Minimum Partition Problem Using Two Different Approaches

Ammar Bektash
Computer Science
MISR International University
Cairo, Egypt
ammar2201210@miuegypt.edu.eg

Youssef Abdelshahid
Computer Science
MISR International University
Cairo, Egypt
youssef2205890@miuegypt.edu.eg

Abdelrahman Khalil
Computer Science
MISR International University
Cairo, Egypt
abdelrahman2202178@miuegypt.edu.eg

Ziad Samy
Computer Science
MISR International University
Cairo, Egypt
ziad2205959@miuegypt.edu.eg

Youssef Mahmoud
Computer Science
MISR International University
Cairo, Egypt
youssef2207740@miuegypt.edu.eg

Ashraf AbdelRaouf
Computer Science
MISR International University
Cairo, Egypt
ashraf.raouf@miuegypt.edu.eg

***Abstract*—A common problem regarding combinatorial optimization in computer science, is the minimum partition problem. Given a set of integers, the mission is to divide it into two subsets where the absolute difference between their sums is minimum. In this study, two approaches for taking on the minimum partition problem, brute force and dynamic programming are compared. Although the brute force technique searches through all possible partitions, the dynamic programming approach uses a bottom-up approach to effectively build a solution to find the minimum difference. The method of comparison between these two approaches are time and space complexity, these ways of comparison enable us to measure and compare performance. [1]**

***Index Terms*—Minimum partition problem, dynamic programming, brute force**

I. Introduction

The minimum partition problem involves dividing a collection of integers into two subsets in which the difference between their sums is the minimum partition. Load balancing, scheduling and resource allocation are famous applications for this problem. Multiple ways of solving this have been presented, amongst these ways are dynamic programming and brute force. This paper intents to analyse and compare the performance of the two ways at hand in terms of their space and time complexity.

II. Dynamic Programming Approach

The problem can be solved using the dynamic programming approach, this can be achieved by breaking the problem down into smaller sub-problems using a bottom-up approach. This approach uses a table to store the sums and iteratively updates the table to represent each number's involvement in the set. [2]

A. Approach

This approach uses a Boolean table to track the possible sums that can be calculated with the subset of the given set. Hence, the table is updated iteratively as each element is taken into consideration.

B. The Algorithm

$$DP[i][j] = DP[i-1][j] \vee DP[i-1][j - arr[i-1]] \quad (1)$$

This equation uses the previously calculated values to determine whether a sum j can be achieved by including the current element $arr[i-1]$.

C. Methodology

The dynamic programming approach to solving the minimum partition problem involves the following steps: [3]

- Calculate the total sum of the given set.
- Use a 2D Boolean table to keep track of the possible sums achievable with subsets of the given set.
- Initialize the table for the base cases.
- Fill the table iteratively for each element in the set.
- Determine the minimum difference by checking the possible sums.

Listing 1. Dynamic Programming Approach

```
#include <iostream>
#include <chrono>
using namespace std;
using namespace std::chrono;

int stepsDP = 0;
int stepsBruteForce = 0;

int minPartitionDP(int* arr, int n) {
   int totalSum = 0;
   for (int i = 0; i < n; i++)
      totalSum += arr[i];

   int halfSum = totalSum / 2;

   bool **dp = new bool*[n + 1];
```

```
    for (int i = 0; i <= n; i++)
        dp[i] = new bool[halfSum + 1];

    for (int i = 0; i <= n; i++)
        dp[i][0] = true;

    for (int i = 1; i <= halfSum; i++)
        dp[0][i] = false;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= halfSum; j++) {
            stepsDP++;
            if (arr[i - 1] > j)
                dp[i][j] = dp[i - 1][j];
            else
                dp[i][j] = dp[i - 1][j] || dp[i - 1][j -
                    ↪ arr[i - 1]];
        }
    }

    int min = 0;
    for (int j = halfSum; j >= 0; j--) {
        if (dp[n][j]) {
            min = totalSum - 2 * j;
            stepsDP++;
            break;
        }
    }
    return min;
}
```

D. Complexity Analysis

The dynamic programming approach gave us in this problem a complexity of $O(n \times \text{totalSum})$, where n is the number of elements and totalSum is the total sum of the elements. The reason for this is that the DP table is updated for each element and each possible sum up to totalSum/2.

III. Brute Force Approach

Regarding the brute force approach, it exhaustively searches through all the possible partitions to find the partition with the minimum difference. It involves generating all the possible subsets and calculates the difference between the sums of the two subsets. [4]

A. Brute Force for Minimum Partition (Recursive Solution)

The brute force approach comprises of generating all possible partitions recursively and computing the difference between the sums of the two subsets for each partition. As a result, the complexity of the algorithm exponentially grows as the number of elements grow, this renders this approach unfeasible regarding larger datasets.

Listing 2. Brute Force Approach

```
int minPartitionBruteForce(int * arr, int idx, int
    ↪ sum1, int sum2, int n) {
    stepsBruteForce++;
    if (idx == n)
        return abs(sum1 - sum2);

    int diff1 = minPartitionBruteForce(arr, idx + 1,
        ↪ sum1 + arr[idx], sum2, n);
    int diff2 = minPartitionBruteForce(arr, idx + 1,
        ↪ sum1, sum2 + arr[idx], n);
    return min(diff1, diff2);
}


int main() {
    cout << "Please␣enter␣the␣size␣of␣the␣array:␣";
    int size;
    cin >> size;
    int* arr = new int[size];
    cout << "Please␣enter␣" << size << "␣numbers:␣";
    for (int i = 0; i < size; i++)
        cin >> arr[i];
    auto startDP = high_resolution_clock::now();
    int minDiffDP = minPartitionDP(arr, size);
    auto stopDP = high_resolution_clock::now();
    auto durationDP = duration_cast<microseconds>(
        ↪ stopDP - startDP);
    cout << "Minimum␣difference␣using␣Dynamic␣
        ↪ Programming:␣" << minDiffDP << endl;
    cout << "Steps␣taken␣using␣Dynamic␣Programming:␣"
        ↪  << stepsDP << endl;
    cout << "Time␣taken␣by␣Dynamic␣Programming␣
        ↪ approach:␣"
        << durationDP.count() << "␣microseconds" <<
            ↪ endl;

    cout << endl;

    auto startBF = high_resolution_clock::now();
    int minDiffBruteForce = minPartitionBruteForce(
        ↪ arr, 0, 0, 0, size);
    auto stopBF = high_resolution_clock::now();
    auto durationBF = duration_cast<microseconds>(
        ↪ stopBF - startBF);
    cout << "Minimum␣difference␣using␣Brute␣Force:␣"
        ↪ << minDiffBruteForce << endl;
    cout << "Steps␣taken␣using␣Brute␣Force:␣" <<
        ↪ stepsBruteForce << endl;
    cout << "Time␣taken␣by␣Brute␣force␣approach:␣"
        << durationBF.count() << "␣microseconds" <<
            ↪ endl;

    return 0;
}
```

B. Complexity Analysis

The time complexity of the brute force approach is $O(2^n)$ due to the exhaustive nature of exploring all possible partitions. Where n is the number of elements. This is an exponential complexity which is impractical in large datasets.

IV. Output

A. Abstract

When it comes down to using this practically, here are some example runs of the code to show the difference between the two approaches as the data increases.

B. 6 numbers

```
Minimum difference using Dynamic Programming: 1
Steps taken using Dynamic Programming: 37
Time taken by Dynamic Programming approach: 3 microseconds

Minimum difference using Brute Force: 1
Steps taken using Brute Force: 127
Time taken by Brute force approach: 13 microseconds
```

Fig. 1. Output of the previous code (6 numbers)

Regarding the previous figure, Fig. 1. It shows the output and compares the two approaches. The array used had (3, 1, 2, 1, 4, 2). In both approaches, the numbers 3 and 4 are in one subset, and the rest are in the second subset. Therefore, the minimum difference calculated is 1 as seen in the output. To compare, the brute force completed the calculations in 13 microseconds compared with the dynamic programming with 3 microseconds. As clearly seen, the dynamic programming approach is faster.

C. 17 numbers

```
Minimum difference using Dynamic Programming: 1
Steps taken using Dynamic Programming: 528854
Time taken by Dynamic Programming approach: 2923 microseconds

Minimum difference using Brute Force: 1
Steps taken using Brute Force: 262143
Time taken by Brute force approach: 5036 microseconds
```

Fig. 2. Output of the previous code (17 numbers)

Regarding the previous figure, Fig. 2. It shows the output and compares the two approaches. The array used had 17 integers. The brute force completed the calculations in approximately 0.26 seconds compared with the dynamic programming with approximately 0.003 seconds. As clearly seen, the dynamic programming approach is faster. This highlights the efficiency of the dynamic programming approach as the dataset gets larger.

D. 30 numbers

```
Minimum difference using Dynamic Programming: 1
Steps taken using Dynamic Programming: 1826221
Time taken by Dynamic Programming approach: 8071 microseconds

Minimum difference using Brute Force: 1
Steps taken using Brute Force: 2147483647
Time taken by Brute force approach: 40239242 microseconds
```

Fig. 3. Output of the previous code (30 numbers)

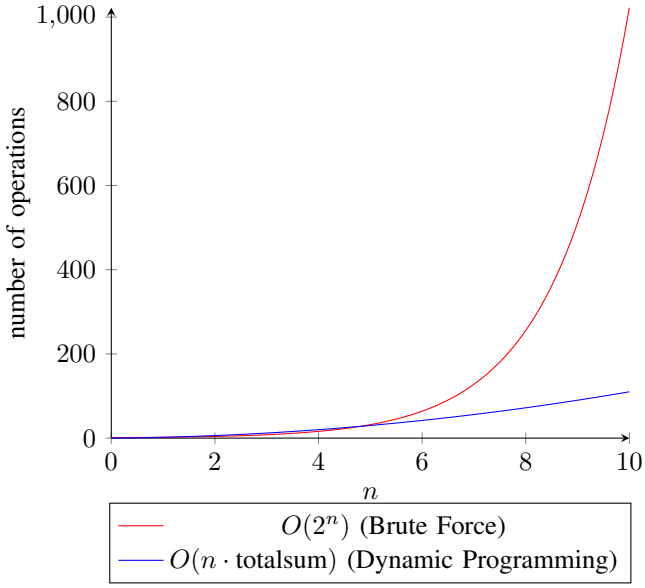
Regarding the previous figure, Fig. 3. It shows the output and compares the two approaches. The array used had 30 integers. The brute force completed the calculations in approximately 40.2 seconds compared with the dynamic programming with approximately 1.82 seconds. As clearly seen, the dynamic programming approach is extremely faster. The dynamic programming approach in this run is 183% faster than the brute force approach.

V. Performance Comparison

A polynomial-time solution $O(n \times \text{totalSum})$ is provided via dynamic programming, this gives us results with exceptionally less computational steps when being compared to the brute-force approach, which has exponential time complexity $O(2^n)$. As a result, this allows us to make a conclusion that dynamic-programming is more suitable for larger problem instances.

VI. Graphical Representation

To better illustrate the differences in complexity between the two approaches, consider the following graph:



VII. Conclusion

This study compares the dynamic programming and brute force approaches in solving the minimum partition problem. Regarding the superior approach (Dynamic programming), it is clearly demonstrated that it surpasses the brute force approach both in space and time complexity. Thus, making it the preferred method for large datasets. On the other hand, the brute force approach is easier and simpler to comprehend and understand. It is however, impractical for large sets of data due to the complexity $O(2^n)$ which is exponential unlike our other approach.

References

[1] "Minimum partition problem," https://www.geeksforgeeks.org/partition-a-set-into-two-subsets-such-that-the-difference-of-subset-sums-is-minimum/, accessed: 2024-05-30.

[2] R. Neapolitan, *Foundations of Algorithms*.

[3] Joy, "Dynamic programming approach," Stack Overflow, October 2020, uRL: https://stackoverflow.com/a/64373348.

[4] A. Levitin, *Introduction to the Design and Analysis of Algorithms*, 3rd ed. Addison-Wesley, 2011.