Time Complexity of Algorithms

Time complexity of an algorithm signifies the total time required by the program to run to completion.

The time complexity of algorithms is most commonly expressed using the **big O notation**.

Time Complexity is most commonly estimated by counting the number of elementary functions performed by the algorithm. And since the algorithm's performance may vary with different types of input data, hence for an algorithm we usually use the **worst-case time complexity** of an algorithm because that is the maximum time taken for any input size.

Calculating Time Complexity:

Now the most common metric for calculating time complexity is Big O notation. This removes all constant factors so that the running time can be estimated in relation to **N**, as N approaches infinity.

In general you can think of it like this :

```
statement;
```

Above we have a single statement. Its Time Complexity will be **Constant**. The running time of the statement will not change in relation to N.

```
for(i=0; i < N; i++)
{
  statement;
}
```

The time complexity for the above algorithm will be **Linear**. The running time of the loop is directly proportional to N.

```
for(i=0; i < N; i++)
{
  for(j=0; j < N;j++)
  {
    statement;
  }
}
```

This time, the time complexity for the above code will be **Quadratic**. The running time of the two loops is proportional to the square of N. When N doubles, the running time increases by N * N.

```
while(low <= high)
{
  mid = (low + high) / 2;
  if (target < list[mid])
    high = mid - 1;
  else if (target > list[mid])
    low = mid + 1;
  else break;
}
```

This is an algorithm to break a set of numbers into halves(two part), to search a particular field.

Now, this algorithm will have a **Logarithmic** Time Complexity. The running time of the algorithm is proportional to the number of times N can be divided by 2. This is because the algorithm divides the working area in half with each iteration.

```
void quicksort(int list[], int left, int right)
{
  int pivot = partition(list, left, right);
  quicksort(list, left, pivot - 1);
  quicksort(list, pivot + 1, right);
}
```

Taking the previous algorithm forward, above we have a small logic of Quick Sort. Now in Quick Sort, we divide the list into halves every time, but we repeat the iteration N times (where N is the size of list). Hence time complexity will be **N*log(N)**. The running time consists of N loops (iterative or recursive) that are logarithmic, thus the algorithm is a combination of linear and logarithmic.

NOTE : In general, doing something with every item in one dimension is linear, doing something with every item in two dimensions is quadratic, and dividing the working area in half is logarithmic.

Types of Notations for Time Complexity

Now we will discuss and understand the various notations used for Time Complexity.

1. **Big Oh** denotes "*fewer than or the same as*" <expression> iterations.
2. **Big Omega** denotes "*more than or the same as*" <expression> iterations.
3. **Big Theta** denotes "*the same as*" <expression> iterations.
4. **Little Oh** denotes "*fewer than*" <expression> iterations.
5. **Little Omega** denotes "*more than*" <expression> iterations.

Understanding Notations of Time Complexity with Example

O(expression) is the set of functions that grow slower than or at the same rate as expression.

Omega(expression) is the set of functions that grow faster than or at the same rate as expression.

Theta(expression) consist of all the functions that lie in both O(expression) and Omega(expression).

Suppose you've calculated that an algorithm takes f(n) operations, where,

f(n) = 3*n^2 + 2*n + 4. // n^2 means square of n

Since this polynomial grows at the same rate as **n^2**, then you could say that the function **f** lies in the set **Theta(n^2)**. (It also lies in the sets **O(n^2)** and **Omega(n^2)** for the same reason.)

The simplest explanation is, because **Theta** denotes *the same as* the expression. Hence, as **f(n)** grows by a factor of **n^2**, the time complexity can be best represented as **Theta(n^2)**.

Examples:

Sequential Search Pseudocode

```
Read all the names into array NAMES.
index           = 0
matchFound      = false
locationOfMatch = -1

while( matchFound == false and index < length_of_NAMES_array ) do
  if( NAMES[ index ] == "Debbie Drawe" ) then
    matchFound = true
    locationOfMatch = index
  else
    index = index + 1
endWhile

if( matchFound == true ) then
  print( "Match found at name " & index )
else
  print( "Match not found" )
EndOfAlgorithm
```


Sequential Search Performance

- What is best case performance?
- 1*(while loop time) + overhead
- What is average case performance?
- (n/2)*(while loop time) + overhead
- What is worst case performance?
- n*(while loop time) + overhead

Insertion Sort

- Given a set to be sorted, use the card player's approach:
  - Add each new item at the appropriate place in the set of already sorted items.

Insertion Sort Pseudocode

```
Read all the numbers into array NUMS.

numberIndex = 1
sortedIndex = 0

while( numberIndex < length_of_NUMS_array ) do
  newNum      = NUMS[ numberIndex ]
  sortedIndex = numberIndex - 1

  /* From high to low, look for the place for the new number.
     If the previously sorted numbers are larger, move them up
     in the array NUMS.
  */
  while( NUMS[ sortedIndex ] > newNum and sortedIndex >= 0 ) do
    NUMS[ sortedIndex + 1 ] = NUMS[ sortedIndex ]
    sortedIndex = sortedIndex - 1
  endWhile

  /* We found the place for the new number, so insert it. */
  NUMS[ sortedIndex + 1 ] = newNum
  numberIndex++
  sortedIndex = numberIndex - 1
endWhile
```

Performance of Insertion Sort

- Execution time will increase with the size of the set to be sorted (outer **`while`** loop)
- Each element to be sorted must be compared one or many times with the elements already sorted (inner **`while`** loop).

Insertion Sort Performance

- What is best case performance?
- Items already sorted
- n*(outerWhile) + 1*(innerWhile) + overhead
- What is average case performance?
- n*(outerWhile) + n*(n/2)*(innerWhile) + overhead
- What is worst case performance?
- Items already sorted in reverse order
- n*(outerWhile) + ((n^2 – n)/2)*(innerWhile) + overhead
- What is Theta?
- **$\Theta(n^2)$.**