**PYTHON PRIMER**

**INTRODUCTION TO DATA STRUCTURES AND ALGORITHMS IN PYTHON**

All computer programs have two parts – data structures and algorithms. Any logic underlying a piece of code can be explained in terms of data structures and algorithms.

**Data structures –** how we store the data

**Algorithm –** how we manipulate the data

There is a significant gain in choosing the right data structure and algorithm for solving a problem.

What is ‘significant gain’?

In computer programs, gain can be in terms of two parameters – time and space. To understand this, we need to understand a term called complexity.

Complexity is the rate at which the difficulty of solving the problem changes. It is not a numerical measurement. It is a measure of growth. Growth in the time and space needed to solve a problem. When we say growth, it is the additional resources needed is measured with respect to the input.

To denote this, we use the asymptotic notation. If the rate of growth of a piece of code is f(x), then the asymptotic notation is given by O(f(x)). There are nuances involved when using this notation which we will cover later. Before we delve into all of that, we will have some exercises in mathematics. It involves finding the relation between the input and the output.

**Exercise 1 – Beautiful Curves**

In the subsequent problems, there will be a table or graph which has x and f(x) explicitly mentioned. You need to look at the given data and tell me f(x) in terms of x. I’ll give an example before we begin,

**Example:**

|  |  |
| --- | --- |
| x | f(x) |
| 12 | 13 |
| -1 | 0 |
| 10 | 11 |
| 993 | 994 |

If you observe the above example, you can see that F(x) is 1 more than the value in the corresponding X column. Simply put,

f(x) = x + 1.

In the following questions, state the relation between x and f(x).

1.

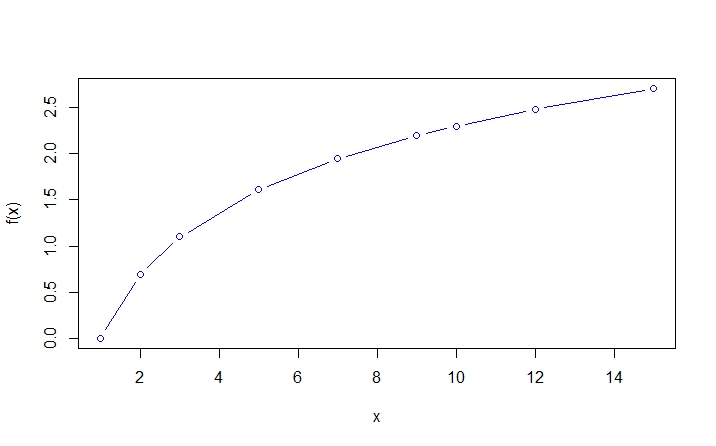
x = 1 4 5 7 10 13 16 19 20

f(x) = 6 21 30 54 105 174 261 366 405

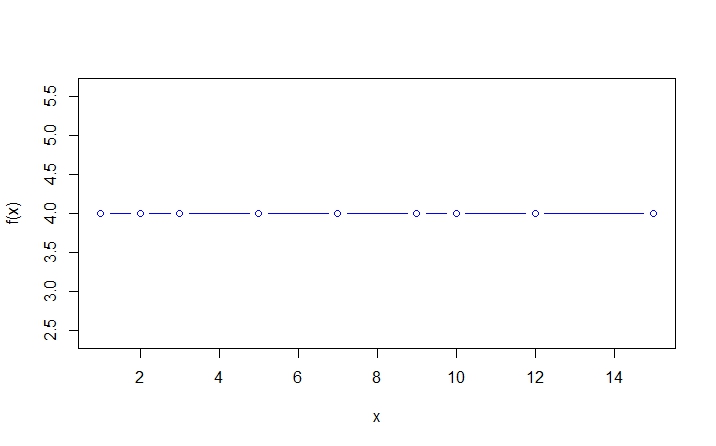
2.

|  |  |
| --- | --- |
| x | f(x) |
| -1 | 1.00000000 |
| 2 | -0.50000000 |
| -3 | 0.33333333 |
| 5 | -0.20000000 |
| -7 | 0.14285714 |
| -9 | 0.11111111 |
| 10 | -0.10000000 |
| 12 | -0.08333333 |
| 15 | -0.06666667 |

3.



4.



**Answers:**

1. f(x) = x^2 + 5
2. f(x) = -(1/x)
3. f(x) = log(x)
4. f(x) = 4

**ANOTHER LOOK AT COMPLEXITY AND OTHER THEORETICAL STUFF**

We looked at the mathematics behind complexity and the kinds of data we will encounter along the way. In this chapter, we will see how to use asymptotic notation and some programming snippets to illustrate the same.

Asymptotic notation is a method to express the complexity of a given problem/solution. We will discuss only the Big Oh notation i.e. worst case complexity analysis.

**Worst case complexity analysis:** Given an algorithm, what will be the worst-case complexity it can attain? As always, this is measured in terms of input parameters. By this what we say is, given an algorithm, the time/space resources consumed may at worst be of some value. It does not mean, the time/space consumed is always the worst.

What is the advantage of this approach? When we do a worst-case analysis, our algorithm now accounts for all lesser complexities it may attain and we have a control over the extent to which our algorithm can perform.

Another point to note is that, in asymptotic notation of f(x), denoted by O(f(x)), f(x) may have many terms like polynomial, logarithmic and exponential terms. When denoting it, we ignore the ‘lower’ terms and consider only the ‘higher’ terms. That is, some terms tend to dominate the equation and the other terms become negligible. Why are they negligible?

To understand that, we must understand the theory used behind asymptotic notation. The asymptote is used by considering very large inputs (i.e. infinity). For those inputs, the bigger terms tend to dominate. Let us see that with an example,

Say *f(x) = x2 +x+1*

For an input x=1, f(x)=1+1+1 = 3

For x=10, f(x)=102+10+1=111

For x=100, f(x)=1002+100+1=10101.

As we can see, the influence of the x2 in f(x) term keeps increasing with increase in x value. This is what we mean by dominate. i.e. there is not much difference in ignoring the other terms. We can predict the growth rate with that one term. i.e. the complexity is denoted in the above case as, O(f(x)) is **O(x2).**

Earlier, we mentioned ‘lower’ and ‘higher’ terms. What are they? To keep it simple, it is a hierarchy of complexities. Some algorithms higher complexity than others. This is because of the presence of certain terms.

The terms can be any one of these or a combination of these,

1. Constants – O(1)
2. Linear – O(n)
3. Polynomial – O(nk) where k is a constant
4. Logarthmic – O(log n)
5. Exponential – O(kn) where k is a constant.

We will take a leap and see some programming snippets and pseudo-code and analyse their complexity.

**INTRODUCTION TO PYTHON – HELLO WORLD AND THE FUNCTIONAL STYLE**

Python is a high level general-purpose programming language. It supports various styles of programming – object oriented, structural and functional. The main philosophy followed in Python is the emphasis on simplicity of code. In any place, simple understandable code is preferred.

Python has a great support for functions. Functions are treated as first-class objects i.e. you can pass functions like objects. Why do we emphasize on functions? Because they can act as a basic block of any large program. If we design functions with the UNIX philosophy i.e. a function must do only one thing and do it well, we can write good code.

Before we start any kind of programming, there are two things we must focus on – modularity and error handling. Modularity is done with functions; error handling with try-except blocks. We will use these principles from the very beginning to make sure we get used to it.

Whatever we are going to write, we’ll make it generic and modular. We will handle all possible exceptions in all stages. In addition, we must also document our functions and operations.