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

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

— FOR

— FOR

← RIEMPIMENTO

1. Si fornisca il codice di un algoritmo iterativo bottom-up Z(A) che, data in input la stringa A restituisca in uscita il valore $z(n,1)$. (vedo (n,1) e la scansione è per colonna)
2. Si valuti il numero esatto $T_Z(n)$ di moltiplicazioni tra interi eseguite dall'algoritmo sviluppato al punto (1).

Reverse column major:
a11 a12 a13

I ↓ S ↑

```
Z(A)
N = LENGTH(A)
FOR I=2 TO N          ← CICLO CON I
Z[I,N] = A_(n+1-i)      ← S=N
Z[1,i] = A_i
   FOR I=2 TO N-1
     FOR S=N-1 DOWNTO 2
       Z[I,S] = z(i-1,j) · z(i,j+1) · z(i-1,j+1)
   RETURN Z(N,1)
```

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.

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

Piuttosto che usare per l'inizializzazione due indici, se ne usa solo uno.
Quando si ha "j", si sa che "i" vale 1, da cui i=j e quindi [1, i]
Quando invece si ha "n+1-i" come indice, si vede che "j" è =n e quindi avremo che per [i, n] avremo [n+1-i].

Vedendo che "j" parte da n e invece "i" parte da 1,
per effettuare una scansione giusta per colonne (avendo che la prima colonna/riga è stata inizializzata dal caso base), allora "j" parte da (n-1) piuttosto che da (n) e "i" parte da 2 piuttosto che da (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 $z[1,n]$.

(2) N. ESATTO DI MOLTIPLICAZIONI...

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

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 $1 \le i \le j \le n$:

REC_M REC_M

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

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 $M(1, n)$.

```
INIT_M(N)
FOR I=1 TO N
 M[I,I]=1
 M[I,I+1]=2
FOR I=1 TO N-1
 FOR J=2 TO N-1
   M[I,J]=0
```

```
REC_M(N)
IF (M(I,J)==0)
M(I,J)=
 REC_M(I+1,J-1)...
```

```
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)
```

Siccome viene già passata la lunghezza, non serve salvarla.
Similmente, anche n=1 e n=2 vengono dai due casi base (array di 1/2 elementi max), quindi o ha un elemento oppure ne ha due soli

(ultimo elemento diagonale inizializzato)

Ricordandosi che:
- "i" va da i ad (n-1), quindi diventa perché abbiamo già inizializzato, (n-2) la scansione.
- "j" va da (n-1) a 0 compresi, quindi dato che abbiamo inizializzato, parte da (n-2)

$K \rightarrow \frac{n(n-1)}{2}$

GAUSS

Una volta inizializzato secondo i casi base, tutto il resto della matrice viene riempita di zeri. Poi, si considera, essendo una scansione per diagonale, noi partiamo dal basso e riempiamo solo la parte sopra delle diagonali principali ovviamente riempita anche questa di zeri.

```
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]
```

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

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

Esercizio 2 (9 punti) Lungo una strada ci sono, in vari punti, n parcheggi liberi e n auto. Un posteggiatore ha il compito di parcheggiare tutte le auto, e lo vuole fare minimizzando lo spostamento totale da fare. Formalmente, dati n valori reali $p_1, p_2, \ldots, p_n$ e altri n valori reali $a_1, a_2, \ldots, a_n$, che rappresentano

le posizioni lungo la strada rispettivamente di parcheggi e auto, si richiede di assegnare ad ogni auto a_i un parcheggio $p_{h(i)}$ minimizzando la quantità

$$\left[\sum_{i=1}^{n} |a_i - p_{h(i)}|.\right]$$

1. Si consideri il seguente algoritmo greedy. Si individui la coppia (auto, parcheggio) con la minima differenza. Si assegni quell'auto a quel parcheggio. Si ripeta con le auto e i parcheggi restanti fino a quando tutte le auto sono parcheggiate. Dimostrare che questo algoritmo non è corretto, esibendo un controesempio.

2. Si consideri il seguente algoritmo greedy. Si assuma che i valori $p_1, p_2, \ldots, p_n$ e $a_1, a_2, \ldots, a_n$ siano ordinati in modo non decrescente. Si produca l'assegnazione $(a_1, p_1), (a_2, p_2), \ldots, (a_n, p_n)$. Dimostrare la correttezza di questo algoritmo per il caso $n = 2$.

1

10 9 8 → 3 PARCHEGGI
7 6 5

20 30 40
10 15 20

10 − 9 = 1
14 − 5 = 9 → 10

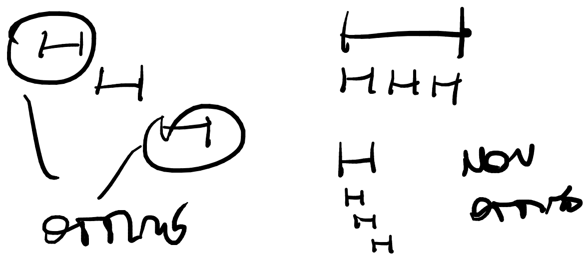
ve

1. Si consideri il seguente input:

$$p_1 = 5, p_2 = 10 \text{ e } a_1 = 9, a_2 = 14.$$

L'algoritmo produce l'assegnazione $(a_1, p_2), (a_2, p_1)$, che ha costo $1 + 9 = 10$, mentre l'assegnazione $(a_1, p_1), (a_2, p_2)$ ha costo $4 + 4 = 8$.

$$\left[\begin{array}{l} 9 - 5 = 4 \\ 14 - 10 = 4 \end{array}\right] \to 8$$



2. Si consideri il seguente algoritmo greedy. Si assuma che i valori $p_1, p_2, \ldots, p_n$ e $a_1, a_2, \ldots, a_n$ siano ordinati in modo non decrescente. Si produca l'assegnazione $(a_1, p_1), (a_2, p_2), \ldots, (a_n, p_n)$. Dimostrare la correttezza di questo algoritmo per il caso $n = 2$.

$[a_1 | p_1]$ e $[a_2 | p_2]$

Dal ragionamento detto, matematicamente, si vede che basta
prendere un qualsiasi ordinamento tra le due auto e i due parcheggi di due gen
e si esprime la somma in termini matematici (l'idea concreta è quella spiegata c

2. Ci sono vari casi possibili:
   (a) Caso $a_1 \leq p_1 \leq p_2 \leq a_2$
   - l'assegnazione $(a_1, p_1), (a_2, p_2)$ ha costo $p_1 - a_1 + a_2 - p_2 = (a_2 - a_1) - (p_2 - p_1)$
   - l'assegnazione $(a_1, p_2), (a_2, p_1)$ ha costo $p_2 - a_1 + a_2 - p_1 = (a_2 - a_1) + (p_2 - p_1)$; siccome $p_2 - p_1 \geq 0$, questa assegnazione ha costo non inferiore rispetto alla precedente

   (b) Caso $a_1 \leq p_1 \leq a_2 \leq p_2$
   - l'assegnazione $(a_1, p_1), (a_2, p_2)$ ha costo $p_1 - a_1 + p_2 - a_2 = (p_2 - a_1) - (a_2 - p_1)$
   - l'assegnazione $(a_1, p_2), (a_2, p_1)$ ha costo $p_2 - a_1 + a_2 - p_1 = (p_2 - a_1) + (a_2 - p_1)$; siccome $a_2 - p_1 \geq 0$, questa assegnazione ha costo non inferiore rispetto alla precedente

   (c) Caso $a_1 \leq a_2 \leq p_1 \leq p_2$
   - l'assegnazione $(a_1, p_1), (a_2, p_2)$ ha costo $p_1 - a_1 + p_2 - a_2 = (p_2 - a_1) + (p_1 - a_2)$
   - l'assegnazione $(a_1, p_2), (a_2, p_1)$ ha costo $p_2 - a_1 + p_1 - a_2 = (p_2 - a_1) + (p_1 - a_2)$, uguale a quello precedente

   Tutti gli altri casi sono simmetrici e si dimostrano nella stessa maniera.

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.

$\sigma = \{S_1, S_2, S_3, S_4\ S_5\}$ s.t. $\sum_{s=1}^{n} C_s = \{3\ \ 1\ \ 2\ \ 4\ \ 0.3\}$

MAKESPAN (A) 0.3 1 2 3 4

(1) N = LENGTH (A)

(2) SORT_ND(A) ← //NON DECRESCENTE

(3) C_OPT = {A_1} $\sum \sigma = 10$ → COMPLETION TIME

(4) FOR I=1 TO N

(5) IF $C_{MAX} - C_i \leq$ C_OPT

(6) C_OPT = C_OPT $\cup \{C_i\}$ //SALVO NUOVO MIN.

(7) RETURN C_OPT

(a) Ordina i programmi per lunghezza crescente. Complessità: $O(n \log n)$.

(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:

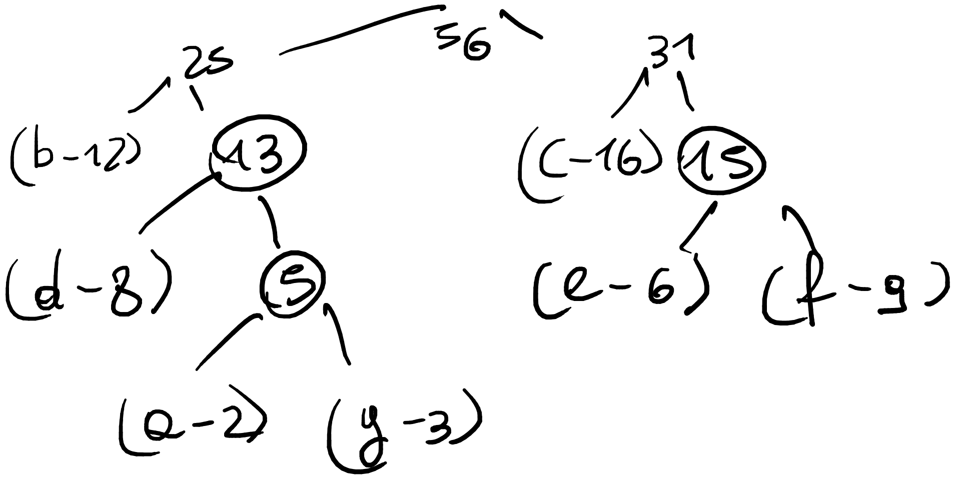
- l'insieme dei primi k programmi $j_1, j_2, \ldots, j_k$ è lo stesso in $\sigma^\star$ e σ', quindi il k-esimo programma ha lo stesso tempo di completamento in $\sigma^\star$ e σ'; lo stesso vale per tutti i programmi successivi al k-esimo, visto che lo scambio non influisce su di loro;
- per quanto 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
$$\sum_{j=1}^{n} C_j(\sigma') \leq \sum_{j=1}^{n} C_j(\sigma^\star);$$
siccome $\sigma^\star$ è una soluzione ottima, allora deve valere che
$$\sum_{j=1}^{n} C_j(\sigma') = \sum_{j=1}^{n} C_j(\sigma^\star),$$
cioè anche σ' è una soluzione ottima.

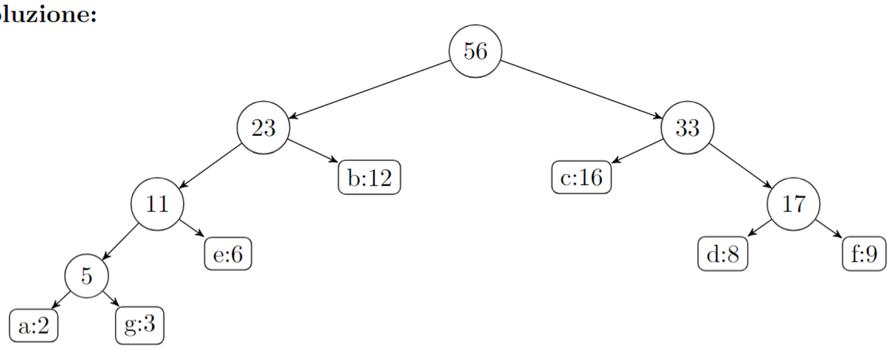
Domanda 45 Indicare il codice prefisso ottenuto utilizzando l'algoritmo di Huffmann per l'alfabeto $\{a, b, c, d, e, f, g\}$, supponendo che ogni simbolo appaia con le seguenti frequenze.

a	b	c	d	e	f	g
2	12	16	8	6	9	3

Spiegare il processo di costruzione del codice.



Soluzione:

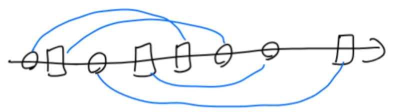


Esercizio: matching sulla linea → ESEMPIO NOTEVOLE

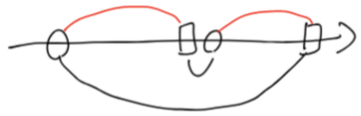
Sia $S = \{s_1, s_2, \dots, s_n\}$ un insieme di punti ordinati sulla retta reale, rappresentanti dei server. Sia $C = \{c_1, c_2, \dots, c_n\}$ un insieme di punti ordinati sulla retta reale, rappresentanti dei client. Il costo di assegnare un client c_i ad un server s_j è $|c_i - s_j|$. Fornire un algoritmo greedy che assegna ogni client ad un server distinto e che minimizzi il costo totale (equiv., medio) dell'assegnamento.

1. ORDINO S e C
2. FOR
3. IF $C_i - S_S <$ MIN
4. MIN $= C_i - S_S$
5. RETURN MIN

esempio:



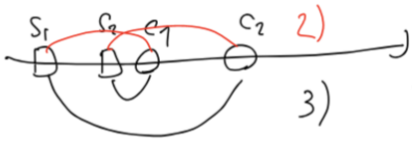
client al server + vicino, partendo dalla coppia client-server con distanza minore

NON funziona: 

$\sqrt{}$ 2) $c_1 - s_1$, $c_2 - s_2$, ...

(equiv. $c_n - s_n$, $c_{n-1} - s_{n-1}$, ...)

~~3)~~ c_1 al server + vicino; c_2 al server + vicino



$\Rightarrow$ sono algoritmi $\neq$

NON funziona: