Swan Ponds

You have an integer matrix representing a plot of land, where the value at that location represents the height above sea level. A value of zero indicates water. A pond is a region of water connected vertically, horizontally, or diagonally. The size of the pond is the total number of connected water cells. Write a method to compute the sizes of all ponds in the matrix.

```
Input: 0 2 1 0
       0 1 0 1
       1 1 0 1
       0 1 0 1

Output: 2, 4, 1 (in any order)
```

(Green book page 184)

IP Misaddress

Greenbook Club has come into possession of an old IPv4 networking system made by Swansea Uni researchers in the 80s (don't ask us how, John Tucker has his ways). The only problem is, some bright spark decided to break the spec, so nothing stored looks like IP addresses!

For example, Swansea's IP is normally 137.44.1.20, however this system has it stored as `"zp.Zt.B.U"`! That's right, it's a mnemonic IP system! Completely out of spec, but Tucker wants us to convert these old "Swansea IP" addresses into normal IPv4. Handily, there's an old comment left in a user manual.

```
/*                                                                          *\
 *                    Swansea IPv4 - ** SIP Address **                      *
 *--------------------------------------------------------------------------*
 * This system, unique to Swansea, is an extension on the ARPANET IPv4 way of *
 * addressing networks. It provides a way to use plaintext addresses for use  *
 * on C based ASCII operating systems (ie. UNIX System V, because this is all *
 * we can afford). Thanks to its incomprehensible nature and hence similarity *
 * to Welsh, this extension proved popular with faculty and students. Well at *
 * least the ones outside the Welsh Studies department. To convert from IPv4: *
 *                                                                            *
 * For each byte, starting with the leading byte, produce printable ASCII in  *
 * the range 97-122 (a-z). If the leading byte is less than 97, then we use   *
 * an uppercase display in the range 65-90 (A-Z), however we always interpret *
 * them as if in lowercase, but with the knowledge that we have offset this.  *
 * This means a byte < 97 is first increased by 97 (upcast to avoid rollover) *
 * but is displayed in uppercase (A-Z) so that we can recognise this state.   *
 * For a byte > 122, we now have two phases. We would like to print out chars *
 * that "sum" to the byte (with following char as 0-24 for a-z), however this *
```

```
 * would mean that we get a large string of 'z' characters in many cases. For *
 * this reason, we first convert these 'z' characters into decimal digits in  *
 * the range 48-53 (0-5) such that, if there is more than one 'z', we produce *
 * the decimal representation of how many were required. Hence, a byte=146 is *
 * the lowest byte with this behaviour, and so contributes the char '2'. If   *
 * the byte is lower than 97, we consider the first Z offset, so for a  byte  *
 * = 49, which is expanded up to 146 for a-z representation, however we force *
 * the first character to be represented as a 'Z' since this is needed so we  *
 * can disambiguate the process. Following this would be a normal 'z', as no  *
 * further characters are required. Once we reach the byte = 74 and above, we *
 * resume this compression scheme and produce the char '2' or greater. Due to *
 * the limitations of a single byte, this should not exceed '5'. Hence 255 is *
 * converted to "5i". This is repeated for each byte, which now do not need a *
 * '.' to be included between them, since this is unambigious, however it can *
 * be included to produce "phonetic groupings" or such to ease readability.   *
 *                                                                            *
 * Example: 137.44.1.20 = "zp.Zt.B.U", 255.0.0.1 = "5iAAB"
 *                                                                            *
 * To convert back to IPv4, we simply invert this process.                    *
 * This is left as an exercise for the reader / unwitting CS-115 student...   *
 *                                                                            *
 *----------------------------------------------------------------------------*
 *      COPYRIGHT - 1985- under the Swansea Fictitious Copyright License      *
 *///                                                                      \\\*
```

As an extension, some IP addresses in our modern table aren't in the normal "quad" format (like 137.44.1.20), but instead are other compatible (in-spec) formats! So, in order to make matching easier (because who on earth canonises anything) Tucker wants you to produce each address in a range of different formats.

(Inspired by this fun trick & `inet_aton(3)` shenanigans.)

Playing Tag

Hierarchical directories are out! Down with the caste (file)system!

Welcome to the 21st century, we have #tags! And your task is to wrangle them!

See, some very clever clogs came up with an AI that can create tags from videos, identifying both general themes from the videos and multiple highlights within. The only problem is that none of them thought about how best to use them! Well, that's not the half of it, they didn't even realise it was outputting tags that are actually the same thing, just in a different language or phrasing!

So, you have to come up with a way to filter a list of videos based on a tag, knowing that the tag you were provided is "incomplete"! For example, if a

person searched for "puppy" videos, then you'd also need to return "pup" videos that didn't get the "puppy" tag assigned (because the AI obviously isn't perfect).

Rather than deal with a massive list of tags, for the purposes of demonstrating your method and testing it, you can assume they are numeric IDs (so perhaps 5 is "puppy" and 9 is "pup"). Each video can also be identified numerically, so we can return a list of videos like `[43,27,90,72]`. Each video can have any number of tags (because variable content, lengths, and numbers of highlights, etc.) and may have multiple "equivalent" tags (i.e., video 90 above has both tags 5 and 9) so account for that, we don't want to see the same video pop up twice (or more)!

As a final thing, don't worry about things being sorted or whatever, your task is simply to retrieve these videos such that we can sort them in any number of ways (relevance, age, length, etc).

As an extension, consider what happens if this is a "live algorithm", as in it is expected to run for multiple queries, rather than one-off. In other words, without knowing anything about the possible queries, can you pre-organise this to optimise lookup? Factor in space considerations to your thinking if you can. (We typically call such "setup" as an "amortised cost", as it's spread over the subsequent queries. For example, dynamic arrays have O(1) amortised resizing!)

(Inspired by Google Hash Code 2019!)