CS 570: Analysis of Algorithms – H2

Submitted by: Indronil Bhattacharjee

Exercise 3.1-2

Let $c = 2^b$ and $n_0 \geq 2a$.

Then for all $n \geq n_0$, we have $(n+a)^b \leq (2n)^b = cn^b$

so $\mathbf{(n+a)^b = O(n^b)}$ …(1)

Now let $n_0 \geq \frac{-a}{1-(1/2)^{1/b}}$ and $c = \frac{1}{2}$.

Then $n \geq n0 \geq \frac{-a}{1-(1/2)^{1/b}}$

if and only if $n - \frac{n}{(2)^{1/b}} \geq -a$

if and only if $n+a \geq (\frac{1}{2})^{\frac{1}{b}}\, n$

if and only if $(n+a)^b \geq cn^b$.

Therefore $\mathbf{(n+a)^b = \Omega(n^b)}$ …(2)

Combining (1) and (2), $\mathbf{(n+a)^b = \Theta(n^b)}$**.**

Exercise 3.1-3

The statement "The running time of algorithm A is at least $O(n^2)$" is indeed meaningless because it conflates two different concepts: the lower bound of a function and the big O notation.

Big O Notation: Big O notation, O(f(n)) represents an upper bound on the growth rate of a function. It characterizes the worst-case behavior of an algorithm's running time. For example, if an algorithm's running time is O(n2), it means the running time grows no faster than a quadratic function of the input size n, up to a constant factor.

"At least": The phrase "at least" implies a lower bound, indicating the minimum growth rate of a function. However, big O notation does not represent lower bounds; it represents upper bounds. Instead, lower bounds are typically represented using big Omega notation, Ω(f(n)) or big Theta notation, Θ(f(n)).

Combining these two concepts leads to confusion and a nonsensical statement. Therefore, if we want to express that the running time of algorithm A has a lower bound, we should use appropriate notation for lower bounds, such as $\Omega(n^2)$ or $\Theta(n^2)$. So, the running time of algorithm A is at least $O(n^2)$ is meaningless.

Exercise 3.1-4

Let's analyze each statement separately:

1) $2^{n+1} = O(2^n)$: Correct

To determine whether 2^{n+1} is $O(2^n)$, we need to check if there exists a constant $c > 0$ and an n_0 such that $2^{n+1} \leq c \cdot 2^n$ for all $n \geq n_0$.

We can simplify 2^{n+1} as $2 \cdot 2^n$. Now, we need to find a c such that $2 \cdot 2^n \leq c \cdot 2^n$ for all n.

Since $2 \cdot 2^n = 2^n$ for all n, we can choose c=2. Then, for all $n \geq 0$, we have $2^{n+1} \leq 2 \cdot 2^n$.

Therefore, $2^{n+1} = O(2^n)$.

2) $2^{2n} = O(2^n)$: Not correct

To determine whether 2^{2n} is $O(2^n)$, we need to check if there exists a constant $c > 0$ and an n_0 such that $2^{2n} \leq c \cdot 2^n$ for all $n \geq n_0$.

Let's rewrite 2^{2n} as $(2^n)^2$. Now, we need to find a c such that $(2^{2n})^2 \leq c \cdot 2^n$ for all n.

Since $(2^n)^2 = 4^n$ and 4^n grows faster than 2n for all $n \geq 0$, we cannot find a constant c such that $(2^n)^2 \leq c \cdot 2^n$ for all n. Therefore, $2^{2n} \neq O(2^n)$.