COMPUTER SCIENCE With python

Textbook for Class XII

- Programming & Computational Thinking
- Computer Networks
- Data Management (SQL, Django)
- Society, Law and Ethics

SUMITA ARORA

DHANPAT RAI & Co.

7 Idea of Algorithmic Efficiency

233 – 250

7 Idea of Algorithmic Efficiency

7.1 INTRODUCTION

An algorithm is a method or procedure for accomplishing a specific task, and which is sufficiently precise and that can be programmed on computer. In Computer Science, it is important to measure **efficiency of algorithms** before applying them on a large scale *i.e.*, on bulk of data. The quality of an algorithm or shall we say, performance of an algorithm depends on many *internal* and *external* factors.

Internal Factors specify algorithm's efficiency in terms of :

- *Time* required to run
- *Space* (or Memory) required to run.

External Factors affect the algorithm's performance. These include :

- *Size of the input* to the algorithm
- *Speed of the computer* on which it is run
- *Quality of the compiler*.

Since, external factors are controllable to some extent, mainly *internal factors* are studied and measured in order to determine an algorithm's efficiency or we can say *complexity*. *Complexity* of an algorithm is determined by studying and measuring internal factors affecting the algorithm.

This chapter is going to introduce how you can determine or measure efficiency of an algorithm in terms of computational complexity. Let us begin our discussion with *'What is computational complexity ?'*

7.2 WHAT IS COMPUTATIONAL COMPLEXITY ?

The term '*Computational complexity*' is made of *two* words : '*Computation*' and '*Complexity*'.

The **Computation** involves the *problems* to be solved and the algorithms to solve them. **Complexity** involves the study of factors to determine how much *resource* is sufficient/ necessary for this algorithm to run efficiently (performance).

COMPLEXITY

Complexity refers to the measure of the efficiency of an algorithm.

The *resources* generally include :

- The *time* to run the algorithm (*Temporal complexity*).
- The space (or the memory/storage) needed to run the algorithm (*Space Complexity*)[1].

The first thing to take into account is the difference between **efficiency** and **effectiveness**. ***Effectiveness*** means that the algorithm carries out its intended function *correctly*. But *efficiency* means the algorithm should be *correct with the best possible performance*. And to measure *efficiency*, we determine complexity. Complexity of an algorithm quantifies the resources needed as a function of the amount of data processed.

Complexity is not the absolute measure, but rather a bounding function characterizing the behaviour of the algorithm as the size of the data set increases. It allows the comparison of algorithm for efficiency, and predicts their behaviour as data size increases.

7.3 ESTIMATING COMPLEXITY OF ALGORITHMS

As mentioned, algorithms are usually compared along *two* dimensions : ***amount of space*** (that is memory) used and the ***time taken***. Of the two, the *time taken* is usually considered the more important. The motivation to study time complexity is to compare different algorithms and use the one that is the most efficient in a particular situation.

Actual run time on a particular computer (external factors) is not a good basis for comparison since it depends heavily on the speed of the computer, the total amount of RAM in the computer, the OS running on the system and the quality of the compiler used. So we need a more abstract way to compare the time complexity of algorithms.

Informally, we can define *time complexity* of a program (for a given input) is the number of elementary instructions that this program executes. This number is computed with respect to the size n of the input data.

7.3.1 Big-O Notation

The Big-O notation is used to depict an algorithm's growth rate. ***The growth rate determines the algorithm's performance when its input size grows***. Through big-O, the upper bound of an algorithm's performance is specified *e.g.*, if we say an algorithm takes $O(n^2)$ time ; this means that this algorithm will carry out its task taking **at the most N^2 steps for input size N.**

The Big-O notation is very useful for comparing the performance of two or more algorithms. For instance, if we say that we have two algorithms for solving a problem ; one has complexity $O(n^2)$, and the other has complexity $O(2^n)$. We can now, compare the number of steps needed to solve the problem for different sizes as depicted in following table :

1. **These days, for web specific algorithms, *Network Traffic* may also be considered.**

Comparing Number of Steps for $O(n^2)$ and $O(2^n)$ Algorithms

Complexity \ Size N	10	20	40	100	400
N^2	100	400	1600	10000	160000
2^n	1024	1048576	10^{12}	1.26×10^{30}	Very Big...

More the number of steps, more is the time taken and lesser is the performance. As for $N = 100$, (from the above table) the time taken by an algorithm with $O(2^n)$ is 1.26×10^{30} compared to algorithm with $O(N^2)$ which is 10000. As 1.26×10^{30} is way more than 10000 *i.e.*, 10^5. Thus, you can say that performance of algorithms is inversely proportional to the wall clock time it records for a given input size. Size $O(2^n)$'s $1.26 \times 10^{30} >> O(N^2)$'s 10^5. $O(N^2)$ is better algorithm than $O(2^n)$ algorithm as it clocks lesser time comparatively. It other words, we can say that the algorithm with complexity $O(n^2)$ is better than the algorithm with complexity $O(2^n)$ for solving the same problem.

NOTE

Performance of algorithms is inversely proportional to the wall clock time it records for a given input size. Programs with a bigger *O* run slower than programs with a smaller *O*.

Dominant Term

Big-O notation indicates the growth rate. It is the class of mathematical formula that best describes an algorithm's performance, and is discovered by looking inside the algorithm.

Big-*O* is a function with parameter *N*, where *N* is usually the *size of the input* to the algorithm. More the input size, more impact it can have on the growth rate of the algorithm. However, while describing the growth rate of an algorithm, we simply consider the term, which is going to affect the most on the algorithm's performance. This term is known as the **dominant term.**

For example, if an algorithm depending on the value n has performance $an^2 + bn + c$ (for constants a, b, c) then we can say that the maximum impact on the algorithm's performance will be of the term an^2. So while describing the complexity of the algorithm, we can ignore the rest of the terms and simply say that the algorithm has performance $O(N^2)$. In other words, for large N, the N^2 term dominates. Only the dominant term is included in big-*O*.

Common Growth Rates

Some Growth rates of algorithms are as shown in following table :

Time complexity		Example
O(1)	*constant*	Adding to the front of a list
O(log *N*)	*log*	Finding an entry in a sorted array
O(*N*)	*linear*	Finding an entry in an unsorted array
O(*N* log *N*)	*n log n*	Sorting *n* items by 'divide-and-conquer'
$O(N^2)$	*quadratic*	Shortest path between two nodes in a graph
$O(N^3)$	*cubic*	Simultaneous linear equations
$O(2^N)$	*exponential*	The Towers of Hanoi problem

7.3.2 Guidelines for Computing Complexity

After talking about some basic terms, let us now discuss how you can actually compute complexity of an algorithm.

The basic steps for computing complexity of an algorithm are :

1. Select the computational resource you want to measure. Normally we have two options : *time* and *memory*. But other studies can be undertaken like, *e.g.*, network traffic. But here, we are mainly interested in measuring time complexity.
2. Look to the algorithm and pay attention to loops or recursion. Try to see the variables and conditions that make the algorithm work more or less. Sometimes it's one variable, some times several. This is going to be our **size of input.** Remember that with complexity analysis, we are interested in getting a function that relates the size of input with the computational resource.
3. Once we have the *size of input* of the algorithm, try to see if there are different cases inside it, such as when the algorithm gives best performance *i.e.*, takes shortest possible time (*best case*); when the algorithm gives worst performance *i.e.*, takes maximum possible time (*worst case*); and when the algorithm performs in between the two cases *i.e.*, performs better than the worst case but does not give best performance (*average case*).

Calculating Complexity

Five guidelines for finding out the time complexity of a piece of code are: (**Assumption:** All steps take precisely same time to execute.)

- Loops
- Nested loops
- Consecutive statements
- If-then-else statements
- Logarithmic complexity

1. Loops

The running time of a loop is, at most, equal to the running time of the statements inside the loop (including tests) multiplied by the number of iterations. *For instance,* consider the following loop :

Loop executed n times →

```
for i in range(n) :
    m = m + 2
```

← *All the steps in this loop take constant time, say* ***c***

So, the total time taken by the above loop to execute itself is :

$$\text{Total time} = c * n = cn \quad i.e., \quad O(n)$$

- c : Time taken by one step
- n : No of steps in the loop
- cn : Total Time taken by the loop to execute all its steps

In a loop executing N times, the body of the loop gets executed $N-1$ times, the condition evaluates to *false* and loop terminates without executing the body.

2. Nested loops

To compute complexity of nested loops, analyze inside out. For nested loops, total running time is the product of the sizes of all the loops.

For instance, consider the following code :

```
for i in range(n) :
    for j in range(n):
        K = K + 1
```

Outer loop executes n times → `for i in range(n) :`

Inner loop executes n times → `for j in range(n):`

All these steps take constant time, say c → `K = K + 1`

So the total time taken by the above shown nested loops to execute is :

$$\text{Total time} = c * n * n = cn^2 \quad \text{i.e.,} \quad O(n^2)$$

- Time taken by one step
- No of steps in the outer loop
- No of steps in the inner loop
- Total Time taken by the nested loops to execute all steps

3. Consecutive statements

To compute the complexity of consecutive statements, simply add the time complexities of each statement. For instance,

```
x = x + 1
for i in range (n):
    m = m + 2
for j in range (n) :
    for k in range (n):
        a = a + 1
```

- `x = x + 1` — *takes constant time, say c_0*
- `for i in range (n):` — *loop1 executes n times*
- `m = m + 2` — *takes constant times say c_1*
- *Outer loop executes n times* → `for j in range (n) :`
- `for k in range (n):` — *inner loop executes n times*
- `a = a + 1` — *takes constant time, say c_2*

So the total time taken by the above shown code to execute is :

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

- Time taken by statement1
- Time taken by loop1
- Time taken by nested loop
- Considering only the dominant term which is n^2

NOTE

Simple programs can be analyzed by counting the nested loops of the program. A single loop over n items yields $f(n) = n$. A loop within a loop yields $f(n) = n^2$. A loop within another loop inside a third loop yields $f(n) = n^3$.

4. If-then-else statements

To compute time complexity of if-then-else statement, we consider worst-case running time, which means, we consider ***the time taken by the test, plus time taken by either the then part or the else part*** (*whichever is larger*).

For instance, consider the following code :

```
if len(list1) != len(list2) :
    return False
else :
    for i in range(n) :
        if list1[i] != list2[i]:
            return False
```

Comparison test takes constant time say c_0

The true part's step takes constant time, say c_1

Test condition takes constant time say c_2

Loop repeats n times

Takes constant time say c_2

The else part has a loop (n times) which in total takes times as $(c_2 + c_3) * n$

So the total time taken by the above shown code to execute is :

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

Time taken by *test*

Time taken by then part

Time taken by else part

Considering only the dominant term which is N.

5. Logarithmic Complexity

The logarithmic complexity means that an algorithms' performance time has logarithmic factor *e.g.* an algorithm is **O(log *N*)** if it takes a constant time to cut the problem size by a fraction (usually by ½).

An example algorithm having logarithmic complexity is *binary search* algorithm.

But, the best performance isn't always desirable. There can be a tradeoff between :

- Ease of understanding, writing and debugging
- Efficient use of time and space

So, maximum performance is not always desirable. However, it is still useful to compare the performance of different algorithms, even if the optimal algorithm may not be adopted.

NOTE

Given a series of for loops that are sequential, the slowest of them determines the asymptotic behavior of the program. 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.

Some Useful Rules for Algorithm Analysis

To analyze a function analyze each statement in the function. The statement that has the greatest complexity determines the order of the function. Some helpful rules are being given for your reference :

1. Simple statements such as assignment statements are $O(1)$, since their execution time is not dependent on the amount of data. The exception would be assigning one array's elements to another which would be $O(n)$, where n represents the size of the array.
2. The running time for a function call is just the running time for the function's body, plus the time to set up the parameters. Setting up the parameters is $O(1)$, unless a large structure is passed as a value parameter, in which case that structure must be copied into the parameter which is $O(n)$.
3. "In an if/else statement, the test (which is usually $O(1)$) and one of the conditional statements are executed. Since you generally cannot determine the result of the test ahead of time, you should assume the worst case, *i.e.*, the *maximum time* and thus the running time of the if/else will be the sum of the running time of the test and the running time of the worst-case statement."
4. **Addition Rule** "*You can combine sequences of statements by using the addition rule, which states that the running time of a sequence of statements is just the maximum of the running times of each individual statement.*"
5. **Multiplication Rule** This is used to determine the running time for a loop. "*You need to determine the running time for the loop's body and the number of times the loop iterates. Frequently you can just multiply the running time of the body by the number of iterations.*"

7.4 BEST, AVERAGE AND WORST CASE COMPLEXITY

We have discussed in the previous section that we can count how many steps our algorithm will take on any given input instance by simply executing it on the given input. However, to really understand how good or bad an algorithm is, we must know how it works over *all* instances, which means :

- the best possible performance of the algorithm (best case)
- the worst possible performance of the algorithm (worst case)
- the average performance of the algorithm (average case)

To understand the notions of the *best, worst,* and *average-case complexity,* one must think about running an algorithm on all possible instances of data that can be fed to it. For the problem of sorting, the set of possible input instances consists of all the possible arrangements of all the possible numbers of keys. We can represent every input instance as a point on a graph, where the x-axis is the *size of the problem* (for sorting, the number of items to sort) and the y-axis is the *number of steps taken* by the algorithm on this instance. Here we assume, quite reasonably, that it

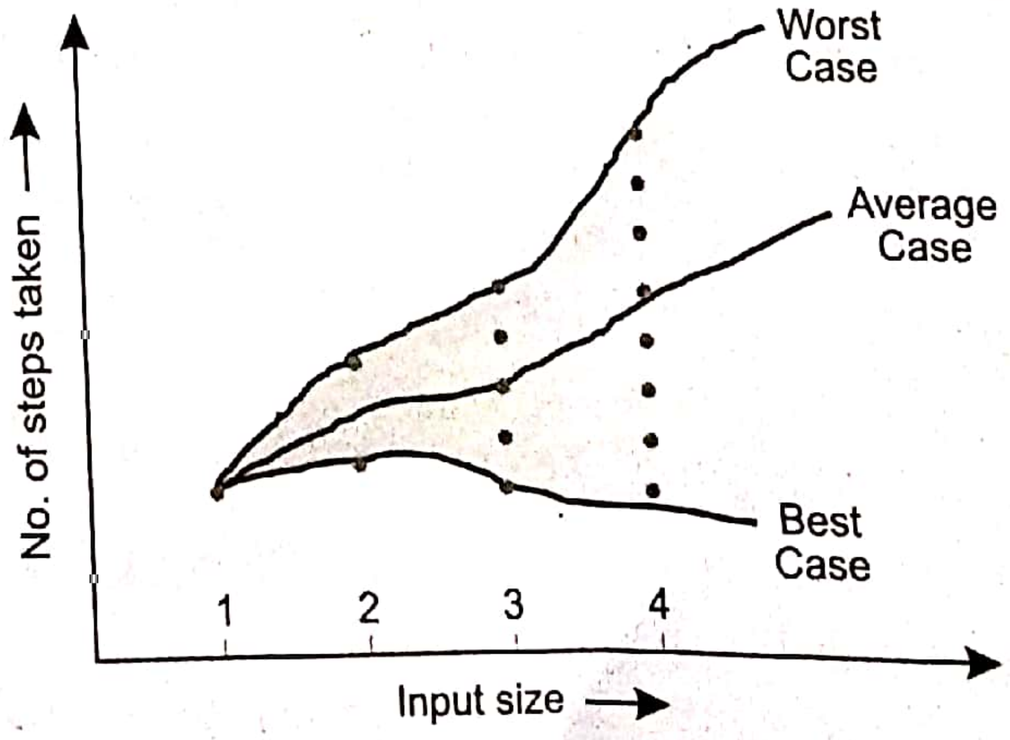

Figure 7.1 Best, worst, and average-case complexity.

doesn't matter what the values of the keys are, just how many of them there are and how they are ordered. It should not take longer to sort 1,000 English names than it does to sort 1,000 French names, for example.

Once we have these points, we can define *three* different functions over them :

- The *worst-case complexity* of the algorithm is the function defined by the maximum number of steps taken on any instance of size *n*. It represents the curve passing through the highest point of each column.
- The *best-case complexity* of the algorithm is the function defined by the minimum number of steps taken on any instance of size *n*. It represents the curve passing through the lowest point of each column.
- Finally, the *average-case complexity* of the algorithm is the function defined by the average number of steps taken on any instance of size *n*.

In practice, the most useful of these three measures proves to be the *worst-case complexity*, which many people find counterintuitive. *Worst case*

- Provides an upper bound on running time of an algorithm.
- An absolute guarantee that no matter what, this algorithm won't take time more than this time.

Let us understand the efficiency calculation in terms of complexity by taking up two different programs for the same problem.

Example 7.1 *Determine the complexity of a program that checks if a number n is prime.*

Solution.

Option 1 : (Linear approach for prime test)

```
# check whether n is a prime or not using the linear approach
n = int(input("Enter a number :"))
```
← *Constant time c_0*

```
# prime flag 0 indicates that n is a prime and the value 1 indicates that it is not a prime
prime_flag = 0
```
← *Constant time c_1*

repeats nearly n times

```
for i in range(2, n):
    if n % i == 0:      # if i is a factor of n, n is not a prime
        prime_flag = 1
        break
```
← *Constant time c_2* (for the `if`)

← *Constant time c_3* (for `prime_flag = 1`)

```
if not prime_flag :
    print(n, "is a prime number")
else:
    print(n, "is not a prime number")
```
← *Constant time c_4 (checks if prime_flag is true)*

← *Constant time c_5*

← *Constant time c_6*

Total time taken by option 1 (the linear approach)

$$= c_0 + c_1 + (c_2 + c_3)n + (c_4 + c_5 + c_6)$$

$$= C_0 + C_1 n \qquad (C_0 = c_0 + c_1 + c_4 + c_5 + c_6,\ C_1 = c_2 + c_3)$$

$= O(n)$ considering the dominant term, which is n.

Option 2 : ($\sqrt{N}$ approach for prime test)

```
#check whether n is a prime or not using the sqrt n approach
n = int(input("Enter a number :"))

#check whether n is a prime or not using the sqrt n approach
prime_flag = 0
i = 2
while (i * i <= n):      # notice that if n does not have a factor less that sqrt n, it will
                         # never have a factor between 2 and n - 1
    if n % i == 0:
        prime_flag = 1
        break
    i = i + 1

if not prime_flag :
    print(n, "is a prime number")
else:
    print(n, "is not a prime number")
```

Constant time c_0

Constant time c_1

Constant time c_2

Condition takes constant time c_3

Loop repeats $\sqrt{N}$ times

Constant time c_4

Constant time c_5

Constant time c_6

Constant time c_7

Constant time c_8

Constant time c_9

Constant time c_{10}

Total time taken for option 2 ($\sqrt{N}$ approach)

$$= c_0 + c_1 + c_2 + \sqrt{N}(c_3 + c_4 + c_5 + c_6 + c_7) + c_8 + c_9 + c_{10}$$

$$= C_1 + \sqrt{N}C_2$$

$$= C_1 + C_2\sqrt{N} \qquad [C_1 = c_0 + c_1 + c_2 + c_8 + c_4 + c_{10},\ C_2 = c_3 + c_4 + c_5 + c_6 + c_7 + c_8]$$

$= O(\sqrt{N})$ Considering the dominant term.

Comparing complexity of option 1 and option 2, we can say that option 2 is better as $\sqrt{N} < N$.

Example 7.2 *Determine the complexity of a program thata searches for an element in an array.*

Solution.

Option 1 : (Linear search)

```
def linearSearch(arr, x) :
    i = 0
    n = len(arr)
```

Constant time c_0

```
while( i < n  and  x != arr[i] ):
    i = i + 1
if i < n :
    return i          # Element x found at index i
else:
    return None       # element x not found
```

Constant time c_2 (`while( i < n and x != arr[i] ):`)
Constant time c_1 (`i < n`)
Loop repeats maximum n times (`while` block)
Constant time c_3 (`i = i + 1`)
Constant time c_4 (`if i < n :`)
Constant time c_5 (`return i  # Element x found at index i`)
Constant time c_6 (`return None  # element x not found`)

Total time taken by option 1 (the linear search)

$$\text{Total time taken} = c_0 + n(c_1 + c_2 + c_3) + c_4 + c_5 + c_6$$
$$= c_0 + c_4 + c_5 + c_6 + n(c_1 + c_2 + c_3)$$
$$= C_0 + nC_1 \qquad [C_0 = c_0 + c_4 + c_5 + c_6,\ C_1 = c_1 + c_2 + c_3]$$
$$= O(n) \text{ considering the dominant term.}$$

Option 2 : (binary search)

```
def binsearch(ar, key) :
    low = 0                  # initialy low end is at 0
    high = len(ar) - 1       # initialy high end is at size - 1
    while  low <= high :
        mid = int((low + high) / 2)
        if key == ar[mid] :
            return mid
        elif key < ar[mid] :
            high = mid - 1
        else:
            low = mid + 1
    else :                   # loop's else; reaches here when key not matched
        return None
```

Constant time c_0 (`def binsearch(ar, key) :`)
Constant time c_1 (`low = 0`, `high = len(ar) - 1`)
Constant time c_2 (`while low <= high :`)
Loop repeats max $\log_2 N$ times because every time segment becomes half in size (explanation below) (`while` block)
Constant time c_3 (`mid = int((low + high) / 2)`)
Constant time c_4 (`if key == ar[mid] :`)
Constant time c_5 (`return mid`)
Constant time c_6 (`elif key < ar[mid] :`)
Constant time c_7 (`high = mid - 1`)
Constant time c_8 (`low = mid + 1`)
Constant time c_9 (`return None`)

Total time taken by option 2 (the binary search)

$$\text{Total time taken} = c_0 + c_1 + \log_2 n\,(c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8) + c_9$$
$$= C_0 + \log_2 nC_1 \qquad (c_0 + c_1 + c_9 = c_0 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8 = C_1)$$
$$= O(\log_2 n) \text{ considering the dominant term.}$$

How Many Times above While Loop Executes

The while loop of above given binary search algorithm executes how many times ? To determine this you need to answer the question :

How many times can you divide N by 2 until you have 1 ? This is because in binary search, the search segment's size begins with N and reduces by half in every iteration and stops when the search segment's size reduces to 1 element.

So if loop repeats k times (until the segment reduces to size 1), then in a formula this would be :

$$N / 2^k = 1$$
$$2^k = N$$

Taking the $\log_2$ on both sides :

$$\log_2(2^k) = \log_2 N$$
$$k * \log_2(2) = \log_2 N$$
$$k * 1 = \log_2 N$$
$$k = \log_2 N$$

This means you can divide $\log N$ *times* until you have everything divided. That means the above loop repeats at max **$\log N$ times.**

Comparing complexity of ***option 1*** and ***option 2***, we can say that **option 2 is better as $\log_2 N < N$.**

With this, we have come to the end of our chapter. Let us quickly revise what we have learnt so far.

Let us Revise

- *An algorithm is a sufficiently precise method or procedure for accomplishing a specific task, which can be programmed on computer.*
- *Complexity refers to the measure of the performance of an algorithm.*
- *Complexity can be related to time (temporal complexity) or to space (space complexity).*
- *Big-O notation is used to depict an algorithm's growth rate i.e., change in algorithm performance when its input size grows.*
- *Dominant term is the one which affects the most, an algorithm's performance.*
- *Only the dominant term is included in Big-O notation.*
- *The Worst case complexity provides an upper-bound on running time.*
- *Average-Case complexity provides expected running time.*
- *Best-Case complexity provides the time of optimal performance.*

Solved Problems

1. *Define Big 'O' notation. State the two factors which determine the complexity of an algorithm.*

Solution. Big O notation is a particular tool for assessing algorithm efficiency. It describes the performance or complexity of an algorithm denoted *via* Big O, *e.g.*, $O(1)$, $O(N)$, $O(N \log N)$ etc.

Performance of an algorithm depends on many *internal* and *external* factors.

Internal Factors specify algorithm's efficiency in terms of :

- *Time* required to run
- *Space* (or Memory) required to run.

External Factors affect the algorithm's performance. These include :

- *Size of the input* to the algorithm
- *Speed of the computer* on which it is run
- *Quality of the compiler.*

2. *Consider the following three algorithms for determining whether anyone in the room has the same birthday as you.*

Algorithm 1 : You say your birthday, and ask whether anyone in the room has the same birthday. If anyone does have the same birthday, they answer yes.

Algorithm 2 : You tell the second person your birthday and ask whether they have the same birthday ; and so forth, for each person in the room.

Algorithm 3 : You only ask questions of person 1, who only asks questions of person 2, who only asks questions of person 3, etc. You tell person 1 your birthday, and ask if they have the same birthday; if they say no, you ask them to find out about person 2. Person 1 asks person 2 and tells you the answer. If it is no, you ask person 1 to find out about person 3. Person 1 asks person 2 to find out about person 3, etc.

(*i*) *What is the factor that can affect the number of questions asked (the "problem size") ?*

(*ii*) *In the worst case, how many questions will be asked for each of the three algorithms ?*

(*iii*) *For each algorithm, say whether it is constant, linear, or quadratic in the problem size in the worst case.*

Solution.

(*i*) The problem size is the number of people in the room.

(*ii*) Assume there are N people in the room. In algorithm 1 you always ask 1 question. In algorithm 2, the worst case is if no one has your birthday. Here you have to ask every person to figure this out. This means you have to ask N questions. In algorithm 3, the worst case is the same as algorithm 2. The number of questions is 1 + 2 + 3 + ... + N – 1 + N. This sum is N(N+1)/2, using formula for sum of first N natural numbers.

(*iii*) Given the number of questions you can see that algorithm 1 is constant time, algorithm 2 is linear time, and algorithm 3 is quadratic time in the problem size.

3. *Distinguish between worst-case and best case complexity of an algorithm.*

Solution. The *worst-case complexity* of the algorithm is the function defined by the maximum number of steps taken on any instance of size *n*. It represents the curve passing through the highest point of each column.

The *best-case complexity* of the algorithm is the function defined by the minimum number of steps taken on any instance of size *n*. It represents the curve passing through the lowest point of each column.

4. *(i) Give the meaning of the following common expression in Big O notation :*

$$O(N) ; \quad O(N^2)$$

(ii) List any two cases to analyse algorithm complexities.

Solution. (*i*) **O(N)** describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set.

$O(N^2)$ represents an algorithm whose performance is directly proportional to the square of the size of the input data set. This is common with algorithms that involve nested iterations over the data set.

(ii) The efficiency of an algorithm is determined through algorithm complexity. Two common cases to analyse algorithm complexities are :

- Sorting algorithms
- Searching Algorithms

5. *What is the worst case complexity of the following code fragment ?*

```
1.  for i in range(n) :
2.          a = i + (i + 1)
3.          print(a)
4.  for j in range(m):
5.          b = i * (i + 1)
6.          print(b)
```

Solution. Time taken in the execution of given code :

loop1 repeats n times

```
for i in range(n) :
    a = i + (i + 1)
    print(a)
```

takes constant time c_0

takes constant time c_1

loop 2 repeats m times

```
for j in range(m) :
    b = i * (i + 1)
    print(b)
```

takes constant time c_2

takes constant time c_3

Total time taken $= n*(c_0 + c_1) + m*(c_2 + c_3)$

$= nC_1 + mC_2 \quad [C_1 = c_0 + c_1,\ C_2 = c_2 + c_3]$

$= O(n+m)$

6. *What is the worst-case complexity of the following code fragment having a nested loop followed by a single loop :*

```
for i in range(n) :
    for j in range(n):
        sequence of statements
for k in range(n):
        sequence of statements
```

Solution. Time taken in the execution of given code :

Outer loop repeats n times

Inner loop repeats n times

```
for i in range(n) :
    for j in range(n) :
        sequence of statements
```

Take up constant time say C_0

Loop 3 repeats n times

```
for k in range(n) :
    sequence of statements
```

Take up constant time say C_1

Total time taken $= n*(n + C_0) + nC_1$

$= C_0 n^2 + C_1 n$

$= O(n^2)$ considering the dominant term only.

7. *(a) What is the worst case complexity of the following code fragment ?*

```
for x in range(a):
    statements

for y in range(b):
    for z in range(c):
        statements
```

(b) How would the complexity change if all the three loops repeated N times instead of a, b and c times respectively ?

Solution. Assuming that each statement has O(1) complexity, then : (*a*) $O(a+bc)$ (*b*) $O(N^2)$

8. *Compute the complexity of following algorithm (it is not written in any programming language) :*

```
# Finding the largest and second largest elements of a list
# 1.  Find the largest element by scanning from left to right
# 2.  Go back through the list and find the second largest element.

big1(array a of  integers) :
    max1 ← a1
    for i ← 2 to n do
        if ai > max1 then max1 ← ai
            if max1 = a1 then max2 ← a2
            else max2 ← a1
            for i ← 2 to n do
                if (ai > max2 and ai ≠ max1) then
                    max2 ← ai
    return(max1, max2)
```

Solution. Analysing the given code.

```
max1 ← a1                                                    1 assignment
for i ← 2 to n do                                            n – 1 iterations
    if ai > max1 then max1 ← ai                              1 assignment and one comparison
        if max1 = a1 then max2 ← a2                          1 assignment and one comparison
        else max2 ← a1
        for i ← 2 to n do                                    n – 1 iterations
            if (a1 > max2 and ai ≠ max1) then                2 comparisons and 1 assignment each time
                max2 ← ai                                    Total for loop 2(n – 1) comparisons and
                                                             at most n – 1 assignments
return(max1, max2)                                           O(1)–constant complexity
```

This gives us the following totals :

Total number of comparisons		Number of assignments
n – 1	first for loop	1 + n - 1 + 1 + n – 1 = 2n
1	if statement	
2n – 2	second for loop	
3n - 2		

For the first for loop, we have $\sum_{i=2}^{n} 1 = n-1$ and in the second we have $\sum_{i=2}^{n} 2 = 2(n-1)$

That is $n-1+2(n-1)=3(n-1)=O(n)$.

9. *(i) Reorder the following efficiencies from the smallest to the largest :*

(a) 2^n (b) n! (c) n^5 (d) 10,000 (e) $n\log_2(n)$

(ii) Reorder the following efficiencies from the smallest to the largest :

(a) $n\log_2(n)$ (b) $n+n^2+n^3$ (c) 2^4 (d) $n^{0.5}$

Solution.

(a) $10{,}000 < n\log_2(n) < n^5 < 2^n < n!$

(b) $2^4 < n^{0.5} < n\log_2(n) < n+n^2+n^3$

10. *Calculate the run-time efficiency of the following program segment :*

```
i = 1
while i <= n :
    j = 1
    while j <= n :
        k = 1
        while k <= n :
            print(i, j, k)
            k = k + 1
        j = j + 1
    i = i + 1
```

Solution. There are *three* nested loops, each loop is executed in n times, so run-time efficiency is

$$n \times n \times n = n^3$$

11. *Calculate the run time efficiency of the following program segments.*

(a) If the function **doIt** *has an efficiency factor of 5n.*

```
i = 1
while i <= n :
    doIt(...)
    i = i + 1
```

(b) If the efficiency of the algorithm **doIt** *can be expressed as* $O(n) = n^2$.

```
i = 1
while i <= n :
    j = 1
    while j < n :
        doIt(...)
        j = j + 1
    i = i + 1
```

(c) If the efficiency of the algorithm **doIt** *can be expressed as* $O(n) = n^2$.

```
i = 1
while i < n :
    doIt(...)
    i = i * 2
```

Solution.

(a) Function **doIt** is executed in n times, so the run-time efficiency is $5n^2$ *i.e.*, $O(n^2)$.

(b) The iteration of variable i is executed in n times while the iteration of variable j is executed in $n-1$ times. Therefore, the run-time efficiency is

$$n \times (n-1) \times n^2 = n^4 - n^3 = O(n^4)$$

(c) The function *doIt* is executed inside a logarithmic loop (notice i is updated as $i = i*2$ *i.e.*, i doubles every time), with size n, so the run time efficiency of the program is :

$$\log_2(n) \times n^2 = O(n^2 \log_2 n)$$

12. (a) Given that the efficiency of an algorithm is $5n^2$, if a step in this algorithm takes 1 nanosecond (10^{-9}), how long does it take the algorithm to process an input of size 1000 ?

(b) Given that the efficiency of an algorithm is n^3, if a step in this algorithm takes 1 nanosecond (10^{-9}), how long does it take the algorithm to process an input of size 1000 ?

(c) Given that the efficiency of an algorithm is $5n\log_2(n)$, if a step in this algorithm takes 1 nanosecond (10^{-9}), how long does it take the algorithm to process an input of size 1000 ?

Solution.

(a) Given size is $n=1000$, hence the time taken is : $5\times1000^2\times10^{-9}=5$ms

(b) Given size is $n=1000$, hence the time taken is : $1000^3\times10^{-9}=1$s

(c) Given size is $n=1000$, hence the time taken is : $5\times1000\times\log_2(1000)\times10^{-9}\approx\mu$s.

[Recall that 10^{-3} is *milli* (m) ; 10^{-6} is *micro* (μ) ; 10^{-9} is *nano* (n) ; 10^{-12} is *pico* (p)]

GLOSSARY

Algorithm	Sufficiency finite and precise method for accomplishing a task.
Average-Case Complexity	Expected running time of an algorithm.
Best-Case Complexity	Running time of an algorithm in case of optimal performance.
Complexity	Measure of an algorithm's performance.
Worst-Case Complexity	Upper bound on running time of an algorithm.

Assignment

Type A : Short Answer Questions/Conceptual Questions

1. What is an algorithm? What do you understand by algorithm performance ?
2. What is computational complexity ?
3. Which factors affect an algorithm's performance ?
4. What are different types of complexities that are considered ?
5. What do you understand by Big-O notation? What is its significance ?
6. What do you understand by best-case, worst-case and average case complexities ? When are they considered ?
7. Determine the big-O notation for the following calculated complexity :

   (a) $5n^{5/2}+n^{2/5}$ (b) $6\log_2(n)+9n$ (c) $3n^4+n\log_2(n)$ (d) $5n^2+n^{3/2}$
8. Reorder the following efficiencies from the smallest to the largest :

   (a) 2^n (b) $n!$ (c) n^5 (d) 10,000 (e) $n\log_2(n)$
9. Reorder the following efficiencies from the smallest to the largest :

   (a) $n\log_2(n)$ (b) $n+n^2+n^3$ (c) 2^4 (d) $n^{0.5}$
10. Determine the big-O notation for the following :

    (a) $5n^{5/2}+n^{2/5}$ (b) $6\log_2(n)+9n$ (c) $3n^4+n\log_2(n)$ (d) $5n^2+n^{3/2}$

11. Given that the efficiency of an algorithm is $5n^2$, if a step in this algorithm takes 1 nanosecond (10^{-9}), how long does it take the algorithm to process an input of size 1000 ?
12. Given that the efficiency of an algorithm is n^3, if a step in this algorithm takes 1 nanosecond (10^{-9}), how long does it take the algorithm to process an input of size 1000 ?
13. Given that the efficiency of an algorithm is $5n \log_2(n)$, if a step in this algorithm takes 1 nanosecond (10^{-9}), how long does it take the algorithm to process an input of size 1000 ?

Type B : Application Based Questions

1. *Calculate the run-time efficiency of the following program segment :*

```
i = 1
while i <= n :
    print (i)
    i = i + 1
```

2. *Calculate the run-time efficiency of the following program segment :*

```
i = 1
while i <= n :
    j = 1
    while j <= n :
        k = 1
        while j <= n :
            print(i, j, k)
            k = k + 1
        j = j + 1
    i = i + 1
```

3. *If the function doIt( ) has an efficiency factor of 5n, calculate the run time efficiency of the following program segment.*

```
i = 1
while i <= n :
    doIt(...)
    i = i + 1
```

4. *If the efficiency of the function doIt( ) can be expressed as $O(n) = n^2$, calculate the efficiency of the following program segment.*

```
i = 1
while i <= n :
    j = 1
    while j <= n :
        doIt(...)
        j = j + 1
    i = i + 1
```

5. *If the efficiency of the function doIt( ) can be expressed as $O(n) = n^2$, calculate the efficiency of the following program segment.*

```
i = 1
while i < n :
    doIt(...)
    i = i * 2
```

6. Given a list A of appropriate size, what is the complexity of following code in terms of n ?

```
i = x = n
while i > 0 :
    i = i - 1
    x += A[ i ]
```

7. Given three lists A, B, C of appropriate sizes, what is the complexity of following code in terms of n, m and p ?

```
for in in range(n):
    for k in range(p) :
        x = 0
        for j in range(m) :
            x += A[i] * B[k]
```

8. Given integer variable x and a list A of appropriate size, what is the complexity of following code in terms of n ?

```
p = -1
q = n
while p + 1 < q :
    m = (p + q) / 2
    if A[m] < x :
        p = m
    else :
        q = m
```

9. Given a list A of appropriate size, what is the complexity of following code in terms of n ?

```
m = A[0]
for i in range(n) :
    if A[i] > m :
        m = A[i]
k = 0
for i in range(n) :
    if A[i] == m :
        k = k + 1
```

10. Given a list A of appropriate size, what is the complexity of following code in terms of *n* ?

```
for I in range(n-1, 0, -1) :
    x = A[I]
    A[I] = A[0]
    p = 0
    while True :
        c = 2 * p + 1
        if c >= I :
            break
        if c != I - 1 and A[c] < A[c + 1] :
            c = c + 1
        if x >= A[c] :
            break
        A[p] = A[c]
        p = c
    A[p] = x
```

11. Given a list A of appropriate size, what is the complexity in terms of n ?

```
i = x = n
while i > 0 :
    i = i - 1
    x = x + A[i]
```

12. Given integer variable *x* and a list A of appropriate size, what is the complexity of following code in terms of *n* ?

```
p = -1
q = n
while (p + 1) < q :
    m = (p + q)/2
    if A[m] < x :
        p = m
    else :                    # assigning other value
        q = m
```

13. Given a list A of appropriate size, what is the complexity in terms of *n* ?

```
m = A[0]
for I in range(n) :
    if A[i] > m :
        m = A[i]
k = 0
for i in range(n ):
    if A[i] == m :
        k = k + 1
```

14. What is the time complexity of following algorithms ?

    (*i*) Insertion Sort (*ii*) Binary search (*iii*) linear search ?

15. Based on the complexity analysis, which sorting algorithm is more efficient and why ?