1. (a) V runs M on input x for C steps, which is a polynomial-time operation. The running time of V is O(C) since the number of steps C is an integer. So, V is efficient and its running time is polynomial in the size of the input. Also, V correctly verifies membership which is a valid verification process for L_{ACC}.

∴ V is a valid verifier and L_{ACC} is in NP since there exists an effecient verifier for it

(b) The algorithm F is relatively straightforward and consists of a loop that multiplies integers from 2 to n, which requires O(n) operations. The algorithm F is effecient and runs in polynomial time with respect to the input size n. So, First-Digit-Factorial is in P because it can be decided by an algorithm that runs in polynomial time.

2. (a) Verifier V for Ham-Cycle:

① Check if G1 is a finite number of vertices and edges

② Check if G1 has the same number of vertices as the number of nodes in the input Hamiltonian cycle

③ Verify that the input visits each node exactly once and ends at the starting node.

If all the above conditions are met, ACCEPT; otherwise, REJECT.

The verifier V checks if the given Hamiltonian cycle is indeed valid in G1, ensuring it visits each node exactly once and ends at starting node. If the conditions are satisfied, V correctly accepts the input. Also, V runs in polynomial time, making it efficient. Therefore, Ham-Cycle is in NP

(b) Verifier V for Graph-Isomorphism:

Given an input (<G>, <H>)

① Check if G and H are valid graphs.

② Check if G and H have the same number of vertices and edges . If not, reject

③ For each vertex u in G:

a. Find the vertex v in H that corresponds to u under the proposed bijection.

b. Check if the edges connected to u in G correspond to edges connected to v in H. If not, reject.

If all the above conditions are met, ACCEPT; otherwise, REJECT.

The verifier V checks if the given bijection between vertices preserve edge relationships and ensures that G and H have the same number of vertices and edges. If these conditions are satisfied, V correctly accepts the input. Also, each steps can be performed in polynomial time. Therefore, the verifier V runs in polynomial time, making it efficient. Since the V for Graph-Isomorphism is both correct and efficient, the language is in the class NP.

(c) Verifier V for Not-Prime:

Given an input n:

① Check if n less than 2. If it is, reject.

② Check if n is even and greater than 2. If it is, accept.

③ Check for divisibility by integers from 2 up to square root of n.

If n is divisible by any of these integers, accept; otherwise, reject.

V correctly accepts the input as not prime; otherwise, it rejects.

Also, each steps can be performed in polynomial time. Since the verifier V for Not-Prime is both correct and efficient, the language Not-Prime is in the class NP.

3. (a) Let's assume that we have algorithms that decide languages A and B, denoted by A_decider and B_decider, respectively. Since A and B are in P, these deciders run in polynomial time.

Now, we can define an algorithm to decide A ∪ B as follows:

Algorithm for A ∪ B:
On input w:
Run A_decider on w.
Run B_decider on w.
If A_decider accepts w or B_decider accepts w, ACCEPT; otherwise, REJECT.

The algorithm for A ∪ B runs A_decider and B_decider on the input w.
If either A_decider or B_decider accepts w, it means that w is in either A or B, or both, which is precisely what A ∪ B represents.
If either A_decider or B_decider accepts w, the algorithm accepts the input w. Otherwise, it rejects. The algorithm for A ∪ B runs in polynomial time because it essentially consists of two runs of algorithms that are known to be in P. As the running time of A_decider and B_decider is bounded by a polynomial function of the input size, the overall running time of the algorithm for A ∪ B is also polynomial.

Since we've shown that there exists a polynomial-time algorithm that decides A ∪ B, we can conclude that A ∪ B is in P when A and B are both in P.

(b) Let's assume that we have verifiers V_A for language A and V_B for language B. Both of these verifiers are capable of verifying membership in their respective languages A and B in polynomial time.

Verifier for A ∪ B:
Given an input string x:
Run verifier V_A on input x. If V_A accepts, then accept x.
If V_A rejects x, run verifier V_B on input x. If V_B accepts, then accept x. If V_B rejects, reject x.

The verifier for A ∪ B first checks if x is in language A by using the verifier V_A.
If V_A accepts x, it means x is in A, and therefore, it can be accepted for A ∪ B.
If V_A rejects x, it means x is not in A. In this case, the verifier proceeds to check if x is in B by using the verifier V_B.
If V_B accepts x, it means x is in B, and it can be accepted for A ∪ B.
If both V_A and V_B reject x, then x is not in either A or B, and it is rejected for A ∪ B.

If x is in A ∪ B, it means that x is in either A or B (or both). The verifier for A ∪ B checks both possibilities correctly. If x is in A, V_A will accept it, and if x is in B (but not in A), V_B will accept it. Therefore, the verifier ensures correctness.

The verifier for A ∪ B combines the verifiers V_A and V_B, both of which run in polynomial time. The verifier for A ∪ B performs a constant number of polynomial-time operations. Thus, it also runs in polynomial time.

Since we have shown that there exists a polynomial-time verifier for A ∪ B and it satisfies correctness and efficiency, we can conclude that A ∪ B is in NP when A and B are both in NP.

(c) Let's assume we have two algorithms, A_decider and B_decider, that decide languages A and B, respectively, and run in polynomial time.

Now, we can define an algorithm to decide AB as follows:

Algorithm for AB:
On input w:
For each possible split of w into two substrings a and b:
Check if a is in language A using A_decider.
Check if b is in language B using B_decider.
If both checks are successful (i.e., a is in A and b is in B), ACCEPT.
If no split of w satisfies the conditions in step 2, REJECT.

The algorithm for AB considers all possible splits of the input string w into two substrings a and b.
For each split, it checks if a is in language A using A_decider and if b is in language B using B_decider.
If both substrings a and b belong to their respective languages A and B, the algorithm accepts w.
If no split of w satisfies these conditions, the algorithm rejects w.

If w is in AB, it means there exist two strings a in A and b in B such that ab = w. The algorithm correctly checks each possible split of w and verifies that a is in A and b is in B. If such splits exist, the algorithm accepts, ensuring correctness.

The algorithm for AB considers all possible splits of the input string w, but since the lengths of the substrings a and b are bounded by the length of w, the number of splits is also bounded.

For each split, it checks membership in A and B using A_decider and B_decider, both of which run in polynomial time.
Therefore, the overall running time of the algorithm is polynomial in the size of the input w.
Since we have shown that there exists a polynomial-time algorithm to decide AB, we can conclude that AB is in P when A and B are in P.

4. (a) Given a language L in P, it means there exists a deterministic polynomial-time algorithm (decider) D_L that decides L in polynomial time.

We want to show that the complement /L is also in P. To do this, we need to define a deterministic polynomial-time algorithm, let's call it D_/L, that decides /L.

Algorithm D_/L for /L:

Given an input string w:
Run D_L on input w.
If D_L accepts w, REJECT; if D_L rejects w, ACCEPT.

In this algorithm, we use the decider D_L to decide the language L.
If D_L accepts the input w, it means that w is in L, and therefore, we reject it for /L (the complement).
If D_L rejects the input w, it means that w is not in L, and therefore, we accept it for /L.

If w is in L, then D_L will accept it, and D_/L will correctly reject it for /L.
If w is not in L, then D_L will reject it, and D_/L will correctly accept it for /L.

D_/L uses D_L, which is a polynomial-time algorithm. Therefore, the overall running time of D_/L is also polynomial.

Since we have shown that there exists a polynomial-time algorithm D_/L to decide the complement /L for any language L in P, we can conclude that P is closed under set complement. In other words, if a language is efficiently solvable in P, its complement is also efficiently solvable in P.

(b) (i) NP ⊆ coNP:

If P = NP, it implies that every problem in NP can be solved in polynomial time. In part (a), we proved that P is closed under set complement, meaning that for any language L in P, its complement /L is also in P.

Now, consider any language L in NP. By the definition of NP, there exists a non-deterministic polynomial-time algorithm (verifier) V_L that can efficiently verify whether a given input is in L. This verifier runs in polynomial time and decides whether the input is a "yes" instance of L.

Since P = NP, this implies that there also exists a deterministic polynomial-time algorithm D_L that can decide L. We can use D_L to efficiently verify whether a given input is not in L (i.e., in /L). In this case, /L is in P.

Hence, for any language L in NP, /L is also in P, which means that NP ⊆ coNP.

(ii) coNP ⊆ NP:

Similarly, if P = NP, it implies that every problem in NP can be solved in polynomial time. Now, consider any language /L in coNP. By the definition of coNP, there exists a non-deterministic polynomial-time algorithm (verifier) V_/L that can efficiently verify whether a given input is not in /L (i.e., in L).

Since P = NP, this implies that there also exists a deterministic polynomial-time algorithm D_/L that can decide /L. We can use D_/L to efficiently verify whether a given input is in L, which means L is in NP.

Hence, for any language /L in coNP, L is also in NP, which means that coNP ⊆ NP.

(c) In part (b), we established that if P = NP, then NP = coNP. Now, let's use the contrapositive of this statement to conclude that if NP ≠ coNP, then P ≠ NP.

Contrapositive: If P = NP, then NP = coNP.

Therefore, if NP ≠ coNP, then P ≠ NP.

This means that if the classes NP and coNP are not equal, which is a widely believed conjecture in computational complexity theory (i.e., NP ≠ coNP), then it implies that P and NP are also not equal (i.e., P ≠ NP).