Session for lecture 11: Understanding Program Efficiency: 2

LAST TIME

- ➔ *The important performance measures for programs*
- ➔ Methods for evaluating program performance
- ➔ “Big-oh” notation
- ➔ Estimating the running time of programs using the big-oh notation

TODAY

- ➔ *Classes of complexity*
- ➔ *Examples characteristic of each class*

1. O(1) denotes constant running time
2. O(log n) denotes logarithmic running time
3. O(n) denotes linear running time
4. O(n log n) denotes log-linear running time
5. O(n^c) denotes polynomial running time (c is a constant)
6. O(c^n) denotes exponential running time (c is a constant being raised to a power based on size of in put)

 # Hints: you may make use of the following functions.

```
import time
import random
def plot(sizes, times):
    import matplotlib.pyplot as plt
    plt.figure()
    plt.plot(sizes, times)
    plt.xlabel("Sizes")
    plt.ylabel("Times(s)")
    plt.grid(True, which="both")
    plt.show()

def generate_list(length):
    L = []
    for i in range(length):
        L.append(random.randint(1,1000))
    return L
```

 # Q.1 [1 mk] → run the following program and notice its complexity order.

```
def someFunction(x,y):
    return x**y

t = []
s = []
for i in range(5, 25):
    time_start = time.clock()
    someFunction(i, i)
    time_end = time.clock() - time_start
    t.append(time_end)
    s.append(i)
plot(s,t)
```

 # Q.2 [1 mk] → run the following program and notice its complexity order.

```
def bisect_search(L, e):
    def bisect_search_helper(L, e, low, high):
        #print('low: ' + str(low) + '; high: ' + str(high))  #added to visualize
        if high == low:
            return L[low] == e
        mid = (low + high)//2
        if L[mid] == e:
            return True
        elif L[mid] > e:
            if low == mid: #nothing left to search
                return False
            else:
                return bisect_search_helper(L, e, low, mid - 1)
        else:
            return bisect_search_helper(L, e, mid + 1, high)
    if len(L) == 0:
        return False
    else:
        return bisect_search_helper(L, e, 0, len(L) - 1)

# testList = []
# for i in range(100):
#         testList.append(i)
# test the code
# print(bisect_search(testList, 76))

for i in range(1, 5):
    L = generate_list(10**i)
    time_start = time.clock()
    bisect_search(L, len(L))
    time_end = time.clock() - time_start
    t.append(time_end)
    s.append(i)
plot(s,t)
```

 # Q.3 [1 mk] → run the following program and notice its complexity order.

```
def linear_search(L, e):
    found = False
    for i in range(len(L)):
        if e == L[i]:
            found = True
    return found

for i in range(1, 5):
    L = generate_list(10**i)
    time_start = time.clock()
    linear_search(L, len(L))
    time_end = time.clock() - time_start
    t.append(time_end)
    s.append(i)
plot(s,t)
```

 # Q.4 [1 mk] → run the following program and notice its complexity order.

```
def bubbleSort(L):
   for i in range(len(L)):
       for j in  range(len(L)-i-1): #   Last i elements are already in place
           if L[j] > L[j+1]:
               tmp = L[j]
               L[j] = L[j+1]
               L[j+1] = tmp

for i in range(1, 5):
    L = generate_list(10**i)
    time_start = time.clock()
    bubbleSort(L)
    time_end = time.clock() - time_start
    t.append(time_end)
    s.append(i)
plot(s,t)
```

 # Q.5 [1 mk] → run the following program and notice its complexity order.

```
def fibonacci(n):
    if n <= 2:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
t = []
s = []
for i in range(5, 25):
    time_start = time.clock()
    fibonacci(i)
    time_end = time.clock() - time_start
    t.append(time_end)
    s.append(i)
plot(s,t)
```

"Take pride in how far you've come. Have faith in how far you can go. But don't forget to enjoy the journey."