1. Scrivere un metodo char[] stringInvert(CharQueue str) che, ricevuta in ingresso una coda con elementi di tipo char, restituisce un array corrispondente alla coda di ingresso invertita. Si supponga di avere a disposizione una classe CharStack che implementa una pila i cui elementi sono char (i metodi sono tutti e solo quelli di una pila, i loro nomi sono a scelta dello studente).
Nota: si supponga che la classe CharQueue implementi una pila di caratteri. Valgono le stesse considerazioni fatte per CharStack.
Esempio: se la coda contiene ‘a’, ‘b’, ‘c’ (in questo ordine) l’array corrispondente deve contenere ‘b’, ‘c’, ‘a’, in questo ordine.

2. Scrivere un metodo boolean palindroma(char[] str) che, ricevuto in ingresso un array di char, verifica se esso corrisponda a una stringa palindroma o meno. E’ possibile risolvere il problema senza fare uso di pile, ma per esercizio si usi la classe CharStack definita al punto 1.

3. Dimostrare che:
- 1/2 n^2 = \Omega(n^2)
- 3n = o(n^3) (o "piccolo")
- 10n^2+3n=O(n^2)

4. E' vero o meno che 10n^2=o(n^2) ?
Non è vero, perché basta moltiplicare n^2 per qualunque costante > 10 e la parte destra è più grande della sinistra, per ogni n.

5. Mostrare che, se T(n)=2T(n/3)+3 allora T(n)=\O(nlogn)

6. Heap: mostrare che il metodo moveDown (page 260 del libro) ha costo \Omega(log n) nel caso peggiore. Si tratta di osservare che nel caso peggiore la radice va spostata fino all'altezza delle foglie dell'albero heap e che quest'ultimo ha altezza \Theta(log n).

7. Dedurre che, poiché l'ordinamento di n elementi usando algoritmi basati sul confronto ha costo \Omega(log n) nel caso peggiore, non è possibile costruire un albero binario di ricerca usando un metodo basato su confronti in tempo migliore di \Omega(n log n).
Bisogna osservare che la visita simmetrica di un albero binario di ricerca richiede tempo O(n) e stampa gli elementi ordinati per chiave. Dunque, se l'affermazione precedente fosse falsa, per ordinare n elementi basterebbe costruire un albero binario in tempo o(nlog n) (ragionando per assurdo) e poi fare una visita simmetrica. Ma allora avremmo trovato un algortimo di ordinamento con tempo o(n log n) nel caso peggiore. Questo è un assurdo.

8. Si consideri la ricerca binaria ricorsiva in un albero binario, descritta dai seguenti metodi overloaded:

```
boolean recSearch(BinaryTree T, BinaryNode v, int val) {
   if (val == v.valore)
      return true;
   else if (val < v.valore)
      return recSearch(T, v.leftChild, val);
   else return recSearch(T, v.rightChild, val);
}

boolean recSearch(BinaryTree T, val) {
   return recSearch(T, T.root, val);
}
```

Mostrare che tale procedura ha costo $O(h)$, dove h è l'altezza dell'albero.

Il costo $T(h)$ della procedura si può scrivere così:

$T(h) = c_1$ (costo if - else if -else) + $T(h-1)$ (perché sono sceso di un livello).
A questo punto basta sostituire, nella parte destra, $c(h-1)$ al posto di $T(h-1)$ (c è una costante da determinare) e il gioco è fatto.