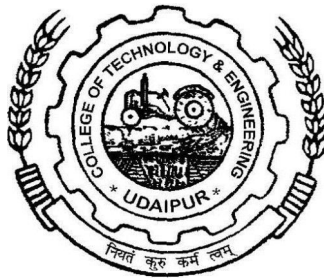


A Training Report On

**Data Structures and Algorithms in Python**

Submitted in partial fulfillment of the requirements  
For The Degree Of  
**BACHELOR OF TECHNOLOGY**  
in  
**ELECTRONICS AND COMMUNICATION ENGINEERING**



Session 2019-2020

Training Duration (from 3/5/20 to 27/5/20)

**Submitted to**  
Navneet  
**Enrollment No.**

**Submitted by**  
Navneet  
**Subject Code**

III year ECE

---

**DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING**  
**COLLEGE OF TECHNOLOGY AND ENGINEERING**

MAHARANA PRATAP UNIVERSITY OF AGRICULTURE AND TECHNOLOGY, UDAIPUR, RAJASTHAN

---

Compiled in L<sup>A</sup>T<sub>E</sub>X

---

## Certificate



---

(i)

## **Acknowledgement**

I would like to express my special thanks of gratitude to my teacher **Dr.Navneet Agarwal Dr.Sunil Joshi** as well as our principal (Name of the principal)who gave me the golden opportunity to do this wonderful project on the topic (Write the topic name), which also helped me in doing a lot of Research and i came to know about so many new things I am really thankful to them. Secondly i would also like to thank my parents and friends who helped me a lot in finalizing this project within the limited time frame.

# Contents

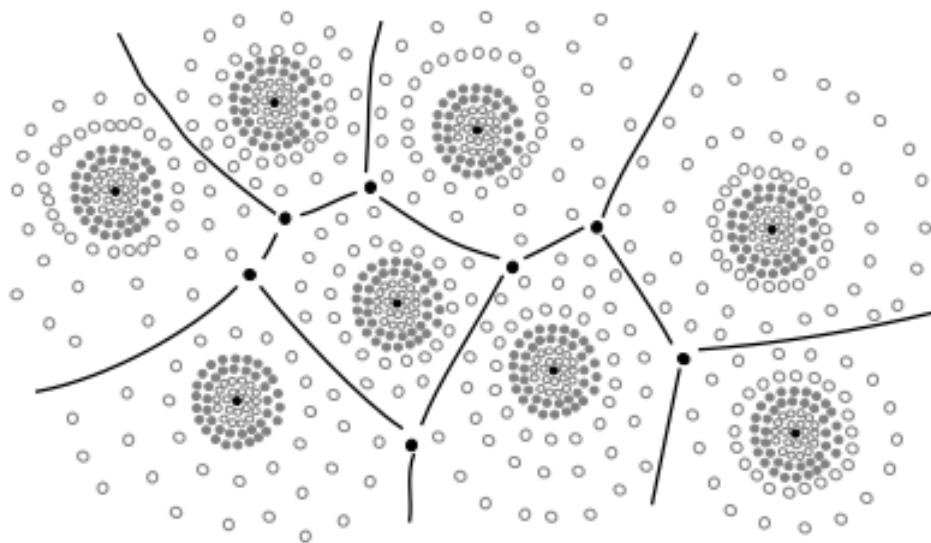
<b>1</b>	<b>Analysing Time Complexity</b>	<b>7</b>
1.1	Introduction . . . . .	8
1.2	Counting Primitive Operations . . . . .	9
1.3	Functions used in Analysis of Algorithms . . . . .	10
<b>2</b>	<b>Stacks</b>	<b>11</b>
2.1	Introduction . . . . .	12
2.2	Basic Functions that Stack supports . . . . .	12
2.3	Implementation of Stack . . . . .	12
<b>3</b>	<b>new</b>	<b>15</b>
3.1	new1 . . . . .	15

# List of Figures

I.1	Results of an experimental study on the running time of an algorithm. A dot with coordinates $(n, t)$ indicates that on an input of size $n$ , the running time of the algorithm was measured as $t$ milliseconds (ms). . . . .	8
I.2	The difference between best-case and worst-case time. Each bar represents the running time of some algorithm on a different possible input. . . . .	9
I.3	Output of Listing I.1 . . . . .	10
2.1	Output of listing 2.2 . . . . .	14

# Chapter I

## Analysing Time Complexity



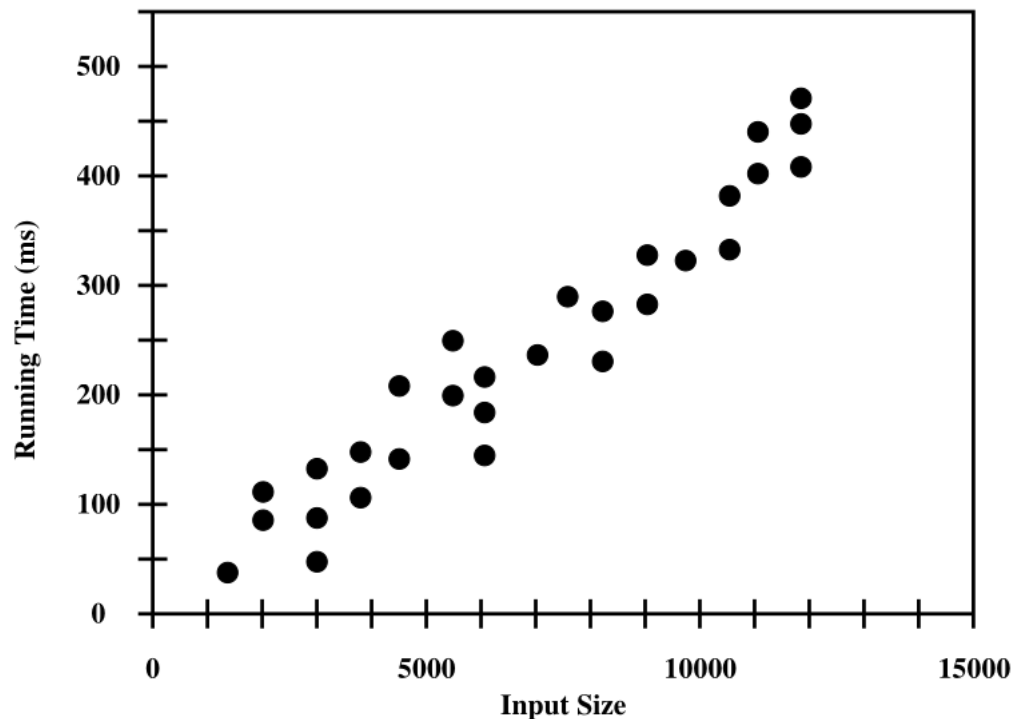
### Contents

1.1	Introduction . . . . .	8
1.2	Counting Primitive Operations . . . . .	9
1.3	Functions used in Analysis of Algorithms . . . . .	10

## 1.1 Introduction

Algorithmic complexity is concerned about how fast or slow particular algorithm performs. We define complexity as a numerical function  $T(n)$  - time versus the input size  $n$ . We want to define time taken by an algorithm without depending on the implementation details. But you agree that  $T(n)$  does depend on the implementation! A given algorithm will take different amounts of time on the same inputs depending on such factors as: processor speed; instruction set, disk speed, brand of compiler and etc. The way around is to estimate efficiency of each algorithm asymptotically. We will measure time  $T(n)$  as the number of elementary "steps" (defined in any way), provided each such step takes constant time.

Let us consider two classical examples: addition of two integers. We will add two integers digit by digit (or bit by bit), and this will define a "step" in our computational model. Therefore, we say that addition of two  $n$ -bit integers takes  $n$  steps. Consequently, the total computational time is  $T(n) = c * n$ , where  $c$  is time taken by addition of two bits. On different computers, addition of two bits might take different time, say  $c_1$  and  $c_2$ , thus the addition of two  $n$ -bit integers takes  $T(n) = c_1 * n$  and  $T(n) = c_2 * n$  respectively. This shows that different machines result in different slopes, but time  $T(n)$  grows linearly as input size increases.



**Figure 1.1:** Results of an experimental study on the running time of an algorithm. A dot with coordinates  $(n, t)$  indicates that on an input of size  $n$ , the running time of the algorithm was measured as  $t$  milliseconds (ms).

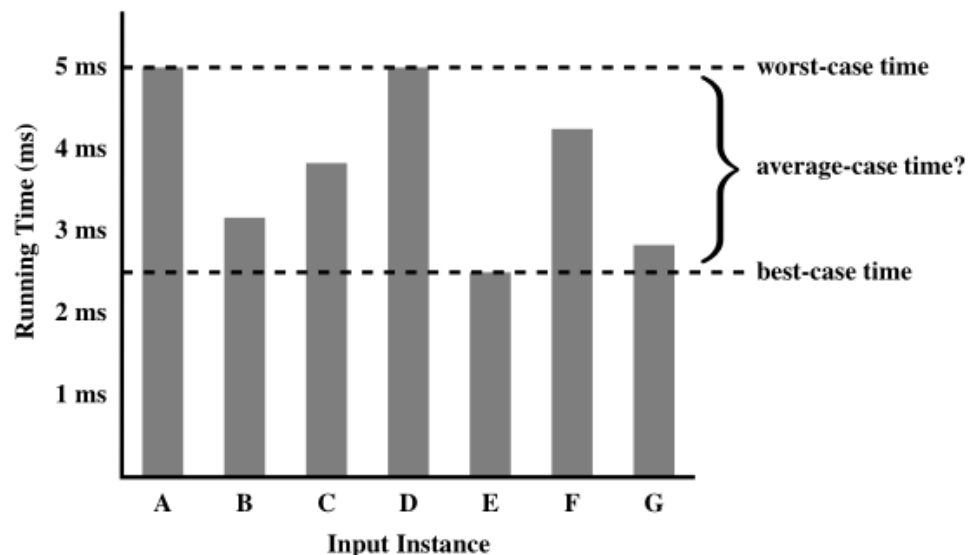


## 1.2 Counting Primitive Operations

To analyze the running time of an algorithm without performing experiments, we perform an analysis directly on a high-level description of the algorithm (either in the form of an actual code fragment, or language-independent pseudo-code). We define a set of *primitive operations* such as the following:

- Assigning an identifier<sup>1</sup> to an object
- Determining the object associated with an identifier
- Performing an arithmetic operation (for example, adding two numbers)
- Comparing two numbers
- Accessing a single element of a Python list by index
- Calling a function (excluding operations executed within the function)
- Returning from a function.

An algorithm may run faster on some inputs than it does on others of the same size. Thus, we may wish to express the running time of an algorithm as the function of the input size obtained by taking the average over all possible inputs of the same size. Unfortunately, such an average-case analysis is typically quite challenging. It requires us to define a probability distribution on the set of inputs, which is often a difficult task. **Figure 1.2** schematically shows how, depending on the input distribution, the running time of an algorithm can be anywhere between the worst-case time and the best-case time. For example, what if inputs are really only of types “A” or “D”? <sup>1</sup>



**Figure 1.2:** The difference between best-case and worst-case time. Each bar represents the running time of some algorithm on a different possible input.

An average-case analysis usually requires that we calculate expected running times based on a given

<sup>1</sup>Python’s identifier is similar to Reference variable in Java or pointer variable in C++

input distribution, which usually involves sophisticated probability theory. We will characterize running times in terms of the *worst case*, as a function of the input size,  $n$ , of the algorithm.

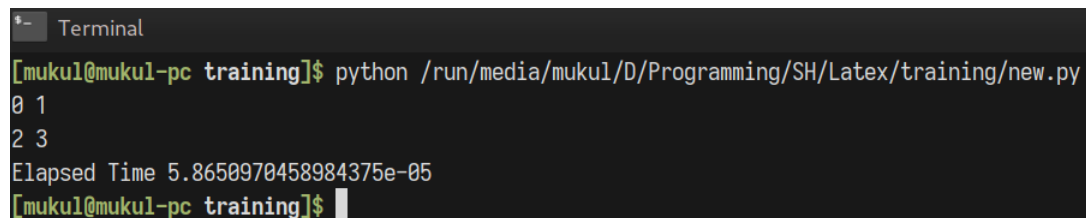
If an algorithm has been implemented, we can study its running time by executing it on various test inputs and recording the time spent during each execution. A simple approach for doing this in Python is by using the time function of the time module. This function reports the number of seconds, or fractions thereof, that have elapsed since a benchmark time known as the epoch. The choice of the epoch is not significant to our goal, as we can determine the elapsed time by recording the time just before the algorithm and the time just after the algorithm, and computing their difference, as follows:

```

1 ! /run/media/mukul/D/SH/test.py
2 from time import time
3 start_time = time()      # record starting time
4 num=[1,2,-2,5,6,6]
5 target=3
6 for i in range(len(num)): #Two Sum
7     x=num[i]
8     for j in range(i+1,len(num)):
9         if num[j]==target-x:
10             print(i,j)
11 end_time = time()      # record ending time
12 elapsed = end_time - start_time # compute the elapsed time

```

**Listing 1.1:** Calculating Run time



```

*- Terminal
[mukul@mukul-pc training]$ python /run/media/mukul/D/Programming/SH/Latex/training/new.py
0 1
2 3
Elapsed Time 5.8650970458984375e-05
[mukul@mukul-pc training]$

```

**Figure 1.3:** Output of Listing 1.1

### 1.3 Functions used in Analysis of Algorithms

# **Chapter 2**

## **Stacks**

## 2.1 Introduction

A **stack** is a collection of objects that are inserted and removed according to the **last-in, first-out (LIFO) principle**. A user may insert objects into a stack at any time, but may only access or remove the most recently inserted object that remains (at the so-called “top” of the stack). The name “stack” is derived from the metaphor of a stack of plates in a spring-loaded, cafeteria plate dispenser. In this case, the fundamental operations involve the “pushing” and “popping” of plates on the stack. When we need a new plate from the dispenser, we “pop” the top plate off the stack, and when we add a plate, we “push” it down on the stack to become the new top

## 2.2 Basic Functions that Stack supports

- **push(e)**: Add element to the top of the Stack
- **pop(e)**: Remove and return the top element of the Stack
- **top()**: Return a reference to the top element of Stack
- **isempty()**: Return True if stack does not contain any elements

*The following table shows a series of stack operations*

Operation	Return Value	Stack Contents
push(5)	-	[5]
push("mukul")	-	[5,'mukul']
len(S)	2	[5,'mukul']
pop()	('mukul')	[5]
len(S)	1	[5]
isempty()	False	[5]
pop()	5	[]
push(9)	-	[9]
top()	9	[9]

## 2.3 Implementation of Stack

**Exception Class** When pop is called on an empty Python list, it formally raises an IndexError, as lists are index-based sequences. That choice does not seem appropriate for a stack, since there is no assumption of indices. Instead, we can define a new exception class that is more appropriate. listing 2.1 shows such an Empty class

```

1 class Empty{Exception}
2 # Error when the container is empty
3 pass # Does nothing

```

**Listing 2.1:** Exception Class

```

1 class ArrayStack:
2     # Initialize the list
3     def __init__(self):
4         self.data = []
5
6     # returns the length of list
7     def __len__(self):
8         return len(self.data)
9
10    # returns True if the list is empty
11    def isempty(self):
12        return len(self.data) == 0
13
14    # add an element to the end of the list
15    def push(self, e):
16        self.data.append(e)
17
18    # returns the top element of the stack
19    def top(self):
20        if self.isempty():
21            raise Empty('Stack is empty')
22        return self.data[-1]
23
24    # delete and prints the last element of stack
25    def pop(self):
26        if self.isempty():
27            raise Empty('Stack is empty')
28
29        temp = self.data[-1]
30        del self.data[-1]
31        return temp
32
33 a = ArrayStack()
34 a.push(3)
35 print("Pushed {} into the stack".format(a.top()))
36
37 a.push('mukul')
38 print("Pushed {} into the stack".format(a.top()))
39
40 print("Top Element:{}".format(a.top()))
41
42 a.pop()
43 print("Top Element after popping:{}".format(a.top()))

```

**Listing 2.2:** Implementation of Stack using List

```

2+ new1.tex 2+ new.py
22     # add an element to the end of the list
21     def push(self, e):
20         self.data.append(e)
19
18     # returns the top element of the stack
17     def top(self):
16         if self.isempty( ):
15             raise Empty('Stack is empty')
14             return self.data[-1]
13
12     # delete and prints the last element of stack
11     def pop(self):
10         if self.isempty( ):
9             raise Empty('Stack is empty')
8             return self.data.pop()
7 a = ArrayStack()
6 a.push(3)
5 print("Pushed {} into the stack".format(a.top()))
4 a.push('mukul')
3 print("Pushed {} into the stack".format(a.top()))
2 print("Top Element:{}".format(a.top()))
1 a.pop()
36 print("Top Element after popping:{}".format(a.top()))
~
new.py 36,32
Top Element after popping:8
[mukul@mukul-pc training]$ python new.py
Pushed 3 into the stack
Pushed mukul into the stack
Top Element:mukul
Top Element after popping:3
[mukul@mukul-pc training]$
<sh (1) [mukul@mukul-pc:/run/media/mukul/D/Programming/SH/Latex/training] 1,1
"new.py" 36L, 850C written

```

Figure 2.1: Output of listing 2.2

# Chapter 3

## new

### 3.1 newI