
VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

# Data Structure and Algorithm (CSE2003)

## Sub Set Sum Problem
## REVIEW-3

By;

- V.V.SAI DILEEP - 18BCE0419
- S.SUSANTH - 18BCE0552

Submitted to;

- Faculty Name: Shalini.L Mam
- Slot: B2
- Class Number: SJT315

# Contents

# ABSTRACT :

In this presentation, Subset Sum problem (Non Polynomial Complete problem) and analysis of its two solutions is discussed. These solutions are the Dynamic Solution algorithm and the Back Tracking algorithm. Dynamic Solution is a sound and complete algorithm that can be used to determine satisfiability and un satisfiability with certainty. Backtracking also finds the satisfiability but will have a time complexity of exponential degree.

In addition, analogy between these two algorithms and two other algorithms namely PL-Resolution and Walk-Sat algorithms for 3-CNF SAT problem, which is also NPC problem is discussed.

The Subset Sum problem is an non deterministic problem where we need to find a resultant number from a set of numbers to perform the addition of subset of a set. Non-polynomial problems comprise of the set of decision problems where answer to any instance of the problem is true then it can be easily proved why the solution is true. Non-Deterministic Polynomial problems are the collection of problems where if the solution is true, then it provides the complexity of the problem in Polynomial.



## Keywords :

- **SUBSETSUM**: Subset sum states that let there be a set A of n positive Integers and a sum s. Then the sub set of A whose elements add up to the given sum is a sub set sum.

- **PL-RESOLUTION**: Propositional resolution is a powerful rule of inference for propositional logic. Using propositional resolution it is possible to build a theorem prover that is sound and complete for all of propositional logic.

# INTRODUCTION TO THE PROBLEM:

The Subset-Sum Problem (SSP) is defined as follows: given a set of positive integers S, e.g., {s1, s2, s3, s4, s5, s6}, and a positive integer C. This problem is to find one/all subsets of S that sum as close as possible to, but do not exceed, C [1, 2]. For an example, consider the set S={1, 2, 3, 4, 5} and let the target sum C be 10. The total number of subsets of S in this case is 25. Some of the valid solutions to this problem are the sets {1, 2, 3, 4}, {1, 4, 5}, and {2, 3, 5}.

In general, we notice that the total number of subsets taken from a set of n elements is 2n [7, 8]. An algorithm that tests all of these possible solution subsets needs an exponential time. Let the number of inputs, that is the size of set S, be n. Using a computer that can generate and test one subset in one microsecond we need 0.001 sec to solve an SSP problem of input size 10. However, as this number of inputs grows to 100,we need 4.1016 years on that same machine! This was the main motive to work on this problem.

Approximation Algorithms try to attack NP-complete problems [1, 5, 6]. Since it's unlikely that there can be efficient algorithms that solve such problems, one settles for non-optimal solutions in order to have approximate solution be found in a polynomial time [14, 16]. Unlike heuristics, which just usually find good solutions reasonably fast, one may need provable solution quality and provable run time bounds both achieved together.

The current paper introduces a new approach that promises finding some of the possible solutions to the SSP problems based on a set of current user's constraints. These constraints mainly decide upon the maximum number of subsets required. Based on the provided constraints, and using a polynomial number of iterations the proposed algorithm can successfully produce some of approximated and valid solutions to the problem. The proposed algorithm can successfully produce all possible solutions.

# RELATED WORK :

The Subset-Sum Problem (SSP) is one the most fundamental NP-complete problems, and perhaps the simplest of its kind. Given a set of n data items with positive weights and a capacity c, the decision version of SSP asks whether there exists a subset whose corresponding total weight is exactly the capacity c; the maximization version of SSP is to find a subset such that the corresponding total weight is maximized without exceeding the capacity c. The Subset-Sum Problem has many applications; for example, a decision version of SSP with unique solutions represents a secret message in a SSP-based cryptosystem.

It also appears in more complicated combinatorial problems, scheduling problems, 0-1 integer programs , and bin packing algorithms. The Subset-Sum Problem is often thought of as a special case of the Knapsack Problem, where the weight of a data item is proportional to its size. Therefore, algorithms for the Knapsack Problem can be applied for SSP automatically. However, as simple as it is, the Subset-Sum Problem has a better structure and hence sometimes admits better algorithms.

# MOTIVATION :

In computational complexity theory, problems within the NP-complete class have no known algorithms that run in polynomial time. The study of NP-completeness is significant, as computer science problems lurk in many guises across a variety of disciplines, from chemical informatics to networking. As such, identifying a problem as NP-complete will conserve both time and effort for the computer scientist, who can avoid a fruitless pursuit for an efficient algorithm by knowing beforehand that there are currently none in existence. Of course, NP-complete problems can still be solved, but either the input data must be restricted to reasonably small sizes to accommodate super polynomial time algorithms or accuracy must be compromised in implementing faster approximation algorithms, neither of which are amenable condition.

Although most algorithms for NP-complete problems are just short of a brute-force search, the immense computational power of a quantum computer may give impetus to more efficient solutions.A quantum search algorithm—namely Grover's algorithm—provides a framework for finding a solution in a finite search space defined in terms of the problem, running faster than a classical algorithm but admittedly still exponential. As such, this paper will demonstrate the utility of quantum search in solving the NP-complete class by applying Grover's algorithm to a specific instance of NP-completeness—the Subset Sum Problem. As it shall be soon revealed, a classical algorithm can do no better than a complete search while a quantum search runs drastically faster by exploiting the principles of superposition and quantum parallelism on an unstructured database.

# PROPOSED METHODOLOGY :

## Walksat Algorithm :

In computer science, GSAT and WalkSAT are local search algorithms to solve Boolean Satisfiability problems.

Both algorithms work on formulae in Boolean logic that are in, or have been converted into Conjunctive normal form. They start by assigning a random value to each variable in the formula. If the assignment satisfies all clauses, the algorithm terminates, returning the assignment. Otherwise, a variable flipped and the above is then repeated until all the clauses are satisfied. WalkSAT and GSAT differ in the methods used to select which variable to flip.

## Recursive Approach :

For every element in the array has two options, either we will include that element in subset or we don't include it.

- So if we take example as int[] A = {3,2,7,1},S=6
- If we consider an another int array with the same size as A.
- If we include the element in sub set we will put 1in that particular index else put0.
- so we need to make every possible subsets and check if any of the subset makes the sum as S.
- If we think carefully this problem is quite similar to "Generate All Strings of n bits "

- See the code for better explanation.
- Time Complexity: O(2n).

## Dynamic programming approach:

- Base Cases:
- If no elements in these t then we can't make any subset exceptfor0.
- If sum needed is 0 then by returning the empty sub set we can make the sub set with sum0.
- Given–Set = arrA[], Size = n, sum=S
- Now for every element in the set we have 2 options, either we include it or exclude it.
- for any i element
- If include it => S = S-arrA[i], n=n-1

If exclude it=> S, n=n-1.

## Sub Set Sum Problem:

The problem is a classic problem it states that given a finite set S of positive integers and a integer target T>0, we check whether there exists a subset S", sum of whose elements is T. A new approach to address Subset Sum Problem.

In computer science, the subset sum problem is an important decision problem in complexity theory and cryptography. There are several equivalent formulations of the problem. One of them is: given a set (or multiset) of integers, is there a non-empty subset whose sum is zero? For example, given the set the answer is yes because the subset sums to zero. The problem is NP-complete, meaning roughly that while it is easy to confirm whether a proposed solution is valid, it may inherently be prohibitively difficult to determine in the first place whether any solution exists.

The problem can be equivalently formulated as: given the integers or natural numbers does any subset of them sum to precisely Subset sum can also be thought of as a special case of the knapsack problem. One interesting special case of subset sum is the partition problem, in which W is half of the sum of all elements in the set.

# CODING :

## ALGORITHM 1: DYNAMIC PROGRAMMING :

## CODE:

```
//Code implementation technique 1 using memorization and dynamic programming and recursion.
#include <bits/stdc++.h>
using namespace std;
// dp[i][j] is going to store true if sum j is
// possible with array elements from 0 to i.
bool dp[100][100];
void display(const vector<int>& v)
{
    for (int i = 0; i < v.size(); ++i)
        printf("%d ", v[i]);
    printf("\n");
}
// A recursive function to print all subsets with the
// help of dp[][]. Vector p[] stores current subset.
void printSubsetsRec(int arr[], int i, int sum, vector<int> &p)
{

  if (i == 0 && sum != 0 && dp[0][sum])
   {
      p.push_back(arr[i]);
      display(p);
      return;
   }
    // If sum becomes 0 if(sum == 0)
     if (i == 0 && sum == 0)
    {
        display(p);
```

```
        return;
    }
    if(i <= 0 || sum < 0) return;
    // If given sum can be achieved after ignoring
    // current element.
    if (dp[i-1][sum])
    {
        // Create a new vector to store path
        vector<int> b = p;
        printSubsetsRec(arr, i - 1, sum, p);
    }
    // If given sum can be achieved after considering // current element.
    if (sum >= arr[i - 1] && dp[i - 1][sum - arr[i - 1]])
    {
        p.push_back(arr[i - 1]);
        printSubsetsRec(arr, i - 1, sum - arr[i - 1], p);
        p.pop_back();
    }
}
// Prints all subsets of arr[0..n-1] with sum 0.
void printAllSubsets(int arr[], int n, int sum)
{
    if (n == 0 || sum < 0)
        return; // If sum is 0, then answer is true
    for (int i = 0; i <= n; i++) dp[i][0] = true;
    // If sum is not 0 and set is empty, then answer is false
    for (int i = 1; i <= sum; i++) dp[0][i] = false;
    // Fill the subset table in botton up manner
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= sum; j++)
        {
```

```
            if (j < arr[i - 1])
                dp[i][j] = dp[i - 1][j];
            if (j >= arr[i - 1])
                dp[i][j] = dp[i - 1][j] || dp[i - 1][j - arr[i - 1]];
        }
    }
    for (int i = 0; i <= n; i++)
    {
        for (int j = 0; j <= sum; j++)
        {
            printf("%4d", dp[i][j]);
        }
        if (dp[n][sum] == false)
        {
            printf("There are no subsets with sum %d\n", sum);
            return;
        }
        // Now recursively traverse dp[][] to find all
        // paths from dp[n-1][sum]
        vector<int> p;
        printSubsetsRec(arr, n, sum, p);
    }
}
int main()
{
    int n,sum;
    printf("Enter the number of terms in the set:");
    scanf("%d", &n);
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        printf("Enter the element:");
```

```
        scanf("%d", &arr[i]);
    }
    printf("Enter the value of sum:");
    scanf("%d", &sum);
    printAllSubsets(arr, n, sum);
    return 0;
}
```

## OUTPUT:

```
Enter the number of terms in the set:5
Enter the element:1
Enter the element:5
Enter the element:2
Enter the element:6
Enter the element:8
Enter the value of sum:15
   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   1   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   1   1   0   0   0   1   1   0   0   0   0   0   0   0   0   0
   1   1   1   1   0   1   1   1   1   0   0   0   0   0   0   0
   1   1   1   1   0   1   1   1   1   1   0   1   1   1   1   0
   1   1   1   1   0   1   1   1   1   1   1   1   1   1   1   1
8 2 5
8 6 1

--------------------------------
Process exited after 16.72 seconds with return value 0
Press any key to continue . . .
```

## ALGORITHM 2: BACK TRACKING:

## CODE:

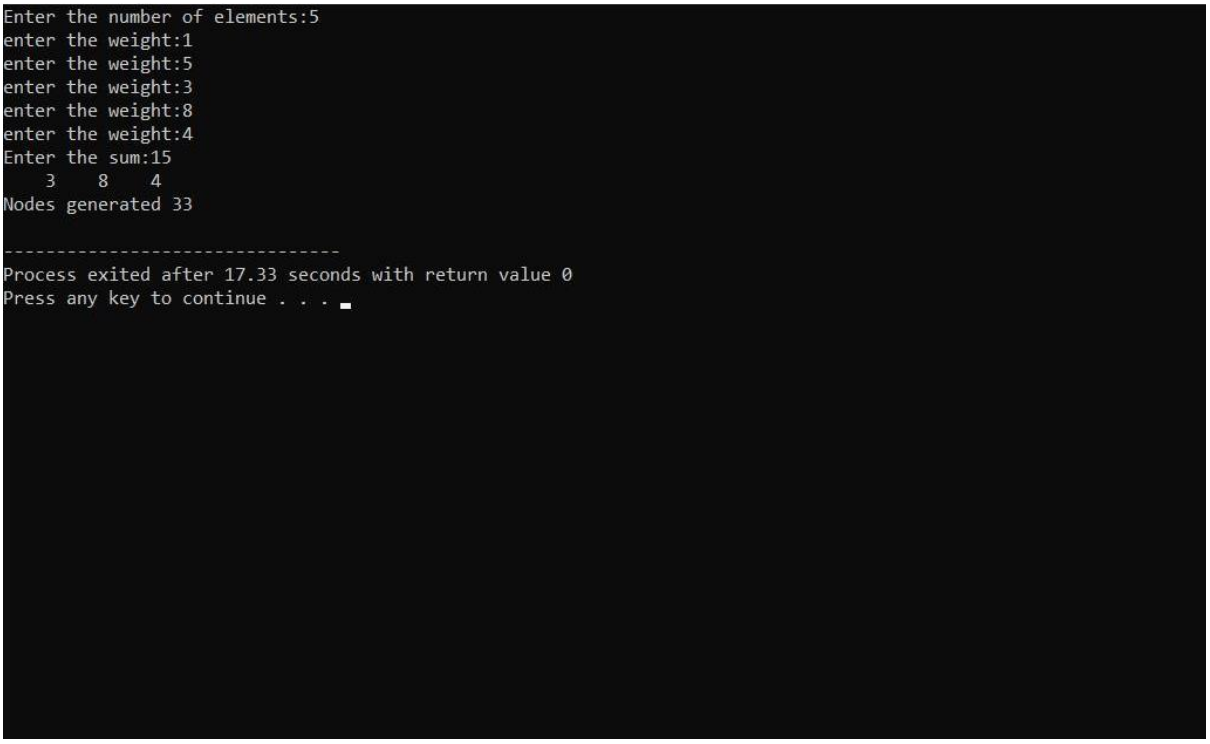
```
#include <stdio.h>
#include <stdlib.h>
#define ARRAYSIZE(a) (sizeof(a))/(sizeof(a[0]))
```

```
static int total_nodes;
// prints subset found
void printSubset(int A[], int size)
{
   for (int i = 0; i < size; i++)
   {
      printf("%*d", 5, A[i]);
   }
   printf("\n");
}
// inputs
// s - set vector
// t         - tuplet vector
// s_size  - set size
// t_size  - tuplet size so far
// sum     - sum so far
// ite- nodes count
// target_sum - sum to be found
void subset_sum(int s[], int t[], int s_size, int t_size, int sum, int ite, int const target_sum)
{
   total_nodes++;
   if (target_sum == sum)
   {
      // We found subset
      printSubset(t, t_size);
      // Exclude previously added item and consider next candidate
      subset_sum(s, t, s_size, t_size-1, sum - s[ite], ite + 1, target_sum); return;
   }
   else
   {
      // generate nodes along the breadth
      for( int i = ite; i < s_size; i++ )
```

```
    {
        t[t_size] = s[i];
        // consider next level node (along depth)
        subset_sum(s, t, s_size, t_size + 1, sum + s[i], i + 1, target_sum);
    }
}
}
// Wrapper to print subsets that sum to target_sum
// input is weights vector and target_sum
void generateSubsets(int s[], int size, int target_sum)
{
    int *tuplet_vector = (int *)malloc(size * sizeof(int));
    subset_sum(s, tuplet_vector, size, 0, 0, 0, target_sum);
    free(tuplet_vector);
}
int main()
{
    int n;
    printf("Enter the number of elements:");
    scanf("%d", &n);
    int weights[n];
    for (int i = 0; i < n; i++)
    {
        printf("enter the weight:");
        scanf("%d", &weights[i]);
    }
    int size = ARRAYSIZE(weights);
    int sum;
    printf("Enter the sum:");
    scanf("%d", &sum);
    generateSubsets(weights, size, sum);
    printf("Nodes generated %d\n", total_nodes);
```

```
    return 0;
}
```

## OUTPUT :




## CONCLUSION :

The proposed algorithm presented above is a good approximation algorithm for sub set sum problem. It finds all solutions for a given input or based on a maximum number of iterations which prevents it from going very long search paths.the dynamic programming is a good approximation for sub set problem.Because the time complexity of this algorithm is less when compared to the backtracking.

# FUTURE WORK:

## COMPARISION :

The time complexity of the dynamic programming algorithms O(sum*n) where n is the input size and sum is the input that we have entered to check whether any combination of the set elements(subset) can generate the input.

The time complexity of back tracking algorithms O(2^n) where n is the input size.

## GRAPH :

X - AXIS: input size nor number of elements in the set.

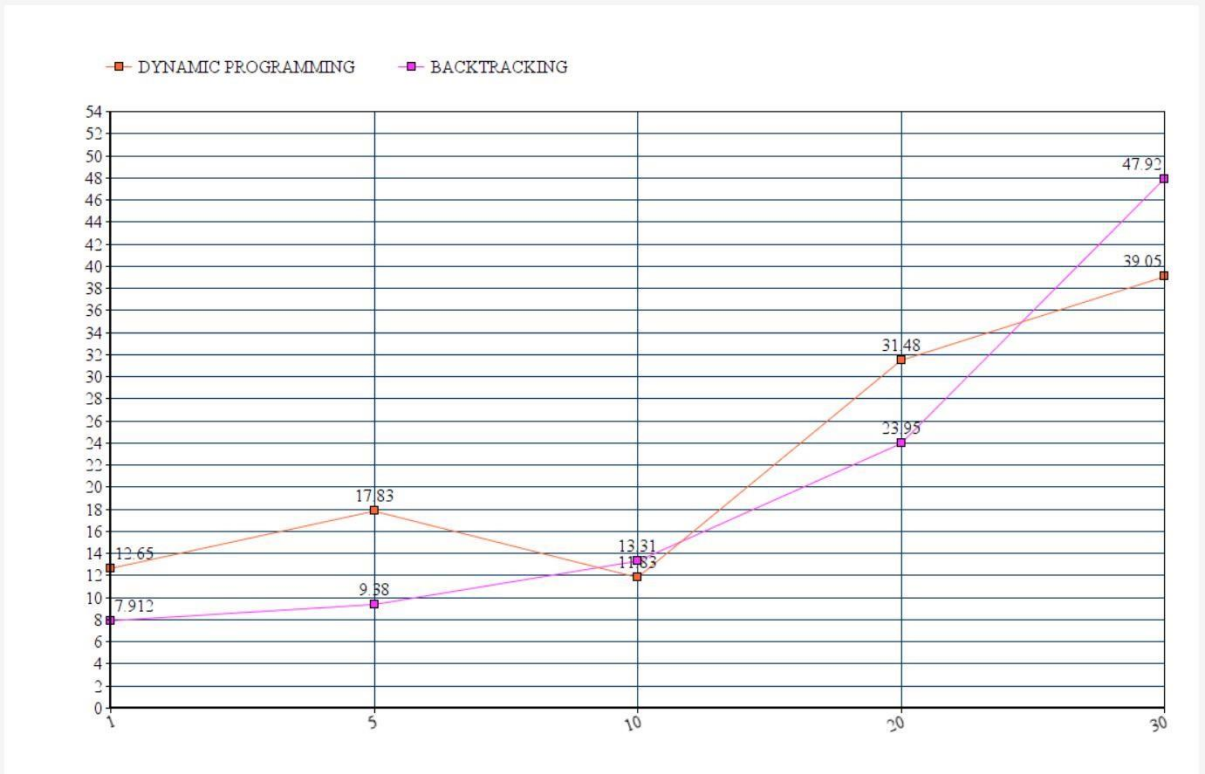
Y-AXIS: time taken.

## DYNAMIC PROGRAMMING ALGORITHM:

| SIZE OF INPUT(n) | TIME TAKEN (secs) |
|---|---|
| 1 | 12.65 |
| 5 | 17.83 |
| 10 | 11.83 |
| 20 | 31.48 |
| 30 | 39.05 |

## BACKTRACKING ALGORITHM:

| SIZE OF INPUT(n) | TIME TAKEN(secs) |
|---|---|
| 1 | 7.912 |
| 5 | 9.38 |
| 10 | 13.31 |
| 20 | 23.95 |
| 30 | 47.92 |




# REFERRED PAPERS:

[1] Baase, S. and Gelder, A. V. (2000). Computer Algorithms. Addison Wesley Longman.

[2] Bazgan, C., Santha, M., and Tuza, Z. (1998). Efficient approximation algorithms for the subset-sum equality problem.

[3] Bentley, J. (1986). Programming Pearls, Addison-Wesley Reading.

[4] Blair, C. (1994). Notes on Cryptography. Business Administration Dept., University of Illinois, http://www.math.sunysb.edu/~scott/blair/Blair_s_Cryptography_Notes.html

[5] Borwein, J. and Bailey, D. (2003) Mathematics by Experiment: Plausible Reasoning in the 21st Century, Natick, MA: A. K. Peters.