TRAVESTY: PUTTING IT ALL TOGETHER

We have covered considerable ground in our tour of data structures. By now you should have not only an appreciation of the abstractions and principles behind the choice of structures to represent information in a computer, but also have a battery of useful techniques to help you solve programming problems. In each chapter so far, we have introduced new data structures and then provided examples of how these structures can be applied in practice. Of course, the real world isn't as logically arranged as that. When faced with a programming problem, you probably won't have someone around to tell you what data structure would be best suited for your program. In this chapter, we will look at one interesting problem for which there is no immediate best choice for representation, and we will investigate some of the avenues of approach.

10.1 THE PROBLEM

The “monkeys and typewriters” story is probably familiar to most of you. Imagine a room full of monkeys, each sitting before a typewriter, diligently pecking away at the keys in a completely random manner. In principle, at least, if we were prepared to wait long enough, it is not beyond the realm of possibility that eventually we might find the complete works of Shakespeare among the mountains of garbage that had been produced. In fact, if we imagine this “thought experiment” continuing for an indefinite time, we would eventually expect to find any text whatsoever, such as a recipe for stir-fried
philosophers, Cleopatra's old love letters, and a verbatim transcript of the hiring interview of Rudolph the Red-Nosed Reindeer, in Esperanto. This compelling idea has been around since at least the 1690s; it was cast in its familiar form by Arthur Eddington in 1927 and has formed the basis of a number of works of fiction, comedy, and scholarship since then. Nowadays, we would replace the monkeys with computers: The cost per unit is about the same, computers are cheaper than monkeys to feed and care for, and computers (unlike monkeys) will work tirelessly and without complaint as long as there is sufficient electrical power available.

It is trivial to write a program that prints characters at random, and so we will write one here. We use a 27-symbol alphabet consisting of the blank and the letters A through Z. To make the programming task easier, we will represent the blank internally by '@', since '@' immediately precedes 'A' in ASCII order.

```cpp
void main()
{
    for (int i = 0; i < 1500; i++)
    {
        char c = char('0' + rand() % 27);
        if (c == '0'
            cout << ' ';
        else
            cout << c;
    }
}
```

If we crank up this electronic monkey, we get output that looks like this:

```
CWOERUDNYVYNJS FCULQCH BKEDWDBOMJPJFOMZITWZLID
TCXGA KOLCENMGKSUM OT CDFZIFERBIHVXTWYKHZZNB
EFDXVQKMGILA HZYLKWRVXMSVPVPPXPYNTCV SKBFMZYP
ZODBCBTBOGLXKOSI HLBP ENHGSZCJMBKJVEE EY RHYSYWZ
UUFJCDUW GMVDMZSMGCWMDOR
```

and so on. Evidently, we'll have to wait a long time before we get even one recognizable English word, much less the text of *Hamlet*. In fact, even producing output at the rate of 1000 characters per second, we can show that we should expect to wait about 93 billion years to find the phrase “WORKING CLASSES.”

There are some obvious reasons that this random text doesn't resemble English: There are far too few vowels, and far too many uncommon letters like X, and the “words” are way too long. (You should be able to convince yourself that the average word length for text produced this way is 26.) All these problems have a common cause, namely, that the distribution of characters in the output text is *uniform*, which is to say that each character is as
likely to occur as any other. But we know that the characters in an English
text are decidedly not uniformly distributed; the space is the most common,
accounting for about 19% of all characters, followed by E, which accounts
for about 10% of the characters, and so on, to Z (or J, depending on what
authority you read).

This suggests an immediate improvement over our first program. We will
take a typical text and produce a frequency table, like the following, listing
all the characters and the percentage of time each appeared in the text.

<table>
<thead>
<tr>
<th>Char.</th>
<th>Frequency</th>
<th>Char.</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blank</td>
<td>0.18050</td>
<td>N</td>
<td>0.05764</td>
</tr>
<tr>
<td>A</td>
<td>0.05837</td>
<td>O</td>
<td>0.05993</td>
</tr>
<tr>
<td>B</td>
<td>0.01310</td>
<td>P</td>
<td>0.01733</td>
</tr>
<tr>
<td>C</td>
<td>0.02615</td>
<td>Q</td>
<td>0.00128</td>
</tr>
<tr>
<td>D</td>
<td>0.02634</td>
<td>R</td>
<td>0.05468</td>
</tr>
<tr>
<td>E</td>
<td>0.10679</td>
<td>S</td>
<td>0.05308</td>
</tr>
<tr>
<td>F</td>
<td>0.02180</td>
<td>T</td>
<td>0.09122</td>
</tr>
<tr>
<td>G</td>
<td>0.01352</td>
<td>U</td>
<td>0.02201</td>
</tr>
<tr>
<td>H</td>
<td>0.03927</td>
<td>V</td>
<td>0.00797</td>
</tr>
<tr>
<td>I</td>
<td>0.06290</td>
<td>W</td>
<td>0.01531</td>
</tr>
<tr>
<td>J</td>
<td>0.00053</td>
<td>X</td>
<td>0.00250</td>
</tr>
<tr>
<td>K</td>
<td>0.00338</td>
<td>Y</td>
<td>0.01093</td>
</tr>
<tr>
<td>L</td>
<td>0.03515</td>
<td>Z</td>
<td>0.00057</td>
</tr>
<tr>
<td>M</td>
<td>0.01686</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Using this table, we will pick letters based on their likelihood of occurrence
in the source text. To make the table shown here, we preprocessed the text
of Chapters 5 and 6 of this book, changing all letters to capitals, stripping
away everything that was not a letter or a blank, compressing all strings of
consecutive blanks to a single blank, and replacing each blank by "@". The
resulting text consisted of 139,528 characters and was used as input for
every program in the rest of this chapter. From this source file our frequency
table was prepared, and the table was used by the following function to pro-
duce an output text.

```c
void GenerateText(double freq[])
{
    for (int i = 0; i < 1500; i++)
    {
        double r = double(rand() % 32767) / 32767.0;
        int index = 0;
        double cumulative = freq[index];
        while (cumulative < r)
            index++;
```
The only part of the algorithm that may be unfamiliar is the process of choosing the character. To pick a character with probability equal to its frequency, we first choose a number $r$ that is uniformly distributed in the range $[0, 1]$ and move through the array from '@' to 'Z', keeping a cumulative sum of the frequencies. We stop when the cumulative frequency is greater than or equal to $r$. In Figure 10.1, for instance, if the number $r$ was 0.2471, we would pick 'B', since index 'B' is the first time the cumulative frequency is greater than or equal to 0.2471.

We see that we would pick 'B' only when $r$ is greater than 0.2392 and less than or equal to 0.2523. Since $r$ could equally well take on any value from 0 to 1, the probability of stopping at index 'B' is equal to the length of the 'B' segment, which is $0.2523 - 0.2392 = 0.0131$, exactly as we wanted. In Appendix C, there is a more detailed description of generating a random variable according to a predefined probability distribution.

When we run this algorithm, we get the following output, in part:

```
EEBS NE OV AD IICEEMTCI OW HGTLI HPNLRFICMRONSSITAT
ISN AOVNYSY T IKT SHF BT SETE EE AHTCDTH UUSHOYUEKYI
IRSAEIA H NO NTSS OTMOHR DHPKIVK NLEEAA A STRARAORT
GHLIEQR E M NTI T TISGAPCANDISHFEGSGIOIN NSURAK
EHIMHRS TSMR YNPIITOTOTENFETBAIENCAIUHNA 1 NHIEFE
M AAHACN CTEIG Ex T TEIA O H U E FF LHIUONWRU HNIRY L L
PSSIFLS ISS HA ICLHHN EG ECSEOSELF OOU DLIR ELSSUNWILH
```

That's a considerable improvement over our first sample. The word lengths look more like what we would expect in English (although there are still too many very long words), the letters occur as often as we would expect them, and we even have some recognizable words, like ALE, NON, l, AS, NO (twice), and A (four times). In the full 1500-character output, there were 14 recognizable words out of 230, or about 6%. In addition, there were several words that could have been English, in that they sound right to a native

---

* At 1000 characters per second, we should expect to have to wait a mere 22 billion years to find "WORKING CLASSES" in the output.
FIGURE 10.1
Producing a random variable with a given distribution

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>'G'</td>
<td>'A'</td>
<td>'B'</td>
<td>'C'</td>
<td>'Z'</td>
</tr>
<tr>
<td>Frequency</td>
<td>0.1806</td>
<td>0.0586</td>
<td>0.0131</td>
<td>0.0261</td>
</tr>
<tr>
<td>Cumulative</td>
<td>0.1806</td>
<td>0.2392</td>
<td>0.2523</td>
<td>0.2784</td>
</tr>
</tbody>
</table>

speaker, such as SETE, OT, RIOD, and MESNO. Still, the output would probably be recognizable as garbage, even to someone with only a passing familiarity with written English.

One problem with this reconstituted text is that there are a number of groups of letters that either never occur in English or are very rare, like LHNN, JXA, QR, and RRR. This points out something that is obvious when you stop to consider it: There is a close connection between any string of characters in a text and the possible characters that can immediately follow that string. In English, the letter Q, for example, is almost invariably followed by U, and TH is likely to be followed by a vowel or R, but very unlikely to be followed by N, in spite of the fact that N is a relatively common letter in English. We will make use of this property to further improve our text regeneration algorithms.

Instead of preparing a frequency table of single characters, we will construct many frequency tables, one for every string of a fixed length in the input text. We will set a group size, g, and for each string of length g = 1 in the input, we will construct a frequency table for all the characters in our alphabet, containing, for each character c, the frequency with which c follows the prefix string. For example, if the group size was g = 2, we would have 27 frequency tables, and the table for 'Q' would have 1.00 (or very close) in entry 'U', while all the rest of the entries would be zero (or very close), reflecting the fact that the prefix 'Q' is almost 100% likely to be followed by 'U' and is almost never followed by any other character. Having made these frequency tables, we would then construct our output text by repeating the following steps as often as we wished:

1. Begin with any prefix of length g = 1. We will denote this prefix by \( P = p_1, p_2, \ldots, p_{g-1} \).
2. Look in the frequency table for \( P \) and choose a suffix character, \( c \), with probability equal its frequency in \( P \)'s table.
3. Print \( c \).
4. Make a new prefix by setting \( P \) equal to \( p_2, p_3, \ldots, c \); in other words, strip the oldest character from the front of \( P \) and place the newly found character at the end to make the new \( P \).

We can view this as a form of computerized solitaire, if you will. This game dates back to work done in the late 1940s by Claude Shannon. It has been a part of computer science folklore for a number of years; anyone who has hung around a computer center long enough has likely heard of it. It goes
by a number of names, the most popular probably being Travesty. Travesty has been the subject of several amusing articles, both popular and scholarly; we cite some of them in the summary and recommend all of them highly. The shell of the Travesty program is simple enough: We read the input text and process it to form the frequency tables, and then we use those tables to generate the reconstituted text as output.

```c
const int GROUP_SIZE =  // Number of characters in each substring
    PREFIX_SIZE =  // One less than group size

void Initialize(GroupStructure& s)
{
    // Set initial values for s and any other variables.
}

void BuildStructure(GroupStructure& s)
{
    // Read the input textfile and construct frequency tables.
}

void GenerateText(GroupStructure s)
{
    // Print out text as constructed by the rules of Travesty.
}

void main()
{
    GroupStructure s;  // Some structure to store frequency tables
    Initialize(s);
    BuildStructure(s);
    GenerateText(s)
}
```

This, then, will be the subject of the rest of this chapter: How to program Travesty. You certainly have the necessary tools, since by now you have been exposed to about a dozen abstract data types, with perhaps twice that number of implementations. There is a nice symmetry here, since in this last chapter we are again dealing with the ideas of the first chapter, namely, the choice of data structure to solve a problem. We will consider a number of possible representations for the frequency tables and will weigh them according to how efficiently they make use of both storage and time.

**THE SOLUTIONS**

What follows is a description of our quest for a data structure that is capable of storing the frequency tables of increasingly larger orders of models for the
Travesty game. We'll present our solutions in pretty much the same sequence as we devised them, and you'll see that the process is an example of what happens so often in the real world: We find a solution that works for a small instance of the problem, only to discover that our solution simply won't work on larger instances. We then have to discard our prior solution and use what we know about data structures to seek a better one, only to find that our new solution still fails on even larger instances.

Arrays

The simplest way to store the frequency tables is to use an array with as many dimensions as the size of the character groups. Starting with groups of 2, this means that we would use an array \([27][28]\) of integers to store the frequencies. The first index (or indices, in case the group size is larger than 2) will represent the prefix, so that anything in the array with first index 5, for instance, will be taken to be part of the frequency table for the prefix 'E', the fifth character in our ordering. Doing that, the \([5][6]\) entry of the array will count the number of times we have seen an 'E' followed by an 'F'. It will be easy enough to extend this to longer prefixes by just increasing the number of dimensions. We'll overload the second dimension by using the 27 coordinate to store the total number of entries for a given prefix, since that will eliminate the need to recompute the frequencies in a table every time we read a character. With this convention, the \([5][27]\) entry is the total number of strings with prefix 'E'.

The program read and processed the entire input file (139,528 characters, remember) in 2.7 seconds and printed the reconstituted text far faster than it could be read. Part of the output was

```
ULEEES I O AD  THEQUT  NT  THE   THERE  WERE AL FOR A  ON PE
BANG KE U  ATH  FISTHEW  CUBE  FUMOS  AN  TH  APRURDEPRE
KND  CHAT  RY AT  F ACONOTIGGE  HERNG  TE  D R A  FES  BYN
TUSINTO PAL  VE Cot irerd CT  FUNSTH N PT  BI  IVEE  AM  ARS  IG
HE  CHETPATHIFONCTRES  DE  THE  ABOUPLE  THOF  F LERO
ASTED  THTO  THE  ON  TOR R T  THAPONOO  TTH  KELOROD
NSUG  TH  FINE  LGHE  BLERE T CE A  ALLONDE  AK
```

Of the 298 words in the output, 48 (16%) were recognizable English, and there were a lot of words that could be English, like DEMIORD, OTIO, COPIM, CALIMABE, and—our favorite—BLEXEMISE. Almost all the words were pronounceable, which is a considerable improvement over the first two samples. It would be most interesting to hear what would happen if this output were sent to a good voice synthesizer.

This means of generating events whose outcome depends on one prior event is called an order 1 Markov process. The program used all 27 possible
prefixes, which is no surprise, but encountered only 511 two-character
groups. That means that when we extend to an order 2 Markov process,
using two-character prefixes, it will be using only about two-thirds of the
$27^2=729$ first two indices. If we do rewrite the program to use prefixes of size
2, we get our first unpleasant surprise: The program crashes. Let’s see how
big our GroupStructure array is: 27 first indices, 27 second indices, 28 third
indices, for a total of $(27^2)28=20,412$, at 2 bytes per integer entry, or 40,824
bytes for the array. Well, that is indeed a problem, since the compiler doesn’t
allow variables to be larger than 32,767 bytes long. So, it’s back to the drawing
board. We should mention that changing computers and compilers is no
real help; it only postpones the problem. It might run just fine for groups of
three, but groups of four would take 1,102,248 bytes, and a megabyte just
for one array variable is probably too much to expect of almost any system.
In fact, is is easy to see that the amount of space, $S_A(n)$, in bytes required by
this array data structure for an order $n$ process is

$$S_A(n) = 2 \times 28 \times 27^n$$

Certainly, we would like to avoid an algorithm for which the space required
increases exponentially as the size of the problem, and we will improve this
performance shortly. See if you can come up with an improvement before
you turn to the next section.