Chapter VI	Big O								
	Asymptotic notations								
	O(Big O)	Describes an upper bound on time		on the runtime; similar to a less-	then we also say that X<=	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	elements greater than pivot - this gives partial sort. then it recursively sorts the left and						
	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)						
		both the above best and worst conditions would rarely happen; thus we can expect a runtime of O(nlogn)							
		etween Asymptotic notations , Worst Case and Expected							

There is no particular relationship between the two concepts										
	10 15 110									
Best Case, Wor	st Case and Expected Case act	ually describe the big O or big	g Theta time for part	ticular scenarios w	hereas these asy	mptotic notations	describe the upp	er, lower and tigh	t bounds for the ru	ıntime
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)										
$O(x!) > O(2^x) > 0$	$O(x^2) > O(x\log x) > O(x)$									
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"				
Multiply	Nesteu A anu B	O(AB)	TOO DO THAT				
Amortized time							
	That copying might take additional O(N) time after accounting for initial O(N) time of adding the elements	into a new array does not	that the worst ca every once in a happens it won't				
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		The total runtime is then a matter of how many steps we can take before it becomes 1	$2^k = N \Rightarrow k = \log N$ 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		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 I							
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 Sorting each string would take elements of the array; thus O(a array would take O(s * aloga) a would take O(s) time in additio take O(aloga); thus adding the aloga) = O(a *s (log a + log s)	O(slogs) and we do this for a a * slogs) . Now, sorting the as each string comparison n to array sorting that would					
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						

) as the for loop stant time and runs N)) time			
process c	ne recursive alls from N to N-1 to o on until 1			
that we br time. this string. So path with	h 1: We make a tree for an example string say 'abc anch 4 times at the root , then 3 times, then 2 time gives us 4*3*2*1 leaf nodes. We could say n! leaf r , total nodes would be n * n! as each leaf node is a n nodes. Also, string concatenation will also take C me complexity in worst case would be O(n * n * n!)	es, and then 1 nodes for n length attached to a D(n) time. Thus,		
nodes; at total node 1/n!). Nov n! * e who Thus, the	n 2: At level 6, we have 6! / 0! nodes; at level 5 we level 4 we have 6!/2! nodesat level 0, we have 6! s in the tree in terms of n can be: n!(1/0! + 1/1	!/6! nodes. so, the 1/2! + 1/3!+ of Euler's number: Iropped further.		
pattern for	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 of chepth) = O(2^N).			
time. And, 2^2 + 2^3	vious example, we deduce that fib(n) taken 2^n we have fib(1) + fib(2) + fib(3) +fib(n) = 2^1 ++ 2^n = 2^n (n+1) - 2, thus we can say that the s approx. 2^n			
are doing which the reduces to and fib(i - array at ea we are do amount of	e in this program we memoization due to amount of work b looking up fib(i - 1) 2) values in memo ach call fib(i). Thus, ing a constant f work n times in n ce time complexity:			
times we until we guntil we gunte case 1. As number of	ne is the number of can divide n by 2 et down to the base s, we know the firmes we can ntil we get 1 is O			
Additional Problems				
	op iterates through ne complexity is O			
through b	sive call iterates calls as it subtracts iteration, thus time y 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						