Algorithm Analysis

Suppose we want to know what effects the input size has on the runtime of functions. We want to measure whether or not this function will be efficient given a large dataset.

Efficiency: A measure of the use of computing resources by code.

- Can be relative to speed (runtime), memory(space), etc.
- Most commonly refers to run time

Runtime: is the most common way to judge how efficient a data structure or algorithm is. Some algorithm may be fast on small datasets but slows done as the dataset gets larger.

- For example, insertion sort is quicker than merge sort for small input but, as the input size increases insertion becomes slower and slower.
- Input size is often a measure by the number of items in its dataset, often refer to as *n*
- **Growth rate**: change in runtime as input size "*n*" changes
- Calculating the exact runtime of an algorithm is difficult because its dependent on many factors such as the machine you run it on, the programming language, and implementation. Instead of calculating the exact runtime, we estimate the runtime using big O notation

Assumptions

- Any single statement takes the same amount of time to run
- Basic operations such as comparisons, assignment, and arithmetic takes constant time
- A function call's runtime is measured by the total of the statements inside the method's body
- A loop that repeats n times, with constant number of operations per loop, takes O(n).
- A loop that repeatedly halves the input takes O(log n)

Say an algorithm runs $3n^3 + 25n^2 + 8n + 17$ statements

- Consider the runtime when *n* is extremely large
- We ignore constants like 25 because they can be tiny compare to *n*
- The highest-order term (*n* ^3) dominates the overall runtime
- We say that this algorithm runs "on the order of" *n* ^3
- Or O(*n* ^3) for short (Big -O)

Big O Notation:

We say that $f(n)$ is $O(g(n))$ if there is a real number $c > 0$ and a constant integer $n_0 \geq 1$ such that:

$$f(n) \leq c\, g(n), \text{ for } n \geq n_0$$

- Example:
  - Suppose the function $f(n) = 8n - 2$, is $O(n)$
  - Justification: By the definition of Big O, we need to find a $c > 0$ and $n_0 \geq 1$ such that $8n - 2 \leq cn$
    - $8n - 2 \leq cn$
    - If $c = 8$
      - $8n - 2 \leq 8n$
    - Then if $n_0 = 1$
      - $6 \leq 8$
  - The possible choices for c is 8 and n_0 is 1

- Largest exponent rule: If $a_k > 0$, then $a_k n^k + a_{k-1} n^{k-1} + ... + a_1 n + a_0$ is $\Theta(n^k)$.
  (the a's and k's are constants
- Addition rule: If f(n) is O(g(n)) and h(n) is O(i(n)), then f(n) + h(n) is O(g(n) + i(n))
- Multiplication rule: If f(n) is O(g(n)) and h(n) is O(i(n)), then f(n)h(n) is O(g(n)i(n))
- For any constant k > 0, n^k increases at a faster rate than logn. In other words, logn is $O(n^k)$, but n^k is not O(logn)

Big Theta, Big Omega:

- Big O gives an upper bound to an algorithm. Big Theta and Big Omega gives the average case and best case (Lower Bound) respectively.

Divide and Conquer recursion:

- Divide and conquer recursions are analyzed by the master theorem
- Divide and conquer recursions analyzes functions with parameter ***n*** by making some ***S*** recursive calls to functions with parameters ***n/d*** and does some non-recursive work

Master Theorem:

- If **T(n) = S*T(n/d) + f(n)** where ***f(n)*** is the total cost of the non-recursive steps
- **T(n) =**
  - **$\Theta(n^{\log(dS)})$** if f(n) < $n^\wedge\log_d 5$
  - **Θ(f(n))** if f() > $n^\wedge\log_d 5$
  - **Θ(f(n)lgn)** if f(n) = $n^\wedge\log_d 5$

Example 1:

T(n) = 4T(n/2) + n

S = 4

Non recursive = n

d = 2

$T(n) = n^\wedge\log_2 4 + n$

$T(n) = n^2 + n$

$T(n) = \Theta(n^2)$ Case 1

Example 2:

$T(n) = 2T(n/2) + n^3$

S = 2

Non recursive = n^3

d = 2

$T(n) = n^\wedge\log_2 2 + n^3$

$T(n) = n + n^3$

$T(n) = \Theta(n^3)$ Case 2

Example 3:

$T(n) = 4T(n/2) + n^2$

$S = 4$

Non recursive = n^2

$d = 2$

$T(n) = n$^$\log_2 4 + n^2$

$T(n) = n^2 + n^2$

$T(n) = \Theta(n^2 \lg n)$ Case 3