PRACTICA 1 ES-DAT

VICTOR GONZALEZ DEL CAMPO- JUAN GARCIA LOPEZ

CONTENIDO

ENUNCIADO PRACTICA

Deseamos localizar el mismo subconjunto de objetos localizados en posiciones contiguas dentro de dos vectores diferentes de objetos. No se debe confundir con el problema de las diferentes sub-secuencias comunes a dos colecciones de objetos. En ese caso, se busca solo aquella que sea la de mayor longitud.

Por ejemplo, supongamos los vectores:

[1,2,4,5,3,4,6,7,9,4,5,33,45,63,2,4,44,32]

[4,2,1,2,4,5,3,4,6,7,9,4,5,33,45,63,2,4,55,2,9,10]

En este caso, podemos solucionarlo a simple vista y obtener como resultado:

[1,2,4,5,3,4,6,7,9,4,5,33,45,63,2,4]

Para simplificar el ejercicio, no se considerará el buscar si existen más vectores solución con el mismo tamaño que la solución aportada: solamente hay que devolver la última secuencia de mayor tamaño encontrada en ambos vectores.

OBJETIVOS

Se espera que el alumno se familiarice con el concepto de complejidad computacional1.

Para ello, el ejercicio consiste en su análisis y cálculo teórico de algoritmos de ordenación, así como su comprobación práctica del consumo de recursos.

Se comprobará como, a pesar de poder darse diferentes algoritmos que solucionan correctamente un mismo problema, esta característica de los algoritmos hace claramente ventajosa a unas soluciones frente a otras.

Este conocimiento se aplicará a lo largo del resto de la asignatura, permitiendo comparar las ventajas de las diferentes Estructuras de Datos.

Adicionalmente se trabajará con la Genericidad de Java, que permite definir los tipos de datos a almacenar en una colección.

Es una característica muy importante que mantienen todos los componentes del Collections Framework.

DINAMICA

Se pretende que el alumno se familiarice con el concepto de Eficiencia Algorítmica 1 implementando diferentes métodos de consulta y procesamiento sobre conjuntos de datos de diferentes tamaños.

El objetivo final del ejercicio es poder completar un informe en el que se incluyan los resultados de los experimentos realizados por el alumno, así como las conclusiones que se alcanzan a la vista de los mismos.

Por la naturaleza de este ejercicio, se debe tener en cuenta que en esta práctica se valorará de forma especial el informe presentado (además del código fuente del programa).

ALGORITMOS

APROXIMACION ITERATIVA

DESCRIPCIÓN

El algoritmo es una implementación para encontrar el subarray común más largo entre dos secuencias de elementos de manera iterativa. Aquí está el funcionamiento del algoritmo:

1. Se inicializan algunas variables, como una lista para almacenar el resultado (**R**), la longitud máxima del subarray común (**maxLongitud**), y variables para controlar los índices y tamaños de las secuencias.
2. Se inicia un bucle externo que recorre la primera secuencia (**secuencia1**).
3. Dentro del bucle externo, hay otro bucle que itera sobre las longitudes posibles del subarray común (**tamProbado**), comenzando desde 1 hasta la longitud restante de la secuencia.
4. Dentro de estos bucles, hay otro bucle que itera sobre la segunda secuencia (**secuencia2**), buscando coincidencias con la subsecuencia de la primera secuencia.
5. Si se encuentra una subsecuencia común, se compara su longitud con la longitud máxima actual (**maxLongitud**). Si es mayor, se actualiza **R** con la nueva subsecuencia y se actualiza **maxLongitud** con la longitud actual de la subsecuencia común.
6. Finalmente, se devuelve la lista **R**, que contiene el subarray común más largo encontrado entre las dos secuencias.

CODIGO

```
/**
 * Encuentra el subarray común más largo entre dos secuencias de
elementos de manera iterativa.
 *
 * @return el subarray común más largo encontrado entre las dos
secuencias
 */
public List<T> metodoIterativo() {
   List<T> R = new ArrayList<>();
   int maxLongitud = 0;
```

```
   int inicioSecuencia1 = 0;
   int inicioSecuencia2 = 0;
   int tamProbado = 0;
   int tamSecuencia1 = secuencia1.size();
   int tamSecuencia2 = secuencia2.size();

   for (inicioSecuencia1 = 0; inicioSecuencia1 < tamSecuencia1;
inicioSecuencia1++) {
      for (tamProbado = 1; tamProbado <= (tamSecuencia1 -
inicioSecuencia1); tamProbado++) {
         for (inicioSecuencia2 = 0; inicioSecuencia2 <= (tamSecuencia2
- tamProbado); inicioSecuencia2++) {
            if (secuencia1.subList(inicioSecuencia1, inicioSecuencia1
+ tamProbado).equals(secuencia2.subList(inicioSecuencia2,
inicioSecuencia2 + tamProbado))) {
               if (maxLongitud < tamProbado) {
                  R = new
ArrayList<>(secuencia1.subList(inicioSecuencia1, inicioSecuencia1 +
tamProbado));
                  maxLongitud = tamProbado;
               }
            }
         }
      }
   }
   return R;
}
```

ANALISIS COMPLEJIDAD

El primer bucle externo itera sobre la primera secuencia, lo que representa una complejidad de O(n), donde n es la longitud de la primera secuencia.

El segundo bucle interno itera sobre las longitudes posibles del subarray común, que va desde 1 hasta n (la longitud de la primera secuencia).

El tercer bucle interno itera sobre la segunda secuencia, lo que representa una complejidad de O(m), donde m es la longitud de la segunda secuencia.

Dentro de este tercer bucle, se realiza una comparación de subsecuencias de longitud m, lo que también añade una complejidad de O(m).

Dado que cada bucle está anidado, la complejidad total del algoritmo es el producto de las complejidades individuales de cada bucle, como nos da igual que sea O(n) o O(m),el resultando en O(n^3)

PROGRAMACION DINAMICA

DESCRIPCIÓN

El código implementa un algoritmo de programación dinámica para encontrar la subsecuencia común más larga entre dos secuencias.

Vamos a desglosar el funcionamiento del código:

1. Se declara un método llamado **metodoProgDinamica()** que devuelve una lista de elementos de tipo **T**.
2. Se inicializa una lista llamada **resultado** que contendrá la subsecuencia común más larga encontrada.
3. Se obtienen las longitudes de las dos secuencias **secuencia1** y **secuencia2** mediante **secuencia1.size()** y **secuencia2.size()** respectivamente.
4. Se crea una matriz de dimensiones **(n + 1) x (m + 1)**, donde **n** es la longitud de **secuencia1** y **m** es la longitud de **secuencia2**. Esta matriz se utiliza para almacenar los resultados intermedios del algoritmo.
5. Se inicializa una variable **maxLongitud** que almacenará la longitud máxima de la subsecuencia común encontrada.

Luego viene la parte principal del algoritmo:

6. Se itera sobre cada elemento de ambas secuencias utilizando dos bucles **for** anidados.
7. Para cada par de elementos **(i, j)** de las secuencias, se verifica si **i** o **j** son igual a cero. Si alguno de ellos es cero, significa que una de las secuencias es vacía y, por lo tanto, la longitud de la subsecuencia común es cero.
8. Si los elementos **i** y **j** no son cero, se verifica si los elementos correspondientes de **secuencia1** y **secuencia2** son iguales.
9. Si los elementos son iguales, se incrementa el valor de la matriz **matrizCalculo[i][j]** con respecto al valor de la posición diagonal superior izquierda (**matrizCalculo[i - 1][j - 1]**) más uno.
10. Se actualiza la variable **maxLongitud** si se encuentra una subsecuencia común más larga.
11. Si los elementos no son iguales, se establece el valor de **matrizCalculo[i][j]** como cero, ya que no hay una subsecuencia común en esta posición.

Una vez que se completa la iteración sobre ambas secuencias, el método devuelve la lista **resultado**, que contiene la subsecuencia común más larga encontrada.

Es importante mencionar que este algoritmo tiene una complejidad temporal de O(n*m), donde **n** es la longitud de la primera secuencia y **m** es la longitud de la segunda secuencia. Esto se debe a que se realiza una exploración exhaustiva de todas las combinaciones de elementos de ambas secuencias.

CODIGO

```java
/**
 * Encuentra el subarray común más largo entre dos secuencias de
elementos utilizando programación dinámica.
 *
 *
 *
 * @return el subarray común más largo encontrado entre las dos
secuencias
 */

public List<T> metodoProgDinamica() {
      List<T> resultado = new ArrayList<>();
      int n = secuencia1.size();
      int m = secuencia2.size();
      int[][] matrizCalculo = new int[n + 1][m + 1];
      int maxLongitud = 0;

      for (int i = 0; i <= n; i++) {
         for (int j = 0; j <= m; j++) {
            if (i == 0 || j == 0) {
               matrizCalculo[i][j] = 0;
            } else if (secuencia1.get(i - 1).equals(secuencia2.get(j -
1))) {
               matrizCalculo[i][j] = matrizCalculo[i - 1][j - 1] + 1;
               if (matrizCalculo[i][j] > maxLongitud) {
                  maxLongitud = matrizCalculo[i][j];
                  resultado = new ArrayList<>(secuencia1.subList(i -
maxLongitud, i));
               }
            } else {
               matrizCalculo[i][j] = 0;
            }
         }
      }
      return resultado;
   }
}
```

ANALISIS COMPLEJIDAD

El algoritmo que has proporcionado implementa una solución utilizando programación dinámica para encontrar la longitud de la subsecuencia común más larga entre dos secuencias.

La complejidad temporal del algoritmo se puede analizar de la siguiente manera:

- La matriz **matrizCalculo** tiene dimensiones **(n + 1) x (m + 1)**, donde **n** es la longitud de la primera secuencia (**secuencia1**) y **m** es la longitud de la segunda secuencia (**secuencia2**).
- El algoritmo itera sobre cada elemento de ambas secuencias utilizando dos bucles **for** anidados. Estos bucles recorren la matriz **matrizCalculo** y tienen una complejidad de tiempo de O(n^2).

- Dentro de los bucles anidados, las operaciones realizadas son simples asignaciones y comparaciones, lo cual tiene una complejidad constante O(1).

Por lo tanto, la complejidad temporal total del algoritmo es O(n^2), donde **n** es la longitud de la secuencia de entrada.

Es importante destacar que la complejidad del algoritmo puede ser eficiente para entradas de tamaño razonable, pero puede volverse costosa para secuencias muy largas debido al cuadrado del tamaño de las secuencias. Sin embargo, la programación dinámica permite evitar recalcular subproblemas, lo que optimiza significativamente el rendimiento en comparación con un enfoque de fuerza bruta.

EXPERIMENTOS

TIEMPOS DE EJECUCION

Los experimentos se han llevado a cabo con las siguientes especificaciones:

Procesador	Intel(R) Core(TM) i5-7300U CPU @ 2.60GHz 2.71 GHz
RAM instalada	16,0 GB (15,7 GB usable)

Con sistema operativo :

Especificaciones de Windows

Edición	Windows 10 Pro
Versión	22H2
Instalado el	05/10/2022
Compilación del sistema operativo	19045.3803
Experiencia	Windows Feature Experience Pack 1000.19053.1000.0

Aquí se muestran los resultados de los tiempo de ejecución:

Midiendo tiempo para un array de longitud : 10

Tiempo del método iterativo: 102 microsegundos

Tiempo del método de programación dinámica: 157 microsegundos

Midiendo tiempo para un array de longitud : 50

Tiempo del método iterativo: 686 microsegundos

Tiempo del método de programación dinámica: 533 microsegundos

Midiendo tiempo para un array de longitud : 100

Tiempo del método iterativo: 116 milisegundos

Tiempo del método de programación dinámica: 2 milisegundos

Midiendo tiempo para un array de longitud : 300

Tiempo del método iterativo: 378 milisegundos

Tiempo del método de programación dinámica: 9 milisegundos

Midiendo tiempo para un array de longitud : 600

Tiempo del método iterativo: 2589 milisegundos

Tiempo del método de programación dinámica: 27 milisegundos

Midiendo tiempo para un array de longitud : 1000

Tiempo del método iterativo: 12231 milisegundos

Tiempo del método de programación dinámica: 25 milisegundos

Midiendo tiempo para un array de longitud : 2000

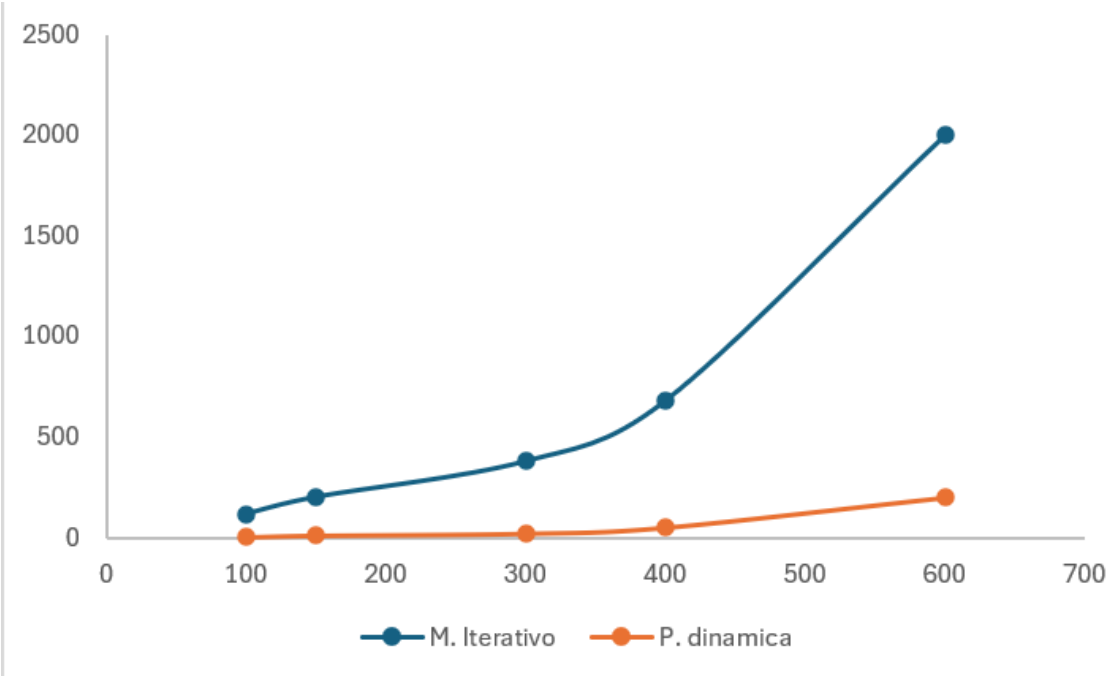
Tiempo del método iterativo: 106951 milisegundos

Tiempo del método de programación dinámica: 62 milisegundos

Midiendo tiempo para un array de longitud : 3000

Tiempo del método iterativo: 245712 milisegundos

Tiempo del método de programación dinámica: 56 milisegundos



Todos estos tiempos se pueden medir ejecutando el main de la clase Comprobacion_tiempos.java

CONCLUSION

En conclusión, los dos algoritmos implementados, la aproximación iterativa y la solución basada en programación dinámica, abordan el problema de encontrar el subarray común más largo entre dos secuencias de elementos de manera efectiva, aunque con enfoques diferentes.

La aproximación iterativa funciona mediante un enfoque de fuerza bruta, explorando exhaustivamente todas las posibles subsecuencias entre las dos secuencias de entrada. Aunque es simple de implementar, su complejidad temporal es cuadrática, lo que significa que su rendimiento puede degradarse rápidamente a medida que aumenta el tamaño de las secuencias. Este algoritmo es adecuado para conjuntos de datos pequeños, pero puede volverse ineficiente para entradas más grandes.

Por otro lado, la solución basada en programación dinámica utiliza una matriz para almacenar los resultados intermedios y evitar recalcular subproblemas, lo que conduce a una complejidad temporal más eficiente de O(n^2), donde n es la longitud de las dos secuencias de entrada. Aunque su implementación puede ser más compleja, especialmente en términos de la gestión de la matriz y la lógica de programación dinámica, su rendimiento es notablemente mejor en comparación con la aproximación iterativa para conjuntos de datos de tamaño considerable.

En general, la elección del algoritmo depende del contexto específico y de las restricciones de rendimiento. Para conjuntos de datos pequeños o cuando la simplicidad es prioritaria, la aproximación iterativa puede ser adecuada. Sin embargo, para conjuntos de datos más grandes donde se requiere una eficiencia significativa, la solución basada en programación dinámica es preferible debido a su menor complejidad temporal y mejor rendimiento en general.