Pythagorean Triads

Arthur Sale
University of Tasmania

1 THE PROBLEM AND A PERSONAL BACKGROUND

The object of this report is to examine algorithms for generating *pythagorean triads*: triplets [a,b,c] of positive integers such that the pythagorean relation

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

is satisfied. Such triplets can be interpreted, of course, as examples of right-angled triangles with integer sides and are therefore constructable by the methods of euclidean geometry (compass and ruler). The best-known such triplet is [3,4,5] which is also the smallest. Every schoolchild must be familiar with [3,4,5] in constructed trigonometric exercises; possibly also with [5,12,13] which is also frequently met in such circumstances.

Why is this seemingly useless problem worthy of investigation? Before answering this question, let me relate my own personal involvement with pythagorean triads. My first brush with them came in 1963, when in the process of searching for suitable demonstration programs for a commissioning ceremony for an early mini-computer which our group had designed and built as a prototype, I came across some comments in an early mathematical text. Since the mini-computer concerned had only integer arithmetic (software multiply and divide), the problem seemed an appropriate demonstration, and was accordingly programmed, performed well (producing triads faster than the printing rate), and was forgotten.

The problem revived itself again two years later, in a different institution in the form of a challenge to demonstrate efficiency differences between hand-coded assembly language routines and high-level language routines (with hindsight, I probably did the challenger a disservice by showing such a marked difference). And again a year later when the problem was brought to me by a lecturer in electrical engineering who was tired of constructing example problems involving phasor (complex number) calculations with the [3,4,5] triad, and wanted to know if there were any more such numbers. He at least went away happy; as I shall show, there is no shortage of pythagorean triads: there is a countable infinity of them.

The final and last straw came a few months ago when reviewing a copy of a book intended for high-school computer studies, I idly remarked that the chapter which contained a discussion of Pythagorean triads as a search problem looked incorrect, and that there were other algorithms, only to be met with a request to document my assertion! Such a persistent problem, 1 thought, probably deserves looking at before I become totally haunted by the problem.

In more serious vein, the generation of Pythagorean triads cannot be said to be very important from an application point of view. However I present the problem here as one of *intrinsic* interest; it illustrates very well various dimensions of algorithm and method choice, and this without real numbers and without complex mathematics. I get tired of sorting and searching, so I now proffer pythagorean triads as a partial supplementation to those overworked examples. The algorithms presented illustrate some of the choices available to programmers in designing programs; show some of the hack-tricks and their value in appropriate light; and illustrate arguments for and feeling for orders of magnitude in algorithm efficiency. The report is accordingly dedicated to the computer science teachers who try to get their students to *think...*

2. BRUTE FORCE SEARCH

To reiterate: the problem is to find algorithms that generate and print triplets of integers a, b and c such that $a^2 + b^2 = c^2$. Of necessity, the generation must be limited, so let us impose a restriction on the values such that they are less than or equal to a chosen limit L. Presume that even the most naive programmer will realize that zero-values are uninteresting; then the simplest and most obvious technique is a brute-force search through the three-dimensional cubic lattice defined by:

$$1 \le a \le L \qquad 1 \le b \le 1 \qquad 1 \le c \le L$$

Assuming some obvious variable declarations, this might be coded in the following way;

```
for c:= 1 to L do
        for b:= 1 to L do
                for a:= 1 to L do
                        if a² + b² = c² then print (a,b,c);
```

If our naive programmer is capable of reason, he ought to notice that the outermost loop is traversed L times; thus the body of the b-loop is traversed L^2 times; and thus the body of the innermost a-loop is traversed L^3 times. Regardless of how often the print is invoked as a triad is found (for all we know the probability may be inversely proportional to L), the test in the inner loop is always executed and therefore for large L the running time of the program will have a term proportional to L^3. Since this will dominate for large L we say that the algorithm has a run-time *of order* L^3.

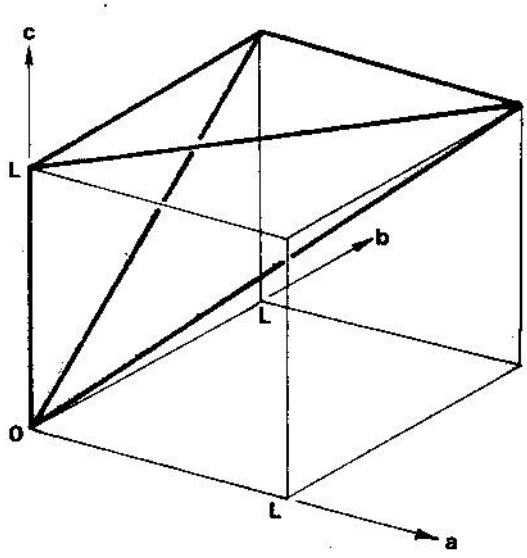
It is difficult to do anything other than improve from this very crude start. Indeed running the above program for some small L (say 100) will immediately show some room for improvement. Firstly it is obvious that c > a and c > b, so it is pointless looking at candidate triplets which violate this. Secondly, for each distinct triad such as [3,4,5], we shall get two versions printed: [3,4,5] and [4,3,5]. To remedy this (and realizing that the case a = b can never give a pythagorean triad since $\sqrt{2}$ is irrational) we can restrict the search to lattice-points such that

$$1 \leq a < b < c \leq L$$

Many programs can be written to this specification; one of the simpler versions is:

```
for c:= 3 to L do
        for b:= 2 to (c – 1) do
                for a:= 1 to (b – 1) do
                        if a² + b² = c² then print (a,b,c);
```

Figure 2.1 – Sub-volume of cubic space defined by $0 \leq a \leq b \leq c \leq L$



This seems dramatically better. And better it is: just less than 1/6 of the lattice-points (triplets) examined earlier are considered by the new algorithm (see Figure 2.1). Surely a worthwhile improvement! But look more deeply: the algorithm still searches a volume of the space, and therefore the number of candidate triplets still increases proportionally to L^3. Although it is six times faster, it is still of order L^3. Such improvements are only worth having if no better methods exist of lower order. And several other improvements might occur to the programmer more concerned with applying rules than with thinking about the problem itself; for example might he or she not be proud of refining the above into:

```
for c:= 3 to L do begin
        csquared := c²;
        for b:= 2 to (c – 1) do begin
```

```
                remnant:= csquared – b²;
                for a:= 1 to (b – 1) do
                        if remnant = a² then print (a,b,c)
        end
end
```

Of course still further changes are possible, for example converting the squarings into recursive formulations, but it is time to turn to superior algorithms. Even using the triangle-relation ($c < a + b$) does not change the algorithm from its essential dependence on L^3.

3 TWO-DIMENSIONAL SEARCH

Another obvious algorithm can be easily explained: search through all candidate *pairs* [a,b] and test each value ($a^2 + b^2$) to see if it is a perfect square, hence finding c. Ignoring for the moment the details of testing for perfect squaredness, this might be programmed with our current knowledge of the problem as:

```
for b:= 2 to (L-1) do
        for a:= 1 to (b-1) do
                if perfectsquare(a² + b²) then
                        print(a, b, sqrt(a² + b²));
```

Many programmers tend to reject this algorithm quite quickly, but let us look at it objectively. However complex the task of testing for perfect squaredness, provided the amount of work involved is more or less independent of the size of the number tested, or at least provided that the work does not increase proportionally to the size, then we have an algorithm which for large values of L is superior to those we have considered before. It is worth repeating this point: if the limit L is small the question of a best algorithm requires careful investigation both of algorithm, implemented code and machine characteristics; however when L gets large, an algorithm of order L^2 will eventually be better than an algorithm of order L^3 regardless of implementation details.

Fig. 3.1 shows graphically the sort of behavior we might expect of order L^3 algorithms compared with order L^2 algorithms: for small L the time taken to execute depends upon other terms in the expression relating the exact run-time with L, but with large L the L^2 or L^3 terms dominate. The exact form of the statements that can be made regarding the comparative merits of two algorithms can depend upon these other factors; thus algorithm A is always better than algorithm C, but in comparing algorithms B and C we can see that C is better for small L and B for large L with the changeover point depending on the sizes of the overheads and other factors.

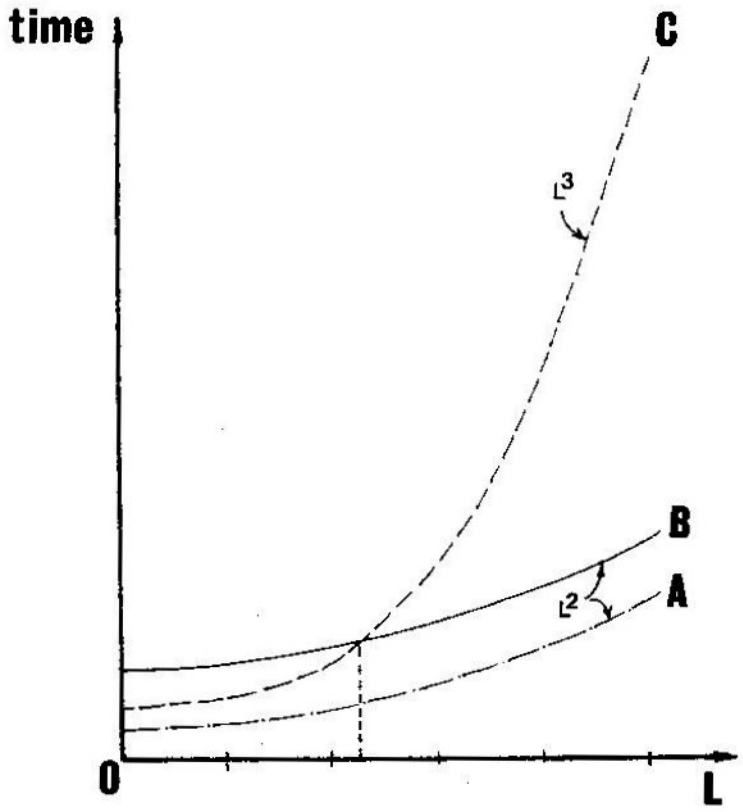
In actuality, practical methods of testing for perfect squaredness seem to generally involve an amount of work which is proportional to log L. For example the necessary number of Newton-Raphson iterations to find a sufficiently accurate square root is approximately proportional to log L, and the square-rooting method which is analogous to long divisions has a similar number of steps before determining whether any remainder exists. We could expect therefore that simple two-dimensional searches with root-testing procedures will be found to be of order (L^2 log L).

But we are still being stupid about this search process, testing each [a,b] pair without any thought as to whether previous information collected might be of use. In fact we will always program our search through [a,b] pairs in a systematic way (not randomly), and perhaps we can exploit this regularity in some way? The most obvious thing is that if one test gives us a candidate c-value (whether we reject it or not), then the next systematically generated pair ought to have its candidate c-value related to the previous one. Suppose that the last candidate pair tried produced a c-value which satisfies:

$$(c-1)^2 \leq a^2 + b^2 < c \qquad \text{[relation 3.1]}$$

Now if we step up a by 1 say, then since $a < c$ the only possible c-value that could form a triad with [a + 1, b] is the old value c; the possible outcomes are either an exact match (print the triad) or a new c-value satisfying relation 3.1 (with the upper limit now either remaining the old c or becoming (c + 1).

Figure 3.1 – Comparison of run-times



Here we have a two-dimensional search algorithm where the inner loop takes a time which is not significantly dependent on L (in fact decreases asymptotically to a limit as L increases so that this algorithm appears to have run-times of the order of L^2. Better still! It also does not need any complex computations of square roots since it keeps a running tally of putative roots as it goes.

```
for b:= 2 to (L-1) do begin
    trialc:= b + 1;
    for a:= 1 to (b – 1) do begin
     if a² + b² ≥ trialc² then begin
                        if a² + b² = trialc² then print(a,b,trialc);
                        trialc:= trialc + 1;
                end
        end
     end;
```

And once again of course any sensible programmer would endeavor to move unchanging computations out of the inner loop and to convert recalculations to recursive modifications, thereby speeding up the process without changing its essential speed dependence. Remember that this algorithm may be say 1000 times faster than any three-dimensional search for L=1000; improvements of another 2 or 3 times are still welcome but pale into relative insignificance.

4 PRIMITIVE TRIADS

Having played around with the problem briefly, it is about time that we made some effort to understand it rather better. It seems that even the ancient Babylonians knew several pythagorean triads besides [3, 4, 5] and we could start by looking at a list of the smallest such triads:

3	4	5
5	1	13
	2	

6	8	10*
7	2	25
	4	
8	1	17
	5	
9	1	15*
	2	
9	4	41
	0	
1	2	26*
0	4	
…	…	…

Of this small sample, one thing is immediately obvious, and perhaps should have been noted before: some of these triads are simple integer multiples of others earlier in the list. These are marked above with an t, for example [6, 8, 10] is 2 x [3, 4, 5]. Such triads are not very interesting (since once we know the simplest one we can find all the others) and accordingly we shall modify the original problem so as to require the generation only of the simplest triads which are not multiples of smaller triads. Such a triad we shall call a *primitive triad* (since it is the ancestor of a lot of others). Our list, if confined to primitive triads, shrinks to:

3	4	5
5	1	13
	2	
7	2	25
	4	
8	1	17
	5	
9	4	41
	0	

What would we have to do to our earlier algorithms to make them only print primitive triads? Obviously tighten up the print condition, by requiring that the greatest common divisor (gcd) of a, b, and c be 1 (in other words they must have no common factor). Knuth (1969) has discussed gcd algorithms in some detail and I shall not discuss them further here except to note that it is necessary only to do the gcd-test once a candidate triad has been found to satisfy the pythagorean relation; consequently the gcd procedure is invoked sufficiently infrequently that it will not upset any of my previous arguments.

A bit more examination of our list of primitive triads yields the interesting coincidence that all the c-values found are odd. Or is it a coincidence? Looking further at more values [11, 60, 61], [12, 35, 37], [13, 84, 85] it seems to hold up, so let's try the conjecture that c must be odd in a primitive pythagorean triad.

If c were instead even, then since c^2 is the sum of a^2 and b^2, then either a and b are both odd, or they are both even. The case of both a and b even can be easily disposed of since if c were also even then we would not have a primitive triad as two would be a common factor. The case of both a and b odd requires a little more work. Suppose we rewrite these as

$$a = 2p + 1 \quad \text{and} \quad b = 2q + 1$$

where p and q are still both natural numbers. Then we have

$$\begin{aligned} a^2 + b^2 &= (2p + 1)^2 + (2q + 1)^2 \\ &= (4p^2 + 4p + 1) + (4q^2 + 4q + 1) \\ &= 4(p^2 + p + q^2 + q) + 2 \end{aligned}$$

Clearly under these conditions the sum $(a^2 + b^2)$ is not exactly divisible by 4 (it always leaves a remainder of two), and yet if c is even, c^2 is divisible by 4. Conclusion: no pythagorean triad can have the set [odd,odd,even] whether primitive or not.

Therefore since c must be odd, all primitive triads must conform to one of the patterns:

[odd, even, odd] for example [7,24,25]
[even, odd, odd] for example [8,15,17]

This fact too can be used to slightly improve any of our earlier search algorithms by immediately eliminating half the candidates tested, or better still, by not generating them. For example, our last two-dimensional search (of order L^2 in run-time) might become:

```
for b.= 2 step 1 until (L – 1) do begin
        astart:= (if odd(b) then 2 else 1);
        trialc:= b + (if odd(b) then 2 else 1);
        for a:= astart step 2 until (b – 1) do  begin
                if a² + b² ≥ trialc² then begin
                        if a² + b² = trialc² then begin
                                if gcd(a,b,trialc) = 1 then print(a,b,trialc)
                        end;
                        trialc:= trialc + 2;
                end
        end
end;
```

5 THE Z^2 SEARCH

Since a, b and c have interesting odd/even properties, their differences should also retain interesting odd/even features. Since the pythagorean relation is symmetrical in a and b, consider the sums and differences:

$$x = c - a$$
$$y = c - b$$
$$z = (a + b) - c$$

Now if $a < b < c$ then x and y are always non-zero natural numbers, and for any a, b and c that can possibly form a triangle (and hence satisfy the pythagorean relation) z is also a non-zero natural number. Solving for a, b and c from the above we get

$$a = x + z$$
$$b = y + z$$
$$c = x + y + z$$

Now if such a triplet of numbers is to be a pythagorean triad:

$$a^2 + b^2 = c^2$$
$$(x + z)^2 + (y + z)^2 = (x + y + z)^2$$

which resolves itself down to:

$$z^2 = 2xy$$

Notice that although we could have decided that z must be even and that x and y form an odd/even pair from our earlier knowledge, this is also implicit in the above relation. It follows also that x and y cannot have any common factors if the triplet is a primitive pythagorean triad, for then the common factor would divide x, y and z; and hence a, b and c also.

Here then is the basis of a new family of algorithms, one of which involves the enumeration of all even values of z and hence z^2, followed by factorization of $z^2/2$ in all possible ways that give relatively prime x and y. Each such factorization (there is a least one for each z-value) can be transformed into a triad to be printed.

```
for z:= 2 step 2 until zmx do begin
    zsquare:= (z²) ÷ 2;
    x:= 1;
    y:= zsquare + x;
    while x < y do begin
```

```
            if (zsquare modulo x) = 0 then begin
              if gcd(x,y) = 1 then
                print(x + z, y + z, x + y + z)
            end;
              x:= x + 1;
              y:= zsquare + x;
        end
    end.
```

However this program solves a somewhat different problem to the one we have tackled hitherto: it generates all triads with a z-value less than a particular limit, and not all the triads with a c-value less than a specified limit. The two conditions are far from being the same. In addition the order of generation is no longer simply related to what we might term simple lexicographical orderings, though this may be less important. To tackle the first problem, we might insert a test to skip the print statement if the c-value is over its limit L; it will then be necessary to choose an upper zmx limit which is related to L by:

$$zmx = \frac{L}{1+\sqrt{2}}$$

We will therefore have to try a number of z-values which are directly proportional to L, and since the factorization loop will require $\sqrt{z^2/2}$ iterations for a given z-value, such a modification will give a program whose run-time still appears to be of order L^2! To be sure, we can do better than this by using some intelligence about the inner loop, but it is not worth it: a better method still exists with much the same scope for twiddling.

6 A DIRECT METHOD

Looking around for other clues, a good heuristic is to write down everything one knows about squares and squaring. Amongst the facts and junk there may appear the two equations:

$(m + n)^2 = m^2 + 2mn + n^2$
$(m – n)^2 = m^2 – 2mn + n^2$

A bit of playing around will soon produce

$(m + n)^2 – (m – n)^2 = 4mn$

or more excitedly

$(m – n)^2 + 4mn = (m + n)^2$

We note the resemblance of this last to $a^2 + b^2 = c^2$: two of the terms are already squares; is there anything we can do to make the third (4mn) also a perfect square? Obviously there is no problem with the 4 ($=2^2$); equally obviously putting m = n is useless as it reduces the $(m – n)^2$ term to zero. But if both m and n are individually squares, then we should have at least some pythagorean triads... Let us set:

$m = r^2$ and $n = s^2$

so that given suitable r and s we can immediately find a pythagorean triad

$[r^2 – s^2, 2rs, r^2 + s^2]$

Since in what follows, the odd- and even-ness of the a and b members of a triad appears more important than their numerical ranking, I shall in the following sections use *a as the odd member and b as the even member* of the pair (and it will not necessarily be true that a < b).

We must ask ourselves whether all pythagorean triads fit the pattern we have assumed; the answer interestingly is both yes and no. Suppose that we have an arbitrary triad [a,b,c] given to us. Then using only a,c we can compute suitable r and s values from

$r^2 = (c + a) / 2$ $s^2 = (c – a) / 2$

and hence derive the b-value (which must be the given b-value; there cannot be two different valuesl) as

$$b = \sqrt{(c+a)} \times \sqrt{(c-a)}$$

which can only be integer if the two roots are exact integers (and hence could have been found from examining all integer r,s pairs), or if they contain a common irrational factor, as for example

$$r = 2\sqrt{3} \qquad s = \sqrt{3}$$

which gives rise to [9,12,15]. However as this case hints, if a common irrational factor appears in r and s, then it certainly occurs (as an integer) in r^2 and s^2, and hence in a, b and c; such triads can never be primitive. We conclude then, curiously, that all primitive triads are expressible in our [r,s] form with r and s both integer, but all non-primitive triads are not! Some actually are included; a smallish example is

r = 6 s = 3

which generates [27,36,45]: 9 times the [3,4,5] triad. Clearly only non-primitive triads with perfect square multipliers can so masquerade, and these can be eliminated by testing r and s for common factors.

If we are to look only at primitive pythagorean triads, we know that c must be odd; it follows then that one of r and s is odd and the other even. It also follows that the even member of a and b (here the term 2rs) must be more than even — it must be divisible by 4. A look back at our table of small primitive triads confirms this deduction; here is an interesting fact which could have been used to Improve our earlier algorithms by halving the number of candidate triplets.

And so to programs ... two algorithms immediately suggest themselves:

(1) Generate successively all possible b-values (4, 8, 12, 16, 20,…)
and attempt to factorize (b/2) into integer factors r and s.
For each successful factorization with r and s relatively prime,
print a primitive triad.

(2) Generate directly all possible [r,s] pairs, test for relative primality, and print triads. It is only necessary to generate (odd,even) and (even,odd) pairs, so check [2,1], [4,1] ... [3,2], [5,2], [7,2] …

The first method, though potentially slower than the second (it examines more possibilities), repays study with some interesting results, A possible implementation would be:

```
for halfb:= 2 step 2 until (L ÷ 2) do begin
        s:= 1;
        r:= halfb + s;
        while s < r do begin
                if (halfb modulo s) = 0 then begin
                        if (gcd(r,s) = 1) and (odd(r) eor odd(s)) then beqin
                                c:= r² + s² ;
                                if c ≤ L then print(r² - s², 2 × halfb, c)
                        end
                end;
                s:= s+1;
                r:= halfb + s;
        end
end;
```

Clearly if a limit L is to be set on c-values of interest then b-values up to L will have to be considered; there will be $\lfloor L/4 \rfloor$ of them determined by the outer loop. For each b-value, trial divisors from 1 to $\sqrt{b/2}$ will have to be examined, consequently the body of the inner loop is executed

$$\sqrt{1} + \quad \ldots \quad + \sqrt{\frac{L-4}{4}} + \sqrt{\frac{L}{4}} \text{ times}$$

giving a run time which should be proportional to $L^{3/2}$. This is faster than any algorithm that we have examined hitherto.

Another interesting result is that at least one successful factorization will result for every b-value tested: factors 1 and b/2. Thus there is at least one primitive pythagorean triad for every suitable b-value, and the number of triads must therefore grow at a rate which is at least proportional to L. Actually, for all b-values which are not exact powers of two, there is another predictable factorization: (the highest power of 2 in b/2) and (the odd remnant).

The second method however seems both simpler and more direct, since it generates triads directly, requiring only tests for primitiveness. A simple version is:

```
for r:= 2 step 1 until rmx do
        for s:= (r – 1) step (–2) until 1 do
                if gcd(r,s) = 1 then begin
                        c:= r² + s²;
                        if c ≤ L then print (r² – s², 2 × r × s, c)
                end;
```

Unfortunately, like the z^2-search discussed in the previous section, this program is afflicted with some annoying defects.

(1) If the rmx limit is set to ensure the inclusion of all triads with a c-value such that $c \le L$, then a considerable number of oversize triads will be generated and rejected just before printing.

(2) The generated-triads are not produced with their [a,b,c] components in size order; the relative sizes of the first two components are also immaterial since their odd/even-ness determine the print order. (This can be patched fairly easily of course.)

(3) The triads are not (even considering the above change) generated in anything which could be regarded as a natural lexicographical order. Instead for each r-value the algorithm establishes a run of triads with decreasing a-value and increasing b- and c-values; different runs may well overlap.

Let us investigate the first problem. For a given r, the largest possible c-value that can be generated is

$$r^2 + (r-1)^2 = 2r^2 - 2r + 1$$

while the smallest is

$$r^2 + 1^2$$

If the program guarantees generation of all triads with $c \le L$, then all r-values up to $\sqrt{L-1}$ must be examined (giving a value for rmx). Since simply generating all triads composed of r lying below this limit and values of s < r will involve

$$1 + 2 + 3 + \quad \ldots \quad + \sqrt{L-1} \text{ executions of the loop body}$$

such an implementation will have a run-time of order L! And a considerable number of generated triads will have to be rejected as oversize: will their elimination in some more sophisticated way reduce the order of the algorithm?

TABLE 7.1 Triads generated from [r,s], $r \le 9$, $L \le 100$.

$$rmx = \left\lfloor \sqrt{L-1} \right\rfloor = 9$$

r	s	a	b	c
2	1	3	4	5
3	2	5	12	13
4	3	7	24	25

4	1	15	8	17
5	4	9	40	41
5	2	21	20	29
6	5	11	60	61
6	3	non-primitive		
6	1	35	12	37
7	6	13	84	85
7	4	33	56	65
7	2	45	28	53
8	7	oversize		
8	5	39	80	89
8	3	55	48	73
8	1	63	16	65
9	8	oversize		
9	6	oversize and non-primitive		
9	4	65	72	97
9	2	77	36	85

7 GENERATION ORDER AND EFFICIENCY

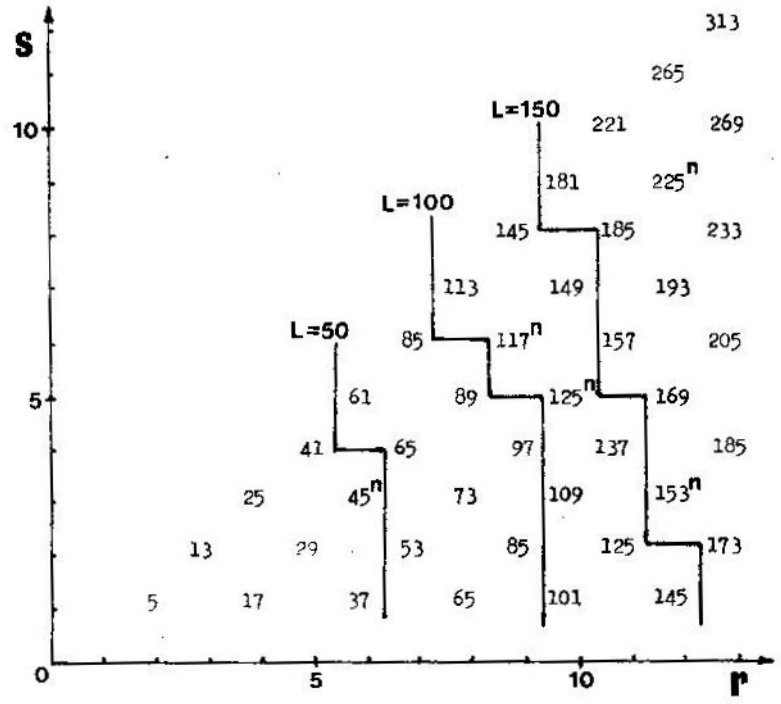
To improve the method, we again need better understanding of the problem. A useful heuristic to employ is to look at a small sample of the generated triads. Table 7.1 shows the triads generated using our previous algorithm with a limit L = 100 and hence rmx = 9. In this sample only two triads have to be rejected as non-primitive, and only three triads are oversize (one of which is included in the non-primitives). Is then the problem of generating oversize triads and rejecting them of no consequence? Actually a bit of both: more sophistication does not reduce the essential dependence of the run-time on L, but does reduce the proportionality factor: as L becomes large, the number of triads within the size limit approaches

$$\frac{\pi}{4} = 0.78539\ldots$$

of the total number tested by the simple algorithm.

To show this, or even to find it, we must resort to another heuristic: when in doubt draw a diagram. Figure 7.2 shows one plot that occurred to me: to show the [r,s] points in the two-dimensional plane together with the corresponding c-values and the limits imposed by L.

Figure 7.2 – Showing c-values for small r, s and L-boundaries
[Note: n = non-primitive triad]

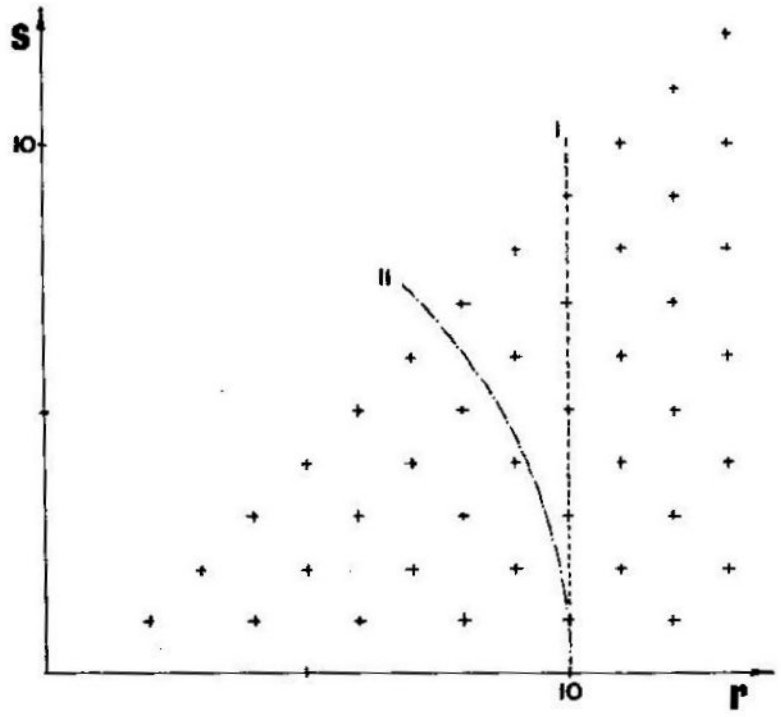


Another thought was to visualize this in three dimensions: a surface defined by r,s,c. The last step in the thought-process was to resort to continuous mathematics (which is often easier to visualize than discrete mathematics) whence everything clicked: the surface defined by continuous r,s,c is of course a surface of revolution: all points with the same c-value (a contour) lie on a circle centred on the [r,s] axis since $r^2 + s^2 = c$! Ignoring non-primitive [r,s] pairs, all lattice points with integer [r,s] falling inside the boundary for a given L are within size; all those falling outside are oversize. Since our first [r,s] program generated over a triangular area, and assuming that L is sufficiently large to ignore edge and discrete effects, the number of lattice points that are in size-limits compared to the total generated is:

$$\frac{\frac{1}{2}rmx^2\theta}{\frac{1}{2}rmx^2} = \theta = \frac{\pi}{4}$$

See Figure 7.3 for illustration of these ideas.

Figure 7.3 – (I) Limit of triads tested by first [r,s] program, (II) Limit of triads actually within L ≤ 100

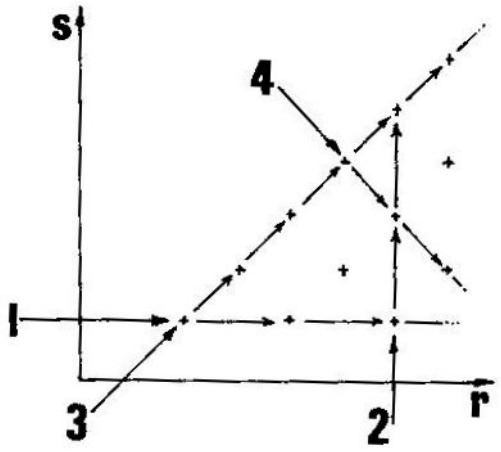


We can now also see what it would mean to generate triads in ascending order of c: they would have to arise from [r,s] lattice points which are chosen in order of distance from the [r,s] origin (ie run along contours); clearly a difficult task. The obvious (and fairly easily programmed) generation orderings are:

1. along horizontal rows of constant s, successively changing s for new rows;
2. along vertical columns of constant r, successively changing r for new columns;
3. along 45° upward diagonals, increasing both r and s by 1 for each step, then commencing on a new diagonal;
4. along 45° downward diagonals, increasing r by 1 and decreasing s by 1 for each step, then commencing on a new diagonal.

Our first program used generation order 2; these directions are illustrated in Figure 7.4. All have their good points perhaps: for example ordering 1 seems to give the longest runs of lattice points and fewest rows; hence resulting in the lowest loop housekeeping overhead. Orderings 1 and 3 have some generation rows which are (almost) parallel to radii from the origin, and hence generate triads with rapidly increasing c-values along such rows; even the most acute rows for these two orders are 45° from a radius. Ordering 2 and 4 perhaps are the nearest to contour followers: one end of each row is at right-angles to a radius, deteriorating to 45° at the other end; if as close an adherence to increasing order of c-value as possible is desired, then one of these is preferable.

Figure 7.4 – Showing easy scan directions

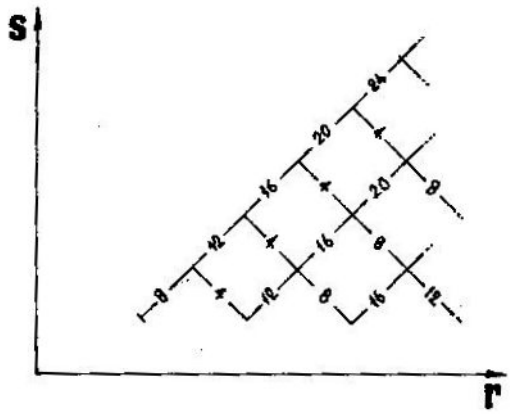


Let me choose to implement a program using ordering 4, and try to play the usual programming tricks on the inner loop. Since this is a problem involving squares, we might expect that second-differences are constant

by analogy with quadratic expressions, but instead over a two-dimensional space. And they are: Figure 7.5 shows differences near the origin which should clearly show the way to replace recalculation of squares etc by recursive computations within the innermost loop. Exactly similar situations arise for the values of a and b, and it becomes possible then to produce a new version of the [r,s] program.

```
diffa:= 6;
for j:= 2 to jmax do begin
  diffc:=4;
  a:= 2 × j – 1; b:= (a – 1) × j; c:= b + 1;
  for k:= 2 to j do begin
        if c > L then escape {from the k-loop};
        if gcd(a,b) = 1 then print(a,b,c);
        a:= a + diffa; b:= b – diffc; c:= c + diffc;
        diffc:=diffc + 4;
  end;
  diffa:= diffa + 4;
end;
```

Figure 7.5 – Showing first differences for c on [r,s] diagram



Of course, despite the work that has gone into producing a 'polished' piece of code recursively formulated, this may not be significantly faster than our earlier program; indeed may not be faster at all! This is often quite surprising to novice programmers, but notice that although this program has only additions and subtractions in the inner loop and no squarings, any actual speed differences will depend upon actual code generated for a particular computer. Also, it is possible that the gcd and print-formatting calculations will dominate the inner loop so that the savings are negligible.

Actually the time required to compute the gcd depends on the smaller of the two numbers involved, and grows slowly with increase of values. Since the computation of a gcd is no longer a relatively rare event, we cannot continue to ignore this time dependence, and the presented algorithm is not strictly of order L. Sieving techniques, or prefactorizations of (say) s can improve the situation, but would lead us too far away from my theme.

8 FURTHER PROBLEMS FOR SOPHISTICATED READERS

If any readers yet remain, and if perchance they may feel cheated of opportunities to exercise their own mental muscles, I have saved some small problems. (I trust no-one thought that triads had been exhausted in interest?)

(1) In the original L^3 search problem, I suggested that even if the search area were restricted to lattice points satisfying

$$0 < a < b < c < a + b$$

the algorithm would remain of order L^3. However, how many of the lattice-points in the original cubic space will need to be examined (as a fraction of the total, assuming large L)? Can you implement the algorithm?

(2) Writing $a^2 = c^2 – b^2 = (c + b) \times (c – b) = (2b + f) \times f$ (where $f = c – b$)

then $b = (a^2 - f^2) / (2f)$
If I assume that a is divisible by f so that $a = f \times g$, then:

$$b = (f \times (g^2 - 1)) / 2$$

which can only be integer if (i) f is even, or (ii) g is odd. Is this justifiable? Is it the basis of new triad programs? How fast is it? In what order does it generate triads?

(3) Assume that it is imperative that the triads below a limit L be available in increasing order of c-value. To avoid having to generate a whole set and then sort them, it is suggested that a modification of the [r,s] algorithm be used, whereby a scan direction which always gives increasing c-values is chosen (all those given in section 7 satisfy this), and that the new algorithm keeps a record of how far it has gone along *each* scan-line in the chosen direction. At each generation step the inner loop selects that new [r,s] point with the smallest c-value to generate a triad, afterwards updating it to the next scan-point on its line. This will clearly require an activation record for each scan-line, holding (say) r,s and c. How many activation records are required for scan-direction-3 and scan-direction-4 for a given L? (In other words what are the space requirements of this algorithm?)

(4) On the same problem of (3), simple methods of making the selection (eg maintain a sorted vector of the activation records) seem to involve a time which depends linearly upon r, thereby implying a degradation of the algorithm to order $L^{3/2}$. Can you do better by organizing the selection/updating in another way? Can you program this method in scan-direction-4 which involves a queue-discipline? Scan-lines become irrelevant as well as relevant, as the generation proceeds.

9 OVERVIEW

I have attempted in the foregoing sections to illustrate how one might go about looking for algorithms for solving problems for which there are not ready-made solutions which might simply be looked up (for example in Knuth). In the spirit of Polya, it is then necessary to encourage the conscious employment of heuristics, both to throw up algorithms or hints of them, and to approximately analyse the performance of resultant implementations in programs. Although the problem chosen is apparently trivial, it has called into play for me remembered snippets of information concerning quadratic equations, elementary number theory, the theory of differences, geometry and surfaces, series summation and integrals. Clearly a richly connected set of experiences helps.

More usefully, the previous sections should have shown how one can cast around, trying various tacks and investigating them, writing down as many possibly relevant relations as you can and seeing where they lead, generating samples of the desired results and examining them for patterns and clues, drawing diagrams or pictures to show relations in a new light, re-examining old methods in the light of new facts, and looking for analogies and extrapolations. Hopefully no-one actually imagines that all these methods and techniques developed in the order shown in full bloom! That would be too much, and indeed I was given a nudge to demonstrate the odd/even properties of a and b in the neat way shown fairly late in the investigation; due to an inexplicable mental block I was only able to demonstrate these properties in a devious way through the $z^2/2$ search. It would be far too lengthy to try to illustrate all the thought processes involved.

This particular problem was used as an extended exercise in the first-year course in computer science at the University of Tasmania in 1975 and some years thereafter to develop skills in estimation and algorithm finding. The students are given initially the problem and some algorithms to implement and invited to estimate their speed. Later, after these have been discussed they may be given some clues for further trials, and they are expected to follow these up, the tutorials serving as both brainstorming sessions, and critical evaluation of previous attempts. It is not expected, nor is it necessary, that all students achieve the sophistication of all the methods presented here (or even of unknown to me): it is more important that they try. Exercising talents that are rusty or perhaps latent is not easy at first attempt. As a student exercise pythagorean triads have one deficiency: only speed dependencies are involved, not space. However as a starting exercise perhaps this is a good thing?

I trust that this paper can be considered as a useful contribution to the literature of teaching about the skills of programmers. I have tried to communicate the insights and thought-processes involved in exploration of algorithms. Unfortunately, with the elevation of structured programming to a there has grown up the notion that all problems are reducible to (1) look it up in Knuth, a textbook, Collected Algorithms of the ACM, or

whatever, or (2) use a top-down design method. Here, in a small-program situation bereft of any large-program excuses, the cut-and-try/iterate process typical of poorly understood design situations can be clearly perceived. Skill in tackling such problems is at least as important (if not more) than the techniques of producing maintainable code, or workable implementations, even if much more difficult to teach…

10 ACKNOWLEDGEMENTS

This work was carried out at the University of Tasmania, Hobart. I should like to record my thanks to Richard Watkins and Jack Oliver for letting me bounce ideas off them and for injecting noise and information into my thought processes.

11 REFERENCES

A select list referred to in the text, or the sources of some ideas.

BROWNELL S, MURRAY G, and OLIVER J. (1975): *Computer Studies*, Advance Publicity Co., Hobart, Tasmania, p166-168 (triads).

HOGBEN L.. (1960): *Mathematics in the Making*. Galahad Books, p65-68 (triads).

KNUTH DE. (1968): *The Art of Computer Programming - Vol. 1: Fundamental Algorithms*. Addison Wesley, p2-9 (gcd).

KNUTH DE. (1969): *The Art of Computer Programming - Vol. 2: Semi-numerical Algorithms*. Addison-Wesley, p293-338 (gcd) and p338-360 (factoring).

POLYA G. (1945): *How to Solve It - A New Aspect of Mathematical Method*. Princeton University Press (also Doubleday Anchor Books,1957). (Problem-solving heuristics.)

SALE AHJ. (1964). Programming of the PII Computer - Final Report. Philips International Institute Report No 138. Section 7.7 (triads).

APPENDIX ON DIFFERENCES

This appendix justifies the differencing employed in the last program of section 7. Though self-evident from the diagram and the tables it is satisfying to be able to satisfy oneself that no mistake has been made. It is well-known that

$$x^2 = 1 + 3 + 5 + \ldots + 2x - 1 = \sum_{j=1}^{x} 2j - 1$$
$$xy = x + x + x + \ldots + x = \sum_{j=1}^{y} x$$

In going therefore from an [r,s] point to the next at (r + 1, s – 1] the respective c-values will differ by:

$$((r+1)^2 + (s-1)^2) - (r^2 + s^2) = (2(r+1) - 1) - (2s - 1) = 2(r - s + 1)$$

In other words add a new term (2(r + 1) – 1) and remove a term previously present (2s – 1).

Clearly the second differences will be 2(1 – (–1)) = 4

Similarly the a-values can be shown to differ by:

$$((r+1)^2 - (s-1)^2) - (r^2 - s^2) = 2(r + s)$$

which is constant along the chosen generation lines.

The b-values require different evaluation:

$$(r+1) \times (s-1) = ((r+1) \times s) - (r+1) = ((r \times s) + s) - (r+1)$$

giving a difference of –2(r – s + 1); the negative of the c-difference.

APPENDIX ON ACTUAL TIMINGS

Actual timings of programs implemented on a Burroughs B6700, for varying limits L on acceptable triads.