My Life as a Performance Analyst

One of the key outcomes of COMP1927 is that students learning not only how to become better programmers, but to begin thinking like computer scientists. To that end – it is essential that we learn how to, and understand the importance of performance analysis. Firstly – what is performance analysis? In short, it is a series of activities undertaken to understand how efficiently something will operate, expressed in terms qualitative and quantitative, abstract and empirical.

What is Program Performance Analysis?

A program is an "expression of an algorithm in a programming language". When we analyse a program, we're dealing with the finished product, a nicely executable and compiled piece of code. As a result, our analysis will have to handled in the context of a specific computer system, specific hardware and such.

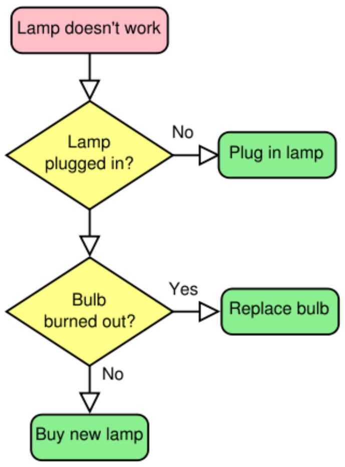
How to do it?

The goal is to measure the "execution costs" or a particular program – what resources are spent by the computer in running the computer. The easiest example of this is time measurements – run the program multiple times, adjusting the inputs and taking multiple time recordings to express how fast the program operates.

When is it useful?

Lots of scenarios! When porting a program across to a new platform (e.g. mobile device, operating system, programming language), or when the underlying algorithms are difficult to analyse. Really any time that the performance of a program is important to us (and as computing resources are finite, it usually is). By understanding how efficient programs are, and under what circumstances they perform efficiently – we can make decisions on which program/platform combinations to use; and based on certain conditions, where performance improvements are most needed.

What is Algorithm Performance Analysis?



An algorithm is a "description of a computational process". When we analyse an algorithm, we are studying the underlying mathematics of a particular program. We're explicitly not considering how this algorithm will perform in other languages, on other systems, other compilers or specifications – just the intrinsic performance of the algorithm itself.

How to do it?

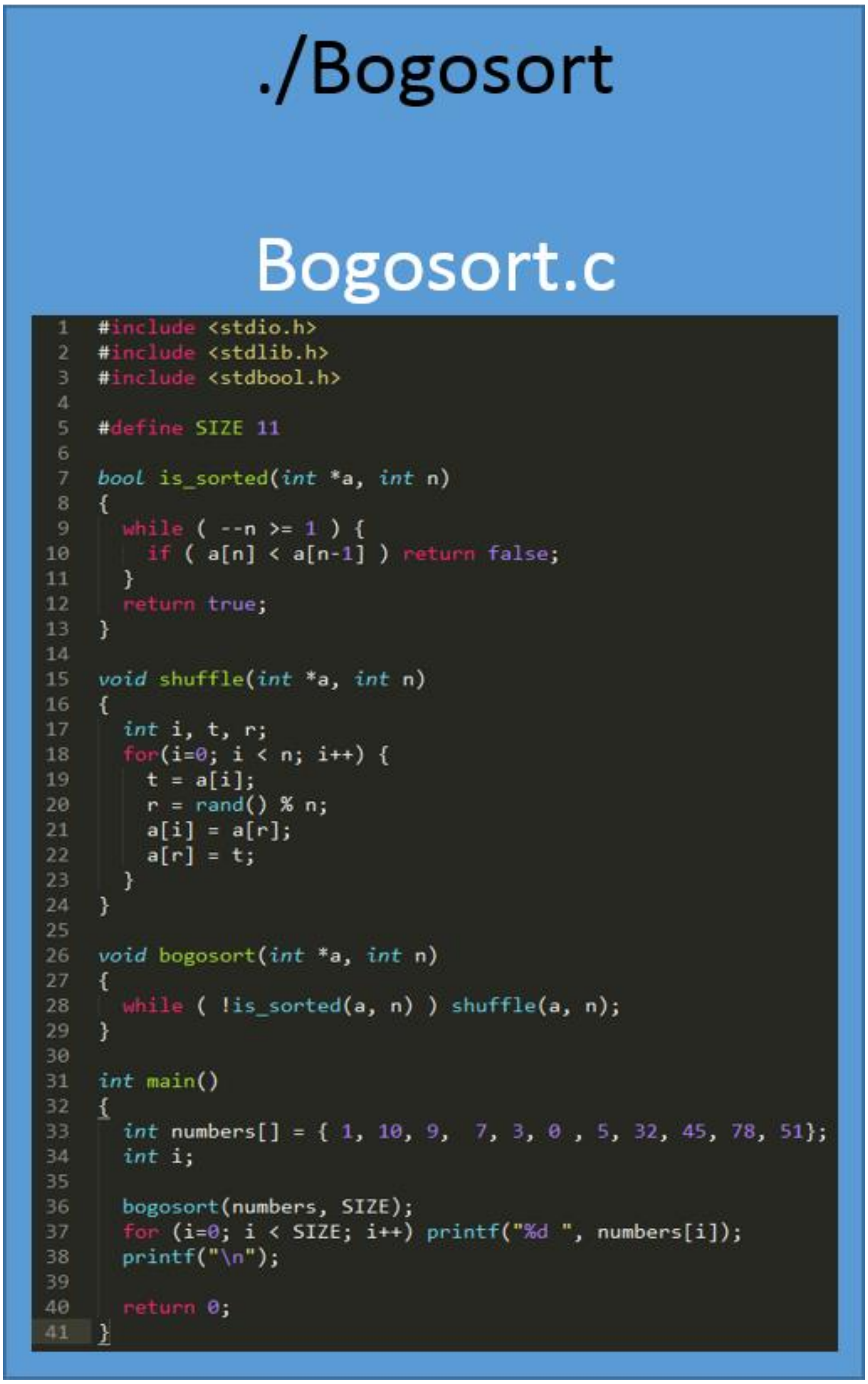
Rather than pursue empirical results, to analyse an algorithm we look at the source code, identify the core operations being conducted (e.g. swap, comparison), and step through the algorithm by hand to see how many operations are performed. For most algorithms, the number of operations performed will depend on some input (e.g. no. of items in an array), so rather than count how many operations will be performed for various sizes, we consider how to number of operations performed will grow as the size of the input tends to infinity. We normally express this in "Big O" notation, $O(n)$ for example would indicate that each additional unit of input results in a linear increase in the number of operations executed.

When is it useful?

Again – lots of scenarios. Algorithm performance analysis is particularly useful when deciding between different types of algorithms. For example, you may want to know whether mergesort or bogosort will be more efficient. Without needing the write the entire program, you can analysis the algorithm in advance, saving on programming time (hint: bogosort is ALMOST always the absolute least efficient option).

What is the difference between the two types of analysis?

If what I've said so far hasn't made the difference between the two clear, consider this: Bogosort[1].



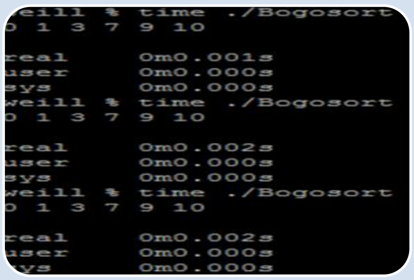
```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define SIZE 11

bool is_sorted(int *a, int n)
{
  while ( --n >= 1 ) {
    if ( a[n] < a[n-1] ) return false;
  }
  return true;
}

void shuffle(int *a, int n)
{
  int i, t, r;
  for(i=0; i < n; i++) {
    t = a[i];
    r = rand() % n;
    a[i] = a[r];
    a[r] = t;
  }
}

void bogosort(int *a, int n)
{
  while ( !is_sorted(a, n) ) shuffle(a, n);
}

int main()
{
  int numbers[] = { 1, 10, 9,  7, 3, 0 , 5, 32, 45, 78, 51};
  int i;

  bogosort(numbers, SIZE);
  for (i=0; i < SIZE; i++) printf("%d ", numbers[i]);
  printf("\n");

  return 0;
}
```

Firstoff – Bogosort is terrible. But why do we know that it is terrible? When we analyse the code inside the black box, when we think about the steps the computer is taking – that's algorithm analysis. When we look at the blue box – the compiled Bogosort, and take measurements and observations of how it performs on a particular computer – that's program analysis.

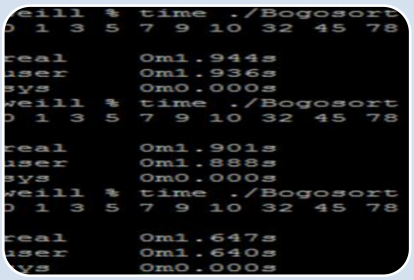
[1] Code modified from rosettacode.org http://rosettacode.org/wiki/Sorting_algorithms/Bogosort

Give an example of how to analyse the performance of a program

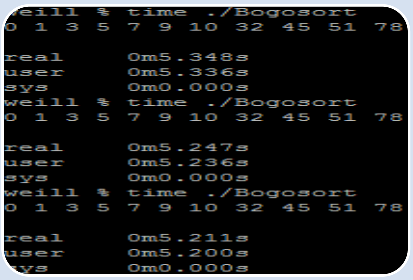
So first, we compile the program. We then run it – this program conveniently includes its own array of integers, so in this case we're just going to take a couple of readings of how long the program takes to run, and then analyse the results.



```
eill % time ./Bogosort
0 1 3 7 9 10

real     0m0.001s
user     0m0.000s
sys      0m0.000s
weill % time ./Bogosort
0 1 3 7 9 10

real     0m0.002s
user     0m0.000s
sys      0m0.000s
weill % time ./Bogosort
0 1 3 7 9 10

real     0m0.002s
user     0m0.000s
ys       0m0.000s
```



```
eill % time ./Bogosort
0 1 3 5 7 9 10 32 45 78

real     0m1.944s
user     0m1.936s
sys      0m0.000s
weill % time ./Bogosort
0 1 3 5 7 9 10 32 45 78

real     0m1.901s
user     0m1.888s
sys      0m0.000s
weill % time ./Bogosort
0 1 3 5 7 9 10 32 45 78

real     0m1.647s
user     0m1.640s
ys       0m0.000s
```



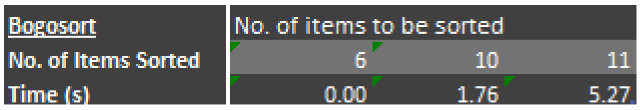
```
eill % time ./Bogosort
0 1 3 5 7 9 10 32 45 51 78

real     0m5.348s
user     0m5.336s
sys      0m0.000s
weill % time ./Bogosort
0 1 3 5 7 9 10 32 45 51 78

real     0m5.247s
user     0m5.236s
sys      0m0.000s
weill % time ./Bogosort
0 1 3 5 7 9 10 32 45 51 78

real     0m5.211s
user     0m5.200s
ys       0m0.000s
```

At this stage, it may be tempting to think that Bogosort is pretty fast. Given 6 inputs, it only took on average, 0.002 seconds to sort.

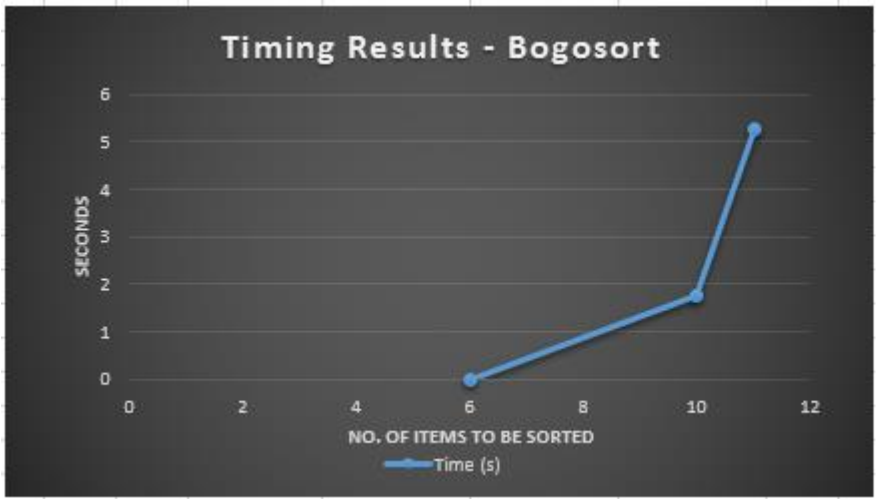
Let's try 10 inputs this time. A huge increase in time, but I wonder if it will perform differently with more inputs?

So it more than doubled in time by adding one input? This is a terrible program.

Let's look at our results:



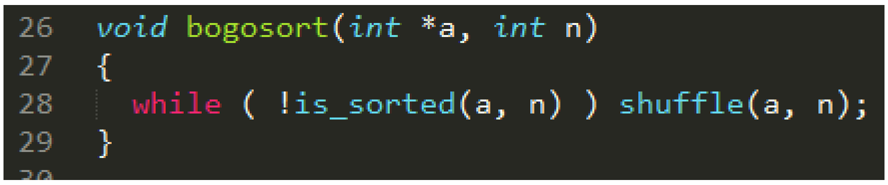
Bogosort	No. of items to be sorted		
No. of Items Sorted	6	10	11
Time (s)	0.00	1.76	5.27



We now have empirical evidence that confirms Bogosort is terrible. Just terrible. Now, if this were a less terrible program, we'd also try different types of inputs (e.g. pre-sorted data). In fact, Bogosort is exceptional at sorting pre-sorted data (i.e doing nothing). But the question now is – why is Bogosort terrible?

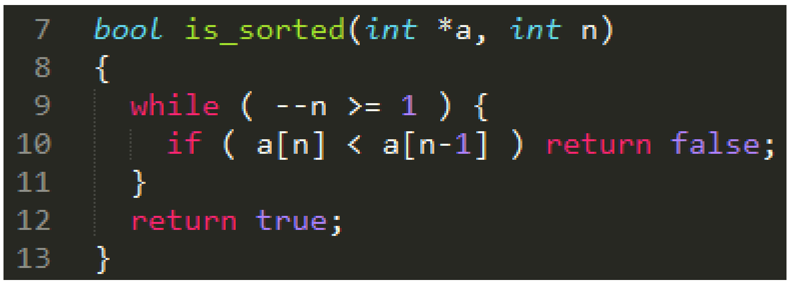
Give an example of how to analyse the performance of an algorithm

Let's have another look at that code from before. Starting in main, we can see that there's a function bogosort within main, so we'll go there. Looking at bogosort – there's an interesting while loop:



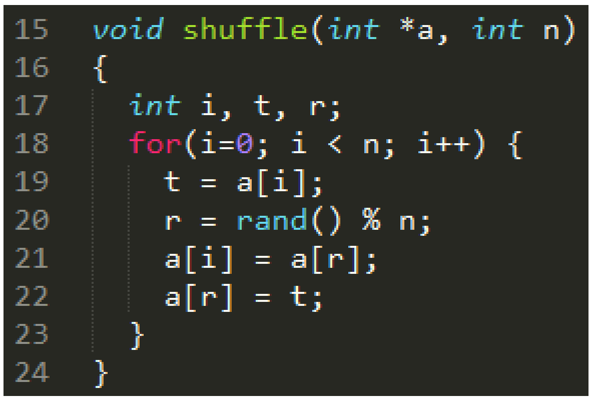
```
26  void bogosort(int *a, int n)
27  {
28    while ( !is_sorted(a, n) ) shuffle(a, n);
29  }
```

Let's look first at is_sorted.



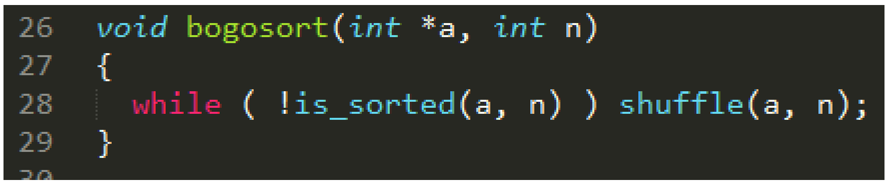
```
 7  bool is_sorted(int *a, int n)
 8  {
 9    while ( --n >= 1 ) {
10      if ( a[n] < a[n-1] ) return false;
11    }
12    return true;
13  }
```

This looks like a fairly innocuous loop to check if the array is sorted. It performs a comparison for n – 1 inputs. We can generalise that performance as linear, so in Big O notation: $O(n)$. Interestingly, we can see that if the array input is already sorted prior, then the program as a whole will have performance $O(n)$. Now let's look at the other function from is_sorted, shuffle.



```
15  void shuffle(int *a, int n)
16  {
17    int i, t, r;
18    for(i=0; i < n; i++) {
19      t = a[i];
20      r = rand() % n;
21      a[i] = a[r];
22      a[r] = t;
23    }
24  }
```

Another loop here – it calls a function rand() on each of the n inputs, randomly changing their order. Even though there are a couple of operations here (lines 19-22), they will each execute once per loop, so we ignore this constant multiple, and still consider the algorithm as being linear, coming to be $O(n)$ (think of it as the constant multiple being irrelevant as n tends towards infinity. Now we don't know how rand() operates as it comes from another file, but for arguments sake we'll assume that it's a single operation. So back to bogosort:



```
26  void bogosort(int *a, int n)
27  {
28    while ( !is_sorted(a, n) ) shuffle(a, n);
29  }
```

At this stage its painfully clear what the problem is – the bogosort loop will run until the randomised array is sorted. Because that process is random, and there's only a probability that the array will eventually be sorted, we can see that this algorithm as a whole, is incredibly inefficient. For larger enough arrays, it may take longer to sort than time until the end of the universe! But thanks to algorithm analysis we now know never, ever to use bogosort.