Northeastern University

Combinatorial Optimization Research Project

Subset Sum Problem

Bryan Keller and Stav Rones

Professor: Waleed Meleis

30 June 2022

PART 1: DEVELOPING A BENCHMARK / EXHAUSTIVE ALGORITHM

I. Introduction

The subset sum problem is an NP-complete optimization problem that determines the following: provided with a set of integers, S, and a target value, T, is there a subset of those integers which sums up to the target value? Subset sum either returns a ‘true’ or ‘false’ depending on whether a subset’s sum is equal to the target value.

Because subset sum is NP-complete, no efficient solution has been found. Thus, the goal of this project is to determine the best algorithm to use to solve this problem.

II. Developing a Benchmark

A benchmark is a set of instances that is used as a comparison tool for each algorithm. The way it works is each algorithm will run on the benchmark, and based on the results, the most effective algorithm can be determined since they are tested on the same instances. However, there are requirements for the benchmark to provide a variety of different instances, some of which can be solved and some of which cannot. Below is the list of constraints:

1. The largest instances must be at least 20 times larger than the smallest instance
2. The structure of the instances themselves should vary
3. Some instances should be randomly generated
4. There should be at least 100 instances
5. These instances should be representative of a large class of possible instances

Based on these above requirements, six types of instances were created of varying size and complexity.

Instance I: Worst-Case Scenarios

This first case is the most basic of the instances. It is simply an array of ones varying from size 1 to size 30, for a total of 30 instances. The target value is set to 0, so the result will always turn up false. However, in making it zero, we can visualize the worst-case scenario since the algorithm must run in full. The goal of this instance type was to test the raw run-time of the algorithm and determine when it fails to compute within a 1- minute and 10-minute window.

Instance II: Average Case Scenarios

The next set of instances attempts to find the "average" case, which we defined as a situation in which there is exactly one solution in the set of numbers. This benchmark was included to see the algorithm's response to cases with a guaranteed solution. This instance type consists of 20 instances of size 20. The target value is defined as a value ranging from 1 to 20, and the set of numbers is a series of randomly placed 1's and 0's which add up to the target. See below for the pseudocode:

```
function avgCase(int num) 'num is the target value 1-20
    target = num
    indexes = num sized array w/ rand values 1-20 used for placing 1's

    for i = 1 to 20
      if (i in indexes):
        instance(i) = 1
      else
        instance(i) = 0
      end
    end
    return instance, target
end
```

Instance III: Random Function in Python [1]

For the next instances, we generated uniformly random arrays. We used the random function in Python to generate numbers in a range of [1,10E3]. The problem sizes (n) are 15, 16, …, 29 with a target value of T = (n/4)*10^3. These instances will account for 15 of the total instances. See below for pseudocode of instance III.

```
function RandArray(int num)
   for i = 1 to 15
     instance(i) =  rand_array_fn(1,10^3,num+14) 'start range, end range, size
   end
   T = num/4*10^3
   return instance, T
end
```

Instance IV: Middle-Square Method [2, 3]

This method is an early rendition of a pseudorandom number generator. Because of its short period, numbers and patterns can sometimes repeat. Though, that is not necessarily a hinderance for us in developing instances. It may be more beneficial because in a real-world scenario, a given set may not be completely random, for there may be patterns or many duplicate values. The problem sizes (n) are 15, 16, 17, …, 29 for a total of 15 instances with a target value of the next generated random number. For example, if the problem size is 24, the target value is the 25th generated random number.

The method works by generating n-digit pseudorandom numbers. First, an n-digit number is initialized and squared, which results in a 2n-digit number. The middle n digits of this number are then the next number in the sequence. However, if the resultant number has fewer than 2n digits then zeros are added in the beginning to make it 2n digits. See below for the following pseudocode:

```
function middleSM(int num)
    seed = 43512 ‘arbritrary number selected (must be even)

    for i in range(num): ‘num is the size of the array
      seed = int(str(seed * seed).zfill(8)[2:6])  ‘squares seed and adds extra zeroes if needed
      instance(i) = seed ‘add random value to array
    end

    T = int(str(seed * seed).zfill(8)[2:6]) ‘target is next random value
    return instance, target
end
```

Instance V: Xorshift [4,5]

Xorshift is another pseudorandom number generator which produces random numbers in the range of 1 to 2^{32}. This process is known to be simple and computationally fast while also appearing highly random.

Xorshift is a process that works with the help of bit shifting. Bit shifting does as it sounds. Xorshift moves the bits in a binary number to the left or the right. The void space is now a zero, and the shifted bit is removed, which results in a binary number that is completely different. Next, the XOR part compares two binary numbers, and when one of the bits is a 1, the final binary has a 1 in that place. Otherwise, a 0 is put in that spot.

Just as some of the other instances, the problem sizes are 20, 21, 22, …, 29 for a total of 10 instances. The target value is the last generated random number within the range after the arrays are created. See below for the pseudocode:

```
function xorShift(int num)
  xorshift_seed = 23525 'arbritrary
  for i in range(num):
      xorshift_seed = xorshift_seed^xorshift_speed << 13
      repeat but shift over 17 bits to right
      repeat but shift over 5 bits to the left
      xorshift_seed = xorshift_seed % int("ffffffff", 16) 'limits to 32-bit number
      instance(i) = xorshift_seed
    end

    repeat process inside for loop once but instead of setting instance(i) =, set
    target = xorshift_seed

    return instance, target
end
```

Instance VI: Linear Congruential Generator [4, 6, 7]

A linear congruential generator (LCG) is a pseudorandom integer generator. The generator itself is defined with recursion and appears in the following form:

$$Z_{n+1} = (aZ_n + c)\ mod\ m$$

$$U_n = \frac{Z_n}{m}$$

Above, $0 < a < m$, $0 \leq c \leq m$, and *mod* stands for the modulus, which gives a remainder. Regarding the variables, *m* is the range, *c* is the increment, and *m* is the multiplier. Like XORShift, the problem sizes (n) are 20, 21, 22, …, 29 for a total of 10 instances. The value of *a* is 1664525, while the value of *m* is 2^{32}, and *c* is 1013904223. The target value T is Z_{101}. LCG was chosen for its easy and fast implementation as well as its reputation for being one of the best pseudorandom number generators. LCG also does not duplicate sequences until after 2^{16} results and does not require any number storage. See below for the pseudocode:

```
Function linearCG(int num)
   a = 1664525 ‘initialize a, modulus, c, and m to chosen seeds
   modulus = 2**32
   c = 1013904223
   m = 19332
   for i in range(num):
       m = (a * m + c) % modulus
       instance(i) = m
   end

   target = (a * m + c) % modulus
   return instance, target
end
```

Below is a summary of the instance types:

Table 1: Benchmark Instances

Instance Types	Size Range	Total Instances
Worst-Case Scenario	0-29	30
Average Case	1-20	20
Random Function	15-29	15
Middle Squared Method	15-29	15
XORShift	20-29	10
LCG	20-29	10

III. Exhaustive Solution

An exhaustive solution for subset sum must consider every subset in the initial instance. For a given set of length *N*, there are 2^N subsets. An efficient way to consider every subset is to use a binary tree structure, where the root is 0 and every proceeding level represents one element of the initial set. Every node has a left child equal to 0, and a right child equal to the next element in the set. An example of this structure is visualized in Figure 1.

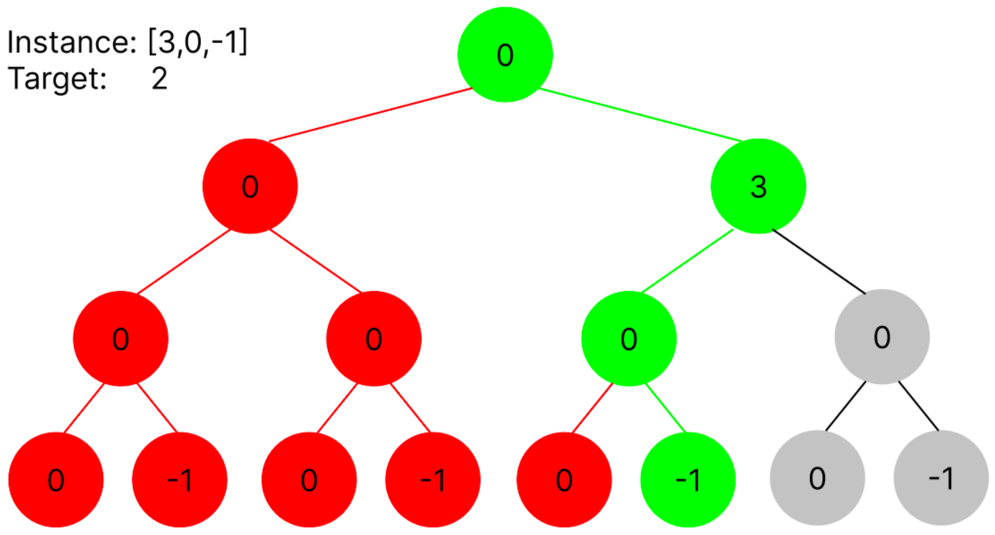

Figure 1: Subsets of instance Represented as Binary Tree

The tree shown in Figure 1 considers the sum of every subset of the instance [3,0,-1] as a path from root downwards until it reaches a leaf node. For example, the solution subset which is [3,-1], is achieved by using the path highlighted in green in Figure 1. Using the right most path considers every element in the set. The exhaustive algorithm uses Depth First Search (DFS), a tree traversal order, to find a solution. The solution first checks to see if the current node value is equal to the target, and if not, repeats subset sum for the left and right subtrees with a target equal to the difference between the current target and the node. If a solution is found, it returns immediately without checking any other subsets. This is demonstrated in the figure by showing already traversed paths in red and non-traversed paths in grey. For example, the solution [3,0,-1] is not returned because a different solution was found before it was traversed. To return false, the solution must check every subset, or every node of the tree. This makes the runtime complexity of this solution $O(2^n)$.

The memory complexity would be the same, but an optimization is implemented to reduce it to $O(n)$. Namely, the entire left tree for any node is explored before the right node is created, and once a subtree is determined to not contain a solution, it is removed from memory. This means that the largest the tree gets in memory is limited to the size of the list. Pseudocode demonstrating this exhaustive algorithm is shown below.

```
Function solveBinTreeRecursive(node, lst, target):
    n = lst.length()
    if (node.val == target):                                              <-- Solution found
        return True
    if (n == 0):                                                          <-- Base case: Reached leaf node
        return False
    if (list.size not = 0):                                               <-- More sets to check
        node.left = BNode(0, node)
        if (solveBinTreeRecursive(node.left, lst[1:n], target - node.val)):     <-- Left subtree
            return True
        node.right = BNode(lst[0], node)
        if (solveBinTreeRecursive(node.right, lst[1:n], target - node.val)):    <-- Right subtree
            return True
    if (node.parent.left == node):                                        <-- Delete node before returning to parent
        node.parent.left = None
    else:
        node.parent.right = None
    return False                                                          <-- Input tree does not contain solution

Function solveInstance(instance, target):                                 <-- Call this to start
    root = BNode(0, None)
    return solveBinTreeRecursive(root, instance, target)
```

IV. Results of Exhaustive Search

To determine the experimental worst-case runtime, the first 30 benchmarks consist of sets with no solutions of increasing size from 0 to 29. Figure 2 shows experimental results for set sizes 0 to 18 and 19 to 29.

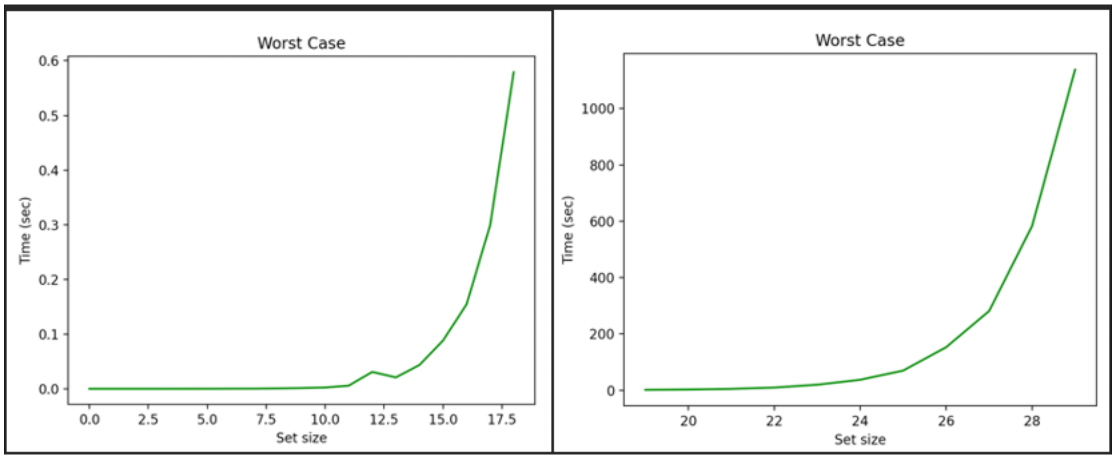

Figure 2: Worst-Case Time vs. Size

The exhaustive algorithm, with a one-minute limit on CPU time, solved 24 of the 29 worst-case instances, which is 82.7%. With a ten-minute limit, the algorithm solved 27 of the 29 instances, which is 93.1%.

Next, the “average” case was analyzed as a function of target value. Based on the graph below, the increase in target value does not make a significant impact on the time it takes to find a solution. Instead, the set size, which was not altered here, makes the biggest difference in time to find a solution.

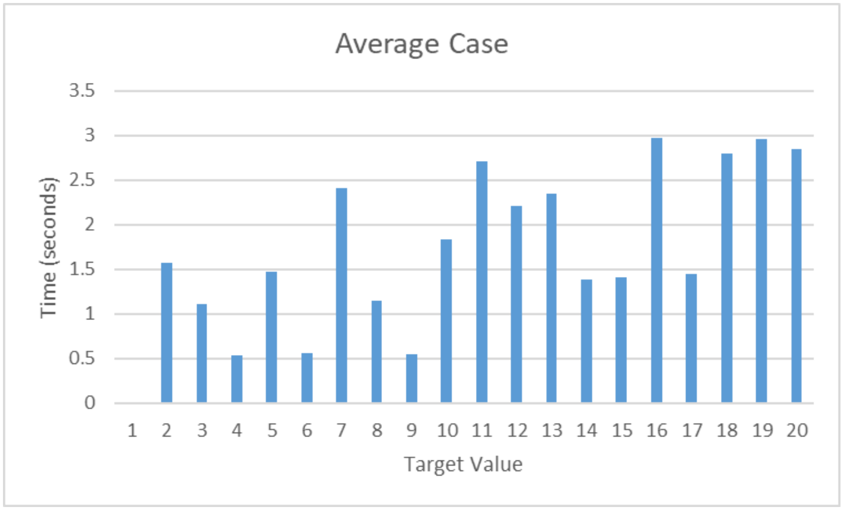

Figure 3: Average Case Target Value vs. Time

With both a one-minute and ten-minute time limit, the algorithm solved every instance of average case. However, the goal of these instances was to capture an “average” value solve time. The average of all 20 instances was 1.71 seconds.

For the random set of instances, a graph like Figure 1 was created to show the same relationship of set size vs. time.

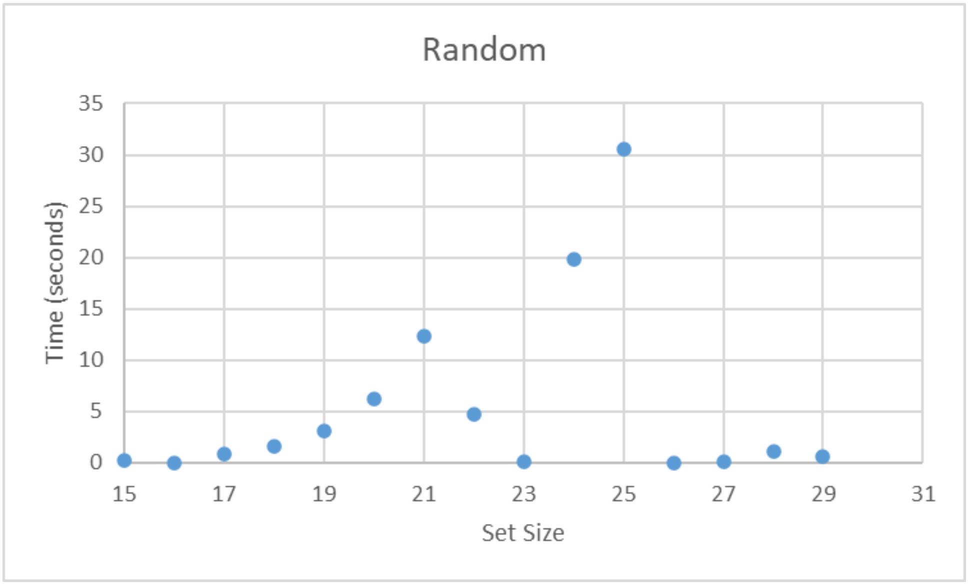

Figure 4: Average Run Time for Random Instances of Varying Size

In both the 1-minute and 10-minute CPU limit all instances were solved for the random set of instances.

Next, the middle-squared method (MSM) instances were run to find the one-minute and ten-minute time limit. The graph below shows how the set size impacts the time to takes to solve.

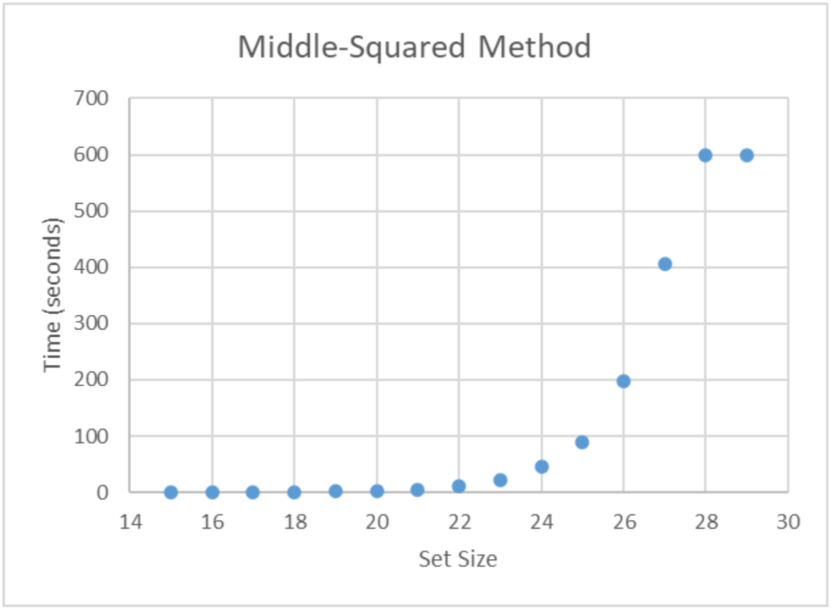

Figure 5: Middle-Squared Method Set Size vs. Time

For MSM, in the 1-minute CPU time limit 24 of the 29 instances are solved, which is 82.7%. Within the 10-minute limit, the algorithm solves 27 of the 29 instances, which is 93.1%. After MSM, XORShift was checked for its solve rate within the two time limits. The graph below demonstrates how set size impacts the time limit.

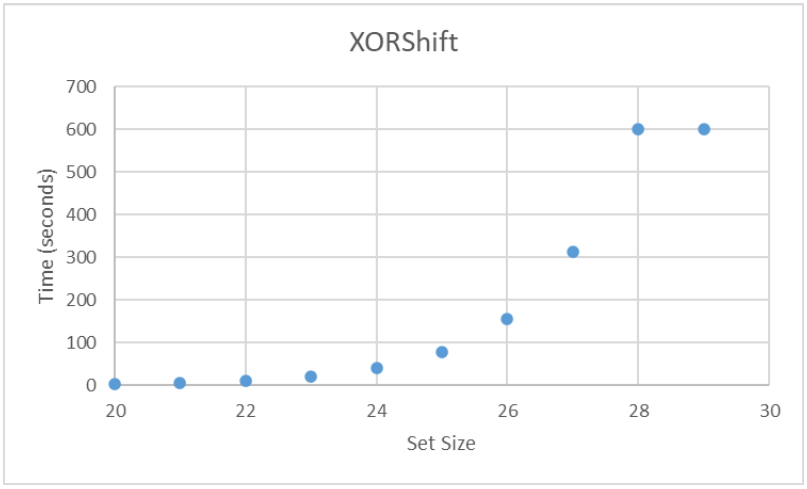

Figure 6: XORShift Set Size vs. Time

For XORShift, the 1-minute CPU time limit yields 24 solved instances or 82.7%, and the 10-minute CPU time limit yields 27 solved instances, or 93.1%.

Finally, the LCG was checked for the same as above. Its 1-minute CPU limit allowed for 24 solved instances, or 82.7%, and the 10-minute CPU limit allowed for 27 solved instances, or 93.1%.

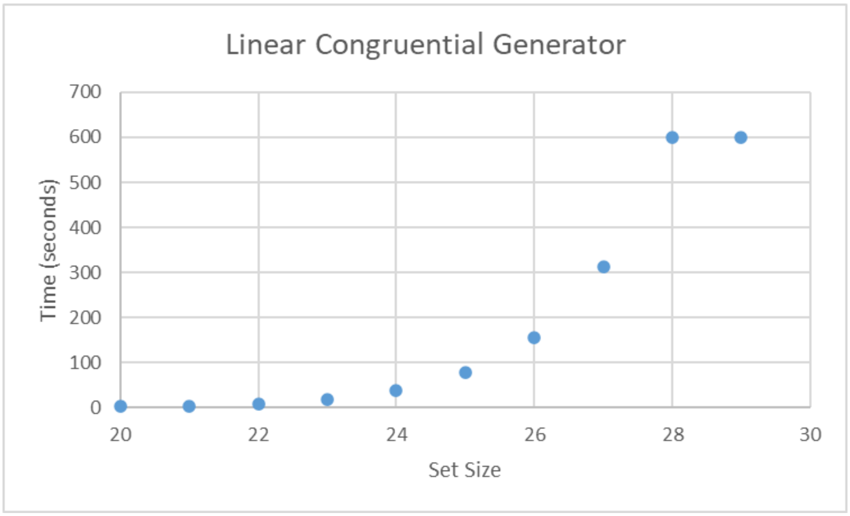

Figure 7: LCG Set Size vs. Time

A table is included below to give a summary of each instance type and its respective 1-minute and 10-minute solve rate.

Table 2: Solve Rate for Each Instance Type

Instance Types	1 Min Limit # Solved	1 Min Limit % Solved	10 Min Limit # Solved	10 Min Limit % Solved
Worst-Case Scenario	24	82.7%	27	93.1%
Random Function	29	100%	29	100%
Middle Squared Method	24	82.7%	27	93.1%
XORShift	24	82.7%	27	93.1%
LCG	24	82.7%	27	93.1%

V. Conclusion

The main purpose of project 1 is to create a starting point for how to solve a subset sum problem efficiently. Making a benchmark of varying types of instances allows us to compare algorithms based on run-time and solve rate. These are both quantifiable metrics that are important in the world of computer science. Obviously, the goal for an efficient algorithm is to reduce the run-time and increase the solve rate.

This first project dove into the exhaustive algorithm, which is one of the more rudimentary types of algorithms because of its complete approach that looks through every single combination of subsets. This may or may not be the most effective algorithm to use, but it creates a base case to use for comparison.

PART 2: EXPLORING COMPLEXITY SPACE

I. Introduction

As a reminder, the subset sum (SS) problem can be stated as follows: given set of n integers, does there exists a subset whose sum equals a target value, T [8]. To attain a better understanding of the subset sum problem, the complexity of both sub and super problems are analyzed to determine SS's place in the complexity space. Tractable sub problems can often be used in heuristic algorithms to approximate or accelerate parent problems. SS has subproblems that exist in both NPC and P.

II. Exploring Complexity Space

Subproblems of SS include subset sum positive (SSP), subset sum zero (SSZ), subset partition (SP), 3SUM, and kSUM. These variations and their algorithms are discussed in Section III. Parent problems include the knapsack problem and the multiple subset sum problem [9].

The knapsack problem says that given a set of n items each with weight w_I and a specific value v_i, find a subset of those items such that the total weight is less than a given upper limit, and the total values are maximized [10]. In simpler terms, it can be described as the most effective way to fill a limited-size backpack with items of varying weights and varying values to maximize the total value.

The second super-problem of subset sum, the multiple subset sum problem (MSSP), has multiple variations, but the main input is the same. The input is a set S containing n integers and m number of subsets. The first variant, Max-Sum MSSP, states that for every subset there is a maximum capacity, and the objective is to make each subset's sum as large as possible such that each subset's sum is less than or equal to the capacity. The next variant, Max-Min MSSP, is like the Max-Sum MSSP, but its objective is instead to make the smallest subset sum such that the sum does not exceed the capacity. The final variation, the Fair SSP, has no capacity. Instead, each subset is assigned to a person, and the goal is to create subsets that exhibit fairness [11].

Figure 1 shows the relationship SS to the super and sub problems discussed.

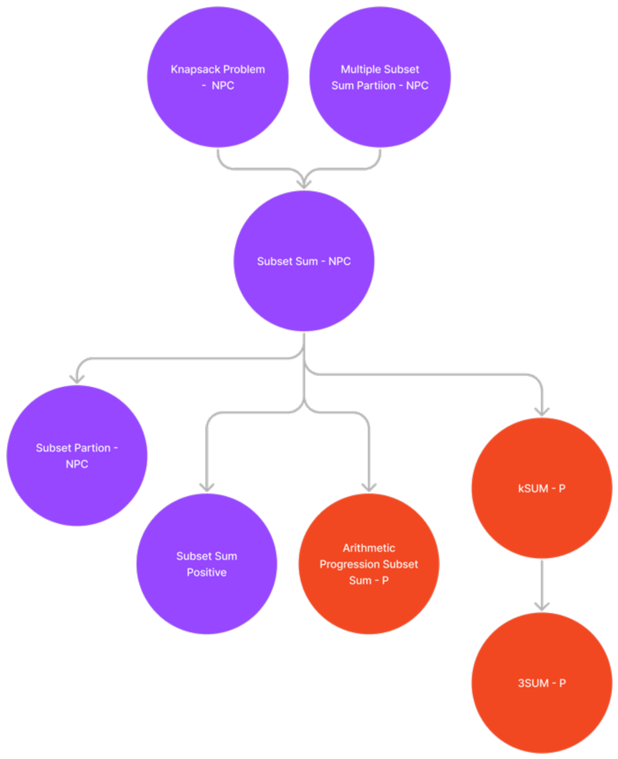

Figure 8: Complexity Space of Subset Sum

III. Complexity of Sub-Problems

Subset Partition (SP)

Subset partition looks to find if a given set of integers can be partitioned into two subsets, each of which containing the same sum [8]. The sub-problem subset partition is NP-complete. However, there are heuristics that can oftentimes solve this problem optimally or approximately [12]. In fact, there are pseudo-polynomial time algorithms for this problem such that the algorithm's run time is polynomial but only in the input's values. Sometimes in SP, there exists an exponential number of optimal solutions such that it is often referred to as the "easiest hard problem." SP is not minimally polynomial-solvable because it is unsolvable in P. SP is considered minimally NP-complete as there are no sub-problems that are known to be NP-complete.

Subset Sum Positive (SSP)

Subset sum positive has the same definition as subset sum, however, the given set consists of positive integers [9]. This problem allows for the simplification of ignoring all sets that contain a subset whose sum is larger than the target. Although a true solution still has an exponential time complexity, in practice many subsets do not need to be checked if a subset has a sum that is already greater than the target.

3SUM & KSUM - P

3Sum is a subset of SS that can be defined as follows: given a set of n integers, is there a set of 3 elements whose sum is zero? This problem is a subproblem of subset sum in that the possible solutions are limited to a size of 3 instead of n, which reduces the total number of possible solutions from 2^n to $nC3 = O(n^3)$. 3Sum is a generalization of kSUM, which uses an arbitrary size k for the size of the subset that should be equal to 0. The search space of kSUM is also polynomial, namely $O(n^k)$. kSUM has been shown to have a time complexity of $O(n^{k/2})$ when using $O(n^{k/2})$ memory space [13]. A recursive algorithm for solving kSUM is shown below in Algorithm 1, which reduces k-sum to (k-1)-sum all the way down to 2-sum.

```
kSUM(instance: [int],  k: int, target: int) {

    if (k == 1) return (k in instance)

    n = length(instance)
    for $i \in [1, n-1]$ {
        if (kSUM(instance[i:n], k-1, target- instance[i])) {
            return True
        }
    }

    return False
}
```

Arithmetic Progression Subset Sum (APSS)

Arithmetic Progression Subset Sum (APSS) is a subproblem of SS where the set is an arithmetic progression, namely $a_i = a_1 + (i-1) * j$. The whole set can then be simplified to a triple of (a_1, n, j). This is shown to be solvable in polynomial time in [14] using the following observation:

Observation. Let a, n, k be integers with $1 \leqslant k \leqslant n$. Then, if there exists $s \subseteq T = \{a, a+j, \dots, a+(n-1)j\}$ with $|s| = k$, $s \neq \{a+(n-k)j, \dots, a+(n-1)j\}$ such that $\sum_{i\in s} i = t$ then there exists $s' \subseteq T$ with $|s'| = k$, such that $\sum_{i \in s'} i = t + j$.

If there exists a $k = a^{-1}t(mod\ j)$ then there is a solution, and the k can be found using binary search in polynomial time.

IV. Conclusion

When analyzing sub problems of SS, we see that two variants are solvable in polynomial time, namely KSUM and APSS. If there are portions of the set that are arithmetic, then they can be searched for target subsets in polynomial time instead of exponential time, which can reduce the exponential runtime of a SS problem with size n. Additionally, the kSUM subproblem can be used to find all subsets of size k in polynomial time, up to a k where k is not dependent on n. This can be very useful if there is a high probability of achieving the target sum with a subset size less than or equal to k. These subproblems can further be used to help solve parent problems or other NPC problems of SS such as KP and MSSP.

PART 3: GREEDY ALGORITHM

I. Introduction

The greedy algorithm is a relatively simple approach that selects the best possible choice at that current time. In this iteration of the project, this algorithm was developed and implemented to solve instances of subset sum. Specifically, it was run on our benchmark of 100 instances. We sought to find where greedy worked the best and where it did not, and to see how it compared to our exhaustive algorithm.

II. Greedy Algorithm Explained

We designed our natural greedy algorithm to be simple, and we did not make much effort in trying to improve it. The algorithm works by first sorting the instance in descending order.

```
GreedySolver(instance[], target)
{
    sort instance[] in descending order
    sum = 0

    for i = 1 to length(instance)
    {
        element = instance[i]
        if sum + element <= target then
            sum = sum + element
    }

    if sum + instance[end] - target < target – sum then
        sum = sum + instance[end]

    return sum
}
```

Then, it runs through a ‘for’ loop which iterates through n times, in which n is the length of the instance. In each iteration, the algorithm checks, in order, if the element plus the sum (which is initially set to zero) is less than the target value. If it is, that element value is added to the sum. Once the loop completes all its iterations, it reaches one last ‘if’ statement. This statement says that if sum + last element of the instance – target value is the less than target – sum, then add the final instance value to the sum. See below for the pseudocode:

III. Results

The greedy algorithm has time complexity $O(n * \log(n) + n)$ from the python Timesort algorithm followed by a single iteration of the list. This resulted in producing a result for every instance in around $10^{-6}\ s$. The greedy algorithm does not always find an optimal solution, as seen by the example in Figure 1. The greedy algorithm first picks 20 because it is less than 23 and does not select 11 because $(20 + 11 - 23) > (23 - 20)$. The accuracy of the greedy solution is only 87%, while the optimal solution is 100% accurate.

$$S = \{20, 12, 11\}, t = 23$$
$$A_{greedy} = \{20\}$$
$$A_{optimal} = \{12, 11\}$$

Figure 9: Greedy Algorithm Failure

Overall, the greedy algorithm performed relatively well on our benchmark. Accuracy was one of the metrics of success, and it is calculated as the greedy algorithm’s result divided by the target value. On instances 1 through 29, the accuracy was 0, or rather undefined, because these instances were arrays of 1’s with a target value of 0. However, on instances 30-100, the accuracy only fell below 0.9 or 90% three times. See below for the graph of the instances vs. the greedy accuracy.

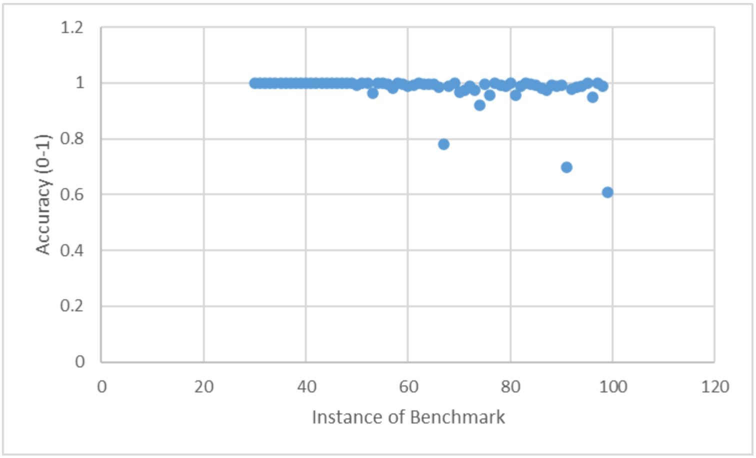

Figure 10: Benchmark Instance vs. Greedy Algorithm Accuracy

A table was also developed to show a different representation of the figure above. This table shows the number of instances above a certain accuracy. Note, this table only shows 71 total instances because samples 1-29 have undefined accuracy.

Table 3:Number of Instances Above Given Accuracy

Number of Instances	Accuracy Range
20	100%
46	> 99%
58	> 98%
62	> 97%
64	> 96%
66	> 95%
68	> 90%
68	> 80%
69	> 70%
71	> 60%

On samples 30-100, we also investigated how instance size impacts the greedy accuracy. There was no clear correlation between the two. The algorithm performed equally well during the smaller instance sizes of 15 and the larger instance sizes of 30. There were a few outliers.

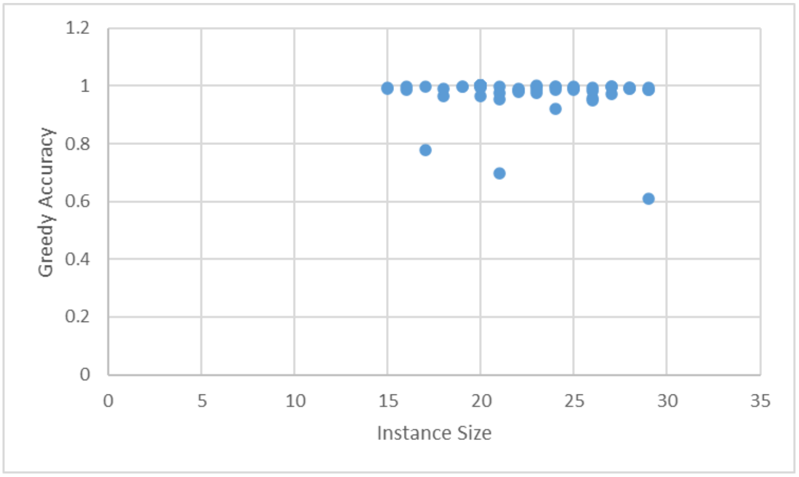

Figure 11: Instance Size vs. Greedy Algorithm Accuracy

Now that we have two algorithms, exhaustive and greedy, we can compare their results. The exhaustive algorithm returns a true if a solution was found. It returns a false if either no solution is found, or the 10-minute time limit is hit. The exhaustive algorithm did not track "near" solutions because it was always aiming for the exact target value. In total, the exhaustive solution found 28 true solutions. On the other hand, the greedy algorithm only found solutions for 20 instances. The tables below break down the number of solutions found for each algorithm.

Solution?	Number of Instances
False (Ran Out of Time)	7
False (No Solution)	64
True	28

Figure 12: Solutions Found for Exhaustive Algorithm

Solution?	Number of Instances
True	20
False (No Solution)	79

Figure 13: Solutions Found for Greedy Algorithm

Next, here is the number of correct solutions found in greedy vs. the exhaustive search at each instance size. It is interesting to note that instance size does not have a large impact on these algorithms.

Table 4: Correct Solutions Found in Greedy vs. Exhaustive at Different Instance Sizes

Instance Size	Greedy	Exhaustive
15	0	0
16	0	1
17	0	0
18	0	0
19	0	0
20	20	20
21	0	0
22	0	1
23	0	1
24	0	1
25	0	1
26	0	1
27	0	1
28	0	1
29	0	1

Each algorithm has its benefits and drawbacks. For increased accuracy in finding a perfect solution, the exhaustive algorithm is the best option. However, for speed and simplicity, greedy is better. The desired algorithm depends on the application of subset sum and the severity of a right vs. wrong answer.

IV. Conclusion

The main purpose of Project 3 is to acquire another algorithm in our tool belt and see its effectiveness for the subset sum problem. Through this process, we have seen the benefits of the greedy algorithm as a quick, simple, and relatively accurate algorithm, at least for our benchmarks of subset sum. We have also seen its shortcomings, in that perfect accuracy is not its main priority, rather, getting close is good enough. We will continue to test different algorithms to find the best possible fit for subset sum.

PART 4: ILP ALGORITHM

I. Introduction

This iteration of the project dives into solving the subset sum problem with the use of integer linear programming (ILP). ILP is another solver that looks to minimize or maximize a linear function, subject to constraints. The constraints usually consist of linear equalities and inequalities, and being that it's called ILP, the variables must be integers.

II. ILP Formulation

As a reminder, subset sum seeks to find the following: given a set of integers (*S*), is there a subset (*s*) such that the sum of the integers in that subset equal a target value (*T*). The ILP formulation is as follows:

Maximize: $\sum_{i=1}^{n} w_i x_i$

Constraint 1: $\sum_{i=1}^{n} w_i x_i \leq T$

Constraint 2: w_i must be an integer for i = 1, 2, …, n

Above, x_i represents a binary integer. If x_i is 1, then that component is included in the subset, but if x_i is 0, then the component is not included in the subset. Additionally, w_i represent the weights of the integers. These weights are assigned uniquely for each benchmark we have previously established.

III. Results

LP Bound

Using AMPL and CPLEX to find the LP bound, we removed the constraint that x_i must be binary integers. Otherwise, it has the same formulation as ILP. Because LP is not constrained to integrality, a solution for every instance was found. The only way a solution could not be found is if the sum of every component of the set does not reach or exceed the target value. This is never the case with our benchmarks. Thus, LP bound is 100% accurate.

ILP Solution

Using AMPL and CPLEX, we applied the ILP to all instances in the benchmark. Out of the 100 instances, ILP found exact solutions for 56 of them. For instances 1-30, a solution was found for each case. With a target value of 0 and an array of all 1's with varying size, these instances all had the same solution as an empty subset. Thus, their accuracy was undefined because the target value was set to 0. For the remaining instances, the accuracy was calculated, and can be seen in the figure below.

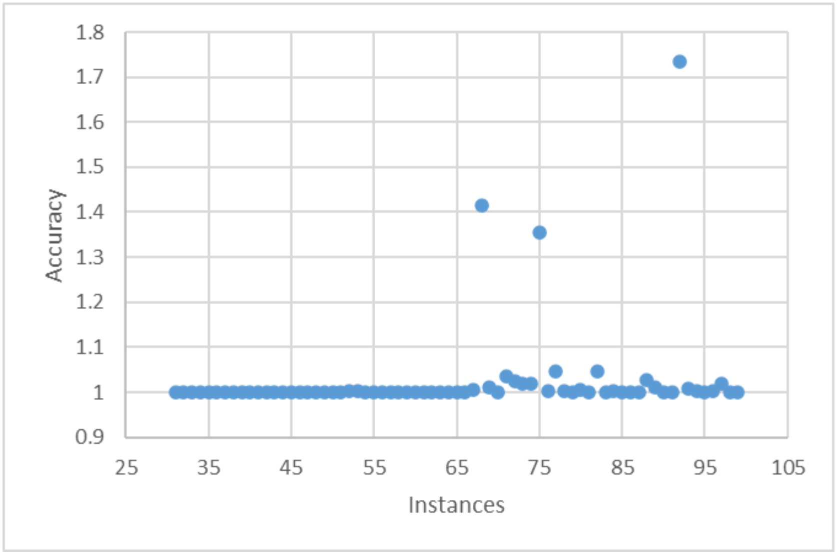

Figure 14: Accuracy of Every Instance in ILP

Evident from the graph, all but 3 instances found a subset solution within 5% of the actual solution. In addition to the high accuracy, to solve each individual instance takes no longer than 5 seconds. The figure below shows each instance, and the time it took to find a solution using ILP.

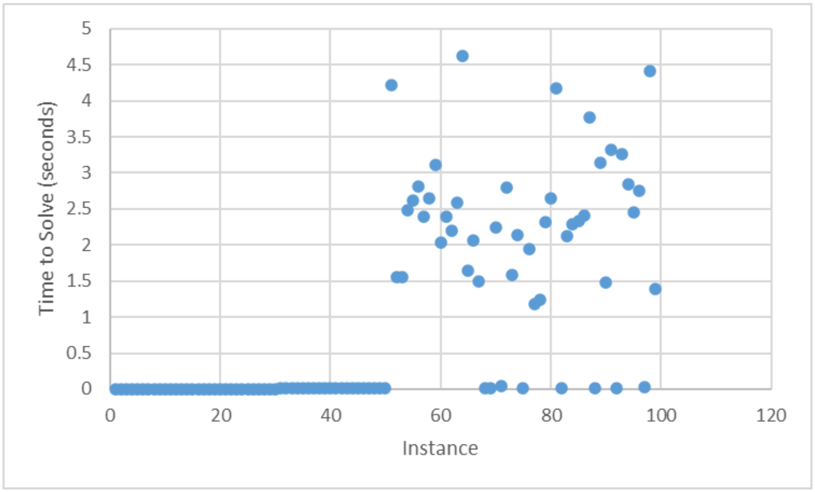

Figure 15: Instance vs. Solve Time

Regarding solve time, for large instances, the solve time does not go up dramatically. It is generally true that a smaller instance size is solved quicker than a larger instance size, but the difference is not significant on the benchmarks we used. All instances can be solved within 103 seconds, and larger instances of size 25 or more account for 42 of those seconds. Instances of size 20 or more account for 85 of those seconds. The graph below shows the relationship between instance size and the time to solve.

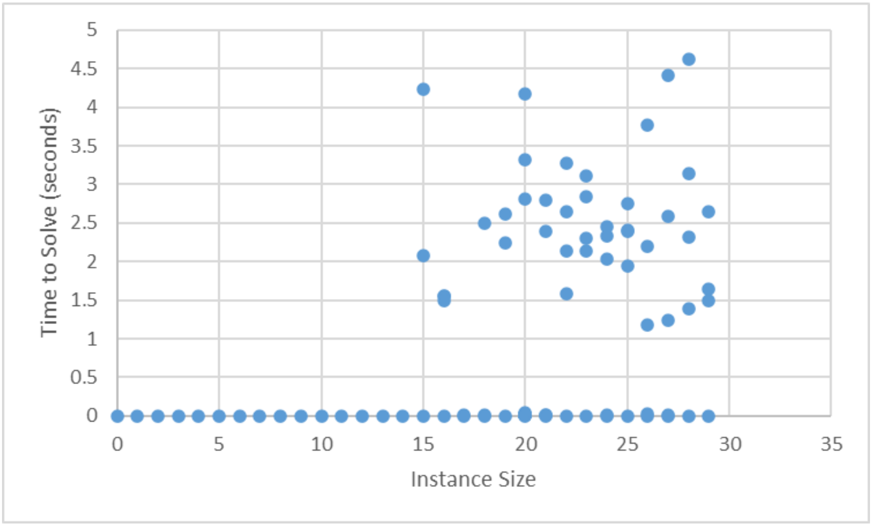

Figure 16: Instance Size vs. Solve Time

Greedy and LP Bound Optimality

Exact solutions were found for all instances in the greedy algorithm, LP lower bounds, and ILP. There were no instances in which only greedy and LP lower bounds were found, and ILP was not. To assess the quality of the greedy solution and LP lower bound, we found the relative size of the gap between the target value and the exact solution. In addition, we found the percent difference between the target and the exact. The formulas to calculate both are below.

$$Relative\ Gap = \left|\frac{Target\ Value}{Found\ Solution}\right|$$

$$Percent\ Error = \frac{Target\ Value - Found\ Solution}{Found\ Solution} * 100$$

For the both the greedy algorithm and the LP lower bound, the first 30 instances could not be evaluated because the found solution was always 0 but as was the target value. Effectively, the absolute gap here was zero, and the percent error was 0. For the remaining instances, the relative gap and percent error were calculated and graphed on the figures below.

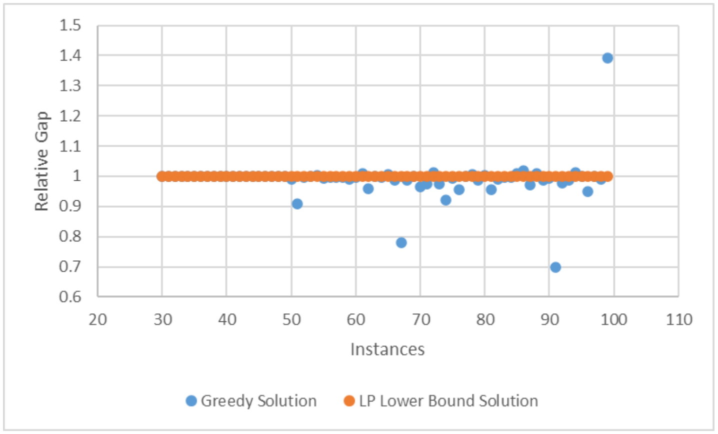

Figure 17: Relative Gap for Instances 30-100 for Greedy and LP

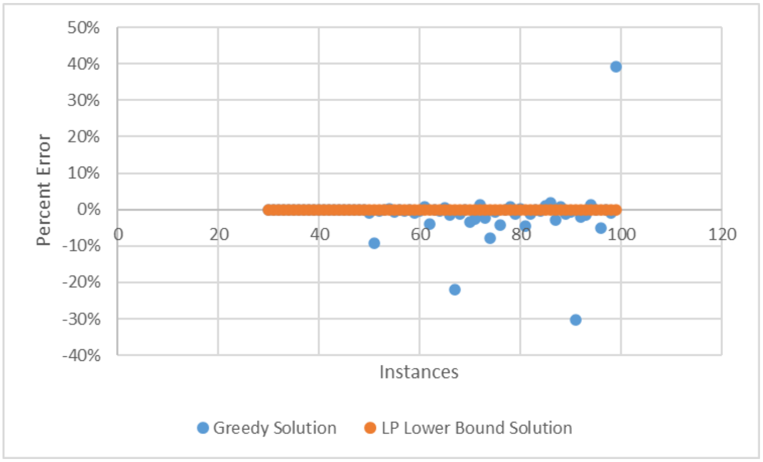

Figure 18: Percent Error for Instances 1-30 for Greedy and LP

For greedy, the average relative gap for instances 30-100 was 0.990365, which is close to the ideal value of 1, and the average percent error -0.9635%. Of the 100 instances, the greedy solution was optimal for 48 total instances.

For LP, the average relative gap for instance 30-100 was 1 while the average percent error was 0%. Of the 100 instances, the LP lower bound solution was optimal for every instance. However, it is important to note that the LP solution is not a feasible solution.

ILP Compared to Exhaustive

Compared to ILP, exhaustive is a much more time-consuming optimization technique. ILP solved every instance of the benchmark within 103 seconds, with no single instance taking longer than 5 seconds. The exhaustive algorithm, on the other hand, only found a solution for 79 of the instances within a 1-minute time limit and 91 in a 10-minute time limit. For the 91 solutions it solved, the average solve time was 30.5 seconds, which means it took 46 minutes to solve those 91 instances.

As for the accuracy of exhaustive, it's difficult to compare to ILP. The exhaustive algorithm did not track "near" solutions because it was always aiming for the exact target value. In total, the exhaustive solution found 91 solutions with 28 being true and 63 being false. The remaining 9 were unsolved in 10 minutes. This is a solve rate of 91% with 100% perfect accuracy. ILP has a solve rate of 100% but with a perfect accuracy of 56%. This is assuming perfect accuracy as finding a true or false solution.

In summary, if one is looking for an optimization technique with a quick run-time and a solution that fits within +/- 1% of the actual error, then ILP is the best option. However, if an exact solution is absolutely necessary, then the exhaustive technique would be the best bet.

IV. Conclusion

ILP has proven to be a very effective optimization technique for our instances of subset sum. Not only is the solve time quick, but the accuracy is high, with only three of the instances exceeding a deviation from the actual value of greater than 5%. This is an excellent tool to have in our toolbelt when solving optimization problems.

PART 5: DYNAMIC PROGRAMMING / LOCAL SEARCH ALGORITHMS

I. Introduction

In the final part of the project, we explore two more optimization algorithms to use on the subset sum problem. The first of which is the steepest descent local search algorithm that uses the greedy and random techniques to find an initial solution, and a 1-Opt neighborhood. The second type is top-down dynamic programming. We test these both on our benchmark and compare them along with the previous optimization algorithms: exhaustive search, greedy solution, integer linear programming (ILP), and LP lower bound.

II. Local Search Algorithm

For the local search algorithm, an initial solution was first generated using a random solution and a greedy solution. The greedy solution sorted the instance, and then added numbers from largest to smallest until no more numbers could be added without exceeding the target. The random solution chose a random subset of the instance. The neighborhood was defined using the 1-Opt algorithm. In this algorithm, the neighborhood is defined as every subset that either includes or excludes 1 element relative to the original solution. This is achieved by iterating the instance, and for each iteration if the element is included in the subset solution than it is excluded, and if it is excluded then it is included. This is repeated for every element, and a neighbor is deemed to be more optimal if its sum is closer to the target than the original solution's sum. The local search ends when either the solution equals the target, or the neighborhood contains no closer solutions. In the case when there are two neighbors that are equidistant from the target, a tabu list of length 1 is used to ensure that an endless cycle does not occur.

For local random search, the average number of iterations required was 8.55 for each instance. The number of iterations fell between 1 and 20 per instance. As for the computation time, the total time required was 0.125 seconds with the longest instance taking 0.0195 seconds. Thus, we did not have to change the solve limit because everything was solved very quickly.

As for local greedy search, the average number of iterations was 1.11 and fell between 1 and 3 per instance. Like the local greedy search, the computation time required was very short.

The total time was 0.0073 seconds with the longest instance taking 0.00162 seconds to solve. Therefore, the solve limit did not have to be adjusted.

III. Dynamic Programming Algorithm

Dynamic programming (DP) is an optimization technique that exploits the optimal substructure of the subset sum problem. The pseudocode for DP is shown below:

```
memo = {}

function dynamic_programming(instance: list[int], target: int):

    global memo

    if target == 0:
        return true

    if len(instance) == 0 or target < 0:
        return false

    key = (len(instance), target)

    if key not in memo:

        include = dynamic_programming(instance[0:-1], target - instance[-1])

        exclude = dynamic_programming(instance[0:-1], target)

        memo[key] = include or exclude

    return memo[key]
```

This DP algorithm is an inclusion – exclusion with top down memoization. The function checks if the given instance and target pair has already been solved. If not, then it recurses with a subset of the instance minus the last element for either the inclusion or exclusion case. In the inclusion case, the new target is the target minus the value of the last element, and for the exclusion case the target remains the same. The base case is when the target is 0, meaning a solution was found, or the target is less than 0, meaning any further subset would exceed the target.

We ran this on our benchmark of 100 instances. Out of all the instances, 28 of them had a true solution while 72 of them did not. Regarding the run time, every single instance was solved in 0.453 seconds with the slowest one taking 0.108 seconds and the fastest one taking 3.2e-07 seconds. The average time to solve an instance was just 0.00453 seconds. Below, we graphed how the instance size affected the run time.

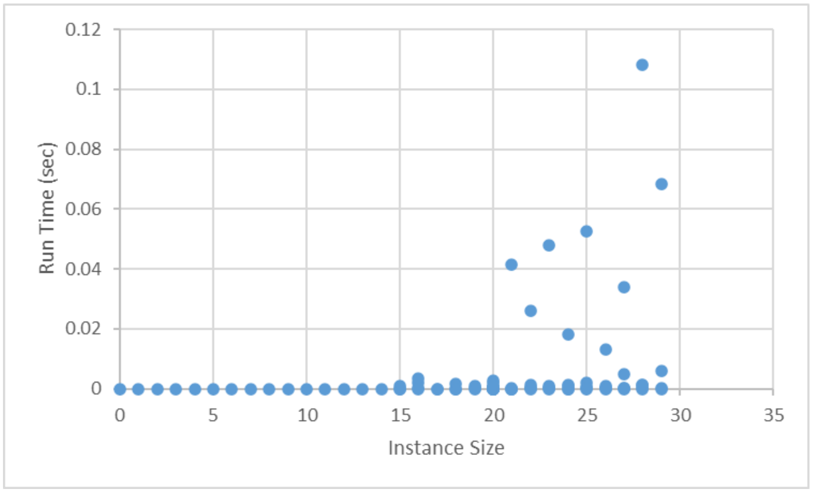

Figure 19: Instance Size vs. Run Time - Dynamic Programming

There is a relationship between instance size and run time, with the higher instance sizes requiring more time to find a solution. However, the time it takes to solve even the slowest instances is insignificant.

IV. Evaluating All Algorithms

We have now evaluated multiple optimization algorithms including exhaustive search, greedy, ILP, LP lower bound, local search, and DP. These methods had varying degrees of success on the subset sum problem.

One of the ways to measure success is to analyze the run time of the algorithm. The longer it takes to run, the more difficult it may be to implement, especially for instances of large size. Below, we have created a table which shows some of the metrics regarding run time (in seconds).

Table 5: Run Time Metrics of All Algorithms

Algorithm	Total	Avg	Max	Avg Instances (0-10)	Avg Instances (10-15)	Avg Instances (15-20)	Avg Instances (20+)
Exhaustive (10 min)	7794	77.9	600	0.000609	0.055	1.12	111
Greedy	1.02E-03	1.02E-05	9.84E-05	3.84E-06	5.02E-06	9.85E-06	1.22E-05
ILP	101.6	1.02	3.76	0.00	0.717	0.725	1.15
DP	0.453	0.0045	0.108	4.66E-07	1.30E-04	4.59E-04	0.0120
Local Random	0.125	0.00125	0.0195	6.16E-05	2.38E-04	5.00E-04	0.00337
Local Greedy	0.00726	7.26E-05	0.000162	1.90E-05	4.28E-05	7.64E-05	0.000102

The run time metrics included are the total time to run all 100 instances, the average time to solve an instance, the maximum time to solve an instance, and the average time to solve instances of size 0-10, 10-15, 15-20, and 20+. One key takeaway is that the exhaustive solution is the most time-consuming by a significant margin and the most affected by large instance sizes. As for greedy, DP, local random search, and local greedy search, these solutions are found almost instantaneous and are relatively unaffected by large instance sizes. The time is takes to solve them does increase for a larger instance size, but the increase is marginal. ILP is somewhere in between. Though, it is still fast compared to most algorithm standards.

The next measure of success is based on solution quality. Some of the algorithms had an "all or nothing" approach, which means they search for a subset that sums up to the target. If they find one, they return a true, but if they don't, they return a false. These algorithms include exhaustive search and dynamic programming. To compare these, we looked at how many solutions it successfully classified as having a subset. The table below demonstrates how many instances each algorithm classified as true and as false, and it displays the success rate based on the confirmed number of true instances, which is 30. The success rate is the number of true values classified by the confirmed number of true instances.

Table 6: Exhaustive and DP Classification Results

Algorithm	TRUE	FALSE	Success Rate
Exhaustive (10 min)	28	72	93.3%
DP	28	72	93.3%

As evident from the table, both exhaustive and greedy have the same success rate. Considering both run time and success rate, DP appears to have the edge because of its faster run time.

Other solutions, like ILP, greedy, local search, and LP lower bound did not have such an approach. Instead, they return values that were as close to the solution as possible. These approaches were measured in a different way. For these, we measured the absolute percent error of instances 30-100, since 1-29 were guaranteed to be false, so there was no point in looking at those. Absolute percent error is defined as the following formula:

$$Absolute\ Percent\ Error = \frac{|Target\ Value - Found\ Solution|}{Found\ Solution} * 100$$

We found the percent error for every instance and averaged those up over 100 instances. We also found the percent error for certain instance sizes, as well. Here are the results for each algorithm:

Table 7: Absolute Percent Error for Algorithm

Algorithm	Avg % Error	Max % Error	% Err Instances (15-20)	% Err Instances (20-25)	% Err Instances (25+)
Greedy	2.5%	43%	1.4%	1.8%	2.8%
ILP	2.6%	74%	1.4%	2.9%	0.6%
LP Lower Bound	0%	0%	0%	0%	0%
Local Random	2.3%	28%	1.6%	1.7%	3.7%
Local Greedy	3.6%	74%	1.7%	3.3%	2.9%

Overall, each algorithm performed relatively similar. For the average percent error, LP lower bound is the best solution, but it is not feasible because it does not have a binary constraint. Thus, a subset of that make-up does not exist. Next best is local random search, which has 2.3% average error, though it performs the worst on instances larger than 25. For consistency through instances of every size, ILP may be the best optimization technique because it only has a 0.6% error on large instances and an overall percent error of 2.5.

V. Conclusion

Based on our analysis, there is no one true method that is clearly the best for subset sum. It entirely depends on one's needs. For example, if one needs the exact solution in a timely manner, then dynamic programming is the best option. However, if one needs a close solution as quickly as possible, then local greedy search is the best option. There are also other factors we did not consider such as instance sizes larger than 30. There is a chance that the increase in instance size makes a significant impact that we did not discover on one of the algorithms. In short, this analysis is a great start in assessing what optimization technique may be successful in solving subset sum.

References

[1] H. Liao and T. Wang, “Research Project of Subset Sum Problem,” 07-Dec-2013. [Online]. Available: http://liaohaofu.com/documents/combinatorial_optimization/final_project.pdf. [Accessed: 16-May-2022].

[2] “Middle-square method,” Wikipedia, 08-Dec-2021. [Online]. Available: https://en.wikipedia.org/wiki/Middle-square_method. [Accessed: 16-May-2022].

[3] “Pseudo-random numbers middle-square method,” Pseudo-random numbers/Middle-square method - Rosetta Code. [Online]. Available: https://rosettacode.org/wiki/Pseudo-random_numbers/Middle-square_method#Python. [Accessed: 16-May-2022].

[4] V. Koushik, “3 amazing ways to generate random numbers without math.random(),” Github.com, 04-Oct-2019. [Online]. Available: https://svijaykoushik.github.io/blog/2019/10/04/three-awesome-ways-to-generate-random-number-in-javascript/. [Accessed: 16-May-2022].

[5] G. Marsaglia, “Xorshift rngs,” Journal of Statistical Software, vol. 8, no. 14, 2003.

[6] “Linear Congruential Generator,” Wikipedia, 20-Apr-2022. [Online]. Available: https://en.wikipedia.org/wiki/Linear_congruential_generator. [Accessed: 16-May-2022].

[7] K. Sigman, “Random number generators - Columbia University,” Columbia. [Online]. Available: http://www.columbia.edu/~ks20/4106-18-Fall/Simulation-LCG.pdf. [Accessed: 16-May-2022].

[8] “NP-complete problems and approximation problems,” Chapter 35. [Online]. Available: http://www2.lawrence.edu/fast/GREGGJ/CMSC510/Ch35/Chapter35.html#:~:text=The%20subset%20sum%20problem%20is,sum%20is%20close%20to%20t. [Accessed: 23-May-2022].

[9] “Subset sum problem,” Wikipedia, 15-May-2022. [Online]. Available: https://en.wikipedia.org/wiki/Subset_sum_problem. [Accessed: 23-May-2022].

[10] “Knapsack problem,” Wikipedia, 18-May-2022. [Online]. Available: https://en.wikipedia.org/wiki/Knapsack_problem. [Accessed: 23-May-2022].

[11] “Multiple subset sum,” Wikipedia, 07-Jun-2021. [Online]. Available: https://en.wikipedia.org/wiki/Multiple_subset_sum. [Accessed: 23-May-2022].

[12] K. Gokcesu and H. Gokcesu, “Efficient locally optimal number set partitioning for scheduling, allocation and fair selection,” arXiv.org, 10-Sep-2021. [Online]. Available: https://arxiv.org/abs/2109.04809. [Accessed: 24-May-2022].
[13] Goldstein, Isaac, Moshe Lewenstein, and Ely Porat. "Improved space-time tradeoffs for kSUM." arXiv preprint arXiv:1807.03718 (2018).
[14] Alfonsín, JL Ramírez. "On variations of the subset sum problem." Discrete Applied Mathematics 81.1-3 (1998): 1-7.