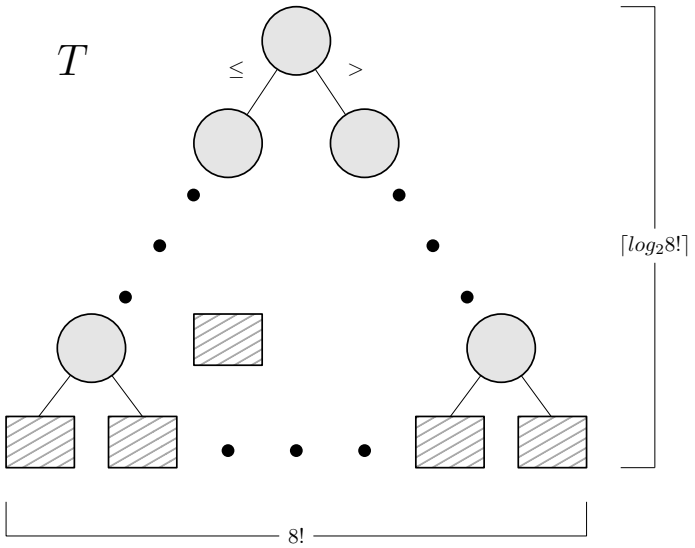
Problem Set 3

docente: Luciano Gualà

Esercizio 1 *(una domanda semplice ma non troppo)*
Dire se può esistere un algoritmo di ordinamento basato su confronti che ordina un insieme di 8 elementi facendo nel caso peggiore al più 15 confronti. Motivare la risposta.

Soluzione esercizio 1
Consideriamo un **qualsiasi** algoritmo di ordinamento A basato su confronti. Tale algoritmo può essere rappresentato tramite un opportuno albero binario di decisione T in cui ogni nodo interno rappresenta un confronto tra due elementi ed ogni foglia rappresenta un possibile output di A, cioè una particolare sequenza ordinata degli elementi in input.

Dal momento che l'algoritmo A deve ordinare correttamente qualsiasi insieme di 8 elementi (forniti in qualsiasi ordine), l'albero T deve contenere almeno tante foglie quante le possibili permutazioni di 8 elementi. Dunque il numero di foglie di T è almeno 8!.



È noto che per un qualsiasi albero binario T', l'altezza $h(T')$ è almeno logaritimica nel numero di foglie $\ell(T')$, più precisamente vale: $h(T') \geq \log_2 \ell(T')$.

Applicando la precedente proprietà all'albero T si ottiene: $h(T) \geq \log_2 8! = \log_2 40320 > 15.299$ per cui l'altezza di T deve essere almeno pari a 16. Ciò significa che esiste un cammino, dalla radice ad una foglia di T che attraversa almeno 16 nodi interni (radice inclusa), dunque l'algoritmo A (**qualunque esso sia**) deve eseguire, nel caso peggiore, più di 15 confronti.

Esercizio 2
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 profontifà 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 volta e il tempo speso per esso è costante.

Esercizio 3
Siano dati n punti disposti sul piano Euclideo, dove il punto p_i ha cordinate (x_i, y_i), $i = 1, \ldots, n$. Si progetti un algoritmo che, preso in input l'insieme degli n punti, un valore $k \in \{1, 2, \ldots, n\}$ e un ulteriore punto *target* t di coordinate (x_t, y_t), restituisca i k punti dell'insieme che sono più vicini a p_t (rispetto alla distanza euclidea). L'algoritmo deve avere complessità temporale (nel caso peggiore) $O(n + k \log n)$.

Soluzione esercizio 3
Notiamo che è possibile calcolare la distanza di ogni punto p_i dal punto target t in un numero costante di operazioni, tramite la ben nota formula:

$$d(p_i, t) = \sqrt{(x_i - x_t)^2 + (y_i - y_t)^2}$$

L'algoritmo inizializza un array D di dimensione n, in cui ogni elemento è associato ad un punto. In particolare nella cella i sarà contenuta la coppia (p_i, d_i) in cui $d_i = d(p_i, t)$ è la distanza del punto p_i dal punto target t.

Tramite la procedura `Heapify`, si riordinano gli elementi di D in modo da formare un Min-Heap rispetto al secondo elemento di ogni coppia. La costruzione dell'heap richiede tempo $O(n)$.

A questo punto i k punti più vicini a t possono essere ottenuti tramite k estrazioni del minimo da D, ognuna delle quali richiede tempo $O(\log n)$.

Algorithm 3: `FindKNearestNeighbors`$(p_1, \ldots, p_n)$

```
D ← Array con n elementi del tipo (punto, distanza)
for i ← 1 to n do
    D[i] ← sqrt((x_i − x_t)^2 + (y_i − y_t)^2)

// Costruisci un Min-Heap rispetto alle distanze.
Heapify(D)                                          // O(n)

for i ← 1 to k do
    yield Min(D)                                    // O(1)
    DeleteMin(D)                                    // O(log 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.
- `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 4: `Median`(S)

return `findMin`(H^-);

Algorithm 5: `Insert`(S, k)

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

Algorithm 6: `Delete`(S, x)

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

Algorithm 7: `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 $O(\log n)$.