

Complejidad de las Funciones HummanStructures.

Función getbits O(n)

```
void getBits(struct node* HuffmanTree, struct bits bytesCode[], int bits, int sizeBits){  
    if(!isEmpty(HuffmanTree)){//if notempty  
        getBits(HuffmanTree->left, bytesCode, (bits << 1), sizeBits + 1); //irte al nodo izq, sumandole un 0  
        if(isLeaf(HuffmanTree)){  
            int hashKey = HuffmanTree->data.byte;  
            bytesCode[hashKey].bits = bits;  
            bytesCode[hashKey].byte = HuffmanTree->data.byte;  
            bytesCode[hashKey].sizeBits = sizeBits;  
        }  
        getBits(HuffmanTree->right, bytesCode, (bits<<1) + 1, sizeBits + 1); //irte al nodo der sumandole un 1  
    }  
}
```

Handwritten annotations for `getBits`:
- $T(n-1) = O(n)$ (blue)
- $O(1)$ (orange, next to leaf check)
- $O(1)$ (orange, next to hashKey assignment)
- $O(1)$ (orange, next to bytesCode assignment)
- $O(1)$ (orange, next to bytesCode assignment)
- $O(n)$ (red, next to recursive call)
- $T(n-1) = O(n)$ (green, at bottom)

Nota: hace una recursión $T(n-1)$ con cota de complejidad de $O(n)$ y por jerarquía queda

$O(n)$ Función CreateHeap $O(1)$

```
Heap *CreateHeap(int capacity){  
    Heap *heap = (Heap *)malloc(sizeof(Heap));  
    heap->count = 0;  
    heap->capacity = capacity;  
    heap->arrayOfNodes = (struct node**)malloc(capacity * sizeof(struct node));  
    return heap;  
}
```

Handwritten annotations for `CreateHeap`:
- $O(1)$ (red, next to malloc for Heap)
- $O(1)$ (red, next to count assignment)
- $O(1)$ (red, next to capacity assignment)
- $O(1)$ (red, next to arrayOfNodes malloc)
- $O(1)$ (red, next to return)
- $O(1)$ (red, overall complexity)

Funcion mergeTrees $O(n \log n)$

```
//unir todos los nodos en uno solo  
struct node* mergeTrees(Heap* heap){  
    while(heap->count > 1){  
        struct node* node1 = PopMin(heap);  
        struct node* node2 = PopMin(heap);  
        insert(heap, mergeNodes(node1, node2));  
    }  
    return PopMin(heap);  
}
```

Handwritten annotations for `mergeTrees`:
- $O(\log n)$ (red, next to while loop)
- $O(n)$ (yellow, next to PopMin calls)
- $O(n)$ (yellow, next to insert call)
- $O(n \log n)$ (red, overall complexity)

Nota se utiliza el valor de la complejidad insert y dentro de insert se llama a la funcion mergeNodes con pmplejidad de $O(1)$ ya que es llamada durante la funcion mergeTrees al gual que PopMin es llamada dos veces por ciclo y tiene una complejidad $O(n)$.

Complejidad de los programas Compress y Descompress

Funcion `heapify_bottom_Top` $O(n)$

```
void heapify_bottom_top(Heap *heap, int index){  
    struct node* temp;  
    int parent_node = (index-1)/2;  
    if(heap->arrayOfNodes[parent_node]->data.frequency > heap->arrayOfNodes[index]->data.frequency){  
        temp = heap->arrayOfNodes[parent_node];  
        heap->arrayOfNodes[parent_node] = heap->arrayOfNodes[index];  
        heap->arrayOfNodes[index] = temp;  
        heapify_bottom_top(heap, parent_node);  
    }  
}
```

$O(n)$

$T(n-1) = O(n)$

$O(n)$

Función `insert` $O(n)$

```
void insert(Heap *heap, struct node* node){  
    if(heap->count < heap->capacity){  
        heap->arrayOfNodes[heap->count] = node;  
        heapify_bottom_top(heap, heap->count);  
        heap->count++;  
    }  
}
```

$O(1)$

$O(n)$

Nota: se utilizó la complejidad de `heapify_bottom_top` ya que es usada en la función

Función `PushTree` $O(1)$

```
void pushTree(struct node* root, unsigned char byte, int frequency){  
    root->data.frequency = frequency;  
    root->data.byte = byte;  
    root->left = NULL;  
    root->right = NULL;  
}
```

$O(1)$

Complejidad de los programas Compress y Descompress

Función InsertTree $O(n^2)$

```
/**
//Funciones con el arbol y heap
//formar en el arbol las frecuencias y formar esos arboles en la cola
void insertTree(struct data bytesFrequency[], struct node roots[], Heap* heap){
    int i;
    for(i = 0; i < 256; i++){
        if(bytesFrequency[i].frequency > 0){
            pushTree(&roots[i], bytesFrequency[i].byte, bytesFrequency[i].frequency);
            insert(heap, &roots[i]);
        }
    }
}
```

Handwritten annotations for the `insertTree` function:

- $O(\ln)$ next to the `for` loop.
- $O(n)$ next to the `if` statement.
- $O(n)$ next to the `pushTree` call.
- $O(n)$ next to the `insert` call.
- A bracket on the right side of the function body indicates a total complexity of $O(n^2)$.

Nota se toma la complejidad de insert ya que es llamada en la función.

Función mergeNodes $O(1)$

```
//unir los nodos
struct node* mergeNodes(struct node* node1, struct node* node2){
    struct node* new_node = malloc(sizeof(struct node));
    new_node->data.byte = 0;
    new_node->left = node1;
    new_node->right = node2;
    new_node->data.frequency = node1->data.frequency + node2->data.frequency;
    return new_node;
}
```

Handwritten annotation for the `mergeNodes` function:

- $O(1)$ next to the function body, indicating constant time complexity.

Complejidad de los programas Compress y Descompress

Funcon heapify_Top_bottom $O(n)$

```
void heapify_top_bottom(Heap *heap, int parent_node){
    int left = parent_node * 2 + 1;
    int right = parent_node * 2 + 2;
    int min;
    struct node* temp;

    if(left >= heap->count || left < 0)
        left = -1;
    if(right >= heap->count || right < 0)
        right = -1;

    if(left != -1 && heap->arrayOfNodes[left]->data.frequency < heap->arrayOfNodes[parent_node]->data.frequency)
        min = left;
    else
        min = parent_node;
    if(right != -1 && heap->arrayOfNodes[right]->data.frequency < heap->arrayOfNodes[min]->data.frequency)
        min = right;

    if(min != parent_node){
        temp = heap->arrayOfNodes[min];
        heap->arrayOfNodes[min] = heap->arrayOfNodes[parent_node];
        heap->arrayOfNodes[parent_node] = temp;

        heapify_top_bottom(heap, min);
    }
}
```

Nota hace una recursión N veces por lo que su cota de complejidad es $O(n)$

Función PopMin $O(1)$

```

struct node* PopMin(Heap *heap){
    struct node* pop;
    if(heap->count == 0){
        printf("Heap is Empty.\n");
        return NULL;
    }
    pop = heap->arrayOfNodes[0];
    heap->arrayOfNodes[0] = heap->arrayOfNodes[heap->count-1];
    heap->count--;
    heapify_top_bottom(heap, 0);
    return pop;
}

```

Nota llama a la funcion `heapify_Top_bottom` que tiene una complejidad de $O(n)$

Complejidad de los programas Compress y Descompress

Funcion getCharacters O(n)

```
int getCharacters(struct node* HuffmanTree, unsigned char* cadena, int posInString, int posInBits, unsigned char* byteToWrite){
    if(isLeaf(HuffmanTree)){
        *byteToWrite = HuffmanTree->data.byte;
        return posInBits;
    }
    else{
        if(posInBits < 0){
            (*posInString)++;
            posInBits = 7;
        }
        if(((int)CONSULTARBIT(cadena[*posInString]), (posInBits))) == 0)
            return getCharacters(HuffmanTree->left, cadena, posInString, (posInBits)-1, byteToWrite);
        else
            return getCharacters(HuffmanTree->right, cadena, posInString, posInBits-1, byteToWrite);
    }
}
```

Handwritten annotations:

- $O(1)$ next to `*byteToWrite = HuffmanTree->data.byte;`
- $O(1)$ next to `return posInBits;`
- $O(1)$ next to `if(posInBits < 0){`
- $O(1)$ next to `(*posInString)++;`
- $O(1)$ next to `posInBits = 7;`
- $T(n-1) = O(n)$ next to the recursive call `return getCharacters(HuffmanTree->left, ...)`
- $T(n-1) = O(n)$ next to the recursive call `return getCharacters(HuffmanTree->right, ...)`
- A large yellow bracket on the right side of the function, spanning from the first recursive call to the end of the function, is labeled $O(n)$.

Nota: hace una recursión $T(n-1)$ con cota de complejidad de $O(n)$ y por jerarquía queda $O(n)$

Complejidad de Funciones de CompressHuffman

Función writeFrequencyTable $O(n)$

```
void writeFrequencyTable(struct data bytesFrequency[], int fileSize)
{
    int i;
    FILE *frequencyTable;

    //Abrimos y verificamos que si se abrió correctamente
    frequencyTable = fopen("frequencyTable.txt", "wt");
    if (frequencyTable == NULL)
    {
        puts("The file could not be opened.\n");
        exit(1);
    }

    //Escribimos el tamaño original al inicio del txt
    fprintf(frequencyTable, "%d\n", fileSize);
    for (i = 0; i < 256; i++)
    {
        if (bytesFrequency[i].frequency > 0) //Si el símbolo aparecía en el archivo, escríbelo
            fprintf(frequencyTable, "%d %d\n", bytesFrequency[i].byte, bytesFrequency[i].frequency);
    }
    fclose(frequencyTable);
}
```

Handwritten annotations for `writeFrequencyTable`:

- `int i;` and `FILE *frequencyTable;` are grouped with $O(1)$.
- `frequencyTable = fopen(...)` and the `if` block are grouped with $O(1)$.
- `fprintf(frequencyTable, "%d\n", fileSize);` is marked with $O(1)$.
- The `for` loop and its body are grouped with $O(n)$.
- `fclose(frequencyTable);` is marked with $O(1)$.
- A red bracket on the right side of the function is labeled $O(n)$.
- A note "Por jerarquía $O(n)$ " is written in blue.

Función ReadFile $O(n)$

```
unsigned char *readFile(const char *fileToOpen, struct data bytesFrequency[], int *fileSize)
{
    int i;
    FILE *file;
    unsigned char c;

    //Abrimos y verificamos que si se abrió correctamente
    file = fopen(fileToOpen, "rb");
    if (file == NULL)
    {
        puts("The file could not be opened.\n");
        exit(1);
    }

    //Obtenemos el tamaño del archivo.
    fseek(file, 0L, SEEK_END);
    (*fileSize) = ftell(file);
    rewind(file);

    //Reservamos memoria para la cadena de bytes leídos
    unsigned char *bytesRead = malloc((*fileSize) * sizeof(unsigned char));

    //Guardamos los bytes leídos y su frecuencia con su valor en decimal
    for (i = 0; i < (*fileSize); i++)
    {
        fread(&c, sizeof(unsigned char), 1, file);
        bytesRead[i] = c;
        bytesFrequency[c].byte = c; // Enmascaramiento
        bytesFrequency[c].frequency++; // frecuencia de apariciones
    }
    fclose(file);
    return bytesRead;
}
```

Handwritten annotations for `readFile`:

- `int i;` and `FILE *file;` are grouped with $O(1)$.
- `unsigned char c;` is marked with $O(1)$.
- `file = fopen(...)` and the `if` block are grouped with $O(1)$.
- `fseek`, `ftell`, and `rewind` are grouped with $O(1)$.
- `malloc` is marked with $O(1)$.
- The `for` loop and its body are grouped with $O(n)$.
- `fclose(file);` is marked with $O(1)$.
- `return bytesRead;` is marked with $O(1)$.
- A red box on the right side of the function is labeled "Por jerarquía $O(n)$ ".
- A note "Por jerarquía $O(n)$ " is written in green.

Complejidad de los programas Compress y Descompress

Funcion WriteBinaryCode es $O(n \log n)$

```
void writeBinaryCode(unsigned char *bytesHead, struct node *mainTree, struct bits *byteCode, int *compressibleFileSize)
{
    int i, j, bitsizeofWrite = 0;
    FILE *binaryCode;

    //Comprobamos y verificamos que si se abren correctamente
    binaryCode = fopen("ByteCode.dat", "wb");
    if (binaryCode == NULL)
    {
        puts("The file could not be opened.\n");
        exit(1);
    }

    //Preparamos para poder escribir los bits
    int bitsize, sizeofWrite, tempSize = 0;
    int bits, tempbits, tempSizebits, tempSizebits;
    unsigned char tempbits[16];

    for (i = 0; i < fileSizes; i++)
    {
        bitsize = ByteCode[bytesHead[i]].bitsize;
        bits = ByteCode[bytesHead[i]].bits;

        //Creamos el Array bits que no escribimos
        if (tempSize != 0)
        {
            bits = (tempbits << bitsize) + bits;
            bitsize = tempbits;
            tempSize = 0;
            tempSize = 0;
        }

        //Creamos escribir o no el bit
        if (bitsize == 1)
        {
            if (bitsize == 0)
            {
                //Creamos los bits de los que podemos escribir, hay que reducir
                //Creamos los que no podemos escribir, los que no podemos escribir
                tempSize = bitsize - 1;
                for (j = 0; j < tempSize; j++)
                {
                    tempbits[j] = (tempbits[j] << 1) + bits;
                    tempbits[j] = tempbits[j] >> tempbits[j];
                }
                //Ajustamos los bits para poder escribir los primeros 8
                bits = bits >> tempSize;
                sizeofWrite = bits;
                bitsizeofWrite = 0;
            }
            else
            {
                tempSize = bits;
                tempSize = 0; // Cuando nos en una variable temporal
                tempSize = 0; // Lo que no pudimos escribir
            }
        }

        // Hay bytes por escribir
        if (sizeofWrite != 0)
        {
            sizeofWrite = 0;
            fwrite(ByteCode[bytesHead[i]].bits, sizeof(unsigned char), 1, binaryCode);
        }

        //Creamos que no se desborde tempSize
        while (tempSize >= 8)
        {
            //Ajustamos tempSize para poder escribir los primeros 8
            tempSize = tempSize - 8;
            tempSizebits = tempSize;
            bits = tempbits >> tempSizebits;
            tempSizebits = 0;
        }

        //Cuando nos el exceso en una variable temporal
        for (j = 0; j < tempSize; j++)
        {
            tempbits[j] = ((CONQUETAR[tempSizebits] << 1) + bits);
            tempbits[j] = tempbits[j] >> tempbits[j];
        }
        sizeofWrite = bits;
        fwrite(ByteCode[bytesHead[i]].bits, sizeof(unsigned char), 1, binaryCode);
    }
}
```

Nota: tomamos los dos for como $O(1)$ ya que timesize sera maximo 32

Diagram illustrating the recursive steps of Merge Sort:

- $O(1)$ (green)
- $O(1)$ (purple)
- $O(1)$ (orange)
- $O(\log n)$ (orange)
- $O(n \log n)$ (green and blue)

Por jerarquía es $O(\log n)$

Por jerarquía es $O(\log n)$

Complejidad del Programa CompressHuffman

$O(n^2)$

```

int main(int argc, char *argv[])
{
    //*****
    //Variables a usar en el programa

    //Variables para el control de tiempo
    double utime0, stime0, wtime0, utime1, stime1, wtime1;
    //Arreglo para almacenar la frecuencia de bytes
    struct data *bytesFrequency = malloc(256 * sizeof(struct data));
    //Arreglo para almacenar la codificación de Huffman
    struct bits *bytesCode = malloc(256 * sizeof(struct bits));
    //Arreglo para almacenar el byte y frecuencia en cada nodo del arbol
    struct node *roots = (struct node *)malloc(256 * sizeof(struct node));
    //Arbol de Huffman ya unificado
    struct node *HuffmanTree;
    //Cadena de bytes leídas del archivo original
    unsigned char *bytesRead;
    //Cola de prioridad
    Heap *heap = CreateHeap(511);
    //Variable para almacenar el tamaño del archivo y del comprimido
    int fileSize, compressedFileSize;

    puts("\nComprimir archivos usando el algoritmo de Huffman.\n");
    //*****
    // Iniciar el conteo del tiempo para las evaluaciones de rendimiento
    //*****
    uswtime(&utime0, &stime0, &wtime0);

    //Guardamos en un arreglo todos los bytes leídos
    bytesRead = readFile(argv[1], bytesFrequency, &fileSize);
    //formar en el arbol las frecuencias y formar esos arboles en la cola
    insertTree(bytesFrequency, roots, heap);
    //unir los arboles en el arbol de Huffman
    HuffmanTree = mergeTrees(heap);
    //Obtener los bits que vale cada byte
    getBits(HuffmanTree, bytesCode, &fileSize);
    //escribir el .dat con los bits de cada byte correspondiente
    writeBinaryCode(bytesRead, HuffmanTree, bytesCode, fileSize, &compressedFileSize);
    //escribir la tabla de frecuencias y el tamaño
    writeFrequencyTable(bytesFrequency, fileSize);

    //*****
    // Evaluar los tiempos de ejecución
    //*****
    uswtime(&utime1, &stime1, &wtime1);

    printf("Tamaño de %s: %d bytes\n", argv[1], fileSize);
    printf("Tamaño del archivo comprimido: %d bytes\n", compressedFileSize);
    printf("Porcentaje de compresión alcanzado: %.2f%%\n", ((float)fileSize / (float)compressedFileSize) * 100);
    printf("Tiempo real de Ejecución: %.10e s\n", wtime1 - wtime0);

    return 0;
}

```

Por jerarquías:
la complejidad es de $O(n^2)$

$O(n)$

$O(n^2)$

$O(n \log n)$

$O(n^2)$

$O(n \log n)$

$O(n)$

Nota: Son llamadas funciones donde ya se sacó su complejidad anteriormente y con jerarquía la complejidad queda como $O(n^2)$.

Complejidad Funciones de DescompressedHumman

Funcion writeFileO(n)

```
void writeFile(unsigned char* bytesRead, struct node* mainTree, const char* file, int byteFileSize, int* fileSize){
    FILE* finalFile;

    //Abrimos y verificamos que si se abrio correctamente
    finalFile = fopen(file, "wb+");
    if(finalFile == NULL){
        puts("The file could not be opened.\n");
        exit(1);
    }

    //Variables para escribir en el archivo
    int i, posInBits = 7, bytesWritten = 0;
    unsigned char byteToWrite;

    //Leemos los bits extra que escribimos
    int bitsExtraWritten = bytesRead[byteFileSize-1];

    for(i = 0; i < byteFileSize - 1 && bytesWritten < (*fileSize);){
        posInBits = getCharacters(mainTree, bytesRead, &i, posInBits, &byteToWrite);
        //escribimos el byte correspondiente a los bits
        fwrite(&byteToWrite, sizeof(unsigned char), 1, finalFile);
        bytesWritten++; //conteo para no escribir de mas

        //Cuidamos no escribir los bits extra que escribimos al compair
        if(1 == (*fileSize) - 2 && 7 - posInBits == bitsExtraWritten){
            bytesWritten++;
        }

        //Checamos que no nos desbordamos de posicion de bits
        if(posInBits < 0){
            posInBits = 7;
            i++;
        }
    }
    (*fileSize) = bytesWritten;
    fclose(finalFile);
}
```

Handwritten annotations for `writeFile`:

- Red bracket around the opening file operation and error handling, labeled $O(1)$.
- Red bracket around the loop body, labeled $O(n)$.
- Red bracket around the bit position check, labeled $O(1)$.
- Red bracket around the final file size assignment, labeled $O(1)$.
- Text "por jerarquia $O(n)$ " with an arrow pointing to the loop.

Funcion readFrequencyTable $O(\log n)$

```
void readFrequencyTable(struct data bytesFrequency[], int* fileSize){
    int byte, frequency, i = 0;
    FILE* frequencyTable;

    //Abrimos y verificamos que si se abrio correctamente
    frequencyTable = fopen("frequencyTable.txt", "r");
    if(frequencyTable == NULL){
        puts("The file could not be opened.\n");
        exit(1);
    }

    //Leemos el tamaño de archivo y las veces que se repite cada byte
    while(!feof(frequencyTable)){
        if(i == 0){
            fscanf(frequencyTable, "%d", fileSize);
        }
        fscanf(frequencyTable, "%d", &byte);
        fscanf(frequencyTable, "%d", &frequency);
        bytesFrequency[byte].byte = byte;
        bytesFrequency[byte].frequency = frequency;
        i++;
    }
    fclose(frequencyTable);
}
```

Handwritten annotations for `readFrequencyTable`:

- Purple bracket around the file opening and error handling, labeled $O(1)$.
- Green bracket around the `while` loop, labeled $\log n$.
- Text "por jerarquia $O(\log n)$ " with an arrow pointing to the loop.

Funcion ReadByteCode O(n)

```
unsigned char* readByteCode(int* byteFileSize){
    unsigned char c;
    int i;
    FILE* file;

    //Abrimos y verificamos que si se abrio correctamente
    file = fopen("byteCode.dat", "rb");
    if(file == NULL){
        puts("Open file Failed");
        exit(1);
    }

    //Obtenemos el tamaño del archivo .dat
    fseek(file, 0L, SEEK_END);
    (*byteFileSize) = ftell(file);
    rewind(file);

    //Reservamos memoria donde guardaremos los bytes leidos
    unsigned char* bytesRead = malloc((*byteFileSize) * sizeof(unsigned char));

    //Leemos los bytes del .dat y los guardamos en un arreglo
    for(i = 0; i < (*byteFileSize); i++){
        fread(&c, sizeof(unsigned char), 1, file);
        bytesRead[i] = c;
    }
    fclose(file);
    return bytesRead;
}
```

Diagramas de complejidad:

- Verde: $O(1)$ (para `fopen`)
- Púrpura: $O(1)$ (para `fseek`, `ftell`, `rewind`)
- Naranja: $O(1)$ (para `malloc`)
- Rojo: $O(n)$ (para el bucle `for` y `fread`)

Nota: "por jerarquia O(n)" se refiere a la complejidad total de la función.

Complejidad Programa DescompressHumman

$O(n^2)$

```
int main(int argc, char* argv[]){
    //*****
    //Variables a usar en el programa
    //Variables para el control de tiempo
    double utime0, stime0, wtime0, utime1, stime1, wtime1;
    //Arreglo para almacenar la frecuencia de bytes
    struct data* bytesFrequency = malloc(256 * sizeof(struct data));
    //Arreglo para almacenar la codificación de Huffman
    struct bits* bytesCode = malloc(256 * sizeof(struct bits));
    //Arreglo para almacenar el byte y frecuencia en cada nodo del arbol
    struct node* roots = (struct node*)malloc(256 * sizeof(struct node));
    //Arbol de Huffman ya unificado
    struct node* HuffmanTree;
    //Cadena de bytes leídos del archivo original
    unsigned char* bytesRead;
    //Cola de prioridad
    Heap* heap = CreateHeap(511);
    //Variables para almacenar los tamaños de archivos
    int fileSize = 0, byteFileSize = 0;

    puts("\nDescomprimir archivos usando el algoritmo de Huffman\n");
    //*****
    // Iniciar el conteo del tiempo para las evaluaciones de rendimiento
    //*****
    uswtime(&utime0, &stime0, &wtime0);

    //Leer la tabla de frecuencias
    readFrequencyTable(bytesFrequency, &fileSize);
    //hacer el arbol con la tabla de frecuencias
    insertTree(bytesFrequency, roots, heap);
    //unir los arboles en uno solo
    HuffmanTree = mergeTrees(heap);
    //Obtener el código de bits de cada byte
    getBits(HuffmanTree, bytesCode, 0, 0);
    //Leer el .dat generado en la compresion
    bytesRead = readByteCode(&byteFileSize);
    //sustituir la cadena de .dat por su valor de arbol
    writeFile(bytesRead, HuffmanTree, argv[1], byteFileSize, &fileSize);

    //*****
    // Evaluar los tiempos de ejecución
    //*****
    uswtime(&utime1, &stime1, &wtime1);

    printf("Tamaño de %s: %d bytes\n", argv[1], fileSize);
    printf("Tiempo real de Ejecucion: %.10e s\n\n", wtime1 - wtime0);

    return 0;
}
```

$\rightarrow O(\log n)$
 $\rightarrow O(n^2)$ Por jerarquía la complejidad es
 $\rightarrow O(n \log n)$
 $\rightarrow O(n)$
 $\rightarrow O(n)$
 $\rightarrow O(n^2)$
 $\rightarrow O(n)$