Algoritmi e Strutture Dati
12 Settembre 2022

Cognome .... **Nome** .... **Matricola** ....

Note

1. La leggibilità è un prerequisito: parti difficili da leggere potranno essere ignorate.
2. Quando si presenta un algoritmo è fondamentale spiegare l'idea soggiacente e motivarne la correttezza.
3. L'efficienza e l'aderenza alla traccia sono criteri di valutazione delle soluzioni proposte.
4. Si consegnano tutti i fogli, con nome, cognome, matricola e l'indicazione *bella copia* o *brutta copia*.

Domande

Domanda A (7 punti) Definire formalmente la classe $\Omega(f(n))$. Dimostrare che la seguente ricorrenza ha soluzione $T(n) = \Omega(n)$

$$T(n) = \frac{1}{3}T(n-1) + 2n + 1$$

Soluzione: L'insieme $\Omega(f(n))$ è definito come:

$$\Omega(f(n)) = \{g(n) : \exists c > 0.\ \exists n_0.\ \forall n \geq n_0.\ 0 \leq cf(n) \leq g(n)\}.$$

Si deve provare che asintoticamente, per un'opportuna costante $c > 0$

$$T(n) \geq cn$$

Si procede per induzione:

$$\begin{array}{rcll} T(n) & = & \frac{1}{3}T(n-1) + 2n + 1 & \text{[per definizione della ricorrenza]} \\ & \geq & \frac{1}{3}c(n-1) + 2n + 1 & \text{[per ipotesi induttiva } T(n-1) \geq c(n-1)\text{]]} \\ & \geq & 2n & \\ & \geq & cn & \end{array}$$

dove l'ultima disuguaglianza vale quando $c \leq 2$. Risulta dunque dimostrata la tesi.

Domanda B (6 punti) Si consideri una tabella hash di dimensione $m = 8$, gestita mediante chaining (liste di trabocco) con funzione di hash $h(k) = k \ mod \ m$. Si descriva in dettaglio come avviene l'inserimento della sequenza di chiavi: 28, 19, 10, 35, 26.

Soluzione: La tabella hash T contiene, in corrispondenza di ciascuna entry $T[i]$ la lista degli elementi x tali che $h(x.key) = i$. L'inserimento in testa alla lista garantisce complessità dell'inserimento $O(1)$.

Si ottiene

0	
1	
2	$\rightarrow 26 \rightarrow 10$
3	$\rightarrow 35 \rightarrow 19$
4	$\rightarrow 28$
5	
6	
7	

Esercizi

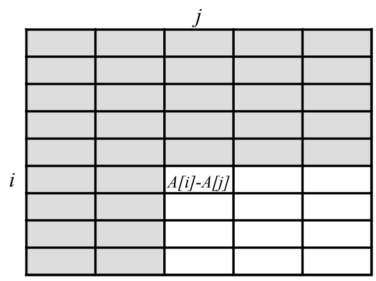
Esercizio 1 (10 punti) Realizzare una funzione `Diff(A,k)` che, dato un array `A[1,n]` ordinato in senso decrescente, verifica se esiste una coppia di indici `i, j` tali che `A[i]` $-$ `A[j]` $=$ `k`. Restituisce la coppia di indici se esiste e (0,0) altrimenti. La funzione non deve alterare l'input e deve operare in spazio costante. Scrivere lo pseudocodice, provarne la correttezza e valutarne la complessità.

Soluzione:
Il codice può essere:

```
diff(A, n, k):
    i=1
    j=1
    while (i<=n) and (j<=n) and (A[i]- A[j] <> k)
        if (A[i]- A[j] < k)
            j++
        else
            i++

    if (i <= n) and (j<=n)
        return (i,j)
    else
        return (0,0)
```

È facile vedere che si mantiene l'invariante $\forall (i', j') \in [1, n].(i' < i) \vee (j' < j) \Rightarrow A[i] - A[j] \neq k$, ovvero una coppia (i', j') tale che $A[i] - A[j] = k$ può esistere solo tra le coppie ancora esplorabili ($i' \geq i$ e $j' \geq j$), ovvero, graficamente nella parte non grigia:



Infatti, inizialmente, con $i = j = 1$, l'invariante è vacuamente vero.
Ad ogni iterazione, se entro nel ciclo, ci sono due possibilità:

- Se $A[i] - A[j] < k$, allora incremento j. In questo modo, escludo dall'esplorazione le coppie (i', j) con $i' \geq i$ per le quali, dato che l'array è decrescente e quindi $A[i] \geq A[i']$, vale

  $$A[i'] - A[j] \leq A[i] - A[j] < k.$$

  Dunque non escludo coppie utili, l'invariante continua a valere.

- Dualmente, $A[i] - A[j] > k$, allora incremento i. In questo modo, escludo dall'esplorazione le coppie (i, j') con $j' \geq j$ per le quali, dato che l'array è decrescente e quindi $A[j] \geq A[j']$, vale

  $$A[i] - A[j'] \geq A[i] - A[j] > k.$$

  Dunque, anche in questo caso, non escludo coppie utili, l'invariante continua a valere.

Quando esco dal ciclo, se $A[i] - A[j] = k$, ho concluso con successo. Altrimenti deve essere $i > n$ o $j > n$, che unitamente all'invariante, mi permettono di concludere che per ogni $i, j \in [1, n]$, $A[i]-A[j] \neq k$, come desiderato.

Da questo la correttezza segue immediatamente. La complessità è lineare. Il numero di iterazioni è pari al più a $2n - 1$, dato che i e j partono da 1, sono limitate da n ed ogni iterazione aumenta una delle due. Dato che ciascuna iterazione ha costo costante, ottengo $T(n) = O(2n - 1) = O(n)$.

Esercizio 2 (9 punti) Data una stringa di numeri interi $A = (a_1, a_2, \ldots, a_n)$, si consideri la seguente ricorrenza $c(i, j)$ definita per ogni coppia di valori (i, j) con $1 \leq i, j \leq n$:

$$c(i,j) = \begin{cases} a_j & \text{if } i = 1, 1 \leq j \leq n, \\ a_{n+1-i} & \text{if } j = n, 1 < i \leq n, \\ c(i-1,j) \cdot c(i,j+1) \cdot c(i-1,j+1) & \text{altrimenti.} \end{cases}$$

1. Si fornisca il codice di un algoritmo iterativo bottom-up COMPUTE_C(A) che, data in input la stringa A restituisca in uscita il valore $c(n, 1)$.
2. Si valuti il numero esatto $T_{CC}(n)$ di moltiplicazioni tra interi eseguite dall'algoritmo sviluppato al punto (1).

Soluzione:

1. Date le dipendenze tra gli indici nella ricorrenza, un modo corretto di riempire la tabella è attraverso una scansione "reverse column-major", in cui calcoliamo gli elementi della tabella in ordine decrescente di indice di colonna e, all'interno della stessa colonna, in ordine crescente di indice di riga. Il codice è il seguente.

```
COMPUTE_C(A)
n = length(A)
for i=1 to n do
   c[1,i] = a_i
   c[i,n] = a_{n+1-i}
for j=n-1 downto 1 do
   for i=2 to n do
      c[i,j] = c[i-1,j] * c[i,j+1] * c[i-1,j+1]
return c[n,1]
```

   Si osservi che un altro modo corretto di riempire la tabella è attraverso una scansione "reverse diagonal", che scansiona per diagonali parallele alla diagonale principale partendo da quella contenente solo $c[1, n]$.

2. Ogni iterazione del doppio ciclo dell'algoritmo esegue due operazioni tra interi, e quindi

$$\begin{aligned} T_{CC}(n) &= \sum_{j=1}^{n-1} \sum_{i=2}^{n} 2 \\ &= \sum_{j=1}^{n-1} 2(n-1) \\ &= 2(n-1)^2. \end{aligned}$$

   Equivalentemente, basta osservare che l'algoritmo esegue due moltiplicazioni per ogni elemento di una tabella $(n-1) \times (n-1)$.