2.1P Algorithm Complexity

For each of the above algorithms answer the following sub-questions for both algorithms.

a) First go through the above table and calculate the worst case for each algorithm.

i.

$$w = 14$$

ii.

$$w = 4N + 3$$

iii.

$$w = \frac{3N^2 + 13 + 6}{2}$$

iv.

$$w = 3logN + 4$$

v.

$$w = 8N + 6$$

vi.

$$w = \frac{4N^2}{2} + \frac{-N}{2} + \frac{1}{2}$$

Or

$$w = 2N^2 - \frac{N}{2} + \frac{1}{2}$$

b) Describe the equations found in (a) for the best, worst and average cases using Big-Θ notation.

i. Worst case: $\Theta(1)$

Best case: $\Theta(1)$

Average case: $\Theta(1)$

ii. Worst case: Θ(n)

Best case: Θ(n)

Average case: Θ(n)

iii. Worst case: Θ(n^2)

Best case: Θ(n)

Average case: Θ(n^2)

iv. Worst case: Θ(logn)

Best case: Θ(1)

Average case: Θ(logn)

v. Worst case: Θ(n)

Best case: Θ(n)

Average case: Θ(n)

vi. Worst case: Θ(n^2)

Best case: Θ(n^2)

Average case: Θ(n^2)

c) Describe each algorithm's overall performance using the tightest possible class in Big-O notation.

i. O(1)

ii. O(n)

iii. O(n^2)

iv. O(logn)

v. O(n)

vi. (n^2)

d) Describe each algorithm's overall performance using the tightest possible classing Big- Ω notation.

i. $\Omega(1)$

ii. $\Omega(n)$

iii. $\Omega(n)$

iv. $\Omega(1)$

v. $\Omega(n)$

vi. $\Omega(n^2)$

e) Describe each algorithm's overall performance using Big-Θ notation.

i. $\Theta(1)$

ii. $\Theta(n)$

iii. $\Theta()$ NA

iv. $\Theta()$ NA

v. $\Theta(n)$

vi. $\Theta(n^2)$

f) Selecting from one or more of the above which is the best way to succinctly describe the performance of each algorithm using asymptotic notation.

i. $\Theta(1)$

ii. $\Theta(n)$

iii. $O(n^2)$

iv. O(logn)

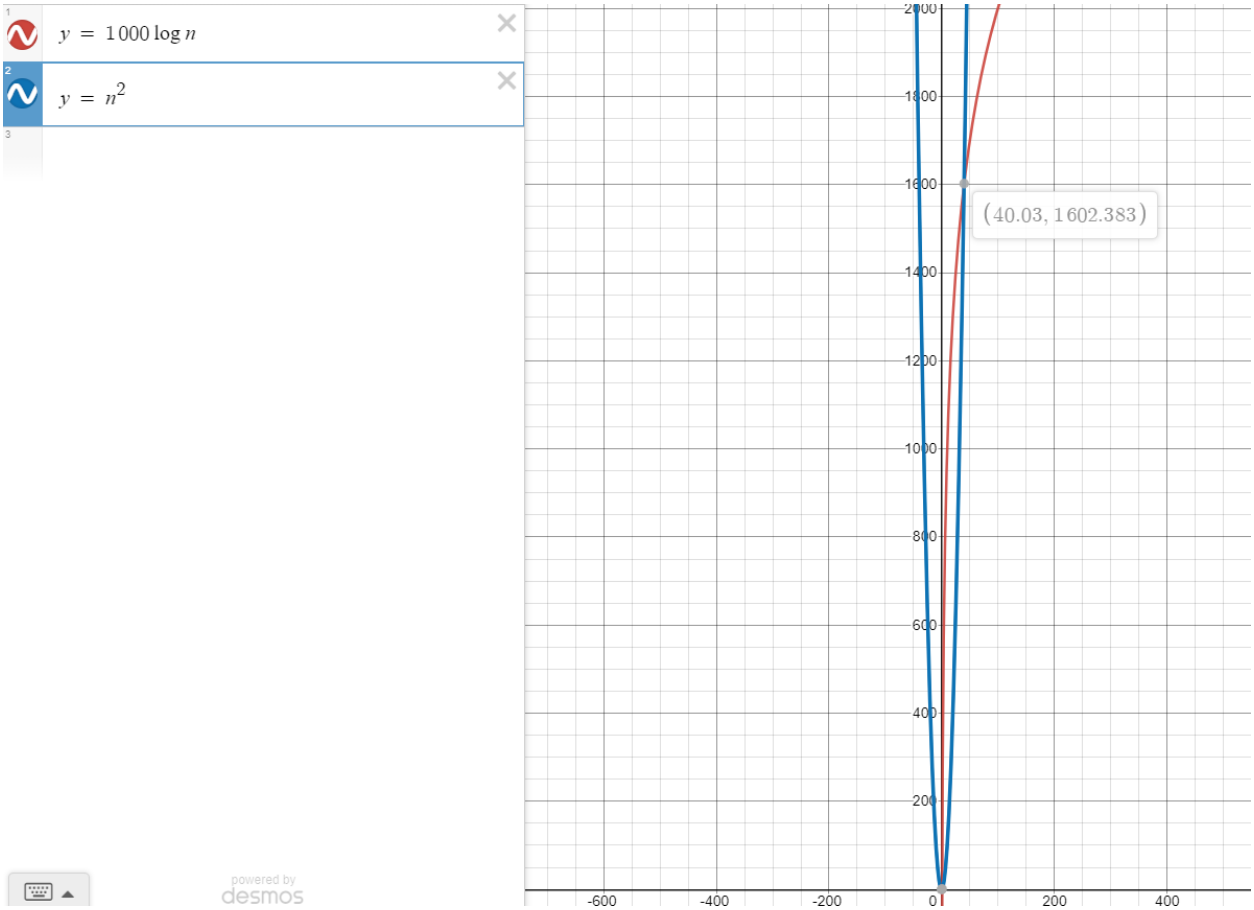
v. $\Theta(n)$

vi. $\Theta(n^2)$

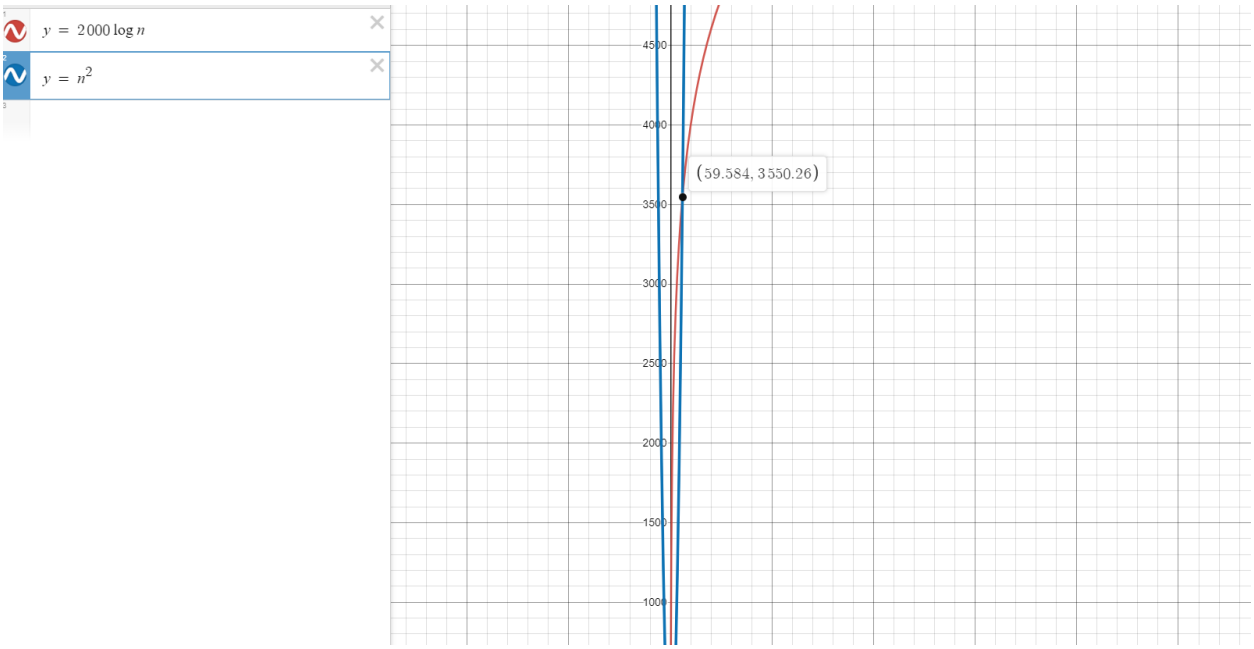
2. Arguably, the most commonly used asymptotic notation used is Big-O. Discuss why this is so commonly the case and why more people do not use Big-Θ.

Big-O is more commonly used to describe an upper bound and Big-Θ is used to describe the tight bound. The reason that Big-O is the most commonly used is because to get an upper bound only requires the upper bound however to get a tight bound you need both the upper bound and the lower bound which is unnecessary since developers only really care for how badly an algorithm performs, hence why the upper bound is mostly used due to it being only one factor as opposed to needing the upper bound and lower bound before getting the tightly bound which in some cases is not possible to achieve. When it comes to the running time of an algorithm it is usually important to optimize for the worst case thus, knowing how it will perform in the worst case, using Big-O and upper bounding gives us the worst that the algorithm will perform which is extremely useful as we only really care for the worst case of an algorithm that in which it will perform because we don't expect our algorithm to run in the best or average case because that is quite unlikely. Big-O gives us a view on the worst the algorithm will behave and how quickly it will run as the data set grows. When only focusing on the worse case for an algorithm this generates the ability to produce better quality algorithms with regards to the worst case resulting in a better overall algorithm.

3. Is it true that $\Theta(n^2)$ algorithm always takes longer to run than an $\Theta(\log n)$ algorithm? Explain your answer.



No this is not true, as can be seen in the plotted graphs above although x^2 has a very steep rise it can be seen that the 1000logn algorithm has a slower and more inefficient runtime initially, it is until at n^0 which can be seen at operation 40 where the 1000logn algorithm crosses over into becoming a quicker algorithm since the nature of a logn algorithm no matter how big the constant at the beginning will always be a faster runtime then the n^2 algorithm as the dataset progresses past n^0. This proves that it would be incorrect to assume that an n^2 algorithm will always take longer then a logn algorithm due to the constant that could be at the beginning of the quicker known algorithm, this leads me onto the point that for larger datasets however the quicker algorithm will eventually take over no matter the constant at the beginning due to the nature of the algorithm. Which can also be seen when the logn algorithm starts with the constant 2000 to represent 2000logn in the figure below:



This shows that n^2 will not always initially take longer to run but as the dataset increases the logn algorithm will always naturally become faster at n^0.

4. Answer whether the following statements are right or wrong and explain your answers.

$n^2 + 6^{13}n = O(n^2)$
$nlog\, n = O(n)$
$n^2 + n + 10^6 = \Theta(n^3)$
$n\, log\, n + 51n^2 = \Omega(n)$

1. $n^2 + 6^{13}n = O(n^2)$
   This is correct since the algorithm equation is based on the quadratic runtime so using the upper bound of n^2 means that the algorithm will not perform worse then this class of algorithms.

2. $n\, log\, n = O(n)$

This is incorrect since the algorithm equation is based on the n log n runtime so using the upper bound of n means that the algorithm is wrong since n is faster then n log n runtime complexity, it would have a lower bound of n however.

3. $n^2 + n + 10^6 = \Theta(n^3)$
   Incorrect. Big theta is a tight bounding and cannot be used in the cubic runtime complexity also since it is of a different runtime complexity class, n^2 seems to be a better fit for the theta tight bounding run complexity.

4. $n \log n + 51n^2 = \Omega(n)$

This is correct, this is because nlogn and the n^2 runtime complexity is slower and takes more operations than the linear runtime complexity n, due to this it is correct to say the n class is a lower bound and best case since it is faster than both of those run time complexities.