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 $O(\log N)$ 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			
Chapter 4	**Trees and Graphs**					
	Searching a tree is more complicated than searching any linear data structure	The worst case and the average case time may vary wildly and we must evaluate both the aspects of any problem	***Tree is actually a type of graph in which cycles / loops are not possible***			
Trees	A tree is a data structure composed of nodes. In programming, each tree has a root node. The root node has zero or more children. Each child node has zero or more children and so on.		The nodes may be in any order and may have any data types as values and may or may not have links back to their parent nodes.	A node is called a leaf node if it has no children.		
Trees vs. binary trees	A binary tree is a tree in which each node has upto two children. Not all trees are binary trees.	For example, a 10-ary tree representing a bunch of phone numbers is not a binary tree.				
Binary Tree vs. Binary Search Tree	A binary search tree is a binary tree in which evry node fits a specific ordering property: all left descendants <= n < all right descendants. This must be true for each node n		This inequality condition must be true for all of a node's descendants, not just its immediate children.			
Balanced vs. unbalanced tree	not neccessarily perfectly balanced but ensures O(log n) times for insert and find	Two common types of balanced trees: Red-black trees and AVL trees				
Complete binary trees	A complete binary tree is a binary tree in which every level of the tree is fully filled, except for the last level. To the extent the last level is filled, it is filled from left to right					
Full binary tree	each node has either 2 or zero children. There is no node having only one child					

Perfect binary tree	All interior nodes have two children and all leaf nodes are at the same level.		It must have exactly 2^k - 1 nodes where k is the number of levels.				
In-order traversal	"visit" the left branch, then the current node, and finally the right node	when performed on a binary search tree, it visits the nodes in ascending order.					
Pre-order traversal	"visits" the current node before its child nodes	The root is always the first node visited					
Post-order traversal	"visits" the current node after its child nodes	The root is always the last node visited					
Binary Heaps (min-heaps and max-heaps)	A min heap is a complete binary tree where each node is smaller than its children.	The root thus is the minimum element in the tree.	Two key operations: insert and extract-min				
Insert operation on min-heap	We start by inserting at the next available spot (looking left to right on the bottommost level). We fix the tree by swapping the new element with its parent, until we find an appropriate spot for the element. We essentially bubble up the minimum spot.		This takes O(logN) time, where N is the number of nodes in the heap.				
Extract-min operation on min-heap	The minimum element of a min-heap is always at the top.	for extracting the min element, we remove the minimum element and swap it with the last element in the heap (the bottommost, rightmost element). Then, we bubble down this element, swapping it with one of its children until the min-heap property is restored.	This takes O(logN) time, where N is the number of nodes in the heap.				
Tries (Prefix trees)	A trie is a variant of an n-ary tree in which characters are stored at each node. Each path down the tree may represent a word. The * nodes (sometimes called "null nodes") are often used to indicate complete words.		The actual implementation of these * nodes might be a special type of child (such as a TreminatingTrieNode, which inherits from Trie node). Or, we could use just a boolean flag that terminates within the parent node.	A node in a trie could have anywhere from 1 through ALPHABET_SIZE + 1 children (or, 0 through ALPHABET_SIZE if a boolean flag is used instead of * node)		A trie is usally used to store the entire (English) language for quick prefix lookups.	While a hashtable can quickly look up whether a string is a valid word or not, it cannot tell us if a string is a prefix of a any valid word. A trie can do this very quickly.

	A trie can check if a string is a valid string in O(k) time where k is the length of the string.	In situations, when we search through the tree on related prefixes repeatedly, (e.g. looking up M, then MA, then MAN, then MANY), we might pass around a reference to the current node in the tree. This will allow us to check if Y is a child of MAN, rather than starting from the root each time.					
Graphs	A tree is actually a graph but not all graphs are trees. A tree is simply a connected graph without cycles.		A graph is simply a collection of nodes with edges between (some of) them.	Graphs can either be directed or undirected.	An uncyclic graph is one without cycles.		
	There are two common ways to represent a graph:	***Adjacency list***: This is the most common way to represent a graph. Every vertex or node stores a list of adjacent vertices. In an undirected graph, an edge like (a,b) would be stored twice: once in a's adjacent vertices, and once in b's vertices.	The graph class is used because, unlike in a tree, you can't necessarily reach all the nodes from a single node.	An array (or a hashtable) of lists (arrays, arraylists, linkedlists, etc.) can store the adjacency list.			
		Adjacency matrix: An adjacency matrix is a NXN boolean matrix where N is the number of nodes, a true value indicates the presence of an edge from node i to j. In an undirected graph, adjacency matrix would be symmetric whereas in a directed graph, it need not be symmetric.	All the algorithms used for adjacency lists can be used for adjacency matrices, but they would be less eficient. The reason is in adjacency list, the user can easily iterate through the neighbors of a node whereas in an adjacency matrix, the user will need to iterate through all the nodes to identify a node's neighbors.				
	Two ways to search a graph:	***Depth-first search:*** we start from the root and explore each branch completely before moving onto the next branch. It is preferred only when we wish to visit every node in the graph.	***Breadth-first search***: we start at the root and explore each neighbor before going to any children. To find the shortest path or any path between two nodes, BFS is simpler.	DFS is recursive whereas a BFS uses a queue.			
		Pre-order and other forms of traversals are an example of DFS. The key difference is we must check if a node has been visited or else we might get stuck in an infinite loop.					

	Bidirectional search	Used to find the shortest path between a source and a destination node. It operates by running two simultaneous BFS, one from each node. When their searches collide, we have found a path(formed by merging two paths). If the graph is directed, it searches forward from s and backwards from t (i.e. running the opposite way on the edges).					
	BFS vs. bidirectional search	BFS is the single search from s to t that collides after four levels whereas bidirectional search is actually two searches (one from s and another from t) that collides after four levels total (two levels each)	Basically, if every node has at most k adjacent nodes and the shortest path from node s to node t has length d, then in BFS, time complexity would be O(k^d) as it would be searching k nodes at each level whereas in bidirectional graph, time complexity would be O (k^d/2) because they would collide midway. Hence it is more time efficient.				
Chapter 5	**Bit Manipulation**						
	Bit Manipulation By Hand						
	Question	***Calculation***					
	0110 + 0010	1000					
	0011 * 0101	1111					
	0110 + 0110	1100	Equivalent to 0110 * 2 which is equivalent to shifting 0110 left by 1				
	0011 + 0010	101					
	0011 * 0011	1001					
	0100 * 0011	1100	0100 is 4 in binary numbers and multiplying by 4 is just shifting left by 2 places				
	0110 - 0011	0011					
	1101 >> 2	0011	You shift all the bits and the extra bits fall off the end. So your bits are marching right and zeros march in from the left to take their places				

	1100 ^ (~1101)	1111	it is a^(~a). if you XOR a bit with its own negated value, the result is 1				
	1000 - 0110	0010					
	1101 ^ 0101	1000					
	1011 & (~0 << 2)	1011 & 1100 = 1000	~0 is a sequence of 1's , so ~0 << 2 is 1100.				
	Bit Facts and Tricks						
	Remember the following expressions:						
	x ^ 0s = x	x & 0s = 0s	x | 0s = x				
	x ^ 1s = ~x	x & 1s = x	x | 1s = 1s				
	x ^ x = 0s	x & x = x	x | x = x				
	Two's complement Representation and Negative numbers						
	A positive number is represented as itself whereas a negative number is represented by the 2s complement of its absolute value with a 1 in its sign bit indicating that its a negative number		For an N-bit number, the two's complement is taken with respect to 2^N.				
	In other words, the binary representation of -K as a N-bit number is concat(1, 2^(N-1) - K)	For example, -3 in 4 bits: concat(1, 2^3 - 3) = concat(1, 5 which is 101) = 1101	Another way is first onvert the binary representation of the negative number, for example 3 is 011 , thus inverting it would be 100. Add 1 to it to get 101 and then prepend it with the sign bit = 1101				
	Some common representations:		The absolute values of the integers on the left and right side always sum to 2^3	The values on the left and the right are identical other than the sign bit			
	Positive values	Representation	Negative values	Representation			
	7	0111	-1	1111			
	6	0110	-2	1110			
	5	0101	-3	1101			

4	0100	-4	1100
3	0011	-5	1011
2	0010	-6	1010
1	0001	-7	1001
0	0000		

Arithmetic versus Logical Shift

Two types of right shift operations: Arithmetic shift (>>) which is essentially divide by 2 and logical shift (>>>)	In a logical shift, we shift the bits and put a 0 in the sign / most significant bit. For example, -75 i.e. 10110101 >>> 1 = 01011010 i.e. 90	In an arithmetic shift, we shift the values to the right but fill in the new bits with the value of the sign bit. For example, -75 i.e. 10110101 >> 1 = 11011010 (-38)

Common Bit Tasks And Setting

1. Get Bit

This method shifts 1 over by i bits, creating a value that looks somewhat like 00010000. By performing AND with the num, we clear all other bits but the bit at i. Finally, we compare that to 0. If the new value is not equal to zero, then bit i must be 1. Otherwise it is 0.

```
boolean getBit(int num, int i){
return ((num & (1<< i) != 0);}
```

2. Set Bit

SetBit shifts 1 over i bits, creating a value like 00010000. By performing OR operation with the given num, only the value at i bit will change and the rest will remain the same as the other bits of the mask are zero.

```
int setBit(int num, int i){
return num | (1<<i);}
```

3. ClearBit

This method operates in almost the reverse of setBit. First, we create a number like 11101111 by creating the reverse of it (00010000) and negating it. Then, we AND it with the num. This clears the bit at i and leaves the rest unchanged.

```
int clearBit(int num, int i){
int mask = ~(1<<i);
return num & mask;}
```

	To clear all the bits from the most significant bit through i (inclusive), we create a mask with a 1 at ith bit (1 << i). Then, we subtract 1 from it, which gives the sequence of zeros followed by i ones. We then AND our number with this mask to leave just the last i bits.						
	int clearBitMSBThroughI(int num, int i){						
	int mask = (1<<i) - 1;						
	return num & mask;}						
	To clear all bits from i through 0 (inclusive), we take a sequence of all ones (which is -1) and shift it left by i + 1 bits. This gives us a sequence of ones followed by i+1 zeroes						
	int clearBitsIThrough0(int num, int i){						
	int mask = (-1 << (i+1));						
	return num & mask;}						
	4. UpdateBit						
	To set the ith bit value to v, we first clear the ith position by using a mask which looks like 11101111. Then, we shift the value v left by i bits. This will create a number with bit i equal to v and all others bits equal to 0. Finally we OR these two numbers, updating the ith bit if v is 1 and leaving it 0 otherwise.						
	int updateBit(int num, int i, boolean bitIs1){						
	int value = bitIs1 ? 1 : 0;						
	int mask = ~(1 << i);						
	return (num & mask) | (value << i);}						
Chapter 6	**Math and Logic Puzzles**						
	Prime Numbers:						
	Every positive integer can be decomposed into a product of primes.	For example, 84 = 2^2 * 3^1 * 5^0 * 7^1 * 11^0 * 13^0 *17^0....					
	Divisibility						
	Prime law states that in order for a number x to divide a number y or mod(y, x) = 0, all primes in x's prime factorization must be in y's prime factorization.	Let x = 2^j0 * 3^j1 * 5^j2 * 7^j3....., y = 2^k0 * 3^k1 * 5^k2 * 7^k3..., if x/y, then for all i, ji <= ki	In fact, the greatest common diviisor, gcd(x,y) = 2^min(j0, k0) * 3^min(j1,k1) * 5^min(j2, k2)....	least common multiple would be lcm(x,y) = 2^max(j0,k0) * 3^max(j1,k1) * 5^max(j2,k2).....			

		Thus, gcd * lcm = 2^(min(j0, k0) + max(j0,k0)) * 3^(min(j1, k1) + max(j1,k1)) * 5^(min(j2, k2) + max(j2,k2))...	thus, gcd * lcm = 2^(j0 + k0) * 3^(j1 + k1) * 5^(j2 + k2)..... = 2^j0 * 3^j1 * 5^j2..... * 2^k1 * 3^k2 * 5^k2.... = xy				
	Checking for primality						
	A naive way would be to check for divisibility with integers from 2 to n-1	A slightly improved way is to iterate through 2 to sqrt(n)	The root(n) is sufficient, because, for every prime number a which divides n evenly, there is a complement b, where a*b = n. If a>root(n), then b<root (n). We do not need to check for a's primality since we would have checked for b's primality.				
	In reality, all we really need to check is if n is divisible by a prime number. This is where Sieve of Eratosthenes comes in.						
	Generating a list of primes numbers: Sieve of Eratosthenes						
	It recognizes that all non prime numbers are divisible by prime numbers.	We start with a list of all the numbers up through some value max. first, we cross off all the numbers divisible by 2, then we look for next prime number, and cross off all numbers divisible by it and so on. By crossing off all the numbers, divisible by 2,3,5,7,11 and so on till max, we wind up with a list of prime numbers from 2 through max					