

Big 'O' Notation

Lecturer: Dr Anwarul Patwary

Outline

- ▶ How can we describe how fast or efficient an algorithm is?
- ▶ What is asymptotic complexity?
- ▶ What is “Big ‘O’ ” notation?

Introduction

- ▶ If someone says they have a “fast” or “efficient” algorithm for some task, what exactly do they mean?
- ▶ They don't mean some exact time the algorithm takes to run – that could differ from computer to computer.
- ▶ Even if you and I were both running the same sorting algorithm that someone said was “efficient”, and running it on the same data . . .
 - if my computer is old and slow, or I am running lots of other programs on it at the same time –
 - and your computer is fast and new –
 - then we would expect the algorithm to take different amounts of time on each computer.

Comparing running time

- ▶ So we can't describe how "fast" some algorithm is by saying "It takes so many seconds" – that will vary from computer to computer.
- ▶ And even on the same computer – if we run the sorting algorithm once to sort, say, four items, and then again, to sort four *billion* items, we expect the second task to take much longer.
- ▶ The same algorithm is being used, and it's just as efficient in each case, but it's working with different data.

Asymptotic complexity

- ▶ Instead, we compare algorithms using something called *asymptotic complexity*.
- ▶ And we have a notation, “Big ‘O’ ” (or “Big Oh”) notation, for writing down different sorts of asymptotic complexity.

Asymptotic

- ▶ “Asymptotic” means “the way something behaves, as a variable it depends on increases towards infinity”.
- ▶ Let’s look at an example.

Number of characters in a string

- ▶ Suppose I want to count the number of characters in a string.
- ▶ We can imagine one way to do this – we start at the first character in the string, and “walk” along it, incrementing the number of characters we have seen each time by 1.
- ▶ And when we get to the end, we’ll know the total number of characters in the string.

Linear time

- ▶ Using this method, we would expect that, if we count the number of characters in a 1-million-character string, and the number of characters in a 2-million-character string, the second task will take roughly twice as long as the first.
- ▶ When describing complexity, we often use the variable n to denote the size of our input. In this case, it means the length of the string being counted.

Linear time

- ▶ In this case, when we double the size of the input – that is, we double n – we expect the running time of the program to double as well.
- ▶ We call two things that are related in this way, *linearly related*.

Growth as n tends towards infinity

- ▶ Why are we interested in the behaviour of our algorithm “as n tends towards infinity”?
- ▶ Well, for very small strings, it may not be true that the run-time of the algorithm is exactly linearly related to n .
- ▶ Perhaps when the algorithm starts, there will be some time needed to do things like initialize variables, and so on – let’s say these take 4 milliseconds on my computer.
- ▶ And suppose counting the length of a 10-character array takes 14 milliseconds, and counting the length of a 20-character array, 24 milliseconds.
- ▶ Is two times 14 equal to 24? It is not.

Growth as n tends towards infinity

- ▶ So for these small arrays, the relationship between n (the size of our input) and the run-time of the program is *not* exactly linear.
- ▶ But as n gets bigger and bigger, the 4 milliseconds needed to initialize variables will be a smaller and smaller fraction of the running time.
- ▶ And as n “*tends towards infinity*”, the relationship *will* be linear.

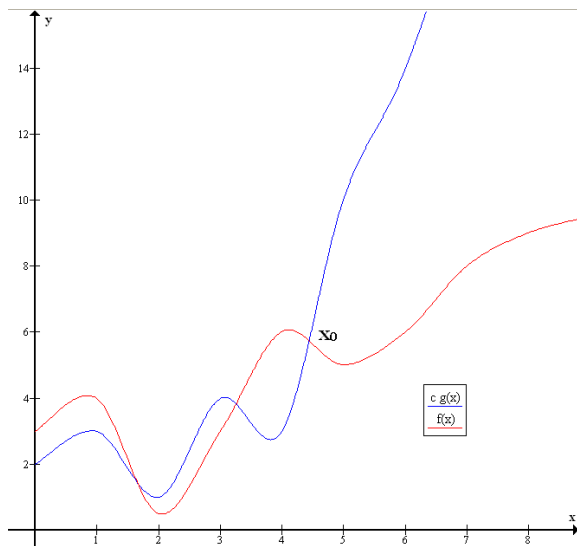
Notation for linear growth

- ▶ We say that this algorithm runs “in linear time”.
- ▶ And we write this in “Big ‘O’ ” notation as $O(n)$.
- ▶ What is the formal definition of what $O(n)$ means?

Formal definition of $O(n)$

- ▶ Well, suppose we have a function $f(x)$ describing the running time of an algorithm.
- ▶ And suppose we have a second function, $g(x)$, which we would like to say is the “order of complexity” of $f(x)$.
- ▶ When are we allowed to do so?
- ▶ We are allowed to do so when, beyond a certain point, $f(x)$ is no bigger than *some constant multiple* C of $g(x)$.

Diagram of two functions $f(x)$ and $g(x)$



[Image courtesy Wikimedia Commons, <https://en.wikipedia.org/wiki/File:Big-O-notation.png>]

Formal definition of $O(n)$

- ▶ In the diagram shown, there *is* a certain point, and a constant C , beyond which $f(x)$ is no bigger than $C \times g(x)$.
- ▶ That point is labelled x_0 ; and it looks like we could give C the value 1.

Our string-length example

- ▶ This formal definition is true of our string-length example as well.
- ▶ We imagined that the formula for the running time of that algorithm (on my computer, in milliseconds) was $f(n) = 14 + n$.
- ▶ We would like to say that the algorithm has *linear* running time; so we're proposing that there's a function $g(n) = n$ which describes this running time.
- ▶ Are we *allowed* to say the algorithm has linear running time, according to the formal definition?

Our string-length example

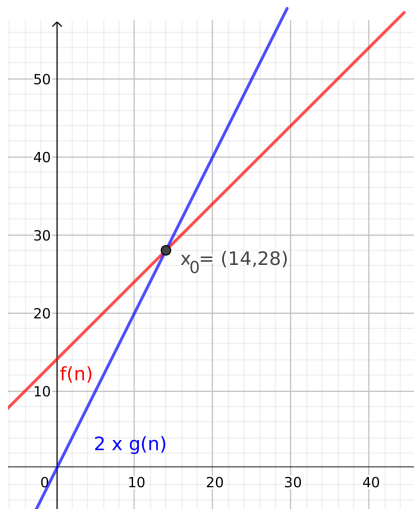
- ▶ We may do so if we can supply a constant C , and a “threshold point” x_0 , beyond which $f(n)$ is always less than or equal to $C \times g(n)$.

i.e., for all values of n where $n \geq x_0$,

$$f(n) \leq C \times g(n).$$

- ▶ Is there such a C , and such an x_0 ?
Yes, there is.
- ▶ We will let $C = 2$, for the moment, and take a look at the graphs of $f(n)$ and $2 \times g(n)$.

Graph of our string-length example



Graph of our string-length example

- ▶ From the graph, we can see that there is a point x_0 – where $n = 14$ – beyond which, $f(n)$ is *always* less than or equal to $C \times g(n)$.
- ▶ So we may say that $g(n)$ describes the asymptotic complexity of our algorithm.
- ▶ Since $g(n) = n$, we use the notation $O(n)$ to write this.
(If, say, $g(n)$ were instead equal to n^2 , we would use the notation $O(n^2)$.)

Informal intuition

- ▶ But we will not make you work too much with the formal definition.
- ▶ Instead, we will just ask you to remember that it means, roughly: when we say the complexity of some algorithm is $O(\text{something})$ – where the “**something**” is some formula involving n – we mean that as the size of our input grows larger, that formula is a *bound* on the running time of our algorithm.

Constant time

- ▶ Let's look at another sort of complexity an algorithm could have.
- ▶ We could have an algorithm where, no matter how large our input grows, it always takes the *same* length of time to run.
- ▶ Let's see an example.

Counting string-length by cacheing

- ▶ Imagine again that we are wanting to count the length of strings.
- ▶ But instead of strings that are 10, or 20, or even a million characters long, we are now working with strings that are *billions* of characters long.
- ▶ For strings this big, we decide that even our $O(n)$ algorithm isn't fast enough.
- ▶ So we try something different . . .

Counting string-length by cacheing

- ▶ Whenever we *first* construct a string, presumably we know at that point how big it is.
- ▶ So together with the actual string, we store a number, its length.
- ▶ And if we join two strings together, say – then we know the length of the new string is just the sum of the lengths of the original two strings.
- ▶ Eventually, our program produces the billions-of-characters-long string we want to know the length of.

Counting string-length by cacheing

- ▶ How can we find its length? We just look at the number we stored with it describing its length.

We could imagine that we have a class that looks something like this:

```
public class KnownLengthString {  
    //fields (attributes)  
    public int strLength;  
    public String str;  
    // ... methods and constructors ...  
}
```

- ▶ So to find the length of the string, we just look at the value of strLength.
- ▶ How long will this take us? It will take the same length of time, *regardless of how large the string is.*
- ▶ You may sometimes see this sort of strategy referred to as *cacheing* (storing) the length so that we can use it quickly later.

Complexity of our new string-length algorithm

- ▶ What is the complexity of our new algorithm?
- ▶ Well, it will always take the same length of time, regardless of the length n of the string.
Perhaps on my computer, it takes 2 milliseconds.
- ▶ So what is the “Big ‘O’ ” complexity of the algorithm? It is $O(1)$.
- ▶ Because we can come up with a value of C (in this case, 2) and a value of x_0 (in this case, 0) such that for all strings with length greater than x_0 , $C \times 1$ is a bound on the run-time of the program.
- ▶ We say that our new algorithm “*runs in constant time*” (with respect to the size of the input).

Drawbacks of our new string-length algorithm

- ▶ So our new string-length algorithm is much faster than the previous, $O(n)$ algorithm.
- ▶ But it comes with some drawbacks.
- ▶ For instance, we now have to spend a little bit of memory, for each string, storing the length. If we were storing many, many strings, this might become significant.
- ▶ Our code will also be a bit more complicated to write – every time we create a new string (for instance by joining other strings together) we will need to correctly calculate and store its length.

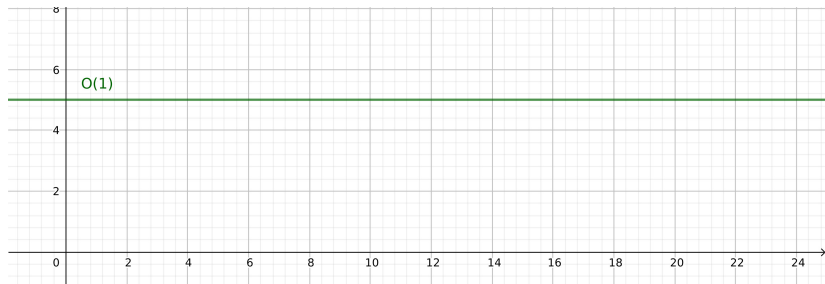
Other common sorts of complexity

- ▶ We have seen an algorithm which has *linear*, $O(n)$ complexity
- ▶ And another that has *constant*, $O(1)$ complexity
- ▶ But there are many other formulas we sometimes see.

Advantages of Big 'O' Notation

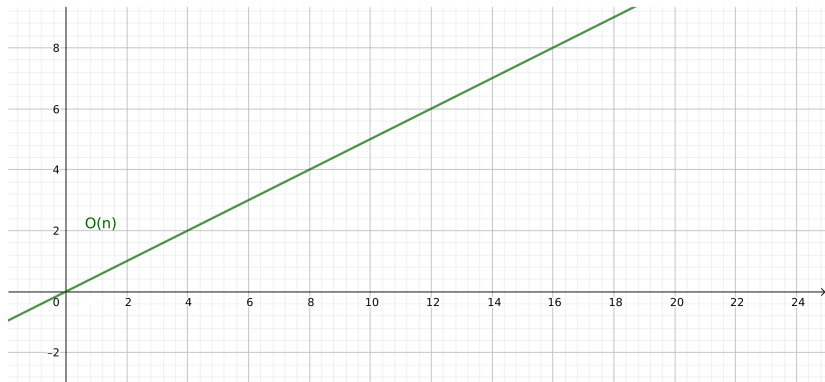
- ▶ Big 'O' complexity is unaffected by running-time being doubled, halved, etc (in fact, being multiplied by any constant).
- ▶ Which is good, because if it wasn't, algorithms would have different complexity on (say) your fast laptop as opposed to my slow laptop – and we want a measure which *ignores* those differences.
- ▶ Instead, Big 'O' Notation tells us about the *general shape* of the run-time graph as the size of input tends towards infinity.

Examples of complexity types - $O(1)$



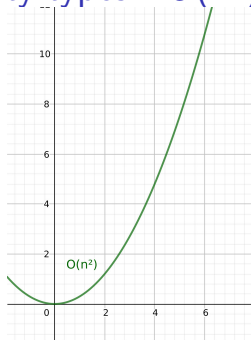
- ▶ $O(1)$: constant time.
- ▶ As the size of input increases, the run-time of the algorithm remains constant.
- ▶ Examples: reading from a variable; looking up a cell from an array

Examples of complexity types - $O(n)$



- ▶ $O(n)$: linear time.
- ▶ As the size of input increases, the run-time of the algorithm increases proportionately.
- ▶ Examples: inspecting each element of an array or list; any loop (e.g. a “for” loop) that iterates over the input.

Examples of complexity types - $O(n^2)$



- ▶ $O(n^2)$: quadratic time.
- ▶ As the size of input increases, the run-time of the algorithm increases in proportion to the *square* of the size of the input.
- ▶ Examples: algorithms with nested “for” loops:

```
for(int i = 0; i < n; i++) {  
    for(int j = 0; j < n; j++) {  
        // ... some operation  
    }  
}
```

Choosing algorithms

- ▶ We say that $O(n^2)$ algorithms are “*asymptotically slower*” than $O(n)$ algorithms.
 - ▶ Meaning, the $O(n^2)$ algorithm might actually run *faster* than the $O(n)$ algorithm, for small values of n .
 - ▶ But there's some point beyond which, as the size of the input n grows, the $O(n^2)$ algorithm is *always* slower.
- ▶ So if we have the choice between an $O(n)$ and an $O(n^2)$ algorithm, which should we choose?

Choosing algorithms

- ▶ It depends on exactly what task we are doing.
- ▶ If the $O(n^2)$ algorithm is easier to code than the $O(n)$ algorithm, and we know we will only be dealing with relatively small inputs, then perhaps the $O(n^2)$ algorithm is fine – we probably won't know until we measure how long it takes.
- ▶ But the larger the input, the more of a problem the run-time of our $O(n^2)$ algorithm will become, so eventually, we may need to spend time carefully coding the trickier $O(n)$ algorithm.

Complexity types

- ▶ We will see other sorts of run-time complexity later – for instance, algorithms that take *logarithmic time* (written as $O(\log n)$).