Choosing a Good Hash Function

In several of our examples we used \( h(k) = k \mod m \). This may not be a good choice. A well-designed hash function should try to incorporate all the information in the key - when we use something as simple as \( h(k) = k \mod m \) we are potentially throwing away most of the information. For example if \( m = 100 \), the key values 16 and 34324392316 will both hash to the same address.

Another issue to consider is that ideally, every address in the table should be equally likely to be the first address in a probe sequence. As an example of a hash function that uses all the information in the key but fails on this second issue is this … which I have also used in some of the examples given previously!

\[
h(k) = \text{sum of the digits of } k
\]

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 probe sequence will start with an address in the range \([0 \ldots 90]\). This will result in an enormous number of collisions, no matter how large we make the hash table. Bad!

So why did I use these two hash functions in our examples if I am now saying they are bad? Well, they are easy to understand, so we could focus on the collision-resolution problem. But now we need to talk about creating better hashing functions.

Fortunately the 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 \( \ell \) be the number of digits in the keys, and let \( c \) be any constant such that \( 9 \times c^{\ell - 1} > m \). In our previous example, \( \ell = 10 \). Suppose \( m = 1024 \). We can let \( c = 2 \), since \( 9 \times 2^9 > 1024 \). Now our hash function becomes

\[
h(k):
\begin{align*}
\text{sum} &= 0 \\
\text{for each digit } x \text{ of } k: \\
\text{sum} &= \text{sum} \times c + x \\
\text{return sum}
\end{align*}
\]
Note that this is really just Horner’s Rule applied to the polynomial

\[ h(k) = x_{d-1} \cdot c^{d-1} + x_{d-2} \cdot c^{d-2} + \cdots + x_1 \cdot c + x_0 \]

where the \( x_i \) values are the digits of the key. Horner’s Rule was covered back in Week 1 of this course … don’t you love it when the pieces connect?

When we take this returned value \( \mod m \) we will get hash values that cover the full range from 0 to \( m-1 \) … but it still may not be ideal in that if 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. 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 such as 37 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 … ie they choose \( c = 2^k \) (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.

Another popular hash function is the **mid-square method**. As you might guess from the name, this involves squaring \( k \) and taking the middle digits of the square (where "middle" needs to be carefully defined).

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

\[
\begin{array}{c}
12345 \\
\times \quad 12345 \\
\hline
61725 \\
49380 \\
37035 \\
24690 \\
12345 \\
\hline
152399025
\end{array}
\]

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 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. 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 hash 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 $m$ (the table size) will be under 1000, then we can pull 3 digits out of the middle of
$k^2$ … this gives hash values in the range $[0 \ldots 999]$ and it involves all digits of the key more or
less equally.

The hash function for this example would look something like this:

```python
mid_square(k):
    s = k*k  # s will have up to 10 digits
    # (note that if k is small, the first digits
    # of s will all 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 $[0 \ldots 99999]$ then many
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 small value of $k$. 
Thus the mid-square method is most useful when we can be sure that none of the keys have leading zeros.

One popular hashing method is called the **multiplication method**:  

choose a value $V$ in the range $(0...1)$  

$h(k)$:  

$x = \text{fractional part of } V \times k$  

return $\text{floor}(m \times x)$  

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

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

$128 \times 0.61854 = 79.17312$  

$\text{floor}(79.17312) = 79$  

So $h(4982) = 79$  

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. Math is cool.

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

There are hundreds (if not thousands) of hashing functions in the literature and on the web — some simple and some complex. I encourage you to explore them.
Hashing Functions for Strings

So far we have assumed that all keys are integers. Obviously that is not necessarily true – keys can be arbitrary objects. All that is really required is that we can test two keys for equality. They don’t even need to form a partially or fully ordered set.

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 number 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

```python
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

```python
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:


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.