Basic Algorithms

Signal Data Science

This lesson consists of a collection of standard algorithmic questions. The material below is likely to show up on programming-focused interviews, so study it well!

Project Euler

To whet your appetite, we list below the first ten Project Euler exercises. Register on the Project Euler website to submit and verify your answer to each problem. For each one, use a computational approach and try to minimize the runtime required.

- If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.

  Find the sum of all the multiples of 3 or 5 below 1000.

- Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be:

  $$1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

  By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.

- The prime factors of 13195 are 5, 7, 13, and 29.

  What is the largest prime factor of the number 600851475143?

- A palindromic number reads the same both ways. The largest palindrome made from the product of two 2-digit numbers is $9009 = 91 \times 99$.

  Find the largest palindrome made from the product of two 3-digit numbers.

- 2520 is the smallest number that can be divided by each of the numbers from 1 to 10 without any remainder.

What is the smallest positive number that is evenly divisible (*i.e.*, divisible with no remainder) by all of the numbers from 1 to 20?

- The sum of the squares of the first ten natural numbers is

  $$1^2 + 2^2 + \cdots\ 10^2 = 385.$$

  The square of the sum of the first ten natural numbers is

  $$(1 + 2 + \cdots + 10)^2 = 55^2 = 3025.$$

  Hence the difference between the sum of the squares of the first ten natural numbers and the square of the sum is $3025 - 385 = 2640$.

  Find the difference between the sum of the squares of the first one hundred natural numbers and the square of the sum.

- By listing the first six prime numbers: 2, 3, 5, 7, 11, and 13, we can see that the 6th prime is 13.

  What is the 10001st prime number?

- The four adjacent digits in the 1000-digit number below that have the greatest product are $9 \times 9 \times 8 \times 9 = 5832$.

  ```
  73167176531330624919225119674426574742355349194934
  96983520312774506326239578318016984801869478851843
  85861560789112949495459501737958331952853208805511
  12540698747158523863050715693290963295227443043557
  66896648950445244523161731856403098711121722383113
  62229893423380308135336276614282806444486645238749
  30358907296290491560440772390713810515859307960866
  70172427121883998797908792274921901699720888093776
  65727333001053367881220235421809751254540594752243
  52584907711670556013604839586446706324415722155397
  53697817977846174064955149290862569321978468622482
  83972241375657056057490261407972968652414535100474
  82166370484403199890008895243450658541227588666881
  16427171479924442928230863465674813919123162824586
  17866458359124566529476545682848912883142607690042
  24219022671055626321111109370544217506941658960408
  07198403850962455444362981230987879927244284909188
  84580156166097919133875499200524063689912560717606
  05886116467109405077541002256983155200055935729725
  71636269561882670428252483600823257530420752963450
  ```

  Find the thirteen adjacent digits in the 1000-digit number that have the greatest product. What is the value of this product?

- A Pythagorean triplet is a set of three natural numbers $a < b < c$ for which

  $$a^2 + b^2 = c^2.$$

  For eaxmple, $3^2 + 4^2 = 9 + 16 = 25 = 5^2$.

  There exists exactly one Pythagorean triplet for which $a + b + c = 1000$. Find the product abc.

- The sum of the primes below 10 is $2 + 3 + 5 + 7 = 17$.

  Find the sum of all the primes below two million.

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 form of the algorithm are as follows:[1]

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:

- Write a function `quicksort(L)` that sorts a vector of numbers `L` from least to greatest with quicksort. Verify that `quicksort(c(2, 4, 1, 2, 3))` returns `c(1, 2, 2, 3, 4)`. Compare the performance of `quicksort()` to that of `sort()`.

Quickselect

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

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

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`.

- Write a function `quickselect(L, k)` which finds the kth smallest element of L with quickselect. Verify that `quickselect(c(4, 1, 5, 9), 3)` returns `5`.

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()`.

Edit distance

It is often useful to be able to quantify the dissimilarity of two strings. This can be accomplished via computing the edit distance between them, corresponding to the minimum number of operations required to transform one string into the other. A natural application of edit distances is in the identification of misspellings and the suggestion of possible corrections.

Levenshtein distance

The most common edit distance is the Levenshtein distance, which allows for single-character insertions, deletions, and substitutions.[^lhist] "Edit distance" is commonly used to refer specifically to the Levenshtein distance.

To illustrate, the Levenshtein distance between "kitten" and "sitting" is 3, because the series of edits

$$\text{kitten} \xrightarrow{\text{substitution}} \text{sitten} \xrightarrow{\text{substitution}} \text{sittin} \xrightarrow{\text{insertion}} \text{sitting}$$

transforms "kitten" into "sitting" *and* there is no shorter series of allowed edits between the two strings.

Consider two strings s and t of lengths l_s and l_t; furthermore, let s_i and t_i represent the ith characters of each string. Let $s^\star$ and $t^\star$ respectively denote substrings of s and t containing all but the final character.

If $s^\star$ can be transformed into t in k edit operations, then with a **deletion** we can perform $s \to s^\star \to t$ in $k+1$ operations. Similarly, if s can be transformed into $t^\star$ in k operations, then with an **insertion** we can perform $s \to t^\star \to t$ in $k+1$ operations. Finally, if $s^\star$ can be transformed into $t^\star$ in k operations, then we can perform $s \to s^\star \to t^\star \to t$ in k or $k+1$ operations depending on whether or not a **substitution** is required to transform s_{l_s} to t_{l_t}.

This suggests a *recursive* strategy where we define the Levenshtein distance $L(s,t)$ as

$$L(s,t) = \min\left\{L(s^\star,t)+1, L(s,t^\star)+1, L(s^\star,t^\star)+c\right\}$$

where $c = 0$ if $s_{l_s} = t_{l_t}$ and $c = 1$ otherwise. However, a naive recursive implementation will be computationally costly, because the distance between identical substrings of s and t will be evaluated multiple times. Instead of implementing memoization directly, we can use the Wagner–Fischer algorithm, which cleverly builds up the calculation of $L(s,t)$ from the "bottom up" *à la* dynamic programming fashion.

The algorithm works as follows:

1. Initialize a matrix $\mathbf{M}$ of dimensions $(l_s+1)\times(l_t+1)$. In each entry $\mathbf{M}_{i,j}$ (where $i \in [0, l_s]$ and $j \in [0, l_t]$), we will store the Levenshtein distance between the first i characters of s and the first j characters of t. (Note that the rows and columns of $\mathbf{M}$ are indexed starting from 0 in this description, *not* from 1.)

2. Substrings of s can be transformed into the empty string purely via deletions and empty strings can be transformed into substrings of t purely via insertions. Fill in the leftmost column and uppermost row of M correspondingly.

3. Iterate over the remaining entries of $\mathbf{M}$ (starting at $\mathbf{M}_{1,1}$ and proceeding either downward or rightward). For each position $\mathbf{M}_{i,j}$, set $c = 0$ if $s_i = t_j$ and $c = 1$ otherwise, and then set

$$\mathbf{M}_{i,j} = \min\{\underbrace{\mathbf{M}_{i-1,j}+1}_{\text{deletion}}, \underbrace{\mathbf{M}_{i,j-1}+1}_{\text{insertion}}, \underbrace{\mathbf{M}_{i-1,j-1}+c}_{\text{substitution}}\}.$$

4. Return $\mathbf{M}_{l_s,l_t}$ as the answer.

This algorithm has both time and space complexity of $O(l_s l_t)$.

- Write a function `levenshtein(s, t)` which calculates the Levenshtein distance between `s` and `t` using the Wagner–Fischer algorithm. Verify that `levenshtein("kitten", "sitting")` returns 3.

Damerau–Levenshtein distance

Also used is the Damerau–Levenshtein distance, which is identical to the Levenshtein distance except for the allowance of transpositions between two adjacent characters (*e.g.*, "ab" $\to$ "ba") as well as insertions, deletions, and substitutions. The motivation behind the development of this new edit distance metric was to better represent the types of errors humans make when misspelling words and consequently improve the performance of *e.g.* spellcheck programs.

To calculate the Damerau–Levenshtein distance, only a small modification to the Wagner–Fischer algorithm is needed to represent transpositions. For each entry $\mathbf{M}_{i,j}$, if $i \geq 2$, $j \geq 2$, $s_i = t_{j-1}$, and $s_{i-1} = t_j$, $\mathbf{M}_{i,j}$ should be calculated as

$$\mathbf{M}_{i,j} = \min\{\underbrace{\mathbf{M}_{i-1,j}+1}_{\text{deletion}}, \underbrace{\mathbf{M}_{i,j-1}+1}_{\text{insertion}}, \underbrace{\mathbf{M}_{i-1,j-1}+c}_{\text{substitution}} + \underbrace{\mathrm{M}_{i-2,j-2}+c}_{\text{transposition}}\}.$$

- Write a function `d_levenshtein(s, t)` which calculates the Damerau–Levenshtein distance between `s` and `t`. Verify that `d_levenshtein("teacup", "taecop")` returns 2.