**Practical – 7**

**Practical Statement** : Encode a given input file using dictionary method “LZ77” and store the result in file. Also perform decoding on the same file.

**Code :**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <string.h>

// windowSize = Size of dictionary

// bufferSize = Size of lookahead buffer

// Important: windowSize < 255 & windowSize > bufferSize!

#define windowSize 60

#define bufferSize 40

#define arraySize bufferSize + windowSize

// Data Compression

// Practical 7: 'LZ77' Compression and Decompression Algorithm

typedef enum { false, true } bool;

char file\_name\_path\_[1000];

char file\_output\_name\_path\_[1000];

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

// This method searches for a match from str[] in window[] of strLen length.

// Returns the position of the match starting from the beginning of window[],

// or -1 if no match is found.

// Is invoked during every iteration of the compression algorithm.

int findMatch(unsigned char window[], unsigned char str[], int strLen) {

int i, j, k, pos = -1;

for (i = 0; i <= windowSize - strLen; i++) {

pos = k = i;

for (j = 0; j < strLen; j++) {

if (str[j] == window[k])

k++;

else

break;

}

if (j == strLen)

return pos;

}

return -1;

}

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

// This method contains the logic of the compression algorithm.

// Is invoked when "-c" option is specified in launch command, followed by file name.

int compress() {

FILE \*fileInput;

FILE \*fileOutput;

int k, j, l;

bool last = false;

int inputLength = 0;

int outputLength = 0;

int endOffset = 0;

int pos = -1;

int i, size, shift, c\_in;

size\_t bytesRead = (size\_t) -1;

unsigned char c;

unsigned char array[arraySize];

unsigned char window[windowSize];

unsigned char buffer[bufferSize];

unsigned char loadBuffer[bufferSize];

unsigned char str[bufferSize];

// Open I/O files

char path[30] = "input/";

//strcat(path, inputPath);

fileInput = fopen(file\_name\_path\_, "rb");

fileOutput = fopen(file\_output\_name\_path\_, "wb");

// If unable to open file, return alert

if (!fileInput) {

fprintf(stderr, "Unable to open fileInput %s", file\_name\_path\_);

return 0;

}

// Get fileInput length

fseek(fileInput, 0, SEEK\_END);

inputLength = ftell(fileInput);

fseek(fileInput, 0, SEEK\_SET);

fprintf(stdout, "Input file size: %d bytes", inputLength);

// If file is empty, return alert

if (inputLength == 0)

return 3;

// If file length is smaller than arraySize, not worth processing

if (inputLength < arraySize)

return 2;

// Load array with initial bytes

fread(array, 1, arraySize, fileInput);

// Write the first bytes to output file

fwrite(array, 1, windowSize, fileOutput);

// LZ77 logic beginning

while (true) {

if ((c\_in = fgetc(fileInput)) == EOF)

last = true;

else

c = (unsigned char) c\_in;

// Load window (dictionary)

for (k = 0; k < windowSize; k++)

window[k] = array[k];

// Load buffer (lookahead)

for (k = windowSize, j = 0; k < arraySize; k++, j++) {

buffer[j] = array[k];

str[j] = array[k];

}

// Search for longest match in window

if (endOffset != 0) {

size = bufferSize - endOffset;

if (endOffset == bufferSize)

break;

}

else {

size = bufferSize;

}

pos = -1;

for (i = size; i > 0; i--) {

pos = findMatch(window, str, i);

if (pos != -1)

break;

}

// No match found

// Write only one byte instead of two

// 255 -> offset = 0, match = 0

if (pos == -1) {

fputc(255, fileOutput);

fputc(buffer[0], fileOutput);

shift = 1;

}

// Found match

// offset = windowSize - position of match

// i = number of match bytes

// endOffset = number of bytes in lookahead buffer not to be considered (EOF)

else {

fputc(windowSize - pos, fileOutput);

fputc(i, fileOutput);

if (i == bufferSize) {

shift = bufferSize + 1;

if (!last)

fputc(c, fileOutput);

else

endOffset = 1;

}

else {

if (i + endOffset != bufferSize)

fputc(buffer[i], fileOutput);

else

break;

shift = i + 1;

}

}

// Shift buffers

for (j = 0; j < arraySize - shift; j++)

array[j] = array[j + shift];

if (!last)

array[arraySize - shift] = c;

if (shift == 1 && last)

endOffset++;

// If (shift != 1) -> read more bytes from file

if (shift != 1) {

// Load loadBuffer with new bytes

bytesRead = fread(loadBuffer, 1, (size\_t) shift - 1, fileInput);

// Load array with new bytes

// Shift bytes in array, then splitted into window[] and buffer[] during next iteration

for (k = 0, l = arraySize - shift + 1; k < shift - 1; k++, l++)

array[l] = loadBuffer[k];

if (last) {

endOffset += shift;

continue;

}

if (bytesRead < shift - 1)

endOffset = shift - 1 - bytesRead;

}

}

// Get fileOutput length

fseek(fileOutput, 0, SEEK\_END);

outputLength = ftell(fileOutput);

fseek(fileOutput, 0, SEEK\_SET);

fprintf(stdout, "\nOutput file size: %d bytes\n", outputLength);

// Close I/O files

fclose(fileInput);

fclose(fileOutput);

return 1;

}

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

// This method contains the logic of the inverse algorithm, used to decompress.

// Is invoked when "-d" option is specified in launch command.

int decompress() {

FILE \*fileInput;

FILE \*fileOutput;

int shift, offset, match, c\_in;

bool done = false;

int i, j;

unsigned char c;

unsigned char window[windowSize];

unsigned char writeBuffer[windowSize];

unsigned char readBuffer[2];

// Open I/O files

fileInput = fopen(file\_name\_path\_, "rb");

fileOutput = fopen(file\_output\_name\_path\_, "wb");

if (!fileInput) {

fprintf(stderr, "Unable to open fileInput %s", file\_name\_path\_);

return 0;

}

// Load array with initial bytes and write to file

fread(window, 1, windowSize, fileInput);

fwrite(window, 1, windowSize, fileOutput);

// Inverse algorithm beginning

while (true) {

// Read file by couples/triads to reconstruct original file

size\_t bytesRead = fread(readBuffer, 1, 2, fileInput);

if (bytesRead >= 2) {

offset = (int) readBuffer[0];

match = (int) readBuffer[1];

// If first byte of readBuffer is 255 -> offset = 0, match = 0

if (offset == 255) {

offset = 0;

c = (unsigned char) match;

match = 0;

shift = match + 1;

}

else {

shift = match + 1;

c\_in = fgetc(fileInput);

if (c\_in == EOF)

done = true;

else

c = (unsigned char) c\_in;

}

// Load and write occurrence to file

for (i = 0, j = windowSize - offset; i < match; i++, j++)

writeBuffer[i] = window[j];

fwrite(writeBuffer, 1, (size\_t) match, fileOutput);

if (!done)

fputc(c, fileOutput);

// Shift window

for (i = 0; i < windowSize - shift; i++)

window[i] = window[i + shift];

for (i = 0, j = windowSize - shift; i < match; i++, j++)

window[j] = writeBuffer[i];

window[windowSize - 1] = c;

}

else {

break;

}

}

// Close I/O files

fclose(fileInput);

fclose(fileOutput);

return 1;

}

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

// This method is the controller, reads user inputs.

// Is invoked on program launch.

int main(int argc, char\* argv[]) {

clock\_t begin;

char d\_or\_c;

printf("\n190420116071 : Meet Vaghasiya\n\nPractical No. 7: Encode a given input file using directory method \"LZ77\" and store the result in file. Also perform decoding on the same file.\n\n");

printf("\n\nThis is file compression and decompression LZ77 algorithm.\n\n");

printf("Enter your file name or file path(if in different directory): ");

scanf("%s", file\_name\_path\_);

again1:

printf("\n\nDo you want to compress or decompress?('c' or 'd'): ");

scanf(" %c", &d\_or\_c);

//printf("\nThe File name is: %s", file\_name\_path\_);

if (d\_or\_c!='c' && d\_or\_c!='d') {

printf("Please enter a correct choice('c' or 'd'): ");

goto again1;

} else {

// Start decompression

printf("Enter your file name or file path(if in different directory) for OUTPUT: ");

scanf("%s", file\_output\_name\_path\_);

begin = clock();

if (d\_or\_c=='d') {

int result = decompress();

if (result == 0) {

fprintf(stderr, "\nDecompression FAIL");

} else if (result == 1) {

printf("\nDecompression OK");

}

}

// Start compression

else if (d\_or\_c == 'c') {

int result = compress();

if (result == 0) {

fprintf(stderr, "\nCompression FAIL\n");

} else if (result == 1) {

printf("\nCompression OK");

} else if (result == 2) {

fprintf(stderr, "\nFile too small\n");

} else if (result == 3) {

fprintf(stderr, "\nFile is EMPTY\n");

}

} else {

printf("Invalid arguments");

}

}

// Print execution time

clock\_t end = clock();

printf("\n\nExecution time: ");

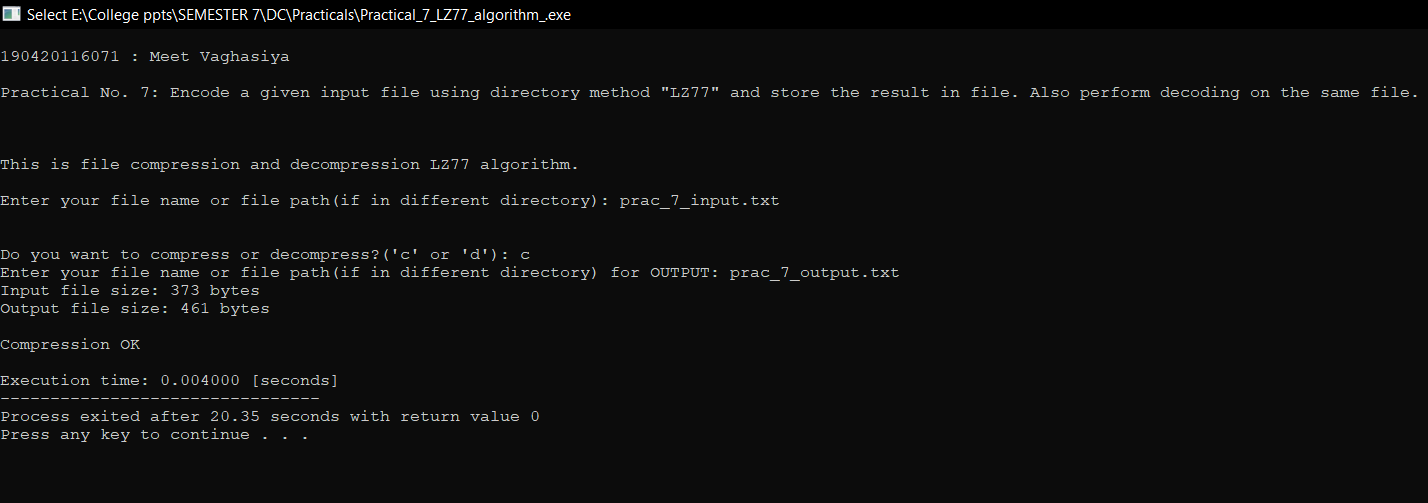
printf("%f", ((double) (end - begin) / CLOCKS\_PER\_SEC));

printf(" [seconds]");

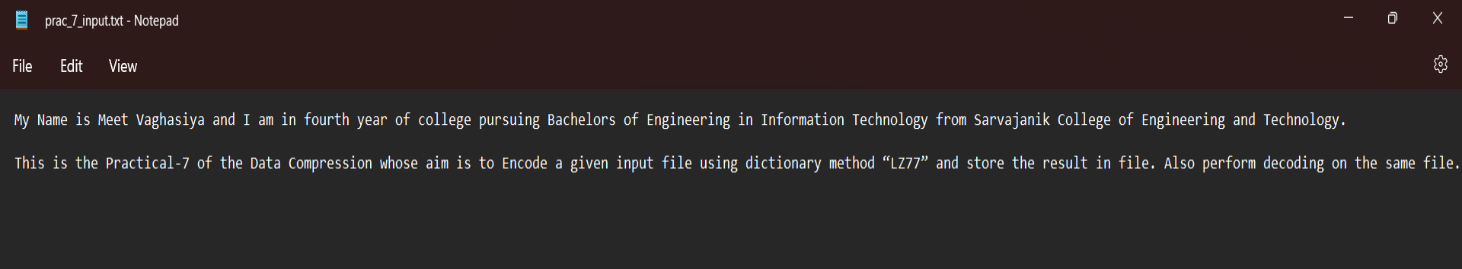
return 0;

}

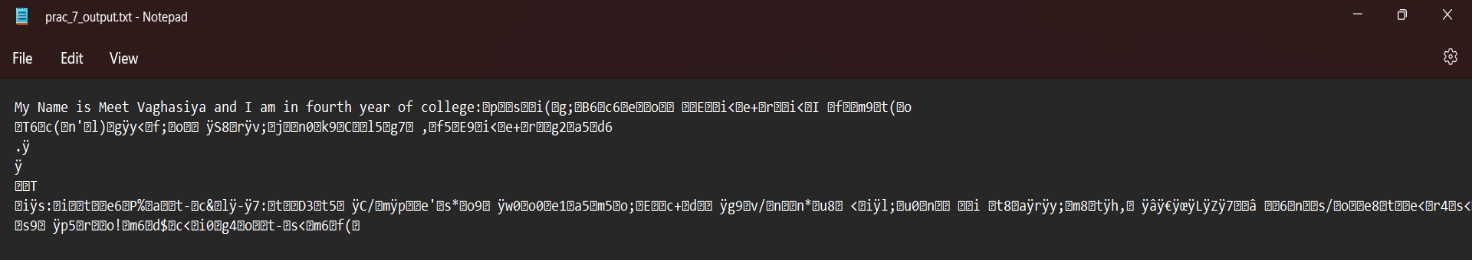
Output-1 : (Compression)



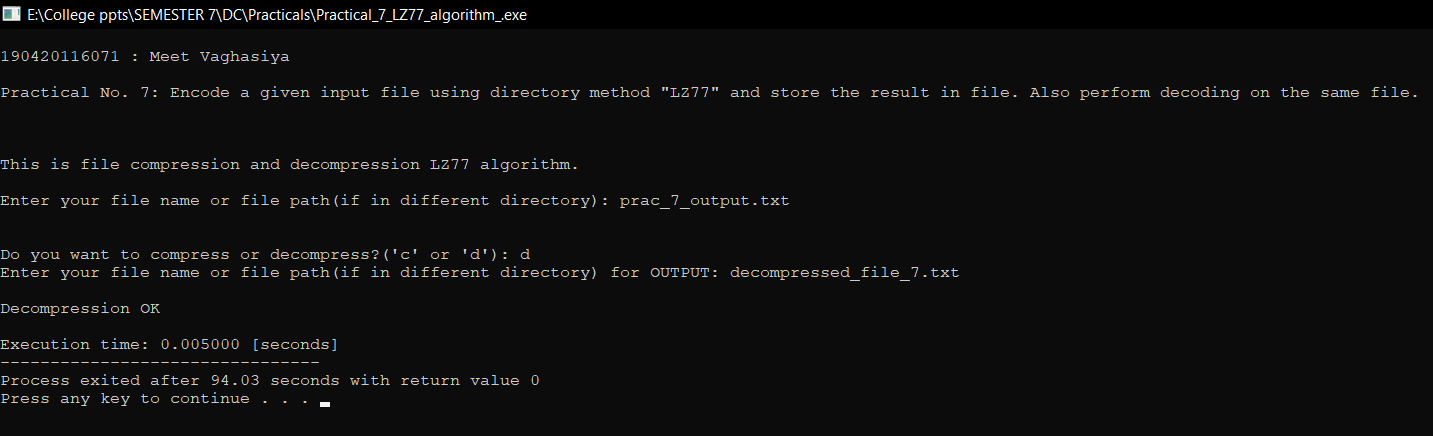
Input File (prac\_7\_input.txt):



Compressed File (prac\_7\_output.txt):



Output-2 : (Decompression)



Decompressed File (decompressed\_file\_7.txt):

