Top 'K' Numbers (easy)

We'll cover the following

Problem Statement

Given an unsorted array of numbers, find the 'K' largest numbers in it.

Note: For a detailed discussion about different approaches to solve this problem, take a look at Kth Smallest Number.

Example 1:

```
Input: [3, 1, 5, 12, 2, 11], K = 3
Output: [5, 12, 11]
```

Example 2:

```
Input: [5, 12, 11, -1, 12], K = 3
Output: [12, 11, 12]
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```python
from heapq import *


def find_k_largest_numbers(nums, k):
  result = []
  # TODO: Write your code here
  return result


def main():

  print("Here are the top K numbers: " +
        str(find_k_largest_numbers([3, 1, 5, 12, 2, 11], 3)))

  print("Here are the top K numbers: " +
        str(find_k_largest_numbers([5, 12, 11, -1, 12], 3)))


main()
```

Run | Save | Reset

Solution

A brute force solution could be to sort the array and return the largest K numbers. The time complexity of such an algorithm will be $O(N * logN)$ as we need to use a sorting algorithm like Timsort if we use Java's `Collection.sort()`. Can we do better than that?

The best data structure that comes to mind to keep track of top 'K' elements is Heap. Let's see if we can use a heap to find a better algorithm.

If we iterate through the array one element at a time and keep 'K' largest numbers in a heap such that each time we find a larger number than the smallest number in the heap, we do two things:

1. Take out the smallest number from the heap, and
2. Insert the larger number into the heap.

This will ensure that we always have 'K' largest numbers in the heap. The most efficient way to repeatedly find the smallest number among a set of numbers will be to use a min-heap. As we know, we can find the smallest number in a min-heap in constant time $O(1)$, since the smallest number is always at the root of the heap. Extracting the smallest number from a min-heap will take $O(logN)$ (if the heap has 'N' elements) as the heap needs to readjust after the removal of an element.

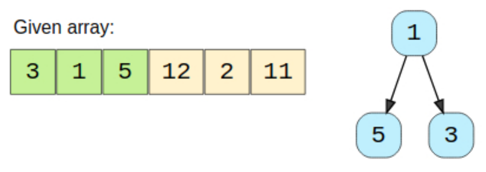
Let's take Example-1 to go through each step of our algorithm:

Given array: [3, 1, 5, 12, 2, 11], and K=3

1. First, let's insert 'K' elements in the min-heap.
2. After the insertion, the heap will have three numbers [3, 1, 5] with '1' being the root as it is the smallest element.
3. We'll iterate through the remaining numbers and perform the above-mentioned two steps if we find a number larger than the root of the heap.
4. The 4th number is '12' which is larger than the root (which is '1'), so let's take out '1' and insert '12'. Now the heap will have [3, 5, 12] with '3' being the root as it is the smallest element.
5. The 5th number is '2' which is not bigger than the root of the heap ('3'), so we can skip this as we already have top three numbers in the heap.
6. The last number is '11' which is bigger than the root (which is '3'), so let's take out '3' and insert '11'. Finally, the heap has the largest three numbers: [5, 12, 11]

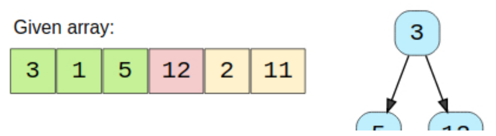
As discussed above, it will take us $O(logK)$ to extract the minimum number from the min-heap. So the overall time complexity of our algorithm will be $O(K * logK + (N - K) * logK)$ since, first, we insert 'K' numbers in the heap and then iterate through the remaining numbers and at every step, in the worst case, we need to extract the minimum number and insert a new number in the heap. This algorithm is better than $O(N * logN)$.

Here is the visual representation of our algorithm:



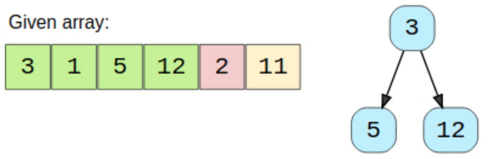
Insert the first three numbers in the heap

1 of 4



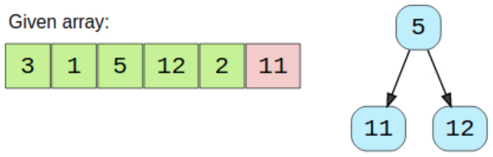
The root '1' is smaller than '12', so take '1' out and insert '12'

2 of 4



Skip '2' as it is not bigger than the root '3'

3 of 4



The root '3' is smaller than '11', so take '3' out and insert '11'

4 of 4

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```python
from heapq import *


def find_k_largest_numbers(nums, k):
  minHeap = []
  # put first 'K' numbers in the min heap
  for i in range(k):
    heappush(minHeap, nums[i])

  # go through the remaining numbers of the array, if the number from the array is bigger than the
  # top(smallest) number of the min-heap, remove the top number from heap and add the number from array
  for i in range(k, len(nums)):
    if nums[i] > minHeap[0]:
      heappop(minHeap)
      heappush(minHeap, nums[i])

  # the heap has the top 'K' numbers, return them in a list
  return list(minHeap)


def main():

  print("Here are the top K numbers: " +
        str(find_k_largest_numbers([3, 1, 5, 12, 2, 11], 3)))

  print("Here are the top K numbers: " +
        str(find_k_largest_numbers([5, 12, 11, -1, 12], 3)))


main()
```

Run | Save | Reset

Time complexity

As discussed above, the time complexity of this algorithm is $O(K * logK + (N - K) * logK)$, which is asymptotically equal to $O(N * logK)$

Space complexity

The space complexity will be $O(K)$ since we need to store the top ‘K’ numbers in the heap.