Once the right-to-left packaging process is completed, a left-to-right labelling stage similar to that already detailed in Algorithm 4.4 is required to calculate the lengths of the paths to the original symbols. In the case of the example, this generates the output code (which can for brevity also be described in a runlength form) $|C| = [(2;1),(3;4),(4;4)]$, that is, one codeword of length two; four of length three; and four of length four.

Both the packaging and labelling processes are extremely fast. The last package generated, regular or irregular, will have weight $m$, where $m$ is the length of the source message that gave rise to the probability distribution, and so exactly $\lfloor \log_2 m \rfloor$ levels are required. Furthermore, each step in the right-to-left packaging process, and each step in the left-to-right labelling process, takes $O(1)$ time. Total time is thus $O(\log m)$. To put this into perspective, the `WSJ.Words` data has $m = 86 \times 10^6$, and $\lceil \log_2 m \rceil < 27$. Including both phases, fewer than 60 loop iterations are required.

A generalization of this mechanism is possible. Suppose that $k$ is an integer, and that $T$ is the $k$th root of two, $T = \sqrt[k]{2} = 2^{1/k}$. Suppose further that the input probability distribution is such that all weights are integer powers of $T$. For example, with $k = 1$ we have the situation already described; when $k = 2$ we allow weights in

$$[1, \sqrt{2}, 2, 2\sqrt{2}, 4, 4\sqrt{2}, 8, \ldots] \approx [1, 1.41, 2, 2.83, 4, 5.66, 8, \ldots],$$

and so on. Then by adopting a similar algorithm – in which symbols or packages at level $d$ combine to form new packages at level $d + k$ – a minimum-redundancy code can be constructed in $\log_T m$ steps, that is, in $O(k \log m)$ time and space [Turpin and Moffat, 2001].

By now, however, the sceptical reader will be actively scoffing – why on earth would a probability distribution consist solely of weights that are powers of some integer root of two? The answer is simple: because we might force them to be! And while this may seem implausible, we ask for patience – all is revealed in Section 6.10 on page 179.

## 4.8 Doing the housekeeping chores

It is tempting to stop at this point, and say "ok, that's everything you need to know about implementing minimum-redundancy coding". But there is one more important aspect that we have not yet examined – how to pull all the various parts together into a program that actually gets the job done. That is what this section is about – doing the coding housekeeping so that everything is neat, tidy, and operational. In particular, we now step back from the previous assumptions that the source alphabet is probability-sorted and consists of integers in $S = [1 \ldots n]$, all of which have non-zero probability. We also admit that