Sabancı University

Fall 2019

CS 301-Algorithms

Project Report

SUBSET-SUM PROBLEM

YAVUZ GÜLEŞEN–KAAN GÜNEY KEKLİKÇİ –SELEN UYGUN–BERK KARAİBRAHİMOĞLU- YUNUS EMRE TAŞCI

PROBLEM DESCRIPTION

The subset sum problem asks us to find a subset B of positive elements which are selected from a given finite set A whose sum adds up to a given positive number *K*, with given inputs of the finite set A, its elements denoted by "*a*" and the positive target integer *K*. In addition, the finite set is constructed with non-negative values.

More formally, given a finite set $A \subset N$ and a target $K \in N$. It asks us that whether there exists a subset $B \subseteq A$ whose elements sum up to *K*. For example, if A = {1, 3, 5, 11, 48, 94, 343, 593, 2124, 3293, 16808}, and *K* =4391, then the subset B = {1, 3, 5, 11, 48, 94, 343, 593, 3293} is a solution.

To define the problem as a language:

SUBSET-SUM = {< A, *K* >: there exists a subset $B \subseteq A$ such that $K = \sum_{b \in B} b$}.

To understand it better, we can observe its modern-day applications such as;

- Consider a vehicle that cannot carry more than K kilos and n different boxes to ship, and i^{th} of weighs x_i kilos. To optimize the problem, we do not want the exceed the limit of transportee while loading as heavy as possible.
- In message verification, a sender wants to send the messages to a receiver but to be sure that the message he is receiving is not from imposter (keeping the message is a secret is not important). Both the sender and the receiver decide on set of;

- a_i (500) and the set of totals T_j(200). Numbers can be known by anyone but only sender knows which subset of a_i (A) correspond to which T_j(K). S sends the message which is a subset of size of {1,…,200}. By this way sender 100 subset of a_i corresponding to the message he wanted to send to sender.

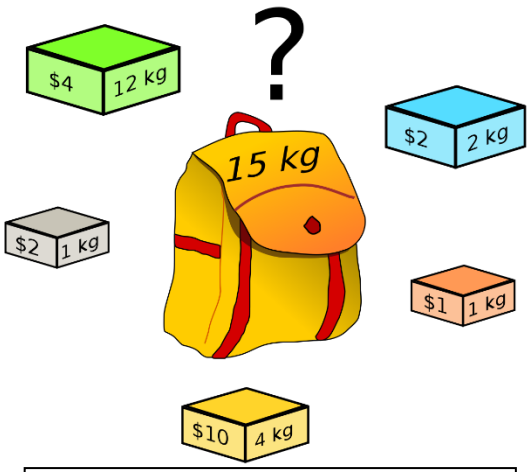

Fig.1. Subset Sum problem can be reduced to 0-1 knapsack problem

PROOF

Theorem: Subset-Sum is an NP-complete problem.

- For a problem to be defined as an NP-complete problem; two specifications are needed:
  a. The problem should belong to NP class.
  b. The problem needs to be a reduction of an existing NP-complete problem; in polynomial time.

Proof

a)

→ In order to prove Subset-Sum is an NP-complete problem, we will take an input of 2 parameters {B, k} where B is our certificate (sample subset) and k is the summation of the members belonging to B. To explain in technical terms; our proof relies on proving whether we could run an algorithm to satisfy for our certificate B such that we have $\{k \in Z^+ \text{ s.t. } k = \Sigma(s), \text{ for } s \in B\}$.

→ According to our claim above; for some s_i in A; checking whether the summation of such s variables in set A may clearly be done in polynomial time, making Subset-Sum a member of the NP-class. Therefore, we move on to the second step where we will perform a reduction from 3-CNF SAT.

b)

→ Let (x_1, x_2, x_3) be a 3-CNF SAT problem with 4 clauses (C_1, C_2, C_3, C_4). We already know that 3-CNF SAT problem is an NP-complete problem. Therefore, if there is a valid reduction from the Satisfiability problem to the Subset-Sum problem, by this way Subset-Sum may be considered as an NP-complete problem. In other words;
3-CNF SAT only has a solution if and only if Subset-Sum has a solution, implying a verified reduction.

→ Before performing the table operations (**Figure 2**); we introduce a function for the clauses C_1, C_2, C_3, C_4.

→ Let there be 3 literals (x_1, x_2, x_3) for 4 predetermined clauses; (C_1, C_2, C_3, C_4).

(*) F (x_1, x_2, x_3) = (x_1 v ¬x_2 v ¬x_3) ∧ (¬ x_1 v ¬x_2 v ¬x_3) ∧ (¬ x_1 v ¬x_2 v x_3) ∧ (x_1 v x_2 v x_3).

→ Now, we will introduce the steps for the reduction with respect to the **(Fig. 2)** provided below. However, first we need an assumption which satisfies the function provided above because the expression must be true in order to proceed.

- **Assumption:** (x_1, x_2, x_3) = (True, False, True) = (1,0,1)

(I) Note that while introducing the table; we are going to utilize this assignment and add the matching 1's to our total and determine the helper variables for each column of the target variable. Please refer to the table (**Fig. 2**) below for full intuitive analysis.

(II) **Definitions:** For any for x_i in a literal set of X = {x_i, x_{i+1}, x_{i+2}, …,x_m for $i \leq m$}.

i) If literal is True, then the literal is represented such that the literal; Y_i is 1, else it is assigned a 0.

ii) Negated literals are represented with a negation symbol (¬ Y_i).

iii) Note that (i) and (ii) refer to the possible values literals can take, not the actual literal values.

iv) Definition (a) and definition (b) are tabulated in the first quarter of the table below. (**1)**

v) Helper variables are denoted by a set of S such that $S = \{s_j, s_{j+1}, s_{j+2}, \ldots, s_n \text{ for } j \leq n\}$. Each S_j is a helper variable in order to exactly add up to the sum of the subset we are looking for.

vi) For each C_j; helper variables are assigned with respect to the result of the matching 1's obtained by our assumption in the first place. In the table below; we will provide two numeric values for each clause; 1 for the positive helper (s_j) and 0 for the negated helper ($\neg s_j$).

(III) Example Tabularization Method

(1) Consider our assignment $(x_1, x_2, x_3) = (1,0,1)$.

(2) Note that the first quarter of the table entries were determined before. *See Definitions, (iii).*

(3) Now, let us fill the initial row of the table for the clause (C_1) with respect to the 3-CNF SAT expression we have created at the beginning (*).

(4) By implications of **(2);** the starting 4 entries in that row are (1,0,0). By placing these truth values; in the function; C_1 evaluates to 1.

(5) The next 5 rows are completed with the same method.

(6) Now, the upper half of the table is completed.

(7) Next, we introduce a target integer for all horizontal entries of the table denoted by a set of $T = \{t_k, t_{k+1}, t_{k+2}, \ldots, t_{j+m} \text{ for } (k \leq j+m)\}$. Each t_k is initially 0.

(8) Let every target value be 4 for the target integer, Subset-Sum.

(9) Lower half of the table is consisting of helper variables and their negations. With respect to our assumption, in this case (1,0,1); now we start matching the entries in C_1 column with (1,0,1). If we obtain a match for the literal, we add 1 to our t_k. In this case; this is t_1.

(10) From now on; we obtain a literal summation but are threatened to fall short of the target summation for each clause.

(11) Since we have only 3 literals maximum summation attainable is 3, therefore one of the helper variables has to be 1 and the other helper variable is assigned such that the

minimum summation attainable is 2, therefore the negated helper variable is assigned 2 to make up for the total of 4 in the worst case.

(12) The remaining of the table is filled according to steps (1-11).

Table Representation of 3-CNF SAT Reduction to Subset-Sum (Fig. 2)

----------	x_1	x_2	x_3	C_1	C_2	C_3	C_4
Y_1	1	0	0	1	0	0	1
¬ Y_1	1	0	0	0	1	1	0
Y_2	0	1	0	0	0	0	1
¬ Y_2	0	1	0	1	1	1	0
Y_3	0	0	1	0	0	1	1
¬ Y_3	0	0	1	1	1	0	0
S_1	0	0	0	1	0	0	0
¬ S_1	0	0	0	2	0	0	0
S_2	0	0	0	0	1	0	0
¬ S_2	0	0	0	0	2	0	0
S_3	0	0	0	0	0	1	0
¬ S_3	0	0	0	0	0	2	0
S_4	0	0	0	0	0	0	1
¬ S_4	0	0	0	0	0	0	2
T	1	1	1	4	4	4	4

(13) All table entries are filled; each row is now a valid candidate to be one of the additives which as a whole; add up to the target integer.

(IV) Conclusion:

i. T = 1114444

ii. S = {1000110,100001,11100,1000,2000,200,10,20,1,2} is subset where each s_i for $i \leq m$ is an additive of summation which evaluates to T.

To be precise, we have taken the rows which are shaped by our assignment on the literals, and the needed helper variables which are needed to create our target variable T.

iii. We have proven that Subset-Sum is at least as hard as 3-CNF SAT problem; hence it's an NP-complete problem.

ALGORITHM DESCRIPTION

There is not an exact algorithm that solves SUBSET-SUM problem in polynomial time, but there are some heuristics such as; greedy subset-sum algorithm which works in O(n logn) with ½ worst-case performance ratio, Trim Algorithm which works in O(n) time, and another greedy algorithm suggested by Silvano Martello and Paolo Toth(1984) with $O(n^2)$.

For this project, we will cover and analyze the performance of the greedy algorithm suggested by Martello and Toth.

Explanation of the algorithm:

1. Given a set, and a number *K*, we sort the elements in the set in the descending order with an efficient sorting algorithm such as Merge Sort.

2. For each element in the sorted list, we take the largest element which is smaller than *K* from the set and decrease *K* by that element.

3. Do this until *K* becomes 0. (Maximum n many times)

4. If *K* becomes 0, then we found a subset in the set such that some elements in that subset adds up to *K*. So, return.

5. If not, maybe this is because the largest possible element in the set should not be in the subset. So, refresh *K* (make it equal to original *K* again), and do the step 2 again, but now firstly take the next largest element that is smaller than *K* from the set.

6. Do step 5 until *K* becomes 0 (Step 4 is satisfied, so subset is found).

ALGORITHM ANALYSIS

First, we sort our elements in a decreasing order. The algorithm does not guarantee that it will find a subset such that its elements add up to a given target number *K*, because the idea assumes that the maximum element in our main set should be included in the subset we aim to return as a sum of our target value *K*.

However, the combination we need to get the target *K* does not always include the largest element in our main set. This results in losing the maximum element which is equal or lower than the target value *K*. So, we skip to the next element which is the maximum in our main set. And at the i^{th} iteration, the needed element will stay at the left-side of the i^{th} largest element.

Consider the case where the given main set = {9, 8, 6, 5, 3} and *K* = 18. Firstly 9 will be taken and *K* becomes 18-9 = 9, then we need 9-8=1 and 1 is not found in our set. So, the second largest element is considered which is 8. *K*=18-8=10 now we should look for numbers to be summed to 10. 6 is taken and *K* becomes 4. then taking 3 makes the *K*=1 but there is no 1. Consider the next largest element 6, 18-6=*K* becomes 12, then taking 5 makes *K*=7(12-5=7), then taking 3 makes *K*=1, but we do not have 1 in the set. However, 6+3+9 makes 18. But 9 remains at the left side of 6, so it is not considered when 6 is the largest number in 3rd considered subset.

Time Complexity Analysis:

```
import random
def subset_greedy(set, k):
    set.sort(reverse=True)
    for i in range(0, len(set)):
        #subset =[]
        k_ = k
        for j in range(i, len(set)):
            if set[j] <= k_:
                k_ = k_ - set[j]
        #         subset.append(set[j])
        if k_ == 0:
            print("There's a subset whose elements sum up to", k)
         #   print("Subset:", subset)
            return True
    print("No subset found such that its elements sum up to", k)
    return False
```

Sorting method in Python3: O(n logn)

O(n)

$O(n^2)$

- ⇨ The total running time complexity = O(n logn) + $O(n^2)$ = $O(n^2)$ in the worst case.
- ⇨ If first subset found is wanted to be seen, the space complexity is O(n) coming from array named “subset” which may store maximum n elements.

EXPERIMENTAL ANALYSIS

Our inputs are; the set and *K*.

Experiment 1: We generated several sets with 10, 25, 50, 100, 250 elements that randomly assigned in the (0, 100], (0, 500], (0, 1000], (0, 5000], (0, 10000].

And K values are given as constant from 100 to 5000 as seen below.

Setsize / K	**100**	**500**	**1000**	**5000**
10	0.16	0.02	0.02	0.01
25	0.56	0.19	0.08	0.01
50	0.9	0.43	0.17	0.08
100	0.99	0.83	0.63	0.14
250	1.0	0.99	0.98	0.203

Fig.3 Success Rates w.r.t Iteration count = 100

We see that as size increases and *K* decreases with the given conditions, the success rate increases. Because as size increases, there will be more elements in the set and low *K* value is easier to reach.

Experiment 2: We took K as a derived formula that we have seen from researches and text which is approx.. (n/4) * range where n is set size and range is 1,10000. (range is chosen by trying out)

Range=(1,10000)

Setsize / Iteration Count	**50**	**100**	**250**	**500**	**750**
25	0.019	0.009	0.027	0.033	0.042
50	0.098	0.158	0.147	0.160	0.152
75	0.215	0.267	0.290	0.305	0.295
100	0.450	0.445	0.498	0.512	0.504
150	0.705	0.772	0.768	0.764	0.761

Fig.4 Success Rates w.r.t SetSizes of range[25,150] and IterationCount of range [50,750]

This table covers lots of possibilities where set size ≤ 150. And as the set size increases, the values become more realistic and diverted which, in result, have given the best results. We saw that in a normal case we have a correctness ratio in ½ and ¾ range.

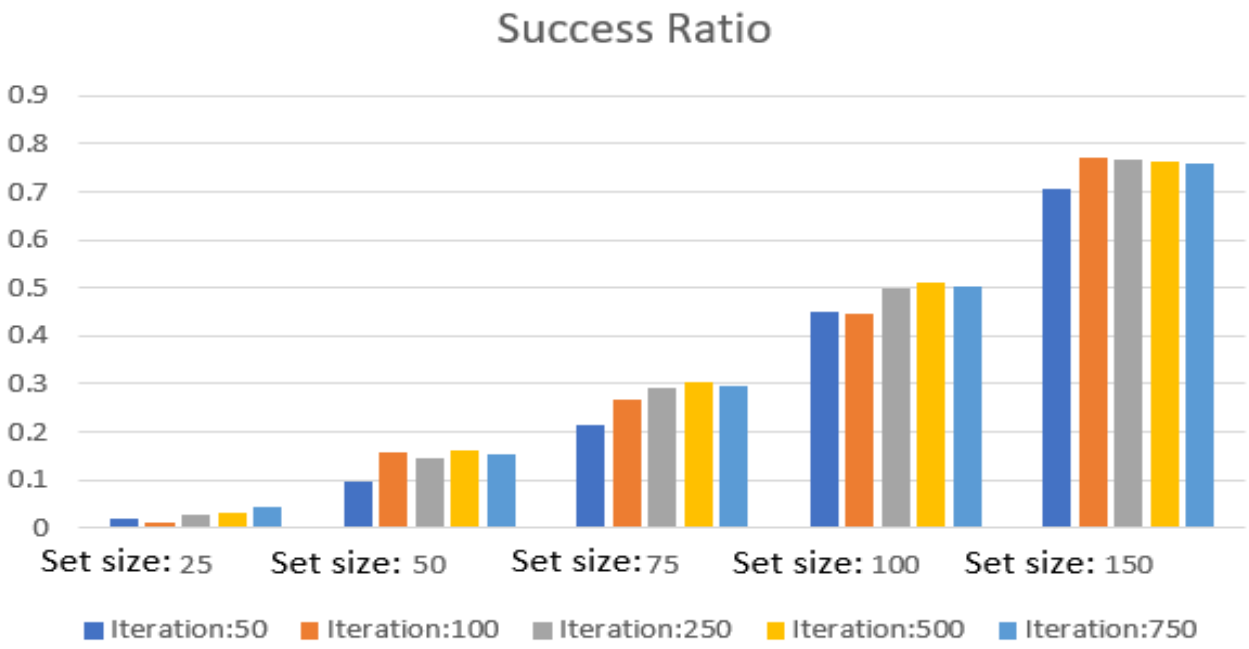

Fig.5 - Visualization of Fig.5, Success Ratio w.r.t Fig.4

Set Sizes	Success Rate
25	0.042
50	0.152
75	0.295
100	0.504
150	0.761

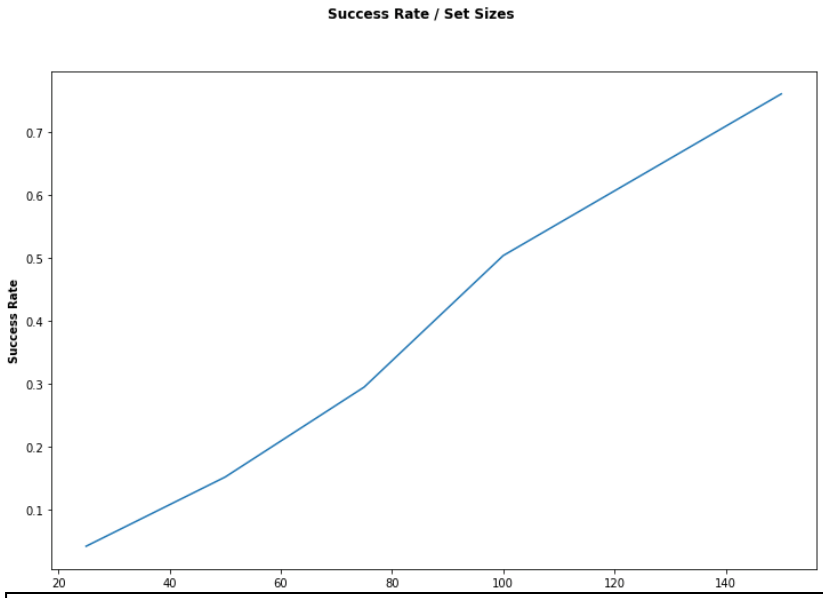

Fig.6 - Success Rate w.r.t to Different SetSizes under IterationCount = 750

Referring to *Fig.5; under controlled variable IterationCount = 750;* we conclude that as the set size increases, success rate of the algorithm increases proportionally.

Experiment 3: Same as experiment 2 took K as a derived formula that we have seen from researches and text which is approx.. (n/4) * range where n is set size but now we took range 1,1000 to show that with lower numbers, higher set size has very high accuracy.

Range=(1,1000)

Setsize / Iteration Count	**50**	**100**	**250**	**500**	**750**
25	0.21	0.227	0.254	0.286	0.281
50	0.72	0.783	0.769	0.774	0.772
75	0.893	0.928	0.913	0.917	0.909
100	0.957	0.952	0.947	0.948	0.944
150	1	1	0.997	0.997	0.996

Running Time Experimental Measurement

In order for us to make statistical inferences about our algorithm; we ran different Python codes to calculate for various statistical measurements such as the *standard error, standard deviation, sample mean, upper-mean, lower-mean, and confidence intervals* of 90%, 95% .

```
from time import process_time # for timing

import statistics # for standard deviation

tval90 = 1.645 # fixed

tval95 = 1.96 # fixed

alltimes = [] # storage for processing times

def stats(size,times):

  set=[0]*size # construct the set with intended SetSize

  k=(size/4)*float(10000)  # capacity assignment

  for x in range(1,times):

    for i in range(0,size):

      set[i]=random.randint(0,10000)

    # RunningTime calculation start...

    starttime = process_time()

    mtgs(set,k)
```

```
    endtime = process_time()
    # RunningTime calculation end...
    alltimes.append(endtime-starttime)
# Here, we provide an example call for SetSize = 25, IterationCount = 100
totaltime = sum(alltimes)
mean25 = totaltime/100 # example sample mean calculation
deviation25 = statistics.stdev(alltimes)  # example deviation calculation
standarterror25 = deviation25 /10 # example standard error calculation
upper95_25 = mean25+standarterror25 * tval95 # confidence level 95% upper bound
lower95_25 = mean25-standarterror25 * tval95 # confidence level 95% lower bound
# other confidence levels (90%,99% ) are calculated with the same approach under different sample means and standard errors
alltimes = []
# Other calls are made in the same method for different SetSizes and IterationCounts
```

We have generated 90% and 95% confidence levels. Higher percentage of a confidence interval implies a larger accuracy, because a larger percentage also implies a wider interval range.

100 RUN PER INPUT SIZE

SubsetSize	***Mean Time***	***Std. Error***	***Std. Deviation***	***%90-CL***	***%95-CL***
25	0.000096459	0.0000554501	0.00005545	0.0000873379 - 0.0001055810	0.00008559126 - 0.0001073277
50	0.00022832	0.000011796529	0.000117965	0.000219208026 - 0.0002374511	0.000205208 - 0.0002514507
75	0.0004524648	0.000025360373	0.000253603739	0.000402758547 - 000502171212	0.0004107470 - 0.000494182
100	0.0005192592	0.000040520840	0.000405208402	0.000452602487 - 0.00058591605	0.000439838421 - 0.0005986801
150	0.0007042182	0.0000817379015	0.000817379015	0.000607741858 - 0.0008803420	0.000581641844 - 0.00090644201

Fig.7; MeanTime(s), standard error & deviation, Confidence Levels = 90% & 95% tabulated for SubsetSizes=[25,150], IterationCount = 100.

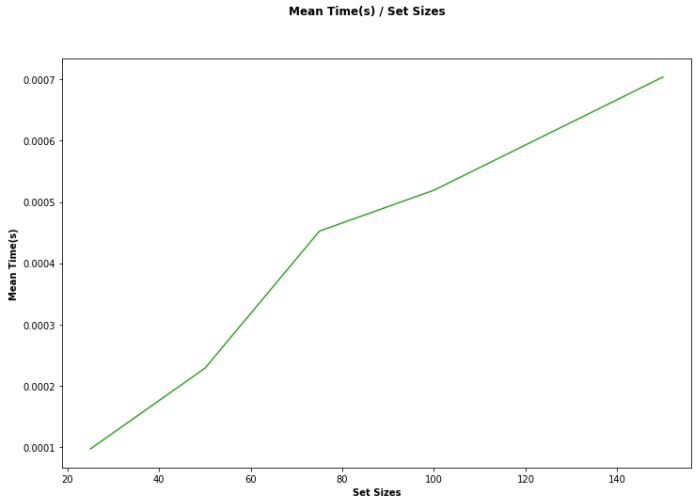

Fig.8- MeanTime(s) w.r.t to SetSizes of range[25,150], IterationCount = 100.

250 RUN PER INPUT SIZE

SubsetSize	***Mean Time***	***Std. Error***	***Std. Deviation***	***%90-CL***	***%95-CL***
25	0.000073492	0.000002403597	0.0000380042099	0.000226699990 - 0.0002346078	0.000225942857418 50664 - 0.0002353
50	0.000215985	0.0000047652372	0.0000753450167	0.000187711004 - 0.00020338863	0.000186209954965 687 - 0.00020488
75	0.0003866860	0.000012314764	0.0001947135156	0.000351311489 - 0.00039182706	0.000347432338531 9860 - 0.00039570
100	0.0005572228	0.0000243553323	0.0003850916175	0.000490697802 - 0.00057082684	0.000483025872527 27 - 0.0005784987
150	0.0006904984	0.0000459001234	0.0007257446742	0.000614992756 - 0.00076600416	0.000600534218124 - 0.0007804627018

Fig.9 - MeanTime(s), standard error & deviation, Confidence Levels = 90% & 95% tabulated for SubsetSizes=[25,150], IterationCount = 250.

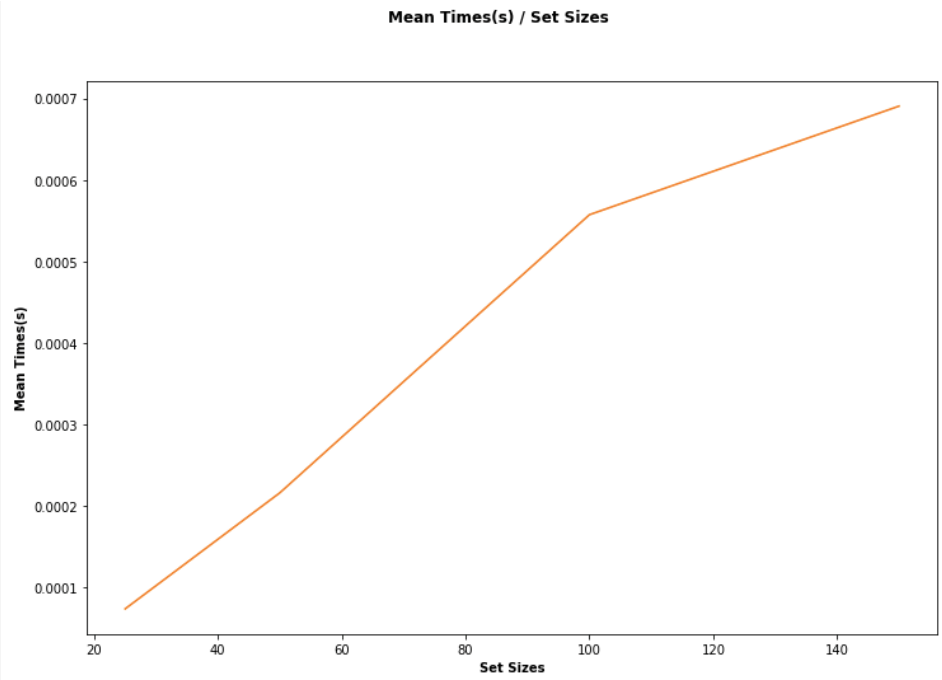

Fig.10- MeanTime(s) w.r.t to SetSizes of range[25,150], IterationCount = 250.

500 RUN PER INPUT SIZE

SubsetSize	***Mean Time***	***Std. Error***	***Std. Deviation***	***%90-CL***	***%95-CL***
25	0.0000736341	0.000016287791	0.0000036420607	0.00006806295 - 0.000077000254	0.00006720725931- 0.00007785595268
50	0.0002058067	0.000004452807	0.0000995677916	0.000198481848 - 0.00021313158	0.000197079214257- 0.0002145342177
75	0.0003797868	0.0000090407984	0.0001973520047	0.000364914738 - 0.00039465896	0.000362066886972 - 0.0003975068170
100	0.000540782	0.0000158653072	0.0003547590551	0.000514683637 - 0.00056688049	0.00050968606577 - 0.0005718780702
150	0.0006776647	0.0000326471635	0.0007300127703	0.000623960121 - 0.00073136929	0.000613676265384- 0.0007416531466

Fig.11 - MeanTime(s), standard error & deviation, Confidence Levels = 90% & 95% tabulated for SubsetSizes=[25,150], IterationCount = 500.

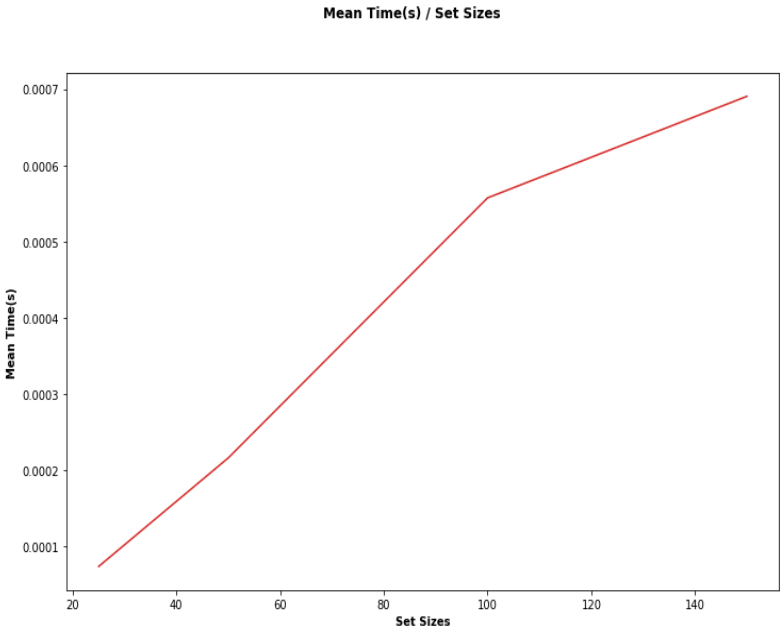

Fig.12 - MeanTime(s) w.r.t to SetSizes of range[25,150], IterationCount = 500.

1000 RUN PER INPUT SIZE

SubsetSize	***Mean Time***	***Std. Error***	***Std. Deviation***	***%90-CL***	***%95-CL***
25	0.000071534	0.000001370685	0.0000433448789	0.000064586574 - 0.00007378881	0.00006439171470 - 0.00006681663729
50	0.0002218246	0.0000030698556	0.0000970773605	0.000216774728 - 0.00022687455	0.00021580772385 - 0.0002278415581
75	0.0003517221	0.0000062660954	0.0001981513377	0.000341414447 - 0.00036202990	0.00033944062784 - 0.00036400372215
100	0.0005016412	0.0000109952049	0.0003476989087	0.000483554105 - 0.00051972833	0.00048009061636 - 0.0005231918196
150	0.0006984795	0.0000221197712	0.0006994885842	0.000662092566 - 0.00073486661	0.00065512483838 - 0.0007418343416

Fig.13 - MeanTime(s), standard error & deviation, Confidence Levels = 90% & 95% tabulated for SubsetSizes=[25,150], IterationCount = 1000.

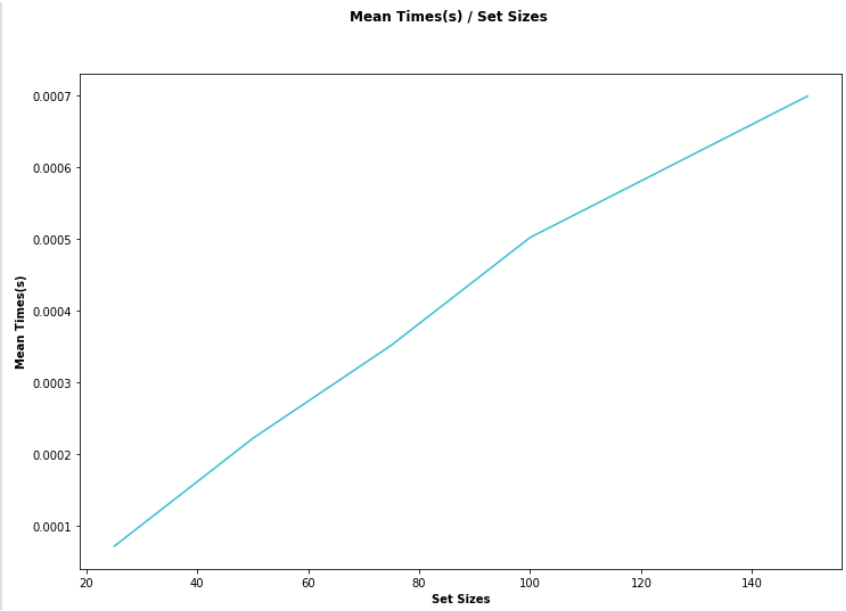

Fig.14 - MeanTime(s) w.r.t to SetSizes of range[25,150], IterationCount = 1000

TESTING

We tested the algorithm with various inputs (different set size, different range for elements, different K) which are selected at random, but with some specified upper limit for random numbers. Therefore, with black box technique, we test whether:

- An exact algorithm finds a solution and also our algorithm also finds a solution. In this case, our algorithm passes the test.
- The exact algorithm finds a solution and our algorithm does not find any solution, then failed.
- The exact algorithm and our algorithm both does not find any solution, then our algorithm passes the test.

```
import random

def greedy_subsetsum(set, k):

    set.sort(reverse=True)
    for i in range(0, len(set)):
        subset = []
        k_ = k
        for j in range(i, len(set)):
            if set[j] <= k_:
                k_ = k_ - set[j]
                subset.append(set[j])
```

```
        if k_ == 0:
            print("There's a subset whose elements sum up to", k)
            print("Subset: ", subset)
            return 1
    print("No valid subset found.")
    return 0


def exact_algorithm(set, n, sum):
    # Base Cases
    if (sum == 0):
        return True
    if (n == 0 and sum != 0):
        return False
    if (set[n - 1] > sum):
        return exact_algorithm(set, n - 1, sum)
    return exact_algorithm(set, n - 1, sum) or exact_algorithm(set, n - 1,
 sum - set[n - 1])




def test(setsize, elementsize, ksize):
    k = random.randint(1, ksize)
    set = [0] * random.randint(1, setsize)
    for i in range(0, len(set)):
        set[i] = random.randint(1, elementsize)
    print("set:", set)
    print("k:", k)

    test = greedy_subsetsum(set, k)
    truth = exact_algorithm(set, len(set), k)

    if test == 1:
        if truth == True:
            print("Truth: There exists a valid subset.")
            print("Test Passed.")
        else:
            print("Truth: No such subset.")
            print("Test Failed.")
    else:
        if truth == True:
            print("Truth: There exists a valid subset.")
            print("Test Failed")
        else:
            print("Truth: No such subset.")
            print("Test Passed")
```

Some Test Results:

Test 1: `test(15, 100, 500)`

```
set: [97, 8, 36, 67, 3, 42, 29, 9, 96, 18, 74, 15]
k: 421
There's a subset whose elements sum up to 421
Subset:  [97, 96, 74, 67, 42, 36, 9]
Truth: There exists a valid subset.
```

Test Passed.

Test 2: `test(15, 100, 1000)`

```
set: [25, 12, 29, 43, 46, 29]
k: 685
No valid subset found.
Truth: No such subset.
```

Test Passed

Test 3: `test (15, 100, 1000)`

```
set: [46, 18, 95, 92, 42, 94]
k: 88
There's a subset whose elements sum up to 88
Subset:  [46, 42]
Truth: There exists a valid subset.
```

Test Passed.

Test 4: `test (25, 100, 500)`

```
set: [13, 82, 98, 91, 93, 32, 68, 86, 3, 65, 89, 92, 24, 94,
38, 48, 8, 1, 65, 29, 100, 83, 15, 88, 24]
k: 410
There's a subset whose elements sum up to 410
Subset:  [100, 98, 94, 93, 24, 1]
Truth: There exists a valid subset.
```

Test Passed.

Test 5: `test (25, 100, 500)`

`set: [87, 86, 6, 7, 36, 40, 3, 88, 44, 20, 59, 86, 1, 31, 28]`

`k: 134`

`There's a subset whose elements sum up to 134`

`Subset:  [87, 44, 3]`

`Truth: There exists a valid subset.`

Test Passed.

Test 6: `test(25, 100, 1250)`

`set: [51, 76, 99, 44, 60, 70, 29, 11, 38, 22, 72, 50, 54, 49, 25, 70]`

`k: 516.`

`No valid subset found.`

`Truth: There exists a valid subset.`

Test Failed

Test 7: `test(25, 10000, 62500)`

`set: [7436, 2251, 6495, 1753, 1422, 6304, 6432, 5330, 6124, 2360, 1282, 3640, 2803, 8009, 4948, 5731, 7898, 7118, 154, 1483, 6787, 9759, 6270]`

`k: 53185`

`No valid subset found.`

`Truth: There exists a valid subset.`

Test Failed

Test 8: `test(50, 10000, 125000)`

`set: [839, 5312, 2369, 854, 3348, 9304, 8907, 3830, 2692, 6472, 7385, 4175, 5266, 4399, 7062, 9531, 9293, 3606, 2304, 5742, 3579, 4986, 4284, 9791, 3522, 2536, 7220, 2621, 2386, 1003, 8933, 7778, 9641, 8522, 5443, 2751, 1384, 2599, 2653, 8316, 6071, 3294, 6703, 9309, 7664, 679, 5595]`

`k: 25273`

`There's a subset whose elements sum up to 25273`

`Subset:  [7385, 7220, 7062, 3606]`

`Truth: There exists a valid subset.`

Test Passed.

CONCLUSION

To summarize; we have observed that subset sum problem is an NP-complete problem. We have demonstrated that Subset-Sum problem is a reduction from 3-CNF SAT and the problem itself is NP-hard. It needs to be indicated that there is not an exact algorithm that solves the problem in polynomial time after some extent. There exist approximate approaches just like we have used. Greedy approach is one of them that we have worked on. We have analyzed and examined this algorithm in our report and demonstrated that the greedy approach does not always come up with the solution.

Despite the algorithm that we ran while experimenting occasionally fails, we have obtained an approximate of between %60-%75 success rate while running the greedy algorithm on different iteration levels and set sizes. On average, accuracy rates increased with proportion to wider set-sizes. Based on that inference; we generated different sets with different increased set sizes to display the increasing accuracy rates. In that stage of the experimentation; we also inferred increased set sizes also result in increased running mean times. An improvement in the algorithm would be to find an even more efficient k-value (capacity) to both improve the running times and accuracy ratings. The value for maximum possible range for an element in the set (element size) is another possible parameter that could be improved. However; based on our findings; in general, multiplication of set size and the element range must at least be equal to K to obtain a higher accuracy.

Lastly, when analyzing for time complexity; we obtained that for all running time graphs; the graph behavior is quadratic. Even though the graph behavior seems like it’s concave instead of convex at subset size of 150; this is due to the increased possibility for the algorithm to return a correct output in longer subset sizes. This probability increase results in a relatively faster performance at subset size of 150 compared to previous sizes such as 50 and 75.The algorithm running time was expected to be more at 150, however, since the accuracy is very high at that level, the algorithm ran faster than expected which is understandable. In the end, the inference

about the algorithm's performance still stands true, previous subset sizes show strong implications of quadratic behavior.

REFERENCES

1. Martello, S., & Toth, P. (2006). *Knapsack problems: algorithms and computer implementations*. Ann Arbor, MI: UMI Books on Demand.

2.Tardos, E. (2015). *Reduction from 3 Sat to Subset Sum.*