A POLYNOMIAL TIME HEURISTIC ALGORITHM FOR CERTAIN INSTANCES OF 3-PARTITION

A THESIS

FOR THE DEPARTMENT OF COMPUTER SCIENCE

AT BALL STATE UNIVERSITY

SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS

FOR THE DEGREE

MASTER OF SCIENCE

IN COMPUTER SCIENCE

BY

RONALD DOUGLAS SMITH

PROFESSOR DOLORES ZAGE - ADVISOR

BALL STATE UNIVERSITY

MUNCIE, INDIANA

MAY 2014

ACKNOWLEDGEMENTS

Special thanks to Professor Frank W. Owens who challenged the class to improve upon known algorithms for Karp's 21 original NP-Complete problems which inspired an epiphany and resulted in a new algorithm for 3-Partition.

Special thanks to Professor Wayne M. Zage for his masterful insight and direction in how best to write a research proposal and a thesis.

Special thanks to Professor Dolores Zage for her technical programming skills, strategizing skills and management expertise in crafting the thesis.

Special thanks to Professor Lan Lin for her insights into computational complexity and for her crucial help with improving the central ideas of the algorithm.

TABLE OF CONTENTS

I. Problem Description

i. NP-Complete Problems

There are problems in computational complexity so difficult that the only known ways to solve them are impractical. A problem that has a few hundred inputs may take more time to solve with a super computer than the time that has passed since the beginning of the universe. These problems, called NP-Complete problems, do have solutions. Some can be easily solved until a phase transition point is reached after which they become intractable. For others, the 'strongly' NP-Complete problems, there is no range of inputs that has known useful algorithms. 3-Partition falls into the second category.

Polynomial (P) time is considered to be a reasonable amount of time to solve a problem, and is bound by the number of inputs. Nondeterministic polynomial (NP) time is not since all known algorithms are exponential. When the number of inputs is large enough, n^{10} is faster than 10^n. Pseudo-polynomial time, though exponential, is bound by the number of inputs and the magnitude of the largest input. Pseudo-polynomial time falls somewhere between polynomial time and nondeterministic polynomial time.

The solution to an NP-Complete problem can be recognized by a nondeterministic polynomial (NP) time Turing machine. If an NP-Complete problem can be solved in polynomial (P) time (or pseudo-polynomial time for a strongly NP-Complete problem) then P = NP is true.

Problems in P are easy to solve while problems in NP are easy to check. Password permutations are an example of a problem in NP. It is easy to check to see if a password is correct, but it may require many permutations of characters to solve a password. If we were able to devise an algorithm that solved one NP-Complete problem in polynomial time that would imply that all NP-Complete problems have undiscovered 'reasonable' solutions because problems such as these can be reduced [19] to one another in polynomial time.

If P = NP is true, the implications are profound. A mathematical proof of 'reasonable' length could be computed by a program. Combinatorial problems such as recombinant DNA or logistics could be computed far more easily. Some applications that rely on problems that are easy to build but difficult to solve may suffer, such as security. Difficult problems would still exist but they may become nearly as hard to devise as they are to solve.

Even if P = NP is false, the NP-Complete class of problems is so pervasive that innovative workarounds for special cases are constantly being discovered. It is also known that some algorithms for NP-Complete problems exhibit exponential complexity only in the worst case scenario and in the average case can be solved with polynomial time algorithms [13]. One such class of NP-Complete problems is Partition. However, our specific problem 3-Partition is different from Partition in that it has no pseudo-polynomial time algorithm thus identifying it as strongly NP-Complete.

The focus of this thesis will be solving instances of the strongly NP-Complete problem 3-Partition. No attempt will be made to resolve the open question of P versus NP.

ii. 3-Partition

Informally, 3-Partition asks: Can you divide 3*m* inputs into *m* subsets of three elements each such that each subset sums to the desired amount and includes each input once? The 3-

Partition problem decides whether a set of non-negative integers (from Z) can be partitioned into triples that all have the same sum. The number of inputs (n) must be a multiple of three ($3m$ inputs). The sum of the inputs must be divisible by the number of multiples of three (the sum is divisible by m). Additionally, each and every input must be used in the solution exactly once.

More formally, the 3-Partition problem is described by authors Garey and Johnson in their book, Computers and Intractability, pg. 96 [11]:

Instance: A finite set A of $3m$ elements, a bound $B \in Z^+$, and a "size" $s(a) \in Z^+$ for each $a \in A$, such that each $s(a)$ satisfies $B/4 < s(a) < B/2$ and such that $\sum_{a \in A} s(a) = mB$.
Question: Can A be partitioned into m disjoint sets $S_1, S_2, \ldots, S_m$ such that, for $1 \leq i \leq m$, $\sum_{a \in S_i} s(a) = B$?

As an example, consider the set {1, 2, 3, 4, 5, …, 153} which sums to 11,781. To solve 3-Partition for this set, fifty-one subsets (3m = 153) consisting of three inputs each summing to an amount equal to the total sum divided by m (11,781 / 51 = 231) need to be created.

There are many possible approaches to solving this problem. One way is to iterate through every possible combination. Each integer can be assigned to 51 subsets, so that there are 153!/(51!3!) partitions to consider. This is a brute force algorithm. If we could compute a billion possible solutions per second, this problem would take many millennia to complete.

In this thesis, the algorithms created to solve 3-Partition are compared to the brute force algorithm. With the extreme slope of the exponential curve for the brute force algorithm, only small values of $3m$ can be visually observed. For larger values, the brute force search would not provide even a single solution within our lifetimes. This is true even with the advantage of trying only subsets of the inputs that are valid as part of a solution (and not all possible subsets).

It is often possible to apply some clever techniques to reduce the number of iterations of a brute force algorithm. The use of heuristics to guide the solution process and reduce work is one possibility. A heuristic approach that allows the algorithm to choose and discard subsets based

on the input characteristics of valid subsets of a solution instead of trying (exponential time) combinations can narrow down the search to a manageable space.

The goal of this thesis is to create a fast heuristic algorithm that finds an exact solution for certain instances of 3-Partition in polynomial time but in the worst case may not find a solution even though one exists. The current recursive algorithm has no known counter examples. We cannot confirm that a "no solution" result truly has no solution without the aid of a super-computer to try an exhaustive brute force search for problems that are limited to our observance space. Beyond a certain number of inputs, even a supercomputer would take years to confirm that no solution exists.

Comparison of the heuristic algorithm with a brute force search is impractical for more than a few inputs. Running a brute force search to enumerate the solutions for 27 inputs took 5½ days on an Intel i5 core 2.5 GHz laptop. The brute force algorithm from Reingold [27] runs in constant time per combination. By extrapolation, we can estimate that 33 inputs would take 3½ years to *count* all solutions. We shall revisit this particular problem later and demonstrate how long it takes with the newly created heuristic algorithm.

II. Research Objective

Strongly NP-complete problems are considered to be intractable for all ranges of inputs even if non-concise unary encoding is used and yet these problems come up in many practical applications. The goal of this research is to explore three algorithms we have implemented (including brute force) that can solve 3-Partition and examine the time complexity of each.

Solving 3-Partition for distinct inputs may help to solve other problems. 3-Partition is used to prove 'strong' NP-Completeness in the same way that Satisfiability is used to prove regular NP-Completeness. 3-Partition is reducible to other NP-complete problems in polynomial time. A usable range of inputs with solutions that can be reduced to solutions for other problems may provide helpful insights to researchers. New heuristics for solving other NP-complete problems may be discovered.

There are many practical applications. If a solution is found for a 3-Partition problem instance of two thousand or so inputs in less than six minutes, a supercomputer thousands of times faster could optimize a one billion transistor chip in a matter of weeks. The new faster chip could run the new algorithm and optimize a newer larger chip. Progress could continue until new limitations were encountered.

Scheduling jobs or server balancing could become easier. This means that access to the Internet and cloud computing could be faster. There are many other practical applications.

In principle, the 3-Partition Problem could be solved in exponential time by checking through all possible solutions, one by one as a brute force search. An algorithm that performs this method is all but useless in practice. We needed to find a way that did not involve trying combinations to find a solution because all known algorithms for combinatorial search are exponential. Identifying clever methods to bypass the process of combinatorial exhaustive search and using clues from the inputs in order to narrow down the search space is the practical objective of this research.

Our first heuristic algorithm used clues from the inputs and included a stack to make the search closer to exhaustive. Our second heuristic algorithm used clues from the inputs and accepts or rejects valid subsets recursively. The recursive version was rewritten as a tail recursion and converted to an iterative program for scalability and to avoid problems with Java heap size.

We shall provide a comparison of a brute force algorithm (which tries only valid subsets), the heuristic stack algorithm and the heuristic accept or reject recursive algorithm. Each program has been modified to stop when the first solution is found to make the comparisons valid. The time of execution is recorded after inputs are accepted and before the solution found is printed.

III. Definitions

3-Partition is a sequence of 3m nonnegative integers such that A = $\{a_1, a_2, ..., a_{3m}\}$ whose sum is equal to *m times B*. With 3-partition, there are *m* disjoint subsets of three elements each that exactly cover set A. Each of the *m* subsets of three elements must sum to *B*. The product of *m* and *B* must equal the sum of the elements of set A. For example, the set {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15} sums to 120 (*m* times *B*), a solution would have 5 subsets (*m*, since 3*m*=15), and each subset would sum to 24 (*B*). One such solution for this input is {{1, 8, 15}, {2, 10, 12}, {3, 7 14}, {4, 9 11}, {5, 6, 13}}. Five subsets each sum to twenty-four and every input is covered (used).

Heuristic is a set of rules to guide decision making in an algorithm.

Strongly NP-Complete means that the problem is considered to be intractable even when the concise encoding requirement is dropped and unary encoding is used [11]. 3-Partition is considered to be strongly NP-Complete whenever *m* is three or greater (three or more groups of three element subsets in a solution).

Pseudo-Polynomial Time is exponential but growth of such a function is limited by the magnitude of the largest input, t = O (*mB*) [11]. The size of *B* becomes a factor when the number of bits needed to encode the problem exceeds the number of inputs.

Subset Ranking is a term used for this research. To rank the subsets, count the overall occurrences of each unique element within all of the valid subsets. Each element in a subset then contributes that occurrence amount to the rank of an individual subset.

Lowest Frequency Element is a designation given to the input element(s) with the fewest occurrences that is (are) still available for building solution subsets from valid subsets that have not been eliminated or previously selected.

Multi-sets are subsets that contain duplicate input elements.

Subset States track the state of each subset during processing. A value of 0 is available, 1 is a primary subset, 2 is of interest, 3 is non-determined, 4 is eliminated, and 5 is finished.

Lexicographical Order is one of the criteria used for creating the sort order of the valid subsets. Subset states and rank are the other criteria.

Complex Ranking is a way to rank multi-sets so that a second or third occurrence of the same element within a subset does not have an undue influence on ranking. The number of first, second or third occurrences of duplicated inputs would instead be the contribution for that element to the ranking, rather than the total occurrences (also called positional ranking).

For example, the inputs {0, 0, 1, 1, 1, 2, 2, 2, 3} would have the valid subsets { {0 1 3} {0 2 2} {1 1 2} }. The resulting frequencies based on the supplied solution with complex rank are first 0's (two), first 1's (two), second 1's in a subset (one), first 2's (two), second 2's in a subset (one) and 3's (one). Complex ranks for each subset would be { {0 1 3}(two + two + one) {0 2 2}(two + two + one) {1 1 2}(two + one + two) }. Without complex rank, the rank would be { {0 1 3}(two + three + one) {0 2 2}(two + three + three) {1 1 2}(three + three + three) }.

Java BitSet is a vector of ones and zeroes that grows as needed. A BitSet is initially created at a default size depending on the system but grows dynamically as more bits are set. The BitSet is considered to be empty if it is composed entirely of zeroes. A single BitSet object holds the subset states stored for refreshing the states of the subset array when we pop the stack that we fashioned from a BitSet (see Appendix E and Appendix F). Four bits are enough to represent each state of a subset since that last bit must be a one to be easily found (there is a java BitSet method that detects the last bit set to 1). Available = 0001, primary = 0011, of interest = 0101, nondetermined = 0111, eliminated = 1001 and finished = 1011. $4m$ bits (in one BitSet object) can represent each block of subset states instead of m Java Integer objects pushed onto an object stack.

IV. Literature Review

There are few examples in the literature of solutions of instances of 3-Partition. Most of these are m=2 solutions, a special case that is not strongly NP-Complete. For example, the set {2, 3, 4, 5, 7, 9} resolves to a solution {{2, 4, 9}, {3, 5, 7}}. When only two 'buckets' are used, the problem is equivalent to Partition which is considered by many to be tractable over a wide range of inputs [10, 13]. A thesis by Joosten [18] considers six buckets for instances of 3-Partition with a relaxed definition 3-Partition. Joosten's relaxed definition allows inputs that are duplicates and zeroes.

It is important to know the worst case execution time of an algorithm compared to the number of inputs. Polynomial time is considered to be a reasonable amount of time for an algorithm. Software and algorithmic methods exist to aid this calculation [7, 17]. Time and space usage are the most critical statistics for evaluating an algorithm, and are related [22]. One solution may be to implement a Java BitSet as a stack [21]. In our implementation the state of each subset would be represented by four bits rather than a Java integer object therefore saving space (the BitSet is a single object of 4m bits instead of m^2 Java objects that would be stored for each potential solution). We have not implemented BitSet for our recursive lowest rank version.

A dynamic programming algorithm exists for a special case of Partition which runs in O (n^2) time [15]. If the number of bits of the largest input is bound, the number of blocks is less

than or equal to the number of data points, and the data space is one dimensional then the dynamic programming algorithm executes in polynomial O (n^2) time.

For the Partition problem, researchers have identified a phase transition [12, 24]. Partition is easy to solve until the number of bits needed to represent the inputs exceeds the number of inputs. How a set is divided into subsets or dimensionality may be a factor when a problem goes from P to NP in its complexity [24]. Partition could be considered to be in P until the phase transition [13] which occurs when the number of bits to describe the problem exceeds the number of bits to describe the inputs. One dares to speculate that Partition to 3-Partition (two buckets to three buckets) may also be a phase transition to strongly NP-Complete.

Multi-sets and disjoint sets (no duplicates) of 3-Partition are both strongly NP-Complete. Ramamoorthy explored this topic in his thesis, 3-Partition Remains Intractable for Distinct Numbers, [26]. A paper by Gent, et al., states that heuristics should be used to approximate results except for small inputs since 3-Partition is a number problem that is strongly NP-Complete [12]. In Gent, no indication as to what constitutes a small or large input was given.

Many examples exist of proofs that 3-Partition is indeed intractable (strongly NP-Complete) [8, 9, 10, 11, 26]. There have been no published solutions of larger instances (seven or more buckets) of 3-Partition as of this writing. We have solutions for as many as 3333 buckets. Korf presented a 100 input problem solved for Partition [21] which is defined as an NP-Complete problem after the phase transition. His algorithm provides an approximation. In contrast, our algorithm tests for an exact solution. In addition, we are solving 3-Partition which is strongly NP-Complete not simply NP-Complete.

Examples from related NP-Complete problems are cited in the references [3, 5, 19]. All NP-Complete problems are reducible to one another by a polynomial transformation [19]. Many

of the transformations are well known. Joosten in his thesis solves his version of 3-Partition by transformations to Bin Packing, Graph Traversal, and 3-Dimensional Matching [18]. Parts of solutions to these problems could possibly be applied to 3-Partition.

Combinatorial algorithms are the exponential or brute force way to solve NP-Complete problems [11, 21, 27], CKK or Complete Karmarkar Karp is 2^n worst case time. Models of the time required for some ranges of inputs to combinatorial algorithms can be expressed as exponential or even factorial time. Pseudo-polynomial approximations can be calculated for discrete intervals [15]. Described in Jackson, et al., this generalization of the traveling salesman problem shows how complexity grows with the degree of the problem similar to the difference between Partition and 3-Partition.

A two-dimensional graph traversal algorithm for partitioning a plane uses rank to coordinate the topological connections [25]. Rank is an important factor for our recursive algorithm and is used to determine the order in which subsets are chosen for potential solutions.

An algorithm is described for finding frequents elements in streams, as in streaming media, and bags, as in offline collections, that works in two passes [20]. We seek the lowest frequency element and perhaps this idea could be adapted to serve in our algorithm.

A dynamic programming algorithm exists that solves a special case of 3-Partition [6]. The authors describe forced triples from which are drawn the solution subsets (we have a similar routine). This algorithm can find a solution if the subsets selected contain as a member the only element of its kind in the entire inputs list. This suggests that $m/3 - 1$ singleton elements must exist in the valid subsets before the solution can work. The last selection is deterministic.

A combination of pebbles and branching [5] and subsets of structured sets [23] may help us to optimize the non-deterministic portion of our algorithm. We suspect that this portion of our

algorithm can be better designed and is a topic for further research. Other topics for further research include a comment by Dyer, et al., stating that 3-Partition is strongly NP-Complete when $B = \Omega(m^4)$ and referencing Garey and Johnson, Computers and Intractability. B is the sum required for each subset in a 3-Partition solution. We could not find the original reference within Garey and Johnson. The derivation and rationale are of great interest to this thesis.

Johnson has practical advice on the experimental analysis of algorithms [17]. We will follow Johnson's advice and report on all findings.

A new brute force unary bitwise encoding of 3-Partition [16] has been recently developed which in theory would be much faster than a decimal brute force search. We were unable to evaluate this algorithm since the required Appendix A was not included with the publication.

V. Importance of the Study

What is the most important problem in computer science? Solving *any* NP-Complete problem could lead to faster processor speed and server balancing (internet speed), faster and easier modeling of genetics and pharmaceuticals (combinatorial problems). Computational Complexity affects every one of these things and much more. Because NP-complete problems are reducible to one another, a solution for some instances of 3-Partition could help us solve problems many of which we have yet to conceive. If a constructive proof of P = NP was found, there would be an explosion of discovery in every science that benefits from computers, especially computer science and mathematics.

Optimization problems such as logistics would become easy to solve. Computer programs could prove or disprove finite mathematical theorems. The advances in computer science and mathematics would push advances in any science that uses computers or math to solve complex combinations of variables. The magnitude of such a discovery cannot be overstated. It may be the most important known problem left undecided.

An algorithm that provided exact solutions for certain instances of 3-Partition, a strongly NP-Complete problem, could provide some of the benefits of P = NP.

VI. Research Design

i. Introduction

A daunting task when introducing an algorithm for an NP-Complete problem is proving the algorithm's efficiency. In fact it may not provable one way or the other. Let us summarize the learned opinions of selected experts in the field for problems related to 3-Partition.

About P vs. NP, Cook [4] says:

"Most complexity theorists, including the author, believe that P ≠ NP." ... "Millions of smart people, including engineers and programmers, have tried hard for many years to find a provably efficient algorithm for one or more of the 1000 or so NP-Complete problems, but without success."

If an efficient algorithm *was* found, Cook's [4] comments are not understated:

"If P=NP is proved by exhibiting a truly feasible algorithm for an NP-Complete problem" ... "the practical consequences would be stunning."

Part of the problem in finding a "provably efficient algorithm" could be that the question itself 'does P = NP?' is not decidable. Aaronson [1] states:

"If P vs. NP were independent"… (and SATISFIABILITY was solved as polynomial) … *"there would be such an algorithm, but it would be impossible to prove that it works."*

In addition, Aaronson says [1]:

"P ≠ NP is either true or false" … "But we may not be able to prove which way it goes, and we may not be able to prove that we can't prove it."

It is possible that if we find an algorithm with polynomial average case time we will be unable to prove that it always works. The algorithm may work efficiently in many cases but

worst-case time complexity may still be exponential. An algorithm that solves 3-Partition most of the time in an efficient average case time could realize some of the benefits of P = NP. This is one of the five possible worlds of P vs. NP called Heuristica described by Impagliazzo [14]:

"Heuristica is in some sense a paradoxical world. Here, there exist 'hard' instances of NP problems, but to 'find' such hard instances is in itself an intractable problem!"

Impagliazzo [14] goes on to say that:

"... Heuristica is basically equivalent to knowing a method of quickly solving almost all instances of one of the average-case complete problems ... and having a lower bound for the worst-case complexity of some NP-Complete problem."

Is there hope that Heuristica can be accomplished? In Garey and Johnson, Computers and Intractability, pg. 106 [11], the strong NP-Completeness result for MULTIPROCESSOR SCHEDULING where n is the number of tasks, m is the number of processors and L is the length of the longest task *rules out* a polynomial time solution in n, m and log L (NP-Completeness) and in n, m and L (strong NP-Completeness) unless P = NP. However:

"Our subproblem results do not rule out an algorithm polynomial in m^n *and log L,"* (where n is fixed) *"and indeed exhaustive search algorithms having such a time can be designed." … "It leaves open the possibility of an algorithm polynomial in* $(n\ L)^m$ *(which would give a pseudo-polynomial time algorithm for each fixed value of m), and again such an algorithm can be shown to exist."*

Is it possible to improve upon the times known to be possible quoted above? In Garey and Johnson, Computers and Intractability, pg. 122 [11]:

"... it is sometimes possible to reduce substantially the worst case time complexity of exhaustive search merely by making a more clever choice of the objects over which the exhaustive search is performed."

In the case of our algorithms, we believe that the order of choice is most important.

The preceding statements demonstrate how difficult it is to address the P vs. NP question. The goal of our inquiry is less ambitious. We intend to compare our new heuristic algorithms for 3-Partition with our original algorithm and with a brute force algorithm that has been given the advantage of trying only valid subsets.

ii. Thesis Statement

A naïve brute force search that finds solutions to 3-Partition for a given instance of 3*m* inputs would consider *every possible subset* of three elements each, taken from the set of inputs, and then every grouping of *m* subsets from the list of every possible subset. A more efficient way would be to reduce the subsets considered to only those subsets that sum to the amount *B* required for 3-Partition. This is a more clever brute force search, but it still utilizes combinations which are exponential in nature.

We have found an approach that provides an alternative to relying on combinations. A recursive approach may help to avoid a naïve exhaustive search. Element frequency and subset rank are metadata contained within the valid subsets. For example: for inputs 1, 2, 3, 4, 5, 6, 7, 8, and 9 the valid subsets would be {1,5,9} {1,6,8} {2,4,9} {2,5,8} {2,6,7} {3,4,8} {3,5,7} {4,5,6}. Within the valid subsets; 1 occurs twice, 2 occurs three times, 3 occurs twice, 4 occurs three times, 5 occurs four times, 6 occurs three times, 7 occurs twice, 8 occurs three times, and 9 occurs twice. To rank subsets, count occurrences of each element in the valid subsets. Each element then contributes that amount to the rank of an individual subset.

The ranks would be: {1,5,9}(8) {1,6,8}(8) {2,4,9}(8) {2,5,8}(10) {2,6,7}(8) {3,4,8}(8) {3,5,7}(8) {4,5,6}(10). Select solutions by highest rank first. If we choose {2,5,8}, we must eliminate {~~1,**5**,9~~} {~~1,6,**8**~~} {~~**2**,4,9~~} {2,5,8} {~~**2**,6,7~~} {~~3,4,**8**~~} {~~3,**5**,7~~} {~~4,**5**,6~~} or all other valid subsets because 3-Partition requires that each input is used exactly once in a solution.

Select subsets by lowest rank. If {1,5,9} is selected we must eliminate {1,5,9} {~~**1**,6,8~~} {~~2,4,**9**~~} {~~2,**5**,8~~} {2,6,7} {3,4,8} {~~3,**5**,7~~} {~~4,**5**,6~~} which leaves a solution {1,5,9} {2,6,7} {3,4,8}. The order of selection is important because we must eliminate any other subsets that contain elements from a subset we select. In addition, selecting subsets with low frequency elements

early in the process makes it less likely that subsequent choices will eliminate an element that must be included in a solution. One dares to postulate that if we always knew the correct order in which to select valid subsets, we would always find a solution if one exists.

Assume that the order of subset selection has no effect on the general problem of solving an instance of 3-Partition without using brute force combinations. A null hypothesis, using this assumption could be stated as:

> H_0 = subsets may be selected in any order when selecting or eliminating subsets when finding a recursive cover solution for 3-Partition.

If H_0 is unsupported, then our assumption is incorrect. In fact H_0 is not supported, and therefore the order of selection is important. If we wish to avoid an exponential time brute force search, we must instead make a *clever orderly* search.

There exists solvers for 3-Partition that solve 3-Partition in less than exponential time for special cases. One of these solvers was proposed by Dyer, et al., [6]. This algorithm runs in n^2 time and finds solutions if enough valid subsets, (stated as forced triples with one of the input elements with a frequency equal to one) exist to construct a solution. Our algorithm is also a special case but has succeeded without relying on finding subsets containing an input element with a frequency of one and is broader in that sense.

iii. Methods:

Each method of reducing all possible subsets of the inputs systematically to produce a solution for a given instance of 3-Partition will be explained in this section.

1. The Valid Subsets Reduction

Solving the subproblem of which valid subsets can be accepted as part of a solution is an important step. Subtract the smallest element and the largest element from B(the sum required for each subset) to obtain a remainder. Then search the sorted elements in decreasing order until

a subscript with an element of a 'size' equal to the remainder value is found or an element with a 'size' less than the remainder is found. This process continues until $3m-1$ operations have occurred. We increase the smallest subscript by one. The remainder subscript is decreased until next remainder value is found. Each subsequent pass has one less operation, $3m-2$, $3m-3$, etc. until $3m - {}^{3m}/_{2}$ operations are performed. The time function for the number of operations performed by this method is $t(m) = 3^{3}/_{8}m^{2} - 1^{1}/_{2}m + {}^{1}/_{8}$ resulting in O (m^2) time. We use $3m$ here to represent the n inputs of 3-partition. Appendix A has detailed descriptions of the definition of valid subsets, the derivation of valid subsets and the derivation of the time function for valid subsets.

2. Lowest Ranked Subset

Derive valid subsets from the input elements. Collect element frequency and calculate the subset rank. Sort the valid subsets by subset state, by rank then by lexicographical order. Select the first subset from the list of valid subsets. Remove the elements of the selected subset from the input list (by changing their state). Save the selected subset. Repeat the process until a solution is found or less than m subsets are available making a solution no longer possible.

3. Orderly Search

There will be m iterations of the Lowest Ranked Subset process. The first subset is always selected at each iteration. If no solution is found, the very first subset tried is retired and we try again with the next lowest ranked subset.

iv. Implementation

The Valid Subset Reduction is processed first. From the valid subsets, select the lowest ranked subset. The orderly search begins by marking the very first subset selected as primary.

There is often not enough information available to always solve 3-Partition in a single

pass (*m* recursions). Counter-examples have been found. That is why there needs to be an orderly search which we accomplish by retiring the primary subset. The clever selection of the search objects has made it possible to solve certain instances of 3-Partition, currently up to 9999 consecutive inputs as of this writing.

The elements within the valid subsets are not likely to be evenly distributed. Each subset can be ranked by the number of elements it has in common within itself and within the remainder of the set of available subsets. The rank can be used to determine the order in which subsets are chosen while trying to build a 3-Partition solution. One improvement that will be added at a future date is handling of multi-sets and zeroes as inputs.

VII. Conclusions and Discussion

The Brute Force Search

Until now except for narrow special cases, no algorithm existed for solving 3-Partition except in exponential time. Our algorithm also solves a special case of 3-Partition. It has been tested most often with consecutive inputs {1, 2, 3, 4, … 3*m*} and also with random inputs.

The following table describes the brute force algorithm:

number of inputs	solutions found	solution subsets (*m*)	valid subsets(v)	number of combinations(v choose m)
9	2	3	8	56
15	11	5	25	53,130
21	84	7	50	99,884,400
27	1296	9	85	411,731,930,610

The jump in execution time for finding *the first solution* from 21 inputs to 27 inputs was from 9.35 seconds to 10 hours, 12 minutes and 58 seconds. We did not attempt 33 inputs. Perhaps this is what Dyer meant when stating that 3-Partition is strongly NP-Complete when $B = \Omega(m^4)$. One could infer that 411 billion combinations imply a strongly NP-Complete problem. The brute force algorithm from Reingold, pg. 181 [27] finds combinations in constant time. We can extrapolate that 33 inputs would require years (128 choose 11). Clearly, even with the advantage of trying only valid subsets, the brute force approach is unacceptable even though it is always guaranteed to (some day) find a solution if one exists.

The Orderly Search

The orderly search sorts a space of $^9/_8m^2 - {}^3/_4m + {}^3/_8$ valid subsets up to m times. All other calculations are m^2 in time or less. Worst case time for the inner loop in terms of m where m is the number of solution subsets desired for 3-Partition is $t(m) = O(m * (m^2 \log m^2))$. Time $t(m)$ resolves to a worst case time of $O(m^3 \log m)$. We seldom have had to execute m recursions m times to find a solution. Retirement of the primary subset means that the $m^3 \log m$ process could occur up to $m^2 - m - 1$ times for a total worst case time of $O(m^5 \log m)$.

The program listed in Appendix C is *proof by construction* that recursive selection of the lowest ranked subset is an efficient (if not complete) heuristic algorithm for solving 3-Partition. Additional evidence is the very large solution for 2523 consecutive inputs listed in Appendix G.

The orderly search is not exhaustive. The program in Appendix C was first designed to solve 3-Partition in a single pass (m recursions). However, a small counter example was found that failed: {2, 3, 6, 8, 10, 14, 16, 18, 19, 21, 22, 24, 27, 30, 31, 32, 34, 36, 37, 38, 41}. This counter example also proved that the algorithm is not greedy. Simply choosing the lowest ranked subset in lexicographical order recursively does not guarantee a solution will be found if one exists. Several large counter examples were also found.

The addition of a second loop that refreshes the status and rank of all but the very first (primary) subset selected for a solution is exhaustive in checking the *primary subset only*. This addition allowed the algorithm to solve all currently known counter examples.

An earlier and much slower (exponential time, but faster than brute force) version of the program based on element frequency found solutions including a solution for 2523 inputs which let us know that certain solutions did indeed exist.

The orderly search algorithm is not an estimate, it finds an exact solution. The algorithm

is set up for disjoint sets, not multi-sets. It is not a *complete* algorithm in that we cannot *prove* that a solution will always be found if one exists.

The largest published solution found as of this writing was 6 subsets (18 inputs) by Joosten [18]. 2523 inputs were solved by our algorithm in just over six minutes. The 2523 limit was close to the limit for the earlier stack version of the algorithm. More recently 9999 inputs have been solved in just over 15 days.

Table 1: Comparison of Execution Times for Three Algorithms in Seconds

inputs	brute force	stack	recursion
9	0.006	0.005	0.005
15	0.095	0.010	0.007
21	9.348	0.033	0.010
27	36778.314	0.050	0.016
33	-	0.078	0.025
39	-	0.090	0.032
45	-	0.138	0.042
51	-	0.127	0.039
57	-	0.258	0.046
63	-	0.133	0.060
69	-	0.177	0.052
75	-	0.172	0.052
81	-	0.402	0.153
87	-	0.201	0.133
93	-	0.609	0.308
99	-	0.246	0.087
105	-	1.758*	0.303
111	-	0.241	0.170
117	-	2.822*	0.181
123	-	0.284	0.356
129	-	4.051	0.125
135	-	0.333	0.201
141	-	6.991*	0.161
147	-	0.386	0.434
153	-	3.801	0.362

Table 1 is a comparison of execution times for the brute force, the stack and the recursion algorithms we tried. The table is expressed in terms of execution time which is known to be inexact because of processes that run in the background of the operating system. The asterisked

occurrences for the stack algorithm found no solution even though one exists. We believe one of our lists in the stack algorithm is missing the last occurrence. After implementation of the recursive algorithm, debugging of the stack algorithm was abandoned. The times in the table above were run with windows update, virus scans and other processes set to manual. The command prompt console executed at fifty percent of CPU. It is unlikely that the three to four percent of CPU taken by operating system processes had much of an impact on the times. The times quoted are the median time of three runs excepting the time for brute force 27 inputs which was run only once.

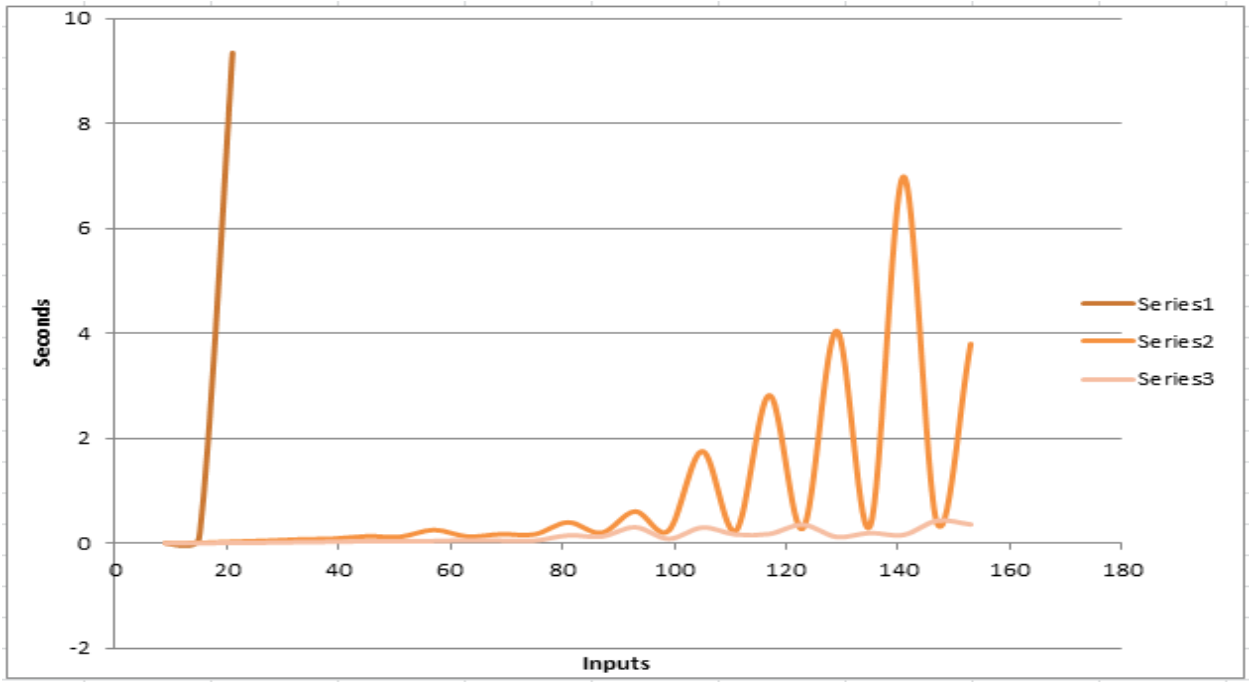

Figure 1: Graph of Performance of 3-Partition Algorithms

Figure 1 displays a graphical view of the various algorithms' performance. The stack algorithm times are an improvement but an exponential trend is clearly shown. Some occurrences show increased stack use while others do not. We speculate that this is related to the number of subsets tied for the same rank. The recursive algorithm times are shown to be superior to either of the other algorithms, for what we now call *small* inputs.

There is empirical evidence for an average case time of O (m^2) for the recursive accept and reject algorithm. The execution time for 999 inputs was 49.627 seconds. For 2523 inputs the execution time was 6 minutes and 10.38 seconds. 2523 inputs is approximately 2.5 times the size of 999 inputs; 2.5 squared equals 6.25; 6.25 times 50 seconds is 312.5 seconds. This is approximately the time expected empirically for our proposed average case time.

We call our new approach *the Clever Choice Algorithm*.

Future Research

Other counter examples of inputs that have solutions that are not solved by this algorithm might found. In fact, we hope that this is the case. Each counter example that is found allows us to test a new interpretation of element frequency and/or subset rank information to determine the best use of that information. For now, we have a polynomial time algorithm that solves 3-Partition for certain inputs. Various inputs up to 9999 have been computationally tested including random inputs (from program in Appendix D) and have provided solutions in polynomial time. Up to this writing, solutions to Partition problems of any kind with inputs no greater than 100 have been published. We have increased the ceiling.

The next improvement to the algorithm may be to add a stack and push the status of subsets when there is a tie based on information that we are using to select our subsets. We hope that this is not the case. It would be better to discover the best criteria for selecting subsets in a deterministic optimal order. Adding a stack is a way to accommodate non-determinism which is exponential in nature.

There are two known types of information that are useful in selecting subsets; element frequency and subset rank. They are closely related, like waves and particles in physics. If the frequency of an element in the valid subsets is 'one', we must select that subset because each

element must be used in a solution exactly once. If we select subsets in order of low rank, we are more likely to find a solution than by selecting subsets in order of high rank. This was the key observation (epiphany) that led to this algorithm.

If you wish to implement and test the implementation of the recursive algorithm, the source code is available in Appendix C. A user's manual can be found to help with executing the program in Appendix B. For those that wish to test random input sets, Appendix D provides a Java program to produce such inputs.

Future research will involve further evaluation of the orderly search algorithm *by element frequency* and/or *by low rank*. Subsets that are tied for the same criteria as the subset that would have been selected are instead chosen using the more subtle nuances of these criteria. The algorithm will be adjusted to accommodate multi-set and zeroes as inputs which may allow solutions for a broader range of inputs.

It will be interesting to see what the time complexity is for the future *complete* exhaustive version that will always find a solution.

VIII. References

1 Aaronson, S. (2003). Is P Versus NP Formally Independent? In Bulletin of the European Association for Theoretical Computer Science. Vol 81, 109.

2 Alidaee, B., Glover, F., Kochenberger, G. A. & Cesar, R. (2005). A New Modeling and Solution approach for the Number Partitioning Problem. In Journal of Applied Mathematics & Decision Sciences. Vol 2005, (2), 113-121.

3 Berger, F. & Klein, R. (2010). A Traveller's Problem. In Proceedings of the 2010 Annual Symposium for Computational Geometry. ACM, 176-182.

4 Cook, S., (2003). The Importance of the P Versus NP Question. In J. ACM 50 (1), 27-29.

5 Cook, S., McKenzie, P., Wehr, D., Braverman, M. & Santhanam, R. (2010). Pebbles and Branching Programs for Tree Evaluation. Retrieved from http://www.cs.toronto.edu/~sacook/

6 Dyer, M. & Frieze, A. (1989). The Solutions of Some Random NP-Hard Problems in Polynomial Expected Time. In Journal of Algorithms, Vol 10, 451-489.

7 Ermedahl, A., Stappert, F. & Engblom, J. (2003). Clustered Calculation of Worst-Case Execution Times. In Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems. ACM, 51-62.

8 Garey, M. R. & Johnson, D. S. (1976). Complexity Results for Multiprocessor Scheduling Under Resource Constraints. In SIAM Journal on Scientific Computing, Vol 4, (4).

9 Garey, M. R., Johnson, D. S & Ravi Sethi. (1976). The Complexity of Flowshop and Jobshop Scheduling. In Mathematics of Operations Research, Vol 1, (2).

10 Garey, M. R. & Johnson, D. S. (1978). Strong NP-Completeness Results: Motivation, Examples, and Implications. In J. ACM 25, (3), 499-508.

11 Garey, M. R. & Johnson D. S. (1979). Computers and intractability: A guide to the theory of np-completeness. New York, NY: W. H. Freeman.

12 Gent, I. P. & Walsh T. (1998). Analysis of Heuristics for Number Partitioning. In Computational Intelligence, Malden, MA: Vol 14, (3).

13 Hayes, B. (2002). The Easiest Hard Problem. In American Scientist, Research Triangle Park, NC: Vol 90, (3), 113-118.

14 Impagliazzo, R. (1995). A Personal View of Average-Case Complexity. Retrieved from http://cseweb.ucsd.edu/users/russell/ from link titled A Personal View of Average-Case Complexity.

15 Jackson, B., Scargle J. D., Barnes, D., Arabhi, S., Alt, A., Gioumousis, E. G., et al. (2005). An Algorithm for Optimal Partitioning of Data on an Interval. In Signal Processing Letters, IEEE. Vol 12, (2), 105-108.

16 Jain, R., Chaudhari, N. S. (2012). A New Bit Wise Technique for 3-Partitioning Algorithm. In Special Issue of International Journal of Computer Applications (0975–8887) on Optimization and On-chip Communication. No. 1, Feb.2012, www.ijcaonline.org.

17 Johnson, D. S. (2002). A Theoretician's Guide to the Experimental Analysis of Algorithms. In DIMACS Series in Discrete Mathematics and Theoretical Computer Science. Vol 59.

18 Joosten, S. (2010). Relaxations of the 3-Partition Problem. (Thesis) Retrieved from http://essay.utwente.nl/61447/1/MSc_S_Joosten.pdf

19 Karp, R. M. (1972). Reducibility Among Combinatorial Problems. In R. E. Miller and J. W. Thatcher (editors). Complexity of Computer Computations. New York: Plenum., 85–103.

20 Karp, R. M., Shenker, S. & Papadimitriou, C. H. (2003). A Simple Algorithm for Finding Frequent Elements in Streams and Bags. ACM Trans. Database Syst. 28, (1), 51-55.

21 Korf, R. E., (1998). A Complete Anytime Algorithm for Balanced Number Partitioning. Artificial Intelligence, Vol 106, (2), 181-203.

22 Lokshtanov, D. & Nederlof, J. (2010). Saving Space by Algebraization. In Proceedings of the 42nd ACM Symposium on Theory of Computing. ACM, 321-330.

23 Mendelzon, A.O. & Pu, K. Q. (2003). Concise Descriptions of Subsets of Structured Sets. In Proceedings of the Twenty-Second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems. ACM, 123-133.

24 Mertens, S. (1998). Phase Transition in the Number Partitioning Problem. In Physical Review Letters, Institut für Physik, Magdeburg, Germany: Vol 81, (20).

25 Mulmuley, K. (1989). A Fast Planar Partition Algorithm, II. In Proceedings of the Fifth Annual Symposium on Computational Geometry. ACM, 33-43.

26 Ramamoorthy, S. (1989). 3-Partition Remains Intractable for Distinct Numbers. (Thesis). University of California, Santa Cruz, inter-library loan.

27 Riengold, E. M., Nievergelt, J. & Deo, N. (1977). Combinatorial algorithms: Theory and practice. Englewood Cliffs, NJ: Prentice-Hall Inc.

Appendix A. Valid Subsets

Define the set of input elements
let set $A = \{a_1, a_2, ..., a_{3m}\}$
$\forall\ a$ of a "*size*" $s(a) \in A$: $s(a_1) < s(a_2) < \ldots < s(a_{3m})$
$\forall\ a \in A$: $s(a) \in Z^+$
$m, B \in Z^+$
$\sum_{a \in A} s(a) = mB$

Define the set of valid subsets of set *A*
let set $V = \{ A_1, A_2, ..., A_t\}$
let $r = \sum_{a \in An} |\text{count } s(a_i) \in V| + |\text{count } s(a_j) \in V| + |\text{count } s(a_k) \in V|$
$\forall\ A_t$ of a "*rank*" $r(A) \in A$: $r(A_1) \leq r(A_2) \leq \ldots \leq r(A_t)$ then by status then lexicographical order
$t \in Z^+, t > m$
$\forall\ A_i, A_j \in A$: $A_i \cap A_j = \emptyset$
$\forall\ A_i \in A = |3|$
$\forall\ A_i \in V$: $\sum_{a \in Ai} s(a) = B$

Define lowest frequency element and lowest ranked subset in lexicographical order
let set $L_{ijk} = \{a_i, a_j, a_k\}$: $L_{ijk} \in V, L_{ijk} \subseteq A$
$a_{low} = (\min |\text{count } s(a_n)|: a_n \in V|) \wedge (\min |\text{size } s(a_n)|: a_n \in A|)$
L_{ijk} : $(a_i = a_{low} \vee a_j = a_{low} \vee a_k = a_{low}) \wedge (\min |\text{count } s(a_i) + \text{count } s(a_j) + \text{count } s(a_k)|$
in lexicographical order)

Define a set of solution subsets from set *A*
iff $\exists S$ then let set $S_n = \{ A_1, A_2, ..., A_m\}$
n, the number of solutions
$m,n \in Z^+$
$\forall\ A_i, A_j \in A$: $A_i \cap A_j = \emptyset$
$\forall\ A_i \in A = |3|$
$\forall\ A_i \in V$: $\sum_{a \in Ai} s(a) = B$

Valid subsets are analogous to the n integer partitions of B. This is not the same as the strongly NP-Complete 3-Partition problem. The n integer partition of 7, Reingold, pg. 192 [27] is

{1,1,1,1,1,1,1}, {2,1,1,1,1}, {2,2,1,1,1}, {2,2,2,1}, {3,1,1,1,1}, {3,2,1,1,}, {3,2,2}, {3,3,1}, {4,1,1,1}, {4,2,1}, {4,3}, {5,1,1}, {5,2}, {6,1}, {7}.

The $n = 3$ integer partition of 7 is {3,2,2}, {3,3,1}, {4,2,1}, {5,1,1}. It is the integers that add to 7 that have exactly three members. For the formal definition of 3-Partition, only those that are disjoint (no duplicates) would be considered, {4,2,1}. An enumeration of 3-Partition

solutions might involve an adaptation of the n=3 integer partitions of B.

Instead of using integer partitions and reducing the selections to those we desire, we iterate from largest input in decreasing order in an inner loop and increment from smallest input in increasing order in an outer loop and stop when smallest, remainder and largest are within 1 or less of each other. See Appendix C, 3-Partition Source Code and find 'Derive valid subsets' near the bottom of page 36 of this thesis for source code. For valid subsets time, if $3m$ = the number of inputs, we would have steps taking a time of:

$t(m) = 3m–1 + 3m–2 + ... + 3m–{}^{3m}/_{2}$ (${}^{3m}/_{2}$ is an integer division, dropping the decimal)

m=3, 8+7+6+5 = 26 total operations
m=5, 14+13+12+11+10+9+8 = 77 total operations
m=7, 20+19+18+17+16+15+14+13+12+11 = 155 total operations
m=9, 26+25+24+23+22+21+20+19+18+17+16+15+14 = 260 total operations

From quadratic regression at http://science.kennesaw.edu/~plaval/applets/QRegression.html, with m and *total operations* entered as inputs, $t(m) = 3{}^{3}/_{8}m^2 – 1{}^{1}/_{2}m + {}^{1}/_{8}$. The valid subsets routine is accomplished in polynomial quadratic time.

Appendix B. User Manual for Recursion Algorithm

Open a command prompt:
Click Windows start button, click All Programs, click Accessories, open Command Prompt

To Compile and run a program:

javac ThreePartition.java	compiles
java ThreePartition	runs

The Path for Java:
If you get an error message that javac is not recognized, update your Path
go to Control Panel, click System, click Advanced System Settings, click Environment Variables
under System Variables select and edit Path (take great care, cancel if in doubt)

;C:\Program Files\Java\jdk1.7.0_25\bin	if you do not see something like this, install java
;C:\users\yourusername\desktop	add a path to your desktop (Win 7)

Seek help if you are not fairly comfortable with the instructions

Pasting inputs (you can also manually input data)
The menu is the small symbol in front of "Command Prompt" at the top
select Edit then Paste to input to a running program with input you have selected and copied

Redirecting an input file into the program

java ThreePartition < input.txt	accepts input.txt as input

More memory and buffers for the console
The menu is the small symbol in front of "Command Prompt" at the top of the window
select Command Prompt Properties then Buffer Size then Number of Buffers
adjust until you retain all program output, the solutions can overrun the default buffer size

To add additional java JVM memory
type systeminfo on the command line to find available physical memory
use up to half of the available physical memory, example 4000m = 4g so set parameters to 2g

java -Xmx:2g -Xms:2g ThreePartition	runs with more memory

Appendix C. 3-Partition Source Code

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.InputMismatchException;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.NoSuchElementException;
import java.util.Scanner;

public class ThreePartition {
	public static void main(String[] args) {
		// Initial setup
		boolean solutionFound = false;
		int solve = 0;
		int inputDone = 0;
		int oldSize = 0;
		int oldSolve = 0;
		int availableSubsets = 0;
		int elementSum = 0; // Sum of inputs
		int m = 0; // How many multiples of three elements
		int B = 0; // Sum divided by m, "Bound" of subset size
		long startTime = 0;
		Scanner input = new Scanner(System.in);
		ArrayList<Integer> rawInputs = new ArrayList<Integer>();
		// Input rules display
		System.out.println("Enter nonnegative elements in multiples of three (3m)");
		System.out.println("Elements must add to a multiple of the above multiple (m)");
		System.out.println("For example, if fifteen elements are entered that is");
		System.out.println("equal to five multiples of three, therefore the sum");
		System.out.println("must be a multiple of five...");
		System.out.println("Or, if eighteen elements are entered that is equal to");
		System.out.println("six multiples of three, therefore the sum must be a");
		System.out.println("multiple of six...");
		System.out.println("Enter the elements separated by a space");
		System.out.println("Enter -1 to end");
		System.out.println("For example 1 2 3 4 5 6 7 8 9 -1\n");

		// Accept 3m inputs
		while (inputDone != -1) {
			// user input to array
			try {
				rawInputs.add(new Integer(input.nextInt()));
			} catch (InputMismatchException e) {
				System.out.format("Error: Invalid character was input! \n");
				System.out.print("Exiting... \n");
				System.exit(0);
			} catch (NoSuchElementException e) {
				// user entered CTRL C
				System.exit(0);
			}
			// Sum of elements except -1 here (only execute when -1 entered)
			elementSum = 0;
```

```
		if (rawInputs.contains(new Integer(-1))) {
			if ((rawInputs.size() != 1) && ((rawInputs.size() % 3) == 1)) {
				for (int i = 0; i < rawInputs.size() - 1; i++) {
					elementSum += rawInputs.get(i); // Sum inputs
				}
				// Subtract one for the -1 added when quitting
				m = (rawInputs.size() - 1) / 3;
				B = elementSum / m;
			}
			if (rawInputs.size() == 1) {
				System.out.print("Exiting... \n");
				System.exit(0);
			} else if (((rawInputs.size() % 3) == 1)
					&& ((elementSum % m) == 0)) {
				inputDone = -1;
				rawInputs.remove(new Integer(-1));
				// Put 0 in row 0 which is not used
				rawInputs.add(0, new Integer(0));
			} else if ((rawInputs.size() % 3) != 1) {
				rawInputs.remove(new Integer(-1));
				System.out.println("ERROR: Enter inputs in multiples of 3");
				System.out.println("Continue data entry or CTRL C\n");
				System.out.format("\n");
			} else if ((elementSum % m) != 0) {
				rawInputs.remove(new Integer(-1));
				System.out.println("ERROR: Sum (" + elementSum
						+ ") not a multiple of " + m);
				System.out.println("Continue data entry or CTRL C\n");
				System.out.format("\n");
			}
		}
	}
	input.close();

	// time how long it takes to find a solution
	startTime = System.nanoTime();
	// Now we know the value of m
	Integer[][] solution = new Integer[m + 1][5];
	// Sort raw inputs
	Collections.sort(rawInputs);

	Map<Integer, Integer> inputsHash = new LinkedHashMap<Integer, Integer>();
	Integer[] setA = new Integer[rawInputs.size()];
	setA = rawInputs.toArray(setA);
	ArrayList<Integer> matchSet = new ArrayList<Integer>();

	// Initialize the LinkedHashMap for inputs and frequencies
	for (int i = 1; i < setA.length; i++) {
		int temp = 0;
		inputsHash.put(setA[i], temp);
	}

	// Method 1 The Valid Subsets Reduction - begin
	matchSet.add(0, new Integer(0)); // The zero occurrence, not used
	int diff = 0;
```

```
int hsub = setA.length - 1;
int msub = hsub - 1;
int lsub = 1;
int old_setA_lrgSub = setA[hsub];
int old_setA_smlSub = setA[lsub];
while (hsub > msub) {
	lsub = 1;
	msub = hsub - 1;
	diff = B - (setA[hsub] + setA[lsub]);
	for (int i = hsub - 1; i > lsub - 1; i--) {
		if (setA[i] == diff) {
			msub = i;
			break;
		}
		if (setA[i] < diff) {
			break;
		}
	}
	while (lsub < msub) {
		diff = B - (setA[hsub] + setA[lsub]);
		for (int j = msub; j > lsub; j--) {
			if (setA[j] == diff) {
				int temp;
				msub = j;
				try {
					matchSet.add(new Integer(setA[lsub]));
					temp = inputsHash.get(setA[lsub]);
					temp++;
					inputsHash.put(setA[lsub], temp);
					matchSet.add(new Integer(setA[msub]));
					temp = inputsHash.get(setA[msub]);
					temp++;
					inputsHash.put(setA[msub], temp);
					matchSet.add(new Integer(setA[hsub]));
					temp = inputsHash.get(setA[hsub]);
					temp++;
					inputsHash.put(setA[hsub], temp);
				} catch (OutOfMemoryError e) {
					System.out.format("Too many subsets!\n");
					System.out.print("Exiting... \n");
					System.exit(0);
				}
				break;
			}
		}
		old_setA_smlSub = setA[lsub];
		while ((lsub < hsub) && (old_setA_smlSub == setA[lsub])) {
			lsub++;
		}
	}
	old_setA_lrgSub = setA[hsub];
	while (old_setA_lrgSub == setA[hsub]) {
		hsub--;
	}
} // Method 1 The Valid Subsets Reduction - end
```

```
// Build subset array, update subset rank
if (matchSet.size() == 1) {
        System.out.format("0 solutions were found. \n");
        System.exit(0);
}
int sub = 0;
int subscript = 0;
Integer[][] subset = new Integer[((matchSet.size() / 3) + 1)][5];
subset[0][4] = new Integer(0);
subset[0][3] = new Integer(0);
subset[0][2] = new Integer(0);
subset[0][1] = new Integer(0);
subset[0][0] = new Integer(0);
while (sub < (matchSet.size() - 1)) {
        subscript++;
        subset[subscript][4] = new Integer(0);
        subset[subscript][3] = matchSet.get(sub + 3);
        subset[subscript][2] = matchSet.get(sub + 2);
        subset[subscript][1] = matchSet.get(sub + 1);
        subset[subscript][0] = inputsHash.get(matchSet.get(sub + 3))
                        + inputsHash.get(matchSet.get(sub + 2))
                        + inputsHash.get(matchSet.get(sub + 1));
        sub += 3;
}
matchSet.clear();
// Method 3 Orderly Search - begin
availableSubsets = 1999999999;
// ************************
// ** Outer program loop **
// ************************
while ((availableSubsets > m-1) && (solutionFound == false)) {

solve = 0;
// Method 2 Lowest Ranked Subset - begin
// ************************
// ** Main Program Loop **
// ************************
while ((rawInputs.size() > 1) && (rawInputs.size() != oldSize)) {

        solve++;

        // Java comparison sort columns 0 1 2 of subsets, O(n log n)
        Arrays.sort(subset, new Comparator<Integer[]>() {
                public int compare(Integer[] o1, Integer[] o2) {
                        Integer[] row1 = o1;
                        Integer[] row2 = o2;
                        if (row2[0] == null || row1[0] == null) {
                                return 0;
                        } else {
                                if ((row1[4].equals(row2[4])) &&
                                        (row1[0].equals(row2[0])) &&
                                        (row1[1].equals(row2[1])) ) {
                                        return row1[2].compareTo(row2[2]);
                                } else if ( (row1[4].equals(row2[4])) &&
```

```
                                                    (row1[0].equals(row2[0])) ) {
                                    return row1[1].compareTo(row2[1]);
                            } else if ( (row1[4].equals(row2[4])) ) {
                                    return row1[0].compareTo(row2[0]);
                            } else {
                                    return row1[4].compareTo(row2[4]);
                            }
                    }
            }
});

if (subset[1][4].equals(0)) {
            oldSolve = solve;
}

// Initialize frequencies
int initialize = 0;
@SuppressWarnings("rawtypes")
Iterator iter = inputsHash.entrySet().iterator();
while (iter.hasNext()) {
            @SuppressWarnings("rawtypes")
            Map.Entry pairs = (Map.Entry) iter.next();
            int inputValue = (Integer) pairs.getKey();
            int frequency = (Integer) pairs.getValue();
            frequency = initialize;
            inputsHash.put(inputValue, frequency);
}

for (int u = 1; u < subset.length; u++) {
            // Set status of subsets with elements from selected subset
            if  ((subset[u][4].equals(0)) &&
                        ((subset[u][1].equals(subset[1][1])) ||
                         (subset[u][1].equals(subset[1][2])) ||
                         (subset[u][1].equals(subset[1][3])) ||
                         (subset[u][2].equals(subset[1][1])) ||
                         (subset[u][2].equals(subset[1][2])) ||
                         (subset[u][2].equals(subset[1][3])) ||
                         (subset[u][3].equals(subset[1][1])) ||
                         (subset[u][3].equals(subset[1][2])) ||
                         (subset[u][3].equals(subset[1][3])))) {
                        // Different statuses set for future bitset stack
                        if ( (u == 1) && (solve == 1) ) {
                                    subset[u][4] = new Integer(1); // Primary
                        } else if ( (u == 1) &&
                                                (subset[u][0].equals(subset[u+1][0])) &&
                                                (!(subset[u][0].equals(3))) ) {
                                    subset[u][4] = new Integer(3); // Nondeterm
                        } else if (u == 1) {
                                    subset[u][4] = new Integer(2); // Selected
                        } else {
                                    subset[u][4] = new Integer(4); // Eliminated
                        }
            }

            // Reset the frequencies of unused subsets
```

```
                int temp = 0;
                if ((subset[u][4].equals(0))) { // Status 0 = unused
                        // Increment each element frequency
                        temp = inputsHash.get(subset[u][1]);
                        temp++;
                        inputsHash.put(subset[u][1], temp);
                        temp = inputsHash.get(subset[u][2]);
                        temp++;
                        inputsHash.put(subset[u][2], temp);
                        temp = inputsHash.get(subset[u][3]);
                        temp++;
                        inputsHash.put(subset[u][3], temp);
                }
        }

        for (int v = 1; v < subset.length; v++) {
                // Update subset ranks
                if (subset[v][4] == 0) {
                        subset[v][0] = new Integer(
                                inputsHash.get(subset[v][1]) +
                                inputsHash.get(subset[v][2]) +
                                inputsHash.get(subset[v][3]));
                }
        }

        // Add valid subset to solution
        if (oldSolve == solve) {
                solution[solve][1] = subset[1][1];
                solution[solve][2] = subset[1][2];
                solution[solve][3] = subset[1][3];
        }

        oldSize = rawInputs.size();
        // Delete low ranked subset from rawInputs/inputsHash, loop control
        if (oldSolve == solve) {
                rawInputs.remove(new Integer(solution[solve][1]));
                rawInputs.remove(new Integer(solution[solve][2]));
                rawInputs.remove(new Integer(solution[solve][3]));
                inputsHash.remove(solution[solve][1]);
                inputsHash.remove(solution[solve][2]);
                inputsHash.remove(solution[solve][3]);
        }

}  // End of main loop here
// Method 2 Lowest Ranked Subset - end
// Print solution or not found message
if (rawInputs.size() == oldSize) {
        // Do nothing
} else {
        System.out.format("solution = \n");
        solutionFound = true;
        for (int f = 1; f < m + 1; f++) {
                System.out.format("  %d %d %d ", solution[f][1],
                                solution[f][2], solution[f][3]);
                if (f % 7 == 0) {
```

```
                                        System.out.format("\n");
                                }
                        }
                        System.out.format("\n");
                }

                // Rebuild initial inputsHash and rawInputs
                for (int i = 1; i < setA.length; i++) {
                        int temp = 0;
                        inputsHash.put(setA[i], temp);
                        if (!rawInputs.contains(new Integer(setA[i]))) {
                                rawInputs.add(new Integer(setA[i]));
                        }
                }

                // Rebuild subset status to available
                availableSubsets = 0;
                for (int u = 1; u < subset.length; u++) {
                        if  (!(subset[u][4].equals(1))) { // Not the primary subsets
                                subset[u][4] = new Integer(0); // Available
                                availableSubsets++;
                        }

                        int temp = 0;
                        if ((subset[u][4].equals(0))) { // Available
                                temp = inputsHash.get(subset[u][1]);
                                temp++;
                                inputsHash.put(subset[u][1], temp);
                                temp = inputsHash.get(subset[u][2]);
                                temp++;
                                inputsHash.put(subset[u][2], temp);
                                temp = inputsHash.get(subset[u][3]);
                                temp++;
                                inputsHash.put(subset[u][3], temp);
                        }
                }

                // Rebuild subset rank
                for (int v = 1; v < subset.length; v++) {
                        subset[v][0] = new Integer(
                                inputsHash.get(subset[v][1]) +
                                inputsHash.get(subset[v][2]) +
                                inputsHash.get(subset[v][3]));
                }

                }  // End of outer loop here
                // Method 3 Orderly Search - end
                double endTime = ((double) System.nanoTime() - (double) startTime) / 1000000000;
                System.out.format("results found in %.3f seconds.\n", endTime);

        }

}
```

Appendix D. Random Inputs Source Code

```
import java.util.ArrayList;
import java.util.Random;
public class RandomInputs {

	public static void main(String[] args) {
	ArrayList<Integer> inputList = new ArrayList<Integer>();
	Random rand1 = new Random();
	rand1.setSeed(System.currentTimeMillis());
	// Generate a random number of buckets from 3to7
	int randBuckets = rand1.nextInt(4) + 3;  // Update to change range of m (buckets)
	Random rand2 = new Random();
	rand2.setSeed(System.currentTimeMillis());
	int inputSum = 0;
	int compareSum = 0;
	int m = 0;
	int B = 0;
	boolean valid3P = false;
	while (true) {
	   // Generate a number from 1to 42
	   int randNum = rand2.nextInt(41) + 1;  // Update to change range of inputs
	   // Allow only disjoint inputs
	   if (inputList.contains(randNum)) {
	      continue;
	   } else {
	     inputSum += randNum;
	      inputList.add(randNum);
	      m = inputList.size() / 3;
	      if (m > 0){
	         B = inputSum / m;
	      }
	      compareSum = m * B;
	      // Check for 3m inputs, divisible by m
	      valid3P = false;
	      if ( ((inputList.size() % 3) == 0) && (inputSum == compareSum)) {
	         valid3P = true;
	      }
	   }
	   if ((valid3P == true) && (m >= randBuckets) && ((m % 2) == 1)) {
	     System.out.format("m = %d\n", m);
	      for (int j=0;j<inputList.size();j++) {
	         System.out.format("%d ", inputList.get(j));
	      }
	               System.out.format("-1\n");
	      break;
	   }
	   if ((inputList.size() > (4 * randBuckets)) == true) {
	      System.out.println("3P input list not found, please try again.");
	      break;
	   }
	}
   }
}
```

Appendix E. BitSet Stack Implementation

```
import java.util.BitSet;

	public class BitSetStack extends BitSet {

		public BitSetStack() {
		}

		public void pushbit(int index) {
			this.set(index);
		}

		public void pushbit(int index, boolean v) {
			this.set(index, v);
		}

		public boolean popbit(int index) {
			boolean b = this.peekbit(index);
			this.clear(index);
			return b;
		}

		public boolean peekbit(int index) {
			return this.get(index);
		}

		public boolean emptybits() {
			return this.isEmpty();
		}

		public int lastindex() {
			int i = this.length() - 1;
			if (i > -1) {
				return i;
			} else {
				return 0;
			}
		}

		private static final long serialVersionUID = 192L;

	}
```

Appendix F. BitSet Snippets of Code

```
// Code snippets for implementing a bitset stack
        BitSetStack bsStack = new BitSetStack();

        bsStack.clear();

        if (bsStack.emptybits()) {//put stuff in here}

        // System.out.print("stack size = " + bsStack.lastindex() + "\n");

        // subset is the array for valid subsets with status & rank, subS is size
        subset = PopSubsetStates(subS + 1, subset, bsStack);

        // nondeterm status is converted to eliminated so restored stack causes new processing
        try {
                stackUses++;
                bsStack = PushSubsetStates(subS + 1, subset, bsStack, nondetermSubscript);
        } catch (IndexOutOfBoundsException e) {
                System.out.format("Error: Exceeded maximum bitset size! \n");
                System.out.format("Too many items were pushed to the stack! \n");
                System.out.print("Exiting... \n");
                System.exit(0);
        }

        public static Integer[][] PopSubsetStates(int numOfSubsets,
                        Integer[][] subsetArray, BitSetStack bitStack) {

                int lastBit = bitStack.lastindex();
                for (int f = numOfSubsets - 1; f > 0; f--) {
                        boolean bit4 = bitStack.popbit(lastBit);
                        boolean bit3 = bitStack.popbit(lastBit - 1);
                        boolean bit2 = bitStack.popbit(lastBit - 2);
                        boolean bit1 = bitStack.popbit(lastBit - 3);
                        if (bit1 == false && bit2 == false && bit3 == false && bit4 == true) {
                                subsetArray[f][0] = new Integer(0);
                        } else if (bit1 == false && bit2 == false && bit3 == true
                                        && bit4 == true) {
                                subsetArray[f][0] = new Integer(1);
                        } else if (bit1 == false && bit2 == true && bit3 == false
                                        && bit4 == true) {
                                subsetArray[f][0] = new Integer(2);
                        } else if (bit1 == false && bit2 == true && bit3 == true
                                        && bit4 == true) {
                                subsetArray[f][0] = new Integer(3);
                        } else if (bit1 == true && bit2 == false && bit3 == true
                                        && bit4 == true) {
                                subsetArray[f][0] = new Integer(4);
                        } else if (bit1 == true && bit2 == true && bit3 == false
                                        && bit4 == true) {
                                subsetArray[f][0] = new Integer(5);
                        } else {
                                System.out.println("Error: incorrect bit translation");
                        }
```

```
                lastBit -= 4;
        }
        return subsetArray;
}

public static BitSetStack PushSubsetStates(int numOfSubsets,
                Integer[][] subsetArray, BitSetStack bitStack, int nondeterm) {

        int lastBit = bitStack.lastindex();
        for (int f = 1; f < numOfSubsets; f++) {
                if (f == nondeterm) { // eliminated 1011, was previously nondeterm
                        bitStack.pushbit(lastBit + 1, true);
                        bitStack.pushbit(lastBit + 2, false);
                        bitStack.pushbit(lastBit + 3, true);
                        bitStack.pushbit(lastBit + 4, true);
                } else if (subsetArray[f][0] == 0) { // available 0001
                        bitStack.pushbit(lastBit + 1, false);
                        bitStack.pushbit(lastBit + 2, false);
                        bitStack.pushbit(lastBit + 3, false);
                        bitStack.pushbit(lastBit + 4, true);
                } else if (subsetArray[f][0] == 1) { // primary 0011
                        bitStack.pushbit(lastBit + 1, false);
                        bitStack.pushbit(lastBit + 2, false);
                        bitStack.pushbit(lastBit + 3, true);
                        bitStack.pushbit(lastBit + 4, true);
                } else if (subsetArray[f][0] == 2) { // of interest 0101
                        bitStack.pushbit(lastBit + 1, false);
                        bitStack.pushbit(lastBit + 2, true);
                        bitStack.pushbit(lastBit + 3, false);
                        bitStack.pushbit(lastBit + 4, true);
                } else if (subsetArray[f][0] == 3) { // nondeterm 0111
                        bitStack.pushbit(lastBit + 1, false);
                        bitStack.pushbit(lastBit + 2, true);
                        bitStack.pushbit(lastBit + 3, true);
                        bitStack.pushbit(lastBit + 4, true);
                } else if (subsetArray[f][0] == 4) { // eliminated 1011
                        bitStack.pushbit(lastBit + 1, true);
                        bitStack.pushbit(lastBit + 2, false);
                        bitStack.pushbit(lastBit + 3, true);
                        bitStack.pushbit(lastBit + 4, true);
                } else if (subsetArray[f][0] == 5) { // finished 1101
                        bitStack.pushbit(lastBit + 1, true);
                        bitStack.pushbit(lastBit + 2, true);
                        bitStack.pushbit(lastBit + 3, false);
                        bitStack.pushbit(lastBit + 4, true);
                } else {
                        System.out.println("Error: incorrect subset state");
                }
                lastBit += 4;
        }
        return bitStack;
}
```

Appendix G. A Large Solution for Inputs 1 through 2523

1 1264 2521 2 1266 2518 3 1267 2516 5 1890 1891 12 1251 2523 16 1248 2522 24 1242 2520

4 1270 2512 9 1888 1889 6 1286 2494 7 1282 2497 8 1288 2490 11 1292 2483 10 1304 2472

13 1294 2479 14 1308 2464 46 1221 2519 60 1209 2517 15 1316 2455 17 1312 2457 18 1338 2430

20 1324 2442 19 1318 2449 22 1352 2412 21 1330 2435 23 1340 2423 27 1322 2437 29 1337 2420

30 1325 2431 28 1349 2409 31 1319 2436 26 1362 2398 34 1341 2411 32 1370 2384 35 1350 2401

25 1378 2383 170 1101 2515 192 1080 2514 172 1103 2511 196 1077 2513 638 640 2508 636 641 2509

224 1052 2510 210 1069 2507 232 1048 2506 208 1073 2505 288 994 2504 310 974 2502 235 1051 2500

261 1024 2501 239 1044 2503 338 950 2498 317 970 2499 360 930 2496 347 944 2495 388 906 2492

395 898 2493 402 893 2491 434 864 2488 437 863 2486 390 909 2487 442 860 2484 448 858 2480

461 836 2489 417 884 2485 475 830 2481 532 772 2482 523 786 2477 522 788 2476 498 810 2478

502 813 2471 621 692 2473 618 698 2470 596 721 2469 620 700 2466 612 707 2467 474 837 2475

131 1187 2468 552 771 2463 550 774 2462 554 758 2474 569 752 2465 309 1016 2461 587 740 2459

496 832 2458 125 1201 2460 158 1172 2456 186 1147 2453 313 1022 2451 384 952 2450 316 1029 2441

318 1023 2445 324 1010 2452 159 1184 2443 368 964 2454 619 728 2439 598 741 2447 600 742 2444

370 978 2438 669 684 2433 376 976 2434 508 838 2440 530 808 2448 119 1238 2429 678 680 2428

120 1245 2421 357 1002 2427 134 1220 2432 122 1239 2425 639 732 2415 670 699 2417 582 778 2426

584 780 2422 180 1193 2413 169 1171 2446 298 1074 2414 191 1190 2405 346 1021 2419 590 793 2403

418 962 2406 588 799 2399 406 973 2407 420 956 2410 398 980 2408 624 770 2392 412 972 2402

663 730 2393 690 706 2390 649 750 2387 454 908 2424 484 911 2391 499 890 2397 233 1135 2418

480 902 2404 214 1177 2395 257 1113 2416 462 938 2386 460 955 2371 446 963 2377 468 937 2381

369 1028 2389 599 802 2385 227 1159 2400 205 1202 2379 470 936 2380 514 894 2378 562 842 2382

691 722 2373 528 862 2396 455 966 2365 505 912 2369 504 914 2368 190 1233 2363 456 971 2359

279 1131 2376 36 1431 2319 478 946 2362 270 1141 2375 483 942 2361 538 878 2370 156 1263 2367

672 759 2355 273 1119 2394 184 1230 2372 466 979 2341 391 1042 2353 476 954 2356 264 1134 2388

597 840 2349 485 958 2343 564 865 2357 542 870 2374 254 1181 2351 544 892 2350 307 1132 2347

536 896 2354 294 1161 2331 341 1081 2364 293 1133 2360 486 967 2333 396 1050 2340 287 1164 2335

602 859 2325 325 1095 2366 492 981 2313 592 857 2337 682 756 2348 33 1648 2105 375 1072 2339

174 1254 2358 339 1102 2345 570 882 2334 38 1457 2291 315 1150 2321 300 1169 2317 566 888 2332

40 1473 2273 520 920 2346 353 1106 2327 37 1830 1919 301 1162 2323 258 1199 2329 168 1276 2342

321 1121 2344 578 886 2322 595 880 2311 408 1026 2352 617 866 2303 397 1082 2307 608 881 2297

606 879 2301 646 812 2328 738 749 2299 41 1722 2023 540 928 2318 614 852 2320 39 1780 1967
710 781 2295 704 744 2338 44 1479 2263 42 1511 2233 50 1485 2251 45 1460 2281 47 1750 1989
43 1714 2029 667 804 2315 54 1487 2245 291 1183 2312 560 910 2316 48 1477 2261 652 829 2305
58 1517 2211 49 1468 2269 53 1796 1937 51 1676 2059 55 1700 2031 52 1653 2081 56 1781 1949
59 1768 1959 61 1752 1973 152 1310 2324 57 1732 1997 66 1811 1909 62 1591 2133 63 1670 2053
64 1671 2051 72 1817 1897 430 1046 2310 65 1836 1885 69 1774 1943 70 1449 2267 282 1211 2293
162 1298 2326 697 800 2289 263 1214 2309 67 1650 2069 68 1529 2189 71 1808 1907 73 1672 2041
78 1547 2161 76 1539 2171 79 1698 2009 82 1753 1951 80 1523 2183 74 1751 1961 77 1597 2112
75 1710 2001 100 1356 2330 88 1783 1915 84 1839 1863 81 1784 1921 140 1346 2300 83 1619 2084
289 1212 2285 89 1535 2162 94 1589 2103 91 1756 1939 90 1621 2075 85 1590 2111 86 1545 2155
99 1408 2279 452 998 2336 87 1509 2190 92 1647 2047 101 1402 2283 102 1380 2304 95 1560 2131
97 1762 1927 98 1819 1869 96 1531 2159 548 940 2298 103 1790 1893 93 1833 1860 211 1300 2275
104 1481 2201 110 1483 2193 500 992 2294 568 948 2270 107 1430 2249 106 1793 1887 141 1374 2271
108 1569 2109 111 1704 1971 105 1624 2057 112 1587 2087 113 1432 2241 319 1180 2287 534 960 2292
130 1701 1955 132 1399 2255 128 1368 2290 116 1620 2050 123 1802 1861 118 1673 1995 160 1344 2282
129 1680 1977 126 1480 2180 202 1320 2264 181 1328 2277 109 1652 2025 124 1805 1857 117 1563 2106
150 1348 2288 147 1400 2239 153 1390 2243 114 1451 2221 127 1728 1931 138 1649 1999 331 1208 2247
510 1000 2276 135 1428 2223 220 1260 2306 236 1236 2314 719 814 2253 121 1682 1983 144 1731 1911
572 918 2296 400 1129 2257 230 1291 2265 556 982 2248 115 1692 1979 154 1759 1873 242 1272 2272
371 1156 2259 558 968 2260 137 1642 2007 146 1561 2079 187 1372 2227 149 1646 1991 622 922 2242
136 1809 1841 133 1618 2035 526 1020 2240 148 1725 1913 178 1661 1947 151 1694 1941 142 1429 2215
267 1290 2229 428 1078 2280 171 1396 2219 139 1513 2134 458 1076 2252 701 854 2231 450 1070 2266
449 1100 2237 642 900 2244 671 916 2199 143 1452 2191 161 1758 1867 182 1679 1925 753 820 2213
164 1787 1835 157 1730 1899 594 934 2258 212 1729 1845 155 1450 2181 163 1510 2113 166 1537 2083
165 1404 2217 176 1645 1965 183 1398 2205 177 1640 1969 260 1296 2230 725 826 2235 188 1595 2003
218 1651 1917 145 1454 2187 488 996 2302 209 1702 1875 204 1505 2077 512 1054 2220 424 1153 2209
343 1240 2203 426 1128 2232 762 806 2218 206 1559 2021 189 1592 2005 175 1458 2153 198 1643 1945
322 1210 2254 179 1726 1881 238 1334 2214 199 1455 2132 217 1459 2110 506 1018 2262 401 1160 2225
240 1565 1981 482 1030 2274 200 1593 1993 213 1410 2163 167 1482 2137 323 1268 2195 185 1541 2060
193 1567 2026 203 1724 1859 234 1533 2019 586 988 2212 215 1489 2082 723 856 2207 216 1515 2055
720 828 2238 241 1622 1923 610 990 2186 173 1512 2101 290 1218 2278 207 1562 2017 295 1326 2165

194 1617 1975 201 1507 2078 350 1244 2192 372 1130 2284 777 834 2175 229 1508 2049 374 1186 2226
668 872 2246 269 1376 2141 231 1426 2129 262 1571 1953 268 1491 2027 344 1232 2210 432 1104 2250
245 1456 2085 340 1262 2184 427 1182 2177 197 1782 1807 292 1461 2033 237 1406 2143 221 1755 1810
222 1699 1865 259 1342 2185 754 868 2164 228 1727 1831 195 1754 1837 373 1216 2197 266 1625 1895
724 776 2286 366 1152 2268 265 1354 2167 286 1721 1779 392 1158 2236 255 1424 2107 696 782 2308
429 1188 2169 314 1427 2045 243 1623 1920 285 1669 1832 271 1703 1812 342 1615 1829 726 844 2216
748 850 2188 296 1677 1813 244 1757 1785 643 1004 2139 451 1178 2157 312 1401 2073 320 1377 2089
394 1234 2158 283 1697 1806 422 1108 2256 225 1723 1838 345 1314 2127 616 997 2173 405 1246 2135
297 1674 1815 348 1375 2063 219 1675 1892 311 1613 1862 284 1403 2099 694 932 2160 453 1154 2179
479 1192 2115 256 1433 2097 644 993 2149 253 1453 2080 404 1274 2108 563 1098 2125 223 1777 1786
378 1347 2061 281 1641 1864 364 1644 1778 248 1484 2054 349 1566 1871 509 1126 2151 666 924 2196
702 939 2145 648 1110 2028 226 1536 2024 399 1486 1901 272 1540 1974 674 995 2117 246 1568 1972
367 1616 1803 252 1564 1970 326 1594 1866 365 1538 1883 336 1534 1916 247 1705 1834 531 1136 2119
746 969 2071 352 1434 2000 280 1588 1918 403 1543 1840 423 1157 2206 337 1373 2076 377 1269 2140
351 1241 2194 393 1185 2208 673 891 2222 537 1079 2170 477 1075 2234 535 1049 2202 457 1105 2224
299 1321 2166 251 1405 2130 557 1047 2182 645 943 2198 751 807 2228 363 1371 2052 431 1217 2138
421 1237 2128 511 1107 2168 503 1127 2156 695 913 2178 274 1514 1998 529 1155 2102 425 1189 2172
561 1025 2200 613 1019 2154 591 1053 2142 779 833 2174 327 1323 2136 419 1345 2022 803 831 2152
611 1071 2104 805 861 2120 379 1351 2056 727 915 2144 775 885 2126 433 1295 2058 447 1343 1996
783 889 2114 481 1215 2090 747 965 2074 755 999 2032 693 1045 2048 459 1297 2030 501 1265 2020
539 1243 2004 334 1506 1946 647 1163 1976 675 1109 2002 473 1213 2100 729 941 2116 585 1207 1994
354 1488 1944 615 1099 2072 801 887 2098 835 917 2034 811 1027 1948 583 1235 1968 513 1379 1894
527 1317 1942 665 1425 1696 407 1293 2086 593 1271 1922 855 991 1940 565 1353 1868 559 1423 1804
533 1261 1992 507 1191 2088 276 1596 1914 883 1017 1886 589 1151 2046 809 1299 1678 637 945 2204
278 1542 1966 250 1760 1776 277 1695 1814 249 1749 1788 275 1668 1843 304 1570 1912 555 1055 2176
305 1733 1748 330 1614 1842 717 919 2150 664 1001 2121 335 1407 2044 358 1478 1950 329 1599 1858
435 1289 2062 703 935 2148 409 1397 1980 567 1125 2094 302 1532 1952 380 1315 2091 487 1206 2093
306 1516 1964 303 1667 1816 308 1435 2043 333 1706 1747 361 1462 1963 773 867 2146 718 921 2147
757 907 2122 332 1558 1896 784 989 2013 328 1586 1872 676 1043 2067 439 1369 1978 541 1179 2066
609 1137 2040 389 1273 2124 827 841 2118 385 1463 1938 387 1381 2018 359 1666 1761 413 1598 1775
743 947 2096 580 1083 2123 472 1219 2095 745 1056 1985 386 1544 1856 444 1327 2015 362 1382 2042

623 1124 2039 440 1612 1734 383 1519 1884 356 1422 2008 414 1490 1882 553 1165 2068 355 1395 2036

651 1097 2038 382 1504 1900 415 1436 1935 493 1287 2006 839 961 1986 516 1205 2065 381 1585 1820

494 1503 1789 441 1475 1870 411 1639 1736 605 1111 2070 785 987 2014 467 1409 1910 416 1626 1744

661 1138 1987 853 975 1958 679 1015 2092 731 1122 1933 436 1313 2037 443 1355 1988 410 1656 1720

438 1530 1818 798 977 2011 445 1518 1823 469 1610 1707 495 1367 1924 463 1259 2064 635 1139 2012

713 1057 2016 677 1204 1905 465 1628 1693 525 1301 1960 471 1464 1851 521 1557 1708 464 1476 1846

650 1258 1878 733 1096 1957 549 1247 1990 823 1084 1879 524 1572 1690 653 1123 2010 515 1421 1850

489 1394 1903 497 1383 1906 576 1437 1773 825 1031 1930 547 1438 1801 575 1584 1627 658 1166 1962

490 1448 1848 574 1601 1611 581 1249 1956 603 1285 1898 604 1447 1735 869 933 1984 851 1003 1932

519 1439 1828 765 1085 1936 627 1231 1928 551 1581 1654 903 949 1934 577 1600 1609 630 1329 1827

543 1502 1741 518 1630 1638 705 1257 1824 824 1058 1904 491 1549 1746 579 1465 1742 685 1420 1681

662 1275 1849 689 1168 1929 763 1041 1982 629 1302 1855 546 1575 1665 895 1014 1877 631 1392 1763

632 1492 1662 657 1203 1926 634 1573 1579 716 1194 1876 683 1149 1954 607 1357 1822 767 1175 1844

633 1384 1769 659 1411 1716 797 1087 1902 545 1583 1658 687 1546 1553 573 1366 1847 711 1167 1908

794 1112 1880 601 1393 1792 517 1556 1713 628 1358 1800 626 1339 1821 792 1140 1854 655 1412 1719

787 1229 1770 571 1551 1664 625 1364 1797 874 1059 1853 656 1493 1637 760 1228 1798 923 1068 1795

712 1303 1771 848 1086 1852 715 1331 1740 660 1471 1655 739 1520 1527 688 1332 1766 736 1176 1874

686 1391 1709 737 1311 1738 959 1033 1794 709 1386 1691 815 1145 1826 1005 1093 1688 681 1474 1631

846 1222 1718 761 1418 1607 819 1195 1772 735 1284 1767 929 1032 1825 764 1501 1521 734 1497 1555

818 1223 1745 654 1446 1686 789 1469 1528 926 1061 1799 816 1365 1605 708 1445 1633 817 1306 1663

790 1360 1636 791 1278 1717 843 1444 1499 1089 1115 1582 1006 1091 1689 1007 1067 1712 871 1280 1635

876 1250 1660 925 1336 1525 951 1416 1419 1008 1035 1743 897 1255 1634 766 1466 1554 845 1441 1500

1011 1060 1715 904 1143 1739 1013 1114 1659 953 1225 1608 1034 1065 1687 873 1256 1657 769 1440 1577

905 1117 1764 986 1197 1603 1039 1063 1684 822 1173 1791 714 1443 1629 796 1388 1602 795 1417 1574

877 1414 1495 984 1037 1765 1009 1094 1683 847 1467 1472 1040 1252 1494 821 1385 1580 768 1470 1548

849 1226 1711 931 1413 1442 985 1064 1737 983 1277 1526 927 1174 1685 957 1253 1576 1012 1224 1550

1116 1148 1522 1066 1088 1632 1196 1227 1363 1062 1200 1524 875 1305 1606 899 1283 1604 1036 1198 1552

901 1389 1496 1142 1146 1498 1038 1170 1578 1090 1281 1415 1120 1279 1387 1092 1333 1361 1118 1309 1359

1144 1307 1335

Solution was found in less than 6 minutes and 10.38 seconds on an Intel i5 core 2.5 GHz laptop.