Understanding different Time Complexities of algorithm:

Time Complexity of an algorithm gives us mathematical ways to describe the time required by an algorithm to complete execution. It is estimated by counting the number of elementary steps performed by any algorithm to finish execution.

i.e: The actual time required by an algorithm can vary from device to device and input however the working of algorithm stays same. Due to different factors we can get 0.5ms time for N=10 while 0.3ms for N=1000 for same algorithm. So it’s rather calculated by considering how much times statements execute.

The time complexity of algorithms can be denoted by using asymptotic notation “Big Oh, O()” which gives us worst case for time complexity i.e. the O(expression) is the set of functions that grow slower than or at the same rate as expression. It indicates the maximum required by an algorithm for all input values. It also excludes coefficients and lower order terms.

How to find a formula for time required for execution:

Supposing a model machine with following specifications:

* Single processor
* 32 bit
* Sequential execution
* 1 unit time for arithmetic and logical operations
* 1 unit time for assignment and return statements

Pseudocode:

sum(int A[], int n){//A->array and n->number of elements in the array

total =0 // Time=1, Loops=1

for i=0 to n-1 // Time=2, Loops=n+1 (+1 for the end false condition)

sum = sum + A[i] //Time=2, Loops=n

return sum // Time=1, Loops=1

}

Now the formula for time required for execution for given algorithm ‘T’ can be calculated as

T = 1 + (2 \* (n+1)) + (2 \* n) + 1

= 4n + 1

=C1 \* n + C2

Big Oh notation will exclude the coefficients and constants as well as the lower degree terms thus time complexity of given algorithm can be denoted as, ‘T = O(n)’.

Suppose you've calculated that an algorithm takes f(n) operations for ‘n’ input, such that

f(n) = 4\*n^2 + n + 3.

Then the growth of this polynomial function corresponds to growth in n^2. Then the time complexity is given by O(n^2).

Time complexity of algorithms:

a. O(1) time complexity: This category includes algorithms whose run time which doesn’t depend on the input size ‘n’. Thus it will always run in constant time so it is O(1).

Example 1:

#include<stdio.h>

int main()

{

print “hello”;

}

The code always runs for a fixed amount of time so has time complexity O(1).

Example 2:

function(n){

for(int i=0;i<5;i++)

{

print “hello”;

}

}

b. O(n) time complexity: This category includes algorithms whose run time which depend on the input size ‘n’ linearly. i.e: time required for execution increases linearly with increase in input size ‘n’.

Example 1:

function(n){

for(int i=0;i<n;i++)

{

print “hello”;

}

}

Example 2:

This algorithm is little different, it will print hello n/2 times. If we ignore the constant ½ we see that algorithm is O(n).

function(n){

for(int i=0;i<n;i+2)

{

print “hello”;

}

}

c. O(n^2) time complexity: This category includes algorithms whose run time which depend on the square of input size ‘n’. i.e: time required for execution increases quadratically with increase in input size ‘n’.

Example 1:

function(n){

for(int i=0;i<n;i++)

{

For(int j=0;j<n;j++)

{

print “hello”;

}

}

}

Example 2:

function(n){

for(int i=0;i<n;i++)

{

for(int j=0;j<n;j+2)

{

print “hello”;

}

}

}

As we can see the O(n^2) complexity is obtained by nesting two linear complexities ‘O(n)’s.

d. O(log(n)) time complexity: To understand O(log(n)) we must understand the logarithms. Logarithm is the inverse of exponential functions.

i.e a^x = y also means x = log\_a(y).

for T = O(log\_a(n)), n = a^T, which means when the input size ‘n’ increases exponentially time required for execution ‘T’ goes up linearly. Commonly we can find logarithm of base 2 being used in algorithms since computer is binary based.

Divide and Conquer algorithms have O(log(n)) complexity. For example binary search has O(log(n)) complexity since the binary tree is divided into two parts to search for the required element and the process is repeated ‘log(n)’ times for binary tree with ‘n’ elements.

Example 1:

function(n){

for(int i = 1; i <= n; i = i \* 2)

print "hello";

}

}

Example 2:

while(low <= high)

{

mid = (low + high) / 2;

if (target < list[mid])

high = mid - 1;

else if (target > list[mid])

low = mid + 1;

else break;

}

e. O(n\*log(n)) time complexity: This category consists of ‘n’ loops that are logarithmic.

Example 1:

function(n){

for(int i = 0; i < n; i++)

{

for(int j = 1; j < n; j = j \* 2)

{

print "hello";

}

}

}

Quick Sort algorithm has O(n\*log(n)) complexity which is given below.

Example 2:

void quicksort ( int list[], int left, int right )

{

int pivot = partition ( list, left, right );

quicksort ( list, left, pivot - 1 );

quicksort ( list, pivot + 1, right );

}

We can better understand time complexities through this real life examples:

Suppose a dictionary with ‘n’ pages with ‘n’ words in each page.

O(1) complexity: Opening a page of dictionary and finding any five words from the dictionary since the time required to do those tasks is independent of the input and constant.

O(n) complexity: Finding a page of dictionary by going through each page from first page to last.

O(n^2): Finding a word in the dictionary by going from first page to last (n pages) and also through each word in each page. (n words in each n page). Thus the number of times we must repeat the process can be calculated as n\*n=n^2.

O(log(n)) complexity: Finding a page of dictionary by dividing the pages into two equal halves and repeating the process. Since dictionary is ordered alphabetically one of two sections will contain the required word. The number of times we must repeat the process can be calculated as log\_2(n).

O(n\*log(n)) complexity: Finding a word in the dictionary by going through each page from first to last and dividing each individual page into two and repeating the process. That is go through pages serially and use logarithmic process for each page to find the word. The number of times we must repeat the process can be calculated as n\*log\_2(n).

Sources:

<https://stackoverflow.com/questions/2307283/what-does-olog-n-mean-exactly#:~:text=Logarithmic%20running%20time%20(%20O(log,an%20O(log%20n)%20time>

<https://www.studytonight.com/data-structures/time-complexity-of-algorithms>