Esercizi - 1

1. Si abbia a disposizione una classe **Pila** che implementi il tipo astratto *pila* su elementi di tipo **int.** Sia **Pila()** il costruttore (crea una pila vuota) e siano **int pop(), int top(), void push()** e **boolean isEmpty()** i metodi che implementano le operazioni corrispondenti del tipo astratto. Si scriva un metodo **Pila copia(Pila p)** che restituisca una nuova pila avente lo stesso contenuto di **p**. Si osservi che il metodo deve lasciare inalterato il contenuto di **p** alla fine dell'esecuzione (lo può modificare durante). Si può usare al più un'altra pila per risolvere il problema.

2. Si abbia a disposizione la classe **Pila** precedente. Si scriva un metodo **Pila inverti(Pila p)** che restituisce una nuova pila ottenuta invertendo il contenuto di **p**. Si osservi che il metodo deve lasciare inalterato il contenuto di **p** alla fine dell'esecuzione (lo può modificare durante).

3. Calcolare le complessità degli algoritmi di cui ai punti precedenti.

4. Si abbiano a disposizione la classe **Pila** di cui al punto 1 e una classe **Coda** che implementi il tipo astratto *coda* su elementi di tipo **int**. Sia **Coda()** il costruttore (crea una pila vuota) e siano **int remove(), int head(), void insert()** e **boolean isEmpty()** i metodi che implementano le operazioni corrispondenti del tipo astratto. Si scriva un metodo **Pila copia(Pila p)** che restituisca una nuova pila avente lo stesso contenuto di **p**. Si cerchi di ottenere un metodo asintoticamente più efficiente di quello ottenuto al punto 2. Si osservi che il metodo deve lasciare inalterato il contenuto di **p** alla fine dell'esecuzione (lo può modificare durante).

5. Considerare la seguente versione del MergeSort, in cui n si assume per semplicità potenza di 3:

```
MergeSort(int A[1,..,n]) {
        if(n == 1)
                return;
        MergeSort(A[1,…,n/3]);
        MergeSort(A[(n/3)+1,…,(2/3)*n];
        MergeSort(A[(2/3)*n+1,…,n]);
        merge(A[1,…,n/3]), A[(n/3)+1,…,(2/3)*n], A[(2/3)*n+1,…,n]); /*merge ha costo O(n) */
        return;
}
```

Mostrare che l'algoritmo ha complessità O(n logn).

6. Si consideri una classe **BinTree** che implementi il tipo *albero binario*. La classe è costruita a partire dalla classe **BinNode**, che implementa il generico nodo dell'albero binario. Si supponga che le due classi abbiano già, tra gli altri, i seguenti campi e metodi:

- **BinNode BinNode.left (BinNode BinNode.right)** → restituisce il fratello sinistro (destro) del nodo in considerazione.
- **BinNode BinTree.root** → restituisce la radice dell'albero binario.
- **boolean BinTree.isLeaf(BinNode v)** → restituisce **true** se **v** è una foglia, **false** altrimenti.

Scrivere un metodo **int height(BinNode v)** della classe **BinTree** che restituisce l'altezza del sottoalbero avente per radice **v**. Scrivere un metodo **boolean balanced(BinNode v)** della classe **BinTree** che verifichi se il sottoalbero avente radice **v** sia perfettamente bilanciato o meno. Scrivere un metodo **boolean balanced()** della classe **BinTree** che verifichi se l'albero binario sia perfettamente bilanciato o meno. Per semplicità si assuma la seguente definizione di albero binario perfettamente bilanciato: tutti i livelli (compreso quello delle foglie) devono essere completi.

Sono riportate di seguito le soluzioni degli esercizi più significativi. In genere si tratta di tracce di soluzione piuttosto che di soluzioni complete, allo scopo di obbligare lo studente a ripercorrere autonomamente la soluzione.

Soluzione punto 1:

```
Pila copia (Pila p) {
Pila temp = new Pila();
int n=0, i=0;
int el=0;

        while (!p.isEmpty()) {
                temp.push(p.pop());
                n++;
        } /* while1 */
        while (!temp.isEmpty()) /* while2 */
                p.push(temp.pop());
        /* Conta elementi presenti nella pila */
        i=n;
        while (i>0) { /* while3 */
                for (int j=0; j<i-1; i++)
                        temp.push(p.pop());
                el=p.top();
                for (int j=0; j<i-1; i++)
                        p.push(temp.pop());
                temp.push(el);
                i--;
        }
        /* Copia */
        return temp;
}
```

Soluzione punto 2:

Nel primo caso Semplicissimo: basta trasferire il contenuto di **p** in due pile, diciamo **p2** e **p3**. Il contenuto di **p2** e **p3** sarà ovviamente quello di **p** invertito. Si tratterà poi di ripristinare il contenuto originario di **p**. Ciò si può fare, ad esempio ritrasferendo il contenuto di **p2** in **p**. A questo punto il metodo può restituire **p3**, che conterrà il contenuto di **p** invertito.

Soluzione punto 3:

Nel primo caso il costo è $O(n^2)$. Si osservi infatti che il costo dei primi due cicli while (while 1 e while2) è chiaramente O(n) se n è il numero di elementi nella pila, mentre le operazioni dominanti sono quelle eseguite all'interno dei cicli for contenuti nel ciclo while3. Quest'ultimo è eseguito n volte. La i-esima volta, il ciclo for più interno (non importa quale dei due, poiché sono eseguiti lo stesso numero di volte) è eseguito i volte. Quindi, ciascuna delle istruzioni dominanti (ad esempio temp.push(p.pop());) è eseguita un numero di volte dato da:

$$n+(n-1)+(n-2)+\ldots..+2+1=n(n+1)/2 = O(n^2)$$

Questo contributo determina la complessità asintotica.

Soluzione punto 6:

Il calcolo dell'altezza è stato visto a lezione. Supponendo di aver implementato il metodo **int height(BinNode v), boolean balanced(BinNode v)** può essere realizzata così:

```
boolean balanced (BinNode v) {


        if  (isLeaf(v))
                return true;
        else
                return  balanced(v.left)&&(balanced(v.right)&&(height(v.left)-height(v.right) == 0);
}
```

L'espressione booleana restituita dalla seconda return si giustifica osservando che il sottoalbero radicato in **v** è perfettamente bilanciato secondo la definizione data sopra se le seguenti condizioni sono verificate: i) i sottoalberi destro e sinistro sono a loro volta perfettamente bilanciati; ii) la differenza tra le altezze dei sottoalberi destro e sinistro è 0.

```
boolean balanced () {


        return balanced(root);
}
```