FUNDAMENTOS DE ANÁLISIS DE ALGORITMOS

Práctica 4: Divide y Vencerás

Algoritmos de Búsqueda y Ordenación Avanzados

Luis Airam Saavedra Paiseo

Contenido

1.Búsqueda Binaria con Variantes.

1.1. Análisis Teórico.

Hemos utilizado, para hallar el coste temporal en los diversos casos de las 3 versiones del algoritmo de búsqueda "Búsqueda Binaria", la técnica de divide y vencerás, proponiendo, para todas las versiones, el siguiente sistema recurrente:

$$T(n) = \begin{cases} O(1) & si\ n = 0\ . \\ T(\lfloor n/2 \rfloor) + O(1) & si\ n > 0. \end{cases}$$

Dado que los 3 tienen la misma estructura, utilizaremos el mismo sistema para todas las versiones propuestas.

Si resolvemos dicho sistema recurrente, obtendremos:

1.1.1. Primer algoritmo (Búsqueda Binaria clásica) .

Su mejor caso viene provocado por la entrada de datos en la cual el elemento X a buscar se encuentre justo en el medio de este, O(1). Por el contrario, el peor caso viene dado por la entrada de datos en la cual dicho elemento no se encuentra en los datos o se encuentre en los extremos de la entrada O(log n). Para el caso promedio hemos utilizado la siguiente formula: P*(X pertenece) +(1-P)*(X no pertenece), siendo P la probabilidad de que el elemento se encuentre en el vector, y 1-P la probabilidad del evento contrario, simplificada para cálculos mas simples a 0.5 para cada caso, pues asumimos que son equiprobables.

En el caso de que el elemento X pertenezca al vector, nos encontramos con dos posibles salidas, que se encuentre en la primera iteración (siendo las OE realizadas de O(1) por lo que no las tendremos en cuenta), y de que se encuentre, pero en los extremos, realizando log n -1 OE para encontrarlo.

En el caso de que el elemento X no pertenezca al vector, realizaremos log n +1 OE para asegurarnos de que dicho elemento no se encuentra en los extremos.

Si sustituimos dichos valores en la formula, nos encontramos con que, el caso promedio pertenece al O(log n), igual que el caso peor, y el caso mejor pertenece al O(1), tal y como mencionamos al principio.

1.1.2. Segundo algoritmo (Búsqueda Binaria con índice de primera ocurrencia).

Igual que la versión clásica, pertenecen sus casos peor y promedio al O(log n) y su mejor caso al O(1) , y utilizaremos también para hallar la complejidad temporal el mismo sistema recurrente.

En el mejor de los casos, la entrada que lo provoca, al igual que en la versión anterior, es aquella en la que encontramos el elemento X buscado justo en el medio, con el matiz de que en esta versión, tiene que ser la primera ocurrencia de dicho elemento, es decir, que no se encuentre otra ocurrencia a la izquierda/derecha de donde se encuentre ubicado; Por el contrario, en el peor de sus casos, esto estaría provocado por una entrada en la cual el elemento, o bien no se encuentre en la entrada de datos, o todas sus ocurrencias estén agrupadas a los extremos de esta.

Para el caso promedio, hemos utilizado el mismo enfoque que para la versión anterior, por lo que no entraremos en los mismos detalles y especificaciones, pero si diremos que, efectivamente, el caso peor y promedio pertenecen al O(log n) y el caso mejor al O(1).

1.1.3. Tercer algoritmo (Búsqueda Binaria con interpolación).

El enfoque y sistema recurrente utilizado será el mismo que para sus 2 versiones anteriores ya tratadas, en el mejor de sus casos, la entrada que lo provoca es aquella en la que el elemento x está en la posición estimada por interpolación en la primera iteración, en cambio, en el peor de los casos, la entrada que lo provoca es aquella tal que los datos de entrada no estén distribuidos de acuerdo a una distribución uniforme, por ejemplo una gaussiana, ya que la interpolación estimada puede fallar y no dividir la entrada de datos en mitades de igual tamaño.

Para el caso promedio, también en esta versión, hemos utilizado el mismo enfoque que para las versiones anteriores, por lo que no entraremos en los mismos detalles y especificaciones, pero si diremos que, efectivamente, el caso peor y promedio pertenecen al O(log n) y el caso mejor al O(1).

1.2. Comparación entre versiones 2 y 3 del algoritmo.

1.2.1. ¿Cuándo es mejor usar la versión con interpolación? .

Las situaciones en las que encontraremos que la versión por interpolación es más eficiente son:

- -Datos ordenados siguiendo una distribución uniforme, pues la interpolación acierta a la primera casi asegurado.
- -Ocurrencias ubicadas en los extremos, siempre y cuando siga una distribución uniforme, en entradas de datos grandes, esta versión funcionará de manera mas eficiente, a pesar de encontrar la ocurrencia en los extremos, la interpolación será mas eficiente que la búsqueda de la primera ocurrencia.

1.2.2. ¿Cuándo podría suponer un peor rendimiento dicha versión?.

Las situaciones en las que encontraremos que esta versión no es la mas eficiente son:

- Datos ordenados, pero no siguiendo una distribución uniforme, pues la interpolación está diseñada específicamente para ser mas eficiente para datos que siguen dicha distribución.
- Elemento a buscar no existente, pues esta versión puede llegar a requerir de mas operaciones a realizar para determinar que el elemento es inexistente.
- Entradas de datos de tamaño pequeño, pues la sobrecarga del cálculo de la interpolación no compensará la mejora.
- Elementos duplicados, pues la condición de parada es más compleja.

1.2.3. ¿Qué consideraciones hay que tener en cuenta para utilizar dicha versión?.

Las consideraciones numéricas que hemos tenido en cuenta para utilizar la versión con interpolación, son:

- División por cero: Cuando A[der] == A[izq], necesita manejo especial.
- Overflow: El cálculo (x-A[izq])*(der-izq) puede desbordar con números grandes.
- Precisión: Al trabajar con enteros, se pierde precisión frente a una implementación con floats.
- Posiciones fuera de rango: La posición estimada puede caer fuera del rango válido.

Podrá encontrar la implementación o bien en el fichero adjunto o en el siguiente enlace: Comparación versión 2 y 3.

2.Implementación y Análisis Empírico.

2.1. Implementaciones de las 3 versiones de los algoritmos.

La implementación utilizada para estos 3 algoritmos, se podrá encontrar en el fichero adjunto a este documento pdf, o bien en el siguiente enlace a github, en el cual he alojado este proyecto al completo, para todo aquel que quiera verlo y aportar ideas/mejoras, este enlace llevara concretamente a la subactividad en la que he implementado y comparado las 3 versiones: Comparación de las 3 versiones.

2.2. Diseño y ejecución para un conjunto de pruebas concreto.

Para esto, y aprovechando la implementación básica de los 3 algoritmos que es la utilizada, utilice la misma implementación para hacer el conjunto de pruebas exhaustivas para la realización de este estudio puramente empírico, utilizando una entrada de datos generada de forma aleatoria acorde a diversas distribuciones, y utilizando unas funciones y llamadas especificas para poder verificar que se cubren los siguientes casos concretos:

- Arrays vacíos
- Arrays con un solo elemento
- Arrays con elementos repetidos
- Búsqueda de elementos al principio, medio y final
- Búsqueda de elementos inexistentes
- Para BÚSQUEDABINARIA2: verificación de que siempre devuelve la primera ocurrencia
- Para BÚSQUEDABINARIAINTERPOLACIÓN: arrays con distribución uniforme y no uniforme.

Encontrará dicha implementación, igual que la anterior, en el fichero adjunto a este pdf, o bien en el siguiente enlace: Implementación para verificar estudio empírico.

2.3. Verificaciones.

La implementación de BusquedaBinaria2 garantiza que siempre devuelva la primera ocurrencia mediante una condición adicional que verifica si el elemento encontrado es efectivamente el primero en la secuencia . Cuando encuentra el valor, pero no es la primera ocurrencia, continúa buscando en la submatriz izquierda. Esto se evidencia en las pruebas con arrays como [1,1,1,1, 1,...] donde siempre retorna posición 0, y en el mayor número de Operaciones Elementales respecto a Binaria1, derivadas de las comprobaciones adicionales.

Un enfoque diferente, seria basarnos en los siguientes 3 aspectos:

- Mecanismo del código implementado: La garantía de que esto siempre ocurra, es la siguiente línea de código

```
if (A[medio] ==    && (medio == izq || A[medio - 1] != x)) {
            return medio; // ← Punto clave
}
```

Pues esta condición, verifica simultáneamente que el elemento medio sea igual al buscado (A[medio]==x), y que sea la primera ocurrencia porque, o bien esta en el límite izquierdo (medio==izq), o bien el elemento anterior es diferente (A[medio-1] != x).

- Comportamiento recursivo del algoritmo, esto lo comprobamos con la siguiente línea de código

```
else if (A[medio] >= x) {

    return BusquedaBinaria2(A, izq, medio - 1, x); // ← Busca en la mitad izquierda
}
```

Pues el uso del >= asegura que, si hay repeticiones, siga buscando hacia la izquierda hasta llegar al límite, y que no p ueda “saltarse” la primera ocurrencia porque el algoritmo siempre explorara la mitad relevante al completo.

- Evidencia empírica, pues en mi implementación encontramos las siguientes demostraciones

1.Caso manual de repeticiones controladas, que verificamos en la siguiente línea de código.

```
<int> testRepetidos = {1,1,1,1,1,2,2,3,4,4,4,4,5,6,7,8,9,10,10,10};
```

y que, para el valor 1 devuelve siempre la posición 0 (la primera ocurrencia) , y para el valor 4 devolverá la posición 8 (que es la primera ocurrencia de dicho valor) , se podría comprobar de igual manera para cualquier valor de la entrada de datos y obtendríamos siempre la posición de la primera ocurrencia.

2.Operaciones elementales (OE).

La segunda versión de este algoritmo muestra consistentemente mayor OE que la versión clásica para diversos tamaños, coincidiendo con el overhead de verificar A[medio-1] != x

3.Comparación y Análisis.

3.1. Implementación iterativa.

La implementación iterativa de las 3 versiones del algoritmo Búsqueda Binaria podrán encontrarse, o bien en el fichero adjunto al pdf, o en el siguiente enlace:

Implementación versión iterativa.

3.2. Estudio empírico.

Igual que en el anterior punto, dicho estudio empírico ha sido realizado aprovechando la anterior implementación iterativa de los algoritmos, su implementación se podrá encontrar de igual manera en el fichero adjunto o en el siguiente enlace: Comparación iterativas/recursivas.

Su fichero Excel en el que se alojan las gráficas también podrá encontrarse en los ficheros adjuntos, o bien, en la información dejada a modo de interés en el final de este mismo informe.

3.3. Comparación rendimiento versión 3 y versión 1.

Esta comparación, lo único que nos dice de forma clara, es que la versión iterativa de la búsqueda binaria por interpolaciones no es para nada efectiva en comparación, en este caso, con la búsqueda binaria iterativa clásica, en el Excel de dicha implementación apreciamos como, incluso para una distribución uniforme, no es tan efectiva como su versión recursiva, pues en este caso la interpolación hace operaciones de más, incluso cuando el elemento se encuentra en la entrada de datos, cosa que la iterativa básica optimiza debido a su forma de actuar una vez se ejecuta, puede encontrar su implementación aquí: Comparación versión 1 y 3 iterativos.

3.4 Análisis de rendimiento y verificaciones.

En términos de coste temporal, podíamos llegar a esperar que la versión por interpolación fuese mas eficiente con una distribución uniforme, cosa que no ha podido ser verificada, pues la grafica refleja un comportamiento atípico para este algoritmo, mostrando unos picos irregulares que probablemente van de la mano, o bien de valores que no se encuentran en el vector, o de valores que se ubican en los extremos, haciendo que la interpolación, además de no funcionar adecuadamente, requiera de un coste temporal mayor a la vez que mas operaciones elementales realizadas, cosa que su versión clásica ha logrado realizar de manera mas eficiente, pues mantiene su comportamiento para la distribución uniforme, y otras como la gaussiana o la exponencial que han sido las utilizadas en este caso.

4.MergeSort y optimizaciones.

4.1. Diseño e implementación.

El diseño e implementación de la función MERGE solicitada se podrá ver en el fichero adjunto, y en este enlace: Implementación versión estándar. Destacar que esta implementación mantendrá para todos sus casos un orden de O(n*log n), y que utiliza el enfoque de divide y vencerás, que se aprecia implementado casi al completo en la función auxiliar MERGE que combina y ordena las subdivisiones realizadas por las llamadas recursivas de la función MergeSort.

4.2. Implementación versión optimizada.

El único dato que cabria destacar de esta implementación, es que utilizamos, además del mergesort, el algoritmo insertionsort o algoritmo por inserción para subarrays pequeños, necesitando determinar ese valor k para el cual predominara el algoritmo de inserción por encima del de mezcla, se dejara un ejemplo de su ejecución a modo de información de interés en el final de este mismo informe, si se desea ver el código de la implementación con mas detalle, se puede ver en el fichero auxiliar y en este enlace: Implementación MergeSort hibrido. Como pequeña conclusión acerca de esta implementación hibrida podemos decir que, para tamaños de entrada grandes relativamente, el valor k oscilara entre el 30 y el 40, mientras que para tamaños de entrada pequeños se encontrara entre el 5 y el 15.

4.3.Diseño e implementación, valor k optimo .

Prosiguiendo con lo mencionado en el apartado anterior, pues he utilizado la misma implementación, he añadido una función auxiliar que genere vectores de distintos tamaños de entrada y valores aleatorios y unos valores de k concretos para calcular los diferentes tiempos de ejecución y asi poder determinar cual es ese valor k optimo que buscamos, como bien se ha mencionado, este valor ira aumentando a medida que también lo haga el tamaño de entrada, siendo en el peor de los casos, 40 dicho valor.

Podemos concluir con que, para tamaños de entrada pequeños, siempre es preferible utilizar el algoritmo por inserción frente al algoritmo por mezcla debido a la rápida ejecución de este, mientras que, para tamaños de entrada

mas grandes, es preferible utilizar el algoritmo de ordenación por mezcla, pues funciona de manera mas eficiente gracias a ese enfoque de Divide y Vencerás y a esas subdivisiones del problema.

Dado que su implementación es la misma que la del apartado anterior, puede encontrar la misma clickando en el enlace del apartado anterior o en el fichero adjunto.

4.4.Diseño e implementación MergeSort modificado

Para esta implementación, hemos diseñado e implementado una variante del MergeSort clásico, con el único detalle de que esta nueva versión no genera subarrays temporales, solamente crea y utiliza un único array, con el cual realizara las ordenaciones y combinaciones necesarias para la ordenación.

Ambas versiones son muy eficientes, pero a partir de un tamaño considerable (en este caso de 1000), podemos observar como la versión modificada empieza a generar una diferencia notable en cuanto a eficiencia, pues empieza a surtir efecto el no hacer ninguna subdivisión ni llamadas recursivas adicionales, cosa que hace que la versión clásica del mismo llegue a ser mas lenta o ineficiente para tamaños de entrada más grandes.

Se dejara al final de este informe un ejemplo de ejecución de ambos algoritmos para la demostración de esta pequeña conclusión, asi como en el fichero adjunto se podrá encontrar la implementación utilizada para comparar los mismos, o también, podrá encontrarla en este enlace: Implementación para comparación de versiones.

5.Aplicacion Practica .

Resumiremos todas las actividades propuestas, en un único punto, pues hablaremos constantemente del mismo algoritmo.

Yo he utilizado la versión estándar del algoritmo MergeSort, pues cumple con la condición requerida de que sea menor a O(n^2) en el peor de los casos, siendo este de O(n*log n), adicionalmente a esto, uso el MergeSort únicamente para ordenar de menor a mayor el vector de entrada, pero mi algoritmo realiza un filtrado de valores gracias a una función auxiliar, en este caso gracias a esta:

```
vector<int> filtrarUnicos(vector<int>& A) {
    vector<int> unicos;
    int n = A.size();
    if (n == 0) return unicos;

    for (int i = 0; i < n; i++) {
        bool esUnico = true;
        if (i > 0 && A[i] == A[i - 1]) esUnico = false;
        if (i < n - 1 && A[i] == A[i + 1]) esUnico = false;

        if (esUnico) {
            unicos.push_back(A[i]);
        }
    }
    return unicos;
}
```

Función con la cual podremos filtrar para el tamaño del vector, todos aquellos valores que tengan una única ocurrencia, generando un vector de valores únicos, el cual ya estará ordenado de menor a mayor gracias a haber ejecutado el algoritmo MergeSort antes de buscar aquellos valores únicos, que nos acabará devolviendo el vector de valores únicos ordenados tal y como nos pide nuestro problema.

En cuanto al coste temporal y espacial, podemos decir que:

- Coste temporal: dado que utilizamos el algoritmo MergeSort, predomina O(n * log n) a pesar de que la función de filtrado de elementos pertenezca a O(n), por lo que tendrá un coste temporal igual al del algoritmo MergeSort en su totalidad.
- Coste espacial: el espacio auxiliar requerido, que es el vector auxiliar es de O(n), pues es del mismo tamaño que el vector original utilizado para las fusiones.
  Su espacio de recursión, o lo que es lo mismo, la profundidad de la pila de llamadas recursivas es de O(log n).
  El filtrado de elementos únicos, como ya hemos mencionado, pertenece a O(n) en el peor de los casos, ya que si todos los elementos son únicos, se almacenaran en un vector de igual tamaño al original.

La implementación de este algoritmo la podrá encontrar, o bien en este enlace

Implementación Algoritmo propio o bien en el fichero adjunto.

Si deseásemos encontrar todos los elementos que aparecen exactamente k veces, donde k es un parámetro de entrada, podemos decir que quizás sea un poco menos eficiente en cuanto a coste espacial, pues, además de tener que realizar todo lo que ya hacía antes, ahora deberá comprobar que para cada elemento, tiene un numero de ocurrencias igual al parámetro de entrada k, lo cual puede llegar a ser un problema de eficiencia para tamaños de entrada grandes debido a esas comparaciones extras necesitadas.

En conclusión El algoritmo adaptado conserva la misma eficiencia teórica O(n*log n) y es igualmente robusto para cualquier valor de k. La modificación principal está en la lógica del filtrado, no en la complejidad computacional.

6.Información de Interés .

6.1. Pseudocodigos utilizados.

6.1.1. Pseudocodigos Busquedas Binarias.

Para las versiones recursivas, han sido utilizados los pseudocodigos proporcionados por el profesorado en el pdf de instrucciones, los cuales son:

- Versión clásica:

```
BUSQUEDABINARIA1(A[izq..der], x):
 if izq > der
       return -1
else

medio ← ⌊(izq + der)/2⌋

if A[medio] = x

return medio else if A[medio] > x

return BUSQUEDABINARIA1(A[izq..medio-1], x)

 else

return BUSQUEDABINARIA1(A[medio+1..der], x).
```

- Versión para primera ocurrencia:

```
BUSQUEDABINARIA2(A[izq..der], x):
 if izq > der
       return -1
        else
       medio ← ⌊(izq + der)/2⌋
        if A[medio] = x AND (medio = izq OR A[medio-1] ≠ x)
              return medio
              else if A[medio] >= x
               return BUSQUEDABINARIA2(A[izq..medio-1], x)
               else
              return BUSQUEDABINARIA2(A[medio+1..der], x)
```

- *Versión con interpolaciones:*

```
BUSQUEDABINARIAINTERPOLACION(A[izq..der], x):
 if izq > der OR x < A[izq] OR x > A[der]
       return -1
        else // La posición estimada si los datos estuvieran uniformemente
        // distribuidos
        pos ← izq + ⌊((x - A[izq]) * (der - izq)) / (A[der] - A[izq])⌋
        if pos < izq OR pos > der  // Por si acaso hay problemas numéricos
        pos ← ⌊(izq + der)/2⌋
        if A[pos] = x
       return pos
       else if A[pos] > x
       return BUSQUEDABINARIAINTERPOLACION(A[izq..pos-1], x)
        else
       return BUSQUEDABINARIAINTERPOLACION(A[pos+1..der], x)
```

- *Versión clásica iterativa:*

```
FUNCIÓN BusquedaBinariaIterativa1(A, n, x):
    izq = 0
    der = n - 1

    MIENTRAS izq <= der HACER:
        medio = (izq + der) / 2

        SI A[medio] == x ENTONCES:
            DEVOLVER medio
        SINO SI A[medio] > x ENTONCES:
            der = medio - 1
        SINO:
            izq = medio + 1
        FIN SI
    FIN MIENTRAS

    DEVOLVER -1
FIN FUNCIÓN
```

- *Versión primera ocurrencia:*

```
FUNCIÓN BusquedaBinariaIterativa2(A, n, x):
    izq = 0
    der = n - 1

    MIENTRAS izq <= der HACER:
        medio = (izq + der) / 2

        SI A[medio] == x Y (medio == izq O A[medio - 1] != x) ENTONCES:
            DEVOLVER medio
        SINO SI A[medio] >= x ENTONCES:
            der = medio - 1
        SINO:
            izq = medio + 1
        FIN SI
    FIN MIENTRAS

    DEVOLVER -1
FIN FUNCIÓN
```

- *Versión con interpolación:*

```
FUNCIÓN BusquedaBinariaIterativa3(A, n, x):
    izq = 0
    der = n - 1

    MIENTRAS izq <= der HACER:
        // Evitar división por cero si todos los elementos son iguales
        SI A[der] == A[izq] ENTONCES:
            pos = (izq + der) / 2
        SINO:
            pos = izq + ((x - A[izq]) * (der - izq)) / (A[der] - A[izq])
        FIN SI

        // Asegurar que pos esté dentro de los límites
        SI pos < izq O pos > der ENTONCES:
            pos = (izq + der) / 2
        FIN SI

        SI A[pos] == x ENTONCES:
            DEVOLVER pos
        SINO SI A[pos] > x ENTONCES:
            der = pos - 1
        SINO:
            izq = pos + 1
        FIN SI
    FIN MIENTRAS

    DEVOLVER -1
```

```
FIN FUNCIÓN
```

6.1.2. Pseudocodigos MergeSorts.

- Versión estándar:

```
MERGESORT(A[izq..der]):
if izq < der
medio ← ⌊(izq + der)/2⌋
 MERGESORT(A[izq..medio])
MERGESORT(A[medio+1..der])
MERGE(A, izq, medio, der)
```

- Versión hibrida:

```
MERGESORTHIBRIDO(A[izq..der], k):
 if (der - izq + 1) <= k
INSERTIONSORT(A[izq..der])
else if izq < der
medio ← ⌊(izq + der)/2⌋
 MERGESORTHIBRIDO(A[izq..medio], k) MERGESORTHIBRIDO(A[medio+1..der], k)
MERGE(A, izq, medio, der)
```

- Version que crea un solo array auxiliar:

```
FUNCIÓN Merge2(A, izq, medio, der, Aux):
    PARA i = izq HASTA der HACER:
        Aux[i] = A[i]
    FIN PARA

    i = izq       // Índice para la mitad izquierda (Aux[izq..medio])
    j = medio + 1  // Índice para la mitad derecha (Aux[medio+1..der])
    k = izq       // Índice para A

    MIENTRAS i <= medio Y j <= der HACER:
        SI Aux[i] <= Aux[j] ENTONCES:
            A[k] = Aux[i]
            i = i + 1
        SINO:
            A[k] = Aux[j]
            j = j + 1
        FIN SI
        k = k + 1
    FIN MIENTRAS

    MIENTRAS i <= medio HACER:
        A[k] = Aux[i]
        i = i + 1
        k = k + 1
    FIN MIENTRAS
FIN FUNCIÓN
```

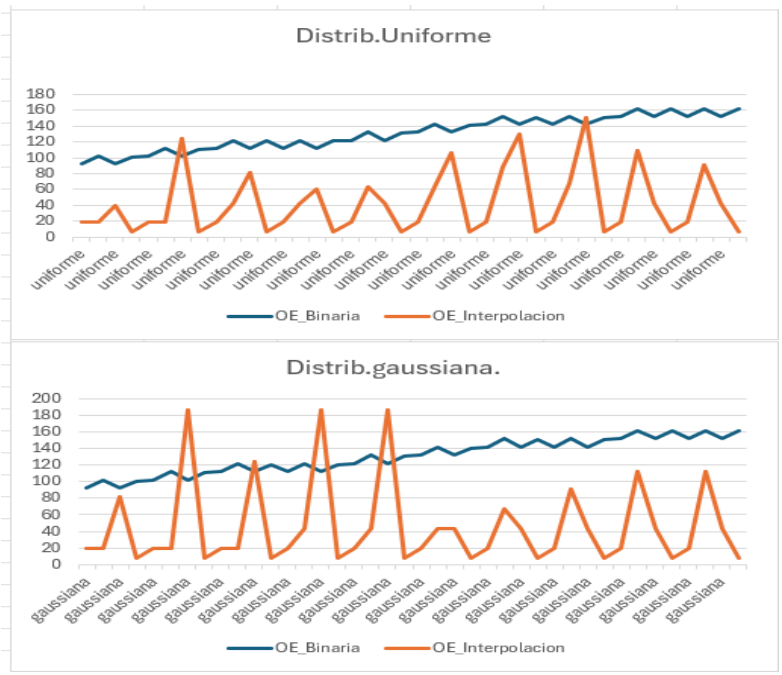
6.1.3. Pseudocodigo Algoritmo Practico.

```
FUNCIÓN Mialgoritmo(T, k):

    n = longitud(T)

    Aux = vector de tamaño n  // Array auxiliar para MergeSort


    // Paso 1: Ordenar con MergeSort

    MergesortEstandar(T, 0, n - 1, Aux)


    // Paso 2: Filtrar elementos que aparecen 'k' veces

    elementos_k = filtrarPorRepeticiones(T, k)
```

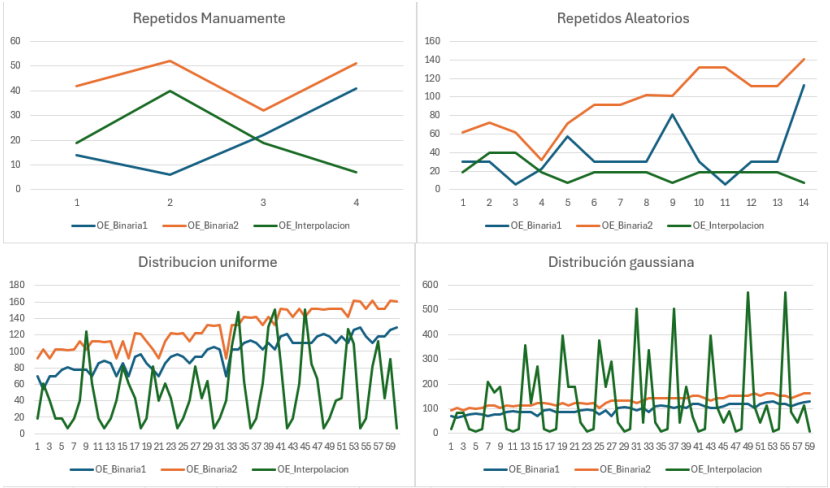
```
    // Mostrar resultados
    IMPRIMIR "Elementos que aparecen " + k + " veces (ordenados): "
    PARA cada num EN elementos_k HACER:
        IMPRIMIR num + " "
    FIN PARA
    IMPRIMIR salto_de_línea
FIN FUNCIÓN
```

6.2. Gráficas para demostración de estudios empíricos.

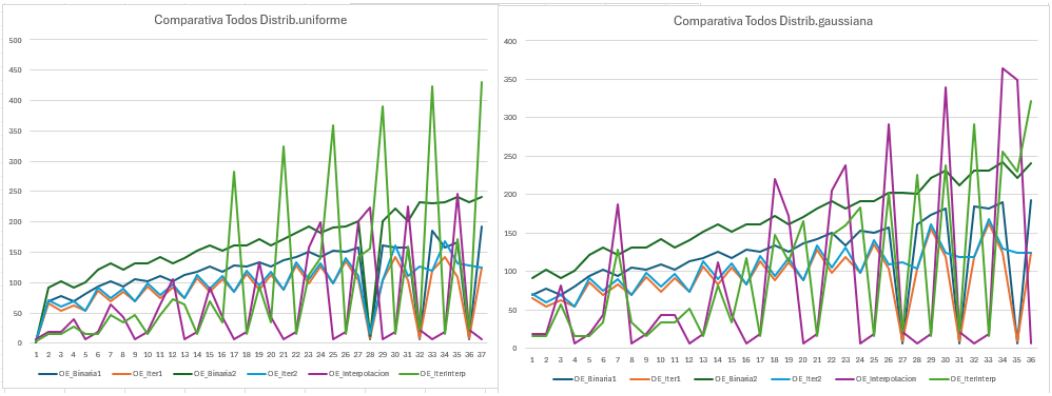
Gráfica utilizada para la comparación de la versión por primera ocurrencia e interpolación:



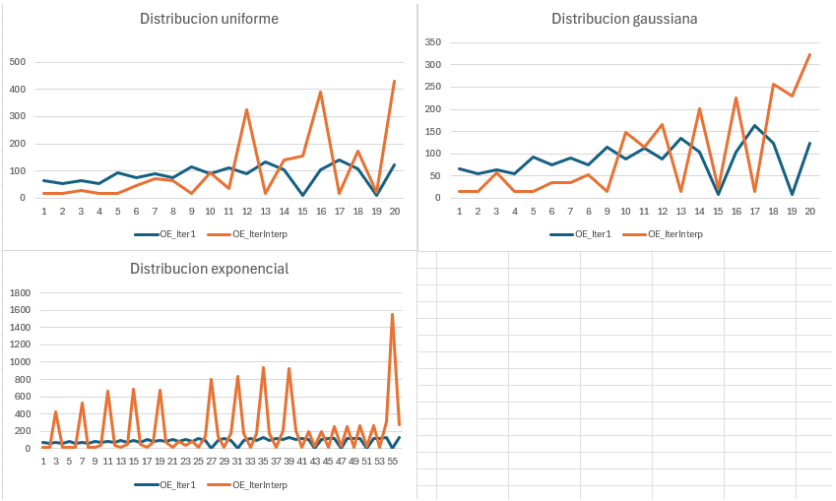
Gráfica utilizada para el estudio empírico de las 3 versiones binarias:



Gráfica Utilizada para la comparación de las versiones iterativas y binarias:



Gráfica utilizada para comparar iterativo clásico con interpolación:



6.3. Ejemplos de ejecuciones algoritmos MergeSort.

Versión estándar:

```
Array original: 3 7 9 2 5 8
Array ordenado: 2 3 5 7 8 9
```

Version hibrida:

```
Tama├▒o del array: 100
k       Tiempo promedio (ms)
5       0
10      0
15      0
20      0
25      0
30      0
35      0
40      0
45      0
50      0

Tama├▒o del array: 1000
k       Tiempo promedio (ms)
5       0
10      0
15      0
20      0
25      0
30      0
35      0
40      0
45      0
50      0

Tama├▒o del array: 10000
k       Tiempo promedio (ms)
5       4.4
10      3.2
15      3.2
20      3
25      3
30      3
35      3
40      3
45      3
50      3

Tama├▒o del array: 100000
k       Tiempo promedio (ms)
5       64.4
10      51.6
15      44.6
20      42.4
25      43
30      42
35      40.6
40      40.8
```

Comparación:

```
Comparaci├│n de MergeSort est├índar vs. optimizado
-------------------------------------------------

Tama├▒o del array: 10
MergeSort est├índar: 0 ms
MergeSort optimizado: 0 ms
Diferencia: 0 ms

Tama├▒o del array: 50
MergeSort est├índar: 0 ms
MergeSort optimizado: 0 ms
Diferencia: 0 ms

Tama├▒o del array: 100
MergeSort est├índar: 0 ms
MergeSort optimizado: 0 ms
Diferencia: 0 ms

Tama├▒o del array: 500
MergeSort est├índar: 0 ms
MergeSort optimizado: 0 ms
Diferencia: 0 ms

Tama├▒o del array: 1000
MergeSort est├índar: 0 ms
MergeSort optimizado: 0 ms
Diferencia: 0 ms

Tama├▒o del array: 2500
MergeSort est├índar: 2 ms
MergeSort optimizado: 0.2 ms
Diferencia: 1.8 ms

Tama├▒o del array: 5000
MergeSort est├índar: 5 ms
MergeSort optimizado: 1 ms
Diferencia: 4 ms

Tama├▒o del array: 7500
MergeSort est├índar: 7.2 ms
MergeSort optimizado: 2.2 ms
Diferencia: 5 ms

Tama├▒o del array: 10000
MergeSort est├índar: 10 ms
MergeSort optimizado: 3.2 ms
Diferencia: 6.8 ms

Tama├▒o del array: 12500
MergeSort est├índar: 12.4 ms
```

6.4. Ejemplo de ejecución Algoritmo Practico.

Versión original:

```
Vector original: 4 2 2 8 5 1 2 4
Elementos ├╢nicos ordenados: 1 5 8
Vector original: 3 1 2
Elementos ├╢nicos ordenados: 1 2 3
Vector original: 1 1 3 4 3
Elementos ├╢nicos ordenados: 4
```

Versión para K ocurrencias:

```
Vector original: 4 2 2 8 5 1 2 4
Elementos que aparecen 2 veces (ordenados): 4
Vector original: 3 1 2 3 3 4 4
Elementos que aparecen 1 veces (ordenados): 1 2
Vector original: 1 1 1 2 2 3 3 3 3
Elementos que aparecen 3 veces (ordenados): 1
```

6.5. Enlaces de interés.

Búsqueda Binaria:

Comparación de las 3 versiones.

Implementación para verificar estudio empírico.

Implementación versión iterativa.

Comparación iterativas/recursivas.

Comparación versión 2 y 3.

Mergesort:

Implementación versión estándar.

Implementación MergeSort hibrido.

Implementación para comparación de versiones.

Implementación Algoritmo propio