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
Sia $V[1; n]$ un vettore di n interi non necessariamente ordinato. Un indice $1 \leq i < n$ è detto *double-gap* se $V[i + 1] - V[i] \geq 2$. Si consideri un vettore V tale che $V[n] - V[1] \geq n$, e si argomenti sul fatto che tale vettore ha necessariamente almeno un indice double-gap. Poi si progetti un algoritmo che, dato un vettore V tale che $V[n] - V[1] \geq n$, ritorna un indice double-gap in tempo $O(\log n)$.

Soluzione Esercizio 2 Argomentiamo prima sul fatto che la condizione data sul primo e l'ultimo valore del vettore garantisce l'esistenza di un indice double-gap.

Lemma 1. *Sia V in vettore di n interi tale che $V[n] - V[1] \geq n$. Allora esiste sempre un indice double-gap in V.*

Dimostrazione. La dimostrazione è per induzione su n. Il caso base è $n = 2$. Si ha quindi che $V[2] - V[1] \geq 2$, e quindi 1 è un indice double-gap. La tesi è vera.

Consideriamo ora il caso $n > 2$. Se $V[2] - V[1] \geq 2$ allora 1 è un indice double-gap e quindi la tesi è vera. Altrimenti, $V[2] - V[1] \leq 1$, e quindi $V[1] - V[2] \geq -1$. Facciamo vedere che vale $V[n] - V[2] \geq n - 1$ da cui la tesi seguirà per ipotesi induttiva. Si ha:

$$V[n] - V[2] = \underbrace{V[n] - V[1]}_{\geq n} + \underbrace{V[1] - V[2]}_{\geq -1} \geq n - 1.$$

□

Ora, descriviamo un algoritmo di complessità temporale $O(\log n)$ che restituisce un indice double-gap. L'idea è quella di usare l'approccio della ricerca binaria. Infatti, guardando una coppia adiacente di elementi, per esempio in posizione i e $i+1$ possiamo capire se i è un indice double-gap; inoltre, se i non fosse un tale indice, possiamo controllare se il sottoarray alla sinistra o quello alla destra di i rispetta la condizione che ci garantisce l'esistenza di un indice double-gap, e in tal caso ricorrere in uno solo dei due sottoarray. Argomenteremo fra un po' sul fatto che uno dei due sottoarray deve necessariamente rispettare tale condizione, da cui poi segue la correttezza dell'algoritmo (per il Lemma 1).

Lo pseudocodice è riportato di seguito. L'algoritmo `GetDGric`(V, i, j) cerca un indice double-gap fra gli indici i e j (compresi) e, al momento della chiamata, assume che valga la condizione $V[j] - V[i] \geq j - i + 1$, cosa che, come abbiamo già argomentato (Lemma 1), garantisce l'esistenza di un tale indice. La correttezza dell'algoritmo segue essenzialmente dal fatto che, per ogni m, o vale $V[m] - V[i] \geq m - i + 1$ (cosa che garantisce l'esistenza di un indice double-gap in $[i, m]$), o vale $V[j] - V[m] \geq j - m + 1$ (cosa che garantisce l'esistenza di un indice double-gap in $[m, j]$). Infatti, se entrambe le condizioni fossero false, avremmo che $V[m] - V[i] \leq m - i$ e $V[j] - V[m] \leq j - m$ che, sommate insieme, darebbero $V[j] - V[i] \leq j - i$: questo è chiaramente in contrasto con l'assunzione iniziale $V[j] - V[i] \geq j - i + 1$.

Un 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, lanciando la procedura ricorsiva solo sulla porzione di indici che va da 1 a $n - 1$. La complessità temporale dell'algoritmo è uguale a quella della ricerca binaria e quindi è $O(\log n)$.

Algorithm 1: `GetDG`(V)

return `GetDGric`$(V, 1, n - 1)$

Dove, la procedura ricorsiva ausiliaria (che di fatto fa tutto il lavoro) è:

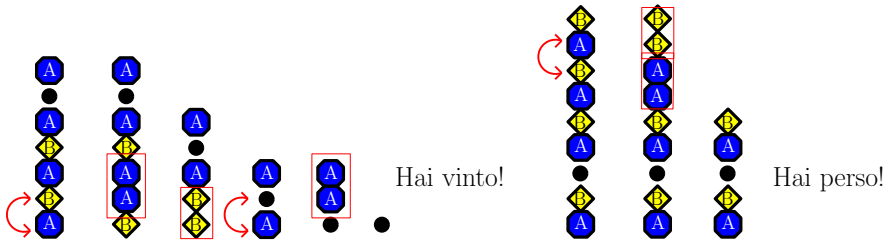

Figura 1: Due esempi di partita, una in cui il giocatore vince (eliminando tutte le caramelle), e una in cui perde. Il sassolino è rappresentato da un pallino nero.

Algorithm 2: `GetDGric`(V, i, j)

$m = \lfloor \frac{i+j}{2} \rfloor$;
if $V[m+1] - V[m] \geq 2$ **then**
  return m
if $V[m] - V[i] \geq m - i + 1$ **then**
  return `GetDGric`$(V, i, m-1)$
else
  `/* vale` $V[j] - V[m] \geq j - m + 1$ `*/`
  return `GetDGric`$(V, m+1, j)$

Esercizio 3 *(Un nuovo videogame)*
Gira voce di un imminente lancio da parte della King di un nuovo videogioco chiamato *Candy Column Crush*. La configurazione iniziale di gioco consiste in una colonna di n caramelle e un sassolino. Le caramelle sono di due tipi, A e B. L'obiettivo del gioco è rimuovere tutte le caramelle. Una sequenza di due o più caramelle dello stesso tipo è chiamata *blocco*. Il giocatore può effetture la seguente tipologia di mossa: scambiare il contenuto di due posizioni adiacenti. Se lo scambio porta il gioco in una configurazione in cui non ci sono blocchi, allora lo scambio non produce effetto e il contenuto delle due posizioni torna nella posizione iniziale. Altrimenti, tutti i blocchi che si sono formati scoppiano contemporaneamente e le caramelle rimaste, per forza di gravità, si ricompattano verso il basso; a questo punto, se si sono formati nuovi blocchi, anche questi scoppiano contemporaneamente e via dicendo, fino a che non ci sono più blocchi e il controllo del gioco è passato nuovamente al giocatore che può effettuare la prossima mossa. Alcuni esempi di partita sono mostrati in figura.

Si progetti un algoritmo (veloce) che, data una configurazione iniziale di gioco, trovi una sequenza di mosse in grado di rimuovere tutte le caramelle o, se questo non è possibile, segnali che il gioco non è risolvibile.

Soluzione Esercizio 3 Un algoritmo semplice è il seguente:

1. Scambia due caramelle sotto il sassolino finché puoi;
2. Scambia due caramelle sopra il sassolino finché puoi;
3. Scambia il sassolino finché puoi;
4. Se resta almeno una caramella rispondi "istanza non risolvibile".

Per quanto riguarda la complessità temporale, è facile convincersi che l'algoritmo può essere implemementato in modo che impieghi tempo $O(n)$. Qui, invece, la domanda principale è: l'algoritmo è corretto? Risponde sempre bene? Sa distinguere le istanze risolvibili da quelle non risolvibili? E' chiaro che se al punto 4 non sono rimaste caramelle, l'algoritmo ha trovato una sequenza di mosse che ha rimosso tutte le caramelle; in questo caso quindi l'istanza è risolvibile (e l'algoritmo l'ha effettivamente risolta) e l'algoritmo ha risposto correttamente. Ma cosa succede quando l'algortimo risponde "istanza non risolvibile"? Siamo effettivamente sicuri che stia rispondendo bene? In altre parole, la domanda può essere riformulata così: quando al punto 4 resta qualche caramella, siamo sicuri che l'istanza non era risolvibile usando una sequenza di mosse diversa da quella fatta dall'algoritmo? Argomenteremo che questo non può essere. Prima però dobbiamo capire meglio la combinatorica del problema.

Facciamo quindi qualche osservazione preliminare. Chiaramente, in ogni istante di tempo, ai due lati del sasso ci sono sempre due sequenze alternate di A e B (eventualmente vuote) che possono cominciare/finire con una A/B. Definiamo una sequenza di *tipo A* o di *tipo B* a seconda che la caramella più vicina al sasso sia A o B. Si consideri una sequenza (da un lato del sasso) composta da almeno 5 caramelle, e si consideri una mossa che ne scambia due. Allora valgono le seguenti osservazioni:

(i) ogni mossa fa scoppiare *esattamente* 4 caramelle;

(ii) dopo qualsiasi mossa, il tipo della sequenza non cambia (la caramella più vicina al sasso sarà sempre dello stesso tipo);

(iii) la sequenza può cambiare tipo solo se: (1) si scambiano due caramelle e la sequenza ha inizialmente 3 caramelle (in questo caso resta una sola caramella); oppure (2) ci sono esattamente 4 caramelle, se ne scambiano due e non resta niente, oppure (3) si scambia il sassolino con una caramella (ma questa mossa è possibile solo se le due sequenze ai lati del sasso sono dello stesso tipo).

Con queste osservazioni in mente, argomentiamo ora sulla correttezza dell'algoritmo. Facciamo essenzialmente vedere che ogni qual volta al passo 4 l'algoritmo resta con qualche caramella deve essere che l'istanza di partenza non era risolvibile (con alcuna sequenza di mosse). Cosa può restare al passo 4 dell'algoritmo ai due lati del sasso? Chiaramente le sequenze che restano ai due lati del sasso hanno una lunghezza di al più 2, altrimenti l'algoritmo è in grado di eseguire almeno un'altra mossa. I possibili casi (a meno di simmetrie) sono i seguenti (si assumerà senza perdita di generalità che la prima sequenza –se non vuota– cominci sempre per A):

0-0: In questo caso l'algoritmo ha trovato una sequenza di mosse che risolve l'istanza e risponde correttamente.

1-0: Resta solo una caramella da un lato del sasso. Allora, visto che le caramelle scoppiano a coppie, vuol dire che il numero iniziale di caramelle era dispari. Quindi l'istanza non era risolvibile.

1-1: Le due caramelle, una per ogni lato, devono essere diverse (o l'algoritmo le farebbe scoppiare al punto 3). Quindi ciò che resta deve essere: $A \bullet B$. In tal caso, vuol dire che il numero iniziale di A (e anche di B) era dispari. Allora l'istanza non era risolvibile.

2-0: In questo caso abbiamo $AB\bullet$. Ancora: in tal caso, vuol dire che il numero iniziale di A (e anche di B) era dispari. Allora l'istanza non era risolvibile.

2-1: Sono rimaste tre caramelle, un numero dispari. Ma allora vuol dire che il numero iniziale di caramelle era dispari. Quindi l'istanza non era risolvibile.

2-2: Non può essere che restano $AB \bullet BA$, altrimenti l'algoritmo risolveva l'istanza usando il passo 3. Resta da considerare il caso $AB \bullet AB$, quello più difficile. Si noti che in questo caso è vero che: l'algoritmo non ha fatto scoppi da due caramelle (o doveva essere rimasta una sola caramella da un lato); l'algoritmo non ha mai scambiato il sassolino (la prima volta che ci prova ci sono due caramelle per lato, ovvero $AB \bullet AB$). Quindi, per le proprietà (i), (ii) e (iii) fin dall'inizio le due sequenze ai due lati del sasso erano di tipi diversi. In particolare, per le stesse proprietà nessun'altra sequenza di mosse poteva cambiare il tipo alle sequenze. Quindi anche in questo caso l'istanza non era risolvibile, e l'algoritmo ha risposto bene.