Basic Algorithms

Signal Data Science

We'll conclude with a selection of exercises about common algorithms. The material below is likely to show up on programming-focused interviews!

Run-length encoding

Run-length encoding is a simple form of data compression which represents data as a series of *runs* (sequences that consist of the same character repeated multiple times). It was originally used in the transmission of television signals and was used as an early form of image compression on CompuServe before the development of GIF. Indeed, the modern JPEG image compression algorithm incorporates run-length encoding into its functionality.

- Write a function `arg_max(v)` which takes in a numeric vector `v` and returns the *position* of its greatest element. If its greatest element occurs in multiple places, print out the position of its first occurrence. You may find `max()` and `match()` helpful.
- Write a function `longest_run(v)` that prints out the longest "run" (sequence of consecutive identical values) in `v`. If there are multiple runs of the same length which quality, print out the first one. You may find `rle()` helpful. The function evaluated on `v = c(1, 2, 3, 3, 2)` should return `c(3, 3)`.

The Sieve of Erastosthenes

The Sieve of Erastosthenes is an algorithm for finding all prime numbers up to some prespecified limit N. It works as follows:

1. List all the integers from 2 to N.
2. We begin with the first and smallest prime number $p = 2$.
3. Remove all the multiples of p ($2p, 3p, \ldots$) aside from p itself from the list.
4. Find the first number greater than p in the list and set p equal to that number. Repeat step 3 or terminate if no such number exists.

The numbers in the list constitute the primes between 2 and N.

- Write a function `sieve(N)` which uses the Sieve of Erastosthenes to find and return a vector of all prime numbers from 2 to `N`. Check your function by evaluating `sieve(100)`, which should return 25 prime numbers from 2 to 97.

The Sieve is useful for generating primes, but not so much for *testing primality*; to know whether or not n is prime, one would have to generate all the prime numbers from 1 to n. There are much faster ways to check whether or not a *specific* number is prime, such as the Miller–Rabin primality test.

Reservoir sampling

A classic task in data analysis is the problem of reading in n data items one by one for a very large and *unknown* n and choosing a random sample of k items. This can be done with reservoir sampling, introduced in 1985 by Jeffrey Vitter as "Algorithm R".

The algorithm consists of the following:

1. Initialize a "reservoir" of size k populated with the first k data items.
2. Continue reading in the data items. For the ith data element, generate a random integer j between 1 and i inclusive. If $j \leq k$, then the jth item in the reservoir is replaced with the ith data item.

Now, following the above description:

- Write a function `reservoir(v, k)` which iterates over the elements of `v` a *single time* and randomly chooses k of them with reservoir sampling. (For the random integer generation, combine `floor()` with `runif()`.)
- Run `reservoir()` repeatedly, choosing 5 elements randomly from a vector of 20 elements. For each item, calculate the probability of it being chosen for the sample.

Permutation generation

Given a finite set of items in a given order, a permutation of those items is a distinct reordering of them. For example, a permutation of $\{A, B, C\}$ is $\{B, A, C\}$. The generation of permutations is yet another classic algorithms problem.

Suppose we wish to generate all permutations of the integers from 1 to n. The easiest way to do so is as follows: Begin with the set of all permutations of the integers from 1 to $n - 1$. For each of those permutations, insert n in every possible position to form a permutation of the integers from 1 to n. Discard the

repeats. To get the permutations of 1 to $n-1$, use the permutations of 1 to $n-2$; for those, use the permutations of 1 to $n-3$, and so on and so forth ...

- Following the above strategy, write a function `perm_naive(n)` to return a list of all permutations of the integers from 1 to n. You may find `unique()` helpful. Test your function on small values of n like 2, 4, and 6.

The above method is very slow, but there are much faster algorithms. Indeed, it is not even necessary to generate the permutations of 1 to $n-1$ in order to generate all permutations of 1 to n.

- We can generate permutations in lexicographic order. Follow the Wikipedia description of the algorithm to write a function `perm_lexico(n)` which returns a list of all the permutations of 1 to n in lexicographic order.

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. It was developed by Tony Hoare at Moscow State University as part of a translation project for the National Physical Laboratory requiring the alphabetical sorting of Russian words.

The steps of a simplified[1] 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 kth 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`.

[1] The presented algorithm does not operate *in place*.

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

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 $a^1, a^2, a^4, a^8, \ldots \bmod c$), 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()`.