The World's Smallest Hash Table

2023-03-04

This December I once again did the Advent of Code, in Rust. If you are interested, my solutions are on Github. I wanted to highlight one particular solution to the day 2 problem as it is both optimized completely beyond the point of reason yet contains a useful technique. For simplicity we're only going to do part 1 of the day 2 problem here, but the exact same techniques apply to part 2.

We're going to start off slow, but stick around because at the end you should have an idea what on earth this function is doing, how it works, how to make one and why it's the world's smallest hash table:

```
pub fn phf_shift(x: u32) -> u8 {
    let shift = x.wrapping_mul(0xa463293e) >> 27;
    ((0x824a1847u32 >> shift) & 0b11111) as u8
}
```

The problem

We receive a file where each line contains `A`, `B`, or `C`, followed by a space, followed by `X`, `Y`, or `Z`. These are to be understood as choices in a game of rock-paper-scissors as such:

```
A = X = Rock
B = Y = Paper
C = Z = Scissors
```

The first letter (`A`/`B`/`C`) indicates the choice of our opponent, the second letter (`X`/`Y`/`Z`) indicates our choice. We then compute a score, which has two components:

1. If we picked Rock we get 1 point, if we picked Paper we get 2 points, and 3 points if we picked Scissors.

2. If we lose we gain 0 points, if we draw we gain 3 points, if we win we get 6 points.

As an example, if our input file looks as such:

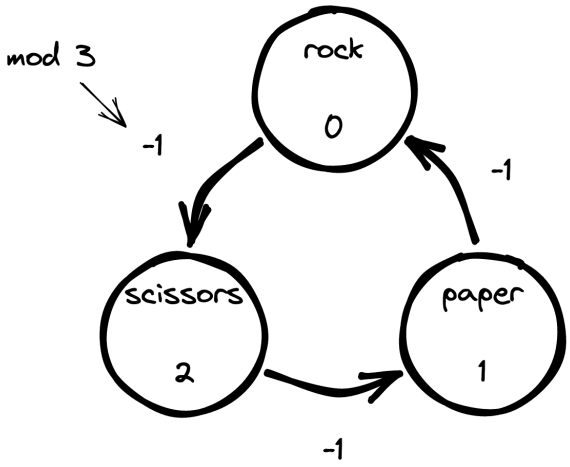
```
A Y
B X
C Z
```

Our total score would be `(2 + 6) + (1 + 0) + (3 + 3) = 15`.

An elegant solution

A sane solution would verify that indeed our input lines have the format `[ABC] [XYZ]`, before extracting those two letters. After converting these letters to integers `0`, `1`, `2` by subtracting either the ASCII code for `'A'` or `'X'` respectively we can immediately calculate the first component of our score as `1 + ours`.

The second component is more involved, but can be elegantly solved using modular arithmetic. Note that if Rock $= 0$, Paper $= 1$, Scissor $= 2$ then we always have that choice $k + 1 \bmod 3$ beats k. Alternatively, k beats $k - 1$, modulo 3:



If we divide the number of points that Advent of Code expects for a loss/draw/win by three we find that a loss is 0, a draw is 1 and a win is 2 points. From these observations we can derive the following modular equivalence

$$1 + \text{ours} - \text{theirs} \equiv \text{points} \pmod 3.$$

To see that it is correct, note that if we drew, `ours - theirs` is zero and we correctly get one point. If we add one to `ours` we change from a draw to a win, and `points` becomes congruent with 2 as desired. Symmetrically, if we add one to `theirs` we change from a draw to a loss, and `points` once again becomes congruent with 0 as desired.

Translated into code we find that our total score is

```
1 + ours + 3 * ((1 + ours + (3 - theirs)) % 3)
```

Instead of `ours - theirs` we do `ours + (3 - theirs)` because Rust's remainder operator can unfortunately return negative remainders for positive divisors. One could use `rem_euclid` instead, but I feel bad for recommending it as that one is unfortunately defined for negative divisors. I should write a blog post about this…

A general solution

We found a neat closed form, but if we were even slightly less fortunate it might not have existed. A more general method for solving similar problems would be nice. In this particular instance that is possible. There are only $3 \times 3 = 9$ input pairs, so we can simply hardcode the answer for each situation:

```
let answers = HashMap::from([
    ("A X", 4),
    ("A Y", 8),
    ("A Z", 3),
    ("B X", 1),
    ("B Y", 5),
    ("B Z", 9),
    ("C X", 7),
    ("C Y", 2),
    ("C Z", 6),
]);
```

Now we can simply get our answer using `answers[input]`. This might feel as a bit of a non-answer, but it is a legitimate technique. We have a mapping of inputs to outputs, and sometimes the simplest or fastest (in either programmer time or execution time) solution is to write it out explicitly and completely rather than compute the answer at runtime with an algorithm.

Perfect Hash Functions

The above solution works fine, but it pays a cost for its genericity. It uses a full-fledged string hash algorithm, and lookups involve the full codepath for hash table lookups (most notably hash collision resolution).

We can drop the genericity for a significant boost in speed if we were to use a perfect hash function. A perfect hash function is a specially constructed hash function on some set S of values such that each value in the set maps to a different hash output, without collisions. It is important to note that we only care about its behavior for inputs in the set S, with a complete disregard for other inputs.

A *minimal* perfect hash function is one that also maps the inputs to a dense range of integers $[0, 1, \ldots, |S| - 1]$. This can be very useful because you can then directly use the hash function output to index a

lookup table. This effectively creates a hash table that maps set S to anything you want. However, strict minimality is not necessary for this as long as you are okay with wasting some of the space in your lookup table.

There are fully generic methods for constructing (minimal) perfect hash functions, such as the *"Hash, displace and compress"* algorithm by Belazzougui et. al., which is implemented in the phf crate. However, they tend to use lookup tables to construct the hash itself. For small inputs where speed and size is absolutely critical I've had good success **just trying stuff**. This might sound vague—because it is—so let me walk you through some examples.

Reading the input

> *This is where we leave the realm of reasonable solutions for the sake of education and fun. For simplicity we're not going to handle things such as Windows-style newlines (`\r\n`) or invalid inputs.*

As a bit of a hack we can note that each line of our input from the Advent of Code consists of exactly four bytes. One letter for our opponent's choice, a space, our choice, and a newline byte. So we can simply read our input as a `u32`, which simplifies the hash construction immensely instead of dealing with strings.

For example, consulting the ASCII table we find that `A` has ASCII code `0x41`, space maps to `0x20`, `X` has code `0x58` and the newline symbol has code `0x0a` so the input `"A X\n"` can also simply be viewed as the integer `0x0a582041` if you are on a little-endian machine. If you are confused why `0x41` is in the last position remember that we humans write numbers with the least significant digit on the right as a convention.

Note that on a big-endian machine the order of bytes in a `u32` is flipped, so reading those four bytes into an integer would result in the value `0x4120580a`. Calling `u32::from_le_bytes` converts four bytes assumed to be little-endian to the native integer representation by swapping the bytes on a big-endian machine and doing nothing on a little-endian machine. Almost all modern CPUs are little-endian however, so it's generally a good idea to write your code such that the little-endian path is fast and the big-endian path involves a conversion step, if a conversion step can not be avoided.

Doing this for all inputs gives us the following desired integer → integer mapping:

```
Input       LE u32      Answer
------------------------------
 A X       0xa582041       4
 A Y       0xa592041       8
 A Z       0xa5a2041       3
 B X       0xa582042       1
 B Y       0xa592042       5
 B Z       0xa5a2042       9
 C X       0xa582043       7
 C Y       0xa592043       2
 C Z       0xa5a2043       6
```

Example constructions

When I said I just try stuff, I mean it. Let's load our mapping into Python and write a test:

```
inputs =  [0xa582041, 0xa592041, 0xa5a2041, 0xa582042,
           0xa592042, 0xa5a2042, 0xa582043, 0xa592043,
0xa5a2043]
answers = [4, 8, 3, 1, 5, 9, 7, 2, 6]

def is_phf(h, inputs):
    return len({h(x) for x in inputs}) == len(inputs)
```

There are nine inputs, so perhaps we get lucky and get a minimal perfect hash function right away:

```
>>> [x % 9 for x in inputs]
[0, 7, 5, 1, 8, 6, 2, 0, 7]
```

Alas, there are collisions. What if we don't have to be absolutely minimal?

```
>>> next(m for m in range(9, 2**32)
...      if is_phf(lambda x: x % m, inputs))
12
>>> [x % 12 for x in inputs]
[9, 1, 5, 10, 2, 6, 11, 3, 7]
```

That's not too bad! Only three elements of wasted space. We can make our first perfect hash table by placing the answers in the correct spots:

```
def make_lut(h, inputs, answers):
    assert is_phf(h, inputs)
    lut = [0] * (1 + max(h(x) for x in inputs))
    for (x, ans) in zip(inputs, answers):
        lut[h(x)] = ans
    return lut
```

```
>>> make_lut(lambda x: x % 12, inputs, answers)
[0, 8, 5, 2, 0, 3, 9, 6, 0, 4, 1, 7]
```

Giving the simple mapping:

```
const LUT: [u8; 12] = [0, 8, 5, 2, 0, 3, 9, 6, 0, 4, 1, 7];

pub fn answer(x: u32) -> u8 {
    LUT[(x % 12) as usize]
}
```

Compressing the table

We stopped here on the first modulus that works, which is honestly fine in this case because only three bytes of wasted space is pretty good. But what if we didn't get so lucky? We have to keep looking. Even though modulo m has $[0, m)$ as its *codomain*, when applied to our set of inputs its *image* might span a smaller subset. Let's inspect some:

```
>>> [(m, max(x % m for x in inputs))
...  for m in range(1, 30)
...  if is_phf(lambda x: x % m, inputs)]
[(12, 11), (13, 11), (19, 18), (20, 19), (21, 17), (23, 22),
 (24, 19), (25, 23), (26, 21), (27, 25), (28, 19), (29, 16)]
```

Unfortunately but also logically, there is an upwards trend of the maximum index as you increase the modulus. But 13 also seems promising, let's take a look:

```
>>> [x % 13 for x in inputs]
[3, 6, 9, 4, 7, 10, 5, 8, 11]
>>> make_lut(lambda x: x % 13, inputs, answers)
[0, 0, 0, 4, 1, 7, 8, 5, 2, 3, 9, 6]
```

Well, well, well, aren't we lucky? The first three indices are unused, so we can shift all the others back and get a minimal perfect hash function!

> *Ironically this one would almost surely perform worse than the previous one because Rust has to do a bounds check now whereas the previous version is infallible, and it has an extra subtraction.*

```
const LUT: [u8; 9] = [4, 1, 7, 8, 5, 2, 3, 9, 6];

pub fn answer(x: u32) -> u8 {
    LUT[(x % 13 - 3) as usize]
}
```

In my experience with creating a bunch of similar mappings in the past, **you'd be surprised to see how often you get lucky**, as long as your mapping isn't too large. As you add more 'things to try' to your toolbox, you also have more opportunities of getting lucky.

Fixing near-misses
Another thing to try is fixing near-misses. For example, let's take another look at our original naive attempt:

```
>>> [x % 9 for x in inputs]
[0, 7, 5, 1, 8, 6, 2, 0, 7]
```

Only the last two inputs give a collision. So a rather naive but possible way to resolve these collisions is to move those to a different index:

```
>>> [x % 9 + 3*(x == 0xa592043) - 3*(x == 0xa5a2043) for x in
inputs]
[0, 7, 5, 1, 8, 6, 2, 3, 4]
```

Oh look, we got slightly lucky again: both are using the constant 3, which can be factored out. It can be quite addictive to try out various permutations of operations and tweaks to find these (minimal) perfect hash functions using as few operations as possible.

An interlude: integer division

So far we've just been using the modulo operator to reduce our input domain to a much smaller one. However, integer division/modulo is rather slow on most processors. If we take a look at Agner Fog's instruction tables we see that the 32-bit `DIV` instruction has a latency of 9-12 cycles on AMD Zen3 and 12 cycles on Intel Ice Lake. However, we don't need a fully generic division instruction, since our divisor is constant here. Let's take a quick look at what the compiler does for mod 13:

```rust
pub fn mod13(x: u32) -> u32 {
    x % 13
}
```

```
example::mod13:
        mov     eax, edi
        mov     ecx, edi
        imul    rcx, rcx, 1321528399
        shr     rcx, 34
        lea     edx, [rcx + 2*rcx]
        lea     ecx, [rcx + 4*rdx]
        sub     eax, ecx
        ret
```

It translates the modulo operation into a multiplication with some shifts / adds / subtractions instead. To see how that works let's first consider the most magical part: the multiplication by 1321528399 followed by the right shift of 34. That magical constant is actually $\lceil 2^{34}/13 \rceil$ which means it's computing

$$q = \left\lfloor \frac{x \cdot \lceil 2^{34}/13 \rceil}{2^{34}} \right\rfloor = \lfloor x/13 \rfloor.$$

To prove that is in fact correct we note that $2^{34} + 3$ is divisible by 13 allowing us to split the division in the correct result plus an error term:

$$\frac{x \cdot \lceil 2^{34}/13 \rceil}{2^{34}} = \frac{x \cdot (2^{34} + 3)/13}{2^{34}} = x/13 + \frac{3x}{13 \cdot 2^{34}}.$$

Then we inspect the error term and substitute $x = 2^{32}$ as an upper bound to see it never affects the result after flooring:

$$\frac{3x}{13 \cdot 2^{34}} \leq \frac{3 \cdot 2^{32}}{13 \cdot 2^{34}} \leq \frac{3}{13 \cdot 4} < 1/13.$$

For more context and references I would suggest *"Integer division by constants: optimal bounds"* by Lemire et. al.

After computing $q = \lfloor x/13 \rfloor$ it then computes the actual modulo we want as $m = x - 13q$ using the identity

$$x \bmod m = x - \lfloor x/m \rfloor \cdot m.$$

It avoids the use of another (relatively) expensive integer multiplication by using the `lea` instruction which can compute `a + k*b`, where `k` can be a constant 1, 2, 4, or 8. This is how it computes $13q$:

> *The `LEA` instruction was originally intended for array index computations, because `arr[i]` is found at address `arr_start + sizeof(T)*i`, and `sizeof(T)` is very often a small power of two.*

```
Instruction                 Translation     Effect
lea     edx, [rcx + 2*rcx]  t := q + 2*q    t = 3q
lea     ecx, [rcx + 4*rdx]  o := q + 4*t    o = (q + 4*3q) =
13q
```

Bit mixing

We have seen that choosing different moduli works, and that compilers implement fixed-divisor modulo using multiplication. It is time to cut out the middleman and go straight to the good stuff: integer multiplication. We can get a better understanding of what integer multiplication actually does by multiplying two integers in binary using the schoolbook method:

```
4242 = 0b1000010010010
4871 = 0b1001100100111 = 2^0 + 2^1 + 2^2 + 2^5 + 2^8 + 2^9 +
2^12

          Binary                     Decimal
               1000010010010   |   4242 * 2^0
              1000010010010    |   4242 * 2^1
             1000010010010     |   4242 * 2^2
            1000010010010      |   4242 * 2^5
         1000010010010         |   4242 * 2^8
        1000010010010          |   4242 * 2^9
   1000010010010               |   4242 * 2^12
--------------------------------|---------------- +
   10011101101001001111111110   |   20662782
```

There is a beautiful property here we can take advantage of: all of the upper bits of the product $x \cdot c$ for some constant c depend on most of the bits of x. That is, for good choices of the constants c and s, `c*x >> s` will give you a result that is wildly different even for small differences in x. It is a strong *bit mixer*.

Hash functions like bit mixing functions, because they want to be unpredictable. A good measure of unpredictability is found in the avalanche effect. For a true random oracle changing one bit in the input should flip all bits in the output with 50% probability. Thus having all your output bits depend on the input is a good property for a hash function, as a random oracle is the ideal hash function.

So, let's just **try something**. We'll stick with using modulo 2^k for maximum speed (as those can be computed with a binary AND instead of needing multiplication), and try to find constants c and s that work. We want our codomain to have size $2^4 = 16$ since that's the smallest power of two bigger than 9. We'll use a $32 \times 32 \to 32$ bit multiply since we only need 4 bits of output, and the top 4 bits of the multiplication will depend sufficiently on most of the bits of the

input. By doing a right-shift of 28 on a `u32` result we also get our mod 2^4 for free.

> *If we needed more than four bits of output, or we couldn't find a constant that works, I would try a $32 \times 32 \rightarrow 64$ bit multiply as this gives you more output bits to work with.*

```
import random
random.seed(42)

def h(x, c):
    m = (x * c) % 2**32
    return m >> 28

while True:
    c = random.randrange(2**32)
    if is_phf(lambda x: h(x, c), inputs):
        print(hex(c))
        break
```

It's always a bit exciting to hit enter when doing a random search for a magic constant, not knowing if you'll get an answer or not. In this case it instantly printed `0x46685257`. Since it was so fast there are likely many solutions, so we can definitely be a bit greedier and see if we can get closer to a minimal perfect hash function:

```
best = float('inf')
while best >= len(inputs):
    c = random.randrange(2**32)
    max_idx = max(h(x, c) for x in inputs)
    if max_idx < best and is_phf(lambda x: h(x, c), inputs):
        print(max_idx, hex(c))
        best = max_idx
```

This quickly iterated through a couple of solutions before finding a constant that gives a minimal perfect hash function, `0xedc72f12`:

```
>>> [h(x, 0xedc72f12) for x in inputs]
[2, 5, 8, 1, 4, 7, 0, 3, 6]
>>> make_lut(lambda x: h(x, 0xedc72f12), inputs, answers)
[7, 1, 4, 2, 5, 8, 6, 9, 3]
```

Ironically, if we want the optimal performance in safe Rust, we still need to zero-pad the array to 16 elements so we can never go out-of-bounds. But if you are **absolutely certain** there are no inputs other than the specified inputs, and you wanted optimal speed, you could reduce your memory usage to 9 bytes with unsafe Rust. Sticking with the safe code option we'll get:

```
const LUT: [u8; 16] = [7, 1, 4, 2, 5, 8, 6, 9, 3,
                       0, 0, 0, 0, 0, 0, 0];

pub fn phf_lut(x: u32) -> u8 {
    LUT[(x.wrapping_mul(0xedc72f12) >> 28) as usize]
}
```

Inspecting the assembly code using the Compiler Explorer it is incredibly tight now:

```
example::phf_lut:
        imul    eax, dword ptr [rdi], -305713390
        shr     rax, 28
        lea     rcx, [rip + .L__unnamed_1]
        movzx   eax, byte ptr [rax + rcx]
        ret

.L__unnamed_1:
        .asciz
"\007\001\004\002\005\b\006\t\003\000\000\000\000\000\000"
```

The World's Smallest Hash Table

You thought 9 bytes was the world's smallest hash table? We're only just getting started! You see, it is actually possible to have a small lookup table without accessing memory, by storing it in the code.

> *Code ultimately has to be stored in memory as well, but it saves an indirection.*

A particularly effective method for storing a small lookup table with small elements is to store it as a constant, indexed using shifts. For example, the lookup table `[1, 42, 17, 26][i]` could also be written as such:

```
(0b110100100001101010000001 >> 6*i) & 0b111111
```

Each individual value fits in 6 bits, and we can easily fit $4 \times 6 = 24$ bits in a `u32`. In isolation this might not make sense over a normal lookup table, but it can be combined with perfect hashing, and can be vectorized as well.

Unfortunately we have 9 values that each require 5 bits, which doesn't fit in a `u32`... or does it? You see, by co-designing the lookup table with the perfect hash function we could theoretically *overlap* the end of the bitstring of one value with the start of another if we directly use the hash function output as the shift amount.

Update on 2023-03-05: As tinix0 rightfully points out on reddit, our values only require 4 bits, not 5. I've made things unnecessarily

harder for myself by effectively prepending a zero bit to each value. That said, you would still need overlapping for fitting $4 \times 9 = 36$ bits in a `u32`.

> *We could also just use a `u64` to store the data, but that's boring and we're trying to create the smallest possible hash table here.*

We are thus looking for two 32-bit constants `c` and `d` such that

```
(d >> (x.wrapping_mul(c) >> 27)) & 0b11111 == answer(x)
```

Note that the magic shift is now 32 - 5 = 27 because we want 5 bits of output to feed into the second shift, as $2^5 = 32$.

Luckily we don't have to actually increase the search space, as we can construct `d` from `c` by just placing the answer bits in the indicated shift positions. Doing this we can also find out whether `c` is valid or not by detecting conflicts in whether a bit should be 0 or 1 for different inputs. Will we be lucky?

```
def build_bit_lut(h, w, inputs, answers):
    zeros = ones = 0

    for x, a in zip(inputs, answers):
        shift = h(x)
        if zeros & (a << shift) or ones & (~a % 2**w << shift):
            return None  # Conflicting bits.
        zeros |= ~a % 2**w << shift
        ones |= a << shift

    return ones

def h(x, c):
    m = (x * c) % 2**32
    return m >> 27

random.seed(42)
while True:
    c = random.randrange(2**32)
    lut = build_bit_lut(lambda x: h(x, c), 5, inputs, answers)
    if lut is not None and lut < 2**32:
        print(hex(c), hex(lut))
        break
```

It takes a second or two, but we found a solution!

```rust
pub fn phf_shift(x: u32) -> u8 {
    let shift = x.wrapping_mul(0xa463293e) >> 27;
    ((0x824a1847u32 >> shift) & 0b11111) as u8
}
```

```
example::phf_shift:
        imul    ecx, dword ptr [rdi], -1537005250
        shr     ecx, 27
        mov     eax, -2109073337
        shr     eax, cl
        and     al, 31
        ret
```

We have managed to replace a fully-fledged hash table with one that is so small that it consists of 6 (vectorizable) assembly instructions without any further data.

Conclusion

Wew, that was a wild ride. Was it worth it? Let's compare four hash-based versions on how long they take to process ten million lines of random input and sum all answers.

1. `hashmap_str` processes the lines properly as newline delimited strings, as in the general solution.
2. `hashmap_u32` still uses a hashmap, but reads the lines and does lookups using `u32`s like the perfect hash functions do.
3. `phf_lut` is the earlier defined function that feeds a perfect hash function into a lookup table.
4. `phf_shift` is our world's smallest hash function.

The complete test code can be found here. On my 2021 Apple M1 Macbook Pro I get the following results with `cargo run --release` on Rust 1.67.1:

Algorithm	Time
`hashmap_str`	262.83 ms
`hashmap_u32`	81.33 ms
`phf_lut`	2.97 ms
`phf_shift`	1.41 ms

So not only is it the smallest, it's also the fastest, beating the original string-based `HashMap` solution by over 180 times. The reason `phf_shift` is two times faster than `phf_lut` on this machine is because it can be fully vectorized by the compiler whereas `phf_lut` needs to do a lookup in memory which is either impossible or

relatively slow to do in vectorized code, depending on which SIMD instructions you have available.

Your results may vary, and you might need `RUSTFLAGS='-C target-cpu=native'` for `phf_shift` to autovectorize.

← Prev post Archive Next post →