```
!date
from math import sqrt

     Wed Jan 31 09:23:17 AM UTC 2024
```

Please run the above line to refresh the date before your submission.

CSCI-SHU 210 Data Structures

Recitation2 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 Friday 11:59pm.
- For students who have recitation on Thursday, you should submit your solutions by Saturday 11:59pm.
- For students who have recitation on Friday, you should submit your solutions by Sunday 11:59pm.

Name: Margad Gereltbaatar (MJ)

NetID: mg7502

Please submit the following items to the Gradescope:

- URL: Your Colab notebook link. Click the Share button at the top-right, share with NYU, and paste to Gradescope
- PDF: The printout of your run in Colab notebook in pdf format

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

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

```
# Your anwser
# When there are multipe segment of code that handles input resulting in a runtime with multibel variable
# like this, to calculate the runtime, we only take into consideration the one with largest denominator,
# that in this case is n^2 making the running time O(n^2)
```

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

```
# Your answer
#This is incorrect as the runtime is O(N^2).
```

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 log N.

```
# Similar to Binary search, I will start at N/2 and if it breaks I will go to N/2 - N/2/2 if wont break, I will go to N/2 + N/2/2 and so on.
```

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

```
# Your answer
```

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):
    """
    Divide N by every number from 2 to N - 1,
    if it is not divisible by any of them hence it is a prime.

    :param N: Int -- The number being checked.
    :return: True if N is a prime number, return False otherwise.
    """
    for i in range(2, N):
        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):
    """
    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.

    :param N: Int -- The number being checked.
    :return: True if N is a prime number, return False otherwise.
    """
    for i in range(2 , int(sqrt(N))+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?

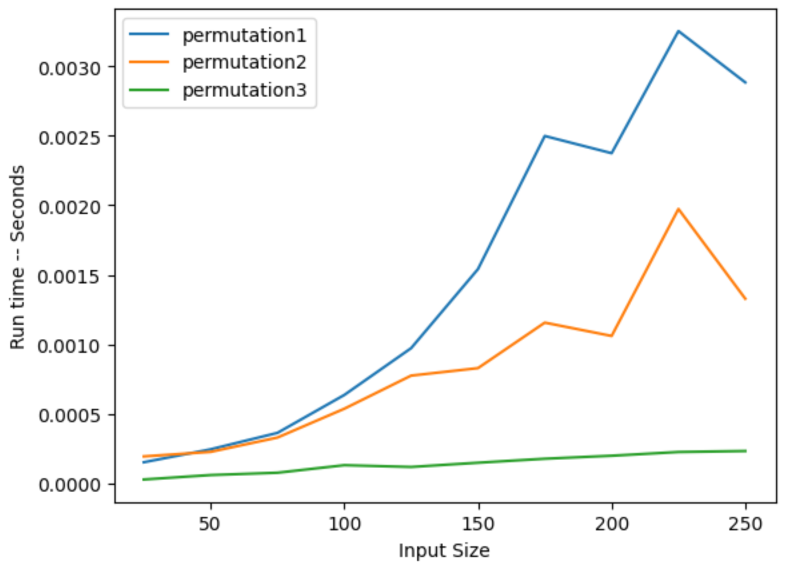
```
# O(sqrt(N))
```

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__,number=repeat)/repeat


def permutation1(N):

    random_arr = [None]*N
    for i in range(N):
        while True:
            num = random.randint(0,N-1)
            if num not in random_arr:
                random_arr[i] = num
                break
    return random_arr

def permutation2(N):

    random_arr = [None]*N
    used = [False]*N
    for i in range(N):
        while True:
            num = random.randint(0,N-1)
            if used[num] == False:
                random_arr[i] = num
                used[num] = True
                break
    return random_arr

def permutation3(N):

    random_arr = [i for i in range(N)]
    for i in range(N):
        c = random.randint(0,N-1)
        temp = random_arr[i]
        random_arr[i] = random_arr[c]
        random_arr[c] = temp

    return random_arr


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?

```
# Your answer: O(N^2)
```

What is the expected runtime for algorithm 2?

```
# Your answer: O(N)
```

What is the expected runtime for algorithm 3?

```
# Your answer: O(N)
```

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

```
# Your answer: Since the runtime of the first program is the highest, the larger the
# input size grows, the longer it took for it to finish, as it is a O(N^2).
# Even though the overall runtime of the last 2 algortihm is same, algorithm 3 is much more efficient
# since it's truly O(N) and the second is O(2N) because in the loop we check if the element is in another list
```