# Problem 1

We can write a generic CRT solver that can, up to the limits of the chosen integer type (signed 64-bit integer in my implementation), iteratively solve a system of linear congruences:

$$\begin{aligned} x &= a_0 \mod m_0 \\ x &= a_1 \mod m_1 \\ &\dots \\ x &= a_{n-1} \mod m_{n-1} \end{aligned}$$

Solving a system of 1 congruence is trivial, since we can simply return $a_0$. Now that we have a base case, we can inductively build up the solution to the entire system: suppose we have a solution to the first $k$ congruences, denoted by $c_k$, then we can find a solution to the first $k+1$ congruences by making two observations.

First, $c_{k+1}$ must also be a solution to the first $k$ congruences, meaning that for some integer $y$:

$$c_{k+1} = c_k + (\Pi_{i=0}^{k-1} m_k) \cdot y \tag{1}$$

Second, $c_{k+1}$ must also satisfy the $k+1$'s congruence, meaning that:

$$c_{k+1} \equiv a_k \mod m_k \tag{2}$$

Combining the two equations above we have:

$$c_k + (\Pi_{i=0}^{k-1} m_k) \cdot y \equiv a_k \mod m_k$$

Which can be easily transformed to find a value of $y$:

$$y \equiv (\Pi_{i=0}^{k-1} m_k)^{-1} \cdot (a_k - c_k) \mod m_k$$

After that, we can plug $y$ back into the first obervsation and obtain a solution to the first $k+1$'s congruences.

The CRT solver's source code can be found at the end of this write up.

## a)

The solution is 31

## b)

The solution is 5764

## c)

The solution is 221

## d)

Note that all modulo arithmetics are well-defined, since addition, subtraction, and multiplication are always well-defined, and the multiplicative inverse is well-defined iff the two modulos are relatively prime. However, when the two modulos are not relatively prime, it is possible that the inverse does not exist. For example, the element $2 \in \mathbb{Z}_4$ has no inverse. We can use this two construct impossible congruences such as:

$$x \equiv 1 \mod 2$$
$$x \equiv 2 \mod 4$$

This system is impossible because the first congruence requires $x$ to be odd, but the second one requires $x$ to be even.

# Appendix

Source code for the CRT solver (written in Rust, btw):

```
/// subtract y from x within the input modulo
pub fn modulo_sub(x: i64, y: i64, modulo: i64) -> i64 {
    let x = x % modulo;
    let y = y % modulo;

    if x - y >= 0 {
        return x - y;
    }
    return x - y + modulo;
}

/// Returns (gcd, s, t) such that s*x + t*y = gcd is the Bezout identity
pub fn extended_gcd(x: i64, y: i64) -> (i64, i64, i64) {
    let (mut prev_r, mut r) = (x, y);
    let (mut prev_s, mut s) = (1, 0);
    let (mut prev_t, mut t) = (0, 1);

    while r != 0 {
        let q = prev_r / r;
        (prev_r, r) = (r, prev_r - q * r);
        (prev_s, s) = (s, prev_s - q * s);
        (prev_t, t) = (t, prev_t - q * t);
    }

    return (prev_r, prev_s, prev_t);
}

/// Attempt to find a multiplicative inverse of x (mod y). This is possible
iff
/// x and y are relatively prime. If no multiplicative inverse if possible,
/// return None
```

```rust
pub fn modulo_invert(x: i64, y: i64) -> Option<i64> {
    let (gcd, s, _t) = extended_gcd(x, y);
    if gcd > 1 {
        return None;
    }
    return Some(s);
}

#[derive(Debug)]
pub struct CRT {
    /// The modulo up to wihch the solution is unique. In teh context of
Chinese
    /// remainder theorem, it is the product of all modulos in all congruences
    modulo: i64,

    /// The solution to the set of congruences, unique up to self.modulo.
    sol: Option<i64>,
}

impl CRT {
    pub fn new() -> Self {
        return Self {
            sol: None,
            modulo: 1,
        };
    }

    /// Update the internal state to solve the union of the existing system
and
    /// the input congruence. Return the solution after the update
    pub fn add_congruence(
        &mut self,
        remainder: i64,
        modulo: i64,
    ) -> Option<i64> {
        if self.sol.is_none() {
            self.sol = Some(remainder);
            self.modulo = modulo;
        } else {
            let mut sol = self.sol.unwrap();
            let diff = modulo_sub(remainder, sol, modulo);
            let inverse = modulo_invert(self.modulo, modulo).unwrap();
            sol = sol + self.modulo * diff * inverse;
            self.modulo = self.modulo * modulo;
            self.sol = Some(modulo_sub(sol, 0, self.modulo));
        }
        return self.sol;
    }
}
```