ok ✓

Exercise 4

a) For example, if n equals 10, then p contains 3, 5, and 7. Since 3, 5, and 7 are all prime numbers, and 3 = 2^2-1, 5 = 2^2 + 1, and 7 = 2^3-1. They satisfy the conditions of being prime and having distances from powers of 1 to 2. (✓) what about $2 = 2^0 + 1$?

b) We can devise an algorithm for evaluating whether a number is prime, and an algorithm for evaluating whether a number is a number whose distance is a power of 1 to 2. Then combine them together to get a list of p's. ok ✓

```
c) is_prime(num):
   if num < 2 then
      return False
   for i <- 2 to (num+1)/2 + 1 do
      if num mod i = 0 then
         return False
   return True
```

can this be done more efficiently?

```
distance(num):
   n <- 1
   while 2^n - 1 <= num do
      if |2^n - num| = 1 then
         return True
      n <- n+1
   return False
```

what about 2^0

ok

```
main(num)
   P <- ∅
   for n <- 2 to num+1 do
      if is_prime(n) and distance(n) then
         P = P ∪ {n}
   return P
```

?

ok ✓

c)

d) The "is_prime" function in worst case has a time complexity of O(n/2) or simply O(n) ✓ and the "distance" function has a time complexity of O(log(n)). Combining them together using the main function of the for loop will, in the worst case, call them all at the same time for each iteration, giving a total time complexity of O(n*(n+log(n))) =

O(n²) = O(n^2+n*log(n)). ✓ *how can this be improved ?*

e) To provide a proof sketch for the correctness of the algorithm, we need to proof:

1. Correctness of "is_prime" function: ✓

so far only a description

If num < 2, then the function correctly returns False because 0 and 1 are not prime number. If num > 2, the functions checks for divisors up to num/2. If any divisor is found, the function returns False. Otherwise, it returns True. This is correct because any divisor of num must be less than or equal to num/2. *intuition ✓*

2. Correctness of "distance" function: ✓

The function uses a while loop to find the smallest n such that 2**n - 1 is greater than num. Inside the loop, it checks if abs(2**n - num) == 1. This condition ensures that num is either one less or one more than a power of 2. *ok ✓*

why is this the correct choice?

The loop will terminate when 2**n - 1 exceeds num, making n the smallest power of 2 such that 2**n - 1 is greater than num. The function correctly returns True in this case. *what happens if there is no such n?*

3. Correctness of "main" function:

It iterates through numbers from 2 to num and checks if each number is both prime and one less than a power of 2 using the "is_prime" and "distance" functions.

If both conditions are met, the number is appended to the list p.

The final result is a list of prime numbers that are one less than a power of 2 up to the given num.

The correctness of the main function relies on the correctness of the "is_prime" and "distance" functions, which we have established. Therefore, we can conclude that the algorithm is correct based on the correct implementation and logic of its constituent functions. *details?*

consider showing an equivalence

f)

```
def is_prime(num):
  if num < 2:
    return False
  for i in range(2,int((num+1)/2)+1):
    if num % i == 0:
```

```
      return False
  return True
```

```
def distance(num):
  n = 1                  what about 2^0 ?
  while 2**n - 1 <= num:
    if abs(2**n - num) == 1:
      return True
    n += 1
  return False
```

```
def main(num):
  p = []
  for i in range(2,num+1):
    if is_prime(i) and distance(i):
      p.append(i)
  return p
```