Huffman Algorithm

for

C++

Version 1.0

Prepared by

Group Name: ***<Enes's team>***

ENES KILIÇ	**2000006890**	**2000006890@stu.iku.edu.tr**
İBRAHİM AĞCABAY	**1900002653**	**1900002653@stu.iku.edu.tr**

Instructor: ***EZGİ DEMİRCAN TÜREYEN***

Course: **CSE 4014 Data Structures and Algorithms-I**

Lab Section: ***A***

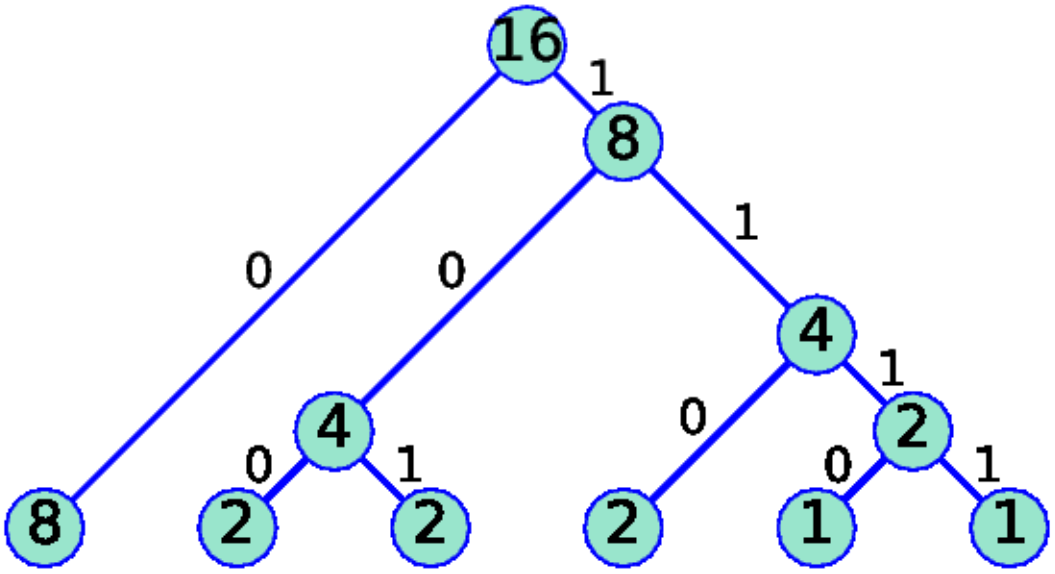
Date: **24.05.2021**

Table of Contents

Revisions

Version	Primary Author(s)	Description of Version	Date Completed
1.0	Enes Kılıç	The entire content of this document has been prepared	23/05/21

IMG_



1. INTRODUCTION

Huffman encoding (also known as Huffman Coding) is an algorithm for data compression and forms the basic idea behind file compression.

In this document, we have touched on a detailed description of the Huffman Algorithm, with examples related to it. Then, we explained our program we prepared on this subject with all its features and explained it in detail.

1.1. Overview

Whe already know that every character is sequences of 0's and 1's and stored using 8-bits. This is known as "fixed-length encoding", as each character uses the same number of fixed-bit storage.

How to reduce the amount of space required to store a character when given a text value?

We can use "variable length encoding" for this. We can represent the given text using fewer bit numbers to represent the same part of the text. In

variable-length coding, we assign a variable number of bits to characters depending on their frequency in the given text. Therefore, some characters can take even a single bit, while others can be encoded using two or more bits.

So given a string of bits, how can we solve it in a unique way?

Let's consider the string aabacdab. It has 8 characters in it and uses 64-bits storage. This is normal fixed length encoding. The frequency of characters a, b, c and d are 4, 2, 1, 1, respectively. Let's try to represent aabacdab suing a lesser number of bits by using the fact that a accurs more frequently than b, and b occurs more frequently than c and d. We starting by randomly assigning a single bit code 0 to a, 2-bit code 11 to b, and 3-bit code 100 and 011 to characters c and d, respectively;

a 0

b 11

c 100

d 011

The string aabacdab will be encoded to 00110100011011 (0|0|11|0|100|011|0|11) using the above codes. The main problem is decoding. If we try to decipher this code, different results will always occur depending on the point we divide. This will cause great uncertainty;

0|011|0|100|011|0|11 adacdab

0|0|11|0|100|0|11|011 aabacabd

0|011|0|100|0|11|0|11 adacabab

…

and so on

To avoid any ambiguity in decoding, we will make sure it meets the "prefix rule" that will result in "uniquely decodable codes". The definition of the prefix rule indicates that no code is a prefix of another code. When I say code, I am actually talking about the bits used for a certain character. In the example above, a prefix of 0 is a prefix violation of 011. Decoding is clear if bits use the prefix rule (and vice versa.)

Let's consider the above example again. This time we assign codes that satisfy the prefix rule to characters 'a', 'b', 'c', and 'd' .

```
a 0

b 10

c 110

d 111
```

Using the above codes, the string aabacdab will be encoded to 00100110111010 (0|0|10|0|110|111|0|10). Now we can uniquely decode 00100110111010 back to our original string aabacdab.

We needed to know these two situations before moving on to Huffman Coding. Let's continue with Huffman coding ..

1.2. Huffman Algorithm

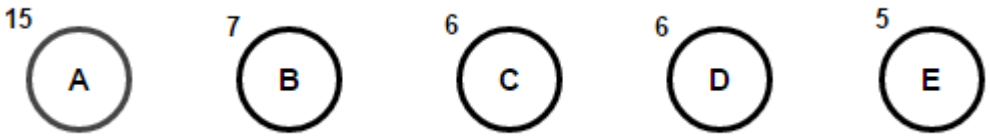
The technique works by creating a binary tree of nodes. A node are often either a leaf node or an indoor code. Initially, all nodes are leaf nodes, which contain the character itself, the weight(frequency of appearance) of the character. Internal nodes contain character weight and links to 2 child

nodes. As a cammon convention, bit 0 represents following the left child, and a touch 1 represents following the proper(right) child. A finished tree has n leaf nodes and n-1 internal nodes. It's recommended that Huffman Tree should discard unused characters within the text to supply the foremost optimal code lengths. This may give better results.

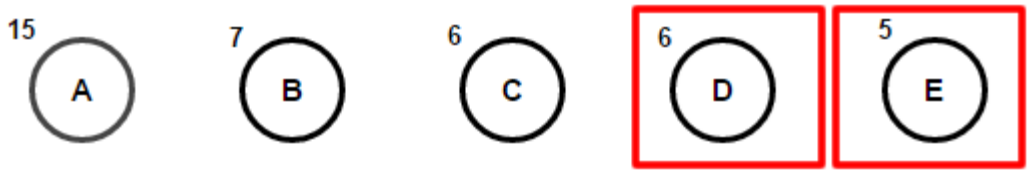
We will use a priority queue for building Huffman Tree, where the node with the lowest frequency has the very best priority. Following are the entire steps:

1. We must create a leaf node for each character and add them to the priority queue.
2. While there is more than one node in the queue:
   - Remove the two nodes of the highest priority (the lowest frequency) from the queue.
   - Create a new internal node with these two nodes as children and a frequency equal to the sum of both nodes' frequencies.
   - Add the new node to the priority queue.
3. The remaining node is the root node and the tree is complete.

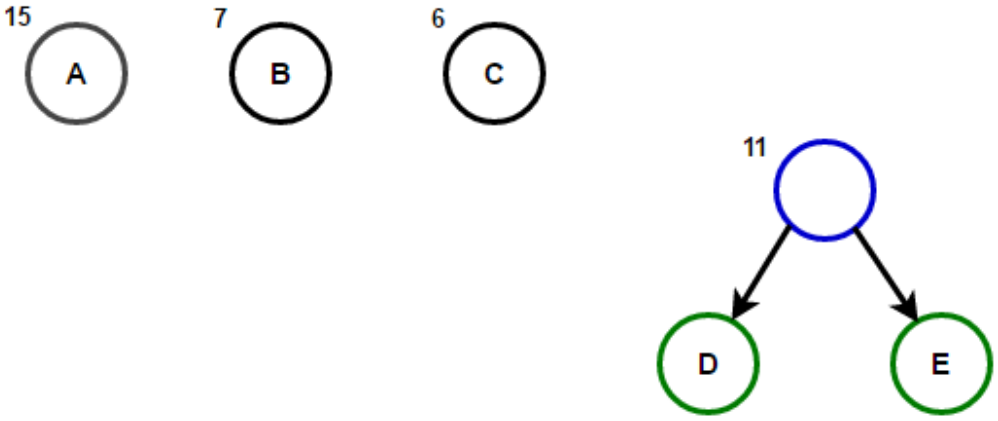
Consider some text consisting of only 'A', 'B', 'C', 'D', and 'E' characters, and their frequencies are 15, 7, 6, 6, 5, respectively. The following figures illustrate the steps followed by the alogrithm :



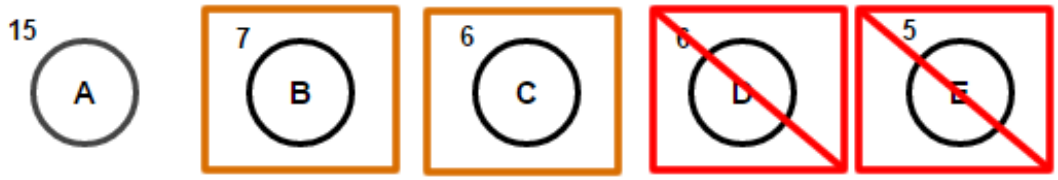
We select the character with the two smallest frequencies and try to add our tree node from bottom to top.



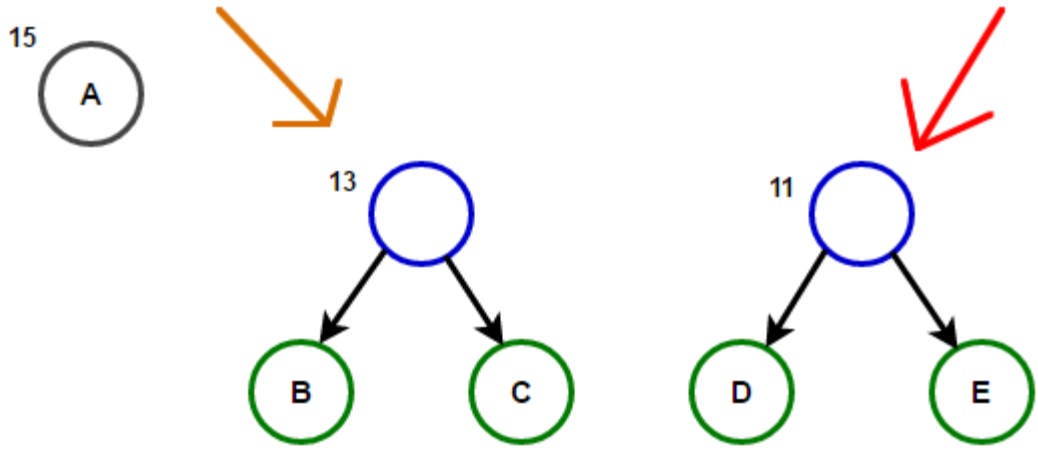
We collect the frequency values of the characters D and E and write them outside the blue circle.



Then we select the characters 'B' and 'C' with the other smallest bivalent frequencies.

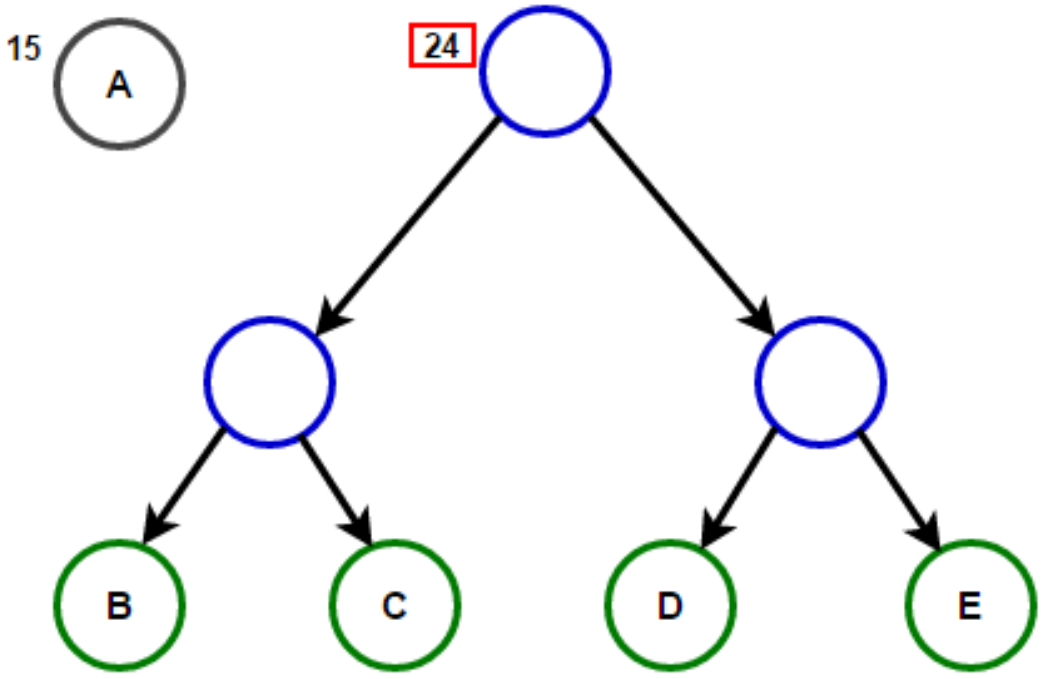


We form the other leg of our tree. We collect the characters B and C and create a new blue circle.



Our character A is waiting at this step because it is larger than the nodes with 13 and 11 frequency values that we created by adding the remaining character A.

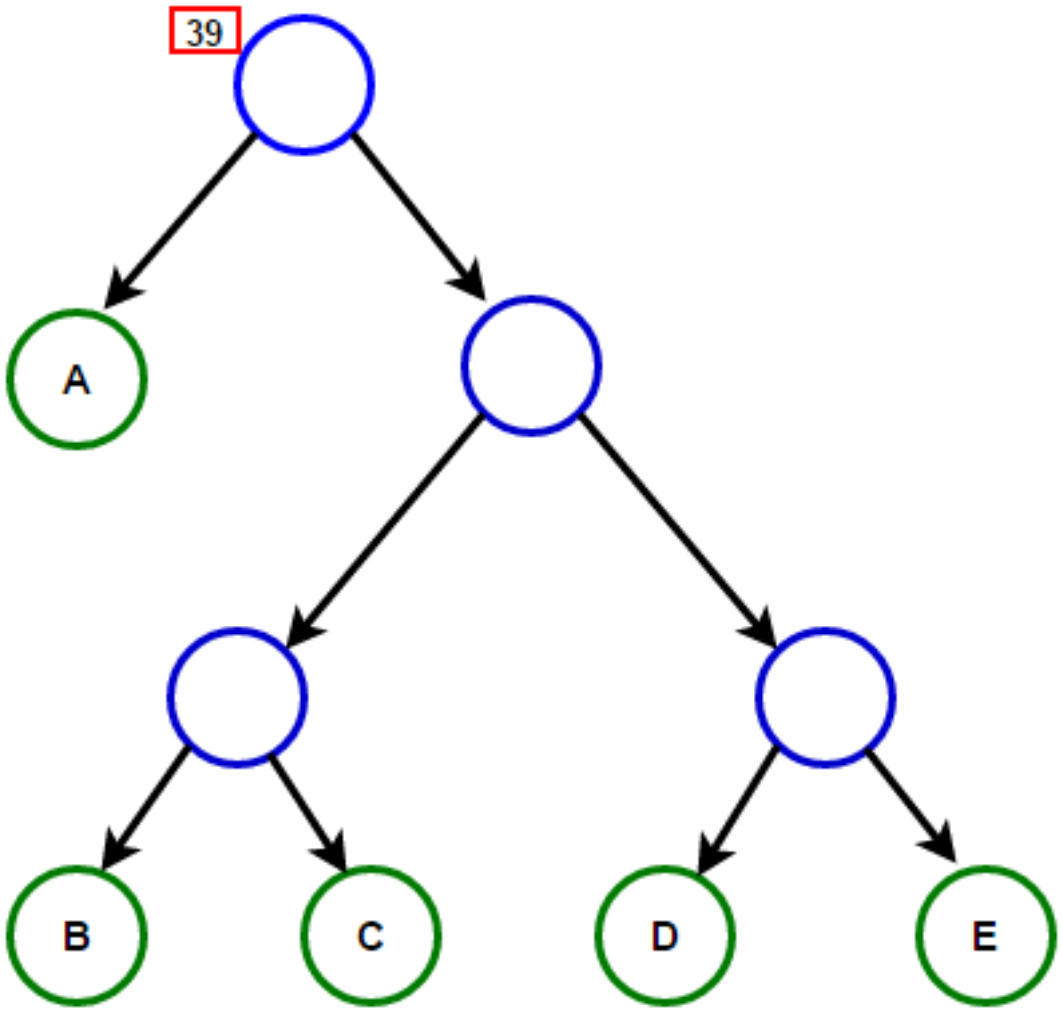
In this step, we add frequencies 13 and 11 and create a new blue circle. This is our top knot for now..



Our last node we have has 24 frequencies. Whereas in the previous step 15 was larger than the existing nodes, now our last incoming node has remained small because it has 24 valued frequencies.

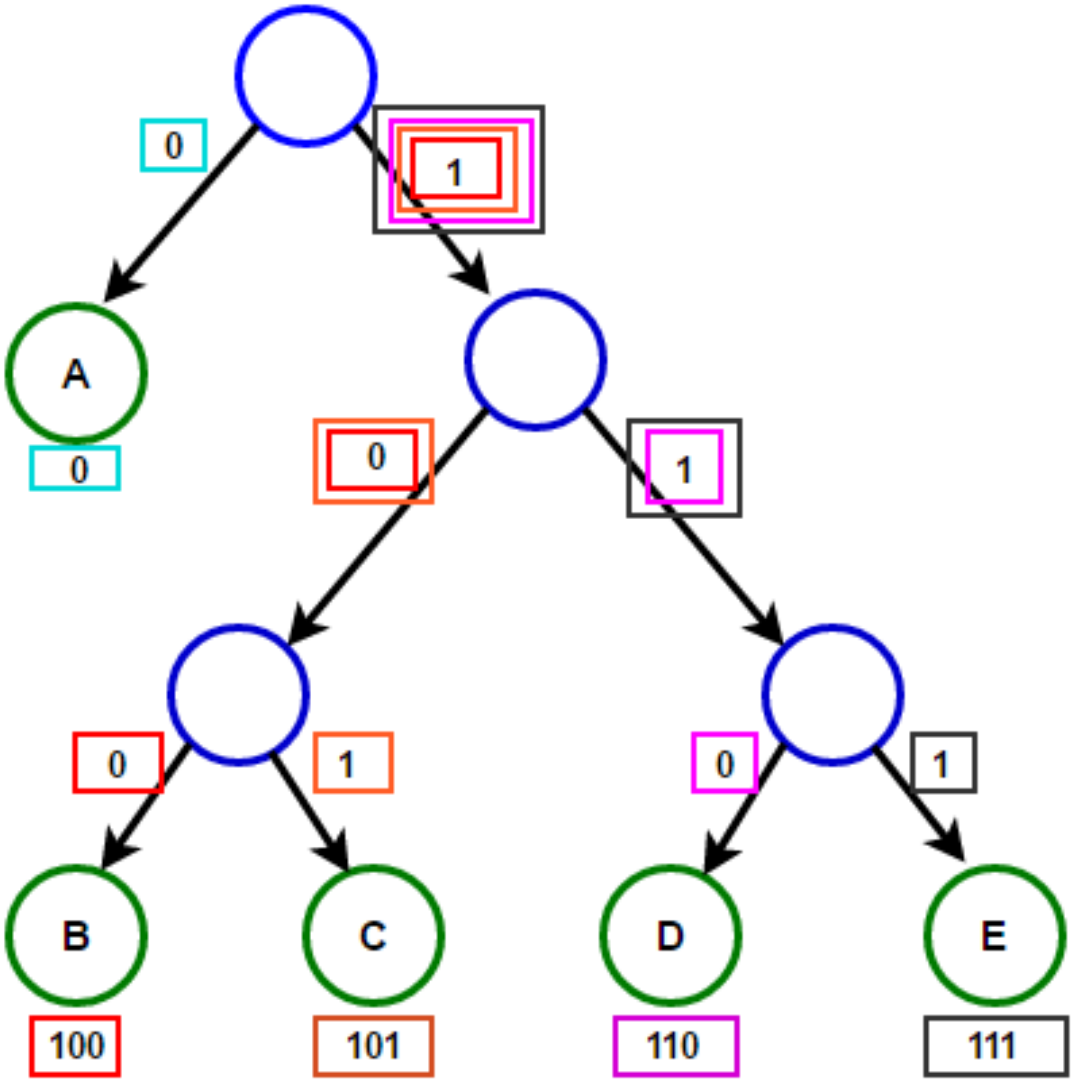
In this step, we collect our last node with 15 frequencies with the character A, with a 24-frequency value, and determine the top node of our Huffman algorithm. The top node now has 39 frequency values.

Thus, the tree node is complete.



The path from the root to any leaf node stores the optimal prefix code (also called Huffman code) corresponding to the character associated with that leaf node.

Thus, the bit values of each character after compression are determined.



2. OUR APPLICATION(C++)

Our application is using the Console Application infrastructure, using C++ programming language.

2.1. Purpose

The purpose of our program is to rearrange(compress) the data we want to use in any system / program in a way that covers less space in the memory. We used Huffman Algorithm in doing this. In this way, our programs will run more efficiently with less system requirements.

2.2. Scope

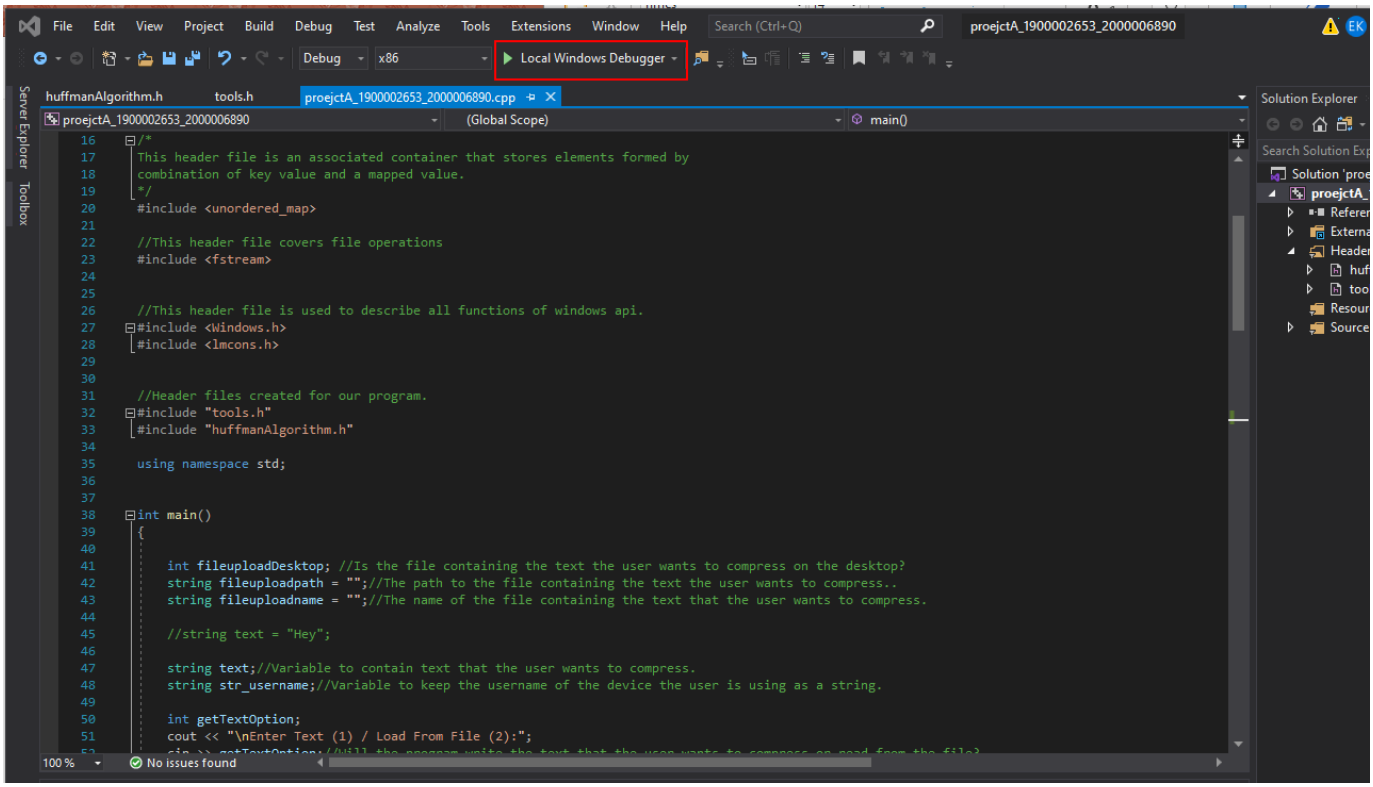
You can enter the data you want to compress into our program in two different ways. The first is that you can write it yourself or copy and paste it. Secondly, you save the data you want to compress in a txt file, then select this txt file through the program, allowing the program to read the data in this txt file.

It shows you the result of our program in both ways. On the Console screen, you can see the original version of your text, then the 8-bit uncompressed (Ascii) state in memory, the state after it has been compressed (Huffman) and finally the original version from its compressed state. Second, it will show you the uncompressed and compressed state separately as new txt files. (Of the type requested within the scope of the project.)

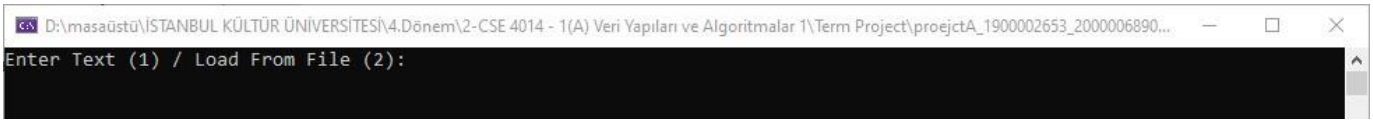
2.3. App(C++)

In this section, we will examine how to use our application step by step.

a) We start by running our application;

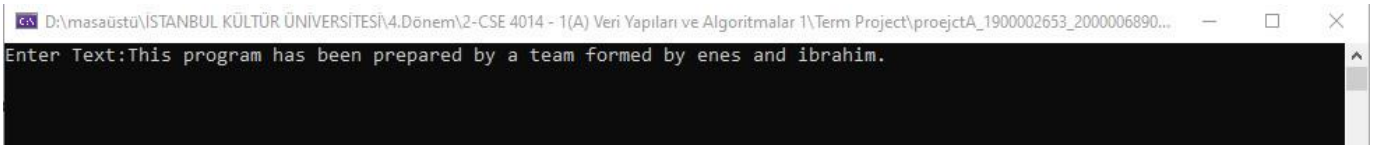


b) The screen you will see when the program runs is below. You are presented with two options on this screen. The first is to type or copy and paste your data entry from the keyboard. In the second option, if your data is in a txt file, it will direct you in that direction.



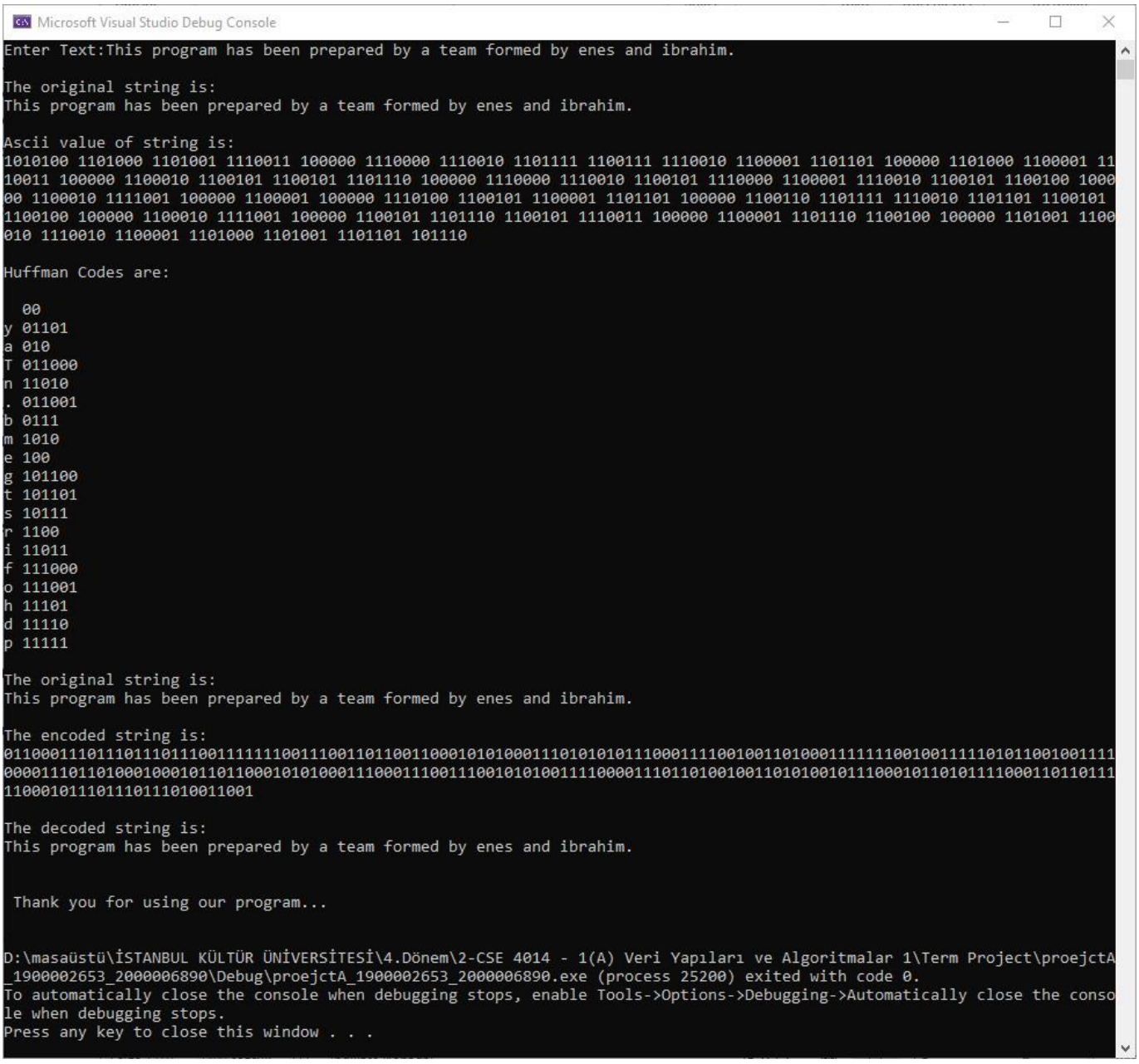
c) Let's continue our program by choosing the first option;

d) It asks us to enter data on the screen that appears. I am writing a short text for better understanding.

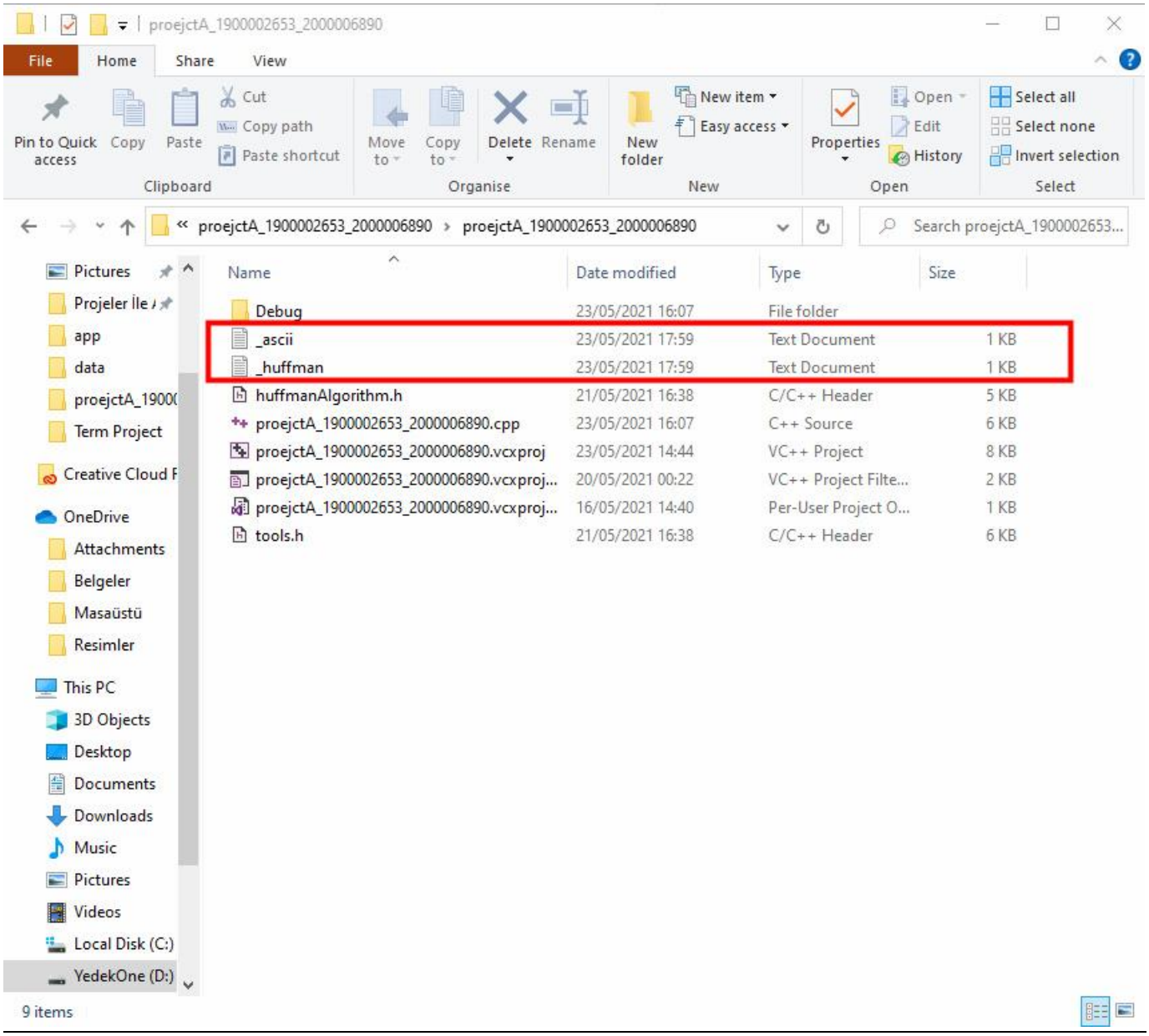


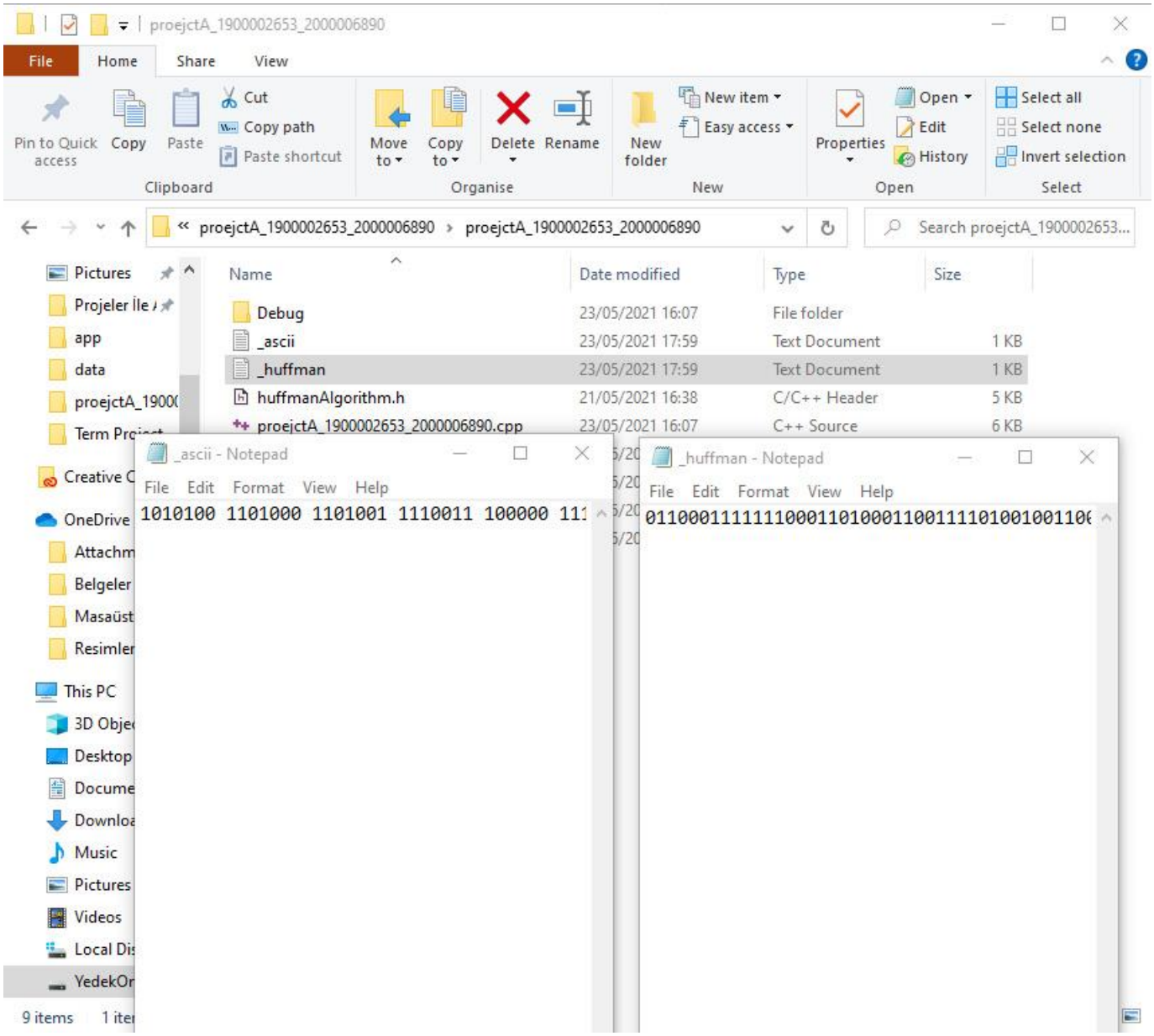
e) Let's examine the result we will see when we approve the text;

1. As seen in the picture below, we first show the original text of the text we wrote.
2. Then we view the space it occupies in the memory in binary type before it is compressed.
3. Then we see the characters of the text encoded with Huffman codes.
4. Again, we see the original version of the original text.
5. Then we see the text compressed with Huffman Algorithm.
6. Finally, we convert it from the compressed text, ie its binary equivalent, back to the original text. Thus, we see how we move from the coded text to its original uncoded state.



Together with the result screen, the uncompressed, ie Ascii codes and compressed values according to the Huffman Algorithm are presented to the user in the form of a new txt file.


projctA_1900002653_2000006890
File
Home
Share
View
Pin to Quick access
Copy
Paste
Cut
Copy path
Paste shortcut
Clipboard
Move to
Copy to
Delete
Rename
Organise
New item
Easy access
New folder
New
Properties
Open
Edit
History
Open
Select all
Select none
Invert selection
Select
« projctA_1900002653_2000006890 › projctA_1900002653_2000006890
Search projctA_1900002653...
Pictures
Projeler İle /
app
data
projctA_1900(
Term Project
Creative Cloud F
OneDrive
Attachments
Belgeler
Masaüstü
Resimler
This PC
3D Objects
Desktop
Documents
Downloads
Music
Pictures
Videos
Local Disk (C:)
YedekOne (D:)
Name
Date modified
Type
Size
Debug 23/05/2021 16:07 File folder
_ascii 23/05/2021 17:59 Text Document 1 KB
_huffman 23/05/2021 17:59 Text Document 1 KB
huffmanAlgorithm.h 21/05/2021 16:38 C/C++ Header 5 KB
projctA_1900002653_2000006890.cpp 23/05/2021 16:07 C++ Source 6 KB
projctA_1900002653_2000006890.vcxproj 23/05/2021 14:44 VC++ Project 8 KB
projctA_1900002653_2000006890.vcxproj... 20/05/2021 00:22 VC++ Project Filte... 2 KB
projctA_1900002653_2000006890.vcxproj... 16/05/2021 14:40 Per-User Project O... 1 KB
tools.h 21/05/2021 16:38 C/C++ Header 6 KB
9 items

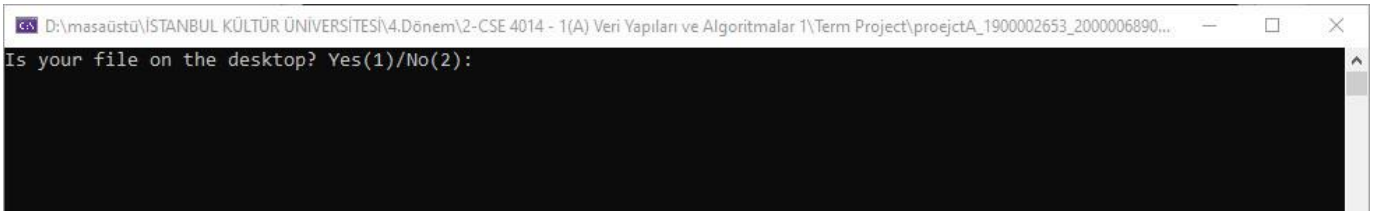


Thus, we learned the part of our program related to the first option. Now let's examine how it would have been if we chose the second option.

In the second option, we will repeat the same process using your txt file that you saved your data before.

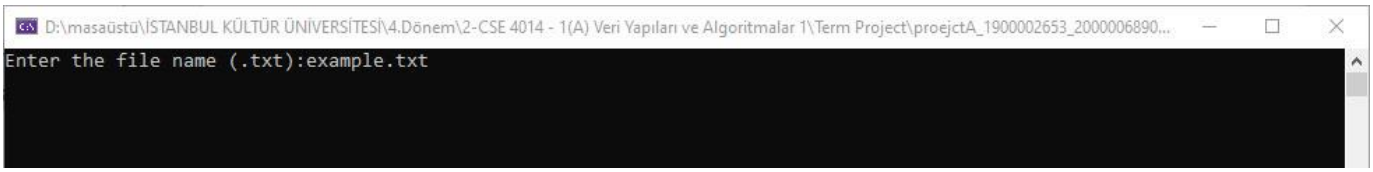
We choose the second option;

Then we see the screen that comes before us. For the sake of convenience, if our txt file is located on the desktop, we provide the program directly to the program, and without showing the full path to the program, simply enter the name of the txt file and reach the relevant file.

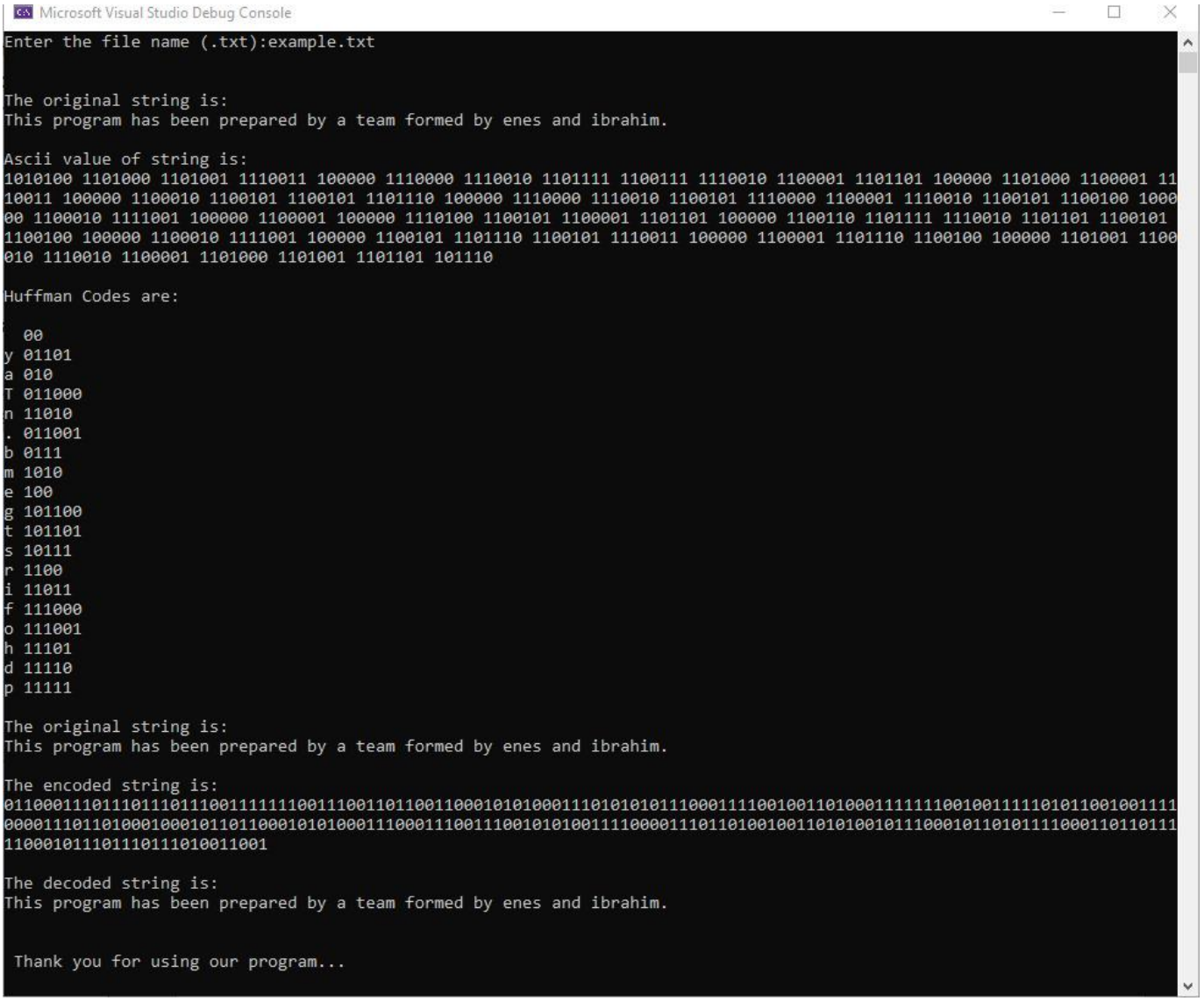


Let's act by thinking that our file is on the desktop, and then we will examine it in the other method.

After selecting it, it is waiting for us to enter the name of our txt file on the desktop. Our file on the desktop is "example.txt".

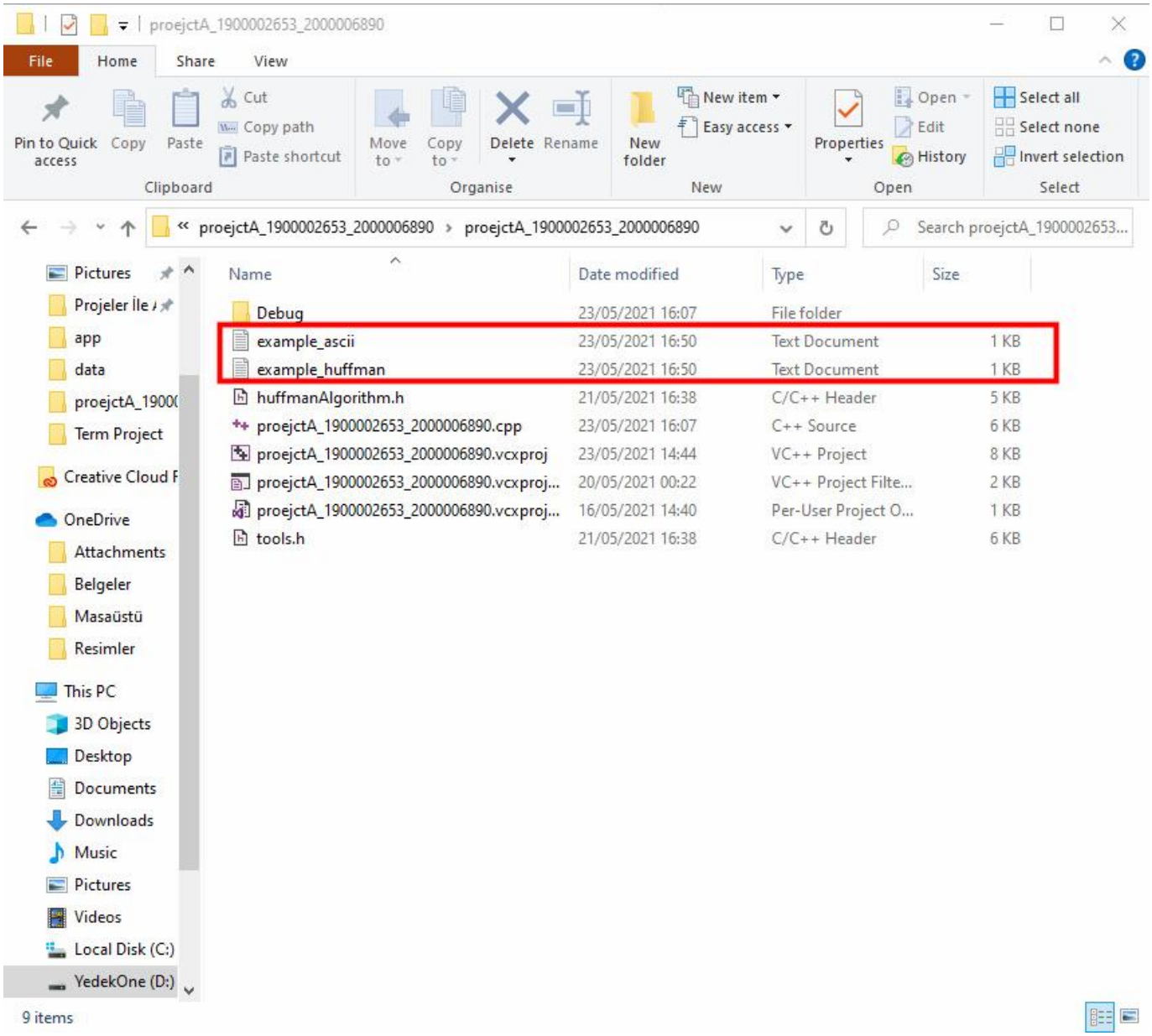


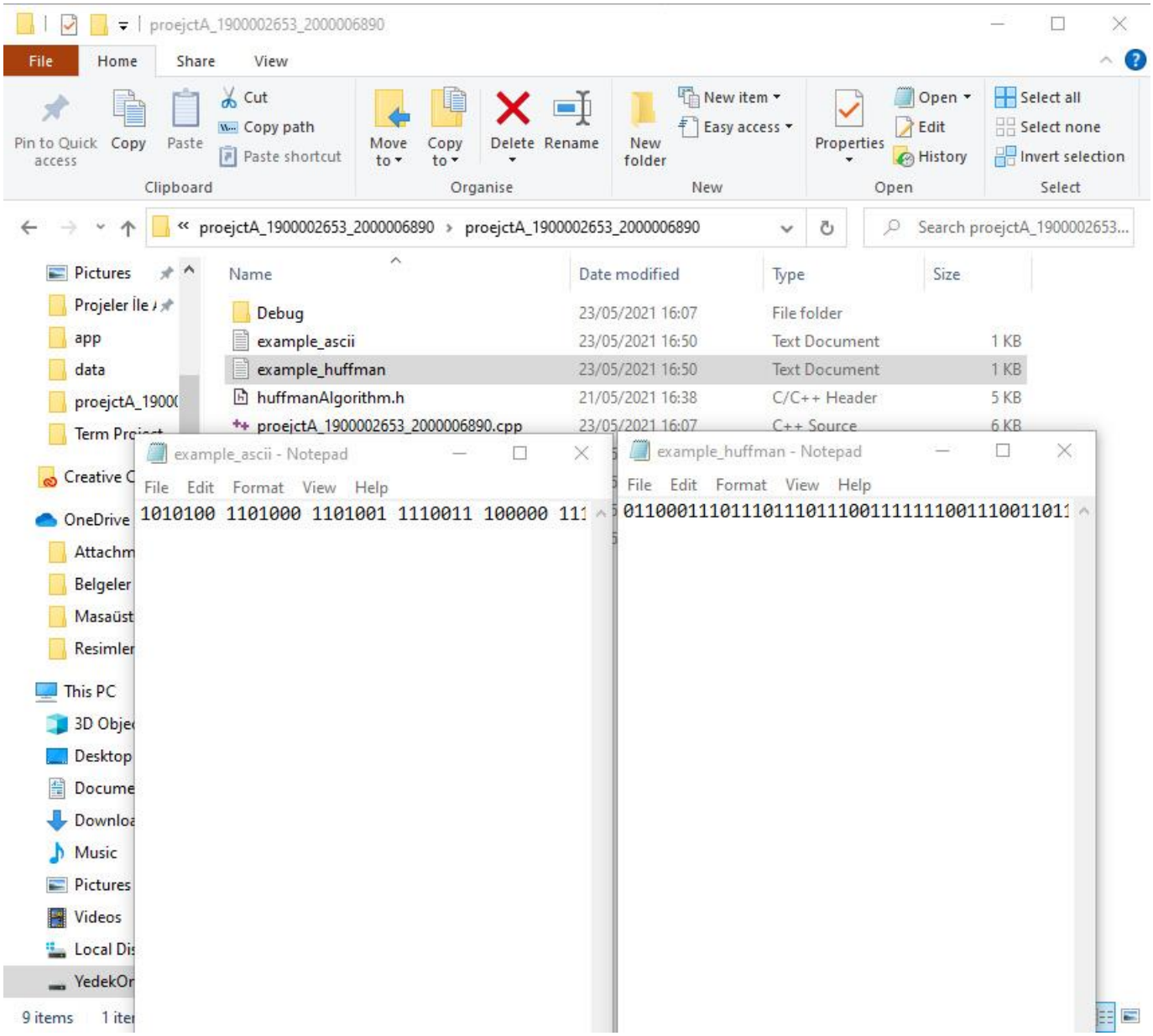
Then, when we examine the screen that appears, we see a similar result screen here, too. Therefore, there is no need to explain the same screen again.



Together with the result screen, the uncompressed, ie Ascii codes and compressed values according to the Huffman Algorithm are presented to the user in the form of a new txt file.

You can see these txt files in the red area in the picture below.

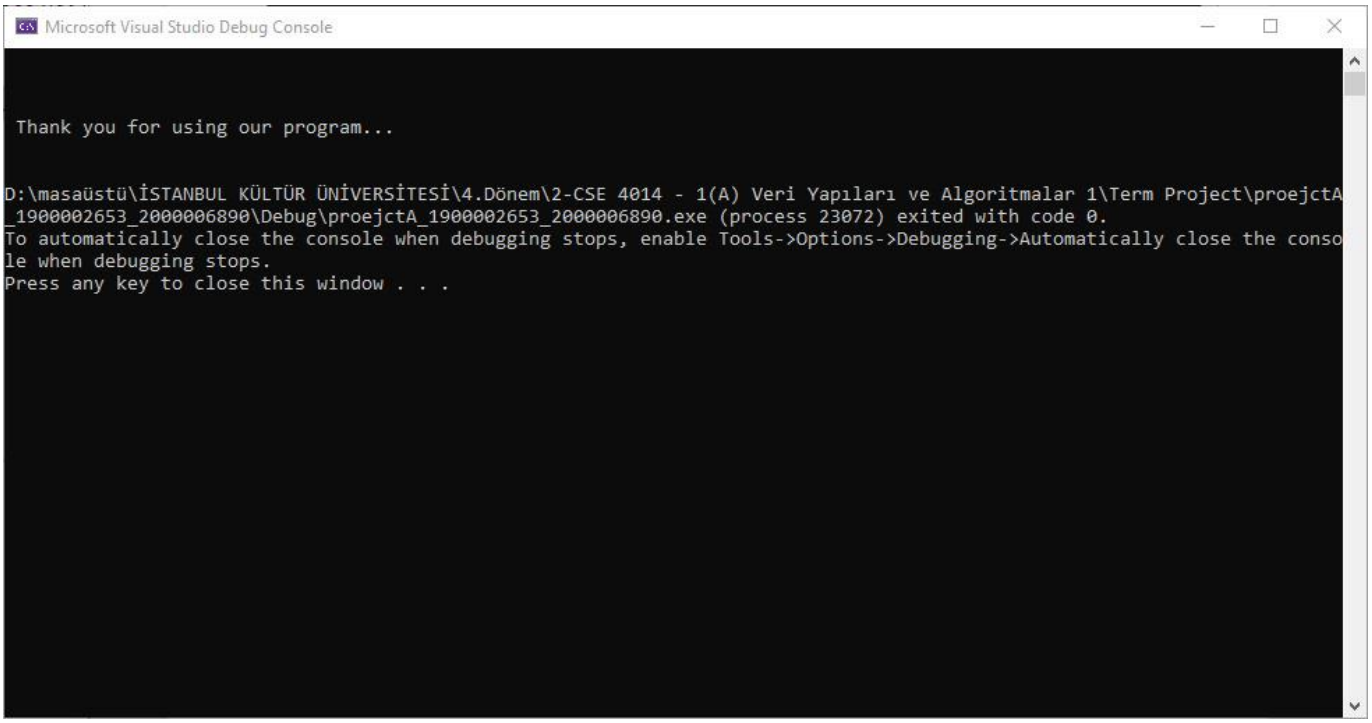




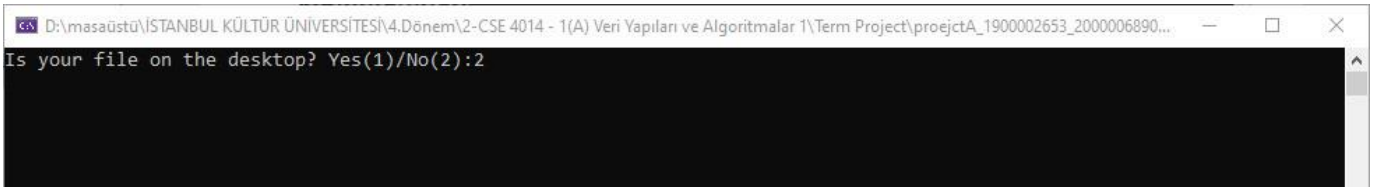
If the txt file is not found, the user is given a chance to try again. The program terminates if the user accepts it and does not want to try again.

If we choose the first option and want to try again;

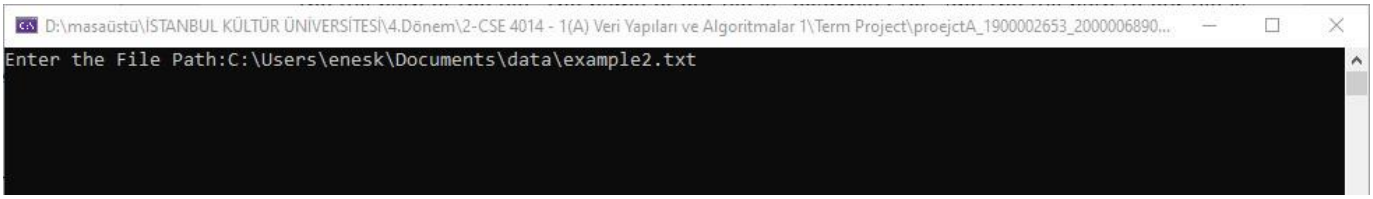
If we choose the second option and do not want to use the program;



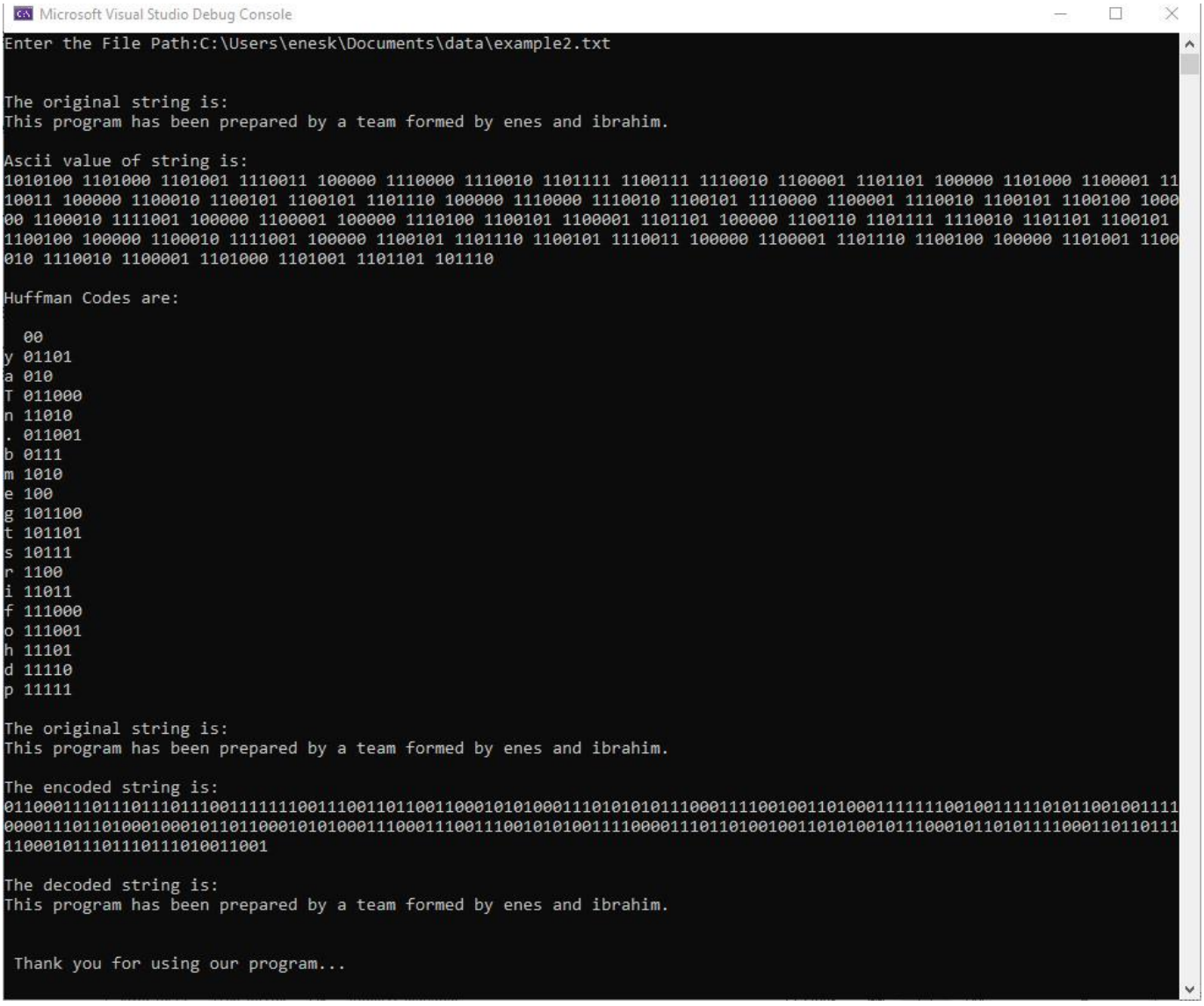
Now suppose our txt file is in any location, not on the desktop. Therefore, we must choose the second option ;



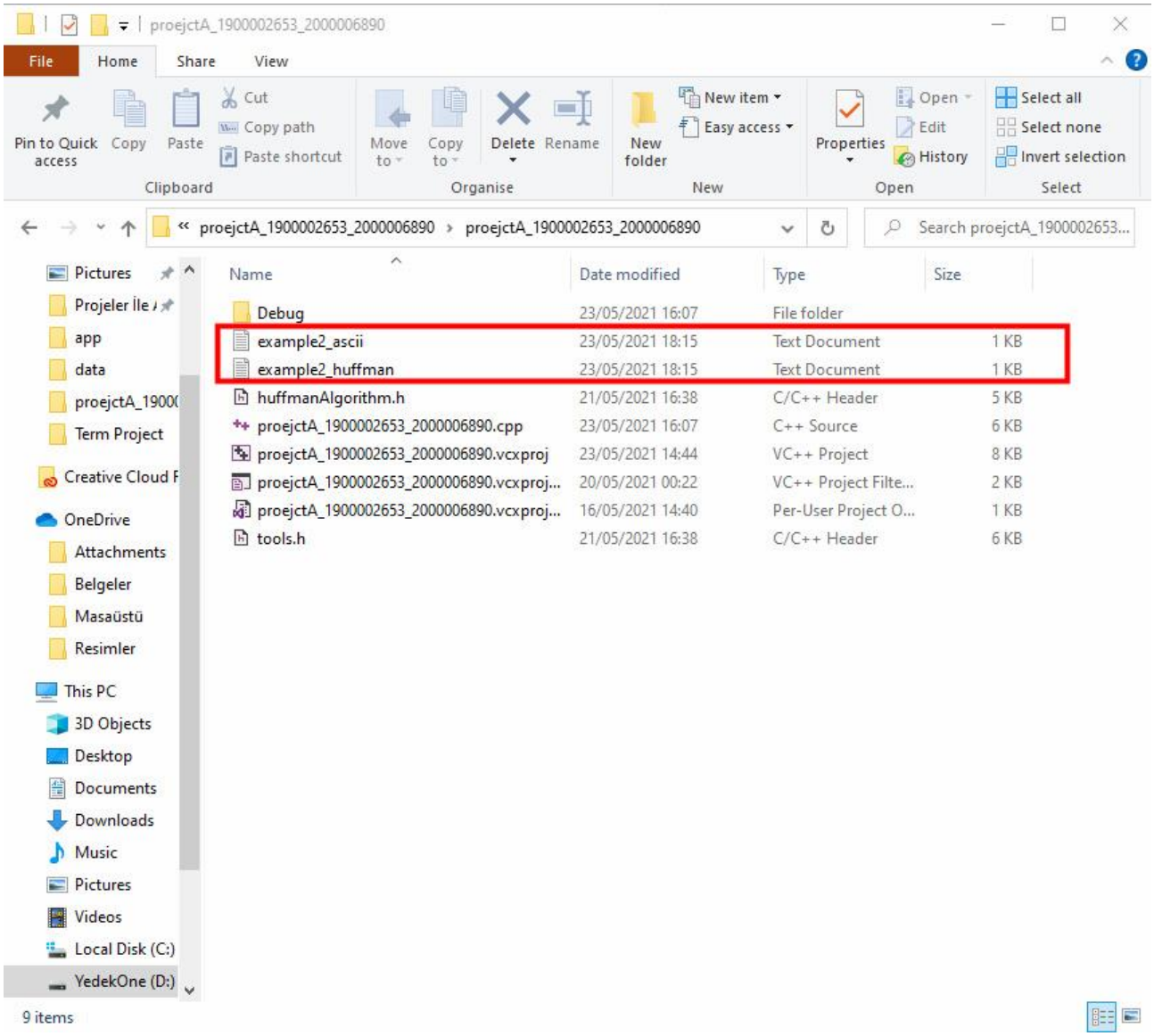
The screen that appears after selecting it asks us where to find the txt file and asks us for the full path of the file. The name of our file is "example2.txt" and the full path to our file is " C:\Users\enesk\Documents\data\example2.txt"

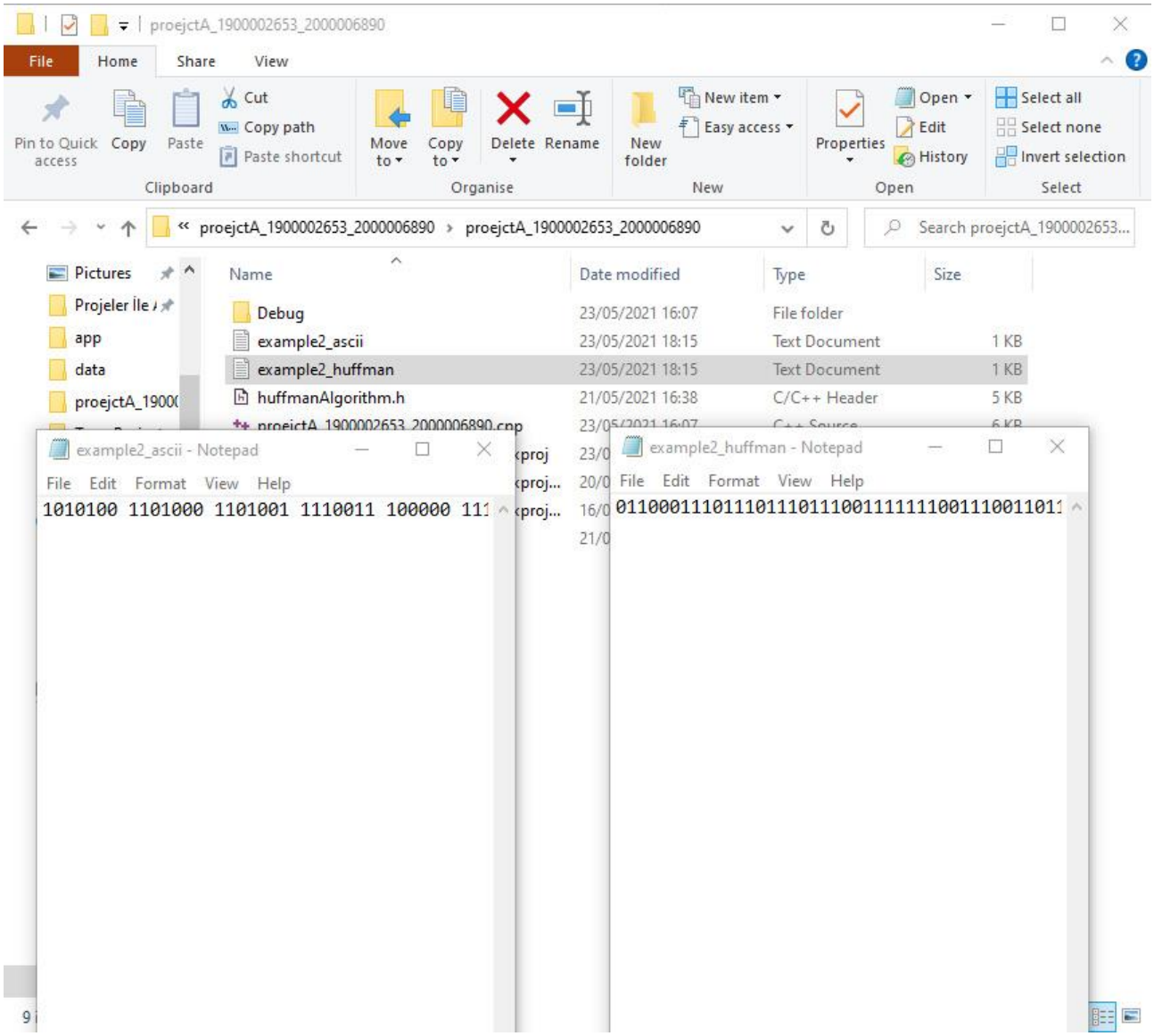


We see our repeat result screen;

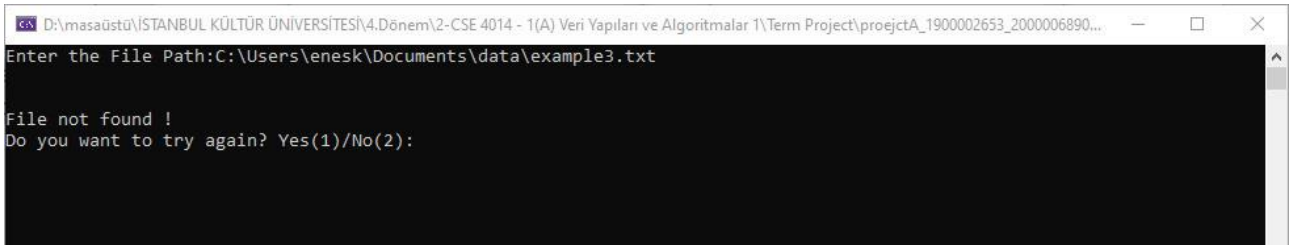


Together with the result screen, the uncompressed, ie Ascii codes and compressed values according to the Huffman Algorithm are presented to the user in the form of a new txt file.

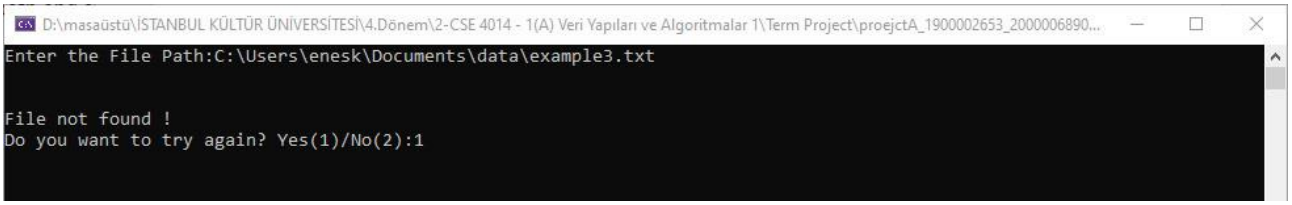

proejctA_1900002653_2000006890
File
Home
Share
View
Pin to Quick access
Copy
Paste
Cut
Copy path
Paste shortcut
Clipboard
Move to
Copy to
Delete
Rename
Organise
New folder
New item
Easy access
New
Properties
Open
Edit
History
Open
Select all
Select none
Invert selection
Select
« proejctA_1900002653_2000006890 › proejctA_1900002653_2000006890
Search proejctA_1900002653...
Pictures
Projeler İle /
app
data
proejctA_1900(
Term Project
Creative Cloud F
OneDrive
Attachments
Belgeler
Masaüstü
Resimler
This PC
3D Objects
Desktop
Documents
Downloads
Music
Pictures
Videos
Local Disk (C:)
YedekOne (D:)
Name
Date modified
Type
Size
Debug 23/05/2021 16:07 File folder
example2_ascii 23/05/2021 18:15 Text Document 1 KB
example2_huffman 23/05/2021 18:15 Text Document 1 KB
huffmanAlgorithm.h 21/05/2021 16:38 C/C++ Header 5 KB
proejctA_1900002653_2000006890.cpp 23/05/2021 16:07 C++ Source 6 KB
proejctA_1900002653_2000006890.vcxproj 23/05/2021 14:44 VC++ Project 8 KB
proejctA_1900002653_2000006890.vcxproj... 20/05/2021 00:22 VC++ Project Filte... 2 KB
proejctA_1900002653_2000006890.vcxproj... 16/05/2021 14:40 Per-User Project O... 1 KB
tools.h 21/05/2021 16:38 C/C++ Header 6 KB
9 items

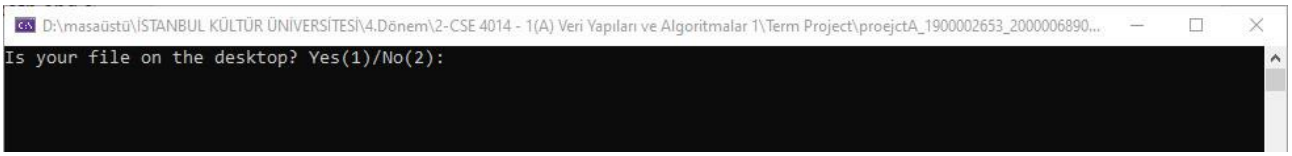


If the txt file is not found, the user is given a chance to try again. The program terminates if the user accepts it and does not want to try again.

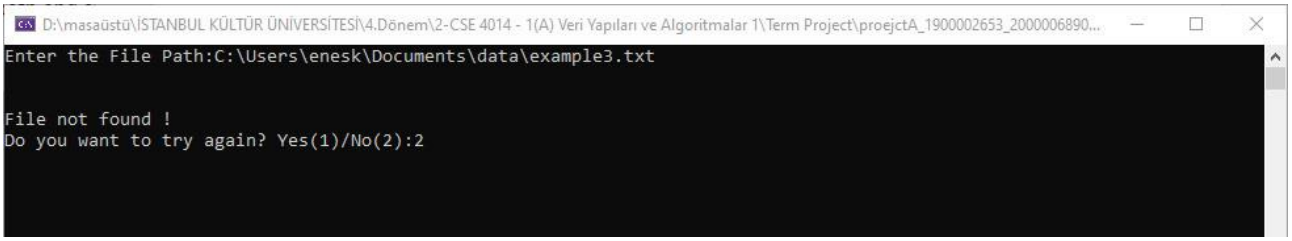


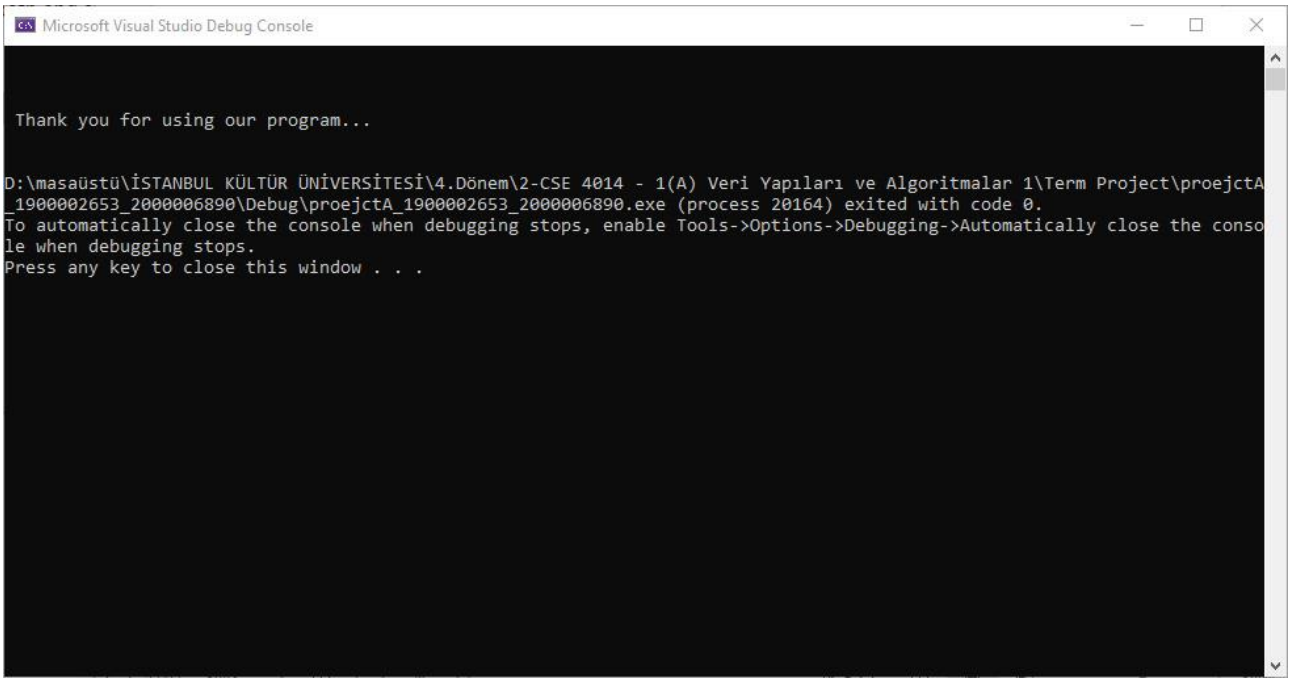
If we select an option and ask them to enter a file path and search again;





If we choose the second option and do not want to use the program;





3. REFERANCES

https://tr.wikipedia.org/wiki/Huffman_kodu

https://www.techiedelight.com/huffman-coding/

https://www.youtube.com/watch?v=DOUKPGK1X-c

https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

https://nirav.com.np/2019/02/14/writing-huffman-compression-in-cpp.html

https://www.codesdope.com/course/algorithms-huffman-codes/