Algorítmica

Práctica 2: Divide y Vencerás

Doble Grado en Ingeniería Informática y Matemáticas

V de Vendetta

Granada, 2023-2024

José Juan Urrutia Milán: jjum@correo.ugr.es

Airam Falcón Marrero: airamfm@correo.ugr.es

Lucas Hidalgo Herrera: lucashidalgo@correo.ugr.es

Irina Kuzyshyn Basarab: kuzirina@correo.ugr.es

Arturo Olivares Martos: arturoolivares@correo.ugr.es

Índice

1. Autores **3**

2. Objetivos **4**

3. Definición del problema **4**
3.1. Entornos de análisis 4
3.2. Medidición de tiempo 4
3.3. Problema 1 5
3.4. Problema 2 5
3.5. Problema 3 5

4. Algoritmo específico **6**
4.1. Problema 1 6
4.1.1. Descripción del algoritmo 6
4.1.2. Análisis teórico 7
4.1.3. Análisis empírico e híbrido 8
4.2. Problema 3 8
4.2.1. Descripción del algoritmo 8
4.2.2. Análisis teórico 9
4.2.3. Análisis empírico e híbrido 10

5. Algoritmo divide y vencerás **12**
5.1. Problema 1 12
5.1.1. Descripción del algoritmo 12
5.1.2. Análisis teórico 13
5.1.3. Análisis empírico e híbrido 15
5.1.4. Cálculo de umbrales 16
5.2. Problema 2 17
5.2.1. Descripción del algoritmo 17
5.2.2. Análisis teórico 20
5.2.3. Análisis empírico e híbrido 20
5.3. Problema 3 21
5.3.1. Descripción del algoritmo 21
5.3.2. Análisis teórico 26
5.3.3. Análisis empírico e híbrido 28
5.3.4. Cálculo de umbrales 28

6. Conclusiones **32**

1. Autores

Esta práctica ha sido realizada por los miembros de la siguiente lista. Se repartió el trabajo según unas directrices que considerábamos equitativas y que se describirán a continuación, aunque al final todos los integrantes han acabado colaborando en todo.

- José Juan Urrutia Milán. Se me ocurrió la idea del algoritmo del problema 3 DyV, que lo implementé con la ayuda de Irina y conseguí que no tuviera errores gracias a los testeos de Arturo. Además, realicé su análisis teórico así como la descripción del propio algoritmo. Fui el encargado de crear los generadores para los tres programas, capaces de generar instancias, ejecutarlas, generar gráficas de sus tiempos y (en el caso del problema 3) representar gráficamente sus salidas. También me he dedicado a darle formato a las tablas de datos, haciendo uso del editor *Vim*.

- Airam Falcón Marrero. Se me ocurrieron las ideas del algoritmo del problema 1 iterativo y del ejercicio 2. He implementado ambos algoritmos y he comprobado que no tuvieran errores. Realicé el análisis teórico del ejercicio 2, así como la explicación de amobos algoritmos. He ayudado con los umbrales del tercer ejercicio y colaborado en la obtención de su idea en la lluvia de ideas que se realizó. Además, he ayudado a la depuración del código del algoritmo especifico creado para el problema 3, proponiendo una solución a un problema encontrado por mis compañeros y explicando la causa del problema.

- Lucas Hidalgo Herrera. Yo me he encargado de realizar, tanto el análisis teórico del algoritmo iterativo del problema 1, como del algoritmo DyV. Además, he implementado este último con la perfección y limpieza impuesta e implementada por Arturo. Asimismo, he realizado el análisis empírico e híbrido del algoritmo DyV de este mismo problema y la explicación de cómo llegamos a dicha solución junto con su desarrollo. Por último, he calculado el umbral híbrido de este mismo problema.

- Irina Kuzyshyn Basarab. Respecto a la implementación de algoritmos, he implementado el Específico del Problema 3, descrito en la Sección 3.5. Además, he ayudado a José en la implementación y depuración de la versión DyV de este mismo problema. Por otra parte, he realizado el análisis teórico de mi algoritmo (el Específico del problema 3), he estimado el umbral del Problema 3 y he llevado a cabo el análisis empírico e híbrido de dicho problema en su totalidad, tanto del Específico como del DyV. Además he adaptado todos los programas `main` al generador.

- Arturo Olivares Martos. En primer lugar, respecto a la implementación de algoritmos, he terminado el Problema 1 descrito en la Sección 3.3. Aunque Lucas lo comenzó y lo hemos hecho entre ambos, yo retoqué algunos aspectos, corregí algunos fallos concretos que tenía para algunos casos y, finalmente, mejoré en cierto modo la eficiencia. Además, he documentado todos los archivos `cpp` del proyecto, siguiendo el estándar de documentación de Doxygen. Asimismo, todos los archivos `makefile` disponibles los he diseñado yo, que nos

facilitan no solo la compilación de los ejecutables sino también la organización de las carpetas con las reglas `save`. Por último, también he sido el encargado de calcular los umbrales para el Problema 1.

2. Objetivos

El objetivo que persigue esta práctica es aprender sobre el diseño e implementación de algoritmos mediante la técnica Divide y Vencerás. Gracias a la naturaleza de este tipo de algoritmos (recursiva), a su vez nos sirve para practicar el cálculo de umbrales (establecer un tamaño hasta el que ejecutar un algoritmo y desde el que ejecutar otro, para obtener menores tiempos de ejecución), tanto de forma teórica como práctica.

3. Definición del problema

3.1. Entornos de análisis

Los resultados empíricos han sido obtenidos en distintos ordenadores, cuyo hardware y configuraciones se detallan en la Tabla 1.

Alumno	CPU	RAM	L1 (d)/(i)	L2	L3	SO	Compilador
Airam	i7 1165G7	8 GB	192KB / / 128KB	5 MB	12 MB	Ubuntu 22.04.4	g++
Arturo	i5 5350U	8GB	64KB / / 64KB	512KB	3MB	Ubuntu 22.04.3	g++
Irina	Ryzen 7 5800H	16GB	256KB / / 256KB	4 MB	16 MB	Ubuntu 23.10	g++
José Juan	Ryzen 7 4800H	16GB	256KB / / 256KB	4 MB	8 MB	Ubuntu 23.10	g++
Lucas	i7 1165G7	8 GB	192KB / / 128KB	5 MB	12 MB	Ubuntu 22.04.4	g++

Tabla 1: Especificaciones de los ordenadores de cada uno de los integrantes.

3.2. Medidición de tiempo

Para medir los tiempos hemos usado la biblioteca `chrono`. Haciendo uso del siguiente tipo (definido en dicha biblioteca) `high_resolution_clock::time_point` y con la función `now()`, nos quedamos con los instantes de tiempo anterior y posterior a ejecutar nuestro algoritmo. Tras ello restamos estos tiempos y obtenemos el tiempo transcurrido entre ambos instantes, es decir, el tiempo que tarda el algoritmo en ejecutarse. Para ello, hace falta hacer un *casting* del tipo de dato para el tiempo a `double`, que es el dato con el que podemos trabajar. Podemos ver un ejemplo en el Código Fuente 1.

```
1  high_resolution_clock::time_point tantes, tdespues;
2  duration<double> transcurrido;
3
4  tantes = high_resolution_clock::now();
5  mejor_camino_Iterativo(ciudades, 0, n-1);
6  tdespues = high_resolution_clock::now();
7
8  transcurrido = duration_cast<duration<double>>(tdespues - tantes);
```

Código fuente 1: Ejemplo de medición de tiempo.

3.3. Problema 1

Dado una secuencia de valores reales (positivos o negativos), almacenada en un vector, buscamos encontrar una subsecuencia de elementos consecutivos que cumpla que la suma de los elementos de la misma sea la máxima posible. Se puede considerar la subsecuencia vacía, que contiene 0 elementos y su suma es nula (que será la solución en el caso de un vector cuyos elementos sean todos negativos).

3.4. Problema 2

Queremos cubrir el suelo de una habitación cuadrada con baldosas en forma de "ele" (L, que ocupan tres losetas). En el suelo sabemos de la ubicación de un sumidero que ocupa exactamente una loseta. Se pide encontrar la forma de colocar las losetas (sin romper ninguna) en nuestro suelo. Asumiremos que el suelo se representa como una matriz bidimensional A de tamaño $n \times n$, donde $n = 2^k$ para algún $k \geqslant 1$. La posición del sumidero la conocemos a priori por el par de valores i, j con $0 \geqslant i, j \geqslant n-1$. En nuestra matriz almacenaremos en la celda $A[i][j]$ el valor 0. Para cada baldosa que coloquemos en la matriz le asociaremos un identificador (entero) distinto. Se pide diseñar una algoritmo que permita encontrar la forma de rellenar el suelo (la matriz) de baldosas.

Observación. De este problema, es importante notar que no conocemos una solución iterativa, por lo que no se incluirá su estudio, y tampoco se realizará el cálculo de los umbrales.

3.5. Problema 3

Es el conocido problema del *Viajante del Comercio* (*TSP*, por sus siglas en inglés). Tenemos un conjunto de n ciudades (puntos en un plano), cada una definida por las coordenadas en el mapa (x_i, y_i), con $i = 1, \ldots, n$. La distancia entre dos ciudades viene dada por la distancia euclídea entre sus coordenadas:

$$dist((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

El problema de viajante de comercio consiste en encontrar el orden en el que un viajante, partiendo de la ciudad de origen (por ejemplo $(x1, y1)$) pase por todas y cada una de las ciudades una única vez, para volver a la ciudad de partida, formando

un ciclo. El costo del ciclo será la suma de las distancias que hay entre las ciudades. El problema original del viajante de comercio consiste en encontrar el ciclo de costo mínimo entre todas las posibilidades existentes. Aunque este problema es NP-Difícil y por tanto no podemos esperar encontrar una solución óptima al mismo, lo que se pretende en esta práctica es utilizar la estrategia del divide y vencerás para encontrar una solución aproximada que puede ser de utilidad en situaciones como las que se plantea en este problema.

4. Algoritmo específico

4.1. Problema 1

4.1.1. Descripción del algoritmo

Para encontrar la subsecuencia máxima, consideramos la suma local de una subsecuencia como la suma de todos sus valores, empezamos suponiéndola 0. Tendremos también la suma máxima hasta el momento, es decir, la de la subsecuencia que ha dado mejor resultado.

Iteraremos sobre la colección de la siguiente forma: Si la suma local de la subsecuencia es negativa, perjudicará a una posible subsecuencia futura, por lo que se dejará de tener en cuenta y se empezará con una nueva subsecuencia. Si la suma de la subsecuencia no es negativa, beneficia (o al menos no perjudica) a los siguientes términos, por lo que se mantiene la subsecuencia y se añade el nuevo elemento. Se calculará la suma del máximo local, y si este supera al máximo antes encontrado, se marcará la subsecuencia actual como la máxima.

Demostración. Demostraremos por inducción que este algoritmo nos permite llegar a la solución óptima:

- Si $n = 0$, es cierto por ser la única subsecuencia la vacía
- Si se cumple para una colección de n elementos, demostraremos que se cumple para una colección de $n + 1$ elementos:

  Sea A nuestra colección de $n+1$ elementos. Sea a_{n+1} su último elemento. Consideramos la colección $A \setminus \{a_{n+1}\}$, que claramente tiene n elementos. Como tiene n elementos, podemos utilizar la hipótesis de inducción sobre $A \setminus \{a_{n+1}\}$ para encontrar la subsecuencia máxima. Tendremos además la última subsecuencia de la colección calculada con su suma. Sea S esta subsecuencia, y ΣS su suma. Si $\Sigma S < 0 \implies \Sigma S + a_{n+1} < a_{n+1}$, por lo que la subsecuencia $\{a_{n+1}\}$ tiene una suma mayor que $S \cup \{a_{n+1}\}$. Análogamente $\Sigma S \geqslant 0 \implies \Sigma S + a_{n+1} \geqslant a_{n+1}$, por lo que la subsecuencia $\{a_{n+1}\}$ tiene una suma mayor que $S \cup \{a_{n+1}\}$. Sea V la subsecuencia con la suma más grande de las 2. Ahora tomamos el máximo entre la subsecuencia máxima encontrada en $A \setminus \{a_{n+1}\}$ y V. Entonces está claro que ese máximo es la subsecuencia máxima de la colección.

Estos pasos son los que sigue nuestro algoritmo para elegir la subsecuencia, por lo que hemos demostrado que es óptimo. □

El algoritmo iterativo para la resolución del Problema 1 descrito en la Sección 3.3 se encuentra en el código fuente del archivo `Subsecuencia_Iterativo.cpp`. Como podemos ver, se parte suponiendo que la subsecuencia máxima es la vacía. Esto es debido a que en un problema de la vida real suele ser interesante tener la opción de no hacer nada, cuando esto implica no tener pérdidas debido a una decisión que intente hacer lo menos malo pero siempre algo. Como ejemplo de este tipo de situaciones, podemos pensar que un banco modela la ganancia o pérdida de dinero en una compra de acciones de forma que $v[i+1]$ represente la ganancia respecto a $v[i]$, en la que la ganancia puede ser negativa. Entonces en estas condiciones es mejor no hacer nada. Una vez se encuentre una subsecuencia con suma positiva, esta pasa a ser la nueva subsecuencia máxima.

4.1.2. Análisis teórico

En esta sección se va a realizar el análisis teórico del código proporcionado como solución del problema sobre el cálculo de la subsecuencia de suma máxima dentro de una secuencia de números enteros dada. El código iterativo, el cual se analiza en esta sección, se encuentra en el archivo `Subsecuencia_Iterativo.cpp`.

Antes de empezar, vamos a fijar la notación del problema. Para tomar el tamaño del problema, consideraremos que el tamaño del problema es la longitud de la secuencia de números enteros `v`, notémoslo $n \in \mathbb{N}$. A su vez, llamaremos $T(n)$ a la función que representa el tiempo de ejecución de la función `subsecMax_Iterativo`. Por último, aclarar que los parámetros `init` y `fin` representan los extremos sobre los cuales se delimita el problema dentro del vector `v`, respectivamente.

Para comenzar con el análisis teórico, es claro que la líneas de código implementadas antes del bucle `for` son de orden de eficiencia constante y serán acotadas por una constante $a \in \mathbb{R}^+$.

Pasamos ahora a analizar el bucle `for` de la función, el cuál aporta el grueso del tiempo de ejecución. Es fácil ver que el bucle se ejecutará $n = fin - init + 1$ veces. Veamos ahora la eficiencia del cuerpo del bucle, pues la eficiencia de la inicialización, la evaluación de la condición y el incremento es constante.

Dentro del bucle encontramos una estructura condicional donde la evaluación de la condición es, al igual que el cuerpo de ambas partes de la estructura, constante. Por tanto, lo acotaremos por una constante $b \in \mathbb{R}^+$.

Uniendo lo visto por ahora, si $T'(n)$ es la eficiencia del bucle `for` entonces:

$$T'(n) = \sum_{i=0}^{n} b = b \cdot (n+1) = bn + b$$

Por tanto, $T'(n) \in O(n)$, donde he aplicado la regla del máximo. Veamos ahora el tiempo de ejecución completo de la función `subsecMax_Iterativo`; que será la suma de la eficiencia del bucle y de las líneas constantes acotadas al principio del análisis. Es decir:

$$T(n) = T'(n) + a = bn + b + a$$

Entonces $T(n) \in O(n)$, donde he aplicado la regla del máximo.

Tamaño	Tiempo (seg)	Tamaño	Tiempo (seg)
50000	$7{,}0641 \cdot 10^{-5}$	530000	0,0008368
70000	$9{,}5429 \cdot 10^{-5}$	550000	0,000473344
90000	0,000127403	570000	0,000428318
110000	0,000130402	590000	0,000480426
130000	0,000200481	610000	0,00049864
150000	0,00024666	630000	0,000473369
170000	0,000227284	650000	0,000516248
190000	0,00016681	670000	0,000498081
210000	0,000175369	690000	0,000558189
230000	0,000187859	710000	0,000583036
250000	0,000205601	730000	0,000592455
270000	0,000218972	750000	0,000634078
290000	0,000231397	770000	0,000569266
310000	0,000254723	790000	0,000633966
330000	0,000272306	810000	0,000648165
350000	0,000280353	830000	0,00064319
370000	0,000333783	850000	0,000615067
390000	0,000313073	870000	0,000704042
410000	0,000312292	890000	0,000729641
430000	0,000346368	910000	0,000873555
450000	0,000360181	930000	0,000757012
470000	0,00040389	950000	0,000782079
490000	0,00040032	970000	0,000789721
510000	0,000412015	990000	0,000804991

Tabla 2: Tiempos de ejecución del Programa 1 iterativo en el PC de Arturo.

4.1.3. Análisis empírico e híbrido

Tras ejecutar el algoritmo específico para tamaños de n desde $5 \cdot 10^4$ a 10^7 con saltos de $2 \cdot 10^4$, obtenemos la Tabla 2 que relaciona el tamaño de n con su tiempo de ejecución en segundos. Representamos estos datos en la gráfica de la Figura 1, con ayuda de `gnuplot`. El rango empleado ha sido $x \in [-4000, 4000]$.

Habiendo estudiado el análisis teórico, sabemos que el algoritmo es de eficiencia $O(n)$. Buscamos una regresión según la función $f(x) = ax + b$, donde a y b son constantes que vamos a calcular mediante los datos empíricos medidos. Con la ayuda de `gnuplot`, en la Figura 1 también se muestra la regresión calculada. La función de regresión es:

$$f(x) = 7{,}60449 \cdot 10^{-10} \cdot x + 4{,}40934 \cdot 10^{-5}$$

4.2. Problema 3

4.2.1. Descripción del algoritmo

Como hemos de cerrar el camino, hemos convenido en dejar la primera ciudad estática (además, para el algoritmo DyV que hemos diseñado esto es necesario, por razones que explicará José en la Sección 5.3.1). Para generar todas las permutaciones

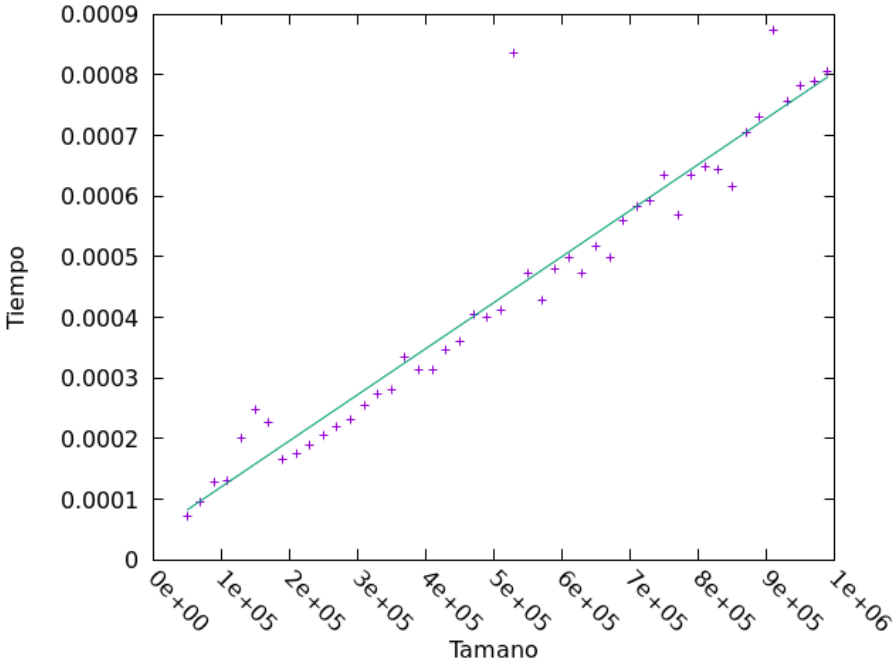


Figura 1: Ajuste de los tiempos del Programa 1 en PC Arturo.

posibles de mi vector de ciudades y así poder calcular todas las posibles distancias; usamos la función de la STL `next_permutation`. Esta función necesita que le demos los elementos ordenados en orden ascendente, por lo que antes de nada ordenamos todos los elementos menos el primero (por lo anteriormente mencionado). El orden final lo almacenaremos en el propio vector que se nos pasa como argumento. Además vamos a crear un vector auxiliar que iremos permutando, obteniendo así todos los caminos posibles; y tendremos una variable que nos guarda la distancia mínima.

Simplemente vamos permutando nuestro vector auxiliar y calculando la distancia que cada camino genera. Tras esto, vemos si ha superado a la distancia mínima que tenemos hasta ahora, y si es así, actualizamos la distancia mínima y el vector con la permutación correspondiente.

En resumen, el algoritmo se basa en calcular todas las permutaciones del vector y quedarnos con la que optimice la distancia recorrida por el Viajante de Comercio.

4.2.2. Análisis teórico

Vamos a referenciar al código del problema 3, descrito en la sección 3.5 e implementado en el archivo `ViajanteDelComercio_Iterativo.cpp`.

Al principio del algoritmo nos encontramos un `if` de orden constante pero como el resto del bloque de la función va a ser más que orden constante, por la Regla de la Suma nos quedaremos con la otra parte. Pasemos ahora a analizarla.

En primer lugar, tenemos un `sort`, que es de eficiencia $O(n \log n)$. A continuación tenemos varias sentencias de orden constante que acotaremos por $a \in \mathbb{R}$. Ahora vemos un bloque `for` con una orden constante dentro, por lo que tenemos orden lineal. Y por último, antes de adentrarnos en el bloque más pesado de la función, tenemos una simple asignación que añadiremos a nuestra constante a antes definida. Estudiemos ahora el bloque `do-while`.

Tamaño	Tiempo (seg)
1	$1{,}1 \cdot 10^{-7}$
2	$2{,}044 \cdot 10^{-6}$
3	$1{,}493 \cdot 10^{-6}$
4	$2{,}044 \cdot 10^{-6}$
5	$2{,}555 \cdot 10^{-6}$
6	$4{,}329 \cdot 10^{-6}$
7	$1{,}571 \cdot 10^{-5}$
8	0,000106233
9	0,000936603
10	0,00770162
11	0,0819529
12	0,977274
13	12,7141

Tabla 3: Tiempos del algoritmo iterativo del Viajante del Comercio en el PC Irina.

Primero vemos una asignación, orden constante como ya sabemos. A continuación vemos un bloque `for` con $n-1$ iteraciones más la iteración que hay fuera. El código en cuestión al que nos referimos es la llamada a la función `dist`, que vemos fácilmente que es de orden constante, y que acotaremos por la constante real $b \in \mathbb{R}$. Y por último, dentro de este bloque tenemos un `if` con una asignación y un `for` de orden lineal, lo que se resume en que dicho bloque es de orden lineal.

Este bloque `do-while` que acabamos de describir se repite $(n-1)!$ veces, pues calcula todas las permutaciones posibles en un vector de $n-1$ elementos. Y además, la función `next_permutation` de la STL es de orden lineal, por lo que le tenemos que añadir este al bloque. En resumidas cuentas, sea $T(n)$ el tiempo del algoritmo:

$$T(n) = n \cdot \log n + a + n + (n-1)! \cdot (nb + 2n) = n \cdot \log n + a + n + b \cdot n! + 2 \cdot n! \in O(n!)$$

4.2.3. Análisis empírico e híbrido

Como acabamos de estudiar, estamos hablando de un algoritmo $n!$, por lo que no va a soportar valores de n muy grandes. Hemos probado a ejecutarlo con $n = 14$, y este tarda 177,6 segundos. Podríamos considerar este dato, pero lo único que va a pasar es que vamos a perder precisión en la gráfica, y no vamos a llegar a ninguna conclusión fácilmente. Este caso nos sirve para ilustrar eso sí, la magnitud del factorial. Vamos a acotar por tanto nuestro estudio a $n = 13$.

Para los valores desde $n = 1$ a $n = 13$ obtenemos los tiempos que podemos ver en la Tabla 3, y estos podemos verlos gráficamente en las Figuras 2 y 3, en la primera vemos la dispersión de puntos simplemente y en la segunda podemos observar el ajuste respecto a la función factorial.

Como podemos observar tanto en la tabla de tiempos como visualmente en las gráficas, ya incluso para $n = 13$ el tiempo sube bastante con respecto a los anteriores, por no hablar del $n = 14$, donde hablamos de casi 3 minutos de ejecución, en relación al segundo que tenemos para $n = 12$. El ajuste que ha realizado `gnuplot` es el

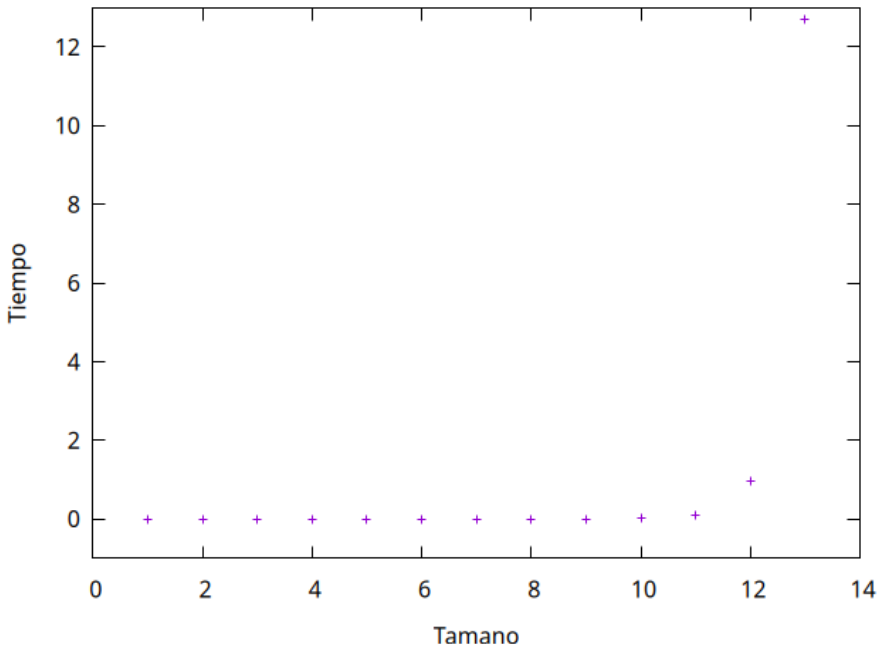


Figura 2: Dispersión de puntos para la versión iterativa del Problema 3 en PC Irina.

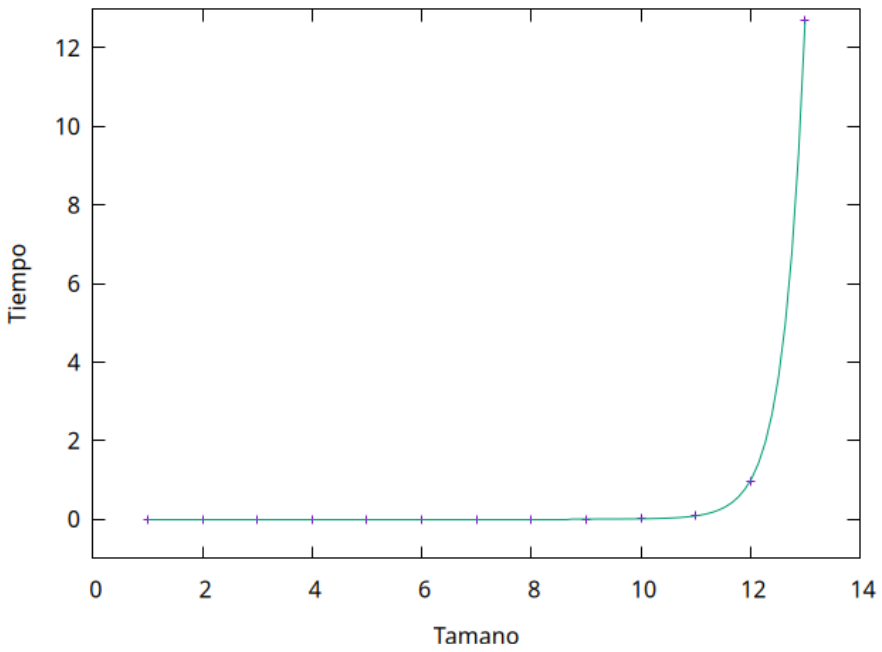


Figura 3: Regresión para la versión iterativa del Problema 3 en PC Irina.

siguiente, donde se ha usado la función Gamma[1]:

$$f(x) = 2{,}04175 \cdot 10^{-9} \cdot x!$$

Si evaluamos esta función en 14, que sería el tiempo estimado para $n = 14$, obtenemos lo siguiente:

$$f(14) = 2{,}04175 \cdot 10^{-9} \cdot 14! \approx 178,$$

que, como podemos ver, se acerca muchísimo al dato tomado empíricamente. Por tanto, el ajuste teórico que hemos obtenido es muy bueno para estimar más resultados. Haciendo algunos simples cálculos para $n = 15$ tenemos que tardaría unos 44 minutos y para $n = 16$ unas 11 horas. Así, se observa muy fácilmente la gran magnitud del factorial. Esto nos sirve también para reiterar que si alguna vez topamos con este orden de eficiencia desechemos el algoritmo, a no ser que nos aseguremos de que lo vamos a usar para n muy pequeños. Además, algunos de los hallazgos hechos en esta sección nos servirán para estudiar un umbral bueno para nuestro algoritmo Divide y Vencerás que equilibre un tiempo factible y una solución que se acerque lo mejor posible a la solución óptima. Esta discusión se hará en la sección 5.3.4.

5. Algoritmo divide y vencerás

5.1. Problema 1

5.1.1. Descripción del algoritmo

Nuestro problema consiste en estudiar un algoritmo que use la técnica Divide y Vencerás para, dada una secuencia de números enteros cualquiera se obtenga la subsecuencia que tiene suma máxima de todas las subsecuencias posibles. Vamos a dividir la explicación en tres partes.

Caso base

Para estudiar el caso base, hemos considerado que la secuencia que se nos aporta como parámetro `v` cumple que $fin \leqslant 1$, o lo que es lo mismo, $n \leqslant 1$; siendo $I = [init, fin] \cap \mathbb{N}$ el dominio de posiciones del vector que engloba nuestro problema. Por convenio, hemos decidido que se devuelva el par de datos $(init, init)$ indicando que la mayor subsecuencia empieza y acaba en el primer elemento del vector cuya posición esté en I.

División del problema

Tal y como cabría esperar, nos dimos cuenta de que el mejor caso de división sería partir el vector en dos mitades iguales, salvo si `v.size()` es impar ocasionando que una parte del vector tenga un elemento más; no obstante, no ocasiona que haya distinciones notorias entre las llamadas recursivas. Nosotros nos dimos cuenta de que si tomamos alguna otra división de casos obteníamos una eficiencia, en algunos casos parecida, pero normalmente, ya sea por constantes implícitas o por órdenes de eficiencia bastante peor.

[1] $\Gamma(x+1) = x!$ es la extensión del factorial a $\mathbb{R}$.

Combinación de casos

Para la combinación de casos, llegamos a la conclusión de que la solución al problema podía ser una de las tres siguientes:

- La solución propuesta para subproblema de la izquierda.
- La solución propuesta para el subproblema de la derecha.
- Una solución que engloba al dato por el cual hemos partido el vector.

Es claro que, los dos primeros casos pueden ser solución y de hecho la óptima. Un ejemplo de cada uno son los vectores `v={-1,5,6,1,0,-9,-8,7}` y `v'={1,1,1,1,-3,0,0,0}`. Para el tercer caso, tenemos que un ejemplo es `v'={-10,1,1,1,-1,4,0,0}`. Veamos ahora por qué la solución óptima es una de las soluciones descritas en el listado.

- Supongamos que la solución no contiene al punto medio, entonces estamos en uno de los casos anteriores y se tiene lo que se quería.
- Supongamos que la solución contiene al punto medio. Para encontrarla es necesario sumar, punto por punto, desde el punto de medio hasta el primero de los extremos da ambas partes, haciendo una especie de "barrido". Es decir, de la parte izquierda, sumamos todas las posiciones del vector hasta llegar a `bounds_max_izq.second` y, de la parte derecha, lo haremos análogo hasta llegar a `bounds_max_dcha.first`. Una vez llegado a dichas subsecuencias máximas se sumará directamente su valor y se actualizará el límite temporal al otro extremo de la subsecuencia máxima del lado correspondiente.

  Cada vez que uno de los límites de la subsecuencia, y por tanto la suma de la subsecuencia en sí, se actualiza; es necesario comprobar si la suma es mayor que la máxima, actualizándose esta en dicho caso.

Para acabar con la combinación de casos, una vez se obtienen las tres posibles soluciones se compara cuál tiene suma mayor, y se actualiza el par de valores `bounds_max` devolviendo el resultado correcto. Para concluir, como posible solución alternativa desarrollada a partir de esta, se ha añadido la función `suma` a parte del código con el objetivo de dar como resultado ese valor de la suma máxima si el usuario lo prefiere, solo debe cambia el programa principal.

5.1.2. Análisis teórico

En esta sección se va a realizar el análisis teórico del código proporcionado como solución del problema sobre el cálculo de la subsecuencia de suma máxima dentro de una secuencia de números enteros dada. El código implementado con la técnica Divide y Vencerás, el cual se analiza en esta sección, se encuentra en el archivo `Subsecuencia_DyV.cpp`.

Antes de nada, aparece una función auxiliar `suma` encargada de realizar la suma iterativa de una serie de elementos del vector entre unos límites `init` y `fin`. El análisis de esta función es muy sencillo, es un bucle `for` que se ejecuta n veces, siendo $n = fin - init + 1$ el tamaño del problema. Para aclararlo, es como realizar la

suma de una secuencia de números enteros tal que el tamaño de dicha subsecuencia es el número de sus componentes. Por tanto, es de eficiencia lineal.

A continuación, pasamos a realizar el análisis teórico de la función `subsecMax_DyV`, que es realmente lo que nos concierne. Para poder trabajar con comodidad, definimos por $T(n)$ a la función que refleja el tiempo de ejecución de la función citada para un tamaño n.

Es claro que tanto las primeras líneas como el caso base recogido en la primera parte de la estructura condicional que engloba la ejecución de la función, es decir, la parte `if`, son de orden constante y las acotaremos por una constante $a \in \mathbb{R}^+$.

Pasamos ahora a estudiar la parte donde se aplica la división en subproblemas, es decir, la parte `else`. En esta parte, destacamos una serie de líneas donde se aporta el grueso de la eficiencia del problema, el resto de líneas o bloques de código no se citarán por tener eficiencia constante y serán acotadas por una constante $b \in \mathbb{R}^+$. Como podemos ver, nada más empezar a leer el código de dicha parte vemos que se realizan dos llamadas recursivas, con motivo de darle valor a las variables `bounds_max_izq` y `bounds_max_dcha`, cada una de ellas con un tamaño de $n/2$ dando lugar a la división del problema. A continuación se llama dos veces a la función `suma` añadiendo tiempo de ejecución al problema, como ya se comentó, tiene eficiencia lineal.

Tras esto, se realiza la combinación de casos donde; grosso modo, se realiza una nueva ejecución de la función `suma` y se usan dos bucles `while` cuyo tiempo de ejecución es lineal pues se ejecutan; en el caso del primero, $n/2 - bounds\_max\_izq.second$ veces y, en el caso del segundo, $bounds\_max\_izq.first - (n/2 + 1)$ veces.

En conclusión, tenemos que el tiempo de ejecución de la parte `else` viene dado por la ecuación:

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

En definitiva, bastará, por las reglas de la notación O, estudiar la eficiencia de la ley de recurrencia:

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + n & si \quad n > 1 \\ 1 & si \quad n \leqslant 1 \end{cases}$$

Haciendo en la primera ecuación el cambio de variable $n = 2^m$:

$$T(2^m) = 2T(2^{m-1}) + 2^m \qquad \text{si } m > \log_2(1) = 0$$

Renombrando $t_m = T(2^m)$, tenemos:

$$t_m - 2t_{m-1} = 2^m$$

La ecuación característica asociada a la parte homogénea, con $t_m = x^m$, es:

$$x^m - 2x^{m-1} = x^{m-1}(x-2) = 0$$

Como $x^m > 0$ para todo $m \in \mathbb{N}$, tenemos que la única solución de la ecuación característica es $x = 2$, con multiplicidad simple. Respecto a la parte no homogénea, tenemos que $2^m = 2^m(m^0)$, por lo que el polinomio característico de la ecuación en diferencias queda:

$$p(x) = (x-2)(x-2) = (x-2)^2$$

Tamaño	Tiempo (seg)	Tamaño	Tiempo (seg)
50000	0,00199564	530000	0,0228614
70000	0,00283659	550000	0,0255104
90000	0,00367284	570000	0,0295555
110000	0,00500505	590000	0,0257198
130000	0,00545177	610000	0,029649
150000	0,00644428	630000	0,0269512
170000	0,00992647	650000	0,0314426
190000	0,0111087	670000	0,0291709
210000	0,00926159	690000	0,0413729
230000	0,0103055	710000	0,042965
250000	0,0116896	730000	0,0386688
270000	0,0149273	750000	0,034915
290000	0,01471	770000	0,0405849
310000	0,0169635	790000	0,0349807
330000	0,0134414	810000	0,035445
350000	0,0151472	830000	0,0375914
370000	0,0159321	850000	0,0386329
390000	0,0168884	870000	0,0495277
410000	0,0239922	890000	0,0424824
430000	0,0235116	910000	0,0397836
450000	0,0219365	930000	0,04034
470000	0,0205712	950000	0,0429315
490000	0,028404	970000	0,0476641
510000	0,0297836	990000	0,0452864

Tabla 4: Tiempos de ejecución del algoritmo DyV del Problema 1 en el PC de Arturo.

Por tanto, la solución general de la ecuación en diferencias será de la forma:

$$t_m = c_1 \cdot 2^m + c_2 \cdot m \cdot 2^m \qquad c_2, c_2 \in \mathbb{R}$$

Deshaciendo el cambio de variable, $m = \log_2 n$, tenemos que:

$$T(n) = c_1 \cdot 2^{\log_2 n} + c_2 \cdot \log_2 n \cdot 2^{\log_2 n} = c_1 \cdot n + c_2 \cdot n \log_2(n) \in O(n \log n)$$

Por tanto, se tiene que $T(n) \in O(n \log n)$.

5.1.3. Análisis empírico e híbrido

Tras ejecutar el algoritmo específico para tamaños de n desde $5 \cdot 10^4$ a 10^7 con saltos de $2 \cdot 10^4$ y en un rango de $|x| \leqslant 4000$, obtenemos la Tabla 4 que relaciona el tamaño de n con su tiempo de ejecución en segundos. Representamos estos datos en la gráfica de la Figura 4, con ayuda de `gnuplot`.

Como ya hemos estudiado el análisis teórico, sabemos que nos encontramos ante un algoritmo lineal-logarítmico. Buscamos una regresión según la función dada por $f(x) = ax \log x + b$, donde $a, b \in \mathbb{R}$ son constantes que vamos a calcular mediante

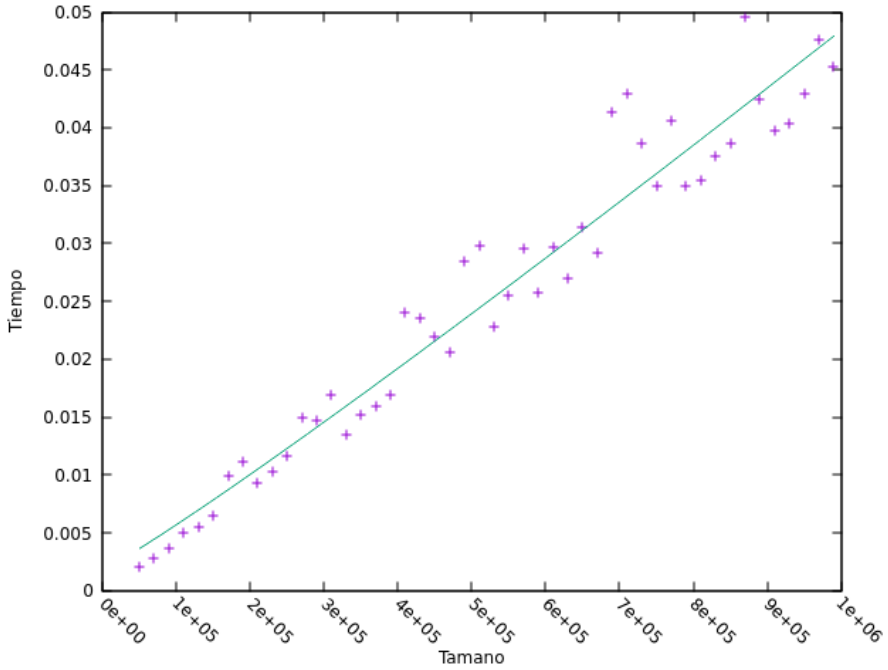


Figura 4: Ajuste de los tiempos del Programa 1 DyV en PC Arturo.

los datos empíricos de la Tabla 4. Con la ayuda de `gnuplot`, en la Figura 4 también se muestra la regresión calculada. La función de regresión es:

$$f(x) = 3{,}37351 \cdot 10^{-9} \cdot x \cdot \log x + 0{,}00179816$$

5.1.4. Cálculo de umbrales

Calcularemos los umbrales siguiendo 3 métodos, el método teórico, el híbrido y el de tanteo.

Umbral Teórico

En este caso, sabemos que el algoritmo iterativo es de orden lineal $O(n)$, mientras que el tiempo del algoritmo Divide y Vencerás viene dado por:

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

Suponiendo que tan solo hay un nivel de recursividad, tenemos que:

$$n_0 = T(n_0) = 2 \cdot \frac{n_0}{2} + n_0 = 2n_0 \Longrightarrow n_0 = 2n_0$$

Por tanto, cabría suponer dos soluciones, $n_0 = 0$ o $n_0 = \infty$. El primer valor no podemos considerarlo, ya que sería que el tamaño del problema es 0, algo irreal. Por tanto, $n_0 = \infty$. Esto tiene sentido, ya que el algoritmo iterativo, de orden lineal, siempre va a ser más rápido, por lo que el Divide y Vencerás no se ejecutará nunca.

Observación. En el caso de que hubiésemos conseguido un algoritmo Divide y Venderás de orden $O(n)$, con la recurrencia $T(n) = 2T\left(n/2\right) + 1$, tendríamos que:

$$n_0 = T(n_0) = 2 \cdot \frac{n_0}{2} + 1 = n_0 + 1 \Longrightarrow n_0 = n_0 + 1$$

De igual forma, llegamos a que $n_0 = \infty$.

Umbral Híbrido

En este caso, se nos pide desarrollar cuál es el umbral de nuestro algoritmo para determinar, a partir de qué valor de $n \in \mathbb{N}$, siendo n el tamaño del problema, es más eficiente en cuanto a tiempo de ejecución ejecutar el algoritmo específico. Para ello, calcularemos el punto de corte de cada una de las funciones obtenidas en cada uno de los análisis híbridos de este problema.

Denotaremos por $f(x)$ y $g(x)$ a cada una de las funciones del algoritmo iterativo y Divide y Vencerás, respectivamente. Resolvemos $f(x) = g(x)$:

$$7{,}60449 \cdot 10^{-10} \cdot x + 4{,}40934 \cdot 10^{-5} = 3{,}37351 \cdot 10^{-9} \cdot x \cdot \log x + 0{,}00179816$$

Usando herramientas como WolfranAlpha o Geogebra, se ve que dicha ecuación no presenta soluciones. Esto tiene sentido, ya que hemos visto que no existe valor real para el umbral, sino que este deberá ser ∞. Esto nos da a entender que no es lógica la presencia de un umbral en este problema pues el algoritmo iterativo siempre es mucho mejor que el algoritmo Divide y Vencerás. Por tanto, como la regresión es un ajuste cerca de los datos con los que hemos trabajado, su comportamiento asintótico no está bien definido. Concluimos entonces que no vamos a tener un umbral válido para nuestro problema.

Tras hacer este razonamiento, lo respaldamos con el hecho de que el algoritmo iterativo es mucho mejor en tiempo, tanto asintótico como fijado un n arbitrario, que el divide y vencerás. Por ende, no tiene sentido implementar un umbral para nuestro algoritmo.

Umbral de Tanteo

En este caso, pierde sentido ejecutar para valores cercanos, ya que no podemos tomar valores cercanos a ∞. No obstante, mostramos en la Tabla 5 que, conforme el umbral aumenta, se reduce el tiempo de ejecución. Esto se ve gráficamente en la Figura 5. Los datos se han tomado para $n = 10^5$. Refuerza la idea de que la implementación del algoritmo Divide y Vencerás, en este caso, no tiene sentido.

5.2. Problema 2

5.2.1. Descripción del algoritmo

El código del algoritmo se encuentra adjunto en el archivo `zip` (`Enlosar.cpp`)

Nuestro problema consiste en embaldosar un suelo con baldosas en forma de L, que se pueden ver como tres cuadrados del mismo tamaño unidos. Por lo tanto, como queremos dejar un hueco libre para nuestro sumidero (que es del mismo tamaño que cada cuadrado del que se compone la baldosa), y el tamaño de la superficie a

Umbral	Tiempo (seg)
1	0,868221
2	0,504466
3	0,482415
4	0,473473
5	0,211082
6	0,176306
7	0,162726

Tabla 5: Tiempos para distintos valores del umbral en PC Arturo.

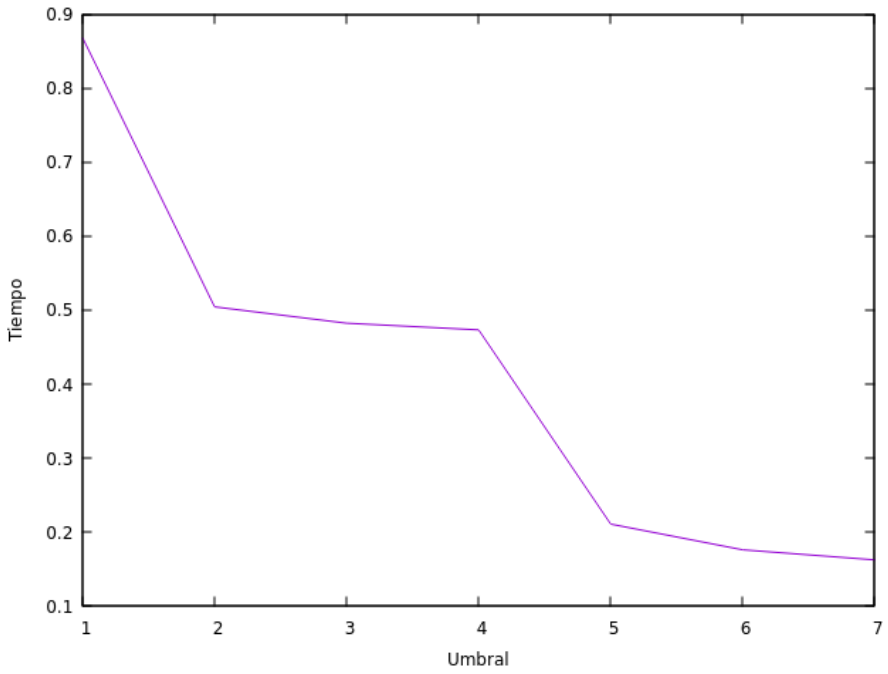


Figura 5: Tiempos para distintos valores del umbral en PC Arturo.

embaldosar compuesta por $2^n \times 2^n$ cuadrados, hemos representado la superficie como una matriz $2^n \times 2^n$.

Para llegar a un algoritmo con la técnica de divide y vencerás, hemos empezado pensando en el caso base (que es el caso de una matriz 2×2), puesto que queríamos buscar una idea que ocurriese en este caso para poder generalizarla en el caso general. En una matriz 2×2, es obvio que solo hay una forma poner las baldosas sin cubrir el sumidero. Si queremos poder utilizar esto en un caso general, para que se asemeje al caso base, deberemos partir nuestra superficie cuadrada en cuatro cuadrados del mismo tamaño; con esto tomamos la decisión de dividir por la mitad de cada eje. Esta forma de dividir es, además, buena para aplicar un algoritmo recursivo, pues tendremos una instancia menor de un problema similar (porque solo hay un sumidero pero hay 4 cuadrados), que después se buscará que sea el mismo problema.

Teniendo en cuenta que partimos en 4 cuadrados, lo siguiente que pensamos es que el sumidero esta solo en uno de ellos, por lo que el cuadrado que contenga al sumidero será la instancia menor del problema a resolver. Entonces podremos aplicar el mismo algoritmo que en el cuadrado anterior una vez encontremos dicho algoritmo (recordemos que estamos viendo el proceso para encontrarlo).

Nos falta ahora saber qué hacer con los tres cuadrados que se quedan fuera. Suponiendo que sabemos resolver una instancia del problema de tamaño menor $(n-1)$, veamos cómo podemos resolver un problema de tamaño n. Si sabemos rellenar un cuadrado con sumidero, pero no uno sin sumidero, buscamos como llevar el segundo caso al primero. Como una baldosa son tres cuadrados de tamaño 1, tenemos tres cuadrados de tamaño $n-1$, y el sumidero es un cuadrado de tamaño uno, parece que si podemos hacer que una baldosa esté en los tres cuadrados de tamaño $n-1$ podremos llevar el segundo caso al primero; pues quedará en cada cuadrado $n-1$ una parte cuadrada de tamaño 1 de la baldosa. Si esto ocurre, interpretaremos cada cuadrado de tamaño 1 de esta baldosa como un sumidero falso, y una vez esto ocurra nos habremos reducido a tres instancias de tamaño $n-1$ del mismo problema, que hemos supuesto que sabemos resolver.

Con todo este razonamiento, ya estamos en condiciones de encontrar un algoritmo bueno. Sea M una matriz de tamaño $2^n \times 2^n$, tenemos una fila y columna en la que no escribir. Determinamos si la casilla está en la parte superior o en la parte inferior de la matriz ($fila \geqslant 2^{(n-1)}$). A su vez determinamos con el mismo criterio aplicado a la columna si la baldosa está en el lado izquierdo o en el lado derecho.

Ahora,partimos la matriz en cuatro submatrices de tamaño $2^{(n-1)} \times 2^{(n-1)}$, de forma de que no se solapen las submatrices. Localizaremos en que submatriz está el sumidero. En la esquina en la que las 4 matrices se tocan se pondrá una baldosa que cubra la esquina que toca a las otras tres de cada submatriz que no tiene el sumidero. Se interpretará en cada submatriz el cuadrado que se ha añadido como un sumidero falso. Entonces, tendremos 4 submatrices con lo que interpretamos como un sumidero cada una, luego podemos repetir el algoritmo en las submatrices repetidas veces para solucionar el problema en las submatrices. La condición de parada viene dada por el tamaño de la submatriz; cuando se llegue a una submatriz 2×2, se colocará una baldosa de forma que no cubra el sumidero.

5.2.2. Análisis teórico

En este problema, tenemos que la superficie a recubrir por baldosas es siempre un cuadrado que por lado tiene a una potencia de 2. Por lo tanto, nuestro problema de tamaño n tendrá un tamaño de $2^n \cdot 2^n = 4^n$.

Debido a esto, no podemos esperarnos encontrar una solución que de menos de $4^n - 1$ pasos, pues debemos escribir en cada celda de la matriz el número de la baldosa que se ha usado para indicar la solución al problema.

Además de esto, es poco intuitivo hablar sobre la eficiencia de un algoritmo exponencial por naturaleza, ya que su rápido crecimiento nos impide realmente visualizar cuan bueno es un algoritmo. Por esto, hemos decidido analizar su eficiencia en una escala logarítmica. Si $T(n)$ es la función que modela el tiempo de ejecución de nuestro algoritmo, nosotros estudiaremos la función $F(n) := log_4(T(n))$ una vez conozcamos T, ya que esta modela mejor el problema si queremos ver el crecimiento del tiempo de ejecución respecto el tamaño del cuadrado. De esta forma se verá como de buena es realmente nuestra solución.

Dicho esto, tenemos que $T(n) = 4T(n-1) + 1$, ya que n es el exponente que utilizamos en nuestro código para conseguir el tamaño del cuadrado y lo reducimos en una unidad por cada llamada recursiva. Por otro lado, el resto de funciones y operaciones usadas son todas de orden constante, pues solo se hacen sumas, accesos a posiciones de la matriz y operaciones booleanas. Sabido esto, podemos en este sencillo caso usar el desarrollo en serie de nuestra T para averiguar su orden de eficiencia.

$$T(n) = 4T(n-1) + 1 = 16T(n-2) + 4 + 1 = 64T(n-3) + 16 + 4 + 1 = ...$$
$$= 4^{n-1}T(1) + \sum_{k=0}^{n-2} 4^k = 4^{n-1} + \frac{4^{n-1} - 1}{3}$$

en donde en la última igualdad aplicamos la suma de una progresión geométrica y que en $n = 1$ llegamos al caso base, con $T(1) = 0$.

A la vista de esto, tenemos que $T \in O(4^n)$, pero entonces tenemos que la función F antes definida es de orden lineal. Por lo tanto, deducimos que nuestro algoritmo es bastante bueno ya que F nos dice como aumenta el tiempo de ejecución respecto el tamaño de nuestro cuadrado. Si por otra parte tomamos n como el número de celdas de una fila de la matriz, tenemos que el algoritmo es cuadrático, razonando de manera similar.

Informalmente, se puede apreciar algoritmo es teóricamente óptimo, puesto que en todas las celdas de la matriz escribimos una única vez menos en la del sumidero, realizamos operaciones de orden constante para poder escribir bien las baldosas. Como el problema requiere que escribamos en 4^n casillas, es imposible bajar de $O(4^n)$, que es el orden de nuestro algoritmo, tomando n como el número que nos da el lado de tamaño 2^n. Por tanto, el orden de eficiencia de este algoritmo es el *número de casillas* que tenga la matriz.

5.2.3. Análisis empírico e híbrido

Ahora en esta sección llamaremos n al numero de celdas que hay en una fila. Entonces, según esta n, el análisis teórico nos dice que el algoritmo es cuadrático,

Tamaño	Tiempo(seg)
8	$3{,}98 \cdot 10^{-7}$
16	$1{,}292 \cdot 10^{-6}$
32	$2{,}94 \cdot 10^{-6}$
64	$8{,}356 \cdot 10^{-6}$
128	$4{,}504 \cdot 10^{-5}$
256	0,000121097
512	0,000481561
1024	0,00203506
2048	0,00702923
4096	0,0290864
8192	0,120917
16384	0,493566
32768	2,04945

Tabla 6: Tiempos de la ejecución del programa 2 DyV en PC Airam.

pues si tomas m como el número de casillas de la matriz tenemos que $m = n^2$.

Hemos ejecutado instancias desde 2^3 celdas por fila hasta 2^{15} celdas por fila. Los resultados nos indican que el algoritmo se comporta como uno cuadrático, pues su dato de entrada es el tamaño de la fila, como habíamos razonado anteriormente. A continuación mostramos los datos recogidos en la Tabla 6 y el ajuste realizado en la gráfica de la Figura 6. La función que hemos obtenido para la regresión es:

$$f(x) = 1{,}73076 \cdot 10^{-9}x^2 + 1{,}02065 \cdot 10^{-6}x - 0{,}000237051$$

5.3. Problema 3

5.3.1. Descripción del algoritmo

Dado un conjunto de puntos sobre el plano, trataremos de encontrar el ciclo (camino que parta de un punto, recorra todos los puntos que une una única vez y vuelva al punto de inicio) de menor longitud que una a todos los puntos. Debido a la complejidad del problema, sabemos que no obtendremos un algoritmo óptimo. Procedemos por tanto, a buscar una solución que ofrezca resultados no muy dispa-ratados a la hora de buscar un camino que una todos los puntos tratando que la distancia recorrida no sea muy elevada, mediante la técnica Divide y Vencerás.

La entrada al algoritmo es un conjunto de n puntos (hemos considerado enteros positivos, por simplicidad), mientras que la salida es un conjunto de $n+1$ puntos de forma ordenada, donde el primer punto es el punto de partida, el punto que precede al $i-$ésimo (el $(i+1)-$ésimo) es el punto al que nos tendremos que dirigir una vez que nos encontremos en el punto $i-$ésimo (para todo i entre 1 y n); y el último punto coincide con el primero (ya que volvemos a la ciudad de inicio).

Caso base

Como caso base, consideramos que el número de puntos (n) sobre el plano es menor o igual que 3. En dicho caso, todas las combinaciones de n puntos (con

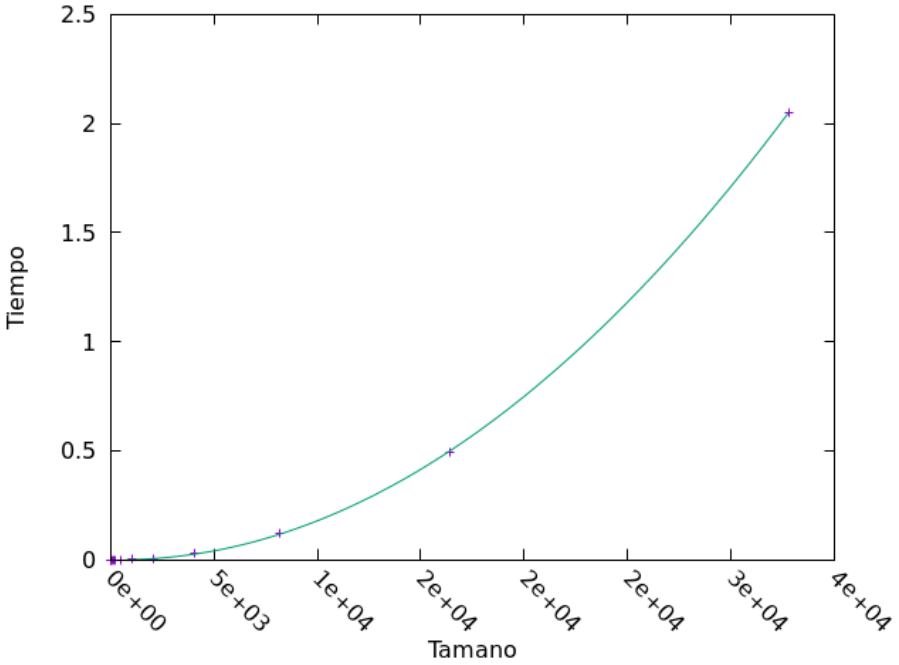


Figura 6: Ajuste de los tiempos del Programa 2 DyV en PC Airam.

$0 \leqslant n \leqslant 3$) y vuelta al punto de inicio tendrán la misma longitud, por lo que no es relevante qué camino escoger.

Cabe destacar que a la hora de implementar el algoritmo, para tamaños no muy grandes (por debajo de un cierto umbral), lo que haremos será ejecutar el algoritmo que encuentra la solución óptima por fuerza bruta (el que de todos los caminos, seleccione el de menor longitud); para reducir los tiempos de ejecución cuando n sea lo suficientemente pequeño. Es decir, nuestro umbral no será de 3, pero para considerar de forma teórica el algoritmo Divide y Vencerás puro, hablamos de caso base cuando $n \leqslant 3$.

División del problema

Dados n puntos sobre el plano, los ordenamos por una de sus coordenadas (por ejemplo, x) y seleccionamos el punto del vector que se encuentre en la posición $n/2$. Este punto recibirá el nombre de `pivote`. A continuación, disponemos de dos conjuntos de puntos: el conjunto `Izquierda`, formado por aquellos puntos que se encuentren en una posición del vector menor o igual que la del `pivote` (notemos que `pivote` $\in$ `Izquierda`); y el conjunto `Derecha`, formado por aquellos puntos que se encuentren en una posición del vector mayor o igual que la del `pivote` (notemos que `pivote` $\in$ `Derecha`). Una vez realizada la división, obtendremos la resolución al problema en los dos conjuntos, el menor camino en `Izquierda` y el menor en `Derecha`. Estos dos parten de `pivote` y llegan hasta `pivote` (es decir, `pivote` es el primer y último punto en el vector solución de ambos caminos).

Para mayor eficacia del algoritmo (encontrar caminos más cortos), cuando lo ejecutemos de forma recursiva, alternaremos entre la ordenación por x y por y (por ejemplo, comenzamos ordenando por x, luego ordenamos por y, seguimos

por x, ..., hasta llegar al caso base). De esta forma, cuando ordenamos por y y mencionemos a los conjuntos `Izquierda` y `Derecha`, podemos renombrarlos respectivamente como `Abajo` y `Arriba`.

Combinación de casos

Una vez tengamos el camino más corto que parte del `pivote` y vuelve a él en `Izquierda` y en `Derecha` (para esto, hemos decido que las funciones que resuelven el Viajante del Comercio por Divide y Vencerás y por el específico sean estables en relación al primer punto. Es decir, que el primer punto del vector pasado como argumento a la función sea el primer punto del que se parte y el último al que se llega en el recorrido), procedemos a explicar cómo obtener la solución al problema.

Para ello, debemos antes prefijar una notación: en `Izquierda`, habrá dos puntos (llamémoslos a y b) que serán el segundo (al que se llega al partir de `pivote`) y el penúltimo (el punto que precede a volver al `pivote`) en el vector solución del problema de `Izquierda`. Es decir, estos dos puntos estarán unidos con el `pivote`. De forma análoga, en `Derecha`, tendremos dos puntos que llamaremos c y d que son los que se encuentran unidos al `pivote`:

$$\begin{aligned} \text{Solución de Izquierda:} &\quad \{pivote, a, \ldots, b, pivote\} \\ \text{Solución de Derecha:} &\quad \{pivote, c, \ldots, d, pivote\} \end{aligned}$$

Por estar unidos al pivote, tendremos que se encuentran de alguna forma "cerca" de este, que a su vez es el punto medio entre los de `Izquierda` y los de `Derecha`. De esta forma, podemos esperar en la mayoría de casos que a y b no se encuentren muy lejos de `Derecha` y que c y d tampoco se encuentren muy lejos de `Izquierda`. Una vez tenemos estos 4 puntos (y el `pivote`), buscamos una forma de unir el ciclo de `Izquierda` con el de `Derecha`. Tenemos 4 posibilidades:

1. Partir del `pivote`, pasar al punto a, recorrer el resto del conjunto `Izquierda` por el camino ya resuelto hasta llegar el punto b, que enlace con el punto c, recorrer el conjunto `Derecha` hasta llegar al punto d y que este enlace con el `pivote`. Representamos a esta posibilidad de forma esquemática como:
$$p \to a \xrightarrow{\cdots} b \to c \xrightarrow{\cdots} d \to p$$
2. Unir `Izquierda` con `Derecha` pasando de b a d:
$$p \to a \xrightarrow{\cdots} b \to d \xrightarrow{\cdots} c \to p$$
3. Unir `Izquierda` con `Derecha` pasando de a a c:
$$p \to b \xrightarrow{\cdots} a \to c \xrightarrow{\cdots} d \to p$$
4. Unir `Izquierda` con `Derecha` pasando de a a d:
$$p \to b \xrightarrow{\cdots} a \to d \xrightarrow{\cdots} c \to p$$

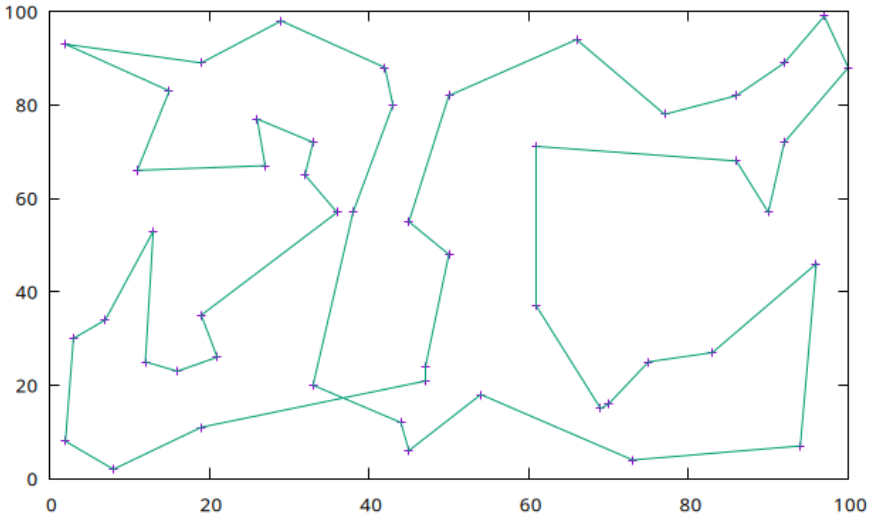


Figura 7: Ejemplo de la salida para el Problema del Viajante del Comercio.

De estas 4 posibilidades, calculamos el coste de todos estos caminos, quedándonos con el que nos dé la menor longitud. Notemos que el camino $a \overset{\cdots}{\longrightarrow} b$ se recorre en todos los casos (si no, tomar $b \overset{\cdots}{\longrightarrow} a$, de igual longitud), al igual que $c \overset{\cdots}{\longrightarrow} d$ (con la misma salvedad), por lo que este costo no lo incluiremos y sólo calcularemos por ejemplo, en el caso de la primera opción:

$$L_1 = d(p, a) + d(b, c) + d(d, p)$$

Análogamente, se definirá L_i para las otras tres posibilidades.

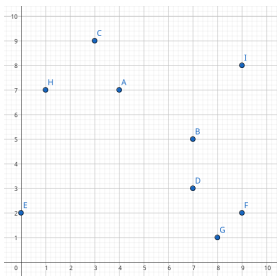
Un ejemplo de la salida que generará este programa se encuentra disponiblen en la Figura 7, donde se ha usado $n = 50$ ciudades y el umbral 8.

Ejemplo

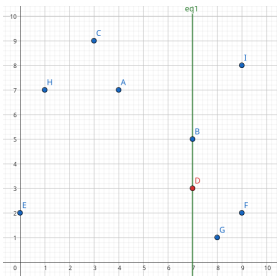
Un ejemplo de la ejecución del algoritmo la vemos con la siguiente instancia (`umbral = 3`, para centrarnos en el algoritmo) de la Tabla 7. Hemos marcado a los pivotes en rojo.

x	4	7	3	7	0	9	8	1	9
y	7	5	9	3	2	2	1	7	8

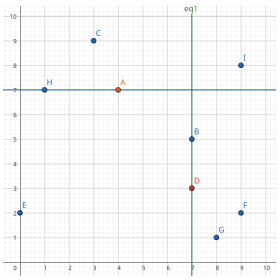
Tabla 7: Puntos empleados en el ejemplo del DyV del Programa 3.



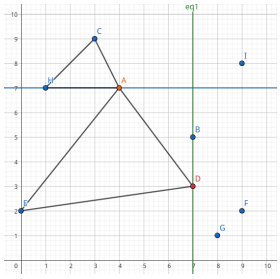
(a) Puntos a unir



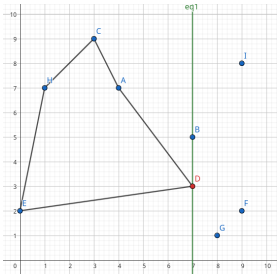
(b) División horizontal



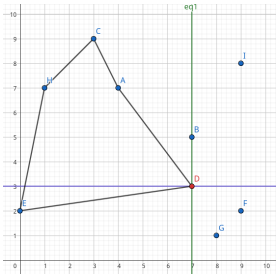
(c) División vertical



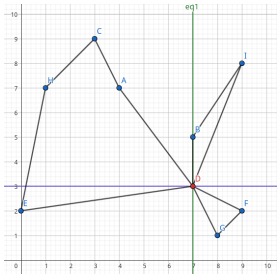
(d) Caso base



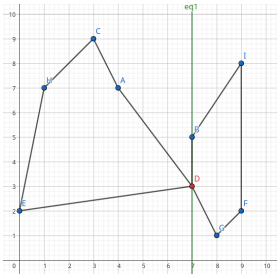
(e) Combina



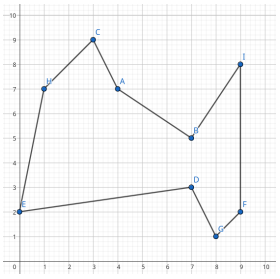
(f) División vertical



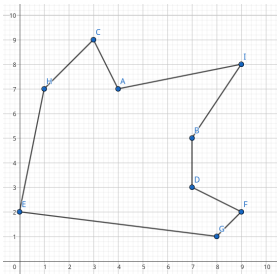
(g) Caso base



(h) Combina



(i) Combina y solución



(j) Solución óptima

Obteniendo en la Figura 8j una distancia de 32,5 y en la Figura 8i de 34. Podemos ver que para este caso en concreto no obtenemos una solución muy

disparatada, por lo que podríamos pensar que nos dará una solución aceptable para tamaños mayores.

5.3.2. Análisis teórico

Para el análisis teórico de la función `mejor_caminoDyV`, primero calculamos el orden del resto de funciones que esta usa (entendiendo que $n = sup - inf + 1$):

dist Se trata de una función que realiza operaciones con dos pares de enteros. Por tanto, es del orden $O(1)$.

mejor_camino_Iterativo Sabemos por el análisis teórico del algoritmo específico que tiene eficiencia $O(n!)$.

donde_romper Se trata de una función que calcula cuatro distancias entre 3 puntos luego devuelve un par de valores en función de cual de esas cuatro distancias es la mínima. Tiene un orden de eficiencia $O(1)$.

une_caminos En el mejor caso, las dos condiciones de los `if` serán falsas y tendremos un orden de $\Omega(1)$. En el peor caso, las dos condiciones de los `if` serán ciertas y se ejecutarán las dos llamadas a la función `reverse` de `algorithm`, de orden lineal, luego obtenemos una eficiencia en cada estructura condicional del orden $O\left({}^{n}/_{2}\right)$, obteniendo un orden de la función en general de:

$$O\left({}^{n}/_{2}\right) + O\left({}^{n}/_{2}\right) \in O(n)$$

ordena_y Se trata de un functor que establece el orden de los pares de enteros por la segunda coordendada. Por tanto, $O(1)$.

ordena Se trata de una función que, dependiendo de un valor booleano, ordena un vector de $n = sup - inf + 1$ elementos por la primera coordenada o por la segunda, haciendo uso de la función `sort` de `algorithm`, de orden $O(n \log n)$. Por tanto, nuestra función tendrá el mismo orden: $O(n \log n)$.

rota Llama a la función `rotate` de `algorithm`, de orden $O(n)$, luego nuestra función tiene orden $O(n)$.

busca Busca un elemento en un vector desordenado por búsqueda lineal, luego tiene orden de eficiencia $O(n)$.

Ahora sí, nos disponemos a evaluar el orden de eficiencia de la función que hemos llamado `mejor_caminoDyV`.

- Si $n \leqslant UMBRAL\_ITERATIVO$, entonces se ejecuta la función denominada `mejor_camino_Iterativo`, de orden $O(n!)$. Sin embargo, como estamos acotando el valor de n, tendrá una eficiencia teórica de $O(1)$.
- Si $n > UMBRAL\_ITERATIVO$, analizamos la eficiencia:
  - Las tres primeras líneas de código almacenan el pivote anterior, ordenan el vector y seleccionan el pivote a utilizar. Tienen un orden de eficiencia de $O(n \log n)$.

- Posteriormente, se resuelve el problema en la parte derecha, añadiendo un tiempo de $T(n/2)$ si $T(n)$ es la función que nos da el tiempo de ejecución de nuestra función.
- A continuación, se intercambian dos posiciones en el vector, se resuelve el problema en la parte de la izquierda, y se llama a la función `reverse` con la mitad del vector, añadiendo un tiempo de $T(n/2) + n/2$.
- Se llama a las funciones que se encargan de combinar los casos: `donde_romper` y `une_caminos`, de orden $O(1)$ y $O(n)$, luego añadimos un tiempo de n.
- Finalmente, se llama a las funciones `busca` y `rota`, añadiendo tiempos de $n + n = 2n$.

Obtenemos así un tiempo de ejecución de:

$$T(n) = \begin{cases} 1 & \text{si} \quad n \leqslant UMBRAL\_ITERATIVO \\ 2T(n/2) + n\log n + n/2 + 3n & \text{si} \quad n > UMBRAL\_ITERATIVO \end{cases}$$

De forma que:

$$T(n) = \begin{cases} 1 & \text{si} \quad n \leqslant UMBRAL\_ITERATIVO \\ 2T(n/2) + n\log n & \text{si} \quad n > UMBRAL\_ITERATIVO \end{cases}$$

Buscamos por tanto, resolver la recurrencia:

$$T(n) = 2T(n/2) + n\log n$$

Para ello, realizamos el cambio de variable $n = 2^k$:

$$T\left(2^k\right) = 2T\left(2^{k-1}\right) + 2^k \log 2^k = 2T\left(2^{k-1}\right) + k2^k \tag{1}$$

Donde la homogénea asociada es:

$$T\left(2^k\right) = 2T\left(2^{k-1}\right)$$

De ecuación característica homogénea $x - 2 = 0$ y por tanto, de solución (para la homogénea):

$$x_k^{(h)} = c \cdot 2^k \qquad c \in \mathbb{R}$$

Además, por tener en la parte no homogénea $k \cdot 2^k$, sabemos que una solución particular de la Ecuación 1 es un polinomio de grado $\deg(k) = 1$ (como $a + bk$, para ciertos $a, b \in \mathbb{R}$) por 2^k por k (por ser la base de 2^k una raíz de la ecuación característica homogénea):

$$x_k^{(p)} = k \cdot 2^k(a + bk)$$

Finalmente, como sabemos que todas las soluciones de la Ecuación 1 son de la forma una particular más una solución homogénea, tenemos que todas las soluciones son de la forma:

$$x_k = c \cdot 2^k + k \cdot 2^k(a + bk) = 2^k(c + a \cdot k + b \cdot k^2)$$

Para ciertos $a, b, c \in \mathbb{R}$. Deshaciendo el cambio de variable $k = \log_2 n$, llegamos a:

$$x_n = 2^{\log_2 n}(c + a \cdot \log_2 n + b \cdot \log_2^2 n) = n(c + a \cdot \log_2 n + b \cdot \log_2^2 n)$$

Por lo que tendremos que $T(n) \in O(n \log^2 n)$.

n	Tiempo (seg)	n	Tiempo (seg)
10000	0,0040595	460000	0,335775
40000	0,0204162	490000	0,357599
70000	0,0388603	520000	0,384785
100000	0,0588425	550000	0,409239
130000	0,0790486	580000	0,437235
160000	0,100793	610000	0,461745
190000	0,122837	640000	0,486231
220000	0,144389	670000	0,516803
250000	0,166476	700000	0,542357
280000	0,189905	730000	0,569053
310000	0,212849	760000	0,595847
340000	0,238198	790000	0,621799
370000	0,263302	820000	0,649758
400000	0,287999	850000	0,676105
430000	0,310853	880000	0,700278

Tabla 8: Tiempos Viajante de Comercio Divide y Vencerás en PC José

5.3.3. Análisis empírico e híbrido

Como se ha estudiado en la sección correspondiente, estamos tratando con un algoritmo linear-logarítmico. Por tanto, para estudiar la eficiencia empírica e híbrida vamos a tomar una gran cantidad de puntos. En concreto, estudiaremos el algoritmo para $n \in [10^4, 9 \cdot 10^5]$ con saltos de $3 \cdot 10^4$. Ejecutamos nuestro algoritmo para los casos descritos y obtenemos los tiempos de ejecución reflejados en la Tabla 8, que podemos ver gráficamente en las Figuras 9 y 10, que son la dispersión de puntos y el ajuste respectivamente.

El ajuste obtenido es:

$$f(x) = 4{,}27 \cdot 10^{-9} x \log^2 x$$

5.3.4. Cálculo de umbrales

Umbral teórico

El umbral teórico del algoritmo lo calculamos de la siguiente manera:

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + n\log^2(n) & si \quad n > n_0 \\ n! & si \quad n \leqslant n_0 \end{cases}$$

Se puede ver que es complicado encontrar una solución entera, y realizar un ajuste en los números reales, aunque posible, es muy laborioso debido a la necesidad de trabajar con la función Gamma. Por lo tanto, Aunque no sepamos en dónde está la solución, como n al final será un número natural, podemos calcular entre que dos valores se cortan las gráficas y tomar el entero mas cercano por la derecha a la solución. Para demostrar que la solución está entre 2 valores, utilizaremos el Teorema de Bolzano.

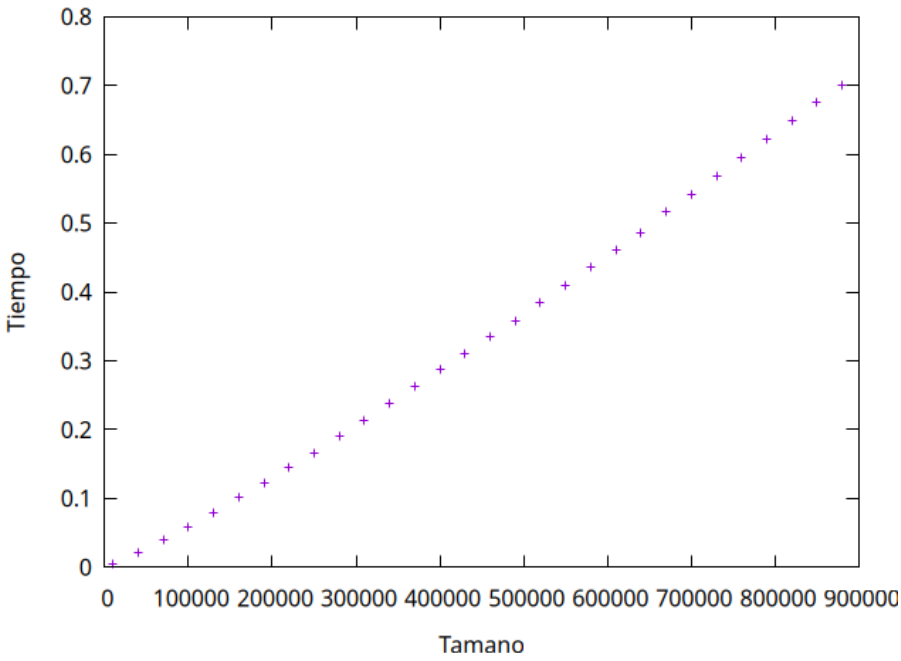


Figura 9: Tiempos de ejecución Viajante de Comercio DyV en PC José Juan.

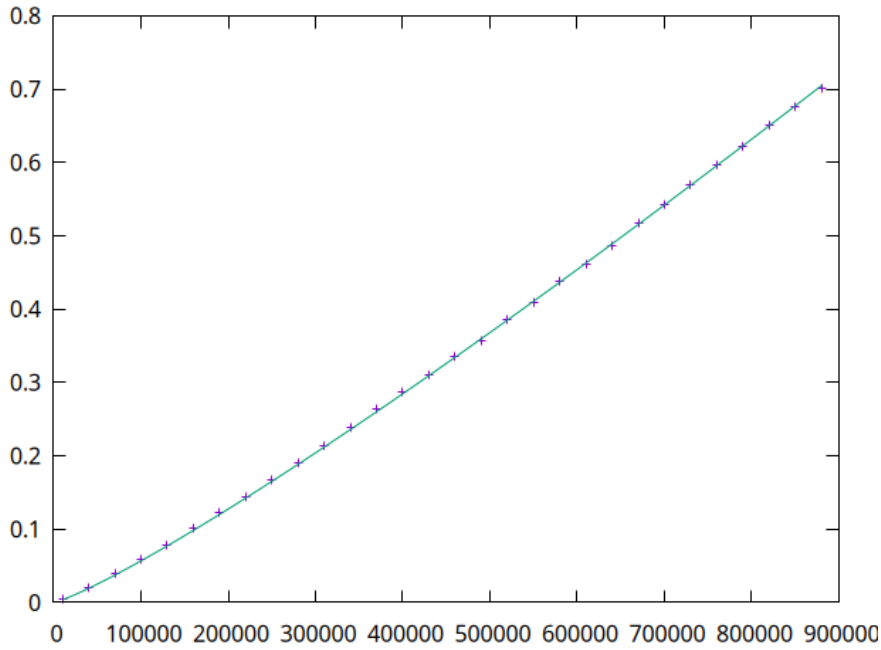


Figura 10: Regresión del Viajante de Comercio DyV en PC José Juan.

Definimos las siguientes funciones, entendiendo por $x!$ como una notación de la función gamma para recordar que queremos generalizar al factorial:

$$\begin{aligned} f(x) &= x! \\ g(x) &= 2(x/2)! + x \ln x \\ h(x) &= g(x) - f(x) \end{aligned}$$

Buscar $g(x) = f(x)$ se reduce ahora a encontrar las raíces de h. Con la ayuda de una calculadora, obtenemos que $h(2) \approx 1{,}39$ y que $h(3) \approx -0,05$. Entonces, por el Teorema de Bolzano se tiene que h tiene una raíz en el intervalo $]2, 3[$. Por lo tanto, tomamos $n = 3$ como nuestro umbral teórico.

Umbral óptimo

Gracias a los estudios empíricos de los algoritmos específico y Divide y Vencerás, tenemos que las curvas de regresión de las nubes de puntos de los tiempos vienen dadas por las funciones:

$$\begin{aligned} f(x) &= 2{,}04175 \cdot 10^{-9} x! && \text{Para el algoritmo específico} \\ g(x) &= 4{,}27 \cdot 10^{-9} x \log^2 x && \text{Para el algoritmo Divide y Vencerás} \end{aligned}$$

Calculando el punto de corte de estas dos funciones, obtenemos que se cortan en el punto $(0{,}3064258, 1{,}18 \cdot 10^{-9})$. Al trabajar con valores discretos para el umbral, decidimos por tanto escoger el primer $n \in \mathbb{N}$ tal que $n > 0{,}3064258$, es decir, tendremos un umbral óptimo de 1.

Umbral de tanteo

Calculando el umbral teórico y óptimo hemos visto que no sale rentable usar el algoritmo específico en ningún caso. No obstante, en este problema tenemos que tener en cuenta que el algoritmo DyV nos dará una solución mejor si subimos el umbral. Así que nuestra primera tarea será ver cuándo esta rapidez empieza a merecer realmente la pena, cuándo aplicar el algoritmo específico buscando una solución mejor empieza a costarnos más tiempo del que nos gustaría. En la Tabla 9 y en los gráficos de las Figuras 11 y 12 podemos ver la comparativa de tiempos entre el algoritmo específico y el algoritmo Divide y Vencerás. Podemos observar que a partir de 9 ya se separa la gráfica del específico. Así que vamos a considerar el 8 como cota, puesto que nos puede dar mucho mejores resultados a nivel de buscar el resultado óptimo.

Vamos a probar entonces con umbrales desde 3 (es el caso base sin usar el algoritmo específico) a 8. Hemos obtenido las Tablas 10 y 11, con dos instancias distintas:

Tamaño	Específico	DyV
4	$1{,}844 \cdot 10^{-6}$	$2{,}445 \cdot 10^{-6}$
5	$2{,}084 \cdot 10^{-6}$	$3{,}257 \cdot 10^{-6}$
6	$4{,}509 \cdot 10^{-6}$	$3{,}256 \cdot 10^{-6}$
7	$1{,}5981 \cdot 10^{-5}$	$5{,}1 \cdot 10^{-6}$
8	0,000105574	$4{,}018 \cdot 10^{-6}$
9	0,000760437	$4{,}128 \cdot 10^{-6}$
10	0,00748786	$3{,}677 \cdot 10^{-6}$
11	0,0817935	$4{,}358 \cdot 10^{-6}$

Tabla 9: Comparación del tiempo del Específico y el DyV (Problema 3) en PC Irina.

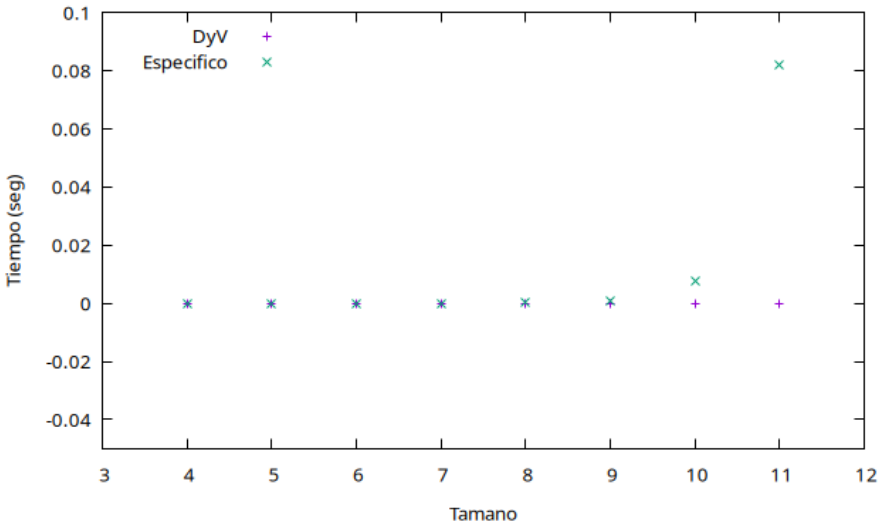


Figura 11: Nube de puntos del Específico y el DyV (Problema 3) en PC Irina.

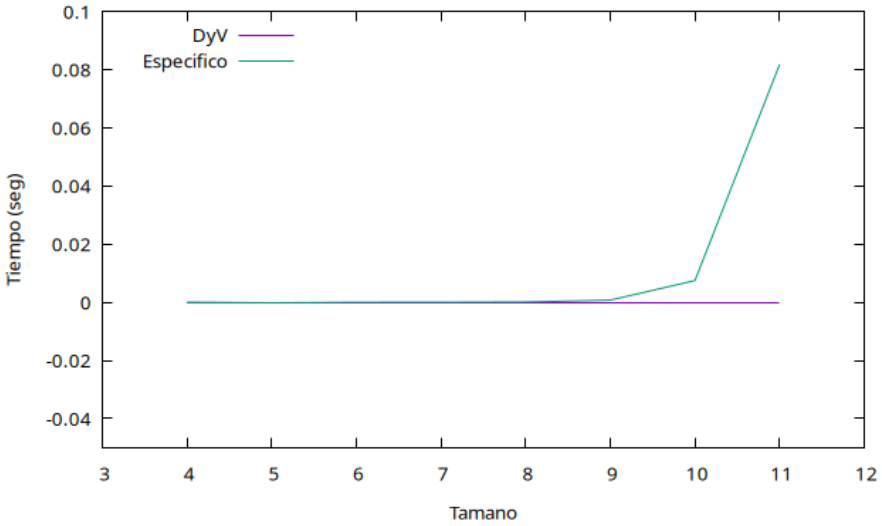


Figura 12: Comparación del Específico y el DyV (Problema 3) en PC Irina.

Umbral	Tiempo (seg)	Distancia
3	0,063515	$6{,}74844 \cdot 10^7$
4	0,0734879	$6{,}72127 \cdot 10^7$
5	0,0714458	$6{,}71728 \cdot 10^7$
6	0,0728542	$6{,}71728 \cdot 10^7$
7	0,223619	$6{,}27753 \cdot 10^7$
8	0,380699	$6{,}20762 \cdot 10^7$

Tabla 10: Tiempo y distancia según el umbral para $n = 10^5$ en PC José Juan.

Umbral	Tiempo (seg)	Distancia
3	0,076296	$6{,}56096 \cdot 10^7$
4	0,071646	$6{,}53803 \cdot 10^7$
5	0,0692665	$6{,}53803 \cdot 10^7$
6	0,0643691	$6{,}52962 \cdot 10^7$
7	0,232349	$6{,}06217 \cdot 10^7$
8	0,239041	$6{,}06217 \cdot 10^7$

Tabla 11: Tiempo y distancia según el umbral para $n = 97573$ en PC José Juan.

Observando los tiempos y las distancias notamos que, a menor umbral, obtenemos mejor tiempo; y a mayor umbral, obtenemos mejor distancia. Entonces, no se puede llegar a ninguna conclusión definitiva. Depende de lo que queramos priorizar bajaremos o subiremos el umbral.

6. Conclusiones

Hemos podido observar cómo la técnica de resolución de problemas mediante la técnica Divide y Vencerás puede ayudarnos a resolver ciertos problemas. Por ejemplo, resolver el Problema 2 mediante una técnica que no sea divide y vencerás requiere un cierto trabajo, mientras que por la técnica de Divide y Vencerás puede resolverse fácilmente. Además, hemos podido comprobar que aunque esta técnica no resuelva de forma óptima un problema (como el problema del Viajante de Comercio), sí puede darnos una primera aproximación a él.

Además, hemos aprendido una nueva forma de pensar, basada en la deconstrucción de un problema en subproblemas mas pequeños, mientras mantenemos una mentalidad computacional. Con esto, hemos aprendido a pensar en formas de bajar un tiempo de ejecución aplicando esta técnica, lo cual puede ser muy útil cuando se pretenda desarrollar un algoritmo.

No obstante, también hemos visto los problemas que tienen los algoritmos en algunas ocasiones, en las que no es viable aplicar un algoritmo divide y vencerás, como es el caso del primer ejercicio propuesto. Esto nos ha hecho pensar y hemos llegado a la conclusión de que por muy buena que sea una técnica de programación, si se intenta aplicar en un problema en el que no es adecuado o se aplica de forma incorrecta puede llegar a dar soluciones en tiempos de ejecución mayores.

Con esto, llegamos a la conclusión de que la Técnica de Divide y Vencerás es una poderosa herramienta a tener en cuenta cuando se diseña un algoritmo; pero como cualquier herramienta, debe ser utilizada cuando sea correspondiente y de forma adecuada al problema. De lo contrario, lo único que conseguiremos será un algoritmo ineficiente.