Partition problem is to determine whether a given set can be partitioned into two subsets such that the sum of elements in both subsets is same.

Examples

```
arr[] = {1, 5, 11, 5}
Output: true
The array can be partitioned as {1, 5, 5} and {11}

arr[] = {1, 5, 3}
Output: false
The array cannot be partitioned into equal sum sets.
```

Following are the two main steps to solve this problem:
1) Calculate sum of the array. If sum is odd, there can not be two subsets with equal sum, so return false.
2) If sum of array elements is even, calculate sum/2 and find a subset of array with sum equal to sum/2.

The first step is simple. The second step is crucial, it can be solved either using recursion or Dynamic Programming.

Recursive Solution
Following is the recursive property of the second step mentioned above.

```
Let isSubsetSum(arr, n, sum/2) be the function that returns true if
there is a subset of arr[0..n-1] with sum equal to sum/2

The isSubsetSum problem can be divided into two subproblems
 a) isSubsetSum() without considering last element
    (reducing n to n-1)
 b) isSubsetSum considering the last element
    (reducing sum/2 by arr[n-1] and n to n-1)
If any of the above the above subproblems return true, then return true.
isSubsetSum (arr, n, sum/2) = isSubsetSum (arr, n-1, sum/2) ||
                              isSubsetSum (arr, n-1, sum/2 - arr[n-1])
```

```
// A recursive solution for partition problem
#include <stdio.h>

// A utility function that returns true if there is a subset of arr[]
// with sun equal to given sum
bool isSubsetSum (int arr[], int n, int sum)
{
   // Base Cases
   if (sum == 0)
     return true;
   if (n == 0 && sum != 0)
     return false;

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

   /* else, check if sum can be obtained by any of the following
      (a) including the last element
      (b) excluding the last element
   */
   return isSubsetSum (arr, n-1, sum) || isSubsetSum (arr, n-1, sum-arr[n-1]);
}
```

```
// Returns true if arr[] can be partitioned in two subsets of
// equal sum, otherwise false
bool findPartiion (int arr[], int n)
{
    // Calculate sum of the elements in array
    int sum = 0;
    for (int i = 0; i < n; i++)
       sum += arr[i];

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

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

// Driver program to test above function
int main()
{
  int arr[] = {3, 1, 5, 9, 12};
  int n = sizeof(arr)/sizeof(arr[0]);
  if (findPartiion(arr, n) == true)
     printf("Can be divided into two subsets of equal sum");
  else
     printf("Can not be divided into two subsets of equal sum");
  getchar();
  return 0;
}
```

Output:

```
Can be divided into two subsets of equal sum
```

Time Complexity: O(2^n) In worst case, this solution tries two possibilities (whether to include or exclude) for every element.

Dynamic Programming Solution

The problem can be solved using dynamic programming when the sum of the elements is not too big. We can create a 2D array part[][] of size (sum/2)*(n+1). And we can construct the solution in bottom up manner such that every filled entry has following property

```
part[i][j] = true if a subset of {arr[0], arr[1], ..arr[j-1]} has sum
             equal to i, otherwise false
```

```
// A Dynamic Programming solution to partition problem
#include <stdio.h>

// Returns true if arr[] can be partitioned in two subsets of
// equal sum, otherwise false
bool findPartiion (int arr[], int n)
{
    int sum = 0;
    int i, j;

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

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

    bool part[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;

     // Fill the partition table in botton up manner
     for (i = 1; i <= sum/2; i++)
     {
       for (j = 1; j <= n; j++)
       {
         part[i][j] = part[i][j-1];
         if (i >= arr[j-1])
           part[i][j] = part[i][j] || part[i - arr[j-1]][j-1];
       }
     }

    /* // uncomment this part to print table
     for (i = 0; i <= sum/2; i++)
     {
       for (j = 0; j <= n; j++)
          printf ("%4d", part[i][j]);
       printf("\n");
     } */

     return part[sum/2][n];
}

// Driver program to test above funtion
int main()
{
  int arr[] = {3, 1, 1, 2, 2, 1};
  int n = sizeof(arr)/sizeof(arr[0]);
  if (findPartiion(arr, n) == true)
     printf("Can be divided into two subsets of equal sum");
  else
     printf("Can not be divided into two subsets of equal sum");
  getchar();
  return 0;
}
```

Output:

```
Can be divided into two subsets of equal sum
```

Following diagram shows the values in partition table. The diagram is taken form the wiki page of partition problem.

The entry part[i][j] indicates whether there is a subset of {arr[0], arr[1], .. arr[j-1]} that sums to i

	{}	{3}	{3,1}	{3,1,1}	{3,1,1,2}	{3,1,1,2,2}	{3,1,1,2,2,1}
0	True	True	True	True	True	True	True
1	False	False	True	True	True	True	True
2	False	False	False	True	True	True	True
3	False	True	True	True	True	True	True
4	False	False	True	True	True	True	True
5	False	False	False	True	True	True	True

Dynamic Programming table for

arr[] ={3,1,1,2,2,1}

Time Complexity: O(sum*n)
Auxiliary Space: O(sum*n)
Please note that this solution will not be feasible for arrays with big sum.

References:
http://en.wikipedia.org/wiki/Partition_problem