1. Introduction:
In 1993, investment banker and math enthusiast Andrew Beal devised a generalization of Fermat's Last Theorem [1]. This generalization is now referred to as Beal's Conjecture. Twenty years or so later, the Conjecture continues to be an elusive problem in number theory. Anyone who can either prove or disprove Beal's conjecture is up for a monetary prize of one million dollars [2].

2. Beals Conjecture
Beal's Conjecture states that if
$A^x + B^y = C^z$ (where $A, B, C, x, y, z \in \mathbb{Z}^+$ (positive integers) and $x, y, z > 2$),
then A, B, C must have a common factor [2].

Examples:
$3^4 + 7^2 = 3^4$ (does not qualify, as we are raising to a power less than 3)

$3^3 + 6^3 = 3^5$ $A = 1 * 3 = C,$ $B = 2 * 3 \rightarrow A, B, C$ share a common prime factor of 3.

$33^5 + 66^5 = 33^6$ 11 is a prime factor for $33, 66$ and 33.

$17^x + 18^y \neq 19^z$ 17,18 and 19 have no common prime factors.
GCD(17,18) = GCD(17,19) = GCD(18,19) = 1. No solutions exist according to Beal's Conjecture (where $x, y, z \in \mathbb{Z}^+$ with $x, y, z > 2$).

In order for Beal's Conjecture to fail we would need to find a combination of six numbers: $A, B, C, x, y, z \in \mathbb{Z}^+$ with $x, y, z > 2$ such that $A^x + B^y = C^z$.
Additionally A, B and C must not share common prime factor (coprime); or equivalently their greatest common divisor (**GCD**) is 1 .
$GCD(A, B) = 1$ or $GCD(A, C) = 1$ or $GCD(B, C) = 1$.

3. Roadblocks to Finding a Proof
Counter-examples to Beal's Conjecture come in two varieties: We can have a solution to the system where A and B are coprime, or we can have a solution to the system where A and B have a common factor but don't have a common factor with C.

First, we consider a system where A, B have no common factors (coprime).
Let's assume that $A^x + B^y = C^z$ is true for some positive integers C, x, y, z with $x, y, z > 2$.
Since multiplication is simply sequential adding up to the multiplier a number of times, we know we can define multiplication as addition. For example:
$3^2 + 4^2 = 5^2 \rightarrow 3 * 3 + 4 * 4 = 5 * 5 \rightarrow$
$(3 + 3 + 3) + (4 + 4 + 4 + 4) = (5 + 5 + 5 + 5 + 5) \rightarrow 9 + 16 = 25$
So if we have $A^x + B^y = C^z \rightarrow A * A * A... * A + B * B... * B = C * C... * C \rightarrow$
$(A+A+...+A)*A*A...*A+(B+B+...+B)*B*B...*B = (C+C+...+C)*C*C...*C$

$A^{x-1}A + B^{y-1}B = C^{z-1}C \longrightarrow \sum_1^{A^{x-1}} A + \sum_1^{B^{y-1}} B = \sum_1^{C^{z-1}} C$

We know that C^z is a simply a scaled version of C. ($C^z = C^{z-1}C$).
Next we should a find a linear combination of A and B such that:
$pA + qB \equiv 0 \bmod C$ for some $p, q \in Z^+$
$(pA + qB) = rC$ (for some $r \in Z^+$ p $\leq A^{x-1}, q \leq B^{y-1}$)
So our system can be expressed as:

$$A^x + B^y = C^z$$
$$(A + A... + A) + (B + B... + B) = (pA + qB)_1 + (pA + qB)_2 + ...(pA + qB)_n$$
$$= n(pA + qB)$$
$$= n(rC)$$
$$= C^{z-1} * C$$
$$= C^z.$$

So if we're dead center on finding a solution there must exist a linear combination $pA + qB$ that is congruent 0 mod $C \leftrightarrow pA + qB = rC$. Then we have to find powers x, y so that we have a precise ratio of A and B. $A^x + B^y = n(pA + qB) + 0$. The sum of all these linear combinations $n(pA + qB)$ has to equal $(C^{z-1})C$. There is too many moving parts. As of today, there is no mathematics that can guarantee or refute that these six distinct numbers can be a solution.

Next, we consider the scenario when A and B have a common factor. Let's assume that $A^x + B^y = C^z$ is true for some positive integers C, x, y, z with $x, y, z > 2$.
Let $k_1, k_2, ..., k_n$ be the common prime factors of A and B.
$A = k_1k_2...k_na$ and $B = k_1k_2...k_nb$ where a and b are the other remaining prime factors to A and B respectively.
Also let $c_1c_2...c_m$ be the unique prime factorization of C.
$A^x + B^y = C^z$ (where $x \leq y$)
$(k_1k_2...k_na)^x + (k_1k_2...k_nb)^y = C^z$
$(k_1k_2...k_n)^xa^x + (k_1k_2...k_n)^yb^y = C^z$
$(k_1k_2...k_n)^x(a + (k_1k_2...k_n)^{y-x}b^y) = C^z$
$(k_1k_2...k_n)^x(a + (k_1k_2...k_n)^{y-x}b^y) = (c_1c_2...c_m) * (c_1c_2...c_m)... * (c_1c_2...c_m)$ (z-times)
Hence, $(k_1k_2...k_n)$ divides $(c_1c_2...c_m) * (c_1c_2...c_m)... * (c_1c_2...c_m)$. Therefore the prime factors of A and B will divide numbers which are prime factors in C. We may conclude that if $A^x + B^y = C^z$, where A and B share a common factor, then C also shares a common factor with both A and B. Counter-examples to Beal's Conjecture cannot be produced under these parameters.

4. Search for Counter-examples:
Since we can't conclude that the Conjecture is indeed true, the next step is to perform an exhaustive search. We could never try all combinations to the System $A^x + B^y = C^z$. Nonetheless, we can search through for possible insights, solutions and near solutions. Alternatively, we can also search for counter-examples to the conjecture. The key is to build an efficient algorithm that only searches where counter examples may lie [7]. It would also be prudent to build a second algorithm to seek solutions that adhere to the conjecture's constraints [5].

The Algorithm

Beal's Conjecture has an infinite number of combinations. Therefore, it is essential to have a maximum value for $C^z = A^x + B^y$ [5]. I will generate all A^x less than the maximum value entered by the user. I will store these values in an Array of linked Lists.

```
For example, below we have A^x less than 2^9 = 512 [5].
[arrayIndex = base = 1]: -->  [1] --> null
[arrayIndex = base = 2]: -->  [8, 16, 32, 64, 128, 256] --> null
[arrayIndex = base = 3]: -->  [27, 81, 243] --> null
[arrayIndex = base = 4]: -->  [64, 256] --> null
[arrayIndex = base = 5]: -->  [125] --> null
[arrayIndex = base = 6]: -->  [216] --> null
[arrayIndex = base = 7]: -->  [343] --> null
```

Next, I build a second data structure for all C^z less than the maximum number. The algorithm will be doing a lot searches. Hence, a hash table will be ideal for quick look ups [3]. Once the data structures are constructed, each element in the array of linked lists A^x will be added to every element in a second array of linked lists A^x. We will search for this sum in the hash table C^z.

Pseudo Code: [6]

```
for( Integer A^z : ArrayLinkedList)
    for( Integer B^z : ArrayLinkedList)
        sum = A^z +B^z
        if ( hashmap_Cz contains sum)
            if ( coprime(A,B,C)== true)
                  return counterExample
```

The Java implementation to come is more intricate than the pseudo-code above. Nonetheless, the basic idea is the same.

Implementation

In this section, I will provide a detailed walk-through of my Java implementation. My Program consists of four java classes. AnswerStructures.java is where we generate the data structures we alluded to in previous section. abstractfunctions.java holds some useful functions pertaining to greatest common denominator. Finally, BealsCounter.java puts everything together and loops through the data. If a counter example is found, a message will be triggered. Similar to BealsCounter, BealsSolver.java will also loop through data. The only difference is that it will print answers in accordance to Beal's Conjecture.

```
// The main class BealsCounter.java
public class BealsCounter {
  public static void main(String[] args) {
    final long startTime = System.currentTimeMillis(); //start timer
    abstract_functions test1 = new abstract_functions(); //create
       instances of other classes
    AnswerStructures datagetter = new AnswerStructures();
    Random rand = new Random();
    int numBits = 46; // find counter-examples up to number of
       bits, range 5 to 91
    HashMap<BigInteger,Integer> myHash =
       datagetter.HashedAnswers(numBits);
    LinkedList<BigInteger>[] vectorList =
       datagetter.ArrayLinked(numBits);
    TreeSet<Long> powerTree = datagetter.PowersList(numBits);
```

Next, I explain how the data structures myHash, vectorList and powerTree are put together. For now we jump to AnswerStructures.java.

```
 // AnswerStructures.java
 // Given number of Bits, we calculate the largest possible Integer
    (C^z) as well as the maximum base (C) for our hash table.
HashMap<BigInteger,Integer> HashedAnswers(int numBits) {
    BigInteger two = new BigInteger("2"); // user entered
    BigInteger maxNumber = two.pow(numBits -
       1).subtract(BigInteger.ONE);
    BigInteger maxBase = floor3rdRoot(maxNumber, 3); //[4.
       Boddington]
    //Any base larger than maxBase will be larger than maxNumber
       when cubed.

    HashMap<BigInteger,Integer> myHash = new
       HashMap<BigInteger,Integer>();
    // Hash table holding C^z as BigIntegers (Keys) and
       corresponding base C (values) [3.]
    BigInteger next, base;
    for (int i = 2; i <= maxBase.intValue(); i++) {// Raise base
```

```
        (i) to several powers, until it surpasses maxNumber. Add
        each entry to hash table.
        int power = 3;
        next = BigInteger.ZERO;
        while (maxNumber.compareTo(next) > 0) {
          base = BigInteger.valueOf(i);
          next = base.pow(power);
          if (maxNumber.compareTo(next) < 0)
            break;
          if (myHash.containsKey(next) &&
              test1.coprime(base.intValue(), myHash.get(next)))
            throw new IndexOutOfBoundsException("Different coprime
              Bases yield the same result. \nRewrite Hashtable or
              data structure to accept duplicate answers.");
          myHash.put(next, base.intValue());
          power++;
        }
      }
      return myHash;
    }


// In the following array of linked lists we store all valid A^x
// Each linked list contains the same base (A) being raised to
   different powers until we have surpassed maxNumber. The Base is
   given by the array index. [5.]
LinkedList<BigInteger>[] ArrayLinked(int numBits) {
      BigInteger two = new BigInteger("2");
      BigInteger maxNumber = two.pow(numBits -
         1).subtract(BigInteger.ONE);
      BigInteger maxBase = floor3rdRoot(maxNumber, 3);
      LinkedList<BigInteger>[] vectorList = new
         LinkedList[maxBase.intValue()];
      BigInteger next;
      int power;
      vectorList[0] = new LinkedList<BigInteger>();
      vectorList[0].add(BigInteger.ONE); // 1^x must be checked.
      for (int i = 1; i < vectorList.length; i++) { //for each base
         2... maxBase
        next = BigInteger.ZERO;
        power = 3;
        while (maxNumber.compareTo(next) > 0) { // build linked list
           of a single base to several powers
          if (vectorList[i] == null) {
            vectorList[i] = new LinkedList<BigInteger>();
          }
          BigInteger base = BigInteger.valueOf(1 + i);
          next = base.pow(power);
```

```
          vectorList[i].addLast(next);
          power++;
        }
        vectorList[i].removeLast(); // remove entry bigger than
           maxNumber.
      }
      return vectorList;
    }


//Some entries in the array linked list are represented more than
   once. This happens when bases share unique prime factorization
   with one being raised further (powers). For example 8 = 2^3. So
   any power of 8 can be represented as (2^3)^x. All Base 8 powers
   are contained in Base 2 linked List . Once we identify these
   duplicate bases, we will store them in a red-black Tree.
TreeSet<Long> PowersList(int numBits) {
     BigInteger two = new BigInteger("2");
     BigInteger maxNumber = two.pow(numBits -
        1).subtract(BigInteger.ONE);
     BigInteger maxBase = floor3rdRoot(maxNumber, 3); //[4.
        Boddington]
     BigInteger maxSquare = floor3rdRoot(maxBase, 2); //any base
        larger than maxSquare will be larger than maxBase when
        squared.
     TreeSet<Long> squares = new TreeSet<Long>();
     for (int i = 2; i <= maxSquare.intValue(); i++) {
       long next = 0, raised = 2;
       while (next < maxBase.intValue()) { // Storing numbers that
          have integer roots.
         next = (long) Math.pow(i, raised);
         if (next > maxBase.intValue())
           break;
         squares.add(next);
         raised++;
       }
     }
     return squares;
   }
```

We now jump back to BealsCounter.java

```
// BealsCounter.java continued...
// Here we match up A^x + B^y then we look up the sum in the
   hash-table.
 int numSoln = 0, powerB, powerA = 2;
     BigInteger raised_A, sum;
```

```
for (int a = 0; a < vectorList.length; a++) { //for each A^x
  if(powerTree.contains((long) a+1))
    continue; // Bases with integer roots are not considered,
       they are duplicates.
  if(rand.nextInt(60)== 49) // Informs the user where the
     program is.
    System.out.println("Searching Base " + a + " out of " +
       vectorList.length);

  ListIterator<BigInteger> listIter_A =
     vectorList[a].listIterator();// get all A^x, for a
     particular base (a).
  powerA = 2;
  while (listIter_A.hasNext()) {
    powerA++;
    raised_A = listIter_A.next(); // Store current A^x.

    for (int b = a; b < vectorList.length; b++) {// Cut the
       search in half. Only consider A <= B, A and B are
       arbitrary. [3.]
      if (test1.coprime(a + 1, b + 1) == false)
        continue; // As we proved earlier, If counter examples
           do exist they must occur when A & B are coprime.
         //skip non coprime Bases [3.],[6.]

      ListIterator<BigInteger> listIter_B =
         vectorList[b].listIterator();
      powerB = 2;
      while (listIter_B.hasNext()) {
        powerB++;
        sum = raised_A.add(listIter_B.next()); //A^x + B^y, A
           & B are coprime
        if (myHash.get(sum) != null ) { // Potential counter
           example in our hashtable, Begin search [7.].
          prtCounter(a+1, powerA, b+1, powerB, sum,
             myHash.get(sum));
        }
      }

    }
  }
}

final long endTime = System.currentTimeMillis(); // stop timer
System.out.println("Total program run time: " + (endTime -
   startTime) + " milliseconds."
    + "\nTotal counter-examples found (up to 2^" + numBits +
       "): " + numSoln + " ");
```

```
   }
private static void prtCounter( int a, int powerA, int b,
       int powerB, BigInteger sum, int base) {
     System.out.println("Counter Example Found!! "
          + a + "^" + powerA + " + " + b
          + "^" + powerB + " = " + base + "^z = " + sum);
       System.exit(0);
   }

}// end BealsCounter.java
```

.

Results

I had originally written my program to solve for 32-bit integers (2^{32}). I ran the program on a home computer with a clock speed of 3.2 Ghz. My first run, found no counter-examples within 5 seconds. This prompted me to optimize my program and see how fast I could make it. The final program can search 42-bit numbers in under ten seconds. I took my program a step further. I ran the program up to 60-bits. For this search I used two computers in parallel. This allowed me to complete the search over a span of two days. Again, the conclusion was no counter-examples.

Another limitation of the program was memory usage. Any search for counter-examples larger 78-bits quickly runs out memory. It requires more than 8GB of Ram. Possible solutions to this problem would be to search for counter-examples in confined ranges one at a time [5]. I am currently storing my numbers BigIntegers. This java class stores numbers in 128-bits. Storing these numbers in 32-bit Integers using modular arithmetic, would save four times the space. Unfortunately, any such en devour will probably slow down my program.

Next Step

Lacking some revealing mathematical intuition, the best that can be done is a search for counter-examples. My program stonewalled past 60-bit numbers. The next logical step is to parallelize the code. Putting several machines to the task would increase the speed. Modern GPUs have hundreds of cores. This could also make them ideal for continuing the search [6].

Works Cited

1. "American Mathematical Society." *American Mathematical Society*. N.p., n.d. Web. 20 Dec. 2016. <http://www.ams.org/profession/prizes-awards/ams-supported/beal-prize>.
2. "Beal Conjecture." *Wikipedia*. Wikimedia Foundation, n.d. Web. 20 Dec. 2017. <https://en.wikipedia.org/wiki/Beal_conjecture>.
3. "Beal's Conjecture." N.p., n.d. Web. 8 Dec. 2016. <http://www.danvk.org/wp/beals-conjecture/>.
4. Boddington, Paul. "Calculating Nth Root in Java Using Power Method." N.p., n.d. Web. 11 Jan. 2017. <http://stackoverflow.com/questions/32553108/calculating-nth-root-in-java-using-power-method>.
5. Butler, Bill. "Beal's Conjecture." *Beal's Conjecture*. N.p., n.d. Web. 8 Dec. 2016. <http://www.durangobill.com/BealsConjecture.html>.
6. Desu, Noah. "Beals-conjecture." *GitHub*. N.p., 26 Jan. 2015. Web. 5 Jan. 2016. <https://github.com/noahdesu/beals-conjecture>.
7. Norvig, Peter. "Beal's Conjecture Revisited." *Beal*. N.p., 22 Oct. 2015. Web. 12 Dec. 2016. <http://norvig.com/beal.html>.