

An Introduction to Big-Oh Notation

Andrew Clausen

March 14, 2005

Big-Oh notation is widely used by computer scientists to concisely describe the behaviour of algorithms. An assertion such as “quicksort has an average case running time in $O(n \log n)$ but a worst case running time of $O(n^2)$ ” can succinctly summarize the performance characteristics of an algorithm.

In my experience, students find it rather difficult to learn, and many of my colleagues share common misunderstandings about it. This introduction to big-Oh notation aims to overcome these problems by carefully introducing each concept one step at a time, and giving “simple” and “tricky” examples along the way.

The first section gives one of many possible definitions of *running time* - the time it takes for an algorithm to perform a computation on a particular input. Section 2 describes three ways that running time can be aggregated across all inputs of a particular size. These allow the efficiency of an algorithm to be described by a single function that says how much time the algorithm requires for input of a given size. Section 3 presents a technique to compare these functions, so that the efficiency of different algorithms can be compared. This technique is used in section 4 to group together many algorithms that have similar performance characteristics. Section 5 explains how big-Oh analysis is used in practice. This introduction is summarized in section 6. Section 7 gives some review questions with solutions in section 8. Appendix A derives the average case running time of the linear search algorithm, which is a running example throughout this introduction.

1 Running Time

Consider the following standard C library function, `strlen()`, which returns the length of a string `str`:

```
/* Find the length of the string "str". */
int strlen(char *str)
{
    int i;
    for (i = 0; str[i]; i++)
        ;
    return i;
}
```

As the input string `str` gets longer, the time `strlen()` takes to compute its answer increases. Unfortunately, it is rather difficult to predict exactly how long it takes. This would depend on the type of computer, how many other programs are running, which compiler was used to compile the program, and so on. While these things can be important for designing algorithms, we are usually interested in the properties of algorithms that are independent of these implementation details. Thus, a simplifying assumption is in order.

To simplify, I am going to assume that each line of C code ending with a `;` takes the same amount of time to execute. Then to measure the running time of a computation, we only need to count the number of `;`'s visited, counting multiple visits multiple times. For example, the computation `strlen("hi")` would go through the following steps:

```

int i;
i = 0;
str[i];
i++;
str[i];
i++;
str[i];
return i;

```

This includes 8 steps, so the running time is 8. To do some mathematics with this, I am going to denote the running time of an algorithm A on input s as $\text{runtime}_A(s)$. For example,

$$\text{runtime}_{\text{strlen}}(\text{"hi"}) = 8.$$

The reader can verify that `find_a()` requires $4 + 2n$ steps for every string of length n . This can be expressed more formally. Let $X^n = \{s : s \text{ is a string of length } n\}$. Then for every $x \in X^n$,

$$\text{runtime}_{\text{strlen}}(x) = 4 + 2n.$$

Despite the simplification of counting `;`'s, this is still rather complicated for two reasons. First, $\text{runtime}(x)$ is defined in terms of strings rather than input sizes, so it does not relate running time to input size. This concern is addressed in the following section with worst case, best case and average case running time.

Second, $4 + 2n$ is a complicated description of the running time of `strlen()`. Later, $4 + 2n$ will be summarized by $O(n)$.

2 Best, Worst and Average Case

The goal of this section is to develop a way of describing how much the running times of algorithms increases as their inputs increase. In the algorithm from the previous section, `strlen()`, the running time was the same for every string of a particular length. Or, in formal mathematical language, for every $x, y \in X^n$,

$$\text{runtime}_{\text{strlen}}(x) = \text{runtime}_{\text{strlen}}(y).$$

Most algorithms do not have this property. For example, consider the function `find_a()` that finds the first occurrence of the letter `a` in a string. This function is a simple version of the standard C library function, `strchr()`. The underlying algorithm is often called *linear search*.

```

/* Find 'a' in the string str. */
char *find_a(char *str)
{
    int i;
    for (i = 0; str[i]; i++)
    {
        if (str[i] == 'a')
            return &str[i];
    }
    return NULL;
}

```

On input `"alibi"`, the running time is 4, whereas the running time is 19 for input `"never"`, even though the lengths of these two inputs are the same. This is because `'a'` occurs in `"alibi"` at the start, but does not occur in `"never"`.

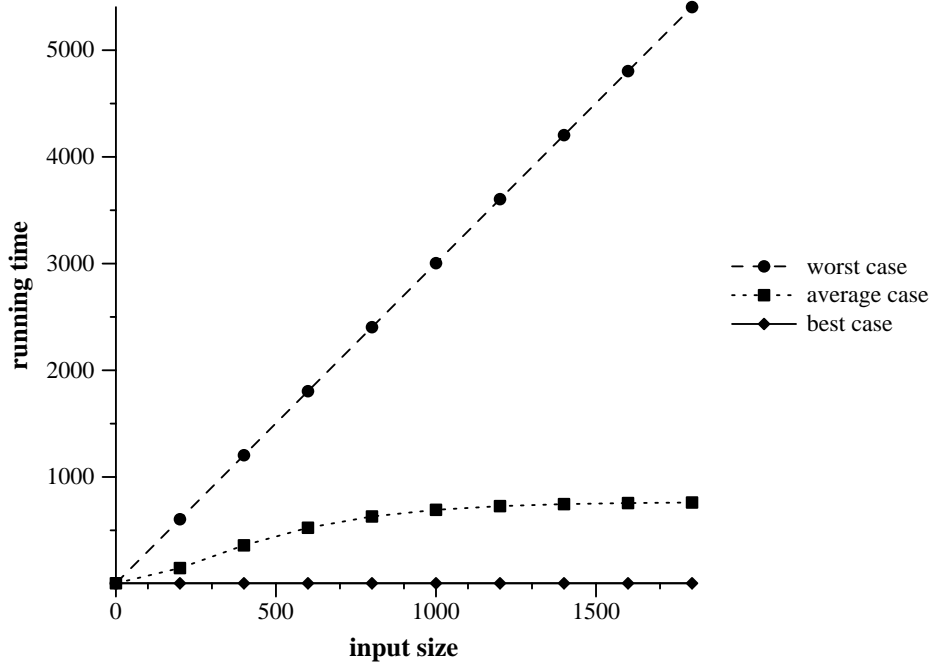


Figure 1: The worst, average and best case running times of `find_a()`. The average running time is always between the worst and best running times.

Thus, input size is not the only thing that matters: the particular input matters as well. But, input size is very important, and we would like to be able to describe how the performance of algorithms vary as the input size changes. Three solutions are available: worst case, average case, and best case running time. For algorithms such as `strlen()` which have the same running time for all inputs of a particular size, the three measures coincide. The three approaches are summarized in figure 1.

The worst case approach gives the worst (longest) running time for each input size. It is a pessimistic / conservative measure of an algorithm's performance. For example, the worst case running time of `find_a()` is $4 + 3n$. The worst case running time of an algorithm A is written in mathematical notation as

$$\text{worst}_A(n) = \max_{x \in X^n} \text{runtime}_A(x).$$

The function $\text{worst}_A : \mathbf{N} \rightarrow \mathbf{R}$ takes a single number in (input size) and gives a single number out (worst case running time).

The best case approach gives the best (shortest) running time for each input size. It is an optimistic measure of an algorithm's performance. For example, the best case running time of `find_a()` is 4. This is written in mathematical notation as

$$\text{best}_A(n) = \min_{x \in X^n} \text{runtime}_A(x).$$

Finally, the average case running time of an algorithm is the average running time of all inputs of a particular size. That is, given an input size, the average of all running times on all inputs of that size is taken. It is formally written as

$$\text{avg}_A(n) = \frac{1}{|X^n|} \sum_{x \in X^n} \text{runtime}_A(x).$$

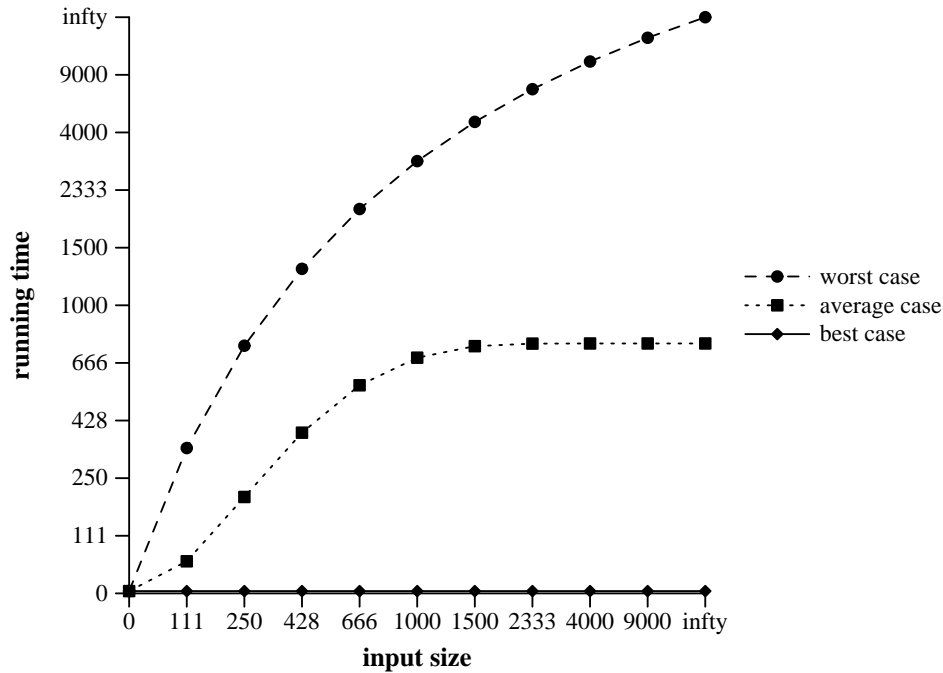


Figure 2: The worst, average and best case running times of `find_a()`, on a hyperbolic scale. This scale includes the entire positive real line $[0, \infty]$, and shows the asymptotic behaviour of the functions.

This is often rather tricky to figure out: the average case running time of `find_a()` is

$$\text{avg}_{\text{find}_a}(n) = 4 + 3 \left[254 - 254 \left(\frac{254}{255} \right)^n - n \left(\frac{254}{255} \right)^{n+1} \right].$$

Which brings us to the topic of the next two sections: How can we compare horrendous-looking running time formulas? And how can we simplify them without losing our ability to accurately compare them?

3 Asymptotic Comparison of Functions

The previous section described how the performance of algorithms can be summarized by a single function which relates input size to running time. Three approaches for constructing such a function were described: worst case, best case and average case. Unfortunately, these functions can be very complicated, and hard to interpret. In particular, ugly equations are difficult to compare, so it can be difficult to determine which algorithm is “faster”. This section develops a technique for comparing the behaviour of algorithms as the input size approaches infinity. This is sometimes called *asymptotic behaviour*.

Infinity is difficult to visualize. The graphs in this section use the hyperbolic scale, because it includes infinity on the end of the axes. For example, figure 2 is a hyperbolic version of figure 1, which shows the worst, average and best case running times of `find_a()`. Note that the hyperbolic-scale graph clearly shows that the average case running time never goes beyond 1000, even when the input size goes to infinity. The linear-scale graph implies this, but truncates the graph, leaving out the information that this section focuses on.

The following definition specifies a way of comparing the asymptotic behaviour of two functions $f(n)$ and $g(n)$. It says that f is asymptotically smaller to g if, as n increases, f gets larger more slowly than g .

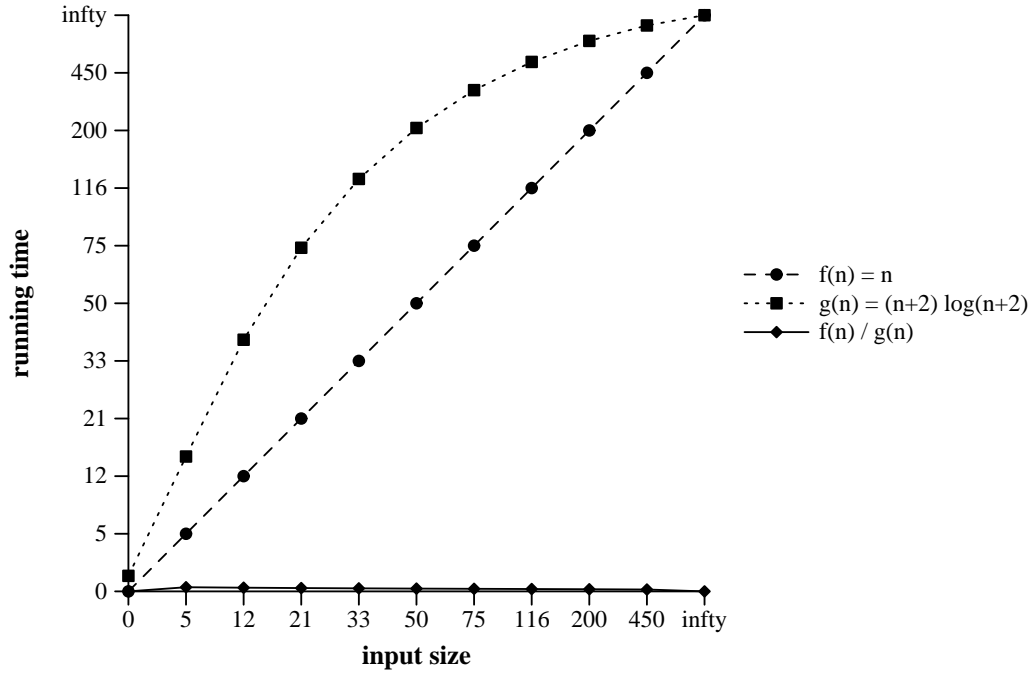


Figure 3: Since $f(n)/g(n) \rightarrow 0$, it is clear that $f \preceq g$.

Definition 1 (Asymptotically smaller or equal). Let $f, g : \mathbf{N} \rightarrow \mathbf{R}$ be functions. Then $f \preceq g$ if

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty.$$

For example, imagine the average case running time of one algorithm is $f(n) = n$, and the average case running time of another algorithm is $g(n) = (n+2) \log(n+2)$. These two functions are depicted in figure 3.

It is straightforward to verify that $f \preceq g$, as

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} &= \lim_{n \rightarrow \infty} \frac{n}{(n+2) \log(n+2)} \\ &= \lim_{n \rightarrow \infty} \frac{n+2}{(n+2) \log(n+2)} \\ &= \lim_{n \rightarrow \infty} \frac{1}{\log(n+2)} \\ &= 0. \end{aligned}$$

Since $0 < \infty$, f is asymptotically smaller or equal to g , and $f \preceq g$. Moreover, $g \not\preceq f$ because

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} &= \lim_{n \rightarrow \infty} \frac{(n+2) \log(n+2)}{n} \\ &= \lim_{n \rightarrow \infty} \log(n+2) \\ &= \infty. \end{aligned}$$

Since $f \preceq g$ but $g \not\preceq f$, we can write $f \prec g$. This means that f is asymptotically smaller, and therefore an algorithm with average case running time $f(n)$ is asymptotically faster (in the average case) to an algorithm with average case running time $g(n)$.

If $f \preceq g$ and $g \preceq f$, then this situation can be abbreviated to $f \sim g$. Clearly, it is always true that $f \sim f$.

The working example from the previous section was the function, $\text{find_a}()$. The section described the worst, best and average case running times of $\text{find_a}()$ as

$$\begin{aligned}\text{worst}_{\text{find_a}}(n) &= 4 + 3n, \\ \text{best}_{\text{find_a}}(n) &= 4, \\ \text{avg}_{\text{find_a}}(n) &= 4 + 3 \left[254 - 254 \left(\frac{254}{255} \right)^n - n \left(\frac{254}{255} \right)^{n+1} \right].\end{aligned}$$

So, how do these functions compare? It turns out that $\text{best}_{\text{find_a}} \sim \text{avg}_{\text{find_a}}$, but $\text{avg}_{\text{find_a}} \prec \text{worst}_{\text{find_a}}$. Verifying these facts is an exercise. The worked solution is presented later.

4 Big-Oh Complexity Classes

In the previous section, two very different looking functions, $\text{best}_{\text{find_a}}$ and $\text{avg}_{\text{find_a}}$ were described as being asymptotically equivalent. This begs the question, what other functions are these two equivalent to? This is what big-Oh notation is all about. Big-Oh groups functions that are similar to each other together.

Definition 2 (Big-Oh). *Let $f : \mathbf{N} \rightarrow \mathbf{R}$ be a function. Then*

- $O(f) = \{g : g \preceq f\}$. *This is the set of functions that are asymptotically smaller or equal to f .*
- $\Omega(f) = \{g : g \succeq f\}$. *This is the set of functions that are asymptotically greater or equal to f .*
- $\Theta(f) = \{g : g \sim f\} = O(f) \cap \Omega(f)$. *This is the set of functions that are asymptotically equivalent to f .*

The most widely used (in computer science) of these is $O(f)$, which is pronounced “big-Oh f ”. For example, if the average case running time of an algorithm is in $O(n^2)$, this means that its running time is asymptotically smaller (quicker) or equal to n^2 .

Most algorithms have worst and average case running times that are in one of $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$ or $O(2^n)$. Note that this is an increasing sequence of sets, with $O(1) \subset O(\log n) \subset \dots \subset O(2^n)$. The further to the left an algorithm’s worst and/or average case running times lies in to this sequence, the better its performance is.

In the running example, $\text{best}_{\text{find_a}} \in O(1)$, $\text{avg}_{\text{find_a}} \in O(1)$ but $\text{worst}_{\text{find_a}} \in O(n)$.

Big-Oh is a lazy summary of a function: for example, all of the functions considered so far are in $O(2^n)$. While it is correct to say that $n + 2 \in O(2^n)$, this is a poor summary of $n + 2$. Big-Theta is much more precise: $n + 2 \in \Theta(n)$, but $n + 2 \notin O(2^n)$. Good analyses of algorithms use big-Theta rather than big-Oh, but big-Oh is more widely used. Big-Omega is almost never used.

All of these statements are true:

- $n^k \in O(n^k)$.
- $n^2 \in O(n^3)$.
- $1/n \in O(1)$.
- $n^2 + n \in O(n^2)$.
- $2n + 3 \log n \in O(n)$.
- $\log n^8 \in O(\log n)$. (Note: $\log n^8 = 8 \log n$.)
- If $f, g \in O(h)$, then $f + g \in O(h)$.
- For all $a, b > 0$, $f \in \Theta(f)$ if and only if $af + b \in \Theta(f)$.

- $f \in \Theta(f)$ for all increasing functions, f .
- $f \in O(g)$ if and only if $g \in \Omega(f)$.
- $f \in \Theta(g)$ if and only if $f \in O(g)$ and $f \in \Omega(g)$.
- $f \in \Theta(g)$ if and only if $f \in O(g)$ and $g \in O(f)$.

WARNING: Most text books misleadingly write $f = O(g)$ rather than $f \in O(g)$. This is misleading because the equality sign indicates that the thing on the left (i.e. f) is the same as the thing on the right (i.e. $O(g)$). They are in fact very different things. Many students incorrectly apply algebraic tricks that would work if the left and right sides really were the same thing. For example, in this notation, it is valid to write $3n = O(n^2)$ and $3n = O(n \log n)$, which tempt many students to deduce $O(n^2) = O(n \log n)$. Therefore, I recommend you avoid this notation to avoid confusion.

5 Applying and Interpreting Big-Oh Notation

The previous sections explained the definitions and motivation for big-Oh notation. The running examples included in these explanations focused on explaining the concepts rather than demonstrating how computer scientists use big-Oh concepts in practical algorithm analysis. For example, computer scientists almost never count the exact number of times a semicolon is visited in a computation. This section explains how to apply and interpret big-Oh notation in practise, using an algorithm based on `for` loops as an example.

Consider this (inefficient) program that displays the athletic Olympic Games time records for each year. For each year, the best time to date is displayed.

```
float get_record(float times[], int year)
{
    float record = INFINITY;
    int i;
    for (i = 0; i <= year; i++)
    {
        if (times[i] < record)
            record = times[i];
    }
    return record;
}

void print_record_times(float times[], int year_count)
{
    int year;
    for (year = 0; year < year_count; year++)
        printf("%d: %f seconds\n", year, get_record(times, year));
}
```

The running time of this program varies with the size of the array, `times`, which is counted in the parameter `year_count`. For the remainder of this analysis, n refers to `year_count`. The running time of this program only depends on the input size n , regardless of the contents of `times`.¹ This means $\text{worst}(n) = \text{avg}(n) = \text{best}(n)$.

Because constants do not affect asymptotic behaviour, only the inner loops matter. For example,

$$2(n + 1) \log(3n + 4) \in \Theta(n \log n).$$

¹The running time of `printf()` might somewhat vary according to the size of the numbers to be displayed.

This means that the number lines of code inside or outside a loop is irrelevant: only the number of iterations of inner loops matter. In this example, the inner loop is the `for` loop inside `get_record()`. Each time `get_record()` is called, the instructions inside the `for` loop are visited `year` times. The outer loop calls `get_record()` `n` times. This means the inner loop is called

$$1 + 2 + \dots + n = \sum_{i=0}^n i$$

times in total. The following standard result describes exactly this situation:

Proposition 3 (Arithmetic Series). *For all $n \in \mathbf{N}$,*

$$\sum_{i=0}^n i = \frac{n(n+1)}{2}.$$

Therefore, the worst case running time, `worst(n)` of `print_time_records()` is in $\Theta(n^2)$.

Now, consider this alternative algorithm for solving the same problem:

```
void print_record_times_fast(float times[], int year_count)
{
    float record = INFINITY;
    int year;
    for (year = 0; year < year_count; year++)
    {
        if (times[i] < record)
            record = times[i];
        printf("%d: %f seconds\n", year, record);
    }
}
```

Here there is only one `for` loop, which goes through `n` iterations. Therefore, $\text{worst}(n) \in \Theta(n)$. Since $n \prec n^2$, the worst case (which equals the average and best case) running time of this algorithm is asymptotically smaller than the previous algorithm. For sufficiently large inputs, the second algorithm will be faster than the first algorithm. In principle, “sufficiently large” might be very large, so the asymptotic analysis is not strong evidence that the second algorithm is faster than the first one for practical applications. Therefore, asymptotic analysis is usually supplemented with experiments that test how large “sufficiently large” actually is.

6 Summary

The running time of most algorithms vary over different inputs. A natural way to compare algorithms is to compare how the running times increase as the input size increases. However, many algorithms’ running times are different for inputs of the same size. The worst case, average case and best case functions simplify comparisons by aggregating the various running times for a single input size down into a single number. The worst case function is particularly useful, because it gives an upper bound on running time.

These aggregated running time functions can be very complicated, and difficult to compare by inspection. Asymptotics provides some simple rules for comparing the behaviour of functions as their parameters approach infinity. Asymptotics also provides a theory of which functions grow in an equivalent way, and the rules for comparing functions also allow complicated functions to be simplified into simpler but equivalent functions. The worst and average case running times of most algorithms can be simplified in this way into one of 1 , $\log n$, n , $n \log n$, n^2 or 2^n . The sets of functions that are asymptotically smaller or equivalent to these are $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$ and $O(2^n)$ respectively.

Because the asymptotic behaviour of functions do not vary when constants are added in arbitrary locations in formulas, the number of steps inside a `for` loop or a function are irrelevant. Only the number of iterations of inner loops contribute to the asymptotic behaviour of running times.

Asymptotic running time functions only describe the performance of algorithms when input sizes get “sufficiently large”. Other evidence (usually experimental) is needed to verify if running times are acceptable before input sizes get “large”, and to verify that “sufficiently large” is not an unrealistically large number.

7 Questions

1. Prove that $n^2 \in O(2^n)$.
2. Prove that $\text{avg}_{\text{find}_a} \prec \text{worst}_{\text{find}_a}$.
3. Prove that $\text{best}_{\text{find}_a} \sim \text{avg}_{\text{find}_a}$.
4. Prove that if $f \in O(g)$ then $2f \in O(g)$.
5. What is the relationship between worst case running time and big-Oh?

8 Answers

1. *Prove that $n^2 \in O(2^n)$.* By the definition of $O(\cdot)$, $n^2 \in O(2^n)$ if $n^2 \preceq 2^n$. By the definition of \preceq , $n^2 \preceq 2^n$ if

$$\lim_{n \rightarrow \infty} \frac{n^2}{2^n} < \infty.$$

By repeated application of L'Hopital's rule,

$$\lim_{n \rightarrow \infty} \frac{n^2}{2^n} = \lim_{n \rightarrow \infty} \frac{2n}{\log 2 \cdot 2^n} = \lim_{n \rightarrow \infty} \frac{2}{(\log 2)^2 \cdot 2^n} = 0 < \infty,$$

as required.

2. *Prove that $\text{avg}_{\text{find}_a} \prec \text{worst}_{\text{find}_a}$.* Recall that

$$\text{worst}_{\text{find}_a}(n) = 4 + 3n,$$

$$\text{best}_{\text{find}_a}(n) = 4,$$

$$\text{avg}_{\text{find}_a}(n) = 4 + 3 \left[254 - 254 \left(\frac{254}{255} \right)^n - n \left(\frac{254}{255} \right)^{n+1} \right].$$

We need to show

$$\lim_{n \rightarrow \infty} \frac{4 + 3 \left[254 - 254 \left(\frac{254}{255} \right)^n - n \left(\frac{254}{255} \right)^{n+1} \right]}{4 + 3n} < \infty.$$

This follows with some simple algebra of limits,

$$\begin{aligned} & \lim_{n \rightarrow \infty} \frac{4 + 3 \left[254 - 254 \left(\frac{254}{255} \right)^n - n \left(\frac{254}{255} \right)^{n+1} \right]}{4 + 3n} \\ &= 4 + 3 \times 254 - \lim_{n \rightarrow \infty} \frac{254 \left(\frac{254}{255} \right)^n + n \left(\frac{254}{255} \right)^{n+1}}{4 + 3n} \\ &< 4 + 3 \times 254 \\ &< \infty. \end{aligned}$$

3. Prove that $\text{best}_{find_a} \sim \text{avg}_{find_a}$. Again, this follows with some simple algebra of limits,

$$\begin{aligned} & \lim_{n \rightarrow \infty} \frac{4}{4 + 3 \left[254 - 254 \left(\frac{254}{255} \right)^n - n \left(\frac{254}{255} \right)^{n+1} \right]} \\ &= \frac{4}{4 + 3 \times 254 - 3 \times \lim_{n \rightarrow \infty} \left[254 \left(\frac{254}{255} \right)^n + n \left(\frac{254}{255} \right)^{n+1} \right]} \\ &= \frac{4}{4 + 3 \times 254 - 3 \times 0} \\ &< \infty. \end{aligned}$$

4. Prove that if $f \in O(g)$ then $2f \in O(g)$. If $f \in O(g)$, then there is some c such that

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c < \infty.$$

Trivially,

$$\lim_{n \rightarrow \infty} \frac{2f(n)}{g(n)} = 2 \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 2c < \infty,$$

and $2f \in O(g)$.

5. What is the relationship between worst case running time and big-Oh? A common misconception is that worst case running time is somehow defined by big-Oh, and that best case is defined by big-Omega. There is no formal relationship like this. However, worst case and big-Oh are commonly used together, because they are both techniques for finding an upper bound on running time.

A The Average Case Running Time of `find_a`

The section on best/worst/average case running time asserted that the average case running time of `find_a` is

$$\text{avg}_{\text{find_a}}(n) = 4 + 3 \left[254 - 254 \left(\frac{254}{255} \right)^n - n \left(\frac{254}{255} \right)^{n+1} \right].$$

This appendix proves this claim.

This fact often comes as a surprise to computer scientists, because this formula is bounded, and is in $O(1)$. The average running time never goes above 3×254 , no matter how big the input is. The reason is that an 'a' is extremely likely to be found in the first 10000 or so characters. The probability that 'a' does not appear in the first 10000 characters is less than 0.0000000001, regardless of the input size. Therefore, input beyond 10000 characters is almost irrelevant.

The following formula for the geometric series is standard.

Proposition 4 (Geometric Series). *For all $c \in \mathbf{R} - \{0\}$ and all $n \in \mathbf{N}$,*

$$\sum_{i=0}^n c^i = \frac{1 - c^{n+1}}{1 - c}.$$

The following standard result can be derived from the geometric series formula with a little trickery and a lot of painful working. The casual reader can safely skip the proof of this lemma.

Proposition 5. *For all $c \in \mathbf{R} - \{0\}$ and all $n \in \mathbf{N}$,*

$$\sum_{i=0}^n ic^i = \frac{c}{1 - c} \left(\frac{1 - c^n}{1 - c} - nc^n \right).$$

Proof.

$$\sum_{i=0}^n ic^i = \sum_{i=0}^{n+1} ic^{i-1} - \sum_{i=0}^n c^i \tag{1}$$

$$= \frac{d}{dc} \left(\sum_{i=0}^{n+1} c^i \right) - \sum_{i=0}^n c^i \tag{2}$$

$$= \frac{d}{dc} \left(\frac{1 - c^{n+2}}{1 - c} \right) - \frac{1 - c^{n+1}}{1 - c} \tag{3}$$

$$= (n+2) \frac{c^{n+1}}{1 - c} - \frac{c^{n+2} - 1}{(1 - c)^2} - \frac{1 - c^{n+1}}{1 - c} \tag{4}$$

$$= \frac{1 - c^{n+2}}{(1 - c)^2} - \frac{(n+1)c^{n+1} + 1}{1 - c} \tag{5}$$

$$= \frac{1}{(1 - c)^2} [1 - c^{n+2} - (n+1)c^{n+1} - 1 + (n+1)c^{n+2} + c] \tag{6}$$

$$= \frac{1}{(1 - c)^2} [-(n+1)c^{n+1} + nc^{n+2} + c] \tag{7}$$

$$= \frac{c}{(1 - c)^2} [1 + nc^{n+1} - (n+1)c^n] \tag{8}$$

$$= \frac{c}{(1 - c)^2} [1 - c^n - nc^n(1 - c)] \tag{9}$$

$$= \frac{c}{1 - c} \left(\frac{1 - c^n}{1 - c} - nc^n \right). \tag{10}$$

□

Proposition 6. *The average case running time of `find_a` is*

$$\text{avg}_{\text{find}_a}(n) = 4 + 3 \left[254 - 254 \left(\frac{254}{255} \right)^n - n \left(\frac{254}{255} \right)^{n+1} \right].$$

Proof. The portion of the formula in the square brackets is the tricky bit: it represents the average number of times the body of the `for` loop must be executed. This average consists of the sum of the running times for all inputs of a size n , divided by the total number of inputs of size n ,

$$\text{avg}_{\text{find}_a}(n) = \frac{\sum_{x \in X^n} \text{runtime}_{\text{find}_a}(x)}{|X^n|}. \quad (11)$$

Let α be the size of the alphabet. That is, α represents the number of possible characters in each location in the input string. In the standard C setting, the possible characters are $1, \dots, 255$, so $\alpha = 255$. (0 is not possible, as it is the null-terminator.)

The denominator of the fraction is clearly, $|X^n| = \alpha^n$. The numerator is more complicated, with

$$\sum_{x \in X^n} \text{runtime}_{\text{find}_a}(x) = \sum_{i=0}^n i(\alpha - 1)^i \alpha^{n-i-1}. \quad (12)$$

Each term in the sum represents the total running time from all inputs which have their first 'a' in the i^{th} position.

Putting these together, the formula “simplifies” to

$$\text{avg}_{\text{find}_a}(n) = \frac{\sum_{i=0}^n i(\alpha - 1)^i \alpha^{n-i-1}}{\alpha^n} \quad (13)$$

$$= \sum_{i=0}^n i(\alpha - 1)^i \alpha^{-i-1} \quad (14)$$

$$= \frac{1}{\alpha} \sum_{i=0}^n i \left(\frac{\alpha - 1}{\alpha} \right)^i. \quad (15)$$

After setting $c = (\alpha - 1)/\alpha$, the sum is in the form of the previous lemma, and the result follows after much tedious cancellation. \square