Asymptotic Running Time

Manoj Gupta

August 7, 2016

1 An Example

Consider the following program that finds minimum element in an array$A[1 \dots n]$. In this lecture note, we will just concentrate on the running time of this program. In coming lectures, we will look at some strategies to prove the correctness of the program.

```
1 min ← A[1] ;                                  // Time taken = c_1
2 for i ← 2 to n do                             //Time taken = c_2(n-1)
3 |   if A[i] < min then                        //Time taken = c_3(n-1)
4 |   |   min ← A[i] ;                          // Time taken = c_4(n-1)
5 |   end
6 end
7 return min ;                                  // Time taken = c_5
8
```

Figure 1: FINDMIN($A[1 \dots n]$)

One can simply code the above procedure in his/her favorite language and run it on a machine to find its running time. However, this running time not only depends on the machine but also the current load on the machine. Therefore, we want to represent the running time in a way which is **machine independent**.

To this end, we will assume that each statement in the program can be executed in a finite or constant time. So, assume that the running time of the first statement is c_1. Similarly, we can give running time of other statement of the procedure. One always needs to remember an important point while giving these running times: *we need to find the worst case running time of each statement*.

For example, the execution of statement 4 in the above procedure depends on the truth value of **if** condition. However, we can come up with the following array $A = [n, n-1, n-2, \dots, 3, 2, 1]$ in which the **if** condition always evaluates to true. So, this statement is executed exactly $n - 1$ times (and we assume each such execution takes c_4 time).

Hence, the running time of the above algorithm is $c_1 + (c_2 + c_3 + c_4)(n - 1) + c_5$. To simplify the calculation, assume that $c_i = c$ for all $1 \leq i \leq 5$. So the running time is simplified to $(3n - 1)c$. There are two components in the running time:

1. The first terms tell us how the running time increases as n increases.
2. The second term is machine dependent.

Since we wanted a running time which is machine independent, we will drop the term c and state that the running time of the above algorithm is $3n - 1$.

2 Comparing Running Time of two Algorithms

Consider two algorithm A and B for the same problem with running time $f(n) = 2n^2 + 5$ and $g(n) = 50n + 5$ respectively. Some elementary calculation suggests the following:

1. $f(n) \leq g(n)$ when $n \leq 25$.

   The ratio $\frac{1}{c} \leq \frac{f(n)}{g(n)} \leq 1$ where $c \geq 0$ is some constant.

2. $f(n) > g(n)$ when $n > 25$.

   The ratio $\frac{g(n)}{f(n)}$ tends to 0 as n tends to infinity.

By Case(2), we conclude that for high value of n, $g(n)$ is *way lesser* than $f(n)$. However, by case (1), for small value of n, $g(n)$ is some constant factor away from $f(n)$. This is depicted in the following plot.

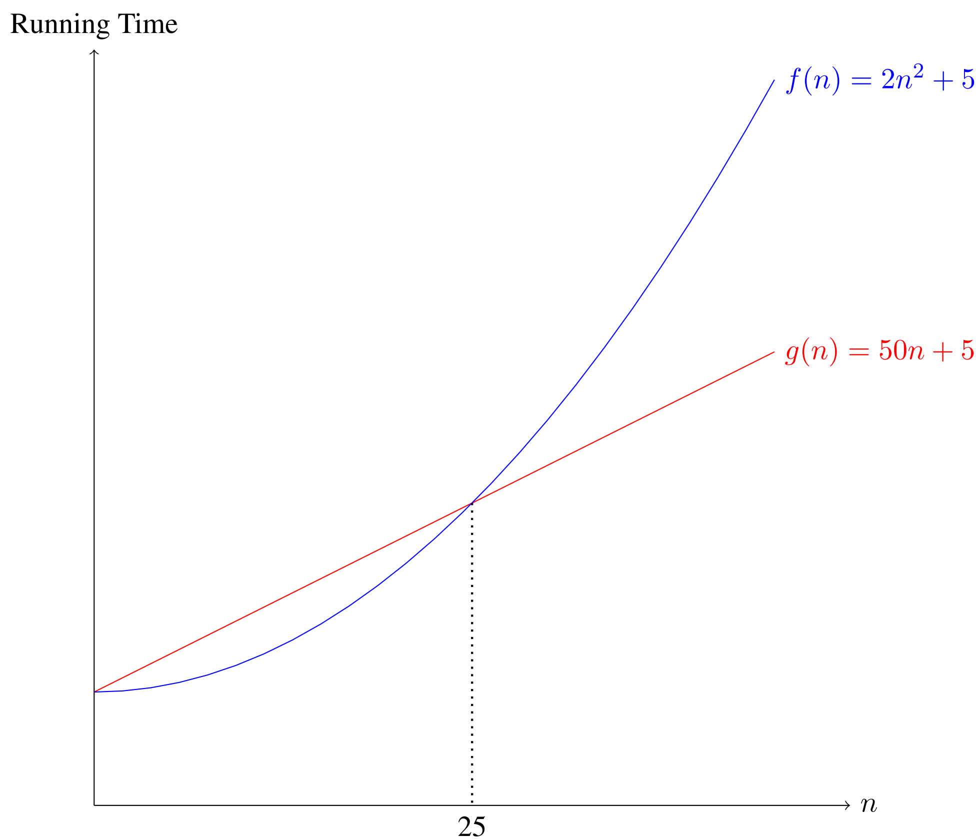

Figure 2: Pictorial depiction of $f(n)$ and $g(n)$

The following observations is of importance here:

Observation 2.1. *In the ratio $\frac{g(n)}{f(n)}$, the constant factors and the lower order term does not matter. Only the highest order term of $f(n)$ and $g(n)$ determines the fate of this ratio for high values of n.*

We will now put the above observation to use and give the following abstract notion of asymptoctics.

Definition 2.2. *Let $f(n)$ and $g(n)$ be two monotonically increasing function. Then $f(n)$ is order of $g(n)$ or $f(n) = O(g(n))$ if there exists a constant $c \geq 0$ such that for all $n \geq n_0$, $f(n) \leq c.g(n)$.*

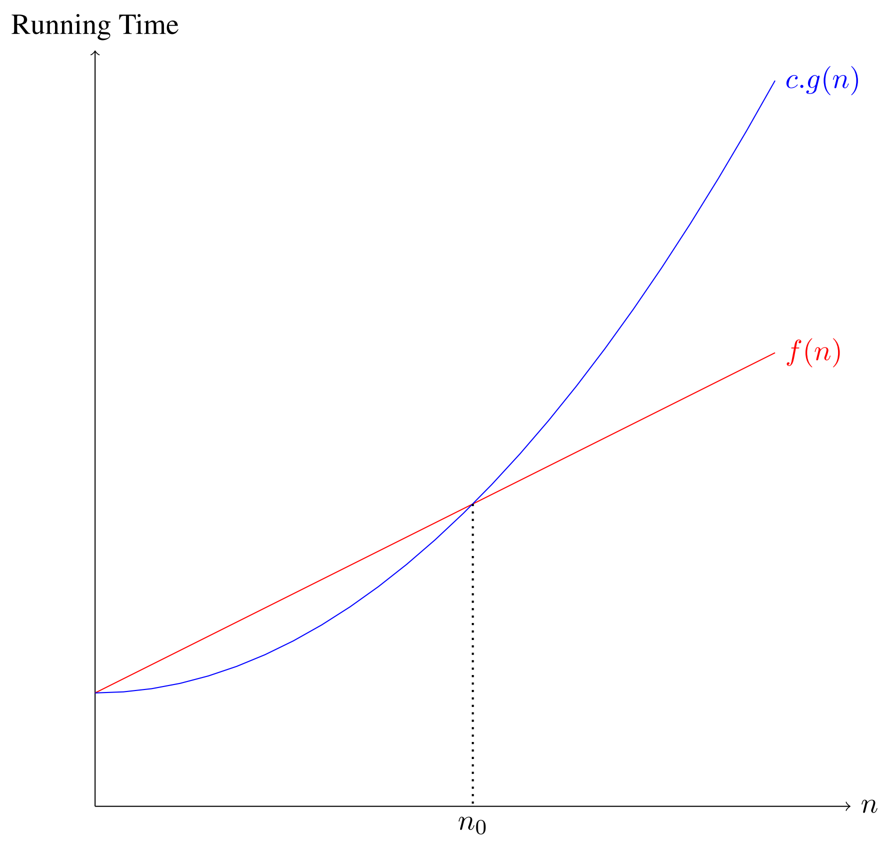

Figure 3: $f(n) = O(g(n))$

Some examples are in order.

1. $20n^2 = O(n^2)$
   $f(n) = 20n^2$, $g(n) = n^2$, $c = 20$ and $n_0 = 1$.
2. $20n^2 + 20n + 20 = O(n^2)$
   $f(n) = 20n^2 + 20n + 20$, $g(n) = n^2$, $c = 60$ and $n_0 = 1$.
3. $n^2 = O(n^3)$.
   One can show that $n^2 = O(n^2)$. However, by definition, the above equality also holds.
4. $20 = O(1)$
5. $50n \log n \neq O(n)$

A little trick that one can take from the above example is that if the highest order term of n in $f(n)$ is n^k, then $f(n) = O(n^k)$.

3 Theory Vs Practice

Consider the following functions $f(n) = 100n$ and $g(n) = n \log n$. One can calculate that the ratio $\frac{f(n)}{g(n)}$ tends to 0 as n tends to infinity. So, $f(n)$ beats $g(n)$ for higher values of n. However one can do the exact calculations as follows:

$$100n = n \log n \implies \log n = 100 \implies n = 2^{100}$$

That is, $f(n)$ beats $g(n)$ when $n > 2^{100}$. However, for all practical purposes, we can be sure that we will never get an input of such size. One should always understand the perils of trying to develop highly complex algorithms that have very high constants. Also, one should also understand the disadvantages of the big-O notation. The big-O notation disregards constants and in the above example we see that this strategy might not always be the best.