Problem Set 2

docente: Luciano Gualà

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

(*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 (*visite di alberi*)
Sia T un albero binario di n nodi con radice r. La *profondità* di un nodo v è il numero di archi del cammino da v alla radice. Un nodo u è un *discendente* di v se u si trova nel sottoalbero di T radicato in v. Si assuma che T è mantenuto attraverso una struttura collegata e che ogni nodo v mantenga i puntatori al padre e ai figli ($v.p$, $v.s$, $v.d$). Si progetti un algoritmo con complessità temporale $O(n)$ che, preso T, restituisca il nodo v di profondità minima la cui profondità è maggiore o uguale al numero dei suoi discendenti. Si fornisca lo pseudocodice dettagliato dell'algoritmo e possibilmente non si usino variabili globali e passaggi di parametri per riferimento.

Soluzione Esercizio 2
Chiamiamo un generico nodo che rispetta la condizione dell'algoritmo, ovvero, la cui profondità è maggiore o uguale al numero dei suoi discendenti, nodo speciale. L'idea è quella di effettuare una visita dell'albero in modo tale che, quando si visita un generico nodo v, sia possibile capire se v è un nodo speciale e quindi conteggiarlo nel calcolo del nodo speciale di profondità minima. Come succede spesso quando si progettano algoritmi di visita, la cosa complicata è capire come raccogliere informazioni durante il processo. Qui per esempio occorre tenere traccia della profondità dei nodi, del numero dei discendenti, e trovare un modo per mantenere il nodo speciale di profondità minima che sarà poi restituito in output. Una soluzione è questa. Per un dato nodo v, denotiamo con T_v il sottoalbero di T radicato in v. Diamo un algoritmo ricorsivo `MinProfSpecialeRic`(v, h) che, quando è chiamato su un nodo v di profondità h, restituisce una tripla (u_m, h_m, k), dove u_m è il nodo speciale di profondità minima contenuto in T_v, h_m è la profondità di u_m e k è il numero di discendenti di v (ovvero il numero di nodi di T_v).

Lo pseudocodice dell'algoritmo è il seguente:

Algorithm 1: `MinProfSpeciale`(r)

$(u, t, k) =$ `MinProfSpecialeRic`$(r, 0)$;
return u

Algorithm 2: `MinProfSpecialeRic`(v, h)

if $v = null$ **then**
  return $(null, +\infty, 0)$
$(u_s, h_s, k_s) =$ `MinProfSpecialeRic`$(v.s, h + 1)$;
$(u_d, h_d, k_d) =$ `MinProfSpecialeRic`$(v.d, h + 1)$;
$ndisc = 1 + k_s + k_d$;
if $h \geq ndisc$ **then**
  return $(v, h, ndisc)$
if $h_s \leq h_d$ **then**
  return $(u_s, h_s, ndisc)$
else
  return $(u_d, h_d, ndisc)$

L'algoritmo ha chiaramente complessità temporale $O(n)$: ogni nodo è visitato una sola

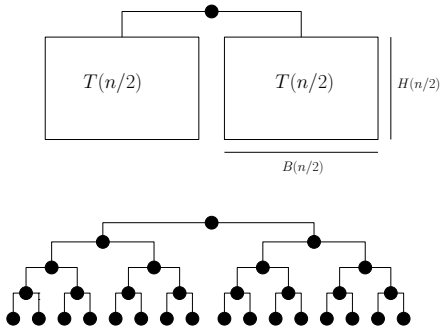

Figura 1: Embedding il cui bounding box ha area $\Theta(n \log n)$.

volta e il tempo speso per esso è costante.

Esercizio 3 (*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 3 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 1. 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 $B(n) = 2B(n/2) + \Theta(1)$.

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 2.

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 $\Theta(n)$.

Esercizio 4 *(Strutture dati: mantenere dinamicamente il mediano)*
Si progetti una struttura dati che mantiene un insieme S di elementi con chiavi prese da un dominio totalmente ordinato soggetto alle seguenti operazioni:

- `Insert`(S, k): inserisce un nuovo elemento di valore k in S.

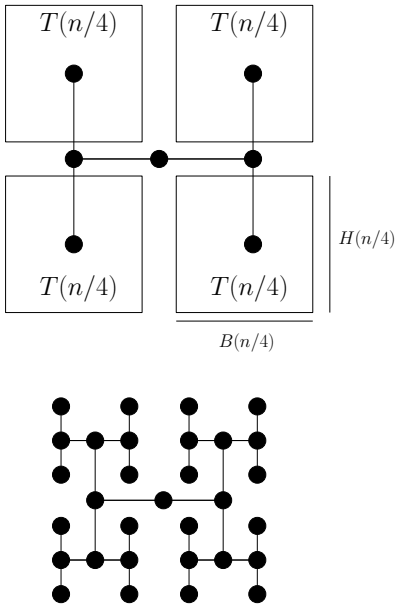

Figura 2: Embedding (asintoticamente) ottimo il cui bounding box ha area $\Theta(n)$.

- `Delete`(S, x): rimuove l'elemento x da S. Si pensi ad x come al puntatore (riferimento diretto) all'elemento da cancellare.
- `Median`(S): restituisce (senza rimuoverlo) il valore dell'elemento mediano contenuto in S; ovvero, se indichiamo con n il numero di elementi contenuti in S, si vuole restituire il valore dell'$\lceil \frac{n}{2} \rceil$-esimo minimo di S.

Tutte le operazioni devono avere complessità temporale (asintoticamente) logaritmica nel numero di elementi presenti nella struttura dati.

Soluzione Esercizio 4
L'idea è quella di mantenere l'insieme S mantenendo di fatto due code con priorità H^- e H^+ in modo che risulti sempre vera la seguente invariante. Se il numero di elementi di S è n, vogliamo che la coda con priorità H^- contenga gli $\lceil \frac{n}{2} \rceil$ elementi più piccoli di S mentre la coda con priorità H^+ contenga i restanti $\lfloor \frac{n}{2} \rfloor$ elementi più grandi di S. Implementiamo H^- come un max-heap, mentre H^+ come un min-heap. Le operazioni sugli heap sono le solite: è possibile trovare o estrarre il minimo o il massimo, a seconda del tipo di heap, inserire e cancellare. Inoltre assumiamo di aver anche un metodo – che chiameremo `size` – che ritorna il numero di elementi contenuti nell'heap. E' possibile implementare gli heap in modo tale che tutte le operazioni richiedano tempo asintoticamente logaritmico nel numero di elementi contenuti nell'heap (è possibile per esempio usare un semplice heap binario).

Si noti che una volta che questa invariante risulta vera il mediano di S è l'elemento di valore massimo dentro H^-. Per implementare l'inserimento e la cancellazione di elementi di S l'idea è quella di inserire/cancellare l'elemento nell'opportuno heap e, se l'invariante è inficiata, ristabilirla opportunamente. Gli pseudocodici delle operazioni si spiegano da soli. Usiamo una procedura ausiliaria `rebalance`(H^-, H^+) per ristabilire l'invariante ogni volta che essa è inficiata a fronte di un'operazione di inserimento o cancellazione.

Algorithm 3: `Median`(S)

return `findMin`(H^-);

Algorithm 4: `Insert`(S, k)

if $size(H^-) = 0$ *o* $Median(S) \geq k$ **then**
    `inser`(H^-, k)
else
    `inser`(H^+, k)
`rebalance`(H^-, H^+)

Algorithm 5: `Delete`(S, x)

if $x \in H^-$ **then**
    `delete`(H^-, x)
else
    `delete`(H^+, x)
`rebalance`(H^-, H^+)

Algorithm 6: `rebalance`(H^-, H^+)

if $size(H^-) < size(H^+)$ **then**
    $m =$ `deleteMin`(H^+);
    `inser`(H^-, m)
if $size(H^-) > size(H^+)+1$ **then**
    $m =$ `deleteMax`(H^-);
    `inser`(H^+, m)

Chiaramente, tutte le operazioni implementate hanno complessità $O(\log n)$ dove n è il numero di elementi correntemente nella struttura dati, poiché ogni gli algoritmi forniti effettuano un numero costante di operazioni sulle due code con priorità, ognuna della quale ha costo $O(\log n)$.