1. **Given a *m* x *n* grid filled with non-negative numbers, find a path from top left position to bottom right position such that the sum of all numbers along its path is minimum.**

   Note: You can only move down or right.

   Example:
   Input:
   [[1, 2, 5, 4, 10],
   [8, 9, 6, 2, 1],
   [5, 5, 7, 11, 1],
   [6, 4, 5, 9, 2],
   [9, 8, 4, 9, 3]]

   Output: 21

   Path is:
   [1, 2, 5, 4, 10]
   [8, 9, 6, 2, 1]
   [5, 5, 7, 11, 1]
   [6, 4, 5, 9, 2]
   [9, 8, 4, 9, 3]

   Solution:-

   a. **what is the recursive relationship between problem and its subproblems.**

      minCost(m, n) = min (minCost(m-1, n), minCost(m, n-1)) + cost[m][n]

      In the above function, the both sub problem minCost(m-1,n) & minCost(m,n-1) needs to compute again and again and there will be some overlap sub problems, in order to get final solution of given problem. Recomputation of same subproblem (overlaping subproblem) can be avoided by constructing temporary array tc[][] in bottom up manner.

   b. **does this problem have optimal substructure property? explain the reason**

      Yes, given problem have optimal substructure property.

      The path to reach (m, n) must be through one of the 2 cells: (m-1, n) or (m, n-1). So minimum cost to reach (m, n) can be written as “minimum of the 2 cells plus cost[m][n]”.

      minCost(m, n) = min (minCost(m-1, n), minCost(m, n-1)) + cost[m][n]

      Therefore, we can compute both subproblems minCost(m-1, n) & minCost(m, n-1) individually and can find minimum out of that 2 at the end to achieve final solution of the given problem.

c. design a bottom up algorithm and implement the problem (use your favorite language to write a program so that it can run and produce the result, use the example above as your test case)

```
import java.util.*;
import java.math.*;
class MinimumCostPath
{
	public static void main(String args[])
	{
		int cost[][]= {{1, 2, 5, 4, 10},
				{4, 9, 6, 2, 1},
				{5, 5, 7, 11,1},
				{6, 4, 5, 9, 2};
				{9, 8, 4, 9, 3}
				};
	System.out.println(minCost(cost,4,4));
   }

   private static int minCost(int cost[][], int m, int n)
   {
	int i, j;
	int tc[][]=new int[m+1][n+1];
	tc[0][0] = cost[0][0];
	for (i = 1; i <= m; i++)
	{
-		tc[i][0] = tc[i-1][0] + cost[i][0];
	}
```

```
        for (j = 1; j <= n; j++)
        {
                tc[0][j] = tc[0][j-1] + cost[0][j];
        }

        for (i = 1; i <= m; i++)
        {
                for (j = 1; j <= n; j++)
                {
                        tc[i][j] = Math.min(tc[i-1][j],tc[i][j-1])+cost[i][j];
                }
         }
         return tc[m][n];
   }
}
```

2. **Use dynamic programming technique to design and implement an algorithm to check a list of positive integers to see if the list can be partitioned into two subsets such that the sums of numbers in the two subsets are equal.**
   You program only need to return true or false.

For example.
given: {4, 6, 11, 7, 1, 5}
can be partitioned into: {4, 7, 5, 1} and {6, 11}, both sum = 17
the program should return true.
given: {1, 2, 4}
the program should return false.

you should:
1. Explain the problem has optimal substructure property
2. Find out the recursive equation (doesn't has to be same form as in the lecture)
3. Design the bottom up algorithm and implement it.
Draw the table for the example: {1, 2, 1, 4}
(hint: given set S, you can check if there is a subset Sk that SUM(Sk) == SUM(S)/2)

Solution:-

We can say that the problem has optimal substructure property because the given problem can be divided into subproblems to achieve optimal solutions of subproblems.

We can divide this given problem into 2 subproblems like,

(1) Sum of array list is divided by 2 or not.
(2) subset of {arr[0], arr[1],…,a[j-1]} has sum equal to I or not.

By solving this 2 subproblems individually we can get final solution of given problem.

Recursive equation for given problem can be given as,

part[i][j] = part [i][j] , i<a[j-1]

part[i][j] = part [i][j] || part [i-a[j-1]][j-1] , i>=a[j-1]

```
import java.io.*;

class Partition
{

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

	{

		int sum = 0;
		int i, j;

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

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

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

		for (i = 0; i <= n; i++)
		{
			part[0][i] = true;
		}

		for (i = 1; i <= sum/2; i++)
		{
			part[i][0] = false;
		}

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

```
    }
    public static void main (String[] args)
    {
        int arr[] = {1,2,1,4};
        int n = arr.length;
        if (findPartition(arr, n) == true)
        {
            System.out.println("Can be divided into two subsets of equal sum");
        }
        else
        {
            System.out.println("Can not be divided into two subsets of equal sum");
        }
    }
}
```

Table for above example is as per below,

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