Instituto Politécnico Nacional
Escuela Superior de Cómputo

Analisis de algoritmos, Sem: 2021-1, 3CV1,Práctica 5, 11/12/2020

Práctica 5: Algoritmos Greedy

Payán Téllez René
rpayant1500@alumno.ipn.mx

Resumen: En esta practica se analizaran 2 algoritmos que utilizan la tecnica de divide y venceras para resolver un problema, uno de ellos el quick sort y otro el sub arreglo maximo.
Palabras clave: Huffman, Kruskal, Gredy, STL, C/C++

1 Introduccion

Los algoritmos glotones, son en su mayoria utilizados para resolver problemas extremadamente complejos de forma simple de programar, normalmente no es la mejor solucion pero casi siempre bastante cercana y se programa de forma extremadamente rapida, sin mencionar que normalmente no toman mucha complejidad computacional. Por ello aprender respecto a estos algoritmos es extremadamente importante, inclusive de los 2 que vamos a hablar a continuacion, ya que tienen usos tanto en redes (Huffman) como en proyectos (Kruskal).

2 Conceptos Basicos

2.1 Algoritmo

La palabra algoritmo proviene del sobrenombre de un matemático árabe del siglo IX, Al-Khwarizmi, que fue reconocido por enunciar paso a paso las reglas para las operaciones matemáticas básicas con decimales (suma, resta, multiplicación y división). Vemos definición de algoritmo como un grupo de órdenes consecutivas que presentan una solución a un problema o tarea. Algunos ejemplos de algoritmos los podemos encontrar en las matemáticas (como el algoritmo para resolver una multiplicación) y en los manuales de usuario de un aparato (como una lavadora o una impresora). Sin embargo, hoy en día se relaciona la palabra algoritmo con el mundo de la informática, más concretamente en la programación; los conocidos como algoritmos informáticos.[1]

2.2 Complejidad algoritmica

Así que, por su naturaleza, un problema tiene la capacidad de ser solucionado por uno o varios métodos, pero si bien es importante llegar a la respuesta, más importante es evaluar su viabilidad. Siempre que se analiza y evalúa adecuadamente la efectividad de una solución, disminuye drásticamente el costo que representa su producción y mantenimiento, pues los recursos que se invierten posteriormente en codificación, pruebas y revisión es mucho menor siempre (como el tiempo, dinero y talento humano). Entrando en materia, la complejidad algorítmica es una métrica teórica que nos ayuda a describir el comportamiento de un algoritmo en términos de tiempo de ejecución (tiempo que tarda un algoritmo en resolver un problema) y memoria requerida (cantidad de memoria necesaria para procesar las instrucciones que solucionan dicho problema). Esto nos ayuda a comparar entre la efectividad de un algoritmo y otro, y decidir cuál es el que nos conviene implementar.[2]

2.3 Algoritmos Greedy o glotones

Un algoritmo Greedy o gloton es un algoritmo muy util para encontrar soluciones aproximadas e inclusive la mas optima a problemas complejos, ya que las entregan en muy corto tiempo. Se llaman Greedy porque siempre "comen lo que tienen a la mano", no garantizan encontrar la mejor solucion, pero si una aproximacion bastante buena.[3] Estas son sus caracteristicas principales:

- Se utilizan generalmente para resolver problemas deoptimización (obtener el máximo o el mínimo).optimización (obtener el máximo o el mínimo).
- Toman decisiones en función de la información que está disponible en cada momento. está disponible en cada momento. está disponible en cada momento. está disponible en cada momen
- Una vez tomada la decisión, ésta no vuelve a replantearse en el futuro.replantearse en el futuro.
- Suelen ser rápidos y fáciles de implementar.
- No siempre garantizan alcanzar la solución óptima[4]

2.4 Algoritmo de codificación de Huffman

El código de Huffman es un tipo particular de código de prefijo óptimo que se usa comúnmente para la compresión de datos sin pérdida. Comprime los datos de manera muy efectiva, ahorrando de 20% a 90% de memoria, dependiendo de las características de los datos comprimidos. Este algoritmo se aplica solo si se considera a la entrada como una cadena de caracteres, ya que es un algoritmo boraz que utiliza una tabla que proporciona la frecuencia con la que aparece cada carácter (es decir, su frecuencia) para crear una forma óptima de representar cada carácter como una cadena binaria. Fue propuesto por David A. Huffman en 1951.[5]

Algoritmo 1: HuffmanTree(C)

```
Data: Entrada: C (Los caracteres de la cadena a codificar y su ocurrencia)
Result: Retorna el arbol de huffman de la cadena
n = C.size;
Q = priority_queue();
for i ← 0 to i < n do
    n=node(C[i]);
    Q.push(n);
while Q.size()>1 do
    Z = new node();
    Z.left = x = Q.pop();
    Z.right = y = Q.pop();
    Z.frequency = x.frequency+y.frequency;
    Q.push(Z);
return Q;
```

Algoritmo 2: HuffmanDecompression(tree, S)

```
Data: Entrada: tree (El arbol de Huffman), S (la cadena binaria a descomprimir)
Result: Retorna una cadena de caracteres descomprimida
n = S.length;
retorno = "";
for i ← 0 to i < n do
    current = root;
    while current.left != NULL and current.right!=NULL do
        if S[i] == '0' then
            current = current.left;
        else
            current = current.right;
        i+=1;
    retorno+=current.symbol;
return retorno;
```

2.5 Algoritmos de Kruskal

El algoritmo de Kruskal es un algoritmo de la teoría de grafos para encontrar un árbol recubridor mínimo en un grafo conexo y ponderado. Es decir, busca un subconjunto de aristas que, formando un árbol, incluyen todos los vértices y donde el valor de la suma de todas las aristas del árbol es el mínimo. Si el grafo no es conexo, entonces busca un bosque expandido mínimo (un árbol expandido mínimo para cada componente conexa).[6]
Kruskal funciona de la siguiente forma:

1. Se selecciona, de entre todas las aristas restantes, la de menor peso siempre que no cree ningún ciclo.
2. Se repite el paso 1 hasta que se hayan seleccionado $\|V\| - 1$ aristas.[7]

Data: Entrada: G (un grafo)
Result: Retorna el arbol de extencion minima
foreach v **to** *V[G]* **do**
  Z = union_find(C[v]**to** v);
Q = priority_queue(A[G]);
T = new Tree();
while *A[T]* $\leq$ *n-1 and !Q.empty()* **do**
  (u,v) **to** Q.sacarMin(); **if** *C(v)* $\neq$ *C(u)* **then**
    T.push((u,v));
    Merge(C(v), C(u));
return T;

Su complejidad es $O(mlog_2(n))$[8], donde m es el numero de aristas y n el numero de nodos.

3 Experimentacion y Resultados

3.1 Implementar el algoritmo de codificacion de Huffman.

Para esta primera parte de pruebas se implemento el algoritmo de codificacion y decodificacion de Huffman, asi que la ejecución se divide en 2 partes, la compresion de una cadena de texto y para la segunda parte la descompresion de la misma

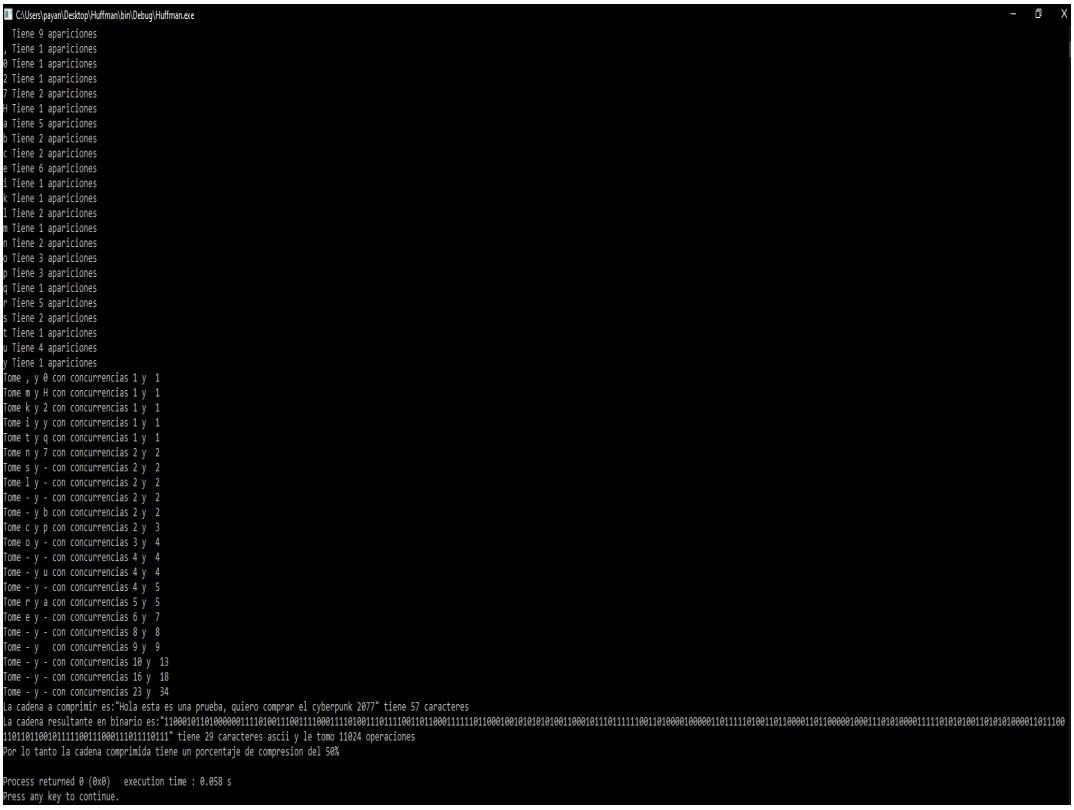

Figure 1: Compresión de la cadena "Hola esta es una prueba, quiero comprar el cyberpunk 2077"

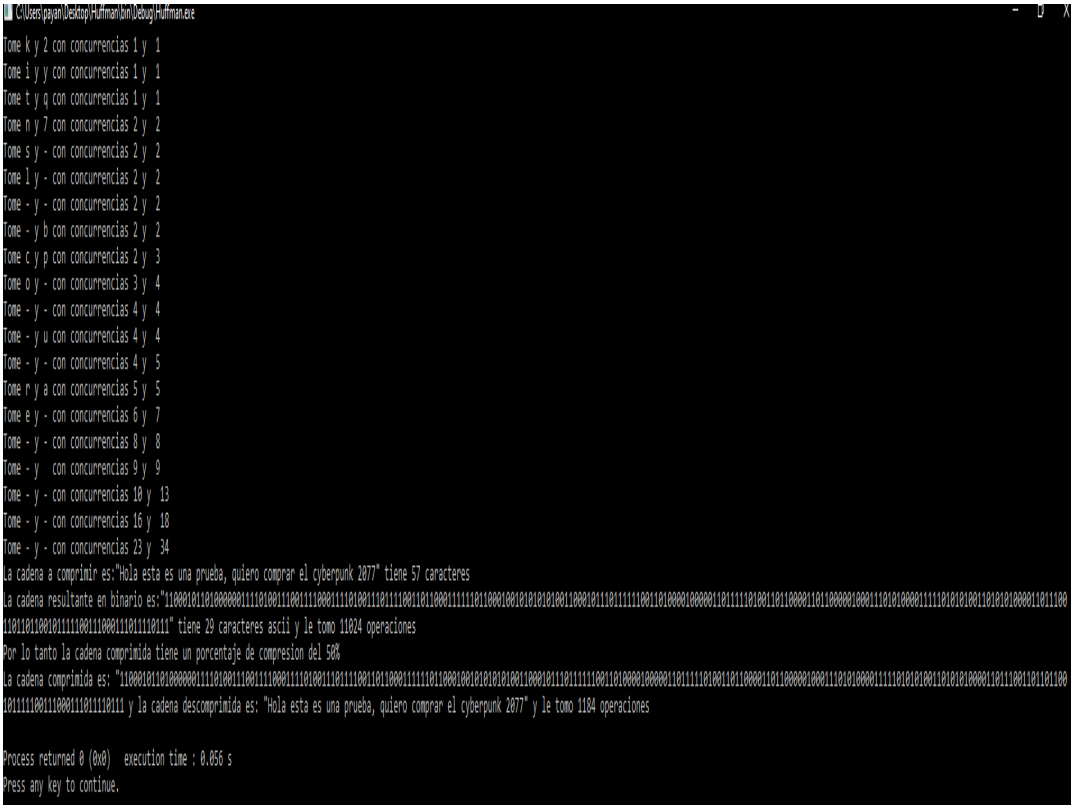

Figure 2: Descompresión de la cadena "Hola esta es una prueba, quiero comprar el cyberpunk 2077"

Como se pudo apreciar en las capturas, la capturas, el programa tomo la entrada "Hola esta es una prueba, quiero comprar el cyberpunk 2077" y retorno la siguiente salida como respuesta: "", posteriormente al introducir la salida de la compresion en la seccion de descompresion, el programa arrojo la cadena original. Podemos decir que el algoritmo es eficiente, porque esta es la cantidad de caracteres de la cadena original:

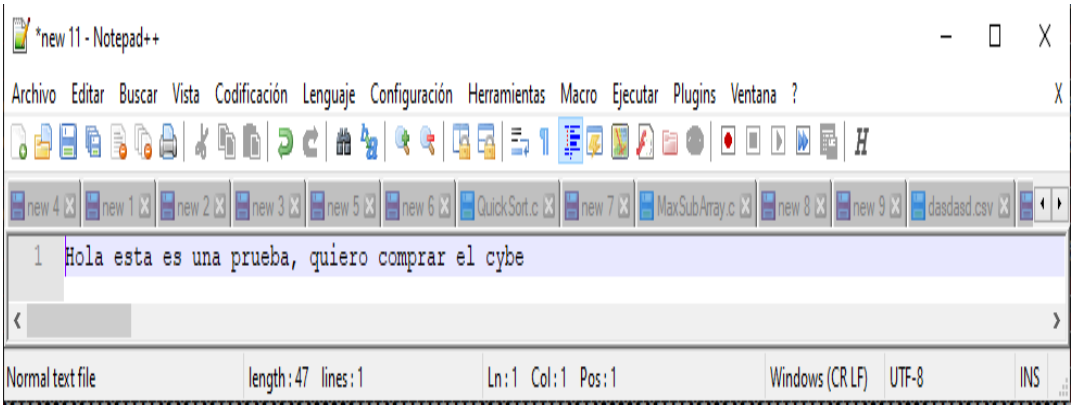

Figure 3: Conteo de caracteres de la entrada

esta es la cantidad de caracteres de la salida:

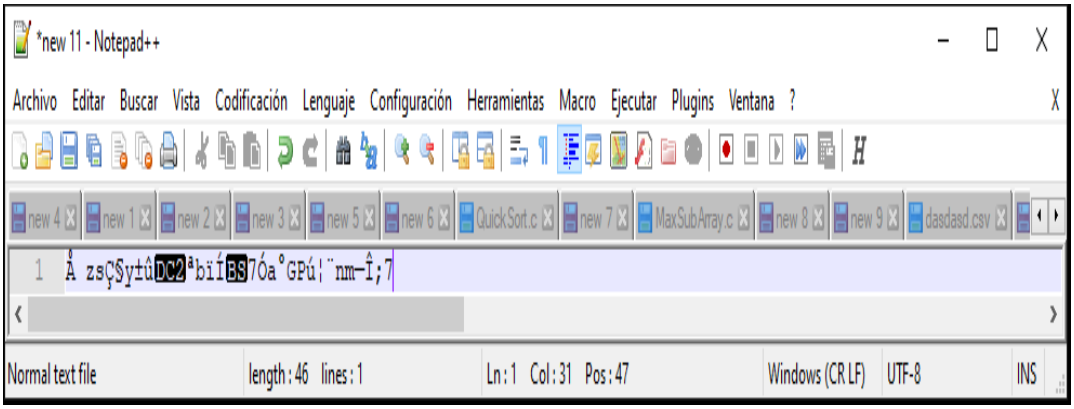

Figure 4: Conteo de caracteres de la salida

por ultimo, el binario de cada caracter asignado

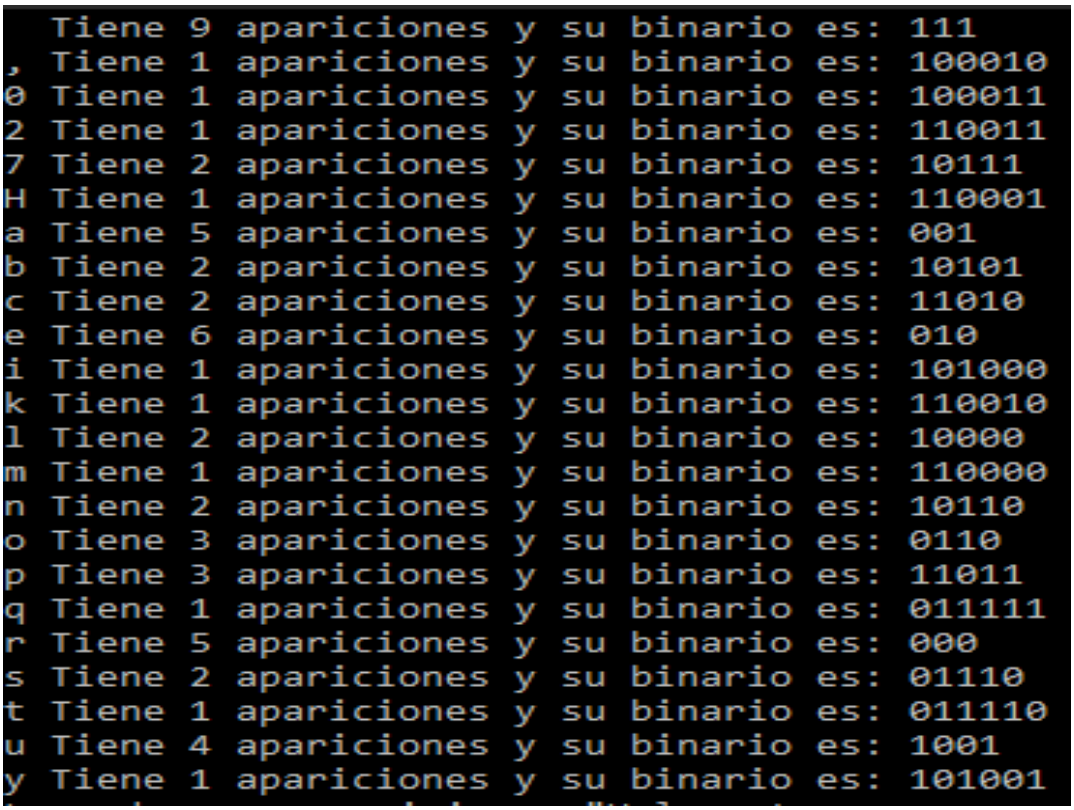

Figure 5: Asignacion de binario

A continuacion se probo con una cadena mas larga, un "Lorem Ipsum" de 1000 bytes: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer porttitor turpis eget erat auctor varius. Vestibulum maximus scelerisque dui ac vulputate. Mauris eleifend mauris vel ex sodales, ut blandit odio dictum. Ut efficitur eu lectus nec ullamcorper. Integer hendrerit justo in augue consectetur, eget porttitor quam rutrum. Cras porta justo at fermentum condimentum. Nunc lacinia convallis tortor in tempor. Donec tincidunt tempor ipsum, sit amet aliquet justo semper at. Donec nibh urna, faucibus ac mauris eu, mattis imperdiet dolor. Nam et nulla at nisi efficitur efficitur. Morbi bibendum scelerisque risus, at sollicitudin ligula rutrum et. Cras sagittis eget velit aliquam cursus. Nunc magna metus, ullamcorper sed ante id, tincidunt ornare libero. Proin pharetra orci felis, eget

pellentesque nisi venenatis eget. Morbi tincidunt ut risus non iaculis. Proin id consectetur metus, sed placerat eros. Aenean cursus augue a ipsum tempor interdum. Vivamus viverra efficitur felis a aenean.

Figure 6: Compresión del lorem ipsum

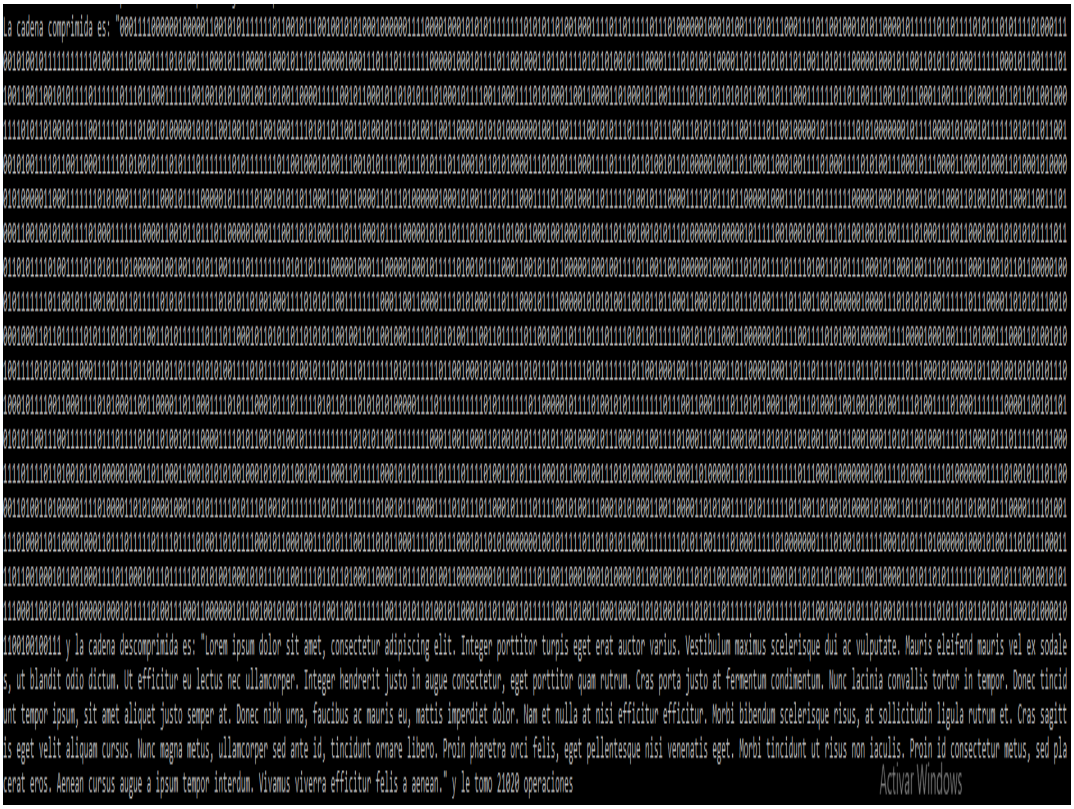

Figure 7: Descompresión del lorem ipsum

Finalmente se contaron los caractres de la entrada y de la salida:

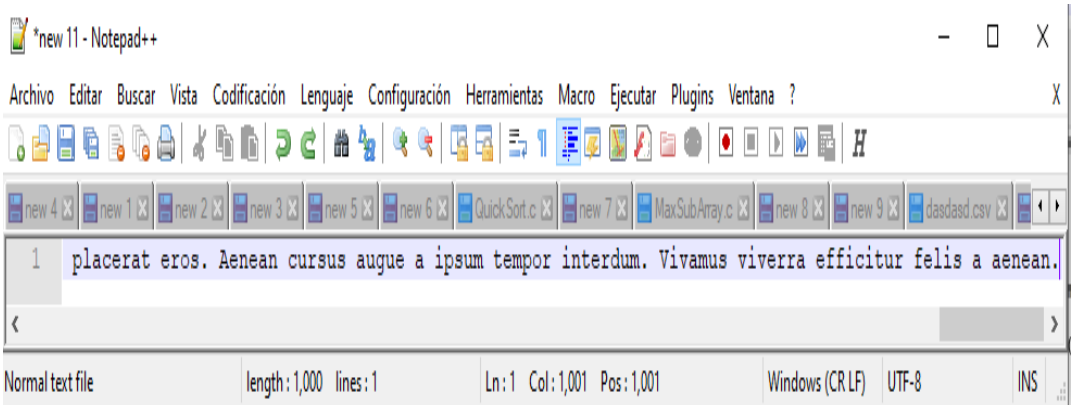

Figure 8: Conteo de caracteres de la entrada

y esta es la cantidad de caracteres de la salida:

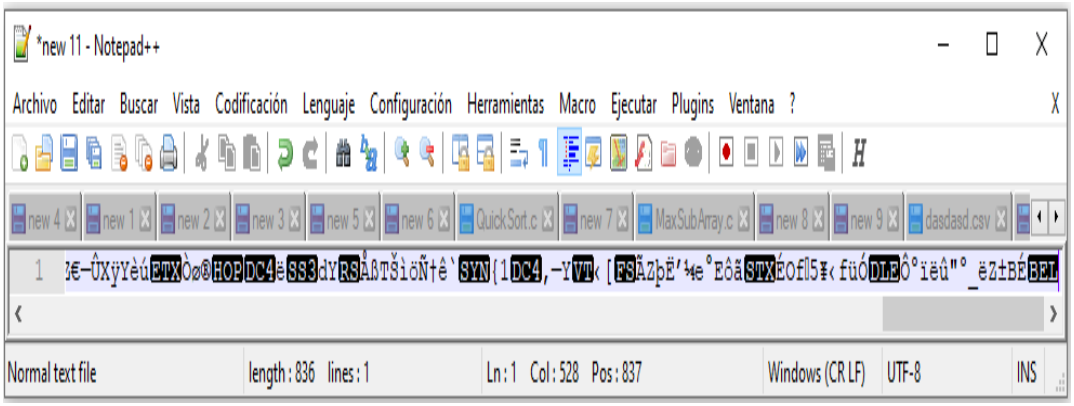

Figure 9: Conteo de caracteres de la salida

por ultimo, el binario de cada caracter asignado

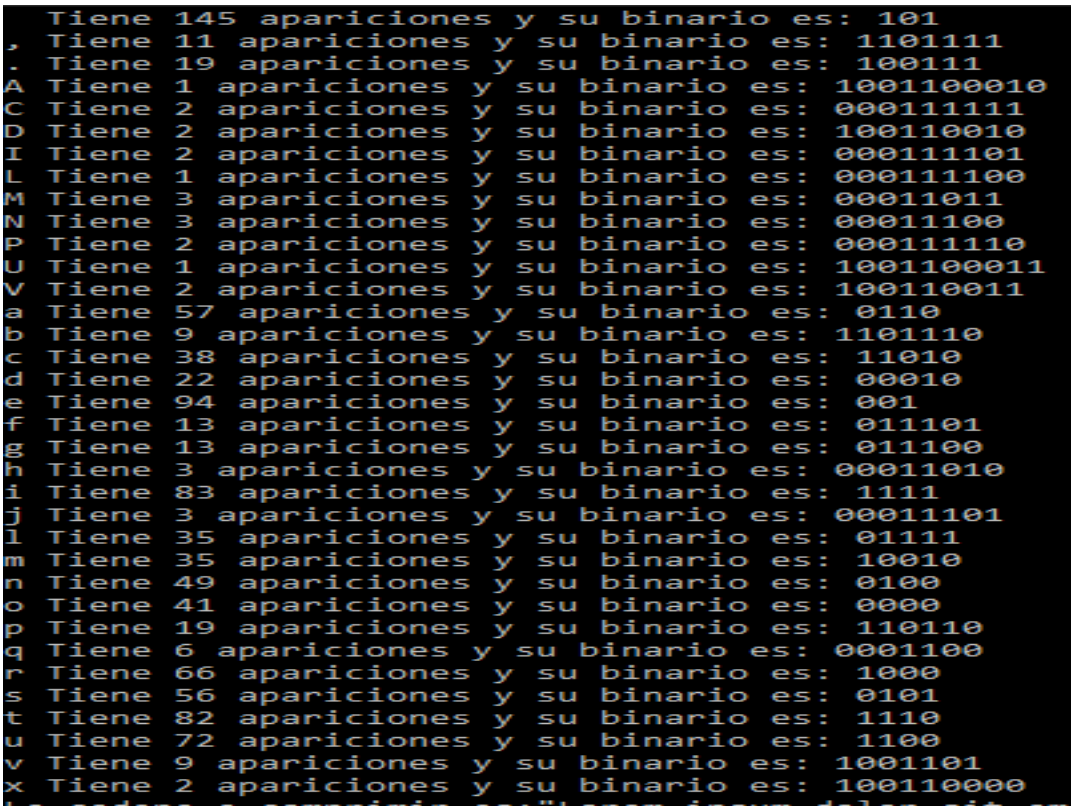

Figure 10: Asignacion de binario

A continuación se comprimieron 10 archivos de texto con extencion .txt de distinto tamaño, se anexan las capturas del binario de cada caracter, del archivo de entrada, el archivo comprimido, el archivo descomprimido y el tamaño de ambos archivos.

Primer archivo

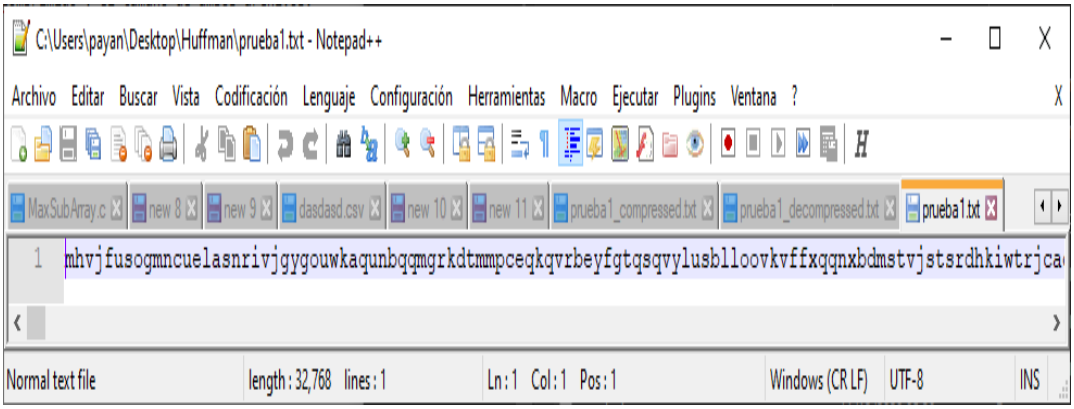

Figure 11: Archivo de entrada

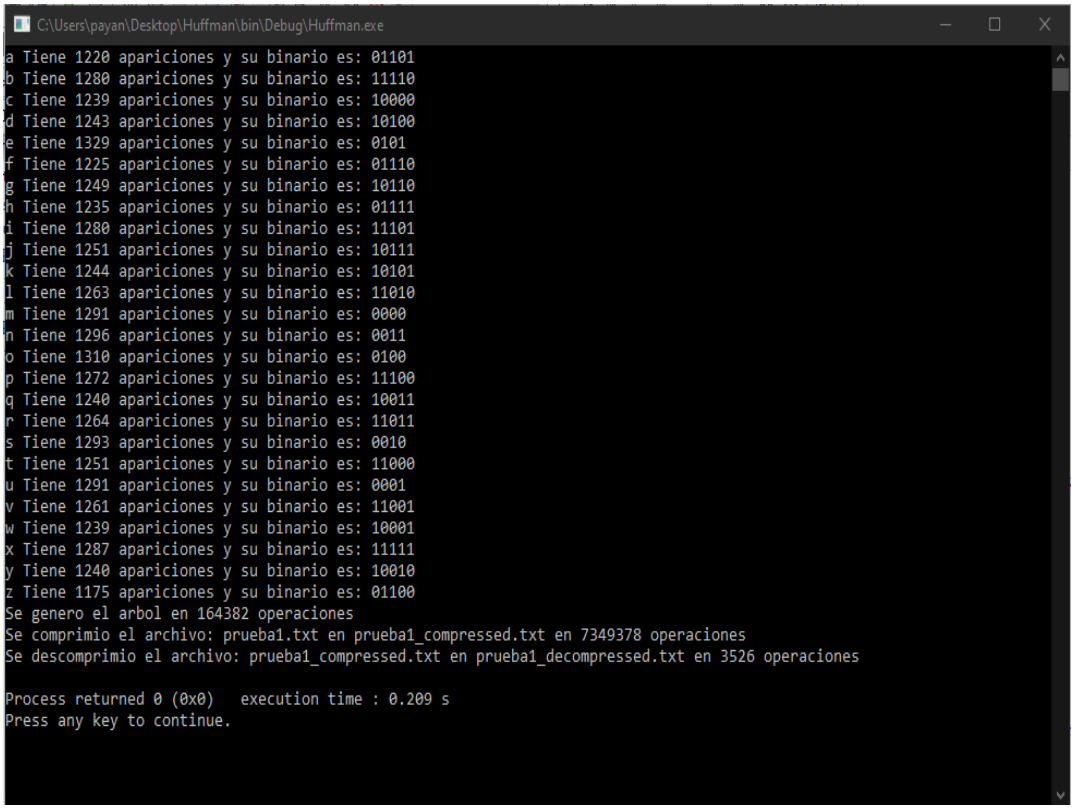

Figure 12: Ejecucion del programa

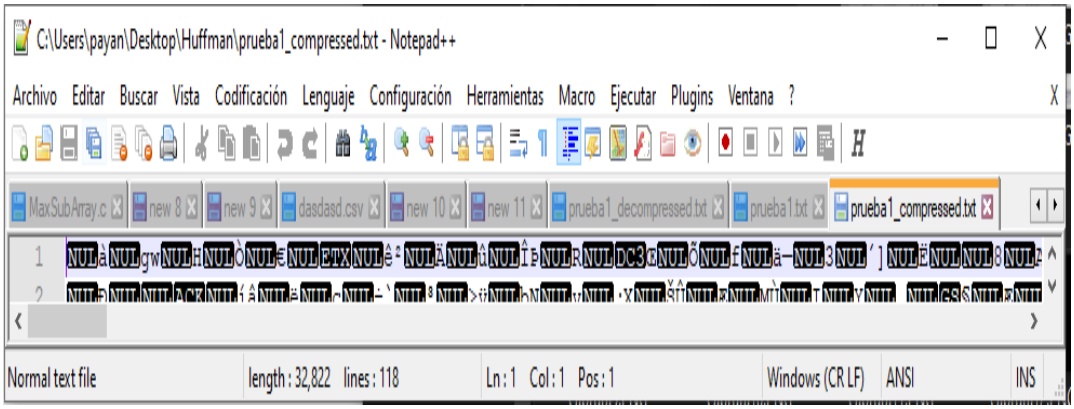

Figure 13: Archivo de salida

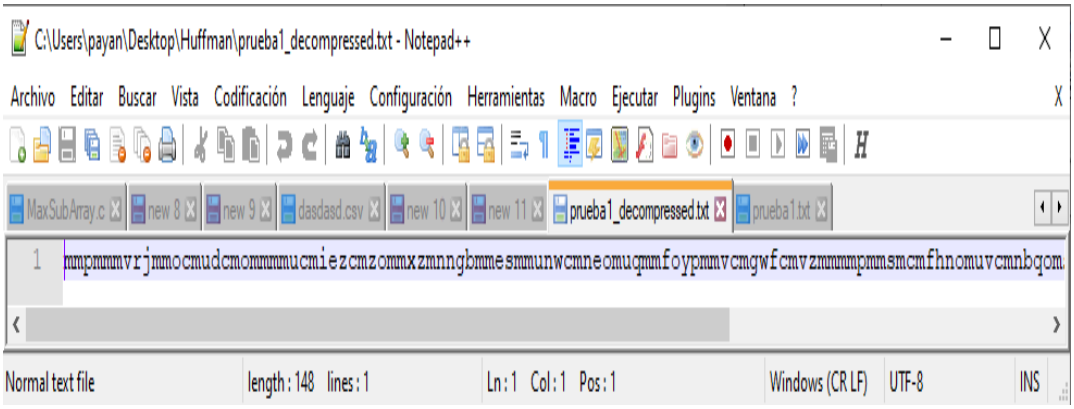

Figure 14: Archivo descomprimido

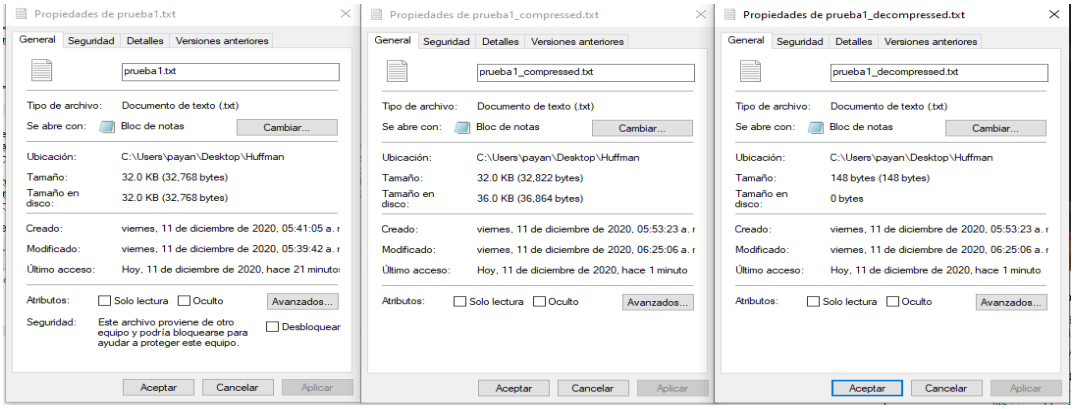

Figure 15: peso de los 3 archivos

Segundo archivo

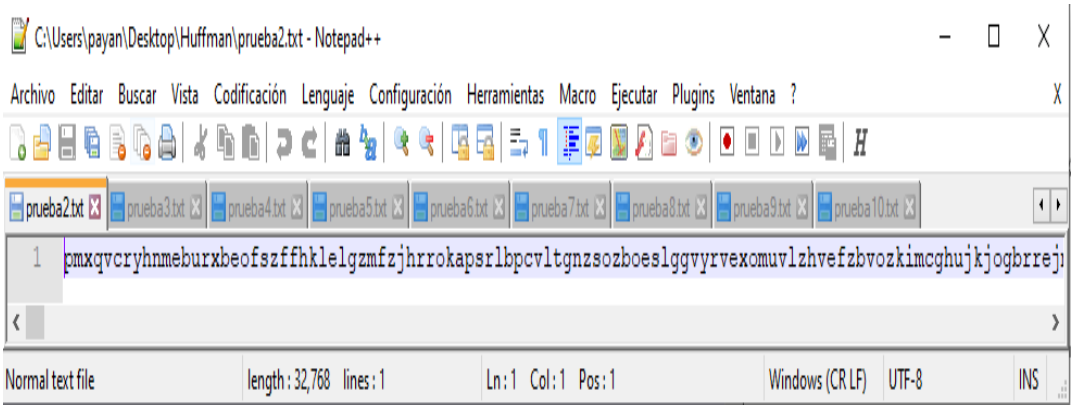

Figure 16: Archivo de entrada

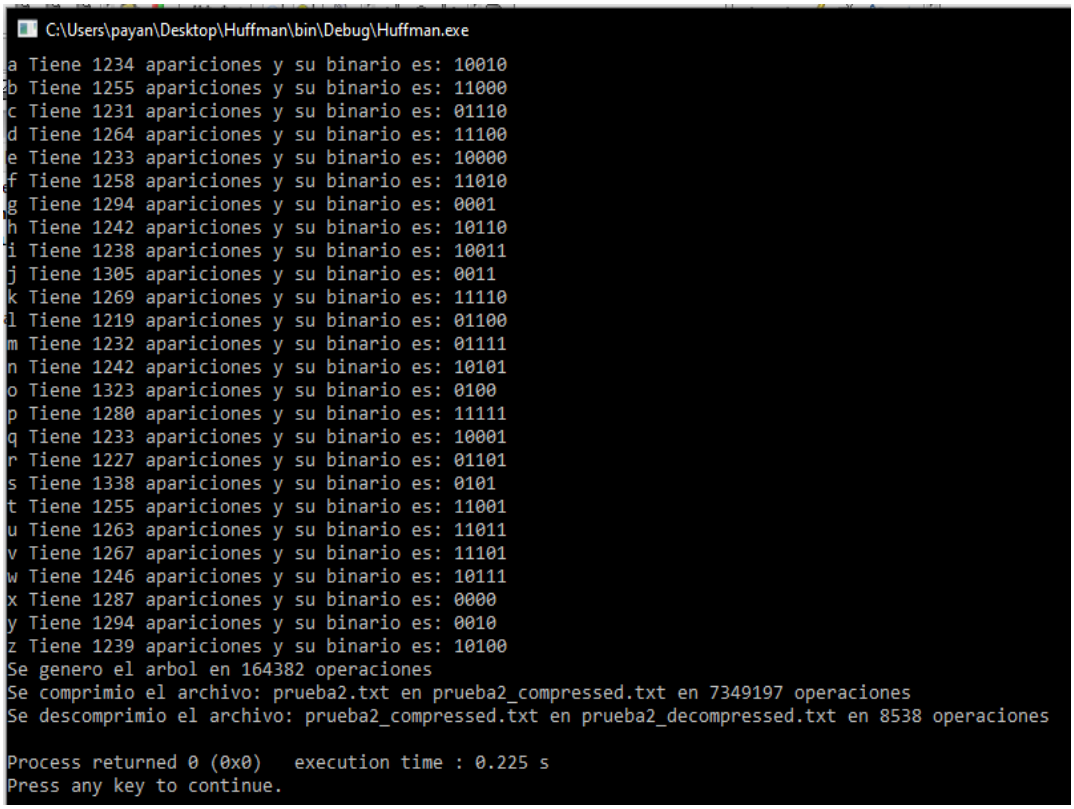

Figure 17: Ejecucion del programa

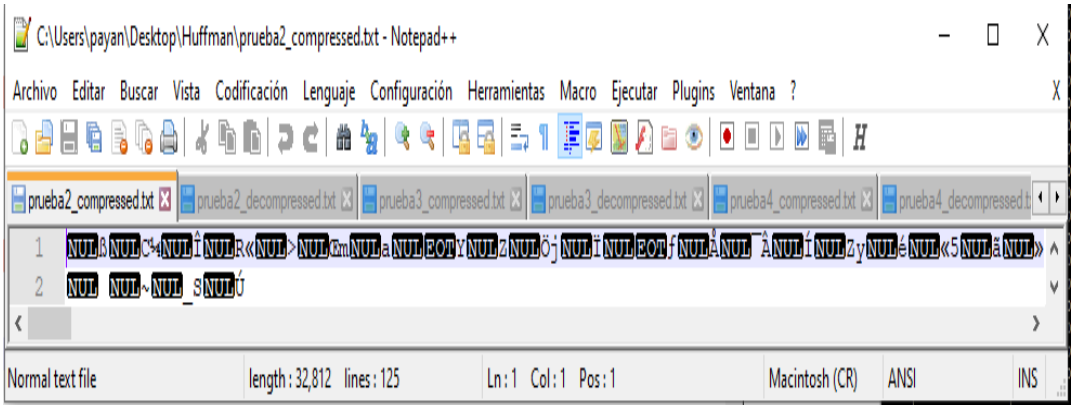

Figure 18: Archivo de salida

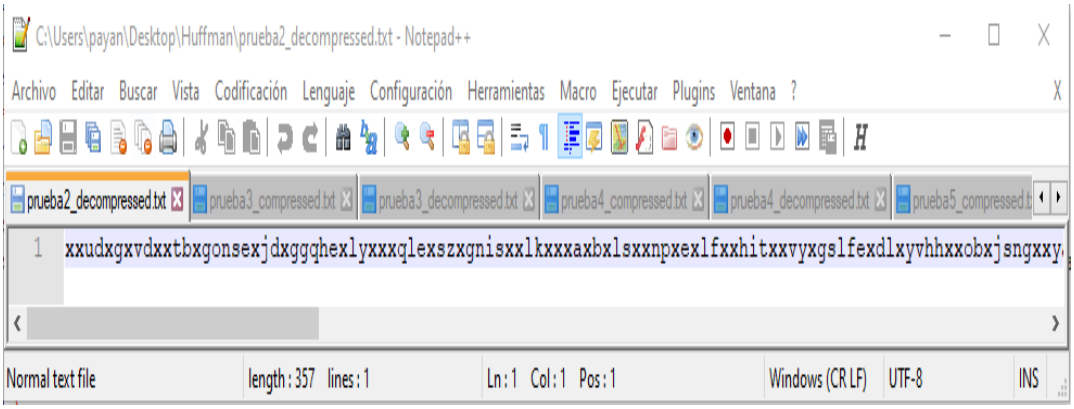

Figure 19: Archivo descomprimido

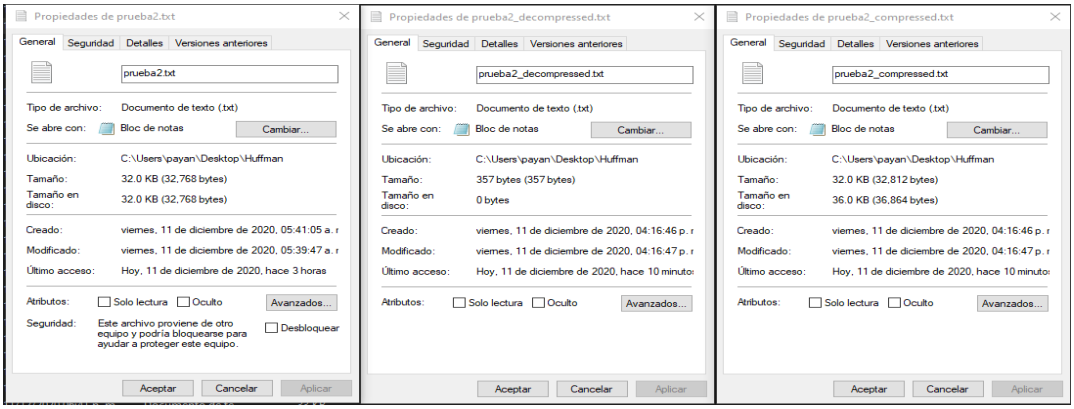

Figure 20: peso de los 3 archivos

Tercer archivo

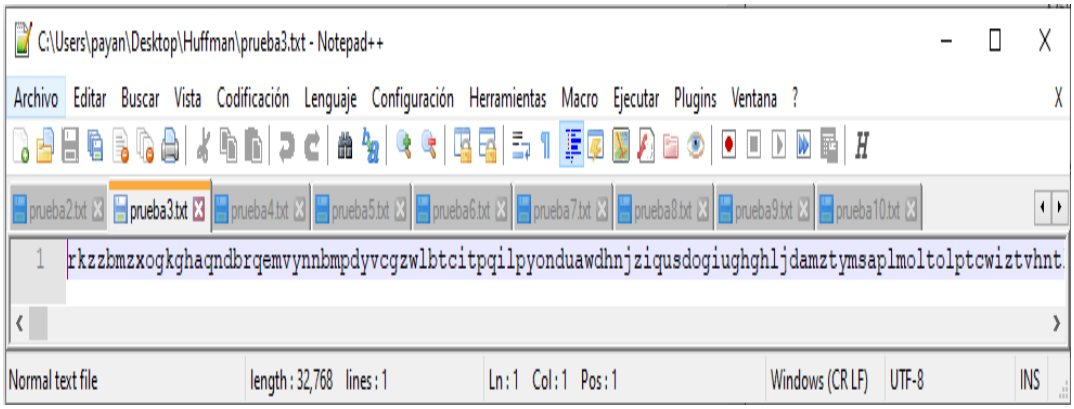

Figure 21: Archivo de entrada

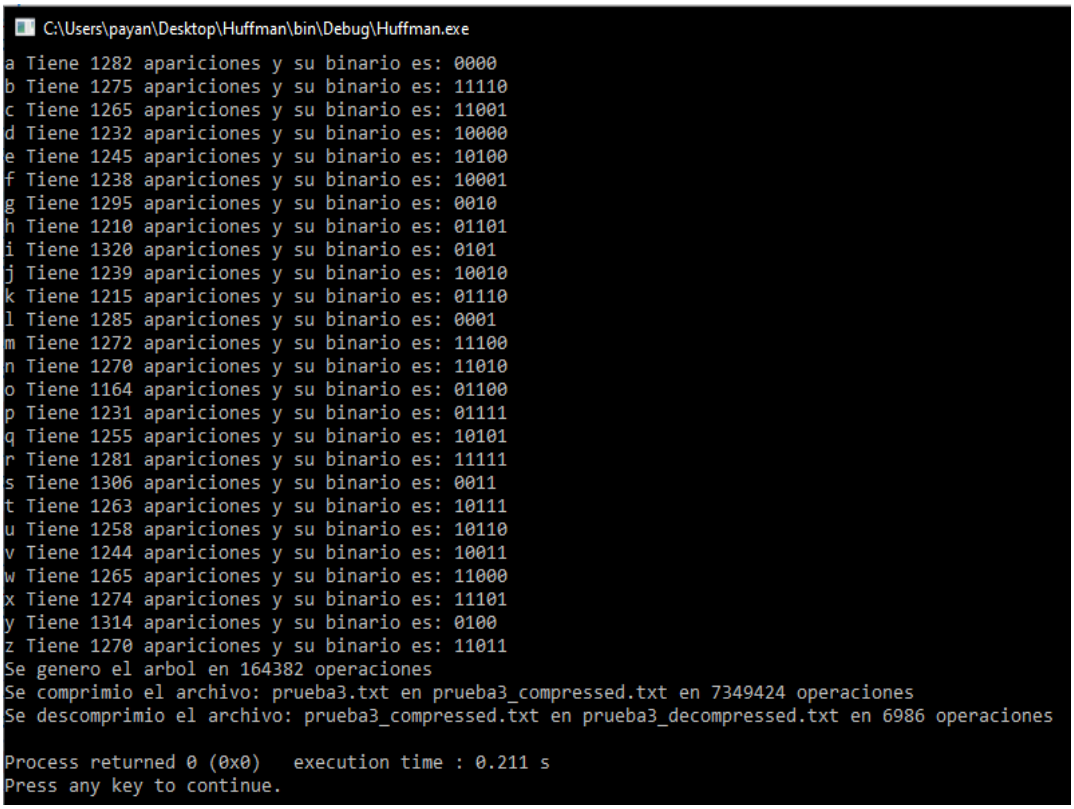

Figure 22: Ejecucion del programa

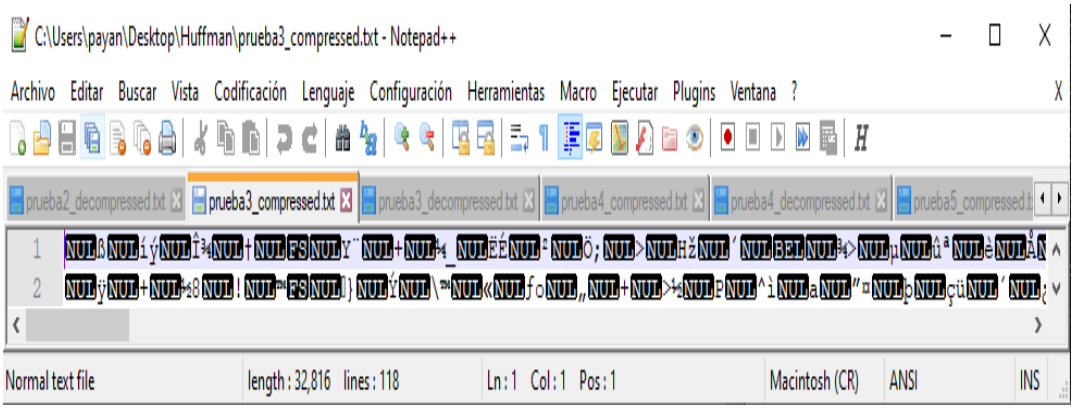

Figure 23: Archivo de salida

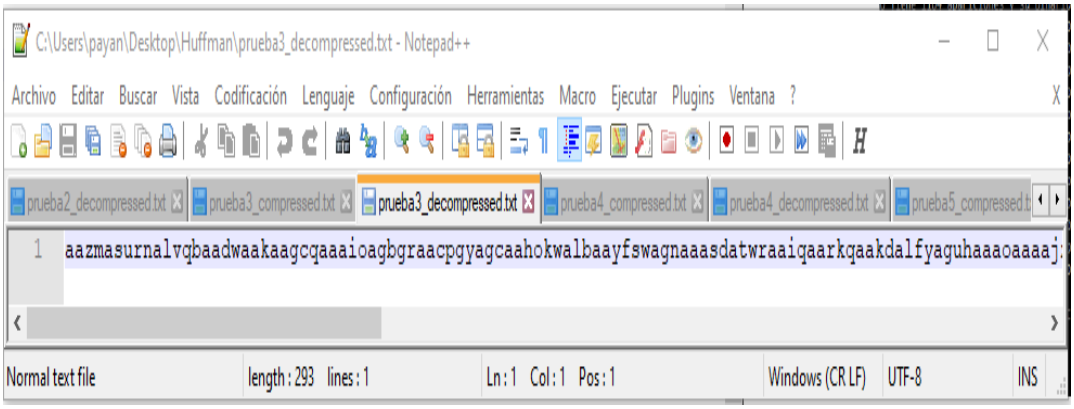

Figure 24: Archivo descomprimido

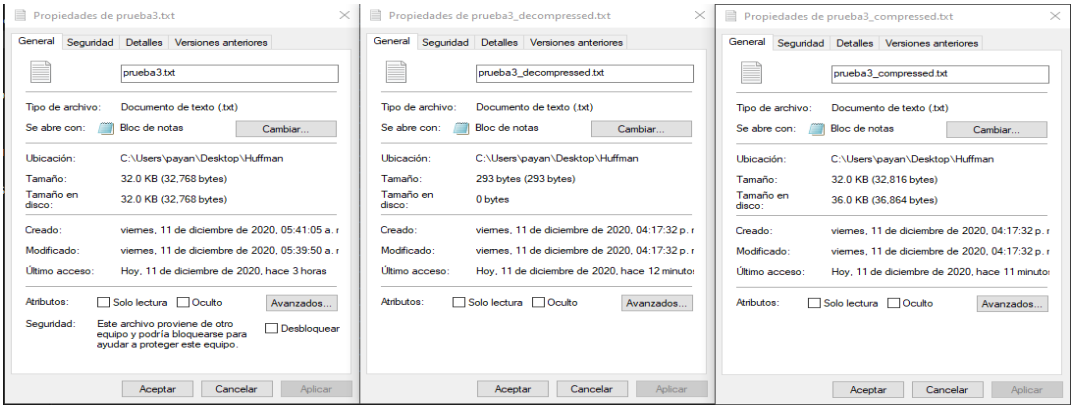

Figure 25: peso de los 3 archivos

Cuarto archivo

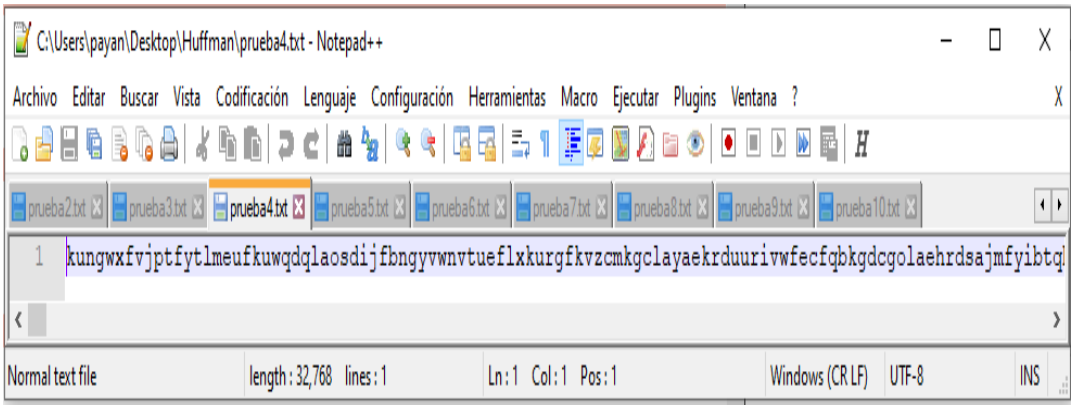

Figure 26: Archivo de entrada

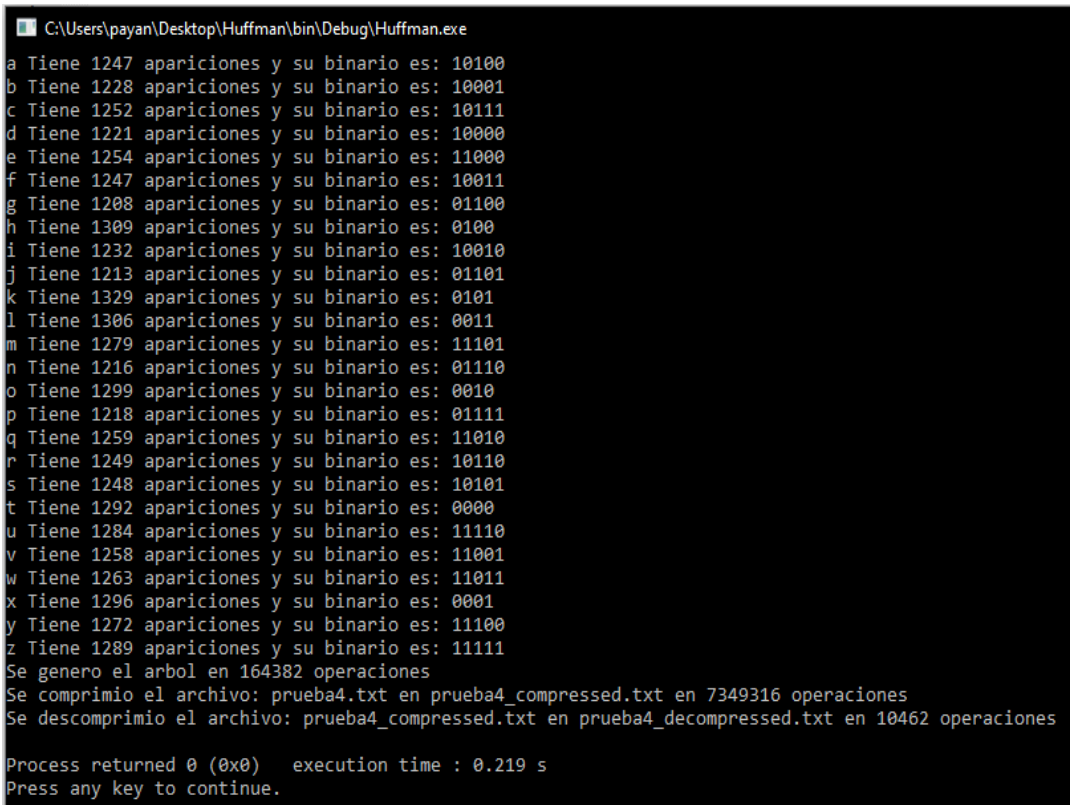

Figure 27: Ejecucion del programa

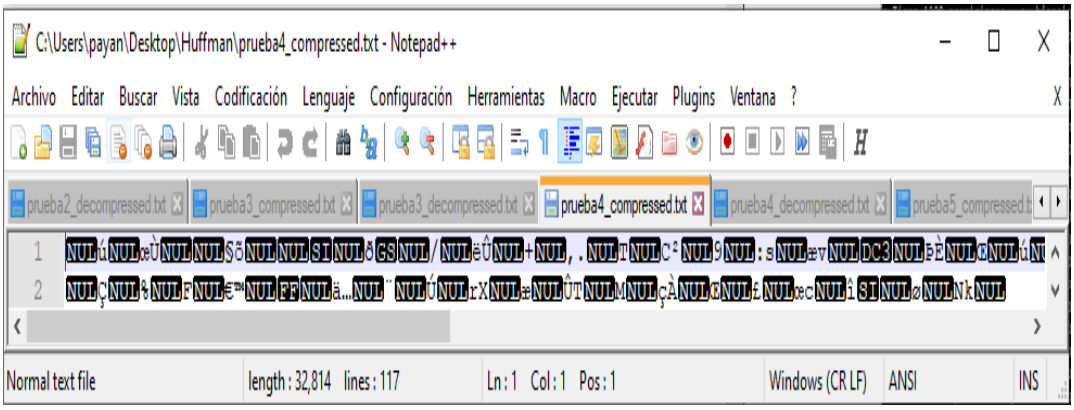

Figure 28: Archivo de salida

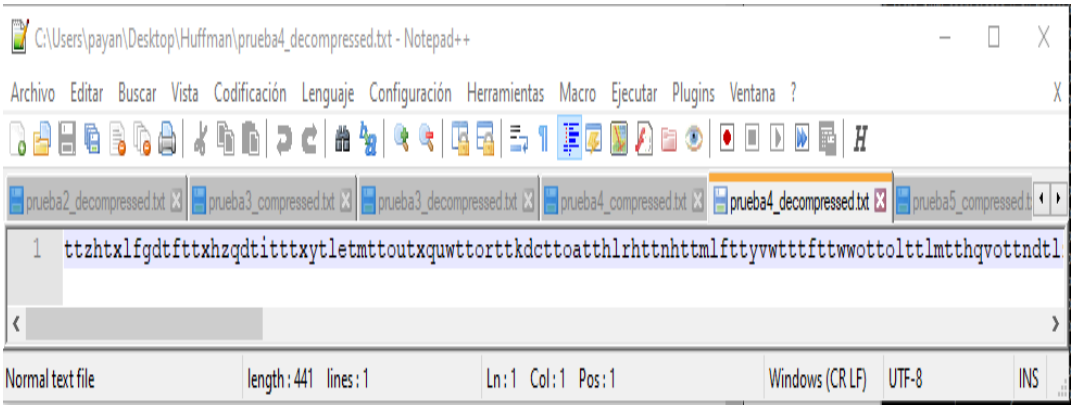

Figure 29: Archivo descomprimido

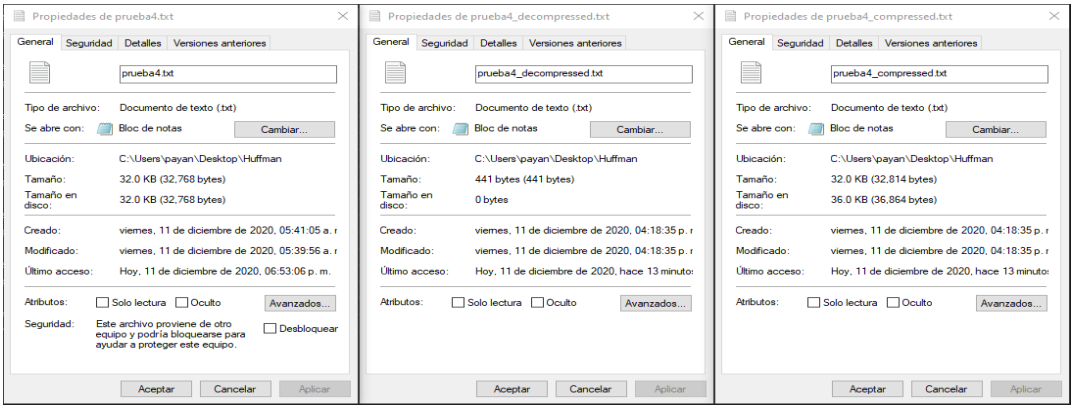

Figure 30: peso de los 3 archivos

Quinto archivo

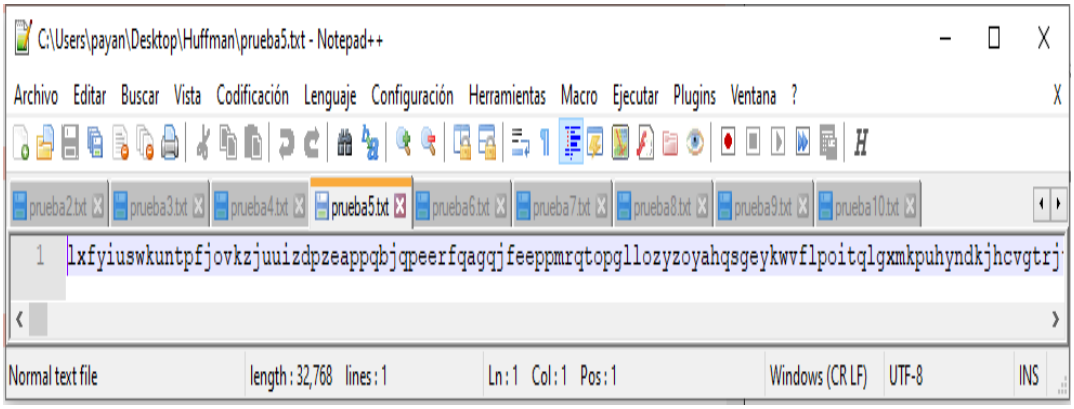

Figure 31: Archivo de entrada

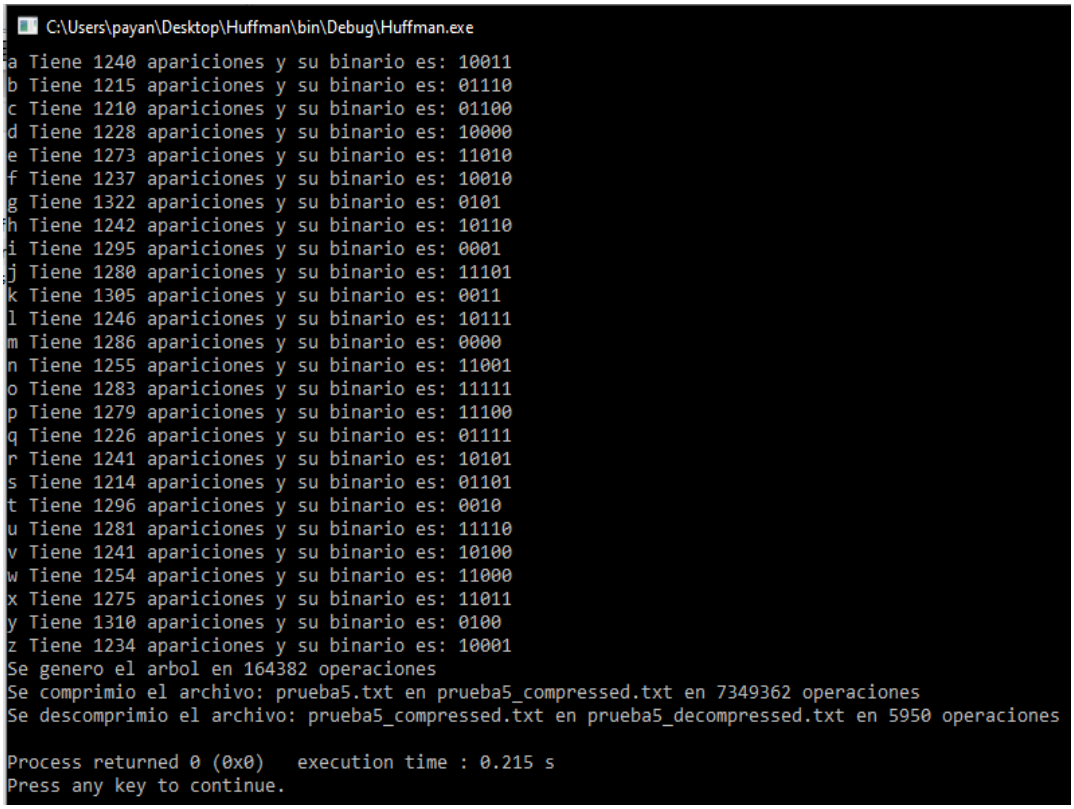

Figure 32: Ejecucion del programa

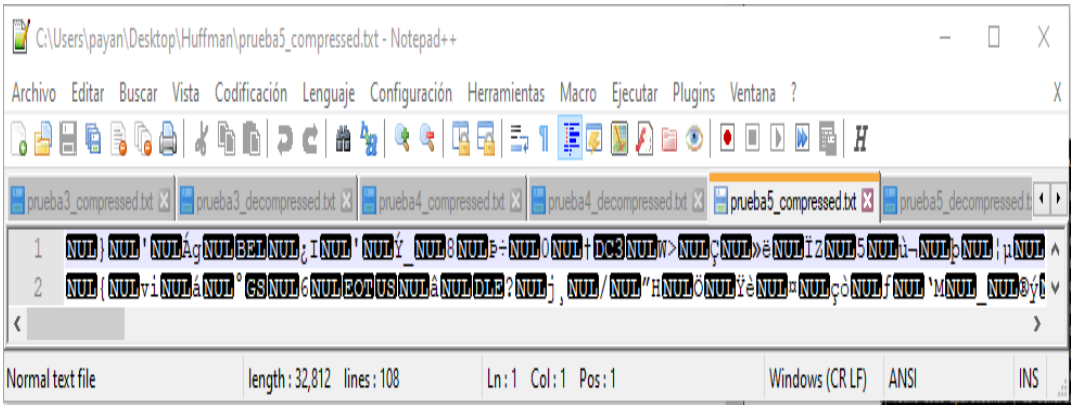

Figure 33: Archivo de salida

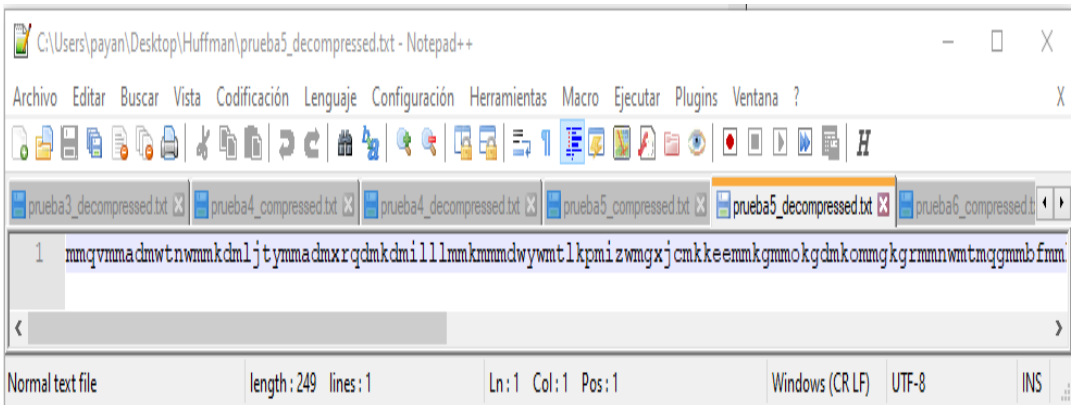

Figure 34: Archivo descomprimido

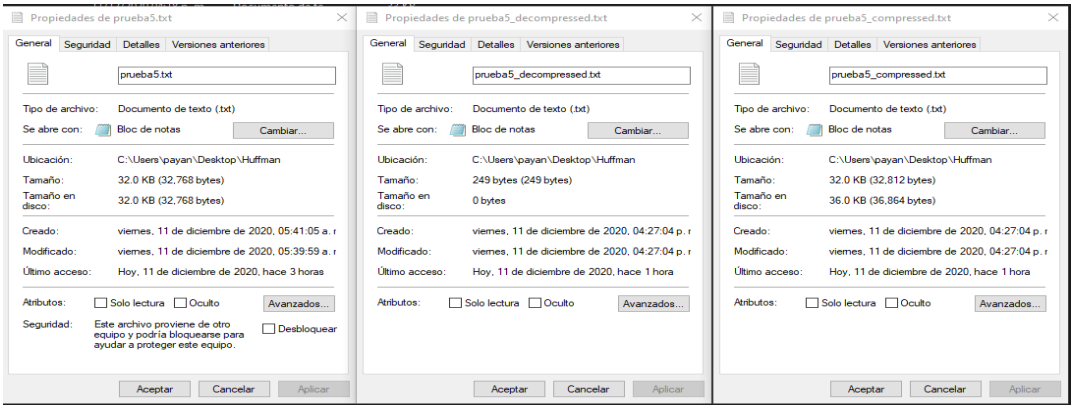

Figure 35: peso de los 3 archivos

Sexto archivo

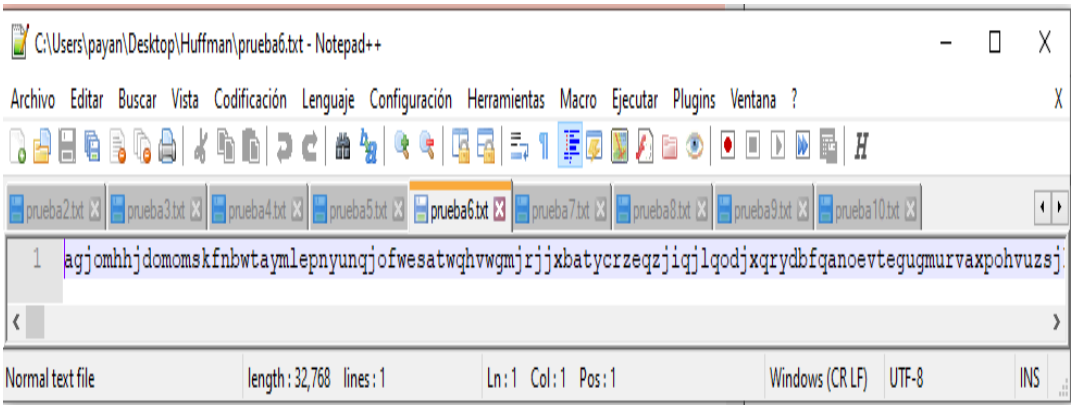

Figure 36: Archivo de entrada

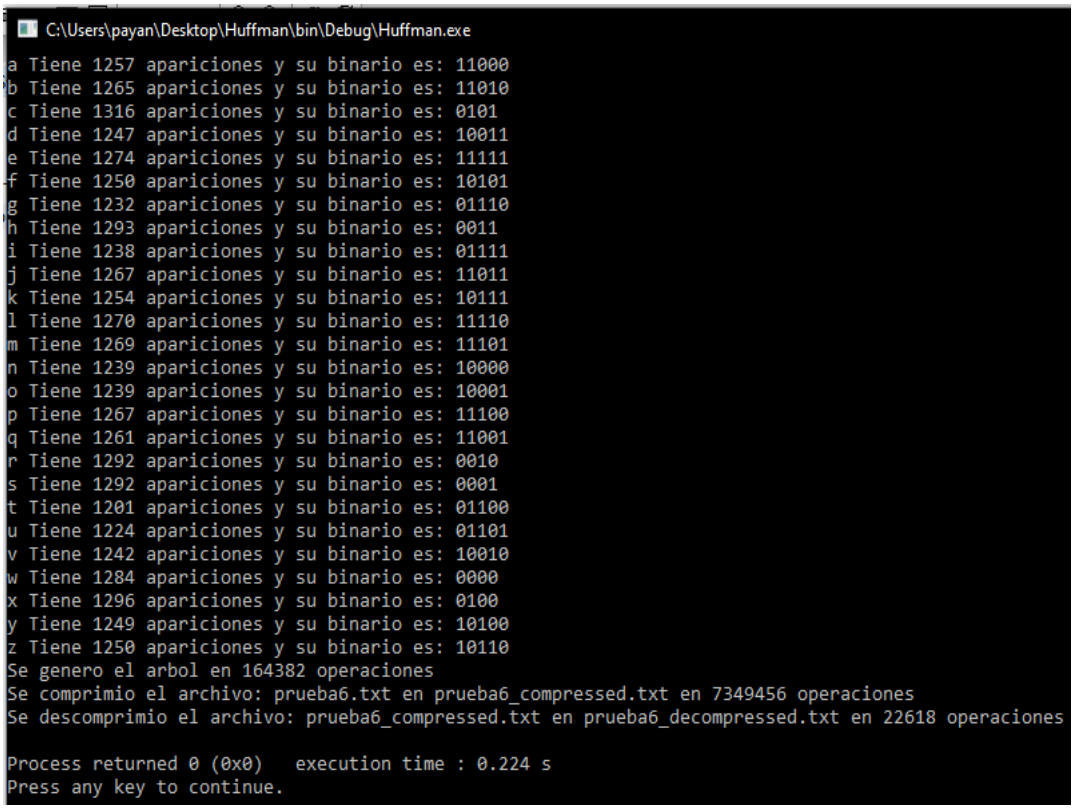

Figure 37: Ejecucion del programa

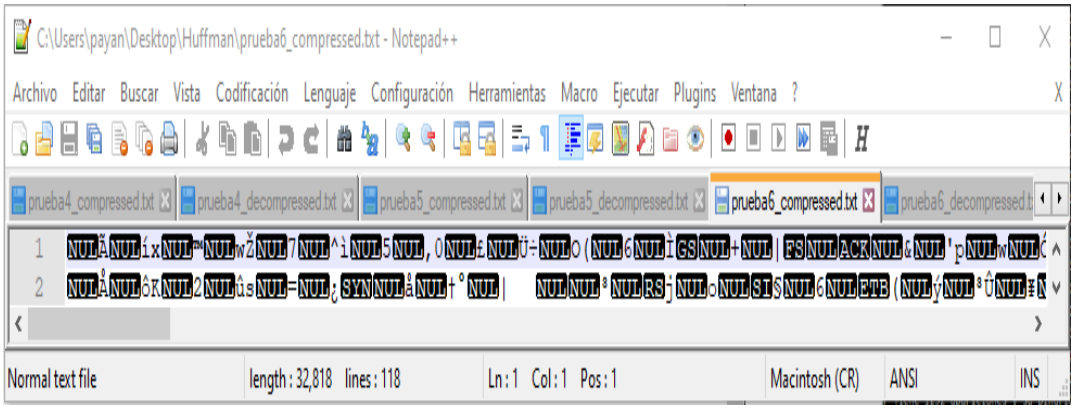

Figure 38: Archivo de salida

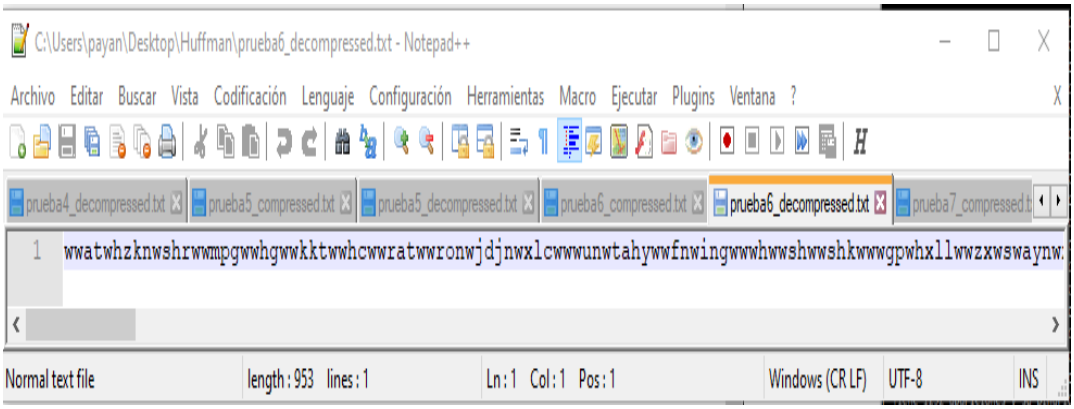

Figure 39: Archivo descomprimido

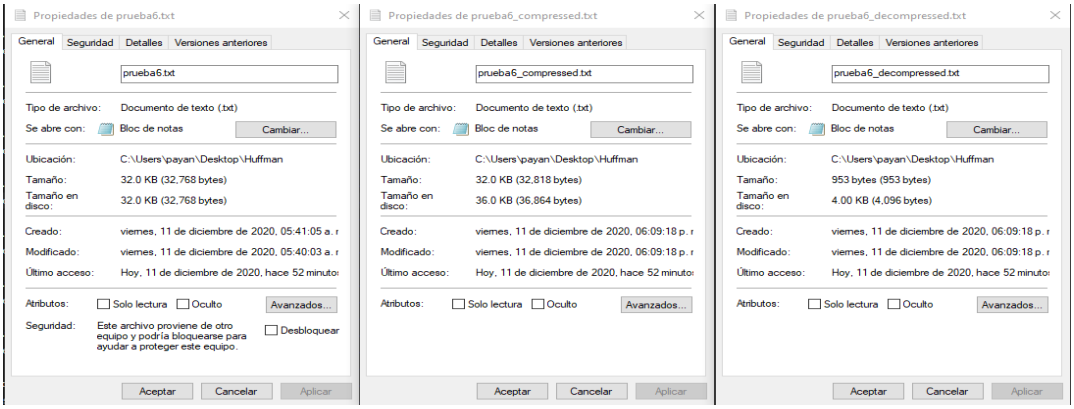

Figure 40: peso de los 3 archivos

Septimo archivo

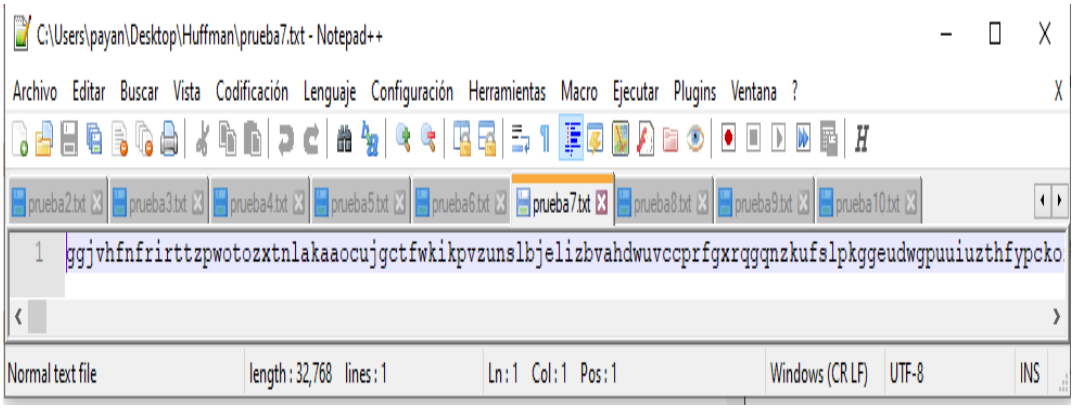

Figure 41: Archivo de entrada

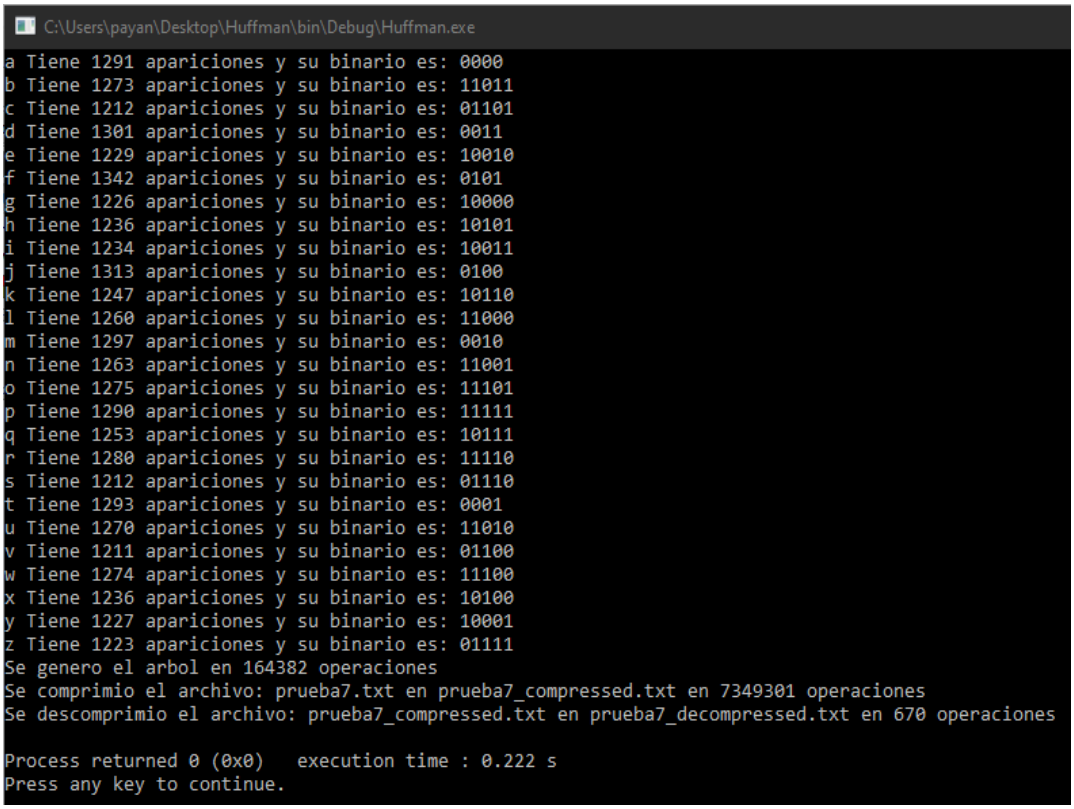

Figure 42: Ejecucion del programa

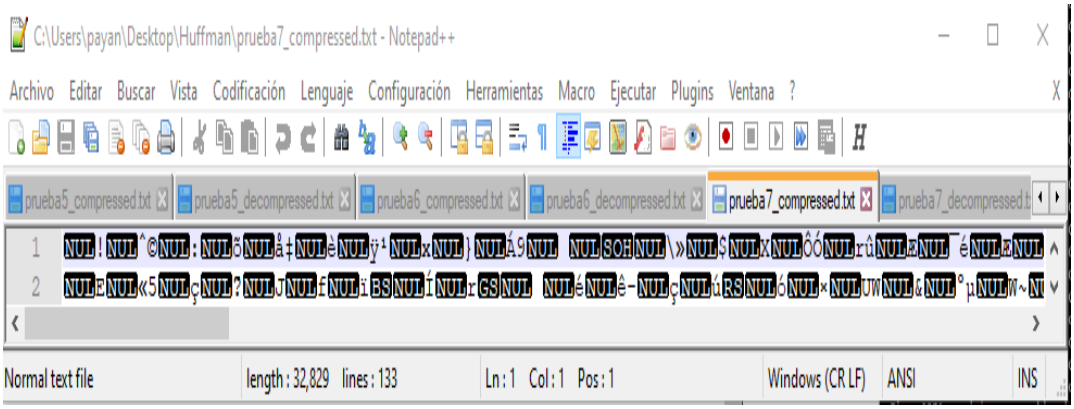

Figure 43: Archivo de salida

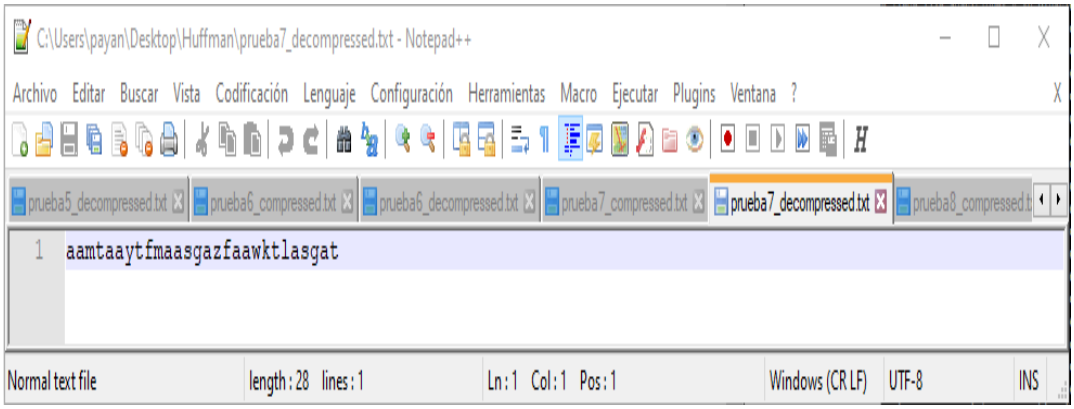

Figure 44: Archivo descomprimido

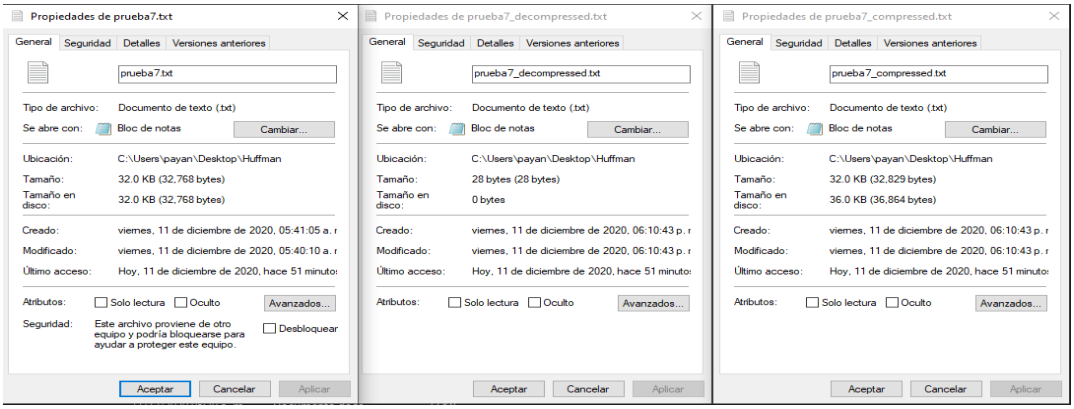

Figure 45: peso de los 3 archivos

Octavo archivo

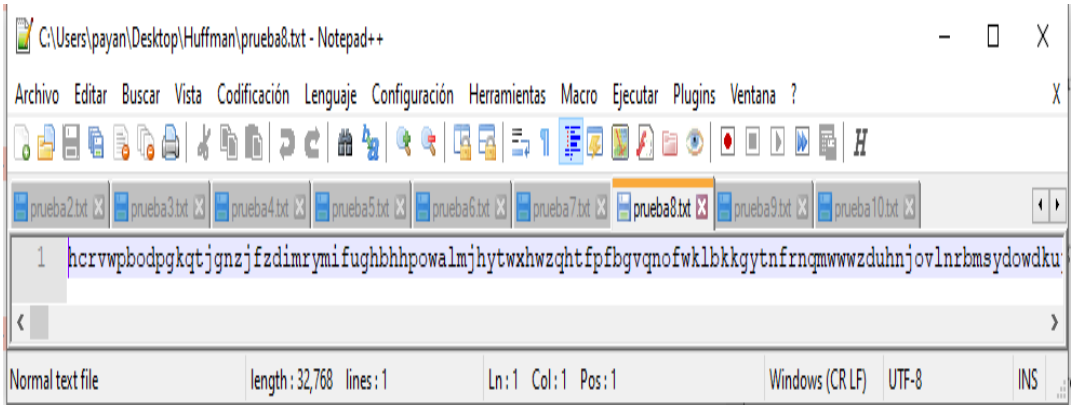

Figure 46: Archivo de entrada

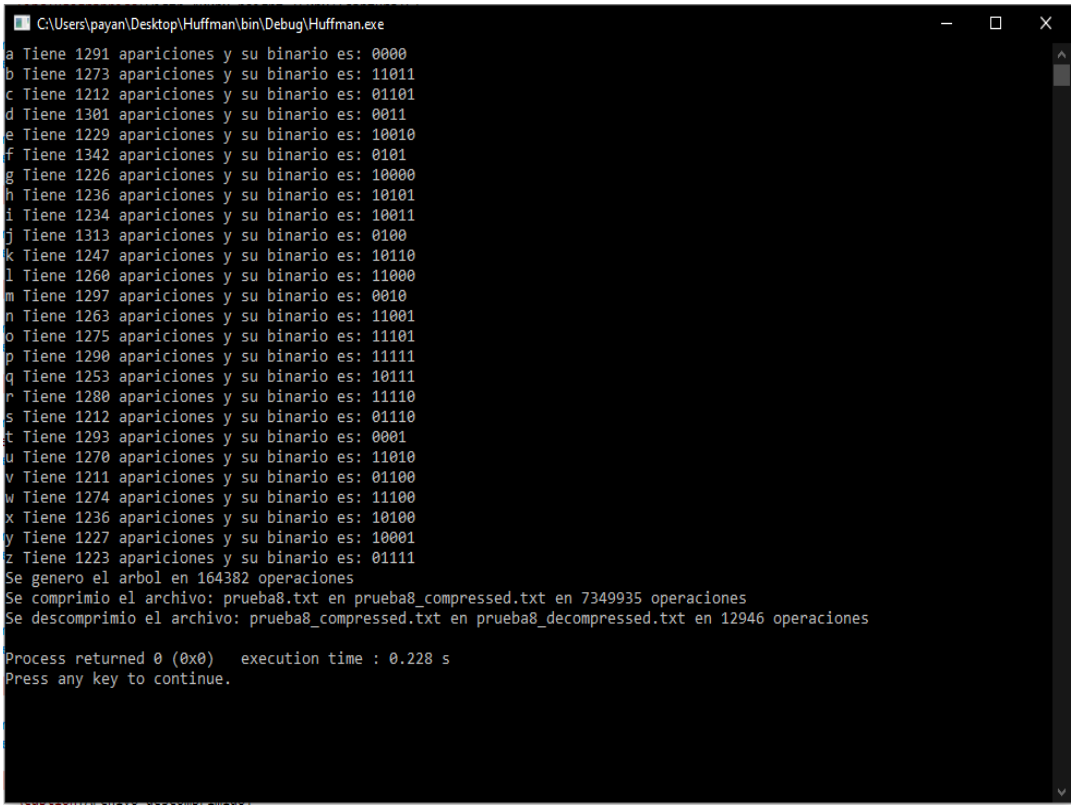

Figure 47: Ejecucion del programa

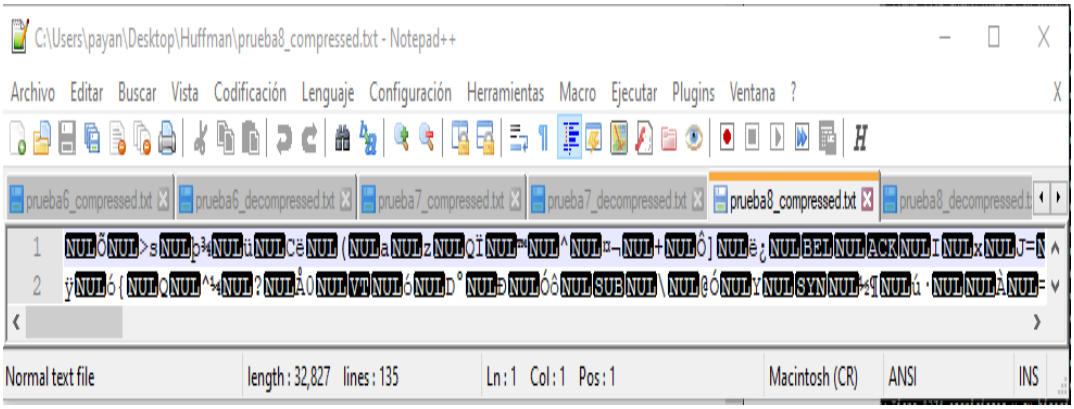

Figure 48: Archivo de salida

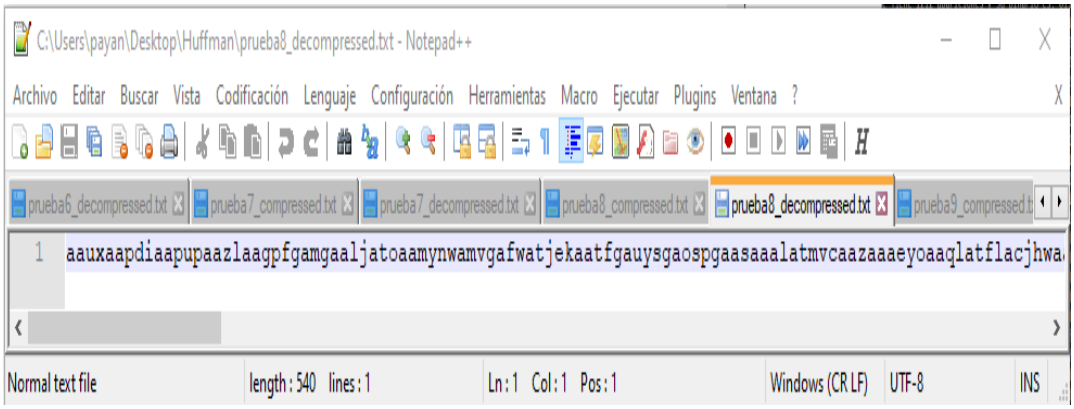

Figure 49: Archivo descomprimido

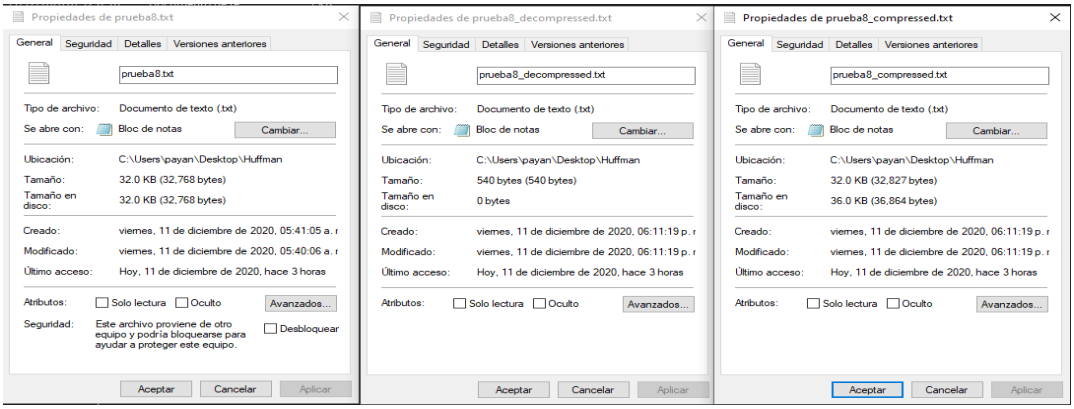

Figure 50: peso de los 3 archivos

Noveno archivo

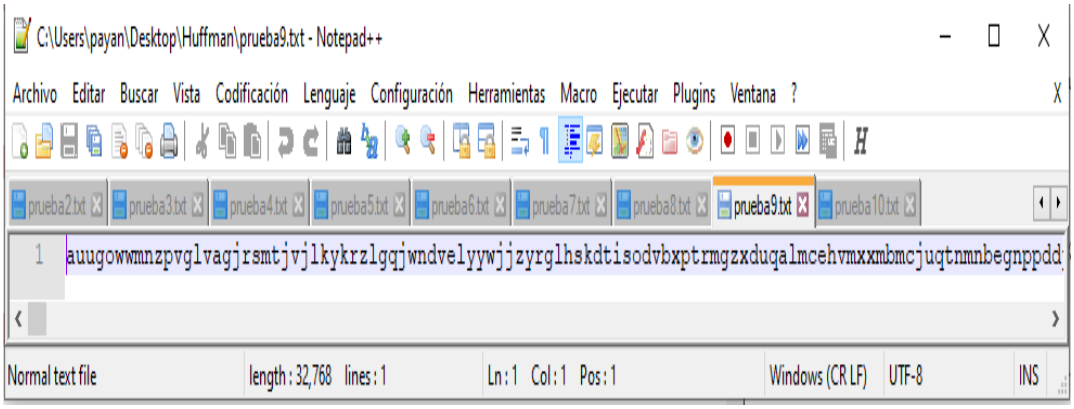

Figure 51: Archivo de entrada

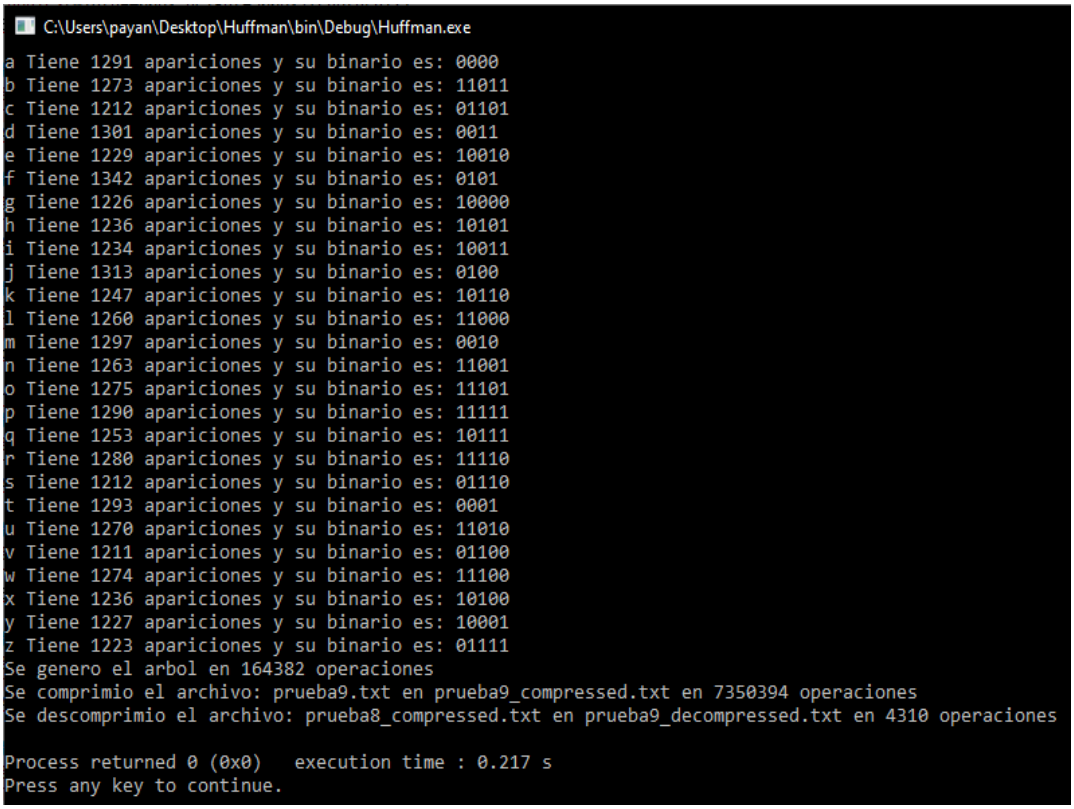

Figure 52: Ejecucion del programa

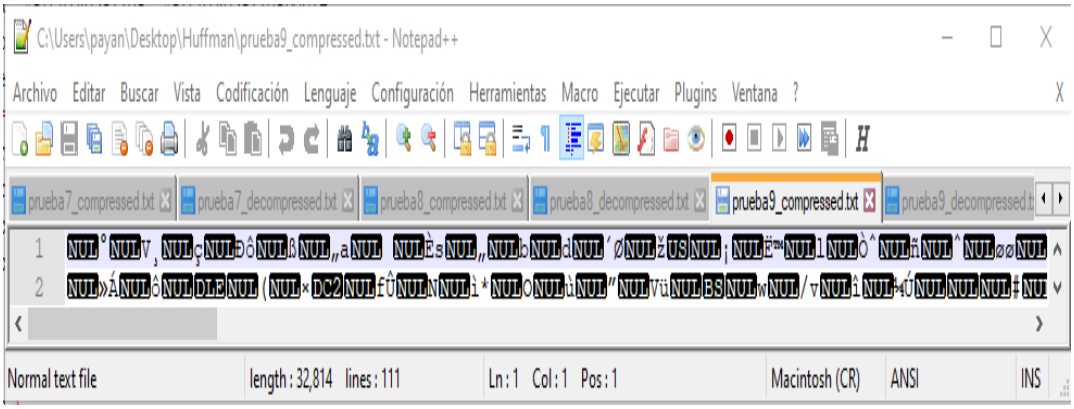

Figure 53: Archivo de salida

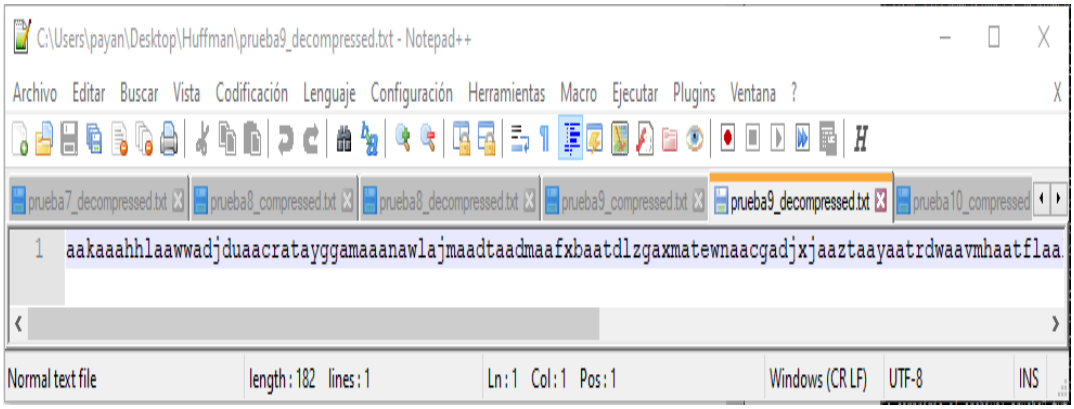

Figure 54: Archivo descomprimido

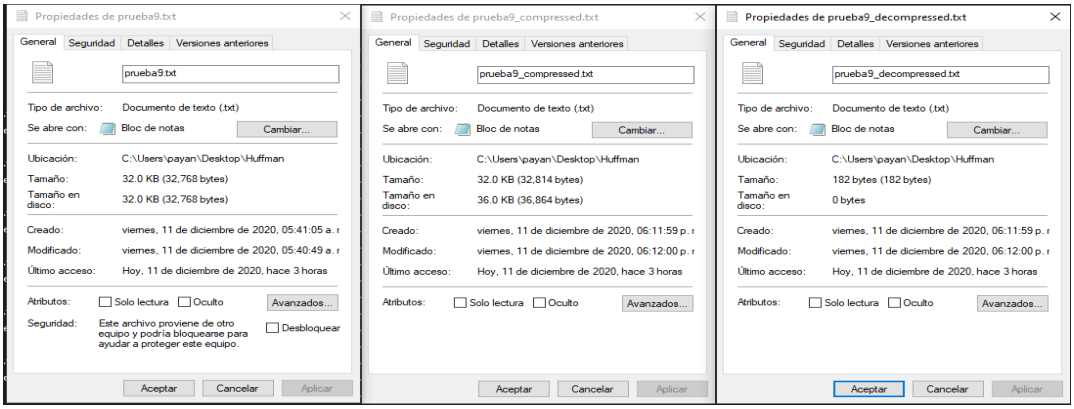

Figure 55: peso de los 3 archivos

Decimo archivo

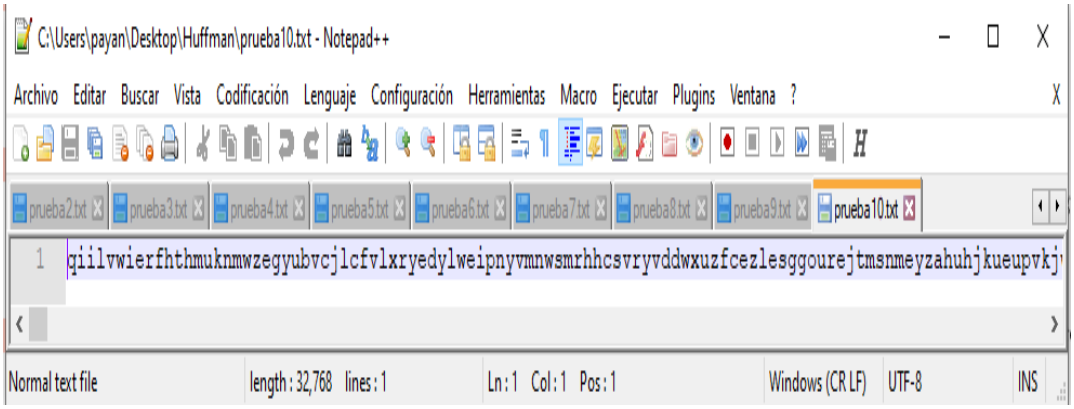

Figure 56: Archivo de entrada

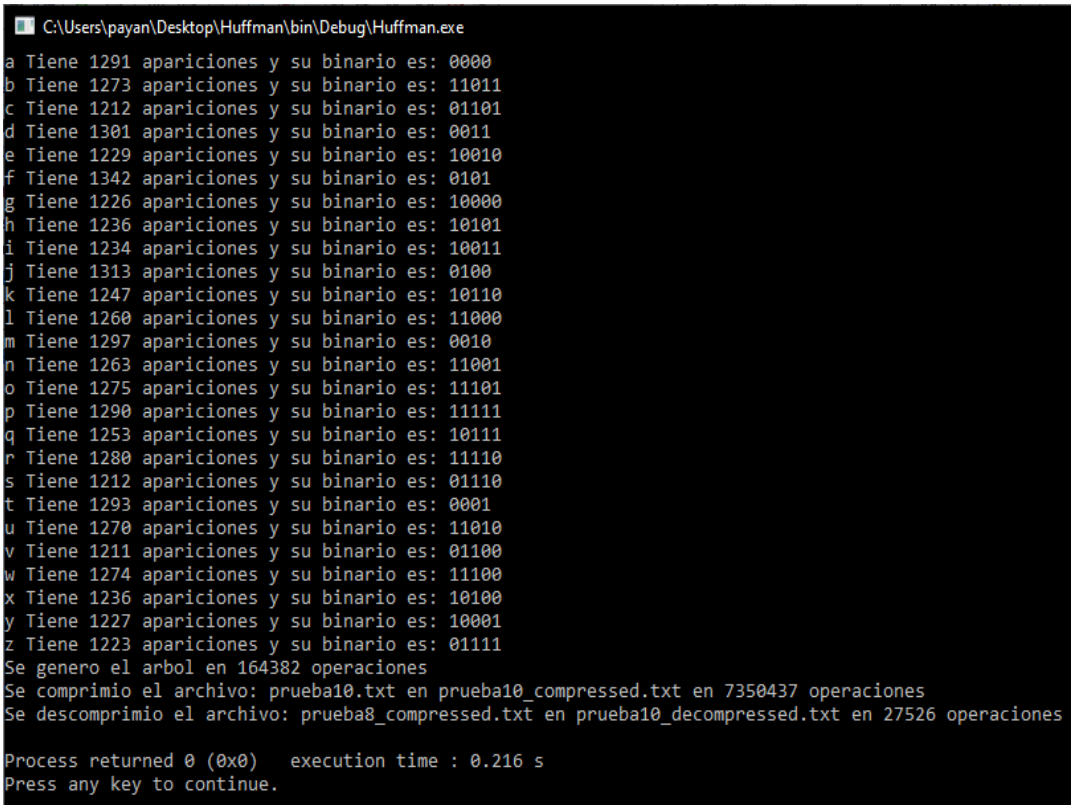

Figure 57: Ejecucion del programa

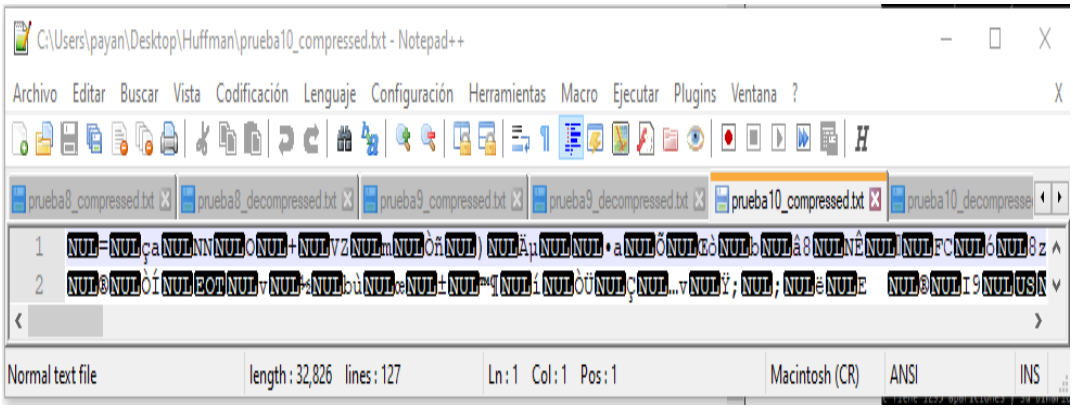

Figure 58: Archivo de salida

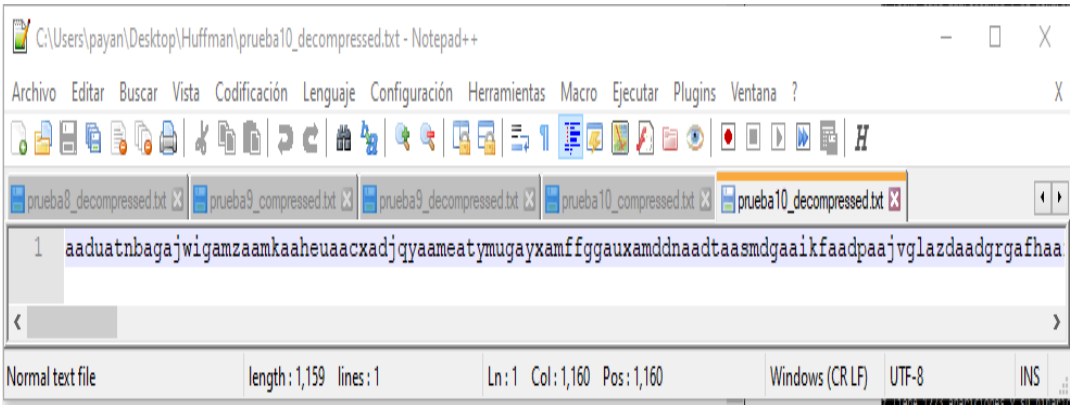

Figure 59: Archivo descomprimido

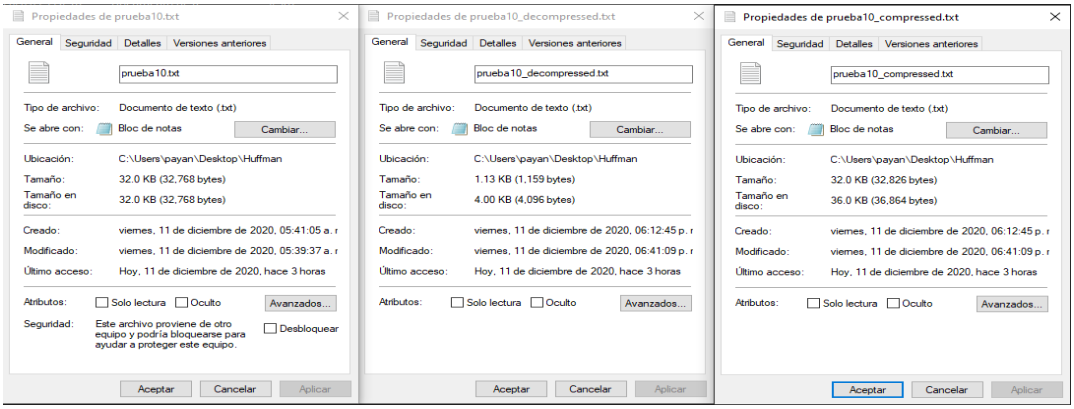

Figure 60: peso de los 3 archivos

Finalmente para concluir este algorimo, se realizo una modificacion al codigo para generar una cadena aleatoria de longitud "n", con ($1 \leq n \leq 10000$), posteriormente a esta se le genero el arbol, se comprimio y descomprimio, para poder obtener la grafica de complejidad de los algorimos de compresion y descompresion de Huffman.

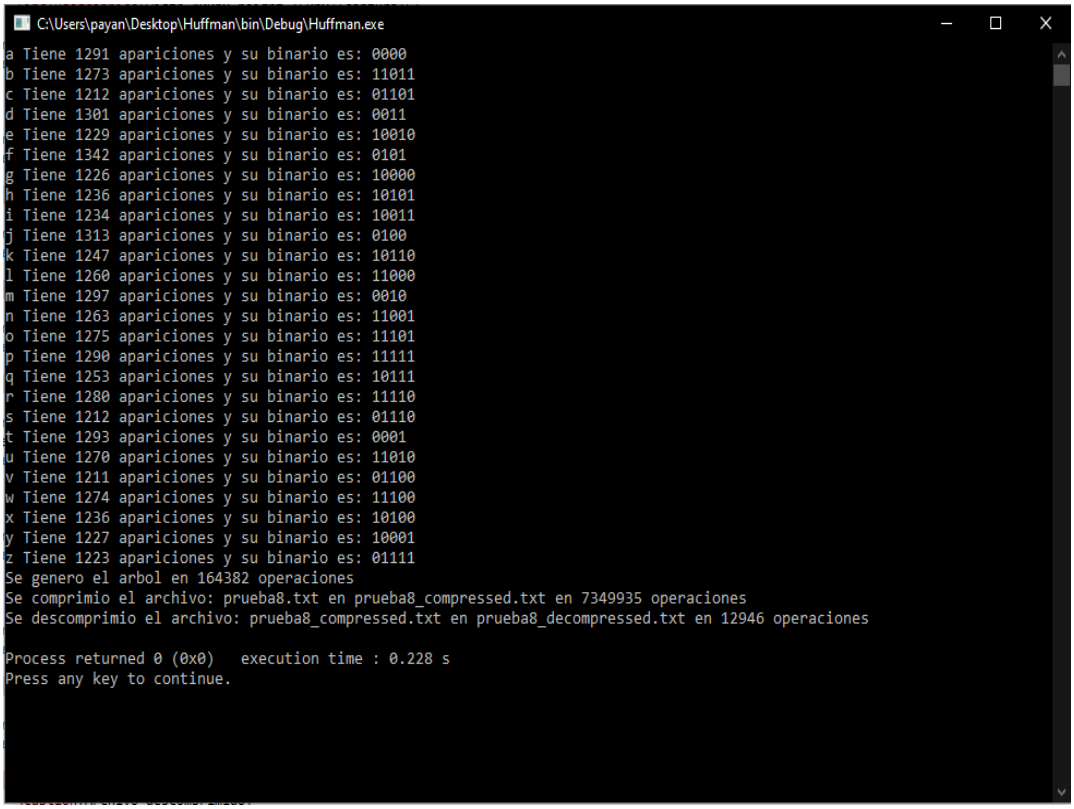

Figure 61: Ejecucion del programa generando cadenas aleatorias, comprimiendolas y descomprimiendolas

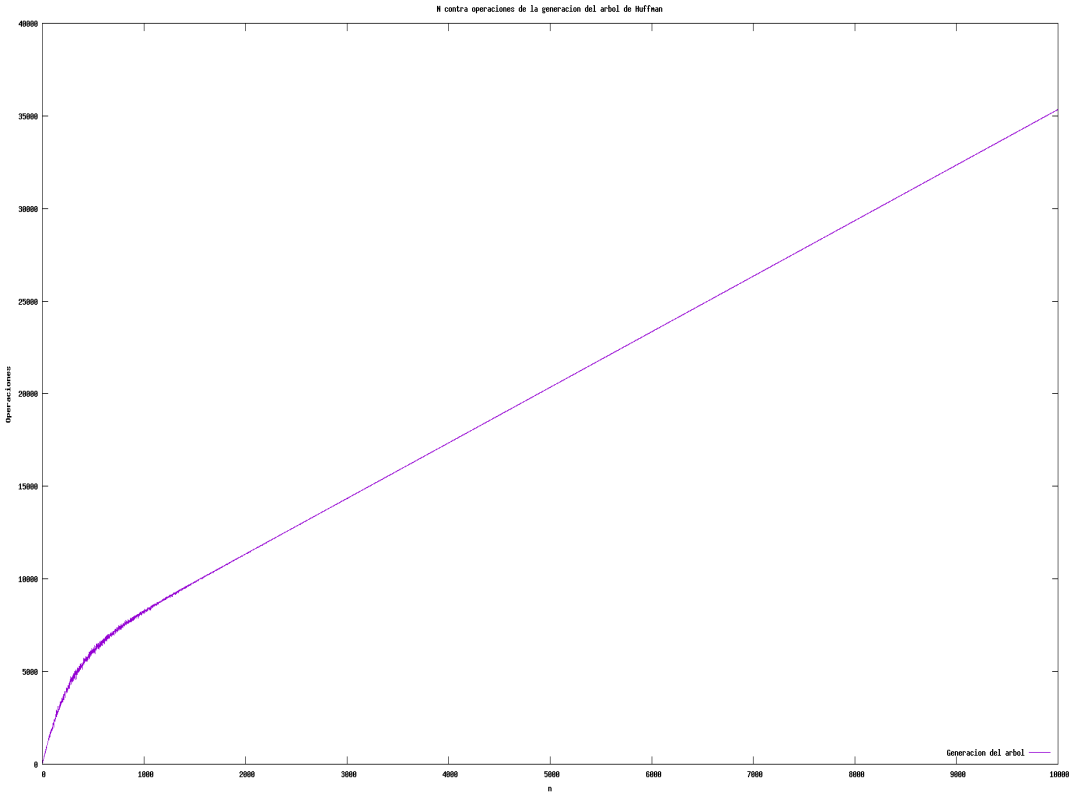

Figure 62: N contra operaciones de la generacion del arbol de Huffman

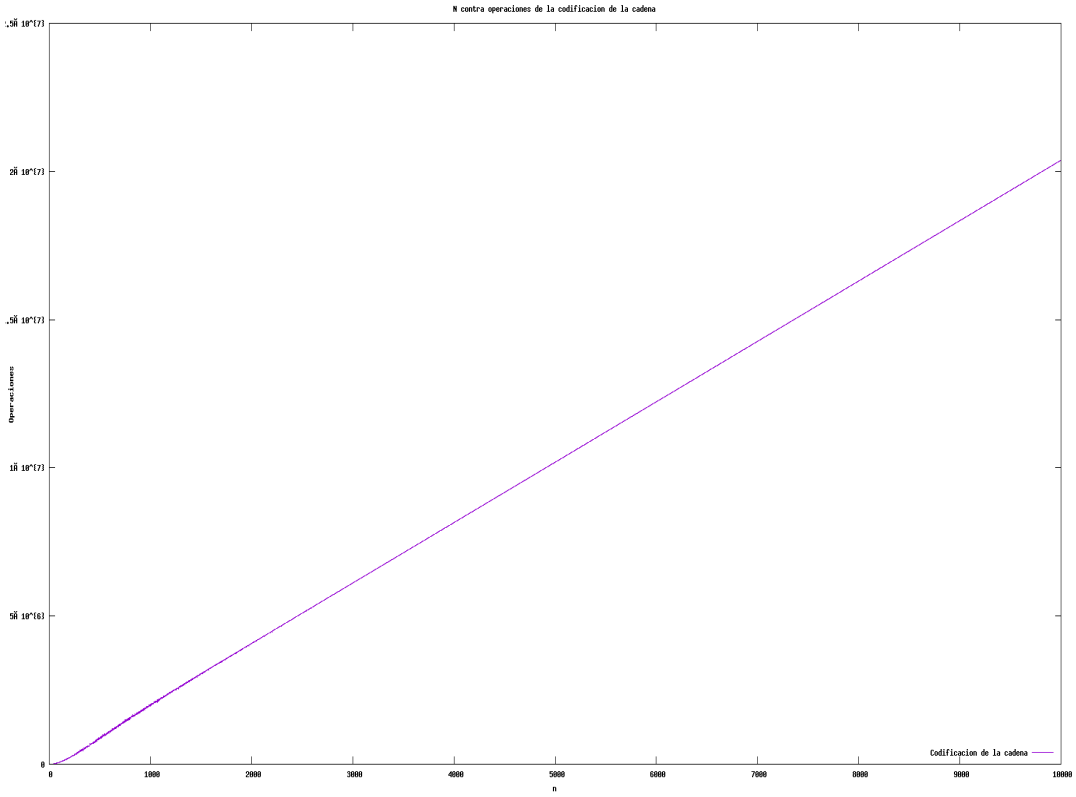

Figure 63: N contra operaciones de la codificacion de la cadena

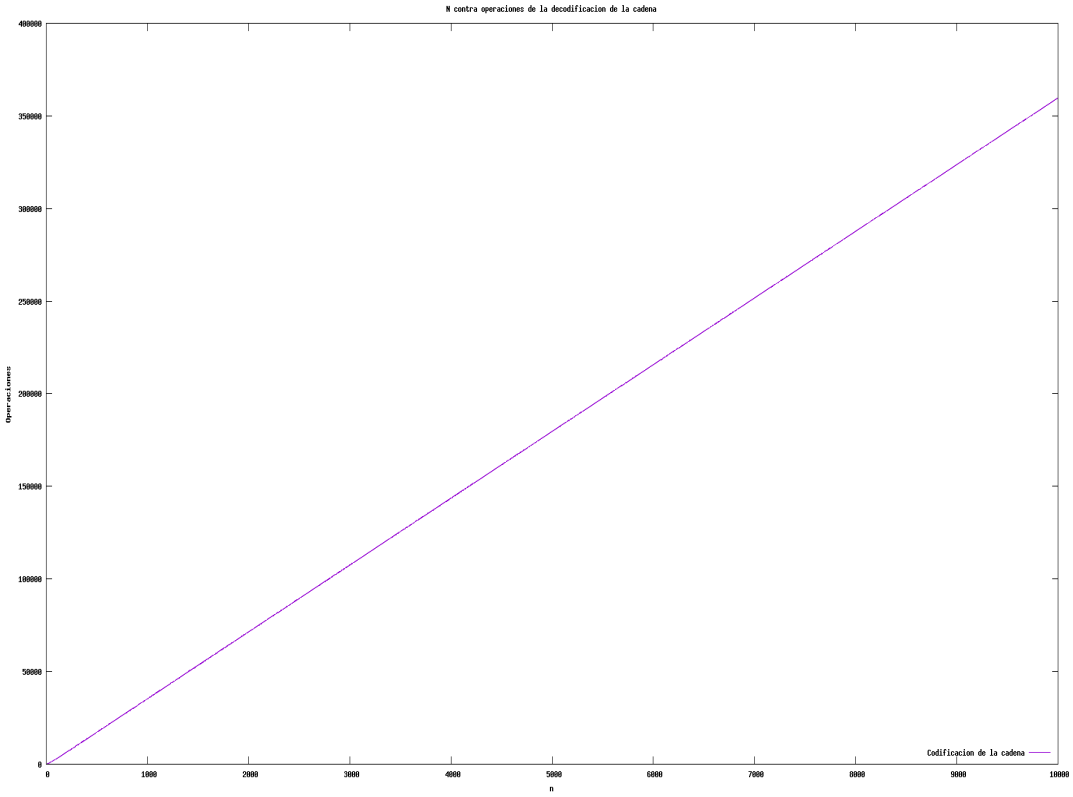

Figure 64: N contra operaciones de la decodificacion de la cadena

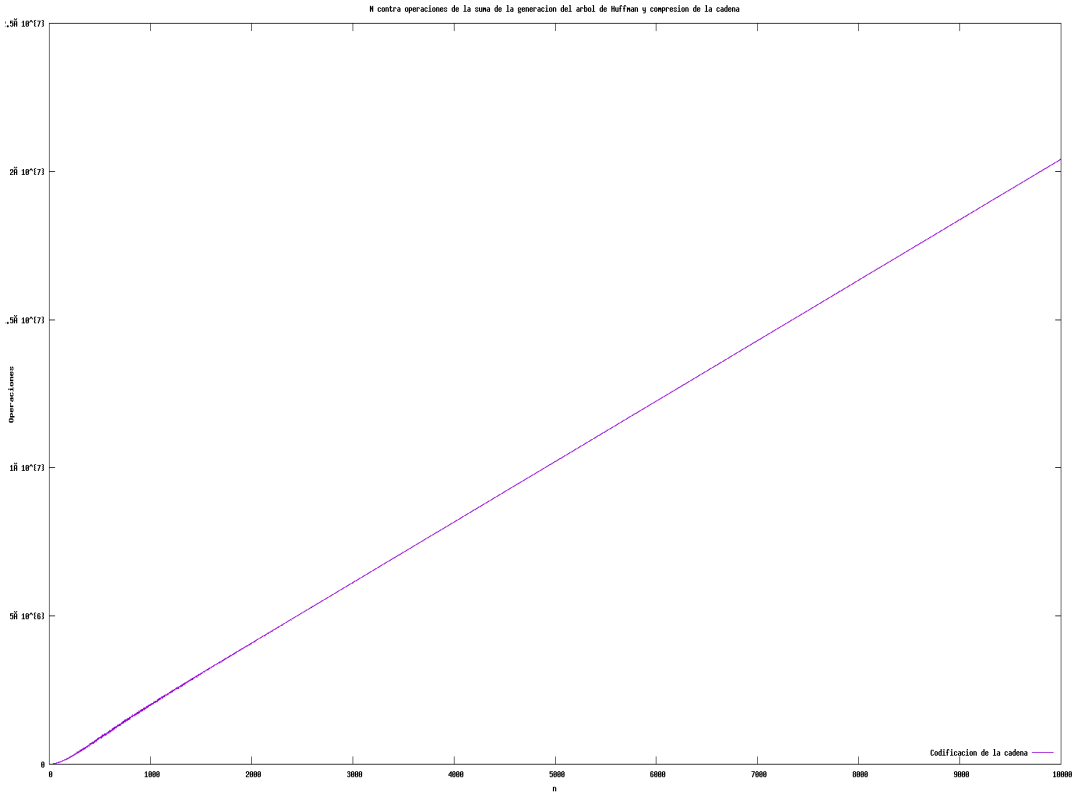

Figure 65: N contra operaciones de la suma de la generacion del arbol de Huffman y compresion de la cadena

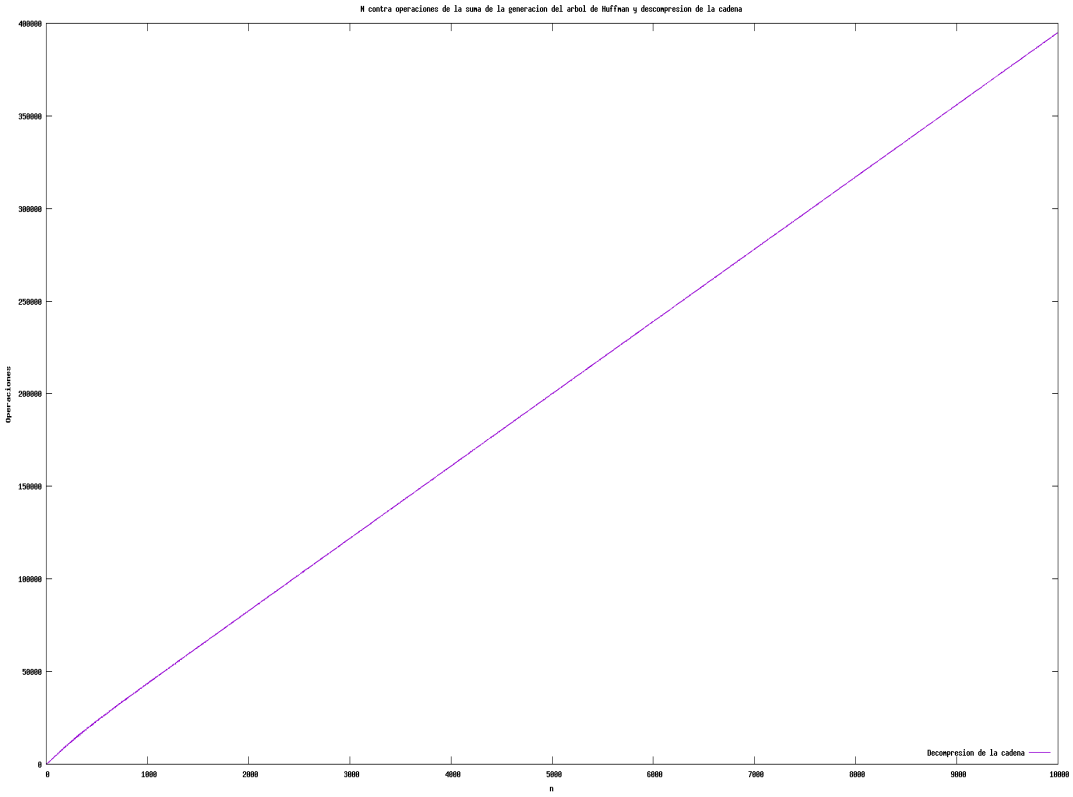

Figure 66: N contra operaciones de la suma de la generacion del arbol de Huffman y descompresion de la cadena

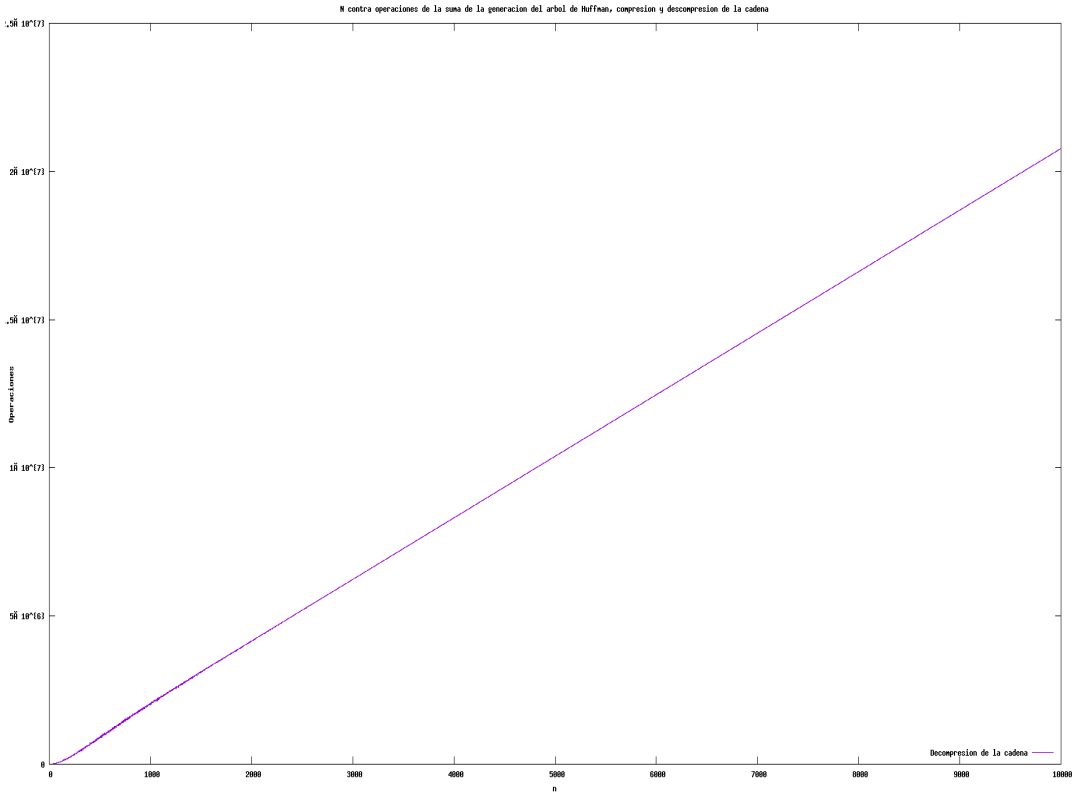

Figure 67: N contra operaciones de la suma de la generacion del arbol de Huffman, compresion y descompresion de la cadena

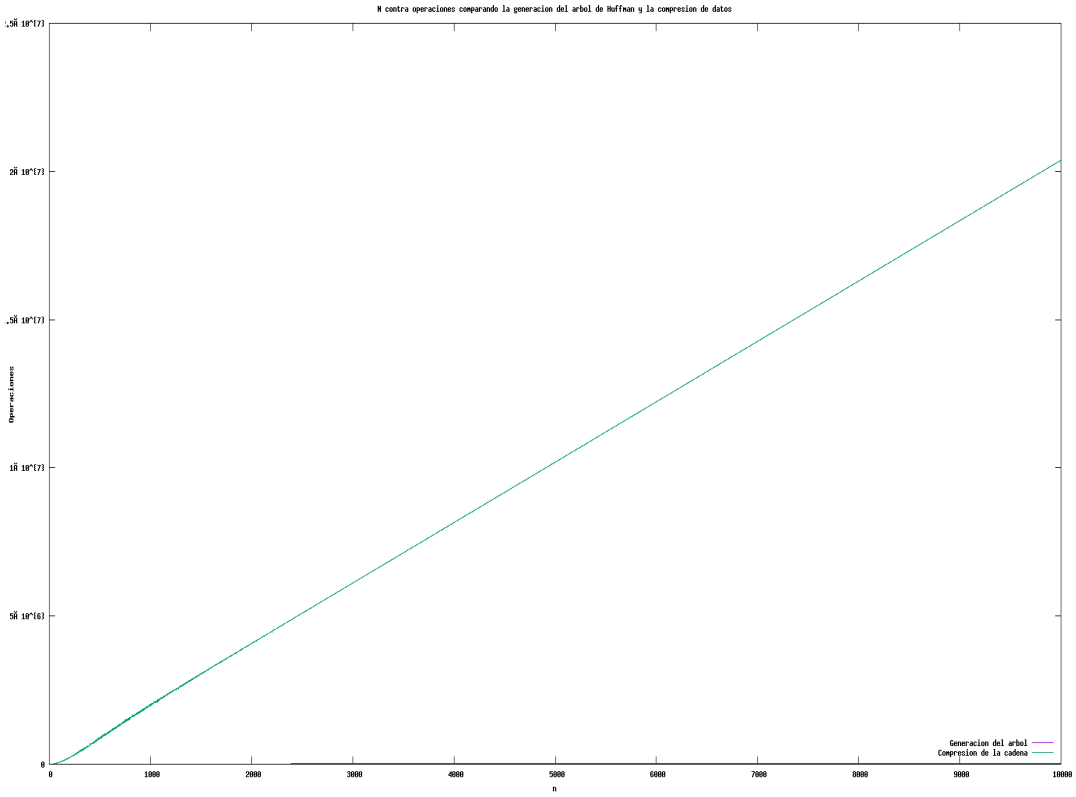

Figure 68: N contra operaciones comparando la generacion del arbol de Huffman y la compresion de datos

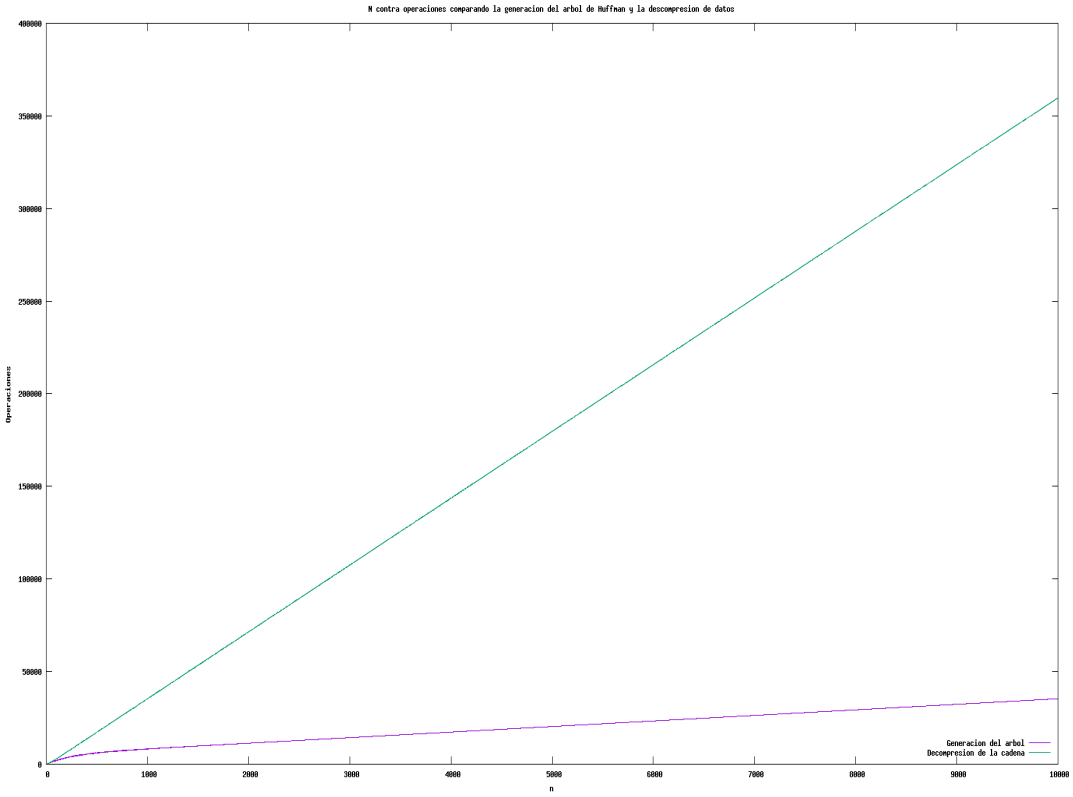

Figure 69: N contra operaciones comparando la generacion del arbol de Huffman y la descompresion de datos

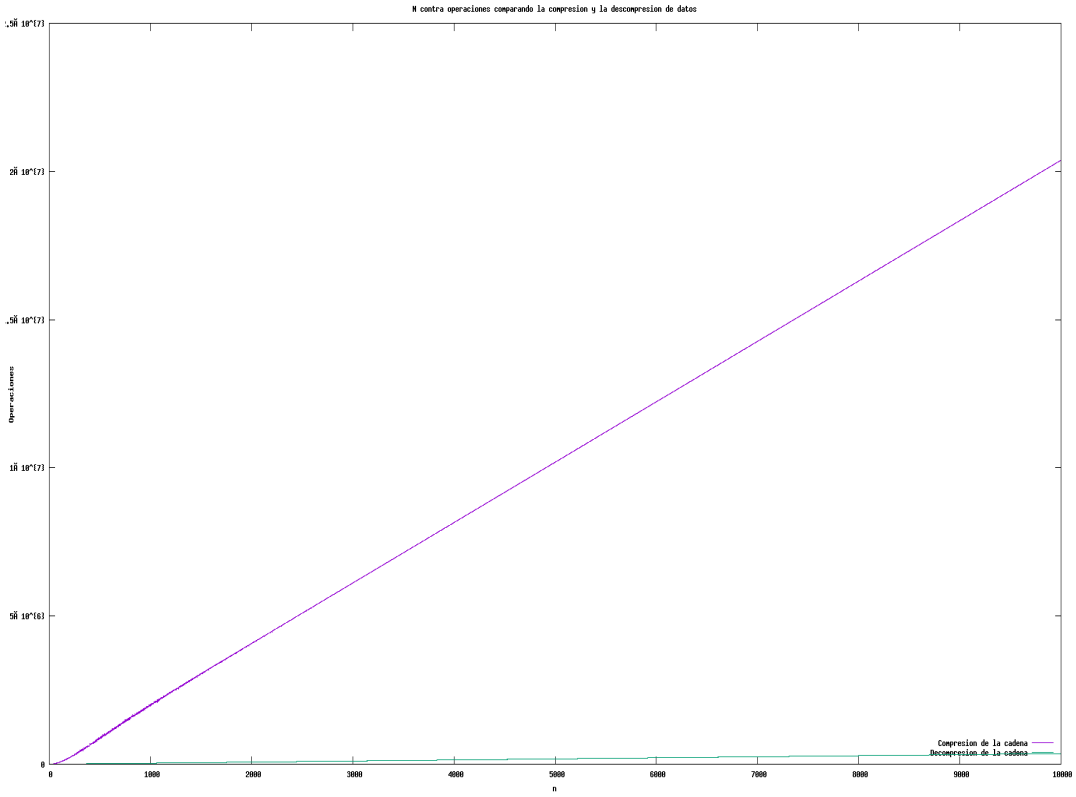

Figure 70: N contra operaciones comparando la compresion y la descompresion de datos

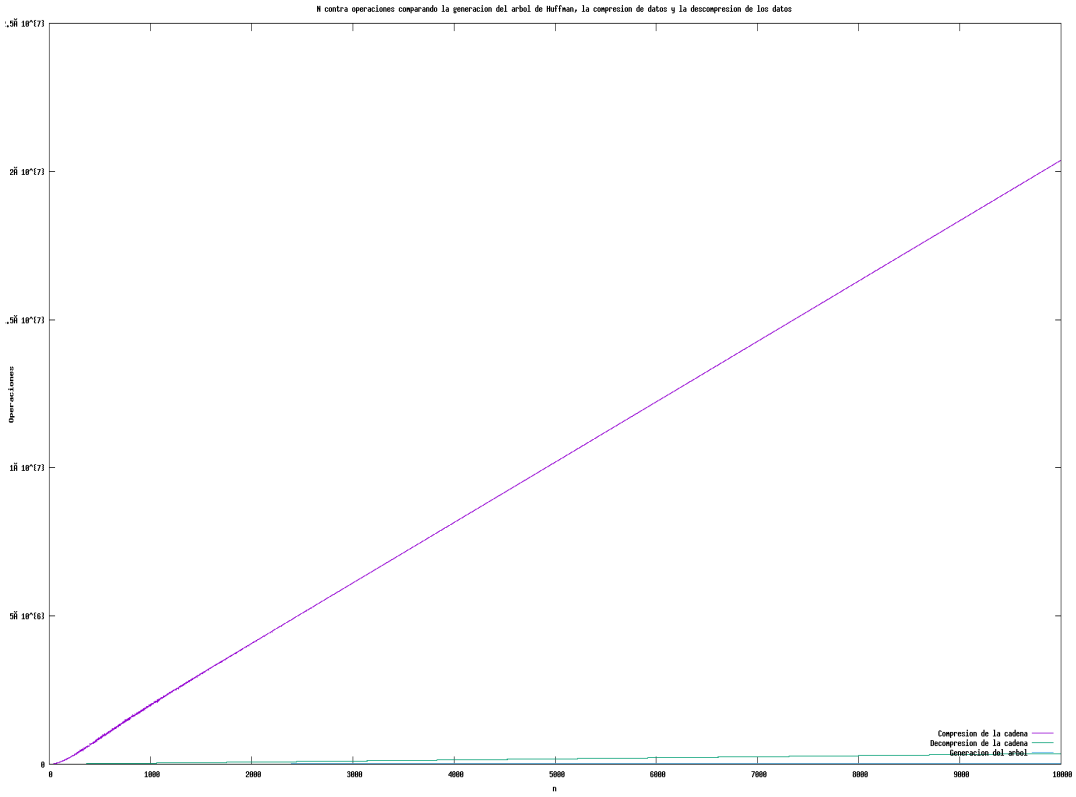

Figure 71: N contra operaciones comparando la generacion del arbol de Huffman, la compresion de datos y la descompresion de los datos

Para realizar la comparacion correspondiente, se realiza el calculo de complejidad a priori usando el analisis linea a linea del pseudo codigo. primero la funcion **HuffmanTree(C)**

Codigo	Costo	Veces ejecutado
$n = C.size()$	$\emptyset(1)$	1
Q=$priority\_queue()$	$\emptyset(1)$	1
$for(i=0;i\leq n;i++)$	$\emptyset(n)$	n+1
n=$node(C[i])$	$\emptyset(1)$	n
$Q.push(n)$	$\emptyset(log_2(n-i))$	n
$while(Q.size()>1)$	$\emptyset(log_2(n))$	$log_2(n)+1$
$z = new\ node()$	$\emptyset(1)$	$log_2(n)$
$z.left = x = Q.pop()$	$\emptyset(log_2(n))$	$log_2(n)$
$z.right = y = Q.pop()$	$\emptyset(log_2(n))$	$log_2(n)$
$z.frequency = x.frequency$+$y.frequency$	$\emptyset(1)$	$log_2(n)$
$Q.push(z)$	$\emptyset(log_2(n))$	$log_2(n)$
$return\ Q.pop()$	$\emptyset(1)$	1

Como podemos observar la mayor complejidad se encuentra dentro de los ciclos, donde en el peor escenario llega a ser $\emptyset(nlog_2(n))$, considerando lo anterior se grafica la funcion $f(x) = xlog_2(x)$ y se anexa la grafica para ser comparada con los resultados del programa.

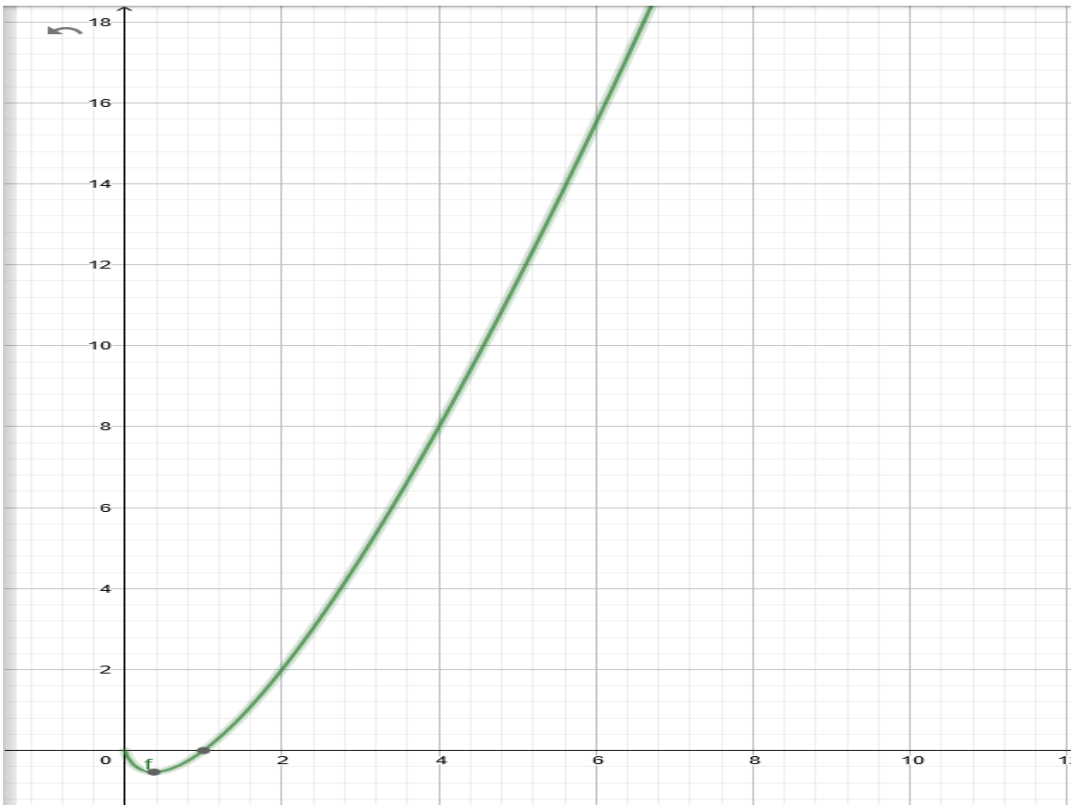

Figure 72: Grafica de la funcion $f(x) = xlog_2(x)$

Dada la grafica anterior, el analisis a posteriori y las graficas: Figure 62 y Figure 67, podemos concluir que la funcion para generar el arbol de Huffman tiene complejidad de $nlog_2(n)$
Ahora se realiza el calculo de complejidad de la funcion **HuffmanDecompression(tree, S)**

Codigo	Costo	Veces ejecutado
n = S.length()	$\emptyset(1)$	1
retorno=””	$\emptyset(1)$	1
for(i=0;i≤n;i++)	$\emptyset(n)$	n+1
actual=tree	$\emptyset(1)$	n
while(actual.left!=NULL and actual.right!=NULL)	$\emptyset(log_2(255))$	n
if(S[i] == '0')	$\emptyset(1)$	$log_2(255) * n$
actual = actual.left	$\emptyset(1)$	$\frac{log_2(255)*n}{2}$
else	$\emptyset(1)$	$log_2(255) * n$
actual = actual.right	$\emptyset(1)$	$\frac{log_2(255)*n}{2}$
i++	$\emptyset(1)$	$log_2(255) * n$
retorno+=actual.symbol	$\emptyset(1)$	n
return retorno	$\emptyset(1)$	1

Entonces despues de analizar la funcion, podemos concluir que la funcion tiene complejdad $\emptyset(n)$ siendo n la cantidad de bits en la cadena a descomprimir.

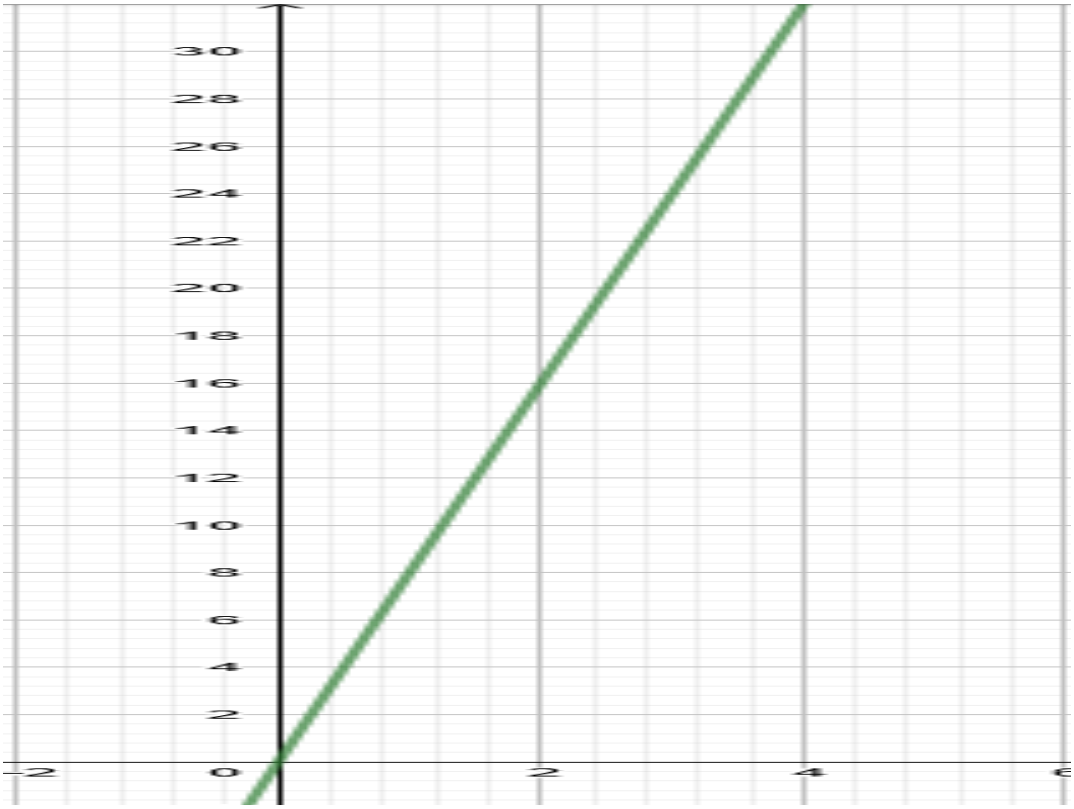

Figure 73: Grafica de la funcion $f(x) = xlog_2(255)$

Comparando la grafica de la funcion (Figure 73) y las graficas Figure 64, Figure 66, Figure 67, Figure 70, Figure 71, se verifica que la funcion tiene orden de complejidad lineal.

3.2 Implementar los algoritmos de Kruskal

Como ya conocemos la complejidad del algoritmo, no es necesario realizar el analisis a priori, por lo tanto graficamos directamente la complejidad $Ø(mlog_2(n))$, considerando m = n.

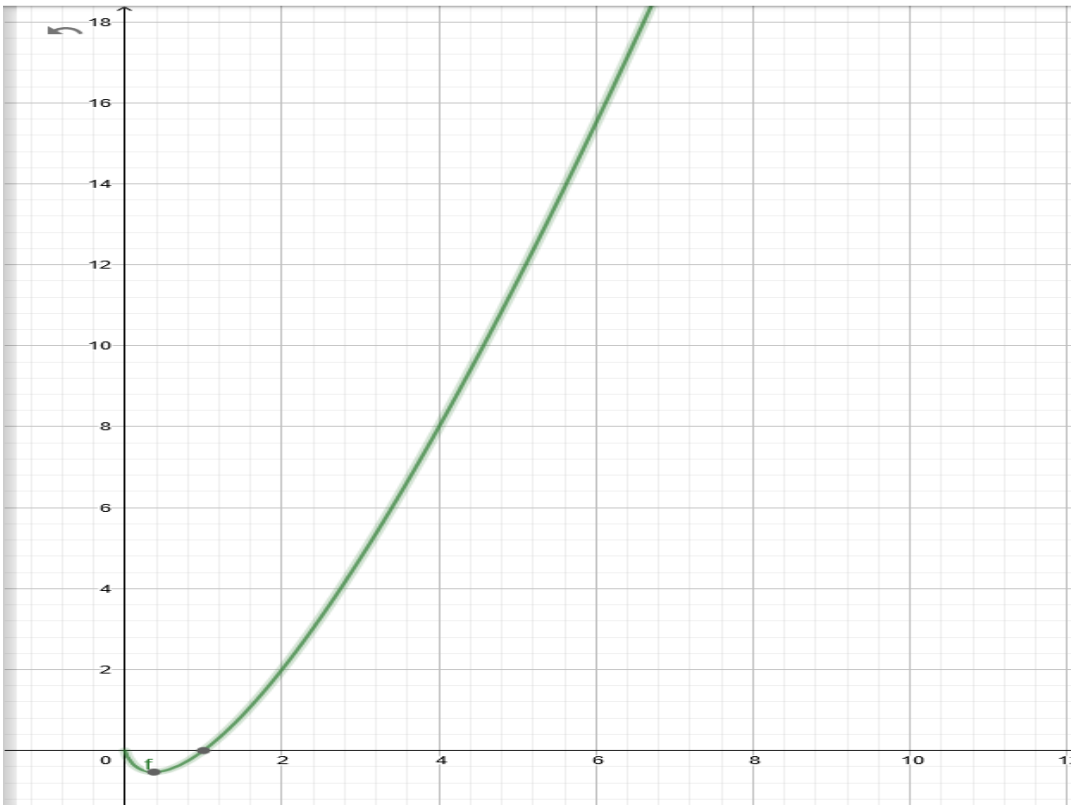

Figure 74: Grafica de la funcion $f(x) = xlog_2(x)$

Ahora realizamos la implementacion del algoritmo para calcular su complejidad a posteriori. Se generan arboles aleatorios de n nodos y n aristas, donde ($1 \leq n \leq 10000$) y graficamos la correspondiente informacion para validar la proposicion de complejidad.

Figure 75: Ejecucion del algoritmo programado

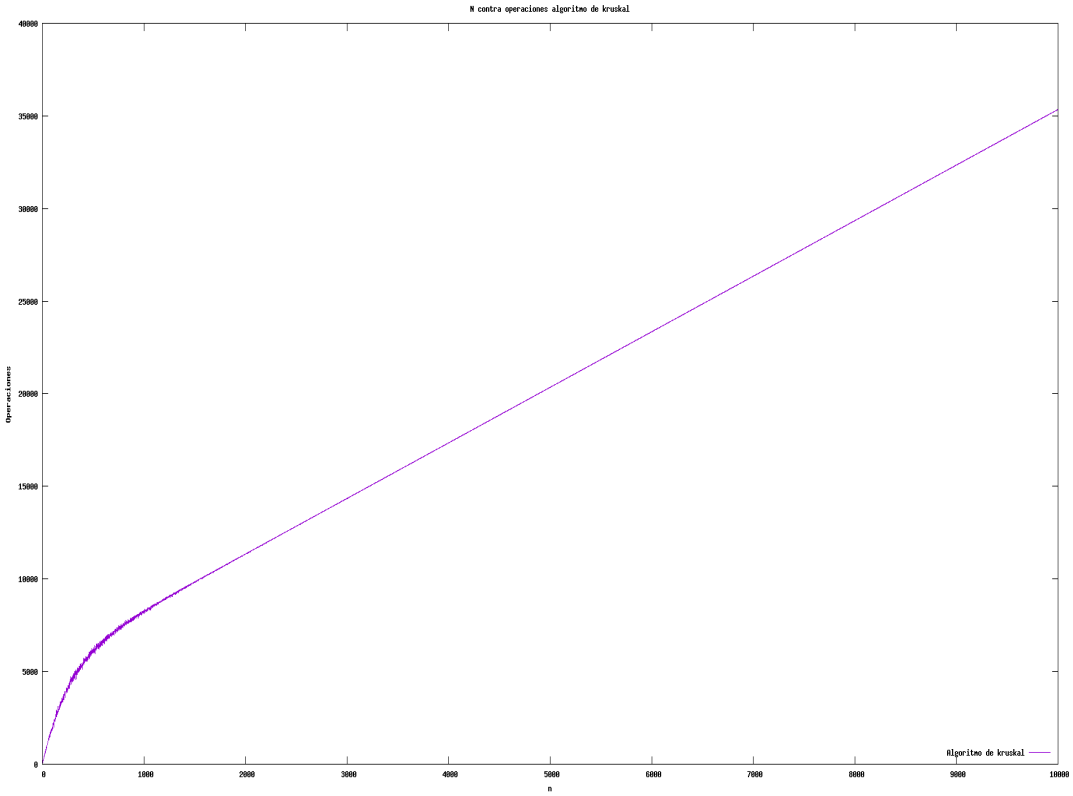

Figure 76: Grafica de n contra operaciones del algoritmo de Kruskal

4 Conclusiones

4.1 Payán Téllez René

Durante el desarrollo de esta practica hubo muchas cosas inesperadas, como el hecho de que por usar la STL no se pudo medir de forma eficiente la cantidad de operaciones, o que al momento de generar los archivos comprimidos, el metodo de encodeo de C/C++ los mantuvo relativamente del mismo tamaño que los originales. Tambien ocurrio que al momento de guardar el archivo comprimido y volverlo a leer sucedieron varios errores (se puede ver en los tamaños de los archivos), ya que se guardaba el caracter "EOF" o "End Of File", probocando que las funciones getfc dejaran de leer al momento de importar, esto sucedia ya que este caracter era uno posible por su codigo ascii (0x05), asi que se genero siempre que existiera la combinacion binaria "0000101". Aunque fue bastante interesar leer sobre el algoritmo de Kruskal, ya que es algo que estamos haciendo a mano en la clase de Metodos Cuantitativos para la Toma de Decisiones, entonces al menos con eso se que uno de sus usos en la vida real es el analisis de costos y resolucion del menor costo en un proyecto o menor tiempo, tambien sirve para encontrar la ruta critica.

5 Anexo

6 Bibliografia

[1]http://www.lcc.uma.es/~av/Libro/CAP3.pdf
[2]https://medium.com/@joseguillermo_/qu%C3%A9-es-la-complejidad-algor%C3%ADtmica-y-con-qu%C3%A9-se-come-2638e7fd9e8c
[3]https://www.tamps.cinvestav.mx/~ertello/algorithms/sesion15.pdf
[4]http://elvex.ugr.es/decsai/algorithms/slides/4%20greedy.pdf
[5]https://riptutorial.com/es/algorithm/example/23995/codificacion-huffman
[6]https://es.wikipedia.org/wiki/Algoritmo_de_Kruskal
[7]https://www.wextensible.com/temas/voraces/kruskal-prim.html
[8]https://www.ecured.cu/Algoritmo_de_Kruskal