01/12

RECAP → HASH TABLES → CHAINING
→ DOPPIO HASHING

Domanda 31 Si consideri una tabella hash di dimensione $m = 8$, e indirizzamento aperto con doppio hash basato sulle funzioni $h_1(k) = k \ mod \ m$ e $h_2(k) = 1 + k \ mod \ (m-2)$. Si descriva in dettaglio come avviene l'inserimento della sequenza di chiavi: 12, 3, 22, 14, 38.

DOPPIO HASH → H(KEY, I)

$$H(K, I) = (\underline{H_1(K)} + I \cdot \underline{H_2(K)}) \bmod m$$

$$(K \bmod m) \quad + I \cdot \ldots \quad (i + K (\bmod (m-2)] \bmod (m)$$

→ [RIASSUNTO DI ALGORITMI]

SOLUZIONE:

$m = 8 \qquad h(k,i) = [(k \bmod m) + i \cdot (1 + k \bmod (m-2))] \bmod m$

n° da inserire: 12, 3, 22, 14, 38

0	
1	
2	
3	
4	12
5	
6	
7	

$h(12,0) = 12 \bmod 8 + 0$
$= 4$

0	
1	
2	
3	3
4	12
5	
6	
7	

$h(3,0) = 3 \bmod 8 + 0$
$= 3$

0	
1	
2	
3	3
4	12
5	
6	22
7	

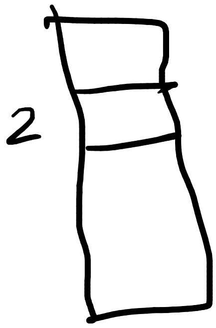
$h(22,0) = 22 \bmod 8 + 0$
$= 6$

0	
1	14
2	
3	3
4	12
5	
6	22
7	

$h(14,0) = 14 \bmod 8 + 0$
$= 6!$

$h(14,1) = (6 + 1 + 14 \bmod 6) \bmod 8$
$= (7 + 2) \bmod 8$
$= 1$

- **Domanda 34** Si consideri una tabella hash di dimensione $m = 8$, gestita mediante chaining (liste di trabocco) con funzione di hash $h(k) = k \bmod m$. Si descriva in dettaglio come avviene l'inserimento della sequenza di chiavi: 14, 10, 22, 18, 19.



(PRE)

14 ≡ 10 (COLLISIONE)

10 → 14 (POST)

• CHAINING: T[j] è lista di elementi con la stessa chiave (LIFO)

SOLUZIONE:

CIMA

m = 8 h(k) = k mod m no da inserire: 14, 10, 22, 18, 19

DOPO ...

14 mod 8 = 6

10 mod 8 = 2

22 mod 8 = 6

18 mod 8 = 2

19 mod 8 = 3

3.5.2 Master Theorem

Dato un problema con `size` n, vogliamo dividerlo in a sottoproblemi con `size` $\frac{n}{b}$. Otteniamo la seguente ricorrenza (ricordiamo che il caso base è omesso per semplicità):

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

con $a \geq 1,\ b > 1$, allora possiamo confrontare

- $f(n)$;
- $n^{\log_b a}$.

Tre possibili casi:

1. Se $f(n) = O(n^{\log_b a - \varepsilon})$ per qualche $\varepsilon > 0$, allora
$$T(n) = \Theta\big(n^{\log_b a}\big)$$
2. Se $f(n) = \Theta(n^{\log_b a})$ allora
$$T(n) = \Theta\big(n^{\log_b a} \cdot \log n\big)$$
3. Se $f(n) = \Omega(n^{\log_b a + \varepsilon})$ per qualche $\varepsilon > 0$, e vale la **regolarità**
$$\exists\, 0 < k < 1 \ :\ a \cdot f\left(\frac{n}{b}\right) \leq k \cdot f(n)$$
allora
$$T(n) = \Theta\big(f(n)\big)$$

Domanda A (8 punti) Si consideri la seguente funzione ricorsiva con argomento un array A di interi e due indici $1 \le p \le r \le A.length$:

```
Minimum(A,p,r)
   if p=r
      return A[p]
   else
      q = (p+r)/2
      return min(Minimum(A,p,q), Minimum(A,q+1,r))
```

QUICK - SORT

Dimostrare induttivamente che la funzione calcola il minimo del sottoarray $A[p..r]$. Inoltre, determinare la ricorrenza che esprime la complessità della funzione e mostrare che la soluzione è $\Theta(n)$, dove n indica la lunghezza del sottoarray. Motivare le risposte.

(1)

$T(\frac{n}{2}) + T(\frac{n}{2}) + c$

RICORR. $2T(\frac{n}{2}) + c$ (2)

(3) SOL. $\rightarrow \Theta(n)$

3.5.2 Master Theorem

Dato un problema con `size` n, vogliamo dividerlo in a sottoproblemi con `size` $\frac{n}{b}$. Otteniamo la seguente ricorrenza (ricordiamo che il caso base è omesso per semplicità):

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

$\longleftrightarrow T(n) = 2 \cdot T(\frac{n}{2}) + c$

con $a \ge 1,\ b > 1$, allora possiamo confrontare

- $f(n)$;
- $n^{\log_b a}$.

Tre possibili casi:

SUPERIORE $\rightarrow$ $\emptyset$

1. Se $f(n) = O(n^{\log_b a - \varepsilon})$ per qualche $\varepsilon > 0$, allora
$$T(n) = \Theta(n^{\log_b a})$$
2. Se $f(n) = \Theta(n^{\log_b a})$ allora $\rightarrow$ MISTA $\rightarrow k$
$$T(n) = \Theta(n^{\log_b a} \cdot \log n)$$
3. Se $f(n) = \Omega(n^{\log_b a + \varepsilon})$ per qualche $\varepsilon > 0$, e vale la **regolarità**
$$\exists\, 0 < k < 1 : \boxed{a \cdot f(\frac{n}{b}) \le k \cdot f(n)}$$
$\hookrightarrow \infty$

INFERIORE

allora
$$T(n) = \Theta(f(n))$$

$\lim_{n \to \infty} \frac{f(n)}{n^{\log_b(a)}} \approx n^{1-\varepsilon}$

$\approx n$

$\downarrow$

CASO 1

$n^{\log_b(a)} \rightarrow \Theta(n)$

$2T(\frac{n}{2}) + c \approx \Theta(n)$

Limite	Risultato	Soluzione
	$k = l$	$T(n) = \Theta\left(n^{\log_b a} \log(n)\right)$
$\lim_{n\to\infty} \frac{f(n)}{n^{\log_b a}} = k$	$k = 0$	$T(n) = \Theta\left(n^{\log_b a}\right)$ se $\exists k > 0 \mid \lim_{n\to\infty} \frac{f(n)}{n^{\log_b a - \varepsilon}} = k$
	$k = \infty$	$T(n) = \Theta(f(n))$ se $\exists k > 0 \mid \lim_{n\to\infty} \frac{f(n)}{n^{\log_b a - \varepsilon}} = k$ se $\exists k, 0 < k < 1 \mid a\, f\left(\frac{n}{b}\right) \le k f(n)$

Domanda A (7 punti) Dare la definizione della classe $\Theta(f(n))$. Mostrare che la ricorrenza

$$T(n) = \frac{1}{4}T(n-1) + \underline{3n^2} \text{ — TERMINE NOTO}$$

ha soluzione in $\Theta(n^2)$.

$\Theta(n)$ → $O(n^2) \Rightarrow T(n) \le c n^2$

→ $\Omega(n^2) \Rightarrow T(n) \ge d n^2$

$\forall c, d > 0$

$$T(n) = \frac{1}{4} T(n-1) + 3 \cdot n^2 \le c \cdot n^2$$

[SWAP "T" CON "c"
AND
SWAP n CON $\underline{n^2}$
CURRENT N] $\le \frac{1}{4} c (n-1)^2 + 3n^2$

$$\le \frac{1}{4} c (n^2 + 1 - 2n) + 3n^2$$

(APPROSSIMATO PER ECCESSO)

$$\le \frac{1}{4} c n^2 + 3n^2 \le c n^2$$

n^2 MAGGIORA $\approx n^2$

$$\frac{1}{4} cn^2 - cn^2 \le -3n^2$$

$$\cancel{4} \cdot \frac{cn^2 - 4cn^2}{\cancel{4}} \le -12n^2$$

$$-3cn^2 \le -12n^2$$

$$\frac{1}{3} \cdot \frac{3cn^2}{n^2} \ge \frac{12\cancel{n^2}}{\cancel{n^2}} \cdot \frac{1}{3} \leadsto c \ge 4, \forall n \ge 0$$

$$-\ T(n) = \Omega(n^2) \Rightarrow T(n) \ge d(n^2)$$

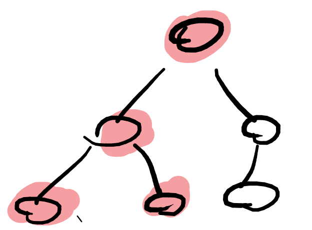
$$\dots\ d \le 3,\ \forall n \ge 0$$

Esercizio 1 (10 punti) Realizzare un arricchimento degli alberi binari di ricerca che permetta di ottenere per ogni nodo x, il grado di bilanciamento del sottoalbero radicato in x, definito come $h_x / \log_2(n_x + 1)$ dove h_x e n_x indicano rispettivamente l'altezza e il numero di nodi del sottoalbero radicato in x (si intende che un albero costituito da un solo nodo abbia altezza 1).
Indicare quali campi occorre aggiungere ai nodi. Fornire lo pseudo-codice per la funzione `bal(x)` che restituisce il grado di bilanciamento del sottoalbero radicato in x e la procedura di inserimento di un nodo `insert(T,z)`. Valutare la complessità delle funzioni definite.

BST (T, x)

X.ROOT / KEY / L / R

INSERT
DELETE
+ BAL(X)



3 + 1 = 4
(4/4 = 1)

+ X.H → ALTEZZA (CAMPO AGGIUNTIVO)

+ X.SIZE → N. DI SOTTONODI

H / SIZE → VALORIZZATI DA INSERT

Esercizio 1 (10 punti) Realizzare un arricchimento degli alberi binari di ricerca che permetta di ottenere per ogni nodo x, il grado di bilanciamento del sottoalbero radicato in x, definito come $h_x/\log_2(n_x+1)$ dove h_x e n_x indicano rispettivamente l'altezza e il numero di nodi del sottoalbero radicato in x (si intende che un albero costituito da un solo nodo abbia altezza 1).

Indicare quali campi occorre aggiungere ai nodi. Fornire lo pseudo-codice per la funzione `bal(x)` che restituisce il grado di bilanciamento del sottoalbero radicato in x e la procedura di inserimento di un nodo `insert(T,z)`. Valutare la complessità delle funzioni definite.

- BAL (X)

→ IF (X <> NIL)

→ RETURN X.H / LOG_2(X.SIZE)

→ ELSE

→ RETURN ERROR

↓ INSERT...

INSERT

SIZE / H

```
INSERT(T, z)
  x = T.root          -> RADICE
  y = NIL
  while x ≠ NIL
      y = x
      if z.key < x.key
          x = x.left
      else                     -> RIEMPIMO   -> SIZE ++;
          x = x.right
  z.p = y
  if y = NIL
      T.root = z              -> NODO Z -> PARENT ULTIMO
  else
      if z.key < y.key
          y.left = z
      else                     -> POSIZIONE Z
          y.right = z
```

Y.H ++

$h \sim \log_2(n+1)$

$O(h)$

ALTEZZA "FONTALE"

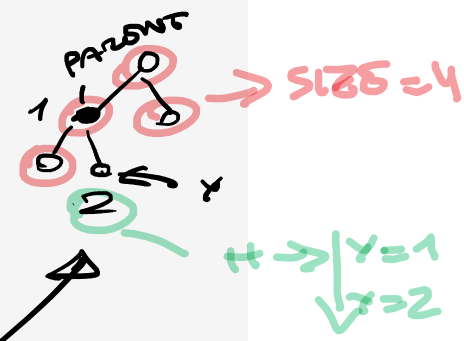
```
Insert(T,z)
   y = nil
   x = T.root

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

   // inizializza i campi del nodo z
   z.size = 1
   z.left = z.right = nil
   z.h = 1

   z.p = y
   if (y <> nil)       // se z non radice
      if (z.key < y.key)
         y.left =  z   <- H ++
      else
         y.right = z   <- H ++

   // se y era una foglia, ovvero se l'altezza di y era 1, la sua
   // altezza e conseguementemente quella degli antenati va
   // aumentata di 1 risalendo i nodi attraversati nella discesa
   if y.h == 1
      while y <> nil
           y.h ++
           y = y.p
```



Esercizio 1 (9 punti) Realizzare una funzione `strongBST(T)` che dato un albero binario `T` con chiavi numeriche non negative, verifica se, per ogni nodo, la chiave è maggiore uguale del doppio di ogni chiave nel sottoalbero sinistro e minore o uguale della metà di ogni chiave nel sottoalbero destro, 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`). Valutarne la complessità.

$$\frac{1}{2} \cdot K(\text{RIGHT}) \le K \ge 2 \cdot K(\text{LEFT})$$

$$K \ge 2 \cdot K(\text{LEFT})$$

AND

$$K \le \frac{1}{2} \cdot K(\text{RIGHT})$$

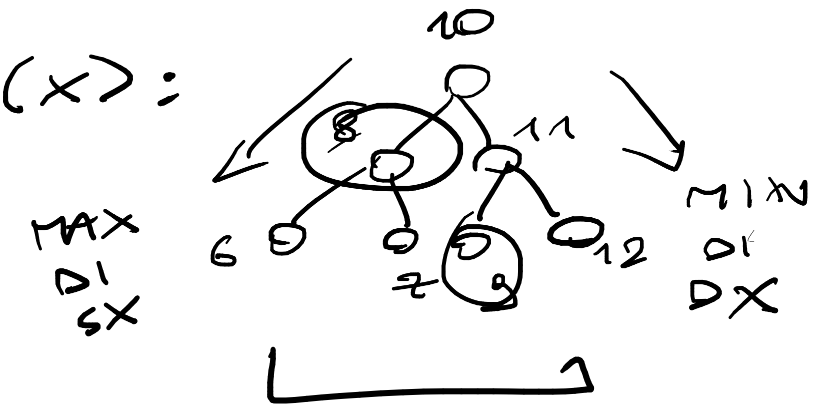
Esercizio 1 (9 punti) Realizzare una funzione `strongBST(T)` che dato un albero binario `T` con chiavi numeriche non negative, verifica se, per ogni nodo, la chiave è maggiore uguale del doppio di ogni chiave nel sottoalbero sinistro e minore o uguale della metà di ogni chiave nel sottoalbero destro, 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`). Valutarne la complessità.

LOG → L / R

STRONGBST (T)

S, m, M = STRONGBST_REC (T.ROOT)

STRONGBST_REC (X):



-> IF X = NIL
  -> RETURN TRUE, 0, +∞

INTERNAL TREES

-> ELSE
  -> SL, mL, ML = SB_REC(X.LEFT)
  -> SR, mR, MR = SB_REC (X.RIGHT)

MAX DI SX → TUTTO IL RESTO PIÙ PICCOLO

MIN DI DX ← TUTTO IL RESTO PIÙ GRANDE

≈ RISPARMIO ITERAZIONI

[MIN/MAX] → FUNZIONI PER BST ANNESSO

$$\underbrace{2\cdot M}_{MAX} \geq S \leq \frac{1}{2}\underbrace{m}_{MIN}$$

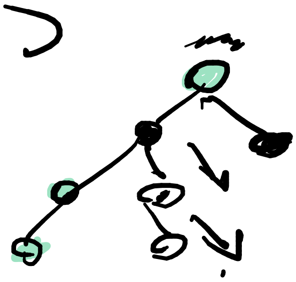
x.k ≥ 2 · M

x.k ≤ 1/2 m

m = MIN {mL, mR, x.KEY}

M = MAX {ML, MR, x.KEY}

```
IF (x.KEY >= 2·M) AND
   (x.KEY <= 1/2 · m)
   RETURN TRUE
ELSE
   RETURN FALSE
```



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 $1 \leq i < j \leq n$, 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 $S_r \cap S_t = \emptyset, 1 \leq r \neq t \leq k$) che soddisfa inoltre i seguenti vincoli:

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

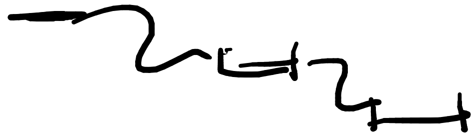
In altre parole, ogni sottoinsieme contiene al più due valori la cui somma è al più uno. Dato S, si vuole determinare un (2,1)-boxing che minimizza il numero di sottoinsiemi della partizione.

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.

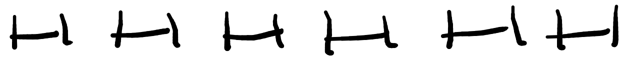
2-1 BOXING → 3/4 ESAMI

GREEDY — ATTIVITÀ



(PROBLEMI GREEDY?

BIN - MATCHING [1]



0.4 0.2 0.1 0.5 0.7

→ 0.1 / 0.2 / 0.4 / 0.5 / 0.7

L K R

< K >

0.1 / 0.2 / 0.9

2-1 BOXING

↳ 2 ELEMENTI LA CUI SOMMA È 1

$$\underbrace{\underset{\text{MIN}}{0.1\ldots} + \underset{\text{MAX}}{0.9}}_{1} \quad \left[\begin{array}{l} \sum a \leq 1 \\ a \in S \end{array} \right]$$

GREEDY

→ SCELTA OTTIMA → $S_1 = \{a_1, a_n\}$

2-1 BOXING (S)

→ $n = |S|$

→ GREEDY = NIL

[POSIZIONI] FIRST = 1
LAST = N

→ WHILE (FIRST <= LAST)

→ IF (FIRST < LAST)
AND (A_F + A_L ≤ 1)

→ GREEDY = GREEDY ∪ {A_F, A_L}
→ FIRST++ (AVANZIAMO)

0.1 0.2 + 0.8

ELSE

→ GREEDY = GREEDY ∪ {A_L}

0.1 0.2 + 0.3 0.8

1

→ RETURN GREEDY

SORT (SUBSUNT)

0.1 0.2 --- 0.3

FIRST LAST

```
(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
```

$P = \{a_1, a_n\}$

CONSEGNA

S/6 VOLTE ...

GREEDY-SEL(S, f)

```
n = S.length
A = {a_1}
last = 1 // indice dell'ultima attività selezionata
for m = 2 to n
      if s_m ≥ f_last
            A = A ∪ {a_m}
            last = m
return A
```

← RICORDA...

Esercizio 2 (10 punti) Abbiamo n programmi da eseguire sul nostro computer. Ogni programma j, dove $j \in \{1, 2, \ldots, n\}$, ha lunghezza ℓ_j, che rappresenta la quantità di tempo richiesta per la sua esecuzione. Dato un ordine di esecuzione $\sigma = j_1, j_2, \ldots, j_n$ dei programmi (cioè, una permutazione di $\{1, 2, \ldots, n\}$), il tempo di completamento $C_{j_i}(\sigma)$ del j_i-esimo programma è dato quindi dalle somma delle lunghezze dei programmi $j_1, j_2, \ldots, j_i$. L'obiettivo è trovare un ordine di esecuzione σ che minimizza la somma dei tempi di completamento di tutti i programmi, cioè $\sum_{j=1}^{n} C_j(\sigma)$.

(a) Dare un semplice algoritmo greedy per questo problema, e valutarne la complessità.

(b) Dimostrare la proprietà di scelta greedy dell'algoritmo del punto (a), cioè che esiste un ordine di esecuzione ottimo $\sigma^\star$ che contiene la scelta greedy.

MAKESPAN (C)

-> N = LENGTH (C)

-> SORT (C)

-> C = {C_1}

GREEDY-SEL(S, f)

```
n = S.length
A = {a_1}
last = 1 // indice dell'ultima attività selezionata
for m = 2 to n
      if s_m ≥ f_last
            A = A ∪ {a_m}
            last = m
return A
```


1 2 3 4 5 -> 10

min -> 10 (1 2 3 4) -> 4 PROGRAMMI

1 2 3 4 5 < 10

FOR $M = 2$ TO N

GIÀ GREEDY IL PRIMO

PARTO DA 2

IF $C_M \geq C_{LAST}$

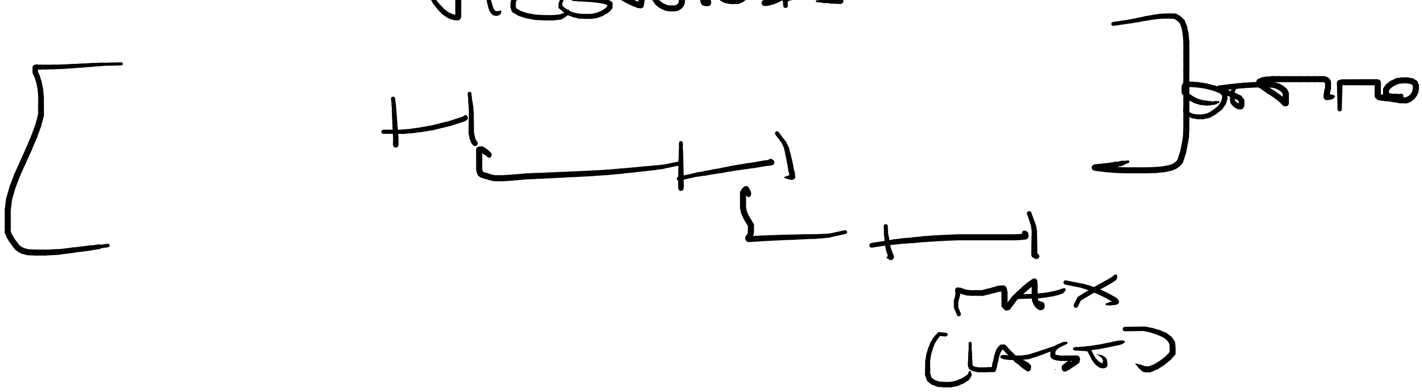
$C = C \cup \{C_M\}$

$LAST = M$

RETURN C

```
GREEDY-SEL(S, f)
n = S.length
A = {a_1}
last = 1 // indice dell'ultima attività selezionata
for m = 2 to n
    if s_m ≥ f_last
        A = A ∪ {a_m}
        last = m
return A
```

VICEVERSA



Esercizio 2 (10 punti) Abbiamo n programmi da eseguire sul nostro computer. Ogni programma j, dove $j \in \{1, 2, \ldots, n\}$, ha lunghezza ℓ_j, che rappresenta la quantità di tempo richiesta per la sua esecuzione. Dato un ordine di esecuzione $\sigma = j_1, j_2, \ldots, j_n$ dei programmi (cioè, una permutazione di $\{1, 2, \ldots, n\}$), il tempo di completamento $C_{j_i}(\sigma)$ del j_i-esimo programma è dato quindi dalle somma delle lunghezze dei programmi $j_1, j_2, \ldots, j_i$. L'obiettivo è trovare un ordine di esecuzione σ che minimizza la somma dei tempi di completamento di tutti i programmi, cioè $\sum_{j=1}^{n} C_j(\sigma)$.

(a) Dare un semplice algoritmo greedy per questo problema, e valutarne la complessità.

(b) Dimostrare la proprietà di scelta greedy dell'algoritmo del punto (a), cioè che esiste un ordine di esecuzione ottimo $\sigma^\star$ che contiene la scelta greedy.

CUT-AND-PASTE

(b) La scelta greedy consiste nello scegliere, come prossimo programma da eseguire, quello di lunghezza minima. Sia $\sigma^\star$ una soluzione ottima. Se il programma di lunghezza minima è il primo in $\sigma^\star$, abbiamo finito. Consideriamo quindi il caso in cui il programma di lunghezza minima sia in posizione $k > 1$ in $\sigma^\star$. Costruiamo una nuova soluzione σ' scambiando, in $\sigma^\star$, il k-esimo programma con il primo. Possiamo osservare che:

→ ⋆ = GREEDY

- l'insieme dei primi k programmi $j_1, j_2, \ldots, j_k$ è lo stesso in $\sigma^\star$ e σ', quindi il k-esimo programma ha lo stesso tempo di completamento in $\sigma^\star$ e σ'; lo stesso vale per tutti i programmi successivi al k-esimo, visto che lo scambio non influisce su di loro;
- per quanto riguarda tutti gli altri programmi, cioè quelli fino alla posizione $k-1$, questi hanno un tempo di completamento inferiore o uguale in σ', perché lo scambio può solo avere ridotto la lunghezza del primo programma.

Quindi

$$\sum_{j=1}^{n} C_j(\sigma') \leq \sum_{j=1}^{n} C_j(\sigma^\star);$$

1 2 3 ≤ 6

siccome $\sigma^\star$ è una soluzione ottima, allora deve valere che

$$\sum_{j=1}^{n} C_j(\sigma') = \sum_{j=1}^{n} C_j(\sigma^\star),$$

6 = 6

cioè anche σ' è una soluzione ottima.

Esercizio 2 (9 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 $0 \leq i \leq j \leq n-1$, definita come segue:

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

RICORRENZA SUI COSTI

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

(a) Fornire un algoritmo iterativo bottom-up per il calcolo di m. → ITERATIVO

(b) Valutare la complessità *esatta* dell'algoritmo, associando costo unitario alle moltiplicazioni tra numeri interi e costo nullo a tutte le altre operazioni.

MEMOIZZATO → RICORSIVO

COMPUTE(A,B)
→ N = LENGTH(A)
→ M = -INFTY

FOR I = 1 TO N-1
C[I, N-1] = A[I]
M = MAX(M, C[I, N-1])
FOR J = 1 TO N-1
C[0, J] = B[J]
M = MAX(M, C[0, J])

Esercizio 2 (9 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 $0 \leq i \leq j \leq n-1$, definita come segue:

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

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

(a) Fornire un algoritmo iterativo bottom-up per il calcolo di m.

(b) Valutare la complessità *esatta* dell'algoritmo, associando costo unitario alle moltiplicazioni tra numeri interi e costo nullo a tutte le altre operazioni.

FOR I = 1 TO N-2 (SCENDO) [BOTTOM]

FOR J = N-2 DOWNTO 1 (SALGO) [UP]

$C[I,J] = C(I-1,J-1) \odot C(I,J+1)$

$M = MAX(M, C[I,J])$

RETURN M

1 MOLTIPLICAZIONE

$$\sum_{i=1}^{n-2} \sum_{j=1}^{n-2} 1 = \sum_{i=1}^{n-2} (\underbrace{n-1-i}_{n-3}) = \sum_{k=1}^{n-2} k$$

$$\sum_{k=1}^{n-2} k = \frac{n(n-1)}{2} = \frac{(n-2)(n-1)}{2}$$

GAUSS

```
COMPUTE(a,b)
n <- length(a)
m = -infinito
for i=1 to n-1 do
   C[i,n-1] <- a_i
   m <- MAX(m,C[i,n-1])
for j=0 to n-1 do
   C[0,j] <- b_j
   m <- MAX(m,C[0,j])
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 <- MAX(m,C[i,j])
return m
```

INIT

INIT

CALCOLO

(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 (9 punti) Sia $n > 0$ un intero. Si consideri la seguente ricorrenza $M(i, j)$ definita su tutte le coppie (i, j) con $1 \le i \le j \le n$:

$$M(i,j) = \begin{cases} 1 & \text{se } i = j, \\ 2 & \text{se } j = i+1, \\ M(i+1, j-1) \cdot M(i+1, j) \cdot M(i, j-1) & \text{se } j > i+1. \end{cases}$$

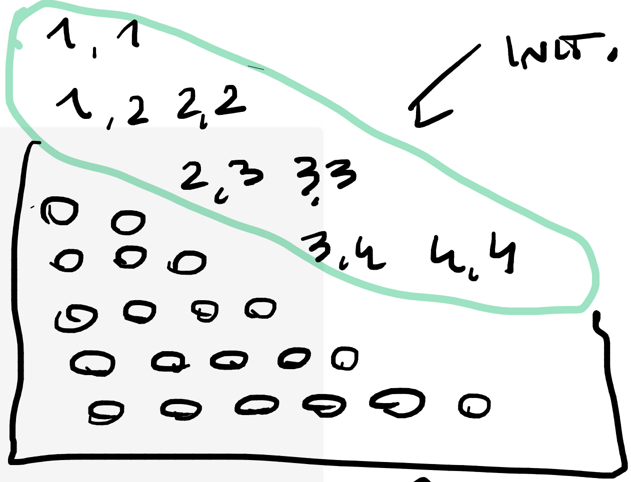
INIT / REC

1. Scrivere una coppia di algoritmi INIT_M(n) e REC_M(i, j) per il calcolo memoizzato di $M(1, n)$.
2. Calcolare il numero esatto $T(n)$ di moltiplicazioni tra interi eseguite per il calcolo di $M(1, n)$.

Soluzione:

1. Pseudocodice:

INIT.




```
INIT_M(n)
if n=1 then return 1
if n=2 then return 2
for i=1 to n-1 do
   M[i,i] = 1
   M[i,i+1] = 2
M[n,n] = 1
for i=1 to n-2 do
   for j=i+2 to n do
      M[i,j] = 0
return REC_M(1,n)

REC_M(i,j)
if M[i,j] = 0 then
   M[i,j] = REC_M(i+1,j-1) * REC_M(i+1,j) * REC_M(i,j-1)
return M[i,j]
```

RISULTATO

SOL.

2.

$$T(n) = \sum_{i=1}^{n-2} \sum_{j=i+2}^{n} 2 = 2 \sum_{i=1}^{n-2} n - i - 1 = 2 \sum_{k=1}^{n-2} k = \frac{(n-2)(n-1)}{\cancel{2}} \cdot \cancel{2} = (n-2)(n-1)$$

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

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