1. Let \( m \) and \( n \) be relatively prime integers (they have no common factors other than 1). Prove, using the indirect method, the statement:

If \( mn \) is a square, then both \( m \) and \( n \) are squares.

(You can use the true statement: if \( r = p_1^{a_1} p_2^{a_2} \ldots p_k^{a_k} \) is the prime factorization of \( r \), then \( r \) is a square if and only if \( a_1, a_2, \ldots, a_k \) are all even.)

Assume that \( m \) and \( n \) are not both squares. Without loss of generality we can assume \( m \) is not a square (this means the proof would be essentially the same if we assumed that perhaps \( m \) was a square but \( n \) was not.)

By the parenthetical statement given, this means if \( m = p_1^{a_1} p_2^{a_2} \ldots p_k^{a_k} \) is the prime factorization of \( m \), then for some \( s \), \( a_s \) is odd.

Let \( n = q_1^{b_1} q_2^{b_2} \ldots q_l^{b_l} \) be the prime factorization of \( n \).

Since \( m \) and \( n \) are relatively prime, \( p_i \) not \( = q_j \) for all \( i, j \) in the ranges indicated. (Otherwise they would have that value as a common factor, contradicting the assumption.)

Consequently, the prime factorization of \( mn = p_1^{a_1} p_2^{a_2} \ldots p_k^{a_k} q_1^{b_1} q_2^{b_2} \ldots q_l^{b_l} \). Since \( a_s \) is odd, \( mn \) is not a square by the parenthetical statement.

We have thus proved that if \( m \) and \( n \) are not both squares, then \( mn \) is not a square. By the contrapositive of this, if \( mn \) is a square, then \( m \) and \( n \) are both squares, as was to be proved (i.e. QED).

Indirect method for a proof means assume that the conclusion is false and prove that the hypothesis is therefore false. Then use the contrapositive to conclude “if the hypothesis, then the conclusion.”
2. Use induction to prove the statement about positive integers.  
For all n, 3 divides \( n^3 - n \). (a divides b means that b = qa for some integer q.)

Base case: Let \( n = 1 \). \( n^3 - n = 1 \) and 3 divides 0.

Induction case: Assume 3 divides \( n^3 - n \). Consequently, \( n^3 - n = 3r \) for some integer \( r \) (definition of “divides”).

Consider \( (n+1)^3 - (n+1) = n^3 + 3n^2 + 3n + 1 - n - 1 \) (by multiplication)

\[= n^3 - n + 3(n^2 + n) \text{ (algebraic simplification)}\]

\[= 3r + 3(n^2 + n) \text{ (substitution)}\]

\[= 3(r + n^2 + n) \text{ (algebraic simplification)}\]

By definition, this means that 3 divides \( (n+1)^3 - (n+1) \).

We have proved that if 3 divides \( n^3 - n \), then 3 divides \( (n+1)^3 - (n+1) \).

By mathematical induction, for all \( n \geq 1 \), 3 divides \( n^3 - n \).

For an induction proof, the first step is the base case. Prove \( P(x) \) for a specific value of \( x = k \) (in this case 0 or 1). Here \( P(x) \) is the statement “3 divides \( x^3 - x \”).

Second, assume \( P(n) \) and from that assumption prove \( P(n+1) \). (Note, if you did not use \( P(n) \) in your proof of \( P(n+1) \), something is probably wrong.)

The simply say “By mathematical induction, \( P(n) \) for all \( n \geq k \).”

The proof is not complete without the last step.
3. Consider the following function.

```java
int IndexMin(apvector<apstring> list)
// precondition:  list.length() > 0
// postcondition: return n such that for all j, 0 <= j < list.length(),
//                list[n] <= list[j]
{
    int imin = 0;
    for(int k = 1; k < list.length(); k++){
        // loop invariant:  for all j, 0 <= j < k, list[j] >= list[imin]
        // and k <= list.length()
        if(list[k] < list[imin])
            imin = k;
    }
    return imin;
}
```

(a) Let P(n) be the loop invariant after the nth iteration:
For all j, 0 <= j < k = n, list[j] >= list[imin]
and k <= list.length()

Use induction to prove P(n) for all n <= list.length()

Assume list.length > 0.

Base case: n = 0, so we are at the beginning of the first time through the loop and imin = 0, k = 1.
In the invariant, the only value of j to be considered is j = 0. But since imin = 0, list[j] >=
list[imin] since they are equal. And k <= list.length. If k = list.length, then we stop the loop,
otherwise we continue.

Assume true for n (after the nth iteration). If n = list.length, then the loop exits and there is no n
+1 iteration to check. Otherwise:

During the (n+1)th iteration, the if checks if list[k_n] < list[imin_n], where imin_n is the value of imin
after the nth iteration. We have two cases: (1) this is true, or (2) this is false.

Case 1: since the condition is true, we execute the conditional statement and at the end of
the loop, imin = k_n. This will be imin_{n+1}. After the n+1 loop, we now have for 0 <= j < k_{n+1}, so
j < k_n or j = k_n < k_{n+1}. For the former (j < k_n), we have already that list[j] >= list[imin] > list[k_n]
= list[imin_{n+1}]. But for the latter, j = k_n, list[j] = list[k_n] = list[imin_{n+1}].
So for all j, 0 <= j < k_{n+1}, list[j] >= list[imin_{n+1}]).

Case 2: since the condition is false, we do not execute the conditional statement and imin_n
= imin_{n+1}. For all j, 0 <= j < k_{n+1} = k_{n+1}, either j < k_n or j = k_n. For the former, we already had
list[j] >= list[imin] = list[imin_{n+1}]. For the latter, j = k_n and in this case we have list[j] = list[k_n]
>= list[imin] = list[imin_{n+1}].
The invariant is true in either case. Since if the invariant is true for \( n \), it is true for \( n+1 \), by induction, it is true for all \( n \geq 0 \) and \( n \leq \text{list.length} \).

(b) Use (a) and the loop exit condition to prove the postcondition for the function.

The loop condition is \( k < \text{list.length} \), so the loop exits when \( k \geq \text{list.length} \). Since one part of the invariant says \( k \leq \text{list.length} \), we may conclude when the loop exits \( k = \text{list.length} \). The other part of the invariant says: for all \( j \), \( 0 \leq j < k \), \( \text{list}[j] \geq \text{list}[\text{imin}] \). By substitution, this becomes

For all \( j \), \( 0 \leq j < \text{list.length} \), \( \text{list}[j] \geq \text{list}[\text{imin}] \).

This is exactly the post-condition.

The proof of part (a) follows the pattern of an induction proof but uses the code to define what can happen during the loop. The use of a loop invariant like this depends on a proof that the loop invariant is always true after the \( n \)th iteration of the loop – until the loop exits. This is almost always an induction proof if done formally.

This technique is a small part of formal methods that allow us to prove the correctness of programs – that they do what they say they do. Of course, this assumes that the compiler correctly translates code into the equivalent machine language.