UNIVERSIDAD AUSTRAL DE CHILE

Facultad de Ciencias de la Ingeniería, Escuela de Ingeniería Civil en Informática

Técnicas de Representación y Compresión de Arreglos Ordenados.

INFO 145 - Diseño y análisis de algoritmos.

Primer Semestre, 2024.

Docente responsable
Héctor Ferrada Escobar.

Alumnos
Alann Kahler C. alann.kahler@alumnos.uach.cl
Francisco Morales A. francisco.morales02@alumnos.uach.cl
Martín Maza D. martin.maza@alumnos.uach.cl
Rudy Richter M. rudy.richter@alumnos.uach.cl

09 de julio de 2024

Índice

1. Abstract

Esta tarea aborda diferentes técnicas de representación y compresión de arreglos ordenados. El objetivo es comparar el tiempo de ejecución y espacio utilizado al aplicar la estrategia de búsqueda binaria adaptada en tres representaciones distintas: Arreglo Explícito, Gap-Coding, y Shannon-Fano.

2. Introducción

En el área de la programación, existe una variedad de estructuras que permiten almacenar una gran cantidad de datos. Uno de los problemas que pueden surgir con estas estructuras es la cantidad de memoria que pueden llegar a utilizar.

Para solucionar este problema, existen una variedad de técnicas de diseño relacionadas con los algoritmos que permiten optimizar el uso de memoria en estructuras ordenadas, aprovechando diversas características, como la posición de los elementos o la recurrencia de estos.

Algunas de las soluciones al problema del espacio se abordarán en la metodología; estas son: Arreglo Explícito, Gap-Coding y Shannon Fano. Serán codificados en pseudocódigo para luego ser implementados en C++ y analizados en términos de costos asintóticos y de rendimiento (espacio de memoria).

Posteriormente, se realizará una experimentación y comparación entre los diferentes métodos, acompañada de diversas gráficas que contrastarán los tres métodos y su forma de abordar los diferentes puntos a evaluar: tiempo de ejecución y espacio utilizado. Todas las metodologías se aplicarán a dos tipos de arreglos con valores aleatorios en orden creciente, uno con distribución normal y el otro lineal.

Con toda esta información, podremos determinar los puntos a favor y en contra de utilizar cierta estrategia de diseño y, de esa forma, decidir cuál método es más efectivo a la hora de ser implementado dado un caso específico.

3. Metodología

La sintaxis, métodos y operaciones utilizadas son implementadas en C++. *Se define como índice inicial 0 e índice final n-1 para un arreglo de largo n.*

3.1. Arreglo Explicito

3.1.1. Inicialización de los datos

Esta primera solución únicamente solicita un arreglo de enteros como entrada, el cual puede ser:

- Arreglo Lineal L ordenado de tamaño n definido por:

$$L[0] = \text{valor\_aleatorio} \,\%\, \epsilon$$

$$L[i] = L[i-1] + \text{valor\_aleatorio} \,\%\, \epsilon, \quad 1 \leq i < n$$

  Se utilizará un $\epsilon = 8$ para poder utilizar el tipo de variable short (16 bits en el dispositivo a experimentar), dar abasto a las codificiaciones y almacenamiento de diferencias en los demás métodos. Además es un tipo de variable de largo menor al int (32 bits) que es el tipo de los arreglos originales. Se previene overflow.

- Arreglo Normal N de tamaño n definido por:

$$N[i] = \text{valor\_normal(media, desviación estándar)} \,\%\, \frac{n \cdot (e-1) + 1}{2}, \quad 0 \leq i < n$$

Se utilizarán métodos de la librería *random* de C++ para generar los valores de forma aleatoria y poder generar una distribución normal de estos.

La desviación estándar será variable hasta encontrar un valor estable para los experimentos.

Se usará $media = 0$ para poder generar valores dentro del mismo rango que el arreglo lineal. Para generar el arreglo lineal con valor máximo posible, $rand() \,\%\epsilon$ generará siempre $\epsilon - 1$, este será sumado al valor anterior n veces, por lo que el mayor valor teórico será $n \cdot (\epsilon - 1)$. Entonces para generar valores desde 0 (posible inicio) del arreglo lineal y $n \cdot (\epsilon - 1)$, se usa una distribución normal de media 0, pudiendo generar valores entre $-\frac{n \cdot (\epsilon - 1)}{2}$ y $\frac{n \cdot (\epsilon - 1)}{2}$. Luego, se suma el primer valor del arreglo en valor absoluto a todo el arreglo, desplazando la distribución normal iniciando en 0 y con un máximo posible de $n \cdot (\epsilon - 1)$.

3.1.2. Metodología

Se realizará búsqueda binaria sobre el arreglo ordenado lineal y normal. Se medirá para cada caso tanto el tiempo de ejecución como el espacio utilizado para posterior análisis.

3.2. Arreglo representado con Gap-Coding

3.2.1. Inicialización de los datos

Esta segunda solucion requiere de uno de los arreglos definidos anteriormente (Lineal o Normal) para generar los arreglos GC y sample. Estos vienen definidos de la siguiente forma:

- Arreglo GC de tamaño n. Utilizando un arreglo ordenado A (lineal L o normal N) también de tamaño n, GC se define por:

$$GC[i] = A[i] - A[i-1], \quad n > i \geq 1$$

$$GC[0] = A[0]$$

  Este arreglo almacena las diferencias del valor en la posición i del arreglo A con su antecesor exceptuando el valor en la posición 0 el cual se mantiene.

- Arreglo sample de tamaño m. Utilizando un arreglo ordenado A (lineal L o normal N) de tamaño n, sample se define por:

$$sample[i] = A[i \cdot b], \quad b = \frac{n}{m}, \quad 0 \leq i < m$$

  El valor b viene representando la cantidad de elementos que omite entre el ultimo elemento que se escogió hasta el siguiente a escoger en el arreglo. Esto con el fin de tomar una muestra menor del arreglo A.

3.2.2. Metodología

Se realizará búsqueda binaria sobre el arreglo ordenado sample. A raíz de esto, si el valor se encuentra en el arreglo se detendrá la ejecución, de otro modo, se recuperará el índice y valor de la última iteración de la búsqueda binaria. Existen casos en los que la búsqueda binaria retornará la posición de un valor mayor al

elemento a buscar, se identificará esto y se corregirá[1]. Puesto que los elementos en la posición i de sample son los que se encuentran en $i \cdot b$ del arreglo original, el índice obtenido se multiplicará por b para tener la ubicación del elemento que comenzará a reconstruirse. En este punto, se iterará sobre el arreglo GC aumentando el valor a reconstruir hasta que sea igual que el elemento a buscar (el elemento se encuentra), sea mayor que el elemento a buscar o intente iterar en la posición n (el elemento no se encuentra). Se hará esto tanto para el arreglo lineal como el normal. Se medirá para cada caso tanto el tiempo de ejecución como el espacio utilizado para posterior análisis.

3.3. Arreglo representado con Shannon Fano

3.3.1. Codificación de Shannon-Fano

La explicación del Algoritmo de Shannon-Fano sigue los pasos descritos por Ferrada, 2024.

1. Los símbolos se ordenan descendentemente por su probabilidad.
2. Los símbolos se dividen en dos subconjuntos, los cuales suman igual probabilidad conjunta (o la más cercana posible).
3. A los códigos del primer subconjunto se le añade un 1, y un 0 a los del segundo subconjunto.
4. Prosiga recursivamente en cada subconjunto (ir al paso 2) hasta que no queden más símbolos.

La forma en la que se implementa este algoritmo viene siendo:

- Se genera un arreglo de objetos, cada uno con los campos: símbolo, probabilidad, código. Cada objeto será por un elemento único en un arreglo. Luego se ordenan descendentemente según su campo de probabilidad.
- El arreglo anterior se separa en dos partes. El divisor será el índice que separa los elementos con la probabilidad más baja de los demás. Los objetos de cada parte tendrán un prefijo diferente en su codificación, los que tengan la probabilidad menor se considerarán outliers y sus códigos iniciarán con 1, el resto se considerará común e inciarán con 0.

[1]Por ejemplo, para sample = [3, 5, 7] buscando el valor 6, la búsqueda binaria arrojará la posición 2 (posición del 7) ya que $7 > 6$ no se podrá reconstruir hasta encontrar el 6 ya que el arreglo original está ordenado. La corrección retrocederá una posición (posición 1 del 5), permitiendo poder reconstruir el valor hasta encontrar el 6 (si es que se encuentra).

- Ahora, se ejecutará el método Shannon-Fano para cada partición (comunes y outliers) a partir del siguiente bit de codificación. Se obtendrá la división de la partición, los cuales suman igual probabilidad conjunta (o la más cercana posible) y se añadirá un 1 al primer subconjunto y un 0 al segundo.

Se definirá un diccionario de decodificación tanto para los elementos comunes (DSFC) como para los outliers (DSFO).

Esto con el fin que los outliers no empeoren la codificación de los elementos más comunes y evitar overflow.

3.3.2. Inicialización de los datos

Esta tercera solución necesita del arreglo GC para crear el arreglo GCSF. Además hace uso de sample para la búsqueda.

- Diccionarios DSFC y DSFO. A partir del arreglo GC se obtendrán las codificaciones de Shannon-Fano para los elementos comunes (DSFC) en este arreglo y los outliers (DSFO). Tendrán la forma de código: símbolo. Aunque en la codificación del arreglo GC se invertirá el orden para que a partir del símbolo se obtenga el código.

- Arreglo sample de largo m. Se mantiene el sample utilizado en el método de GAP-Coding.

- Arreglo GCSF de largo n. Utilizando el arreglo GC también de tamaño n, los diccionarios DSFC (codificación de valores comunes) y DSFO (codificación de valores outliers) con la codificación de Shannon-Fano para cada elemento en GC, GCSF define por:

$$GCSF[i] = \text{código en DSFC para GC[i]}, \quad \text{si GC[i]} \in DSFC$$

$$GCSF[i] = \text{código en DSFO para GC[i]}, \quad \text{si GC[i]} \in DSFO$$

$$0 \leq i < n$$

3.3.3. Metodología

Al igual que el método de GAP-coding, se hará una búsqueda binaria sobre sample. Si encuentra el valor termina la ejecución. De otro modo se obtendrá el índice y valor de la última iteración. De ser el caso, se corregirá el índice. Este índice

será multiplicado por b para comenzar a reconstruir el valor desde la posición de este en GCSF. Iniciará la iteración sobre este arreglo y por cada elemento se decodificará haciendo uso de los diccionarios DSFC y DFCO para sumar el valor real al valor que se reconstruye. Se realizará esto hasta que sea igual que el elemento a buscar (el elemento se encuentra), sea mayor que el elemento a buscar o intente iterar en la posición n (el elemento no se encuentra). Se hará esto tanto para el arreglo lineal como el normal. Se medirá para cada caso tanto el tiempo de ejecución como el espacio utilizado para posterior análisis.

3.4. Hipótesis inicial

3.4.1. Rendimiento (Operaciones)

- Arreglo Explícito: Se espera que este método sea el más rápido de los tres, esto debido a que su costo asintótico es el de la búsqueda binaria sobre un arreglo ordenado $O(\log_2 n)$ realizando $\log_2 n$ operaciones en el peor caso. Comparado a los demás métodos en el peor caso, es el mejor.

- GAP-Coding: Si este método es implementado con largos constantes (ej: int de 32 bits) es esperable que sea menos eficiente en tiempo de ejecución que el Arreglo Explícito. Esto debido a que realizará $\log_2 m$ comparaciones en el arreglo sample (de largo m) con la búsqueda binaria, se le suma a las b comparaciones y b sumas que realizará en el arreglo GC si es que el elemento no se encuentra (peor caso). Si se tiene en cuenta que $b = \frac{n}{m}$ la cantidad de operaciones que ejecutará en el peor caso viene dado por:

  $$\log_2 m + 2 \cdot \frac{n}{m}$$

  Por lo que su costo asintótico vendría siendo O(n).
  Sin embargo, a pesar de esto, si fuese implementado usando largos variables (el mínimo de bits necesarios para representar un valor) podría llegar a marcar una diferencia suficiente para ser más efectivo que el Arreglo explícito al ejecutar operaciones sobre valores representados con menos bits ya que GC almacena diferencias entre los valores originales. Por ende, a pesar que deba ejecutar la misma cantidad de operaciones en el peor caso, estas operaciones serán más eficientes. En la presente tarea **solo se analizarán largos constantes** (aunque menores al de los arreglos originales).

- Shannon-Fano: Se espera que su rendimiento sea menor al de GAP-Coding. La razón de esto es que, al ser una variación de este último, tiene que ejecutar la misma cantidad de operacions, sin embargo, se agrega la decodificación del valor a sumar:

  $$\log_2 m + 3 \cdot \frac{n}{m}$$

  A pesar de que la decodificación implementada es constante al tener disponible el diccionario DSF esta operación provocará que su rendimiento sea, aunque de forma leve, peor que el método de GAP-Coding. Al seguir teniendo que acceder a un valor del mismo largo que en el método GAP-coding y realizar las operaciones de suma y comparación sobre este, la decodificación termina por

agregar un paso extra.
Un caso parecido sucede si este método es implementado con largo variable, su rendimiento será levemente peor que GAP-coding con largo variable debido a la operación ejecutada en el diccionario.
Ya que es una variante de GAP-coding que solo cambia la representación de valores del arreglo GC, realizando también en el peor caso $\log_2 m + 2 \cdot \frac{n}{m}$ operaciones, su costo asintótico es O(n).

Resumiendo:

Método	**Rendimiento**
GAP-Coding Variable	Mejor
Shannon-Fano Variable	
Arreglo Explícito	
GAP-Coding Constante	
Shannon-Fano Constante	Peor

3.4.2. Memoria (Espacio)

- Se espera que el espacio en memoria utilizado en el método de Arreglo Explícito, teniendo en cuenta que tanto el arreglo lineal como el normal serán de tamaño n almacenando variables de tipo int (32 bits), sea de:

$$P(n) = n \cdot 32 \text{ bits}$$

- La implementación del GAP-Coding, viendo que se recorrerá una vez el arreglo A (lineal o normal) para generar el arreglo GC, se puede prescindir de este. Tenemos el arreglo GC de tamaño n que almacena variables de largo constante (short de 16 bits), a esto se le suma el arreglo sample de tamaño m que almacena variables de largo constante (int de 32 bits). Tenemos para este caso que se utilizarán:

$$P(n, m) = n \cdot 16 + m \cdot 32 \text{ bits}$$

Para la implementación de largo variable tenemos que por cada valor a perteneciente al arreglo GC usarán $\lceil \log_2 a \rceil$ bits. A esto también se le suma el peso del arreglo sample que guardará m valores de tipo int (32 bits).

$$P(GC, m) = \sum_{a \in GC} \lceil \log_2 a \rceil + m \cdot 32 \text{ bits}$$

- El método de Shannon-Fano hará uso del arreglo GC para generar GCSF (short) de tamaño n, por lo que se puede prescindir del primer arreglo. A este se le suma que seguirá utilizando el arreglo sample (int) de tamaño m. Además se hará uso de los diccionarios DSFC y DSFO que guardarán los a elementos distintos del arreglo GC, por cada elemento se hará uso de del valor (short) y su codificación (short). Utilizando un espacio de:

$$P(n, m) = n \cdot 16 + a \cdot 2 \cdot 16 + m \cdot 32 \text{ bits}$$

La implementación de Shannon-Fano con largos variables hará uso de:

$$P(GCSF, DSFC, DSFO, m) =$$

$$\sum_{a \in GCSF} \lceil \log_2 a \rceil +$$

$$\sum_{a \in DSFC} (\log_2 \text{a.llave} + \log_2 \text{a.código}) +$$

$$\sum_{a \in DSFO} (\log_2 \text{a.llave} + \log_2 \text{a.código}) +$$

$$m \cdot 32 \text{ bits}$$

A partir de las fórmulas se teoriza que Arreglo Explícito usará más espacio, seguido de Shannon-Fano para finalmente ser GAP-Coding.

Resumiendo:

Método	**Memoria**
Shannon-Fano Variable	Mejor
GAP-Coding Variable	
GAP-Coding Constante	
Shannon-Fano Constante	
Arreglo Explícito	Peor

3.5. Pseudocódigos

3.5.1. Arreglo Explícito

```
// Función para generar un arreglo ordenado de tamaño n con
// aumento máximo e
generar_arreglo_lineal(n, e) {
    // Complejidad asintotica O(n)
    Sea L[0 .. n-1] un arreglo de enteros de tamaño n
    L[0] = random() % e
    for i = 1 to n-1 do
        L[i] = L[i-1] + random() % e
    return L
}

// Función para generar un arreglo dinámico con distribución
// normal de tamaño n, aumento máximo e, media m y desviación
// estándar s.
generar_arreglo_normal(n, m, s) {
    // Complejidad asintotica O(n)
    Sea N[0 .. n-1] un arreglo de enteros de tamaño n

    // Para que tanto el arreglo Lineal como el Normal
    // tengan valores dentro del mismo intervalo se propone:
    // En el caso que la función rand()% e siempre genere el valor
    // máximo posible, para generar el arreglo lineal
    // es decir (e-1) este será sumado al valor anterior n
    // veces, por lo que el mayor valor teórico será n*(e-1).
    // Entonces para generar valores desde 0 (posible inicio)
    // del arreglo lineal y n*(e-1), se usa una distribución
    // normal de media 0, puediendo generar un máximo n*(e-1)/2
    // positivo y n*(e-1)/2 negativo. Luego, se suma el primer
    // valor del arreglo en valor absoluto a todo el arreglo,
    // desplazando la distribución normal iniciando en 0 y con
    // un máximo posible de n*(e-1).

    for i = 0 to n-1 do
        N[i] = valor_normal(m, s) % ((n * (e-1) + 1)/2);

    N.sort()
```

```
    if (N[0] < 0)
        min = valor_absoluto(N[0])
        for i = 0 to n-1 do
            N[i] += min

    return N
}

// Implementación de búsqueda binaria no recursiva sobre un
// arreglo ordenado A de tamaño n y valor a buscar val.
busqueda_binaria(A, n, val) {
// Complejidad asintotica O(log(n))
    izq = 0
    der = n-1

    while izq <= der do
        med =  izq + (der - izq) / 2

        if A[med] == val
            return med // Se encontró el valor
        if A[med] < val
            izq = med + 1
        else
            der = med - 1

    return -1 // No se encontró el valor
}
```

3.5.2. GAP-Coding

```
// Generar un arreglo gap_coding a partir de un Arreglo A de
// tamaño n. Cada valor del arreglo generado tendrá el valor de
// A[i] - A[i-1] con i > 0 el primer valor del arreglo será
// igual a A[0].
generar_arreglo_gap_coding(X, n) {
    // Complejidad asintótica O(n)
    Sea GC[0 .. n-1] un arreglo de enteros de largo n
    for i = n-1 to 1 con paso -1
        GC[i] = X[i] - X[i-1]
    GC[0] = X[0]
    return GC
}

// Generar un arreglo sample de tamaño m a partir de un Arreglo
// A de tamaño n. Cada valor en la posición i del sample será
// A[i * (n/m)].
generar_sample(X, n, m) {
    // Complejidad asintótica O(m)
    Sea sample[0 .. m-1] un arreglo de enteros de largo m
    b = n / m
    for i = 0 to m-1
        sample[i] = X[i * b]
    return sample
}

// Realiza la busqueda binaria sobre un arreglo sample y
// complementa con un arreglo gap_coding para buscar val
busqueda_binaria_gap_coding(GC, sample, n, m, val) {
    // Complejidad asintótica O(log(m)) + O(n/m) -> O(n)
    izq = 0
    der = m-1
    b = n / m

    // Se ejecuta la búsqueda binaria en el sample
    while izq <= der do
        med = der - (der - izq) / 2
```

```
    if sample[med] == val
        return med * b
    if sample[med] < val
        izq = med + 1
    else
        der = med - 1


// Si no se encuentra el valor en el sample, continúa

// Si med es 0, significa que el valor a buscar es menor
// al primer valor del arreglo. Al estar este ordenado,
// significa que el valor no se encuentra.
if med == 0 return -1;

// Si el valor a buscar es menor que el que se encuentra en
// la posición med, entonces se corrige retrocediendo una
// posición.
//      0  1  2
// Ej: [1, 3, 5], buscando el 4 retorna la posición 2
// 4 < 5 entonces se continúa con la posición 1
if val < sample[med]
    med -= 1

// Se obtiene el valor en la posición med para comenzar
// a reconstruir el valor
num = sample[med]

// Obtenemos la posición en el arreglo gap_coding en la que
// se encontraría el valor a reconstruir
med = med * b

// Comenzamos a reconstruir el valor.
// Retorna -1 si se pasa del tamaño del arreglo med >= n o
// si el valor reconstruido pasa del valor a buscar. Valor
// no se encuentra.
while med < n and num <= val
    // Si se encuentra se retorna su posición
```

```
        if num == val
            return med

        // Se agrega la siguiente diferencia al valor que se
        // reconstruye
        num = num + GC[med]
        med = med + 1

    return -1
}
```

3.5.3. Shannon-Fano

```
// A partir de un arreglo X de tamaño n identifica los valores
// únicos contenidos, su probabilidad y asigna un campo de
// código que inicia en 0b00000000. Luego, estos elementos son
// ordenados por probabilidad.
conseguir_valores(X, n) {
    // Complejidad asintótica O(n) + O(n log(n)) -> O(n log(n))
    Sea C un diccionario que almacena valores en la forma
    {simbolo, probabilidad, codigo}

    for i = 0 to n-1
        if C.find(X[i])
            C[X[i]].probabilidad += 1/n
        else
            C[X[i]] = {X[i], 1/n, 0b00000000}

    quicksort(C, 0, n-1)

    return C
}

// Intercambia de posición el elemento a con el elemento b
swap(a, b) {
    // Complejidad asintótica O(c)
    temp = a
    a = b
    b = temp
}


particion_probabilidad(X, low, high) {
    // Complejidad asintótica O(n)
    p = X[high].probabilidad
    i = low - 1

    for j = low to high-1
        if X[j].probabilidad > p
            i += 1
```

```
            swap(X[i], X[j]) // O(c)

    swap(X[i + 1], X[high]) // O(c)
    return i + 1
}

quicksort(X, low, high) {
    // Complejidad asintótica O(n log(n))
    if (low < high)
        pi = particion_probabilidad(X, low, high)

    quicksort(X, low, pi - 1)
    quicksort(X, pi + 1, high)
}

codificarSF(C, a, b, probTotal, pot) {
    // Complejidad asintótica O(n log(n))
    // Precondition: C es un diccionario donde sus datos son
    // de la forma: {simbolo, probabilidad, codigo}
    // ordenados de mayor a menor según sus valores de
    // probabilidad. a es el índice de inicio del conjunto
    // y b es el índice final del conjunto. probTotal es la
    // suma de todas las probabilidades de los objetos en el
    // conjunto. pot es la posición (de derecha a izquierda)
    // que se está alterando en el código de los datos.

    // Postcondition: Cada elemento tendrá en su campo
    // de código su codificación correspondiente
    // utilizando el método de Shannon-Fano.

    if a >= b return;

    p, probActual = divisorProbabilidad(C, a, b, probTotal);

    for i = a to p
        C[i].codigo += pow(2,pot)

    if (b-a > 1) then
        if (p-a > 0) then Shannon-Fano(C, a, p, probActual, pot+1);
```

```
        if (b-(p+1) > 0) then Shannon-Fano(C, p+1, b, probTotal -
        probActual, pot+1);
}

divisorProbabilidad(C, a, b, probTotal) {
    // Complejidad asintótica O(n)
    // Precondition: C es un conjunto con objetos
    // (cada uno con los campos: probabilidad y codigo)
    // ordenados de mayor a menor según sus valores
    // de probabilidad. a es el índice de inicio del conjunto
    // y b es el índice final del conjunto. probTotal es la
    // suma de todas las probabilidades de los objetos en el
    // conjunto.

    // Postcondition: Si el conjunto tiene dos elementos,
    // se retorna el índice de inicio 'a'. De otro modo se
    // retorna el índice p que divide el conjunto en dos
    // subconjuntos con la menor diferencia entre la suma de
    // las probabilidades de sus objetos.

    if (b-a == 1) then return (a, C[a].probabilidad);

    probActual = C[a].probabilidad
    probMedia = probTotal / 2
    diferencia = |probMedia - probActual|
    indice = a

    for i = a+1 to b:
        if |probActual + C[i].probabilidad - probMedia| <= diferencia then
            probActual += C[i].probabilidad
            diferencia = |probActual - probMedia|
        else
            indice = i-1
            break;

    return (indice, probActual)
}

// Genera un Diccionario Shannon Fano (DSF) que toma
```

```
// los elementos desde el índice a hasta n-1 de C para
// generar un diccionario para decodificarlo.
generar_DSF(C, a, n) {
    // Complejidad asintótica O(n)
    // Sea DSF un diccionario con llaves de {tipo1} y como
    // valores de {tipo2}

    for i = a to n-1
        DSF[C[i].codigo] = C[i].simbolo

    return DSF
}

// Codifica el arreglo Gap Coding GC de tamaño n de acuerdo
// a los Diccionarios Shannon Fano para valores Comunes
// (pares) y Outliers (impares)
generar_GCSF(GC, DSFC, DSFO, n) {
    // Complejidad asintótica O(n)
    // Se define el arreglo GCSF de 'tipo' ya que
    // durante la experimentación el tipo de variable
    // que se van a almacenar variarán, siendo la variación
    // en el largo del tipo de dato (32, 16, 8, 4 ... bits)

    Sea GCSF[0 .. n-1] un arreglo de largo n de {tipo}

    for i = 0 to n-1
        if GC[i] es común
            GCSF[i] = Llave de DSFC para el valor GC[i]
        else
            GCSF[i] = Llave de DSFO para el valor GC[i]

    return GCSF
}

// A partir de los Diccionarios Shannon Fano para valores
// Comunes (pares) y Outliers (impares) retorna la
// decodificación del código val
decodificar_SF(DSFC, DSFO, val) {
    // Complejidad asintótica O(c)
```

```
        if (val % 2 == 0) return DSFC[val];
        return DSFO[val]
}
```

3.6. Métodos de medición

3.6.1. Medición Temporal

Para la medicion de tiempo se importara una libreria llamada *Chrono* la cual permite obtener el tiempo actual y almacenarlo en una variable.

Antes de iniciar las búsquedas para cada método se obtendrá el tiempo de inicio. Tras concluir con todas las búsquedas se obtendra el tiempo de finalización. Luego se restará este último con el primero y se obtendrá el tiempo total de ejecución de las búsquedas.

3.6.2. Medición de Memoria

Para la medición de la memoria utilizada se tiene la función sizeof() de C++. Tras las búsquedas, se usará esta función sobre las variables explícitas de los arreglos (o diccionarios) sumado al uso de esta función sobre los primeros valores de las estructuras multiplicado por el largo del arreglo (o diccionario). Puesto que estas estructuras son de largo constante, encontrar el tamaño del primer valor y multiplicarlo por el largo bastará.

3.7. Análisis de resultados

Se realizarán 5 ejecuciones para cada combinación de parámetros de cada categoría de experimento. Se calculará el promedio de las ejecuciones y se graficarán utilizando el lenguaje de programación python y su librería matplotlib. Además previo a estas ejecuciones, se buscará el valor de desviación estándar a utilizar, se verficará que los arreglos sean generados correctamente y se identificará el tamaño de sample que permitan un análisis más justo y equilibrado. Los resultados de la experimentación se verán en el siguiente punto.

4. Experimentación

4.1. Desviación estándar

Para visualizar cómo afecta la desviación estándar en el rendimiento de los métodos, se experimentó con valores para este desde 10^0 hasta 10^6. Solo se visualizó el comportamiento para el Arreglo Normal puesto que esta no afecta al arreglo Lineal.

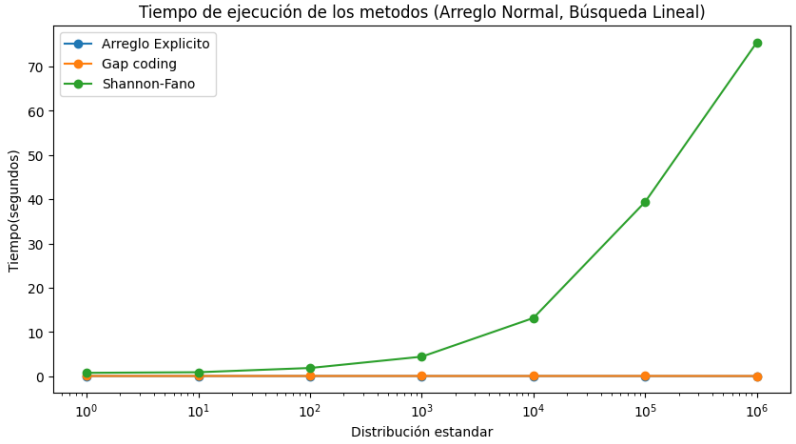


Figura 1: Tiempo de ejecución de los métodos (Arreglo Normal, Búsqueda Lineal)

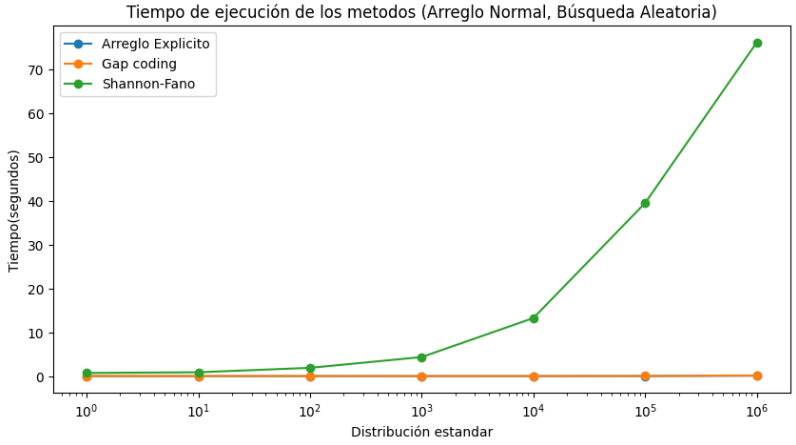


Figura 2: Tiempo de ejecución de los métodos (Arreglo Normal, Búsqueda Aleatoria)

Se aprecia un crecimiento exponencial en el tiempo de ejecución en el método de Shannon-Fano a medida que la distribución estándar aumenta.

Para no agravar el rendimiento de este método, se decide utilizar el valor de 100 para la desviación estándar de los siguientes experimentos.

4.2. Generación de arreglos

En primera instancia se comprobó que los arreglos son generados correctamente viendo sus distribuciones a través de histogramas. Se obtuvieron los siguientes:

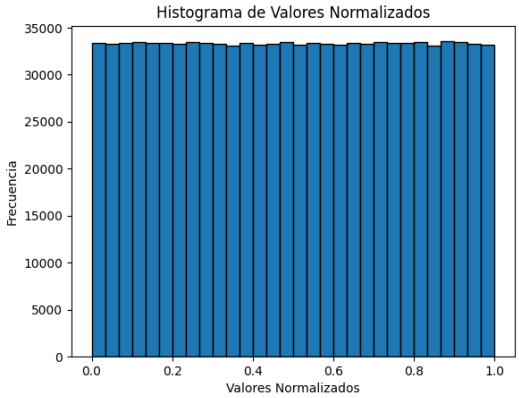


Figura 3: Histograma de distribución para Arreglo Lineal

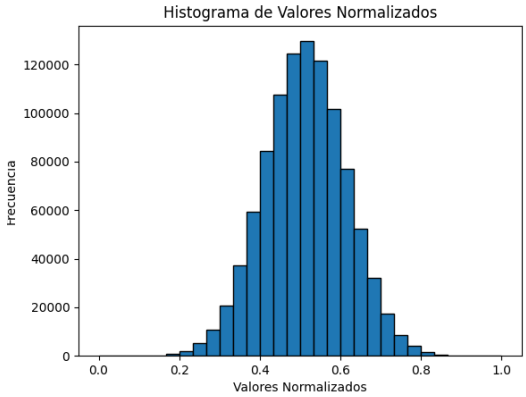


Figura 4: Histograma de distribución para Arreglo Normal

Se aprecia que los valores de los arreglos son generados correctamente.

4.3. Tamaño de sample

Para ver como el tamaño del sample afecta en el rendimiento de los métodos se experimento con distintas proporciones (f) respecto al tamaño del arreglo original (n). El tamaño del arreglo sample se define por $\frac{n}{f}$. Con un tamaño n = 10^6 se obtuvo:

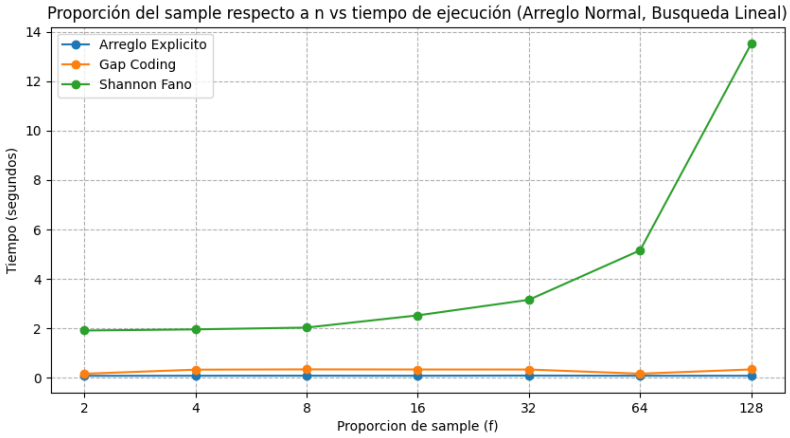


Figura 5: Proporción del sample respecto a n vs tiempo de ejecución (Arreglo Normal, Búsqueda Lineal

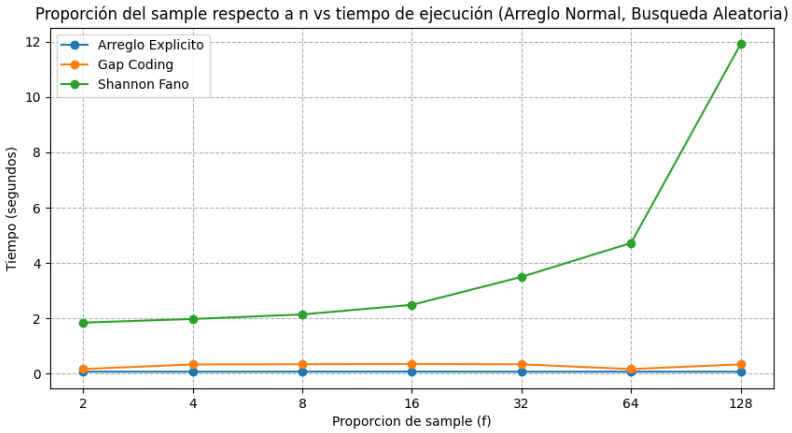


Figura 6: Proporción del sample respecto a n vs tiempo de ejecución (Arreglo Normal, Búsqueda Aleatoria)

Se puede ver que a medida que el tamaño de sample disminuye el tiempo de ejecución aumenta para los métodos de Shannon-Fano y GAP-Coding, aunque el

aumento del primero es mucho mayor que el leve aumento del último.
Se comenta el evento que tanto para la búsqueda lineal como la aleatoria, el rendimiento de GAP-Coding se acercó al de Arreglo Explícito para f = 64. Además, el rendimiento de este último se mantuvo constante puesto que no hace uso del arreglo sample.

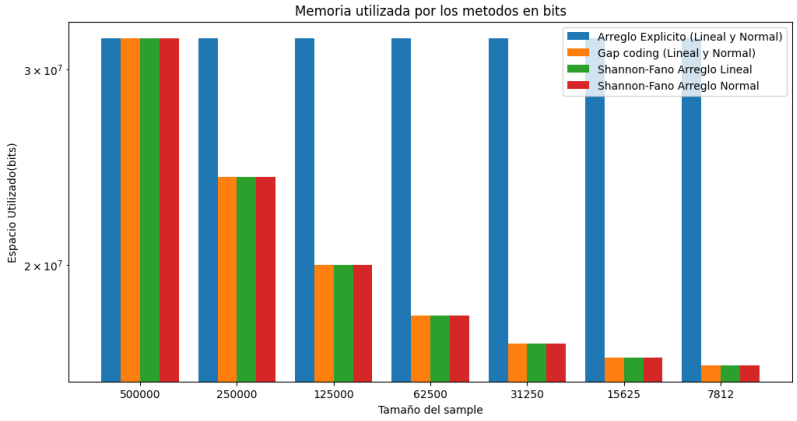


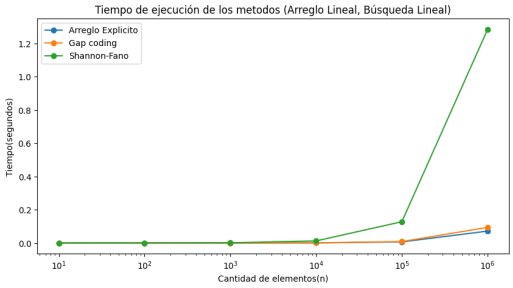
Figura 7: Memoria utilizada para los métodos en bits.

Se ve que la primera gran diferencia ocurre cuando la proporción f = 4 siendo la mayor disminución respecto al valor anterior. Las siguientes disminuciones tienden a ser la mitad de la diferencia anterior, mostrando un comportamiento logaritmico.

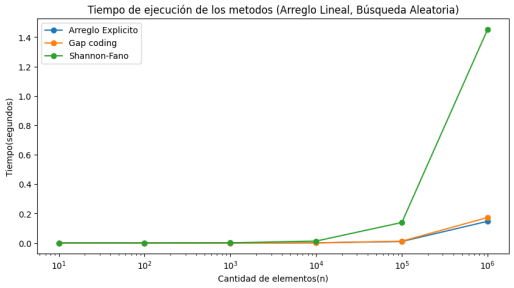
Finalmente se continuó la experimentación usando uno de los tamaños con mejor rendimiento siendo a la vez una reducción notable del arreglo original f = 4.

4.4. Rendimiento (Operaciones)

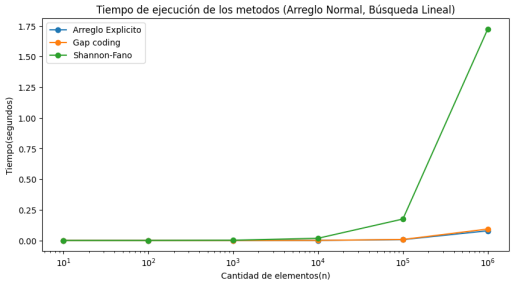
Variando el tamaño de los arreglos originales desde 10^1 hasta 10^6 se ejecutó la búsqueda binaria implementada para cada método realizando búsquedas de valores lineales (0 hasta n) y n valores aleatorios. Se obtuvieron los siguientes resultados:



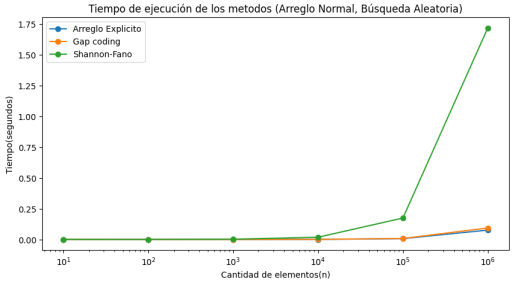
(a) Tiempo de ejecución de los métodos (Arreglo Lineal, Búsqueda Lineal)



(b) Tiempo de ejecución de los métodos (Arreglo Lineal, Búsqueda Aleatoria)



(c) Tiempo de ejecución de los métodos (Arreglo Normal, Búsqueda Lineal)



(d) Tiempo de ejecución de los métodos (Arreglo Normal, Búsqueda Aleatoria)

Figura 8: Comparación de los tiempos de ejecución de los diferentes métodos

Se puede apreciar que a partir de 10^5 en las cuatro figuras, el método de Shannon-Fano aumenta drásticamente respecto a los otros métodos. Además, Gap-coding es levemente más tardío que Arreglo explícito.

4.5. Memoria (Espacio)

Variando el tamaño de los arreglos originales desde 10^1 hasta 10^6 se calculó el tamaño total utilizado por cada uno y se graficó obteniendo lo siguiente:

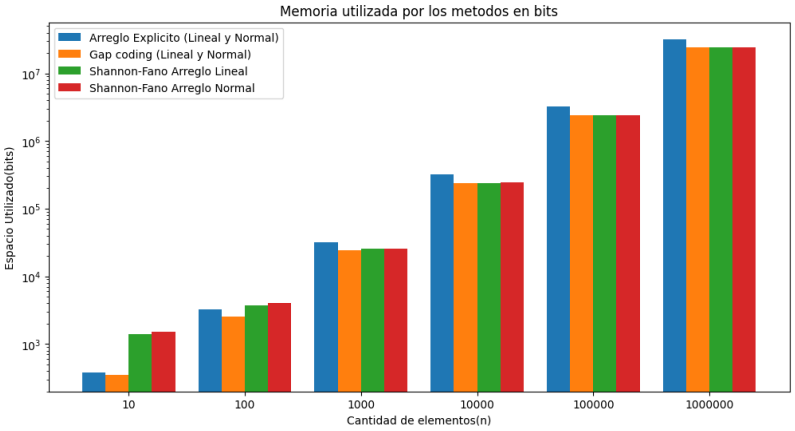


Figura 9: Memoria utilizada por los métodos en bits

Se puede ver que el método de arreglo explícito es el que más memoria utiliza entre todos los métodos. También se identifica que GAP-Coding usa levemente menos espacio que Shannon-Fano. Además, en el caso del arreglo normal, Shannon-Fano tiende a usar más espacio que para el arreglo lineal, esto se debe a que las diferencias entre valores para este último es a lo más ϵ, sin embargo, para el arreglo normal las diferencias pueden variar más allá de esa cantidad, por lo que los valores codificados son mayores, teniendo diccionarios más grandes y por consiguiente, se usa más espacio.

5. Conclusión

5.1. Rendimiento (Operaciones)

Respecto a las hipotesís iniciales, estas están acorde a los resultados. Aunque no se esperaba un aumento tan drástico de Shannon-Fano, se cumple que el Arreglo Explícito tiene el mejor tiempo, Shannon-Fano tiene el peor y en segundo lugar está GAP-Coding.
Esto se evidencia en la Figura 8. Además utilizando las fórmulas de operaciones presentadas anteriormente y los valores $n = 10^6$, $m = 10^2$ tenemos que:

$$\text{Arreglo Explícito: } \log_2 n = \log_2 10^6 = 19{,}931\ldots <$$

$$\text{GAP-Coding: } \log_2 m + \frac{2n}{m} = \log_2 100 + \frac{2000000}{100} = 20006{,}643\ldots <$$

$$\text{Shannon-Fano: } \log_2 m + \frac{3n}{m} = \log_2 100 + \frac{3000000}{100} = 30006{,}643\ldots$$

Por lo tanto, las hipotesis iniciales son correctas.

5.2. Memoria (Espacio)

El Arreglo Lineal siempre mantuvo una cantidad de espacio utilizada mayor a la del resto de métodos, esto es acorde a la hipótesis inicial. Además, debido a los largos constantes y peso de la variable que contiene los diccionario de Shannon-Fano, este método tiene un mayor peso que GAP-Coding.
Todo esto se puede evidenciar en la figura 9. Además si utilizamos las fórmulas entregadas anteriormente para los pesos de los métodos y los valores $n = 10^3$, $m = 250$, $a = 8$ (buen caso para Shannon-Fano) tenemos que:

$$\text{Arreglo Explícito: } n \cdot 32 = 32000 >$$

$$\text{Shannon-Fano: } n \cdot 16 + a \cdot 2 \cdot 16 + m \cdot 32 = 16000 + 256 + 8000 = 24256 >$$

$$\text{GAP-Coding: } n \cdot 16 + m \cdot 32 = 16000 + 8000 = 24000$$

Por lo tanto, las hipotesis iniciales son correctas.

5.3. Resumen

Tenemos que si lo que se busca es velocidad de ejecución si importar el espacio total utilizado, la búsqueda binaria sobre un arreglo explícito es la opción más viable.

Teniendo en cuenta el análisis con los largos constantes:

Si se busca optimizar el espacio, aún pudiendo tener un posible buen rendimiento (dependiendo de los parámetros), GAP-Coding es una opción que puede llegar a ser un punto medio entre velocidad y espacio.

No se recomienda Shannon-Fano para largos constantes debido a que empeora mucho en rendimiento, además que no es mucho mejor que GAP-Coding en cuanto a espacio.

A pesar de esto, se teoriza que si Shannon-Fano fuese implementado con largos variables, es decir, que cada valor utilice solo la mínima cantidad de bits necesaria para ser representado, muy probablemente sería una opción ganadora en cuanto a opimización de espacio. Este experimento quedará por verse ...

Referencias

Ferrada, H. (2024). INFO145 - Diseño y Análisis de Algoritmos 8 - Entropía y Códigos de Huffman [Algoritmo de Shannon-Fano].