Solution Approach

The challenge was to create a solution to the package problem described in the readme file, to be used as a **cross platform library**.
I have used .NET5 as the development platform because.

- It supports cross platform.
- It includes .net core runtime.
- It has a lot of improvements over the .NET core framework.
- .NET6 version is fairly new and .NET5 is currently supported.
- C# 9 support

The solution was divided into three projects. As the requirement suggests, a class library project was needed as the solution to be run on across platforms. The three projects are as follows.

1. **Com.mobiquity.packer** - Class library project
2. **Com.mobiquity** - Console application to act as the presentation layer
3. **Com.mobiquity.packer.tests** - Unit Test project (following TDD)

Usage of Test driven development

I followed the Test driven development(TDD) approach to develop the solution. It is a programming practice where the test cases are written first covering every functionality. And then the functionality is developed to make the test cases pass while taking small programming steps at a time. The three phases are as follows,

1. Based on the requirements specified in the documents, the automated test cases are written
2. Tests are executed, and in some cases, they fail as they are developed before the development of an actual feature
3. Refactor the code for the test to pass successfully

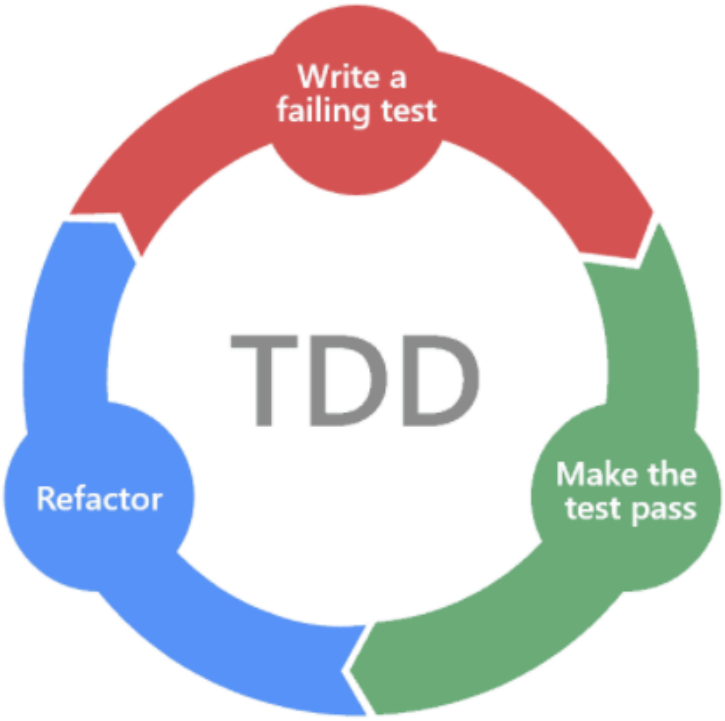

Image: https://marsner.com/

I created the test project Using XUnit project template and created the packerTest class to test the output of the pack method in the class library. And refactored the class library method to return a string.

```
public class PackerTests
{
    [Theory]
    [InlineData("example_input")]
    0 references
    public void PackerShouldReturnString(string fileName)
    {
        var returnValue = Packer.Pack(Path.Combine(Environment.CurrentDirectory, "Resources", fileName));

        Assert.True(typeof(string) == returnValue.GetType());
    }
```

It was the same pattern I continued throughout the solution.

After going through the requirements, I divided the class library solution into 4 parts.

1. Read the specified file
2. Map the file contents to actual objects
3. Optimize the package
4. Print or respond as required

I created interfaces to map to the above implementation adhering to **SOLID principles**. Below are the interfaces created

1. IFileService
2. IPackageService
3. IWrappingService
4. IResponseService

The file service tests were written and then the **FileService.cs** concrete implementation was created to read the file inheriting **IFileService** contract. These implementations were done with single responsibility and dependency inversion principle in mind. And it will be easier to plug in a different implementation of the file service later if that is required. Following the same pattern, the other interfaces were implemented using concrete classes.

Algorithms

There were two ways to solve the problem and to create the optimum package with the given requirements.

1. **Recursive solution** where we can loop through all the possible scenarios with the given number of package items
2. Using **Dynamic programming** where we can keep track of overlapping sets and by constructing a temporary array to store it using bottom up manner.

I selected the dynamic programming solution because of the time complexity. The recursive solution takes **2^N** times to solve the problem, but using dynamic programming we can reduce that to **N*W** (with N being the number of packageItems and W being the maximum weight). Even though the dynamic programming solution takes a bit more space while executing, the time complexity is much more important for the solution.

For example,
If we take 4 items with below details,

Item1
Weight 8
Price 50

Item2
Weight 2
Price 150

Item3
Weight 6
Price 210

Item4
Weight 1
Price 30

We can form the below matrix to get the optimal cost of a package and map this problem to the knapsack problem.

		Maximum Weight										
		0	1	2	3	4	5	6	7	8	9	10
Items	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	0	0	0	0	0	0	0	50	50	50
	2	0	0	150	150	150	150	150	150	150	150	200
	3	0	0	150	150	150	150	210	210	360	360	360
	4	0	30	150	180	180	180	210	240	360	390	390

Therefore 390 can be taken as the optimal price of this package. And a two dimensional array is created to hold the value of the maximum weight and items. After the optimal price is known, we loop through the dataset to get the items to be included in the package.

Class library solution structure

The class library was refactored and the solution was divided into three main namespaces.

1. com.mobiquity.packer.Models namespace is to hold the two models (package, packagesItem)
2. com.mobiquity.packer.Services namespace is to hold the contracts and implementations of the four services (file, package, wrapping and response)
3. com.mobiquity.packer.Utilities namespace was created to hold utility classes of the solution such as, constants.

Note: **APIException** class was created to extend the *System.Exception* and to handle library level constraints and exceptions.