Do you remember?

- Big-Oh gives an *asymptotic upper bound* on resource usage
- \( f \in O(g) \) if \( g \) eventually dominates \( f \)
- \( f \in O(g) \) if \( \exists k, C \) such that \( \forall n > k : f(n) \leq Cg(n) \)

Some things to think about:

- Is \( f(n) = \log_2(n) \in O(n) \)? Is \( f(n) = \frac{n(n-1)}{2} \in O(n) \)?
- If \( n < 100 \) is \( n^2 \) or \( 101n \) better?
- How much *time* do I need to compute the first \( n \) primes?
- How much *memory* do I need to compute the first \( n \) primes?
- \( P = NP \)? Or more importantly, what does that mean?
Tomorrow (1st May) I will hold a revision lecture. I will spend a short time reviewing the topics and then answer questions - please email me questions.

Next Tuesday (8th May) Sean will hold a revision lecture. Please send him questions.

Reminder: last week’s Blackboard quiz was postponed to this week; it is due on Friday.
Lecture 10
More Complexity and Asymptotic Analysis
COMP11212

Giles Reger

April 2018
Reminder: Big Oh Notation

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 \( n > k \):

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

Q: What do we do when coming up with the Big-Oh complexity of a program?

A: Find a function \( g \) such that if \( f \) was a function counting the actual operations in the program we would have \( f \in O(g) \).

Hint: We can mostly do this by working out how the ‘size’ of repetitive behaviour (loops/recursions) relates to the input
What is the complexity of this program?

```
sum := 0; n := 10000;
while (n > 0) do (sum := sum+n; n := n−1)
```

The number of operations performed is independent of the size of the input. Therefore the program will always take a constant amount of time and the complexity is $O(1)$. Counting might get us $f(n) = 50002$ and $f \in O(1)$ for $k = 0$, $C = 50002$. 
A Program

What is the complexity of this program?

```
sum := 0; n := 10000;
while (n > 0) do (sum := sum+n; n:= n-1)
```

The number of operations performed is independent of the size of the input. Therefore the program will always take a constant amount of time and the complexity is $O(1)$. 
What is the complexity of this program?

```
sum := 0; n := 10000;
while (n > 0) do (sum := sum+n; n:= n−1)
```

The number of operations performed is independent of the size of the input. Therefore the program will always take a constant amount of time and the complexity is $O(1)$.

Counting might get us $f(n) = 50002$ and $f \in O(1)$ for $k = 0$, $C = 50002$. 
Another Program

What is the complexity of this program?

```plaintext
sum := 0;
while (n > 0) do (sum := sum + n; n := n - 1)
```

The number of operations performed is linearly dependent on the size of the input. Therefore the complexity is $O(n)$. Counting might get us $f(n) = 2 + 5n$ and $f \in O(n)$ for $k = C$. 

Giles Reger
Lecture 10
April 2018
What is the complexity of this program?

```plaintext
sum := 0;
while (n > 0) do (sum := sum+n; n:= n−1)
```

The number of operations performed is \textit{linearly dependent} on the size of the input. Therefore \textit{the complexity is} $O(n)$. 
What is the complexity of this program?

```
sum := 0;
while (n > 0) do (sum := sum+n; n:= n−1)
```

The number of operations performed is linearly dependent on the size of the input. Therefore the complexity is $O(n)$.

Counting might get us $f(n) = 2 + 5n$ and $f \in O(n)$ for $k = 0$, $C = 7$.

$$2 + 5n \leq 7n$$
What is the complexity of this program?

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

The number of operations performed is \textit{linearly dependent} on the size of the input. Therefore the complexity is \(O(n)\).

Counting might get us \(f(n) = 2 + 5n\) and \(f \in O(n)\) for \(k = 0, \ C = 7\).

\[1 \leq n\]
What is the complexity of this program?

```plaintext
sum := 0;
while (n > 0) do (sum := sum+n; n:= n−1)
```

The number of operations performed is **linearly dependent** on the size of the input. Therefore the complexity is \( O(n) \).

Counting might get us \( f(n) = 2 + 5n \) and \( f \in O(n) \) for \( k = 1, \ C = 6 \).

\[
2 + 5n \leq 6n
\]
Another Program

What is the complexity of this program?

```plaintext
sum := 0;
while (n > 0) do (sum := sum+n; n:= n−1)
```

The number of operations performed is linearly dependent on the size of the input. Therefore the complexity is $O(n)$.

Counting might get us $f(n) = 2 + 5n$ and $f \in O(n)$ for $k = 1$, $C = 6$.

$$2 \leq n$$
It is easier to create programs with obvious complexities when using arrays. As well as arrays, let us also add

\[
[\text{for } x = n \text{ to } m \text{ do } P] \equiv [x := n; \text{while}(x \leq m) \text{ do (} P; x := x + 1\text{)]}
\]

This program computes the dot-product of two vectors

\[
\text{dot} := 0; \text{for } i = 1 \text{ to } n \text{ do } \text{dot} := \text{dot} + a[i] \star b[i]
\]

What is its complexity?
Programs with Arrays

It is easier to create programs with obvious complexities when using arrays. As well as arrays, let us also add

\[
\text{[for } x = n \text{ to } m \text{ do } P] \equiv [x := n; \text{ while}(x \leq m) \text{ do } (P; x := x + 1)]
\]

This program computes the dot-product of two vectors

\[
\text{dot} := 0; \text{ for } i = 1 \text{ to } n \text{ do } \text{dot} := \text{dot} + a[i] \star b[i]
\]

What is its complexity?

It performs an addition and multiplication \( n \) times so there are \( 2n \) operations and the complexity as a function of \( n \) is \( f(n) = 2n \). As we ignore the constant scaling factor the complexity is \( O(n) \).
Consider the problem of finding the telephone number of an employee with employee number \( ex \) where we have a \textit{sorted} array of employee numbers \( e \) and an array of their corresponding telephone numbers \( t \).

We could solve this in the following way

\[
\begin{align*}
tel & := -1; \\
\text{for } i = 1 \text{ to } n \text{ do} \\
& \quad \text{if } e[i] = ex \text{ then } tel = t[i] \text{ else skip}
\end{align*}
\]

What is the complexity?
Consider the problem of finding the telephone number of an employee with employee number \( \text{ex} \) where we have a *sorted* array of employee numbers \( e \) and an array of their corresponding telephone numbers \( t \).

We could solve this in the following way:

```plaintext
tel := -1;
for i = 1 to n do
    if e[i] = ex then tel = t[i] else skip
```

What is the complexity?

We can view the body of the loop as constant. The loop goes round \( n \) times so the complexity is \( O(n) \).
Binary Search

The previous linear search is inefficient. Based on the assumption that \( e \) is sorted we can perform binary search.

\[
\text{lo} := 0; \quad \text{hi} := n; \quad \text{mid} := n/2; \quad \text{tel} := -1;
\]

\[
\text{while } \neg (e[\text{mid}] = \text{ex}) \land \neg (\text{lo} = \text{hi}) \text{ do }
\]

\[
(\text{if } e[\text{mid}] \leq \text{ex} \text{ then } \text{lo} := \text{mid}
\]

\[
\text{else } \text{hi} := \text{mid}; \quad \text{mid} := (\text{hi} - \text{lo})/2;)
\]

\[
\text{if } e[\text{mid}] = \text{ex} \text{ then } \text{tel} := t[\text{mid}] \text{ else } \text{tel} := -1
\]

What is the complexity?
The previous linear search is inefficient. Based on the assumption that e is sorted we can perform binary search.

\[
\begin{align*}
\text{lo} &:= 0; \quad \text{hi} := n; \quad \text{mid} := n/2; \quad \text{tel} := -1; \\
\text{while } \neg (e[\text{mid}] = \text{ex}) \land \neg (\text{lo} = \text{hi}) \text{ do} \quad \\
\quad &\begin{cases}
\text{if } e[\text{mid}] \leq \text{ex} \text{ then } \text{lo} := \text{mid} \\
\text{else } \text{hi} := \text{mid}; \quad \text{mid} := (\text{hi} - \text{lo})/2;
\end{cases} \\
\text{if } e[\text{mid}] = \text{ex} \text{ then } \text{tel} := t[\text{mid}] \quad \text{else } \text{tel} := -1
\end{align*}
\]

What is the complexity?

There is a single loop. Everything inside and outside the loop can be considered constant. So we just need to know how many times the loop executes.
The previous linear search is inefficient. Based on the assumption that e is sorted we can perform *binary search*.

\[
\begin{align*}
\text{lo} & := 0; \quad \text{hi} := n; \quad \text{mid} := n/2; \quad \text{tel} := -1; \\
\text{while } & \neg (e[\text{mid}] = \text{ex}) \land \neg (\text{lo} = \text{hi}) \text{ do} \\
& \quad \text{if } e[\text{mid}] \leq \text{ex} \text{ then } \text{lo} := \text{mid} \\
& \quad \quad \text{else } \text{hi} := \text{mid}; \quad \text{mid} := (\text{hi} - \text{lo})/2; \\
\text{if } e[\text{mid}] = \text{ex} \text{ then } \text{tel} := t[\text{mid}] \text{ else } \text{tel} := -1
\end{align*}
\]

What is the complexity?

There is a single loop. Everything inside and outside the loop can be considered constant. So we just need to know how many times the loop executes. The range halves each time.
Binary Search

The previous linear search is inefficient. Based on the assumption that e is sorted we can perform binary search.

\[
\begin{align*}
\text{lo} & := 0; \quad \text{hi} := n; \quad \text{mid} := n/2; \quad \text{tel} := -1; \\
\text{while } & \neg(e[\text{mid}] = \text{ex}) \land \neg(\text{lo} = \text{hi}) \text{ do} \\
& \quad (\text{if } e[\text{mid}] \leq \text{ex} \text{ then } \text{lo} := \text{mid} \\
& \quad \quad \text{else } \text{hi} := \text{mid}; \quad \text{mid} := (\text{hi} - \text{lo})/2;) \\
\text{if } e[\text{mid}] & = \text{ex} \text{ then } \text{tel} := t[\text{mid}] \text{ else } \text{tel} := -1
\end{align*}
\]

What is the complexity?

There is a single loop. Everything inside and outside the loop can be considered constant. So we just need to know how many times the loop executes. The range halves each time. The complexity is \(O(\log_2(n))\).
But how do we get a sorted array? We need to sort it.

This is *bubble sort*. What is its complexity?

```plaintext
if n>1 then
    for i = 0 to n-2 do
        for j = 0 to n-2 do
            if a[j] > a[j+1] then
                t = a[j];
                a[j] = a[j+1];
                a[j+1] = t
```
But how do we get a sorted array? We need to sort it.

This is *bubble sort*. What is its complexity?

```plaintext
if n > 1 then
    for i = 0 to n - 2 do
        for j = 0 to n - 2 do
            if a[j] > a[j + 1] then
                t = a[j];
                a[j] = a[j + 1];
                a[j + 1] = t
```

There are two loops. They both making $n - 2$ iterations so we have $(n - 2) \times (n - 2)$
But how do we get a sorted array? We need to sort it.

This is \textit{bubble sort}. What is its complexity?

\begin{verbatim}
if n>1 then
  for i=0 to n-2 do
    for j = 0 to n-2 do
      if a[j] > a[j+1] then
        t = a[j];
        a[j] = a[j+1];
        a[j+1] = t
\end{verbatim}

There are two loops. They both making \( n - 2 \) iterations so we have \((n - 2) \times (n - 2)\) which is quadratic in \( n \) and the complexity is \( O(n^2) \)
Better Bubble Sort

But how do we get a sorted array? We need to sort it.

This is a better bubble sort (see inner loop). What is its complexity?

```java
if n>1 then
    for i = 1 to n-1 do
        for j = 0 to n-i do
            if a[j] > a[j+1] then
                t = a[j];
                a[j] = a[j+1];
                a[j+1] = t
```
Better Bubble Sort

But how do we get a sorted array? We need to sort it.

This is a better bubble sort (see inner loop). What is its complexity?

```python
if n > 1 then
  for i = 1 to n-1 do
    for j = 0 to n-i do
      if a[j] > a[j+1] then
        t = a[j];
        a[j] = a[j+1];
        a[j+1] = t
```

Now we do \((n - 1) + (n - 2) + \ldots + 1\). Which is \(\frac{n(n-1)}{2}\). This is also \(O(n^2)\) complexity even though the algorithm is obviously more efficient.
Some drawbacks of Big-Oh

Definition

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

$$f(n) \leq Cg(n)$$

But

- $C$ can be very large. Our justification for ignoring $C$ was that we could buy a faster machine. This is not always realistic.

- $k$ can be larger than practically relevant. If $f$ only dominates $g$ for input sizes larger than the ones we care about then $g$ is better for our application.
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
Lower Bounds

Related to the “Big-Oh” notation: is the “Big-Omega” notation.

Definition

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

$$Cg(n) \leq f(n)$$
Related to the “Big-Oh” notation: is the “Big-Omega” notation.

In saying \( f \in \Omega(g) \) we are saying that the function \( f \) \textit{eventually dominates} \( g \); i.e. \( g \) is a \textit{lower bound} to \( f \).

\[
\text{Definition}
\]

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 \( n > k \):

\[
Cg(n) \leq f(n)
\]
If a function has an equal lower and upper bound then it has an exact bound.

**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: $O(n^2)$ vs $O(n)$
- Sorting: $O(n \log(n))$ vs $O(n)$
Space Complexity

So far we have discussed time complexity.

Another resource of interest is space i.e. the amount of memory required to run a program.

Note that space complexity is necessarily less than or equal to time complexity based on the assumption that every bit of memory takes some amount of time to access.

The interesting point here is the space-time-tradeoff i.e. we can trade space for time by precomputing results and storing them.

An example is rainbow tables for reversing cryptographic hash functions. The idea is to precompute the results of brute force in some clever way and then ‘just’ look up the hashed value. Another example is caching.
A famous question in computer science is $P = NP$?

What does this mean?

- $P$ stands for Polynomial time and is the set
  
  $\bigcup_{k} O(n^k)$

- $NP$ stands for Nondeterministic Polynomial time and consists of problems whose solutions can be checked in polynomial time.

The Nondeterministic in NP comes from the fact that such problems can be solved in polynomial time on a non-deterministic machine. Or alternatively, if we can non-deterministically guess an answer we can check it in polynomial time.

We call problems in $P$ tractable and those not in $P$ intractable.
Asymptotic complexity is about guessing a function that approximates (in a well-defined way) the number of operations of a program.

Big-Oh is a useful abstraction but can be practically unhelpful

We care about program’s upper bounds and problem’s lower bounds.

It is also important that we are aware of:

- Space Complexity (space-time-tradeoff)
- Complexity Classes
- What the \( P \) vs \( NP \) question means