Problem Set 2

docente: Luciano Gualà

Esercizio 1 (*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 1

(*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 2 (*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 2 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. 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 3 (*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*.

Soluzione Esercizio 3 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.

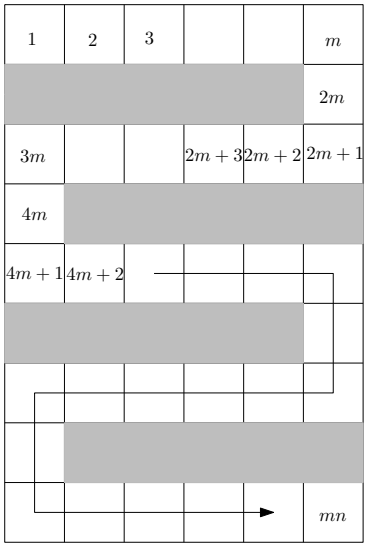


Figura 1: Istanza per la quale l'algoritmo greedy impiega tempo $\Theta(nm)$. La matrice ha n righe e m colonne. Le zone grigie contengono zeri. Se l'algoritmo parte dalla posizione $(1, 1)$, segue il cammino a "serpente" e trova l'unico picco in posizione (n, m) solo alla fine, quando ha visto circa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

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 4 (*sul codice Gray*)
Un *codice Gray* di lunghezza n è costituito da tutte le 2^n sequenze di n bit e consente di rappresentare tutti gli interi da 0 a $2^n - 1$. La caratteristica di un tale codice è che le codifiche di due valori consecutivi differiscono di esattamente un bit. Un esempio è mostrato in figura. E' possibile costruire ricorsivamente un codice gray a n bits partendo da uno ad $n-1$ bits nel seguente modo (si veda la figura):

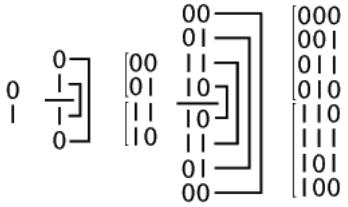
- Le prime 2^{n-1} parole del codice ad n bits sono uguali a quelle del codice ad $n-1$ bits estese con uno 0 come bit più significativo;
- Le seconde 2^{n-1} parole del codice ad n bits sono uguali a quelle del codice ad $n-1$ bits ma elencate in ordine inverso (riflesso) ed estese con un 1 come bit più significativo.

Si progetti un algoritmo ricorsivo che, dato n e un indice $i \in \{0, 1, \ldots, 2^n - 1\}$, calcoli la codifica di i in tempo $O(n)$.

Soluzione Esercizio 4 L'idea è quella di progettare un algoritmo lasciandoci guidare dalla definizione ricorsiva del codice Gray. L'algoritmo chiaramente sarà anche esso ricorsivo, ed è descritto di seguito. La chiamata a `Gray`(n, i) restituisce la codifica di i nel codice gray con n bit. Assume che $i \in \{0, 1, \ldots, 2^n - 1\}$. Si noti che il risultato di una tale chiamata è di fatto l'i-esima riga della tabella del codice con n bit. Questa tabella ha le seguenti caratteristiche: le prime 2^{n-1} righe (numerate da 0 a $2^{n-1} - 1$) cominciano per 0 e, se non si considera questo bit più significativo, coincidono con le 2^{n-1} righe della tabella del codice con $n-1$ bit. Invece, tutte le seconde 2^{n-1} righe cominciano per 1 e, se si esclude questo bit più significatico, coincidono con le 2^{n-1} righe della tabella del codice con $n-1$ bit che però sono elencate in ordine inverso. Sfruttando questa struttura è facile vedere che `Gray`(n, i)

i	Gray(*i*)
0	0000
1	0001
2	0011
3	0010
4	0110
5	0111
6	0101
7	0100
8	1100
9	1101
10	1111
11	1110
12	1010
13	1011
14	1001
15	1000

(a) esempio condice Gray con *n*=4 bit



(b) Generazione codice per riflessione

Figura 2:

coincide con la riga numero i della tabella del codice con $n-1$ bit, se $i < 2^{n-1}$, mentre altrimenti coincide con la riga numero $2^n - 1 - i$ del codice con $n-1$ bit.

A questo punto è facile progettare l'algoritmo. Di seguito è riportato lo pseudocodice, in cui si è usato il simbolo "$\cdot$" per indicare l'operatore che concatena due stringhe. La complessità dell'algoritmo è descritta dall'equazione di ricorrenza $T(n) = T(n-1) + O(1)$ che risolta ci dice che $T(n) = O(n)$.

Algorithm 1: `Gray`(n, i)

```
if n = 1 then
    if i = 0 then
        return 0
    else
        return 1
/* n > 1                                                    */
k = 2^{n-1} ;
if i < k then
    return 0· Gray(n − 1, i)
else
    return 1· Gray(n − 1, 2k − i − 1)
```