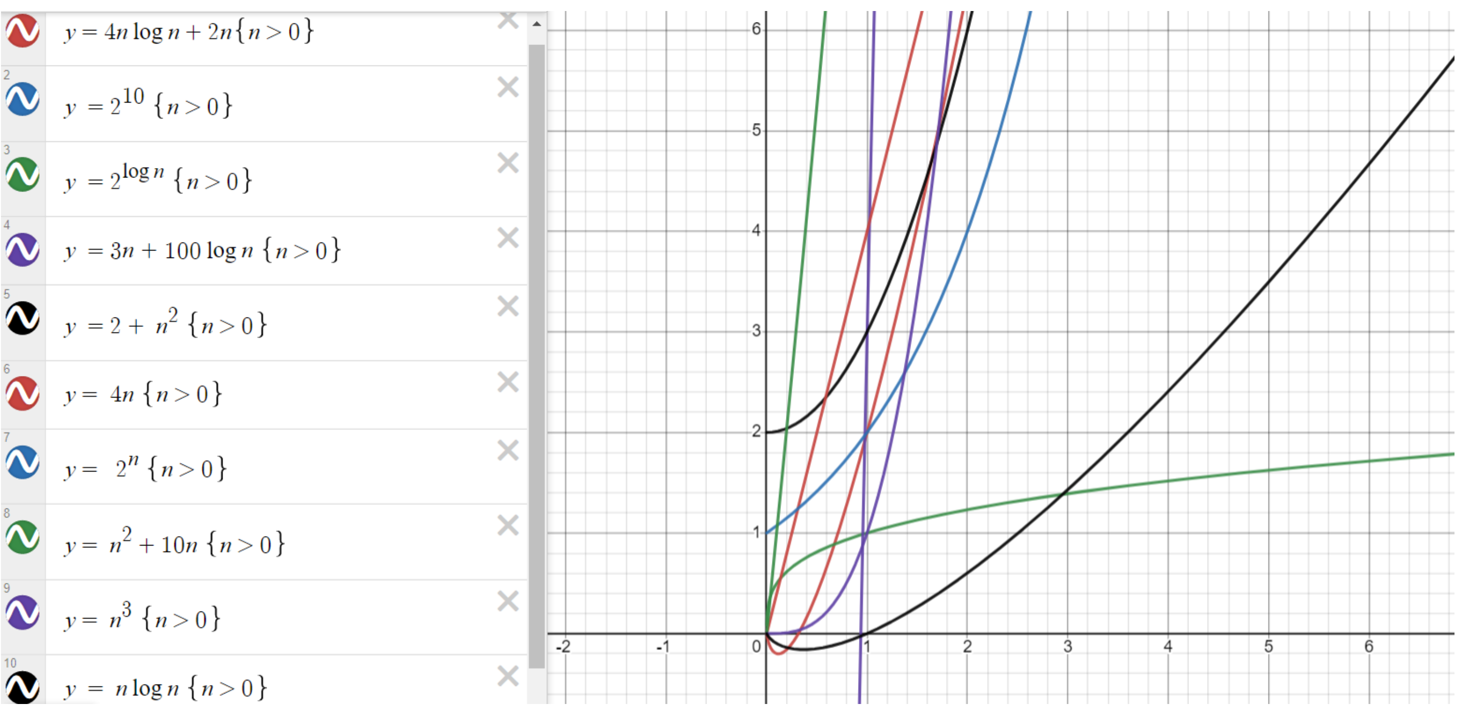
MECHTRON 2MD3 | Assignment 3

Question 1

> Order the following functions by asymptotic growth rate:
> (1) $4nlog(n) + 2n$
> (2) 2^{10}
> (3) $2^{log(n)}$
> (4) $3n + 100log(n)$
> (5) $2 + n^2$
> (6) $4n$
> (7) 2^n
> (8) $n^2 + 10n$
> (9) n^3
> (10) $nlog(n)$

If we plot all of these $O(n)$ functions, on a Big O complexity chart. We would get the following diagram:



Using this the functions with the slowest growth rate at the top would be:
(2) 2^{10}
(3) $2^{log(n)}$

(4) $3n + 100log(n)$?
(6) $4n$
(10) $nlog(n)$
(1) $4nlog(n) + 2n$
(5) $2 + n^2$
(8) $n^2 + 10n$
(9) n^3
(7) 2^n

Question 2

> Show that if d(n) is O(f(n)), then ad(n) is O(f(n)), for any constant a > 0. (Hint: use the formal definition of Big-O)

In big O notation, we often ignore any constants in front of function. For example, if we had a function that had a big-O time of $log(n)$ and another function of time of $a * log(n)$. For small values of n, the value of a would have a difference on the running time of the algorithm. But in **Big**-O notation revolves around **big** numbers, in this case as $\lim_{n \to \infty}$ we would have two functions that are ∞ and $8 * \infty$ which are both just the same value.

Question 3

> Give a big-Oh characterization, in terms of n, of the running time of Algorithm Ex1 on page 3.

```
Algorithm Ex1(A):
    Input: An array A storing n ≥ 1 integers.
    Output: The sum of the elements in A.
    s ← A[0]
    for i ← 1 to n - 1 do
        s ← s + A[i]
    return s
```

This function would be in $O(n)$ time due to there being a for loop that iterates through each element in the `A[n]`

Question 4

> Give a big-Oh characterization, in terms of n, of the running time of Algorithm Ex2 on page 3.

```
Algorithm Ex2(A):
    Input: An array A storing n ≥ 1 integers.
    Output: The sum of the elements at even cells in A.
    s ← A[0]
    for i ← 2 to n - 1 by increments of 2 do
        s ← s + A[i]
    return s
```

This function would be also in $O(n)$ time. This is because for a list of n items, the algorithm would go through n/2 items. For Big-O notation as we have $\lim_{n \to \infty}$ the constant $O(n/2)$ would become $O(n)$.

Question 5 (change!)

> Give a big-Oh characterization, in terms of n, of the running time of Algorithm Ex3 on page 3.

```
Algorithm Ex3(A):
    Input: An array A storing n ≥ 1 integers.
    Output: The sum of the prefix sums in A.
    s ← 0
    for i ← 0 to n - 1 do
        s ← s + A[0]
        for j ← 1 to i do
            s ← s + A[j]
    return s
```

This algorithm take in an array of `n` items and for each item it sums up the previous items before it. To find the time complexity, you would consider:

1. That the algorithm considers `n` items for the first for loop
2. For the second for loop it considers a total of `i` items up to `n` times
3. For the maximum possible case, the first for loop would have `n` items and the second for loop would also have a max of `n` items.

Since the second for loop scales from `1` item to `n` items, the total sum of `s` would be the sum of the two loops giving up $O(n * n)$ time which would be a final time of $O(n^2)$ complexity.

Question 6

> Give a big-Oh characterization, in terms of n, of the running time of Algorithm Ex4 on page 3.

```
Algorithm Ex4(A):
    Input: An array A storing n ≥ 1 integers.
    Output: The sum of the prefix sums in A.
    s ← A[0]
    t ← s
    for i ← 1 to n - 1 do
        s ← s + A[i]
        t ← t + s
    return t
```

Since this algorithm goes through `n` variables, the function would be at $O(n)$ time based on n items in the array.

Question 7

> Give a big-Oh characterization, in terms of n, of the running time of Algorithm Ex5 on page 3.

```
Algorithm Ex4(A):
    Input: Arrays A and B each storing n ≥ 1 integers.
    Output: The number of elements in B equal to the sum of prefix sums in A.
    c ← 0
    for i ← 0 to n - 1 do
        s ← 0
        for j ← 0 to n - 1 do
            s ← s + A[0]
            for k ← 1 to j do
                s ← s + A[k]
                if B[i] = s then
                    c ← c + 1
    return c
```

Since we have two for loops that go through `n` items each we would have a time complexity of $O(n^2)$. But since the second loop goes only up to a worst case of `n` items we would have a time of $O(n^2 * n)$, leaving us with a final time of $O(n^3)$.

Question 8

> Given an n-element array X, Algorithm D calls Algorithm E on each element X[i]. Algorithm E runs in O(i) time when it is called on element X[i]. What is the worst-case running time of Algorithm D?

Here we have:

- D calls E on each value of n, running in O(n)
- E calls on each value running in O(n) time for each element X[n]

The worst-case running time would be that algorithm D calls each element from X[0], X[1], ... , X[n] and algorithm E runs on each element up to X[n] giving a total runtime of $O(n^2)$

Question 9

> An array A contains n-1 unique integers in the range [0, n-1], that is, there is one number from this range that is not in A. Design an O(n)-time algorithm for finding that number. You are only allowed to use O(1) additional space besides the array A itself.

```
outOfBounds(int* A, int n){  // n takes up 1 space
    int sum = 0; //takes up 1 space, 2 total
    int missingSum = 0; //takes up 1 space, 3 total
    for(int i =0;i<n;i++){
        sum += i;  //this gives the value of the sum of the range [0, n-1]. This also has a time
    }
    for(int i =0;i<n;i++){
        missingSum += A[i];  //this gives the value of the sum of array. This also gives a total
    }
    return sum - missingSum;
}
```

This would result in a function with:

- O(3) = O(1), space complexity
- O(2n) = O(n), time complexity

Question 10

> Suppose you have a deque D containing the numbers (1,2,3,4,5,6,7,8), in this order. Suppose further that you have an initially empty queue Q. Give a pseudo-code description of a function that

> uses only D and Q (and no other variables or objects) and results in D storing the elements (1,2,3,5,4,6,7,8), in this order

```
while(D is not empty){
    if(D.front is 4){
      Q.emplace_back(5); //swaps 4 with 5
      D.pop_front();
      continue;
    } else if(D.front is 5){
      Q.emplace_back(4); //swaps 5 with 4
      D.pop_front();
      continue;
    }
    Q.emplace_back(D.front()); //inserts the first element of D to the last element of Q
    D.pop_front(); //deletes the first element
  }
  Q.swap(D); //swap D and Q
```

Question 11

> Suppose Alice has picked **three distinct integers** and placed them into a stack S in random order. Write a short, straight-line piece of pseudo-code **(with no loops or recursion)** that uses only **one comparison and only one variable x**, yet guarantees with probability **2/3** that at the end of this code the **variable x will store the largest** of Alice's three integers. Argue why your method is correct

```
x = S.pop(); if(x<S.top()) {x=S.pop()};
```

First we start of with a stack of [`A` , `B` , `C`],

We then set x = `A` and have a resulting stack of [`B` , `C`]

We then compare `A` to `B` , leaving us with two options:

1. $A > B$
2. $A < B$

If `A` is greater than `B` , then we have either `A` or `C` being the greatest.

- the pseudocode above sets `x` to be `A`

If B is greater than `A` , then we have either `B` or `C` being the greatest.

- the pseudocode above sets `x` to be `B`

Thus if the largest number is in the first or second slot of the stack, `x` will store the largest number between the two else if it is at the end it will not. This leaves us with a 2/3 chance of getting the largest number due to random chance since the algorithm is dependent on the random number generator landing the largest number in either `A` or `B`.