1. Why is a special notation needed to classify algorithms? Doesn't it suffice to merely measure the runtime in seconds? Explain.

Algorithmic complexity measures the performance of the given algorithm mainly in terms of time and space. Complexity is defined as a numerical function (T(n)) time versus the input size n. T(n) doesn't depend on the implementation. But time taken by an algorithm to run depend on implementation. Depending on the processor speed; instruction set, disk speed and etc a algorithm takes different amount of time to run on the same input. So we need some kind of notation that can give information about the time taken by an algorithm to run based on the input size. We consider time T(n) as the number of steps such that each such step takes a constant amount of time.

Let us take a classic example of adding two integer digit by digit. Addition of two n digit integer takes n number of steps to add digit by digit. Consequently, time taken by an algorithm to add this two n digit integer will be T(n) = c*n, where c is the time taken add two digits. Different computer take different amount of time to add two digits say c1 and c2. Computational time for the two computers to run n size data will be T1(n) = c1*n and T2(n) = c2*n respectively. Below Fig 1.0 shows the example of big-O notation for condition f(x) belongs to O(g(x)) as c>0 and x0 such that $f(x) \leq cg(x)$ whenever $x \geq x0$.

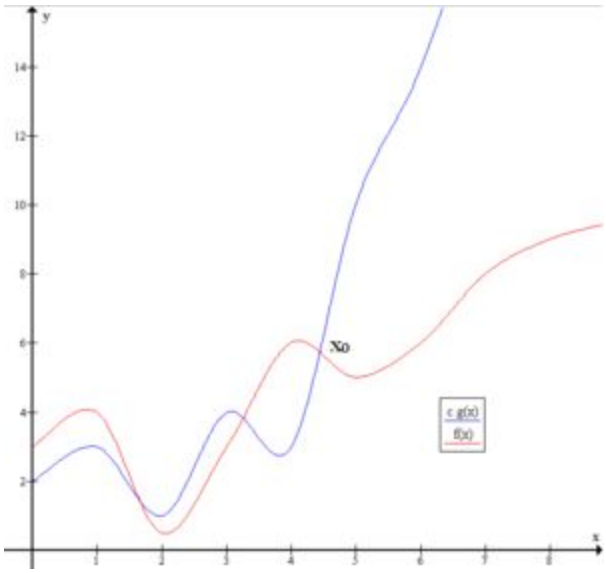

Fig 1.0

Reference:https://en.wikipedia.org/wiki/File:Big-O-notation.png

Finding runtime in seconds doesn't give information about the amount of data has been processed. It gives information about the time it takes to run a program. In order to find the amount of data being processed by a computer we represented the complexity by finding the numerical equation that depends on the input size n.

2. Let g : N → R+ be an arbitrary function. The set of functions f : N → R+ , which do not grow faster than the function g after a specific point n 0 , is denoted as O(g(n)). More specifically: O(g(n)) := f (n)| ∃ c # R and ∃ n 0 # N : 0 ≤ f (n) ≤ cg(n) ∀ n ≥ n 0.

1.Given $f(n) = 100 * n^2 \in O(n^2)$
f(n) = O(g(n))

$g(n) = n^2$

We know that $f(n) \leq c * g(n)$

$100 * n^2 \leq c * n^2$

Hence for every $c > 100$ and $n > n_0$ above equation satisfy the condition.

Therefore f(n) = O(g(n)), $f(n) = 100 * n^2 \in O(n^2)$

2. Given $f(n) = n^6 + 100 * n^5 \in O(n^6)$

f(n) = O(g(n))

$g(n) = n^6$

We know that $f(n) \leq c * g(n)$ for $n > n_0$

$0 \leq f(n) \leq c * g(n)$ divide by g(n)

Now $0 \leq f(n)/g(n) \leq c$

$0 \leq (n^6 + 100 * n^5)/n^6 \leq c$

Hence for every $c \geq (n^6 + 100 * n^6)/n^6$ and $n > n_0$ f(n) = c(g(n)) = O(g(n))

3. What is the running time of the following python-code in O-Notation? Assume, that ANY_CONST is an arbitrary constant in your program, which receives a 2d array arr as parameter.

```
1 sum = 0
2 for i in range(0, J):
3     for j in range(0, K):
4         if arr[i][j] <= ANY_CONST:
5             sum = sum + arr[i][j]
6 print(sum)
```

Worst case Scenario:

Line number 1 and 6 takes one step of iteration. Line 4 and 5 will run for K times and for loop in line 3 will run for J times.

So O(n) = (J*(2K) +2) = JK

4. For each function f (n) and time t in the following table, determine the largest size n of problem that can be solved in time t, assuming that the algorithm to solve the problem takes f (n) microseconds. Please briefly mention how you do the calculations for items: log(n), 2 n , and nlog(n).

	1 Second	1 Minute	1 Hour	1 Day	1 Month	1 Year	1 Century
$log(n)$	2^{10^6}	2^{60x10^6}	$2^{3.6\,x10^9}$	$2^{8.64x10^{10}}$	$2^{2.592x10^{12}}$	$2^{3.1536x10^{13}}$	$2^{3.1556x10^{15}}$
$\sqrt{n}$	10^{12}	$3.6x10^{15}$	$1.296x10^{19}$	$7.464x10^{21}$	$6.72x10^{24}$	$9.95x10^{26}$	$9.96x10^{30}$
n	10^6	$60x10^6$	$3.6x10^9$	$8.64x10^{10}$	$2.592x10^{12}$	$3.15x10^{13}$	$3.16x10^{15}$
$nlog(n)$	62500	2801417	133378058	2755147513	71870856404	797633893349	6.86×10^{13}
n^2	10^3	7745	60000	293938	1609968	5615692	56176151
n^3	100	391	1532	4420	13736	31593	146679
2^n	19	25	31	36	41	44	51
$n!$	9	11	12	13	15	16	17

$log(n)$: Assumption:

1. Considering log base 2
2. 30 days per month

$1\ Second\ =\ 10^6\ x\ 10^{-6}\ Seconds = 10^6\ Microsecond$

$log(n)\ =\ 10^6\ \Rightarrow\ n\ =\ 2^{10^6}$

$1\ Minutes\ =\ 60\ Seconds\ =\ 60x10^6\ Microsecond$

$log(n)\ =\ 60x10^6\ \Rightarrow n\ = 2^{60x10^6}$

$1\ Hour\ =\ 60\ Minutes\ =\ 60x60x10^6\ Microsecond$

$log(n)\ = 60x60x10^6\ \Rightarrow n\ = 2^{3.6\,x10^9}$

$1\ Day\ =\ 24\ Hour\ =\ 24x60x60x10^6\ Microsecond$

$log(n)\ = 24x60x60x10^6 \Rightarrow\ n\ = 2^{8.64x10^{10}}$

$1\ Month\ =\ 30\ Day\ =\ 30x24x60x60x10^6\ Microsecond$

$log(n)\ = 30x24x60x60x10^6 \Rightarrow\ n\ = 2^{2.592x10^{12}}$

$1\ Year\ =\ 12\ Month\ =\ 365x24x60x60x10^6\ Microsecond$

$log(n)\ = 365x24x60x60x10^6 \Rightarrow\ n\ = 2^{3.1536x10^{13}}$

$1\ Century\ =\ 100\ Year\ =\ (365x100 + 24)x24x60x60x10^6\ Microsecond$

$log(n)\ = 36524x24x60x60x10^6 \Rightarrow\ n\ = 2^{3.1556x10^{15}}$

2^n

$1\ Second\ =\ 10^6\ x\ 10^{-6}\ Seconds = 10^6\ Microsecond$

$2^n\ =\ 10^6\ \Rightarrow n * log(\ 2)\ =\ 6 * log(10)$

$n\ = 19$

$1\ Minute\ =\ 60\ Seconds\ =\ 60x10^6\ Microsecond$

$2^n = 60x10^6\ \Rightarrow n * log(\ 2)\ =\ log(60) + 6 * log(10)$

$n\ =\ 25$

$n * log(n)$

$1\ Second\ =\ 10^6\ x\ 10^{-6}\ Seconds = 10^6\ Microsecond$

$n * log(n)\ =\ 10^6$

Let $n\ =\ 2^t$ *and after substituting on above equation having base* 2 *log*

$t * 2^t\ =\ 10^6$

$16 * 2^{16} \approx 10^6$

$10^6 \approx 2^{20}$

$16 * 2^{16} = 2^{20}$

$log(n)\ =\ 16$

$n\ \approx 10^6/16\ \approx 62500$

Calculation for nlog(n) can also be made using **Newton–Raphson method.**

Reference

1. **https://math.stackexchange.com/questions/406707/elegant-way-to-solve-n-log-2n-le-106**

2. **https://en.wikipedia.org/wiki/Big_O_notation**

3. **https://mitpress.mit.edu/books/introduction-algorithms-third-edition**