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 $A[1:n]$ un vettore ordinato di n elementi booleani, ovvero di zeri e di uni. Si progettino due algoritmi che restituiscano il numero di zeri nel vettore. Il primo algoritmo deve avere complessità temporale $O(\log n)$, mentre il secondo deve avere una complessità computazionale di $O(\log k)$, dove k è il numero di zeri nel vettore. Algoritmi come quest'ultimo sono detti *output sensitive*, perché la loro complessità temporale è espressa in funzione dell'output dell'algoritmo (piuttosto che in funzione della dimensione dell'istanza). Si noti, inoltre, che la complessità richiesta per il secondo algoritmo non è mai peggiore di $O(\log n)$ ma può essere (anche asintoticamente) migliore per le istanze in cui k è molto più piccolo di n.

Soluzione Esercizio 2 Diamo prima un algoritmo di complessità temporale $O(\log n)$ che ritorna la posizione dell'ultimo zero, -1 se non ci sono zeri. L'idea è quella di usare l'approccio della ricerca binaria. Infatti, guardando una coppia adiacente di elementi, possiamo capire se abbiamo trovato l'ultimo zero del vettore o se dobbiamo cercarlo alla destra di questi elementi o 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'ultimo zero si trova in posizione n. La complessità temporale è uguale a quella della ricerca binaria e quindi è $O(\log n)$.

Algorithm 1: `UltimoZero`(A)

```
n = lunghezza di A ;
if A[n] = 0 then
    return n
else
    return UltimoZeroRic(A, 1, n − 1)
```

Dove, la procedura ricorsiva ausiliaria è:

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

```
if i > j then
    return −1
m = ⌊(i+j)/2⌋ ;
if A[m] = 0 e A[m + 1] = 1 then
    return m
if  A[m] = 1 then
    return UltimoZeroRic(A, i, m − 1)
else
    return UltimoZeroRic(A, m + 1, j)
```

Progettiamo ora un algoritmo che trova la posizione k dell'ultimo zero in tempo $O(\log k)$. L'idea è quella di trovare in tempo $O(\log k)$ due indici i^* e j^* con $i^* < j^*$ tale che (i) $A[i^*] = 0$ e $A[j^*] = 1$, e (ii) la porzione dell'array $A[i^* : j^*]$ contiene $O(k)$ elementi. Una volta trovati tali indici è possibile trovare in tempo $O(\log k)$ la posizione dell'ultimo zero facendo ricerca binaria (come nell'algoritmo precedente) sulla porzione $A[i^* : j^*]$.

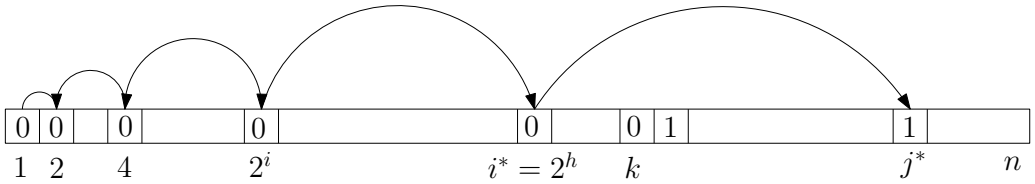

Figura 1: Idea dell'algoritmo con complessità $O(\log k)$ che trova la posizione k dell'ultimo zero.

Per trovare i^* e j^* procediamo nel seguente modo. Guardiamo gli elementi dell'array le cui posizioni sono potenze di due. Più precisamente, guardiamo gli elementi $A[2^i]$, per $i = 0, 1, 2...$, e ci fermiamo appena troviamo una posizione che contiene un 1 (o quando $2^i > n$). Sia $i^* = 2^h$ l'ultima posizione che abbiamo visto contenente il valore 0, mentre j^* lo definiamo come $j^* = \min\{2^{h+1}, n\}$ (si veda la Figura 1). L'osservazione cruciale è che l'ultimo zero si deve trovare a destra di i^* (potenzialmente è quello in i^* stesso), da cui segue che $k \geq i^* = 2^h$, per cui $h \leq \log_2 k$. Da questo immediatamente segue che la porzione di array $A[i^* : j^*]$ ha al più $2^{h+1} - 2^h + 1 = 2^h + 1$ elementi, e quindi ha dimensione $O(2^h) = O(k)$. Per finire, il numero di elementi che abbiamo visto prima di trovare i^* e j^* sono esattamente $h + 2$, perché abbiamo visto gli elementi in posizione 2^i per $i = 0, 1, \ldots, h + 1$. Quindi abbiamo visto $h + 2 = O(\log k)$ elementi, ognuno in tempo costante.

Esercizio 3 Si consideri una sequenza di n pedine allineate in cui ogni pedina è colorata di rosso, di verde o di bianco. Una mossa consiste nello scambiare di posto due pedine adiacenti della sequenza. Si vogliono riposizionare le pedine in modo che i loro colori formino la bandiara italiana (cioè compaiano nell'ordine, da sinistra a destra, prima tutte le pedine verdi, poi le bianche e infine le rosse).

Progettare un algoritmo che risolve il problema con un numero asintoticamente ottimo di scambi. Dimostrarne l'ottimalità.

Soluzione Esercizio 3 Possiamo pensare che ad ogni colore corrisponda un valore. Poiché vogliamo in ordine le pedine verdi, poi le bianche, e poi le rosse, possiamo pensare che al verde sia associato il valore 1, al bianco il valore 2 e al rosso il valore 3. In quest'ottica ri-arrangiare le pedine vuol dire ordinarle in ordine non decrescente facendo solo scambi di pedine adiacenti. Un approccio possibile è quello basato sul `BubbleSort`: l'algoritmo è composto di $n - 1$ fasi, numerate da $n - 1$ (la prima fase) a scendere fino a 1 (ultima fase). Nella fase i si guardano in ordine (da sinistra a destra) le pedine dalla prima fino alla pedina i e la generica pedina è scambiata con la successiva ogni volta che il suo valore è strettamente maggiore di quella che la segue immediatamente. E' facile vedere che dopo la prima fase la pedina di valore massimo è in ultima posizione, dopo la seconda fase il secondo massimo è in penultima posizione e via dicendo. Per quanto riguarda la complessità temporale, in questo caso misurata come numero di scambi (nel caso peggiore), è chiaro che ogni fase esegue al più n scambi e, poiché le fasi sono $n-1$, il numero di scambi effettuati dall'algoritmo nel caso peggiore è $O(n^2)$.

Questo algoritmo fa un numero asintoticamente ottimo di scambi. Per dimostrarne l'ottimalità procederemo nel seguente modo: mostreremo un'istanza del problema per la

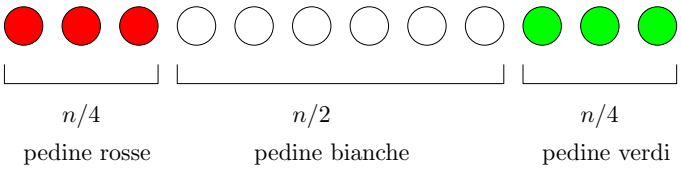

Figura 2: Istanza per la quale ogni algoritmo deve fare $\Omega(n^2)$ scambi di pedine (adiacenti).

quale un *qualsiasi* algoritmo (corretto) deve eseguire $\Omega(n^2)$ scambi. L'istanza è la seguente: ci sono $n/4$ pedine rosse, seguite da $n/2$ pedine bianche, seguite da $n/4$ pedine verdi (si veda la Figura 2). Ora, si consideri un *generico* algoritmo. Tale algoritmo per risolvere il problema deve certamente spostare tutte le $n/4$ pedine rosse dall'inizio alla fine della sequenza. Poiché ogni pedina può essere scambiata solo con la pedina adiacente, è chiaro che *ogni* pedina rossa dovrà essere scambiata (verso destra) *almeno* $n/2$ volte. Poiché le pedine rosse sono $n/4$, segue che il numero complessivo di scambi che l'algoritmo deve eseguire è almeno $\frac{n}{4} \cdot \frac{n}{4} = \frac{n^2}{16} = \Omega(n^2)$. Per cui il nostro algoritmo è asintoticamente ottimo.

Esercizio 4 (*problemi algoritmici, puzzle e modelli di calcolo.*) Si consideri una sequenza di n pedine allineate in cui ogni pedina è colorata di rosso, di verde o di bianco. Una mossa consiste nello scambiare di posto due pedine della sequenza (non necessariamente adiacenti). Si vogliono riposizionare le pedine in modo che i loro colori formino la bandiara italiana (cioè compaiano nell'ordine, da sinistra a destra, prima tutte le pedine verdi, poi le bianche e infine le rosse).

- Progettare un algoritmo che risolve il problema effettuando al più $O(n)$ scambi.
- Si noti che, nella precedente versione, si sta implicitamente assumendo un modello di calcolo in cui le uniche operazioni che hanno un costo sono le operaioni di scambio. In particolare, non è conteggiato il costo per individuare le pedine da scambiare. Si supponga adesso che le pedine siano elementi di un vettore $A[1 : n]$ e si progetti un algoritmo con complessità temporale $O(n)$ e complessità spaziale $O(1)$. Anche in questo caso l'unica mossa ammissibile è quella di scambiare due elementi del vettore, ma il costo computazionale dell'algoritmo è misurato come numero (asintotico) di operazione elementari nel modello RAM (a costi uniformi).

Soluzione Esercizio 4 Come abbiamo osservato nell'esercizio precedente, il problema di ri-arrangiare le pedine può essere visto come il problema di ordinarle in ordine non decrescente assumendo che al colore verde è associato il valore 1, al bianco il valore 2 e al verde il valore 3.

Concentriamoci prima sul problema di progettare un algoritmo che esegua $O(n)$ scambi. Visto che ora ci è consentito scambiare pedine non adiacenti, una soluzione semplice è usare una strategia simile al `SelectionSort`. Procediamo in modo incrementale in $n-1$ fasi. Nella fase i, si individua la pedina di valore minimo fra quelle in posizioni $i, i+1, i+2, \ldots, n$ e si scambia con quella in posizione i. Chiaramente, dopo la prima fase la pedina di valore minimo è messa in prima posizione, dopo la seconda fase il secondo minimo è messo in seconda posizione e così via. Per quanto riguarda gli scambi effettuati questi sono chiaramente $n-1 = O(n)$, come richiesto.

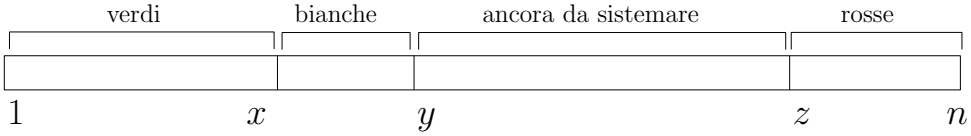

Figura 3: Invariante mantenuta dall'algoritmo lineare che risolve il problema dell'esercizio 4 (secondo punto).

Il secondo punto dell'esercizio, invece, è più delicato. Infatti l'algoritmo appena presentato se eseguito sul modello RAM ha una complessità di $\Theta(n^2)$. Questo dipende dal modo con cui misuriamo la velocità di un algoritmo che a sua volta dipende dal modello di calcolo di riferimento (sul quale eseguiamo i nostri algoritmi). Ad ogni modo un algoritmo con complessità temporale di $O(n)$ nel caso peggiore sul modello RAM è in questo caso possibile.

L'idea dell'algoritmo è quella di procedere in n fasi e mantenere la seguente *invariante*: ad ogni fase il mio array A è logicamente partizionato in quattro porzioni individuate dalle variabili x, y e z; più precisamente la porzione $A[1 : x]$ contiene solo pedine verdi, la porzione $A[x+1 : y-1]$ contiene pedine bianche, la porzione $A[y, z-1]$ contiene pedine potenzialmente di tutti i colori (queste rappresentano le pedine che vanno ancora sistemante), e la porzione $A[z : n]$ contiene solo pedine rosse (si veda la Figura 3).

L'algoritmo quindi in ogni fase considera la pedina in $A[y]$ e, dipendentemente dal colore, in tempo costante riduce di uno il numero di pedine che restano ancora da sistemare. Poiché le fasi sono n questo implicherà che la complessità temporale dell'algoritmo sarà $O(n)$. Più precisamente, se $A[y]$ è bianca semplicemente incrementiamo y (aumentando così di uno la porzione di pedine bianche). Se $A[y]$ è verde scambiamo $A[y]$ con $A[x]$ e incrementiamo y e x (aumentiamo così di uno la porzione di pedine verdi). Se $A[y]$ è rossa, scambiamo $A[y]$ con $A[z-1]$ e decrementiamo z (aumentiamo così di uno la porzione di pedine rosse). E' facile vedere che se all'inizio della fase vale l'invariante, tale invariante vale anche alla fine della fase. L'algoritmo termina quando non ci sono più pedine da sistemare (la porzione $A[y, z-1]$ è diventata vuota). Per assicurare che l'invariante valga all'inizio della prima fase, le variabili sono impostate nel seguente modo: $x = 0$, $y = 1$, e $z = n + 1$. Lo pseudocodice dell'algoritmo è riportato di seguito. Come già argomentato è facile verificare che la complessità temporale dell'algoritmo è $O(n)$.

Algorithm 3: `Bandiera`(A)

```
n = lunghezza di A ;
x = 0; y = 1; z = n + 1;
while y < z do
    if A[y]=bianca then
        y = y + 1
    else
        if A[y]=rossa then
            scambia A[y] e A[z − 1];
            z = z − 1;
        else
            scambia A[y] e A[x + 1];
            x = x + 1;
            y = y + 1;
```