**Introduction**

The subset sum problem is an important problem in complexity theory and cryptography. The subset problem is one of the most fundamental NP-complete problems. This simply means that if our input is big enough we may be in trouble.Given integers and .The decision version of subset sum problem asks whether there exists a subset of such that; i.e., whether there exists a subsequence of with sum .

For example, suppose the given a collection

A = {2, 3, 5, 7, 10}

And the sum,

s = 14

Now the problem is to check if there exist any subset of the given collection A whose sum is 14. In this case the answer is ‘true’. We can form a subset as {2, 5, 7} which sums up to get a total of 14 (2 + 5 + 7 = 14).

The maximization version of subset sum problem is to find a subset such that the corresponding total of the elements in the subset is maximized without exceeding the capacity c. 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.

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 problem, scheduling problems, 0-1 integer programs and bin packing algorithms.This problem arises in practical applications. Similar to the knapsack problem we may have a truck that can carry at most *t*pounds and we have *n* different boxes to ship and the box weighs pounds. The problem arises in situations where a quantitative target should be reached, such that its negative deviation (of loss of, e.g., trim, space, time, money) must be minimized and a positive deviation is not allowed.

**Algorithms For subset sum problem**

The complexity of the subset sum problem depends on two parameters, the number of given variables and the precision of the problem i.e. the number of binary place values that it takes to state the problem. The problem gets more difficult when these parameters are large. It becomes easy if either of them becomes very small. If the number of variables is small, then an *Exhaustive Search* for the solution is practical. And if the number of place values is a small fixed number, the thereare dynamic programming algorithms that can solve the problem.

1. **Exhaustive Search**

The Exhaustive Search is an algorithm including or considering all the possible subsets of the given Set of numbers. The Exhaustive search has exponential time complexity. The problem with this algorithm is its runtime complexity. This algorithm gives an exact solution for a small number of inputs. But when the number of inputs increases this algorithm is really a big problem.

Let's say our input looks like:

[1, -3, 2, 4]

We need to iterate through the values and on every iteration produce all the possible subsets that can be made with all the numbers we've looked at up until now. Here is how it looks:

*Iteration 1:*

[[1]]

*Iteration 2:*

[[1], [-3], [1, -3]]

*Iteration 3:*

[[1], [-3], [1, -3], [2], [1, 2], [-3, 2], [1, -3, 2]]

*Iteration 4:*

[[1], [-3], [1, -3], [2], [1, 2], [-3, 2], [1, -3, 2], [4], [1, 4], [-3, 4], [1, -3, 4], [2, 4], [1, 2, 4], [-3, 2, 4], [1, -3, 2, 4]]

On every iteration we simply take the number we're currently looking at as well as a clone of the list of all the subsets we have seen so far, we append the new number to all the subsets (we also add the number itself to the list since it can also be a subset) and then we concatenate this new list to the list of subsets that we generated on the previous iteration. Here is the previous example again, but demonstrating this approach:

1. **Bitwise Algorithm**

To make sure we have considered all the subsets in of our instance, we use a binary number to represent each subset. The length of the binary number is equal to the size of the instance, i.e. the number of elements in the data set. Every bit of the binary number stands for a number among the instance. If a bit is one, it means this number is chosen and should be add to this subset. If a bit is zero, it means we won’t take this number into our subset. For example, below is an instance of size eight, i.e. there are eight numbers in our data set. Then our length of the binary number should be eight. If we have a binary number 01100110, we can know that its , , and bit are one. So we should take , , and into our subset.

**Data set: {}**

**Binary Number: 0 1 1 0 0 1 1 0**

**Subset: {}**

If we change to another binary number, we can get a new subset which is definitely different from our current one. Since the binary numbers is a one to one map to our subsets and the number of subsets is , which is equal to the amount of numbers of a n length binary number, we can know that in this way we considered all possible subsets and each subset is being considered only once.

Once, we’ve got all the subsets, all we should do in the next is just sum all the elements of each subset and compare the sum with the given integer to determine whether they are the same. If the answer is yes, we can know we got our solution.

1. **Recursive Algorithm**

In the recursive algorithm, we start from every single element of our instance. Assume that the size of instance is n, the data set can be denoted as {}, current level is 1. Then for any element , we add all the elements whose index is larger than the current element’s index to it. That is, we can add respectively and get subsets {}, {} , … , and {} . After we did this for every element of the instance, we can get a new bunch of subsets. All those subsets have two elements, and they are different from each other. We say those subsets are on the same level. This level is 2. Based on this new level, we then start 12 from every subset of this level, i.e. from subset {} to {}. Hence, all the two-elements subsets are considered. For each two-elements subset, we add all the elements whose index is larger than the current subset’s maximum element index. That is we add , , , … , and to . After that, we can also get a new bunch of subsets. At this time, all those subsets will have three elements, and they are different from each other. They are now on level 3. Continue doing this kind of job recursively, we can finally traverse all the subsets. And since all the subsets are different from each other, we can know we consider every solution only once. For all these subsets we will consider the sum and compare it with the given integer K. Once one sum is found equal to K, we stop our recursive and return true. Otherwise, we continue our job until all the subsets are considered. One nice thing should be mentioned is that since every current level subset are based on the last level subset, then when we count the summation, we don’t need to sum all current level subset’s elements together. All we need to do is just add the new element from current level to the last level’s sum results. The following figure is an example of a four elements data set. We compute the sum results recursively.

1. **Dynamic programming**

The Exponential time algorithms are only good and applicable for the problems with less number of inputs. When the size of the input grow the complexity of the problem grows exponentially. Thus there is a need of different approach to solve the problem and hence the Dynamic programming comes into action.

Dynamic programming is a method for efficiently solving a broad range of search and optimization problems which exhibit the characteristics of overlapping subproblems and optimal substructure.

1. Optimal Substructure

A problem is said to have optimal substructure if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems. For example, let us consider the following:

inputs: {1, 2, 4, 5, 9} and sum = 15

Our problem is to find a subset from the given inputs which sums up to give a total of 15.

Now, let us first take 1. Our problem changes to:

inputs: {2, 4, 5, 9} and sum = 14

Here our problem reduced to finding a sum of 14 from the reduced inputs. If we can find a subset of {2, 4, 5, 9} that sums up to give 14 then we can add 1 to it so that we can have our initial sum i.e. 15. Thus, this problem has optimal substructure property that if we can get the optimal solution for the subproblem then we can get the optimal solution for the problem too.

The shortest path problem has following optimal substructure property: If a node x lies in the shortest path from a source node u to destination node v then the shortest path from u to v is combination of shortest path from u to x and shortest path from x to v. On the other hand the Longest path problem doesn’t have the Optimal Substructure property. Here by Longest Path we mean longest simple path (path without cycle) between two nodes. Consider the following graph. There are two longest paths from q to t: q -> r ->t and q ->s->t. Unlike shortest paths, these longest paths do not have the optimal substructure property. For example, the longest path q->r->t is not a combination of longest path from q to r and longest path from r to t, because the longest path from q to r is q->s->t->r.

1. Overlapping Subproblems

A problem is said to have overlapping subproblems property if the problem can be broken down into subproblems which are reused several times or a recursive algorithm for the problem solves the same subproblem over and over rather than always generating new subproblems.

Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored so that these don’t have to be recomputed. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again. For example, Binary Search doesn’t have common subproblems. If we take example of following Fibonacci Numbers, there are many subproblems which are solved again and again.

Recursion tree for execution of *fib(5)*

fib(5)

/ \

fib(4) fib(3)

/ \ / \

fib(3) fib(2) fib(2) fib(1)

/ \ / \ / \

fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)

/ \

fib(1) fib(0)

We can see that the function f(3) is being called 2 times. If we would have stored the value of f(3), then instead of computing it again, we would have reused the old stored value.

References

1. AndrisAmbainis, Quantum walk algorithm for element distinctness, SIAM Journal on Computing 37 (2007), 210–239