CSC 495 – Problem Set 8 – Due Tuesday, March 20

Problem 8.R1: Sparse Populations

Required Problem
Points: 50 points

Background

This is a "sparse" (but one-dimensional) version of Problem 7.R1 (Star Census). Recall that in that problem you had to quickly answer how many stars were in a particular area of the sky. In this problem, you are to consider a huge one-dimensional array of zeros and ones (ones representing stars), where indices are 60-bit numbers. Note that you would need a one million terabyte memory to actually store this entire array, so that's out of the question. The array is very sparse, with no more than a million ones located in the array. Given a description of this array you are asked to answer a series of queries asking how many 1's are in some sub-array `data[a..b]` for endpoints a and b. Other than this property, this is exactly a one-dimensional version of Problem 7.R1.

For an example with small numbers, consider the following set of indices: 123, 178, 221, 269, 270, 271, 301 (these are given in sorted order for simplicity, but indices that are given as input to this problem are not guaranteed to be sorted). The following query ranges (given by a and b range endpoints) give counts as shown in the table.

a	b	count
200	300	4
200	250	1
140	160	0
269	300	3
123	178	2

The Problem to Solve

You are give an unordered list of indices (each up to 60 bits long) of the positions that contain 1's. There will be at most one million indices, and you are guaranteed that there are no duplicates in this list. This is followed by a number of queries, where a query is made up of a pair of indices (a and b) and for each query you need to answer how many 1's are in the array between indices a and b (inclusive). There can be as many as one million queries, and you must be able to answer the full set of queries in less than 30 seconds.

Hints and Techniques

Think about the sparse array representation described in Column 10 of *Programming Pearls*, combined with techniques you used in both problem 7.R1 and problem 7.C1.

Input and Output

The input consists of a number n on a line, followed by n lines that contain the indices where 1's are found. This is followed by an integer m that reflects the number of queries, which is followed by m lines containing the endpoints of each query range. For each query, you are to output the count of the number of 1's found in that range of indices. The sample input and output below reflects the example problem given in the "Background" section.

Sample Input

```
7
271
123
269
221
270
178
301
5
200 300
200 250
140 160
269 300
123 178
```

Sample Output

```
4
1
0
3
2
```

Problem 8.R2: Parse This!

Required Problem
Points: 50 points

Background

One way to do data compression is to use a dictionary of commonly occurring words or sequences of characters, and to replace words in an input text with references to the dictionary. However, with some dictionaries and some input strings, there is more than one way to divide the string into words (or to "parse" the input). Since we're interested in compression, we obviously want the input subdivision with the fewest possible dictionary references.

For example, consider a dictionary that contains the following sequences of characters (which we will call "words" even though they're not English words): {"A", "AG", "GTA", "CA", "CAG", "T"}

Using this dictionary, we can parse the input string "AGCAGTAGTA" into dictionary words in two possible ways:

AG+CAG+T+AT+T+A (6 references)
AG+CA+GTA+GTA (4 references)

The second way is preferable, since it requires fewer references.

The Problem to Solve

For this problem, you are to write a program that reads in a dictionary and an input string, and outputs the optimal division of the input string into dictionary words (in other words, the division with the fewest possible references). For some input strings, there is no way to divide the input string up into references to dictionary words — your program should detect this case and print an error message. There will be at most 60 dictionary words, with each word being at most 10 characters, and the text to encode will be at most 100 characters. Given these constraints, your program should easily be able to find the optimal parsing in under a second. You may also assume that the input text will not contain any plus signs.

Note that in some cases multiple optimal parsings are possible. For example, with dictionary {"A", "AA"} the string "AAA" can be parsed as either "A+AA" or "AA+A" — if there are multiple solutions always output the one with the shortest possible dictionary words first (so the correct output in this case would be "A+AA").

Hints and Techniques

If you visualize words as "jumping forward" in the input text, you want a "path" with as few "jumps" as possible. Thinking of paths is a good way to look at this problem.

Input and Output

The input will consist of a line containing n, the number of words in the dictionary, followed n lines each containing a dictionary word, followed finally by the single-line input text.

Your program should output "No parsing is possible!" if there is no way to divide the input text up into dictionary words. Otherwise, your program should print out a line specifying how many words are in the best parsing, followed by a line that shows what that parsing is by repeating the input text with plus signs between the dictionary words referenced in the text. Use the exact wording and format shown in the sample outputs below.

The first sample input/output below shows the problem described in the "Background" section.

Sample Input 1

```
6
A
AG
GTA
CA
CAG
T
AGCAGTAGTA
```

Sample Output 1

```
Optimal parsing with 4 words:
AG+CA+GTA+GTA
```

Sample Input 2

```
5
A
AG
CA
CAG
C
AGGCAGCCACA
```

Sample Output 2

```
No parsing is possible!
```

Problem 8.C1: Repetitive Repetitions

Challenge Problem
Ninja Points: This challenge problem is worth up to 20 base ninja points
5 extra ninja points for the fastest solution on a large input.

Background

One technique for squeezing space is to represent repeated data using a single copy of the data, and have individual instances point to a shared representation (this only works if the data is static, of course). This is discussed in *Programming Pearls* for the problem of storing calendars for many different years since there are really only 14 distinct calendars, no matter how many years you are considering. A first step in doing this could be to analyze your data to find what information repeats. For example, if you are dealing with strings representing DNA sequences, you might ask which substrings of 5 or more characters occur more than once. For example, in the string

```
TAGACTACGATTTGCACTACTACGAAATGA
```

there are three 5-character strings that repeat: `TACGA` and `CTACG` both occur twice, and `ACTAC` occurs 3 times.

The Problem to Solve

You will be given a substring length (an integer m) and a long string — the string may be spread over multiple lines of input (in fact, no line will be longer than 1000 characters), and you should ignore all newlines and other whitespace and treat it as a single long string. An example input is provided under "Assignments" on the web site, which includes an actual DNA sequence for *E. coli* that is 4,570,938 characters spread over 65,300 lines of text. You should find all strings of length m that occur more than once in the input, and output each one with its occurrence count (sorted from most frequent to least frequent, and alphabetically within the same number of occurrences). You are guaranteed that the input string will be at most 10,000,000 characters, and m will be at most 1,000, and you should be able to process such an input in under a minute.

Hints and Techniques

There is a really easy solution to this problem for smaller inputs: Pull m-character strings out of the input one at a time, and use those to index into a map data structure to count how many times each one occurs. However, with the sizes stated in "The Problem to Solve" this could involve up to almost 10 million strings of length 1,000, which would require over 10 gigabytes to store. Therefore, you need a more efficient representation in order for everything to fit in memory (and think about how much memory it would take to store substrings of the human genome, which is over three *billion* characters long!).

Input and Output

Input will consist of an integer m (the substring length), followed by the string to scan. The string may contain newlines or other whitespace characters, which should be ignored (they are not considered part of the string). Unlike other problems we've looked at in this class, there is no count for the string length given in advance of the string: you simply read until end-of-file, and you must deal with allocation issues to store as much information as you need. Output should list all m-character strings that occur more than once along with the number of times they appeared, sorted first by the occurrence count, and then ordered alphabetically within each group of strings that has the same occurrence count. The sample input and output shown below reflects the problem stated in the "Background" section.

Sample Input

```
5
TAGACTACGATTTGCACTACTACGAAATGA
```

Sample Output

```
ACTAC 3
CTACG 2
TACGA 2
```

Problem 8.C2: Have you seen my CAT?

Challenge Problem
Ninja Points: This challenge problem is worth up to 20 base ninja points

Background

The previous two problems dealt with different parts a larger problem: finding repeated substrings and replacing those with compact references into a dictionary of strings rather than repeating the string. This can save substantial space for some types of data (although it is somewhat limited for real DNA sequences, despite our use of that in examples), but the more compact representation is more difficult to work with when processing these strings. For example, consider the following dictionary (note that this dictionary uses a common trick: setting the first entries in the dictionary to all possible individual characters, which guarantees that any string can be parsed).

Index:	0	1	2	3	4	5	6	7
String:	A	C	G	T	GAC	TAGCTA	TCA	GACTG

Using this dictionary, we can parse the string "GACATAGCTATCATCAGACTGCAT" as 4,0,5,6,6,7,1,0,3. Note that if we were to store each of the 24 original characters in a compact form, using two bits each, it would require 48 bits. Using the dictionary we were able to use 9 dictionary indices that are 3-bits each, for a total of 27 bits: a savings of 21 bits, or around 44%.

Now if we want to find the string "CAT", there are many ways it could appear. Obviously, the sequence 1,0,3 represents "CAT", but there are in fact 9 different sequences of indices that contain "CAT", as shown below:

Index Sequence	String
1,0,3	CAT
1,0,5	CATAGCTA
1,0,6	CATCA
4,0,3	GACAT
4,0,5	GACATAGCTA
4,0,6	GACATCA
6,3	TCAT
6,5	TCATAGCTA
6,6	TCATCA

Through appropriate use of finite automata (take CSC 553!), you can actually do a little preprocessing and then search for all of these index sequences simultaneously (in other words, one scan of the string is sufficient — you don't need to do 9 different searches), but you have to identify these sequences first. That's what you need to do in this problem!

The Problem to Solve

You will be given a dictionary of strings, followed by a search string. You are to find and print all sequences of indices that represent strings containing the search string. The dictionary will contain at most 1,000,000 strings, and the substring will be at most 10,000 characters long. Your output should be sorted lexicographically, and you are guarnateed that three will be no more than 10,000 sequences in the output. Your program should be able to process any such input in under a minute.

Input and Output

Input will consist of a line containing an integer n, the number of entries in the dictionary, followed by the words in the dictionary, one per line. The dictionary entries are ordered so that the first word is index 0, the second is index 1, and so on (up to index $n - 1$). The dictionary is followed by a single line containing the search string. You should output all index sequences that represent strings containing the search string, ordered lexicographically. The sample input and output below reflects the example problem described in the "Background" section.

Sample Input

```
8
A
C
G
T
GAC
TAGCTA
TCA
GACTG
CAT
```

Sample Output

```
1,0,3
1,0,5
1,0,6
4,0,3
4,0,5
4,0,6
6,3
6,5
6,6
```