Question One(1):

To reduce the given DFA, we can apply the state reduction algorithm. Starting with the initial states, we iteratively merge states that are indistinguishable until no more merges are possible. Let's go through the reduction steps:

Step 1: Group the states into accepting (final) and non-accepting (non-final) sets:
Accepting states: {q5}
Non-accepting states: {q1, q2, q3, q4, q6}

Step 2: Initialize the partition with the two sets obtained in Step 1:
Partition: {{q5}, {q1, q2, q3, q4, q6}}

Step 3: Iterate until no more merges are possible:
- For each partition set P, split it further based on the transitions for each symbol in Σ.
  - For symbol 'a':
    - {q5}: δ(q5, a) = q5 (Remains in the same set)
    - {q1, q2, q3, q4, q6}: δ(q1, a) = q5, δ(q2, a) = q5, δ(q3, a) = q5, δ(q4, a) = q5, δ(q6, a) = q2
      (The states in this set have different transitions, so we split it into two sets: {q5} and {q1, q2, q3, q4, q6})
  - For symbol 'b':
    - {q5}: δ(q5, b) does not exist (Remains in the same set)
    - {q1, q2, q3, q4, q6}: δ(q1, b) = q4, δ(q2, b) = q6, δ(q3, b) = q1, δ(q4, b) = q3, δ(q6, b) = q4
      (The states in this set have different transitions, so we split it into two sets: {q5} and {q1, q2, q3, q4, q6})

The partition now becomes: {{q5}, {q5}, {q1, q2, q3, q4, q6}}

Step 4: Remove duplicate sets from the partition:
Partition: {{q5}, {q1, q2, q3, q4, q6}}

Since there are no more merges possible, we have reached the reduced DFA with the following states:

Q' = {A, B}
A: Initial state
B: Accepting State

Σ = {a, b}

δ'(A, a) = A (Transition from A to A on 'a')
δ'(A, b) = B (Transition from A to B on 'b')
δ'(B, a) = A (Transition from B to A on 'a')
δ'(B, b) = A (Transition from B to A on 'b')

The reduced DFA accepts strings that have an odd number of 'b' symbols.

Question Two(2)(a):

The regular expression R(0)44 for the language L(0)44 can be specified as follows:

R(0)44 = b(a+b+c)*b + b(a+b+c)*a(a+b+c)*b + b(a+b+c)*c(a+b+c)*b

This regular expression represents the set of all strings that start with 'b', followed by any combination of 'a', 'b', and 'c', but without visiting any state numbered larger than 0 (excluding the first and last states), and finally ending with 'b' in state q4.

Question Two(2)(b):

The regular expression R(0)12 for the language L(0)12 can be specified as follows:

R(0)12 = a(a+b+c)*b

This regular expression represents the set of all strings that start with 'a', followed by any combination of 'a', 'b', and 'c', without visiting any state numbered larger than 0 (excluding the first and last states), and finally ending with 'b' in state q2.

Question Two(2)(c):

The regular expression R(4)12 for the language L(4)12 can be specified as follows:

R(4)12 = ε + a(a+b+c)*b

This regular expression represents the set of all strings that go from state q1 to state q2 without visiting a state numbered larger than 4 (excluding the first and last states). It includes the empty string (ε) and strings that start with 'a', followed by any combination of 'a', 'b', and 'c', and finally end with 'b' in state q2.

Question Two(2)(d):

The regular expression R(0)11 for the language L(0)11 can be specified as follows:

R(0)11 = ε + a(a+b+c)*a

This regular expression represents the set of all strings that go from state q1 to state q1 without visiting a state numbered larger than 0 (excluding the first and last states). It includes the empty string (ε) and strings that start and end with 'a', with any combination of 'a', 'b', and 'c' in between.

Question Two(2)(e):

The regular expression R(4)11 for the language L(4)11 can be specified as follows:

R(4)11 = ε + a(a+b+c)*a(a+b+c)*

This regular expression represents the set of all strings that go from state q1 to state q1 without visiting a state numbered larger than 4 (excluding the first and last states). It includes the empty string (ε) and strings that start and end with 'a', with any combination of 'a', 'b', and 'c' in between.

Question Three(3)(a):

The regular expression for the language "The set of all strings with at most one pair of consecutive 1's" can be expressed as:

(0*1:0*)*

Explanation:
- (0*1:0*) represents a sequence of zeros and ones, with an optional single 1 (represented by 1:) surrounded by any number of zeros (represented by 0*).
- The outermost * allows for repetition of this pattern any number of times, including zero times, allowing for strings with at most one pair of consecutive 1's.

This regular expression generates the desired language because:
- The initial (0*1:0*) ensures that the string can start with any number of zeros (0*) followed by an optional single 1 (:1) and then any number of zeros (0*).
- The outermost * allows for this pattern to repeat any number of times, including zero, capturing strings with no consecutive 1's, a single pair of consecutive 1's, or multiple non-consecutive 1's.
- Overall, the regular expression ensures that there are at most one pair of consecutive 1's in the string.

Question Three(3)(b):

The regular expression for the language "The set of all strings not containing '010' as a substring" can be expressed as:

(0+1)* (1+01+001)*

Explanation:
- (0+1)* matches any combination of zeros and ones, including empty strings.
- (1+01+001)* matches any combination of '1', '01', or '001' repeated zero or more times, allowing for any sequence of ones and zeros that doesn't contain '010' as a substring.
- Combining the two expressions, (0+1)* (1+01+001)* generates the language of all strings not containing '010' as a substring.

This regular expression generates the desired language because:
- The first part, (0+1)*, allows for any sequence of zeros and ones at the beginning of the string, including an empty string.
- The second part, (1+01+001)*, matches any combination of '1', '01', or '001' repeated zero or more times, allowing for any sequence of ones and zeros that doesn't contain '010' as a substring.
- By combining the two parts, the regular expression ensures that the entire string doesn't contain '010' as a substring, covering all possible combinations of zeros and ones that satisfy this condition.

Question Four(4)(a):

To remove the useless productions from the given grammar, we need to identify the non-terminals and terminals that cannot be reached from the start symbol.

The start symbol is S. Let's analyze each production rule:

1. S → a
   - 'a' is a terminal symbol, and it can be derived from the start symbol S. This production is reachable.

2. S → aA
   - 'a' is a terminal symbol, and it can be derived from the start symbol S. This production is reachable.
   - A is a non-terminal. We need to check its reachability.

3. S → B
   - B is a non-terminal. We need to check its reachability.

4. S → C
   - C is a non-terminal. We need to check its reachability.

5. A → aB
   - 'a' is a terminal symbol, and it can be derived from the start symbol S. Therefore, A is reachable.
   - B is a non-terminal. We need to check its reachability.

6. A → λ
   - λ represents an empty production. It is always reachable.

7. B → Aa
   - A is a non-terminal, and we have determined it is reachable.
   - 'a' is a terminal symbol, and it can be derived from the start symbol S. Therefore, B is reachable.

8. C → cCD
   - 'c' is a terminal symbol, and it can be derived from the start symbol S. This production is reachable.
   - C is a non-terminal. We need to check its reachability.
   - D is a non-terminal. We need to check its reachability.

9. D → ddd
   - 'd' is a terminal symbol, and it can be derived from the start symbol S. Therefore, D is reachable.

10. D → Cd
    - C is a non-terminal, and we have determined it is reachable.

- 'd' is a terminal symbol, and it can be derived from the start symbol S. Therefore, D is reachable.

Based on the analysis, the reachable non-terminals are S, A, B, C, and D. Therefore, all the production rules in the grammar are reachable, and no useless productions need to be removed.

Question Four(4)(b):

To remove the λ (epsilon) productions from the given grammar, we need to eliminate any production rules that derive the empty string. Let's analyze each non-terminal:

1. S → a
   - This production does not contain λ, so no changes are needed.

2. S → aA
   - This production contains A, which has a λ production. We can replace A with its productions in S.
   - Updated production: S → a | aa

3. S → B
   - This production does not contain λ, so no changes are needed.

4. S → C
   - This production does not contain λ, so no changes are needed.

5. A → aB
   - This production does not contain λ, so no changes are needed.

6. A → λ
   - This is a λ (epsilon) production. We need to eliminate it.

7. B → Aa
   - This production contains A, which has a λ production. We can replace A with its productions in B.
   - Updated production: B → aa | a

8. C → cCD
   - This production does not contain λ, so no changes are needed.

9. D → ddd
   - This production does not contain λ, so no changes are needed.

10. D → Cd
    - This production does not contain λ, so no changes are needed.

After removing the λ production(s), the updated grammar is:

S → a | aa | B | C,
A → aB
B → aa | a
C → cCD
D → ddd | Cd

Note: It's important to mention that the removal of λ productions may affect the language generated by the grammar.

Question Four(4)(c):

To simplify the given grammar by removing unit productions and eliminating any useless productions, we follow the steps below:

Step 1: Remove unit productions
A unit production is a production of the form A → B, where A and B are non-terminals. We replace the unit production with all the productions of B.

After removing unit productions:

S → a | aA | B | C,
A → aB | λ
B → aB | aa | a
C → cCD
D → ddd | Cd

Step 2: Remove useless productions
A useless production is a production that cannot be reached from the start symbol S. To identify and remove useless productions, we need to find the reachable non-terminals.

Starting with the start symbol S, we traverse the grammar to find all reachable non-terminals. We mark each non-terminal that we encounter. After traversing, any non-terminal that is not marked is considered unreachable.

From the given grammar, we start with S and find the reachable non-terminals:

Reachable non-terminals: S, A, B, C, D

Step 3: Remove unused non-terminals and their productions
We remove any non-terminals and their productions that are not reachable from the start symbol S.

After removing unused non-terminals and their productions:

S → a | aA | B | C,
A → aB | λ

B → aB | aa | a
C → cCD
D → ddd | Cd

The resulting simplified grammar has the same language as the original grammar but without the unit productions and any useless productions.

Question Five(5):

To write a right-linear grammar for language L using at most 3 variables (S, A, B), we can start by defining the productions:

1. S → aA | B | ε
2. A → aB | ε
3. B → cD
4. D → dD | ε

Here, 'a', 'c', and 'd' are terminals, and ε represents the empty string.

To convert this regular grammar to a finite automaton (FA) that recognizes language L, we can follow these steps:

1. Create states for each variable and terminal in the grammar. In this case, we have states S, A, B, and D for variables, and terminals 'a', 'c', and 'd'.
2. Set S as the initial state.
3. Create transitions from one state to another based on the productions of the grammar.
   - From S:
     - If S → aA, create a transition from S to A on input 'a'.
     - If S → B, create a transition from S to B on input ε (empty string).
     - If S → ε, create a transition from S to the final/accepting state on input ε.
   - From A:
     - If A → aB, create a transition from A to B on input 'a'.
     - If A → ε, create a transition from A to the final/accepting state on input ε.
   - From B:
     - If B → cD, create a transition from B to D on input 'c'.
   - From D:
     - If D → dD, create a transition from D to D on input 'd'.
     - If D → ε, create a transition from D to the final/accepting state on input ε.
4. Mark the final/accepting state(s) in the FA.

The resulting FA will have transitions corresponding to the productions of the grammar and will recognize the language L described by the right-linear grammar.