Module 2

HW 1

For all of the programs below, write the program as efficiently as you can. Do not use any built-in libraries for the array. You may use referenced source code from the internet. You may use **the built-in uniform random number generator** and **assume it operates in Θ(1).** For each problem:

- From an analysis of your code, give a function representing the running time of your code. Give a tight asymptotic bound for that function.
- Run your code for various values of n and time it,
    - Create a chart showing the running times for various values of “n”,
    - Create a graph of the running times vs various values of “n”. Use a linear scale on the axes.
    - Describe how the running times support your analysis of the asymptotic running times.
- Give an estimation of how long it would take for n = 1 trillion
- Include your source code with your submission.

1. [25 pts] Write a program that takes a value “n” as input and prints “Hello, World” n times.

Source of the program code:

```
def print_hello_world(n):
  for i in range(n):
    print("Hello, World")


n = int(input("Enter a value for n: "))
print_hello_world(n)
```

A function representing time:

- f(n) = c * n.
- f(n) is Θ (n)

Here is the running time chart based on the output of the running time:

n	Time(seconds)
10000	0.02324986457824707
20000	0.04589986801147461
30000	0.06679844856262207

40000	0.1156919002532959
50000	0.11989736557006836
60000	0.16595005989074707

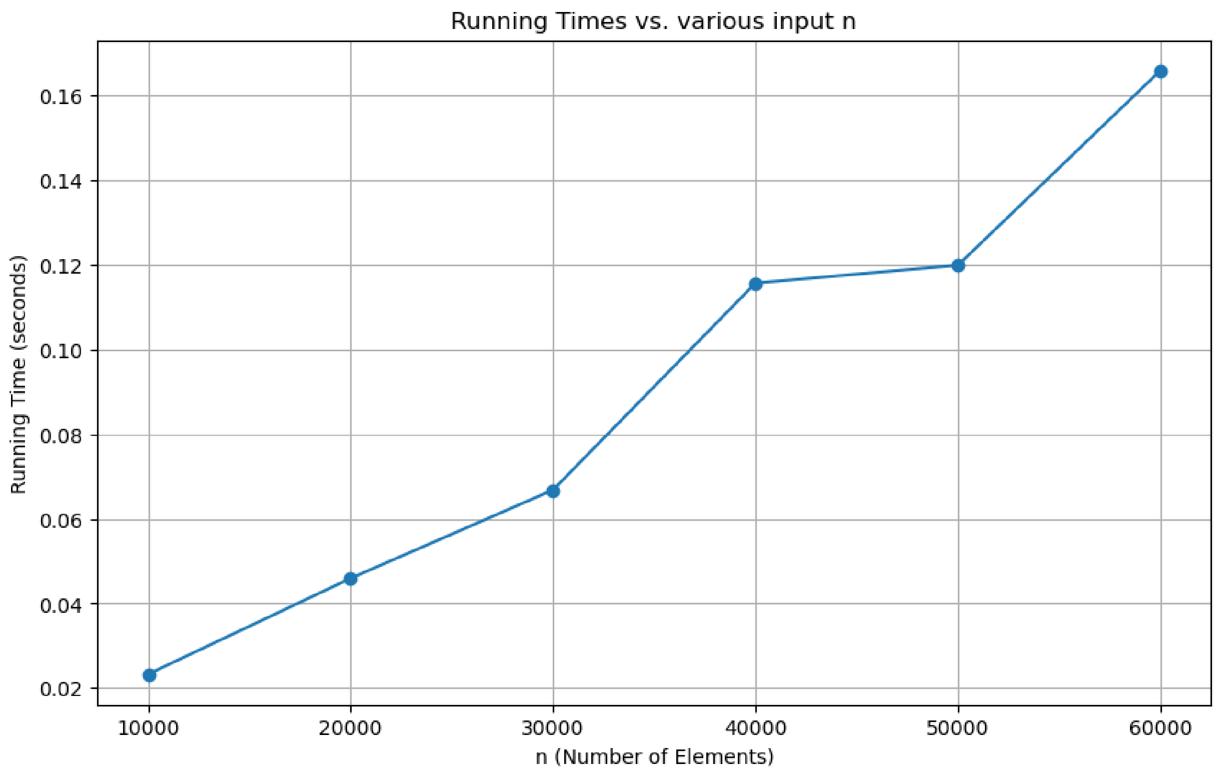
Details of the code and output for running times:

```
import time

def print_hello_world(n):

   for i in range(n):

       print("Hello, World")


def measure_running_time(n):

   """

   Measures the average time per iteration of the print_hello_world funtion.

   """

   start_time = time.time()

   print_hello_world(n)

   end_time = time.time()

   running_time = end_time - start_time


   return running_time


# n = int(input("Enter a value for n: "))

n_values = [10000, 20000, 30000, 40000, 50000, 60000]

running_times = []


# Measuring the execution time for each n

for n in n_values:

   running_time = measure_running_time(n)

   running_times.append(running_time)
```

```
for n, y in zip(n_values, running_times):
    print(f"When input value n is {n}: the running time is {y}")

# When input value n is 10000: the running time is 0.02324986457824707
# When input value n is 20000: the running time is 0.04589986801147461
# When input value n is 30000: the running time is 0.06679844856262207
# When input value n is 40000: the running time is 0.1156919002532959
# When input value n is 50000: the running time is 0.11989736557006836
# When input value n is 60000: the running time is 0.16595005989074707
```

Here is the running time graph:

```
import matplotlib.pyplot as plt
plt.figure(figsize=(10, 6))
plt.plot(n_values, running_times, marker='o')
plt.title('Running Times vs. various input n')
plt.xlabel('n (Number of Elements)')
plt.ylabel('Running Time (seconds)')
plt.grid(True)

plt.xticks(n_values)
plt.xscale('linear') # Setting x-axis to linear scale
plt.yscale('linear')  # Setting y-axis to linear scale
plt.show()
```



Give an estimation of how long it would take for n = 1 trillion:

If 50,000 is 0.12 seconds, then 1,000,000 will be 20 times as large as the time so it would be 2.4 seconds.

1,000,000,000,000 would be 1,000,000 times as large, so the time would be 2.4× 1,000,000 = 2,400,000 seconds equal to how many seconds

2. [25 pts] Write a program that takes a value "n" as input; produces "n" random numbers with a uniform distribution between 1 and n and places them in an array in sorted order. Place them in the array in order, do not sort the array after placing them there.

Source of the program code:

```
import random
def binary_search(arr, val, start, end):
  # This function finds the index where 'val' should be inserted in the sorted array 'arr'.
  if start == end:
    return start
    mid = (start + end) // 2
    if arr[mid] < val:
        return binary_search(arr, val, mid +1, end)
    else:
        return binary_search(arr, val, start, mid)

def insert_sorted(arr, val):
    # This function inserts "val" in the sorted array 'arr'.
    i = binary_search(arr, val, 0, len(arr))
    arr.insert(i, val)


def generate_sorted_randoms(n):
    arr = []
    for _ in range(n):
        num = random.uniform(1, n)
        insert_sorted(arr, num)
    return arr
```

A function representing time:

- f(n) = logn + n + n^2
- f(n) is Θ (n^2)

  binary_search: O(logn)
  insert_sorted: O(n) (due to the list insertion operation)
  The time complexity of the generate_sorted_randoms function is O(n^2). This is because the insert_sorted function is called n times, and each call takes O(n) time to insert the element into the sorted array using binary search. Therefore, the total time complexity of the function is O(n^2).

Here is the running time chart based on the output of the running time:

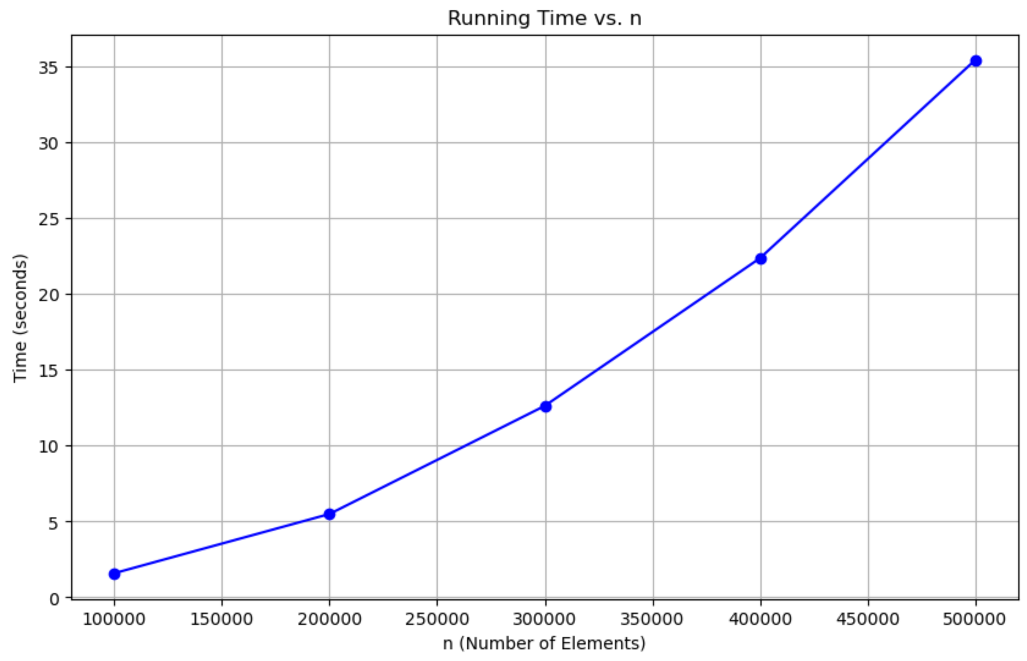
n	**Time(seconds)**
100000	1.5922186374664307
200000	5.497781276702881
300000	12.624443769454956
400000	22.361177444458008
500000	35.424288511276245

Details of the code and output for running times:

```
import time

def measure_running_time(n):
        """
        Measures the average time per iteration of the print_hello_world funtion.
        """
        start_time = time.time()
        generate_sorted_randoms(n)
        end_time = time.time()
        running_time = end_time - start_time

        return running_time

    n_values = [100000, 200000, 300000, 400000, 500000]
    running_times = []

    # Measuring the execution time for each n
    for n in n_values:
        running_time = measure_running_time(n)
        running_times.append(running_time)
    for n, y in zip(n_values, running_times):
        print(f"When input value n is {n}: the running time is {y}")

    # When input value n is 100000: the running time is 1.5922186374664307
    # When input value n is 200000: the running time is 5.497781276702881
    # When input value n is 300000: the running time is 12.624443769454956
    # When input value n is 400000: the running time is 22.361177444458008
    # When input value n is 500000: the running time is 35.424288511276245
```

Here is the running time graph:

```
# Plotting the results

plt.figure(figsize=(10, 6))

plt.title('Running Time vs. n')

plt.plot(n_values, running_times, marker='o', linestyle='-', color='b')

plt.xlabel('n (Number of Elements)')

plt.ylabel('Time (seconds)')

plt.xticks(n_values)

plt.xscale('linear') # Setting x-axis to linear scale

plt.grid(True)

plt.show()
```



Give an estimation of how long it would take for n = 1 trillion:

If 100,000 take 2 seconds, then 1,000,000,000,000 would be 10,000,000 times as large.

f(n) is Θ (n^2), so 1,000,000,000,000 will be $(2 \times 10{,}000{,}000)^2$ = 4e+14 seconds

3. [25 pts] Write a program that takes a value "n" as input; produces "n" random numbers with a uniform distribution between 1 and 3, places them in an array and counts how many of each number is produced.

Source of the program code:

```
import random
from collections import Counter

def generate_and_count(n):
    numbers = []
    for _ in range(n):
        # Generate a random number between 1 and 3 (inclusive)
        num = random.uniform(1, 3)
        # Round the number to the nearest whole number since uniform() generates a floating-
point number
        num = round(num)
        numbers.append(num)

    # Count the occurrences of each number
    count = Counter(numbers)
    return count

# Get input from the user
n = int(input("Enter a number: "))
count = generate_and_count(n)

# Print the counts
for number, frequency in count.items():
    print(f"Number {number}: {frequency} times")
```

A function representing time:

- f(n) = O(n) + O(n) = O(2n)
- f(n) is Θ(n)

The function consists of two main parts: generation 'n' random numbers and then counting their occurrences:

1. *generation 'n' random numbers: the loop runs 'n' times. The operations inside the loop, generating a random number, rounding it, and appending it to a list all have constant time complexity, O(1). The time complexity is O(n)*
2. *Counting Occurrences with Counter: The time complexity of counting the occurrences of elements in a list of length n is O(n)*

Overall Time Complexity is O(n) + O(n) = O(2n) which simplifies to Θ(n) as the asymptotic bound

Here is the running time chart based on the output of the running time:

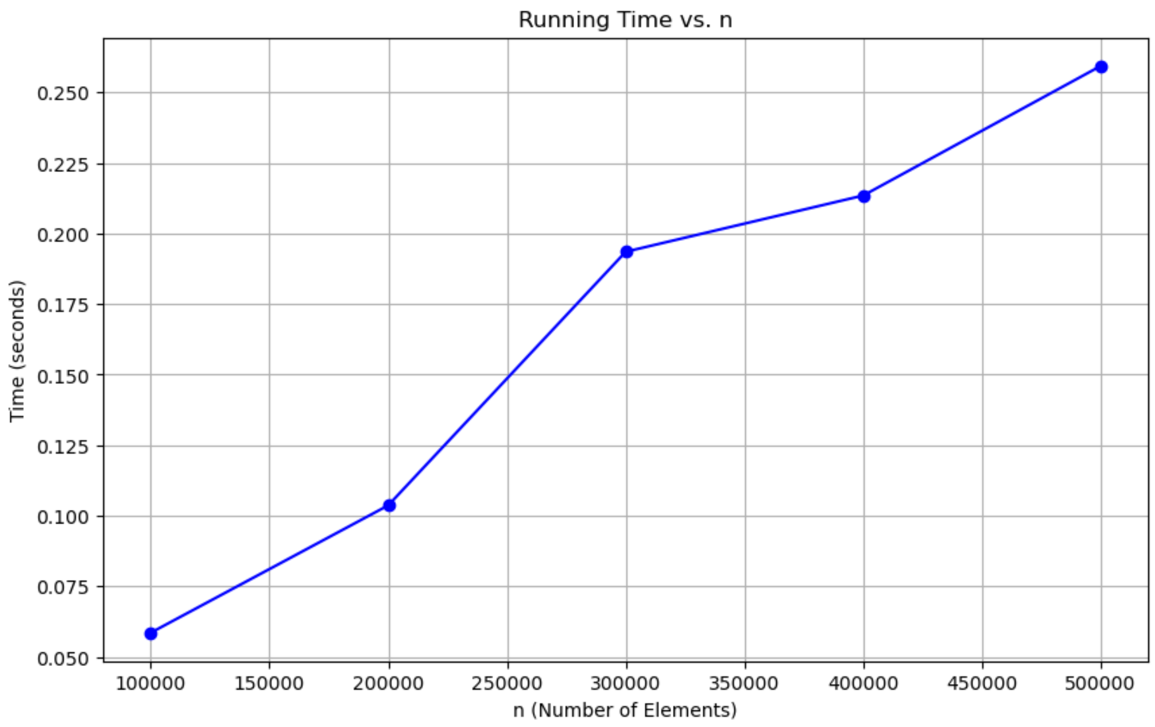
n	**Time(seconds)**
100000	0.058580636978149414
200000	0.10369420051574707
300000	0.1935122013092041
400000	0.21342968940734863
500000	0.2593114376068115

Details of the code and output for running times:

```
import time

def measure_running_time(n):
    """
    Measures the average time per iteration of the print_hello_world funtion.
    """
    start_time = time.time()
    generate_and_count(n)
    end_time = time.time()
    running_time = end_time - start_time

    return running_time

n_values = [100000, 200000, 300000, 400000, 500000]
running_times = []
# Measuring the execution time for each n
for n in n_values:
    running_time = measure_running_time(n)
    running_times.append(running_time)

for n, y in zip(n_values, running_times):
    print(f"When input value n is {n}: the running time is {y}")

# When input value n is 100000: the running time is 0.058580636978149414
# When input value n is 200000: the running time is 0.10369420051574707
# When input value n is 300000: the running time is 0.1935122013092041
# When input value n is 400000: the running time is 0.21342968940734863
# When input value n is 500000: the running time is 0.2593114376068115
```

Here is the running time graph:

```
# Plotting the results

plt.figure(figsize=(10, 6))

plt.title('Running Time vs. n')

plt.plot(n_values, running_times, marker='o', linestyle='-', color='b')

plt.xlabel('n (Number of Elements)')

plt.ylabel('Time (seconds)')

plt.xticks(n_values)

plt.xscale('linear') # Setting x-axis to linear scale

plt.grid(True)

plt.show()
```



Give an estimation of how long it would take for n = 1 trillion

From my input data, we see the following running times for different values of n:

- For n=100,000, time is approximately 0.0586 seconds.
- For n=500,000, time is approximately 0.2593 seconds.

Since 1 trillion is 2,000,000 times larger than 500,000, the time it would take is expected to be 0.2593×2,000,000 seconds, in a very rough estimation.

4. [25 pts] Sort the array created in Problem #3 using the most efficient way you know. You may use source code from the internet for your sort if you wish, but be sure to reference it. You may not use a built-in sorting library call. Run for n as large as you can without crashing or until it takes at least 30 seconds.

Source of the program code:

```
import random
import time
from collections import Counter
import matplotlib.pyplot as plt

def quicksort(arr):
  if len(arr) <= 1:
    return arr
  else:
    pivot = arr[len(arr) // 2] # use the middle element of the array.
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)


def generate_and_count(n):
  numbers = []
  for _ in range(n):
    num = random.uniform(1, 3)
    num = round(num)
    numbers.append(num)

  sorted_numbers = quicksort(numbers)
  count = Counter(sorted_numbers)
  return count, sorted_numbers
```

A function representing time:

- f(n) = O(nlogn) + O(n)
- f(n) is Θ(nlogn)

The function consists of two main parts generating a list of random numbers followed by sorting them using the Quicksort algorithm, and then counting the occurrences of each number using Python's Counter class.

1. *Quicksort Algorithm: The best and average case time complexity of Quicksort is O(nlogn) when the pivot divides the array into two roughly equal parts, leading to a balanced division at each level of recursion. The worst-case time complexity is O(n^2) when the pivot is divided in an unbalanced partition. In my algorithm, the pivot is chosen as the middle element of the array, avoiding hitting the worst case so the time complexity is O(nlogn).*
2. *Generating Random Numbers and Counting: Generating n Random Numbers will have a time complexity of O(n). Counting will also have a time complexity of O(n)*

Given that the Quicksort operation dominates the computational complexity, the overall asymptotic bound will be Θ(nlogn).

Here is the running time chart based on the output of the running time:

n	Time(seconds)
1000000	0.9249532222747803
2000000	1.6297001838684082
3000000	2.5987300872802734
4000000	3.3289573192596436

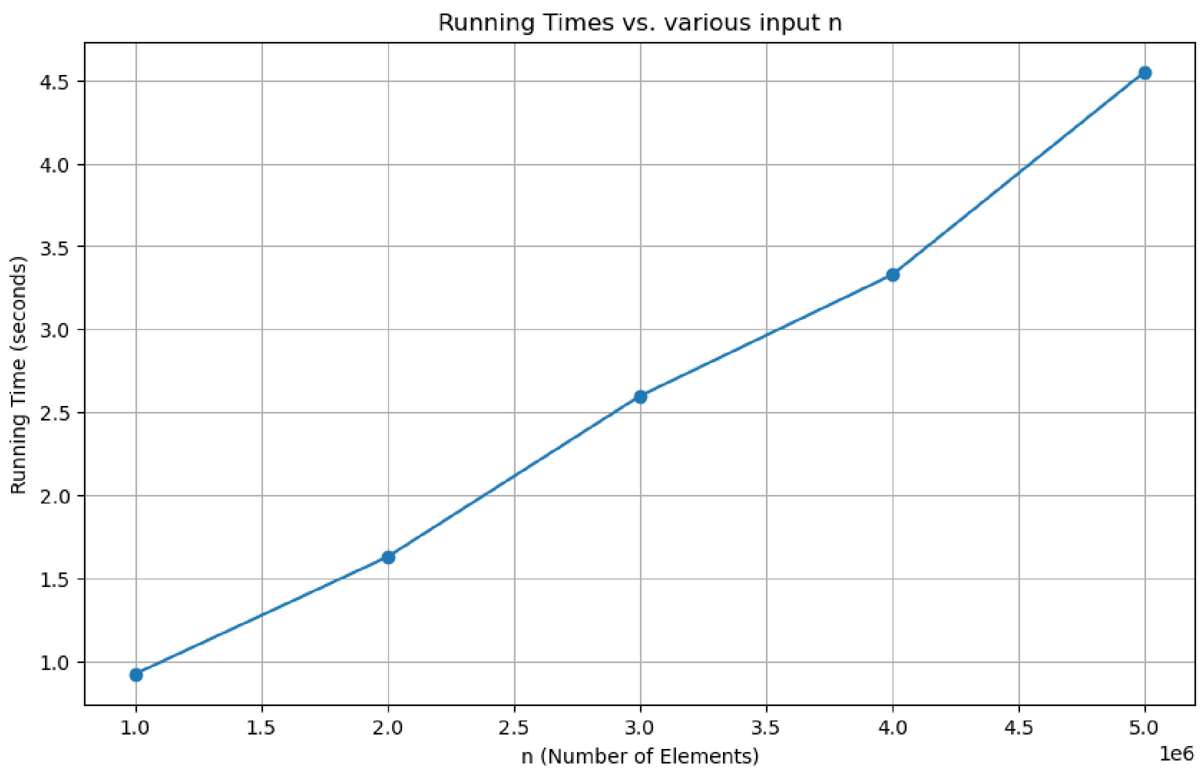
5000000	4.551220178604126

Details of the code and output for running times:

```
import time

def measure_running_time(n):

    """

    Measures the average time per iteration of the print_hello_world funtion.

    """

    start_time = time.time()

    count, sorted_numbers = generate_and_count(n)

    end_time = time.time()

    running_time = end_time - start_time


    return running_time


n_values = [1000000, 2000000, 3000000, 4000000, 5000000]

running_times = []

# Measuring the execution time for each n

for n in n_values:

    running_time = measure_running_time(n)

    running_times.append(running_time)

for n, y in zip(n_values, running_times):

    print(f"When input value n is {n}: the running time is {y}")


# When input value n is 1000000: the running time is 0.9249532222747803

# When input value n is 2000000: the running time is 1.6297001838684082

# When input value n is 3000000: the running time is 2.5987300872802734

# When input value n is 4000000: the running time is 3.3289573192596436

# When input value n is 5000000: the running time is 4.551220178604126
```

Here is the running time graph:

```
# Creating the chart

import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))

plt.plot(n_values, running_times, marker='o')

plt.title('Running Times vs. various input n')

plt.xlabel('n (Number of Elements)')

plt.ylabel('Running Time (seconds)')

plt.grid(True)

plt.xticks(n_values)

plt.xscale('linear') # Setting x-axis to linear scale

plt.yscale('linear')  # Setting y-axis to linear scale

plt.show()
```



Give an estimation of how long it would take for n = 1 trillion

From my data:

For n=1,000,000, time is approximately 0.925 seconds.

For n=5,000,000, time is approximately 4.551 seconds.

Estimate the time for n=1 trillion (10^{12}):

Going from 5,000,000 (5×10^{6}) to 1 trillion (10^{12}) is a factor of 200,000 times in n.

$t_{5\ million}$ = $\log 5million$ = 4.551 seconds

$t_{1\ trillion}$ = $\log 1trillion$

$= \log(5million \times 200000) \times t_{5\ million}$

= ($log\, 5million + \log 200000$) $\times\, t_{5\ million}$

= ($t_{5\ million}$ $+$ 17.61) ×4.551 seconds

= (4.551 $+$ 17.61) ×4.551 seconds

= 100.809201 seconds