**Chapter 3: Growth of Functions**

**Exercise 3.1 – 1**

**Let f(n) and g(n) be asymptotically nonnegative functions. Using the basic definition of -notation, prove that max(f(n), g(n)) = (f(n) + g(n)).**

* To show that max(f(n), g(n)) = (f(n) + g(n)), we need to show that there exists two constants c1 and c2 such that for n >= n0,

0 <= c1 \* (f(n) + g(n)) <= max(f(n), g(n)) <= c2 \* (f(n) + g(n))

The above conditions are satisfied when c2 >= 1 and when c1 <= 0.5.

* Therefore, max(f(n), g(n)) = (f(n) + g(n)).

**Exercise 3.1 – 2**

**Show that for any real constants a and b, where b > 0, (n + a)b = (nb).**

* (n + a)b = nb(1 + (a / n))b

But a / n 0 for some constant a and large values of n.

Therefore, nb(1 + (a / n))b nb

* Now, we have to show that there exists two constants c1 and c2 such that for all n >= n0,

0 <= c1 \* nb <= nb <= c2 \* nb

c1 = 1 and c2 >= 1 satisfy the above conditions.

**Exercise 3.1 – 3**

**Explain why the statement, “The running time of algorithm A is at least O(n2)”, is meaningless.**

* Let us represent running time of the algorithm A as T(n). It is given that T(n) >= O(n2). But by definition of O-notation, T(n) <= c \* n2.
* The two equations contradict each other, and hence the statement does not make sense.

**Exercise 3.1 – 4**

**Is 2n + 1 = O(2n)? Is 22n = O(2n)?**

* When we say 2n + 1 = 2 \* 2n = O(2n), we mean there exists a constant c such that for all n >= n0,

c \* 2n >= 2 \* 2n

For all the values of c > 1, the above equation holds, therefore 2n + 1 = O(n2).

* Similarly, we need to prove

c \* 2n >= 22n

No value of c can satisfy the above equation and therefore 22n des not belong to the set O(n2).

**Exercise 3.1 – 5**

**Prove Theorem 3.1.**

* Theorem 3.1 says that for any two functions f(n) and g(n), we have f(n) = (g(n)) if and only if f(n) = O(g(n)) and f(n) = Ω(g(n)).
* We know f(n) = (g(n)), it means there exists a constant c1 such that for n >= n0,

c1 \* g(n) <= f(n)

Therefore, by the definition of Ω-notation f(n) = Ω(g(n)).

* There also exists a constant c2 such that for n >= n0,

f(n) <= c2 \* g(n)

Therefore, by the definition of O-notation f(n) = O(g(n)).

**Exercise 3.1 – 6**

**Prove that running time of an algorithm is (g(n)) if and only if its worst-case running time is O(g(n)) and its best-case running time is Ω(g(n)).**

* Let the running time of the algorithm be represented as f(n).
* If its worst-case running time is O(g(n)), it means that for all n >= no there exists a constant c1 such that,

0 <= f(n) <= c1 \* g(n) – (1)

This means that for all possible inputs, running time of the algorithm (as a function) will always belong to the set (g(n)).

* Similarly, if its best-case running time is Ω(g(n)), it means that for all n >= no there exists a constant c2 such that,

0 <= c2 \* g(n) <= f(n) – (2)

This means that for all possible inputs, running time of the algorithm (as a function) will always belong to the set Ω(g(n)).

* From equations 1 and 2 it is clear that its running time is (g(n)).

**Exercise 3.1 – 7**

**Prove that o(g(n)) (g(n)) is the empty set.**

* By definition, f(n) = o(g(n)) if there exists a constant c such that for all n >= n0,

f(n) < c \* g(n)

Meaning, every function in the set o(g(n)) lie above function f(n).

* On the other hand, by definition, f(n) = (g(n)) if there exists a constant c such that for all n >= n0,

c \* g(n) < f(n)

Meaning, every function in the set (g(n)) lie below function f(n).

* Therefore the two sets do not contain a single common function.

**Exercise 3.1 – 8**

**We can extend out notation tot the case of two parameters n and m that can go to infinity independently at different rates. For a given function g(n, m), we denote by O(g(n, m)) the set of functions**

**O(g(n, m)) = f(n, m): there exists positive constants c, n0, and m0 such that 0 <= f(n, m) <= c \* g(n, m) for all n >= n0 or m <= m0 .**

**Give corresponding definitions for Ω(g(n, m)) and (g(n, m)).**

**Chapter 4: Divide-and-Conquer**

**Exercise 4.1 – 1**

**What does FIND-MAXIMUM-SUBARRAY return when all elements of A are negative?**

* When all the elements of A are negative, subroutine FIND-MAXIMUM-SUBARRAY returns a tuple containing index of the smallest element in the array and its value.

**Exercise 4.1 – 2**

**Write pseudocode for the brute-force method for solving the maximum-subarray problem. Your procedure should run in (n2) time.**

* FIND-MAXIMUM-SUBARRAY