Algorithms

Running time

The amount of time it takes to execute a program depends on both the amount of basic operations, and the time it takes to execute each operation. This concept is shown below as an equation.

$$\texttt{Count of Basic Operations} - C(n)$$
$$\texttt{Time to Execute Basic Operations} - C_{op}$$
$$\texttt{Running Time}\ X(n) = C_{op} * C(n)$$

Example: How much longer would it take if we double the size of the input?

$$C(n) = n^2/2$$
$$X(n) = C_{op} * n^2/2$$

To show how the run time is affected by the size of the input doubling and find the ratio of running time, we show the ratio of running times and expand the variables.

$$X(2n)/X(n)$$
$$(C_{op} * 4n^2/2)/(C_{op} * n^2/2)$$
$$(C_{op} * 4n^2/2) * (2/C_{op} * n^2)$$

Cross out the variables that appear on the top and bottom of the equation which leaves a final answer of 4 times

$$X(2n)/X(n) = 4$$

As seen in the above example, C_{op} is crossed out in the equation which shows that run time does not depend on the computer. This means that order of growth is the more important factor.

Example: When is A_1 the preferred algorithm and when is A_2 the preferred algorithm for the same problem?

This example is important because it shows us that a specific program does not work best with one algorithm alone. It shows that after the conditions change like the input size in this case, one algorithm might work better than the previous one given the new conditions

Let t be the time it takes for Algorithm 2's basic operation to be performed

For input size n,

Algorithm 1 takes n*1000t time
Algorithm 2 takes n^2 * t time

$$n^2 * t < n * 1000t$$
$$n \leq 1000$$

From the equation above, it is shown that for an input size n of less than or equal to 1000, Algorithm 2 would be a better choice for the program than Algorithm 1.

Runtime Speed: In terms of n (Complexity Categories)

`From fastest to slowest`

$\Theta(1)$ - `Runtime of 1, ie; runs once`

$\Theta(log * n)$ - `Runtime of iterated log`

$\Theta(lgn)$ - `Runtime of regular log`

$\Theta(n)$ - `Runtime of n`

$\Theta(n * lgn)$ - `Runtime of n * log of n`

$\Theta(n^2)$ - `Runtime of n squared`

$\Theta(n^j)$ - `j < k`

$\Theta(n^k)$ - `j < k`

$\Theta(a^n)$ - `a < b`

$\Theta(b^n)$ - `a < b`

As the number n inside theta becomes bigger, the runtime of course will increase. While not every value will be represented by the equations here, these can be used to give a rough estimation as to the runtime of a algorithm, and the overall effectiveness of such a equation.

Determining Order of Growth from an Equation

$T(n) = n^2$ will be worse than $T(n) = n$ when used as an algorithm for a problem because as the input gets bigger, the time it takes to complete the problem will increase.

Eventually, the linear algorithm (n) will always be faster than the quadratic algorithm (n^2).

Now, when you see an equation for an algorithm, you might wonder what terms matter in terms of order of growth.

For example, we have an equation for an algorithm as shown below

$$100n^2 + 20t - 50$$

The term with the largest order will always count for the majority of the order of growth so there are two steps for simplifying the equation in order to find the order of growth.

1) Drop Multiplicative Constants

$$n^2 + 20t - 50$$

2) Drop Lower Order Terms

$$\Theta(n^2)$$

This is what we are left with after the simplification and is also what we will use later in the lecture(s) when learning about Big-O and Big-Omega notation.

U.S. Population Example

Let us assume that a basic operation (n) has a completion time of 10^{-9} seconds

Algorithm	Complexity Category	Actual Running Time
`Algo 1`	$\theta(n^2)$	2.3 years
`Algo 2`	$\theta(n)$	0.27 seconds
`Algo 3`	$\theta(lgn)$	$2.8 * 10^{-3}$ seconds

Fig. 1 running time of the algorithms

As shown in Fig-1 above, Algorithm 3 is better than the options. It is important to keep run time and efficiency in mind when designing algorithms, as not only does it make things faster, it will require less resources and therefore be more effective in the real world

Definition of Big- O Notation

Big - O

For a given complexity function $f(n)$, $O(f(n))$ is the set of complexity functions $g(n)$ for which there exists some positive real constant c and some nonnegative integer n_0 such that for all $n \geq n_0$,

$$g(n) \leq c \times f(n).$$