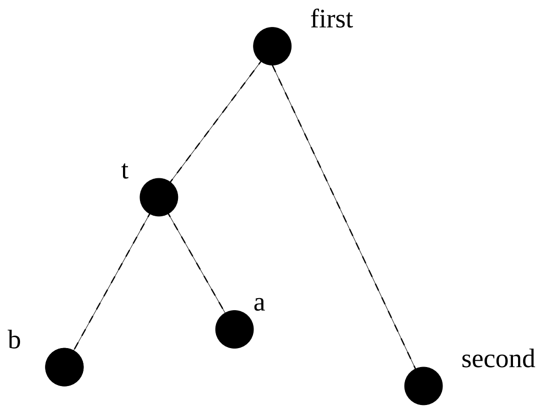

Figura 2. Caso 1.

$$\begin{array}{rcll} \delta(second, b) & = & \delta(second, first) + \delta(first, t) + \delta(t, b) & \\ & \geq & \delta(second, first) + \delta(t, b) & \text{ho tralasciato un addendo} \\ & \geq & \delta(a, first) + \delta(t, b) & second \text{ era a distanza massima da } first \\ & \geq & \delta(a, t) + \delta(t, b) & t \text{ é sul cammino tra } a \text{ e } b \\ & = & \delta(a, b) & \end{array}$$

Quindi abbiamo $\delta(second, z) \geq \delta(second, b)$ in quanto z è a distanza massima da *second* e $\delta(second, b) \geq \delta(a, b)$. Possiamo concludere $\delta(second, z) \geq \delta(a, b)$.

Caso 2. $p(first, t)$ e $p(first, second)$ hanno in comune tutti gli archi che si trovano tra $first$ ed un nodo r. Se *second* si trova nel sottoalbero radicato in t (ovvero se $r = t$), allora potrebbe esserci una parte di cammino tra t e *second* in comune con il cammino tra t ed a o con quello tra t e b, ma non con entrambi, visto che questi due non hanno archi in comune. In tal caso non è restrittivo supporre che ci potrebbero essere archi in comune con il cammino tra t ed a. Si veda la Figura 3. Quindi abbiamo che

$$\begin{array}{rcll} \delta(second, b) & = & \delta(second, r) + \delta(r, b) & \\ & \geq & \delta(a, r) + \delta(r, b) & (*) \\ & = & \delta(a, b) & \end{array}$$

dove in (*) abbiamo potuto rimpiazzare $\delta(second, r)$ con $\delta(a, r)$ in quanto *second* è a distanza massima da $first$ e quindi la distanza tra *second* e r è maggiore di quella tra a ed r.

Quindi abbiamo $\delta(second, z) \geq \delta(second, b)$ in quanto z è a distanza massima da *second* e $\delta(second, b) \geq \delta(a, b)$. Possiamo concludere $\delta(second, z) \geq \delta(a, b)$. $\square$

Complessità. Le procedure BFS_dist e BFS_nodo hanno complessità $\Theta(|V| + |E|) = \Theta(|E|)$, in quanto non sono altro che banali modifiche di BFS ed il grafo in input è connesso.

Di conseguenza la procedura Diametro_Naive ha complessità pari a $\Theta(|V| * |E|)$, in quanto BFS_dist viene richiamata $|V|$ volte, mentre Diametro ha complessità $\Theta(|E|)$, in quanto si richiama una volta BFS_nodo ed una volta BFS_dist.

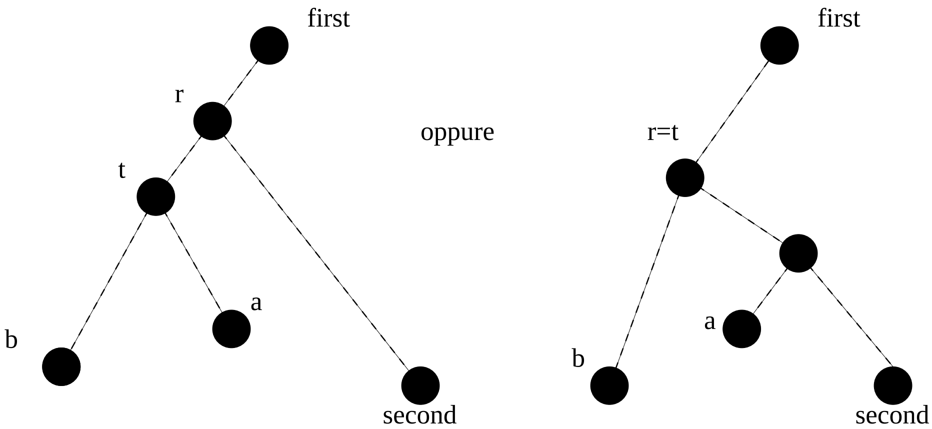

Figura 3. Caso 2.

ESERCIZIO DI ASD DEL 11 MAGGIO 2009

Minimum Spanning Tree

Siano T_1 e T_2 due minimum spanning tree di un grafo $G = (N, E, W)$ e siano L_1 ed L_2 le liste ordinate dei pesi degli archi di T_1 e T_2 rispettivamente.

1 Dimostrare che $L_1 = L_2$.
2 Dimostrare o refutare la seguente affermazione. Se $A \subseteq E$ è tale che la lista ordinata dei pesi di A è uguale ad L_1, allora A è un minimum spanning tree di G.

Date: 11 Maggio 2009.

ESERCIZIO DI ASD DEL 11 MAGGIO 2009

Minimum Spanning Tree

Siano T_1 e T_2 due minimum spanning tree di un grafo $G = (N, E, W)$ e siano L_1 ed L_2 le liste ordinate dei pesi degli archi di T_1 e T_2 rispettivamente.

1 Dimostrare che $L_1 = L_2$.
Suggerimento: Si proceda per assurdo e si ragioni sul minimo indice su cui le due liste differiscono.

2 Dimostrare o refutare la seguente affermazione. Se $A \subseteq E$ è tale che la lista ordinata dei pesi di A è uguale ad L_1, allora A è un minimum spanning tree di G.
Suggerimento: Per dimostrare che l'affermazione è corretta occorre verificare che A soddisfa tutte le condizioni presenti nella definizione di minimum spanning tree. Per refutare l'affermazione occorre invece trovare un controesempio.

Date: 11 Maggio 2009.

ESERCIZIO DI ASD DEL 11 MAGGIO 2009

Minimum Spanning Tree

$L_1 = L_2$**.** Si noti che $L_1 = L_2$ non implica $T_1 = T_2$. Se si considera il grafo $G = (\{a,b,c\}, \{\{a,b\},\{b,c\},\{c,a\}\}, W)$, dove W assegna ad ogni arco peso 1, abbiamo che G ha tre minimum spanning tree distinti, ma questi hanno sempre lista dei pesi pari a $L = [1,1]$.

Dimostrazione. Supponiamo per assurdo che esistano due minimum spanning tree T_1 e T_2 a cui corrispondono le due liste ordinate di pesi L_1 ed L_2 rispettivamente tali che $L_1 \neq L_2$.

Sia i il minimo indice tale che $L_1[i] \neq L_2[i]$. Non è restrittivo supporre che $L_1[i] < L_2[i]$ (se questo non vale basta scambiare il ruolo di L_1 ed L_2). $L_1[i]$ è il peso di un arco $\{x,y\}$ che appartiene a T_1.

Siano $\{x_1,y_1\},\{x_2,y_2\},\ldots,\{x_m,y_m\}$ gli archi che compaiono in T_1 e che hanno peso $L_1[i]$. Sicuramente almeno uno di questi archi non compare in T_2 perché $L_2[i] > L_1[i]$. Sia $\{x_j,y_j\}$ uno degli archi sopra menzionati che manca in T_2.

In T_2 c'è un cammino che connette x_j ed y_j. Indichiamo tale cammino con $path_{T_2}(x_j,y_j)$.

Se in $path_{T_2}(x_j,y_j)$ c'è un arco di peso maggiore di $\{x_j,y_j\}$, allora T_2 non è un minimum spanning tree (potrei togliere l'arco e sostituirlo con $\{x_j,y_j\}$. Quindi siamo giunti ad un assurdo.

Se in $path_{T_2}(x_j,y_j)$ tutti gli archi hanno peso minore di $\{x_j,y_j\}$, allora T_1 non è un minimum spanning tree (c'è almeno un arco che manca in T_1 quindi potrei togliere $\{x_j,y_j\}$ da T_1 e sostituirlo con questo arco). Quindi siamo giunti ad un assurdo.

Quindi l'unica possibilità che resta aperta è che tutti gli archi che compaiono in $path_{T_2}(x_j,y_j)$ e che non sono in T_1 abbiano peso uguale al peso di $\{x_j,y_j\}$ ed inoltre c'è almeno un arco in $path_{T_2}(x_j,y_j)$ che non compare in T_1 e che ha peso uguale al peso di $\{x_j,y_j\}$. D'altra parte sicuramente questa situazione non può succedere per tutti gli archi della forma $\{x_k,y_k\}$ che compaiono in T_1 e non in T_2, perchè sappiamo che in T_2 ci sono meno archi di peso $L_1[i]$ rispetto a T_1. Quindi anche in questo caso giungiamo ad un assurdo. □

A **Minimum Spanning Tree.** L'affermazione è falsa in quanto non è detto che A sia un albero di copertura. Come controesempio si consideri il grafo $G = (\{a,b,c,d\}, \{\{a,b\},\{b,c\},\{c,a\},\{c,d\}\}, W)$, dove W assegna peso 1 ad ogni arco. Un minimum spanning tree di G è $T = \{\{b,c\},\{c,a\},\{c,d\}\}$ a cui corrisponde la lista $L = [1,1,1]$. Quindi tutti i minimum spanning tree di G avranno lista dei pesi uguale ad L. D'altra parte l'insieme $A = \{\{a,b\},\{b,c\},\{c,a\}\}$ ha lista dei pesi uguale ad L, ma non è un albero di copertura (ha un ciclo e non connette il vertice d).

Date: 11 Maggio 2009.