```
!date
```

```
Fri Feb 18 00:37:58 UTC 2022
```

CSCI-SHU 210 Data Structures

Recitation 2 Analysis of Algorithms

Important for this week:

1. Determine the tightest big O runtime for a given “iterative” code snippet;
2. Code under big O runtime restrictions;
3. Know what is space complexity;

- For students who have recitation on Wednesday, you should submit your solutions by **Feb 18th** Friday 11:59pm.
- For students who have recitation on Thursday, you should submit your solutions by **Feb 19th** Saturday 11:59pm.
- For students who have recitation on Friday, you should submit your solutions by **Feb 20th** Sunday 11:59pm.

Question 1 (Theory) - just making sure you understand the Big-O definition:

1. Prove that running time T(n) = n^2 + 20n + 1 is O(n^2)

   Solution:

   Let $c = 2$, $n_0 = 21$,

   then $n^2 + 20n + 1 \leqslant 2n^2$ for all $n \geqslant 21$

2. Prove that running time T(n) = n^2 + 20n + 1 is not O(n)

   Solution:

   For any $c \in \mathbb{R}^+$ and any large n_0, we can always find $n_1 = \max\{c, n_0\}$, such that

   $$n_1^2 + 20n_1 + 1 > cn_1$$

Question 2 (Code snippet analysis):

Determine the tightest big O runtime for each of the following code fragment:

```
#Fragment1:

def func1(N):
        for i in range(N):
                for j in range(N, 0, -2):
                        print("hi")
```

```
#Fragment 1 tightest big O:: O(n^2)
```

```
#Fragment2:

def func2(N):
        for i in range(N):
                for j in range(N, 0, -2):
                        print("hi")

        x = 0
        while x < N:
                x += 1
                print("hiii")
```

```
#Fragment 2 tightest big O:: O(n^2)
```

```
# Fragment3:

def func3(N):
        i = 0
        while i < N:
                j = N
                while j > 0:
                        j //= 2
                        print("hi")
                i += 1
```

```
#Fragment 3 tightest big O:: O(nlogn)
```

Question 3 (Concept):

You have an N-floor building and plenty of eggs. Suppose that an egg is broken if it is thrown from floor F or higher, and unhurt otherwise. Suppose F is within the range of N floor which means that you can always find the floor F such that the egg breaks

1. Describe a strategy to determine the value of F such that the number of throws is at most

1. Describe a strategy to determine the value of F such that the number of throws is at most log N.

```
class Eggs:
    def __init__(self, F):
        self._floor = F
        self.count = 0

    def isnotbroken(self, flr):
        self.count += 1
        if flr >= self._floor:
            return 0
        else:
            return 1


# Method 1, finding F in at most logN + 1 throws
def findFromTop(eggs, N):
    top, bottom =  N, 1
    while top > bottom:
        mid = int((top+bottom)/2)
        if eggs.isnotbroken(mid):
            bottom = mid + 1
        else:
            top = mid        # F could be at floor `mid'
    return top, eggs.count


def main1(F,N):
    eggs = Eggs(F)
    t1, t2 = findFromTop(eggs, N)
    if t1 != F:
        raise Exception('An error occurred')
    else:
        print('you find the floor in {} goes'.format(t2))
```

2. Find a new strategy to reduce the number of throws to at most 2 log F. (optional)

```
# Method 2, finding F in at most 2logF + 1 throws
def findFromBot(eggs, N):
        k = 0
        while eggs.isnotbroken(2**k):
                k += 1
                if 2**k > N:
                        break
        top = min(N, 2**k)

        bottom = 2**(k-1) + 1
        while top > bottom:
                mid = int((top+bottom)/2)
                if eggs.isnotbroken(mid):
                        bottom = mid + 1
```

```
            else:
                top = mid
        return top, eggs.count


def main2(F,N):
    eggs = Eggs(F)
    t1, t2 = findFromBot(eggs, N)
    if t1 != F:
        raise Exception('An error occurred')
    else:
        print('you find the floor in {} goes'.format(t2))
```

Question 4 (Prime number):

A number is said to be prime if it is divisible by 1 and itself only, not by any third variable. The following are the descriptions of two algorithms for deciding whether a number is a prime or not. Please implement the two algorithms is_prime1 and is_prime2 by yourself. And also answer the questions of "What is the runtime for algorithm 1&2?"

1. Divide N by every number from 2 to N - 1, if it is not divisible by any of them hence it is a prime.
2. Instead of checking until N, we can check until $\sqrt{N}$ because a larger factor of N must be a multiple of smaller factor that has been already checked.

```
def is_prime1(N):
        for i in range(2, N - 1):
                if N % i == 0:
                        return False
        return True


def main():
        if not is_prime1(1299827) == True:
                print('1299827 should be a prime but you returned False.')
        if not is_prime1(1296041) == True:
                print('1296041 should be a prime but you returned False.')
        if is_prime1(1296042) == True:
                print('1296042 should not be a prime but you returned True.')

if __name__ == '__main__':
        main()
```

What is the runtime for algorithm 1?

```
# O(N)
```

```
def is_prime2(N):
    for i in range(2, int(N ** 0.5) + 1):
        if N % i == 0:
            return False
    return True


def main():
    if not is_prime2(1299827) == True:
        print('1299827 should be a prime but you returned False.')
    if not is_prime2(1296041) == True:
        print('1296041 should be a prime but you returned False.')
    if is_prime2(1296042) == True:
        print('1296042 should not be a prime but you returned True.')

if __name__ == '__main__':
    main()
```

What is the runtime for algorithm 2?

```
# O(N^0.5)
```

Question 5 (permutation):

Suppose you need to generate a random permutation from 0 to N-1. For example, {4, 3, 1, 0, 2} and {3, 1, 4, 2, 0} are legal permutations, but {0, 4, 1, 2, 1} is not, because one number (1) is duplicated and another (3) is missing. This routine is often used in simulation of algorithms. We assume the existence of a random number generator, r, with method randInt(i,j), that generates integers between i and j (i & j included) with equal probability. The following are three algorithms. Please implement the three algorithms by yourself and answer the question of the expected runtime for the three algorithms.

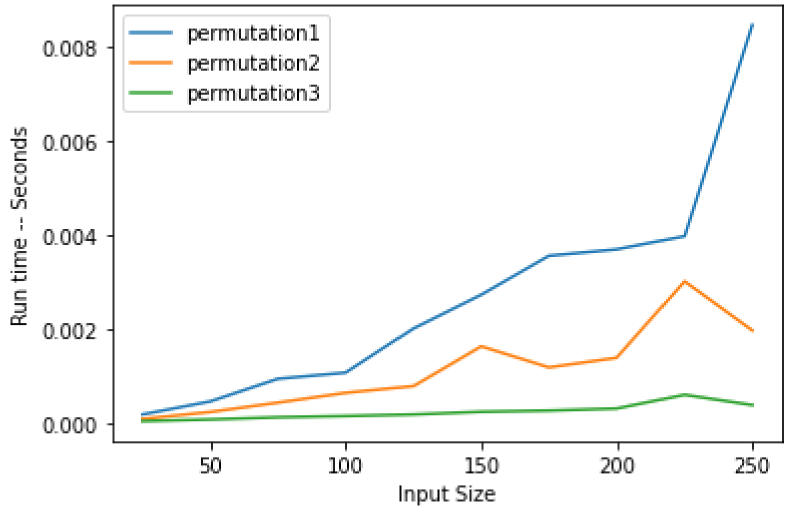
1. Create a size N empty array. (array = [None] * N) Fill the array a from a[0] to a[N-1] as follows: To fill a[i], generate random numbers until you get one that is not already in a[0], a[1], . . . , a[i-1].

2. Same as algorithm (1), but keep an extra array called the used array. When a random number, ran, is first put in the array a, set used[ran] = true. This means that when filling a[i] with a random number, you can test in one step to see whether the random number has been used, instead of the (possibly) i steps in the first algorithm.

3. Fill the array such that a[i] = i. Then: for i in range(len(array)): swap(a[i], a[randint(0, i)]);

```
import timeit
import matplotlib.pyplot as plt
import random

def timeFunction(f,n,repeat=1):
    return timeit.timeit(f.__name__+'('+str(n)+')',setup="from __main__ import "+f.__name__


def permutation1(N):
    array = [None] * N
    for i in range(N):
        rand = random.randint(0, N - 1)
        while rand in array:
            rand = random.randint(0, N - 1)
        array[i] = rand


def permutation2(N):
    array = [None] * N
    used = [False] * N
    for i in range(N):
        rand = random.randint(0, N - 1)
        while used[rand] == True:
            rand = random.randint(0, N - 1)
        array[i] = rand
        used[rand] = True


def permutation3(N):
    array = [i for i in range(N)]
    for i in range(N):
        idx = random.randint(0, N - 1)
        array[i], array[idx] = array[idx], array[i]


def plot_data():
    x = [25, 50, 75, 100, 125, 150, 175, 200, 225, 250]
    y = []
    z = []
    j = []
    for each in x:
        y.append(timeFunction(permutation1, each))
        z.append(timeFunction(permutation2, each))
        j.append(timeFunction(permutation3, each))
    line1, = plt.plot(x, y, label="permutation1")
    plt.legend()
    line2, = plt.plot(x, z, label="permutation2")
    plt.legend()
    line3, = plt.plot(x, j, label="permutation3")
    plt.legend(handles=[line1,line2,line3])
```

```
        plt.xlabel("Input  Size")
        plt.ylabel("Run  time  --  Seconds")
        plt.show()

if  __name__  ==  '__main__':
        plot_data()
```



What is the expected runtime for algorithm 1?

```
#  O(N^2logN)
```

What is the expected runtime for algorithm 2?

```
#  O(NlogN)
```

What is the expected runtime for algorithm 3?

```
#  O(N)
```

Plot the runtime of algorithm 1, 2, 3 using the given plot_data( ) function. What are your observations?

```
#  On  average,  given  the  same  input  size,  permutation  1  takes  longer  than  permutation
#  and  permutation  2  takes  longer  than  permutation  3  to  finish.
```

0 秒 完成时间：08:37