

Assignment – 2

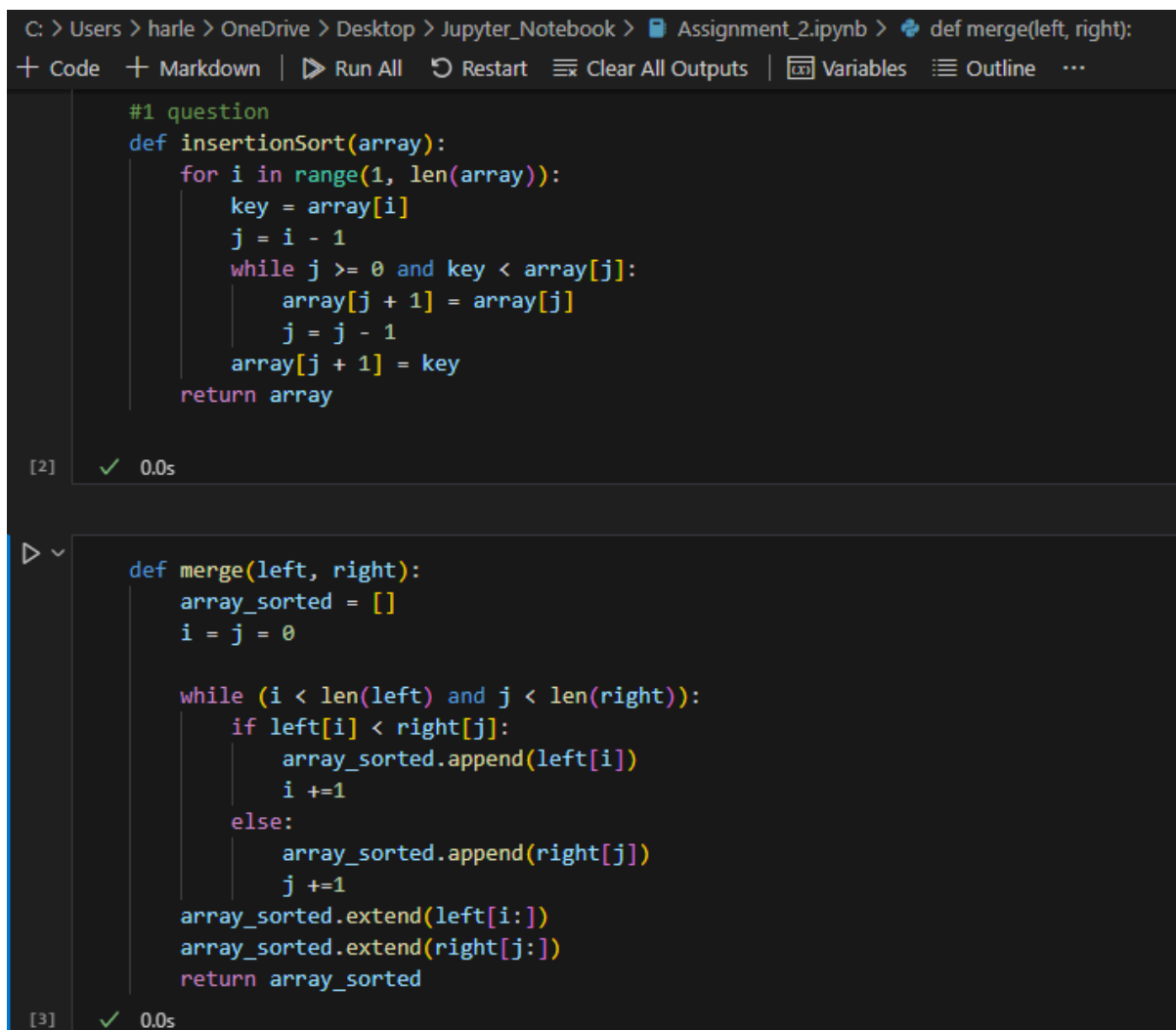
Analysis of Algorithms

Ayyappa Reddy – R11888944

Task 1: Implement the provided algorithm in a language of your choice, also providing your own implementations of Insertion Sort and Merge. Test your implementation thoroughly for correctness (e.g., use your programming language's built-in sorting algorithm and make sure it agrees with your algorithm on every input).

Deliverable 1: Your implementation's code and your verification of correctness. (35 points)

Solution: Now let's see the implementation of codes for both Insertion sort and Merge sort Algorithms.



```
C: > Users > harle > OneDrive > Desktop > Jupyter_Notebook > Assignment_2.ipynb > def merge(left, right):
+ Code + Markdown | Run All Restart Clear All Outputs | Variables Outline ...

#1 question
def insertionSort(array):
    for i in range(1, len(array)):
        key = array[i]
        j = i - 1
        while j >= 0 and key < array[j]:
            array[j + 1] = array[j]
            j = j - 1
        array[j + 1] = key
    return array

[2] ✓ 0.0s

def merge(left, right):
    array_sorted = []
    i = j = 0

    while (i < len(left) and j < len(right)):
        if left[i] < right[j]:
            array_sorted.append(left[i])
            i += 1
        else:
            array_sorted.append(right[j])
            j += 1
    array_sorted.extend(left[i:])
    array_sorted.extend(right[j:])
    return array_sorted

[3] ✓ 0.0s
```

Fig.1 Insertion and Merge Sort Algorithms

So, from above that we completed with the implementation of codes and now we need to implement the hybrid sort by using above Sorting Algorithms and have to test the implementation thoroughly for correctness. And, we have to perform the sorting for the input array of size 6.

```
C: > Users > harle > OneDrive > Desktop > Jupyter_Notebook > Assignment_2.ipynb > #now lets test merge sort and insertion sort
+ Code + Markdown | Run All Restart Clear All Outputs Variables Outline ...

def hybridSort(arr, k):
    if len(arr) < k:
        return insertionSort(arr)
    else:
        mid=len(arr)//2
        left=arr[:mid]
        right=arr[mid:]
        return merge(hybridSort(left,k),hybridSort(right,k))

[4] ✓ 0.0s

#now lets test merge sort and insertion sort
print(insertionSort([6,5,4,3,2,1]))
print(merge([1,3,5],[4,4,5]))
print(hybridSort([9,8,7,6,5,4,3,2,1],3))

[13] ✓ 0.0s

... [1, 2, 3, 4, 5, 6]
    [1, 3, 4, 4, 5, 5]
    [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Fig.2 Hybrid Sort and test for correctness

From the above implementation we can clearly say from the output the array is sorted in ascending order.

And finally, Now let's check with the array size of 1000 so, we create a random array of size 1000 and we have to see that with the Hybrid sort function to sort the algorithm and if the both the arrays are equal it results in Hybrid is working fine. or else it will result in developed sorting algorithm does not provide correct input. Here we are using the assert keyword for checking or verifying the correctness.

```
C: > Users > harle > OneDrive > Desktop > Jupyter_Notebook > Assignment_2.ipynb > import random
+ Code + Markdown | Run All Restart Clear All Outputs Variables Outline ...

import random
my_list = [random.randint(0, 5000) for i in range(1000)]
sortedList = hybridSort(my_list,3)
try:
    assert sorted(my_list) == sortedList
except AssertionError as e:
    print("Developed sorting algorithm doesnot provide correct input")
else:
    print("Hybrid sort is working fine")

[8] ✓ 0.0s

... Hybrid sort is working fine
```

Fig3. Hybrid sort working

From the above implementation we can say that the hybrid sort is also working fine.

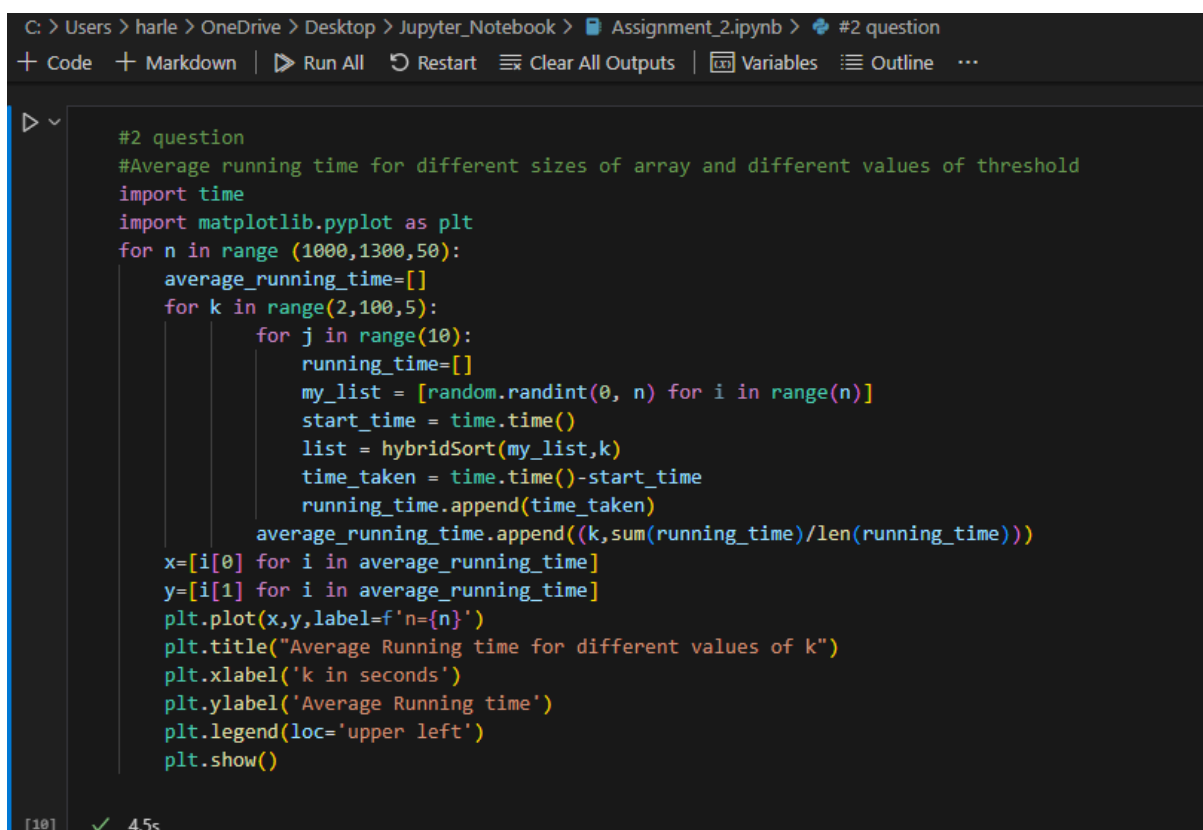
Task 2: Generate a plot (or plots) which depicts your algorithm's average running time as a function of K and n on input arrays that you generate randomly (i.e., input arrays for this deliverable should not be pre-sorted in any way). Your "mental model" here should be that K will be relatively small (under 100) and n will be relatively large (as large as your implementation can handle in a reasonable amount of time). As you are running the tests, select values of K and n which showcase interesting phenomena. Finally, a note on "average running time": for a specific value of K and n, a single run of your algorithm is not enough to tell you anything about its average running time. To compute an average for a specific value of n, you'll need to generate several random arrays of length n, feed them all to the algorithm for your fixed value of K, then compute the average of the running times you obtained.

Deliverable 2: A plot showing the average run time of your algorithm as a function of K, with a separate trace for at least 5 representative values of n. (20 points)

Solution: Now let's see the implementation first for the average running time for different sizes of array and different values of threshold.

From the below implementation we can calculate the running time of Hybrid Sort multiple times and finds the average of these calculated running time instances.

To plot the instances for the average running time of different values of K and for different array sizes, we used one loop for array size and one more for threshold values. We also have another loop which calculates the average running time so, the below mentioned is the function that which calculate the average running time.



```

C: > Users > harle > OneDrive > Desktop > Jupyter_Notebook > Assignment_2.ipynb > #2 question
+ Code + Markdown | ▶ Run All ⏮ Restart ⌵ Clear All Outputs | 📄 Variables 📖 Outline ...

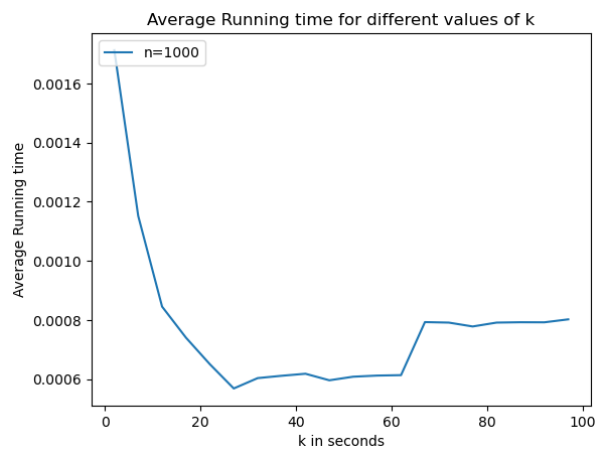
▶ #2 question
#Average running time for different sizes of array and different values of threshold
import time
import matplotlib.pyplot as plt
for n in range(1000,1300,50):
    average_running_time=[]
    for k in range(2,100,5):
        for j in range(10):
            running_time=[]
            my_list = [random.randint(0, n) for i in range(n)]
            start_time = time.time()
            list = hybridSort(my_list,k)
            time_taken = time.time()-start_time
            running_time.append(time_taken)
        average_running_time.append((k,sum(running_time)/len(running_time)))
x=[i[0] for i in average_running_time]
y=[i[1] for i in average_running_time]
plt.plot(x,y,label=f'n={n}')
plt.title("Average Running time for different values of k")
plt.xlabel('k in seconds')
plt.ylabel('Average Running time')
plt.legend(loc='upper left')
plt.show()

[10] ✓ 4.5s

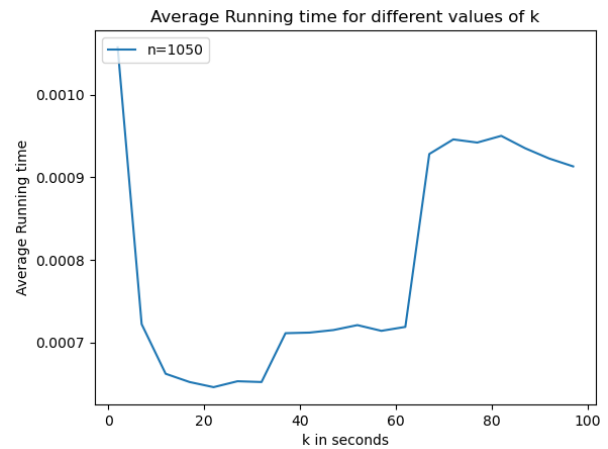
```

Fig.4 Implementation of Average Running time

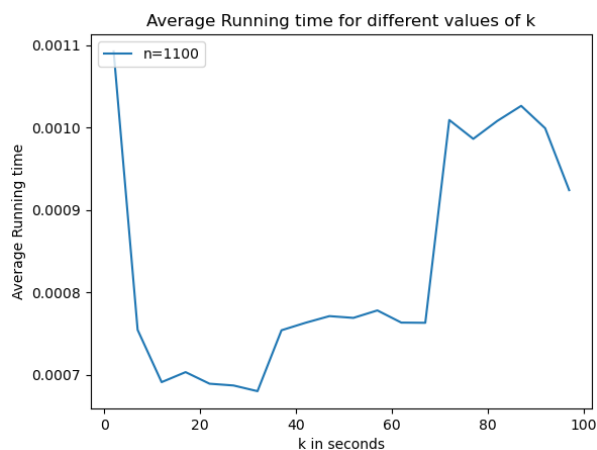
Now let's plot the graphs for all different thresholds and their corresponding running times.



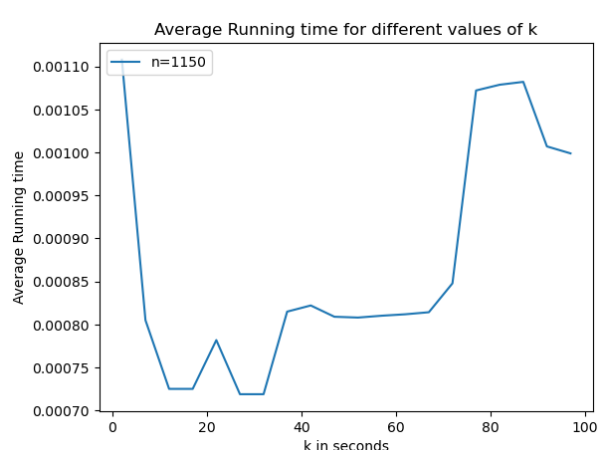
Size of Array: 1000



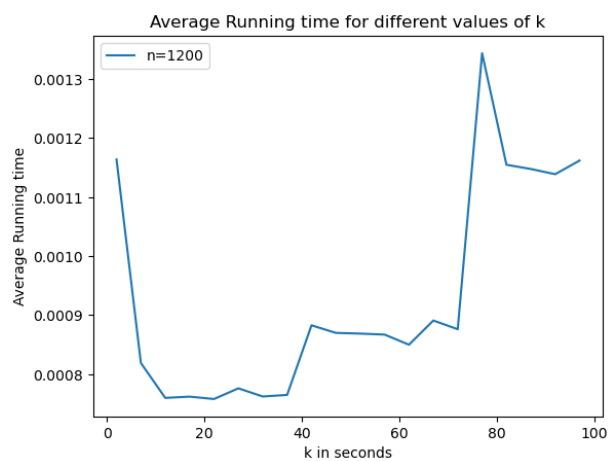
Size of Array: 1050



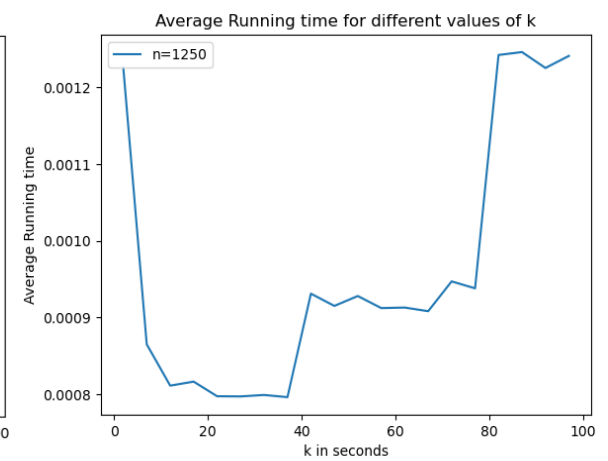
Size of Array: 1100



Size of Array: 1150



Size of Array: 1200



Size of Array: 1250

Task 3: Identify the optimal value of K as a function of array length n. This should be informed by (or answered by) your analysis in the previous step. How much does the best choice of K depend on n? Note: depending on your implementation, it could happen that this optimal K will be the same for all n. If that's the case, report it anyway. Try to understand any relationship (or lack of relationship) that you find.

Deliverable 3: A plot showing the optimal value of K as a function of array length n. Explain why you think the relationship between n and optimal K is the way that it is. (20 points)

Solution:

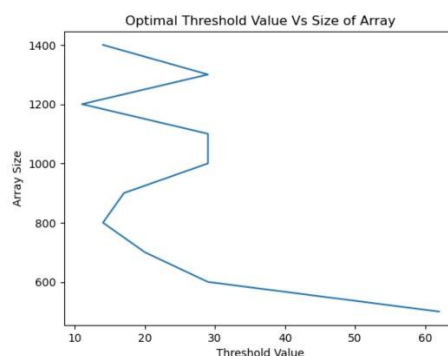
Let's implement for optimal value of K iterations.

```
C: > Users > harle > OneDrive > Desktop > Jupyter_Notebook > Assignment_2.ipynb > #3 question
+ Code + Markdown | Run All | Clear All Outputs | Outline ...

#3 question
def optimal_k():
    k=[]
    array_size=[]
    for n in range(500,5000,1000):
        avg_running_time=[]
        for threshold in range(2,100,3):
            for j in range(10):
                running_time=[]
                my_list = [random.randint(0, n) for i in range(n)]
                start_time = time.time()
                list = hybrid_sort(my_list,threshold)
                time_taken = time.time()-start_time
                running_time.append(time_taken)
            avg_running_time.append((threshold,sum(running_time)*1000/len(running_time)))
        x=[i[0] for i in avg_running_time]
        y=[i[1] for i in avg_running_time]
        i = np.argmin(y)
        min_value = x[i]
        k.append(min_value)
        array_size.append(n)

    plt.plot(k,array_size)
    plt.xlabel('Threshold Value')
    plt.ylabel('Array Size')
    plt.title("Optimal Threshold Value Vs Size of Array")
    plt.show();
```

Fig5. Implementation for Optimal value of K iterations.



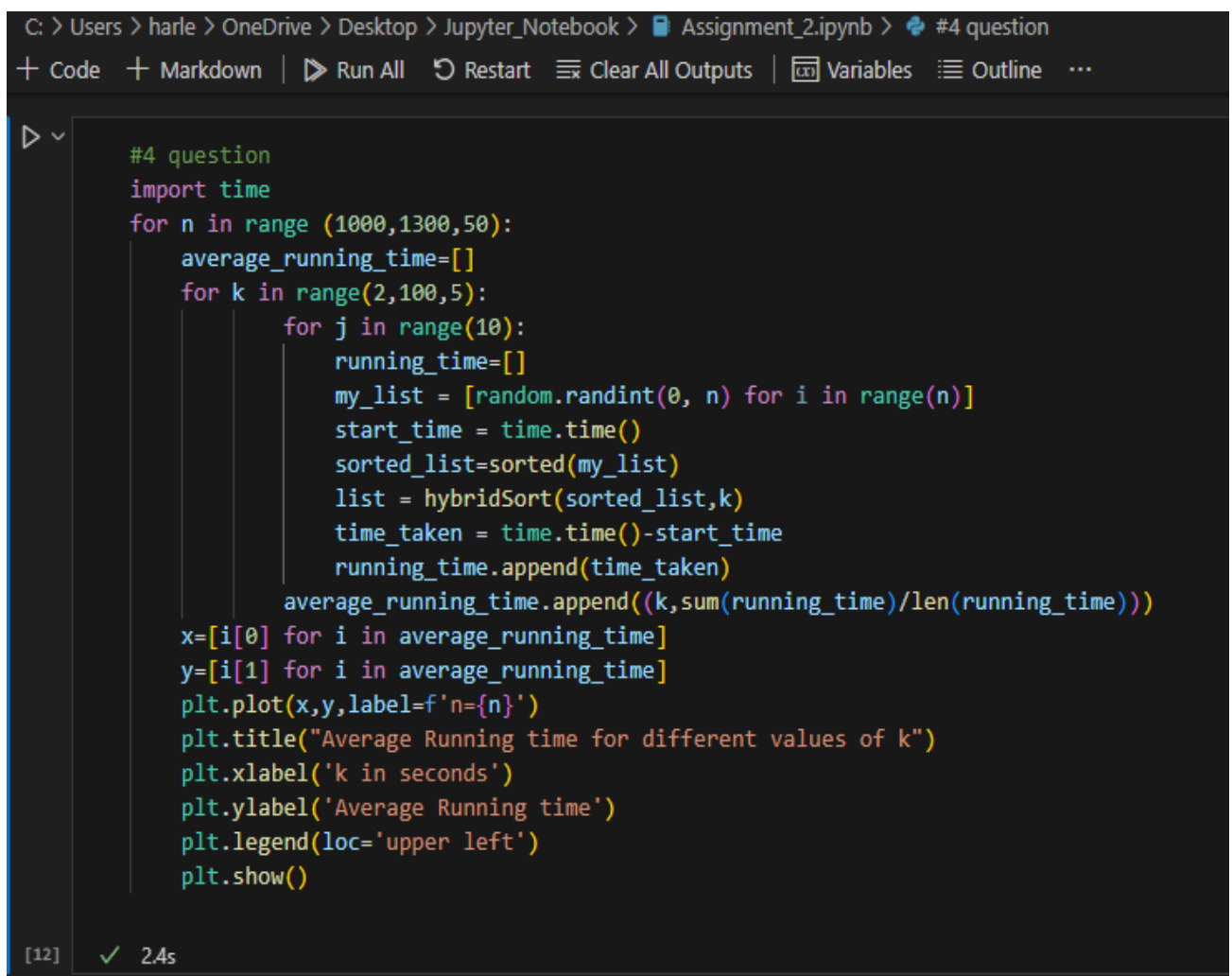
Here the k values are mostly in the range of (15,30) for most of the array sizes. We can say that optimal value k is independent of the array size for most of the cases and will be in the limited range.

Task 4: Repeat deliverables 2 and 3; however, this time, test your algorithm only on sorted arrays. How do the results differ from what you reported in Deliverables 2 and 3? Explain these differences to the best of your ability.

Deliverable 4: Your observations and findings from Task 4. (25 points)

Solution: Here we use the above used hybrid sorted array and calculates the running time of hybrid sort on the sorted array.

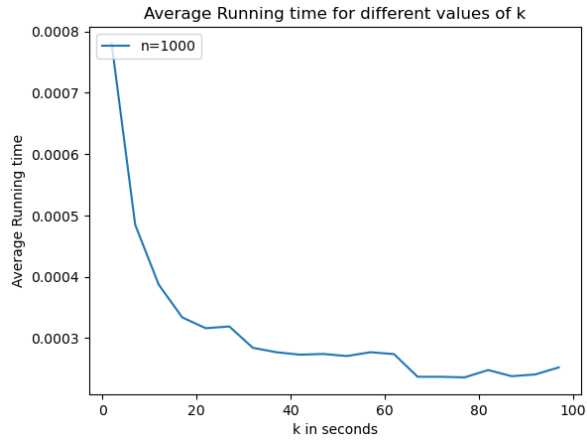
To perform this activity, we will use the code that which is already implemented in the Task 2 and before entering the list we sort the list by using `sorted_list ()` method of python and then we send this `sorted_list` as an input to hybrid sort.



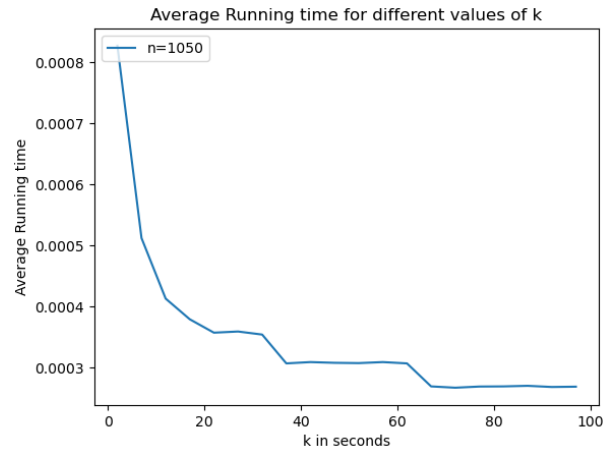
```
#4 question
import time
for n in range (1000,1300,50):
    average_running_time=[]
    for k in range(2,100,5):
        for j in range(10):
            running_time=[]
            my_list = [random.randint(0, n) for i in range(n)]
            start_time = time.time()
            sorted_list=sorted(my_list)
            list = hybridSort(sorted_list,k)
            time_taken = time.time()-start_time
            running_time.append(time_taken)
        average_running_time.append((k,sum(running_time)/len(running_time)))
x=[i[0] for i in average_running_time]
y=[i[1] for i in average_running_time]
plt.plot(x,y,label=f'n={n}')
plt.title("Average Running time for different values of k")
plt.xlabel('k in seconds')
plt.ylabel('Average Running time')
plt.legend(loc='upper left')
plt.show()
```

[12] ✓ 2.4s

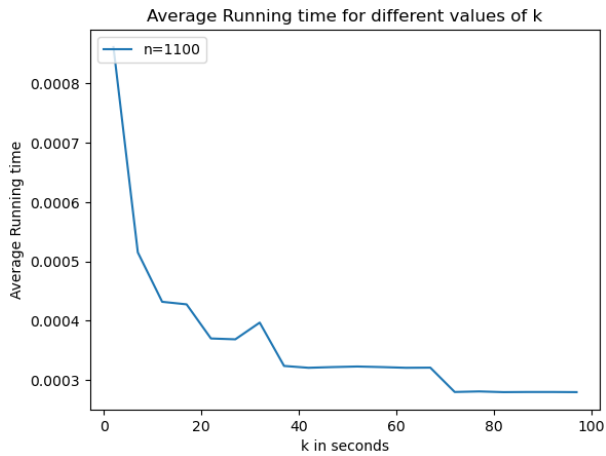
Fig.6 Testing only on sorted arrays.



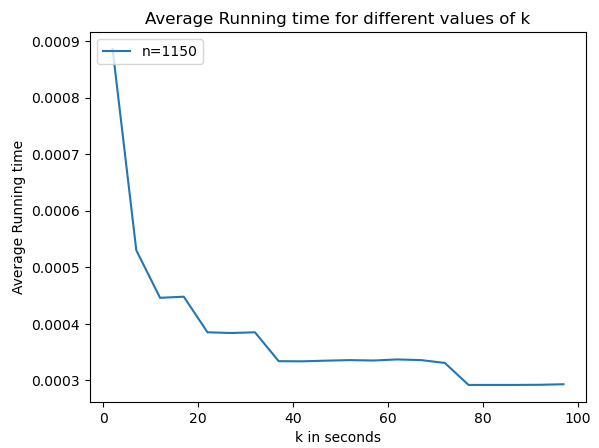
Size of Array: 1000



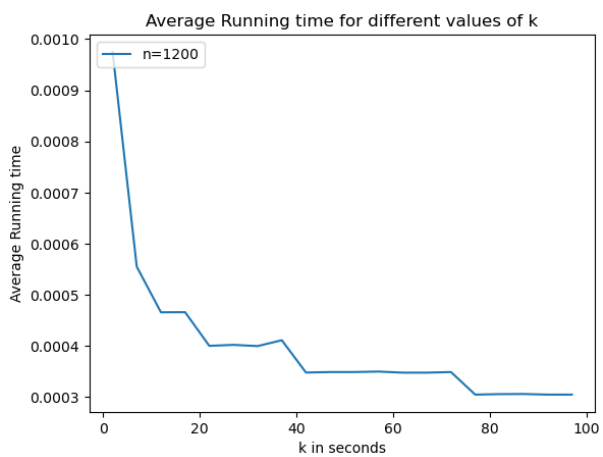
Size of Array: 1050



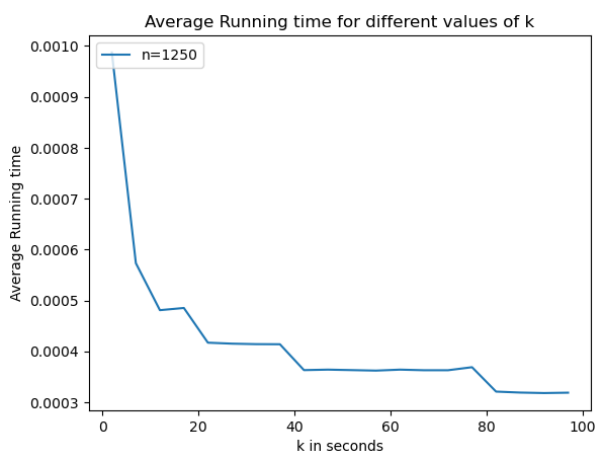
Size of Array: 1100



Size of Array: 1150



Size of Array: 1200



Size of Array: 1250