Esther

February 7, 2021

Before you turn this assignment in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

[ ]:
```
NAME = "Esther"
COLLABORATORS = "Ha, Meliane, Yufei, Shreya, Felipe"
```

---

1 CS110 Assignment 1

2 Algorithm design & sorting - Problem set

Instead of a mini-project, which you will encounter in the upcoming CS110 assignments, this time you will be solving three independent problems. Use this opportunity to start your work early, finishing a question every week or so of class. Ideally, in the last week before the deadline, you will only have Q3 left to complete.

Fell free to add more cells to the ones always provided in each question to expand your answers, as needed. Make sure to refer to the CS110 course guide on the grading guidelines, namely how many HC identifications and applications you are expected to include in each assignment.

If you have any questions, do not hesitate to reach out to the TAs in the Slack channel "#cs110-algo", or come to one of your instructors' OHs.

2.0.1 Submission Materials

Your assignment submission needs to include the following resources: 1. A PDF file must be the first resource and it will be created from the Jupyter notebook template provided in these instructions. Please make sure to use the same function names as the ones provided in the template. If your name is "Dumbledore", your PDF should be named "Dumbledore.pdf". 2. Your second resource must be a single Python/Jupyter Notebook named "Dumbledore.ipynb". You can also submit a zip file that includes your Jupyter notebook, but please make sure to name it "Dumbledore.zip" (if your name is Dumbledore!).

For details on how to create a nice PDF from a Jupyter notebook, refer again to the CS110 course guide.

2.1 Question 0 [#responsibility]

Take a screenshot of your CS110 dashboard on Forum where the following is visible: * your name. * your absences for the course have been set to excused up to the end of week 3 (inclusively).

This will be evidence that you have submitted acceptable pre-class and make-up work for a CS110 session you may have missed. Check the specific CS110 make-up and pre-class policies in the syllabus of the course.

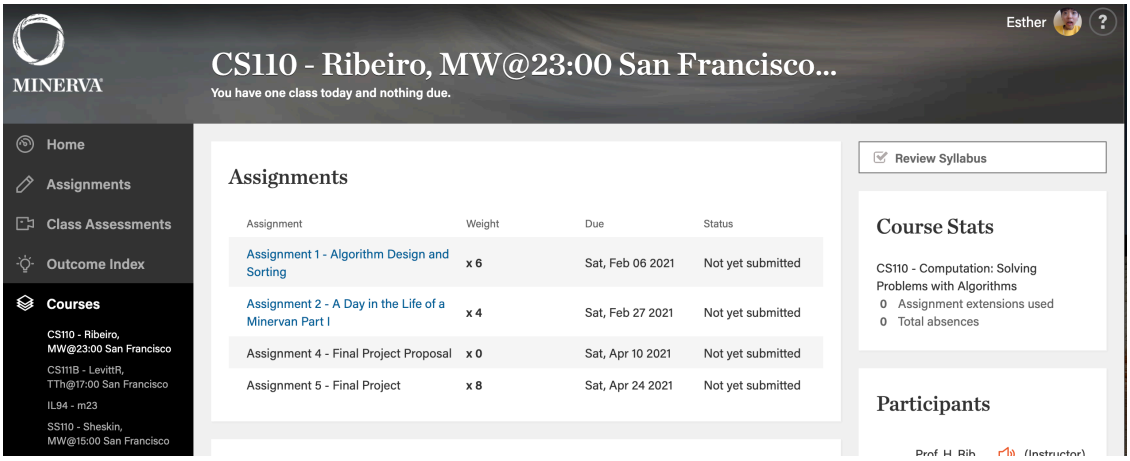
[ ]:
```
# Image(filename="")
from google.colab import files
from IPython.display import Image
```

[ ]:
```
upload = files.upload()
```

```
<IPython.core.display.HTML object>


Saving forum.png to forum.png
```

[ ]:
```
Image('forum.png')
```

[ ]:



2.2 Question 1: Iteration vs. recursion [#ComputationalCritique, #PythonProgramming]

As a famous mathematical puzzle, the Tower of Hanoi is often used in introductory computer science classes to teach recursion. To learn about the puzzle and to understand the recursive solution, **watch this video**. Take your time to understand why and how the presented approach works.

Use the code skeletons below to **provide two solutions** to the problem:

First, a **recursive solution**. You are welcome to use the code from the video, with the following modifications: - The *start* and *end* parameters have defaults, $start = 1$ and $end = 3$, so that the user doesn't have to bother with inputting them (but can still change them if desired). - Instead

of printing the rod to rod transitions, make the function output a list of tuples, where each tuple specifies one transition. For example, calling hanoi(2) should return [(1, 2), (1, 3), (2, 3)].

Second, provide an **iterative solution**, and **explain** how it works. If you use an online source for your algorithm idea, cite it, but your algorithm implementation and explanation needs to be your own. The outputs of both your functions should be identical.

Then, **answer** the following 4 questions, in approximately 150 words: - **Explain** how your algorithmic solutions follow the iterative and recursive paradigms, respectively. - **Discuss the pros and cons** of each approach to this problem. - Perform an **experimental analysis** (based on criteria that you deem relevant and important) to contrast the computational efficiency of both approaches. This analysis should include **at least one plot**. - Finally, building on the previous answers, state **which of your solutions you think is better and why**.

Note: Watch out for the unintuitive Python behaviour if defining functions with mutable default parameter values.

2.3 How iterative solution works?

First, let's observe the number of steps needed for * 2 layer: 1+2 = 3 * 3 layer: 3+1+3 = 7 * 4 layer: 7+1+7 = (3+1+3) + 1 + (3+1+3) = 15 * 5 layer: recursive 5: 15+1+15 = (7+ 1+ 7) + 1 + (7+1+7) = ((3+1+3)+ 1 + (3+1+3)) + 1 + ((3+1+3)+ 1 + (3+1+3))= 31 We see that the steps for is 2 **n - 1. We can also see that we can break it down a complicated layer into simple layers. Aftering breaking down, it is obvious that every three step is a cycle. Based on the cycle, we construct start, middle, and end list to keep track of layers in each pillar.

Now, the constraint is that only small can move on the top so we need to compare the top layer at each pillar when moving around.

Iterative solution sources

2.4 Pros & Cons for iterative solution

Pros: Easy for debugging. We can check if every moves is correct or not.

Cons: More computational power. For every step, we need to check and move.

2.5 How recursive solution works?

If there is only one layer, it is the base case that we record the starting to the ending point. For multiple layer, we start from moving the n-1 layer from the starting to the spare(middle) and the bottom layer from the starting to ending pillar. And then, after we move the bottom layer, will move the n-1 layer from the middle to the end.

Recursive solution sources

2.6 Pros & Cons for recursive solution

Pros: 1. Short code for implementation 2. Mathematical proof through the relationship when n = 1 is true and n-1 is true, n will be true too.

Cons: 1. Hard to debug 2. We need more extra resources to fully understand the code

[ ]:
```
def hanoi_recursive(n, start = 1, end = 3, spare =2, result = None):
  if result == None:
```

```
    result = []

  if n == 1:
    my_tuple = (start, end)
    result.append(my_tuple)
  else:
    hanoi_recursive(n-1,start, spare, end, result) #need to specify the new␣
 ↪result here in order so result won't be back to none again
    my_tuple = (start, end)
    result.append(my_tuple)
    hanoi_recursive(n-1,spare, end, start, result)
  return result

# hanoi_recursive(4)


def hanoi_iterative(n, start=1, middle = 2, end=3):
    MyList = []

    StartList = [float('inf')]
    MiddleList = [float('inf')]
    EndList = [float('inf')]
    StartList.extend([i for i in range(n,0,-1)])

    steps = 2 **n  - 1

    if n% 2 == 0: #if the number of layer is even, we change the end and the␣
 ↪middle point.
      end = 2
      middle = 3

    #Three steps as a cycle
    for i in range(steps):
      if i % 3 == 0:
        # print(StartList, EndList)
        if StartList[-1] > EndList[-1]: #move StartList to EndList or vice␣
 ↪versa depends on who is bigger
          StartList.append(EndList.pop())
          MyTuple = (end, start)
        else:
          EndList.append(StartList.pop())
          MyTuple = (start, end)

        MyList.append(MyTuple)


      elif i % 3 == 1:
```

```
        if StartList[-1] > MiddleList[-1]: #move StartList to EndList or vice␣
↪versa depends on who is bigger
          StartList.append(MiddleList.pop())
          MyTuple = (middle, start)
        else:
          MiddleList.append(StartList.pop())
          MyTuple = (start, middle)
        MyList.append(MyTuple)


      else:
        if MiddleList[-1] > EndList[-1]: #move MiddleList to EndList or vice␣
↪versa depends on who is bigger
          MiddleList.append(EndList.pop())
          MyTuple = (end, middle)
        else:
          EndList.append(MiddleList.pop())
          MyTuple = (middle, end)

        MyList.append(MyTuple)

    return MyList

# hanoi_iterative(4)
```

[ ]:
```
#Method from the video
def hanoi(n, start, end):
  if n == 1:
    return start, end
  else:
    other = 6-(start+end)
    honoi(start, other)
    pm(start, end)
    honoi(other, end)

  honoi(3, 1,3)
```

[ ]:
```
assert hanoi_recursive(2) == [(1, 2), (1, 3), (2, 3)]
assert hanoi_iterative(2) == [(1, 2), (1, 3), (2, 3)]
```

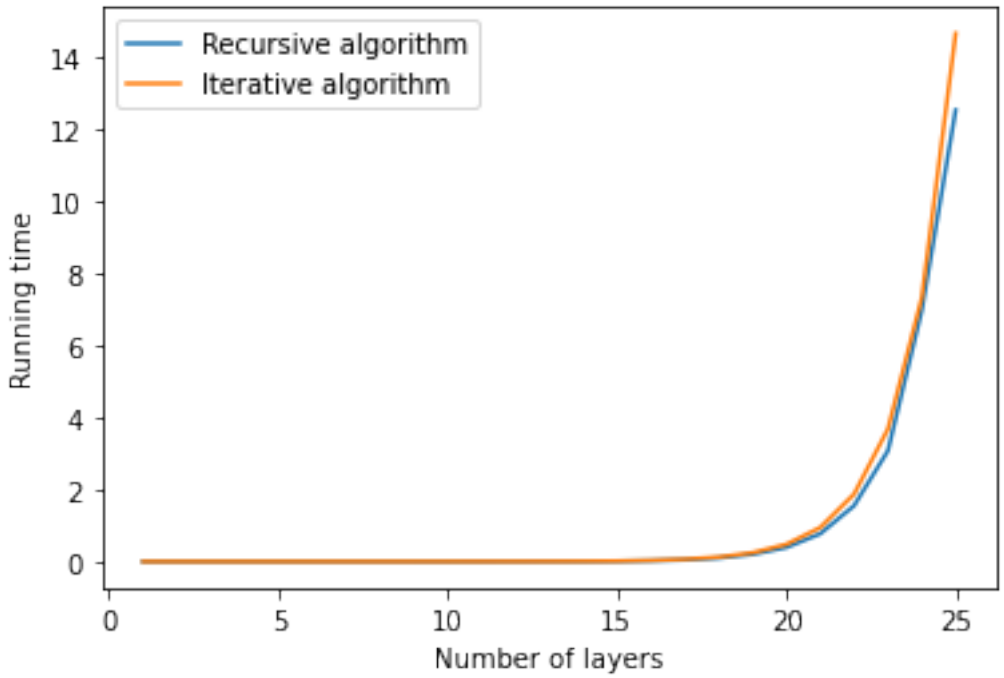
2.7 Experinmental Analysis

[ ]:
```
#Compare the computational time for two algorithm
import time

time_hanoi_recursive = []
time_hanoi_iterative = []
```

```
#Time for hanoi recursive algorithm up to 20 layers
for i in range(1,26):
  start = time.time()
  hanoi_recursive(i)
  end = time.time()
  duration = end - start
  time_hanoi_recursive.append(duration)

#Time for hanoi iterative algorithm up to 20 layers
for i in range(1,26):
  start = time.time()
  hanoi_iterative(i)
  end = time.time()
  duration = end - start
  time_hanoi_iterative.append(duration)
```

[ ]:
```
import matplotlib.pyplot as plt

layers = [i for i in range(1,26)]
plt.plot(layers, time_hanoi_recursive, label = "Recursive algorithm")
plt.plot(layers, time_hanoi_iterative, label = "Iterative algorithm")
plt.xlabel("Number of layers")
plt.ylabel("Running time")
plt.legend()
```

[ ]:
```
<matplotlib.legend.Legend at 0x7f9866b13ef0>
```



From the plot, we can see recursive algorithm is a little bit better than the iterative solution. However, looking at the slope, we might expect the recursive solution will perform better when we have a larger number of layers.

2.8 Question 2: Understanding and documenting code [#CodeReadability, #AlgorithmicStrategies]

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 show the code working correctly, but they seem far from comprehensive. Your tasks are listed below. Here is the code:

[ ]:
```
def my_sort(lst):
    """
    YOUR DOCSTRING HERE
    """

    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
```

2.8.1 Question 2a: Understanding code

Explain, in your own words, **what the code is doing** (it's sorting an array, yes, but how?). Feel free to use diagrams, play around with the code in other cells, print test cases or partially sorted arrays, draw step by step images. In the end, you should produce an approximately 150-word write-up of how the code is processing the input array.

We can understand the code block by block. First, we append an element in piles to create an initial setup. Next, the purpose of creating a

```
while lst:
```

is to create several nested lists for piles if the new value added in is smaller. Therefore, we will get multiple sorted nested list for piles. Lastly, in the block of

```
while len(lst) < n:
```

We try to find the biggest element in every nested list. Find the index of a list which has the smallest element among all the biggest elements. We return the first element, which is the smallest, in that specific list.

2.8.2 Question 2b: Documenting code

Explain the difference between docstrings and comments. **Add** both a proper **docstring** and several **in-line comments** to the code (there is an editable copy below). You can follow the empty comments to guide you, but you can deviate, within reason. If this topic seems new to you, here, here and here are some resources to get you started. Anyone from your section should be able to understand the code from your documentation. Remember, however, that brevity is also a desirable feature.

We can view docstrings as a documentation for the class, module, and packages. Comments are mainly used to explain the code inside or point out the place that bugs need to be fixed.

We usually put a docstring right under a function to explain the whole purpose of the function. We commonly put the purpose of the function, meaning of input and expected ouput in a docstring.

We can view docstring of as the output when we run help of a function, while comments indicate a step-by-step of how the code is implemented.

[ ]:
```
def my_sort(lst):
    ''' Implements a sorting algorithm

    Parameters
    ----------
    lst: list
      A list.

    Returns
    ----------
    A sorted list

    '''

    n = len(lst)

    piles = []
    piles.append([lst.pop(0)]) #Take out the first value in the list and put it␣
↪in the pile in a nested way


    #we create new nested list for piles if the new value added in is smaller.
    #Therefore, for piles, we will get multiple sorted nested list.

    while lst:

        item = lst.pop(0) #find the next item of the lst
        # print('item', item)
        placed = False

        #place the next "item" in a new nested list if it is smaller than␣
↪smallest value in the pile
        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] #the biggest element in the nested␣
↪list

        idx_smallest = tops.index(min(tops)) #indexing the smallest value in␣
↪the biggest element so far
```

```
        lst.append(piles[idx_smallest].pop(0)) #Break the code: being smallest␣
↪value at the end of the nested list doesn't mean that their first value will␣
↪be the smallest too

        piles = list(filter(None, piles)) #throw away the empty nested list

    return lst
```

2.8.3 Question 2c: Testing & fixing code

Why are the tests that you are presented with insufficient?

Find at least two 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: Play the scenario out with a small example, e.g. 3 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.

1. my_sort([2,5,3,4,1,6,7])
2. my_sort([0,3,2,6,7,1,5])
3. my_sort([])
4. my_sort([[3,1]])

[ ]:
```
print(my_sort([2,5,3,4,1,6,7])) #wrong case
print(my_sort([0,3,2,6,7,1,5])) #wrong case
```

```
[1, 3, 4, 2, 5, 6, 7]
[1, 2, 5, 0, 3, 6, 7]
```

The reason behind 1. &2. is that being smallest value at the end of the nested list doesn't mean that their first value will be the smallest too. In my fixed code, I also don't allow people to input an empty list or a nested list.

[ ]:
```
def my_fixed_sort(lst):
  '''Implement a sorting algorithm

    Parameters
    ----------
    lst: list
      A list.

    Returns
    ----------
    A sorted list

  '''

    n = len(lst)
```

```
   if (n<= 0) or (type(lst) != list) or  any(isinstance(sub, list) for sub in␣
↪lst):
       # check if it is an empty list, the type of the lst is the list, or if␣
↪there is a empty lists
       # print('wrong input')
       raise Exception('wrong input')


   piles = []
   piles.append([lst.pop(0)]) #Take out the first value in the list and put it␣
↪in the pile in a nested way

   # print('1', piles)

   #we create new nested list for piles if the new value added in is smaller.
   #Therefore, for piles, we will get multiple sorted nested list.
   while lst: #Only run len(lst) - 1 times

       item = lst.pop(0) #find the next item of the lst

       placed = False

       #place the next "item" in a new nested list if it is smaller than␣
↪smallest value in the pile
       for pile in piles:
           if item > pile[-1]:
               pile.append(item)
               placed = True
               break
       if not placed:
           piles.append([item])
       # print('2',piles)
   # print('3',piles)


   while len(lst) < n:
       bottom = [pile[0] for pile in piles] #the smallest element in the␣
↪nested list

       idx_smallest = bottom.index(min(bottom)) #indexing the smallest value␣
↪in the biggest element so far

       lst.append(piles[idx_smallest].pop(0)) #Fix the code: pop out the␣
↪smallest element among the smallest values

       piles = list(filter(None, piles)) #throw away the empty nested list
```

```
    return lst

print(my_fixed_sort([2,5,3,4,1,6,7])) #wrong case in my_sort
print(my_fixed_sort([0,3,2,6,7,1,5])) #wrong case in my_sort
# print(my_fixed_sort([])) #don't allow people to input an empty list
# print(my_fixed_sort([[3,1]])) #don't allow people to input a nested list
```

```
[1, 2, 3, 4, 5, 6, 7]
[0, 1, 2, 3, 5, 6, 7]
```

2.9 Question 3: New and mixed sorting approaches [#AlgorithmicStrategies, #ComputationalCritique, #PythonProgramming, #ComplexityAnalysis]

In this question, you will implement and critique a sorting algorithm we haven't discussed in class. You will then combine it with another, known sorting algorithm, to see whether you can reach better behavior. This is the most difficult of the assignment questions, so schedule enough time for it.

2.9.1 Question 3a: Implementation from pseudocode

Use the following pseudocode to **implement recursive bucket_sort()**.

Bucket sort (or Bin sort) is an algorithm that takes as inputs an n-element array and the number of buckets, k, to be used during sorting. Then, the algorithm distributes the elements of the input array into k different buckets and proceeds to sort the individual buckets. Then, it merges the sorted buckets to obtain the sorted array.

Here is pseudocode for the bucket_sort algorithm:

[ ]:
```
# bucket_sort(lst, k):
#   mn = the minimum value in the array
#   mx = the maximum value in the array
#   sz = size of the numerical interval of every bucket (ceiling((max - min)/
 ↪k))
#   buckets <- array of k empty buckets
#
#   # distribute elements in k buckets
#   for i = 1 to lst.length:
#     b = get_bucket_num(lst[i], mn, mx, sz, k)
#     buckets[b].append(lst[i])
#
#   # sort buckets idividually
#   for i = 1 to k:
#     sort(buckets[i])
#
#   # concatenate contents of buckets
#   lst = buckets[1] + buckets[2] + ... + buckets[k]
#
#   return sorted list
```

where for the sort() function in the second for loop, you should use bucket_sort recursively (remember that this will require you to also introduce a base case).

The bucket_sort above calls the function get_bucket_num (see the pseudocode below) to distribute all the elements of array *lst* into *k* buckets. Every element in the array is assigned a bucket number based on its value (positive or negative numbers). get_bucket_num returns the bucket number that corresponds to element $lst[i]$. It takes as inputs the element of the array, $lst[i]$, the max and min elements in *lst*, the size of the intervals in every bucket (e.g., if you have numbers with values between 0 and 100 numbers and 5 buckets, every bucket has an interval of size $20 = [1000]/5$). Notice that in pseudocode the indices of the arrays are from 1 to n. Thus, get_bucket_num consistently returns a number between 1 and n (make sure you account for this in your Python program).

Note: Watch out for the unintuitive Python behaviour if defining functions with mutable default parameter values.

[ ]:
```
import sys
import math

sys.setrecursionlimit(50000) #increase the recursion limit

def get_bucket_num(a, mn, mx, sz, k):
    """Calculates which bucket a given element should be sorted to.

    Parameters
    ----------
    a: an element of the list (usually a number)
    mn: smallest element value
    mx: largest element value
    sz: bucket size
    k: number of buckets

    Returns
    -------
    int: the correct bucket number for element a

    """
    if a == mx:
      j = k -1
    elif a == mn:
      j = 0
    else:
      j = 0
      while a >= mn+(sz*(j+1)): #when the bucket threshold is bigger than the␣
↪value, we place it into that bucket. If a is bigger than the upper limit of␣
↪a buckets, we need a larger bucket.
        j += 1
    return j
```

```
def bucket_sort(lst, k):
    """Implements recursive BucketSort

    Parameters
    ----------
    array : Python list or numpy array
    k : int
        number of buckets to used in sorting

    Returns
    -------
    array: a sorted Python list

    """
    #Lowest min for the number of bucket
    if k <= 1:
      raise Exception('wrong input')

    #duplicate checking system
    same = True # to check if every element in the list are the same
    for l in range(len(lst)):
      if lst[0] != lst[l]:
        same = False
        break
    #Ha's code: all(element == lst[0] for element in lst)


    if len(lst) <= 1 or same == True: #base case: no need to create more bucket␣
↪when 1. only one element in the list or 2. every element in the list is the␣
↪same
      return lst


    else:  #Other cases: we create buckets
      mx = max(lst)
      mn = min(lst)
      sz = math.ceil((mx -mn)/k)
      buckets = [[] for l in range(k)] #The number of buckets we need

      #we finish the buckets creation
      for i in range(len(lst)):
        b = get_bucket_num(lst[i], mn, mx, sz, k)
        buckets[b].append(lst[i])


      #Run through the every buckets to check if we need to create sub buckets␣
↪through recursion
```

```
      for j in range(len(buckets)):
        buckets[j] = bucket_sort(buckets[j], k)
      # print(buckets)
      lst = [item for sub_bucket in buckets for item in sub_bucket] #flatten␣
 ↪the bucket lists = add all the buckets togethers

    return lst



input_array = [8, 5, 4, 6, 7, 2, 9, 1, 3]
input_array_2 = [77,3,8,2,1,6,0,22,844]
input_array_3 = [-3,8,1,-33, 10,22,5,9,2]
input_array_4 = [10,9,8,7,6,5,4,3,2,1]
input_array_5 = [1,10,20,30,40,50,60,70,80,90]
input_array_6 = [1,6,3,6,6,6,4,8,1,2]


assert bucket_sort(input_array, 3) == sorted(input_array)
assert bucket_sort(input_array_2, 10) == sorted(input_array_2) #include big␣
 ↪positive number: lots of recursion in this case
assert bucket_sort(input_array_3, 2) == sorted(input_array_3) #include negative␣
 ↪number: lots of recursion in this case
assert bucket_sort(input_array_4, 5) == sorted(input_array_4) #include the list␣
 ↪in opposite order
assert bucket_sort(input_array_5, 20) == sorted(input_array_5) #include where␣
 ↪edge cases might happen, when bucket is twice bigger as the list number␣
 ↪increases
assert bucket_sort(input_array_6, 4) == sorted(input_array_6) #include lots of␣
 ↪duplicates
```

2.9.2 Question 3b: Testing your code

To the singular test provided below, **add** at least **5 more assert statements**, which showcase that your code works as intended. In a few sentences, **justify** why your set of tests is appropriate and possibly sufficient.

[ ]:
```
input_array = [8, 5, 4, 6, 7, 2, 9, 1, 3]
assert bucket_sort(input_array, 3) == sorted(input_array)
```

In my test cases, 1. I input a big positive number negative numbers, and reset the limit of recursion. 2. I includes negative numbers to see if the code will works in this scenario. 3. I include special case such as the list in a total opposite order 4. I include a list with arithmetic progression but the bucket size is two times the difference of the list to test if the algorithm will pass the edge case. 5. I include duplicates. I add a duplicate checking system to prevent endless loop to break same values into different buckets.

2.9.3 Question 3c: Mixing algorithms

The algorithm becomes unnecessarily complicated when it tries to sort a really short piece of the original array (e.g. bucketing a 10-element array into 333 buckets). To work around this, **implement the following**: if the input sublist length is below a certain threshold (which you decide), sort it by bubble sort instead of continuing to recurse. To ensure you won't break your old code, first copy it to the cell below, and then create the new version here.

Justify on the basis of analytical or even experimental arguments what might be the optimal threshold for switching to bubble sort.

Remember that **tests** are important.

Bubble Sort Experinment

[ ]:
```
#Bubble sort time
import time
def bubble_sort_count(A):
    step_count = 0
    start_time  = time.time()
    for i in range(0, len(A)-1):
      step_count += 1
      for j in range(len(A)-1,i,-1):
        if A[j] < A[j-1]:
          A[j], A[j-1] = A[j-1],A[j]
          step_count += 2 #count swap as two steps

    end_time = time.time()
    duration = end_time - start_time


    return A, step_count, duration
```

[ ]:
```
#Random input generator
import random
def generate_inputs(type_input, N, interval):
    '''
    This function generates data of size 1..N in the format of the inputted␣
 ↪'type_input' variable.
    --------
    Inputs:
    -type_input: options are 'Sorted', 'Reversed' or 'Random' that indicate the␣
 ↪format of the data generated
    -N: maximum size of input list
    -interval: interval of sizes for the elements on the list
    Outputs:
    -A list of lists to be sorted
    '''
    data = []
    for i in range(1, N+1, interval):
```

```
        if type_input == 'Reversed':
            data.append([j for j in range(i+1,1,-1)])
        elif type_input == 'Random':
            data.append([random.randint(1,1000) for j in range(i)])
        elif type_input == 'Sorted':
            data.append([j for j in range(1, i,1)])

        else:
            raise ValueError('This is an unsupported type_input; please try␣
 ↪again.')
    return data

N = 1000
random_input = generate_inputs('Random',N,1)
```

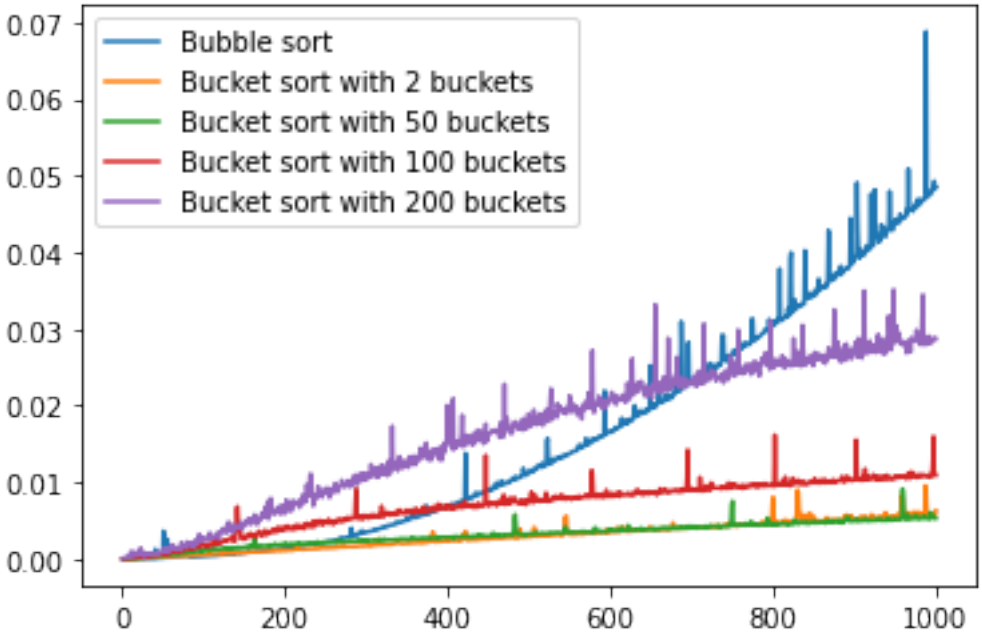
[ ]:
```
import matplotlib.pyplot as plt

x = [i for i in range(1,N+1)]
bubble_time = []
for k in range(len(random_input)):
  bubble_time.append(bubble_sort_count(random_input[k])[2])
```

[ ]:
```
#Experinment bucekt sort performance with different input length and buckets
#Random input

bucket_time_small = []
for k in range(len(random_input)):
  start = time.time()
  bucket_sort(random_input[k],2) #fix the numbers of bucket
  end = time.time()
  duration = end - start
  bucket_time_small.append(duration)

bucket_time_medium = []
for k in range(len(random_input)):
  start = time.time()
  bucket_sort(random_input[k],50) #fix the numbers of bucket
  end = time.time()
  duration = end - start
  bucket_time_medium.append(duration)

bucket_time_large = []
for k in range(len(random_input)):
  start = time.time()
  bucket_sort(random_input[k],100) #fix the numbers of bucket
  end = time.time()
  duration = end - start
  bucket_time_large.append(duration)
```

[ ]:
```
bucket_time_max = []
for k in range(len(random_input)):
  start = time.time()
  bucket_sort(random_input[k],200) #fix the numbers of bucket
  end = time.time()
  duration = end - start
  bucket_time_max.append(duration)
```

[ ]:
```
plt.plot(x, bubble_time, label = 'Bubble sort')
plt.plot(x, bucket_time_small, label = 'Bucket sort with 2 buckets')
plt.plot(x, bucket_time_medium, label = 'Bucket sort with 50 buckets')
plt.plot(x, bucket_time_large, label = 'Bucket sort with 100 buckets')
plt.plot(x, bucket_time_max, label = 'Bucket sort with 200 buckets')
plt.legend()
```

[ ]:
```
<matplotlib.legend.Legend at 0x7f936401d978>
```



I experinment different length of random input and different number of bucket. It shows that the performance of bucket sort correlates with both variables. When there are more buckets, the more we should use bubble sort. For instance, when there is only 2 or 50 buckets, we will choose bubble sort when the length of input is less than 200. When there are 100 buckets. We will choose to use bubble sort when the length of input is smaller than 400. When we have 200 buckets, input length of 700 becomes the threshold.

Therefore, we cannot only use the length of input to decide the threshold but also need to consider the numbers of buckets.

[ ]:
```
#Optimized bucket sort which consider the threshold
import sys
```

```
import math

sys.setrecursionlimit(50000) #increase the recursion limit


def optimized_bucket_sort(lst, k, threshold): #include the threshold
    """Implements recursive BucketSort

    Parameters
    ----------
    array : Python list or numpy array
    k : int
        number of buckets to used in sorting

    Returns
    -------
    array: a sorted Python list

    """
    #duplicate checking system
    same = True # to check if every element in the list are the same
    for l in range(len(lst)):
      if lst[0] != lst[l]:
        same = False
        break

    #If the length of list is below a certain threshold, we use bubble sort
    if len(lst) <= threshold or same == True: #base case: no need to create␣
↪more bucket when 1. only one element in the list or 2. every element in the␣
↪list is the same
      return bubble_sort_count(lst)[0]


    else:  #Other cases: we create buckets
      mx = max(lst)
      mn = min(lst)
      sz = math.ceil((mx -mn)/k)
      buckets = [[] for l in range(k)] #The number of buckets we need

      #we finish the buckets creation
      for i in range(len(lst)):
        b = get_bucket_num(lst[i], mn, mx, sz, k)
        buckets[b].append(lst[i])


      #Run through the every buckets to check if we need to create sub buckets␣
↪through recursion
```

```
    for j in range(len(buckets)):
      buckets[j] = bucket_sort(buckets[j], k)
    # print(buckets)
    lst = [item for sub_bucket in buckets for item in sub_bucket] #flatten␣
↪the bucket lists = add all the buckets togethers

  return lst
```

2.9.4 Question 3d: Algorithmic comparison

Finally, **assess** the computational complexity of the following: - Strictly recursive bucket_sort() - Recursive bucket_sort() with bubble_sort() from a certain threshold - Recursive merge_sort() (you can use your PCW code)

Make this comparison as complete as possible. This should include both an analytical BigO complexity analysis, as well as graphical experimental evidence. Make sure to identify and experimentally study the computational efficiency of these three algorithms with regards to their best and worst-case scenarios. Obtain statistically meaningful metrics to measure computational efficiency (you may consider including the confidence intervals for the run times, as we have discussed in class). Finally, you need to **include a write-up** summarizing the discovered information, comparing and contrasting the algorithms in terms of any metrics you deem important (approximately 200-250 words).

For random input

[ ]:
```
#Merge Sort


def merge(A, p, q, r):

    #Create two lists
    n_1 = q-p+1
    n_2 = r - q
    L = [0]*n_1
    R = [0]*n_2
    for i in range(n_1):
      L[i] = A[p+i-1]
    for j in range(n_2):
      R[j] = A[q+j]

    #Add a sentinal value at the end of both lists
    L.append(float('inf'))
    R.append(float('inf'))

    #Merge the two lists
    i = 0
    j = 0
    for k in range(p-1, r):
```

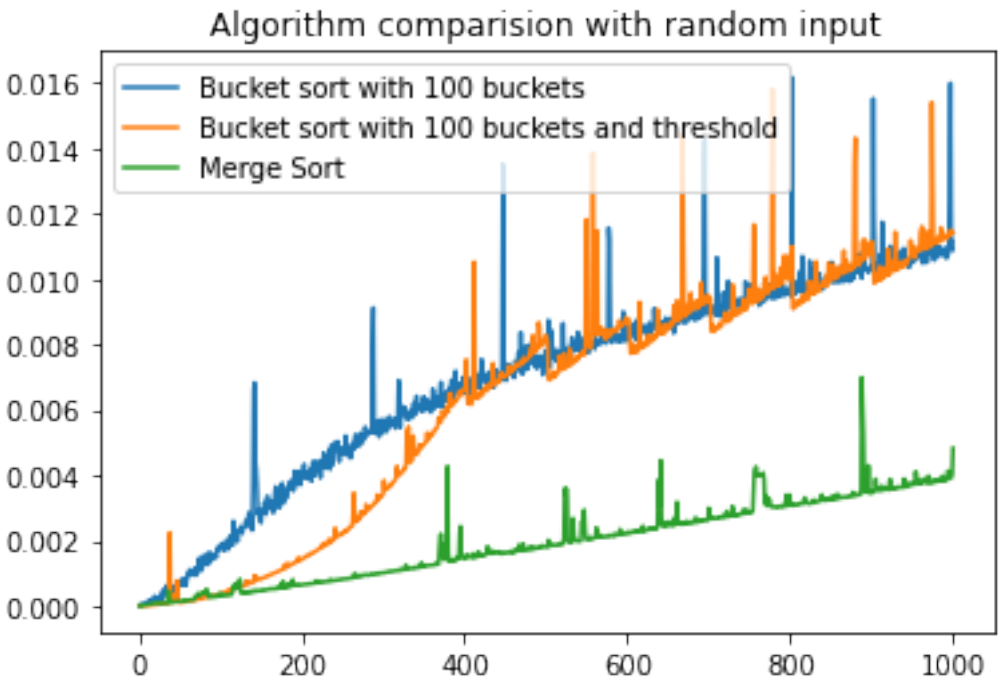
```
      if L[i] <= R[j]:
        A[k] = L[i]
        i += 1
      else:
        A[k] = R[j]
        j += 1
    return A

def merge_sort(A,p,r):

    if p < r:
        q = (p+r)//2
        merge_sort(A,p,q)
        merge_sort(A,q+1,r)
        merge(A,p,q,r)
    return(A)
```

[ ]:
```
#Time for merge sort
merge_time= []
for k in range(len(random_input)):
  start = time.time()
  merge_sort(random_input[k],0, len(random_input[k])-1)
  end = time.time()
  duration = end - start
  merge_time.append(duration)
```

[ ]:
```
#Time for bucket sort with threshold
#Random input
#Experinment for 100 buckets, threshold = 400

bucket_threshold = []
for k in range(len(random_input)):
  start = time.time()
  optimized_bucket_sort(random_input[k],100, 400) #When there are 100 buckets,␣
 ↪the threshold is 400.
  end = time.time()
  duration = end - start
  bucket_threshold.append(duration)
```

[ ]:
```
plt.plot(x, bucket_time_large, label = 'Bucket sort with 100 buckets')
plt.plot(x, bucket_threshold, label = 'Bucket sort with 100 buckets and␣
 ↪threshold')
plt.plot(x, merge_time, label = 'Merge Sort')
plt.legend()
plt.title('Algorithm comparision with random input')
```

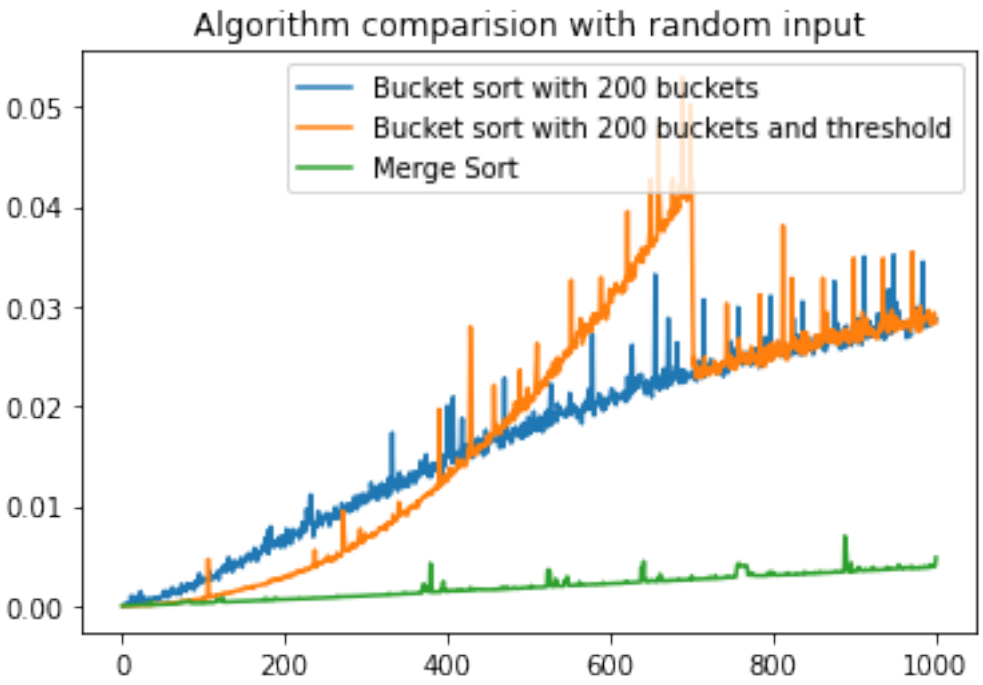
[ ]:
```
Text(0.5, 1.0, 'Algorithm comparision with random input')
```



[ ]:
```
#Time for bucket sort with threshold
#Random input
#Experinment for 200 buckets, threshold = 700

bucket_threshold = []
for k in range(len(random_input)):
  start = time.time()
  optimized_bucket_sort(random_input[k],200, 700) #When there are 100 buckets,␣
 ↪the threshold is 400.
  end = time.time()
  duration = end - start
  bucket_threshold.append(duration)
```

[ ]:
```
plt.plot(x, bucket_time_max, label = 'Bucket sort with 200 buckets')
plt.plot(x, bucket_threshold, label = 'Bucket sort with 200 buckets and␣
 ↪threshold')
plt.plot(x, merge_time, label = 'Merge Sort')
plt.legend()
plt.title('Algorithm comparision with random input')
```

[ ]:
```
Text(0.5, 1.0, 'Algorithm comparision with random input')
```



I take an example from last question: * When there are 100 buckets, the threshold is 400. * When there are 100 buckets, the threshold is 700.

I use a optimized bucket sort with this threshold compare with bucket sort without threshold and merge sort. Here, we can see the performance with threshold is much more better than the one without a threshold. However, neither of them can compare with merge sort.

For sorted input

[ ]:
```
N = 1000
sorted_input = generate_inputs('Sorted',N,1)
```

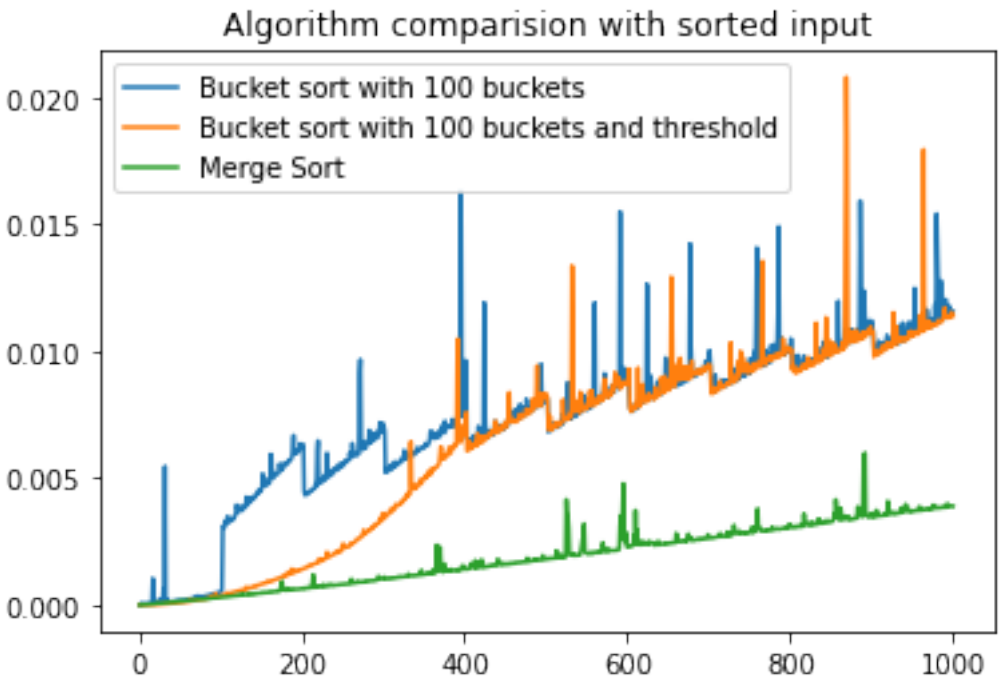
[ ]:
```
#Time for merge sort
merge_time_sorted= []
for k in range(len(sorted_input)):
  start = time.time()
  merge_sort(sorted_input[k],0, len(sorted_input[k])-1)
  end = time.time()
  duration = end - start
  merge_time_sorted.append(duration)
```

[ ]:
```
#Time for bucket sort without threshold
#Sorted input
bucket_time_large_sorted = []
for k in range(len(sorted_input)):
  start = time.time()
  bucket_sort(sorted_input[k],100) #fix the numbers of bucket
  end = time.time()
```

```
  duration = end - start
  bucket_time_large_sorted.append(duration)

#Time for bucket sort with threshold
#Sorted input
#Experinment for 100 buckets, threshold = 400

bucket_threshold_sorted = []
for k in range(len(sorted_input)):
  start = time.time()
  optimized_bucket_sort(sorted_input[k],100, 400) #When there are 100 buckets,␣
 ↪the threshold is 400.
  end = time.time()
  duration = end - start
  bucket_threshold_sorted.append(duration)
```

[ ]:
```
plt.plot(x, bucket_time_large_sorted, label = 'Bucket sort with 100 buckets')
plt.plot(x, bucket_threshold_sorted, label = 'Bucket sort with 100 buckets and␣
 ↪threshold')
plt.plot(x, merge_time_sorted, label = 'Merge Sort')
plt.legend()
plt.title('Algorithm comparision with sorted input')
```

[ ]:
```
Text(0.5, 1.0, 'Algorithm comparision with sorted input')
```

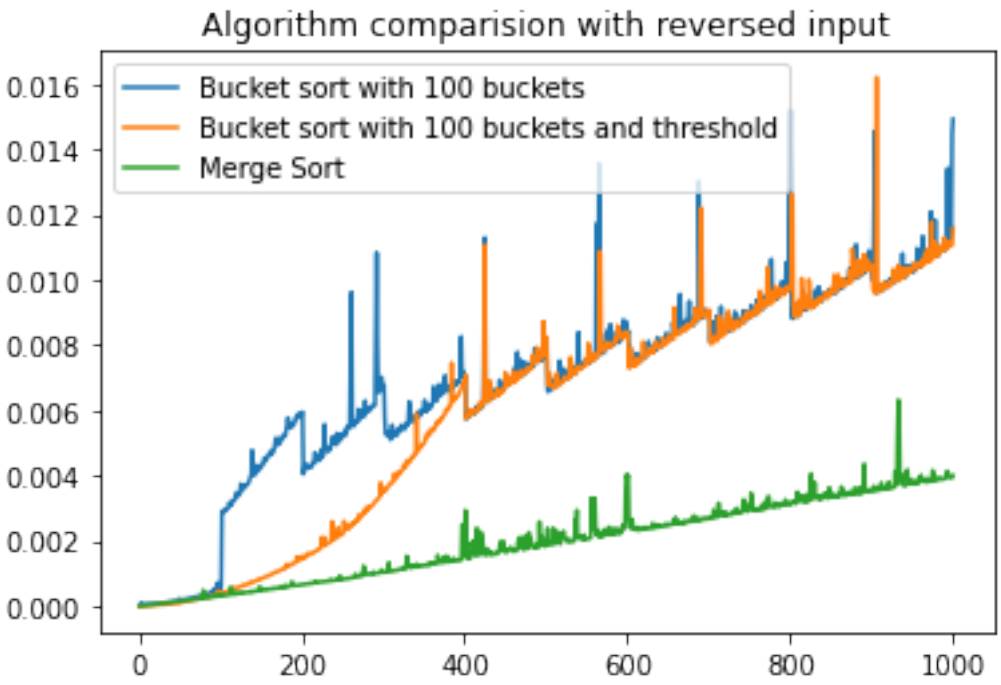


Reversed Input

[ ]:
```
reversed_input = generate_inputs('Reversed',N,1)

#Time for merge sort
merge_time_reversed= []
for k in range(len(reversed_input)):
  start = time.time()
  merge_sort(reversed_input[k],0, len(reversed_input[k])-1)
  end = time.time()
  duration = end - start
  merge_time_reversed.append(duration)

#Time for bucket sort without threshold
#Sorted input
bucket_time_large_reversed = []
for k in range(len(reversed_input)):
  start = time.time()
  bucket_sort(reversed_input[k],100) #fix the numbers of bucket
  end = time.time()
  duration = end - start
  bucket_time_large_reversed.append(duration)

#Time for bucket sort with threshold
#Sorted input
#Experinment for 100 buckets, threshold = 400

bucket_threshold_reversed = []
for k in range(len(reversed_input)):
  start = time.time()
  optimized_bucket_sort(reversed_input[k],100, 400) #When there are 100␣
 ↪buckets, the threshold is 400.
  end = time.time()
  duration = end - start
  bucket_threshold_reversed.append(duration)
```

[ ]:
```
plt.plot(x, bucket_time_large_reversed, label = 'Bucket sort with 100 buckets')
plt.plot(x, bucket_threshold_reversed, label = 'Bucket sort with 100 buckets␣
 ↪and threshold')
plt.plot(x, merge_time_reversed, label = 'Merge Sort')
plt.legend()
plt.title('Algorithm comparision with reversed input')
```

[ ]:
```
Text(0.5, 1.0, 'Algorithm comparision with reversed input')
```



All in all, we can see that 1. merge sort is always better than all kinds of bucket sort. Since the time complexity of merge sort is (), we can guess that time complexity bucket sort has is worst than (). 2. Bucket sort with threshold is better than the one without threshold. 3. Having a sorted or reversed order doesn't really affect bucket sorts performance because they will create the same number of buckets.

The analytical complexity analysis for bucket sort without threshold is T() .

1. It takes O(n) to check duplicates.
2. It takes O(k) to create k buckets.
3. It takes O(n) to find element's bucket number and another O(n) to put it inside the bucket.
4. Merge k buckets take O(k)

The total runtime for recursive part is 2O(n+k) and to check the duplicates is O(n), which we can express as O(n+k). Since every time we run the recursion, we will create k buckets, but reduce the length of list to $\frac{n}{k}$. The analytical complexity is $T(n) = kT(\frac{n}{k}) + O(n+k)$

Using the tree, we see that the recurrance relation is $O(nlog_k n + kn)$

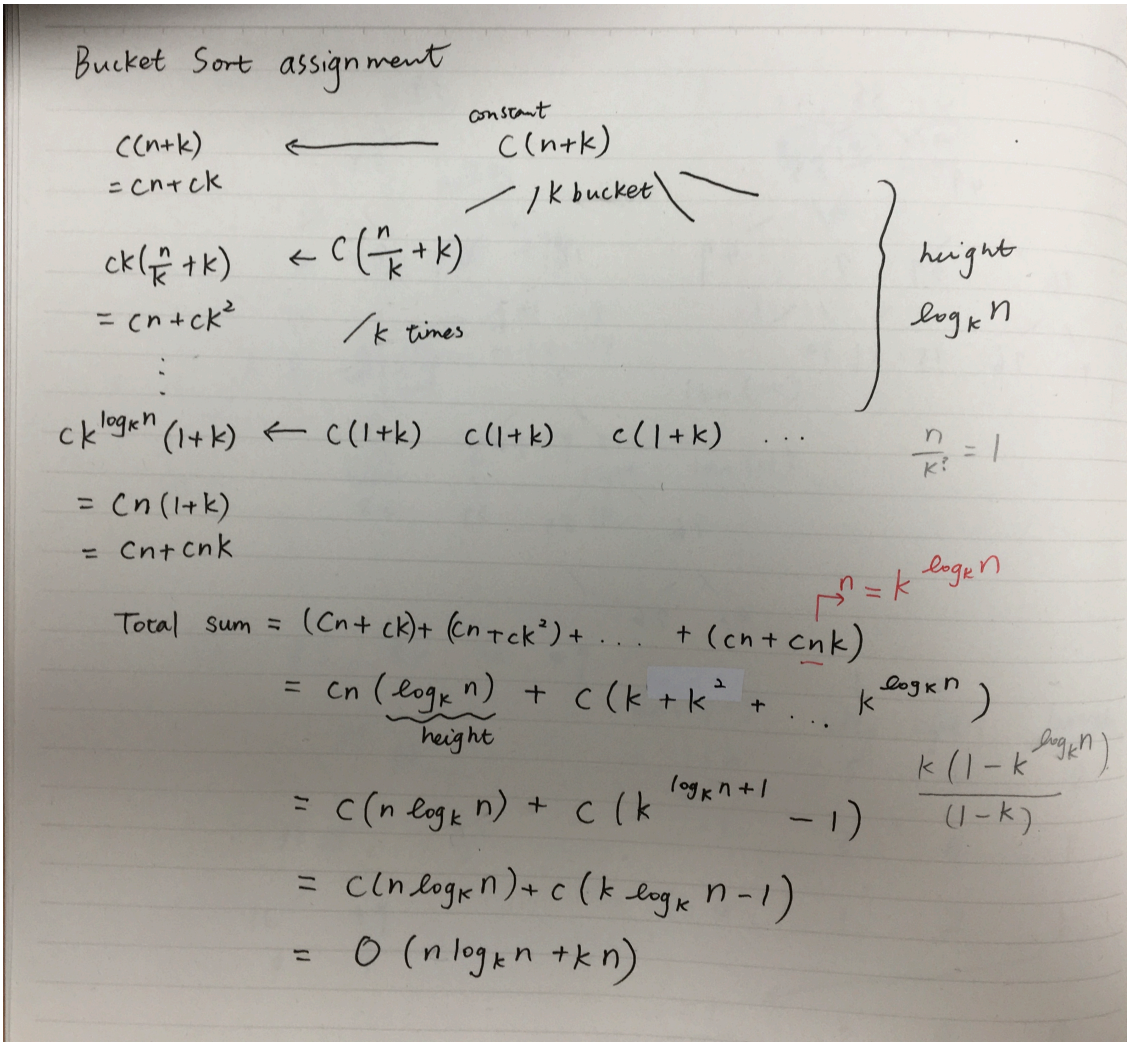
(Resource: Ha's explanation)

[ ]:
```
upload_2 = files.upload()
```

```
<IPython.core.display.HTML object>


Saving IMG_8626.jpg to IMG_8626.jpg
```

[ ]:
```
Image('IMG_8626.jpg')
```

[ ]:



Reflection: Things I still don't understand well

1. Bucket sort: relationship between k and len(list).
2. Recurrence relation of bucket sort
3. Bucket sort worst case
4. Hanoi recursive algorihtm

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

[ ]:
```
!cp "./drive/My Drive/Colab Notebooks/Esther_Assignment 1 - Algorithm Design␣
↪and Sorting.ipynb" ./
```