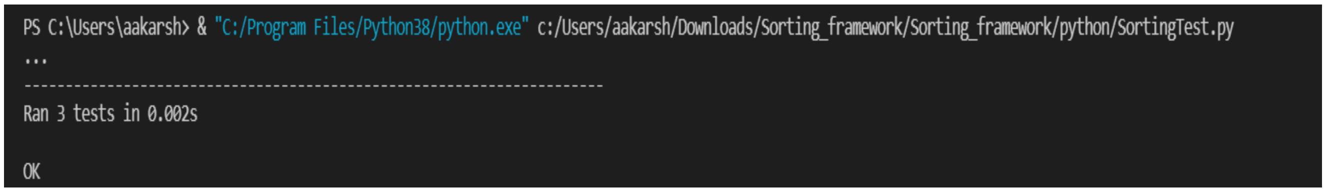
Unity Id: asatish2
Mail: asatish2@ncsu.edu
Aakarsh Satish

1. Purpose: Implementing algorithms. Implement insertion sort, merge sort, and heap sort, and count the number of comparisons they perform. Follow the description in our textbook in the following sections: Section 2.1, Page 19 (INSERTION-SORT) for insertion sort; Section 2.3, Pages 36 (MERGE) - 39 (MERGE-SORT) for merge sort; and Sections 6.2-6.4, Pages 165 (MAX-HEAPIFY) - 170 (HEAPSORT) for heapsort. Pay attention to ties and special case considerations. Please only count comparisons between the input values, not between index variables. i) In insertion sort, only count the number of comparisons between elements in the array A (the second compare on line 5, page 19), not the compares that check if the index variable i is larger than zero; ii) in merge sort, count the number of comparisons on line 13 page 36; iii) in heapsort, count the number of comparisons between the elements of array A (the second compares on line 3 and line 6 on page 165), not the compares between variables l, r, A.heapsize, and largest. (12 points, 4 points for each algorithm)


```
PS C:\Users\aakarsh> & "C:/Program Files/Python38/python.exe" c:/Users/aakarsh/Downloads/Sorting_framework/Sorting_framework/python/SortingTest.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.002s

OK
```

In sorting.py file

2) Practice solving recurrences. For a-c, use the Master Theorem to derive asymptotic bounds for T(n) or argue why the Master Theorem does not apply. You don't have to solve the recurrence if the MT does not apply. Please assume that small instances need constant time c if not explicitly stated. Justify your answers, i.e., give the values of a, b, n^log_b(a), for case 3 of the Master Theorem also show that the regularity condition is satisfied. (3 points each)

(a) T(n)=8T(n/2)+n4+n.

Here a = 8, b = 2 and f(n) is n^4.
We shall calculate watershed function n^log_b(a) = n^log_2(8) = n^3.
Now comparing it with f(n) we see that f(n) is greater which is n^4 > n^3.
So this belongs to case 3 of MT if the regularity condition holds true.
We get f(n) = Ω(n^log_b(a) + ε)

To prove regularity condition for f(n) i.e af(n) <= cf(n) for some c< 1and sufficiently large n.
Using the values of a and b we get 8*(n/2)^4 <= c *n^4.
1 > c >= 8/(2^4) = 0.5 and n>=0. So case 3 applies and we now have T(n) = Θ(n^4).

(b) T(n)=3T(n/3)+n/lg(n)
T(n) = aT(n/b) + f(n)
Comparing with this we get a = 3 and b = 3 and f(n) = n/lg(n). We have n^log_b(a) = n.
f(n) = n * lg^-1(n).
Comparing f(n) and n^log_b(a) to find out MT case does not apply as the value of f(n) has ln^-1(n). But according to master theorem case 2 we need value of k>=0
So this cannot be solved by using master theorem.

(c) T(n)=2T(n/4)+nlg(n)
T(n) = aT(n/b) + f(n)
Comparing with this we get a = 2 and b = 4 and f(n) = nlg(n). We have n^log_b(a) = n^0.5.
Now comparing it with f(n) we see that f(n) is greater which is nlg(n) > n^0.5.
So this belongs to case 3 of MT if the regularity condition holds true.
We get f(n) = Ω(n^log_b(a) + ε)
To prove regularity condition for f(n) i.e af(n) <= cf(n) for some c< 1and sufficiently large n.
Using the values of a and b we get 2*(n/4)*lg(n/4) = n/2 * (lg(n) - 2) <= c*nlg(n).
This inequality holds true for 1>c>=½ and n>0. So case 3 applies and we now have T(n) = Θ(nlg(n)).

(d) Assume T(n) = (n+1)T(n-1)/n+c(2n-1)/n for n>1 and T(1) = 0.
Show T(n) = c(n+1)∑ 2i−1 / i(i+1) i=2 by forward iteration

T(n) = (n+1) T(n-1) / n + c(2n-1) / n
T(1) = 0, given
T(2) = [3 * T(1) / 2] + c(2*2 - 1) / 2
        [3 * 0 /2] + c(2*2 - 1) / 2
        => 3*c[(2*2 - 1) / 2*3]
T(3) = [(4*c*3)/ (3*2)] + 5*(c) / 3
        [4 * c * (1/2 + 5/12)] =
        => (3+1) * c * [(2*2-1) / (2*3) + (2 * 3 - 1) / (3*4)]
.
.
.

.

T(n) = c * (n + 1) * [(2)(2) - 1 / (2)(3) + (2)(3) - 1 / (3)(4) + (2)(4) - 1 / (4)(5) + (2)(5) - 1 / (5)(6) + ………… + 2(n) - 1 / n* (n+1)]

T(n) = c(n+1) Σ 2*i*−1 / *i*(*i*+1) *i*=2

Thus the given equations are satisfied.

3. (9 points) Purpose: Practice algorithm design. This problem was an interview question! To avoid deductions, please follow Eric Demaine's instructions about how to "give/describe" an algorithm. Let X[1..n] and Y[1..n] be two arrays, each containing n numbers already in sorted order. Give an O(lgn)-time algorithm to find the median of all 2n elements in arrays X and Y. You may assume that all 2n numbers are distinct, and that n is a power of 2.

Let there be two given arrays X and Y with n unique elements each. First we shall find out the median of both of these arrays. Let us name them median_x and median_y. As the algorithm needs to be run in O(log n) times, we need to use divide and conquer method, where we split the array size in half and compute one of them in every recursive call. After computing the medians of both of these arrays, we shall compare it with each other. In our problem statement, if both of these medians are the same, then we shall return that answer as that will be the answer. If not, then we will find ourselves in 2 cases:
Case 1: If the median_y value is higher than median_x value, this means our answer is in the second half of the input array of X and the first half of input array Y. So hence we can delete or discard the first half array of X and second half array of Y. This shall reduce the input array size of X and Y by 2.
Case 2: If the above condition doesent hold good then it means that median_x value is higher than median_y value. In this case the reverse of case 1 happens. Here we discard the lower half of Y and the upper half of X. Even in this scenario we shall reduce the size of the array of X and Y by 2.
This shall be done recursively calling the function with the updated arrays. The base case shall be when both the medians are same or when the array element size is 1 in both the arrays. If this is the case then we take the average of these 2 elements and return the overall median.

The algorithm:

1) Compare the medians of X and Y,which are median_x and median_y respectively. median_x = X[n/2] where [n/2] represents the middle index of X.

median_y = Y[n/2] where [n/2] represents the middle index of Y.

If n == 1, then take the average of X[0] and Y[0] i.e ((X[0] +Y[0]) /2))

2) If median_x is less than median_y, discard the elements in X from the beginning to the middle index [n/2], and discard the elements in Y from the middle index [n/2] to the end. Essentially, reduce the problem to finding the median of the remaining halves of X and Y.

3) If median_x is greater than median_y, do the opposite: discard the elements in X from the middle index [n/2] to the end and discard the elements in Y from the beginning to the middle index [n/2].

   Repeat steps 1 to 3 recursively until we find the median.

One worked example of this code:

1) Let us take array X to have 1,3,5,7 and array Y to have 2,4,6,8 where n = 4 and 2n is 8 (power of 2).
2) Here median_x computes to be X[4 / 2] = X[2] which equals 5 and we compute median_y computes to be Y[4 / 2] = Y[2] = 6.
3) The n value is not equal to 1 so it goes to the next if condition.
4) Next the condition median_Y is greater than median_X is satisfied, hence the recursive call to the same function with the updated array X and Y are passed as parameters. Array X will contain the second half of the elements and the first half discarded whereas the Array Y will have the 1st half of the elements and second half discarded.
   Array X will now have [5,7] and Array Y will now have [2,4].
5) The n value will have 2. Now to the updated arrays we find the median again. median_X will now have X[2/1] = X[1] which equals 7 and median_Y will now have Y[2/1] = Y[1] which equals 4.
6) Now checking the base condition it gives us false. The condition median_X is greater than median_Y is satisfied in this case.
7) Now we discard the first half of Y and the second half of X and update the array values. The updated Array values of X and Y will be [5] and [4].
8) Now n value is equal to 1. As it is equal to 1 we just find the average of the two values in the array. In our case X[0] = 5 and Y[0] = 4. So average of these two will get us the median which comes upto (5+4) / 2 = 4.5.
9) Now return the obtained overall median.

Proof of correctness:
Loop Invariant: The output of the recursion call is the overall median of array x and array y.

Initialisation: Prior to the recursive call, the algorithm ensures that the input arrays X and Y are both sorted in ascending order, having a length of n (where n is a power of 2), contain distinct elements, and that the median of X is less than or equal to the median of Y.

Maintenance: During the maintenance phase, as the function proceeds with its recursive call, it guarantees that these properties are preserved. It correctly calculates the median values for X and Y within the current recursive call and properly eliminates the left half of either X or Y and the right half of the other array, ensuring that the overall median remains within the remaining subarrays.

Termination: Upon termination, when the base case is reached (i.e., both X and Y have only one element), the algorithm returns the correct median of the combined arrays X and Y

Analysis of the space complexity and Time Complexity.

1) Space Complexity: The algorithm computes the overall median of the two given input sorted arrays by using just few variables median_X and median_Y as discussed. So the storage is not dependent on the the input array which we give. Hence the space complexity is constant.
2) Time Complexity: This algorithm uses the concept of Divide and Conquer method. The algorithm computes the overall median by basically dividing the array by 2 by discarding the half of the elements in array X and array Y based on the condition it satisfies. This takes O(log n) time. The conquer in this algorithm is just computing the median value so it takes constant time O(1). So hence combining both these values we see that the time complexity is O(1) + O(log n). So time complexity is O(log n) for this algorithm.

4) Purpose: Practice algorithm design and the use of data structures. This problem was an interview question! To avoid deductions, please follow Eric Demaine's instructions about how to "give/describe" an algorithm. Consider a situation where your data is almost sorted—for example, you are receiving time-stamped stock quotes, and earlier quotes may arrive after later quotes because of differences in server loads and network traffic routes. Focus only on the time stamps. Assume that each time stamp is an integer, all time stamps are different, and for any two time stamps, the earlier time stamp corresponds to a smaller integer than the later time stamp. The time stamps arrive in a stream that is too large to be kept in memory. The time stamps in the stream are not in their correct order, but you know that every ime stamp (integer) in the stream is at most a hundred positions away from its correctly

sorted position. Design an algorithm that outputs the time stamps in the correct order and uses only a constant amount of storage, i.e., the memory used should be independent of the number of time stamps processed. Solve the problem using a heap.

We need to store the store the incoming numbers in memory until the smallest number has arrived in it. We know that the number is not off by more than 100 elements. With this information we can be sure that the smallest number should arrive within the first 101 elements. Hence we shall initialise a min heap with 101 time stamps. Now we shall accept the incoming traffic or elements. We shall take the first 101 elements in the min heap until the min heap is complete. Now we can extract the minimum from the min heap. The given element is the smallest among all the heap elements, so we replace the root of the heap with a new element. Now we can heapify the min heap and continue and repeat the process for the total size of the heap which is 101 in our case. As we do it 101 times, the space does not matter on the incoming number of elements.

Algorithm:

1) First we shall initialise the input stream size as 101.
2) Take the 101 elements from the array as an input.
3) Now we shall build a min heap from the array.
4) Take out the root of the heap as it contains the minimum element in the heap.
5) Take the next element from the array and heapify.
6) Heap sort the remaining elements from the array and output.

```
Algorithm : AlgorithmTimeStampHeap
1 Initialize array X with size 101
2 X ← Read the first 101 elements from input array
3 Buildheap(X)
4 while S ≠ 0 do
5       minimumElement ← Minimum(A)
6       Extract minimumElement
7       X[0] ← Read the next element
8       Heapify(X[0])
9 end
10 Output X
```

Output Example:

Considering the element is atmost 2 positions away from correct positions.
Let the input array be : [5,7,4,6,9,10]

1) Initialisation Step: Initialise array with first three elements : X = [5,7,4]
2) Build the min heap: [4,7,5]
3) Output the root of the min heap: 4
4) Replace the root of the heap: [5,7,6]
5) Output the root of the min heap: 5
6) Replace the root with a new element and perform heapify: [6,7,9]
7) Output the root of the heap: 6
8) Replace the root of the heap with aa new element and heapify: [7,10,9]

Output : [7,9,10]

a proof (or other indication) of the correctness of the algorithm;

Proof of correctness: Using Loop invariant: Output of the loop is always the smallest element in the input and is larger than the previous elements.

Initialisation: Before the first loop, there are no input elements to output. Since at the start of the loop we have no elements and as we have will have the smallest element in the root of the heap, we shall have no elements to output too. Here we can see that no element is more than 100 positions away from the appropriate position. So the smallest element must be in the first 101 elements.

Maintenance: Each time an element is taken from X, it is made sure it is the smallest element out of 101. Let us assume that an element B<C will occur in future. So if B is less than the 101 other components then it means that it is positioned incorrectly by more than 100 elements. This is a contradiction that C might be the smallest element. So hence by maintaining loop invariant, the timestamps are always produced in the right sequence.

Termination: At the end, a heap-sort is used to order the elements in increasing order. As a result, all of the elements are in the output in the proper sequence.

The heap's constant size during algorithm execution results in a space complexity of Θ(1). We are not taking any extra space or memory to perform our computation and it is independent of the number of input elements.
The buildmin heap takes a time of Θ(1). The while loop in this algorithm takes (n-100) or a maximum of 100 iterations. So in our case the time complexity of that will be Θ(n). X's root can be found by using Θ(1) as well which is a constant time. Heapify X takes Θ(1) for each element read from the input stream. In the worst-case scenario, it takes Θ(n) to sort the first n elements.