I1 Algorithm Analysis

I1.1 Order Notation

The order of a task measures how the time required for the task grows as the input grows. In this notation, the variable \( n \) always stands for the size of the input. We write that a task takes time \( O(f(n)) \) if it is at most proportional to \( f(n) \). This is said “O of \( f(n) \)” or “order \( f(n) \)”.

The order of a function is the simplest smallest function that it is \( O \) of. It ignores coefficients and everything except the dominant term. Terminology: Linear means proportional to \( n \).

Example: Some would say \( f(n) = 2n^2 + 3n + 1 \) is \( O(n^3) \) and \( O(n^2) \). But its order is \( n^2 \).

The most common case is that the time taken is proportional to the input. This is written \( O(n) \), also called linear time. For example, printing out the contents of an array takes time \( O(n) \).

The second most common case is that the time taken does not depend at all on the size of the input. This is called constant time and is written \( O(1) \). For example, accessing the first entry in the array, takes time \( O(1) \).

Sometimes the length of the task might vary depending on the actual data etc. The big-O notation only considers the worst case. For example, searching through an array for a particular value is considered \( O(n) \). If you’re lucky there will be a match on the first element; but if the value is not there, you will have to look at the entire array.

I1.2 Algorithm Analysis

The goal of algorithm analysis is to determine how the running time behaves as \( n \) gets large. The value \( n \) is usually the size of the structure or the number of elements it has. For example, traversing an array takes \( O(n) \) time.

We want to measure either time or space requirements of an algorithm. Time is the number of atomic operations executed. We cannot count everything; we just want
an estimate.

Long Arithmetic: Long addition of two $n$-digit numbers is linear. Long multiplication of two $n$-digit numbers is quadratic. (Check!)

For an example, consider primality testing.

```plaintext
function result isPrime(int N)
    for F=2:N-1
        if mod(N,F)==0
            result=false;
            return;
        end
    result=true;
end
```

This takes $O(\sqrt{N})$ time if the number is not prime, since then the smallest factor is at most $\sqrt{N}$. But if the number is prime, then it takes $O(N)$ time. And, if we write the input as a $B$-bit number, this is $O(2^{B/2})$ time. (Can one do better?)