Computer science problems.

About the problems. This year's problems allow you to explore many aspects of developing efficient programs, including algorithms and data representation.

What you need to do.

You may use any programming language you want for your programs, as long as its full implementation is available at no cost and with an easy installation for both a Mac and Windows (free trial versions do not qualify). Note that you are not required to chose the most efficient programming language, but to develop the most efficient program in the language of your choice.

We will be looking for the following in your submissions:

- Correct code that we can run. You need to send us all your code files, including the header files for languages like C++. If you are using standard libraries, make sure to include all "import" statements, as required by the language you are using. Make sure to send the files under the correct names, including the file extension (.java, .c, etc). Make sure that the file names do not contain any identifying information about you, such as your first or last name.

- Test data for your code that you have used (you can write it in comment or in a separate file). Make sure to test your code well – you don't want it to fail our tests!

- Code documentation and instructions. **Important: do not include your name in comments or in any file names.** If you are submitting your answers to non-code problems in a separate file, also make sure that it does not have your name in the contents or in the file name. The only place where you specify your name is the zip file with your solutions which must be of the form `yourlastname-CS-solution.zip` (replace `yourlastname` by your actual last name). **Make sure that you use zip compression, and not any other one, such as tar**. In the beginning of each file specify, in comments:

  1. Problem number(s) in the file. The file name should indicate the purpose of the file: utilities (such as input/output), a specific data structure implementation (such as a hashtable), etc.
  2. The *programming language*, including the *version* (Java 1.12, for instance), the *development framework* (such as Visual Studio) that you used, unless you were using just a plaintext editor (notepad, emacs, etc), and the *platform* (such as Windows, Mac, Linux)
  3. Instructions for running your program (how to call individual functions, pass the input (if any), etc), either in comments in your program file or as a separate file, clearly named. Your program will need get input from the user in a specified format. You need to clearly explain how to set internal parameters, if any.

4. Some of your code may be commented out if it is not used in the final run of your program. Make sure it is clear what needs to be uncommented to run code for each of the problems.
5. All of your test data.
6. If you were using sources other than the ones listed here (i.e. textbooks, online resources, etc) for ideas for your solutions, please clearly credit these contributions. This is a courtesy to work of others and a part of ethics code for scholars.
7. Make sure that you clearly specify where input files are supposed to be located, provide an example input file and an example of how the file name would be specified in the input. Use relative paths (from the top of the project or from the executable), not absolute paths. **All input files must have an extension `.in`, all output files an extension `.out`.**
8. Documentation (in comments or in a separate document) that includes the following:
   (a) Why your program always produces a correct solution.
   (b) Efficiency of your program in terms of $O(n)$, where n is the size of the input, in the worst case. Make sure to specify what kind of data constitutes the worst case.
   (c) If the worst case happens rarely, explain what the expected efficiency is (you don't need the exact computation of the expected efficiency: informal reasoning is fine, as long as it's clear and refers to the code and data patterns).
   (d) A list of programming choices and optimizations that you made to reduce the running time.

- Clear, understandable, and well-organized code. This includes:

  1. Clear separation between problems and functions; comments that help find individual problems and explain how to run the corresponding functions.
  2. Breaking down code into functions that are clearly named and described (in comments), using meaningful names for variables and function parameters. Your code should be as self-explanatory as possible. While using comments helps, naming a variable `average` is better that naming it `x` and writing a comment "x represents the average".
  3. Minimization of code repetition. Rather than using a copy-paste approach, use functions for repeated code and reuse these functions.
  4. Using well-chosen storage structures (use an array or a list instead of ten variables, for instance) and well-chosen programming constructs (use loops or recursion when you can, rather than repeated code).

Problem 1. The program input is a string of non-zero digits, for instance

```
365
```

Your goal is to partition these digits into (possibly multi-digit) numbers to maximize the combined number of prime divisors in the sequence. Repeated divisors are counted as many time as they appear.

For instance, for the example above you can partition it into the three single-digit numbers `3,6,5`, where commas indicate where the partitions are. In this case there are four prime divisors $3, 2, 3$, and 5 since 3 and 5 are prime, and 6 is the product of 2 and 3. I will be writing divisors as multisets (sets with repeated elements), so the divisors in this case are $\{2, 3, 3, 5\}$. Just like in sets, the order of elements in multisets doesn't matter, and I am writing them in increasing order.

Another way of partitioning is `36,5`. In this case 36 can be factored into $\{2, 2, 3, 3\}$, and 5 is prime, so the multiset of prime divisors for `36,5` is $\{2, 2, 3, 3, 5\}$, which is larger than the multiset of four elements in the partition `3,6,5`.

There are two more partitions:

- `3,65` with the multiset of three elements $\{3, 5, 13\}$ (since $65 = 5 \cdot 13$)
- `365` treated as the entire 3-digit number, with the multiset $\{5, 73\}$.

Thus in this case the partition `36,5` produces the largest number of prime divisors, which is 5.

Your goal is to write a program that, given a string of digits as input (on its own line, no spaces), will output a partition that produces the largest number of prime divisors, followed by the number of divisors. In the example given above the output will be

```
36,5 5
```

Note that you don't need to print out the divisors themselves (although you might want to implement this feature as a debugging option).

Some details to be mindful of:

- 1 is not a prime number.
- If more than one partition gives the maximal number of elements, printing any one of them is fine.
- Your program must work on strings of length up to 30 digits. It may run too long on some of longer strings, and we will use a timeout when running programs. However, it shouldn't crash, and should still in principle output a correct answer if it ever finishes.
- Your program must always produce a correct answer. Using heuristics, even with a high degree of certainty, is not allowed. For this reason probabilistic primality testing is not allowed.

- Your goal is to write a program that produces an answer within a reasonable amount of time on as long of an input string as possible. Think of what kinds of cases may make it too slow and what you can do about those.

- If you are using multisets as data structures, you need to implement your own. Don't use libraries, even if they exist.

In addition to your program code and instructions for how to run it, you need to submit a write-up (in comments or in a separate document) that explains the following:

1. Why your program always gives a correct answer.

2. What is the worst case efficiency of your program as $O(n)$, where n is the number of digits in the input string. Clearly explain what kind of data results in the worst case. See [1] or [2] for definition of $O(n)$.

3. If the expected (common) case when given a randomly chosen string of n non-zero digits is different from the worst case, please explain what it is and why. This part doesn't need to be completely formal, but needs to be very clear. You may want to address it in conjunction with the next item.

4. The point of the problem is optimization, so please **describe** and **justify** all decisions that you made in order to make your program run faster (these may include choices of data structures and data representation, choices of algorithms and their stopping conditions, etc.).

You can get some ideas for optimizations in [1] and/or any other algorithms book. Make sure to cite your references, including online ones.

Problem 2. The program also takes input as a string of non-zero digits. Additionally it takes a number between 1 and the multi-digit number represented by the string.

Just as in the previous question, the program considers all possible ways of partitioning this string into smaller numbers. However, your goal is to find a partition that multiplies to the given number. For instance, the input

```
365 180
```

prints the partition `36, 5`.

Note that for many combinations there is no solution. For instance, the input

```
365 185
```

doesn't have a solution. In this case your program should print

```
No solution.
```

Note that the input number is chosen between 1 and the integer represented by the string with no partitions (in this case, 365).

Your goal is not only to find the combination quickly if it exists, but also to quickly discover that there is no solution when there is none. If the input number is selected at random, there will be more *No solution* cases than those for which the solution exists. It may be that some cases for which a solution exists take a long time to run for long sequences, but you should attempt to optimize it for as large percentage of cases as possible.

As in the previous problem, please submit explanations of

1. Why your program always gives a correct answer.

2. What is the worst case efficiency of your program as $O(n)$, where n is the number of digits in the input string. Clearly explain what kind of data results in the worst case.

3. If the expected (common) case when given a randomly chosen string of n non-zero digits and a randomly (uniformly) chosen number between 1 and the multi-digit number represented by the string is different from the worst case, please explain what it is and why. This part doesn't need to be completely formal, but needs to be very clear. You may want to address it in conjunction with the next item.

4. The point of the problem is optimization, so please **describe** and **justify** all decisions that you made in order to make your program run faster (these may include choices of data structures and data representation, choices of algorithms and their stopping conditions, etc.).

Happy programming!

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[2] Michael Sipser. *Introduction to the Theory of Computation.* Course Technology, Boston, MA, third edition, 2013.