Ch. 3: Growth Functions

● in Ch 2 we learned to find the exact run time of an algorithm (θ( )), but this is not usually worth the effort of computing

● for large enough inputs, the multiplicative constants and lower-order terms of an exact running time are dominated by the effects of the input size itself

− large enough inputs make only the order of growth of a running time relevant → studying asymptotic efficiency of algorithms; meaning, we are concerned with how the running time of an algorithm increases with the size of input in the limit, as the size of the input increases without bound

● usually, an algorithm that is asymptotically efficient will be that best choice for all but very small inputs

**3.1 Asymptotic notation**

**Asymptotic notation, functions, and running times**

● asymptotic notations are well-suited for characterizing running times not matter the input (meaning we don’t need to worry if the notation is describing the worst-case, average-case, or best case b/c the notation encompasses all of them)

**θ-notation**

● defining θ-notation**:** for a given function g(n), we denote by θ(g(n)) the set of functions

● for functions f(n) and g(n), where f(n) = θ(g(n)), we can say the g(n) is an asymptotically tight bound for f(n) if the following is true:

− for all values of n at and to the right of , the value of f(n) lies at or above and at or below

− Figure 3.1 (a) on Pg 45 describes the above



● the definition of θ(g(n)) requires that every member be asymptotically nonnegative, that is, that f(n) be nonnegative whenever n is sufficiently large. (An asymptotically positive function is one that is positive for all sufficiently large n.)

− therefore we will assume that every function used within θ- notation is asymptotically nonnegative

● derived by getting rid of all lower order terms and ignoring the constant

**O-notation**

● defining O-notation: for a given function g(n), we denote O9g(n)) the set of functions

● remember, θ-notation asymptotically bounds a function from above and below

● when we have only an asymptotically upper bound, we use O-notation to give an upper bound on a function, to within a constant factor

− for all values n at and to the right of , the value of the function f(n) is on or below cg(n)

− Figure 3.1 (b) on Pg 45 describes the above



● we write f(n) = O(g(n)) to indicate that a function f(n) is a member of the set O(g(n))

● note that f(n) = θ(g(n)) implies f(n) = O(g(n)), since θ-notation is a stronger notation than O-notation

− AKA:

● using O-notation, we can often describe the running time of an algorithm merely by inspecting the algorithm’s overall structure (what we did in Java II (Discrete Structures))

**Ω-notation**

● defining Ω-notation: for a given function g(n), we denote by Ω(g(n)) the set of functions

*there exist positive constants* c *and such that*

● Ω-notation provides an asymptotically lower bound

− for all values n at or to the right of , the value of f(n) is on or above cg(n)

− Figure 1.3 (c) on Pg 45 show the above



**Theorem 3.1**

● Theorem 3.1: For any two function f(n) and g(n), we have f(n) = θ(g(n)) if and only if f(n) = O(g(n)) and f(n) = Ω(g(n))

**Asymptotic notation in equations and inequalities**

● interpreting asymptotic notation (two ways they can appear):

1. the asymptotic notation stands alone on the right-hand side of an equation (or inequality)

♦ EX:

♦ means, we have already defined the equals sign (or inequality sign?) to mean set membership (EX: )

2. the asymptotic notation stands within a formula

♦ EX:

♦ means, we interpret the asymptotic notation as standing in for some anonymous function that we do not care to name (EX is interpreted as: , where f(n) is some function in the set θ(n); in this case, we let f(n) = 3n + 1, which indeed is in θ(n))

● note in Ch. 2 we expressed the worst-case running time of merge sort as the recurrence

T(n) = 2T(n/2) + θ(n)

If we are interested in only the asymptotic behavior of T(n), there’s no point in specifying all the lower-order terms exactly; they are all understood to be included in the anonymous function denoted by the term θ(n)

● the number of anonymous functions in an expression is understood to be equal to the number of times the asymptotic notation appears

− EX: , has one anonymous function (a function of i)

● when asymptotic equations appear on the left-hand side of the equation, we interpret the equation using the following rule: No matter how the anonymous functions are chosen on the left of the equal sign, there is always a way to choose the anonymous functions on the right of the equal sign to make the equation valid

− EX: , this means that for any function f(n) 𝜖 θ(n), there is some function g(n) 𝜖 θ( such that for all n

− the right-hand side provides more detail than the left

● we can a number of relationships and use the above rule to interpret them

− EX:

we can interpret the above as follows: the 1st equation says that there is some function f(n) 𝜖 θ(n) such that for all n; the 2nd equation says that for any g(n) 𝜖 θ(n), there is some function h(n) 𝜖 θ( such that for all n; note this interpretation implies , which is what the chaining of equations gives us

**o-notation (little-oh notation)**

● defining o-notation: for a given function g(n), we denote by o(g(n)) the set of functions

o(g(n)) = {f(n) : for any positive constant c > 0, there exists a constant such that for all }

● o-notation provides an upper bound that is not asymptotically tight

● the main difference of O-notation and o-notation:

− in f(n) = O(g(n)), the bound holds for some constant c > 0

− in f(n) = o(g(n)), the bound holds for all constants c > 0

● in o-notation, the function f(n) becomes insignificant relative to g(n) as n approaches infinity

**ω-notation**

● definition of ω-notation: for a given function g(n), we denote by ω (g(n)) the set of functions

ω(g(n)) = {f(n) : for any positive constant c > 0, there exists a constant such that for all }

● ω-notation provides a lower bound that is not asymptotically tight