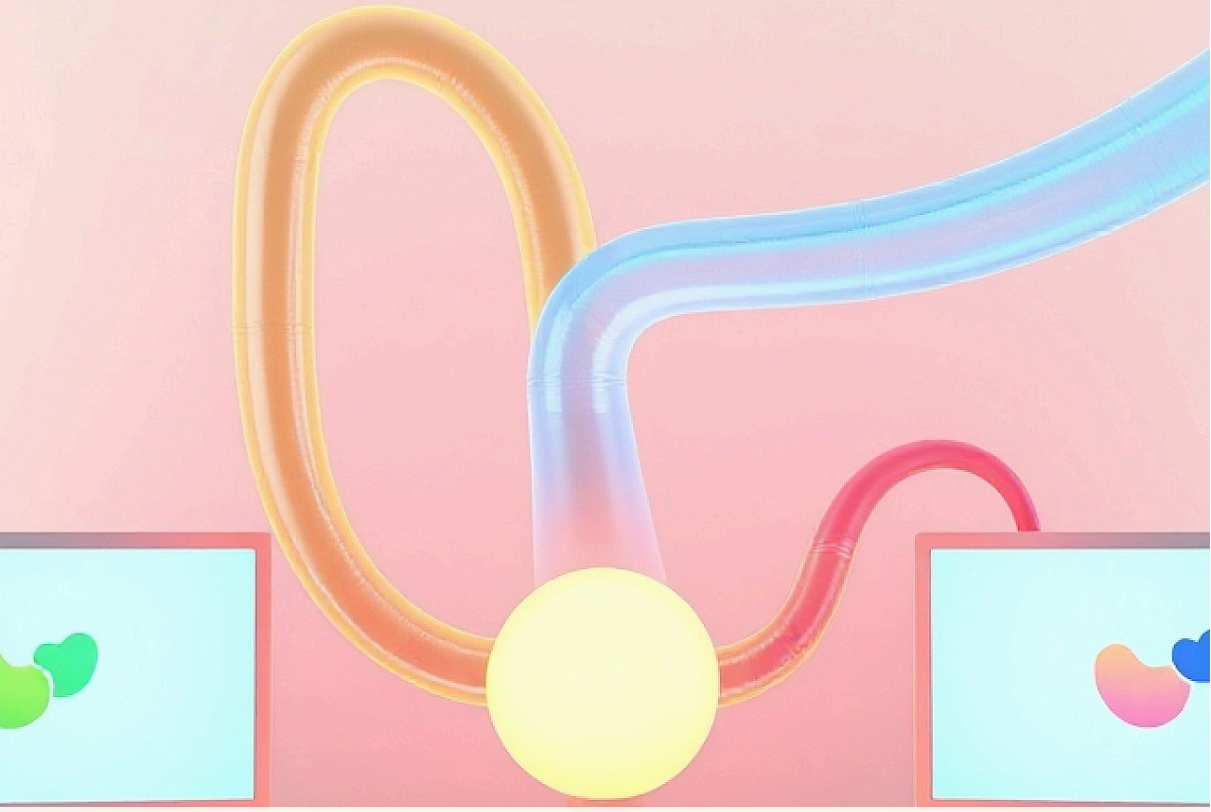
Why do two programs that do the same job run at different speeds?



www.yzokul.com.tr

Why do two programs that do the same job run at different speeds?



Two programs produce the same output, yet one finishes in a second and the other takes a minute. The short answer is that speed depends on four things: the algorithm, the data structure, the execution system, and the hardware. This book explains each one from zero, with small examples you can follow. You will see why the gap grows as data gets bigger.



A program is a list of steps that takes input and produces output. For example, input a list of names, output the longest name. Two programs can produce the same output through different lists of steps. One list might check every name twice; another checks each name once. The output is identical, but the number of steps is not.



Count how many basic operations a program performs: comparisons, additions, memory reads. More operations usually means more time. To compare two programs fairly, count their steps on the same input size. For example, finding a name in a list of ten names might take ten comparisons in one program and five in another. Counting steps is the first way to compare speed.



An algorithm is the recipe of steps a program follows to solve a problem. Two recipes for the same dish can differ in how much work they require. One recipe might stir the pot fifty times; another stirs ten times and gets the same result. In programs, a better algorithm reaches the answer with fewer operations, so it runs faster on the same machine.



One program might check the same items over and over without need. For example, to find the largest number, it compares every pair of numbers repeatedly. Another program scans the list once, keeping the largest seen so far. Both give the same answer, but the second skips repeated work. Fewer operations mean less time, especially when the list is long.



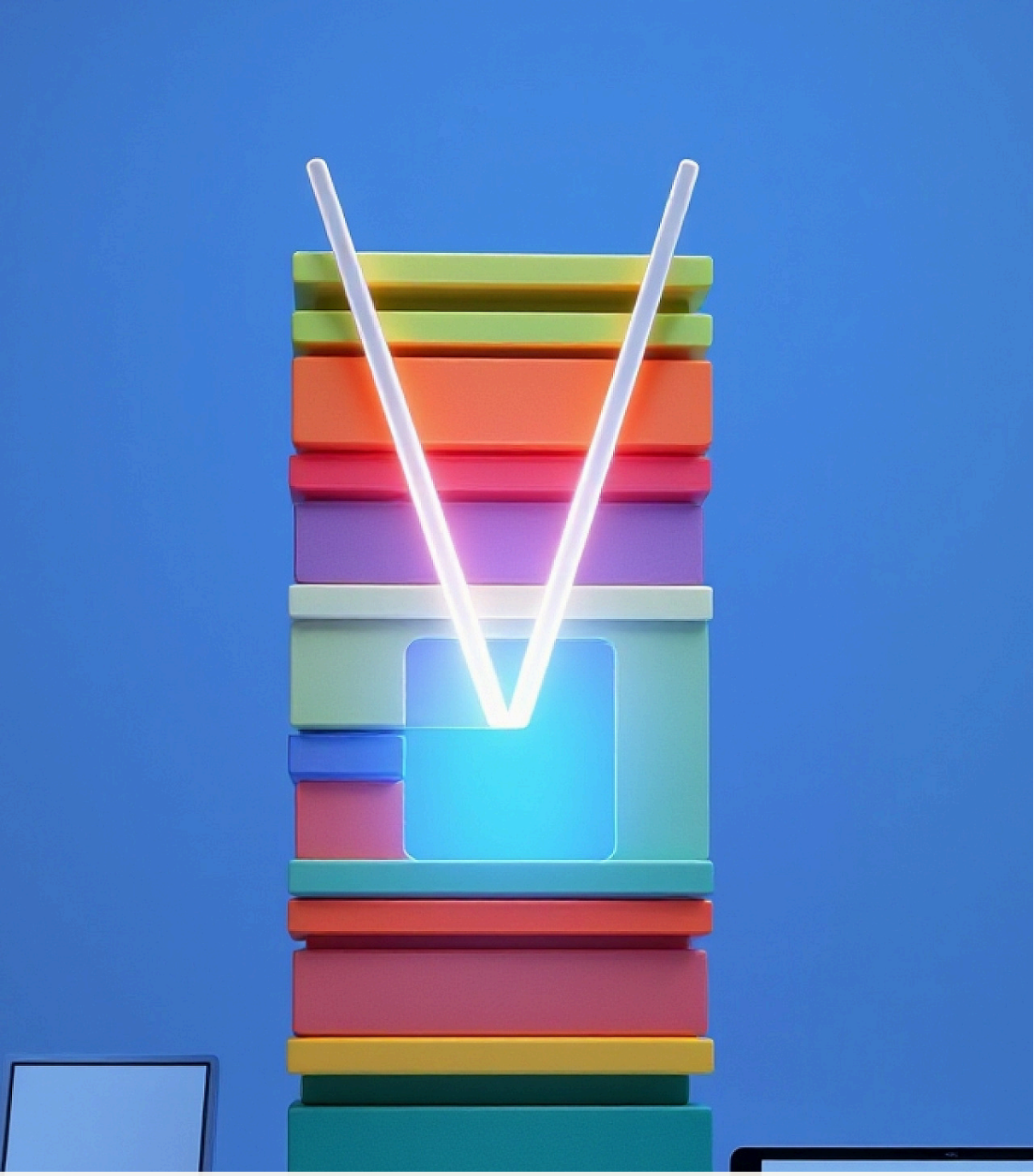
A data structure is how items are arranged in memory. The same algorithm can be fast or slow depending on that arrangement. For example, finding a name is slow if names are stored in a plain list you must scan. If names are stored in a sorted list or a hash table, the program can jump closer to the answer. Choosing the right structure saves many operations.



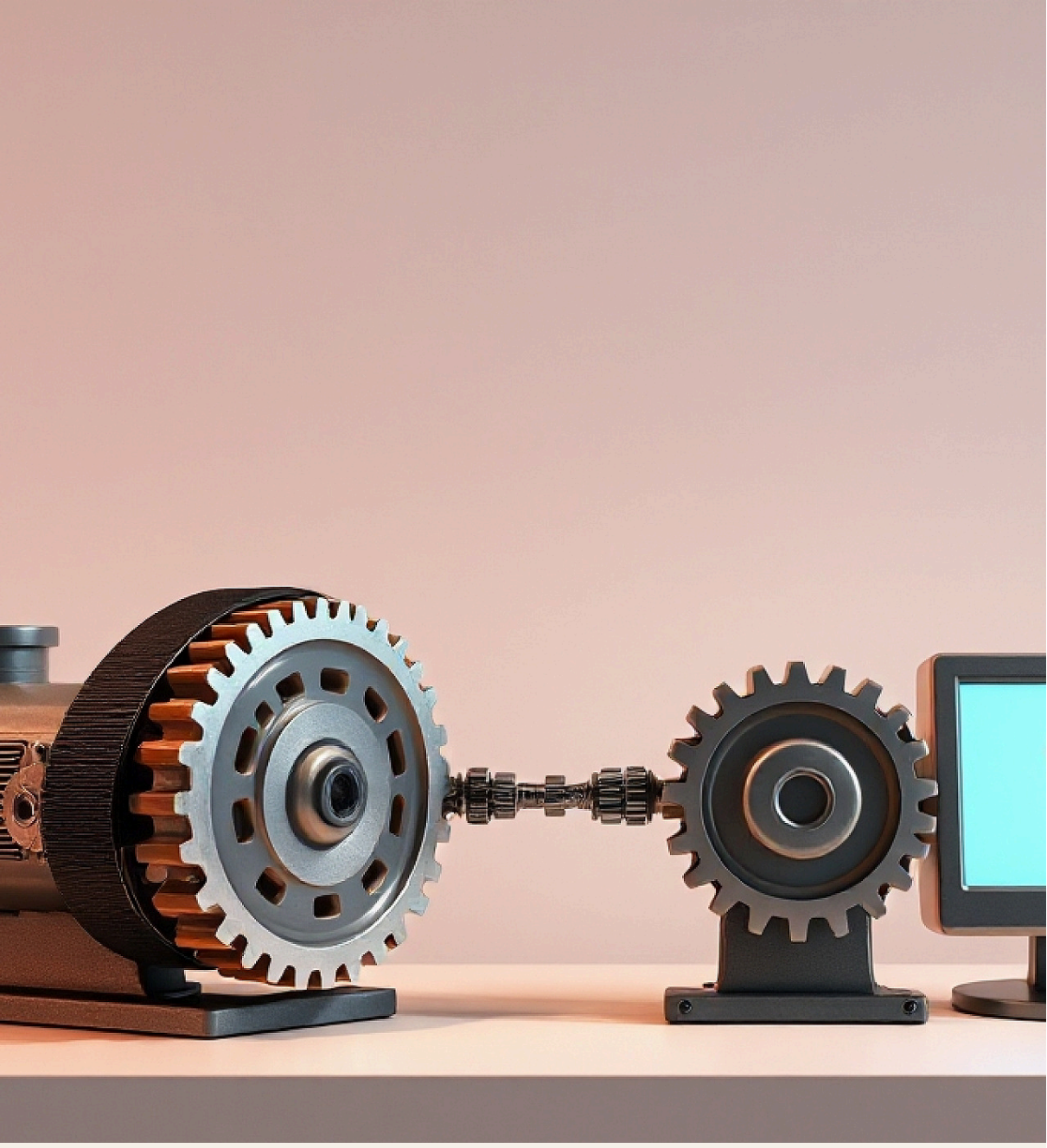
A simple search checks items one by one until it finds the target. For example, to find the number 7 in a list of ten numbers, you might check all ten. This works, but the number of checks grows with the size of the list. If the list has a thousand items, you might need a thousand checks. That is slow for large data.



When data is sorted, a suitable search method can cut the work a lot. Order lets the program skip whole sections instead of checking every item. For example, in a sorted list of numbers, if the middle number is too small, you know the target is not in the left half. You can ignore that half entirely. Sorted data enables faster methods.



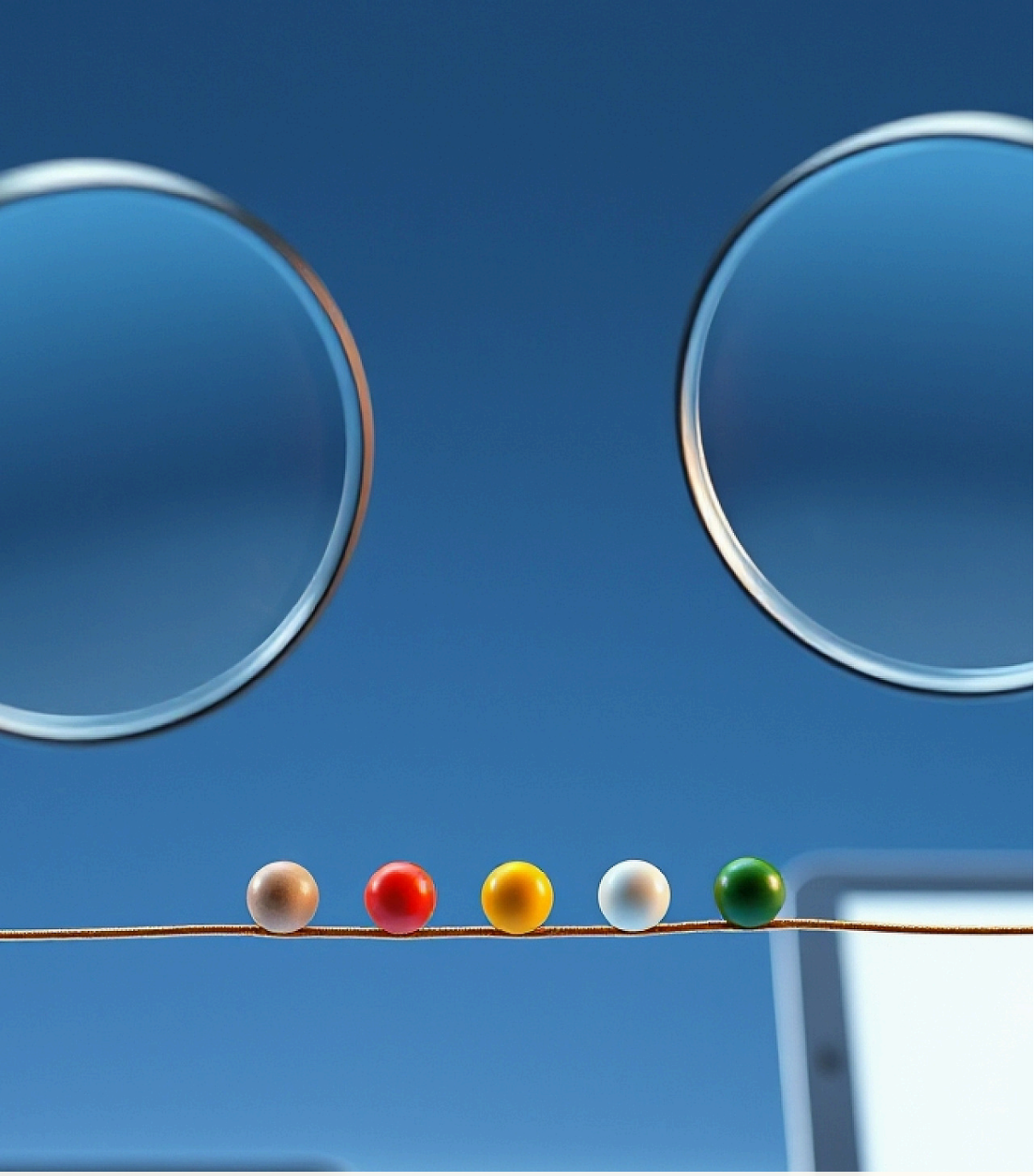
A search that looks at the middle item and throws away half the list each time needs far fewer steps. For example, in a sorted list of one thousand numbers, it takes about ten checks, not one thousand. Each check halves the remaining items: 1000, 500, 250, and so on. This is called binary search, and it shows how a good search method helps with ordered data.



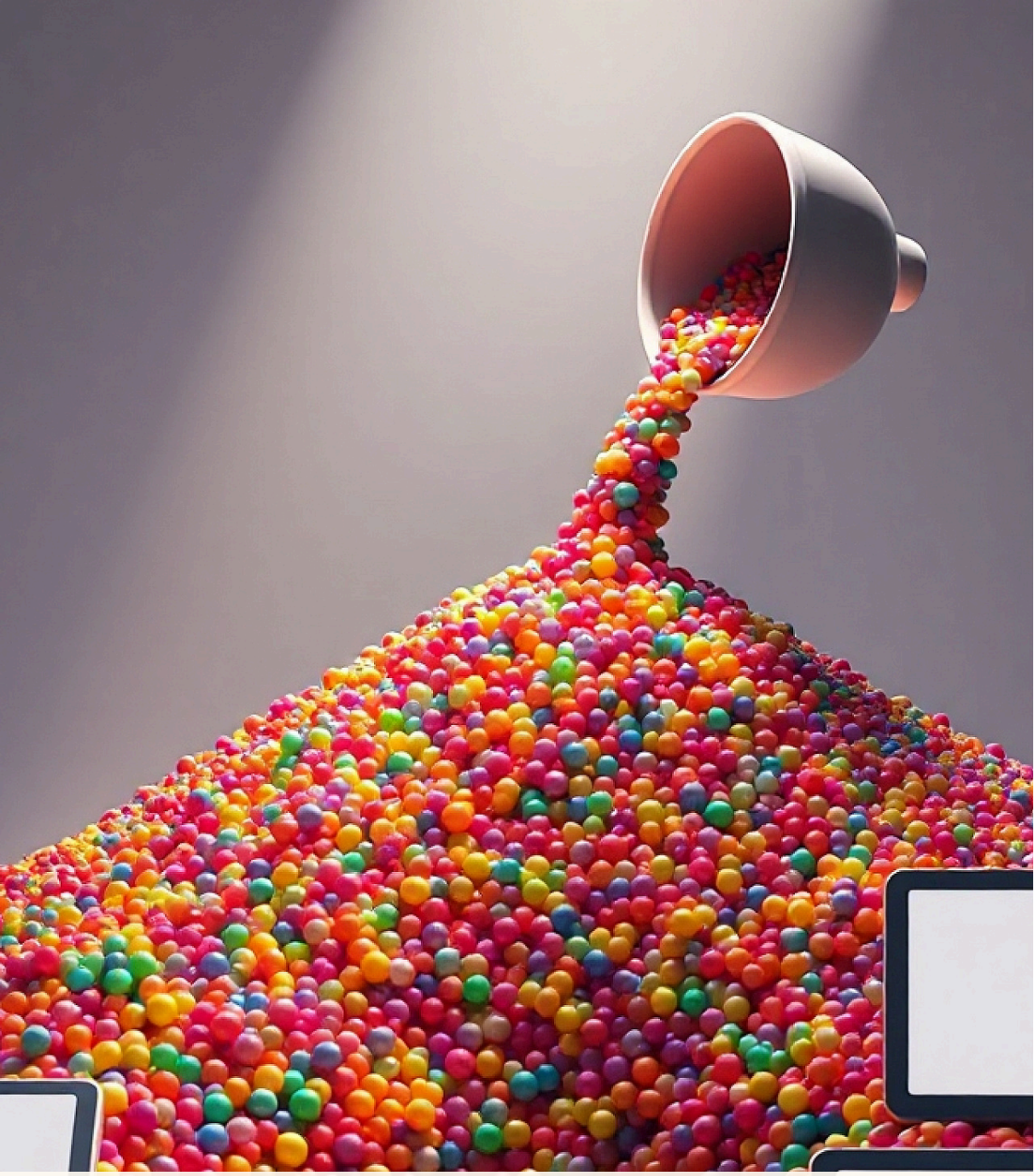
The execution system of a language affects speed. Some code is translated ahead of time into machine instructions, so it runs directly on the processor. Other code is interpreted line by line while it runs, which adds overhead. The same steps can therefore run faster or slower depending on how the language executes them. This is one reason two programs with similar logic can differ.



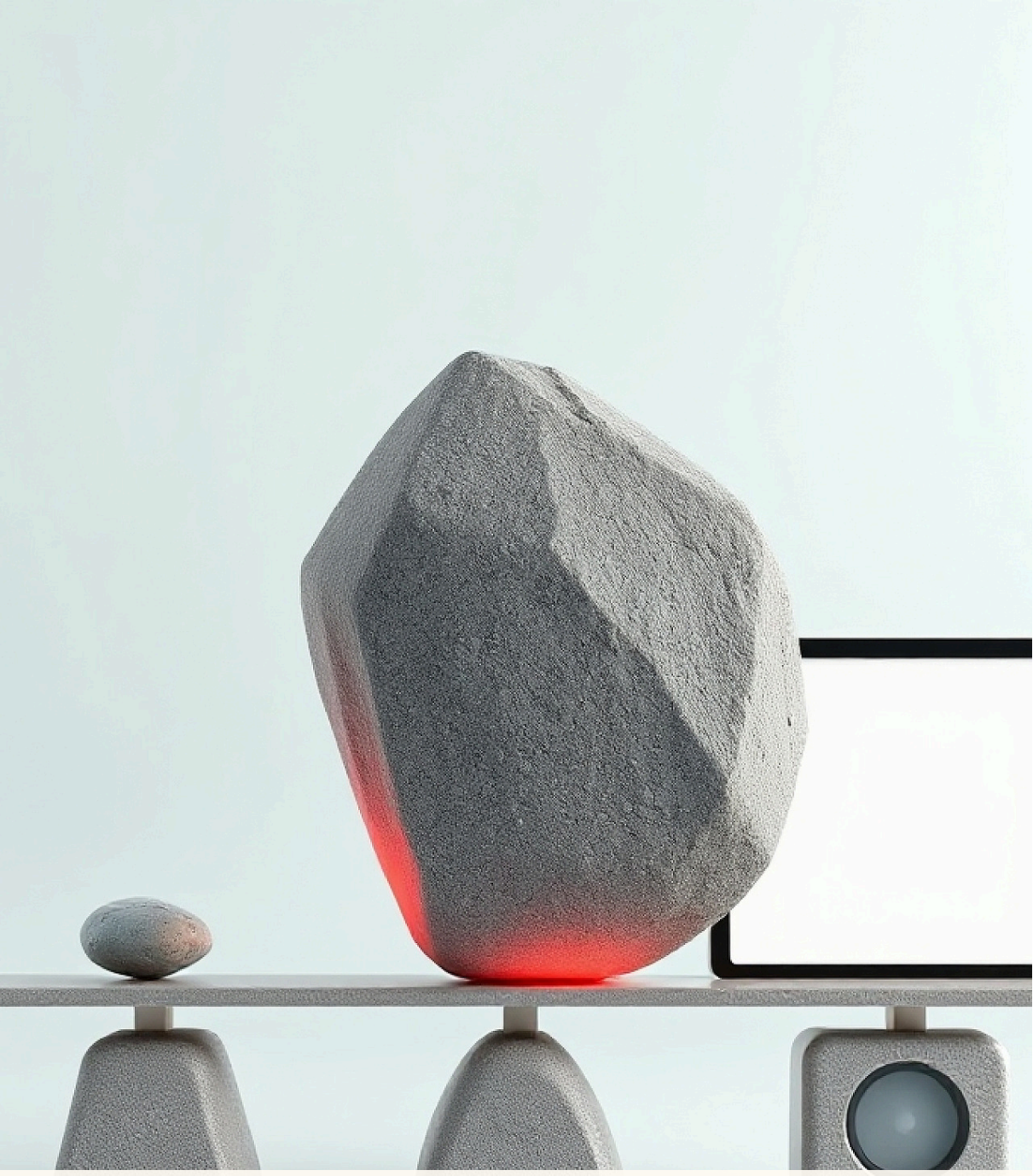
The same program runs at different speeds on different machines. Processor speed, memory size, and memory speed set limits on how fast any program can go. A fast processor can do more operations per second; more memory can hold more data without slowdowns. Hardware does not change the algorithm, but it changes the time each operation takes.



Two programs on a tiny list finish at almost the same time. With few items, the wasted operations are too few to notice. For example, on a list of five names, checking each name twice takes only ten checks, which is instant. The slower program looks just as fast. Small examples hide the difference because the total work is small.



Grow the data and watch the slower program fall far behind. For example, on a list of one million names, checking each name twice takes two million checks, while checking once takes one million. The gap becomes obvious. As the data grows, the cost of unnecessary operations becomes clear. This is why you must test with realistic sizes.



Do not judge speed from a single small test. A program that looks fine on tiny data can be very slow on large data. For example, a search that checks every item seems instant on ten items but takes forever on ten million. Always test with data sizes close to what you will actually use. One small test can mislead you completely.



Some methods need preparation, like sorting data first. That setup costs time, so you must count it when comparing programs. For example, binary search is fast, but only after the list is sorted, and sorting a large list takes many operations. If you search only once, the setup may outweigh the gain. Count setup plus search together.



Speed comes from the algorithm, the data structure, the execution system, and the hardware. Fewer unnecessary operations and suitable methods matter most as data grows. A good search method on ordered data can turn a million checks into about twenty. Small tests hide the gap; large data reveals it. Remember these four factors when comparing programs.