Table of Content

[Introduction 1](#_Toc468202274)

[Application of Subset Sum Problem 1](#_Toc468202275)

[Exact Algorithms 3](#_Toc468202276)

[1. Exhaustive Search 3](#_Toc468202277)

1. [Bitwise Algorithm 4](#_Toc468202278)
2. [Recursive Algorithm 4](#_Toc468202279)

[2. Dynamic programming 6](#_Toc468202280)

[Approximate Algorithms 13](#_Toc468202281)

[1. Greedy Algorithms 13](#_Toc468202282)

[2. Polynomial-Time Approximation Schemas 15](#_Toc468202283)

[3. Full Polynomial-time approximation schemas 15](#_Toc468202284)

[Conclusion 16](#_Toc468202285)

[References 17](#_Toc468202286)

# 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

And the sum,

Now the problem is to check if there exist any subset of the given collection A whose sum is . In this case the answer is ‘true’. We can form a subset as which sums up to get a total of

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 . 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.

# Application of Subset Sum Problem

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, 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.

There are many more application of Subset Sum Problem in computer technology. Some of them are listed below.

1. Resource Allocation
2. Computer Passwords
3. Message Verification
4. *Resource Allocation*

*Suppose, you want to backup your files into DVD's. What would be the optimal way of doing that, filling up a DVD with a myriad files, making sure that it gets filled up as much as possible? With no free space left unused?*

Think about it. When you have more than a hundred files, there are literally millions of combinations you can try. Which one is the best?

Now, in such situation subset sum can be applied. We can use subset sum algorithms to find the combinations of files which is close enough to the total capacity of a DVD. This will result in minimum waste of unused DVD storage.

1. *Computer Passwords*

A computer needs to verify a user's identity before allowing him or her access to an account. The simplest system would have the machine keep a copy of the password in an internal file, and compare it with what the user types. A drawback is that anyone who sees the internal file could later impersonate the user.

Computer generates a large number of set (say ). They are stored in the internal file. A password is a subset of . (in practice, there is a program to convert a normal sequence-of-symbols password to such a subset.) Instead of having the password for the user, the computer keeps the total associated with the appropriate subset. When the user types in the subset, the computer tests whether the total is correct. It does not keep a record of the subset. Thus impersonation is possible only if somebody can reconstruct the subset knowing the set  and the total.

1. *Message Verification*

A sender (S) wants to send messages to a receiver (R). Keeping the message secret is not important. However, R wants to be sure that the message he is receiving is not from an imposter and has not been tampered with. Sender and Receiver agree on a set (say ) and a set of totals  (say 200). These numbers may be publicly known, but only Sender knows which subsets of the  correspond to which . The message sent by Sender is a subset of size of . He does this by sending 100 subsets of the  corresponding to the message he wants to send.

# Exact Algorithms

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(Brute-force search)* for the solution is practical. And if the number of place values is a small fixed number, then there are 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, also known as generate and test, is a very general problem-solving technique that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement. 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.

*Complexity*

The time complexity for the Exhaustive search is . This means that if we have n numbers of input then we can have subsets of it. So we need to check times in order to get our result.

The Exhaustive Search 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. For a problem with really big numbers of inputs, the Exhaustive search might take months and also years to solve it.

1. **Dynamic programming**

Since for large inputs Exhaustive search is not really practical, there is a need of different approach to solve the problem and hence the Dynamic programming comes into action.

There are two key attributes that a problem must have in order for dynamic programming to be applicable: optimal substructure and overlapping sub-problems. If a problem can be solved by combining optimal solutions to non-overlapping sub-problems, the strategy is called "divide and conquer" instead.

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 = and

sum =

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

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

inputs = and

sum =

Here our problem reduced to finding a sum of from the reduced inputs. If we can find a subset of that sums up to give then we can add to it so that we can have our initial sum i.e. . 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 lies in the shortest path from a source node to destination node then the shortest path from is combination of shortest path from and shortest path from . 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 : and . Unlike shortest paths, these longest paths do not have the optimal substructure property. For example, the longest path is not a combination of longest path from and longest path from , because the longest path from is .

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(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 is being called times. If we would have stored the value of then instead of computing it again, we would have reused the old stored value.

Dynamic programming is a fancy name for using divide-and-conquer technique with a table. As compared to divide-and-conquer, dynamic programming is more powerful and subtle design technique. This technique was developed back in the days when "programming" meant "tabular method" (like linear programming). It does not really refer to computer programming. Dynamic programming is a stage-wise search method suitable for optimization problems whose solutions may be viewed as the result of a sequence of decisions. The most attractive property of this strategy is that during the search for a solution it avoids full enumeration by pruning early partial decision solutions that cannot possibly lead to optimal solution. In many practical situations, this strategy hits the optimal solution in a polynomial number of decision steps. However, in the worst case, such a strategy may end up performing full enumeration.

Dynamic programming takes advantage of the duplication and arrange to solve each subproblem only once, saving the solution (in table or in a globally accessible place) for later use. The underlying idea of dynamic programming is: avoid calculating the same stuff twice, usually by keeping a table of known results of subproblems. Unlike divide-and-conquer, which solves the subproblems top-down, a dynamic programming is a bottom-up technique. The dynamic programming technique is related to divide-and-conquer, in the sense that it breaks problem down into smaller problems and it solves recursively. However, because of the somewhat different nature of dynamic programming problems, standard divide-and-conquer solutions are not usually efficient.

The dynamic programming is among the most powerful for designing algorithms for optimization problem. This is true for two reasons. Firstly, dynamic programming solutions are based on few common elements. Secondly, dynamic programming problems are typical optimization problems i.e., find the minimum or maximum cost solution, subject to various constraints.

In other words, this technique used for optimization problems:

* Find a solution to the problem with the optimal value.
* Then perform minimization or maximization.

*Example*

Let us consider the following:

input:

sum: 15

Let us build a matrix. the key is to figure out what it's a matrix of (how do we label the rows and how do we label the columns). In this case the rows are simply the indexes of our input array and the columns are labeled from through the given sum i.e. . So, our uninitialized matrix looks like this:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 5 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 9 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Now we will fill every cell of the matrix with either T (true) or F (false). A T value in a cell means that the sum that the column is labeled with can be constructed using the input array numbers that are indexed by the current row label and the labels of all the previous rows we have already looked at. An F in a cell means the sum of the column label cannot be constructed. Let's try to fill in our matrix to see how this works.

*First Row:*

We start with the first row, the number indexed by the row label is 1. If we consider our input to be:

input:

then we can get two subsets from this,

subsets:

Hence, we can get the sum of 0 and 1 using the input 1. So, filling 'T' for 1 and 0 and 'F' for the rest of the columns in the first we get our matrix as

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 1 | T | T | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| 2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 5 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 9 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

*Second row:*

For the second our input will be

input:

We already have our outputs for . So, for this case we can use first row to get the second row. The cell value with T in it can be copied to the second row since the sum that we can get from just using can be get using . So first and second column gets the value .

For the rest of the rows which has in the above cell, we subtract current row label with the sum and get the value for the result column from the above row. i.e. for the third column (), when we subtract (current row label) from (the sum) we get the sum . Since the cell value for the sum in above row(for input ) is , this column also gets a .

Again for the fourth column,

sum = ,

row label = ,

sum - row label

The value for the sum in the first row is so the fourth column of second row also gets the value .

Similarly, rest of the cell values for the second row in filled accordingly and we finally get the following matrix.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 1 | T | T | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| 2 | T | T | T | T | F | F | F | F | F | F | F | F | F | F | F | F |
| 4 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 5 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 9 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

*Third row:*

Similarly, other row of the table is filled up.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 1 | T | T | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| 2 | T | T | T | T | F | F | F | F | F | F | F | F | F | F | F | F |
| 4 | T | T | T | T | T | T | T | T | F | F | F | F | F | F | F | F |
| 5 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 9 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

*Fourth row:*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 1 | T | T | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| 2 | T | T | T | T | F | F | F | F | F | F | F | F | F | F | F | F |
| 4 | T | T | T | T | T | T | T | T | F | F | F | F | F | F | F | F |
| 5 | T | T | T | T | T | T | T | T | T | T | T | T | T | F | F | F |
| 9 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

*Fifth row:*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 1 | T | T | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| 2 | T | T | T | T | F | F | F | F | F | F | F | F | F | F | F | F |
| 4 | T | T | T | T | T | T | T | T | F | F | F | F | F | F | F | F |
| 5 | T | T | T | T | T | T | T | T | T | T | T | T | T | F | F | F |
| 9 | T | T | T | T | T | T | T | T | T | T | T | T | T | T | T | T |

After filling all the cells, the last cell gives us the solution of the problem. Here the last cell of the matrix is so it implies that we can get a subset from our inputs(i.e. ) that sums up to get .

If we represent the matrix as a multi-dimensional array of rows(inputs) and (sum) columns. Then represents each cell. The value of the cell can be calculated using

*Tracking elements*

1. Start from the bottom-right corner and back­track and check from where the True is coming.
2. If value in the cell above is false that means current cell has become true after including the current element. So include the current element and check for the sum = current sum - current element and move to the upper element.
3. If value in the cell above is true that means current cell's value came from the above cell. So skip current element and move to the upper element.
4. Repeat the step until the sum becomes zero.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 1 | T | T | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| 2 | T | T | T | T | F | F | F | F | F | F | F | F | F | F | F | F |
| 4 | T | T | T | T | T | T | T | T | F | F | F | F | F | F | F | F |
| 5 | T | T | T | T | T | T | T | T | T | T | T | T | T | F | F | F |
| 9 | T | T | T | T | T | T | T | T | T | T | T | T | T | T | T | T |

For the above problem. The last cell has a true value so the sum of can be gained using giving inputs i.e. For tracking the subset that makes the sum we begin from the bottom cell.

For and , the cell above has a F value so we include current element '9' in the subset. We calculate new sum as

sum = current sum - current element

=

=

And then we move to the upper element i.e. 5.

For sum = 6 and element = 5, the cell above has a T value so current element '5' is skipped and we move upward. Now the element is 4 and sum is still 6.

For sum = 6 and element = 4, the cell above has a F value so we include current element '4' in the subset. We calculate new sum as

sum = current sum - current element

=

=

And the move to upper element i.e.

For and , the cell above has a value so we include current element '2' in the subset. Then we calculate the new sum as

sum = current sum - current element

=

=

Since the new sum is , we end our tracking. Hence, our subset from the input that gives a sum is .

*Complexity*

Since this dynamic programming algorithm solves each subproblem exactly once and the space of all subproblems can be represented by an matrix, the time complexity of this algorithm is equivalent to the total number of elements in the matrix. With rows and columns, there are subproblems. Therefore, the time and space complexity of this dynamic programming algorithm is . For a more thorough evaluation of time complexity, let be the number of distinct sums that must be created with the given set so that the following cases can explain its pseudo-polynomial time behavior.

In the worst case scenario, is greater than or equal to the sum of all elements in the set. As a result, the algorithm must visit every possible sum in order to determine if can be reached at all. By elementary number theory, there are distinct nonempty subsets, so is bounded above as such:

Therefore, the original time complexity (where ) can be rewritten as .

Consider the alternate case when is less than the sum of all elements in the set. The algorithm does not consider every possible subset sum since some would clearly be greater than . Consequently, the algorithm visits only the unique sums under the worst conditions for this particular case, so . For example, consider the set with . There are possible subset sums, but only the sums will be considered because any greater sum is unnecessary, so . Therefore, these conditions preserve the complexity.

This dynamic programming algorithm is considered to be pseudo-polynomial because it behaves as a polynomial time algorithm for large elements in and relatively small , but it is not actually polynomial time as previously shown. However, it is reasonable to conclude that its runtime is because this represents the worst-case conditions according to order of growth analysis, and one cannot ensure that is indeed bounded by the sum of the elements in the set. Note that the complete search algorithm given earlier also runs in . Although the time complexities of both algorithms are identical, the dynamic programming one is generally faster due to its use of optimal substructure and overlapping subproblems. In fact, this is the fastest known runtime of any classical algorithm for the Subset Sum Problem.

# Approximate Algorithms

An approximate algorithm is a way of dealing with NP-completeness for optimization problem. This technique does not guarantee the best solution. The goal of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at most polynomial time. Approximate Algorithms are applied for very large inputs considering the time for finding the optimal solution with the exact algorithms.

1. **Greedy Algorithms**

Greedy algorithms are simple and straightforward. They are shortsighted in their approach in the sense that they take decisions on the basis of information at hand without worrying about the effect these decisions may have in the future. They are easy to invent, easy to implement and most of the time quite efficient. Many problems cannot be solved correctly by greedy approach. Greedy algorithms are used to solve optimization problems.

Greedy Algorithm works by making the decision that seems most promising at any moment, it never reconsiders this decision, whatever situation may arise later. Unlike Dynamic Programming, which solves the subproblems bottom-up, a greedy strategy usually progresses in a top-down fashion, making one greedy choice after another, reducing each problem to a smaller one.

To construct the solution in an optimal way. Algorithm maintains two sets. One contains chosen items and the other contains rejected items.

The greedy algorithm consists of four () function.

1. A function that checks whether chosen set of items provide a solution.
2. A function that checks the feasibility of a set.
3. The selection function tells which of the candidates is the most promising.
4. An objective function, which does not appear explicitly, gives the value of a solution.

Structure of Greedy Algorithm is as follows:

* Initially the set of chosen items is empty i.e., solution set.
* At each step
* item will be added in a solution set by using selection function.
* IF the set would no longer be feasible

reject items under consideration (and is never consider again).

* ELSE IF set is still feasible THEN

add the current item.

*Example*

Input:

Sum:

Here, in this problem, greedy algorithm at first searches for the largest number in the input which is feasible to get the desired sum i.e. . So it first takes rejecting because is greater than (required sum). After that it takes rejecting . Because is already in the solution set so adding or to will yield a sum that exceed (required sum). Hence, the problem is solved with as the solution to set the sum .

*Cases of Failure*

The Greedy algorithm does not always find the optimal solution. For example, consider a given set and . If we use greedy algorithm, the solution will be to get the greatest sum closest to required sum (i.e. ) , but if we just pick , is an optimal solution. In such situation, the greedy algorithm cannot find optimal solution. The reason this situation will happen is because greedy algorithm always try to contain the heaviest item into the optimal solution. But actually the optimal solution doesn’t necessarily contain such item. There may be several light objects that have a total weight that much more close to the target value.

*Complexity*

The time complexity of greedy algorithm is dominated by time required by the sorting algorithm used to sorting the numbers in hand. The Greedy algorithm itself needs linear time which is as small as

1. **Polynomial-Time Approximation Schemas**

When faced with an NP-hard problem one cannot expect to find a polynomial-time algorithm  
that always gives an optimal solution. Hence, one has to settle for an approximate solution.  
Of course one would prefer that the approximate solution is very close optimal, for example  
at most worse. In other words, one would like to have an approximation ratio very  
close to . The approximation algorithms we have seen so far do not quite achieve this:  
for Load Balancing we gave an algorithm with approximation ratio , for *Weighted  
Vertex Cover* we gave an algorithm with approximation ratio , and for Weighted the approximation ratio was even . Unfortunately it is not always possible  
to get a better approximation ratio: for some problems one can prove that it is not only  
NP-hard to solve the problem exactly, but that there is a constant such that there is  
no polynomial-time *c*-approximation algorithm unless . *Vertex Cover*, for instance,  
cannot be approximated to within a factor unless , and for Set Cover one  
cannot obtain a better approximation factor than .

Fortunately there are also problems where much better solutions are possible. In particular, some problems admit a so-called polynomial-time approximation scheme, or PTAS for  
short. A Polynomial-Time Approximation Schemas (PTAS) for any problem is an approximation scheme whose time complexity is polynomial in the input size. Such an algorithm works as follows. Its input is, of course, an instance of the problem at hand, but in addition there is an input parameter . The output of the algorithm is then a solution whose value is at most · *OPT* (for a minimization problem) or at least · *OPT* (for a maximization problem).

The running time of the algorithm should be polynomial in ; its dependency on can be exponential however. So the running time can be for example, or , or , etc.

1. **Full Polynomial-time approximation schemas**

A Full Polynomial-Time Approximation Scheme (FPTAS) is a PTAS with a running time that is polynomial not only in but also in .

Example: a PTAS with a running time bound of is an FPTAS.

# Conclusion

The Subset Sum Problem is an important problem in complexity theory and cryptography. It can be applied to various practical fields. Considering the Computer science field too it is very important and have various applications. Some modification to this problem can more extend its usage.

There are various algorithm to solve the Subset sum problem. Different algorithms can be used considering the number of decision variables and the precision of the problem. If the number of variables is small, then exponential time algorithm such as 'Exhaustive search' can be applied. Exhaustive search is very simple algorithm. It is very practical for the problem with small number of variables. When the number of variables is large then the complexity of Exhaustive search increases exponentially to the number of variables. Hence we have Dynamic algorithm to solved the problem with large number of variables. Dynamic algorithm reduces its complexity by using the Optimal Substructure and Overlapping Subproblems property of the problem.

For extremely large inputs, in some cases, even dynamic programming too is inefficient. In such cases it's better to find an optimal solution rather than the exact solution. In order to find the optional solution we can use various Approximate Algorithms. Approximate Algorithms does not ensures to provide the exact solution but it provides a solution close to exact solution and make problem with extremely large inputs practical.

# References

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