R: 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.

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 print all permutations of the integers from 1 to n. 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 prints 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`.

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

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