Partition Problem

Dima Robert-Ercean

University Polytechnic of Bucharest

1 Introduction

1.1 Problem Description

The **Partition Problem** is the task of deciding whether a given set S of positive integers can be partitioned into two subsets *S1* and *S2* such that the sum of the numbers in *S1* equals the sum of the numbers in *S2*. It is a special case of two related problems:

- In the **Subset Sum** problem, the goal is to find a subset of S whose sum is a certain target number T given as input (the partition problem is the special case in which T is half the sum of S).
- In **Multiway Number Partitioning**, there is an integer parameter k, and the goal is to decide whether S can be partitioned into k subsets of equal sum (the partition problem is the special case in which $\mathbf{k = 2}$).

1.2 Applications

One practical application of the Partition Problem is in election manipulation. Consider an election with three candidates: A, B, and C. A scoring-based voting rule, such as the veto rule, is used to determine the winner. Under this rule, each voter casts a veto against a single candidate, and the candidate with the fewest vetoes wins.

If a coalition of voters wants to ensure that C is elected, they must strategically distribute their votes between A and B to balance the number of vetoes each receives. Their goal is to maximize the minimum number of vetoes assigned to either A or B, ensuring neither of them has fewer vetoes than C.

2 NP-Completeness

Although the **Partition Problem** is **NP-Complete**, it admits a *pseudo-polynomial time dynamic programming solution.* Additionally, various heuristic approaches exist that can efficiently solve many instances either optimally or approximately. Due to these characteristics, Partition is sometimes referred to as "the easiest hard problem".

2.1 Subset Sum

The **Subset Sum** problem is a decision problem, where, given a set S of integers and a target sum T, the goal is to determine whether there exists a subset of S whose elements sum exactly to T. This problem is known to be **NP-complete**.

2.2 Reduction from Subset Sum

Proof. Let (L, B) be an instance of the **Subset Sum** problem, where L is a set of numbers, and B is the target sum. Define $S = \sum L$. We'll construct a new set L' by adding the elements $S + B$ and $2S - B$ to L. Therefore, the following holds true:

$$\sum L' = S + B + 2S - B + \sum L = 4S.$$

We now show that solving **Partition** for L' is equivalent to solving **Subset Sum** for (L, B).

Implication: If a subset of L sums to B, then L' can be partitioned into two equal parts.

Suppose there exists a subset $M \subseteq L$ such that:

$$\sum M = B.$$

Then, L' can be partitioned into two subsets:

$$P_1 = M \cup \{2S - B\}, \quad P_2 = (L \setminus M) \cup \{S + B\}.$$

Calculating the sums, we get:

$$\sum P_1 = B + (2S - B) = 2S, \quad \sum P_2 = (S - B) + (S + B) = 2S.$$

Thus, L' can be partitioned into two equal-sum subsets, as required.

Reverse Implication: If L' can be partitioned into two equal parts, then a subset of L sums to B.

Suppose L' can be partitioned into two subsets P_1, P_2 such that:

$$\sum P_1 = \sum P_2 = 2S.$$

Since $S + B + (2S - B) = 3S \neq 2S$ and $\forall x \in L', x \geq 0$, the two elements **must belong to different partitions**.

Next, we assume that:

$$2S - B \in P_1.$$

Since all the remaining elements in P_1 can only come from L and:

$$\sum P_1 = 2S$$

these elements must sum to B, meaning that we have found a subset $M \subseteq L$ such that:

$$\sum M = B.$$

Finally, we have shown the polynomial-time reduction:

$$\textbf{Subset Sum} \leq_p \textbf{Partition}$$

Therefore:

$$\textbf{Partition} \in \textbf{NP-Hard}.$$

Moreover, since a solution to **Partition** (i.e: *a valid partition*) can be verified in polynomial time, it is implied that:

$$\textbf{Partition} \in \textbf{NP}.$$

Consequently, we conclude that:

$$\textbf{Partition} \in \textbf{NP-Complete}$$

3 Algorithm Solutions

3.1 Brute Force Approach

The **Partition Problem** can be formulated as a **Subset Sum Problem**. Given a list of integers, the goal is to determine whether it can be split into two subsets of equal sum.

To achieve this, we first compute the **total sum** of the array:

$$\text{totalSum} = \sum_{i=1}^{n} \text{nums}[i]$$

For a valid partition to exist, this sum **must be even**, since an odd sum could only be divided into two subsets of different parity. If totalSum is odd, we simply return false, otherwise, we set:

$$\text{subSetSum} = \frac{\text{totalSum}}{2}$$

The problem then reduces to finding a **subset** whose sum is exactly subSetSum since we know for sure the remaining elements of the other subset would be the complement of *subSetSum* in regard to *totalSum*:

$$subSetSum_2 = totalSum - subSetSum = totalSum - \frac{totalSum}{2} = subSetSum$$

This approach explores **all possible subsets** of the given array and checks whether any subset sums to subSetSum. This is done using **recursion with depth-first search (DFS)**.

Each element in the array has **two choices**:

1. **Include** the element in the subset, reducing subSetSum accordingly.
2. **Exclude** the element, keeping subSetSum unchanged.

Therefore, any partition *nums* of size n, described by the recursive call

$$\text{isSum}(\text{subSetSum}, n)$$

will consider both of the mentioned decisions:

$$\text{isSum}(\text{subSetSum} - \text{nums}[n], n-1) \vee \text{isSum}(\text{subSetSum}, n-1)$$

where:

- $\text{isSum}(\text{subSetSum} - \text{nums}[n], n-1)$: **Includes** `nums[n]` in the subset.
- $\text{isSum}(\text{subSetSum}, n-1)$: **Excludes** `nums[n]` from the subset.

Listing 1.1. Brute Force Solution

```cpp
class Solution {
public:
    bool canPartition(vector<int> &nums) {
        int totalSum = 0;
        // Find sum of all array elements
        for (int num : nums) {
            totalSum += num;
        }
        // If totalSum is odd, it cannot be partitioned into equal sum subsets
        if (totalSum % 2 != 0) return false;
        int subSetSum = totalSum / 2;
        int n = nums.size();
        return dfs(nums, n - 1, subSetSum);
    }

    bool dfs(vector<int> &nums, int n, int subSetSum) {
        // Base Cases
        if (subSetSum == 0)
            return true;
        if (n == 0 || subSetSum < 0)
            return false;
        bool result = dfs(nums, n - 1, subSetSum - nums[n - 1]) || dfs(nums, n - 1, subSetSum);
        return result;
    }
};
```

Decision Tree Visualization

Let's analyze the path of the decision tree which returns **true** for the following input:

$$\text{nums} = [6, 7, 4, 3]$$

The target subset sum is computed as:

$$\text{subSetSum} = \frac{\sum \text{nums}}{2} = \frac{6+7+4+3}{2} = 10$$

dfs(10, [6,7,4,3])
Exclude 3
↓
dfs(10, [6,7,4])
Include 4
↓
dfs(6, [6,7])
Exclude 7
↓
dfs(6, [6])
Include 6
↓
dfs(0, [ ])
Returns True

Complexity Analysis

- **Time Complexity**:

  $$O(2^n)$$

  If there are n elements to be partitioned, the time complexity of this algorithm is $O(2^n)$, since there are 2^n subsets of an n element set and in the worst-case scenario we'll have to explore up until the very last possible partition.
- **Space Complexity**:

  $$O(n)$$

  The recursion stack stores at most n recursive calls at any time.

Possible improvements

Many subproblems are solved multiple times, so one key observation is that we could get rid of redundant computations by somehow storing previously calculated results. For this, we can use **memoization (top-down DP)** or **dynamic programming (bottom-up DP)** to store previously computed results and avoid redundant calculations, as presented in the following paragraphs.

3.2 Memoization by Top-Down Dynamic Programming Approach

As mentioned earlier, we could significantly reduce the number of repeated cumputations by storing previously calculated results, thus sacrificing space for time.

Given an array of n elements, we compute the total sum:

$$\text{totalSum} = \sum_{i=1}^{n} \text{nums}[i]$$

If totalSum is **odd**, partitioning is impossible. Otherwise, we define:

$$\text{subSetSum} = \frac{\text{totalSum}}{2}$$

Optimization

The solution to the subset sum problem can be derived from the optimal solutions of smaller subproblems. Specifically, for any given partition of size n and a target sum, we can express the recursive relation as follows:

- If the last element: arr$[n-1]$ >sum, we cannot include it in our subset, so that means the only possible solution must come from the remaining *n - 1* elements.

$$\text{isSubsetSum}(\text{arr}, n, \text{sum}) \implies \text{isSubsetSum}(\text{arr}, n-1, \text{sum})$$

- If the last element is less than or equal to sum, we have two choices:
  - **Include the last element**:

$$\text{isSubsetSum}(\text{arr}, n, \text{sum}) = \text{isSubsetSum}(\text{arr}, n-1, \text{sum} - \text{arr}[n-1])$$

  - **Exclude the last element**:

$$\text{isSubsetSum}(\text{arr}, n, \text{sum}) = \text{isSubsetSum}(\text{arr}, n-1, \text{sum})$$

Listing 1.2. Memoization Approach

```
class Solution {
public:
    bool isSubsetSum(int n, vector<int>& arr, int sum,
                     vector<vector<int>> &memo) {

    // base cases
    if (sum == 0)
        return true;
    if (n == 0)
        return false;

    if (memo[n-1][sum]!=-1) return memo[n-1][sum];

    // If element is greater than sum, then ignore it
    if (arr[n-1] > sum)
        return isSubsetSum(n-1, arr, sum, memo);

    // Check if sum can be obtained by any of the following
    // (a) including the current element
    // (b) excluding the current element
    return memo[n-1][sum] =
             isSubsetSum(n-1, arr, sum, memo) ||
             isSubsetSum(n-1, arr, sum - arr[n-1], memo);
}

// Returns true if arr[] can be partitioned in two
// subsets of equal sum, otherwise false
bool canPartition(vector<int>& arr) {

    // Calculate sum of the elements in array
    int sum = accumulate(arr.begin(), arr.end(), 0);

    // If sum is odd, there cannot be two
    // subsets with equal sum
    if (sum % 2 != 0)
        return false;

    vector<vector<int>> memo(arr.size(), vector<int>(sum+1,
        -1));

    // Find if there is subset with sum equal
    // to half of total sum
    return isSubsetSum(arr.size(), arr, sum / 2, memo);
}

};
```

Complexity Analysis

- **Time Complexity**: The recursive function `isSubsetSum(n, sum)` depends on two parameters:
  - n — the number of elements considered.
  - sum — the target subset sum.

  Since we use memoization, each subproblem (n, sum) is solved only **once** and stored in a **2D table** of size $n \times \text{sum}$. As a result, the total number of computations is $O(n \times \text{sum})$, leading to a *time complexity* of:

$$O(n \times S)$$

Pseudo-Polynomial Nature of the Complexity

Although this complexity appears polynomial, it is actually **pseudo-polynomial**. The reason is that the runtime depends not only on n, the number of elements, but also on sum, which is the sum of all elements in the set.
In a **true polynomial-time algorithm**, complexity is expressed as $O(n^c)$ for some constant c, where only the input size n influences runtime. However, in this case, the complexity also depends on the **numerical values** of the input elements.

Bit Complexity Analysis

Let x be the number of input bits encoding the entire problem. Each input number will have $O(x/n)$ bits since we assume the input array has a homogenous data type. Therefore, the magnitude of each input number will be:

$$O(2^{x/n})$$

Since sum is the total sum of the numbers, we approximate its size as:

$$S = O(n \cdot 2^{x/n})$$

Thus, the overall time complexity of the DP approach becomes:

$$O(n^2 \cdot 2^{x/n})$$

which increases **exponentially** as x (the number of input bits) increases.
Since the runtime scales with the **magnitude** of numbers rather than just the number of elements, the approach is considered **pseudo-polynomial**. Despite this, it remains significantly more efficient than the brute force approach $O(2^n)$ and is practical for small sum values.

– **Space Complexity**: The algorithm utilizes a *2D memoization table* of size $n \times \text{sum}$, where each entry stores the result of a subproblem. This results in a *space complexity of*:

$$O(n \times \text{sum})$$

Additionally, the recursive function calls require *stack space* proportional to the recursion depth, which is at most $O(n)$, which will not change the asymptotic complexity. However, this extra space can be optimized using an iterative DP approach.

4 Graphs

4.1 Execution time analysis based on the length of the input array

As we've seen earlier, one of the advantages of the *Dynamic Programming* approach is that it only *pseudo-polynomially* increases with the length of the input array, as long as the total sum of the elements doesn't surpass a certain threshold. Meanwhile, the brute force will exponentially increase as n gets bigger.

Test cases construction
To illustrate this fact, test cases **1 - 7** were built in the following manner:

– For each test case, we incrementally increase the size of the array by a value of **5**
– The test case generator code is managed in such a way that the total sum of the numbers will **never** exceed the *median value of the array* $\times$ n and because each value is in the interval *[10, 100]*, we'll never reach huge sum values.

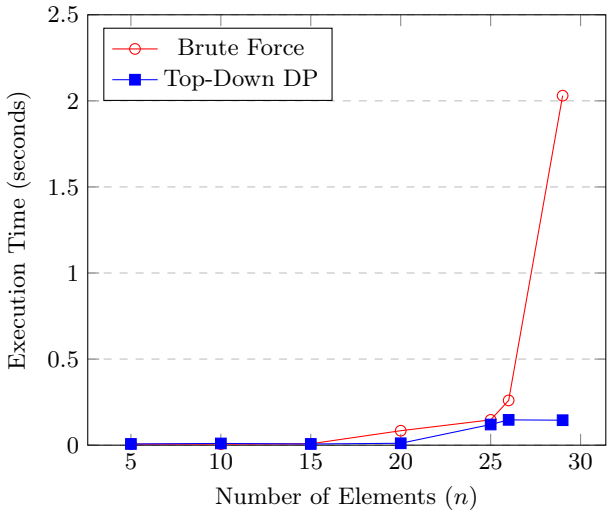

Fig. 1. Execution Time against Input Size

Interpretation
As seen above, the execution time values of the two approaches behaves as expected. Since for each test case, the total sum changes only by a small, negligible value *(i.e. 200)*, the *Dynamic Programming* execution time can be seen as linearly increasing in regards to the input size. Meanwhile, the execution time for the *Brute Force* approach bursts once the threshold of $n = 27$ is surpassed.

4.2 Execution time analysis based on the size in bits of the total sum

Next, we'll test the fact that the *Dynamic Programming* approach, while giving better execution times for larger input array lengths, will start bursting to **very large** values, once this array's total sum gets bigger.

Test cases construction
To exemplify this, test cases **8 - 15** were constructed such that:

- Each test case has a fixed array length of *10*, so we illustrate the variation of the execution time in regards to the total sum of the array's elements.
- Each array element gets increasingly bigger by an arbitrary value, unique to each test case.
- Each test case has a median array value that gets doubled at each test iteration, starting from an initial value of 1.5×10^5.

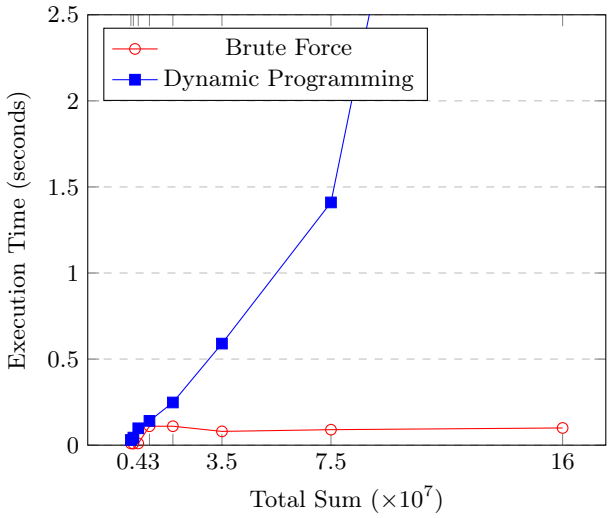

Fig. 2. Execution Time vs. Total Sum size

Interpretation
As demonstrated, the *Brute Force* approach doesn't really vary in regards to the total sum of the elements and the total execution time stays somewhat constant.

The *Dynamic Programming* method begins experiencing large execution times compared to the other approach even from the first test case. We can see a huge burst and observe the *exponential growth* pattern once the total sum reaches the value 1×10^7.

4.3 System Specifications

- **Processor**: Intel i5-10300H @ 2.5GHz, 4 cores, 8 threads
- **OS**: WSL2.3, Kernel: 5.15.167.4-1, Host: Windows 10.0.19045
- **Memory**: 16GB

5 Conclusion

These results confirm that while the **Brute Force** approach is feasible for small inputs, it becomes impractical as the input size increases due to its **exponential time complexity** $O(2^n)$. The **execution time skyrockets** once a certain threshold is reached, making it unsuitable for real-world applications with large datasets.

On the other hand, the Dynamic Programming approach demonstrates superior scalability due to its pseudo-polynomial time complexity $O(n \times \text{sum})$. It efficiently handles larger inputs where brute force fails, but its performance is

affected by extremely large sum values due to the growing DP table. Additionally, DP requires more memory, which can be a limiting factor in constrained environments.
The algorithm choosing decision should be guided by the specific application constraints, balancing **execution time, memory usage, and scalability** for optimal performance.