Dati e Algoritmi (Pietracaprina)

Esercizi sulle Nozioni di Base

Problema 1. *Specificare come problema computazionale* Π *la verifica se due insieme finiti di oggetti da un universo* U *sono disgiunti oppure no.*

Soluzione. Il problema dato può essere formalmente specificato così:

$\mathcal{I} = \{(X, Y) \ : \ X, Y \subseteq U\}$;

$\mathcal{S} = \{\text{yes,no}\}$;

Π = insieme delle coppie $((X, Y), s)$ con $(X, Y) \in \mathcal{I}$ e $s \in \mathcal{S}$ tali che:

- Se $X \cap Y = \emptyset$ allora $s = \text{yes}$
- Se $X \cap Y \neq \emptyset$ allora $s = \text{no}$.

□

Problema 2. *Specificare come problema computazionale* Π *la ricerca dell'inizio e della lunghezza del più lungo segmento di* 1 *consecutivi in una stringa binaria.*

Soluzione. Il problema dato può essere formalmente specificato così:

$\mathcal{I}$ = stringhe binarie;

$\mathcal{S}$ = coppie di interi;

Π = insieme di coppie $(X, (j, k))$, con $X \in \mathcal{I}$ e $(j, k) \in \mathcal{S}$, tali che:

- Se X contiene almeno un 1, allora il segmento più lungo di 1 consecutivi in X è $X[j \div j + k - 1]$
- Se X non contiene 1 (vuota o tutti 0), allora $j = -1$ e $k = 0$.

□

Problema 3. (Esercizio R-4.16 del testo [GTG14]) *Dimostrare che se* $d(n) \in O(f(n))$ *e* $e(n) \in O(g(n))$*, non è detto che valga che* $d(n) - e(n) \in O(f(n) - g(n))$.

Soluzione. Basta prendere: $d(n) = 3n$, $e(n) = 2n$, $f(n) = n + 3$, e $g(n) = n$. □

Problema 4. *Siano* A *e* B *due array di* n *ed* m *interi, rispettivamente. Scrivere un algoritmo in pseudocodice per calcolare il seguente valore:*

$$v = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} A[i] \cdot B[j].$$

Soluzione. L'algoritmo è il seguente:

```
Algoritmo sumOfProducts(A,B)
input Array A e B di n ed m interi, rispettivamente
output Somma di tutti i prodotti A[i]*B[j], ∀ 0 ≤ i ≤ n-1 e 0 ≤ j ≤ m-1

v ⟵ 0
for i ⟵ 0 to n-1 do {
  for j ⟵ 0 to m-1 do v ⟵ v+A[i]*B[j]
}
return v
```

Un algoritmo più efficiente per risolvere lo stesso problema è il seguente:

```
Algoritmo sumOfProducts2(A,B)
input Array A e B di n ed m interi, rispettivamente
output Somma di tutti i prodotti A[i]*B[j], ∀ 0 ≤ i ≤ n-1 e 0 ≤ j ≤ m-1

vA ⟵ 0; vB ⟵ 0;
for i ⟵ 0 to n-1 do vA ⟵ vA+A[i]
for i ⟵ 0 to m-1 do vB ⟵ vB+B[i]
return vA*vB
```

□

Problema 5. *Sia A un array di n interi distinti, ordinato in senso crescente, e x un valore intero. Scrivere due algoritmi in pseudocodice che implementano la ricerca binaria per trovare x in A. Entrambi gli algoritmi restituiscono l'indice i tale che $A[i] = x$, se x appare in A, e -1 altrimenti. Il primo algoritmo deve essere iterativo e usare un solo ciclo, e il secondo algoritmo deve essere ricorsivo.*

Soluzione. La versione iterativa è la seguente:

```
Algoritmo BinarySearchIT(A,x)
input Array ordinato A di n interi distinti, intero x
output r ∈ [0,n-1] tale che A[r]=x, se x ∈ A, -1, altrimenti

n ⟵ A.length; s ⟵ 0; t ⟵ n-1
while (s≤t) do {
  r ⟵ ⌊(s+t)/2⌋
  if (A[r]=x) then return r
  if (x < A[r]) then t ⟵ r-1 else s ⟵ r+1
}
return -1
```

Anche se non richiesto dall'esercizio accenniamo all'analisi dell'algoritmo. La correttezza dell'algoritmo è basata sul seguente invariante che vale alla fine di ciascuna iterazione del ciclo while: per $0 \leq i < s$ si ha che $A[i] < x$, mentre per $t < i < n$ si ha che $A[i] > x$; quindi se $x \in A$ deve per forza essere che $x = A[j]$ per un qualche $s \leq j \leq t$. La prova della correttezza utilizzando l'invariante è lasciata come esercizio. Per quanto riguarda la complessità, si osserva che ad ogni iterazione del while la taglia dell'intervallo di indici delimitato da s e t si riduce e di almeno la metà. Dato che inizialmente la taglia dell'intervallo è n, saranno eseguite $O(\log n)$ iterazioni del while, ciascuna delle quali richiede un numero costante di operazioni. La complessità è quindi $O(\log n)$.

La versione ricorsiva dell'algoritmo prende come parametri di input, oltre ad A e x, anche due indici s, t che delimitano la porzione di array entro la quale fare la ricerca. L'algoritmo è il seguente:

```
Algoritmo BinarySearchREC(A,x,s,t)
input Array ordinato A di interi distinti, intero x, e indici s e t
output r ∈ [s,t] tale che A[r]=x, se x ∈ A[s...t], altrimenti -1

if (t<s) then return -1
r ⟵ ⌊(s+t)/2⌋
if (A[r]=x) then return r
if (x < A[r]) then return BinarySearchREC(A,x,s,r-1)
else return BinarySearchReEC(A,x,r+1,t)
```

Anche se non richiesto dall'esercizio, analizziamo la complessità di `BinarySearchREC` usando l'albero della ricorsione. Usiamo come taglia dell'istanza il numero n di elementi di A tra i quali cercare x. In una invocazione generica dell'algoritmo,

$$n = \max\{0, t - s + 1\}.$$

Consideriamo l'albero della ricorsione associato all'esecuzione di `BinarySearchREC` per una istanza di taglia n. Si vede facilmente che l'albero è costituito da una catena di $O(\log n)$ nodi corrispondenti alle diverse invocazioni ricorsive fatte su istanze di taglia $n_i \leq n/2^i$, per $i \geq 0$, dove l'istanza iniziale ha taglia $n_0 = n$. Ciascuna invocazione ricorsiva, ovvero ciascun nodo dell'albero, contribuisce un numero costante di operazioni, escludendo le operazioni fatte dalle invocazioni ricorsive al suo interno. Se ne deduce quindi che la complessità è $O(\log n)$. □

Problema 6. (Esercizio R-4.12 del testo [GTG14])

Soluzione. Nel codice dato ci sono tre cicli for innestati, ciascuno dei quali è costituito da al più n iterazioni. Ogni ciclo esegue un numero costante di operazioni al di là degli eventuali cicli for al suo interno. Di conseguenza, il numero totale di operazioni eseguite sarà $\leq c_1 + n(c_2 + n(c_3 + nc_4)) \leq cn$, per delle opportune costanti $c_1, c_2, c_3, c_4, c > 0$. Quindi la complessità al caso pessimo è $O\left(n^3\right)$. Anche se non richiesto dall'esercizio, è facile osservare che ciascuna iterazione del ciclo esterno esegue un numero di operazioni $\geq c' \sum_{i=1}^{n} i$ per un'opportuna costante $c' > 0$. Quindi la complessità è anche $\Omega\left(n^3\right)$, ovvero $\Theta\left(n^3\right)$. □

Problema 7. *Sia A un algoritmo che ricevuto in input un array X di n interi esegue*

- *$c_1 n$ operazioni per ogni intero pari in X*
- *$c_2 \lceil \log_2 n \rceil$ operazioni per ogni intero dispari in X,*

dove c_1 e c_2 sono costanti positive. Analizzare la complessità in tempo dell'algoritmo esprimendola con $\Theta(\cdot)$.

Soluzione. Per l'analisi asintotica consideriamo valori di n sufficientemente grandi da garantire che $c_1 n \geq c_2 \lceil \log_2 n \rceil$. La complessità è:

- $O\left(n^2\right)$: in quanto per ogni intero in X si eseguono $\leq c_1 n$ operazioni;
- $\Omega(n \log n)$: in quanto per ogni intero in X si eseguono $\geq c_2 \lceil \log_2 n \rceil$ operazioni (questo lower bound vale quindi per ogni istanza);
- $\Omega\left(n^2\right)$: in quanto se tutti i valori in X sono pari, per ogni intero in X si eseguono $c_1 n$ operazioni (questo lower bound vale quindi per un'istanza specifica).

Se ne deduce immediatamente che la complessità in tempo di A è $\Theta\left(n^2\right)$. □

Problema 8. *Il seguente pseudocodice descrive l'algoritmo di ordinamento chiamato* InsertionSort*. (Si assume che la sequenza S da ordinare sia una variabile globale che dopo la fine dell'algoritmo è accessibile e ordinata.)*

```
Algoritmo InsertionSort(S)
input Sequenza S[0 ... n-1] di n chiavi
output Sequenza S ordinata in senso crescente

for i ⟵ 1 to n-1 do {
  curr ⟵ S[i]
  j ⟵ i-1
  while ((j ≥ 0) AND (S[j]> curr)) do {
    S[j+1] ⟵ S[j]
    j ⟵ j-1
  }
  S[j+1] ⟵ curr
}
```

a. *Trovare delle opportune funzioni $f_1(n)$, $f_2(n)$ e $f_3(n)$ tali che le seguenti affermazioni siano vere, per una qualche costante $c > 0$ e per n abbastanza grande.*

   - *Per ciascuna istanza di taglia n l'algoritmo esegue $\leq cf_1(n)$ operazioni.*
   - *Per ciascuna istanza di taglia n l'algoritmo esegue $\geq cf_2(n)$ operazioni.*
   - *Esiste una istanza di taglia n per la quale l'algoritmo esegue $\geq cf_3(n)$ operazioni.*

   La funzione $f_1(n)$ deve essere la più piccola possibile, mentre le funzioni $f_2(n)$ e $f_3(n)$ devono essere le più grandi possibili.

b. *Sia $t_{\mathrm{IS}}(n)$ la complessità al caso pessimo dell'algoritmo. Sfruttando le affermazioni del punto precedente trovare un upper bound $O(\cdot)$ e un lower bound $\Omega(\cdot)$ per $t_{\mathrm{IS}}(n)$.*

Soluzione.

a. Le funzioni sono le seguenti

   - $f_1(n) = n^2$. Infatti, per ciascuna istanza di taglia n l'algoritmo esegue $n-1$ iterazioni del ciclo for e, per ogni iterazione del for, al più n iterazioni del ciclo while le quali richiedono un numero costante di operazioni ciascuna.
   - $f_2(n) = n$. Infatti, per ciascuna istanza di taglia n l'algoritmo esegue $n-1$ iterazioni del ciclo for.
   - $f_3(n) = n^2$. Infatti si consideri una sequenza S ordinata in senso decrescente. Nell'iterazione i del for l'algoritmo esegue i iterazioni del ciclo while le quali

richiedono un numero costante di operazioni ciascuna. Quindi il numero totale di operazioni eseguite dall'algoritmo per ordinare S sarà proporzionale a

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

b. Dal punto precedente si deduce immediatamente che $t_{\text{IS}}(n) = O\left(n^2\right)$ e $t_{\text{IS}}(n) = \Omega\left(n^2\right)$, ovvero $t_{\text{IS}}(n) = \Theta\left(n^2\right)$.

□

Problema 9. (Esercizio C-4.35 del testo [GTG14]) *Dimostrare la seguente proprietà per induzione:*

$$Q(n): \quad \sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \in \Theta\left(n^3\right) \quad \forall n \geq 0.$$

Soluzione. Per l'induzione scegliamo $n_0 = 0$ e $k = 0$.

Base: Per $n = 0$ la relazione è banalmente verificata in quanto entrambi i membri dell'equazione sono 0.
Passo induttivo: Sia $n \geq 1 = n_0 + k$ e assumiamo $Q(m)$ sia vera per ogni $0 \leq m \leq n$. Dimostriamo che $Q(n+1)$ vale.

$$\begin{aligned}
\sum_{i=0}^{n+1} i^2 &= (n+1)^2 + \sum_{i=0}^{n} i^2 \\
&= (n+1)^2 + \frac{n(n+1)(2n+1)}{6} \qquad \text{(per hp induttiva)} \\
&= \frac{6(n+1)^2 + n(n+1)(2n+1)}{6} \\
&= \frac{(n+1)(n+2)(2(n+1)+1)}{6}.
\end{aligned}$$

□

Problema 10. *Dimostrare che*

$$F(n) = \frac{1}{\sqrt{5}}\left(\Phi^n - \hat{\Phi}^n\right) \quad \forall n \geq 0,$$

dove $\Phi = \frac{1+\sqrt{5}}{2}$ *(*golden ratio*) e* $\hat{\Phi} = \frac{1-\sqrt{5}}{2}$.

Soluzione. Per l'induzione scegliamo $n_0 = 0$ e $k = 1$.

Base: Per $n = 0, 1$ la relazione è banalmente verificata dalla definizione di $F(n)$, che pone $F(0) = 0$ e $F(1) = 1$, e dal fatto che

$$\frac{1}{\sqrt{5}}\left(\Phi^0 - \hat{\Phi}^0\right) = 0 \quad \text{e} \quad \frac{1}{\sqrt{5}}\left(\Phi^1 - \hat{\Phi}^1\right) = 1,$$

come dimostrabile da semplici passaggi algebrici.
Passo induttivo: Sia $n \geq 1$ e assumiamo la relazione per $F(m)$ vera per ogni $0 \leq m \leq n$. Dimostriamo che vale per $F(n+1)$. Abbiamo che

$$\begin{aligned} F(n+1) &= F(n) + F(n-1) \quad \text{(per definizione)} \\ &= \frac{1}{\sqrt{5}}\left(\Phi^n + \Phi^{n-1} - \left(\hat{\Phi}^n + \hat{\Phi}^{n-1}\right)\right) \quad \text{(per hp. induttiva)} \end{aligned}$$

Si osservi che

$$\Phi^n + \Phi^{n-1} = \Phi^{n-1}(\Phi + 1) = \Phi^{n-1}\left(\frac{1+\sqrt{5}}{2} + 1\right) = \Phi^{n-1}\frac{3+\sqrt{5}}{2}.$$

E dato che

$$\frac{3+\sqrt{5}}{2} = \left(\frac{1+\sqrt{5}}{2}\right)^2,$$

concludiamo che

$$\Phi^n + \Phi^{n-1} = \Phi^{n-1} \cdot \Phi^2 = \Phi^{n+1}.$$

Analogamente si dimostra che $\hat{\Phi}^n + \hat{\Phi}^{n-1} = \hat{\Phi}^{n+1}$. Di conseguenza,

$$F(n+1) = \frac{1}{\sqrt{5}}\left(\Phi^{n+1} - \hat{\Phi}^{n+1}\right).$$

□

Problema 11. *Sia $S = S[1], S[2], \ldots, S[n]$ una sequenza di n bit. Il seguente algoritmo determina la lunghezza del più lungo segmento continuo di 1:*

```
Algoritmo maxSegment1(S)
input Sequenza S di n bit
output Lunghezza del piu' lungo segmento continuo di 1 in S

max ⟵ 0; curr ⟵ 0;
for i ⟵ 1 to n do {
  if (S[i]=1) then {
    curr ⟵ curr+1
    if (curr > max) then max ⟵ curr
  }
  else curr ⟵ 0
}
return max
```

Dimostrare la correttezza del ciclo (ovvero dell'algoritmo) tramite un'opportuno invariante.

Soluzione. La proprietà $\mathcal{L}$ che vogliamo raggiungere alla fine del ciclo è che il valore di `max` sia effettivamente la lunghezza del più lungo segmento continuo di 1 in S. A tale scopo, dimostriamo che il seguente invariante vale alla fine dell'iterazione i, per ogni $0 \leq i \leq n$:

- `max` è la lunghezza del più lungo segmento continuo di 1 in $S[1 \ldots i]$;
- `curr` è la lunghezza del più lungo suffisso continuo di 1 in $S[1 \ldots i]$.

All'inizio del ciclo ($i = 0$), l'inizializzazione delle variabili `max` e `curr` e il fatto che l'intervallo $[1 \ldots i]$ è vuoto rende vero l'invariante. Supponiamo che esso valga alla fine della iterazione $i-1$. Se $S[i] = 1$ allora il valore di `curr` viene correttamente incrementato di 1, mentre quello di `max` viene aggiornato solo se la lunghezza del più lungo suffisso continuo di 1 in $S[1 \ldots i]$, cioè il valore aggiornato di `curr`, è maggiore del valore attuale di `max` che rappresenta la lunghezza del più lungo segmento continuo di 1 in $S[1 \ldots i-1]$. Se invece $S[i] = 0$ allora il valore di `curr` viene correttamente azzerato mentre quello di `max` rimane invariato. In ogni caso, si evince che l'invariante vale anche alla fine dell'iterazione i. Alla fine dell'ultima iterazione ($i = n$), l'invariante implica immediatamente che `max` è la lunghezza del più lungo segmento continuo di 1 in $S[1 \ldots n] = S$, e quindi $\mathcal{L}$ è verificata. □

Problema 12. *Sia A una matrice $n \times n$ di interi, con $n \geq 2$, dove righe e colonne sono numerate a partire da 0.*

a. *Sviluppare un algoritmo iterativo che restituisce l'indice $j \geq 0$ di una colonna di A con tutti 0, se tale colonna esiste, altrimenti restituisce -1. L'algoritmo deve contenere un solo ciclo.*

b. *Trovare un upper bound e un lower bound alla complessità dell'algoritmo sviluppato.*

c. *Provare la correttezza dell'algoritmo sviluppato, scrivendo un opportuno invariante per il ciclo su cui esso si basa.*

Soluzione.

a. L'algoritmo è il seguente:

```
Algoritmo allZeroColumn(A)
input Matrice A n × n
output j ≥ -1:  se j ≥ 0 allora la colonna j di A ha tutti 0; se j=-1
A non ha colonne con tutti 0

i ⟵ 0; j ⟵ 0;
while (j < n) do {
  if (i=n) then return j
  if (A[i,j]=0) then i ⟵ i+1
  else {
    j ⟵ j+1
    i ⟵ 0
  }
}
return -1
```

b. Ogni iterazione del ciclo while esegue un numero costante di operazioni e tocca una entry distinta di A. La complessità è quindi $O\left(n^2\right)$. Se la matrice A ha l'ultima riga fatta da entry diverse da 0 mentre tutte le altre entry sono 0 si vede facilmente che l'algoritmo in questo caso tocca tutte le entry di A. Quindi la sua complessità sarà anche $\Omega\left(n^2\right)$, e quindi $\Theta\left(n^2\right)$.

c. Per quanto riguarda la correttezza, la proprietà $\mathcal{L}$ che vogliamo raggiungere alla fine del ciclo è che il valore di restituito sia l'indice di una colonna di tutti 0, se tale colonna esiste, oppure -1 se A non ha colonne di tutti 0. Il seguente invariante vale alla fine di ciascuna iterazione del while:

- Nessuna delle colonne di indice $0, 1, \ldots j-1$ è costituita da tutti 0.
- $A[\ell, j] = 0$ per ogni $0 \leq \ell < i$.

All'inizio del ciclo, l'inizializzazione di i e j implica che gli intervalli $[0, j-1]$ e $[0, i-1]$ sono vuoti e quindi l'invariante vale per vacuità. Supponiamo che sia vero all'inizio di una certa iterazione in cui l'algoritmo deve guardare la entry $A[i, j]$ e dimostriamo che rimane vero alla fine della iterazione. Se $i = n$ gli indici i e j non vengono modificati

e quindi l'invariante rimane vero. Inoltre, in questo caso il ciclo termina restituendo l'indice j in output che, grazie alla seconda proprietà dell'invariante è l'output corretto. Se invece $i < n$ e $A[i, j] = 0$ allora è vero che $A[\ell, j] = 0$ per ogni $0 \leq \ell < i + 1$, e quindi l'incremento di i mantiene vero l'invariante. Se infine $i < n$ e $A[i, j] \neq 0$, allora significa che anche la colonna j non contiene tutti 0, e l'incremento di j mantiene vero l'invariante. Consideriamo l'ultima iterazione. Se si esce dal ciclo perchè $i = n$, la seconda proprietà dell'invariante garantisce che l'output è corretto. Se si esce perchè $j = n$ la prima proprietà dell'invariante garantisce che non ci sono colonne di tutti 0 in A e, anche in questo caso, l'output è corretto. Quindi alla fine del ciclo vale la proprietà $\mathcal{L}$.

□

Problema 13. *Sia A una matrice binaria $n \times n$ per la quale vale la proprietà che in ciascuna riga gli 1 vengono prima degli 0. Si supponga anche che il numero di 1 nella riga i sia $\leq$ del numero di 1 nella riga $i + 1$, per $i = 0, 1, \ldots, n - 2$. Descrivere un algoritmo che in tempo $O(n)$ conti il numero di 1 in A, e provarne la correttezza. L'algoritmo deve contenere un solo ciclo.*

Soluzione. L'idea è di seguire il profilo degli 1 più a destra di ciascuna riga, muovendosi sempre o a destra o in basso, e accumulando il numero di 1 in ciascuna riga prima di abbandonarla. L'algoritmo è il seguente (con Q si denota la proprietà che in ciascuna riga gli 1 vengono prima degli 0, e che il numero di 1 nella riga i è $\leq$ del numero di 1 nella riga $i + 1$, per $i = 0, 1, \ldots, n - 2$):

Algoritmo `Count(A)`
input `Matrice A n × n di 0/1 in cui vale Q`
output `Numero di 1 in A`

```
i ⟵ 0; j ⟵ 0; count ⟵ 0;
while (i < n) do {
  if (j < n) and (A[i,j]=1) then j ⟵ j+1
  else {
    count ⟵ count+j
    i ⟵ i+1
  }
}
return count
```

La complessità dell'algoritmo è $O(n)$ in quanto:

- in ciascuna iterazione del **while** si esegue un numero costante di operazioni;
- le iterazioni del **while** sono al più $2n$ dato che in ciascuna iterazione uno dei due indici i o j cresce, gli indici non decrescono mai e nessuno dei due può crescere più di n volte.

In effetti la complessità dell'algoritmo è $\Theta(n)$ in quanto, per qualsiasi istanza l'indice i deve crescere n volte, e quindi il numero di iterazioni del while è almeno n. Analizziamo la correttezza dell'algoritmo. All'inizio del ciclo vale che $i = j =$ `count` $= 0$. La proprietà $\mathcal{L}$ che vogliamo raggiungere alla fine del ciclo è che il valore di `count` sia effettivamente il numero di 1 in A. La correttezza dell'algoritmo discende dal seguente invariante che vale alla fine di ciascuna iterazione del while:

- `count` è il numero di 1 nelle righe di indice compreso tra 0 e $i-1$;
- Se $i < n$, allora $A[i, \ell] = 1$ per $0 \leq \ell \leq j-1$

La verifica che l'invariante è vero all'inizio del ciclo, che viene mantenuto da un'iterazione alla successiva, e che alla fine del ciclo implica la proprietà $\mathcal{L}$, è lasciata come esercizio.

Un algoritmo alternativo, la cui analisi è lasciata come esercizio, è il seguente.

```
Algoritmo Count1(A)
input Matrice A n × n di 0/1 in cui vale Q
output Numero di 1 in A

i ⟵ 0; j ⟵ 0; count ⟵ 0;
while ((i < n) and (j < n)) do {
  if (A[i,j]=1) then {
    count ⟵ count+n-i
    j ⟵ j+1
  }
  else i ⟵ i+1
}
return count
```

□

Problema 14. *Si consideri l'algoritmo* `reverseArray` *presentato a lezione, il cui pseudocodice è il seguente:*

Algoritmo `reverseArray`(A, i, j)
input `array` A, `indici` $i, j \geq 0$
output `array` A `con gli elementi in` $A[i \ldots j]$ `ribaltati`

if $(i < j)$ **then** {
  `swap(`$A[i]$`,`$A[j]$`)`
  `reverseArray(`A`,`$i+1$`,`$j-1$`)`
}
return A

Per la generica invocazione `reverseArray`(A, i, j)*, si consideri come taglia dell'istanza la lunghezza del segmento da ribaltare, ovvero* $n = \max\{0, j - i + 1\}$.

a. *Dimostrare che* `reverseArray` *è corretto.*

b. *Analizzare la complessità di* `reverseArray` *tramite l'albero della ricorsione.*

Soluzione.

a. Dimostriamo la correttezza per induzione su n.
Base: Scegliamo come base $n = 0, 1$, ovvero $i \geq j$. In questi casi non c'è alcuna operazione da fare e l'algoritmo correttamente restituisce lo stesso array ricevuto in input.
Passo induttivo: Fissiamo $n \geq 1$ e assumiamo che l'algoritmo ribalti correttamente segmenti di lunghezza al più n (hp.induttiva). Consideriamo adesso l'algoritmo applicato a un segmento di array $A[i \ldots j]$ con $j - i + 1 = n + 1 > 1$. Dopo aver scambiato gli elementi $A[i]$ e $A[j]$, l'algoritmo ribalta correttamente (per ipotesi induttiva) il segmento $A[i + 1 \ldots j - 1]$, e la correttezza segue immediatamente.

b. Una generica invocazione di `reverseArray` per ribaltare un segmento di lunghezza n, esegue $\Theta(1)$ operazioni e un'eventuale chiamata ricorsiva su un segmento di lunghezza $n - 2$ (se $n > 1$). È facile allora vedere che per una qualsiasi istanza di taglia $n > 1$ l'albero della ricorsione ha $\lfloor n/2 \rfloor + 1$ nodi ciascuno dei quali contribuisce $\Theta(1)$ operazioni al numero di operazioni totale per quell'istanza. Di conseguenza la complessità dell'algoritmo è $\Theta(n)$.

□

Problema 15. *Dimostrare la correttezza dell'algoritmo* `power` *presentato a lezione, che calcola la potenza n-esima di un numero reale* x.

Soluzione. Dimostriamo la correttezza per induzione su n.
Base: Scegliamo come base $n = 0$. In questo caso l'algoritmo restituisce correttamente 1.
Passo induttivo: Fissiamo $n \geq 0$ e assumiamo (hp induttiva) che l'algoritmo calcoli correttamente x^m per ogni $m \in [0, n]$ e ogni reale x. Consideriamo l'esecuzione di `power`$(x, n+1)$, per un qualche reale x. Se $n+1$ è dispari, l'ipotesi induttiva assicura che $y = x^{n/2}$ e l'algoritmo restituisce correttamente $x \cdot y^2 = x \cdot x^{2n/2} = x^{n+1}$. Se invece n è pari, l'ipotesi induttiva assicura che $y = x^{(n+1)/2}$ e l'algoritmo restituisce correttamente $y^2 = x^{2(n+1)/2} = x^{n+1}$. □

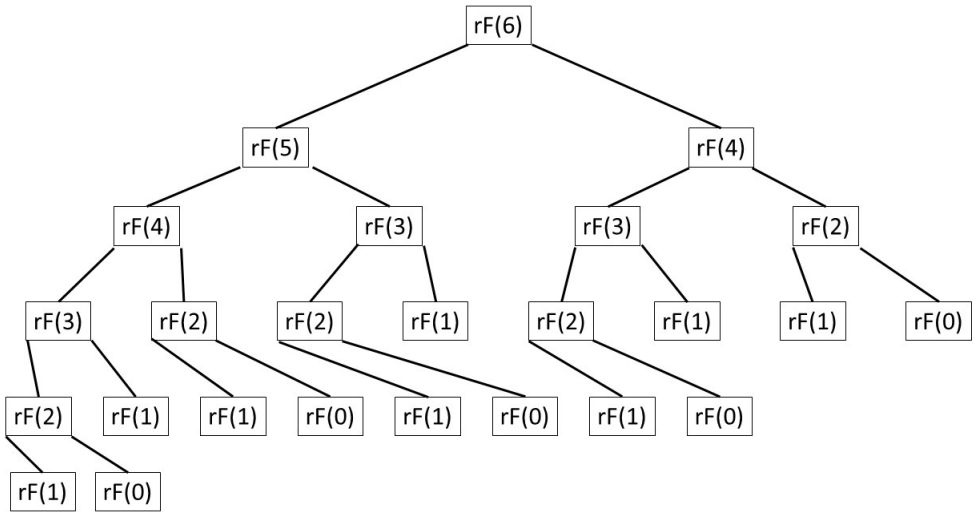
Problema 16. *Si consideri l'algoritmo ricorsivo* `recurFib` *per il calcolo dell'n-esimo numero di Fibonacci $F(n)$, il cui pseudocodice è il seguente:*

Algoritmo `recurFib`(n)
input `intero` $n \geq 0$
output $F(n)$

if $n \leq 1$ **then return** n
return `recurFib`$(n-1)$`+recurFib`$(n-2)$

Disegnare l'albero della ricorsione per $n = 6$.

Soluzione. L'albero della ricorsione per $n = 6$ è illustrato nella seguente figura.



L'esempi evidenzia l'inefficienza dell'algoritmo dovuta alle numerose invocazioni ricorsive fatte con lo stesso valore di input (ad es., `recurFib`(1) è invocato 8 volte!). Come semplice esercizio,

dimostrare per induzione su $n \geq 0$ che `recurFib`(n) esegue *almeno* $F(n)$ operazioni, che implica che la complessità è esponenziale in n. □

Problema 17. (Esercizio C-5.10 del testo [GTG14]) *Il problema della* element uniqueness*, definito a pagina 162 del testo* [GTG14]*, chiede che dato un array A di n elementi si determini se tali elementi sono tutti distinti.*

a. *Progettare un algoritmo ricorsivo efficiente per risolvere questo problema, senza fare ricorso all'ordinamento.*

b. *Analizzare la complessità $t_{\text{EU}}(n)$ dell'algoritmo usando l'albero della ricorsione.*

Soluzione.

a. L'algoritmo è basato sulla seguente idea. Se $n = 1$ chiaramente la risposta al problema è true. Se $n > 1$ allora prima si verifica che $A[0]$ sia diverso da ciascun $A[k]$, con $0 < k < n$. Se ciò non è vero, la risposta al problema è false, altrimenti la risposta è ottenuta risolvendo il problema per il suffisso $A[1 \ldots n-1]$. Lo pseudocodice è il seguente (la prima invocazione sarà fatta con $i = 0$):

```
Algoritmo EU(A,i)
input Array A di n ≥ 1 elementi, indice i, con 0 ≤ i < n
output true/false se gli elementi in A[i,n-1] sono/non sono tutti distinti

n ⟵ |A|
if (i ≥ n) then return true
k ⟵ i+1
while (k < n) do {
  if (A[k]=A[i]) then return false
  else k ⟵ k+1
}
return EU(A,i+1)
```

b. Consideriamo una generica istanza data da un array A di taglia n. L'albero della ricorsione associato a tale istanza sarà costituito da n nodi, corrispondenti alle invocazioni `EU`(A, i) con $0 \leq i < n$. Il costo attribuito al nodo corrispondente all'invocazione `EU`(A, i) è rappresentato dalle operazioni eseguite da tale invocazione, escluse quelle della (eventuale) chiamata ricorsiva al suo interno. Esso è proporzionale al numero di iterazioni del while (che sono al più $n - i - 1$), dato che in ciascuna iterazione si fa un numero costante di operazioni, ed è quindi $O(n - i)$. Ne consegue che il costo complessivo di tutta l'istanza è $O\left(\sum_{i=1}^{n}(n - i)\right) = O\left(n^2\right)$. Dato che questo argomento si applica a

qualsiasi istanza di taglia n, abbiamo che $t_{\text{EU}}(n) \in O\left(n^2\right)$. È facile vedere che se A ha tutti elementi distinti, l'algoritmo esegue $\Omega\left(n^2\right)$ operazioni, quindi la complessità risulta essere $\Theta\left(n^2\right)$.

□

Problema 18. (Esercizio C-5.20 del testo [GTG14]) *Si consideri il seguente algoritmo* `RicSum` *che somma n interi memorizzati in un array A, dove n è una potenza di 2. Se $n = 1$ allora l'algoritmo restituisce $A[0]$, altrimenti crea un array B di $n/2$ interi con $B[i] = A[2i] + A[2i+1]$, per $0 \leq i < n/2$, e restituisce la somma degli interi in B calcolata ricorsivamente. Descrivere* `RicSum` *tramite pseudocodice analizzandone la complessità tramite l'albero della ricorsione.*

Soluzione. Lo pseudocodice è il seguente:

```
Algoritmo RicSum(A,n)
input Array A[0 ... n-1] di n ≥ 1 interi (n potenza di 2)
output Somma degli interi in A

if (n=1) then return A[0]
B ⟵ array vuoto
for i ⟵ 0 to n/2-1 do {
  B[i] ⟵ A[2i]+A[2i+1]
}
return RicSum(B,n/2)
```

Per quanto riguarda la complessità, l'albero della ricorsione associato a un'istanza (ovvero un array A) di taglia $n = 2^d$ ha $d + 1$ nodi, corrispondenti a invocazioni su istanze di taglia $n/2^i$, per $0 \leq i \leq d$. Il costo attribuito al nodo corrispondente all'invocazione su un'istanza di taglia $n/2^i$ è principalmente determinato dalle $n/2^{i+1}$ iterazioni del ciclo for, ed è quindi $\Theta\left(n/2^i\right)$. Ne consegue che il costo complessivo di tutta l'istanza è $\Theta\left(\sum_{i=1}^{d} n/2^i\right) = \Theta(n)$. Dato che questo argomento si applica a ogni istanza di taglia n, abbiamo che la complessità è $\Theta(n)$. □

Problema 19. *Sia A un array di n interi arbitrari. Progettare e analizzare un algoritmo che trova, se esiste, l'intero che occorre in A almeno $\lfloor n/2 \rfloor + 1$ volte* (maggioranza assoluta). *L'algoritmo deve avere complessità $O(n)$.*

- *Versione 1: assumere A ordinato (facile).*
- *Versione 2: nessuna assunzione su A (difficile).*

Soluzione. Si assuma $n > 1$, altrimenti la soluzione è banale. Per la Versione 1 è sufficiente una semplice scansione di A che conta quante volte occorre ciascun intero e tiene traccia del massimo numero di occorrenze. Lo pseudocodice è il seguente:

```
Algoritmo Majority(A)
input Array A di n > 1 interi, ordinato
output m maggioranza in A, se esiste, null, altrimenti.

n ⟵ A.length; m ⟵ A[0]; count ⟵ 1;
for i ⟵ 1 to n-1 do {
  if (A[i]=A[i-1]) then {
    count ⟵ count +1;
    if (count = ⌊n/2⌋ + 1) then return m
  }
  else { m ⟵ A[i]; count ⟵ 1 }
}
return null
```

L'analisi di complessità e correttezza è lasciata come semplice esercizio. Una alternativa più efficiente, sempre nel caso di A ordinato, è basata sulla seguente osservazione. Se A contiene un elemento m di maggioranza, allora deve per forza valere che $A[\lfloor n/2 \rfloor] = m$. Infatti, se non fosse vero questo si avrebbe che le occorrenze di m sarebbero tutte a sinistra o tutte a destra di $A[\lfloor n/2 \rfloor]$ e quindi non potrebbero essere $\geq \lfloor n/2 \rfloor + 1$ in numero. In base a questa osservazione, una volta identificato m, posso contare il numero delle sue occorrenze con una semplice scansione di A (non migliorando però la complessità precedente), oppure determinare l'indice i della prima e l'indice j dell'ultima occorrenza di m, restituendo true se $j - i + 1 \geq \lfloor n/2 \rfloor + 1$. La complessità sarebbe $O(\log n)$. I dettagli della specifica e dell'analisi dell'algoritmo sono lasciati come esercizio.

La Versione 2 dell'algoritmo si basa su un'idea semplice ed elegante presentata nel 1981 da Boyer e Moore. Si fa una scansione di A mantenendo un intero `m` candidato a essere la maggioranza e un contatore che tiene un conteggio parziale delle occorrenze di `m`, escludendone un numero pari al numero di interi diversi da `m` incontrati sino a quel momento. Alla fine della scansione, se esiste un intero di maggioranza, `m` è tale intero. La verifica però richiede una seconda scansione di A. Lo pseudocodice è il seguente:

```
Algoritmo Majority(A)
input Array A di n > 1 interi
output m maggioranza in A, se esiste, null, altrimenti.

n ⟵ A.length; m ⟵ A[0]; count ⟵ 1;
for i ⟵ 1 to n-1 do {
```

```
  if (count = 0) then { m ⟵ A[i]; count ⟵ 1 }
  else {
    if (A[i]=m) then count ⟵ count+1
    else count ⟵ count-1
  }
}
count ⟵ 0;
for i ⟵ 0 to n-1 do {
  if (A[i]=m) then count ⟵ count+1
}
if (count > ⌊n/2⌋ + 1) then return m
else return null
```

È immediato vedere che l'algoritmo ha complessità $\Theta(n)$. Per quanto riguarda la correttezza è sufficiente dimostrare che alla fine del primo ciclo for vale la seguente proprietà $\mathcal{L}$: `m` è l'unico intero che può essere la maggioranza (ovvero tutti gli altri interi in A sicuramente non sono interi di maggioranza). Alla fine di ogni interazione $i \geq 0$ del for vale il seguente invariante:

$A[0 \ldots i]$ contiene

- `count` occorrenze di `m` e `count` è ≥ 0;
- $(i + 1 - \texttt{count})/2$ coppie (x, y) con $x \neq y$. (Si noti che alcune di queste coppie possono contenere `m`.)

La dimostrazione che l'invariante vale alla fine di ciascuna iterazione è lasciato come semplice esercizio. Supponiamo che esso valga alla fine del ciclo (cioè alla fine della iterazione $n - 1$). Dall'invariante e dal fatto che `count` è sempre ≥ 0, si deduce immediatamente che qualsiasi intero $x \neq \texttt{m}$ ha al più $(n - \texttt{count})/2 \leq n/2$ occorrenze in A e non può quindi essere l'intero di maggioranza. □