Domanda 1 Risolvere la ricorrenza $T(n) = 4\,T(n/2) + n\log n$ utilizzando il master theorem.

Soluzione: Rispetto allo schema generale si ha $a = 4$, $b = 2$, $f(n) = n\log n$. Si osserva che $\log_b a = 2$ quindi $f(n) = O(n^{\log_b a - \epsilon})$ (per $0 < \epsilon < 1$) e quindi $T(n) = \Theta(n^2)$.

Domanda 2 Mostrare che la ricorrenza $T(n) = T(n/2) + T(n/4) + n$ ammette soluzione $T(n) = \Theta(n)$ utilizzando il metodo di sostituzione.

Soluzione: Si prova separatamente che asintoticamente

1. $T(n) \leq cn$, per un'opportuna costante c
2. $T(n) \geq dn$, per un'opportuna costante d

Ad esempio, per la prima,

$$\begin{aligned} T(n) &= T(n/2) + T(n/4) + n \\ &\leq n/2\,c + n/4\,c + n \\ &= (1 + 3/4\,c)n \\ &\leq cn \end{aligned}$$

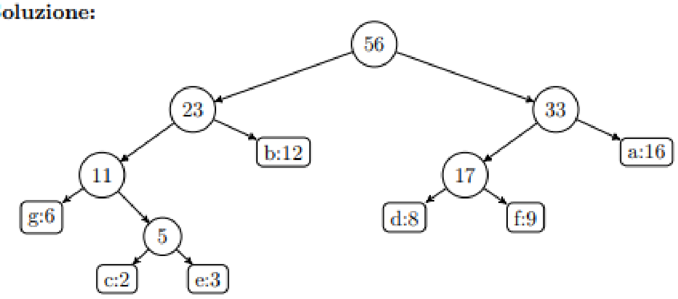
quando $1 + 3/4\,c \leq c$, ovvero $c \geq 4$.

Domanda 2 (4 punti) 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

Spiegare il processo di costruzione del codice.

Soluzione:



Domanda 23 Scrivere una funzione `IsMaxHeap(A)` che dato in input un array di interi `A[1..n]` che verifica se `A` è organizzato a max-heap e ritorna un corrispondente valore booleano. Valutarne la complessità.

Soluzione: Versione ricorsiva

```
IsMaxHeap(A,i,j) // verifica se l'array A[i,j] e' un max-heap
                    (o meglio, se contiene un max-heap radicato in i,
                     dato che al passo ricorsivo non tutto [i,j]
                     conterra' il sottoalbero scelto
  l = 2*i
  r = 2*i+1
  if l<j
    leftOk = A[i] >= A[l] and IsMaxHeap(l, j)
  else
    leftOk = true

  if r<j
    rightOk = A[i] >= A[r] and IsMaxHeap(r, j)
  else
    rightOk = true

  return leftOk and rightOk
```

Esercizio 6 Fornire lo pseudocodice di una procedura `min(A,B)` che dati due array `A` e `B` che siano uno la permutazione dell'altro ne trova l'elemento minimo confrontando esclusivamente elementi di `A` ed elementi di `B` (non si possono confrontare due elementi di `A` o due elementi di `B` tra loro, e non si possono fare copie degli array, ovvero la funzione deve operare con spazio costante). Valutare la complessità della funzione.

Soluzione: L'idea è quella di confrontare gli elementi di A e B, scorrendoli con due indici i, j avanzando in B quando $A[i] \leq B[j]$ e in B altrimenti. Sul lato di A l'avanzamento si fermerà ad un elemento $A[i]$ che sarà minore o uguale di tutti gli elementi di B. Il vero e proprio invariante è un po' complicato.

```
min(A,B)
  i=j=1
  // invariante:   (forall h \in [0,i) exists k in [0,j] A[h] > B[k]) or
                   (i <= n) and (forall k \in [0,j) exists h in [0,i] A[h] <= B[k])
  while (j <= n)
     if A[i] <= B[j]
        j = j+1
     else
        i = i+1
  return A[i]
```

Si noti che l'invariante implica implica $i \leq n$, quindi questo controllo nel ciclo non è necessario.

Il numero di iterazioni è al massimo $2n$ dato che uno dei due indici (i oppure j) viene sempre incrementato. Dunque la complessità è $\Theta(n)$.

Esercizio 2 (11 punti) Dare un algoritmo per individuare, all'interno di una stringa $a_1 \dots a_n$ una sottostringa (di caratteri consecutivi) palindroma di lunghezza massima. Ad esempio, nella stringa "*colonna*" la sottostringa palindroma di lunghezza massima è "*olo*". Più precisamente:

i. dare una caratterizzazione ricorsiva della lunghezza massima $l_{i,j}$ di una sottostringa palindroma di $a_i \dots a_j$;

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

iii. trasformare l'algoritmo in modo che permetta anche di individuare la stringa, non solo la sua lunghezza;

iv. valutare la complessità dell'algoritmo.

Soluzione: La caratterizzazione ricorsiva della lunghezza massima $l_{i,j}$ di una sottostringa palindroma di $a_{i,j} = a_i \dots a_j$ deriva dall'osservazione che

- se $i > j$, quindi $a_{i,j} = \varepsilon$ è la stringa vuota, dato che la stringa vuota è palindroma $l_{i,j} = 0$;
- se $i = j$ quindi $a_{i,j}$ è la stringa di un solo carattere, che è palindroma, vale $l_{i,j} = 1$;
- altrimenti, se $i < j$ quindi $a_{i,j}$ consta di almeno due caratteri, distinguiamo due sottocasi:
  - se $a_i = a_j$ e la sottostringa $a_{i+1,j-1}$ è palindroma, anche $a_{i,j}$ è palindroma. Quest'ultima condizione si riduce a $l_{i+1,j-1} = |a_{i+1,j-1}| = j-i-1$. In questo caso vale $l_{i,j} = j-i+1$ (lunghezza di $a_{i,j}$)
  - se invece $a_i \neq a_j$, una sottostringa palindroma non può comprendere entrambi gli estremi e quindi ci si riduce ai sottoproblemi $a_{i+1,j}$ e $a_{i,j-1}$ e si prende il massimo.

In sintesi:

$$l_{i,j} = \begin{cases} 0 & \text{if } i > j \\ 1 & \text{if } i = j \\ l_{i+1,j-1} + 2 & \text{if } a_i = a_j \text{ and } l_{i+1,j-1} = j-i-1 \\ \max\{l_{i,j-1}, l_{i+1,j}\} & \text{otherwise } (a_i = a_j \text{ and } l_{i+1,j-1} \neq j-i-1 \text{ or } a_i \neq a_j) \end{cases}$$

Ne segue l'algoritmo che riceve in input la stringa, nella forma di un array di caratteri $A[1,n]$ e usa una matrice $L[1..n, 0..n]$ dove $L[i,j]$ rappresenta la lunghezza di una sottostringa palindroma in $A[i,j]$

```
palindrome(A, n)
  for i = 1 to n
    L[i,i-1] = 0
    L[i,i] = 1

  for len = 2 to n
    for i = 1 to n-len+1
      j = i + len - 1
      if (A[i] = A[j]) and (L[i+1,j-1] = j-i-1)
         L[i,j] = L[i+1,j-1] + 2
      else
         L[i,j] = max (L[i,j-1], L[i+1,j])

  return L[1,n]
```

Se vogliamo anche la sottostringa, occorre ricordare il massimo e dove viene raggiunto, lo facciamo mediante un array $P[1..n, 1..n]$ tale che $P[i, j]$ contenga l'indice dal quale inizia la sottostringa palindroma più lunga di $a_{i,j}$

```
palindrome(A, n)
  for i = 1 to n
    L[i,i-1] = 0
    L[i,i] = 1
    P[i,i] = i

  for len = 2 to n
    for i = 1 to n-len+1
      j = i + len - 1
      if (A[i] = A[j]) and (L[i+1,j-1] = j-i-1)
         L[i,j] = L[i+1,j-1] + 2
         P[i,j] = i
      else
         if L[i,j-1] >= L[i+1,j]
            L[i,j] = L[i,j-1]
            P[i,j] = P[i][j-1]
         else
            L[i,j] = L[i+1,j]
            P[i,j] = P[i+1][j]

   start = P[1,n]
   end   =  start + L[1,n]-1

   return A[start..end]
```

La complessità è $\Theta(n^2)$.