Exercise 1

Based on Corollary 31.26, the equation $ax \equiv 1(\mathrm{mod}\, n)$ has a unique solution, module n if for any $n > 1, \gcd(a,n) = 1$, clearly then $x = a^{-1}(\mathrm{mod}(n))$.

From this we can conclude that when executing EXTENDED-EUCLID algorithm, we will obtain a solution for x, because EXTENDED-EUCLIDI algorithm outputs an answer in the form of $d = \gcd(a,n) = ax + ny$, and so when d=1 given a and n as input, we can just read off value for x, which will be our answer.

EXTENDED –EUCLID is polynomial in the size of the input.

Exercise 2

Algorithm:

Let us define $c_i = m_i(m_i^{-1} \bmod(n_i))$ for $i = 1,2,...,k$.This will always be defined since m_i and n_i are relatively prime and by Corollary 31.26 (in the book) we are guaranteed that $c_i = m_i(m_i^{-1} \bmod(n_i))$ always exists.

Given that we can compute *a* as follows:

$$a \equiv (a_1c_1 + a_2c_2 + ... + a_kc_k)(\mathrm{mod}\, n)$$

This equation holds because of the following:

$a \equiv a_ic_i(\mathrm{mod}\, n_i)$ [What we had]
$a \equiv a_i(m_i(m_i^{-1} \bmod(n_i)))(\mathrm{mod}\, n_i)$ [Substitute for c_i]
$(m_i(m_i^{-1} \bmod(n_i))) = 1 \bmod(n_i) = 1$ [From the definition of c_i]
$a \equiv a_i(\mathrm{mod}\, n_i)$ [What we needed to show]

Running time analysis:

We can compute c_i in time polynomial to the number of the bits (we can multiply and add anything (modulo) in time polynomial to the number of bits due to Euclidian algorithm which runs in polynomial time in the size of the bits)

Therefore, we can compute both c_i and a in *O(lg(b)).*

Exercise 3

Intuition:

We have an extra character at the end *"?"* which is supposed to match any character from the text *T*. So, when we are constructing our finite automata, we just modify the last char of the pattern, so that we do not check weather or not it matches the $char_q$ from the text (in other words, we always match whatever we see in the text with the last *char* we see in the pattern).

Modified finite automata builder:

Main Idea:

$m \leftarrow length[P]$
for $q \leftarrow 0$ *to* m
  do for each character $a \in \sum$
    if $(q = 0(m+1))$
      $\delta(q,a) \leftarrow (m+1)$
      $\delta(q,\varepsilon) \leftarrow (m+1)$
    else
      do $k \leftarrow \min(m+1, q+2)$
        repeat $k \leftarrow k-1$
          until $P_k \rceil P_q a$
        $\delta(q,a) \leftarrow k$
return δ

The main idea here is that when we are trying to match the last character of the patter, we match any character in the alphabet and transition to the next state. We do not even need the line $\delta(q,a) \leftarrow (m+1)$ because $\delta(q,\varepsilon) \leftarrow (m+1)$ already takes care of any input and make and transition to the next state.

Exercise 4

(a)

We know that we can construct an automation that accepts pattern of length *m* in $O\left(m\left|\sum\right|\right)$. So if we have two texts T_1 and T_2 we would simply need to construct *n* automations that accept all possible patterns of length *m* in T_1. This will take $O\left(nm\left|\sum\right|\right)$.

We can then combine these machines which we have constructed so that we have one machine instead of n (As I am sure there is a union algorithm).
Upon finishing the construction, we simply scan T_2 character by character advancing in our automation that we have constructed and if we end up in an accepting state then we have found a pattern which is in both T_1 and T_2.

(b)*

This turns out to be very simple to implement.

Rabin-Karp algorithm represents each string T as a sequence of digits.
We take that sequence of digits and compute values modulo *g* for each possible position of length *m.* (both *g* and *m* are specified at run-time)

We store these values in a hash table for the first text string. Next we compute the same for the second string and see if a digit matches for some modulo *g* in some possible position of length *m.* If this is the case then we do the real check *bit by bit* to see if we found a true match. If no such matches are found, then we are certain that no similar patterns of length *m* exist in both strings.

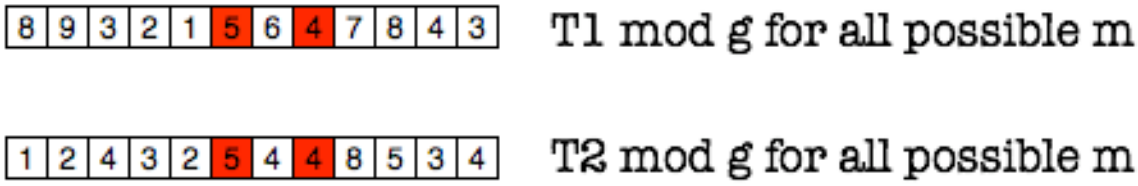

Fig 1: Here we see that after all *mods* were computed for all possible lengths *m*, we try to see if any match and in this case we have two matches.
So, what we would do now, is go back and determine if the digits from which the patterns were computed, really match.