Problem Set 1

docente: Luciano Gualà

Esercizio 1 (*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 1

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 2 (*trovare l'intero mancante*)
Sia $A[1 : n]$ un vettore ordinato di n interi distinti compresi fra 1 e $n + 1$. Chiaramente A contiene tutti gli elementi dell'insieme $\{1, 2, \ldots, n+1\}$ tranne uno. Progettare un algoritmo con complessità temporale $o(n)$ che trova l'elemento mancante.

Soluzione Esercizio 2 Descriviamo un algoritmo di complessità temporale $O(\log n)$ che restituisce l'intero mancante. L'idea è quella di usare l'approccio della ricerca binaria. Infatti, guardando una coppia adiacente di elementi, per esempio in posizione $i-1$ e i possiamo capire se i è l'elemento mancante o se l'elemento mancante è più grande di i (e quindi dobbiamo cercarlo alla destra di questi elementi) o se è più piccolo (e quindi dobbiamo cercarlo alla sinistra). Lo pseudocodice riportato di seguito si spiega da solo. L'unico dettaglio tecnico del codice è il seguente: poiché ad ogni passo stiamo guardando due elementi vicini, dobbiamo stare attenti che gli indici dell'array non escano dal range. Questo è ottenuto, nel codice, controllando a parte se l'elemento mancante è 1. Un'altra cosa che per comodità tecnica è fatta a parte è contollare se l'elemento mancante non sia $n+1$. La complessità temporale dell'algoritmo è uguale a quella della ricerca binaria e quindi è $O(\log n)$.

Algorithm 1: `InteroMancante`(A)

```
n = lunghezza di A ;
if A[1] = 2 then
    return 1
if A[n] = n then
    return n + 1
return InteroMancanteRic(A, 2, n)
```

Dove, la procedura ricorsiva ausiliaria è:

Algorithm 2: `InteroMancanteRic`(A, i, j)

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

Esercizio 3 *(Uno strano algoritmo di ordinamento)*
L'algoritmo ricorsivo `gualasort` è un algoritmo di ordinamento ricorsivo, la cui chiamata iniziale `gualasort`$(V, 1, n)$ ordina il vettore $V[1; n]$ di n numeri su cui è chiamato. O almeno si spera. Dimostrate la correttezza dell'algoritmo e discutetene la sua complessità computazionale.

Algorithm 3: `gualasort`(V, i, j)

```
N ← j − i + 1 ;
if N = 2 then
    if V[i] > V[i + 1] then
        scambia V[i] e V[i + 1]
if N > 2 then
    m1 = i + ⌊N/3⌋ ;
    m2 = i − 1 + ⌈(2/3)N⌉ ;
    gualasort(V, i, m2);
    gualasort(V, m1, j);
    gualasort(V, i, m2);
```

Soluzione Esercizio 3 Discutiamo prima la correttezza dell'algoritmo, facendo vedere che la chiamata `gualasort`(V, i, j) ordina in modo corretto la porzione dell'array $V[i; j]$ di $N = j - i + 1$ elementi.

L'idea dell'algoritmo è quella di considerare tre porzioni contigue di $V[i; j]$, ovvero $V[i; m_1 - 1]$, $V[m_1; m_2]$ e $V[m_2 + 1; j]$, ognuna di dimensione circa $N/3$ e di ordinare ricorsivamente prima la prime due porzioni, ovvero $V[i; m_2]$, poi le seconde due porzioni, ovvero $V[m_1; j]$, e infine di nuovo le prime due, $V[i; m_2]$. I casi base, invece, sono costituiti da porzioni di array di un solo elemento (in questo caso l'algoritmo non fa niente, perché non c'è niente da ordinare) o di due elementi (in questo caso l'algoritmo ordina i due elementi confrontandoli e eventualmente scambiandoli).

L'osservazione cruciale è la seguente:

Proprietà 1 *Il numero di elementi in $V[m_1; m_2]$ è maggiore o uguale al numero di elementi in $V[m_2 + 1; j]$.*

Dim. Infatti, il numero di elementi di $V[m_1; m_2]$ è:

$$m_2 - m_1 + 1 = i - 1 + \left\lceil \frac{2}{3}N \right\rceil - (i + \left\lfloor \frac{N}{3} \right\rfloor) + 1 = \left\lceil \frac{2}{3}N \right\rceil - \left\lfloor \frac{N}{3} \right\rfloor \geq N/3;$$

mentre il numero di elementi di $V[m_2 + 1; j]$ è:

$$j - m_2 = j - i + 1 - \left\lceil \frac{2}{3}N \right\rceil = N - \left\lceil \frac{2}{3}N \right\rceil \leq N/3.$$

□

Sia $k = j - m_2$ la dimensione di $V[m_2 + 1; j]$. Si considerino ora i k elementi più grandi contenuti in $V[i, j]$. Per la Proprietà 1, si ha che:

- dopo la prima chiamata `gualasort`(V, i, m_2), i k elementi più grandi di $V[i, j]$ devono trovarsi in $V[m_1; j]$; da cui segue che:
- dopo la seconda chiamata `gualasort`(V, m_1, j), i k elementi più grandi di $V[i, j]$ sono ordinati e si trovano in $V[m_2 + 1; j]$; da cui segue infine che:
- dopo la terza chiamata `gualasort`(V, i, m_2) il vettore $V[i; j]$ è ordinato.

Per quanto riguarda la complessità dell'algoritmo, essa può essere descritta dall'equazione di ricorrenza:

$$T(n) = 3T\left(\frac{2n}{3}\right) + O(1),$$

che, risolta con il teorema Master, dà come soluzione $T(n) = \Theta(n^{\log_{3/2} 3})$. Dato che $\log_{3/2} 3 = 2.709511..$, l'algoritmo ha complessità $\omega(n^{2.7})$ ed è quindi peggiore per esempio del `SelectionSort`.[1]

Esercizio 4 *(Un gioco di rimozione di pedine)*
Il gioco si gioca con un vettore di n elementi. Inizialmente, ogni posizione del vettore può essere vuota o contenere una pedina bianca o contenere una pedina nera. Ad ogni turno potete eseguire una mossa del seguente tipo: potete prendere due pedine di diverso colore che si trovano in posizioni adiacenti e rimuoverle dal vettore (e quindi le posizioni prima occupate da queste due pedine dopo la mossa diventano vuote). Il gioco termina quando non potete più fare nessuna mossa. Il vostro obiettivo è rimuovere il maggior numero possibile di pedine dal vettore.

Progettate un algoritmo veloce che, presa un'istanza del gioco, calcola una strategia ottima, ovvero una sequenza di mosse che rimuove il massimo numero di pedine. Dimostrate la correttezza dell'algoritmo e fornitene l'analisi della complessità computazionale.

Soluzione Esercizio 4 L'algoritmo che proponiamo è un algoritmo *greedy*, ovvero un algoritmo *goloso*: fra tutte le mosse disponibili in un certo istante di tempo, l'algoritmo esegue quella che sembra più appetibile. Ma quale è la mossa più appetibile in un certo momento? Se si pensa abbastanza a lungo al problema, ci si rende conto che alcune mosse possono portare a soluzioni non ottimali. Per esempio, se l'istanza è costituita da quattro pedine allineate, di colori alternati, per esempio bianca-nera-bianca-nera, una mossa "non ottimale" è quella di rimuovere le due pedine centrali, inibendo quindi la rimozioni delle due pedine esterne. E' chiaro che in questo caso la soluzione ottima è quella di rimuovere prima le prime due pedine più a sinistra, e poi rimuovere le ultime due (rimuovendo quindi quattro pedine invece di due). Insomma, visto che la prima pedina bianca a sinistra può essere accoppiata solo con quella nera alla sua destra, perché non farlo subito? Questo ragionamento porta ad un'interessante nozione di appetibilità: la mossa più appetibile è quella che rimuove le due pedine più a sinistra. Una domanda a questo punto è lecita: siamo sicuri che facendo così rimuoviamo sempre il maggior numero di pedine? La risposta a questa domanda è sì, ed è quello che dimostreremo quando argomenteremo la correttezza dell'algoritmo. Descriviamo però prima in modo più dettagliato l'algoritmo.

Definiamo con `rimuovi`(i) la mossa che rimuove le pedine in posizione i e $i+1$, qualora nelle posizioni i e $i+1$ siano presenti due pedine di colore diverso (in tal caso diciamo che la mossa è ammissibile). L'idea dell'algoritmo quindi, come già accennato, è quella di scegliere, ad ogni istante di tempo, fra tutte le mosse disponibili, la mossa che rimuove le pedine più a sinistra, ovvero la mossa `rimuovi`(i) scegliendo il più piccolo i ammissibile.

Di seguito è mostrato un possibile pseudocodice. Si assume che l'istanza è codificata con un vettore $V[1;n]$ di n elementi e che per ogni posizione i, $V[i] \in \{B, N, \perp\}$ specifica se tale posizione contiene una pedina bianca (B), nera (N), o è una posizione vuota ($\perp$). La sequenza delle mosse è salvata dentro una lista L.

[1] Da cui possiamo dedurre che il prof. Gualà non diventerà famoso per questo suo algoritmo.

Algorithm 4: `CalcolaMosse`(V)

$L \leftarrow \emptyset$;
for $i = 1, 2, \ldots, n-1$ **do**
    if $(V[i] \neq \perp) \wedge (V[i+1] \neq \perp) \wedge (V[i] \neq V[i+1])$ **then**
        appendi in coda a L la mossa `rimuovi`(i) ;
        $V[i] \leftarrow \perp$;
        $V[i+1] \leftarrow \perp$;
return L

La complessità temporale dell'algoritmo è chiaramente $O(n)$. Resta ancora da rispondere alla domanda: l'algoritmo trova sempre una soluzione ottima, ovvero rimuove sempre il maggior numero di pedine?

La correttezza dell'algoritmo segue immediatamente dalla seguente proprietà che, se applicata ripetutamente, mostra che l'algoritmo progettato restituisce sempre una soluzione ottima.

Proprietà 2 *Per ogni istanza del gioco, esiste sempre una soluzione ottima che rimuove le due pedine adiacenti di colore diverso più a sinistra.*

Dim. Si consideri una generica istanza di gioco e una sua soluzione ottima $\mathcal{O}pt$ (che quindi rimuove il maggior numero di pedine). E si considerino le due pedine adiacenti di colore diverso più a sinistra. Siano esse in posizione i e $i+1$. Si assuma che $\mathcal{O}pt$ non contenga la mossa `rimuovi`(i), altrimenti la tesi è chiaramente vera. Allora possiamo dedurre che $\mathcal{O}pt$ deve contenere la mossa `rimuovi`$(i+1)$. Infatti, se $\mathcal{O}pt$ non contenesse la mossa `rimuovi`$(i+1)$, potrei aggiungere la mossa `rimuovi`(i) ad $\mathcal{O}pt$ ottenendo una soluzione per l'istanza strettamente migliore di $\mathcal{O}pt$, violando così l'ottimalità di $\mathcal{O}pt$.

Quindi $\mathcal{O}pt$ contiene la mossa `rimuovi`$(i+1)$. Allora, si consideri l'insieme di mosse $\mathcal{O}pt'$ ottenuto da $\mathcal{O}pt$ sostituendo la mossa `rimuovi`$(i+1)$ con la mossa `rimuovi`(i). E' facile vedere che $\mathcal{O}pt'$ è ancora un insieme di mosse ammissibili e che $|\mathcal{O}pt'| = |\mathcal{O}pt|$. Da cui segue che anche $\mathcal{O}pt'$ è una soluzione ottima per l'istanza e inoltre soddisfa la proprietà descritta dall'enunciato. □