NP Complete Problem: Partition Problem

December 16, 2021

Charles Vincent A. Cordial • CMSC142(G)

Presentation Outline

- Problem Statement
- Algorithms used
    a. Recursive
    b. Dynamic Programming
- Analysis
- Demo in Java

Problem Statement

Problem Statement

- We are given a multiset of positive integers
  - e.g {3,1,1,2,2,1}
- **Goal:**
  - Find out if the set can be **divided into two subsets** such that the **sum of elements in both subsets is the same/equal.**

Problem Statement - Example

- Given: S = {3,1,1,2,2,1}
  - A = {3,1,1}
  - B = {2,2,1}

Sum of A = 3 + 1 + 1 = 5

Sum of B = 3 + 1 + 1 = 5

Hence, Sum of A = Sum of B.

The set can be equally partitioned.

Analogy

Pencil Problem

- **Problem 1**
  - We have 6 pencil available
  - Each of the pencil cost 1 peso
  - Give 3 pencils on each group.

- **Problem 2**
  - 7 pencil available
  - Impossible to distribute pencils equally to 2 groups.

Key points

1. The SUM has to be EVEN. (Base condition)
   a. If it is ODD then the set CANNOT be divided into two subsets with equal sum.

Problem Statement - Example

- Given: S = {3,1,1,2,2,1}

  => SUM = 10 (even)

  => TARGET = SUM/2 = 5

Passed the first condition.

SUBSET 1 SUM = 5

AND

SUBSET 2 SUM = 5

Key points

1. **The SUM of the set has to be EVEN.**
   a. **If it is ODD then the set CANNOT be divided into two subsets with equal sum.**

      If there is a subset of integers that sum up to SUM/2 then the remaining integers in the set will also sum to SUM/2.

      SUM/2 + SUM/2 = SUM

Algorithms

Algorithm: Recursive

Recursive - Visualization

S = {1,5,11,5}

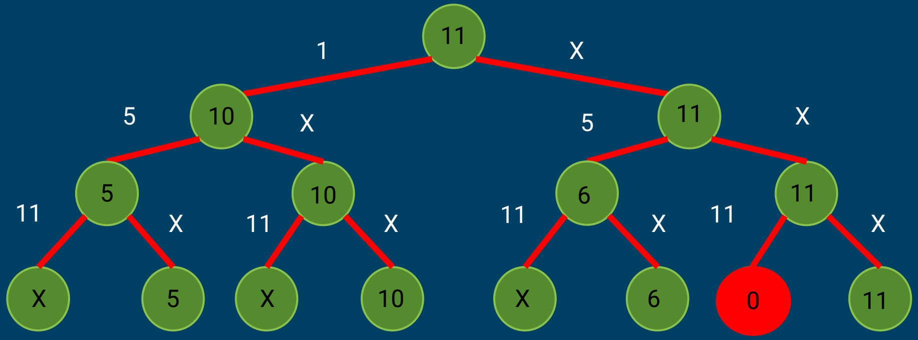
SUM = 1 + 5 + 11 + 5 = <u>22</u>

Since Sum is <u>even</u>, we proceed.

TARGET = SUM/2 = <u>11</u>

Target = 11

S = {1,5,11,5}



X - skip or 0

Black - SUM

White - Entry

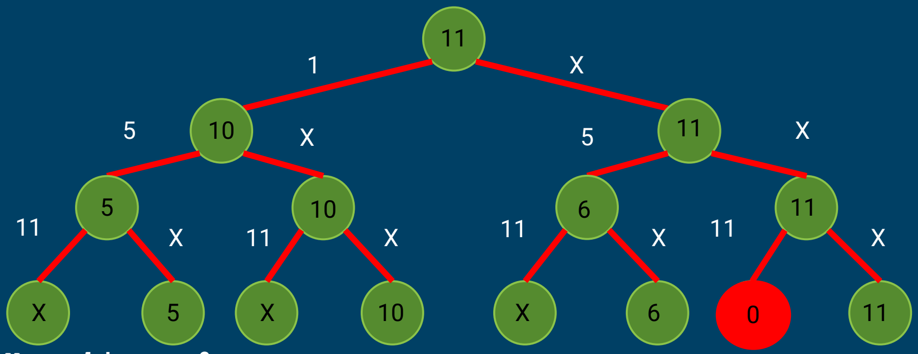
Recursive - Pseudocode

```
 static boolean findPartition(int arr[],
int n)
    {

        int sum = 0;
        for (int i = 0; i < n; i++)
             sum += arr[i];

        if (sum % 2 != 0)
             return false;

        return isSubsetSum(arr, n, sum / 2);
    }
```

```
static boolean isSubsetSum(int arr[], int n, int sum)
    {

        // Base Cases
        if (sum == 0)
            return true;
        if (n == 0 && sum != 0)
            return false

//case subtrahend is larger. skip
        if (arr[n - 1] > sum)
            return isSubsetSum(arr, n - 1, sum);

        return isSubsetSum(arr, n - 1, sum)||
isSubsetSum(arr, n - 1, sum - arr[n - 1]);

    }
```

Target = 11

S = {1,5,11,5}



X - skip or 0

Black - SUM

White - Entry

Brute Force Method

OUTPUT:

16 possibilities = 2^n

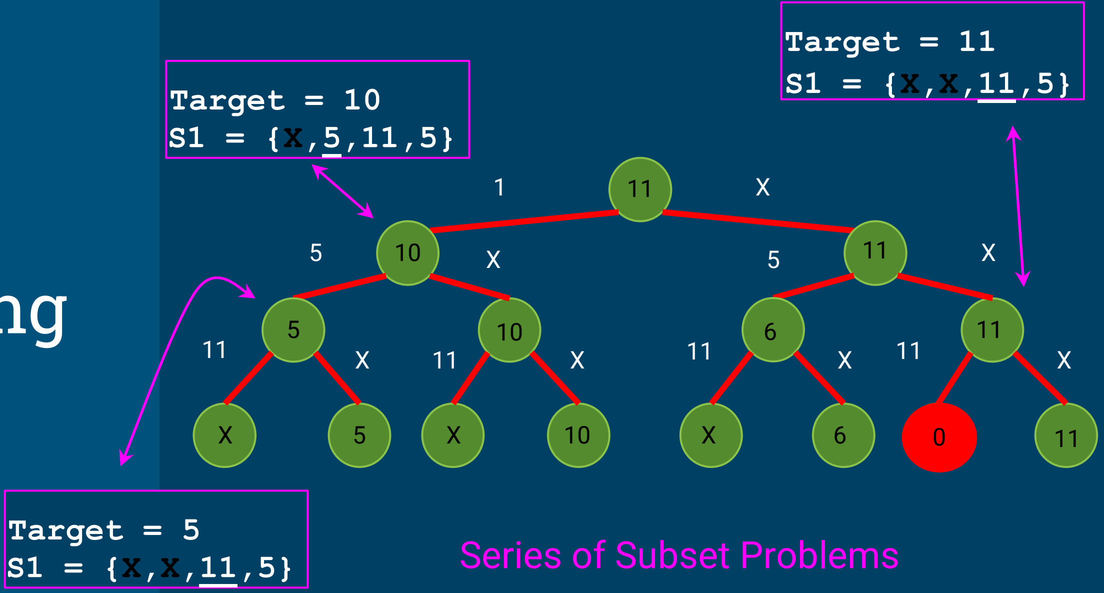
Exponential Time Complexity

Algorithm:
Dynamic Programming

Overview: Dynamic Programming

Target = 11

S = {1,5,11,5}



Series of Subset Problems

Reduction to “Subset Sum Problem (NP-Hard)”

DP- Visualization

arr = {3,2,1}

SUM = 3 + 2 + 1 = 6

Since Sum is even, we proceed.

TARGET = SUM/2 = 3

Target = 3
arr = {3,2,1}

ELEMENTS

SUM	{}	{3}	{3,2}	{3,2,1}
0	T	T	T	T
1	F			
2	F			
3	F			

DP- Visualization

```
static boolean findPartition(int arr[], int n)
    {
        int sum = 0;
        int i, j;

        // Calculate sum of all elements
        for (i = 0; i < n; i++)
            sum += arr[i];

        if (sum % 2 != 0)
            return false;

        boolean part[][] = new boolean[sum / 2 +
1][n + 1];

        // initialize top row as true
        for (i = 0; i <= n; i++)
            part[0][i] = true;

        // initialize leftmost column, except
part[0][0], as
        // 0
        for (i = 1; i <= sum / 2; i++)
            part[i][0] = false;
```

Target = 3 | Sum = 6
arr = {1,2,3}

ELEMENTS

SUM

	{}	{3}	{3,2}	{3,2,1}
0	T	T	T	T
1	F			
2	F			
3	F			

DP- Visualization

Target = 5 | Sum = 10
arr = {3,2,1}

```
// Fill the partition table in bottom up manner
       for (i = 1; i <= sum / 2; i++) { //i column, j
row
           for (j = 1; j <= n; j++) {
              part[i][j] = part[i][j - 1];
//(CHECKS IF SUM is LESS than ELEMENT)
              if (i >= arr[j - 1])
                 part[i][j]
                       = part[i][j]
                        || part[i - arr[j - 1]][j - 1];
//(go left and subtract element go upward)
            }
        }
              return part[sum / 2][n];
    }
```

ELEMENTS

SUM

	{}	{3}	{3,2}	{3,2,1}
0	T	T	T	T
1	F			
2	F		-subset explain	
3	F			

DP- Visualization

Target = 5 | Sum = 10
arr = {3,2,1}

```
// Fill the partition table in bottom up manner
      for (i = 1; i <= sum / 2; i++) { //i column, j row
          for (j = 1; j <= n; j++) {
             part[i][j] = part[i][j - 1];
//(CHECKS IF SUM is LESS than ELEMENT)

             if (i >= arr[j - 1])
         Part[i][j] = part[i][j] || part[i - arr[j -1]][j - 1];

//(go left and subtract element go upward)
            }
        }
              return part[sum / 2][n];
    }
```

ELEMENTS

SUM	{}	{3}	{3,2}	{3,2,1}
0	T	T	T	T
1	F	F	F	T
2	F	F	T	T
3	F	T	T	T

Dynamic Programming Method

=> O((Number of sum + 1) * number of elements in array)
=> O((sum/2) * n) => O(sum*n)
=> ~Polynomial Time Complexity

Dynamic Programming Method

=> O(sum*n)
=> If we increase the sum up to 2^n, the time becomes exponential
=> O(2^n * n)

Wrap up

Recursion

- Brute Force Algorithm
- Takes a lot of calculation
- O(2^n) time complexity
- Inefficient

Dynamic Programming

- Multidimensional Array (Saves data)
- Lesser calculation relative to Brute Force
- O(sum * n) time complexity.
- Exponentially large sum relative to array:
  - Sum = 2^n
  - Gives O(2^n * n) time complexity, same with recursion.

Key points

1. The SUM of the set has to be EVEN.
    a. If it is ODD then the set CANNOT be divided into two subsets with equal sum.

       If there is a subset of integers that sum up to SUM/2 then the remaining integers in the set will also sum to SUM/2
2. Brute Force can be used to solve problems with smaller values.

3. Dynamic Programming might be efficient with higher values (relative to brute force capacity) as long as the sum does not go near 2^n values (exponential sum).

Analysis

Recursion

- Brute Force Algorithm
- Takes a lot of calculation
- O(2^n) time complexity
- Inefficient

Dynamic Programming

- Multidimensional Array (Saves data)
- Lesser calculation relative to Brute Force
- O(sum * n) time complexity.
- Exponentially large sum relative to array:
  - Sum = 2^n
  - Gives O(2^n * n) time complexity, same with recursion.

This is why the “Partition Problem” is considered as NP-COMPLETE and not P.

END