I. Internal Transfer of Order

1. Problem Background:

For each market, Kaiser's Automated Trading System (ATS) maintains trading orders from different auto-trading strategies. Occasionally, different strategies may result in offsetting orders, which are orders that share the same price but have quantities summing up to 0. For example, strategy A may have an order to buy 10 contracts in the Gold future market while strategy B and C each has an order to sell 5 contracts in the same market with the same price.

In such a case, the ATS will generate internal transfer trades (fills) as 1 buy fill of +10 and 2 sell fills of -5 each, and allocate the fills to the corresponding strategy. By doing this, the ATS saves the cost of physicaly executing the orders at the exchange. It would be ideal to have an algorithm to identify the potential offsettings given a collection of orders of the same price.

2. Task:

Given a collection of `Order`'s defined in such a structure

```
class Order
{
    int     Id;
    int     Quantity;
    decimal Price;
}
```

and assuming they share the same price, implement the function in following class

```
class InternalTransfer
{
    List<int> Collect(List<Order> orders)
    {
        // your code here
    }
}
```

, such that it returns **largest** set of offsetting orders. If a tie exists, return any one.

You may write your solution in any language, preferably C#. Apart from the solution code, you should also provide unit test methods for us to verify your work. *You don't have to test your solution on very large inputs for this problem.*

Please also analyze the time and space complexity of your algorithm in terms of Big-O notation.

3. Examples:

Example 1:

- Input: [Order(1, 10, 100); Order(2, -5, 100); Order(3, -5, 100); Order(4, 5, 100); Order(5, -5, 100); Order(6, 2, 100);]
- Output (The `Id` of orders): [1,2,3,4,5];

Example 2:

- Input: [Order(1, 10, 100); Order(2, -5, 100);]
- Output: [];

Example 3:

- Input: [Order(1, 10, 100); Order(2, -3, 100); Order(3, -5, 100); Order(4, 5, 100); Order(5, -2, 100);]
- Output: [1,2,3,5];

Example 4:

- Input: [Order(1, 10, 100); Order(2, -10, 200);]
- Output (): []

N.B. A notation of `Order(x,y,z)` means an `Order` object with `Id == a`, `Quantity == b` and `Price == c`.

II. Minimum Over-cancellation

1. Problem Background

Kaiser's Automated Trading System (ATS) has a list of candidate orders with the same price and same direction (buy/sell) but different quantities (e.g. all orders are to buy at $100, but order A has a quantity of 9, order B has a quantity of 3 etc.).

The ATS also has a target quantity to cancel, which can be denoted by the integer value X. It is ideal to have an algorithm that, given a list of candidate orders and a cancel target X, find a collection of orders whose quantities sum to X, or sum to a quantity which exceeds X by as little as possible (minimum over-cancel).

2. Task:

Given a collection of `Order`'s defined in such a structure

```
class Order
{
    int     Id;
    int     Quantity;
    decimal Price;
}
```

and assuming they share the same price and direction (buy/sell), implement the function in following class

```
class MinOverCancellation
{
    List<int> Collect(List<Order> orders, int cancelTarget)
    {
        // your code here
    }
}
```

, such that it returns set of orders with sum of quantity overceeds `cancelTarget` as litter as possible. If a tie exists, return any one.

You may write your solution in any language, preferably C#. Apart from the solution code, you should also provide unit test methods (and better test cases) for us to verify your work.

Please also analyze the time and space complexity of your algorithm in terms of Big-O notation and address how your algorithm would perform given very large inputs. E.g. large number of orders.

3. Example:

Example 1:

- Input: [Order(1,3,100); Order(2,2,100); Order(3,3,100); Order(4,1,100); Order(5,5,100);], 2
- Output (The `Id` of orders): [2]

Example 2:

- Input: [Order(1,3,100); Order(2,2,100); Order(3,3,100); Order(4,1,100); Order(5,5,100);], 7
- Output (The `Id` of orders): [2,5]

Example 3:

- Input: [Order(1,5,100); Order(2,2,100); Order(3,7,100); Order(4,2,100);], 3
- Output: [2,4]

Example 4:

- Input: [Order(1,5,100); Order(2,5,100); Order(3,5,100); Order(4,6,100); Order(5,15,100); Order(6,55,100);], 12
- Output: [5]

N.B. An `Order(x,y,z)` means an `Order` object with `Id == a`, `Quantity == b` and `Price == c`.