```
      else
         L[j] = A[j]

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

  return max
```

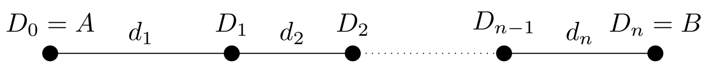
Se vogliamo anche il sottoarray, occorre ricordare il massimo e dove viene raggiunto, lo facciamo mediante un array $S[1..n]$ tale che $S[j]$ contenga l'indice dal quale inizia il sottoarray massima che termina con $A[j]$.

```
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 $D_1, \ldots, D_{n-1}$, a distanze di $d_1, \ldots, d_n$ 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.

i. 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 $D_0 (= A), D_1, \ldots, D_n (= B)$. 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 lungheza maggiore di d ovvero:

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

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

Sottostruttura ottima. Sia $S = D_{i_0} \ldots D_{i_k}$ una soluzione ottima per il problema $D_0, D_1, \ldots, D_n$. Se consideriamo il sottoproblema di andare da D_{i_1} a D_n, ovvero $D_{i_1}, D_{i_1+1} \ldots D_n$, allora $S_1 = D_{i_1}, \ldots, D_{i_k}$ 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 $D_0, D_1, \ldots, D_n$, 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 $S = D_{j_0} \ldots D_{j_k}$, certamente $j_0 = 0 = i_0$ e $j_1 \leq i_1$. Quindi è immediato verificare che anche $S' = D_{j_0} D_{i_1} D_{j_2} \ldots D_{j_k}$ è una soluzione per il problema $D_0, D_1, \ldots, D_n$, 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 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
```

1 2 3 4 5 6 7

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 $\vec{r} = r_1, \ldots, r_n$ di richieste, dove r_i è l'orario della i-ma prenotazione, si vuole determinare una sequenza di turni $\vec{t} = t_1, \ldots, t_k$, 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 greedy `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 $\vec{t} = t_1, \ldots, t_k$ tale che per ogni $i = 1, \ldots, n$ l'istante $r_i \in I(t_j)$ per qualche $j = 1, \ldots, k$, 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 $\vec{t} = t_1 q \ldots, t_k$ è ottima per le richieste $\vec{r} = r_1, \ldots, r_n$ allora $t_2, \ldots, t_k$ 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 $\vec{r} = r_1, \ldots, r_n$ 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 $\vec{t'} = t'_1 \ldots t'_k$ 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 $\Theta(n)$.

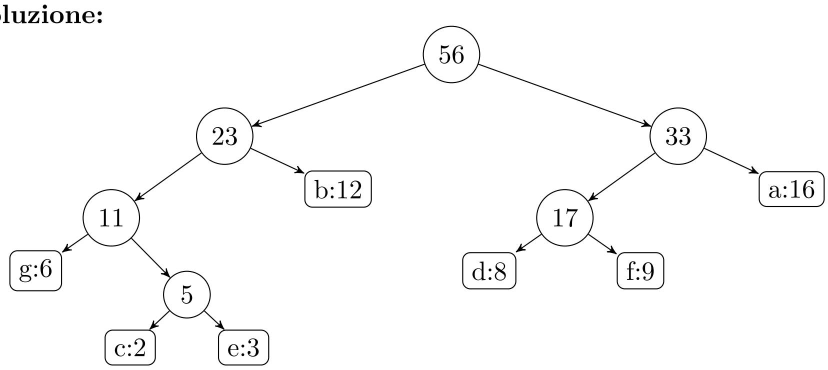
Domanda 39 Indicare il codice prefisso ottenuto utilizzando l'algoritmo di Huffmann per l'alfabeto $\{a, b, c, d, e, f, g\}$, supponendo che ogni simbolo appaia con le seguenti frequenze.

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:

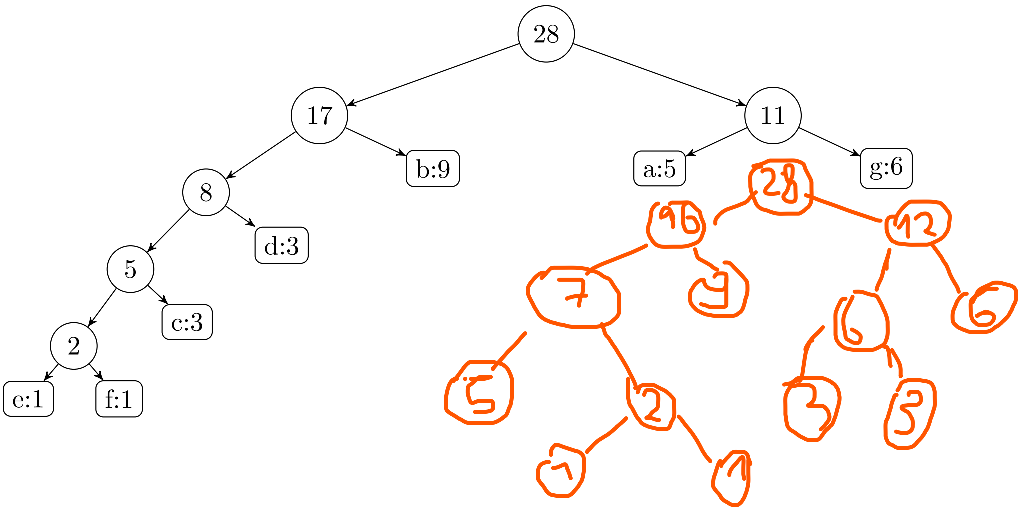


Domanda 40 Indicare il codice prefisso ottenuto utilizzando l'algoritmo di Huffmann per l'alfabeto $\{a, b, c, d, e, f, g\}$, supponendo che ogni simbolo appaia con le seguenti frequenze.

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

Spiegare il processo di costruzione del codice.

Soluzione:

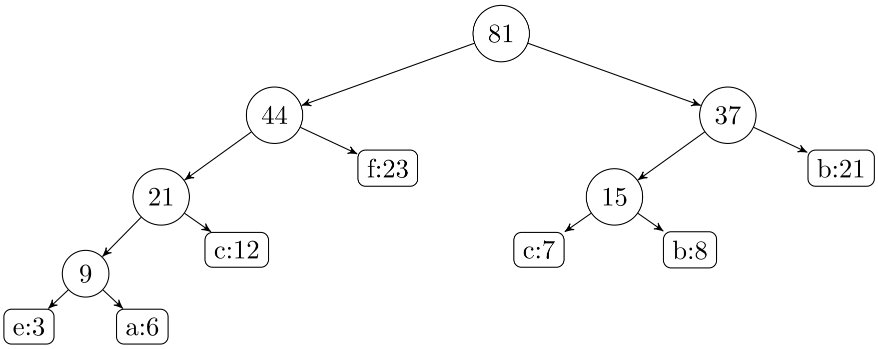


Domanda 41 Indicare il codice prefisso ottenuto utilizzando l'algoritmo di Huffmann per l'alfabeto $\{a, b, c, d, e, f, g\}$, supponendo che ogni simbolo appaia con le seguenti frequenze.

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

Spiegare il processo di costruzione del codice.

Soluzione:

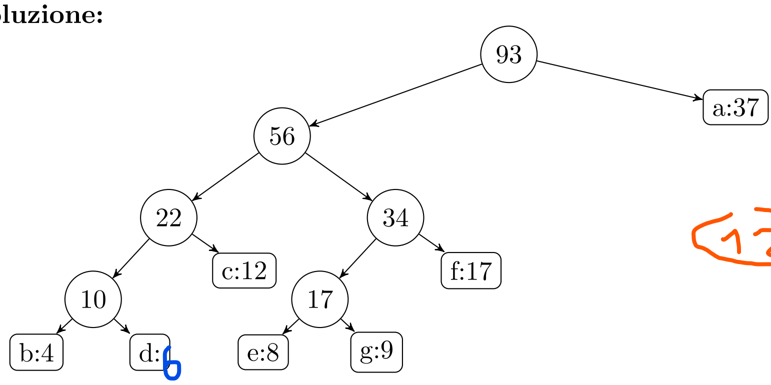


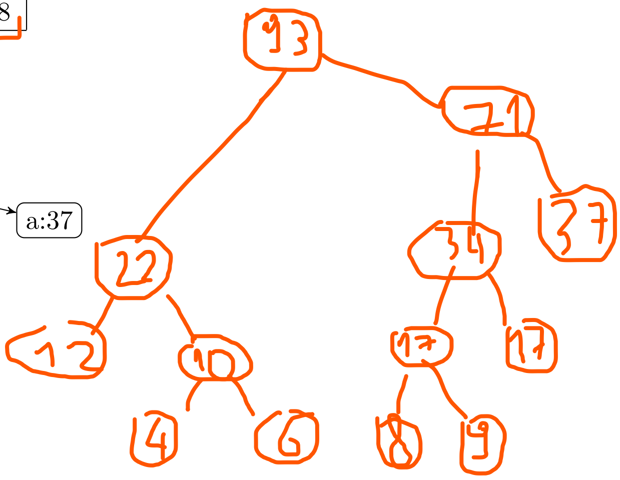
Domanda 42 Indicare il codice prefisso ottenuto utilizzando l'algoritmo di Huffmann per l'alfabeto $\{a, b, c, d, e, f, g\}$, supponendo che ogni simbolo appaia con le seguenti frequenze.

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

Spiegare il processo di costruzione del codice.

Soluzione:



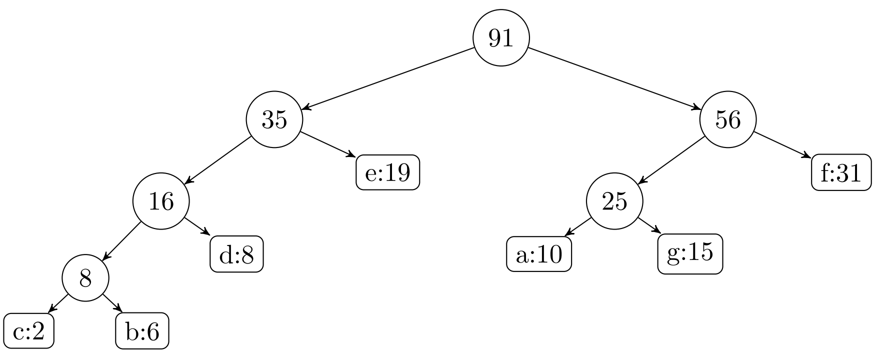


Domanda 43 Indicare il codice prefisso ottenuto utilizzando l'algoritmo di Huffmann per l'alfabeto $\{a, b, c, d, e, f, g\}$, supponendo che ogni simbolo appaia con le seguenti frequenze.

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

Spiegare il processo di costruzione del codice.

Soluzione:

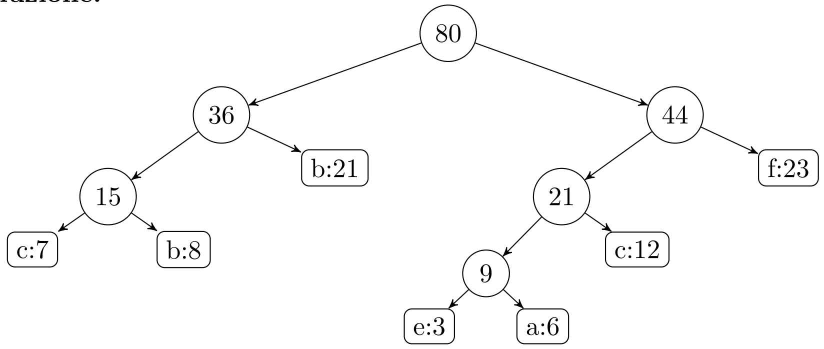


Domanda 44 Indicare il codice prefisso ottenuto utilizzando l'algoritmo di Huffmann per l'alfabeto $\{a, b, c, d, e, f, g\}$, supponendo che ogni simbolo appaia con le seguenti frequenze.

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

Spiegare il processo di costruzione del codice.

Soluzione:

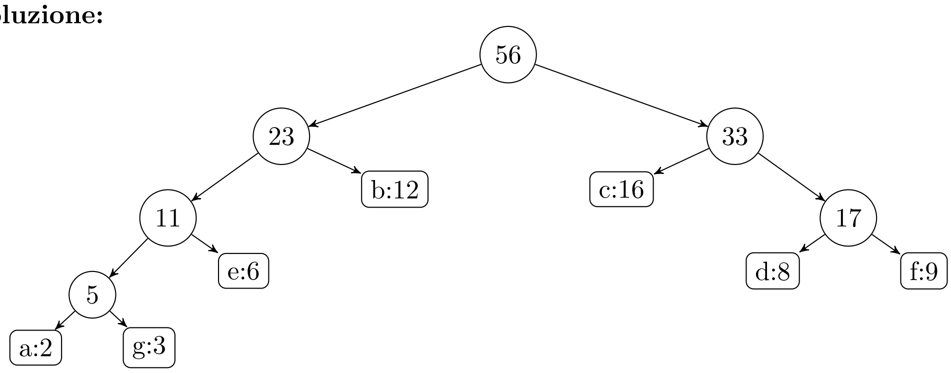


Domanda 45 Indicare il codice prefisso ottenuto utilizzando l'algoritmo di Huffmann per l'alfabeto $\{a, b, c, d, e, f, g\}$, supponendo che ogni simbolo appaia con le seguenti frequenze.

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 **s** e **f** di tempi di inizio e fine:

$$\mathbf{s} = (1, 4, 2, 3, 7, 8, 11) \qquad \mathbf{f} = (3, 6, 9, 10, 11, 12, 13).$$

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 $p_1, p_2, \ldots, p_n$ 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

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

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 coppie, ha la minima differenza.

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 $n = 2$.

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.

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:

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).

   (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.

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

Si veda la risoluzione di pag. 52
per capire come ragionarlo

$$\mathbf{s} = (1, 1, 2, 4, 2, 5, 6, 9) \qquad \mathbf{f} = (3, 4, 5, 6, 7, 9, 10, 12).$$

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 $S = \{a_i, a_2, \ldots, a_n\}$, 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:

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

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:

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

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

Esempio:
1 2 3 4 5 6 7
P = (7, 6, 5, 4, 3, 2, 1)

Inizializziamo l'insieme, la partizione e gli estremi.

Si considera un ciclo per cui "first" è <= "last" (perché scansioniamo gli estremi, come detto)

Se l'estremo inf. è < dell'estremo sup.
non abbiamo ancora salvato nulla in P (non abbiamo estr. inf)
allora salvo in P sia l'estremo inf che l'estremo sup
e incremento first. Salvandolo una volta sola, so che la somma è sempre > 1.

Altrimenti,
salvo solo l'estremo superiore migliore, la cui somma
è sempre >= 1

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

2. La scelta greedy è $\{a_1, a_n\}$ se $n > 1$ e $a_1 + a_n \le 1$, altrimenti $\{a_n\}$. Ora dimostriamo che esiste sempre una soluzione ottima che contiene la scelta greedy. I casi $n = 1$ e $a_1 + a_n > 1$ sono banali, visto che in questi casi ogni soluzione ammissibile deve contenere il sottoinsieme $\{a_n\}$. Quindi assumiamo che la scelta greedy sia $\{a_1, a_n\}$. Consideriamo una qualsiasi soluzione ottima dove a_1 e a_n non sono accoppiati nello stesso sottoinsieme. Quindi, esistono due sottoinsiemi S_1 e S_2, con $a_1 \in S_1$ e $a_n \in S_2$. Sostituiamo questi due sottoinsiemi con $S'_1 = \{a_1, a_n\}$ (cioè, la scelta greedy) e $S'_2 = S_1 \cup S_2 \setminus \{a_1, a_n\}$. $|S'_2| \le 2$ 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 scegliamo 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 $j \in \{1, 2, \ldots, n\}$, 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.

Soluzione:

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. La complessità è O(nlogn) perché evidentemente il prof usa MergeSort

(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:
- 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.

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.