Subset sum problem using backtracking and recursive brute force approaches

*Note: Using the same problem in two different approaches

1st Minna Hany Mohammed Abdelhady
2021/09122
Misr International University
Egypt, Cairo.
minna2109122@miuegypt.edu.eg

2nd Mohamed Hisham Sayed
2021/06505
Misr International University
Egypt, Cairo.
mohamed2106505@miuegypt.edu.eg

3rd Mohanad Ahmed Mohamed
2021/07828
Misr International University
Egypt, Cairo.
mohanad2107828@miuegypt.edu.eg

4th Abdulrahman Abdalmoniem Ahmed
2021/11656
Misr International University
Egypt, Cairo.
abdulrahman2111656@miuegypt.edu.eg

5th Dr / Ashraf Abdel Raouf
Supervisor and Professor at MIU
Misr International University
Egypt, Cairo.
ashraf.raouf@miuegypt.edu.eg

***Abstract*—The Subset Sum problem entails identifying a subset of elements from a given list such that their sum equals a specific target value. This problem has wide-ranging applications in fields like optimization, cryptography, and data analysis. Efficiently solving the Subset Sum problem is crucial for addressing real-world challenges requiring the identification of valid subsets.**

***Index Terms*—subset sum, Recursive brute force, Backtracking, complexity**

I. Introduction

The Subset Sum Problem is a well-known computational problem that involves finding a subset of a set of integers that adds up to a target sum. Recursive brute force and backtracking is a common approach to solving this problem, which involves systematically generating all possible subsets and checking whether they add up to the target sum. This approach has been studied and implemented in various contexts, as seen in works such as "A Recursive Brute-Force Algorithm for the Subset-Sum Problem" (Storer, 1982) and "A Backtracking Algorithm for the Subset Sum Problem" (Aho and Ullman, 1972). These works demonstrate the effectiveness and versatility of recursive brute force and backtracking for tackling the Subset Sum Problem.

II. Subset sum problem

The subset sum problem is a well-known computational problem in computer science and mathematics. Given a set of integers and a target sum, the problem asks whether there exists a non-empty subset of the input set that sums to the target value. The problem is typically formulated as a decision problem, where the answer is either "yes" or "no", depending on whether such a subset exists or not. The subset sum problem has numerous applications in cryptography, optimization, and computer science, and is often used as a benchmark problem for testing the efficiency of algorithms. However, the problem is known to be NP-complete, which means that it is unlikely that there exists a polynomial-time algorithm that solves the problem for all possible inputs. As a result, various approximate and heuristic algorithms have been developed to solve the subset sum problem for practical purposes.

Identify applicable funding agency here. If none, delete this.

III. Recursive brute force approach

A. What is recursive brute force?

Recursive brute force is a problem-solving approach that involves systematically generating all possible solutions to a problem by exhaustively exploring the solution space. This is typically done by recursively exploring all possible combinations of choices and evaluating each one until a solution is found. This approach can be computationally expensive, especially for large problem instances, but it guarantees finding a solution if one exists. Recursive brute force is commonly used in algorithm design and optimization, and is often applied to problems such as the Subset Sum Problem, Traveling Salesman Problem, and Knapsack Problem, among others.

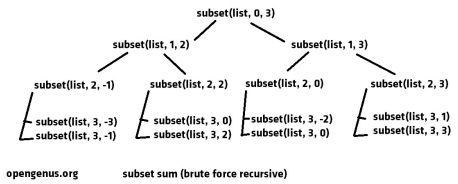


Fig. 1. subset sum recursive brute force

B. Efficiency

1) Time complexity: The subset sum problem is an optimization problem that asks whether a given set of integers contains a non-empty subset that sums to a particular target value. One way to solve it is by using a recursive brute force approach, which involves generating all possible subsets of the input set and checking each one to see if it sums to the target value. The time complexity of this approach is exponential, specifically

$$O(2^n)$$

, where n is the size of the input set. This is because each element in the input set can either be included or excluded from a subset, resulting in

$$2^n$$

possible subsets to check. As the size of the input set grows, the number of possible subsets and the time required to check each one increases exponentially, making this approach impractical for large input sizes.

2) Space complexity: The space complexity of the subset sum problem using recursive brute force approach is also exponential, specifically

$$O(2^n)$$

. This is because the algorithm needs to store all possible subsets of the input set in memory in order to check each one for the subset sum property. Since there are

$$2^n$$

possible subsets, the algorithm needs to allocate exponential memory to store them all. This can quickly become impractical for large input sizes, as the space required grows exponentially with the input size. Additionally, since the algorithm generates all possible subsets, it may generate many redundant subsets, further increasing the space requirements. As a result, the recursive brute force approach to solving the subset sum problem is not suitable for large input sizes due to its exponential space complexity.

C. Recursive Brute Force Method

The recursive brute force approach for the subset sum problem involves generating all possible subsets of the input set and checking each one to see if it sums to the target value. The algorithm starts by considering the first element of the input set and recursively generates all subsets that include or exclude that element. For each subset, the algorithm checks if the sum of its elements equals the target value. If a subset is found that has the desired sum, the algorithm returns true; otherwise, it continues generating subsets until all possible subsets have been considered. This approach is straightforward to implement, but its time and space complexity are exponential, making it impractical for large input sizes. As a result, more efficient algorithms, such as dynamic programming or backtracking, are typically used to solve the subset sum problem in practice.

IV. Backtracking approach

A. What is backtracking?

Backtracking is an algorithmic approach for recursively solving problems by trying to construct a solution step-by-step, one piece at a time, removing those solutions that fail to satisfy the restrictions of the problem at any point in time the amount of time it took to reach any level of the search tree. Backtracking is the process of seeking for a solution to a problem among all accessible choices. Begin retracing your steps from one viable alternative, and if the problem is solved with that option, return the answer; otherwise, backtrack and choose another option from the remaining available possibilities. There is also the possibility that none of the possibilities will provide you with the solution, in which case we recognise that retracing is not an option. There could also be situations where none of the possibilities provide a solution, in which case we realise that backtracking will not provide a solution. Backtracking can also be considered a type of recursion. This is because the procedure of selecting a solution from the available options is continued recursively until we either don't find a solution or reach the end state. So we may infer that backtracking at each stage removes options that will not lead us to the answer and advances us to options that will lead us to the solution.

B. Efficiency

- SUBSET SUM, there is an apparent exponential-time exhaustive search algorithm: produce all subsets sequentially and compute their sums. If S contains n items, it has

  $$2^n$$

  subsets, and the exhaustive search algorithm for some polynomial function p(n) is

  $$O(p(n)2n)$$

  . The search, like many other NP-complete problems, may be carried out using a depth-first technique using multiple bounds conditions to decrease the size of the search space. While their worst-case time complexity remains

  $$O(p(n)2^n)$$

  , such backtracking algorithms are typically the most feasible when both time and space are considered.

C. Backtracking Method

We use exhaustive search to explore all subsets, regardless of whether they fulfil the stated restrictions or not. Backtracking can be used to conduct a thorough examination of the elements to be chosen.

Assume a collection of four items, w[1]... w[4]. Backtracking algorithms may be designed using tree diagrams. The strategy to generate variable sized tuples is depicted in the tree diagram below.

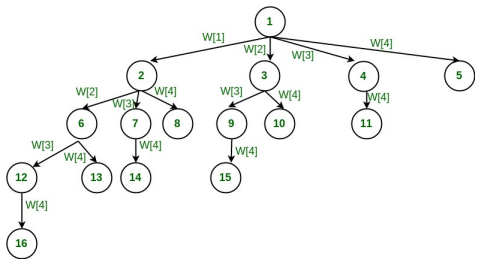


Fig. 2. state-space tree

A node in the preceding tree represents a function call, and a branch indicates a potential element. The root node has 4 children. In other words, root treats each element of the set as a separate branch. The subset that include the parent node correspond to the next level sub-trees. Each level's branches indicate a tuple element to be examined. For example, if we are at level 1, tuple_vector[1] can take any of the four created values. If we are at level 2 of the leftmost node, tuple_vector[2] can take any one of the three created branches, and so on...

For example, root's leftmost child yields all subsets that contain w[1]. Similarly, root's second child yields all subsets that contain w[2] but exclude w[1].

As we descend the depth of the tree, we add items thus far, and if the additional total meets the defined requirements, we will continue to construct child nodes. When the restrictions are not fulfilled, we stop generating subtrees of that node and go back to the previous node to examine nodes that have not yet been visited. It saves a significant amount of processing time in many instances.

The tree should provide a hint on how to perform the backtracking method . It prints all subsets whose sums equal the provided integer. We must investigate the nodes along the tree's breadth and depth. The loop controls the generation of nodes along the breadth, whereas recursion (post order traversal) is used to construct nodes along the depth.

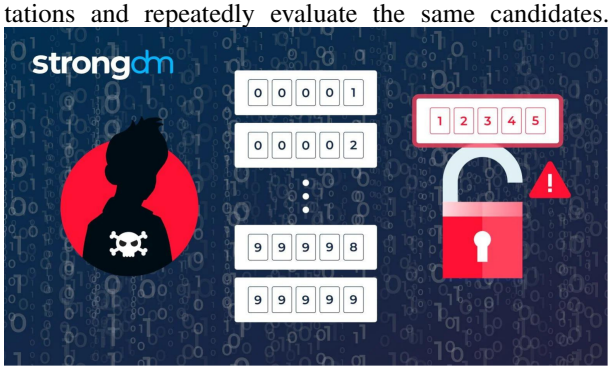
Comparison between Backtracking Approach and Recursive Brute Force Approach

The backtracking approach and the recursive brute force approach are algorithmic techniques used for problem-solving, particularly in combinatorial search or optimization problems. While they share some similarities, there are notable differences between the two approaches. Let's examine each approach in detail and compare them.

Recursive Brute Force Approach

The recursive brute force approach is a simple and straightforward method that systematically explores all possible solutions by exhaustively generating and evaluating all candidate solutions. It relies on recursive function calls to generate all possible combinations or permutations of a problem's input. Here are some characteristics of the recursive brute force approach:

1) **Exhaustive Search:** The recursive brute force approach explores all possible solutions, making it suitable for problems where the solution space is relatively small and feasible to enumerate completely.
2) **Simple Implementation:** Recursive brute force algorithms are typically easier to implement compared to other complex algorithms, as they involve straightforward recursive function calls and base cases.
3) **Inefficiency:** Recursive brute force algorithms can be highly inefficient, especially when dealing with large solution spaces. They often involve redundant computations and repeatedly evaluate the same candidates.

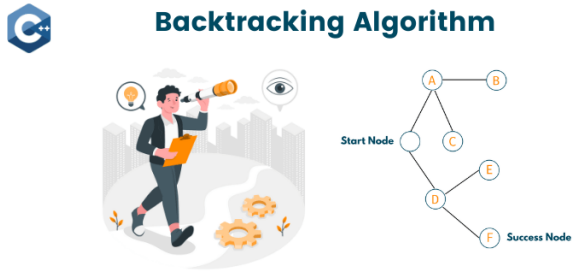


Backtracking Approach

The backtracking approach is an optimization of the brute force approach that reduces unnecessary exploration of the solution space by intelligently discarding invalid or unpromising paths. It is particularly useful for problems with larger solution spaces where the exhaustive search becomes impractical. Let's examine the characteristics of the backtracking approach:

1) **Depth-First Search with Pruning:** Backtracking employs a depth-first search strategy to explore the solution space incrementally. It generates a partial solution, checks if it can be completed further, and if not, backtracks to explore other branches. Pruning techniques are used to discard paths that are guaranteed to lead to invalid solutions, thereby reducing unnecessary exploration.
2) **Smart Decision-Making:** Backtracking incorporates a notion of "smart" decision-making at each step. It selectively chooses the most promising candidates based on certain constraints or rules defined by the problem. If a candidate violates any constraint, it is immediately discarded, saving time and effort.
3) **Improved Efficiency:** Backtracking's ability to discard unpromising paths and avoid redundant computations makes it more efficient than the

recursive brute force approach for larger solution spaces. It significantly reduces the number of solution candidates that need to be evaluated.



In general, backtracking is faster than recursive brute force for solving the subset sum problem. This is because backtracking prunes the search space by eliminating subsets that cannot contribute to the desired target sum, resulting in a much smaller search space and faster running times. Recursive brute force, on the other hand, generates all possible subsets and checks each one for the subset sum property, leading to an exponential time and space complexity that can quickly become impractical for large input sizes.

However, the actual running time of either algorithm depends on the specific input and implementation details. In some cases, recursive brute force may be faster than backtracking, particularly if the input set is small or if the backtracking implementation is not optimized. Nonetheless, in general, backtracking is considered to be a more efficient algorithm for solving the subset sum problem.

REFERENCES

[1] https://dl.acm.org/doi/abs/10.1145/361219.361224/
[2] https://iq.opengenus.org/subset-sum-problem-backtracking/
[3] https://www.geeksforgeeks.org/subset-sum-problem/
[4] Li, Z., Cao, J., Yao, Z., Li, W., Yang, Y. and Wang, J., 2020, October. Recursive Balanced k-Subset Sum Partition for Rule-constrained Resource Allocation. In Proceedings of the 29th ACM International Conference on Information Knowledge Management (pp. 2121-2124).
[5] Kondrak, G. and Van Beek, P., 1997. A theoretical evaluation of selected backtracking algorithms. Artificial Intelligence, 89(1-2), pp.365-387.
[6] Priestley, H.A. and Ward, M.P., 1994. A multipurpose backtracking algorithm. Journal of Symbolic Computation, 18(1), pp.1-40.
[7] https://www.researchgate.net/publication/268397336$_{A}S imple_{2}O_{x}A lgorithm_{f}or_{P}ARTITION_{a}nd_{S}UBSET_{S}UM$