Starter Questions

Feel free to discuss these with your neighbour:

- How many instructions does it take to compute $8 \times 4$? On ARM? On x86? What about $2^{40} \times 2^{80}$?
- If a program takes a constant amount of time to run is it fast or slow?
- Company A and B have both written programs to solve your problem of processing your data records. You test the programs on 50 records but it will need to work on thousands. Which program do you choose?
What do we mean by Complexity?

Not

- How complicated the text of the program is
- How easy it is to understand the program
- How long the program is

Or anything other than

A measure of the resources required to execute the program

i.e. the complexity of the computation

By default we are talking about time i.e. efficiency
How long does this program take to execute?

\[
\text{result} := 0 \\
\text{while} (n > 0 \land \text{result} = 0) \\
\quad \text{if} \ n \neq 1000 \ \text{then} \ n := n - 1 \ \text{else} \ \text{result} := 1
\]

We need to answer some questions:

- What operations do we count?
- How long does it take to execute those operations? Is this uniform?
- How many times does the loop go round?
How long does this program take to execute?

\[
\text{result} := 0 \\
\text{while} (n > 0 \land \text{result} = 0) \\
\quad \text{if } n \neq 1000 \text{ then } n := n - 1 \text{ else result} := 1
\]

We need to answer some questions:

- What operations do we count? arithmetic, checks, memory
- How long does it take to execute those operations? Is this uniform?
- How many times does the loop go round?
How long does this program take to execute?

```plaintext
result := 0
while (n > 0 ∧ result = 0)
  if n != 1000 then n := n - 1 else result := 1
```

We need to answer some questions:

- What operations do we count? arithmetic, checks, memory
- How long does it take to execute those operations? Is this uniform? Architecture dependent: instruction sets and memory architectures differ. Different operations may take different numbers of cycles.
- How many times does the loop go round?
Analysing Programs

How long does this program take to execute?

\[
\text{result} := 0
\]

\[
\text{while}(\text{n} > 0 \land \text{result} = 0)
\]

\[
\quad \text{if } \text{n} \neq 1000 \text{ then } \text{n} := \text{n} - 1 \text{ else } \text{result} := 1
\]

We need to answer some questions:

- What operations do we count? arithmetic, checks, memory
- How long does it take to execute those operations? Is this uniform? Architecture dependent: instruction sets and memory architectures differ. Different operations may take different numbers of cycles.
- How many times does the loop go round? It depends on the input
How long does this program take to execute?

\[ \text{result} := 0 \]

\[ \text{while}(n > 0 \land \text{result} = 0) \]
\[ \quad \text{if } n \neq 1000 \text{ then } n := n - 1 \text{ else } \text{result} := 1 \]

- If \( n = 1000 \) there is 1 iteration
- If \( n < 1000 \) there are \( n \) iterations
- If \( n > 1000 \) there are \( n - 1000 \) iterations
- If \( n \) is randomly selected from \( 1 - 10000 \) uniformly...
Analysing Programs

How long does this program take to execute?

```
result := 0
while (n > 0 ∧ result = 0)
    if n != 1000 then n := n − 1 else result := 1
```

- If $n = 1000$ there is 1 iteration
- If $n < 1000$ there are $n$ iterations
- If $n > 1000$ there are $n − 1000$ iterations
- If $n$ is randomly selected from $1 − 10000$ uniformly...

\[
\frac{1}{10000} + \left( \sum_{n=1}^{999} \frac{1}{10000} n \right) + \left( \sum_{n=1001}^{10000} \frac{1}{10000} (n − 1000) \right) = \\
\frac{1}{10000} \left( 1 + \frac{999 \times 1000}{2} + \left( \frac{10000 \times 10001}{2} - \frac{1000 \times 1001}{2} \right) + (10000 \times −1000) \right) = \\
\frac{1}{10000} \left( 1 + 499500 + (50005000 − 500500) − 10000000 \right) = 4000.4
\]
How long does this program take to execute?

\[
\begin{align*}
\text{result} & := 0 \\
\text{while} (n>0 \land \text{result} = 0) \\
& \quad \text{if } n \neq 1000 \text{ then } n := n-1 \text{ else } \text{result} := 1
\end{align*}
\]

Do we care about the \textbf{Best} case, the \textbf{Worst} case, the \textbf{Average} case, or the \textbf{Common} case?
Analysing Programs

How long does this program take to execute?

\[
\text{result} := 0 \\
\text{while} (n > 0 \land \text{result} = 0) \\
\quad \text{if } n \neq 1000 \text{ then } n := n - 1 \text{ else } \text{result} := 1
\]

Do we care about the Best case, the Worst case, the Average case, or the Common case?

What is the common case?
When might common mean something different than average?
Can the worst and best cases be the same?
More Counting

```plaintext
z := 0; r := 0;

while x >= 2 do
  s := x; d := 0;
  while 2 ≤ s do (d := d+1; s := s − 2);
  z := z+1; r := r+(z*s); x := d

How many times does the loop go round?

- The outer loop executes $M = \lfloor \log_2(x) \rfloor$ times
- The inner loop executes $N(x) = \lfloor \frac{x}{2} \rfloor$ times
- As $x$ is halving each time, the total number of inner loop executions is

\[
N = \lfloor \frac{x}{2} \rfloor + \lfloor \frac{x}{4} \rfloor + \lfloor \frac{x}{8} \rfloor + \cdots + \lfloor \frac{x}{2^i} \rfloor + \cdots + \lfloor \frac{x}{2^M} \rfloor \approx x(1 - \frac{1}{2^M})
\]

- We have $N + M + 2$ comparisons; $N$ subtractions; $N + 2M$ additions; $M$ multiplications
```
Problems with Counting

- The exact operation counts often end up being extremely complicated;
- It is often necessary to consider many different cases;
- There is no easy way to analyze sub-components of an algorithm and then combine the results for the whole program; and
- Different compilers (or compiler options) might affect the result.
Instead: Capture Order of Growth

We will introduce a notation called the “Big-Oh” notation for abstracting programs in terms of how fast their running times grows with respect to their input size.

This approach is ubiquitous in computer science and software engineering.

Another way of saying the above: You Need to Know This.

Today we meet the idea with examples and formally.

On Friday I will give you some warnings about using it, and cover some extra complexity stuff.
First Observation

What is the relationship between the size of the input of these two program and the number of steps they take?

Program 1

\[
\text{sum := 0; while (n > 0) do (sum := sum+n; n := n - 1)}
\]

Program 2

\[
\text{sum := 0; while (n > 0) do (m := n; while (m > 0) do (sum := sum+1; m := m - 1); n := n - 1)}
\]

Imagine we have two computers A and B that perform addition/subtraction in 100s and 1s respectively.

Which is faster: A running Program 1 or B running Program 1?
What is the relationship between the size of the input of these two program and the number of steps they take?

Program 1

\[
\text{sum} := 0; \text{while } (n > 0) \text{ do } (\text{sum} := \text{sum} + n; \ n := n - 1)
\]

Program 2

\[
\text{sum} := 0; \text{while } (n > 0) \text{ do }
\ (m := n; \text{while } (m > 0) \text{ do } (\text{sum} := \text{sum} + 1; m := m - 1);
\ n := n - 1)
\]

Imagine we have two computers A and B that perform addition/subtraction in 100s and 1s respectively.

Which is faster: A running Program 1 or B running Program 2?
First Observation

Observation

This observation allows us to ignore the constant scaling factor for the time taken.

In other words: if we want constant speedup we can ‘just’ buy a faster machine.
Second Observation

\[ f(x) = \frac{x}{100} \]

\[ \log_2(x) \]

\[ \log_{10}(x) \]
Definition

We say the function $g$ \textit{eventually dominates} function $f$, whenever there exists $k : \mathbb{N}$ such that:

$$\forall (n : \mathbb{N}). \ n > k \Rightarrow g(n) > f(n)$$

In other words, whenever $n$ is greater than $k$, and we have $g(n) > f(n)$, then we can say that $g$ eventually dominates $f$.

Observation

This observation allows us to ignore functions that are eventually dominated by another function.
The polynomial

\[ T(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0 \]

will eventually be dominated by \( Cn^k \), for some \( C > 0 \).

Ignoring the scaling constant, we can simply talk about the function \( n^k \).

**Definition**

If \( f, g : \mathbb{N} \to \mathbb{R}_{\geq 0} \) and \( f \in O(g) \), then there exists \( k \in \mathbb{N} \) and \( C > 0 \) such that for all \( n > k \):

\[ f(n) \leq Cg(n) \]
We will often write $O(1)$ or $O(n)$ to mean $O(f)$ for $f(n) = 1$ or $f(n) = n$. This is lazy and inaccurate but useful.

$O(f)$ is a set of functions and it is important that we understand how this set relates to other sets

We will now establish some important properties and you will prove more in the exercises
Lemma (Reflexivity)

$f \in O(f)$

Proof

Pick $k = 0$ and $C = 1$. Then for all $n \in \mathbb{N}$:

$$f(n) = Cf(n) \leq Cf(n)$$
Relating $O(1)$ and $O(n)$

What does $f \in O(1)$ mean?

**Lemma**

$O(1)$ is a strict subset of $O(n)$ i.e.

$$O(1) \subset O(n)$$

How do we prove this?
Relating $O(1)$ and $O(n)$

What does $f \in O(1)$ mean?

**Lemma**

$O(1)$ is a strict subset of $O(n)$ i.e.

$$O(1) \subset O(n)$$

How do we prove this?

- **Step 1:** show that $O(1) \subseteq O(n)$
- **Step 2:** show that there exists $f \in O(n)$ such that $f \not\in O(1)$
Relating $O(1)$ and $O(n)$

What does $f \in O(1)$ mean?

**Lemma**

$O(1)$ is a strict subset of $O(n)$ i.e.

$O(1) \subset O(n)$

How do we prove this?

- Step 1: show that $O(1) \subseteq O(n)$
  
  Suppose $f \in O(1)$ then $\exists C : f(n) \leq 1C \leq nC$ hence $f \in O(n)$

- Step 2: show that there exists $f \in O(n)$ such that $f \notin O(1)$
Relating \( O(1) \) and \( O(n) \)

What does \( f \in O(1) \) mean?

**Lemma**

\( O(1) \) is a strict subset of \( O(n) \) i.e.

\[ O(1) \subset O(n) \]

How do we prove this?

- **Step 1**: show that \( O(1) \subseteq O(n) \)
  
  Suppose \( f \in O(1) \) then \( \exists C : f(n) \leq 1C \leq nC \) hence \( f \in O(n) \)

- **Step 2**: show that there exists \( f \in O(n) \) such that \( f \notin O(1) \)
  
  Let \( f(n) = n \in O(n) \) and assume \( f \in O(1) \) then
  
  \[ \exists C, k : n > k \rightarrow f(n) \leq C \] but this fails for \( f(\max(C, k) + 1) \)
More Relationships

**Lemma**

Suppose $g(n)$ is a polynomial, i.e.

$$g(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0$$

Provided that the leading coefficient $a_k > 0$, then the set $O(g)$ is the same set as $O(n^k)$

How do we prove this?
More Relationships

Lemma

Suppose $g(n)$ is a polynomial, i.e.

$$g(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0$$

Provided that the leading coefficient $a_k > 0$, then the set $O(g)$ is the same set as $O(n^k)$.

How do we prove this?

This is a standard set equality proof. Assume $f \in O(g)$, which implies certain $k, C$ and use these to construct $k', C'$ such that $f \in O(n^k)$ and vice-versa.
More Relationships

Lemma

Polynomials of order $m$ are strictly contained in polynomials of order $m + 1$

$O(n^m) \subset O(n^{m+1})$

Lemma

The logarithmic class exists strictly between the constant and linear class i.e.

$O(1) \subset O(\log n) \subset O(n)$

(In Computer Science we usually take $\log(n)$ to be $\log_2(n)$.)
*Programs* have an upper bound on their execution time ("quicksort is average-case $O(n \log n)$")

*Problems* have a lower bound ("sorting must be at least linear in $n$ since we must compare each element at least once").
Programs vs Problems

Programs have an upper bound on their execution time ("quicksort is average-case $O(n \log n)$")

Problems have a lower bound ("sorting must be at least linear in $n$ since we must compare each element at least once").
Programs vs Problems

*Programs* have an upper bound on their execution time ("quicksort is average-case $O(n \log n)$")

*Problems* have a lower bound ("sorting *must* be at least linear in $n$ since we must compare each element at least once").

When talking about the complexity of a problem we are talking about the complexity of the fastest possible program to solve that problem.
Related to the “Big-Oh” notation: is the “Big-Omega” notation.

**Definition**

If \( f, g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0} \) and \( f \in \Omega(g) \), then there exists \( k \in \mathbb{N} \) and \( C > 0 \) such that for all \( n > k \):

\[
Cg(n) \leq f(n)
\]

**Definition**

We write \( f \in \Theta(g) \), if, and only if, \( f \in \Omega(g) \land f \in O(g) \)
Related to the "Big-Oh" notation: is the "Big-Omega" notation. In saying \( f \in \Omega(g) \) we are saying that the function \( f \) eventually dominates \( g \); i.e. \( g \) is a lower bound to \( f \).

**Definition**

If \( f, g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0} \) and \( f \in \Omega(g) \), then there exists \( k \in \mathbb{N} \) and \( C > 0 \) such that for all \( n > k \):

\[
Cg(n) \leq f(n)
\]

**Definition**

We write \( f \in \Theta(g) \), if, and only if, \( f \in \Omega(g) \land f \in O(g) \).
Algorithmic Gaps

If the best algorithm for a problem has the same as the problem complexity then we have – in some sense – a near ideal program.
Algorithmic Gaps

If the best algorithm for a problem has the same as the problem complexity then we have – in some sense – a near ideal program.

Alternatively – and much more common – there is an Algorithmic Gap in that the best algorithm is worse than the problem’s lower bound.
Algorithmic Gaps

If the best algorithm for a problem has the same as the problem complexity then we have – in some sense – a near ideal program.

Alternatively – and much more common – there is an Algorithmic Gap in that the best algorithm is worse than the problem’s lower bound.

Examples of problems with no algorithmic gap:
- Finding the maximum in an array: $\Theta(n)$
- Binary search of a sorted list: $\Theta(\log_2(n))$

Examples of problems with algorithmic gaps:
- Minimum Spanning Tree: $O(n^2) \text{ vs } O(n)$
- Sorting: $O(n \log(n)) \text{ vs } O(n)$
Reflective Questions

- Is it easier to show $O(f)$ or $\Omega(f)$?
- What properties of Big-Oh apply to Big-Ω?
- What about functions with more than one input variable?
- What are the limitations of Big-Oh?
- What are the assumptions we have missed?
- What models of computation does this ignore?