



	There is no particular relationship between the two concepts										
	Best Case, Worst Case and Expected Case actually describe the big O or big Theta time for particular scenarios whereas these asymptotic notations describe the upper, lower and tight bounds for the runtime										
	<b>Space complexity</b>										
	Memory or space required by an algorithm	to create an array - if it is unidimensional, $O(N)$ space complexity; for a 2-D array, $O(N^2)$									
	Stack space in recursive calls counts too. Each call adds a level to the stack and takes up actual memory.	However, just because you have $N$ calls does not mean it will take $O(N)$ time: check the example on Page 41 for more details									
	<b>Drop the constants</b>										
	$O(2N)$ is actually $O(N)$										
	<b>Drop the non-dominant terms</b>										
	$O(N^2 + N)$ becomes $O(N^2)$										
	$O(N + \log N)$ becomes $O(N)$										
	$O(5 \cdot 2^N + 1000N^{100})$ becomes $O(2^N)$										
	$O(x!) > O(2^x) > O(x^2) > O(x \log x) \dots > O(x)$										
	<b>Multi-Parts algorithms: add versus multiply</b>										
	Add:	Non-nested chunk of work A and B	$O(A + B)$	"DO THIS THEN WHEN YOU ARE ALL DONE, DO THAT"							





	Example 10	O(root(N)) as the for loop does constant time and runs in O(root(N)) time									
	Example 11	O(N) as the recursive process calls from N to N-1 to N-2 and so on until 1									
	Example 12	<b>Approach 1:</b> We make a tree for an example string say 'abcd' and we see that we branch 4 times at the root , then 3 times, then 2 times, and then 1 time. this gives us 4*3*2*1 leaf nodes. We could say n! leaf nodes for n length string. So, total nodes would be n * n! as each leaf node is attached to a path with n nodes. Also, string concatenation will also take O(n) time. Thus, the final time complexity in worst case would be O(n * n * n!) =. O((n + 2) !)									
		<b>Approach 2:</b> At level 6, we have 6! / 0! nodes; at level 5 we have 6! / 1! nodes; at level 4 we have 6!/2! nodes...at level 0, we have 6!/6! nodes. so, the total nodes in the tree in terms of n can be : n!(1/0! + 1/1! + 1/2! + 1/3! ....+ 1/n!). Now, the term in the bracket can be defined in terms of Euler's number: n! * e whose value is around 2.718. The constant e can be dropped further. Thus, the time complexity would be O(n! * n) where n is due to permutation; thus, time complexity: O((n+1)!)									
	Example 13	We have to use the earlier pattern for recursive calls: O (branches^depth) = O(2^N).	We can also get tighter runtime as O(1.6^N) if we consider that there might be just one call instead of 2 at the bottom of call stack sometimes.								
	Example 14	From previous example, we deduce that fib(n) taken 2^nn time. And, we have fib(1) + fib(2) + fib(3) + ....fib(n) = 2^1 + 2^2 + 2^3 .....+2^nn = 2^nn+1) - 2, thus we can say that the run time is approx. 2^nn									
	Example 15	Now, here in this program we are doing memoization due to which the amount of work reduces to looking up fib(i - 1) and fib(i - 2) values in memo array at each call fib(i). Thus, we are doing a constant amount of work n times in n calls, hence time complexity: O(n)									
	Example 16	The runtime is the number of times we can divide n by 2 until we get down to the base case 1. As, we know the number of times we can halve n until we get 1 is O (logN)									
	Additional Problems										
	1	The for loop iterates through b, thus time complexity is O (b)									
	2	The recursive call iterates through b calls as it subtracts 1 in each iteration, thus time complexity is O(b)									

[illegible]