1. Write a function that finds the largest and smallest numbers in a list.

Solution with a complexity of O(n):

```
public class MinMaxResult

{
    public int Min { get; set; }
    public int Max { get; set; }
}

public MinMaxResult FindMinMax(List<int> numbers)
{
    if (numbers.Count == 0)
    {
        throw new ArgumentException("The list cannot be empty",
    nameof(numbers));
    }

    int min = numbers.First();
    int max = numbers.First();

    foreach (int number in numbers)
    {
        if (number > max)
        {
            max = number;
        }

        if (number < min)
        {
            min = number;
        }
    }
    return new MinMaxResult { Min = min, Max = max };
}
```

NodeJs:

```
interface MinMaxResult {
  min: number;
  max: number;
}

function findMinMax(numbers: number[]): MinMaxResult {
  if (!numbers || numbers?.length === 0) {
    throw new Error('The list cannot be empty');
  }

  let min = numbers[0];
  let max = numbers[0];

  numbers.forEach((number: number) => {
    if (number > max) {
      max = number;
    }

    if (number < min) {
      min = number;
    }
  });
  return { min, max };
}
```

2. Write a function that removes duplicate characters from string. Provide at least 3 solutions. Which is best in your opinion? Why?

Solution 1: Using LINQ

```
public string RemoveDuplicates(string str)
{
    if (str.Length == 0) {
      return str;
    }

    return new string(str.Distinct().ToArray());
}
```

NodeJS:
There is no LINQ in nodejs, so we'll use lodash "uniq" for demonstration:

```
function removeDuplicates(str: string): string {
  return _.uniq(str);
}
```

or javascript native Set()

```
function removeDuplicates(str: string): string {
  return [...new Set(str)].join('');
}
```

Solution 2: Using a HashSet

```
public string RemoveDuplicates(string str)
{
    if (str.Length == 0) {
      return str;
    }
    HashSet<char> uniqueChars = new HashSet<char>();
    StringBuilder sb = new StringBuilder();

    foreach (char c in str)
    {
        if (!uniqueChars.Contains(c))
        {
            uniqueChars.Add(c);
            sb.Append(c);
        }
    }

    return sb.ToString();
}
```

NodeJS:

```
function removeDuplicates(str: string): string {
  if (str?.length === 0) {
    return str;
  }

  const uniqueChars: Set<string> = new Set();
  let uniqueString: string = '';

  for (let i = 0; i < str.length; i++) {
    if (!uniqueChars.has(str[i])) {
      uniqueChars.add(str[i]);
      uniqueString += str[i];
    }
  }
  return uniqueString;
}
```

Solution 3: Using sorting:

```
string RemoveDuplicates(string str)
{
    if (str.Length == 0) {
      return str;
    }

    char[] arr = str.ToCharArray();
    Array.Sort(arr);
    StringBuilder sb = new StringBuilder();

    for (int i = 0; i < arr.Length - 1; i++)
    {
        if (arr[i] != arr[i + 1])
        {
            sb.Append(arr[i]);
        }
    }

    sb.Append(arr[arr.Length - 1]);
    return sb.ToString();
}
```

NodeJS:

```
function removeDuplicates(str: string): string {
  if (str?.length == 0) {
    return str;
  }

  const arr: string[] = str.split('').sort();
  let uniqueString: string = '';

  for (let i = 0; i < arr.length - 1; i++) {
    if (arr[i] != arr[i + 1]) {
      uniqueString += arr[i];
    }
  }

  uniqueString += str[str.length - 1]
  return uniqueString
}
```

Conclusion:

The best solutions are 1 and 2, depending on the usecase.

If there are no performance considerations, solution 1 is the best because it's straight forward and easy to maintain, basically delegates all the computation to LINQ.

If there are performance constraints in terms of speed, then solution no. 2 is our favorite with a complexity of O(n). In terms of memory, it's not ideal, because we must store hash set in memory.

Bonus: Solution 2 without HashSet:

```
string RemoveDuplicates(string str)
{
    StringBuilder sb = new StringBuilder();

    foreach (char c in str)
    {
        if (sb.ToString().IndexOf(c) == -1)
        {
            sb.Append(c);
        }
    }

    return sb.ToString();
}
```

This is an alternative implementation without HashSet. This is better in terms of memory, but worst in speed, because it requires extra computation on sb.ToString() and IndexOf() (not the same with HashSet.Contains(), which is O(1)).

As a conclusion, I will go with solution no. 2 as my favorite for C# and for NodeJS I will go with solution no. 1.

3. Write a function that checks if two strings are Anagram

Solution with a complexity of O(n) using a dictionary to keep chars occurrences in both strings.

```
    bool isAnagram(String str1, String str2)
    {
        if (str1.Length != str2.Length)
        {
            return false;
        }
        Dictionary<char, int> dict
            = new Dictionary<char, int>();

        // loop through the first string and track each occurence of a character
by increasing its count.
        for (int i = 0; i < str1.Length; i++)
        {
            if (dict.ContainsKey(str1[i]))
            {
                dict[str1[i]] = dict[str1[i]] + 1;
            }
            else
            {
                dict.Add(str1[i], 1);
            }
        }

        // loop through the second string and subtract each occurence of a
character from previously populated dictionary with string one.
        for (int i = 0; i < str2.Length; i++)
        {
            if (dict.ContainsKey(str2[i]))
            {
                dict[str2[i]] = dict[str2[i]] - 1;
            }
            else
            {
                return false;
            }
        }

        var keys = dict.Keys;

        // check if all character occurences quals 0. If not, then the two
strings are not anagram.
        foreach (char key in keys)
        {
            if (dict[key] != 0)
            {
                return false;
            }
        }

        return true;
    }
```

This can also be achieved by sorting the strings then compare them. It’s not as efficient as the above solution, but I’ll demonstrate it in NodeJS:

```
function isAnagram(str1: string, str2: string): boolean {
  if (str1?.length !== str2?.length) {
    return false;
  }
  const sorted1: string = str1.split('').sort().join('');
  const sorted2: string = str2.split('').sort().join('');

  return sorted1 === sorted2;
}
```

For simplicity, in both examples I assumed the strings are case sensitive, so the algorithms will check for a case sensitive anagram.

4. Write a RegEx to match an Australian mobile phone

@"^(?:\+61|0)[2-478](?:[-]?[0-9]){8}$"