# Problem Set 1 (con soluzioni)

docente: Luciano Gualà

**Esercizio 1** (*riscaldamento*)
Si consideri l'algoritmo `Fibonacci3` presentato a lezione che, preso in ingresso un intero $n$, calcola il numero di Fibonacci di ordine $n$. Si assuma che la linea di codice $j$-esima esegua $c_j$ operazioni elementari. Si dimostri formalmente che la complessità temporale dell'algoritmo è $\Theta(n)$. Si ricordi che il modello di calcolo considerato è quello RAM a costi uniformi.

**Soluzione esercizio 1** Si fa riferimento allo pseudocodice presentato a lezione.

Sia $T(n)$ il numero di passi elementari che `Fibonacci3` esegue con input $n$. Poiché le linee 1,2 e 5 vengono eseguite una sola volta, mentre le linee di codice 3 e 4 vengono eseguite al più $n$ volte, abbiamo:

$$T(n) \leq c_1 + c_2 + c_5 + c_3 n + c_4 n = c_1 + c_2 + c_5 + (c_3 + c_4)n = O(n).$$

Per dimostrare che l'upper bound al numero di operazioni eseguite è asintoticamente stretto (e che quindi $T(n) = \Theta(n)$), basta osservare che la linea di codice 4 è eseguita *almeno* $n-2$ volte. Quindi, abbiamo:

$$T(n) \geq c_4(n-2) = \Omega(n).$$

**Esercizio 2** (*notazione asintotica*)
Siano $f(n), g(n), h(n)$ tre funzioni asintoticamente positive. Inoltre, sia $c > 1$ una costante reale positiva. Si dimostrino o confutino le seguenti affermazioni:

1. $2^{f(n)+2^c} = \Theta(2^{f(n)})$.
2. $g(n) = \Theta(1)$ implica $2^{f(n)+g(n)} = O(2^{f(n)})$.
3. $g(n) = o(f(n))$ implica $2^{f(n)+g(n)} = O(2^{f(n)})$.
4. $f(n) + g(n) + h(n) = \Theta(\max\{f(n), g(n), h(n)\})$.
5. $f(n) = \Theta(\log n)$ implica $\log n^{f(n)} = O(\log^c n^{g(n)})$.
6. $f(n) = \Theta(f(c \cdot n))$.
7. $f(n) = \Theta(f(c + n))$.

**Soluzione Esercizio 2**

1. *Vera.* Infatti $2^{f(n)+2^c} = 2^{f(n)}2^{2^c} = \Theta(2^{f(n)})$, perché $2^{2^c}$ è una costante.
2. *Vera.* Infatti, $g(n) = \Theta(1)$ implica che esistono due costanti $c > 1, n_0$ tale che $g(n) \leq c$ per ogni $n \geq n_0$. Quindi, quando $n \geq n_0$, abbiamo $2^{f(n)+g(n)} = 2^{f(n)}2^{g(n)} \leq 2^{f(n)}2^c = O(2^{f(n)})$, perché $2^c$ è una costante.
3. *Falsa.* Un controesempio è: $g(n) = \sqrt{n}$, $f(n) = n$. Chiaramente $2^{n+\sqrt{n}} = 2^n 2^{\sqrt{n}} = \omega(2^n)$.

4. *Vera.* Infatti, poiché le funzioni sono asintoticamente positive, sicuramente abbiamo che per $n$ sufficientemente grande:

$$\max\{f(n), g(n), h(n)\} \leq f(n) + g(n) + h(n) \leq 3\max\{f(n), g(n), h(n)\},$$

da cui segue la tesi (le costanti della definizione di $\Theta(\cdot)$ sono $c_1 = 1$ e $c_2 = 3$ e $n_0$ il valore dopo il quale tutte le funzioni sono sempre positive).

5. *Falsa.* Poiché si ha che $\log n^{f(n)} = f(n)\log n$ e $\log^c n^{g(n)} = g(n)^c \log^c n$, un controesempio è: $f(n) = \log n$, $g(n) = 1$ e $c = 1.5$.

6. *Falsa.* Un controesempio è $f(n) = 2^n$ e $c = 2$. Infatti $2^n = o(2^{2\cdot n})$.

7. *Falsa.* Un controesempio è $f(n) = 2^{2^n}$ e $c = 2$. Infatti $2^{2^n} = o(2^{2^{n+2}})$, perché $2^{2^{n+2}} = 2^{2^n 4} = (2^{2^n})^4$.

**Esercizio 3** Sia $A[1:n]$ un array di $n$ ordinato in modo non decrescente, tale che per ogni $i = 1, \ldots, n$, $A[i] \in \{1, 2, 3\}$. Si progetti un algoritmo con complessità temporale $O(\log n)$ che calcoli il numero di due presenti in $A$.

**Solizione Esercizio 3** L'idea è trovare velocemente l'indice $h$ dell'ultimo 1, e l'indice $k$ dell'ultimo 2. A questo punto, il numero di 2 sarà $k - h$. Per trovare velocemente tali indici è possibile usare l'idea della ricerca binaria.

Di seguito riportiamo lo pseudocodice dell'algoritmo `NumDiDue`$(A)$ che usa una procedura ausiliaria ricorsiva `UltimoX`$(A, x, i, j)$. Tale procedura ricorsiva restituisce in output l'indice dell'array $A$ che contiene l'ultimo $x$ (con $x = 1, 2$) nella porzione contigua di $A$ compresa fra gli indici $i$ e $j$. Se nessuna occorrenza di $x$ è presente in tale porzione, l'algoritmo restituisce il valore $-1$.

**Algorithm 1:** `NumDiDue`$(A)$

```
n = size of A ;
if A[n] = 1 then
    return 0
if A[1] = 2 then
    h = 0
else
    h = UltimoX(A, 1, 1, n − 1)
if A[n] = 2 then
    k = n
else
    k = UltimoX(A, 2, 1, n − 1)
return (k − h)
```

Dove, la procedura ricorsiva ausiliaria è:

**Algorithm 2:** `UltimoX`$(A, x, i, j)$

```
if i > j then
  return −1
m = ⌊(i+j)/2⌋ ;
if A[m] = x e A[m + 1] > x then
  return m
if A[m] > x then
  return UltimoX(A, x, i, m − 1)
else
  return UltimoX(A, x, m + 1, j)
```

**Esercizio 4** Si consideri una tavoletta di cioccolata rettangolare composta da $n$ file di $m$ quadratini di cioccolata. Si vuole spezzarla in modo da avere tutti i quadratini di cioccolata separati. Una strategia consiste in una serie di *spezzate*, dove ogni spezzata è può essere vista come una procedura che prende un pezzo di cioccolata (di qualsiasi forma) e lo separa in due pezzi di cioccolata (di qualsiasi forma). Una semplice strategia è quella di separare prima le $n$ file eseguendo $n-1$ spezzate orizzontali, e poi per ognuna delle $n$ file eseguire $m-1$ spezzate verticali per separare i relativi quadratini. Questa strategia esegue complessivamente:

$$n - 1 + n(m - 1) = n - 1 + nm - n = nm - 1$$

spezzate. Esiste una strategia migliore, cioè una strategia che separa tutti i quadratini eseguendo un numero minore di spezzate? Si argomenti la risposta.

**Solizione Esercizio 4** Dimostriamo che *ogni* strategia effettua esattamente $mn - 1$ spezzate, e che quindi la strategia proposta è ottima. Forniremo due argomentazioni.

*Prima argomentazione.* E' possibile vedere una strategia di spezzate come un albero binario (radicato) $T$ con le seguenti caratteristiche. Ogni nodo rappresenta un pezzo di cioccolata dove: (i) la radice rappresenta la tavoletta intera, e (ii) se un nodo interno rappresenta un certo pezzo $x$, i suoi due figli rappresentano i due pezzi $y$ e $y'$ ottenuti una volta che la data strategia ha spezzato $x$. Ora è facile vedere che $T$ ha esattamente $mn$ foglie, una ogni quadratino, e che ogni nodo interno ha esattamente due figli. Inoltre il numero di spezzate è uguale al numero di nodi interni. Essi quindi devono essere necessariamente $mn - 1$, perché sono esattamente uno in meno del numero di foglie (Lemma 1.2 del libro di testo).

*Seconda argomentazione.* E' possibile ragionare nel seguente modo. Per come abbiamo definito le regole, ogni volta che si effettua una spezzata, il numero complessivo di pezzi aumente di (esattamente) uno, perché prendiamo un pezzo e lo dividiamo in due. All'inizio il numero di pezzi di cioccolata è uguale a 1, perché la tavoletta è intera, e alla fine il numero di pezzi è esattamente $mn$. Quindi il numero di spezzate deve essere $mn - 1$.