Famiglia

Trovare i limiti superiori e inferiori più stretti possibili per la seguente famiglia di equazioni di ricorrenza, per valori di a interi positivi.

$$T(n) = \begin{cases} aT(\lfloor n/2 \rfloor) + n^{a-1} & n \geq 2 \\ 1 & n < 2 \end{cases}$$

Inserisci l'albero

Sia T un albero binario contenente $n \geq 1$ nodi e A un vettore ordinato contenente n interi distinti.

Oltre ai normali campi di un albero binario, ogni nodo t di T contiene un campo $t.size$ che contiene il numero di nodi del sottoalbero radicato in t (già inizializzato) e un campo $t.value$ (inizialmente vuoto).

Scrivere un algoritmo

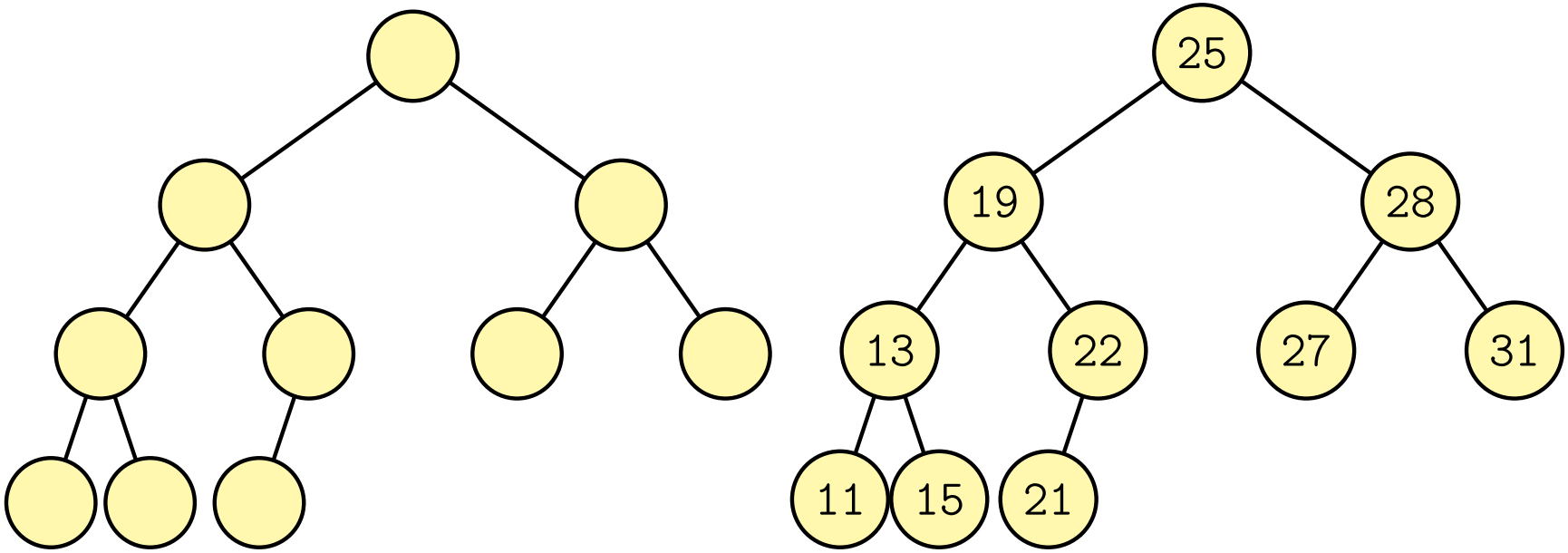
binaryInsert(Tree T, **int**[ ] A, **int** n)

che inserisca tutti i valori di A in T, in modo tale che l'albero risultante sia un albero binario di ricerca.

Discutere correttezza e complessità computazionale dell'algoritmo proposto.

Inserisci l'albero

Esempio: a sinistra, un sottoalbero T dato in input, a destra lo stesso sottoalbero dopo che sono stati inseriti i valori
$A = [11, 13, 15, 19, 21, 22, 25, 27, 28, 31]$.



Maggioranza

Scrivere una funzione

boolean hasMajority(**int**[ ] A, **int** n)

che prenda in input un vettore ordinato A contenente n interi e restituisca **true** se A contiene un valore di maggioranza, ovvero un valore che compare più di $n/2$ volte; restituisca **false** altrimenti. Soluzioni di costo computazionale lineare non verranno prese in considerazione.

Discutere correttezza e complessità dell'algoritmo proposto.

Minecraft

In Minecraft, una carta geografica è rappresentata da una matrice $n \times n$ di celle, ognuna delle quali rappresenta un'altitudine intera. Un valore ≤ 0 corrisponde ad un mare, mentre un valore > 0 corrisponde ad un'isola.

Scrivere un algoritmo che prenda in input una matrice A di interi e la sua dimensione positiva n, e restituisca l'altezza media dell'isola più elevata.

Discutere informalmente la correttezza della soluzione proposta e calcolare la complessità computazionale.

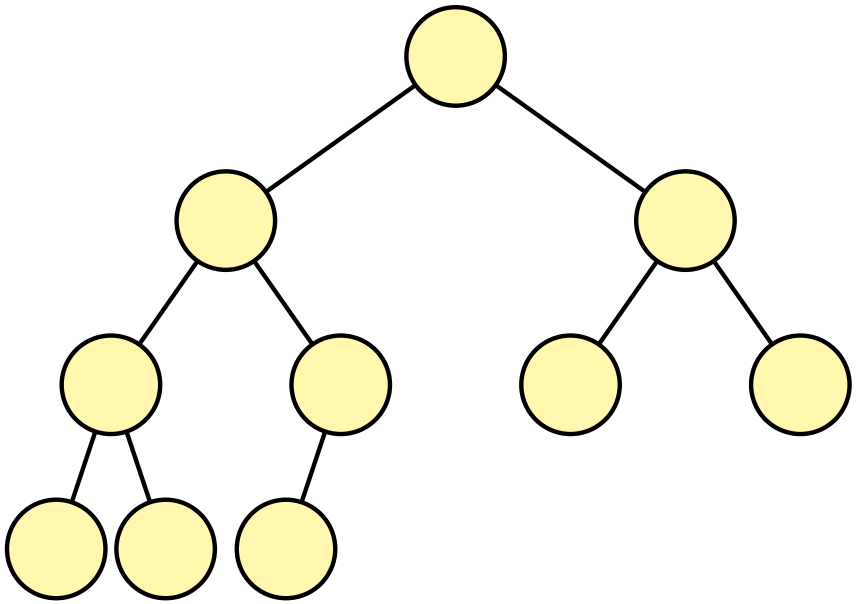
Minecraft

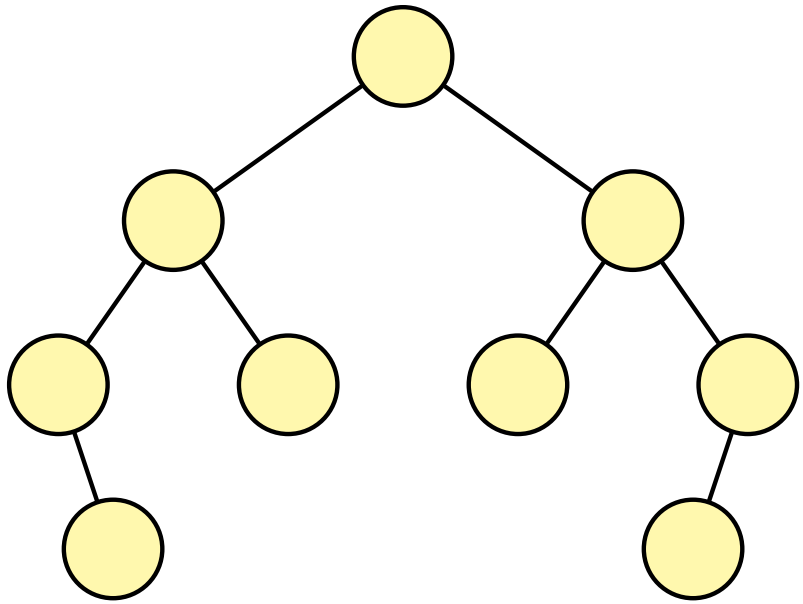
Ad esempio, nella matrice seguente ci sono tre isole, una 2×3 in alto a sinistra con altezza media 1.5, una 4×1 a destra con altezza media 1, una 1×2 in basso a sinistra con altezza media 2. L'isola con altezza media più alta è quella in basso a sinistra, e quindi l'algoritmo deve restituire 2.

1	1	1	0	1
2	2	2	-1	1
0	-1	0	-2	1
0	0	-1	-2	1
2	2	-1	0	0

Alberi simmetrici

Scrivere un algoritmo **boolean** isSymmetric(TREE T) che prenda in input un albero binario T non vuoto e restituisca **true** se T è simmetrico, **false** altrimenti. Un albero binario è simmetrico se il sottoalbero sinistro della radice è un'immagine *speculare* del sottoalbero destro della radice. L'albero binario a sinistra non è simmetrico, mentre lo è quello di destra.





Discutere correttezza e complessità computazionale dell'algoritmo

Spoiler alert!

Famiglia (24/07/20)

Data la forma della ricorrenza, è possibile utilizzare il Master Theorem, versione base.

- Per $a = 1$, abbiamo $\alpha = \log_2 1 = 0$, $\beta = 0$; siamo nel secondo caso, $T(n) = \Theta(\log n)$.
- Per $a = 2$, abbiamo $\alpha = \log_2 2 = 1$, $\beta = 1$; siamo nel secondo caso, $T(n) = \Theta(n \log n)$.
- Per $a = 3$, abbiamo $\alpha = \log_2 3 < 2$, $\beta = 2$; siamo nel terzo caso, $T(n) = \Theta(n^2)$.
- In generale, è possibile dimostrare che $\log_2 a < a - 1$ per tutti i valori interi $a \geq 3$; siamo nel terzo caso e si ottiene $T(n) = \Theta(n^{a-1})$.

Inserisci l'albero (03/07/20)

Divide-et-impera su albero. Dato un nodo t, siano t_l e t_r i suoi sottoalberi sinistro e destro di dimensione s_l e s_r. Si chiamerà quindi ricorsivamente la procedura sul sottoalbero sinistro, con s_l valori; si piazzerà il valore $s_l + 1$-esimo nella radice; e quindi si richiamerà la procedura sul sottalbero destro, con s_r valori.
Poichè questo algoritmo è sostanzialmente una visita, la sua complessità sarà pari a $\Theta(n)$.

Inserisci l'albero (03/07/20)

---

binaryInsert(TREE T, **int**[ ] A)

---

biRec($T, A, 1$)

---

biRec(TREE t, **int**[ ] A, **int** i)

---

int $leftSize = 0$
if $T.left \neq$ **nil then**
  $leftSize = t.left.size$
  biRec($t.left, A, i$)
$t.value = A[i + leftSize]$
if $T.right \neq$ **nil then**
  biRec($t.right, A, i + leftSize + 1$)

Maggioranza (03/07/20)

Se esiste un valore di maggioranza, deve essere il valore contenuto nella posizione mediana $\lfloor (n+1)/2 \rfloor$.

- Se n è dispari, questo è l'elemento centrale. Qualunque maggioranza deve includere questo elemento e altri $(n-1)/2$ elementi; infatti, a sinistra e destra di tale elemento ci stanno esattamente $(n-1)/2$ elementi.
- Se n è pari, sia $n/2$ che $n/2+1$ devono appartenere alla maggioranza, per lo stesso ragionamento di cui sopra.

Dato questo valore, si effettua ricerca dicotomica che assuma l'esistenza di tale valore e restituisca l'indice più basso in cui tale valore si presenta.

Conoscendo l'indice del primo elemento, è necessario vedere se l'elemento in posizione $A[\mathit{first} + \lfloor n/2 \rfloor]$ è uguale ad esso, nel qual caso esiste una maggioranza di elementi uguali, essendo il vettore ordinato.

Maggioranza (03/07/20)

```
int hasMajority(int[] A, int n)
int candidate = A[⌊(n + 1)/2⌋])
int first = searchFirst(A, 1, n, candidate)
return A[first] == A[first + ⌊n/2⌋]
```

```
int searchFirst(int[] A, int i, int j, int v)
if i == j then
    return i
else
    int m = ⌊(i + j)/2⌋
    if v ≤ A[m] then
        return searchFirst(A, i, m, v)
    else
        return searchFirst(A, m + 1, j, v)
```

Minecraft (21/08/18)

Utilizziamo la matrice M di input come vettore *visited*; in altre parole, una cella è visitabile se il suo valore è maggiore di zero. Una volta scoperta, il valore viene posto a zero. Se la matrice deve essere conservata, sarà necessario farne una copia prima (con costo $O(n^2)$).

L'algoritmo scritto qui simula una ricerca delle componenti connesse nel grafo. Si parte da ogni cella non visitata, e si lancia una visita DFS. Per ogni nodo visitato, si aggiunge l'altezza ad un accumulatore di altezze *height*, si aggiunge 1 ad un contatore *count* e si marca il nodo come visitato. Si continua quindi la visita affrontando le quattro caselle vicine. Al termine della visita ricorsiva, si calcola l'altezza media e la si confronta con il massimo.

Visto che ogni cella può essere visita al più una volta, la complessità è $O(n^2)$.

Minecraft (21/08/18)

```
int topIsland(int[][] M, int n)
float max = 0
for r = 1 to n do
    for c = 1 to n do
        if M[r][c] > 0 then
            int count = 0
            int height = 0
            dfsRec(M, n, r, c, &count, &height)
            max = max(max, height/count)
return max
```

Minecraft (21/08/18)

---

dfsRec(**int**[][] M, **int** n, **int** r, **int** c, **int** $count$, **int** $height$)

---

if $1 \leq r < n$ **and** $1 \leq c < n$ **and** $M[r][c] > 0$ **then**

- $height = height + M[r][c]$
- $count = count + 1$
- $M[r][c] = 0$ % Visited
- dfsRec$(M, n, r-1, c, \&count, \&height)$
- dfsRec$(M, n, r+1, c, \&count, \&height)$
- dfsRec$(M, n, r, c-1, \&count, \&height)$
- dfsRec$(M, n, r, c+1, \&count, \&height)$

Alberi simmetrici (24/07/20)

Un sottalbero è simmetrico se i suoi sottoalberi destro e sinistro sono speculari.

Due sottoalberi t_1, t_2 sono speculari se e solo se:

- il sottoalbero sinistro di t_1 è speculare al sottoalbero destro di t_2
- il sottoalbero destro di t_1 è speculare al sottoalbero sinistro di t_1.

Il caso base è dato due nodi **nil** (si ritorna **true**) o da uno nodo **nil** e un nodo non **nil** (si ritorna **false**).

La procedura effettua una visita su entrambi gli alberi, e quindi ha complessità $\Theta(n)$.

Alberi simmetrici (24/07/20)

boolean isSymmetric(TREE T)

return isMirror($T.left, T.right$)

boolean isMirror(TREE t_{L}, TREE t_{R})

```
if t_L == nil and t_R == nil then
    return true
else if t_L ≠ nil and t_R ≠ nil then
    return isMirror(t_L.right, t_R.left) and isMirror(t_L.left, t_R.right)
else
    return false
```