Author: Rafael

1

$$f(n) = O(g(n))$$
$$\text{By definition}$$
$$f(n) \leq c \cdot g(n) \text{ for any } n \geq n_0$$
$$g(n) = O(h(n))$$
$$g(n) \leq c' \cdot h(n) \text{ for any } n \geq n_0'$$
$$\text{Substitute } g(n)$$
$$f(n) \leq c \cdot c' h(n) \text{ for any } n \geq max(n_0, n_0')$$
$$c \cdot c' = c''$$
$$max(n_0, n_0') = n_0''$$
$$\text{Substitute new values}$$
$$f(n) \leq c'' h(n) \ for \ any \ n \geq n_0''$$
$$\text{By definition } f(n) = O(h(n))$$
$$QED$$

2

$$n^2 log^{10} n \leq n^{2.1} \ for \ any \ n \geq n_0$$
$$log^{10} n \leq n^{0.1}$$

Any polylog grows slower than any polynomial of positive power. Therefore

$$n^2 log^{10} n = O(n^{2.1})$$

3

$$2^{2n} \leq 2^n \ for \ any \ n \geq n_0$$
$$2^n \cdot 2^n \leq 2^n$$
$$2^n \leq 1$$

No growing function is less than 1 for sufficiently large n. Therefore

$$2^{2n} \neq O(2^n)$$

4

$$4^n \le 2^n \; for \; any \; n \ge n_0$$
$$2^{2n} \le 2^n$$
$$2^n \cdot 2^n \le 2^n$$
$$2^n \le 1$$

No growing function is less than 1 for sufficiently large n. Therefore

$$4^n \ne O(2^n)$$

Which also implies

$$4^n \ne \Theta(2^n)$$

5A

$$\lim_{n \to \infty} nlg(n^3)/nlgn$$
$$\lim_{n \to \infty} 3nlgn/nlgn$$
$$\lim_{n \to \infty} 3$$
$$= 3$$

$$\lim_{n \to \infty} nlgn/nlg(n^3)$$
$$\lim_{n \to \infty} ngln/3nlgn$$
$$\lim_{n \to \infty} 1/3$$
$$= 1/3$$

Both of the above limits approach a constant. Therefore

$$f(n) = \Theta(g(n))$$

5B

$$\lim_{n\to\infty} 2^{2n}/3^n$$
$$\lim_{n\to\infty} 4^n/3^n$$
$$\lim_{n\to\infty} (4/3)^n$$
$$= \infty$$

Therefore

$$f(n) \neq O(g(n))$$

Through the reversible property

$$f(n) = \Omega(g(n))$$

5C

$$f(n) = \sum_{i=1}^{n} lg\, i$$

$$g(n) = nlg\, n$$

Estimating the bound for the sum $f(n)$ can be done by multiplying the number of terms with the smallest and largest term, $n \cdot 0 \leq f(n) \leq n \cdot lg\, n$. Therefore

$$f(n) = O(g(n))$$

For a better lower bound, consider the second half of $f(n)$, $S = lg(n/2) + lg(n/2 + 1) \ldots lg(n)$. Using the same approach for the bounds of $f(n)$, $n \cdot lg(n/2) \leq S \leq n \cdot lg\, n$. Since $S < f(n)$, $n \cdot lg(n/2) \leq f(n) \leq n \cdot lg\, n$. Therefore

$$f(n) = \Theta(g(n))$$

6

$$
\begin{aligned}
f(n) &= \sum_{i=0}^{lg\,n} n/2^i \\
&= n \cdot 1(\frac{1 - \frac{1}{2}^{lg\,n}}{1 - \frac{1}{2}}) \\
&= n \cdot \frac{1 - n^{lg\,\frac{1}{2}}}{\frac{1}{2}} \\
&= 2n \cdot (1 - n^{-1}) \\
&= 2n - 2
\end{aligned}
$$

Therefore

$$f(n) = \Theta(n)$$

In the definition of $f(n)$, for sufficiently large n, I can ignore the ceiling function

7A

`A[i]` accumulates `min(max(A[:i+1], max(A[i:])) - A[i]` units

7B

Using 0-indexed arrays

```
Procedure TrappedWater(A)
    leftMaxs = []
    curMax = A[0]
    For each number in A do
        If number > curMax then
            curMax = number

        Append curMax to leftMaxs

    rightMaxs = []
    curMax = A[last index]
    For each number in reversed A do
        If number > curMax then
            curMax = number

        Append curMax to rightMaxs

    accumulated = 0
    For i from 0 to length of A do
        accumulated += min(leftMaxs[i], rightMaxs[i]) - A[i]

    Return accumulated
```

- Find the maximum number to the left and right of `A[i]` including itself, and store that in two arrays, ensuring `leftMaxs[i]` and `rightMaxs[i]` correspond to the appropriate value for `A[i]`
- Iterate over all three arrays
    - Find the minimum between `leftMaxs[i]` and `rightMaxs[i]` and subtract `A[i]`
    - Increment `accumulated` by the value calculated in the previous step
- `accumulated` stores how much water the terrain accumulated
- Important: including `A[i]` as a possible maximum to its left or right removes the need for countless other checks

7C

The code contains three loops that go from `[0, n-1]` and perform an `O(1)` operation for each iteration (finding all values for `leftMax`, `rightMax`, and then incrementing `accumulated`). Although there are `3n` `O(1)` operations, ignoring constants, this is still linear time, `O(n)`

#b2200 #homework #cs