Chapter VI	**Big O**										
	Asymptotic notations										
	O(Big O)	Describes an upper bound on time	for example: algo that prints all the values in an array: could have Big O time as O (n), O(n^2), O(n^3) or O(2^n) and many other Big O's	Upper bounds on the runtime; similar to a less-than-or-equal-to relationship	if X <= 130, then we also say that X<= 1000 or X<=1000000	In industry, O and theta have been put together and we have to give the tightest description of runtime					
	Omega(n)	Describes the lower bound	for example: printing the values in an array is Omega (n) as well as Omega(logn) as well as Omega(1)								
	Theta(n)	Describes the tight bound on runtime	Theta here means both O and Omega; in this example, it would be Theta(n)								
	Best Case, Worst Case and Expected Case										
	Best Case:	For example, in Quick Sort, if all the elements are equal, then quick sort will, on average, just traverse through the array once - O(N) time	Quick Sort as we know picks random element as a pivot and then swaps values in the array such that the elements less than pivot appear before elements greater than pivot - this gives partial sort. then it recursively sorts the left and right sides uing same process								
	Worst Case:	The pivot could be repeatedly the biggest element in the array. If pivot is the first element in a reversely sorted array. In this cae, our recursion does not divide the array in half and recurse on other half. Instead, it justs shrinks the subarray by 1 element.	Time taken would O(N^2)								
	Expected Case:	both the above best and worst conditions would rarely happen; thus we can expect a runtime of O(nlogn)									
	Relationship between Asymptotic notations and Best Case, Worst Case and Expected Case Concepts										

	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										
	Space complexity										
	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 tot he 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									
	Drop the constants										
	O(2N) is actually O(N)										
	Drop the non-dominant terms										
	O(N^2 + N) becomes O (N^2)										
	O(N + logN) becomes O(N)										
	O(5*2^N + 1000N^100) becomes O (2^N)										
	O(x!) > O(2^x) > O(x^2) > O(xlogx)...> O(x)										
	Multi-Parts algorithms: add versus mutiply										
	Add:	Non-nested chunk of work A and B	O(A + B)	"DO THIS THEN WHEN YOU ARE ALL DONE, DO THAT"							

	Multiply	Nested A and B	O(AB)	"DO THIS FOR EACH TIME YOU DO THAT"							
	Amortized time										
	arrayList adding actually takes copying N elements in a filled array in a new array with double capacity	That copying might take additional O(N) time after accounting for initial O(N) time of adding the elements to the array	But, this copying of elements into a new array does not happen quite often infact once in every some time	Amortized time allows to describe that the worst case can happen every once in a while but once it happens it won't happen again for so long, that the cost is amortized							
	Adding X more space to an array takes additional O(X) time; thus the amortized time for each adding is O(1)	X + X/2 + X/4 + X/8.... = 2X									
	logN runtimes										
	Example: Binary search. We are looking for an element x in a sorted array. We first compare to the midpoint. If x == middle, then we return else if x < middle, we search on the left side of array else on the right side.	We basically start off with N elements, after a single step, we are down to N/2 elements and in another step to N/4 elements and so on.	The total runtime is then a matter of how many steps we can take before it becomes 1	$2^k = N \Rightarrow k =$ logN with base 2	Basically, when you see a problem with logN runtime, the problem space gets halved in each step						
	Recursive runtimes										