# 1a

Suppose g is a generator of the group $Z_p^{\times}$.

Since the order of g is (p-1), $g^q$ where $q < p$ can never be 1. Our algorithm sets q to $p-1/p_i$ which is less than $p-1$. Thus a generator always produces true.

Suppose g is not a generator of the group.

Then g is a generator of a subgroup.
By LaGrange's theorem, the order of the group, p, is divisible by the order of any subgroup, q. Thus q divides p. if q=$p_i^{e_i}$ for some i, then $g^{(p-1)/p_i}$ would be 1 since g is a generator for that subgroup. If q is a subgroup of the subgroup of order $p_i^{e_i}$, then q divides that group's order, and so raising it to the groups order would also produce 1. Thus the algorithm would always produce false.

Let A be the event g is a generator
Let B be the event the algorithm produces true

I've proved A => B and ¬A => ¬B
But I can take the contrapositive (¬A => ¬B) => (B => A)
Thus I have A=> B and B=>A
Thus A <=> B

# 1b

We have a loop over k. Inside the loop we are raising g to the exponent $(p-1)/p_i$. We can use the square and multiply algorithm to do this exponentiation in O(log(p-1)). Thus the total algorithm takes $O(k\log(p-1))$.

# 1c

The algorithm produces `False` since

$$g^{(p-1)/3} = 1 \text{ mod p}$$

# 2a

for i from 0..n-1:
$t = h^{3^{n-1-i}}$

if ($t = 1$): $x_i = 0$
if ($t = g^{3^{n-1}}$): $x_i = 1$

if ($t = g^{2\times 3^{n-1}}$): $x_i = 2$

$h = hg^{-x_i}$
g = $g^3$

# 2b

We look from 0 to n-1, which is n iterations.
In each iteration we are calculating $t = h^{3^{n-1-i}}$ which takes log($3^n - 1 - i$) multiplications using square and multiply technique. This is about O(n).
Calculating $g^{3^{n-1}}$ and $g^{2*3^{n-1}}$ is also O(n) for the same reason.

Everything else is constant time so we have O(1) time complexity for those operations.

Thus in total we have O(n) operations iterated n times, meaning our final time complexity is $O(n^2)$

# 2c

I got x = `26215193381549031346244544481960.

```
p = 20602102921755074907947094535687

g = 15074692835850319635499377698538

h = 19341277950553269760848569026015



n = 65

x = [0 for _ in range(n)]

g_prime = g

S  = pow(g, pow(3, n-1, p-1), p)

S2 = pow(S, 2, p)



for i in range(n):

    print("i =",i)

    t = pow(h,pow(3,n-1-i,p-1),p)
```

```
    if (t == 1):

        x[i] = 0

    if (t == S):

        x[i] = 1

    if (t == S2):

        x[i]= 2

    h = h * pow(g_prime, -x[i],p)

    g_prime = pow(g_prime,3,p)



s = 0

for i in range(65):

    s += x[i] * pow(3,i)

print(x)

print(s)
```

# 3a

d private key, (n public key, e public exponent)
x message

**Generate**
Choose two primes p,q. calculate n = pq.
Compute $\phi(n) = (p-1)(q-1)$
choose e coprime to $\phi(n)$
compute d = $e^{-1}$ mod $\phi(n)$

d is private key, (n,e) is public key
return (n,e)

**Signature** (x, d, n)
h = hash(m)
s = $h^d$ mod n
return x,s

**Verify** (x, s, n, e)
h = hash(x)
y = $s^e$ mod n
b = (h =? y)
accept if b=1, reject if b = 0

# 3b

This works because the x is hashed in the signature. When trying to forge, the signature and fake message must be linked through hashing rather than just through exponentiation. To break this, you must have hash(x) = $s^e$ which would require you to be able to calculate hash collisions.

# 3c

The drawback would be that EMSA uses a salt as well, meaning that a signing the same message twice would produce two different signatures while my hashing scheme results in the same signature every time.