Esercizio

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.)

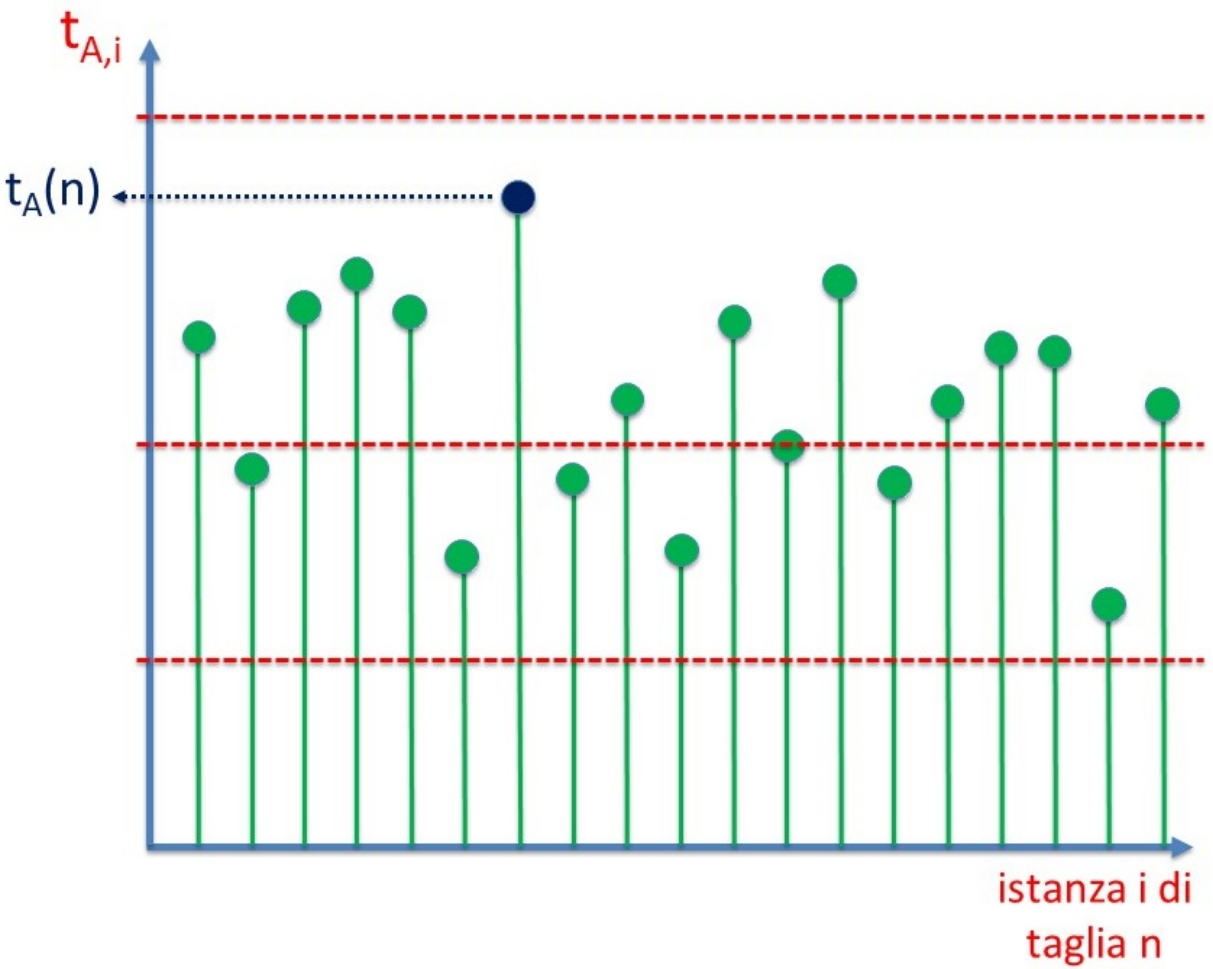
Algoritmo `InsertionSort(S)`
input `Sequenza S[0 ... n-1] di n chiavi`
output `Sequenza S ordinata in senso crescente`

```
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
}
```

Esercizio (continua)

1. 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.
2. 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 $t_{\mathrm{IS}}(n)$.



$cf_1(n)$

$cf_3(n)$

$cf_2(n)$

$f_1(n)$: n^2 perché

* $n-1$ iterazioni del for
* in ciascuna iterazione del for
  - $\Theta(1)$ operazioni fuori dal while
  - $< n$ iterazioni del while, ciascuna con $\Theta(1)$ operazioni

$\Rightarrow \leq cn^2$ operazioni $\forall$ istanza di taglia n

$\Rightarrow t_{IS}(n) \in \mathcal{O}(n^2)$

$f_2(n)$: n perché per ogni istanza di taglia n

* n-1 iterazioni del for
* $\Omega(1)$ operazioni in ogni iterazione del for

$\Rightarrow \geq cn$ operazioni in totale

$\Rightarrow t_{is}(n) \in \Omega(n)$

$f_3(n) : n^2$ perché scegliendo una istanza

Sordinate in modo decrescente si ha

* $n-1$ iterazioni del for
* nella iterazione i del for $(i = 1 .. n-1)$

  si eseguono i iterazioni del while con

  $\Theta(1)$ operazioni ciascuna, e $\Theta(1)$ altre

  operazioni

$\Rightarrow$ il numero totale di operazioni è

$$\geq c \cdot \sum_{i=1}^{n-1} i \geq c' \cdot n^2$$

$\Rightarrow t_{IS}(n) \in \Omega(n^2)$

$\Rightarrow t_{IS}(n) \in O(n^2)$ (da $f_1(n)$)

$t_{IS}(n) \in \Omega(n^2)$ (da $f_3(n)$)

$\Rightarrow t_{IS}(n) \in \Theta(n^2)$

Esercizio

Sia $S[1 \div n]$ 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 più lungo segmento continuo di 1

```
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 (e quindi dell'algoritmo) tramite un opportuno invariante.

Esempio: $S \equiv 0\,1\,1\,0\,1\,0\,1\,1\,0\,0\,\underbrace{1\,1\,1\,1}_{4}\,0 \Rightarrow output = 4$

PROPRIETÀ $\mathcal{L}$: max = lunghezza massima di un segmento continuo di 1 in S

INVARIANTE : Alla fine delle iterazioni $i \geq 0$

* max = lunghezza massima di un segmento continuo di 1 in $S[1 \div i]$
* cur = lunghezza massima di un SUFFISSO continuo di 1 in $S[1 \div i]$

Dimostriamo che l'invariante è mantenuta in ciascuna iterazione

(1) All'inizio ($i = 0$) vale per vacuità $S[1 \div 0] = \beta$

(2) Supponiamo che valga alla fine della iterazione i

$\Rightarrow$ max = massimo |segmenti di 1| in $S[1 \div i]$

cum = massimo |suffissi di 1| in $S[1 \div i]$

Nella iterazione $i+1$

$$S[i+1] = \begin{cases} 0 \Rightarrow \text{cum} \leftarrow 0; \text{ max invariato} \\ 1 \Rightarrow \text{cum} \leftarrow \text{cum} + 1 \text{ e} \\ \qquad \text{max} \leftarrow \text{massimo}\{\text{max}, \text{cum}\} \end{cases}$$

Si vede facilmente che l'invariante continua a valere alla fine della iterazione $i+1$

(3) Se l'invariante vale alla fine dell'ultima iterazione $(i = n)$ allora

max = lunghezza massima di un segmento continuo di 1 in $S[1 \div n] \equiv S$

$\Rightarrow$ vale $\mathcal{L}$

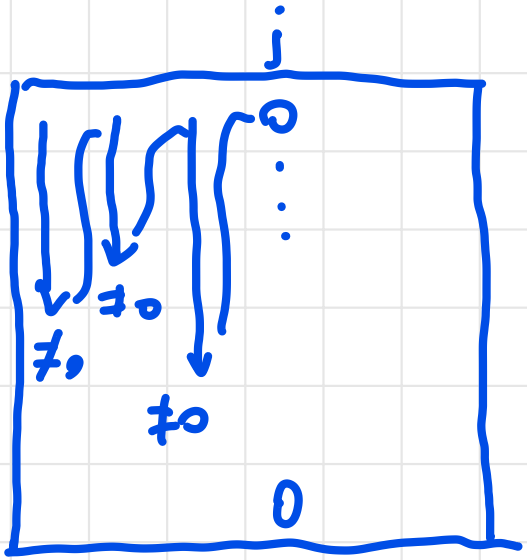
Esercizio

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

1. Sviluppare un algoritmo 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.
2. Trovare un upper bound e un lower bound alla complessità dell'algoritmo sviluppato.
3. Provare la correttezza dell'algoritmo sviluppato, scrivendo un opportuno invariante per il ciclo su cui esso si basa.

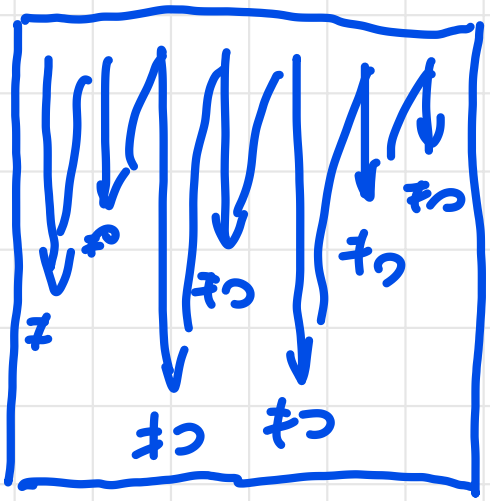
Idea

CASO CON COLONNA $\equiv 0$



return j

CASO SENZA COLONNA $\equiv 0$



return -1

Algoritmo AllZeroColumn(A)

input matrice A $n \times n$ di interi

output Indice $j \geq 0$ se la colonna $j \equiv 0$, -1 altrimenti

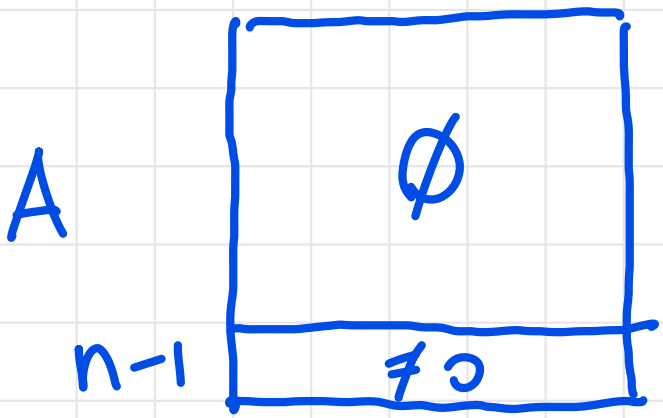
```
i←0; j←0;
while (j<n) do {
  if (i=n) then return j
  if (A[i,j] ≠ 0) then {j←j+1 ; i←0}
  else i←i+1
}
return -1
```

COMPLESSITÀ

UPPER BOUND $O(n^2)$

* $\leq n^2$ iterazioni del while
* $\Theta(1)$ operazioni in ciascuna iterazione del while e fuori del while

LOWER BOUND basato su una particolare istanza

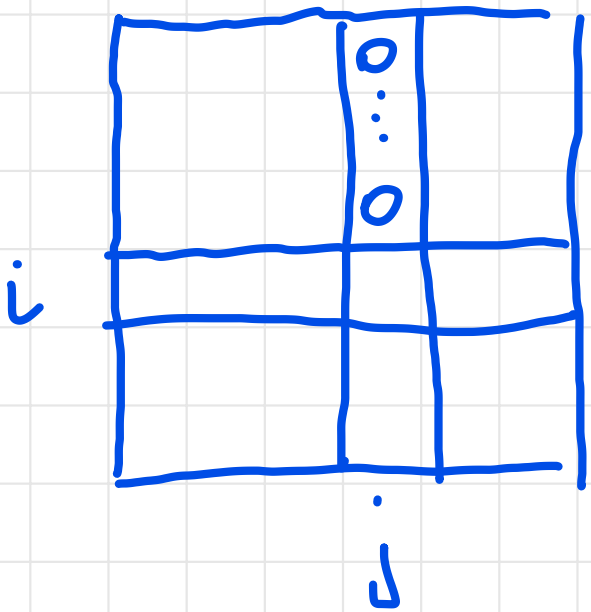


$\Rightarrow n^2$ iterazioni del while $\Rightarrow \Omega(n^2)$

$\Rightarrow$ Complessità $\in \Theta(n^2)$

CORRETTEZZA: basata sul seguente invariante

INVARIANTE che vale alla fine di ogni iterazione del while quando l'algoritmo è posizionato sulla riga i e sulla colonna j



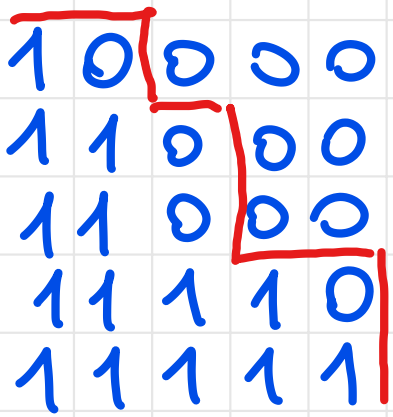
* nessuna delle colonne di indice $0, 1, \ldots, j-1$ ha tutti $\varnothing$
* $A[0,j] = A[1,j] = \ldots = A[i-1,j] = \varnothing$

Esercizio: completare le frasi di cronologia usando l'imperfetto scritto sopra

Esercizio

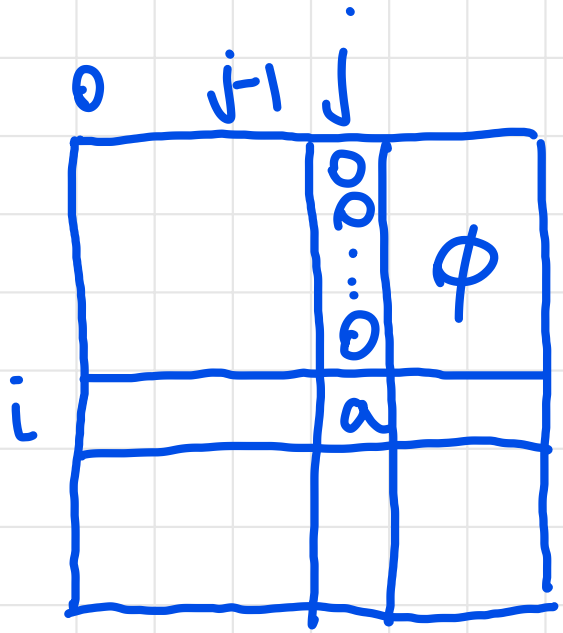
Sia A una matrice binaria $n \times n$ per la quale valgono le seguenti proprietà: (a) in ciascuna riga gli 1 vengono prima degli 0; e (b) il numero di 1 nella riga i è $\leq$ del numero di 1 nella riga $i+1$, per $0 \leq i \leq n-2$. Descrivere un algoritmo che in tempo $O(n)$ conti il numero di 1 in A, e provarne la correttezza. L'algoritmo deve contenere un solo ciclo.

Esempio $n=5$



Idea: seguire il profilo degli 1 più a dx in ogni riga (includendo l'eventuale 0 alla sua dx)

Usiamo un solo ciclo while con 2 indici i, j (riga, colonna) e un contatore count (± 1) in modo da mantenere il seguente invariante



$A[i,j]$ = a prossimo elemento da eliminare

* count = #1 nelle colonne dalle 0 alle $j-1$ e nelle righe $0..i-1$
* $A[0,j] = A[1,j] = \dots = A[i-1,j] = 0$

Come mantenere l'invariante?

Inizializzazione: $i = j = 0$, $count = 0$

Nella generica iterazione che esamina $A[i,j] = a$

* se $a = 1 \Rightarrow \{count \leftarrow count + n - i \; ; j \leftarrow j + 1\}$
* se $a = 0 \Rightarrow i \leftarrow i + 1$

Esco quando i o j arrivano a n

Algoritmo Count(A)

input matrice binaria A nxn che soddisfa Q

output: #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
```

Complessità: $O(n)$ perché

* $\leq 2n$ iterazioni del while (in quanto operazione i o j cresce e arriva al più a n)
* $\Theta(1)$ operazioni in ciascuna iterazione del while e fuori del while

Correttezza: esercizio

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.

1. Progettare un algoritmo ricorsivo EU eficiente per risolvere questo problema, senza fare ricorso all'ordinamento.
2. Analizzare la complessità $t_{\mathrm{EU}}(n)$ usando l'albero della ricorsione.

Osservazione chiave: A contiene tutti elementi distinti se e solo se

* $A[0] \neq A[i] \quad \forall \quad 1 \leq i \leq n-1$
* $A[1 \div n-1]$ contiene tutti elementi distinti

Algoritmo EU(A, i)

Input array A[0 ÷ n-1], indice i $0 \leq i < n$

Output True/False se gli elementi di A[i ÷ n-1]

sono/non sono tutti distinti

Prima invocazione i = 0

```
n ← |A|
if (i = n-1) then return True // CASO BASE
k ← i+1;
while (k < n) do {
    if (A[k] = A[i]) then return False
    else k ← k+1
}
return EU(A, i+1)
```

COMPLESSITÀ: Albero delle ricorsioni per $EU(A, 0)$ con A array arbitrario di n elementi

* $\leq n$ chiamate ricorsive $i = 0, 1, \ldots, n-1$
* Costo associato alle chiamate ricorsive $EU(A, i)$ è dovuto alle $< n-i$ iterazioni del while ($\Theta(1)$ operazioni per iterazione)

  $\Rightarrow$ Costo $O(n-i)$

$\Rightarrow$ Costo totale $\in O\left(\sum_{i=0}^{n-1}(n-i)\right) = O\left(\sum_{i=1}^{n} i\right) = O(n^2)$

LOWER BOUND basato su una specifica istanza

$A \equiv$ array fatto di tutti elementi distinti

l'albero delle ricorsioni ha

* esattamente n nodi
* il nodo associato alla chiamata $EV(A, i)$ ha un costo $\Theta(n-i)$

Per questa istanza il costo totale è $\Theta(n^2)$

$\Rightarrow$ l'algoritmo ha complessità $\Theta(n^2)$

Esercizio

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)

Versione 1

(Assumiamo gli indici da 1 a n)

Proprietà: se $\exists$ un valore di maggioranza x allora $x = A[\lceil n/2 \rceil]$

$\Rightarrow$ È sufficiente contare il numero di

occorre di $A[\lceil n/2 \rceil]$

* banalmente con una scansione di A : $O(n)$
* 2 ricerche binarie : $O(\log n)$

Completare i dettagli per esercizio

Versione 1: A ordinato

Versione 2: nessuna assunzione su A