Xuan(James) Zhai - CS 5350 - Assignment 1 (Simple Growth)

1: Write a program that takes a value “n” as input and prints “Hello, World” n times.

1.1: A function representing the running time of the code. Give a tight asymptotic bound for that function.

Since the code is simply one for loop, with one print function inside of the loop, the function should be

$$f(x) = n * c$$

Where c is the total time for running the one line of the for loop statement(check+increment) and the print statement; c is a constant.

From the function, we can figure out that the asymptotic bound would be Θ(n); the processing time increases linearly with n.

1.2: Run your code for various times of n and run it.

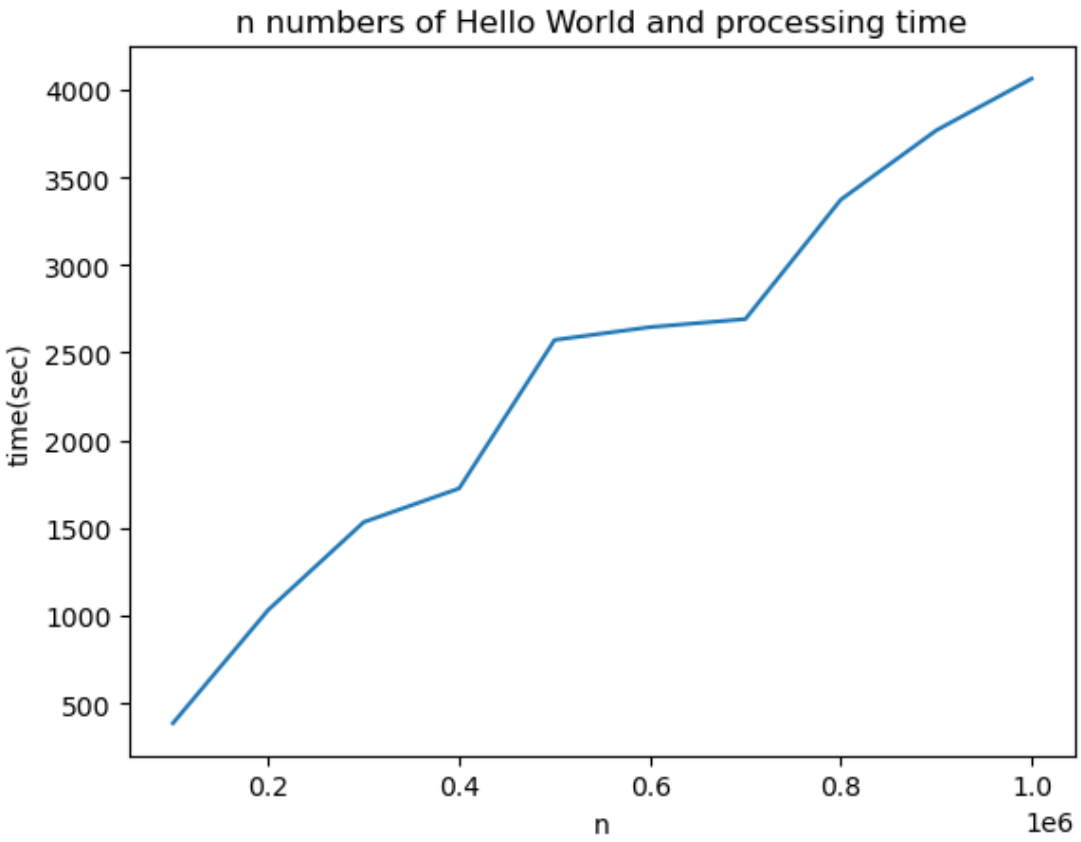
The n I chose for this task is [100000, 200000, 300000, 400000, 500000, 600000, 700000, 800000, 900000, 1000000(1 million)], and the time I counted is in millisecond.

In [3]:
```
import time
import matplotlib.pyplot as plt



def Task1(size):
    start = time.time()
    for i in range(size):
        print("Hello, World")

    end = time.time()
    ans = (end-start)
    print("this is", ans)
    return ans
```

In [ ]:
```
Q1table = []
Q1Xaxis = [100000, 200000, 300000, 400000, 500000, 600000, 700000, 800000, 900000, 1000000]
for a in range(100000, 1100000, 100000):
    Q1table.append(Task1(a))

plt.plot(Q1Xaxis, Q1table)
plt.title("n numbers of Hello World and processing time")
plt.xlabel("n")
plt.ylabel("time(sec)")
plt.show()
print(Q1table)
```

n	Time(ms)
100000	386.00

n	Time(ms)
200000	1033.00
300000	1532.99
400000	1725.00
500000	2572.00
600000	2645.00
700000	2691.01
800000	3371.99
900000	3767.07
1000000	4062.00



1.3&1.4 Analysis and prediction

From the graph presented above, as n increases, the running time also shows a linear increasing; This result supports our function f(x) = c*n, with a constant c is around 4e-3.

If n equal to 1 trillion, the time will be around 4e-3 * 1e12 = 4e9ms or 4000000 seconds.

2: 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.

2.1: A function representing the running time of the code. Give a tight asymptotic bound for that function.

The code has a for loop, and in the for loop it has a while loop that will find the location of the inserted number. For the bigger for loop, it will always run n times which n is the number of values. for the while loop, it will run 1-n times since in the best case the new value does not need to move and in the worst case the new number needs to move all the way to the head. Therefore, the running time is Ω(n) and O(n^2), and that running time T(n) for an input n satisfies

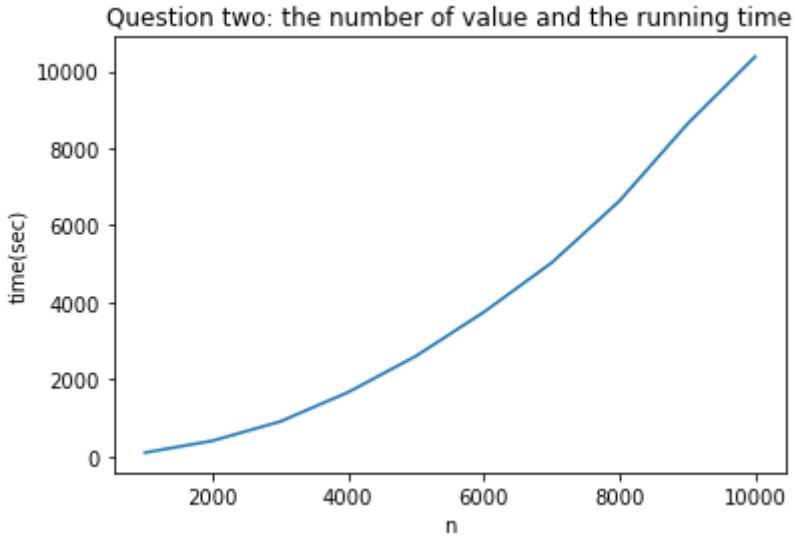
$$c1|n| \leq T(n) \leq c2|n|^2$$

2.2: Run your code for various times of n and run it.

The n I chose for this task is [1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000], and the time I counted is in millisecond.

In [7]:
```
import random

def Task2(size):

    newarray = []
    start = time.time()
    for i in range(size):
        temp = random.randint(1,size)
        newarray.append(temp)
        j = len(newarray)-2
        while(newarray[j] > temp and j>=0):
            temp1 = newarray[j+1]
            newarray[j+1] = newarray[j]
            newarray[j] = temp1
            j = j - 1
    end = time.time()
    ans = (end-start) * 1000
    return ans
```

In [8]:
```
Q2table = []
Q2Xaxis = [1000,2000,3000,4000,5000,6000,7000,8000,9000,10000]
for b in range(1000, 11000, 1000):
    Q2table.append(Task2(b))
plt.plot(Q2Xaxis, Q2table)
plt.title("Question two: the number of value and the running time")
plt.xlabel("n")
plt.ylabel("time(sec)")
plt.show()
print(Q2table)
```



[102.00643539428711, 410.9950065612793, 911.9975566864014, 1672.0080375671387, 2611.999988555908, 3751.9922256469727, 5031.986951828003, 6638.0205154418945, 8630.982637405396, 10380.018472671509]

n	Time(ms)
1000	102.01
2000	411.00
3000	911.00
4000	1672.01
5000	2612.00
6000	3752.00
7000	5031.99
8000	6638.02
9000	8630.98
10000	10380.02

2.3&2.4 Analysis and prediction

From the graph above we can find out that it has an exponential growth. as n increases, the time will increase much higher. It corresponds with our though that

$$c1|n| \leq T(n) \leq c2|n|^2$$

Furthermore, this function is almost a insertion sort, and we know insertion sort has Ω(n) at best case and O(n2) at the worst case.

It is hard to predict the running time when n is 1e12, but from the data we collection, that when n is multiple by 10, the time will multiple by 10^2, we can think that the running time will be around 1e20.

3: 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.

3.1: A function representing the running time of the code. Give a tight

asymptotic bound for that function.

In this function, it has only one for loop because it will identify the new value while appending it to the list. In the for loop, it has a if statement which helps identify the value and store the result. As a result, the function of this code is

$$f(x) = c * n$$

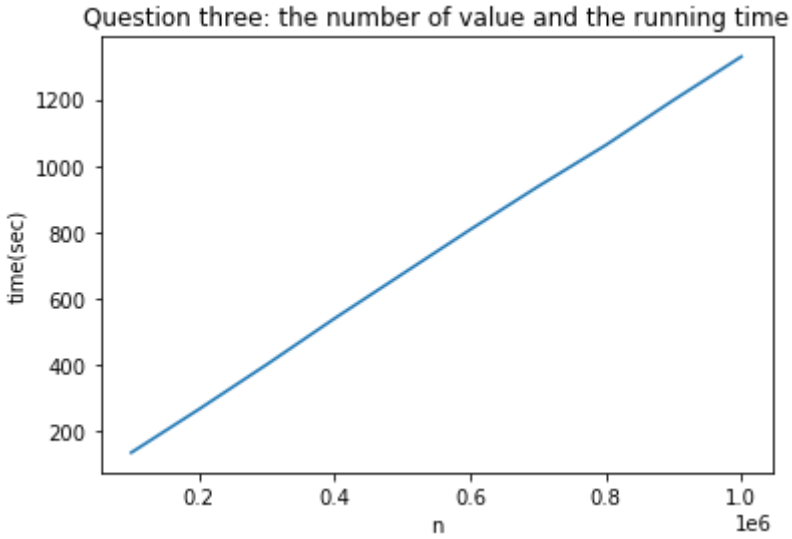
which c is a constant refers to the if-statements and the append action, and the n is the number of value.The code has an asymptotic bound Θ(n).

3.2: Run your code for various times of n and run it.

The n I chose for this task is [100000, 200000, 300000, 400000, 500000, 600000, 700000, 800000, 900000, 1000000(1 million)], and the time I counted is in millisecond.

In [9]:
```
def Task3(size):
    newarray = []
    num1,num2,num3 = 0,0,0
    start = time.time()
    for i in range(size):
        temp = random.randint(1,3)
        if(temp == 1):
            num1 = num1 + 1
        elif(temp == 2):
            num2 = num2 + 1
        else:
            num3 = num3 + 1
        newarray.append(temp)
    end = time.time()
    ans = (end-start) * 1000
    return ans
```

In [10]:
```
Q3table = []
Q3Xaxis = [100000,200000,300000,400000,500000,600000,700000,800000,900000,1000000]
for c in range(100000, 1100000, 100000):
    Q3table.append(Task3(c))

plt.plot(Q3Xaxis, Q3table)
plt.title("Question three: the number of value and the running time")
plt.xlabel("n")
plt.ylabel("time(sec)")
plt.show()
print(Q3table)
```



[133.99910926818848, 265.0001049041748, 400.00319480895996, 538.9978885650635, 673.0005741119385, 807.0023059844971, 936.9864463806152, 1061.9995594024658, 1197.998285293579, 1328.9990425109863]

n	Time(ms)
100000	134.00
200000	256.00
300000	400.00
400000	539.00
500000	673.00
600000	807.00
700000	936.99
800000	1062.00
900000	1198.00
1000000	1329.00

3.3&3.4: Analysis and prediction

The graph shows a straight line which supports our though. Also, from the table we can find out that when n from 100000 increases to 1000000, the time is also 10 times bigger that before.

When n is a trillion which is 1e12, the time would be around 1.3e8

4: 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.

4.1: A function representing the running time of the code. Give a tight asymptotic bound for that function.

In this question, I will use the merge sort method since it has a $O(nlogn)$. Also, the fundimental idea of merge sort is split the array into two parts until there are only two elements in each part, and then sort each part and merge them back. Therefore, from the book "introduction to algorithm" by Corman, we can know that for merge sort time T(n):

- the dividing operation is constant Θ(1) - you just need to find the middle element of the array by dividing it's length by 2
- the recursion on each part of the array takes T(n/2) time which makes 2T(n/2) for both of them
- the merge operation is known to have Θ(n) complexity

So finally T(n) = 2*T(n/2) + Θ(n) = Θ(nlogn) if n > 1

4.2: Run your code for various times of n and run it.

Since the function in question 3 only returns the time, I have to modify the Task3 to make it to return the array.

I uses the merge sort from https://www.geeksforgeeks.org/merge-sort/

The n I chose for this task is [100000, 200000, 300000, 400000, 500000, 600000, 700000, 800000, 900000, 1000000(1 million)], and the time I counted is in millisecond.

In [12]:
```
def Task3v2(size):
    newarray = []
    num1,num2,num3 = 0,0,0
    for i in range(size):
        temp = random.randint(1,3)
        if(temp == 1):
            num1 = num1 + 1
        elif(temp == 2):
            num2 = num2 + 1
        else:
            num3 = num3 + 1
        newarray.append(temp)
    return newarray



def mergeSort(arr):     # Reference: https://www.geeksforgeeks.org/merge-sort contrib
    if len(arr) > 1:

         # Finding the mid of the array
        mid = len(arr)//2
        # Dividing the array elements
        L = arr[:mid]
        # into 2 halves
        R = arr[mid:]
        # Sorting the first half
        mergeSort(L)
        # Sorting the second half
        mergeSort(R)
        i = j = k = 0
        # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1
        # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i += 1
```

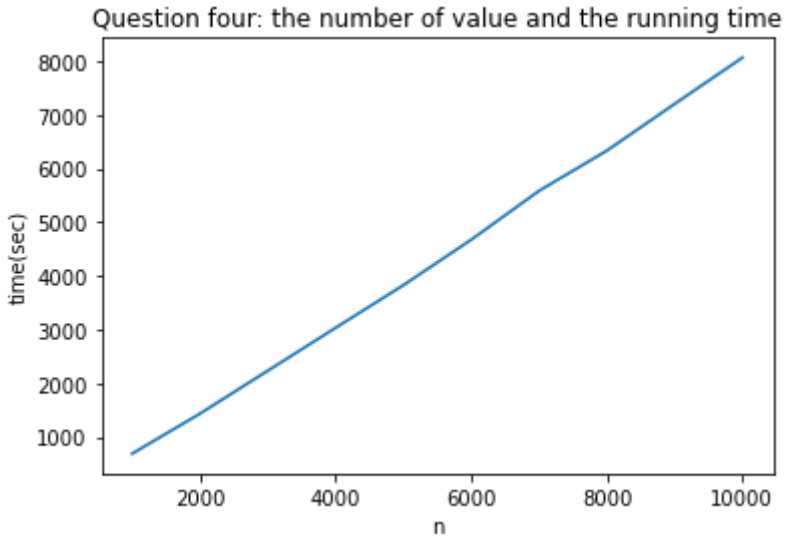
```
            k += 1
        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
```

In [13]:

```
def Task4(newarray):
    start = time.time()
    mergeSort(newarray)
    end = time.time()
    ans = (end-start) * 1000
    return ans
```

In [14]:

```
Q4table = []
Q4Xaxis = [1000,2000,3000,4000,5000,6000,7000,8000,9000,10000]
for d in range(100000, 1100000, 100000):
    Q4table.append(Task4(Task3v2(d)))
plt.plot(Q4Xaxis, Q4table)
plt.title("Question four: the number of value and the running time")
plt.xlabel("n")
plt.ylabel("time(sec)")
plt.show()
print(Q4table)
```



[686.9857311248779, 1432.0192337036133, 2231.9986820220947, 3031.000852584839, 3826.0040283203125, 4673.009395599365, 5582.995891571045, 6331.979036331177, 7202.022075653076, 8064.996242523193]

n	Time(ms)
100000	686.99
200000	1432.02
300000	2231.99
400000	3031.00
500000	3826.00
600000	4673.01
700000	5583.00
800000	6331.98
900000	7202.02

n	Time(ms)
1000000	8065.00

4.3&4.4: Analysis and Prediction

Due to the fact that there are only three different numbers in the array, It may take less time to do the swap, so the line shown on the graph may be hard to reflect a O(nlogn), but you can still see the curve in it. And from the table we can find out, as n increases, the time is getting greater than c*n.

When n is equal to 1 trillion, from the table we can guess that the time will be around 1e10.

Reference List:

Thomas H. Cormen. Introduction to Algorithms. MIT Press; 3rd edition, ISBN-10:9780262033848, September 1, 2009.