PART V

Miscellaneous topics

Chapter 26

Efficiency of programs

For this chapter, switch languages in DrRacket to "Intermediate Student Language".

26.1 Timing function calls

Computer programs, of course, *must* produce correct answers. But that's not enough: the main reason computers were invented was to produce correct answers *quickly*. If a program doesn't run fast enough, you can buy a more expensive, faster computer. But perhaps surprisingly, you can often get much more dramatic improvements by changing the *program*.

In Intermediate Student Language, there's a built-in function named `time` that allows you to measure how long something takes. For example, type `(time (* 3 4))` in the Interactions pane of DrRacket. You should see something like

```
cpu time:  0 real time:  0 gc time:  0
12
```

The 12, of course, is the result of `(* 3 4)`; the previous line shows how much time the computation took, by three different measures. "CPU time" is how much time (in milliseconds) the computer's processor spent actually solving the problem (as opposed to managing the operating system, managing memory, managing DrRacket itself, *etc.*). "Real time" is the total time (in milliseconds) from when you hit ENTER to when the answer came out. "GC time" is how much time (in milliseconds) DrRacket spent "garbage-collecting", *i.e.* releasing things from memory that are no longer needed.

In the case of `(* 3 4)`, all three are well under a millisecond, so the answers are all 0. To see nonzero times, you need to use some of the functions defined in Chapters 22, 23, 24, and 25.

Recall the `add-up-to` function of Exercise 24.1.7, and type the following lines into the Definitions pane (after the definition of `add-up-to`):

```
(time (add-up-to 10))
(time (add-up-to 100))
(time (add-up-to 1000))
(time (add-up-to 10000))
(time (add-up-to 100000))
(time (add-up-to 1000000))
```

Hit "Run" and see what happens. Each of the examples shows a line of timing figures. Not surprisingly, it takes longer to solve a larger problem than a smaller one.

Try running `(time (add-up-to 100000))` several times, writing down the CPU time, real time, and GC time each time. How much do they vary from one trial to the next? How well can you predict them? Can you predict how long it'll take to compute `(add-up-to 200000)`?

Exercise 26.1.1 *Choose some functions defined in Chapters 22, 23, 24, and 25, and try timing them on various sizes of arguments. How much does the time vary from one trial to the next? From timing a few arguments, can you predict how long it'll take on a new argument?*

Hint: *You can use* `randoms`*, from Exercise 24.1.11, to generate large lists of numbers. Make sure you don't count the time to generate the numbers in the time to run the function. One way to do this is to define a variable to hold a list of, say, 10000 random numbers, and* then *call* `(time (the-function-I'm-testing lotsa-numbers))`*.*

Some good ones to try are
`convert-reversed`*, exercise 22.5.10*
`all-match?`*, exercise 22.5.13*
`largest`*, exercise 22.5.17*
`count-blocks`*, exercise 22.5.18*
`stutter`*, exercise 23.1.6*
`backwards`*, exercise 23.1.10*
`unique`*, exercise 23.2.4*
`tally-votes`*, exercise 23.2.7*
`sort`*, exercise 23.6.1*
`subsets`*, exercise 23.6.3*
`scramble`*, exercise 23.6.4*
`factorial`*, exercise 24.1.8*
`fibonacci`*, exercise 24.1.9*
`wn-add`*, exercise 24.3.1*
`wn-mult`*, exercise 24.3.2*
`wn-raise`*, exercise 24.3.3*
`wn-prime?`*, exercise 24.3.8*
`binary-add`*, exercise 25.4.9*
`binary-mult`*, exercise 25.4.10*
`binary-raise`*, exercise 25.4.11*
`substring?`*, exercise 25.4.12*
`subsequence?`*, exercise 25.4.13*
`lcsubstring`*, exercise 25.4.14*
`lcsubsequence`*, exercise 25.4.15*

26.2 Review of important words and concepts

A computer is generally doing a lot of things at once, only some of which are running the program you just wrote. Some of a computer's time goes into managing the operating system (Windows, MacOS, *etc.*), some goes into managing DrRacket, some goes into reclaiming memory that is no longer needed ("garbage collection"), and some goes into solving your problem.

A computer typically takes longer to evaluate a function on a large or complicated argument than on a small or simple one. *How much* longer is an important area of research in computer science: for some problems, doubling the size of the argument roughly

doubles the time it takes (as you might expect), but other problems behave differently. Furthermore, different programs to solve the same problem can have dramatically different efficiencies.

26.3 Reference: New syntax for timing

This chapter introduced one new function (technically a "special form"), `time`, which evaluates an expression, prints out how long it took, and then returns the result.