Choosing a Good Hashing Function

In several of our examples we used the hashing function $h(k) = k \mod m$. This is a popular choice in a lot of introductory discussions of hashing, partly because it is easy to understand and implement. Unfortunately it is not necessarily a particularly good hashing function.

But in order to justify that statement I need to offer some criteria by which we can judge the quality of a hashing function. Alas, there is no universally agreed-upon list of such criteria. Hashing functions are used in a wide variety of applications and properties that might be essential in one application are undesirable in others.

I'm going to a mention a few properties of good hashing functions that seem to be generally accepted by most of the sources I have found. For simplicity I am assuming that the keys are integers – we will discuss using Strings as keys a bit later.

A good hashing function should …

- incorporate all the information in the key, giving equal significance to all digits (unless some digits of the key have restricted ranges)

- map the keys uniformly to the addresses [0..m-1] - that is, approximately equal numbers of keys should map to each address in the table

- be fairly fast to compute – the whole purpose of using a hash table is to have quick access to the data. If computing the hashing function for each key takes a long time then we may lose this advantage

- be discontinuous – keys that are "very close" should not always map to addresses that are "very close". The idea here is that if there happens to be a cluster among the keys – for example, a group of keys that only differ in their last digit – this should not create a cluster in the table T. This property is definitely *not* universally considered desirable – some authors go so far as to recommend the opposite: they suggest that similar keys should always map to similar addresses. It is not clear to me what the supposed advantage of this might be.

So let's consider $h(k) = k \mod m$ in the light of these criteria.

Suppose we are working with decimal arithmetic and $m = 10$. Then it is clear that we fail the first criterion because we are simply discarding all but the last digit of each key. Similarly if $m = 2^x$ then we are discarding all but the last x bits of the binary representation of k. This is not very good (although we looked at some computational benefits when we discussed double hashing).

On the other hand, suppose we let $m = 11$. If our keys are integers, we can represent an arbitrary key as $k = a_t a_{t-1} a_{t-2} \ldots a_1 a_0$ where the a_i's are the digits of k.

So $k = 10^t a_t + 10^{t-1} a_{t-1} + 10^{t-2} a_{t-2} + \cdots + 10 a_1 + a_0$

which is equivalent to saying

$$k = (11-1)^t a_t + (11-1)^{t-1} a_{t-1} + (11-1)^{t-2} a_{t-2} + \cdots + (11-1) a_1 + a_0$$

which means that

$$k \mod 11 = ((-1)^t a_t + (-1)^{t-1} a_{t-1} + \cdots - a_1 + a_0) \mod 11$$

(Why!? Because if we multiply out $(11-1)^i a_i$ each term contains some power of 11 except for the last one (which is just $(-1)^i a_i$) so when we apply the $\mod 11$ all the terms except the last one just disappear.)

And since all powers of -1 are either -1 or 1, this simply resolves to an expression in which each digit of k is either added or subtracted … so every digit of the key plays an equal role in determining the final value.

This is kind of cool – just by changing m from 10 to 11 we can improve the performance of this hashing function. But it's still not very good … it fails the discontinuity test. Consecutive key values will map onto consecutive table addresses. It does well on the other two criteria – if the keys are uniformly distributed in the key space, they will be uniformly distributed across the addresses in T (assuming m = |T|) … and computing $k \mod 11$ can be done in $O(\log k)$ time.

For interest, you might wonder if we can pull the same trick and show that $k \mod m$ will be kind of ok for any $m > 10$ by writing

$$k = 10^t a_t + 10^{t-1} a_{t-1} + 10^{t-2} a_{t-2} + \cdots + 10a_1 + a_0$$

as

$$k = (m-x)^t a_t + (m-x)^{t-1} a_{t-1} + (m-x)^{t-2} a_{t-2} + \cdots + (m-x)a_1 + a_0$$

where $x = m - 10$

This will give

$$k \mod m = ((-x)^t a_t + (-x)^{t-1} a_{t-1} + \cdots + -xa_1 + a_0) \mod m$$

which looks good – each digit of k is playing a role in the hash value. But there's still a problem. An example will show what it is.

Suppose $m = 15$ … so $x = 5$. The equation above becomes

$$k \mod 15 = ((-5)^t a_t + (-5)^{t-1} a_{t-1} + \cdots + -5a_1 + a_0) \mod 15$$

But what happens if $a_i = 3$ or 6 or 9 for some $i \geq 1$? Then $(-5)^i a_i$ is a multiple of 15, so it disappears when we apply the $\mod 15$ operation. The result is that a $3, 6$ or 9 digit in k cannot contribute anything to the hash value of k. This is clearly not good – and it's not immediately obvious how we could have recognized this problem with $m = 15$ without going through the analysis.

As an exercise, work out the details of this problem when $m = 24$

So while there *are* values of m (such as 11) where each decimal digit of the key contributes to the hash value, there are also many values of m where this doesn't happen. It's worth noting that if $m > 10$ and m is prime this is not an issue because none of the expanded terms can be multiples of m – meaning they won't disappear when we apply $\mod m$. This just reinforces our earlier observation that letting m be prime is often a good idea … not just for the purposes of collision resolution, but also (as we now see) for the design of hashing functions.

Let's look at another very popular (and very simple) hashing function:

$h(k)$ = (sum of the digits of k) $\bmod\ m$

This passes the first test (all digits contribute equally) and the third test (fast computation).

To see that this fails the second test, suppose the keys are 10-digit telephone numbers. There are 10^{10} possible keys (with a tiny fraction of them, such as 000-000-0000, ruled out because nobody gets that phone number). The maximum possible value of $h(k)$ in this example is 90 (the sum of 10 digits) so every probing sequence will start with an address in the range [0 .. 90]. If we are storing even just a few hundred keys, very soon every single insertion will start with one or more collisions even if the table is otherwise almost empty. This is not good!

This hashing function also fails the fourth criterion – keys that differ by 1 in any position will hash to consecutive addresses.

Fortunately the clustering problem with the "sum the digits" method is relatively easy to fix. All we need to do is introduce a multiplier that will increase the range of the hash values of the keys. Ideally we want to do this in such a way that all m addresses in T are equally likely to be the starting point of a probe sequence. That's hard, but at least we can ensure that the range of hash values goes as high as m. Let d be the number of digits in the keys, and let c be any constant such that $9c^{d-1} > m$. In our previous example, $d = 10$. Suppose $m = 1024$. We can let $c = 2$, since $9 * 2^9 > 1024$. A more formal explanation of how to choose c is given by the following equivalent statements

$$9c^{d-1} > m$$

$$\Leftrightarrow c^{d-1} > \frac{m}{9}$$

$$\Leftrightarrow c > \left(\frac{m}{9}\right)^{\frac{1}{(d-1)}}$$

Once we have chosen c our hash function becomes

```
h(k):
       sum = 0
       for each digit x of k:
           sum = sum*c + x
       return sum % m        // treating m as a global variable!
```

Note that this is really just Horner's Rule applied to the polynomial

$$h(k) = (x_{d-1} * c^{d-1} + x_{d-2} * c^{d-2} + \cdots + x_1 * c + x_0) \mod m$$

where the x_i values are the digits of the key.

This will give hash values that cover the full range from 0 to m-1 ... but it still may not be ideal in that the hash values may not be completely evenly distributed. For example, if the maximum sum value is only slightly larger than m, the first part of the table will be "hit" more often. Similarly, if the maximum result is quite a bit less than m then the last part of the table will not be "hit" at all. In an ideal world, the sum values would be evenly distributed.

The selection of an optimal value for c is outside the scope of this course but I recommend studying it if you are interested. For our purposes, it is worth noting that many people choose a prime for c when using this hash method, on the grounds that a prime value of c is less likely to lead to clustering among the keys. Other people make c a power of 2 (for example, 128 seems to be popular). One reason for this choice is that if we can perform bit level operations on integers, multiplying by a power of 2 is just a left-shift.

Here's another hashing function that is sometimes proposed:

$$h(k) = k^2 \mod m$$

This violates at least two of the criteria. First, consider the last digit of the key. We can make a table of the full relationship between the last digit of k and the last digit of k^2

Last digit of k	Last digit of k^2
0	0
1	1
2	4
3	9
4	6
5	5
6	6
7	9
8	4
9	1

It's pretty obvious that even if the keys have uniformly distributed final digits, the final digits of the squares will not be uniformly distributed (for example, the last digit of the square of the key will never be 3 or 7 or 8).

Now suppose our keys are 5 digit numbers. Consider the square of 12345:

```
    12345
  x 12345
---------
    61725
   49380
  37035
 24690
12345
---------
152399025
```

When we look at this result, we can see that its last digit is completely determined by the last digit of the key, and the first digit is determined by the first digit of the key (with the possibility of a carry from the second column). Similarly the second-last digit is determined only by the last two digits of the key, and the second digit is determined mostly by the first two digits of the key (again, with the possibility of a carry from the previous column).

So the digits of the key do not all contribute equally to the hash value. Fixing this problem leads to a very well-known hashing function called the **mid-square method**. As you might guess from the name, this involves squaring k and taking only the middle digits of the square (where "middle" needs to be carefully defined).

If we want to give equal weight to all digits of the key, it makes sense to throw away the first digits and the last digits of the square since these are based on just a few digits of the key. But here we have to compromise. Based on the argument just given, the only digit of the square that is based on all 5 digits of the key is the middle one where we see

```
6
9
0
9
5
```

But if we throw away all the other digits of the square except the one at the foot of this column, we end up with a one digit hash value (in this case, `9`). Since we started with 10^5 possible keys, a hashing function that only produces 10 possible hash values is not very useful.

So we include some of the digits on either side of this central digit. It's a trade-off: the more digits we include, the greater the range of values we get … but also the more bias we create by giving greater importance to the beginning and ending digits of the key. Here we can use information about the expected size of our data set to guide our decision. For example, if we know that m will be under 1000, then we can pull 3 digits out of the middle of k^2 … this gives hash values in the range [0 … 999] and it involves all digits of the key more or less equally.

The hashing function **for this example** would look something like this:

```
mid_square(k):
    s = k*k             # s will have up to 10 digits
                        # (note that if k is very small, most digits
                        # of s will be 0)

    x = s / 1000        # this gets rid of the last three digits
    a = x % 1000        # this keeps just the three digits we want
    return a
```

Once again we can see that if we do all operations at the bit level, extracting the middle bits of the square can be done very quickly using shifts etc.

We need to return to the point raised in the algorithm, regarding what happens when k is small. If the keys are 5-digit integers drawn uniformly from the range [0 ... 99999] then some of them will be so small that when we apply the mid-square method we end up with 0. For example, if k = 12 (ie. 00012) then s = 0000000144 and the mid-square method gives a = 0 ... as it will for any other very small value of k. This is not much of a problem, but it illustrates that when the key is very small, all (or almost all) of its information is lost when we discard the right-hand digits of the square.

Thus the mid-square method is most useful when we can be sure that few of the keys are very small.

The mid-square method (with suitable precautions regarding small key values) satisfies the four criteria pretty well.

A very popular hashing method is called the **multiplication method.** The basic idea is to multiply each key by a fixed value , throw away the "high end" of that, then multiply the result by m and throw away the "low end" of that – which makes it a bit like the mid-square method. There are different forms of the multiplication method. Here we will use a multiplier that is between 0 and 1 – this makes it very easy to discard the parts of the numbers that we don't want to keep:

```
# choose a value V in the range (0...1)
# the same V is used for hashing all keys

h(k):
    x = fractional part of V*k
    return floor(m*x)
```

For example, let $m = 128$, let $V = 0.12397$ and let $k = 4982$

$V * k = 617.61854$, of which the fractional part is 0.61854

$128 * 0.61854 = 79.17312$

$floor(79.17312) = 79$

So $h(4982) = 79$

Note that we don't have to add a $\mod m$ operation to the end of this calculation. Since $0 \leq x < 1$, $floor(m * x)$ must be in the range $[0...m-1]$

The choice of V is obviously important. Choosing V = 0.5 would be very bad since x would always be either 0 or 0.5. Donald Knuth, who writes with a great deal of authority and is usually right about such things, says that a very good value for V is $\frac{\sqrt{5}-1}{2}$

This works out to approximately 0.61803398875 . Interestingly, the Golden Ratio is 1.61803398875… In other words, Knuth's magic hashing number is the fractional part of the Golden Ratio … and it is also exactly equal to the inverse of the Golden Ratio. Math is cool.

Once again it is worth pointing out that if m is a power of 2 then computing $m * x$ is just a left-shift, which can be done very quickly at the hardware level.

The multiplication method is hugely popular.

The last hashing function we will look at is called **tabulation hashing**. Its earliest form was invented by Albert Zobrist in 1969. Zobrist used it as a method to keep track of chess-game positions.

Tabulation hashing is usually described in terms of bit-wise operations. For simplicity we will assume that all our keys are exactly b bits in length, and that $b = r * t$ for some integers r and t – we will talk about how to choose r and t a bit further along . We think of each key as consisting of r blocks, each block consisting of t bits.

For example, we can think of 40-bit keys as consisting of 5 blocks of 8 bits.

Let m be a power of 2 … that is, let $m = 2^p$. Thus we want to map keys in the range $[0 \ldots 2^b - 1]$ to addresses in the range $[0 \ldots 2^p - 1]$

We create a table (array) Z with 2^t rows and r columns, and we populate each cell of the table with a randomly chosen integer in the range $[0 \ldots 2^p - 1]$ (that is, a randomly chosen bitstring of length p). The table is created once and then used for all calculations of $h(k)$.

The hashing function is now defined by

```
tabulation(k):
     # treat k as a string of b bits
     h = bitstring consisting of p 0's   # or p 1's
     #   that is, h = 000000...0   or h = 111111...1
     for (i = 0; i < r; i++)
          x = k[i*t .. (i+1)*t-1]
          # x is the next block of t bits in k
          row = (int) x
          #row is an integer in the range [0..2^t - 1]
          temp = Z[row][i] # pick one of the bitstrings in
                           # column i of the array
          h = h XOR temp   # we exclusive-or h and temp, giving
                           # an updated value for h
     return (int) h        # interpret h as an integer in the range
                           # [0..2^p - 1]
```

The basic idea is that we use each block of t bits to choose a particular bitstring from one of the columns of Z, then we exclusive-or all the chosen bitstrings together to get the final hash value.

An example may help: Let $b = 40$, and choose $r = 5$ and $t = 8$. Also, let $m = 2^{10}$. The array Z will have $2^8 = 256$ rows and 5 columns. Each cell will be occupied by a randomly-chosen bitstring of length 10.

Row	Block 0	Block 1	Block 2	Block 3	Block 4
0	Table is filled	With	Random	Bitstrings	Of length 10
1					
2					
...					
109	1011101001				
...					
167		0111001111			
...					
255					

Suppose k = 01101101 10100111 11001100 01010110 11100011

h is initialized to 0000000000

The first block is 01101101, which is the integer 109. Suppose Z[109][0] = 1011101011

We XOR this with h, giving 1011101011 as the new value of h

The second block is 10100111, which is the integer 167. Suppose Z[167][1] = 0111001111

We XOR this with h, giving 1100100100 as the new value of h

We continue, using the third block to choose a bitstring from column [2] of Z, and then
using the fourth block to choose a bitstring from column [3], and
the last block to choose a bitstring from column [4]

Suppose the final value of h is 0111101101, which is the integer 493

Thus h(01101101 10100111 11001100 01010110 11100011) = 493

This is a very good hashing function, according to our criteria – assuming the random bitstrings in the table are uniformly distributed, each bit of the key has equal weight and the hash values cover the entire address space uniformly. The computation is fast – just type conversions, table look-up and XOR. Finally, changing any bit of the key will potentially change every bit of the result – it is highly discontinuous.

The choice of r and t determine the size of the table Z (it has 2^t rows and r columns). Clearly there can be options: if $b = 24$, we could use a table with 1 column and 2^{24} rows, 2 columns and 2^{12} rows, 3 columns and 2^8 rows, etc all the way up to 24 columns and 2 rows. It seems that $t = 8$ is a popular choice in the literature since then each block is exactly one byte. If b is a prime or some other awkward number we may need to pad each key with some extra bits at the end to bring it up to a length that is easy to break into blocks.

You may be wondering why we populate the Z table with randomly chosen bitstrings … why not work out an optimal set of bitstrings to maximize the quality of the hashing function? One answer is that the expected performance of randomly chosen bitstrings can be shown to be very good. Since hashing is about good expected performance, this is all we need.

Going Forward

There are hundreds (if not thousands) of other hashing functions in the literature and on the web – some simple and some complex – and more are created every year. Hashing functions and other related functions have become essential tools in such diverse fields as cryptography, document verification and pattern matching. I encourage you to explore these topics.

Hashing Functions for Strings

A very frequently encountered situation is where the keys are strings of characters (personal names, for example, or significant words in a document).

Our approach will be to look at algorithms that convert strings to integers. Once we have done that we can apply any of the hashing functions we have already seen (or any of the limitless set of hashing functions that we did not look at).

All of these algorithms work on the individual characters of the string to be hashed.

In some languages **characters** and **integers** are not distinguished. This means we can simply do arithmetic directly on the characters. In other languages we use a function that is typically called **ord()** to find an unique integer associated with each character. You may want to read about the history of the ASCII sequence, the UNICODE sequence, and the ancient EBCDIC sequence.

Kernighan and Ritchie offer the following simple algorithm

```
h(s):          # s is a string
     a = 0
     for x in s:
          a += ord(x)
     return a
```

It's simple … and terrible. It has all the flaws of the "sum the digits" hashing function for integers that we looked at earlier.

However, we can easily fix it the same way as we fixed that one: by introducing a constant multiplier c and using this algorithm

```
h(s):          # s is a string
     a = 0
     for x in s:
          a = a*c + ord(x)
     return a
```

A popular and widely cited version of this is credited to Dan Bernstein. It is reported to give excellent results

```
djb2(s):       # s is a string
     a = 5381
     for x in s:
          a = a*33 + x
     return a
```

The reasons for starting a at 5381 instead of 0, and for choosing 33 as the value of c are complex – you can read about this here:

http://stackoverflow.com/questions/1579721/why-are-5381-and-33-so-important-in-the-djb2-algorithm

but there is one simple thing we can note about 33. Since 33 = 32 +1, we can rewrite

```
a = a*33 + x
```

as

```
a = a*32 + a + x
```

and as we have observed so many times, multiplying by a power of 2 (in this case we are using $32 = 2^5$) is just a left shift of the bits. So we don't actually have to do any multiplication.

A sophisticated consideration when hashing strings that are English words of a fixed length (such as you are asked to do in Assignment 4) is that the letters of the alphabet are not distributed equally in common words, and not distributed equally in certain positions in the words. For example the letter "e" shows up far more frequently than the letter "z" - a hashing function might be designed to downplay the significance of "e" and increase the significance of "z" in a word. Similarly the distribution of letters that occur as the first letter of words is extremely non-uniform: comparatively few words start with "j", and hardly any start with "x", whereas there are many thousands of words that start with "t". A function that converts words into (hopefully unique) integers might emphasize positions, letters (and even combinations of letters) that distinguish words from each other.