9.5 randomized algorithms

Suppose that you are a professor who is giving weekly programming assignments. You want to ensure that the students are doing their own programs or, at the very least, that they understand the code that they are submitting. One solution is to give a quiz on the day each program is due. However, these quizzes take time from class and doing so might be practical for only roughly half the programs. Your problem is to decide when to give the quizzes.

Of course, if you announce the quizzes in advance, that could be interpreted as an implicit license to cheat for the 50 percent of the programs that will not get a quiz. You could adopt the unannounced strategy of giving quizzes on alternate programs, but students would quickly figure out that strategy. Another possibility is to give quizzes on what seem like the important programs, but that would likely lead to similar quiz patterns from semester to semester. Student grapevines being what they are, this strategy would probably be worthless after one semester.

One method that seems to eliminate these problems is to flip a coin. You make a quiz for every program (making quizzes is not nearly as time consuming as grading them), and at the start of class, you flip a coin to decide whether the quiz is to be given. This way neither you nor your students can know before class whether a quiz will be given. Also, the patterns do not repeat from semester to semester. The students can expect a quiz to occur with 50 percent probability, regardless of previous quiz patterns. The disadvantage of this strategy is that you could end up giving no quizzes during an entire semester. Assuming a large number of programming assignments, however, this is not likely to happen unless the coin is suspect. Each semester the expected number of quizzes is half the number of programs, and with high probability, the number of quizzes will not deviate much from this.

This example illustrates the *randomized algorithm,* which uses random numbers, rather than deterministic decisions, to control branching. The running time of the algorithm depends not only on the particular input, but also on the random numbers that occur.

The worst-case running time of a randomized algorithm is almost always the same as the worst-case running time of the nonrandomized algorithm. The

important difference is that a good randomized algorithm has no bad inputs—only bad random numbers (relative to the particular input). This difference may seem only philosophical, but actually it is quite important, as we show in the following example.

The running time of a randomized algorithm depends on the random numbers that occur, as well as the particular input.

Let us say that your boss asks you to write a program to determine the median of a group of 1,000,000 numbers. You are to submit the program and then run it on an input that the boss will choose. If the correct answer is given within a few seconds of computing time (which would be expected for a linear algorithm), your boss will be very happy, and you will get a bonus. But if your program does not work or takes too much time, your boss will fire you for incompetence. Your boss already thinks that you are overpaid and is hoping to be able to take the second option. What should you do?

The quickselect algorithm described in Section 8.7 might seem like the way to go. Although the algorithm (see Figure 8.23) is very fast on average, recall that it has quadratic worst-case time if the pivot is continually poor. By using median-of-three partitioning, we have guaranteed that this worst case will not occur for common inputs, such as those that have been sorted or that contain a lot of duplicates. However, there is still a quadratic worst case, and as Exercise 8.9 showed, the boss will read your program, realize how you are choosing the pivot, and be able to construct the worst case. Consequently, you will be fired.

Randomized quickselect is statistically guaranteed to work in linear time.

By using random numbers, you can statistically guarantee the safety of your job. You begin the quickselect algorithm by randomly shuffling the input by using lines 10 and 11 in Figure 9.7.[1] As a result, your boss essentially loses control of specifying the input sequence. When you run the quickselect algorithm, it will now be working on random input, so you expect it to take linear time. Can it still take quadratic time? The answer is yes. For any original input, the shuffling may get you to the worst case for quickselect, and thus the result would be a quadratic-time sort. If you are unfortunate enough to have this happen, you lose your job. However, this event is statistically impossible. For a million items, the chance of using even twice as much time as the average would indicate is so small that you can essentially ignore it. The computer is much more likely to break. Your job is secure.

Instead of using a shuffling technique, you can achieve the same result by choosing the pivot randomly instead of deterministically. Take a random item in the array and swap it with the item in position `low`. Take another

1. You need to be sure that the random number generator is sufficiently random and that its output cannot be predicted by the boss.

random item and swap it with the item in position high. Take a third random item and swap it with the item in the middle position. Then continue as usual. As before, degenerate partitions are always possible, but they now happen as a result of bad random numbers, not bad inputs.

Let us look at the differences between randomized and nonrandomized algorithms. So far we have concentrated on nonrandomized algorithms. When calculating their average running times, we assume that all inputs are equally likely. This assumption does not hold, however, because nearly sorted input, for instance, occurs much more often than is statistically expected. This situation can cause problems for some algorithms, such as quicksort. But when we use a randomized algorithm, the particular input is no longer important. The random numbers are important, and we get an *expected* running time, in which we average over all possible random numbers for any particular input. Using quickselect with random pivots (or a shuffle preprocessing step) gives an $O(N)$ expected time algorithm. That is, *for any input*, including already sorted input, the running time is expected to be $O(N)$, based on the statistics of random numbers. On the one hand an expected time bound is somewhat stronger than an average-case time bound because the assumptions used to generate it are weaker (random numbers versus random input) but it is weaker than the corresponding worst-case time bound. On the other hand, in many instances solutions that have good worst-case bounds frequently have extra overhead built in to assure that the worst case does not occur. The $O(N)$ worst-case algorithm for selection, for example, is a marvelous theoretical result but is not practical.

Some randomized algorithms work in a fixed amount of time but randomly make mistakes (presumably with low probability). These mistakes are *false positives* or *false negatives*.

Randomized algorithms come in two basic forms. The first, as already shown, always gives a correct answer but it could take a long time, depending on the luck of the random numbers. The second type is what we examine in the remainder of this chapter. Some randomized algorithms work in a fixed amount of time but randomly make mistakes (presumably with low probability), called *false positives* or *false negatives*. This technique is commonly accepted in medicine. False positives and false negatives for most tests are actually fairly common, and some tests have surprisingly high error rates. Furthermore, for some tests the errors depend on the individual, not random numbers, so repeating the test is certain to produce another false result. In randomized algorithms we can rerun the test on the same input using different random numbers. If we run a randomized algorithm 10 times and get 10 positives—and if a single false positive is an unlikely occurrence (say, 1 chance in 100)—the probability of 10 consecutive false positives (1 chance in 100^{10} or one hundred billion billion) is essentially zero.