Learning Outcome 6 – Evaluating Complexity of Algorithms

How do we measure the amount of work done by computers? How can we compare algorithms or programs to see which is more efficient? (Efficiency can be speed/time/number of steps, or storage space/memory, or other resources.)

* We could measure the time used to run a program, but that depends on the data used and machine used
  + Looks at a particular implementation, rather than the solution in general terms
  + Depends on the machine being used (so often such **benchmarks** are used to compare machines)
  + Depends on the test cases (data used)
* Another solution: Analyze the general properties of algorithms
  + We’ll focus mostly on time/speed/number of steps (could also look at storage space)
  + We’ll focus mostly on worst-case analysis (rather than best-case or average-case)
  + We’re interested in “large” amounts of data – we’re generally interested in how the efficiency is related to some key indicator (usually denoted by *n*)
  + We also need to take into account that not all steps are the same – choose a key step (most important step or most related to the algorithm); make sure that all other steps are comparable
    - Beware of method calls (or similar algorithmic steps), because they may involve a lot of steps – must calculate and account for those steps too

For instance, to add an item to a sorted list (and maintain the sort order), we could do the following:

Add the item to the end (or to the start) of the list

**Sort the list**

This *appears* to be two steps, but sorting is hard (it takes from *n* \* log *n* steps to *n*2 steps, depending on how it’s done), so the order of this process is the same as the order of the sorting process.

Do a binary search on the list *(takes log2n steps)*

Add the item in the correct spot in the list *(could take n steps if we’re working with an array)*

This would be O(log *n*) + O(*n*) if we’re working with an array, but we ignore smaller terms, so it is O(*n*).

Instead, we could do the following:

Do a sequential search on the list and add the item in the correct spot when encountered.

This would be O(*n*) for the sequential search and O(1) for putting the item in the right spot, so overall it would be O(*n* + 1) but we ignore constants, so O(*n*).

* Ignore value of a lower order in the number of steps (they become unimportant for large values of *n*) **if added**
  + For instance, if there are 3*n*2 + 7*n* steps, ignore the 7*n* and focus on the 3*n*2
  + **CANNOT ignore lower orders if they are multiplied** (so for instance, for *n* \* log *n*, we cannot ignore the log *n*, so our count would be *n* log *n*).
* Ignore constants (they become unimportant compared to other orders of complexity)
  + For instance, if there are 4*n*2 + 3 steps, ignore the 4 and the 3 and focus on the *n*2, so this algorithm would be O(*n*2)
* Look at loops/recursion – they cause the work to be done repetitively.
  + If a loop cuts the data in half each time, it is doing log2(*n*) steps
  + If loops are nested, multiply the number of steps that each loop does (for instance, *n* \* *n* = *n*2 total loops)
  + If the loops are NOT nested (one part of the algorithm is completely done before the next part starts), simply add the steps done by each loop (for instance, *n* + *n* = 2*n*, so this algorithm would be O(*n*) – ignore the constant 2).

**When done, you can compare the order of similar algorithms (working on the same kind of data, with the same kind of operations/situations) to see which is more efficient – which order of complexity is lower.**