R: Numerical Algorithms

Signal Data Science

things to cover:

- profvis

Now that you're acquainted with the basics of R's functional programming toolkit and have a strong grasp of the most important aspects of R's internals, we'll wrap up our R curriculum with some materials on a variety of useful and interesting numerical algorithms.

Numerical optimization

Gradient descent

Backpropagation

Stochastic gradient descent

The Newton–Raphson method

The Nelder–Mead method

Fast modular exponentiation

Before we can implement more complex algorithms, we'll need a fast implementation of modular exponentiation, consisting of the task of calculating $a^b \bmod c$, *i.e.*, the remainder of dividing a^b by c. In addition to being intrinsically useful, modular exponentiation through repeated squaring (which is the end goal of this section) is a common programming question in interviews.

- Write a function `pow(a, b, c)` that calculates $a^b \bmod c$. Begin with a naive implementation that simply evaluates the calculation directly. Verify that $6^{17} \bmod 7 = 6$ and that $50^{67} \bmod 39 = 2$.

- To improve the runtime of `pow()`, start at 1 and repeatedly multiply an intermediate result by a, calculating the answer mod c each time, until the bth power of a is reached. Implement this as `pow2()`.
- Using the `tictoc` package, quantify the resulting improvement in runtime. How does runtime improve as a or c increase in size? Is the runtime improvement merely a constant-factor scaling change (is the new runtime a constant multiple of the previous runtimes)?

In order to make our algorithm even faster, we'll want to write a short utility function:

- Write a function `decompose(n)` which takes as input an integer `n` and returns a vector of integers such that when you calculate 2 to the power of each element of the result and take the sum of those powers of 2, you obtain n. (*Hint:* First, calculate all powers of 2 less than or equal to n. After that, iteratively subtract off the highest power from n, keeping track of *which* power of 2 it was, until you get to 0.)

Now, we can implement a quite rapid algorithm for modular exponentiation with the trick of repeated squaring:

- You can improve the runtime of `pow()` further by decomposing b into a sum of powers of 2, starting with a and repeatedly squaring modulo c (to calculate $a^1, a^2, a^4, a^8, \ldots \mod c$), and then forming the final answer as a *product* of those intermediate calculations. (For example, for $6^{17} \mod 7$, you are essentially calculating $17 = 2^0 + 2^4$ and $6^{17} \mod 7 = 6^{2^0} \cdot 6^{2^4} \mod 7$.) Using `decompose(n)`, implement this improvement as `pow3()`, making sure to calculate every intermediate result modulo c. Verify that `pow3()` is faster than `pow2()`.

Random number generation

Random number generators are not truly random (unless you use quantum techniques!) and are in fact pseudorandom, meaning that their output only *approximates* true randomness. A pseudorandom number generator (pRNG) can take a starting point, known as a *seed*, as input; a pRNG, given the same seed twice, will produce the exact same output in the exact same order both times. R uses inversion transform sampling by default to generate random numbers.

First, we will

Afterward, we will undertake the implementation of a xorshift pRNG, one of the simplest and fastest classes of pRNGs which work by repeatedly taking the bitwise XOR of a number with bit-shifted versions of itself. The speed of xorshift pRNGs results from the fact that the numerical operations involved are directly implemented by the CPU. (Regrettably, they do fail certain statistical tests for randomness because they are fundamentally based on linear recurrences.)

Linear congruential generators

Bitwise operations in R

In order to write a xorshift pRNG, we first need to implement bitwise operations in R.

Instead of operating with bits directly, we'll store binary representations of numbers as strings and operate on those, which will be quite slow but will better illustrate how bit shifts work. R does have native bitwise operation functions, but we'll implement our own versions for pedagogical purposes![1]

Before we can even think of operating on bitwise representations of integers, we need functions which allow us to convert between decimal and binary representations of integer! The binary representation of a number encodes it as sums of powers of 2; for example, the binary number "100101" is equal to $2^5 + 2^2 + 2^0$, because (counting from the right and starting at 0) the 0th, 2nd, and 5th positions in "100101" are 1s. Representations of integers as sums of powers of 2 are *unique*, meaning that no two numbers have the same binary representation.

- Write a function `to_binary(n)` which takes an integer `n` and returns its binary representation in a string with no leading zeroes (*e.g.*, "10100" instead of "0010100").
- Write a function `to_decimal(b)` which takes a binary representation `b` and returns the corresponding decimal integer. To handle large integers, return

The bitwise XOR operation takes two binary numbers of equal length and outputs another number of the same length, where the ith position in the output is 1 if the ith positions in the two input numbers are different and 0 if they are the same. For example, $0101 \oplus 0011 = 0110$ and $0010 \oplus 1010 = 1000$.

- Implement bitwise XOR as `bitwise_xor(a, b)`. If the inputs are of different lengths, remember to pad the shorter binary number with zeroes on the left.

A logical left shift of k bits can be thought of as discarding the leftmost k digits of a binary number and appending k zeroes to the right end. Similarly, a logical right shift of k bits discards the k rightmost digits and appends k zeroes to the

[1] The behavior of base R's logical bit shift functions is a little annoying because they can end up changing the sign of a number; the numerical value is invariant modulo 2^{32} but having to correct for a shift of 2^{32} is too much complication for this present assignment.

left end. (If k is equal to or greater than the length of the binary number, then the entire number is placed with zeroes.)

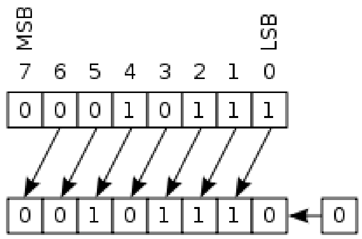

Figure 1: Illustration of a logical left shift by 1 bit.

These operations are called *shifts* because of how they are carried out in the CPU register.

- Implement left and right logical shifts of k bits as `left_shift(b, k)` and `right_shift(b, k)`.

Implementing a xorshift pRNG

Now, we can implement an algorithm to generate a random positive 32-bit integer. In order for logical left shifting to work properly, our binary numbers must be long enough to encode 32 bits of information:

- Write a function `to_binary_len(n, k)` which converts an integer `n` to a binary representation and then pads it with 0s on the left until the length of the string is equal to `k`.

Finally, we are ready to implement a simple xorshift algorithm. It will take as input 4 *seed* values x, y, z, and w which determine its initial state. In the following, let $x \ll n$ represent x left logical shifted by n bits, let $x \gg n$ represent x right logical shifted by n bits, and let $\oplus$ represent the bitwise XOR operation. The algorithm is as follows:

1. Set $t = x$.
2. Set $t = t \oplus (t \ll 11)$.
3. Set $t = t \oplus (t \gg 8)$.
4. Set $x = y$, $y = z$, and $z = w$.
5. Set $w = w \oplus (w \gg 19)$.
6. Set $w = w \oplus t$.
7. Return w.

At this point, you have everything you need to write a custom implementation of a xorshift pRNG!

- Fill in the following code template for a `xorshift()` function:

  ```
  xorshift = function(x, y, z, w) {
    # Convert x, y, z, w to 32-bit binary representations.
    function() {
      # Implement the xorshift algorithm, using <<- for
      # assignment to x, y, z, w.

      # Call return() here on the output.
    }
  }
  ```

  `xorshift()` will return a xorshift pRNG seeded with the specified values which can then be repeatedly called to generate random values, *e.g.*, `r = xorshift(0, 3, 93, 59); r();`. Verify that with $(x, y, z, w) = (1, 2, 3, 4)$ as the seed, the first three generated numbers are 2061, 6175, and 4. Visualize 10,000 randomly generated numbers with a histogram of their values.

Saving and loading pRNG state in R

It is occasionally important to be able to save and load the pRNG state in R. For example, you may want to be able to reproduce the output of two "interwoven" random sequences beginning from different seeds.

The state of R's built-in pRNG is simply saved in the `.Random.seed` variable in the global environment. It can be written to a different variable and restored via simple assignment. There is one caveat: attempting to set `.Random.seed` within a function call will *create a local variable* called `.Random.seed` instead of changing the value of the *global* `.Random.seed` variable. Thankfully, the seed variable corresponding to R's pRNG can always be accessed with `.GlobalEnv$.Random.seed`.

Fast primality testing

Checking whether a number is prime or composite is a classic algorithmic task, stretching all the way back to 200 BC with the Sieve of Erastosthenes developed by Erastosthenes of Cyrene. We will first implement the Sieve and then work toward writing an implementation of the Miller–Rabin primality test, a modern test for primality known to be very fast in practice for reasonably small numbers.

The Sieve of Erastosthenes

The Miller–Rabin primality test

In the Miller–Rabin primailty test, we test the primality of a number $n > 2$ as follows: Since n is odd, $n - 1$ must be even, so we can write $n - 1 = 2^s \cdot d$, where d is odd. (For example, if $n = 13$, then $n - 1 = 12 = 2^2 \cdot 3$ with $s = 2$ and $d = 3$.) The Miller–Rabin primality test is based on the observation that if we can find a number a such that $a^d \not\equiv 1 (\mathrm{mod}\ n)$ *and* $a^{2^r d} \not\equiv -1 (\mathrm{mod}\ n)$ for all integers r in the range $0 \leq r \leq s - 1$, then n is not prime. Otherwise, n is likely to be prime.

Note that the Miller–Rabin primality test, as formulated here for a specific value of a, is *probabilistic* rather than *deterministic* – it cannot definitively establish that n is prime. It can be made deteterministic by checking all $a \leq 2(\ln n)^2$. Better yet, when n is sufficiently small, it has been found that we only need to consider a couple different values of a; for example, for $n < 4,759,123,141$, we only have to check $a \in \{2, 7, 61\}$.

We have one more utility function to write:

- Write a function `decompose_even(n)` which takes as input an *even* integer `n` and returns a vector of two integers `c(s, d)` such that `n` is equal to `2^s * d` and `d` is odd.

With `decompose()`, `decompose_even()`, and `pow3()`, we are now ready to implement the entire primality test.

- Following the above description, implement the deterministic Miller–Rabin test as `miller_rabin(n)` for $n < 4,759,123,141$, returning `TRUE` for a prime number and `FALSE` otherwise. (Note that checking if $x \equiv -1 (\mathrm{mod}\ n)$ is equivalent to checking if $x \equiv n - 1 (\mathrm{mod}\ n - 1)$.)
- Write a function `simple_check(n)` that checks if `n` is a prime by checking if `n` is divisible by any integers from 2 up to `floor(sqrt(b))`. Verify that `miller_rabin()` and `simple_check()` produce the same output for the first 100 integers. Use `timeit` to compare the performance of the two functions as `n` grows.

A small primality problem

We can apply the Miller–Rabin primality test to solve a simple problem in computational number theory.

- Find a counterexample to the following statement: By changing at most a single digit of any positive integer, we can obtain a prime number. Use the `memoise` package to easily perform memoization for the output of `miller_rabin()`. How much faster is your code with memoization compared to without memoization?