Project Report
Combinatorial Optimizations

Subset Sum

By
Shrikanth Showri Rajan
Christopher Magana

Introduction

The problem explored in this project is the Subset Sum problem which states that in a finite set A of size s and a positive integer B, provide a result of whether a subset within A exists where the sum of each element in that subset will be equal to B. This is a decision problem and is in NP as the certificate to determine if the sum of the elements in the subset is equal to the integer B can be done in polynomial time.

This report presents the exhaustive algorithm written to solve this decision problem and reports on the instances used to run this problem with benchmarks.

Benchmarking

- Instances:
  The inputs to this algorithm are considered as instances. These instances are generated as a structure which contains the Set of integers, the target sum and the size of the set. An array of 100 instances are passed individually into the algorithm to determine the results for each instance.

  The contents of the instance are determined in random.
  1. The length of the array in each instance is provided and here it begins with a size of 10 elements and increases by 2 for every next instance.
  2. The integers within the set are generated randomly by the rand function from 1 to 99. All instances contain the integers only between 1 and 99.
  3. The target sum is determined by taking a random number from 1 to the total sum of the elements from the array that was generated in the second step.

  These input instances into the benchmark cover all necessary combinations and makes sure that the right ranges are covered.

Algorithm

1. Exhaustive Algorithm:
   The exhaustive algorithm runs over all the possible subsets and gets the sum for each subset and compares with the provided integer. This works by recursively calling each number in the total set and subtracting that number from the total sum. [4]

   Take the last element so the required sum will be the target sum - last element and number of elements = total elements - 1. For the next call the required sum will be the target sum and number of elements = total elements - 1.

Pseudocode:

```
if (target_sum == 0)  return true;
if (n == 0)       return false;
if (arr[n - 1] > target_sum)
    return subset_sum(arr, n - 1, sum);
return subset_sum(arr, n - 1, sum) ||  subset_sum(arr, n - 1, sum - arr[n - 1]);
```

From the pseudocode above we can see that the subset goes through every possible combination present within that set. So by varying the number of elements required to search within the set, the algorithm takes a different starting point in that set and goes from the end of the set to the start. This way all the subsets are covered and there may be many redundant subsets that are checked.

Results

Time taken to solve an instance of given doesn't really give the right representation since the instance contains a set of random integers and the target sum is also varied so some instances might take longer as it needs to check more subsets but sometimes the first subset might contain the target sum in the first try even though its of a bigger size. To truly test out this characteristic, maintaining the sum to be a constant number and checking with different sizes would make sense.

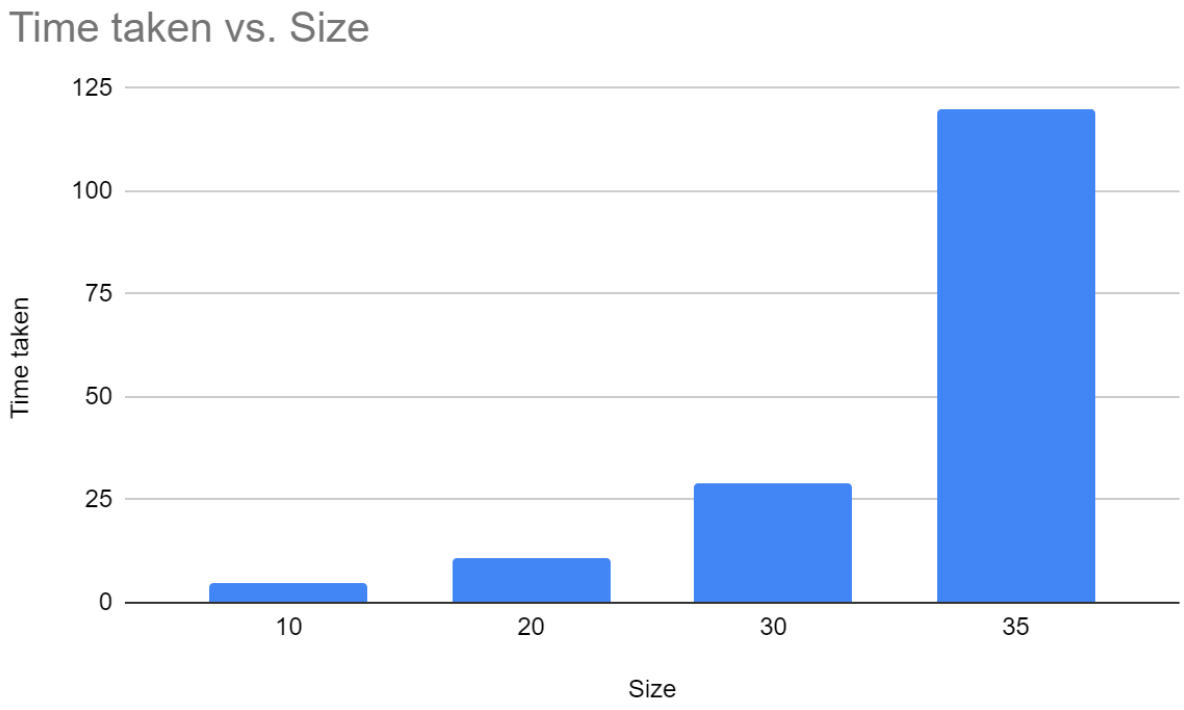

Figure 1. Time taken v Size of instance (Constant Sum)

Based on time limits for each instance the more time provided the higher the chances of the instance being calculated. Three time limits were checked in this test.

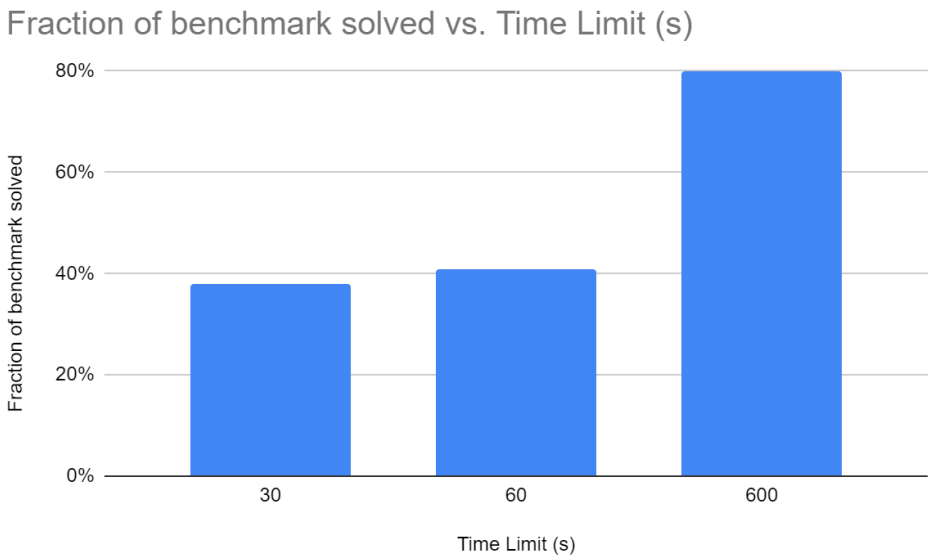

Figure 2. Time Limit vs Fraction of Benchmark Solved *

* In this test for the 600s limit, due to time restriction all the 100 instances couldn't be run. Only 35 in total were run, please consider this number to vary if all 100 instances were solved and this will be updated in the next report.

The largest size instance this algorithm was able to solve within the time limit was of size 82 elements but this was based on random inputs with random target_sum so average runs must be considered but due to time constraints this test was run only twice and provided with the above result.

Conclusion

From the tests performed above it can be inferred that with increasing sizes of the input set and considering a constant sum which the input set will never contain, this makes sure that all subsets are checked, and not a random one the time required to solve the algorithm keeps increasing. This is mainly because of the traversal required to create all subsets and check their sums and also running over many redundant subsets that have already been checked. This is a place for optimization.

Providing a time limit to the instance only determines the amount of that should be spent on solving that instance and it is observed with a higher time limit more instances are solved which makes since higher size sets require more time to traverse through but this is not an average case since these tests were run only twice in total. Running it at least 10 times would provide a better understanding but executing this program and completing it takes a lot of time.

References

[1] Kogure, J. et al. “On the Hardness of Subset Sum Problem from Different Intervals.” *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.* 95-A (2012): 903-908.

[2] Impagliazzo, R., Naor, M. Efficient cryptographic schemes provably as secure as subset sum. *J. Cryptology* 9, 199–216 (1996)

[3] T. E. O'Neil and T. Desell, "Empirical support for the high-density subset sum decision threshold," 2015 IEEE 14th Canadian Workshop on Information Theory (CWIT), 2015, pp. 160-164

[4] Hu G., Pan Y., Zhang F. (2014) Solving Random Subset Sum Problem by *lp*-norm SVP Oracle. In: Krawczyk H. (eds) Public-Key Cryptography – PKC 2014. PKC 2014. Lecture Notes in Computer Science, vol 8383. Springer, Berlin, Heidelberg.