errors in the individual pieces are unlikely to point in the same direction. Some pieces will be underestimates, some will be overestimates, and the product of all the pieces is likely to be close to the true value.

This numerical example is our first experience with the random walk. Their crucial feature is that the expected wanderings are significantly smaller than if one walks in a straight line without switching back and forth. How much smaller is a question that we will answer in Chapter 8 when we introduce special-cases reasoning.

2.6 The Unix philosophy

Organizing complexity by breaking it into manageable parts is not limited to numerical estimation; it is a general design principle. It pervades the Unix and its offspring operating systems such as GNU/Linux and FreeBSD. This section discusses a few examples.

2.6.1 Building blocks and pipelines

Here are a few of Unix's building-blocks programs:

- `head`: prints the first n lines from the input; for example, `head -15` prints the first 15 lines.
- `tail`: prints the last n lines from the input; for example, `tail -15` prints the last 15 lines.

How can you use these building blocks to print the 23rd line of a file? Divide and conquer! One solution is to break the problem into two parts: printing the first 23 lines and, from those lines, printing the last line. The first subproblem is solved with `head -23`. The second subproblem is solved with `tail -1`.

To combine solutions, Unix provides the pipe operator. Denoted by the vertical bar |, it connects the output of one program to the input of another command. In the numerical estimation problems, we combined the solutions to the subproblems by using multiplication. The pipe operator is analogous to multiplication. Both multiplication in numerical estimation, and pipes in programming, are examples of composition operators, which are essential to a divide-and-conquer solution.

To print the 23rd line, use this combination:

```
head -23 | tail -1
```

To tell the system where to get the input, there are alternatives:

1. Use the preceding combination as is. Then the input comes from the keyboard, and the combination will read 23 typed lines, print out the final line from those 23 lines, and then will exit.
2. Tell `head` to get its input from a file. An example file is the dictionary. On my GNU/Linux laptop it is the file `/usr/share/dict/words`, with one word per line. To print the 23rd line (i.e. the 23rd word):

```
head -23 /usr/share/dict/words | tail -1
```

3. Let `head` read from its idea of the keyboard, but connect the keyboard to a file. This method uses the < syntax:

   ```
   head -23 < /usr/share/dict/words | tail -1
   ```

   The < operator tells the shell (the Unix command interpreter) to connect `/usr/share/dict/words` to the input of `head`.

4. Like the preceding method, but use the `cat` program. The `cat` program copies its input file(s) to the output. So this extended pipeline has the same effect as the preceding alternative:

   ```
   cat /usr/share/dict/words | head -23 | tail -1
   ```

   It is slightly less efficient than letting the shell redirect the input itself, because the longer pipeline requires running one extra program (`cat`).

This example introduced the Unix philosophy: To enable divide-and-conquer reasoning, provide useful small utilities and ways to combine them. The next section applies this philosophy to a whimsical example from a scavenger hunt created by Donald Knuth: Find the next word in the dictionary after 'angry', where the dictionary is alphabetized starting with the last letter, then the second-to-last letter, etc.

2.6.2 Sorting and searching

So, how do you find the next word in the dictionary after 'angry', where the dictionary is alphabetized starting with the last letter, then the second-to-last letter, etc.?

Divide the problem into two parts:

1. Make a reverse dictionary, alphabetized starting with the last letter, then the second-to-last letter, etc.
2. Printing the line after 'angry'.

The first problem subdivides into:

1. Reverse each line of a dictionary.
2. Sort the reversed dictionary.
3. Unreverse each line.

Unix provides `sort` for the second subproblem. For the first and third problems, a search through the Unix toolbox, using `man -k`, says:

```
$ man -k reverse
build-rdeps (1)      - find packages that depend on a specific package to
bui...
col (1)              - filter reverse line feeds from input
git-rev-list (1)     - Lists commit objects in reverse chronological order
rev (1)              - reverse lines of a file or files
tac (1)              - concatenate and print files in reverse
xxd (1)              - make a hexdump or do the reverse.
```

Ah! `rev` is just the program for us. So the first subproblem is solved with this pipeline:

```
rev < /usr/share/dict/words | sort | rev
```

The second problem – finding the line after 'angry' – is a task for the pattern-finding program `grep`. In the simplest usage, you tell `grep` a pattern, and it prints every line from its input that matches the pattern.

The patterns are regular expressions. Their syntax can become arcane, but the most important features are simple. For example,

```
grep '^angry$' < /usr/share/dict/words
```

prints all lines that exactly match `angry`: The ^ character matches the beginning of the line, and the $ character matches the end of the line.

That invocation of `grep` is not useful except as a spell checker, since it tells us only that `angry` is in the dictionary. However, the `-A` option, you can tell `grep` how many lines to print after each matching line. So

```
grep -A 1 '^angry$' < /usr/share/dict/words
```

will print 'angry' and the word after it (in the regular dictionary):

```
angry
angst
```

To print just the word after 'angry', follow the `grep` command with `tail`:

```
grep -A 1 '^angry$' < /usr/share/dict/words | tail -1
```

Now combine these two solutions into solving the scavenger hunt problem:

```
rev </usr/share/dict/words | sort | rev | grep -A 1 '^angry$' | tail -1
```

This pipeline fails with the error

```
rev: stdin: Invalid or incomplete multibyte or wide character
```

The `rev` program is complaining that it doesn't understand some of the characters in the dictionary. `rev` is from the old, ASCII-only days of Unix, whereas the dictionary is modern and includes non-ASCII characters such as accented letters.

To solve this unexpected problem, clean the dictionary before passing it to `rev`. The cleaning program is again `grep`, which can allow through only those lines that are pure ASCII. This command

```
grep '^[a-z]*$' < /usr/share/dict/words
```

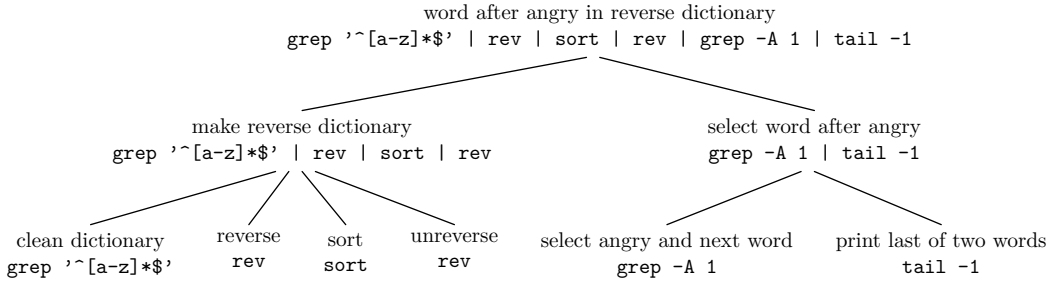
will print a dictionary made up only of unaccented, lowercase letters. In a regular expression, the * operator means 'match 0 or more occurrences of the preceding regular expression'.

The full pipeline is

```
grep '^[a-z]*$' < /usr/share/dict/words \
| rev | sort | rev \
| grep -A 1 '^angry$' | tail -1
```

where the backslashes at the end of the lines tell the shell to keep reading the command even though the line ended.

The tree representing this solution is



which produces 'hungry'.

2.6.3 Further reading

To learn more about the principles of Unix, especially how the design facilitates divide-and-conquer programming, see [1, 2, 3].