

# Algorítmica

Curso 2023-2024

## Grupo Viterbi



## PRÁCTICA 2-DIVIDE Y VENCERÁS

### Integrantes:

<b>Miguel Ángel De la Vega Rodríguez</b>	miguevrod@correo.ugr.es
<b>Alberto De la Vera Sánchez</b>	joaquin724@correo.ugr.es
<b>Joaquín Avilés De la Fuente</b>	adelaveras01@correo.ugr.es
<b>Manuel Gomez Rubio</b>	e.manuelgmez@go.ugr.es
<b>Pablo Linari Perez</b>	e.pablolinari@go.ugr.es

Facultad de Ciencias UGR  
Escuela Técnica Ingeniería Informática UGR  
Granada  
2023-2024

# Índice general

<b>1</b>	<b>Autores</b>	<b>3</b>
<b>2</b>	<b>Objetivos</b>	<b>4</b>
<b>3</b>	<b>Definicion Problema</b>	<b>5</b>
<b>4</b>	<b>Algoritmo Especifico</b>	<b>6</b>
4.1	Problema 1: Subsecuencia de suma máxima. . . . .	6
4.1.1	Estudio teórico . . . . .	6
4.1.2	Estudio empírico . . . . .	7
4.1.3	Estudio híbrido . . . . .	7
4.2	Problema 2: Enlosar un espacio . . . . .	8
4.3	Problema 3: Viajante de comercio . . . . .	8
<b>5</b>	<b>Algoritmo Divide y Vencerás</b>	<b>10</b>
5.1	Problema 1: Subsecuencia de suma máxima. . . . .	10
5.1.1	Estudio teórico . . . . .	10
5.1.2	Estudio empírico . . . . .	11
5.1.3	Estudio híbrido . . . . .	12
5.2	Problema 2: Enlosar un espacio . . . . .	12
5.3	Problema 3: Problema del viajante de comercio. . . . .	12
5.3.1	Estudio Empírico . . . . .	12
<b>6</b>	<b>Conclusiones</b>	<b>15</b>

## Autores

- **Miguel Ángel De la Vega Rodríguez:** 20%
  - Plantilla y estructura del documento  $\text{\LaTeX}$
  - Programación Viajante
  - Programación SumaMax (DyV)
  - Tests de eficiencia
- **Joaquín Avilés De la Fuente:** 20%
  - Programacion SumaMax (DyV)
  - Tests de eficiencia
- **Alberto De la Vera Sánchez:** 20%
  - Redacción  $\text{\LaTeX}$
  - Graficas y ajustes
- **Manuel Gomez Rubio** 20%
  - Programacion SumaMax (Kadane)
  - Programacion Loetas
- **Pablo Linari Pérez:** 20%
  - Programacion SumaMax (Kadane)
  - Programacion Loetas

## Apartado 2

### Objetivos

En esta práctica, se pretende resolver problemas de forma eficiente aplicando la técnica de Divide y Vencerás. Para ello, se han planteado varios problemas cuya solución es conocida (excepto para el problema del viajante), y se han implementado algoritmos que los resuelven mediante el método convencional y mediante la técnica de Divide y Vencerás. Posteriormente, se ha buscado un umbral en el cual ambos tengan el mismo tiempo de ejecución, finalmente, se ha buscado el umbral óptimo para cada problema.

Apartado **3**

## Definicion Problema

## Apartado 4

# Algoritmo Especifico

En este apartado, estudiaremos la eficiencia teórica, empírica e híbrida de los algoritmos específicos de cada uno de los problemas.

## 4.1 Problema 1: Subsecuencia de suma máxima.

Para el primer problema, el algoritmo específico que empleamos es el algoritmo de Kadane.

### 4.1.1 Estudio teórico

```
1  int kadane(int *a, int size){
2      int max_global = a[0];
3      int max_current = a[0];
4
5      for (int i = 1; i < size; i++) {
6          max_current = max(a[i], max_current + a[i]);
7          if (max_current > max_global) {
8              max_global = max_current;
9          }
10     }
11     return max_global;
12 }
```

Como podemos observar la eficiencia del código en las líneas 6-8, tienen eficiencia  $O(1)$ . Por tanto, su tiempo de ejecución es constante y notaremos por  $a$ . Luego, el bucle `for` se ejecutará  $(size - 1) - i + 1$  veces, es decir,  $size - i$  veces. Sabiendo que el resto de líneas del código tienen eficiencia  $O(1)$ , tenemos el siguiente resultado

$$\sum_{i=1}^{size-1} a$$

Tomaremos  $size = n$  e  $inicial = 1$  para simplificar el cálculo y veamos que obtenemos ahora

$$\sum_{i=1}^{n-1} a = a \cdot \sum_{i=1}^{n-1} 1 = a \cdot (n - 1)$$

Es claro que  $a \cdot (n - 1) \in O(n)$  y por tanto la eficiencia teórica del algoritmo de Kadane es  $O(n)$ .

### 4.1.2 Estudio empírico

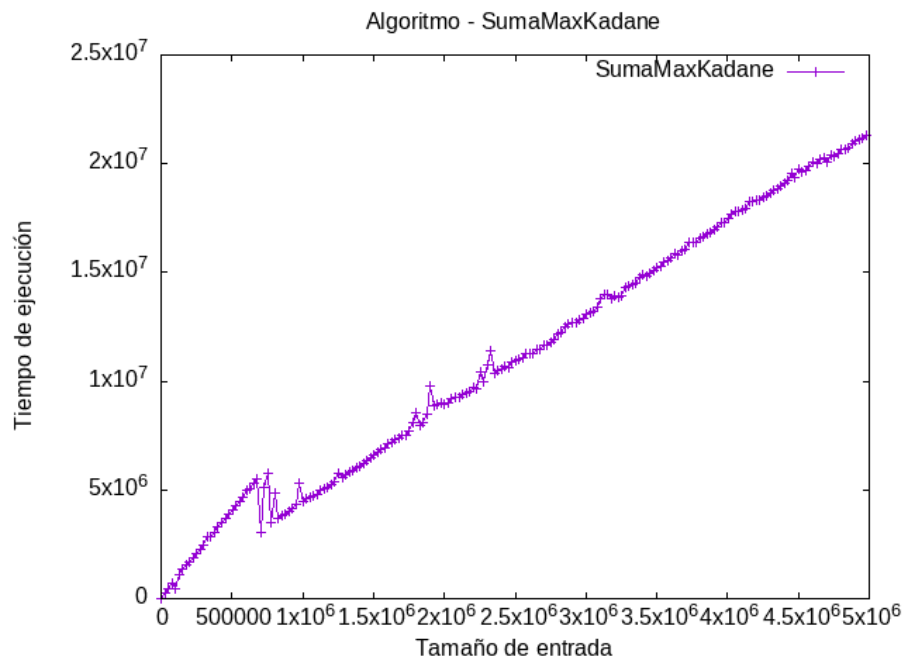


Figura 4.1: Ejecución algoritmo SumaMaxKadane

### 4.1.3 Estudio híbrido

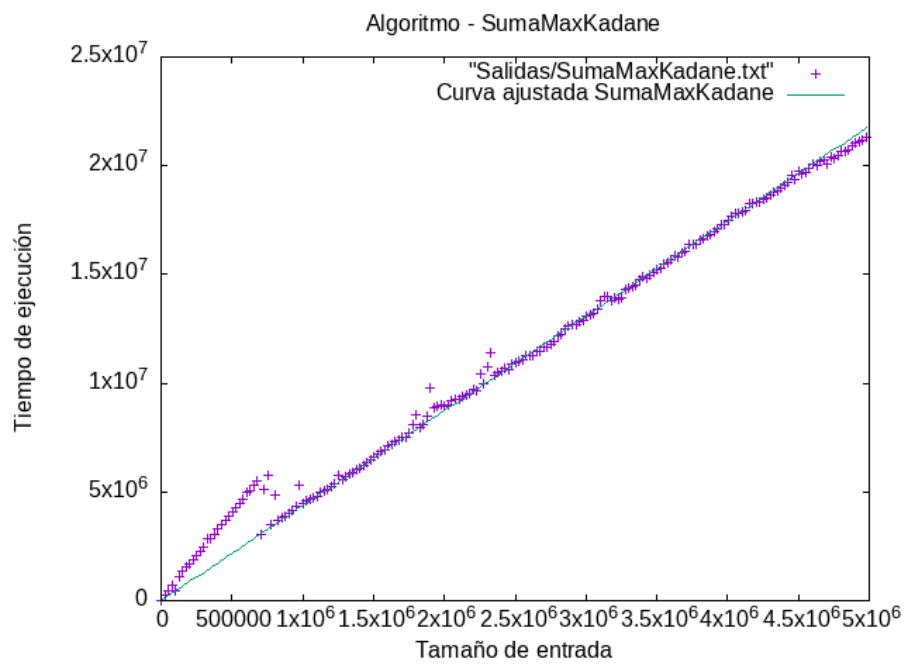


Figura 4.2: Ajuste híbrido

Tras la interpretación de los datos empíricos en gnuplot y de la formula teorica del método, obtenemos que las constantes ocultas son:

$$T_{Kadane} = 1.64263x + 0.996451$$

## 4.2 Problema 2: Enlosar un espacio

Para el segundo problema el algoritmo específico que empleamos para resolver el problema es el siguiente :

```

1  void resuelve2(int i,int j,int n, vector<vector<int>> & mat){
2      losas++;
3      for(int l = 0; l< n; l++){
4          for(int k = 0; k<n; k++){
5              if(mat[i+l][j+k] == 0){
6                  mat[i+l][j+k] = losas;
7              }
8          }
9      }
10 }
11
12 }
```

El código consiste en rellenar las posiciones de una matriz 2x2 con un valor entero en las posiciones donde el valor sea 0. Podemos observar que hay dos bucles for anidados los cuales son  $O(n)$  y el interior del bucle más profundo es  $O(1)$  por tanto la eficiencia de este código es de  $O(n^2)$

## 4.3 Problema 3: Viajante de comercio

Para el tercer problema el algoritmo específico que empleamos para resolver el problema es el siguiente :

```

1  vector<Point> bruteForceTSP(const std::vector<Point>& points) {
2      std::vector<int> permutation(points.size());
3
4      std::iota(permutation.begin(), permutation.end(), 0);
5
6      double minDistance = std::numeric_limits<double>::max();
7      std::vector<int> bestPermutation;
8      do {
9          double distance = 0;
10         for (int i = 0; i < permutation.size() - 1; ++i) {
11             distance += points[permutation[i]].distanceTo(
12                 points[permutation[i + 1]]);
13         }
14
15         distance += points[permutation.back()].distanceTo(
16             points[permutation.front()]);
17         if (distance < minDistance) {
18             minDistance = distance;
19             bestPermutation = permutation;
20         }
21     } while (std::next_permutation(permutation.begin(),
22         permutation.end()));
23 }
```



```

21
22         std::vector<Point> bestPath;
23
24         for (int index : bestPermutation) {
25             bestPath.emplace_back(points[index]);
26         }
27
28         return bestPath;
29     }

```

Destacar que hemos implementado una clase Point que no tiene nada en particular, pues solo se ha implementado como método útil el método **distanceTo(const &Point p)** que calcula la distnacia entre dos puntos mediante la distnacia euclídea, veámoslo:

```

1         double distanceTo(const Point &p) const {
2             return sqrt(pow(x - p.x, 2) + pow(y - p.y, 2));
3         }

```

Para estudiar la eficiencia de dicho algoritmo hay que tener claro como funcionan ciertas funciones de las librerías de C++, así como su eficiencia teórica. En este caso, la función **std::iota** tiene eficiencia  $O(n)$  y se encarga de rellenar un **vector<int>** con una permutación que se inicia en 0 (último parámetro dado) y acaba en  $n - 1$  (tamaño del vector), definido a partir de los puntero al inicio y final del vector.

Por otro lado, la función **std::next\_permutation** tiene eficiencia  $O(n)$  y se encarga de generar la siguiente permutación posible, devolviendo false si ya ha devuelto todas las permutaciones posible. Por tanto, el bucle do-while se ejecutará  $n!$  veces, donde  $n$  es el tamaño del vector de puntos.

Por ahora tenemos una eficiencia de  $O(n)$  en la llamda a la función **std::iota**, el bucle **do-while**, que continuación veremos su eficiencia, y el bucle for de la línea 24 que es claro que tiene eficiencia  $O(n)$  pues recorre la mejor permutación con  $n$  elementos, es decir, tenemos que la eficiencia teórica de dicha función es  $\max\{O(\text{bucle do-while}), O(n)\}$ .

Dentro del bucle **do-while** tenemos un bucle for que recorre la permutación de tamaño  $n$  y realiza una operación de eficiencia  $O(1)$ , por tanto, la eficiencia del bucle for es  $O(n)$ . El resto de operaciones de dicho bucle son de eficiencia  $O(1)$ , por lo que la eficiencia del bucle **do-while** es  $O(n \cdot n!)$ , es decir, la eficiencia de la función es  $O(n \cdot n!)$ .

## Apartado 5

# Algoritmo Divide y Vencerás

En este apartado, estudiaremos la eficiencia teórica, empírica e híbrida de los algoritmos divide y vencerás de cada uno de los problemas.

## 5.1 Problema 1: Subsecuencia de suma máxima.

Para el primer problema, el algoritmo específico que empleamos es

### 5.1.1 Estudio teórico

```
1  int maxSubArray(const int * const arr, int low, int high, int
2      threshold) {
3      if (high - low + 1 <= threshold) {
4          return kadane(arr, low, high);
5      }
6
7      int mid = low + (high - low) / 2;
8
9      int leftSum = maxSubArray(arr, low, mid, threshold);
10     int rightSum = maxSubArray(arr, mid + 1, high, threshold);
11
12     int crossLeftSum = numeric_limits<int>::min();
13     int crossRightSum = numeric_limits<int>::min();
14     int sum = 0;
15
16     for (int i = mid; i >= low; --i) {
17         sum += arr[i];
18         crossLeftSum = max(crossLeftSum, sum);
19     }
20
21     sum = 0;
22     for (int i = mid + 1; i <= high; ++i) {
23         sum += arr[i];
24         crossRightSum = max(crossRightSum, sum);
25     }
26
27     int crossSum = crossLeftSum + crossRightSum;
```

```

28         return max(max(leftSum, rightSum), crossSum);
29     }

```

Este algoritmo hace uso de la técnica de **Divide y Vencerás** para resolver el problema de la subsecuencia de suma máxima, en el cual se divide el problema en subproblemas más pequeños y se resuelven de forma recursiva. Como se observa se usa el **algoritmo de Kadane** para resolver el caso base dado como el parámetro **threshold**.

El caso base solo usará cuando lleguemos a tamaños del problema menores o iguales a **threshold**, por lo que podemos obviar su eficiencia, ya que tardará un tiempo constante en resolverlo. Por otro lado, el resto de las operaciones son la mayoría de eficiencia  $O(1)$ , excepto los bucles for que tienen eficiencia  $O(n/2) = O(n)$  y las llamadas recursivas al algoritmo con un tamaño  $n/2$  (destacar que tomaremos  $n = high - low$  como el tamaño del problema). Tomando estas ideas previas y usando la regla del máximo, veamos los calculos de la eficiencia teórica:

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

Pasemos ahora a resolver dicha ecuación de recurrencia. Aplicando el siguiente cambio de variable  $n = 2^m$  obtenemos

$$T(2^m) = 2T(2^{m-1}) + 2^m \implies T(2^m) - 2T(2^{m-1}) = 2^m$$

Resolvamos la parte homogénea de la ecuación, es decir, la ecuación  $T(2^m) - 2T(2^{m-1}) = 0$ . Obtenemos el polinomio característico de la parte homogénea que es  $p_H(x) = x - 2$  cuya raíz es  $x = 2$ .

Obtengamos ahora la parte no homogénea

$$2^m = b_1^m q_1(m) \implies b_1 = 2 \wedge q_1(m) = 1 \text{ con grado } d_1 = 0$$

Tenemos entonces el siguiente polinomio característico

$$p(x) = (x - 2)(x - b_1)^{d_1+1} = (x - 2)^2$$

Por tanto la solución general es

$$t_m = c_{10}2^m m^0 + c_{11}2^m m^1 \stackrel{*}{\implies} t_n = c_{10}n + c_{11}n \log_2(n) \implies T(n) = c_{10}n + c_{11}n \log_2(n)$$

donde en (\*) hemos deshecho el cambio de variable

Aplicando la regla del máximo tenemos  $T(n) \in O(n \log(n))$

### 5.1.2 Estudio empírico

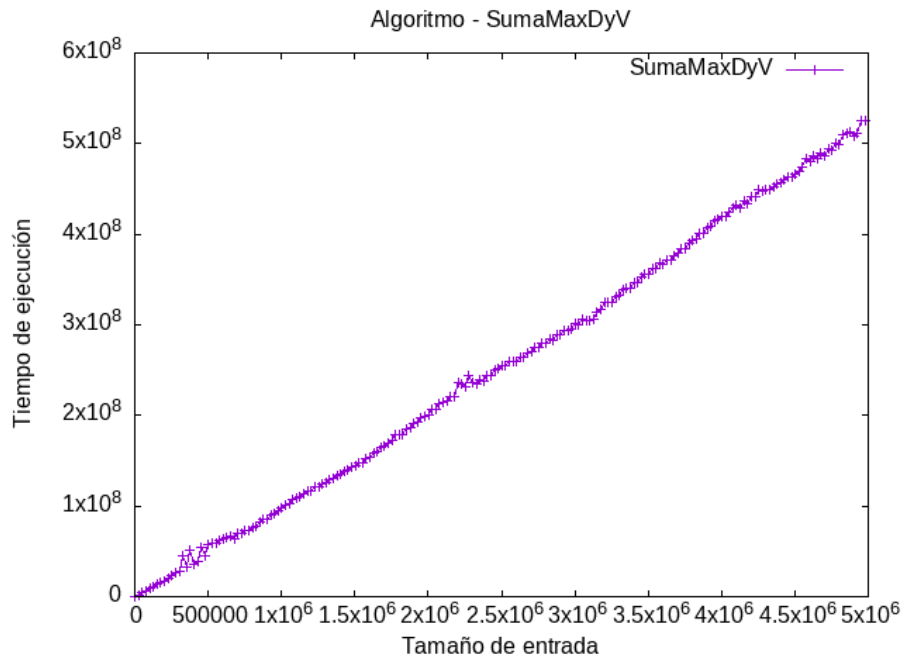


Figura 5.1: Ejecución algoritmo SumaMaxDyV

### 5.1.3 Estudio híbrido

Tras la interpretación de los datos empíricos en gnuplot y de la formula teorica del método, obtenemos que las constantes ocultas son:

$$T_{SumaMaxDyV} = 1.65508x + 0.991967$$

## 5.2 Problema 2: Enlosar un espacio

## 5.3 Problema 3: Problema del viajante de comercio.

Para el tercer problema, nos encontramos ante un problema de complejidad NP-duro, por lo que no podemos encontrar una solución exacta en tiempo polinómico. Sin embargo, podemos encontrar una solución aproximada aplicando la técnica de Divide y Vencerás. Para ello, usamos el algoritmo de fuerza bruta para encontrar la solución exacta para subconjuntos del problema total y luego unimos las soluciones parciales para obtener una solución que se optimiza para asegurar un cierto mínimo de precisión.

### 5.3.1 Estudio Empírico

Para determinar el umbral con el que obtenemos la mejor relación entre eficiencia y precisión, hemos realizado una serie de pruebas con diferentes valores de umbral, a partir de tamaño 10, el algoritmo de fuerza bruta se vuelve completamente inviable, de forma similar, para tamaño menor a 4, el algoritmo no nos produce ninguna mejora en cuanto a eficiencia debido a la cantidad de llamadas recursivas que saturan la pila, y la precisión disminuye de forma considerable. Por ello, hemos decidido realizar las pruebas para tamaños de 4 a 10, con distintas ciudades con solución ya conocida, para comparar los tiempos de ejecución y la precisión de los resultados obtenidos:

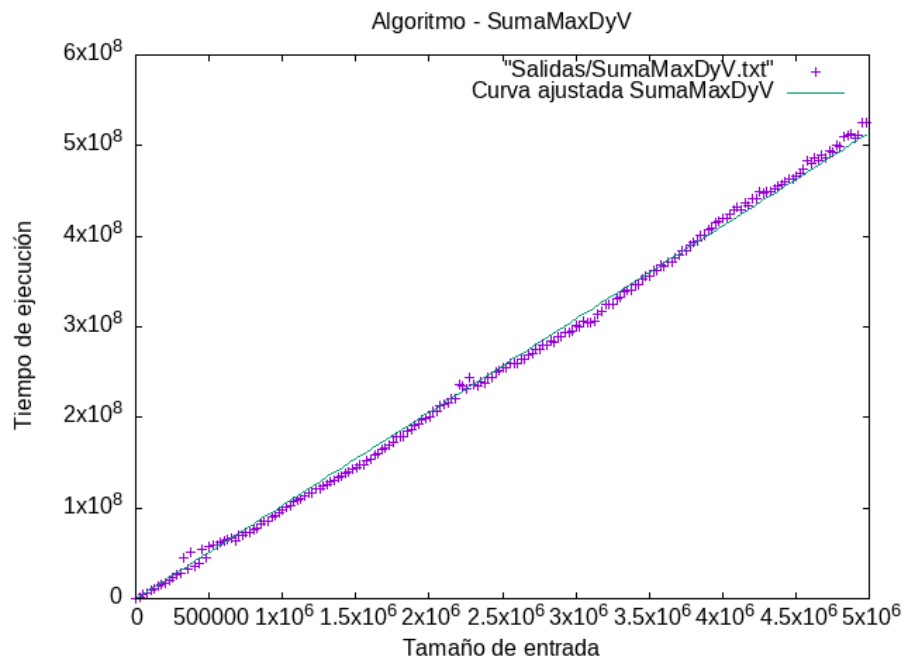


Figura 5.2: Ajuste híbrido algoritmo SumaMaxDyV

### Tiempo de ejecución

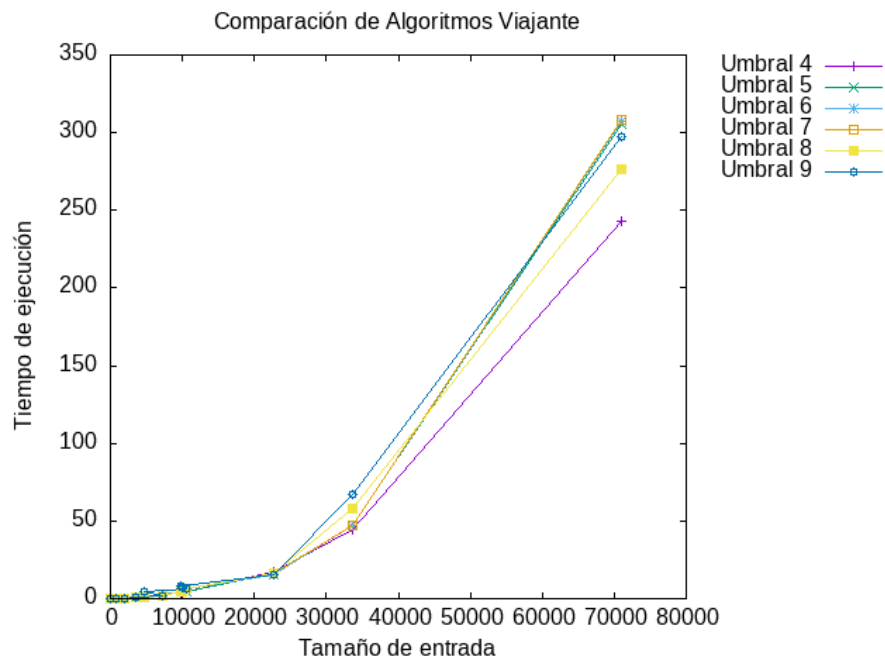


Figura 5.3: Ejecución algoritmo Viajante

Como se puede apreciar, los mejores resultados se han obtenido con los umbrales 4,8. Para decidir cual de los

dos umbrales es el mejor, nos fijamos en la precisión que nos proporciona cada uno de ellos:

**Precisión**

# Apartado 6

## Conclusiones