MATH40006: An Introduction To Computation
Module notes, Section 11

These notes, together with all the other resources you'll need, are available on Blackboard, at

`https://bb.imperial.ac.uk/`

11 Algorithms and complexity

11.1 Algorithms

What is a computer program? One one level, it's a set of instructions written in a particular language, telling a particular computer to do a particular thing. But computer programs are embodiments of something more abstract and general: an **algorithm**.

An algorithm is a finite sequence of instructions which, if followed, will solve a particular problem in finite time. An algorithm is something that doesn't depend on a particular choice of programming language: a concrete computer program, written in a particular language, may be said to **implement** an algorithm.

For example, consider the problem of searching a data structure to see whether it contains a particular data item. Here, shorn of all docstrings and comments, is how we might implement that in Python:

```
def search(data, element):

    for x in data:

        if x == element:
            return True

    return False
```

In Matlab, we might write:

```
function isThere = search(data, element)

for x = data
    disp(x)
    if x == element
        isThere = 1;
        return
    end
```

```
end

isThere = 0;
end
```

In Maple:

```
search := proc(data, element)

    local x;

    for x in data do

        if x = element then
            return True;
        end if

    end do;

    return False;

end proc
```

In the Wolfram Language:

```
search[data_, element_] :=
 Module[{isThere = False},
  Do[If[data[[i]] == element, isThere = True; Break[]], {i, 1,
    Length[data]}]; isThere]
```

and so on. Widely differing programming languages; just one algorithm. That algorithm: search through `data` term by term, and if you encounter `element` give the output `True`; if you never encounter `element`, return `False`.

Notice that this process must terminate in finite time: no collection of data is infinite, and at some point we will either encounter what we're looking for or use up all the data. This is one of the things that makes it an algorithm.

11.2 Pseudocode

It's useful to have a way of describing an algorithm that doesn't depend on the particular programming language we're working with. Natural language (as above) is one such way, but it has the drawback of being a bit imprecise. Another is **pseudocode**: a way of laying out the parts of an algorithm that's informal, but not quite so informal as natural language.

Pseudocode is meant to be human-readable, and has no strict rules. Our search algorithm, in a typical pseudocode, might look something like this:

```
algorithm search
    input:  1D array data, particular data item element
    output:  boolean value True or False

    for each x in data do
        if x = element
            return True, then stop

    otherwise return False
```

For all that algorithms are independent of choice of language, this is definitely "Python-style" pseudocode, with indenting being used to delineate blocks, and so on; people tend to write pseudocode that looks a bit like the code they're used to. However, it tends to be less bogged down in the details of the implementation, and (to really take the pressure off) it doesn't have to work as such; it only needs to make sense to the reader.

11.3 Counting operations: big O, big Ω

Let's consider again the problem of searching a list for a particular data item, and now let's make the assumption that our data is **sorted**: that it's in ascending order. We could still use our `search` function, but we'd be silly to. It would be better to use that information to speed up the search. Here's some pseudocode for a function to do that:

```
algorithm binary_search
    input:  1D array data, particular data item element
    output:  boolean value True or False

    if no data remaining
        return False
    else
        middle_item := item at midpoint of data
        if middle_item = element
            return True
        else if middle_item > element
            return binary_search(left half, element)
        else
            return binary_search(right half, element)
```

Notice that is is a **recursive** implementation; this feels like a natural for such an approach.

Here's what it looks like as a Python function:

```
def binary_search(data, element):
    """
```

```
    Searches data for element, using binary search
    data is assumed to be in ascending order
    """

    # number of data items
    ndata = len(data)

    # no data remaining
    if ndata == 0:
        return False
    else:
        # find halfway point
        half_length = ndata//2
        # have we found element?
        if data[half_length] == element:
            return True
        # recursively search left half
        elif data[half_length] > element:
            return binary_search(data[0:half_length], element)
        # recursively search right half
        else:
            return binary_search(data[half_length+1:ndata], element)
```

Now, this certainly feels like a much better way of searching for data in a sorted list than `search`. But can we make that intuition more precise?

Both of our algorithms are capable, by pure dumb luck, of hitting upon the element we're searching for, first go. So they have identical **best case** behaviour. But things look very different in the **worst case**. In the case of `search`, our luck might be out: either the element might not be in the list, or it might be last ("It's always the last place you look," as people say.) In that case the number of iterations would be the length of `data`.

Now, what about `binary_search`? Well, suppose the data starts off consisting of 255 items. After one inspection of the midpoint, either we've found `element` or we haven't; we're thinking about the worst case, so let's assume we're out of luck. We then throw away the midpoint, and half of the rest of the list, leaving us with a list of length 127. After another inspection of the midpoint, we're down to 63 (assuming we haven't found it). Then we get 31, then 15, then 7, then 3, then 1; one final inspection and we're done. That was a total of 8; `search` would, in the worst-case assumption, have taken 255 iterations.

You'll have noticed that if the list starts off of length $2^m - 1$, then after one inspection of the midpoint with no luck, it's down to $2^{m-1} - 1$, then $2^{m-2} - 1$ and so on. The precise number of inspections of the midpoint necessary for a list of length $2^m - 1$, under "worst case" assumptions, is m.

So if $n = 2^m - 1$, the number of iterations necessary is $\log_2(n + 1)$. This will work as a decent approximation for other values of n as well.

Now, as n gets large, the difference between $\log_2(n + 1)$ and $\log_2 n$ becomes negligible.

Moreover, $\log_2 n$ is just a constant multiple of $\log_e n$ or $\log_{10} n$; so the base of the logarithm doesn't matter. We say that the worst-case complexity of this algorithm is logarithmic, and we use the notation $O(\log n)$.

More generally, if $f(n)$ is a function so that, under worst case assumptions, the number of operations in a task of size n is asymptotically equal to a multiple of $f(n)$, we write that the algorithm is $O(f(n))$. This is, unsurprisingly, known as **big O notation**.

By contrast, then, `search` is $O(n)$, which is much worse than $O(\log n)$. Though that doesn't alter the fact that if the data isn't sorted, we're going to have to use it!

What about the best case? Well, here our two algorithms are on all fours. In either case, we may just be lucky, and hit our `element` first time we look. The best-case complexity is represented using **big Ω notation**: we say that both `search` and `binary_search` are $\Omega(1)$.

Just to see this in action, here are versions of our functions with global variables that keep a count of the number of comparisons each method makes:

```
def search(data, element):
    """
    Searches data for element, term by term
    """

    global inspection_count

    # for loop
    for x in data:
        inspection_count += 1

        # have we found element?
        if x == element:
            return True

    return False

def binary_search(data, element):
    """
    Searches data for element, using binary search
    data is assumed to be in ascending order
    """

    global inspection_count

    # number of data items
    ndata = len(data)
```

```
    # no data remaining
    if ndata == 0:
        return False
    else:
        inspection_count += 1
        # find halfway point
        half_length = ndata//2
        # have we found element?
        if data[half_length] == element:
            return True
        # recursively search left half
        elif data[half_length] > element:
            return binary_search(data[0:half_length], element)
        # recursively search right half
        else:
            return binary_search(data[half_length+1:ndata], element)
```

Then define a data set in which our target element is unlikely to appear:

```
from random import randint
data = [randint(1,10**5) for r in range(255)]
data.sort()
```

Then:

```
inspection_count = 0
print(search(data, 1111))
print(inspection_count)
```

```
False
255
```

```
inspection_count = 0
print(binary_search(data, 1111))
print(inspection_count)
```

```
False
8
```

This matters little when the data set is as small is 255, but for much, much larger data sets there will be an appreciable difference in execution time, becoming more severe as the data lists get longer.

11.4 Big Θ

Let's consider another task: **sorting** a list of data. Now, you take a deep dive into this on this week's problem sheet, but for now, let's look at a very simple sorting algorithm; the one you implemented yourselves a few weeks ago.

This is called **selection sort**, and it works like this, for a data list of length 16:

- Find the minimum of all the items, and place that in position 0, swapping positions with the item that's there now.
- Find the minimum of the items in positions 1, 2, 3 etc, and place that in position 1.
- Find the minimum of the items in positions 2, 3, 4 etc, and place that in position 2.
- Continue until you've placed an item in position 14; the final item will then automatically be in its correct position, namely 15.

Now, the first search for a minimum involves 15 comparisons between data items: you start by comparing item 0 with item 1, then whichever is the smallest of those with item 2, then the mininum of those 3 with item 4, and so on.

The second search for a minimum only involves 14 comparisons, though: you leave item 0 where it is, and compare starting with item 1.

Then there are 13 comparisons, and so on.

All in all, there are $15 + 14 + 13 + \cdots + 1 = 120$ comparisons. More generally, with a data list of length n, there are $\dfrac{n\,(n-1)}{2}$.

(You can view animations of this search algorithm on Blackboard.)

Suppose we decide to measure the complexity of a search method by the number of comparisons of data items (there's room for debate about whether we should, but we generally do). Then what's the worst-case complexity, and what's the best-case complexity?

The answer is the same in both cases. Asymptotically, $\dfrac{n\,(n-1)}{2}$ is a multiple of n^2; for large n, the quadratic term dominates. So this algorithm is both $O(n^2)$ and $\Omega(n^2)$; when that's the case, we say it's $\Theta(n^2)$; this is **big Θ notation**.

Note that the best-case and worst-case performances don't have to be the same, as they are here; for a Θ-complexity to be well defined, they simply have to be, asymptotically as n gets large, multiples of the same function of n. So an algorithm that required $40000\,n^2 + 500000n$ operations in the worst case, and $(n^2 - n)/2$ in the best, would be both $O(n^2)$ and $\Omega(n^2)$, and hence $\Theta(n^2)$.

Here's a listing of an implementation of selection sort, complete with a global variable called `comparison_count`.

```
def selection_sort(data):
    """
    Sorts data in place, using selection sort
    """
```

```
    global comparison_count

    # start position goes from 1 to len(data) - 2
    for start_position in range(len(data)-1):
        # search for minimum and record its position
        min_position = start_position
        for index in range(start_position+1, len(data)):
            comparison_count += 1
            # compare minimum so far with next data item
            if data[min_position] > data[index]:
                min_position = index
        # swap positions of minimum and starting elements
        data[start_position], data[min_position] = \
        data[min_position], data[start_position]
```

Then

```
data = [3, 34, 12, 22, 27, 17, 31, 29, 40, 24, 21, 19, 7, 18, 26, 5]
comparison_count = 0
selection_sort(data)
print(data)
print(comparison_count)
```

```
[3, 5, 7, 12, 17, 18, 19, 21, 22, 24, 26, 27, 29, 31, 34, 40]
120
```

Notice that `data` is sorted **in place**; our function doesn't return any output, but instead sorts `data` as a side-effect, changing its value. For that reason, when writing sort algorithms, it's often convenient to have a "master" data list, from which one makes a copy, but which doesn't itself undergo any change.

Now, you might thing the following would work for that:

```
masterList = \
[3, 34, 12, 22, 27, 17, 31, 29, 40, 24, 21, 19, 7, 18, 26, 5]
data = masterList
```

But in fact it doesn't; as you may remember, in Python, when we set one list equal to another like that, they end up pointing to the same object, meaning that if one changes so does the other. The way to get round that is to use a function called `copy`, which lives in the `copy` module.

```
masterList = \
[3, 34, 12, 22, 27, 17, 31, 29, 40, 24, 21, 19, 7, 18, 26, 5]
```

```
from copy import copy
data = copy(masterList)
```

Now we can make as many changes as we like to `data`, and `masterList` will remain unaltered.

11.5 A cool Maths example

Challenge 1: write your own version of the `pow` function, for calculating the **modular exponent**: that is, the residue of a^b modulo c.

Here's a first go at that:

```
def mypow1(a, b, c):
    """
    Calculates (a**b) % c by repeated multiplication
    """

    # initialize output
    d = 1

    # loop b times
    for r in range(b):
        # multiply d by a
        d = (d*a) % c

    return d
```

And, I mean, that works fine, as the following example shows:

```
print(pow(123, 456, 789))
print(mypow1(123, 456, 789))
```

```
699
699
```

However, the number of iterations is always exactly b, making this method $\Theta(n)$. We can do a lot better than that!

Our improved implementation works, in essence, by representing b as a binary number: that is, as a sum of powers of 2. So, for example

$$456 = 256 + 128 + 64 + 8.$$

Our plan is this.

- Set $d = 1$.

- Start with a^1. But 1 doesn't appear in b's binary expansion, so do nothing yet.
- Square this number to get a^2, reducing mod c. 2 doesn't appear either, so sit tight.
- Square again to get a^4, reducing mod c. Still no need to do anything.
- Square a third time to get a^8, reducing mod c. This time,as 8 does appear in b's binary expansion, multiply d by this number, reducing mod c.
- Keep squaring till you get to a^{64} and multiply d by that, always reducing mod c.
- Square to get a^{128} and multiply d by that.
- Square to get a^{256} and multiply d by that.

At the end of the process, we'll have multiplied d by $a^8 \times a^{64} \times a^{128} \times a^{256} = a^{456}$, but we've only used a few iterations to do so. This "repeated squaring" algorithm is tamer of large numbers.

Here's an implementation:

```
def mypow2(a, b, c):
    """
    Calculates (a ** b) % c by repeated squaring
    """

    # initialize d
    d = 1

    # while loop
    while b > 0:
        # last binary digit of b
        digit = b % 2

        # if it's 1, multiply d by a
        # otherwise, do nothing
        if digit == 1:
            d = (d*a) % c

        # square a
        a = (a**2) % c

        # halve b using integer divsion
        # to get rid of final binary digit
        b //= 2

    return d
```

Then it works ...

```
print(pow(123, 456, 789))
print(mypow1(123, 456, 789))
print(mypow2(123, 456, 789))
```

```
699
699
699
```

..., but it's way, way quicker for large values of b; in fact, it's almost as fast as `pow` itself (and you can bet this is how `pow` is implemented).

```
from random import randint
a = randint(10**99,10**100)
b = randint(10**99,10**100)
c = randint(10**99,10**100)
print(a)
print(b)
print(c)
print(pow(a,b,c))
print(mypow2(a,b,c))
```

```
65747536772064001083941071512280362408965691447664185594081691029815679905521968465350995
87386707965931861835426662601898685610202160807469191665027109736741963342704620037403606
79129013959580129784335711019653429753068421862170661450928414621411964109915150395616511
26760265388233163272475055429871890071628127635698429981726373922561413241213384723500932
26760265388233163272475055429871890071628127635698429981726373922561413241213384723500932
```

Just what is its complexity? Well, the number of iterations is equal to the number of times we can integer-divide b by 2 until we hit zero, which is approximated well by $\log_2 b$. This iteration count doesn't vary, meaning that this algorithm is $\Theta(\log n)$.