The output of the given code will be:

```

1

4

2

3

```

Now, let's understand why this sequence of output occurs:

1. `console.log(1);`: This statement logs the number `1` to the console immediately.

2. `setTimeout(printNumber, 0, 2);`: This sets up a timer to call the `printNumber` function with the argument `2` after a delay of 0 milliseconds. Although the delay is specified as 0, the function will still be executed asynchronously after the current synchronous code finishes executing.

3. `setTimeout(printNumber, 100, 3);`: This sets up another timer to call the `printNumber` function with the argument `3` after a delay of 100 milliseconds. This means that the function will be executed approximately 100 milliseconds after the current synchronous code finishes executing.

4. `console.log(4);`: This statement logs the number `4` to the console immediately.

Now, let's understand the sequence of execution:

- `1` is logged by the first `console.log(1);` statement immediately.

- `4` is logged by the second `console.log(4);` statement immediately after the first one.

- `2` is logged by the `printNumber` function with the argument `2`. Even though the `setTimeout` delay is 0 milliseconds, it is still executed asynchronously after other synchronous tasks are completed. Since the current code execution is finished (all synchronous code), the `printNumber(2)` gets pushed to the end of the event loop queue. Then, it gets executed, and `2` is logged to the console.

- `3` is logged by the `printNumber` function with the argument `3`. This `setTimeout` has a delay of 100 milliseconds, so it will be executed approximately 100 milliseconds after the current code finishes executing. Since the `printNumber(3)` is pushed to the event loop queue after the other tasks, it gets executed after the previous tasks are completed, and `3` is logged to the console.

Hence, the final output sequence is `1`, `4`, `2`, and `3`.