Lab 2: Big-O Analysis

September 18, 2017

Note that this is an **individual** lab. However, you are encouraged to discuss the problems with your neighbors, and, of course, ask the TAs for help.

Contents

Objectives

By the end of this lab, you will:

- understand how to find the Big-O runtime of a function.
- gain an intuitive understanding of runtimes.
- be able to solve basic recurrences and establish runtimes from them.

Note: This lab is adapted from *Programming and Programming Languages (2016): Chapter 10, Predicting Growth.*

1 Introduction to runtime and Big-O

As you've learned from class, functionality is only a part of the overall picture when it comes to programming. One of the other major concerns we have when writing a program is its ***runtime*** – how long the program takes to run.

You can think about runtime in many ways, but in much of computer science, we focus on determining the ***worst-case runtime*** of a function. You can think of worst-case runtime as looking for the input that would cause a function to take as long as it possibly can, and measuring runtime accordingly. For example, with the `member` function, which determines if a value is an element of a list, the best possible scenario is that the element is the first value in the list, so the function only has to check one link before returning. However, the worst case – the case we're interested in – is that the element is not in the list, which requires that we check each element for equality before returning false.

We want to find an ***upper bound*** on the runtime. In most cases, we don't have enough information about a program to say something as specific as "Every time the size of the input to my function increases by one, the function takes an extra five milliseconds to run." Instead, we might be able to determine something a bit more general: "The runtime of the function varies linearly with the size of my input."

However, that's a bit of a mouthful. Instead, computer science has a handy notation for runtime, called Big-O, that we borrowed from mathematics. With it, we can rewrite the sentence above as:

$$f \in O([k \to k])$$

where k is the size of our input and f is the function we're describing.

By this point, you've already seen Big-O notation but a few things bear reiterating before we move on.

- The formal definition of Big-O states:

  $$f \in O(g) \text{ if and only if } \exists c \in \mathbb{R}^+, n_0 \in \mathbb{R}, \text{such that } \forall n \geq n_0 \in \mathbb{R}, f(n) \leq c \cdot g(n)$$

  In English, this reads, "f is an element of $O(g)$ if and only if there exists a c in the positive real numbers, and an n_0 in the real numbers, such that for all n greater than n_0, $f(n)$ is less than or equal to c times $g(n)$."

  What this is saying is that, after some point (n_0), f is always within a constant factor (c) of g, so the function g provides an estimate of an upper bound for f.

- We are only interested in the most reduced upper bound. For example, instead of saying that f is in $O(g)$ where $g(x) = 5x^2 + x$, we would say that f is in $O(g)$ where $g(x) = x^2$.

Task: Why can we disregard the 5 and x in the previous paragraph, and still have the statement f is in $O(g)$ be valid?

Before continuing, call over a TA to check that your answer is right.

2 Notation

When writing Big-O notation, there are a few common mistakes, such as writing something like $f = O(n)$.

This is a mistake because $O(n)$ is a *set of functions* bounded by n. Thus, it does not make mathematical sense to say that some function $f = O(n)$. Rather, since $O(n)$ is a set of functions, we should say that f is an element of the set, or $f \in O(n)$. This is read: f is in O of n.

Above, you also may have noticed the notation $[k \to k]$. This is shorthand for saying there is some generic function that maps from k to k, and is equivalent to saying $f(k) = k$. Thus, we can write $g \in O([k \to k])$ instead of the more verbose:

$$f(k) = k$$
$$g \in O(f)$$

Task: The statements below either have improper notation or are not in the simplest form possible. Fix the mistakes in the following expressions:

- $f \in O(w \to w^3)$
- $f = O([k \to k])$
- $f \in O([m \to 3m^3])$
- $f \in O(h^2)$
- $f \in O([k \to k^2 + k^3])$
- $f(k) = k \cdot \log k,\ g \in O(f)$

Before continuing, call over a TA to check that your answer is right.

Next, we'll explore three methods you can use to calculate the worst-case runtime of a function.

3 The Tabular Method

In the simplest case, given sizes for the arguments, we can simply examine the body of the function and add up the costs of the individual operations, including things like evaluating variables and constants. However, most interesting functions execute different pieces of code depending on their input, and many recur. Analyzing these functions requires special techniques. Here, we will assume there is one recursive call in the body of the function, and we will get to more general cases later (Section 5).

In this scenario, we have a handy technique we can use to deal with conditionals, such as **cases**, **ask**, or **if-else**. We will set up a table, with the same number of rows as the conditional has clauses. Each row will have seven(!) columns. This sounds daunting, but you'll soon see where they come from and why they're there.

We'll use the term **question** to refer to the predicate that is evaluated when determining whether the branch will execute. In an **if-else**, this is simply the condition, and in a **cases** statement, this is the implied type check (that determines, for example, whether a list is `empty` or a `link`). Similarly, the **answer** is the code that is evaluated if that branch of the conditional is executed.

For each row, fill in the columns as follows:

1. $|Q|$: the number of operations in the question
2. #Q: the number of times the question will execute
3. TotQ: the total cost of the question (multiply the previous two)
4. $|A|$: the number of operations in the answer
5. #A: the number of times the answer will execute
6. TotA: the total cost of the answer (multiply the previous two)
7. Total: add the two totals to obtain an answer for the clause (TotA + TotQ)

Finally, the total cost of the conditional expression is obtained by summing the Total column.

In the process of computing these costs, we may come across recursive calls in an answer expression. So long as there is only one recursive call in the entire answer, as we assumed earlier, we can ignore it.

Let's go through an example of this process, using the function `len`, and considering an arbitrary list input of size k:

```
fun len(l :: List<Any>) -> Number:
    doc: "Counts the number of items in l"
    cases (List) l:
        | empty => 0
        | link(_, r) => 1 + len(r)
    end
  end
```

First, note that the entire body of `len` is a conditional, so we can jump straight to creating the table. It's also worth noting that, as we mentioned, `len` only has one recursive call, so it's a valid use of this method. We begin by filling in the first row of the table, corresponding to the `empty` question.

1. The question costs three units (i.e. $|Q| = 3$); it takes one each to evaluate `is-empty` (you can think about the `cases` interally using `is-empty` to actually check this case) and `l`, and one to apply `is-empty` to `l`.
2. This is evaluated once per element in the list and once more when the list is empty, i.e., $k + 1$ times, so #Q $= k + 1$.
3. TotQ is thus $3(k + 1)$.
4. The answer takes one unit of time to compute ($|A| = 1$).
5. This answer is evaluated only once overall (when the list is empty) so #A = 1.
6. Thus, TotA is 1.
7. Adding TotQ and TotA, the total cost (or runtime) is $3k + 4$.

Task: Using the evaluation of the first row above as a guide, fill out the table for the second row of the **cases** statement, referring to the `link` case. Put your answer in to a table like the one below.

$\|Q\|$	#Q	TotQ	$\|A\|$	#A	TotA	Total
3	$k+1$	$3(k+1)$	1	1	1	$3k+4$

With the two rows of the table, compute the total cost of `len`. What is the total run-time of `len`, using Big-O notation?

Before continuing, call over a TA to check that your answer is right.

4 Big-O With Less Math

The tabular method is great for determining a program's runtime, but it has a few downsides. For the example in the last section, we got a total time of $12k + 4$, but $12k + 4$ is in $O([k \rightarrow k])$. Wouldn't it be nice to just look at a program and be able to know that it's linear?

While it's important to understand both the tabular method and the basis of Big-O, in practice, there is thankfully a more intuitive way to calculate Big-O runtimes. This is due to three basic facts about Big-O:

1. If a function $f \in O(F)$ is executed a *constant* number of times, the overall runtime is still in $O(F)$. For example, if the length function is in $O([k \to k])$, a function that runs length five times in a row would still be in $O([k \to k])$
2. If a function g is in $O(G)$ and a function f is in $O(F)$, running them back-to-back will yield a total runtime that's in $O(F+G)$. For example, if `remove-duplicates` is in $O([k \to k^2])$ and `length` is in $O([k \to k])$, a function that ran `length` and then `remove-duplicates` would have a runtime in $O([k \to k + k^2])$, which is equivalent to $O([k \to k^2])$
3. If a function $f \in O(F)$ makes calls to another function, $g \in O(G)$ at every one of its steps, then the overall runtime will be in $O(F \times G)$.

   For example, let's say we have two lists, a and b, that both have size k. We write a function, f to see whether every element in a is also in b. For every one of a's k elements, we run the `member` function on b. Since `member` is in $O([k \to k])$, and we're making k calls to it, the total runtime of this program is in $O([k \to k \cdot k]) = O([k \to k^2])$

When talking about Big-O complexity, we often use phrases like "linear", "polynomial", and "logarithmic." Here's a table containing more information about what these descriptions mean:

Phrase	Math	Explanation
Constant	$f(x) = 1$	The same amount of work, regardless of input size
Logarithmic	$f(x) = \log x$	Appears in functions where the recursive call is done on a fraction of the original input.
Linear	$f(x) = x$	Cost increases at the same rate as input size increases.
Linearithmic	$f(x) = x \log x$	Often appears when the number of recursive calls is logarithmic, but the total size of input at each level of recursion stays the same. Frequently seen in sorting algorithms.
Quadratic	$f(x) = x^2$	Often appears when comparing each element of a data structure to all other elements in that structure.
Exponential	$f(x) = c^x$	Increasing the size of the input by 1 causes the cost to increase by a factor of c.

Task: Below are functions and descriptions of functions. Determine their Big-O runtimes using the above facts.

1. A function is called with an input of size k. For each of these k elements it makes a call to another function, $g \in O([j \to \log j])$, passing an input of size k.
2. A function k calls a function $f \in O([n \to n])$ ten times and then calls a function $g \in O([m \to m])$, passing an input of size j to each function.

3. A function is called with an input of size k. For each of these k elements it makes a call to a function $g \in O([m \rightarrow m^2])$, passing an input of size k.

4. Consider the following code which deletes all entries from one list from another. Let k represent the size of `to-delete` and m represent the size of the list `l`.

```
fun delete<a>(to-delete :: List<a>, l :: List<a>) -> List<a>:
  cases (List) to-delete:
    |empty => l
    |link(f, r) => delete(r, l.remove(f))
  end
end
```

5. A function has two lists as inputs. Let k represent the size of the first list and m represent the size of the second list. It returns the sum of the length of the two lists.

6. A function is called with an input of size k. For each of these k elements, it makes a call to another function $g \in O([k \rightarrow k])$ passing an input of size k . For every element, the program also makes a call to a function $h \in O([k \rightarrow k^2])$ also passing an input of size k.

7. Consider the following code to remove duplicates from a list. Let the size of the input list be denoted by k.

```
fun rem-dups<a>(l :: List<a>) -> List<a>:
  cases(List) l:
    | empty => empty
    | link(f, r) =>
      link(f, rem-dups(rem-helper(r, f)))
  end
end

fun rem-helper<a>(l :: List<a>, cur :: a) -> List<a>:
  cases(List) l:
    | empty => empty
    | link(f, r) =>
      if (f == cur):
        rem-helper(r, cur)
      else:
        link(f, rem-helper(r, cur))
      end
  end
end
```

Before continuing, call over a TA to check that your answer is right.

5 Recurrences

Recurrences are a way of analyzing programs slightly more complex than the ones we saw in our tabular method. The basic idea is to define a function $T(k)$ that tells us how long our code takes to run on an input of size k.

Since Pyret uses recursion heavily, the time for a k-sized input will often depend on the time for a smaller input, like $k-1$ or $k/2$. Thus, we may end up in a state where $T(k)$ depends on $T(k-1)$, which in turn depends on $T(k-2)$, which in turn depends on $T(k-3)$ and so on. At first glance, this might seem like our equation will never end, but luckily for us, we have base cases! If our program is written correctly, we will eventually reach one, which will end the recursion. Therefore, our function will look like:

$$T(k) = \begin{cases} \text{something that depends on } T(\text{something smaller than k}), & k > 0 \\ \text{a constant}, & k = 0 \end{cases}$$

(**Note:** Since k corresponds to the size of our input, we can assume it's not negative.)

Our goal, then, is to simplify this into a closed form of T (one that doesn't refer to itself) so that we can determine the Big-O runtime. We will illustrate this method with an example.

Going back to our analysis of `len`, we would define our runtime T in two parts: the value of $T(0)$ (when the list is empty) and the value for non-zero values of k. We know that $T(0) = 4$ (the cost of the first conditional and its corresponding answer). If the list is non-empty, the cost is $T(k) = 3 + 3 + 6 + T(k-1)$ (respectively from the first question, the second question, the remaining operations in the second answer, and the recursive call on a list one element smaller). This gives the following recurrence:

$$T(k) = \begin{cases} 4, & \text{when } k = 0 \\ 12 + T(k-1), & \text{when } k > 0 \end{cases}$$

For an input of size $p \geq 0$, this would take 12 steps for the first element, 12 steps for the second, and so on, until we run out of list elements and need 4 more steps: a total of $12p + 4$ steps. Notice this is precisely the same answer we obtained by the tabular method! If we wrote this out, we would obtain something like:

$$\begin{aligned} T(p) &= 12 + T(p-1) \\ &= 12 + (12 + T(p-2)) \\ &\dots \\ &= 12 + (12 + (12 + \cdots + (12 + T(1)))) \\ &= 12 + (12 + (12 + \cdots + (12 + (12 + T(0))))) \\ &= 12p + 4 \end{aligned}$$

5.1 Solving Recurrences

Earlier we saw recurrences that had two cases: one for the empty input and one for all others. In general, we should expect to find roughly one per cases clause.

For example, we may have a recurrence of the form:

$$T(k) = \begin{cases} T(k-1)+k, & k>0 \\ c_0, & k=0 \end{cases}$$

(**Note:** c_0 represents a constant.)

In order to solve such equations, it is generally easiest to write out a few steps of the recurrence until a pattern becomes clear. In this case we get:

$$\begin{aligned} T(k) &= T(k-1)+k \\ &= (T(k-2)+(k-1))+k \\ &= (T(k-3)+(k-2))+(k-1)+k \\ &\ \ \vdots \\ &= T(0)+(k-(k-1))+(k-(k-2))+\cdots+(k-2)+(k-1)+k \\ &= c_0+1+2+\cdots+(k-2)+(k-1)+k \end{aligned}$$

We can now use the fact that $1+2+\ldots+n = \frac{n(n+1)}{2}$ to simplify the above line to $c_0 + \frac{k(k+1)}{2}$, allowing us to determine that $T \in O([k \rightarrow k^2])$.

Task: Find the closed form of these recurrence relations:

- $T(k) = \begin{cases} c_0, & k \leq 1 \\ T(k-2)+1, & k>1 \end{cases}$
- $T(k) = \begin{cases} c_0, & k=0 \\ 2T(k-1)+1, & k>0 \end{cases}$

Now, find the recurrence function and closed form for `delete`, which we encountered earlier. It is duplicated below. You can use c_0 and c_1 to represent constants.

```
fun delete<a>(to-delete :: List<a>, l :: List<a>) -> List<a>:
  cases (List) to-delete:
    |empty => l
    |link(f, r) => delete(r, l.remove(f))
  end
end
```

If you're looking for an extra challenge, feel free to try the following recurrence relations. However, they are not required to check off this lab.

- $T(k) = \begin{cases} c_0, & k = 0 \\ T(k/2) + 1, & k > 0 \end{cases}$
- $T(k) = \begin{cases} c_0, & k = 0 \\ T(k/2) + k, & k > 0 \end{cases}$
- $T(k) = \begin{cases} c_0, & k = 0 \\ 2T(k/2) + k, & k > 0 \end{cases}$

Once a TA signs off, you've finished the lab – congratulations!

6 Just for Fun

6.1 Average Case Runtime

In some cases, we may be interested in analyzing the runtime of a function based off of a "typical" input, as opposed to the worst-possible input. For example, you may be writing a sorting function, and you expect your input to be somewhat sorted; that is, you expect that no element will be more than k indices away from where it should be in the sorted array. Using this information, we could construct an "average-case" analysis, which wouldn't bound the runtime of our sorting function, but it would provide useful information about the typical case you might use it for. For some examples of this, take a look at this Wikipedia page, which details some sorting algorithms, along with their average and worst case runtimes.

Specifically, `quicksort` is a sorting algorithm which has a $O([k \rightarrow k \log k])$ average case runtime, but runs $O([k \rightarrow k^2])$ in the worst case. It works by randomly picking a pivot element, and then dividing the remaining elements into two lists: those smaller and those larger than the pivot. It then recurs on, and later merges, these two lists. Over all the recursive calls, the *average* pivot will like near the median of the list. However, in the extremely unlikely case that the pivot is always the largest or smallest element, the sort will take far longer, as each recursive call is sorting only one fewer element.

6.2 Best Case Runtime

Similarly, you may be interested in the best possible runtime. It is the least common analysis, but in some applications, you may want to optimize for the minimum time your function takes to run. For example, the `member` function has constant best-case runtime, as if the desired element is the first element in the list, we will terminate after only having examined one thing.

6.3 Talking About Big-O

In this lab, we've introduced some complex notation and terminology related to runtime and Big-O analysis. While these are the mathematically rigorous ways of talking about this topic, it's rarely the phrasing or notation you'll hear or see in day-to-day conversation. Instead, these are all examples of common ways of talking about Big-O, which you will likely need to use for other classes:

- $O(n^2)$, instead of $O([n \rightarrow n^2])$.
- "f is a linear function"
- "f is $n \log n$ with respect to the size of its input"
- "f is $O(n)$, instead of f is in $O(n)$"