Brute force

CS Osvita

January 16, 2025

Modern computers have a few gigahertz clock rates, meaning billions of operations per second. Since doing something interesting takes a few hundred instructions, one can hope to search millions of items per second on contemporary machines.

Once you have designed an algorithm to solve a particular problem in a programming contest or production, you must ask this question. Given the maximum input bound, can the currently developed algorithm, with its time/space complexity, pass the time/memory limit given? A **good strategy** is to brainstorm for many possible algorithms and then pick the most straightforward solution that works (i.e., is fast enough to pass the time and memory limit and yet still produce the correct answer).

Modern computers can process up to 10^9 operations in a second. You can use this information to determine if your algorithm will run in time. For example, if the maximum input size n is $100K$, and your current algorithm has a time complexity of $O(n^2)$, common sense (or your calculator) will inform you that $(100K)^2$ is a very large number that indicates that your algorithm will require (on the order of) tens of seconds to run. You will thus need to devise a faster (and also correct) algorithm to solve the problem. Suppose you find one that runs with a time complexity of $O(nlogn)$. Now, your calculator will inform you that $10^5 * log10^5$ is just $1.7 * 10^6$, and common sense dictates that the algorithm (which should now run in under a second) will be able to pass the time limit.

The problem bounds are as crucial as your algorithm's time complexity in determining if your solution is appropriate. Suppose you can only devise a

relatively simple-to-code algorithm with a horrendous time complexity of $O(n^4)$. This may appear to be an infeasible solution, but if $n \leq 50$, then you have solved the problem. You can implement your $O(n^4)$ algorithm with impunity since 50^4 is just $6.25M$, and your algorithm will run in less than a second.

However, note that the order of complexity does not necessarily indicate the actual number of operations that your algorithm will require. If each iteration involves a large number of operations (many floating point calculations or a significant number of constant sub-loops), or if your implementation has a high "constant" in its execution (unnecessarily repeated loops or multiple passes, or even I/O or execution overhead), your code may take longer to execute than expected.

Stress test

You can use stress testing to test your solution against many randomly generated tests to find a test case in your solution that produces a wrong answer for this case, and hopefully, this can help you find the bug in the code.

To run stress tests, you should have the code to test (the code giving the Wrong Answer). You should also write a brute-force solution. This solution is usually very slow, but you are 100% sure it provides the right solution. In addition, you need to write the test generator to generate tests.

```
def stress_test():
    test = 0
    while True:
        test += 1
        if test % 10 == 0:
            print(f"Test {test}")
        run_once()

def run_once():
    n = random.randint(1, 10)
    a = [0] * n
    for i in range(n):
        a[i] = random.randint(-10, 10)
    if brute_force(a) != optimal(a):
        raise Exception(f"bug found: {a}")
```