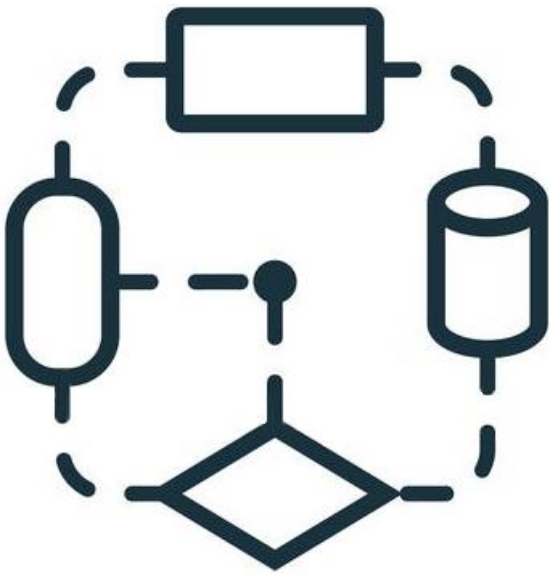
Esquema Divide-and-Conquer

PRÁCTICA 1



Autor: Pablo Gómez Rivas y Jesús Fornieles Muñoz

Materia: Estructura de Datos y Algoritmos II

Grupo de Prácticas: GTAB2

Fecha: Viernes, 31 de marzo de 2023

Índice

Objetivo **3**
Antecedentes **4**
Estudio de la Implementación **5**
Iterativo orden N al cubo: 5
Iterativo orden N al cuadrado: 5
Iterativo orden N: 5
Recursivo con DyV orden N * Log(N): 5
Recursivo con DyV orden N: 6
Estudio Teórico **6**
Algoritmo Iterativo O(n3) 6
Algoritmo Iterativo O(n2) 7
Algoritmo Iterativo O(n) 8
Algoritmo DyV O(n log n) 9
Algoritmo DyV O(n) 9
Estudio Experimental **10**
Algoritmo Iterativo O(n3) 10
Algoritmo Iterativo O(n2) 11
Algoritmo Iterativo O(n) 13
Algoritmo DyV O(n log n) 14
Algoritmo DyV Compare O(n) 15
Algoritmo DyV Linear O(n) 17
Anexo **18**
Diagrama de clases: 18
Archivos fuente: 19
Fuentes Bibliográficas **20**

Objetivo

El método divide y vencerás es una técnica comúnmente utilizada en la informática para resolver una amplia variedad de problemas. A continuación, se detallan algunos de los usos más comunes de esta técnica en la informática:

Ordenación de arreglos: El algoritmo Quicksort es un ejemplo común de cómo el método divide y vencerás se puede utilizar para ordenar un arreglo. El arreglo se divide en subarreglos más pequeños, se ordenan de forma recursiva y luego se combinan en el arreglo final.

Búsqueda binaria: El método divide y vencerás también se puede utilizar para realizar búsquedas en una lista o arreglo ordenado. El arreglo se divide en dos mitades y se busca en la mitad correspondiente al elemento deseado. Este proceso se repite de forma recursiva hasta que se encuentra el elemento.

Multiplicación de matrices: La multiplicación de matrices es un problema común en la informática y se puede resolver mediante el método divide y vencerás. Las matrices se dividen en submatrices más pequeñas y se multiplican de forma recursiva hasta que se obtiene la matriz final.

Árboles de búsqueda: El método divide y vencerás se puede utilizar para construir y buscar árboles de búsqueda binarios. Cada nodo se divide en dos subárboles más pequeños, que se manejan de forma recursiva.

Transformada rápida de Fourier (FFT): La FFT es una técnica matemática comúnmente utilizada en la señalización digital y se puede resolver mediante el método divide y vencerás. La señal se divide en subseñales más pequeñas y se procesan de forma recursiva.

En general, el método divide y vencerás es una técnica útil para resolver problemas complejos en informática, ya que puede reducir significativamente la complejidad del problema y mejorar el tiempo de ejecución. Sin embargo, se debe tener cuidado al utilizar esta técnica, ya que la recursión puede aumentar la complejidad y el uso de memoria.

Por otro lado, en lo que respecta a este trabajo, hemos decidido hacer a Jesús Fornieles Muñoz líder de esta práctica, encargado de supervisar y verificar la calidad y el correcto funcionamiento de los códigos del trabajo del resto de miembros, Pablo Gomez Rivas en este caso, de forma que la elaboración del proyecto ha quedado de la siguiente forma:

#	Tipo	Tarea		Peso	
1	Implementacion	Algoritmo iterativo de coste O(n^3)	Pablo	40%	40%
2	Implementacion	Algoritmo iterativo de coste O(n^2), mejora del anterior	Jesús		
3	Implementacion	Algoritmo iterativo de coste O(n), mejora del anterior	Pablo		
4	Implementacion	Algoritmo de tipo **divide-and-conquer** de coste O(n log n)	Jesús		
5	Implementacion	Algoritmo de tipo **divide-and-conquer** de coste O(n), mejora del anterior	Pablo		
6	Implementacion	Generación de secuencuas aleatorias. Considerar casos especiales	Jesús		
7	Implementacion	Main con medidas de tiempos de ejecución	Pablo		
8	Implementacion	Juegos de prueba, considerando los casos especiales	Jesús		
9	Documentación	**Estudio de la implementación**	Pablo	10%	60%
10	Documentación	**Estudio teórico**	Jesús	15%	
11	Documentación	**Estudio experimental**	Pablo	35%	
12	Documentación	**Anexos** y bibliografia	Jesús		

Antecedentes

Desde el departamento EDASoft de la prestigiosa empresa theBestSoft, nos vemos agradecidos de aceptar a EDAland-Gran-Casino para tratar de ayudarles lo máximo posible.

Como hemos visto, el casino ha tenido días buenos y días malos a lo largo de este último enero de 2023, pero lo que podemos afirmar es que la casa siempre gana, y aunque tenga malas rachas también las tiene buenas, y son estas buenas rachas de hecho lo que más nos concierne en esta ocasión.

Nos centramos en las mejores rachas que ha tenido el gran casino durante el último mes, para ello emplearemos diversas técnicas para averiguar cuál ha sido la mayor cantidad de dinero que ha amasado nuestro casino en una serie de días seguidos.

Existen diversas formas de calcular esta cifra, pero la duda principal está entre dos métodos:

Algoritmo de Kadane: Este es una forma de calcular la subsecuencia máxima de un conjunto de números (enteros en nuestro caso) que nos ofrece una alternativa iterativa de orden lineal al problema a partir de tan solo un bucle y una serie de asignaciones y condicionales cumple a la perfección con la eficiencia que exigimos en EDASoft.

Divide y Vencerás: esta alternativa de carácter recursivo también nos ofrece un mecanismo de complejidad lineal para el cálculo de subsecuencias máximas a partir de un único método con dos llamadas recursivas y el uso de una serie de atributos necesarios para su desarrollo, sin necesidad de bucles ni condicionales de ningún tipo.

Visto la eficiencia estudiada estudio experimental, optamos por la opción iterativa de Kadane, debido a que, pese a ambos ser lineales, este nos ofrece una media de tiempos menores que los de la variante recursiva.

Por otro lado, también resulta más sencillo de comprender para los trabajadores junior de la empresa.

Estudio de la Implementación

Iterativo orden N al cubo:

Este algoritmo al tener una complejidad elevada de hasta n^3 resulta sencillo de explicar.

La base de este consiste en buscar por fuerza bruta cuál será la subsecuencia máxima, probando así todas las combinaciones posibles y guardado en una variable el mayor resultado obtenido por este método.

Iterativo orden N al cuadrado:

Al igual que pasa con el ejemplo anterior, la alta complejidad nos permite probar con todas y cada una de las combinaciones posibles, guardando aquellas que superen al máximo anterior y devolviendo así el máximo alcanzado durante toda la iteración del algoritmo, y por consecuencia, la máxima subsecuencia.

Iterativo orden N:

Para este algoritmo nos basamos en el algoritmo de Kadane, en el cual recorremos nuestro array con una variable j. Mientras lo recorremos, iremos sumando los valores de las posiciones del array, y en función del resultado de esa suma planteamos dos situaciones:

- Es la suma actual mayor a la suma obtenida en alguna posición anterior del array?

Si la respuesta es sí, entonces reemplazamos el valor de nuestra suma actual por el nuevo valor obtenido, pues hemos encontrado una nueva subsecuencia máxima.

Si la respuesta es no, nos planteamos una nueva pregunta:

- Es la suma actual menor que 0?

Esto resulta de gran importancia debido a que una subsecuencia máxima jamás empezará con un número negativo. De forma que si la suma de los elementos anteriores del array ofrece un número menor que 0, significa que no es ahí donde vamos a encontrar la máxima subsecuencia.

Recursivo con DyV orden N * Log(N):

Estos algoritmos se basan en la técnica de divide y vencerás. Esta estrategia de resolución de problemas resulta muy útil en problemas de

este tipo, pues se basa en la división de un problema en varios subproblemas (en nuestro caso dos), los cuales deben ser del mismo tipo que el problema original y del mismo tamaño que el resto de los subproblemas.

El objetivo de este tipo de recursividad consiste en subdividir de nuevo los subproblemas de forma continua hasta llegar al punto en el que lleguemos a conseguir información de utilidad de cada una de las llamadas, juntándose esos pequeños fragmentos para resolver el problema principal.

Bajo esa premisa, la forma en la que opera este método será dividir el array en 2 de forma continua hasta que lleguemos a un solo número (array.length = 0, caso base). Después de eso pasaremos a coger parejas las cuales devolverán el máximo entre uno, otro o la suma de ambos (Math.Max(Math.Max(left, right) , left + right)).

Esto continuará hasta que sea sabida cual es la subsecuencia máxima de la división establecida al comienzo, tanto la izquierda como la derecha desde la mitad del array, calculando así en la última iteración el valor máximo de la subsecuencia.

Recursivo con DyV orden N:

Este algoritmo se basa en el mismo modus operandi que el anterior, a diferencia de que prescindimos de los bucles for que necesitábamos para calcular si la subsecuencia mejoraba o no. Dicha prescidencia supone la adición de unos nuevos tipos de atributos, los sufijos y los prefijos, los cuales se suman a la recursividad con el fin de cumplir con la función del cálculo de maxsum (suma máxima). Para ello observamos cómo durante cada llamada recursiva maxsum hará uso de ambos:

maxSum = Math.max(Math.max(leftSS.maxSum, rightSS.maxSum), leftSS.maxSuffix + rightSS.maxPrefix);

Consiguiendo así el mismo efecto que el algoritmo anterior, pero reduciendo su coste de forma considerable.

Estudio Teórico

Algoritmo Iterativo O(n3)

```
    public static Subsequence
maxSubsequenceSumIterativeCubic(int[] arr) {
            Subsequence maxSubsequence = new
Subsequence();
            maxSubsequence.sum = 0;
```

```
                for (int i = 0; i < arr.length; i++)
                        for (int j = i; j < arr.length; j++) {
                                int thisSum = 0;
                                for (int k = i; k <= j; k++)
                                        thisSum += arr[k];
                                if (thisSum > maxSubsequence.sum)
{
                                        maxSubsequence.sum = thisSum;
                                        maxSubsequence.start = i;
                                        maxSubsequence.end = j;
                                }
                        }
                return maxSubsequence;
        }
```

- El orden de complejidad de las líneas no marcadas es $O(1)$ ya que son operaciones elementales de tiempo constante (c y d).
- El orden de complejidad de los bucles FOR es:
- `for (int i = 0; i < arr.length; i++)` y

  `for (int j = i; j < arr.length; j++)` y

  `for (int k = i; k <= j; k++)`

  se calcula:

$$\sum_{i=1}^{n} \left(\sum_{j=i}^{n} (c + \sum_{k=i}^{j} d) \right)$$

Realizando las sumatorias queda:

$$\bar{T}(n) \leq \frac{dn^3}{6} + \frac{dn^2}{2} + \frac{cn^2}{2} + \frac{dn}{3} + \frac{cn}{2} \in \boldsymbol{O}(\mathbf{n^3})$$

Algoritmo Iterativo O(n2)

```
public static Subsequence
maxSubsequenceSumIterativeQuadratic(int[] arr) {
        Subsequence maxSubsequence = new Subsequence();
        maxSubsequence.sum = 0;

        for (int i = 0; i < arr.length; i++) {
                int thisSum = 0;
                for (int j = i; j < arr.length; j++) {
```

```
                    thisSum += arr[j];

                    if (thisSum > maxSubsequence.sum) {
                        maxSubsequence.sum = thisSum;
                        maxSubsequence.start = i;
                        maxSubsequence.end = j;
                    }
                }
            }
            return maxSubsequence;
            }
```

- El orden de complejidad de las líneas no marcadas es $O(1)$ ya que son operaciones elementales de tiempo constante (c y d).
- El orden de complejidad de los bucles FOR es:
- `for (int i = 0; i < arr.length; i++)` y

  `for (int j = i; j < arr.length; j++)`

  se calcula:

$$\sum_{i=1}^{n} (c + \sum_{j=i}^{n} (d))$$

Realizando las sumatorias queda:

$$\bar{T}(n) \leq \frac{dn^2}{2} + \frac{dn}{2} + cn \in \boldsymbol{O(n^2)}$$

Algoritmo Iterativo O(n)

```
public static Subsequence
maxSubsequenceSumIterativeLinear(int[] arr) {
        Subsequence maxSubsequence = new Subsequence();
        maxSubsequence.sum = 0;
        int thisSum = 0;

        for (int i = 0, j = 0; j < arr.length; j++) {
            thisSum += arr[j];
            if (thisSum > maxSubsequence.sum) {
                maxSubsequence.sum = thisSum;
                maxSubsequence.start = i;
                maxSubsequence.end = j;
            } else if (thisSum < 0) {
                i = j + 1;
                thisSum = 0;
```

```
            }
        }
        return maxSubsequence;
    }
```

- El orden de complejidad de las líneas no marcadas es $O(1)$ ya que son operaciones elementales de tiempo constante (c y d).
- El orden de complejidad del bucle FOR es:

```
for (int i = 0, j = 0; j < arr.length; j++)
```

se calcula:

$$\sum_{j=1}^{n} c$$

Realizando las sumatorias queda:

$$\bar{T}(\mathrm{n}) \leq \mathrm{cn} \in \boldsymbol{O}(\mathbf{n})$$

Algoritmo DyV O(n log n)

Para calcular la complejidad de un algoritmo recursivo es común acudir al Teorema Maestro. De esta forma, contabilizamos las siguientes partes:

- a = 2: Número de llamadas recursivas.
- b = 2: Tamaño de las llamadas recursivas.
- c = 1: Orden de la parte no recursiva.

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

$$c = \log_a b = \log_2 2 = 1; \boldsymbol{c = 1}$$

$$Entonces, el\ algoritmo \in O(n^c \cdot \log n) = \boldsymbol{O(n \cdot \log n)}$$

Algoritmo DyV O(n)

- a = 2: Número de llamadas recursivas.
- b = 2: Tamaño de las llamadas recursivas.
- c = 0: Orden de la parte no recursiva.

$$T(n) = 2T\left(\frac{n}{2}\right) + O(1)$$

$$c = \log_a b = \log_2 2 = 1; \boldsymbol{c < 1}$$

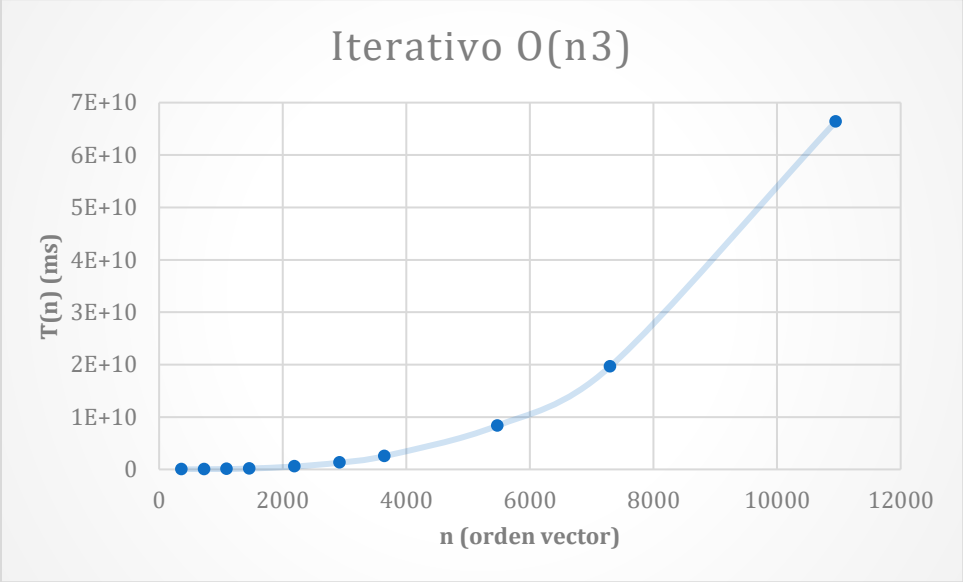
$$Entonces, el\ algoritmo \in O(n \cdot \log_a b) = O(n \cdot \log_2 2) = \boldsymbol{O(n)}$$

Estudio Experimental

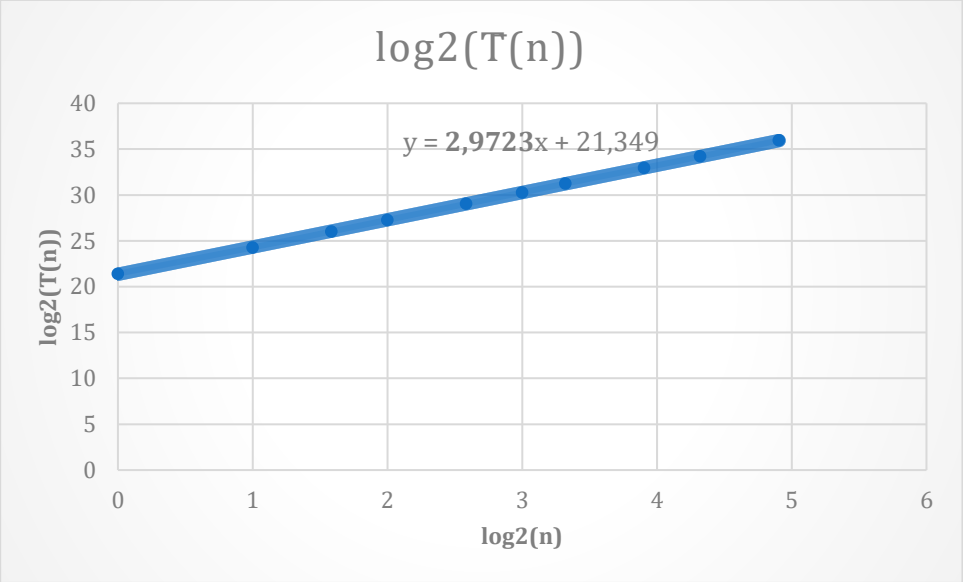
Algoritmo Iterativo O(n3)

Años	n (orden matriz)	T̄(n) (ms)	log2(n)
1	365	2793875	0
2	730	20321045	1
3	1095	68130350	1,584962501
4	1460	162758615	2
6	2190	547916995	2,584962501
8	2920	1292063605	3
10	3650	2545290485	3,321928095
15	5475	8350393170	3,906890596
20	7300	19647239805	4,321928095
30	10950	66384461555	4,906890596

Tiempo de Ejecución VS Tamaño de la Entrada



Gráfica tiempo de ejecución VS Tamaño de la entrada

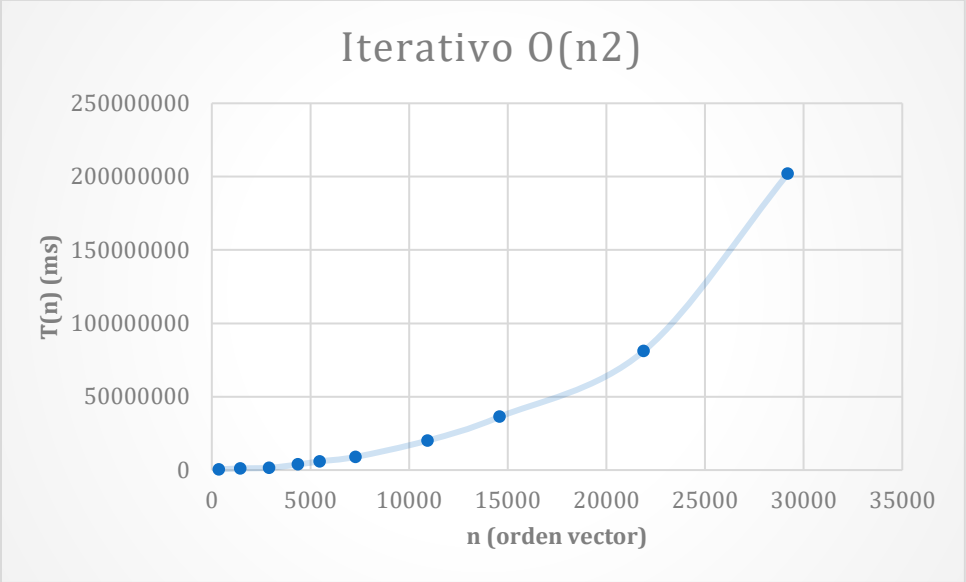


Fijándonos en la anterior gráfica, podemos ver que la pendiente de la ecuación de la recta es **k = 2,9723**. Por tanto, podemos decir que este algoritmo tiene orden de complejidad $\boldsymbol{O(n^{2,9723})}$.

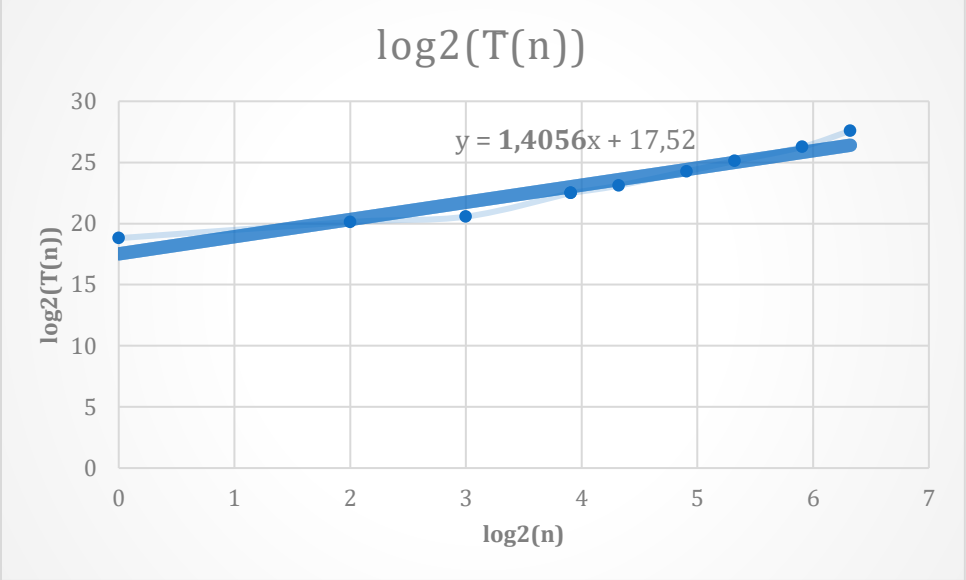
Algoritmo Iterativo O(n2)

Años	n (orden matriz)	$\bar{T}$(n) (ms)	log2(n)
1	365	457640	0
4	1460	1145635	2
8	2920	1552275	3
12	4380	3803560	3,906890596
15	5475	5946785	3,906890596
20	7300	8997820	4,321928095
30	10950	20155100	4,906890596
40	14600	36338780	5,321928095
60	21900	81147400	5,906890596
80	29200	201945845	6,321928095

Tiempo de Ejecución VS Tamaño de la Entrada



Gráfica tiempo de ejecución VS Tamaño de la entrada

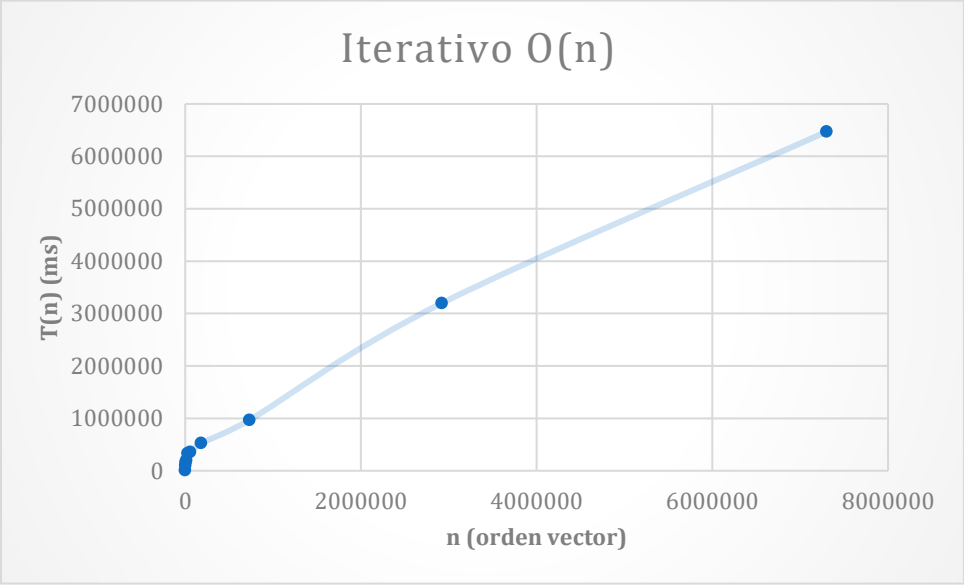


Fijándonos en la anterior gráfica, podemos ver que la pendiente de la ecuación de la recta es **k = 1,4056**. Por tanto, podemos decir que este algoritmo tiene orden de complejidad $\boldsymbol{O(n^{1,4056})}$.

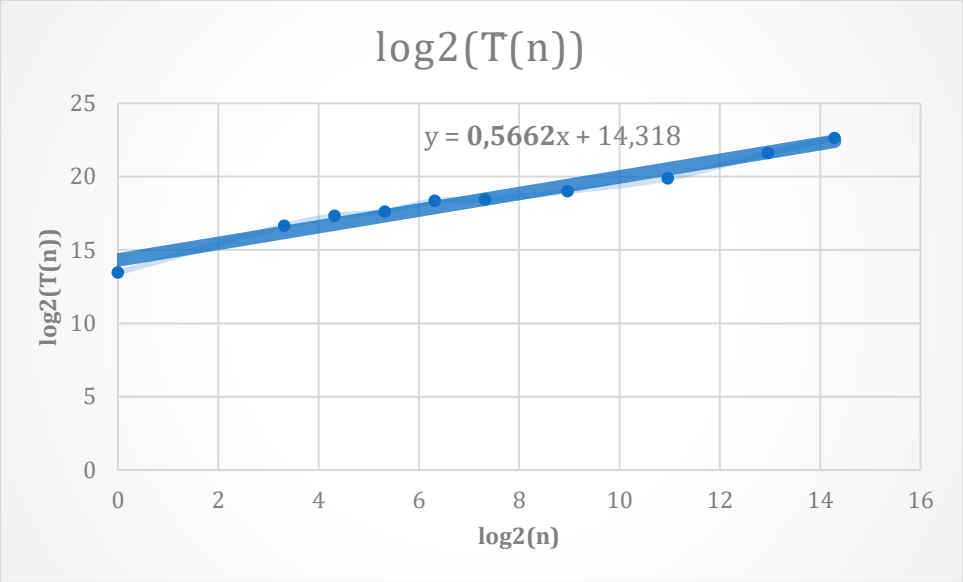
Algoritmo Iterativo O(n)

Años	n (orden matriz)	$\bar{T}(n)$ (ms)	log2(n)
1	365	11310	0
10	3650	102805	3,321928095
20	7300	163785	4,321928095
40	14600	200365	5,321928095
80	29200	334955	6,321928095
160	58400	354780	7,321928095
500	182500	525935	8,965784285
2000	730000	966855	10,96578428
8000	2920000	3197475	12,96578428
20000	7300000	6470390	14,28771238

Tiempo de Ejecución VS Tamaño de la Entrada



Gráfica tiempo de ejecución VS Tamaño de la entrada

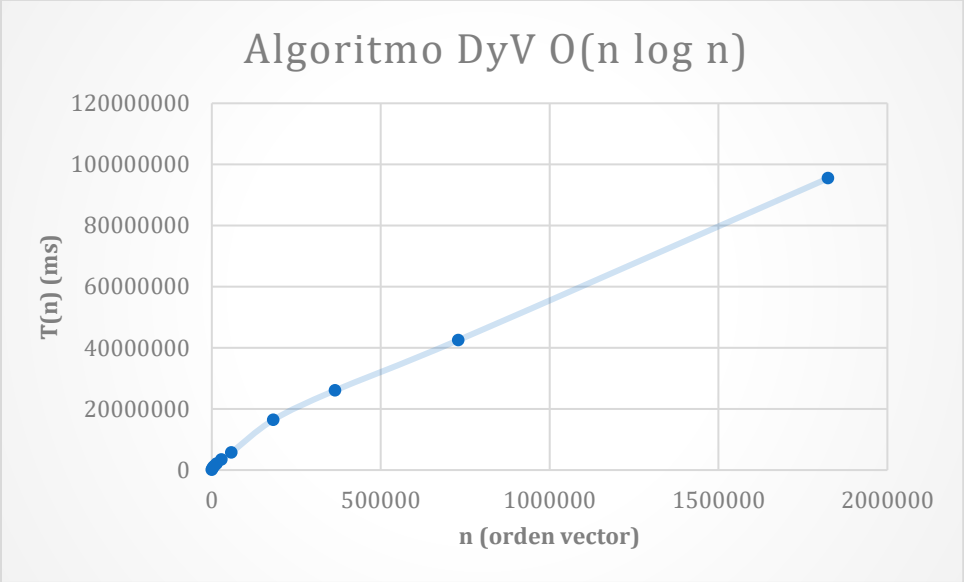


Fijándonos en la anterior gráfica, podemos ver que la pendiente de la ecuación de la recta es **k = 0,5662**. Por tanto, podemos decir que este algoritmo tiene orden de complejidad $\boldsymbol{O(n^{0,5662})}$.

Algoritmo DyV O(n log n)

Años	n (orden matriz)	$\bar{T}(n)$ (ms)	log2(n)
1	365	75225	8,511752654
10	3650	725740	11,83368075
20	7300	1265330	12,83368075
40	14600	2107725	13,83368075
80	29200	3453730	14,83368075
160	58400	5778470	15,83368075
500	182500	16446560	17,47753694
1000	365000	26064320	18,47753694
2000	730000	42508610	19,47753694
5000	1825000	95474205	20,79946503

Tiempo de Ejecución VS Tamaño de la Entrada

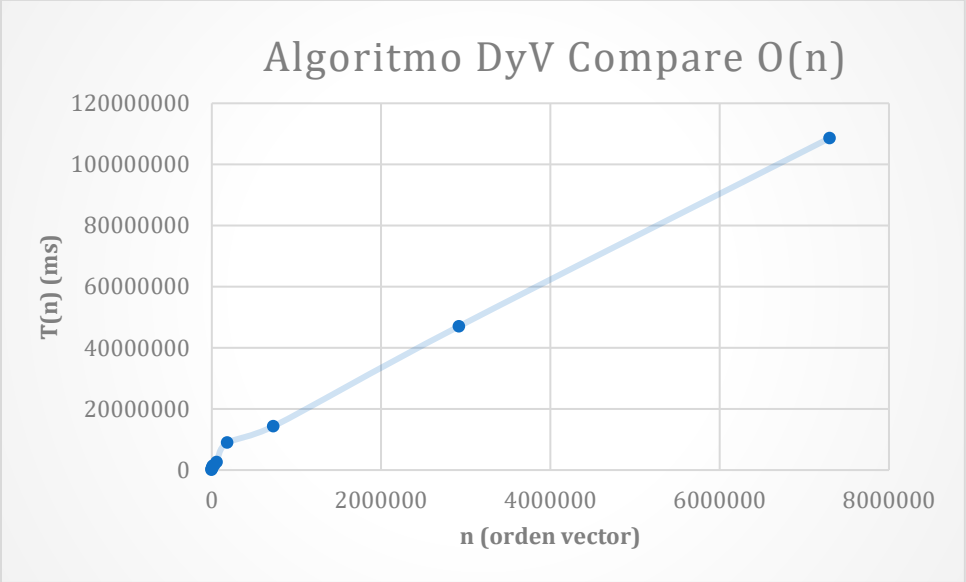


Gráfica tiempo de ejecución VS Tamaño de la entrada

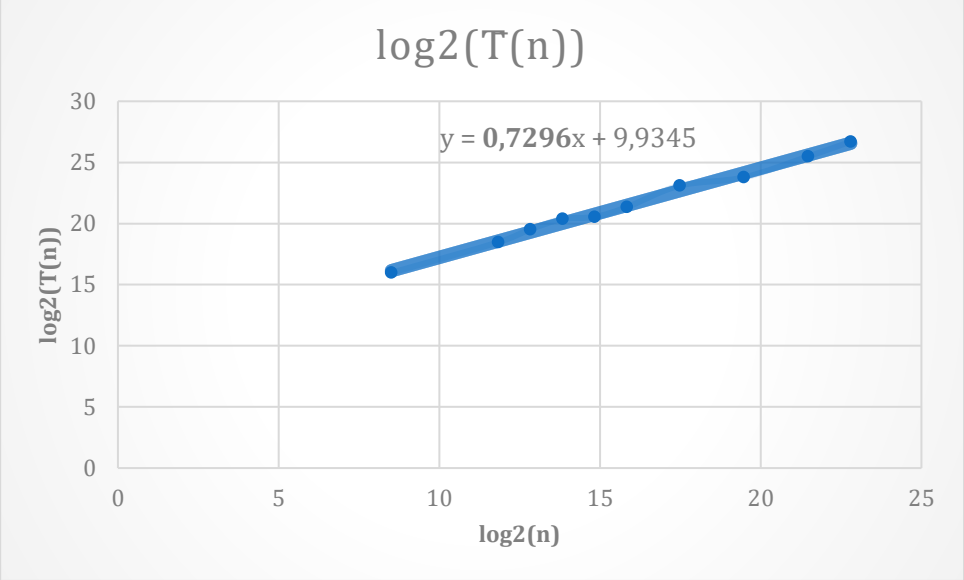
Algoritmo DyV Compare O(n)

Años	n (orden matriz)	T̄(n) (ms)	log2(n)
1	365	64545	8,511752654
10	3650	362735	11,83368075
20	7300	756235	12,83368075
40	14600	1358640	13,83368075
80	29200	1524665	14,83368075
160	58400	2650350	15,83368075
500	182500	8992675	17,47753694
2000	730000	14351855	19,47753694
8000	2920000	47026975	21,47753694
20000	7300000	108612905	22,79946503

Tiempo de Ejecución VS Tamaño de la Entrada



Gráfica tiempo de ejecución VS Tamaño de la entrada

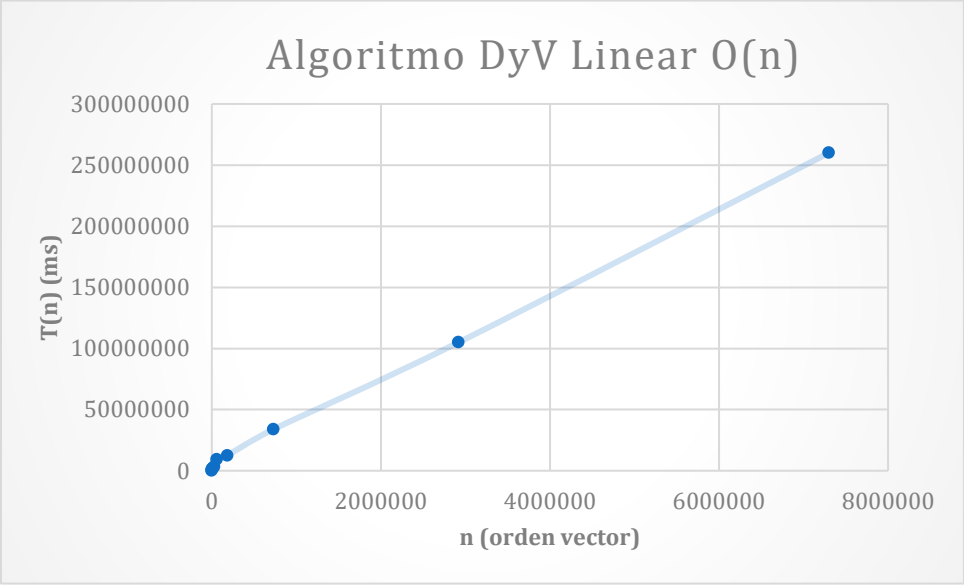


Fijándonos en la anterior gráfica, podemos ver que la pendiente de la ecuación de la recta es **k = 0,7296**. Por tanto, podemos decir que este algoritmo tiene orden de complejidad $\boldsymbol{O(n^{0,7296})}$.

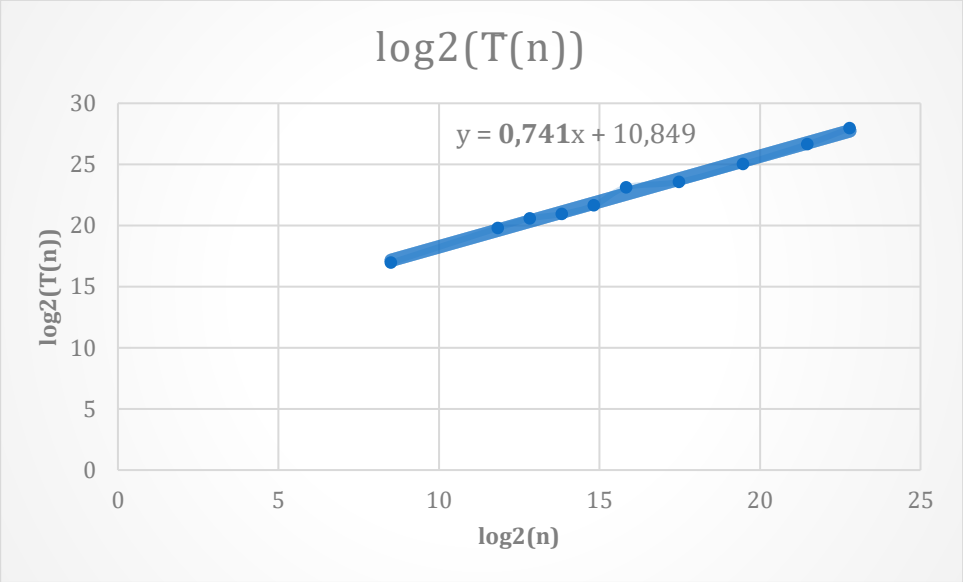
Algoritmo DyV Linear O(n)

Años	n (orden matriz)	T̄(n) (ms)	log2(n)
1	365	128200	8,511752654
10	3650	910145	11,83368075
20	7300	1561215	12,83368075
40	14600	2014780	13,83368075
80	29200	3270180	14,83368075
160	58400	9180420	15,83368075
500	182500	12486830	17,47753694
2000	730000	33938640	19,47753694
8000	2920000	105045175	21,47753694
20000	7300000	260001170	22,79946503

Tiempo de Ejecución VS Tamaño de la Entrada



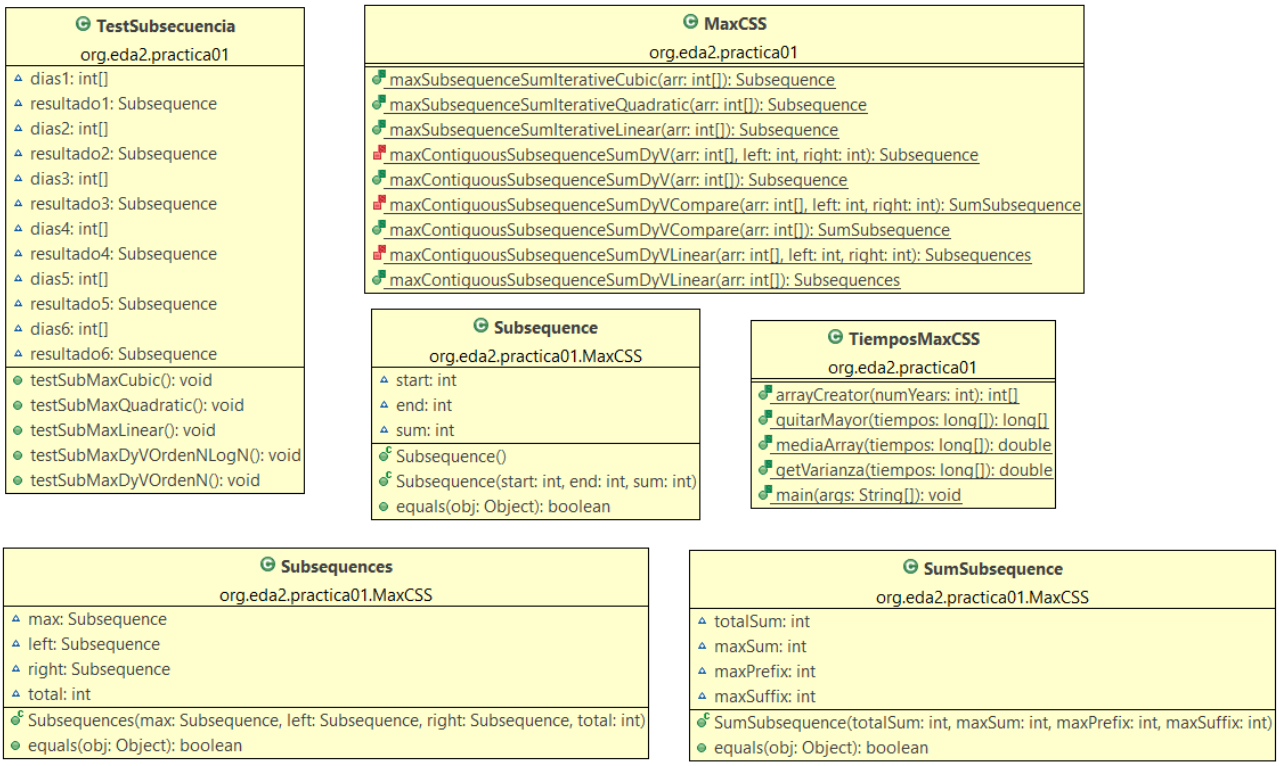
Gráfica tiempo de ejecución VS Tamaño de la entrada



Fijándonos en la anterior gráfica, podemos ver que la pendiente de la ecuación de la recta es **k = 0,741**. Por tanto, podemos decir que este algoritmo tiene orden de complejidad $\boldsymbol{O(n^{0,741})}$.

Anexo

Diagrama de clases:



Archivos fuente:

- **MaxCSS:**

En este archivo dispuesto como clase java podremos encontrar los 5 métodos exigidos durante la práctica destinados al cálculo de subsecuencias máximas, tanto iterativos como recursivos.

- **TiemposMaxCSS:**

En este archivo dispuesto como clase java podemos encontrar una amigable interfaz que nos permite comprobar el tiempo de cualquiera de los algoritmos de la práctica con el n(tamaño de vector de entrada) que deseemos agregar.

- **TestSubsecuencia:**

En este archivo dispuesto como clase java observamos la existencia de una serie de diversos test Junit 5 que nos permiten probar nuestro código tanto con los días de enero de 2023 que se plantean al comienzo de la práctica como con otras más extensas secuencias, además de otro tipo de pruebas para probar al maxímo nuestro código.

Fuentes Bibliográficas

1. "Introduction to Divide and Conquer Algorithms" - GeeksforGeeks. Disponible en: https://www.geeksforgeeks.org/divide-and-conquer-algorithm-introduction/

   Esta página ofrece una introducción clara y concisa al método divide y vencerás, con ejemplos y pseudocódigo.

2. "Divide and Conquer" - Stanford University. Disponible en: https://web.stanford.edu/class/cs103/lectures/03/DivideAndConquer.pdf

   Esta presentación de diapositivas de la Universidad de Stanford ofrece una introducción detallada al método divide y vencerás, con ejemplos y análisis de complejidad.

3. "Divide and Conquer Algorithms" - Tutorialspoint. Disponible en: https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_divide_and_conquer.htm

   Este tutorial ofrece una introducción al método divide y vencerás, con explicaciones detalladas y ejemplos de algoritmos de ordenación y búsqueda.

4. "Divide and Conquer" - Khan Academy. Disponible en: https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms

   Este tutorial de Khan Academy ofrece una introducción al método divide y vencerás, con ejemplos de algoritmos de ordenación y búsqueda, así como una explicación detallada del algoritmo Merge Sort.

5. "Divide and Conquer Algorithms" - Brilliant. Disponible en: https://brilliant.org/wiki/divide-and-conquer-algorithm/

   Este artículo de Brilliant.org ofrece una introducción al método divide y vencerás, con ejemplos de algoritmos de ordenación y búsqueda, así como una explicación detallada del algoritmo Quick Sort.