Thinking about algorithm complexity

and estimating it

Sukrit Gupta and Nitin Auluck

February 25, 2024

Outline

Acknowledgement and disclaimer

All mistakes (if any) are mine.
I have used several other sources which I have referred to in the appropriate places.

Section 1

Different use cases, different priorities

Intuition

- Different problems have different requirements from the solutions. You have different set of performance measures to determine the efficacy of the solution.
- Besides producing results that can be relied upon, it is important that we cater to the performance metrics that are specific to the situation at hand.
- The most straightforward solution is often not the most efficient. Computationally efficient algorithms often employ subtle tricks that can make them difficult to understand.

Some applications

- A program that warns airplanes of potential obstructions needs to issue the warning before the obstructions are encountered.
- For stock exchanges, it is necessary to maintain low latency in the transactions.
- Fuel injectors in our automobiles for injecting (approximately) appropriate amounts of fuel at the right time.

Some more applications

- For missile systems, it is important to hit the targets precisely (and timely).
- For space shuttles, it is important to get into the right orbit, but the time they take to do it may not be of much consequence.
- For predictions from medical imaging models we are more concerned about their correctness and not about the time taken to make the prediction.

However, ...

While time may not be critical for some applications, it is always desirable to have algorithms that are fast.

Linear search again.

```
def linearSearch(L, x):
    '''
    L is a list
    x is what you wish to search
    '''
    for e in L:
        if e == x:
            return True
    return False
```

If L is a million elements long and consider the call linearSearch(L, 3).

- Best case: If the first element in L is 3, linearSearch will return True almost immediately.
- Worst case: On the other hand, if 3 is not in L, linearSearch will have to examine all one million elements before returning False.
- Average case? Average running time over all possible inputs.

Murphy's Law

All engineers share a common article of faith, Murphy's Law: "*If something can go wrong, it will go wrong.*"

The worst-case provides an upper bound on the running time. This is critical in situations where there is a time constraint on how long a computation can take.

For example: it is not good enough to know that "most of the time" the air traffic control system warns of impending collisions before they occur.

Section 2

Breaking down the run time

How many milliseconds will this take to run?

```
def fact(n):
    """Assumes n is a natural number
    Returns n!"""
    answer = 1
    while n > 1:
        answer *= n
        n -= 1
    return answer
```

We could run the program on some input and time it.

Timing the program wouldn't be particularly informative because the result would depend upon:

- the speed of the computer on which it is run;
- the efficiency of the Python implementation on that machine; and
- the value of the input.

We will look into the third issue later, but for the first two issues we use a more abstract measure of time.

Instead of measuring time in milliseconds, we measure time in terms of the number of basic steps executed by the program.

What is a basic computational step

A step is an operation that takes a fixed amount of time, such as:

- binding a variable to an object;
- making a comparison;
- executing an arithmetic operation; or
- accessing an object in memory.

Constant time steps

```
def fact(n):
    """Assumes n is a natural number
    Returns n!"""
    answer = 1
    while n > 1:
        answer *= n
        n -= 1
    return answer
```

What are the number of constant steps? 2

What are the number of steps in the loop? 5 (1 comparison, 2 assignment, 2 operations)

Considering additive constants

- Total = 2 (constant time steps) + $5(n-1)$ (times the loop will be run) + 1 (conditional statement for the while loop) = $5n-2$
- So, for example, if n is 1000, the function will execute roughly 4998 steps. If n is 2000, function will execute roughly 9998 steps.
- As n gets large, the difference between $5n$ and $5n-2$ becomes insignificant. For this reason, we typically ignore additive constants when reasoning about running time.
- Multiplicative constants are more problematic.

Considering multiplicative constants

- Do you remember linear search and binary search?
- Let there be k steps in the loop in Linear search. For a list of size n, what are the approximate number of steps in linear search?
- Let there be l steps in the loop in binary search. For a list of size n, what are the approximate number of steps in binary search?
- Let's compare the worst case when there are 1 million (1,000,000) elements to search from.
- How many steps for linear search?
- How many steps for binary search?
- The linear search will take $k \times 10^6$ steps. The binary search will take $l \times 20$ steps.
- When the difference in the number of iterations is this large, it doesn't really matter how many instructions are in the loop, i.e., the multiplicative constants are irrelevant.

Section 3

Asymptotic Notation

Imagine ...

- For the last code snippet, we started from the actual $5n - 2$ number of steps, and simplified it to just n steps.
- Any algorithm can run efficiently for small inputs. However, efficiency related issues arise when the algorithm is run on very large inputs.
- In general, when you think about algorithm efficiency, imagining that the size of the input goes to ∞ and then see what factor really matters.

Asymptotic notation

- The *asymptotic notation* provides a formal way to talk about the relationship between the running time of an algorithm and the size of its inputs.
- As a proxy for "very large", asymptotic notation describes the complexity of an algorithm as the size of its inputs approaches infinity.
- Let's develop some intuition.

Intuition behind asymptotic notation

Let's assume that the number of steps in some code snippet are given by: $n^2 + 5n + 1000$

- For n = 10, total number of steps?
- 1150. The third term contributes to 87% of the steps, the second term contributes 4.3% and the first term 8.7%.
- For n = 1000, total number of steps? 1006000
- The third term contributes to 0.1% of the steps, the second term contributes 0.5% and the first term 99.4%.
- Pause and observe.
- This is also expected since the rate of growth of the quadratic polynomial (n^2 in this case) is much higher than the linear polynomial ($5n$ in this case).

$n^2 + 5n + 1000$

- For n = 10, the third term contributes to 87% of the steps, the second term contributes 4.3% and the first term 8.7%.
- For n = 1000, the third term contributes to 0.1% of the steps, the second term contributes 0.5% and the first term 99.4%.
- What does this tell you?
- For a large input size, the term with the highest polynomial is important, the rest can be dropped.

$n^2 + 5n + 1000$

- If the code took $2n^2$ instead of n^2 steps, the run time would be twice (approximately).
- However, the difference is not significant. You would probably care about the factor, if it reduces the runtime *significantly*.

This kind of analysis leads us to use the following rules of thumb in describing the asymptotic complexity of an algorithm:

- If the running time is the sum of multiple terms, keep the one with the largest growth rate, and drop the others.
- If the remaining term is a product, drop any constants.

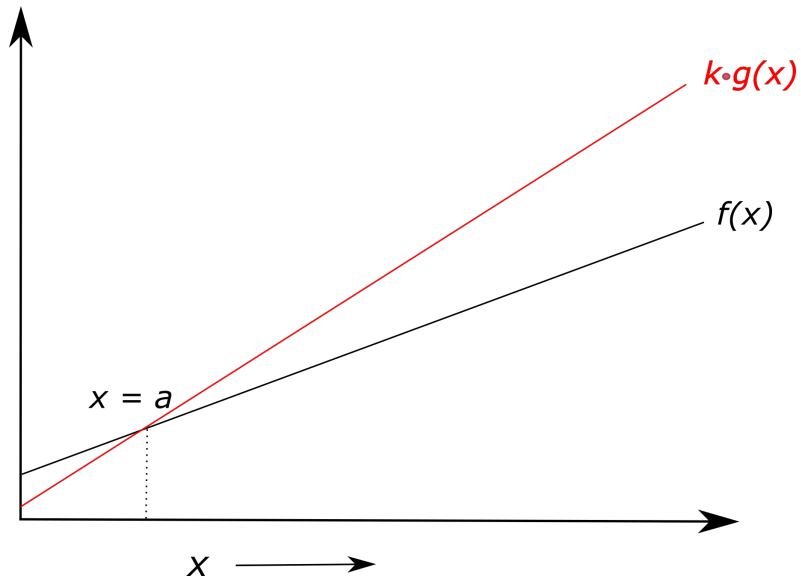
The Big O Notation (loose upper bound)

Definition

For functions f and g, we say that $f \in O(g)$ when there exists **at least one** choice of a constant $k > 0$, where you can find a constant a such that the inequality $0 \leq f(x) \leq k \cdot g(x)$ holds for all $x > a$.

- Big O notation is used to give an upper bound on the asymptotic growth (often called the order of growth) of a function.
- For example, the formula $f(x) \in O(x^2)$ means that the function f grows no faster than the quadratic polynomial x^2.

The Big O Notation (loose upper bound)



Example

Is $5n^2 \in O(n^2)$? If yes, what are examples of constants (a, k) where this relationship holds?

Proof by induction

Start with the statement that you want to prove: "Let $P(n)$ be the statement ...". To prove that $P(n)$ is true for all $n \geq 0$, you must prove the following two facts:

- **Base case**: Prove that P(0) is true. You do this directly. This is often easy.
- **Inductive case**: Prove that $P(k) \rightarrow P(k+1)$ for all $k \geq 0$, i.e. prove that for any $k \geq 0$ if $P(k)$ is true, then $P(k+1)$ is true as well.

Assuming you are successful on both parts above, you can conclude, "Therefore by the principle of mathematical induction, the statement P(n) is true for all $n \geq 0$."

Inductive case: Prove that $P(k) \rightarrow P(k+1)$ for all $k \geq 0$, i.e. prove that for any $k \geq 0$ if P(k) is true, then P(k + 1) is true as well.

You can assume P(k) is true. You must then explain why P(k + 1) is also true, given that assumption.

Example: Proof by induction

Show that $1 + 2 + ... + n = \frac{n(n+1)}{2}$ for all $n \geq 1$.

In other words, $P(n) = \frac{n(n+1)}{2}$ is true for any $n \geq 1$.

What is the base case here?

P(1) is true. Easy?

What is the inductive case?
For any $k \geq 1$, prove that if P(k) is true then P(k + 1) is true. Easy?
Let's see.

Prove that if P(k) is true, then P(k + 1) is true (for any $k \geq 1$).

- Since $P(k)$ is true, $P(k) = 1 + 2 + ... + k = \frac{k(k+1)}{2}$.
- For $P(k+1)$, we should get $\frac{(k+1)(k+2)}{2}$. How?
- $P(k+1) = P(k) + (k+1)$ which translates to $P(k+1) = \frac{k(k+1)}{2} + (k+1)$
- If $P(k)$ was true, then $P(k+1)$ is also true.

Therefore by the principle of mathematical induction, the statement P(n) is true for all $n \geq 1$.

Q for U

Using proof by induction show that $1 + a + a^2 + ... + a^n = \frac{a^{n+1}-1}{a-1}$ for all $n \geq 0$. Assume $a \neq 1$.

Coming back to asymptotic analysis

Prove that $n \in O(n^2)$ for $n \geq 1$.

Base Case: $n \leq k \cdot n^2$ when $n = 1$.

When $k \geq 1$, $1 \leq k \cdot 1^2$, which is True.

Inductive Case: Assume $n = c$, $c \leq k \cdot c^2$, is True.

Since $c \geq 1$, eliminate c from both sides to give $1 \leq c$

Prove that $(c+1) \leq k \cdot (c+1)^2$ is True.

Eliminate $(c+1)$ from both sides. $1 \leq k(c+1)$

Since $kc \geq 1$, $kc + k \geq 1$.

Q for U

Prove that $2^n \in O(n!)$ for $n \geq 4$.

Comments on the Big O notation

- The Big O notation is loose and can often be *abused* by making statements like, "the complexity of f(x) is O(x^2)". This means that in the worst case f will take O(x^2) steps to run.
- However, it can be much less than O(x^2).
- Thus, the Big O notation gives us a loose upper bound on the complexity.

Section 4

Complexity classes

Constant Complexity: $O(1)$

- This indicates that the asymptotic complexity is independent of the inputs.
- All programs have pieces (for example finding out the length of a Python list or multiplying two floating point numbers) that fit into this class.
- Constant running time does not imply that there are no loops or recursive calls in the code, but it does imply that the number of iterations or recursive calls is independent of the size of the inputs.

Logarithmic Complexity: $O(\log n)$

- Such functions have a complexity that grows as the log of at least one of the inputs. Example: Binary search.
- BTW, the base of the log is immaterial, since the difference between using one base and another is merely a constant multiplicative factor. For example, $\mathrm{O}(\log_2(x)) = \mathrm{O}(\log_2(10) * \log_{10}(x))$.

```
def get_digits(i):
    """Assumes i is a nonnegative int"""
    digits = []
    if i == 0:
        return i
    while i > 0:
        digit = i%10
        digits.append(digit)
        i = i//10
    return digits
```

Linear Complexity: $O(n)$

Many algorithms that deal with lists or other kinds of sequences are linear because they touch each element of the sequence a constant (greater than 0) number of times.

```
def sum_of_digits(s):
    """Assumes s is a str each character of which is a decimal digit.
    Returns an int that is the sum of the digits in s"""
    val = 0
    for c in s:
        val += int(c)
    return val
```

Q for U: Recursive Factorial

What is the time complexity of the following code:

```
def fact_recursive(x):
    """Assumes that x is a positive int
    Returns x!"""
    if x == 0 or x == 1:
        return 1
    else:
        return x*fact_recursive(x-1)
```

There are no loops in this code. In order to analyze the complexity we need to figure out how many recursive calls get made.

Polynomial Complexity: $O(n^k)$

The most commonly used polynomial algorithms are quadratic, i.e., their complexity grows as the square of the size of their input.

```
def is_subset(L1, L2):
    """Assumes L1 and L2 are lists.
    Returns True if each element in L1 is also in L2
    and False otherwise."""
    for e1 in L1:
        matched = False
        for e2 in L2:
            if e1 == e2:
                matched = True
                break
        if not matched:
            return False
    return True
```

Exponential Complexity: $O(c^n)$

- A function f(n) is exponential, if it has the form c^n, where c is a constant. For example, 2^n, is an exponential function.
- A program or a function that has exponential running time is bad news because such programs run extremely slowly!
- Let's see why!

A program with exponential time

- Suppose the running time of a function is 2^n. Further suppose that each primitive operation can be executed in one nano second (i.e., 10^{-9} seconds).
- Then solving the problem for n equals 100 will take $10^{-9} \times 2^{100} \simeq 10^{21}$ seconds or some $\sim 3.2 \times 10^{13}$ years.
- You get the point?
- The function starts innocuously enough but grows extremely rapidly and reaches astronomical proportions very quickly and for a fairly small value of n.
- Programs or functions whose running time is exponential can be useful only for tiny inputs.

Q for U: Recursive Fibonacci

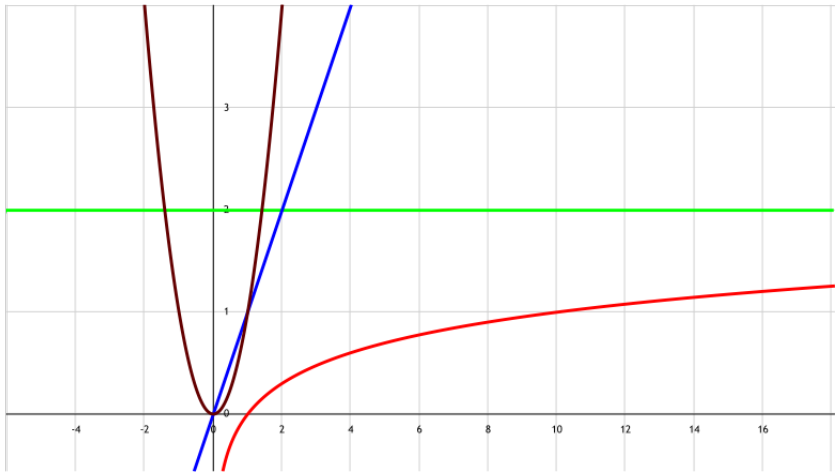
What is the time complexity of the following code:

```
def fibo_recursive(x):
    """Assumes that x is a positive int
    Returns x!"""
    if x == 0:
        return 0
    elif x == 1:
        return 1
    else:
        return fibo_recursive(x-1) + fibo_recursive(x-2)
```

In order to analyze the complexity we need to figure out how many recursive calls get made.

Q for U

Can you recognize the complexity classes for the curves?



Online demo for polynomial vs. exponential: Link

What did we learn today?

1. Different use cases, different priorities
2. Breaking down the run time
3. Asymptotic Notation
4. Complexity classes

Thank you!