Esercizi risolti

R1. Esercizio

Sia dato il seguente problema: ordinare in ordine crescente gli elementi di una matrice di dimensioni n×m. Più precisamente, data una matrice M di n×m elementi, gli elementi di M devono essere ridistribuiti in modo che, per ogni i=1, ..., n, per ogni j=1, ... m-1 si abbia M[i, j] ≤ M[i, j+1], ed inoltre, per ogni i=1, ..., n-1 deve essere M[i, m] ≤ M[i+1, 1].

Per esempio, la seguente matrice 2×3

$$\begin{bmatrix} 5 & 1 \\ 9 & 13 \\ 2 & 6 \end{bmatrix}$$

viene modificata nella seguente matrice:

$$\begin{bmatrix} 1 & 2 \\ 5 & 6 \\ 9 & 13 \end{bmatrix}$$

Si scriva un algoritmo che risolva il problema suddetto e se ne calcoli la complessità temporale.

NB: si supponga pure che dato una matrice *M*, questa abbia un attributo *height*[*M*] che corrisponde al numero di righe della matrice, ed un attributo *width*[*M*] che ne rappresenta il numero di colonne.

Si supponga inoltre che per accedere all'elemento di coordinate [i, j], la sintassi da usare sia *M*[*i*][*j*].

Soluzione

Un possible algoritmo che risolve il problema dato è il seguente.

```
Sort-Matrix (M)
1 crea un array A di lunghezza width[M]*heigth[M]
2 for i ← 1 to height[M]
3   do for j ← 1 to width[M]
4         do A[(i-1)*width[M]+j] = M[i][j]
5 MERGE-SORT(A)
6 for i ← 1 to height[M]
7   do for j ← 1 to width[M]
8         do M[i][j] = A[(i-1)*width[M]+j]
```

La riga 1 richiede un tempo che è Θ(n*m). I cicli **for** delle righe 2-4 e 6-7 (che sono molto simili) richiedono entrambi un tempo Θ(n*m). La lunghezza dell'array *A* è uguale a n*m, per cui la complessità temporale del passo di MERGE-SORT è Θ(n*m*log(n*m)). La complessità globale dell'algoritmo SORT-MATRIX è quindi Θ(n*m)+Θ(n*m*log(n*m)), in cui il termine dominante è Θ(n*m*log(n*m)), che è quindi la complessità dell'algoritmo. Si noti inoltre che poichè log(n*m) = log(n)+log(m), la complessità è Θ(n*m*log(max(n,m))).

Si noti infine che se m è Θ(n), la complessità dell'algoritmo SORT-MATRIX diventa $\Theta(n^2*\log(n))$.

R2. Esercizio

Si scriva un algoritmo che, dato un array *a*, determina se l'array contiene 3 elementi x, y e z che formano una terna pitagorica (tali cioè che $x^2 + y^2 = z^2$). Se tre elementi di questo tipo esistono esso ritorna *true*, altrimenti ritorna *false*.
Quale è la complessità dell'algoritmo ideato?

NB: migliore è la complessità dell'algoritmo trovato, meglio sarà valutato l'esercizio.

Soluzione

Un semplice algoritmo che risolve il problema è il seguente:

```
FIND-PITNUM(a)
1 for i ← 1 to length[a]
2   do for j ← 1 to length[a]
3        do for k ← 1 to length[a]
4             do if a[i]*a[i] + a[j]*a[j] = a[k]*a[k]
5                  then return true
6 return false
```

Questo algoritmo ha complessità $\Theta(n^3)$, in quanto è fatto di 3 cicli innestati, ognuno che viene eseguito n volte.

Un algoritmo migliore è il seguente.

```
FIND-PITNUM(a)
1 if length[a] < 3
2   then return false
3 MERGE-SORT(a, 1, length[a])
4 i ← 3
5 while i ≤ length[a]
6   do if EXACT-SUM2(a, i-1, a[i])
7        then return true
8        else  i ← i + 1
9 return false
```

laddove l'algoritmo EXACT-SUM2 (che presuppone che gli sia passato un array *A* di lunghezza maggiore di 1 e *già ordinato*) è il seguente:

```
EXACT-SUM2(A, n, x)
1  i ← n
2  j ← 1
3  while j < i do
4    if A[i] + A[j] = x then
5       return true
6    else
7       if A[i] + A[j] < x then
8         j ← j+1
9       else
```

```
10        i ← i-1
11 return false
```

L'algoritmo EXACT-SUM2 ha complessità O(n), in quanto l'indice *i* può solo avanzare nell'array, mentre l'indice *j* può solo arretrare, per cui gli elementi dell'array *A* vengono toccati al massimo una volta durante l'algoritmo.

L'algoritmo FIND-PITNUM ha quindi complessità $O(n^2)$, in quanto l'operazione di sorting ha complessità $\Theta(n \cdot \log(n))$, mentre il susseguente ciclo viene eseguito n volte, ed ogni esecuzione ha il costo di EXACT-SUM2, che è O(n). In totale, quindi la complessità è $O(n^2)$.

R3. Esercizio

Scrivere un algoritmo che prende in ingresso una matrice quadrata di interi e la trasforma in modo da ottenere che tutti gli elementi sopra la diagonale secondaria siano minori o uguali dei valori sulla diagonale secondaria stessa, e tutti gli elementi sotto la diagonale secondaria siano maggiori o uguali degli elementi della diagonale.

Deve cioè valere la seguente proprietà (*size*(*M*) è la dimensione della matrice *M*):

$\forall i,j\ (1<i+j\leq size(M) \rightarrow \forall k(1\leq k\leq size(M) \rightarrow M[i,j] \leq M[size(M)-k+1,k])) \wedge$
$\forall i,j\ (1\leq i,j\leq size(M) \wedge size(M)+1<i+j \rightarrow \forall k(1\leq k\leq size(M) \rightarrow M[i,j] \geq M[size(M)-k+1,k]))$

Per esempio, la seguente matrice 3×3

$$\begin{bmatrix} 7 & 2 & 1 \\ -10 & 3 & 4 \\ -12 & 5 & 14 \end{bmatrix}$$

può essere trasformata (tra le altre) in una delle due seguenti matrici:

$$\begin{bmatrix} -12 & 1 & 3 \\ -10 & 4 & 7 \\ 2 & 5 & 14 \end{bmatrix}$$ oppure $$\begin{bmatrix} 1 & -12 & 3 \\ -10 & 2 & 14 \\ 4 & 5 & 7 \end{bmatrix}$$ (oppure altre, non mostrate qui).

Quale è la complessità dell'algoritmo ideato?
NB: si supponga pure che dato una matrice quadrata *M*, questa abbia un attributo *size*[*M*] che corrisponde al numero di righe e di colonne della matrice.

Si supponga inoltre che per accedere all'elemento di coordinate [i, j], la sintassi da usare sia *M*[*i*][*j*].

Soluzione

Un possibile algoritmo che risolve il problema è il seguente:

SEPARA(*M*)

```
1  crea un array A di lunghezza size[M]*size[M]
2  for i ← 1 to size[M]
3    do for j ← 1 to size[M]
4         do A[(i-1)*size[M]+j] ← M[i][j]
```

```
5  MERGE-SORT(A)
6  k ← 1
7  for i ← 1 to size[M]-1
8    do for j ← 1 to size[M]-i
9         do M[i][j] ← A[k]
10           M[size[M]-i+1][size[M]-j+1] ← A[length[A] - k]
11           k ← k + 1
12  for i ← 1 to size[M]
13    do M[size[M]-i+1][i] ← A[k]
14       k ← k + 1
```

Se consideriamo come "parametro di dimensione del problema" il numero di righe/colonne della matrice M, l'array A ha dimensione n^2, per cui la complessità temporale del passo di MERGE-SORT è $\Theta(n^2 \cdot \log(n^2))$, che è $\Theta(n^2 \cdot \log(n))$. Il ciclo alle righe 7-11, invece, ha complessità $\Theta(n^2)$, e l'ultimo ciclo (righe 12-14) è $\Theta(n)$. In definitiva la complessità globale dell'algoritmo è $\Theta(n^2 \cdot \log(n)) + \Theta(n^2) + \Theta(n)$, cioè $\Theta(n^2 \cdot \log(n))$.

R4. Esercizio

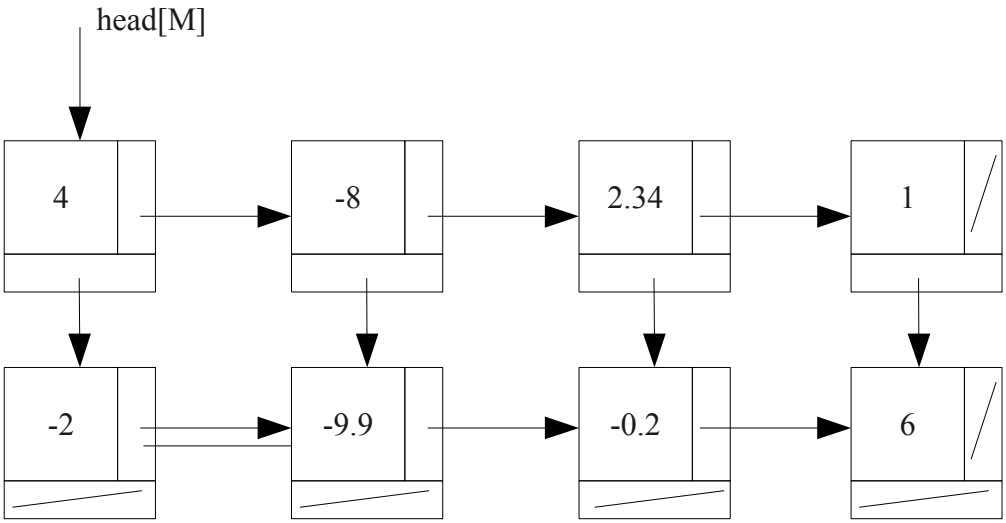
Per rendere la rappresentazione di una matrice di numeri reali dinamica, in modo da poter cambiare le sue dimensioni dinamicamente, si decide di usare una struttura dati a puntatori. Più precisamente, ogni elemento *e* della struttura corrisponde ad un elemento della matrice, ed ha i seguenti attributi: *val[e]* è il valore contenuto nell'elemento; *row[e]* è la riga su cui l'elemento si trova, mentre *col[e]* è la colonna; *right[e]* è il riferimento all'elemento che si trova a destra, cioè sulla stessa riga, ma nella colonna successiva; *down[e]* è il riferimento all'elemento che si trova in basso, cioè sulla stessa colonna, ma sulla riga successiva. Gli elementi sull'ultima colonna hanno NIL come valore dell'attributo *right*, mentre quelli sull'ultima riga hanno NIL come valore dell'attributo *down*.

Una matrice M è una struttura con un attributo, *head[M]*, che è il riferimento all'elemento in posizione [1,1] della matrice (NIL se la matrice è vuota).

Per esempio, la matrice

$$\begin{bmatrix} 4 & -8 & 2.34 & 1 \\ -2 & -9.9 & -0.2 & 6 \end{bmatrix}$$

è rappresentata dalla seguente struttura:



Scrivere un algoritmo che, data in ingresso una matrice M di dimensioni qualsiasi, la trasforma, cambiandola nella sua trasposta.

Quale è la complessità dell'algoritmo?

Soluzione

Un possibile algoritmo è il seguente:

TRANS(*M*)

```
1  if M = NIL
2    return
3  TRANSSUBMATRIX(head[M])
```

Il cuore dell'algoritmo è dato dal seguente sottoalgoritmo, che traspone una sottomatrice, indicata dal primo elemento in alto a sinistra.

TRANSSUBMATRIX(*E*)

```
1  if E = NIL
2    return
3  if down[E] ≠ NIL and right[E] ≠ NIL
4    TRANSSUBMATRIX(right[down[E]])
5  curr ← right[E]
6  while curr ≠ NIL
7    do next ← right[curr]
8       temp ← right[curr]
9       right[curr] ← down[curr]
10      down[curr] ← temp
11      vtemp ← col[curr]
12      col[curr] ← row[curr]
13      row[curr] ← vtemp
14      curr ← next
15 curr ← down[E]
16 while curr ≠ NIL
17   do curr ← down[curr]
18      temp ← down[curr]
```

```
19      down[curr] ← right[curr]
20      right[curr] ← temp
21      vtemp ← col[curr]
22      col[curr] ← row[curr]
23      row[curr] ← vtemp
24      curr ← next
```

Si noti che l'algoritmo si compone di 3 parti.

Nella prima parte, l'algoritmo viene invocato ricorsivamente per trasporre la sottomatrice che comincia nell'elemento di posizione (2,2).

Fatto questo, viene prima trasposta la prima riga della matrice, poi la sua prima colonna. Per trasporre una riga/colonna è sufficiente scambiare il puntatore *right* con il puntatore *down*, e si ottiene l'effetto desiderato.

Se la matrice M ha dimensione m× n, la complessità dell'algoritmo è semplicemente Θ(mn), in quanto tutti gli elementi della matrice vengono toccati (e trasposti) esattamente una volta.

R5. Esercizio

Si scriva un algoritmo che, dato un array di elementi, ritorna un albero binario quasi completo contente gli stessi elementi dell'array.

Quale è la complessità dell'algoritmo scritto?

Note.

- Gli alberi binari sono da intendersi come implementati mediante l'usuale struttura a puntatori fatta di nodi con attributi *left*, *right*, *key* (per semplicità, si trascuri l'attributo *p*).
- Per creare un nodo di un albero si usi pure una pseudo istruzione "crea nodo *x*", di costo costante.
- E' possibile riutilizzare (sotto)algoritmi noti per la realizzazione degli algoritmi dell'esercizio.

Soluzione

Definiamo prima un algrotimo di supporto ricorsivo, che sarà poi il cuore dell'algoritmo generale. Questo sottoalgoritmo prende in ingresso un array A e l'indice di un elemento di A, ed interpreta questo come la radice di un sottoalbero quasi completo. Il sottalbero quasi completo è costritui quindi in modo ricorsivo a partire dalla radice, e alla fine viene ritornato il nodo radice dell'albero così costruito.

```
CreaSottoAlberoQC(A, i)
1  if i ≤ length[A] then
2     crea nodo x
3     key[x] ← A[i]
```

```
4     left[x] ← CreaAlberoQC(A, 2*i)
5     right[x] ← CreaAlberoQC(A, 2*i+1)
6     return x
7   else
8     return NIL
```

L'algoritmo globale è quindi il seguente:

```
CreaAlberoQC(A)
1  crea albero vuoto T
2  root[T]← CreaSottoAlberoQC(A, 1)
3  return T
```

La complessità dell'algoritmo CreaSottoAlberoQC è sempre lineare nel numero di elementi (n) del sottoalbero (i quali vengono "toccati" una volta sola, e l'aggiunta di ognuno dei quali nell'albero ha costo costante, in quanto questa operazione viene fatta semplicemente creando un nuovo nodo radice nel sottoalbero corrispondente), quindi Θ(n).

Di conseguenza, l'algoritmo CreaAlberoQC ha complessità anch'esso Θ(n), con n lunghezza dell'array A.

R6. Esercizio

Si scriva un algoritmo che, dato un array di elementi, ritorna un albero binario *di ricerca bilanciato* contente gli stessi elementi dell'array.

Quale è la complessità dell'algoritmo scritto?

Note.

- Per questo esercizio diciamo che un albero binario è *bilanciato* se, per ogni nodo *x* dell'albero, le altezze del sottoalbero destro e del sottoalbero sinistro di *x* differiscono *al più* di 1.
- Gli alberi binari sono da intendersi come implementati mediante l'usuale struttura a puntatori fatta di nodi con attributi *left*, *right*, *key* (per semplicità, si trascuri l'attributo *p*).
- Per creare un nodo di un albero si usi pure una pseudo istruzione "crea nodo *x*", di costo costante.
- E' possibile riutilizzare (sotto)algoritmi noti per la realizzazione dell'algoritmo dell'esercizio.

Soluzione

Definiamo prima un algrotimo di supporto ricorsivo, che sarà poi il cuore dell'algoritmo generale. Questo sottoalgoritmo prende in ingresso un sottoarray A[i] .. A[f] di un array A ordinato, e

costruisce un albero binario di ricerca bilanciato a partire da esso, ritornando alla fine il nodo radice dell'albero costruito. Per costruire l'albero in modo che sia bilanciato, nella radice dello stesso viene messo l'elemento di mezzo del sottoarray, e poi i due sottoalberi sinistro e destro sono costruiti in modo ricorsivo.

```
BSTDaSottoArray(A, i, f)
1  if i ≤ f then
2    m  ← ⌊i + f / 2⌋
3    crea nodo x
4    key[x] ← A[m]
5    left[x] ← BSTDaSottoArray(A, i, m-1)
6    right[x] ← BSTDaSottoArray(A, m+1, f)
7    return x
8  else
9    return NIL
```

L'algoritmo globale è quindi il seguente:

```
BSTDaArray(A)
1  crea albero vuoto T
2  MergeSort(A)
3  root[T]← BSTDaSottoArray(A, 1, length[A])
4  return T
```

La complessità dell'algoritmo BSTDaSottoArray è sempre lineare nel numero di elementi (n) del sottoalbero (i quali vengono "toccati" una volta sola, e l'aggiunta di ognuno dei quali nell'albero ha costo costante, in quanto questa operazione viene fatta semplicemente creando un nuovo nodo radice nel sottoalbero corrispondente), quindi $\Theta(n)$.

La complessità dell'algoritmo BSTDaArray è quindi $\Theta_{MergeSort} + \Theta(n)$, e siccome $\Theta_{MergeSort}$ è $\Theta(n \log(n))$, la complessità globale è anch'essa $\Theta(n \log(n))$.

R7. Esercizio

parte a.

Si scriva un algoritmo che, dato un grafo *G* non orientato, un nodo di partenza *s*, un nodo di arrivo *t*, ed un array di nodi "proibiti" *P*, dice se esiste un cammino da *s* a *t* che non tocca nessun nodo dell'insieme *P* (si supponga che sia sempre $s \neq t$).

Quale è la complessità dell'algoritmo ideato nel caso peggiore? E nel caso migliore? (Specificare anche quali sono il caso peggiore ed il caso migliore.)

parte b.

Modificare l'algoritmo in modo che, se un cammino tra *s* e *t* che non passa per i nodi di *P* esiste, esso ritorni la lunghezza del cammino più corto tra *s* e *t* che non passa per nodi di *P* (si ricorda che la lunghezza di un cammino è data dal numero di archi che lo compongono).

Soluzione

Per ottenere un algoritmo che risolve il problema desiderato (inclusa la richiesta del punto b) basta modificare opportunamente l'algoritmo di BFS per grafi, nella seguente maniera:

```
CAMMINONONPROIBITO(G, s, t, P)
1  for each vertex u ∈ V[G] − {s} do
2    color[u] ← WHITE
3    dist[u] ← ∞
4  color[s] ← GREY
5  dist[s] ← 0
6  for i ← 1 to length[P] do
7    color[P[i]] ← RED
8  ENQUEUE(Q, s)
9  while Q ≠ ∅ do
10   u ← DEQUEUE(Q)
11   for each v ∈ Adj[u] do
12     if v = t then
13       return dist[u] + 1
14     else if color[v] = WHITE then
15       color[v] ← GRAY
16       dist[v] ← dist[u] +1
17       ENQUEUE(Q, v)
18   color[u] ← BLACK
19 return ∞
```

Le uniche modifiche da fare all'algoritmo di BFS riguardano una colorazione iniziale dei nodi proibiti a "rosso" (che impedisce che questi vengano visitati all'istruzione 14 del successivo ciclo), ed il controllo che il nodo v da visitare non sia il nodo di arrivo desiderato (nel qual caso l'algoritmo deve uscire).

La distanza da ritornare è esattamente quella calcolata dall'algoritmo di BFS, che si può appunto dimostrare essere quella minima tra 2 nodi.

La complessità dell'algoritmo è la stessa di BFS (il ciclo aggiuntivo 6-7 ha complessità la massimo O(|V|), che è dominata dalla complessità del ciclo *while*). Nel caso peggiore (quello in cui un

cammino tra *s* e *t* non esiste), il grafo va visitato interamente, e la complessità è $\Theta(|V|+|E|)$. Nel caso migliore, invece (quello in cui *s* e *t* sono adiacenti, e *t* è il primo nodo analizzato alla linea 11) la complessità è $\Theta(1)$ per il ciclo *while* 9-18, mentre è $\Theta(|V|)$ per il ciclo *for* 1-3, quindi, in totale è $\Theta(|V|)$ (il ciclo *for* 6-7 non sarà mai più costoso di $\Theta(|V|)$). Possiamo quindi dire che, nel complesso, l'algoritmo `CamminoNonProibito` è $\Omega(|V|)$ e $O(|V|+|E|)$.

R8. Esercizio

Si vuole realizzare un tipo di dato per rappresentare insiemi di numeri reali. Le operazioni che si vogliono poter fare su un insieme di numeri sono le seguenti:

- `INSERT(S, v)`, che prende ingresso un insieme *S* ed un valore reale *v*, e se il valore *v* non esiste già in *S* lo inserisce in *S*, altrimenti non fa nulla;
- `DELETE(S, v)`, che prende in ingresso insieme *S* ed un valore reale *v*, ed elimina il valore *v* da *S* (si supponga pure che il valore *v* esista in *S*);
- `CLOSERTOAVG(S)`, che prende in ingresso un insieme *S*, e ritorna il valore contenuto di *S* che più si avvicina alla media dei valori contenuti in *S*.

Definire una struttura dati (come è fatta, quali attributi ha, ecc.) che implementi il tipo "insieme di numeri reali" descritto sopra.

Si descriva un algoritmo che realizzi l'operazione `CLOSERTOAVG(S)`, e se ne dia la complessità. Si descriva inoltre un algoritmo che realizzi una (*e una sola!*) a scelta tra le operazioni di `INSERT` e di `DELETE`, e se ne dia la complessità.

NB: E' preferibile che gli algoritmi vengano descritti in pseudocodice; tuttavia, verranno accettate anche soluzioni che descrivano solo in modo informale (cioè in italiano) gli algoritmi, purché queste siano sufficientemente precise per poter valutare la complessità dell'algoritmo.

Soluzione

La maniera più semplice di implementare l'insieme di reali richiesto è mediante una lista concatenata di elementi. Per facilitare l'operazione “caratterizzate” dell'insieme (cioè `CLOSERTOAVG`) si può mantenere la lista ordinata in ordine crescente di chiave, e si può aumenta le lista con un attributo, *avg*[*L*], che tiene traccia della media degli elementi inseriti nell'insieme, e che va aggiornato (con costo costante) ad ogni inserimento/cancellazione.

L'operazione di inserimento diventa quindi come un semplice passo di INSERT-SORT, che ha complessità $\Theta(n)$ nel caso peggiore. L'operazione di `CLOSERTOAVG(S)` può essere fatta scorrendo la lista fino a che non si arriva un numero più alto di *avg*[*L*], operazione che comunque, nel caso peggiore, ha complessità lineare $\Theta(n)$. La cancellazione è duale dell'inserimento, e richiede alla peggio di scorrere la lista per trovare l'elemento da cancellare, quindi ha complessità anch'essa $\Theta(n)$.

Una maniera alternativa di implementare l'insieme di numeri reali richiesto è tramite un array di elementi aumentato con 3 attributi: uno (*size*[*A*]), che tiene traccia dell'effettiva occupazione dell'array, uno (*avg*[*A*]) che tiene traccia della media degli elementi inseriti nell'array, ed uno (*avg_el*[*A*]) che tiene traccia dell'indice dell'elemento più vicino alla media.

Si possono implementare le operazioni di inserimento e cancellazione in modo da mantenere l'array ordinato, e da aggiornare opportunamente gli attributi *avg* e *avg_el*.

In questo modo, l'operazione di CLOSERTOAVG(S) si riduce ad una banale istruzione (di costo ovviamente costante):

```
CLOSERTOAVG(S)
1  return avg_el[S]
```

Naturalmente, il prezzo che si paga per la semplicità dell'operazione CLOSERTOAVG è una maggiore complicatezza nelle operazioni di inserimento e cancellazione degli elementi da un insieme.

Prendiamo per esempio l'operazione di inserimento di un elemento nell'array. Per mantenere l'array ordinato si può ricorrere ad un algoritmo di tipo INSERT-SORT, in cui prima si individua il punto in cui inserire il nuovo elemento nell'array, quindi "si fa spazio" nell'array per il nuovo elemento e lo si inserisce. Contestualmente si aggiornano gli attributi *avg* e *avg_el* dell'insieme. La complessità dell'algoritmo è lineare ($\Theta(n)$), in quanto l'operazione più costosa è la "creazione del buco" nell'array:

```
INSERT(S, v)
1  if size[S] = 0 then
2    S[1] ← v
3    avg[S] ← v
4    avg_el[S] ← 1
5  else
6  ▷ troviamo, con meccanismo di tipo ricerca binaria, il punto in cui
inserire il nuovo elemento
7    i ← 1
8    j ← size[S]
9    while i ≤ j do
10     m ← ⌊i + j /2⌋
11     if S[m] = v then
12       return       ▷ l'elemento esiste, non c'è niente da fare
13     else if S[m] > v then
14       j ← m-1
15     else
16       i ← m+1
17   ▷ il nuovo elemento va inserito nell'indice n corrispondente
all'ultimo valore di i
18   n ← i
19   ▷ creiamo il "buco"
20   for i ← size[S] downto n do
21     S[i+1] ← S[i]
22   S[n] ← v
```

```
23   size[S] ← size[S]+1
24   avg[S] ← ((avg[S] * (size[S]-1))+v) / size[S]
25   ▷ da ultimo, aggioriamo l'attributo avg_el, di nuovo usando un
algoritmo in stile ricerca binaria
26   if n ≤ avg_el then
27     i ← n
28     j ← avg_el+1
29   else
30     i ← avg_el
31     j ← n
32   while j > i+1 do
33     m ← ⌊i + j /2⌋
34     if S[m] = avg[S] then
35       avg_el[S] ← m
36       return
37     else if S[m] ≥ avg[S] then
38       j ← m
39     else
40       i ← m
41   if |avg[S] − S[i]| ≤ |avg[S] − S[i]| then
42     avg_el[S] ← i
43   else
44     avg_el[S] ← j
```

La complessità di tale algoritmo è Θ(log(n)) per le parti di ricerca binaria (linee 6-16 e 25-44), mentre è Θ(n) per il ciclo `for` 20-21. In ultima analisi, la complessità dell'algoritmo nel caso peggiore è quindi Θ(n).

Similmente si può implementare l'algoritmo di DELETE.

Un limite dell'implementazione di cui sopra è naturalmente dato dalla finitezza degli array, che non permettono di inserire un numero illimitato di elementi. Una soluzione alternativa (e dalla migliore complessità) potrebbe prevedere per esempio l'uso di un albero bilanciato, di nuovo aumentato con un attributo *avg* che tenga traccia della media degli elementi nell'insieme. Usando alberi bilanciati, si possono ottenere complessità Θ(log(n)) sia per l'inserimento, che per la cancellazione, che la ricerca dell'elemento più vicino alla media.

R9. Esercizio

Si vuole realizzare un tipo di dato per rappresentare insiemi di oggetti contenenti dati di persone. Ognuno di questi oggetti *P* è caratterizzato da 3 attributi: `Nome[P]` contiene il nome della persona, `CF[P]` il codice fiscale, e `DataN[P]` la data di nascita.

Le operazioni che si vogliono fare su un insieme *SP* di persone sono le seguenti:

- `INSERT(SP, P)`, che prende ingresso un insieme *SP* ed un oggetto *P* rappresentate i dati di una persona, ed inserisce *P* in *SP* (si supponga pure che i dati della persona *P* non siano ancora contenuti nell'insieme);
- `FINDBYNAME(SP, N)`, che prende in ingresso insieme *SP* ed il nome *n* di una persona e, se i dati di una persona di nome *n* sono presenti in *SP*, li restituisce, altrimenti restituisce *NIL*;
- `FINDBYCF(SP, CF)`, che prende in ingresso insieme *SP* ed il codice fiscale *cf* di una persona e, se i dati di una persona con codice fiscale *cf* sono presenti in *SP*, li restituisce, altrimenti restituisce *NIL*.

Definire una struttura dati (come è fatta, quali attributi ha, ecc.) che implementi il tipo "insieme di dati di persone" descritto sopra.

Si descrivano degli algoritmi che realizzino le operazioni `INSERT(SP, P)`, `FINDBYNAME(SP, N)` e `FINDBYCF(SP, CF)`, e se ne diano le complessità.

NB1: Si supponga pure che sia possibile confrontare interi nomi e codici fiscali mediante i soliti operatori di >, <, =.

NB2: Il punteggio ottenuto sarà tanto più alto quanto più efficienti saranno gli algoritmi implementati.

NB3: E' preferibile che gli algoritmi vengano descritti in pseudocodice; tuttavia, verranno accettate anche soluzioni che descrivano solo in modo informale (cioè in italiano) gli algoritmi, purché queste siano sufficientemente precise per poter valutare la complessità dell'algoritmo.

Soluzione

Un modo efficiente per implementare un insieme in cui le chiavi di ricerca sono 2 (in questo caso nome e codice fiscale) potrebbe essere quello di usare un albero di ricerca (anche bilanciato) modificato, in cui la struttura "di ricerca" è duplice: una basata sull'attributo "nome", ed una basata sull'attributo "codice fiscale".

Di fatto, in un tale albero ogni nodo *N* avrebbe 4 attributi: *leftN*[*N*] e *rightN*[*N*] per il nome, e *leftCF*[*N*] e *rightCF*[*N*] per il codice fiscale, con i vincoli che:

- $Nome[key[N]] \geq Nome[key[leftN[N]]]$
- $Nome[key[N]] \leq Nome[key[rightN[N]]]$
- $CF[key[N]] \geq CF[key[leftCF[N]]]$
- $CF[key[N]] \leq CF[key[rightCF[N]]]$

Inoltre, l'albero avrebbe una duplice radice, *rootN*[*T*] e *rootCF*[*T*], per tenere conto della duplice struttura.

Gli algoritmi di ricerca, quindi, sono dei semplici `TREE-SEARCH`, modificati opportunamente:

```
FINDBYNAME(SP, n)
1  TREE-SEARCHN(rootN[SP], n)
```

laddove

```
TREE-SEARCHN(N, n)
1  if N = NIL or n = Name[key[N]]
2  return N
3  if n < Name[key[N]]
4    then return Tree-SearchN(leftN[N], n)
5    else return Tree-SearchN(rightN[N], n)
```

Similmente:

```
FINDBYCF(SP, cf)
1  TREE-SEARCHCF(rootCF[SP], cf)
```

laddove

```
TREE-SEARCHCF(N, cf)
1  if N = NIL or n = CF[key[N]]
2  return N
3  if n < CF[key[N]]
4    then return Tree-SearchCF(leftCF[N], n)
5    else return Tree-SearchCF(rightCF[N], n)
```

Similmente, l'algoritmo di inserimento è una combinazione di 2. Nel caso di un albero non bilanciato, l'algoritmo sarebbe il seguente:

```
TREE-INSERT(T, z)
1  y ← NIL
2  x ← rootN[T]
3  while x ≠ NIL
4    do y ← x
5      if Name[key[z]] < Name[key[x]]
6        then x ← leftN[x]
7        else x ← rightN[x]
8  pN[z] ← y
9  if y = NIL
10   then rootN[T] ← z
11   else if Name[key[z]] < Name[key[x]]
12       then leftN[y] ← z
13       else rightN[y] ← z
14 y ← NIL
15 x ← rootCF[T]
16 while x ≠ NIL
```

```
17   do y ← x
18     if CF[key[z]] < CF[key[x]]
19       then x ← leftCF[x]
20       else x ← rightCF[x]
21 pCF[z] ← y
22 if y = NIL
23   then rootCF[T] ← z
24   else if CF[key[z]] < CF[key[x]]
25       then leftCF[y] ← z
26       else rightCF[y] ← z
```

Le complessità sono identiche a quelle dei BST normali. Nel caso non bilanciato, sono tutte O(h), con h l'altezza dell'albero.

R10. Esercizio

parte a.

Si supponga di scrivere una variante di MERGE-SORT, chiamata MERGE-SORT-4X che, invece di suddividere l'array da ordinare in 2 parti (e ordinarle separatamente), lo suddivide in 4 parti, le ordina ognuna riapplicando MERGE-SORT-4X, e le riunifica usando un'opportuna variante MERGE-4X di MERGE (la quale, naturalmente, fa la fusione su 4 sottoarray invece di 2).

Come cambia, se cambia, la complessità temporale di MERGE-SORT-4X rispetto a quella di MERGE-SORT?

Come cambia, se cambia, la complessità temporale di una variante MERGE-SORT-KX di MERGE-SORT che spezza l'array in K sottoarray?

Giustificare brevemente le risposte.

NB: Non si richiede di scrivere lo pseudocodice né di MERGE-SORT-4X, né di MERGE-4X, né di MERGE-SORT-KX.

parte b.

Si vuole calcolare il massimo di un array tramite un algoritmo di tipo divide-et-impera.

Più precisamente, l'algoritmo funziona nella seguente maniera: divide l'array in 2 parti, e calcola separatamente il massimo sm del sottoarray di sinistra ed il massimo dm del sottoarray di destra; il valore ritornato (il massimo richiesto) è il maggiore tra sm e dm.

Si scriva lo pseudocodice dell'algoritmo delineato, e se ne calcoli la complessità temporale mediante il master theorem.

Soluzione

parte a.

L'equazione alle ricorrenze per un generico MERGE-SORT-KX è la seguente:

$T(n) = kT(n/k) + \Theta(n)$ (con $T(n) = \Theta(1)$ se $n<1$)

in quanto la complessità di MERGE-KX è banalmente lineare nel numero di elementi da fondere. Ora, poichè $n^{\log_k k} = n$, applicando il master theorem la soluzione dell'equazione risulta essere una funzione della classe $\Theta(n \log(n))$, qualunque sia k, che corrisponde anche alla complessità di MERGE-SORT.

parte b.

Lo pseudocodice dell'algoritmo è il seguente:

```
MAX-ARRAY(A, p, r)
1  if r<p then
2    return -∞
3  m  ← ⌊p + r / 2⌋
4  sm ← MAX-ARRAY(A, p, m)
5  dm ← MAX-ARRAY(A, m+1, r)
6  return MAX(sm, dm)
```

La corrispondente equazione alle ricorrenze è la seguente (la complessità temporale della funzione MAX è banalmente costante, in quanto realizzabile per esempio mediante un semplice if):

$T(n) = 2T(n/2) + \Theta(1)$ (con $T(n) = \Theta(1)$ se $n<1$)

la quale ha per soluzione una funzione in $\Theta(n)$, in quanto $f(n) = \Theta(1)$ e $n^{\log_b a} = n^{\log_2 2} = n$.

R11. Esercizio

Si abbia un array *A* di oggetti, ognuno con 2 attributi, *color* e *data*. L'attributo *color* può avere valore o 'bianco', o 'giallo', o 'rosso', mentre l'attributo *data* contiene i dati anagrafici di una persona, e non viene dettagliato ulteriormente.

Si vuole scrivere un algoritmo che prende in ingresso un array *A* fatto come sopra, e deve riordinare *A* in modo che gli elementi con attributo *color* uguale a 'rosso' precedano quelli con *color* uguale a 'giallo', che a loro volta precedono quelli con *color* uguale a 'bianco' (l'ordine in cui alla fine si trovano tra di loro gli elementi di colore rosso può essere qualunque, ed analogamente per gli elementi di colore giallo e bianco).

Inoltre, per questioni di limiti della memoria disponibile sul processore su cui si vuole implementare l'algoritmo, questo dovrebbe essere scritto in modo da usare, oltre all'array A, non più di 10 variabili di tipo semplice (cioè di tipo diverso da array), e nessun altro array oltre ad A.

Si scriva un algoritmo che risolve il problema richiesto con i vincoli di memoria dati, e se ne dia la complessità temporale.

Soluzione

Il problema può essere risolto per esempio mediante una variante di INSERTION-SORT che ordini sull'attributo *color* dei vari elementi $A[i]$. INSERTION-SORT in effetti rispetta i vincoli desiderati (non usa array ausiliari), però ha complessità temporale quadratica nel caso peggiore ($T(n) = O(n^2)$).

Un algoritmo più efficiente, di complessità lineare, che risolve il problema desiderato è il seguente (che è una variante di COUNTING-SORT):

```
SORT-COLORS(A)
1   r ← g ← b ← 0
2   for i ← 1 to length[A] do
3     if color[A[i]] = 'rosso' then
4       r  ← r + 1
5     if color[A[i]] = 'giallo' then
6       g  ← g + 1
7     if color[A[i]] = 'bianco' then
8       b  ← b + 1
9   i ← 1
10  g ← g + r
11  b ← b + g
12  while(i ≤ length[A]) do
13    if (color[A[i]] = 'rosso' and i < r) then
14      SWAP A[i] ↔ A[r]
15      r ← r − 1
16    else if (color[A[i]] = 'giallo' and i < g) then
17      SWAP A[i] ↔ A[g]
18      g ← g − 1
19    else if (color[A[i]] = 'bianco' and i < b) then
20      SWAP A[i] ↔ A[b]
21      b ← b − 1
22    else i ← i + 1
```

La complessità di questo algoritmo è lineare, in quanto il ciclo **for** delle linee 2-8 viene eseguito n volte (con n la lunghezza dell'array A), mentre il ciclo **while** delle istruzioni 12-22 viene eseguito al massimo $2n$ volte (nel caso in cui vengano prima fatti tutti gli swap, che fanno decrementare r, g e b, e poi venga fatto avanzare l'indice i fino alla fine dell'array).

Altre soluzioni, anch'esse di complessità lineare nella lunghezza dell'array A sono possibili (ad esempio mediante un algoritmo, che, con un'unica passata sull'array A, mette tutti gli elementi di colore 'rosso' nella prima parte di A e tutti gli elementi di colore 'bianco' nell'ultima parte di A, il che, per costruzione, garantisce che tutti gli elementi di colore 'giallo' rimangano in mezzo).

R12. Esercizio

Si consideri il caso di alberi qualunque, tali cioè che ogni nodo possa avere un numero di figli arbitrario, grande a piacere. E' possibile rappresentare tali alberi mediante strutture a puntatori in cui ogni nodo `n` ha i seguenti attributi:

- `key[n]` rappresenta il valore contenuto nel nodo;
- `fc[n]` (abbreviazione per "first child") è il puntatore al figlio più a sinistra;
- `rs[n]` (abbreviazione per "right sibling") è il puntatore al nodo "fratello" immediatamente a destra di `n`;
- `ls[n]` (abbreviazione per "left sibling") è il puntatore al nodo "fratello" immediatamente a sinistra di `n`;
- `p[n]` è il puntatore al nodo padre di `n`.

Se `T` è un albero generico, `root[T]` è il suo nodo radice. Si noti che il nodo radice, oltre a non avere padre (i.e. sarà sempre `p[root[T]] = NIL`), non ha neanche fratelli (i.e. sarà sempre `rs[root[T]] = NIL`, oltre che `ls[root[T]] = NIL`).

Si vuole definire per alberi generici l'analogo dei BST per alberi binari (si vogliono cioè definire degli alberi "di ricerca" con numero arbitrario di figli).

1. Che proprietà dovrebbe soddisfare un albero generico implementato come indicato sopra per essere "di ricerca"?

2. Si definisca uno (e uno solo!) dei 2 seguenti algoritmi per alberi generici di ricerca (secondo la definizione data al punto 1):

- search(T, k)
- insert(T, n)

laddove T è un albero, k una chiave, ed n un nodo. Si dia la complessità temporale dell'algoritmo definito.

Soluzione

Per definire quale deve essere la proprietà mantenuta da un albero generico perchè questo sia "di ricerca" è sufficiente notare che, in un nodo `n` di un albero generico, i puntatori `fc[n]` e `rs[n]` di fatto possono giocare il ruolo che, in un albero binario, giocano `left[n]` e `right[n]`.

Di conseguenza, basta richiedere che, dato un nodo `n`, il valore della sua chiave sia >= delle chiavi di tutti i suoi figli, e sia <= delle chiavi di tutti i suoi fratelli (e dei loro sottoalberi), e di fatto si è ricreata la stessa situazione che si aveva per gli alberi binari di ricerca.

Più precisamente, è sufficiente richiedere che, per ogni nodo `n`, per ogni nodo `d` che è suo discendente sia

```
key[d] <= key[n]
```

e per ogni nodo `rsd` che è o un suo fratello di destra, o un discendente di quest'ultimo sia

```
key[n] <= key[rsd]
```

A questo punto, gli algoritmi di `search(T, k)` e di `insert(T, n)` sono di fatto gli stessi (con piccole modifiche) di quelli per gli alberi binari. Più precisamente.

```
search(T, k)
1 search-node(root[T], k)
```

laddove si ha:

```
search-node(x, k)
1 if x = NIL or k = key[x]
2  return x
3 if k < key[x]
4   then return search-node(fc[x], k)
5   else return search-node(rs[x], k)
```

Si noti che, per come è definita la proprietà di essere "di ricerca" per alberi generici, il nodo radice è anche quello che contiene la chiave con valore massimo.

Inoltre:

```
insert(T, n)
1  y ← NIL
2  x ← root[T]
3  while x ≠ NIL
4    do y ← x
5      if key[n] < key[x]
6        then x ← fc[x]
7        else x ← rs[x]
8  p[z] ← y
9  if y = NIL
10   then root[T] ← n          ▷ L'albero era vuoto
11   else if key[n] < key[y]
12     then fc[y] ← n
```

```
13      else if p[y] = NIL
14        ▷ Il nuovo nodo andrebbe attaccato come fratello della
15        ▷ radice, ma non si può, quindi va creata una nuova
16        ▷ radice
17        fc[n] ← y
18        root[T] ← n
19        p[y] ← n
20      else
21        rs[y] ← n
22        ls[n] ← y
23        p[n] ← p[y]
```

Per entrambi gli algoritmi, la complessità nel caso peggiore corrisponde alla distanza massima md che un nodo può avere dalla radice (che non è più esattamente la profondità dell'albero), cioè $\Theta(md)$.

R13. Esercizio

Si vuole rappresentare una situazione in cui si ha un insieme di n pugili. Alcuni di questi pugili sono tra loro rivali (a 2 a 2, cioè ci sono coppie di pugili che sono tra loro rivali). Ogni pugile può avere un numero arbitrario di rivali (anche zero).

a. Si descriva una struttura dati che sia adatta a rappresentare la situazione di cui sopra.

Si vorrebbe suddividere l'insieme dei pugili in 2 sottoinsiemi B (che sta per "buoni") e C (che sta per "cattivi"), in modo tale che non ci siano 2 pugili dello stesso sottoinsieme che sono rivali tra loro.

b. Si scriva un algoritmo che, dato un insieme di pugili descritto mediante una struttura dati come quella identificata al punto a, faccia quanto segue:

- se c'è almeno una maniera per suddividere i pugili in "buoni" e "cattivi" come descritto sopra, identifica *una* tra le possibili suddivisioni e "ritorna true";
- altrimenti "ritorna false".

Si dia la complessità temporale dell'algoritmo definito.

NB1: Non è necessario che i 2 sottoinsiemi abbiano lo stesso numero di pugili (in effetti un insieme può anche essere molto più grande dell'altro), purché sia rispettato il vincolo sulle rivalità.

NB2: E' preferibile che l'algoritmo venga descritto in pseudocodice; tuttavia, verranno accettate anche soluzioni che descrivano solo in modo informale (cioè in italiano) l'algoritmo, purché queste siano sufficientemente precise per poter valutare la complessità dell'algoritmo.

Soluzione

a. La maniera migliore per rappresentare la situazione desiderata è mediante un grafo (rappresentato mediante liste di adiacenza), in cui i nodi rappresentano i pugili, e gli archi le relazioni di rivalità. E' naturale pensare che il grafo sia indiretto, che comunque può essere semplicemente rappresentato come un grafo diretto in cui se c'è un arco (n1, n2), ci deve essere anche un arco simmetrico (n2, n1) (il che corrisponde al fatto appunto che la relazione di rivalità sia simmetrica, per cui non è possibile che un pugile sia rivale di un altro ma non il viceversa).

b. Avendo scelto di rappresentare la situazione con un grafo, per suddividere i pugili i 2 categorie è sufficiente modificare l'algoritmo di DFS, nella maniera descritta sotto, usando come "colori" o "U" (per "undef"), o "B" (per "buono"), o "C" (per "cattivo"). Nell'algoritmo, ad ogni invocazione successiva dell'algoritmo di visita il colore con cui viene marcato il nuovo nodo viene alternato tra "B" e "C" (partendo da "B"), secondo quanto indicato nel secondo parametro dell'algoritmo.

```
Buoni-cattivi(G)
1  for each vertex u ∈ V[G] do
2    color[u] ← UNDEF
3  for each vertex u ∈ V[G] do
4     if color[u] = UNDEF then
5       ris ← visita_e_partiziona(u, B)
6       if ris = false then
7         return false
8  return true;
```

```
visita_e_partiziona(u, c)
1  color[u] ← c
2  for each v ∈ Adj[u] do
3    if color[v] = U then
4       if c = B then
5         ris ← visita_e_partiziona(v, C)
6       else
7         ris ← visita_e_partiziona(v, B)
8       if ris = false then
9         return ris
10   else if color[v] = c then
11      return false
12 return true
```

La complessità dell'algoritmo è la stessa di `DFS`, cioè $\Theta(|V| + |E|)$, laddove |V| è il numero di pugili, e |E| è il numero di rivalità.

R14. Esercizio

1. Scrivere un algoritmo che, data una lista monodirezionale (cioè "singly linked") in cui ogni elemento contiene un carattere, ritorna `true` se la parola rappresentata dalla lista è un palindromo (cioè se è uguale sia che sia letta da destra a sinistra, sia che sia letta da sinistra a destra, per esempio "radar", o "ossesso").

Dare la complessità dell'algoritmo scritto.

2. Scrivere un algoritmo che risolva lo stesso problema del punto 1, ma con il vincolo di poter utilizzare, oltre alla lista (che *non può essere modificata*), al massimo 10 variabili *semplici* (diciamo che una variabile è semplice se può contenere solo o un numero, o un carattere, o un riferimento a un oggetto già esistente prima dell'invocazione dell'algoritmo).

Dare la complessità dell'algoritmo scritto.

NB: gli algoritmi possono essere descritti o in pseudocodice, o informalmente (in linguaggio naturale), purchè comunque la descrizione sia sufficientemente precisa per poter valutare la complessità dell'algoritmo.

Soluzione

1.

Se non si hanno vincoli sulla crezione di nuovi oggetti, il modo più semplice per dire se la parola è un palindromo senza poter scorrere all'indietro la lista L in input (che è il vincolo imposto dal fatto che la lista sia "monodirezionale") è quello di scorrere L, creando una nuova lista L', che contiene gli stessi elementi di L invertiti (cosa che si può facilmente ottenere semplicemente inserendo via via gli elementi di L in testa ad L'), e poi confrontando le 2 liste L ed L'.

In pseudocodice (in cui si fa uso di un algoritmo LIST-INSERT che è analogo a quello visto a lezione, con la differenza che questo agisce su liste monodirazionali invece che bidirezionali):

```
Palindromo(L)
1  x ← head[L]
2  crea lista L' vuota
3  while x ≠ NIL do
4     crea nodo n
5     key[n] ← key[x]
6     LIST-INSERT(L', n)
7     x ← next[x]
8  x ← head[L]
```

```
9  y ← head[L']
10 while x ≠ NIL do
11   if key[x] ≠ key[y] then
12     return false
13   x ← next[x]
14   y ← next[y]
15 return true
```

Naturalmente la complessità di questo algoritmo è, nel caso peggiore (quello in cui la parola è un palindromo), Θ(n), in quanto occorre semplicemente scorrere 2 volte la lista L per intero.

2.

In questo secondo caso non si può utilizzare una lista L' di supporto. Di conseguenza occorre ogni volta scorrere la lista L dall'inizio per andare a trovare quale è via via l'ultimo elemento, poi il penutlimo, poi il terzultimo, ecc. Per tenere traccia del punto in cui si è arrivati si può usare una variabile, *t*, che inizialmente farà riferimento all'ultimo elemento di L, poi al penultimo, ecc.

In pseudocodice:

```
Palindromo2(L)
1  if head[L] = NIL then
2    return true
3  t ← head[L]
4  while next[t] ≠ NIL do
5     t ← next[t]
6  x ← head[L]
7  while x ≠ t and next[t] ≠ x do
8    if key[x] ≠ key[t] then
9      return false
10   y ← x
11   while next[y] ≠ t then
12     y ← next[y]
13   t ← y
14   x ← next[x]
15 return true
```

In questo caso la complessità nel caso peggiore è $\Theta(n^2)$, in quanto per ogni iterazione del ciclo 7-14 occorre scorrere la lista in avanti un numero di volte proporzionale alla lunghezza *n* della lista (la prima volta compiendo *n* passi, la seconda compiendone *n*-2, la terza *n*-4 e così via). Dal fatto che il ciclo 7-14 viene eseguito *n*/2 volte risulta che la complessità dell'algoritmo è $\Theta(n^2)$.

Si noti che, potendo usare variabili globali, si potrebbe, pur rispettando il vincolo dal sul numero delle variabili, ottenere una complessità lineare, usando un algoritmo ricorsivo (di fatto in questo caso usando la pila delle chiamate ricorsive per allocare un numero qualsiasi di variabili).

In pseudocodice (chiamando *glob_x* la variabile globale suddetta; si noti che *glob_x* diventa NIL nel momento in cui si è finito di analizzare tutta la lista (più precisamente, *glob_x* diventa NIL alla **fine** della chiamata ricorsiva in cui il parametro *x* vale *head*[*L*])):

```
Palindromo3(L)
1  glob_x ← head[L]
2  return Palindromo3_ric(head[L])


Palindromo3_ric(x)
1  if x = NIL then
2    return true
3  if not Palindromo3_ric(next[x]) then
4    return false
5  if key[glob_x] ≠ key[x] then
6    return false
7  glob_x ← next[glob_x]
8  return true
```

In questo caso il numero di chiamate ricorsive è pari alla lunghezza *n* della lista *L*, ed ogni chiamata ricorsiva viene eseguita in tempo costante (è una catena di *if*), di conseguenza la complessità è $\Theta(n^2)$

Peraltro, si ricorda che, per come è stato definito al lezione, lo pseudocodice usato prevede solo variabili locali, non variabili globali, quindi questa sarebbe una soluzione "spuria", o comunque non totalmente conforme al modello di computazione usato.

R15. Esercizio

Si vuole progettare una struttura dati che rappresenti una coda di priorità, in cui però si vuole anche tenere traccia dell'ordine di inserimento degli elementi (indipendentemente dalle loro priorità).

Più precisamente, su tale struttura dati sono disponibili 2 operazioni di prelevamento di elementi: secondo la priorità (`PopByPriority`, che preleva l'elemento a più alta priorità), e secondo il tempo di arrivo (`PopFIFO`, che preleva, tra gli elementi della coda, quello che da più tempo è in coda, indipendentemente dalla sua priorità).

1. Si definisca la struttura dati.

2. Si definisca in pseudocodice **un** algoritmo (e uno solo) a scelta tra l'algoritmo di inserimento, l'algoritmo `PopByPriority`, e l'algoritmo `PopFIFO`.

3. Si dia la complessità dell'algoritmo definito.

NB: Nella descrizione della struttura dati si indichino gli attributi della struttura, e gli attributi degli eventuali oggetti che la compongono.

Soluzione

Una possibile maniera (non l'unica) di realizzare la struttura dati desiderata è per esempio combinando una coda FIFO (implementata mediante un array) ed una heap (anch'essa implementata mediante un array). Naturalmente l'implementazione mediante array ha lo svantaggio della limitatezza, per cui altre soluzioni (per esempio basate su una o più liste) si possono adottare.

In definitiva, una struttura dati `S` che combina una coda FIFO ed una heap potrebbe avere 2 attributi, `Q[L]` (la coda FIFO) e `H[L]` (la heap), che sono riferimenti a 2 array diversi ma di stessa dimensione (indicate come `size[Q[L]]` e `size[H[L]]`).

Gli elementi di questi 2 array sono degli oggetti `o` che hanno 3 attributi: `key[o]`, che rappresenta la priorità dell'elemento, `ind_q[o]`, che tiene traccia dell'indice associato all'elemento nella coda `Q`, e `ind_h[o]`, che tiene traccia dell'indice associato all'elemento nella heap `H`.

Con la struttura dati definita sopra, l'algoritmo di inserimento sarebbe il seguente (`p` è la priorità da inserire):

```
Insert(S,p)
1  crea oggetto o
2  key[o] ← p
3  Enqueue(Q[S], o)
4  Min-heap-insert(H[Q], o)
```

Gli algoritmi di `Enqueue` e di `Min-heap-insert` (e `Min-decrease-key`, su cui `Min-heap-insert` si basa) vanno leggermente adattati al particolare oggetto da inserire, nel modo seguente:

```
Enqueue(Q, o)
1  Q[tail[Q]] ← o
2  ind_q[o] ← tail[Q]
3  if tail[Q] = size[Q]
4    then tail[Q] ← 1
5    else tail[Q] ← tail[Q] + 1
```

```
Min-heap-insert (H, o)
1  size[H] ← size[H] + 1
2  p ← key[o]
3  key[o] ← +∞
4  H[size[H]] ← o
```

```
5  Heap-decrease-key(H, size[H], p)


Heap-decrease-key (H, i, key)
1  if key > H[i] then
2    error "new key bigger than existing one"
3  key[H[i]] ← key
4  ind_h[H[i]] ← i
5  while i > 1 and key[H[PARENT(i)]] > key[H[i]] do
6    swap H[i] ↔ H[PARENT(i)]
7    ind_h[H[i]] ← i
8    ind_h[H[PARENT(i)]] ← PARENT(i)
9    i ← PARENT(i)
```

Le modifiche non cambiano la struttura degli algoritmi, per cui le complessità rimangono quelle degli algoritmi originali, e cioè $\Theta(1)$ per `Enqueue` e $\Theta(\log(n))$ per `Min-heap-insert`.

L'algoritmo `PopFIFO` diventa il seguente:

```
PopFIFO(S)
1  x ← Dequeue(Q[S])
2  H ← H[S]
3  H[ind_h[x]] ← H[size[H]]
4  size[H] ← size[H] - 1
5  Min-heapify(H, ind_h[x])
6  return x
```

dove l'algoritmo `Dequeue` è lo stesso visto a lezione. Si noti come, alla riga 3, l'elemento che deve essere cancellato dalla heap viene sostituito con l'ultima foglia, che poi viene riposizionata al posto giusto mediante l'algoritmo `Min-heapify`, che deve essere ritoccato come segue:

```
Min-heapify (H, i)
1  l ← LEFT(i)
2  r ← RIGHT(i)
3  if l ≤ size[H] and key[H[l]] < key[H[i]] then
4    min ← l
5  else
6    min ← i
7  if r ≤ size[H] and key[H[r]] < key[H[min]] then
8    min ← r
9  if min ≠ i then
```

```
10   swap H[i] ↔ H[min]
11   ind_h[H[i]] ← i
12   ind_h[H[min]] ← min
13   Min-heapify(H, min)
```

Anche in questo caso gli algoritmi strutturalmente non cambiano, per cui i costi rimangono gli stessi di quelli visti a lezione. In definitiva, il costo di `PopFIFO` è dominato dal costo di `Min-heapify`, che è $\Theta(\log(n))$.

Da ultimo, l'algoritmo `PopByPriority` è il seguente.

```
PopByPriority (S)
1  x ← Heap-extract-min(H[S])
2  Q ← Q[S]
3  if tail[Q] < ind_q[x] then
4    for i ← ind_q[x] downto head[Q]
5      Q[i] ← Q[i-1]
6      ind_q[Q[i]] ← i
7    head[Q] ← head[Q] + 1
8  else
9    for i ← ind_q[x] to tail[Q]-1
10     Q[i] ← Q[i+1]
11     ind_q[Q[i]] ← i
12   tail[Q] ← tail[Q] - 1
```

dove l'algoritmo `Heap-extract-min` è esattamente uguale a quello visto in classe. Si noti che lo scopo dei cicli `for` delle linee 4-6 e 9-11 è di eliminare l'elemento che è stato cancellato dalla heap anche dalla coda FIFO, "coprendo il buco" creato dalla sua eliminazione (la condizione `tail[Q] < ind_q[x]` dice se l'array va fatto shiftare verso la testa, oppure verso la coda).

Si noti come in questo caso la complessità sia dominata dai cicli `for` suddetti, che nel caso peggiore hanno complessità $\Theta(n)$.

Si noti infine che se si usasse, almeno per la coda FIFO, una lista (bidirezionale) con un attributo `tail` che punta la coda della lista, invece che un array, si potrebbe ridurre la complessità dell'algoritmo `PopByPriority` a $\Theta(\log(n))$, cioè alla complessità di `Heap-extract-min`.

R16. Esercizio

4.1. Scrivere un algoritmo che, dato un array *S* di interi, ritorna `true` se è possibile partizionare *S* in **coppie** di elementi che hanno tutte lo stesso valore totale (inteso come la somma degli elementi della coppia).

Dare la complessità dell'algoritmo scritto.

4.2. Scrivere un algoritmo che, dato un array *S* di interi, ritorna `true` se è possibile partizionare *S* in **terne** di elementi che hanno tutte lo stesso valore totale (inteso come la somma degli elementi della terna). Si supponga pure che l'array *S* sia tale che, se una tale partizione esiste, per ogni elemento di *S* sia unica la terna la cui somma dei valori sia quella desiderata.

Dare la complessità dell'algoritmo scritto.

NB: Gli algoritmi possono essere descritti o in pseudocodice, o informalmente (in linguaggio naturale), purchè comunque la descrizione sia sufficientemente precisa per poter valutare la complessità dell'algoritmo.

Soluzione

1.

Nell'algoritmo seguente, l'array S viene prima ordinato, poi gli elementi in posizioni simmetriche rispetto alla metà dell'array (il primo e l'ultimo, il secondo e il penultimo, ecc.) vengono sommati e le somme confrontate. Se sono tutte uguali, l'array è partizionabile come desiderato, in caso contrario non lo è.

```
PartizioneInCoppie(S)
1  if lenght[S] mod 2 ≠ 0 then
2    return false
3  MERGE-SORT(S, 1, length[S])
4  i ← 2
5  j ← length[S]-1
6  while i<j do
7    if S[i] + S[j] ≠ S[i-1] + S[j+1] then
8      return false
9    i ← i+1
10   j ← j-1
11 return true
```

La complessità di questo algoritmo è, nel caso peggiore (quello in cui l'array è partizionalbile), Θ(n log(n)) in quanto è la complessità del passo di rdinamento quella dominante.

2.

Si noti che, perchè l'array sia partizionabile in terne come desiderato, il numero di elementi nell'array deve essere multiplo di 3. Inoltre, la somma delle terne deve essere la somma totale dei valori presenti nell'array diviso il numero di terne (cioè length[S]/3), e questo deve essere un numero intero. E' quindi facile determinare quale è la somma desiderata dei valori delle varie terne.

Quindi, nel caso di partizione in terne, una volta ordinato l'array, si procede dal primo i elemento a cercare una coppia di ulteriori elementi j e k tali che S[j]+S[k] = T – S[i], con T somma desiderata

dei valori della terna. questo può essere fatto con una sola passata dell'array S, uasando una versione modificata dell'algoritmo EXACT-SUM visto ad esercitazione.

```
PartizioneInTerne(S)
1  if length[S] mod 3 ≠ 0 then
2    return false
3  MERGE-SORT(S, 1, length[S])
4  T ← 0
5  for i ← 1 to length[S] do
6    T ← T + S[i]
7  if T mod (length[S]/3) ≠ 0 then
8    return false
9  T ← T / (length[S]/3)
10 i ← 1
11 while i ≤ length[S] do
12   trovato ← false
13   j ← i+1
14   k ← length[S]
15   while j < k and not trovato do
16     if S[j] ≠ ⊥ and S[j] ≠ ⊥ S[j] + S[k] = T- S[i] then
17       S[k] ← ⊥
18       S[j] ← ⊥
19     else if S[j] = ⊥ or S[j] + S[k] < T- S[i] then
20       j ← j+1
21     else
22       k ← k-1
23   if not trovato then
24     return false
25   else
26     i ← i+1
27     while i ≤ length[S] and S[i] = ⊥ do
28       i ← i+1
29 return true
```

L'idea dell'algoritmo è che, ogni volta che vengono trovati due valori S[j] e S[k] la cui somma con S[i] dà il valore desiderato T, gli elementi S[j] e S[k] vengono settati al valore convenzionale ⊥ (che, se l'array S è fatto di soli numeri naturali, potrebbe essere per esempio -1), per marcarli come “già usati”. Il resto dell'algoritmo deve solo fare attenzione a “saltare” gli elementi già usati (questo per esempio è lo scopo del ciclo alle righe 27-28.

L'ordinamento mediante MERGE-SORT ha naturalmente complessità $\Theta(n \log(n))$. Il resto dell'algoritmo, e soprattutto il ciclo 11-28 ha complessità $\Theta(n^2)$, in quanto sia il ciclo interno che quello esterno vengono eseguiti al massimo $\Theta(n)$ volte.

R17. Esercizio

Si supponga di avere un albero binario (non necessariamente di ricerca) in cui ogni nodo *N*, oltre ai soliti attributi *key*[*N*], *left*[*N*], *right*[*N*], *p*[*N*], ha anche altri 4 attributi, *depth_L*[*N*], *depth_R*[*N*], *depth_P*[*N*], e *depth_MAX*[*N*]. Questi 4 attributi sono inizialmente non inizializzati, e vanno calcolati dopo che l'albero è stato costruito interamente.

Più precisamente, *depth_L*[*N*] e *depth_R*[*N*] devono contenere la profondità, rispettivamente, del sottoalbero sinistro e del sottoalbero destro di N, incrementati di 1. Detto in altro modo, *depth_L*[*N*] (risp. *depth_R*[*N*]) contiene la distanza massima da *N* ad una foglia del suo sottoalbero sinistro (risp. destro).

depth_P[*N*], invece, deve contenere la distanza massima da *N* ad una foglia dell'albero, raggiungibile passando atteaverso il padre di *N*. *depth_MAX*[*N*] è il massimo di *depth_L*[*N*], *depth_R*[*N*], *depth_P*[*N*].

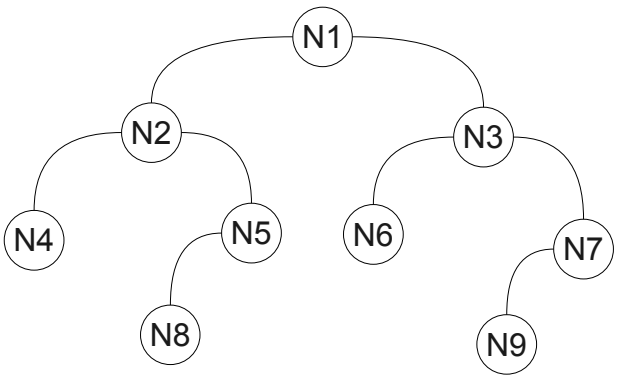
Per esempio, nell'albero disegnato sotto, per il nodo N3 i valori degli attributi devono essere i seguenti:

depth_L[N3] = 1 (che corrisponde alla distanza tra N3 e la foglia N6)

depth_R[N3] = 2 (che corrisponde alla distanza tra N3 e la foglia N9)

depth_P[N3] = 4 (che corrisponde alla distanza tra N3 e la foglia N8)

depth_MAX[N3] = 4



Per il nodo N1, invece, i valori sono i seguenti:

depth_L[N1] = 3, depth_R[N1] = 3, depth_P[N1] = 0, depth_MAX[N3] = 3

Si svolgano **2 a scelta** tra i seguenti punti.

1.

Si scriva un algoritmo SET_DEPTH_LR(*N*) che, dato il nodo radice *N* di un (sotto)albero binario in cui gli attributi *depth_L* e *depth_R* dei nodi non sono ancora inizializzati, per ogni nodo dell'albero di radice *N* calcola ed assegna i valori di *depth_L* e *depth_R*.

Si dia la complessità dell'algoritmo scritto.

2.

Si scriva un algoritmo SET_DEPTH_PMAX(*N, d*) che, dato il nodo radice *N* di un (sotto)albero binario in cui gli attributi *depth_L* e *depth_R* dei nodi sono **già stati** inizializzati, ma in cui gli attributi *depth_P* e *depth_MAX* sono ancora da inizializzare, e data la distanza massima *d* da *N* ad una foglia dell'albero raggiungibile passando attraverso il padre di *N*, per ogni nodo dell'albero di radice *N* calcola ed assegna i valori di *depth_P* e *depth_MAX*.

Si dia la complessità dell'algoritmo scritto.

3.

Si scriva un algoritmo SEARCH_MIN_DEPTH(*N*) che, dato il nodo radice *N* di un (sotto)albero binario in cui tutti gli attributi *depth_L*, *depth_R*, *depth_P* e *depth_MAX* dei nodi sono **già stati** inizializzati, ritorna il riferimento al nodo dell'albero il cui valore dell'attributo *depth_MAX* è minimo.

Si dia la complessità dell'algoritmo scritto.

Soluzione

Gli algoritmi desiderati sono i seguenti.

```
SET_DEPTH_LR(N)
1  if N = NIL then
2    return -1
3  depth_L[N] ← SET_DEPTH_LR(left[N])+1
4  depth_R[N] ← SET_DEPTH_LR(right[N])+1
5  return max(depth_L[N], depth_R[N])
```

```
SET_DEPTH_PMAX(N, d)
1  if N ≠ NIL then
2    depth_P[N] ← n
3    depth_MAX[N] ← max(depth_L[N], depth_R[N], depth_P[N])
4    SET_DEPTH_PMAX(left[N], max(n, depth_R[N])+1)
5    SET_DEPTH_PMAX(right[N], max(n, depth_L[N])+1)
```

```
SEARCH_MIN_DEPTH(N)
1  if N = NIL then
2    return NIL
3  l ← SEARCH_MIN_DEPTH(left[N])
4  r ← SEARCH_MIN_DEPTH(right[N])
5  min ← N
6  if (l ≠ NIL and depth_MAX[l] < depth_MAX[min]) then
7    min ← l
```

```
8  if (r ≠ NIL and depth_MAX[r] < depth_MAX[min]) then
9    min ← r
10 return min
```

La complessità di tutti e tre gli algoritmi è Θ(n) in quanto essi sono di fatto dei tree-walking.

R18. Esercizio

Sia dato un algoritmo `INSERT_BAL(r,x)` che, data la radice `r` di un albero binario di ricerca bilanciato ed un nuovo nodo `x` da inserire, aggiunge `x` all'albero di radice `r` mantenendo quest'ultimo bilanciato; l'algoritmo ritorna il puntatore alla radice dell'albero modificato. Dualmente, sia dato un algoritmo `DELETE_BAL(r,x)` che, data la radice `r` di un albero binario di ricerca bilanciato ed un suo nodo `x` da eliminare, cancella `x` dall'albero, mantenendo quest'ultimo bilanciato; l'algoritmo ritorna il puntatore alla radice dell'albero modificato. Entrambi gli algoritmi hanno complessità temporale O(log(n)), con `n` numero di nodi nell'albero.

Scrivere un algoritmo `NEW_ROOT(T,x)` che, dato un albero binario di ricerca `T` *bilanciato* ed un nodo `x` di `T`, modifica l'albero in modo che `x` diventi la sua radice, in modo però che i sottoalberi destro e sinistro della nuova radice dell'albero siano bilanciati (non è invece detto che `T` stesso rimanga bilanciato).

Dare la complessità temporale dell'algoritmo scritto nel caso pessimo.

NB1: Si ricorda che un albero `T` è bilanciato se per ogni nodo `r` di `T` vale la proprietà che l'altezza del sottoalbero sinistro di `r` e quella del sottoalbero destro di `r` differiscono al più di 1.

NB2: Il punteggio massimo verrà dato se l'algoritmo è descritto in pseudocodice e se non fa uso di strutture dati ausiliarie; tuttavia, verranno accettate anche soluzioni in cui strutture dati ausiliarie sono usate e/o in cui l'algoritmo è descritto in linguaggio naturale, purché comunque la descrizione sia sufficientemente precisa per poterne valutare la complessità temporale.

Soluzione

L'algoritmo funziona nel modo seguente.

Innanzi tutto, il nodo *x* viene messo alla radice dell'albero *T*, e i nodi diversi da x vengono "staccati" da *T*, ma mantenendo la struttura a puntatori, che ha come nodo radice *r*. In seguito, viene invocato l'algoritmo `BUILD_POSTORDER`, che altro non è che una visita in post-order dell'albero. Questo algoritmo, per ogni nodo *z* dell'albero (che è quello di radice *r* individuata sopra), prima visita il sottoalbero di sinistra, staccando via via tutti i nodi del sottoalbero, ed inserendoli nell'albero di radice *x*, quindi fa altrettanto con il sottoalbero destro di *z*. Dopo aver terminato la visita del sottoalbero destro, il nodo *z* è necessariamente una foglia (in quanto tutti i suoi nodi figli sono stati staccati), quindi può essere a sua volta staccato senza problemi, ed inserito nell'albero di radice *x*.

```
NEW_ROOT(T,x)
1 r ← DELETE_BAL(root[T],x)
```

```
2 root[T] ← x
3 p[x] ← NIL
4 left[x] ← NIL
5 right[x] ← NIL
6 BUILD_POSTORDER(x,r)
```

L'algoritmo `BUILD_POSTORDER` è il seguente:

```
BUILD_POSTORDER(x,z)
1  if z ≠ NIL
2    BUILD_POSTORDER(x,left[z])
3    BUILD_POSTORDER(x,right[z])
4    p[z] ← NIL
5    left[z] ← NIL
6    right[z] ← NIL
7    if key[z] < key[x] then
9      left[x] ← INSERT_BAL(left[x],z)
9    else
10     right[x] ← INSERT_BAL(right[x],z)
```

Ogni singolo inserimento di un nodo *z* nell'albero di radice *x* costa O(log(n)); il numero di inserimenti è un O(n), in quanto tutti gli elementi dell'albero originario devono essere messi nell'albero di radice *x*, quindi la complessità globale è O(n log(n)).

R19. Esercizio

Premessa: Diciamo che una funzione è sub-lineare se essa è O(n), ma non è Ω(n).

1.

Scrivere un algoritmo con complessità temporale nel caso pessimo sub-lineare che, dato un array *A* **ordinato** di numeri interi **con ripetizioni**, e dato un numero *k* da cercare in *A*, restituisce l'indice corrispondente **all'ultima** occorrenza di *k* in *A* (restituisce -1 se *k* non esiste in *A*)

Dare la complessità dell'algoritmo scritto, mostrando appunto che è sub-lineare.

2.

Scrivere un algoritmo che, dato un array A **ordinato** di numeri interi **con ripetizioni**, restituisce il valore dell'intero presente nell'array che ha il numero massimo di ripetizioni.

Si dia la complessità temporale dell'algoritmo scritto nel caso pessimo sotto l'ipotesi che nell'array non ci siano due numeri diversi con lo stesso numero di ripetizioni.

NB: Il punteggio massimo verrà dato se l'algoritmo è descritto in pseudocodice; tuttavia, verranno accettate anche soluzioni in cui l'algoritmo è descritto in linguaggio naturale, purché comunque la descrizione sia sufficientemente precisa per poterne valutare la complessità temporale.

Soluzione

Parte 1.

Per risolvere il problema dato ci possiamo appoggiare al seguente algoritmo, che, dato un array *A*, un valore *k* da cercare in esso, e gli estremi *p* ed *r* del sottoarray di *A* in cui cercare *k*, ritorna l'indice del primo numero più grosso di *k* in *A*. L'algoritmo è essenzialmente un `binary-search` modificato nella condizione di arresto.

```
SEARCH_GT(A,k,p,r)
1  if (p>r) then
2    return -1
3  q ← ⌊(p+r)/2⌋
4  if ( (q<length[A] and A[q]=k and A[q+1]>k) or
5       (q=length[A] and A[q]=k)) then
6    return q
7  if (A[q]>k) then
8    SEARCH_GT(A,k,p,q-1)
9  else
10   SEARCH_GT(A,k,q+1,r)
```

La complessità temporal di questo algorimo è la stessa di `binary-search`, quindi è O(log(n)), che è sub-lineare.

L'algoritmo desiderato si risolve quindi semplicemente nella chiamata `SEARCH_GT(A,k,1,length[A])`.

Parte 2.

Per risolevere il problema desiderato possiamo appoggiarci all'algoritmo sviluppato al punto 1, nel modo seguente.

```
MAX_INT(A)
1  v ← A[1]
2  max ← SEARCH_GT(A,A[1],1,length[A])
3  i ← max+1
4  while(i ≤ length[A]) do
5    n ← SEARCH_GT(A,A[i],i,length[A])
6    if (n-i > max) then
7      v = A[i]
```

```
8       max = n-1
9     i = n+1
```

La complessità temporale di questo secondo algoritmo è O(v log(n)), con *n* numero di elementi totali dell'array, e *v* numero di valori *unici* (cioè non contando le ripetizioni) dentro all'array.

Poiché si sa che non ci sono in *A* 2 elementi diversi con lo stesso numero di ripetizioni, il massimo nmero di elementi unici si ha quando $n = 1 + 2 + 3 + \ldots + v$ cioè il massimo numero *v* possibile è tale che $n = (v+1)v/2$, cioè nel caso pessimo $n = \Theta(v^2)$, e $v = \Theta(n^{1/2})$.

Di conseguenza, la complessità dell'algoritmo in questione è $O(n^{1/2} \log(n))$.

Esercizi non risolti

NR1. Esercizio

Scrivere un algoritmo che prende in ingresso gli elementi di un array, e crea un albero binario di ricerca (BST) contenente gli stessi elementi dell'array.

Quale è la complessità dell'algoritmo scritto?

Nota: è possibile riutilizzare (sotto)algoritmi noti per la realizzazione dell'algoritmo dell'esercizio.

NR2. Esercizio

Sia data una struttura dati ad albero binario, in cui l'elemento contenuto in ogni nodo dell'albero è a sua volta un albero binario.

Scrivere un algoritmo che visita *tutti* i nodi di *tutti* gli alberi memorizzati nella struttura dati (si lascia la scelta della tecnica di visita: pre/post o inorder).

Quale è la complessità dell'algoritmo scritto?

Nota: è possibile riutilizzare (sotto)algoritmi noti per la realizzazione dell'algoritmo dell'esercizio.

NR3. Esercizio

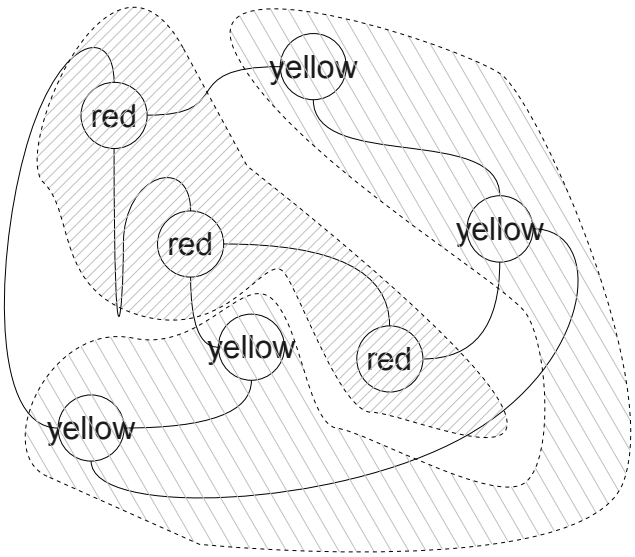
Sia dato un grafo *non orientato* G in cui ogni nodo N ha un attributo *color*[N] che ne indica il colore, e che può assumere solo o il valore *yellow*, o il valore *red*. Si scriva un algoritmo che, dato un grafo siffatto, ritorna *true* se e solo se è possibile partizionare il grafo in 2 sottografi connessi R e Y, in modo che il sottografo R contenga *tutti e soli* i nodi *red* di G, e il sottografo Y contenga tutti e soli i nodi *yellow* di G.

Si dia la complessità dell'algoritmo scritto.

NB1: si ricorda che un grafo è connesso se e solo se esiste almeno un cammino tra ogni coppia di suoi nodi.

NB2: E' preferibile che l'algoritmo venga descritto in pseudocodice; tuttavia, verranno accettate anche soluzioni che descrivano solo in modo informale (cioè in italiano) l'algoritmo, purché queste siano sufficientemente precise per poter valutare la complessità dell'algoritmo.

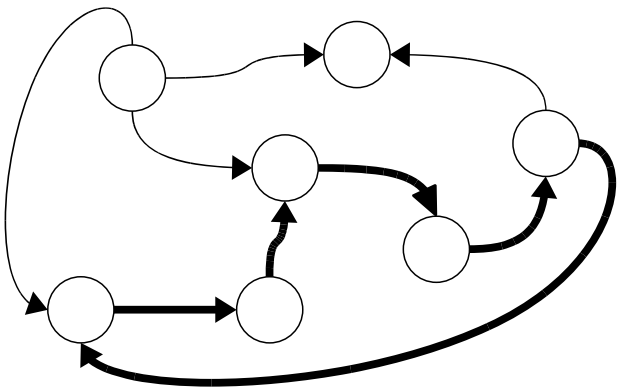
Esempio di grafo su cui l'algoritmo deve ritornare *true*:



NR4. Esercizio

Scrivere un algoritmo che, dato un grafo diretto *DG*, ritorna `true` se *DG* è un grafo **aciclico**, `false` altrimenti. Si dia la complessità dell'algoritmo nel caso peggiore, indicando anche quale sia il caso peggiore.

Si ricorda che un grafo si dice ciclico se esiste in esso almeno un cammino che inizia e finisce sullo stesso nodo, come per esempio nel grafo seguente (in cui il ciclo è evidenziato da archi più spessi).



NR5. Esercizio

1. Scrivere in pseudocodice un algoritmo che, dati due alberi binari di ricerca *A* e *B* contenenti numeri interi, ritorna `true` se gli alberi contengono gli stessi elementi, `false` altrimenti.

Si supponga pure che i due alberi non contengano elementi ripetuti più di una volta.

2. Indicare quale è il caso **peggiore** di funzionamento dell'algoritmo e darne la sua complessità.

3. Indicare quale è il caso **migliore** di funzionamento dell'algoritmo e darne la sua complessità.

NR6. Esercizio

Un "puzzle lineare" è una sequenza di tessere in cui ognuna di esse (tranne la prima e l'ultima) deve incastrarsi sia con la tessera che la precede (quella "a sinistra") che con quella che la segue (quella "a destra"). Naturalmente la prima (risp. l'ultima) tessera si incastra solo con quella che la segue (risp. precede).

Il numero di profili possibili per le tessere è arbitrario; i possibili profili sono rappresentati mediante numeri interi: un profilo rappresentato da un numero positivo si incastra con uno rappresentato dal suo opposto. I numeri positivi sono sempre sul lato destro di una tessera, quelli negativi sul sinistro). I bordi (sia il sinistro che il destro) sono rappresentati dal numero 0.

Un esempio di puzzle è il seguente:

<0, 12> <-12, 4> <-4, 54> <-54, 0>

Ogni tessera sia rappresentata da un oggetto con 2 attributi, rappresentanti i 2 lati della tessera: `left[T]` e `right[T]`. Si supponga che in ogni sequenza di tessere un numero ed il suo opposto possano comparire al massimo una sola volta.

1. Scrivere un algoritmo che, data una sequenza arbitraria di tessere, la riordina in modo da comporre il puzzle (se ciò è possibile).

L'algoritmo può essere scritto in pseudocodice, oppure può essere descritto in modo informale a parole, purché tale descrizione sia sufficientemente precisa per capire il funzionamento dell'algoritmo.

2. Indicare quale è, nel caso peggiore, la complessità dell'algoritmo scritto.

3. Indicare come cambia (se cambia) la complessità dell'algoritmo se per rappresentare i profili delle tessere si usano, invece che numeri interi, le lettere dell'alfabeto (usando la convenzione che una lettera minuscola si "incastra" con la corrispondente lettera maiuscola e che i bordi sono rappresentati dal carattere "#", come per esempio nel seguente puzzle: #g Gs Sm Mf Fl L#).

NR7. Esercizio

Scrivere un algoritmo che, data una matrice quadrata M, ne riordina gli elementi in modo che:

1. gli elementi che si trovano sulla stessa colonna prima del riordino si troveranno sulla stessa colonna anche dopo il riordino (anche se la colonna può cambiare di posizione);
2. dopo il riordino, le colonne al loro interno sono ordinate in ordine crescente;
3. gli elementi sulla diagonale principale sono ordinati in ordine crescente.

Si dia la complessità dell'algoritmo scritto.

NB1: Si noti che, data una matrice, diverse sono le possibili matrici riordinate che rispettano i vincoli di cui sopra. L'algoritmo deve produrre **una** di queste matrici, non importa quale.

Per esempio, data la matrice

15	22	88
30	40	5
20	6	28

le due seguenti matrici rispettano i vincoli di cui sopra

6	5	15
22	28	20
40	88	30

5	15	6
28	20	22
88	30	40

NB2: l'algoritmo può essere descritto o in pseudocodice, o informalmente (in linguaggio naturale), purchè comunque la descrizione sia sufficientemente precisa per poter valutare la complessità dell'algoritmo.

NR8. Esercizio

L'implementazione di heap mediante array ha l'inconveniente di limitare alla dimensione dell'array il numero massimo di elementi che possono essere contenuti nella heap.

Proporre una soluzione alternativa per implementare una min-heap, che non ponga limiti al numero massimo di elementi che si possono memorizzare in essa.

Spiegare (anche solo in modo informale, senza fare uso di pseudocodice, ma in modo sufficientemente preciso) come su questa struttura dati vengono realizzati gli algoritmi che operano sulla min-heap, ed in particolar modo quelli di decremento della chiave di un elemento, di inserimento di un nuovo elemento nella min-heap e di estrazione (e cancellazione) del minimo dalla heap.

Dire come cambia, se cambia, la complessità di questi algoritmi rispetto alle versioni che operano su min-heap implementata mediante array.

NR9. Esercizio

Scrivere in pseudocodice un algoritmo `TreeInterval` che, dato un albero binario di ricerca T contenente elementi le cui chiavi sono valori interi, e gli estremi L, U (con $L \leq U$) di un intervallo di numeri interi, stampa a video gli elementi di T che appartengono all'intervallo $[L,U]$.

Si supponga che l'albero T non contenga valori ripetuti, e che ci sia in T un elemento con chiave uguale a L (mentre non è detto che in T ci sia un elemento con chiave uguale a U).

Supponendo che l'albero T sia *bilanciato* (cioè tale per cui per ogni nodo r di T vale la proprietà che l'altezza del sottoalbero sinistro di r e quella del sottoalbero destro di r differiscono al più di 1), dare la complessità temporale nel caso pessimo dell'algoritmo scritto, in funzione del numero n di nodi dell'albero e della lunghezza l dell'intervallo $[L,U]$ (si noti che $l = U-L+1$).

NB: Il punteggio dato sarà tanto più alto quanto più efficiente sarà l'algoritmo.

NR10.Esercizio

Scrivere un algoritmo che prende in ingresso un array di numeri naturali, e lo modifica in modo da separare i numeri pari dai numeri dispari, mettendo prima i numeri pari in ordine crescente, seguiti dai numeri dispari in ordine descrescente.

Dare la complessità dell'algoritmo scritto.

NR11.Esercizio

Si vogliono scrivere algoritmi per alberi binari di ricerca *quasi completi*, realizzati mediante array. Più precisamente, se A è l'array che rappresenta l'albero, $A[1]$ è l'elemento radice, $A[2]$, $A[3]$ sono i suoi figli sinistro e destro rispettivamente, $A[4]$ e $A[5]$ i figli sinistro e destro di $A[2]$, e così via. L'attributo $size[A]$ indica il numero di elementi nell'albero, e $A[size[A]]$ è la foglia più a destra sull'ultimo livello dell'albero (quello quasi completo).

Si scrivano i seguenti algoritmi per alberi binari di ricerca quasi completi implementati mediante array, indicando, per ogni algoritmo, la sua complessità.

a. Un algoritmo per ritornare, dato un albero A, il suo elemento minimo.

b. Un algoritmo che, dato un albero A e l'indice h di un suo elemento, calcola la profondità dell'elemento $A[h]$.

c. Un algoritmo per calcolare, dato un albero A e l'indice h di un suo elemento, il massimo del sottoalbero di radice A[h].

NB: il punteggio massimo verrà dato se gli algoritmi saranno a complessità minima tra quelli che risolvono i problemi desiderati.

NR12.Esercizio

Scrivere un algoritmo che, data in ingresso una sequenza di numeri interi, la riordina in modo da mettere per primo il numero più grande, per secondo il più piccolo, per terzo il secondo più grande, per quarto il secondo più piccolo, e così via.

Si dia la complessità dell'algoritmo scritto.

Per esempio, data in ingresso la sequenza `[-4, 8, -23, 10, 17, 0, 3]`, l'algoritmo la deve riordinare nella seguente maniera: `[17, -23, 10, -4, 8, 0, 3]`.

NB: Il punteggio massimo verrà dato se l'algoritmo sarà a complessità minima tra quelli che risolvono il problema desiderato, e se l'algoritmo è scritto ex-novo, senza invocare (né reimplementare) algoritmi visti a lezione.

NR13.Esercizio

Scrivere in pseudocodice un algoritmo TreeSearchUpper che, dato un albero binario di ricerca T contenente elementi le cui chiavi sono valori interi ed un valore V, restituisce il nodo di T che ha la

chiave più piccola, tra quelle presenti in *T*, che è maggiore o uguale a *V*; se un tale elemento non esiste, l'algoritmo restituisce *NIL*.

Si dia la complessità temporale dell'algoritmo nel caso pessimo, sia nel caso generale (in cui *T* ha una forma qualunque), sia nel caso in cui *T* sia bilanciato (cioè tale per cui per ogni nodo *r* di *T* vale la proprietà che l'altezza del sottoalbero sinistro di *r* e quella del sottoalbero destro di *r* differiscono al più di 1).

NB: Il punteggio dato sarà tanto più alto quanto più efficiente sarà l'algoritmo.