August 1983 Report No. STAN-CS-83-977

Word Hy-phen-a-tion by Com-put-er

by

Franklin Mark Liang

Department of Computer Science

Stanford University
Stanford, CA 94305

WORD HY-PHEN-A-TION BY COM-PUT-ER

Franklin Mark Liang
Department of Computer Science
Stanford University
Stanford, California 94305

Abstract

This thesis describes research leading to an improved word hyphenation algorithm for the TEX82 typesetting system. Hyphenation is viewed primarily as a data compression problem, where we are given a dictionary of words with allowable division points, and try to devise methods that take advantage of the large amount of redundancy present.

The new hyphenation algorithm is based on the idea of hyphenating and inhibiting patterns. These are simply strings of letters that, when they match in a word, give us information about hyphenation at some point in the pattern. For example, '-tion' and 'c-c' are good hyphenating patterns. An important feature of this method is that a suitable set of patterns can be extracted automatically from the dictionary.

In order to represent the set of patterns in a compact form that is also reasonably efficient for searching, the author has developed a new data structure called a packed trie. This data structure allows the very fast search times characteristic of indexed tries, but in many cases it entirely eliminates the wasted space for null links usually present in such tries. We demonstrate the versatility and practical advantages of this data structure by using a variant of it as the critical component of the program that generates the patterns from the dictionary.

The resulting hyphenation algorithm uses about 4500 patterns that compile into a packed trie occupying 25K bytes of storage. These patterns find 89% of the hyphens in a pocket dictionary word list, with essentially no error. By comparison, the uncompressed dictionary occupies over 500K bytes.

This research was supported in part by the National Science Foundation under grants IST-82-01926 and MSC-83-00984, and by the System Development Foundation. 'TEX' is a trademark of the American Mathematical Society.

WORD HY-PHEN-A-TION
BY COM-PUT-ER

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

by

Franklin Mark Liang

June 1983

Acknowledgments

I am greatly indebted to my adviser, Donald Knuth, for creating the research environment that made this work possible. When I began work on the TEX project as a summer job, I would not have predicted that computer typesetting would become such an active area of computer science research. Prof. Knuth's foresight was to recognize that there were a number of fascinating problems in the field waiting to be explored, and his pioneering efforts have stimulated many others to think about these problems.

I am also grateful to the Stanford Computer Science Department for providing the facilities and the community that have formed the major part of my life for the past several years.

I thank my readers, Luis Trabb Pardo and John Gill, as well as Leo Guibas who served on my orals committee on short notice.

In addition, thanks to David Fuchs and Tom Pressburger for helpful advice and encouragement.

Finally, this thesis is dedicated to my parents, for whom the experience of pursuing a graduate degree has been perhaps even more traumatic than it was for myself.

Table of contents

Chapter 1

Introduction

The work described in this thesis was inspired by the need for a word hyphenation routine as part of Don Knuth's TEX typesetting system [1]. This system was initially designed in order to typeset Prof. Knuth's seven-volume series of books, *The Art of Computer Programming*, when he became dissatisfied with the quality of computer typesetting done by his publisher. Since Prof. Knuth's books were to be a definitive treatise on computer science, he could not bear to see his scholarly work presented in an inferior manner, when the degradation was entirely due to the fact that the material had been typeset by a computer!

Since then, TEX (also known as Tau Epsilon Chi, a system for technical text) has gained wide popularity, and it is being adopted by the American Mathematical Society, the world's largest publisher of mathematical literature, for use in its journals. TEX is distinctive among other systems for word processing/document preparation in its emphasis on the highest quality output, especially for technical material.

One necessary component of the system is a computer-based algorithm for hyphenating English words. This is part of the paragraph justification routine, and it is intended to eliminate the need for the user to specify word division points explicitly when they are necessary for good paragraph layout. Hyphenation occurs relatively infrequently in most book-format printing, but it becomes rather critical in narrow-column formats such as newspaper printing. Insufficient attention paid to this aspect of layout results in large expanses of unsightly white space, or (even worse) in words split at inappropriate points, e.g. new-spaper.

Hyphenation algorithms for existing typesetting systems are usually either rule-based or dictionary-based. Rule-based algorithms rely on a set of division rules such as given for English in the preface of Webster's Unabridged Dictionary [2]. These include recognition of common prefixes and suffixes, splitting between double consonants, and other more specialized rules. Some of the "rules" are not particularly

amenable to computer implementation; e.g. "split between the elements of a compound word". Rule-based schemes are inevitably subject to error, and they rarely cover all possible cases. In addition, the task of finding a suitable set of rules in the first place can be a difficult and lengthy project.

Dictionary-based routines simply store an entire word list along with the allowable division points. The obvious disadvantage of this method is the excessive storage required, as well as the slowing down of the justification process when the hyphenation routine needs to access a part of the dictionary on secondary store.

Examples

To demonstrate the importance of hyphenation, consider Figure 1, which shows a paragraph set in three different ways by TEX. The first example uses TEX's normal paragraph justification parameters, but with the hyphenation routine turned off. Because the line width in this example is rather narrow, TEX is unable to find an acceptable way of justifying the paragraph, resulting in the phenomenon known as an "overfull box".

One way to fix this problem is to increase the "stretchability" of the spaces between words, as shown in the second example. (TEX users: This was done by increasing the stretch component of `spaceskip` to `.5em`.) The right margin is now straight, as desired, but the overall spacing is somewhat loose.

In the third example, the hyphenation routine is turned on, and everything is beautiful.

In olden times when wishing
still helped one, there lived a king
whose daughters were all beautiful,
but the youngest was so beautiful
that the sun itself, which has seen
so much, was astonished whenever
it shone in her face. Close by
the king's castle lay a great dark
forest, and under an old lime-tree
in the forest was a well, and when
the day was very warm, the king's
child went out into the forest and
sat down by the side of the cool
fountain, and when she was bored
she took a golden ball, and threw
it up on high and caught it, and
this ball was her favorite plaything.

In olden times when wishing
still helped one, there lived a
king whose daughters were all
beautiful, but the youngest was
so beautiful that the sun itself,
which has seen so much, was
astonished whenever it shone in
her face. Close by the king's
castle lay a great dark forest,
and under an old lime-tree in
the forest was a well, and when
the day was very warm, the
king's child went out into the
forest and sat down by the side
of the cool fountain, and when
she was bored she took a golden
ball, and threw it up on high
and caught it, and this ball was
her favorite plaything.

In olden times when wish-
ing still helped one, there lived
a king whose daughters were all
beautiful, but the youngest was
so beautiful that the sun itself,
which has seen so much, was as-
tonished whenever it shone in her
face. Close by the king's castle
lay a great dark forest, and un-
der an old lime-tree in the forest
was a well, and when the day was
very warm, the king's child went
out into the forest and sat down
by the side of the cool fountain,
and when she was bored she took
a golden ball, and threw it up on
high and caught it, and this ball
was her favorite plaything.

Figure 1. A typical paragraph with and without hyphenation.

sel-fadjoint	as-so-ciate	as-so-ci-ate
Pit-tsburgh	prog-ress	pro-gress
clearin-ghouse	rec-ord	re-cord
fun-draising	a-rith-me-tic	ar-ith-met-ic
ho-meowners	eve-ning	even-ing
playw-right	pe-ri-od-ic	per-i-o-dic
algori-thm		
walkth-rough	in-de-pen-dent	in-de-pend-ent
Re-agan	tri-bune	trib-une

Figure 2. Difficult hyphenations.

However, life is not always so simple. Figure 2 shows that hyphenation can be difficult. The first column shows erroneous hyphenations made by various typesetting systems (which shall remain nameless). The next group of examples are words that hyphenate differently depending on how they are used. This happens most commonly with words that can serve as both nouns and verbs. The last two examples show that different dictionaries do not always agree on hyphenation (in this case Webster's vs. American Heritage).

TEX and hyphenation

The original TEX hyphenation algorithm was designed by Prof. Knuth and the author in the summer of 1977. It is essentially a rule-based algorithm, with three main types of rules: (1) suffix removal, (2) prefix removal, and (3) vowel-consonant-consonant-vowel (vccv) breaking. The latter rule states that when the pattern 'vowel-consonant-consonant-vowel' appears in a word, we can in most cases split between the consonants. There are also many special case rules; for example, "break vowel-q" or "break after ck". Finally a small exception dictionary (about 300 words) is used to handle particularly objectionable errors made by the above rules, and to hyphenate certain common words (e.g. pro-gram) that are not split by the rules. The complete algorithm is described in Appendix H of the old TEX manual.

In practice, the above algorithm has served quite well. Although it does not find all possible division points in a word, it very rarely makes an error. Tests on a pocket dictionary word list indicate that about 40% of the allowable hyphen points are found, with 1% error (relative to the total number of hyphen points). The algorithm requires 4K 36-bit words of code, including the exception dictionary.

The goal of the present research was to develop a better hyphenation algorithm. By "better" we mean finding more hyphens, with little or no error, and using as little additional space as possible. Recall that one way to perform hyphenation is to simply store the entire dictionary. Thus we can view our task as a data compression problem. Since there is a good deal of redundancy in English, we can hope for substantial improvement over the straightforward representation.

Another goal was to automate the design of the algorithm as much as possible. The original TEX algorithm was developed mostly by hand, with a good deal of trial and error. Extending such a rule-based scheme to find the remaining hyphens seems very difficult. Furthermore such an effort must be repeated for each new language. The former approach can be a problem even for English, because pronunciation (and thus hyphenation) tends to change over time, and because different types of publication may call for different sets of admissible hyphens.

Time magazine algorithm

A number of approaches were considered, including methods that have been discussed in the literature or implemented in existing typesetting systems. One of the methods studied was the so-called Time magazine algorithm, which is table-based rather than rule-based.

The idea is to look at four letters surrounding each possible breakpoint, namely two letters preceding and two letters following the given point. However we do not want to store a table of $26^4 = 456{,}976$ entries representing all possible four-letter combinations. (In practice only about 15% of these four-letter combinations actually occur in English words, but it is not immediately obvious how to take advantage of this.)

Instead, the method uses three tables of size 26^2, corresponding to the two letters preceding, surrounding, and following a potential hyphen point. That is, if the letter pattern wx-yz occurs in a word, we look up three values corresponding to the letter pairs wx, xy, and yz, and use these values to determine if we can split the pattern.

What should the three tables contain? In the Time algorithm the table values were the probabilities that a hyphen could occur after, between, or before two given letters, respectively. The probability that the pattern wx-yz can be split is then estimated as the product of these three values (as if the probabilities were independent, which they aren't). Finally the estimated value is compared against a threshold to determine hyphenation. Figure 3 shows an example of hyphenation probabilities computed by this method.

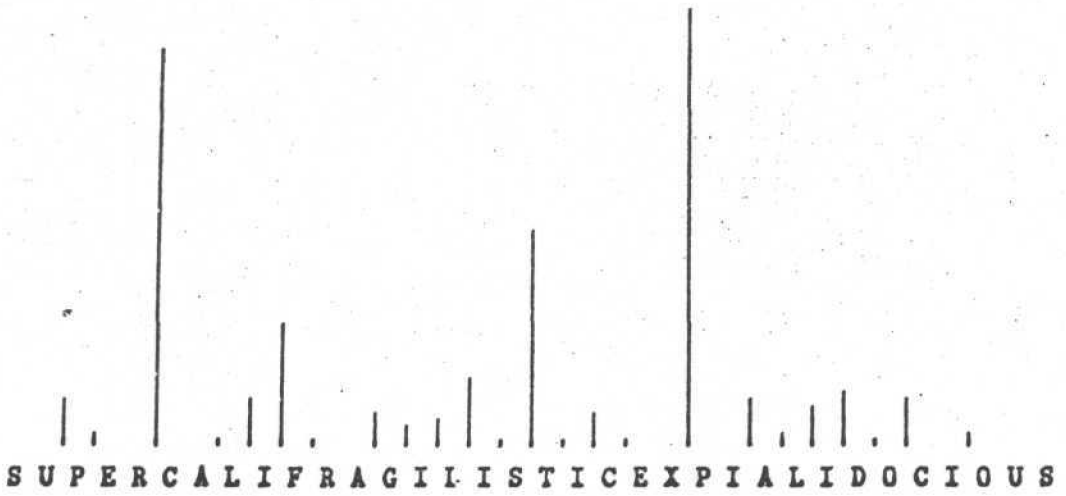

Figure 3. Hyphenation probabilities.

The advantage of this table-based approach is that the tables can be generated automatically from the dictionary. However, some experiments with the method yielded discouraging results. One estimate is 40% of the hyphens found, with 8% error. Thus a large exception dictionary would be required for good performance.

The reason for the limited performance of the above scheme is that just four letters of context surrounding the potential break point are not enough in many cases. In an extreme example, we might have to look as many as 10 letters ahead in order to determine hyphenation, e.g. dem-on-stra-tion vs. de-mon-stra-tive.

So a more powerful method is needed.

Patterns

A good deal of experimentation led the author to a more powerful method based on the idea of hyphenation *patterns*. These are simply strings of letters that, when they match in a word, will tell us how to hyphenate at some point in the pattern. For example, the pattern 'tion' might tell us that we can hyphenate before the 't'. Or when the pattern 'cc' appears in a word, we can usually hyphenate between the c's. Here are some more examples of good hyphenating patterns:

```
.in-d .in-s .in-t .un-d b-s -cia con-s con-t e-ly er-l er-m
ex- -ful it-t i-ty -less l-ly -ment n-co -ness n-f n-l n-si
n-v om-m -sion s-ly s-nes ti-ca x-p
```

(The character '.' matches the beginning or end of a word.)

Patterns have many advantages. They are a general form of "hyphenation rule" that can include prefix, suffix, and other rules as special cases. Patterns can even describe an exception dictionary, namely by using entire words as patterns. (Actually, patterns are often more concise than an exception dictionary because a single pattern can handle several variant forms of a word; e.g. `pro-gram`, `pro-grams`, and `pro-grammed`.)

More importantly, the pattern matching approach has proven very effective. An appropriate set of patterns captures very concisely the information needed to perform hyphenation. Yet the pattern rules are of simple enough form that they can be generated automatically from the dictionary.

When looking for good hyphenating patterns, we soon discover that almost all of them have some exceptions. Although `-tion` is a very "safe" pattern, it fails on the word `cat-ion`. Most other cases are less clear-cut; for example, the common pattern `n-t` can be hyphenated about 80 percent of the time. It definitely seems worthwhile to use such patterns, provided that we can deal with the exceptions in some manner.

After choosing a set of hyphenating patterns, we may end up with thousands of exceptions. These could be listed in an exception dictionary, but we soon notice there are many similarities among the exceptions. For example, in the original TEX algorithm we found that the vowel-consonant-consonant-vowel rule resulted in hundreds of errors of the form `X-Yer` or `X-Yers`, for certain consonant pairs `XY`, so we put in a new rule to prevent those errors.

Thus, there may be "rules" that can handle large classes of exceptions. To take advantage of this, patterns come to the rescue again; but this time they are *inhibiting* patterns, because they show where hyphens should *not* be placed. Some good examples of inhibiting patterns are: `b=ly` (don't break between b and ly), `bs=`, `=cing`, `io=n`, `i=tin`, `=ls`, `nn=`, `ns=t`, `n=ted`, `=pt`, `ti=al`, `=tly`, `=ts`, and `tt=`.

As it turns out, this approach is worth pursuing further. That is, after applying hyphenating and inhibiting patterns as discussed above, we might have *another* set of hyphenating patterns, then another set of inhibiting patterns, and so on. We can think of each level of patterns as being "exceptions to the exceptions" of the previous level. The current TEX82 algorithm uses five alternating levels of hyphenating and inhibiting patterns. The reasons for this will be explained in Chapter 4.

The idea of patterns is the basis of the new TEX hyphenation algorithm, and it was the inspiration for much of the intermediate investigation, that will be described.

Overview of thesis

In developing the pattern scheme, two main questions arose: (1) How can we represent the set of hyphenation patterns in a compact form that is also reasonably efficient for searching? (2) Given a hyphenated word list, how can we generate a suitable set of patterns?

To solve these problems, the author has developed a new data structure called a *packed trie*. This data structure allows the very fast search times characteristic of indexed tries, but in many cases it entirely eliminates the wasted space for null links usually present in such tries.

We will demonstrate the versatility and practical advantages of this data structure by using it not only to represent the hyphenation patterns in the final algorithm, but also as the critical component of the program that generates the patterns from the dictionary. Packed tries have many other potential applications, including identifier lookup, spelling checking, and lexicographic sorting.

Chapter 2 considers the simpler problem of recognizing, rather than hyphenating, a set of words such as a dictionary, and uses this problem to motivate and explain the advantages of the packed trie data structure. We also point out the close relationship between tries and finite-state machines.

Chapter 3 discusses ways of applying these ideas to hyphenation. After considering various approaches, including minimization with don't cares, we return to the idea of patterns.

Chapter 4 discusses the heuristic method used to select patterns, introduces dynamic packed tries, and describes some experiments with the pattern generation program.

Chapter 5 gives a brief history, and mentions ideas for future research.

Finally, the appendix contains the WEB [3] listing of the portable pattern generation program PATGEN, as well as the set of patterns currently used by TEX82.

Note: The present chapter has been typeset by giving unusual instructions to TEX so that it hyphenates words much more often than usual; therefore the reader can see numerous examples of word breaks that were discovered by the new algorithm.

Chapter 2

The dictionary problem

In this chapter we consider the problem of recognizing a set of words over an alphabet. To be more precise, an *alphabet* is a set of characters or symbols, for example the letters A through Z, or the ASCII character set. A *word* is a sequence of characters from the alphabet. Given a set of words, our problem is to design a data structure that will allow us to determine efficiently whether or not some word is in the set.

In particular, we will use spelling checking as an example throughout this chapter. This is a topic of interest in its own right, but we discuss it here because the pattern matching techniques we propose will turn out to be very useful in our hyphenation algorithm.

Our problem is a special case of the general set recognition problem, because the elements of our set have the additional structure of being variable-length sequences of symbols from a finite alphabet. This naturally suggests methods based on a character-by-character examination of the key, rather than methods that operate on the entire key at once. Also, the redundancy present in natural languages such as English suggests additional opportunities for compression of the set representation.

We will be especially interested in space minimization. Most data structures for set representation, including the one we propose, are reasonably fast for searching. That is, a search for a key doesn't take much more time than is needed to examine the key itself. However, most of these algorithms assume that everything is "in core", that is, in the primary memory of the computer. In many situations, such as our spelling checking example, this is not feasible. Since secondary memory access times are typically much longer, it is worthwhile to try compressing the data structure as much as possible.

In addition to determining whether a given word is in the set, there are other operations we might wish to perform on the set representation. The most basic are insertion and deletion of words from the set. More complicated operations include performing the union of two sets, partitioning a set according to some criterion,

determining which of several sets an element is a member of, or operations based on an ordering or other auxiliary information associated with the keys in the set. For the data structures we consider, we will pay some attention to methods for insertion and deletion, but we shall not discuss the more complicated operations.

We first survey some known methods for set representation, and then propose a new data structure called a "packed trie".

Data structures

Methods for set representation include the following: sequential lists, sorted lists, binary search trees, balanced trees, hashing, superimposed coding, bit vectors, and digital search trees (also known as tries). Good discussions of these data structures can be found in a number of texts, including Knuth [4], Standish [5], and AHU [6]. Below we make a few remarks about each of these representations.

A sequential list is the most straightforward representation. It requires both space and search time proportional to the number of characters in the dictionary.

A sorted list assumes an ordering on the keys, such as alphabetical order. Binary search allows the search time to be reduced to the logarithm of the size of the dictionary, but space is not reduced.

A binary search tree also allows search in logarithmic time. This can be thought of as a more flexible version of a sorted list that can be optimized in various ways. For example if the probabilities of searching for different keys in the tree are known, then the tree can be adapted to improve the expected search time. Search trees can also handle insertions and deletions easily, although an unfavorable sequence of such operations may degrade the performance of the tree.

Balanced tree schemes (including AVL trees, 2-3 trees, and B-trees) correct the above-mentioned problem, so that insertions, deletions, and searches can all be performed in logarithmic time in the worst case. Variants of trees have other nice properties, too; they allow merging and splitting of sets, and priority queue operations. B-trees are well-suited to large applications, because they are designed to minimize the number of secondary memory accesses required to perform a search. However, space utilization is not improved by any of these tree schemes, and in fact it is usually increased because of the need for extra pointers.

Hashing is an essentially different approach to the problem. Here a suitable randomizing function is used to compute the location at which a key is stored. Hashing methods are very fast on the average, although the worst case is linear; fortunately this worst case almost never happens.

An interesting variant of hashing, called superimposed coding, was proposed by Bloom [7] (see also [4, §6.5], [8]), and at last provides for reduction in space,

although at the expense of allowing some error. Since this method is perhaps less well known we give a description of it here.

Superimposed coding

The idea is as follows. We use a single large bit array, initialized to zeros, plus a suitable set of d different hash functions. To represent a word, we use the hash functions to compute d bit positions in the large array of bits, and set these bits to ones. We do this for each word in the set. Note that some bits may be set by more than one word.

To test if a word is in the set, we compute the d bit positions associated with the word as above, and check to see if they are all ones in the array. If any of them are zero, the word cannot be in the set, so we reject it. Otherwise if all of the bits are ones, we accept the word. However, some words not in the set might be erroneously accepted, if they happen to hash into bits that are all "covered" by words in the set.

It can be shown [7] that the above scheme makes the best use of space when the density of bits in the array, after all the words have been inserted, is approximately one-half. In this case the probability that a word not in the set is erroneously accepted is 2^{-d}. For example if each word is hashed into 4 bit positions, the error probability is 1/16. The required size of the bit array is approximately $nd \lg e$, where n is the number of items in the set, and $\lg e \approx 1.44$.

In fact Bloom specifically discusses automatic hyphenation as an application for his scheme! The scenario is as follows. Suppose we have a relatively compact routine for hyphenation that works correctly for 90 percent of the words in a large dictionary, but it is in error or fails to hyphenate the other 10 percent. We would then like some way to test if a word belongs to the 10 percent, but we do not have room to store all of these words in main memory. If we instead use the superimposed coding scheme to test for these words, the space required can be much reduced. For example with $d = 4$ we only need about 6 bits per word. The penalty is that some words will be erroneously identified as being in the 10 percent. However, this is acceptable because usually the test word will be rejected and we can then be sure that it is not one of the exceptions. (Either it is in the other 90 percent or it is not in the dictionary at all.) In the comparatively rare case that the word is accepted, we can go to secondary store, to check explicitly if the word is one of the exceptions.

The above technique is actually used in some commercial hyphenation routines. For now, however, TEX will not have an external dictionary. Instead we will require that our hyphenation routine be essentially free of error (although it may not achieve complete hyphenation).

An extreme case of superimposed coding should also be mentioned, namely the bit-vector representation of a set. (Imagine that each word is associated with a single bit position, and one bit is allocated for each possible word.) This representation is often very convenient, because it allows set intersection and union to be performed by simple logical operations. But it also requires space proportional to the size of the universe of the set, which is impractical for words longer than three or four characters.

Tries

The final class of data structures we will consider are the digital search trees, first described by de la Briandais [9] and Fredkin [10]. Fredkin also introduced the term "trie" for this class of trees. (The term was derived from the word re*trie*val, although it is now pronounced "try".)

Tries are distinct from the other data structures discussed so far because they explicitly assume that the keys are a *sequence* of values over some (finite) alphabet, rather than a single indivisible entity. Thus tries are particularly well-suited for handling variable-length keys. Also, when appropriately implemented, tries can provide compression of the set represented, because common prefixes of words are combined together; words with the same prefix follow the same search path in the trie.

A trie can be thought of as an m-ary tree, where m is the number of characters in the alphabet. A search is performed by examining the key one character at a time and using an m-way branch to follow the appropriate path in the trie, starting at the root.

We will use the set of 31 most common English words, shown below, to illustrate different ways of implementing a trie.

A	FOR	IN	THE
AND	FROM	IS	THIS
ARE	HAD	IT	TO
AS	HAVE	NOT	WAS
AT	HE	OF	WHICH
BE	HER	ON	WITH
BUT	HIS	OR	YOU
BY	I	THAT	

Figure 4. The 31 most common English words.

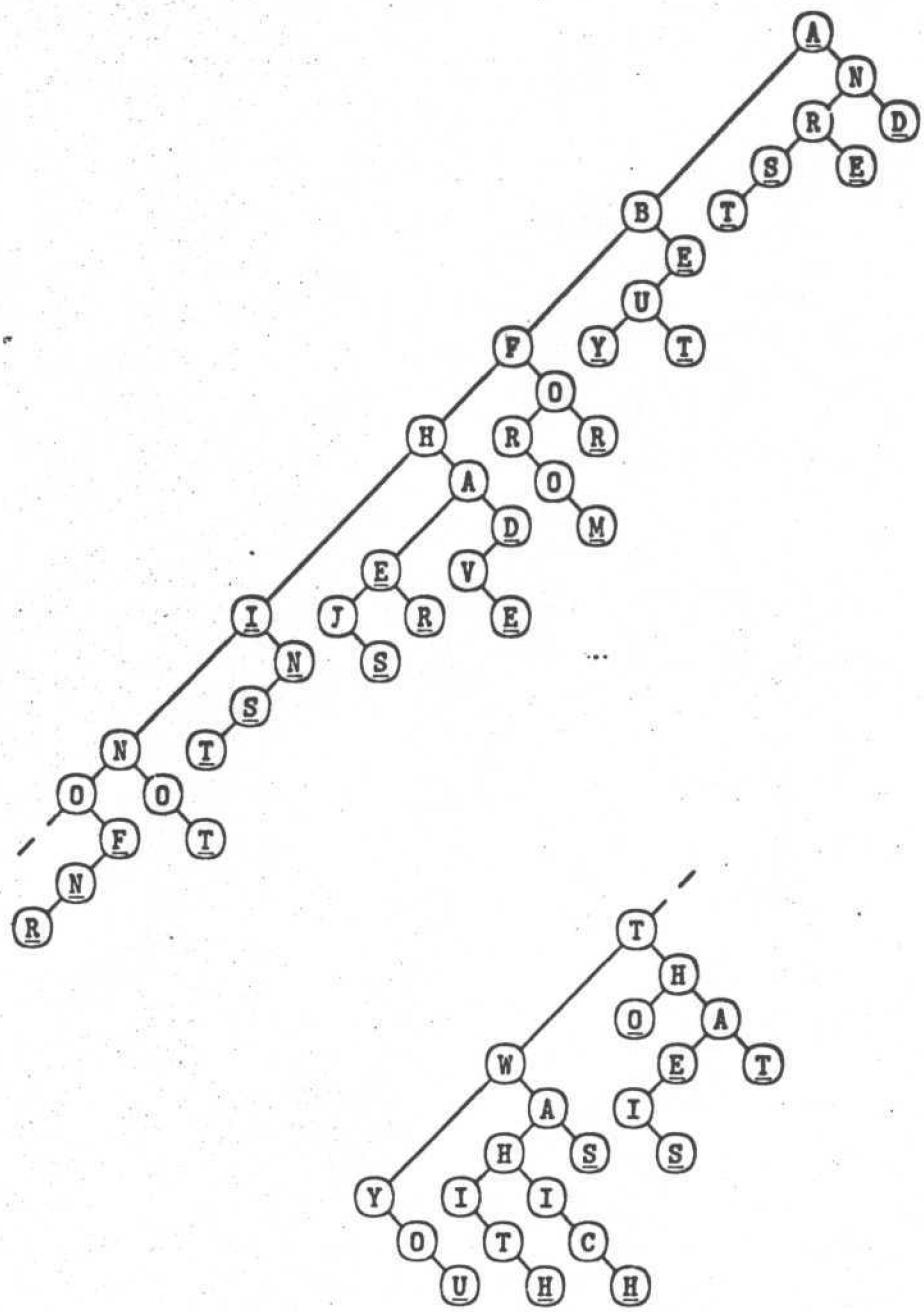

Figure 5. Linked trie for the 31 most common English words.

Figure 5 shows a *linked trie* representing this set of words. In a linked trie, the m-way branch is performed using a sequential series of comparisons. Thus in Figure 5 each node represents a yes-no test against a particular character. There are two link fields indicating the next node to take depending on the outcome of the test. On a 'yes' answer, we also move to the next character of the key. The underlined characters are terminal nodes, indicated by an extra bit in the node. If the word ends when we are at a terminal node, then the word is in the set.

Note that we do not have to actually store the keys in the trie, because each node implicitly represents a prefix of a word, namely the sequence of characters leading to that node.

A linked trie is somewhat slow because of the sequential testing required for each character of the key. The number of comparisons per character can be as large as m, the size of the alphabet. In addition, the two link fields per node are somewhat wasteful of space. (Under certain circumstances, it is possible to eliminate one of these two links. We will explain this later.)

In an *indexed trie*, the m-way branch is performed using an array of size m. The elements of the array are pointers indicating the next family of the trie to go to when the given character is scanned, where a "family" corresponds to the group of nodes in a linked trie for testing a particular character of the key. When performing a search in an indexed trie, the appropriate pointer can be accessed by simply indexing from the base of the array. Thus search will be quite fast.

But indexed tries typically waste a lot of space, because most of the arrays have only a few "valid" pointers (for words in the trie), with the rest of the links being null. This is especially common near the bottom of the trie. Figure 6 shows an indexed trie for the set of 31 common words. This representation requires $26 \times 32 = 832$ array locations, compared to 59 nodes for the linked trie.

Various methods have been proposed to remedy the disadvantages of linked and indexed tries. Trabb Pardo [11] describes and analyzes the space requirements of some simple variants of binary tries. Knuth [4, ex. 6.3–20] analyzes a composite method where an indexed trie is used for the first few levels of the trie, switching to sequential search when only a few keys remain in a subtrie. Mehlhorn [12] suggests using a binary search tree to represent each family of a trie. This requires storage proportional to the number of "valid" links, as in a linked trie, but allows each character of the key to be processed in at most $\log m$ comparisons. Maly [13] has proposed a "compressed trie" that uses an implicit representation to eliminate links entirely. Each level of the trie is represented by a bit array, where the bits indicate whether or not some word in the set passes through the node corresponding to

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	2	5				7		11	16					17	19					20			24		31	
2														3				4	0	0						
3				0																						
4					0																					
5					0																6				0	
6																				0						
7															8			9								
8																		0								
9															10											
10													0													
11	12				14				15																	
12				0																		13				
13					0																					
14																		0								
15																			0							
16														0					0	0						
17															18											
18																				0						
19						0								0				0								
20								21							0											
21	22				0				23																	
22																				0						
23																			0							
24	25							26	29																	
25																			0							
26									27																	
27			28																							
28								0																		
29																				30						
30								0																		
31															32											
32																					0					

Figure 6. Indexed trie for the 31 most common English words.

that bit. In addition each family contains a field indicating the number of nonzero bits in the array for all nodes to the left of the current family, so that we can find the desired family on the next level. The storage required for each family is thus reduced to $m+\log n$ bits, where n is the total number of keys. However, compressed tries cannot handle insertions and deletions easily, nor do they retain the speed of indexed tries.

Packed tries

Our idea is to use an indexed trie, but to save the space for null links by packing the different families of the trie into a single large array, so that links from one family may occupy space normally reserved for links for other families that happen to be null. An example of this is illustrated below.

(In the following, we will sometimes refer to families of the indexed trie as *states*, and pointers as *transitions*. This is by analogy with the terminology for finite-state machines.)

When performing a search in the trie, we need a way to check if an indexed pointer actually corresponds to the current family, or if it belongs to some other family that just happens to be packed in the same location. This is done by additionally storing the character indexing a transition along with that transition. Thus a transition belongs to a state only if its character matches the character we are indexing on. This test always works if one additional requirement is satisfied, namely that different states may not be packed at the same base location.

The trie can be packed using a first-fit method. That is, we pack the states one at a time, putting each state into the lowest-indexed location in which it will fit (not overlapping any previously packed transitions, nor at an already occupied base location). On numerous examples based on typical word lists, this heuristic works extremely well. In fact, nearly all of the holes in the trie are often filled by transitions from other states.

Figure 7 shows the result when the indexed trie of Figure 6 is packed into a single array using the first-fit method. (Actually we have used an additional compression technique called suffix compression before packing the trie; this will be explained in the next section.) The resulting trie fits into just 60 locations. Note

	0	1	2	3	4	5	6	7	8	9
00		A 8	B11			D 0	F 3	E 0	H30	I23
10	C 5			H 0	N25	O32	E 0		O12	M 0
20	T33	R14	N 1	W46	T 0	Y37	R 2	S 0	T 0	0 6
30	R 0	A29	U 4	D 0	S 0	E12	Y 0	N 0	F 0	I15
40	0 4	H44	S 0	T 0	I 7	A 4	N 0	A15	0 0	E 0
50	R 0	V 2	O38	I15	H35	I36	T 5			U 0

Figure 7. Packed trie for the 31 most common English words.

that the packed trie is a single large array; the rows in the figure should be viewed as one long row.

As an example, here's what happens when we search for the word HAVE in the packed trie. We associate the values 1 through 26 with the letters A through Z. The root of the trie is packed at location 0, so we begin by looking at location 8 corresponding to the letter H. Since 'H30' is stored there, this is a valid transition and we then go to location 30. Indexing by the letter A, we look in location 31, which tells us to go to 29. Now indexing by V gets location 51, which points to 2. Finally indexing by E gets location 7, which is underlined, indicating that the word HAVE is indeed in the set.

Suffix compression

A big advantage of the trie data structure is that common prefixes of words are combined automatically into common paths in the trie. This provides a good deal of compression. To save more space, we can try to take advantage of common suffixes.

One way of doing this is to construct a trie in the usual manner, and then merge common subtries together, starting from the leaves (lieves) and working upward. We call this process *suffix compression.*

For example, in the linked trie of Figure 5 the terminal nodes for the words HIS and THIS, both of which test for the letter S and have no successors, can be combined into a single node. That is, we can let their parent nodes both point to the same node; this does not change the set of words accepted by the trie. It turns out that we can then combine the parent nodes, since both of them test for I and go to the S node if successful, otherwise stop (no left successor). However, the grandparent nodes (which are actually siblings of the I nodes) cannot be combined even though they both test for E, because one of them goes to a terminal R node upon success, while the other has no right successor.

With a larger set of words, a great deal of merging can be possible. Clearly all leaf nodes (nodes with no successors) that test the same character can be combined together. This alone saves a number of nodes equal to the number of words in the dictionary, minus the number of words that are prefixes of other words, plus at most 26. In addition, as we might expect, longer suffixes such as -ly, -ing, or -tion can frequently be combined.

The suffix compression process may sound complicated, but actually it can be described by a simple recursive algorithm. For each node of the trie, we first compress each of its subtries, then determine if the node can be merged with some other node. In effect, we traverse the trie in depth-first order, checking each node to see if it is equivalent to any previously seen node. A hash table can be used to identify equivalent nodes, based on their (merged) transitions.

The identification of nodes is somewhat easier using a binary tree representation of the trie, rather than an m-ary representation, because each node will then have just two link fields in addition to the character and output bit. Thus it will be convenient to use a linked trie when performing suffix compression. The linked representation is also more convenient for constructing the trie in the first place, because of the ease of performing insertions.

After applying suffix compression, the trie can be converted to an indexed trie and packed as described previously. (We should remark that performing suffix compression on a linked trie can yield some additional compression, because trie families can be partially merged. However such compression is lost when the trie is converted to indexed form.)

The author has performed numerous experiments with the above ideas. The results for some representative word lists are shown in Table 1 below. The last three

columns show the number of nodes in the linked, suffix-compressed, and packed tries, respectively. Each transition of the packed trie consists of a pointer, a character, and a bit indicating if this is an accepting transition.

word list	words	characters	linked	compressed	packed
pascal	35	145	125	104	120
murray	2720	19,144	8039	4272	4285
pocket	31,036	247,612	92,339	38,619	38,638
unabrd	235,545	2,256,805	759,045	—	—

Table 1. Suffix-compressed packed tries.

The algorithms for building a linked trie, suffix compression, and first-fit packing are used in TEX82 to preprocess the set of hyphenation patterns into a packed trie used by the hyphenation routine. A WEB description of these algorithms can be found in [14].

Derived forms

Most dictionaries do not list the most common derived forms of words, namely regular plurals of nouns and verbs (`-s` forms), participles and gerunds of verbs (`-ed` and `-ing` forms), and comparatives and superlatives of adjectives (`-er` and `-est`). This makes sense, because a user of the dictionary can easily determine when a word possesses one of these regular forms. However, if we use the word list from a typical dictionary for spelling checking, we will be faced with the problem of determining when a word is one of these derived forms.

Some spelling checkers deal with this problem by attempting to recognize affixes. This is done not only for the derived forms mentioned above but other common variant forms as well, with the purpose of reducing the number of words that have to be stored in the dictionary. A set of logical rules is used to determine when certain prefixes and suffixes can be stripped from the word under consideration.

However such rules can be quite complicated, and they inevitably make errors. The situation is not unlike that of finding rules for hyphenation, which should not be surprising, since affix recognition is an important part of any rule-based hyphenation algorithm. This problem has been studied in some detail in a series of papers by Resnikoff and Dolby [15].

Since affix recognition is difficult, it is preferable to base a spelling checker on a complete word list, including all derived forms. However, a lot of additional space will be required to store all of these forms, even though much of the added data is

redundant. We might hope that some appropriate method could provide substantial compression of the expanded word list. It turns out that suffix-compressed tries handle this quite well. When derived forms were added to our pocket dictionary word list, it increased in size to 49,858 words and 404,946 characters, but the resulting packed trie only increased to 46,553 transitions (compare the pocket dictionary statistics in Table 1).

Hyphenation programs also need to deal with the problem of derived forms. In our pattern-matching approach, we intend to extract the hyphenation rules automatically from the dictionary. Thus it is again preferable for our word list to include all derived forms.

The creation of such an expanded word list required a good deal of work. The author had access to a computer-readable copy of Webster's Pocket Dictionary [16], including parts of speech and definitions. This made it feasible to identify nouns, verbs, etc., and to generate the appropriate derived forms mechanically. Unfortunately the resulting word lists required extensive editing to eliminate many never-used or somewhat nonsensical derived forms, e.g. '`informations`'.

Spelling checkers

Computer-based word processing systems have recently come into widespread use. As a result there has been a surge of interest in programs for automatic spelling checking and correction. Here we will consider the dictionary representations used by some existing spelling checkers.

One of the earliest programs, designed for a large timesharing computer, was the DEC-10 SPELL program written by Ralph Gorin [17]. It uses a 12,000 word dictionary stored in main memory. A simple hash function assigns a unique 'bucket' to each word depending on its length and the first two characters. Words in the same bucket are listed sequentially. The number of words in each bucket is relatively small (typically 5 to 50 words), so this representation is fairly efficient for searching. In addition, the buckets provide convenient access to groups of similar words; this is useful when the program tries to correct spelling errors.

The dictionary used by SPELL does not contain derived forms. Instead some simple affix stripping rules are normally used; the author of the program notes that these are "error-prone".

Another spelling checker is described by James L. Peterson [18]. His program uses three separate dictionaries: (1) a small list of 258 common English words, (2) a dynamic 'cache' of about 1000 document-specific words, and (3) a large, comprehensive dictionary, stored on disk. The list of common words (which is static) is represented using a suffix-compressed linked trie. The dynamic cache is maintained

using a hash table. Both of these dictionaries are kept in main memory for speed. The disk dictionary uses an in-core index, so that at most one disk access is required per search.

Robert Nix [19] describes a spelling checker based on the superimposed coding method. He reports that this method allows the dictionary from the SPELL program to be compressed to just 20 percent of its original size, while allowing 0.1% chance of error.

A considerably different approach to spelling checking was taken by the TYPO program developed at Bell Labs [20]. This program uses digram and trigram frequencies to identify "improbable" words. After processing a document, the words are listed in order of decreasing improbability for the user to peruse. (Words appearing in a list of 2726 common technical words are not shown.) The authors report that this format is "psychologically rewarding", because many errors are found at the beginning, inducing the user to continue scanning the list until errors become rare.

In addition to the above, there have recently been a number of spelling checkers developed for the "personal computer" market. Because these programs run on small microprocessor-based systems, it is especially important to reduce the size of the dictionary. Standard techniques include hash coding (allowing some error), in-core caches of common words, and special codes for common prefixes and suffixes. One program first constructs a sorted list of all words in the document, and then compares this list with the dictionary in a single sequential pass. The dictionary can then be stored in a compact form suited for sequential scanning, where each word is represented by its difference from the previous word.

Besides simply detecting when words are not in a dictionary, the design of a practical spelling checker involves a number of other issues. For example many spelling checkers also try to perform spelling *correction*. This is usually done by searching the dictionary for words similar to the misspelled word. Errors and suggested replacements can be presented in an interactive fashion, allowing the user to see the context from the document and make the necessary changes. The contents of the dictionary are of course very important, and each user may want to modify the word list to match his or her own vocabulary. Finally, a plain spelling checker cannot detect problems such as incorrect word usage or mistakes in grammar; a more sophisticated program performing syntactic and perhaps semantic analysis of the text would be necessary.

Conclusion and related ideas

The dictionary problem is a fundamental problem of computer science, and it has many applications besides spelling checking. Most data structures for this problem consider the elements of the set as atomic entities, fitting into a single computer word. However in many applications, particularly word processing, the keys are actually variable-length strings of characters. Most of the standard techniques are somewhat awkward when dealing with variable length keys. Only the trie data structure is well-suited for this situation.

We have proposed a variant of tries that we call a packed trie. Search in a packed trie is performed by indexing, and it is therefore very fast. The first-fit packing technique usually produces a fairly compact representation as well.

We have not discussed how to perform dynamic insertions and deletions with a packed trie. In Chapter 4 we discuss a way to handle this problem, when no suffix compression is used, by repacking states when necessary.

The idea of suffix compression is not new. As mentioned, Peterson's spelling checker uses this idea also. But in fact, if we view our trie as a finite-state machine, suffix compression is equivalent to the well-known idea of state minimization. In our case the machine is acyclic, that is, it has no loops.

Suffix compression is also closely related to the common subexpression problem from compiler theory. In particular, it can be considered a special case of a problem called acyclic congruence closure, which has been studied by Downey, Sethi, and Tarjan [21]. They give a linear-time algorithm for suffix compression that does not use hashing, but it is somewhat complicated to implement and requires additional data structures.

The idea for the first-fit packing method was inspired by the paper "Storing a sparse table" by Tarjan and Yao [22]. The technique has been used for compressing parsing tables, as discussed by Zeigler [23] (see also [24]). However, our packed trie implementation differs somewhat from the applications discussed in the above references, because of our emphasis on space minimization. In particular, the idea of storing the character that indexes a transition, along with that transition, seems to be new. This has an advantage over other techniques for distinguishing states, such as the use of back pointers, because the character requires fewer bits.

The paper by Tarjan and Yao also contains an interesting theorem characterizing the performance of the first-fit packing method. They consider a modification suggested by Zeigler, where the states are first sorted into decreasing order based on the number of non-null transitions in each state. The idea is that small states, which can be packed more easily, will be saved to the end. They prove that if the

distribution of transitions among states satisfies a "harmonic decay" condition, then essentially all of the holes in the first-fit packing will be filled.

More precisely, let $n(l)$ be the total number of non-null transitions in states with more than l transitions, for $l \geq 0$. If the harmonic decay property $n(l) \leq n/(l+1)$ is satisfied, then the first-fit-decreasing packing satisfies $0 \leq b(i) \leq n$ for all i, where $n = n(0)$ is the total number of transitions and $b(i)$ is the base location at which the ith state is packed.

The above theorem does not take into account our additional restriction that no two states may be packed at the same base location. When the proof is modified to include this restriction, the bound goes up by a factor of two. However in practice we seem to be able to do much better.

The main reason for the good performance of the first-fit packing scheme is the fact that there are usually enough single-transition states to fill in the holes created by larger states. It is not really necessary to sort the states by number of transitions; any packing order that distributes large and small states fairly evenly will work well. We have found it convenient simply to use the order obtained by traversing the linked trie.

Improvements on the algorithms discussed in this chapter are possible in certain cases. If we store a linked trie in a specific traversal order, we can eliminate one of the link fields. For example, if we list the nodes of the trie in preorder, the left successor of a node will always appear immediately after that node. An extra bit is used to indicate that a node has no left successor. Of course this technique works for other types of trees as well.

If the word list is already sorted, linked trie insertion can be performed with only a small portion of the trie in memory at any time, namely the portion along the current insertion path. This can be a great advantage if we are are processing a large dictionary and cannot store the entire linked trie in memory.

Hyphenation

Let us now try to apply the ideas of the previous chapter to the problem of hyphenation. TEX82 will use the pattern matching method described in Chapter 1, but we shall first discuss some related approaches that were considered.

Finite-state machines with output

We can modify our trie-based dictionary representation to perform hyphenation by changing the output of the trie (or finite-state machine) to a multiple-valued output indicating how the word can be hyphenated, instead of just a binary yes-no output indicating whether or not the word is in the dictionary. That is, instead of associating a single bit with each trie transition, we would have a larger "output" field indicating the hyphenation "action" to be taken on this transition. Thus on recognizing the word hy-phen-a-tion, the output would say "you can hyphenate this word after the second, sixth, or seventh letters".

To represent the hyphenation output, we could simply list the hyphen positions, or we could use a bit vector indicating the allowable hyphen points. Since there are only a few hundred different outputs and most of them occur many times, we can save some space by assigning each output a unique code and storing the actual hyphen positions in a separate table.

To conveniently handle the variable number of hyphen positions in outputs, we will use a linked representation that allows different outputs to share common portions of their output lists. This is implemented using a hash table containing pairs of the form (*output*, *next*), where *output* is a hyphenation position and *next* is a (possibly null) pointer to another entry in the table. To add a new output list to the table, we hash each of its outputs in turn, making each output point to the previous one. Interestingly, this process is quite similar to suffix compression.

The trie with hyphenation output can be suffix-compressed and packed in the same manner as discussed in Chapter 2. Because of the greater variety of outputs more of the subtries will be distinct, and there is somewhat less compression.

From our pocket dictionary (with hyphens), for example, we obtained a packed trie occupying 51,699 locations.

We can improve things slightly by "pushing outputs forward". That is, we can output partial hyphenations as soon as possible instead of waiting until the end of the word. This allows some additional suffix compression.

For example, upon scanning the letters hyph at the beginning of a word, we can already say "hyphenate after the second letter" because this is allowed for all words beginning with those letters. Note we could not say this after scanning just hyp, because of words like hyp-not-ic. Upon further scanning ena, we can say "hyphenate after the sixth letter".

When implementing this idea, we run into a small problem. There are quite a few words that are prefixes of other words, but hyphenate differently on the letters they have in common, e.g. ca-ret and care-tak-er, or as-pi-rin and as-pir-ing. To avoid losing hyphenation output, we could have a separate output whenever an end-of-word bit appears, but a simpler method is to append an end-of-word character to each word before inserting it into the trie. This increases the size of the linked trie considerably, but suffix compression merges most of these nodes together.

With the above modifications, the packed trie for the pocket dictionary was reduced to 44,128 transitions.

Although we have obtained substantial compression of the dictionary, the result is still too large for our purposes. The problem is that as long as we insist that only words in the dictionary be hyphenated, we cannot hope to reduce the space required to below that needed for spelling checking alone. So we must give up this restriction.

For example, we could eliminate the end-of-word bit. Then after pushing outputs forward, we can prune branches of the trie for which there is no further output. This would reduce the pocket dictionary trie to 35,429 transitions.

Minimization with don't cares

In this section we describe a more drastic approach to compression that takes advantage of situations where we "don't care" what the algorithm does.

As previously noted, most of the states in an indexed trie are quite sparse; that is, only a few of the characters have explicit transitions. Since the missing transitions are never accessed by words in our dictionary, we can allow them to be filled by arbitrary transitions.

This should not be confused with the overlapping of states that may occur in the trie-packing process. Instead, we mean that the added transitions will actually become part of the state.

There are two ways in which this might allow us to save more space in the minimization process. First, states no longer have to be identical in order to be merged; they only have to agree on those characters where both (or all) have explicit transitions. Second, the merging of non-equivalent states may allow further merging that was not previously possible, because some transitions have now become equivalent.

For example, consider again the trie of Figure 5. When discussing suffix compression, we noted that the terminal S nodes for the words HIS and THIS could be merged together, but that the parent chains, each containing transitions for A, E, and I, could not be completely merged. However, in minimization with don't cares these two states can be merged. Note that such a merge will require that the DV state below the first A be merged with the T below the second A; this can be done because those states have no overlapping transitions.

As another example, notice that if the word AN were added to our vocabulary, then the NRST chain succeeding the root A node could be merged with the NST chain below the initial I node. (Actually, it doesn't make much sense to do minimization with don't cares on a trie used to recognize words in a dictionary, but we will ignore that objection for the purposes of this example.)

Unfortunately, trie minimization with don't cares seems more complicated than the suffix-compression process of Chapter 2. The problem is that states can be merged in more than one way. That is, the collection of mergeable states no longer forms an equivalence relation, as in regular finite-state minimization. In fact, we can sometimes obtain additional compression by allowing the same state to appear more than once. Another complication is that don't care merges can introduce loops into our trie.

Thus it seems that finding the minimum size trie will be difficult. Pfleeger [25] has shown this problem to be NP-complete, by transformation from graph coloring; however, his construction requires the number of transitions per state to be unbounded. It may be possible to remove this requirement, but we have not proved this.

So in order to experiment with trie minimization with don't cares, we have made some simplifications. We start by performing suffix compression in the usual manner. We then go through the states in a bottom-up order, checking each to see if it can be merged with any previous state by taking advantage of don't cares. Note that such merges may require further merges among states already seen.

We only try merges that actually save space, that is, where explicit transitions are merged. Otherwise, states with only a few transitions are very likely to be mergeable, but such merges may constrain us unnecessarily at a later stage of the minimization. In addition, we will not consider having multiple copies of states.

Even this simplified algorithm can be quite time consuming, so we did not try it on our pocket dictionary. On a list of 2726 technical words, don't care minimization reduced the number of states in the suffix-compressed, output-pruned trie from 1685 to just 283, while the number of transitions was reduced from 3627 to 2427. However, because the resulting states were larger, the first-fit packing performed rather poorly, producing a packed trie with 3408 transitions. So in this case don't care minimization yielded an additional compression of less than 10 percent.

Also, the behavior of the resulting hyphenation algorithm on words not in the dictionary became rather unpredictable. Once a word leaves the "known" paths of the packed trie, strange things might happen!

We can get even wilder effects by carrying the don't care assumption one step further, and eliminating the character field from the packed trie altogether (leaving just the output and trie link). Words in the dictionary will always index the correct transitions, but on other words we now have no way of telling when we have reached an invalid trie transition.

It turns out that the problem of state minimization with don't cares was studied in the 1960s by electrical engineers, who called it "minimization of incompletely specified sequential machines" (see e.g. [26]). However, typical instances of the problem involved machines with only a few states, rather than thousands as in our case, so it was often possible to find a minimized machine by hand. Also, the emphasis was on minimizing the number of *states* of the machine, rather than the number of state transitions.

In ordinary finite-state minimization, these are equivalent, but don't care minimization can actually introduce extra transitions, for example when states are duplicated. In the old days, finite-state machines were implemented using combinational logic, so the most important consideration was to reduce the number of states. In our trie representation, however, the space used is proportional to the number of transitions. Furthermore, finite-state machines are now often implemented using PLA's (programmed logic arrays), for which the number of transitions is also the best measure of space.

Pattern matching

Since trie minimization with don't cares still doesn't provide sufficient compression, and since it lead to unpredictable behavior on words not in the dictionary,

we need a different approach. It seems expensive to insist on complete hyphenation of the dictionary, so we will give up this requirement. We could allow some errors; or to be safer, we could allow some hyphens to be missed.

We now return to the pattern matching approach described in Chapter 1. Some further arguments as to why this method seems advantageous are given below. We should first reassure the reader that all the discussion so far has not been in vain, because a packed trie will be an ideal data structure for representing the patterns in the final hyphenation algorithm. Here the outputs will include the hyphenation level as well as the intercharacter position.

Hyphenating and inhibiting patterns allow considerable flexibility in the performance of the resulting algorithm. For example, we could allow a certain amount of error by using patterns that aren't always safe (but that presumably do find many correct hyphens).

We can also restrict ourselves to partial hyphenation in a natural way. That is, it turns out that a relatively small number of patterns will get a large fraction of the hyphens in the dictionary. The remaining hyphens become harder and harder to find, as we are left with mostly exceptional cases. Thus we can choose the most effective patterns first, taking more and more specialized patterns until we run out of space.

In addition, patterns perform quite well on words not in the dictionary, if those words follow "normal" pronunciation rules.

Patterns are "context-free"; that is, they can apply anywhere in a word. This seems to be an important advantage. In the trie-based approach discussed earlier in this chapter, a word is always scanned from beginning to end and each state of the trie 'remembers' the entire prefix of the word scanned so far, even if the letters scanned near the beginning no longer affect the hyphenation of the word. Suffix compression eliminates some of this unnecessary state information, by combining states that are identical with respect to future hyphenation. Minimization with don't cares takes this further, allowing 'similar' states to be combined as long as they behave identically on all characters that they have in common.

However, we have seen that it is difficult to guide the minimization with don't cares to achieve these reductions. Patterns embody such don't care situations naturally (if we can find a good way of selecting the patterns).

The context-free nature of patterns helps in another way, as explained below. Recall that we will use a packed trie to represent the patterns. To find all patterns that match in a given word, we perform a search starting at each letter of the word. Thus after completing a search starting from some letter position, we may have to

back up in the word to start the next search. By contrast, our original trie-based approach works with no backup.

Suppose we wanted to convert the pattern trie into a finite-state recognizer that works with no backup. This can be done in two stages. We first add "failure links" to each state that tell which state to go to if there is no explicit transition for the current character of the word. The failure state is the state in the trie that we would have reached, if we had started the search one letter later in the word.

Next, we can convert the failure-link machine into a true finite-state machine by filling in the missing transitions of each state with those of its failure state. (For more details of this process, see [27], [28].)

However, the above state merging will introduce a lot of additional transitions. Even using failure links requires one additional pointer per state. Thus by performing pattern matching with backup, we seem to save a good deal of space. And in practice, long backups rarely occur.

Finally, the idea of inhibiting patterns seems to be very useful. Such patterns extend the power of a finite-state machine, somewhat like adding the "not" operator to regular expressions.

Pattern generation

We now discuss how to choose a suitable set of patterns for hyphenation. In order to decide which patterns are "good", we must first specify the desired properties of the resulting hyphenation algorithm.

We obviously want to maximize the number of hyphens found, minimize the error, and minimize the space required by our algorithm. For example, we could try to maximize some (say linear) function of the above three quantities, or we could hold one or two of the quantities constant and optimize the others.

For TEX82, we wanted a hyphenation algorithm meeting the following requirements. The algorithm should use only a moderate amount of space (20–30K bytes), including any exception dictionary; and it should find as many hyphens as possible, while making little or no error. This is similar to the specifications for the original TEX algorithm, except that we now hope to find substantially more hyphens.

Of course, the results will depend on the word list used. We decided to base the algorithm on our copy of Webster's Pocket Dictionary, mainly because this was the only word list we had that included all derived forms.

We also thought that a larger dictionary would contain many rare or specialized words that we might not want to worry about. In particular, we did not want such infrequent words to affect the choice of patterns, because we hoped to obtain a set of patterns embodying many of the "usual" rules for hyphenation.

In developing the TEX82 algorithm, however, the word list was tuned up considerably. A few thousand common words were weighted more heavily so that they would be more likely to be hyphenated. In fact, the current algorithm guarantees complete hyphenation of the 676 most common English words (according to [29]), as well as a short list of common technical words (e.g. `al-go-rithm`).

In addition, over 1000 "exception" words have been added to the dictionary, to ensure that they would not be incorrectly hyphenated. Most of these were found by testing the algorithm (based on the initial word list) against a larger dictionary obtained from a publisher, containing about 115,000 entries. This produced about

10,000 errors on words not in the pocket dictionary. Most of these were specialized technical terms that we decided not to worry about, but a few hundred were embarrassing enough that we decided to add them to the word list. These included compound words (camp-fire), proper names (Af-ghan-i-stan), and new words (bio-rhythm) that probably did not exist in 1966, when our pocket dictionary was originally put online.

After the word list was augmented, a new set of patterns was generated, and a new list of exceptions was found and added to the list. Fortunately this process seemed to converge after a few iterations.

Heuristics

The selection of patterns in an 'optimal' way seems very difficult. The problem is that several patterns may apply to a particular hyphen point, including both hyphenating and inhibiting patterns. Thus complicated interactions can arise if we try to determine, say, the minimum set of patterns finding a given number of hyphens. (The situation is somewhat analogous to a set cover problem.)

Instead, we will select patterns in a series of "passes" through the word list. In each pass we take into account only the effects of patterns chosen in previous passes. Thus we sidestep the problem of interactions mentioned above.

In addition, we will define a measure of pattern "efficiency" so that we can use a greedy approach in each pass, selecting the most efficient patterns.

Patterns will be selected one level at a time, starting with a level of hyphenating patterns. Patterns at each level will be selected in order of increasing pattern length.

Furthermore patterns of a given length applying to different intercharacter positions (for example -tio and t-io) will be selected in separate passes through the dictionary. Thus the patterns of length n at a given level will be chosen in $n+1$ passes through the dictionary.

At first we did not do this, but selected all patterns of a given length (at a given level) in a single pass, to save time. However, we found that this resulted in considerable duplication of effort, as many hyphens were covered by two or more patterns. By considering different intercharacter positions in separate passes, there is never any overlap among the patterns selected in a single pass.

In each pass, we collect statistics on all patterns appearing in the dictionary, counting the number of times we could hyphenate at a particular point in the pattern, and the number of times we could not.

For example, the pattern tio appears 1793 times in the pocket dictionary, and in 1773 cases we can hyphenate the word before the t, while in 20 cases we can

not. (We only count instances where the hyphen position occurs at least two letters from either edge of the word.)

These counts are used to determine the efficiency rating of patterns. For example if we are considering only "safe" patterns, that is, patterns that can always be hyphenated at a particular position, then a reasonable rating is simply the number of hyphens found. We could then decide to take, say, all patterns finding at least a given number of hyphens.

However, most of the patterns we use will make some error. How should these patterns be evaluated? In the worst case, errors can be handled by simply listing them in an exception dictionary. Assuming that one unit of space is required to represent each pattern as well as each exception, the "efficiency" of a pattern could be defined as $eff = good/(1 + bad)$ where *good* is the number of hyphens correctly found and *bad* is the number of errors made.

(The space used by the final algorithm really depends on how much compression is produced by the packed trie used to represent the patterns, but since it is hard to predict the exact number of transitions required, we just use the number of patterns as an approximate measure of size.)

By using inhibiting patterns, however, we can often do better than listing the exceptions individually. The quantity *bad* in the above formula should then be devalued a bit depending on how effective patterns at the next level are. So a better formula might be

$$eff = \frac{good}{1 + bad/bad\_eff},$$

where *bad_eff* is the estimated efficiency of patterns at the next level (inhibiting errors at the current level).

Note that it may be difficult to determine the efficiency at the next level, when we are still deciding what patterns to take at the current level! We will use a pattern selection criterion of the form $eff \geq thresh$, but we cannot predict exactly how many patterns will be chosen and what their overall performance will be. The best we can do is use reasonable estimates based on previous runs of the pattern generation program. Some statistics from trial runs of this program are presented later in this chapter.

Collecting pattern statistics

So the main task of the pattern generation process is to collect count statistics about patterns in the dictionary. Because of time and space limitations this becomes an interesting data structure exercise.

For short (length 2 and 3) patterns, we can simply use a table of size 26^2 or 26^3, respectively, to hold the counts during a pass through the dictionary. For longer patterns, this is impractical.

Here's the first approach we used for longer patterns. In a pass through the dictionary, every occurrence of a pattern is written out to a file, along with an indication of whether or not a hyphen was allowed at the position under consideration. The file of patterns is sorted to bring identical patterns together, and then a pass is made through the sorted list to compile the count statistics for each pattern.

This approach makes it feasible to collect statistics for longer length patterns, and was used to conduct our initial experiments with pattern generation. However it is still quite time and space consuming, especially when sorting the large lists of patterns. Note that an external sorting algorithm is usually necessary.

Since only a fraction of the possible patterns of a particular length actually occur in the dictionary, we could instead store them in a hash table or one of the other data structures discussed in Chapter 2. It turns out that a modification of our packed trie data structure is well-suited to this task. The advantages of the packed trie are very fast lookup, compactness, and graceful handling of variable length patterns.

Combined with some judicious "pruning" of the patterns that are considered, the memory requirements are much reduced, allowing the entire pattern selection process to be carried out "in core" on our PDP-10 computer.

By "pruning" patterns we mean the following. If a pattern contains a shorter pattern at the same level that has already been chosen, the longer pattern obviously need not be considered, so we do not have to count its occurrences. Similarly, if a pattern appears so few times in the dictionary that under the current selection criterion it can never be chosen, then we can mark the pattern as "hopeless" so that any longer patterns at this level containing it need not be considered.

Pruning greatly reduces the number of patterns that must be considered, especially at longer lengths.

Dynamic packed tries

Unlike the static dictionary problem considered in Chapter 2, the set of patterns to be represented is not known in advance. In order to use a packed trie for storing the patterns being considered in a pass through the dictionary, we need some way to dynamically insert new patterns into the trie.

For any pattern, we start by performing a search in the packed trie as usual, following existing links until reaching a state where a new trie transition must be

added. If we are lucky, the location needed by the new transition will still be empty in the packed trie, otherwise we will have to do some repacking.

Note that we will not be using suffix compression, because this complicates things considerably. We would need back pointers or reference counts to determine what nodes need to be unmerged, and we would need a hash table or other auxiliary information in order to remerge the newly added nodes. Furthermore, suffix merging does not produce a great deal of compression on the relatively short patterns we will be dealing with.

The simplest way of resolving the packing conflict caused by the addition of a new transition is to just repack the changed state (and update the link of its parent state). To maintain good space utilization, we should try to fit the modified state among the holes in the trie. This can be done by maintaining a dynamic list of unoccupied cells in the trie, and using a first-fit search.

However, repacking turns out to be rather expensive for large states that are unlikely to fit into the holes in the trie, unless the array is very sparse. We can avoid this by packing such states into the free space immediately to the right of the occupied locations. The size threshold for attempting a first-fit packing can be adjusted depending on the density of the array, how much time we are willing to spend on insertions, or how close we are to running out of room.

After adding the critical transition as discussed above, we may need to add some more trie nodes for the remaining characters of the new pattern. These new states contain just a single transition, so they should be easy to fit into the trie.

The pattern generation program uses a second packed trie to store the set of patterns selected so far. Recall that, before collecting statistics about the patterns in each word, we must first hyphenate the word according to the patterns chosen in previous passes. This is done not only to determine the current partial hyphenation, but also to identify pruned patterns that need not be considered. Once again, the advantages of the packed trie are compactness and very fast "hyphenation".

At the end of a pass, we need to add new patterns, including "hopeless" patterns, to the trie. Thus it will be convenient to use a dynamic packed trie here as well. At the end of a level, we probably want to delete hopeless patterns from the trie in order to recover their space, if we are going to generate more levels. This turns out to be relatively easy; we just remove the appropriate output and return any freed nodes to the available list.

Below we give some statistics that will give an idea of how well a dynamic packed trie performs. We took the current set of 4447 hyphenation patterns, randomized them, and then inserted them one-by-one into a dynamic packed trie.

(Note that in the situations described above, there will actually be many searches per insertion, so we can afford some extra effort when performing insertions.) The patterns occupy 7214 trie nodes, but the packed trie will use more locations, depending on the setting of the first-fit packing threshold. The columns of the table show, respectively, the maximum state size for which a first-fit packing is attempted, the number of states packed, the number of locations tried by the first-fit procedure (this dominates the running time), the number of states repacked, and the number of locations used in the final packed trie.

thresh	pack	first_fit	unpack	trie_max
∞	6113	877,301	2781	9671
13	6060	761,228	2728	9458
9	6074	559,835	2742	9606
7	6027	359,537	2695	9606
5	5863	147,468	2531	10,366
4	5746	63,181	2414	11,209
3	5563	33,826	2231	13,296
2	5242	19,885	1910	15,009
1	4847	8956	1515	16,536
0	4577	6073	1245	18,628

Table 2. Dynamic packed trie statistics.

Experimental results

We now give some results from trial runs of the pattern generation program, and explain how the current TEX82 patterns were generated. As mentioned earlier, the development of these patterns involved some augmentation of the word list. The results described here are based on the latest version of the dictionary.

At each level, the selection of patterns is controlled by three parameters called *good_wt*, *bad_wt*, and *thresh*. If a pattern can be hyphenated *good* times at a particular position, but makes *bad* errors, then it will be selected if

$$good * good\_wt - bad * bad\_wt \geq thresh.$$

Note that the efficiency formula given earlier in this chapter can be converted into the above form.

We can first try using only safe patterns, that is, patterns that can always be hyphenated at a particular position. The table below shows the results when all safe patterns finding at least a given number of hyphens are chosen. Note that

parameters	patterns	hyphens	percent
1 ∞ 40	401	31,083	35.2%
1 ∞ 20	1024	45,310	51.3%
1 ∞ 10	2272	58,580	66.3%
1 ∞ 5	4603	70,014	79.2%
1 ∞ 3	7052	76,236	86.2%
1 ∞ 2	10,456	83,450	94.4%
1 ∞ 1	16,336	87,271	98.7%

Table 9. Safe hyphenating patterns.

an infinite *bad_wt* ensures that only safe patterns are chosen. The table shows the number of patterns obtained, and the number and percentage of hyphens found.

We see that, roughly speaking, halving the threshold doubles the number of patterns, but only increases the percentage of hyphens by a constant amount. The last 20 percent or so of hyphens become quite expensive to find.

(In order to save computer time, we have only considered patterns of length 6 or less in obtaining the above statistics, so the figures do not quite represent all patterns above a given threshold. In particular, the patterns at threshold 1 do not find 100% of the hyphens, although even with indefinitely long patterns there would still be a few hyphens that would not be found, such as `re-cord`.)

The space required to represent patterns in the final algorithm is slightly more than one trie transition per pattern. Each transition occupies 4 bytes (1 byte each for character and output, plus 2 bytes for trie link). The output table requires an additional 3 bytes per entry (hyphenation position, value, and next output), but there are only a few hundred outputs. Thus to stay within the desired space limitations for TEX82, we can use at most about 5000 patterns.

We next try using two levels of patterns, to see if the idea of inhibiting patterns actually pays off. The results are shown below, where in each case the initial level of hyphenating patterns is followed by a level of inhibiting patterns that remove nearly all of the error.

The last set of patterns achieves 86.7% hyphenation using 4696 patterns. By contrast, the 1 ∞ 3 patterns from the previous table achieves 86.2% with 7052 patterns. So inhibiting patterns do help. In addition, notice that we have only used "safe" inhibiting patterns above; this means that none of the good hyphens are lost. We can do better by using patterns that also inhibit some correct hyphens.

After a good deal of further experimentation, we decided to use *five* levels of patterns in the current TEX82 algorithm. The reason for this is as follows. In

parameters	patterns	hyphens		percent	
1 20 20	816	51,359	505	58.1%	0.6%
1 ∞ 1	315	0	463	58.1%	0.1%
1 10 10	1510	64,893	1694	73.5%	1.9%
1 ∞ 1	824	0	1531	73.5%	0.2%
1 5 5	2573	76,632	5254	86.7%	5.9%
1 ∞ 1	2123	0	4826	86.7%	0.5%

Table 4. Two levels of patterns.

addition to finding a high percentage of hyphens, we also wanted a certain amount of guaranteed behavior. That is, we wanted to make essentially no errors on words in the dictionary, and also to ensure complete hyphenation of certain common words.

To accomplish this, we use a final level of safe hyphenating patterns, with the threshold set as low as feasible (in our case 4). If we then weight the list of important words by a factor of at least 4, the patterns obtained will hyphenate them completely (except when a word can be hyphenated in two different ways).

To guarantee no error, the level of inhibiting patterns immediately preceding the final level should have a threshold of 1 so that even patterns applying to a single word will be chosen. Note these do not need to be "safe" inhibiting patterns, since the final level will pick up all hyphens that should be found.

The problem is, if there are too many errors remaining before the last inhibiting level, we will need too many patterns to handle them. If we use three levels in all, then the initial level of hyphenating patterns can allow just a small amount of error.

However, we would like to take advantage of the high efficiency of hyphenating patterns that allow a greater percentage of error. So instead, we will use an initial level of hyphenating patterns with relatively high threshold and allowing considerable error, followed by a 'coarse' level of inhibiting patterns removing most of the initial error. The third level will consist of relatively safe hyphenating patterns with a somewhat lower threshold than the first level, and the last two levels will be as described above.

The above somewhat vague considerations do not specify the exact pattern selection parameters that should be used for each pass, especially the first three passes. These were only chosen after much trial and error, which would take too long to describe here. We do not have any theoretical justification for these parameters; they just seem to work well.

The table below shows the parameters used to generate the current set of TEX82 patterns, and the results obtained. For levels 2 and 4, the numbers in the "hyphens"

level	parameters	patterns	hyphens		percent	
1	1 2 20 (4)	458	67,604	14,156	76.6%	16.0%
2	2 1 8 (4)	509	7407	11,942	68.2%	2.5%
3	1 4 7 (5)	985	13,198	551	83.2%	3.1%
4	3 2 1 (6)	1647	1010	2730	82.0%	0.0%
5	1 ∞ 4 (8)	1320	6428	0	89.3%	0.0%

Table 5. Current TEX82 patterns.

column show the number of good and bad hyphens inhibited, respectively. The numbers in parentheses indicate the maximum length of patterns chosen at that level.

A total of 4919 patterns (actually only 4447 because some patterns appear more than once) were obtained, compiling into a suffix-compressed packed trie occupying 5943 locations, with 181 outputs. As shown in the table, the resulting algorithm finds 89.3% of the hyphens in the dictionary. This improves on the one and two level examples discussed above. The patterns were generated in 109 passes through the dictionary, requiring about 1 hour of CPU time.

Examples

The complete list of hyphenation patterns currently used by TEX82 appears in the appendix. The digits appearing between the letters of a pattern indicate the hyphenation level, as discussed above.

Below we give some examples of the patterns in action. For each of the following words, we show the patterns that apply, the resulting hyphenation values, and the hyphenation obtained. Note that if more than one hyphenation value is specified for a given intercharacter position, then the higher value takes priority, in accordance with our level scheme. If the final value is odd, the position is an allowable hyphen point.

```
computer 4m1p pu2t 5pute put3er co4m5pu2t3er com-put-er
algorithm l1g4 lgo3 1go 2ith 4hm al1g4o3r2it4hm al-go-rithm
hyphenation hy3ph he2n hena4 hen5at 1na n2at 1tio 2io
    hy3phe2n5a4t2ion hy-phen-ation
concatenation o2n on1c 1ca 1na n2at 1tio 2io
    co2n1cate1n2a1t2ion con-cate-na-tion
mathematics math3 ath5em th2e 1ma at1ic 4cs
    math5e1mat1i4cs math-e-mat-ics
```

```
typesetting type3 e1s2e 4t3t2 2t1in type3s2e4t3t2ing
    type-set-ting
program pr2 1gr pr2o1gram pro-gram
supercalifragilisticexpialidocious
    u1pe r1c 1ca al1i ag1i gil4 il1i il4ist is1ti st2i s1tic
    1exp x3p pi3a 2i1a i2al 2id 1do 1ci 2io 2us
    su1per1cal1ifrag1il4is1t2ic1ex3p2i3al2i1do1c2io2us
    su-per-cal-ifrag-ilis-tic-ex-pi-ali-do-cious
```

Below, we show a few interesting patterns. The reader may like to try figuring out what words they apply to. (The answers appear in the Appendix.)

```
ain5o    hach4    n3uin    5spai
ay5al    h5elo    nyp4     4tarc
ear5k    if4fr    o5a5les  4todo
e2mel    l5ogo    orew4    uir4m
```

And finally, the following patterns deserve mention:

```
3tex  fon4t  high5
```

History and Conclusion

The invention of the alphabet was one of the greatest advances in the history of civilization. However, the ancient Phoenicians probably did not anticipate the fact that, centuries later, the problem of word hyphenation would become a major headache for computer typesetters all over the world.

Most cultures have evolved a linear style of communication, whereby a train of thought is converted into a sequence of symbols, which are then laid out in neat rows on a page and shipped off to a laser printer.

The trouble was, as civilization progressed and words got longer and longer, it became occasionally necessary to split them across lines. At first hyphens were inserted at arbitrary places, but in order to avoid distracting breaks such as `the-rapist`, it was soon found preferable to divide words at syllable boundaries.

Modern practice is somewhat stricter, avoiding hyphenations that might cause the reader to pronounce a word incorrectly (e.g. `considera-tion`) or where a single letter is split from a component of a compound word (e.g. `cardi-ovascular`).

The first book on typesetting, Joseph Moxon's *Mechanick Exercises* (1683), mentions the need for hyphenation but does not give any rules for it. A few dictionaries had appeared by this time, but were usually just word lists. Eventually they began to show syllable divisions to aid in pronunciation, as well as hyphenation.

With the advent of computer typesetting, interest in the problem was renewed. Hyphenation is the 'H' of 'H & J' (hyphenation and justification), which are the basic functions provided by any typesetting system. The need for automatic hyphenation presented a new and challenging problem to early systems designers.

Probably the first work on this problem, as well as many other aspects of computer typesetting, was done in the early 1950s by a French group led by G. D. Bafour. They developed a hyphenation algorithm for French, which was later adapted to English [U.S. Patent 2,762,485 (1955)].

Their method is quite simple. Hyphenations are allowed anywhere in a word except among the following letter combinations: before two consonants, two vowels,

or x; between two vowels, consonant-h, e-r, or s-s; after two consonants where the first is not l, m, n, r, or s; or after c, j, q, v, consonant-w, mm, lr, nb, nf, nl, nm, nn, or nr.

We tested this method on our pocket dictionary, and it found nearly 70 percent of the hyphens, but also about an equal amount of incorrect hyphens! Viewed in another way, about 65% of the erroneous hyphen positions are successfully inhibited, along with 30% of the correct hyphens. It turns out that a simple algorithm like this one works quite well in French; however for English this is not the case.

Other early work on automatic hyphenation is described in the proceedings of various conferences on computer typesetting (e.g. [30]). A good summary appears in [31], from which the quotes in the following paragraphs were taken.

At the Los Angeles Times, a sophisticated logical routine was developed based on the grammatical rules given in Webster's, carefully refined and adapted for computer implementation. Words were analyzed into vowel and consonant patterns which were classified into one of four types, and rules governing each type applied. Prefix, suffix, and other special case rules were also used. The results were reportedly "85–95 percent accurate", while the hyphenation logic occupies "only 5,000 positions of the 20,000 positions of the computer's magnetic core memory, less space than would be required to store 500 8-letter words averaging two hyphens per word."

Perry Publications in Florida developed a dictionary look-up method, along with their own dictionary. An in-core table mapped each word, depending on its first two letters, into a particular block of words on tape. For speed, the dictionary was divided between four tape units, and "since the RCA 301 can search tape in both directions," each tape drive maintained a "homing position" at the middle of the tape, with the most frequently searched blocks placed closest to the homing positions.

In addition, they observed that many words could be hyphenated after the 3rd, 5th, or 7th letters. So they removed all such words from the dictionary (saving some space), and if a word was not found in the dictionary, it was hyphenated after the 3rd, 5th, or 7th letter.

A hybrid approach was developed at the Oklahoma Publishing Company. First some logical analysis was used to determine the number of syllables, and to check if certain suffix and special case rules could be applied. Next the probability of hyphenation at each position in the word was estimated using three probability tables, and the most probable breakpoints were identified. (This seems to be the origin of the Time magazine algorithm described in Chapter 1.) An exception

dictionary handles the remaining cases; however there was some difference of opinion as to the size of the dictionary required to obtain satisfactory results.

Many other projects to develop hyphenation algorithms have remained proprietary or were never published. For example, IBM alone worked on "over 35 approaches to the simple problem of grammatical word division and hyphenation".

By now, we might have hoped that an "industry standard" hyphenation algorithm would exist. Indeed Berg's survey of computerized typesetting [32] contains a description of what could be considered a "generic" rule-based hyphenation algorithm (he doesn't say where it comes from). However, we have seen that any logical routine must stop short of complete hyphenation, because of the generally illogical basis of English word division.

The trend in modern systems has been toward the hybrid approach, where a logical routine is supplemented by an extensive exception dictionary. Thus the in-core algorithm serves to reduce the size of the dictionary, as well as the frequency of accessing it, as much as possible.

A number of hyphenation algorithms have also appeared in the computer science literature. A very simple algorithm is described by Rich and Stone [33]. The two parts of the word must include a vowel, not counting a final **e**, **es** or **ed**. The new line cannot begin with a vowel or double consonant. No break is made between the letter pairs **sh**, **gh**, **p**, **ch**, **th**, **wh**, **gr**, **pr**, **cr**, **tr**, **wr**, **br**, **fr**, **dr**, vowel-**r**, vowel-**n**, or **om**. On our pocket dictionary, this method found about 70% of the hyphens with 45% error.

The algorithm used in the Bell Labs document compiler Roff is described by Wagner [34]. It uses suffix stripping, followed by digram analysis carried out in a back to front manner. In addition a more complicated scheme is described using four classes of digrams combined with an attempt to identify accented and nonaccented syllables, but this seemed to introduce too many errors. A version of the algorithm is described in [35]; interestingly, this reference uses the terms "hyphenating pattern" (referring to a Snobol string-matching pattern) as well as "inhibiting suffix".

Ocker [36], in a master's thesis, describes another algorithm based on the rules in Webster's dictionary. It includes recognition of prefixes, suffixes, and special letter combinations that help in determining accentuation, followed by an analysis of the "liquidity" of letter pairs to find the character pair corresponding to the greatest interruption of spoken sound.

Moitra et al [37] use an exception table, prefixes, suffixes, and a probabilistic break-value table. In addition they extend the usual notion of affixes to any letter

pattern that helps in hyphenation, including 'root words' (e.g. `line`, `pot`) intended to handle compound words.

Patterns as paradigm

Our pattern matching approach to hyphenation is interesting for a number of reasons. It has proved to be very effective and also very appropriate for the problem. In addition, since the patterns are generated from the dictionary, it is easy to accommodate changes to the word list, as our hyphenation preferences change or as new words are added. More significantly, the pattern scheme can be readily applied to different languages, if we have a hyphenated word list for the language.

The effectiveness of pattern matching suggests that this paradigm may be useful in other applications as well. Indeed more general pattern matching systems and the related notions of production systems and augmented transition networks (ATN's) are often used in artificial intelligence applications, especially natural language processing. While AI programs try to understand sentences by analyzing word patterns, we try to hyphenate words by analyzing letter patterns.

One simple extension of patterns that we have not considered is the idea of character groups such as vowels and consonants, as used by nearly all other algorithmic approaches to hyphenation. This may seem like a serious omission, because a potentially useful meta-pattern like 'vowel-consonant-consonant-vowel' would then expand to $6 \times 20 \times 20 \times 6 = 14400$ patterns. However, it turns out that a suffix-compressed trie will reduce this to just $6 + 20 + 20 + 6 = 52$ trie nodes. So our methods can take some advantage of such "meta-patterns".

In addition, the use of inhibiting as well as hyphenating patterns seems quite powerful. These can be thought of as rules and exceptions, which is another common AI paradigm.

Concerning related work in AI, we must especially mention the Meta-DENDRAL program [38], which is designed to infer automatically rules for mass-spectrometry. An example of such a rule is N—C—C—C → N—C * C—C, which says that if the molecular substructure on the left side is present, then a bond fragmentation may occur as indicated on the right side. Meta-DENDRAL analyzes a set of mass-spectral data points and tries to infer a set of fragmentation rules that can correctly predict the spectra of new molecules. The inference process starts with some fairly general rules and then refines them as necessary, using the experimental data as positive or negative evidence for the correctness of a rule.

The fragmentation rules can in general be considerably more complicated than our simple pattern rules for hyphenation. The molecular "pattern" can be a tree-like or even cyclic structure, and there may be multiple fragmentations, possibly involving "migration" of a few atoms from one fragment to another. Furthermore, there are usually extra constraints on the form of rules, both to constrain the search and to make it more likely that meaningful or "interesting" rules will be generated. Still, there are some striking similarities between these ideas and our pattern-matching approach to hyphenation.

Packed tries

Finally, the idea of packed tries deserves further investigation. An indexed trie can be viewed as a finite-state machine, where state transitions are performed by address calculation based on the current state and input character. This is extremely fast on most computers.

However indexing usually incurs a substantial space penalty because of space reserved for pointers that are not used. Our packing technique, using the idea of storing the index character to distinguish transitions belonging to different states, combines the best features of both the linked and indexed representations, namely space and speed. We believe this is a fundamental idea.

There are various issues to be explored here. Some analysis of different packing methods would be interesting, especially for the handling of dynamic updates to a packed trie.

Our hyphenation trie extends a finite-state machine with its hyphenation "actions". It would be interesting to consider other applications that can be handled by extending the basic finite-state framework, while maintaining as much of its speed as possible.

Another possibly interesting question concerns the size of the character and pointer fields in trie transitions. In our hyphenation trie half of the space is occupied by the pointers, while in our spelling checking examples from one-half to three-fourths of the space is used for pointers, depending on the size of the dictionary. In the latter case it might be better to use a larger "character" size in the trie, in order to get a better balance between pointers and data.

When performing a search in a packed trie, following links will likely make us jump around in the trie in a somewhat random manner. This can be a disadvantage, both because of the need for large pointers, and also because of the lack of locality, which could degrade performance in a virtual memory environment. There are probably ways to improve on this. For example, Fredkin [10] proposes an interesting 'n-dimensional binary trie' idea for reducing pointer size.

We have presented packed tries as a solution to the set representation problem, with special emphasis on data compression. It would be interesting to compare our results with other compression techniques, such as Huffman coding. Also, perhaps one could estimate the amount of information present in a hyphenated word list, as a lower bound on the size of any hyphenation algorithm.

Finally, our view of finite-state machines has been based on the underlying assumption of a computer with random-access memory. Addressing by indexing seems to provide power not available in some other models of computation, such as pointer machines or comparison-based models. On the other hand, a 'VLSI' or other hardware model (such as programmed logic arrays) can provide even greater power, eliminating the need for our perhaps contrived packing technique. But then other communication issues will be raised.

If all problems of hyphenation have not been solved,
at least some progress has been made since that night,
when according to legend, an RCA Marketing Manager
received a phone call from a disturbed customer.
His 301 had just hyphenated "God".

— Paul E. Justus (1972)

TEX82 hyphenation patterns

.ach4
.ad4der
.af1t
.al3t
.am5at
.an5c
.ang4
.ani5m
.ant4
.an3te
.anti5s
.ar5s
.ar4tie
.ar4ty
.as3c
.as1p
.as1s
.aster5
.atom5
.au1d
.av4i
.awn4
.ba4g
.ba5na
.bas4e
.ber4
.be5ra
.be3sm
.be5sto
.bri2
.but4ti
.cam4pe
.can5c
.capa5b
.car5ol
.ca4t
.ce4la
.ch4
.chill5i
.ci2
.cit5r
.co3e
.co4r
.cor5ner
.de4moi
.de3o
.de3ra
.de3ri
.des4c
.dictio5
.do4t
.du4c
.dumb5
.earth5
.eas3i
.eb4
.eer4
.eg2
.el5d
.el3em
.enam3
.en3g
.en3s
.eq5ui5t
.er4ri
.es3
.eu3
.eye5
.fes3
.for5mer
.ga2
.ge2
.gen3t4
.ge5og
.gi5a
.gi4b
.go4r
.hand5i
.han5k
.he2
.hero5i
.hes3
.het3
.hi3b
.hi3er
.hon5ey
.hon3o
.hov5
.id4l
.idol3
.im3m
.im5pin
.in1
.in3ci
.ine2
.in2k
.in3s
.ir5r
.is4i
.ju3r
.la4cy
.la4m
.lat5er
.lath5
.le2
.leg5e
.len4
.lep5
.lev1
.li4g
.lig5a
.li2n
.li3o
.li4t
.mag5a5
.mal5o
.man5a
.mar5ti
.me2
.mer3c
.me5ter
.mis1
.mist5i
.mon3e
.mo3ro
.mu5ta
.muta5b
.ni4c
.od2
.odd5
.of5te
.or5ato
.or3c
.or1d
.or3t
.os3
.os4tl
.oth3
.out3
.ped5al
.pe5te
.pe5tit
.pi4e
.pio5n
.pi2t
.pre3m
.ra4c
.ran4t
.ratio5na
.ree2
.re5mit
.res2
.re5stat
.ri4g
.rit5u
.ro4q
.ros5t
.row5d
.ru4d
.sci3e
.self5
.sell5
.se2n
.se5rie
.sh2
.si2
.sing4
.st4
.sta5bl
.sy2
.ta4
.te2
.ten5an
.th2
.ti2
.til4
.tim5o5
.ting4
.tin5k
.ton4a
.to4p
.top5i
.tou5s
.trib5ut
.un1a
.un3ce
.under5
.un1e
.un5k
.un5o
.un3u
.up3
.ure3
.us5a
.ven4de
.ve5ra
.wil5i
.ye4
4ab.
a5bal
a5ban
abe2
ab5erd
abi5a
ab5it5ab
ab5lat
ab5o5liz
4abr
ab5rog
ab3ul
a4car
ac5ard
ac5aro
a5ceou
ac1er
a5chet
4a2ci
a3cie
ac1in
a3cio
ac5rob
act5if
ac3ul
ac4um
a2d
ad4din
ad5er.
2adi
a3dia
ad3ica
adi4er
a3dio
a3dit
a5diu
ad4le
ad3ow
ad5ran
ad4su
4adu
a3duc
ad5um
ae4r
aeri4e
a2f
aff4
a4gab
aga4n
ag5ell
age4o
4ageu
ag1i
4ag4l
ag1n
a2go
3agog
ag3oni
a5guer
ag5ul
a4gy
a3ha
a3he
ah4l
a3ho
ai2
a5ia
a3ic.
ai5ly
a4i4n
ain5in
ain5o
ait5en
a1j
ak1en
al5ab
al3ad
a4lar
4aldi
2ale
al3end
a4lenti
a5le5o
al1i
al4ia.
ali4e
al5lev
4allic
4alm
a5log.
a4ly.
4alys
5a5lyst
5alyt
3alyz
4ama
am5ab
am3ag
ama5ra
am5asc
a4matis
a4m5ato
am5era
am3ic
am5if
am5ily
am1in
ami4no
a2mo
a5mon
amor5i
amp5en
a2n
an3age
3analy
a3nar
an3arc
anar4i
a3nati
4and
ande4s
an3dis
an1dl
an4dow
a5nee
a3nen
an5est.
a3neu
2ang
ang5ie
an1gl
a4n1ic
a3nies
an3i3f
an4ime
a5nimi
a5nine
an3io
a3nip
an3ish
an3it
a3niu
an4kli
5anniz
ano4
an5ot
anoth5
an2sa
an4sco
an4sn
an2sp
ans3po
an4st
an4sur
antal4
an4tie
4anto
an2tr
an4tw
an3ua
an3ul
a5nur
4ao
apar4
ap5at
ap5ero
a3pher
4aphi
a4pilla
ap5illar
ap3in
ap3ita
a3pitu
a2pl
apoc5
ap5ola
apor5i
apos3t
aps5es
a3pu
aque5
2a2r
ar3act
a5rade
ar5adis
ar3al
a5ramete
aran4g
ara3p
ar4at
a5ratio
ar5ativ
a5rau
ar5av4
araw4
arbal4
ar4chan
ar5dine
ar4dr
ar5eas
a3ree
ar3ent
a5ress
ar4fi
ar4fl
ar1i
ar5ial
ar3ian
a3riet
ar4im
ar5inat
ar3io
ar2iz
ar2mi
ar5o5d
a5roni
a3roo
ar2p
ar3q
arre4
ar4sa
ar2sh
4as.
as4ab
as3ant
ashi4
a5sia.
a3sib
a3sic
5a5si4t
ask3i
as4l
a4soc
as5ph
as4sh
as3ten
as1tr
asur5a
a2ta
at3abl
at5ac
at3alo
at5ap
ate5c
at5ech
at3ego
at3en.
at3era
ater5n
a5terna
at3est
at5ev
4ath
ath5em
a5then
at4ho
ath5om
4ati.
a5tia
at5i5b
at1ic
at3if
ation5ar
at3itu
a4tog
a2tom
at5omiz
a4top
a4tos
a1tr
at5rop
at4sk
at4tag
at5te
at4th
a2tu
at5ua
at5ue
at3ul
at3ura
a2ty
au4b
augh3
au3gu
au4l2
aun5d
au3r
au5sib
aut5en
au1th
a2va
av3ag
a5van
ave4no
av3era
av5ern
av5ery
av1i
avi4er
av3ig
av5oc
a1vor
3away
aw3i
aw4ly
aws4
ax4ic
ax4id
ay5al
aye4
ays4
azi4er
azz5i
5ba.
bad5ger
ba4ge
bal1a
ban5dag
ban4e
ban3i
barbi5
bari4a
bas4si
1bat
ba4z
2b1b
b2be
b3ber
bbi4na
4b1d
4be.
beak4
beat3
4be2d
be3da
be3de
be3di
be3gi
be5gu
1bel
be1li
be3lo
4be5m
be5nig
be5nu
4bes4
be3sp
be5str
3bet
bet5iz
be5tr
be3tw
be3w
be5yo
2bf
4b3h
bi2b
bi4d
3bie
bi5en

bi4er b5uto 3chemi co3pa 4daf d2gy 5dren e4ben efil4
2b3if biv ch5ene cop3ic 2dag d1h2 dri4b e4bit e3fine
1bil 4b5w ch3er. co4pl da2m2 5di. dril4 e3br ef5i5nite
bi3liz 5by. ch3ers 4corb dan3g 1d4i3a dro4p e4cad 3efit
bina5r4 bys4 4chiin coro3n dard5 dia5b 4drow ecan5c efor5es
bin4d 1ca 5chine. cos4e dark5 di4cam 5drupli ecca5 e4fuse.
bi5net cab3in ch5iness covi 4dary d4ice 4dry eice 4egal
bi3ogr cai3bl 5chini cove4 3dat 3dict 2di32 ec5essa eger4
bi5ou cach4 5chio cow5a 4dativ 3did ds4p ec2i eg5ib
bi2t ca5den 3chit coz5e 4dato 5di3en d4sw e4cib eg4ic
3bi3tio 4cag4 chi2z co5zi 5dav4 diif d4sy ec5ificat eg5ing
bi3tr 2c5ah 3cho2 ciq dav5e di3ge d2th ec5ifie e5git5
3bit5ua ca3lat ch4ti cras5t 5day di4lato 1du ec5ify eg5n
b5itz cal4la 1ci 5crat. d1b diin di1a ec3im e4go.
bij call5in 3cia 5cratic d5c 1dina du2c eci4t e4gos
bk4 4calo ci2a5b cre3at d1d4 3dine. diuca e5cite eg1ul
b212 can5d cia5r 5cred 2de. 5dini duc5er e4clam e5gur
blath5 can4e ci5c 4c3reta deaf5 di5niz 4duct. e4clus 5egy
b4le. can4ic 4cier cre4v deb5it 1dio 4ducts e2col e1h4
blen4 can5is 5cific. cri2 de4bon dio5g du5el e4comm eher4
5blesp can3iz 4cii cri5f decan4 di4pl du4g e4compe ei2
b3lis can4ty ci4la c4rin de4cil dir2 d3ule e4conc ei5c
b4lo cany4 3cili cris4 de5com di1re dum4be e2cor ei5d
blun4t ca5per 2cim 5criti 2die4 dirt5i du4n ec3ora eig2
4bim car5om 2cin cro4pl 4dee. dis1 4dup eco5ro ei5gl
4b3n cast5er c4ina crop5o de5if 5disi du4pe eicr e3imb
bne5g cas5tig 3cinat cros4e deli4e d4is3t div e4crem e3inf
3bod 4casy cin3em cru4d del5i5q d2iti diw ec4tan eiing
bod3i ca4th c1ing 4c3s2 de5lo 1di1v d2y ec4te e5inst
bo4e 4cativ c5ing. 2c1t d4cm dij 5dyn eicu eir4d
bol3ic cav5al 5cino cta4b 5dem. d5k2 dy4se e4cul eit3e
bom4bi c3c cion4 ct5ang 3demic 4d5la dys5p ec3ula ei3th
bon4a ccha5 4cipe c5tant dem5ic. 3dle. eia4b 2e2da e5ity
bon5at cci4a ci3ph c2te de5mil 3dled e3act 4ed3d eij
3boo ccompa5 4cipic c3ter de4mons 3dles. eadi e4dier e4jud
5bor. ccon4 4cista c4ticu demor5 4dloss ead5ie ede4s ej5udi
4biora ccou3t 4cisti ctim3i 1den 2d3lo ea4ge 4edi eki4n
bor5d 2ce. 2ciit ctu4r de4nar 4d5lu ea5ger e3dia ek4la
5bore 4ced. cit3iz c4tw de3no 2dly ea4l ed3ib e1la
5bori 4cedem 5ciz cud5 denti5f dim eal5er ed3ica e4la.
5tos4 3cei ck1 c4uf de3nu 4d1n4 eal3ou ed3im e4lac
b5ota 5cel. ck3i c4ui deip 1do eam3er ediit elan4d
both5 3cell 1c4l4 cu5ity de3pa 3do. e5and edi5z el5ativ
bo4to 1cen 4clar 5culi depi4 do5de ear3a 4edo e4law
bound3 3cenc c5laratio cul4tis de2pu 5doe ear4c e4dol elaxa4
4bp 2cen4e 5clare 3cultu d3eq 2d5of ear5es edon2 e3lea
4brit 4ceni cle4m cu2ma d4erh d4og ear4ic e4dri el5ebra
broth3 3cent 4clic c3ume 5derm do4la ear4il e4dul 5elec
2b5s2 3cep clim4 cu4mi dern5iz doli4 ear5k ed5ulo e4led
bsor4 ce5ram cly4 3cun der5s do5lor ear2t ee2c el3ega
2bt 4cesa c5n cu3pi des2 dom5iz eart3e eed3i e5len
bt4l 3cessi 1co cu5py d2es. do3nat ea5sp ee2f e4lier
b4to ces5si5b co5ag cur5a4b deisc doni4 e3ass eel3i eiles
b3tr ces5t coe2 cu5ria de2s5o doo3d east3 ee4ly el2f
buf4fer cet4 2cog 1cus des3ti dop4p ea2t ee2m el2i
bu4ga c5e4ta co4gr cuss4i de3str d4or eat5en ee4na e3libe
bu3li cev4 coi4 3c4ut de4su 3dos eath3i ee4pi e4li5ic.
bumi4 2ch co3inc cu4tio de1t 4d5out e5ati5f ee2s4 el3ica
bu4n 4ch. col5i 4c5utiv de2to do4v e4a3tu eest4 e3lier
bunt4i 4ch3ab 5colo 4cutr de1v 3dox ea2v ee4ty el5igib
bu3re 5chanic col3or icy dev3il dip eav3en e5ex e5lim
bus5ie ch5a5nis com5er cze4 4dey 1dr eav5i eif e413ing
buss4e che2 con4a id2a 4dif drag5on eav5o e4f3ere e3lio
5bust cheap3 c4one 5da. d4ga 4drai 2e1b ieff e2lis
4buta 4ched con3g 2d3a4b d3ge4t dre4 e4bel. e4fic el5ish
3butio che5lo con5t dach4 dgli drea5r e4bels 5efici e3liv3

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓

```
4ella      e3ny.      er3ine     4es2to     1fa        flin4      4geno      go3ni      1head
el4lab     4en3z      eirio      e3ston     fa3bl      flo3re     4geny      5goo       3hear
ello4      e5of       4erit      2estr      fab3r      f21y5      1geo       go5riz     he4can
e5loc      eo2g       er4iu      e5stro     fa4ce      4fm        ge3om      gor5ou     h5ecat
el5og      e4oi4      eri4v      estruc5    4fag       4fn        g4ery      5gos.      h4ed
el3op.     e3ol       e4riva     e2sur      fain4      1fo        5gesi      govi       he5do5
el2sh      eop3ar     er3m4      es5urr     fall5e     5fon       geth5      g3p        he3l4i
el4ta      e1or       er4nis     es4w       4fa4ma     fon4de     4geto      1gr        hel4lis
e5lud      eo3re      4ernit     eta4b      fam5is     fon4t      ge4ty      4grada     hel4ly
el5ug      eo5rol     5erniz     eten4d     5far       fo2r       ge4v       g4rai      h5elo
e4mac      eos4       er3no      e3teo      far5th     fo5rat     4gig2      gran2      hem4p
e4mag      e4ot       2ero       ethod3     fa3ta      for5ay     g2ge       5graph.    he2n
e5man      eo4to      er5ob      eti1c      fa3the     fore5t     g3ger      g5rapher   hena4
em5ana     e5out      e5roc      e5tide     4fato      for4i      gglu5      5graphic   hen5at
em5b       e5ow       ero4r      etin4      fault5     fort5a     ggo4       4graphy    heo5r
e1me       e2pa       eriou      eti4no     4f5b       fos5       gh3in      4gray      hep5
e2mel      e3pai      eris       e5tir      4fd        4f5p       gh5out     gre4n      h4era
e4met      ep5anc     er3set     e5titio    4fe.       fra4t      gh4to      4gress.    hera3p
em3ica     e5pel      ert3er     et5itiv    feas4      f5rea      5gi.       4grit      her4ba
emi4e      e3pent     4ertl      4etn       feath3     fres5c     1gi4a      g4ro       here5a
em5igra    ep5etitio  er3tw      et5ona     fe4b       fri2       gia5r      gruf4      h3ern
emiin2     ephe4      4eru       e3tra      4feca      fril4      giic       gs2        h5erou
em5ine     e4pli      eru4t      e3tro      5fect      frol5      5gicia     g5ste      h3ery
em3i3ni    e1po       5erwau     et3ric     2fed       2f3s       g4ico      gth3       hies
e4mis      e4prec     e1s4a      et5rif     fe3li      2ft        gien5      gu4a       he2s5p
em5ish     ep5reca    e4sage.    et3rog     fe4mo      f4to       5gies.     3guard     he4t
e5miss     e4pred     e4sages    et5ros     fen2d      f2ty       gil4       2gue       het4ed
em3iz      ep3reh     es2c       et3ua      fend5e     3fu        g3imen     5gui5t     heu4
5emniz     e3pro      e2sca      et5ym      fer1       fu5el      3g4in.     3gun       hif
emo4g      e4prob     es5can     et5z       5ferr      4fug       gin5ge     3gus       hih
emoni5o    ep4sh      e3scr      4eu        fev4       fu4min     5g4ins     4gu4t      hi5an
em3pi      ep5ti5b    es5cu      e5un       4f1f       fu5ne      5gio       g3w        hi4co
e4mul      e4put      e1s2e      e3up       f4fes      fu3ri      3gir       1gy        high5
em5ula     ep5uta     e2sec      eu3ro      f4fie      fusi4      gir4l      2g5y3n     h4il2
emu3n      e1q        es5ecr     eus4       f5fin.     fus4s      g3isl      gy5ra      himer4
e3my       equi3l     es5enc     eute4      f2f5is     4futa      gi4u       h3ab4l     h4ina
en5amo     e4q3ui3s   e4sert.    euti5l     f4fly      1fy        5giv       hach4      hion4e
e4nant     eri1a      e4serts    eu5tr      f2fy       1ga        3giz       hae4m      hi4p
ench4er    era4b      e4serva    eva2p5     4fh        gaf4       gl2        hae4t      hir4l
en3dic     4erand     4esh       e2vas      1fi        5gal.      gla4       h5agu      hi3ro
e5nea      er3ar      e3sha      ev5ast     fi3a       3gali      glad5i     ha3la      hir4p
e5nee      4erati.    esh5en     e5vea      2f3ic.     ga3lo      5glas      hala3m     hir4r
en3em      2erb       eisi       ev3ell     4f3ical    2gam       igle       ha4m       his3el
en5ero     er4bl      e2sic      evel3o     f3ican     ga5met     gli4b      han4ci     his4s
en5esi     er3ch      e2sid      e5veng     4ficate    g5amo      g3lig      han4cy     hith5er
en5est     er4che     es5iden    even4i     f3icen     gan5is     3glo       5hand.     hi2v
en3etr     2ere.      es5igna    evier      fi3cer     ga3niz     glo3r      han4g      4hk
e3new      e3real     e2s5im     e5verb     fic4i      gani5za    gim        hang5er    4hil4
en5ics     ere5co     es4i4n     eivi       5ficia     4gano      g4my       hang5o     hlan4
e5nie      ere3in     esis4te    ev3id      5ficie     gar5n4     gn4a       h5a5niz    h2lo
e5nil      er5el.     esi4u      evi4l      4fics      gass4      g4na.      han4k      hlo3ri
e3nio      er3emo     e5skin     e4vin      fi3cu      gath3      gnet4t     han4te     4him
en3ish     er5ena     es4mi      evi4v      fi5del     4gativ     gini       hap3l      hmet4
en3it      er5ence    e2sol      e5voc      fight5     4gaz       g2nin      hap5t      2hin
e5niu      4erene     es3olu     e5vu       fil5i      g3b        g4nio      ha3ran     h5odiz
5eniz      er3ent     e2son      eiwa       fill5in    gd4        gino       ha5ras     h5ods
4enn       ere4q      es5ona     e4wag      4fily      2ge.       g4non      har2d      ho4g
4eno       er5ess     e1sp       e5wee      2fin       2ged       igo        hard3e     hoge4
eno4g      er3est     es3per     e3wh       5fina      geez4      3go.       har4le     hol5ar
e4nos      eret4      es5pira    ewil5      fin2d5     gel4in     gob5       harp5en    3hol4e
en3ov      eri1h      es4pre     ew3ing     fi2ne      ge5'is     5goe       har5ter    ho4ma
en4sw      eri1i      2ess       e3wit      f1in3g     ge5iiz     3g4o4g     has5s      home3
ent5age    e1ria4     es4si4b    1exp       fin4n      4gely      go3is      haun4      hon4a
4enthes    5erick     estan4     5eyc       fis4ti     1gen       gon2       5haz       ho5ny
en3ua      e3rien     es3tig     5eye.      f412       ge4nat     4g3o3na    haz3a      3hood
en5uf      eri4er     es5tim     eys4       f5less     ge5niz     gondo5     hib        hoon4
```

1760

hor5at
ho5ris
hort3e
ho5ru
hos4e
ho5sen
hos1p
1hous
house3
hov5el
4h5p
4hr4
hreo5
hro5niz
hro3po
4his2
h4sh
h4tar
ht1en
ht5es
h4ty
hu4g
hu4min
hun5ke
hun4t
hus3t4
hu4t
h1w
h4wart
hy3pe
hy3ph
hy2s
2i1a
i2al
iam4
iam5ete
i2an
4ianc
ian3i
4ian4t
ia5pe
iass4
i4ativ
ia4tric
i4atu
ibe4
ib3era
ib5ert
ib5ia
ib3in
ib5it.
ib5ite
i1bl
ib3li
i5bo
i1br
i2b5ri
i5bun
4icam
5icap
4icar
i4car.
i4cara
icas5
i4cay
iccu4
4iceo
4ich
2ici
i5cid
ic5ina
i2cip
ic3ipa
i4cly
i2c5oc
4i1cr
5icra
i4cry
ic4te
ictu2
ic4t3ua
ic3ula
ic4um
ic5uo
i3cur
2id
i4dai
id5anc
id5d
ide3al
ide4s
i2di
id5ian
idi4ar
i5die
id3io
idi5ou
id1it
id5iu
i3dle
i4dom
id3ow
i4dr
i2du
id5uo
2ie4
ied4e
5ie5ga
ield3
ien5a4
ien4e
i5enn
i3enti
i1er.
i3esc
i1est
i3et
4if.
if5ero
iff5en
if4fr
4ific.
i3fie
i3fl
4ift
2ig
iga5b
ig3era
ight3i
4igi
i3gib
ig3il
ig3in
ig3it
i4g4l
i2go
ig3or
ig5ot
i5gre
igu5i
ig1ur
i3h
4i5i4
i3j
4ik
i1la
il3a4b
i4lade
i2l5am
ila5ra
i3leg
il1er
ilev4
il5f
il1i
il3ia
il2ib
il3io
il4ist
2ilit
il2iz
ill5ab
4iln
il3oq
il4ty
il5ur
il3v
i4mag
im3age
ima5ry
imenta5r
4imet
im1i
im5ida
imi5le
i5mini
4imit
im4ni
i3mon
i2mu
im3ula
2in.
i4n3au
4inav
incel4
in3cer
4ind
in5dling
2ine
i3nee
iner4ar
i5ness
4inga
4inge
in5gen
4ingi
in5gling
4ingo
4ingu
2ini
i5ni.
i4nia
in3io
in1is
i5nite.
5initio
in3ity
4ink
4inl
2inn
2i1no
i4no4c
ino4s
i4not
2ins
in3se
insur5a
2int.
2in4th
in1u
i5nus
4iny
2io
4io.
ioge4
io2gr
i1ol
io4m
ion3at
ion4ery
ion3i
io5ph
ior3i
i4os
io5th
i5oti
io4to
i4our
2ip
ipe4
iphras4
ip3i
ip4ic
ip4re4
ip3ul
i3qua
iq5uef
iq3uid
iq3ui3t
4ir
i1ra
ira4b
i4rac
ird5e
ire4de
i4ref
i4rel4
i4res
ir5gi
ir1i
iri5de
ir4is
iri3tu
5i5r2iz
ir4min
iro4g
5iron.
ir5ul
2is.
is5ag
is3ar
isas5
2is1c
is3ch
4ise
is3er
3isf
is5han
is3hon
ish5op
is3ib
isi4d
i5sis
is5itiv
4is4k
islan4
4isms
i2so
iso5mer
is1p
is2pi
is4py
4is1s
is4sal
issen4
is4ses
is4ta.
is1te
is1ti
ist4ly
4istral
i2su
is5us
4ita.
ita4bi
i4tag
4ita5m
i3tan
i3tat
2ite
it3era
i5teri
it4es
2ith
i1ti
4itia
4i2tic
it3ica
5i5tick
it3ig
it5ill
i2tim
2itio
4itis
i4tism
i2t5o5m
4iton
i4tram
it5ry
4itt
it3uat
i5tud
it3ul
4itz.
i1u
2iv
iv3ell
iv3en.
i4v3er.
i4vers.
iv5il.
iv5io
iv1it
i5vore
iv3o3ro
i4v3ot
4i5w
ix4o
4iy
4izar
izi4
5izont
5ja
jac4q
ja4p
1je
jer5s
4jestie
4jesty
jew3
jo4p
5judg
3ka.
k3ab
k5ag
kais4
kal4
k1b
k2ed
1kee
ke4g
ke5li
k3en4d
k1er
kes4
k3est.
ke4ty
k3f
kh4
k1i
5ki.
5k2ic
k4ill
kilo5
k4im
k4in.
kin4de
k5iness
kin4g
ki4p
kis4
k5ish
kk4
k1l
4kley
4kly
k1m
k5nes
1k2no
ko5r
kosh4
k3ou
kro5n
4k1s2
k4sc
ks4l
k4sy
k5t
k1w
lab3ic
l4abo
laci4
l4ade
la3dy
lag4n
lam3o
3land
lan4dl
lan5et
lan4te
lar4g
lar3i
las4e
la5tan
4lateli
4lativ
4lav
la4v4a
2l1b
lbin4
4l1c2
lce4
l3ci
2ld
l2de
ld4ere
ld4eri
ldi4
ld5is
l3dr
l4dri
le2a
le4bi
left5
5leg.
5legg
le4mat
lem5atic
4len.
3lenc
5lene.
1lent
le3ph
le4pr
lera5b
ler4e
3lerg
3l4eri
l4ero
les2
le5sco
5lesq
3less
5less.
l3eva
lev4er.
lev4era
lev4ers
3ley
4leye
2lf
l5fr
4l1g4
l5ga
lgar3
l4ges
lgo3
2l3h
li4ag
li2am
liar5iz
li4as
li4ato
li5bi
5licio
li4cor
4lics
4lict.
l4icu
l3icy
l3ida
lid5er
3lidi
lif3er
l4iff
li4fl
5ligate
3ligh
li4gra
3lik
4l4i4l
lim4bl
lim3i
li4mo
l4im4p
l4ina
1l4ine
lin3ea
lin3i
link5er
li5og
4l4iq
lis4p
l1it
l2it.
5litica
l5i5tics
liv3er
l1iz
4lj
lka3
l3kal
lka4t
l1l
l4law
l2le
l5lea
l3lec
l3leg
l3lel
l3le4n
l3le4t
ll2i
l2lin4
l5lina
ll4o
lloqui5
ll5out
l5low
2lm
l5met
lm3ing
l4mod
lmon4
2l1n2
3lo.
lob5al
lo4ci
4lof
3logic
l5ogo
3logu
lom3er
5long
lon4i
l3o3niz
lood5
5lope.
lop3i
l3opm
lora4
lo4rato
lo5rie
lor5ou
5los.
los5et
5losophiz
5losophy
los4t
lo4ta
loun5d
2lout
4lov
2lp
lpa5b
l3pha
l5phi
lp5ing
l3pit
l4pl
l5pr
4l1r
2l1s2
l4sc
l2se
l4sie
4lt
lt5ag
ltane5
l1te
lten4
ltera4
lth3i
l5ties.

```
ltis4   4me.    m4nin   n5act   ne4po   nk3in   nti2f   o2fi    ol3ume
litr    2med    mn4o    nag5er. ne2q    nikl    n3tine  of5ite  ol3un
lt2n2   4med.   1mo     nak4    nier    4nil    n4t3ing ofit4t  o5lus
ltur3a  5media  4mocr   na4li   nera5b  n5m     nti4p   o2g5a5r o12v
lu5a    me3die  5mocratiz na5lia n4erar  nme4    ntrol5li og5ativ o21y
lu3br   m5e5dy  mo2di   4nalt   n2ere   nmet4   nt4s    o4gato  om5ah
luch4   me2g    mo4go   na5mit  n4er5i  4nin2   ntu3me  oige    oma5l
lu3ci   mel5on  mois2   n2an    ner4r   nne4    nu1a    o5gene  om5atiz
lu3en   mel4t   moi5se  nanci4  1nes    nni3al  nu4d    o5geo   om2be
luf4    me2m    4mok    nan4it  2nes.   nni4v   nu5en   o4ger   om4bl
lu5id   memio3  mo5lest nank4   4nesp   nob4l   nuf4fe  o3gie   o2me
lu4ma   1men    mo3me   nar3c   2nest   no3ble  n3uin   1oigis  om3ena
5lumi   men4a   mon5et  4nare   4nesw   n5ocl   3nu3it  og3it   om5erse
l5umn.  men5ac  mon5ge  nar3i   3netic  4n3o2d  n4um    o4gl    o4met
5lumnia men4de  moni3a  nar4l   ne4v    3noe    nu1me   o5g2ly  om5etry
lu3o    4mene   mon4ism n5arm   n5eve   4nog    n5umi   3ogniz  o3mia
luo3r   men4i   mon4ist n4as    ne4w    noge4   3nu4n   o4gro   om3ic.
4lup    mens4   mo3niz  nas4c   n3f     nois5i  n3uo    ogu5i   om3ica
luss4   mensu5  monol4  nas5ti  n4gab   no5l4i  nu3tr   1ogy    o5mid
lus3te  3ment   mo3ny.  n2at    n3gel   5nologis n1v2   2ogyn   om1in
1lut    men4te  mo2r    na3tal  nge4n4e 3nomic  n1w4    o1h2    o5mini
l5ven   me5on   4mora.  nato5miz n5gere  n5o5miz nym4    ohab5   5ommend
l5vet4  m5ersa  mos2    n2au    n3geri  no4mo   nyp4    oi2     omo4ge
21lv    2mes    mo5sey  nau3se  ng5ha   no3my   4nz     oic3es  o4mon
1ly     3mesti  mo3sp   3naut   n3gib   no4n    n3za    oi3der  om3pi
4lya    me4ta   moth3   nav4e   ng1in   non4ag  4oa     oiff4   ompro5
4lyb    met3al  m5ouf   4n1b4   n5git   non5i   oad3    oig4    o2n
ly5me   meite   3mous   ncar5   n4gla   n5oniz  o5a5les oi5let   on1a
ly3no   me5thi  mo2v    n4ces.  ngov4   4nop    oard3   o3ing   on4ac
21ys4   m4etr   4m1p    n3cha   ng5sh   5nop5o51i oas4e  oint5er o3nan
l5yse   5metric mpara5  n5cheo  n1gu    nor5ab  oast5e  o5ism   on1c
1ma     me5trie mpa5rab n5chil  n4gum   no4rary oat5i   oi5son  3oncil
2mab    me3try  mpar5i  n3chis  n2gy    4nosc   ob3a3b  oist5en 2ond
ma2ca   me4v    m3pet   nc1in   4n1h4   nos4e   o5bar   oi3ter  on5do
ma5chine 4m1f   mphas4  nc4it   nhe4    nos5t   obe4l   o5j     o3nen
ma4cl   2mh     m2pi    ncour5a nhe4    no5ta   o1bi    2ok     on5est
mag5in  5mi.    mpi4a   n1cr    nhe4    1nou    o2bin   o3ken   on4gu
5magn   mi3a    mp5ies  n1cu    3n4ia   3noun   ob5ing  ok5ie   on1ic
2mah    mid4a   m4p1in  n4dai   ni3an   nov3el3 o3br    o1la    o3nio
maid5   mid4g   m5pir   n5dan   ni4ap   now13   ob3ul   o4lan   on1is
4mald   mig4    mp5is   n1de    ni3ba   n1p4    o1ce    olass4  o5niu
ma3lig  3milia  mpo3ri  nd5est. ni4bl   npi4    och4    o12d    on3key
ma5lin  m5i5lie mpos5ite ndi4b  ni4d    npre4c  o3chet  oldie   on4odi
mal4li  m4ill   m4pous  n5d2if  ni5di   n1q     ocif3   o13er   on3omy
mal4ty  min4a   mpov5   n1dit   ni4er   n1r     o4cil   o3lesc  on3s
5mania  3mind   mp4tr   n3diz   ni2fi   nru4    o4clam  o3let   onspi4
man5is  m5inee  m2py    n5duc   ni5ficat 2n1s2  o4cod   ol4fi   onspir5a
man3iz  m4ingl  4m3r    ndu4r   n5igr   ns5ab   oc3rac  o12i    onsu4
4map    min5gli 4m1s2   nd2we   nik4    nsati4  oc5ratiz o3lia   onten4
ma5rine. m5ingly m4sh   2ne.    n1im    ns4c    ocre3   o3lice  on3t4i
ma5riz  min4t   m5si    n3ear   ni3miz  n2se    5ocrit  ol5id.  ontif5
mar4ly  m4inu   4mt     ne2b    n1in    n4s3es  octor5a o3li4f  on5um
mar3v   miot4   1mu     neb3u   5nine.  nsidi   oc3ula  o5lil   onva5
ma5sce  m2is    mula5r4 ne2c    nin4g   nsig4   o5cure  ol3ing  oo2
mas4e   mis4er. 5mult   5neck   ni4o    n2sl    od5ded  o5lio   ood5e
masit   mis5l   multi3  2ned    5nis.   ns3m    od3ic   o5lis.  ood5i
5mate   mis4ti  3mum    ne4gat  nis4ta  n4soc   odi3o   ol3ish  oo4k
math3   m5istry mun2    neg5ativ n2it   ns4pe   o2do4   o5lite  oop3i
ma3tis  4mith   4mup    5nege   n4ith   n5spi   odor3   o5litio o3ord
4matiza m2iz    mu4u    ne4la   3nitio  nsta5bl od5uct. o5liv   oost5
4m1b    4mk     4mw     ne15iz  n3itor  n1t     od5ucts ol1i4e  o2pa
mba4t5  4m11    1na     ne5mi   ni3tr   nta4b   o4el    ol5ogiz ope5d
m5bil   mim     2n1a2b  ne4mo   n1j     nter3s  o5eng   olo4r   opier
m4b3ing mma5ry  n4abu   1nen    4nk2    nt2i    o3er    ol5pl   3opera
mbi4v   4min    4nac.   4nene   n5kero  n5tib   oe4ta   o12t    4operag
4m5c    mn4a    na4ca   3neo    n3ket   nti4er  o3ev    o13ub   2oph
```

o5phan o4tes pear4l pind4 proit rb4o rev5olu riv3et r5peat
o5pher 4oth po2c p4ino 2pis2 ric re4vh riv3i rp5er.
op3ing oth5esi 2p2ed 3pi1o p2se r2ce rif r3j r3pet
o3pit oth3i4 3pede pion4 ps4h rcen4 rfu4 r3ket rp4h4
o5pon ot3ic. 3pedi p3ith p4sib r3cha r4fy rk4le rp3ing
o4posi ot5ica pedia4 pi5tha 2pit rch4er rg2 rk4lin r3po
o1pr o3tice ped4ic pi2tu pt5a4b r4ci4b rg3er ril rir4
op1u o3tif p4ee 2p3k2 p2te rc4it r3get rle4 rre4c
opy5 o3tis pee4d 1p212 p2th rcum3 r3gic r2led rre4f
o1q oto5s pek4 3plan pti3m r4dal rgi4n r4lig r4reo
o1ra ou2 pe4la plas5t ptu4r rd2i rg3ing r4lis rre4st
o5ra. ou3bl peli4e pli3a p4tw rdi4a r5gis rl5ish rri4o
o4r3ag ouch5i pe4nan pli5er pub3 rdi4er r5git r3lo4 rri4v
or5aliz ou5et p4enc 4plig pue4 rdin4 rigl rim rron4
or5ange ou4l pen4th pli4n puf4 rd3ing rgo4n rma5c rros4
ore5a ounc5er pe5on ploi4 pul3c 2re. r3gu r2me rrys4
o5real oun2d p4era. plu4m pu4m reial rh4 r3men 4rs2
or3ei ou5v pera5bl plum4b pu2n re3an 4rh. rm5ers risa
ore5sh ov4en p4erag 4pim pur4r re5arr 4rhal rm3ing rsa5ti
or5est. over4ne p4eri 2p3n 5pus 5reav ri3a r4ming. rs4c
orew4 over3s peri5st po4c pu2t re4av ria4b r4mio r2se
or4gu ov4ert per4mal 5pod. 5pute r5ebrat ri4ag r3mit r3sec
4o5ria o3vis perme5 po5em put3er rec5oll r4ib r4my rse4cr
or3ica oviti4 p4ern po3et5 pu3tr rec5ompe rib3a r4nar rs5er.
o5ril o5v4ol per3o 5po4g put4ted re4cre ric5as r3nel rs3es
ori1n ow3der per3ti poin2 put4tin 2r2ed r4ice r4ner rse5v2
o1rio ow3el pe5ru 5point p3w reide 4rici r5net rish
or3ity ow5est periv poly5t qu2 re3dis 5ricid r3ney r5sha
o3riu owli pe2t po4ni qua5v red5it ri4cie r5nic risi
or2mi own5i pe5ten po4p 2que. re4fac r4ico rinis4 r4si4b
orn2e o4wo pe5tiz 1p4or 3quer re2fe rid5er r3nit rson3
o5rof oy1a 4pf po4ry 3quet re5fer. ri3enc r3niv risp
or3oug 1pa 4pg 1pos 2rab re3fi ri3ent rno4 r5sw
or5pe pa4ca 4ph. pos1s ra3bi re4fy ri1er r4nou rtach4
3orrh pa4ce phar5i p4ot rach4e reg3is ri5et r3nu r4tag
or4se pac4t phe3no po4ta r5acl re5it rig5an rob3l r3teb
ors5en p4ad ph4er 5poun raf5fi re1li 5rigi r2oc rten4d
orst4 5pagan ph4es. 4p1p raf4t re5lu ril3iz ro3cr rte5o
or3thi p3agat phi1c ppa5ra r2ai r4en4ta 5riman ro4e riti
or3thy p4ai 5phie p2pe ra4lo ren4te rim5i ro1fe rt5ib
or4ty pain4 ph5ing p4ped ram3et rei0 3rimo ro5fil rti4d
o5rum p4al 5phisti p5pel r2ami re5pin rim4pe rok2 r4tier
o1ry pan4a 3phiz p3pen rane5o re4posi r2ina ro5ker r3tig
os3al pan3el ph2l p3per ran4ge re1pu 5rina. 5role. rti13i
os2c pan4ty 3phob p3pet r4ani r1er4 rin4d rom5ete rti14i
os4ce pa3ny 3phone ppo5site ra5no r4eri rin4e rom4i r4tily
o3scop pa1p 5phoni pr2 rap3er rero4 rin4g rom4p r4tist
4oscopi pa4pu pho4r pray4e 3raphy re5ru ri1o ron4al r4tiv
o5scr para5bl 4phs 5preci rar5c r4es. 5riph ron4e r3tri
os4i4e par5age ph3t pre5co rare4 re4spi riph5e ro5n4is rtroph4
os5itiv par5di 5phu pre3em rar5ef ress5ib ri2pl ron4ta rt4sh
os3ito 3pare 1phy pref5ac 4rari1 res2t rip5lic 1room ru3a
os3ity par5el pi3a pre4la r2as re5stal r4iq 5root ru3e4l
osi4u p4a4ri pian4 pre3r ration4 re3str r21s ro3pel ru3en
os4l par4is pi4cie p3rese rau4t re4ter r4is. rop3ic ru4gl
o2so pa2te pi4cy 3press ra5vai re4ti4z ris4c ror3i ru3in
os4pa pa5ter p4id pre5ten rav3el re3tri r3ish ro5ro rum3pl
os4po 5pathic p5ida pre3v ra5zie reu2 ris4p ros5per ru2n
os2ta pa5thy pi3de 5pri4e r1b re5uti ri3ta3b ros4s runk5
o5stati pa4tric 5pidi prin4t3 r4bab rev2 r5ited. ro4the run4ty
os5til pav4 3piec pri4s r4bag re4val rit5er. ro4ty r5usc
os5tit 3pay pi3en pris3o rbi2 rev3el rit5ers ro4va rut15n
o4tan 4p1b pi4grap p3roca rbi4f r5ev5er. rit3ic rov5el rv4e
otele4g pd4 pi3lo prof5it r2bin re5vers ri2tu rox5 rvel4i
ot3er. 4pe. pi2n pro3l r5bine re5vert rit5ur rip r3ven
ot5ers 3pe4a p4in. pros3e rb5ing. re5vil riv5el r4pea rv5er.

3543

r5vest s5ened 2s1m s2tag tal3i 2ti1f t5lo 4tuf4 ug5in
r3vey sen5g s3ma s2tal 4talk 4tig 4tim 5tu3i 2ui2
r3vic s5enin small3 stam4i tal4lis 2th. tme4 3tum ui15iz
rvi4v 4sentd sman3 5stand ta5log than4 2tin2 tu4nis ui4n
r3vo 4sentl smel4 s4ta4p ta5mo th2e 1to 2t3up. u1ing
r1w sep3a3 s5men 5stat. tan4de 4thea to3b 3ture uir4m
ry4c 4s1er. 5smith s4ted tanta3 th3eas to5crat 5turi uita4
5rynge s4erl smol5d4 stern5i ta5per the5at 4todo tur3is uiv3
ry3t ser4o s1n4 s5tero ta5pl the3is 2tof tur5o uiv4er.
sa2 4servo 1so ste2w tar4a 3thet to2gr tu5ry u5j
2s1ab s1e4s so4ce stew5a 4tarc th5ic. to5ic 3tus 4uk
5sack se5sh soft3 s3the 4tare th5ica to2ma 4tv u1la
sac3ri ses5t so4lab st2i ta3riz 4thil tom4b tw4 ula5b
s3act 5se5um sol3d2 s4ti. tas4e 5think to3my 4tiva u5lati
5sai 5sev so3lic s5tia ta5sy 4thl ton4ali twis4 ulch4
salar4 sev3en 5solv sitic 4tatic th5ode to3nat 4two 5ulche
sal4m sew4i 3som 5stick ta4tur 5thodic 4tono 1ty ul3der
sa5lo 5sex 3s4on. s4tie taun4 4thoo 4tony 4tya ul4e
sal4t 4s3f sona4 s3tif tav4 thor5it to2ra 2tyl u1len
3sanc 2s3g son4g st3ing 2taw tho5riz to3rie type3 ul4gi
san4de s2h s4op 5stir tax4is 2ths tor5iz ty5ph ul2i
s1ap 2sh. 5sophic sitle 2t1b 1tia tos2 4tz u5lia
sa5ta sh1er s5ophiz 5stock 4tc ti4ab 5tour tz4e ul3ing
5sa3tio 5shev s5ophy stom3a t4ch ti4ato 4tout 4uab ul5ish
sat3u sh1in sor5c 5stone tch5et 2ti2b to3war uac4 ul4lar
sau4 sh3io sor5d s4top 4t1d 4tick 4t1p ua5na uan4i ul4li4b
sa5vor 3ship 4sov 3store 4te. t4ico 1tra uan4i ul4lis
5saw shiv5 so5vi st4r tead4i t4iciu tra3b uar5ant 4ul3m
4s5b sho4 2spa s4trad 4teat 5tidi tra5ch uar2d u1l4o
scan4t5 sh5uld 5spai 5stratu tece4 3tien traci4 uar3i 4uls
sca4p shon3 spa4n s4tray 5tect tif2 trac4it uar3t uls5es
scav5 short4 spen4d s4trid 2t1ed ti5fy trac4te u1at ul1ti
s4ced short5 2s5peo 4stry te5di 2tig tras4 uav4 ultra3
4scei 4shw 2sper 4st3w 1tee 5tigu tra5ven ub4e 4ultu
s4ces si1b s2phe s2ty teg4 till5in trav5es5 u4bel u3lu
sch2 s5icc 3spher 1su te5ger 1tim tre5f u3ber ul5ul
s4cho 3side. spho5 su1al te5gi 4timp tre4m u4bero ul5v
3s4cie 5sides spi14 su4b3 3tel. tim5ul trem5i u1b4i um5ab
5scin4d 5sidi sp5ing su2g3 teli4 2t1in 5tria u4b5ing um4bi
scle5 si5diz 4spio su5is 5tels t2ina tri5ces u3ble. um4bly
s4cli 4signa s4ply suit3 te2ma2 3tine. 5tricia u3ca u1mi
scof4 sil4e s4pon s4ul tem3at 3tini 4trics uci4b u4m3ing
4scopy 4sily spor4 su2m 3tenan 1tio 2trim uc4it umor5o
scour5a 2s1in 4spot sum3i 3tenc ti5oc tri4v ucle3 um2p
s1cu s2ina squal4l su2n 3tend tion5ee tro5mi u3cr unat4
4s5d 5sine. s1r su2r 4tenes 5tiq tron5i u3cu u2ne
4se. s3ing 2ss 4sv 1tent ti3sa 4trony u4cy un4er
se4a 1sio s1sa sw2 ten4tag 3tise tro5phe ud5d u1ni
seas4 5sion ssas3 4swo 1teo tis4m tro3sp ud3er un4im
sea5w sion5a s2s5c s4y te4p ti5so tro3v ud5est u2nin
se2c3o si2r s3sel 4syc te5pe tis4p tru5i udev4 un5ish
3sect sir5a s5seng 3syl ter3c 5tistica trus4 u1dic uni3v
4s4ed 1sis s4ses. syn5o 5ter3d ti3tl 4tis2 ud3ied un3s4
se4d4e 3sitio s5set sy5rin 1teri ti4u t4sc ud3ies un4sw
s5edl 5siu s1si 1ta ter5ies tiv4a tsh4 ud5is unt3ab
se2g 1siv s4sie 3ta. ter3is t4sw u5dit un4ter.
seg3r 5siz ssi4er 2tab teri5za 1tiz 4t3t2 u4don un4tes
5sei sk2 ss5ily ta5bles 5ternit ti3za t4tes ud4si unu4
seile 4ske s4sl 5taboliz ter5v ti3zon t5to u4du un5y
5self s3ket ss4li 4taci 4tes. 2tl ttu4 u4ene un5z
5selv sk5ine s4sn ta5do 4tess t5la 1tu uens4 u4ors
4seme sk5ing sspend4 4taf4 t3ess. tlan4 tu1a uen4te u5os
se4mol s1l2 ss2t tai5lo teth5e 3tle. tu3ar uer4il u1ou
sen5at s3lat ssur5a ta2l 3teu 3tled tu4bi 3ufa u1pe
4senc s2le ss5w ta5la 3tex 3tles. tud2 u3fl uper5s
sen4d slith5 2st. tal5en 4tey t5let. 4tue ugh3en u5pia

4137

up3ing
u3pl
up3p
upport5
upt5ib
uptu4
u1ra
4ura.
u4rag
u4ras
ur4be
urc4
urid
ure5at
ur4fer
ur4fr
u3rif
uri4fic
uriin
u3rio
uirit
ur3iz
ur2l
uri5ing.
ur4no
uros4
ur4pe
ur4pi
urs5er
ur5tes
ur3the
urti4
ur4tie
u3ru
2us
u5sad
u5san
us4ap
usc2
us3ci
use5a
u5sia
u3sic
us4lin
usip
us5sl
us5tere
usitr
u2su
usur4
uta4b
u3tat
4ute.
4utel
4uten
uten4i
4uit2i
ut15liz
u3tine
ut3ing
ution5a
u4tis
5u5tiz
u4til
ut5of
uto5g

uto5matic
u5ton
u4tou
uts4
u3u
uu4m
u1v2
uxu3
uz4e
1va
5va.
2via4b
vac5il
vac3u
vag4
va4ge
va5lie
val5o
val1u
va5mo
va5niz
va5pi
var5ied
3vat
4ve.
4ved
veg3
v3el.
vel3li
ve4lo
v4ely
ven3om
v5enue
v4erd
5vere.
v4erel
v3eren
ver5enc
v4eres
ver3ie
vermi4n
3verse
ver3th
v4e2s
4ves.
ves4te
ve4te
vet3er
ve4ty
vi5ali
vian
5vide.
5vided
4v3iden
5vides
5vidi
v3if
vi5gn
vik4
2vil
5vilit
v3i3liz
viin
4vi4na
v2inc
vin5d

4ving
vio3l
v3io4r
viiou
vi4p
vi5ro
vis3it
vi3so
vi3su
4viti
vit3r
4vity
3viv
5vo.
voi4
3vok
vo4la
v5ole
5volt
3volv
vom5i
vor5ab
vori4
vo4ry
vo4ta
4votee
4vv4
v4y
w5abl
2wac
wa5ger
wag5o
wait5
w5al.
wam4
war4t
was4t
waite
wa5ver
wib
wea5rie
weath3
wed4n
weet3
wee5v
wel4l
wier
west3
w3ev
whi4
wi2
wil2
will5in
win4de
win4g
wir4
3wise
with3
wiz5
w4k
wl4es
wl3in
w4no
iwo2
womi
wo5ven

w5p
wra4
wri4
writa4
w3sh
ws4l
ws4pe
w5s4t
4wt
wy4
xia
xac5e
x4ago
xam3
x4ap
xas5
x3c2
xie
xe4cuto
x2ed
xer4i
xe5ro
x1h
xhi2
xhil5
xhu4
x3i
xi5a
xi5c
xi5di
x4ime
xi5miz
x3o
x4ob
x3p
xpan4d
xpecto5
xpe3d
x1t2
x3ti
xiu
xu3a
xx4
y5ac
3yar4
y5at
y1b
y1c
y2ce
yc5er
y3ch
ych4e
ycom4
ycot4
yid
y5ee
y1er
y4erf
yes4
ye4t
y5gi
4y3h
yii
y3la
y1la5bl
y3lo

y5lu
ymbol5
yme4
ympa3
yn3chr
yn5d
yn5g
yn5ic
5ynx
y1o4
yo5d
y4o5g
yom4
yo5net
y4ons
y4os
y4ped
yper5
yp3i
y3po
y4poc
yp2ta
y5pu
yra5m
yr5ia
y3ro
yr4r
ys4c
y3s2e
ys3ica
ys3io
3ysis
y4so
yss4
ysit
ys3ta
ysur4
y3thin
yt3ic
y1w
zai
z5a2b
zar2
4zb
2zc
ze4n
ze4p
z1er
ze3ro
zet4
2z1i
z4il
z4is
5zl
4zm
izo
zo4m
zo5ol
zte4
4z1z2
z4zy

4462

Answers

moun-tain-ous vil-lain-ous
be-tray-al de-fray-al por-tray-al
hear-ken
ex-treme-ly su-preme-ly
tooth-aches
bach-e-lor ech-e-lon
riff-raff
anal-o-gous ho-mol-o-gous
gen-u-ine
any-place
co-a-lesce
fore-warn fore-word
de-spair
ant-arc-tic corn-starch
mast-odon
squirmed

References

[1] Knuth, Donald E. TEX and METAFONT, *New Directions in Typesetting*. Digital Press, 1979.

[2] *Webster's Third New International Dictionary*. G. & C. Merriam, 1961.

[3] Knuth, Donald E. *The WEB System of Structured Documentation*. Preprint, Stanford Computer Science Dept., September 1982.

[4] Knuth, Donald E. *The Art of Computer Programming, Vol. 3, Sorting and Searching*. Addison-Wesley, 1973.

[5] Standish, T. A. *Data Structure Techniques*. Addison-Wesley, 1980.

[6] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. *Algorithms and Data Structures*. Addison-Wesley, 1982.

[7] Bloom, B. Space/time tradeoffs in hash coding with allowable errors. *CACM 13*, July 1970, 422–436.

[8] Carter, L., Floyd, R., Gill, J., Markowsky, G., and Wegman, M. Exact and approximate membership testers. *Proc. 10th ACM SIGACT Symp.*, 1978, 59–65.

[9] de la Briandais, Rene. File searching using variable length keys. *Proc. Western Joint Computer Conf. 15*, 1959, 295–298.

[10] Fredkin, Edward. Trie memory. *CACM 3*, Sept. 1960, 490–500.

[11] Trabb Pardo, Luis. Set representation and set intersection. Ph.D. thesis, Stanford Computer Science Dept., December 1978.

[12] Mehlhorn, Kurt. Dynamic binary search. *SIAM J. Computing 8*, May 1979, 175–198.

[13] Maly, Kurt. Compressed tries. *CACM 19*, July 1976, 409–415.

[14] Knuth, Donald E. TEX82. Preprint, Stanford Computer Science Dept., September 1982.

[15] Resnikoff, H. L. and Dolby, J. L. The nature of affixing in written English. *Mechanical Translation 8*, 1965, 84–89. Part II, June 1966, 23-33.

[16] *The Merriam-Webster Pocket Dictionary*. G. & C. Merriam, 1974.

[17] Gorin, Ralph. SPELL.REG[UP,DOC] at SU-AI.

[18] Peterson, James L. Computer programs for detecting and correcting spelling errors. *CACM 23*, Dec. 1980, 673–687.

[19] Nix, Robert. Experience with a space-efficient way to store a dictionary. *CACM 24*, May 1981, 297–298.

[20] Morris, Robert and Cherry, Lorinda L. Computer detection of typographical errors. *IEEE Trans. Prof. Comm. PC-18*, March 1975, 54–64.

[21] Downey, P., Sethi, R., and Tarjan, R. Variations on the common subexpression problem. *JACM 27*, Oct. 1980, 758–771.

[22] Tarjan, R. E. and Yao, A. Storing a sparse table. *CACM 22*, Nov. 1979, 606–611.

[23] Zeigler, S. F. Smaller faster table driven parser. Unpublished manuscript, Madison Academic Computing Center, U. of Wisconsin, 1977.

[24] Aho, Alfred V. and Ullman, Jeffrey D. *Principles of Compiler Design*, sections 3.8 and 6.8. Addison-Wesley, 1977.

[25] Pfleeger, Charles P. State reduction in incompletely specified finite-state machines. *IEEE Trans. Computers C-22*, Dec. 1973, 1099–1102.

[26] Kohavi, Zvi. *Switching and Finite Automata Theory*, section 10-4. McGraw-Hill, 1970.

[27] Knuth, D. E., Morris, J. H., and Pratt, V. R. Fast pattern matching in strings. *SIAM J. Computing 6*, June 1977, 323–350.

[28] Aho, A. V. In R. V. Book (ed.), *Formal Language Theory: Perspectives and Open Problems.* Academic Press, 1980.

[29] Kucera, Henry and Francis, W. Nelson. *Computational Analysis of Present-Day American English.* Brown University Press, 1967.

[30] Research and Engineering Council of the Graphic Arts Industry. Proceedings of the 13th Annual Conference, 1963.

[31] Stevens, M. E. and Little, J. L. *Automatic Typographic-Quality Typesetting Techniques: A State-of-the-Art Review.* National Bureau of Standards, 1967.

[32] Berg, N. Edward. *Electronic Composition, A Guide to the Revolution in Typesetting.* Graphical Arts Technical Foundation, 1975.

[33] Rich, R. P. and Stone, A. G. Method for hyphenating at the end of a printed line. *CACM 8*, July 1965, 444–445.

[34] Wagner, M. R. The search for a simple hyphenation scheme. Bell Laboratories Technical Memorandum MM-71-1371-8.

[35] Gimpel, James F. *Algorithms in Snobol 4.* Wiley-Interscience, 1976.

[36] Ocker, Wolfgang A. A program to hyphenate English words. *IEEE Trans. Prof. Comm. PC-18*, June 1975, 78–84.

[37] Moitra, A., Mudur, S. P., and Narwekar, A. W. Design and analysis of a hyphenation procedure. *Software Prac. Exper. 9*, 1979, 325–337.

[38] Lindsay, R., Buchanan, B. G., Feigenbaum, E. A., and Lederberg, J. *DENDRAL*. McGraw-Hill, 1980.