COMP20230: Data Structures & Algorithms
Lecture 3: Running Time and Analysis

Dr Andrew Hines

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

andrew.hines@ucd.ie

Outline

Today:

- Running time and theoretical analysis
- Big-$\mathcal{O}$ notation

Take home message

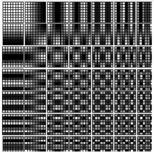
The complexity of an algorithm is given by an analysis of the number of basic operations in the pseudo-code.
Asymptotic analysis for Big-$\mathcal{O}$ allows us to understand the **worst case** algorithm speed.

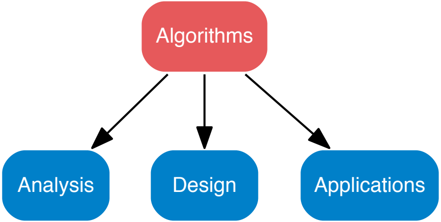
Last week

Big Picture

Problems → D.S. & Algorithms → Programs

Streaming
Bandwidth
Limitations





Attributes of a good algorithm

- **Correctness**
- **Speed**
- **Efficiency**
- Security
- Robustness
- Clarity
- Maintainability

"Does exactly what it says on the tin"

How Correct/Fast/Efficient is an algorithm?

What could we measure?

We could instrument (measure) for many parameters:

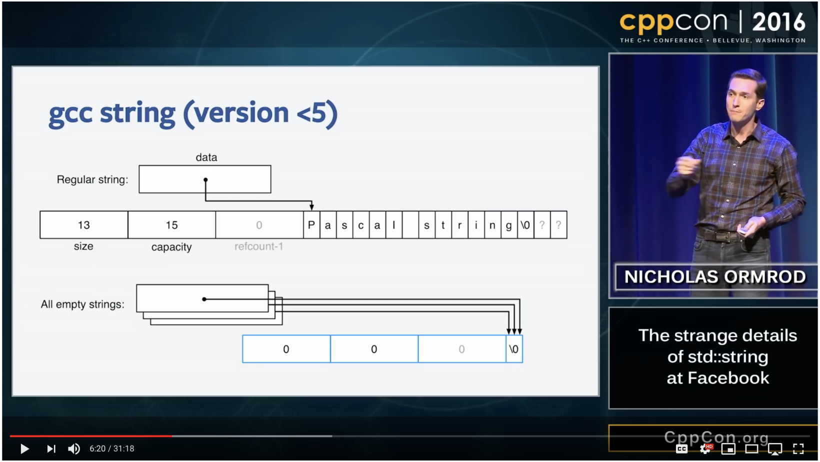
- Memory (how low is low memory usage)?
- Time to execute (smallest amount of time measured on a stopwatch?)
- Power consumption?

We will focus on **efficiency** through looking at the **running time** of algorithms

Why do we care?

Facebook scale

Saving even 1% in production is a massive benefit



https://youtu.be/kPR8h4-qZdk

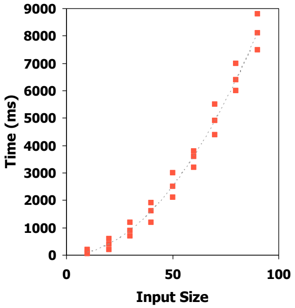
Running Time

- The running time of an algorithm varies with the input and typically grows with the input size
- **Average** case difficult to determine
- In most of computer science we focus on the **worst case** running time
  - Easier to analyse
  - Crucial to many applications: what would happen if an autopilot algorithm ran drastically slower for some unforeseen, untested inputs?

How can we measure running time?

Experimentally

- Write a program to implement algorithm
- Run it for different inputs (quantity and variety)
- Measure the actual running times and plot the results: stop-watch, `mprof`, or debug timestamps



Problems

- You have to implement the algorithm – not always practical!
- Your inputs may not entirely test the algorithm
- The **running time** depends on your **computer's hardware and software** speed

Theoretical Analysis

- Use a **high-level description** of the algorithm (e.g. pseudo-code) instead of an implementation
- Takes into account all possible inputs
- Allows us to evaluate speed of an algorithm **independent of the hardware or software environment**
- By inspecting pseudo-code, we can determine the **number of statements executed** by an algorithm as a **function of the input size**

Pseudocode

Question

Why do we use pseudocode?

Pseudocode

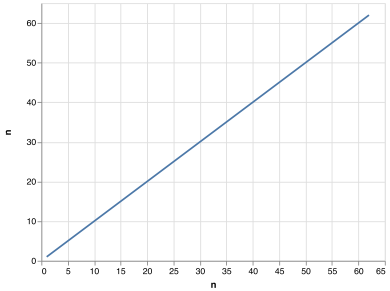
Question

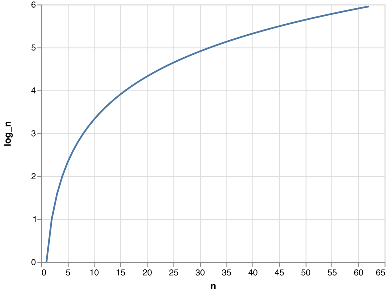
Why do we use pseudocode?

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

(Mathematical) Functions in Algorithm Analysis

Constant	1
Logarithmic	$log n$
Linear	n
N-Log-N	$n log n$
Quadratic	n^2
Cubic	n^3
Exponential	2^n





Primitive Operations

Algorithmic "time" is measured in elementary operations:

- 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, all of these operations take different amounts of time

For the purpose of algorithm analysis, we assume each of these operations takes the same time: "1 operation"

Estimating Running Time

By inspecting the pseudocode/implementation, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

```
1 def find_max(data):
2     '''Return the max element from a non-empty Python list'''
3     biggest = data[0]           # initial max val to beat
4     for val in data             # for each value
5         if val > biggest:       # is it bigger than current biggest
6             biggest = val       # found a new biggest
7     return biggest
```

Step Line : Operations

1: 2 ops, 3: 2 ops, 4: 2n ops, 5: 2n ops, 6: 0 to n ops, 7: 1 op

Running Time gives Growth Rate

Algorithm `find_max` executes $5n + 5$ primitive operations in the worst case, $4n + 5$ in the best case.

Define

$a =$ Time taken by the fastest primitive operation
$b =$ Time taken by the slowest primitive operation

Let $T(n)$ be **worst-case** time of `find_max`.
Then:
$a(4n + 5) \leq T(n) \leq b(5n + 5)$

Hence, the running time $T(n)$ is bounded by two linear functions.

Growth Rate

Changing the hardware/ software environment

- **Does affect** $T(n)$ by a constant factor, but
- **Does not** alter the growth rate of $T(n)$

The linear growth rate of the running time T(n) is an intrinsic property of algorithm `find_max`

Growth Rate Factors

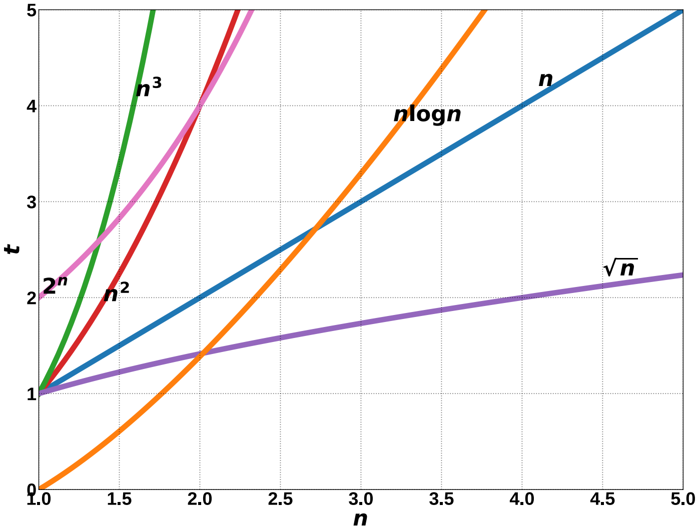
The growth rate is not affected by:

- constant factors or
- lower-order terms

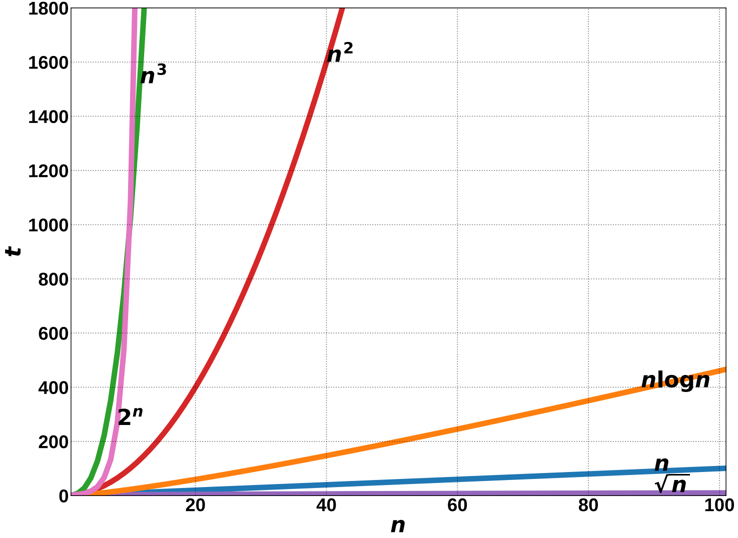
Examples

$10^2 n + 10^5$ is a linear function $10^5 n^2 + 10^8 n$ is a quadratic function

Plotting Running Time



Plotting Running Time



Asymptotic Algorithm Analysis: Big-$\mathcal{O}$

Asymptotic Algorithm Analysis

Big-$\mathcal{O}$ notation gives an upper bound to the growth rate of a running time function.

To perform asymptotic analysis we will:

1. Find the worst-case number of primitive operations executed as a function of the input size
2. We express this function with Big-$\mathcal{O}$ notation
3. Disregard constant factors and lower-order terms when counting primitive operations

Example

We say that algorithm `find_max` "runs in $\mathcal{O}(n)$ time"

Example: 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?

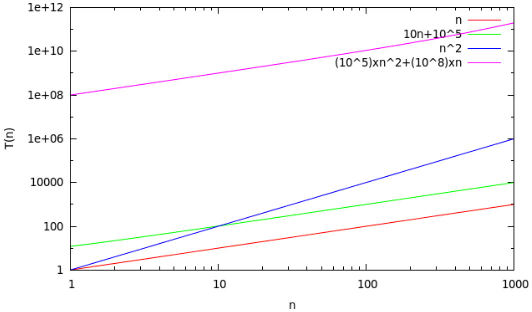
Number of operations varies proportional to the size of the input list: $6n + 2$

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

Summarising Function Growth

For very large inputs, the growth rate of a function becomes less affected by:

- constant factors
- lower-order terms



Examples

- $10^5 n^2 + 10^8 n$ and n^2 both grow with same slope despite differing constants and lower-order terms
- $10n + 10^5$ and n both grow with same slope as well

Big-$\mathcal{O}$ Notation

Given functions $f(n)$ and $g(n)$, we say that:

$f(n)$ is $\mathcal{O}(g(n))$

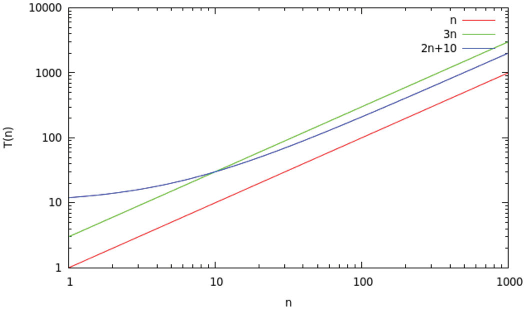
if there are positive **constants** c and n_0 such that

$f(n) \leq cg(n)$ for all $n \geq n_0$

Example: $2n + 10$ is $\mathcal{O}(n)$

$2n + 10 \leq cn$
$(c - 2)n \geq 10$
$n \geq 10/(c - 2)$

Pick $c = 3$ and $n_0 = 10$



Big-$\mathcal{O}$ Notation

Example: n^2 is not $\mathcal{O}(n)$

$n^2 \leq cn$ (divide both sides by n implies)
$n \leq c$

The above inequality cannot be satisfied because c must be a constant, therefore for any $n > c$ the inequality is false.

Pulling it together: Big-$\mathcal{O}$ and Growth Rate

- Big-O notation gives an **upper bound** on the growth rate of a function
- We say "an algorithm is $O(g(n))$" if the growth rate of the algorithm is **no more than** the growth rate of $g(n)$
- We saw on the previous slide that n^2 is not $O(n)$
    - But... n is $O(n^2)$
    - And... n^2 is $O(n^3)$
    - Why?

Pulling it together: Big-$\mathcal{O}$ and Growth Rate

- Big-O notation gives an **upper bound** on the growth rate of a function
- We say "an algorithm is $O(g(n))$" if the growth rate of the algorithm is **no more than** the growth rate of $g(n)$
- We saw on the previous slide that n^2 is not $O(n)$
  - But... n is $O(n^2)$
  - And... n^2 is $O(n^3)$
  - Why?

Because Big-$\mathcal{O}$ is an upper bound!

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

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

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

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
```

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

This is fast and requires less than n comparisons: approx. $log(n)$

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
```

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

This is fast and requires less than n comparisons: approx. $log(n)$

Growth in operations as a result of the input not # lines of code that matters!

Big-$\mathcal{O}$ Rules

If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$, 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**

Constants in Algorithm Analysis

Find the number of primitive operations executed as a function (T) of the input size. For the examples yesterday:

`firstElementofArray`:	$T(n) = 2$
`argmax`:	$T(n) = 5n + 2$
`possible_products`:	$T(n) = 5n2 + n + 3$

In the future we can skip counting operations and replace any constants with c since they become irrelevant as n grows:

`firstElementofArray`:	$T(n) = c$
`argmax`:	$T(n) = c_0 n + c_1$
`possible_products`:	$T(n) = c_0 n^2 + n + c_1$

Big-$\mathcal{O}$ in Algorithm Analysis

Easy to express T in Big-$\mathcal{O}$ by dropping constants and lower-order terms.

In Big-$\mathcal{O}$ notation

`firstElementofArray`	$\mathcal{O}(1)$
`argmax`	$\mathcal{O}(n)$
`possible_products`	$\mathcal{O}(n^2)$

The convention for representing $T(n) = c$ in Big-$\mathcal{O}$ is $\mathcal{O}(1)$.