> *Our two weapons are fear and surprise...and ruthless efficiency*

Complexity and Computability

Efficiency

It is not enough to write a program that works correctly, the program must also be efficient. What makes a program efficient? There are two types of efficiency that we consider in software development: time (how fast the program runs) and space (how much of the computer's memory is being used).

Writing fast software is important because users will be less satisfied if a program is slow. Writing software that uses minimal space is important because computers have limited resources. In this section, we will discuss time efficiency, but similar techniques can be applied to work out space efficiency.

How different can two programs be in time efficiency? Let's look at an example involving the Fibonacci sequence, which is a sequence of numbers defined as:

```
fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2) for n > 1
```

This recursive definition can be directly translated into a recursive function in Python as follows.

```
def fibonacci1(n) :
    if n == 0 : return 0
    elif n == 1 : return 1
    else : return fibonacci1(n-1) + fibonacci1(n-2)


>>> for i in range(10) : print(fibonacci1(i), end=" ")

0 1 1 2 3 5 8 13 21 34
```

It turns out that this direct translation does not produce a very efficient algorithm because we end up recomputing Fibonacci numbers over and over again. We can do much better by keeping track of the previous two Fibonacci numbers and that leads to the solution below.

```
def fibonacci2(n) :
    fib1 = 0
    fib2 = 1
    for i in range(n) :
        # loop invariant: a is the i'th fib and b is the (i+1)'th fib
        fib1, fib2 = fib2, fib1+fib2
    return fib1
```

These functions are in `fib.py`. We have also included functions to determine the runtime of the two Fibonacci functions using the time module. Let's run them to see the results:

```
>>> time_fib1(30)
fibonacci1(30) took 309.849 ms
>>> time_fib1(31)
fibonacci1(31) took 455.199 ms
>>> time_fib1(32)
fibonacci1(32) took 763.001 ms
```

We would like to analyse these results, and estimate the running time of `fibionacci1(n)` for large values of `n`. How does the running time change when we increase `n` by 1?

```
>>> 455.199/309.849
1.469099
>>> 763.001/455.199
1.676192
```

It turns out that if we compute `fibionacci1(n+1)`, the time taken is about 1.6 times the time it takes to compute `fibionacci1(n)`. From this, we can determine that the time to compute `fibionacci1(n+10)` is a little over 100 times the time to compute `fibionacci1(n)`.

```
>>> 1.6**10
109.95116277760006
```

How long will it take to compute `fibionacci1(40)`? We estimate that it would take `1.6**10 * 309.849 ms`, which is about 34 seconds. Extrapolating further, we find that `fibionacci1(50)` would take just over one hour, and `fibionacci1(100)` would take over 1,900,000 years! On the other hand we get the following for the second algorithm.

```
>>> time_fib2(30)
fibonacci2(30) took 0.000 ms
>>> time_fib2(10000)
fibonacci2(30000) took 31.215 ms
>>> time_fib2(30000)
fibonacci2(300000) took 709.301 ms
```

Note that the dramatic difference is not because one is recursive and one is not — if we rewrote the second algorithm as a recursive function, it would still be very efficient.

Note also that the exact running times will be different when it is run on different processors, in different programming languages, etc. However, the resulting analysis will still be the same, that as `n` increases, the running time increases by a factor of about 1.6.

Measuring Efficiency

In the above analysis, we have seen how to estimate the running time of the `fibionacci1` function for certain values of `n`. We would like to make this more general and describe the efficiency, called the **complexity**, of `fibionacci1` so that it can be compared to other functions. Since the exact running time can differ, we will instead describe how the running time grows with larger inputs, without having to worry about exact details.

From the analysis above, we see that the running time of `fibionacci1(n)` is related to 1.6^n. In fact, the only important information is that the running time is exponential in `n`. The 1.6 is not important, and the complexity could equally be described as 2^n or e^n (it is common to use either of these to describe exponential functions).

Before continuing, we need a notation that we can use to mean that we do not care about the exact details.

Big O Notation

The **big O notation** is used to provide an upper bound on a mathematical function. Formally, we say that "$f(n)$ is of order $g(n)$", and write "$f(n)$ is $O(g(n))$", if there are constants a and k such that $f(n) \leq a \times g(n)$ for all $n > k$. Informally, $f(n)$ is of order $g(n)$ if some multiple of $g(n)$ is an upper bound of $f(n)$.

It is common to use only the most significant term when describing the order of a function, and to ignore any coefficients. For example, $3n^2 + 4n - 2$ is $O(n^2)$, $2n^2 + 8n$ is also $O(n^2)$, and $3^n + n^2$ is $O(2^n)$.

Big O notation is used to describe the complexity of a program, where n represents the size of the input. For example, the running time of `fibionacci1` is $O(2^n)$.

A **complexity class** is a collection of functions with the same complexity (when expressed in the big O notation). Some of the common complexity classes are:

Complexity	Common Name
O(1)	Constant
O(log n)	Logarithmic
O(n)	Linear
O(n^2)	Quadratic
O(2^n)	Exponential

Once the complexity of a function is known, we can compare it to the complexity of other functions. In the table above, we have listed the classes from the most efficient (constant) to the least efficient (exponential). When designing algorithms we strive for the most efficient algorithm.

Calculating Rate of Growth

How can we determine the complexity of a function? Instead of performing an experimental analysis, we can directly analyse the source code. In practice this can be quite difficult, so we will consider several examples. In our analysis, we will think about how many steps the function takes to complete if it were executed. Looking at how many times a loop body is executed, or how many recursive calls are needed to reach the base case, will give a good indication of the complexity.

Constant Time

A function that runs in constant time — $O(1)$ — cannot have any loops or recursive steps which depend on the size of the input. The defining characteristic of constant time functions is that changing the size of the input has no effect on the running time. For example, these functions run in constant time:

```
def square(n) :
    return n**2
```

```
def abs(n) :
    if n > 0 :
        return n
    else :
        return -n
```

Linear Time

A function runs in linear time — $O(n)$ — if the number of iterations of loops in the function is proportional to the size of the input (and the body of the loops run in constant time). A characteristic to observe in linear-time functions is that doubling the size of the input makes the function twice as slow. Below are examples of functions that run in linear time.

```
def find(char, string) :
    for i, c in enumerate(string):
        if c == char :
            return i
    return -1

def sumto(n) :
    total = 0
    i = 0
    while i <= n :
        total += i
        i += 1
    return total
```

It is important to specify what inputs determine the complexity of the function. The `find` function above is linear in terms of the length of the string, and the `sumto` function is linear in terms of the input `n`.

A recursive function runs in linear time if the number of recursive steps to the base case is proportional to the size of the input. The `factorial` function below must make a total of `n` recursive steps, so its running time is $O(n)$.

```
def factorial(n) :
    if n == 0 :
        return 1
    else :
        return factorial(n-1) * n
```

Quadratic Time

A characteristic of quadratic time — $O(n^2)$ — functions is that doubling the size of the input makes the running time four times as large. Many functions that run in quadratic time have two nested loops, where the number of iterations in each loop is proportional to the size of the input. The following functions run in quadratic time.

```
def pairs(lst) :
    result = []
    for element1 in lst :
        for element2 in lst :
            result.append((element1, element2))
    return result
```

```
def primes_to(n) :
    """Return a list of primes <= n."""
    primes = []
    for i in range(2, n+1) :
        is_prime = True
        for j in range(2, i) :
            if i % j == 0 :
                is_prime = False
        if is_prime :
            primes.append(i)
    return primes
```

The `pairs` function runs in $O(n^2)$ time where n is the length of the list. This is because the inner loop takes n steps to run, which is then repeated n times by the outer loop.

The `primes_to` function is slightly different. The outer loop runs `n-1` times, which is $O(n)$. The inner loop runs `i-2` times, but `i` takes a different value on each iteration, so how do we represent the running time in terms of `n`?

If we sum up the number of iterations of the inner loop, we get $0 + 1 + 2 + 3 + \cdots + (n-3) = (n-3)(n-2)/2$, which is quadratic. A less formal reasoning would be that `i` averages to about $n/2$ over all iterations of the outer loop, so the total running time is about $n^2/2$, which is quadratic.

Logarithmic Time

The defining characteristic of logarithmic — $O(\log(n))$ — functions is that doubling the input size will only increase the running time by a fixed amount. A way of recognising a logarithmic function is that at each step through the function, the "remaining problem" will be reduced by a significant factor. For example, consider the following function, which computes the binary digits of a number, starting with the least significant.

```
def binary(number) :
    result = []
    while number > 0 :
        if number % 2 == 0 :
            result.append(0)
        else :
            result.append(1)
        number /= 2
    return result
```

Notice that each step through the loop halves the `number`, which halves the size of the remaining problem. Because of this, the `binary` function runs in $O(\log(n))$ time.

Two of the methods in the `Node` class from the module about recursion are also logarithmic: the `insert` and `__contains__` methods both make a recursive call into one of the two children of the node. In effect, this (roughly) halves the size of the "remaining problem", which is the number of nodes that still need to be considered. Because binary search trees have very efficient `insert` and `__contains__` methods, they can be very useful ways of storing data.

This only holds true if the tree is balanced, that is, if each node in the tree has a roughly equal distribution of nodes in the left and right subtrees. The other possible extreme is that each node in the tree has only one child, then the tree behaves more like a consecutive list of values, and `insert` and `__contains__` will run in linear time, and the tree will no longer be a useful way

of storing data. For this reason, many forms of self-balancing binary search trees have been developed, which ensure they remain balanced when data is inserted or removed.

Exponential Time

An example of a function which runs in exponential time — $O(2^n)$ — is the `fibionacci1` function from above (which is available in `fib.py`). This function is exponential because there are two recursive calls made in the recursive step. The effect of this is that when the input `n` increases by 1, our function needs to do nearly twice as many steps (more accurately, 1.6 times as many steps) to compute a result.

The important characteristic of exponential time functions is that increasing the input size by 1 will multiply the running time by a factor.

> **Aside: Things aren't as they seem...**
>
> Though it may seem obvious that a function appears to have a certain time complexity, other factors can come into the running time as well. If we analyse the efficient Fibonacci algorithm it appears to be linear. However, if we do the timing experiment we see that it is not linear. It turns out that Fibonacci numbers get very big very quickly and so the time taken to add two adjacent Fibonacci numbers becomes significant — and has to be factored into the calculations. It appears from timing experiments that, for very large `n`, doubling `n` will make the time go up by about a factor of 3 — so the complexity is worse than linear but better than quadratic.

Functions within Functions

How is the complexity of a function affected when it calls other functions? If a function calls another function, the complexity of that other function will have an impact on the complexity of the first function. As an example, consider these two functions:

```
def factorial(n) :
    if n == 0 : return 1
    else : return factorial(n-1) * n

def sum_factorial(n) :
    """ Return 1! + 2! + ... + n! - i.e. the sum of the factorials up to n."""
    sum = 0
    m = 1
    while m <= n :
        sum += fact(m)
        m += 1
    return sum
```

We already know that the `factorial` function runs in $O(n)$ time, but what about the `sum_factorial` function? In the loop, `m` starts at `1` and increments until it gets to `n`. However, this does NOT make it a linear algorithm because the body of the loop contains a call that is linear in `m`. So the time taken is proportional to $1 + 2 + 3 + \cdots + n = n \times (n + 1)/2$ — making the function quadratic.

What about when we are using other functions that we did not write? Without knowing the complexity of other functions, we cannot determine the running time of our own programs. If the source code is available, we could analyse it as above. The author of the other code may have included information about the time and space complexities in the documentation. This

would be very helpful, especially if we do not have access to the source code. Alternatively, we could carry out timing experiments, as we did with `fibionacci1`, to deduce the complexity.

The complexity of standard Python functions and operations also needs to be considered. What is the complexity of the built-in functions and methods, such as `sum`, `min`, or `list.append`? What about list addition? Or string addition?

The code in `list_time.py` will experimentally determine the running time of many list operations. Below are some results (each command was repeated 1000 times, the results were averaged, and are measured in microseconds).

```
xs.append(0):
  list of length 10000: 0.8640663673243391 us
  list of length 20000: 0.9903276535270789 us
  list of length 40000: 1.578083090180371 us
xs[0]:
  list of length 10000: 0.3674386415868369 us
  list of length 20000: 0.6851047181877234 us
  list of length 40000: 0.684738743433666 us
xs[-1]:
  list of length 10000: 0.3872012777055289 us
  list of length 20000: 0.7191403692736742 us
  list of length 40000: 0.8018506610625309 us
xs[0:100]:
  list of length 10000: 1.2208917414007203 us
  list of length 20000: 1.7446015980686624 us
  list of length 40000: 1.8748886064123838 us
xs[0:10000]:
  list of length 10000: 53.55271811269802 us
  list of length 20000: 57.665908245043696 us
  list of length 40000: 62.512145786778106 us
xs[1:-1]:
  list of length 10000: 54.147427069450416 us
  list of length 20000: 112.26568009637639 us
  list of length 40000: 230.8202701698896 us
xs.insert(0, 0):
  list of length 10000: 8.216132971785584 us
  list of length 20000: 15.295182418274322 us
  list of length 40000: 29.147692401203074 us
xs.pop(0):
  list of length 10000: 6.027604010938603 us
  list of length 20000: 12.176711636406878 us
  list of length 40000: 22.701779259406862 us
xs.pop(-1):
  list of length 10000: 0.806608332709402 us
  list of length 20000: 1.1729490500478335 us
  list of length 40000: 1.3764310069532826 us
xs + xs:
  list of length 10000: 112.45232721501708 us
  list of length 20000: 232.56779956593476 us
  list of length 40000: 470.1520149288214 us
```

It can be seen that `append` and list indexing both run in constant time (allowing for slight variations in the running times). A list slice runs in time proportional to the length of the slice. This means that if a slice is taken from the start to the end of the list, it will take linear time in terms of the length of the list. Inserting to the front of a list runs in linear time, as does list

addition. The `pop` method is somewhat more interesting: it runs in linear time if the first element is removed, and constant time if the last element is removed.

Note that one consequence of this is that inserting and removing from a list is best done from the end of the list, using `append` and `pop(-1)`, because these are both constant, whereas the same operations on the start of the list are slower. Being able to take advantage of properties like this one is an important skill in software engineering.

> **Aside: Guess the implementation**
>
> Sometimes it can be insightful to "guess" what the implementation of a function would look like to get a better understanding of why the time complexity is what it is, and whether or not it can be improved. For example, it's not difficult to determine that the `sum` function must visit every element of the list to calculate the total, so `sum` must be linear, it can't be logarithmic or constant.
>
> From the experiment above on lists, we can guess that slicing and addition work by copying all of the required elements into a new list, which is why they are linear. If inserting and removing from the start of a list runs in linear time, we might guess that the underlying implementation needs to do some extra work on the rest of the list. It may be interesting to research two different implementations of lists: "arrays" and "linked lists".

Computability: The Halting Problem

In the remainder of this section we will look at a new topic, **computability theory**, which studies functions and algorithms from a mathematical perspective. One question in this field is deciding whether or not a given function will finish executing or loop forever. In this section, we will only consider functions that take a single string argument. For example, consider this function:

```
def factorial(number) :
    n = int(number)
    fact = 1
    while n != 0 :
        fact *= n
        n -= 1
    return fact
```

Calling this function with a positive integer, such as `factorial('3')`, will finish executing. However, `factorial('-2')` will never finish executing, as the loop will never stop. Note that we are considering if the function will ever finish, even if it takes a long time. For example, `factorial('1000000000000')` will take a very long time, but it will still be able to finish.

We now pose the problem: Write a program which takes the source code of a function (as a string) and an input, and determines whether or not the function will stop executing. This problem is known as the **halting problem**. This would not be an easy task, because it would have to handle every possible function we could write.

In fact, Alan Turing proved in 1936 that writing such a program is impossible. This is an alarming result, as it was one of the first problems where it could be proved that a solution could not be found. Here we will give a **proof by contradiction**, where we assume that we can solve the halting problem, and then prove that assumption leads to a contradiction.

Assume that we have a function `halt`, which takes the source code of a function (as a string), and an `input`, and returns `True` if the function will finish when given that input. The function definition would look like this:

```
def halt(code, input) :
    """Determine if the given function 'code' will finish executing.

    Parameters:
        code (str): Source code for the function to be analysed.
        input: The input to be passed to the function 'code'.

    Return (bool):
        True if the function stops when given the input.
        False if the function continues forever when given the input.
    """
    # Solution goes here...
    # ...
```

For example, `halt(factorial_code, '3')` is `True` and `halt(factorial_code, '-2')` is `False`, where `factorial_code` is the function definition above. Notice that, in a proof by contradiction, it is enough to assume that a working function is available to us, we do not actually need to write it. Now, consider this function, which takes a string `x`:

```
def g(x) :
    if halt(x, x) :
        while True :
            print("Keep going...")
    else :
        print("Stop!")
```

Now we ask, if `g_code` is the source code of the function above, what does `g(g_code)` do?

- If `halt(g_code, g_code)` is `True`, then `g(g_code)` runs forever, meaning that `halt(g_code, g_code)` gave an incorrect result.
- If `halt(g_code, g_code)` is `False`, then `g(g_code)` stops, meaning that `halt(g_code, g_code)` gave an incorrect result.

This means that our initial assumption that a working `halt` function exists leads to a contradiction. Therefore, it is impossible to write a function that will solve the halting problem.