PK Font Compression

Obviously, these words are hard to read because the individual characters feature different styles and sizes. In a beautifully-typeset document, all the letters of a word and all the words of the document (with perhaps a few exceptions) should have the same size and style; they should belong to the same *font*.

Traditionally, the term "font" refers to a set of characters of type that are of the same size and style, such as Times Roman 12 point. A typeface is a set of fonts of different sizes but in the same style, like Times Roman. A typeface family is a set of typefaces in the same style, such as Times.

The size of a font is normally measured in points (more accurately, printer's points, where 72.27 points equal one inch. The style of a font describes its appearance. Traditional styles are roman, **boldface**, *italic*, *slanted*, `typewriter`, and sans serif.

Old operating systems did not support fonts. Even the DOS operating system, which was widely used on IBM-PC-compatible personal computers from 1980 to 1995, did not support fonts and was based on ASCII codes instead.

It was not until the mid 1980s that digital fonts became a part of many operating systems. In 1984, Adobe launched the PostScript language, which supported two types of digital fonts. In the same year, Apple released the first models of the Macintosh computer, whose operating system used fonts (Figure 1.1 illustrates some of the early fonts included in the Macintosh OS 6). In 1985, Apple announced the LaserWriter, one of the first laser printers available to the mass market. These developments paved the way for future operating systems to support digital fonts, thus enabling users to create, print, and publish beautiful, professionally-looking documents.

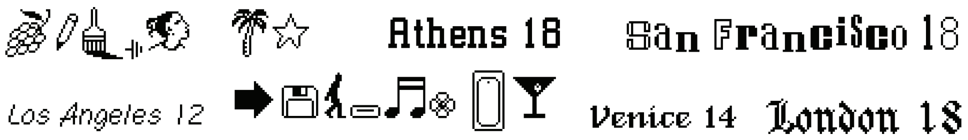

Figure 1.1: Old Macintosh OS 6 Fonts.

A digital font is a set of symbols or characters that can be stored in the computer's memory or on an I/O device such as a disk. The symbols of the font are either displayed or printed. Each symbol consists of a glyph (the actual shape of the symbol) and bookkeeping information, such as the height, depth, and width of the symbol.

Modern fonts are referred to as outline fonts. A symbol in such a font is fully defined by a small set of control points that are connected by curves to form the outline of the symbol. The symbol π in Figure 1.2 is an example. It is easy to see how this symbol is fully defined by about 30 control points that are connected with a few smooth curves. An outline font is easy to store in a computer file simply by storing the coordinates of the control points. It is also easy to change the size of the font by scaling the coordinates of the points.

In contrast, early digital fonts were of the bitmap variety. The font designer would draw the glyph of each symbol on a grid of small squares (pixels) and then select the

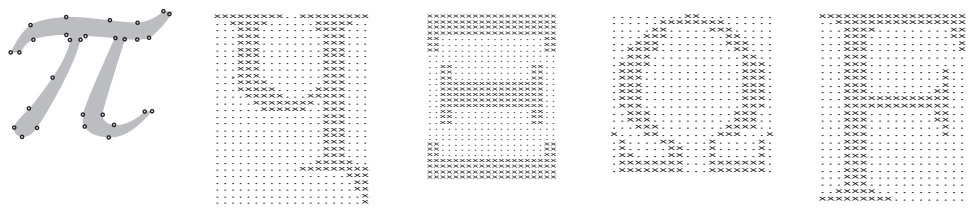
Figure 1.2: Outline and Bitmap Fonts.

best pixels for the symbol. Figure 1.2 shows four examples of bitmap symbols where the white pixels are represented by dots and the black pixels are shown as an `x`. In the computer's memory, a bitmap symbol is stored as a map of bits, zeros for white pixels and 1's for black pixels. When such a font is written in raw format on a file, the file tends to be large and should be compressed. Also, a high-quality bitmap font has to be designed from scratch for each font size. Simply scaling a bitmap results in badly-looking characters.

The TEX project, started by Donald Knuth in 1975, is an early example of the use of bitmap fonts. The goal was to develop software for typesetting beautiful documents, especially documents with a lot of mathematics. Such software requires a set of fonts, even if the operating system does not support fonts. Thus, METAFONT, an application to generate fonts, was developed in parallel with the TEX application itself. METAFONT was then used to create a set of fonts that is referred to as computer modern (CM).

The CM set of fonts consists of 75 fonts that are described in volume E of [Knuth 86] as well as the "line," "circle," and symbol fonts associated with LaTeX.

The output of METAFONT is called a generic font (GF), to indicate that this file format does not follow the conventions of any font foundry. However, it is easy to convert GF font files to the special format required by most digital phototypesetting equipment.

In 1985, Tomas Rokicki, a student of Knuth's, developed the PK (for PacKed) font format. The idea was to develop a simple, fast compression method such that the fonts would be saved in small files and even the slow computers available at that time would be able to decompress a font, or any part of it, quickly. The main references for the format and organization of PK fonts are [Rokicki 95], [Rokicki 90], and [Haralambous 07].

The method selected for PK compression was based on run-length encoding (RLE) and on the special features of font bitmaps. The method is not very efficient, but it is described here because it offers an original approach to RLE, an approach that does not use Huffman codes and makes minimal use of variable-length codes (the packed numbers, described later, are the only variable-length code used by PK). A quick glance at the four bitmaps of Figure 1.2 shows two important features (1) they are narrow, resulting in mostly short runs of black and white pixels and (2) they tend to have consecutive identical rows of pixels.

This section discusses the compression of PK fonts, but a full understanding of this compression method requires a little knowledge of the organization of these fonts and of the way TEX employs font information.

When a font of type is designed, the designer determines, for each symbol in the font, its glyph and its dimensions. Symbols in a font are two-dimensional, but each has three dimensions, height, depth, and width. It is best to think of the symbol as if it is surrounded by a box (or a bitmap) as illustrated by the leftmost item of Figure 1.3. There is a baseline that separates the height and depth of the box, and there is a reference point at the left end of the baseline.

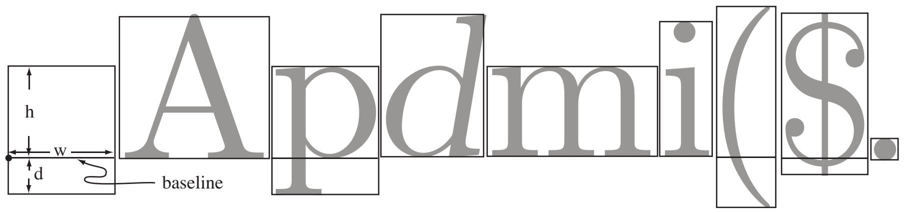

Figure 1.3: Three-Dimensional Character Boxes.

TEX uses the three dimensions of each font symbol but is not concerned with the actual shape of the symbol. As a result, symbols may stick out of their bitmap boxes, as illustrated by the italic "*d*" in the figure. TEX reads the input text and strings boxes horizontally to construct a line of text. The boxes butt together (in Figure 1.3 there are small spaces between boxes for better readability), so the font designer leaves spaces to the left and right of each symbol, as illustrated in the figure. (In those rare cases where a symbol, such as an m-dash, should touch their neighbors, the bitmap box is as wide as the symbol and no spaces are left.)

When the current line of text is ready, TEX starts on the next line. When that line is also ready, it is placed under the current line (and it then becomes the new current line) such that the baselines of the two lines are separated by a user-controlled parameter. If this causes the lines to overlap (because the top line has a symbol with a large depth and the bottom line has a symbol with a large height), TEX leaves some space (Figure 1.4) between the bottom of the upper line (the symbol with the largest depth) and the top of the lower line (the symbol with the largest height). This space can also be controlled by the user, but the font designer does not have to worry about it. Thus, there are spaces to the left and right of font symbols in their bitmap boxes but no spaces above and below them.

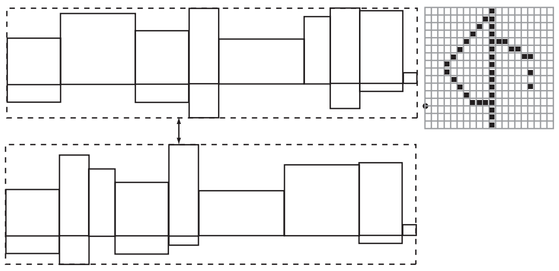

Figure 1.4: Two Lines of Text.

PK compression is based on run-length encoding. It encodes the runs of black and white pixels in the bitmap of a symbol. In order to decrease the run lengths and thereby increase compression performance, the encoder first determines the bounding box of a bitmap, as illustrated by the 16×20 bitmap of Figure 1.4. The bounding box of a glyph bitmap is the smallest rectangle that encompasses all the black pixels of the glyph. It is easy to see that our bounding box is 15 pixels wide and 16 pixels tall. It starts at column 3 from the left and 13 of the 16 pixel rows are above the baseline. Once the bounding box is decompressed, the decoder needs the following dimensions to create the complete bitmap: The horizontal escapement (the width of the bitmap, 20), the width of the bounding box (15), the vertical escapement (the height of the bitmap, 16), the height of the bounding box (also 16), the horizontal offset (the distance between the reference point and leftmost column of the bounding box, 2), and the vertical offset (the distance from the reference point to the top of the bounding box, 13). Three of these dimensions are horizontal and three are vertical, but often only two vertical dimensions are needed, because the bitmap and bounding box tend to have the same height.

We are now ready to look at the details of the compression. A PK font file is organized in records, one for each symbol. A record starts with a character preamble that contains (in addition to other information) the five dimensions mentioned above and one bit that specifies the top-left pixel of the bounding box (1 for a black pixel and 0 for a white one). The character preamble is followed by the encoded run lengths of the symbol's pixels and by the preamble of the next character. Because of the use of nybbles to encode run lengths, the preamble may start in the middle of a byte. There are also a preamble and postamble for the entire file.

The first step of the encoder is to locate groups of repeating (i.e., identical consecutive) pixel rows in the bounding box. When a group of n such rows is located, $n-1$ of them are removed from the bounding box, and a repeat count of $[n-1]$ is suitably encoded between runs in the remaining row of the group. Figure 1.5 illustrates this process. Part (a) of the figure shows a 5×15 bitmap (note that this is not a bounding box) where the three middle rows repeat. In part (b), two of these rows have been deleted and part (c) shows the valid points between runs where the repeat count [2] may be inserted.

```
000001111000000   000001111000000   000001111000000   000001111000000
011100111101100   011100111101100   011100111101100   000000000000000
011100111101100   000111111000011    ↑  ↑ ↑   ↑↑ ↑    000000000000000
011100111101100                     000111111000011   000000000000000
000111111000011                                       000000000000000
                                                      110000000000000
      (a)               (b)               (c)               (d)
```

Figure 1.5: Repeating Rows in a Bitmap.

Thus, the run lengths of this example become the following sequence 5, 4, 7, [2], 3, 2, 4, 1, 2, 5, 6, 4, and 2. The repeat count was placed at the first valid point, but could have been placed at any of the other five points indicated by the figure. The repeat count for a group of identical rows is placed in the remaining row between runs of pixels, but this rule fails in bitmaps such as the one of Figure 1.5d, where the repeating rows are uniform and are preceded by a run of the same color pixels. In such a case, the encoder does not delete the repeating rows and ends up encoding a long run (66 white pixels in the figure), which reduces the overall compression somewhat.

It is now clear that we have to encode two types of data, run lengths of pixels and repeat counts. Since a typical bounding box is small, we expect most run lengths to be short and most repeat counts to be small. Thus, the principle of PK compression is to pack, whenever possible, two items in the two nybbles of a byte. Often, the first nybble is a flag and the second nybble is a data item. A nybble is four bits long and can have 16 values. We expect to have more run lengths than repeat counts, so we reserve nybble values 14 and 15 to indicate repeat counts. A nybble of 15 indicates a repeat count of 1 (the most common) and a nybble of 14 will be followed by a repeat count stored as a packed number (a term to be discussed shortly).

The remaining nybble values 0 through 13 indicate run lengths. Value 0 indicates a long run, occupying three or more nybbles. The length of the run is stored in as many nybbles as needed, encoded as a packed number. Values 1 through 13 indicate the short and medium run lengths. In order to use these 13 values to maximum effect, the encoder counts the run lengths of the bounding box of the current character, and uses their values, in a fast, one-pass algorithm, to compute a variable *dyn*. This variable is computed for each character of the font and is stored in the character's preamble. Variable *dyn* is used as follows: run lengths 1 through *dyn* are considered short and are stored in one nybble each. The medium run lengths are stored in two nybbles, the first of which, to be denoted by a, is between $dyn + 1$ and 13 and the second one, b, can be any 4-bit value. The run length represented by a and b is defined as $16(a - dyn - 1) + b + dyn + 1$. The shortest run length is therefore $dyn + 1$ (for $a = dyn + 1$ and $b = 0$) and the longest is $16(13 - dyn) + dyn$ (for $a = 13$ and $b = 15$). This convention is best illustrated by examples.

We first select $dyn = 4$. In this case, run lengths 1 through 4 are the short ones and are represented by one nybble each, thus 0001, 0010, 0011, and 0100. The medium run lengths are 5 (for $a = 4 + 1$ and $b = 0$) through $16(13 - 4) + 4 = 148$ (for $a = 13$ and $b = 15$). Runs of more than 148 pixels are considered long and will start with a zero nybble.

We now select $dyn = 12$. In this case, run lengths 1 through 12 are short and are represented by one nybble each. The medium run lengths are 13 (for $a = 12 + 1$ and $b = 0$) through $16(13 - 12) + 12 = 28$ (for $a = 13$ and $b = 15$). Runs of more than 28 pixels are considered long.

It is clear that the value of *dyn* is critical. Small *dyn* values allow for a large range of medium run lengths, while large values of *dyn* should be used in cases where most run lengths are short.

Next, we discuss the format of a packed number. Given an integer i, the idea is to create its hexadecimal representation (let's say it occupies n nybbles), remove any leading zero nybbles and prepend $n-1$ zero nybbles. Thus, i values 1 through 15 occupy one nybble (nothing is prepended). Values $16 \le i \le 255$ are two nybbles long, so one zero nybble should be prepended, for a total of three nybbles. Values $256 \le i \le 4{,}095$ occupy three nybbles, so two nybbles should be prepended, for a total of five nybbles.

This variable-length format is used to encode repeat counts (indicated by a nybble flag of 14) and the long run lengths (indicated by a nybble flag of 0). However, the long run lengths are always greater than $16(13 - dyn) + dyn$. The shortest of the long run lengths is therefore $s \overset{\text{def}}{=} 16(13 - dyn) + dyn + 1$, which is why it makes sense to subtract s from such a run length before it is encoded as a packed number. Recall that the long run lengths are indicated by a nybble flag of 0, and a packed integer in the interval $[16, 255]$ is also preceded by a single zero nybble. Thus, it makes sense to add 16 to the long run lengths after s is subtracted.

Example. If $dyn = 4$, the longest medium run length is 148, so $s = 149$. Thus, a run length of 200 is long and is encoded by computing $200 - 149 + 16 = 67 = 43_{16}$ and constructing the 3-nybble packed number 043_{16}.

The algorithm for determining the optimal value of dyn (between 1 and 13) can now be described. This algorithm is executed after the sequence of run lengths has been determined. We start with a naive version of the algorithm. For each value of dyn between 1 and 13, it is easy to compute the ranges of the short, medium, and long runs. Each run in the sequence is examined, its type (short, medium, or long) is easily determined, and its length after being encoded is computed (short runs are one nybble long, medium runs are two nybbles, and long runs occupy three or more nybbles and their lengths take a bit more work to determine). The 13 total lengths for the values of dyn are saved and their minimum is then found.

A better version utilizes the fact that as dyn is increased, the range of short run lengths increases while the range of medium run lengths decreases (for $dyn = 4$ this range is $[5, 148]$ but for $dyn = 12$ it is only $[13, 28]$). Thus, a run of pixels that was medium for a certain value of dyn may remain medium but may also become short or long when dyn is incremented by 1.

This version of the algorithm starts by setting dyn to zero. Thus, initially there are no short runs. The algorithm goes over all the run lengths. It determines the type and computes the encoded length of each run. The lengths are added into variable `total`. The algorithm then increments dyn by 1 and goes over all the runs again. If a run was short in the previous iteration, it will remain short. If it was medium and has now become short, it decreases the value of `total` by 1. If it was medium and has now become long, it increases the value of `total` by 1 (or in rare cases, by 2). After each iteration, the new value of `total` is saved in an array. After 13 iterations, the smallest `total` is located in the array and is used to determine the optimal value of dyn.

The PK compression method described so far is simple, but not very efficient. Given a bitmap with many short runs (such as a gray character, where black and white pixels alternate), this RLE-based method may produce large expansion. In such cases, the run length encoding described here is abandoned and the bitmap is simply written on the font file in raw format (one bit per pixel).

We end with a summarizing example. The middle bitmap of Figure 1.2 is the Greek letter Ξ (Xi). Its size is 20×29 pixels and it has several groups of repeating rows, although only five groups are not uniform and can employ repeat counts. Counting the runs of pixels produces the sequence 82, [2], 16, 2, 42, [2], 2, 12, 2, 4, [3], 16, 4, [2], 2, 12, 2, 62, [2], 2, 16, and 82. The optimal value of dyn turns out to be 8, but in order to make this example useful and have short, medium, and long runs, we assume that dyn is set to 12. Thus, the short runs are 1 to 12 pixels, the medium runs are 13 to 28 pixels, and the long runs are longer than 28 pixels.

The first run is 82; a long run. Subtracting $s = 29$ and adding 16 yields $69 = 45_{16}$. This is encoded as the three nybbles `045`. The repeat count of 2 is encoded as the nybble $14 = E_{16}$, followed by the packed number representation of 2, thus `E2`. The medium run of 16 is encoded in two nybbles `ab`, but since a is between $dyn+1$ and 13, its value must be $13 = D_{16}$, implying that $b = 4$ and the run of 16 is encoded as `D4`. The next run, 2, is short and is encoded as the single nybble `2`. The run 42 is long. We first compute $42-29+16 = 29 = 1D_{16}$ and then encode `01D`. The next two runs [2] and 2 are encoded as `E22`. So far we have `045E2D4201DE22`. The next run, 12, is short and is encoded as the single nybble `C`. The remaining runs do not provide any new examples and should be encoded by the reader as an exercise.

References

Haralambous, Yannis (2007) *Fonts and Encodings*, (Translated by P. Scott Horne), Sebastopol, CA, O'Reilly Associates.

Knuth, D. E. (1986) *Computers and Typesetting*, Reading, Mass., Addison Wesley.

Rokicki, Tomas (1985) "Packed (PK) Font File Format," *TUGboat*, **6**(3):115–120. Also available online at `http://www.tug.org/TUGboat/Articles/tb06-3/tb13pk.pdf`.

Rokicki, Tomas (1990) "GFtoPK, v. 2.3," available at `http://tug.org/texlive/devsrc/Build/source/texk/web2c/gftopk.web`.