COMP2720: Automating Tools for New Media

Complexity: Not all programs are created equal, and some will never finish...

More than one way to solve a problem

- There's always more than one way to solve a problem.
  - To go from the CSIT building to the COMP2720 lecture room you can cut across the creek OR you can walk via the Union.
- Some solutions are better (faster, shorter, less muddy) than others.
- How do you compare them?

Our programs (functions) implement algorithms

- *Algorithms* are descriptions of behavior for solving a problem.
- Programs (functions for us) are executable interpretations of algorithms.
- The same algorithm can be implemented in many different programming languages.

Recall these two functions

```
def half(filename):
  source = makeSound(filename)
  target = makeSound(filename)

  sourceIndex = 1
  for targetIndex in range(1, getLength( target)+1):
    setSampleValueAt( target, targetIndex,
            getSampleValueAt( source,
            int(sourceIndex)))
    sourceIndex = sourceIndex + 0.5

  play(target)
  return target
```

```
def copyBarbsFaceLarger():
  # Set up the source and target pictures
  barbf=getMediaPath("barbara.jpg")
  barb = makePicture(barbf)
  canvas = makeEmptyPicture(640,480)
  # Now, do the actual copying
  sourceX = 45
  for targetX in range(100,100+((200-45)*2)):
    sourceY = 25
    for targetY in range(100,100+((200-25)*2)):
      color = getColor(
              getPixel(barb,int(sourceX),int(sourceY)))
      setColor(getPixel(canvas,targetX,targetY), color)
      sourceY = sourceY + 0.5
    sourceX = sourceX + 0.5
  show(barb)
  show(canvas)
  return canvas
```

Both of these functions implement a *sampling algorithm*

- Both of them do very similar things:
  - Get an index to a source.
  - Get an index to a target.
- For all the elements that we want to process:
  - Copy an element from the source at the integer value of the source index to the target at the target index.
  - Increment the source index by 1/2.
- Return the target when completed.

This is a description of the algorithm.

How do we compare algorithms?

- There's more than one way to sample.
  - How do we compare algorithms to say that one is faster than another?
- Computer scientists use something called *Big-O notation.*
  - It's the *order of magnitude* of the algorithm.
  - The goal is to describe what happens to the running time of the algorithm as the size of the input grows.
- Big-O notation tries to ignore differences between programming languages, even between compiled vs. interpreted, and focus on the number of steps to be executed.

Which one of these is more complex in Big-O notation?

```
def increaseRed(picture):
  for p in getPixels(picture):
    value=getRed(p)
    setRed(p,value*1.2)
```

```
def increaseVolume(sound):
  for sample in getSamples(sound):
    value = getSample(sample)
    setSample(sample,value*2)
```

It's hard to answer this question because getPixels and getSamples hide some of the complexity.

Spelling out the complexity

```
def increaseRed2(picture):
  for x in range(1,getWidth(picture)):
    for y in range(1,getHeight(picture)):
      px = getPixel(picture,x,y)
      value = getRed(px)
      setRed(px,value*1.1)
```

```
def increaseVolume2(sound):
  for sample in range(1,getLength(sound)):
    value = getSampleValueAt(sound,sample)
    setSampleValueAt(sound,sample,value * 2)
```

Call these bodies each (roughly) one *step*. Of course, it's more than one, but it's a *constant* difference—it doesn't vary depending on the size of the input.

Does it make sense to clump the body as one step?

- Think about it as the sound length increases, or the size of the picture increases.
- Does the body of the loop take any longer?
  - Not really.
- Then where does the time go? In the looping.
  - In applying the body of the loop to all those samples or all those pixels.

Loops are multiplicative

```
def loops():
  count = 0
  for x in range(1,5):
    for y in range(1,3):
      count = count + 1
      print x,y,"--Ran it", count,"times"
```

```
>>> loops()
1 1 --Ran it  1 times
1 2 --Ran it  2 times
2 1 --Ran it  3 times
2 2 --Ran it  4 times
3 1 --Ran it  5 times
3 2 --Ran it  6 times
4 1 --Ran it  7 times
4 2 --Ran it  8 times
```

Complexity in Big-O notation

- In general, $O(n)$ algorithms are less complex than $O(n^2)$ which are less than complex than $O(n^3)$.
- The code to increase the volume will execute its body *(number of samples)* times.
  - If we call that *n*, we say that's order *n* or $O(n)$.
- The code to increase the red will execute its body *(the number of pixels)* times.
  - You might think that the body is executed *O(width*height)* times.
  - We would still call this $O(n)$ because we address each pixel only once.
  - That explains why smaller pictures take less time to process than larger ones (you're processing fewer pixels (smaller *n*) in a smaller picture).

Not all algorithms are the same complexity

- There is a group of algorithms called *sorting algorithms* that place things (numbers, names) into a sorted sequence.
- Some of the sorting algorithms (insertion sort, selection sort, bubble sort and others) have complexity around $O(n^2)$.
  - If the list has 100 elements, it'll take about 10,000 steps to sort them.
- However, others (quick sort, merge sort) have complexity $O(n \log n)$.
  - The same list of 100 elements would take only 460 steps.
- Think about the difference if you're sorting your 100,000 customers…

Finding something in a dictionary

- *O(n)* algorithm.
  - Start from the beginning.
  - Check each page, until you find what you want.
- Not very efficient.
  - Best case (you find the word on the first page you check): One step.
  - Worse case (it's on the last page in the dictionary): n steps where *n* = number of pages.
  - Average case: *n/2* steps.

Implementing a *linear search* algorithm

```
def findInSortedList(something, alist):
 for item in alist:
  if item == something:
   return "Found it!"
 return "Not found"
```

```
>>> findInSortedList
    ("bear",["apple","bear","cat","dog","ele
    phant"])
'Found it!'
>>> findInSortedList
    ("giraffe",["apple","bear","cat","dog","e
    lephant"])
'Not found'
```

This program implements that style of search for searching a list.

Digression on the underlying hardware

- Analysis of algorithms efficiency is based on a certain assumption about how the computation process is organised ***physically,*** *and therefore it ultimately depends on the laws of physics which govern computers*
- *All algorithms which computer scientists have used so far were* ***classical*** because computers which performed those algorithms were classical, too
- But deep down the small scale Nature obeys the **quantum laws**, and in recent years scientists were thinking how this would change the computation. The changes can be very **dramatic** (but the Quantum Computer is yet to be built).
- Eg, the search in an unordered array becomes $O(\sqrt{N})$

Back to classics
A better search in a sorted list

- *O(log n)* ($\log_2 n = x$, where $2^x = n$).
  - Split the dictionary in the middle.
  - Is the word you're at *before* or *after* the page you're looking at?
    - If *after*, look from the *middle* to the *end*.
    - If *before*, look from the *start* to the *middle*.
  - Keep repeating until done or until it couldn't be there.
- More efficient:
  - Best case: It's there in the first place you look.
  - Average and worst case: *log n* steps.

Implementing a binary search

```
def findInSortedList(something, alist):
  low = 0
  high = len(alist) - 1
  while (low <= high):
    middle = int((low+high)/2.0)
    printNow("Checking at: "+str(middle)+" low="+str(low)+" high="+str(high))
    if alist[middle]==something:
      return "Found it!"
    if alist[middle]<something:
      low = middle+1
    if alist[middle]>something:
      high = middle-1
  return "Not found"
```

While there's any more pages to search…

Running the binary search

```
>>> findInSortedList('kitten',['apple','bear','cat','dog','elephant','fox'])
Checking at: 3 low=1 high=5
Checking at: 4 low=4 high=5
Checking at: 5 low=5 high=5
'Not found'
>>> findInSortedList('cat',['apple','bear','cat','dog','elephant','fox'])
Checking at: 3 low=1 high=5
Checking at: 1 low=1 high=2
Checking at: 2 low=2 high=2
'Found it!'
>>> findInSortedList('fox',['apple','bear','cat','dog','elephant','fox'])
Checking at: 3 low=1 high=5
Checking at: 4 low=4 high=5
Checking at: 5 low=5 high=5
'Found it!'
```

Thought experiment: Optimise your song

- You're writing a song that will be entirely generated by computer by assembling portions of other sounds.
  - Splicing one onto the end of the other (as dome in labs).
- You've got a bunch of short sounds, say 60.
- You want to try every combination of these 60 sounds.
- You want to find the combination that:
  - Is less than 2 minutes 30 seconds (optimal radio time).
  - And has the right amount of high and low volume sections (assume you've got a checkSound() function to do that).

How many combinations are there?

- Let's ignore order for right now.
- Let's say that you've got three sounds *a*, *b*, and *c*.
  - Your possible songs are: *a, b, c, ab, ac, bc, abc*
- Try 2 and 4, and you'll see the same pattern we saw earlier with *bits*.
- For *n* things, every combination of in-or-out is 2^n.
  - If we ignore the empty combination, it's $2^n - 1$.

Handling our 60 sounds

- Therefore, our 60 sounds will result in 2^{60} combinations to test against our desired time and our time check.
- That's *1,152,921,504,606,846,976* combinations!
- Let's imagine that we can test each one in only a single machine instruction (unbelievable, but pretend).
  - On a 1.5 GHz laptop, that's still *768,614,336* seconds!

Spelling it out

- *768,614,336* seconds is *12,810,238* minutes.
  - *12,810,238 / 60* is *213,504* hours.
  - Divided by *24* is *8,896* days.
  - Which is *24* years!
- But since *Moore's Law* doubles the processor speed every 18 months, we'll be able to cut that down to 12 years in just a year and a half!
- If we cared about order, too (*abc* vs. *bca* vs. *cba*…) we'd have to multiply the number of combinations by 7 followed by 63 zeros: *7,000, ... (then 60 more 0s)*.

Optimisation is a very complex problem

- Trying to find the optimum combination of a set of things turns out to have simple algorithms for it.
- But it *always* takes a very long time.
- Other problems seem like they should be do-able, but they aren't.

The *Traveling Salesman Problem*

- Imagine that you're a sales person, and you're responsible for a bunch of different clients.
  - Let's say 30 — half the size of our optimisation problem.
- To be efficient, you want to find the *shortest path* that will let you visit each client exactly once, and not more than once.
- Being a smart ANU student, you decide to write a program to do it.

The *Traveling Salesman Problem* currently can't be solved

- The best known *algorithm* that gives an optimal solution for the *Traveling Salesman Problem* is *O(n!)* (that's factorial).
  - There are algorithms that are better that give close-to but not-guaranteed-best paths (heuristics).
- For 30 cities, the number of steps to be executed is *265,252,859,812,191,058,636,308,480,000,000* (30!)
- The *Traveling Salesman Problem* is real.
  - For example, several manufacturing problems are actually based on this problem, e.g. moving a robot on a factory floor to process things in an optimal order.

Class P, Intractable, and Class NP

- Many problems (like sorting) can be solved with an order of complexity that's a *polynomial*, O(n^p)*: Class P Problems*
- Other problems, like optimisation, have known solutions but are so hard and big that we know that we just can't solve them in a reasonable amount of time for even reasonable amounts of data: Intractable Problems (no solution)
- Still other problems, like the *Traveling Salesman Problem, set partitioning, the Frobenius (coin/stamp) problem(?)* etc seem intractable, but there's a method which requires only polynomial number of operations to check whether any sample solution is indeed the solution to the problem. In other words: the general solution of the P Class is unknown, yet correctness of any solution can be checked with an algorithm of P Class: Class NP Problems
- Is *P=NP*? BIG QUESTION! (Million $ Prize from The Clay Institute)

The Stamp/Coin Problem

- First considered by J. J. Sylvester, and known as Frobenius problem
- You have two coins, a_1, a_2 which are mutually prime, *ie,*

$$gcd(a_1, a_2) = 1$$

- The sum of money which can be repre-sented by **any combination** of a_1 and a_2

$$N = \sum_{i=1}^{n} a_i \cdot x_i, \ \forall x_i \geq 0$$

- Now the problem: find N_{max} when the above representation is not possible, *ie*, there is no solution --- **all** sums greater than N_{max} can be represented with those coins, but *some* numbers smaller then N_{max} cannot ("no change").

Is Stamp/Coin problem NP-complete?

- Sylvester found the solution for two coins/stamps:

$$g(a_1, a_2) = a_1 \cdot a_2 - a_1 - a_2$$

- For any number p of coins/stamps greater than 2, there is no explicit formula, **but** there is a well defined algorithm of complexity $O(N^p)$ (*N* is the sum of all digits in denomination values of all *p* coins/stamps)
- It is proved that the general Frobenius problem (for all numbers *p* at once) is NP-complete: a subset of all NP problems, in which if an efficient solution of one of them is found it is also applicable to all problems from this subset.

Then there are impossible problems

- There are some problems that are provably impossible.
  - We know that no algorithm can ever be written to solve this problem.
- The most famous of these is the *Halting Problem.*
  - Which is, essentially, to write a program to completely understand and debug another program.

The *Halting Problem*

- We've written programs that can read another program and even write a new program.
- Can we write a program that will input another program (say, from a file) and tell us if the program will ever stop or not?
  - Think about while loops with some complex expression—will the expression ever be false?
  - Now think about nested while loops, all complex…
- It's been *proven* that such a program can never be written.

Alan Turing

- Brilliant mathematician and computer scientist.
- Came up with a mathematical definition of what a computer could do… before one was even built!
  - The *Turing machine* was invented in answer to the question of what the limits of mathematics were: What is computable by a function?
- He proved that the *Halting Problem* had no solution in 1936 — almost ten years before the first computers were built.

Why is **PhotoShop** faster than **Python**?

- First, **PhotoShop** is compiled.
  - Compiled programs run faster than interpreted programs.
- Second, **PhotoShop** is optimised.
  - Where things can be done smarter, **PhotoShop** does it smarter.
  - For example, finding colors to be replaced can be made faster than the linear search that we used.

Can we write a program that thinks?

- Are human beings computable?
- Can human intelligence be captured in an algorithm?
  - Yes, we *can* debug programs, but there may be some programs that are too complex for humans to debug — we may fall under the *Halting Problem*, too.
- Is it *Class P*? *Class NP*? *Intractable*?
- Are humans just computers in flesh?
  - These are questions that *artificial intelligence researchers* and *philosophers* study today.

What to do now

- Assignment 2 is due this Tuesday 21 October 6 pm!
- Lab 6 on Visual Python is now out, **do it in advance** to be ready with the results this week.