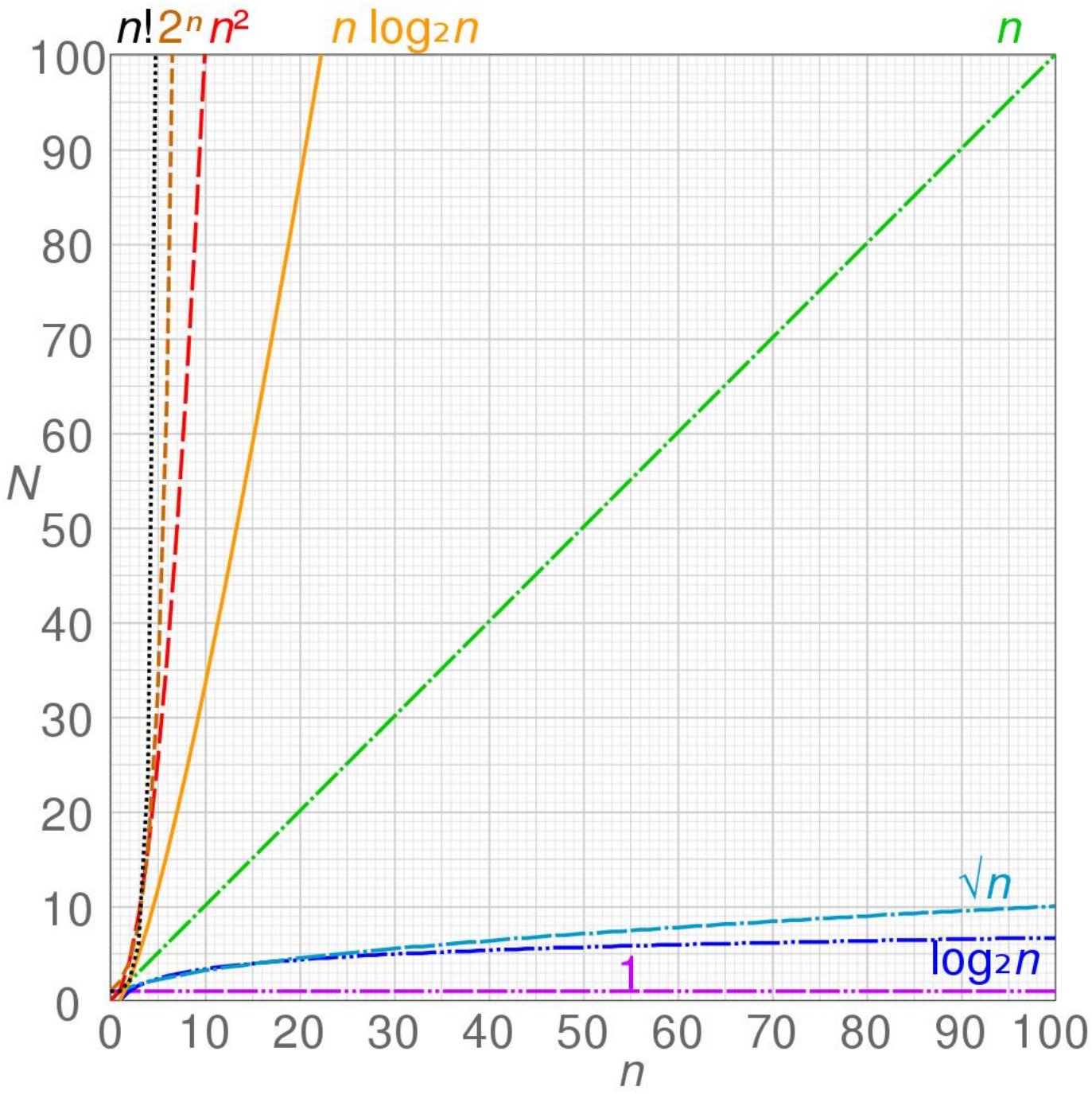
In computer science, the **time complexity** is the computational complexity that describes the amount of computer time it takes to run an algorithm. Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform. Thus, the amount of time taken and the number of elementary operations performed by the algorithm are taken to be related by a constant factor.

Since an algorithm's running time may vary among different inputs of the same size, one commonly considers the worst-case time

complexity, which is the maximum amount of time required for inputs of a given size. Less common, and usually specified explicitly, is the average-case complexity, which is the average of the time taken on inputs of a given size (this makes sense because there are only a finite number of possible inputs of a given size). In both cases, the time complexity is generally expressed as a function of the size of the input. Since this function is generally difficult to compute exactly, and the running time for small inputs is usually not consequential, one commonly focuses on the behavior of the complexity when the input size increases—that is, the asymptotic behavior of the

complexity. Therefore, the time complexity is commonly expressed using big O notation, typically , , , , etc., where *n* is the size in units of bits needed to represent the input.

Algorithmic complexities are classified according to the type of function appearing in the big O notation. For example, an algorithm with time complexity is a *linear time algorithm* and an algorithm with time complexity for some constant is a *polynomial time algorithm*.



An algorithm is said to be **constant time** (also written as time) if the value of is bounded by a value

that does not depend on the size of the input. For example, accessing any single element in an array takes constant time as only one operation has to be performed to locate it. In a similar manner, finding the minimal value in an array sorted in ascending order; it is the first element. However, finding the minimal value in an unordered array is not a constant time operation as scanning over each element in the array is needed in order to determine the minimal value. Hence it is a linear time operation, taking time. If the number of elements is known in advance and does not change, however, such an algorithm can still be said to run in constant time.

Despite the name "constant time", the running time does not have to be independent of the problem size, but an upper bound for the running time has to be independent of the problem size. For example, the task "exchange the values of *a* and *b* if necessary so that is called constant time even though the time may depend on whether or not it is already true that . However, there is some constant *t* such that the time required is always *at most t*.

Here are some examples of code fragments that run in constant time:

```
int index = 5;
int item = list[index];
```

```
if (condition true) then
    perform some operation that
runs in constant time
else
    perform some other operation
that runs in constant time
for i = 1 to 100
    for j = 1 to 200
        perform some operation that
runs in constant time
```

Time complexity measures the time taken to execute each statement of code in an algorithm. If a statement is set to execute repeatedly then the number of times that statement gets executed is equal to N multiplied

by the time required to run that function each time.

For example, look at the code below:

```
%%time
print("This code is to demonstrate Time complexity!") # Only once this statement is executed (N = 1)
```

```
This code is to demonstrate Time complexity!
Wall time: 1 ms
```

```
%%time
for i in range(10):       # 10 times this statement is executed (N=10)
    print("This code is to demonstrate Time complexity!") # 10 times this statement is executed (N=10)
```

```
This code is to demonstrate Time complexity!
This code is to demonstrate Time complexity!
This code is to demonstrate Time complexity!
This code is to demonstrate Time complexity!
This code is to demonstrate Time complexity!
This code is to demonstrate Time complexity!
This code is to demonstrate Time complexity!
This code is to demonstrate Time complexity!
This code is to demonstrate Time complexity!
This code is to demonstrate Time complexity!
Wall time: 999 µs
```

The first algorithm is defined to print the statement only once. The

time taken to execute is shown as **0 nanoseconds**. While the second algorithm is defined to print the same statement but this time it is set to run the same statement in FOR loop 10 times. In the second algorithm, the time taken to execute both the line of code – FOR loop and print statement, is **2 milliseconds**. And, the time taken increases, as the N value increases, since the statement is going to get executed N times.

By now, you could have concluded that when an algorithm uses statements that get executed only once, will always require the same amount of time, and when the

statement is in loop condition, the time required increases depending on the number of times the loop is set to run. And, when an algorithm has a combination of both single executed statements and LOOP statements or with nested LOOP statements, the time increases proportionately, based on the number of times each statement gets executed.

This leads us to ask the next question, about how to determine the relationship between the input and time, given a statement in an algorithm. To define this, we are going to see how each statement gets an order of notation to describe

time complexity, which is called Big **O Notation**.

What are the Different Types of Time complexity Notation Used?

As we have seen, Time complexity is given by time as a function of the length of the input. And, there exists a relation between the input data size (n) and the number of operations performed (N) with respect to time. This relation is denoted as Order of growth in Time complexity and given notation O[n] where O is the order of growth and n is the length of the input. It is also called as **‘Big O Notation’**

Big O Notation expresses the run time of an algorithm in terms of how

quickly it grows relative to the input ‘n’ by defining the N number of operations that are done on it. Thus, the time complexity of an algorithm is denoted by the combination of all O[n] assigned for each line of function.

There are different types of time complexities used, let’s see one by one:

1. Constant time – O (1)

2. Linear time – O (n)

3. Logarithmic time – O (log n)

4. Quadratic time – O (n^2)

5. Cubic time – O (n^3)

and many more complex notations like **Exponential time, Quasilinear time, factorial time, etc.** are used based on the type of functions defined.

Linear time

An algorithm is said to take **linear time**, or time, if its time complexity is . Informally, this means that the running time increases at most linearly with the size of the input. More precisely, this means that there is a constant *c* such that the running time is at most for every input of size *n*. For example, a procedure that adds up all elements of a list

requires time proportional to the length of the list, if the adding time is constant, or, at least, bounded by a constant.

Linear time is the best possible time complexity in situations where the algorithm has to sequentially read its entire input. Therefore, much research has been invested into discovering algorithms exhibiting linear time or, at least, nearly linear time. This research includes both software and hardware methods. There are several hardware technologies which exploit parallelism to provide this. An example is content-addressable memory. This

concept of linear time is used in string matching algorithms such as the Boyer-Moore algorithm and Ukkonen's algorithm.

Logarithmic time – O (log n)

An algorithm is said to have a logarithmic time complexity when it reduces the size of the input data in each step. This indicates that the number of operations is not the same as the input size. The number of operations gets reduced as

the input size increases. Algorithms with Logarithmic time complexity are found in binary trees or binary search functions. This involves the search of a given value in an array by splitting the array into two and starting searching in one split. This ensures the operation is not done on every element of the data.

Quadratic time – O (n^2)

An algorithm is said to have a non-linear time complexity where the running time increases non-linearly (n^2) with the length of the input. Generally, nested loops come under this time complexity order where one loop takes O(n) and if the function involves a loop within a loop, then it goes for O(n)*O(n) = O(n^2) order.

Similarly, if there are ‘m’ loops defined in the function, then the order is given by O

(n ^ m), which are called **polynomial time complexity** functions.

How to evaluate an algorithm for Time complexity?

We have seen how the order notation is given to each function and the relation between runtime vs no of operations, input size. Now, it is time to know how to evaluate the Time complexity

of an algorithm based on the order notation it gets for each operation & input size and compute the total run time required to run an algorithm for a given n.

Let us illustrate how to evaluate the time complexity of an algorithm with an example:

The algorithm is defined as:

1. Given 2 input matrix, which is a square matrix with order n

2. The values of each element in both the matrices are selected randomly using np.random function

3. Initially assigned a result matrix with 0 values of order equal to the order of the input matrix

4. Each element of X is multiplied by every element of Y and the resultant value is stored in the result matrix

5. The resulting matrix is then converted to list type

6. For every element in the result list, is added together to give the final answer

Let us assume cost function C as per unit time taken to run a function while ‘n’ represents the number of times the statement is defined to run in an algorithm.

For example, if the time taken to run print function is say 1 microseconds (C) and if the algorithm is defined to run

PRINT function for 1000 times (n),