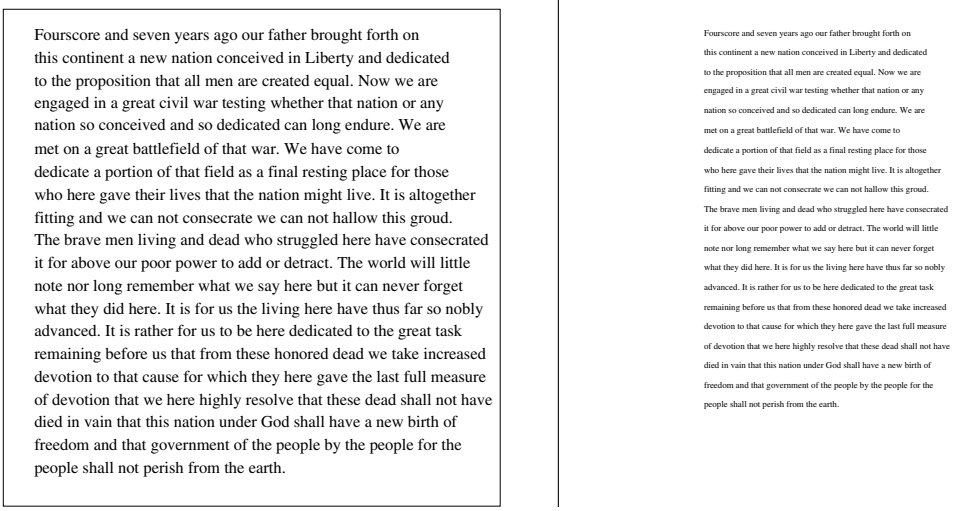

INPUT OUTPUT

18.5 Text Compression

Input description: A text string S.

Problem description: Create a shorter text string S' such that S can be correctly reconstructed from S'.

Discussion: Secondary storage devices fill up quickly on every computer system, even though their capacity continues to double every year. Decreasing storage prices only seem to have increased interest in data compression, probably because there is more data to compress than ever before. *Data compression* is the algorithmic problem of finding space-efficient encodings for a given data file. The rise of computer networks provided a new mission for data compression, that of increasing the effective network bandwidth by reducing the number of bits before transmission.

People seem to *like* inventing ad hoc data-compression methods for their particular application. Sometimes these outperform general methods, but often they don't. Several issues arise in selecting the right compression algorithm:

- *Must we recover the exact input text after compression? – Lossy* versus *lossless* encoding is the primary issue in data compression. Document storage applications typically demand lossless encodings, as users become disturbed

whenever their data files are altered. Fidelity is not such an issue in image or video compression, because the presence of small artifacts are imperceptible to the viewer. Significantly greater compression ratios can be obtained using lossy compression, which is why most image/video/audio compression algorithms exploit this freedom.

- *Can I simplify my data before I compress it?* – The most effective way to free space on a disk is to delete files you don't need. Likewise, any preprocessing you can do to reduce the information content of a file pays off later in better compression. Can we eliminate redundant white space from the file? Might the document be converted entirely to uppercase characters, or have formatting information removed?

  A particularly interesting simplification results from applying the *Burrows-Wheeler transform* to the input string. This transform sorts all n cyclic shifts of the n character input, and then reports the last character of each shift. As an example, the cyclic shifts of $ABAB$ are $ABAB$, $BABA$, $ABAB$, and $BABA$. After sorting, these become $ABAB$, $ABAB$, $BABA$, and $BABA$. Reading the last character of each of these strings yields the transform result: $BBAA$.

  Provided the last character of the input string is unique (e.g., end-of-string), this transform is perfectly reversible to the original input! The Burrows-Wheeler string is typically 10-15% more compressible than the original text, because repeated words turn into blocks of repeated characters. Further, this transform can be computed in linear time.

- *Does it matter whether the algorithm is patented?* – Certain data compression algorithms have been patented—most notoriously the LZW variation of the Lempel-Ziv algorithm discussed below. Mercifully, this patent has now expired, although legal battles are still being fought over JPEG. Typically there are unrestricted variations of any compression algorithm that perform about as well as the patented variant.

- *How do I compress image data* – The simplest lossless compression algorithm for image data is *run-length coding*. Here we replace runs of identical pixel values with a single instance of the pixel and an integer giving the length of the run. This works well on binary images with large contiguous regions of similar pixels, like scanned text. It performs badly on images with many quantization levels and random noise. Correctly selecting (1) the number of bits to allocate to the count field, and (2) the right traversal order to reduce a two-dimensional image into a stream of pixels, has a surprisingly important impact on compression.

  For serious audio/image/video compression applications, I recommend that you use a popular lossy coding method and not fool around with implementing it yourself. JPEG is the standard high-performance image compression

method, while MPEG is designed to exploit the frame-to-frame coherence of video.

- *Must compression run in real time?* – Fast decompression is often more important than fast compression. A YouTube video is compressed only once, but decompressed every time someone plays it. In contrast, an operating system that increases effective disk capacity by automatically compressing files will need a symmetric algorithm with fast compression times, as well.

Literally dozens of text compression algorithms are available, but they can be classified into two distinct approaches. *Static algorithms*, such as Huffman codes, build a single coding table by analyzing the entire document. *Adaptive algorithms*, such as Lempel-Ziv, build a coding table on the fly that adapts to the local character distribution of the document. Adaptive algorithms usually prove to be the correct answer:

- *Huffman codes* – Huffman codes replace each alphabet symbol by a variable-length code string. Using eight bits-per-symbol to encode English text is wasteful, since certain characters (such as "e") occur far more frequently than others (such as "q"). Huffman codes assign "e" a short code word, and "q" a longer one to compress text.

  Huffman codes can be constructed using a greedy algorithm. Sort the symbols in increasing order by frequency. We merge the two least-frequently used symbols x and y into a new symbol xy, whose frequency is the sum of the frequencies of its two child symbols. Replacing x and y by xy leaves a smaller set of symbols. We now repeat this operation $n - 1$ times until all symbols have been merged. These merging operations define a rooted binary tree, with the original alphabet symbols as leaves. The left or right choices on the root-to-leaf path define the bits of the binary code word for each symbol. Priority queues can efficiently maintain the symbols by frequency during construction, yielding Huffman codes in $O(n \lg n)$ time.

  Huffman codes are popular but have three disadvantages. Two passes must be made over the document on encoding, first to build the coding table, and then to actually encode the document. The coding table must be explicitly stored with the document to decode it, which eats into any space savings on short documents. Finally, Huffman codes only exploit nonuniform symbol distributions, while adaptive algorithms can recognize the higher-order redundancies such as in *0101010101...*.

- *Lempel-Ziv algorithms* – Lempel-Ziv algorithms (including the popular LZW variant) compress text by building a coding table on the fly as we read the document. The coding table changes at every position in the text. A clever protocol ensures that the encoder and decoder are both always working with the exact same code table, so no information is lost.

> Lempel-Ziv algorithms build coding tables of frequent substrings, which can get arbitrarily long. Thus they can exploit often-used syllables, words, and phrases to build better encodings. It adapts to local changes in the text distribution, which is important because many documents exhibit significant locality of reference.
>
> The truly amazing thing about the Lempel-Ziv algorithm is how robust it is on different types of data. It is quite difficult to beat Lempel-Ziv by using an application-specific algorithm. My recommendation is not to try. If you can eliminate application-specific redundancies with a simple preprocessing step, go ahead and do it. But don't waste much time fooling around. You are unlikely to get significantly better text compression than with *gzip* or some other popular program, and you might well do worse.

Implementations: Perhaps the most popular text compression program is *gzip*, which implements a public domain variation of the Lempel-Ziv algorithm. It is distributed under the GNU software license and can be obtained from *http://www.gzip.org*.

There is a natural tradeoff between compression ratio and compression time. Another choice is *bzip2*, which uses the Burrows-Wheeler transform. It produces tighter encodings than *gzip* at somewhat greater cost in running time. Going to the extreme, other compression algorithms devote enormous run times to squeeze every bit out of a file. Representative programs of this genre are collected at *http://www.cs.fit.edu/~mmahoney/compression/*.

Reasonably authoritative comparisons of compression programs are presented at *http://www.maximumcompression.com/*, including links to all available software.

Notes: A large number of books on data compression are available. Recent and comprehensive books include Sayood [Say05] and Salomon [Sal06]. Also recommended is the older book by Bell, Cleary, and Witten [BCW90]. Surveys on text compression algorithms include [CL98].

Good expositions on Huffman codes [Huf52] include [AHU83, CLRS01, Man89]. The Lempel-Ziv algorithm and variants are described in [Wel84, ZL78]. The Burrows-Wheeler transform was introduced in [BW94].

The annual IEEE Data Compression Conference (*http://www.cs.brandeis.edu/~dcc/*) is the primary research venue in this field. This is a mature technical area where most current work is shooting for fairly marginal improvements, particularly in the case of text compression. More encouragingly, we note that the conference is held annually at a world-class ski resort in Utah.

Related Problems: Shortest common superstring (see page 654), cryptography (see page 641).