2/7 Asymptotic Notation

(formal way to model growth of functions)
Three Notations:

1. Big-Oh: O()
2. Big-Omega: Ω()
3. Theta: thetasymbol()

Big-Oh

Represents the upper bound of an algorithm's running time

- Formal Definition: The function f(n) = O(g(n)) if
    - revE (there exists) constants c and n0 such that..
    - f(n) <= c * g(n)
    - For all n>= n0
    - Then it will be true
    - Usually we think n>=1 since we don't worry abt empty/neg sized arrays/data (works if nothing if specified)

Ex: math example

- f(n) = 10n+100
- A c*g(n) combo that is >= f(n) for n >= 1
    - 10n + 100 <= 110n for all n>=1
    - 10n+100 = O(n) [c = 110, g(n) = n]
    - Trick to quickly find function that satisfies:
        - Take summation of highest degree terms by looking at the f(n)
        - Eg: 4n + 7 <= 4n + 7n → 4n + 7 <= 11n for all n >=1
            - 4n+7 = O(n) [c=11 , g(n) = n]
- ** ~~but we cant~~ we can say that for f(n) = 10n+100
    - 10n+100 <= 110n^2 , for all n>=1 i.e f(n) = O(n^2)
    - Works for higher order like O(n^3)
    - Yes it can
- But we try to use a g(n) that as close as possible , b/c its not helpful to say i am shorter than an giraffe
    - We need a frame of reference that is close

Big-Omega

*represents the lower bound of na algorithms running time

- Formal Def: the function f(n) =Ω g(n) if
    - There exists constants c and n0
    - c*g(n) <= f(n)
    - For all n >- n0

Ex: math example
- f(n) = 10n + 100
- Can you think of a c*g(n) combo that is less than/ equal to f(n) for n >=1
    - Possible solution: n <= 10n+100, for all n >=1
    - 10n + 100 = Ω (n)
    - Similar to O(n) the following are also true
        - 10n + 100 = Ω (rt n)
        - 10n + 100 = Ω (log n)
        - 10n +100 =Ω(1)
        - But we want to get as close of a bound for it to be helpful, there is no point saying i am definitely bigger than a mouse

Theta

*represents BOTH upper and lower bounds of an algorithms running time
Formal Definition:
- The function f(n) = θg(n)) if..
    - C1 * g(n) <= f(n) <= c2 * g(n)
    - For all n>= n0

Ex: 10n+ 100
Alr know lower and upper bounds of this fn
Possible sol.
- n<= 10n +100 <= 110n, for all n >= 1
- C1 = 1, c2 = 110
- Then 10n + 100 = θ(n)
    - We can't say that 10n+100 = θ(n^2) or .. θ(logn) (for this def)

Visual Summary

Check slides
Ok to cross the cg(n) , if its before the n0, for O(n), shows the upper bound
- Opposite of omega
- For θ, we're trying to trap f(n) between the two boundaries , if it isnt true before n0 it is fine, but after n0 everything is true
- N0 = 1, since its p much useless for us to do an empty array, which is special case

Practice:

f(n) = n^2 + 2n + 1 , n0 = 1

Big-o : factor out the largest term?

- : 4n^2, c = 4, g(n) = n^2, → **O(n^2)**

big-omega: ~~1,~~ → we want the closest one which is → **omega(n^2)**

- **Take the biggest term on its own.,**
- n^2 <= n^2 + 2n +1, we want the closest one so, o(n) still works but we want the closest so omega(n^2)

theta : ~~1~~ <= n^2 + 2n + 1 <= 4n^2, →

- Since f(n) = O(n^2) and f(n) = omega(n^2), we can conclude that f(n) = theta(n^2)

Ex2:

f(n) = n^2 logn + 2n ; (n0 = 1)

big -O: just take the highest order term

- O(n): n^2 logn + 2n <= n^2 log n + 2n^2 logn
- **O(n^2 logn)**

big-Omega:

- omega(n^2 logn)

Theta:

- theta(n): since both big=0 and big-omega are n^2logn , we can conclude that
- Theta is n^2 logn

Ex3: practice finding out O(n), Omega(n), Theta

f(n) = n! , n0 = 1

O: n!= n*(n-1)*(n-2) → <= n*n*n*n*n → f(n) = O(n^2)

- O(n^n)

Omega: 1*1*1*1*1 <= n*(n-1)*(n-2)... → f(n) = Omega(1)

- Omega(1) , ** just because **its a constant** it is omega(1), not b/c it is 1

Theta:

- Since **upper(big o) and lower(big omega) bounds** are different we **cannot define theta** for this f(n)
- We can only say f(n) = O(n^n) and f(n) = Omega(1)

Desmos might help on hw*

*Asymptotic Notation in Code

- We can say that the order of growth for the linear function(defined above) is O(n)

Best, worst, and average cases

- Linear searching (goes thru every value)
    - **Best case** if key is in first value (1 comparison)
        - In any size, consider it all , as array grows, comparisons don't
        - In asymptotic notation , the best case time complexity is O(1), Omega(1), Theta(1) , **can use all three do not confuse best and worse case with upper and lower bound**
    - **Worst Case** if key is last or not in the array
        - As size of array grow, the # of comparisons grow
        - In asymptotic notation, worst case time complexity O(n)... works for ohm and they also since is the same
    - **Average case** if key is somewhere in the middle
        - Comparisons do increase as size of array increase
            - Avg comparisons (1+2+3…n)/2 = n(n+1)/n
        - Is also O(n).... though we probably would expect it to be n/2, but in big o , you drop constants

So in Samson's example, even though the function that satisfies the constraints is $4n^2$, we ignore the '4' and simply write $O(n^2)$.

When you see $O(n^2)$, it doesn't mean g(n) is just n^2 - it's *some* quadratic polynomial. It's equally possible that g(n) was something like $n^2 + 10n + 10000000$.

Remember, we ignore lower-order terms because they become insignificant as the input size grows towards infinity.

As n grows larger, the largest (highest degree) term, n^2, increasingly influences the behavior of the function. The lower order terms (10n and 10000000 in this case) become relatively less significant compared to the highest order term (n^2). This means that, for sufficiently large values of n, the behavior of the function is primarily determined by n^2.

Ignoring lower-order terms in asymptotic notation allows us to focus on the most significant factors that impact the growth rate of a function. It simplifies the analysis and makes it easier to compare the performance of algorithms or functions without getting bogged down in unnecessary details.

-