# Anagrams

|  |
| --- |
| using System; |
|  | using System.Collections.Generic; |
|  | using System.Linq; |
|  | using System.Text; |
|  | using System.Threading.Tasks; |
|  |  |
|  | namespace AreAnagrams |
|  | { |
|  | class Program |
|  | { |
|  | static void Main(string[] args) |
|  | { |
|  | string s1 = String.Empty; |
|  | string s2 = String.Empty; |
|  | Console.WriteLine("Check if two strings are anagrams of each other"); |
|  | Console.WriteLine("Enter First String"); |
|  | s1 = Console.ReadLine(); |
|  | Console.WriteLine("Enter Second String"); |
|  | s2 = Console.ReadLine(); |
|  | Console.WriteLine(); |
|  | Console.WriteLine("Ara Anagram: " + AreStringsAnagrams(s1, s2).ToString()); |
|  | Console.ReadLine(); |
|  | } |
|  |  |
|  | public static bool AreStringsAnagrams(string s1, string s2) |
|  | { |
|  | Dictionary<char, int> dic = new Dictionary<char, int>(); |
|  |  |
|  | foreach (char c in s1) |
|  | { |
|  | if (!dic.ContainsKey(c)) |
|  | { |
|  | dic.Add(c, 1); |
|  | } |
|  | else |
|  | { |
|  | dic[c] = dic[c] + 1; |
|  | } |
|  | } |
|  |  |
|  | foreach (char c in s2) |
|  | { |
|  | if (!dic.ContainsKey(c) || dic[c] <= 0) |
|  | { |
|  | return false; |
|  | } |
|  | else |
|  | { |
|  | dic[c] = dic[c] - 1; |
|  | } |
|  | } |
|  | return true; |
|  | } |
|  | } |
|  | } |

# in the path

|  |
| --- |
| using System; |
|  | using System.Collections.Generic; |
|  | using System.Linq; |
|  | using System.Text; |
|  | using System.Threading.Tasks; |
|  |  |
|  | namespace DosPath |
|  | { |
|  | class Program |
|  | { |
|  | static void Main(string[] args) |
|  | { |
|  | Console.WriteLine("Read \"Read Me\" File in the Repo"); |
|  | while (true) |
|  | { |
|  | Console.WriteLine("Type in a DOS path"); |
|  | string s1 = Console.ReadLine(); |
|  | Console.WriteLine("Type in CD DOS command path"); |
|  | string s2 = Console.ReadLine(); |
|  |  |
|  | Path path = new Path(s1); |
|  | string result = path.CD(s2); |
|  |  |
|  | Console.WriteLine(); |
|  | Console.WriteLine("You are in Path " + result); |
|  | Console.WriteLine(); |
|  |  |
|  | Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"); |
|  | } |
|  | } |
|  | } |
|  |  |
|  | public class Path |
|  | { |
|  | private readonly Stack<string> stack; |
|  |  |
|  | public Path(string path) |
|  | { |
|  | bool expectFolderName = false; |
|  | string folderName = String.Empty; |
|  |  |
|  | stack = new Stack<string>(); |
|  | foreach (char c in path) |
|  | { |
|  | if (c != '/' && expectFolderName) |
|  | { |
|  | folderName = folderName + c; |
|  | } |
|  | else if (c == '/' && expectFolderName) |
|  | { |
|  | stack.Push(folderName); |
|  | //expectFolderName = false; |
|  | folderName = String.Empty; |
|  | } |
|  | else if (c == '/') |
|  | { |
|  | expectFolderName = true; |
|  | } |
|  | else |
|  | { |
|  | throw new NotImplementedException(); |
|  | } |
|  | } |
|  |  |
|  | //Add last character |
|  | if (expectFolderName && folderName != String.Empty) |
|  | { |
|  | stack.Push(folderName); |
|  | } |
|  | } |
|  |  |
|  | public string CD(string pathAdjustment) |
|  | { |
|  | string output = String.Empty; |
|  | bool expectingFolder = false; |
|  | string folderName = String.Empty; |
|  |  |
|  | for (int i = 0; i < pathAdjustment.Length; i++) |
|  | { |
|  | Char thisChar = pathAdjustment[i]; |
|  | if (thisChar == '.') |
|  | { |
|  | // parent |
|  | stack.Pop(); |
|  | i++; |
|  | } |
|  | else if (thisChar != '.' && thisChar != '/' && expectingFolder) |
|  | { |
|  | folderName = folderName + thisChar; |
|  | } |
|  | else if (thisChar == '/' && expectingFolder && folderName != String.Empty) |
|  | { |
|  | stack.Push(folderName); |
|  | folderName = String.Empty; |
|  | //expectingFolder = false; |
|  | } |
|  | else if (thisChar == '/') |
|  | { |
|  | // new folder |
|  | expectingFolder = true; |
|  | //stack.Push(pathAdjustment[i + 1]); |
|  | } |
|  | else |
|  | { |
|  | throw new NotImplementedException(); |
|  | } |
|  | } |
|  |  |
|  | //Add last character |
|  | if (expectingFolder && folderName != String.Empty) |
|  | { |
|  | stack.Push(folderName); |
|  | } |
|  |  |
|  | foreach (string d in stack.Reverse()) |
|  | { |
|  | output = output + "/" + d; |
|  | } |
|  | return output; |
|  | } |
|  | } |
|  | } |

# Frog leap

|  |
| --- |
|  |
| using System; |
|  | using System.Collections.Generic; |
|  | using System.Linq; |
|  | using System.Text; |
|  | using System.Threading.Tasks; |
|  |  |
|  | namespace TestDomeAlgoEasy |
|  | { |
|  | class Program |
|  | { |
|  | public static List<move[]> lisofpathes; |
|  |  |
|  | static void Main(string[] args) |
|  | { |
|  | Console.WriteLine("Frog Leap: Calculates the number of different combinations a frog can use to cover a given distance"); |
|  | Console.WriteLine(new string('\*', 20)); |
|  |  |
|  | while (true) |
|  | { |
|  | PrintMenu(); |
|  | } |
|  | } |
|  |  |
|  | public static void PrintMenu() |
|  | { |
|  | Console.WriteLine("Enter your distance for Froggie to walk in"); |
|  | string s1 = Console.ReadLine(); |
|  | int goal = 0; |
|  | if (!int.TryParse(s1, out goal)) { return; } |
|  |  |
|  | lisofpathes = new List<move[]>(); |
|  | Node rootNode = new Node(move.none); |
|  | Stack<move> pathStack = new Stack<move>(); |
|  |  |
|  | Frog(rootNode, goal, pathStack); |
|  |  |
|  | if(lisofpathes.Count <= 0 || lisofpathes[0].Length <= 0) { Console.WriteLine("No distance for Froggie to walk"); return; } |
|  |  |
|  | //Print All Possible Pathes |
|  | Console.WriteLine("Froggie possible pathes are: "); |
|  | StringBuilder sb = new StringBuilder(); |
|  | foreach (move[] path in lisofpathes) |
|  | { |
|  | foreach (move p in path) |
|  | { |
|  | sb.Append(p); |
|  | sb.Append(','); |
|  | } |
|  | sb.Remove(sb.Length - 1, 1); |
|  | Console.WriteLine(sb.ToString()); |
|  | sb.Clear(); |
|  | } |
|  | Console.WriteLine(new string('\*', 20)); |
|  | } |
|  |  |
|  | public static void Frog(Node node, int steps, Stack<move> stack) |
|  | { |
|  | if (steps > 1) |
|  | { |
|  | //If remaining steps > 1 (meaning froggies can either Jump or Step) |
|  | for (int i = 0; i < 2; i++) |
|  | { |
|  | //Each Parent Node has Two Child Nodes |
|  | if (i == 0) |
|  | { |
|  | //First Child Node - Holds the Jump |
|  | node.LNode = new Node(move.jump); //Move to the Left Node |
|  | Node n = node.LNode; //Pass in the Left Node |
|  | int s = steps - 2; //Holds the no of the remaining steps at the parent level before the recurssive call to the children, so when the recurssive returns I will have the (no of steps at the parent node). |
|  | stack.Push(move.jump); //push froggie move into the stack |
|  | Frog(n, s, stack); |
|  | stack.Pop(); //pop the child node step, stack pointer now points at the parent node step |
|  | } |
|  | else if (i == 1) |
|  | { |
|  | //Second Child Node - Holds the Step |
|  | node.RNode = new Node(move.step); |
|  | Node n = node.RNode; |
|  | int s = steps - 1; |
|  | stack.Push(move.step); |
|  | Frog(n, s, stack); |
|  | stack.Pop(); |
|  | } |
|  | } |
|  | } |
|  | else if (steps == 1) |
|  | { |
|  | //If remaining step is Only 1: Froggie can only Step, he/she cannot jump |
|  | node.CNode = new Node(move.step); |
|  | int s = steps - 1; |
|  | stack.Push(move.step); |
|  | Frog(node, s, stack); |
|  | stack.Pop(); |
|  | } |
|  | else if (steps <= 0) |
|  | { |
|  | //If remaining steps is zero: No more remaining steps - Log the stack which has the path Froggie made |
|  | // add pathStack to list of pathes |
|  | move[] a = stack.Select(q => q).ToArray(); |
|  | lisofpathes.Add(a); |
|  | } |
|  | } |
|  | } |
|  |  |
|  | enum move |
|  | { |
|  | none, |
|  | step, |
|  | jump |
|  | } |
|  |  |
|  | class Node |
|  | { |
|  | public move value { get; set; } |
|  | public Node RNode { get; set; } |
|  | public Node LNode { get; set; } |
|  | public Node CNode { get; set; } |
|  |  |
|  | private Node() { } |
|  |  |
|  | public Node(move move) |
|  | { |
|  | this.value = move; |
|  | } |
|  | } |
|  | } |
|  |  |

# TWOSum

using System;

using System.Collections.Generic;

class TwoSum

{

public static Tuple<int, int> FindTwoSum(IList<int> list, int sum)

{

HashSet<int> hs = new HashSet<int>();

for (int i = 0; i < list.Count; i++)

{

var needed = sum - list[i];

if (hs.Contains(needed))

{

return Tuple.Create(list.IndexOf(needed), i);

}

hs.Add(list[i]);

}

return null;

}

public static void Main(string[] args)

{

Tuple<int, int> indices = FindTwoSum(new List<int>() { 3, 1, 5, 7, 5, 9 }, 10);

if (indices != null)

{

Console.WriteLine(indices.Item1 + " " + indices.Item2);

}

}

}

# Longest run

Abbcccddddcccbba should print 6

static void Main(string[] args)

{

string s1 = String.Empty;

Console.WriteLine("Enter String");

// s1 = Console.ReadLine();

s1 = "abbcccddddcccbba";

Console.WriteLine();

Console.WriteLine("Longest Run Letter Index is: " + IndexOfLongestRun(s1).ToString());

Console.WriteLine();

Console.Read();

}

static int IndexOfLongestRun(string input)

{

int longestRunStart = -1, longestRunLength = 0;

for (int i = 0; i < input.Length; )

{

var runValue = input[i];

int runStart = i;

while (++i < input.Length && input[i] == runValue) { }

int runLength = i - runStart;

if (longestRunLength < runLength)

{

longestRunStart = runStart;

longestRunLength = runLength;

}

}

return longestRunStart;

}

public static int IndexOfLongestRun3(string str)

{

var longestRunCount = 1;

var longestRunIndex = 0;

var isNew = false;

var dic = new Dictionary<int, int>();

for (var i = 0; i < str.Length - 1; i++)

{

if (str[i] == str[i + 1])

{

if (isNew) longestRunIndex = i;

longestRunCount++;

isNew = false;

}

else

{

isNew = true;

dic.Add(longestRunIndex, longestRunCount);

longestRunIndex = 0;

longestRunCount = 1;

}

}

return dic.OrderByDescending(x => x.Value).First().Key;

}

# Flags

[Flags]

public enum Options : byte

{

None = 0,

One = 1 << 0, // 1

Two = 1 << 1, // 2

Three = 1 << 2, // 4

Four = 1 << 3, // 8

// combinations

OneAndTwo = One | Two,

OneTwoAndThree = One | Two | Three,

}

[Flags]

public enum MyColors

{

Yellow = 1,

Green = 2,

Red = 4,

Blue = 8

}

To retrieve the distinct values in your property, one can do this:

if((myProperties.AllowedColors & MyColor.Yellow) == MyColor.Yellow)

{

// Yellow has been set...

}

if((myProperties.AllowedColors & MyColor.Green) == MyColor.Green)

{

// Green has been set...

}

or, in .NET 4 and later:

if (myProperties.AllowedColors.HasFlag(MyColor.Yellow))

{

// Yellow has been set...

}

**Under the covers**

This works because you previously used powers of two in your enumeration. Under the covers, your enumeration values look like this (presented as bytes, which has 8 bits which can be 1's or 0's)

Yellow: 00000001

Green: 00000010

Red: 00000100

Blue: 00001000

Likewise, after you've set your property AllowedColors to Red, Green and Blue (which values where OR'ed by the pipe |), AllowedColors looks like this

myProperties.AllowedColors: 00001110

So when you retrieve the value you are actually bitwise AND'ing the values

myProperties.AllowedColors: 00001110

MyColor.Green: 00000010

-----------------------

00000010 // Hey, this is the same as MyColor.Green!

# Reverse a string

public static class ReverseWords

{

//Reverse words in a string

public static string Reverse(string s)

{

string result = "";

Stack temp = new Stack();

s = s.Trim();

//Remove extra white space between characters

while (s.Contains(" "))

s = s.Replace(" ", " ");

//Store each word on the Stack

foreach (string x in s.Split(' '))

temp.Push(x);

//Add each word to the result string in reverse order since a stack is a FIFO data structure

while (temp.Count != 0)

result += temp.Pop() + " ";

return result.Trim();

}

}

}

OR

public static class ReverseString

{

public static string Reverse(string x)

{

string result = "";

for (int i = x.Length - 1; i >= 0; i--)

result += x[i];

return result;

}

}

# If an array contains duplicates

public static class ArrayDuplicates

{

//Dictionary solution

public static bool ContainsDuplicates(params int[] x)

{

Dictionary<int, int> d = new Dictionary<int, int>();

foreach (int i in x)

{

if (d.ContainsKey(i))

return true;

else

d.Add(i, 1);

}

return false;

}

}

# Romans to Integers

public static int RomanToInteger(string input)

{

int result = 0;

Dictionary<char, int> romMap = new Dictionary<char, int>()

{

{'I', 1},{'V', 5},

{'X', 10},{'L', 50},

{'C', 100},{'D', 500},

{'M', 1000}

};

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

{

// dont check the last && current is less than next number

if (i + 1 < input.Length && romMap[input[i]] < romMap[input[i + 1]])

result -= romMap[input[i]];

else

result += romMap[input[i]];

}

return result;

}

}

// int ii = Solution.RomanToInteger("MCMLXI"); should return 1961

# Fibonacci

int Fibonacci(int i) => i <= 1 ? i : Fibonacci(i - 1) + Fibonacci(i - 2);

public static int **Fibonacci**(int n)

{

int a = 0;

int b = 1;

// In N steps compute Fibonacci sequence iteratively.

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

{

int temp = a;

a = b;

b = temp + b;

}

return a;

}

static void Main()

{

for (int i = 0; i < 15; i++)

{

Console.WriteLine(Fibonacci(i));

}

}

0

1

1

2

3

5

8

13

21

34

55

89

144

233

377

or

1. **using** System;
2. **public** **class** FibonacciExample
3. {
4. **public** **static** **void** Main(**string**[] args)
5. {
6. **int** n1=0,n2=1,n3,i,number;
7. Console.Write("Enter the number of elements: ");
8. number = **int**.Parse(Console.ReadLine());
9. Console.Write(n1+" "+n2+" "); //printing 0 and 1
10. **for**(i=2;i<number;++i) //loop starts from 2 because 0 and 1 are already printed
11. {
12. n3=n1+n2;
13. Console.Write(n3+" ");
14. n1=n2;
15. n2=n3;
16. }
17. }
18. }

Or using recursion

Fibonacci(0, 1, 1, number);

public static void Fibonacci(int a, int b, int counter, int number)

{

Console.WriteLine(a);

if (counter < number) Fibonacci(b, a+b, counter+1, number);

}

public static int Fibonatchi(int position) {

if(position == 0) {

return 1;

}

if(position == 1) {

return 1;

} else {

return Fibonatchi(position - 2) + Fibonatchi(position - 1);

}

}

# Deepest string

string str = "[a [b [c [d value]]]]";

while (str.Trim().Length > 0)

{

int start = str.LastIndexOf('[');

int end = str.IndexOf(']');

string s = str.Substring(start + 1, end - (start + 1)).Trim();

string[] pair = s.Split(' ');// this is what you are looking for. its length will be 2 if it has a value

str = str.Remove(start, (end + 1) - start);

}

# Longest string in array

# list.Aggregate("", (max, cur) => max.Length > cur.Length ? max : cur);

# Unique characters in string

string code = "AABBDDCCRRFF";

string answer = new String(code.Distinct().ToArray());

 public static class UniqueString

    {

        public static bool IsUnique(string s)

        {

            Dictionary<char, int> d = new Dictionary<char, int>();

            foreach (char c in s)

            {

                if (d.ContainsKey(c))

                    return false;

                else

                    d.Add(c, 1);

            }

            return true;

        }

        //Compares each character to every other character without using an additional data structure

        //O(n^2) time complexity

        public static bool IsUnique1(string s)

        {

            string temp1 = "";

            string temp2 = "";

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

            {

                temp1 = s.Substring(i, 1);

                for (int k = 0; k < s.Length; k++)

                {

                    temp2 = s.Substring(k, 1);

                    if ((temp1 == temp2) && (i != k))

                        return false;

                }

            }

            return true;

        }

    }

}

# Palindrome

new string(Enumerable.Range(1, input.Length).Select(i => input[input.Length - i]).ToArray())

 /// <summary>

        /// Determines whether the string is a palindrome.

        /// </summary>

        public static bool IsPalindrome(string word)

        {

            int min = 0;

            int max = word.Length - 1;

            while (true)

            {

                if (min > max)

                {

                    return true;

                }

                char a = word[min];

                char b = word[max];

                if (char.ToLower(a) != char.ToLower(b))

                {

                    return false;

                }

                min++;

                max--;

            }

        }

# Prefixes

using System;

using System.Collections.Generic;

using System.Linq;

public class Prefix

{

public static IEnumerable<string> Prefixes(IEnumerable<string> words, int length)

{

return words.Where(w => w.Length >= 3).Select(w => w.Substring(0, 3)).Distinct() ;

}

public static void Main(string[] args)

{

foreach (var p in Prefixes(new string[] { "many", "manly", "men", "maybe", "my" }, 3))

Console.WriteLine(p);

Console.Read();

}

}

# Movies overlap

using System;

using System.Collections.Generic;

using System.Linq;

public class Movie

{

public DateTime Start { get; private set; }

public DateTime End { get; private set; }

public Movie(DateTime start, DateTime end)

{

this.Start = start;

this.End = end;

}

}

public static class MovieNight

{

public static bool CanViewAll(IEnumerable<Movie> movies)

{

var startMovies = movies.OrderByDescending(s => s.Start);

int i = 0;

do

{

Movie a = startMovies.ElementAt(i);

Movie b = startMovies.ElementAt(i+1);

if (a.Start < b.End && b.Start < a.End)

return false;

i++;

}

while (i < startMovies.Count()-1);

return true;

}

public static void Main1(string[] args)

{

var format = System.Globalization.CultureInfo.InvariantCulture.DateTimeFormat;

Movie[] movies = new Movie[]

{

new Movie(DateTime.Parse("1/1/2015 20:00", format), DateTime.Parse("1/1/2015 21:30", format)),

new Movie(DateTime.Parse("1/1/2015 23:10", format), DateTime.Parse("1/1/2015 23:30", format)),

new Movie(DateTime.Parse("1/1/2015 21:30", format), DateTime.Parse("1/1/2015 23:00", format))

};

bool b = MovieNight.CanViewAll(movies);

}

}

# Candies algorithm

using System;

public class Candies

{

public static int CountCandies(int startingAmount, int newEvery)

{

int total = startingAmount-1;

do

{

startingAmount = startingAmount - newEvery + 1;

total = total + 1;

}

while (startingAmount > 0);

/\* while (startingAmount > 0)

{

startingAmount = startingAmount - newEvery + 1;

total = total + 1;

}

\*/

return total;

}

public static void Main(string[] args)

{

int i = Candies.CountCandies(3, 2);

}

}

# Binary Search

int binary\_search(int A[], int key, int imin, int imax)

{

// test if array is empty

if (imax < imin)

// set is empty, so return value showing not found

return KEY\_NOT\_FOUND;

else

{

// calculate midpoint to cut set in half

int imid = midpoint(imin, imax);

// three-way comparison

if (A[imid] > key)

// key is in lower subset

return binary\_search(A, key, imin, imid - 1);

else if (A[imid] < key)

// key is in upper subset

return binary\_search(A, key, imid + 1, imax);

else

// key has been found

return imid;

}

}

# Largest 3 elements of an array

var topThree = (from i in array

orderby i descending

select i).Take(3);

// C# code to find largest

// three elements in an array

using System;

class PrintLargest

{

// Function to print three

// largest elements

static void print2largest(int[] arr,

int arr\_size)

{

int i, first, second, third;

// There should be atleast two elements

if (arr\_size < 3)

{

Console.WriteLine("Invalid Input");

return;

}

third = first = second = 000;

for (i = 0; i < arr\_size; i++)

{

// If current element is

// smaller than first

if (arr[i] > first)

{

third = second;

second = first;

first = arr[i];

}

// If arr[i] is in between first

// and second then update second

else if (arr[i] > second)

{

third = second;

second = arr[i];

}

else if (arr[i] > third)

third = arr[i];

}

Console.WriteLine("Three largest elements are " +

first + " " + second +

" " + third);

}

// Driver code

public static void Main()

{

int[] arr = new int[] { 12, 13, 1, 10, 34, 1 };

int n = arr.Length;

print2largest(arr, n);

}

}

// This code is contributed by KRV.

Output :

Three largest elements are 34 13 12

# HashSet vs List

A lot of people are saying that once you get to the size where speed is actually a concern that HashSet<T> will always beat List<T>, but that depends on what you are doing.

Let's say you have a List<T> that will only ever have on average 5 items in it. Over a large number of cycles, if a single item is added or removed each cycle, you may well be better off using a List<T>.

I did a test for this on my machine, and, well, it has to be very very small to get an advantage from List<T>. For a list of short strings, the advantage went away after size 5, for objects after size 20.

# T VS A i.e. .ToList() vs .AsEnumerable()

.ToList() is expensive as it copies the entire array first in a list.

**Genius sports**

My client are recognised as one of the fastest growing sports technology companies in the world. They are a global leader in providing innovative and data-driven solutions to sports media and regulated betting markets.  
  
They are currently looking for someone who’s got strong and current C# skills, preferably with a web background. The successful candidate will be managing large volumes of data, working in a fully Agile environment and will be based in a fantastic location in Soho, London!  
  
Essential Skills:  
.Net  
C#  
Unit testing  
TDD  
Agile practices