Cryptography Motivation

As a motivating example, suppose we wanted to write a program to break a simple cryptographic system, such as a *Caesar Cipher* or a *Vigenére Cipher*. The first of these, the Caesar Cipher, named after the ancient Roman emperor Julias Caesar, encrypts a message by adding a fixed shift (*e.g.*, +3) to each letter. For example, encrypting *Hello* would yield *Khoor*, as *K* is 3 letters after *H*, *h* is 3 letters after *e,* and so on. The Vigenére cipher applies the same concept of shifting each letter, but uses a key with more than one amount to shift by. For example, we might have a key of (3,5,7,2,9), in which case we would add 3 to the first, sixth, eleventh, sixteenth, and so on *etc.* letters of the message; we would add 5 to the second, seventh, twelfth, seventeenth, and so on letters; . We will note that for a long time, Vigenére was regarded as unbreakable, and that many novice programmers think of variants of it when trying to devise their own encryption schemes. However, it is easily broken.

Breaking both of these cipher relies on *frequency counting* —determining how often each letter appears in the encrypted message—and then using the fact that the frequency distribution of letters in natural language is highly uneven. In English, *e* is by far the most common letter, while letters such as *q, x, z* are quite uncommon. If we were trying to break a Caesar Cipher, once we have frequency counted the message, we can guess that the most frequently occurring letter is *e*, and then try to decrypt the message. If all characters (not just letters) are encrypted, we would guess that the most frequently occurring symbol is space, and try to decrypt the message. If the decrypted message does not make sense, we can try again with the second most common letter/symbol. Although it might take a few tries, we would typically succeed on the first attempt (as long as the message is long enough to give reasonable frequency counts).

If we actually wanted to break such messages, we would want to write a program to do it—specifically, an algorithm which would take an encrypted message as input, and output the decrypted message (if we are worried about needing multiple tries, we could have it output the decrypted message for the top few possibilities). Writing this program requires a few pieces. First, we need to frequency count the encrypted message, and then we need to find out which letter's count is the largest. Since we do not know how to work with strings yet (our message would be a string—and we cannot learn about strings until we know about arrays), let us just look at the problem of finding the largest of the frequency counts—of which there would be 26, one for each letter. You *could* write a function to implement this algorithm right now, however, it would be horrendously ugly:

```
19

20

21
```

```
22
23
24
25
16
17
18
14
15
11
12
13
8
9
10
6
7
3
4
5
1
2
  …  //many lines of code omitted
  if (numZs > bestCount) {
    bestLetter = 'z';
    bestCount = numZs;
  }
  return bestLetter;
```

```
}
    bestLetter = 'd';
    bestCount = numDs;
  }
  }
  if (numDs > bestCount) {
  if (numCs > bestCount) {
    bestLetter = 'c';
    bestCount = numCs;
    bestLetter = 'b';
    bestCount = numBs;
  }
  unsigned bestCount = numAs;
  if (numBs > bestCount) {
                      … // many parameters omitted
                      unsigned numXs, unsigned numYs, unsigned numZs) {
  char bestLetter = 'a';
char highestFrequency(unsigned numAs, unsigned numBs, unsigned numCs,
                      unsigned numDs, unsigned numEs, unsigned numFs,
```

We sketch this code out, but leave out large parts of it—in particular, there are 17 other parameters (never write a function that takes 26 parameters!), and 84 lines of code (also, never write a function that is 100+ lines long) for the cases for e–y which are missing. This approach would work, but it is tedious and error prone to write (how likely are you to miss changing something as you copy/paste 24 cases?). Matters would get even worse if we were dealing with 1000 items instead of 26—could you imagine trying to write this function to find the max of 1000 things? Looking at it, we can see that we are repeating almost the same thing over and over—suggesting there should be a way to generalize this code into an algorithm where the program can do the repetitive work, instead of you doing it. It seems like there has to be a better way, and fortunately there is—we can use an array!