

Esercizio 2 (10 punti) Si consideri il problema di selezione di attività compatibili, con n attività a_1, \dots, a_n che ci vengono date attraverso due vettori s e f di tempi di inizio e fine, e ordinate per tempo di *inizio* (cioè $0 < s_1 \leq s_2 \leq \dots \leq s_n$). \rightarrow MUTUALLY COMPATIBLE

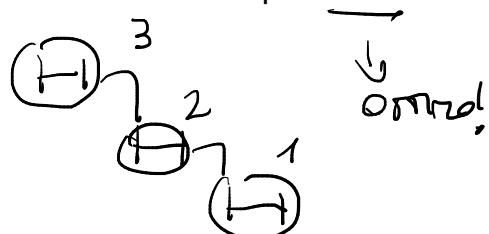
(a) Scrivere un algoritmo greedy iterativo che implementa la scelta greedy di selezionare l'attività che inizia per ultima.

(b) Determinare l'insieme di attività restituito dall'algoritmo al punto (a) quando eseguito sul seguente insieme di 6 attività, caratterizzate dai seguenti vettori s e f di tempi di inizio e fine:

$$s = (1, 2, 3, 5, 7, 10) \quad f = (3, 9, 10, 7, 11, 12)$$

(c) Dimostrare la proprietà di scelta greedy, cioè che esiste soluzione ottima che contiene l'attività che inizia per ultima.

(a) Attività che inizia per ultima?

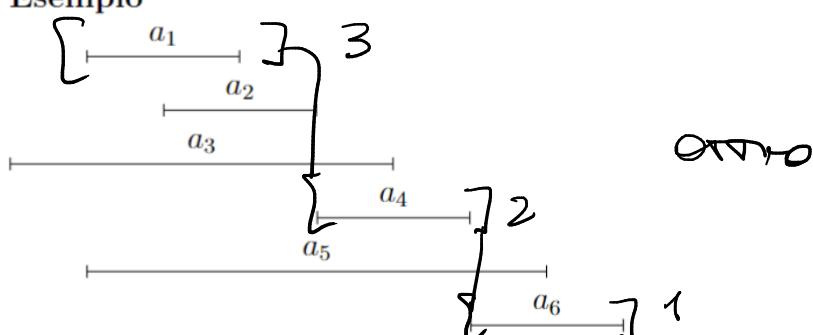


GREEDY-SEL(S, f) \rightarrow ITERATIVO

```

1   $n = S.length$ 
2   $A = \{a_n\}$ 
3   $last = n //$  indice dell'ultima attività selezionata
4  for  $m = 2$  to  $n$     FOR  $i = N-1$  DOWNTO 1
5      if  $s_m \geq f_{last}$ 
6           $A = A \cup \{a_m\}$ 
7           $last = m$ 
8  return  $A$ 
  
```

Esempio



$$A = a_1$$

$$last = 1$$

$$A = a_1, a_4$$

$$last = 4$$

$$A = a_1, a_4, a_6$$

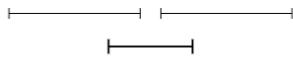
$$last = 6$$

8.2.4 Scelte Greedy Alternative → Controesempi

Oltre alla scelta greedy vista precedentemente, ne esistono altre:

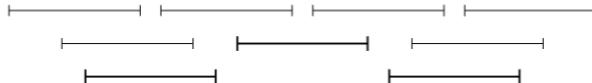
- Scegli l'attività di durata inferiore → non è ottima.

Controesempio:



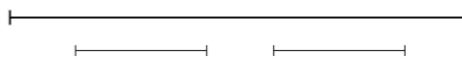
- Scegli l'attività col minor numero di sovrapposizioni → non è ottima.

Controesempio:



- Scegli l'attività che inizia per prima → non è ottima.

Controesempio:



- Scegli l'attività che inizia per ultima → è ottima.

(c) Dimostrare la proprietà di scelta greedy, cioè che esiste soluzione ottima che contiene l'attività che inizia per ultima.

$$\begin{aligned}
 & A_m \quad (+) + \\
 & S^* \quad \text{---} \quad \left[\begin{array}{l} \text{CUT-AND-PASTE} \\ \rightarrow \text{GREEDY...} \end{array} \right] \\
 & S^* \subset S^* \cup A_i = S^* \setminus A_s \\
 & \text{UNION} = \text{CUP} \quad \text{NO OPT} \\
 & i \in S^*, S \not\in S^*
 \end{aligned}$$

Sia S^* l'insieme delle attività ottime.

Partendo dall'assunzione che S^* sia ordinato, l'algoritmo greedy seleziona, partendo dall'attività "n", la successiva attività (mutualmente compatibile) tale che $S^* = S^* \cup A_{-i}$ (ndr - "cut-and-paste").

Una qualsiasi attività che non rispetta il mio ordine (e.g. A_j) non rientra nell'insieme ottimo

$$S^{\text{out}} = S^* \setminus A_j$$

Se la soluzione greedy ne comprende una, il nostro algoritmo non è ottimo.

(c) La dimostrazione della proprietà greedy si basa sul fatto che esiste sempre una soluzione ottima che include l'attività con tempo di inizio massimo:

Sia OPT una soluzione ottima che non include l'attività j con massimo tempo di inizio.

Sia i l'ultima attività (per tempo di inizio) in OPT.

Poiché $s[j] \geq s[i]$ (j ha il massimo tempo di inizio) e le attività sono compatibili, possiamo sostituire i con j in OPT ottenendo una soluzione ottima che include j .

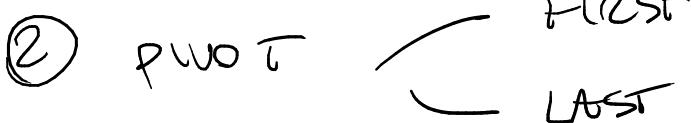
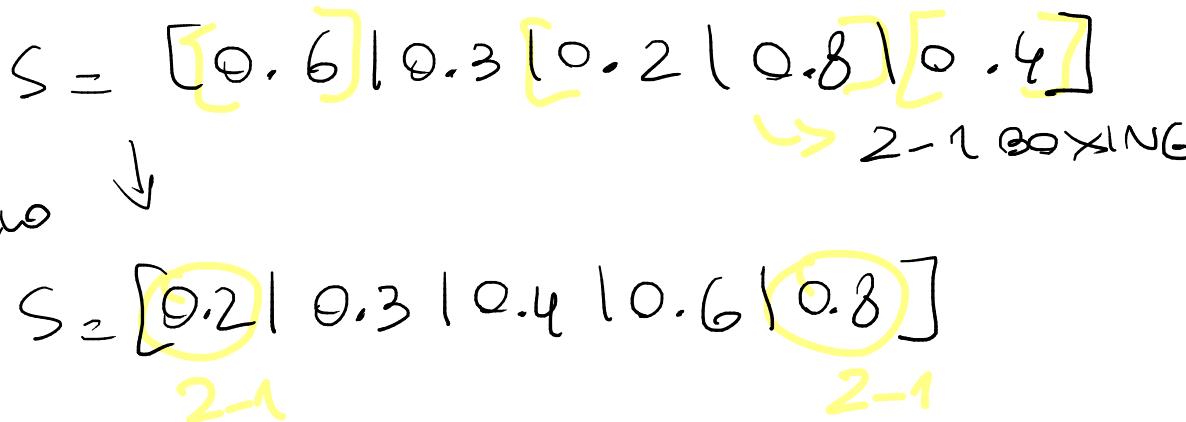
Questo dimostra che la scelta greedy dell'attività con massimo tempo di inizio è sicura.

Esercizio 2 (10 punti) Dato un insieme di n numeri reali positivi e distinti $S = \{a_1, a_2, \dots, a_n\}$, con $0 < a_i < a_j < 1$ per $1 \leq i < j \leq n$, un $(2,1)$ -boxing di S è una partizione $P = \{S_1, S_2, \dots, S_k\}$ di S in k sottoinsiemi (cioè, $\bigcup_{j=1}^k S_j = S$ e $S_r \cap S_t = \emptyset, 1 \leq r \neq t \leq k$) che soddisfa inoltre i seguenti vincoli:

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

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.

- (a) Progettare 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.)
- (b) Enunciare 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.



Soluzione:

- 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
Inizializziamo l'insieme, la partizione e gli estremi.
first <- 1
last <- n
while (first <= last)   Si considera un ciclo per cui "first" è <= "last" (perché scansioniamo gli estremi, come detto)
    if (first < last) and a_first + a_last <= 1 then
        P <- P U {{a_first, a_last}}      Esempio:
                                            1 2 3 4 5 6 7
        first <- first + 1                P = (7, 6, 5, 4, 3, 2, 1)
    else
        P <- P U {{a_last}}            Altrimenti,
        last <- last - 1              salvo solo l'estremo superiore migliore, la cui somma
                                        è sempre >= 1
```

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

- 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 ammessa 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 $a_n \in S_2$. Sostituendo questi due sottoinsiemi con $S'_1 = \{a_1, a_n\}$ (cioè, la scelta greedy) e $S'_2 = S_1 \cup S_2 \setminus \{a_1, a_n\}$, $|S'_1| \leq 2$ 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 ammessa e ancora ottima.

Esercizio 1 (9 punti) Realizzare una procedura `triplet(A)` che dato un array $A[1, n]$ di interi verifica se esistono tre indici, non necessariamente distinti, i, j e k tali che $A[i] + A[j] = A[k]$. Fornire lo pseudocodice, motivare la correttezza della soluzione e valutarne la complessità.

WHILE (CORPO) $i \leq j \leq k$

WHILE $(i \leq n \text{ AND } j \leq n \text{ AND } k \leq n)$
 $(\text{AND } A[i] + A[j] <= k)$

IF $(A[i] + A[j] < k)$

$i++$

$j--$

BUS

$j++$

$i--$

Domanda B (6 punti) Dare la definizione di max-heap. Dato l'array A con elementi 7, 1, 17, 0, 5, 4, 22, si specifichi il max-heap ottenuto applicando ad A la procedura `BuildMaxHeap`. Si descriva sinteticamente come si procede per arrivare al risultato (ad esempio illustrando come opera `BuildMaxHeap` e riportando il contenuto dello heap nei passi intermedi).

MAX HEAP \Rightarrow

ALBERO
QUASI-COMPLETO
CON PROPRIETÀ
DI HEAP

Un max heap è un albero binario che soddisfa due proprietà fondamentali: la proprietà strutturale dell'heap e la proprietà di ordinamento dell'heap. La proprietà strutturale richiede che l'albero sia completo fino al penultimo livello, con l'ultimo livello riempito da sinistra verso destra senza interruzioni (proprietà di addossamento a sinistra).

La proprietà di ordinamento stabilisce che il valore di ogni nodo deve essere maggiore o uguale ai valori dei suoi figli, garantendo così che il nodo radice contenga sempre il valore massimo dell'heap. Formalmente, per ogni nodo i dell'array che rappresenta l'heap, deve valere $A[i] \geq A[2i+1]$ per il figlio sinistro e $A[i] \geq A[2i+2]$ per il figlio destro, dove presenti.

2. Proprietà dell'Heap: Per ogni nodo i , il valore del nodo è maggiore o uguale ai valori dei suoi figli

- $A[i] \geq A[2i+1]$ (figlio sinistro)
- $A[i] \geq A[2i+2]$ (figlio destro)

```
HEAPSORT( $A$ )
1 BUILD-MAX-HEAP( $A$ ) //  $O(n)$ 
2 for  $i = A.length$  down to 2
3    $A[1] \leftrightarrow A[i]$ 
4    $A.heapsize = A.heapsize - 1$ 
5   MAX-HEAPIFY( $A, 1$ ) //  $O(\log n)$ 
```

Complessità $O(n \log n)$.

```
MAX-HEAPIFY( $A, i$ )
1  $l = \text{LEFT}(i)$ 
2  $r = \text{RIGHT}(i)$ 
3 if ( $l \leq A.heapsize$ ) and ( $A[l] > A[i]$ )
4    $max = l$ 
5 else
6    $max = i$ 
7 if ( $r \leq A.heapsize$ ) and ( $A[r] > A[max]$ )
8    $max = r$ 
9 if ( $max \neq i$ )
10   $A[i] \leftrightarrow A[max]$ 
11  MAX-HEAPIFY( $A, max$ )
```

FUNZIONI
NORDIC

Esercizio 2 (11 punti) Una *longest common substring* di due stringhe X e Y è una sottostringa di X e di Y di lunghezza massima. Si vuole progettare un algoritmo efficiente per calcolare la lunghezza di una longest common substring. Per semplicità si assuma che entrambe le stringhe di input abbiano stessa lunghezza n .

- Qual è la complessità dell'algoritmo esaustivo che analizza tutte le possibili sottostringhe comuni?
- Assumendo di conoscere un algoritmo che determina se una stringa di m caratteri è sottostringa di un'altra stringa di n caratteri in tempo $O(m+n)$, come si può modificare l'algoritmo del punto precedente per renderlo più efficiente?
- Progettare un algoritmo di programmazione dinamica più efficiente di quello del punto precedente. Sono richiesti relazione di ricorrenza sulle lunghezze (senza dimostrazione) e algoritmo bottom-up. (Suggerimento: considerare la lunghezza della longest common substring dei prefissi $X_i = \langle x_1, \dots, x_i \rangle$ e $Y_j = \langle y_1, \dots, y_j \rangle$ che termina con x_i e y_j , rispettivamente.)

$X = m$
 $Y = m$
 $\rightarrow O(m^2)$

SSONPO $\left\{ \begin{array}{l} X = \text{"PATTERN"} \\ Y = \text{"MATCH PATT"} \end{array} \right\}$ |LCS| = 4

LCS \rightarrow non ottenibile (Bruteforce)

FOR $i = 1$ $\rightarrow n$
FOR $s = 1$ $\rightarrow n$ $\rightarrow O(m^2)$

$|X| = m$
 $|Y| = n$
IF $(X_{-i} = Y_{-s})$
 $LCS += X_{-i}$

$O(m^2) \rightsquigarrow O(m+n)$

1. Generazione delle sottostringhe di X :

- Ogni sottostringa è definita da una coppia (i, j) con $0 \leq i \leq j < n$.
- Esistono $O(n^2)$ coppie possibili, quindi $O(n^2)$ sottostringhe.

2. Verifica per ogni sottostringa:

- Per verificare se una sottostringa di lunghezza m è presente in Y , dobbiamo scansionare Y (operazione $O(n)$).
- Ripetiamo questa verifica per tutte le $O(n^2)$ sottostringhe di X .

Complessità totale:

$$O(n^2) \text{ (numero di sottostringhe)} \cdot O(n) \text{ (tempo per ciascuna verifica)} = O(n^3).$$

↑ RAGLIORSVUS ..

(b) $O(\underline{m+n})$
 $\times \quad Y$

① $\text{MAX} = \emptyset$

For $i = 1 \dots n$

For $s = 1 \dots n$

$\left[\text{if } X_i = Y_s \right] m + n$

$\text{MAX} = \overbrace{\text{MAX} + X_i}$

m

(b) Miglioramento con l'algoritmo di substring matching $O(m+n)$:

Possiamo ridurre la complessità a $O(n^2)$ in quanto:

1. Generiamo ancora tutte le sottostringhe della prima stringa: $O(n^2)$
2. Per ogni sottostringa, verifichiamo se è presente nella seconda usando l'algoritmo dato in $O(m+n)$ Complessità risultante: $O(n^2)$

```

Algorithm ImprovedLCS(X, Y)
Input: stringhe X, Y di lunghezza n
Output: lunghezza della longest common substring

max_len = 0

for i = 1 to n do
    for l = 1 to n-i+1 do
        substring = X[i...(i+l-1)]

        if SubstringMatch(Y, substring) then // O(m+n) algorithm dato
            if l > max_len then
                max_len = l
            endif
        endfor
    endfor

return max_len

```

Domanda B (6 punti) Calcolare la lunghezza della longest common subsequence (LCS) tra le stringhe *store* e *shoes*, calcolando tutta la tabella $L[i, j]$ delle lunghezze delle LCS sui prefissi usando l'algoritmo visto in classe.

Soluzione: Si ottiene

	s	h	o	e	s	
s	0	0	0	0	0	0
t	0	1	1	1	1	1
o	0	1	1	2	2	2
r	0	1	1	2	2	2
e	0	1	1	2	3	3

$(q, s) = \infty$

$\leftarrow \text{print-LCS}$

$(l, p) = 0$

$$l(i, j) = |LCS(X_i, Y_j)|$$

$$l(i, j) = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 \quad (\text{caso 0}) \\ l(i-1, j-1) + 1 & \text{se } i, j > 0 \text{ e } x_i = x_j \quad (\text{caso 1}) \\ \max\{l(i, j-1), l(i-1, j)\} & \text{se } i, j > 0 \text{ e } x_i \neq x_j \quad (\text{caso 2}) \end{cases}$$

Alla fine ci interessa calcolare $l(m, n)$.

Per calcolare $l(i, j)$ mi possono servire tre valori:

RICORSIVITÀ

D&R
COSTR

$$L = \begin{bmatrix} & (i-1, j-1) & (i-1, j) \\ & \swarrow & \uparrow \\ (i, j-1) & \leftarrow & (i, j) \end{bmatrix}$$

Informazione addizionale per costruire la sequenza (vera e propria):

$$b(i, j) = \begin{cases} '↖' & \text{se } x_i = y_j \\ '←' & \text{se } x_i \neq y_j \text{ e } \max = LCS(i, j - 1) \\ '↑' & \text{se } x_i \neq y_j \text{ e } \max = LCS(i - 1, j) \end{cases}$$

↓
Esercizio
6 punti ...

$LCS(X, Y)$

```

1  m = X.[length]
2  n = Y.length
3  for i = 0 to m
4      L[i, 0] = 0
5  for j = 0 to n
6      L[0, j] = 0
7  for i = 1 to m
8      for j = 1 to n
9          if x_i = y_j
10         L[i, j] = L[i - 1, j - 1] + 1
11         B[i, j] = '↖'
12     else if L[i - 1, j] ≥ L[i, j - 1]
13         L[i, j] = L[i - 1, j]
14         B[i, j] = '↑'
15     else
16         L[i, j] = L[i, j - 1]
17         B[i, j] = '←'
18 return (L[m, n], B)

```

```

PRINT-LCS(B, X, i, j)
1  if i = 0 or j = 0
2      return ε
3  if B[i, j] = '↖'
4      PRINT(x_i)
5  else if B[i, j] = '←'
6      PRINT-LCS(B, X, i - 1, j - 1)
7  else // B[i, j] = '↑'
8      PRINT-LCS(B, X, i - 1, j)
9

```

Esercizio

$$X = \langle b, d, c, d \rangle$$

$$Y = \langle a, b, c, b, d \rangle$$

Restituisce $LCS(X, Y)$ e $|LCS(X, Y)|$

$$L = \begin{bmatrix} a & b & c & b & d \\ b & 0 & 0 & 1 & 1 & 1 \\ d & 0 & 0 & 1 & 1 & 2 \\ c & 0 & 0 & 1 & 2 & 2 \\ d & 0 & 0 & 1 & 2 & 3 \end{bmatrix} \quad B = \begin{bmatrix} a & b & c & b & d \\ b & ↑ & ↖ & ← & ↖ & ← \\ d & ↑ & ↑ & ↑ & ↑ & ↖ \\ c & ↑ & ↑ & ↖ & ← & ↑ \\ d & ↑ & ↑ & ↑ & ↑ & ↖ \end{bmatrix}$$

$$LCS(X, Y) = \langle b, c, d \rangle \quad |LCS(X, Y)| = 3$$

RICORSO N°6 → RISOLVI DIRETTAMENTE
 → PUNZONI CON POSSIBILI CODICI
 → E DIMOSSA
 → CATEGORIA DI DISUGUAGLIANZA

$$O(n) \rightarrow T(n) \leq C(n)$$

$$\Theta(n) \leq O(n)$$

$$\Omega(n) \rightarrow T(n) \geq \alpha(n)$$

)
\\.

INDUZIONE PER IND → QUANDO INDUZIONE
SOPRALEGA
FALLISCE

$$5.6. T(n) \leq C(n)$$

$$n^2 \leq n$$

DISUGUAGLIANZA
NON RISOTTATA



$$T(n-1) \leq C(n-1)$$



RISOLVENDO $\forall n, \forall c, d \geq 0$
(costanti) ...

HASHING

→ CHAINING (LINKED LIST)

→ DOPO IL HASH

$f_1, f_2 \rightarrow$ funzioni di sovrapposizione

$$h(k) = [f_1(k) + \underbrace{i \cdot f_2(k)}_{\emptyset}] \bmod m$$

NO COLLISIONS

$$h(k) = [f_1(k)] \bmod m$$

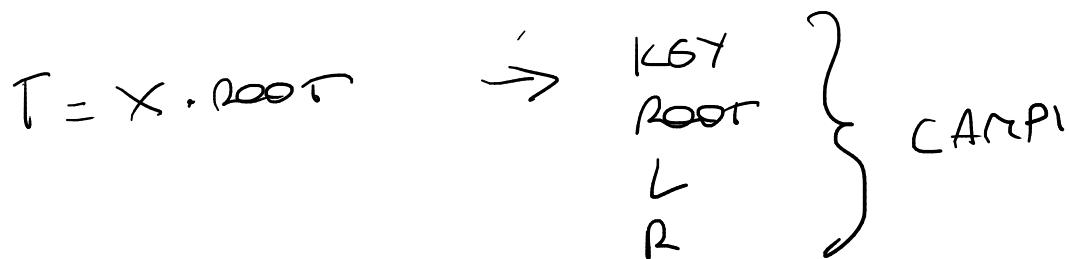
SI COLLISIONS

$$h(k) = [f_1(k) + 1 \cdot f_2(k)] \bmod m$$

FUNCTIONS STANDALONE (NUOUS)

ALBERI

ARRICCHIMENTO (CAMPIONE Dopo)



Esercizio 1 (9 punti) Realizzare un arricchimento degli alberi binari di ricerca che permetta di ottenere per ogni nodo x , in tempo costante, la media dei valori memorizzati nel sottoalbero radicato in x .

Indicare quali campi occorre aggiungere ai nodi. Fornire il codice per la funzione `avg(x)` che restituisce la media delle chiavi nel sottoalbero radicato in x e la procedura di inserimento di un nodo `insert(T, z)`.

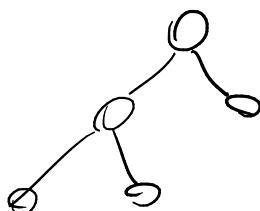
BST [< O >] RANGE / CHIavi Sono
UNIVOCHE

`avg(x)`

```
if (x == nil)
    return avg_rec(x)
```

`avg_rec(x)`

```
suml, countl = avg_rec(x.l)
sumr, countr = avg_rec(x.r)
```



$$\text{AVG} = \text{SUM} / N$$

```
return (suml + sumr) / (countl + countr)
```

Soluzione: L'idea è quella di arricchire la struttura facendo in modo che ogni nodo x , oltre ai campi usuali, abbia il campo $x.sum$ che contiene la somma delle chiavi per i nodi del sottoalbero radicato in x e $x.size$ che contiene il numero dei nodi in tale sottoalbero.

A questo punto è immediato realizzare la funzione `avg(x)` di tempo costante

```
avg(x)
if x == nil
    return x.sum/x.size
else
    error
```

Invece per l'inserimento, si modifica la procedura standard, che discende l'albero dalla radice fino alla posizione in cui inserire il nuovo nodo, facendo in modo che tutti i nodi attraversati e che quindi conterranno nel sottoalbero il nuovo nodo, abbiano i campi `sum` e `size` aggiornati.

```

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

while (x <> nil)
    y = x
    x.size++
    x.sum = x.sum + z.key
    if (z.key < x.key)
        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.sum = z.key
z.size = 1
z.left = z.right = nil

```

La complessità resta $O(h)$ dove h è l'altezza dell'albero.

Esercizio 1 (9 punti) Realizzare una funzione `avgTree(T)` che dato un albero binario T con chiavi numeriche, verifica se, per ogni nodo che abbia discendenti, la chiave del nodo è maggiore o uguale della media delle chiavi dei discendenti e ritorna conseguentemente un valore booleano (la radice dell'albero è `T.root` e ogni nodo x ha i campi `x.left` `x.right` e `x.key`).

→ LEFT] sum / sum / count / count
 → RIGHT]
 IF (== x.keySet)
 TRUE
 BIAS FALSE

```

avgTree(T)
v, n, s = avgTreeRec(T.root)
return v

# avgTreeRec(x):
# verifica se il sottoalbero radicato in x e' un avgTree e ritorna tre valori:
# - un booleano,
# - il numero dei discendenti
# - la somma delle loro chiavi

avgTreeRec(x)

if x = nil
    return true, 0, 0
else
    # ispeziono i sottoalberi sinistro e destro
    vl, nl, sl = avgTreeRec(x.left)
    vr, nr, sr = avgTreeRec(x.right)

```

Homework [Alberi / HSAF] → più frequenti

Esercizio 1 (9 punti) Realizzare una funzione `append(T, x, z)` che dato un albero binario di ricerca T , un suo nodo `foglia` x ed un nuovo nodo z , verifica se è possibile inserire z come figlio (destro o sinistro) di x in T . In caso affermativo ritorna l'albero esteso, altrimenti ritorna `nil`. (Assumere che i nodi dell'albero abbiano gli usuali campi $x.left$, $x.right$, $x.p$, $x.key$ e la radice sia $T.root$.) Valutarne la complessità.

↑

4 APP. 18-20

Esercizio 1 (9 punti) Realizzare una funzione `intersect(A1, A2, n)` che dati due array di interi $A1$ e $A2$, organizzati a min-heap, con capacità n , restituisce un nuovo array A , ancora organizzato a min-heap, che contiene l'intersezione dei valori contenuti in $A1$ e $A2$. Nel caso gli array contengano più occorrenze dello stesso valore v , l'intersezione mantiene il numero minimo di occorrenze di v (ad es. se $A1$ contiene i valori $1, 2, 2, 2$ e $A2$ contiene i valori $1, 1, 2, 2$ allora A conterrà $1, 2, 2$). Valutarne la complessità.

↑

5 APP. 18-20

Esercizio 1 (9 punti) Scrivere una funzione `Anc(T, k1, k2)` che dato un albero binario di ricerca T nel quale tutte le chiavi sono distinte, e due chiavi $k1, k2$ presenti in T , verifica se il nodo contenente $k1$ è antenato del nodo contenente $k2$. Valutare la complessità della funzione.

↑

1° APP. 20-21

Domanda B (7 punti) Scrivere una funzione `toTree(A)` che dato un array A organizzato a max-heap (dimensione $A.heapSize$), lo trasforma in un albero binario realizzato con strutture linked, ancora organizzato a max-heap e ritorna la radice di tale albero. Il nuovo albero è costituito da nodi x con i campi $x.p$ (parent), $x.k$ (chiave), $x.l$ e $x.r$ (figlio sinistro e figlio destro). Per allocare un nuovo nodo si assuma di avere a disposizione un costruttore `node()`. Valutare la complessità.

↑

2° APP. 20-21

Esercizio 1 (9 punti) Si consideri un albero binario T , i cui nodi x hanno i campi $x.l$, $x.r$, $x.p$ che rappresentano il figlio sinistro, il figlio destro e il padre, rispettivamente. Un *cammino* è una sequenza di nodi x_0, x_1, \dots, x_n tale che per ogni $i = 0, \dots, n-1$ vale $x_{i+1}.p = x_i$. Il cammino è detto *nil-terminated* se $x_n.l = nil$ oppure $x_n.r = nil$. Dato un certo k , diciamo che l'albero è k -bilanciato se tutti i cammini nil-terminated che iniziano dalla radice hanno lunghezze che differiscono al più di k . Scrivere una funzione `bal(T, k)` che dato in input l'albero T e un valore k verifica se T è k -bilanciato e ritorna un corrispondente valore booleano. Valutarne la complessità.

↑

3° APP. 20-21

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.

↑

1° APP. 21-22

[neo array]

3^o APP 20-21

Esercizio 1 (8 punti) Realizzare una funzione `missing(A, n)` che dato un array $A[1..n]$ che contiene interi nell'intervallo $[1, n]$ verifica se esiste qualche valore $v \in [1, n]$ che non compare in A , ovvero tale che $A[i] \neq v$ per ogni $i \in [1, n]$. Valutare la complessità.

2^o APP. 21-22

Esercizio 1 (10 punti) Un array di interi $A[1..n]$ si dice *triangolare* se esiste $q \in [1, n]$ tale che $A[1..q]$ è ordinato in modo crescente e $A[q..n]$ è ordinato in modo decrescente. Realizzare una funzione `maxTr(A)` che dato un array triangolare $A[1..n]$, senza elementi ripetuti, ne trova il massimo. Valutarne la complessità.

3^o APP. 21-22

Esercizio 1 (10 punti) Diciamo che un array senza ripetizioni $A[1, n]$ è *semi-ordinato* se esiste un indice k , con $1 \leq k < n$, tale che $A[k+1..n]A[1..k]$ sia ordinato, ovvero i sottoarray $A[k+1..n]$ e $A[1..k]$ sono ordinati e $A[n] < A[1]$. In questo caso l'indice k viene detto il centro dell'array. Ad esempio l'array che segue è semi-ordinato con centro $k = 4$.

1	2	3	4	5	6	7
4	9	12	18	-1	1	2

Scrivere una funzione `centre(A)` che dato un array A semi-ordinato ne restituisce il centro. Giustificare la correttezza dell'algoritmo e valutarne la complessità.

4^o APP. 21-22

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.

Esercizio 1 (9 punti) Realizzare una funzione `diff(A1, A2, n)` che dati due array di interi $A1$ e $A2$, organizzati a min-heap, con capacità n , restituisce un nuovo array A , ancora organizzato a min-heap, che contiene la differenza insiemistica dei valori contenuti in $A1$ e $A2$. Nel caso gli array contengano più occorrenze dello stesso valore v , nella differenza si inseriscono il numero di occorrenze di v in $A1$ meno il numero di occorrenze di v in $A2$ (ad es. se $A1$ contiene i valori $1, 2, 2, 2, 2$ e $A2$ contiene i valori $1, 2, 1, 2$ allora la differenza A conterrà $2, 2$). Valutarne la complessità.

3^o APP. 22-23

4^o APP. 22-23

Esercizio 1 (10 punti) Un ricco possidente deve lasciare in eredità le sue $2n$ proprietà a n figli. Il valore delle proprietà è contenuto in un array di numeri (reali positivi) $A[1..2n]$.

1. Sviluppare un algoritmo `Split(A, n)` che dato in input l'array $A[1..2n]$ dei valori delle proprietà e n , verifica se le proprietà possano essere partizionate in n coppie, tutte con lo stesso valore complessivo (di modo da assegnare una coppia di proprietà a ciascun figlio). Si assume $n > 0$.

Ad esempio, con $n = 2$, se $A = [1, 3, 4, 2]$ la risposta è positiva visto che le proprietà possono essere partizionate nelle coppie 1, 4 e 3, 2, mentre se $A = [1, 3, 5, 2]$ la risposta è negativa.

2. Si valuti la complessità dell'algoritmo proposto.
3. Si dimostri la correttezza dell'algoritmo proposto.

VARIANTE RUSSA

Domanda B (5 punti) Dare la definizione di albero binario di ricerca. Specificare l'albero ottenuto inserendo, con la procedura vista a lezione, a partire da un albero vuoto, i nodi aventi le seguenti chiavi: 5, 4, 8, 6, 12, 7. Si supponga che dall'albero così ottenuto si cancelli il nodo con chiave 5 e si indichi l'albero ottenuto. Sia per gli inserimenti che per la cancellazione, motivare sinteticamente il risultato ottenuto.

Esercizio 1 (7 punti) Scrivere una funzione `RBTree(T)` che dato in input un albero binario di ricerca T , i cui nodi x , oltre ai campi $x.key$, $x.left$ e $x.right$, hanno un campo $x.col$ che può essere B (per "black") oppure R (per "red"), verifica se questo è un Red-Black tree. In caso negativo, restituisce -1 , altrimenti restituisce l'altezza nera della radice. Valutarne la complessità.