1. Design a θ(n log(n)) time algorithm that given a set S of n integer numbers and another integer x, determines whether or not there exist 2 elements in S sum is exactly x

Approach:

1. Sort the array using merge-sort;
2. Use 2 pointers “low” and “high”, assign to the first and the last element of the array.
3. Scan through the array”
   - If the element at “low” + element at “high” is the wanted number (x) then return true.
   - Else if the element at “low” + element at “high” is lower than x, then shift the “low” pointer to the right (+1).
   - Else if the element at “low” + element at “high” is larger than x, then shift the “high” pointer to the left (-1).
4. If after scan through the array but there are no pair of elements are eligible, then return false.

This operation will have the average time complexity of θ(n log(n)).

```
public bool FindTwoIntForSum(int[] inputArray, int sum)
{
    int left = 0;
    int right = inputArray.Length - 1;

    //Sort the array using mergesort algorithm,
    //Mergesort will run with the complexity of Theta(n.Log(n))
    MergeSort(inputArray, Comparer<K>.Default);

    /*
    Scan the array,
    left is the left most index (0)
    right is right most index (length-1)
    if element in left most + rightmost is the wanted number return true
    if < the wanted number then increase left by 1
    if > the wanted number then decrease right by 1
    if can not find the 2 numbers, then return false

    This will run with the complexity of O(n)
     */
    while (left < right)
    {
        if (inputArray[left] + inputArray[right] == sum)
            return true;
        else if (inputArray[left] + inputArray[right] < sum)
            left++;
        else
            right--;
    }
    return false;
}
```

2. A stack data structure that provide Push and Pop, provide a Min operation that return the current smallest element of the stack. With constant time complexity for all three operations.

Assume that a stack interface will have these methods:

```
public interface Stack<K>
{
    void Push(K input);
    K Pop();
    K Peek();
}
```

And a “Stack” class has implemented this interface, along with properties such as “IsEmpty”, etc.

For a stack data structure that fits the requirement, 2 different stacks object will be required:

```
    //The main stack
    private Stack<T> _stack;
    //Auxiliary stack for minimum element storing
    private Stack<T> _auxStack;
```

The Push method needs to push the desired item to the main stacks, and also push to the auxiliary one if the latest element of it is no less than the new element:

```
    public void Push(T x)
    {
        //Push the item to stack
        _stack.Push(x);

        //Also push to the Auxiliary stack if it is empty
        if (_auxStack.IsEmpty)
        {
            _auxStack.Push(x);
        }
        else
        {
            if (_auxStack.Peek() >= x)
            {
                _auxStack.Push(x);
            }
        }
    }
```

The Pop method will make sure the stack is not empty, then remove the latest element in the stack, if it is equal to the latest in the auxiliary, that element will also be removed from the auxiliary:

```
public T Pop()
{
    if (_stack.IsEmpty)
    {
        throw new InvalidOperationException("Stack Underflow!");
    }

    //Remove the top element
    T theTop = _stack.Pop();

    //Peak the Auxiliary stack,
    //if it is equal to that top element then remove it also
    if (theTop == _auxStack.Peek())
    {
        _auxStack.Pop();
    }

    //Return the top element
    return theTop;
}
```

And finally, the Minimum property, this will make sure there will be something in the Auxiliary stack, then just peek the Auxiliary:

```
public int Minimum
{
    get
    {
        if (_auxStack.IsEmpty)
        {
            throw new InvalidOperationException("Stack Underflow!");
        }
        return _auxStack.Peek();
    }
}
```

These operations will run with constant time complexity.

```
namespace solution
{
    public class FindTwoInt
    {
        public bool FindTwoIntForSum(int[] inputArray, int sum)
        {
            int left = 0;
            int right = inputArray.Length - 1;

            //Sort the array using mergesort algorithm,
            //Mergesort will run with the complexity of Theta(n.Log(n))
            MergeSort(inputArray, Comparer<K>.Default);

            /*
            Scan the array,
            left is the left most index (0)
            right is right most index (length-1)
            if element in left most + rightmost is the wanted number return true
            if < the wanted number then increase left by 1
            if > the wanted number then decrease right by 1
            if can not find the 2 numbers, then return false

            This will run with the complexity of Theta(n)
             */
            while (left < right)
            {
                if (inputArray[left] + inputArray[right] == sum)
                    return true;
                else if (inputArray[left] + inputArray[right] < sum)
                    left++;
                else
                    right--;
            }
            return false;
        }
        private void MergeSort<K>(K[] sequence, IComparer<K> comparer) where K :
IComparable<K>
        {
            int n = sequence.Length;
            //when array length is 1, it is sorted
            if (n < 2) return;

            //chop chop
            int mid = n / 2;
            K[] s1 = sequence.Take(mid).ToArray();
            K[] s2 = sequence.Skip(mid).ToArray();

            //conquer, sort the 2 sequences recursively
            MergeSort(s1, comparer);
            MergeSort(s2, comparer);
```

```
            //Merge result to original
            Merge(s1, s2, sequence, comparer);
        }
        private void Merge<K>(K[] s1, K[] s2, K[] s, IComparer<K> comparer) where
K : IComparable<K>
        {
            int i = 0, j = 0;
            while (i + j < s.Length)
            {
                if (j == s2.Length || (i < s1.Length && comparer.Compare(s1[i],
s2[j]) < 0))
                    s[i + j] = s1[i++];
                else
                    s[i + j] = s2[j++];
            }
        }
    }

    public class DummyStack<T>
    {
        //The main stack
        private Stack<T> _stack;
        //Auxiliary stack for minimum element storing
        private Stack<T> _auxStack;

        //Constructor
        public DummyStack()
        {
            _stack = new Stack();
            _auxStack = new Stack();
        }

        public int Minimum
        {
            get
            {
                if (_auxStack.IsEmpty)
                {
                    throw new InvalidOperationException("Stack Underflow!");
                }
                return _auxStack.Peek();
            }
        }

        public void Push(T x)
        {
            //Push the item to stack
            _stack.Push(x);

            //Also push to the Auxiliary stack if it is empty
            if (_auxStack.IsEmpty)
```

```
            {
                _auxStack.Push(x);
            }
            else
            {
                if (_auxStack.Peek() >= x)
                {
                    _auxStack.Push(x);
                }
            }
        }

        public T Pop()
        {
            if (_stack.IsEmpty)
            {
                throw new InvalidOperationException("Stack Underflow!");
            }

            //Remove the top element
            T theTop = _stack.Pop();

            //Peak the Auxiliary stack,
            //if it is equal to that top element then remove it also
            if (theTop == _auxStack.Peek())
            {
                _auxStack.Pop();
            }

            //Return the top element
            return theTop;
        }
    }

    public interface Stack<K>
    {
        void Push(K input);
        K Pop();
        K Peek();
    }
}
```