Subanagram

We wish to make a website for helping sneaky Scrabble players. A Scrabble player has a set of letters, with which they try to form words from a given dictionary. In this exercise, we can ignore all other Scrabble-rules, and only consider what we call *subanagrams*.

Let us assume that the player's letters are given as a string S, and that a word W is also given as a string. We say that W is a *subanagram* of S if W can be written using only letters from S.

Note that we do not allow the reuse of letters from S. For example, `"hole"` is a subanagram of `"ehlo"`. However, `"hello"` is *not* a subanagram of `"ehlo"`, because `"ehlo"` does not contain two `"l"`'s. Both `"hole"` and `"hello"` are subanagrams of `"hheelloo"`.

Here are some examples of subanagrams of `"aghilmort"`:

- `"algorithm"`
- `"logarithm"`
- `"alright"`
- `"right"`
- `"math"`
- `"git"`

In the following exercises, we consider different ways of finding subanagrams for a given dictionary.

(a)

ALGORITHM: IS W A SUBANAGRAM OF S?

Input: A string W and a string S
Output: Returns **true** if W is a subanagram of S, **false** otherwise

```
1 Procedure IsSubanagramOf1(W, S)
2     r ← copy of W
3     for c in S do
4         if c is in r then
5             remove an occurrence of c from r
6     return r.length = 0
```

The algorithm above checks whether a word W can be written using letters from S.

- Assume that a string works just like an array of letters.
- Copying an array is done in linear time.
- Checking whether an element is in an array is done in linear time.
- Removing an element from an array is done in linear time.
- Finding the length of an array is done in constant time.
- We let w specify the size of W and s specify the size of S.

What is the runtime complexity of the algorithm?

(b)

You will now implement an alternative to `IsSubanagramOf1` using *frequency tables*. Suppose you have a procedure `FreqTable` available, which builds a map from letters to the number of occurrences in linear time.

If F is a frequency table for the string `"abbcccdddd"`, then F.`get("a")` returns 1 and F.`get("d")` returns 4. You can assume that F.`get`(x) returns 0 for all letters that are not in the word the frequency table was built from. You can update the value in the frequency table by use F.`put`, just as with a regular map. Both `get` and `put` are in $O(1)$.

Complete the implementation of `IsSubanagramOf2` so that it has runtime complexity $O(w + s)$, where w is the size of W and s is the size of S.

ALGORITHM: IS W A SUBANAGRAM OF S?

Input: A string W and a string S
Output: Return **true** if W is a subanagram of S, **false** otherwise

```
1 Procedure IsSubanagramOf2(W, S)
2     F ← FreqTable(W)
      // Fill in
```

(c)

Strategy 1

```
1  Procedure SubanagramsOf1(D, S)
2      r ← empty list
3      for i ← 0 to d − 1 do
4          W ← D[i]
5          if IsSubanagramOf2(W, S)
            then
6              add W to r
7      return r
```

In the pseudocode above, we are given a dictionary `D` of length d, which is an array of strings. It uses `IsSubanagramOf2` from the previous exercise to find all words in the dictionary which is a subanagram of S. Finding subanagrams using this technique runs in $O(d \cdot (w + s))$, where d is the size of the dictionary, w is the size of the largest word in the dictionary, and s is the size of S.

Strategy 2

```
1  Procedure Build(D)
2      M ← empty map
3      for i ← 0 to d − 1 do
4          sortedword ← Sort(D[i])
5          if M.get(sortedword) is empty then
6              M.put(sortedword, empty list)
7          add D[i] to M.get(ws)
8      return M
9  Procedure SubanagramsOf2(M, S)
10     r ← empty list
11     for x in sorted substrings of S do
12         append M.get(x) to r
13     return r
```

In the pseudocode above, we create a hashmap where each key k being a string of letters sorted alphabetically, and the value is a list of anagrams of k found in the dictionary. To find all subanagrams in the dictionary, we must look up all sorted substrings of S in the hashmap. To find subanagrams using this technique runs in $0(2^s)$, i.e. exponential time with respect to the size of S.

Discuss advantages/disadvantages of strategy 1 and strategy 2 with respect to runtime.

(d)

Now that our website can find subanagrams of a string S, all that remains is to present the results to the user. Assume the subanagrams returned by `Subanagrams0f2` are given as an array, sorted *alphabetically*. We want the subanagrams to be sorted by *length*, and that words of the same length are sorted alphabetically, as in the following example:

Input	**Output**
`"algorithm"`	`"algorithm"`
`"alright"`	`"logarithm"`
`"git"`	`"alright"`
`"logarithm"`	`"right"`
`"math"`	`"math"`
`"right"`	`"git"`

Fill in the pseudocode below. Your procedure should have as low runtime complexity as possible.

ALGORITHM: SORTING OF SUBANAGRAMS

Input: An array A of size n containing strings
Output: An array with the same n strings sorted by length where words of the same length are alphabetized

```
1 Procedure SortSubanagrams(A)
    // Fill in
```