10. Return the elements of a numeric array found at given coordinates.

If the dyadic function RE provides the solution, it should work as follows:

```
    ⎕←A←3 4⍴78 90 22 2.3 43.9
        92 12 67 23 33 88
        9.34
 78   90 22  2.3
 43.9 92 12 67
 23   33 88  9.34
    ⎕←B←2 2⍴1 3 2 4
 1 3
 2 4
    A RE B
22 67

    ⎕←C←2 5 6⍴60?1000
 554 684 485 119 559  530
 193 631 783   1 838  366
 380 987 986 224 269  670
 572 312 314 779 160  285
 168 883 895 230 361  764

 763 891 592  47  51  589
 705 297 689 215 208 1000
 688 772 946 957  16  721
 172 822 982 379 781  163
 797  12 702 723 847  735
    D←⊃(1 3 4) (1 5 5) (2 1 6)
    C RE D
224 361 589
```

Restrictions: assume that a looping solution is not allowed and that index origin is 1.

If you are a developer working with APL, you should be able to propose at least three solutions or idioms in respect of the problems above. The restrictions imposed in formulating the solution should help in determining an idiom.

Subject to the readers' active participation —via correspondence with the author— the solutions are posted at the end of this issue.

A Boundless Compression Algorithm in APL2

Manuel Alfonseca

Introduction

Compression algorithms are very useful for a variety of applications. Originally, their main use was what their name suggests, reducing the size of computer files without losing any information. It was interesting to see that they were equally useful for all types of files, such as text, bit maps, many types of images, sound, scientific experimental data, computer code (source and object code), combined data (text and images, for instance), and so on, and so forth. Only when the files have been coded with an imbedded compression algorithm (as happens in some image formats, such as JPEG, or in sound renditions in MP3) the additional compression by generalized algorithms is negligible.

The reason why compressors usually make a good work is the fact that human generated information is largely redundant. In fact, this redundancy is even greater than what is currently taken advantage of by typical compressors, as our languages includes lots of words with a null information content, that only make it simpler to understand what somebody else is speaking. This kind of redundancy is not used by current compressors to perform an even better job, as it would give rise to a change in the way the information is expressed, making it impossible to recover the original contents fully. Typical compressors work under the assumption that the information they compress is 100% recoverable, i.e. that the result of uncompressing the compressed file is identical to the original uncompressed file.

As an example of this, the Japanese language contains many words which are pronounced exactly in the same way, but have completely different meanings and syntactic values. For instance, the sequence `GO-GO-GO` might mean *five p.m.*, as *five*

is said GO, *after* is said GO, and *noon* is also said GO. The written language prevents mistakes by using different Kanji symbols for each of these meanings, which means that a single pronunciation may correspond to twenty or thirty different words. However, the spoken language cannot make use of these guides. A Japanese friend told me that what they do is to use redundancy (saying the same thing in different ways) so as to provide the missing information. Although this may be an extreme case, all human languages contain lots of this type of redundancy.

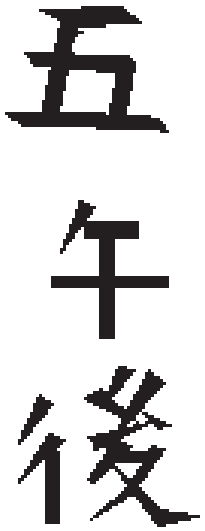

Figure 1. GO-GO-GO (5 p.m.) in Kanji.

Another way to see it is to compute the average number of letters in a word, or the average number of characters in a given text. These numbers are widely different for different human languages, because some languages include a larger amount of redundancy. For instance, Spanish words have an average of about six characters, while English words only reach five. Table 1 shows the size in characters of three of my novels, which I wrote originally in Spanish and later translated into English. It is clear to see that the amount of redundancy in Spanish is 5 to 20% greater than in English.

Book title	Thousand chars.	
	Spanish	English
The journey of Tivo the dauntless	203	182
The mystery of the Black Lake	194	156
The ruby of the Ganges	204	188

Table 1: Size comparison of the same texts in English and Spanish.

But this is not what the typical compressors do to attain compression. They don't look for hidden redundancies, which are not easy to find out, but just try to detect explicit redundancies, i.e. full repetitions of whatever the file contains (i.e. actual repetitions of byte strings). This is what makes the compressors independent of the file type, which although makes them not as good as they could be, on the other hand gives them their surprising type independency.

Problems found with standard compressors

There are many well-known compressors, such as PKZIP, WINZIP, GZIP, BZIP2, and many others [1-2]. Some of them use statistics [3], others algorithms of other types. Most aim to attain the maximum compression in the minimum time. The problem is that these two objectives are usually incompatible (i.e. to get a larger compression, the compressor should use more time), so that most compressors try to reach a balance between them, giving up some compression efficiency to attain a faster speed. This is done by setting certain areas (work blocks or sliding and look ahead windows) that move around the file to be compressed. All compression decisions are taken inside those blocks or windows, which means that redundancies outside them are usually not detected, and the compression won't be as good as it could be expected.

Let us consider, as an example, PKZIP, one of the best-known general-use compressors. The version of this compressor which I've been using appears to use a comparison window of 32 kBytes. How did I find this out? By building a set of high redundancy sample files, compressing them with

PKZIP, and comparing their original sizes with the sizes of the respective compressed files. To build the files, I took a set of text files of different sizes and concatenated them to themselves. In this way, the second half of each file is exactly equal to its first half, thus putting in the file a lot of redundancy which the compressor should take advantage to attain a high degree of compression. Table 2 shows the results of these experiments.

Observe that for files below 16 kBytes (i.e. files which concatenated to themselves are smaller than 32 kBytes) the size of the duplicated file is almost the same as the size of the non-duplicated file (with increases of about 3%). This happens because PKZIP was able to detect that the second half of the file was exactly the same as the first, and therefore could compress it a lot. However, for larger sizes, the full file does not fit in its work block, and PKZIP becomes unable to find the redundancy of larger and larger pieces of text in the latest part of the file, until, for file sizes larger than 32 kBytes, it is completely unable to find any redundancy between the two repeated sections, and the size of the compressed double file becomes almost twice as large as the size of the compressed single file.

Files in the sample set	Size of file (Bytes)	Size of PKZIP compressed files		
		File	File,File	%Δ
File 1	10075	4001	4132	3.3
File 2	15107	5433	5628	3.6
File 3	19928	6701	6981	4.2
File 4	24976	8218	9588	16.7
File 5	30099	9683	15875	63.9
File 6	39940	12893	24667	91.3
File 7	47839	15535	29902	92.5
File 8	68332	20334	39819	95.8

Table 2: Detection of work block size in PKZIP.

For certain experiments which we wanted to do, related to the automatic generation of music by using an approximation of the Kolmogorov complexity to compute the normalized information distance [4] we needed to be able to use a compressor which would not present the same problems as PKZIP. We made a similar study for two other compressors, GZIP and BZIP2 [5], and found that though one of them had a very large sliding window, so that the problem did not appear as quickly, sooner or later all of them came to limits similar as those found for PKZIP.

The LZ77 compression algorithm

There was, however, a solution: the LZ77 algorithm, one of the oldest compression algorithms [6], which can be implemented without any window limitation, and therefore is not subject to the above mentioned problems. This algorithm can be written as follows in APL2:

```
[0]    Z←LZ77 X;I;J;Q;Q1;Q2;
         Q3
[1]    Q←Q1←Z←''
[2]    I←0
[3]  L:→((I←I+1)>⍴X)/0
[4]    J←0
[5]  L1:Q2←Q ◊ Q3←Q1
[6]    →((⍴X)=I+J-1)/L2
[7]    Q←X[I+¯1+⍳J←J+1]
[8]    Q1←Q⍷X[⍳I-1]
[9]    →(~1∈Q1)/L2
[10]   →L1
[11] L2:→(J>1)/L3
[12]   Z←Z,Q
[13]   →L
[14] L3:Z←Z,⊂((⍋Q3)⍳1),⍴Q2
[15]   I←¯1+I+⍴Q2
[16]   →L
```

The following is a description of how this algorithm works:

- *X* is the text (or a Byte string of any type) that we want to compress.
- *Z* is the result of the compression, initialized to an empty vector.

- `I` is a pointer to `X` which controls the outer loop. This loop (and the algorithm) is ended when `I` reaches the end of vector `X`.

- Starting at `X[I]`, we take the next `J` characters and put them at variable `Q`. This is the inner loop around `L1`: `J` starts at `1` and ends when there are no more characters in `X` or when the longest section of `X` which had been found before is located (see next).

- We search for `Q` in the section of `X` already compressed. If found, we go on with the inner loop. If not found, the longest repeated section is the former value of `Q` (which has been kept in `Q1`).

- If `Q1` is empty (`J=1`), we put the next character in the result `Z` (it is a totally new character) and proceed with the outer loop.

- Otherwise, we have found the next longest repeated string. We add to the result the number of characters we must go back to find the beginning of the string, plus the size of the repeated string. Then we proceed with the outer loop.

We shall now apply this algorithm to compress the string `'abcdabpqabcdabpq'`:

- Initially, `I←0` and `Z←''`

- We set `I←1`, `J←1` and find that `Q←'a'` is not in the section already seen of `X` (which is empty). Therefore, this character is new and we add it to `Z`.

- We start again the outer loop at `I←2`. We set `J←1` and find that `Q←'b'` is not in the section already seen of `X` (which is `'a'`). Therefore, this character is new and we add it to `Z`.

- The same happens to `'c'` and `'d'`.

- We start again the outer loop at `I←4`. We set `J←1` and find that `Q←'a'` is in the section already seen of `X` (which is `'abcd'`). Therefore, this character is not new and we proceed with the inner loop.

- The next character is `'b'`. We add it to `Q` and find that `Q←'ab'` is in the section already seen of `X` (which is `'abcd'`). Therefore, this string is not new and we proceed with the inner loop.

- The next character is `'p'`. We add it to `Q` and find that `Q←'abp'` is not in the section already seen of `X` (which is `'abcd'`). Therefore, this string is not new and we exit the inner loop, add to `Z` the vector `(4 2)` which means that the repeated string `Q1` (the previous value of `Q`, `'ab'`) can be found by going back 4 characters in `X` and taking 2. After doing this, we go on with the outer loop, with `I←7` (to point again to the new character).

- Characters `'p'` and `'q'` are new, therefore they will be directly added to `Z`.

- The remaining characters, `'abcdabpq'`, are repeated, so they will be replaced by the vector `(8 8)`, meaning that they will be found by going back 8 characters in `X` and taking 8 characters starting there.

- With this, we have come to the end of X. The result of the algorithm will be `'abcd'(4 2)'pq'(8 8)`.

The inverse algorithm is straightforward: copy all characters from Z to X until a two element integer vector `(a b)` is found. Then go back `a` elements from the end of the `X` string just generated, take `b` elements from there and copy them to the end of `X`. Proceed like this until all the elements in `Z` are exhausted.

Performance considerations

This algorithm is not very fast, specially so because it is written in APL2 with loops, but it works as desired. Whatever the size of its argument, it will perform a perfect compression, without any bounds because of the existence of a slide window.

On the other hand, why write it in APL2? Couldn't better results be attained if it were written in a compilable language?

Doubtlessly, but we had to take into account that all our application algorithms for music generation are written in APL2. In our first experiment, when we started using PKZIP, we actually set an environment where everything was done in APL2, but when the compressor had to be invoked, the string to be compressed was written into a file and a batch program was started to invoke PKZIP, then the compressed file was read back into APL2 and processed there. All these steps made the experiment quite slow, and compensated the loss of performance when the `LZ77` function is used instead of PKZIP.

On the other hand, execution time was not our main concern. All applications using genetic algorithms (as our music generator does) are inherently slow. To attain an interesting piece of music takes several hours, and a fractional increase of the speed of the compressor wouldn't change this much.

In conclusion, we have found APL2 very useful for several applications which use genetic algorithms to generate music, interesting fractal curves with a given dimension [7], and other applications. While using an approximation of the Kolmogorov universal distance to compute the fitness of some of our genetic algorithms, we discovered a serious problem in most of the standard processors (such as PKZIP, GZIP and BZIP2) that made them practically unusable for our purposes. We needed a boundless compressor, and there was one available in the literature, LZ77, although we did not have an implementation. The easiest solution was to implement it in APL2, in this way attaining a tighter environment for our algorithms (everything was done in the same language). The function implementing the LZ77 algorithm is quite simple, as it contains just 16 lines of quite simple code. On the other hand, although this compressor is not fast, this was not a problem for our application, so we did not dedicate any time to try and improve the algorithm so as to make it as fast as possible.

We consider this work to be a good APL success story.

References

[1] T. C. Bell, G. J. Cleary, I. H. Witten, *Text Compression*, Prentice Hall, Englewood Cliffs, Jersey, 1990.

[2] M. Burrows, D. J .Wheeler, *A Block Sorting Lossless Data Compression Algorithm*, Research Center of Digital Equipment Corporation, Technical Report 124, Palo Alto, Calif. 1994.

[3] D. Huffman, *A Method for the Construction of Minimum Redundancy Codes*, Proc. IRE, Vol. 40:9, 1952.

[4] M. Alfonseca, M. Cebrián, A. Ortega, *Evolving Computer-generated Music by Means of the Normalized Compression Distance*, WSEAS Transactions on Information Science and Applications, Vol. 9:2, p.1367-1372, Sep. 2005.

[5] M. Cebrián, M. Alfonseca, A. Ortega, *Common Pitfalls using the Normalized Compression Distance: to Watch out for in a Compressor*, submitted to Communications in Information and Systems.

[6] J. Ziv, A. Lempel, *A Universal Algorithm for Sequential Data Compression*, IEEE Trans. On Information Theory, 23:3, p. 337-343, 1977.

[7] A. Ortega, A. Abu Dalhoum, M. Alfonseca: *Grammatical evolution to design fractal curves with a given dimension*, IBM Journal of Res. and Dev., Vol. 47:4, p. 483-493, Jul. 2003.