# Problem Set 2—Sorting Algorithms

Important notes:

- Watch this video recorded by Joram Erbarth (M23) with advice on how to prepare for CS110 assignments. Most of the suggestions will also apply to other CS courses, so make sure to bookmark this video for future reference. You will also notice that the video refers to submitting primary and secondary resources, but for this specific assignment, since you will answer questions directly in the notebook, you won't need to worry about uploading your work.
- Make sure to include your work whenever you see the labels `###YOUR DOCSTRING HERE` or `###YOUR CODE HERE` (there are several code cells per question you can use throughout the notebook, but you need not use them all).
- Please refer to the CS110 course guide on how to submit your assignment materials.
- If you have any questions, do not hesitate to reach out to the TAs in the Slack channel #cs110-algo, or come to the instructors' OHs.

Question 1 of 8

🏁 Setting up:

Start by stating your name and identifying your collaborators. Please comment on the nature of the collaboration (for example, if you briefly discussed the strategy to solve problem 1, say so, and explicitly point out what you discussed). Example:

Name: Ahmed Souza

Collaborators: Lily Shakespeare, Anitha Holmes

Details: I discussed the iterative strategy of problem 1 with Lily, and asked Anitha to help me design an experiment for problem 2.

Normal

## Problem 1

Imagine that you land your dream software engineering job, and among the first things you encounter is some previously written, poorly commented code.

Asking others how it works proves fruitless, as the original developer left. You are left with no choice but to understand the code's inner mechanisms and document it properly for both yourself and others. The previous developer also left behind several tests that suggest that the code is working correctly, but they seem far from comprehensive. Your tasks are listed below. Here is the code:

Code Cell 1 of 42 - Read Only

In [1]
```python
## READ ONLY
def my_sort(lst):
    n = len(lst)
    piles = []
    piles.append([lst.pop(0)])

    while lst:
        item = lst.pop(0)
        placed = False
        for pile in piles:
            if item > pile[-1]:
                pile.append(item)
                placed = True
                break
        if not placed:
            piles.append([item])

    while len(lst) < n:
```

```
        tops = [pile[-1] for pile in piles]
        idx_smallest = tops.index(min(tops))
        lst.append(piles[idx_smallest].pop(0))
        piles = list(filter(None, piles))
    return lst
```

Run Code

Code Cell 2 of 42 - Read Only

In [2]

```
## READ ONLY
assert my_sort([8, 5, 7]) == [5, 7, 8]
assert my_sort([10, 9, 8, 7, 6, 5, 4, 3, 2, 1]) == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Run Code

Question 2 of 8

**A.** Explain, in your own words, what the code is doing (it's sorting an array, yes, but how?). You may find it helpful to sketch step-by-step diagrams, play around with the code in other cells, and print test cases or partially sorted arrays. You should produce an approximately 150-word write-up of how the code is processing the input array.

Normal

**The explanation for the code above:**

We can imagine that there is an unsorted row of numbered cards and we want to sort it. We start by taking the first card (going from left to right) of the row and placing it separately alone. Then we take the next card from the row (it is the first card in the row from the left since we took the previous card out). Now, with this card in hand, we compare it to the card that lies separately. If the card in our hand is bigger than the card on the table, we place the card in hand on top of the card on the table, creating a pile of 2 cards with a larger card on top. If, however, the card in our hand is smaller than the card on the table, we place the card in our hand separately to the right of the card that was already on the table, thus forming 2 different piles consisting of one card each. We then take the next card from the row (the first card on the left). We compare this card to the top card of each pile going from left to right. If the card in hand is bigger than the top card of a pile, place the card in hand on top of the pile and take the next card from the row. If not, keep checking the top card of each pile until the card in hand is bigger than the top card, then place it on top. If our card is smaller than any of the top cards, place our card in a separate pile on its own to the right of the other piles and take the next card from the row. We repeat this process until there are no more cards left in a row. After this operation, we will end up with a pile or multiple piles of cards, each of which will be sorted from smallest to highest card internally.

Now, we look at the top card in each pile and find which one is the smallest. After we find the smallest top card, we take the bottom card from that pile and place it in a new row at the first position. We then repeat the process by finding the smallest top card across piles. Once we find the smallest top card, we take the bottom card from that pile and place it in the second position in a new row (to the right of the first card). We do so until there are no more piles left and we get a new sorted row of cards

Question 3 of 8

Feel free to add a flowchart if you think it will provide additional insights which are otherwise difficult to grasp from the code alone.

Drop or upload a file here

**B.** Add both a proper docstring and in-line comments to the code (there is an editable copy below). Anyone from your section should be able to understand the code from your documentation. Remember, however, that brevity is also a desirable feature.

Python 3 (384MB RAM) | Edit Run All Cells Kernel Stopped |

Code Cell 3 of 42

In [3]

```
def my_sort(lst):
    '''
    The function sorts an array of numbers by separating it into internally sorted subarrays and merging the obtained subarrays into a final sorted array
    -------
    Inputs:
    lst: list
        An unsorted array of numbers (integers)
    -------
    Outputs:
    lst: list
        Sorted input array of numbers (integers) from smallest to biggest
    '''

    n = len(lst) #store input array's length
    piles = []
    piles.append([lst.pop(0)])  #place the first number into a separate pile


    #run until there are no more numbers in the initial array left
    while lst:
        item = lst.pop(0) #store the first number from the input array and delete it from the array
        placed = False
        #go through each pile
        for pile in piles:
            #if the stored number is bigger than the last number in a pile, place the stored number in that pile to be the last number
            if item > pile[-1]:
                pile.append(item)
                placed = True #say that we placed a number in one of the exsisting piles
                break #stop the search and move to the next number
        if not placed: #if we did not place a number in onw of the piles, place it in a new separate pile (last pile)
            piles.append([item])

   #run until the list contains all numbers
    while len(lst) < n:
        tops = [pile[-1] for pile in piles] #take the biggest number (the last position in a pile array) from each pile
        idx_smallest = tops.index(min(tops)) #find the index of the smallest number among the biggest numbers
        lst.append(piles[idx_smallest].pop(0))#use the index to take the smallest number from the pile the smallest top number came from
        piles = list(filter(None, piles)) #place that number is a new sorted array
    return lst
```

Run Code

Question 4 of 8

**C.** Why are the tests that you are presented with insufficient?

Find at least three reasonable test cases that you think the code should pass, but it doesn't. What is wrong with the code that leads to this error?

Fix the code to pass your new tests.

**Hint to guide your work:** Play with the code for a small example, e.g., three numbers in different permutations. What's the idea behind the algorithm, and where is it implemented incorrectly? If you still struggle, come to your instructor's OH.

Normal

Python 3 (384MB RAM) | Edit Run All Cells Kernel Stopped |

The test cases given are insufficient because they do not encompass all possible input categories, which means even though the algorithm may pass the given test cases, it might not work for all types of inputs and we would not know about it. For instance, we did not test inputs with repeating numbers, different orders of numbers, or empty inputs.

For example, for some of the arrays, such as [6,8,7], [6,9,7,8], [6,9,7,9], and [ ], the algorithm above would not work.

**[6,8,7]**
Let's follow the algorithm for this input. We start by placing the first element in a separate pile, making piles=[[6]]. Then, we take the next element from the array, 8, and check if it is bigger than the last element in each pile. Since we have only one pile with 1 element, we compare 8 to 6. Since 8 is bigger than 6, we add 8 to the pile with 6 (to the right of it). It makes piles=[[6,8]]. Since we placed it in a pile, we stop comparing it to other piles. We repeat the while loop with the last remaining element in the list - 7. We start comparing 7 to 8. Since it is smaller than 8, we place it in a separate pile, making piles=[[6,8], [7]]. Since there are no more elements in the list left, we move on to merging the piles. We take the last numbers from each pile, making tops=[8,7]. We find the smallest number in tops list, which is 7, and record its index - 1 (in python notation). Finally, we take the first number from the pile where 7 came from and place it in the new sorted list. It means we take 7 out of the pile, leaving piles=[[6,8], []], and place it in lst, making lst=[7]. We then delete the empty pile, leaving piles=[[6,8]]. We repeat the process. We take the smallest number among the top cards from the only pile we have, which is 8, record its index in the tops list, which is 0 since we only have 8 in the tops list, and then we take the first number in the pile where 8 came from, which is 6. We place 6 in the sorted array to the last position, making lst=[7,6]. We perform the same operation with the remaining number 8, resulting in adding it to the sorted array: lst=[7,6,8].

As we can see, the array is not sorted properly. The reason this algorithm does not work sometimes is that we choose the **smallest number among the top numbers** from each pile, and base our pile choice to take the first number from its index (lines 35-37 in the code cell above). This leads to the selection of not the smallest number overall, but the smallest number in a chosen pile, which might not contain the overall smallest number as we saw in the example above. To fix this problem, we need to select the smallest number among the **bottom (smallest) numbers from each pile** and base our pile choice to take the first number from its index (lines 36-38 in the code cell below)

To solve the problem of an empty array, we simply need to terminate the algorithm in the beginning by checking if the length of the input array is 0. If it is, we return an empty list (lines 15-18 in the code cell below).

Below is the implementation of the same code but with a few corrections as described above.

Code Cell 4 of 42

In [4]
```
1 def my_sort(lst):
2     '''
3     The function sorts an array of numbers by separating it into internally sorted subarrays and merging the obtained subarrays into a final sorted array
4     -------
5     Inputs:
6     lst: list
7         An unsorted array of numbers (integers)
8     -------
9     Outputs:
10     lst: list
11         Sorted input array of numbers (integers) from smallest to biggest
12     '''
13     n = len(lst) #store input array's length
14     piles = []
15     if n>0:
16         piles.append([lst.pop(0)])  #place the first number into a separate pile
17     else:
18         return []
19 
20     #run until there are no more numbers in the initial array left
21     while lst:
22         item = lst.pop(0) #store the first number from the input array and delete it from the array
23         placed = False
24         #go through each pile
25         for pile in piles:
26             #if the stored number is bigger than the last number in a pile, place the stored number in that pile to be the last number
```

```
                pile.append(item)
                placed = True #say that we placed a number in one of the exsisting piles
                break #stop the search and move to the next number
        if not placed: #if we did not place a number in onw of the piles, place it in a new separate pile (last pile)
            piles.append([item])

   #run until the list contains all numbers
    while len(lst) < n:
        bottoms = [pile[0] for pile in piles] #take the smallest number (the first position in a pile array) from each pile
        idx_smallest = bottoms.index(min(bottoms)) #find the index of the smallest number among the smallest numbers
        lst.append(piles[idx_smallest].pop(0))#use the index to take the smallest number from the pile the smallest bottom number came from
        piles = list(filter(None, piles)) #place that number is a new sorted array
    return lst
```

Run Code

Code Cell 5 of 42

In [5]

```
#test cases to check for repeats, empty arrays, situations when there would be multiple piles with smallest top number but not smallest bottom number
assert my_sort([6,9,7,8]) == [6,7,8,9]
assert my_sort([6,8,7]) == [6,7,8]
assert my_sort([6,9,7,9]) == [6,7,9,9]
assert my_sort([6,9,9,7]) == [6,7,9,9]
assert my_sort([]) == []
```

Run Code

Code Cell 6 of 42

In [5]

Run Code

Code Cell 7 of 42

In [5]

Run Code

Code Cell 8 of 42

In [5]

Run Code

Please run the following code cell to check if your code passes some corner cases.

Code Cell 9 of 42 - Hidden Code

Run Code

Out [6]
```
Testing your code...
▶ All tests have completed successfully! Excellent work!
```

Code Cell 10 of 42

In [7]
```
### PLEASE DO NOT USE THIS CELL, IT WILL BE USED FOR FURTHER TESTING
```

Run Code

Code Cell 11 of 42

In [8]
```
### YOUR CODE HERE
```

Run Code

Code Cell 12 of 42

In [9]
```
### YOUR CODE HERE
```

Run Code

Code Cell 13 of 42

In [10]
```
### YOUR CODE HERE
```

Run Code

Code Cell 14 of 42

In [11]
```
### YOUR CODE HERE
```

Run Code

Python 3 (384MB RAM) | Edit Run All Cells Kernel Stopped |

## Question 2

> Have you ever read statements on Wikipedia and taken them for granted? It happens to all of us, but now you will have the chance to use the tools we have learned in class to corroborate or disprove statements you read in claimed reputable sources. Consider the following statement:"... insertion sort is one of the fastest algorithms for sorting very small arrays, even faster than quicksort; indeed, good quicksort implementations use insertion sort for arrays smaller than a certain threshold, also when arising as subproblems; the exact threshold must be determined experimentally and depends on the machine, but is commonly around ten."

From Wikipedia contributors. (2022, April 1). Insertion sort. Wikipedia. https://en.wikipedia.org/wiki/Insertion_sort

For the purposes of this problem, you do not need to know what is quicksort, it suffices to know that it competes with merge sort, so you can consider a different version of the statement above where you replace all the "quicksort" references with "merge sort." We would like to investigate whether we can design a better algorithm than merge sort, by analysing a variety of inputs distributions and sizes. To do this, address the following questions below.

**A.** Write a Python implementation that runs merge sort until the array gets fewer than 10 elements ($k = 10$) when it switches to applying insertion sort. Use the skeleton code provided below, add appropriate docstrings as comments where needed, and provide at least three test cases to demonstrate that your code is correct.

Code Cell 15 of 42

In [12]
```python
def merge(arr, start, mid, end):
    '''
    The function sorts a given subarray by splitting it in half,comparing the first numbers of the first and second halves,
    and putting the smaller number in the original array at the corresponding position
    -------
    Inputs:
    arr: list
        an array that needs to be sorted
    start: int
        the first index of the array
    mid: int
        the middle index of the array
    end: int
        the last index of the array
    ------
    Outputs:
    arr: list
        a sorted array
    '''
    #split the array in half
    left_array=arr[start:mid+1]
    right_array=arr[mid+1:end+1]
    #initiate indexes to iterate over subarrays
    index_left=0
    index_right=0
    #ensure we do not run out of range
    left_array.append(float('inf'))
    right_array.append(float('inf'))

    for i in range(start,end+1):
        #compare corrsponding numbers of both array to put the smaller in the correct position
        if left_array[index_left] <= right_array[index_right]:
            arr[i]=left_array[index_left]
            index_left+=1
        else:
            arr[i]=right_array[index_right]
            index_right+=1
    return arr

def merge_sort_until_k(arr, start, end, k = 10):
    '''
```

Python 3 (384MB RAM) | Edit Run All Cells Kernel Stopped |

```
into sorted array
    -------
    Inputs:
    arr: list
        an array that needs to be sorted
    start: int
        the first index of the array
    end: int
        the last index of the array
    k: int
        threshold subarray length value for switching from merge sort to insertion sort
    ------
    Outputs:
    arr: list
        a sorted array
    '''

    #switch to insertion when reach subarray of length k
    if end-start<k:
            arr=arr[:start] + insertion_sort(arr[start:end+1]) + arr[end+1:]
    else:
        #define the middle index of the array
        mid =(start+end)//2
        #keep dividing the array into left parts
        arr=merge_sort_until_k(arr,start,mid, k)
        #keep dividing the array into right parts
        arr=merge_sort_until_k(arr,mid+1,end, k)
        #if the subarray is bigger than k elements, perform merge sort, otherwise - insertion sort
        merge(arr,start,mid,end) #merge the subarrays into bigger arrays until we get the original array sorted
    return arr


def insertion_sort(arr):
    '''
    The function implements insertion sort algorithm : compares every consequetive number in the array to the number on the left until find the right position to place the current number
    -------
    Inpput:
    arr: list
        an array of numbers - array
    -------
    Output:
    arr: list
        sorted array
    '''
    for j in range(1, len(arr)): #iterating through all the indexes in the array starting with the second element (index=1)
        key = arr[j] #assigning the second number in the array to a key variable
        i = j-1 #defining another index, smaller than the previous by one

        while i >= 0 and arr[i] > key:
            arr[i+1] = arr[i] #if the element at index i is bigger than the key, put it at the position of the key
            i -= 1 #update index i by decreasing by one

        arr[i+1] = key #position the key element after the elemnt that is smaller than key
    return arr
```

Run Code

Code Cell 16 of 42

Python 3 (384MB RAM) | Edit Run All Cells Kernel Stopped |

In [12]

```

```

Run Code

Code Cell 17 of 42

In [13]

```
arr=[8,7,6,5,4,3,2,1]
merge_sort_until_k(arr, 0, len(arr)-1, k = 4)
```

Run Code

Out [13]

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

Code Cell 18 of 42

In [13]

```

```

Run Code

Code Cell 19 of 42

In [14]

```
arr = [3,1,4,5]
assert merge_sort_until_k(arr, 0, len(arr) - 1, k = 5) == [1, 3, 4, 5]

arr_2 = [3, 1, 4, 5, 7, 8, 12, 75, 44, 7, 0, 63, 11, 33, 28, 810, 8]
assert merge_sort_until_k(arr_2, 0, len(arr_2) - 1,k=5) == sorted(arr_2)



```

Run Code

Code Cell 20 of 42

In [15]

```
# add more test cases here
arr_3 = []
assert merge_sort_until_k(arr_3, 0, len(arr_3) - 1, k = 5) == []

arr_4 = [4,3,3,6,1,2,1,7,4]
assert merge_sort_until_k(arr_4, 0, len(arr_4) - 1) == sorted(arr_4)

arr_5= [3,2,4,2,3]
assert merge_sort_until_k(arr_5, 0, len(arr_5) - 1, k = 6) == [2,2,3,3,4]

arr_6= [4,3,2,1]
assert merge_sort_until_k(arr_6, 0, len(arr_6) - 1,k=3) == [1,2,3,4]
```

Run Code

Code Cell 21 of 42

In [16]
```
### YOUR CODE HERE
```

Run Code

Code Cell 22 of 42

In [17]
```
### YOUR CODE HERE
```

Run Code

Code Cell 23 of 42

In [18]
```
### YOUR CODE HERE
```

Run Code

**B.** Write another Python implementation that runs insertion sort until the array gets fewer than 10 elements ($k = 10$), and then the remaining array is sorted using merge sort.

Remember to add at least 3 test cases to demonstrate that your function is correctly implemented in Python.

Code Cell 24 of 42

In [19]
```
def insertion_sort_until_k(arr, start,end,k = 10):
    '''
    The function implements insertion sort algorithm until k elements left. Then it swtiches to merge sort. It also counts steps it takes to turn the algorithm
    -------
    Inpput:
    arr: list
        an array of numbers - array
    start: int
        the first index of the array
    end: int
        the last index of the array
    k: int
        the threshold length of the subarray to switch algorithms
    -------
    Output:
    arr: list
        sorted array
    '''
    for j in range(1, len(arr)-k+1): #iterating through all the indexes in the array starting with the second element (index=1)
        key = arr[j] #assigning the second number in the array to a key variable
        i = j-1 #defining another index, smaller than the previous by one

```

```
            arr[i+1] = arr[i] #if the element at index i is bigger than the key, put it at the position of the key
            i -= 1 #update index i by decreasing by one

        arr[i+1] = key # position the key element after the elemnt that is smaller than key

    #perform merge_sort on the last k elements and update the array
    arr=arr[:len(arr)-k+1]+merge_sort(arr[len(arr)-k+1:],0,len(arr[len(arr)-k+1:])-1)

    #merge 2 subarrays sorted using insertion sort and merge sort
    arr=merge(arr,0,len(arr)-k,len(arr)-1)
    return arr


def merge_sort(arr,start,end):
    '''
    The function recursively devides an array into smallest subarrays until they reach length 1 and then merges them back into sorted array.
    -------
    Inputs:
    arr: list
        an array that needs to be sorted
    start: int
        the first index of the array
    end: int
        the last index of the array
    ------
    Outputs:
    arr: list
        a sorted array
    '''
    #base case to reach smallest subarrays of length 1
    if start < end:
        #define the middle index of the array
        mid = (start+end)//2
        #keep deviding the array into left parts
        merge_sort(arr,start,mid)
        #keep deviding the array into right parts
        merge_sort(arr,mid+1,end)
        merge(arr,start,mid,end) #merge the subarrays into bigger arrays until we get the original array sorted
    return arr
```

Run Code

Code Cell 25 of 42

In [20]
```
### YOUR CODE HERE
```

Run Code

Code Cell 26 of 42

In [21]
```
### YOUR CODE HERE
```

Run Code

Code Cell 27 of 42

In [22]:
```
### YOUR CODE HERE

```

Run Code

Code Cell 28 of 42

In [23]:
```
arr = [3, 1, 4, 5]
assert insertion_sort_until_k(arr,0,len(arr)-1) == [1, 3, 4, 5]

arr_2 = [3, 1, 4, 5, 7, 8, 12, 75, 44, 7, 0, 63, 11, 33, 28, 810, 8]
assert insertion_sort_until_k(arr_2, 0,len(arr)-1,k = 5) == sorted(arr_2)
```

Run Code

Code Cell 29 of 42

In [24]:
```
# add more test cases here
# add more test cases here
arr_3 = []
assert insertion_sort_until_k(arr_3,0,len(arr)-1, k = 5) == []

arr_4 = [4,3,3,6,1,2,1,7,4]
assert insertion_sort_until_k(arr_4,0,len(arr)-1) == sorted(arr_4)

arr_5= [3,2,4,2,3]
assert insertion_sort_until_k(arr_5,0,len(arr)-1,k = 6) == [2,2,3,3,4]

arr_6= [4,3,2,1]
assert insertion_sort_until_k(arr_6, 0,len(arr)-1,k=3) == [1,2,3,4]
```

Run Code

Code Cell 30 of 42

In [25]:
```
### YOUR CODE HERE
```

Run Code

Code Cell 31 of 42

In [26]:
```
### YOUR CODE HERE
```

Python 3 (384MB RAM) | Edit Run All Cells Kernel Stopped |

Run Code

Code Cell 32 of 42

In [27]
```
### YOUR CODE HERE
```

Run Code

Question 5 of 8

Why are your test cases appropriate or possibly sufficient?

Normal

My test cases are appropriate and possibly sufficient because they encompass multiple types of inputs and edge cases: an empty list, a list in reverse order, a list with duplicates, and lists of different combinations of k and length of the list. Since these test cases cover a wider range of potential inputs, when the algorithm passes them we can be more confident that it is correct.

**C.** Run the function **merge_sort_until_k** in such a way so as to run merge sort until the array is of the length 10% of the original array, `arr` (i,e., $k = 0.1n$, where $n$ is the length of the array); below this size, your algorithm should run insertion sort for the remaining subarrays. Do the same by running the other function, `insertion_sort_until_k`.

Code Cell 33 of 42

In [28]
```
import random
arr = [random.randrange(-1000, 1000) for _ in range(100)]
### YOUR CODE HERE
merge_sort_until_k(arr, 0, len(arr)-1, k = 0.1*len(arr))

```

Run Code

Out [28]
```
[-982,
 -973,
 -945,
 -910,
 -872,
 -853,
 -836,
 -783,
 -779,
 -749,
 -690,
 -661,
 -635,
 -625,
 -592,
 -565,
 -527,
 -522,
 -515,
 -473,
 -444,
 -441,
 -394,
 -359,
 -339,
 -301,
 -293,
 -260,
```

```
 -239,
 -234,
 -229,
 -193,
 -173,
 -147,
 -131,
 -113,
 -72,
 -59,
 -55,
 -46,
 -32,
 -16,
 -8,
 -4,
 31,
 55,
 93,
 116,
 161,
 181,
 211,
 253,
 255,
 256,
 259,
 289,
 305,
 312,
 323,
 326,
 335,
 335,
 349,
 363,
 366,
 368,
 401,
 409,
 417,
 443,
 488,
 516,
 538,
 622,
 629,
 634,
 637,
 659,
 669,
 701,
 713,
 721,
 821,
 831,
 836,
 839,
 854,
 866,
 867,
 902,
 910,
 927,
 949,
 951,
 967,
 975,
 978,
 990]
```

Code Cell 34 of 42

In [29]:
```
### YOUR CODE HERE
insertion_sort_until_k(arr,0, len(arr)-1,k = int(0.1*len(arr)))
```

Python 3 (384MB RAM) | Edit Run All Cells Kernel Stopped |

Out [29]

```
[-982,
 -973,
 -945,
 -910,
 -872,
 -853,
 -836,
 -783,
 -779,
 -749,
 -690,
 -661,
 -635,
 -625,
 -592,
 -565,
 -527,
 -522,
 -515,
 -473,
 -444,
 -441,
 -394,
 -359,
 -339,
 -301,
 -293,
 -260,
 -258,
 -247,
 -239,
 -234,
 -229,
 -193,
 -173,
 -147,
 -131,
 -113,
 -72,
 -59,
 -55,
 -46,
 -32,
 -16,
 -8,
 -4,
 31,
 55,
 93,
 116,
 161,
 181,
 211,
 253,
 255,
 256,
 259,
 289,
 305,
 312,
 323,
 326,
 335,
 335,
 349,
 363,
 366,
 368,
 401,
 409,
 417,
 443,
 488,
 516,
 538,
 622,
 629,
 634,
 637,
 659,
 669,
 701,
 713,
```

Python 3 (384MB RAM) | Edit Run All Cells Kernel Stopped |

```
            821,
            831,
            836,
            839,
            854,
            866,
            867,
            902,
            910,
            927,
            949,
            951,
            967,
            975,
            978,
            990]
```

Code Cell 35 of 42

```
In [30]  ### YOUR CODE HERE
```

Run Code

Code Cell 36 of 42

```
In [31]  ### YOUR CODE HERE

```

Run Code

Code Cell 37 of 42

```
In [32]  ### YOUR CODE HERE
```

Run Code

Question 6 of 8

**D.** Perform a theoretical complexity analysis of the two algorithms you have implemented, and conveyed by the functions **merge_sort_until_k** and **insertion_sort_until_k**. Justify your analysis.

Normal B I U

**merge_sort_until_k:**

We can start deriving the recurrence relation equation by analyzing the structure of the algorithm. Given that we have an input array bigger in length than k, we perform the merge_sort algorithm by consecutively dividing the array in half until the subarrays reach the length of k. This is when we switch to the insertion_sort algorithm. In cases when the input array is already smaller than k, we simply perform insertion sort. Thus, for the merge_sort part, we start by finding the middle value (finding the mean of start and end indexes), which always takes a constant time - O(1). We then call the merge_sort function recursively twice on arrays half the length of the input array. The time it takes to perform this recursion can be expressed as 2T(n/2), where 2 means calling the function twice for each recursion (for the first and second half of the split array), and n/2 represents the length of the new input array, which is half of the previous input array. Lastly, merge_sort runs the merge function once for each recursion. This function takes an array, divides it in half, and goes through all the elements once, comparing the first elements of each subarray and placing them back from smallest to largest. Since it goes through all n numbers of the input array of length n, and it is run once for each

Combining all 3 main parts of the algorithm together, we can express the runtime for the merge_sort part of the merge_sort_until_k algorithm:
T(n)=2T(n/2)+O(n) +O(1)

Now that we have the recurrence relation, we can solve it by consecutively substituting T() terms to represent levels of recursion and get runtime:

T(n)=2T(n/2)+O(n) +O(1)
T(n/2) = 2T(n/4)+O(n/2)+O(1)
T(n/4)=2T(n/8)+O(n/4)+O(1)
and so on.

Hence, by substituting the above values, we get
T(n) = 4T(n/4)+2O(n) +2O(1)= 8T(n/8)+3O(n)+3O(1)

Once we see a pattern, we can generalize and get the equation
T(n)=2kT(n/2^k)+kO(n) +kO(1)

From this equation, we first can find the total number of divisions the algorithm would perform until it would reach the base case (in other words, the depth of the recursion). We know that we stop recursion when we hit the base case, which is when the length of the input array is 1. Then, the sorting does not take much time since an array of length 1 is already sorted. Coming back to the question, we can express the concept above as n/2^k = 1

Solving for k, we can find the number of recursions (depth of recursion):
2^k=n
k=log_2 (n), or simply log n

For each level of recursion, we perform O(n) work (merge function), thus for all levels of recursion the runtime scales as O(n)*log n = O(n*log n).

Now we need to modify the results a little to account for the insertion_sort when the subarray reaches length k. If it wasn't for insertion_sort, the merge_sort algorithm would still perform log k recursions on an input subarray of length k. However, it does not perform these recursions as it stops. Thus, the actual number of recursions it performs is log n (total number) - log k (recursions left until subarray of length 1). Hence, the runtime of merge_sort as part of merge_sort_until_k is O(n) * (log n - log k). The maximum runtime this algorithm could achieve is when it needs to perform all log n levels of recursion, that is, do not stop at all. This can be achieved when k=1 or 0. Thus this algorithm still scales as O(n* log n), which is the upper bound on the runtime when the merge sort needs to do all the work and the insertion sort does not even start.

Now we need to analyze the runtime of the insertion sort part of the merge_sort_until_k algorithm. In the best case input, that is, when the array is already sorted, the insertion sort algorithm only goes through each number of the array of length k once and does not perform any swaps. In code, only for loop runs, and while loop never starts. Thus, for the best-case input, the runtime scales as O(k). For the worst-case input when the array is in the reverse order, the algorithm needs to go through each number of the array of length k once and compare and swap each number with every number to the left of it. In code, both for loop and nested in it while loop, which depends on length k, run. Thus, for the worst-case input, the insertion sort algorithm scales as O(k^2).

Note that the merge sort part does not care about the order of the array.

Thus, combining both parts of the whole algorithm, we get a runtime of O(n* log n) + O(k) for the best case input for insertion sort and O(n* log n) + O(k^2) for the worst case input for insertion sort.

In the task, we were given that k=10 or k = 10% of the array length. The first condition means that k is a constant and does not depend on n. It means that when n grows large (and that is when we apply the asymptotic notation O() ), k stays the same and is smaller than n by a lot. Hence, it should not affect the runtime significantly as it is not the dominant factor. This makes the runtime of the whole merge_sort_until_k algorithm scale as **O(n*log n)** for any input type. The second condition means that k grows as the input size grows, but is still much smaller than the length of the input array (always 10%). Since k<n, the dominant term in O(n* log n) + O(k^2) and O(n* log n) + O(k) is **O(n* log n)** - runtime of merge_sort_until_k. Thus, on average, the runtime of the algorithm is **O(n* log n).**

We can interpret this result as the upper bound on the scaling of the runtime when n grows large. In other words, it means that there exist such positive constants n_0 and c that at and to the right of n_o, f(n) - the function that represents the scaling of the algorithm's runtime - is always equal or less than c*g(n), where g(n)=n*logn. Hence, we know that the algorithm can not run longer than described by n*logn. As we mentioned before, for the merge sort function the input type does not matter, meaning merge_sort_until_k will always scale as n*logn, since the merge sort part is a dominant part of the algorithm. Hence, n*logn is not just an upper bound, but also a lower bound. Because of that, the runtime can also be described using Omega(n*log n) and Theta(n*log n). Omega(n*log n) represents the lower bound on the scaling of the algorithm's runtime, saying that there exist such positive constants n_0 and c that at and to the right of n_o, f(n) - the function that represents the scaling of the algorithm's runtime - is always equal or bigger than c*g(n), where g(n)=n*logn. Since we have the same lower and upper bounds, the best way to describe the runtime of the algorithm is using **Theta (n*log n).** Theta notation provides us with both lower and upper boundaries.

**insertion_sort_until_k**

For this algorithm, we perform insertion sort on the array until we have k numbers left, and that's when we switch to merge sort and perform it on the remaining k numbers. Insertion sort goes through every number and compares it to the numbers to the left of it, placing it in the correct position. In the best-case input scenario, when the list is already ordered, it only goes through each element once, compares them to the numbers on the left, and does not perform any swaps because all numbers to the left are smaller than the current number. In code, it means we only run the first for loop (going through each element), and never run the second nested while loop. Thus, the runtime for the best-case input is O(n). For the worst-case input, when the list is in reverse order, we need to go through every number in the list, compare it to the numbers on the left, and swap the current number with all the numbers on the left. In code, we run both for and nested while loop fully, meaning the complexity rises to O(n^2) for the worst-case input.

For the merge_sort part of the algorithm, we already explained and derived the recurrence relation above. The only difference is that now the runtime depends on k as the input to the merge sort function is a subarray of the last k elements of the input array: T(k)=2T(k/2)+O(k) +O(1). Solving the equation as we did before, we get the runtime of O(k* log k): the depth of the recursive tree is log k and we perform O(k) work at each level.

Combining both together, we get the runtime for the whole insertion_sort_until_k algorithm O(n)+O(k* log k) for the best case input for insertion sort and O(n^2)+O(k* log k) for the worst case input. The dominant term in the best-case input is **O(n)** (n>k), and the dominant term in the worst-case input is **O(n^2).**
As n would grow larger, on average the runtime would be **O(n^2).**

To interpret the results, we can say that if we have the best-case input, and the input size doubles, for example, then the runtime will also double (O(n)). Hence, the scaling is linear. For the worst-case input, if the input size doubles, the runtime quadruples (O(n^2)). Hence, the scaling is quadratic. The same is true for the average case input.

What we have found is the upper bound on the scaling of the runtime of the algorithm. Given the results that we obtained, we can also say that regardless of the input type, the upper bound on the scaling of the runtime is given by O(n^2), while the lower bound is given by Omega(n). Since we have different lower and upper bounds, we can not express the runtime in terms of Theta.

**Note: interpret runtimes, O/omega/theta & why**

Question 7 of 8

**E.** Which algorithm would you choose for the same input and $k = 10$? How would this answer change as $k$ changes for a large input size and different inputs? Justify your answer:

- analytically (by using insights from your analysis to question D),
- experimentally (specifically, $k$=5 and $k$=30),
- and by using at least two different metrics to evaluate efficiency.

Write your conclusions in the text box provided.

Normal

From the analyses above, we established that merge_sort_until_k average runtime is O(n*log n), while insertion_sort_until_k - O(n^2). This notation gives us the upper bound on the scaling of the runtime of the algorithm in an average case scenario, meaning we can be sure it will not run longer than indicated in O notation. From this notation we can also see that the runtime does not depend on k, so no matter if k is constant or increases for larger inputs, the scaling of the runtime will not change as n increases. Thus, we can see that the scaling of merge_sort_until_k is smaller than insertion_sort_until_k as n increases, meaning it would take less time on average to perform the same task using merge_sort_until_k. I would choose merge_sort_until_k.

For the first experiment, we used runtime as a metric of efficiency. It clearly reflects how long it takes for an algorithm to reach the solution given a particular input and hence can be used to compare how well different algorithms perform their task. From the experiments and plots below, we can see that for different values of k - either 5 or 30, the merge_sort_until_k's runtime still scales smaller than the insertion_sort_until_k as the input size grows larger. These results confirm our analytical conclusions - merge_sort_until_k is better for small or large values of k if we consider the asymptotic scaling of the runtime as n grows larger. One thing to note is the unusual behavior of the runtime when n<30 for the k=30 graph. We know that when the array gets smaller than 30, merge_sort_until_k switches to insertion sort, and insertion_sort_until_k switches to merge sort. Up until around n=20, merge_sort_until_k (that runs insertion_sort on this interval) is performing better than insertion_sort_until_k (that runs merge_sort on this interval). This can be explained by the fact that when the input size is very small (such as n<=20 in our case), the insertion sort algorithm performs surprisingly well and even better than the merge sort as we can see from the graph. Then, in the interval 20<n<30, the insertion sort is performing worse than the merge sort since the size of the array grew. Hence, we see that the insertion_sort_until_k function (that runs merge sort on this interval) performs better on this interval than merge_sort_until_k (that runs insertion sort on this interval). After n>30, we are no longer dealing with the subarray of length k, and the algorithm that runs merge sort (merge_sort_until_k) performs better again. For the k=5 graph, the value of k is too small compared to the length of the array so it does not really interfere with the general trend.

The second metric we used is the number of steps used by an algorithm to reach the solution. Note that this metric is very ambiguous as different people can define steps differently, hence it creates certain limitations for comparisons between the efficiency of different algorithms. We can still use it as a rough estimate of algorithms' efficiency, however. Here, the steps are defined as the comparisons, swaps, and array splits made. As we can see from the steps graphs, the overall trend is the same as for the average runtime graphs: merge_sort_until_k performs better than insertion_sort_until_k. For k=5, there are no deviations from the expected trend described earlier. For k=30, however, we observe the same pattern as in the runtime plots: insertion_sort_until_k performs better up until n=30 because on this interval it runs merge sort, while merge_sort_until_k runs insertion sort. After this threshold, the algorithms switch and we can again see that merge_sort_until_k performs better overall.

Thus, all experiments and theoretical analyzes agree on the conclusion that overall, as n grows larger, merge_sort_until_k performs better than insertion_sort_until_k regardless of the value of k.

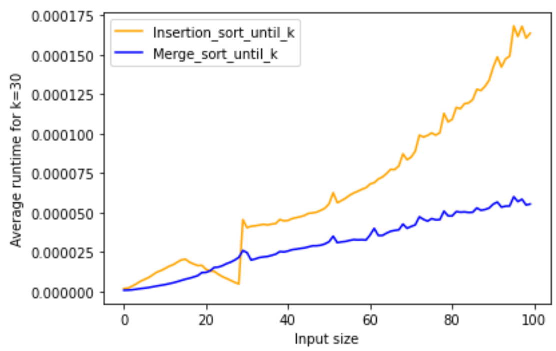
Code Cell 38 of 42

In [33]
```python
av_ins_runtime = []
av_mer_runtime = []

import random
import numpy as np
import time
import matplotlib.pyplot as plt

for i in range(100):#input sizes
    ins_runtime = []
    mer_runtime = []
    for j in range(1000):#to average
        arr = random.choices(range(1000), k=i) #generate random list of length i

        start = time.time()
        insertion_sort_until_k(arr, 0,len(arr)-1,k = 30)
        end=time.time()
        ins_runtime.append(end-start)#record insertion_sort_until_k time for k=30

        start = time.time()
        merge_sort_until_k(arr, 0, len(arr) - 1, k = 30)
        end=time.time()
        mer_runtime.append(end-start) #record merge_sort_until_k time for k=30

    #average runtime
    av_ins_runtime.append(np.mean(ins_runtime))
    av_mer_runtime.append(np.mean(mer_runtime))

plt.plot(range(100), av_ins_runtime, color = 'orange',label='Insertion_sort_until_k')
plt.plot(range(100), av_mer_runtime, color = 'blue',label='Merge_sort_until_k')
plt.legend()
plt.xlabel('Input size')
plt.ylabel('Average runtime for k=30')
plt.title('Average runtime as a function of input size for k=30')
plt.show()
```

Run Code

Python 3 (384MB RAM) | Edit Run All Cells Kernel Stopped |

Out [33]



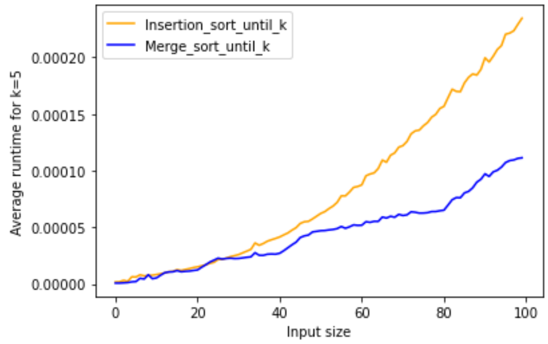

Code Cell 39 of 42

In [34]

```
av_ins_runtime = []
av_mer_runtime = []
import random
import numpy as np
import time
import matplotlib.pyplot as plt

for i in range(100):#inout sizes
    ins_runtime = []
    mer_runtime = []
    for j in range(1000): #to average
        arr = random.choices(range(1000), k=i)#generate random list of length i

        start = time.time()
        insertion_sort_until_k(arr, 0,len(arr)-1,k = 5)
        ins_runtime.append(time.time()-start) #record insertion_sort_until_k time for k=5

        start = time.time()
        merge_sort_until_k(arr, 0, len(arr) - 1, k = 5)
        mer_runtime.append(time.time()-start) #record merge_sort_until_k time for k=5

    #average runtime
    av_ins_runtime.append(np.mean(ins_runtime))
    av_mer_runtime.append(np.mean(mer_runtime))

plt.plot(range(100), av_ins_runtime, color = 'orange',label='Insertion_sort_until_k')
plt.plot(range(100), av_mer_runtime, color = 'blue',label='Merge_sort_until_k')
plt.legend()
plt.xlabel('Input size')
plt.ylabel('Average runtime for k=5')
plt.title('Average runtime as a function of input size for k=5')
plt.show()
```

Run Code

Out [34]




Code Cell 40 of 42

In [35]

```
def merge_steps(arr, start, mid, end,steps):
    '''
    The function sorts a given subarray by splitting it in half,comparing the first numbers of the first and second halves,
    and putting the smaller number in the original array at the corresponding position. It also counts the steps it takes to
run the    function
    -------
    Inputs:
    arr: list
        an array that needs to be sorted
    start: int
        the first index of the array
    mid: int
        the middle index of the array
    end: int
        the last index of the array
    steps: int
        the variables that stores the number of steps executed
    ------
    Outputs:
    arr: list
        a sorted array
    steps: int
        the number of steps it takes to execute the function added to the number of steps previously executed
    '''

    #split the array in half
    left_array=arr[start:mid+1]
    right_array=arr[mid+1:end+1]
    #initiate indexes to iterate over subarrays
    index_left=0
    index_right=0
    #ensure we do not run out of range
    left_array.append(float('inf'))
    right_array.append(float('inf'))

    for i in range(start,end+1):
        steps+=1 #count comparisons as steps
        #compare corrsponding numbers of both array to put the smaller in the correct position
        if left_array[index_left] <= right_array[index_right]:
            arr[i]=left_array[index_left]
            index_left+=1
        else:
            arr[i]=right_array[index_right]
            index_right+=1
```

```python

def merge_sort_until_k_steps(arr, start, end, k = 10,steps=0):
    '''
    The function recursively devides an array into smallest subarrays until they reach length 2 and then merges them back into sorted array. Also counts steps it takes to execute the function
    -------
    Inputs:
    arr: list
        an array that needs to be sorted
    start: int
        the first index of the array
    end: int
        the last index of the array
    k: int
        the threshold length of the subarray to switch algorithms
    steps: int
        initiates a variable to track steps
    ------
    Outputs:
    arr: list
        a sorted array
    steps: int
        the number of steps it takes to execute the function
    '''

    #switch to insertion when reach subarray of length k
    if end-start<k:
        ins = insertion_sort_steps(arr[start:end+1],steps) #sort the subarray of length k
        steps = ins[1] #add steps for insertion_sort part
        arr=arr[:start] + ins[0] + arr[end+1:]
    else:
        #define the middle index of the array
        mid =(start+end)//2
        steps+=1 #for splitting an array in half
        #keep dividing the array into left parts
        arr=merge_sort_until_k_steps(arr,start,mid, k,steps)[0]
        #keep dividing the array into right parts
        arr=merge_sort_until_k_steps(arr,mid+1,end, k,steps)[0]
        #if the subarray is bigger than k elements, perform merge sort, otherwise - insertion sort
        steps = merge_steps(arr,start,mid,end,steps)[1] #merge the subarrays into bigger arrays until we get the original array sorted
    return [arr,steps]


def insertion_sort_steps(arr,steps):
    '''
    The function implements insertion sort algorithm : compares every consequetive number in the array to the number on the left until find the right position to place the current number. Also counts steps it takes to run the function
    -------
    Inpput:
    arr: list
        an array of numbers - array
    steps: int
        a variable that counts steps
    -------
    Output:
    arr: list
        sorted array
    steps: int
        the number of steps it took to run the function plus the steps already counted
    '''
    for i in range(1, len(arr)): #iterating through all the indexes in the array starting with the second element (index=1)
```

```
        key = arr[j] #assigning the second number in the array to a key variable
        i = j-1 #defining another index, smaller than the previous by one
        steps+=1 #to go through each element and swap it after the while loop

        while i >= 0 and arr[i] > key:
            steps+=1 #for each comparison to the left and swap
            arr[i+1] = arr[i] #if the element at index i is bigger than the key, put it at the position of the key
            i -= 1 #update index i by decreasing by one

        arr[i+1] = key #position the key element after the elemnt that is smaller than key
    return [arr,steps]



def insertion_sort_until_k_steps(arr, start,end,k = 10,steps=0):
    '''
    The function implements insertion sort algorithm until k elements left. Then it swtiches to merge sort. It also counts steps it takes to turn the algorithm
    -------
    Inpput:
    arr: list
        an array of numbers - array
    start: int
        the first index of the array
    end: int
        the last index of the array
    k: int
        the threshold length of the subarray to switch algorithms
    steps: int
        a variable that counts steps
    -------
    Output:
    arr: list
        sorted array
    steps: int
        the number of steps it took to run the function plus the steps already counted
    '''
    for j in range(1, len(arr)-k+1): #iterating through all the indexes in the array starting with the second element (index=1)
        key = arr[j] #assigning the second number in the array to a key variable
        i = j-1 #defining another index, smaller than the previous by one
        steps+=1 #to go through each element and swap it after the while loop

        while i >= 0 and arr[i] > key:
            steps+=1 #for each comparison to the left and swap
            arr[i+1] = arr[i] #if the element at index i is bigger than the key, put it at the position of the key
            i -= 1 #update index i by decreasing by one

        arr[i+1] = key #position the key element after the elemnt that is smaller than key

    #perform merge_sort on the last k elements and record the steps it takes
    arr=arr[:len(arr)-k+1]+merge_sort_steps(arr[len(arr)-k+1:],0,len(arr[len(arr)-k+1:])-1,steps)[0]
    steps=merge_sort_steps(arr[len(arr)-k+1:],0,len(arr[len(arr)-k+1:])-1,steps)[1]

    #merge 2 subarrays sorted by insertion sort and merge sort into one sorted array and record the steps it takes
    arr=merge_steps(arr,0,len(arr)-k,len(arr)-1,steps)[0]
    steps=merge_steps(arr,0,len(arr)-k,len(arr)-1,steps)[1]

    return [arr,steps]
```

```
    '''
    The function recursively devides an array into smallest subarrays until they reach length 1 and then merges them back into sorted array. Also counts steps it takes to execute the function
    -------
    Inputs:
    arr: list
        an array that needs to be sorted
    start: int
        the first index of the array
    end: int
        the last index of the array
    steps: int
        initiates a variable to track steps
    ------
    Outputs:
    arr: list
        a sorted array
    steps: int
        the number of steps it takes to execute the function
    '''

    #base case to reach smallest subarrays of length 1
    if start < end:
        #define the middle index of the array
        mid = (start+end)//2
        steps+=1 #to split the array in half
        #keep deviding the array into left parts
        merge_sort_steps(arr,start,mid,steps)
        #keep deviding the array into right parts
        merge_sort_steps(arr,mid+1,end,steps)
        steps=merge_steps(arr,start,mid,end,steps)[1] #merge the subarrays into bigger arrays until we get the original array sorted
    return [arr,steps]
```

Run Code

Code Cell 41 of 42

In [36]

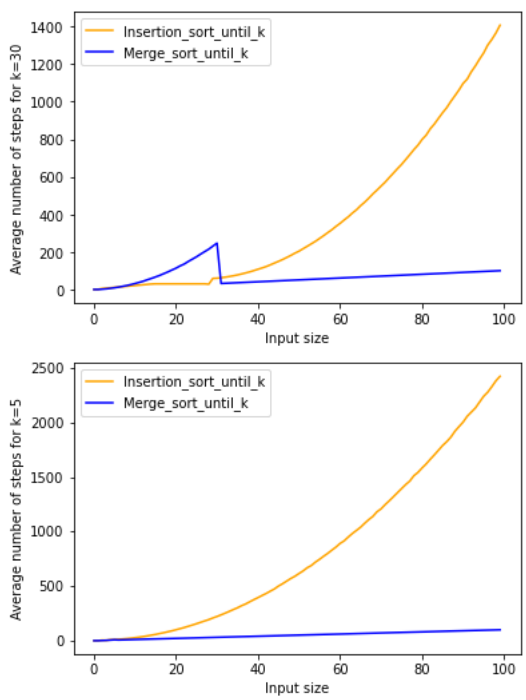
```
av_ins_steps=[]
av_mer_steps=[]
for i in range(100): #input size
    ins_steps=[]
    mer_steps=[]
    for j in range(1000): #to average
        arr = random.choices(range(1000), k=i)#generate random list of size i
        #record the steps
        ins_steps.append(insertion_sort_until_k_steps(arr, 0, len(arr)-1, k = 5)[1])
        mer_steps.append(merge_sort_until_k_steps(arr, 0, len(arr)-1, k = 5)[1])

    #average the steps
    av_ins_steps.append(np.mean(ins_steps))
    av_mer_steps.append(np.mean(mer_steps))


av_ins_steps_k30=[]
av_mer_steps_k30=[]
for i in range(100):#input size
    ins_steps=[]
    mer_steps=[]
    for j in range(1000): #to average
```

Python 3 (384MB RAM) | Edit Run All Cells Kernel Stopped |

```
        arr = random.choices(range(1000), k=i)#generate random list of size i
        #record the steps
        ins_steps.append(insertion_sort_until_k_steps(arr, 0, len(arr)-1, k = 30)[1])
        mer_steps.append(merge_sort_until_k_steps(arr, 0, len(arr)-1, k = 30)[1])

    #average the steps
    av_ins_steps_k30.append(np.mean(ins_steps))
    av_mer_steps_k30.append(np.mean(mer_steps))



plt.plot(range(100), av_ins_steps_k30, color = 'orange',label='Insertion_sort_until_k')
plt.plot(range(100), av_mer_steps_k30, color = 'blue',label='Merge_sort_until_k')
plt.legend()
plt.xlabel('Input size')
plt.ylabel('Average number of steps')
plt.title('Average number of steps as a function of input size for k=30')
plt.show()

plt.plot(range(100), av_ins_steps, color = 'orange',label='Insertion_sort_until_k')
plt.plot(range(100), av_mer_steps, color = 'blue',label='Merge_sort_until_k')
plt.legend()
plt.xlabel('Input size')
plt.ylabel('Average number of steps for k=5')
plt.title('Average number of steps as a function of input size for k=5')
plt.show()
```

Run Code

Out [36]




Code Cell 42 of 42

In [37]

```
### PLEASE DO NOT USE THIS CELL, IT WILL BE USED FOR FURTHER TESTING
```

Python 3 (384MB RAM) | Edit Run All Cells Kernel Stopped |

Run Code

Question 8 of 8

# References

Please write here all the references you have used for your work.

Normal

s

🏁 You are all done! Congratulations on finishing your second CS110 assignment!
🎉

Python 3 (384MB RAM) | Edit Run All Cells Kernel Stopped |