Facultad de Ciencias
de la Ingeniería

Técnicas de Representación y Compresión de Arreglos Ordenados

Integrantes:
Martín Alvarado
Isaías Cabrera
Osvaldo Casas-Cordero
Andrés Mardones
Profesor:
Héctor Ferrada
Curso:
Diseño y Análisis de Algoritmos

Valdivia, 09 de julio de 2024

Resumen

Es fundamental en la programación el uso de arreglos ordenados para diversos contextos, principalmente debido a la gran optimización que ofrece a la hora de buscar datos en ellos. Es por esto que resulta importante identificar técnicas de representación de arreglos y valorarlas con el propósito de encontrar la forma que entregue mejores resultados de rendimiento en la búsqueda. El objetivo principal de este proyecto es encontrar entre 3 representaciones de un arreglo la mejor estructura en la cual realizar búsqueda binaria, mediante análisis teórico y empírico. Las 3 estructuras a investigar son: la representación explícita, Gap-Coding , y gaps comprimidos con Huffman.

En la primera solución (arreglo explícito) se define un arreglo común con “n” elementos donde se realiza la búsqueda binaria. Para la solución de Gap-Coding se hará la búsqueda binaria sobre una pequeña muestra del arreglo original, para luego hacer una búsqueda secuencial en el intervalo restante. La Solución 3 (Gaps comprimidos con Huffman) es creada a partir de la solución anterior, pero con la diferencia que el arreglo de gaps que utilizamos está codificado con la representación de Huffman, usando tipos de datos más livianos con respecto a los elementos del arreglo original.

Para hacer nuestra experimentación usaremos 2 arreglos distintos, uno con distribución normal y otro lineal, repetiremos la ejecución para distintos tamaños de entrada.

Tras los experimentos, podemos ver que la solución con codificación de Huffman es la más óptima en términos de memoria pero tiene los peores tiempos de búsqueda. En cambio la solución 2 es la que más memoria ocupa pero presenta la mejor velocidad. La solución con representación explícita generalmente es la más equilibrada, presentando buenos tiempos de búsqueda y buenos valores en memoria.

Introducción

En el presente informe se investigarán 3 maneras de representación de arreglos, se estimarán sus costos en tiempo y espacio en memoria, y finalmente, mediante métodos para medir el rendimiento de cada una, se evaluará cuál es la mejor, validando o refutando las hipótesis iniciales.

Las maneras de representar arreglos que serán tratadas en esta investigación son:

- Arreglo explícito: se inicializa un arreglo de tamaño n, en el que se aplica la búsqueda binaria común para encontrar el valor buscado.

- Arreglo representado con Gap-Coding: se inicializan 2 arreglos extra. El primero se denomina *gapCodingArray* el cual se construye asignando en sus celdas las diferencias entre cada elemento consecutivo del arreglo original. Y el segundo lleva por nombre *sample*, el cual representa una muestra del arreglo original (tamaño "m"). El objetivo de esta solución es aplicar una búsqueda binaria en el arreglo *sample*, para encontrar el intervalo del arreglo original donde se encuentra el valor buscado "x" (intervalo de tamaño "b"). Finalmente, se recorre este intervalo en el arreglo *gapCodingArray* reconstruyendo linealmente los elementos y así encontrar "x".

- Representación con Shannon-Fano o Huffman: aquí se codifica el arreglo *gapCodingArray* usado en la solución 2, con el fin de usar un tipo de datos más ligero. El procedimiento de esta solución es idéntico al anterior, con la única diferencia de que es necesario decodificar los valores del gapCodingArray.

Más adelante se presentan los siguientes apartados:

a) Metodología: Aquí se explican la inicialización de los datos, hipótesis iniciales, costos, pseudocódigos, representaciones y métodos para medir el rendimiento en cada una de las soluciones.
b) Experimentación: En este apartado se exponen los resultados obtenidos en las mediciones correspondientes a cada una de las soluciones.
c) Conclusión: Finalmente se resumen los resultados obtenidos y se responden las hipótesis planteadas en la Metodología.

Metodología

Hipótesis inicial

Para poder generar hipótesis acerca de los tiempos que demorarán las soluciones y así ser capaz de obtener la mejor, es necesario analizar cada una identificando sus ventajas y defectos.

1) Arreglo Explícito: este caso destaca en memoria, ya que sin duda es la que menos espacio requiere, pues solo necesita un arreglo de *n* enteros. En términos de costos no debería presentar malos resultados, debido a que en esta solución se aplica la búsqueda binaria a todo el arreglo, lo cual no demora mucho tiempo en el peor caso. Con respecto a los arreglos con distribución uniforme y normal, se prevé que no habrá diferencias en los costos. Esto debido a que el tiempo de la búsqueda binaria no depende de cómo están distribuidos los datos, pues en el peor caso será el mismo.

2) Gap-Coding: esta representación sacrifica memoria gracias a dos arreglos adicionales. El primero de ellos, GapCoding, contiene *n* enteros, y el segundo, sample, presenta *m* enteros. Aquí es sumamente importante la elección de los valores *m* y *b*, debido a que de ellos depende el rendimiento de esta solución. Si se escoge un *m* pequeño, por ejemplo, m = $log_2(n)$, se obtiene un beneficio en memoria, puesto que se creará un arreglo de sample muy pequeño. Pero, si se elige b = n / m, para un n muy grande, b será muy grande y por tanto la búsqueda lineal será extremadamente costosa. Por otro lado, si el m elegido es grande, por ejemplo: n / 4, la memoria se verá afectada, gracias al tamaño del arreglo de sample, pero al calcular b = n / m = 4, b será pequeño y por tanto la búsqueda lineal no será costosa (además, teniendo en cuenta que la búsqueda binaria en el arreglo de sample será ínfimamente mejor que en la solución 1). En resumen, esta solución empeora los tiempos obtenidos en el arreglo explícito. Se estima que no habrá cambios significativos entre los tiempos de distribución normal y lineal.

3) Representación con Huffman: este caso presenta una mejor solución en términos de memoria que la solución 1 y 2. Principalmente gracias a la compresión del arreglo de gaps, lo que significaría reducir el espacio ocupado por este en la mitad. Para la implementación de la codificación se requiere de al menos un heap y tres hashmap. El heap es importante para el proceso de codificación, en el cual se utiliza una cola de prioridad para obtener dos valores con la mínima frecuencia y así construir el árbol de huffman. Luego, los

hashmap son necesarios para guardar las codificaciones obtenidas del árbol de huffman (uno para codificar y otro para decodificar). Incluso considerando estas estructuras adicionales se prevé que en términos de memoria, este caso sea mejor a las soluciones anteriores. Con respecto al costo de la búsqueda binaria, teóricamente y basado en la notación asintótica, debería ser igual al de la solución 2, ya que el único costo adicional es el de la decodificación, pero al realizarse sobre un hashmap, esta es constante. Sin embargo, empíricamente debería ser menos costoso, ya que gracias a la compresión de los datos, estos son más ligeros y por tanto son más rápidas las operaciones que se ejecutan sobre ellos. Al igual que en las soluciones anteriores se estima que no habrá cambios significativos entre los tiempos de distribución normal y lineal.

1. Arreglo explícito

Inicialización de datos específica

Para evaluar la técnica de representación explícita, se generarán dos arreglos de números enteros en orden ascendente, diferenciándose en cuanto a sus distribuciones, siendo estas uniforme y normal.

Para generar el arreglo Lineal de forma aleatoria se pide un “ε” que será utilizado como número máximo que puede entregar la diferencia entre dos números consecutivos. Para el primer elemento del arreglo se usa el rand() de c++ y una seed a través de la librería “ctime”, el resultado se le aplica mod de “ε ”, y finalmente, para las n-1 casillas siguientes se suman un número aleatorio y el número de la casilla anterior.

Para generar el arreglo con Distribución Normal se pide un “sd” que será la desviación estándar de la distribución del arreglo. Se inicializa un generador de números aleatorios con la clase “mt19937”, se usa de semilla un objeto “random_device” y se declara un objeto “d” perteneciente a la clase normal_distribution<> con los argumentos size/2 (media) y “sd”. Las tres clases son sacadas de la librería “random”, se rellena el arreglo con los números generados por “d”, para posteriormente ordenar el arreglo ascendentemente con la función sort de la librería “algorithm”.

Pseudocódigo y costo

Lineal

Input: **n** (tamaño del arreglo a generar), **e** (máximo incremento entre los elementos del array).

Output: array de números random ordenado ascendentemente con una distribución uniforme.

```
lineal(n, e){
      Sea array[0..n-1] un arreglo de enteros
      array[0] = rand()
      for (i = 1 to n){
            array[i] = array[i-1] + rand() % e
      }
      return array
}
```

Costo: T(n) = O(n), pues se generan n números para completar el arreglo.

distNormal

Input: **n** (tamaño del arreglo a generar), **sd** (desviación estándar para los elementos del arreglo).

Output: array de números random ordenado ascendentemente con una distribución normal.

```
distNormal(n, sd){
      Sea array[0..n-1] un arreglo de enteros
      random_device rd
      mt19937 gen(rd())
      normal_distribucion<> d(size/2, sd)
      for(i = 0 to n){
            array[i] = (int)(d(gen))
      }
      sort(array, array+n)
      return array
}
```

Costo: T(n) = O(n), pues se recorre todo el arreglo para poder generarlo.

binarySearch
Input: arreglo original **arr**[0..n-1], **l** (índice izquierdo), **r** (índice derecho), **x** (elemento a buscar).
Output: True si x fue encontrado, False si no.

```
binarySearch(arr, l, r, x){
      if (r >= l) then {
            mid = l + (r - l) / 2
            if (arr[mid] == x) then {
                  return verdadero
            }
            if (arr[mid] > x) then {
                  return binarySearchBool(arr, l, mid - 1, x)
            }
            return binarySearchBool(arr, mid + 1, r, x)
      }
      retornar falso
}
```

Costo: T(n) = O(log(n)), ya que por cada iteración, se divide el arreglo *arr* por la mitad.

Espacio requerido

El espacio teórico requerido por esta solución para cada arreglo es de Memoria = (n*4) bytes, siendo n el largo de estos y 4 el tamaño ocupado por una variable de tipo ||INT||.

2. Arreglo representado con Gap-Coding

Inicialización de datos específica

Para los dos arreglos generados se construirá su representación gap-coding. Este se crea calculando la diferencia entre x_i y x_{i+1}, entonces el arreglo de gap-coding tendrá n elementos, al igual que los arreglos originales. Luego se crean los samples para cada arreglo, el cual tendrá “m” elementos elegidos con saltos de “b” casillas en el arreglo original.

Luego, para elegir las variables “m” y “b”, se escogen m = n / 4 y b = n / m = 4. Se llegó a estos valores luego de realizar algunas pruebas teóricas teniendo en cuenta la cantidad de iteraciones que la búsqueda puede hacer en el peor caso. Estas pruebas dieron los siguientes resultados:

Si m = n / 4, la solución 2 realizará, en el peor caso, 2 iteraciones más que la solución 1, debido a que:

$$T_{sol2}(n) = log_2(m) + b = log_2(n/4) + 4 = log_2(n) + 2$$

Luego, si m es un número más pequeño (n/6 o n/8), hará más iteraciones, pues b es más grande.

Sin embargo, si m es más grande, por ejemplo, m = n / 2, se tiene que:

$T_{sol2}(n) = log_2(m) + b = log_2(n/2) + 2 = log_2(n) + 1$, lo cual significa una iteración menos que con m = n / 4, pero esto implica un 25% más de memoria utilizada. Por tanto, se considera que la opción más balanceada en términos de rendimiento y memoria es con m = n / 4.

Pseudocódigos y costos

GapCoding

Input: El arreglo original **arr**[0..n-1] y **n** (tamaño del arreglo).
Output: Arreglo gap[0..n-1] con los gaps del arreglo *arr*.

```
gapCoding(arr[n], n){
      Sea gap[0..n-1] un arreglo de enteros
      gap[0] = arr[0]
      for (i = 1 to n - 1){
            gap[i] = arr[i] - arr[i - 1]
      }
      return gap
}
```

Costo: T(n) = O(n), ya que recorre linealmente el arreglo *arr* para construir *gap*.

CreateSample

Input: El arreglo original **arr**[0..n-1], **m** (tamaño del sample) y **b** (saltos del arreglo arr).
Output: Arreglo sample con *m* números del arreglo *arr* elegidos mediante saltos de *b* casillas.

```
createSample(arr, m, b){
      Sea sample[0..m-1] un arreglo de enteros
      for ( i = 0 to m - 1){
            sample[i] = arr[i * b]
      }
      return sample
}
```

Costo: T(m) = O(m), ya que recorre linealmente el arreglo *sample* y solo accede a elementos específicos de *arr*.

binarySearchSample

Input: El arreglo sample **samp**[0..m-1], **m** (tamaño del arreglo sample) y **x** (valor a buscar).

Output: Arreglo res[0..1] siendo res[0] (el índice del primer número menor a x en *samp*) y res[1] (el primer número menor a x en *samp*)

```
binarySearchSample(samp[], m, x){
      l = 0
      r = m- 1
      result = -1
      Sea res[1..2] un arreglo de enteros
      while(l ≤ r){
            mid = l + (r - 1) // 2
            if (samp[mid] == x){
                  result = mid
                  break
            } elif (samp[mid] < x){
                  result = mid
                  l = mid + 1
            }else{
                  r = mid -1
            }
      }
      res[0] = result
      res[1] = samp[result]
      return res
}
```

Costo: T(m) = O(log(m)), ya que por cada iteración, se divide el arreglo *sample* por la mitad.

busquedaLinealGap

Input: El arreglo **gap**[0..n-1], **n** (tamaño del arreglo *gap*), **x** (elemento a buscar en *gap*), **índice** (índice del primer número menor a x en *sample*, obtenido de binarySearchSample, res[0]), **b** (saltos del arreglo *sample*), **actual** (es el resultado obtenido de binarySearchSample, res[1]), **m** (tamaño del arreglo *sample*).

Output: El índice de x en el arreglo original si es que está, -1 si no.

```
busquedaLinealGap(gap[], n, x, índice, b, actual, m){
      i = b * índice
      if (indice == m - 1){
            lim = n
      }else {
            lim = b * (indice + 1)
      }
      while (i < lim AND actual < x){
            i = i + 1
            actual = actual + gap[i]
      }
      if (actual == x){
            return i
      }else{
            return -1
      }
}
```

Costo: T(n) = b, como b = n/m, T(n) = O(b), ya que es una búsqueda lineal en un intervalo del arreglo *gap* que siempre es n / m.

El costo de la búsqueda total en estos algoritmos será: T(n) = O(log(m) + C). Esto debido a que en primer lugar se realiza una búsqueda binaria sobre el arreglo “sample”, cuyo tamaño es “m”, para encontrar los índices donde se buscará en el GapCodingArray. Luego se recorre linealmente dicho intervalo buscando el número original (tamaño “b”). Expresando el costo asintótico en función de “n” resulta: T(n) = O(log(n/4)) = O(log n).

Espacio requerido

El espacio adicional teórico que requiere la solución 2 por cada arreglo es de m*4 bytes y n*4 bytes, esto porque se requiere de un arreglo *sample* de largo

“m”, y un arreglo *gap* de largo “n”. Se multiplica por 4 debido al tamaño del tipo de variable ||INT||, ocupando un total de: Memoria = (4*(n+m)) bytes.

3. Representación con Huffman

Inicialización de datos específica

Se mantendrán las estructuras ya usadas en el segundo caso, el sample y Gap-CodingArray, con la diferencia de que en este último los elementos serán codificados según la representación de Huffman. Esto con el objetivo de reducir el espacio requerido en RAM.

Para la creación de los códigos se usarán 4 estructuras adicionales:

- Nodo de Huffman (struct): nos permitirá guardar cada valor con su frecuencia para luego ser usados por el HuffmanTree.
- HashMap (x3): Son seleccionados pues nos facilitaran el acceso a la codificación y decodificación de los valores.
- PriorityQueue: Se usará por su fácil manipulación y creación.
- HuffmanTree: será empleado pues es requerido para la codificación.

Cada *nodo de Huffman* contendrá un dato (||INT||), su frecuencia en el arreglo de *gapCoding* (||INT||) y los punteros a los hijos izquierdo y derecho si es que los tiene. Luego se inicializará un *HashMap* “m”, para contar y guardar las frecuencias de los elementos de *gapCoding*, con “m” se crea una *priorityQueue* de nodos, que servirá para generar nuestro *HuffmanTree* (que tiene 2ε-1 nodos como máximo).

Todo esto en conjunto entregarán dos *HashMaps*. El primero con el par (clave:valor) en donde la clave corresponde al código de Huffman de un elemento y el valor corresponde al número que representa. El segundo en cambio tendrá el número como clave y su codificación como valor, con el propósito de generar el arreglo de GapCodificado en tiempo lineal y decodificar en tiempo constante.

Pseudocódigo y costo

countFreq

Input: ***gap*** arreglo de gaps, ***n*** tamaño del arreglo.

Output: *hashMap* con la frecuencia de cada elemento de *gap*

```
countFreq(gap[0,n-1], n){
      Sea hashMap un HashMap de tipo (data(int): frec(int))
      hashMap.insert(gap[0],1)
      for (i=0 to n-1){
            if(hashMap.find(gap[0]) == hashMap.end()){
                  hashMap.insert(gap[i],1)
            }
            else{
                  hashMap.at(gap[i])++
            }
      }
      return hashMap
}
```

Costo: T(n) = O(n), ya que se recorre el arreglo de gapCoding por completo.

GenerateTree

Input: **pq** (cola de prioridad con a lo más ε nodos).

Output: raíz del HuffmanTree(con a lo más 2ε-1 nodos).

```
generateTree(pq){
      while(pq.size() != 1){
            nodoHuffman* izq = pq.top()
            pq.pop()
            nodoHuffman* der = pq.top()
            pq.pop()
            nodoHuffman* nodo= new nodoHuffman('$',izq.frec+der.frec)

            nodo.izq = izq
            nodo.der = der
            pq.push(nodo)
      }
      return pq.top()
}
```

Costo: T(ε) = O(2ε-1*log(2ε-1) = O(ε logε), siendo ε la cantidad de gaps distintos.

binarySearchSample

Input: El arreglo sample **samp**[0..m-1], **m** (tamaño del arreglo sample) y **x** (valor a buscar).

Output: Arreglo res[0..1] siendo res[0] (el índice del primer número menor a x en *samp*) y res[1] (el primer número menor a x en *samp*)

Misma rutina utilizada en la solución 2. Costo: T(m) = O(log(m)), ya que por cada iteración, se divide el arreglo *sample* por la mitad.

busquedaLinealGapHuffman

Input: Arreglo codificado **gap**[0..n-1], **n** (tamaño del arreglo *gap*), **x**(elemento a buscar en *gap*) , **índice** (índice del primer número menor a x en *sample*, obtenido de binarySearchSample, res[0]), **b** (saltos del arreglo *sample*), **actual** (es el resultado obtenido de binarySearchSample, res[1]), **m** (tamaño del arreglo *sample*), **hashmap** (hashmap que contiene las las codificaciones y su valor).

Output: El índice de x en el arreglo original si es que está, -1 si no.

```
busquedaLinealGapHuffman(gap[], n, x, indice, b, actual, m, hashmap){
	i = b * índice
	if (indice == m - 1){
		lim = n
	}else {
		lim = b * (indice + 1)
	}
	while (i < lim AND actual < x){
		i = i + 1
		actual = actual + hashmap[gap[i]]
	}
	if (actual == x){
		return i
	}
	else{
		return -1
	}
}
```

Costo: T(n) = b, como b = n/m, T(n) = O(b), ya que es una búsqueda lineal en un intervalo del arreglo *gap* que siempre es n/m.

El costo de la búsqueda total en estos algoritmos será igual que en la solución anterior: T(n) = O(log(n/4)) = O(log n).

Manejo de Outliers

El manejo de outliers se realiza para lograr una menor compresión a la hora de aplicar codificación de Huffman, y así mejorar la memoria que utiliza el programa. Sin embargo, como se guardan los códigos en una variable unsigned short, todos los códigos ocupan el mismo espacio en memoria sin importar su largo. Por lo tanto el manejo de outliers no repercute en ningún aspecto de la solución en la práctica.

Si es que fuera posible ocupar un tipo de dato con un largo variable de bits, una buena forma de tratar los outliers sería ocupar medidas estadísticas tales como la media y la desviación estándar de las frecuencias para determinar los outliers. Y una vez identificados convendría no codificarlos para evitar empeorar drásticamente la compresión. Una buena manera de identificar outliers puede ser:

1) Calcular media y desviación estándar de las frecuencias.
2) Elegir un "umbral".
3) Para cada dato, calcular si este se encuentra "umbral" veces la desviación estándar alejado de la media. Es decir:

```
if (dato < media - (umbral * desvEstandar) OR dato > umbral * (desvEstandar + media))
```

Espacio requerido

La solución tres, al igual que la anterior ocupará un sample de m elementos, y un arreglo de gapCoding, pero este ya estará codificado, usando solo 2*n bytes, la mitad de lo que ocupaba en su versión anterior. Esto debido al tamaño del tipo de dato ||SHORT|| (tipo de dato que se decidió utilizar). El total de memoria usada en la búsqueda es de:

Memoria = (2*n + 4*m + 56) bytes teóricamente.

Para la medición de memoria de esta solución no se tienen en cuenta las estructuras necesarias para la codificación ya mencionadas anteriormente. Cada nodo de Huffman ocupa 24 bytes, la priorityQueue al ser un vector ocupa 12 bytes, y tiene ε nodos (con ε siendo la cantidad de gaps distintos), ocupando en total 12 + ε*24 bytes. Luego el HuffmanTree al estar compuesto solo por nodos, ocupará (2ε-1)*24 bytes y los HashMaps ocupan 56 bytes cada uno.

Rendimiento

La metodología a seguir para medir el rendimiento de cada solución es la siguiente:

Para cada solución se inicializan los datos y se realiza la búsqueda de 10000 valores aleatorios que están en el rango entre 0 y el último número del arreglo original (el máximo valor generado). Esto fue diseñado así para intentar provocar búsquedas de valores que se encuentren en el arreglo y búsquedas de valores que no estén. Se medirán los tiempos que demora cada solución en sus respectivas búsquedas, variando los tamaños de los arreglos. Se experimentará con arreglos de largos 10^6, 10^7 y 10^8.

Luego, se construirán tablas y gráficos que muestren los resultados de los tiempos obtenidos, para luego generar un análisis y así encontrar la mejor solución.

Además, se medirá la memoria usando el método “sizeof” de c++, por lo que se obviará la cantidad de memoria usada por instrucciones.

Y finalmente, para los arreglos de distribución normal, se presentarán los resultados con una desviación estándar 100, debido a que genera una cantidad de gaps distintos bastante similar a la producida por un arreglo lineal, por tanto es más “justa” la comparación.

Experimentación

En este punto se buscará comparar los tres métodos ocupados, con el fin de comprobar la veracidad de nuestra hipótesis y encontrar una solución óptima en cuanto a tiempo y espacio. Esta, será realizada en un equipo con las siguientes especificaciones:

- Procesador: i5 9400f, 64 bits
- RAM: 16GB, 2666 Mhz
- GPU: 1650 GeForce GTX
- OS: Linux

Resultados

Gráfico 1

En el siguiente gráfico se presentan las distribuciones de los arreglos de entrada a las soluciones.

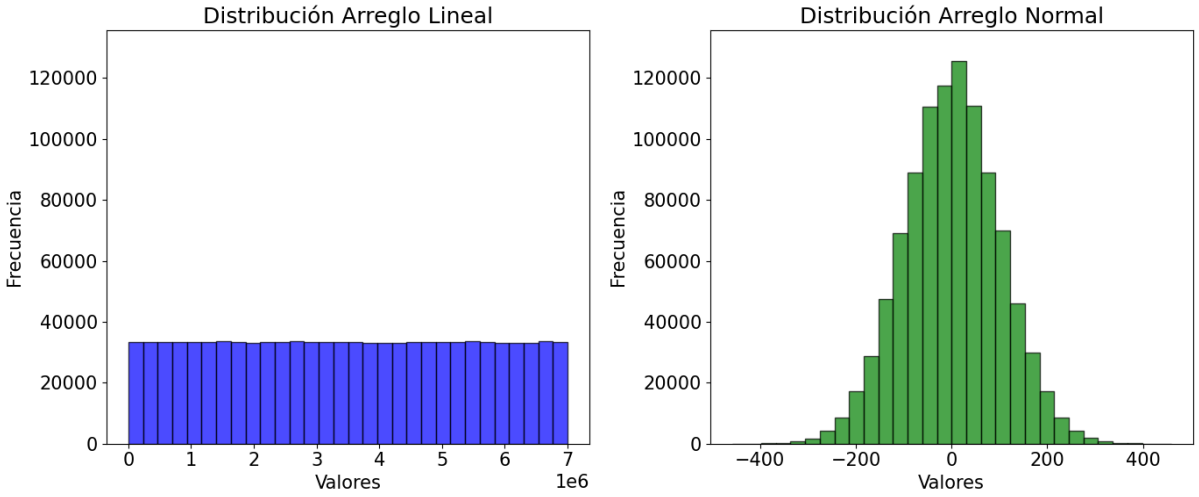

Figura 1

En los siguientes 3 gráficos se muestran los tiempos obtenidos en cada solución, clasificado por la cantidad de números generados (“n”) y por el tipo de distribución (lineal y normal).

Gráfico 2

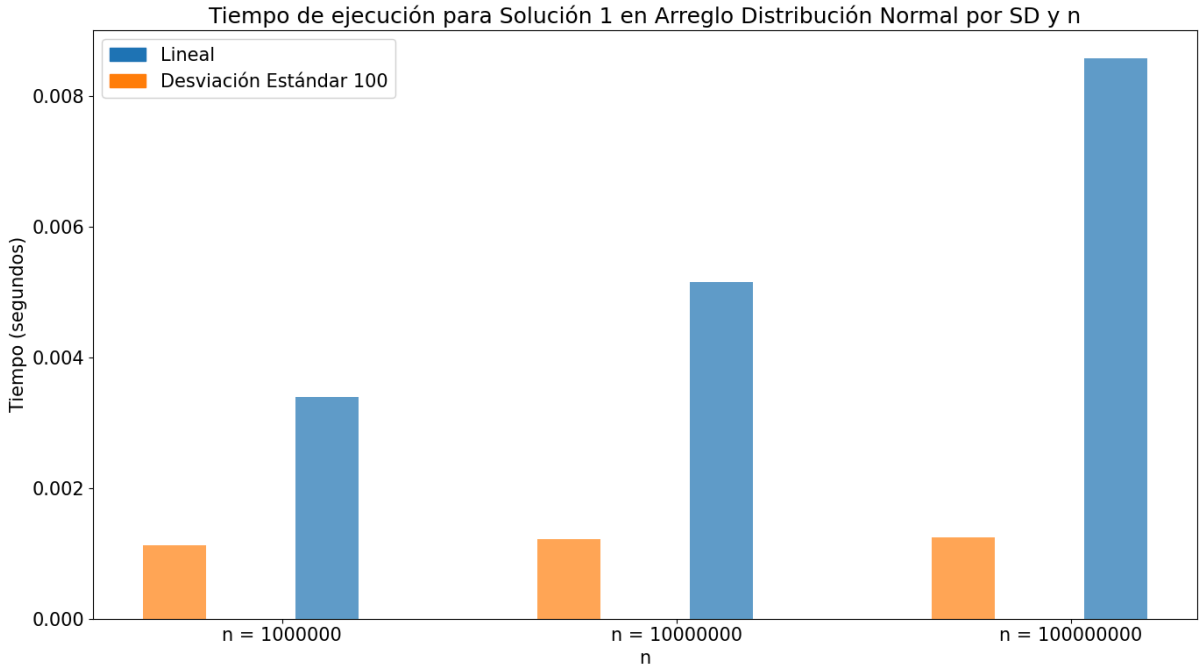

Figura 2

Gráfico 3

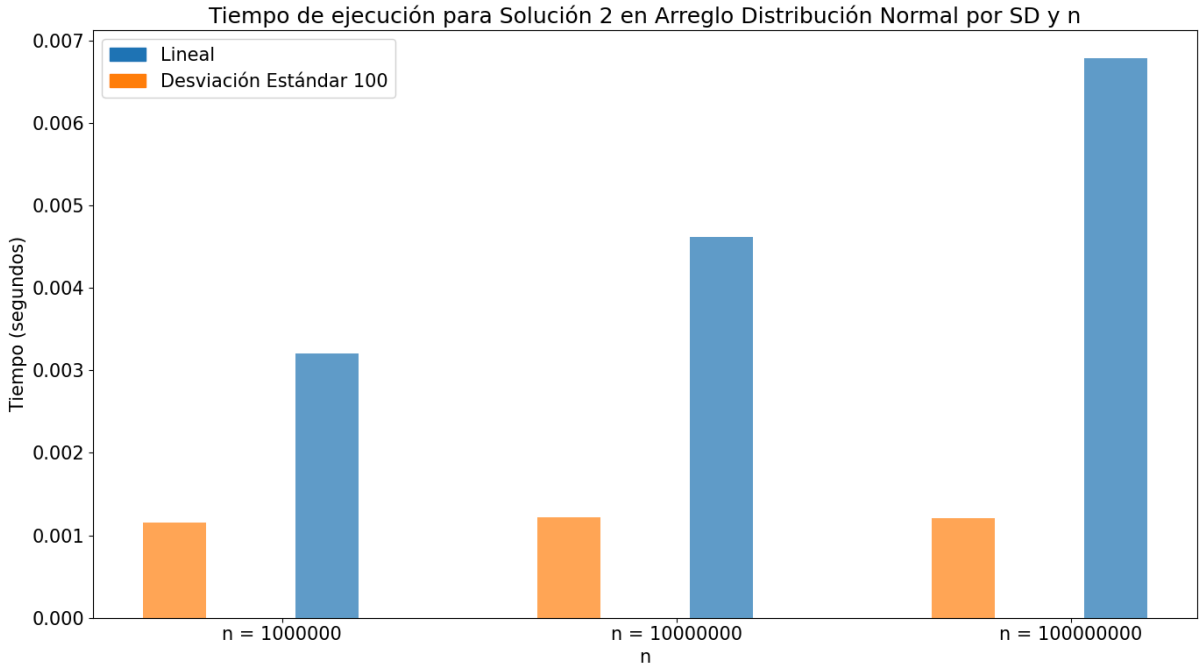

Figura 3

Gráfico 4

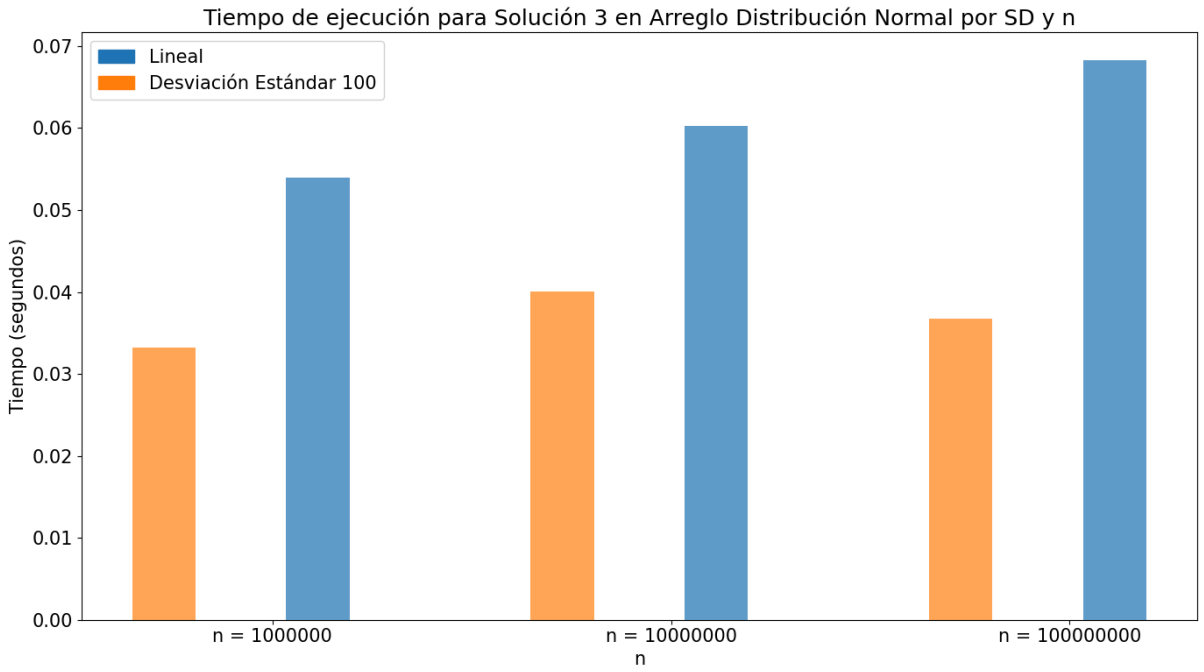

Figura 4

Gráfico 5

En el siguiente gráfico se muestran los tiempos de búsqueda clasificados por la cantidad de números generados (“n”) y por solución. Cabe destacar que cada tiempo fue calculado como el promedio entre la medición del arreglo con distribución lineal y el arreglo con distribución normal.

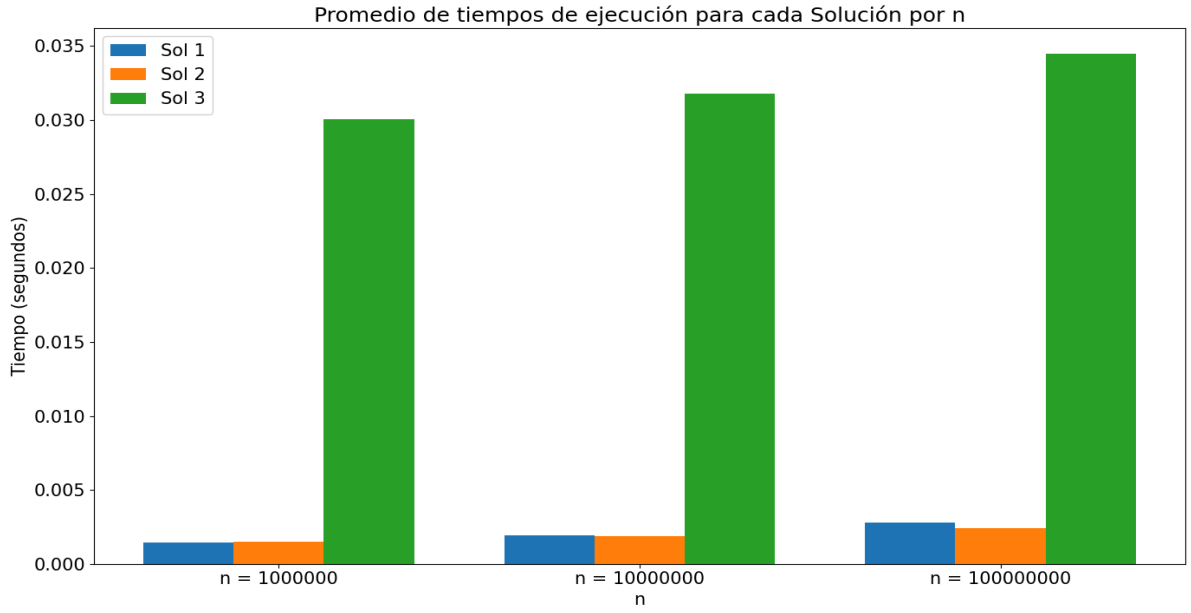

Figura 5

Gráfico 6

En el siguiente gráfico se expone la cantidad de memoria usada por todas las estructuras, clasificada por la cantidad de números generados (“n”) y categorizada por solución. (Solo en las distribuciones lineales).

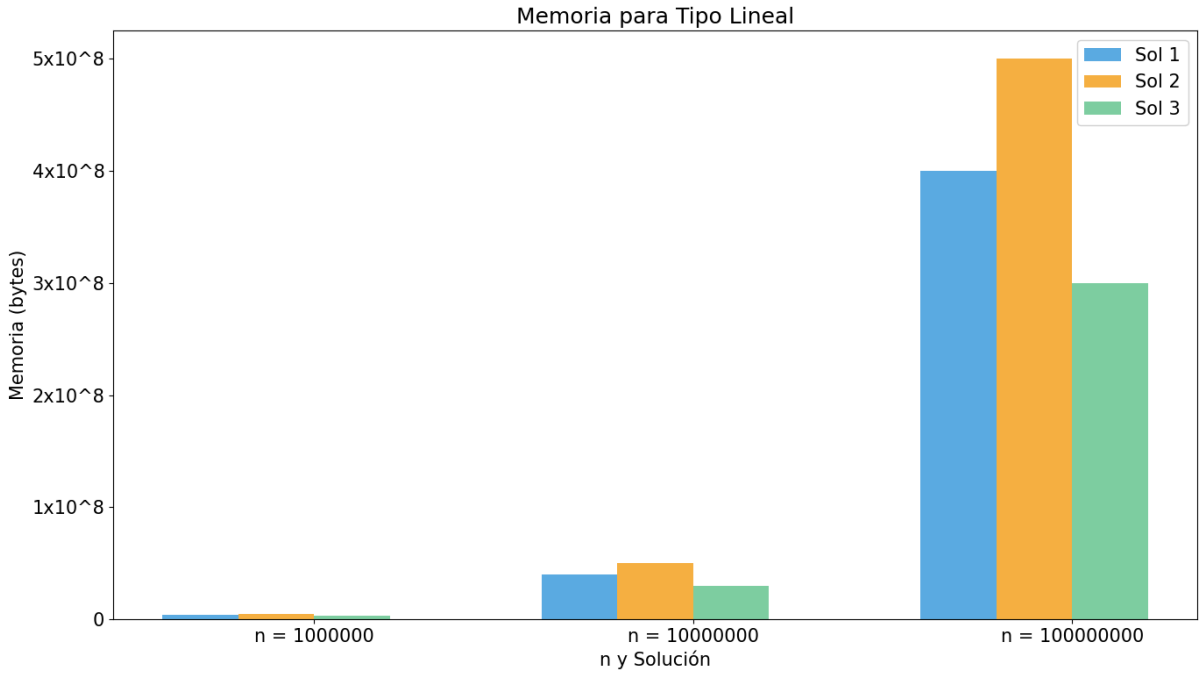

Figura 6

Tabla 1

En la siguiente tabla se comparan los valores predecidos y los valores resultantes de los experimentos relacionados con la memoria ocupada por la búsqueda.

	Comparación Predicción-Resultado Memoria					
N	10^6		10^7		10^8	
Solución	Predicción	Resultado	Predicción	Resultado	Predicción	Resultado
Solución 1	4,0 MB	4,000012 MB	40,0 MB	40,000012 MB	400,0 MB	400,000012 MB
Solución 2	5,0 MB	5,000020 MB	50,0 MB	50,000020 MB	500,0 MB	500,000020 MB
Solución 3	3,000056 MB	3,000174 MB	30,000056 MB	30,000174 MB	300,000056 MB	300,000174 MB

Figura 7

Tabla 2

En la siguiente tabla se muestran los valores resultantes de los experimentos relacionados con los tiempos de las búsquedas.

	Tiempos de Búsqueda registrados en Segundos					
N	10^6		10^7		10^8	
Solución	Lineal	Normal	Lineal	Normal	Lineal	Normal
Solución 1	0.003398	0.001125	0.005146	0.001216	0.008576	0.00125
Solución 2	0.003207	0.001158	0.004612	0.001223	0.006786	0.00121
Solución 3	0.053914	0.033264	0.060245	0.04007	0.068306	0.036726

Figura 8

Conclusión

Siendo el objetivo principal de este trabajo comparar tres técnicas de representación y comprensión de arreglos ordenados, mediante la medición de tiempo y uso de memoria en RAM, se detallan las principales contradicciones y comprobaciones de la hipótesis planteada en un principio, contrastando los resultados obtenidos con esta.

Para empezar, entre las figuras *2,3 y 4*, es claro que el tiempo de búsqueda en el arreglo con distribución normal es menor que en el arreglo con distribución uniforme, esto sin importar el tamaño del arreglo para las tres soluciones, lo cual puede deberse a que la distribución de los datos si afecta en la eficiencia de búsqueda, contrario a lo dicho en la hipótesis. Esta clara disminución de tiempo puede ser derivada de varios factores, puesto que en una distribución normal la mayoría de los datos se concentran muy próximos a la media, por lo que al hacer una búsqueda de elementos que estén dentro de este intervalo, se pueden reducir drásticamente los tiempos, mejorando mucho el caso promedio de la búsqueda. Otro punto a tomar en cuenta, es que la codificación de Huffman se ve favorecida con la distribución normal ya que los elementos tienen frecuencias distintas entre sí, lo que sería el mejor caso para la codificación, caso contrario a la distribución uniforme que se acerca a la entropía del peor caso.

Como se detalló en la hipótesis y confirmándose a partir de la figuras 6 y 7, en términos de memoria la solución tres es la mejor, seguida por la solución uno, siendo la solución dos la peor.

Todo esto es útil para distinguir cuales son las ventajas y desventajas de cada técnica, los casos en los cuales se destacan, para finalmente comparar las soluciones. La solución 1 presenta resultados bastante equilibrados, siendo la segunda mejor en tiempo y la segunda mejor en memoria. Si bien la segunda solución ocupa más espacio que su antecesora, los tiempos se reducen, y estos tiempos mejoran a medida que el “n” es más grande, por lo que se puede concluir que su uso conviene cuando la cantidad de datos sea considerablemente alta y se necesiten tiempos notablemente bajos. No así la solución 3, que empeora sus tiempos de búsqueda a costa de reducir su uso en memoria, lo que la hace destacar en casos donde la entrada de datos sea masiva y no se requiera de una solución excesivamente rápida. La solución 3 muestra peores resultados en rendimiento que las demás, porque pareciera que es más costoso de lo que se pensaba en un principio el acceso al hashmap para decodificar.

Una de las principales limitaciones a las que nos enfrentamos es la incapacidad de ocupar datos con largos variables para realizar la codificación de Huffman, dado que no se contaba con la capacitación y conocimiento para trabajar de esta manera. La solución tres mejoraría significativamente en cuanto a uso de memoria con códigos de largo variable.

En conclusión, este trabajo ofrece un análisis profundo de las tres técnicas analizadas en cuanto a tiempo y memoria basados en los resultados. Como se pudo apreciar a lo largo de la experimentación, no sólo las estructuras y algoritmos usados afectan el rendimiento, sino, también es afectado por cúal es la distribución de los datos de entrada. Con vistas a futuro, sería ventajoso realizar más experimentaciones con respecto a esta área para así identificar tendencias con “n” más grandes y con tipos de datos con largos variables, para lograr dar con la mejor solución posible.

Bibliografía

std::bitset - cppreference.com. (s. f.).

https://en.cppreference.com/w/cpp/utility/bitset

IBM i 7.5. (s. f.).

https://www.ibm.com/docs/es/i/7.5?topic=functions-atoi-convert-character-string-integer

GeeksforGeeks. (2023, 20 marzo). *Huffman Coding using Priority Queue*. GeeksforGeeks.

https://www.geeksforgeeks.org/huffman-coding-using-priority-queue/?ref=lbp

Castrillo, M. (2023, 14 septiembre). Cómo identificar y tratar outliers con Python - Marta Castrillo - Medium. *Medium*.

https://medium.com/@martacasdelg/c%C3%B3mo-identificar-y-tratar-outliers-con-python-bf7dd530fc3