Domanda A (4 punti) Sia data la seguente equazione di ricorrenza:

$$T(n) = T(n-1) + \log n$$

Si dimostri che $T(n) = O(n \log n)$.

Soluzione: Si procede una prova induttiva del fatto che $T(n) \leq cn \log n$.

$$\begin{aligned} T(n) &= T(n-1) + \log n \\ &\leq c(n-1)\log(n-1) + \log n \\ &\leq c(n-1)\log n + \log n \\ &= (cn - c + 1)\log n \\ &\leq cn \log n \end{aligned}$$

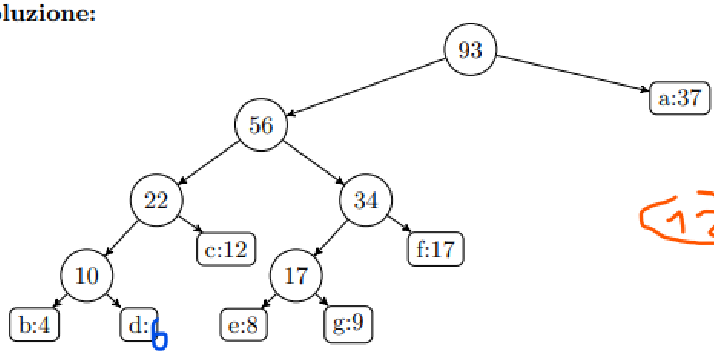
per $c \geq 1$.

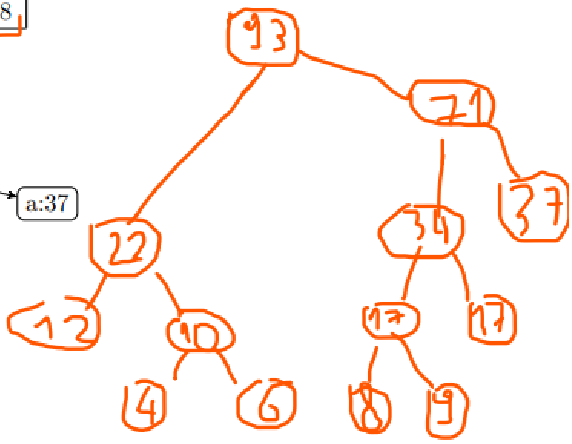
Domanda 42 Indicare il codice prefisso ottenuto utilizzando l'algoritmo di Huffmann per l'alfabeto $\{a, b, c, d, e, f, g\}$, supponendo che ogni simbolo appaia con le seguenti frequenze.

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

Spiegare il processo di costruzione del codice.

Soluzione:





Domanda C (5 punti) Realizzare una funzione `RevCountingSort(A,B,n,k)` che, dato un array `A[1..n]` contenente interi nell'intervallo `[0..k]`, restituisce in `B[1..n]` una sua permutazione ordinata in modo decrescente utilizzando una variante del counting sort. Valutare la complessità.

Soluzione:

```
RevCountingSort (A, B, n, k)
  C[1..k] = 0
  for i = 1 to n
      C[A[i]] = C[A[i]]+1

   for j = k-1 downto 0 do
      C[j] = C[j] + C[j+1]

   for j = 1 to n
      B[C[A[i]]] = A[i]
      C[A[i]] = C[A[i]] - 1
```

Esercizio 14 Realizzare una procedura `BST(A)` che dato un array `A[1..n]` di interi, ordinato in modo crescente, costruisce un albero binario di ricerca di altezza minima che contiene gli elementi di `A` e ne restituisce la radice. Per allocare un nuovo nodo dell'albero si utilizzi una funzione `mknod(k)` che dato un intero `k` ritorna un nuovo nodo con `x.key=k` e figlio destro e sinistro `x.left = x.right = nil`. Valutarne la complessità.

Soluzione:

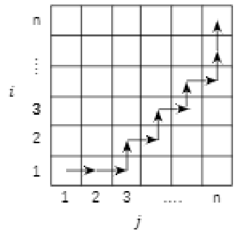
i. L'implementazione è la seguente:

```
BST(A)
   return BST-rec(A,1,n)

BST-rec(T,A,p,q)
   if p <= q
      m = floor(p+q/2)
      x=mknode(A[m])
      x.l = BST-rec(A,p,m-1)
      x.r = BST-rec(A,m+1,q)
    else
      x = nil

    return x
```

ii. Si ottiene la ricorrenza $T(n) = 2T(n/2) + \Theta(1)$ e quindi il costo è $T(n) = O(n)$.

Esercizio 22 Si supponga di avere una scacchiera $\mathtt{n} \times \mathtt{n}$. Si vuole spostare un pezzo dall'angolo in basso a sinistra (1,1) a quello in alto a destra `(n,n)`.

Il pezzo può muoversi di una casella verso l'alto (↑) o verso destra (→). Un passo dalla casella `(i,j)` ha un costo `u(i,j)` se verso l'alto e `r(i,j)` se verso destra. Realizzare un algorimo `MinPath(u,r,n)` che dati in input gli array `u[1..n,1..n]` e `r[1..n,1..n]` dei costi dei singoli passi fornisce il cammino minimo. Più in dettaglio:



i. fornire una caratterizzazione ricorsiva del costo minimo di un cammino `C(i,j)` per andare dalla casella `(i,j)` alla casella `(n,n)`

ii. tradurre tale definizione in un algoritmo `MinPath(u,r,n)` (bottom up o top down con memoization) che determina il costo di un cammino minimo da `(1,1)` a `(n,n)`

iii. trasformare l'algoritmo in modo che stampi la sequenza di passi di costo minimo;

iv. valutare la complessità dell'algoritmo.

Soluzione:

i. La caratterizzazione ricorsiva del costo è:

$$C(i,j) = \begin{cases} 0 & \text{se } x = n \text{ e } y = n \\ r(i,j) + C(i,j+1) & \text{se } i = n \text{ e } j < n \\ u(i,j) + C(i+1,j) & \text{se } i < n \text{ e } j = n \\ \min\{r(i,j) + C(i,j+1), u(i,j) + C(i+1,j)\} & \text{altrimenti} \end{cases}$$

ii. Ne segue l'algoritmo che riceve in input gli array `u(1..n,1..n)` e `r(1..n,1..n)` dei costi e calcola il costo minimo di un cammino da `(1,1)` a `(n,n)`

```
// u[1..n,1..n], r[1..n,1..n] costi per andare in alto
// e a destra, rispettivamente

MinPath(u,l,n)
  C[n,n] = 0


  for j = n-1 downto 1
      C[n,j] = C[n,j+1] + r[n,j]

  for i = n-1 downto 1
      C[i,n] = C[i+1,n] + u[i,n]

for i=n-1 downto 1
      for j = n-1 downto 1
                 C[i,j] = min { C[i+1,j] + u[i,j], C[i,j+1] + r[i,j] }

  return C[1,1]
```

iii. Se vogliamo anche una soluzione ottima

```
// u[1..n,1..n], r[1..n,1..n] costi per andare in alto
// e a destra, rispettivamente

MinPath(u,l,n)
  C[n,n] = 0

  for j = n-1 downto 1
      C[n,j] = C[n,j+1] + r[n,j]
      D[n,j] = 'right'


for i = n-1 downto 1
    C[i,n] = C[i+1,n} + u[i,n]
    D[i,n] = 'up'

for i=n-1 downto 1
    for j = n-1 downto 1
              up    = C[i+1,j] + u[i,j]
        right = C[i,j+1] + r[i,j]

        if up <= right
           C[i,j] = up
           // memorizza anche la dirazione ottima
           D[i,j] = 'up'
        else
           C[i,j] = right
           D[i,j] = 'right'

i = j =1
while (i < n) or (j < n)
   print (i,j)
   if D[i,j] = up
      i=i+1
   else
      j=j+1
```

iv. La complessità è $O(n^2)$.