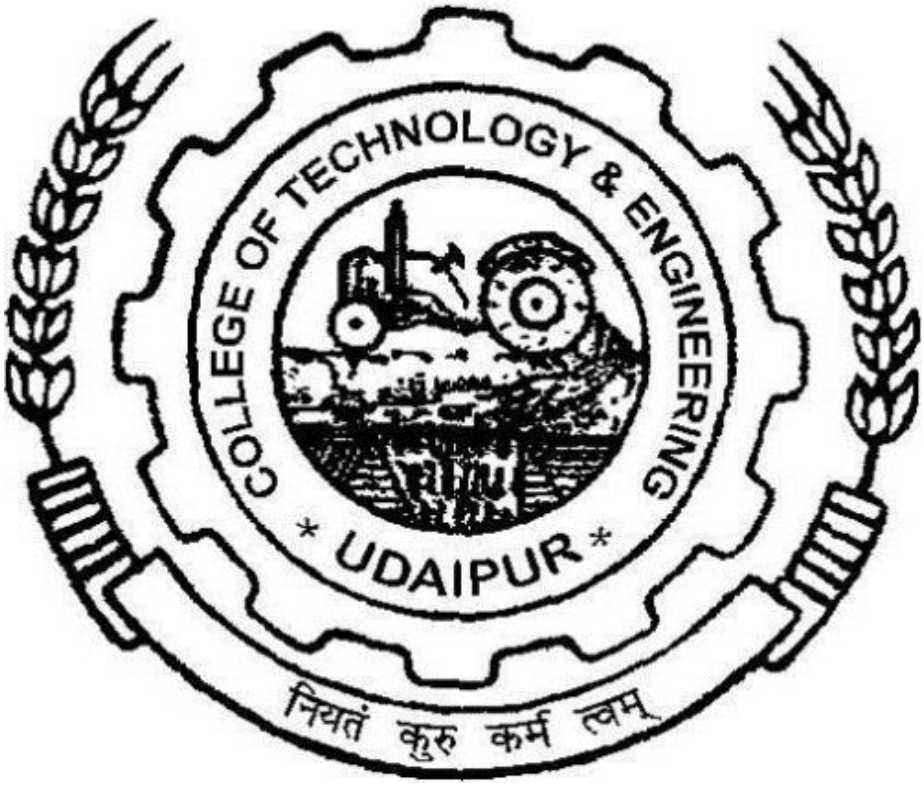
A Training Report On
Data Structures and Algorithms in Python
Submitted in partial fulfillment of the requirements
For The Degree Of
BACHELOR OF TECHNOLOGY
in
ELECTRONICS AND COMMUNICATION ENGINEERING



Session 2019-2020

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

2

Submitted to
Navneet
Enrollment No.

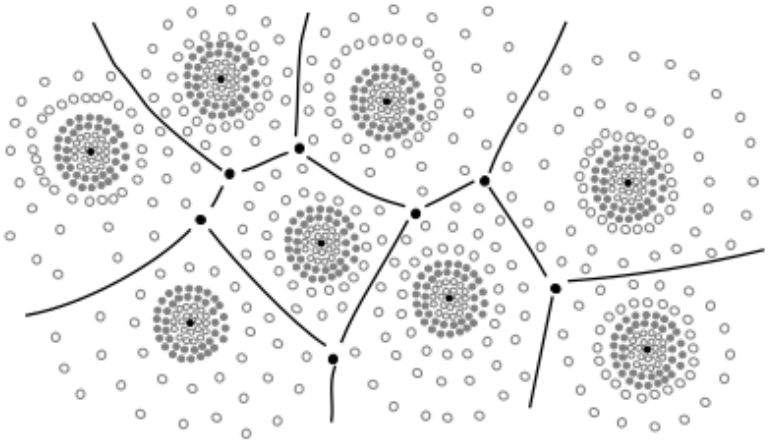
Submitted by
Navneet
Subject Code

III year ECE

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
COLLEGE OF TECHNOLOGY AND ENGINEERING
MAHARANA PRATAP UNIVERSITY OF AGRICULTURE AND TECHNOLOGY,UDAIPUR,RAJASTHAN

<> in LaTeX

Analysing Time Complexity



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, additon of two bits might take different time, say c1 and c2, thus the additon of two n-bit integers takes $T(n) = c1 * n$ and $T(n) = c2 * n$ respectively. This shows that different machines result in different slopes, but time $T(n)$ grows linearly as input size increases.

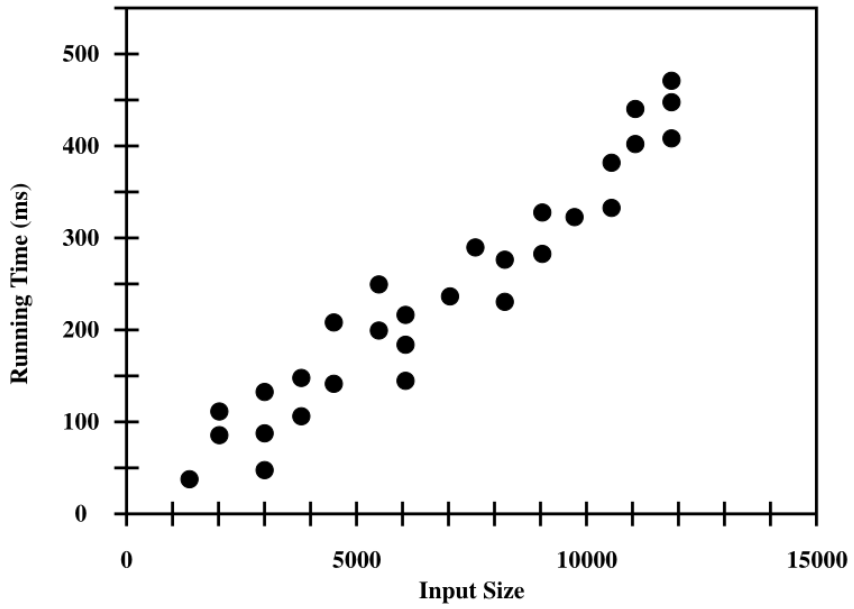


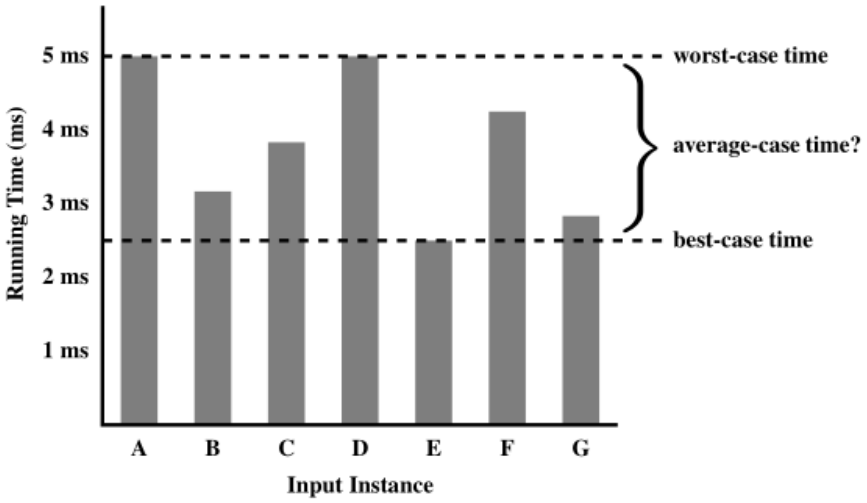
Figure 1: Results of an experimental study on the running time of an algorithm. A dot with coordinates (n, t) indicates that on an input of size n, the running time of the algorithm was measured as t milliseconds (ms).

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[1] 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** 1.1 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"?



[fig2]

1

An average-case analysis usually requires that we calculate expected running times based on a given input distribution, which usually involves sophisticated probability theory. Therefore, for the remainder of this book, unless we specify otherwise, we will characterize running times in terms of the ***worst case***, as a function of the input size, n, of the algorithm.

[1]Python's identifier is similar to Reference varible in Java or pointer variable in C++

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:

```
! /run/media/mukul/D/SH/test.py
from time import time
start time = time()          # record starting time
num=[1,2,-2,5,6,6]
target=3
for i in range(len(num)): #Two Sum
    x=num[i]
    for j in range(i+1,len(num)):
        if num[j]==target-x:
            print(i,j)
end time = time()        # record ending time
elapsed = end_time - start_time # compute the elapsed 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 2: Output of Listing 1.1

Functions used in Analysis of Algorithms