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 (*equazioni di ricorrenza*)
Si risolvano le seguenti equazioni di ricorrenza. Si assuma sempre $T(1) = 1$.

(*a*) $T(n) = T(n-10) + 10$.

(*b*) $T(n) = T(n/2) + 2^n$.

(*c*) $T(n) = T(n/3) + T(n/6) + n^{\sqrt{\log n}}$.

(*d*) $T(n) = T(\sqrt{n}) + \Theta(\log\log n)$.

(*e*) $T(n) = T(n/2 + \sqrt{n}) + \Theta(1)$.

(*f*) $T(n) = \sqrt{n}T(\sqrt{n}) + n$.

Soluzione Esercizio 2

(*a*) Per iterazione.

$$T(n) = T(n-10) + 10 = T(n-20) + 20 = T(n-30) + 30 = \cdots = T(n-10i) + 10i$$

Quando $i = (n-1)/10$, abbiamo $T(n) = T(1) + n - 1 = n = \Theta(n)$.

(*b*) Usiamo il teorema Master. Dobbiamo confrontare $n^{\log_2 1}$ con 2^n. Quest'ultima funzione è chiaramente polinomialmente più veloce della prima. Siamo nel caso 3 (verificare per esercizio tutte le condizioni) e quindi $T(n) = \Theta(2^n)$.

(*c*) Dimostriamo che $T(n) = \Theta(n^{\sqrt{\log n}})$. Possiamo stimare $T(n)$ per eccesso nel seguente modo. $T(n) \leq S(n) = 2S(n/3) + n^{\sqrt{\log n}}$. Usando il teorema Master su $S(n)$ abbiamo (caso 3) $S(n) = \Theta(n^{\sqrt{\log n}})$. Da cui segue che $T(n) = O(n^{\sqrt{\log n}})$. La relazione $T(n) = \Omega(n^{\sqrt{\log n}})$ è ovvia.

(*d*) Facciamo un cambio di variabile, $m = \log_2 n$, da cui abbiamo che $T(n) = S(m) = S(m/2) + \log m$. Possiamo risolvere $S(m)$ con il metodo dell'albero della ricorsione: abbiamo $O(\log m)$ livelli, ognuno dei quali costa al più $\log m$. Quindi $S(m) = O(\log^2 m)$, da cui segue $T(n) = O((\log\log n)^2)$.

(*e*) Visto che $T(n)$ è monotonicamente crescente, per n abbastanza grande possiamo minorare e maggiorare il termine $T(n/2 + \sqrt{n})$ nel seguente modo:

$$T(n/2) \leq T(n/2 + \sqrt{n}) \leq T(n/2 + n/4) = T(3n/4).$$

Quindi un lower bound a $T(n)$ è $S(n) = S(n/2) + \Theta(1)$, mentre un upper bound è $S'(n) = S'(3n/4) + \Theta(1)$. Entrambe le precedenti equazioni di ricorreza hanno soluzione $\Theta(\log n)$ (si può usare il Teorema Master), da cui segue che $T(n) = \Theta(\log n)$.

(*f*) Definiamo $S(n) := \frac{T(n)}{n}$. Abbiamo quindi: $n\,S(n) = \sqrt{n}\sqrt{n}S(\sqrt{n}) + n$, ovvero $S(n) = S(\sqrt{n}) + 1$, che si può risolvere facendo il cambio di variabile $n = 2^m$ e ottenendo $S(n) = \Theta(\log\log n)$. Quindi $T(n) = \Theta(n\log\log n)$.

Esercizio 3 (*ricerca di un picco in una dimensione*)
Sia $V[1:n]$ un vettore di n numeri non negativi. Un *picco* in V è una posizione $p \in \{1,\ldots,n\}$ il cui elemento ha a sinistra e a destra elementi non più grandi di lui, ovvero $V[p] \geq \max\{V[p-1], V[p+1]\}$. Quando p è sul "bordo", la condizione è da considerarsi relativa solo all'unico elemento esistente adiacente a p. Pertanto, per semplicità e in modo equivalente considereremo $V[i] = -\infty$, quando $i < 1$ o $i > n$. Progettare un algoritmo che, dato V, trovi un picco in tempo $o(n)$.

Soluzione Esercizio 3 Per prima cosa si noti che nel vettore ci deve essere *almeno* un picco. Per definizione, infatti, l'indice dell'elemento massimo in V è chiaramente un picco. Questo suggerisce la seguente idea: trovare il massimo di V. Questo algoritmo, benché corretto, ha complessità $\Theta(n)$. Possiamo fare molto meglio utilizzando l'idea che è alla base della ricerca binaria.

L'algoritmo ricorsivo che progettiamo usa la tecnica del *divide et impera*. In un generico passo dell'algoritmo si sta cercando un picco nella porzione del vettore individuata da due indici i e j (all'inizio $i = 1$ e $j = n$) in cui si è sicuri che ci sia almeno un picco. Si guarda quindi l'elemento in posizione centrale rispetto a i e j, quello in posizione $m = \lfloor \frac{i+j}{2} \rfloor$. Se m è un picco si ritorna m, altrimenti ci deve essere un elemento adiacente a $V[m]$ che è strettamente più grande. Se tale elemento è $A[m-1]$ si ricorre nella porzione $[i, m-1]$, altrimenti si ricorre in $[m+1, j]$. E' facile vedere che nella porzione in cui si chiama ricorsivamente l'algoritmo deve esserci almeno un picco, per esempio l'elemento più grande contenuto nella porzione di array in cui si ricorre (si noti che è qui che stiamo usando la proprietà che l'elemento adiacente a $A[m]$ è più grande di $A[m]$ per come abbiamo scelto la porzione su cui fare ricorsione). Lo pseudocodice dettagliato dell'algoritmo è lasciato per esercizio allo studente. La complessità temporale dell'algoritmo è descritta dalla relazione di ricorrenza che descrive la complessità della ricerca binaria, ovvero: $T(n) = T(n/2) + O(1)$, che ha soluzione $T(n) = \Theta(\log n)$.

Esercizio 4 (*ricerca di un picco in due dimensioni*)
Sia $M[1:n, 1:m]$ una matrice con n righe, m colonne, tale che $M[i,j]$ è un numero non negativo. Un *picco* in M è una posizione (p,q), $p \in \{1,\ldots,n\}$ e $q \in \{1,\ldots,m\}$ il cui elemento ha a sinistra, a destra, sopra e sotto elementi non più grandi di lui, ovvero $M[p,q] \geq \max\{M[p-1,q], M[p+1,q], M[p,q-1], M[p,q+1]\}$. Quando (p,q) è sul "bordo", la condizione è da considerarsi relativa solo agli elementi esistenti adiacenti a (p,q). Pertanto, per semplicità e in modo equivalente considereremo $M[i,j] = -\infty$, quando uno dei due indici i o j è minore di 1 o maggiore di n.

- Si consideri il seguente algoritmo che essenzialmente esegue una ricerca di un picco spostandosi sempre su elementi più grandi. Più in dettaglio, l'algoritmo parte da una data posizione (i,j). Se tale elemento non è un picco si controlla gli (al più 4) elementi adiacenti a (i,j), si sposta sull'elemento di valore massimo, e procede ricorsivamente. Si dimostri se tale algoritmo è corretto e se ne studi la complessità asintotica nel caso peggiore.

- Si progetti un algoritmo che trovi un picco in M con complessità temporale $o(nm)$ e che usi la tecnica *divide et impera*.

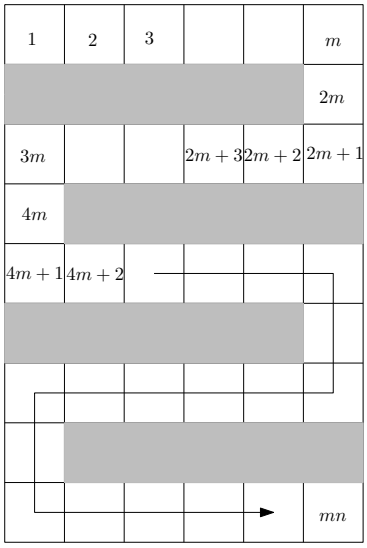

Figura 1: Istanza per la quale l'algoritmo greedy impiega tempo $\Theta(nm)$. La matrice ha n righe e m colonne. Le zone grigie contengono zeri. Se l'algoritmo parte dalla posizione $(1, 1)$, segue il cammino a "serpente" e trova l'unico picco in posizione (n, m) solo alla fine, quando ha visto circa metà degli elementi della matrice, e quindi $\Theta(nm)$ elementi.

Soluzione Esercizio 4 Anche nel caso di due dimensioni, è facile vedere che data una matrice c'è sempre un picco. La posizione dell'elemento massimo della matrice, per esempio, è chiaramente un picco. Anche in questo caso, trovare l'elemento di valore massimo richiederebbe tempo $\Theta(nm)$, e noi vogliamo fare meglio.

Analizziamo prima l'algoritmo suggerito nel primo punto dell'esercizio, che chiameremo *algoritmo greedy*. Vogliamo rispondere alle seguenti domande: è corretto? Quale è la sua complessità temporale?

Argomentiamo prima sulla correttezza dell'algoritmo. L'intuizione è che l'algoritmo ad ogni passo guarda un certo elemento in posizione (i, j), se questo non è un picco, ci si sposta su un elemento adiacente che è strettamente più grande di $M[i, j]$. Questo garantisce che l'elemento su cui ci si sposta deve essere diverso da tutti quelli precedentemente visti, in quanto è l'elemento più grande visto fino a quel momento (mentre l'elemento da cui si parte è quello più piccolo, e il cammino che si sta seguendo individua una sequenza strettamente crescente). Quindi, dopo al più mn passi, l'algoritmo ha eventualmente visitato tutta la matrice e deve aver necessariamente trovato un picco (che sappiamo esiste sempre).

Tale ragionamento è utile anche per dare una delimitazione superiore alla complessità temporale dell'algoritmo. Prima di trovare un picco, l'algoritmo eseguirà al più nm passi, ognuno dei quali richiede tempo costante. Per cui il suo tempo di esecuzione è $O(mn)$. Domandiamoci ora: è questa un'analisi stretta (*tight*) o l'algoritmo in realtà ha un costo $o(nm)$? Purtroppo, nel caso peggiore l'algoritmo richiede comunque tempo $\Omega(mn)$ e quindi la coplessità è $\Theta(mn)$. Un esempio è mostrato in Figura 1.

Ora progettiamo un algoritmo che usa la tecnica del *divide et impera* con complessità $o(nm)$. L'idea è la seguente: si guarda la colonna centrale della matrice, sia essa la colonna di indice $q = \lfloor \frac{m+1}{2} \rfloor$, e si cerca in tempo $O(n)$ il massimo di tale colonna, sia p la riga di tale massimo (e quindi $M[p, q]$ è il massimo della colonna q). Se (p, q) è un picco lo ritorniamo,

altrimenti almeno uno degli elementi a sinistra o a destra di (p, q) è strettamente più grande di $M[p, q]$. Se tale elemento è $M[p, q-1]$, ricorriamo nella sottomatrice composta dalle prima $q-1$ colonne, altrimenti, ricorriamo nella matrice composta dalle colonne da $q+1$ a m.

Per quanto riguarda la correttezza dell'algoritmo, dobbiamo solo argomentare sul fatto che quando ricorriamo su una delle due metà della matrice, siamo sicuri che in quella metà ci sarà almeno un picco. Assumiamo senza perdita di generalità, che $M[p, q-1] > M[p, q]$, e quindi ricorriamo sulla metà a sinistra della colonna q (per l'altro caso, valgono argomentazioni simili). Per vedere che la metà di sinistra deve contenere almeno un picco, si immagini di lanciare l'algoritmo greedy (del punto precedente) sull'elemento $M[p, q-1]$. Sappiamo che l'algoritmo terminerà trovando un picco. Bene, vogliamo argomentare sul fatto che tale picco deve essere nella metà di sinistra. Infatti, se così non fosse, ovvero, se l'algoritmo trovasse un picco nella metà della matrice a destra di q, vuol dire che nel cammino strettamente crescente che segue, deve attraversare almeno una volta la colonna q. Sia (p', q) la posizione dell'elemento della colonna q che l'algoritmo greedy visita quando passa dalla metà di sinistra a quella di destra. Allora deve essere che $M[p', q] > M[p, q-1] > M[p, q]$. Ma $M[p, q]$ era il massimo della colonna q, quindi abbiamo un assurdo: il picco trovato dall'algoritmo greedy deve trovarsi nella metà di sinistra.

Per quanto riguarda la complessità dell'algoritmo, la relazione di ricorrenza che possiamo scrivere è la seguente (si noti che dipende dai due parametri n e m): $T(n, m) = T(n, m/2) + O(n)$. Si osservi ora che in tale equazione di ricorrenza, il termine n è costante rispetto a m nelle chiamate ricorsive. Quindi l'equazione di ricorrenza si può riscrivere come $T(m) = T(m/2) + O(n)$, che ha come soluzione $\Theta(n \log m)$ (si può risolvere con il metodo dell'albero della ricorsione). Del resto, stiamo facendo ricerca binaria sulle m colonne (e quindi guardiamo $O(\log m)$ colonne) e per ognuna di esse spendiamo tempo $O(n)$ per cercare il massimo.

Esercizio 5 (*embedding di un albero binario completo su una griglia bidimensionale*)
Sia dato un albero binario completo T con n foglie e una griglia bidimensionale (foglio a quadretti). Si vuole disegnare T sulla griglia in modo da non far intrecciare gli archi e minimizzando lo spazio utilizzato. Più precisamente, un *embedding* di T è un disegno di T sul foglio tale che: (i) i nodi di T sono disegnati sulle intersezioni della griglia (ovvero come cerchi centrati su un qualche spigolo di un qualche quadratino), (ii) gli archi sono delle linee "seghettate" che seguono le linee del foglio, (iii) le linee che rappresentano gli archi non possono incrociarsi reciprocamente. Dato un embedding di T il suo *bounding box* è il rettangolo più piccolo che contiene l'intero embedding. Trovare un embedding di T che minimizza (asintoticamente) l'area del bounding box.

Soluzione Esercizio 5 Prima di descrivere e discutere l'embedding di area (asintoticamente) minima, discutiamo brevemente una soluzione semplice ma meno efficiente. Tale soluzione consiste nel disegnare l'albero nel modo classico, rispettando i vincoli imposti. E' di fatto uno schema ricorsivo che usa la tecnica del divide et impera. Lo schema generale con un esempio per $n = 16$ è riportato in Figura 2. Chiediamoci ora quanta area occupa il bounding box di tale embedding. E' facile convincersi che l'altezza del bounding box è $\Theta(\log n)$, mentre la base è lunga $\Theta(n)$, il che porta ad un'area di $\Theta(n \log n)$. Si noti che le equazioni di ricorrenza che descrivono rispettivamente l'altezza e la base del bounding box secondo lo schema ricorsivo sono: $H(n) = H(n/2) + \Theta(1)$, e $B(n) = 2B(n/2) + \Theta(1)$.

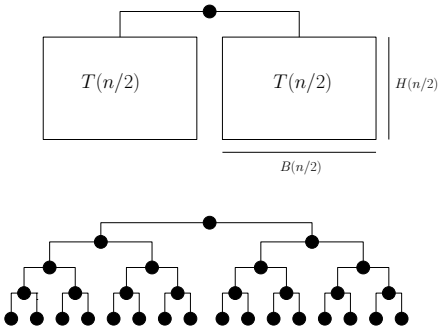

Figura 2: Embedding il cui bounding box ha area $\Theta(n \log n)$.

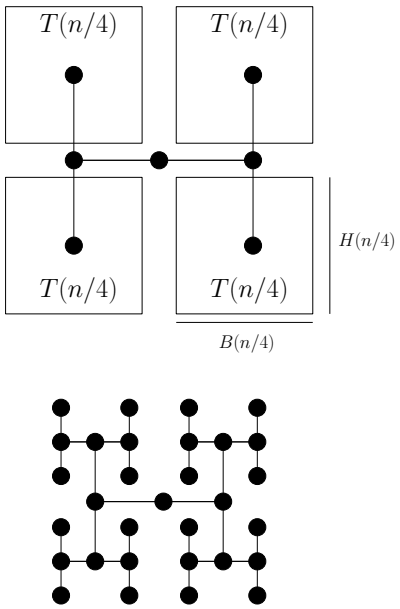

Figura 3: Embedding (asintoticamente) ottimo il cui bounding box ha area $\Theta(n)$.

Descriviamo ora un embedding che richiede area $\Theta(n)$. Si noti che, poiché dobbiamo comunque disegnare da qualche parte le n foglie, nessuna soluzione può richiedere asintoticamente area $o(n)$, per cui la soluzione che segue è asintoticamente ottima. Lo schema e un esempio per $n = 16$ è riportato in Figura 3.

Le equazioni di ricorrenza che descrivono rispettivamente l'altezza e la base del bounding box secondo lo schema ricorsivo sono: $H(n) = 2H(n/4) + \Theta(1)$, e $B(n) = 2B(n/4) + \Theta(1)$, che hanno entrambe soluzione $\Theta(\sqrt{n})$, da cui segue che l'area del bounding box è $\Theta(n)$.