1. Write a code to generate constrained random numbers in Verilog.

```
module constrained_random();
   // Declare an array for the constrained range
   int values[0:9]; // Example range 0-9
   int rand_value;

   initial begin
      // Initialize the array
      for (int i = 0; i < 10; i++) begin
         values[i] = i;
      end

      // Shuffle the array for randomization
      for (int i = 9; i > 0; i--) begin
         int j = $urandom_range(0, i);
         // Swap values[i] and values[j]
         int temp = values[i];
         values[i] = values[j];
         values[j] = temp;
      end

      // Print out shuffled values
      foreach (values[i]) begin
         $display("Value %0d: %0d", i, values[i]);
      end
   end
endmodule
```

2. How can you be sure that the constrained random stimulus has covered all the values in range without repetition in a cyclic random fashion? Explain with an example.
   By shuffling the values in an array and accessing them sequentially, you can ensure that all values are covered without repetition. The above example achieves this with a single cycle through the shuffled array.

```
module constrained_random_example();
   // Define the range and array to hold the values
   int values[0:9]; // Example range: 0-9
   int rand_value;
   int index = 0;

   initial begin
      // Initialize the values array
      for (int i = 0; i < 10; i++) begin
         values[i] = i;
      end

      // Randomly shuffle the array
      shuffle_array();

      // Generate random stimulus
      for (int i = 0; i < 10; i++) begin
```

```
            // Access the shuffled value
            rand_value = values[index];
            $display("Random Value: %0d", rand_value);
            index++;

            // Reshuffle after exhausting the array
            if (index == 10) begin
                index = 0; // Reset index
                shuffle_array(); // Reshuffle for the next cycle
            end
        end
    end

    // Task to shuffle the array
    task shuffle_array();
        for (int i = 9; i > 0; i--) begin
            int j = $urandom_range(0, i);
            // Swap values[i] and values[j]
            int temp = values[i];
            values[i] = values[j];
            values[j] = temp;
        end
    endtask
endmodule
```

3. Write a code to change the sequence of constrained random stimulus?

```
for (int i = 0; i < 10; i++) begin
    // Use a different randomization strategy
    int j = $urandom() % (i + 1);
    // Swap logic remains the same
end
```

4. What is a weighted random stimulus? Explain with an example.

Weighted random stimuli allow certain values to appear more frequently than others. You can create a weighted distribution using a selection process based on defined weights.

```
module weighted_random();
    // Define weights
    int weights[0:4] = {1, 2, 3, 4, 5}; // Higher weight means more likely to be chosen
    int total_weight = 0;

    initial begin
        foreach(weights[i]) total_weight += weights[i];

        // Generate a random number based on weights
        int random_num = $urandom() % total_weight;
        int chosen_value = -1;
        int cumulative_weight = 0;

        for (int i = 0; i < 5; i++) begin
            cumulative_weight += weights[i];
            if (random_num < cumulative_weight) begin
                chosen_value = i;
                break;
            end
        end

        $display("Weighted Random Value: %0d", chosen_value);
    end
endmodule
```

5. What metrics help in defining the completeness of the random stimulus?
    - Coverage Metrics: Measures how many unique values or combinations have been generated.
    - Functional Coverage: Ensures that all desired behaviors and states are exercised.
    - Code Coverage: Ensures that all parts of the testbench code have been executed.
6. What are some stimulus generation techniques when the stimulus is not reproducible using BFMs? Explain using Verilog examples.

When the stimulus is not reproducible, you can use techniques like:

a. Using a Random Number Generator: Generate sequences based on system states.
b. Generating stimuli from external data sources (like files).

   Example:

```
module stimulus_generator(input clk, output reg [7:0] data);

    initial begin

        forever begin

            #10 data = $urandom_range(0, 255); // Generate random data

        end

    end

endmodule
```

7. What is SDF back-annotation, and how is it implemented in Verilog testbench?
   SDF (Standard Delay Format) back-annotation involves providing timing information to the simulation to accurately reflect the timing behaviour of the design.

```
`timescale 1ns/1ps
module testbench;
    initial begin
        $sdf_annotate("design.sdf", top_module_instance);
        // Your simulation code here
    end
endmodule
```

8. What is the difference between unit delay and full-timing simulations?

Unit Delay Simulation: Assumes a single delay unit for signal propagation, simplifying the timing analysis but may not capture actual delays.
Full Timing Simulation: Considers all delays specified in the design (setup, hold, and propagation delays) for a more accurate representation of the design's timing behaviour

9. My gate simulation is not passing, some tests hang. What are the key points to look for?
    - Signal Misconnections: Ensure all signals are connected properly.
    - Clock and Reset: Check if the clock is toggling and the reset is functioning correctly.
    - Infinite Loops: Verify FSMs and combinatorial logic for conditions that may lead to infinite loops.
    - Resource Conflicts: Look for shared resources that may lead to contention.
10. Using a synchronous FSM approach, design a circuit that takes a single bit stream as an input at the pin in. An output pin match is asserted high each time the pattern 10101 is detected. A reset pin initialises the circuit synchronously. Input pin clk is used to clock the circuit. Apply constrained randomised stimulus to this design for verification.

```
module fsm_detector(
  input clk,
  input reset,
  input din,
  output reg match
);
  typedef enum logic [2:0] {
    S0, S1, S2, S3, S4
  } state_t;

  state_t state, next_state;

  always_ff @(posedge clk or posedge reset) begin
    if (reset)
      state <= S0;
    else
      state <= next_state;
  end

  always_ff @(posedge clk) begin
    match <= (state == S4);
  end

  always_comb begin
    case (state)
      S0: next_state = din ? S1 : S0;
      S1: next_state = din ? S1 : S2;
      S2: next_state = din ? S3 : S0;
      S3: next_state = din ? S4 : S2;
      S4: next_state = din ? S1 : S0;
      default: next_state = S0;
    endcase
  end
endmodule
```