1. Examples of sets O(…) and the definition of the set O(g(n))
2. Proofs of f(n) in O(g(n))
3. About function calls
4. Some terminology

**1. Examples of sets O(…) and the definition of the set O(g(n))**

Let f(n) be the number of basic steps made by executing an algorithm, depending on input value n. We want to classify such functions (and algorithms) depending on how f(n) behaves as n grows large —we don’t care about what happens when n is small. For example:

O(1) is the set of constant functions f(n).  
Example: f(n) = 5 is in O(1). Since O(1) is a set, we can write this as f(n) ∈ O(1).  
Example: f(n) = 800 is in O(1).

O(n) is the set of functions that are no larger than linear in n.  
Example: f(n) = n + 50 is in O(n).  
Example: f(n) = 5 is in O(n).  
Example: f(n) = n2 is *not* in O(n).

O(n2) is the set of functions that are no larger than quadratic in n.|  
Example: f(n) = n2 – 10n + 900 is in O(n2).  
Example: f(n) = 5 is in O(n2).  
Example: f(n) = n3 is *not* in O(n2).

Such a classification will allow us to compare, discuss, and choose algorithms for a particular task. For example, if we have two sorting algorithms, one taking time O(n2)[[1]](#footnote-1) and the other taking time O(n3), we would rather use the one taking time O(n2) if large arrays are to be sorted.

We need a general definition of the set of functions O(g(n)), where g(n) is itself a function. This definition of O(g(n)) is best understood using the graph to the right. The x-axis is n, starting at 0 and growing to the right. A function f(n) is shown in red, and a function c \* g(n) (for some positive constant c) is shown in green. For n < N, sometimes f(n) ≤ c \* g(n) and sometimes f(n) > c \* g(n). But for all integers n, N ≤ n, f(n) ≤ c \* g(n). Therefore, f(n) is in O(g(n)), which we write as f(n) ∈ O(g(n)).

n

N

c g(n)

f(n)

With the aid of this graph, we define the set O(g(n)) as follows:

**Definition**. Function f(n) is in set O(g(n)) iff there exist positive constants c and N such that for all n, n ≥ N, f(n) ≤ c \* g(n).

**2. Proofs of: f(n) ∈ O(g(n))**

We give three proofs of f(n) ∈ O(g(n)). Reading these proofs and developing your own proofs will give you more understanding for the definition of O(g(n)). In developing such proofs, we find it best to start with f(n) and change it using =, <, and ≤ steps into c \* g(n). As the proof progresses, some step may not be possible without deciding on what c or N should be, and the structure of the proof being developed will help us choose them.

Proof of f(n) = 30\*n + 6 ∈ O(n). Here, g(n) = n. We start with f(n) and end up with c\*g(n):

f(n)

= <Definition of f(n)>

30n + 6

≤ <Choose N = 6, so we consider only values of n that are >= 6. This allows us to replace 6 by n >

30n + n

= <Arithmetic>

31n

= <Chose c = 31, and use the definition of g(n)>

c \* g(n)

Proof of 6n2 + 6n - 5 ∈ O(n2). Here, f(n) = 6n2 + 6 n – 5 and g(n) = n2. We transform f(n) into c \* g(n).

6n2 + 6n + 5

< <Choose N = 3, because for n ≥ 3, 5 < n2>

6n2 + 6n + n2

< <For n ≥ 3, 6n < 6n2>

6n2 + 6n2 + n2

= <Arithmetic>

13n2

= <Choose c = 13, use definition of g(n)>

c \* g(n)

Proof of 8(log n) + 6 ∈ O(log n). This proof requires knowing a little bit about logarithms. In general, we will not require proofs that depend on knowledge of algorithms because not everyone has that knowledge. More important than that little bit of knowledge is the ability to develop these proofs. The proof below relies only on a basic property of logs to the base 2: If x = 2y, the log x = y.

8(log n) + 6

≤ <Choose N = 26 = 64, because for n ≥ 26, log n ≥ 6>

8(log n) + log n

= <Arithmetic>

9(log n)

= <Choose c = 9, use definition of g(n)>

c \* g(n)

**Some terminology**

Consider any method m, as shown in the box to the right. The number of basic steps taken by a call like m(arg) consists of the number of basic steps to (1) push a frame for the call onto the call stack, (1) assign the argument(s) to the parameter, (2) execute the method body, and finally (3) pop the frame from the stack and return the value. The number of basic steps taken for (1), (2), and (3) is bounded above by some constant, is in O(1). Only the time to evaluate the argument(s) and to execute the method body may not be in O(1). Therefore, when considering the time-complexity O(…) of a function call, we can

int m(int n) {

…

}

Because the number of basic steps is in set O(1), we say that the method is in O(1),

Obviously, a call on incr takes time O(1), which we can write as incr ∈ O(1). But then also incr ∈ O(n), incr ∈ O(n2). incr ∈ O(n3), etc.

1. “The algorithm taking time O(n2)” is an informal way of saying this: Let the number of basic steps executed by the algorithm be bounded above by f(n); then f(n) ∈ O(n2). [↑](#footnote-ref-1)