Algorithms

Every Case Time Complexity

When determining the time complexity of an algorithm, it is important to know how many times the basic operation is performed for each value of the input size, n. The time complexity, $\boldsymbol{T(n)}$, is termed as the **every-case time complexity**. The determination of the every-case time complexity is known as an **every-case time complexity analysis**. An example of an every-case time complexity analysis follows.

Matrix Multiplication

The algorithm used to multiply two matrices, with n rows and columns, has a triple nested for loop. Given the matrices, there are always n passes through each for loop, and since the basic operation is in the inner most for loop, the time complexity is $T(n) = n * n * n$. For the matrix multiplication alogrithm, regardless of what the input size n is, $T(n) = n^3$ all of the time considering that the basic operation is done the same number of times for all instances with an input size of n.

Worst Case Time Complexity

For many algorithms, the basic operation is not always done the same number of times as shown for the matrix multiplication algorithm, meaning those algorithms do not have an every-case time complexity. When an algorithm does not have an every-case time complexity, the worst-case time complexity should be considered. The worst-case time complexity, $\boldsymbol{W(n)}$, is the maximum amount of times that the algorithm with an input size of n will perform its basic operation. If $T(n)$ exists, then $\boldsymbol{T(n) = W(n)}$. An analysis of $W(n)$ for an algorithm in which $T(n)$ does not exist follows.

The basic operation for sequential search is the comparison of an item in an array, containing n items, with x. This basic operation is done at most n times. The cases where this would occur is if x is at the end of the array or not in the array at all, so $W(n) = n$.

Best Case Time Complexity

For an algorithm, there can also be a best-case time complexity, $\boldsymbol{B(n)}$, which is the least amount of times that the algorithm with an input size of n will perform its basic operation. If $T(n)$ exists, then $\boldsymbol{T(n) = B(n)}$. The following is an analysis of $B(n)$ for the sequential search algorithm.

The basic operation for sequential search will be executed n times. Given an array, S, that is not empty, n will always be greater than or equal to 1. This means that the algorithm will do its basic opeartion **at least one time**. The best case for sequential search would be when x is found at $S[0]$. When x is found at index 0, $B(n) = 1$ because the basic operation was executed only once.

Average Case Time Complexity

For a given algorithm, how the algorithms performs on average may be of more interest. The average of the number of times the algorithm does its basic operation for an input size of n is defined as the **average-case time complexity** which is termed as $\boldsymbol{A(n)}$. If $T(n)$ exists, then $\boldsymbol{A(n) = T(n)}$. In some cases it may be difficult to analyze the average case. The following is an average-case time complesity analysis for the sequential search algorithm.

Average Case Time Complexity Analysis of Sequential Search

Note, for this analysis we let the list index start at 1 and to to n.

Case 1: x is in the list L and is equally likely to be in any position.

That means for $1 \leq k \leq n, Prob[L[k] = x] = 1/n$

If x is at position k, the number of comparisons is k.

Hence

$$\begin{aligned} A(n) &= \sum_{k=1}^{n} \left(k \cdot \frac{1}{n} \right) \\ &= \frac{1}{n} \sum_{k=1}^{n} k \\ &= \frac{1}{n} \left(\frac{n(n+1)}{2} \right) \\ &= \frac{n+1}{2} \end{aligned}$$

In other words, about 1/2 of the list elements are checked, on average.

Case 2: x may not be in list L. Let x be in list L with probability P.

Let $P = Prob[x \in L]$

Then $1 - P = Prob[x \notin L]$

Then

$$\begin{aligned}
A(n) &= \sum_{k=1}^{n} \left(k \cdot \frac{1}{n}\right) P + n(1-P) \\
&= \left(\frac{1}{n} \sum_{k=1}^{n} k\right) P + n(1-P) \\
&= \left(\frac{1}{n}\right) \left(\frac{n(n+1)}{2}\right) P + n(1-P) \\
&= \left(\frac{n+1}{2}\right) P + n(1-P) \\
&= \frac{Pn}{2} + \frac{P}{2} + n - Pn \\
&= \frac{P}{2} + n - \frac{Pn}{2} \\
&= \frac{P}{2} + n\left(1 - \frac{P}{2}\right)
\end{aligned}$$

Complexity Functions

Any function that maps positive integers to non-negative real numbers are **complexity functions**. These functions are usually termed as $f(n)$ or $g(n)$. Examples of complexity functions follow:

$$f(n) = n$$

$$f(n) = n^2$$

$$f(n) = lgn$$

Since overhead operations (such as initializing variables prior to entering a loop) do not contribute to an algorithm's efficiency as the input size grows, these can be ignored when performing complexity anslysis.

Example

Suppose there are two algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ *for the same problem.*

Also suppose:

$\mathcal{A}_1$ has every case time complexity $T_1(n) = n$
$\mathcal{A}_2$ has every case time complexity $T_1(n) = n^2$

So $\mathcal{A}_1$ is more efficient than $\mathcal{A}_2$

But suppose each basic operation takes 1,000 times longer to execute in $\mathcal{A}_1$ than in $\mathcal{A}_2$.

Let t be the time $\mathcal{A}_2$ takes for its basic operations, then $\mathcal{A}_1$ would take $1000t$

Then for an input size of n

$\mathcal{A}_1$ takes $n \cdot 1000t$ time, and
$\mathcal{A}_2$ takes $n^2 \cdot t$ time

This means $\mathcal{A}_2$ is a better choice when

$$\begin{aligned} n^2 \cdot t &< n \cdot 1000t \\ n &< 1000 \end{aligned}$$

If you know all instances of the problem are $n < 1000$, then you should select $\mathcal{A}_2$, even though it is theoretically slower than $\mathcal{A}_1$.