Revision Class 7 (Dated: 02-Sep-2022)

Aggregation

Aggregation is a form of reusability, where some class becomes part of another class. In aggregation "has a" relation exists between two classes. For example, tournament has matches, match has teams, team has players. Another example is room has fans, fan has motors (sometimes motors, one to revolve the fins of the fan and other to rotate upper part of fan like pedestal fan or rotating wall fan)

Aggregation is widely use in software development. Consider following address class:

```
class Address:
    def __init__(self, hn, bn, tn, cty, ctry):
        self.__house_no = hn
        self.__block_no = bn
        self.__town = tn
        self.__city = cty
        self.__country = ctry

    def __str__(self):
        return f'House #: {self.__house_no}\nBlock #:…
    def get_str(self):
        return f'{self.__house_no}\t{self.__block_no}\t…
    def __eq__(self, adr):
        return self.__house_no == adr.__house_no and …
```

This address class can be part of many class. Like Employee, Player, Student, Teacher, Worker, Company, Publisher, Distributor etc.

Strong Aggregation

In case of strong aggregation, object of class A becomes permanent part of class B. When object of class B will be created, object(s) of class A will also create and when object of class B will destroy (finish/ deleted), object of class A will also destroy. Consider following Employee class:

```
import address as ad
class Employee:
    emp_count = 0
    def __init__(self, n, d, s , c_h, c_b, c_t, c_cty, c_ctry, p_h, p_b, p_t, p_cty, p_ctry):
        Employee.emp_count += 1
        self.__emp_no = Employee.emp_count
        self.__name = n
        self.__designation =    d
        self.__salary = s
        self.__c_add = ad.Address(c_h, c_b, c_t, c_cty, c_ctry)
        self.__p_add = ad.Address(p_h, p_b, p_t, p_cty, p_ctry)
```

```
    def __str__(self):
        return f'Employee No: {self.__emp_no}\nName: {self.__name}\n…

    def get_str(self):
        return f'{self.__emp_no} {self.__name}\t{self.__designation}\t…
```

Weak Aggregation

In weak aggregation, classes have relatively weaker relation. The existence need not to be permanent. For example, class room has students. It is possible that there will student in class room at the time of class; however, after classes, there might not be any student. So as, the example of road has car. Person has car.

However, there is some manipulation required in this case, see example of Employee classes with weaker relation.

```
import address as ad
class Employee:
    emp_count = 0
    def __init__(self, n, d, s):
        Employee.emp_count += 1
        self.__emp_no = Employee.emp_count
        self.__name = n
        self.__designation =    d
        self.__salary = s
        self.__c_add = ''
        self.__p_add = ''

    def set_c_add(self, c_h, c_b, c_t, c_cty, c_ctry):
        self.__c_add = ad.Address(c_h, c_b, c_t, c_cty, c_ctry)

    def set_p_add(self, p_h, p_b, p_t, p_cty, p_ctry):
        self.__p_add = ad.Address(p_h, p_b, p_t, p_cty, p_ctry)

    def __str__(self):
        address=''
        if self.__c_add == '':
            address = 'No Current Address'
        else:
            address = str(self.__c_add)
        if self.__p_add == '':
            address += f'\tNo Permanent Address'
        else:
            address += f'\t{str(self.__c_add)}'
```

```
        return f'Employee No: {self.__emp_no}\nName: {self.__name}\nDesignation: {self.__designation}\nSalary:{self.__salary}\nCurrent Address:\n{address}'
```

Recursion

Here, we will discuss some problems and their solution will be provided in python; however, it is instructed to try them yourself first. For the following descriptions, you have to implement recursive functions without loop:

1. Find sum of all elements of list
2. Find product of all elements of list
3. Find x power n
4. Do binary search
5. Find maximum pair sum (where pair has two adjacent elements)

Discussion

First of all, for every recursive function in general, we need more parameters than iterative function. Typically, we take extra parameters with default value to call them outside the function without those extra parameters.

1. In first problem, apart from list, we will take another parameter that is index, where index will start from zero. In each function, we will have base case that if index is equal to length of the list then return zero (additive identity). Otherwise, take sum from next recursive function call (by adding one to index) and add current element (at index) and return the sum
2. In second problem, apart from list, we will take another parameter that is index, where index will start from zero. In each function, we will have base case that if index is equal to length of the list then return zero (multiplicative identity. Otherwise, take sum from next recursive function call (by adding one to index) and add current element (at index) and return their product
3. In third problem, we don't need extra parameter; however, we know very well that if power is zero, the result will be one. So, we have a base case that if power is zero, return one. Next, we will get power from next recursive call (by reducing power by one) and the function will return product of current element with the power returned by calls
4. In binary search, we have two extra parameters, **start** and ***end***. The parameter ***start*** has default value zero and ***end*** has default value length of the list. Inside, we will calculate mid and write two recursive calls with if else, in one call, we will set parameter ***start*** to

mid+1 and in second call, we will set ***end*** to ***mid-1***. The base case are very obvious, if element found at middle, return the middle (index). If parameter ***start*** become greater or equal parameter ***end*** (means the element doesn't exist) return sentinel value or raise exception

5. Again, one extra parameter index required, with default value zero. The base case will be if index become equal to length of list minus two (because, we are working on current and next to current making pair). In base case return 0 (the minimum sum in case of positive numbers.

   Next, get maximum sum from next recursive call, again by adding one into the index and store sum into variable say s2. Get sum of current pair that is at index and index plus one, say it s1. Now compare s1, s2 and return greater value