Práctica 4. Divide y Vencerás: Algoritmos de Búsqueda y Ordenación Avanzados
GUIÓN DE LA PRÁCTICA

1.- Objetivos de la práctica

- Implementar y analizar algoritmos de búsqueda y ordenación basados en la estrategia «divide y vencerás»
- Profundizar en el análisis de algoritmos recursivos y sus variantes iterativas
- Realizar estudios empíricos comparativos de eficiencia algorítmica
- Optimizar algoritmos clásicos para casos específicos

2.- Actividades a realizar

BUSQUEDA BINARIA CON VARIANTES

La búsqueda binaria es un algoritmo eficiente para encontrar un elemento en un array ordenado. Consideremos tres variantes:

Variante 1: Búsqueda binaria recursiva estándar

```
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)
```

Variante 2: Búsqueda binaria con índice de 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)
```

Variante 3: Búsqueda binaria con interpolación

```
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)
```

Actividad 1: Análisis teórico

1.1. Analiza el comportamiento de las tres variantes de búsqueda binaria.

A partir de la expresión del algoritmo, se aplicarán las reglas conocidas para contar el número de operaciones que realiza un algoritmo. Este valor será expresado como una función de T(n) que dará el número de operaciones requeridas para un caso concreto del problema caracterizado por tener un tamaño n. El análisis lo realizaremos para los casos mejor, peor y medio.

Para cada una de las variantes de la búsqueda binaria propuestas, determina:

- Coste temporal en el mejor caso (con la entrada específica que lo provoca)
- Coste temporal en el peor caso (con la entrada específica que lo provoca)
- Coste temporal en el caso promedio

1.2. Compara específicamente BÚSQUEDABINARIA2 y BÚSQUEDABINARIAINTERPOLACIÓN:

- ¿En qué situaciones BÚSQUEDABINARIAINTERPOLACIÓN sería significativamente más eficiente?
- ¿En qué situaciones podría tener un rendimiento peor?
- ¿Qué consideraciones numéricas hay que tener en cuenta en BÚSQUEDABINARIAINTERPOLACIÓN?

Actividad 2: Implementación y análisis empírico

2.1. Implementa en C++ las tres variantes de búsqueda binaria.

2.2. Diseña y ejecuta un conjunto exhaustivo de pruebas que verifiquen la corrección de tus implementaciones.

Se trata de llevar a cabo un estudio puramente empírico, es decir, estudiar experimentalmente el comportamiento de cada algoritmo. Para ello mediremos los recursos empleados (tiempo/OE) para cada tamaño dado de las entradas.

Debes cubrir al menos:

- 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

2.3. Explica claramente cómo has verificado que tu implementación de BÚSQUEDABINARIA2 siempre devuelve la primera ocurrencia de un elemento, incluso cuando aparece múltiples veces.

Actividad 3: Comparación y análisis

3.1. Implementa una versión iterativa de cada una de las tres variantes.

3.2. Realiza un estudio empírico comparando el rendimiento de las versiones recursivas e iterativas para cada variante, utilizando arrays de diferentes tamaños (desde 10^3 hasta 10^7 elementos). Presenta tus resultados en forma de gráficas.

3.3. Para la versión BÚSQUEDABINARIAINTERPOLACIÓN, compara su rendimiento con BÚSQUEDABINARIA1 utilizando:

- Arrays con valores uniformemente distribuidos
- Arrays con valores exponencialmente distribuidos (ej. 2^i)
- Arrays con valores que sigan otras distribuciones (por ejemplo, normal)

3.4. Analiza las diferencias de rendimiento y explica si los resultados empíricos concuerdan con tus predicciones teóricas.

ORDENACIÓN POR MEZCLA (MERGE SORT) Y SUS VARIANTES

Merge Sort 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)
```

Actividad 4: Optimizaciones de Merge Sort

4.1. Diseña e implementa la función MERGE para combinar dos subarrays ordenados.

4.2. Implementa una versión optimizada de Merge Sort que utilice ordenación por inserción para subarrays pequeños (de tamaño ≤ k, donde k es un parámetro a determinar):

```
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)
```

4.3. Diseña un experimento para determinar el valor óptimo de k para diferentes tamaños de array. Presenta tus resultados y conclusiones.

4.4. Implementa una variante de Merge Sort que evite la creación de subarrays temporales en la función MERGE, utilizando un único array auxiliar durante todo el proceso. Compara su rendimiento con la implementación estándar.

Actividad 5: Aplicación práctica

Supongamos que tenemos que resolver el siguiente problema:

Dado un vector A de n enteros, donde algunos elementos podrían repetirse, necesitamos encontrar todos los elementos que aparecen exactamente una vez en el array, y devolverlos ordenados de menor a mayor.

5.1. Diseña un algoritmo que resuelva este problema utilizando alguna variante de los algoritmos estudiados (búsqueda binaria y/o merge sort). El algoritmo debe tener un coste temporal mejor que $O(n^2)$ en el peor caso.

5.2. Analiza el coste temporal y espacial de tu algoritmo.

5.3. Implementa tu solución en C++ y verifica su ejecución con varios ejemplos.

5.4. ¿Cómo cambiaría tu solución si ahora necesitáramos encontrar todos los elementos que aparecen exactamente k veces, donde k es un parámetro de entrada?

3.- Entrega

- La fecha de entrega será la del día correspondiente al turno de práctica (L1-L9) de la **semana del 28 de abril al 2 de mayo**. Por tanto, tendremos 2 sesiones presenciales para la realización de esta práctica (semana 7-11 abril y semana 21-25 abril), más el trabajo personal fuera del aula.
- La entrega de la práctica se realizará por Moodle en la tarea puesta al efecto y tendrá el siguiente formato:

  Crear y subir una carpeta comprimida ***Apellido1Apellido2Nombre.rar*** la cual debe contener:
  - Una subcarpeta con las fuentes del programa (.cpp y .h)
  - Una subcarpeta con la memoria de la práctica, de 12 páginas como máximo, en formato pdf con el esquema especificado a continuación:
    - Respuestas detalladas a todas las preguntas
    - Pseudocódigo
    - Gráficas de tiempos de ejecución claras y bien etiquetadas
    - Análisis empíricos solicitados
    - Análisis crítico de los resultados obtenidos
    - Conclusiones fundamentadas
  - Una subcarpeta con todo el material suplementario que consideres oportuno (tablas de datos, hojas de cálculos, gráficas, etc…)