

R: Advanced Problems

Signal Data Science

things to cover: - profvis - memoise

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 a series of more challenging problems and exercises.

Run length encoding

- Write a function that prints out the longest “run” (sequence of consecutive identical values) in an input vector. (If there's more than one, print out the one that occurs first.)
 - Rewrite your function, incorporating the usage of `rle()`.

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 Eratosthenes](#) developed by [Eratosthenes of Cyrene](#). We will 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.

Modular exponentiation

First, 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 .

- 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 b th 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)?

To make further improvements easily, we will want to write the following 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.)

Finally, 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 $a^1, a^2, a^4, a^8, \dots \bmod c$), and then forming the final answer as a *product* of those intermediate calculations. (For example, for $6^{17} \bmod 7$, you are essentially calculating $17 = 2^0 + 2^4$ and $6^{17} \bmod 7 = 6^{2^0} \cdot 6^{2^4} \bmod 7$.) Using `decompose(n)`, implement this improvement as `pow3()`, making sure to calculate every intermediate result modulo c . Verify that `pow3()` is faster than `pow2()`.

The Miller–Rabin primality test

In the Miller–Rabin primality 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 $a^d \not\equiv 1 \pmod{n}$ and $a^{2^r d} \not\equiv -1 \pmod{n}$ 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 deterministic 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 `millier_rabin(n)` for $n < 4,759,123,141$, returning `TRUE` for a prime number and `FALSE` otherwise. (Note that checking if $x \equiv -1 \pmod{n}$ is equivalent to checking if $x \equiv n - 1 \pmod{n - 1}$.)
- 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 `millier_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

- 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 `millier_rabin()`.

Saving and loading the RNG

Quicksort and quickselect

One of the most straightforward sorting algorithms is *quicksort*, which sorts a list of length n in $O(n \log n)$ time. The steps of a simplified form of the algorithm are as follows:

1. For a vector L , pick a random position i . The element $L[i]$ is called the *pivot*. (If the pivot is the only element, return it.)
2. Form two vectors of elements `lesser` and `greater` which hold elements of L at positions *other than* i which are respectively lesser than or greater than $L[i]$. (Elements equal to $L[i]$ can go in either one.)
3. Call the algorithm thus far `qs()`. Our result is the combination of concatenating together `qs(lesser)`, $L[i]$, and `qs(upper)`.

Now it's your turn:

- Implement a `quicksort(L)` function that sorts a vector of numbers L from least to greatest. Verify that your function works by writing a loop

which generates 100 vectors of 10 random integers and compares the output of `quicksort()` to the built-in `sort()`. Compare the performance of `quicksort()` to that of `sort()`.

The *quickselect* algorithm, which is similar to quicksort, allows you to find the k th largest (or smallest) element of a list of n elements in $O(n)$ time. The difference in the algorithms is that in each iteration, we only have to recurse into *one* of the two subdivisions of the vector, because we can tell which one holds our desired value based on the value of k and the sizes of lesser and greater.

- Implement a `quickselect(L, k)` function which finds the k th smallest element of L .

Writing a simple spellcheck function

Spelling correction is one of the most natural and oldest natural language processing tasks. It may seem like a difficult task to you at the moment, but it's surprisingly easy to write a spellchecker that does fairly well. (Of course, companies like Google spend millions of dollars making their spellcheckers better and better, but we'll start with something simpler for now.)

- Read Peter Norvig's [How to Write a Spelling Corrector](#). Recreate it in R and reproduce his results. **After** doing so yourself, read about [this 2-line R implementation](#) of Norvig's spellchecker.

More n -dominoes

The following continuation to the study of n -dominoes (from the data frames lesson) is optional because of its open-ended nature.

Suppose that you have a single copy of every unique n -domino for some value of n .

- Write a function `make_circle(n)` that tries to construct a valid circle of n -dominoes from a *single copy* of every unique n -domino.
 - In the process of doing so, keep track of your various approaches.
 - Are there values of n for which no approach seems to work?
 - If so, can you make an argument about why you can't make a valid circle of n -dominoes for those values of n (using a single copy of every n -domino)? It may be instructive to look at the intermediate steps of your algorithm and how it fails.
 - Give a proof of your heuristic results.