Starter Questions

Feel free to discuss these with your neighbour:

- Are there things computers cannot do?
- How do we know what a program will do without running it?
- What does `i = i++ + 1;` do in C?
- Can I tell how long a program will run for before running it?
- Can I tell if it will ever stop running?
- Why should I come to these lectures?
Lecture 0
An Introduction to the Fundamentals of Computation
(Part II)
COMP11212

Giles Reger

March 2017
What are we doing?

We will be looking at three topics:

1. Correctness
2. Complexity
3. Computability

This lecture will introduce and motivate those topics.
But Why?

It is important to note that we are doing Theoretical Computer Science!

Part 1 contained practical tools that every computer scientist should know.

Part 2 includes one such tool (complexity analysis) and a few abstract concepts that every computer scientist should know.
What is Computation?

We can ask this from two perspectives:

1. What is a computer doing?
2. What is happening when we compute, for example, $1 + 2$?

At a suitable level of abstraction, the answer is manipulating symbols.

It is important to point out that there is no universally accepted, well-defined, succinct description of what computation is.

Computability predates Modern Computers

- The concept of computability was introduced by Alan Turing in 1936.
- The Manchester Baby ran its first program on 21 June 1948.
Three Things

1. Computation:

2. Function:

3. Program:
Three Things

1. Computation: The active process of carrying out operations

2. Function:

3. Program:
Three Things

1. Computation: The active process of carrying out operations

2. Function: A mathematical object that transforms inputs to outputs

3. Program:
Three Things

1. Computation: The active process of carrying out operations

2. Function: A mathematical object that transforms inputs to outputs

3. Program: A static description of the operations to perform
Three Things

1. Computation: The active process of carrying out operations

2. Function: A mathematical object that transforms inputs to outputs

3. Program: A static description of the operations to perform

To compute a function we run a program
Three Things

1. Computation: The active process of carrying out operations

2. Function: A mathematical object that transforms inputs to outputs

3. Program: A static description of the operations to perform

To compute a function we run a program

To make programs unambiguous we need a model of computation.
The while Language

We will use a simple programming language as our model of computation.

We will not be using Turing Machines, but we will talk about them.

The while language is a very simple language that is not usable as a real programming language.

But in its favour:

- It contains the core familiar constructs from an imperative language.
- It is simple enough to describe and use in an undergraduate course.
- It is equally expressive as Java, Python, or Turing Machines.
A Sample Program in while

Programs in while look like this

\[
\begin{align*}
  r & := x ; \\
  d & := 0 ; \\
  \textbf{while} & \ y \leq r \ \textbf{do} \ (d := d+1 ; \ r := r - y)
\end{align*}
\]

This program computes the function

\[
f(x, y) = (d, r) = (x \ \text{div} \ y, x \ \text{mod} \ y)
\]

whenever \( y > 0 \land x \geq 0 \)
Questions

1. Does a program $P$ correctly compute a function $f$?
2. Given programs $P_1$ and $P_2$ that both compute a function $f$, which one is the better program?
3. What functions can we compute i.e. are there functions that cannot be computed by any program?
1. Does a program $P$ correctly compute a function $f$? **CORRECTNESS**
2. Given programs $P_1$ and $P_2$ that both compute a function $f$, which one is the better program?
3. What functions can we compute i.e. are there functions that cannot be computed by any program?
Questions

1. Does a program $P$ correctly compute a function $f$? CORRECTNESS
2. Given programs $P_1$ and $P_2$ that both compute a function $f$, which one is the better program? COMPLEXITY
3. What functions can we compute i.e. are there functions that cannot be computed by any program?
Questions

1. Does a program $P$ correctly compute a function $f$? **CORRECTNESS**

2. Given programs $P_1$ and $P_2$ that both compute a function $f$, which one is the better program? **COMPLEXITY**

3. What functions can we compute i.e. are there functions that cannot be computed by any program? **COMPUTABILITY**
Questions

1. Does a program $P$ correctly compute a function $f$? **CORRECTNESS**

2. Given programs $P_1$ and $P_2$ that both compute a function $f$, which one is the better program? **COMPLEXITY**

3. What functions can we compute i.e. are there functions that cannot be computed by any program? **COMPUTABILITY**

The Three C’s of Computer Science
public int f(int[] array){
    int len = array.length;
    int sum = 1
    for(int i=0; i<=len; i++){
        sum = sum*array[i];
    }
    return sum;
}
public int f(int[] array) {
    int len = array.length;
    int sum = 1;
    for (int i = 0; i < len; i++) {
        sum = sum * array[i];
    }
    return sum;
}
Is it Correct? Does this correctly compute the product of the values in the given array?

```java
public int f(int[] array) {
    int len = array.length;
    int sum = 1;
    for (int i = 0; i < len; i++) {
        sum = sum * array[i];
    }
    return sum;
}
```
Is it Correct? Does this correctly compute the product of the values in the given array?

1 public int f(int [] array){
2     int len = array.length;
3     int sum = 1;
4     for(int i=0;i<len;i++){
5         sum = sum*array[i];
6     }
7     return sum;
8 }

We could test the program, but we would theoretically need to consider every possible input.

Instead, in this course we see how to prove correctness.
Does this correctly compute the product...

```java
1  public int f(int[] array) {
2      int product = 1;  boolean isProduct = false;
3      do {
4          product = random.nextInt();
5          isProduct = true;
6          int check = product;
7          for (int i = 0; i < array.length; i++) {
8              if (product % array[i] != 0) {
9                  isProduct = false;  break;
10             }
11          }
12          check = check / array[i];
13      } while (!isProduct);
14      return product;
15  }
```
Correctness Summary

- To answer the correctness question we need a **specification**

- We need to treat **termination** separately

- What I haven’t done is motivate this topic by referring to
  - Arianne 5 where integer overflow caused the loss of 370m USD
  - Intel Pentium 5 where a floating point bug cost Intel over 475m USD
  - Therac-25 where a radiation therapy machine caused at least 5 deaths
  - A 1983 Soviet nuclear early warning system bug almost causing WW3

- Now I have
Which is Better? (they both correctly find the maximum)

```java
public int max(int[] a) {
    int max = a[0];
    int len = a.length;
    for (int i = 1; i < len; i++) {
        if (a[i] > max) {
            max = a[i];
        }
    }
    return max;
}
```

```java
public int max(int[] a) {
    int len = a.length;
    for (int i = 0; i < len; i++) {
        boolean isMax = true;
        for (int j = 0; j < len; j++) {
            if (a[i] < a[j]) {
                isMax = false;
            }
        }
        if (isMax) {
            return a[i];
        }
    }
    // unreachable
    return 0;
}
```
Asymptotic Complexity

- In this course we will meet a special tool for talking about complexity called asymptotic complexity analysis.

- The general idea is to abstract the complexity of a program as a function of its inputs.

- We introduce Big-O notation where \( O(g(x)) = f(x) \) if \( a \times g(x) \) eventually dominates \( f(x) \) i.e. for big enough inputs \( a \times g(x) \) will be bigger than \( f(x) \) for some constant \( a \).

- In the above we call the first max \( O(n) \) and the second max \( O(n^2) \).
We can make anything run in **constant time** if we precompute the results.

Is this cheating?

```java
int count = 0;
int result = 1;
while (count < n) {
    if (isPrime(result)) {
        count++;
    }
    result++;
}

if (n == 1) {
    return 2;
}
if (n == 2) {
    return 3;
}
...
if (n == 1000) {
    return 7919;
}
...
if (n == 10000) {
    return 104729;
}
...
return otherMethod(n);
```
Computability

We will see:

- How to prove that there are uncomputable functions
- The famous Halting Problem as an example of an uncomputable function
- That all sensible models of computation are equivalent, this is known as the Church-Turing thesis
- That such languages are Universal, they can implement each other

But why?

- There is no point trying to write a program for an uncomputable function
- The existence of uncomputable (and intractable) functions motivates the development of techniques to find approximate solutions, or practically useful heuristics
Reflective Questions

- Why do we need an abstract model of computation?
- Why does it matter that programs are unambiguous, why do we want to give formal definitions of their meaning?
- How do I check my programs are correct?
- Why isn’t testing sufficient?
- Are programming languages really equivalent for practical purposes?
- Is the notion of complexity related to the program or the function?
- What if I just extend my language to reduce the number of operations, do I make things faster?
- How does the notion of computable function relate to regular language (hint, it does)
- Who are Church and Turing and why were they so much fun?