12

Computational Complexity

An algorithm is a well-defined list of steps for solving a particular problem. It is a sequence of computational steps that transform the input into the output. The complexity of an algorithm is the function which gives the running time and/or space in terms of input size. The performance of a program is measured through its complexity.

Computational complexity, focuses on the amount of computing resources needed for a particular algorithm to run efficiently. The efficiency of a program can be measured in terms of space and time.

Space Complexity

The space complexity of an algorithm is the amount of memory required for its execution.

Time Complexity

The time complexity of an algorithm is the amount of time required for its execution. Sometimes, the choice of program involves a time-space trade off : by increasing the amount of space for storing the data, one may be able to reduce the time needed for processing the data or *vice-versa*.

Estimating Complexity of Algorithm

To choose the best algorithm, we need to check efficiency of each algorithm. The efficiency can be measured by computing time complexity of each algorithm. Actual runtime of an algorithm depends upon speed of computer, amount of RAM, operating system and the compiler being used. Hence, we cannot use actual runtime of algorithms for comparison between two algorithms. So, for time complexity estimation of an algorithm, we need to count the number of elementary instructions that executes this algorithm. This number is computed with respect to the size n of the input data.

Big-O Notation

It is a method of representing the upper bound of algorithm's running time. It is also known as Omega Notation. Using Big-O notation, we can give longest amount of time taken by the algorithm to complete. e.g. if we have two algorithms having time complexity $O(N)$ and $O(N^2)$ respectively then by varying value of N, we can easily check which algorithm is more efficient, i.e. which one takes less time to run.

Classification of Different Order of Growth

Class Name	Order of Growth	Description	Example
Constant	1	As input size grows, we get larger running time.	Scanning array elements.
Logarithmic	$\log n$	The algorithm does not consider all its input, divided into smaller parts on each integration.	Performing binary search.
Linear	n	The running time of algorithm depends on the input size n.	Performing sequential search operation.
$n\log n$	$n\log n$	Some instance of input is considered for the list of size n.	Merge sort or quick sort.
Quadratic	n^2	When the algorithm has two nested loops then this type of efficiency occurs.	Scanning matrix elements.
Cubic	n^3	It occurs when algorithm has three nested loops.	Performing matrix multiplications.
Exponential	2^n	When the algorithm has very faster rate of growth then it occurs.	Generating rebuts of n inputs.

Calculating Complexity

For finding out the time complexity of a piece of code, we need to consider for loops, nested loops, if-then-else, consecutive statements and logarithmic complexity.

for loop

e.g.
```
for (i = 0; i < N; i++)
{
    count++;
}
```

Every time this statement will take constant time i.e. 'C' for execution.

Loop will execute for i values from 0 to $N - 1$.

$$\text{Total time} = CN \text{ i.e. } O(N) \text{ [Linear]}$$

for nested loop

e.g.
```
for (i = 0; i < N; i++)
{                                                          Outer loop will
    for (j = 0; j < N; j++)                                execute N times
    {
        count++; //will take constant    Inner loop will
        time 'C' to execute.             execute N times
    }
}
```

i.e. whenever outer loop condition will be true inner loop will execute N times.

Since, outer loop is executing total N times hence, inner loop will execute $N \times N$ times.

$$\text{Total time} = C * N * N = CN^2, \text{ i.e. } O(N^2)$$

if-then-else statement

e.g.
```
if (amt > cost + tax) -> c0 time
{
    count = 0; -> c1 time
    while (count < n)
    {
        amt = (cost + tax); -> c2 time     will execute
        count++; -> c3 time                  N times
    }
    count<< "Capacity :" << count; -> c4 time
}
else
    count<< "Insufficient funds"; -> c5 time
```

$$\text{Total time} = c_0 + c_1 + n * (c_2 + c_3) + c_4 + c_5, \text{ i.e. } O(n).$$

consecutive statement

e.g.
```
for (i = 0; i < n; i++)
{                                                            will execute
    A[i]= (1- t) * X[i]+ t * y[i]; //constant time c0          N times
    B[i]= (1- s) * X[i]+ s * y[i];
}
for (i = 0; i < n; i++)
{
    for (j = 0; j < n; j++)                                  outer loop will
    {                                                        execute N times
        c[i,j]= j * A[i]+ i * B[j]; //constant time c1
        z= z+ 1; //constant time c2
    }
}
Inner loop
will execute
N times
```

So, $\text{Total time} = n * c_0 + n^2 c_1 + c_2, \text{ i.e. } O(n^2)$

logarithmic complexity

Algorithm taking logarithmic time are commonly found in operations on binary trees or when using binary search. An algorithm is said to take logarithmic time if,

$$T(n) = O(\log n)$$

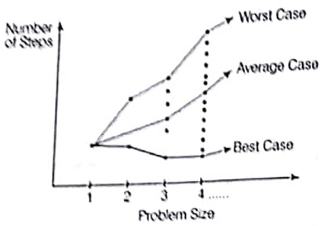
Best, Average and Worst Case Complexity

There are three cases to analyse algorithm complexities:

In **best case complexity**, we calculate lower bound of running time of an algorithm. We must know, the case that cause **minimum number of operations** to be executed. It describes an algorithm's behaviour under optimal conditions.

In **worst case complexity**, we calculate upper bound of running time of an algorithm. We must know, the case that causes **maximum number of operations** to be executed.

In **average case complexity**, we consider all possible inputs and calculate computing time for all of the inputs.



Best, Worst and Average Case Complexity

Running Time Complexity

It determines how much times does each function require?

Running Time Complexities

$O(1)$	Fast
$O(\log(n))$	Fast
$O(n)$	Fast for some things, slow for others
$O(n*\log(n))$	Fast-compared to $O(n^2)$
$O(n^2)$	Slow for most things
$O(2^n)$	Intractable
$O(n!)$	Intractable

Analysing Algorithms

Analysing an algorithms, means predicting the resources that the algorithm requires. Now, we consider following examples of sortings and searchings to analysis the algorithm.

(i) Bubble Sort

```
BubbleSort (a : data_array; n : integer)
  for i ← 1 to n do
   for j ← n to i + 1 do
     if a[j] < a[j - 1] then
        swap( a[j], a[j - 1])
```

In best, worst and average case, bubble sort takes $O(n)^2$ time.

(ii) Selection Sort

```
SelectionSort (a : data_array; n : integer)
  for i ← 1 to n - 1 do          loop executes n - 1 times
    low ← j
    for j ← i + 1 to n do        Number of comparisons
     if a[j] < a[low] then       n-1, then n-2, ...
        low ← j
  swap(a[i], a[low])             n-1 swaps, one for each value of i
```

In best, worst and average case, selection sort takes $O(n)^2$ time.

(iii) **Insertion Sort** Insertion sort (a : data_array; n : integer)

```
for i ← 2 to n do
    j← i, tmp ← A[i]           //n-1 iterations
    while(tmp < A[j-1])        //2 assignments
        A[j] ← A[j-1]          //do up to i list item comparisons
        j ← j-1                //1 assignment
    A[j] ← tmp                 //1 assignment
                               //1 assignment
```

Best Case When the array is already sorted, insertion sort takes $O(n)$ time.

Worst Case In worst case insertion sort takes $O(n^2)$ time.

We start by presenting the insertion sort procedure with the time cost of each statements and the number of times each statement is executed. For each j = 2,3 n, where n = length [A], we let t_j be the number of times the while loop test in line 3 is executed for that value of j. When a for or while loop exits in the usual way (i.e. due to the test in the loop header), the test is executed one time more than the loop body.

(iv) **Linear Search** A linear search is a sequential search. A linear search sequentially moves through your collection looping for a matching value.

```
LinearSearch(int arr[], int val)
i ← 1                             //1 assignment
while(i ≤ n and val ≠ arr[i])     //2 comparisons each time
      i ← i + 1                   //1 assignment each time
if i ≤ n then location ← i        //1 comparison and 1 assignment
else location ← 0
```

Best Case val must be present in the first position on the list (named arr [ ]). In this case, two comparisons of while loop and one comparison of if statement will be required. Hence, complexity is $O(1)$ i.e. constant.

Worst Case for unsuccessful search (i.e. when value to be searched will not be there in list), total number of comparisons would be $2n + 2$ and for successful search (i.e. when the required element will be the last element in the list), total number of comparisons required will be $2n + 1$. Hence, complexity is $O(n)$.

(v) **Binary Search**

```
BinarySearch(x : int a1, a2, . an, int in)
                               //left endpoint to search interval(1 assignment)
i ← 1                          //right endpoint of search interval(1 assignment)
j ← n                          //1 assignment
i ← j
in ← [(i + j) / 2]
if x > an then i ← n + 1       //1 comparison and 1 assignment
else j ← n
or
if x = a1 then location ← i    //1 comparison and 1 assignment
else location ← 0
```

Best Case Binary search gives best case when the element to be searched is the middle element. Best case time complexity of binary search is $O(1)$, i.e. constant.

Worst Case In worst case as well as in average case binary search takes $O(\log n)$ time.

Previous Years' Examination Questions

1 Mark Questions

1. **What is the worst case complexity of the following code segment?** 2010

```
for(int i = 0; i < N; i ++)
{
    sequence of statements:        //Statement 1
}
for(int j=0; j<M; j ++)
{
    sequence of statements:        //Statement 2
}
```

Ans The first loop will execute N times, so the worst case complexity of the first loop is $O(N)$. The second loop will execute M times, so the worst case complexity of the second loop is $O(M)$. So, total time complexity of the code segment would be $O(N + M)$.

2. **How would the complexity change if the second loop went on N instead of M?** 2010

Ans If the second loop went to N instead of M then, the complexity will become $O(N)$, because both loops will execute N times each.

2 Marks Questions

1. **Define the terms complexity and Big-O notation.** 2014

Ans The term complexity means how much time and space the algorithm takes and how efficiently it can execute in terms of time and space. The Big-O notation is denoted by O. It is a method of representing the upper bound of algorithm's running time. Using Big-O notation we can give longest amount of time taken by the algorithm to complete.

2. **What is the worst case time complexity of the following code segment?** 2013

```
for(int p=0; p<N; p++)
{
    for(int q=0; q<M; q++)
    {
        sequence of statements:  //Statement 1
    }
}
for(int r = 0; r < X; r++)
{
    sequence of statements:      //Statement 2
}
```

Ans In the given code first loop is nested loop. The outer loop of first loop will execute N times and inner loop will execute M times and sequence of statements will take constant time i.e. c_1. So, total execution time of this outer loop would be $N * M * c_1$. And, second loop will execute X times and its sequence statement 2 will take constant time c_2. So,

$$\text{Total time} = N * M * c_1 + X * c_2 = O(NM + X)$$

3. **How would the complexity change if all the loops went upto the same limit N?** 2013

Ans If all the loop execute N times, the body of the loop get executed $N-1$ times, the condition will evaluate to false and loop terminated without executing the body.
So, the time complexity for such code would be:

$$\text{Total time} = c * N = cN \text{ i.e. } O(n)$$

4. **What is the role of constants in complexity? Explain briefly with an example.** 2012

Ans **Role of Constants in Complexity** Constant Time $O(1)$ function needs fixed amount of time to execute program of algorithm. It does not depend upon number of inputs.
e.g. You want to write a function that returns true or false based on the value of first element in an integer array. Program needs input of an integer array. If first element in array is greater than 0, function must return true. In all other conditions, function must return false.
In the above example, function execution time does not depend on number of elements. It just check the first element and returns the result. It does not matter if function get array of 1 integer or 1 million of integers. Hence, we can say that the running time of this function is constant. This function can be represented as $O(1)$ time complexity.

5. **What is Big-O notation?** 2011

or

What is Big-O notation? State its significance. 2010

Ans It is a method of representing the upper bound of algorithm's running time. It is used to evaluate time complexity of a program code.
Using Big-O notation we can give longest amount of time taken by the algorithm to complete. e.g. if we have two algorithms having time complexity $O(N)$ and $O(N^2)$ respectively then by varying value of N we can easily check which algorithm is more efficient, i.e. which one takes less time to run.

6. **Distinguish between worst case and best case complexity of an algorithm.** 2011

Ans Difference between worst case and best case complexity of an algorithm:

Worst Case Complexity	Best Case Complexity
It is the function defined by the maximum number of steps taken on any instance of size n.	It is the function defined by the minimum number of steps taken on any instance of size n.
In the worst case analysis, we calculate upper bound on running time of an algorithm.	In the best case analysis, we calculate lower bound on running time of an algorithm.
The worst case time complexity of linear search would be $O(n)$.	The time complexity in the best case of linear search would be $O(1)$.

I. O(l) - Constant Time ... O(1) or O(one)
This means that the algorithm requires the same fixed number of steps regardless of the size of the task.
Examples (assuming a reasonable implementation of the task):
A. Printing the name of a student
B. Pop operation in a stack/queue
II. O(n) - Linear Time
This means that the algorithm requires a number of steps proportional to the size of the task.
Examples (assuming a reasonable implementation of the task):
A. Finding the minimum element in an unsorted list of n elements;
B. Calculating iteratively n-factorial; finding iteratively the nth Fibonacci number
III. O(n^2) - Quadratic Time
The number of operations is proportional to the size of the task squared. Nested loops.
Examples:
A. Some more simplistic sorting algorithms, for example Selection Sort of n elements;
B. Comparing equality of two 2-dimensional arrays of size n x n;
IV. O(log n) - Logarithmic Time
Examples:
A. Binary search in a sorted list of n elements (the method of halving);
B. Insert and Find operations for a binary search tree with n nodes;
V. O(n log n) - "n log n " Time
Examples:
A. More advanced sorting algorithms - quicksort, merge sort (recursive implementation)
VI. O(2^n) - Exponential Time
Examples:
A. Recursive Fibonacci implementation
B. Towers of Hanoi

Comparison of the Big O Notations

- The best time in the above list is constant time
- The worst time is the exponential time
- **Polynomial** growth (linear, quadratic, cubic, etc.) is considered manageable as compared to exponential growth

Efficiency of an algorithm
The three cases to measure the efficiency of an algorithm -

- the best case,
- the worst case, and
- the average case.

The **best** case is how the algorithm will work on the best possible input.
For example – in Linear search on an unsorted list of size n, best case results in O(1). It will be when the element is found in the first position.
The **worst** case is how the algorithm runs on the worst possible input.
For example – in Linear search on an unsorted list of size n, worst case results in O(n). It will be when the element is present in the last position.
And the **average** case is how it runs on most inputs.
For example, to sort 1,000,000 numbers, the Quick sort takes 20,000,000 steps on average, while the Bubble sort takes 1,000,000,000,000 steps!