Problem Statement

You have gone fishing in the river. The river is located to the right of the buildings. There are 'n' buildings in a line. You are given an integer array of heights of size n that represents the heights of the buildings in the line. A building has a river view if the building can see the river without obstructions. Formally, a building has a river view if all the buildings to its right have a smaller height. Return a list of indices (0-indexed) of buildings that have a river view, sorted in increasing order.

Visualization of the Given Problem

Height	25	20	45	30	35	40	30	30	50	30	30	25
Index	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]

- For visualization purposes, we assume that there are 12 buildings in a row and the river is on the right side of the buildings.
- The height of each building is stored in an array A.
- As given, a building can view the river if its height is greater than the height of buildings on its right side. Therefore, the last building, i.e. the building at A[11] will always be able to see the river as we do not have any building after that.
- If the height of the building is greater than the height of all the building to its right, then the given building has the river view.

Expected output

- A list of indices (0-indexed) of buildings that have a river view, sorted in increasing order.
- Analyzing the time complexity of the algorithm.

Approach to the Solution

Data Structure used with Justification

To solve this problem, we used **Stack** as the data structure because it can:

- Help us in reversing the a given order because of its Last In – First Out way for managing data.
- Automatically clean up the objects.
- Control and handle memory allocation and deallocation using several methods like
  - __init__(self, size)
  - push(self, value)
  - pop(self)
  - isEmpty(self)
  - len__(self)

Psuedo Code & It's Explanation

1. Read heights of buildings from input file -> heightList
2. Initialize empty stack with size equal to the number of buildings -> stack
3. Set maxHeight = Height of the last building in heightList
4. Push the index of the last building onto stack
5. for each building in heightList in reverse order (except the last building):
   5.1. if the height of the current building is greater than maxHeight:
   5.2. Push the index of the current building onto stack
   5.3. Update maxHeight with the height of the current building
6. Initialize Empty List reversedList
7. While stack is not empty:
8. Pop an element from stack and append the poppedElement to reversedList
9. Write reversedList to output file

Explanation:

- The above pseudo code reads a list of building heights from a file, creates a Stack object, and iterates through the list of building heights in reverse order.
- For each building height, the code pushes the index of the building to the stack if the building's height is greater than the current maximum height.
- Finally, the code pops all elements of the stack and writes them to an output file.
- The resulting list of indices represents the buildings that have a river view, in the order in which they were added to the stack (i.e., in increasing order of index).

Platform used

Python 3.7

Alignment of content/ Deliverables

- inputPS02.txt file used for testing.
- outputPS02.txt file generated while testing.
- main.py file contains the python program solving this problem

Time Complexity of Each Operation

- The time complexity of the Stack class **depends on the number of elements in the stack**.
- The ***__init__, isEmpty,*** and ***__len__*** methods have a constant **time complexity of O(1)**, since they perform a constant number of operations regardless of the size of the stack.
- The **push method** has an average time complexity of O(1), since it usually takes a constant amount of time to add an element to the top of the stack. However, if the stack is full, the method raises an exception, which takes O(1) time. Therefore, the **worst-case time complexity of the push method is O(1).**
- The **pop method** has an average time complexity of O(1), since it usually takes a constant amount of time to remove an element from the top of the stack. However, if the stack is empty, the method raises an exception, which takes O(1) time. Therefore, the **worst-case time complexity of the pop method is O(1).**
- The **main loop** in the rest of the code has a **time complexity of O(N),** where N is the number of elements in the list of building heights. This is because the loop iterates through all elements in the list.
- Overall, the time complexity of this code is O(N) for the main loop, plus the constant time complexities of the Stack class methods, which results in an **overall time complexity of O(N).** **<u>This means that the running time of the code grows linearly with the size of the input (i.e., the number of building heights).</u>**

Alternate Way of Modelling the Problem and its Cost Implications

<u>There are different options here:</u>

- **Using List -** This problem can also be solved by creating a List and appending the indices of the building with river view to it instead of adding it to the Stack. However, this introduces an **additional effort of sorting the list** in incremental order of the indices.
- **Using Binary Search Tree -** This problem can also be solved by creating a Binary Search Tree and inserting the indices of the building having river view. The advantage is you can traverse tree in in-order fashion to get list of buildings having Riverview no additional cost for sorting needed. However, **tree insertion operation is costly and will take O(n) time complexity.**