2010 SEEKING RECOGNITION

I've seen this before …

In this question we will be building a system to recognize files. The system will have simple functionality: we need to be able to *store* a file, and given a file which we may or may not have seen before, we need to be able to *study* a file. This operation should return `STORED` if that file has been stored at some point in the past, or `NEVER STORED` otherwise.

Our system will never need to 'un-store' a file which has already been stored.

We never need to recognize files which are merely similar to a file we have seen before — only files which are exactly the same, bit for bit. We can think of a file as being a long string of `0`s and `1`s, and hence as a (very) large binary number. We will imagine that our computer language can manipulate numbers this large as easily as any other kind of number.

```
initialise()
{
  N ← 0
  mylist ← []
}
```

figure 1

One simple way of storing the files is to keep a simple list. Figures 1 - 3 give pseudocode for this approach.

Figure 1 shows pseudocode for *initialise*, which is run before we use our system for the first time (← is how we assign values, and [] indicates an empty list). Our system will use *N* to keep track of the number of files that have been seen and keep the list of files in *mylist*. Figure 2 shows the *store* function, which first resizes the list to make space for one more element before adding it to the list. Finally figure 3 shows the *study* function, which loops through the list until it either finds *file* or runs out of list.

```
store(file)
{
  N ← N + 1
  resize(mylist, N)
  mylist[N] ← file
}
```

figure 2

```
study(file)
{
  for i ← 1 upto N
  {
    if mylist[i] ≡ file
      return STORED
  }
  return NEVER STORED
}
```

figure 3

It might be better to keep the list *mylist* in sorted order; so that the element stored in location *mylist[1]* is always smaller (when considered as a number) than the element in *mylist[2]*, and so on.

Question 1

Modify the pseudocode of the *store* routine to keep the list sorted. You should assume *mylist* already contains a sorted list of files and that we are just trying to insert a single new file. Is the new *store* function faster or slower than the old one?

The *study* operation can now be made faster. Suppose we are studying *file*. We examine the element in the middle of the list (if there is no element exactly in the middle, we choose one as close to the middle as possible). Let's say this element is *mylist[k]*. If this matches *file*, we can report that the file had been `STORED`. If not, then either *file* < *mylist[k]* or *file* > *mylist[k]*.

Question 2

Suppose that in fact *file* < *mylist[k]*. Why, if *file* is stored at all, must it be somewhere amongst *mylist[1]* … *mylist[k-1]*? Briefly justify your answer and indicate what can be concluded if *file* > *mylist[k]*.

If we know that *file* can only lie amongst *mylist[1] … mylist[k-1]*, we then examine the element in the middle of the range *mylist[1] … mylist[k-1]*. It might be *file*, in which case we can report that the file had been `STORED`. Otherwise, we can narrow down further our knowledge of where in the list *file* must be, if it is stored at all.

Question 3

Using these ideas, briefly explain how we can continue this approach to eventually either find *file* or show that it was `NEVER STORED`.

Question 4

Suppose that the list contains files represented by the following numbers: 1,5,10,13,17,20,22. (We have used unrealistically small numbers for convenience.) Which elements in the list would need to be examined if we were trying to see if the file 7 had been stored?

Question 5

Suppose that there are 127 files stored in the list, and that we are studying a file which, in fact, has not been stored. What is the largest number of elements in the list we might have to examine? What is the smallest? What if there were 128 files stored in the list?

One disadvantage of this approach is that, if the files which are stored are very large, it requires a great deal of memory, since we must remember a copy of every file we store in our list. In fact, any approach which has a 100% accuracy rate in determining whether a given file has been `STORED` or not must take up a great deal of memory.

Question 6

Imagine that we know that 10 files have been stored, but we do not know what they are. Explain how, just by calling the *study* function with certain inputs, we can (in principle) reconstruct what those 10 files were. (Note that your approach need not be practical.)

Bloom Filters

If we are willing to accept a small chance that we will report that we have `STORED` something which in fact we have not, then we can use much less space. We will imagine that we have a function called *random-oracle*, which inputs a *file* and a number *M*. For each possible *file*, the output of *random-oracle* will be a random number between 1 and *M*, but the output is always *consistent*: whenever we use the same input (*file* and *M)* we will always get the same output. The outputs of *random-oracle* for different *file*s are completely independent of one another.

Under this approach, we choose a number *M* about 100 times bigger than the number of files we plan to ever store. As an example, we will imagine that we plan to store about 10000 items, and we take *M* to be one million.

Figures 4 and 5 give pseudocode for the *store* and *initialise* functions. The *initialise* function create a list of *M* elements all of which are set to `0`. The *store* function no longer stores the entire *file*, but uses the *random-oracle* to modify the list.

Notice that the list only ever stores `1`s and `0`s, so each entry in the list need only occupy one bit of computer memory.

```
initialise()
{
  mylist ← []
  resize(mylist, M)
  for i ← 1 upto M
    mylist[i] ← 0
}
```

figure 4

```
store(file)
{
  r ← random-oracle(file, M)
  mylist[r] ← 1
}
```

figure 5

Question 7

Write a *study* routine for this system. Remember, it is acceptable for your routine to have a small chance of reporting that something has been stored when it in fact has not. On the other hand, if something has been stored, you should always correctly say so.

Question 8

It is possible to show that, after storing 10000 files, we would expect to have about 9950 `1`s stored in the list. If we did have this number of `1`s, what is the chance that, if we were asked to study a file which had never been stored, we would mistakenly report that it had been?

Question 9

For our system with *M* taken to be 1000000, roughly how much memory will our system take up? (You need only account for the data, not the storage space required for the code.)

It is possible to dramatically decrease the chance of inaccurately reporting that a file has been stored without using any more memory. To do this, we imagine that *random-oracle* has been modified so that it no longer returns 1 random number from 1 to *M*, but an array of 70 random numbers, each one independent of the others. As before, this is done consistently — you always get the same 70 numbers when you give it the same input.

```
store(file)
{
  r ← random-oracle(file, M)
  for i ← 1 upto 70
    mylist[r[i]] ← 1
}
```

figure 5

The initialise function is unchanged, and the new *store* function is shown in figure 5. Note that we update *mylist* using each of the numbers returned by *random-oracle*.

Question 10

Write pseudocode for a *study* routine to match this *store* routine. Remember, it is acceptable for your routine to occasionally report that something has been stored when it in fact has not, but you should try to minimize the chance of this happening. As before, if something has been stored, you should always correctly say so.

It is possible to show that, after storing 10000 files, we would expect to have about half of the memory spots filled with `1`s (actually 503415 of them but who's counting?).

Question 11

If we did have *exactly half* the memory filled with `1`s what would be the chance of reporting the file `STORED` when in fact it had not been? You should express your answer roughly in the form ***1 in 2^k*** for some whole number *k*, as well as expressing it in the form ***1 in 10^r*** for some whole number *r*.

Finally, let's suppose that Romulus and Remus have both been using the scheme above to store and study (possibly different) files. They both use the same *random-oracle* function. They each have lists (*romulusList* and *remusList*), which have `1`s and `0`s in them according to the scheme above.

Question 12

They wish to join forces, and make a single list, *mylist*, such that if one uses the *study* function with this new list, one will be told that something has been `STORED` if *either* Romulus *or* Remus had seen it before. Give an algorithm to compute the list *mylist* given *romulusList* and *remusList*, and explain why it works.

How about if something should only be reported as `STORED` if *both* Romulus *and* Remus had seen it before?