Module COMP20230: Data Structure and Algorithms
Lecture 3: Running Time and Analysis, including Big-O

Scribes: David Geraghty and Yvette Cripwell

4th February 2019

1 Previous Lecture - Lecture 2

An **algorithm** is any well defined computational procedure that takes some value or a set of values as input and produces some value or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input in the output. (*Cormen et al, Introduction to Algorithms, 2009*)

1.1 Phases of an algorithm

The phases of an algorithm are:
Design
Identifying the problem and thoroughly understanding it. When designing an algorithm, begin by listing any or all assumptions you make. Then specify the inputs, and the method to achieve the outputs.
Analysis
Once you have a basic framework begin to analyze the algorithm for how efficient it is in solving the problem.
Application
This is when the algorithm is implemented in a coding language and put into use.

1.2 Attributes of a good algorithm

1. Correctness - with respect to its specification, finite
2. Speed - time to execute
3. Efficiency - its resource usage, usually time or space
4. Security
5. Robustness - works for wide variety of inputs, including outliers
6. Clarity - simple, precise, unambiguous
7. Maintainability - easy to understand, change

2 This Lecture - Lecture 3

2.1 Algorithm Efficiency

Algorithm efficiency is important for scaling and performance. Efficiency can measured using space(memory), time or power consumption. This lecture focuses on using the **running time** of an algorithm as the **measurement of its efficiency.**The running time of an algorithm varies with the input and typically grows with input size. As the **average case** can be difficult to determine, for algorithms the **worst case** running time is focused on as it is easier to analyse and crucial to many applications such as games, finance, robotics and real time systems.

2.2 Measuring Running Time

We measure running time using a **high-level description** of the algorithm (e.g. pseudo-code) instead of writing a program to implement it, and then running it for various inputs, measuring the actual running times and plotting the results. Using **pseudo-code** rather than a program **avoids problems** such as

- the running time being dependent on the computer's hardware and software
- the inputs not entirely testing the algorithm
- the situation where it is not practical to implement the algorithm

Instead, it lets us

- take into account **all possible inputs**
- evaluate the speed of an algorithm **independent of the hardware and software environment**
- determine the **number of statements executed** by an algorithm **as a function of the input size**, by inspecting the pseudo-code

We use **pseudo-code** because is a **high-level description** of an algorithm. Pseudo-code is more structured than English prose but less detailed than a program. It is the preferred notation for describing algorithms as it hides program design issues. It typically uses the structural conventions of a programming language but is intended to be human understandable rather than machine understandable.

2.3 Algorithm Analysis

The following table contains the **mathematical functions** used to describe how the running time of an algorithm scales with the increasing size of the input in Big-O. The functions are shown in decreasing order of efficiency.

$f(c)$	Constant
$f(logn)$	Logarithmic
$f(n)$	Linear
$f(nlogn)$	N-Log-N
$f(n^2)$	Quadratic
$f(n^3)$	Cubic
$f(2^n)$	Exponential

2.4 Algorithm Complexity

The **complexity** of an algorithm is determined by identifying the number of **basic/primitive/elementary operations** in each line of the **pseudo-code**. These **elementary operations** are as follows:

- Math (e.g. $+, -, *, /, max, min, log, sin, cos, abs$, ...)
- Comparisons ($==, >, <=$, ...)
- Function calls and value returns (excluding operations executed within the function)
- Variable assignment
- Variable increment or decrements
- Array allocation
- Array access (e.g. accessing a single element of a Python list by index)
- Creating a new object (careful, object's constructor may have elementary ops too!)

In practice, these operations all take different amounts of time but for the purpose of algorithm analysis, we assume each of these operations takes the same time: **"1 operation".**

2.5 Estimating Running Time

By inspecting an algorithm or a program, we can determine the **maximum number of primitive operations executed as a function of the input size**. Each line is examined to determine how many operations, from the previous list (subsection 2.4), it contains. These operations are then summed together. This is done for both the **worst case** and **best case** scenarios.
Take the following python function **'findmax'** as an example. It returns the largest element from a Python list.

1	def findmax (data):		2 operations
2	biggest = data[0]	#initial max value to beat	2 operations
3	for val in data	# for each value	2 operations per loop
4	if val > biggest:	# if bigger than current biggest	2 operations per loop
5	biggest = val	# found new biggest	0 to n operations
6	return biggest	# return biggest value in list	1 operation

The operations in lines 1,2 and 6 are constant for the program irrespective of the size of the input (n). The operations in lines 3 and 4 are dependent on the size of the input and therefore occur 2n times. Line 5 is dependent on the size of the input and the result of the evaluation of the if statement in line 4 so the operation occurs 0 to n times. Thus the **worst case** number of operations for an input size of n is $\boldsymbol{5n + 5}$ and the **best case** is $\boldsymbol{4n + 5}$. The difference between the two results is whether line 5 is run n or 0 times.

3 Running time and complexity

In general the running time of an algorithm has an upper and lower bound. The algorithm for the findmax(data) function is bounded by two linear time functions:

$$a(4n+5) \leq T(n) \leq b(5n+5)$$

where a is the time taken by the fastest operation and b is the time taken by the slowest. The functions differ in there coefficients of n because the worst case scenario models the situation in which the for-loop executes the value reassignment statement n times, and the best case where it never executes the reassignment. In a given situation the running time will be between these two extremes.

3.1 What About Implementation?

We need a way to assess the efficiency of an algorithm that is independent of the hardware and software environment. These implementation details do affect the running time, $T(n)$, by a constant factor but do not change the **growth rate** of the **algorithm** with respect to the input size.

The **linear** growth rate of the function $T(n)$, is an **"intrinsic property"** of the algorithm.

3.2 Algorithm Analysis and Big-o Notation

We are interested in the performance of algorithms when the input sizes are extremely large (i.e. the worst case scenario). We express the time taken by an algorithm (independent of implementation) in the number of operations it performs with respect to the input n. So we can derive a function $f(n)$ for the algorithm.

When we analyse this function, we can ignore the low-order n-terms and the constants, because as the input size gets bigger, the low-order terms and the constants make smaller and smaller contributions to the total time it takes for the whole algorithm to get to the end. You can see how this works by using an inductive proof, which can be applied to a function for a given algorithm.

Say we have a function: $f(n) = 7n - 2$, for an algorithm. From just looking at the function we can tell that the highest order n-term is $7n$. We can ignore the coefficient as well and say that the time complexity of the function is $O(n)$, or that it "runs in $O(n)$ time."

3.3 Some Formalism

More formally, the function $f(n)$ can be described as $O(g(n)$, where $g(n) = n$, (that is $O(n)$), if there are constants c and n_0, that allow the following inequality to hold true:

$$f(n) \leq c(g(n))$$

for all $n \geq n_0$

The constants $\boldsymbol{c}$ and $\boldsymbol{n_0}$ can be derived. For starters $\boldsymbol{n_0}$ can be found by finding the minimal value of $\boldsymbol{n}$ that gives a positive value for $\boldsymbol{f(n)}$.

Try $\boldsymbol{n_0 = 0, 1, 2}$ etc.

Set $\boldsymbol{n_0 = 0}$
$\boldsymbol{f(n_0) = 7(0) - 2 = -2}$
The running time can't be less than 0, so this won't work.

Set $\boldsymbol{n_0 = 1}$
$\boldsymbol{f(n_0) = 7(1) - 2 = 5}$
So $\boldsymbol{n_0 = 1}$ is the minimum value of $\boldsymbol{n}$ that gives a positive running time.

We can find $\boldsymbol{c}$ by inducting values of $\boldsymbol{n_0 \geq 1}$ into the inequality and then rearranging the inequality to isolate $\boldsymbol{c}$.

Recall: $\boldsymbol{f(n) \leq c(g(n))}$
Expanded: $\boldsymbol{7n - 2 \leq c(n)}$

Use a series of inductive steps (from induction notes)

For $\boldsymbol{n = 1}$:
$\boldsymbol{7(1) - 2 \geq c(1)}$
$\boldsymbol{5/1 = 5}$
That means that the ratio of $\boldsymbol{f(n)/g(n) = 5}$
Or $\boldsymbol{c \geq 5}$
In other words the contribution from $\boldsymbol{f(n)}$ is 5 times greater than from $\boldsymbol{g(n)}$.

For $\boldsymbol{n = 2}$:
$\boldsymbol{7(2) - 2 \leq c(2)}$
$\boldsymbol{12/2 = 6}$
$\boldsymbol{c \geq 6}$

For $\boldsymbol{n = 4}$:
$\boldsymbol{7(4) - 2 \leq c(4)}$
$\boldsymbol{26/2 = 13}$
$\boldsymbol{c \geq 6.5}$

For $\boldsymbol{n = 100}$:
$\boldsymbol{7(100) - 2 \leq c(100)}$
$\boldsymbol{698/100 = 6.98}$
$\boldsymbol{c \geq 6.98}$

For $\boldsymbol{n = 1000}$:
$\boldsymbol{7(1000) - 2 \leq c(1000)}$
$\boldsymbol{6998/1000 = 6.98}$
That means that the ratio of $\boldsymbol{f(n)/g(n) = 6.998}$
Or $\boldsymbol{c \geq 6.998}$
In other words the contribution from $\boldsymbol{f(n)}$ is 6.998 times greater than from $\boldsymbol{g(n)}$.

As we can see this trend represents an asymptotic curve, where the contribution of $\boldsymbol{f(n)}$ cannot ever exceed $\boldsymbol{g(n)}$ by more than a factor of $\boldsymbol{c = 7}$.

Therefore we can say that $\boldsymbol{f(n)}$ has an upper bound of $\boldsymbol{c(g(n))}$, where $\boldsymbol{c = 7}$.

$g(n)$ is a function that represents the contribution from the biggest order n-term. Which is $g(n) = n$, so we can say that:
$f(n)$ is in $O(g(n))$, or $O(n)$ (the biggest term) (think of the Big-O as shorthand for saying "the order of") and that $f(n) \leq c(g(n))$

3.4 Putting it together

An algorithm whose running time is given by $f(n)$ is $O(g(n))$, if the growth rate of the algorithm is not greater than $g(n)$.

For algorithms whose biggest n-term is say n^2 or n^3, we can perform a similar analysis as above to find the constants c and n_0.

We are typically concerned with assigning a Big-O order, like $O(n)$, to an algorithm, for the sake of very large inputs. The relative contribution from the largest order n-term is so large that the rest of the algorithm may as well not even exist.

3.5 Summary

To assign the time order of an algorithm, examine the algorithm and figure out the number of operations on each line, paying careful attention to the parts of the algorithm that introduce data inputs that need to be traversed, like arrays, which introduce 'n's into the function.

Multi-dimensional loop structures have higher orders than single-dimension loop structures. Control-flow constructs should be treated as having the order of the subordinate code block that has the highest order (because you assume the worst case scenario).

Once you have the function, drop all the terms except for the highest n-term, and also drop the coefficients. Assign the algorithm to an order using just the order of the highest n-term. It wouldn't be technically wrong to assign $f(n) = n^3 + 5$ to $O(n^4)$, but this isn't really helpful because n^4 is n times larger than n^3, and so isn't a good representation of $f(n)$.

If an algorithm has a running time described as $f(n) = k$, where k is just a constant number, then the convention is to assign $f(n)$ as $O(1)$ because it doesn't grow as n gets bigger.