Section 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 => k = logN with base 2	Basically, when you see a problem with logN runtime, the problem space gets halved in each step
Recursive runtimes				
Program:	int f(int n){			
	if(n <= 1){			
	return 1}			
	return f(n -1) + f(n -1);}			
How many calls in the tree?				
Do not count and say 2				

	It will have recursive calls with a depth N and 2^N nodes at the bottom most level	More genrically, 2^0 + 2^1 + 2^2 +....2^n-1 = 2^n - 1 nodes	O(branches^depth) where branches is the number of times each recursive call branches	The space complexity would still be O (n) - even though we have O(2^n) nodes in tree total, only O(n) exists at a time
	Examples and Exercises			
	Example 1	O(n) time as we iterate through array once in each loop which are non-nested		
	Example 2	O(N^2) time as we have two nested loops		
	Example 3	j basically runs for N-1 steps in first iteration, N-2 steps in second iteration and so on.	1 + 2 + 3 +....+ N-1 = N(N - 1)/2 ~ N^2	O(N^2)
	Example 4	For each element of array A, the inner loop goes through b iterations where b is the length of array B. Thus, time complexity is O(ab)		
	Example 5	Similar to example 4, 100,000 units of work is still constant; so the run time is O(ab)		
	Example 6	O(N) time as the array is iterated even if half of it (constant 1/2 can be ignored)		
	Example 7	There is no established relationship between N and M , thus all but the last one are equivalent to O(N)		
	Example 8	s = length of the longest string; a = length of the array; Sorting each string would take O(slogs) and we do this for a elements of the array; thus O(a * slogs) . Now, sorting the array would take O(s * aloga) as each string comparison would take O(s) time in addition to array sorting that would take O(aloga); thus adding the two parts: O(a * slogs + s * aloga) = O(a *s (log a + log s)		
	Example 9	Approach 1: for summing up the nodes in a BST, each node is exactly traversed once, thus O(N) time complexity		
		Approach 2: The number of recursive calls is 2 and the depth is logN in a BST --> O (branches ^ depth) = O(2 ^ logN) where the base of logN is also 2 => time complexity = O(N) after simpllifying		

	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	***Approach 1***: 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) !)	
		Approach 2: 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^n time. And, we have fib(1) + fib(2) + fib(3) + ...fib(n) = 2^1 + 2^2 + 2^3 .....+2^n = 2^(n+1) - 2, thus we can say that the run time is approx. 2^n	
	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)	

		3	It does constant amount of work, thus time complexity is O(1)									
		4	The variable count will eventually equal a/b. The while loop iterates through count times. Thus time complexity is O(a/b)									
		5	The algorithm is actually doing a binary search to find the square root. Thus the runtime is O(logN)									
		6	This is straightforward loop that stops when guess*guess > n or guess> sqrt(n); hence time complexity is O(sqrt(n))									
		7	Ifa binary tree is not balanced, the max time to find an element would be the depth of the tree. The tree since it is imbalanced could be a straight list downwards and ave depth n. Thus, runtime is O(n)									
		8	Without any ordering or sorting in a binary tree, we might need to traverse through all the nodes in the tree , thus the time complexity is O(n) where n is the number of nodes in the tree.									
		9	The first call to appendToNew takes 1 copy. The second call takes 2 copies. The third call takes 3 copies. And so on. The total time will be sum through 1 to n, which is O(n^2)									
		10	The runtime would be O(d) where d is the number of digits in the given number. A number with d digits can have a value upto 10^d. If n = 10^d, then d = log n. Thus, time complexity is O(log n) where the log is with base 10.									
		11	If the length of the string is k, then to check if the string is inOrder or sorted , takes O(k) time. Also, suppose the length of the string is c characters. Now, to get strings of c characters and k length would be O(c^k). for example, you wish to construct astring length 3 with just two characters a and b, thus the number of strings possible would be 2^3. Similarly here , that runtime would be O(c^k). Thus, overall runtime to get all the strings of k length with c characters and check if they are sorted would be O(kc^k).									

	12	First of all the runtime for mergeSort would be O(blogb) . then , for each element in a , we are doing binary search of b - runtime would be O(a * log b). hence overall runtime is O (b log b + a log b).	
Section IX	**Interview Questions**		
Chapter 1	**Arrays and Strings**		
	Hash Table		
	A hash table is a data structure that maps keys to values for efficient lookup.		
	Hash Table Implementation		
	Approach 1	We use an array of Linked Lists and a hash code function.	
		To insert a key (a string or any other datatype) and value we follow the following steps:	
		1. Compute the key's hashcode, which will usually be an itn or long. Two different keys could have the same hashcode, as there may be numerous keys but finite number of ints.	
		2. Then, we map the hash code to an index in the array. This could be done with something like hash (key) % array_length. Two different hashcodes could of course map to the same index.	
		3. At this index, there is a linked list of keys and values. Store the key and value in the index. We must use a Linked List to tackle collisions: you could have two different keys with same hashcode or two different hashcodes but same index	
		To retrieve the value pair by its key, we repeat the process. Compute the hash code by key, and then index by hashcode. Then, search through the Linked List for the value and its key.	If it is the worst case, collisions are very high, runtime would be O(N) where N is the number of keys. And, if it is the best case , collisions are minimum, look up time would be O(1)
	Approach 2	We can implement a look up system with a balanced binary search tree. This gives us O(logN) lookup time. The advantage of this approach is potentially less space , since we no longer allocate a large array. We can also iterate through keys in order.	
	ArrayList & Resizable Arrays		
	When you need an array-like datastructure with dynamic resizing, you should use an ArrayList.	A typical implementation is that when the array is full, the array doubles in size (in Java, the size might instead increase by 50% or another value). Each resizing takes O (n) time, but happens rarely that its amortized insertion time is O(1) only.	We can work backwards to compute how many elements we copied at each capacity increase to get an array of size N: N/2 + N/4 + N/8 + ...+ 2+ 1 = N. Therefore, inserting N elements takes O(N) worktotal. Thus, each insertion on an average takes O(1), even though some insertions take O(N) time in worst case.
	StringBuilder		

	Normally, concatenating n strings of x characters each would take O (xn^2).	O(x + 2x + 3x .....+ nx) = O (xn^2)									
	StringBuilder can reduce this complexity as it creates a resizeable array of all the strings, copying them to one string only if needed										
Chapter 2	**Linked Lists**										
	LinkedList is a datastructure representing a sequence of nodes.	Singly Linked List --> there is a pointer to the next node	Doubly Linked List --> there is a pointer to the next and previous nodes								
	Unlike an array, LinkedList does not provide constant time access to any element of the list. It takes iterating through K elements to get the Kth element		Benefit of a Linked List is that one can add or remove items from the beginning of the list in constant time								
Chapter 3	**Stacks and Queues**										
	Stack uses LIFO	Operations of a stack: pop(), push(item), peek(), isEmpty()	A stack does not offer constant-time access to the ith item. However, it allows constant time adds and removes as it does not require shifting elements around.	most useful case: recursive algorithms - one needs to push temporary data oto a stack as one recurses, but then remove them as one backtracks							
	Queue implements FIFO	Operations of a queue: add (item), remove(), peek(), isEmpty()	most useful case: breadth-first search and implementing a cache								