Topic - Heaps

Difficulty Med : Min Heap Construction

Implement a MinHeap class that supports: Building a Min Heap from an input array of integers. Inserting integers in the heap. Removing the heap's minimum / root value. Peeking at the heap's minimum / root value. Sifting integers up and down the heap, which is to be used when inserting and removing values. Note that the heap should be represented in the form of an array.

Sample Usage

```
array = [48, 12, 24, 7, 8, -5, 24, 391, 24, 56, 2, 6, 8, 41]

// All operations below are performed sequentially.
MinHeap(array): - // instantiate a MinHeap (calls the buildHeap method and populates the heap)
buildHeap(array): - [-5, 2, 6, 7, 8, 8, 24, 391, 24, 56, 12, 24, 48, 41]
insert(76): - [-5, 2, 6, 7, 8, 8, 24, 391, 24, 56, 12, 24, 48, 41, 76]
peek(): -5
remove(): -5 [2, 7, 6, 24, 8, 8, 24, 391, 76, 56, 12, 24, 48, 41]
peek(): 2
remove(): 2 [6, 7, 8, 24, 8, 24, 24, 391, 76, 56, 12, 41, 48]
peek(): 6
insert(87): - [6, 7, 8, 24, 8, 24, 24, 391, 76, 56, 12, 41, 48, 87]
```

Hint:

For the buildHeap(), remove(), and insert() methods of the Heap, you will need to use the siftDown() and siftUp() methods. These two methods should essentially allow you to take any node in the heap and move it either down or up in the heap until it's in its final, appropriate position. This can be done by comparing the node in question to its child nodes in the case of siftDown() or to its parent node in the case of siftUp(). In an array-based Heap, you can easily access a node's children nodes and parent node by using the nodes' indices. If a node is located at index i, then its children nodes are located at indices 2 * i + 1 and 2 * i + 2, and its parent node is located at index Math.floor((i - 1) / 2). To implement the buildHeap() method, you can either sift every node in the input array down to its final, correct position, or you can sift every node in the input array up to its final, correct position. What are the runtime implications of both approaches? Which methods (siftDown() or siftUp()) will insert() and remove() utilize? What about peek()?

Optimal Space & Time Complexity:

BuildHeap: O(n) time | O(1) space - where n is the length of the input array SiftDown: O(log(n)) time | O(1) space - where n is the length of the heap SiftUp: O(log(n)) time | O(1) space - where n is the length of the heap Peek: O(1) | O(1) Remove: O(log(n)) time | O(1) space - where n is the length of the heap Insert: O(log(n)) time | O(1) space - where n is the length of the heap.

Difficulty Med : Continuous Median

Write a ContinuousMedianHandler class that supports: The continuous insertion of numbers with the insert method. The instant (O(1) time) retrieval of the median of the numbers that have been inserted thus far with the getMedian method. The getMedian method has already been written for you. You simply have to write the insert method. The median of a set of numbers is the "middle" number when the numbers are ordered from smallest to greatest. If there's an odd number of numbers in the set, as in {1, 3, 7}, the median is the number in the middle (3 in this case); if there's an even number of numbers in the set, as in {1, 3, 7, 8}, the median is the average of the two middle numbers ((3 + 7) / 2 == 5 in this case).

Sample Usage

```
// All operations below are performed sequentially.
ContinuousMedianHandler(): - // instantiate a ContinuousMedianHandler
insert(5): -
insert(10): -
getMedian(): 7.5
insert(100): -
getMedian(): 10
```

Hint:

The median of a set of numbers is often, by definition, one of the numbers in the set. Thus, you likely have to store all of the inserted numbers somewhere to be able to continuously compute their median. The median of a set of numbers is either the middle number of that set (if the set has an odd amount of numbers) or the average of the middle numbers (if the set has an even amount of numbers). This means that if you could somehow keep track of the middle number(s) of the set of inserted numbers, you could easily compute the median by finding the indices of the middle numbers and doing some simple calculations. Perhaps storing all of the numbers in a sorted array could

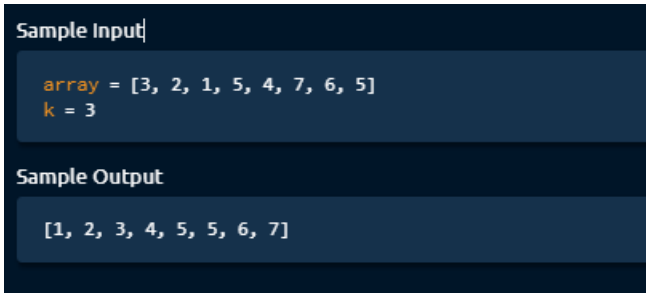
work, but what would be the runtime implication of inserting each new number into a sorted array? Realizing that you only need to keep track of the middle numbers in the set of inserted numbers to compute the median, try keeping track of two subsets of the numbers: a max-heap of the lower half of the numbers and a min-heap of the greater half of the numbers. Any time you insert a number, pick the heap to place it in by comparing it to the max / min values of the heaps. Then, re-balance the heaps in an effort to keep their sizes apart by at most one. Doing so will allow you to access the middle number(s) of the set of inserted numbers very easily, which will make calculating the median a trivial computation. Re-balancing the heaps can be accomplished by simply removing a value from the larger heap and inserting it in the smaller one. What are the runtime implications of all these operations?

Optimal Space & Time Complexity:

Insert: O(log(n)) time | O(n) space - where n is the number of inserted numbers

Difficulty Med : Sort K-Sorted Array

Write a function that takes in a non-negative integer k and a k-sorted array of integers and returns the sorted version of the array. Your function can either sort the array in place or create an entirely new array. A k-sorted array is a partially sorted array in which all elements are at most k positions away from their sorted position. For example, the array [3, 1, 2, 2] is k-sorted with k = 3, because each element in the array is at most 3 positions away from its sorted position.



Hint:

What does the k parameter tell you? How can you use it to come up with an algorithm that runs in O(nlog(k))? Since the input array is k-sorted, try repeatedly sorting k elements at a time and inserting the minimum element of

all those k elements into its final sorted position in the array. What auxiliary data structure would be helpful to quickly determine the minimum element of k elements? As you iterate through the array, use a min-heap to keep track of the most recent k elements. At each iteration, remove the minimum value from the heap, insert it into its final sorted position in the array, and add the current element in the array to the heap. Continue this process until the heap is empty.

Optimal Space & Time Complexity:

O(nlog(k)) time | O(k) space - where n is the number of elements in the array and k is how far away elements are from their sorted position

Difficulty Med : Laptop Rentals

You're given a list of time intervals during which students at a school need a laptop. These time intervals are represented by pairs of integers [start, end], where 0 <= start < end. However, start and end don't represent real times; therefore, they may be greater than 24. No two students can use a laptop at the same time, but immediately after a student is done using a laptop, another student can use that same laptop. For example, if one student rents a laptop during the time interval [0, 2], another student can rent the same laptop during any time interval starting with 2. Write a function that returns the minimum number of laptops that the school needs to rent such that all students will always have access to a laptop when they need one.

Sample Input

```
times =
[
  [0, 2],
  [1, 4],
  [4, 6],
  [0, 4],
  [7, 8],
  [9, 11],
  [3, 10],
]
```

Sample Output

```
3
```

Hint:

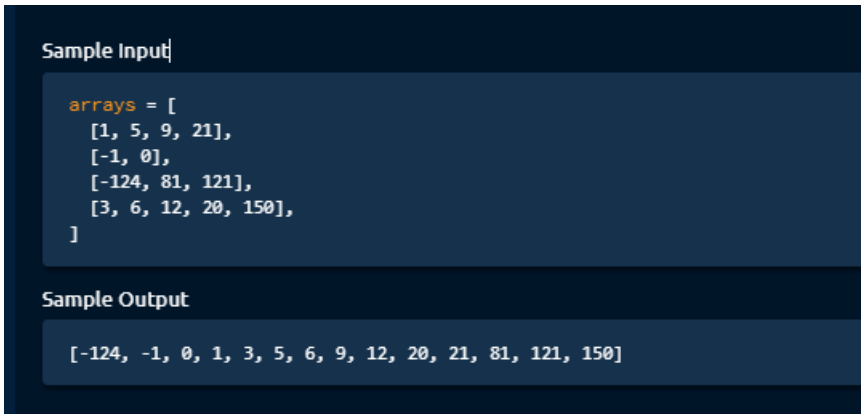
There are many different ways to solve this problem, but only a few of them run in the optimal time. Can you come up with an algorithm that solves this problem in O(nlog(n)) time? Suppose that you're given two time intervals: [s1, e1] and [s2, e2], where s1 < s2. If e1 <= s2, then the second time interval can use the same laptop as the first time interval. One method to solve this problem with an optimal time complexity is to use a Min Heap. If you loop through time intervals that have been sorted by their start times and keep track of the smallest end time of time intervals for laptops that have already been rented out, you can determine how many laptops are required. Use the Min Heap to efficiency determine if any previous rental time intervals have ended as you loop through all the time intervals. If a rental time interval is done and another one starts after it, no extra laptop is required. Another way to efficiently solve this problem is to realize that we don't need to know what start time corresponds with what end time. So long as we know all start times and all end times, we can determine the number of laptops required. Start by creating two arrays—one for start times and one for end times—and sort them both in ascending order. We can simply loop through the start times and end times at the same time and compare the current start time to the current end time. If the current start time is greater than the current end time, then that means a laptop that was previously used is no longer being used and can be given to the student renting a laptop at this starting time. Thus, we can increment both our start-time and end-time pointers and continue without needing an additional laptop. If the current start time is smaller than the current end time, then another rental has started before a previous rental has ended, and we thus require another laptop, so we increment the start pointer and a variable keeping track of the number of laptops required. See the Conceptual Overview section of this question's video explanation for a more in-depth explanation.

Optimal Space & Time Complexity:

O(nlog(n)) time | O(n) space - where n is the number of times

Difficulty Med : Merge Sorted Arrays

Write a function that takes in a non-empty list of non-empty sorted arrays of integers and returns a merged list of all of those arrays. The integers in the merged list should be in sorted order.



Hint:

If you were given just two sorted lists of numbers in real life, what steps would you take to merge them into a single sorted list? Apply the same process to k sorted lists. The first element in each array is the smallest element in the respective array; to find the first element to add to the final sorted list, pick the smallest integer out of all of the smallest elements. Once you've found the smallest integer, move one position forward in the array that it came from and continue applying this logic until you run out of elements.The approach described involves repeatedly finding the smallest of k elements, since there are k arrays. Doing so can be naively implemented using a simple loop through the k relevant elements, which results in an O(k)-time operation. Can you speed up this operation by using a specific data structure that lends itself to quickly finding the minimum value in a set of values. Follow the approach described, using a Min Heap to store the k smallest elements at any given point in your algorithm.

Optimal Space & Time Complexity:

O(nlog(k) + k) time | O(n + k) space - where where n is the total number of array elements and k is the number of arrays