COMP20230: Data Structures & Algorithms
Lecture 4: Complexity Analysis (Big-$\mathcal{O}$)

Dr Andrew Hines

Office: E3.13 Science East
School of Computer Science
University College Dublin

andrew.hines@ucd.ie

Housekeeping

Scribes

If you want to pick a lecture you can use wiki to self-select and organise.
It the wiki gets corrupted or anyone complains I'll just wipe and assign randomly.

Lab

First Lab is **today** at 14:00 in E1.17 SCE

Outline

Today:

- Yesterday: Running time and theoretical analysis, Big-$\mathcal{O}$
- Today: Big-$\mathcal{O}$, Big-Ω (omega) and Big-Θ (theta)

From the tutorial

Estimating Running time analysis can be difficult and subjective – implementation vs pseudocode

Example 1: Constant Running Time

- The simplest function we can think of is the constant function: $f(n) = c$
- For some fixed constant c, such as $c = 5$, $c = 27$, or $c = 210$. For any argument n the constant function assigns the value c. It does not matter what n is, $f(n)$ will always be c.
- The constant function characterises the number of steps to perform a primitive operation (simple arithmetic, assignment, array access etc.) on a computer.

Example 1: Constant Running Time

Algorithm Return the first element of an array

1: function first_element(array):

Input: an array of size n

Output: the first element

2: **return** array[0]   #2 ops: return and access array[0]

How many operations are performed in this function?

What if the list has 10 elements? 1,000 elements?

Independent of input size

Always 2 operations performed (index[0] and return)

Example 2: Linear Running Time

- Given an input n the linear function always assigns the value n itself
  $f(n) = n$
- Arises when we need to perform an operation over n elements (e.g. for loop)
- Represents the best we can achieve for any algorithm which requires reading n elements into memory, since this requires n operations

Example 2: Linear Running Time

Algorithm Return the index of the max value in an array

```
function argmax(array):
Input: an array of size n
Output: the index of the maximum value
index ← 0     #1 op: assignment
foreach i in [1, n-1] do     #2 op per loop
    if array[i] > array[index] then     #3 ops per loop
        index ← i     #1 op per loop, sometimes
    endif
endfor
return index     #1 op: return
```

How many operations if the list has 10 or 10,000 elements?

Number of operations varies proportional to the size of the input list: $6n + 2$

Time in the `foreach` loop gets longer as the input list grows

Example 3: Quadratic Running Time

- For each n assigns the product of n with itself $f(n) = n^2$
- common in algorithm analysis, usually where there are nested loops
- simple nested loops
- loops with changing indices

Beware functions in loops and inner loop ranges

If the function alters the value of the range or if the nested ranges interact!

Example 3: Quadratic Running Time

Algorithm Return possible products of the numbers in an array

```
function possible_products(array):
Input: an array of size n
Output: list of all possible products between elements in the list
products ← []    #1 op: make an empty list
for i in [0, n-1] do    #2 op per loop
    for j in [0, n-1] do    #2 op per loop per loop
        products.append(array[i] * array[j])
                                            #4 ops per loop per loop
    endfor
endfor
return products    #1 op: return
```

How many operations if the list has 10 or 10,000 elements?

Requires about $6n^2 + 2n + 2$ operations

Elements added to list must be multiplied by every other element

Example 4: Logarithmic Running Time

- log function defined as (constant $b > 0$)
  $f(n) = log_b n$
  $x = log_b n$ if and only if $b^x = n$
- computing the logarithm function exactly for any integer n involves the use of calculus, but we can use an approximation that is good enough without calculus
- compute the smallest integer greater than or equal to $log_a n$, since this number is equal to the number of times we can divide n by a until we get a number less than or equal to 1

Example 4: Logarithmic Running Time

Example: Estimating Logs

$log_2 12 = 4$
$12/2/2/2/2 = 0.75 \leq 1$

- This base-2 approximation arises in algorithm analysis, since a common operation in many algorithms is to repeatedly divide an input in half.
- Base 2 logarithm is most common in computer science: $logn = log_2 n$

Example 4: Logarithmic Running Time

Algorithm Return index of a item in an array

```
function binarysearch(myarray, elem):
Input: a sorted array myarray and an element elem
Output: the index of (an) elem in the array or a arbitrary big number
low ← 0
high ← n - 1
while (low <= high) do
    mid ← (low + high) / 2
    if myarray[mid] > elem then
        high ← mid - 1
        else
        if myarray[mid] < elem then
            low ← mid + 1
            else
            return mid
        endif
    endif
return size of myarray + 1 #to show the elem is not in the array
```

Example 4: Logarithmic Running Time

- Given an initial call of `BinarySearch(myarray, elem)`, what is the worst case running time?
- Worst case is if *elem* is not found. Then count how many iteration until we reach the base case ($low > high$)

Algorithm Binary Search

```
function binarysearch(myarray, elem):
Input: sorted array, search element
Output: elemindex
low ← 0
high ← n - 1
while (low <= high) do
    mid ← (low + high) / 2
    if myarray[mid] > elem then
        high ← mid - 1
        else
        if myarray[mid] < elem then
            low ← mid + 1
            else
            return mid
        endif
    endif
return size of myarray + 1
```

Example 4: Logarithmic Running Time

- After iteration number:
  1: there are n/2 items to search in
  2: $(n/2)/2 = n/4$
  k: $(n/2^k)$
- The last step occurs when
- $1 \leq n/2^k$ and $n/2^{k+1} < 1$
- worst case is $\mathcal{O}(logn)$

Algorithm Binary Search

```
function binarysearch(myarray, elem):
Input: sorted array, search element
Output: elemindex
low ← 0
high ← n - 1
while (low <= high) do
    mid ← (low + high) / 2
    if myarray[mid] > elem then
        high ← mid - 1
        else
        if myarray[mid] < elem then
            low ← mid + 1
            else
            return mid
          endif
      endif
return size of myarray + 1
```

Big-$\mathcal{O}$ Rules

If $f(n)$ is a polynomial of degree d, then $f(n)$ is $\mathrm{O}(n^d)$, i.e.

Big-$\mathcal{O}$ Rules

1. Forget about lower-order terms
2. Forget about constant factors
3. Use the smallest possible degree

Example

- It is true that $2n$ is $\mathcal{O}(n^{50})$ – this is not a helpful upper bound
- Instead, we say it is $\mathcal{O}(n)$, by **discarding the constant factor** and **using the smallest possible degree**

Running Time to Big-$\mathcal{O}$ Induction Examples

$T(n) = 7n - 2$

$7n - 2$ is $\mathcal{O}(n)$
Need $c > 0$ and $n_0 \geq 1$ such that $7n - 2 \leq cn$ for $n \geq n_0$
This is true for $c = 7$ and $n_0 = 1$

$T(n) = 3n^3 + 20n^2 + 5$

$3n^3 + 20n^2 + 5$ is $\mathcal{O}(n^3)$
Need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq cn^3$ for $n \geq n_0$
This is true for $c = 4$ and $n_0 = 21$

$T(n) = 3log(n) + 5$

$3log(n) + 5$ is $\mathcal{O}(log(n))$
Need $c > 0$ and $n_0 \geq 1$ such that $3log(n) + 5 \leq clogn$ for $n \geq n_0$
This is true for $c = 8$ and $n_0 = 2$

Estimating Big-$\mathcal{O}$

Simpler Rule for Big-$\mathcal{O}$

Only examine loops / recursion.
A loop over a fixed range [i-n] is typically $\mathcal{O}(n)$.
A loop within a loop is typically $\mathcal{O}(n^2)$
When its not obvious, use induction

Big-Ω (Big-Omega)

Recall that $f(n)$ is $\mathcal{O}(g(n))$ if $f(n) \leq cg(n)$ for some constant c as n grows

Upper Bound: Big-$\mathcal{O}$

Big-$\mathcal{O}$ expresses the idea that $f(n)$ grows no faster than $g(n)$. $g(n)$ acts as an upper bound to $f(n)$'s growth rate

What if we want to express a **lower bound**, i.e., the fact that a function does not grow slower than another one?

Big-Ω (Big-Omega)

Recall that $f(n)$ is $\mathcal{O}(g(n))$ if $f(n) \leq cg(n)$ for some constant c as n grows

Upper Bound: Big-$\mathcal{O}$

Big-$\mathcal{O}$ expresses the idea that $f(n)$ grows no faster than $g(n)$.
$g(n)$ acts as an upper bound to $f(n)$'s growth rate

What if we want to express a **lower bound**, i.e., the fact that a function does not grow slower than another one?

Lower Bound: Big-Ω

We say $f(n)$ is $\Omega(g(n))$ if $f(n) \geq cg(n)$
$f(n)$ grows no slower than $g(n)$

Big-Θ (Big-Theta)

What about an upper and lower bound?

Big-Θ

We say $f(n)$ is $\Theta(g(n))$ `iff` $f(n)$ is **both** $\mathcal{O}(g(n))$ and $\Omega(g(n))$
i.e. $f(n)$ grows the same as $g(n)$ (tight-bound)

Putting some numbers on it...

Example

All this maths can seem a little abstract so what if we had an operation that took 1 nanosecond to execute,
i.e. $T(n) = 1x10^{-9}$ seconds

	Time		
Function	$(n = 10^3)$	$(n = 10^4)$	$(n = 10^5)$
$\log_2 n$	10 ns	13.3 ns	16.6 ns
$\sqrt{n}$	31.6 ns	100 ns	316 ns
n	1 μs	10 μs	100 μs
$n \log_2 n$	10 μs	133 μs	1.7 ms
n^2	1 ms	100 ms	10 s
n^3	1 s	16.7 min	11.6 days
n^4	16.7 min	116 days	3171 yr
2^n	$3.4 \cdot 10^{284}$ yr	$6.3 \cdot 10^{2993}$ yr	$3.2 \cdot 10^{30086}$ yr

What's missing?

- Space Complexity (memory) – will see this later after we study data structures
- Power consumption – we will not look at this, but it is correlated with number of operations
- Examples of Exponential running time and factorial running time – more later

Speed and Efficiency: But what about other algorithm attributes?

Correctness, Security, Robustness, Clarity, Maintainability

What's missing?

- Space Complexity (memory) – will see this later after we study data structures
- Power consumption – we will not look at this, but it is correlated with number of operations
- Examples of Exponential running time and factorial running time – more later

Speed and Efficiency: But what about other algorithm attributes?

Correctness, Security, Robustness, Clarity, Maintainability

Examples of how other evaluation methods

Unit Testing, Penetration Testing, Performance Testing, Code Coverage, Code Reviews

Conclusion

- The Big-$\mathcal{O}$ notation gives an upper bound of the complexity of an algorithm (an algorithm takes at most a certain amount of time)
    - Forget about lower-order terms
    - Forget about constant factors
- Big-Ω is a lower bound (an algorithm takes at least a certain amount of time)
- Big-Θ is an upper and lower bound (tight-bound)

Math to review

Properties of logarithms

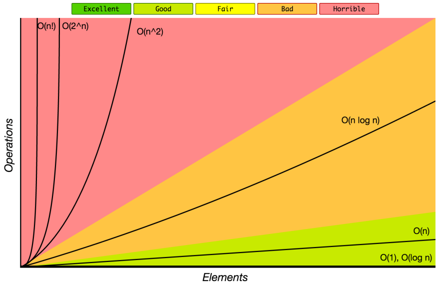
$log_b(xy) = log_b x + log_b y$
$log_b(\frac{x}{y}) = log_b x - log_b y$
$log_b xa = alog_b x \;\; log_b a = \frac{log_x a}{log_x b}$

Properties of exponentials

$a^{(b+c)} = a^b a^c$
$a^{bc} = (a^b)^c$
$\frac{a^b}{a^c} = a^{(b-c)}$
$b = a^{log_a b}$
$b^c = a^{c*log_a b}$

Big-$\mathcal{O}$ Cheat Sheet

Big-O Complexity Chart



Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	O(1)	O(1)	O(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Source: `http://bigocheatsheet.com`