How to Interpret Autolab Output

1 The General Format

Autolab output generally is formatted into 4 distinct sections, mirroring the 4 steps the autograder itself must perform to grade your work:

1. Extraction
2. Compilation
3. Testing
4. Scores

1.1 Extraction

During this section, Autolab reports any errors encountered while uncompressing the `tar.gz` file it received (see Compressing and Uncompressing Files). Generally speaking this step of the autograding causes no errors, and can be ignored. However, in the rare case in which it is unable to successfully extract the files out of your handin, the autograder will be unable to proceed, and will provide you with a score of 0. The most common reason for this to happen is if you accidentally submit a file that already exists in the grading infrastructure.

For reference, if the extraction step succeeds, you should see this output from Autolab:

```
Thu Jan 23 09:16:58 EST 2020
[15-122] Extraction...
============================================
file1.c0
file2.c0
file3.c0

Thu Jan 23 09:16:58 EST 2020
```

If you accidentally submit a file that already exists, you will receive output similar to the following:

```
Thu Jan 23 09:16:58 EST 2020
[15-122] Extraction...
============================================
file1.c0
file2.c0
file3.c0
tar: file4.c0: Cannot open: File exists
tar: Exiting with failure status due to previous errors
make[1]: *** [extract] Error 2
make: *** [default] Error 2

Score for this problem: 0.0
```

1.2 Compilation

The second step in autograding your code is compiling it. A good rule of thumb is to make sure your code compiles locally before submitting to autolab to avoid failures in the compilation step. This method is not foolproof, as it is possible for code to fail to compile on Autolab even if it compiles locally, so it is a good habit to check the compilation step for errors before looking for testing errors. By far the most common reason for this is failing to respect the interface of a data structure (see the guide on Libraries).

Note: Unlike the extraction step, the compilation step will not provide any output if compilation succeeds - it will simply move on to the third step (testing). Thus, a successful compilation phase looks like:

```
Fri Jan 24 18:35:51 EST 2020
[15-122] Compilation...
============================================
Fri Jan 24 18:35:51 EST 2020
[15-122] Grading...
============================================
```

1.3 Testing

The testing section of the output (by far the largest in terms of the space it occupies), lists all of the tests that were run, as well as the status of each test. The tests are separated by the tasks that they are a part of, and within each task, by the nature of the test (for instance, tests that check the strength of your contracts will generally be grouped together). Each test output follows one of the two formats below. Note that the all-caps words are the words which differ between tests. See Section Possible Test Expectations and Possible Test Outcomes for the various possible expectations and outcomes

For successful tests, the output will look something like:

```
Test TESTNAME, expect EXPECTATION - good, EXPECTATION MET.
```

For failed tests, the output will look something like the:

```
Test TESTNAME, expect EXPECTATION
*** TEST FAILED! Expected EXPECTATION
*** Actual outcome: OUTCOME
*** Hint: SOME HINT, WHICH MAY
*** BE MULTIPLE LINES
```

Note: If there are test failures, the autograder may not run the full set of tests for a particular section, so it may be the case that fixing one bug / failing test reveals 5 other (previously hidden) failing tests on a subsequent submission.

1.4 Scores

At the bottom of the output, there will be a section, summarizing the score for each task within the assignment. This might look something like

```
================================================
Task 1: 2/2
Task 2: 2/2
Task 3: 2/2
Task 4: 3/3
Task 5: 0/4
Task 6: 0/12

*** FINISHED!
```

2 Possible Test Expectations

In this section we will list all of the possible test expectations. For each test expectation, we will provide a description of what that expectation means.

- `Successful Execution`: This means that Autolab expects the test to proceed to completion without encountering any errors (including contract failures, incorrect values, etc).
- `Unsuccessful Execution`: This means that Autolab expects the test to call the **`error`** function, or return 1. The test should not fail a contract. This is only used in the Clac/Exp and the Lightsout programming assignments.
- `Assertion Failure`: This means that Autolab expects the test to either fail a contract or fail a call to **`assert`**.

C0VM Specific Expectations. There are some expected results that are only encountered in the C0VM programming assignment. These may not make sense until then.

- `c0_assertion_failure()`: This means that Autolab expects the test to call the `c0_assertion_failure` function. This test should not fail any ASSERT, REQUIRES or ENSURES.
- `c0_memory_error()`: This means that Autolab expects the test to call the `c0_memory_error` function. This test should not cause a segmentation fault.
- `c0_arith_error()`: This means that Autolab expects the test to call the `c0_arith_error` function. This test should not perform any illegal arithmetic operations (such as dividing by 0).
- `c0_user_error() on a` **`string`** `of length NNN`: This means that Autolab expects the test to call the `c0_user_error` function, and that the argument to that function should be a string of length `NNN`.

3 Possible Test Outcomes

Here we discuss the possible outcomes of a test, as well as the most common causes of each outcome. When we discuss the most common causes, we will make the implicit assumption that the given outcome was not one that you wanted.

File Did Not Compile. This result occurs whenever Autolab tries to run a test which failed to compile. Make sure to look at the compilation step of the output for the cause!

Your program ran to completion, but should have.... This error indicates that your program finished and returned successfully when it should have failed.

Possible Causes:

- Your pre/postconditions are not strong enough
- Your test cases do not test enough possibilities and incorrect code passes all of them

An assertion failed unexpectedly. This error indicates that an assertion **within your code, or called by your code** failed, when no assertion failures were expected. Note that this is different from your program producing the wrong answer.

Possible Causes:

- Your pre/postconditions are too strong
- Your code incorrectly call a library function
- Your test cases signal a failure on correct code. Perhaps you are expecting too much of uninitialized or otherwise unimportant memory?

Your Program, when run, produced the wrong answer. This indicates that your program ran to completion but produced a different answer from the one that our test cases expected. This indicates a bug in your code.

Autograder timed out after NNN seconds. The grader for each test has a time after which if the test has not had some outcome (be that an assertion failure, a successful execution or something else), the test will be considered failed.

Possible Causes:

- Your code might have an infinite loop.
- Your code might simply be too slow for this problem. If this is the case, consider the following suggestions:
  - Reduce the number of allocations (**`alloc_array`** and **`alloc`** in C0, `malloc` and `calloc` in C). Allocations are expensive operations in practice, so making many small allocations is significantly slower than making one large allocation.
  - Remove any unnecessary prints - printing (especially in a loop) is extremely expensive
  - Consider using loops instead of recursion to accomplish a task. Recursion tends to be much more expensive in practice.
  - Try to reduce the number of operations in a loop - it is better to compute a sum once rather than computing it 100 times.
  - Generally attempt to minimize redundancy in operations
  - If using a hash table, consider the size of the hash table and what hash function you are using. Having a too-small hash table or a bad hashing function can lead to noticeable decreases in hash table performance
- Your code may be running out of memory. In certain situations Autolab may misreport this as a timeout. See Your Program Ran Out of Memory.

A segfault occurred: This means that a pointer or array was accessed unsafely. This result occurs whenever the program fails with a segmentation fault. In general, this is a result of dereferencing a NULL pointer, or accessing an array out of bounds, but there are a couple of other possible causes.

Possible Causes:

- As the error message suggests, a bad pointer or array access is the most common cause
- It is possible for this error to occur if your program relies on recursion but goes too deep. In that case the program crashes. Make sure your base cases are correct and consider using while loops instead of recursion
- **C ONLY**. It is also possible (however unlikely) for this to occur if you attempt to dereference a pointer after freeing it.

An Arithmetic Error Occurred. This error occurs when the program attempts to perform an arithmetic operation that is illegal.

Possible Causes:

- Division (and modulus) by 0
- Dividing (and modulating) `int_min` by `-1`
- Shifting by an amount greater than 31 or less than 0
- **C ONLY**. If you use floating point numbers, any errors related to using those numbers will show up as an arithmetic error.

Important: In the C0VM assignemnt, there are tests where the expected outcome is `c0_arith_error`. If you get a test which has that expected outcome, and the actual result is that an arithmetic error occurred, what happened is that Autolab expects your code to catch that the C0 bytecode is about to make an arithmetic error and call `c0_arith_error` rather than performing the operation and dividing by 0!

Your Program Ran Out of Memory. This error indicates that your program attempted to allocate more memory than Autolab could support.

Possible Causes:

- Multiple calls to **`alloc_array`** with very large length
- Recursive functions that unnecessarily allocate memory repeatedly. Consider reusing your memory!

Important: The following errors are C0VM specific. If you encounter one of these while doing an assignment that is not C0VM, please post on Ed with what happened so that we can help you figure it out!

`c0_assertion_failure` was called. This means that your code called the `c0_assertion_failure` function when we did not expect it to be called.

Possible Causes:

- Your code incorrectly computed the value of the input and called this function when it should not have been
- Your program counter was in the wrong spot, causing the program to interpret a `c0_assertion_failure` when there was none

`c0_memory_error` was called. This means that your code called the `c0_memory_error` function when we did not expect it to be called.

Possible Causes:

- Your code incorrectly computed the value to be dereferenced
- Your program counter was in the wrong spot, causing the program to interpret a dereference or array access when it should not have

`c0_arithmetic_error` was called. This means that your code called the `c0_arithmetic_error` function when we did not expect it to be called.

Possible Causes:

- Your code incorrectly computed the values to be divided or modulused
- Your program counter was in the wrong spot, causing the program to interpret a division or modulus when it should not have

Status Code NNN, possibly the result of `c0_user_error` on a string of length NNN - 32. This means that your code returned an unexpected status code. This is most likely because your code called `c0_user_error` when it should not have, most likely because your program counter was in the wrong spot, causing the program to interpret a call to `c0_user_error` when it should not have.

4 General Hints

- The name of a test might be in and of itself a hint!
- Every test might test multiple things related to its name. For instance, a test called "null inputs" might test not just inputs that are themselves NULL, but that contain NULL pointers.
- Make sure you understand the difference between what was expected and what happened! A test where we expected an assertion failure, but the program ran to execution is different than a test where we expect successful execution but the program gave an assertion failure!
- When you have an Autolab test failure, if it is not immediately apparent what caused that error, try replicating the problem first on your computer (see the Testing Guide). Having a local failing test case can help you more easily trace through the problem, and provides you with a stronger guarantee that you have solved the problem before having to waste a submission.

- When testing your test files, it is often not necessary to pass all the test cases in order to get full points. Passing all of the test cases will also not result in extra credit, so don't stress out if you don't pass all of the test cases!
- Failures in one task may propagate to future tasks. This means that sometimes the hints provided for later tasks might not be useful. Instead, it may just be that failures in a previous task are causing new failures here.