Unit 1

Time and Space complexity

- Analyzing an algorithm means determining the amount of resources (such as time and memory) needed to execute it.
- Algorithms are generally designed to work with an arbitrary number of inputs, so the efficiency or complexity of an algorithm is stated in terms of time and space complexity.
- The time complexity of an algorithm is basically the running time of a program as a function of the input size.
- Similarly, the space complexity of an algorithm is the amount of computer memory that is required during the program execution as a function of the input size.
- In other words, the number of machine instructions which a program executes is called its time complexity.
- This number is primarily dependent on the size of the program's input and the algorithm used.

- Generally, the space needed by a program depends on the following two parts:
- Fixed part: It varies from problem to problem. It includes the space needed for storing instructions, constants, variables, and structured variables (like arrays and structures).
- Variable part: It varies from program to program.
- It includes the space needed for recursion stack, and for structured variables that are allocated space dynamically during the runtime of a program.
- However, running time requirements are more critical than memory requirements.

Worst-case, Average-case, Best-case, and Amortized Time Complexity

Worst-case running time This denotes the behaviour of an algorithm with respect to the worst-possible case of the input instance. The worst-case running time of an algorithm is an upper bound on the running time for any input. Therefore, having the knowledge of worst-case running time gives us an assurance that the algorithm will never go beyond this time limit.

Average-case running time The average-case running time of an algorithm is an estimate of the running time for an 'average' input. It specifies the expected behaviour of the algorithm when the input is randomly drawn from a given distribution. Average-case running time assumes that all inputs of a given size are equally likely.

Best-case running time The term 'best-case performance' is used to analyse an algorithm under optimal conditions. For example, the best case for a simple linear search on an array occurs when the desired element is the first in the list. However, while developing and choosing an algorithm to solve a problem, we hardly base our decision on the best-case performance. It is always recommended to improve the average performance and the worst-case performance of an algorithm.

Amortized running time Amortized running time refers to the time required to perform a sequence of (related) operations averaged over all the operations performed. Amortized analysis guarantees the average performance of each operation in the worst case.

- Algorithm Efficiency
- If a function is linear (without any loops or recursions), the efficiency of that algorithm or the running time of that algorithm can be given as the number of instructions it contains.
- However, if an algorithm contains loops, then the efficiency of that algorithm may vary depending on the number of loops and the running time of each loop in the algorithm

Linear Loops

To calculate the efficiency of an algorithm that has a single loop, we need to first determine the number of times the statements in the loop will be executed. This is because the number of iterations is directly proportional to the loop factor. Greater the loop factor, more is the number of iterations. For example, consider the loop given below:

```
for(i=0;i<100;i++)
        statement block;
```

Here, 100 is the loop factor. We have already said that efficiency is directly proportional to the number of iterations. Hence, the general formula in the case of linear loops may be given as

```
f(n) = n
```

```
for(i=0;i<100;i+=2)
        statement block;
```

Here, the number of iterations is half the number of the loop factor. So, here the efficiency can be given as

$$f(n) = n/2$$

Logarithmic Loops

We have seen that in linear loops, the loop updation statement either adds or subtracts the loop-controlling variable. However, in logarithmic loops, the loop-controlling variable is either multiplied or divided during each iteration of the loop. For example, look at the loops given below:

```
for(i=1;i<1000;i*=2)
      statement block;
```

```
for(i=1000;i>=1;i/=2)
      statement block;
```

Consider the first `for` loop in which the loop-controlling variable `i` is multiplied by 2. The loop will be executed only 10 times and not 1000 times because in each iteration the value of `i` doubles. Now, consider the second loop in which the loop-controlling variable `i` is divided by 2. In this case also, the loop will be executed 10 times. Thus, the number of iterations is a function of the number by which the loop-controlling variable is divided or multiplied. In the examples discussed, it is 2. That is, when `n = 1000`, the number of iterations can be given by `log 1000` which is approximately equal to 10.

Therefore, putting this analysis in general terms, we can conclude that the efficiency of loops in which iterations divide or multiply the loop-controlling variables can be given as

```
f(n) = log n
```

- Nested Loops
- Loops that contain loops are known as nested loops.
- In order to analyze nested loops, we need to determine the number of iterations each loop completes.
- The total is then obtained as the product of the number of iterations in the inner loop and the number of iterations in the outer loop.

- we analyze the efficiency of the algorithm based on whether it is a
- linear logarithmic, quadratic, or dependent quadratic nested loop.

quadratic, or dependent quadratic nested loop.

Linear logarithmic loop Consider the following code in which the loop-controlling variable of the inner loop is multiplied after each iteration. The number of iterations in the inner loop is `log 10`. This inner loop is controlled by an outer loop which iterates 10 times. Therefore, according to the formula, the number of iterations for this code can be given as `10 log 10`.

```
for(i=0;i<10;i++)
        for(j=1; j<10;j*=2)
                statement block;
```

In more general terms, the efficiency of such loops can be given as `f(n) = n log n`.

Quadratic loop In a quadratic loop, the number of iterations in the inner loop is equal to the number of iterations in the outer loop. Consider the following code in which the outer loop executes 10 times and for each iteration of the outer loop, the inner loop also executes 10 times. Therefore, the efficiency here is 100.

```
for(i=0;i<10;i++)
        for(j=0; j<10;j++)
                statement block;
```

The generalized formula for quadratic loop can be given as $f(n) = n^2$.

Dependent quadratic loop In a dependent quadratic loop, the number of iterations in the inner loop is dependent on the outer loop. Consider the code given below:

```
for(i=0;i<10;i++)
        for(j=0; j<=i;j++)
                statement block;
```

In this code, the inner loop will execute just once in the first iteration, twice in the second iteration, thrice in the third iteration, so on and so forth. In this way, the number of iterations can be calculated as

$$1 + 2 + 3 + \cdots + 9 + 10 = 55$$

If we calculate the average of this loop (55/10 = 5.5), we will observe that it is equal to the number of iterations in the outer loop (10) plus 1 divided by 2. In general terms, the inner loop iterates $(n + 1)/2$ times. Therefore, the efficiency of such a code can be given as

$$f(n) = n\ (n + 1)/2$$

Categories of Algorithms

According to the Big O notation, we have five different categories of algorithms:

- Constant time algorithm: running time complexity given as $O(1)$
- Linear time algorithm: running time complexity given as $O(n)$
- Logarithmic time algorithm: running time complexity given as $O(\log n)$
- Polynomial time algorithm: running time complexity given as $O(n^k)$ where $k > 1$
- Exponential time algorithm: running time complexity given as $O(2^n)$

Table 2.2 shows the number of operations that would be performd for various values of n.

Table 2.2 Number of operations for different functions of n

n	O(1)	O(log n)	O(n)	O(n log n)	$O(n^2)$	$O(n^3)$
1	1	1	1	1	1	1
2	1	1	2	2	4	8
4	1	2	4	8	16	64
8	1	3	8	24	64	512
16	1	4	16	64	256	4096

- Best case O describes an upper bound for all combinations of input. It is possibly lower than the worst case.
- For example, when sorting an array the best case is when the array is already correctly sorted.
- Worst case O describes a lower bound for worst case input combinations. It is possibly greater than the best case.
- For example, when sorting an array the worst case is when the array is sorted in reverse order.

Questions

Multiple-choice Questions

1. Which data structure is defined as a collection of similar data elements?
   (a) Arrays (b) Linked lists
   (c) Trees (d) Graphs
2. The data structure used in hierarchical data model is
   (a) Array (b) Linked list
   (c) Tree (d) Graph
3. In a stack, insertion is done at
   (a) Top (b) Front
   (c) Rear (d) Mid
4. The position in a queue from which an element is deleted is called as
   (a) Top (b) Front
   (c) Rear (d) Mid
5. Which data structure has fixed size?
   (a) Arrays (b) Linked lists
   (c) Trees (d) Graphs
6. If `TOP = MAX-1`, then that the stack is
   (a) Empty (b) Full
   (c) Contains some data (d) None of these
7. Which among the following is a LIFO data structure?
   (a) Stacks (b) Linked lists
   (c) Queues (d) Graphs
8. Which data structure is used to represent complex relationships between the nodes?
   (a) Arrays (b) Linked lists
   (c) Trees (d) Graphs
9. Examples of linear data structures include
   (a) Arrays (b) Stacks
   (c) Queues (d) All of these
10. The running time complexity of a linear time algorithm is given as
    (a) $O(1)$ (b) $O(n)$
    (c) $O(n \log n)$ (d) $O(n^2)$

3. Define data structures. Give some examples.
4. In how many ways can you categorize data structures? Explain each of them.
5. Discuss the applications of data structures.
6. Write a short note on different operations that can be performed on data structures.
7. Compare a linked list with an array.
8. Write a short note on abstract data type.
9. Explain the different types of data structures. Also discuss their merits and demerits.
10. Define an algorithm. Explain its features with the help of suitable examples.
11. Explain and compare the approaches for designing an algorithm.
12. What is modularization? Give its advantages.
13. Write a brief note on trees as a data structure.
14. What do you understand by a graph?
15. Explain the criteria that you will keep in mind while choosing an appropriate algorithm to solve a particular problem.
16. What do you understand by time–space trade-off?
17. What do you understand by the efficiency of an algorithm?
18. How will you express the time complexity of a given algorithm?

19. Discuss the significance and limitations of the Big O notation.
20. Discuss the best case, worst case, average case, and amortized time complexity of an algorithm.
21. Categorize algorithms based on their running time complexity.
22. Give examples of functions that are in Big O notation as well as functions that are not in Big O notation.
23. Explain the little o notation.
24. Give examples of functions that are in little o notation as well as functions that are not in little o notation.
25. Differentiate between Big O and little o notations.
26. Explain the Ω notation.
27. Give examples of functions that are in Ω notation as well as functions that are not in Ω notation.
28. Explain the Θ notation.
29. Give examples of functions that are in Θ notation as well as functions that are not in Θ notation.
30. Explain the ω notation.
31. Give examples of functions that are in ω notation as well as functions that are in ω notation.