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 (*test di infrangibilità di bicchieri*)
Dovete valutare l'infrangibilità di un certo tipo di bicchiere di vetro e organizzate un test per determinare l'altezza massima da cui potete far cadere il bicchiere senza che esso si rompa. Per il test, avete a disposizione una scala con n pioli e voi dovete capire quale è il più alto piolo da cui è possibile far cadere il bicchiere senza conseguenze. Chiameremo questo piolo il *più alto piolo sicuro*.

Ora, essendo dei bravi algoritmisti, se aveste a disposizione tante copie dello stesso tipo di bicchiere, al fine di fare pochi lanci, simulereste una ricerca binaria. Quindi lascereste cadere un bicchiere dal piolo di altezza $n/2$ e, in base all'esito, passereste a provare il piolo ad altezza $n/4$ o $3n/4$, e così via. Questo vi consentirebbe di trovare il più alto piolo sicuro facendo $O(\log n)$ lanci ma potreste rompere molti bicchieri.

Se invece il vostro obiettivo primario fosse quello di conservare quanti più bicchieri potete, allora la strategia migliore sarebbe quella di provare in sequenza il piolo 1, poi il piolo 2, e così via, fino a trovare il più alto piolo sicuro. Questo sacrificherebbe un solo bicchiere ma vi costerebbe in termini di tempo, perché nel caso peggiore potreste dover eseguire un numero lineare di lanci.

In questo esercizio vi si chiede di trovare una soluzione che bilancia il numero di lanci con il numero di bicchieri che potete rompere. Per essere più precisi, considerate il caso in cui avete a disposizione un numero massimo $k \geq 1$ di bicchieri che potete rompere, e voi volete capire quale è il più alto piolo sicuro effettuando meno lanci possibile. In particolare:

(a) Supponete di avere a disposizione $k = 2$ bicchieri. Trovate una strategia che risolva il problema eseguendo al più $f(n)$ lanci, per una qualche funzione $f(n) = o(n)$.

(b) Ora assumete di avere a disposizione $k > 2$ bicchieri. Descrivete una strategia per trovare velocemente il più alto piolo sicuro utilizzando al più k bicchieri. Se $f_k(n)$ denota il numero di lanci che occorrono alla vostra strategia per un certo k, allora le funzioni $f_1(n), f_2(n), \ldots$ dovrebbero avere la caratteristica di possedere un tasso di crescita via via più basso al crescere di k, ovvero, dovrebbe valere che $f_k(n) = o(f_{k-1}(n))$, per ogni k.

Soluzione Esercizio 2 Consideriamo prima il punto (a) e assumiamo per semplicità che n sia un quadrato perfetto, ovvero che $n = \ell^2$ per un qualche intero ℓ. L'algoritmo procede provando i pioli che sono multipli di $\sqrt{n}$, ovvero $\sqrt{n}$, $2\sqrt{n}$, $3\sqrt{n}$... finché il (primo) bicchiere non si rompe. Se questo non accade mai allora vuol dire chiaramente che il più alto piolo sicuro è il piolo n. Altrimenti, sia j il piolo che causa la rottura del bicchiere. Sappiamo con certezza che il più alto piolo sicuro deve trovarsi fra il piolo $(j-1)\sqrt{n}$ e il piolo $j\sqrt{n}$. A questo punto usiamo il secondo bicchiere provando tali pioli in ordine, ovvero proviamo i pioli $(j-1)\sqrt{n}+1, (j-1)\sqrt{n}+2, \ldots, j\sqrt{n}$.

Chiaramente l'algoritmo trova il più alto piolo sicuro. Il numero totale di lanci è $2\sqrt{n}$ perché sia il primo che il secondo bicchiere vengono lanciati al più $\sqrt{n}$ volte. Se n non è un quadrato perfetto, possiamo provare i multipli di $\lceil\sqrt{n}\rceil$ (con l'accortezza che se un multiplo supera n allora proviamo il piolo n). I lanci effettuati complessivamente sono chiaramente $2\lceil\sqrt{n}\rceil = O(\sqrt{n})$.

Ora generalizziamo l'algoritmo dato per il punto (a) per gestire il caso in cui abbiamo a disposizione $k > 2$ bicchieri. Inizialmente, assumiamo per semplicità che n è una potenza k-esima perfetta (ovvero che $n = \ell^k$, per un qualche intero ℓ). Costruiamo un albero completo di altezza k e grado $\sqrt[k]{n}$ e associamo alle foglie di tale albero (da sinistra a destra) gli n pioli,

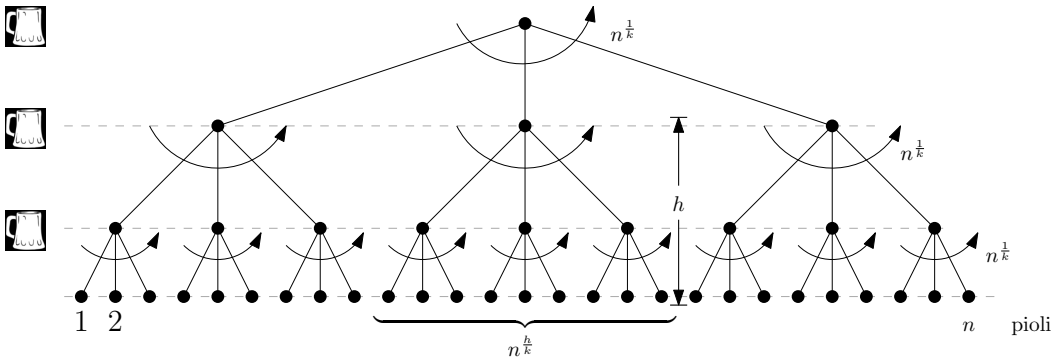

Figura 1: Il caso $k > 2$: l'albero completo di grado $\sqrt[k]{n}$ associato all'istanza.

da 1 a n. Un esempio per $n = 27$ e $k = 3$ è mostrato in Figura 1. Useremo un bicchiere per ogni livello, dal livello 0 al livello $k - 1$. Associamo, inoltre, ad ogni nodo v il piolo di indice maggiore nel sottoalbero con radice v (ovvero, la foglia più a destra del sottoalbero radicato in v).

L'algoritmo procede nel seguente modo. All'inizio ha a disposizione k bicchieri e considera l'albero di altezza k. Parte dalla radice dell'albero e guarda i figli della radice in ordine (da sinistra a destra). Per ogni figlio v, lancia il bicchiere dal piolo associato al nodo v, finché non trova un piolo dal quale il bicchiere si rompe. Se questo non accade mai, vuol dire che n è il più alto piolo sicuro e l'algoritmo termina. Altrimenti, sia v il nodo che fa rompere il bicchiere. Sappiamo che il più alto piolo sicuro deve trovarsi fra le foglie dell'albero radicato in v. Tale albero ha altezza $k-1$ e l'algoritmo ha ancora a disposizione $k - 1$ bicchieri. Procede ricorsivamente considerando v come radice.

E' facile convincersi che l'algoritmo trova correttamente il più alto piolo sicuro. Per quanto riguarda il numero di lanci effettuati, si noti che per ogni livello si eseguono al più $\sqrt[k]{n}$ lanci, e quindi il numero totale di tentativi è $k\sqrt[k]{n}$.

Il caso invece in cui n non è una potenza k-esima esatta può essere gestito costruendo un albero binario completo di grado $\lceil \sqrt[k]{n} \rceil$. In questo caso le foglie dell'albero sono più di n ma questo non è certamente un problema. Si può per esempio rimuovere le foglie dell'albero che non corrispondono a nessun piolo e usare lo stesso algoritmo. Ogni nodo dell'albero ha grado al più $\lceil \sqrt[k]{n} \rceil$ e quindi il numero totale di lanci è al più $k\lceil \sqrt[k]{n} \rceil \leq k\sqrt[k]{n}+k = O(k\sqrt[k]{n})$.

Esercizio 3 (*calcolo della maggioranza*)
Sia dato un vettore $V[1; n]$ di n elementi. Un elemento è di *maggioranza* se il numero di occorrenze dell'elemento è strettamente più di $n/2$. Si assuma che nel vettore c'è un elemento di maggioranza (che si vuole scoprire) ma gli elementi sono in qualche modo "nascosti", non possono essere letti direttamente. L'unica operazione possibile è fare una *richiesta* $q(i, j)$, che consiste nel chiedere se i due elementi in posizione i e j, rispettivamente, sono uguali o meno; ovvero, $q(i, j)$ restituisce `vero` se $V[i] = V[j]$, `falso` altrimenti. Le operazioni di richiesta sono costose e quindi se ne vogliono fare il meno possibile.

(a) Si progetti un algoritmo che usa la tecnica del *divide et impera* che trova l'elemento di maggioranza facendo $o(n^2)$ richieste.

(b) Si progetti un (miglior) algoritmo che fa un numero lineare di richieste e usa memoria costante.

Soluzione Esercizio 4 Consideriamo prima il punto (a). L'idea dell'algoritmo risiede nella seguente proprietà.

Proprietà 1. *Se V ha un elemento di maggioranza x, allora x è un elemento di maggioranza per $V[1 : \lfloor \frac{n}{2} \rfloor]$ o per $V[\lfloor \frac{n}{2} \rfloor + 1 : n]$.*

Dimostrazione. Supponiamo per assurdo che non sia così. Allora x compare al più $\frac{1}{2}\lfloor \frac{n}{2} \rfloor$ volte in $V[1 : \lfloor \frac{n}{2} \rfloor]$ e al più $\frac{1}{2}\lceil \frac{n}{2} \rceil$ in $V[\lfloor \frac{n}{2} \rfloor + 1 : n]$. Da cui segue che x compare al più $\frac{1}{2}\lfloor \frac{n}{2} \rfloor + \frac{1}{2}\lceil \frac{n}{2} \rceil = \frac{1}{2}(\lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil) = \frac{n}{2}$ volte in V. Quindi x non poteva essere un elemento di maggioranza per V. Assurdo. □

L'algoritmo che sfrutta la precedente proprietà è descritto ad alto livello di seguito (si fornisca per esercizio lo pseudocodice dettagliato):

- trova ricorsivamente un elemento di maggioranza x_1 in $V[1 : \lfloor \frac{n}{2} \rfloor]$ e un elemento di maggioranza x_2 in $V[\lfloor \frac{n}{2} \rfloor + 1 : n]$. Caso base della ricorsione: se il vettore ha un solo elemento, l'elemento di maggioranza è quell'elemento.
- Calcola il numero di volte n_1 e n_2 in cui x_1 e x_2 rispettivamente compaiono in V. n_1 e n_2 possono essere trovati facendo $2n$ richieste.
- Restituisci x_1 se $n_1 \geq n_2$, restituisci x_2 altrimenti.

La correttezza dell'algoritmo risiede essenzialmente nella proprietà dimostrata sopra. Per quanto riguarda la complessità, ovvero il numero di richieste fatte, può essere descritta dall'equazione di ricorrenza $T(n) = 2T(n/2) + \Theta(n)$, che ha come soluzione $T(n) = \Theta(n \log n)$.

Passiamo ora al punto (b). Di seguito riportiamo l'algoritmo. Le istruzioni fra parentesi quadre sono operazioni che non vengono fatte davvero, ma sono inserite solo per ragionare sulle proprietà dell'algoritmo. In altre parole, l'algoritmo che risolve il punto (b) è ottenuto dallo pseudocodice rimuovendo le istruzioni fra parentesi quadre.

Algorithm 1: `maggioranza`(V)

```
score=0; candidato=-1; [S: pila];
for i = 1 to n do
    if score=0 then
        candidato=i ;
        score=score+1 ;
        [S.push(V[i])];
    else
        if q(i, candidato) then
            score=score+1; [S.push(V[i]);]
        else
            score=score-1; [S.pop;]
    ;
return candidato
```

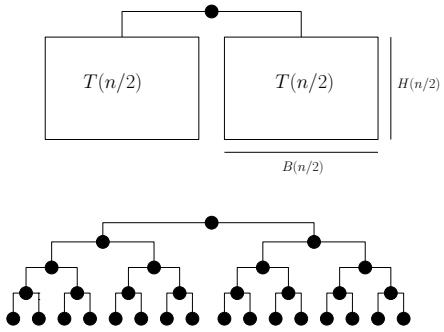

Figura 2: Embedding il cui bounding box ha area $\Theta(n \log n)$.

Per quanto riguarda la complessità, ovvero il numero di richieste fatte dall'algoritmo è chiaro che è $O(n)$. Si noti anche che, escludendo l'uso della pila, la memoria ausiliaria utilizzata è costante. Ma l'algoritmo è corretto? Argomentiamo ora su questo.

Innanzitutto, è facile convincersi che vale la seguente invariante: in ogni istante di tempo, se la pila non è vuota, gli elementi nella pila sono tutti uguali e sono tutti uguali all'elemento candidato (ovvero l'elemento del vettore in posizione `candidato`). Definiamo un elemento *bruciato* se viene confrontato con un altro elemento in una richiesta e la richiesta da esito negativo (gli elementi sono diversi). Allora, un elemento non di maggioranza può bruciare un solo elemento di maggioranza, e poiché il numero di elementi di maggioranza è strettamente più di $n/2$, c'è un elemento di maggioranza x che non è mai bruciato. Allora tale elemento alla fine dell'algoritmo si trova nella pila e, per l'invariante, è uguale all'elemento candidato. Quindi l'algoritmo restituisce l'indice di un elemento di maggioranza.

Esercizio 4 (*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 4 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 $B(n) = 2B(n/2) + \Theta(1)$.

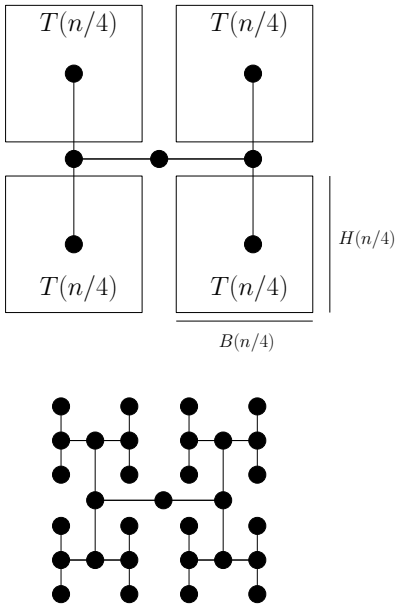

Figura 3: Embedding (asintoticamente) ottimo il cui bounding box ha area $\Theta(n)$.

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 $\Theta(n)$.