Contents

        vi. Dictionary comprehension
    d. Sets
        i. Definition
        ii. .add()
        iii. .remove()
        iv. .update()
        v. Union
        vi. Intersection
        vii. Difference
        viii. Symmetric difference
    e. Strings
        i. initializing (‘, “, ‘’’)
        ii. .index()
        iii. .upper()
        iv. .lower()
        v. .title()
        vi. .startswith()
        vii. .split()
        viii. .join()
        ix. .format() vs f-strings
13. Searching
    a. Linear Search
        i. enumerate()
    b. Binary Search
        i. Time complexity comparison
14. Sorting and Time complexity
    a. Time complexity
        i. Big O notation
    b. Sorting
        i. Bubble sort
        ii. Selection sort
        iii. Insertion sort
15. Recursion
    a. Definition
    b. Merge sort
16. Combinatorics
    a. nPr
    b. nCr
    c. Some standard results

Intermediate-1

Hierarchy of languages (Up to down)

- High-level languages (Python, JavaScript, etc)
- Mid-level languages (C, C++, Java, etc)
- Assembly code
- Machine code (byte code)
  Going ↑ in hierarchy => code simplicity ↑ but execution time ↓

Interpreter vs Compiler

- **Compiler**: Convert entire code to byte code which later get executed by cpu
- **Interpreter**: Read and execute the code line by line

Python (About)

- Dynamically typed language
- Interpreted language
- Supports: First Class Citizens (FCC)

Data Types

- **Primitives (static) dtypes:** Bound to some memory limit (1-byte store max int value of 255)
- **FCC**: Not bound to any memory limit, can store as much as available RAM
- Anything with value is an object
-

Getting the type of any object	>> type(3) # `int`
`isinstance`(a,b) can tell whether a is the type of b or not?	>> instance(3, int) # `True`
Complex number are supported in python	>> a = 1 + 2j >> type(a) # `complex`

Arithmetic Operators

Addition	>> 1 + 2 # `3`
Subtraction	>> 1 - 2 # `-1`
Division	>> 10 / 2

	# 5.0
Floor division operator (largest integer less than the result)	>> -8 // 3 # -3
Exponentiation operator	>> 2 ** 10 # 1024
Modulus operator: **+ve number:** returns remainder after division **-ve numbers**: a % b => b - (a % b)	# for positive numbers >> 8 % 3 # 2 # for negative numbers >> -8 % 3 # -1

Identifiers

- Names that are given to variables
- Rules:
    - Should start with alphabet or _
    - Can contain alphanumeric symbol or _ in between

_ symbol in python automatically stores the value of last executed expression

Mutable/Immutable

- **Mutable**: Whose value can change
- **Immutable**: Whose value cant change
-

Mutable dtypes: list, set, dictionary

-

Getting address of memory of variable	>> id(a) # 11256224
Deleting memory location of a variable	>> del a

If-else

-

```
if condition2:
    print("do this")
elif condition2:
    print("or this")
else:
    print("else this")
```

Iteration protocol

- ***Iterable***: Data structrure with atomic elements which can be iterated.
- ***Iterator***: variable used to store those atomic elements.
- ***Iteration block***: Entire Body of the loop

Getting numbers from [a, b) - Can also define step_size as 3rd argument	>> range(a, b) `# a, a+1, a+2, ……, b-1` >> range(1, 10, 2) `# 1, 3, 5, 7, 9`
Converting iterable to iterator	>> a = “this is a string” >> iter_a = iter(a)
Getting next value of an iterator	>> next(iter_a) `# ‘t’`

Loops

for loop	`for i in range(1, 11):` `    print(“do this”)`
While loop	`while condition:` `    print(“do this”)`

Functions

Definition	`def function_name(arg1, arg2):` `    print(“do something”)` `    return “output”`
Calling function	`>> function_name(1, 2)` `# “output”`

Positional argument: value reaching argument positionally	`>> def x(a,b):` `     pass` `>> x(3,4)`

Keywords argument: values passing according to keys	`>> def x(a,b):` `       pass` `>> x(a=3, b=4)`

Data Structures

List:

- Heterogeneous collection of objects
-

Initializing list	`>> a = ["abc", 23, 45.8, True]`

-

indexing	>> a[0] # `“abc”`
Negative indexing	>> a[-1] # `True`
Size of list	>> len(a) # `4`
Slicing: [`start : end : jump`]	>> a[1:3] # `[23, 45.8]` >> a[-1 : 1 : -1] # `[True, 45.8]`
+ operator	>> [1,2,3] + [4,5] # `[1,2,3,4,5]`
* operator	>> [1,2,3] * 2 # `[1,2,3,1,2,3]`
Iteration over list	`for element in [1,2,3]:` `    print(element, end=” ”)` `# 1 2 3`
Appending element in the last	>> [1,2,3].append(4) # `[1,2,3,4]`
Extending list	>> [1,2,3].extend([4,5,6]) # `[1,2,3,4,5,6]`
Popping element using index	>> [1,2,3].pop(0) # `1` `# list now is: [2,3]`
Removing element using value	>> [1,2,3].remove(1) # `list now is: [2,3]`
Checking if some value present in list	>> 1 in [2,3,1]

	# True
List comprehension	>> [k*2 for k in range(2,6) if k < 5] # [4,6,8]

Tuples

Initializing	>> a = (1,2,3,4)
immutability	>> a[0] = 10 # Error
Packing/Unpacking - need exact amount of variables as elements in tuple - unpacking can be used with `list` but not packing	>> a, b, c = (2, 3, 4) >> print(a, b, c) # 2, 3, 4

Dictionaries

Initializing : { key = value } Only immutable data structures can be key	`d1 = {` `"name" : "Bipin",` `"age": 5000,` `"subject": ["python", "JS"]` `}`
Accessing key: - If key not exist then \`d[key]\` will through error while \`d.get(key)\` will return `None`	>> `d1["name"]` # “Bipin” >> `d1.get("name")` # “Bipin”
Deleting a key-value pair	>> `d1.pop("age")`
Updating dictionary: changing existing values of keys or adding more key-value pairs	`>> d= {"lang": "JS"}` `>> d.update({"lang": "Python", "awesomeness": float("inf")})` `>> print(d)` # `{'lang': 'Python', 'awesomeness': inf}`
Iterating over dictionary	# iterating over keys `for key in d1:` `    print(key)` # iterating over values `for key in d1.values():` `    print(key)`

	# iterating over both keys and values `for key, value in d1.items():` `    print(key, value)`
Dictionary comprehension	>> `{i:i**2 for i in range(5)}` # `{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}`

Sets

• unordered • non-indexable • mutable • stores unique values (repeated gets removed) • iterable • can only contain immutable data structures	>> `s1 = {1,2,3,3,3,4,4,5,5,}` `# s1 = {1,2,3,4,5}`
Adding values to set	>> s1.add(10)
Removing values from set	>> s1.remove(10)
Updating set	>> s1.update([1,2,3,6]) # `s1 = {1,2,3,4,5,6}`
Union	>> `s1 = {1,2}` >> `s2 = {1,3,4,5}` >> s1 | s2 # `{1,2,3,4,5}`
Intersection	>> `s1.intersection(s2)` `# {1}` `>> s1 & s2` `# {1}`
Difference	`>> s1.difference(s2)` # `{2}` `>> s1 - s2` `# {2}`
Symmetric difference	`>> s1.symmetric_difference(s2)` # `{2,3,4,5}` `>> s1 ^ s2` `# {2,3,4,5}`

Strings

Initializing: ***Whitespaces are preserved in string with ''''***	`>> s1 = "This is a string"` `>> s2 = 'this is a string'` `>> s3 = '''  this is a string  '''` `>> s = "python"`
Getting index of character	`>> s.index('t')` `# 2`
Captilizing all the characters	`>> s.upper()` `# "PYTHON"`
Lowercasing all characters	`>> s.lower()` `# "python"`
Title case	`>> s.title()` `# "Python"`
whether a string starts with a particular string or not	`>> s.startswith("THE")` `# False`
Splitting string into list via some `delimiter` character.	`>>s = "this is a string"` `>>s.split(" ")` `# ["this", "is", "a", "string"]`
Joining list element into a string via some joining character	`>> "-".join(s.split(" "))` `# "this-is-a-string"`
Combining strings	`>> age = 20` `>> "age = {}".format(age)` `# "age = 20"` `# using f-strings` `>> f"age = {age}"` `# "age = 20"`

Searching

Linear Search

Linear Search: Going to each element one by one and comparing with target	`def linear_search_enum(li, target):` ` for idx, ele in enumerate(li):` `   if ele == target:`

Time complexity: O(N) Space Complexity: O(1) # `N`: length of list	`        return idx` `    return -1` `a = ["a", "b", "c"]` `linear_search(a, "c")` `# 2`

Binary Search

• Sort the array and keep the start and **end** pointer to the first and last element of array resp. • Find the **mid** element • Compare **target** with **mid** element • If target found then return the mid element, else adjust **start** and **end** Time complexity: O(logN) Space complexity: O(1)	`def binary_search(a, target):` ` start, end = 0, len(a)-1` ` while start <= end:` `   mid = (start + end)//2` `   if a[mid] == target:` `     return mid` `   elif a[mid] > target:` `     end = mid - 1` `   else:` `     start = mid + 1` ` return -1`

Time Complexity

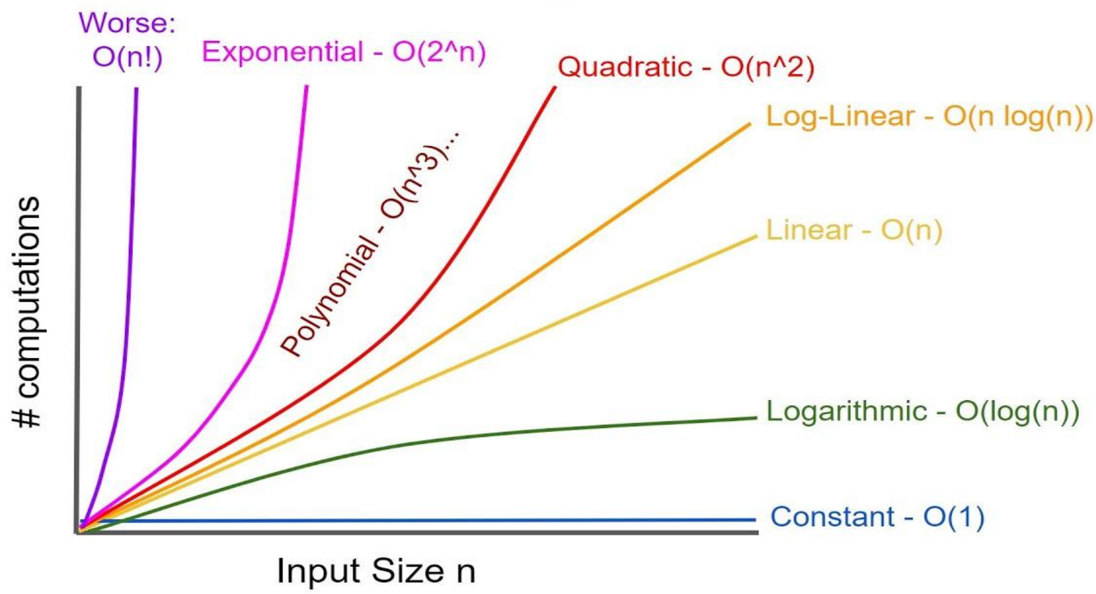
- Efficiency of algorithm evaluated using the count of CPU cycles.

Big O notation:

- Big O notation represents the worst time complexity trend of an algorithm.

Increasing order of Time complexity ↓(top => down)

O(1)	constant
O(logn)	logarithmic
O(n)	linear
O(nlogn)	Linear times logarithmic
$O(n^2)$	quadratic
$O(n^k)$	polynomial
$O(2^n)$	exponential
O(n!)	factorial



Sorting Algorithms

Bubble Sort

At each iteration the biggest element of the sub-array [0:n-i] , where n is the size of the array and i is the iteration number, will reach the last position.

Time complexity:
Worst: $O(n^2)$
Best: $O(n)$

```
def optimised_bubble_sort(a):
    counter = 0
    for i in range(len(a)-1):
        is_sorted = True
        for j in range(len(a)-i-1):
            counter += 1
            if a[j] > a[j+1]:
                a[j], a[j+1] = a[j+1], a[j]
                is_sorted = False
        if is_sorted == True:
            break
    print(counter)
    return a
```

Selection Sort

Loop the list and find the minimum and place it to the right position

Time complexity:
Worst: $O(n^2)$
Best: $O(n^2)$

```
def selection_sort(a):
    for i in range(len(a)-1):
        min_index = i
        for j in range(i+1, len(a)):
            if a[min_index] > a[j]:
                min_index = j
        if min_index != i:
            a[i], a[min_index] = a[min_index], a[i]
    return a
```

Insertion sort

- Start with the initial index = 1 and keep moving forward
- Left part is consider to be sorted and right is unsorted
- Pick the element from unsorted portion and put it into the right place at sorted portion

Worst case: $O(n^2)$
Best Case: $O(n)$

```
def insertion_sort(a):
    for i in range(1, len(a)):
        index_to_insert = i
        j = i - 1
        while j >= 0:
            if a[j] < a[index_to_insert]:
                break
            a[j], a[index_to_insert] = a[index_to_insert], a[j]
```

	`` index_to_insert = j j -= 1 return a ```

Recursion

- Recursion means calling a function from within the function body
- Recursive function must have, body and a base condition
- It uses a special data structure stack which works on LIFO
- Each time the function call is made, it'll store all the relevant sequence of calls in the stack and pops out when a particular call is over.

Merge Sort

- Follows the Divide and Conquer approach
- Keep dividing the list into list of halves until each list contain only 1 element
- Start merging the list in sorted order

Time complexity: O(NlogN)

```
def merge_sort(a):
    # Base condition
    if len(a) <= 1:
        return a
    n = len(a)
    # Splitting Logic
    left = merge_sort(a[:n//2])
    right = merge_sort(a[n//2:])
    # Merging Logic
    i, j = 0, 0
    result = []
    while i < len(left) and j <
len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    if i < len(left):
        result += left[i:]
    elif j < len(right):
        result += right[j:]
    return result
```

Combinatorics

- ${}^{n}P_{r}$: number of ways of r different things can be selected and arranged out of n different things

- ${}^{n}C_{r}$: number of ways of selecting r different things from n different things
- Total number of subsets of a set with n elements are 2^{n}.
- Number of ways of arranging n different things at n places: n!

$$^{n}P_{r} = \frac{n!}{(n-r)!}$$

$$^{n}C_{r} = \frac{n!}{(n-r)!r!}$$

$$^{n}C_{r} =^{n} C_{(n-r)}$$

$$^{n-1}C_{r-1} +^{n-1} C_{r} =^{n} C_{r}$$

-