

<u>S.No.</u>			
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 consumption : 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 repetition, 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 dimensions nXn	
	Two nested for loops -		
	Time complexity: O(N ²)		
	Outer for loop executes for N+1 times		
	Inner for loop executes 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 repeats 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; log 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		
	$1 < \log n < \sqrt[n]{n} < n < n \log n < n^2 < n^3 < \dots < 2^n < 3^n \dots < n^n$		
12	Asymptotic Notations		
	Representation of time complexity 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 $f(n) \leq c * g(n)$ for all $n \geq n_0$; here, $f(n) = O(n)$	e.g. $2n + 3 \leq 10n$; 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 $f(n) \geq c * g(n)$ for all $n \geq n_0$; here, $f(n) = \Omega(n)$	e.g. $2n + 3 \geq 1n$
	Theta Notation - works as an average bound	The function $f(n) = \theta(g(n))$ iff for all positive constants c_1, c_2 and n_0 such that $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$	e.g. $f(n) 2n + 3; 1n \leq 2n + 3 \leq 5n$
	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	$f(n) = 2n^2 + 3n + 4$		
	$2n^2 + 3n + 4 \leq 2n^2 + 3n^2 + 4n^2$ i.e. $9n^2$	$O(n^2)$	
	$2n^2 + 3n + 4 \geq 1n^2$	$\Omega(n^2)$	
	$1n^2 \leq 2n^2 + 3n + 4 \leq 9n^2$	$\Theta(n^2)$	
b	$f(n) = n^2 \log n + n$		
	$n^2 \log n \leq n^2 \log n + n \leq 10n^2 \log n$	$O(n^2 \log n)$	
		$\Omega(n^2 \log n)$	

		$\Theta(n^2 \log n)$	
c	$f(n) = n!$		
	$1 \leq 1 \cdot 2 \cdot 3 \cdot 4 \dots \cdot n-1 \cdot n \leq n \cdot n \cdot n \cdot \dots \cdot n$	$O(n^n)$	
		$\Omega(1)$	
		Cannot find theta for $n!$	
d	$f(n) = \log n!$		
	$1 \leq \log(1 \cdot 2 \cdot 3 \dots \cdot n) \leq \log(n \cdot n \cdot n \dots \cdot n)$	$O(\log n^n)$	
		$\Omega(1)$	
		Cannot find theta for $\log n!$	
14	Properties of Asymptotic notations		
	General properties -		
	if $f(n)$ is $O(g(n))$ then $a \cdot f(n)$ 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 Ω and θ 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 $f(n)$ is $O(g(n))$ 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 $g(n) \leq f(n) \leq g(n)$ 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 -	$\log a b = \log a + \log b$	
	$f(n) = n^2 \log n$; $g(n) = n(\log n)^{10}$	$\log a/b = \log a - \log b$	
	Apply log	$\log a^b = b \log a$	
	$\log(n^2 \log n)$; $\log(n(\log n)^{10})$	$a^{(\log_{cb})} = b^{(\log_{ca})}$	
	$\log(n^2) + \log \log n$; $\log n + \log \log^{10} n$	$a^b = n$ then $b = \log_a n$	
	$2 \log n + \log \log n$; $\log n + 10 \log \log n$		

	here; $2\log n$ is greater than $\log n$ and $\log n$ is a bigger term than $\log \log n$		
	so, first term is greater than the second one		
	$f(n) = 3n^{\sqrt{n}}$; $g(n) = 2^{(\sqrt{n} \log_2(n))}$		
	Applying log		
	$3n^{\sqrt{n}}$; $(n^{\sqrt{n}})\log_2(2)$		
	$3n^{\sqrt{n}}$; $n^{\sqrt{n}}$		
	first term is greater than the second one value wise but asymptotically they are equal		
	$f(n) = n^{\log n}$; $g(n) = 2^{\sqrt{n}}$		
	apply log,		
	$\log(n^{\log n})$; $\log(2^{\sqrt{n}})$		
	$\log n \cdot \log n$; $\sqrt{n} (\log_2(2))$		
	$\log^2 n$; \sqrt{n}		
	cannot judge, so apply log again		
	$2\log \log n$; $1/2 \log n$		
	$\log \log n$ is smaller than $\log n$		
	thus, second term is greater		
	$f(n) = 2^{\log n}$; $g(n) = n^{\sqrt{n}}$		
	$\log n \cdot \log_2(2)$; $\sqrt{n} \cdot \log n$		
	$\log n$; $\sqrt{n} \cdot \log n$		
	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 = $1 + 2 + 3 + \dots + n/2 = \frac{n(n+1)}{2n} = \frac{n+1}{2}$		
	$A(n) = \frac{n+1}{2}$		
b	Binary search tree		
	height = $\log n$		
	time taken for a particular key is $\log n$		
	Best case - element present in the root		
	Best case time - k i.e. $B(n) = O(1)$; $\Omega(1)$; $\Theta(1)$		

	Worst case - searching for a leaf element - depends upon the height of the tree		
	Worst case time - $\log n$ i.e. $O(\log n)$		
	min $w(n) = \log n$; 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		
	<i>Kruskal algorithm: If you take an edge and both the vertices belong to the same set, then there is a cycle in the graph</i>		
	Weighted union is used while adding edges and detecting cycle		
	Collapsing find - process of directly linking node to a direct parent of a set is called collapsing find - reduces the time to find		
18	Divide and conquer - Strategy 1		
	Strategy - an approach for solving a problem		
	If a problem cannot be solved, divide it into sub-problems and find a solution for each sub problem, combine the solutions. <i>One point to note is that each sub problem should be similar to the original problem only.</i>		
	Recursive in nature		
	Should have one method to combine the solutions of each sub problem		
19	Problems under Divide and Conquer		
	Binary search		
	Finding maximum and minimum		

	MergeSort		
	QuickSort		
	Strassen's matrix multiplication		
20	Recurrence relation 1: $T(n) = T(n-1) + 1$		
	void test(int n)		
	{		
	if(n > 0){		
	printf("%d",n);		
	test(n-1)		
	}		
	}		
	test(3)		
	3. test(2)		
	2. test(1)		
	1 test(0)		
	each print statement takes constant time 1 and there are n+ 1 calls made to the function. we can ignore the last call when it is not printing		
	$f(n) = n + 1$ calls ; $O(n)$		
	$T(n) = T(n-1) + 1$; if we ignore if condition		
	Let us solve this relation;		
	if we know $T(n-1)$, we can get $T(n)$		
	$T(n-1) = T(n-2) + 1$		
	$T(n) = [T(n-2) + 1] + 1$		
	$T(n) = T(n-3) + 3$		
continue for k times		
	$T(n) = T(n-k) + k$		
	We would stop after k substitutions; now we need to find k		

	Assume $n - k = 0$; therefore $n = k$		
	$T(n) = T(n-n) + n$		
	$T(n) = T(0) + n$		
	$T(n) = n + 1$ i.e. $\theta(n)$		
21	Recurrence relation 2: $T(n) = T(n-1) + n$ (decreasing function)		
	void test(int n)	$T(n)$	
	{		
	if(n > 0)		1
	{		
	for(i = 0; i < n; i++)	$n+1$	
	{		
	printf("%d", n);	n	
	}		
	test(n-1);	$T(n-1)$	
	}		
	}		
		$T(n) = T(n-1) + 2n + 2$ i.e. $\theta(n)$	
	we can also write $T(n) = T(n-1) + n$ for $n > 0$		
	$T(n) = 1$ for $n = 0$		
	$T(n)$	n time	
	$n \quad T(n-1)$	$n-1$ time	
	$n-1. \quad T(n-2)$	$n - 2$ time	
	$n-2 \quad T(n-3)$	$n - 3$ time	
		
	$T(2)$		
	2 $T(1)$	2 units of time	
	1 $T(0)$	1 unit of time	
	for $T(0)$ it does nothing	0 unit of time	
	time taken -		

		$0 + 1 + 2 + \dots + n-1 + n$	
	$\theta(n^2)$	$T(n) = n(n+1)/2$	
	$T(n) = T(n-1) + n$		
	$T(n-1) = T(n-2) + n-1$		
	thus, $T(n) = T(n-2) + (n-1) + n$	**remember, don't add the terms	
	$T(n) = T(n-3) + (n-2) + (n-1) + n$		
	$T(n) = T(n-k) + (n-(k-1)) + (n-(k-2)) + \dots + (n-1) + n$	if we continue for k times	
	assume $n - k = 0$; $n = k$		
	Thus, $T(n) = T(n-n) + (n - n + 1) + (n - n + 2) + \dots + (n-1) + n$		
	$T(n) = T(0) + n(n+1)/2$		
	$T(n) = 1 + n(n+1)/2$	$\theta(n^2)$; this extra 1 is owing to the calls	
22	Recurrence relation 3: $T(n) = T(n-1) + \log n$		
	void test(int n)	$T(n)$	
	{		
	if(n>0)		
	{		
	for(i = 1; i < n; i = i*2)		
	{		
	printf("%d", i);	log n times	
	}		
	test(n-1);	$T(n-1)$	
	}		
	}		
	$T(n) = T(n-1) + \log n$ for $n > 0$		
	$T(n) = 1$ for $n = 0$		
	Solve using tree method,		

	$T(n)$		
	$\log n \quad T(n-1)$		
	$\log(n-1) \quad T(n-2)$		
	$\log(n-2) \quad T(n-3)$		
		
	$\log 2 \quad T(1)$		
	$\log 1 \quad T(0)$		
	$\log n + \log(n-1) + \dots + \log 2 + \log 1$		
	$\log[n(n-1)(n-2)\dots 2.1] = \log(n!)$	there is no tight bound for this function but there is an upper bound for it	
	$O(n \log n)$		
	<i>Solving using induction method.</i>		
	$T(n) = T(n-1) + \log n$		
	$T(n) = T(n-2) + \log(n-1) + \log(n)$		
	$T(n) = T(n-3) + \log(n-2) + \log(n-1) + \log n$		
		
	$T(n) = T(n-k) + \log n + \log(n-1) + \dots \log 1$		
	Asume $n-k = 0$		
	$T(n) = T(0) + \log n!$		
	$T(n) = 1 + \log n!$		
	$O(n \log n)$		
23	<i>How to get the direct answer for a recurrence relation?</i>		
	$T(n) = T(n-1) + 1$	$O(n)$	
	$T(n) = T(n-1) + n$	$O(n^2)$	
	$T(n) = T(n-1) + \log n$	$O(n \log n)$	
	$T(n) = T(n-1) + n^2$	$O(n^3)$	
	$T(n) = T(n-2) + 1$	$O(n/2) \sim O(n)$	

	$T(n) = T(n-100) + n$	$O(n^2)$	
	$T(n) = 2T(n-1) + 1$???	
24	Recurrence relation 4: $T(n) = 2T(n-1) + 1$		
	Test(int n)	$T(n)$	
	{		
	if(n > 0)		1
	{		
	printf("%d", n);		1
	test(n-1);	$T(n-1)$	
	test(n-1);	$T(n-1)$	
	}		
	}		
		$T(n) = 2T(n-1) + 1$	
	$T(n) = 2T(n-1) + 1$ for $n > 0$		
	$T(n) = 1$ for $n = 0$		
	Solve using recursion tree method		
	1 $T(n-1)$ $T(n-1)$		2
	1 $T(n-2)$ $T(n-2)$	1 $T(n-2)$ $T(n-2)$	4
	1 $T(n-3)$ $T(n-3)$ 1 $T(n-3)$ $T(n-3)$	1 $T(n-3)$ $T(n-3)$ 1 $T(n-3)$ $T(n-3)$	8
		
	$T(0)$. $T(0)$		2^k
	$1 + 2 + 2^2 + \dots + 2^k = 2^{k+1} - 1$		
	as, $a + ar + ar^2 + \dots + ar^k = a(r^{k+1} - 1)/(r - 1)$		
	Assume $n - k = 0$		
	thus, $2^{n+1} - 1$	$O(2^{n+1})$	

	Back substitution method		
	$T(n) = 2T(n-1) + 1$		
	$T(n) = 4T(n-2) + 2 + 1$		
	$T(n) = 8T(n-3) + 4 + 2 + 1$		
		
	$T(n) = 2^k T(n-k) + 2^{k-1} + 2^{k-2} + \dots + 2^3 + 2^2 + 1$		
	Assume $n - k = 0$		
	$n = k$		
	$T(n) = 2^n T(0) + 1 + 2 + 2^2 + \dots + 2^{n-1}$		
	$T(n) = 2^n + 2^n - 1$ i.e. $2^{n+1} - 1$		
25	Master theorem for decreasing function		
	$T(n) = T(n-1) + 1$	$O(n)$	
	$T(n) = T(n-1) + n$	$O(n^2)$	
	$T(n) = T(n-1) + \log n$	$O(n \log n)$	
	$T(n) = 2T(n-1) + 1$	$O(2^n)$	
	$T(n) = 3T(n-1) + 1$	$O(3^n)$	
	$T(n) = 2T(n-1) + n$	$O(n2^n)$	
	$T(n) = 2T(n-2) + 1$	$O(2^{n/2})$	
	$T(n) = aT(n-b) + f(n)$		
	$a > 0, b > 0$ and $f(n) = O(n^k)$ where $k \geq 0$		
	if $a = 1, O(n^{k+1})$ or $O(n \cdot f(n))$		
	if $a > 1, O(n^k \cdot a^{n/b})$		
	if $a < 1, O(n^k)$ or $O(f(n))$		
26	Dividing functions		
	test(int n)	$T(n)$	
	{		

if(n > 1)		
{		
printf("%d", n);		1
test(n/2)	$T(n/2)$	
}		
}		
$T(n) = T(n/2) + 1$ for $n > 1$		
$T(n) = 1$ for $n = 1$		
$T(n)$		
1 $T(n/2)$		
1 $T(n/2^2)$		
1 $T(n/2^3)$		
.....continue for k times		
1 $T(n/2^k)$		
assume , $n/2^k = 1$		
thus, we have taken k steps overall		
since, $n/2^k = 1 \Rightarrow k = \log n$ with base 2	$O(\log n)$	
Solving by substitution method		
$T(n) = T(n/2) + 1$		
$T(n) = T(n/2^2) + 2$		
$T(n) = T(n/2^3) + 3$		
....		
$T(n) = T(n/2^k) + k$		
assume $n/2^k = 1$		
thus, $k = \log n$ with base 2		
$T(n) = T(1) + \log n$		

	$O(\log n)$		
27	Recurrence relation: $T(n) = T(n/2) + n$		
	$T(n) = T(n/2) + n$ for $n > 1$		
	$T(n) = 1$ for $n=1$		
	$T(n)$		
	$T(n/2)$ n		
	$T(n/2^2)$ $n/2$		
	$T(n/2^3)$ $n/2^2$		
		
	$T(n/2^k)$. $n/2^{(k-1)}$		
	$T(n) = n + n/2 + n/2^2 + n/2^3..... + n/2^k$		
	$T(n) = n[1 + 1/2 + 1/2^2 + 1/2^3 +.....1/2^k]$		
	$T(n) = n * 1 = n$		
	$O(n)$		
	Using substitution method		
	$T(n) = T(n/2) + n$		
		
	$T(n) = T(n/2^2) + n/2 + n$		
		
	$T(n) = T(n/2^3) + n/2^2 + n/2 + n$		
		
	$T(n) = T(n/2^k) + n/2^{k-1} + n/2^2 + n/2 + n$		
	Assume $n/2^k = 1$		
	$k = \log n$ with base 2		
	$T(n) = T(1) + n[1/2^{k-1}.....+1/2^2.....+1]$		
	$T(n) = 1 + 2n \sim O(n)$		

28	Recurrence Relation: $T(n) = 2T(n/2) + n$		
	void test(int n)	$T(n)$	
	{		
	if(n > 1)		
	{		
	for(int i = 0; i < n ; i++)		
	{		
	stmt	n	
	}		
	test(n/2);	$T(n/2)$	
	test(n/2);	$T(n/2)$	
	$T(n) = 2T(n/2) + n$ for $n > 1$		
	$T(n) = 1$ for $n = 1$		
	<i>Solve using recursion tree method,</i>		
	$T(n)$		
	$T(n/2).$ $T(n/2)$ n	n	
	$T(n/2^2)$ $T(n/2^2)$ $T(n/2^2)$ $T(n/2^2)$ $n/2$	n	
	$T(n/2^3).$ $T(n/2^3).$ $T(n/2^3).$ $T(n/2^3).$ T $(n/2^3).$ $T(n/2^3).$ $T(n/2^3).$ $T(n/2^3).$	n	
	n	
	$T(n/2^k).....$		
		n	
	assume $n / 2^k = 1$		
	$k = \log n$ with base 2		
	$T(n) = nk \sim O(n \log n)$		
	<i>Using backsubstitution method;</i>		

	$T(n) = 2T(n/2) + n$		
	$T(n/2) = 2T(n/2^2) + n/2$		
	$T(n) = 2[2T(n/2^2) + n/2] + n$		
	$T(n) = 2^2T(n/2^2) + n + n$		
	$T(n) = 2^3T(n/2^3) + 3n$		
continue for k times		
	$T(n) = 2^kT(n/2^k) + kn$		
	Asume $T(n/2^k) = T(1)$		
	$k = \log n$ with base 2		
	Thus, $T(n) = n + n \log n \sim O(n \log n)$		
29	Masters Theorem for dividing functions		
	$T(n) = aT(n/b) + f(n)$	\log_a with b	
	$a > 1; b > 1; f(n) = \theta(n^k \log^p n)$	k	
	case 1: if \log_a with base b > k then $\theta(n^{\log_a \log b})$		
	case 2: if \log_a with base b = k then		
	if $p > -1$ $\theta(n^k \log^{p+1} n)$		
	if $p = -1$ $\theta(n^k \log \log n)$		
	if $p < -1$ then $\theta(n^k)$		
	case 3: if \log_a with base b < k		
	then, if $p \geq 0$, $\theta(n^k \log^p n)$		
	if $p < 0$, $\theta(n^k)$		
	$T(n) = 2T(n/2) + 1$		
	$a = 2$		
	$b = 2$		
	$f(n) = \theta(n^0 \log^0 n)$		
	$k = 0; p = 0$		
	here, \log_a with base b > k		

	$\theta(n^1)$ where \log_a with base b is 1		
	$T(n) = 4T(n/2) + n$		
	\log_a with base $b = 2$		
	$k = 1$		
	$p = 0$		
	this is an example of case 1		
	$\theta(n^2)$		
	$T(n) = 8T(n/2) + n$		
	\log_8 with base $2 = 3 > k = 1$		
	$\theta(n^3)$		
	$T(n) = 9T(n/3) + 1$		
	\log_a with base $b = 2 > k$		
	$\theta(n^2)$		
	$T(n) = 9T(n/3) + n^2$		
	\log_a with base $b = 2 = k$	case 2	
	$\theta(n^2)$		
	$T(n) = 8T(n/2) + n$		
	$\theta(n^3)$		
	$T(n) = 2T(n/2) + n$		
	\log_a with base $b = k = 1$; $p = 0$		
	case 2		
	$\theta(n \log n)$		
	$T(n) = 4T(n/2) + n^2$		

	$\theta(n^2 \log n)$		
	$T(n) = 4T(n/2) + n^2 \log n$		
	$\theta(n^2 \log n^2)$		
	$T(n) = 8T(n/2) + n^3$		
	$\theta(n^3 \log n)$		
	$T(n) = 2T(n/2) + n/\log n$		
	$\log a$ with base $b = k = 1$		
	$p = -1$		
	$\theta(n \log \log n)$		
	$T(n) = 2T(n/2) + n/\log n^2$		
	$p = -2$		
	$\theta(n)$		
	$T(n) = 2T(n/2) + n^2$		
	$\log a$ with base $b < k$		
	$\theta(n^2)$		
	$T(n) = 2T(n/2) + n^2$		
	$\theta(n^2 \log n)$		
	$T(n) = 2T(n/2) + n^3$		
	$\log a$ with base $b < k$		
	$\theta(n^3)$		
30	$T(n) = 2T(n/2) + 1$		
	$\log a$ with base $b = 1$		

	$k = 0$		
	loga with base $b > k$		
	$\theta(n^1)$		
	$T(n) = 4T(n/2) + 1$		
	loga with base $b = 2$		
	$k = 0$		
	$\theta(n^2)$		
	$T(n) = 4T(n/2) + n$		
	loga with base $b = 2$		
	$k = 1$		
	$\theta(n^2)$		
	$T(n) = 8T(n/2) + n^2$		
	loga with base $b = 3$		
	$k = 2$		
	$\theta(n^3)$		
	$T(n) = 16T(n/2) + n^2$		
	loga with base $b = 4$		
	$k = 2$		
	$\theta(n^4)$		
	$T(n) = T(n/2) + n$		
	log a with base $b = 0$		
	$k = 1$		
	$\theta(n)$		
	$T(n) = 2T(n/2) + n^2$		

	$\log a$ with base $b = 1$		
	$k = 2$		
	$\theta(n^2)$		
	$T(n) = 2T(n/2) + n^2 \log n$		
	$\log a$ with base $b = 1$		
	$k = 2$		
	$\theta(n^2 \log n)$		
	$T(n) = 4T(n/2) + n^3 \log^2 n$		
	$\log a$ with base $b = 2$		
	$k = 3$		
	$\theta(n^3 \log^2 n)$		
	$T(n) = 2T(n/2) + n^2 / \log n$		
	$\log a$ with base $b = 1$		
	$k = 2$		
	$\theta(n^2)$		
	$T(n) = T(n/2) + 1$		
	$\log a$ with base $b = 0$		
	$k = 0$		
	$\theta(\log n)$		
	$T(n) = 2T(n/2) + n$		
	$\log a$ with base $b = 1$		
	$k = 1$		
	$p = 0$		
	$\theta(n \log n)$		

	$T(n) = 2T(n/2) + n \log n$		
	log a with base b = 1		
	k = 1		
	p = 1		
	$\theta(n \log^2 n)$		
	$T(n) = 4T(n/2) + n^2$		
	log a with base b = 2		
	k = 2; p = 0		
	$\theta(n^2 \log n)$		
	$T(n) = 4T(n/2) + (n \log n)^2$		
	log a with base b = 2		
	k = 2, p = 2		
	$\theta(n^2 \log^3 n)$		
	$T(n) = 2T(n/2) + n/\log n$		
	log a with base b = 1		
	k = 1; p = -1		
	$\theta(n \log \log n)$		
	$T(n) = 2T(n/2) + n/\log^2 n$		
	log a with base b = 1		
	k = 1; p = -2		
	$\theta(n)$		
31	Root function Recurrence relation		
	$T(n) = T(\sqrt{n} + 1)$ for $n > 2$		
	$T(n) = 1$ for $n = 2$		

	$T(n) = T(\text{root}(n)) + 1$		
	$T(n) = T(n^{(1/2)}) + 1$equation 1		
	using substitution		
	$T(n) = T(n^{(1/2^2)}) + 2$equation 2		
	$T(n) = T(n^{(1/2^3)}) + 3$equation 3		
	$T(n) = T(n^{(1/2^k)}) + k$equation 4		
	assume, $n = 2^m$		
	$T(2^m) = T(2^{(m/2^k)}) + k$		
	assume $T(2^{(m/2^k)}) = T(2)$		
	thus, $m/2^k = 1$		
	$m = 2^k$		
	$k = \log m$ with base 2		
	substituting value of n		
	$m = \log n$ with base 2		
	therefore, $k = \log \log n$ with base 2		
	$\theta(\log \log n \text{ with base } 2)$		
	32 Binary Search Iterative Method		
	To perform binary search, the prerequisite is that the list must be in sorted order	$A = \{3, 6, 8, 12, 14, 17, 25, 29, 31, 36, 42, 47, 53, 55, 62\}$	
	we need two index pointers, one is low at the starting point and the other is high at the end point	$l = 1, h = 15$ (lowest and highest index); $mid = 8$	
	$mid = \text{low} + \text{high} / 2$ and we take the floor value	key value = 42; $A[mid] = 29 \rightarrow \text{key} > A[mid]$	
	the key value is on the right hand side as key value is greater than $A[mid]$		
	we will change low to $mid + 1$	$l = 9, h = 15; mid = 9 + 15 / 2 = 12$	
		$A[mid] = 47 > \text{key}$	
	we will change high to $mid - 1$ as $\text{key} < A[mid]$		

		h = 11, l = 9, mid = 10; A[mid] = 36	
		A[mid] < key	
	we will change low to mid + 1	l = 11; h = 11; mid = 11; A[mid] = 42	
	we can return the index as we have found the key value	A[mid] = key	
	therefore, binary search looks faster than linear search. It just took 4 comparisons		
	int BinSearch(A, n, key)		
	{		
	l = 1, h = n		
	mid = l + h / 2 - take floor value		
	while(l <= h){		
	if(key == A[mid])		
	{ return index i.e.element is found}		
	else if(key < A[mid])		
	{h= mid-1;}		
	else {		
	l = mid + 1;}		
	}		
	return 0;		
	}		
	Time taken for binary search = logn		
	min time: O(1)		
	max time: O(logn)		
	avg time = add time for each element and divide by number of elements		
	33 Binarysearch Recursive method		
	Algorithm RBinarySearch(l,h,key)	T(n)	

	{		
	if(l==h)	1	
	{		
	if(A[low]== key)		
	{		
	return l;		
	}		
	else		
	{		
	return 0;		
	}		
	else		
	{		
	mid = l + h / 2 //taking floor value	1	
	if(key == A[mid])	1	
	{return mid;}		
	if(key < A[mid])	1	
	{		
	return RBinarySearch(l, mid - 1, key)	$T(n/2)$	
	}		
	else		
	{		
	return RBinarySearch(mid+1, h, key)	$T(n/2)$	
	}		
	}		
		$T(n) = 1; n = 1$	
		$T(n) = T(n/2) + 1$ for $n > 1$	
		$\theta(\log n)$	

34	Heaps		
a	Representation of a binary tree using an array		
	T {A, B, C, D, E, F, G}		
	if a node is at index i;		
	its left child is at node $2*i$		
	its right child is at node $2*i + 1$		
	its parent is at node $i/2$		
	if there are missing nodes, we leave a blank in its place in the array		
b	Full binary tree		
	In its height, it has maximum number of nodes and if we wish to add a node, height would increase		
	Max no. of nodes = $2^h - 1$		
c	Complete binary tree		
	there is no missing element from first element to the last element in array representation of the binary tree		
	Every full binary tree is also a complete binary tree		
	A complete binary tree is a full binary tree until height $h - 1$		
	Height of a complete binary tree would be minimum i.e. $\log n$		
d	Heap		
	Heap is a complete binary tree		
	Max Heap: every node has value greater than all its descendants {50, 30, 20, 15, 10, 8, 16}		
	Min Heap: every node has value smaller or equal to than all its descendants {10, 30, 20, 35, 40, 32, 25}		

35	Insert operation in a max heap		
	Insert 60 in the above given max heap		
	this value should be inserted in the last free space in the array		
	i.e. left child of the left most leaf node		
	Then, adjust the elements to make it as a heap		
	So, compare and move 60 up the levels and in the array check at $i/2$ indices where initially i would be the last empty index where 60 was inserted		
	Time taken would be equal to the number of swaps		
	this depends upon the height of the tree i.e. $\log n$, hence $O(\log n)$		
	minimum time is of no swaps $O(1)$; max would be $O(\log n)$		
36	Delete operation in a max heap		
	From the heap, we need to remove the root / top most element only		
	The last element in the complete binary tree would come in its place		
	Adjust the elements to maintain heap order		
	From the root towards the leaf, adjust		
	Compare the children ($2i$ and $2i + 1$) and whichever child is greater than compare with the parent		
	Time taken depends upon the height; max could be $O(\log n)$		
	Whenever you delete from max heap, you get the next max element and in case of min heap, it would be the next min element		
37	HeapSort		

	For a given set of numbers, create a heap		
	Delete all the elements from the heap		
	Total N elements we have inserted; each element we assume is moved up to the root; so time taken $O(N\log N)$		
	Then we delete the elements		
	Store deleted elements in the array in free space in the end		
	Deletion also takes $O(N\log N)$ time		
	Thus, heapsort takes $O(N\log N)$		
38	Heapify		
	The process of creating heap but direction is opposite than creating a heap		
	$O(N)$		
39	Priority Queue		
	elements will have priority and they would be inserted and deleted as per the priority order		
	For min heap, smaller the no. higher the priority		
	For max heap, greater the no. higher the priority		
	$O(\log N)$ for insertion and/or deletion		
40			