TUTORIAL 3

Recap: Complexity and Big O notation.

We wish to evaluate the performance of algorithms in a formal way beyond statements such as "slow" or "fast". We could, as a first pass, evaluate by simply measuring the execution time of the algorithm. But this method has its flaws: the result you get is highly dependent on how fast your computer is, it can be clunky to demonstrate how performance scales with changing input, and it is concrete.

Time complexity is a way of abstractly quantifying the performance of an algorithm relative to its inputs. We do this by counting, as a function of input size, the number of primitive operations done, e.g. additions, array accesses, ... We further break this idea into 3 types:

1. **Worst case:** how many ops are performed when the algorithm has to do the most work?
2. **Best case:** similar to worst case, but replace "most" with "least"
3. **Average case:** what is the average number of ops performed? [Rk: this is NOT just the average of best and worst complexity!]

Big-O notation is a way of viewing time complexities in terms of their limiting behaviour without regard for "minute variations". This is a concept with a formal mathematical definition, but for COMP2521 it suffices to remember some basic rules:

1. "$f(n)$ is $O(g(n))$" means that, eventually (i.e. for big enough $n \gg N$), the function f will grow no faster than a constant multiple of g does, even if this is not immediately true for small n
2. When someone asks for the time complexity of an algorithm, we want the tightest / best estimate you can give: $f(n) = 2n+1$ is $O(n^2)$ for sure, but it is most helpful to say it is $O(n)$
3. Lower order terms are ignored: $f(n) = n^2 + 2n + 1$ is $O(n^2+2n)$, but in the long run the truly dominant term is n^2, so we just say $f(n) = O(n^2)$
4. In general, constants $\ll$ logarithms $\ll$ polynomials $\ll$ exponentials $\ll$ factorials $\ll$ ...

$$1 \ll \log n \ll n^\alpha \text{ for } \alpha > 0 \ll \beta^n \text{ for } \beta > 1 \ll n! \ll \dots$$

There is a similar concept for memory usage called **space complexity**: the amount of extra space used by the algorithm. Note: we don't count the input space in this figure!

1.

0	1	2	3	4	5	6
r	a	c	e	c	a	r

⇒ palindrome.

0	1	2	3	4	5	6
r	a	c	e	b	a	r

⇒ not a palindrome.

2. $a_0 + a_1 x_1 + a_2 x^2 + \ldots + a_{n-1} x^{n-1} + a_n x^n = \sum_{i=0}^{n} a_i x^i$

can be written as a loop.

3.

```
BinaryConversion:
    Input  positive integer n
    Output binary representation of n on a stack

    create empty stack S  = O(1)
    while n > 0 do
        push (n mod 2) onto S  = O(1)
        n = floor(n / 2)  = O(1)
    end while
    return S  = O(1)
```

Continually halve and round down until we hit 0

Question: how many iterations in this loop?

Well, suppose there are k iterations. Then it takes k halvings of n to bring it down to 1, which is the last value of n where the loop actually runs. That is, using some math,

$$\frac{n}{2^k} = 1 \quad \Rightarrow \quad k = \log_2 n.$$

So we do $O(\log n)$ iterations. Thus the time complexity is $O(\log n)$.

Intuitively: halving | multiplying tends to yield $O(\log n)$ complexities.

If an algorithm has a complexity of $O(\log n)$, then the amount of work it performs increases at a decreasing rate when n gets big. (i.e. less and less extra effort required.)