DATA STRUCTURES AND ALGORITHMS

DATA STRUCTURES:

Arranging the data in different ways that can be used effectively (structuring the data)
Ex: Queue DS → organizing data in the form of queue FIFO (cinema tickets)

Stack DS → organizing data in the form of stack FILO (stacking books)

ALGORITHM: Set of Instructions to complete a task (EX: preparing tea, cooking, building house, Graph Algo – short distance path)
Good Algorithm: Correctness, Efficiency

Example why DSA is important:
in a library books are data and arranging them in the library is using data structures and finding books is based on an algorithm.

Types of Data Structures:
Primitive: Integer, String, Float, Boolean
Non-Primitive(which can be further divided):

Linear(data is sequential or linear order): List, Tuple, Array,[up to here pre def], Linked list, Stack, Queue
Non-Linear(data is not in sequential or linear order): Set, Dictionary[pre defined], Tree, Graph

Algorithm Types:
Sorting, Searching, Graph(Graph like data structure),

Dynamic programming(dividing into sub problem and getting the result),

Divide and conquer(dividing into sub problem and solving individually and combining the results), Recursive(calling same function)

BIG O:

It is a metric(also language) used to describe the performance and complexity of algorithm, it helps to describe how much time needed for the algorithm if input grows

Time Complexity how run time increases if input size increases
Example:
downloading 1gb file from internet time increases if file size increases, what if it data is in hard disk(from here take to home) here time is constant doesn't matter with file size

BIG O used to judge an algorithm like is it efficient(here we are measuring no of operations to run the algorithm for large amount of data not how much time since it differs from system to system)

Space Complexity: how much memory algorithm occupies while running.

NOTATIONS: (omega, theta, big O)

Example:
if you take the car the mileage depends on where it drives, if it is in traffic it may give 20miles, if it is in city a bit high, in highway a bit more high
so the best case is represented with "omega"(complexity at least more than best case)

The average is represented with “theta”(complexity within bounds of worst and best case)

The worst case is represented with “Big O”(complexity less or equal to the worst case)

O(1) → constant time complexity

Examples: here the no of operation is 1, for example multiplication operation function, grabbing random card from deck of cards(1 operation)
the Graph line looks like a flat line
here, time complexity is constant irrespective of input data

O(N) → Linear Time Complexity

here time complexity is gonna grow with respective to input data
Examples: loop function (printing o to n) here no of operations increases with increase in size of n, searching a particular card from deck of cards here card searching it is directly related to no of deck of cards

The Graph line looks like a linear

Cases where O(N) can be faster is for smaller input values for example O(1) is for searching element in an array and O(N) is sum function with just array size is 1, here O(N) may be faster compared to O(1) because of smaller inputs.

Dropping Constant Factor→ O(2n) = O(n)

As n approaches infinity doesn’t matter what is the constant factor

Big O is not really for capturing every single detail

“Big O notation is used for comparing algorithms. When comparing, we’re more interested in how the algorithm scales with larger inputs rather than the exact number of operations”

O(N^2) → Quadratic Time Complexity

This happens mostly in nested loops for example for loop inside for loop where first for loop runs for n times and 2nd n times for every time first loop runs so n^2.
Example:
searching number 3 from deck of all cards, and then again searching number 4 from all cards it is like u are doing same operation for different number every time, it is worst time complexity since ur number of operations are going to increase quadratically for rise in input size,

For O(n^3) or more we write it as O(n^2) → since we are only try to get relationship of how time complexity change with input and not going to get exact no of operations.

O(log n) → logarithmic time complexity

The concept of this is to divide the input half and then again half and half until we get our desired output(just like binary search)

Examples:
Finding 1 in ordered 8 list,, it took three operations for input size 8 to find 1 which is equal to logn,,

same goes for finding card in deck of ordered cards

graph is flat but above O(1) better than O(n^2) and O(n)

Note: O(2^n): every function is performing 2 operations.

Dropping Non Dominant Terms
O(n^2 + n) → here we can drop O(n) and can represent it as O(n^2) since if we take n=100 , O(n) doesn't have any value compare to O(n^2)
example: nested and normal loop

Note: sometimes we can use substitution also and check the time complexity If it is kind of complex

Space Complexity: how much memory the algorithm is using as size of input grows

For recursive function, it is going to create a stack one by one since it is calling same function and previous call is waiting for next call return value which means stack is created and for n calls require O(n) Space complexity
on other hand if stack is not created it just require O(1) Space Complexity

Different Inputs:

Not Every time for time complexity is O(n) it depends on input value for a and b it may be O(a+b) for do this and after do that kind of logics,,,, and may be O(a*b) for do this while doing that kind of logic

5 Rules

No	Description	Complexity
Rule 1	Any assignment statements and if statements that are executed once regardless of the size of the problem	O(1)
Rule 2	A simple "for" loop from 0 to n (with no internal loops)	O(n)
Rule 3	A nested loop of the same type takes quadratic time complexity	$O(n^2)$
Rule 4	A loop, in which the controlling parameter is divided by two at each step	O(log n)
Rule 5	When dealing with multiple statements, just add them up	

ARRAY:

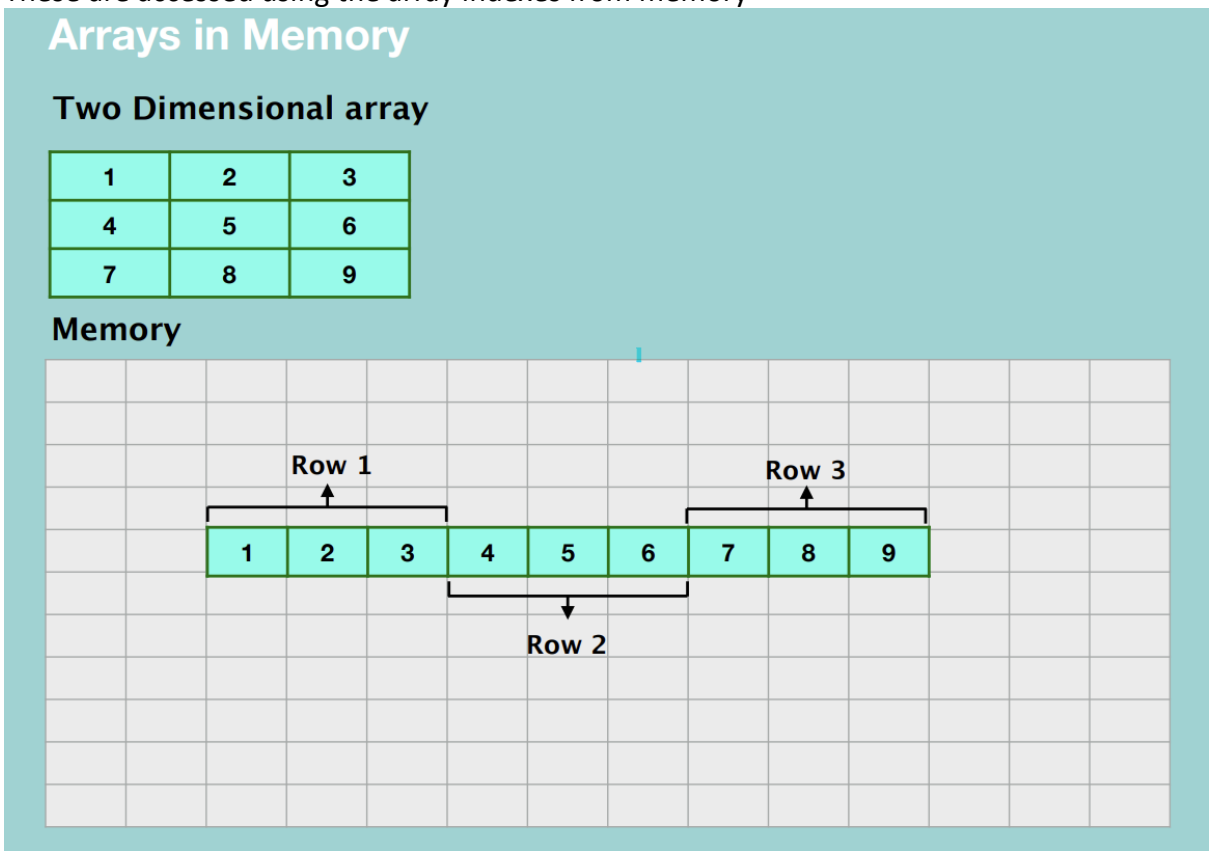
In python there is no concept of arrays we do have list but list can store all data types and mutable
arrays are of of same datatype, contiguous, can't increase size we can use arrays in python using array and numpy module

1-D array: single row and multiple columns
2-D array: multiple rows and multiple columns
3-D array: just like cube with numbers here [depth][row][column] → it is a combination of 2-D with depth

1 and 2-D array are broadly used

Representation in memory:

All elements in the array are represented in contiguous side by side in memory, doesn't matter even 2-dimensional and 3-d also represented in same way.
These are accessed using the array indexes from memory



Creation of array: using numpy and array module (parameters, datatype and data)
Insertion in array: parameters .insert(index, value) → when element is inserted the rest of

values shift towards right,,, time complexity O(n), space complexity O(1)
note: here we are creating new array after insertion
Traversing: for loop
accessing value: using index
Searching: Linear Search
Deletion: after deletion element remaining move to left shift → .remove(value)

Operation	Time complexity	Space complexity
Creating an empty array	O(1)	O(n)
Inserting a value in an array	O(1)/O(n)	O(1)
Traversing a given array	O(n)	O(1)
Accessing a given cell	O(1)	O(1)
Searching a given value	O(n)	O(1)
Deleting a given value	O(1)/O(n)	O(1)

Two dimensional array:

Creation: using numpi → np.array([[],[]]) → more rows and columns,
Insertion: np.insert(array_name, index, [](array), axis=column,row),
accessing: array[m][n],,,, m is row, n is column, tips=len(array) row length, for column arr[0]
traversing: using 2 for loops
searching: for basic we are going with linear search
deletion: same format just we used for insert np.delete(array_name, index, axis)

Time and Space Complexity in 2D Arrays

Operation	Time complexity	Space complexity
Creating an empty array	O(1)	O(mn)
Inserting a column/row in an array	O(mn)/O(1)	O(1)
Traversing a given array	O(mn)	O(1)
Accessing a given cell	O(1)	O(1)
Searching a given value	O(mn)	O(1)
Deleting a given value	O(mn)/O(1)	O(1)

Pros and cons:

used for same data type, and efficient for random accessing → pros
not useful to store another data type,and for every insertion or deletion it is creating new array copying existing items which is in efficient and bad performance

LIST:

List in python is mix of all data types unlike homogeneous data types as like in array ,, defined using square brackets and can also have nested list.

Operations:

Creation: just use square bracket

Memory allocation: Dynamic array pattern , it can grow and shrink based on inputs, it stores pointers inplace of continuous memory created and it points to the original values, so lists are not fixed size like arrays
Traversal: For loop and in operation and len functions are kind of general usefull parameters, and can also do operations whenever required.

Update/insert: .insert(index, value), .extend, .append

Slicing and deletion: general slicing operation and deletion using .pop(index), .remove(element value), del function

Searching the element: in operator, enumerate function(track of index also here)

Operations: +, *, len(), max(), min(), sum(), ==(comparison oper), .sort()

Conversions: list(), .split(returns list), .join(returns string), .sort()

Disadvantages of list:

Most of the list methods returns none

There are 2 to 3 ways to do the same task (removing an element .pop, .remove, .insert, .append)

List vs arrays:

They have some similar features like both are mutable, can be accessed using indexing, can be traversed,,,,,

But arrays are much more optimized to do arithmetic similar kind of operations,, lists are not much optimized,,, may be because arrays are optimized with only same kind of data type compared to list.

Time and Space Complexity in Python Lists

Operation	Time complexity	Space complexity
Creating a List	O(1)	O(n)
Inserting a value in a List	O(1)/O(n)	O(1)
Traversing a given List	O(n)	O(1)
Accessing a given cell	O(1)	O(1)
Searching a given value	O(n)	O(1)
Deleting a given value	O(1)/O(n)	O(1)

Comprehension List:

List comprehension in Python is a concise way to create new lists based on existing lists based on existing lists or other iterable objects.

squares = [x**2 for x in range(10)]

here square list is created where output is squares of numbers from 0 to 9

Array problems:

Diagonal sum, swapping the a[i][j] with a[j][i]

Dictionary:

follows the key, value pair and it is unordered non linear data structure which doesn't follow any sequence, value is mutable not keys

Follows hash table structure

Live Example: original dictionary, token tickets in any store

Creation: using {}, or dict() ,, can create using tuple also dict([(key,value),(key,value)])

Key, value should be in quotes (key should be in quote but its not manda for value)

Memory allocation: using hash function and stores in Hash tables and gets one index and store the dict item in that index, and if another element from the dictionary collides with the previous index it uses a linked list attaching to the previous item in same index, and these indexes hash function is like array contiguous from 0 index

Update/insertion: direct assigning or using **update({key:value, key2:value}), but append wont work**

Traversal: for loop and can access the value using dict[key]

Searching a value: Linear Search (using basic traversing in the dictionary)

Deletion: using .pop(key, default value if key not found) return value of item, del function, .clear() deletes every element from memory

Operations:

.clear(), .copy(), .fromkeys([sequence], value)(insertion in dict), .get(key) -> this will return value if key is present else return none, used to remove key error from normal value accessing method

.items() → returns tuple pair

.keys() → returns all keys in the dictionary

.values() → returns list of values

Key,value = .popitem() → removes last item and return the item pair

.setDefault() → add items in the dictionary previous

IN operation, len(), ==

all() → checks all keys are true , (checking trueness of keys)

any() → all keys true, one value true, all false(boolean or 0) → false

sorted() → sort all keys

Dictionary Comprehension:

New_dict = {key:value for key,value in dict.items()} → general syntax

Time and Space Complexity in Python Dictionary

Operation	Time complexity	Space complexity
Creating a Dictionary	O(len(dict)	O(n)
Inserting a value in a Dictionary	O(1)/O(n)	O(1)
Traversing a given Dictionary	O(n)	O(1)
Accessing a given cell	O(1)	O(1)
Searching a given value	O(n)	O(1)
Deleting a given value	O(1)	O(1)

Tuple:

Tuple is a ordered, immutable data type(read only) which means you can't change value of it once it is declared, allows duplicates

Creation: using () or tuple(), for single element In tuple create using comma.

Accessing: same as list indexing and slicing

Traversing: same as list using for loop

Searching: same as list using IN operator and using linear search

Operations(creates new tuple doesn't changes existing tuple): same as list +(for concatenation), *(for repetition), .count(),.index(), len(), min(), max(), .join()

A tuple is same as a list but just immutable and declared in parenthesis

Note:

Tuples can be stored In a List and a List can be stored in a Tuple without any problem,

we generally use tuples for heterogeneous data types and lists for homogenous(same) data types

Time and Space Complexity in Python Tuples

Operation	Time complexity	Space complexity
Creating a Tuple	O(1)	O(n)
Traversing a given Tuple	O(n)	O(1)
Accessing a given element	O(1)	O(1)
Searching a given element	O(n)	O(1)

SET:

Behaves sames like list and tuples but unlike them set has unique values and it is un ordered,in sets we can't have list or tuple(nested) since set does not allow mutable datatypes like list and tuples inside it(since sets works on hash values)

General set operations: intersection, union, subtraction, symeteric differ(oppo of intersec)

Creation: {} just curly brackets(but with values) or set() for empty (note don't use {} it considers it as dictionary)

Memory allocation: same as dictionary , uses hash function and take values in sets as keys and they are allocated at different places based on the result of hash function

Searching: IN operator

Operation's: .add(), .remove(),.clear(),.pop()->removes random,

.union(), .intersection(), .difference(), .symetricdifference()

.intersection_update(), .issubset(), .issuperset(), isdisjoint()(True if nothing in common, False if common)

Lambda Function:

this is used to replace shorter functions in python:

Syntax: **lambda x,y(input): x+y(output)** (input parameters and output return value