Unit1: Introduction

Prepared By: Asst. Prof. Himani Desai

Contents

- Pseudocode
- Abstract Data Type
- Types of Data Structure
- ADT Model and its implementation
- Algorithm Efficiency
- Properties of Asymptotic Notation

Pseudocode

- When you discuss how a program works, showing the code may not be the easiest way of explaining it — especially if the person you are talking to is less familiar with the programming language you are using. In these situations you may want to express your program in **pseudocode**.
- **Pseudo code** is a methodology that allows the programmer to represent the implementation of an algorithm.
- Often at times, algorithms are represented with the help of pseudo codes as they can be interpreted by programmers no matter what their programming background or knowledge is.
- An algorithm is a sequence of steps which is utilized in order to solve a computational problem whereas pseudocode is nothing but a more simple form of an algorithm which involves some part of natural language to enhance the understandability of the high-level programming constructs or for making it more human-friendly.

Pseudocode (Cont.)

- Pseudocode is an English-like representation of the algorithm logic. It is part English, part structured code.
- The English part provides a relaxed syntax that describes what must be done without showing unnecessary details such as error messages.
- The code part consists of an extended version of the basic algorithmic constructs—sequence, selection, and iteration.

Pseudocode (Cont.)

Statement Constructs:

- Programs are designed using common building blocks. These building blocks, known as programming constructs (or statement constructs), form the basis for all programs.
- There are three main programming constructs. They are:
  - **Sequence** statements
  - **Selection** statements
  - **Iteration** statements
- These three constructs are extremely important. They can help you control the flow of your program; allowing you to specify *how or when parts of your code are executed.*

Pseudocode (Cont.)

Sequence

- Sequence is the order in which programming statements are executed.
- Programming statements usually run one after another in order, unless one of the other programming constructs is used.
- The sequence of a program is extremely important as once these are translated, carrying out instructions in the wrong order leads to a program performing incorrectly.

Selection

- Selection is a programming construct where a section of code is run only if a condition is met. In programming, there are occasions when a decision needs to be made. Selection is the process of making a decision. The result of the decision determines which path the program will take next.
- For example, a program could tell a user whether they are old enough to learn how to drive a car. If the user's age meets the required driving age, the program would follow one path and execute one set of statements. Otherwise, it would follow a different path and execute a different set of statements.

Pseudocode (Cont.)

Iteration

- There are times when a program needs to repeat certain steps until told, or until a condition has been met. This process is known as iteration.
- Iteration is often referred to as looping, since the program ‘loops’ back to an earlier line of code. Iteration is also known as repetition.
- Iteration allows programmers to simplify a program and make it more efficient . Instead of writing out the same lines of code again and again, a programmer can write a section of code once, and ask the program to execute the same line repeatedly until no longer needed.
- When a program needs to iterate a set number of times, this is known as definite iteration and makes use of a FOR loop. A FOR loop uses an extra variable called a loop counter that keeps track of the number of times the loop has been run.

Example of Pseudocode

Example - Task of computing the final price of an item after figuring in sales tax.

1. get price of item
2. get sales tax rate
3. sales tax = price of item times sales tax rate
4. final price = price of item plus sales tax
5. display final price

Atomic data and Composite data

- **Atomic data** are data that consist of a single piece of information; that is, they cannot be divided into other meaningful pieces of data.
- There are four primitive atomic data types: *booleans*, *integers*, *characters* and *floats*.
- For example, the integer 4562 may be considered a single integer value. Of course, we can decompose it into digits, but the decomposed digits do not have the same characteristics of the original integer; they are four single-digit integers ranging rom 0 to 9.
- The opposite of atomic data is **composite data**. Composite data can be broken out into subfields that have meaning.
- Complex data type contains multiple values and provides access to its nested values.

Atomic data and Composite data (Cont.)

- Data types composed of two or more other data types. Here are *some*:

Type	Values	Python implementation
Array	Mutable object containing other values	list or []
Record	Immutable object containing other values	tuple or ()
Union	Contains values that can be multiple types	dict or {}

Data Type

- The Data Type is basically a type of data that can be used in different computer program. It signifies the type like integer, float etc.
- A data type consists of two parts: a set of data and the operations that can be performed on the data.

Type	Values	Operations
integer	-∞, ... , -2, -1, 0, 1, 2,... , ∞	*, +, -, %, /, ...
float	-∞, ... , 0.0, ... , ∞	*, +, -, /, ...
character	\0, ..., 'A', 'B', ... , 'a', 'b', ... , ~	<, >, ...

Data Structure

- A data structure is an aggregation of atomic and composite data into a set with defined relationships. In this definition *structure* means a set of rules that holds the data together.
- In other words, if we take a combination of data and fit them into a structure such that we can define its relating rules, we have made a data structure.
- A **data structure** is a particular way of organizing data in a computer so that it can be used effectively.
- The data structure by name indicates itself that organizing the data in memory. There are many ways of organizing the data in the memory.
- The data structure is not any programming language like C, C++, java, python, etc. It is a set of algorithms that we can use in any programming language to structure the data in the memory.
- To structure the data in memory, 'n' number of algorithms were proposed, and all these algorithms are known as Abstract data types. These abstract data types are the set of rules.

Abstract Data Type (ADT)

- The abstract data type is special kind of datatype, whose behavior is defined by a set of values and set of operations. The keyword “Abstract” is used as by using these data types, we can perform different operations. But how those operations are working that is totally hidden from the user. The ADT is made of with primitive data types, but operation logics are hidden.
- An abstract data type (or ADT) is a programmer-defined data type that specifies a set of data values and a collection of well-defined operations that can be performed on those values.
- Some examples of ADT are Stack, Queue, List etc.
- A data structure is a way of organizing the data so that it can be used efficiently. Here, we have used the word efficiently, which in terms of both the space and time.
- An ADT tells **what** is to be done and data structure tells **how** it is to be done. In other words, we can say that ADT gives us the blueprint while data structure provides the implementation part.

Abstract Data Type (ADT) (Cont.)

- With an ADT users are not concerned with how the task is done but rather with what it can do.
- In other words, the ADT consists of a set of definitions that allow programmers to use the functions while hiding the implementation.
- This generalization of operations with unspecified implementations is known as abstraction.
- We abstract the essence of the process and leave the implementation details hidden.

The concept of abstraction means:

1. We know *what* a data type can do.
2. *How* it is done is hidden.

Abstract Data Type (ADT) (Cont.)

- An abstract data type is a data declaration packaged together with the operations that are meaningful for the data type.
- In other words, we encapsulate the data and the operations on the data, and then we hide them from the user.

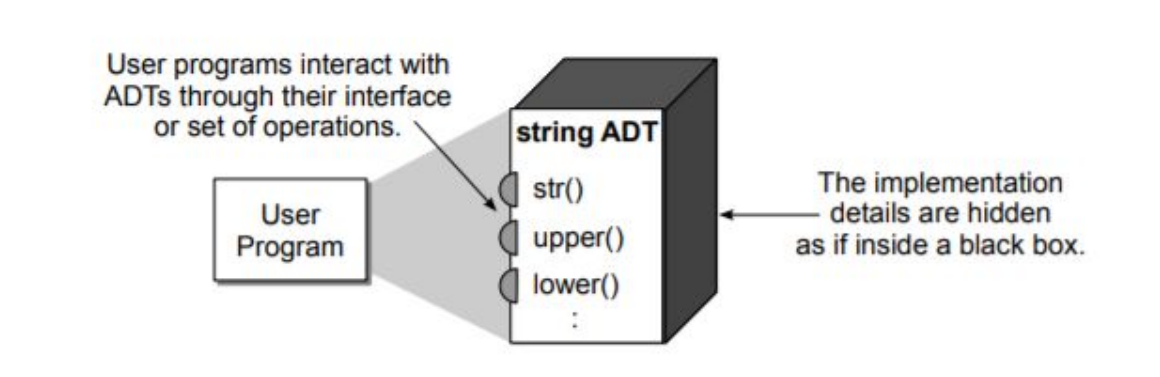
Abstract Data Type

1. Declaration of data
2. Declaration of operations
3. Encapsulation of data and operations

- For example, integers are an ADT, defined as the values ..., —2, —1, 0, 1, 2, ..., and by the operations of addition, subtraction, multiplication, and division, together with greater than, less than, etc., which behave according to familiar mathematics (with care for integer division), independently of how the integers are represented by the computer.

Abstract Data Type (ADT) (Cont.)

Abstract data types can be viewed like black boxes as illustrated in below figure. User programs interact with instances of the ADT by invoking one of the several operations defined by its interface.

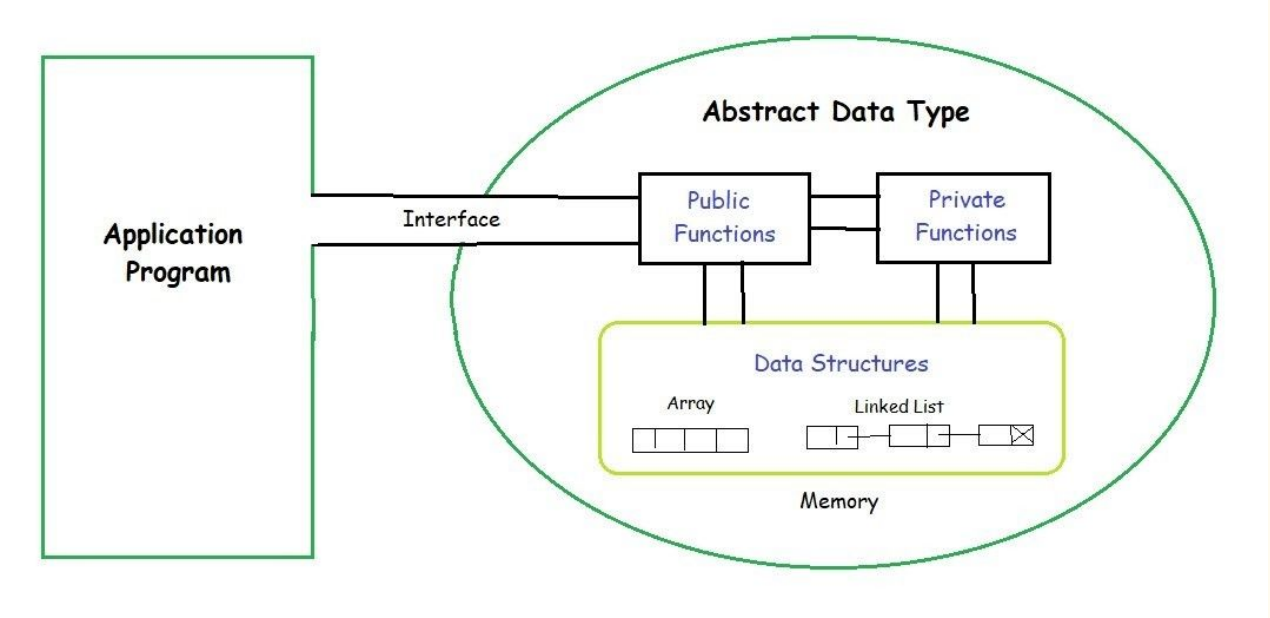


The implementation of the various operations are hidden inside the black box, the contents of which we do not have to know in order to utilize the ADT. There are several advantages of working with abstract data types and focusing on the “what” instead of the “how.”

Why Abstract data type become necessity?

- Earlier if a programmer wanted to read a file, the whole code was written to read the physical file device. So that is how *Abstract Data Type* (ADT) came into existence.
- The code to read a file was written and placed in a *library* and made available for everyone's use. This concept of ADT is being used in the *modern languages* nowadays.
- **Example:** The code to read the *keyboard* is an ADT. It has a *data structure, a character, and a set of operations* that can be used to read that data structure.
    - ADT does the work that is necessary implementation and it is not of much concern that how the work is being done. It is like **unspecified implementation** which can be termed as ***Abstraction.***

Model for Abstract Data Type (Cont.)



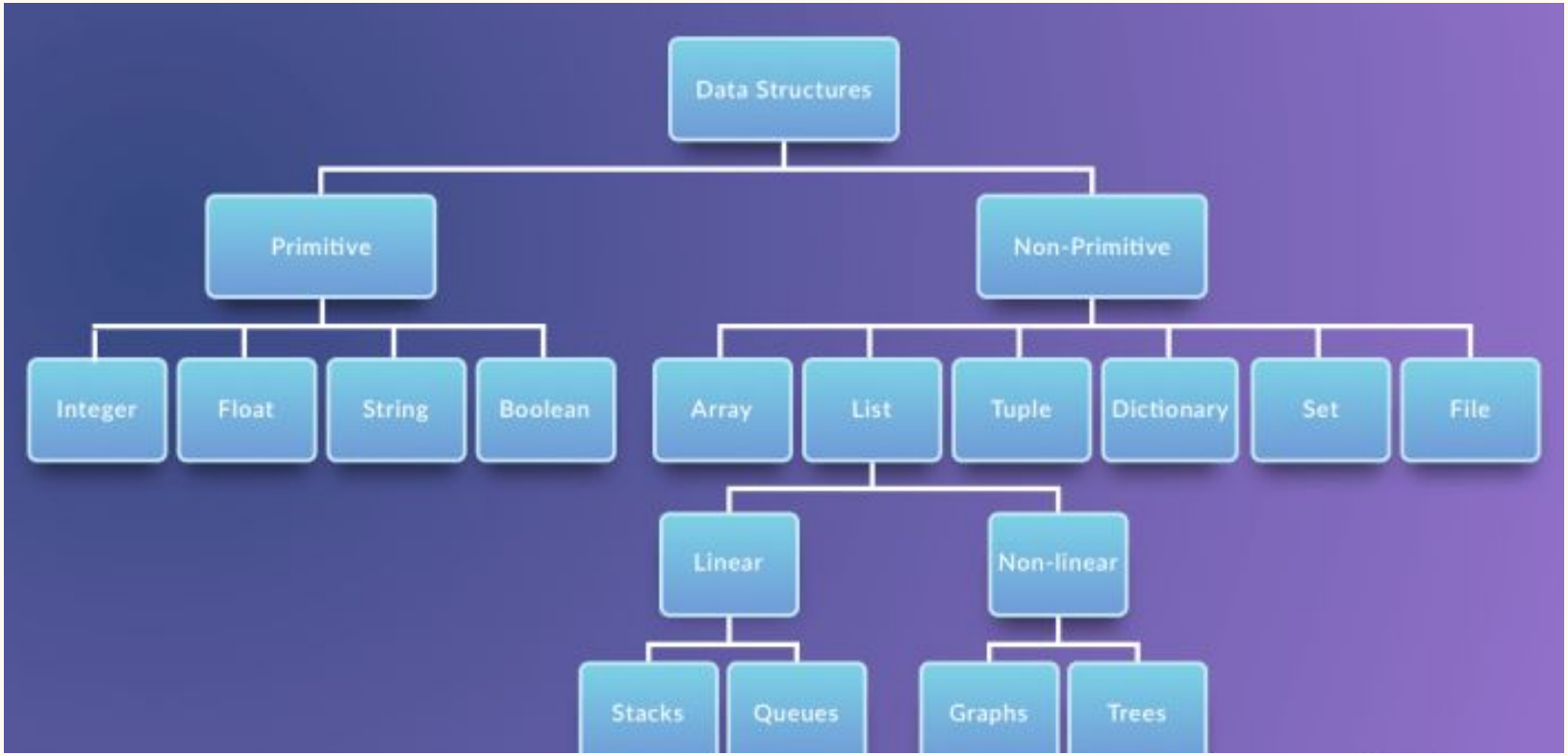
Model for Abstract Data Type (Cont.)

- There is an *interface* between **Application Program** and the **Abstract Data Type** present at the right. ADT consists of the *data structures* and the *functions*(private and public) which are interconnected with each other. Since they are entirely present in the ADT so they are out of the scope of the Application Program.
- **ADT Data Structure**
    - All data that is being processed is maintained in a data structure and its implementation must not to be known to the user.
    - At the same time all the data about the structure should be present inside the ADT because just *encapsulating* the data in ADT is not sufficient.
- **ADT operations**
    - Data is *inserted*, *deleted* and *updated* through the application program via the interface. The functions that are publicly declared are directly accessible otherwise not since only the parameter name and number of such parameters is available to the users.
    - There is a *particular algorithm* for every Abstract Data Type for a specific task to be performed.

Data Structure and ADT

- Data structures are a way of organizing and storing data so that they can be accessed and worked with efficiently. They define the relationship between the data, and the operations that can be performed on the data.
- Data structures help you to focus on the bigger picture rather than getting lost in the details. This is known as data abstraction.
- Now, data structures are actually an implementation of Abstract Data Types or ADT.
- Generally, data structures can be divided into two categories in computer science: primitive and non-primitive data structures. The former are the simplest forms of representing data, whereas the latter are more advanced: they contain the primitive data structures within more complex data structures for special purposes.

Types of Data Structure



Types of Data Structure (Cont.)

Primitive Data Structures

These are the most primitive or the basic data structures. They are the building blocks for data manipulation and contain pure, simple values of a data. Python has four primitive variable types:

- **Integers:** You can use an integer represent numeric data, and more specifically, whole numbers from negative infinity to infinity, like 4, 5, or -1.
- **Float:** "Float" stands for 'floating point number'. You can use it for rational numbers, usually ending with a decimal figure, such as 1.11 or 3.14.
- **Strings:** Strings are collections of alphabets, words or other characters. In Python, you can create strings by enclosing a sequence of characters within a pair of single or double quotes. For example: 'cake', "cookie", etc. You can also apply the + operations on two or more strings to concatenate them
- **Boolean:** This built-in data type that can take up the values: True and False, which often makes them interchangeable with the integers 1 and 0. Booleans are useful in conditional and comparison expressions.

Types of Data Structure (Cont.)

Non-Primitive Data Structures

Non-primitive types are the sophisticated members of the data structure family. They don't just store a value, but rather a collection of values in various formats.

In the traditional computer science world, the non-primitive data structures are divided into:

- Arrays
- Lists
- Tuple
- Dictionary
- Set
- Files

Types of Data Structure (Cont.)

Array

- Arrays in Python are a compact way of collecting basic data types, **all the entries in an array must be of the same data type.** However, arrays are not all that popular in Python, unlike the other programming languages such as C++ or Java.
- In general, when people talk of arrays in Python, they are actually referring to lists. However, there is a fundamental difference between them.
- For Python, arrays can be seen as a more efficient way of storing a certain kind of list. This type of list has elements of the same data type.
- Array in Python can be created by importing array module. **array(*data_type, value_list*)** is used to create an array with data type and value list specified in its arguments.
- The elements stored in an array are constrained in their data type. The data type is specified during the array creation and specified using a type code, which is a single character like the 'i' you see in the example below:

```
import array as arr
arr.array("i",[3,6,9])
type(a) ⇒ array.array
```

Types of Data Structure (Cont.)

List

Lists in Python are used to store collection of heterogeneous items. These are mutable, which means that you can change their content without changing their identity. You can recognize lists by their square brackets [and] that hold elements, separated by a comma ,. Lists are built into Python: you do not need to invoke them separately.

Types of Data Structure (Cont.)

Arrays versus Lists

- Now that you have seen lists in Python, you maybe wondering why you need arrays at all. The reason is that they are fundamentally different in terms of the operations one can perform on them. With arrays, you can perform an operations on all its item individually easily, which may not be the case with lists. Here is an illustration:

  array_char = array.array("u",["c","a","t","s"])

  array_char.tostring()

  print(array_char) ⇒ array('u', 'cats')

- You were able to apply tostring() function of the array_char because Python is aware that all the items in an array are of the same data type and hence the operation behaves the same way on each element. Thus, arrays can be very useful when dealing with a large collection of homogeneous data types. Since Python does not have to remember the data type details of each element individually; for some uses arrays may be faster and uses less memory when compared to lists.

Types of Data Structure (Cont.)

- Traditionally, the list data structure can be further categorised into linear and non-linear data structures. Stacks and Queues are called "linear data structures", whereas Graphs and Trees are "non-linear data structures".

Note: In a linear data structure, the data items are organized sequentially or, in other words, linearly. The data items are traversed serially one after another and all the data items in a linear data structure can be traversed during a single run. However, in non-linear data structures, the data items are not organized sequentially. That means the elements could be connected to more than one element to reflect a special relationship among these items. All the data items in a non-linear data structure may not be traversed during a single run.

Types of Data Structure (Cont.)

Stacks

- A stack is a container of objects that are inserted and removed according to the Last-In-First-Out (LIFO) concept. Think of a scenario where at a dinner party where there is a stack of plates, plates are always added or removed from the top of the pile.
- In computer science, this concept is used for evaluating expressions and syntax parsing, scheduling algorithms/routines, etc.
- Stacks can be implemented using lists in Python. When you add elements to a stack, it is known as a push operation, whereas when you remove or delete an element it is called a pop operation.

Types of Data Structure (Cont.)

Queue

- A queue is a container of objects that are inserted and removed according to the First-In-First-Out (FIFO) principle. An excellent example of a queue in the real world is the line at a ticket counter where people are catered according to their arrival sequence and hence the person who arrives first is also the first to leave.
- Lists are not efficient to implement a queue. As, insertion at the end and deletion from the beginning of a list is not so fast since it requires a shift in the element positions.

Types of Data Structure (Cont.)

Graphs

- A graph in mathematics and computer science are networks consisting of nodes, also called vertices which may or may not be connected to each other. The lines or the path that connects two nodes is called an edge. If the edge has a particular direction of flow, then it is a directed graph, with the direction edge being called an arc. Else if no directions are specified, the graph is called an undirected graph.
- This may sound all very theoretical and can get rather complex when you dig deeper. However, graphs are an important concept specially in Data Science and are often used to model real life problems. Social networks, molecular studies in chemistry and biology, maps, recommender system all rely on graph and graph theory principles.

Types of Data Structure (Cont.)

Trees

- A tree in the real world is a living being with its roots in the ground and the branches that hold the leaves. The branches of the tree spread out in a somewhat organized way. In computer science, trees are used to describe how data is sometimes organized, except that the root is on the top and the branches, leaves follow, spreading towards the bottom and the tree is drawn inverted compared to the real tree.
- To introduce a little more notation, the root is always at the top of the tree. Keeping the tree metaphor, the other nodes that follow are called the branches with the final node in each branch being called leaves. You can imagine each branch as being a smaller tree in itself. The root is often called the parent and the nodes that it refers to below it called its children. The nodes with the same parent are called siblings.

Types of Data Structure (Cont.)

Tuples

- Tuples are another standard sequence data type. The difference between tuples and list is that tuples are immutable, which means once defined you cannot delete, add or edit any values inside it. This might be useful in situations where you might to pass the control to someone else but you do not want them to manipulate data in your collection, but rather maybe just see them or perform operations separately in a copy of the data.

Dictionary

- Dictionaries are exactly what you need if you want to implement something similar to a telephone book. None of the data structures that you have seen before are suitable for a telephone book.
- This is when a dictionary can come in handy. Dictionaries are made up of key-value pairs. key is used to identify the item and the value holds as the name suggests, the value of the item.

Types of Data Structure (Cont.)

Sets

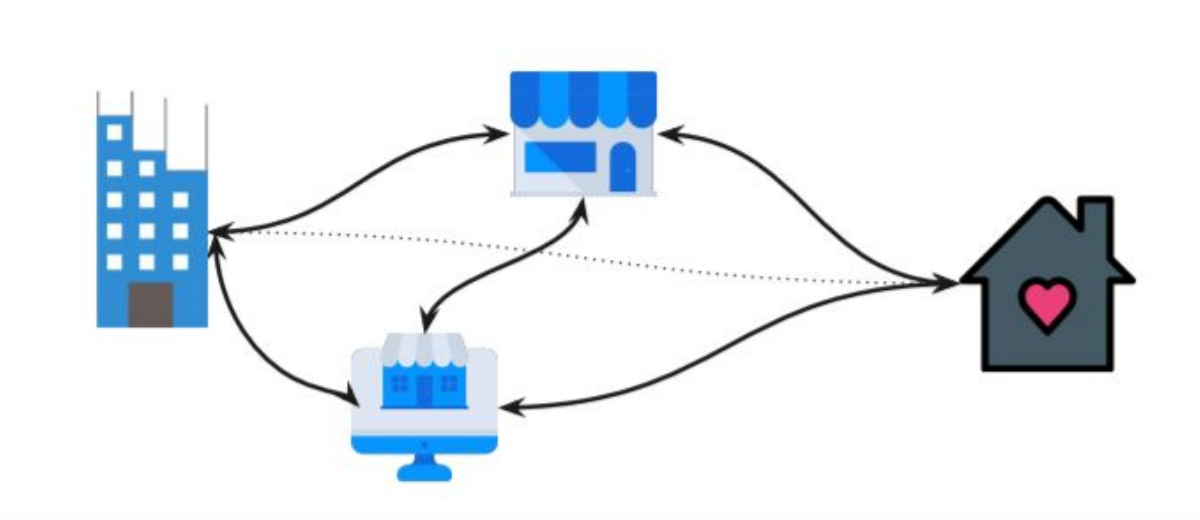
- Sets are a collection of distinct (unique) objects. These are useful to create lists that only hold unique values in the dataset. It is an unordered collection but a mutable one, this is very helpful when going through a huge dataset.

Files

- Files are traditionally a part of data structures. In the data science industry, a programming language without the capability to store and retrieve previously stored information would hardly be useful. Here are some of the basic functions that will help you to work with files using Python:
  - open() to open files in your system, the filename is the name of the file to be opened;
  - read() to read entire files;
  - readline() to read one line at a time;
  - write() to write a string to a file, and return the number of characters written; And
  - close() to close the file.

Algorithm

- Algorithm is the procedure followed for achieving the specific task.



Efficiency and Effectiveness

- Effectiveness ensures that the result is achieved correctly.
- Efficiency ensures that the result is achieved with using minimal energy, memory, time, etc.
- In programming, efficiency depends on :
    - Speed of Computer
    - Size of Input
    - Web Traffic
- These are external factors which can be controlled upto some extent, but varies too much.

Algorithm Efficiency

Internal Complexity

- Internally we can change the program by improving two things :
    - 1. Time taken for execution (Temporal/Time Complexity)
    - 2. Memory required (Space Complexity)
- Optimizing these two will result into faster results with lesser size.

Algorithm Efficiency (Cont.)

Time Complexity

- Time complexity of a program is the number of instructions that a program executes.

```
print("Hello World")
```
→ Time Complexity = 1

```
for i in range(10):
    print(i)
```
→ Time Complexity = 10

Algorithm Efficiency

- **Asymptotic Analysis:** The efficiency and accuracy of algorithms have to be analysed to compare them and choose a specific algorithm for certain scenarios. So,the efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm. The efficiency is measured with the help of asymptotic notations.
- An algorithm may not have the same performance for different types of inputs. With the increase in the input size, the performance will change.
- The study of change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis.

Algorithm Efficiency

- Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as f(n) and may be for another operation it is computed as g(n2). This means the first operation running time will increase linearly with the increase in n and the running time of the second operation will increase exponentially when n increases. Similarly, the running time of both operations will be nearly the same if n is significantly small.

Usually, the time required by an algorithm falls under three types —

- Best Case — Minimum time required for program execution.
- Average Case — Average time required for program execution.
- Worst Case — Maximum time required for program execution.

Algorithm Efficiency(Cont.)

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

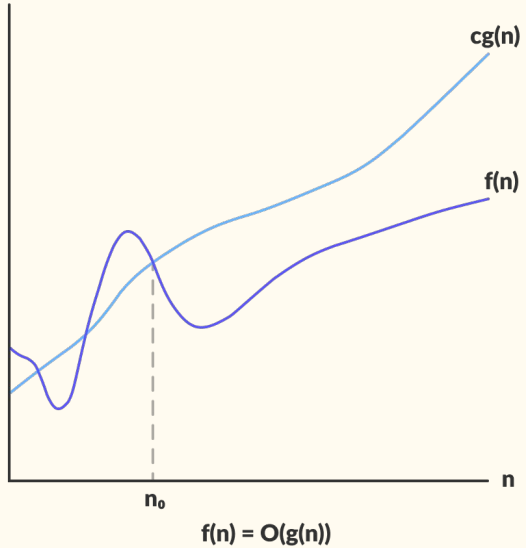
There are mainly three asymptotic notations:

- Big-O notation
- Omega notation
- Theta notation

Algorithm Efficiency(Cont.)

Big-O Notation (O-notation)

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.



Algorithm Efficiency(Cont.)

Big-O Notation (O-notation)

$O(g(n)) = \{ f(n)$: there exist positive constants c and n0

such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n0 \}$

The above expression can be described as a function f(n) belongs to the set O(g(n)) if there exists a positive constant c such that it lies between 0 and cg(n), for sufficiently large n.

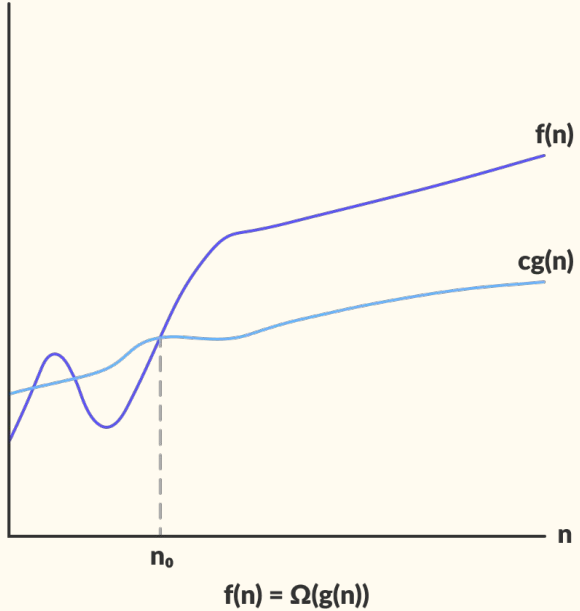
For any value of n, the running time of an algorithm does not cross the time provided by O(g(n)).

Since it gives the worst-case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst-case scenario.

Algorithm Efficiency(Cont.)

Omega Notation (Ω-notation)

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.



Algorithm Efficiency(Cont.)

Omega Notation (Ω-notation)

$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n0

such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n0 \}$

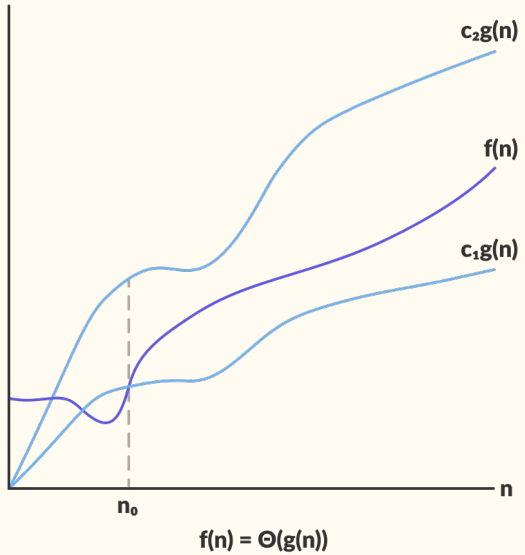
The above expression can be described as a function f(n) belongs to the set Ω(g(n)) if there exists a positive constant c such that it lies above cg(n), for sufficiently large n.

For any value of n, the minimum time required by the algorithm is given by Omega Ω(g(n))

Algorithm Efficiency(Cont.)

Theta Notation (Θ-notation)

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.



Algorithm Efficiency(Cont.)

Theta Notation (Θ-notation)

For a function g(n), Θ(g(n)) is given by the relation:

$\Theta(g(n)) = \{ f(n)$: there exist positive constants c1, c2 and n0

such that $0 \leq c1g(n) \leq f(n) \leq c2g(n)$ for all $n \geq n0 \}$

The above expression can be described as a function f(n) belongs to the set Θ(g(n)) if there exist positive constants c1 and c2 such that it can be sandwiched between c1g(n) and c2g(n), for sufficiently large n.

If a function f(n) lies anywhere in between c1g(n) and c2g(n) for all $n \geq n0$, then f(n) is said to be asymptotically tight bound.

Algorithm Efficiency (Cont.)

Big-O Notation

- Big O Notation is used to depict an algorithm's growth rate. It checks the algorithm's performance as input size grows.
- *O Notation is useful in comparing algorithms

It will take at most N^2 steps for input size N

$A = O(n^2)$ $B = O(2^n)$

It will take at most 2^N steps for input size N

Algorithm Efficiency (Cont.)

Common Asymptotic Notations

Following is a list of some common asymptotic notations:

Name	Time Complexity
Constant Time	O(1)
Logarithmic	O(log n)
Linear	O(n)
n log n	O(n log n)
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Polynomial	$n^{O(1)}$
Exponential	$2^{O(n)}$

Algorithm Efficiency (Cont.)

Constant Time — O(1)

An algorithm is said to have a constant time when it is not dependent on the input data (n). No matter the size of the input data, the running time will always be the same. For example:

```
if a > b:
    return True
else:
    return False
```

Algorithm Efficiency (Cont.)

- Now, let's take a look at the function **get_first** which returns the first element of a list:

```
def get_first(data):

    return data[0]

data = [1, 2, 9, 8, 3, 4, 7, 6, 5]
print(get_first(data))
```

- Independently of the input data size, it will always have the same running time since it only gets the first value from the list.
- An algorithm with constant time complexity is excellent since we don't need to worry about the input size.

Algorithm Efficiency (Cont.)

Linear Time — O(n)

An algorithm is said to have a linear time complexity when the running time increases at most linearly with the size of the input data. This is the best possible time complexity when the algorithm must examine all values in the input data. For example:

```
for value in data:
    print(value)
```

Algorithm Efficiency (Cont.)

Linear Time

- We want to know how many times the body of the loop is repeated in the following code:

```
for i in range(1000):
    Body of loop
```

- Assuming i is an integer, the answer is 1000 times. The number of iterations is directly proportional to the loop factor, 1000. The higher the factor, the higher the number of loops. Because the efficiency is directly proportional to the number of iterations, it is

$$\mathbf{f(n) = n}$$

Algorithm Efficiency (Cont.)

Finding the complexity

```
for i in range(n) :
    print('HI')
for j in range(n):
    print('TANVI')
```

It will take C1 +nC2 +C3 +nC4 time.
We only write the higher exponent variable.
We write it as O(n).

Algorithm Efficiency (Cont.)

Quadratic Time — O(n²)

An algorithm is said to have a quadratic time complexity when it needs to perform a linear time operation for each value in the input data, for example:

```
for x in data:
    for y in data:
        print(x, y)
```

Algorithm Efficiency (Cont.)

Finding the complexity

```
for i in range(n) :
    print('HI')
    for j in range(n):
        print('HEY')
```

Total executions : n(C1 + nC2) => nC1 + n^2C2
We only write the higher exponent variable.
We write it as O(n^2)

Algorithm Efficiency (Cont.)

Important Notes

It is important to note that when analyzing the time complexity of an algorithm with several operations we need to describe the algorithm based on the largest complexity among all operations. For example:

```
def my_function(data):
    first_element = data[0]
    for value in data:
        print(value)
    for x in data:
        for y in data:
            print(x, y)
```

Even that the operations in 'my_function' don't make sense we can see that it has multiple time complexities: $O(1) + O(n) + O(n^2)$. So, when increasing the size of the input data, the bottleneck of this algorithm will be the operation that takes $O(n^2)$. Based on this, we can describe the time complexity of this algorithm as $O(n^2)$.

Algorithm Efficiency (Cont.)

Logarithmic Time — O(log n)

Logarithmic Complexity means the algo. has a log factor.
It is generally found when the problem is divided into fractions (usually in ½).

An algorithm is said to have a logarithmic time complexity when it reduces the size of the input data in each step (it don't need to look at all values of the input data).

Algorithm Efficiency (Cont.)

```
for index in range(0, len(data), 3):

    print(data[index])
```

Algorithms with logarithmic time complexity are commonly found in operations on binary trees or when using binary search.

Algorithm Efficiency (Cont.)

- However, the answer is not always as straightforward as it is in the above example. For instance, consider the following loop. How many times is the body repeated in this loop? Here the answer is 500 times.

```
for i in range(0,1000,2):
    Body of loop
```

- In this example the number of iterations is half the loop factor. Once again, however, the higher the factor, the higher the number of loops. The efficiency of this loop is proportional to half the factor, which makes it

$$\mathbf{f(n) = n/2}$$

- If you were to plot either of these loop examples, you would get a straight line. For that reason they are known as linear loops.

Algorithm Efficiency (Cont.)

Quasilinear Time — O(n log n)

An algorithm is said to have a quasilinear time complexity when each operation in the input data have a logarithm time complexity.

It is commonly seen in sorting algorithms (e.g. mergesort, timsort, heapsort).

Algorithm Efficiency (Cont.)

Exponential Time — O(2^n)

An algorithm is said to have an exponential time complexity when the growth doubles with each addition to the input data set.

Example of an exponential time algorithm is the recursive calculation of Fibonacci numbers:

```
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

Algorithm Efficiency (Cont.)

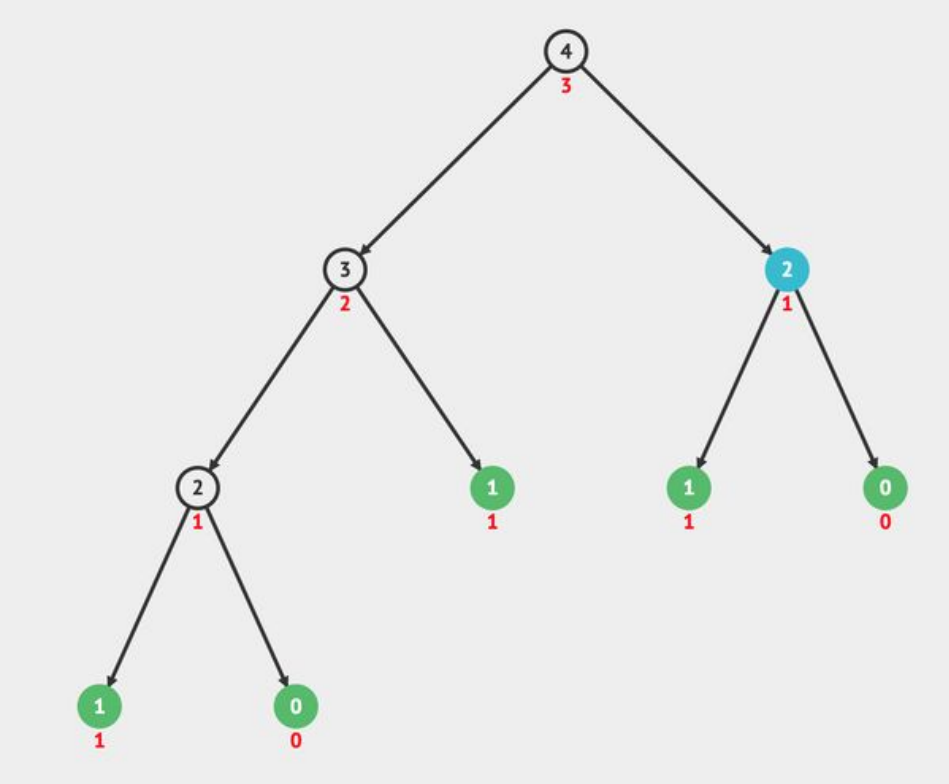
A recursive function may be described as a function that calls itself in specific conditions. As you may have noticed, the time complexity of recursive functions is a little harder to define since it depends on how many times the function is called and the time complexity of a single function call.

It makes more sense when we look at the recursion tree.

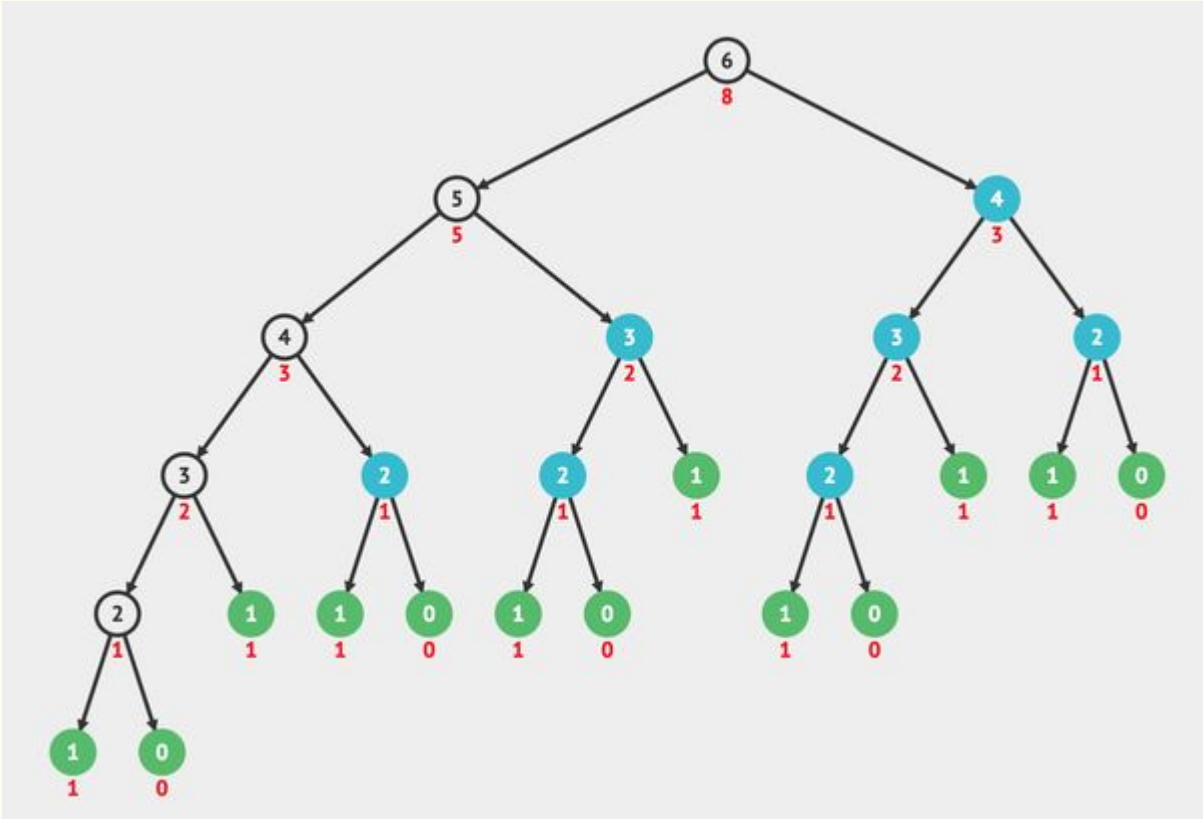
The following recursion tree was generated by the Fibonacci algorithm using $n = 4$:

Note that it will call itself until it reaches the leaves. When reaching the leaves it returns the value itself.

Algorithm Efficiency (Cont.)



Algorithm Efficiency (Cont.)



Algorithm Efficiency (Cont.)

Summarized Rules

- Simple Constant Statements - O(1)
- Iteratives(no. of repetitions) - O(n)/O(n^2)/O(n^3)/etc.
- if/elif/else - take the worst case
- Subsequent statements - add the complexities
- Nested statements - multiply the complexities.
- In case of Function calls, time is taken only in the code of function body and setting up the parameters(which is O(1), if a large structure is copied it is O(n)).

Properties of Asymptotic Notation

Assuming f(n), g(n) and h(n) be asymptotic functions the mathematical definitions are:

1. If $\mathbf{f(n) = \Theta(g(n))}$, then there exists positive constants c1, c2, n0 such that $\mathbf{0 \leqslant c1.g(n) \leqslant f(n) \leqslant c2.g(n)}$, for all $n \geq n0$
2. If $\mathbf{f(n) = O(g(n))}$, then there exists positive constants c, n0 such that $\mathbf{0 \leqslant f(n) \leqslant c.g(n)}$, for all $n \geq n0$
3. If $\mathbf{f(n) = \Omega(g(n))}$, then there exists positive constants c, n0 such that $\mathbf{0 \leqslant c.g(n) \leqslant f(n)}$, for all $n \geq n0$

Properties of Asymptotic Notation

Reflexivity:

If f(n) is given then

f(n) = O(f(n))

Example:

If f(n) = n^3 ⇒ O(n^3)

Similarly,

f(n) = Ω(f(n))

f(n) = Θ(f(n))

Properties of Asymptotic Notation

Symmetry:

f(n) = Θ(g(n)) if and only if g(n) = Θ(f(n))

Example:

If f(n) = n^2

and g(n) = n^2

then f(n) = Θ(n^2) and g(n) = Θ(n^2)

Properties of Asymptotic Notation

Transistivity:

f(n) = O(g(n)) and g(n) = O(h(n)) ⇒ f(n) = O(h(n))

Example:

If f(n) = n, g(n) = n^2 and h(n) = n^3

⇒ n is O(n2) and n2 is O(n3) then n is O(n3)

Properties of Asymptotic Notation

Transpose Symmetry:

f(n) = O(g(n)) if and only if g(n) = Ω(f(n))

Example:

If f(n) = n and g(n) = n^2 then n is O(n^2) and n^2 is Ω(n)

Properties of Asymptotic Notation

Observations:

If f(n) = O(g(n)) and

d(n) = O(e(n))

Then f(n) + d(n) = ?

Ex:

f(n) = n => O(n)

d(n) = n^2 => O(n^2)

f(n) + d(n) = n + n^2

= O(n^2)

Therefore, f(n) + d(n) = O(max(g(n),e(n)))

Properties of Asymptotic Notation

Observations:

If $f(n) = O(g(n))$ and

$d(n) = O(e(n))$

Then $f(n) * d(n) = O((g(n),*e(n)))$

Algorithm Efficiency (Cont.)

How log comes?

e.g. Binary Search Problem is divided by 2 until element is found.
In the worst case it divides until size is left 1.
Let's suppose it takes k steps to divide till 1.

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

Taking log both sides

$$\log_2(2^k) = \log_2 N$$
$$k*\log_2(2)=\log_2 N \qquad [\log_a a = 1]$$
$$k=\log_2 N$$

k was the no. of executions, i.e. Time Complexity

Algorithm Efficiency (Cont.)

Best, Average & Worst Case

for e.g. Finding a number in a list.

Best Case : 0th index

Average Case : In Middle

Worst Case : last index

Thank You