C# algorithms

1> Recursion

Besides calling other methods, a method can also call *itself*. This is called *recursion.*

The mechanics of a method calling itself are exactly the same as if it had called another, different method. A new stack frame is pushed onto the stack for each call to the method.
For example, in the following code, method Count calls itself with one less than its input parameter and then prints out its input parameter. As the recursion gets deeper, the stack gets larger.

```
class Program

    {

        static int fact(int value)

        {

            if (value <=1)

                return value;

         else

           return value* fact(value-1);

        }


        static void Main()

        {

            int y= fact(3);

            Console.WriteLine(y);


        }
```

2> Fisher-Yates Shuffle

To know first: **Static modifer:** Whenever you write a function or declare a variable, it doesn't create instance in a memory until you create object of class. But if you declare any function or variable with static modifier, it directly create instance in a memory and acts globally. The static modifier doesn't reference with any object.

```
 class Program

    {

      static Random rand = new Random();


       public static void suffle<t>(t[] array)

        {

            var random = rand;


            for (int i = array.Length; i >1; i--)

            {

                var o = rand.Next(i);

                var temp = array[o];

                array[o] = array[i - 1];

                array[i - 1] = temp;


            }

        }

        static void Main(string[] args)

        {

            int[] fr = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };


            Program.suffle(fr);


            foreach (var item in fr)


            {
                Console.WriteLine(item);
            }
        }
    }
```

3> Paliandrome

```
 public  class pal
    {
        public void reverseray(string take)
        {
          bool t = true;

          var te = take.Length;
          var y = 0;
            for (int i =te; i>0; i= i-1)
            {
                if (take[i-1] != take[y])
                {
                    t = false;
                    break;
                }

                y++;

            }

            if (t == true)
```

```
                Console.WriteLine(" A Paliandrome !!");


            else

                Console.WriteLine("NOT a paliandrome");

         }

}


    class Program

    {

        static void Main(string[] args)


        {

            pal pal = new pal();

                  pal.reverseray("soppo");


        }

    }
```

3b> To find if a sentence is paliandromic or not

```
public  class pal
    {


        public void reverseray(string take)
        {
          bool t = true;
          string[] th = take.Split(' ');


        var y = 0;
            for (int i =th.Length; i>0; i= i-1)
            {


                if (th[i-1] != th[y])
                {
                    t = false;
                    break;
                }


                y++;

            }

            if (t == true)
                Console.WriteLine(" A Paliandrome !!");

            else
                Console.WriteLine("NOT a paliandrome");


         }
}

    class Program
    {
        static void Main(string[] args)

        {
            pal pal = new pal();


             pal.reverseray("oppo oo oppo");

        }
    }
```

4> Fibonacci Series: 0 1 1 2 3 5 8 13 21.....

```
public static class fib

    {

        public static void fibon(int v)

        {

            for (int j = 0; j< v; j++){

            int a=0, b=1;

            for (int i = 0; i < j; i++)

            {
                int t = a;
                a = b;
                b = a + t;

            }
                Console.WriteLine(a);
        }
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            fib.fibon(15);
        }
    }
```

5> Slice an array

```
 public static class sliceray
  {
      public static t[] sliceit<t>(t[] source, int start, int end)
      {
          int len=  end-start;
          t[] buf = new t[len];
          for(int i= 0; i<len; i++)
          {
              buf[i] = source[i + start];
          }
```

```
                return buf;
            }
        }

    class Program
    {
        static void Main(string[] args)
        {
            string[] ty = new string[] { "w", "a", "s", "S", "u", "p" };

            var e= ( sliceray.sliceit(ty,3,ty.Length));



            foreach (var pw in e)
            {
                Console.WriteLine(pw);
            }
        }
    }
```

6> *Generic stack:* A stack is a data structure based on the principle of LIFO—"Last-In First-Out." A stack has two operations: *push* an object on the stack, and *pop* an object off the stack.

```
public class Stack<T>
{
int position;
T[] data = new T[100];
public void Push (T obj) { data[position++] = obj; }
public T Pop() { return data[--position]; }
}
```

We can use Stack<T> as follows:

```
Stack<int> stack = new Stack<int>();
stack.Push(5);
stack.Push(10);
int x = stack.Pop(); // x is 10
int y = stack.Pop(); // y is 5
```

6b> A more elaborated stack for character. Can be made generic by using <t> instead of char.*

```
// A stack class for characters.
class Stack {
// These members are private.
char[] stck; // holds the stack
int tos; // index of the top of the stack
// Construct an empty Stack given its size.
public Stack(int size) {
stck = new char[size]; // allocate memory for stack
tos = 0;
}
// Push characters onto the stack.
public void Push(char ch) {
if(tos==stck.Length) {
Console.WriteLine(" -- Stack is full.");
return;
}
stck[tos] = ch;
tos++;
}
// Pop a character from the stack.

public char Pop() {
if(tos==0) {
Console.WriteLine(" -- Stack is empty.");
return (char) 0;
}
tos--;
return stck[tos];
}
// Return true if the stack is full.
public bool IsFull() {
return tos==stck.Length;
}
// Return true if the stack is empty.
public bool IsEmpty() {
return tos==0;
}
// Return total capacity of the stack.
public int Capacity() {
return stck.Length;
}v c
// Return number of objects currently on the stack.
public int GetNum() {
return tos;
}

}
```

7>Bubble Sort:

The sort gets its name because values "float like a bubble" from one end of the list to another. Assuming you are sorting a list of numbers in ascending order, higher values float to the right whereas lower values float to the left. This behavior is caused by moving through the list many times, comparing adjacent values and swapping them if the value to the left is greater than the value to the right.

```
class Program
    {
        static void Main(string[] args)
        {
            int[] arr = new int[] { 51, 34, 3, 23 };

            var upper = arr.Length;
            int temp;
            for (int outer = upper; outer >= 1; outer--)
            {
                for (int inner = 0; inner < outer - 1; inner++)
                {
                    if ((int)arr[inner] > arr[inner + 1])
                    {
                        temp = arr[inner];
                        arr[inner] = arr[inner + 1];
                        arr[inner + 1] = temp;
                    }
                }
            }
            for (int i = 0; i < arr.Length; i++)
            {
                Console.WriteLine(arr[i]);
            }

        }
    }
```

8>Selection Sort

The next sort to examine is the Selection sort. This sort works by starting at the beginning of the array, comparing the first element with the other elements in the array. The smallest element is placed in position 0, and the sort then begins again at position 1. This continues until each position except the last position has been the starting point for a new loop. Two loops are used in the SelectionSort algorithm. The outer loop moves fromthe first element in the array to the next to last element, whereas the inner loop moves from the second element of the array to the last element, looking for values that are smaller than the element currently being pointed at by the outer loop. After each iteration of the inner loop, the most minimum value
in the array is assigned to its proper place in the array.

```
int[] arr = new int[] { 51, 34, 3, 6,89,4 };

            var upper = arr.Length;


            int min, temp;

            for (int outer = 0; outer < upper; outer++)

            {

                min = outer;

                for (int inner = outer + 1; inner < upper; inner++)

                    if (arr[inner] < arr[min])

                        min = inner;

                temp = arr[outer];

                arr[outer] = arr[min];

                arr[min] = temp;

            }

            for (int i = 0; i < arr.Length; i++)

            {


                Console.WriteLine(arr[i] );


            }
```

----

*9> Linear(*or *sequential search)*

```
static void Main(string[] args)
        {
            string[] yu = new string[] { "hi", "tell", "bro", "show" };
            int index;
            var t = lsearch.search(yu, "bro", out index);
            if (t)
            {
                Console.Write("found int the {0} position!!", index);
            }
            else
            {
                Console.Write("Not found !!");
            }
            }
        public static class lsearch
        {
            public static bool search(string[] arr, string value, out int index)
        {
            for (int i = 0; i < arr.Length; i++)
            {
                if (arr[i] == value)
                {
                    index = i;
                    return true;
                }
                              }
                index= -1;
            return false;
        }
                }
```

10>Queue

```
 public class queue<T>
    {
        int position=0;
        List<T> data = new List<T>();
        public void enqueue(T obj)
        {

            data.Add(obj);
            position++ ;
        }

        public T dequeue()
        {

            object ob = data[0]; // returns the item at position 0
            data.RemoveAt(0);  // removes the item at position 0

          return (T)ob;
        }

        public void remove()
        {
            data.RemoveAt(0);
        }

    }

    static void Main(string[] args)
    {
        queue<int> stack = new queue<int>();
        stack.enqueue(5);
        stack.enqueue(12);
        stack.enqueue(144);
        int x = stack.dequeue(); // x is 5
        Console.WriteLine(x);

        x = stack.dequeue(); // x is 12
        Console.WriteLine(x);



    }
```

11> Linked list

```
Class Program

    {


        public class node  // has two parts: Element -> Holds the content and Link -> holds
the reference to the next node.

        {

            public object element;

            public node link;


            public node()

            {

                element = null;

                link = null;


            }


            public node(object thestuff)

            {

                element = thestuff;

                link = null;




            }


              }
        public class LinkedList  // this is the linked list class

        {

            protected node header;


            public LinkedList()

            {

                header = new node(header);


            }


            private node Find(object ele) // it is used to find the existing node after which
    the new node is to be added.

            {

                node current = new node();

                current = header;


                while (current.element != ele)

                { current = current.link; }


                return current;


            }




            public void Insert(object newitem, object previtem)

            {

                node current = Find(previtem);

                node newnode = new node(newitem);


                newnode.link = current.link;

                current.link = newnode;
            }



            private node findprev(object ele)   // this is used to find the previous item before
the item to be deleted.

            {

                node current = new node();

                current = header;


                while (!(current.link == null) && (current.link.element != ele))


                {

                    current = current.link;

                }


                return current;


            }




            public void delete(object ele)  //this method deletes a node


            {

                node current = findprev(ele);


                if (current.link != null)

                {

                    current.link = current.link.link;

                    current.link.link = null;

                }
```

```
        }

        public void PrintList()

        {
            node current = header;
            while (current.link != null)
            {
                Console.WriteLine(current.element);
                current = current.link;

            }

        }

    }
```

12> Permutation

```
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Enter a string to permute");
            var theString = Console.ReadLine();

            // Get the items.
            string[] items = theString.Split(' ');

            // Generate the permutations.
            var SW1 = new Stopwatch();
            var SW2 = new Stopwatch();
            SW1.Start();
            List<List<string>> results =
                GeneratePermutations<string>(items.ToList());
            int count = 0;
            SW1.Stop();
            // Display the results.
            //results.Items.Clear();
            SW2.Start();
            foreach (List<string> combination in results)
            {
                string builtit=" ";
                StringBuilder createdS = new StringBuilder();
                foreach (var item in combination)
                {


                   createdS.Append( " " +item);
                   // builtit = builtit+ " "+ item;

                }
                count++;
                Console.WriteLine(createdS.ToString());
            }
            SW2.Stop();
            Console.WriteLine("Count {0}", count);
            Console.WriteLine("Comutation time: {0}", SW1.ElapsedMilliseconds);
            Console.WriteLine("Printing   time: {0}", SW2.ElapsedMilliseconds);
            Console.ReadLine();
        }





        // Generate permutations.
        private static List<List<T>> GeneratePermutations<T>(List<T> items)
        {
            // Make an array to hold the
            // permutation we are building.
            T[] current_permutation = new T[items.Count];

            // Make an array to tell whether
            // an item is in the current selection.
            bool[] in_selection = new bool[items.Count];

            // Make a result list.
            List<List<T>> results = new List<List<T>>();

            // Build the combinations recursively.
            PermuteItems<T>(items, in_selection,
                current_permutation, results, 0);

            // Return the results.
            return results;
        }


        // Recursively permute the items that are
        // not yet in the current selection.
        private static void PermuteItems<T>(List<T> items, bool[] in_selection,
            T[] current_permutation, List<List<T>> results,
            int next_position)
        {
            // See if all of the positions are filled.
            if (next_position == items.Count)
            {
                // All of the positioned are filled.
                // Save this permutation.
                results.Add(current_permutation.ToList());
            }
            else
            {
                // Try options for the next position.
                for (int i = 0; i < items.Count; i++)
                {
                    if (!in_selection[i])
                    {
                        // Add this item to the current permutation.
                        in_selection[i] = true;
                        current_permutation[next_position] = items[i];

                        // Recursively fill the remaining positions.
                        PermuteItems<T>(items, in_selection,
                            current_permutation, results,
                            next_position + 1);

                        // Remove the item from the current permutation.
                        in_selection[i] = false;
                    }
                }
            }//end else

        }//end PermuteItems


    }
```

13> Find a substring in a string and its index.

```
static class Program
{
    static List<string> AllSubstrings = new List<string>();
    static void Main()
    {
        int index;

        string value;
        Console.WriteLine("Enter the string");
        value = Console.ReadLine();
```

```
            Console.WriteLine("Enter the string to find ");
            string stringToFind = Console.ReadLine();


            AllSubstrings = GetAllSubstrings(value);
            if (FindMatch(stringToFind, out index, AllSubstrings))
            {
                Console.WriteLine("Match found at {0}", index);

            }

            else
            {
                Console.WriteLine("No matches found");
            }

        }


        static List<string> GetAllSubstrings(string input)
        {
            List<string> AllSubstrings = new List<string>();
            for (int length = 1; length < input.Length; length++)
            {
                // End index is tricky.
                for (int start = 0; start <= input.Length - length; start++)
                {
                    string substring = input.Substring(start, length) + " : " + start.ToString();
                    AllSubstrings.Add(substring);
                }
            }

            return AllSubstrings;
        }


        static Boolean FindMatch(this string matchWord, out int index, List<string>
    ListOfSubstrings)
        {
            index = -1;
            foreach (var item in ListOfSubstrings)
            {
                var itemtoSearch = item.Split(':')[0].Trim();
                if (itemtoSearch == matchWord)
                {
                    index = int.Parse(item.Split(':')[1]);
                    return true;

                }
            }
            return false;

        }
    }
```