

Debugging Strategies

HYPOTHESIS DEBUGGING

1. Develop a theory about the seen fault by defining a small **question** about the bug
 - a. Make sure the question is small enough that you can investigate it in ~20 minutes
 - b. Ignore all other questions
2. Form a **hypothesis** about the theory effects
3. Gather and **test** data against the **hypothesis**
4. If your program already has tests,
 - a. add a failing test
5. While the cause of the failure is found
 - a. Repeatedly refine and test the theory to find the answer to the question
6. If the going gets tough and you randomly poking the system to find the fault
 - a. Fall back on the systematic process or start over

BACKTRACKING DEBUGGING

1. Run a debugger
2. If the program freezes
 - a. Break the execution by adding breakpoint
 - b. Move up the call stack until you locate the loop that causes the freeze
 - c. Examine the loop's termination condition
 - d. Try to understand why the condition never gets satisfied
3. If there is a particular line of code that breaks the program (e.g., causes crashes, outputs the wrong value)
 - a. Break the execution by adding breakpoint on the line of the error or few lines before the error's line
 - b. Examine the values of variables at the point where the crash occurred
 - c. Look for null, corrupted, or uninitialized values
 - d. If you localize a variable with an incorrect value
 - i. Move up the stack of routine calls
 - ii. Look for incorrect arguments or other reasons associated with the crash
 1. Check the preconditions and postconditions
 2. navigate the calls between routines
 - iii. If you cannot find the problem
 1. Begin a series of program runs
 2. Set a breakpoint near the point where the incorrect value could have been computed
 3. Place breakpoint earlier and further up the call

4. Locate the problem

FORWARD DEBUGGING

1. If you cannot clearly identify the code line associated with the failure
 - The defect is related to *emergent properties* // e.g., performance, security, reliability
 - If your software takes too much memory or takes too long to respond
 - i. Use Profiling: tools and libraries that helps understand which routine clogs the CPU
 - If you find your web application's page defaced
 - i. Examine all the code for vulnerabilities //e.g., those that lead to buffer overflows, code injection, cross-site scripting attacks
 - ii. Use static program analysis
 - If your software fails to provide web service //reliability
 - i. Dig into each of its internal and external dependencies
 - ii. Verify if they work as supposed

PROBLEM SIMPLIFICATION

While the bug is not evident

1. Eliminate portions of the code that are not relevant to the code
2. If you found the buggy code
 - a. Simplify the code's body until you find the root of cause
 - b. If the bug seems to be related to a function call
 - i. Comment out a function call and replace it with a hardcoded value to check if the bug is in function call
 - ii. If the error message doesn't give you a line number,
 1. Until the problems goes away
 - a. Comment out huge chunk of code
 2. Rewrite the code and check if the new version is better
 - c. If the bug is related to data
 - i. Reduce the size or simplify the data

ERROR MESSAGE DEBUGGING

1. Read the error stack in the browser console or terminal
2. If there are many different error messages
 - a. Start fixing the first one; this fix usually will fix the rest
3. If the end of a long error message is not helpful
 - a. Check the beginning of the error message
4. If the error message is in cmd
 - a. Pipe it to less so you can scroll or search through it (./myprogram 2>&1 | less)

5. Locate the message's text in the source code

BINARY SEARCH DEBUGGING

1. Once the failure is associated with a specific component or class
 - a. Remove half the files/changes
 - b. Test
 - c. If the problem persist go to step **1.a**
 - d. Once the problem is fixed, restore the removed code.
 - e. Test.