Instituto Politécnico Nacional
Escuela Superior de Cómputo

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

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: QuickSort,SubArreglo maximo,divide y venceras,C/C++

1 Introduccion

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 ocurren
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

Algoritmo 3: MaxCrossingSubArray(A[0,...,n-1],bajo,mitad,alto)

```
Data: Entrada: A[0,...,n-1], bajo, mitad, alto
Result: Retorna la suma del mayor sub arreglo de la izquierda, la derecha y juntos
suma_izq=INT_MIN;
suma=0;
max_izq=0;
for i ← mitad to bajo do
    suma+=A[i];
    if suma > suma_izq then
        suma_izq=suma;
        max_izq=i;
suma_der=INT_MIN;
suma=0;
max_der=0;
for j ← mitad to alto do
    suma+=A[j];
    if suma > suma_der then
        suma_der=suma;
        max_der=j;
return (max_izq,max_der,suma_der+suma_izq);
```

Algoritmo 4: MaxSubArrayDC(A[0,...,n-1],bajo,alto)

```
Data: Entrada: A[0,...,n-1], bajo, alto
Result: retorna la suma y los indices del mayor sub arreglo, que se puede obtener dentro
        del arreglo A
if alto == bajo then
    return(bajo,alto,A[bajo]);
else
    mitad=(bajo+alto)/2;
    (bajo_izq,alto_izq,suma_izq)=MaxSubArrayDC(A,bajo,mitad);
    (bajo_der,alto_der,suma_der)=MaxSubArrayDC(A,mitad+1,alto);
    (cruz_izq,cruz_der,suma_cruz)=MaxCrossingSubArray(A,bajo,mitad,alto);
    if suma_izq > suma_der and suma_izq > suma_cruz then
        return (bajo_izq,alto_izq,suma_izq);
    else if suma_der > suma_izq and suma_der > suma_cruz then
        return (bajo_der,alto_der,suma_der);
    else
        return (cruz_izq,cruz_der,suma_cruz);
return (max_izq,max_der,suma_der+suma_izq);
```

Algoritmo 5: FuerzaBruta(A[0,...,n-1])

Data: Entrada: A[0,...,n-1]
Result: Retorna la suma y los indices del mayor sub arreglo, que se puede obtener dentro del arreglo A

```
sumaMaxima=−∞;
indiceIzquierdo = 0;
indiceDerecho = 0;
for i ← 0 to n do
    sumaLocal=0;
    for j ← i to n do
        sumaLocal+=A[j];
        if sumaLocal>sumaMaxima then
            sumaMaxima = sumaLocal;
            indiceIzquierdo=i;
            indiceDerecho=j;
return(sumaMaxima,indiceIzquierdo,indiceDerecho);
```

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

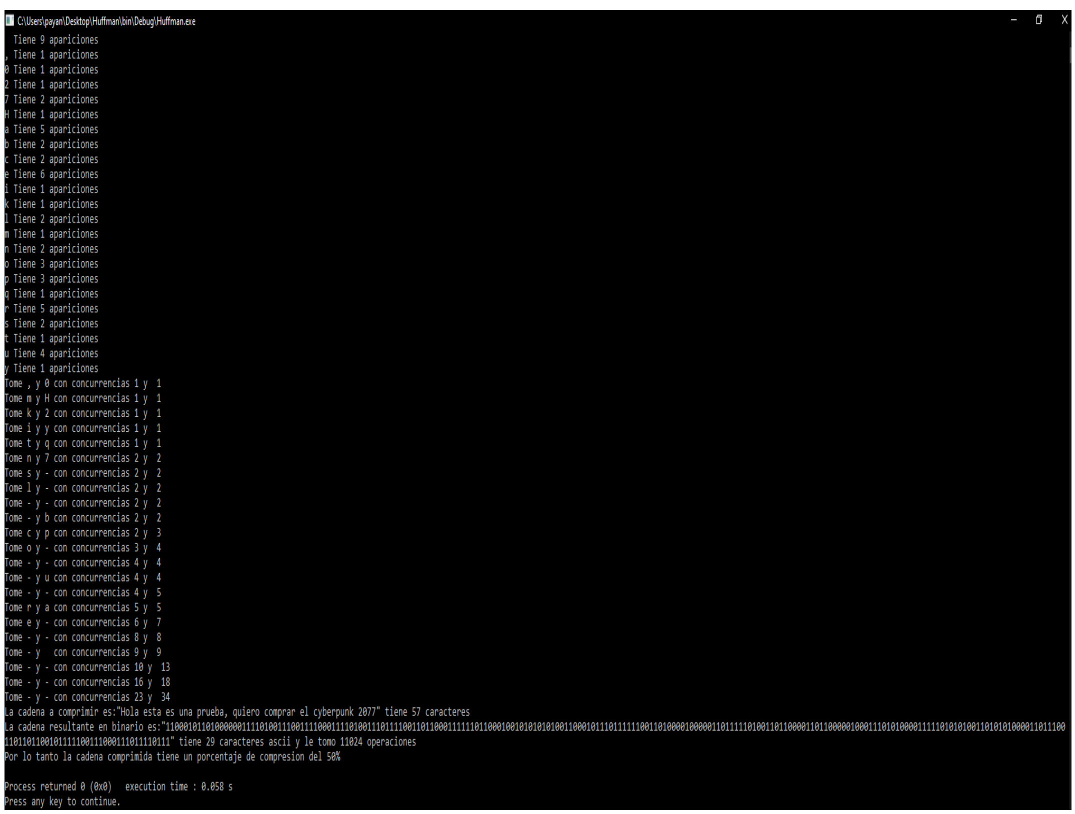


Figure 1: Compresión de la cadena ”Hola esta es una prueba, quiero comprar el cyberpunk 2077”

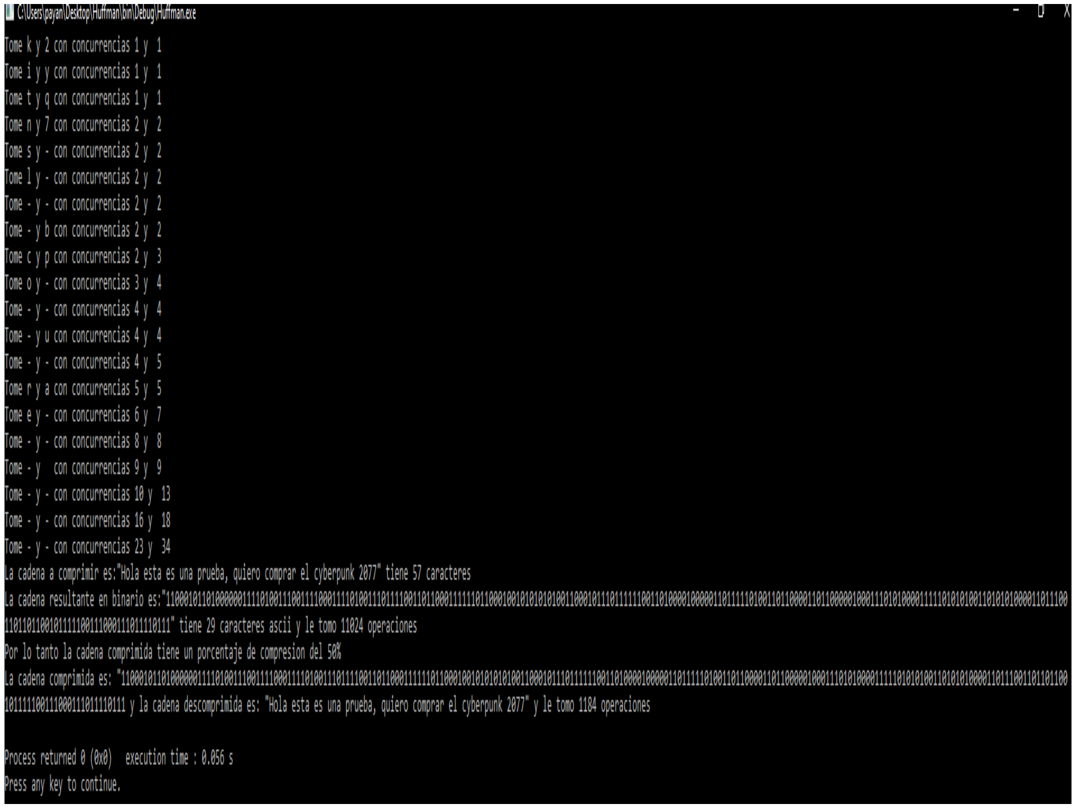


Figure 2: Descompresión de la cadena ”Hola esta es una prueba, quiero comprar el cyberpunk 2077”

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:

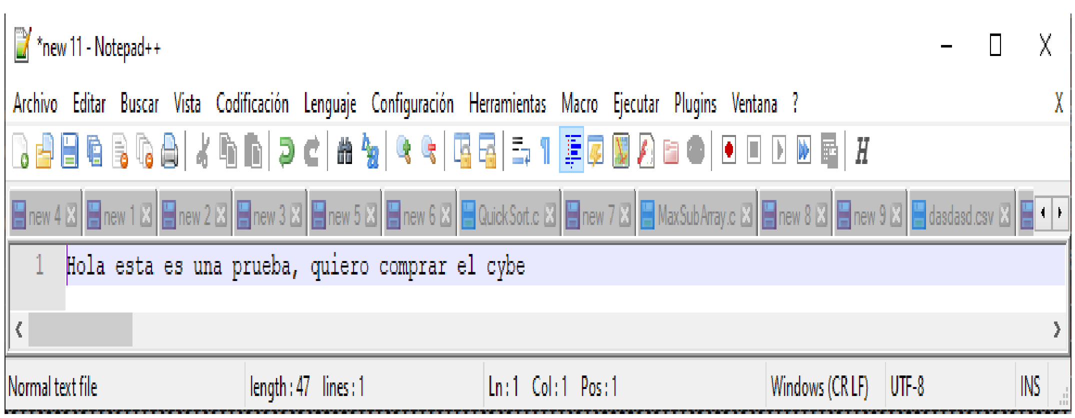


Figure 3: Conteo de caracteres de la entrada

esta es la cantidad de caracteres de la salida:

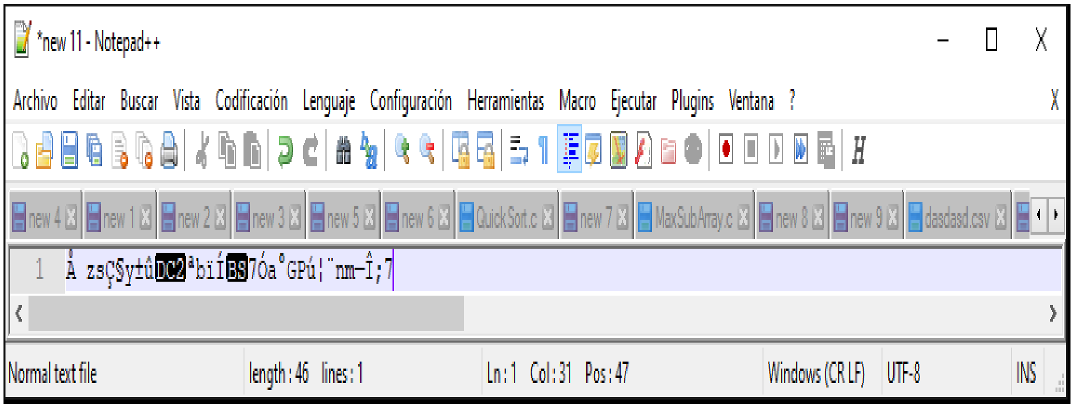


Figure 4: Conteo de caracteres de la salida

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.

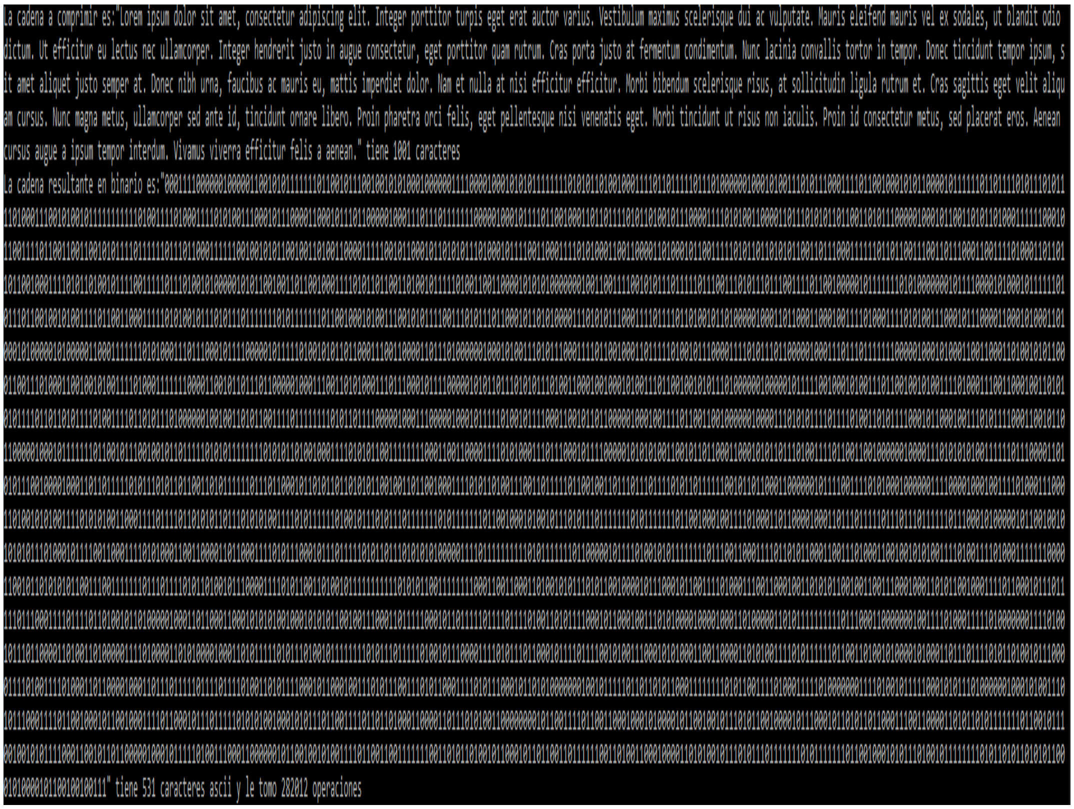


Figure 6: Compresión del lorem ipsum

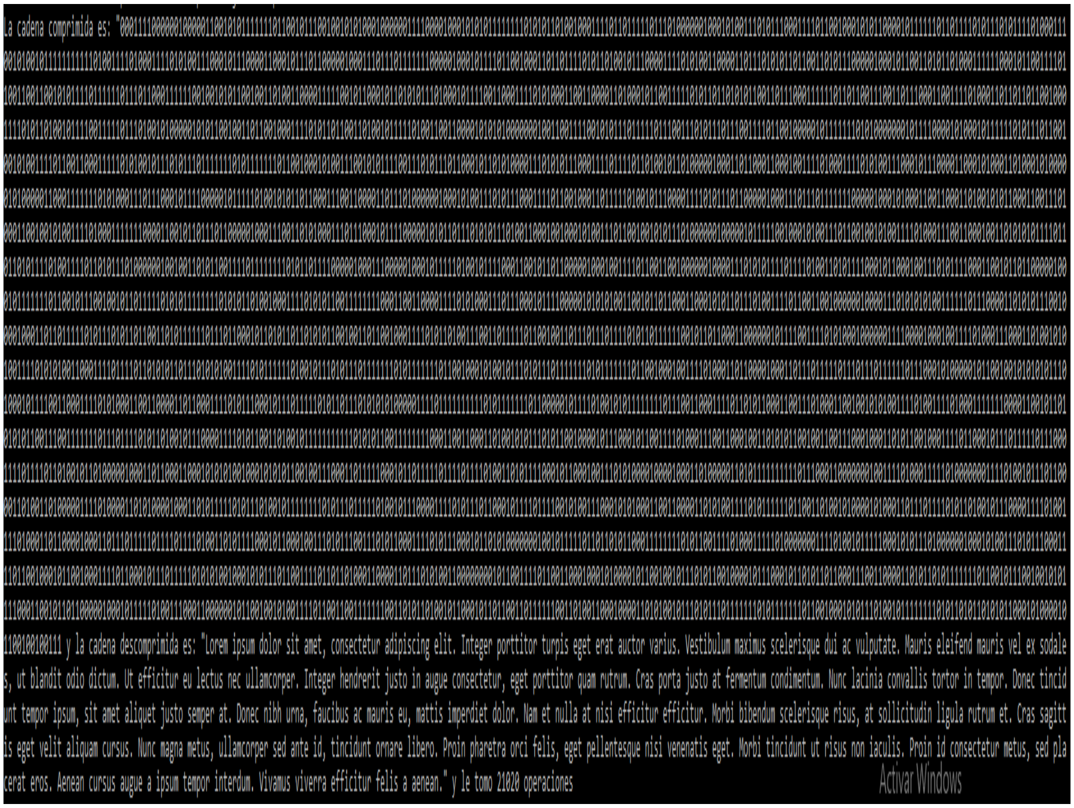


Figure 7: Descompresión del lorem ipsum

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

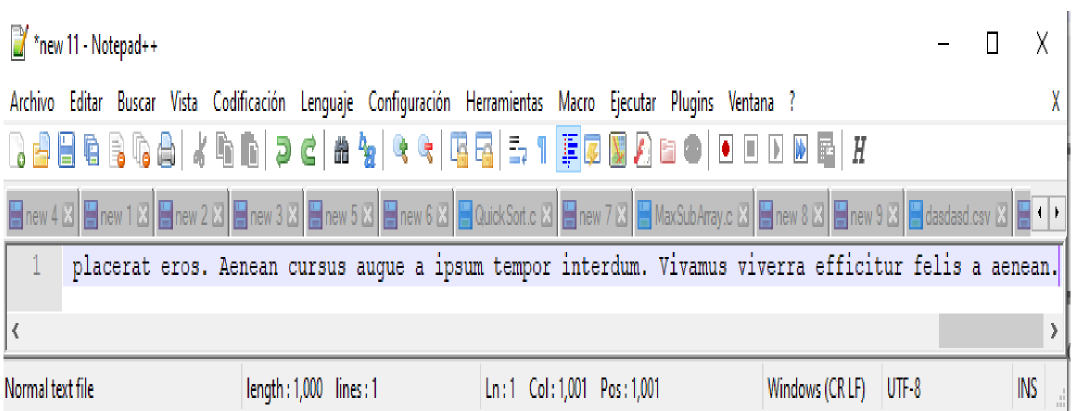


Figure 8: Conteo de caracteres de la entrada

y esta es la cantidad de caracteres de la salida:

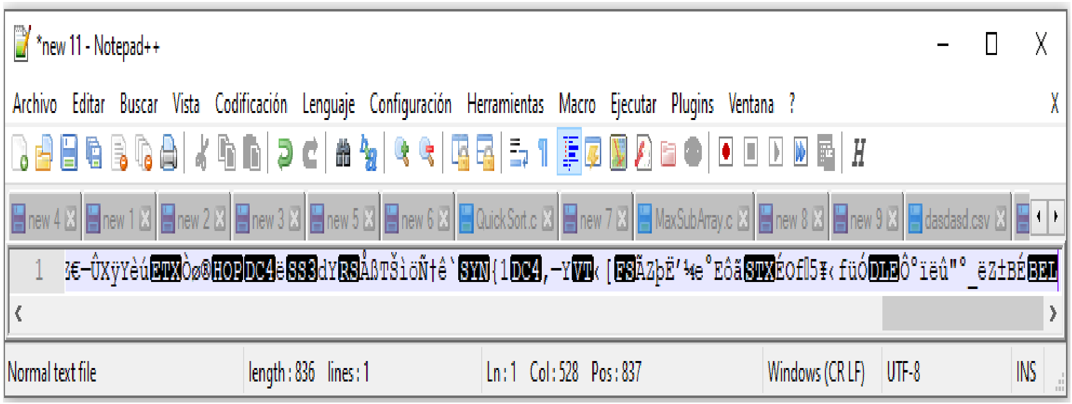
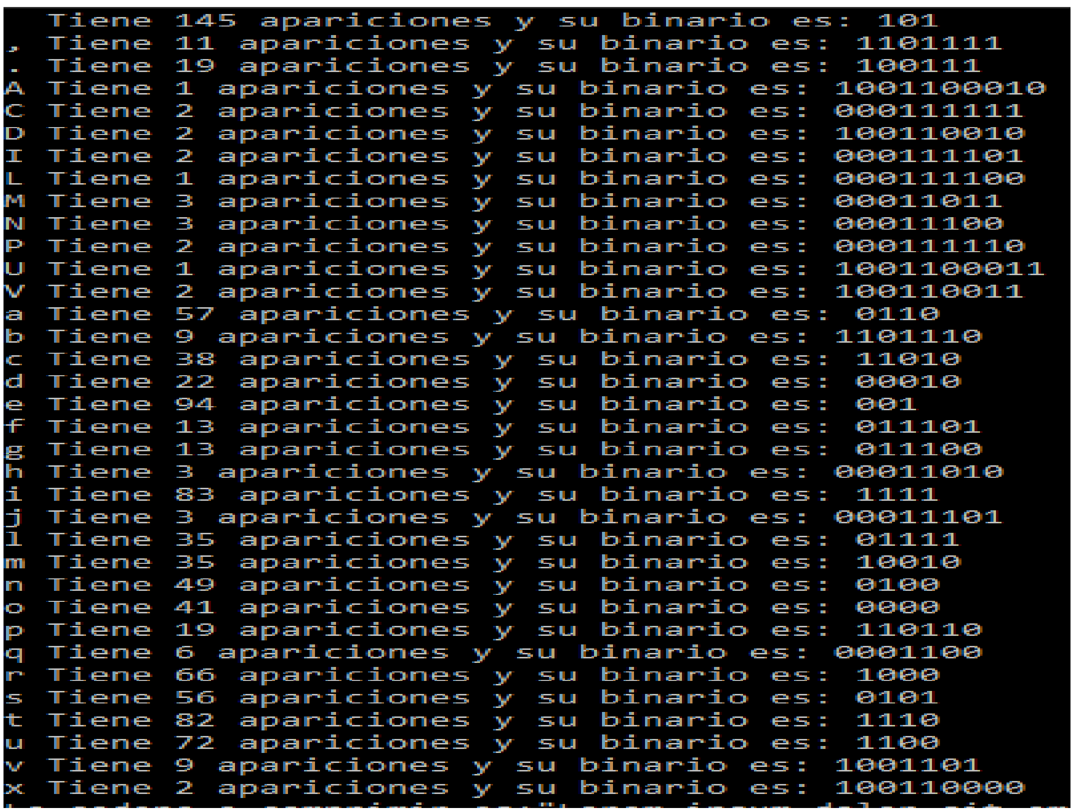


Figure 9: Conteo de caracteres de la salida

por ultimo, el binario de cada caracter asignado



```
  Tiene 145 apariciones y su binario es: 101
, Tiene 11 apariciones y su binario es: 1101111
. Tiene 19 apariciones y su binario es: 100111
A Tiene 1 apariciones y su binario es: 1001100010
C Tiene 2 apariciones y su binario es: 000111111
D Tiene 2 apariciones y su binario es: 100110010
I Tiene 2 apariciones y su binario es: 000111101
L Tiene 1 apariciones y su binario es: 000111100
M Tiene 3 apariciones y su binario es: 00011011
N Tiene 3 apariciones y su binario es: 00011100
P Tiene 2 apariciones y su binario es: 000111110
U Tiene 1 apariciones y su binario es: 1001100011
V Tiene 2 apariciones y su binario es: 100110011
a Tiene 57 apariciones y su binario es: 0110
b Tiene 9 apariciones y su binario es: 1101110
c Tiene 38 apariciones y su binario es: 11010
d Tiene 22 apariciones y su binario es: 00010
e Tiene 94 apariciones y su binario es: 001
f Tiene 13 apariciones y su binario es: 011101
g Tiene 13 apariciones y su binario es: 011100
h Tiene 3 apariciones y su binario es: 00011010
i Tiene 83 apariciones y su binario es: 1111
j Tiene 3 apariciones y su binario es: 00011101
l Tiene 35 apariciones y su binario es: 01111
m Tiene 35 apariciones y su binario es: 10010
n Tiene 49 apariciones y su binario es: 0100
o Tiene 41 apariciones y su binario es: 0000
p Tiene 19 apariciones y su binario es: 110110
q Tiene 6 apariciones y su binario es: 0001100
r Tiene 66 apariciones y su binario es: 1000
s Tiene 56 apariciones y su binario es: 0101
t Tiene 82 apariciones y su binario es: 1110
u Tiene 72 apariciones y su binario es: 1100
v Tiene 9 apariciones y su binario es: 1001101
x Tiene 2 apariciones y su binario es: 100110000
```

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

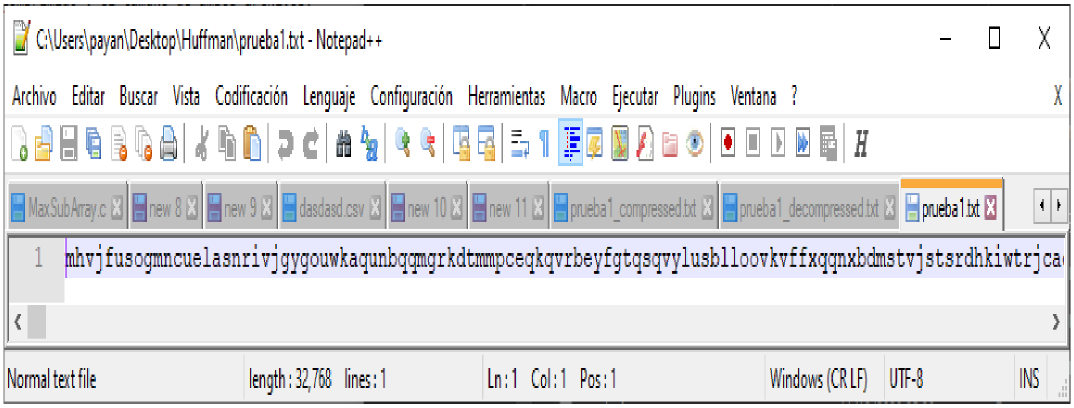


Figure 11: Archivo de entrada

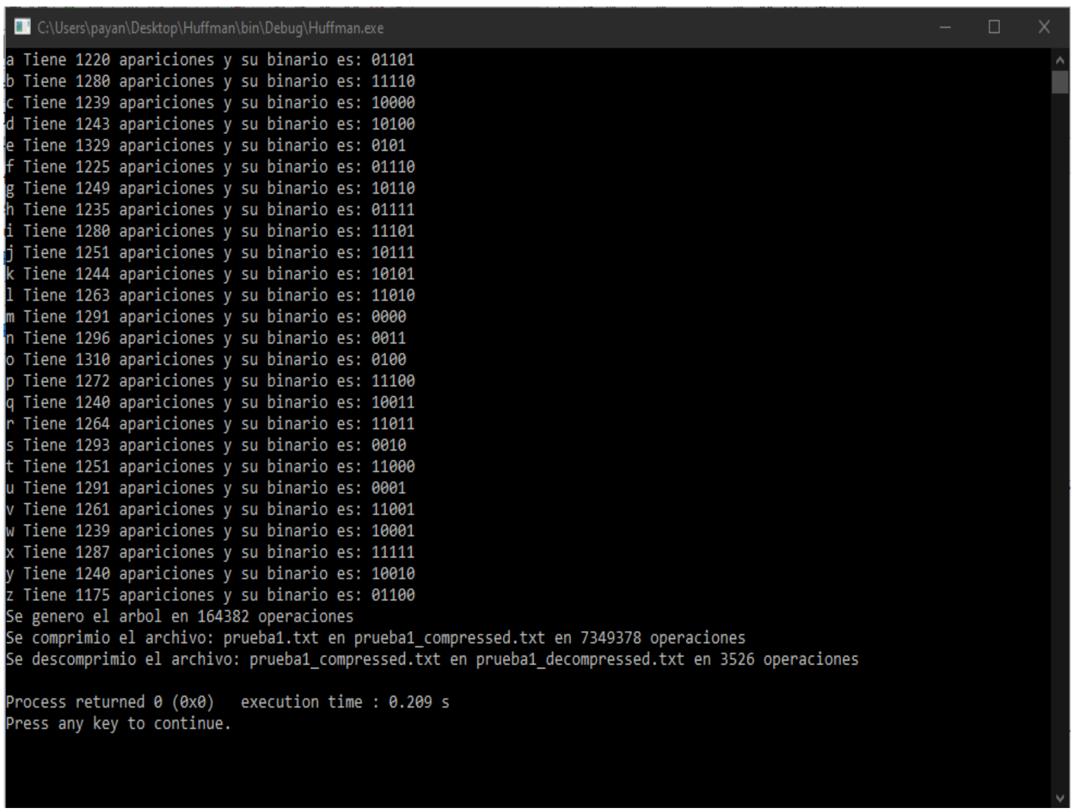


Figure 12: Ejecucion del programa

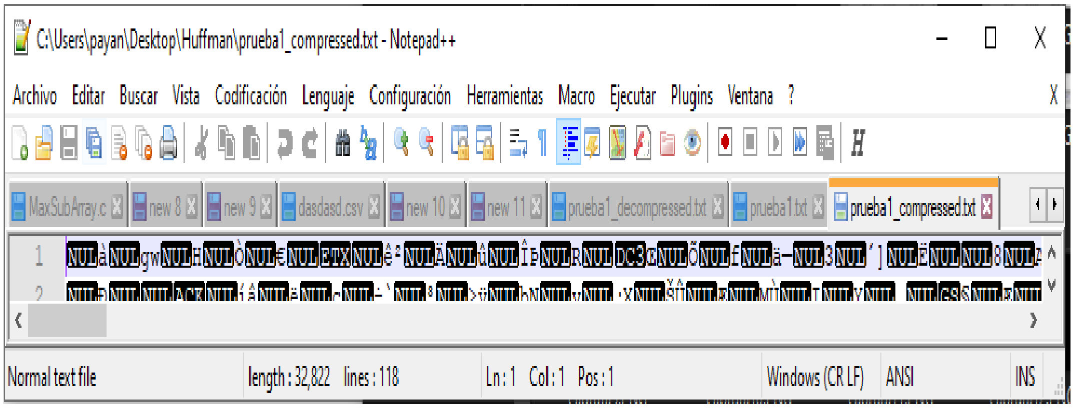


Figure 13: Archivo de salida

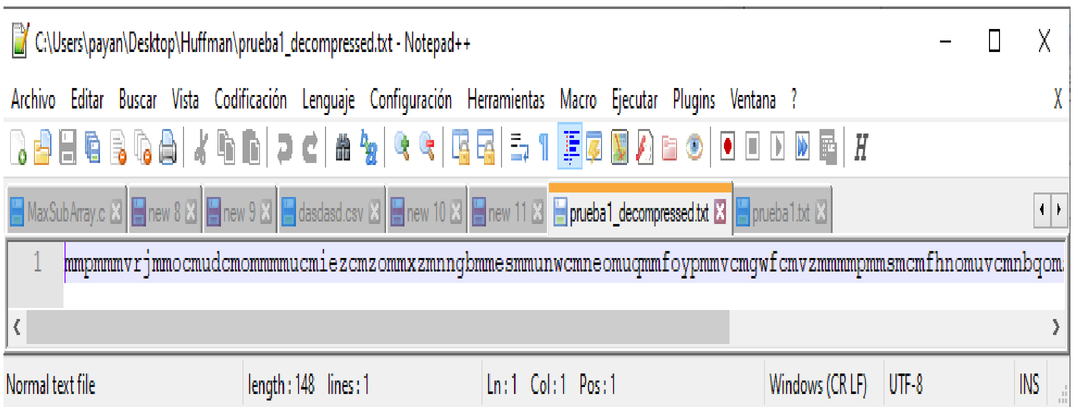


Figure 14: Archivo descomprimido

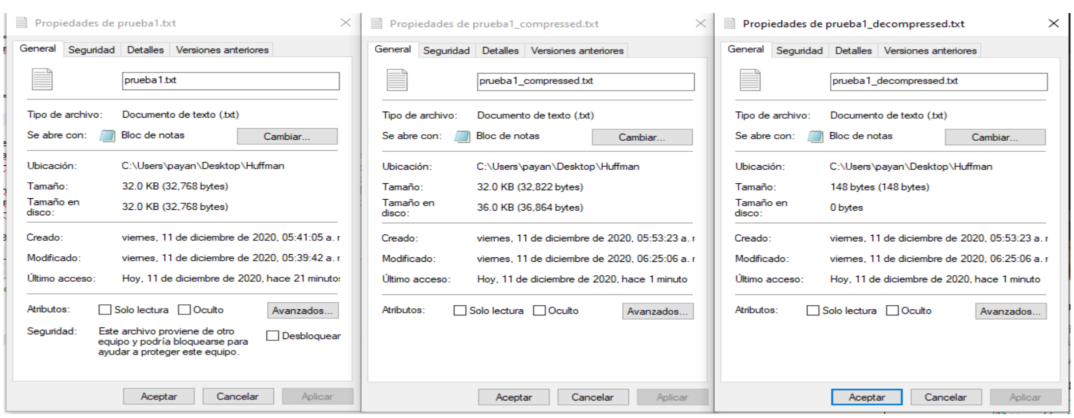


Figure 15: peso de los 3 archivos

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.

4 Conclusiones

4.1 Payán Téllez René

Esta practica se me hizo particulamente interesante porque los algoritmos que se trataron, no fueron tan directos de implementar en un lenguaje de programación, sin mencionar que algunas graficas tuvieron muy pocos valores, debido a lo complicado que era generar mas, porque computacionalmente su complejidad es inmensa. De hecho tuve que cambiar algunas variables de int a long long int en el espacio de los contadores para que siguiera funcionando el contador sin desbordarse. Tambien vi lo interesante de un algoritmo como MostrarPerfectos que tiene una complejidad no polinomial, ya que aunque lo puedo programar tardaria horas en encontrar mas alla del perfecto 5 (de por si toma mas de 5 minutos hallar el perfecto 4) sin mencionar que le tomaria dias encontrar otros números. Tambien cuando estaba demostrando la complejidad del algoritmo recursivo de la secuencia de fibonacci, algunos nucleos del CPU de mi computadora se dispararon, lo cual fue una señal de lo complicado y tardado que se podia volver en N muy grandes.

5 Anexo

5.1 Investigar el algoritmo de Karatsuba que permite obtener la multiplicación de enteros muy grandes

5.2 Resolver los siguientes problemas

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