COMP375 Project 03: Iterative DNS Resolver

Document Last Updated: March 20, 2024 @ 11:03 PM

Assignment Deadline: Wednesday, April 3 @ 10PM

Difficulty Rating (out of 5): ★★★★

You are expected to work with one other student on this assignment. The scope of the project is too large for one person to reasonably complete.

Please see the syllabus for guidance on what constitutes acceptable partnering practices.

Further note that the grading system for this class provides flexibility about which projects you choose in order to earn the grade you are aiming for. For this reason, I will only set up a repository for you *after* you send me a CampusWire DM stating that you plan on doing this project. Please review the syllabus (or specs grading table) about the requirements for the grade level you are aiming for.

1. Overview

For this project you will be implementing the DNS protocol to build your very own iterative DNS query resolver.

Your program will receive one optional flag (`--mx`) and one required argument: the host name the user would like to resolve. If the flag is absent, the program should find the hostname's IP address. If the flag is present, the program should find the name (not the IP address) of mail exchange for a domain. For example, to make a basic DNS query:

```
python3 resolver.py home.sandiego.edu
```

The previous command will print out the following.

```
IP address for home.sandiego.edu: 192.195.155.48
```

Below is an example query for a mail server.

```
python3 resolver.py --mx san.rr.com
```

This should output the following.

```
The mail exchange for san.rr.com resolves to: pkvw-mx.msg.pkvw.co.charter.net
```

You may also use the `--verbose` to have the program print additional information while running. This will be useful for debugging.

You should assume that there will be a file named `root-servers.txt` in your program's current working directory and that it contains a list of IP addresses for root DNS servers. Your program **must** use this file to find a root server. It should iteratively work its way down the DNS hierarchy, querying a root server, then a top-level domain (TLD) server, and finally an authoritative server(s), where it can resolve the requested host name.

2. Learning Objectives

1. Explain the role of each of the following types of DNS records: A, CNAME, NS, and MX.
2. Create and send iterative DNS queries and interpret the resulting DNS response.
3. Use UDP-based sockets to send data to remote hosts.
4. Set socket options such as the amount of time before "timing out."
5. Program sockets in a language other than C/C++.

3. Getting Started

The starter code resides in a Git repository on the `code.sandiego.edu` server. To access the files, use Git to clone the repository, as shown below. Do not forget to send me a CampusWire DM confirming that you are working on the project (and with whom you would like to work).

```
cd ~/comp375/projects/


# replace the "X" below with your group number
git clone git@code.sandiego.edu:comp375-sp24-p03-groupX p03
```

This should create a new directory called `p03` with the following starter files:

1. **`resolver.py`**: Source file that you will use to implement your solution.
2. **`helpers.py`**: Source file with a few classes and functions that you will use to help implement your solution.
3. **`root-servers.txt`**: Contains a list of IP addresses for the DNS root servers (one IP per line).
4. **`test_{locate_answer,parsing,query_servers,resolver}.py`**: Unit tests to check the correctness of your implementation. You should not modify this file in any way.
5. **`test/`**: A directory with binary data used by the unit tests. You should not modify the files in this directory.
6. **`run-tests.sh`**: A script to run your program in several inputs.
7. **`project03.txt`**: Your project report file.

For this project you are **required** to use Git to track your development. You should regularly add and commit changes you make while you are working on this project. No submissions will be accepted that do not have a reasonable number of commits to your Git repository. Furthermore, it is expected that you and your partner will have a roughly equal amount of commits.

WARNING: Your code will be tested on the BEC315DL machines. If you work on your own computer, you should try compiling and running on one of the BEC315DL machines. Please be aware that you will not meet the project requirements if your program doesn't work on the BEC315DL machines, even if it works on your own machine.

4. Requirements

1. You must NOT use any libraries that simplify DNS or hide the details of socket programming! For example, Python's `socket.gethostbyname` function will get the IP address of a hostname, using that would completely negate the learning experience of this project. If you have any doubt about which functions you may use, please ask!

2. If you attempt to query a server and get no response after waiting a short time (approximately 5 seconds), your program should move on and attempt to query the next server.

3. By default, your program should request Type A records (i.e. host name to IP address). If the optional "`--mx`" flag was given when running the program, you should request a Type MX record (i.e. host name to mail server mapping). When generating a "Question" for the DNS query, note that the Type A has a "Type" value of 1 while MX has a "Type" value of 15.

4. You should **NOT** use Python's `print` function in the code you write. Instead, you should use Python's `logging` library. I strong recommend reading through the first couple of sections (including "When to use logging" and "A simple example") for a basic understanding of the logging library.

   For example, instead of `print("Checking for domain", hostname)` you would use something like the following.

   ```
   logging.info(f"Checking for domain {hostname}")
   ```

   The starter code has it set up so that all logging printouts are saves to a file named `output.log`. By default, no logging messages are printed to the screen while running the program, but adding the `--verbose` option will include the debugging output.

5. Your program should log its intermediate steps as it traverses the DNS hierarchy. The starter code logs a message each time the `resolve` function is called: don't change or remove that.

6. You should **never** ask a DNS server to perform a *recursive query* for you.

7. Your code **must** not modify the function signature for any of the starter code functions. By that, I mean it should not add, remove, or change parameters, not should it change the type of value returned.

8. You can *and should* write additional functions to logically split up the work of the program. Those functions **must** include type hints for **all** parameters as well as the return value. They also **must** include a proper Python docstring comment, using the exact style and placement demonstrated in the functions given in the starter code.

5. Development Plan

This section will walk you through the steps required to complete your implementation of the DNS resolver program. Note that most steps build upon one another, so you should plan on following this in the exact order presented.

Before you get started, you should make sure you understand the basic format of a DNS response message, including the header fields and various sections that follow it. You can review this by checking out Section 2.4 of your textbook.

5.1. Phase 1: Parsing the Response

For the first phase, you will be working on parsing the response you would get from a DNS server. The goal will be to get it into a format that will make it easier for you to answer the user's request DNS query.

5.1.1. STARTING THE `parse_response` FUNCTION

The first function you will be working on is the `parse_response` function. Locate this function in `resolver.py` and read through its docstring comment, noting the arguments and their types as well as what it returns. Note that the "Optional" type means that the value can either be the specified type or `None`; for example `Optional[str]` means that it could be an actual string (like "hello") or `None`.

The `parse_response` function will take the binary encoded DNS response and create a `DNSResponse` object that stores information about the original query (the hostname and type of query) as well as the individual DNS records in each of the sections of the response (Answers, Authority, and Additional/Others).

The following code demonstrates how you would create a `DNSResponse` object, with one record per section.

```
response_info = DNSResponse("google.com", DNSRecordType.A)

# create a A type record mapping google.com to the ip address 162.16.25.18
answer_record = DNSRecord("google.com", DNSRecordType.A, [162,16,25,18])

# add this record to the Answers section
response_info.answers.append(answer_record)

# create NS record and add it to Authority section
authority_record = DNSRecord("google.com", DNSRecordType.NS, "ns1.google.com")
response_info.authorities.append(authority_record)

# create a AAAA and add it to the Additional/Other section
additional_record = DNSRecord("google.com", DNSRecordType.AAAA,
        [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16])
response_info.others.append(additional_record)

print(response_info)
```

Take the time now to review the `DNSRecord`, and `DNSResponse` classes in the `helpers.py` file, focusing on what fields and methods are defined in these classes.

The very first step in coding should be to modify the `DNSRecordType` enum to support the following types of records: NS, MX, and SOA. Use Table 166 on this webpage to find value you should assign each of these new types in your updated `DNSRecord` enum. For example, notice how the RR Type value associated with CNAME is 5, which is what it is assigned to in the starter code. (Note that you'll also see references to PTR and TXT types; ignore those.)

Next, it will come time to start processing the response message itself. The starter code provides some hints on the basic steps required for implementing the `parse_response` record. First, you'll need to get the header information using the `struct.unpack` function.

The following code demonstrates how you would get the first two header fields from the DNS response message.

```
query_id, query_flags = struct.unpack("!HH", response[0:4])
```

The `unpack` function takes two parameters, the first being a string representing the format and size of the binary data, and the second being the binary data itself. (Note that Python uses the `bytes` type to represent binary data: that is the type of the `response` parameter of `parse_response`.) This example is saying that the data is in network format (the "!" part) and that it contains two separate 2-byte unsigned short int values (the "HH" part). Because this part of the header will be at the very beginning of the response and will be 4 bytes total, the second parameter includes the range of the total response to use as 0 to 4.

Have a look at the Format Character section of the Python documentation and identify which character (similar to "H") you would use to unpack a 1-byte unsigned char. You won't need to use this immediately, but you should know where to look when you do need to use it later in this project.

Take a moment now to think about how many total fields there are in the DNS message header, what they are, and how many bytes (*not bits*) each one uses. The textbook is a good resource for this, but a more detailed view of this information is found on this webpage; Figure 248 (DNS Message Header Format) might be particularly useful for you.

Now think about how you will need to update the example above to handle all the headers, not just the two in the example. Update the code in `parse_response` to do this unpacking of the header.

Recall that one important step for verifying the authenticity of a DNS response is to check that it's ID matches the ID you used in the query. In your code, check for a mismatch in the response's query ID and the expected query ID (a parameter to `parse_response`) and return `None` if there is a mismatch.

Next up is step 2 of `parse_response`: handling the Question section. Check out this webpage to review the role of the Question section and find it's exact format.

The first thing in the Question section is the name, which will be a variable sized string (e.g. "www.sandiego.edu").

What are the other fields in this section and what sizes are they? Answer the following and write this down somewhere: What are the fields in the question section and what is each of their sizes?

Note that the question name is encoded using a special DNS Name Notation which you can read about on this webpage. Luckily, you don't have to worry about decoding that special notation: the given `decode_dns_name` function will do that for you.

What are the parameters to the `decode_dns_name` function and what does this function return?

Now work on implementing Step 2 of `parse_response` by first calling the `decode_dns_name` function to get the name, then using `struct.unpack` to extract the Question fields that come immediately after the name. You'll need to think carefully about the parameters of the `decode_dns_name` as well as the index in the response where the following two fields start.

You can now start by creating the `DNSResponse` object using the name and type you just extracted from the question. Use the code example above to remind yourself how to create a `DNSResponse` object.

5.1.2. Parsing Resource Records with `parse_record`

Having parsed the Question section of the response, it is now time to parse the resource records in the Answer, Authority, and Additional sections. All of these records will have the same general format, as described on this webpage. Take a moment to make a note of all the resource record fields and the size of each of these. You will also likely want to refer to the webpage on the A, CNAME, and NS record data format as well as the webpage on SOA and webpage on MX (and other) formats. Note that the AAAA record format is similar to the A record, except that it has 16-bytes for the address instead of 4. Also note that for the SOA record you should use the "Master Name" as the value in your `DNSRecord`.

Rather than parsing the records directly in `parse_response`, you should do that in the function named `parse_record`. Locate that function in the starter code and review the docstring comment, focusing on the arguments and return value(s) of this function.

Implement the `parse_record` function now, relying on the `decode_dns_name` and `struct.unpack` functions to help you along the way.

5.1.2.1. Testing the `parse_record` Function

Once you are done implementing `parse_record`, you can test that your implementation is correct by running the `test_parsing.py` file. This file contains a set of unit tests implemented using the `pytest` Python module. To run it, either use you IDE's "run" button or type the following command into a terminal.

```
python3 test_parsing.py
```

At this point, all of the `test_parse_X_record` tests should pass. The other tests won't, because they are testing the incomplete `parse_response` function.

Once all the appropriate tests pass, you may move on to the next step.

5.1.3. Finishing the `parse_response` Function

Next, we can go back to completing the `parse_response` function. All that is left to do for that function is step 3, which involves handling all of the resource records. Now that you have a working `parse_record` function, this step should mostly be trivial. Just don't forget to append each record to the appropriate list inside of the `DNSResponse` object you created in Step 2. (Again, the earlier code example will guide you in appending to the different lists.)

5.1.3.1. Testing the `parse_response` Function

Once you are done with Step 3, you can now test your `parse_response` using the `test_parsing.py` file as before. As this point, 100% of the tests should pass. If they do, you are done with Phase 1 and ready to move to the next phase.

5.2. Phase 2: Interpreting Responses

Thanks to the work you did in Phase 1, you are now able to take a DNS response message and parse it into a Python object (`DNSResponse`). In this next phase, we will work on interpreting the `DNSResponse` object to determine the answer to a user's DNS query.

The focus of this phase will be on the `locate_answer` function. Locate that function in the starter code and note what it is supposed to do, its arguments (and their types) and the

type of the response.

Before we attempt to implement the `locate_answer` function, we need to understand how it should handle specific types of responses. Once you have a firm understanding of those scenarios, you can go about sketching out a flowchart that will guide how you implement the logic of the `locate_answer` function.

5.2.1. Response Scenarios

In this section, you will be presented with some DNS response messages (gathered using the `dig` program) and asked to determine the next action(s) to take. Each of these is one step in the iterative DNS resolution process. By determining the next actions to take, you will be able to know what the next step to perform in the iterative process.

The options for actions are as follows.

1. Get the answer directly from the response.
2. Give up (i.e. the query doesn't have a valid answer).
3. Perform a **new** query to get the IP address of a specific hostname. The exact hostname should be given, and should be different from the hostname of the current query.
4. Send the **same** query to a different server with a specific IP address. This should include the exact IP address of the server (if it is known) or a description of where that IP address would come from.

Note that the first two of these are terminal steps in the iterative process, while the final two will require more steps to be performed in order to reach a final resolution.

5.2.1.1. Scenario 1

In the following scenarios, a request for the IP address (i.e. the A record) for www.sandiego.edu was sent to the server with the IP address 205.251.192.39.

```
; <<>> DiG 9.10.6 <<>> @205.251.192.39 www.sandiego.edu A
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 52540
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 4, ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;www.sandiego.edu.      IN  A

;; ANSWER SECTION:
www.sandiego.edu.   300 IN  A   192.195.155.200

;; AUTHORITY SECTION:
sandiego.edu.       900 IN  NS  ns-1509.awsdns-60.org.
```

```
sandiego.edu.       900 IN  NS  ns-1664.awsdns-16.co.uk.
sandiego.edu.       900 IN  NS  ns-39.awsdns-04.com.
sandiego.edu.       900 IN  NS  ns-872.awsdns-45.net.
```

Here we see that the answer to the question (www.sandiego.edu - A record) was given in the ANSWER section so the next action would be to return the answer of “192.195.155.200”.

5.2.1.2. Scenario 2

In the following scenarios, a request for the IP address (i.e. the A record) for evil.sandiego.edu was sent to the server with the IP address 205.251.197.229.

```
; <<>> DiG 9.10.6 <<>> @205.251.197.229 evil.sandiego.edu
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 39886
;; flags: qr aa rd; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;evil.sandiego.edu.     IN  A

;; AUTHORITY SECTION:
sandiego.edu.              900   IN    SOA   ns-1664.awsdns-16.co.uk.   awsdns-
hostmaster.amazon.com. 1 7200 900 1209600 86400
```

As the response indicates, there is no answer. Looking at the AUTHORITY section, there are no NS records, only an SOA (“Start of Authority”) record. Your book doesn’t talk about this, but in this context it is used to indicate that the query doesn’t have a valid answer. That’s because `evil.sandiego.edu` doesn’t exist (no word on whether `good.sandiego.edu` exists).

In this scenario, we should therefore give up and indicate there is no answer to this question.

5.2.1.3. Scenario 3

Consider the following DNS response message.

```
; <<>> DiG 9.10.6 <<>> @199.19.57.1 ns-1509.awsdns-60.org A
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 25365
;; flags: qr rd; QUERY: 1, ANSWER: 0, AUTHORITY: 4, ADDITIONAL: 9
```

```
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1232
;; QUESTION SECTION:
;ns-1509.awsdns-60.org.     IN  A

;; AUTHORITY SECTION:
awsdns-60.org.      3600    IN  NS  g-ns-766.awsdns-60.org.
awsdns-60.org.      3600    IN  NS  g-ns-1087.awsdns-60.org.
awsdns-60.org.      3600    IN  NS  g-ns-188.awsdns-60.org.
awsdns-60.org.      3600    IN  NS  g-ns-1660.awsdns-60.org.

;; ADDITIONAL SECTION:
g-ns-188.awsdns-60.org. 3600    IN  A   205.251.192.188
g-ns-766.awsdns-60.org. 3600    IN  A   205.251.194.254
g-ns-1087.awsdns-60.org. 3600   IN  A   205.251.196.63
g-ns-1660.awsdns-60.org. 3600   IN  A   205.251.198.124
g-ns-188.awsdns-60.org. 3600    IN  AAAA    2600:9000:5300:bc00::1
g-ns-766.awsdns-60.org. 3600    IN  AAAA    2600:9000:5302:fe00::1
g-ns-1087.awsdns-60.org. 3600   IN  AAAA    2600:9000:5304:3f00::1
g-ns-1660.awsdns-60.org. 3600   IN  AAAA    2600:9000:5306:7c00::1
```

Note that AAAA records are similar to A records, but are used with a newer version of the Internet Protocol, IPv6. You can ignore those entries because we are only working with IPv4.

Now answer the following questions:

- What was the IP address of the server where the query was initially sent (and therefore is the sender of this response.)
- What question was asked to the server? Include the hostname and query type.
- What should the next step be in this scenario?

Check in with your instructor if you are unsure of your answer.

5.2.1.4. Scenario 4

```
; <<>> DiG 9.10.6 <<>> @199.19.57.1 consumerreports.org A
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 5634
;; flags: qr rd; QUERY: 1, ANSWER: 0, AUTHORITY: 4, ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
```

```
; EDNS: version: 0, flags:; udp: 1232
;; QUESTION SECTION:
;consumerreports.org.       IN  A

;; AUTHORITY SECTION:
consumerreports.org.    3600    IN  NS  ns4.p201.dns.oraclecloud.net.
consumerreports.org.    3600    IN  NS  ns3.p201.dns.oraclecloud.net.
consumerreports.org.    3600    IN  NS  ns1.p201.dns.oraclecloud.net.
consumerreports.org.    3600    IN  NS  ns2.p201.dns.oraclecloud.net.
```

What are the next **two** steps here?

5.2.1.5. Scenario 5

```
; <<>> DiG 9.10.6 <<>> @216.239.34.10 gmail.com MX
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 55907
;; flags: qr aa rd; QUERY: 1, ANSWER: 5, AUTHORITY: 0, ADDITIONAL: 11
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;gmail.com.         IN  MX

;; ANSWER SECTION:
gmail.com.      3600    IN  MX  5 gmail-smtp-in.l.google.com.
gmail.com.      3600    IN  MX  30 alt3.gmail-smtp-in.l.google.com.
gmail.com.      3600    IN  MX  20 alt2.gmail-smtp-in.l.google.com.
gmail.com.      3600    IN  MX  10 alt1.gmail-smtp-in.l.google.com.
gmail.com.      3600    IN  MX  40 alt4.gmail-smtp-in.l.google.com.

;; ADDITIONAL SECTION:
gmail-smtp-in.l.google.com. 300 IN  A   142.251.2.26
gmail-smtp-in.l.google.com. 300 IN  AAAA    2607:f8b0:4023:c0d::1a
alt3.gmail-smtp-in.l.google.com. 300 IN A   172.253.113.27
alt3.gmail-smtp-in.l.google.com. 300 IN AAAA    2607:f8b0:4023:1::1a
alt2.gmail-smtp-in.l.google.com. 300 IN A   142.250.152.26
alt2.gmail-smtp-in.l.google.com. 300 IN AAAA    2607:f8b0:4001:c56::1b
alt1.gmail-smtp-in.l.google.com. 300 IN A   108.177.104.27
alt1.gmail-smtp-in.l.google.com. 300 IN AAAA    2607:f8b0:4003:c04::1b
alt4.gmail-smtp-in.l.google.com. 300 IN A   173.194.77.27
alt4.gmail-smtp-in.l.google.com. 300 IN AAAA    2607:f8b0:4023:401::1a
```

What is the next action here?

5.2.1.6. Scenario 6

```
; <<>> DiG 9.10.6 <<>> @205.251.193.54 www.campuswire.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 6011
;; flags: qr aa rd; QUERY: 1, ANSWER: 2, AUTHORITY: 4, ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;www.campuswire.com.        IN  A

;; ANSWER SECTION:
www.campuswire.com. 300 IN  CNAME   campuswire.com.
campuswire.com.     300 IN  A   35.241.17.106

;; AUTHORITY SECTION:
campuswire.com.     172800  IN  NS  ns-1314.awsdns-36.org.
campuswire.com.     172800  IN  NS  ns-1675.awsdns-17.co.uk.
campuswire.com.     172800  IN  NS  ns-310.awsdns-38.com.
campuswire.com.     172800  IN  NS  ns-638.awsdns-15.net.
```

What's the next action here?

5.2.1.7. Scenario 7

```
; <<>> DiG 9.10.6 <<>> @208.80.153.231 en.wikipedia.org
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 64801
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1232
;; QUESTION SECTION:
;en.wikipedia.org.      IN  A

;; ANSWER SECTION:
```

```
en.wikipedia.org.   86400   IN  CNAME   dyna.wikimedia.org.
```

What's the next action here?

5.2.1.8. Scenario 8

```
; <<>> DiG 9.10.6 <<>> @156.154.64.10 amazon.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 11860
;; flags: qr aa rd; QUERY: 1, ANSWER: 3, AUTHORITY: 8, ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;amazon.com.            IN  A

;; ANSWER SECTION:
amazon.com.     900 IN  A   52.94.236.248
amazon.com.     900 IN  A   54.239.28.85
amazon.com.     900 IN  A   205.251.242.103

;; AUTHORITY SECTION:
amazon.com.     7200    IN  NS  ns1.amzndns.co.uk.
amazon.com.     7200    IN  NS  ns1.amzndns.com.
amazon.com.     7200    IN  NS  ns1.amzndns.net.
amazon.com.     7200    IN  NS  ns1.amzndns.org.
amazon.com.     7200    IN  NS  ns2.amzndns.co.uk.
amazon.com.     7200    IN  NS  ns2.amzndns.com.
amazon.com.     7200    IN  NS  ns2.amzndns.net.
amazon.com.     7200    IN  NS  ns2.amzndns.org.
```

What's the next action here?

5.2.1.9. Scenario 9

```
; <<>> DiG 9.10.6 <<>> @205.251.197.229 www.sandiego.edu MX
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 30143
;; flags: qr aa rd; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 1
;; WARNING: recursion requested but not available
```

```
;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;www.sandiego.edu.      IN  MX

;; AUTHORITY SECTION:
sandiego.edu.              900   IN    SOA   ns-1664.awsdns-16.co.uk.   awsdns-
hostmaster.amazon.com. 1 7200 900 1209600 86400
```

What is the next action here?

5.2.2. Creating a Flowchart

Take some time now to reflect on the scenarios you have just seen. These represent the only types of scenarios you can reasonably expect to see when trying to parse a DNS response message.

For the next step, I would recommend creating a flowchart (or decision diagram) to map out the sequence of questions you would ask to determine which of these scenarios a given DNS response (stored in a `DNSResponse` message) maps to. Note that some scenarios may have the same sequence of questions that leads to them; that's a good thing as it will simplify your logic! You can then show this flowchart to your instructor for feedback before delving into implementation.

5.2.3. Implementing the `locate_answer` Function

Once you have the instructor's feedback on your flow chart, you can go ahead and start implementing the `locate_answer` function. Once you identify the scenario (through a series of conditional statements), you can then perform the next actions you determined for each of those scenarios. Note that the `get_answer` method in the `DNSResponse` class will be useful to you when doing these conditional checks.

How exactly do you perform the next actions?

1. For returning the answer directly, you should be able to get the data directly from the `DNSResponse` object you are given.

2. For giving up… well, just return `None`!

3. To perform a **new** query to the IP address of a specific hostname, you can utilize the `resolve` function. This happens to be the function called from the `main` function to do the query that the user requests.

   Take a moment to look at the `resolve` function in the starter code now to note exactly what it does, including its arguments and return value. It's currently not implemented; that's a task for the not too distant future.

   All new queries will initially be sent to the root servers, a list of which you can get using the `get_root_servers` function.

4. To send the same query to a different server, you'll once again use the `resolve` function. Since this isn't a new query, you don't want to start with the root servers, as

that would wipe away your current progress. The authority section will be where you find out about the servers to talk to next. Remember that all listed authorities will give you the same response, so once you have a response from one, you don't have to bother with the others.

Note that one of the project requirements will be that no function is longer than 30 lines of code. Before jumping into implementation, think about how you can decompose the `locate_answer` function into smaller steps, and implement those smaller steps in separate functions. See the next section on how to test (and for additional guidance on the iterative development process).

5.2.3.1. Testing the `locate_answer` Function

While you are iteratively developing your `locate_answer` function, you can use the unit tests found in `test_locate_answer.py` to help see which of the scenarios you are correctly handling.

A good design process will be to pick one of the tested scenarios (ideally the easiest one), update the implementation of `locate_answers` until it passes, then move on to another test scenario and repeat.

5.3. Phase 3: Talking to DNS Servers and Resolving User Queries

At this point, we have worked on parsing and interpreting DNS response messages. In this final phase, we'll work on getting data from real DNS servers and putting together all of our prior work into a final, working application.

5.3.1. Implementing the `query_servers` Function

Next up, we are going to work on sending queries to and getting responses from actual DNS servers. To do that, you will need to implement the `query_servers`. Locate this function in the starter code and review the docstring comment to find out what it does, what arguments it has (and their types), and its return value.

The example below shows you how to send a message and receive a response using a Python `socket` object, setting a timeout of 2 seconds in case of no response from a server with IP address "86.75.30.9" (🎶"I got it!").

```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.settimeout(2)   # socket should timeout after 2 seconds

message_to_send = b'hello' # the b before the string indicates this is a bytes
        object

sock.sendto(message, ("86.75.30.9", 53)) # destination: IP address 86.75.30.9,
        port 53
try:
   response = sock.recv(4096)

except socket.timeout:
   # there was a timeout!
```

```
        print("timed out")  # don't use print in your program, using
            logging.whatever

else:
    # no timeout, yay!
    print("Response:", response)
```

The actual code you write for this function should end up looking very similar to this.

One thing to remember is that all of the servers you query are expected to give you exactly the same response: if one doesn't have the answer you want, you therefore don't have to worry about checking any of the others.

Once you are done with the implementation, you can test it using the unit tests in `test_query_servers.py`. All of the unit tests in this file should pass before you move on to the next step.

5.3.2. Implementing the `resolve` Function

We've reached the final step in our implementation plan: implementing the `resolve` function. Luckily for you, it's one of the easiest functions to implement, as it will be a composition of the functions we've already written.

The comments inside of the function guide you on the implementation, but here are the basic steps.

1. Use the `construct_query` (given to you) to create a query. This function is located in `helpers.py`; review it's docstring to see how to use it.

   You'll want to use a random query ID to avoid cache poisoning. Use the `random.randrange` function to generate that random number.

2. Use the `query_servers` to send the given DNS servers.

3. Parse the response using your `parse_response` function.

4. Use the `locate_answer` function to look through the response and get the answer. One fun thing to note is that because your `locate_answer` function (or a helper function you wrote) calls the `resolve` function, the `resolve` function ends up being a recursive function. Recursion FTW! (The base case is the cases when we can answer the query based only on the response we got from the server.)

   If you couldn't get a response from a server, you should return `None`.

That's it!

Complete the `resolve` function implementation now.

5.3.3. Testing the `resolve` Function

To test it, do the following.

1. Run the unit tests in `test_resolve.py`, which does some checks to make sure you are calling the right functions with the right parameters.

2. Run the program from the command line using various hostnames and testing out the `--mx` and `--verbose` flags. Below is one example to get you started.

```
python3 resolver.py www.sandiego.edu --verbose
```

3. Run the `run-tests.sh` script using the following command. It will invoke the `resolver.py` program using roughly 25 different times, using a variety of hostnames and a mix of A and MX requests.

```
bash run-tests.sh
```

   Running this script shouldn't give you any errors except for the cases where you gave a request that couldn't be resolved. It should also not print out **any** thing other than the final output of the program, which will be one of the following three types of messages.

   - IP address for drsat.sucks: 1.2.3.4
   - Mail Server for google.com: smtp.google.com
   - ERROR: Could not resolve request.

   If you see any other program output, that means you used the `print` function rather than the `logging` library so change that now.

   If any of the tests failed, you can open up the `run-tests.sh` file, find the exact command to run to test it individually. You will likely want to add the `--verbose` option to the command so you can see debugging output directly on your screen (rather than only in the output.log file).

6. Submission

For this project, there is **nothing** to submit via Canvas. To submit your files, you will need to push your final code to your remote Git repository.

Before submitting, fill in the `project03.txt` file then add and commit your changes to your repository. This file will contain things like your names, number of hours worked on the project, and the amount of project bucks being used: see the initial contents of the file for a full description of what needs to be put there. ***Failure to complete the `project03.txt` file will result in failing grade for the assignment.***

Before doing your "final push", make sure you don't have any uncommitted changes. Run the following command:

```
git status
```

Then look for any sections labeled "Changed but not updated:" or "Changes to be committed:". You will also want to make sure that you don't have any "Untracked files" that you intend to submit. (However, do **NOT** add/commit any executable files, e.g. `resolve`.) If none of these are present, you can push your changes as follows:

```
git push
```

6.1. Checking Your Submission

I HIGHLY recommend double checking that your code successfully pushed to the server by doing the following set of commands to clone out a new copy of your repository, compile the code, and run it.

```
# create and change to a tmp directory
mkdir ~/comp375/tmp  # might fail if you already created this directory
cd ~/comp375/tmp

# create a fresh clone of your repository
# replace the "X" below with your group number
git clone git@code.sandiego.edu:comp375-sp24-p03-groupX p03-test

cd p03-test

# Use the make file to compile if you are using a compiled language like C.
# Skip this step if you are using an interpretted language like python.
make

# now do a few queries to make sure the response is as you expect.
```

This set of commands is exactly the set I will run to test your program: if they don't work for you, then they definitely aren't going to work for me.

7. Grading

This project will be graded as pass/fail. To receive a passing grade, your submission must do the following.

1. Correctly resolve A records for valid host names (e.g. www.google.com or www.sandiego.edu). This should work for all top-level domains (TLDs), including but not limited to .com, .net, .edu, .gov, .ca, and .io.

2. Having a correctly working function named `resolve` with the exact interface described earlier.

3. Correctly resolve MX records for a valid hostname (e.g. google.com or sandiego.edu). Again, all TLDs are expected to work correctly.

4. Uses the `logging` module instead of the `print` function to handle printing. The program should only print the final result unless using the `--verbose` option, which will cause it to print all log messages.

5. Correctly detect invalid host names and printing an informative, user-friendly error message. Invalid hostnames include:

    - Those with an invalid TLD (e.g. www.drsat.invalidtld)
    - Those with a valid TLD but an invalid domain (e.g. www.lllj2r12hafha.com).
    - Those with a valid TLD and domain but invalid subdomain (e.g. badbadbad.sandiego.edu).

    Both A and MX type requests should correctly handle these invalid requests.

6. Have well-commented source code. This includes a "header comment" for every function and judicious commenting throughout the body of your functions.

   You should have docstring comments at the beginning of every function, using the exact format given in the starter code functions.

7. All functions should have accurate type hints for all parameters and return values. Running the `mypy` module should result in no errors or warnings being reported.

   ```
   # run the mypy module to verify the correctness of type hints
   python -m mypy resolver.py
   ```

   If you get an error about the mypy module not existing, you can run the following command to install it.

   ```
   python3 -m pip install mypy
   ```

8. Must have reasonable and consistent "code style." This includes having variable names that are meaningful, having consistent indentation, and **not** having large chunks of unused code that have been commented out. There should also be **no** excessively long functions (i.e. > 30 lines of code).

9. Be submitted on time, either the official deadline or an extended deadline that comes from using a "COMP375 buck."

10. Have a fully completed Project Report in the `project03.txt` file.

11. Must demonstrate consistent and reasonable use of Git during development. This means that there are a significant number of commits, each of which have a meaningful message describing what changes they involved. Your Git repository should also be free of extraneous files.

Failure to meet **any** of these requirements will result in a failing grade for this assignment.

If any of the requirements are unclear, it is your responsibility to seek clarification from the instructor **BEFORE** the submission deadline. Ignorance is not an excuse for failing to meet a requirement.