**General Tips**

Use good, simple examples to solve a question.

* It makes it easier to develop a solution
* Careful about using special / edge case examples
* Keep your example as simple as possible, to reduce the amount of time finding bugs
* Starting off with a brute-force solution is okay. It can be optimised after.

**Behavioural Questions (skipped)**

**Big O – This must be mastered**

Big O describes the asymptotic behaviour of functions. It tells us how fast a function grows or declines.

* O(1) = constant
* O(logn) = logarithmic
* O(logn)C = polylogarithmic
* O(n) = linear
* O(n2) = quadratic
* O(nC) = polynomial
* O(cN) = exponential

*What is the relationship between best/worst/expected case and big O?*

* Best, worst and expected cases describe the big O time for particular inputs or scenarios
* Big O describes the upper, lower and tight bounds for the runtime.

Space Complexity

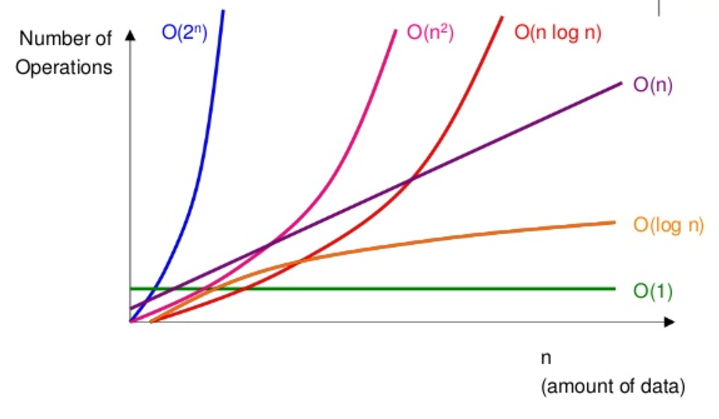
* The amount of memory/space required by an algorithm also matters.
* Space complexity is parallel to time complexity (creating an array of size n requires O(n) space).
* Stack space is a block of memory used to store temporary data for program execution
* A recursive function would add to the same call stack multiple times.
* However, a non-recursive function that calls a second function would not add to the same call stack, as both calls do not exist simultaneously.

Dropping Constants

* We drop constants in runtime as the Big O just describes the rate of increase.
* E.g. a function with two non-nested for loops = O(2N) 🡪 Actually O(N) is more correct

Dropping the Non-Dominant Terms

* Similar concept to dropping constants.
* For O(N2 + N), N wouldn’t matter as it would become less and less significant as N increases.



Multi-Part Algorithms: Add VS. Multiply

* Suppose there is an algorithm with two steps. When do you MULTIPLY and when do you ADD the runtimes?
* If your algorithm is in the form:
  + “Do this, then, when you’re all done, do that” = add the runtimes
  + “Do this for each time you do that” = multiply the runtimes
* E.g. two separate for-loops (add runtime) / two for-loops nested (multiply runtime)

Amortized Time

* ArrayList is an array that is created with an initial size, but when the size is exceed, it can dynamically resized.
* Amortised time explained in simple terms: (ArrayList example)

If you do an operation say a million times, you don't really care about the worst-case or the best-case of that operation - what you care about is how much time is taken in total when you repeat the operation a million times.

So it doesn't matter if the operation is very slow once in a while, as long as "once in a while" is rare enough for the slowness to be diluted away. Essentially amortised time means "average time taken per operation, if you do many operations". Amortised time doesn't have to be constant; you can have linear and logarithmic amortised time or whatever else.

Let's take the example of a dynamic array, to which you repeatedly add new items. Normally adding an item takes constant time (that is, O(1)). But each time the array is full, you allocate twice as much space, copy your data into the new region, and free the old space. Assuming allocates and frees run in constant time, this enlargement process takes O(n) time where n is the current size of the array.

So each time you enlarge, you take about twice as much time as the last enlarge. But you've also waited twice as long before doing it! The cost of each enlargement can thus be "spread out" among the insertions. This means that in the long term, the total time taken for adding m items to the array is O(m), and so the amortised time (i.e. time per insertion) is O(1).

Log N Runtimes

* O(logN) example: Binary search for X in a sorted array
  + We compare X to the midpoint of the array, then we search on the LHS or RHS and repeat until found or not.
  + Total runtime = dividing N/2 each time until X is found.
* Whenever there is a problem where #elements gets halved each time, that will likely be a O(logN) runtime.
* Similarly, finding an element in a Balanced Binary Search Tree is O(logN)
  + With each comparison, we go either left or right

Recursive Runtimes

* Assume a tree with depth N, with each node having two children.
* Each level will have twice as many calls as the one above, therefore the number of nodes on each level are:

|  |  |  |  |
| --- | --- | --- | --- |
| **Level** | **#Nodes** | **Also expressed as…** | **Or…** |
| 0 | 1 |  | 20 |
| 1 | 2 | 2 \* previous level = 2 | 21 |
| 2 | 4 | 2 \* previous level = 2 \* 21 = 22 | 22 |
| 3 | 8 | 2 \* previous level = 2 \* 22 = 23 | 23 |
| 4 | | 16 | 2 \* previous level = 2 \* 23 = 24 | 24 |

There will be 20 + 21 + 22 + 23 + 24 + … + 2N = 2N+1 – 1

* With a recursive function that makes multiple calls, the runtime will often look like **O(branchesdepth)**, where branches is the #times each recursive call branches out. In the above example, branches = 2 (a tree) = O(2N)
* NOTE: The base of a log doesn’t matter for big O, since logs of different bases only differ by a constant factor.  
  However, exponents DO matter. E.g. Compare 2N and 8N 🡪 8N = ( 23 )N = 23N 🡪 Difference = factor of 22N
* The SPACE COMPLEXITY of this algorithm will be O(N), as only O(N) nodes exist at any given time therefore only needing O(N) memory available.

Sum of Integers 1 through N

* This is for problems that are usually in the form of: *What is 1 + 2 + … + n?*
* N/2 = Number of pairs in the problem
* N+1 = Value of each pair
* Therefore: N/2 \* N+1 = = O(N2) runtime

Balanced Binary Search Trees

* In a BBST, if there are N total nodes, then depth is roughly **LogN**

String Permutations

* Finding all permutations of a string = **O(N!)** N Factorial runtime