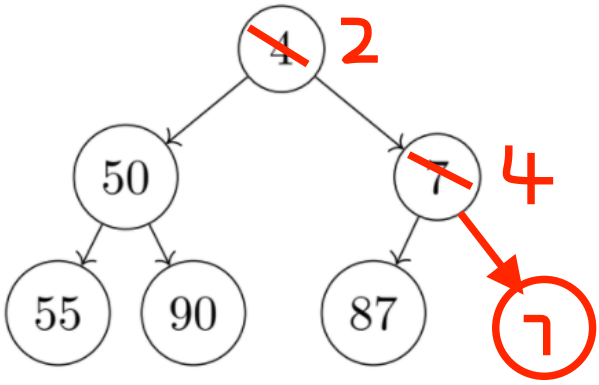
- This lab covers Heaps/ Priority Queues.
- It is assumed that you have reviewed chapter 9 of the textbook. You may want to refer to the text and your lecture notes during the lab as you solve the problems.
- When approaching the problems, think before you code. Doing so is good practice and can help you lay out possible solutions.
- Think of any possible test cases that can potentially cause your solution to fail!
- You must stay for the duration of the lab. If you finish early, you may help other students. If you don't finish by the end of the lab, we recommend you complete it on your own time. Ideally you should not spend more time than suggested for each problem.
- Your TAs are available to answer questions in lab, during office hours, and on Piazza.

Vitamins (25 minutes)

---

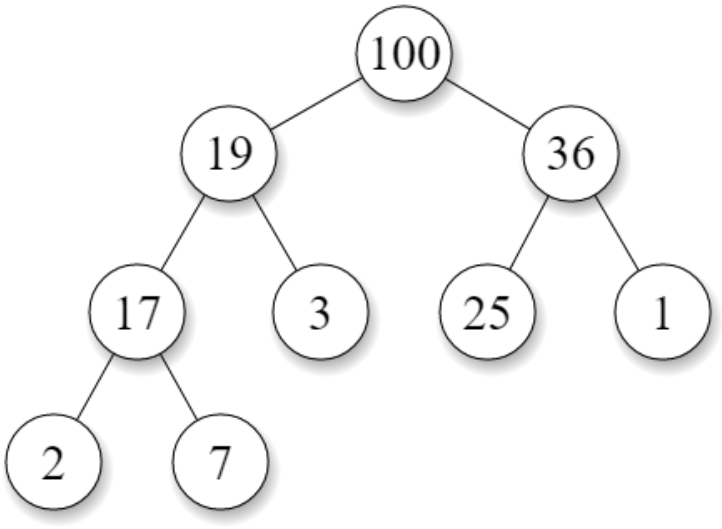
1. Given the following Min heap:



Complete the following operations, show how the final heap looks (both in the array representation and the binary tree representation) after each operation (10 minutes):

1. Insert 2
2. Delete min
3. Insert 6
4. Insert 8
5. Delete min
6. Insert 52
7. Delete min

2. Given the following Max heap:



Complete the following operations, show how the heap looks (both in the array representation and the tree representation) after each operation (5 minutes):

1. Insert 50
2. Delete max
3. Delete max

3. General Concepts (10 minutes)

   a. A student claims that, because in a min heap, each node in its left subtree and right subtree is greater than its parent, searching for an element in the min heap would cost log(n) just like it would for a binary search tree. Is this correct? If not, explain why with a counter example.

   b. A student claims that since a list is used to implement the min heap in class, if he just reverses the order of the list, it becomes a max heap. Is this correct? If not, explain why with a counter example.

2. Describe, without coding, how you would find the kth, where 0 <= k < n, smallest element in a list of n numbers. For instance, given [10, 3, 5, -4, 2, 6], the 0th smallest number would be -4, 2nd smallest number would be 3, the 4th smallest number would be 6, etc.

   How many ways can you solve the problem? What are the worst-case run times of your solutions? (5 minutes)

Coding

In this section, it is strongly recommended that you solve the problem on paper before writing code.

1. In class, we implemented the ArrayMinHeap. Create a new file ArrayMaxHeap.py and copy and paste the ArrayMinHeap code in the new file. Modify it so that it is now an ArrayMaxHeap. (5 minutes)

2. You are given a list of unsorted positive integers representing the length of ropes. Your objective is to consolidate all the ropes into one single long rope. You can only combine two separate ropes at any given time. In addition, the cost to combine two separate ropes is equivalent to the length of the resulting rope.

   ex) Given [10, 5, 2, 14, 3, 8], the cost of combining the first two ropes of length 10 and 5 is $15. My list of ropes would now be [15, 2, 14, 3, 8] (not necessarily in that order).

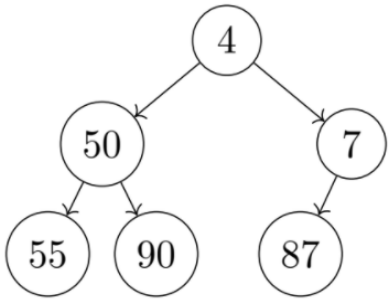
   In the end, the total length of the consolidated rope is equal to 42 (sum of the list).

   Write a function to find the minimum cost of creating the single rope. (25 minutes)

   ```
   def min_merge_cost(ropes):
       …
   ```

   In the given example, your function should return 99. Why 99? You'll have to figure that out yourself. For this question, do not modify the input list. You may want to use the ArrayMinHeap implemented in class.

3. You are given a list of integers. Write a **recursive** function to determine if the array represents a min heap. **You may define recursive helper functions. (25 minutes)**



ex) The array representation for this heap is [4, 50, 7, 55, 90, 87]. Calling is_min_heap on this list would return True.

However, calling is_min_heap on a list such as [10, 50, 7, 55, 90, 87] would return False because 10 is not less than its right child, 7.

```
def is_min_heap(lst):
    ...
```

<u>Hint:</u> In our implementation using the heap example, we will consider index 1, which contains 4, to be the root and min of the min heap.

It's left child would be at index 2 ← (parent index * 2).
It's right child would be at index 3 ← (parent index * 2) + 1.

Please read the following for context the next few questions:

Heapifying and Sorting **both require comparative operators to be implemented.** If you are working with single values, integers, heapifying and sorting will not be a problem. However in the next fews problems, you will be working with multiple numbers.

Python allows you to store multiple information in the form of tuples and provides a comparison support at the 0th, 1st, 2nd, … etc indices of the tuple.

ex) a tuple (3, 5, 2) < (1, 5, 4) would be False since at the 0th index, **3 < 1** is False.
ex) a tuple (3, 5, 2) < (3, 5, 4) would be True since 3 <= 3, 5 <= 5, and **2 < 4**.

The comparison is also built in in other **collections that retain the order in which the elements are added** such as strings, and lists (but not sets and dicts). You can test these yourselves. Note that < and > operators for sets are used to determine subsets and not comparisons.

ex) "ade" < "az" is True because 'a' <= 'a' and 'd' < 'z'. (d lexically appears before z).

For this reason, you can sort these collections like so:

```
lst = [(5, 'z', 2), (6, 'a', 0), (4, 'h', 4), (4, 'c', 1)]
lst.sort()
print(lst) → [(4, 'c', 1), (4, 'h', 4), (5, 'z', 2), (6, 'a',
0)]
```

Note that the more information you choose to store in the tuples, the messier your code becomes. Your code can become more difficult to read. In addition if you are only basing your comparisons based on one value, it would be better to define your own classes and overload the comparative operators. This helps to organize your code better and make it more readable. How you choose to solve these problems is up to you.

<u>Example for question 4:</u>

```
class Number:
        def __init__(self, val, count):
                self.val = val
                self.count = count
        def __lt__(self, other):        #less than
        def __le__(self, other):        #less than or equal to
        def __gt__(self, other):        #greater than
        def __ge__(self, other):        #greater than or equal to
        def __eq__(self, other):        #equal
```

```
def __ne__(self, other):        #not equal
```

Specifically for the ArrayMinHeap implemented in class, you may also use the priority value pair when calling insert. Note that by default, the value is None.

Looking at the insert method for the ArrayMinHeap:

```
    def insert(self, priority, value=None):
        new_item = ArrayMinHeap.Item(priority, value)
        self.data.append(new_item)
        self.fix_up(len(self.data) - 1)
```

For question 4, you could set the priority to the count and its value to be the number.

ex)

```
pq = ArrayMinHeap()
pq.insert(3, 5) #the value 5 occurs 3 times in the list
```

Note that not all priority queue implementations have a (priority, value) pairing, in which case you would want to use a tuple or a separate class with comparison operators overloaded.

How to find the kth smallest/largest in a MinHeap.

With a Min Heap, we can find the k smallest numbers of a heap of size n by simply calling delete min k times. But how do we find the k largest? We would have to leave k numbers remaining in the heap by removing n-k smallest numbers from the heap. That is, we should call delete min n-k times. Then we are left with k numbers, these remaining numbers are the k largest numbers.

Example:
MinHeap = [1, 2, 3, 4, 5], we want k = 3 largest out of n= 5 numbers.

Get rid of the smallest n - k = 5 - 3 = 2 numbers.

MinHeap = [3, 4, 5], now we are left with k =3 numbers.

[3, 4, 5] are the k largest numbers in the MinHeap.

It would be better to use a max heap to find the k largest but it is also important to understand how to do so with a min heap.

4. Recall from the previous Lab (12 - Hashing) the following function:

```
def most_frequent(lst):
```

For this function, you created an instance of a ChainingHashTableMap and set each number as a key and its value as the number of occurrences in the list.

Revise the function so that it now takes in an additional integer, k, as a parameter and returns a list of k most frequently occurring numbers. (30 minutes)

```
def k_most_frequent(lst, k):
```

In the given example, lst = [5,9,2,9,0,5,9,7], k_most_frequent(lst, 2) would return [9, 5] or [5, 9] as they are the 2 most frequent numbers. Order does not matter.

In the case of numbers with equal number of occurrences, 2, 0, 7, any of these numbers may be selected. That is, k_most_frequent(lst, 3) can return [9, 5, 2], [9, 5, 0], or [9, 5, 7].

For this question, do not use any sorting algorithms. Instead, you should use the ArrayMinHeap that was implemented in class to solve this problem.

You may choose to define a separate Number class or simply use a tuple to store information in the heap.

Hint: See how to find the kth smallest/largest in a MinHeap section from the previous page.

5. Recall the following function (you've written this for linked lists, and merge sort):

```
def merge_sorted_lsts(lst1, lst2):
```

The function takes in **two already sorted lists**, and **returns a new sorted** list containing elements from both lists.

ex)
lst1= [1, 5, 6, 9] and lst2= [2, 3, 4, 7, 8]
merge_sorted_lsts(lst1, lst2) would return [1, 2, 3, 4, 5, 6, 7, 8, 9]

Revise the function so that it will take a list of k sorted lists (a matrix) and return a single sorted list containing all the elements of the k sorted lists. Note that the numbers can repeat, and that the length of each list can vary. Assume no list is empty. (40 minutes)

```
def merge_k_sorted_lsts(matrix):
```

ex) index i
matrix = [[1, 5, 47], # 0th list
[-5, 2, 3, 10, 12], # 1st list
[45, 100, 120], # 2nd list
[-3, 0, 0, 1, 120, 1134]] # 3rd list
index j → 4 lists

merge_sorted_lsts(matrix) would return the following:
[-5, -3, 0, 0, 1, 1, 2, 3, 5, 10, 12, 45, 47, 100, 120, 120, 1134]

Consider that worst-case, all lists in the matrix have the same length, n, and there are k lists, you would have a total of n*k elements. If you place all n*k elements in a list and called merge-sort, you would have a run-time of n*k*log(n*k) with n*k extra space.

Instead, you should use the ArrayMinHeap as implemented in class to solve this problem. Optimize the runtime and space complexity.

<u>Hint:</u> Each list in the matrix is already sorted. You should extend the approach you did with 2 lists, to k lists. For 2 lists, we really need 2 indices to walk through the two sorted lists and figure out which one is smaller.

For k lists, how many comparisons do we really need to make at any given time? Consider this when you add elements into the heap. You may choose to define a separate number class or simply use a tuple to store information in the heap.