'K' Closest Numbers (medium)

We'll cover the following

- Problem Statement
- Try it yourself
- Solution
- Code
  - Time complexity
  - Space complexity
- Alternate Solution using Two Pointers
  - Time complexity
  - Space complexity

Problem Statement

Given a sorted number array and two integers 'K' and 'X', find 'K' closest numbers to 'X' in the array. Return the numbers in the sorted order. 'X' is not necessarily present in the array.

Example 1:

```
Input: [5, 6, 7, 8, 9], K = 3, X = 7
Output: [6, 7, 8]
```

Example 2:

```
Input: [2, 4, 5, 6, 9], K = 3, X = 6
Output: [4, 5, 6]
```

Example 3:

```
Input: [2, 4, 5, 6, 9], K = 3, X = 10
Output: [5, 6, 9]
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```
def find_closest_elements(arr, K, X):
  result = []
  # TODO: Write your code here
  return result
```

```


def main():
  print("'K' closest numbers to 'X' are: " +
        str(find_closest_elements([5, 6, 7, 8, 9], 3, 7)))
  print("'K' closest numbers to 'X' are: " +
        str(find_closest_elements([2, 4, 5, 6, 9], 3, 6)))
  print("'K' closest numbers to 'X' are: " +
        str(find_closest_elements([2, 4, 5, 6, 9], 3, 10)))


main()
```

Solution

This problem follows the Top 'K' Numbers (https://www.educative.io/collection/page/5668639101419520/5671464854355968/5728885882748928/) pattern. The biggest difference in this problem is that we need to find the closest (to 'X') numbers compared to finding the overall largest numbers. Another difference is that the given array is sorted.

Utilizing a similar approach, we can find the numbers closest to 'X' through the following algorithm:

1. Since the array is sorted, we can first find the number closest to 'X' through **Binary Search**. Let's say that number is 'Y'.
2. The 'K' closest numbers to 'Y' will be adjacent to 'Y' in the array. We can search in both directions of 'Y' to find the closest numbers.
3. We can use a heap to efficiently search for the closest numbers. We will take 'K' numbers in both directions of 'Y' and push them in a **Min Heap** sorted by their absolute difference from 'X'. This will ensure that the numbers with the smallest difference from 'X' (i.e., closest to 'X') can be extracted easily from the **Min Heap**.
4. Finally, we will extract the top 'K' numbers from the **Min Heap** to find the required numbers.

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```
from heapq import *


def find_closest_elements(arr, K, X):
  index = binary_search(arr, X)
  low, high = index - K, index + K

  low = max(low, 0)  # 'low' should not be less than zero
  # 'high' should not be greater the size of the array
```

```
  high = min(high, len(arr) - 1)

  minHeap = []
  # add all candidate elements to the min heap, sorted by their absolute difference from '
  for i in range(low, high+1):
    heappush(minHeap, (abs(arr[i] - X), arr[i]))

  # we need the top 'K' elements having smallest difference from 'X'
  result = []
  for _ in range(K):
    result.append(heappop(minHeap)[1])

  result.sort()
  return result


def binary_search(arr,  target):
  low, high = 0, len(arr) - 1
  while low <= high:
```

Time complexity #

The time complexity of the above algorithm is $O(logN + K * logK)$. We need $O(logN)$ for Binary Search and $O(K * logK)$ to insert the numbers in the **Min Heap**, as well as to sort the output array.

Space complexity #

The space complexity will be $O(K)$, as we need to put a maximum of $2K$ numbers in the heap.

Alternate Solution using Two Pointers

After finding the number closest to ‘X’ through **Binary Search**, we can use the **Two Pointers** approach to find the ‘K’ closest numbers. Let’s say the closest number is ‘Y’. We can have a `left` pointer to move back from ‘Y’ and a `right` pointer to move forward from ‘Y’. At any stage, whichever number pointed out by the `left` or the `right` pointer gives the smaller difference from ‘X’ will be added to our result list.

To keep the resultant list sorted we can use a **Queue**. So whenever we take the number pointed out by the `left` pointer, we will append it at the beginning of the list and whenever we take the number pointed out by the `right` pointer we will append it at the end of the list.

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```


def binary_search(arr,  target):
  low, high = 0, len(arr) - 1
  while low <= high:
    mid = int(low + (high - low) / 2)
    if arr[mid] == target:
      return mid
    if arr[mid] < target:
      low = mid + 1
```

```
    else:
      high = mid - 1
  if low > 0:
    return low - 1
  return low


def main():
  print("'K' closest numbers to 'X' are: " +
        str(find_closest_elements([5, 6, 7, 8, 9], 3, 7)))
  print("'K' closest numbers to 'X' are: " +
        str(find_closest_elements([2, 4, 5, 6, 9], 3, 6)))
  print("'K' closest numbers to 'X' are: " +
        str(find_closest_elements([2, 4, 5, 6, 9], 3, 10)))


main()
```

Time complexity

The time complexity of the above algorithm is $O(logN + K)$. We need $O(logN)$ for Binary Search and $O(K)$ for finding the 'K' closest numbers using the two pointers.

Space complexity

If we ignoring the space required for the output list, the algorithm runs in constant space $O(1)$.

Back
Kth Largest Number in a Stream (medi...

Next
Maximum Distinct Elements (medium)

Mark as Completed

Report an Issue

Ask a Question
(https://discuss.educative.io/tag/k-closest-numbers-medium__pattern-top-k-elements__grokking-the-coding-interview-patterns-for-coding-questions)