14/01

Domanda B (6 punti) Realizzare una funzione `SearchUnique(T,k)` che dato un albero binario di ricerca `T` (che si assume non vuoto), verifica se la chiave `k` è presente in un unico nodo, e, in caso affermativo restituisce il nodo, altrimenti (ovvero se la chiave non è presente oppure è presente in più nodi) restituisce `nil`. Si possono usare le funzioni standard degli alberi binari di ricerca. Valutarne la complessità.

```
SearchUnique(T,k)

if(Search(T.root, k) ≠ nil)

        if Search(Search(T.root, k).right, k) ≠ nil || Search(Search(T.root, k).left, k) ≠ nil
                return nil
        else
                return Search(T.root, k)
```

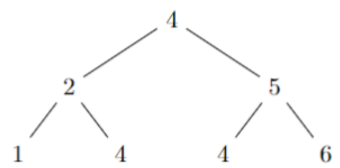
```
SearchUnique(T,k)
   x = Search(T.root,k)

   if (x == nil) or (Search(x.left, k) <> nil) or (Search(x.right, k) <> nil)
      return x
   else
      return nil
```

In alternativa, si può osservare che, se la chiave k dovesse comparire nel sottoalbero sinistro, sarebbe il massimo di questo sottoalbero e, dualmente, se comparisse nel secondo sottoalbero, sarebbe il minimo. Questo porta alla soluzione che segue

```
SearchUnique(T,k)
   x = Search(T.root,k)

   if (x <> nil)
      y = Max(x.left)
      z = Min(x.right)

      if ((y <> nil) and (y.key == k)) or (z <> nil) and (z.key == k))
         return nil
      else
         return x
   else
      return nil
```

La complessità asintotica non cambia, ma questa funzione è più efficiente della precedente dato che `Min` e `Max`, diversamente a `Search` non eseguono confronti sulle chiavi.

Per concludere, è da osservare che l'assunzione che le chiavi duplicate siano solo nel sottoalbero destro (o in quello sinistro) o che siano adiacenti non è legittima. Ad esempio, il seguente è un BST valido:

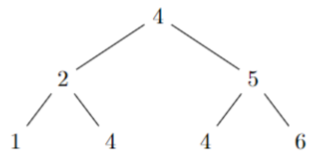


In alternativa, si può osservare che, se la chiave k dovesse comparire nel sottoalbero sinistro, sarebbe il massimo di questo sottoalbero e, dualmente, se comparisse nel secondo sottoalbero, sarebbe il minimo. Questo porta alla soluzione che segue

```
SearchUnique(T,k)
   x = Search(T.root,k)

   if (x <> nil)
      y = Max(x.left)
      z = Min(x.right)

      if ((y <> nil) and (y.key == k)) or (z <> nil) and (z.key == k))
         return nil
      else
         return x
   else
      return nil
```

La complessità asintotica non cambia, ma questa funzione è più efficiente della precedente dato che `Min` e `Max`, diversamente a `Search` non eseguono confronti sulle chiavi.

Per concludere, è da osservare che l'assunzione che le chiavi duplicate siano solo nel sottoalbero destro (o in quello sinistro) o che siano adiacenti non è legittima. Ad esempio, il seguente è un BST valido:



$\rightarrow O(h)$

Esercizio 1 (9 punti) Si consideri una variante degli alberi binari di ricerca nella quale i nodi `x` hanno un campo addizionale `x.min`, che rappresenta il minimo delle chiavi nel sottoalbero radicato in `x`. Realizzare la procedura `Insert(T,z)` che inserisce un nodo `z` nell'albero e la rotazione a sinistra `Left(T,x)` (assumendo che `x` e `x.right` non siano `nil`). Valutarne la complessità.

```
Insert(T, z)

x = T.root

while(x ≠ nil)

        if (z.key > x.key)
                x = x.right
        else
                x = x.left
                if (z.key < x.min)
                        x.min = z.key
```

```
if(x.key > z.key && x.left == nil)
        z.p = x
        x.left = z
else
        z.p = x
        x.right = z

z.min = z.key
```

```
Insert(T,z)
   y = nil
   x = T.root

   while (x <> nil)
      y = x

      if (z.key < x.key)
         if z.key < x.min     /* il sottoalbero radicato in z contiene z  */
            x.min = x.key     /* quindi si aggiorna x.min                 */

         x = x.left
      else
         x = x.right

   z.p = y
   if (y <> nil)
      if (z.key < y.key)
         y.left =  z
      else
         y.right = z

    z.min = z.key    /* il sottoalbero radicato in z contiene solo z */
```

$O(h)$

```
Left(T,z)
    y = x.right
    x.right = y.left
    x.right.p = x
    Transplant(T,x,y)
    y.left = x
    x.p = y

    y.min = x.min

    x.min = min(x.key, x.left.min, x.right.min)
```

→ MODIFICATE...

La complessità resta costante $O(1)$.

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.

h(28) = 28 mod 8 = 4
h(19) = 19 mod 8 = 3
h(10) = 10 mod 8 = 2
h(35) = 35 mod 8 = 3

[3] 35 → 19

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	→ 26 → 10
3	→ 35 → 19
4	→ 28
5	
6	
7	

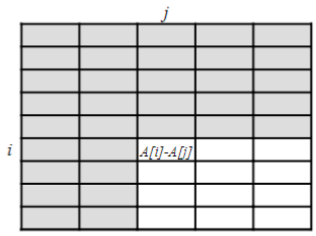
Esercizio 1 (10 punti) Realizzare una funzione `Diff(A,k)` che, dato un array `A[1,n]` ordinato in senso decrescente, verifica se esiste una coppia di indici `i, j` tali che `A[i]` − `A[j]` = `k`. Restituisce la coppia di indici se esiste e (0,0) altrimenti. La funzione non deve alterare l'input e deve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

```
Diff(A, k)

n = A.length
i=1, j=1

while(i ≤ n and j ≤ n and A[i] - A[j] ≠ k)
        if(A[j] - A[i] < k)
                j++;
        else
                i++;

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

I < J < K

ASSUMI ORDINE
INDICI

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.

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).

1

```
COMPUTE_C(A)

n = length(A)
for (i = 1 to n)
        c[1, i] = a_i
        if(i ≠ 1) c[i, n] = a_(n+i-1)

for j = n - 1 downto 2
        for i = 2 to n
                c[i, j] = c(i-1, j) * c(i, j+1) * c(i-1, j+1)

return c[n, 1]
```

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

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 $(n-1) \times (n-1)$.

Esercizio 2 (8 punti) Per $n > 0$, siano dati due vettori a componenti intere $\mathbf{a}, \mathbf{b} \in \mathbf{Z}^n$. Si consideri la quantità $c(i, j)$, con $0 \le i \le j \le n-1$, definita come segue:

$$c(i,j) = \begin{cases} a_i & \text{se } 0 < i \le n-1 \text{ e } j = n-1, \\ b_j & \text{se } i = 0 \text{ e } 0 \le j \le n-1, \\ c(i-1, j-1) \cdot c(i, j+1) & 0 < i \le j < n-1. \end{cases}$$

Si vuole calcolare la quantità $m = \min\{c(i,j) : 0 \le i \le j \le n-1\}$.

INIT / REC → MEMOIZ

1. Si fornisca il codice di un algoritmo iterativo bottom-up per il calcolo di m.
2. Si valuti la complessità esatta dell'algoritmo, associando costo unitario ai prodotti tra numeri interi e costo nullo a tutte le altre operazioni.

(1)

```
COMPUTE(a,b)
n = length(a)

for(i = 1 to n-1)
        c[i, n-1] = a[i]
        m = MIN(m, c[i, n-1])

for(j = 0 to n-1)
        c[0, j] = b[j]
        m = MIN(m, c[0, j])

for(j = n - 2 downto 1)
        for(i = 1 to n - 2)
                c[i, j] = c[i-1, j-1] * C[i, j+1]
                m = MIN(m, c[i, j])

return m
```

(2)

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

```
for i=1 to n-2 do
   for j=n-2 downto i do
      C[i,j] <- C[i-1,j-1] * C[i,j+1]
      m <- MIN(m,C[i,j])
return m
```

(b)

$$T(n) = \sum_{i=1}^{n-2} \sum_{j=i}^{n-2} 1 = \sum_{i=1}^{n-2} n - 1 - i = \sum_{k=1}^{n-2} k = (n-1)(n-2)/2.$$

Esercizio 2 (10 punti) Dato un insieme di n numeri reali positivi e distinti $S = \{a_i, a_2, \ldots, a_n\}$, con $0 < a_i < a_j < 1$ per $1 \le i < j \le n$, un *(2,1)-boxing* di S è una partizione $P = \{S_1, S_2, \ldots, S_k\}$ di S in k sottoinsiemi (cioè, $\bigcup_{j=1}^{k} S_j = S$ e $S_r \cap S_t = \emptyset, 1 \le r \ne t \le k$) che soddisfa inoltre i seguenti vincoli:

$$|S_j| \le 2 \quad \text{e} \quad \sum_{a \in S_j} a \le 1, \quad 1 \le j \le k.$$

In altre parole, ogni sottoinsieme contiene al più due valori la cui somma è al più uno. Dato S, si vuole determinare un (2,1)-boxing che minimizza il numero di sottoinsiemi della partizione.

1. Scrivere il codice di un algoritmo greedy che restituisce un (2,1)-boxing ottimo in tempo lineare. (Suggerimento: si creino i sottoinsiemi in modo opportuno basandosi sulla sequenza ordinata.)
2. Si enunci la proprietà di scelta greedy per l'algoritmo sviluppato al punto precedente e la si dimostri, cioè si dimostri che esiste sempre una soluzione ottima che contiene la scelta greedy.

```
(2-1)-BOXING(S)
n = |S|
OPT = 0
// indici
first = 1
last = n

while(first ≤ last)
        if(a_first + a_last ≤ 1 and first < last)
                OPT = OPT U {a_first + a_last}
                first = first + 1
        else
                OPT = OPT U {a_last}
    last = last - 1
return OPT
```

1. L'idea è provare ad accoppiare il numero più piccolo (a_1) con quello più grande (a_n). Se la loro somma è al massimo 1, allora $S_1 = \{a_1, a_n\}$, altrimenti $S_1 = \{a_n\}$. Poi si procede analogamente sul sottoproblema $S \setminus S_1$.

```
(2,1)-BOXING(S)
n <- |S|
P <- empty_set
first <- 1
last <- n
while (first <= last)
   if (first < last) and a_first + a_last <= 1 then
      P <- P U {{a_first, a_last}}
      first <- first + 1
   else
      P <- P U {{a_last}}
   last <- last - 1
return P
```

Questo algoritmo scansiona ogni elemento una sola volta, quindi la sua complessità è lineare.

2. La scelta greedy è $\{a_1, a_n\}$ se $n > 1$ e $a_1 + a_n \leq 1$, altrimenti $\{a_n\}$. Ora dimostriamo che esiste sempre una soluzione ottima che contiene la scelta greedy. I casi $n = 1$ e $a_1 + a_n > 1$ sono banali, visto che in questi casi ogni soluzione ammissibile deve contenere il sottoinsieme $\{a_n\}$. Quindi assumiamo che la scelta greedy sia $\{a_1, a_n\}$. Consideriamo una qualsiasi soluzione ottima dove a_1 e a_n non sono accoppiati nello stesso sottoinsieme. Quindi, esistono due sottoinsiemi S_1 e S_2, con $a_1 \in S_1$ e $a_n \in S_2$. Sostituiamo questi due sottoinsiemi con $S'_1 = \{a_1, a_n\}$ (cioè, la scelta greedy) e $S'_2 = S_1 \cup S_2 \setminus \{a_1, a_n\}$. $|S'_2| \leq 2$ e, se $|S'_2| = 2$, allora $S'_2 = \{a_s, a_t\}$ con $a_s \in S_1$ e $a_t \in S_2$. Siccome a_t era precedentemente accoppiato con a_n, a maggior ragione può essere accoppiato con $a_s < a_n$, quindi la nuova soluzione così creata è ammissibile e ancora ottima.