Pythagoras' Lost Lecture: A Journey Through Recursion, Stacks, Binary Search, and Hash Tables*

Saad Mneimneh

*** not yet published ***

ABSTRACT

As an advocate of infusing various algorithmic and mathematical aspects when teaching about programming, I have come to realize that an early such practice is essential for a rounded computer science education. In this paper, I pretend that Pythagoras is presenting ideas about his theorem $a^2 + b^2 = c^2$ while pointing out several computer science concepts such as recursion, stacks, binary search, and hash tables.

"You may be right, Pythagoras,
but everybody is going to laugh
if you call it Hypothenuse!"
– Anonymous

1. PART I: RECURSION ON THE HYPOTHENUSE

... therefore, since $a^2 + b^2 = c^2$, when given a right angle triangle with sides a and b, we can find the length of the hypothenuse as $\sqrt{a^2 + b^2}$. The challenge is in computing this square root function using the basic arithmetic operations that we know. Newton's [1] method will come in handy. If y is a guess for the square root of x, then

$$\frac{y + x/y}{2}$$

is a better guess. Based on this observation, we have the following definition for the square root (now a function of two parameters). [2]

$$\text{sqrt}(x, y) = \begin{cases} \text{sqrt}\left(x, \frac{y+x/y}{2}\right) & |y^2 - x| > \epsilon \\ y & |y^2 - x| \leq \epsilon \end{cases}$$

It is easy to convert this mathematical definition to a recursive implementation using any programming language. Figure 1 shows a pseudocode (henceforth pseudocodes will be called algorithms).

```
sqrt(x, y)
  if |y^2 − x| > ε
    then return sqrt(x, (x + x/y)/2)
    else return y
```

Figure 1: Computing the square root of x using an initial guess y.

Here's an example: Let $x = 2$ and assume that our guess for $\sqrt{2}$ is 1. If $\epsilon = 0.001$, our guess will change as follows:

$$1$$

$$\frac{1 + 2/1}{2} = 1.5$$

$$\frac{1.5 + 2/1.5}{2} \approx 1.4167$$

$$\frac{1.4167 + 2/1.4167}{2} \approx 1.4142$$

We stop because $|1.4142^2 - 2| \leq 0.001$.

Now back to our triangle. Since $\max(a, b) \leq c \leq a + b$ (triangular inequality, but can also be obtained from $c^2 = a^2 + b^2$), we can start with the guess $c = [\max(a, b) + a + b]/2$ (the middle). Finally, we have an algorithm to compute the length of the hypothenuse in Figure 2.

But how does recursion really work? Well, it's like transmigration: [3] when the square root function returns, a new incarnation of the function with new parameters is born. This is best illustrated by a stack.

*Either Pythagoras wrote nothing or everything he wrote was lost. Hence the title. Moreover, many detailed explanations in this lecture have been lost and, therefore, it should only serve as a guide for the interested Pythagorean instructors; they are assumed to develop their own lecture, or series of lectures, with the appropriate level of detail for their audience.

[1]Pythagoras lived from 570 to 490 BC and Newton lived from 1642 to 1727. Some ancients, however, believed that Pythagoras had a thigh of gold and he could travel through space and time.

[2]This will work correctly only if $x, y \geq 0$.

[3]Pythagoras believed that the soul begins a new life in a new body after death.

hypothenuse(a, b)
  return sqrt$(a^2 + b^2, (\max(a, b) + a + b)/2)$

Figure 2: An algorithm to find the length of the hypothenuse

2. PART II: STACK THEM UP!

A stack s is a structure that supports the following operations:

- push(s, e) adds element e to the top of the stack
- pop(s) removes the element on the top of the stack and returns its value
- $s[i]$ represents the i^{th} element on the stack ($i = 1$ for the bottom element and i is largest for the top element)

To understand recursion, imagine that sqrt(x, y) makes a series of push and pop operations on a stack s using the various instances of the parameters (x, y) as the elements. When the square root function is called, the parameters are pushed on the stack; when the function call returns, the parameters are popped from the stack. Figure 3 shows the state of the stack for the $\sqrt{2}$ example of the previous section.

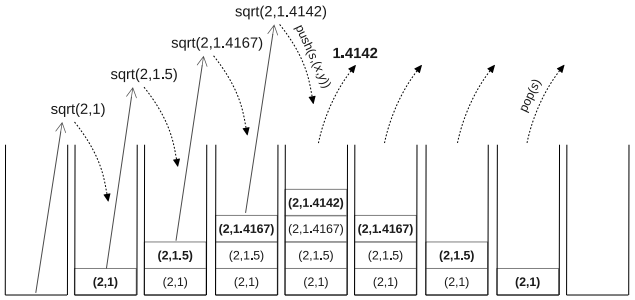

Figure 3: Starting with an empty stack, the first call to the square root function pushes its parameters on the stack. Every subsequent call does the same. The top of the stack is shown in bold. When a call returns, it pops the stack. The first popped value is 1.4142, which is returned. Every subsequent pop ignores the popped value and simply returns whatever was returned previously, leading to the final result 1.4142 (and an empty stack).

Understanding the stack provides a lot of insight about recursion. For instance, the amount of memory needed is proportional to the largest height attained by the stack. In this particular implementation of the square root function, we also observe that upon returning from the function call, the popped value is ignored except in the first time. This means that, throughout the entire execution, we only need to store the top of the stack (and not everything in it). In fact, this is what makes it possible to transform a recursive implementation into an iterative one. Not every recursive implementation can be transformed. We call the ones that can tail recursive (when the recursive call itself is the last thing to be done, observe the square root function again).

This is not to be confused with the fact that we can always eliminate recursion entirely by emulating the stack. Tail recursion means we can eliminate the (redundant) memory requirement of the stack. As a rule of thumb, Figure 4 provides a quick recipe for transforming a tail recursive implementation of a given form into an iterative one.

```
tail_rec_f(x, y, z, ...)
  if condition
    then return tail_rec_f(x', y', z', ...)
    else return w

iter_f(x, y, z)
  while condition
    do x ← x'
       y ← y'
       z ← z'
       ⋮
  return w
```

Figure 4: Tail recursive to iterative: (1) Change the if to a while, (2) replace the recursive call by an update of the parameters, and (3) drop the else (if any).

Applying this technique to the square root example will produce the (rather familiar) form of Figure 5:

```
sqrt(x, y)
  while |y^2 − x| > ε
    do y ← (y + x/y)/2
  return y
```

Figure 5: Iterative version of the square root function using the technique outlined in Figure 4.

```
01000101
10101
0110010
```

3. PART III: THE PLAIN OLD BINARY SEARCH

Now consider the problem of identifying Pythagorean triples.

Definition 3.1 (Pythagorean Triple). *A triple (a, b, c), where $a, b, c \in \mathbb{N}$, is called Pythagorean iff $c^2 = a^2 + b^2$.*

The problem is posed as follows: given a list l of positive integers (indexed $l[1]$ to $l[n]$) and a positive integer c, find

if l contains two integers a and b, such that (a, b, c) is a Pythagorean triple.

This is a problem of search, and it can be solved in a trivial way as shown in Figure 6.

```
search(l, c)
  for i ← 1 to n
    do for j ← i + 1 to n
          do if c² = l[i]² + l[j]²
                then return true
  return false
```

Figure 6: Algorithm to find if integer c forms a Pythagorean triple in list l.

The running time of the algorithm in Figure 6 is proportional to the square of n, i.e. $O(n^2)$. To see this:

$$\sum_{i=1}^{n}\sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n}(n-i) = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = O(n^2)$$

But it can be improved to $O(n \log n)$ using a more efficient search, called binary search. [4]

To introduce binary search, assume that we have a set T of ordered tuples, $T \subset \{(a, b)|a, b \in [n] \cup \{0\}\}$. We will also assume the following property for T:

DEFINITION 3.2 (REVERSE TRANSITIVE). *If $(i, j) \in T$ and $k \in [n] \cup \{0\}$, then $(i, k) \in T$ or $(k, j) \in T$.*

This is equivalent to say:

$$(i, k) \notin T \wedge (k, j) \notin T \Rightarrow (i, j) \notin T$$

which is transitive (thus the name reverse transitive).

Binary search assumes that a starting tuple (i_0, j_0) is in T and that $i_0 < j_0$ and finds an i such that $(i, i + 1) \in T$, as follows: [5]

```
binary_search(i, j)
  if j > i + 1
    then choose i < k < j
          if (i, k) ∈ T
             then return binary_search(i, k)
             else return binary_search(k, j)
    else return i

find(i₀, j₀)   ▷ (i₀, j₀) ∈ T ∧ i₀ < j₀
  return binary_search(i₀, j₀)
```

Figure 7: General binary search, typical value of k is $\lfloor (i + j)/2 \rfloor$.

[4]Pythagoras must have learned about sorting and the big O notation throughout his time travel.

[5]The general binary search algorithm thus presented does not require sortedness.

In Figure 7, if k is chosen to be $\lfloor (i + j)/2 \rfloor$, the search space is reduced by half for each recursive call and, therefore, the running time for binary search will be $O(\log n)$.

But how can this be used for our purpose of finding if a given c forms a Pythagorean triple in list l? The answer is that we have to set up T appropriately. Let's say we want to check if c forms a Pythagorean triple with $l[i]$ and some other element in l. Then we can compute $x = \sqrt{c^2 - l[i]^2}$. If $x \in \mathbb{N}$ and $l[j] = x$ for some j, then $(l[i], l[j], c)$ is a Pythagorean triple. Otherwise, c does not form a Pythagorean triple with $l[i]$ and another element of l. When we check this fact for every $i \in [n]$, we are done. Therefore, we need to set up T in such a way that binary search will determine whether x occurs in l or not, and more generally in $l[i_0], \ldots, l[j_0]$. Upon doing this, we will have achieved a running time of $O(n \log n)$.

To start, assume that l is **sorted**, e.g. $i < j \Rightarrow l[i] \leq l[j]$ (sorting can be done in $O(n \log n)$ time). Assume further that given x, $l[i_0] \leq x$ and $l[j_0] > x$ (we will relax this assumption later). Now define T as follows:

$$T = \{(i, j)|l[i] \leq x \wedge l[j] > x\}$$

By assumption, $(i_0, j_0) \in T$. It is also obvious that if $(i, j) \in T$, then for every k, either $(i, k) \in T$ or $(k, j) \in T$. Therefore, starting with (i_0, j_0), binary search will find an i such that $(i, i + 1) \in T$. If $l[i] = x$, then x occurs in l; otherwise it doesn't because l is sorted and $l[i] < x$ and $l[i + 1] > x$.

Finally, we can virtually consider that $l[i_0 - 1] = -\infty$ and $l[j_0 + 1] = \infty$ (thus $(i_0 - 1, j_0 + 1) \in T$) and work with the virtually modified list $l[i_0 - 1], \ldots, l[j_0 + 1]$ to relax our assumption that $l[i_0] \leq x$ and $l[j_0] > x$; this will also guarantee the condition $i_0 - 1 < j_0 + 1$ if the list has less than two elements, i.e. if $i_0 \geq j_0$ ($j_0 - i_0 = -1$ if the list is empty and $j_0 - i_0 = 0$ for a singleton). Figure 8 shows binary search adapted to our problem.

```
binary_search(l, x, i, j)
  if j > i + 1
    then k ← i + ⌊(j − i)/2⌋   ▷ = ⌊(i + j)/2⌋ but better
          if l[k] > x
             then return binary_search(l, x, i, k)
             else return binary_search(l, x, k, j)
    else return i

find(l, x, i₀, j₀)   ▷ search l[i₀], ..., l[j₀]
 i ←binary_search(l, x, i₀ − 1, j₀ + 1)
 if i = i₀ − 1
    then return false
    else return l[i] = x
```

Figure 8: Binary search for x in a sorted list.

Observe that the above implementation is tail recursive. Following the technique of Figure 4, it can be transformed to iterative as shown below (Figure 9). We are now ready to finalize our Pythagorean triple search with the algorithm of Figure 10 (the algorithm can be modified by removing the condition $l[i] \leq x$ and simply using find$(l, x, 1, n)$ because virtually $l[0] \leq x$ and $l[n + 1] > x$).

```
binary_search(l, x, i, j)
  while j > i + 1
     do k ← i + ⌊(j − i)/2⌋
        if l[k] > x
          then j = k
          else i = k
  return i
```

Figure 9: Iterative version of Figure 8. Again, (1) change the if to a while, (2) replace the recursive call with an update of the parameters, and (3) drop the else.

```
search(l, c)
  sort list l   ▷ can be done in O(n log n) too
  for i ← 1 to n
    do if l[i] ≤ c   ▷ sqrt will work
          then x ←sqrt(c² − l[i]², c)
               if x ∈ ℕ ∧ l[i] ≤ x   ▷ virtually, l[n + 1] > x
                  then if find(l, x, i + 1, n)
                            then return true
  return false
```

Figure 10: Improved algorithm to find if integer c forms a Pythagorean triple in list l.

What if we want to repeat our search for different values of c? Can we process the list for such a repeated use to improve upon the $O(n \log n)$ time? Can we even hope for a constant time, i.e. $O(1)$? Yes, we can, if we use a hash table.

Did you say hash...?

4. PART IV: FINDING PYTHAGOREAN TRIPLES BY HASHING

To introduce hash tables, consider first a simple table t indexed from $t[0]$ to $t[\lfloor 1.5M \rfloor]$, where M is the largest value in our list l. The key idea is that we can process l and determine, for all values of $c \in [\lfloor 1.5M \rfloor]$, whether c forms a Pythagorean triple in l or not, and store that information (true or false) in $t[c]$, which can be later obtained in constant time with a simple access to the table. Observe that c cannot be larger than $\lfloor 1.5M \rfloor$ because $\sqrt{M^2 + M^2} = \sqrt{2}M < 1.5M$. This idea is illustrated by the algorithm in Figure 11.

Therefore, we pay the price of $O(n^2)$ time only once, but then answer every question in constant $O(1)$ time. But there is a problem! We need to guarantee that $t[c]$ is false initially, and to do this, we have to explicitly initialize all $\lfloor 1.5M \rfloor$ entries in the table. But if $M \gg n^2$, this may not be a good idea (too slow). Fortunately, we can skip the initialization with the help of a stack. [6]

[6]Pythagoras learned this trick from the CLRS book.

```
process(l)
  for i ← 1 to n
    do for j ← i + 1 to n
          do c ←hypothenuse(l[i], l[j])
             if c ∈ ℕ
               then t[c] ← true

search(c)
  return t[c]
```

Figure 11: Processing the list l using an auxiliary table t.

Instead of setting $t[c]$ to true, we push c on a stack s. We then store in $t[c]$ the position of c in the stack, i.e. $t[c] = i$ such that $s[i] = c$. We can keep track of the height of the stack. Given a value for c, we can retrieve $i = t[c]$, and check if i is at most the height of the stack. If that's the case, we read $s[i]$. If $s[i] = c$, we must have pushed it on s and that's our witness. There is no need for initialization. The algorithm is revised below in Figure 12.

```
process(l)
  h ← 0   ▷ the height of the stack (initially empty)
  for i ← 1 to n
    do for j ← i + 1 to n
          do c ←hypothenuse(l[i], l[j])
             if c ∈ ℕ
               then push(s, c)
                    h ← h + 1
                    t[c] ← h
search(c)
  i ← t[c]
  if 1 ≤ i ≤ h ∧ s[i] = c
    then return true
    else return false
```

Figure 12: Using a stack s to skip the initialization of t.

When $M \gg n^2$, the explicit initialization of t did not justify the resulting loss in efficiency. By the same token, if $M \gg n^2$, we are wasting too much memory. This will happen to be just the one incentive to do the following: Simply, make t much smaller, e.g. with $m \ll M$ entries indexed from 0 to $m - 1$, and call it a hash table. We can now revert to the algorithm of Figure 11 with the proper initialization of t that we have previously skipped, since $m \ll M$ (we will need to make $m = O(n^2)$ to maintain the same efficiency).

But not every value of c will have a place in t; namely, if $c \geq m$. Subsequently, if we wanted to set a value for $t[c]$, we would set $t[c \bmod m]$ instead, because $c \bmod m \in [0, m-1]$. We say that c hashes into position $c \bmod m$, and we define $\mathfrak{h}(c) = c \bmod m$ as our hash function.

The last detail that we have to handle is the fact that multiple values of c may hash into the same position; for instance, c and $c+m$. To solve this problem, we assume that $t[i]$ consist of a tuple (s_i, h_i) where s is a stack and h is its height. Every c that hashes into position i is pushed on stack s_i (and

h_i is increased by 1). For the look up, we search the corresponding stack for the desired c. The algorithm is shown in Figure 13.

```
process(l)
  for i ← 0 to m − 1
    do t[i] ← (0, empty stack)
  for i ← 1 to n
    do for j ← i + 1 to n
         do c ←hypothenuse(l[i], l[j])
            if c ∈ ℕ
              then (h, s) ← t[ɦ(c)]
                   push(s, c)
                   t[ɦ(c)] ← (h + 1, s)

search(c)
  (h, s) ← t[ɦ(c)]
  for i ← 1 to h
    do if s[i] = c
         then return true
  return false
```

Figure 13: Using a hash table with the hash function $ɦ(c) = c \bmod m$.

If we assume that values of c are hashed uniformly in t, then each entry in t will have a stack of height $O(n^2/m)$. Therefore, it takes $O(1 + n^2/m)$ time to compute $ɦ(c) = c \bmod m$ and search the corresponding stack. With $m = kn^2$ for some k, this is $O(1)$ time.

5. A NOTE TO PYTHAGOREANS

Depending on when recursion is taught, the instructor can make a choice on whether to start with a recursive binary search, or an iterative one. The square root function and the hypothenuse problem are used as means to introduce recursion and tie it to the problems discussed thereafter, but recursion can be illustrated in many traditional ways. Stacks are great structures to play with, and crucial for understanding recursive processes. For the beginners, experimenting with arrays has an important role in providing high level structures, such as the stack. Once familiar, a stack is the perfect tool to accompany recursion. Hash tables are widely misunderstood, but perhaps the most important data structure in everyday computing, especially when it comes to problems of search. The illustration using Pythagorean triples gives the precise amount of intuition needed for considering hash tables and how they work.

All the problems discussed herein can be extended. For example, upon finding that c makes a Pythagorean triple in the list, one may ask to produce a witness, i.e. a and b in the list such that $a^2 + b^2 = c^2$, or to produce all of them (with or without repetition). Depending on the level of treatment of hash tables, different hash functions may be considered. Various aspects of Pythagorean triples can also be explored, such as primitive Pythagorean triples (when a, b, and c are co-primes). In addition, problem targeted for the discovery of properties of Pythagorean triples will provide an opportunity for cultivating the computational thinking process. For instance, are there Pythagorean triples that have exactly one value in common or exactly two values in common?

6. CONCLUDING REMARKS

The Pythagorean theorem $a^2 + b^2 = c^2$ is perhaps one of the most known mathematical results among the entire population of Earth (another is probably $E = mc^2$). Surprisingly, given the ease of proving it, only a few know of a proof. For many proofs, see [2]. In [4], Dijkstra generalized the theorem of Pythagoras for any triangle as $\text{sgn}(a^2 + b^2 - c^2) = \text{sgn}(\alpha+\beta-\gamma)$, where the angles α, β, and γ lie opposite to the corresponding sides a, b, and c, and sgn is the signum function ($\text{sgn}(0) = 0$ and $\text{sgn}(x) = |x|/x$ for $x \neq 0$). Newton's method (also known as Newton-Raphson) is well known in numerical analysis [6]. An excellent treatment of stacks and recursion, including tail recursive and iterative implementations, can be found in [1]. Binary search consists of a few lines of code, but it is very easy to get wrong. The best way (ever) to teach binary search is due to the followers of Dijkstra [5]. The CLRS book Introduction to Algorithms [3] (among others) contains detailed material on hash tables and their analysis, as well as basic and advanced topics in algorithms and data structures.

7. THE AFTERMATH

I envision this hypothetical, but concrete, Pythagorean story as a lecture in a second programming course. Many typical topics are introduced in a non-typical way (especially stacks and binary search). First, this emphasizes a real problem solving driven approach. Second, it provides a broader context for existing concepts: a stack may be a bit more than just a FIFO (e.g. when insertion time is a concern but not the access pattern), and binary search is not confined to sorted arrays. Third, some mathematical and formal concepts are infused into the tasks of writing programs. I believe we are in great need, more than any time, to rescue the declining mathematical level in computer science education. This is one attempt to do so by introducing computer science concept in a mathematical context that remains the focus throughout the entire lecture.

On a different note, I should say that the approach outlined herein prepares a general population of students who, though may not be academically savvy, are career oriented. For one thing, many respectful software companies, for example Google, follow a similar approach for their interview process. They expect the applicant to first solve a given problem in a trivial way, with no regards to the efficiency or the elegance of the solution. They then raise the level by guiding the applicant to pursue a certain direction of thought, hopefully leading to a better solution.

The importance of this exposition lies in the fact that one should be able to engage students in a lively environment where the need for a solution triggers the creation of structures and paradigms, ones they **have possibly seen before**, but **did not imagine their general utility**. As such, I believe students will be given the chance to appreciate how structures and techniques become useful outside their mundane usual settings. Therefore, readers should adapt this in any way they find useful for their own population of students. The topics can be tailored to any level. What I want to convey is not the specificity of the illustration, but a general approach to enlighten the students with mathematical concepts and new perspectives, invite them to think critically, and help them realize that computer science

is not just about writing code.

8. REFERENCES

[1] H. Abelson and G. J. Sussman. Structure and interpretation of computer programs, second edition. *The MIT Press. Also available at https://mitpress.mit.edu/sicp/full-text/book/book.html*, 1999.

[2] A. Bogomolny. Pythagorean theorem. *http://www.cut-the-knot.org/pythagoras*.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to algorithms, any edition. *Mc-Graw Hill*.

[4] E. W. Dijkstra. On the theorem of pythagoras. *http://www.cs.utexas.edu/users/EWD/ewd09xx/EWD975.PDF*, 1996.

[5] N. van Gasteren and W. Feigen. The binary search revisited. *http://www.mathmeth.com/wf/files/wf2xx/wf214.pdf*, 1995.

[6] Wikipedia. Newton's method. *http://en.wikipedia.org/wiki/Newton's_method*.