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.

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

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 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.

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.

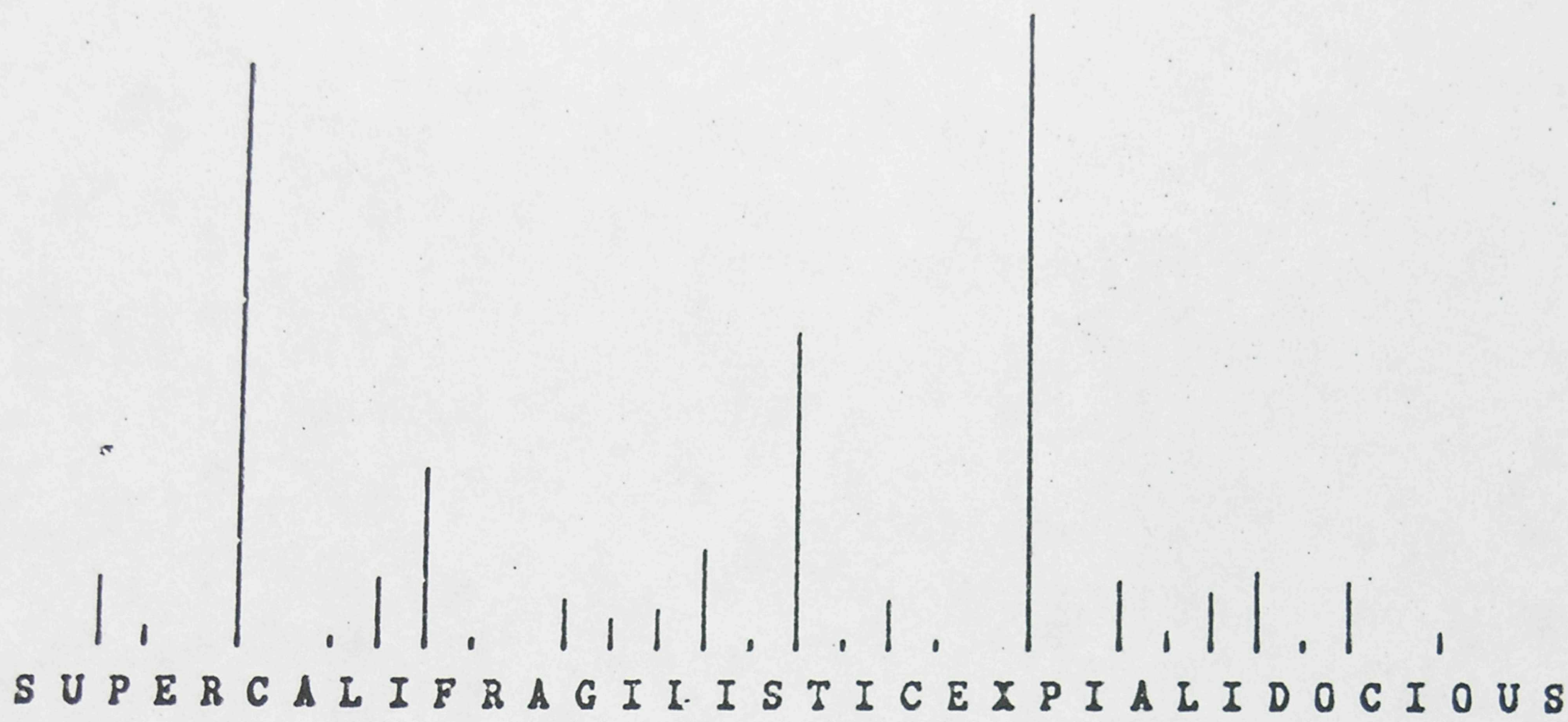

Figure 3. Hyphenation probabilities.

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.

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.

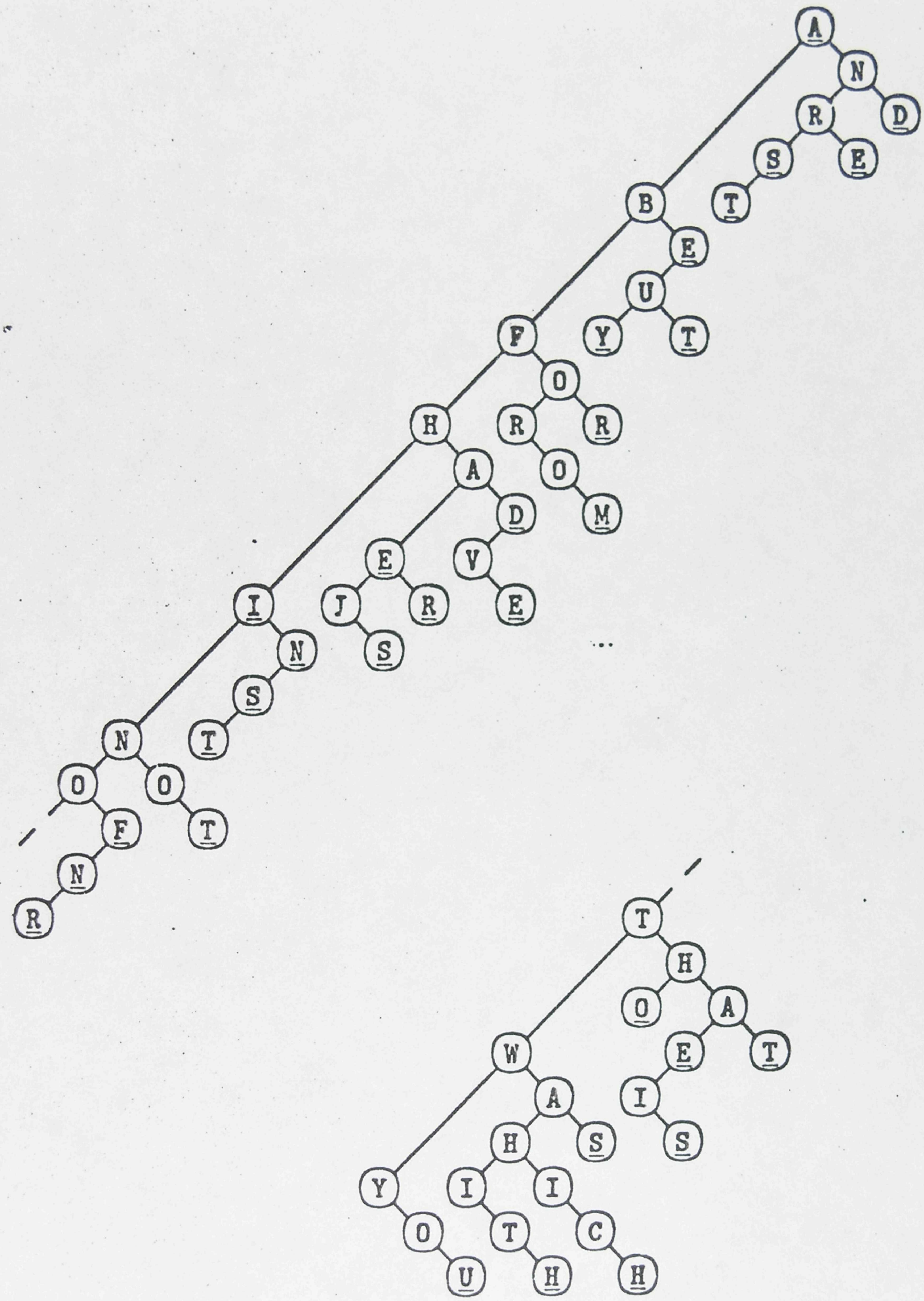

Figure 5. Linked trie for the 31 most common English words.

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

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	2726	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 leads 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 $\mathit{eff} = \mathit{good}/(1 + \mathit{bad})$ 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

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

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 $\mathit{eff} \geq \mathit{thresh}$, 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 3. 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" good wt bad wt threshold max leng

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 1go3 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 $\rightarrow$ 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
.te4
.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
2b3if
1bil
bi3liz
bina5r4
bin4d
bi5net
bi3ogr
bi5ou
bi2t
3bi3tio
bi3tr
3bit5ua
b5itz
bij
bk4
b2l2
blath5
b4le.
blen4
5blesp
b3lis
b4lo
blun4t
4b1m
4b3n
bne5g
3bod
bod3i
bo4e
bol3ic
bom4bi
bon4a
bon5at
3boo
5bor.
4b1ora
bor5d
5bore
5bori
5bos4
b5ota
both5
bo4to
bound3
4bp
4brit
broth3
2b5s2
bsor4
2bt
bt4l
b4to
b3tr
buf4fer
bu4ga
bu3li
bumi4
bu4n
bunt4i
bu3re
bus5ie
buss4e
5bust
4buta
3butio

b5uto
b1v
4b5w
5by.
bys4
1ca
cab3in
ca1bl
cach4
ca5den
4cag4
2c5ah
ca3lat
cal4la
call5in
4calo
can5d
can4e
can4ic
can5is
can3iz
can4ty
cany4
ca5per
car5om
cast5er
cas5tig
4casy
ca4th
4cativ
cav5al
c3c
ccha5
cci4a
ccompa5
ccon4
ccou3t
2ce.
4ced.
4ceden
3cei
5cel.
3cell
1cen
3cenc
2cen4e
4ceni
3cent
3cep
ce5ram
4cesa
3cessi
ces5si5b
ces5t
cet4
c5e4ta
cew4
2ch
4ch.
4ch3ab
5chanic
ch5a5nis
che2
cheap3
4ched
che5lo

3chemi
ch5ene
ch3er.
ch3ers
4ch1in
5chine.
ch5iness
5chini
5chio
3chit
chi2z
3cho2
ch4ti
1ci
3cia
ci2a5b
cia5r
ci5c
4cier
5cific.
4cii
ci4la
3cili
2cim
2cin
c4ina
3cinat
cin3em
c1ing
c5ing.
5cino
cion4
4cipe
ci3ph
4cipic
4cista
4cisti
2c1it
cit3iz
5ciz
ck1
ck3i
1c4l4
4clar
c5laratio
5clare
cle4m
4clic
clim4
cly4
c5n
1co
co5ag
coe2
2cog
co4gr
coi4
co3inc
col5i
5colo
col3or
com5er
con4a
c4one
con3g
con5t

co3pa
cop3ic
co4pl
4corb
coro3n
cos4e
cov1
cove4
cow5a
coz5e
co5zi
c1q
cras5t
5crat.
5cratic
cre3at
5cred
4c3reta
cre4v
cri2
cri5f
c4rin
cris4
5criti
cro4pl
crop5o
cros4e
cru4d
4c3s2
2c1t
cta4b
ct5ang
c5tant
c2te
c3ter
c4ticu
ctim3i
ctu4r
c4tw
cud5
c4uf
c4ui
cu5ity
5culi
cul4tis
3cultu
cu2ma
c3ume
cu4mi
3cun
cu3pi
cu5py
cur5a4b
cu5ria
1cus
cuss4i
3c4ut
cu4tie
4c5utiv
4cutr
1cy
cze4
1d2a
5da.
2d3a4b
dach4

4daf
2dag
da2m2
dan3g
dard5
dark5
4dary
3dat
4dativ
4dato
5dav4
dav5e
5day
d1b
d5c
d1d4
2de.
deaf5
deb5it
de4bon
decan4
de4cil
de5com
2d1ed
4dee.
de5if
deli4e
del5i5q
de5lo
d4em
5dem.
3demic
dem5ic.
de5mil
de4mons
demor5
1den
de4nar
de3no
denti5f
de3nu
de1p
de3pa
depi4
de2pu
d3eq
d4erh
5derm
dern5iz
der5s
des2
d2es.
de1sc
de2s5o
des3ti
de3str
de4su
de1t
de2to
de1v
dev3il
4dey
4d1f
d4ga
d3ge4t
dg1i

d2gy
d1h2
5di.
1d4i3a
dia5b
di4cam
d4ice
3dict
3did
5di3en
d1if
di3ge
di4lato
d1in
1dina
3dine.
5dini
di5niz
1dio
dio5g
di4pl
dir2
di1re
dirt5i
dis1
5disi
d4is3t
d2iti
1di1v
d1j
d5k2
4d5la
3dle.
3dled
3dles.
4dless
2d3lo
4d5lu
2dly
d1m
4d1n4
1do
3do.
do5de
5doe
2d5of
d4og
do4la
doli4
do5lor
dom5iz
do3nat
doni4
doo3d
dop4p
d4or
3dos
4d5out
do4v
3dox
d1p
1dr
drag5on
4drai
dre4
drea5r

5dren
dri4b
dril4
dro4p
4drow
5drupli
4dry
2d1s2
ds4p
d4sw
d4sy
d2th
1du
d1u1a
du2c
d1uca
duc5er
4duct.
4ducts
du5el
du4g
d3ule
dum4be
du4n
4dup
du4pe
d1v
d1w
d2y
5dyn
dy4se
dys5p
e1a4b
e3act
ead1
ead5ie
ea4ge
ea5ger
ea4l
eal5er
eal3ou
eam3er
e5and
ear3a
ear4c
ear5es
ear4ic
ear4il
ear5k
ear2t
eart3e
ea5sp
e3ass
east3
ea2t
eat5en
eath3i
e5atif
e4a3tu
ea2v
eav3en
eav5i
eav5o
2e1b
e4bel.
e4bels

e4ben
e4bit
e3br
e4cad
ecan5c
ecca5
e1ce
ec5essa
ec2i
e4cib
ec5ificat
ec5ifie
ec5ify
ec3im
eci4t
e5cite
e4clam
e4clus
e2col
e4comm
e4compe
e4conc
e2cor
ec3ora
eco5ro
e1cr
e4crem
ec4tan
ec4te
e1cu
e4cul
ec3ula
2e2da
4ed3d
e4d1er
ede4s
4edi
e3dia
ed3ib
ed3ica
ed3im
ed1it
edi5z
4edo
e4dol
edon2
e4dri
e4dul
ed5ulo
ee2c
eed3i
ee2f
eel3i
ee4ly
ee2m
ee4na
ee4p1
ee2s4
eest4
ee4ty
e5ex
e1f
e4f3ere
1eff
e4fic
5efici

efil4
e3fine
ef5i5nite
3efit
efor5es
e4fuse.
4egal
eger4
eg5ib
eg4ic
eg5ing
e5git5
eg5n
e4go.
e4gos
eg1ul
e5gur
5egy
e1h4
eher4
ei2
e5ic
ei5d
eig2
ei5gl
e3imb
e3inf
e1ing
e5inst
eir4d
eit3e
ei3th
e5ity
e1j
e4jud
ej5udi
eki4n
ek4la
e1la
e4la.
e4lac
elan4d
el5ativ
e4law
elaxa4
e3lea
el5ebra
5elec
e4led
el3ega
e5len
e4l1er
e1les
el2f
el2i
e3libe
e4l5ic.
el3ica
e3lier
el5igib
e5lim
e4l3ing
e3lio
e2lis
el5ish
e3liv3

4ella
el4lab
ello4
e5loc
el5og
el3op.
el2sh
el4ta
e5lud
el5ug
e4mac
e4mag
e5man
em5ana
em5b
e1me
e2mel
e4met
em3ica
emi4e
em5igra
em1in2
em5ine
em3i3ni
e4mis
em5ish
e5miss
em3iz
5emniz
emo4g
emoni5o
em3pi
e4mul
em5ula
emu3n
e3my
en5amo
e4nant
ench4er
en3dic
e5nea
e5nee
en3em
en5ero
en5esi
en5est
en3etr
e3new
en5ics
e5nie
e5nil
e3nio
en3ish
en3it
e5niu
5eniz
4enn
4eno
eno4g
e4nos
en3ov
en4sw
ent5age
4enthes
en3ua
en5uf
e3ny.
4en3z
e5of
eo2g
e4oi4
e3ol
eop3ar
e1or
eo3re
eo5rol
eos4
e4ot
eo4to
e5out
e5ow
e2pa
e3pai
ep5anc
e5pel
e3pent
ep5etitio
ephe4
e4pli
e1po
e4prec
ep5reca
e4pred
ep3reh
e3pro
e4prob
ep4sh
ep5ti5b
e4put
ep5uta
e1q
equi3l
e4q3ui3s
er1a
era4b
4erand
er3ar
4erati.
2erb
er4bl
er3ch
er4che
2ere.
e3real
ere5co
ere3in
er5el.
er3emo
er5ena
er5ence
4erene
er3ent
ere4q
er5ess
er3est
eret4
er1h
er1i
e1ria4
5erick
e3rien
eri4er
er3ine
e1rio
4erit
er4iu
eri4v
e4riva
er3m4
er4nis
4ernit
5erniz
er3no
2ero
er5ob
e5roc
ero4r
er1ou
er1s
er3set
ert3er
4ertl
er3tw
4eru
eru4t
5erwau
e1s4a
e4sage.
e4sages
es2c
e2sca
es5can
e3scr
es5cu
e1s2e
e2sec
es5ecr
es5enc
e4sert.
e4serts
e4serva
4esh
e3sha
esh5en
e1si
e2sic
e2sid
es5iden
es5igna
e2s5im
es4i4n
esis4te
esi4u
e5skin
es4mi
e2sol
es3olu
e2son
es5ona
e1sp
es3per
es5pira
es4pre
2ess
es4si4b
estan4
es3tig
es5tim
4es2to
e3ston
2estr
e5stro
estruc5
e2sur
es5urr
es4w
eta4b
eten4d
e3teo
ethod3
et1ic
e5tide
etin4
eti4no
e5tir
e5titio
et5itiv
4etn
et5ona
e3tra
e3tre
et3ric
et5rif
et3rog
et5ros
et3ua
et5ym
et5z
4eu
e5un
e3up
eu3ro
eus4
eute4
euti5l
eu5tr
eva2p5
e2vas
ev5ast
e5vea
ev3ell
evel3o
e5veng
even4i
ev1er
e5verb
e1vi
ev3id
evi4l
e4vin
evi4v
e5voc
e5vu
e1wa
e4wag
e5wee
e3wh
ewil5
ew3ing
e3wit
1exp
5eyc
5eye.
eys4
1fa
fa3bl
fab3r
fa4ce
4fag
fain4
fall5e
4fa4ma
fam5is
5far
far5th
fa3ta
fa3the
4fato
fault5
4f5b
4fd
4fe.
feas4
feath3
fe4b
4feca
5fect
2fed
fe3li
fe4mo
fen2d
fend5e
fer1
5ferr
fev4
4f1f
f4fes
f4fie
f5fin.
f2f5is
f4fly
f2fy
4fh
1fi
fi3a
2f3ic.
4f3ical
f3ican
4ficate
f3icen
fi3cer
fic4i
5ficia
5ficie
4fics
fi3cu
fi5del
fight5
fil5i
fill5in
4fily
2fin
5fina
fin2d5
fi2ne
f1in3g
fin4n
fis4ti
f4l2
f5less
flin4
flo3re
f2ly5
4fm
4fn
1fo
5fon
fon4de
fon4t
fo2r
fo5rat
for5ay
fore5t
for4i
fort5a
fos5
4f5p
fra4t
f5rea
fres5c
fri2
fril4
frol5
2f3s
2ft
f4to
f2ty
3fu
fu5el
4fug
fu4min
fu5ne
fu3ri
fusi4
fus4s
4futa
1fy
1ga
gaf4
5gal.
3gali
ga3lo
2gam
ga5met
g5amo
gan5is
ga3niz
gani5za
4gano
gar5n4
gass4
gath3
4gativ
4gaz
g3b
gd4
2ge.
2ged
geez4
gel4in
ge5lis
ge5liz
4gely
1gen
ge4nat
ge5niz
4geno
4geny
1geo
ge3om
g4ery
5gesi
geth5
4geto
ge4ty
ge4v
4g1g2
g2ge
g3ger
gglu5
ggo4
gh3in
gh5out
gh4to
5gi.
1gi4a
gia5r
g1ic
5gicia
g4ico
gien5
5gies.
gil4
g3imen
3g4in.
gin5ge
5g4ins
5gio
3gir
gir4l
g3isl
gi4u
5giv
3giz
gl2
gla4
glad5i
5glas
1gle
gli4b
g3lig
3glo
glo3r
g1m
g4my
gn4a
g4na.
gnet4t
g1ni
g2nin
g4nio
g1no
g4non
1go
3go.
gob5
5goe
3g4o4g
go3is
gon2
4g3o3na
gondo5
go3ni
5goo
go5riz
gor5ou
5gos.
gov1
g3p
1gr
4grada
g4rai
gran2
5graph.
g5rapher
5graphic
4graphy
4gray
gre4n
4gress.
4grit
g4ro
gruf4
gs2
g5ste
gth3
gu4a
3guard
2gue
5gui5t
3gun
3gus
4gu4t
g3w
1gy
2g5y3n
gy5ra
h3ab4l
hach4
hae4m
hae4t
h5agu
ha3la
hala3m
ha4m
han4ci
han4cy
5hand.
han4g
hang5er
hang5o
h5a5niz
han4k
han4te
hap3l
hap5t
ha3ran
ha5ras
har2d
hard3e
har4le
harp5en
har5ter
has5s
haun4
5haz
haz3a
h1b
1head
3hear
he4can
h5ecat
h4ed
he5do5
he3l4i
hel4lis
hel4ly
h5elo
hem4p
he2n
hena4
hen5at
heo5r
hep5
h4era
hera3p
her4ba
here5a
h3ern
h5erou
h3ery
h1es
he2s5p
he4t
het4ed
heu4
h1f
h1h
hi5an
hi4co
high5
h4il2
himer4
h4ina
hion4e
hi4p
hir4l
hi3ro
hir4p
hir4r
his3el
his4s
hith5er
hi2v
4hk
4h1l4
hlan4
h2lo
hlo3ri
4h1m
hmet4
2h1n
h5odiz
h5ods
ho4g
hoge4
hol5ar
3hol4e
ho4ma
home3
hon4a
ho5ny
3hood
hoon4

1760

hor5at	4iceo	ig3in	4ingu	ir4min	it3uat	k1m	3less	l3leg
ho5ris	4ich	ig3it	2ini	iro4g	i5tud	k5nes	5less.	l3lel
hort3e	2ici	i4g4l	i5ni.	5iron.	it3ul	1k2no	l3eva	l3le4n
ho5ru	i5cid	i2go	i4nia	ir5ul	4itz.	ko5r	lev4er.	l3le4t
hos4e	ic5ina	ig3or	in3io	2is.	i1u	kosh4	lev4era	ll2i
ho5sen	i2cip	ig5ot	in1is	is5ag	2iv	k3ou	lev4ers	l2lin4
hos1p	ic3ipa	i5gre	i5nite.	is3ar	iv3ell	kro5n	3ley	l5lina
1hous	i4cly	igu5i	5initio	isas5	iv3en.	4k1s2	4leye	ll4o
house3	i2c5oc	ig1ur	in3ity	2is1c	i4v3er.	k4sc	2lf	lloqui5
hov5el	4i1cr	i3h	4ink	is3ch	i4vers.	ks4l	l5fr	ll5out
4h5p	5icra	4i5i4	4inl	4ise	iv5il.	k4sy	4l1g4	l5low
4hr4	i4cry	i3j	2inn	is3er	iv5io	k5t	l5ga	2lm
hreo5	ic4te	4ik	2i1no	3isf	iv1it	k1w	lgar3	l5met
hro5niz	ictu2	i1la	i4no4c	is5han	i5vore	lab3ic	l4ges	lm3ing
hro3po	ic4t3ua	il3a4b	ino4s	is3hon	iv3o3ro	l4abo	lgo3	l4mod
4h1s2	ic3ula	i4lade	i4not	ish5op	i4v3ot	laci4	2l3h	lmon4
h4sh	ic4um	i2l5am	2ins	is3ib	4i5w	l4ade	li4ag	2l1n2
h4tar	ic5uo	ila5ra	in3se	isi4d	ix4o	la3dy	li2am	3lo.
ht1en	i3cur	i3leg	insur5a	i5sis	4iy	lag4n	liar5iz	lob5al
ht5es	2id	il1er	2int.	is5itiv	4izar	lam3o	li4as	lo4ci
h4ty	i4dai	ilev4	2in4th	4is4k	izi4	3land	li4ato	4lof
hu4g	id5anc	il5f	in1u	islan4	5izont	lan4dl	li5bi	3logic
hu4min	id5d	il1i	i5nus	4isms	5ja	lan5et	5licio	l5ogo
hun5ke	ide3al	il3ia	4iny	i2so	jac4q	lan4te	li4cor	3logu
hun4t	ide4s	il2ib	2io	iso5mer	ja4p	lar4g	4lics	lom3er
hus3t4	i2di	il3io	4io.	is1p	1je	lar3i	4lict.	5long
hu4t	id5ian	il4ist	ioge4	is2pi	jer5s	las4e	l4icu	lon4i
h1w	idi4ar	2ilit	io2gr	is4py	4jestie	la5tan	l3icy	l3o3niz
h4wart	i5die	il2iz	i1ol	4is1s	4jesty	4lateli	l3ida	lood5
hy3pe	id3io	ill5ab	io4m	is4sal	jew3	4lativ	lid5er	5lope.
hy3ph	idi5u	4iln	ion3at	issen4	jo4p	4lav	3lidi	lop3i
hy2s	id1it	il3oq	ion4ery	is4ses	5judg	la4v4a	lif3er	l3opm
2i1a	id5iu	il4ty	ion3i	is4ta.	3ka.	2l1b	l4iff	lora4
i2al	i3dle	il5ur	io5ph	is1te	k3ab	lbin4	li4fl	lo4rato
iam4	i4dom	il3v	ior3i	is1ti	k5ag	4l1c2	5ligate	lo5rie
iam5ete	id3ow	i4mag	i4os	ist4ly	kais4	lce4	3ligh	lor5ou
i2an	i4dr	im3age	io5th	4istral	kal4	l3ci	li4gra	5los.
4ianc	i2du	ima5ry	i5oti	i2su	k1b	2ld	3lik	los5et
ian3i	id5uo	imenta5r	io4to	is5us	k2ed	l2de	4l4i4l	5losophiz
4ian4t	2ie4	4imet	i4our	4ita.	1kee	ld4ere	lim4bl	5losophy
ia5pe	ied4e	im1i	2ip	ita4bi	ke4g	ld4eri	lim3i	los4t
iass4	5ie5ga	im5ida	ipe4	i4tag	ke5li	ldi4	li4mo	lo4ta
i4ativ	ield3	imi5le	iphras4	4ita5m	k3en4d	ld5is	l4im4p	loun5d
ia4tric	ien5a4	i5mini	ip3i	i3tan	k1er	l3dr	l4ina	2lout
i4atu	ien4e	4imit	ip4ic	i3tat	kes4	l4dri	1l4ine	4lov
ibe4	i5enn	im4ni	ip4re4	2ite	k3est.	le2a	lin3ea	2lp
ib3era	i3enti	i3mon	ip3ul	it3era	ke4ty	le4bi	lin3i	lpa5b
ib5ert	i1er.	i2mu	i3qua	i5teri	k3f	left5	link5er	l3pha
ib5ia	i3esc	im3ula	iq5uef	it4es	kh4	5leg.	li5og	l5phi
ib3in	i1est	2in.	iq3uid	2ith	k1i	5legg	4l4iq	lp5ing
ib5it.	i3et	i4n3au	iq3ui3t	i1ti	5ki.	le4mat	lis4p	l3pit
ib5ite	4if.	4inav	4ir	4itia	5k2ic	lem5atic	l1it	l4pl
i1bl	if5ero	incel4	i1ra	4i2tic	k4ill	4len.	l2it.	l5pr
ib3li	iff5en	in3cer	ira4b	it3ica	kilo5	3lenc	5litica	4l1r
i5bo	if4fr	4ind	i4rac	5i5tick	k4im	5lene.	l5i5tics	2l1s2
i1br	4ific.	in5dling	ird5e	it3ig	k4in.	1lent	liv3er	l4sc
i2b5ri	i3fie	2ine	ire4de	it5ill	kin4de	le3ph	l1iz	l2se
i5bun	i3fl	i3nee	i4ref	i2tim	k5iness	le4pr	4lj	l4sie
4icam	4ift	iner4ar	i4rel4	2itio	kin4g	lera5b	lka3	4lt
5icap	2ig	i5ness	i4res	4itis	ki4p	ler4e	l3kal	lt5ag
4icar	iga5b	4inga	ir5gi	i4tism	kis4	3lerg	lka4t	ltane5
i4car.	ig3era	4inge	ir1i	i2t5o5m	k5ish	3l4eri	l1l	l1te
i4cara	ight3i	in5gen	iri5de	4iton	kk4	l4ero	l4law	lten4
icas5	4igi	4ingi	ir4is	i4tram	k1l	les2	l2le	ltera4
i4cay	i3gib	in5gling	iri3tu	it5ry	4kley	le5sco	l5lea	lth3i
iccu4	ig3il	4ingo	5i5r2iz	4itt	4kly	5lesq	l3lec	l5ties.

ltis4
l1tr
ltu2
ltur3a
lu5a
lu3br
luch4
lu3ci
lu3en
luf4
lu5id
lu4ma
5lumi
l5umn.
5lumnia
lu3o
luo3r
4lup
luss4
lus3te
1lut
l5ven
l5vet4
2l1w
1ly
4lya
4lyb
ly5me
ly3no
2lys4
l5yse
1ma
2mab
ma2ca
ma5chine
ma4cl
mag5in
5magn
2mah
maid5
4mald
ma3lig
ma5lin
mal4li
mal4ty
5mania
man5is
man3iz
4map
ma5rine.
ma5riz
mar4ly
mar3v
ma5sce
mas4e
mas1t
5mate
math3
ma3tis
4matiza
4m1b
mba4t5
m5bil
m4b3ing
mbi4v
4m5c
4me.
2med
4med.
5media
me3die
m5e5dy
me2g
mel5on
mel4t
me2m
mem1o3
1men
men4a
men5ac
men4de
4mene
men4i
mens4
mensu5
3ment
men4te
me5on
m5ersa
2mes
3mesti
me4ta
met3al
me1te
me5thi
m4etr
5metric
me5trie
me3try
me4v
4m1f
2mh
5mi.
mi3a
mid4a
mid4g
mig4
3milia
m5i5lie
m4ill
min4a
3mind
m5inee
m4ingl
min5gli
m5ingly
min4t
m4inu
miot4
m2is
mis4er.
mis5l
mis4ti
m5istry
4mith
m2iz
4mk
4m1l
m1m
mma5ry
4m1n
mn4a
m4nin
mn4o
1mo
4mocr
5mocratiz
mo2d1
mo4go
mois2
moi5se
4mok
mo5lest
mo3me
mon5et
mon5ge
moni3a
mon4ism
mon4ist
mo3niz
monol4
mo3ny.
mo2r
4mora.
mos2
mo5sey
mo3sp
moth3
m5ouf
3mous
mo2v
4m1p
mpara5
mpa5rab
mpar5i
m3pet
mphas4
m2pi
mpi4a
mp5ies
m4p1in
m5pir
mp5is
mpo3ri
mpos5ite
m4pous
mpov5
mp4tr
m2py
4m3r
4m1s2
m4sh
m5si
4mt
1mu
mula5r4
5mult
multi3
3mum
mun2
4mup
mu4u
4mw
1na
2n1a2b
n4abu
4nac.
na4ca
n5act
nag5er.
nak4
na4li
na5lia
4nalt
na5mit
n2an
nanci4
nan4it
nank4
nar3c
4nare
nar3i
nar4l
n5arm
n4as
nas4c
nas5ti
n2at
na3tal
nato5miz
n2au
nau3se
3naut
nav4e
4n1b4
ncar5
n4ces.
n3cha
n5cheo
n5chil
n3chis
nc1in
nc4it
ncour5a
n1cr
n1cu
n4dai
n5dan
n1de
nd5est.
ndi4b
n5d2if
n1dit
n3diz
n5duc
ndu4r
nd2we
2ne.
n3ear
ne2b
neb3u
ne2c
5neck
2ned
ne4gat
neg5ativ
5nege
ne4la
nel5iz
ne5mi
ne4mo
1nen
4nene
3neo
ne4po
ne2q
n1er
nera5b
n4erar
n2ere
n4er5i
ner4r
1nes
2nes.
4nesp
2nest
4nesw
3netic
ne4v
n5eve
ne4w
n3f
n4gab
n3gel
nge4n4e
n5gere
n3geri
ng5ha
n3gib
ng1in
n5git
n4gla
ngov4
ng5sh
n1gu
n4gum
n2gy
4n1h4
nha4
nhab3
nhe4
3n4ia
ni3an
ni4ap
ni3ba
ni4bl
ni4d
ni5di
ni4er
ni2fi
ni5ficat
n5igr
nik4
n1im
ni3miz
n1in
5nine.
nin4g
ni4o
5nis.
nis4ta
n2it
n4ith
3nitio
n3itor
ni3tr
n1j
4nk2
n5kero
n3ket
nk3in
n1kl
4n1l
n5m
nme4
nmet4
4n1n2
nne4
nni3al
nni4v
nob4l
no3ble
n5ocl
4n3o2d
3noe
4nog
noge4
nois5i
no5l4i
5nologis
3nomic
n5o5miz
no4mo
no3my
no4n
non4ag
non5i
n5oniz
4nop
5nop5o5li
nor5ab
no4rary
4nosc
nos4e
nos5t
no5ta
1nou
3noun
nov3el3
nowl3
n1p4
npi4
npre4c
n1q
n1r
nru4
2n1s2
ns5ab
nsati4
ns4c
n2se
n4s3es
nsid1
nsig4
n2sl
ns3m
n4soc
ns4pe
n5spi
nsta5bl
n1t
nta4b
nter3s
nt2i
n5tib
nti4er
nti2f
n3tine
n4t3ing
nti4p
ntrol5li
nt4s
ntu3me
nu1a
nu4d
nu5en
nuf4fe
n3uin
3nu3it
n4um
nu1me
n5umi
3nu4n
n3uo
nu3tr
n1v2
n1w4
nym4
nyp4
4nz
n3za
4oa
oad3
o5a5les
oard3
oas4e
oast5e
oat5i
ob3a3b
o5bar
obe4l
o1bi
o2bin
ob5ing
o3br
ob3ul
o1ce
och4
o3chet
ocif3
o4cil
o4clam
o4cod
oc3rac
oc5ratiz
ocre3
5ocrit
octor5a
oc3ula
o5cure
od5ded
od3ic
odi3o
o2do4
odor3
od5uct.
od5ucts
o4el
o5eng
o3er
oe4ta
o3ev
o2fi
of5ite
ofit4t
o2g5a5r
og5ativ
o4gato
o1ge
o5gene
o5geo
o4ger
o3gie
1o1gis
og3it
o4gl
o5g2ly
3ogniz
o4gro
ogu5i
1ogy
2ogyn
o1h2
ohab5
oi2
oic3es
oi3der
oiff4
oig4
oi5let
o3ing
oint5er
o5ism
oi5son
oist5en
oi3ter
o5j
2ok
o3ken
ok5ie
o1la
o4lan
olass4
ol2d
old1e
ol3er
o3lesc
o3let
ol4fi
ol2i
o3lia
o3lice
ol5id.
o3li4f
o5lil
ol3ing
o5lio
o5lis.
ol3ish
o5lite
o5litio
o5liv
olli4e
ol5ogiz
olo4r
ol5pl
ol2t
ol3ub
ol3ume
ol3un
o5lus
ol2v
o2ly
om5ah
oma5l
om5atiz
om2be
om4bl
o2me
om3ena
om5erse
o4met
om5etry
o3mia
om3ic.
om3ica
o5mid
om1in
o5mini
5ommend
omo4ge
o4mon
om3pi
ompro5
o2n
on1a
on4ac
o3nan
on1c
3oncil
2ond
on5do
o3nen
on5est
on4gu
on1ic
o3nio
on1is
o5niu
on3key
on4odi
on3omy
on3s
onspi4
onspir5a
onsu4
onten4
on3t4i
ontif5
on5um
onva5
oo2
ood5e
ood5i
oo4k
oop3i
o3ord
oost5
o2pa
ope5d
op1er
3opera
4operag
2oph

o5phan
o5pher
op3ing
o3pit
o5pon
o4posi
o1pr
op1u
opy5
o1q
o1ra
o5ra.
o4r3ag
or5aliz
or5ange
ore5a
o5real
or3ei
ore5sh
or5est.
orew4
or4gu
4o5ria
or3ica
o5ril
or1in
o1rio
or3ity
o3riu
or2mi
orn2e
o5rof
or3oug
or5pe
3orrh
or4se
ors5en
orst4
or3thi
or3thy
or4ty
o5rum
o1ry
os3al
os2c
os4ce
o3scop
4oscopi
o5scr
os4i4e
os5itiv
os3ito
os3ity
osi4u
os4l
o2so
os4pa
os4po
os2ta
o5stati
os5til
os5tit
o4tan
otele4g
ot3er.
ot5ers

o4tes
4oth
oth5esi
oth3i4
ot3ic.
ot5ica
o3tice
o3tif
o3tis
oto5s
ou2
ou3bl
ouch5i
ou5et
ou4l
ounc5er
oun2d
ou5v
ov4en
over4ne
over3s
ov4ert
o3vis
oviti4
o5v4ol
ow3der
ow3el
ow5est
ow1i
own5i
o4wo
oy1a
1pa
pa4ca
pa4ce
pac4t
p4ad
5pagan
p3agat
p4ai
pain4
p4al
pan4a
pan3el
pan4ty
pa3ny
pa1p
pa4pu
para5bl
par5age
par5di
3pare
par5el
p4a4ri
par4is
pa2te
pa5ter
5pathic
pa5thy
pa4tric
pav4
3pay
4p1b
pd4
4pe.
3pe4a

pear4l
pe2c
2p2ed
3pede
3pedi
pedia4
ped4ic
p4ee
pee4d
pek4
pe4la
peli4e
pe4nan
p4enc
pen4th
pe5on
p4era.
pera5bl
p4erag
p4eri
peri5st
per4mal
perme5
p4ern
per3o
per3ti
pe5ru
per1v
pe2t
pe5ten
pe5tiz
4pf
4pg
4ph.
phar5i
phe3no
ph4er
ph4es.
ph1ic
5phie
ph5ing
5phisti
3phiz
ph2l
3phob
3phone
5phoni
pho4r
4phs
ph3t
5phu
1phy
pi3a
pian4
pi4cie
pi4cy
p4id
p5ida
pi3de
5pidi
3piec
pi3en
pi4grap
pi3lo
pi2n
p4in.

pind4
p4ino
3pi1o
pion4
p3ith
pi5tha
pi2tu
2p3k2
1p2l2
3plan
plas5t
pli3a
pli5er
4plig
pli4n
ploi4
plu4m
plum4b
4p1m
2p3n
po4c
5pod.
po5em
po3et5
5po4g
poin2
5point
poly5t
po4ni
po4p
1p4or
po4ry
1pos
pos1s
p4ot
po4ta
5poun
4p1p
ppa5ra
p2pe
p4ped
p5pel
p3pen
p3per
p3pet
ppo5site
pr2
pray4e
5preci
pre5co
pre3em
pref5ac
pre4la
pre3r
p3rese
3press
pre5ten
pre3v
5pri4e
prin4t3
pri4s
pris3o
p3roca
prof5it
pro3l
pros3e

pro1t
2p1s2
p2se
ps4h
p4sib
2p1t
pt5a4b
p2te
p2th
pti3m
ptu4r
p4tw
pub3
pue4
puf4
pul3c
pu4m
pu2n
pur4r
5pus
pu2t
5pute
put3er
pu3tr
put4ted
put4tin
p3w
qu2
qua5v
2que.
3quer
3quet
2rab
ra3bi
rach4e
r5acl
raf5fi
raf4t
r2ai
ra4lo
ram3et
r2ami
rane5o
ran4ge
r4ani
ra5no
rap3er
3raphy
rar5c
rare4
rar5ef
4raril
r2as
ration4
rau4t
ra5vai
rav3el
ra5zie
r1b
r4bab
r4bag
rbi2
rbi4f
r2bin
r5bine
rb5ing.

rb4o
r1c
r2ce
rcen4
r3cha rc4h
rch4er
r4ci4b
rc4it
rcum3
r4dal
rd2i
rdi4a
rdi4er
rdin4
rd3ing
2re.
re1al
re3an
re5arr
5reav
re4aw
r5ebrat
rec5oll
rec5ompe
re4cre
2r2ed
re1de
re3dis
red5it
re4fac
re2fe
re5fer.
re3fi
re4fy
reg3is
re5it
re1li
re5lu
r4en4ta
ren4te
re1o
re5pin
re4posi
re1pu
r1er4
r4eri
rero4
re5ru
r4es.
re4spi
ress5ib
res2t
re5stal
re3str
re4ter
re4ti4z
re3tri
reu2
re5uti
rev2
re4val
rev3el
r5ev5er.
re5vers
re5vert
re5vil

rev5olu
re4wh
r1f
rfu4
r4fy
rg2
rg3er
r3get
r3gic
rgi4n
rg3ing
r5gis
r5git
r1gl
rgo4n
r3gu
rh4
4rh.
4rhal
ri3a
ria4b
ri4ag
r4ib
rib3a
ric5as
r4ice
4rici
5ricid
ri4cie
r4ico
rid5er
ri3enc
ri3ent
ri1er
ri5et
rig5an
5rigi
ril3iz
5riman
rim5i
3rimo
rim4pe
r2ina
5rina.
rin4d
rin4e
rin4g
ri1o
5riph
riph5e
ri2pl
rip5lic
r4iq
r2is
r4is.
ris4c
r3ish
ris4p
ri3ta3b
r5ited.
rit5er.
rit5ers
rit3ic
ri2tu
rit5ur
riv5el

riv3et
riv3i
r3j
r3ket
rk4le
rk4lin
r1l
rle4
r2led
r4lig
r4lis
rl5ish
r3lo4
r1m
rma5c
r2me
r3men
rm5ers
rm3ing
r4ming.
r4mio
r3mit
r4my
r4nar
r3nel
r4ner
r5net
r3ney
r5nic
r1nis4
r3nit
r3niv
rno4
r4nou
r3nu
rob3l
r2oc
ro3cr
ro4e
ro1fe
ro5fil
rok2
ro5ker
5role.
rom5ete
rom4i
rom4p
ron4al
ron4e
ro5n4is
ron4ta
1room
5root
ro3pel
rop3ic
ror3i
ro5ro
ros5per
ros4s
ro4the
ro4ty
ro4va
rov5el
rox5
r1p
r4pea

r5pent
rp5er.
r3pet
rp4h4
rp3ing
r3po
r1r4
rre4c
rre4f
r4reo
rre4st
rri4o
rri4v
rron4
rros4
rrys4
4rs2
r1sa
rsa5ti
rs4c
r2se
r3sec
rse4cr
rs5er.
rs3es
rse5v2
r1sh
r5sha
r1si
r4si4b
rson3
r1sp
r5sw
rtach4
r4tag
r3teb
rten4d
rte5o
r1ti
rt5ib
rti4d
r4tier
r3tig
rtil3i
rtil4l
r4tily
r4tist
r4tiv
r3tri
rtroph4
rt4sh
ru3a
ru3e4l
ru3en
ru4gl
ru3in
rum3pl
ru2n
runk5
run4ty
r5usc
ruti5n
rv4e
rvel4i
r3ven
rv5er.

3543

r5vest
r3vey
r3vic
rvi4v
r3vo
r1w
ry4c
5rynge
ry3t
sa2
2s1ab
5sack
sac3ri
s3act
5sai
salar4
sal4m
sa5lo
sal4t
3sanc
san4de
s1ap
sa5ta
5sa3tio
sat3u
sau4
sa5vor
5saw
4s5b
scan4t5
sca4p
scav5
s4ced
4scei
s4ces
sch2
s4cho
3s4cie
5scin4d
scle5
s4cli
scof4
4scopy
scour5a
s1cu
4s5d
4se.
se4a
seas4
sea5w
se2c3o
3sect
4s4ed
se4d4e
s5edl
se2g
seg3r
5sei
se1le
5self
5selv
4seme
se4mol
sen5at
4senc
sen4d
s5ened
sen5g
s5enin
4sentd
4sentl
sep3a3
4s1er.
s4erl
ser4o
4servo
s1e4s
se5sh
ses5t
5se5um
5sev
sev3en
sew4i
5sex
4s3f
2s3g
s2h
2sh.
sh1er
5shev
sh1in
sh3io
3ship
shiv5
sho4
sh5old
shon3
shor4
short5
4shw
si1b
s5icc
3side.
5sides
5sidi
si5diz
4signa
sil4e
4sily
2s1in
s2ina
5sine.
s3ing
1sio
5sion
sion5a
si2r
sir5a
1sis
3sitio
5siu
1siv
5siz
sk2
4ske
s3ket
sk5ine
sk5ing
s1l2
s3lat
s2le
slith5
2s1m
s3ma
small3
sman3
smel4
s5men
5smith
smol5d4
s1n4
1so
so4ce
soft3
so4lab
sol3d2
so3lic
5solv
3som
3s4on.
sona4
son4g
s4op
5sophic
s5ophiz
s5ophy
sor5c
sor5d
4sov
so5vi
2spa
5spai
spa4n
spen4d
2s5peo
2sper
s2phe
3spher
spho5
spil4
sp5ing
4spio
s4ply
s4pon
spor4
4spot
squal4l
s1r
2ss
s1sa
ssas3
s2s5c
s3sel
s5seng
s4ses.
s5set
s1si
s4sie
ssi4er
ss5ily
s4sl
ss4li
s4sn
sspend4
ss2t
ssur5a
ss5w
2st.
s2tag
s2tal
stam4i
5stand
s4ta4p
5stat.
s4ted
stern5i
s5tero
ste2w
stew5a
s3the
st2i
s4ti.
s5tia
s1tic
5stick
s4tie
s3tif
st3ing
5stir
s1tle
5stock
stom3a
5stone
s4top
3store
st4r
s4trad
5stratu
s4tray
s4trid
4stry
4st3w
s2ty
1su
su1al
su4b3
su2g3
su5is
suit3
s4ul
su2m
sum3i
su2n
su2r
4sv
sw2
4swo
s4y
4syc
3syl
syn5o
sy5rin
1ta
3ta.
2tab
ta5bles
5taboliz
4taci
ta5do
4taf4
tai5lo
ta2l
ta5la
tal5en
tal3i
4talk
tal4lis
ta5log
ta5mo
tan4de
tanta3
ta5per
ta5pl
tar4a
4tarc
4tare
ta3riz
tas4e
ta5sy
4tatic
ta4tur
taun4
tav4
2taw
tax4is
2t1b
4tc
t4ch
tch5et
4t1d
4te.
tead4i
4teat
tece4
5tect
2t1ed
te5di
1tee
teg4
te5ger
te5gi
3tel.
teli4
5tels
te2ma2
tem3at
3tenan
3tenc
3tend
4tenes
1tent
ten4tag
1teo
te4p
te5pe
ter3c
5ter3d
1teri
ter5ies
ter3is
teri5za
5ternit
ter5v
4tes.
4tess
t3ess.
teth5e
3teu
3tex
4tey
2t1f
4t1g
2th.
than4
th2e
4thea
th3eas
the5at
the3is
3thet
th5ic.
th5ica
4thil
5think
4thl
th5ode
5thodic
4thoo
thor5it
tho5riz
2ths
1tia
ti4ab
ti4ato
2ti2b
4tick
t4ico
t4ic1u
5tidi
3tien
tif2
ti5fy
2tig
5tigu
till5in
1tim
4timp
tim5ul
2t1in
t2ina
3tine.
3tini
1tio
ti5oc
tion5ee
5tiq
ti3sa
3tise
tis4m
ti5so
tis4p
5tistica
ti3tl
ti4u
1tiv
tiv4a
1tiz
ti3za
ti3zen
2tl
t5la
tlan4
3tle.
3tled
3tles.
t5let.
t5lo
4t1m
tme4
2t1n2
1to
to3b
to5crat
4todo
2tof
to2gr
to5ic
to2ma
tom4b
to3my
ton4ali
to3nat
4tono
4tony
to2ra
to3rie
tor5iz
tos2
5tour
4tout
to3war
4t1p
1tra
tra3b
tra5ch
traci4
trac4it
trac4te
tras4
tra5ven
trav5es5
tre5f
tre4m
trem5i
5tria
tri5ces
5tricia
4trics
2trim
tri4v
tro5mi
tron5i
4trony
tro5phe
tro3sp
tro3v
tru5i
trus4
4t1s2
t4sc
tsh4
t4sw
4t3t2
t4tes
t5to
ttu4
1tu
tu1a
tu3ar
tu4bi
tud2
4tue
4tuf4
5tu3i
3tum
tu4nis
2t3up.
3ture
5turi
tur3is
tur5o
tu5ry
3tus
4tv
tw4
4t1wa
twis4
4two
1ty
4tya
2tyl
type3
ty5ph
4tz
tz4e
4uab
uac4
ua5na
uan4i
uar5ant
uar2d
uar3i
uar3t
u1at
uav4
ub4e
u4bel
u3ber
u4bero
u1b4i
u4b5ing
u3ble.
u3ca
uci4b
uc4it
ucle3
u3cr
u3cu
u4cy
ud5d
ud3er
ud5est
udev4
u1dic
ud3ied
ud3ies
ud5is
u5dit
u4don
ud4si
u4du
u4ene
uens4
uen4te
uer4il
3ufa
u3fl
ugh3en
ug5in
2ui2
uil5iz
ui4n
u1ing
uir4m
uita4
uiv3
uiv4er.
u5j
4uk
u1la
ula5b
u5lati
ulch4
5ulche
ul3der
ul4e
u1len
ul4gi
ul2i
u5lia
ul3ing
ul5ish
ul4lar
ul4li4b
ul4lis
4ul3m
u1l4o
4uls
uls5es
ul1ti
ultra3
4ultu
u3lu
ul5ul
ul5v
um5ab
um4bi
um4bly
u1mi
u4m3ing
umor5o
um2p
unat4
u2ne
un4er
u1ni
un4im
u2nin
un5ish
uni3v
un3s4
un4sw
unt3ab
un4ter.
un4tes
unu4
un5y
un5z
u4ors
u5os
u1ou
u1pe
uper5s
u5pia

up3ing
u3pl
up3p
upport5
upt5ib
uptu4
u1ra
4ura.
u4rag
u4ras
ur4be
urc4
ur1d
ure5at
ur4fer
ur4fr
u3rif
uri4fic
ur1in
u3rio
u1rit
ur3iz
ur2l
url5ing.
ur4no
uros4
ur4pe
ur4pi
urs5er
ur5tes
ur3the
urti4
ur4tie
u3ru
2us
u5sad
u5san
us4ap
usc2
us3ci
use5a
u5sia
u3sic
us4lin
us1p
us5sl
us5tere
us1tr
u2su
usur4
uta4b
u3tat
4ute.
4utel
4uten
uten4i
4u1t2i
uti5liz
u3tine
ut3ing
ution5a
u4tis
5u5tiz
u4t1l
ut5of
uto5g
uto5matic
u5ton
u4tou
uts4
u3u
uu4m
u1v2
uxu3
uz4e
1va
5va.
2v1a4b
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
5vian
5vide.
5vided
4v3iden
5vides
5vidi
v3if
vi5gn
vik4
2vil
5vilit
v3i3liz
v1in
4vi4na
v2inc
vin5d
4ving
vio3l
v3io4r
vi1ou
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
wa1te
wa5ver
w1b
wea5rie
weath3
wed4n
weet3
wee5v
wel4l
w1er
west3
w3ev
whi4
wi2
wil2
will5in
win4de
win4g
wir4
3wise
with3
wiz5
w4k
wl4es
wl3in
w4no
1wo2
wom1
wo5ven
w5p
wra4
wri4
writa4
w3sh
ws4l
ws4pe
w5s4t
4wt
wy4
x1a
xac5e
x4ago
xam3
x4ap
xas5
x3c2
x1e
xe4cuto
x2ed
xer4i
xe5ro
x1h
xhi2
xhil5
xhu4
x3i
xi5a
xi5c
xi5di
x4ime
xi5miz
x3o
x4ob
x3p
xpan4d
xpecto5
xpe3d
x1t2
x3ti
x1u
xu3a
xx4
y5ac
3yar4
y5at
y1b
y1c
y2ce
yc5er
y3ch
ych4e
ycom4
ycot4
y1d
y5ee
y1er
y4erf
yes4
ye4t
y5gi
4y3h
y1i
y3la
ylla5bl
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
ys1t
ys3ta
ysur4
y3thin
yt3ic
y1w
za1
z5a2b
zar2
4zb
2ze
ze4n
ze4p
z1er
ze3ro
zet4
2z1i
z4il
z4is
5zl
4zm
1zo
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, 676–687.

[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.

[illegible] [illegible]

[illegible] [illegible]

[illegible] [illegible]