def mod\_exp(x: int, y: int, N: int) S-> int:

    #return 0

    if y == 0:

        return 1

    # Recursive call to mod\_exp function to calculate x^(y//2) mod N.

    #O(n^2) + O(1)

    z = mod\_exp(x, y // 2, N)

    # If y is even

    #O(1)

    if y % 2 == 0:

        #O(n^2) + O(1)

        return z\*\*2 % N

    else:

        # If y is odd

        #O(n^2) + O(n^2) + O(n^2) + O(1)

        return x \* z\*\*2 % N

the base case y=0: return 1 has a constant time of O(1)

the recursive call z = mod\_exp(x, y // 2, N) has a time of (y/2) because it get reduced by half on each call

if y % 2 == 0: take a constant amount of time O(1) and the return function z^2 takes O(n^2) amount of time while % N. take a constant time of O(1) which give us **O(n^2)** because O(N^2) + O(1) = O(n^2)

if the y is odd then the function x \* z\*\*2 % N will be O(n^2) for the function x\*z^2 and O(n^2) for the function Z^2. The %N has a constant of O(1). Time complexity will be O(n^2) + O(n^2) + O(1) = O(n^2).

Therefore Time complexity is of the mod\_exp function will be T(n) = O(n^2 log n)

Since the dept of the function is O(log(n)) the space complexity is log(n) because each recursive call use a constant amount of space.

def fermat(N: int, k: int) -> str:

    #return "???"

    # If N is even, it is not prime except for 2 itself. the probability of a random number being prime is 1/2. the function runs k times to increase the posibility of N being prime.

#

    for \_ in range(k):

        #in the loop, generate a random number a and check if a^(N-1) mod N != 1, then N is composite.

        a = random.randint(1, N - 1)

        if mod\_exp(a, N - 1, N) != 1:

            return 'composite'

    return 'prime'

The loop runs k times

Generating a random number will take a constant time of O(1)

Mod\_exp (a, N-1, N) ! = 1 will take a time complexity of O(n^2) times

The total time complexity of the function will be O(n^2)

The loop and the mod\_exp take a constant amount of space of O(1)

def miller\_rabin(N: int, k: int) -> str:

    #return "???"

    # If N is even, it is not prime except for 2 itself

    for \_ in range(k):

        # Generate a random number a

        a = random.randint(1, N-1)

        # If a^(N-1) mod N != 1, then N is composite

        if pow(a, N-1, N) == 1:

            # If N is prime, then N-1 = 2^x \* y

            x = N-1

            # If N is prime, then N-1 = 2^x \* y

            while pow(a,x,N) == 1 and x % 2 == 0:

                x = x // 2

            if pow(a,x,N) in (N-1,1):

                return 'prime'

            else:

                return 'composite'

        else:

            return 'composite'

Generation of the random number takes a constant k amount of time.

Mod\_exp function take O(log(n)) time

As x is divided in halves each time take O(log(N)) times

The inner modexp takes O(logN) times through the loop

The total time complexity is = O(k \* log n))

Constant k is not considered and therefore is to be ignored = log n

The space complexity is O(1) because it uses a fix amount of space for the function

def ext\_euclid(a: int, b: int) -> tuple[int, int, int]:

    """

    The Extended Euclid algorithm

    Returns x, y , d such that:

    - d = GCD(a, b)

    - ax + by = d

    Note: a must be greater than b

    """

    if b == 0:

        return a, 1, 0 # return gcd, x, y

    else:

        (gcd, x, y) = ext\_euclid(b, a % b)

        return gcd, y, x - (a // b) \* y

if b == 0:

return a, 1, 0 # this takes a constant amount of time k

(gcd, x, y) = ext\_euclid(b, a % b) # this takes O(n^2) + k amount of run time

return gcd, y, x - (a // b) \* y #the return value takes O(n^2) + O(n) + K3 amount of time

the total time complexity is O(n^3)

Total space complexity is O(1) because it uses a constant amount of run time space.

  p = generate\_large\_prime(bits // 2)

    q = generate\_large\_prime(bits // 2)

    N = p \* q

    a = (p - 1) \* (q - 1)

    for e in primes:

        gcd, x, y = ext\_euclid(e, a)

        if gcd == 1:

            d = x % a

            if d < 0:

                d += a

            return N, e, d

time complexity of a = (p - 1) \* (q - 1) is O(1) while gcd, x, y = ext\_euclid(e, a) is O(n^3).

Each loop takes a constant amount of O(1)

Therefore the total time complexity is O(n^3)

The space complexity is O(1), since the code only uses a small space to store the result.

def fprobability(k: int) -> float:

    return 1 - (1/2)\*\*k

(1/2) represents the probability of a composite number passing a single iteration of the Fermat's primality test.

K represents the iteration of the function

1 - (1/2)^k represents the probability that there will be a composite or prime after k iterations. As k increases, the probability of a composite reduces.

fermat's probability is such that the probability of a random number being prime is 1/2. It being prime after k iterations is 1 - (1/2)^k. therefore, if ((n-1)/2)\*\*a is congruent to 1 mod n, then n is prime with a probability of 1 - (1/2)^k. this continues until it is congruent beyond 1, then the algorithm stops and returns.

def mprobability(k: int) -> float:

    return 1 - (1/4)\*\*k

miller-rabin's probability is such that the probability of a random number being prime is 1/4. It being prime after k iterations is 1 - (1/4)^k.

therefore, if ((n-1)/2)\*\*a is congruent to 1 mod n, then n is prime with a probability of 1 - (1/4)^k. this continues until it congruent beyond 1, then the algorithm stops and returns. once you get to -1 then you have reached the end of the algorithm and the number is prime.