Memoria Práctica 2: Algoritmos divide y vencerás

Laura Mandow Fuentes
Chengcheng Liu
Daniel Hidalgo Chica
Roberto González Lugo
Elías Monge Sánchez

Marzo 2024

Titulación: Doble Grado de Matemáticas e Ingeniería Informática
Curso: 2º, 2º Cuatrimestre
Asignatura: Algorítmica

Correos:

- Laura Mandow Fuentes: e.lauramandow@go.ugr.es
- Roberto González: e.roberlks222@go.ugr.es
- Chengcheng Liu: e.cliu04@go.ugr.es
- Elías Monge: e.eliasmonge234@go.ugr.es
- Daniel Hidalgo Chica: e.danielhc@correo.ugr.es

Índice

1. Participación

- Laura Mandow Fuentes e.lauramandow@go.ugr.es 100 %
- Roberto González Lugo e.roberlks222@go.ugr.es 100 %
- Daniel Hidalgo Chica e.danielhc@go.ugr.es 100 %
- Chengcheng Liu e.cliu04@go.ugr.es 100 %
- Elías Monge Sánchez e.eliasmonge234@go.ugr.es 100 %

1.1. Participación específica

Aunque hayamos trabajado cada uno de forma global los contenidos de la práctica, a la hora de la redacción de la memoria, el trabajo se ha visto dividido en partes de carga de trabajo similar con el fin de aumentar la productividad.
En particular, las máquinas utilizadas para ejecutar los algoritmos son:

- P1
  - Específico:
    - Máquina: Asus TUF fx505dt
    - Procesador: AMD Ryzen 7 3750h with Radeon Vega Mobile Gfx 2.3GHz
    - Tarjeta gráfica: Nvidia Geforce GTX 1650
    - Sistema Operativo: Arch Linux 64bits
  - DyV
    - Máquina: Surface Laptop 4
    - Procesador: Intel Core i7
    - Tarjeta Gráfica: Intel Corporation TigerLake-LP GT2 [Iris Xe Graphics] (rev 01)
    - Sistema Operativo: Ubuntu 22.04 64bits
  - Umbrales
    - Máquina: Asus TUF fx505dt
    - Procesador: AMD Ryzen 7 3750h with Radeon Vega Mobile Gfx 2.3GHz
    - Tarjeta gráfica: Nvidia Geforce GTX 1650
    - Sistema Operativo: Arch Linux 64bits
- P2
  - Específico
    - Máquina: Acer Aspire A315-42
    - Procesador: Procesador: AMD Ryzen 5 3500U 2.10 GHz
    - Tarjeta Gráfica: Radeon Vega Mobile Gfx
    - Sistema Operativo: Ubuntu 22.04 64bits (Oracle VM VirtualBox, 2 cores)
  - DyV
    - Máquina: Acer Aspire A515-45
    - Procesador: AMD Ryzen 5 5500U
    - Tarjeta Gráfica: AMD Radeon Graphics 7 cores, 1800 MHz
    - Sistema Operativo: Ubuntu 22.04 64 bits
  - Umbrales
    - Máquina: Acer Aspire A315-42
    - Procesador: Procesador: AMD Ryzen 5 3500U 2.10 GHz
    - Tarjeta Gráfica: Radeon Vega Mobile Gfx
    - Sistema Operativo: Ubuntu 22.04 64bits (Oracle VM VirtualBox, 2 cores)

- P3
  - Específico
    - Máquina: HP Laptop 15s-eq1xxx
    - Procesador:AMD Ryzen 5 4500U with Radeon Graphics
    - Sistema Operativo: Ubuntu 22.04 64 bits
  - DyV
    - Máquina: Acer Aspire A515-45
    - Procesador: AMD Ryzen 5 5500U
    - Tarjeta Gráfica: AMD Radeon Graphics 7 cores, 1800 MHz
    - Sistema Operativo: Ubuntu 22.04 64 bits
  - Umbrales
    - Máquina: HP Laptop 15s-eq1xxx
    - Procesador:AMD Ryzen 5 4500U with Radeon Graphics
    - Sistema Operativo: Ubuntu 22.04 64 bits

2. Objetivos

Los objetivos de nuestra práctica se centran en comprender y asimilar de manera profunda y sistemática la metodología Divide y Vencerás, aplicada al diseño y desarrollo de algoritmos complejos. Esta metodología, que se basa en descomponer un problema en subproblemas de menor tamaño, más manejables y similares al problema original, se convierte en el eje central de nuestro estudio. Nuestro trabajo, por lo tanto, no solo se orienta hacia el diseño e implementación de diversos algoritmos siguiendo las pautas y requerimientos específicos establecidos en las directrices de la práctica, sino que también busca profundizar en la comprensión teórica y aplicada de esta estrategia de solución de problemas.

Fomentamos así la capacidad de resolver problemas complejos de manera eficiente y efectiva, aplicando los principios de esta técnica de diseño de algoritmos. Además, pretendemos desarrollar una habilidad crítica en la evaluación de las soluciones obtenidas, no solo en términos de su correctitud sino también en términos de eficiencia computacional y optimización de recursos. Esto implica una exploración detallada de cómo la división del problema influye en la complejidad temporal y espacial de los algoritmos diseñados, y cómo se puede lograr un balance entre la simplicidad del diseño y la eficiencia en la ejecución.

Asimismo, otro de nuestros objetivos es fomentar el trabajo colaborativo y el intercambio de ideas dentro del equipo, para potenciar la creatividad en la solución de problemas y la capacidad de adaptación de la metodología a diferentes contextos y tipos de problemas. Buscamos, por ende, no solo la asimilación de conocimientos técnicos sino también el desarrollo de competencias transversales, como el pensamiento crítico, la comunicación efectiva y la gestión del trabajo en equipo.

En conclusión, los objetivos de nuestra práctica abarcan una comprensión exhaustiva de la metodología Divide y Vencerás en el ámbito del diseño de algoritmos, la habilidad para implementar soluciones eficientes y efectivas a problemas complejos, y el desarrollo de habilidades blandas y competencias que complementen nuestra formación técnica. A través de este enfoque integral, buscamos prepararnos no solo para enfrentar desafíos académicos sino también para contribuir de manera significativa en el campo profesional de la ciencia de la computación

3. P1: Subsecuencia de suma máxima

3.1. Definición del problema

Dado una secuencia de valores reales (positivos o negativos), almacenada en un vector, se pide encontrar una subsecuencia de elementos consecutivos que cumpla que la suma de los elementos de la misma sea la máxima posible.

Aunque existe el algoritmo de Kadane que permite encontrar de forma óptima el valor de dicha subsecuencia en $O(n)$, se pide encontrar un algoritmo que siguiendo la estrategia divide y vencerás encuentre una solución al problema lo más eficiente posible, esto es, con dicho orden de eficiencia.

3.2. Algoritmo específico

3.2.1. Diseño e implementación

No sorprende que intentando encontrar solución para el problema de manera no recursiva y buscando eficiencia, hayamos llegado de manera independiente a una solución totalmente equivalente al algoritmo de Kadane. En cualquier caso, merece la pena exponer los razonamientos que nos han derivado a la obtención de este algoritmo además de una demostración formal de su corrección.

La idea es la siguiente: vamos acumulando en una variable la suma de todos los elementos del vector recorridos hasta el momento, y ponemos a 0 este acumulador cada vez que alcance un valor negativo. Y es que esto nos proporciona una delimitación de la Subsecuencia de Suma Máxima, en tanto que, sea cual sea, no puede contener un valor del vector en el que la acumulación haya sido negativa.

Esto es totalmente intuitivo, como vemos en el caso del primer valor del vector para el cual el acumulador sea negativo: Si vamos acumulando y la suma de todos los elementos que llevamos se torna negativa en el elemento k-ésimo, una secuencia que comience en el elemento $k+1$-ésimo, nunca aumentaría su suma extendiéndose hacia la izquierda. Se ve claramente en un ejemplo:

$$[3, 4, 10, -20, 8, 2, ...]$$

Y apreciamos que cualquier secuencia que comience en el 8 nunca aumentaría su suma total al extender hacia la izquierda incluyendo al -20 y a alguno de los valores a su previos a este.

Proponemos ahora una demostración formal de este hecho, generalizando:

Teorema 3.1. *Sea v un vector de enteros de n elementos $v = [a_0, a_1, \ldots, a_{n-1}]$ y una subsecuencia $S = [a_k, a_{k+1}, \ldots, a_{k+r-1}]$ de r elementos contenida en v. Sea también A_s definida como:*

$$A_s = \begin{cases} a_0 & si\ s = 0 \\ A_{s-1} + a_s & si\ A_{s-1} + a_s \geq 0 \\ 0 & si\ A_{s-1} + a_s < 0 \end{cases}$$

Entonces, si $A_{k-1} = 0$ la subsecuencia S tendrá una suma mayor que cualquier extensión suya hacia la izquierda, es decir:

$$\sum_{j=k}^{k+r-1} a_j \geq \sum_{j=k-m}^{k+r-1} a_j, \quad m \in \mathbb{N} : 0 \leq m \leq k$$

Demostración. La demostración es por inducción según el predicado $P(u)$ del tenor:

cualquier subsecuencia del vector v que comience en el elemento posterior a uno k tal que $A_k = 0$ por u-ésima vez, tiene mayor suma que cualquier extensión suya hacia la izquierda.

Demostramos el caso $u = 1$, es decir, tenemos una tenemos una subsecuencia $S = [a_{k+1}, a_{k+2}, \ldots, a_{k+r}]$, donde $A_k = 0$ por primera vez en el recorrido del vector. Es decir, tenemos

$$a_0 + a_1 + a_2 + \ldots + a_k < 0 \tag{1}$$

$$a_0 + a_1 + a_2 + \ldots + a_{k-s} \geq 0 \tag{2}$$

Para cualquier s natural entre 1 y k. Veamos que esto implica que $a_{k-s} + a_{k-s+1} + \ldots + a_k < 0$ y que por tanto al extender hacia la izquierda la subsecuencia S su suma nunca aumenta.

$$a_{k-s} + a_{k-s+1} + \ldots + a_k \leq a_0 + a_1 + a_2 + \ldots + a_{k-s} + a_{k-s+1} + \ldots + a_k < 0$$

Donde la primera desigualdad se da por (2).

Para el caso $u \geq 1$:
Supongamos como hipótesis de inducción que $P(u-1)$ vale y desarrollamos en dos partes:
En primer lugar, de manera totalmente análoga al caso $u = 1$ demostramos que la subsecuencia tiene suma mayor que cualquier extensión suya hacia la izquierda delimitada entre el k-ésimo elemento del vector donde $A_k = 0$ por u-ésima vez y el j-ésimo elemento del vector, donde $A_j = 0$ por $(u-1)$-ésima vez. Además, por hipótesis de inducción, tenemos que la subsecuencia también tiene suma mayor que cualquier extensión suya tomando elementos enteriores al j-ésimo. Por tanto, se tiene $P(u)$ y por el **Principio de Inducción Matemática** sabemos que $P(u)$ vale en todos los casos (incluso considerando un vector de infinitos elementos, y por tanto el caso finito queda probado). □

Por tanto, tenemos que una SSM no puede atravesar un punto en el que la variable acumuladora se anule. Además, es ahora también evidente que la Subsecuencia de Suma Máxima comenzará en un valor del vector inmediatamente posterior a uno en el que se haya anulado la variable acumuladora, pues de no ser así, podría extenderse hacia la izquierda hasta ese punto aumentando su suma (ya que la suma de esos valores es positiva pues la variable acumuladora no se ha anulado al pasar por ellos).

Por tanto, para encontrarla entonces no tendremos más que comenzar una posible SSM cada vez que la variable acumuladora valga 0, y actualizar la mejor subsecuencia cuando la suma de la actual sea mayor que la suma de la anterior SSM, obteniendo la verdadera SSM al recorrer el vector al completo.

3.2.2. Análisis de eficiencia

Eficiencia Teórica del Algoritmo Específico

La eficiencia teórica del algoritmo implementado en la función `lineal`, encargada de calcular la subsecuencia contigua de suma máxima (MCSS) de un array, así como el máximo prefijo, máximo sufijo y la suma total del array, se analiza a continuación. Este análisis se centra en el comportamiento del algoritmo en función del tamaño de entrada n, siendo n la longitud del array.

El algoritmo realiza un único recorrido lineal por el array, determinando las métricas mencionadas mediante operaciones de complejidad constante en cada paso. Por lo tanto, la complejidad temporal del algoritmo es directamente proporcional al tamaño del array.

a) **Inicialización y Recorrido Lineal:** Al inicio, el algoritmo configura los valores iniciales para MCSS y el máximo prefijo a partir del primer elemento del array. Este proceso de inicialización se realiza en tiempo constante.

```
tupla lineal(int ini, int fin, ll a[]){
    tupla ans;
    // Initialise mcss and max_prefix to the first element of the array
    ans.mcss.ini = ans.max_prefix.ini = ini;
    ans.mcss.val = ans.max_prefix.val = a[ini];
    ans.mcss.fin = ans.max_prefix.fin = ini+1;
    ll ac = 0; // Accumulator
    int loc_ini = 0;
    ll max_array_ind = 0; // Index of the maximum element of the array
```

Durante el recorrido por el array (un bucle `for` que va desde `ini` hasta `fin`), se realizan sumas, comparaciones y actualizaciones de variables que son operaciones de complejidad $O(1)$. Este recorrido permite calcular el MCSS, el máximo prefijo y la suma total del array en tiempo lineal.

```
    for(int i=ini; i<fin; ++i){
        ac += a[i];
        if(ac < 0){ // If the accumulator is negative, we reset it (as well as the
     local starting position)
            ac = 0;
            loc_ini = i+1;
        }
        if(ac > ans.mcss.val){ // If the accumulator is greater than the current
    mcss, then it is updated
            ans.mcss.val = ac;
            ans.mcss.fin = i+1;
            ans.mcss.ini = loc_ini;
```

```
        }
        if(a[max_array_ind] < a[i]){ // Calculating subsequent of the maximum element of the array
            max_array_ind = i;
        }
        ans.total += a[i]; // Calculating total of the array
        if(ans.total > ans.max_prefix.val){ // If the current total of the array [0,i] is greater than max_prefix, then it is updated
            ans.max_prefix.val = ans.total;
            ans.max_prefix.fin = i+1;
        }
    }
```

b) **2ª inicialización** Después del recorrido principal, el algoritmo realiza algunas inicializaciones para calcular el sufijo máximo, esto, al igual que en el caso anterior, es constante (O(1))

```
    if(!ans.mcss.val){ // If mcss is 0, then all the array is negative and mcss is the maximum element of the array
        ans.mcss.ini = max_array_ind;
        ans.mcss.fin = max_array_ind+1;
        ans.mcss.val = a[max_array_ind];
    }

    // Initialise max_sufix to the last element of the array
    ans.max_sufix.val = a[fin-1];
    ans.max_sufix.ini = fin-1;
    ans.max_sufix.fin = fin;
    ll sum = 0;
```

c) **Cálculo de Sufijo Máximo:** Después del recorrido principal y la segunda inicialización, el algoritmo ejecuta otro recorrido desde el final del array hacia el inicio para determinar el sufijo máximo. Al igual que el recorrido anterior, este se realiza en tiempo lineal respecto al tamaño del array.

```
    for(int i=fin-1; i>=ini; --i){
        sum += a[i]; // Calculating sum of [i,fin)
        if(sum > ans.max_sufix.val){ // If [i,fin) is greater than max_sufix, then it is updated
            ans.max_sufix.val = sum;
            ans.max_sufix.ini = i;
        }
    }

    return ans;
}
```

Complejidad Total: Considerando que ambos recorridos por el array y las operaciones de inicialización y actualización se realizan en tiempo constante, la complejidad total del algoritmo `"lineal"` [1] es $O(n)$, donde n es la longitud del array.

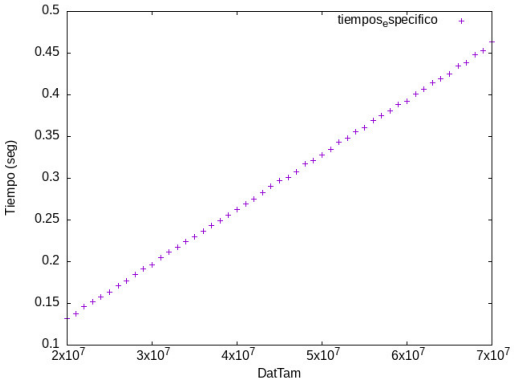
$$T(n) = O(n) \tag{3}$$

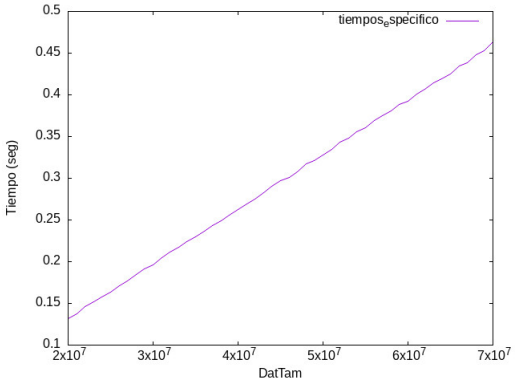
Por lo tanto, la eficiencia teórica del algoritmo `lineal`, responsable de calcular la subsecuencia contigua de suma máxima, máximo prefijo, máximo sufijo y la suma total de un array, se establece como de orden lineal $O(n)$. Esto indica que el tiempo de ejecución del algoritmo crece de manera proporcional al tamaño del problema n.

Eficiencia Empírica del algoritmo específico

Para realizar el análisis de la eficiencia empírica, basta con realizar múltiples ejecuciones del algoritmo para comprobar de forma práctica (empírica) como se comporta. (Que podemos adelantar, que será de forma lineal tal y como hemos desarrollado formalmente)

[1] Evidentemente, es por esto que hemos nombrado a la función de dicha manera





Eficiencia Híbrida del algoritmo específico
Es bastante evidente darnos cuenta de que la regresión que debemos aplicar para encontrar el ajuste, obviamente es lineal, confirmando el análisis teórico. Observemoslo:

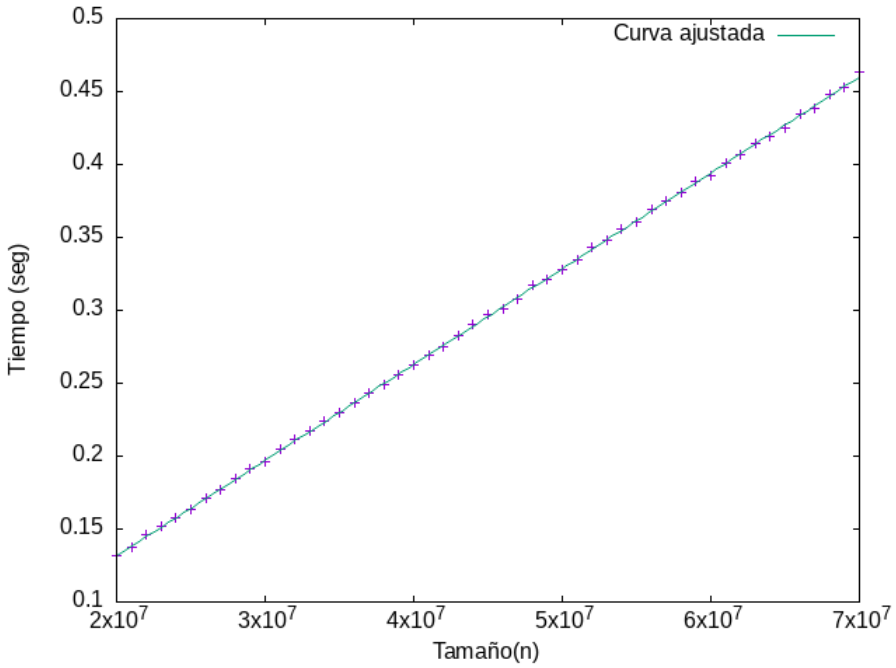


Figura 2: $f(x) = 6.57198 \cdot 10^{-9} \cdot x$

3.3. Algoritmo divide y vencerás

3.3.1. Diseño e implementación

La idea de este algoritmo es más intuitiva si cabe del algoritmo lineal: separamos el vector en dos vectores de la mitad del tamaño original cada uno y calculamos recursivamente en cada uno de ellos:

- Su SSM
- Su SSM que tenga como primer elemento el primero del array: maxPrefix
- Su SSM que tenga como último elemento el último del array: maxSuffix

De esta manera, es clara la manera de unir soluciones de ambos lados del array: la SSM total será la que tenga mayor suma de entre

- La SSM de la izquierda

- La SSM de la derecha
- La SSM resultante de concatenar maxSuffix de la izquierda con maxPrefix de la derecha

Para falcilitar este proceso se usa el tipo de dato sub-secuencia (con un operador de comparación y de salida) y un tipo de dato tupla que reune tres secuencias: la SSM, maxPrefix y maxSuffix. Además, este último tipo también guarda la suma total de los elementos del array que está considerando en ese paso (útil para actualizar el valor de maxPrefix y maxSuffix en la fase de Mezcla de Soluciones).

```
#include <iostream>
#include <cassert>
using namespace std;

typedef long long ll;

struct subsequent{
    ll val,ini,fin;
    subsequent(ll val = 0, ll ini  = 0, ll fin = 0) : val(val), ini(ini), fin(fin) {}

    friend bool operator<(const subsequent & a,const subsequent & b){
        return a.val < b.val;
    }

    friend ostream & operator<<(ostream & os, const subsequent & ind){
        os << ind.val << " [" << ind.ini << ", " << ind.fin << "]";
        return os;
    }
};

struct tupla
{
    subsequent mcss,max_prefix,max_sufix;
    ll total;

    tupla(subsequent mcss = subsequent(), subsequent p = subsequent(), subsequent s =
    subsequent(), ll t = 0) :
     mcss(mcss), max_prefix(p), max_sufix(s), total(t) {}

    friend ostream & operator<<(ostream & os, const tupla & t){
        os << "mcss: " << t.mcss << endl;
        os << "max_prefix: " << t.max_prefix << endl;
        os << "max_sufix: " << t.max_sufix << endl;
        os << "total: " << t.total;
        return os;
    }
};
```

En particular, como no podría ser de otra manera, la forma de obtener las subsecuencias maxSuffix y maxPrefix en cada caso iteración es el siguiente:

- Para obtener maxPrefix: se obtiene la máxima subsecuencia (en cuanto a suma) entre la `maxPrefix_izquierda` y la unión de todo el vector izquierdo con `maxPrefix_derecha`.
- Para obtener maxSuffix: se obtiene la máxima subsecuencia (en cuanto a suma) entre la `maxSuffix_derecha` y la unión de todo el vector derecho con `maxSuffix_izquierda`.

Usando estas ideas, el código queda de la siguiente manera: (Ignórense los comentarios)

```
using namespace std;

typedef long long ll;

const int UMBRAL = 10;

// [ini,fin)
struct subsequent{
    ll val,ini,fin;
    subsequent(ll val = 0, ll ini  = 0, ll fin = 0) : val(val), ini(ini), fin(fin) {}

    friend bool operator<(const subsequent & a,const subsequent & b){
```

```
        return a.val < b.val;
    }

    friend ostream & operator<<(ostream & os, const subsequent & ind){
        os << ind.val << " [" << ind.ini << ", " << ind.fin << "]";
        return os;
    }
};

struct tupla
{
    subsequent mcss,max_prefix,max_sufix;
    ll total;

    tupla(subsequent mcss = subsequent(), subsequent p = subsequent(), subsequent s =
    subsequent(), ll t = 0) :
     mcss(mcss), max_prefix(p), max_sufix(s), total(t) {}

    friend ostream & operator<<(ostream & os, const tupla & t){
        os << "mcss: " << t.mcss << endl;
        os << "max_prefix: " << t.max_prefix << endl;
        os << "max_sufix: " << t.max_sufix << endl;
        os << "total: " << t.total;
        return os;
    }
};

tupla lineal(int ini, int fin, ll a[]){
    tupla ans;
    // Initialise mcss and max_prefix to the first element of the array
    ans.mcss.ini = ans.max_prefix.ini = ini;
    ans.mcss.val = ans.max_prefix.val = a[ini];
    ans.mcss.fin = ans.max_prefix.fin = ini+1;
    ll ac = 0; // Accumulator
    int loc_ini = 0;
    ll max_array_ind = 0; // Index of the maximum element of the array
    for(int i=ini; i<fin; ++i){
        ac += a[i];
        if(ac < 0){ // If the accumulator is negative, we reset it (as well as the
    local starting position)
            ac = 0;
            loc_ini = i+1;
        }
        if(ac > ans.mcss.val){ // If the accumulator is greater than the current mcss,
     then it is updated
            ans.mcss.val = ac;
            ans.mcss.fin = i+1;
            ans.mcss.ini = loc_ini;
        }
        if(a[max_array_ind] < a[i]){ // Calculating subsequent of the maximum element
    of the array
            max_array_ind = i;
        }
        ans.total += a[i]; // Calculating total of the array
        if(ans.total > ans.max_prefix.val){ // If the current total of the array [0,i]
     is greater than max_prefix, then it is updated
            ans.max_prefix.val = ans.total;
            ans.max_prefix.fin = i+1;
        }
    }

    if(!ans.mcss.val){ // If mcss is 0, then all the array is negative and mcss is the
     maximum element of the array
        ans.mcss.ini = max_array_ind;
        ans.mcss.fin = max_array_ind+1;
        ans.mcss.val = a[max_array_ind];
    }

    // Initialise max_sufix to the last element of the array
    ans.max_sufix.val = a[fin-1];
    ans.max_sufix.ini = fin-1;
```

```
    ans.max_sufix.fin = fin;
    ll sum = 0;
    for(int i=fin-1; i>=ini; --i){
        sum += a[i]; // Calculating sum of [i,fin)
        if(sum > ans.max_sufix.val){ // If [i,fin) is greater than max_sufix, then it is updated
            ans.max_sufix.val = sum;
            ans.max_sufix.ini = i;
        }
    }

    return ans;
}

tupla dyv(int ini, int fin, ll a[]){
    // Base case
    if(fin - ini <= UMBRAL){
        //return tupla(subsequent(a[ini],ini,fin),subsequent(a[ini],ini,fin),subsequent(a[ini],ini,fin),a[ini]);
        return lineal(ini,fin,a);
    }

    // Divide
    int mid = (fin + ini)/2;
    tupla t1,t2;
    t1 = dyv(ini,mid,a);
    t2 = dyv(mid,fin,a);

    // Fusion
    tupla ans;

    // mcss
    // mcss is the maximum of the solution of both sides and the sufix of the left side combined with
    // the prefix of the right side

    // ans.mcss = max(max(t1.mcss,t2.mcss),t1.max_sufix + t2.max_prefix)
     ans.mcss = max(t1.mcss, t2.mcss);
    /*if(t1.mcss.val > t2.mcss.val){ // ans.mcss = max(t1.mcss, t2.mcss)
        ans.mcss = t1.mcss;
    }else{
        ans.mcss = t2.mcss;
    }*/
    if(t1.max_sufix.val + t2.max_prefix.val > ans.mcss.val){
        ans.mcss = t1.max_sufix.val + t2.max_prefix.val;
        ans.mcss.ini = t1.max_sufix.ini;
        ans.mcss.fin = t2.max_prefix.fin;
    }

    // max_prefix
    // max_prefix is the maximum between the max_prefix of the left side
    // and the whole of the left side plus the max_prefix of the right side
    if(t1.max_prefix.val > t1.total + t2.max_prefix.val){
        ans.max_prefix = t1.max_prefix;
    }else{
        ans.max_prefix.val = t1.total + t2.max_prefix.val;
        ans.max_prefix.ini = ini;
        ans.max_prefix.fin = t2.max_prefix.fin;
    }

    // max_sufix
    // max_sufix is the maximum between the max_sufix of the right side
    // and the whole of the right side plus the max_sufix of the left side
    if(t2.max_sufix.val > t2.total + t1.max_sufix.val){
        ans.max_sufix = t2.max_sufix;
    }else{
        ans.max_sufix.val = t2.total + t1.max_sufix.val;
        ans.max_sufix.ini = t1.max_sufix.ini;
        ans.max_sufix.fin = fin;
    }

```

```
    // total
    // total is the total sum of both sides
    ans.total = t1.total + t2.total;

    return ans;
}
```

3.3.2. Análisis de eficiencia

Eficiencia teórica

Calculemos la eficiencia teórica del algoritmo, es decir, de la función *dyv*.

```
tupla dyv(int ini, int fin, ll a[]){
    // Base case
    if(fin - ini <= UMBRAL){
        //return tupla(subsequent(a[ini],ini,fin),subsequent(a[ini],ini,fin),
    subsequent(a[ini],ini,fin),a[ini]);
        return lineal(ini,fin,a);
    }
```

Observando el código podemos observar un $if/else$ inicial (aunque no haya *else* explícitamente podemos ver que el resto del código solo se ejecuta cuando no lo hace el if al haber un *return*). Podemos ver que el código del if se ejecuta solo cuando el tamaño del problema ($n = fin - ini$) es menor o igual que la constante $UMBRAL \in \mathbb{N}$. Estos serían los casos base los cúales se resuelven llamando a la función *lineal* que consiste en el algoritmo específico previamente detallado cuya eficiencia se vio que era $O(n)$.

```
    // Divide
    int mid = (fin + ini)/2;
    tupla t1,t2;
    t1 = dyv(ini,mid,a);
    t2 = dyv(mid,fin,a);
```

Mientras tanto en el *else* (caso general) se calcula el punto medio entre ini y fin para luego llamar recursivamente a la función *dyv*. Es decir, se parte el problema por la mitad y se llama recursivamente a la función para resolver cada una de las mitades.

```
    // Fusion
    tupla ans;

    // mcss
    // mcss is the maximum of the solution of both sides and the sufix of the left
    side combined with
    // the prefix of the right side

    // ans.mcss = max(max(t1.mcss,t2.mcss),t1.max_sufix + t2.max_prefix)
     ans.mcss = max(t1.mcss, t2.mcss);
    /*if(t1.mcss.val > t2.mcss.val){ // ans.mcss = max(t1.mcss, t2.mcss)
        ans.mcss = t1.mcss;
    }else{
        ans.mcss = t2.mcss;
    }*/
    if(t1.max_sufix.val + t2.max_prefix.val > ans.mcss.val){
        ans.mcss = t1.max_sufix.val + t2.max_prefix.val;
        ans.mcss.ini = t1.max_sufix.ini;
        ans.mcss.fin = t2.max_prefix.fin;
    }

    // max_prefix
    // max_prefix is the maximum between the max_prefix of the left side
    // and the whole of the left side plus the max_prefix of the right side
    if(t1.max_prefix.val > t1.total + t2.max_prefix.val){
        ans.max_prefix = t1.max_prefix;
    }else{
```

```
        ans.max_prefix.val = t1.total + t2.max_prefix.val;
        ans.max_prefix.ini = ini;
        ans.max_prefix.fin = t2.max_prefix.fin;
    }

    // max_sufix
    // max_sufix is the maximum between the max_sufix of the right side
    // and the whole of the right side plus the max_sufix of the left side
    if(t2.max_sufix.val > t2.total + t1.max_sufix.val){
        ans.max_sufix = t2.max_sufix;
    }else{
        ans.max_sufix.val = t2.total + t1.max_sufix.val;
        ans.max_sufix.ini = t1.max_sufix.ini;
        ans.max_sufix.fin = fin;
    }

    // total
    // total is the total sum of both sides
    ans.total = t1.total + t2.total;

    return ans;
}
```

El resto del código se compone de $if/else$ y operaciones matemáticas las cuales son $O(1)$ y están situadas secuencialemente a las llamadas recursivas previamente mencionadas.

Por tanto, tenemos que la ecuación de la eficiencia teórica es:

$$T(n) = \begin{cases} n & n <= UMBRAL \quad \text{caso base} \\ \\ 2T(\frac{n}{2}) + 1 & n > UMBRAL \quad \text{caso general} \end{cases} \qquad \forall n \in \mathbb{N} \tag{4}$$

Resolvamos la ecuación de recurrencia para obtener la eficiencia del caso general, es decir, $\forall n \in \mathbb{N}$ tal que $n > UMBRAL$:

$$T(n) = 2T(\frac{n}{2}) + 1 \qquad \forall n \in \mathbb{N} \ : \ n > UMBRAL \tag{5}$$

Desarrollándola en serie k veces, con $k = log_2(\frac{n}{UMBRAL})$ (aproximando al entero superior) puesto que el problema es de tamaño n y se va subdividiendo por la mitad hasta llegar a un problema de tamaño $UMBRAL$, tenemos:

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + 1 = 2(2T(\frac{n}{4}) + 1) + 1 = 4T(\frac{n}{4}) + 2 + 1 = 4(2T(\frac{n}{8}) + 1) + 3 = 8T(\frac{n}{8}) + 4 + 3 = \\ &= 8T(\frac{n}{8}) + 7 = ... = 2^k T(\frac{n}{2^k}) + 2^k - 1 = ... = \left\{\text{Sustituimos por } k = log_2(\frac{n}{UMBRAL})\right\} = \\ &= 2^{log_2(\frac{n}{UMBRAL})} T(\frac{n}{2^{log_2(\frac{n}{UMBRAL})}}) + 2^{log_2(\frac{n}{UMBRAL})} - 1 = \\ &= \frac{n}{UMBRAL} T(UMBRAL) + \frac{n}{UMBRAL} - 1 \in O(n) \end{aligned}$$

Como $T(UMBRAL)$ es constante (por serlo $UMBRAL$) tenemos que:

$$\frac{n}{UMBRAL} T(UMBRAL) + \frac{n}{UMBRAL} - 1 \in O(n) \tag{6}$$

Por tanto, recordando la ecuación (13) tenemos que el orden de eficiencia teórica de este algoritmo es $O(n)$ $\forall n \in \mathbb{N}$.

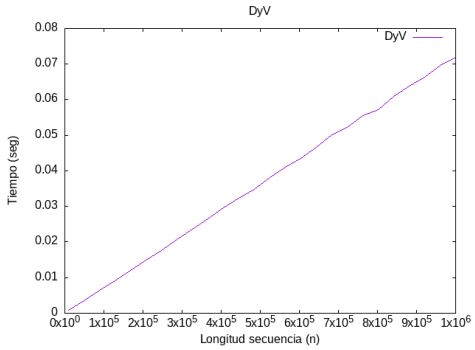
Eficiencia empírica

Realicemos ahora el estudio de la eficiencia empírica. Para ello, el programa ha sido ejecutado con los valores más grandes posibles, es decir, desde secuencias de tamaño $n = 10000$ hasta $n = 1000000$ cada 39600.

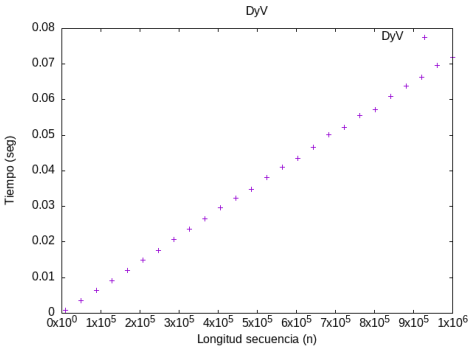
Los tiempos obtenidos para el tamaño de secuencia n se encuentran en la Tabla 1. Además se puede observar su representación en las gráficas 3a y 3b.

Tamaño(n)	Tiempo (seg)
10000	0.0007313
49600	0.00350708
89200	0.00637291
128800	0.00917196
168400	0.0121215
208000	0.0149036
247600	0.0177062
287200	0.0207045
326800	0.0235259
366400	0.0264423
406000	0.0297207
445600	0.0324105
485200	0.0347352
524800	0.0380546
564400	0.0411165
604000	0.0435186
643600	0.0467262
683200	0.0500663
722800	0.052286
762400	0.0555019
802000	0.0571399
841600	0.0609255
881200	0.0637417
920800	0.0662403
960400	0.0695548
1000000	0.0719015

Cuadro 1: Tiempos obtenidos para el algoritmo divide y vencerás del problema 1



(a) Gráfica del algoritmo divide y vencerás para el problema 1 (línea)



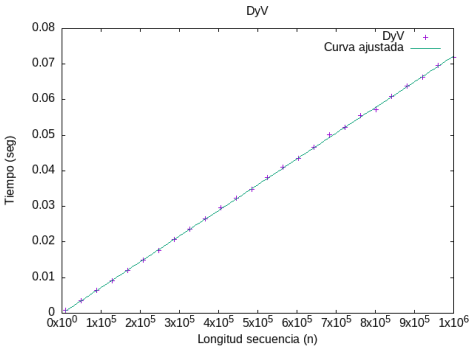
(b) Gráfica del algoritmo divide y vencerás para el problema 1 (puntos)

Como se puede observar, se trata de una función claramente lineal, lo que coincide con el resultado demostrado en el análisis teórico.

Eficiencia híbrida

Previamente en el análisis teórico se vio que la eficiencia del algoritmo es $O(n)$ y en el análisis empírico constatamos que efectivamente el algoritmo tiene complejidad lineal.

Para obtener las constantes ocultas de la ecuación de eficiencia y verificar que en efecto la función que representa la complejidad del algoritmo se trata de una función lineal hemos realizado una regresión lineal por mínimos cuadrados sobre los puntos obtenidos empíricamente con gnuplot, es decir, con una recta de la forma $f(x) = ax + b$.



(a) Gráfica ajustada del algoritmo divide y vencerás para el problema 1

```
After 10 iterations the fit converged.
final sum of squares of residuals : 2.15874e-06
rel. change during last iteration : -7.84748e-15

degrees of freedom    (FIT_NDF)                        : 24
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.000299912
variance of residuals (reduced chisquare) = WSSR/ndf   : 8.99474e-08

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 7.22924e-08      +/- 1.98e-10     (0.2739%)
b               = -1.42768e-05     +/- 0.000116     (812.7%)

correlation matrix of the fit parameters:
                a      b
a               1.000
b              -0.862  1.000
```

(b) Regresión lineal del divide y vencerás del problema 1

La función obtenida es:

$$f(x) = 7.22924 \cdot 10^{-8}x - 1.42768 \cdot 10^{-5}$$

Como podemos observar en la figura 4b la varianza residual es minúscula lo que nos indica que el ajuste realizado es el correcto, tal y como se ve en la gráfica 4a y por tanto el algoritmo es, en efecto, $O(n)$.

3.3.3. Cálculo de umbrales

En esta sección calcularemos los umbrales teórico, óptimo y de tanteo, para posteriormente compararlos gráficamente.

Umbral teórico
Para el cálculo del umbral teórico, planteamos la ecuación del algoritmo específico tal cual ($h(n) = n$), la aplicamos en el primer nivel de recurrencia y resolvemos la ecuación (no recurrente):

$$T(n) = \begin{cases} n & n \leq UMBRAL \\ 2T(\frac{n}{2}) + 1 & n > UMBRAL \end{cases} \qquad \forall n \in \mathbb{N} \tag{7}$$

Entonces nos queda:

$$n = 2(\frac{n}{2}) + 1 \iff 1 = 0$$

Con lo que concluimos que no podemos calcular el umbral teórico para este algoritmo. Este resultado se da por tener ambos algoritmos el mismo orden de eficiencia, y es una prueba más de que el umbral teórico es meramente indicativo.

Umbral óptimo
Para el cálculo del umbral óptimo, plantearíamos la función del algoritmo específico obtenida en la eficiciencia híbrida, la aplicaríamos en el primer nivel de recurrencia y resolvemos, pero al igual que con el teórico, esto no tiene sentido ya que ambos algorítmos son de el mismo orden de eficiencia.

Umbral empírico o real
Teóricamente, el umbral empírico será cuando coinciden los tiempos de ejecución de los dos algoritmos, por tanto buscaremos el punto de corte de las graficas determinadas por ambos algoritmos (especifico y DyV), siendo el específico: $f(x) = 6.57198 \cdot 10^{-9} \cdot x$ y el DyV: $g(x) = 7.22924 \cdot 10^{-8} \cdot x - 1.42768 \cdot 10^{-5}$

$$6.57198 \cdot 10^{-9} \cdot x = 7.2292410^{-8} \cdot x - 1.4276810^{-5} \implies x = 217.24$$

Así que el umbral óptimo $n_o = 217.24 \approx 217$

Umbral de tanteo
Realmente, el umbral de tanteo se realiza en base al óptimo, pero ya que no podemos hacerlo, realizaremos el mismo proceso basándonos en el umbral real, para al menos, poder mostrar el procedimiento. Lo haremos con 4 valores distintos al real, 2 por encima y 2 por debajo, con una variación de 10 unidades. Aquí vemos los resultados:

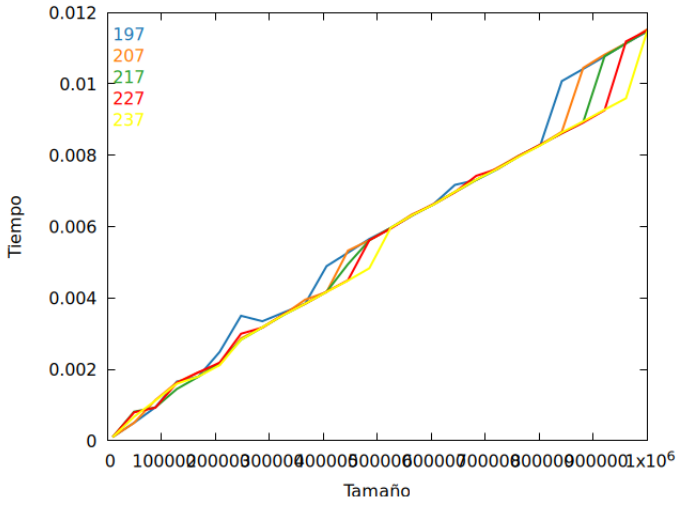


Figura 5: Variación de eficiencia frente a variación de umbrales

Esta gráfica no tiene especial sentido, puesto que esta manera de calcular los umbrales de tanteo no la tiene, aún así, no es díficil comprender la gráfica, vemos que las 5 rectas se comportan de la misma

forma, pero con un desfase de unas 10 unidades en el tamaño del problema, que es la variación de umbrales que hemos aplicado.

4. P2: Enlosar un espacio

4.1. Definición del problema

Queremos cubrir el suelo de una habitación cuadrada con baldosas en forma de 'ele'. 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 $n = 2k$ para algún $k >= 1$. La posición del sumidero la conocemos a priori por el par de valores i, j con $0 \leq i, j \leq n-1$. 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.

Por ejemplo, para una entrada del tipo

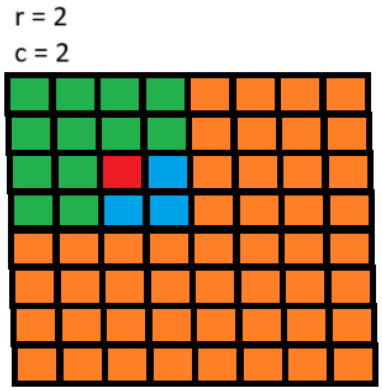
```
n = 2
i = 0
j = 0
Salida:
0    1
1    1
n = 4
i = 0
j = 0
Salida:
0    3    2    2
3    3    1    2
4    1    1    5
4    4    5    5
```

4.2. Algoritmo específico

4.2.1. Diseño e implementación

Aunque la manera más directa y natural de resolver este problema sea utilizar la técnica divide y vencerás, existen otros diseños de algoritmos que resuelven el problema de forma aceptablemente natural. Uno de ellos es el algoritmo de completar cuadrados, cuya idea base es a partir de un cuadrado de tamaño $m \times m$, envolverlo con una L para formar un cuadrado de tamaño $2m \times 2m$.

En resumen, el algoritmo consiste en repetir este proceso desde el hueco 1×1 que nos dan hasta completar el tablero $n \times n$ entero.



Mirando la imagen superior, el algoritmo añadiría primero la L azul, envolviendo al sumidero rojo y completando un cuadrado 2×2 que envuelve después con la L verde, completando un cuadrado 4×4, que finalmente envuelve con la L naranja, completando el tablero 8×8.

La mayor complicación del algoritmo reside en ver en cada paso dónde y cómo colocar la L, además de cómo construir una L grande a partir de eles pequeñas.

Para entenderlo, echemos un ojo a su implementación:

```
void fill_L(vector<vector<int>> & v,int n,int r, int c, int & tile){

    // Add log_2(n) - 1 slabs from level 1 to level n/2
    for (int m = 1; m < n; m *= 2)
    {
        // Calculate orientation
        int orientation = 2*((r/m) % 2) + ((c/m) % 2);

        // m x m square to left empty, starting from the original 1 x 1 in (r,c)
        int i = (r / m)*m;
        int j = (c / m)*m;

        // Calculate position
        i += orientation / 2 == 0 ? m-1 : 0;
        j += orientation % 2 == 0 ? m-1 : 0;

        // Add slab of level m
        add_slab_m(v, m, i, j, orientation, tile);
    }
}
```

La función recibe como parámetros una matriz de enteros `v`, el lado `n` de dicha matriz, la posición `(r,c)` en la matriz, y un entero por referencia que será el número de losa que estemos poniendo (por facilitar la recursividad, aunque la función en sí no es recursiva).

La función consta de un único bucle for con 6 líneas de cuerpo, y comienza calculando la orientación de la L que se va a colocar, siendo esta un entero del 0 al 3. El programa entiende que en el cuadrado

```
0   1
2   3
```

la orientación es la casilla que se deja libre al colocar la L.

Por tanto, para calcular la orientación lo que se hace es dividir de forma abstracta (`r/m` y `c/m`) la matriz en cuadrados de tamaño `m` × `m` y ver en qué fila y columna está ese cuadrado dentro de uno de lado el doble (0 o 1) para finalmente coonvertir los dos dígitos binarios a un entero decimal del 0 al 3.

Las siguientes dos líneas tienen como fin ajustar `i` y `j` a la esquina superior izquierda del cuadrado `m` × `m` que hemos construido.

Las siguientes dos líneas tienn por objetivo ajustar `i` y `j` a la casilla exterior a la L que es adyacente a ambos brazos, es decir, la que se marca por x en los esquemas siguientes:

```
x   1          2   2   2   2
1   1          2   2   2   2
               2   2   x   .
               2   2   .   .
```

Una vez se tiene esa casilla localizada y la orientación, la función `add_slab_m` se encarga de colocar la L de nivel m. Veamos el código de dicha función.

```
void add_slab_m(vector<vector<int>> & v, int m, int i, int j, int orientation, int &
    slab)
{
    // Variables to calculate where and how to add the L slabs
    int inc_i = orientation / 2 == 0 ? 1 : -1;
    int inc_j = orientation % 2 == 0 ? 1 : -1;

    // Level 1, base case
    if (m == 1)
    {
```

```
        v[i][j + inc_j] = v[i + inc_i][j] = v[i + inc_i][j + inc_j] = slab;
        ++slab;
    }
    else
    {
        // Level m > 1, add four L slabs of level m/2 following the pattern:

        // .   .   3   3
        // .   .   1   3
        // 2   1   1   4
        // 2   2   4   4

        // Calculate non-adjacent L orientations

        // orientation --> orientation1 orientation2

        // switch(orientation)
        // case 0 --> 1 2
        // case 1 --> 0 3
        // case 2 --> 3 0
        // case 3 --> 2 1
        int orientation1 = orientation + inc_j;
        int orientation2 = 3 - orientation1;

        // Escale variables to actual level
        inc_i *= (m/2);
        inc_j *= (m/2);

        // Add insider L in the same position of big slab with the same orientation
        add_slab_m(v, m / 2, i, j, orientation, slab);

        // Add 2 corner L's with calculated orientations and the following positions
        // (i +- m/2, j -+ m/2 +- 1)
        // (i -+ m/2 +- 1, j +- m/2)
        add_slab_m(v, m / 2, i + inc_i, j - inc_j + (abs(inc_j)/inc_j), orientation1,
    slab);
        add_slab_m(v, m / 2, i - inc_i + (abs(inc_i)/(inc_i)), j + inc_j, orientation2
    , slab);

        // Add outsider L with the same orientation as the big one.
        add_slab_m(v, m / 2, i + inc_i, j + inc_j, orientation, slab);
    }
}
```

Esta función recibe la matriz `v`, el nivel de la L que vamos a añadir `m`, que no es más que el lado de cada uno de los 3 cuadrados que la forman, la posición donde se inserta, que es la casilla adyacente a ambos brazos de la L, la orientación, que se define como se ha explicado previamente, y el número de losa básica que estamos colocando.

La función comienza calculando dos variables de incremento que servirán determinar donde y cómo colocar las baldosas, que valdrán 1 o -1 en función de la orientación de la L.

A continuación se implementa el caso base `m == 1`, en el que como vemos a partir de la posición original `(i,j)` se añaden losas de tamaño 1×1 en las posiciones `(i,j+inc_j)`, `(i+inc_i,j)` y `(i+inc_i,j+inc_j)`.

Si no estamos en el caso base, utilizamos que podemos formar una L de nivel `m` con 4 eles de nivel `m/2` siguiendo el siguiente patrón:

```
.   .   3   3
.   .   1   3
2   1   1   4
2   2   4   4
```

Donde cada terna de casillas con el mismo número forma una L de nivel `m/2` (recordemos que definimos el nivel de una L como el lado de cada uno de los tres cuadrados que la componen). Ahora la dificultad reside en calcular en que casilla y con qué orientación colocar cada una de las 4 eles. Primero nos damos cuenta de que la L interior y la exterior (1 y 4 en el esquema superior) tienen la misma orientación.

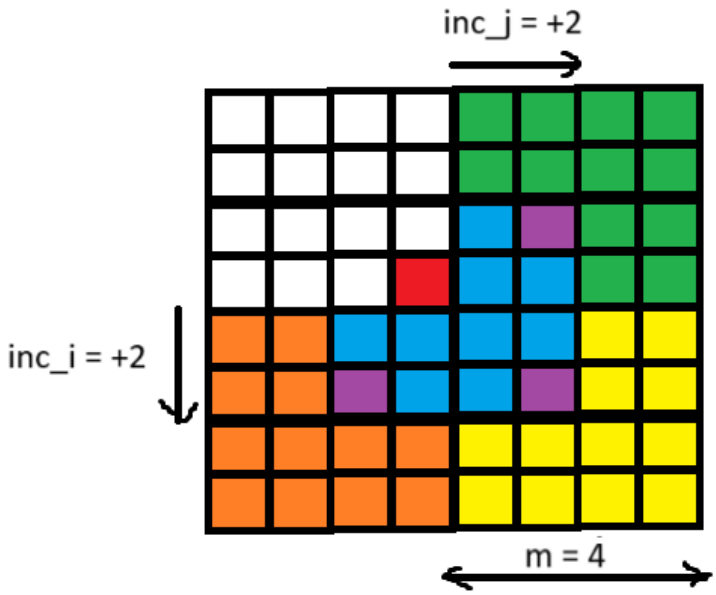
Para las eles no adyacentes entre sí (la 2 y la 3 en el esquema), estudiamos cada caso por separado y vemos que en función de la orientación original, se obtienen los valores que se muestran en el esquema inferior para la orientación de la L 2 y la de la L 3 respectivamente:

```
0 --> 1 2
1 --> 0 3
2 --> 3 0
3 --> 2 1
```

De hecho se podría implementar con un switch tal y como está comentado en el código. No obstante las dos líneas para calcular orientation1 y orientation2 dan un resultado equivalente (no sigue ningún fundamento matemático, simplemente surge de estudiar la casuística).

A continuación, escalamos los incrementos a la mitad del nivel de la L los incrementos para encontrar cada una de las posiciones de las 4 eles de nivel inferior.

Para ver las posiciones donde insertar las eles de nivel inferior, nos apoyamos en el siguiente dibujo:



Partiendo de la casilla roja `(i,j)` pretendemos insertar eles en las casillas moradas con sus respectivas orientaciones, además de en la casilla roja.

La L azul no tiene mucho misterio, pues tiene la misma posición y orientación que la L original, solo que de nivel `m/2`. La L amarilla tampoco tiene mayor complicación, pues tiene la misma orientación pero insertada en la casilla `(i+inc_i,j+inc_j)`.

La L naranja y la verde sí son más complicadas de calcular su posición, ya que aunque tengan el incremento calculado en una de sus coordenadas, también tienen el mismo incremento en la otra pero en sentido contrario y de una unidad menor.

Y con esto termina la implementación del algoritmo, pues se añaden eles de nivel `m/2` recursivamente hasta llegar al caso base.

4.2.2. Análisis de eficiencia

Eficiencia teórica

Ahora procedemos a analizar la eficiencia teórica del algoritmo. En primer lugar observamos analizamos la eficiencia de la función `add_slab_m`.

Si definimos la función T que dado el tamaño del problema m, devuelve el tiempo que tarda en ejecutarse la función, como consta de 4 llamadas recursivas de tamaño $m/2$ entonces adopta la siguiente

forma.

$$T(m) = \begin{cases} 1 & m = 1 \\ 4T(\frac{m}{2}) + 1 & m > 1 \end{cases} \qquad \forall m = 2^k, k \in \mathbb{N} \tag{8}$$

Observamos que:

$$T(m) = 4T(\frac{m}{2})+1 = 4(4T(\frac{m}{4})+1)+1 = 4^2T(\frac{m}{2^2})+1+4 = 4^kT(\frac{m}{2^k})+\sum_{i=0}^{k} 4^i \qquad \forall k = 1 \ldots log_2(m)$$

Desarrollando la suma de la progresión geométrica, nos queda:

$$T(m) = 4^kT(\frac{m}{2^k}) + \frac{4^{k+1}}{3} \qquad \forall k = 1 \ldots log_2(m)$$

Y tomando $k = log_2(m)$ obtenemos:

$$T(m) = 4^{log_2(m)}T(1) + \frac{4^{log_2(m)+1}}{3} = \frac{7}{3}m^2$$

Por lo que concluimos que la eficiencia de la función `add_slab_m` es del orden $O(n^2)$ tomando como tamaño el lado de la matriz, si tomásemos como tamaño el número de elementos de la matriz sería del orden $O(n)$.

Ahora volviendo a la función original, esta consta de un bucle `for` de $log_2(n) - 1$ iteraciones en el que se llama a la función `add_slab_m` con todas las potencias de 2 desde 1 hasta $n/2$.

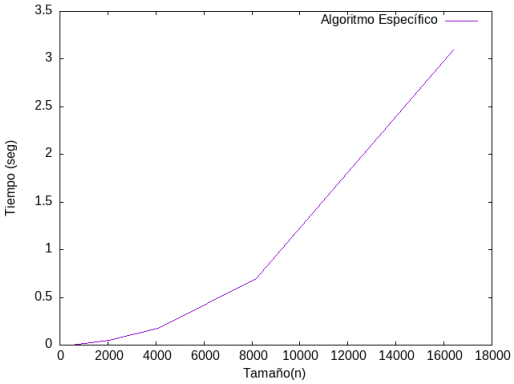
Si consideramos como tamaño del problema el número de elementos de la matriz, esta función es lineal como ya hemos visto. Así que la eficiencia de la función `fill_L` será la suma de las potencias de 2 desde 1 hasta $n/2$, que es bien sabido que es la siguiente potencia de dos menos una unidad, es decir $n - 1$, por tanto nuestra función tendrá nuevamente eficiencia $O(n)$ si consideramos como tamaño del problema el número de elementos de la matriz, y $O(n^2)$ si consideramos como tamaño del problema el lado de la matriz.

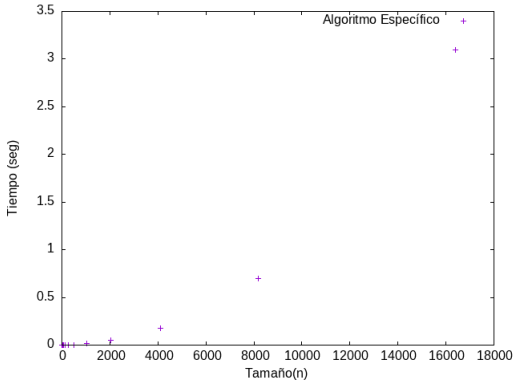
Eficiencia empírica

A continuación procedemos al análisis de la eficiencia empírica. Para ello, hemos ejecutado el programa con todos los tamaños posibles en mi ordenador, es decir desde $n = 2^0 = 1$ hasta $n = 2^{14} = 16384$, ya que la matriz resultante de la siguiente potencia sería de tamaño $n^2 = 2^{30}$ que ya no me cabe en memoria.

Aquí están los tiempos de ejecución resultantes donde se ha considerado el tamaño del problema n como el lado de la matriz:

Tamaño(n)	Tiempo (seg)
1	2.1e-07
2	2.21e-07
4	5.82e-07
8	1.042e-06
16	3.516e-06
32	1.2954e-05
64	3.714e-05
128	0.000135897
256	0.000530396
512	0.00297852
1024	0.0131311
2048	0.0468183
4096	0.178777
8192	0.698544
16384	3.09681





Como el lado de la matriz siempre tiene que ser una potencia de 2, los puntos distan bastante entre sí. A pesar de ello, observamos que el crecimiento de los datos se asemeja a una función cuadrática.

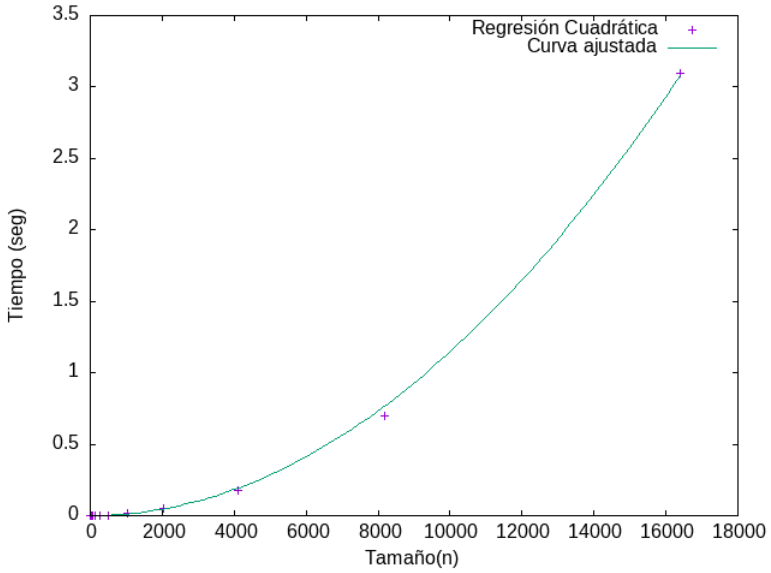
Eficiencia híbrida

En el análisis teórico hemos obtenido que la eficiencia del algoritmo es $O(n^2)$ si tomamos como tamaño n el lado de la matriz cuadrada, y en el análisis empírico hemos visto que la relación tamaño-tiempo parece ser cuadrática.

Veamos con más detalle que el análisis teórico se corresponde con la realidad, para ello, no tenemos más que realizar una regresión cuadrática de los puntos obtenidos con gnuplot, es decir, con una curva de la forma $f(x) = ax^2$. El resultado obtenido es:

$$f(x) = 1.14672 \cdot 10^{-8} \cdot x^2$$

```
*******************************************************************************
Sat Apr  6 16:26:18 2024


FIT:    data read from DATOS
        format = z
        #datapoints = 15
        residuals are weighted equally (unit weight)

function used for fitting: f(x)
        f(x)=a*x*x
fitted parameters initialized with current variable values

iter      chisq       delta/lim  lambda   a
   0 7.6861431878e+16   0.00e+00  7.16e+07    1.000000e+00
   5 5.5763421800e-03  -4.82e-10  7.16e+02    1.146716e-08

After 5 iterations the fit converged.
final sum of squares of residuals : 0.00557634
rel. change during last iteration : -4.82184e-15

degrees of freedom    (FIT_NDF)                        : 14
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.0199577
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.00039831

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 1.14672e-08      +/- 7.199e-11    (0.6278%)
eliasmonge234@Main:~/Documents/GitHub/Algoritmica/P2_dyv_oficial/P2/Especifico/data$
```



Observamos que la varianza residual es casi nula, lo que significa un ajuste casi perfecto, respaldando nuestro resultado teórico. Esto se puede ver visualmente como que la curva pasa muy cerca de todos los puntos.

4.3. Algoritmo divide y vencerás

4.3.1. Diseño e implementación

Veamos el enfoque de Divide y Vencerás de este problema, que es la forma más natural de resolverlo. En primer lugar, debemos ver cómo vamos a subdividir el problema en problemas de tamaño menor.

Si dividimos la matriz en cuatro cuadrantes del mismo tamaño, que es posible porque n es potencia de dos, podemos ver que el hueco siempre pertenece a algún cuadrante de tamaño $\frac{n}{2} \times \frac{n}{2}$. Para el resto de cuadrantes, que son tres, podemos colocar una baldosa L en el medio del tablero, de forma que no tape el cuadrante en el que estaba el hueco, pero sí los otros 3.

Las 3 casillas pertenecen cada una a un cuadrante distinto, por lo que para rellenar cada uno de los 3 cuadrantes tenemos que rellenar todas sus casillas menos una, y tenemos una instancia del mismo problema de tamaño $\frac{n}{2}$.

Siguiendo este proceso recursivamente hasta un caso base de tamaño $UMBRAL$ que podemos resolver con el algoritmo de completar cuadrados ya tenemos un algoritmo Divide y Vencerás para este problema.

Para entenderlo mejor echemos un ojo a su implementación:

```
const int sx[4]={1,1,0,0};
const int sy[4]={1,0,1,0};
```

```
void fill_L_dyv(int n,int r, int c,int start_row, int start_col, vector<vector<int>> &
     v,int & tile){

    if(n <= UMBRAL){
        fill_L(v,n,r,c,tile);
        return;
    }

    // Divide in 4 (n is always a power of 2)
    int mid = n/2;
    int t = tile++;
```

```
11
12      // Locate (r,c) square and put L-shape tile on the center (excluding (r,c) square)
13      for(int i=0; i<4; ++i){
14
15          // New starting row and col of the square (midxmid)
16          //  x   .   x   .
17          //  .   .   .   .
18          //  x   .   x   .
19          //  .   .   .   .
20
21          int new_start_row = start_row + sx[i]*mid;
22          int new_start_col = start_col + sy[i]*mid;
23
24          // Locate (r,c) square
25          if(new_start_row <= r && r < new_start_row + mid && new_start_col <= c && c <
    new_start_col + mid){
26              fill_L_dyv(mid,r,c,new_start_row,new_start_col,v,tile);
27          }else{
28
29              // Fill the center with an L-shape and set the new square tile
30              //  .   .   .   .
31              //  .   .   x   .
32              //  .   x   x   .
33              //  .   .   .   .
34
35              int nr = start_row + mid - 1 + sx[i];
36              int nc = start_col + mid - 1 + sy[i];
37              v[nr][nc] = t;
38              fill_L_dyv(mid,nr,nc,new_start_row,new_start_col,v,tile);
39          }
40      }
41  }
```

En primer lugar describamos los parámetros que recibe. Recibe el lado del tablero `n`, la posición `(r,c)` del sumidero, la posición de la esquina superior izquierda de la matriz `(start_row, start_col)` para facilitar las llamadas recursivas, la matriz `v` y el número de losa `tile` que estamos colocando.

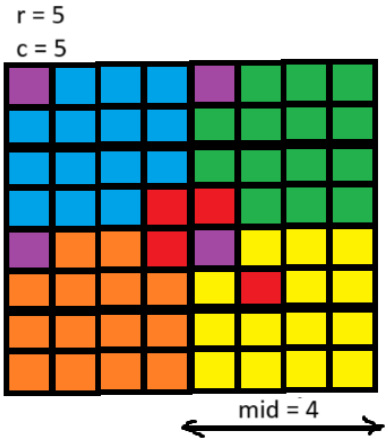
Sigamos el flujo del programa. Si `n` es inferior o igual a un determinado umbral `UMBRAL` entonces enlosamos el tablero con el algoritmo específico de completar cuadrados.

Si no, entonces entramos en un bucle `for` de cuatro iteraciones, que corresponderán a cada uno de los cuadrantes sobre los que vamos a llamar a la función recursivamente.
Primero calculamos la posición inicial del cuadrante, a partir de dos arrays estáticos `sx` y `sy` según la iteración en la que estemos.

Una vez tenemos la posición inicial, distinguimos el caso en el que el sumidero está en la submatriz obtenida, y llamamos a la función recursivamente con el mismo sumidero cambiando solo el tamaño y posiblemente la posición inicial.

Si en el cuadrante en el que estamos no está el sumidero, entonces insertamos una casilla con valor `t` (correspondiente al número de losa original) y la tratamos como sumidero en la siguiente llamada recursiva.

Para entenderlo visualmente, apoyémonos en el siguiente dibujo:



Los colores azul, verde, naranja y amarillo rellenan los cuadrantes sobre los que se van a hacer llamadas recursivas, y la casilla morada dentro de cada uno la posición inicial que se calcula. Las casillas rojas dentro de cada cuadrante serían lo que se va a interpretar como sumidero en cada llamada recursiva, siendo el del cuadrante amarillo el sumidero original.

4.3.2. Análisis de la eficiencia

Eficiencia teórica

Ahora vamos a estudiar cuál es la eficiencia teórica del algoritmo. Vemos que si definimos la función T que dado el tamaño del problema, $m = n^2$, devuelve el tiempo que tarda el programa en ejecutarse, entonces adopta la siguiente forma.

$$T(m) = \begin{cases} 1 & \sqrt{m} <= UMBRAL \\ & \forall n \in \mathbb{N} \\ 4T(\frac{m}{4}) + 1 & \sqrt{m} > UMBRAL \end{cases} \tag{9}$$

En el caso base, el tiempo está acotado por una constante luego podemos considerarlo en notación O grande como constante, por tanto es O(1), y en el caso general el código está compuesto en su totalidad por sentencias simples y un bucle que se ejecuta cuatro veces para repartir la tarea llamando cada vez a la función recursiva. Notese que para cada subllamada, el tamaño de entrada es un cuarto del original.

Supongamos que $\sqrt{m} > UMBRAL$ y efectuando el cambio de variable $4^t = m \implies t = \log_4 m = \log_4 n^2$, calculamos :

$$T(m) = T(4^t) = 4T(4^{t-1}) + 1 = 4(4T(4^{t-2}) + 1) + 1 = 4^2T(4^{t-2}) + 4 + 1 = 4^3T(4^{t-3}) + 4^2 + 4 + 1$$

Tras aplicar k veces:

$$T(m) = 4^kT(4^{t-k}) + \sum_{i=0}^{k-1} 4^i$$

Si $4^{t-k} \leq UMBRAL \iff t - \log_4 UMBRAL \leq k$, entonces tenemos que si elegimos $\bar{k} = t - \log_4 UMBRAL \implies T(4^{\bar{k}}) = 1$. Además, por la forma que hemos elegido t y teniendo en cuenta que n siempre es potencia de dos, podemos deducir que $\bar{k}$ con esta forma es un número entero, por tanto :

$$T(4^t) = 4^{\bar{k}}T(4^{t-\bar{k}}) + \sum_{i=0}^{\bar{k}-1} 4^i = 4^{\bar{k}} + \sum_{i=0}^{\bar{k}-1} 4^i = 4^{\bar{k}} + \frac{1 - 4^{\bar{k}}}{3} = \frac{1 + 2 * 4^{\bar{k}}}{3}$$

mathsf Deshaciendo el cambio de variable y teniendo en cuenta que nos interesa calcular la eficiencia según la notación O grande:

$$4^{\bar{k}} = 4^{t-\log_4 UMBRAL} = \frac{4^t}{UMBRAL} \iff 4^{\bar{k}} = \frac{m}{UMBRAL}$$

$$T(4^t) = \frac{1+2*4^{\bar{k}}}{3} = \frac{1+2*\frac{m}{UMBRAL}}{3} = T(m) \in O(m)$$

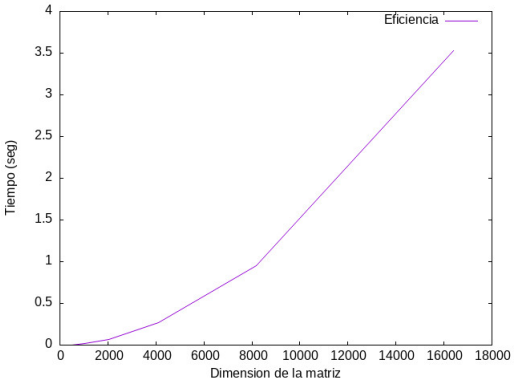
NOTA: Como $m = n^2$, según qué consideremos como tamaño de entrada en la análisis, cambia el orden de eficiencia en el siguiente sentido: si consideramos como tamaño de entrada el número de elementos que tendrá la matriz, entonces, como hemos probado, el algoritmo es lineal. Sin embargo, si consideramos como tamaño de entrada el número de filas/columnas de la matriz, entonces el algoritmo es cuadrático, puesto que $O(m) = O(n^2)$

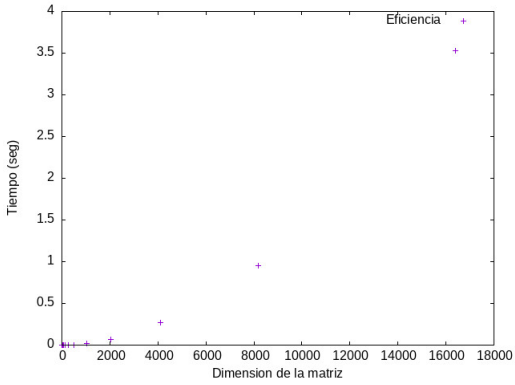
Eficiencia empírica

Con el algoritmo implementado y considerando como entrada el número de filas/columnas de la matriz, al ejecutar el programa con diferentes entradas obtenemos la siguiente tabla:

Nº filas/columnas	Tiempo(seg)
1	1.37E-07
2	2.11E-06
4	2.44E-06
8	3.80E-06
16	7.80E-06
32	2.42E-05
64	6.70E-05
128	0.000262055
256	0.00104772
512	0.00420025
1024	0.0171024
2048	0.0692816
4096	0.267032
8192	0.95623
16384	3.53244

Graficando los datos obtenidos obtenemos también:





Dado que se exige que el número de columnas/filas sean potencias de dos, podemos ver que los datos obtenidos están relativamente distados en la gráfica. Pero aun así, se puede apreciar que el crecimiento de los datos se parece a una curva cuadrática.

Eficiencia híbrida

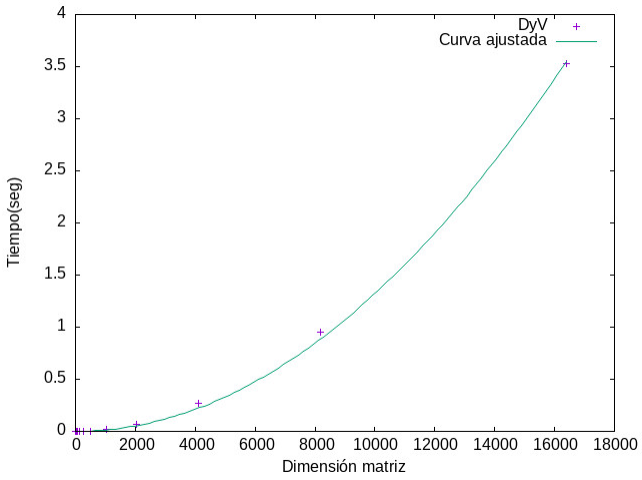
Como teóricamente hemos obtenido que el algoritmo tiene una eficiencia de orden cuadrático, vamos a ajustarla con una curva de la forma $f(x) = a_0 x^2$. Veamos los resultados:

$$f(x) = 1.32341 * 10^{-8} x^2$$

```
iter      chisq       delta/lim  lambda   a0
   0 7.6861431606e+16   0.00e+00  7.16e+07    1.000000e+00
   1 3.0023996721e+14  -2.55e+07  7.16e+06    6.250001e-02
   2 1.3326224316e+08  -2.25e+11  7.16e+05    4.165214e-05
   3 1.3189139420e-02  -1.01e+15  7.16e+04    1.351171e-08
   4 7.2664520901e-03  -8.15e+04  7.16e+03    1.323412e-08
   5 7.2664520901e-03  -3.58e-10  7.16e+02    1.323412e-08
iter      chisq       delta/lim  lambda   a0

After 5 iterations the fit converged.
final sum of squares of residuals : 0.00726645
rel. change during last iteration : -3.58096e-15

degrees of freedom    (FIT_NDF)                        : 14
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.0227823
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.000519032

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 1.32341e-08      +/- 8.218e-11    (0.6209%)
```

De los resultados podemos ver que la varianza residual es casi nula, lo cual quiere decir que nuestro ajuste se ajusta a nuestros datos casi a la perfección – verificando los resultados teóricos que afirman que la eficiencia del algoritmo es de $O(n^2)$.

Veamos la curva de regresión graficada con los datos:



Conclusión: En efecto, los resultados empíricos se ajustan a los teóricos.

4.3.3. Cálculo de umbrales

En esta sección calcularemos los umbrales teórico, óptimo y de tanteo, para posteriormente compararlos gráficamente.

Umbral teórico

Para el cálculo del umbral teórico, planteamos la ecuación del algoritmo específico tal cual ($h(n) = n^2$), la aplicamos en el primer nivel de recurrencia y resolvemos la ecuación (no recurrente):

$$T(n) = \begin{cases} n^2 & n \leq UMBRAL \\ \\ 4T(\frac{n}{2}) + 1 & n > UMBRAL \end{cases} \qquad \forall n \in \mathbb{N} \tag{10}$$

Entonces nos queda:

$$n^2 = 4\frac{n^2}{4} + 1 \iff 0 = 1$$

Con lo que concluimos que no podemos calcular el umbral teórico para este algoritmo. Este resultado se da por tener ambos algoritmos el mismo orden de eficiencia, y es una prueba más de que el umbral teórico es meramente indicativo.

Umbral óptimo

Para el cálculo el umbral óptimo, planteamos la función del algoritmo específico obtenida en la eficiencia híbrida ($h(n) = 1.14672 \cdot 10^{-8} \cdot n^2$), la aplicamos en el primer nivel de recurrencia y resolvemos la ecuación (no recurrente):

$$T(n) = \begin{cases} n^2 & n \leq UMBRAL \\ 4T(\frac{n}{2}) + 1 & n > UMBRAL \end{cases} \qquad \forall n \in \mathbb{N} \tag{11}$$

Entonces nos queda:

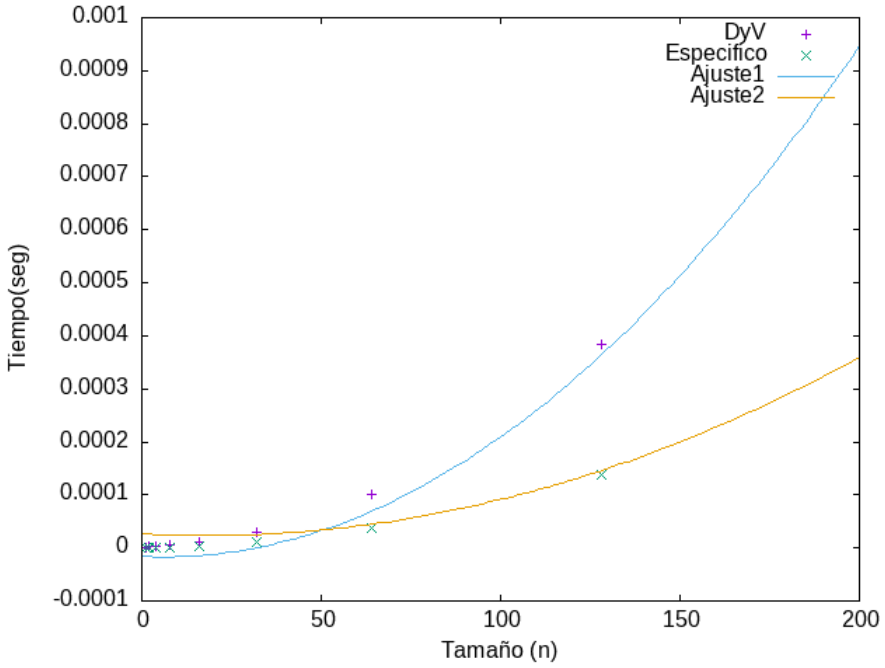
$$1.14672 \cdot 10^{-8} \cdot n^2 = 4\frac{1.14672 \cdot 10^{-8} \cdot n^2}{4} + 1 \iff 0 = 1$$

Al igual que con el umbral teórico, vemos que no tiene sentido resolver esta ecuación dado que ambos algoritmos tienen el mismo orden de eficiencia, y por tanto de forma teórica no tiene sentido tratar de calcular ningún umbral.

Umbral empírico

Al tener ambos códigos la misma eficiencia, como hemos visto, no tienen mucho sentido los cálculos teóricos para el umbral, por lo que tomaremos ahora un enfoque empírico para el cálculo del umbral.

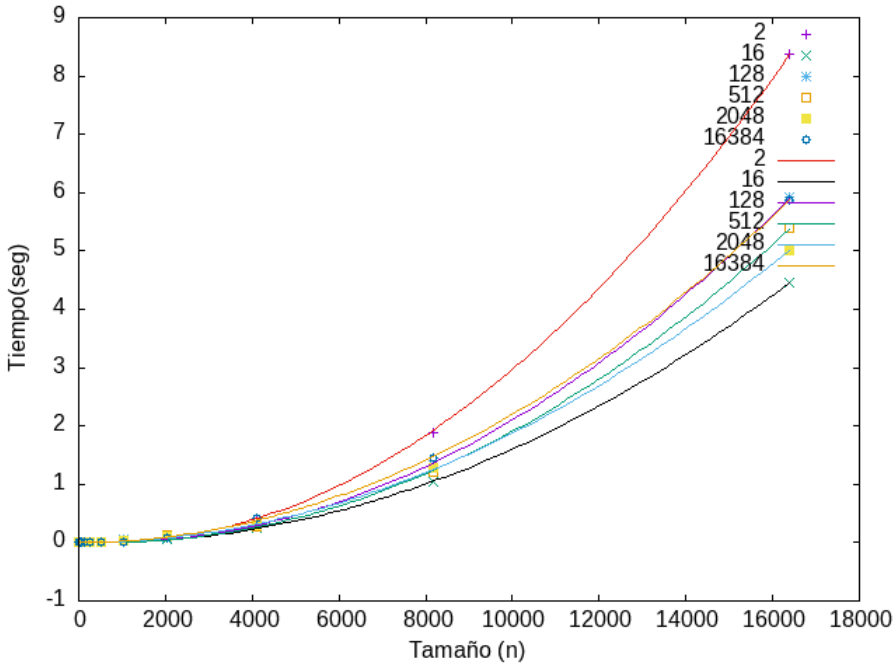
Para ello representamos la ejecución de ambos algoritmos (el divide y vencerás con umbral=0) y vemos a partir de qué tamaño n es más rápido (menor tiempo) el algoritmo divide y vencerás:



Como vemos, la gráfica representa la situación opuesta a la que esperábamos, ya que la gráfica amarilla del algoritmo específico queda por debajo de la del algoritmo divide y vencerás, algo que va completamente en contra de la filosofía del algoritmo. Esto ocurre ya que el algoritmo específico, aunque sea de la misma eficiencia que el algoritmo divide y vencerás, tiene mejores constantes ocultas que el algoritmo divide y vencerás, y por tanto es más eficiente. No obstante podemos destacar el tramo en el que el algoritmo divide y vencerás es más eficiente, que es aproximadamente entre 0 y 50.

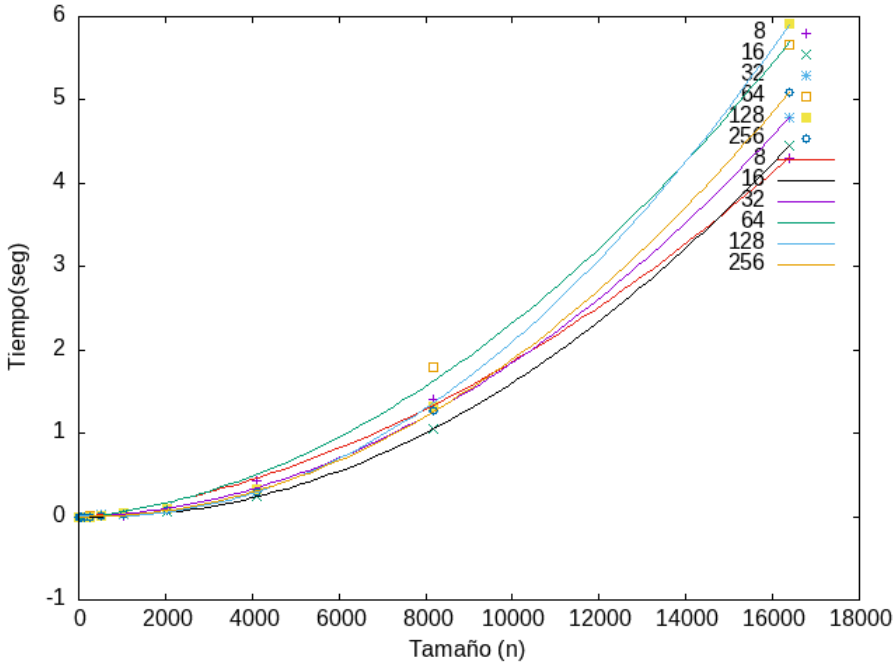
Umbral de tanteo

Aunque no hayamos obtenido un umbral empírico como tal, vamos a tratar de comparar diferentes umbrales en cuanto a tiempos de ejecución para ver qué ocurre. En principio podríamos suponer que a mayor umbral mayor será el tiempo de ejecución pues se delega más trabajo en el algoritmo más eficiente. Como los tamaños del problema son potencias de 2, cogeremos umbrales de tamaño potencia de 2. El resultado para los umbrales 2, 16, 128, 512, 2048 y 16384 es el siguiente:



Como vemos los tiempos son relativamente similares a excepción del caso $UMBRAL = 2$ que sí que es cierto que es más lento que el resto. También llama la atención que no se verifica nuestra hipótesis de que a mayor umbral menor tiempo, pues a pesar de que $UMBRAL = 16384$ es más rápido que la mayoría, $UMBRAL = 16$ es el más rápido de todos.

Podríamos pensar que entonces el umbral óptimo se hallaría cerca del 16. Probemos con umbrales cercanos:



En efecto, en orden de velocidad tendríamos primero el 8, después el 16, y después el 32, cercanos al 16. Curiosamente el siguiente sería el 256 mostrando que aunque no sea óptimo, es bueno coger umbrales grandes delegando gran parte del trabajo en el algoritmo específico.

Conclusión

Como conclusión tendríamos que el cálculo de umbrales no tiene mucho sentido cuando dos algoritmos tienen la misma eficiencia, al menos a nivel teórico. No obstante por tanteo descubrimos que el umbral ideal es $UMBRAL = 8$, aunque también son buenos los umbrales grandes al ser más rápido el algoritmo específico.

5. P3: Problema del viajante de comercio

5.1. Definición del problema

Tenemos un conjunto de n ciudades (puntos en un plano), cada una definida por las coordenadas en el mapa (x_i, y_i), con $i = 1, \ ... \ , n$. 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 (x_1, y_1)) 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 todas las ciudades consecutivas.

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.

5.1.1. struct City

Para resolver este problema con mayor comodidad y facilitar la modularización y legibilidad del código, se ha hecho uso de una *struct City* para representar las ciudades del problema.

```
typedef long long ll;
typedef long double ld;
```

Se ha definido también una constante INF para representar un valor de distancia imposible mayor que cualquier otro de los que pueda haber dentro del problema.

```

// Infinity (biggest possible number)
const ld INF = 1e18;
```

La *struct City* representa las ciudades del problema mediante sus coordenadas (x, y).

```
struct City
{
    ld x,y;
```

Además implementa la función *dist* que calcula la distancia euclídea de una ciudad a otra y tiene sobrecargado el operador $-$ para este mismo propósito.

```
    // Euclidean distance (symetrical)
    ld operator-(const City & other) const{
        return dist(other);
    }

    ld dist(const City & other) const{
        ld dx = x - other.x;
        ld dy = y - other.y;
        return sqrt(dx*dx+dy*dy);
    }
```

El operador $<$ también ha sido sobrecargado puesto que en nuestros algoritmos nos valimos de ordenar las ciudades respectod el eje x para dar con soluciones más óptimas y eficientes.

```
    // Sort by x axis
    friend bool operator<(const City & a, const City & b){
        if (a.x < b.x) {
            return true;
        }
        else if (a.x == b.x) {
            return a.y < b.y;
        }
        return false;
    }
```

Por último se han sobrecargado los operadores distinto (! =) y igual (==) por comodidad a la hora de trabajar con *City* y los operadores de entrada (≫) y salida (≪) para facilitar la lectura y escrituda de datos.

```
    friend bool operator==(const City & a, const City & b){
        return a.x == b.x && a.y == b.y;
    }

    friend bool operator!=(const City & a, const City & b){
        return !(a == b);
    }

    // I/O operators
    friend std::istream & operator>>(std::istream & is, City & p){
        char c;
        is >> c >> p.x >> c >> p.y >> c;
        return is;
    }
    friend std::ostream & operator<<(std::ostream & os, const City &p){
        os << "(" << p.x << "," << p.y << ")";
        return os;
    }
```

Para finalizar, se ha implementado la función *printCycle*() la cual recibe como parámetros el orden de los índices de las ciudades, la ciudad de origin y un array con las ciudades e imprime las ciudades empezando y acabando en la ciudad de origen en el orden indicado. Esto se ha hecho para facilitar la impresión de ciudades en el orden correcto teniendo en cuenta que el array de ciudades original ha sido ordenado.

```
void printCycle(const std::vector<int> & cycle, const City & origin, const City v[]){
    int ini = 0;
    while(v[cycle[ini]] != origin) ++ini;
    for(int i=ini; i<(int)cycle.size(); ++i){
        std::cout << v[cycle[i]] << " ";
    }
    for(int i=0; i<ini; ++i){
        std::cout << v[cycle[i]] << " ";
    }
    std::cout << origin << std::endl;
}
```

5.2. Algoritmo específico

5.2.1. Diseño e implementación

Diseño

El algoritmo específico elegido para resolver el problema del viajante de comercio ha sido uno que nos de la solución óptima al problema, a costa de un gran coste de eficiencia. La única forma conocida de dar con la solución óptima al problema es probar todos los ciclos posibles y quedarnos con el ciclo óptimo de todos ellos, es decir, resolver el problema mediante fuerza bruta. No obstante, hemos introducido pequeñas mejoras que aunque no afectan a la complejidad del algoritmo en el caso general (el orden de eficiencia sigue siendo el mismo) si que mejoran los tiempos obtenidos en media.

Para explorar todos los ciclos posibles, partimos de la ciudad de origen, y de ahí vamos a la primera ciudad que no hayamos visitado. Procedemos a visitarla añadiendola a nuestro ciclo actual y acumulando la distancia de la ciudad origen a esta. Una vez hecho eso repetimos el proceso solo que desde la última ciudad visitada, así hasta a ver visitado todas las ciudades en cuyo caso ya tenemos uno de los ciclos posibles. Una vez completada la investigación de este ciclo probamos a visitar la segunda ciudad posible desde la última ciudad visitada y así sucesivamente. Es decir, si tenemos en total $n = 4$ ciudades numeradas del 0 al 3, con la ciudad de origen el 0, partimos de la ciudad 0. Como no hemos visitada ningún otra ciudad, visitamos la ciudad 1, después la 2 y por último la 3. Una vez completado el ciclo, volvemos a la ciudad 2 y vemos si podemos visitar otra ciudad distinta a la 3. Como no es el caso, volvemos a estar en la ciudad 1, y en este caso en vez de visitar la ciudad 2 visitamos la 3 y así sucesivamente.

Todos los caminos recorridos pueden verse en la figura 8 donde cada flecha representa una llamada a la función a visitar la ciudad apuntada por la flecha con las ciudades previas a la flecha visitadas.
La mejora implementada consiste en a medida que se va calculando el coste de un ciclo, es decir, a

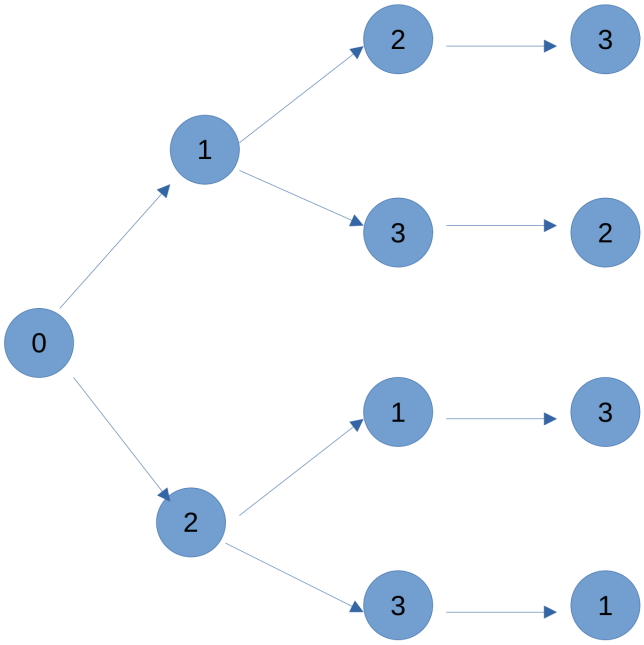
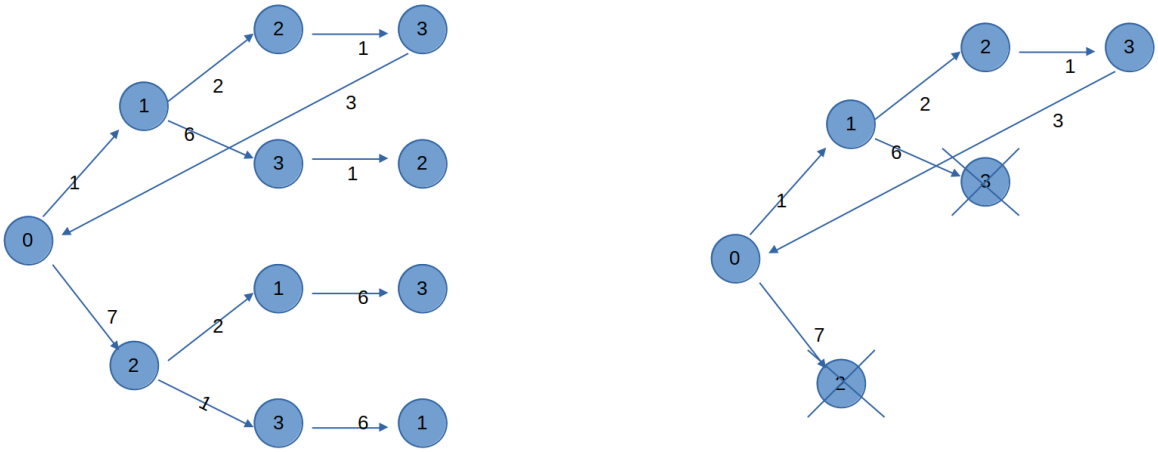


Figura 8: Todos los caminos posibles para n = 4

medida que se van visitando ciudades y acumulando las distancias recorridas, en el momento en el que la distancia recorrida actual es mayor o igual que el coste de nuestro mejor ciclo hasta la fecha, abandonamos ese camino por completo dado que no nos va a proporcionar una solución mejor a la que ya tenemos puesto que todas las distancias son estrictamente positivas.
Veamoslo con un ejemplo. En la figura 9a podemos ver el arbol de recorridos para $n = 4$ ciudades con unas distancias de ejemplo (las distancias escogidas nos son reales dado que no cumplen las propiedades geométricas de la distancia euclídea, no obstante sirven para ejemplificar el algoritmo). En la figura 9b podemos ver los recorridos que haría el algoritmo de todos los que hay, empleando que el primer recorrido que hace es el de las ciudades en orden y que en el momento en el que ir a una ciudad le supone que lleva una distancia mayor al coste mínimo actual no la visita.
En el ejemplo podemos ver como el primer camino recorrido tiene coste 7, y por tanto cuando el al-



(a) Todos los caminos con sus costes

(b) Caminos recorridos por el algoritmo

goritmo detecta que ir a una ciudad le supone ya un coste 7 o superior abandona el camino "podando

esa ramaz pasa a la siguiente opción. De esa forma los caminos (llamadas a la función) recorridos en la figura 9b son muchos menos (muchas menos llamadas recursivas) que en la figura 9a.
Como hemos podido observar, la mejora en eficiencia se aprovecha más cuanto menor es el coste del primer ciclo explorado, el cual siempre es el array de ciudades en orden. Por este motivo hemos optado por ordenar previamente el array para garantizar que el primer ciclo explorado es relativamente"bueno en comparación al todos los demás ciclos posibles.

Implementación
Para implementar el algoritmo hemos empleado una función recursiva que tiene como parámetros:

- n: número total de ciudades a visitar
- *prev*: el índice de la última ciudad visitada
- *homeind*: el índice de la ciudad de origen
- *cnt*: número de ciudades visitadas
- v: el array con las ciudades
- *visited*: un array que indica si hemos visitado la ciudad i o no
- *dist*: distancia actual
- *bestdist*: mejor distancia encontrada hasta ahora (inicialmente se pone a INF)
- *curcycle*: camino recorrido hasta ahora
- *bestcycle*: mejor ciclo encontrado hasta ahora

```
void TSP_branch_bound(int n,int prev,int home_ind, int cnt,const City v[],bool visited
    [], ld dist, ld & best_dist, vector<int> cur_cycle, vector<int> &best_cycle){
```

- **Caso base:**
  Consiste en haber visitado ya todas las ciudades $(n-1)$ y por tanto solo queda añadir la distancia de la última ciudad visitada a la ciudad de origen para tener el coste del ciclo total. Una vez hecho eso comprobamos si este coste es menor que el de nuestro ciclo más óptimo hasta la fecha y si es así lo actualizamos.

```
    // Base case
    if(cnt == n-1){ // Already visited all cities
        // Add distance of the last city to the origin city
        ld new_dist = dist + v[prev].dist(v[home_ind]);
        // If the new distance is better than current best distance
        // best distance and best cycle are updated
        if(new_dist < best_dist){
            best_dist = new_dist;
            best_cycle = cur_cycle;
        }
        return;
    }
```

- **Caso general:**
  Si todavía no hemos visitado todas las ciudades, entonces iteramos por todas ellas hasta dar con una que no hayamos visitado todavía, en cuyo caso la visitamos marcándola como visitada, añadiendola a nuestro camino actual y sumando la distancia de la ciudad elegida a nuestra última ciudad visitada a nuestra distancia actual.

```
    // Iterate for all the cities to visit
    for(int i=0; i<n; ++i){
        // If the city hasn't been visited we try visiting it
        if(!visited[i]){
            visited[i] = true; // Mark the city as visited
            // Add distance of the last city to this city to current distance
            ld new_dist = dist + v[i].dist(v[prev]);
```

La mejora introducida es la siguiente: normalmente, si quisieramos explorar todos ciclos posibles, llamaríamos a nuestra función recurrentemente con la nueva ciudad visitada. No obstante, como todas las distancias son positivas, sabemos que si nuestra distancia actual (*dist*) es mayor o igual que nuestra distancia más óptima hasta ahora, tenemos garantizado que llendo por este camino no podremos mejorar nuestra mínima hasta el momento, por tanto seguir ese camino no nos aporta nueva información y nos hace perder un valioso tiempo. Es por eso que si la distancia actual es peor que el coste de nuestro ciclo mínimo actual, no la visitamos y simplemente pasamos a probar con la siguiente ciudad posible.
En caso contrario, si que llamamos a nuestra función recursiva con una ciudad más visitada para explorar todos los posibles caminos con las ciudades que nos faltan por visitar. Una vez completado esto, volvemos a marcar la ciudad como no visitada y a eliminarla del camino, y pasamos a probar a visitar la siguiente ciudad posible.

```
            // If the new distance is better than current best distance
            // we try visiting this city
            if(new_dist < best_dist){
                // Add city to current cycle
                cur_cycle.push_back(i);
                // Proceded resolving the problem with one more city visited
                TSP_branch_bound(n,i,home_ind,cnt+1,v,visited,new_dist,best_dist,
    cur_cycle,best_cycle);
                // Erase the city of the current cycle
                cur_cycle.pop_back();
            }
            visited[i] = false; // Mark the city as not visited
        }
    }
```

Es fácil ver, que nuestra mejora será más efectiva cuanto menor sea el coste del primer ciclo, puesto que así será más probable que explorar otros caminos sea innecesario. Por ello, antes de llamar a la función, ordenamos el array de las ciudades respecto del eje x, ya que el primer ciclo explorado de todos es el array en orden, y las ciudades ordenadas por el eje x aunque no sean el ciclo óptimo nos garantizan que las ciudades consecutivas estarán más o menos cerca entre sí y por tanto no será una solución del todo "mala".

```
    sort(v,v+n); // sort by x axis
```

5.2.2. Análisis de eficiencia

Eficiencia teórica

De la misma forma que los casos anteriores, definimos la función $T : \mathbb{N} \to \mathbb{R}$ que toma un número natural n como tamaño de la entrada (número de ciudades para las que calcular el camino mínimo necesario para pasar por cada una de ellas) y nos retorna el número de operaciones que necestiamos realizar (en el peor caso) para obtener un resultado para ese tamaño usando un enfoque de fuerza brutal. El código que implementa la solución ya explicada es el siguiente:

```
// TODO compatibilidad entre el brute force y el eficiente

const int UMBRAL = 2;

/**
 * @brief Branch and bound solution to the Travelling Salesman Problem (path).
 * @param n number of cities to visit
 * @param prev index of the previous city
 * @param cnt number of visited cities
 * @param v array of cities to visit
 * @param visited whether city i has been visited or not
 * @param dist distance of current cycle
 * @param best_dist best cycle distance so far
 * @param cur_cycle current cycle
 * @param best_cycle best cycle so far
*/
void TSP_branch_bound(int n,int prev, int cnt,const City v[],bool visited[], ld dist,
    ld & best_dist, vector<int> cur_cycle, vector<int> &best_cycle){
    if(cnt == n){ // Base case
        ld new_dist = dist + v[prev].dist(v[0]);
```

```
        if(new_dist < best_dist){
            best_dist = new_dist;
            best_cycle = cur_cycle;
        }
        return;
    }

    for(int i=0; i<n; ++i){
        if(!visited[i]){
            visited[i] = true;
            ld new_dist = dist + v[i].dist(v[prev]);
            if(new_dist < best_dist){
                cur_cycle.push_back(i);
                TSP_branch_bound(n,i,cnt+1,v,visited,new_dist,best_dist,cur_cycle,
    best_cycle);
                cur_cycle.pop_back();
            }
```

Observando el código es claro que, en el peor de los casos (que se efectúen todas las iteraciones del bucle *for* posibles en cada una de las llamadas recursivas) nos encontramos con que para resolver el problema de tamaño n se hacen n llamadas a recursivas sobre problemas de tamaño $n-1$, luego:

$$T(n) = \begin{cases} 1 & n = 0 \\ \\ n \cdot T(n-1) + 1 & n > 0 \end{cases} \qquad \forall n \in \mathbb{N} \tag{12}$$

Demos un esquema de la demostración por inducción del siguiente hecho:

$$T(n) = n! \cdot \sum_{k=0}^{n} \frac{1}{k!}$$

Demostración. Caso $n = 0$:

$$1 = T(0) = 0! \cdot \sum_{k=0}^{0} \frac{1}{k!}$$

Caso $n \geq 1$: Suponemos $T(n-1)$ se tiene.

$$T(n) = n \cdot T(n-1) + 1 = 1 + n \cdot (n-1)! \cdot \sum_{k=0}^{n-1} \frac{1}{k!} = n! \cdot (\sum_{k=0}^{n-1} \frac{1}{k!} + \frac{1}{n!}) = n! \cdot \sum_{k=0}^{n} \frac{1}{k!}$$

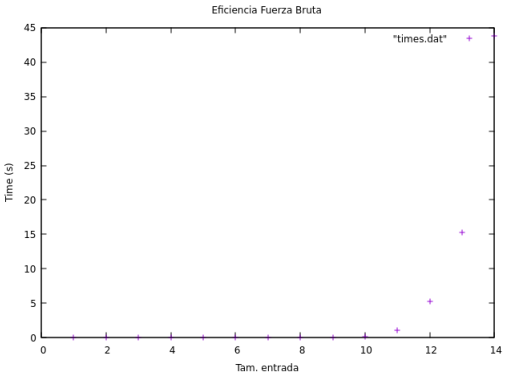
□

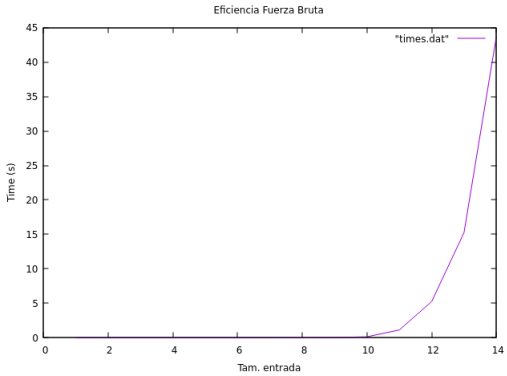
Por tanto, tenemos de manera clara (por la regla del máximo) que la eficiencia de este algoritmo es del orden $O(n!)$

Eficiencia empírica

Usamos de manera análoga a los casos ya vistos en anteriores ocasiones la herramienta de gnuplot para representar gráficamente los resultados de tiempos de ejecución obtenidos para distintos tamaños de entrada del problema. Tomamos un rango de tamaño de 1 a 14, pues al ser factorial crece súmamente rápido y el tiempo de ejecución es inasumbible para mi máquina con tamaños superiores a ese.

Los datos obtenidos son los siguientes:


Eficiencia Fuerza Bruta
"times.dat"
Time (s)
Tam. entrada
0
5
10
15
20
25
30
35
40
45
0
2
4
6
8
10
12
14


Eficiencia Fuerza Bruta
"times.dat"
Time (s)
Tam. entrada
0
5
10
15
20
25
30
35
40
45
0
2
4
6
8
10
12
14

Eficiencia híbrida
Para hacer un ajuste, intentamos poner nuestros resultados como una función $f(x) = c_0 \cdot n!$. El resultado obtenido usando *gnuplot* es el siguiente:

```
gnuplot> f(x) = a0*gamma(x+1)
gnuplot> fit f(x) 'times.dat'via a0
iter      chisq       delta/lim  lambda   a0
   0 1.7363043845e+02   0.00e+00  1.20e+01    5.119012e-10
   1 1.7363043845e+02  -3.27e-11  1.20e+00    5.119012e-10
iter      chisq       delta/lim  lambda   a0

After 1 iterations the fit converged.
final sum of squares of residuals : 173.63
rel. change during last iteration : -3.27382e-16

degrees of freedom    (FIT_NDF)                        : 13
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 3.65461
variance of residuals (reduced chisquare) = WSSR/ndf   : 13.3562

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 5.11901e-10      +/- 4.181e-11    (8.168%)
gnuplot> plot 'times.dat', f(x) title 'Curva ajustada'
```

Figura 11: Comandos usados en gnuplot

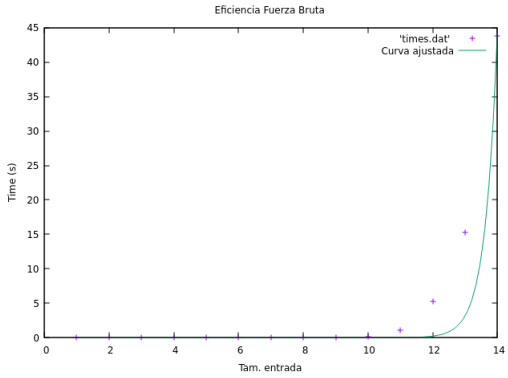


Figura 12: Función de ajuste $5.110 \cdot 10^{-10} \cdot fact(n)$

Nótese que, naturalmente, no hemos utilizado la función factorial (que está definida en los naturales) sino una extensión suya a la los reales positivos, la función Gamma:

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t}\, dt \quad \forall z \in \mathbb{R}^+$$

Pues se puede demostrar que:

$$\Gamma(n+1) = n! \qquad \forall n \in \mathbb{N}$$

5.3. Algoritmo divide y vencerás

5.3.1. Diseño e implementación

Diseño
A la hora de abordar la implementación divide y vencerás de este problema, hemos tenido en cuenta que se trata de un problema NP-Difícil y por tanto no se puede obtener su solución óptima en tiempo polinómico con nuestros ordenadores actuales (se puede calcular en tiempo polinómico con una

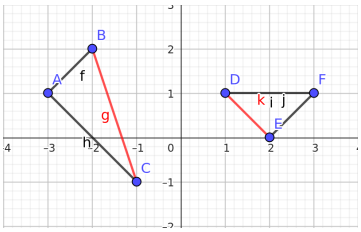
máquina de Turing no determinista). Por tanto, hemos optado por un algoritmo que de una solución **aproximada**.
Tras probar distintos enfoques, hemos concluido que la forma más eficiente y que da la solución más óptima (el error respecto a la solución óptima real es menor) de aplicar divide y vencerás a este problema es la siguiente:

a) Ordenar las ciudades respecto a su coordenada x

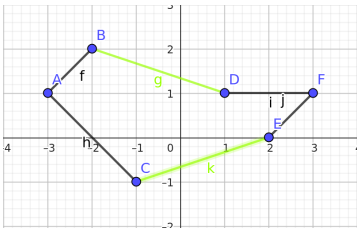
b) Partir el array ordenado por la mitad, diviendolo en dos subgrupos del mismo tamaño (aprox) cuyos integrantes se encuentran cerca los unos de los otros (respecto al eje X)

c) Una vez obtenida la solución para ambos grupos, gracias a que sabemos por donde partimos el array, conocemos las dos ciudades más próximas respecto a eje X entre ambos ciclos. Para cada una, se escoge una de sus ciudades adyadcentes, en particular la que este más centrada respecto al eje X.

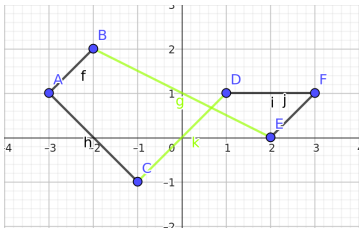
d) Ya seleccionadas ambas parejas, se separan y se une cada uno de ellas con un miembro de la pareja del otro ciclo. Hay dos formas, se realiza de la que sea más eficiente.



(a) Estado inicial

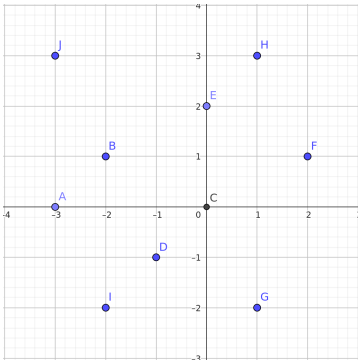


(b) Opción 1

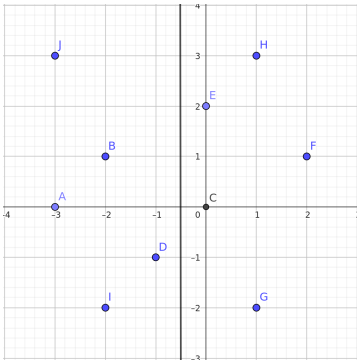


(c) Opción 2

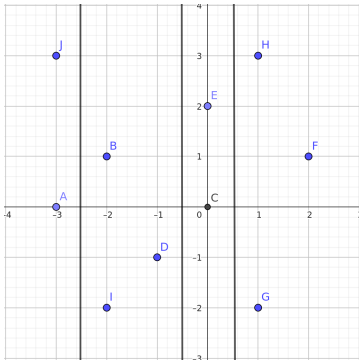
Veamos el algoritmo más claro con el siguiente ejemplo:



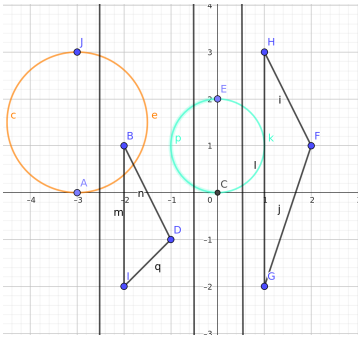
(a) Estado inicial



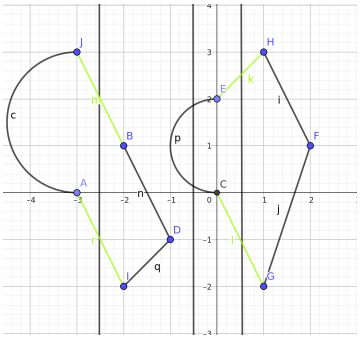
(b) Partimos



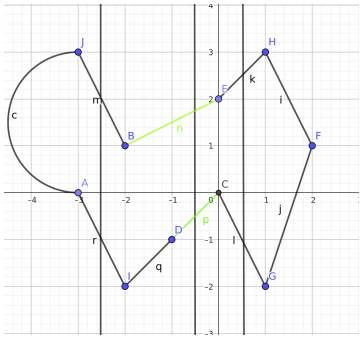
(c) Partimos de nuevo



(d) Casos base

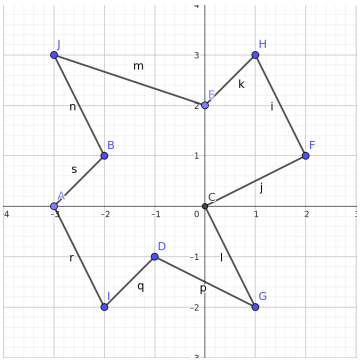


(e) Unimos



(f) Unimos de nuevo

Como podemos observar, la solución dada por el algoritmo no es "mala"pero tampoco es la óptima. La solución óptima se puede ver el la figura 15a. Para ser exactos el coste del ciclo de nuestro algoritmo



(a) Solución óptima

```
10
(-3, 0)
(-2, 1)
(0, 0)
(-1, -1)
(0, 2)
(2, 1)
(1, -2)
(1, 3)
(-2, -2)
(-3 ,3)
```

(b) Input data

es 21.5853 mientras que el coste del ciclo óptimo devuelto por la solución específica es 20.8213. Por tanto el error cometido en este caso es:

$$error = \frac{(21.5853 - 20.8213) \cdot 100}{20.8213} = 3.6693194\,\%$$

Hemos comparado el coste de la solución dada por el algoritmo divida y vencerás y el de la solución dada por el algoritmo específico que da la solución óptima para 1000 casos de pruebas de tamaño $n = 10$ (es decir, 10 ciudades) y la solución dada por el algoritmo divide y vencerás tiene de media un 26.15 % de error relativo (respecto al verdadero coste mínimo) con $UMBRAL = 4$. Con $UMBRAL = 2$ que se verá más adelante que es el umbral óptimo en tiempo de ejecución el error relativo medio es 27.38 %.

Implementación

Para implementar este algoritmo, previamente a la llamada de la función que resuelve el algoritmo se ha ordenado el array de las ciudades en base a su coordenada x, de forma que las ciudades consecutivas son las más próximas respecto del eje X.

```
1     sort(v,v+n); // sort by x axis
```

- **Divide:**
  Nuestra forma de partir el problema ha sido dividir las ciudades por la mitad. Al estar las ciudades ordenadas por el eje X, tenemos que las ciudades que componen los dos grupos entre sí están relativamente próximas (lo están respecto del eje X por lo menos).

```
1     //Divide
2     vector<int> path1,path2;
3     int mid = (fin+ini)/2;
4     dyv(ini,mid, v, path1);
5     dyv(mid,fin, v, path2);
```

- **Vencerás (fusión):**
  Tras las llamadas a la función tenemos el ciclo de coste mínimo de cada uno de los dos subgrupos. Como nuestro array de las ciudades está ordenado, podemos sabar en tiempo constante cuales son las 2 ciudades más próximas entre sí de ambos ciclos (1 de cada uno) ya que sus índices corresponden a los más cercanos al índice por el cual se partió el array. Por tanto, localizamos las ciudades dentro de sus respectivos ciclos.

```
1     // Most x-centered cities
2
3     int r_nearest_city = mid;
4     int l_nearest_city = mid-1;
5
6
7     int x; // index of most x-centered city in path1
8     for (int i=0; i < mid-ini; ++i) {
9         if (path1[i] == l_nearest_city) {
```

```
            x = i;
            break;
        }
    }

    int z; // index of most x-centered in path2
    for (int i=0; i < fin-mid; ++i) {
        if (path2[i] == r_nearest_city) {
            z = i;
            break;
        }
    }
```

Una vez localizadas, vemos cuales de sus ciudades adjadcentes está más centrada respecto a su coordenada x.

```
    // Calculate which of the x adjacent cities (y1,y2) is more x-centered (y = y1
     or y2)

    int inc_x = 1; // y = x + inc_x
    int y1 = (x+1)%(mid-ini), y2 = (x+mid-ini-1)%(mid-ini), y = y1;
    if (path1[y1] < path1[y2]) {
        inc_x = -1;
        y = y2;
    }

    // Calculate which of the z adjacent cities (t1,t2) is more x-centered (t = t1
     or t2)

    int inc_z = 1; // t = z + inc_z
    int t1 = (z+1)%(fin-mid), t2 = (z+fin-mid-1)%(fin-mid), t = t1;
    if (path2[t2] < path2[t1]) {
        inc_z = -1;
        t = t2;
    }
```

Tenemos entonces las 4 ciudades de ambos ciclos más cercanas entre sí conectadas si son del mismo ciclo. Si nuestras ciudades son x e y del primer ciclo y z y t del segundo, calculamos si es más barato uni x con z e y con t o x con t e y con z.

```
    // Calculate distances of possible links

    ld xz = v[path1[x]].dist(v[path2[z]]);
    ld xt = v[path1[x]].dist(v[path2[t]]);
    ld yz = v[path1[y]].dist(v[path2[z]]);
    ld yt = v[path1[y]].dist(v[path2[t]]);

    // true if it's better to link x and z (then y and t), false if it's better to
     link x and t
    // (then y and z)
    bool link_xz = xt + yz > xz + yt;
```

Por último unimos ambos ciclos de la manera determinada en el paso anterior (sin olvidar romper la unión entre x e y además de t y z, lo que implica restar estas distancias). La forma de unirlos es el primer ciclo desde y hasta x más el segundo ciclo desde z hasta t si x se une con t o al revés en caso contrario.

```
    // Push path1 to path from y to x going through the vector in cycle
    for (int i=y; i != x; i = (i+mid-ini+inc_x)%(mid-ini)) {
        path.push_back(path1[i]);
    }
    path.push_back(path1[x]);

    int start_z = link_xz ? z : t;
    int finish_z = start_z == z ? t : z;
    inc_z = start_z == z ? -inc_z : inc_z;

    // Push path2 to path from start_z to finish_z going through the vector in
    cycle
    for (int i=start_z; i != finish_z; i = (i+fin-mid+inc_z)%(fin-mid)) {
```

```
        path.push_back(path2[i]);
    }
    path.push_back(path2[finish_z]);

    // Adjust indexes to v size
    for (int i=0; i < path.size(); ++i) {
        path[i] -= ini;
    }
```

5.3.2. Análisis de eficiencia

Eficiencia teórica

Procedamos con el estudio de la eficiencia teórica. Definamos primero la función $T : \mathbb{N} \to \mathbb{R}$, donde dado el tamaño de entrada n, que es el número de ciudades a visitar, nos devuelve el tiempo empleado en ejecutar el programa que resuelve el problema con nuestro algoritmo. Viendo el código, es fácil ver que toma la siguiente forma:

$$T(n) = \begin{cases} n! & n <= UMBRAL \\ \\ 2T(\frac{n}{2}) + n & n > UMBRAL \end{cases} \qquad \forall n \in \mathbb{N} \tag{13}$$

Hay que tener en cuenta que estamos realizando un análisis con la notación O grande, por tanto, como el tiempo de ejecución del caso base está acotado por una constante, concluimos que es de la eficiencia O(1). En el caso general, vemos que realiza dos llamadas recursivas a la propia función con tamaño de entrada $\frac{n}{2}$ y, aparte de sentencias simples booleanas y condicionales de eficiencia O(1), realiza 8 bucles de tamaño $\frac{n}{2}$, así utilizando la regla de la suma y del máximo concluimos que en el caso general nuestra función es de la forma $T(n) = 2T(\frac{n}{2}) + n$.

Para resolver esta recurrencia, realizamos un cambio de variable $n = 2^m$:

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

Desarrollando k veces:

$$T(2^m) = 2^k * T(2^{m-k}) + k2^m$$

Teniendo en cuenta que $2^{m-k} \leq UMBRAL \iff m - \log_2 UMBRAL \leq k$, tomamos $\bar{k} = [m - \log_2 UMBRAL] + 1$, entonces $T(2^{m-\bar{k}}) = (2^{m-\bar{k}})!$, desarrollamos:

$$T(2^m) = 2^{\bar{k}} * T(2^{m-\bar{k}}) + \bar{k}2^m = 2^{\bar{k}} * (2^{m-\bar{k}})! + \bar{k}2^m$$

Ahora, sustituyendo y deshaciendo el cambio de variable $m = log_2 n$ y $\bar{k} = [m - \log_2 UMBRAL] + 1$:

$$\begin{aligned} T(2^m) &= 2^{\bar{k}} * (2^{m-\bar{k}})! + \bar{k}2^m \\ &= 2^{[m-\log_2 UMBRAL]+1} * (2^{m-\bar{k}})! + ([m - \log_2 UMBRAL] + 1) * 2^m \\ &= 2^{[m-\log_2 UMBRAL]+1} * (2^{m-[m-\log_2 UMBRAL]-1})! + ([m - \log_2 UMBRAL] + 1) * 2^m \\ &== 2^{[\log_2 n-\log_2 UMBRAL]+1} * (2^{\log_2 n-[\log_2 n-\log_2 UMBRAL]-1})! + ([\log_2 n - \log_2 UMBRAL] + 1) * n \\ &= 2^{[\log_2 n-\log_2 UMBRAL]+1} * (2^{[\log_2 UMBRAL]-1})! + ([\log_2 n - \log_2 UMBRAL] + 1) * n \\ &= T(n) \end{aligned}$$

Como estamos trabajando con la notación O grande, nos interesa únicamente el comportamiento asintótico, y por tanto, ignorando las constantes:

$$O(T(n)) = O(2^{\log_2 n} + \log_2 n * n) = O(n + n \log_2 n) = O(n \log n)$$

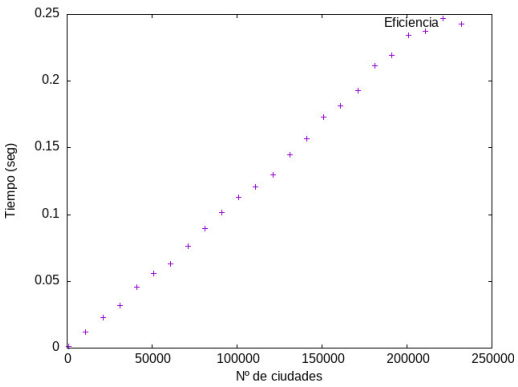
Por tanto, concluimos que nuestro algoritmo tiene una eficiencia teórica de $O(n \log n)$.

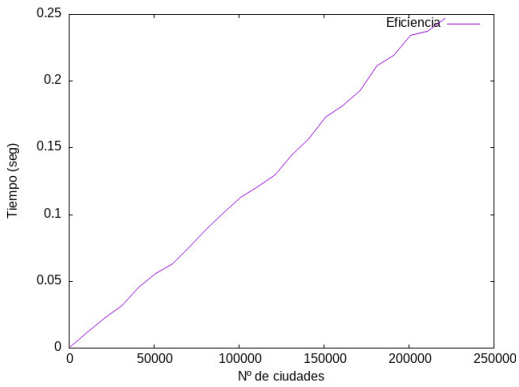
Eficiencia empírica

Con el algoritmo implementado, ejecutamos el código con distintas instancias del problema (generados por el generador de casos), y obtenemos la siguiente tabla:

Nº ciudades	Tiempo(seg)
1000	0.000940859
11000	0.0117779
21000	0.0227117
31000	0.0317387
41000	0.0457
51000	0.0557699
61000	0.0632166
71000	0.0760993
81000	0.089748
91000	0.101802
101000	0.113085
111000	0.120791
121000	0.130018
131000	0.144707
141000	0.156846
151000	0.17327
161000	0.181568
171000	0.193043
181000	0.211589
191000	0.219642
201000	0.234674
211000	0.237546
221000	0.246918

Como era de esperar, hemos definido como tamaño de entrada el número de ciudades. Ahora graficando los datos obtenemos las gráficas:





Podemos observar que los datos siguen un crecimiento que parece o lineal o superlineal, y eso concuerda con el análisis teórico realizado, ya que el algoritmo es de eficiencia $O(n \log n)$.

Eficiencia híbrido

Veamos qué pasa si ajustamos una curva de regresión de la forma $f(x) = a_0 x \log n$ (función obtenida teóricamente) a los datos obtenidos empíricamente:

```
iter      chisq       delta/lim  lambda   a0
   0 5.5372437957e+13   0.00e+00  1.55e+06    1.000000e+00
   1 9.6132704786e+10  -5.75e+07  1.55e+05    4.166676e-02
   2 1.8156742364e+04  -5.29e+11  1.55e+04    1.820217e-05
   3 2.8333015546e-04  -6.41e+12  1.55e+03    9.418180e-08
   4 2.8298693081e-04  -1.21e+02  1.55e+02    9.410307e-08
   * 2.8298693081e-04   7.66e-11  1.55e+03    9.410307e-08
   * 2.8298693081e-04   7.66e-11  1.55e+04    9.410307e-08
   * 2.8298693081e-04   1.72e-10  1.55e+05    9.410307e-08
   5 2.8298693081e-04  -2.87e-10  1.55e+04    9.410307e-08
iter      chisq       delta/lim  lambda   a0

After 5 iterations the fit converged.
final sum of squares of residuals : 0.000282987
rel. change during last iteration : -2.87346e-15

degrees of freedom    (FIT_NDF)                        : 22
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.00358651
variance of residuals (reduced chisquare) = WSSR/ndf   : 1.2863e-05

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 9.41031e-08      +/- 4.82e-10     (0.5122%)
```

Vemos que con este ajuste, la varianza residual es practicamente nula, es decir, la curva se ajusta casi a la perfección a nuestros datos – verificando nuestras conclusiones teóricas de que el algoritmo es de orden $O(n \log n)$.

Graficando la función junto con nuestros datos obtenemos:

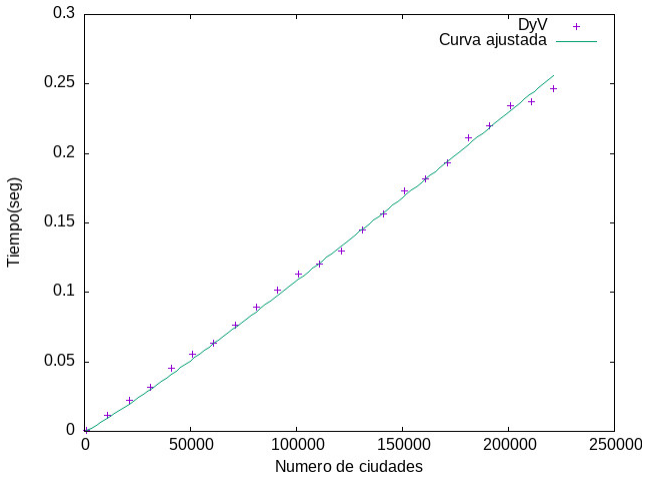


Figura 17: $f(x) = 9.41 * 10^{-8} x \log x$

Conclusión: En definitiva, nuestro algoritmo tiene la eficiencia calculada por el análisis teórico, y la curva de regesión se ajusta a nuestros datos empíricos.

5.3.3. Cálculo de umbrales

El problema del cálculo del umbral es de fundamental importancia si pretendemos algoritmos rápidos usando la técnica *Divide y vencerás*. Proponemos, como en los otro probleas, diferentes aproximaciones a la cuestión:

Umbral teórico

Para el cálculo del umbral teórico procedemos como hemos visto en teoría: usando el tiempo de ejecución teórico del algoritmo con la solución específica (fuerza bruta, en este caso) y la expresión recurrente del tiempo de ejecución para el método recursivo con Divide y Vencerás (suponemos una única ejecución recursiva).

Por tanto, tenemos que hallar n_0 tal que los siguientes valores coincidan, y será el valor que pondremos para el umbral teórico.

$$T(n) = \begin{cases} n! & \text{si } n \leq \text{UMBRAL} \\ 2T(\frac{n}{2}) + n & \text{si} n \geq \text{UMBRAL} \end{cases} \tag{14}$$

Es decir:

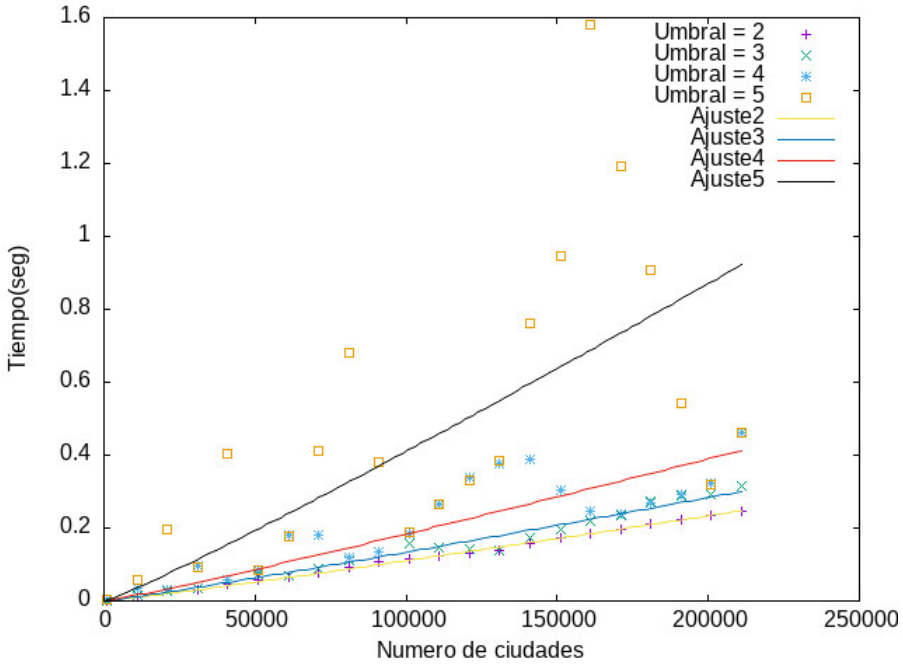
$$n_0\,! = 2 \cdot \left(\frac{n_0}{2}\right)! + n_0$$

Esta ecuación no es bajo ningún concepto fácil de resolver, y de hecho no tiene solución en los naturales. Sin embargo, numericamente obtenemos que, utilizando la extensión real con la función Γ hay una solución alrededor de $2.93527687137463\ldots$ Así, usaremos 3 como umbral teórico.

Umbrales de tanteo

Procedemos ahora a, habiendo obtenido un umbral teórico, probar los tiempos de ejecución para diferentes umbrales cercanos a éste. En particular, probamos con $n_0 = 4$ $n_0 = 5$ $n_0 = 2$ (que es el ya calculado) aparte de, claramente $n_0 = 3$. teórico.

Siguiendo exactamente el mismo procedimiento fue usado para el cálculo de la eficiencia empírica de los algoritmos, obtenemos la siguiente gráfica totalmente explicativa donde se comparan las ejecuciones con diferentes tamaños variando el umbral:



Observamos de manera clara la diferencia de calidad entre los umbrales, siendo 2 y 3 de buena calidad y 4 y 5 (especialmente este último) de muy mala calidad, presentando unos resultados casi caóticos.

Umbral óptimo

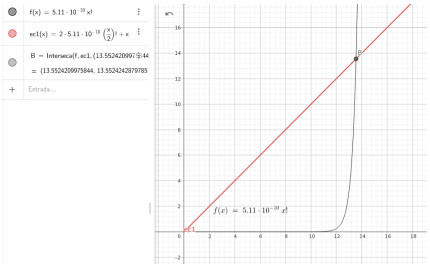
Dado la expresión de la eficiencia híbrida del algorítmo específico, $5.110 \cdot 10^{-10} \cdot fact(n)$, lo igualamos con la expresión recurrente:

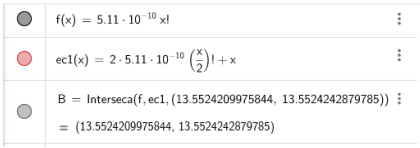
$$2 \cdot T(n/2) + n = 5.110 \cdot 10^{-10} \cdot fact(n)$$

Aplicamos $T(n/2) = 5.110 \cdot 10^{-10} \cdot fact(\frac{n}{2})$:

$$2 \cdot 5.110 \cdot 10^{-10} \cdot fact(\frac{n}{2}) + n = 5.110 \cdot 10^{-10} \cdot fact(n)$$

Vamos a resolver esta ecuación emprícamente:





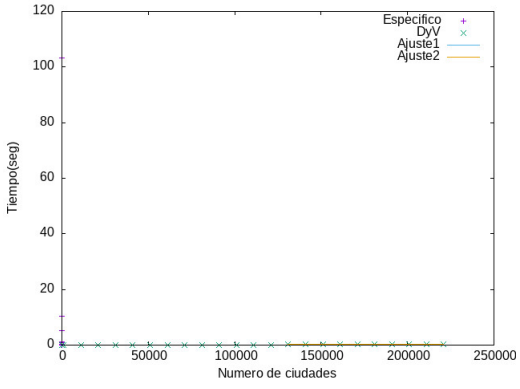
Luego el umbral óptimo es UMBRAL = 13.

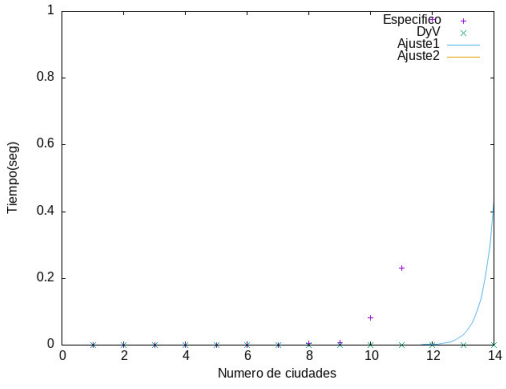
A vista del comportamiento del algoritmo con los umbrales del tanteo y empírico, podemos deducir que con este umbral la eficiencia no va a ser la óptima. En el tanteo obtuvimos que el tiempo de ejecución empeoraba considerablemente cuando aumentabamos el umbral, que se aprecia con UMBRAL = 2,3,4,5. Además, cuanto mayor era UMBRAL veiamos que entre los tiempos de ejecución se producen más fluctuaciones, así, vamos a descartar el caso de UMBRAL = 13.

Umbral empírico

Vamos a comparar las gráficas obtenidas en los estudios de eficiencia híbrida del algoritmo de DyV y el esfecífico.

Si comparamos las gráficas obtenemos:



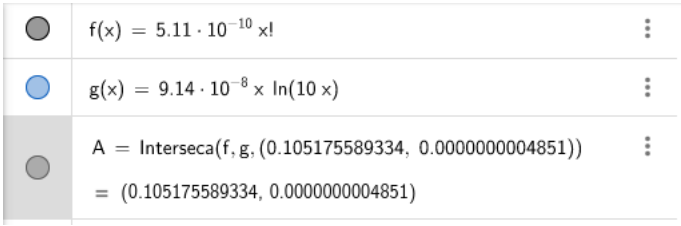


Donde:

- Específico: son los datos obtenidos por la ejecución del algoritmo específico.
- DyV: son los datos obtenidos por la ejecución del algoritmo DyV.
- Ajuste1: es la curva de regresión obtenida para el algoritmo específico $Ajuste1(x) = 5.11 \cdot 10^{-10} x!$.

- Ajuste2: es la curva de regresión obtenida para el algoritmo DyV $Ajuste2(x) = 9.14 \cdot 10^{-8} x \log(x)$.

En la primera gráfica no es posible ver las curvas de regresión porque el Ajuste1 crece demasiado rápido y el Ajuste2 apenas crece. En la segunda gráfica es posible ver la curva que ajusta el algoritmo específico porque hemos cambiado la escala, pero aún así, la segunda curva crece demasiado, haciendo casi imposible visualizarla.
Nota: El hecho de que podemos ver algunos datos de DyV es debido a que la curva de regresión se ajusta a todos los datos, y muy cerca de los entornos de algunos puntos (especialmente los primeros; donde hay bastante fluctuación de los tiempos de ejcución) la curva no se ajusta a la perfección.

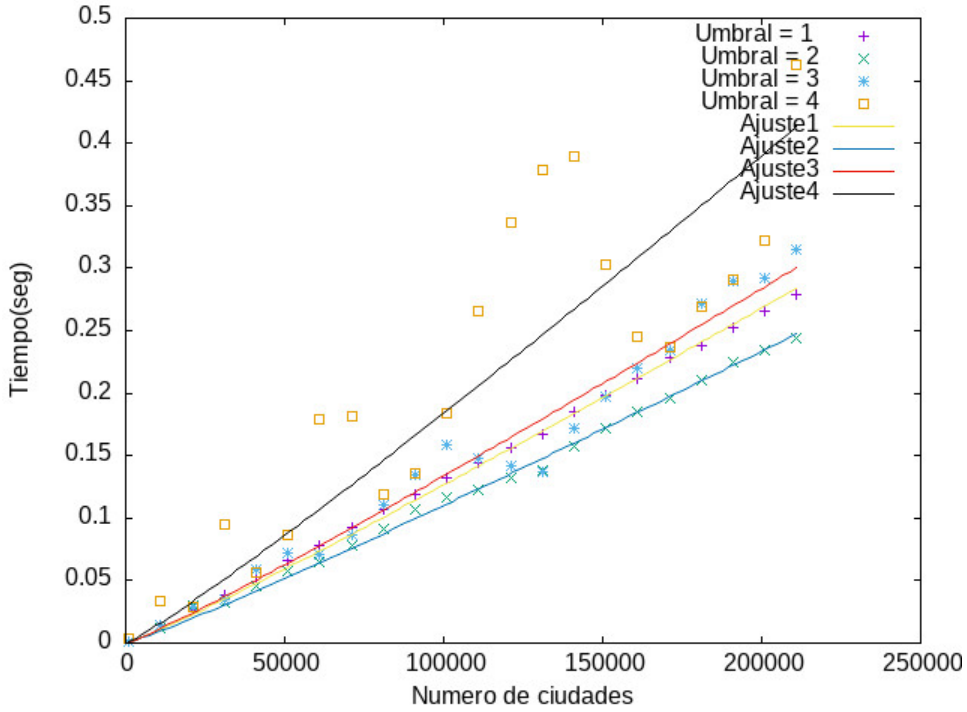


Como ambas curvas intersectan en $x = 0.10$ que haciendo la parte entera de esta obtenemos que $n = 0$.

Luego el umbral empírico es $n = 0$.

Grafica comparativa umbrales

Como para el caso de UMBRAL = 13 los resultados son poco satisfactorios, vamos a comparar única y exclusivamente los tiempos de ejecución para UMBRAL = 1,2,3,4. No se compara UMBRAL = 5 porque se producen grandes fluctuaciones y la eficiencia es peor; y para UMBRAL = 0 (UMBRAL EMPÍRICO), pasamos a estudiar UMBRAL = 1, porque 1 es el mínimo valor de umbral posible. Y obtenemos la siguiente gráfica:



Observamos que resulta que el mejor umbral es UMBRAL = 2. Notemos que el error relativo con respecto a los umbrales empírico y teórico son del 50 %, un error bastante considerable, pero también

tenemos que considerar que los valores de los umbrales obtenidos a comparar son muy pequeños, por tanto, las diferencias/errores relativas/os son muy sensibles a los cambios.

Ahora, también es lógico que los umbrales obtenidos tomen valores pequeños, puesto que el algoritmo de DyV es $O(n\log(n))$ mientras que el específico es $O(n!)$, es decir, la diferencia de eficiencia es considerable.
Pero claro, tampoco hay que perder de vista que ganamos eficiencia a costa de aumentar el error de los resultados obtenidos(los caminos escogidos por el algoritmo específico son exactos mientras que los de DyV tienen unos porcentajes de error bastante considerables). Así, que el UMBRAL sea 2, no implica que la mejor calidad de los resultados se obtenga con dicho valor.

6. Conclusiones

A lo largo de esta investigación, hemos experimentado y validado la aplicabilidad y profundidad del paradigma **divide y vencerás** en la solución de problemas computacionales complejos, tal como se aborda en los contenidos teóricos de nuestra formación.

La práctica ha permitido no solo aplicar de manera concreta los principios teóricos sino también apreciar las significativas diferencias en términos de eficiencia que esta estrategia algorítmica aporta. Durante el proceso, nos hemos encontrado con comportamientos inesperados que resaltan la importancia de considerar las características específicas de los datos de entrada y las peculiaridades de la implementación en los resultados de los algoritmos. Así como verificar que la estrategia *divide y vencerás* no es útil para cualquier caso ni tipo de algoritmo, sino para aquellos que son aptos para implementarse mediante esta técnica.

De manera particular, hemos observado la manifestación de conceptos teóricos clave en el desarrollo de nuestros algoritmos, tales como:

- La relevancia de seleccionar implementaciones de algoritmos que consideren las implicaciones teóricas de su diseño, evitando ineficiencias como las generadas por llamadas recursivas excesivas.
- La constatación de que las variaciones en el tiempo de ejecución al cambiar de plataforma (ya sea por hardware o software) afectan fundamentalmente en una constante de proporcionalidad, manteniendo la relación de eficiencia entre diferentes algoritmos.

Esta práctica ha sido una valiosa oportunidad para trasladar la teoría al ámbito práctico, enfrentando desafíos reales y desarrollando una comprensión más rica de los principios algorítmicos. Consideramos que este ejercicio ha sido un éxito, enriqueciendo nuestra preparación para abordar futuras problemáticas en nuestro recorrido académico y profesional, dotándonos de herramientas y perspectivas cruciales para reconocer casos en los que la estrategia **Divide y Vencerás** sea viable, y saber cómo llevarlos al ámbito real y práctico en base a los conocimientos teóricos.