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 interi distinti. Ogni $A[i]$ può essere positivo, negativo o zero. Progettare un algoritmo con complessità temporale $o(n)$ che trova, se esiste, un indice i tale che $A[i] = i$.

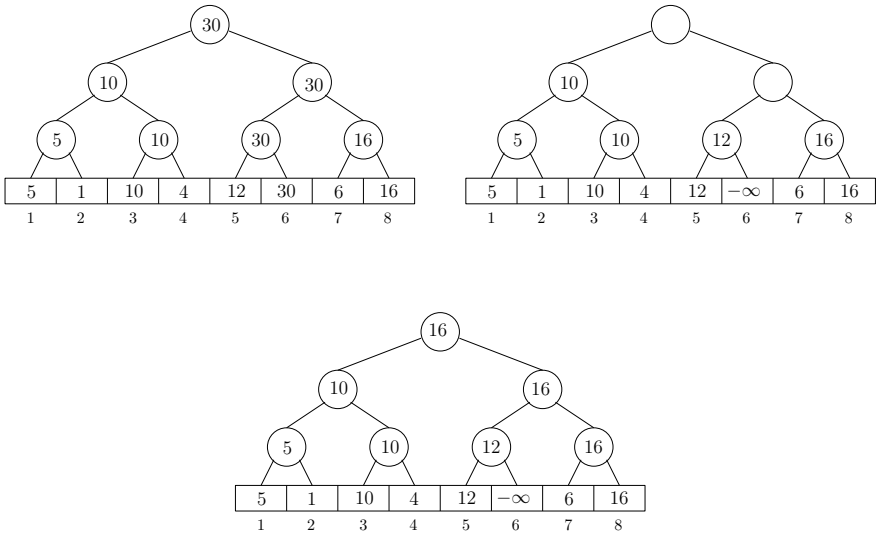

Figura 1: Esempio di istanza e di esecuzione dell'algoritmo che trova i due elementi più grandi di A usando al più $n+\log_2 n-2$ confronti. L'istanza di esempio ha $n = 8$ elementi. La figura in alto a sinistra mostra la situazione dopo l'esecuzione della prima fase dell'algoritmo (dove viene scoperto che 30 è l'elemento più grande), mentre le altre due figure mostrano l'esecuzione della seconda fase (dove viene scoperto che il secondo elemento più grande è 16.

Soluzione Esercizio 2 Descriviamo un algoritmo di complessità temporale $O(\log n)$ che trova, se esiste, un indice i tale che $A[i] = i$, o restituisce -1 altrimenti. L'idea è quella di usare l'approccio della ricerca binaria. Infatti, guardando un elemento, per esempio quello in posizione i possiamo capire se i è l'indice che stiamo cercando (verificando la condizione $A[i] = i$) o se l'indice che stiamo cercando va cercato a destra o sinistra di i. Per esempio quando $A[i] > i$, poiché gli elementi sono tutti distinti e ordinati, avremo che per ogni indice $j > i$ sarà sempre $A[j] > j$, e quindi possiamo restringere la ricerca ai soli indici più piccoli di i. Il caso $A[i] < i$ è analogo: tutti gli indici a sinistra di i possono essere scartati e si può proseguire la ricerca solo fra gli elementi a destra di i. Lo pseudocodice riportato di seguito si spiega da solo. La complessità temporale dell'algoritmo è uguale a quella della ricerca binaria e quindi è $O(\log n)$. Infatti, la complessità dell'algoritmo può essere espressa con l'equazione di ricorrenza $T(n) = T(n/2) + O(1)$, che risolta con il teorema Master ha soluzione $T(n) = O(\log n)$.

Algorithm 1: `CercaIndice`(A)

$n =$ lunghezza di A ;
return `CercaIndiceRic`$(A, 1, n)$

Dove, la procedura ricorsiva ausiliaria è:

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

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

Esercizio 3
Sia $A[1 : n]$ un vettore di n numeri distinti, dove n è una potenza di 2. Progettare un algoritmo che individua i due elementi più grandi di A effettuando al più $n + \log_2 n - 2$ confronti.[1]

E se si volessero individuare i k elementi più grandi in al più $n+k\log_2 n-k-1$ confronti?

Soluzione Esercizio 3 Descriviamo un algoritmo che lavora in due fasi. Nella prima fase, si costruisce un albero binario completo T che ha gli elementi di A come foglie. L'albero T va interpretato come un albero di torneo a eliminazione: ogni nodo interno dell'albero rappresenta una partita in cui si confrontano gli elementi contenuti nei due figli e la partita è vinta dall'elemento più grande dei due (che viene memorizzato nel nodo). Si può calcolare il valore di ogni nodo interno semplicemente visitando l'albero in postordine (bottom-up) e mettendo nel generico nodo non foglia visitato il massimo degli elementi contenuti nei figli. Ogni nodo interno, quindi, per essere riempito necessita di un confronto. Si noti che fino a questo momento il numero di confronti effettuato è uguale al numero di nodi interni dell'albero, ovvero $n-1$. Quando tutti i nodi di T sono riempiti, nella radice finisce chiaramente l'elemento di valore massimo di A, diciamo m. Qui termina la prima fase dell'algoritmo.

Per quanto riguarda la seconda fase, si noti che tutti i nodi di T di valore m individuano un cammino dalla radice all'elemento di A che contiene m. Questo cammino ha esattamente $\log_2 n$ nodi. In questa fase, l'algoritmo prende l'elemento massimo di A, lo sostituisce con un elemento di valore $-\infty$, cancella tutti i valori dei nodi del cammino, mette nel nodo interno padre di $-\infty$ il valore dell'elemento fratello di $-\infty$, e poi riempie (nuovamente) gli altri $\log_2 n - 1$ nodi del cammino dal basso vero l'alto facendo un confronto per nodo fra i due elementi figli. In questo modo, sulla radice finisce il secondo valore più grande di A. Il numero di confronti effettuati in questa seconda fase è quindi $\log_2 n - 1$. Un esempio di istanza e di esecuzione dell'algoritmo è mostrata in Figura 1.

L'algoritmo è facilmente generalizzabile al problema in cui si vogliono trovare i k elementi più grandi di A: basta eseguire la seconda fase k volte invece di una. In questo caso il numero totale di confronti dell'algoritmo è al più $n - 1 + k(\log_2 n - 1) = n + k\log_2 n - k - 1$, come richiesto.

Esercizio 4 *(il dollar game su grafi a cammino)*
Il gioco si gioca su un grafo a cammino di n nodi $1, 2, \ldots, n$. Inizialmente ad ogni nodo i

[1] Si noti che si sta vincolando (superiormente) solo il numero di confronti effettuati dall'algoritmo (e quindi tutte le altre operazioni sono gratis) e che tale delimitazione è esatta e non asintotica.

è associato un intero s_i che rappresenta il numero di dollari che quel nodo possiede; s_i può essere positivo, zero o negativo. Quando s_i è negativo si dice che i è in *debito*. L'unica mossa ammissibile è `push`(i), che trasferisce dal nodo i un dollaro su ogni vicino, quindi un dollaro sul nodo $i+1$ (se esiste) e uno sul nodo $i-1$ (se esiste); il numero di dollari sul nodo i quindi diminuisce di 2, se i non è un estremo del cammino, di 1 altrimenti. La mossa `push`(i) è sempre ammissibile, anche se porta il nodo i in debito o se i è già in debito prima della mossa. L'obiettivo del gioco è trovare una sequenza di mosse che porta ad una configurazione in cui non ci sono nodi in debito. Un esempio di istanza e soluzione è data in Figura 2.

Data un'istanza del gioco, sia $B = \sum_{1=1}^{n} s_i$ il *budget* complessivo dell'istanza.

(a) Si dimostri che l'istanza è risolvibile se e solo se $B \geq 0$.

(b) Si progetti un algoritmo veloce che data un'istanza con $B \geq 0$ trova una sequenza di mosse che la risolve. Si argomenti sulla correttezza dell'algoritmo e sulla sua complessità temporale.

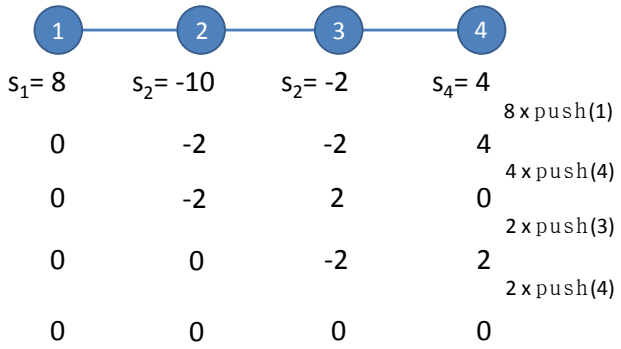

Figura 2: Un'istanza di *dollar game* su un cammino di 4 nodi e una possibile soluzione.

Soluzione Esercizio 4 Discutiamo prima il punto (a).

Sai B il budget iniziale dell'istanza. Vogliamo dimostrare che l'istanza è risolvibile se e solo se $B \geq 0$. Una delle due implicazioni è semplice: infatti, se $B < 0$, è facile vedere che l'istanza non può essere risolta. Questo perché ogni operazione di `push` lascia invariato il budget totale dell'istanza, quindi, dopo ogni sequenza di operazioni di `push` il budget totale è sempre lo stesso. Da cui segue che se si parte da un'istanza con $B < 0$ non è possibile raggiungere una configurazione in cui non ci sono nodi in debito in quanto tale configurazione avrebbe budget non negativo.

Può complicato è invece argomentare sul fatto che ogni istanza con $B \geq 0$ ammette una soluzione. Per fare questo mostriamo come è possibile realizzare, attraverso un'opportuna sequenza di mosse di `push`, due operazioni composite, che definiamo come:

- `MuoviAvanti`(i): è un'operazione che sposta un singolo dollaro dal nodo i al nodo $i+1$. Non è possibile eseguire la mossa per $i = n$.

- `MuoviIndietro`(i): è un'operazione che sposta un singolo dollaro dal nodo i al nodo $i-1$. Non è possibile eseguire la mossa per $i = 1$.

E' facile convincersi che, avendo a disposizione le operazione di `MuoviAvanti`(i) e `MuoviIndietro`(i), ogni istanza con $B \geq 0$ può essere risolta spostando opportunamente dollari da nodi che hanno budget positivo verso nodi con budget negativo. In particolare, forniremo poi, discutendo il punto (b), un algoritmo che trova velocemente una sequenza opportuna di mosse per risolvere l'istanza; in questa fase è sufficiente convincersi che, avendo a disposizione i due nuovi tipi di mosse, questo è possibile. Infatti, se $B \geq 0$, ogni volta che c'è un nodo i in debito deve esserci necessariamente un altro nodo j che ha budget strettamente maggiore di zero. E' possibile a questo punto spostare un dollaro da j a i usando una sequenza di mosse di tipo `MuoviAvanti` (se $j < i$) o una sequenza di mosse di tipo `MuoviIndietro` se $j > i$.

Resta quindi da mostrare come realizzare le due operazioni composite usando solo operazione di `push`. Questa non è una cosa difficile da fare. Infatti è possibile realizzare l'operazione `MuoviAvanti`(i) eseguendo una `push`(j) per ogni $j = 1, 2, \ldots, i$. Se infatti ci si concentra su un arco $(j, j+1)$, con $j < i$, è facile vedere che su tale arco viaggia un dollaro in avanti, ovvero da j a $j+1$, quando si esegue l'operazione `push`(j) e un dollaro indietro, ovvero da $j+1$ a j, quando si esegue l'operazione di `push`($j+1$). Da cui segue che l'intera sequenza ha come effetto quello di spostare un solo dollaro da i a $i+1$. Argomentazioni analoghe mostrano che è possibile realizzare l'operazione di `MuoviIndietro`(i) eseguendo una `push`(j) per ogni $j = i, i+1, \ldots, n$.

Ora discutiamo il punto (b). Assumiamo da adesso in poi che $B \geq 0$. Progetteremo un algoritmo che calcola una soluzione in tempo $O(n)$.

Prima di descrivere l'algoritmo è utile fare un paio di osservazioni.

- La complessità dell'algoritmo è criticamente legata alla rappresentazione dell'output. Come esempio esplicativo si pensi ad un'istanza di n nodi in cui il primo nodo ha un numero di dollari pari a $s_1 = 2^n$, il secondo ha un debito pari a $s_2 = -2^n$ e tutti gli altri hanno 0 dollari iniziali. Ogni soluzione di questa istanza deve effettuare un numero di `push` sul nodo 1 pari almeno a 2^n. Se si richiedesse di dare in output la sequenza delle singole operazioni di `push` l'output sarebbe quindi almeno lungo 2^n e quindi anche la complessità temporale dell'algoritmo sarebbe $\Omega(2^n)$.

- L'ordine con cui si eseguono le mosse di `push` non conta. In particolare, non è difficile convincersi che, poiché non c'è nessun vincolo su quando un nodo può eseguire una `push`, una sequenza di `push` e una sua qualsiasi permutazione producono esattamente lo stesso effetto.

Come conseguenza delle due osservazioni di sopra, decidiamo di rappresentare una soluzione del problema attraverso un vettore $P[1:n]$ di dimensione n, dove $P[i]$ conterrà il numero di operazioni di `push`(i). Nello pseudocodice in input viene passato un vettore $S[1:n]$ in cui $S[i]$ contiene il numero di dollari iniziali del nodo i, ovvero s_i.

Lo pseudocodice dell'algoritmo è dato in figura. L'algoritmo ha due fasi. La prima fase si occupa di trovare un insieme di mosse che modifica l'istanza originale in modo che tutti i nodi tranne il primo abbiano un numero di dollari pari a zero o negativo. Discuteremo questa prima fase dopo. Per il momento concentriamoci sulla seconda fase che trova una soluzione assumendo che tutto il budget in eccesso è contenuto sul primo nodo, mentre tutti

gli altri nodi sono in debito o hanno valore zero. L'idea è quella di procedere, da sinistra a destra, e spostare un'opportuna quantità di dollari in modo da lasciarsi alla spalle (a sinistra) tutti nodi con valore zero. Questo viene fatto con il primo ciclo `for` che calcola per ogni nodo i il numero $A[i]$ di operazioni `MuoviAvanti`(i) che il nodo i deve effettuare. Il secondo ciclo `for` della Fase 2, invece, traduce le operazioni `MuoviAvanti` in operazioni di tipo `push`, che vengono memorizzate nel vettore P_2. Si noti a tal proposito che, come discusso nel punto (a), un'operazione di `MuoviAvanti`(j) può essere realizzata effettuando un'operazione di `push` per ogni nodo $i \leq j$. Quindi ogni nodo i dovrà fare una `push` per ogni operazione `MuoviAvanti`(j) con $j \geq i$. Da cui segue che il numero di `push` che deve effettuare un nodo i è uguale a $P_2[i] = \sum_{j=i}^{n} A[j]$. Questi valori sono calcolati in tempo lineare nel secondo ciclo `for`.

Torniamo ora alla Fase 1. Per realizzarla possiamo usare la stessa idea discussa nella Fase 2: per rendere tutti i nodi tranne il primo con valore non positivo, si considerano i nodi da destra a sinistra e, ogni volta che un nodo ha un valore strettamente positivo si spostano (verso sinistra) con opportune operazioni di `MuoviIndietro` (memorizzate nel vettore I) i dollari in eccesso, lasciandosi così sulla destra solo nodi con valore al più zero. Questo viene effettuato dal primo ciclo `for`, mentre il secondo si occupa di tradurre le operazioni di `MuoviIndietro` in operazione di `push` (memorizzate nel vettore P_1). Infine, l'algoritmo mette insieme le operazioni di `push` della prima e della seconda fase e costruisce il vettore soluzione P. L'algoritmo ha chiaramente complessità computazione $O(n)$.

Algorithm 3: `DollarGame`(S)

```
// Inizializzazione
for i = 1 to n do
    A[i] = 0; I[i] = 0; P_1[i] = 0; P_2[i] = 0; P[i] = 0;
;
// Fase 1: sposta dollari in eccesso tutti sul primo nodo e lascia
   gli altri a valore al più 0
for i = n down to 2 do
    if S[i] > 0 then
        I[i] = S[i];
        S[i] = 0; ;
        S[i − 1] = S[i − 1] + I[i];
    ;
P_1[2] = I[2];
for i = 3 to n do
    P_1[i] = I[i] + P_1[i − 1];
;
// Fase 2: sposta i dollari posizionati sul primo nodo verso destra
   lasciandosi a sinistra tutti nodi con valore zero
for i = 1 to n − 1 do
    A[i] = S[i];
    S[i] = 0; ;
    S[i + 1] = S[i + 1] + A[i];
;
P_2[n − 1] = A[n − 1];
for i = n − 2 down to 1 do
    P_2[i] = A[i] + P_2[i + 1];
// Unione delle operazione delle due fasi
for i = 1 to n do
    P[i] = P_1[i] + P_2[i]
return P
```