Domanda 17 Dare la definizione di $\Omega(f(n))$. Mostrare che se $f(n) = \Omega(g(n))$ e $g(n) = \Omega(h(n))$ allora $f(n) = \Omega(h(n))$.

Soluzione: Si ha che

$$\Omega(f(n)) = \{g(n) \mid \exists c > 0.\ \exists n_0 \in \mathbb{N}.\ \forall n \geq n_0.\ 0 \leq cg(n) \leq f(n)\}.$$

Se $f(n) = \Omega(g(n)$ e $g(n) = \Omega(h(n))$ allora esistono $c_1, c_2 > 0$, $n_1, n_2 \in \mathbb{N}$ tali che per ogni $n \geq n_1$

$$0 \leq c_1 g(n) \leq f(n) \tag{1}$$

e per ogni $n \geq n_2$

$$0 \leq c_2 h(n) \leq g(n) \tag{2}$$

Ne consegue che per ogni $n \geq \max\{n_1, n_2\}$, moltiplicando (2) per c_1 si ha

$$0 \leq c_1 c_2 h(n) \leq c_1 g(n) \leq f(n)$$

ovvero, indicato con $n_0 = \max\{n_1, n_2\}$ e $c = c_1 c_2$, per ogni $n \geq n_0$,

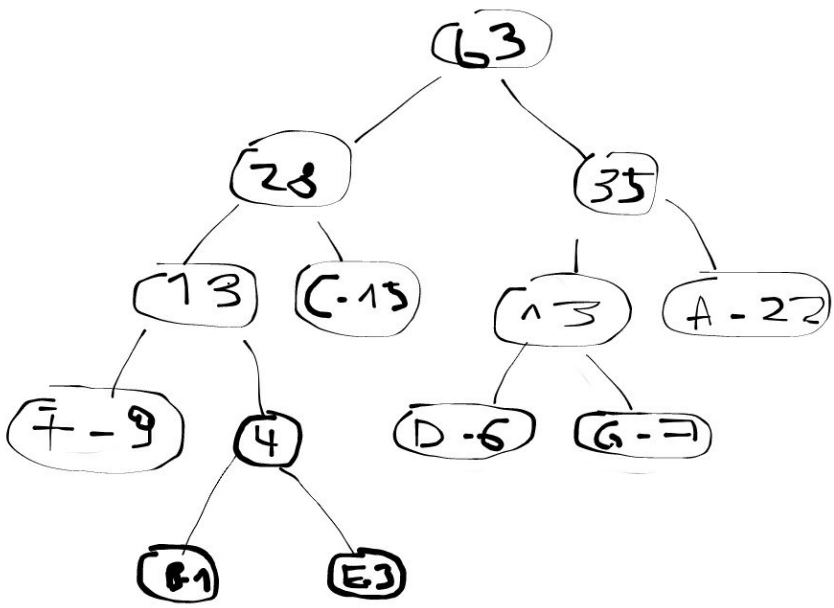
$$0 \leq ch(n) \leq f(n)$$

ovvero $f(n) = \Omega(h(n))$.

<u>Domanda B</u> (4 punti): Indicare il codice prefisso ottenuto utilizzando l'algoritmo di Huffman 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
22	1	15	6	3	9	7

Spiegare il processo di costruzione del codice.



Esercizio 5 Un array $A[1..n]$ di numeri si dice *alternante* se non ha elementi contigui identici (ovvero per ogni $i \leq n-1$ vale $A[i] \neq A[i+1]$) e inoltre per ogni $i \leq n-2$, vale che $a_i < a_{i+1} > a_{i+2}$ oppure $a_i > a_{i+1} < a_{i+2}$. Ad esempio gli array [1, 2, -1, 3, 2] e [5, 1, 2, -1, 3, 2] sono alternanti, mentre non lo sono [1, 2, 3] e [1, 1, 2]. Scrivere una funzione ricorsiva `alt(A,n)` che dato un array $A[1..n]$ di numeri verifica se è alternante. Valutarne la complessità.

Soluzione:

```
# alt: dato un array A[1..n] verifica se  e' alternante
alt(A,n)
    return altRec(A,n,0) or altRec(A,n,1)

altRec(A,i,dir)
    # verifica se A[1..i] e' alternante.
    # usa un ulteriore parametro per indicare se la sequenza alternante deve


      # concludersi crescendo (0) o decrescendo (1).

      if i==1
          return True
      else
          if dir==0
              return  altRec(A,i-1,1) and (A[i-1] < A[i])
          else
              return  altRec(A,i-1,0) and (A[i-1] > A[i])
```

Dato che la parte non ricorsiva è di costo costante, la complessità si può esprimere come $T(n) = 1 + T(n-1)$ che porta a $T(n) = \Theta(n)$.

Esercizio 32 Realizzare, con tecniche di programmazione dinamica, un algoritmo che dato un array $A[1..n]$, non vuoto, trova un sottoarray non vuoto di somma massima, ovvero due indici i, j con $1 \leq i \leq j \leq n$ tali che $A[i] + A[i+1] + \ldots + A[j]$ sia massima. Ad esempio per [-10, 4, 1, -1, 2, -1] il sottoarray di somma massima è [4, 1, -1, 2]. Più precisamente:

i. indicato con l_j la somma massima di una sottoarray di $A[1..n]$ che termini con $A[j]$ (quindi del tipo $A[i..j]$), darne una caratterizzazione ricorsiva;

ii. tradurre tale definizione in un algoritmo (bottom up o top down con memoization) che determina la somma massima;

iii. trasformare l'algoritmo in modo che fornisca anche la sottosstringa, non solo la sua somma;

iv. valutare la complessità dell'algoritmo.

Nota: La soluzione proposta deve articolarsi nei passi sopra descritti.

Soluzione: L'osservazione fondamentale è che un sottoarray con somma massima di $A[1..n]$ che termini in $A[j]$ sarà, un sottoarray di somma massima che termina in $A[j-1]$, concatenato con $A[j]$, se il primo ha somma positiva, altrimenti semplicemente $A[j..j]$. In simboli, per $j = 1, \ldots, n$

$$l_j = \begin{cases} 1 & \text{se } j = 1 \text{ o } l_{j-1} \leq 0 \\ l_{j-1} + 1 & \text{altrimenti, ovvero se } j > 1 \text{ e } l_{j-1} > 0 \end{cases}$$

Il sottoarray con somma massima terminerà in qualche j e quindi sarà poi sufficiente massimizzare i valori trovati.

Ne segue l'algoritmo che riceve in input l'array $A[1,n]$ e usa una matrice $L[1..n]$ dove $L[j]$ rappresenta la somma di una sottoarray di lunghezza massima che termina in $A[j]$.

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

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

```
    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 $\Theta(n)$.