Dati e Algoritmi I (Pietracaprina)

Esercizi sulle Nozioni di Base

Problema 1. *Sia T una stringa arbitraria di lunghezza $n \geq 1$ su un alfabeto Σ. È sempre possibile scrivere T come concatenazione di n/m copie di una stringa P (ovvero $T = PP\cdots P$ n/m volte) dove P ha lunghezza $m \leq n$ ed n/m è intero. La* periodicità *di T è il minimo valore m per cui tale scrittura di T è possibile. Ad esempio, $T =$ `abcd` ha periodicità 4 mentre $T =$ `abababab` ha periodicità 2. Definire come* problema computazionale *il problema di trovare la periodicità di una stringa T specificando l'insieme delle istanze $\mathcal{I}$, l'insieme delle soluzioni $\mathcal{S}$ e il sottoinsieme di $\Pi \subseteq \mathcal{I} \times \mathcal{S}$ che rappresenta il problema.*

Soluzione. Si usi Σ^+ per denotare l'insieme delle stringhe sull'alfabeto Σ di lunghezza maggiore o uguale a 1, e Z^+ per denotare l'insieme degli interi positivi. Il problema dato può essere formalmente specificato così:

$$\begin{aligned}
\mathcal{I} &= \Sigma^+; \\
\mathcal{S} &= Z^+; \\
\Pi &= \{(T,m) : T \in \mathcal{I}, m \in \mathcal{S}, |T| = n, \text{ e } m \text{ è il min intero t.c.} \\
&\qquad T = PP\cdots P, \ n/m \text{ volte, con } P = T[0 \ldots m-1]\}
\end{aligned}$$

□

Problema 2. (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 3. *Siano A e B due array di n interi ciascuno. Scrivere un algoritmo in pseudocodice per calcolare un array C di n interi, tale che*

$$C[i] = A[i] \cdot B[i].$$

per ogni $0 \leq i \leq n-1$.

Soluzione. L'algoritmo è il seguente:

Algoritmo `ArrayProduct(A,B)`
input `Array A e B di n interi`
output `Array C di n interi con C[i]=A[i]*B[i],` $\forall$ `0` $\leq$ `i` $\leq$ `n-1`

for `i` $\longleftarrow$ `0` **to** `n-1` **do** `C[i]` $\longleftarrow$ `A[i]*B[i]`
return `C`

□

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
```

□

Problema 5. *Sia A un array di interi distinti, ordinato in maniera crescente, ed x un valore intero. Scrivere un algoritmo in pseudocodice che restituisca l'indice di x in A, se x appare in A, e -1 altrimenti (l'algoritmo deve implementare la ricerca binaria).*

Soluzione. Scriviamo la versione ricorsiva dell'algoritmo che 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 BinarySearch(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 BinarySearch(A,x,s,r-1)
else return BinarySearch(A,x,r+1,t)
```

Anche se non richiesto dall'esercizio, analizziamo la complessità di `BinarySearch` 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 `BinarySearch` 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. *Il seguente pseudocodice descrive l'algoritmo di ordinamento chiamato* InsertionSort

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 $t_{\text{IS}}(n)$ 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 $t_{\text{IS}}(n)$.*

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 8. (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 9. *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. Alla fine del ciclo vogliamo che il valore della variabile `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 $S[1 \ldots i]$.

L'invariante vale all'inizio del ciclo ($i = 0$) per vacuità, dato che $S[1 \ldots 0]$ non contiene elementi. 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 $S[1 \dots i]$, 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 $S[1 \dots i-1]$. 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 garantisce che `max` rappresenta la lunghezza del più lungo segmento continuo di 1 in $S[1 \dots n] = S$, e quindi l'algoritmo risulta corretto. □

Problema 10. *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 Allzeroscolumn(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. Si consideri un'arbitraria iterazione del ciclo while. All'inizio di tale iterazione (ovvero alla fine della precedente) vale il seguente invariante:

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

L'invariante è vero all'inizio del ciclo 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 $i = n$, come osservato sopra 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.

□

Problema 11. *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 maggiore o uguale al 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, partendo dall'angolo in alto a destra di A, muovendosi sempre o a sinistra o in basso, e accumulando il numero di 1 in ciascuna riga prima di abbandonarla. L'algoritmo (che chiamiamo `Count(A)`) è il seguente (con Q si denota la proprietà che in cuascina riga gli 1 vengno prima degli 0, e che il numero di 1 nella riga i è maggiore o uguale al 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 ⟵ n-1; count ⟵ 0;
while (i < n) do {
  if (j ≥ 0) and (A[i,j]=0) then j ⟵ j-1
  else {
    count ⟵ count+j+1
    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 decresce j (se ≥ 0) oppure cresce i e, inoltre, j non cresce mai e può descrescere al più n volte, mentre i non descresce mai e cresce esattamente 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.

La correttezza dell'algoritmo discende dal seguente invariante che vale all'inizio di ciascuna iterazione del while (ovvero alla fine della precedente):

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

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 ne assicura la correttezza, è lasciata come esercizio. □

Problema 12. *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 $\lceil (n + 1)/2 \rceil$ nodi ciascuno dei quali contribuisce $\Theta(1)$ operazioni al numero di operazioni totale per quell'istanza. Di conseguenza la complessità dell'algoritmo è $\Theta(n)$.

□

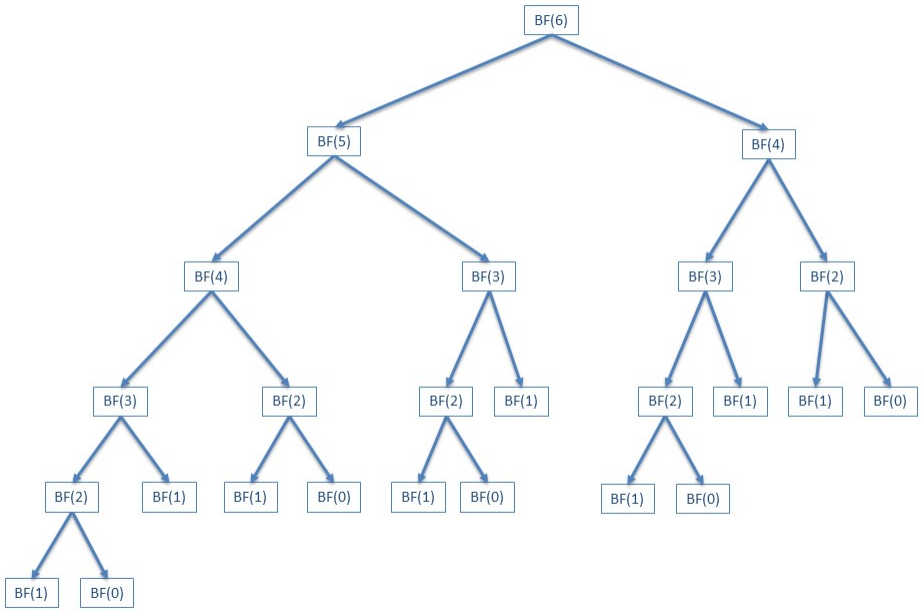
Problema 13. *Si consideri l'algoritmo* `BinaryFib` *presentato a lezione per il calcolo dell'n-esimo numero di Fibonacci* $F(n)$*, il cui pseudocodice è il seguente:*

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

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

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

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



□

Problema 14. (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. Assumiamo che l'output sia 'yes' o 'no' a seconda che gli elementi siano o meno tutti distinti. Se $n = 1$ chiaramente la risposta al problema è 'yes'. Se $n > 1$ allora prima si verifica che $A[1]$ sia diverso da ciascun $A[k]$, con $1 < k \leq n$. Se ciò non è vero, la risposta al problema è 'no', altrimenti la risposta è ottenuta risolvendo il problema per il sottoarray $A[2 \ldots n]$. Lo pseudocodice è il seguente (la prima invocazione sarà fatta con $i = 1$):

```
Algoritmo ElementUniqueness(A,i,n)
input Array A di n ≥ 1 elementi, indice i, con 1 ≤ i ≤ n
output 'yes'/'no' se gli elementi in A[i,n] sono/non sono tutti distinti

if (i ≥ n) then return 'yes'
for k ⟵ i+1 to n do {
  if (A[k]=A[i]) then return 'no'
}
return ElementUniqueness(A,i+1,n)
```

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 `ElementUniqueness`(A, i, n) con $1 \leq i \leq n$. Il costo attribuito al nodo corrispondente all'invocazione `ElementUniqueness`(A, i, n) è rappresentato dalle operazioni eseguite da tale invocazione, escluse quelle della (eventuale) chiamata ricorsiva al suo interno. Esso è principalmente determinato dalle $n - i$ iterazioni del ciclo for, ed è quindi $\Theta(n - i)$. Ne consegue che il costo complessivo di tutta l'istanza è $\Theta\left(\sum_{i=1}^{n}(n - i)\right) = \Theta\left(n^2\right)$. Dato che questo argomento si applica a qualsiasi istanza di taglia n, abbiamo che $t_{\mathrm{EU}}(n) \in \Theta\left(n^2\right)$.

□

Problema 15. (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 $\Theta(n)$. □