Determining Ramanujan numbers

Arjen Markus

April 11, 2024

Introduction

As described at the MathWorld webpage `https://mathworld.wolfram.com/TaxicabNumber.html`, Ramanujan or taxicab numbers come in two flavours, both related to the sum of two cubic integer numbers. In this note I follow the definition given in Sloan's on-line encyclopedia of integer sequences, `https://oeis.org/A001235`: a Ramanujan number is an integer that can be expressed as the sum of two cubic numbers in at least two ways. The smallest example is 1729:

$$1729 \quad = \quad 1^3 + 12^3 = 10^3 + 9^3 \tag{1}$$

The first few are (copied from the OEIS site): 1729, 4104, 13832, 20683, 32832, 39312, 40033, 46683, 64232, 65728, 110656, 110808, 134379, 149389, ...

It is easy enough to determine them with brute force and I do not know if any other methods are known. But that is not the point here. This note explores a few implementations to determine them. The idea is simple: let i and j run over the interval $[1, n]$ and calculate the sums of $i^3 + j^3$ for some suitable value of n. If the values occur more than once, we have found a Ramanujan number. Also note that interchanging i and j does not give a new value, so i should run over the interval $[1, j]$.

Side note: I first let the integers run over the interval $[0, n]$, but then I realised that a Ramanujan number with one of the two being zero would amount to a non-trivial solution to Fermat's last theorem. So I dropped that case.

Plain and simple

Here is a program that takes a straightforward approach:

```
program ramanujan_plain
    implicit none

    integer, parameter :: max_ij  = 100
    integer, parameter :: max_sum = 2 * max_ij ** 3
    integer            :: cnt(max_sum)
```

```
    integer            :: i, j, k

    !
    ! Determine the sums of two cubes
    !
    cnt = 0

    do j = 1,max_ij
        do i = 1,j
            k = i ** 3 + j ** 3

            cnt(k) = cnt(k) + 1
        enddo
    enddo

    !
    ! Write out the numbers for which there are at least two
    ! combinations i,j that give these numbers
    !
    do k = 1,max_sum
        if ( cnt(k) > 1 ) then
            write(*,*) k
        endif
    enddo
end program ramanujan_plain
```

I chose n to be 100 – `max_ij` in the program, so that the maximum value of the sum is 2 million.

The program simply loops over the two integers and counts how often the value occurs. For this, it uses a large array, `cnt`, plain and simple. Then it only prints those values that occur more than once.

The outcome is:

```
     1729
     4104
    13832
    20683
    32832
    39312
    40033
    46683
    64232
    65728
   110656
   110808
   134379
```

```
  149389
  165464
  171288
  195841
  216027
  216125
  262656
  314496
  320264
  327763
  373464
  402597
  439101
  443889
  513000
  513856
  515375
  525824
  558441
  593047
  684019
  704977
  805688
  842751
  885248
  886464
  920673
  955016
  984067
  994688
 1009736
 1016496
```

You would expect to see several numbers beyond 1 million, given that they are not too sparse, the intervals between successive numbers are rather small. The largest one found is much smaller than the limit of 2 million. This has to do with the fact that a value below 2 million may be the result of one of the integers being larger than 100 and the other smaller. So we do not find all Ramanujan numbers smaller than 2 million. You could adapt the program to take care of this, but it would make it less regular. And that is what I want to explore.

Using array operations

If we use Fortran's array operations, then we can make the program quite a bit shorter (leaving out the declarations for brevity, they are more or less the same

as in the first version):

```
!
! Determine the sums of two cubes
!
sums = [((i**3+j**3, i = 1,j), j = 1,max_ij)]
cnt  = 0
cnt(sums) = cnt(sums) + 1

!
! Write out the numbers for which there are at least two
! combinations i,j that give these numbers
!
write(*,*) pack( [(k, k = 1,max_sum)], cnt > 1)
```

The first line constructs an array via two implied-do loops and using the automatic reallocation feature, so we do not have to allocate the array `sums` beforehand and worry about the precise size.

Then to determine how often a particular sum occurs, we use a vector index:

```
cnt(sums) = cnt(sums) + 1
```

The accumulation occurs without us having to program that explicitly.

Finally, to determine which values are indeed Ramanujan numbers we use the `pack()` function. Because it only returns the elements of the array that is being packed, not the indices of the elements that are retained, we need to construct an array of such indices, but that is easily done.

(The outcome of this program is the same as the first one, except that several numbers are printed on the same line. That detail can be fixed of course.)

A compact version

A drawback of both these versions is that they rely on a large array to store the counts. If we want to determine larger and larger Ramanujan numbers, then this becomes a burden. So, here is an alternative: determine only the sums and count per value how often it occurs, then store the Ramanujan numbers in a separate array.

The code can be fairly compact as well, though we need an explicit do-loop, but the memory use is greatly reduced:

```
program ramanujan_compact
    implicit none

    integer, parameter   :: max_ij  = 100
    integer, allocatable :: saved_sums(:)
    integer, allocatable :: sums(:)
```

```
    integer              :: i, j, value

    !
    ! Determine the sums of two cubes
    !
    sums = [((i**3+j**3, i = 0,j), j = 0,max_ij)]
    allocate( saved_sums(0) )

    do while ( size(sums) > 0 )
        value = sums(1)

        if ( count( sums == value ) > 1 ) then
            saved_sums = [saved_sums, value]
        endif

        sums = pack( sums, sums /= value )
    enddo

    !
    ! Write out the numbers for which there are at least two
    ! combinations i,j that give these numbers
    !
    write(*,*) saved_sums
end program ramanujan_compact
```

The program first determines the sums and then examines the values one by one. The Ramanujan numbers are stored and in any case the elements of the array `sums` that are equal to the first value are removed for the next round. Thus a rough estimate of the memory use is at most three times the number of sums ($\approx 3n^2/2$) – once for the sums, once for the logical mask and once for the Ramanujan numbers, instead of $2n^3$. Numerically: 15,000 versus 2 million.

A surprise, perhaps, is that the numbers are not printed in a sorted order (edited to fit on a page):

```
      1729        4104       13832       20683       32832       46683
     39312       40033       65728       64232      110656      134379
    110808      165464      149389      216125      171288      195841
    327763      216027      262656      373464      402597      314496
    320264      525824      439101      593047      515375      513856
    558441      443889      513000      684019      885248      842751
    704977      955016      886464     1016496      805688      984067
    994688     1009736      920673
```

Reduce it to a simple statement

The implementations so far consist of two steps: determine the Ramanujan numbers and then print them. With a bit of ingenuity (the program is not easy to read) you can combine these two steps and end up with a program that essentially consists of one composite statement (leaving out the trivial declarations):

```
write(*,*) pack( [ (k, k = 1,max_sum)], &
                 [ ( count( [((i**3+j**3, i = 1,j), j = 1,max_ij)] &
                        == k ) > 1, k = 1,max_sum )] )
```

Because it computes the set of sums over and over again (two million times) it is very slow. If you store the results of the double implied do-loop, then it is quite a bit faster.

The program seems a challenge for some compilers. This is because compilers will try and evaluate the statement at *compile time* and that seems rather compute-intensive. In fact, the maximum n (or `max_ij` in the program) directly influences the time it takes for the compilation to finish, which would be very unlikely if the statement was treated as any ordinary code that should be run at run-time.

Conclusion

The excuse for this note was a mathematical topic that simply intrigues me, but the real subject is the variety of implementations that are possible, each having its own pros and cons, in terms of length, readability and memory use. Important instruments in the code presented are the array operations and the `pack()` function.