Written Assignment-1 :

① Run-time of Algorithm A:

$$f(n) = 3 \cdot 2^n - n$$

Run-time of Algorithm B:

$$g(n) = n^2 + n + 100$$

when Algorithm B is faster than A,

$$g(n) < f(n)$$

$$\Rightarrow n^2 + n + 100 < 3 \cdot 2^n - n$$

$$\Rightarrow n^2 + 2n + 100 < 3 \cdot 2^n$$

$$\Rightarrow (n+1)^2 + 99 < 3 \cdot 2^n$$

$$\Rightarrow \frac{(n+1)^2}{3} + 33 < 2^n \text{ — Eq. ①}$$

~~Smallest value of~~

Smallest value of 'n' for which equation ① is true ∴

~~is true~~ :

Approach ① [Brute-Force].

(i) when, (n=1), EQ. ① becomes

$$\Rightarrow \frac{(1+1)^2}{3} + 33 < 2^1$$

$$\Rightarrow 4/3 + 33 < 2$$

(ii) Similarly, when n=2, EQ. ① becomes,

$$\Rightarrow \frac{(2+1)^2}{3} + 33 < 2^2$$

$$\Rightarrow 3 + 33 < 4$$

$$\Rightarrow 36 < 4$$

(iii) Similarly, when n=3, EQ ① becomes,

$$\Rightarrow \frac{(3+1)^2}{3} + 33 < 2^3$$

$$\Rightarrow \frac{16}{3} + 33 < 8$$

Now, intuitively, the value of 'n' should be such that $2^n > 33$, since in eq. ①

$$\Rightarrow \frac{(n+1)^2}{3} + 33 < 2^n$$

The first term in the LHS of the above equation, i.e, $\frac{(n+1)^2}{3}$ is always positive.

So, let's try with $n=6$.

So, when $n=6$, eq. ① becomes,

$$\Rightarrow \frac{(6+1)^2}{3} + 33 < 2^6.$$

$$\Rightarrow \frac{49}{3} + 33 < 64$$

$$\Rightarrow 16.33 + 33 < 64$$

$$\Rightarrow \underline{49.33 < 64}$$ → Clearly when $n=6$, eq. ① holds true.

So, the smallest value of input size 'n', for which the Algorithm 'B' performs faster than Algorithm 'A' is $\boxed{n=6}$.

Graphical Approach :

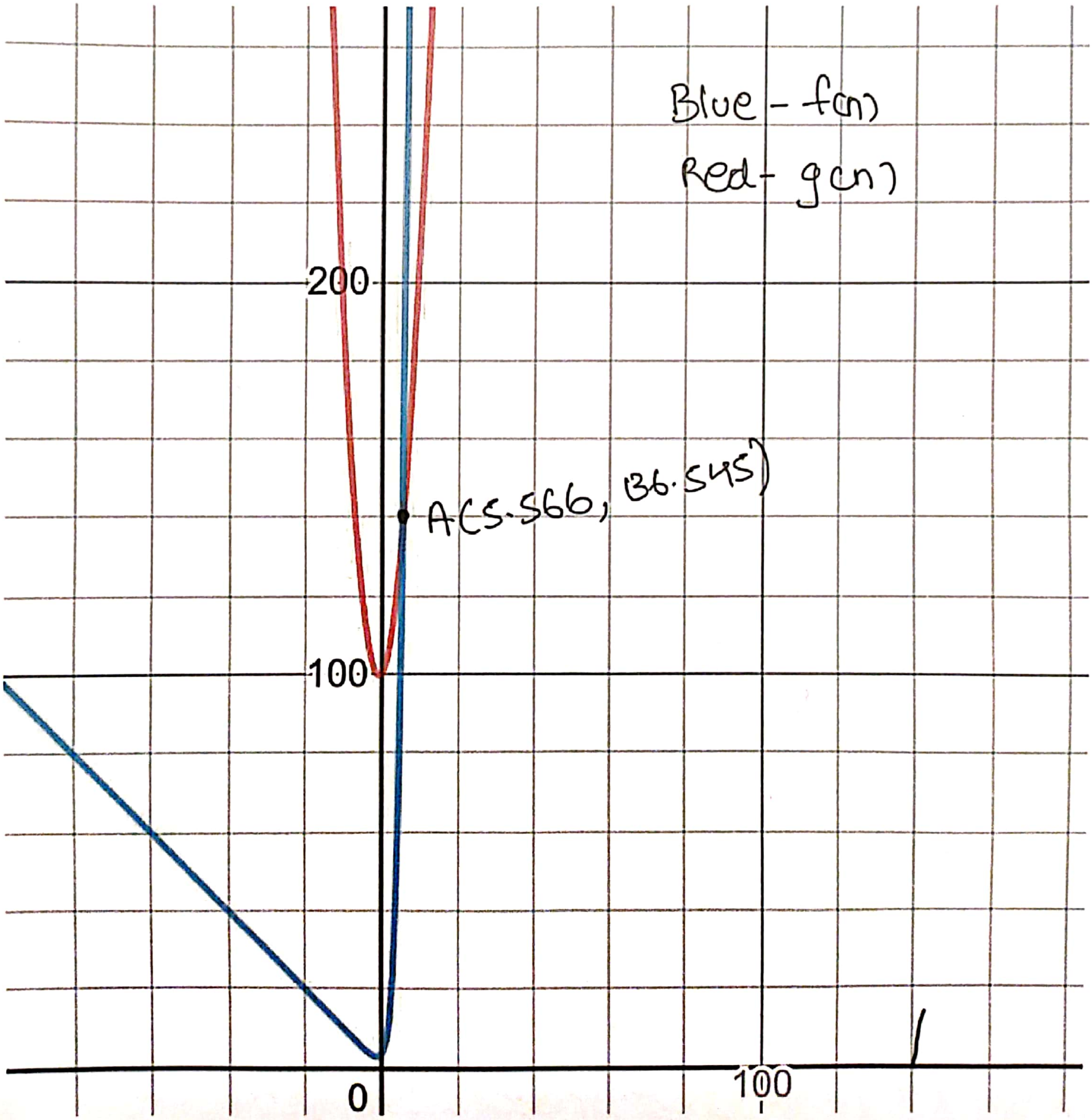

Figure Source: I used an online graph-plotting application called "desmos" to generate the above graph.

Inference from plot:

From the above graph, we can clearly see that the curves intersect at A(5.566, 136.545) and for any 'n' greater than or equals to 5.566, $\boxed{g(n) < f(n)}$, i.e, Algorithm 'B' is faster than algorithm 'A'. Since the input size is always an integer value, 'n' has to be an integer, so $\boxed{n=6}$ is the smallest integer value for which algorithm 'B' is faster than algorithm 'A'.

② O(Big-Oh) notation:

The O-notation asymptotically bounds a function from above. For a given function g(n), we denote $O(g(n))$ as the set of functions:

$O(g(n)) = \{ f(n)$: there exists positive constants c and n_0 such that $0 \leq f(n) \leq c\,g(n)$ for all $n \geq n_0 \}$

So, $f(n) = O(g(n))$, when

$$0 \leq f(n) \leq c\,g(n)$$

when, $c > 0$
$n \geq n_0$
$n_0 > 0$

So, using the above formal definition of Big-Oh, we now have to prove the transitive property of big-oh notation.

Given,

$$f(n) = O(g(n))$$
$$g(n) = O(h(n))$$

we have to prove → $f(n) = O(h(n))$

Using the formal definition, we can write $f(n)$ as -

$$0 \le f(n) \le c_1 g(n) \ \forall \ n \geq n_0,$$

where, $c_1 > 0$
$n_0 > 0$

similarly, $g(n)$ can be written as -

$$0 \le g(n) \le c_2 h(n) \ \forall \ n > n_0',$$

where, $c_2 > 0$
$n_0' > 0$

Now, if we substitute the bounds of $g(n)$ in the bounds of $f(n)$, we get -

$$0 \le f(n) \le c_1 c_2 h(n)$$

$$\Rightarrow 0 \le f(n) \le c_3 h(n), \ \forall \ n > n_0''$$

where, $c_3 > 0$
$n_0'' > 0$

So, we can write $f(n) = O(h(n))$, which means that $h(n)$ asymptotically bounds $f(n)$.

Example: let's say, $f(n) = n$

$g(n) = n^2$

$h(n) = n^3$.

we know that,

$0 \leq g(n) \leq h(n)$

i.e, $0 \leq n^2 \leq cn^3$. for $n \geq 1$, $c > 0$

So, n^3 can be an upper bound for n^2.

Similarly,

$0 \leq n \leq cn^3$, for $n \geq 1$, $c > 0$.

n^3 can also be an upper bound for n.

So, we can write, $\boxed{f(n) = O(h(n))}$

(3) (i) Is $\log(n^3) = O(\log(n))$?

Solⁿ: $f(n) = \log(n^3) = 3\log(n)$

According to the formal definition of Big-oh,

$f(n) = O(g(n))$ if,

$$0 \le f(n) \le c\,g(n) \quad \text{--- ①} \quad \forall n \ge n_0,$$

where, $c > 0, n_0 > 0$

if $g(n) = \log(n)$

EQ ① becomes, $0 \le f(n) \le c\log(n)$.

Now, if we substitute $f(n) = 3\log(n)$, equation ① becomes,

$$0 \le 3\log(n) \le c\log(n). \quad \forall n \ge n_0,$$

where $c > 0$, $n_0 > 0$.

For all values of 'c' greater than or equals to '3', the function $f(n) = 3\log(n)$ can be asymptotically bounded by —

$g(n) = \log(n)$.

so, $f(n) = 3\log(n) = \log(n^3) = O(\log(n))$.

(iii) Is $\log^3(n) = O(\log(n))$?

According to the formal definition of Big-Oh notation,

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

if, $0 \le f(n) \le c\,g(n) \ \forall n \geq n_0$

where $n_0 > 0$

$c > 0$

Substituting $f(n) = \log^3(n)$ and $g(n) = \log(n)$ in the formal definition, we get.

$\Rightarrow 0 \le \log^3(n) \le c\log(n)$,

But the above inequality is wrong, as we cannot have a constant (c) $\forall n \geq n_0$, such that $\log^3(n)$ can be bounded by $\log(n)$.

(4) From the formal definition of Big-Oh,

we know that,

$$f(n) = O(g(n)), \text{ if } 0 \leq f(n) \leq c\,g(n)$$

$\forall\ n \geq n_0$, where

$c > 0$

$n_0 > 0$

Now, to prove that $\min(f(n), g(n))^2 = O(f(n) * g(n))$; let's first bound the minimum of functions $(f(n), g(n))$ by the maximum of the same functions.

Let's assume that $f(n) \leq g(n)$. Then, we can asymptotically bound $f(n)$ by $g(n)$, using the formal definition of Big-Oh, i.e,

$$0 \leq f(n) \leq c\,g(n), \ \forall\ n \geq n_0, \text{ where } c > 0,\ n_0 > 0$$

Now, if we multiply $f(n)$ in the above inequality we get,

$$0 \leq f^2(n) \leq c\, f(n)\, g(n), \forall n \geq n_0',$$ where $c > 0$, $n_0' > 0$

Note- The parity of the above inequation doesn't change after multiplication because $f(n)$ is positive valued.

Similarly, if $g(n)$ is minimum (or) equals to $f(n)$ we get,

$$0 \leq g^2(n) \leq c\, f(n)\, g(n), \forall n \geq n_0, \; c > 0, \; n_0 > 0$$

So, clearly, $\min(f(n), g(n))^2$ can be asymptotically upper bounded by $f(n) \cdot g(n)$.

So, $\boxed{\min(f(n), g(n))^2 = O(f(n) \cdot g(n))}$

5) Code- snippet :

Line

```
(i)   int count = 0
(ii)  for( int i=0; i<n; i++)
(iii)     for( int j=1; j<n; j++)
(iv)          for( int k=j; k<i; k*=2)
(v)               count++;
(vi)  return count;
```

Time- complexity Analysis → Calculating the exact no. of operations is ~~difficult~~ difficult, so let's use a rough approximation.

Line (i) → Since it's an assignment statement, we can assume it to take constant time.

SO, $O(1)$

Line (ii) - There's a looping statement on this line, where we do assigment, comparison, and increment. Assuming these operations to have constant time, we can say that this line is executed (n+1) times.

Line (iii) - There's another looping statement on this line, with the same set of operations as above. Assuming constant time of these operations, we can say that this line is excuted (n) times for every iteration of line (ii). Or (n) (n+1) times in total.

Line (iv) → There's yet another looping statement on this line with the same set of operations. Assuming constant time for these operations, we can say that this line is executed at most $\log_2(n)$ times for each value of $(i), (j)$.

Line (v): This is the innermost statement in the given code snippet and it's representative of total number of operations in this algorithm. This line is executed at most $n^2 \log(n)$.

Line (vi) : This is just a return statement, so we can assume constant execution time, $O(1)$.

The total number of operations of the algorithm is at most - $O((n+1)(n)(\log(n)))$

$$= O(n^2 \log(n) + n \log(n))$$

$$= O(n^2 \log(n))$$

→ For sufficiently, Large (n).

But, Big-Oh only gives us the upper bound of running time complexity.

To calulate the average time complexity, $\Theta(n)$, we can use the property of asymptotic notations, that states:

$f(n) = \Theta(g(n))$, if $f(n) = O(g(n))$

$\&$

$f(n) = \Omega(g(n))$.

Now, to check if, $\Theta(n^2\log(n))$ is the time complexity of this algorithm, we need to check if $f(n) = \Omega(n^2 \log(n))$, where $f(n)$ = running time of the algorithm.

To check if $f(n) = \Omega(n^2 \log(n))$, let's check if for $i \geq n/2$, line (4) executes for $\Omega(n^2 \log(n))$ times.

For, $i \geq n/2$,

(i) line (2) runs for $n/2+1$ times.

(ii) line (3) runs for 'n' times for each 'i', and $n(n/2+1)$ times in total.

(iii) lines (4) & (5) run for a minimum of ~~(...)~~ $\log_2(n/2)$ times, for each 'i' and 'j'.

So, the total number of operations, is at least,

$$\Omega\left[(n/2+1)\, n\, (\log_2(n/2))\right]$$

$$= \Omega\left((n^2/2+n)(\log_2(n/2))\right)$$

$$= \Omega\left(\frac{n^2}{2}\log_2(n/2) + n\log_2(n/2)\right)$$

For large 'n'

$$\simeq \Omega\left(\frac{n^2}{2}\log_2(n/2)\right)$$

$$\simeq \Omega\left(\frac{n^2}{2}\left[\log_2(n) - \log_2(2)\right]\right)$$

$$\simeq \Omega\left(\frac{n^2}{2}\log_2(n)\right)$$

$$\simeq \Omega\left(n^2\log_2(n)\right).$$

So, $f(n) = \Theta(n^2 \log_2(n))$ is the running time of the algorithm.

⑥ Algorithm :

```
Line.
(i)    int count = 0;
(ii)   for( int i=1; i<n; i*=2)
(iii)      for( int j=0; j<n; j+=i )
(iv)           count ++;
(v)    return count;
```

Let's calculate the total number of operations of this algorithm,

Line (i) → Assignment statement and takes constant time. Since, it doesn't scale with the size of the –

input, we can ignore this

for calculating average time complexity.

Line (ii) → Looping statement, and it runs for $K = \log_2(n)$ times.

Line (iii) → looping statement, and the number of time it executes depends on the value of 'i'.

K-times

For $i=1$, it executes n times

for $i=2$, it executes $n/2$ times

For $i=3$, it executes $n/2^2 = n/4$ times

⋮

for $i=n$, it executes $= n/2^K$ times.

So, total number of operations can be written as,

$$= n + n/2 + n/2^2 + n/2^3 + \cdots n/2^K.$$

$$= n\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^K}\right).$$

$$= n\left[\frac{(1)\left(1 - \left(\frac{1}{2}\right)^{K+1}\right)}{1 - 1/2}\right]$$

$$= 2n\left[1 - \left(\frac{1}{2}\right)^{K+1}\right].$$

we know, $K = \log_2(n)$.

so,

$$\text{Run-time } (f(n)) = 2n\left[1 - \frac{1}{(2^K) \times 2}\right].$$

$$= 2n \left[1 - \frac{1}{2^{\log_2(n)} \times 2} \right].$$

$$= 2n \left[1 - \frac{1}{2n} \right].$$

$$= (2n - 1)$$

Therefore, the running time complexity of this algorithm can be represented as –

$$\boxed{f(n) = 2n - 1 = \Theta(n)}$$