Qualche esercizio su analisi ammortizzata

1. Si consideri una estensione del tipo di dati `Pila` in cui, oltre alle usuali operazioni `Push`(e), `Pop`, `Top`, è definita l'operazione aggiuntiva $k-$`Pop`(k), che restituisce i k elementi in cima alla pila. Mostrare che quest'ultima operazione può essere implementata in tempo ammortizzato $O(1)$ o, in alternativa, che essa richiede tempo ammortizzato $\Omega(1)$.

   L'operazione richiede tempo ammortizzato $O(1)$: per mostrare ciò, associamo ad ogni operazione `Push`(e) *1 credito aggiuntivo, che potrà essere "speso" per l'esecuzione della $k-$* `Pop`(k) *che estrae e dalla pila. Alternativamente, definiamo il potenziale Φ come pari al numero di elementi nella pila: il costo ammortizzato della* `Push`(e) *è allora $\hat{c} = c + \Phi' - \Phi = O(1) + 1 = O(1)$ e, allo stesso modo, della* `Pop` *è $\hat{c} = O(1) - 1 = O(1)$, della* `Top` *$\hat{c} = O(1)$ e infine della $k-$* `Pop`(k) *$\hat{c} = (k + O(1)) - k = O(1)$.*

2. Si consideri una estensione del tipo di dati `Pila` in cui, oltre alle usuali operazioni `Push`(e), `Pop`, `Top`, è definita l'operazione aggiuntiva $k-$`Push`(k, e), che inserisce k copie dell'elemento e in cima alla pila. Mostrare che quest'ultima operazione può essere implementata in tempo ammortizzato $O(1)$ o, in alternativa, che essa richiede tempo ammortizzato $\Omega(1)$.

   L'operazione richiede tempo ammortizzato $\Omega(1)$ e, in particolare, lineare in k: infatti, intuitivamente, non possiamo associare i crediti utilizzabili per portare il costo di esecuzione di $k-$ `Push`(k, e) *da k come singola operazione a $O(1)$ in termini ammortizzati. In altri termini, consideriamo come sequenza peggiore una sequenza di lunghezza costante, m, di $k-$* `Push`(k, e)*: indicando con n la dimensione risultante della pila, il costo di esecuzione della sequenza è allora necessariamente n e, in termini ammortizzati, ogni $k-$* `Push`(k, e) *ha costo $\frac{n}{m} = \Theta(n)$*

3. Si consideri una estensione del tipo di dati `Pila` in cui, oltre alle usuali operazioni `Push`(e), `Pop`, `Top`, sono definite entrambe le operazioni aggiuntive $k-$`Push`(k, e) e $k-$`Pop`(k). Mostrare che queste ultime operazioni possono essere implementate entrambe in tempo ammortizzato $O(1)$ o, in alternativa, che almeno una di esse richiede tempo ammortizzato $\Omega(1)$.

   Valgono le considerazioni al punto precedente, relativamente a $k-$ ***Push***(k, e)

4. Si consideri un tipo di dati rappresentato da una lista di interi su cui sono definite le seguenti operazioni

   - `Append`(x): appende l'intero x in fondo alla lista
   - `Extract`$()$: estrae e restituisce l'intero in fondo alla lista
   - `Lookup`(k): restituisce (senza estrarlo) l'intero nella k-esima posizione della lista

   Definire una implementazione della struttura di dati avente tempo ammortizzato $O(1)$ per le prime due operazioni e tempo per singola operazione $O(1)$ per l'ultima.

   La lista di interi può essere implementata mediante un array di dimensione iniziale k*: ad ogni istante, gli* $n \leq k$ *elementi dell'insieme sono memorizzati nelle prime* n *locazioni dell'array, memorizzando il valore* n *in una variabile aggiuntiva. La* ***Append***(x) *viene implementata inserendo* x *alla locazione* $n+1$ *(in tempo* $O(1)$*), se* $n < k$*; altrimenti è necessario allocare un nuovo array di dimensione* $2k$*, copiando nelle stesse posizioni i* $k+1$ *elementi nel nuovo array (in tempo* $O(1)+k$*). Il costo dell'operazione può essere ammortizzato associando ad ogni* ***Append***(x) *2 crediti aggiuntivi. Si osservi che, in corrispondenza ad una* ***Append***(x) *"costosa", effettuata in quanto un array di dimensione* k *è riempito, almeno la metà dei* k *elementi non sono stati coinvolti in* ***Append***(x) *"costose" precedenti: ne deriva che* $2 \cdot \frac{k}{2} = k$ *crediti sono disponibili per ammortizzare a* $O(1)$ *il costo della* ***Append***(x)*. La* ***Extract***$()$ *può essere effettuata in tempo costante (per operazione e quindi ammortizzato) eliminando dall'array l'elemento di indice* n *e indicando che il numero di elementi diviene* $n-1$*. La* ***Lookup***(k) *può essere effettuata accedendo all'elemento di indice* k *nell'array.*

5. Definire una implementazione di una coda per mezzo di due stack (e una quantità costante di memoria aggiuntiva), in modo tale che

le operazioni di inserimento ed estrazione dalla coda abbiano costo ammortizzato $O(1)$.

L'inserimento viene effettuato effettuando una `Push` *sul primo stack* $S1$*; l'estrazione effettuando una* `Pop` *dal secondo stack* $S2$*. Se* $S2$ *è vuoto al momento di una* `Pop`*, gli elementi in* $S1$ *vengono estratti uno dopo l'altro (mediante* `Pop`*) e inseriti, in ordine inverso in* $S2$ *(mediante* `Push`*). Definendo il potenziale come pari al numero di elementi in* $S1$*, è facile osservare che il costo per operazione di un inserimento è* $O(1)$ *e il costo ammortizzato di una estrazione è ancora* $O(1)$*.*

6. Mostrare che, in un heap binario, il costo ammortizzato di una `Insert` è $O(\log n)$ e il costo ammortizzato di `ExtractMin` è $O(1)$.

   (Traccia) Si consideri una funzione potenziale pari alla somma, su tutti gli elementi dello heap, del livello di ogni elemento nello heap (1 per la radice, 2 per i suoi due figli, etc.).

7. Sia $G = (V, E, c)$, dove $c : E \mapsto \{bianco, blu\}$ un grafo di n nodi colorato sugli archi, in cui inizialmente $E = \emptyset$. Si considerino le seguenti operazioni

   - `InsertEdge`(u, v): inserisce l'arco (u, v) con $c(u, v) = bianco$ in E
   - `ColorEdge`(u, v): pone $c(u, v) = blu$
   - `ColorIncidentEdges`(u): pone $c(u, v) = blu$ per tutti gli archi incidenti al nodo u
   - `Blu`(u, v): restituisce `TRUE` se $c(u, v) = blu$, `FALSE` altrimenti

   Definire una implementazione efficiente delle operazioni precedenti. Descrivere le relative complessità sia per operazione che ammortizzate.

   (Traccia) Le operazioni `InsertEdge`(u, v)*,* `ColorEdge`(u, v)*,* `Blu`(u, v) *sono implementabili in tempo* $O(1)$ *utilizzando una rappresentazione medinate matrice di transizione del grafo. È sufficiente inoltre mostrare che la* `ColorIncidentEdges`(u) *può essere eseguita in tempo* k*, dove* k *è il numero di archi bianchi incidenti a* u*: in questo modo, il suo costo può essere ammortizzato sui* k *inserimenti effettuati.*

8. Dato il problema precedente, si consideri, in aggiunta alle operazioni descritte, l'operazione aggiuntiva `UncolorEdge`(u, v), che pone $c(u, v) =$

bianco. Definire una implementazione efficiente di tutte le operazioni. Descrivere le relative complessità sia per operazione che ammortizzate.

La nuova operazione ha costo $O(1)$, non modificando l'analisi del problema precedente.

9. Dato il problema precedente, si consideri, in aggiunta alle operazioni descritte, l'operazione aggiuntiva `BluePath`(u, v), che restituisce `TRUE` se esiste un cammino di archi di colore blu da u a v, `FALSE` altrimenti. Definire una implementazione efficiente di tutte le operazioni. Descrivere le relative complessità sia per operazione che ammortizzate.

   (Traccia) Si rifletta sull'utilizzo di una struttura Union-Find.

10. Dato il problema precedente, definiamo con $BlueCloud(v)$ l'insieme dei nodi connessi a v mediante un cammino di archi blu. Si consideri, in aggiunta alle operazioni descritte, l'operazione aggiuntiva `RemoveBlueCloud`(v), che elimina tutti gli archi tra coppie di nodi $u, w \in BlueCloud(v)$. Definire una implementazione efficiente di tutte le operazioni. Descrivere le relative complessità sia per operazione che ammortizzate.

    (Traccia) Una soluzione ammortizzata efficiente può essere ottenuta se si riesce a effettuare `RemoveBlueCloud`(v) in tempo k, se k è il numero di archi coinvolti.