S.No.			
1	**Algorithm**	**Program**	
	Design time	Implementation time	
	Domain knowledge	Programmer	
	Any language even English and Maths	Programming language	
	Hardware and software independent	Hardware and operating system dependent	
	Analyze an algorithm	Testing of programs	
2	**Priori Analysis**	**Posterior Testing**	
	Algorithm	Program	
	Independent of language	Language dependent	
	Hardware independent	Hardware dependent	
	Time and space function	watch time and bytes	
3	**Characteristics of algorithm**		
	Zero or more inputs		
	Must generate atleast one output		
	Definiteness		
	Finiteness		
	Effectiveness		
4	**How to analyze an algorithm**		
	Time		
	Space		
	Network consumtion : Data transfer amount		
	Power consumption		
	CPU registers		
5	**Frequency Count Method**	Used for time snalysis of an algorithm	
	Assign 1 unit of time for each statement		

	For any repitition, calculate the frequency of repetition		
	for(i = 0; i < n; i++) --> condition is checked for n+1 times	2n + 2 units of time ~ n+1 as we see condition i < n only for now	
	any statement within the loop will execute for n times		
	Space complexity depends upon number and kind of variables used		
6	**Algorithm : sum(A, n)**		
	Single for loop -		
	Time complexity: O(N)		
	Space complexity: O(N)		
7	**Algorithm : Add(A, B, n)**	Sum of two square matrices of dimesions nXn	
	Two nested for loops -		
	Time complexity: O(N^2)		
	Outer for loop executes for N+1 times		
	Inner for loop xecutes for N *(N+1) times		
	Any statement within inner for loop executes for (N + 1) * (N + 1) times		
	Space complexity: O(N^2)		
8	**Algorithm : Multiply(A, B, n)**		
	Three nested for loops -		
	Time complexity: O(N^3)		
	Space complexity: O(N^2)		
9	**Different algorithm conditions**		
	For loops		
	for(i = n; i > 0; i--)	n+1 times	

	for(i = 0; i < n; i = i + 2)	n/2 times	
	2 nested for loops where both i and j range from 0 to n	n^2 times	
	2 nested for loops where j ranges from 0 to i	when i = 0; j loop repeats 0 times; when i = 1; j loop repeates 1 times; and so on...total number of repetitions: 0 + 1+ 2+3+4....+n = O(n^2)	
	p = 0; for(i = 1; p<= n; i++){ p = p + i; }	p = k(k+1)/2 --> assuming that the loop exits when p is greater than n --> k(k+1) / 2 > n	~ k^2 > n --> O(root(n))
	for(i = 1; i < n; i = i *2)	will execute for 2^k times	O(logn)
		Assume i >= n ; i = 2^k >= n	
		k = logn with base 2	
	for(i = n; i >= 1; i = i/2)	i	
		n	
		n/2	
		n/2^2	
		n/2^3	
		.....	
		n/2^k	
		Assume i < 1 => n / 2^k < 1	~ O(logn) with base 2
	for(i = 0; i * i < n; i++)	i*i < n	
		i*i > -n	
		i^2 = n --> i = root(n)	~O(root(n))
	for(i = 0; i < n; i++) {.....}for(j = 0; j < n ; j++){....}	O(n)	
	p = 0; for(i = 1; i < n; i*2){.....} for(j = 1; j < p; j*2){.....}	log n times for upper loop; lop p times for lower loop	~ O(log(logn))
	for(i = 0; i < n; i++) {.....for(j = 0; j < n ; j*2){....}}	Outer loop repeats n times; inner loop repeats logn times	~O(nlogn)
	for(i = 1; i < n; i = i*3)		~O(logn) with base 3
	While loops		
	while vs. do while	do while will execute for minimum one time	

	for and while are almost similar	do while will execute as long as the condition is true; for loop will execute until the condition is false	
	a = 1;		
	while(a < b){ ..... a = a *2;}	1, 2, 2^2, 2^3......2^k repetitions	~O(logb) with base 2
		assume a > b; 2^k > b ==> k = logb with base 2	
	i = n; while(i > 1) {....i = i/2;}		~O(logn) with base 2
	i = 1; k = 1; while(k <n){....k = k + i; i++;}		
	i	k	
	1	1	
	2	1 + 1	
	3	2 + 2	
	4	2 + 2 + 3	
	5	2 + 2 + 3 + 4	
	.....		
	m	m(m + 1) /2	
	Assume, k >= n	m(m + 1)/ 2 >= n	~O(root(n))
	while(m != n) { if(m > n) m = m - n; else n = n - m;}		~O(n)
10	**Types of time functions**		
	O(1) --- constant		
	O(logn) --- logarithmic		
	O(n) --- linear		
	O(n^2) --- quadratic		
	O(n^3) --- cubic		
	O(2^n) --- exponential		

11	**Order of complexity**		
	1 < logn < root(n) < n < nlogn < n^2 < n^3 <.....< 2^n < 3^n .....< n^n		
12	**Asymptotic Notations**		
	Representation of time omplexity in simple form which is understandable		
	Big O Notation - works as an upper bound	The function f(n) = O(g(n)) iff for all positive constants c and n_0, such that f(n) <= c * g (n) for all n >= n_0; here, f(n) = O(n)	e.g. 2n + 3 <= 10n; All those functions in time order complexity above n become upper bound; below n become lower bound and n is the average bound
	Big Omega Notation - works as a lower bound	The function f(n) = Omega(g(n)) iff for all positive constants c and n_0, such that f(n) >= c * g(n) for all n >= n_0; here, f(n) = Omega(n)	e.g. 2n + 3 >= 1n
	Theta Notation - works as an average bound	The function f(n) = theta(g(n)) iff for all positive constants c1, c2 and n0 such that c1 * g(n) <= f(n) <= c2 * g(n)	e.g. f(n) 2n + 3; 1n <= 2n + 3 <= 5n
	Most useful is theta notation, then why do we need the other two?	In case we are not able to get the average bound, then we point to its upper or lower bound	
13	**Examples for asymptotic notations**		
a	**f(n) = 2n^2 + 3n + 4**		
	2n^2 + 3n + 4 <= 2n^2 + 3n^2 + 4n^2 i.e. 9n^2	O(n^2)	
	2n^2 + 3n + 4 >= 1n^2	Omega(n^2)	
	1n^2 <= 2n^2 + 3n + 4 <= 9n^2	Theta(n^2)	
b	**f(n) = n^2logn + n**		
	n^2logn <= n^2logn + n <= 10n^2logn	O(n^2logn)	
		Omega(n^2logn)	
		Theta(n^2logn)	

c	**f(n) = n!**		
	1 <= 1*2*3*4....*n-1*n <= n*n*n*n*......*n	O(n^n)	
		Omega(1)	
		Cannot find theta for n!	
d	**f(n) = logn!**		
	1 <= log(1*2*3.....*n) <= log(n*n*n*n....*n)	O(logn^n)	
		Omega(1)	
		Cannot find theta for logn!	
14	**Properties of Asymptotic notations**		
	General properties -		
	if f(n) is O(g(n)) then a*f(n) is O(g(n))		
	e.g. f(n) = 2n^2 + 5 is O(n^2), then 7f(n) i.e. 14n^2 + 35 is also O(n^2)	This would be true for both Omega and theta n as well	
	Reflexive property -		
	If f(n) is given then f(n) is O(f(n))		
	e.g. f(n) = n^2 then O(n^2)	A function is an upper bound of itself	
		Similarly, a function is a lower bound of itself	
	Transitive property -		
	If f(n) isO(g(n)) and g(n) is O(h(n)) then f(n) is O(h(n))		
	e.g. f(n) = n; g(n) = n^2 and h(n) = n^3	True for all notations	
	n is O(n^2) and n^2 is O(n^3) then n is O(n^3)		
	Symmetric property -		
	If f(n) is theta(g(n)) then g(n) is theta(f(n))	True for only theta(n)	
	e.g. f(n) = n^2 g(n) = n^2 ; f(n) = theta(n^2) and g(n) = theta(n^2)		

	Transpose symmetric -	True for BigO and Omega notations	
	if f(n) = O(g(n)) then g(n) is Omega(f(n))		
	e.g. f(n) = n and g(n) is n^2 then n is O(n^2) and n^2 is Omega(n)		
	If f(n) = O(g(n)) and f(n) = Omega(g(n)) then g(n) <= f (n) <= g(n) therefore f(n) = theta(g(n))		
	If f(n) = O(g(n)) and d(n) = O(e(n)) then f(n) + d(n) = O(max(g(n), e(n))		
	e.g. f(n) = n = O(n) , d(n) = n^2 = O(n^2) then f(n) + d (n) = n + n^2 = O(n^2)		
	If f(n) = O(g(n)) and d(n) = O(e(n)) then f(n) * d(n) = O (g(n) * e(n))		
15	**Comparison of functions**		
	First method is substituting values for n and comparing		
	Second method is applying log on both sides		
		Properties of log -	
	Example -	logab = loga + logb	
	f(n) n^2logn; g(n) = n(logn)^10	loga/b = loga - logb	
	Apply log	loga^b = bloga	
	log(n^2log(n)); log(n(logn)^10)	a^(log_cb) = b^(log_ca)	
	log(n^2) + loglogn; logn + loglog^10	a^b = n then b = log_an	
	2logn + loglogn ; logn + 10loglogn		
	here; 2logn is greater than logn and logn is a bigger term than loglogn		
	so, first term is greater than the second one		
	f(n) = 3n^(rootn) ; g(n) = 2^(rootn log_2(n))		
	Applying log		

	3n(rootn) ; (n^rootn)log_2(2)		
	3n(rootn) ; nrootn		
	first term is greater than the second one value wise but asymptatically they are equal		
	f(n) = n^(logn); g(n) = 2^(rootn)		
	apply log,		
	log(n^logn); log(2^rootn)		
	logn*logn ; rootn (log_2(2))		
	log^2n ; rootn		
	canot judge, so apply log again		
	2loglogn; 1/2logn		
	loglogn is smaller than logn		
	thus, second term is greater		
	f(n) = 2^(logn); g(n) = n^(rootn)		
	logn*log_2(2) ; rootn*logn		
	logn ; rootn*logn		
	second term is greater		
	f(n) = 2n; g(n) is 3n		
	both are equal asymptotically		
	f(n) = 2^n; g(n) = 2^(2n)		
	applying log		
	log(2^n); log(2^2n)		
	n; 2n	after applying log, do not cut coefficients	
	second function is greater		
16	**Best, worst and average case analysis**		
	Example -		

a	**Linear search**		
	A = {8, 6, 12, 5, 9, 7, 4, 3, 16, 18} key = 7		
	In linear search, it will start checking for the given key from left hand side		
	total in 6 comparisons, we would get our key		
	Best case - key element is present at first index		
	Best case time - 1 i.e. B(n) = O(1); Omega(1); Theta (1)		
	Worst case - key element is present at the last index		
	Worst case time - n i.e. W(n) = O(n); Omega(n); Theta(n)		
	Average case = all possible case time / no. of cases		
	average case analysis is very difficult for most of the cases		
	Here, average case time = 1 +2 + 3.... + n/2 = n(n+1) /2n = n+1/2		
	A(n) = n+1/2		
b	**Binary search tree**		
	height = logn		
	time taken for a particular key is logn		
	Best case - element present in the root		
	Best case time - k i.e. B(n) = O(1); Omega(!); Theta (1)		
	Worst case - searching for a leaf element - depends upon the height of the tree		
	Worst case time - logn i.e. O(logn)		
	min w(n) = logn; max w(n) = n		
17	**Disjoint sets**		
	No common numbers between two sets - intersection is zero		
	Operations - find, union		

	Find - search or check membership		
	Union - Add an edge		
	If you take an edge and both the vertices belong to the same set, then there is a cycle in the graph		