#1 "if __name__=='__main__':"

- This line is typically used to check whether the python script is being run directly or being imported as a module into another script. when a python file is run directly, the special variable '__name__' is set to '__main__'

Example, how it is commonly used

```
def main():
    check
    Print("Hi")

if __name__=='__main__':
    main()
```

Example, using class

```
class myclass:
    def __init__(self, name):
        self.name = name
    def greet(self):
        Print("Hello", self.name)

def main():
    obj = myclass("shiva")
    obj.greet()

if __name__=='__main__':
    main()
```

#2

```
c = map(float, input().split())
```

Syntax : map(function_name, sequence)

- This line of code is used to read multiple input values from the user in a single line, split them into separate strings based on white-space, convert each string to a float and stores these float values in an iterable object ('map' object)

→ input() - Reads a line of input from the user as string

→ .split() - splits the input string into a list of substrings based on whitespace

→ float() - converts string datatype to float datatype

→ map(float,..) - applies the float() function to each element produced by 'split()'

Example :

```
c = map (float, input("enter :").split())
Print (type(c))
for i in c:
    Print (i)
```

output :

```
Enter : 1.1 1.3 2.5
<class 'map'>
1.1
1.3
2.5
```

@#5

#3 filter function using lambda function

- inbuild function 'filter' filters the elements of the sequence for which the function returns true. it creates a list of the elements for which the function Evaluates to true and ignores the other elements.

Syntax:

filter (function_name, sequence)

function_name → name of the function or entire lambda function
sequence → list of elements

Example with normal user-defined function, (to filter odd numbers)

```
def odd(x):
    if x%2==0:
        return False
    else:
        return True
Elements = [int(x) for x in input("Enter:").split()]
result = list(filter(odd, Elements))
Print(result)
```

output:
Enter: 1 2 6 7 3 22 11
[1, 7, 3, 11]

Example with lambda function (filters even)

```
Elements = [int(x) for x in input("Enter:").split()]
result = list(filter(lambda x:(x%2!=0), Elements))
Print(result)
```

output:
Enter: 1 2 6 7 3 22 11
[2, 6, 22]

#4

```
lst = list(int(x) for x in input().split())
```

input() → This function reads a line of input from the user.

- split() → This function splits the input into a list of substrings based on the whitespace by default (separates with comma)

int(x) for x in input().split() → This is a generator expression that iterates over the list of substrings produced by split(), converting each substring x to an integer

list(...) → This function takes the generator expression and constructs a list from its values

Example:

```
lst = list(int(x) for x in input("Enter:").split())
Print(lst)
```

output:

Enter: 1 3 6 7 8 11 12

[1,3,6,7,8,11,12]

#5 Map function ~~using~~ ~~lambda~~

- inbuild function 'map' creates a list of modified elements returned by the function. The map function passes the individual element from the sequence to the function and creates a sequence of the returned elements (modified elements)

syntax:

map(function_name, sequence)

function_name → name of the function or entire lambda function
sequence → list of elements

Example with user defined function (squaring the elements)

```
def power(x):
    return x**2
lst = [int(x) for x in input("Enter:").split()]
k=list(map(power, lst))
print('updated:', k)
```

output:
Enter: 1 2 3 4
updated: [1, 4, 9, 16]

Example with lambda

```
lst=[int(x) for x in input("Enter:").split()]
k= list(map(lambda x:(x**2), lst))
Print('updated:', k)
```

output:
Enter: 1 2 3 4 5
updated: [1,4,9,16,25]

#6 Reduce function

- inbuild function that reduces the list of elements to a single value. The reduction is done ~~by~~ according to the processing performed by the function. Reduce takes two values from the sequence passed to it in the first step and then passes two values to the function to reduce and then it passes the returned result and one value from the list to the function to reduce it to a single value again. This process is continued until the last element of the list not processed. The map function is used to pass two individual values from the sequence passed to it. reduce function is defined in the functools module we need to import functools to use reduce function

Syntax:

```
reduce (function_name, sequence)
```

Example using user-defined function,

```
from functools import *
def add(x,y):
    return x+y
lst=[int(x) for x in input("Enter:").split()]
k=reduce(add, lst)
Print('Result:', k)
```

output:

```
Enter: 1 2 3 4 5
Result: 15
```

Example using lambda,

```
from functools import *
lst=[int(x) for x in input("Enter:").split()]
K= reduce(lambda x,y:(x+y), lst)
Print ('Result:', K)
```

output:

```
Enter: 1 2 3 4 5
Result: 15
```

#7 Enumerate()

- It allows you to create a for loop where we can iterate over a sequence and have both the items and their index in the loop.

syntax:

enumerate (sequence, starts_point)

Sequence → list or tuple or set of elements

Starts_point → The starting index of the counter. default is 0

Example 1 (In list)

```
fruits = ['apple', 'Banana', 'Orange']
for index, fruit in enumerate(fruit):
    Print(index, fruit)
```

output:
0 apple
1 Banana
2 Orange

Example 2 (In list) can also do in list comprehension

```
fruits = ['A', 'B', 'C']
for index, fruit in enumerate(fruit, start=1):
    Print(index, fruit)
```

output
1 A
2 B
3 C

Example 3 (In Dict)

```
fruits = {'a':'apple', 'b':'banana'}
for index, (key, Value) in enumerate(fruits.items()):
    Print(index, key, Value)
```

output:
0 a apple
1 b banana

#8 Sorted( )

- Used to return a new sorted list from the elements of any iterable. The original iterable is not modified, ie) original list is not modified. The function can sort the elements in ascending or descending order. ascending is default
  ie) reverse = False

Syntax:

sorted (iterable, key = None, reverse = False / True)

iterable → list of elements to sort

key → a function that serves as a key for the sort comparison. default is none.

reverse → a boolean value, sorts in descending order if reverse = True.

Eg 1: (ascending)

```
num = [3, 4, 6, 7, 1]
sor = sorted (num)
Print (sor)
```

output:

[1, 3, 4, 6, 7]

Eg 2: (descending)

```
num = [3, 4, 6, 7, 1]
sor = sorted (num, reverse = True)
Print (sor)
```

output:

[7, 6, 4, 3, 1]

Eg 3: (key) len, ~~max, min~~ str.lower, abs, etc....

```
words = ['boy', 'girl', 'Horse']
sor = sorted (num, key = len)
Print (sor)
```

output:

['boy', 'girl', 'Horse']

freecampcode

Algorithms in Python

→ helps solve problems efficiently

→ algorithms are specific set of instructions that we write to our computer to tell what to do

what is algorithm

The current term of choice for a problem solving procedure... It is commonly used nowadays for the set of rules a machine (...especially a computer) follows to achieve a particular goal

It does not always apply to computer-mediated activity, however. The term may as accurately be used.... [for] the steps followed in making a pizza or solving a Rubik's cube as for computer-powered data analysis

Algorithm[s]... [are] often paired with words specifying the activity for which a set of rules have been designed

A search algorithm, for example, is a procedure that determines what kind of information is retrieved from a large mass of data

~~An encryption~~ An encryption algorithm is a set of rules by which information or messages are encoded so that unauthorized person cannot read them

1. Simple Recursive Algorithm
2. Algorithms within Data Structure
3. Divide & conquer
4. Greedy Algorithm
5. Dynamic programming

1. Simple Recursive Algorithm

Recursion → function calls itself until the program terminates

LIFO - Last In First Out
(Eg: last number which goes in will be the first number to called out)

Easy:

factorial

```
def fact(n):
    if n==1:
        return n # when n is 1, return 1
    else:
        temp = fact(n-1) # Recursive call
        temp = temp * n # Multiply the result of fact(n-1) by n
    return temp # return computed value
print(fact(5))
```

output:

120

here,
- the function has a base case that stops the recursion when n is 1, the function returns 1
- If n is not 1 the function calls itself with the argument n-1. this continues until n is reduced to 1

→

break down the execution step by step

1. fact(5) calls fact(4)
2. fact(4) calls fact(3)
3. fact(3) calls fact(2)
4. fact(2) calls fact(1)
5. fact(1) returns 1

when fact(1) returns 1, the stack unwinds in reverse order [follows LIFO principle],

1. fact(2) returns $1*2 = 2$
2. fact(3) returns $2*3 = 6$
3. fact(4) returns $4*6 = 24$
4. fact(5) returns $24*5 = 120$

In **LIFO**,

- The last function call made is the first one to be resolved
- when a recursive function calls itself, each call is placed on top of the call stack
- once the base case is reached, the calls start returning values, begining with most recent calls

Advanced :
Permutation

→

what is permutation?

it refers to the different ways in which a set of objects can be arranged. If you have n distinct objects, the total number of permutations of these objects is given by the factorial of n (n!)

Example with "ABCDE"

"ABCDE" has 5 distinct characters

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

so, there are 120 different ways to arrange the characters in "ABCDE"

Position-specific permutations

if you want to determine how many times a specific character appears in a specific position across all permutations.

For 'A' at the 0th index,

Fix 'A' at the 0th index and permute the remaining four characters (B, C, D, E)

$4! = 4 \times 3 \times 2 \times 1 = 24$

Examples →

Algorithms within Data Structures

- Data structures are containers or collection of data they provide a means of managing small or large amounts of values for uses like Databases, Web traffic monitoring user registrations, indexing (Eg: arrays)

1. Linear search
2. Binary search
3. Bubble sort
4. Insertion sort

1. Linear search

How it works,

1. start at beginning : Begin with the first element of the list
2. check each element : compare each element in the list with the target value
3. stop if found : if the target value is found, return the index of the element
4. continue if not found : if the target value is not found, move to the next element and repeat the process
5. End of list : if the end of the list is reached without finding the target, return a value indicating the target is not present

Example,

```
def linear_search(arr, target)
    for index in range(len(arr)):
        if arr[index] == target:
            return index
    return -1
linear_search([1,2,3,4,5], 2)
```

2. Binary Search

- Its an efficient algorithm for finding the position of a target value within a sorted list

How it works,

1. Start with the entire list: consider the whole list as the search interval
2. Find the middle element: calculate middle index of the current interval
3. compare the middle element with the target:
   - if the middle element is equal to the target, the is complete, and index is returned
   - if the middle element is less than the target, search the right half of the interval
   - if the middle element is greater than the target, search the left half of the interval
4. Repeat: continue the process on the remaining half until the target is found or the interval is empty

2 Examples →

Dealing with Complex Numbers

Hacker Rank

@ 1

Dealing with Complex Numbers

```
import math
class complex(object)
    def __init__(self, real, imaginary):
    #constructor to initialize a complex number with real
      and imaginary parts
        self.real = real
        self.imaginary = imaginary
    def __add__(self,no):
    #overloading the + operator to add two complexnumbers
        return self.__class__(self.real + no.real, self.imaginary
                                                + no.imaginary)

    def __sub__(self,no):
    #overloading the - operator to subtract two complex
      numbers
        return self.__class__(self.real - no.real, self.imaginary
                                                + no.imaginary)
    def __mul__(self,no):
    #overloading the * operator to multiply two complex
                                                    numbers
        r = (self.real * no.real) - (self.imaginary * no.imaginary)
        i = (self.real * no.imaginary) + (self.imaginary * no.real)
        # Formating the result to the desired string
          format with two decimal places
        return "{:.2f}{}{:.2f}{}".format(r,"+"if i>=0 else"-",
                                        abs(i),"i")
    def __truediv__(self,no):
    #overloading the / operator to divide two complex numbers
        .
        .
        .
```

```
        :
        :
        div = no.real **2 + no.imaginary **2
        real_part = ((self.real * no.real) + (self.imaginary *
                                    no.imaginary)) / div
        imag_part = ((self.imaginary * no.real) - (self.real *
                                    no.imaginary)) / div

        return self.__class__(real_part, imag_part)

    def mod(self):
    # Method to calculate the ~~norm~~ modulus of a complex number
        no = pow(pow(self.real, 2) + pow(self.imaginary, 2), 0.5)
        return self.__class__(no, 0)
    def __str__(self):
    # overloading the str() function to get the string
      representation of a complex numbers
        if self.imaginary == 0 :
            result = "%.2f + 0.00i" % (self.imaginary)
        elif self.real == 0:
            if self.imaginary >= 0:
                result = "0.00 + %.2fi" % (self.imaginary)
            else:
                result = "0.00 - %.2fi" % (abs(self.imaginary))
        elif self.imaginary > 0:
            result = "%.2f + %.2fi" % (self.real, self.imaginary)
        else:
            result = "%.2f - %.2fi" % (self.real, abs(self.imaginary
                                                      ))

        return result
        :
        :
```

```
:

if __name__ == '__main__':
    # Taking two sets of inputs for complex numbers
    c = map(float, input().split())
    d = map(float, input().split())
    # Creating complex number instances with the input values
    x = complex(*c)
    y = complex(*d)
    # Performing operations and printing results
    # Each operation (+, -, *, /, modulus) is performed
    # and the result is converted to string format using the __str__ method
    print(*map(str, [x+y, x-y, x*y, x/y, x.mod(), y.mod()]), sep='\n')
```

Output:

2 1
5 6 } input

7.00+7.00i
-3.00-5.00i
4.00+17.00i
0.26-0.11i
2.24+0.00i
7.81+0.00i

Question ?

given two complex numbers, and you have to print the result of their addition, subtraction, multiplication, division and modulus operations.
The real and imaginary precision part should be correct up to two decimal places.

Input format :

one line of input : the real and imaginary part of a number, separated by a space

output format:

for two complex numbers C and D

- C + D
- C - D
- C * D
- C / D
- mod (C)
- mod (D)

} output format

for complex numbers with non-zero real (A) and complex part (B). output should be A+Bi,
Replace the plus sign (+) with a minus symbol (-) when B < 0

For complex numbers with a zero complex part, A+0.00i

For complex numbers where the real part is zero and complex part is non-zero, 0.00+Bi

Explanation

→ class Definition : we define a class 'complex' to represent complex numbers

→ constructor ('__init__') : This special method initializes each instance of the class with 'real' and 'imaginary' parts

→ operator overloading :

operator overloading allows us to define custom behavior for operators ('+', '-', '*', '/') when they are used with instances of the class. This makes instances of the class behave more like build-in types

- Addition ('__add__')

How it works: returns a new instance of 'complex' where real & img parts are (sums)

Purpose : To define how to add two complex numbers

Syntax : def __add__(self, other):

- subtraction ('__sub__')

Purpose : To define how to subtract two complex nums

syntax : def __sub__(self, other):

How it works: returns a new instance of 'complex' with real and img parts (subtracted)

- multiplication ('__mul__')

How works: calculates real(r) and img(i), then formats them to a string with two decimal points

Purpose : to multiply two complex numbers

Formula: $(a+bi)*(c+di) = (ac-bd)+(ad+bc)i$

syntax : def __mul__(self, other):

- Division ('__truediv__')

Purpose : to divide ~~two~~ one complex number by another

Formula: $(a+bi)/(c+di) = [(ac+bd)+(bc-ad)i]/(c^2+d^2)$

syntax : def __truediv__(self, other):

How it works: calculates the denominator (div), and results by real & img

→ Modulus Method

Purpose: to calculate modulus (magnitude) of a complex number

formula: $|a+bi| = sqrt(a^2 + b^2)$

for sqrt we imported math

syntax: def ~~module~~ Method_name (self):

How it works: it calculates the modulus and returns a new 'complex' object with the modulus as real and 0 as imaginary part

→ String representation (__str__)

Purpose is to provide string representation of the complex number

syntax: def __str__ (self)

How it works: it formats the real and img parts to two decimal places and returns a string

→ Performing operations.

performs ('+', '-', '*', '/', 'modulus'), converts the results to strings and prints them ~~on~~ each on a new line

```
Print(*map(str, [x+y, x-y, x*y, x/y, x.mod(), y.mod()])
      , sep='\n')
```

→ self.__class__(args)

How it works: it refers to the class of the current instance, allowing the creation of a new instance of the same class

Why it is used: Ensures that the returned object is of the same class, even if the method is used in subclass

Syntax:

→ The operator overloading is invoked when the specific operation is performed on the objects of the class, not when the objects are created

when operator overloading happens ?

operator overloading methods (such as '__add__', ...) are called when the corresponding operation ('+', ...) is performed on the instance of the class. it happens dynamically at runtime, exactly at the moment the operation is encountered in the code.

Detailed explanation

1. object creation

```
{ x = complex(*c)
  y = complex(*d)
```

- at this point, the constructor '__init__' is called for each instance, initializing their 'real' and 'imaginary' parts, no operator overloading is happened yet

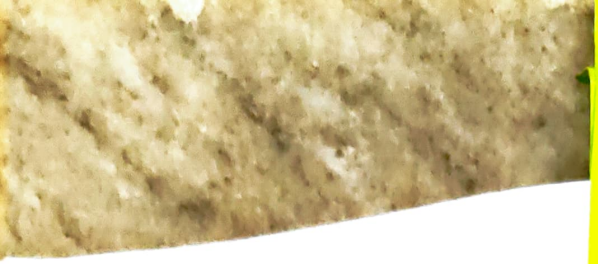
2. operations

```
{ result_add = x + y   [calls or triggers] x.__add__(y)
  result_sub = x - y   [calls or ,,]       x.__sub__(y)
  result_mul = x * y   [calls or ,,]       x.__mul__(y)
  result_div = x / y   [calls or ,,]       x.__truediv__(y)
  result_mod_x = x.mod()  [calls or ,,]    x.mod()
  result_mod_y = y.mod()  [calls or ,,]    y.mod()
```

3. string representation

```
{ print(*map(str, [result_add, result_sub ....]), sep='\n')
```

The '__str__' method is called when the 'str()' function is used, which happens inside the map(str, ...) call



Find a String

HackerRank

@2

Find a String

```
def count_substring(string, sub_string):
    count=0  #initialize a counter to track the number of occurrence
    lst1 = list(string) # convert the input string to a list of characters
    st=[] # Initialize an empty list to store substrings of the same length
          as 'sub_string'

    #iterate over the string to create substrings of the same length as
     'sub_string'
    for i in range(len(lst1)-len(sub_string+1):
        st.extend([''.join(lst1[i:i+len(sub_string)])])
        # create and add substrings to the list 'st'
#iterate over the list of substrings to count occurrences of
 'sub_string'
    for j in st:
        if j == sub_string: #if the substring matches 'sub_string'
            count+=1 #Increment the counter
        else:
            continue #continue to the next iteration if there's
                     no match
    return count # returns the total count of occurrences
if __name__ == '__main__':
    string = input().strip() # read the input string from user.
    sub_string = input().string() # read the substring to
                                    search for from the user
    count=count_sbstring(string, sub_string) # call the
                                    function with the inputs and store the
                                    result
    Print(count) # print the result
```

→ function 'count_substring' takes two arguments 'string' and 'sub_string'

→ 'count' to keep track of the number of sub_string in string

→ The loop 'for i in range(len(lst1)-len(sub_string)+1):' iterates over the indices of 'lst1' such that the slicing operation does not go out of bounds

→ 'st.extent(["".join(lst1[i:i+len(sub_string)]])]' creates a substring starting at index 'i' with a length equal to 'sub_string' and adds it to the list 'st'

→ The loop 'for j in st' iterates over the list of generated substrings

→ if the substring 'j' matches 'sub_string', the 'count' is incremented

Time & space complexity

Time & Space

Big O notation

Omega

Theta

Big O notation

- O(1) — Constant
- O(n) — Linear
- O(logN) — Lograthmic
- O(log(NlogN) — Linear Lograthmic
- O(n^2) — Quadratic

- O(1)

Constant time complexity, means that the execution time of an algorithm or function does not depend on the size of the input. regardless of how big or small the input is, the time it takes to execute the function remains same.

Direct access: Typically involves operations that access a specific element or perform a simple computation

Predictable Performance: Since the execution time is constant, you can predict how long the operation will take, regardless of input size

Scalability: Operations with O(1) complexity scale very well as the input size increases because their performance does not degrade.

- O(n)

Linear time complexity, means that the execution time of an algorithm or function increases linearly with the size of the input. if the input size doubles, the execution time also doubles

Proportional Growth: The runtime grows proportionally with the size of the input

Iteration over input: Typically involves iterating over elements in a data structure

Predictable Growth: you can predict how the runtime will increase as the input size increases

Best case

Eg:

```
L=[1, 2, 3]
a=1
for i in L:
    if i==a:
        Print ("True")
```

Average case

```
L=[2, 1, 3]
a=1
for i in L:
    if i==a:
        Print ("True")
```

worst case

```
L=[3, 3, 1]
a=1
for i in L:
    if i==a:
        Print ("True")
```

- $O(\log n)$

Logarithmic time complexity, means that the execution time of an algorithm increases logarithmically as the input size increases. In simpler terms, if you double the input size, the number of operations required only increases by a constant amount

Slow growth: The runtime grows much more slowly compared to the input size

Divide & conquer: Typically involves algorithms that divide the problem in half at each step

- highly scalable and handle large inputs efficiently

- $O(n \log n)$

Linearithmic time complexity, means that the execution time of an algorithm grows in proportion to the input size n multiplied by the logarithm of the input size. This complexity often arises in algorithms that perform a combination of linear and logarithmic operations, such as divide-and-conquer strategies to each element.

Combination of Linear and Logarithmic: The runtime increases more slowly than quadratic time but faster than linear time

common in Efficient sorting Algorithms: Many Efficient sorting algorithms, like Merge Sort and Quick sort, have $O(n \log n)$ complexity

- $O(n^2)$

quadratic time complexity, means that the execution time of an algorithm grows proportionally to the square of the input size. if the input size doubles, the execution time quadruples This complexity often arises in algorithms that involve nested iterations over the input data

Rapid growth: The runtime grows much faster than linear time as the input size increases

Nested Loops: Typically involves algorithms with nested loops, where each element is compared with every other element.

Space Complexity

- $O(1)$ → constant Space

Defination - The amount of memory used is constant and does not depend on the input size

key points - Fixed memory usage, no additional space required based on input

- $O(n)$ → linear Space

Defination - The amount of memory used grows linearly with the input size

key points - memory usage is directly proportional to input size

- $O(\log n)$ → Logarithmic space

Definition: The amount of memory used grows logarithmically as the input size increases

Key points: Memory usage grows slowly, often related to recursive algorithms that divide the problem

- $O(n \log n)$ → Linearithmic space

Definition: the amount of memory used grows in proportion to the input size multiplied by the logarithm of the input size

Key points: combination of linear and logarithmic growth, often seen in divide and conquer algorithms

- $O(n^2)$ → Quadratic space complexity

Definition: The amount of memory used grows proportionally to the square of the input size.

Key points: Rapid growth in memory usage, often due to storing information in a two-dimensional structure

Example input is 5

$O(1)$ → Best

$O(\log n)$ ~~0 1 2 3 4~~

$O(n)$ 5

$O(n \log n)$ 15

$O(n^2)$ 25

$O(n^3)$ 125

$O(2^n)$ worst

Array, Searching

Array

- collection of data
- Same Data type
- Static in other language, Dynamic in Python

choosing data structure is based on these four operation

1. Access
2. insert
3. Delete
4. search

- Elements of array is accessed by its index value

Time complexity
O(1)

Syntax:

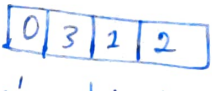
```
array_name [index]
```

- inserting element in array

Index

need to insert '3'

then

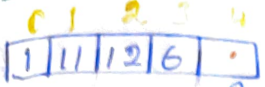


'1' and '2' moves to next index

inserting '3' is first operation
switching of '1' is second operation
switching of '2' is third operation

length
so here
O(n-i)
O(4-1)
O(3)

scene 2



inserting '5' here

so here

$O(n-i)$

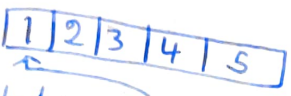
$\rightarrow O(5-4)$

$\rightarrow O(1)$

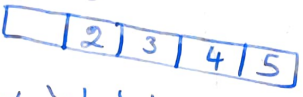
done in constant time

- Deleting an element in array

scene 1



Deleting '1'



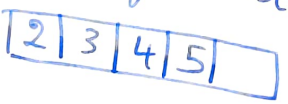
So, here

$O(n-i)$

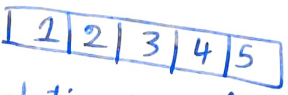
$\rightarrow O(5-0)$

$\rightarrow O(5)$

now '2', '3', '4', '5' will shift to the preceding index



scene 2



Deleting '5'

$O(n-i)$

$O(5-4)$

$O(1)$

now

- Searching an element in an array

Searching '5'

best case

* if '5' in index 0 or (first index) then, $O(1)$

* if '5' in index 1 or 2 or 3 then it is not O(1) or O(n) ?

↓ Average case

* if '5' in last index (4), then it is O(n)

↳ worst case

{Now finally we should declare that O(n) is the time complexity of Searching an element in array}

choosing array for a program

- program where there is a need to access element as it's time complexity is O(1) (if accessing is main need)
- Program where inserting and deleting an element in last index as its time complexity is O(1). (if this is main need) should not choose array in bestcase or average case in inserting or deleting as the time complexity is O(n-i)
- mostly in all data structure the searching time complexity is O(n), so this is optional. in array

Syntax of Array

import array as arr

variable=arr.array('type_code', [ ])

↓

iterable

type_code	C type	Python type	Minimum size in byte
'b'	signed char	int	1
'B'	unsigned char	int	1
'u'	Py_unicode	unicode character	2
'h'	signed short	int	2
'H'	unsigned short	"	2
'i'	signed int	"	2
'I'	un " "	"	2
'l'	signed long	"	4
'L'	un " "	"	4
'q'	signed long long	"	8
'Q'	un " " "	"	8
'f'	float	float	4
'F'	double	float	8

Searching Algorithm

- Linear Search

equates each element from start to end with the search key, if it is equal return index or success message otherwise (search out of the array) doesn't return anything

Time complexity is $O(n)$,

The search key may be in best case or average case but we have to consider time complexity as worst case

[to store the returning index value the space complexity is $O(1)$
maybe we are not storing the index value, but minimum space complexity of a program is $O(2)$]

Example 1

```
array = [1, 2, 3, 4, 5]
a = 3
for i in range (len(array)):
    if array[i] == a:
        Print(i)
else:       break
    Print("not found")
```

Example 2

```
array =[1,2,3,4,5]
a= 3
for i in array:
    if i==a:
        Print("yes")
        break
else:
    Print("not found")
```

- Binary search

-Time complexity is O(log n)

Definition - Binary search is a fast algorithm for finding a target value within a sorted array

Steps:

1. initialize pointers: set two pointers, low (starting at the beginning of the array) and high (starting at the end).
2. Calculate Midpoint: compute the midpoint mid = (low + high)//2
3. comparison:
   if array[mid] == target, the target is found
   if array[mid] < target, update low to 'mid + 1'
   if array[mid] > target, update high to 'mid - 1'

Repeat until low exceeds high or target is found

Example:

```
nums = [1,2,3,4,5,6,7]
key = 5
l = 0
r = len(nums) - 1
while l <= r:
    mid = (l + r) // 2
    if nums[mid] == key:
        Print(mid)
        break
    elif nums[mid] < key:
        l = mid + 1
    elif nums[mid] > key:
        r = mid - 1
else:
    Print("not found")
```

Reason for O(log n) : — GOT

1. initial intervals:
   - you start with an array of size (n)
2. Halving the Interval:
   - Each step of the binary search cuts the interval size in half

- In the first step, the interval is reduced from (n) to $(n/2)$
- In the second step, it's reduced from $(n/2)$ to $(n/4)$
- This halving continues until the interval size becomes 1

3. Number of Halves:

- The number of times you can divide (n) by 2 until you get 1 is the logarithm base 2 of (n) denoted as $(\log_2 n)$
- Mathematically, if you divide (n) by 2, (k) times to get 1, then: $[n \text{times } \left(\frac{1}{2}\right)^k = 1 \implies \left(\frac{1}{2}\right)^k = \frac{1}{n} \implies k = \log_2 n]$

4. Iterations

- Thus, the number of iterations (or steps) required to reduce the interval to 1 (where you either find the target or conclude it isn't present) is $(\log_2 n)$.

Recursion

Recursion

- Function which calls itself
- a logic written number of times can be replaced by recursion with the same logic
- base condition is must or else infinite iteration will happen

For factorial (5)

$5 \times 4 \times 3 \times 2 \times 1$

$5 \times 4!$

$5 \times 4 \times 3!$

$5 \times 4 \times 3 \times 2!$

$5 \times 4 \times 3 \times 2 \times 1!$

$5 \times 4 \times 3 \times 2 \times 1 \times 1!$

$1 \times 1!$

base condition

as we know the value of 1! we break this and return the value (1)

To stop this we use base condition

to find the logic, we have to find that, is there any relation between the sub problems

```
def fact(n):
    if n == 1:
        return 1
    return n * fact(n-1).
print(fact(5))
```

1. n will be 5 and the the n changes to "fact(n-2)" (5-1) and again called with n = 4.

Stack
5

then the 5 will be stored in a stack, (5x4!)

2. now n will be 4 and then the n changes to (4-1) and again called with n = 3, the the 4 will be stored in stack, (4x3!)

Stack
4
5

3. now n will be 3 and then the n changes to (3-1) and again called with n = 2, then 3 will be stored in stack, (3x2!)

Stack
3
4
5

4. now n will be 2 and then the n changes to (2-1) and again called with n = 1 then 2 will be stored in stack, (2x1!)

Stack
2
3
4
5

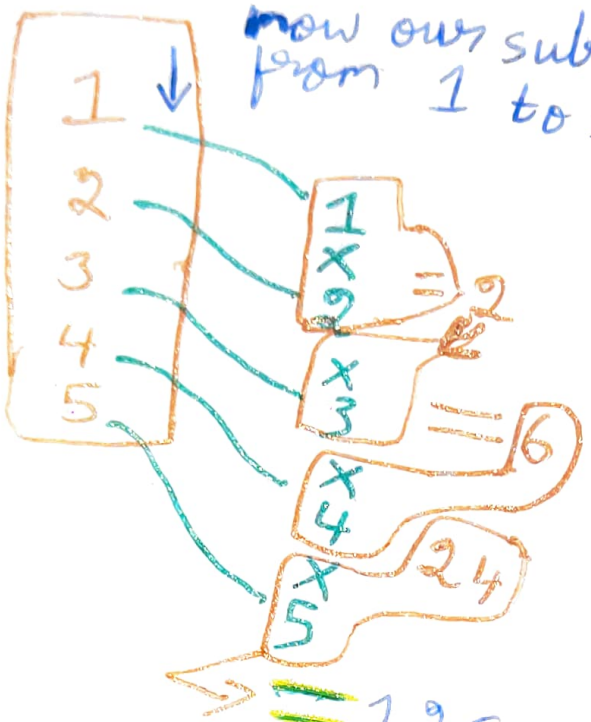
5. Now, it goes to base condition "if n == 1:" and returns the value of 1!

Stack
1
2
3
4
5

At step 5, instead of recursive function base function works as 'n == 1'

now, how stack works,

stored values



now our sub problem starts to multiply from 1 to 5