# Recitation2
# Analysis of Algorithms

# Question 1 (Theory):

Prove that running time $\mathrm{T(n)} = n^2 + 20n + 1$ is $O(n^2)$

Need to show a pair of constants $\begin{cases} c \\ n_0 \end{cases}$, such that for all $n \geq n_0$, $\mathrm{T(n)} \leq c * n^2$

$$n^2 + 20n + 1 \leq cn^2$$

# Question 1 (Theory):

Prove that running time $\mathrm{T(n)} = n^2 + 20n + 1$ is not $O(n)$

Need to show we can't find a pair of constants $\begin{cases} c \\ n_0 \end{cases}$, such that for all $n \geq n_0$, $\mathrm{T(n)} \leq c * n$

# Question 2 (Code snippet analysis)

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

N times

0.5N times

# Question 2 (Code snippet analysis)

```
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")
```

N times

0.5N times

N times

# Question 2 (Code snippet analysis)

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

N times

logN times

# 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.

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

# logN solution

N is the total number of floors. Thus, we can binary search total floors.

Let lowest =1 and highest = N

Then repeat

    Go to floor k = (highest-lowest)//2

    Throw egg and check result

    If egg breaks then highest = k-1, otherwise lowest = k+1

until highest is inferior or equal to lowest

# 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.

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

# 2logF solution

F is where egg just start to break.

Let k = 0

Repeat

Go to floor $2^k$

Throw egg and check result

If egg doesn't break then k = k+1

until egg breaks

I think that this may be between 2^k-1 and 2^k???

Finally, use binary search between floors $2^k$ and $2^{k+1}$

# 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.

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 $\sqrt{N}$.

```python
flag = True
for i in range(2, N):
    if N % i == 0:
        flag = False
return flag
```

Worst-case runtime: $O(N)$

```python
flag = True
for i in range(2, int(math.sqrt(N)) + 1):
    if N % i == 0:
        flag = False
return flag
```

Worst-case runtime: $O(N^{\frac{1}{2}})$

# Question 5 (permutation):

1. 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].

- The Runtime for this algorithm is $O(n^2 logn)$

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

O(N) checks for in operator

Need total O(NlogN) random numbers

# Why total O(NlogN) computations for all the random numbers?

- Suppose we need 10 numbers

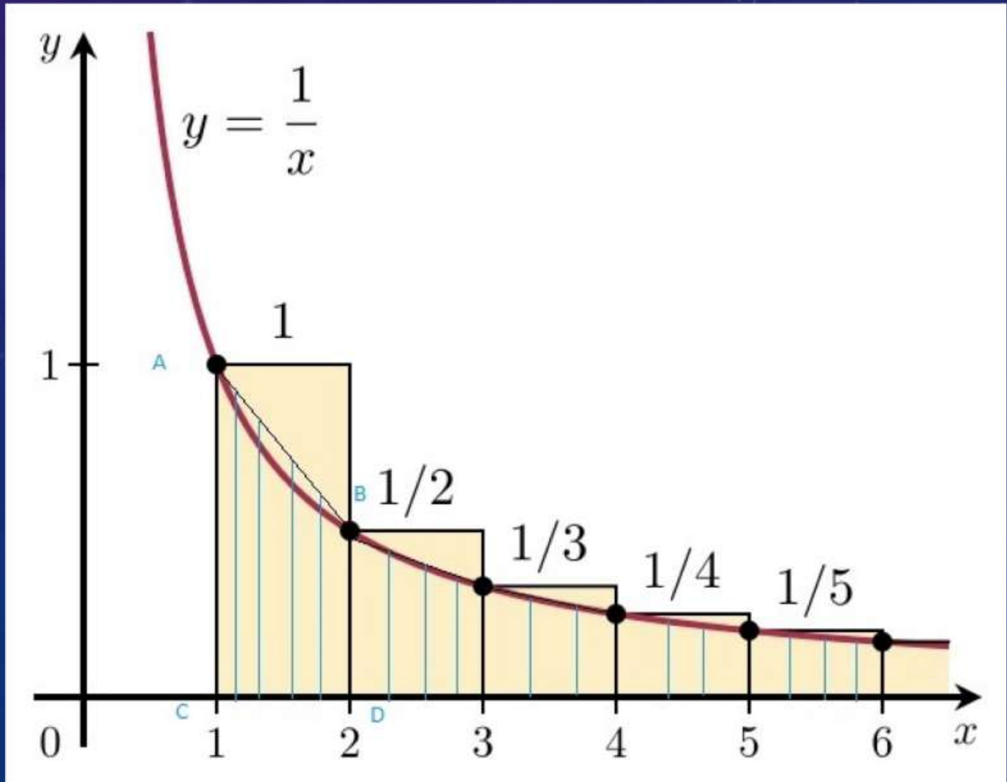
- First random number, the expected try time is $\frac{10}{10} = 1$ try.
- Second random number, the expected try time is $\frac{10}{9} = 1.11\ tries$
- Third random number, the expected try time is $\frac{10}{8} = 1.25\ tries$

- …
- Last random number, the expected try time is $\frac{10}{1} = 10\ tries$

- So we have a series of

$$\frac{N}{N}+\frac{N}{N-1}+\frac{N}{N-2}+\ \ldots+\frac{N}{1}$$
$$=N\left(\frac{1}{N}+\frac{1}{N-1}+\frac{1}{N-2}+\ \ldots+\frac{1}{1}\right)$$

- The summation of $\left(\frac{1}{N}+\frac{1}{N-1}+\frac{1}{N-2}+\ \ldots+\frac{1}{1}\right)$is the integral (area under the curve)
- Integral of $y=\frac{1}{x}$ is:
- $y=\ln x+C$



# Question 5 (permutation):

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.

```python
def permutation2(N):
    result = [None] * N
    extra_array = [False] * N
    for i in range(N):
        rand = random.randint(0, N - 1)
        while (extra_array[rand] == True):
            rand = random.randint(0, N - 1)
        result[i] = rand
        extra_array[rand] = True
    return result
```

look up by index, O(1)

for loop with random, O(nlogn)

The Runtime for this algorithm is $O(nlogn)$

# Question 5 (permutation):

3. Fill the array such that **a[i] = i**. Then:

```
for i in range(len(array)):
        swap( a[ i ], a[ randint( 0, i ) ] );
```

```
def permutation3(array):
    result = []
    for i in range(len(array)):
        result.append(i)
    for i in range(len(array)):
        random_index = random.randint(0, i)
        result[i], result[random_index] = result[random_index], result[i]
    return result
```

Fill the array, O(N)

O(N) swaps

The Runtime for this algorithm is $O(N)$