**Quest #3 Instructions and Rubric**

As a group, implement your version of data compression and decompression algorithm. You may write your solution using C++, C#, or Java. The Unity platform is not allowed. Submit your code, testing, and outputs. Also, present your solution during our next meeting. Submit your code project here together with your multiple test cases to show your generalized code.  Only one submission is required.  Please put all the names of your team members in your final deliverable.  Make sure that all members of the team agree on a solution through collaboration.  In other words, the product should not be the work of only one team member.

Part I of the project is the compression solution. Please show several test cases to prove that your solution is generic and not hardcoded.

Part II of the assignment is to decompressed what you have created in part I.

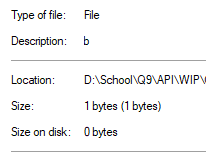
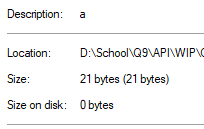
It is recommended that you work on your code while on the campus to experience eXtreme programming. Remote coding will pose a challenge, especially for a group that is not utilizing a dynamic code repository such as GitHub.

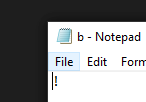
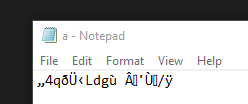
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| GRADING RUBRIC | | | | |
| Grading Criteria | 3  **Exceeds**  *Excellent*  Epic Wow | 2  **Meets**  *Satisfactory*  O.K. | 1  **Partially Meets**  *Below Expectations*  Not Yet | 0  **Does Not Meet**  *Unacceptable*  Fail |
| **Code –** is unique, custom-made, no evidence of being downloaded from the Internet, relevant to video gaming, working as intended, implemented two or more data structures and algorithm, and to the professor’s satisfaction | Code is excellent and understandable :20 | Code is satisfactory :15 | Code is obfuscated, or hard to follow :10 | Unfortunately, no evidence of a working code. |
| **Documentation –** showed logic in layman’s term (preferably not a flowchart), included a detailed documentation such as the approach’s illustration, program code, testing, sample screen shots of working program, etc. | Documentation is excellent and easy to verify Part I and II :10 | Documentation is satisfactory :6 | Documentation is hard to follow, or deficient :3 | Unfortunately, no accompanying documentation. |
| **Presentation –** student collaborated, cooperated, and presented the concept and all artifacts of the project with his/her team in an understandable fashion. | Excellent presentation :10 | Satisfactory presentation :5 | Presentation and collaboration have plenty of room for improvement. :2 | Unfortunately, no presentation. |
| **Time Management –** candidate used time wisely during development, presentation, and all aspects of the work submitted in a timely fashion. | Work submitted promptly :10 | Submitted within the allotted time :5 | Submitted late :3 | Unfortunately, too long, too short, or untimely. |

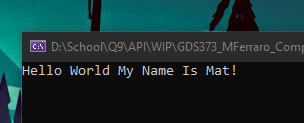
Mathew Ferraro

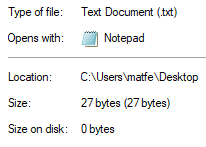
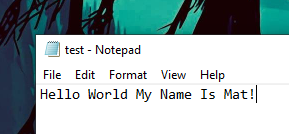
Jason Kozin

Dan Heyen

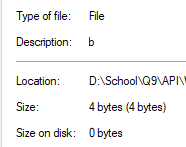
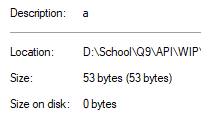


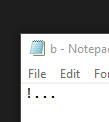
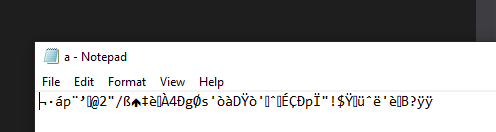


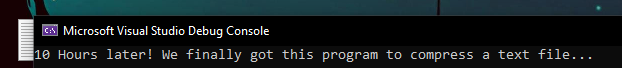
 This is output after we decompressed the above files

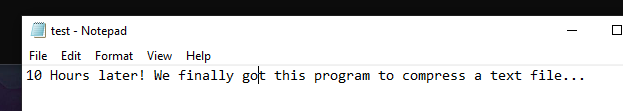


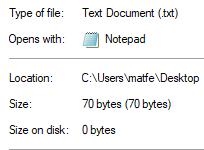
21 + 1 bytes < 27 bytes











53 + 4 bytes < 70 bytes

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Reflection;

using System.IO;

namespace GDS373\_MFerraro\_Compression\_2

{

static class Program

{

// this turns the bitarray into a byte list that we can more easily use

static byte[] ToByteArray(this BitArray bits)

{

int numBytes = bits.Count / 8;

if (bits.Count % 8 != 0) numBytes++;

byte[] bytes = new byte[numBytes];

int byteIndex = 0, bitIndex = 0;

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

{

if (bits[i])

bytes[byteIndex] |= (byte)(1 << (7 - bitIndex));

bitIndex++;

if (bitIndex == 8)

{

bitIndex = 0;

byteIndex++;

}

}

return bytes;

}

static byte ReverseByte(byte inByte)

{

byte result = 0x00;

for (byte mask = 0x80; Convert.ToInt32(mask) > 0; mask >>= 1)

{

// shift right current result

result = (byte)(result >> 1);

// tempbyte = 1 if there is a 1 in the current position

var tempbyte = (byte)(inByte & mask);

if (tempbyte != 0x00)

{

// Insert a 1 in the left

result = (byte)(result | 0x80);

}

}

return (result);

}

static void Main(string[] args)

{

List<List<bool>> conversionTable = new List<List<bool>>();

List<char> unlistedChar = new List<char>();

for (int i = 0; i < 64; i++) // loops through the entire table

{

List<bool> letters = new List<bool>();

int j = i;

do

{

int temp = j % 2;

if (temp == 0) // if even

{

if (j == 2) // it always has to go back to 1

{

j = 1;

}

else

{

j /= 2;

}

letters.Add(false);

}

else // if odd

{

j = (int)(j / 2);

letters.Add(true);

}

} while (j > 0);

List<bool> correctedLetter = new List<bool>();

// if the byte doesn't have 6 bits

// add onto the end

if(letters.Count < 6)

{

for (int k = 0; k < letters.Count; k++)

{

correctedLetter.Add(letters[k]);

}

for (int l = 6 - letters.Count; l > 0; l--)

{

correctedLetter.Add(false);

}

conversionTable.Add(correctedLetter);

}

else

{

conversionTable.Add(letters);

}

}

#region dictionary

// Make a dictionary for all compressed characters.

IDictionary<char , int> sixBitDictionary = new Dictionary<char, int>();

sixBitDictionary.Add('a', 0);

sixBitDictionary.Add('b', 1);

sixBitDictionary.Add('c', 2);

sixBitDictionary.Add('d', 3);

sixBitDictionary.Add('e', 4);

sixBitDictionary.Add('f', 5);

sixBitDictionary.Add('g', 6);

sixBitDictionary.Add('h', 7);

sixBitDictionary.Add('i', 8);

sixBitDictionary.Add('j', 9);

sixBitDictionary.Add('k', 10);

sixBitDictionary.Add('l', 11);

sixBitDictionary.Add('m', 12);

sixBitDictionary.Add('n', 13);

sixBitDictionary.Add('o', 14);

sixBitDictionary.Add('p', 15);

sixBitDictionary.Add('q', 16);

sixBitDictionary.Add('r', 17);

sixBitDictionary.Add('s', 18);

sixBitDictionary.Add('t', 19);

sixBitDictionary.Add('u', 20);

sixBitDictionary.Add('v', 21);

sixBitDictionary.Add('w', 22);

sixBitDictionary.Add('x', 23);

sixBitDictionary.Add('y', 24);

sixBitDictionary.Add('z', 25);

sixBitDictionary.Add('A', 26);

sixBitDictionary.Add('B', 27);

sixBitDictionary.Add('C', 28);

sixBitDictionary.Add('D', 29);

sixBitDictionary.Add('E', 30);

sixBitDictionary.Add('F', 31);

sixBitDictionary.Add('G', 32);

sixBitDictionary.Add('H', 33);

sixBitDictionary.Add('I', 34);

sixBitDictionary.Add('J', 35);

sixBitDictionary.Add('K', 36);

sixBitDictionary.Add('L', 37);

sixBitDictionary.Add('M', 38);

sixBitDictionary.Add('N', 39);

sixBitDictionary.Add('O', 40);

sixBitDictionary.Add('P', 41);

sixBitDictionary.Add('Q', 42);

sixBitDictionary.Add('R', 43);

sixBitDictionary.Add('S', 44);

sixBitDictionary.Add('T', 45);

sixBitDictionary.Add('U', 46);

sixBitDictionary.Add('V', 47);

sixBitDictionary.Add('W', 48);

sixBitDictionary.Add('X', 49);

sixBitDictionary.Add('Y', 50);

sixBitDictionary.Add('Z', 51);

sixBitDictionary.Add('0', 52);

sixBitDictionary.Add('1', 53);

sixBitDictionary.Add('2', 54);

sixBitDictionary.Add('3', 55);

sixBitDictionary.Add('4', 56);

sixBitDictionary.Add('5', 57);

sixBitDictionary.Add('6', 58);

sixBitDictionary.Add('7', 59);

sixBitDictionary.Add('8', 60);

sixBitDictionary.Add('9', 61);

sixBitDictionary.Add(' ', 62);

// use the last entry (63) as a catch all

// which will be an identifier for the switch from bits

#endregion

// ===========================================================

// Compress the data

string test = "Hello World My Name is Mat!";

// total bits of the entire string

List<bool> total = new List<bool>();

using (BinaryWriter writer = new BinaryWriter(File.Open("a", FileMode.Create)))

{

// list of bits

List<bool> bitList = new List<bool>();

for (int i = 0; i < test.Length; i++) // for the length of the string

{

// if in the dictionary

if (sixBitDictionary.TryGetValue(test[i], out int value))

{

bitList = conversionTable[sixBitDictionary[test[i]]];

}

else

{

//Console.WriteLine("This character was not in the dictionary: " + test[i]);

bitList = conversionTable[63];

unlistedChar.Add(test[i]);

}

for (int j = 0; j < bitList.Count(); j++)

{

total.Add(bitList[j]);

}

}

}

// take the total list and now convert it to writable bytes (convert 6bit to 8 bit)

int byteSizeCorrection = 0;

if (total.Count() % 8 > 0)

{

byteSizeCorrection = 8 - (total.Count() % 8);

// add the remaining true's at the end of the list

// this represents the junk character 63 of the dictionary

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

{

total.Add(true);

}

}

// take the boolean list and convert it to a byte[]

byte[] writableBytes = new byte[(int)MathF.Ceiling(total.Count() / 8)];

BitArray bitArray = new BitArray(total.Count());

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

{

bitArray.Set(i, total[i]);

}

writableBytes = ToByteArray(bitArray);

// write out the byte array to a file

File.WriteAllBytes("a", writableBytes);

// store unlisted characters in a seperate file (WIP)

using (StreamWriter w = File.CreateText("b"))

{

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

{

w.Write(unlistedChar[i]);

}

}

// ===========================================================

// Decompress the data

string uncompressedCharacters = System.IO.File.ReadAllText("b");

int numberOfUncompressedChars = uncompressedCharacters.Length;

int counter = 0;

List<byte> readBytes = new List<byte>();

using (BinaryReader reader = new BinaryReader(File.Open("a", FileMode.Open)))

{

while (reader.BaseStream.Position != reader.BaseStream.Length)

{

readBytes.Add(ReverseByte(reader.ReadByte()));

}

}

List<bool> boolList = new List<bool>();

// load all bits in order into one large bit array

BitArray bitArr = new BitArray(readBytes.Count \* 8);

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

{

for (int j = 0; j < 8; j++)

{

bitArr.Set((i \* 8) + j, (readBytes[i] & (1 << j % 8)) != 0);

}

}

// chunk the bit array into 6bit increments to determine the stored character

List<List<bool>> decompressedSixBitList = new List<List<bool>>();

for (int i = 0; i < (int)(bitArr.Count / 6); i++)

{

List<bool> sixBit = new List<bool>();

for (int j = 0; j < 6; j++)

{

sixBit.Add(bitArr[(i \* 6) + j]);

}

decompressedSixBitList.Add(sixBit);

}

// check the dictionary in reverse order

// pass this function a dictionary entry to look at

int dictionaryKey = 0;

var key = 'a';

string decompressedMessage = "";

for(int j = 0; j < decompressedSixBitList.Count; j++)

{

for (int i = 0; i < 64; i++)

{

if (decompressedSixBitList[j].ToArray().SequenceEqual(conversionTable[i].ToArray()))

{

dictionaryKey = i;

key = sixBitDictionary.FirstOrDefault(x => x.Value == dictionaryKey).Key;

if(i != 63)

{

decompressedMessage += key;

}

else

{

// pull from uncompressed character list

if(counter < numberOfUncompressedChars)

{

decompressedMessage += uncompressedCharacters[counter];

counter++;

}

else

{

// this is our edge case

// do nothing

}

}

}

}

}

Console.WriteLine(decompressedMessage);

Console.ReadKey();

}

}

}