Analysis of Algorithms

Dr Nicholas Moore

Dept. of Computing and Software, McMaster University, Canada

Acknowledgments: Material mainly based on the textbook Algorithms by Robert Sedgewick and Kevin Wayne (Chapters 1.4) Prof. Mhaskar's course slides.

Measuring Runtime

There are two ways to anaylze the runtime of algorithms:

- **Empirically** through benchmarking
  - This requires running specific test cases which may not be representative of all cases.
- **Theoretically** through analysis of the program structure itself.

In this class we will learn how to perform a theoretical analysis of the runtime of algorithms.

Runtime Factors

In general, the total running time of a program is determined by two factors (D.E. Knuth):

- The cost of executing each statement
- The frequency of execution of each statement.

Overall, our goal is to derive a **time function** $T(n)$ which describes the amount of time it takes to execute a program based on the size of the input.

- Here, n is the **size of the input**.
    - Normally, this will be the size of the data structure we are processing.

Analysing Swap

```
1: procedure SWAP(∅)
Require: two variables, $x$ and $y$
2:     $temp \leftarrow x$
3:     $x \leftarrow y$
4:     $y \leftarrow temp$
5: end procedure
```

The execution time of each of of these statements is determined by how many machine code instructions each statement compiles to.

Each statement's execution time is, however, indepedent of the values of x and y. We can therefore write them as constants.

- Let's create constants c_2, c_3 and c_4 to represent the runtimes of lines 2, 3 and 4. In this case:

$$T(n) = c_2 + c_3 + c_4 + c_f \tag{1}$$

Where c_f is your overhead cost for calling a function

Analysing Linear Search

```
1: procedure LINEAR-SEARCH($A, k$)
2:     for i from 1 to A.length do
3:         if $A[i] == k$ then
4:             return $True$
5:         end if
6:     end for
7:     return $False$
8: end procedure
```

- Let's create a new set of variables c_i to represent the runtimes of each line in Linear Search.
- How many times is statement 3 executed?

Analysing Linear Search II

Everything inside a loop is run once for each iteration of the loop!

- The number of iterations will vary, however! This loop breaks once k is found!
- For now, let's concern ourselves with the **worst case scenario**. In this case the loop executes n times, where n is the size of A.

In this case:

$$T(n) = c_{2,3,5,6}n + (c_4 + c_f) \tag{2}$$

(Regardless how many times the loop executes, either line 4 or 5 is executed only once!)

Analysing Bubblesort

Applying the same procedure...

```
1: procedure BubbleSort($A$)
2:     for i from 1 to A.length - 1 do
3:         for j from A.length down to i + 1 do
4:             if $A[j] > A[j+1]$ then
5:                 swap $A[j]$ and $A[j+1]$
6:             end if
7:         end for
8:     end for
9: end procedure
```

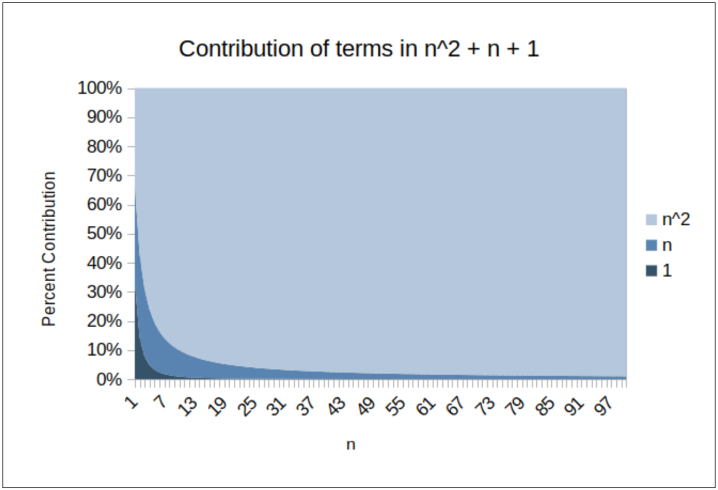
$$T(n) = (c_{3..7})n^2 + (c_2 + c_8)n + c_f \tag{3}$$

This is Getting Complicated...

The three algorithms we've analysed so far are extremely simple, and it's already a pain to keep track of all these constants! We need to make some simplifications.

- First, we can simplify our expression by setting some new constants equal to the values of other constants. For BubbleSort:

$$T(n) = an^2 + bn + c \tag{4}$$



Setting all constants to 1 (for the sake of argument), we can see that as n grows, the highest order term will tend to dominate.

Tilde Approximations

We write $\sim f(n)$ to represent any function for which:

$$\lim_{n\to\infty} \frac{\sim f(n)}{f(n)} = 1 \tag{5}$$

Applying this to our equations, we see that we are dropping everything but the highest order term.

$f(n)$	$\sim f(n)$
$(c_2 + c_3 + c_4 + c_f)$	$(c_2 + c_3 + c_4 + c_f)$
$c_{2:4}n + (c_5 + c_f)$	$c_{2:4}n$
$c_{3:7}n^2 + c_2 n + c_f$	$c_{3:7}n^2$

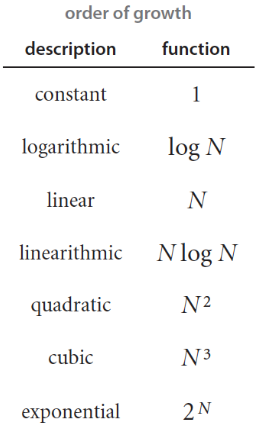
Further Observations and Simplifications

We might also observe that, as n grows, the effect of the given constants is proportionally reduced.

- This leads to a so-called **order of growth** approximation.
- Essentially, we take a tilde approximation and also drop the constant.
- Notationally, we take this approximation x and write it as $O(x)$ (pronounced "big-oh").

$f(n)$	$\sim f(n)$	$O(x)$
$(c_2 + c_3 + c_4 + c_f)$	$(c_2 + c_3 + c_4 + c_f)$	$O(1)$
$(c_2 + c_3)n + (c_5 + c_f)$	$(c_2 + c_3)n$	$O(n)$
$(c_3 + c_4 + c_5)n^2 + c_2 n + c_f$	$(c_3 + c_4 + c_5)n^2$	$O(n^2)$

The Importance of Big-O



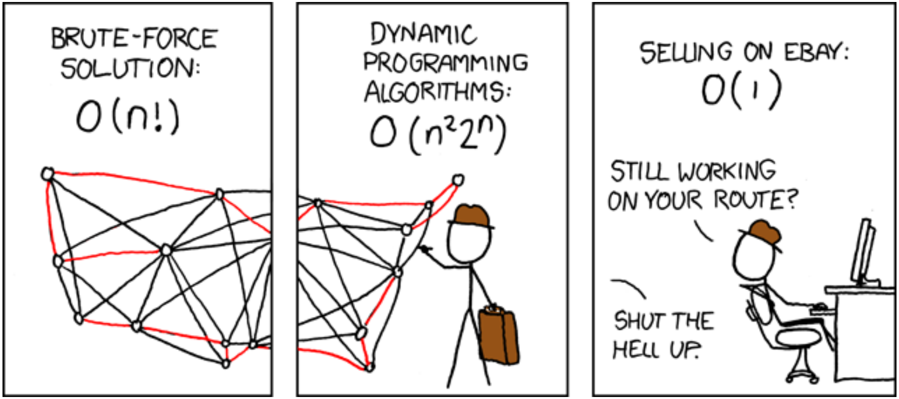
order of growth

description	function
constant	1
logarithmic	$\log N$
linear	N
linearithmic	$N \log N$
quadratic	N^2
cubic	N^3
exponential	2^N

Commonly encountered order-of-growth functions

- Big-O notation provides us with a much-needed classification scheme for categorizing algorithm behaviour.
- When an algorithm is described as “logarithmic”, “exponential”, “linear”, etc., we are describing what function the algorithm’s growth is proportional to.

Big-O

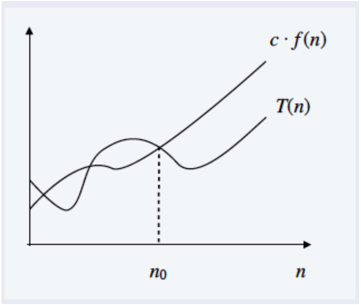

BRUTE-FORCE SOLUTION: O(n!)
DYNAMIC PROGRAMMING ALGORITHMS: O(n²2ⁿ)
SELLING ON EBAY: O(1)
STILL WORKING ON YOUR ROUTE?
SHUT THE HELL UP.

Big-O More Formally

$O(f(n))$ denotes a set of functions.

Given the existance of some constant c, some number of inputs n_0, for all $n \geq n_0$, if $T(n)$ is between 0 and $c \cdot f(n)$, then $T(n) \in O(f(n))$

- Effectively, $O(f(n))$ is an **upper bound** of $T(n)$ past some arbitrarily selected n_0.



$$\exists c > 0 \bullet \exists n_0 \geq 0 \bullet \forall n \geq n_0 \bullet 0 \leq T(n) \leq c \cdot f(n) \,|\, T(n) \in O(f(n)) \tag{6}$$

Big-O Examples

Consider $T(n) = 32n^2 + 17n + 1$. Which of the below are true? In other words, can we select a c and a n_0 such that any of these are true?

- $T(n) \in O(1)$
- $T(n) \in O(n)$
- $T(n) \in O(n \log n)$
- $T(n) \in O(n^2)$
- $T(n) \in O(n^3)$
- $T(n) \in O(2^n)$

Big-O Examples

$$
\begin{aligned}
T(n) \in O(1) &\Rightarrow & No!\\
T(n) \in O(n) &\Rightarrow & No!\\
T(n) \in O(n \log n) &\Rightarrow & No!\\
T(n) \in O(n^2) &\Rightarrow & Yes!\\
T(n) \in O(n^3) &\Rightarrow & Yes!\\
T(n) \in O(2^n) &\Rightarrow & Yes!
\end{aligned}
$$

The last three are all upper bounds of $T(n)$, but this is not necessarily useful information.

- In general, we want our bounds to be *as tight as possible!*

Demonstrating $T(n) \notin O(f(n))$

Consider the proposition $32n^2 + 17n + 1 \notin O(n)$. How might we demonstrate this?

$$\begin{aligned} 32n^2 + 17n + 1 &\notin O(n) \\ 32n^2 + 17n + 1 &\leq c \cdot n \\ 32n + 17 + \frac{1}{n} &\leq c \end{aligned}$$

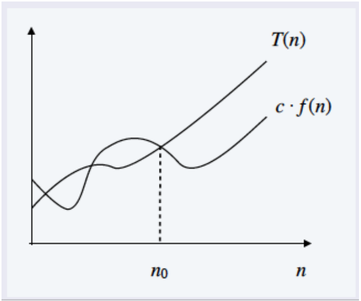
No matter how large a c we select, $32n$ will always be larger for large n. Therefore, the inequality does not hold, and $T(n) \notin O(n)$.

Big-Ω

$\Omega(f(n))$ also denotes a set of functions.

Given the existance of some constant c, some number of inputs n_0, for all $n \geq n_0$, if $T(n)$ is always greater than (or equal to) $c \cdot f(n)$, then $T(n) \in \Omega(f(n))$

- Effectively, $\Omega(f(n))$ is an **lower bound** of $T(n)$ past some arbitrarily selected n_0.



$$\exists c > 0 \bullet \exists n_0 \geq 0 \bullet \forall n \geq n_0 \bullet T(n) \geq c \cdot f(n) | T(n) \in \Omega(f(n)) \quad (7)$$

Big-Ω Examples

Consider $T(n) = 32n^2 + 17n + 1$. Which of the below are true? In other words, can we select a c and a n_0 such that any of these are true?

- $T(n) \in \Omega(1)$
- $T(n) \in \Omega(n)$
- $T(n) \in \Omega(n \log n)$
- $T(n) \in \Omega(n^2)$
- $T(n) \in \Omega(n^3)$
- $T(n) \in \Omega(2^n)$

Big-Ω Examples

$$\begin{aligned}
T(n) \in \Omega(1) &\Rightarrow & Yes!\\
T(n) \in \Omega(n) &\Rightarrow & Yes!\\
T(n) \in \Omega(n \log n) &\Rightarrow & Yes!\\
T(n) \in \Omega(n^2) &\Rightarrow & Yes!\\
T(n) \in \Omega(n^3) &\Rightarrow & No!\\
T(n) \in \Omega(2^n) &\Rightarrow & No!
\end{aligned}$$

The first three are all lower bounds of $T(n)$, but again, this is not necessarily useful information.

- There's not much that $\Omega(1)$ *isn't* a lower bound for!

Big-Omega Notation - useful tips

- To show that $T(n) \in \Omega(n^2), \Omega(n)$ you need to only provide correct c, n_0 values.
- To show that $T(n) = 32n^2 + 17n + 1 \notin \Omega(n^3)$ - You need to give a proof.

  Proof for the above:

  $T(n) = 32n^2 + 17n + 1 \in \Omega(n^3)$ implies that there are contants c, n_0 such that the below inequality holds for all $n \geq n_0$.

  $$32n^2 + 17n + 1 \geq c \cdot n^3$$

  For any such pair of constants c, n_0, let us consider $n = 50\lceil cn_0 \rceil$. Substituting $n = 50\lceil cn_0 \rceil$ in the above equation, we get $32(50\lceil cn_0 \rceil)^2 + 17(50\lceil cn_0 \rceil) + 1 < c \cdot (50\lceil cn_0 \rceil)^3$ – a contradiction.

It was the Best of Cases, it was the Worst of Cases.

In addition to best and worst cases, it is also sometimes useful to consider $T_{avg}(n)$, the average runtime over all inputs of size n.

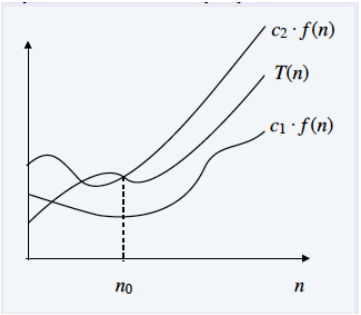
- While $T_{avg}(n)$ appears a fairer measure, it is often fallacious to assume that all inputs are equally likely.
- In practice, the average running time is often much harder to determine than the worst-case running time.
  - Analysis tends to become mathematically intractable.
  - The notion of "average input" frequently has no obvious meaning.
- Thus, we shall use worst-case running time as the principal measure of time complexity, with average-case complexity mentioned wherever meaningful.

Big-Θ

$\Theta(f(n))$ also denotes a set of functions.

Given the existance of two constants $c_1 > 0$ and $c_2 > 0$, some number of inputs n_0, for all $n \geq n_0$, if $T(n)$ is between $c_1 \cdot f(n)$ and $c_2 \cdot f(n)$, then $T(n) \in \Theta(f(n))$

- Effectively, $\Theta(f(n))$ is *both an upper and lower bound* of $T(n)$.



$$\exists c_1, c_2 > 0 \bullet \; \exists n_0 \geq 0 \bullet \; \forall n \geq n_0 \bullet$$
$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n) \mid T(n) \in \Theta(f(n)) \tag{8}$$

Big-Θ Notation Example

Consider $T(n) = 32n^2 + 17n + 1$. Which of the below are true? In other words, can we select a c and a n_0 such that any of these are true?

- $T(n) \in \Theta(1)$
- $T(n) \in \Theta(n)$
- $T(n) \in \Theta(n \log n)$
- $T(n) \in \Theta(n^2)$
- $T(n) \in \Theta(n^3)$
- $T(n) \in \Theta(2^n)$

Big-Θ Examples

$$\begin{aligned}
T(n) \in \Theta(1) \Rightarrow &\quad No! \\
T(n) \in \Theta(n) \Rightarrow &\quad No! \\
T(n) \in \Theta(n \log n) \Rightarrow &\quad No! \\
T(n) \in \Theta(n^2) \Rightarrow &\quad Yes! \\
T(n) \in \Theta(n^3) \Rightarrow &\quad No! \\
T(n) \in \Theta(2^n) \Rightarrow &\quad No!
\end{aligned}$$

Only n^2 serves as upper and lower bound for $T(n)$.

- To find $\Theta(f(n))$, first find the upper bound, then the lower bound, then see if they agree!

Common Errors

- **Equals sign**: $O(f(n))$, $\Omega(f(n))$ and $\Theta(f(n))$ are *set of functions*, but computer scientists often (erroneously) write $T(n) = O(f(n))$ to mean $T(n) \in O(f(n))$.
- **Domain**: The domain of $f(n)$ is typically the set of natural numbers $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$.
    - It doesn't make sense to speak of an array containing 7.34 elements!
- **Nonnegative functions**: When using Big-O (Big-Ω and Big-Θ) notation, we assume that the functions involved are (asymptotically) nonnegative.
    - It doesn't make sense for time to be negative unless you're Dr Who!

Pushing it to it's Limits!

Taking the limit of two functions might tell us whether O, Ω or Θ are applicable.

$$\begin{aligned}
\lim_{n\to\infty}\frac{f(n)}{g(n)} &\neq \infty \implies f(n) \in O(g(n)). \\
\lim_{n\to\infty}\frac{f(n)}{g(n)} &\neq 0 \implies f(n) \in \Omega(g(n)) \\
\lim_{n\to\infty}\frac{f(n)}{g(n)} &\notin \{0,\infty\} \implies f(n) \in \Theta(g(n))
\end{aligned}$$

DISCLAIMER: This is an OK approximation but it doesn't work in all situations! When in doubt, derive from the definitions of O, Ω, and Θ!

Big-Oh Notation with multiple variables

Big-Oh Notation with multiple variables:
$T(m,n) \in O(f(m,n))$ if there exist constants $c > 0$, $m_0 > 0$, and $n_0 \geq 0$ such that $0 \leq T(m,n) \leq c \cdot f(m,n)$ for all $m \geq m_0$ and $n \geq n_0$.

Example

$T(n) = 32mn^2 + 17mn + 32n^3$, then

- $T(m,n)$ is both in $O(n^3 + mn^2)$ and $O(mn^3)$
- $T(m,n) \notin O(mn^2)$ and $T(m,n) \notin O(n^3)$

Some helpful results

- $O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(max(f(n), g(n)))$
- $O(f(n))O(g(n)) = O(f(n)g(n))$
- **Polynomials:** Let $T(n) = a_0 + a_1 n + \ldots + a_d n^d$. with $a_d > 0$, Then $T(n) = \Theta(n^d)$. This is due to the fact that
$$\lim_{n \to \infty} \frac{a_0 + a_1 n + \ldots + a_d n^d}{n^d} = a_d$$
So, the limit is neither 0 nor ∞
- **Exponentials and polynomials:** For every $r > 1$, and every $d > 0$, $n^d \in O(r^n)$. Since
$$\lim_{n \to \infty} \frac{n^d}{r^n} = 0.$$

Why Runtime Matters

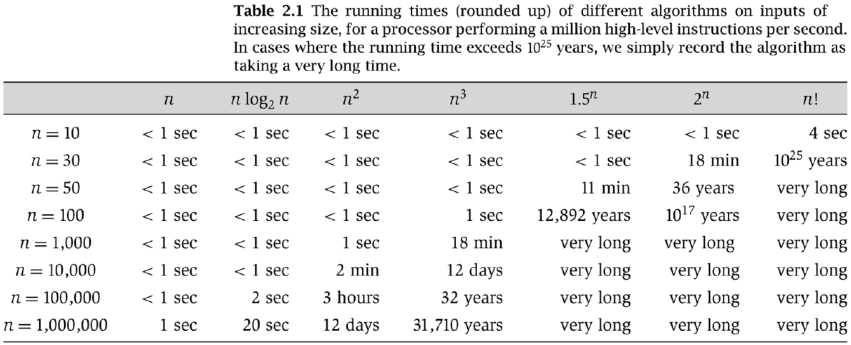
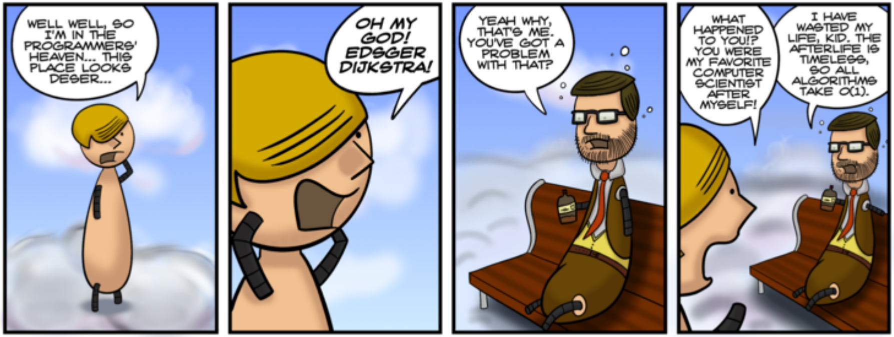


Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Obligatory End of Slides Comic


WELL WELL, SO I'M IN THE PROGRAMMERS' HEAVEN... THIS PLACE LOOKS DESER...
OH MY GOD! EDSGER DIJKSTRA!
YEAH WHY, THAT'S ME. YOU'VE GOT A PROBLEM WITH THAT?
WHAT HAPPENED TO YOU!? YOU WERE MY FAVORITE COMPUTER SCIENTIST AFTER MYSELF!
I HAVE WASTED MY LIFE, KID. THE AFTERLIFE IS TIMELESS, SO ALL ALGORITHMS TAKE O(1).

#203 - "ENCOUNTER WITH A DEPRESSED COMPUTER SCIENTIST" - BY SALVATORE IOVENE, SEP. 7TH, 2009 HTTP://WWW.GEEKHEROCOMIC.COM/