

```

    else
        L[j] = A[j]

max = A[1]
for j=2 to n
    if L[j]>max
        max = L[j]

return max

```

Se vogliamo anche il sottoarray, occorre ricordare il massimo e dove viene raggiunto, lo facciamo mediante un array $S[1..n]$ tale che $S[j]$ contenga l'indice dal quale inizia il sottoarray massima che termina con $A[j]$.

```

maxsum (A, n)
L[1] = A[1]
S[1] = 1

for j=2 to n
    if (L[j-1] > 0)
        L[j] = L[j-1] + A[j]
        S[j] = S[j-1]
    else
        L[j] = A[j]
        S[j] = j

max = 1
for j=2 to n
    if L[j]>max
        max = j

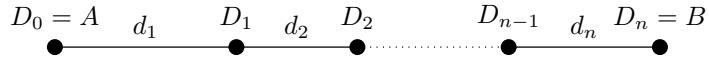
# returns the first and last index of the substring
return S[max], max

```

La complessità è $\Theta(n)$.

9 Greedy

Esercizio 33 Si supponga di voler viaggiare dalla città A alla città B con un'auto che ha un'autonomia pari a d km. Lungo il percorso si trovano $n - 1$ distributori D_1, \dots, D_{n-1} , a distanze di d_1, \dots, d_n km ($d_i \leq d$) come indicato in figura



L'auto ha inizialmente il serbatoio pieno e l'obiettivo è quello di percorrere il viaggio da A a B , minimizzando il numero di soste ai distributori per il rifornimento.

- Introdurre la nozione di soluzione per il problema e di costo della una soluzione. Mostrare che vale la proprietà della sottostruttura ottima e individuare una scelta che gode della proprietà della scelta greedy.

- ii. Sulla base della scelta greedy individuata al passo precedente, fornire un algoritmo greedy `stop(d, n)` che dato in input l'array delle distanze $d[1..n]$ restituisce una soluzione ottima.
- iii. Valutare la complessità dell'algoritmo.

Soluzione:

La scelta ottima del problema consiste nel trovare un insieme di soste tali che la loro lunghezza sia minima.

Soluzione. In generale, la specifica del problema consiste in una sequenza di soste possibili $D_0 (= A), D_1, \dots, D_n (= B)$. Per una coppia di soste D_i, D_j con $i \leq j$ poniamo $d_{i,j} = \sum_{h=i+1}^j d_h$. Quindi una soluzione del problema è una sottosequenza di soste che porta dal punto iniziale al punto finale, senza percorrere tratti di lunghezza maggiore di d ovvero:

$$S = D_{i_0} \dots D_{i_k}$$

(con $i_0 < i_1 < \dots < i_k$) tali che $D_{i_0} = A$ e $D_{i_k} = B$, per ogni $j \in \{0, \dots, k-1\}$ vale $d_{i_j i_{j+1}} \leq d$. Il costo $c(S)$ è il numero $k-1$ di soste.

Sottostruttura ottima. Sia $S = D_{i_0} \dots D_{i_k}$ una soluzione ottima per il problema D_0, D_1, \dots, D_n . Se consideriamo il sottoproblema di andare da D_{i_1} a D_n , ovvero $D_{i_1}, D_{i_1+1} \dots D_n$, allora $S_1 = D_{i_1}, \dots, D_{i_k}$ è una soluzione ottima. Infatti, è chiaramente una soluzione. Inoltre, se ci fosse una soluzione migliore S'_1 , con un numero inferiore di soste per il sottoproblema, ovvero $c(S'_1) < c(S_1)$, aggiungendo la sosta D_{i_0} otterremmo una soluzione S' migliore di S per il problema originale dato che $c(S') = c(S'_1) + 1 < c(S_1) + 1 = c(S)$. Significa che, avendo scelto bene prima, aggiungendo altre soste, la scelta rimane minima.

Scelta greedy. Per il problema D_0, D_1, \dots, D_n , si fissa necessariamente $i_0 = 0$, e la scelta greedy consiste nel raggiungere la sosta più lontana a distanza minore o uguale di d , ovvero definire $i_1 = \max\{j \mid d_{0,j} \leq d\}$.

Data una soluzione ottima per il problema $S = D_{j_0} \dots D_{j_k}$, certamente $j_0 = 0 = i_0$ e $j_1 \leq i_1$. Quindi è immediato verificare che anche $S' = D_{j_0} D_{i_1} D_{j_2} \dots D_{j_k}$ è una soluzione per il problema D_0, D_1, \dots, D_n , ed è ottima, dato che $c(S') = c(S)$. (Si noti che, più precisamente, in prima istanza si potrebbe pensare che D_{i_1} potesse essere anche oltre D_{j_2} , ma questo darebbe una soluzione migliore di quella ottima, portando ad un assurdo).

Algoritmo. Ne segue l'algoritmo che riceve in input la sequenza di distanze nella forma di un array $d[1..n]$ e restituisce un array $S[1..n-1]$ con le soste scelte (la prima e l'ultima sono scelte sempre, non serve indicarlo).

```
stop(d, n)
  dist = d[1]           // distanza percorsa
  for i=2 to n
    if dist + d[i] > d
      S[i-1]=1
      dist=d[i]
    else
      S[i-1] = 0
  return S
```



Per scelta greedy, prendiamo la sosta con distanza <= a quella attuale, tale da capire caso per caso quale i conviene di più.

L'algoritmo ricalca proprio la selezione delle attività e:

- prende come distanza la prima, per inizializzazione
- se la somma tra la prima distanza e quella attuale è > della sequenza delle distanze, allora abbiamo la distanza buona e salviamo la sosta con la distanza ottima (sapendo che ci siamo fermati "prima", quindi $S[i-1]=1$)
- se invece non ci siamo fermati prima, avremo $S[i-1] = 0$

La complessità è $\Theta(n)$.

Esercizio 34 L'ufficio postale offre un servizio di ritiro pacchi in sede su prenotazione. Il destinatario, avvisato della presenza del pacco, deve comunicare l'orario preciso al quale si recherà allo

sportello. Sapendo che gli impiegati dedicano a questa mansione turni di un'ora, con inizio in un momento qualsiasi, si chiede di scrivere un algoritmo che individui l'insieme minimo di turni di un'ora sufficienti a soddisfare tutte le richieste. Più in dettaglio, data una sequenza $\vec{r} = r_1, \dots, r_n$ di richieste, dove r_i è l'orario della i -ma prenotazione, si vuole determinare una sequenza di turni $\vec{t} = t_1, \dots, t_k$, con t_j orario di inizio del j -mo turno, che abbia dimensione minima e tale che i turni coprano tutte le richieste.

- i. Formalizzare la nozione di soluzione per il problema e il relativo costo. Mostrare che vale la proprietà della sottostruttura ottima e individuare una scelta che gode della proprietà della scelta greedy.
- ii. Sulla base della scelta greedy individuata al passo precedente, fornire un algoritmo `time(R,n)` che dato in input l'array delle richieste $r[1..n]$ restituisce una soluzione ottima.
- iii. Valutare la complessità dell'algoritmo.

L'idea è di crearsi un array/sequenza di turni tale che siano con un costo "k" minore possibile.

Soluzione: Una soluzione è una sequenza $\vec{t} = t_1, \dots, t_k$ tale che per ogni $i = 1, \dots, n$ l'istante $r_i \in I(t_j)$ per qualche $j = 1, \dots, k$, dove $I(t_j)$ indica l'intervallo di un'ora con inizio in t_j . Il costo è k .

La sottostruttura ottima vale avendo le richieste e, se abbiamo fatto la scelta minore in partenza tra turni e richieste, di sicuro questa tuttora si mantiene.

Sottostruttura ottima Vale la sottostruttura ottima. Infatti se $\vec{t} = t_1 q \dots, t_k$ è ottima per le richieste $\vec{r} = r_1, \dots, r_n$ allora t_2, \dots, t_k è ottima per il sottoproblema \vec{r}' che comprende le richieste r_i tali che $r_i \notin I(t_1)$. Infatti, se ci fosse un insieme di turni di dimensione minore di $k-1$ per servire le richieste in \vec{r}' , aggiungendo t_1 otterremmo una soluzione del problema originale \vec{r} migliore di \vec{t} .

Scelta greedy Assumiamo che la lista di richieste $\vec{r} = r_1, \dots, r_n$ sia ordinata in modo crescente. La scelta greedy consiste nel considerare come primo turno $t_1 = r_1$.

Esiste sempre una soluzione ottima che la contiene. Infatti se $\vec{t}' = t'_1 \dots t'_k$ è una qualsiasi soluzione ottima e supponiamo che i turni siano ordinati anch'essi in senso crescente, allora certamente $t'_1 \leq r_1$, dato che la prima richiesta deve essere servita. Quindi se sostituiamo t'_1 con $t_1 = r_1$, otteniamo una nuova soluzione (tutte le richieste servite da t'_1 sono anche servite da t_1 !), anch'essa ottima.

Ne segue l'algoritmo che riceve in input l'array delle richieste $r[1..n]$ (che si assume non vuoto), e fornisce in uscita $t[1, k]$.

```
time(r,n)
  t[1] = r[1]
  turni=1
  for i = 2 to n
    if t[turni] < r[i]
      turni++
      t[turni] = r[i]
  return t
```

Salviamo nel vettore dei turni, la prima posizione delle richieste, in maniera tale da avere almeno una richiesta assegnata, ovviamente nel caso in cui non ce ne siano altre.
 La scelta greedy consiste nel scegliere la prima richiesta ed assegnare le altre, conoscendo già il tempo delle richieste in modo crescente (considerando ovviamente tutti i turni).
 Quindi, se abbiamo il turno con una richiesta che ci mette poco, la assegniamo e ritorniamo l'array di turni ottimi.

La complessità è $\Theta(n)$.

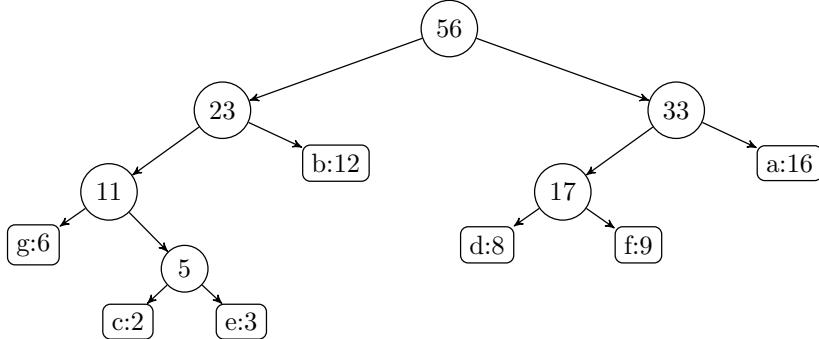
Domanda 39 Indicare il codice prefisso ottenuto utilizzando l'algoritmo di Huffmann per l'alfabeto $\{a, b, c, d, e, f, g\}$, supponendo che ogni simbolo appaia con le seguenti frequenze.

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

Huffman prevede di costruire un albero partendo dalle foglie del livello più basso che hanno i valori numerici più bassi e, per somma, si ottengono le radici composte a loro volta per somma da altri nodi di valore numerico crescente.

Spiegare il processo di costruzione del codice.

Soluzione:

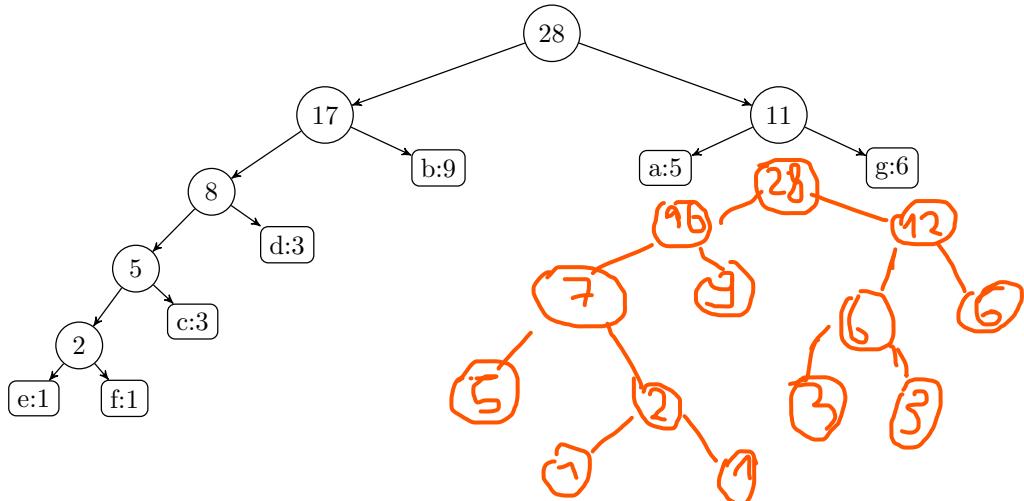


Domanda 40 Indicare il codice prefisso ottenuto utilizzando l'algoritmo di Huffmann per l'alfabeto $\{a, b, c, d, e, f, g\}$, supponendo che ogni simbolo appaia con le seguenti frequenze.

a	b	c	d	e	f	g
5	9	3	3	1	1	6

Spiegare il processo di costruzione del codice.

Soluzione:

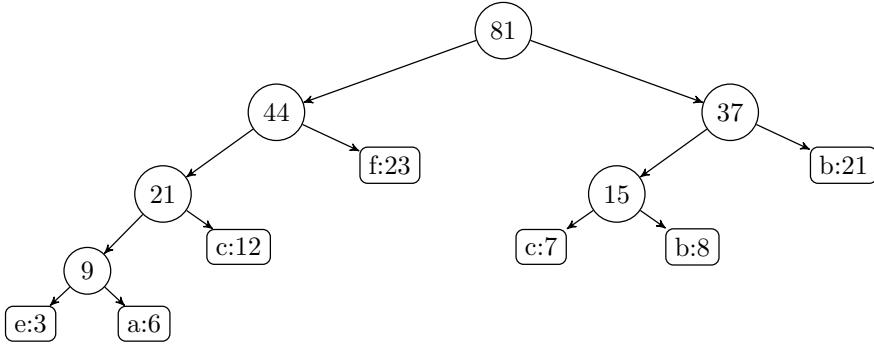


Domanda 41 Indicare il codice prefisso ottenuto utilizzando l'algoritmo di Huffmann per l'alfabeto $\{a, b, c, d, e, f, g\}$, supponendo che ogni simbolo appaia con le seguenti frequenze.

a	b	c	d	e	f	g
6	21	12	8	3	23	8

Spiegare il processo di costruzione del codice.

Soluzione:

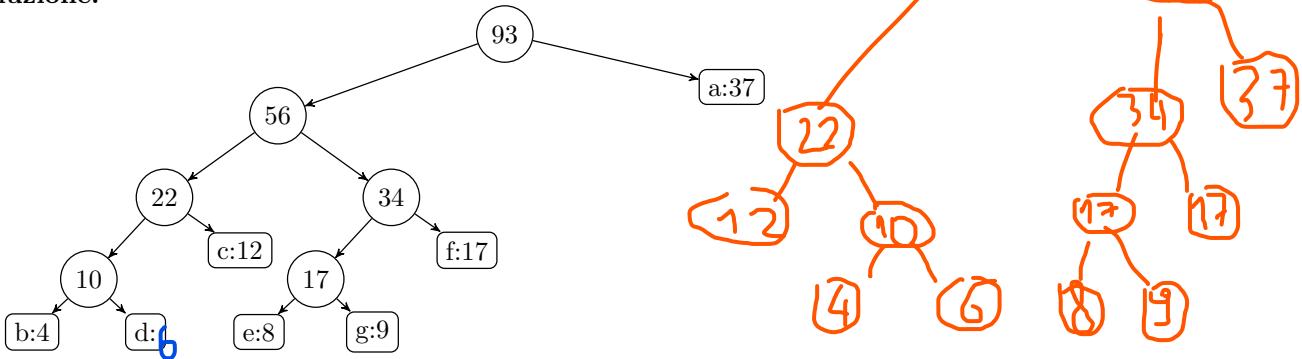


Domanda 42 Indicare il codice prefisso ottenuto utilizzando l'algoritmo di Huffmann per l'alfabeto $\{a, b, c, d, e, f, g\}$, supponendo che ogni simbolo appaia con le seguenti frequenze.

a	b	c	d	e	f	g
37	4	12	6	9	17	8

Spiegare il processo di costruzione del codice.

Soluzione:

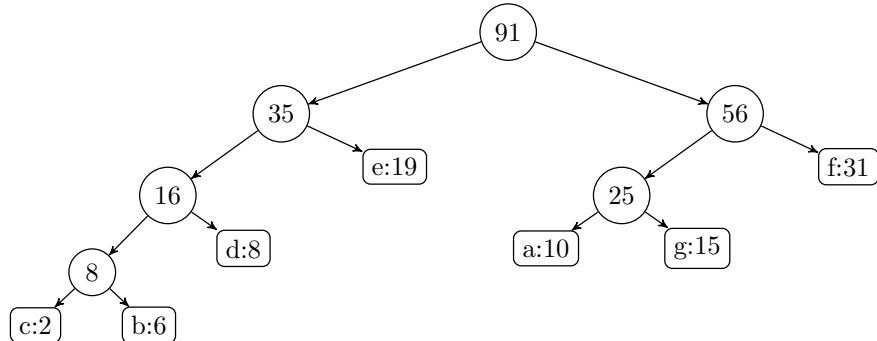


Domanda 43 Indicare il codice prefisso ottenuto utilizzando l'algoritmo di Huffmann per l'alfabeto $\{a, b, c, d, e, f, g\}$, supponendo che ogni simbolo appaia con le seguenti frequenze.

a	b	c	d	e	f	g
10	6	2	8	19	31	15

Spiegare il processo di costruzione del codice.

Soluzione:

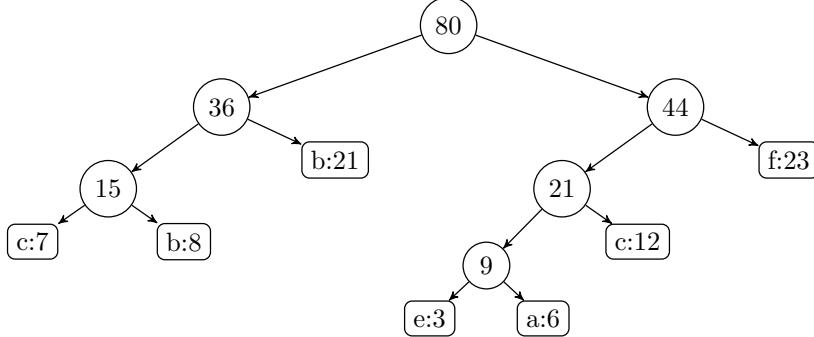


Domanda 44 Indicare il codice prefisso ottenuto utilizzando l'algoritmo di Huffmann per l'alfabeto $\{a, b, c, d, e, f, g\}$, supponendo che ogni simbolo appaia con le seguenti frequenze.

a	b	c	d	e	f	g
3	8	7	12	6	23	21

Spiegare il processo di costruzione del codice.

Soluzione:

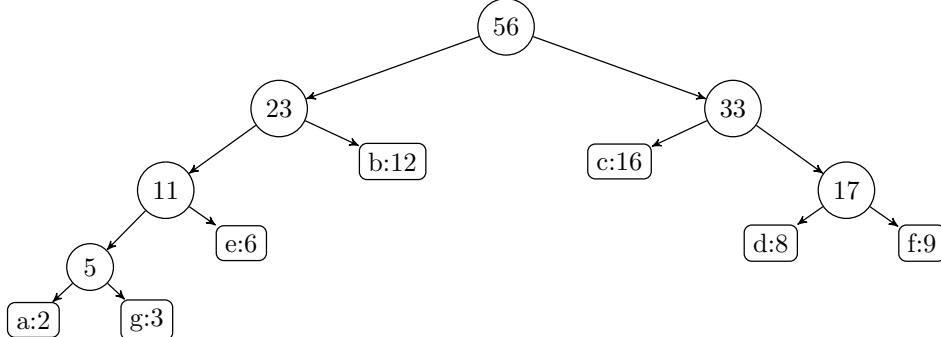


Domanda 45 Indicare il codice prefisso ottenuto utilizzando l'algoritmo di Huffmann per l'alfabeto $\{a, b, c, d, e, f, g\}$, supponendo che ogni simbolo appaia con le seguenti frequenze.

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

Spiegare il processo di costruzione del codice.

Soluzione:



Domanda B (6 punti) Si consideri un insieme di 7 attività $a_i, 1 \leq i \leq 7$, caratterizzate dai seguenti vettori \mathbf{s} e \mathbf{f} di tempi di inizio e fine:

$$\mathbf{s} = (1, 4, 2, 3, 7, 8, 11)$$

$$\mathbf{f} = (3, 6, 9, 10, 11, 12, 13).$$

Determinare l'insieme di massima cardinalità di attività mutuamente compatibili selezionato dall'algoritmo greedy GREEDY_SEL visto in classe. Motivare il risultato ottenuto descrivendo brevemente l'algoritmo.

Soluzione: Si considerano le attività ordinate per tempo di fine, e ad ogni passo si sceglie l'attività che termina prima, rimuovendo quelle incompatibili. Si ottiene così l'insieme di attività $\{a_1, a_2, a_5, a_7\}$.

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, \dots, p_n e altri n valori reali a_1, a_2, \dots, 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

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

L'idea di questo algoritmo, se fosse in codice, sarebbe simile a Metric Matching, quindi cerco di minimizzare la differenza partendo dalla fine (quindi, vedendo quante auto e quanti parcheggi ci sono) e verificando quale, a coppia, ha la minima differenza.

1. Si consideri il seguente algoritmo greedy. Si individui la coppia (auto, parcheggio) con la minima differenza. Si assegna 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 controsenso.
2. Si consideri il seguente algoritmo greedy. Si assuma che i valori p_1, p_2, \dots, p_n e a_1, a_2, \dots, a_n siano ordinati in modo non decrescente. Si produca l'assegnazione $(a_1, p_1), (a_2, p_2), \dots, (a_n, p_n)$. Dimostrare la correttezza di questo algoritmo per il caso $n = 2$.

Soluzione:

1. Si consideri il seguente input:

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

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 $4 + 4 = 8$.

2. Ci sono vari casi possibili:

(a) Caso $a_1 \leq p_1 \leq p_2 \leq a_2$

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

- 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.

Questo significa che l'assegnazione parte dagli ultimi parcheggi e dalle ultime auto (quindi, massima somma al contrario significa minima differenza). Essendo elementi a coppia, si possono "mischiare" le assegnazioni come si vede sotto.

Domanda B (6 punti) Si consideri un insieme di 8 attività $a_i, 1 \leq i \leq 8$, caratterizzate dai seguenti vettori \mathbf{s} e \mathbf{f} di tempi di inizio e fine:

$$\mathbf{s} = (1, 1, 2, 4, 2, 5, 6, 9)$$

$$\mathbf{f} = (3, 4, 5, 6, 7, 9, 10, 12).$$

[Si veda la risoluzione di pag. 52 per capire come ragionarlo](#)

Determinare l'insieme di massima cardinalità di attività mutuamente compatibili selezionato dall'algoritmo greedy GREEDY_SEL visto in classe. Motivare il risultato ottenuto descrivendo brevemente l'algoritmo.

Soluzione: Si considerano le attività ordinate per tempo di fine, e ad ogni passo si sceglie l'attività che termina prima, rimuovendo quelle incompatibili. Si ottiene così l'insieme di attività $\{a_1, a_4, a_7\}$.

Esercizio 2 (9 punti) Si consideri un file definito sull'alfabeto $\Sigma = \{a, b, c\}$, con frequenze $f(a), f(b), f(c)$. Per ognuna delle seguenti codifiche si determini, se esiste, un opportuno assegnamento di valori alle 3 frequenze $f(a), f(b), f(c)$ per cui l'algoritmo di Huffman restituisce tale codifica, oppure si argomenti che tale codifica non è mai ottenibile.

1. $e(a) = 0, e(b) = 10, e(c) = 11$

[Si veda la risoluzione di pag. 94 completa di questo esercizio](#)

2. $e(a) = 1, e(b) = 0, e(c) = 11$

3. $e(a) = 10, e(b) = 01, e(c) = 00$

Soluzione:

1. Questa codifica viene restituita dall'algoritmo di Huffman quando $f(b), f(c) < f(a)$: in questo caso i nodi associati a b e c vengono uniti creando un nuovo nodo interno, che poi viene unito al nodo associato ad a . Quindi, per esempio, $f(a) = 40, f(b) = 25, f(c) = 35$.
2. La codeword $e(a) = 1$ è un prefisso della codeword $e(c) = 11$, cioè questa codifica non è libera da prefissi, e quindi non è una codifica di Huffman.
3. L'albero associato a questa codifica non è pieno perché c'è un nodo interno con un solo figlio (quello nel cammino che porta alla foglia associata al carattere a). Quindi questa codifica non è ottima e quindi non è una codifica di Huffman.

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.

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: ogni sottoinsieme contiene al più 2 valori la cui somma è al più 1. Si chiede di minimizzare il numero di sottoinsiemi della partizione. Per fare ciò, la somma è al più 1 considerando per certo gli estremi (inferiore e superiore). Grazie a questi, sappiamo per certo che, prendendo il valore più grande (sup) e il valore più piccolo (inf), la scelta funziona.

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|           Inizializziamo l'insieme, la partizione e gli estremi.
P <- empty_set
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}}      Se l'estremo inf. è < dell'estremo sup.
    first <- first + 1                non abbiamo ancora salvato nulla in P (non abbiamo estr. inf)
  else                                allora salvo in P sia l'estremo inf che l'estremo sup
    P <- P U {{a_last}}              e incremento first. Salvandolo una volta sola, so che la somma è sempre > 1.
    last <- last - 1                Altrimenti,
  return P                            salvo solo l'estremo superiore migliore, la cui somma
                                      è sempre >= 1
```

Esempio:

1 2 3 4 5 6 7

P = (7, 6, 5, 4, 3, 2, 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 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 $S'_2 = S_1 \cup S_2 \setminus \{a_1, a_n\}$. $|S'_1| \leq 2$ 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 ammissibile e ancora ottima.

Spiegata in termini semplici: la scelta greedy consiste nel scegliere quello che sta agli estremi.

Esiste sempre una soluzione ottima in quanto il sottoinsieme che consideriamo è sempre formato da almeno due elementi (tolto il caso base) per cui tra questi ci siano i nostri estremi inf. e sup.

Andando a fare le differenze ottime (scegliendo di volta in volta gli estremi migliori), di sicuro avremo ancora una differenza ottima, perché sceglieremo gli estremi migliori (più piccolo per inf. e più grande per sup) in ogni momento.

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

- Dare un semplice algoritmo greedy per questo problema, e valutarne la complessità.
- Dimostrare la proprietà di scelta greedy dell'algoritmo del punto (a), cioè che esiste un ordine di esecuzione ottimo σ^* che contiene la scelta greedy.

Abbastanza laconico, ma l'idea è che, ordinando come detto, poi se la somma deve essere minima, facciamo un algoritmo che ricalca proprio la selezione delle attività, quindi prendendo il programma che completa prima.

Soluzione: La complessità è $O(n \log n)$ perché evidentemente il prof usa MergeSort

- Ordina i programmi per lunghezza crescente. Complessità: $O(n \log n)$.
- La scelta greedy consiste nello scegliere, come prossimo programma da eseguire, quello di lunghezza minima. Sia σ^* una soluzione ottima. Se il programma di lunghezza minima è il primo in σ^* , abbiamo finito. Consideriamo quindi il caso in cui il programma di lunghezza minima sia in posizione $k > 1$ in σ^* . Costruiamo una nuova soluzione σ' scambiando, in σ^* , il k -esimo programma con il primo. Possiamo osservare che:
 - l'insieme dei primi k programmi j_1, j_2, \dots, j_k è lo stesso in σ^* e σ' , quindi il k -esimo programma ha lo stesso tempo di completamento in σ^* e σ' ; lo stesso vale per tutti i programmi successivi al k -esimo, visto che lo scambio non influenza 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

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

siccome σ^* è una soluzione ottima, allora deve valere che

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

cioè anche σ' è una soluzione ottima.

Quindi:
 - caso base, abbiamo già il programma ottimo.
 - caso induttivo,
 avendo che ordiniamo i programmi, ogni nostro programma con tempo di completamento minimo
 ricalca la somma di programmi con completamento minimo,
 anche matematicamente.

Ciò detto, avremo per forza una serie ottima.

Per quanto riguarda la complessità, la ricorrenza è del tipo $T(n) = T(n/2) + c$, dato che ad ogni chiamata si dimezza la dimensione dell'array e il costo della parte non ricorsiva è costante. Per il Master Theorem (con $a = 1$, $b = 2$, $f(n) = c = \Theta(n^{\log_b a}) = \Theta(n^0) = \Theta(1)$) si conclude $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(\log n)$.

Esercizio 2 (9 punti) Sia n un intero positivo. Si consideri la seguente ricorrenza $M(i, j)$ definita su tutte le coppie (i, j) con $1 \leq i \leq j \leq n$:

$$M(i, j) = \begin{cases} 2 & \text{(caso base)} \\ 3 & \text{se } i = j, \\ M(i+1, j-1) \cdot M(i+1, j) + M(i, j-1) & \text{se } j = i+1, \\ & \text{se } j > i+1. \end{cases}$$

(vedo (1,n) e so che la scansione è per diagonale)

- (a) Si scriva una coppia di algoritmi INIT_M(n) e REC_M(i, j) per il calcolo memoizzato di $M(1, n)$.
- (b) Si calcoli il numero esatto $T(n)$ di moltiplicazioni tra interi eseguite per il calcolo di $M(1, n)$.

Soluzione:

(a) INIT_M(n)

```
if n=1 then return 2
if n=2 then return 3
for i=1 to n-1 do
    M[i, i] = 2
    M[i, i+1] = 3
    M[n, n] = 2
for i=1 to n-2 do
    for j=i+2 to n do
        M[i, j] = 0
return REC_M(1, n)
```

Ultimo caso dell'indice $[i, i]$

Se gli indici sono uno solo o due soli, allora, seguendo i casi base si ritorna 2 e 3.

Ricordandosi che:

- "i" va da 1 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 $(n-2)$

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

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

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

(b)

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

La sommatoria interna è costituita da soli 1, in quanto richiede una moltiplicazione tra tre numeri interi, nello specifico $(n-2)$, $(n-1)$, n .

Quanti uno?

Beh, da $j=i+2$ a n ci sono esattamente $n-(i+2)+1=n-i-1$ termini (sostituisco j in i , perché la serie è basata su i e il $+1$ si ha per il fatto che $i=1$).

Poi sostituire con k accorgendoti che sono esattamente gli stessi termini della sommatoria, se provi a svilupperli, e l'ultima sommatoria la puoi riscrivere in quel modo, ricordandoti che la somma di $1 \dots n$ in generale è $n(n+1)/2$.

Non avendo il termine 2 per linearità della sommatoria, non viene moltiplicato con il "fratello 2" di $(n-2)(n-1)$ e quindi il risultato è proprio $(n-2)(n-1)/2$

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 $0 \leq i \leq j \leq n - 1$, definita come segue:

Si capisce che si scansione in ordine decrescente di colonna perché quando "i" vale 0, allora "j" vale qualcosa

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

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

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.

Soluzione:

- (a) Date le dipendenze tra gli indici nella ricorrenza, un modo corretto di riempire la tabella è attraverso una scansione in cui calcoliamo gli elementi in ordine crescente di indice di riga e, per ogni riga, in ordine decrescente di indice di colonna. Il codice è il seguente.

```

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

```

(ci prendiamo la lunghezza dell'array e inizializziamo il minimo ad infinito (così, qualsiasi prima quantità minima sarà più piccola))

Siccome devo salvare il minimo, ogni volta si fa il confronto tra il minimo attuale e l'indice di C[i, j] appena salvato.

Ricordandosi che:
- "i" va da 1 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)

- (b)

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

La sommatoria interna è costituita da soli 1, in quanto richiede una moltiplicazione tra tre numeri interi, nello specifico (n-2), (n-1), n.

Quanti uno?

Beh, da $j=i+2$ a n ci sono esattamente $n-(i+2)+1=n-i-1$ termini (sostituisco j in i , perché la serie è basata su i e il $+1$ si ha per il fatto che $i=1$). Poi sostituire con k accorgendoti che sono esattamente gli stessi termini della sommatoria, se provi a svilupperli, e l'ultima sommatoria la puoi riscrivere in quel modo, ricordandoti che la somma di $1 \dots n$ in generale è $n(n+1)/2$.

Non avendo il termine 2 per linearità della sommatoria, non viene moltiplicato con il "fratto 2" di $(n-2)(n-1)$ e quindi il risultato è proprio $(n-2)(n-1)/2$

Uguale a quello precedente, ma abbiamo a che fare con MAX piuttosto che MIN

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

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

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

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.

Soluzione:

1. Date le dipendenze tra gli indici nella ricorrenza, un modo corretto di riempire la tabella è attraverso una scansione in cui calcoliamo gli elementi in ordine crescente di indice di riga e, per ogni riga, in ordine decrescente di indice di colonna. Il codice è il seguente.

```

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] * C[i,j+1]
            M <- MAX(M,C[i,j])
return M

```

(ci prendiamo la lunghezza dell'array e inizializziamo il minimo a meno infinito (così, qualsiasi prima quantità massima sarà più grande))

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)

Siccome devo salvare il massimo, ogni volta si fa il confronto tra il massimo attuale e l'indice di C[i, j] appena salvato.

- 2.

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

La sommatoria interna è costituita da soli 1, in quanto richiede una moltiplicazione tra due numeri interi, nello specifico (n-2), (n-2) (apici della serie).

Quanti uno?

Beh, da $j=n-2$ a \ ci sono esattamente $n - i - 1 = n - 1 - 1$ termini (sostituisco j in i, perché la serie è basata su i e sottraggo 1 in quanto $i=1$ ed i sarebbe il valore di j ($j=i$ è il pedice della seconda serie)).

Poi sostituire con k accorgendoti che sono esattamente gli stessi termini della sommatoria, se provi a svilupparli, e l'ultima sommatoria la puoi riscrivere in quel modo, ricordandoti che la somma di 1...n in generale è $n(n+1)/2$ quando si abbia come qui la serie con termine generale k (serie di Gauss).

Non avendo il termine 2 per linearità della sommatoria, non viene moltiplicato con il "fratto 2" di $(n-2)(n-1)$ e quindi il risultato è proprio $(n-2)(n-1)/2$

Esercizio 2 (9 punti) Data una stringa $X = x_1, x_2, \dots, x_n$, si consideri la seguente quantità $\ell(i, j)$, definita per $1 \leq i \leq j \leq n$:

$$\ell(i, j) = \begin{cases} 1 & \text{se } i = j \\ 2 & \text{se } i = j - 1 \\ 2 + \ell(i+1, j-1) & \text{se } (i < j-1) \text{ e } (x_i = x_j) \\ \sum_{k=i}^{j-1} (\ell(i, k) + \ell(k+1, j)) & \text{se } (i < j-1) \text{ e } (x_i \neq x_j). \end{cases}$$

1. Scrivere una coppia di algoritmi INIT_L(X) e REC_L(X, i, j) per il calcolo memoizzato di $\ell(1, n)$. (vedo (1,n) e so che la scansione è per diagonale)
2. Si determini la complessità *al caso migliore* $T_{\text{best}}(n)$, supponendo che le uniche operazioni di costo unitario e non nullo siano i confronti tra caratteri.

Soluzione:

1. Pseudocodice:

```

INIT_L(X)
  n <- length(X)           Prendo la lunghezza della stringa iniziale
  if n = 1 then return 1   Similmente, anche n=1 e n=2 vengono dai due casi base (array di 1/2 elementi max),
  if n = 2 then return 2   quindi o ha un elemento oppure ne ha due soli
  for i=1 to n-1 do
    L[i,i] <- 1
    L[i,i+1] <- 2
    L[n,n] <- 1 (ultimo elemento diagonale inizializzato)
  for i=1 to n-2 do      Una volta inizializzato secondo i casi base, tutto il resto della matrice viene
    for j=i+2 to n do    riempita di zeri. Poi, si considera, essendo una scansione per diagonale,
      L[i,j] <- 0          noi partiamo dal basso e riempiamo solo la parte sopra delle diagonali principali,
    return REC_L(X,1,n)    ovviamente riempita anche questa di zeri.

REC_L(X,i,j)
  if L[i,j] = 0 then
    if x_i = x_j then L[i,j] <- 2 + REC_L(X,i+1,j-1)
    else for k=i to j-1 do
      L[i,j] <- L[i,j] + REC_L(X,i,k) + REC_L(X,k+1,j)
  return L[i,j]

```

2. Il caso migliore è chiaramente quello in cui tutti i caratteri sono uguali, poiché l'albero della ricorsione in quel caso è unario e i suoi nodi interni corrispondono alle chiamate con indici $(1, n), (2, n - 1), \dots, (k, n - k + 1)$, ognuno associato a un costo unitario. Ci sono al più $\lfloor n/2 \rfloor$ di queste chiamate, e quindi $T_{\text{best}}(n) = O(n)$.
2. Il caso migliore è chiaramente quello in cui tutti i caratteri sono uguali, poiché l'albero della ricorsione in quel caso è unario e i suoi nodi interni corrispondono alle chiamate con indici $(1, n), (2, n - 1), \dots, (k, n - k + 1)$, ognuno associato a un costo unitario. Ci sono al più $\lfloor n/2 \rfloor$ di queste chiamate, e quindi $T_{\text{best}}(n) = O(n)$.

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

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

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:
 $\begin{array}{ccc} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{array}$

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.

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

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

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

La sommatoria interna è costituita da soli 2, in quanto richiede due moltiplicazioni tra due numeri interi, nello specifico $(n-1)$ ed n (apici della serie). Beh, da $i=2$ ad n . Quindi, dovendo sostituire i in j (essendo la serie basata sull'indice più esterno), si ha che $n-1$ semplicemente (in quanto, l'indice j è pari ad 1 e il 2 verrebbe portato poi fuori per linearità).

Ora, siccome abbiamo una serie che ha come apice $(n-1)$, non possiamo esprimere la soluzione in questo modo, dato che $(n-1)$ compare sia a pedice che dentro la serie. Dobbiamo quindi esprimere la serie in termini di j . Ciò comporta che io vada a “portare dentro” $\left\lfloor \frac{n-1}{2} \right\rfloor$ nella serie, in quanto moltiplicheremmo implicitamente $2(n-1)$ per altre $(n-1)$ volte e quindi è come se andassi a fare $2(n-1)(n-2) = 2(n-1)^2$

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 $1 \leq i \leq j \leq n$:

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

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_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      (ultimo elemento diagonale inizializzato)
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]

```

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

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 $(n-2)$

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.

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) Data una stringa di numeri interi $A = (a_1, a_2, \dots, a_n)$, 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 $c(n, 1)$. (vedo (1.n) e so che la scansione è per diagonale)
2. Si valuti il numero esatto $T_{CC}(n)$ di moltiplicazioni tra interi eseguite dall'algoritmo sviluppato al punto (1).

La sommatoria interna è costituita da soli 2, in quanto richiede due moltiplicazioni tra tre numeri interi, nello specifico $(n-2), (n-1), n$. Quanti due? Beh, da $j=i+2$ a n ci sono esattamente $n-(i+2)+1=n-i-1$ termini (sostituisco j in i , perché la serie è basata su i).

Quindi sarebbe sommatoria di $2(n-i-1)$, poi si può portare fuori il 2 per linearità.
Poi sostituire con k accorgendoti che sono esattamente gli stessi termini della sommatoria, se provi a svilupperli, e l'ultima sommatoria la puoi riscrivere in quel modo, ricordandoti che la somma di $1 \dots n$ in generale è $n(n+1)/2$ quando si abbia come qui la serie con termine generale k (serie di Gauss). Infatti, la sommatoria da 1 a $n-2$ è $(n-2)(n-1)/2$, semplificando diventa quello che vedi.
Il 2 come argomento della serie deriva dal fatto che ogni termine $M[i,j]$ richiede due moltiplicazioni tra 3 numeri interi.

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)          Dato che non viene passata la lunghezza dell'array, conviene salvarla nella variabile "n"
for i=1 to n do
    c[1,i] = a_i
    c[i,n] = a_{n+1-i}          Ricordandosi che:
for j=n-1 downto 1 do      - "j" va da n ad 1, quindi diventa perché abbiamo già inizializzato,
    for i=2 to n do           (n-1) la scansione
        c[i,j] = c[i-1,j] * c[i,j+1] * c[i-1,j+1]      - "j" va da 1 a n compresi, quindi dato che abbiamo inizializzato,
                                                                parte da 2
    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 $(n-1) \times (n-1)$.

La sommatoria interna è costituita da soli 2, in quanto richiede due moltiplicazioni tra due numeri interi, nello specifico $(n-1)$ ed n (apici della serie). Beh, da $i=2$ ad n . Quindi, dovendo sostituire i in j (essendo la serie basata sull'indice più esterno), si ha che $n-1$ semplicemente (in quanto, l'indice j è pari ad 1 e il 2 verrebbe portato poi fuori per linearità).

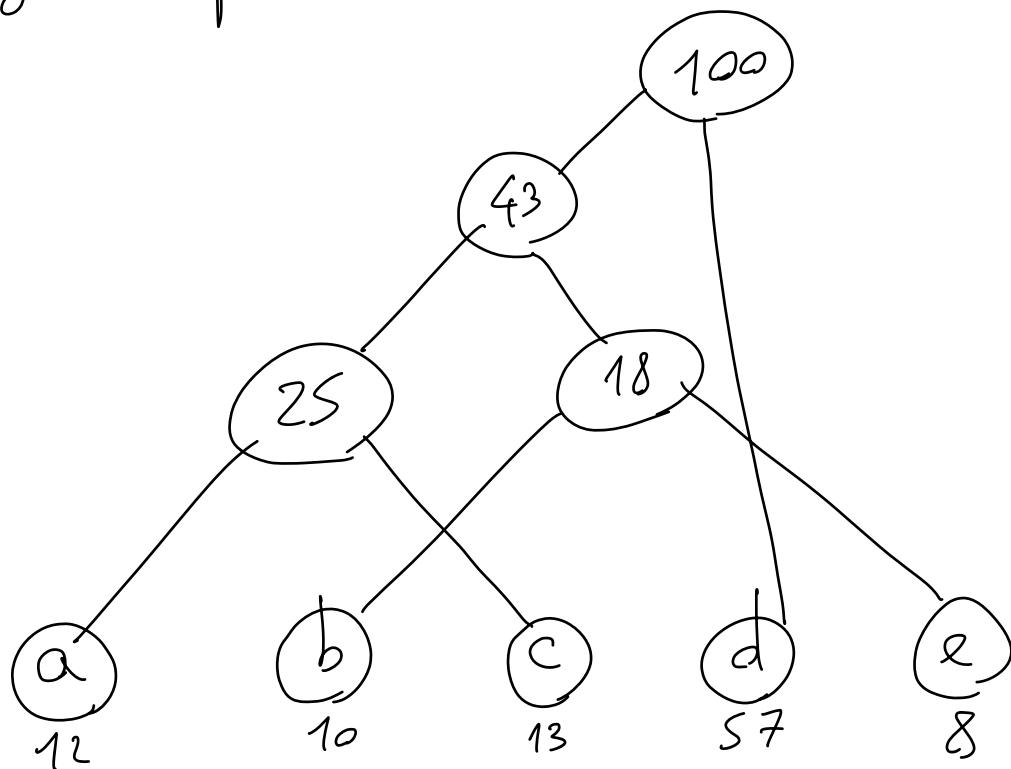
Ora, siccome abbiamo una serie che ha come apice $(n-1)$, non possiamo esprimere la soluzione in questo modo, dato che $(n-1)$ compare sia a pedice che dentro la serie. Dobbiamo quindi esprimere la serie in termini di j . Ciò comporta che io vada a “portare dentro” $\left(\begin{array}{l} n-1 \\ \end{array}\right)$ nella serie, in quanto moltiplicheremmo implicitamente $2(n-1)$ per altre $(n-1)$ volte e quindi è come se andassi a fare $2(n-1)(n-2) = 2(n-1)^2$

Esercitazioni

Domanda (5 punti): Indicare, in forma di albero binario, il codice prefisso ottenuto tramite l'algoritmo di Huffman per l'alfabeto $\{a, b, c, d, e\}$ supponendo che ogni carattere appaia con le seguenti frequenze

a	b	c	d	e
12	10	13	57	8

Spiegare il processo di costruzione del codice



Esercizio (3 punti): Si consideri un file definito sull'alfabeto $\{a, b, c\}$, con frequenze $f(a), f(b), f(c)$. Per ognuna delle seguenti codifiche determinare, se esiste, un opportuno assegnamento di valori alle 3 frequenze per cui l'algoritmo di Huffman restituisce tale codifica, oppure argomentare che tale codifica non è mai ottenibile

- 1) $\ell(a) = 0, \ell(b) = 10, \ell(c) = 11$
- 2) $\ell(a) = 1, \ell(b) = 0, \ell(c) = 11$
- 3) $\ell(a) = 10, \ell(b) = 01, \ell(c) = 00$

2) $\ell(a)$ è prefisso di $\ell(c) \Rightarrow$
 Codifica non libera da prefissi \Rightarrow
 Mai output di Huffman

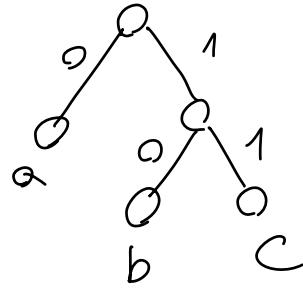
1)

per esempio

$$f(a) = 50$$

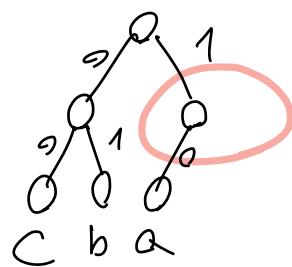
$$f(b) = 25$$

$$f(c) = 25$$



$$f(b), f(c) < f(a)$$

3)



non è pieno

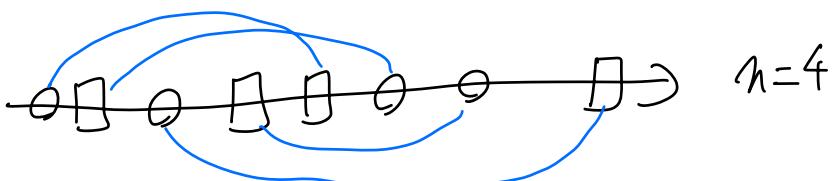
\Rightarrow non è ottima

\Rightarrow mai output
di Huffman

Esercizio: matching sulla linea

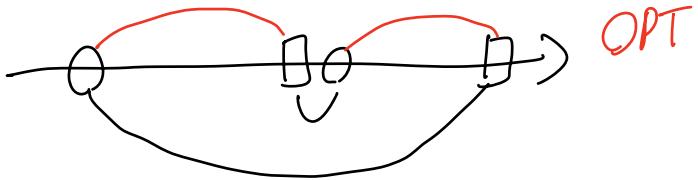
Sia $S = \{s_1, s_2, \dots, s_n\}$ un insieme di punti ordinati sulla retta reale, rappresentanti dei server. Sia $C = \{c_1, c_2, \dots, c_m\}$ un insieme di punti ordinati sulla retta reale, rappresentanti dei client. Il costo di assegnare un client c_i ad un server s_j è $|c_i - s_j|$. Fornire un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

esempio:



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

NON funziona:



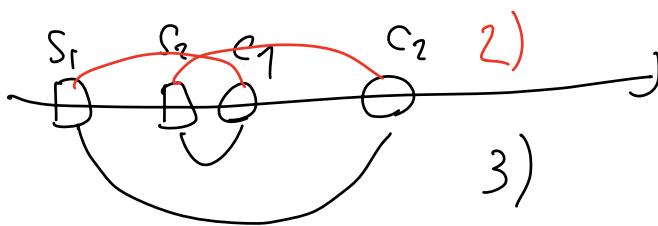
✓ 2)

$c_1 - s_1, c_2 - s_2, \dots$

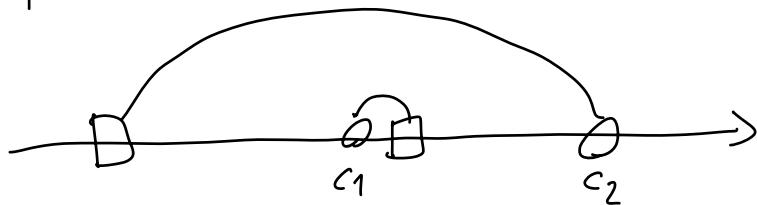
(equiv. $c_n - s_n, c_{n-1} - s_{n-1}, \dots$)

~~3)~~

c_1 al se stesso + vicino; c_2 al se stesso + vicino



NON funziona:



Per cosa: dimostrare ottimalità dell'algoritmo 2)

Esercizio: Sia $X = \{x_1, x_2, \dots, x_n\}$ un insieme di punti ordinati sulla retta reale.

Formare un algoritmo greedy che determini un insieme I di cardinalità minima di intervalli chiusi di ampiezza unitaria ($[a, b] \in I \Rightarrow b-a=1$) tale che $\forall x_i \in X \exists j \in I$ tale che $x_i \in j$.

Esempio:



- \checkmark 1) da sx a dx, inizia un nuovo intervallo nel primo punto non coperto,

$$\text{MIN_COVER}(X)$$

$$n = \text{length}(X)$$

$$C = \{[x_1, x_1 + 1]\}$$

$$\text{last} = 1$$

for $i=2$ to n do

if $x_i > x_{last} + 1$ then

$$C = C \cup \{[x_i, x_i + 1]\}$$

$last = i$

return C

Pu cosa: dimostrare ottimalità di MIN-COVER

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.)

Soluzione del problema Longest Common Substring (LCS)

ChatGPT

(a) Complessità dell'algoritmo esaustivo

L'algoritmo esaustivo analizza tutte le possibili sottostringhe comuni tra (X) e (Y).

1. Generazione di tutte le sottostringhe di (X):

- Ogni sottostringa è definita da una coppia $((i, j))$, dove $(0 \leq i \leq j < n)$.
- 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) (tempo $(O(n))$).
- Ripetiamo questa verifica per tutte le $(O(n^2))$ sottostringhe di (X).

Complessità totale:

$$[O(n^2) \cdot O(n) = O(n^3)]$$

(b) Algoritmo con verifica ottimizzata ($O(m + n)$)

Se disponiamo di un algoritmo che verifica se una stringa di lunghezza (m) è sottostringa di un'altra stringa di lunghezza (n) in $(O(m + n))$, possiamo ottimizzare il procedimento:

- Generiamo tutte le sottostringhe di (X), il cui numero è $(O(n^2))$.

- Per ciascuna sottostringa di (X), utilizziamo l'algoritmo ($O(m + n)$) per verificare la presenza in (Y).

Complessità complessiva:

$$O(n^2) \cdot O(n) = O(n^3)$$

(c) Algoritmo di programmazione dinamica

Descrizione dell'approccio

Usiamo una matrice (L), dove ($L[i][j]$) rappresenta la lunghezza della **Longest Common Substring** tra i prefissi

$(X_i = \langle x_1, \dots, x_i \rangle)$ e $(Y_j = \langle y_1, \dots, y_j \rangle)$, che termina con $(x_i = y_j)$.

Relazione di ricorrenza

$$[L[i][j] = \begin{cases} L[i-1][j-1] + 1 & \text{se } x_i = y_j, \\ 0 & \text{altrimenti.} \end{cases}]$$

Inizializzazione

$$[L[i][0] = 0, \quad L[0][j] = 0 \quad \forall i, j.]$$

Algoritmo bottom-up

- Iteriamo su tutti gli indici ($i, j \in [1, n]$).
 - Calcoliamo ($L[i][j]$) secondo la relazione di ricorrenza.
 - Manteniamo una variabile (`maxLength`) per salvare la lunghezza massima trovata.
-

Pseudocodice

```
def longest_common_substring(X, Y):
    n = len(X)
    L = [[0] * (n + 1) for _ in range(n + 1)] # Matrice di lunghezza (n+1)x(n+1)
    maxLength = 0 # Variabile per salvare la lunghezza massima trovata

    for i in range(1, n + 1):
        for j in range(1, n + 1):
            if X[i - 1] == Y[j - 1]: # Se i caratteri corrispondono
                L[i][j] = L[i - 1][j - 1] + 1
                maxLength = max(maxLength, L[i][j]) # Aggiorna la lunghezza massima
```

```

        else:
            L[i][j] = 0 # Altrimenti resettiamo a 0

    return maxLength

```

Claude

Analizziamo il problema della Longest Common Substring (LCS) punto per punto:

(a) Complessità dell'algoritmo esaustivo:

Per ogni stringa di lunghezza n, dobbiamo:

1. Generare tutte le sottostringhe possibili: $O(n^2)$
2. Per ogni sottostringa della prima stringa, cercarla nella seconda: $O(n)$
Complessità totale: $O(n^3)$

(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: $O(n^2)$

(c) Soluzione con programmazione dinamica:

Definiamo $LCS[i,j]$ come la lunghezza della LCS che termina in $X[i]$ e $Y[j]$.

Relazione di ricorrenza:

```

LCS[i, j] = {
    LCS[i-1, j-1] + 1      se X[i] = Y[j]
    0                      altrimenti
}

```

Algoritmo bottom-up:

```

def lcs_length(X, Y):
    n = len(X)
    dp = [[0] * (n+1) for _ in range(n+1)]
    max_length = 0

    for i in range(1, n+1):
        for j in range(1, n+1):
            if X[i-1] == Y[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
                max_length = max(max_length, dp[i][j])

```

```
return max_length
```

Complessità: $O(n^2)$ tempo e spazio, ma con un fattore costante minore rispetto all'approccio precedente.

Esercizio 2 (9 punti) Supponiamo di avere un numero illimitato di monete di ciascuno dei seguenti valori: 50, 20, 1. Dato un numero intero positivo n , l'obiettivo è selezionare il più piccolo numero di monete tale che il loro valore totale sia n . Consideriamo l'algoritmo greedy che consiste nel selezionare ripetutamente la moneta di valore più grande possibile.

- Fornire un valore di n per cui l'algoritmo greedy *non* restituisce una soluzione ottima.
- Supponiamo ora che i valori delle monete siano 10, 5, 1. In questo caso l'algoritmo greedy restituisce sempre una soluzione ottima: dimostrare che ogni insieme ottimo M^* di monete di valore totale n contiene la scelta greedy.

Soluzione:

- Per esempio $n = 60$, perché la soluzione ottima è 3 monete da 20, mentre l'algoritmo greedy restituisce 11 monete (una da 50 e 10 da 1).
- Sia M^* una soluzione ottima. Sia x il valore maggiore tra 10, 5, e 1 che sia non superiore a n . Se M^* contiene una moneta di valore x , la proprietà è dimostrata. Altrimenti, sia $M \subseteq M^*$ un insieme di (2 o più) monete di valore totale x (si osservi che tale insieme esiste sempre quando i valori delle monete sono 10, 5, 1); consideriamo $M' = M^* \setminus M \cup X$, dove X è l'insieme contenente una moneta di valore x . M' è un insieme di monete di valore totale n e di cardinalità inferiore a quella di M^* : assurdo, quindi questo secondo caso non può verificarsi, e quindi M^* contiene necessariamente una moneta di valore x .

Domanda B (7 punti) Si consideri il problema di selezione di attività compatibili:

- Definire il problema.
- Descrivere brevemente l'algoritmo ottimo GREEDY_SEL visto in classe.
- Fornire un esempio di algoritmo greedy *non* ottimo, motivandone la non ottimalità.

Analizziamo punto per punto:

1. Definizione del problema:

Date n attività caratterizzate da tempo di inizio s_i e tempo di fine f_i , selezionare il massimo numero di attività compatibili tra loro. Due attività i e j sono compatibili se non si sovrappongono temporalmente, ovvero se $f_i \leq s_j$ o $f_j \leq s_i$.

2. GREEDY_SEL ottimo:

L'algoritmo seleziona iterativamente l'attività con tempo di fine minore tra quelle compatibili con le attività già selezionate. La correttezza deriva dal fatto che scegliere l'attività che finisce prima massimizza il tempo disponibile per le attività successive.

Pseudocodice:

```
GREEDY_SEL(s, f):
```

```
    A = {a1} // Attività con minor tempo di fine
```

```

last = a1
for i = 2 to n do
    if s[i] >= f[last] then
        A = A ∪ {ai}
        last = ai
return A

```

3. Esempio non ottimo:

Un algoritmo greedy non ottimo potrebbe selezionare l'attività più breve disponibile:

```

NON_OTTIMO(s, f):
    A = {}
    while esistono attività disponibili do
        scegli l'attività i con durata minima (fi - si)
        A = A ∪ {ai}
        rimuovi le attività incompatibili con i
    return A

```

Non è ottimo perché un'attività breve potrebbe impedire la selezione di più attività lunghe compatibili tra loro. Ad esempio:

- a1(0,2), a2(1,3), a3(2,4), a4(0,10)

L'algoritmo sceglierebbe a1, a2, a3 mentre la soluzione ottima è a4.