Date: 23rd August, 2021

Design and Analysis of Algorithm Lab

2019BTECS00058 Devang
Batch: T7

Assignment: Week 1
Sorting Algorithm

Q1) You are given two sorted arrays, A and B, where A has a large enough buffer at the end to hold B. Write a method to merge B into A in sorted order.

Algorithm:

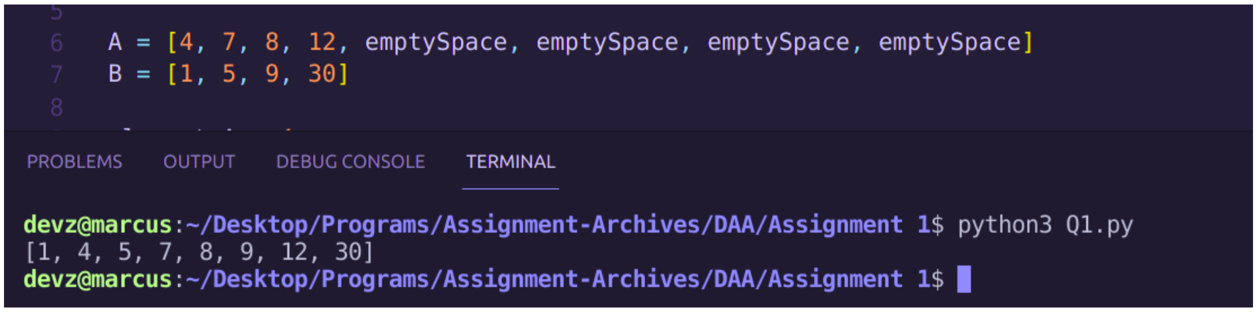
1. Since both arrays are sorted. We compare the largest element of the smaller array with that of the larger.
2. We copy the larger one to the rightmost available index.
3. Then we consider the 2nd largest element of the selected array and largest of the other and repeat the steps till all the elements in the smaller array are copied.

Program:

```
emptySpace = None

A = [4, 7, 8, 12, emptySpace, emptySpace, emptySpace, emptySpace]
B = [1, 5, 9, 30]

elementsA = 4
elementsB = 4

def mergeBintoA(A, B, cA, cB):
 last = cA + cB - 1
```

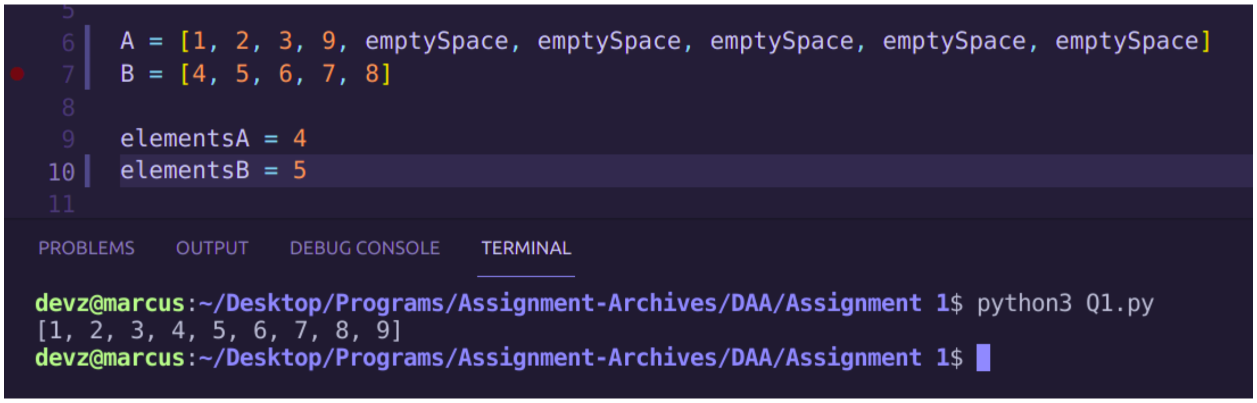
```
indexA = cA - 1
indexB = cB - 1
 while (indexB >= 0) :

  # End of a is greater than end
  # of b
  if (indexA >= 0 and A[indexA] > B[indexB]):

      # Copy Element
      A[last] = A[indexA]
      indexA -= 1
  else:
    # Copy Element
    A[last] = B[indexB]
    indexB -= 1

  # Move indices
  last-= 1
 return A
print(mergeBintoA(A, B, elementsA, elementsB))
```

Output:



```
A = [4, 7, 8, 12, emptySpace, emptySpace, emptySpace, emptySpace]
B = [1, 5, 9, 30]
```

```
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$ python3 Q1.py
[1, 4, 5, 7, 8, 9, 12, 30]
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$
```



```
A = [1, 2, 3, 9, emptySpace, emptySpace, emptySpace, emptySpace, emptySpace]
B = [4, 5, 6, 7, 8]

elementsA = 4
elementsB = 5
```

```
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$ python3 Q1.py
[1, 2, 3, 4, 5, 6, 7, 8, 9]
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$
```

Complexity of proposed algorithm (Time & Space):
Space Complexity: O(1)
Time Complexity: O(m+n)

Your comment (How is your solution optimal?)
The algorithm is very optimised since it does not use any extra space and the complexity is pretty much perfect. It covers all edge-cases and is very scalable.

Q2) Write a method to sort an array of string so that all the anagrams are next to each other.

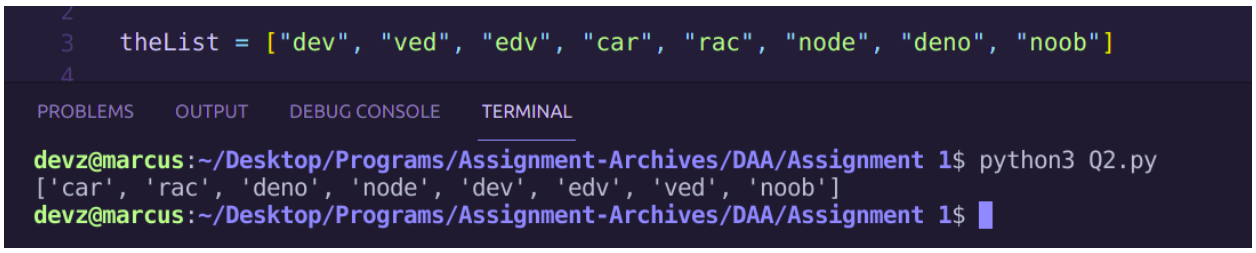
Algorithm:

1. We build a map that consists of a list of all anagrams as its value and the most lexicographically small type in the given list as its key.
2. Sort all the list values in lexicographic order.
3. Sort the list of keys in lexicographic order.
4. Append a list with the value-list of every key of the map in the sequence of the sorted list of keys.

Program:

```python
theList = ["dev", "ved", "edv", "car", "rac", "node", "deno", "noob"]

def isAnagram(a, b):
 a = [ch for ch in a]
 b = [ch for ch in b]
 a.sort()
 b.sort()
 if a == b:
   return True
 return False

def sortListOfStringsAnagrams(theList):
 theList.sort()
 anagramDic = {}
 for ele in theList:
   c=0
   for k in anagramDic.keys():
     if isAnagram(k, ele):
       anagramDic[k].append(ele)
       c+=1
       break
   if c == 0:
     anagramDic[ele] = [ele]
 anagramSequence = list(anagramDic.keys())
 anagramSequence.sort()
 finalList = []
 for seq in anagramSequence:
   specificAnagramList = anagramDic[seq]
   specificAnagramList.sort()
   finalList += specificAnagramList

 return finalList

print(sortListOfStringsAnagrams(theList))
```

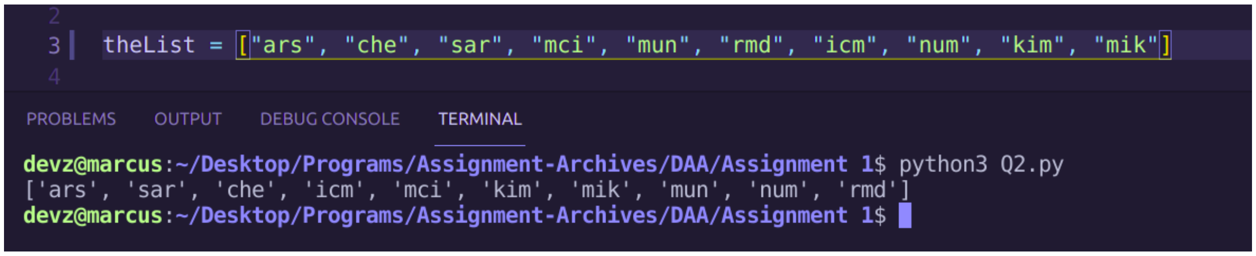
Output:



```
theList = ["dev", "ved", "edv", "car", "rac", "node", "deno", "noob"]
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$ python3 Q2.py
['car', 'rac', 'deno', 'node', 'dev', 'edv', 'ved', 'noob']
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$
```



```
theList = ["ars", "che", "sar", "mci", "mun", "rmd", "icm", "num", "kim", "mik"]
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$ python3 Q2.py
['ars', 'sar', 'che', 'icm', 'mci', 'kim', 'mik', 'mun', 'num', 'rmd']
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$
```

Complexity of proposed algorithm (Time & Space):
Space Complexity: O(n)
Time Complexity: O(n log(n))

Your comment (How is your solution optimal?)
The algorithm is extremely intuitive and robust - covering all edge cases. It's space optimal but time complexity could be better since it requires multiple-sorts.

Q3) Given a sorted array of n integers that has been rotated an unknown number of times, write code to find an element in the array. You may assume that the array was originally sorted in increasing order.

Algorithm:

1. We apply a modified form of the binary-search algorithm taking in mind the sorted status of sub-arrays. We require the lower and upper indices, the list and integer to search as the parameters
2. We find the middle point of the list.

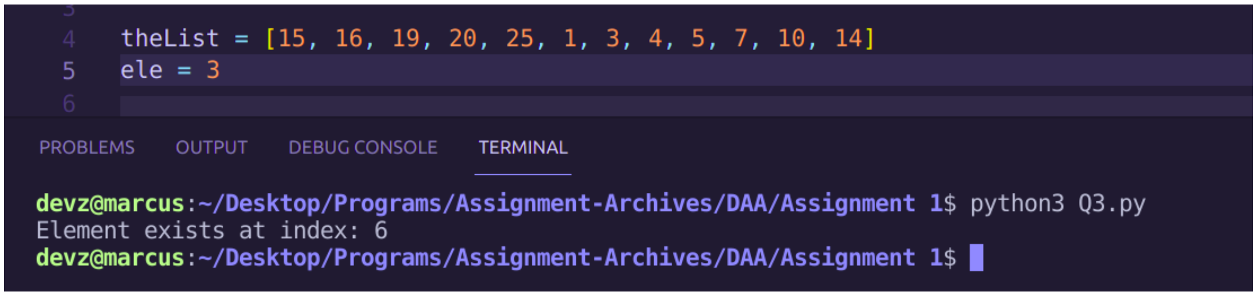
3. If the required element is in the middle, we return its index.
4. Else we check if the left-half of the list is sorted, if yes - we recur the algorithm with them as the space, or else we recur from the middle point to the higher limit.
5. If the left-half is not sorted, it means that the right-half is. Therefore we check if the required element lies from the middle point to the higher limit. If yes - we recur the search there or recurse from lower limit to the middle point.

Program:

```python
theList = [5, 6, 7, 8, 1, 2, 3]
ele = 3

def searchElement(theList, l, h, ele):
 try:
   if l > h:
       return -1

   mid = (l + h) // 2
   if theList[mid] == ele:
       return mid

   # sub-array is sorted
   if theList[l] <= theList[mid]:
       if ele >= theList[l] and ele <= theList[mid]:
           return searchElement(theList, l, mid-1, ele)
       return searchElement(theList, mid + 1, h, ele)

   # sub-array is not sorted, the other side of it must be sorted
   if ele >= theList[mid] and ele <= theList[h]:
       return searchElement(theList, mid + 1, h, ele)
   return searchElement(theList, l, mid-1, ele)
  except:
   return -1
```

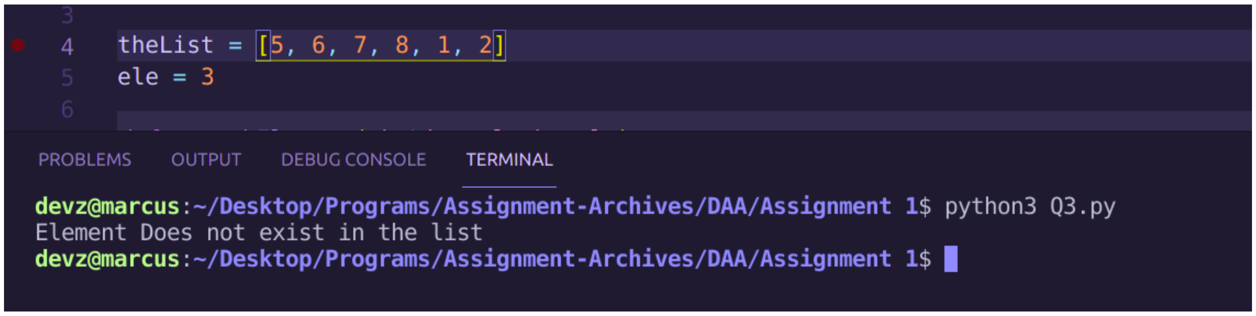
```
val = searchElement(theList, 0, len(theList), ele)
if val == -1:
 print("Element Does not exist in the list")
else:
 print("Element exists at index: "+str(val))
```

Output:



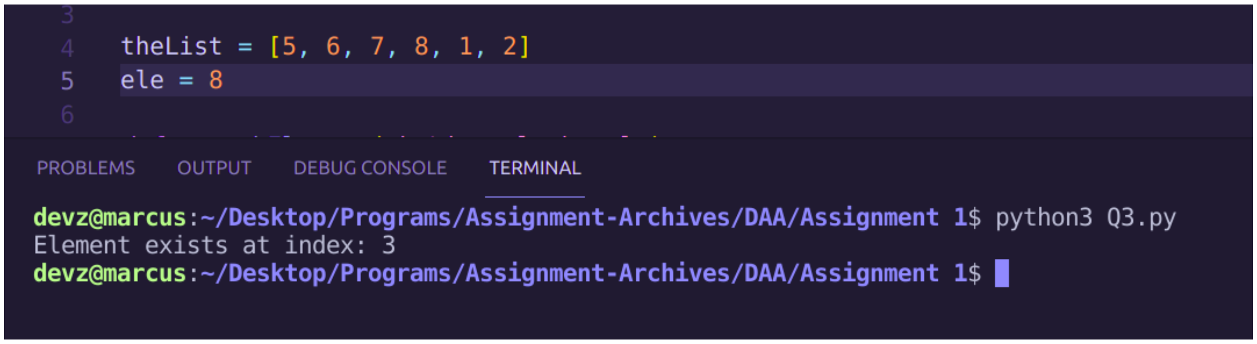
```
theList = [15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10, 14]
ele = 3
```

```
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$ python3 Q3.py
Element exists at index: 6
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$
```



```
theList = [5, 6, 7, 8, 1, 2]
ele = 3
```

```
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$ python3 Q3.py
Element Does not exist in the list
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$
```



```
theList = [5, 6, 7, 8, 1, 2]
ele = 8
```

```
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$ python3 Q3.py
Element exists at index: 3
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$
```

Complexity of proposed algorithm (Time & Space):
Space Complexity: O(1)
Time Complexity: O(log (n))

Your comment (How is your solution optimal?)
The algorithm is extremely efficient and makes use of the property that only one unsorted component may exist in a division. The time complexity is also optimal which makes this algorithm extremely scalable.

Q4) Imagine you have a 20GB file with one string per line. Explain how you would sort the file.

We cannot bring all 20GB of data in the memory at once for sorting. Therefore, we make use of the K-way Merge Sort Algorithm.
We split the data in K parts of equal memory where each contains M variables that can be fit into the memory. Beyond this, we sort each of the K parts by some other internal sort algorithm (Radix or Quick). The complexity for that would be $O(K \times M\log M)$.
Following this, we use the K-way merge to merge all the K parts after sorting. The complexity of that step would be $O(N\log K)$.

Therefore, the overall complexity would be $O(K \times M\log M)$.

Q5) Given a sorted array of string which is interspersed with empty string, write a method to find the location of a given string.

Algorithm:
1. We shall be making use of a modified form of Binary Search algorithm.
2. We first lookup the string in the middle, if it is empty - we look for the closest non-empty string. If the middle non-empty string is equal to the required, we return the index.

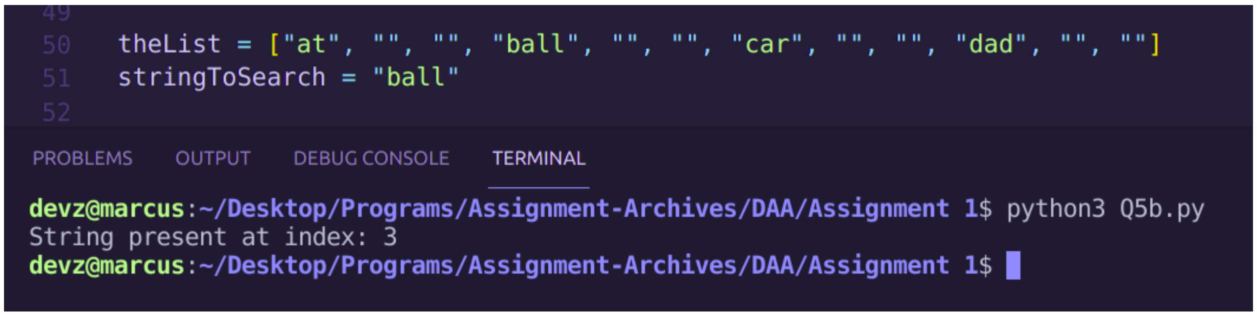
3. Else, we recurse to the left or right side depending on the lexicographic order of the strings.

Program:

```
def compareStrings(str1, str2):
 i = 0
 while i < len(str1) - 1 and str1[i] == str2[i]:
     i += 1
 if str1[i] > str2[i]:
     return -1

 return str1[i] < str2[i]
 def searchStr(arr, string, first, last):
 try:
   if first > last:
       return -1

   # Middle point
   mid = (last + first) // 2

   # If mid is empty - we look for the closest non-empty element
   if len(arr[mid]) == 0:
       # If mid is empty, search in both sides of mid and find the closest
non-empty string, and set mid w.r.t. that.
       left, right = mid - 1, mid + 1
       while True:

           if left < first and right > last:
               return -1

           if right <= last and len(arr[right]) != 0:
               mid = right
               break

           if left >= first and len(arr[left]) != 0:
               mid = left
               break
```

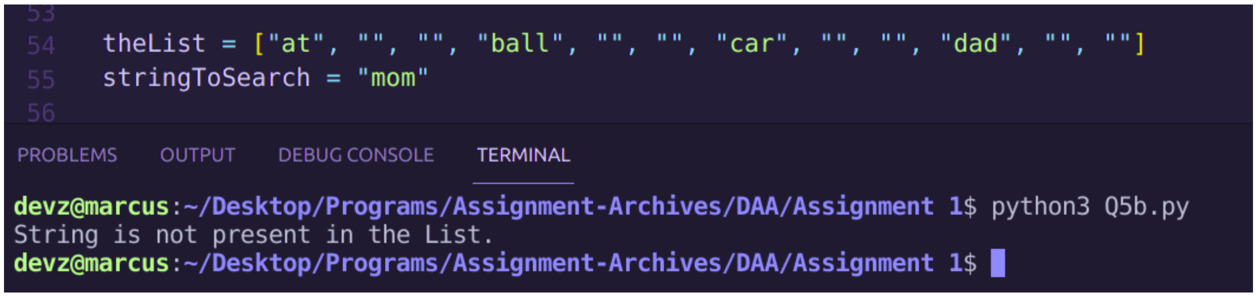
```python
           right += 1
           left -= 1
    # If str is found at mid
   if compareStrings(string, arr[mid]) == 0:
       return mid

   # If str is greater than mid
   if compareStrings(string, arr[mid]) < 0:
       return searchStr(arr, string, mid+1, last)

   # If str is smaller than mid
   return searchStr(arr, string, first, mid-1)
  except:
   return -1


theList = ["at", "", "", "ball", "", "", "car", "", "", "dad", "", ""]
stringToSearch = "mom"


ans = searchStr(theList, stringToSearch, 0, len(theList))
if ans == -1:
 print("String is not present in the List.")
else:
 print("String present at index: "+str(ans))
```

Output:



```
theList = ["at", "", "", "ball", "", "", "car", "", "", "dad", "", ""]
stringToSearch = "ball"
```

```
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$ python3 Q5b.py
String present at index: 3
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$
```



```
theList = ["at", "", "", "ball", "", "", "car", "", "", "dad", "", ""]
stringToSearch = "mom"
```

```
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$ python3 Q5b.py
String is not present in the List.
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$
```

Complexity of proposed algorithm (Time & Space):
Space Complexity: O(1)
Time Complexity: O(L (log N))

Your comment (How is your solution optimal?)
The solution is robust and scalable due to its binary search template.

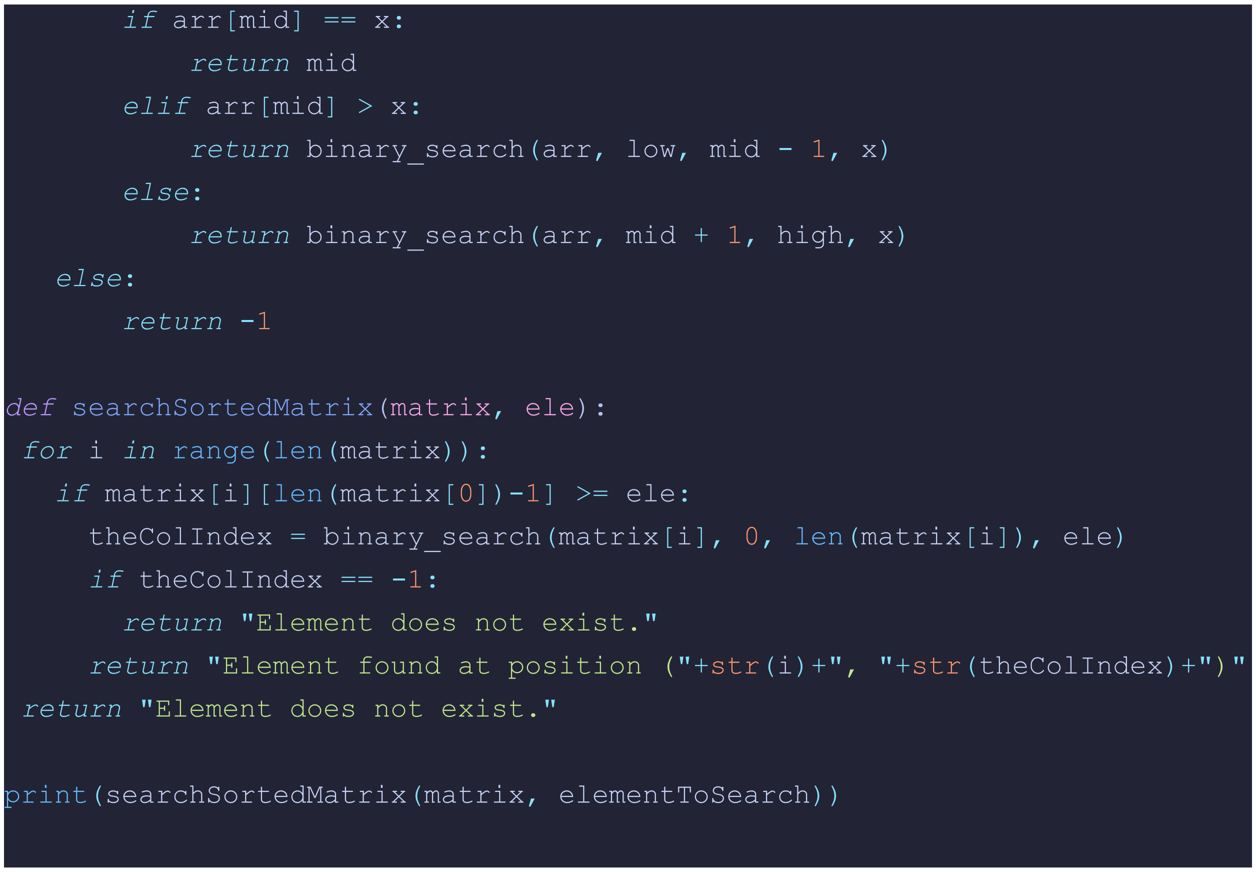
Q6) Given an M*N matrix in which each row and each column is sorted in ascending order, write a method to find an element.

Algorithm:

1. Traverse the last elements of every row and look for the value just greater than the required element.
2. Once found, perform binary search on that row for the element and return the index.

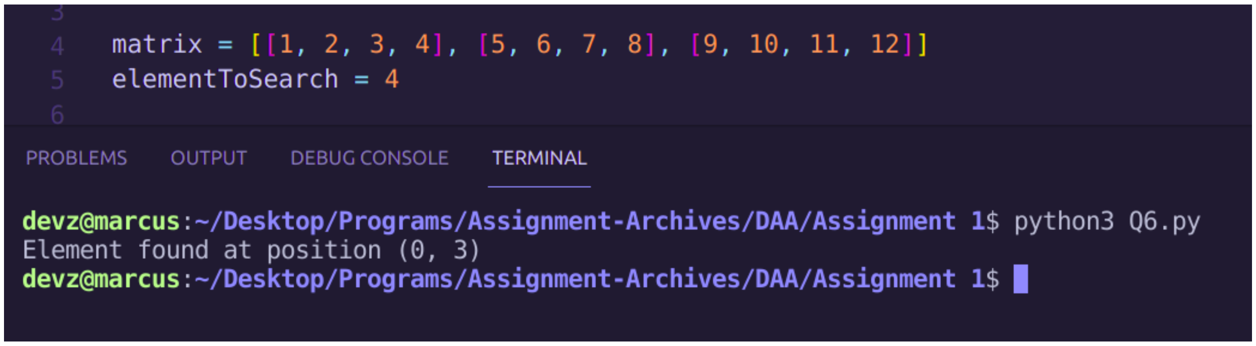
Program:

```
matrix = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
elementToSearch = 1

def binary_search(arr, low, high, x):
    if high >= low:
        mid = (high + low) // 2
```



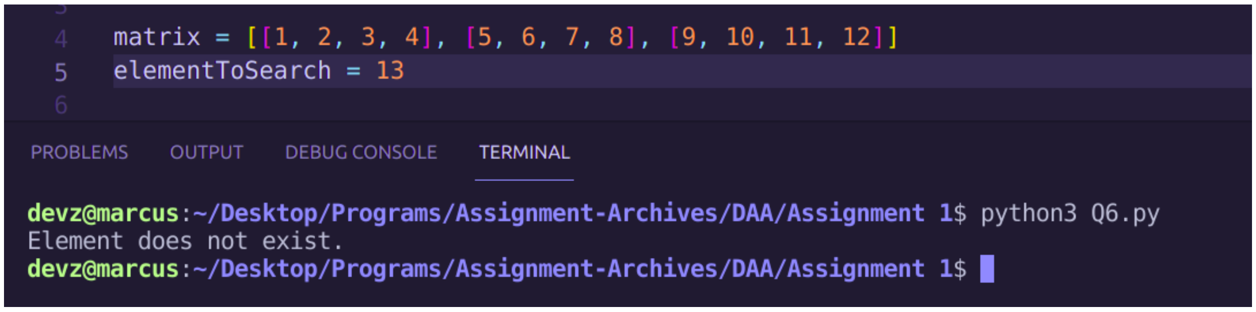
```
       if arr[mid] == x:
           return mid
       elif arr[mid] > x:
           return binary_search(arr, low, mid - 1, x)
       else:
           return binary_search(arr, mid + 1, high, x)
   else:
       return -1

def searchSortedMatrix(matrix, ele):
 for i in range(len(matrix)):
   if matrix[i][len(matrix[0])-1] >= ele:
     theColIndex = binary_search(matrix[i], 0, len(matrix[i]), ele)
     if theColIndex == -1:
       return "Element does not exist."
     return "Element found at position ("+str(i)+", "+str(theColIndex)+")"
 return "Element does not exist."

print(searchSortedMatrix(matrix, elementToSearch))
```

Output:



```
matrix = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
elementToSearch = 4
```

```
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$ python3 Q6.py
Element found at position (0, 3)
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$
```



```
matrix = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
elementToSearch = 13
```

```
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$ python3 Q6.py
Element does not exist.
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$
```

Complexity of proposed algorithm (Time & Space):
Space Complexity: O(1)
Time Complexity: O(N (log N))

Your comment (How is your solution optimal?)
The solution takes advantage of the binary search algorithm alongside the property of the sorted 2-D arrays. Therefore, it's very optimal and also robust to edge-cases.

Q7) A circus is designing a tower routine consisting of people standing atop one another's shoulders. For practical and aesthetic reasons, each person must be both shorter and lighter than the person below him or her. Given the heights and weight of each circus, write a method to compute the largest possible number of people in such a tower.

Algorithm:

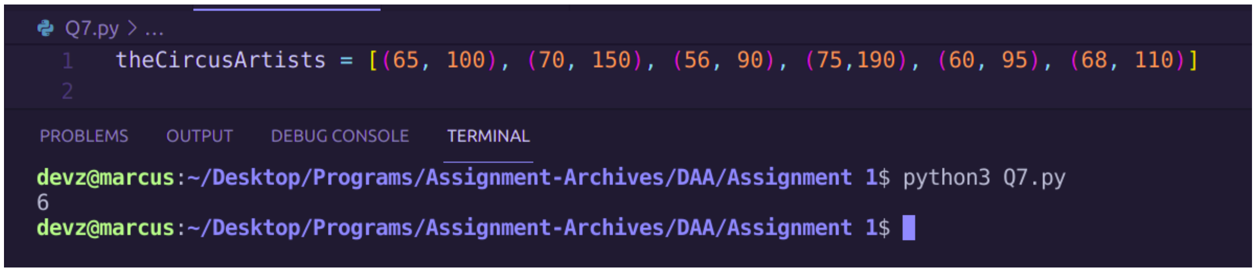
1. Sort the list of values with respect to their weights and heights in the respective priority.
2. Set a counter and traverse through the sorted list and increment when the previous height and weight were both lesser than the current ones.

Program:

```
theCircusArtists = [(65, 100), (70, 150), (56, 90), (75,190), (60, 95),
(68, 110)]

def sizeOfTallestRoutine(theCircusArtists):
  # sort with 1st param then 2nd
```

```
 theCircusArtists = sorted(theCircusArtists, key=lambda element:
(element[0], element[1]))
  theCount = 0
 prev1, prev2 = -1, -1
 for theCircusArtist in theCircusArtists:
   if theCircusArtist[0] > prev1 and theCircusArtist[1] > prev2:
     theCount += 1
     prev1, prev2 = theCircusArtist[0], theCircusArtist[1]

 return theCount

print(sizeOfTallestRoutine(theCircusArtists))
```

Output:

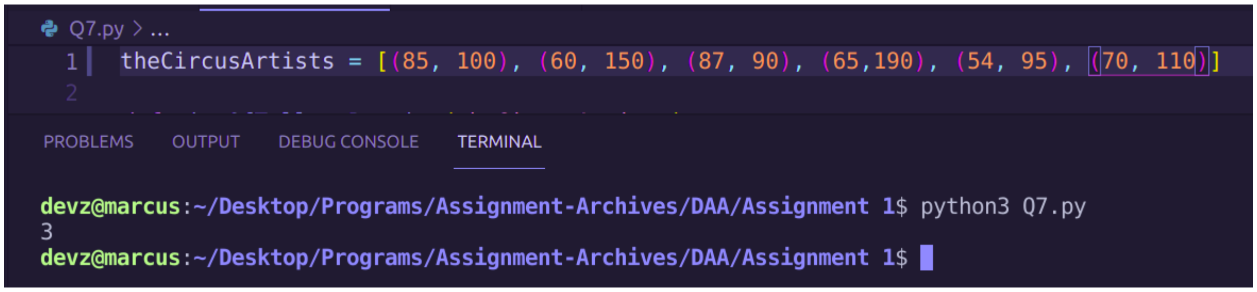


Q7.py > ...

```
theCircusArtists = [(65, 100), (70, 150), (56, 90), (75,190), (60, 95), (68, 110)]
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$ python3 Q7.py
6
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$ 
```



Q7.py > ...

```
theCircusArtists = [(85, 100), (60, 150), (87, 90), (65,190), (54, 95), (70, 110)]
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$ python3 Q7.py
3
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$ 
```

Complexity of proposed algorithm (Time & Space):
Space Complexity: O(1)
Time Complexity: O(N log(N))

Your comment (How is your solution optimal?)

The solution is very intuitive and optimised due to the sorting of the list. It's very robust and scalable.

Q8) Imagine you are reading in a stream of integers. Periodically, you wish to be able to look up the rank of number x (the number of values less than or equal to x). Implement the data structures and algorithms to support these operations. That is, Implement the method track (int x), which is called when each number is generated, and the method getRankOfNumber (int x), which returns the number of values less than or equal to x (not including x itself).

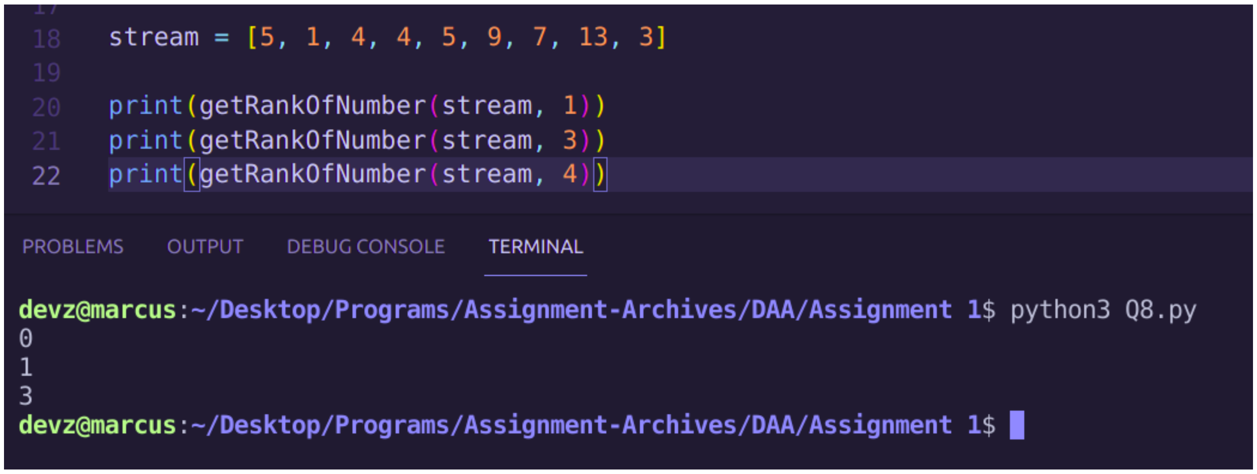
Algorithm:

1. Map all the elements in the list and initialise their values to zero and keep incrementing the values based on the frequency of the elements.
2. Sort the list.
3. Look for the required element in the list and as we find, result would be the index + frequency(required element) - 1.
4. If that number doesn't exist and we traverse to a number larger than the required, we return the index.

Program:

```
def getRankOfNumber(stream, num):
 stream.sort()
 dic = {}
 for s in stream:
   if s in dic.keys():
     dic[s] += 1
   else:
     dic[s] = 1
 for i in range(len(stream)):
   if stream[i] == num:
      return i+(dic[num]-1)
    if stream[i] > num:
      return i
  if num > stream[len(stream)-1]:
    return len(stream)
  return 0

stream = [5, 1, 4, 4, 5, 9, 7, 13, 3]

print(getRankOfNumber(stream, 1))
print(getRankOfNumber(stream, 3))
print(getRankOfNumber(stream, 4))
```

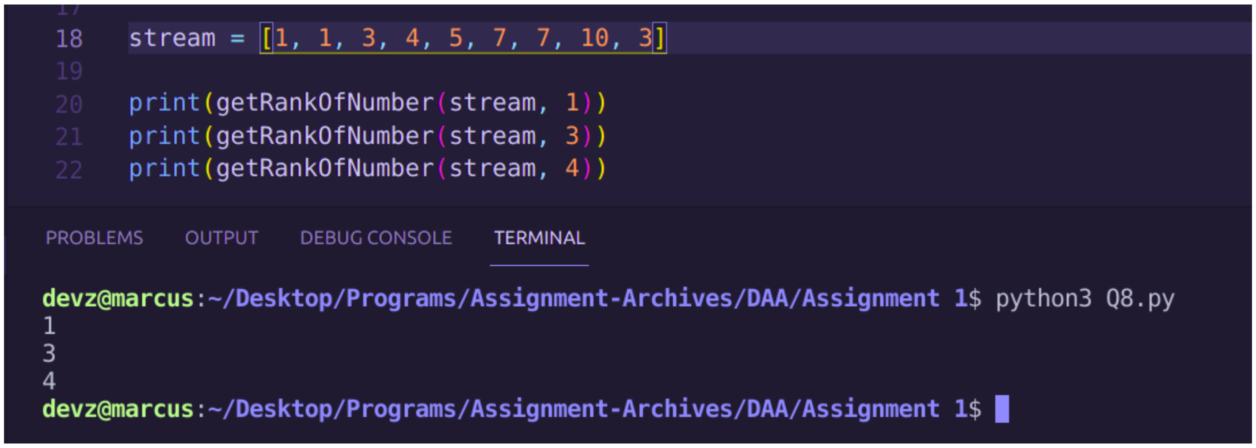
Output:



```
stream = [5, 1, 4, 4, 5, 9, 7, 13, 3]

print(getRankOfNumber(stream, 1))
print(getRankOfNumber(stream, 3))
print(getRankOfNumber(stream, 4))
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$ python3 Q8.py
0
1
3
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$ 
```



```
stream = [1, 1, 3, 4, 5, 7, 7, 10, 3]

print(getRankOfNumber(stream, 1))
print(getRankOfNumber(stream, 3))
print(getRankOfNumber(stream, 4))
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$ python3 Q8.py
1
3
4
devz@marcus:~/Desktop/Programs/Assignment-Archives/DAA/Assignment 1$ 
```

Complexity of proposed algorithm (Time & Space):
Space Complexity: O(N)
Time Complexity: O(N log(N))

Your comment (How is your solution optimal?)
Solution is intuitive and robust and scalable by making use of maps and sorting. It makes use of efficient data structures.