Ayan Kumar Chatterjee
MCA (IGNOU), Diploma in Software Engineering (Aptech Computer Education),
Teacher of Computer Science and Software Technology
1/1A, Ananda Chatterjee Lane, Kolkata 700003. Phone: 033 2554 6084. Email: ayankumarchatterjee@yahoo.com

SORTING ALGORITHM COMPLEXITY

WHAT IS COMPLEXITY?

In general terms, complexity tries to estimate the performance of an algorithm. There are two types of complexities that are analyzed for every algorithm. These are called **Space Complexity** and **Time Complexity**. In space complexity, we try to measure the amount of memory space required by an algorithm to solve the designated problem. As a general rule, all recursive algorithms consume more memory space than linear algorithms. On the other hand the time complexity tries to measure the amount of time taken by an algorithm to solve the given problem. The less time it takes, the more efficient the algorithm proves to be.

HOW TO COMPUTE THE TIME COMPLEXITY?

Step1: count the number of instructions

Let us consider the given code for printing the content of an N element array.

```
int i;

for(i = 0 ; i < n ; i++)
        printf("\t%d", arr[i]);
```

The very first statement is for declaring the counter variable i. (1 instruction)
The initialization statement i = 0. (1 instruction)
The condition i < n (1 instruction)
The body of the loop comprising the printf() function. (1 instruction)
The update statement i++. (1 instruction)

As the loop will be executed n number of times, the printf(), the update statement and the condition will be executed as a set for n number of times as well.

So, taken together, we have
1 declaration statement + 1 initialization statement + 1 condition statement (before entering the body of the loop) + { set of printf() + update + condition } x n

Which equals to 1 + 1 + 1 + (1 + 1 + 1) x n = 3n + 3 number of instructions.

Step2: worst case analysis

We have to find out the maximum number of instructions that our algorithm may take to solve the problem. The first example we are dealing in is no simple that this step is unnecessary, but we shall discuss it in other cases.

Step3: eliminating unnecessary terms by the nature of Asymptotic Behavior

Here we need to find out which of the terms are of real importance and which are not. This is done by considering which of the terms grows fast in case the number of elements in the array (n) increases. For example, it can be understood very easily that the last factor 3 in 3n + 3 will still remain 3 if the value of N is increased from 5 to 10000. So that 3 can be easily eliminated as it does not affect the performance in a big way.

Similarly, we can also drop the 3 from the term 3N, if we understand that the 3 is nothing more but an initialization constant. What really matters is the value of N, if

Ayan Kumar Chatterjee
MCA (IGNOU), Diploma in Software Engineering (Aptech Computer Education),
Teacher of Computer Science and Software Technology
1/1A, Ananda Chatterjee Lane, Kolkata 700003. Phone: 033 2554 6084. Email: ayankumarchatterjee@yahoo.com

that increases the complexity will increase three fold. The value 3 in itself has not much of an importance.

Therefore the complexity of the algorithm finally comes down to n. So f(n) = n.

The process of dropping the factors and keeping the largest growing item is called **Asymptotic Behavior**. The asymptotic behavior of f(n) = 2n + 8 will still be n, as described in the previous example.

Some more example:

f(n) = 109 gives f(n) = 1. In this case 109 is considered as 109 x 1, and the initialization constant 109 can be dropped.

$f(n) = n^2 + 3n + 112$ gives $f(n) = n^2$, as n^2 grows faster than the growth achieved by 3n.

$f(n) = n^3 + 1999n + 1337$ gives $f(n) = n^3$

$f(n) = n + \sqrt{n}$ gives $f(n) = n$

$f(n) = 3^n + 2^n$ gives $f(n) = 3^n$

$f(n) = n^n + n$ gives $f(n) = n^n$

Let us examine another program, finding the largest value in an array of n elements.

```
max = a[0];

for(i = 0 ; i < n ; i++)
{
        if(a[i] > max)
                max = a[i];
}
```

Step1: count the number of instructions
Assigning a[0] to max (1 instruction)
Initializing i = 0 (1 instruction)
The condition (i < n) (1 instruction)
The if statement (1 instruction)
The assignment max = a[i] (1 instruction)
The update i++ (1 instruction)

As the loop will be executed n number of times, the if statement, the assignment statement, the update statement and the condition will be executed as a set for n number of times as well.

So, taken together, we have
1 declaration statement + 1 initialization statement + 1 condition statement (before entering the body of the loop) + { set of if + assignment + update + condition } x n

Which equals to 1 + 1 + 1 + (1 + 1 + 1 + 1) x n = 4n + 3 number of instructions.

Ayan Kumar Chatterjee
MCA (IGNOU), Diploma in Software Engineering (Aptech Computer Education),
Teacher of Computer Science and Software Technology
1/1A, Ananda Chatterjee Lane, Kolkata 700003. Phone: 033 2554 6084. Email: ayankumarchatterjee@yahoo.com

Step2: worst case analysis
f(n) = 4n + 3 is a worst case scenario, because we are assuming that for every condition checking the if condition will be true. This is especially applicable if we consider that the array is ascendingly sorted (1, 2, 3, 4, …).

In case the array is sorted in a descending order (…, 4, 3, 2, 1) none of the if condition will ever evaluate to true, and the execution of the statement max = a[i] will no longer be required, immediately bringing down the complexity to 1 + 1 + 1 + (1 + 1 + 1) x n = 3n + 3. This is the best case scenario.

Step3: eliminating unnecessary terms by the nature of Asymptotic Behavior
As we can understand by now, whether the best case or the worst, f(n) = n.

We can now say as a rule of thumb that any program that does not contain a loop will have f(n) = 1. Any program having a single loop running from 1 to n will have f(n) = n.

Let us now examine a case of nested loop. The following code reports if there is a duplicate value within an array.

```
char duplicate = 'F';

for (i = 0; i < n; i++)
{
        for (j = 0; j < n; j++)
        {
                if ( i != j && a[i] == a[j] )
                {
                        duplicate = 'T';
                        break;
                }
        }
        if (duplicate == 'T')
        {
                printf("\n\tThe array contains duplicate values.");
                break;
        }
}
```

The outer loop rotates n times having complexity of n. Within that the inner loop also rotates for n times. The combined complexity now becomes a multiplication of these two, setting $f(n) = n^2$.

We can understand a loop within a loop within a loop will produce $f(n) = n^3$.

If we have a loop that iterates for n times, and it calls a function from within its body that also has a complexity of n, the complexity of the whole program will still be n^2.

Two nested loops followed by a single loop is asymptotically the same as the nested loops alone, because the nested loops dominate the simple loop and runs more slowly than the simple loop.

THE NOTATIONS USED TO REPRESENT TIME COMPLEXITY
Big O notation: this is used to represent the worst case scenario. In the example of finding the largest value in the array, the worst case scenario for ascendingly sorted array was computed as 4n + 3. So f(n) = O(n). Simply speaking, the complexity of the

Ayan Kumar Chatterjee
MCA (IGNOU), Diploma in Software Engineering (Aptech Computer Education),
Teacher of Computer Science and Software Technology
1/1A, Ananda Chatterjee Lane, Kolkata 700003. Phone: 033 2554 6084. Email: ayankumarchatterjee@yahoo.com

code cannot be any worse than what is represented in Big O notation. This finds the maximum number of times the algorithm would have to perform to do its job.

Omega notation: contrary to the Big O notation the omega notation represents the best case scenario. In that very same algorithm, the best case scenario of the descendingly sorted array was computer as 3n + 3. So f(n) = Ω(n). Again, in simple term the complexity of the algorithm cannot be any better than what is represented in Omega notation. This finds the minimum number of times the algorithm would have to perform to do its job.

Theta notation: the theta notation determines the average case, something in between the best case and the worst. The Big O notation provides the upper bound (like, the algorithm will need no more than so many number of steps, but could still be less than that). The Omega notation provides the upper bound (like, the algorithm will need no less than so many number of steps, but could still be more than that). Theta notation provides the tight or strict bound (like, the algorithm will strictly need so many number of steps, neither more nor less). In case where the Big O notation and the Omega notation are the same the Theta notation will be the same too. We have already seen that for the algorithm for finding the largest in the array f(n) = O(n) and f(n) = Ω(n). Therefore in this case f(n) = Θ(n).

TIME COMPLEXITIES OF DIFFERENT SORTING ALGORITHMS

Bubble Sort

In the **best case**, where the array is already sorted in ascending order, in the first iteration of the outer loop the number of comparisons made will be n-1. In the second iteration there will be n-2 comparisons and so on. Therefore the total number of comparisons will be

$$(n-1) + (n-2) + \ldots + 2 + 1 = O(n^2)$$

In the **worst case**, where the array is sorted in descending order, in the first iteration of the inner loop the number of comparisons made will be n-1. In the second iteration there will be n-2 comparisons and so on. Therefore the total number of comparisons will still be

$$(n-1) + (n-2) + \ldots + 2 + 1 = O(n^2)$$

In the **average case**, where the array is unsorted, the number of comparisons made by the inner loop will be (k-1)/2 for k number of values. Therefore the total number of comparisons will still be

$$(n-1) + (n-2) + \ldots + 2 + 1 = O(n^2)$$

Selection Sort

Exactly the same as Bubble Sort

Insertion Sort

In the **best case**, where the array is already sorted in ascending order, in every iteration of the outer loop the number of swapping will be nil. Therefore the total number of comparisons will be

$$1 + 1 + \ldots \text{ for n number of times} = O(n)$$

Ayan Kumar Chatterjee
MCA (IGNOU), Diploma in Software Engineering (Aptech Computer Education),
Teacher of Computer Science and Software Technology
1/1A, Ananda Chatterjee Lane, Kolkata 700003. Phone: 033 2554 6084. Email: ayankumarchatterjee@yahoo.com

In the **worst case**, where the array is sorted in descending order, in the first iteration of the inner loop the number of comparisons made will be n-1. In the second iteration there will be n-2 comparisons and so on. Therefore the total number of comparisons will still be

$1 + 2 + \ldots + (n-2) + (n-1) = n(n-1)/2 = O(n^2)$

In the **average case**, where the array is unsorted, the number of comparisons made by the inner loop will be (k-1)/2 for k number of values. Therefore the total number of comparisons will still be

$1/2 + 2/2 + \ldots + (n-2)/2 + (n-1)/2 = n(n-1)/4 = O(n^2)$

Merge Sort

Let f(n) denote the number of comparisons needed to sort an n element array. This needs at most (log n) passes. Each pass merges a total number of n elements, so each pass should need at most n number of comparisons. Hence the complexity of the merge sort is O(n*log n).

Quick Sort

In **Best Case**, the array is partitioned into half and each half is sorted. If c is a constant, this means that the number of comparisons would be

$c*n + 2*T(n/2)$
$= c*n + 2*(c*n/2 + 2*T(n/2^2))$
$= c*n + c*n + 22 + T(n/2^2)$
$= 2*c*n + 22 + T(n/2^2)$
:
:
$= k*c*n + 2^k*+ T(n/2^k)$ [at k^{th} step]
$= \log_2 n*c*n + n*T(1)$
$= \log_2 n*c*n + n*c$
$= O(n*\log n)$

In **Worst Case**, the array is partitioned into half but the halves are sorted in descending order.

$T(n-1) + c*n$
$= T(n-2) + c*(n-1) + c*n$
$= T(n-3) + c*(n-2) + c*(n-1) + c*n$
:
:
$= T(1) + c*(2 + 3 + 4 + \ldots + (n-1) + n)$
$= c*n(n+1)/2$
$= O(n^2)$

Radix Sort or Bucket Sort

Suppose A is the array of n elements and the radix is d, and s is the maximum number of digits of all data items. The algorithm would therefore need a maximum of s passes. Hence, T(n) = s*d*n

Although d is independent of n, the number s depends on n. In the **worst case**, s = n, so $T(n) = O(n^2)$. In the **best case**, s = $\log_d n$, so T(n) = O(n*log n). In other words,

Ayan Kumar Chatterjee
MCA (IGNOU), Diploma in Software Engineering (Aptech Computer Education),
Teacher of Computer Science and Software Technology
1/1A, Ananda Chatterjee Lane, Kolkata 700003. Phone: 033 2554 6084. Email: ayankumarchatterjee@yahoo.com

radix sort performs well only when s (the maximum number of digits in a number) is small.

Heap Sort

Suppose A is the array of n elements. The heap sort has two phases; we are discussing the complexity of each phase separately.

Phase 1: Suppose H is the heap. We can understand that the number of comparisons needed to find the appropriate position of a new element in H cannot exceed the depth of H, since H is a complete tree. So, the depth of H is bounded by $\log_2 n$, where n is the number of elements in H. accordingly the total number of comparisons needed to insert the n number of elements into H is bounded by $n*\log_2 n$.

Phase 2: The re-heaping of H needs 4 comparisons, (root < left child, root < right child, left child < right child and comparison with the end of the level). Since the depth of H does not exceed $\log_2 n$, re-heaping uses at most 4* log2n comparisons to find the appropriate place of a node in H. This also means that the total number of comparisons needed to delete n elements from H would not exceed $4* n*\log_2 n$. Therefore the running time of phase 2 of heap sort is proportional to $n* \log_2 n$.

So the **worst case** time complexity of heap sort is T(n) and both **best case** and **average case** is $O(n* \log_2 n)$.