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 $T(n) = \Theta(n\log\log n)$.

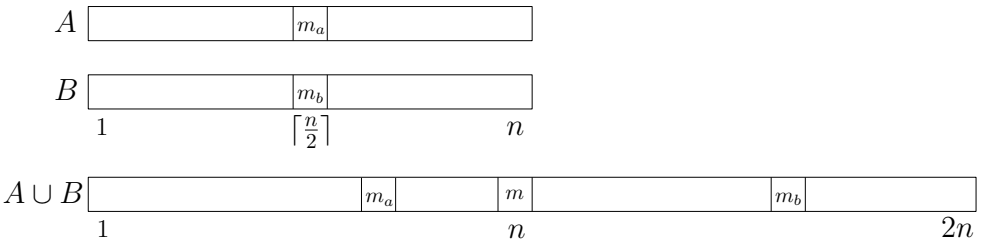

Figura 1: Proprietà cruciale utilizzata dall'algoritmo. I due database (e la loro unione) sono disegnati come vettori ordinati.

Esercizio 2
Siete interessati ad analizzare alcuni dati che sono difficili da ottenere e che sono distribuiti su due database differenti. Ogni database contiene n valori numerici – così i valori in totale sono $2n$ – e potete assumere che sono valori tutti distinti. Voi volete determinare il *mediano* di questi valori, ovvero l'n-esimo valore più piccolo. L'unico modo che avete per accedere ai database è attraverso delle *query*. In ogni singola query potete specificare uno dei due database e un valore k, e ottenere come risposta alla query il k-esimo valore più piccolo del database che state interrogando. Poiché le query sono costose, volete farne il meno possibile. Progettare un algoritmo che è in grado di trovare il mediano facendo nel caso peggiore $O(\log n)$ query.

Soluzione Esercizio 2 Siano A e B i due database, e siano $A(i)$ e $B(i)$ rispettivamente gli i-esimi elementi più piccoli di A e di B. Sia $k = \lceil \frac{n}{2} \rceil$, e siano $m_a = A(k)$ e $m_b = A(k)$ rispettivamente i mediani di A e di B. Assumiamo senza perdita di generalità che $m_a < m_b$ (altrimenti si possono semplicemente scambiare i ruoli di A e B). La proprietà cruciale che useremo per progettare un algoritmo efficiente è la seguente: $m_a \leq m \leq m_b$ dove m è il mediano di $A \cup B$ (si veda la Figura 1).

Per vedere che questa proprietà è vera, si consideri il numero di elementi di $A \cup B$ che sono minori o uguali di m_b. Questi sono almeno gli elementi di A che sono minori o uguali di m_a (che sono $\lceil \frac{n}{2} \rceil$) e gli elementi di B che sono minori o uguali di m_b (che sono ancora $\lceil \frac{n}{2} \rceil$). Quindi questi elementi sono almeno $2\lceil \frac{n}{2} \rceil \geq n$, ovvero m_b non può trovarsi in $A \cup B$ a sinistra di m. Si possono usare argomentazioni simili per mostrare che il numero di elementi in $A \cup B$ che sono maggiori o uguali di m_a sono almeno n, e quindi m_a non può trovarsi in $A \cup B$ a destra di m. Siano $A_<$ e $B_>$ rispettivamente gli elementi di A che sono strettamente più piccoli di m_a e gli elementi di B che sono strettamente più grandi di m_b. Una conseguenza importante della proprietà individuata è che il mediano m non può trovarsi né in $A_<$ né in $B_>$.

Sia $x = \min\{|A_<|, |B_>|\}$. L'idea ora è di buttare x elementi di A che sono più piccoli di m_a e x elementi di B che sono più piccoli di m_b. Siano A' e B' i due insiemi ottenuti da A e B dopo aver eliminato questi elementi. Chiaramente $|A'| = |B'|$. Inoltre, il mediano di $A' \cup B'$ è ancora m, ovvero il mediano di $A \cup B$, perché da $A \cup B$ ho eliminato x elementi a sinistra di m_a e x elementi a destra di m_b.

A questo punto usiamo queste argomentazioni ricorsivamente su A' e B'. Il caso base è quando i due insiemi che sto ancora considerando hanno una dimensione che è al più 2: in tal caso facciamo al più 4 query e troviamo il mediano degli elementi dell'unione. Lo pseudocodice dettagliato dell'algoritmo è lasciato come esercizio allo studente.

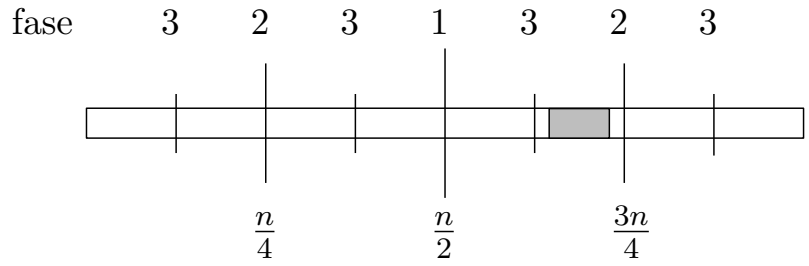

Figura 2: Schema dell'algoritmo. Gli elementi del vettore che vengono guardati (nelle prime tre fasi) sono indicati da barre verticali. In grigio la porzione contigua di uni (la nave nemica) da scovare.

Per quanto riguarda la complessità dell'algoritmo, ovvero il numero di query che l'algoritmo effettua nel caso peggiore, si noti che in ogni fase, effettuiamo un numero di query costante e poi ricorsivamente applichiamo l'algoritmo su un'istanza che ha circa la metà degli elementi. Quindi il numero complessivo di query è $O(\log n)$.

Esercizio 3 (*battaglia navale*)
Un array $A[1:n]$ di n elementi è riempito interamente di zeri, tranne che in una porzione di ℓ celle contigue che contengono degli uni. Progettare un algoritmo che in tempo $O(\frac{n}{\ell})$ trovi almeno un indice i tale che $A[i] = 1$. Si fornisca una soluzione prima assumendo di conoscere la lunghezza ℓ della porzione contigua di uni, poi si rimuova questa assunzione e si progetti un algoritmo che sempre in tempo $O(\frac{n}{\ell})$ trovi almeno la posizione di un 1 senza conoscere ℓ.

Soluzione Esercizio 3 Se conosciamo il valore di ℓ, ovvero sappiamo quanto è lungo il blocco di uni contigui che stiamo cercando (ovverso sappiamo quanto è grande la nave avversaria che stiamo cercando di colpire) l'algoritmo è semplice. Proviamo le posizioni dell'array $i\ell$ con $i = 1, 2, \ldots, \lfloor \frac{n}{\ell} \rfloor$. Chiaramente, in questo modo siamo sicuri di vedere (colpire) almeno una posizione del blocco contiguo di uni (la nave avversaria). Inoltre, il numero di posizioni che guardiamo è banalmente $O(\frac{n}{\ell})$.

Descriviamo ora un algoritmo per il caso più intrigante in cui il valore ℓ è ignoto all'algoritmo. L'idea è quella procedere per fasi e in ogni fase di provare posizioni del vettore in modo da spezzare l'array in blocchi contigui (non ancora esplorati) della stessa dimensione (dimensione che dipenderà dalla fase) e quindi di verificare posizioni dell'array che sono uniformemente distribuite lungo la lunghezza del vettore. Dobbiamo fare questo in modo che il numero di posizioni del vettore che verifichiamo complessivamente sia dell'ordine di $O(\frac{n}{\ell})$. Procediamo nel seguente modo. Nella fase 1 verifichiamo il contenuto del vettore in posizione $n/2$, nella fase 2 proviamo le posizioni $n/4$ e $3n/4$ e nella fase tre spezziamo ancora le porzioni di array non viste in due. Più formalmente, sia L_i l'insieme degli indici degli elementi di A che controlliamo nella fase i. Abbiamo $L_1 = \{n/2\}$; e, per $i > 1$, L_i contiene tutti gli indici $j \pm \frac{n}{2^i}$ per ogni $j \in L_{i-1}$. Lo schema è illustrato in figura 2.

Per analizzare la complessità di questo algoritmo, si noti che completata la fase i l'algoritmo ha spezzato l'array in 2^i sottoarray tutti di dimensione $\frac{n}{2^i}$ e quindi ha guardato $O(2^i)$ elementi. Ora, sia k l'ultima fase che termina senza aver scoperto un 1 del blocco contiguo di uni (blocco che scopro quindi nella fase $k+1$). Per costruizione deve essere

che $\ell \leq \frac{n}{2^k}$ perché il blocco contiguo di uni deve essere interamente contenuto in uno dei 2^k sottoarray non esplorati. Quindi, come conseguenza abbiamo che $2^k \leq \frac{n}{\ell}$. Poiché nella fase $k+1$ l'algoritmo prova al più il doppio delle posizioni provate nella fase k e queste sono al più $O(2^k)$, abbiamo che il numero di elementi guardati complessivamente è al più $2 \cdot O(2^k) = O(\frac{n}{\ell})$.

Esercizio 4 (*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 diposizione $k = 2$ 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 4 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

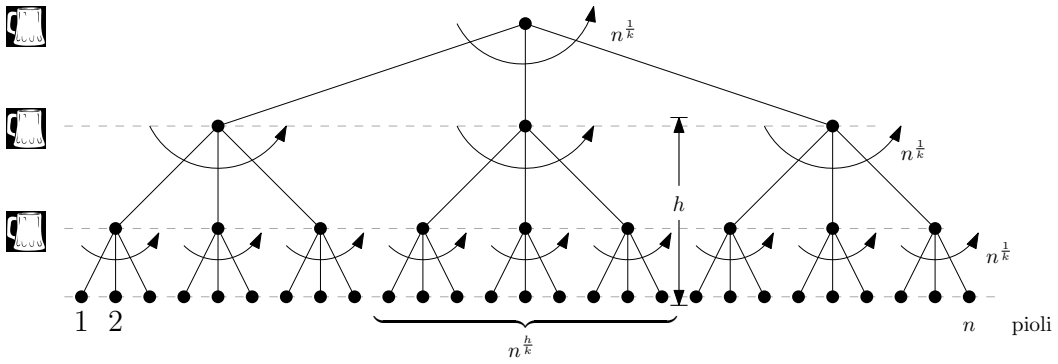

Figura 3: Il caso $k > 2$: l'albero completo di grado $\sqrt[k]{n}$ associato all'istanza.

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, da 1 a n. Un esempio per $n = 27$ e $k = 3$ è mostrato in Figura 3. 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 $k\lceil \sqrt[k]{n} \rceil \leq k\sqrt[k]{n}+k = O(k\sqrt[k]{n})$.