Complexity

15-110: Principles of Computing

Giselle Reis

When writing programs, specially if one is dealing with a big amount of data, it is important to take care how *fast* the program runs. A small modification in an algorithm may bring execution time from hours to seconds. In computer science, the area that studies algorithm efficiency is called complexity theory. We will look at a part of it which allows us to reason about the efficiency of the programs we write.

Computation steps

The efficiency of a program is measured by counting the *number of steps* it takes to reach a solution. Each atomic computation operation counts as one step. These include:

- Assignment (e.g. `L = []`)
- Arithmetic operations (e.g. `x + y`)
- Boolean comparisons (e.g. `c == 't'`)
- Indexing (e.g. `L[i]`)

Take for example the following function that finds if an element `e` is in a list `L`:

```
def isIn(e, L):
    for x in L:
        if e == x:
            return True
    return False
```

If we call this function using `isIn(1, [1,2,3])`, how many steps does it take? Since the element we are looking for is at the first position in the list, this function will take 2 steps: get the first element in `[1,2,3]` (indexing) and compare it with `1` (comparison). What if the function is called using `isIn(1, [9,8,7,6,5,4,3,2,1])`? Then for each element the program will fetch this element from the list to start one `for` iteration (indexing), and

compare the fetched element with 1. Since it will only return at the last element, it will perform both operations for all numbers and thus 20 steps in total. What if we call `isIn(1, [9,8,7,6,5,4,3,2,0])`?

Input size

As you can see, the number of steps taken by a function can greatly vary depending on the input given. That is why the complexity of algorithms is measured as a function of the input size. If a function has more than one input (as our `isIn` function from before), or if there are different ways to measure the size of the input (e.g. if the input is an integer, we can think of its size as the quantity represented by that integer or its number of digits), we will choose the size or combination of sizes that affect the number of steps taken by the program.

In the function above, the size of the first input (whatever type it is) does not really affect the number of steps taken by the program: it is only used in one comparison which takes one step independently of the sizes of the operands. On the other hand, the size of the list determines how many times the loop will run (how many indexing and comparison operations). Therefore, the number of steps of the `isIn` function on the size of the list (the second parameter). Assuming that the size of the list is n, the number of steps of `isIn` is some function $f(n)$.

Input case

Even if we have parametrized the number of steps of a function by the size of the input, the way the input is arranged also plays a role on this measure. Compare for example the number of steps taken by `isIn` for the inputs `(1, [1,2,3,4,5])` and `(1, [5,4,3,2,1])`. Although both lists have size 5, the function will take 2 steps on the first input and 10 steps on the second one. How can we find one function $f(n)$ if there are two possible values for $f(5)$?

Well, we can't.

That is why complexity measures are sometimes divided into: worst case, average case and best case analysis.

The best case is the one where the function will take the least possible number of steps. For our example function, these are the cases where the searched element is the first on the list, and the function takes only 2 steps independently of the input size, so $f(n) = 2$. Best cases are usually easy to find, but they are not very informative. If I tell you, this program will finish in at least 0.5 second, it might take 0.5 second, 3 seconds or one hour.

The worst case is the one where the function will take the greatest possible number of steps. For our example function, these are the cases where the searched element is the last on the list, or not in the list at all. In those cases the function will take 2 steps *for each* element in the list (indexing and

comparison). So for a list of size n, the program takes $2n$ steps, therefore $f(n) = 2n$. When we talk about complexity, we will always talk about the worst case scenario. By being as pessimistic as possible, we are safe that the program will not take longer to run, and if it runs faster, all the better.

The average case is, well, the average case. Finding out the function for this case involves looking into the distribution of the possible inputs, which might be very complicated. We will not look into average cases in this course.

Exercises

For each of the functions below, find the number of steps they take as a function of input size (make sure to specify what is the input size).

```
def sum(n):
    s = 0
    for i in range(n):
        s += (i+1)
    return s

def sum2(n):
    return int(n * (1+n) / 2)

def sumMtoN(m, n):
    sum = 0
    for i in range(m, n+1):
        sum += i # Same as sum = sum + i
    return sum
```

You can consider that each print in the code below takes one step.

```
def boardDiagonal(n):
    for i in range(n):
        for j in range(n):
            if i == j:
                print("O", end=" ")
            else:
                print("*", end=" ")
        print()
    return
```

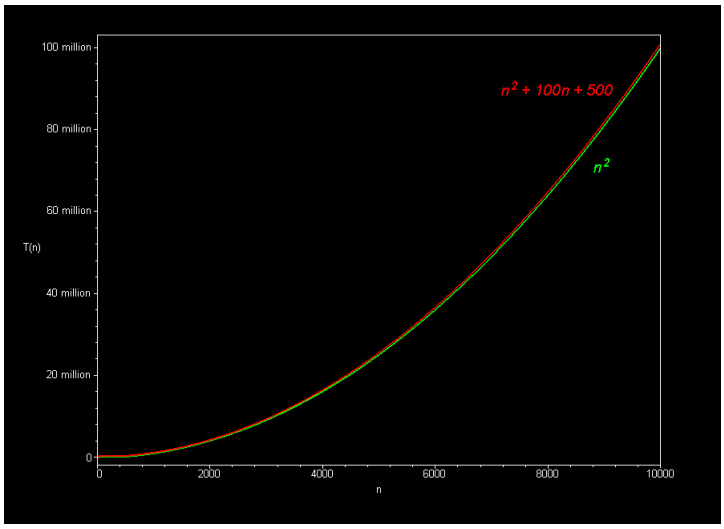
Big-O

In actuality, we are interested in how fast the running time increases with the increase of the input. If you remember your math well, this corresponds to how *steep* the curve of a function is. And steepness, when considering

really big values, is determined basically by the *dominant term* of a function. For example, take the two functions:

$$\begin{aligned} f(n) &= n^2 + 100n + 500 \\ g(n) &= n^2 \end{aligned}$$

Even though it might seem that f is worse than g (in the sense that it will grow faster), for sufficiently large values they are almost the same:

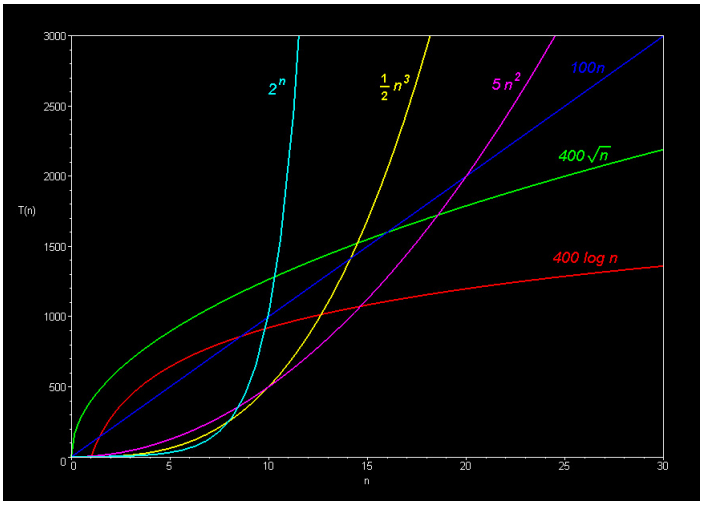


The big-O notation describes functions' *asymptotic behavior* (i.e. how fast it grows, or how steep its curve it). For the functions above, we say that f and g are $O(n^2)$ (read "big oh of n squared" – or quadratic). Typically, the complexity of algorithms will fall in one of these common complexities (in decreasing speed):

- $O(1)$: constant (e.g. if $f(n) = 4$ then f is $O(1)$). This is the fastest you can get.
- $O(\log n)$: logarithmic
- $O(\sqrt{n})$: square root
- $O(n)$: linear (e.g. if $f(n) = 10n + 2$ then f is $O(n)$). This is generally still considered good.
- $O(n \log n)$: linearithmic, loglinear or quasilinear
- $O(n^2)$: quadratic
- $O(n^3)$: cubic
- $O(n^k)$: polynomial

- $O(k^n)$: exponential (e.g. if $f(n) = 2^{4n}$ then f is $O(2^n)$). This is generally considered slow (although we *can* get much worse!).
- $O(n!)$: factorial. Algorithms falling in this category cannot usually be used in practice for really big n.

By comparing the asymptotic behavior of two programs, we can identify which one behaves best when the input size is increased. The following graph gives us an idea of how different functions compare:



Built-in functions

When using python's built-in functions, it is good to have an idea of their complexity to find out how they are impacting your program. Here's the complexity of a few popular ones:

Function	Input size	Complexity
`L.append(x)`	–	$O(1)$
`L.sort()`	length of `L`	$O(n \log n)$
`min(L)`	length of `L`	$O(n)$
`max(L)`	length of `L`	$O(n)$
`len(L)`	–	$O(1)$