Paper 4 (Python)

These are basic templates for programming concepts that come up often in the Paper 4. They differ from paper to paper, but the general code is always similar:

Linear and Binary Search

- **Linear search**

```
def LinearSearch(ItemToFind):
    for i in range(len(Array)):
        if Array[i] == ItemToFind:
            return Array[i]
```

  - ➔ **One by one, compares element at each index position with desired item**
  - ➔ **Can also utilise a boolean Found variable (used especially when something has to be executed if the item is not found)**

- **Binary search**

```
def BinarySearch(DataToFind):
 First = 0
 Last= len(Array-1)

 while(First <= Last):
     MidValue = int((First + Last) / 2)
     if DataToFind == DataStored[MidValue]:
     return MidValue
     if DataToFind < DataStored[MidValue]:
     Last = MidValue - 1
     else:
     First = MidValue + 1
 return -1
```

  - ➔ **"First": the first index/element of section being searched - initialised to 0**
  - ➔ **"Last": the last index/element of section being searched - initialised to last index of array (the position of the last element)**
  - ➔ **The array is halved until element/data at MidValue index position matches DataToFind**
  - ➔ **If the value at the MidValue position is smaller than DataToFind, the upper half of the section is discarded (hence, MidValue becomes the last element in the new section)**
  - ➔ **If the value at the position is larger, the lower half of the section is discarded (hence, MidValue becomes the first element in the new section)**

➔ This is done repeatedly until the entire array has been searched or the value is found and returned (-1 is returned if the value has not been found)

Insertion and Bubble Sort

- Insertion sort

```
def InsertionSort():
    global Array

    for i in range(1,len(Array)):
        Current = Array[i]
        while Current < Array[i-1]:
            Array[i] = Array[i-1]
            i = i - 1
        Array[i] = Current

    return Array
```

➔ Opposite to a bubble sort, the insertion sort compares the current value with the ones that come before it

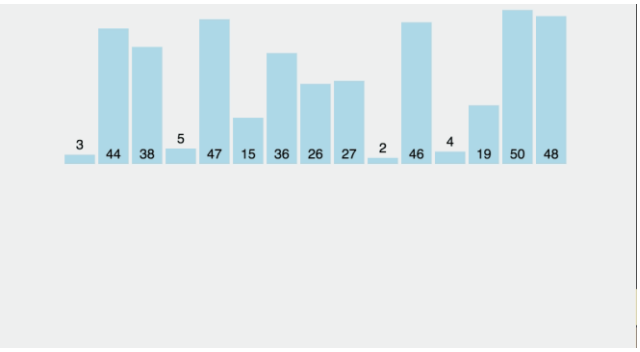
➔ Outside 'for' loop: ensures the code runs through each element in the array and moves it into the right slot (swaps it with the one that was there before)

➔ 'Current': stores the current variable being compared against all others before it

➔ Inside 'while' loop: compares Current value with all other elements that come before it, swapping it with the ones that are larger

➔ When 'while' loop stops running, the value is slotted into the latest index position

Representation of Algorithm

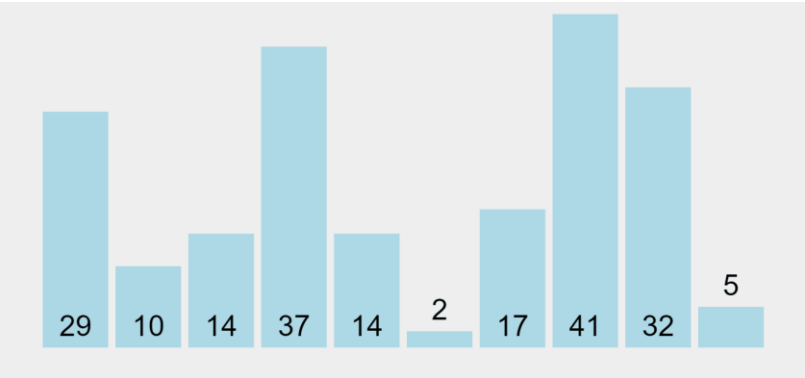


- Bubble sort

```
def BubbleSort(Array):
    for i in range(len(Array)-1):
        for j in range(len(Array)-1-i):
            if Array[i] > Array[i+1]:
                Temp = Array[j]
                Array[j] = Array[j+1]
                Array[j+1] = Temp
    return Array
```

- ➔ **1st 'for' loop: contents run as many times as there are elements in the array**
- ➔ **2nd 'for' loop: contents run through the array, each time leaving out one more element from the end (what the '*-i*' is for)**
- ➔ **Comparison: depending on whether descending or ascending order is wanted, the statement will be slightly different**
- ➔ **Swap: elements in arrays are swapped, usually through use of a variable for temporary storing of an element's contents**

Representation of Algorithm



Recursive Algorithms *(TBA)*

Files

- **Reading from a file**

```
def ReadData():
    FileData = []

    try:
        file = open("textname.txt","r")

        FileData = file.read().split("\n")

        return FileData
        file.close()
    except:
        print("File not found")
```

- ➔ **Implement exception handling**
- ➔ **The "textname.txt" depends on specific file name given in question**
- ➔ **The "\n" is used to indicate a new line, and is removed from the read line when inserting it into the array**
  (Would otherwise save it as "linedata\n" instead of just "linedata")
- ➔ **Always close the file after use**

```
Array = []

try:
 file = open("textname.txt", 'r')
 for line in file:
    Array.append(line.strip("\n"))
    file.close()
except:
 print("File not found")
```

- ➔ **When transferring file contents to an array, a 'for' loop is used (runs through each line individually)**
- ➔ **Each line is read separately and appended to the array**

Classes/Objects (OOP)

- **Declaring a class**

```
class ClassName:
    def __init__(self,ParameterValue1,ParameterValue2):
        self.__Value1 = ParameterValue1
        self.__Value2 = ParameterValue2
        self.__Value3 = 1
```

➔ 'def__init__' is the constructor

➔ 'self.__' sets the attributes (values) to private ('self.' would be public)

- 'Get' methods - return a value from the class

```
def GetValue1(self):
    return self.__Value1
```

➔ Always include 'self' in function parameters

➔ Function is indented into the class, same way as the constructor

- 'Set' methods - change a value to given parameter

```
def SetValue1(self,newValue):
    self.__Value1 = newValue
```

- Creating objects from declared classes

```
ObjectName = ClassName("Computer Science",9618)
```

➔ "ObjectName" would be the name of the object being created

➔ Values in brackets will depend on the declared class - order of given parameters has to match with constructor E.g.

def_init_(self, Subject, Code, Grade) →ClassName("Mathematics",9709,"A")

- Working with objects

```
ObjectName.GetValue1()
ObjectName.SetValue1(20)
```

➔ Returns the value from object

➔ Sets the variable Value1 to 20

```
Array.append(ClassName("Computer Science",9618)
Array[i].GetValue1()
Array[i].SetValue1(20)
```

- ➔ Appends an element of type ClassName to array
- ➔ Returns Value1 from object held in index `i`
- ➔ Sets Value1 of object held in index `i` to 20

Exception Handling

```
try:
    print(Num)
except:
    print("Something went wrong, number couldn't be printed")
```

- ➔ All code that should be executed is indented into “try:” command
- ➔ Code under “except:” command will execute if an error (exception) occurs in code above

Encoding ADTs (Abstract Data Types)

1. Stacks

```
Stack = [] #1D array of 20 elements
TopPointer = 0
```

- ➔ Stacks use an array and a TopPointer, which usually points to the next free space in the array/stack

- **Adding an item - Push**

```
def PushData(Value):
    global Stack
    global TopPointer
    if TopPointer == 20:
        print("Stack is full")
    else:
        Stack.append(Value)
        TopPointer = TopPointer + 1
```

- ➔ If statement checks that the Pointer isn’t pointing to a value out of the stack range (index 20 implies 21st element, so if the stack only stores 20, the stack would already be full)
- ➔ When a value is appended to the stack, the TopPointer is always incremented

- **Removing an item - Pop**

```
def PopVowel():
    global Stack
    global TopPointer
    if TopPointer - 1 >= 0:
        TopPointer = TopPointer - 1
        DataToReturn = Stack[TopPointer]
        Stack.pop()
        return DataToReturn
    else:
        return "No Data"
```

- ➔ If statement checks the stack isn't already empty
- ➔ Value of TopPointer is decremented and the value currently stored there is returned
- ➔ This element is removed from stack using the .pop() function

2. Queues

- Initialising a queue structure

```
Queue = [] #global 1D array of 20 elements
HeadPointer = -1 #global integer
TailPointer = 0 #global integer
```

- ➔ Heapointer points to the first element in the queue (initially -1 as there are no elements to point to yet)
- ➔ TailPointer points to next available slot in the queue (this is initially the first element in the empty array, so index 0)
- ➔ For a circular queue, both the Head and Tail pointers are initialised to 0 (A variable storing the number of items in the queue is also used)

- Adding items to a queue

```
def Enqueue(Value):
    global Queue
    global HeadPointer
    global TailPointer

    if TailPointer == 20:
        print("Queue full")
    else:
        Queue.append(Value)
        TailPointer = TailPointer + 1
    if HeadPointer == -1:
        HeadPointer = 0
```

- ➔ If TailPointer points to index 20 (21st element in the array), the queue is already full
- ➔ Otherwise, value is appended to the queue and TailPointer is incremented
- ➔ If the queue was empty before, HeadPointer is now set to 0
- ➔ For a circular queue, if the TailPointer value is 20, it would be reset to 0, pointing back to the start of the queue

- Outputting an item from a queue

```
def Dequeue():
    global Queue
    global HeadPointer
    if HeadPointer == -1 or HeadPointer == TailPointer:
        return "Queue is empty"
    else:
        HeadPointer = HeadPointer + 1
        return Queue[HeadPointer - 1]
```

- ➔ If HeadPointer == TailPointer, it means all other queue elements have already been outputted and the queue is empty
- ➔ For a circular queue, if the HeadPointer value is 20, it would be reset to 0, pointing back to the start of the queue (when dequeuing, variable storing the number of items in queue is also decremented)

3. Linked Lists

- **Declaring/initialising a linked list and pointer variables**

```
LinkedList = [] #global 2D array
for i in range(19):
    LinkedList.append([-1,i+1])
LinkedList.append([-1,-1])

FirstNode = -1 #global integer
FirstEmpty = 0 #global integer
```

  - ➔ **This code will depend on the specifics of the question, but generally -1 is used to indicate an empty node (index 0 is the data, index 1 is a pointer)**
  - ➔ **FirstNode is set to -1 as there is no node to point to yet (as all are empty)**
  - ➔ **FirstEmpty points to the first empty node (so is set to 0 at start, as it points to the first array element)**
  - ➔ **This particular list has 20 nodes - last node appended has pointer value set to -1, indicating a null pointer**
- **Inserting nodes into a linked list**

```
1  def InsertData():
2      global LinkedList
3      global FirstNode
4      global FirstEmpty
5
6      for i in range(5):
7          if FirstEmpty != -1:
8              Num = int(input("Enter a positive integer: "))
9              nextEmpty = LinkedList[FirstEmpty][1]
10             LinkedList[FirstEmpty][0] = Num
11             LinkedList[FirstEmpty][1] = FirstNode
12             FirstNode = FirstEmpty
13             FirstEmpty = nextEmpty
14
```

  - ➔ **Line 7: only inserts a new node if the list isn't full (if FirstEmpty = -1, it means there are no more empty nodes)**
  - ➔ **Line 9: nextEmpty stores the pointer of the next empty node**
  - ➔ **Line 10: Stores value in empty node**
  - ➔ **Line 11: Node now points to the one before it in the linked list**
  - ➔ **Line 12: FirstNode now points to the node with the newly stored value**
  - ➔ **Line 13: FirstEmpty now points to the next empty node**

- **Removing nodes from a linked list**

```
1  def RemoveData(Num):
2      global LinkedList
3      global FirstNode
4      global FirstEmpty
5
6      if LinkedList[FirstNode][0] == Num:
7          NewFirst = LinkedList[FirstNode][1]
8          LinkedList[FirstNode][1] = FirstEmpty
9          FirstEmpty = FirstNode
10         FirstNode = NewFirst
11     else:
12         if FirstNode != -1:
13             Pointer = FirstNode
14             PreviousNode = -1
15             while Pointer != -1 and LinkedList[Pointer][0] != Num:
16                 PreviousNode = Pointer
17                 Pointer = LinkedList[Pointer][1]
18             if LinkedList[Pointer][0] == Num:
19                 LinkedList[PreviousNode][1] = LinkedList[Pointer][1]
20                 LinkedList[Pointer][1] = FirstEmpty
21                 LinkedList[Pointer][0] = -1
22                 FirstEmpty = Pointer
23
```

- ➔ **Line 6: if the most recent node has the value to be removed...**
- ➔ **Line 7: the first node will now be the one the most recent node is pointing to (this pointer is stored temporarily)**
- ➔ **Line 8: pointer of node will now point to an empty node as it becomes part of the empty list**
- ➔ **Line 9: FirstEmpty will now point to this node**
- ➔ **Line 10: FirstNode will now point to the next node that isn't empty**

- ➔ **Line 12: if the list isn't empty...**
- ➔ **Line 13: Pointer value starts of as FirstNode and will be used to index the nodes in the list**
- ➔ **Line 14: -1 is just a temporary value, gets replaced by value of Pointer later**
- ➔ **Line 15: loop runs while the Pointer hasn't indexed through all the nodes or the node containing the value to be removed hasn't been found yet**
- ➔ **Line 16: the PreviousNode will now contain the value of the Pointer before it is incremented**
- ➔ **Line 17: Pointer gets incremented**
- ➔ **Line 19: node at PreviousNode now points to the node that the one being removed is pointing to**
- ➔ **Line 20: node being removed now points to an empty node as it becomes part of the empty list**
- ➔ **Line 21: value of index 0 is set to -1 to indicate it is empty**

➔ **Line 22: FirstEmpty now points to the removed node**

- **Outputting a linked list**

```
def OutputLinkedList():
    global LinkedList
    global FirstNode
    global FirstEmpty
    Pointer = FirstNode
    while Pointer != -1:
        print(LinkedList[Pointer][0])
        Pointer = LinkedList[Pointer][1]
```

➔ **Pointer value is used to index through all the nodes, starting at FirstNode**

➔ **While loop runs until last node is reached, which will be indicated with a null pointer (-1)**

➔ **Pointer is incremented by being set to the pointer stored in the current node being printed**