Emre Oytun
2001040046099

1) I take the limit result as 0, when denominator is faster (goes to ∞).

a) $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{2^n}{2^{2n}} = \lim_{n\to\infty} \frac{2^n}{(2^n)^2} = \lim_{n\to\infty} \frac{1}{2^n} = 0$$

Since $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, $f(n) \in O(g(n))$.

b) $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^2}{n^3} = \lim_{n\to\infty} \frac{1}{n} = 0.$$

Since $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, $f(n) \in O(g(n))$.

c) $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{3n+1}{2n-5} \overset{L'}{=} \lim_{n\to\infty} \frac{3}{2} = 3/2$$

Since $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 3/2$ (a constant), $f(n) \in \Theta(g(n))$.

d) $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{4n^2}{n^2} = \lim_{n\to\infty} 4 = 4.$$

Since $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 4$ (a constant), $f(n) \in \Theta(g(n))$.

e) $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\log_2(n)}{\log_{10}(n)} \overset{L'}{=} \lim_{n\to\infty} \frac{\frac{1}{n\ln 2}}{\frac{1}{n\ln 10}} = \lim_{n\to\infty} \frac{n\ln 10}{n\ln 2}$$

$$= \lim_{n\to\infty} \frac{\ln 10}{\ln 2} = \ln 10/\ln 2$$

Since $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{\ln 10}{\ln 2}$ (a constant), $f(n) \in \Theta(g(n))$.

f) $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{2^n}{3^n} \overset{L'}{=} \lim_{n\to\infty} \left(\frac{2}{3}\right)^n = 0.$$

Since $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, $f(n) \in O(g(n))$.

g) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^3}{1000n^2} = \lim_{n\to\infty} \frac{n}{1000} = \infty$.

Since $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, $f(n) \in \Omega(g(n))$.

h) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{5n+4}{2n+2} \overset{L'}{=} \lim_{n\to\infty} \frac{5}{2} = 5/2$

Since $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{5}{2}$ (a constant), $f(n) \in \Theta(g(n))$.

i) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\sqrt{n}}{\log_2(n)} \overset{L'}{=} \lim_{n\to\infty} \frac{\frac{1}{2} n^{-1/2}}{\frac{1}{n \ln 2}}$

$= \lim_{n\to\infty} \frac{n \ln 2}{2 n^{1/2}} = \lim_{n\to\infty} \frac{\ln 2}{2} \cdot n^{1/2} = \infty$.

Since $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, $f(n) \in \Omega(g(n))$.

j) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{2^n}{2^{n+1}} = \lim_{n\to\infty} \frac{2^n}{2 \cdot 2^n} = \lim_{n\to\infty} \frac{1}{2} = \frac{1}{2}$.

Since $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{1}{2}$ (a constant), $f(n) \in \Theta(g(n))$.

---

2) The list in order of growth from faster to slower:

$n^{2^n} > n! > 10^n > 2^n > n^2 \log n > n+1 > \sqrt{n+5} > \log(n) > \frac{1}{2^n}$

The list using order class:

$O(\frac{1}{2^n}) \subset O(\log n) \subset O(\sqrt{n+5}) \subset O(n+1) \subset O(n^2 \log n) \subset O(2^n)$

$\subset O(10^n) \subset O(n!) \subset O(n^{2^n})$

⇒ I will prove pairs in order, the final result comes from transitivity.

$\frac{1}{2n} \in O(\log n)$ proof:

$$\lim_{n\to\infty} \frac{\frac{1}{2n}}{\log n} \overset{L'}{=} \lim_{n\to\infty} \frac{-\frac{1}{4n^2}}{\frac{1}{n}} = \lim_{n\to\infty} -\frac{n}{4n^2} = \lim_{n\to\infty} -\frac{1}{4n} = 0$$

$\log n \in O(\sqrt{n+5})$ proof:

$$\lim_{n\to\infty} \frac{\log n}{\sqrt{n+5}} \overset{L'}{=} \lim_{n\to\infty} \frac{\frac{1}{n}}{\frac{1}{2\sqrt{n+5}}} = \lim_{n\to\infty} \frac{2\sqrt{n+5}}{n} \overset{L'}{=} \lim_{n\to\infty} \frac{2\frac{1}{2\sqrt{n+5}}}{1}$$

$$= \lim_{n\to\infty} \frac{1}{\sqrt{n+5}} = 0$$

$\sqrt{n+5} \in O(n+1)$ proof:

$$\lim_{n\to\infty} \frac{\sqrt{n+5}}{n+1} \overset{L'}{=} \lim_{n\to\infty} \frac{\frac{1}{2\sqrt{n+5}}}{1} = \lim_{n\to\infty} \frac{1}{2\sqrt{n+5}} = 0$$

$n+1 \in O(n^2 \log n)$ proof:

$$\lim_{n\to\infty} \frac{n+1}{n^2\log n} \overset{L'}{=} \lim_{n\to\infty} \frac{1}{2n\log n + \frac{1}{n}n^2} = \lim_{n\to\infty} \frac{1}{2n\log n + n} = 0$$

$n^2 \log n \in O(2^n)$ proof:

$$\lim_{n\to\infty} \frac{n^2\log n}{2^n} \overset{L'}{=} \lim_{n\to\infty} \frac{2n\log n + \frac{1}{n}n^2}{2^n \cdot \ln 2} = \lim_{n\to\infty} \frac{2n\log n + n}{2^n \cdot \ln 2}$$

$$\overset{L'}{=} \lim_{n\to\infty} \frac{2\log n + \frac{1}{n}\cdot 2n + 1}{2^n \cdot (\ln 2)^2} = \lim_{n\to\infty} \frac{2\log n + 3}{2^n (\ln 2)^2} \overset{L'}{=} \lim_{n\to\infty} \frac{\frac{2}{n}}{2^n (\ln 2)^3}$$

$$= \lim_{n\to\infty} \frac{2}{n \cdot 2^n \cdot (\ln 2)^3} = 0.$$

$\infty \cdot \infty \cdot (\ln 2)^3 \to \infty$

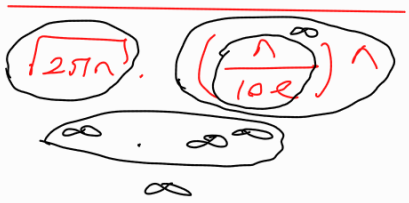
$2^n \in O(10^n)$ proof:

$$\lim_{n\to\infty} \frac{2^n}{10^n} = \lim_{n\to\infty} \left(\frac{2}{10}\right)^n = 0.$$

$10^n \in O(n!)$ proof:

Stirling's Formula : $n! \simeq \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n$, when n goes to ∞.

$$\lim_{n\to\infty} \frac{10^n}{n!} = \lim_{n\to\infty} \frac{10^n}{\sqrt{2\pi n} \cdot \frac{n^n}{e^n}} = \lim_{n\to\infty} \frac{1}{\sqrt{2\pi n} \cdot \left(\frac{n}{10e}\right)^n}$$

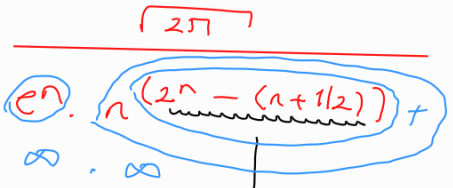


$$= 0$$

$n! \in O(n^{2^n})$ proof:

$$\lim_{n\to\infty} \frac{n!}{n^{2^n}} \overset{L'}{=} \lim_{n\to\infty} \frac{\sqrt{2\pi n} \cdot \frac{n^n}{e^n}}{n^{2^n}} = \lim_{n\to\infty} \frac{\sqrt{2\pi}}{e^n} \cdot n^n \cdot n^{1/2} \cdot n^{-2^n}$$

$$= \lim_{n\to\infty} \frac{\sqrt{2\pi}}{e^n \cdot n^{(2^n - (n+1/2))}} = 0$$



* Since 2^n is greater than $(n+1/2)$ when n goes to infinity as proven below, this expression at least positive.

$$\lim_{n\to\infty} \frac{2^n}{n+1/2} \overset{L'}{=} \frac{2^n \cdot \ln 2}{1} = \infty \quad (2^n > n+1/2)$$

---

3) I used 2 algorithms commonly for the problems.
Therefore, I'm explaining and analyzing these at first.

function getTreeElementsSorted(root, sortedArr[0, ..., n-1]) :

* Returns an array containing the BST elements in sorted way.

i) Firstly, it checks if the given root is None, if it is it just returns.

ii) It makes a recursive call giving the left subtree.

iii) It appends the given root's value to the sorted array.

iv) It makes a recursive call giving the right subtree.

v) It returns the sorted array.

* It's a recursive method, so it can be analyzed using recurrence relation.
* It's actually an inorder traversal of tree.
* There are a few constant order operations inside the function except the recursive calls.
* Each time it calls the method for left and right subtrees.
* Worst case happens when left subtrees are always empty.

$T(0) = 1$ (only 1 condition)

$T(n) = \underbrace{T(0)}_{1} + T(n-1) + c$ ⟶ c is total operation number constant including assignments, arithmetic operations, conditions.

$T(n) = T(n-1) + (c+1)$ $\qquad T(n-1) = T(n-2) + (c+1)$

$= T(n-2) + (c+1).2$

$= T(n-3) + (c+1).3$

$\vdots$

$= T(n-k) + (c+1).k$

$n-k=0$

$\boxed{k=n}$ ⇒ $T(n) = T(0) + (c+1).n$

$\boxed{T(n) = 1 + (c+1).n}$ ⟶ $\boxed{T(n) \in \Theta(n) \Rightarrow T(n) \in O(n)}$

Since $\lim_{n \to \infty} \frac{1+(c+1)n}{n} = c+1$

function createTreeFromSortedArr(arr[0,---,n-1])

* Creates a balanced BST using the given sorted array.

1) It calls the createTreeFromSortedArrRec method by passing the given array, left index as 0, right index as array's length − 1.

2) This recursive method first checks the length and returns None if the length is 0, and returns a node containing the current element if it's 1.

3) It finds the middle index/element. Then makes 2 recursive calls one for left array, one for right array.

4) Finally, it creates a tree using the middle element as root,

the recursive call resulting trees for left and right subtree.

* It's a recursive method, so recurrence relation can be used.
* At worst, there are C constant operations and 2 recursive calls which are always the half size.

$T(1) = 1$
$T(0) = 1$ (if there are no elements at first)

$$T(n) = 2T(n/2) + c$$

$$T(n/2) = 2T(n/4) + c$$

$$= 2^2 T(n/4) + 2 \cdot c + c$$

$$= 2^3 T(n/8) + 2^2 \cdot c + 2 \cdot c + c$$

$$\vdots$$

$$= 2^k T(n/2^k) + c(2^0 + 2 + \cdots + 2^{k-1})$$

$$= 2^k T(n/2^k) + c \sum_{i=0}^{k-1} 2^i \qquad \left(\sum_{i=0}^{k-1} r^i = \frac{r^k - 1}{r - 1} \right)$$

$$= 2^k T(n/2^k) + c \frac{2^k - 1}{2 - 1}$$

Since $T(1) = 1$;

$\frac{n}{2^k} = 1$

$2^k = n$

$k = \log_2 n$

$$= 2^{\log_2 n} T(1) + c(2^{\log_2 n} - 1)$$

$$= n + c(n - 1)$$

$$T(n) = (1 + c)n - c \Rightarrow T(n) \in \Theta(n) \Rightarrow T(n) \in O(n)$$

Since: $\lim_{n \to \infty} \frac{(1+c)n - c}{n} = 1 + c$

NOTE: This is valid when $n = 2^k$.

By interpolation: $T(n) \in \Theta(n)$. (when $n = 2^k$, $k \geq 0$)

i-) $T(n)$ is non-decreasing ✓
ii-) n is non-decreasing. ✓
iii-) $c \cdot n \in \Theta(n)$, n is θ-invariant under scaling ✓

Therefore, it's valid for all n.
I'll use this time function for the questions below when this algorithm is needed. Since I explained that it's theta complexity holds for all n, it's valid when the questions' complexity below is in theta.

a) mergeTwoBST(root1, root2) analyzing:

1) It calls getTreeElementsSorted method to get the elements of the given trees in two separate sorted arrays.

2) It calls mergeTwoSortedArray method to merge the sorted arrays.

Merging sorted arrays:

2.1) Keep two indexes pointing two arrays.
2.2) In while loop, put the smaller element to the result array. If they're equal, it ignores one of them so there's no duplicates.
2.3) Continue after there are no elements in one of the arrays.
2.4) Put the remaining elements of other array into result array.
2.5) Return the result array.

3) After merging two sorted arrays, call createTreeFromSortedArray method and return the result.

* There are C_6 constant operations, 2 getTreeElementsSorted function call and 1 createTreeFromSortedArray call.

$T_1(n)$: getTreeElementsSorted method's time function.
$T_2(n)$: createTreeFromSortedArray method's time function.
$T_3(n,n)$: mergeTwoSortedArray method's time function
$T_4(n)$: mergeTwoBST method's time function

$$T_1(n) = 1 + (1+c_1).n$$
$$T_2(n) = (1+c_2)\,n - c_2$$

$T_3(n,n)$ – mergeTwoSortedArray for (n,n) parameters :

* There are C_3 constant operations outside while loops.
* There are 3 while loops. First one has C_4 constant operations inside, the other 2 are the same and they have C_5 constant operations inside.
* The first while loop runs at least n times so that one array is done. Afterwards, one of the while loops below runs for other array.
* Let say the first while loop runs $n+k$ times.
  The other while loop runs $n-k$ times.

$$T_3(n,n) = C_3 + C_4(n+k) + C_5(n-k)$$
$$\boxed{T_3(n,n) = (C_4+C_5)n + C_3 + C_4k - C_5k}$$

$$T_4(n) = 2T_1(n) + T_3(n,n) + T_2(2n) + C_6$$
$$= 2(1+(1+c_1)n) + (C_4+C_5)n + C_3 + C_4k - C_5k +$$
$$(1+c_2)2n - c_2 + C_6$$

$$T_4(n) = (2+2c_1+c_4+c_5+2+2c_2)n + 2 + C_3 + C_4k - C_5k - c_2 + C_6$$

$T_4(n) \in \Theta(n)$ since $\lim_{n\to\infty} \frac{T(n)}{n} = 2+2c_1+c_4+c_5+2+2c_2$

So, $T_4(n) \in \Theta(n) \Rightarrow \boxed{T_4(n) \in O(n)}$

b) findKthSmallestElement(root, k) analysis:

// k should be equal or greater to 1. $k \in \mathbb{N}^+$

1) It checks if k is equal or smaller than 0, if it's then it returns None.
2) It calls getTreeElementsSorted method by passing root and an empty array as parameter.
3) Then, it checks if k is greater than length of the sorted array containing all tree elements. If it's, then it returns None.
4) It returns the element at (k-1). index from sorted array.

* There are C_2 constant operations, 1 getTreeElementsSorted function call.

$T_1(n)$: getTreeElementsSorted method's time function.
$T_2(n)$: findKthSmallestElement method's time function.

$$\boxed{T_1(n) = 1 + (1+C_1) \cdot n}$$

$$T_2(n) = T_1(n) + C_2$$
$$= 1 + (1+C_1)n + C_2$$

$$\boxed{T_2(n) = (1+C_1) \cdot n + C_2 + 1}$$

$$\boxed{T_2(n) \in \Theta(n)} \quad \text{since} \quad \lim_{n \to \infty} \frac{(1+C_1) \cdot n + C_2 + 1}{n} = 1 + C_1$$

$$\Downarrow$$

$$\boxed{T_2(n) \in O(n)}$$

c) balanceBST(root) analysis:

1) It creates an empty array and calls the getTreeElementsSorted method by passing the root and the array.
2) It calls createTreeFromSortedArray method by passing the sorted array which contains all BST elements in sorted way.
3) It returns the result since the tree returned from this method is balanced.

* There are C_3 constant operations, 1 getTreeElementsSorted function call, 1 createTreeFromSorted array function call.

$T_1(n)$: getTreeElementsSorted method's time function.
$T_2(n)$: createTreeFromSortedArray method's time function.
$T_3(n)$: balanceBST method's time function

$$T_1(n) = 1 + (1+c_1)\cdot n$$
$$T_2(n) = (1+c_2)n - c_2$$

$$T_3(n) = T_1(n) + T_2(n) + C_3$$
$$= 1 + (1+c_1)n + (1+c_2)n - c_2 + c_3$$
$$\boxed{T_3(n) = (2+c_1+c_2)n + 1 - c_2 + c_3}$$

$\boxed{T_3(n) \in \Theta(n)}$ since $\lim_{n\to\infty} \frac{(2+c_1+c_2)n + 1 - c_2 + c_3}{n} = 2+c_1+c_2$

$\Downarrow$

$\boxed{T_3(n) \in O(n)}$

d) findElementsInRange(root, a, b) analysis:

// Returns the elements in range [a, b] from tree.
// Returns an empty list if there is none.

1) It checks if $a > b$, if it's then it returns None. Since, it's an invalid range.

2) It creates an empty array and calls the getTreeElementsSorted method by passing the root and the array so the array is filled with BST elements in sorted way.

3) It iterates through the sorted array, and it adds the elements in the specified range to the result array.

4) It continues to iterate until it encounters an element which is greater than "b" upper bound.

5) It returns the result array.

* There are C_2 constant operations, 1 getTreeElementsSorted method call, a while loop which has C_3 constant operations inside.

* <u>In worst case</u>, while loop iterates through all elements in sorted array which has "n" elements.

$T_1(n)$: getTreeElementsSorted method's time function.
$T_2(n)$: findElementsInRange method's time function for worst case

$$\boxed{T_1(n) = 1 + (1+C_1).n}$$

$T_2(n) = T_1(n) + C_2 + C_3 n$ → while loop in worst-case

$= 1 + (1+C_1)n + C_2 + C_3 n$

$$\boxed{T_2(n) = (1+C_1+C_3)n + 1 + C_2}$$

$\boxed{T_2(n) \in \Theta(n)}$ since $\lim_{n\to\infty} \frac{(1+C_1+C_3)n + 1 + C_2}{n} = 1 + C_1 + C_3$

$\Downarrow$

$\boxed{T_2(n) \in O(n)}$

---

4)

```
i=2
while i<=n :
    if i % 2 != 0
        i = i-1
    else
        i = i * i
        i = i+1
    print(i)
```

⇒ Since i starts as 2 (even);
→ it doubles itself and add one which is an odd number.
→ when it's odd, it subtracts one so it's now an even number.

⇒ Therefore, it's going to n, being even for one iteration and odd for another iteration.

⇒ So, it doubles itself in every 2 iteration.

* If $n = 2^{(2^k)}$ $(k \geq 0)$: 2 5 4 17 16 ...

(2^{2^0}) $(k=0)$ (2^{2^1}) $(k=1)$ (2^{2^2}) $(k=2)$ ...

$\Rightarrow$ There are : $k+1$ even elements,
$+\ k$ odd elements.

$2k+1$ total elements

$\Rightarrow n = 2^{2^k}$

$\log_2 n = 2^k$

$\log_2 \log_2 n = k$

$\Rightarrow$ So, there are : $2k+1 = 2(\log_2 \log_2 n) + 1$ elements.

$W(n) = 2(\log_2 \log_2 n) + 1$.

$(n = 2^{(2^k)},\ k \geq 0)$

$\Rightarrow W(n) \in \Theta(\log \log n)$. Since constant terms and logarithm base does not matter.
$(n = 2^{(2^k)},\ k \geq 0)$

$\Rightarrow$ By interpolation: $W(n) = 2(\log_2 \log_2 n) + 1 \in \Theta(\log \log n)$.
when $n = 2^{(2^k)},\ k \geq 0$.

i-) $W(n) = 2(\log_2 \log_2 n) + 1$ is eventually non-decreasing. ✓

ii-) $\log \log n$ is eventually non-decreasing. ✓

iii-) $g(n) = \log \log n$
$g(c \cdot n) = \log \log (cn)$
$\Big\}$ $g(c \cdot n) \in \Theta(g(n))$, so $g(n)$ is Θ-invariant under scaling. ✓

As a result, $W(n) \in \Theta(\log \log n)$ for all n.

$W(n) \in \Theta(\log \log n) \Rightarrow W(n) \in O(\log \log n)$

Continues at next page ↓

5) The algorithm works like linear search. It iterates through every element and checks if the element is even. If it is, it's done; otherwise it continues till there are no elements remained.

```
def searchEven(arr):
    for num in arr:
        if num % 2 == 0:
            return num
    return None
```

$\Rightarrow$ Best-case occurs when the first element is even.

$B(n) = 1 \in \Theta(1)$.

$\Rightarrow$ Worst-case occurs when there are no even elements or only the last one is even.

$W(n) = n \in \Theta(n)$.

$\Rightarrow$ Average-case analysis: (from 1 to n)

Probability for algorithm to be done at the i-th index $p(i)$.

$$p(i) = \begin{cases} \frac{1}{5n} & , \text{ for } 1 \le i \le n-1 \\ \frac{1}{5n} + \frac{4}{5} & , \text{ for } i = n \end{cases}$$

Cost for algorithm at the i-th index:

$C(i) = i$

$$A(n) = \sum_{i=1}^{n} p(i) \cdot C(i) = \sum_{i=1}^{n-1} \frac{1}{5n} \cdot i + n \cdot \left(\frac{1}{5n} + \frac{4}{5} \right)$$

$$A(n) = \frac{1}{5n} \sum_{i=1}^{n-1} i + \frac{1}{5} + \frac{4n}{5}$$

$$= \frac{1}{5n} \frac{(n-1) \cdot n}{2} + \frac{4n}{5} + \frac{1}{5}$$

$$= \frac{(n-1)}{10} + \frac{4n}{5} + \frac{1}{5}$$

$$= \frac{n}{10} - \frac{1}{10} + \frac{4n}{5} + \frac{1}{5}$$

$$= \frac{9n}{10} + \frac{1}{10} \quad \Rightarrow \quad A(n) \in \Theta(n).$$

Proof: $\lim_{n\to\infty} \frac{A(n)}{n} = \lim_{n\to\infty} \frac{\frac{9n}{10} + \frac{1}{10}}{n}$

$$\overset{L'}{=} \lim_{n\to\infty} \frac{\frac{9}{10}}{1} = \frac{9}{10} \quad //$$