Universidad de Granada

ALGORÍTMICA

Práctica 2

Práctica 2 - Algoritmos Divide y Vencerás

INTEGRANTES
GRUPO B3

Miguel Rodríguez Ayllón
Alberto Cámara Ortiz
Cristóbal Pérez Simón
José Vera Castillo
Pablo Rejón Camacho

ÍNDICE

Preámbulo

Nuestro proyecto cuenta con un makefile para que resulte más fácil a la hora de ejecutar los códigos proporcionados.
Cada ejercicio cuenta con una regla específica que muestra la ejecución de ambos algoritmos, tanto fuerza bruta como divide y vencerás y se muestra el resultado de la ejecución y el tiempo que tarda en ejecutarse (milisegundos o nanosegundos).

Si es necesario probar con unos valores a elegir por el profesor, recomendamos modificar la regla de ese ejercicio, o cambiar el argumento de la ejecución a mano. De todas formas, también podría ejecutarse a mano con los argumentos necesarios ya que no eliminamos los archivos .bin.

Todos los ejercicios cuentan con 1 argumento excepto el primer ejercicio, en el que se debe añadir tanto el número de candidatos como el número de votos (en orden).

1 - La mayoría absoluta

En este problema se nos pide diseñar un algoritmo en el que dado un vector de tamaño “n” con votos para unas elecciones, determinar si hay algún candidato con suficientes votos para ganar con mayoría absoluta (nº de votos que supere la mitad de los votos totales). También se utilizará la variable “m” que determina el número de candidatos diferentes

1.1. Diseño del método básico.

Para el diseño del método básico hemos creado una función que recorre el vector y con cada voto analiza el resto del vector para comprobar que se repite. Con ayuda de un map es que actualizamos el correspondiente número de votos para cada participante. Finalmente, comprobamos si hay mayoría utilizando la restricción del enunciado.

```
Función mayoriaFB(v, conteo):
    n = tamaño de v
    hayMayoria = falso

    // Recorrer el vector v y contar ocurrencias
     para cada i en [0, n):              si v[i] no está en conteo
entonces
     votos = 1
     para cada j en [i + 1, n):   si v[i] es igual a v[j] entonces
          votos++
      // Actualizar el conteo para el candidato correspondiente
      conteo[v[i]] = votos

    // Comprobar si hay mayoría
    para cada k en conteo:
     si k->segundo > n / 2 entonces
          hayMayoria = verdadero

    retornar hayMayoria
```

1.1.1. Estudio de la eficiencia teórica.

Este algoritmo se compone de tres bucles for en total, un bucle que recorre el vector de conteo de tamaño m y dos bucles for anidados que recorren n y n-1 veces respectivamente el vector de votos en el peor de los casos .Aplicando la regla del máximo, se puede asumir que esta función cuenta con un orden de eficiencia de **$O(n^2)$**.

1.2. Diseño del método Divide y Vencerás.

Para crear el método Divide y Vencerás que mejore el orden de eficiencia del algoritmo anterior se ha decidido crear dos funciones:

- La primera que se llama “mayoriaDyV” y es la encargada de llamar a “conteoRecursivo” y una vez se tenga relleno el vector de conteo, determinar si hay algún elemento de ese vector mayor o igual que n/2 lo que indicaría si hay algún candidato con mayoría absoluta.

```
Función mayoriaDyV(v, conteo):
    hayMayoria = falso
    n = tamaño de v
    // Calcular el conteo dentro del vector conteo
    conteoRecursivo(0, n - 1, v, conteo)

    // Recorrer conteo para ver si hay algún número > n/2
    para cada i en [0, tamaño de conteo):
        si conteo[i] > n / 2 entonces
            hayMayoria = verdadero
    retornar hayMayoria
```

- La segunda función, “conteoRecursivo” recibe como parámetros las posiciones entre las que se tiene que dividir el vector. La función se llama a sí misma de manera recursiva dividiendo el vector en dos subvectores de aproximadamente la mitad de componentes. Cuando la función detecta que el vector actual tiene una única componente se analiza a qué candidato pertenece dicho voto y se suma al conteo de dicho candidato.

```
Función conteoRecursivo(inicio, fin, v, conteo):
    si inicio es igual a fin entonces
        conteo[v[inicio]]++
    sino
        mitad = (inicio + fin) / 2
        conteoRecursivo(inicio, mitad, v, conteo)
        conteoRecursivo(mitad + 1, fin, v, conteo)
```

1.2.1. Estudio de la eficiencia teórica.

Vamos a empezar analizando “conteoRecursivo”. Como el vector se va dividiendo cada vez en 2 podemos determinar que aproximadamente se hacen log(n) particiones.Cada vez que se llega a un caso en el que el vector sólo tenga una componente, está se añade directamente al conteo con una operación O(1). Por eso, en esta función tendremos en el peor de los casos un orden de eficiencia al aplicar la recurrencia de **$O(n\log_2(n))$**.

En el caso de mayoriaDyV esta llama a conteoRecursivo de **$O(n\log_2(n))$** y aparte tiene un vector que se recorre “m” veces siendo “m” el tamaño del map utilizado para el conteo. Suponiendo que en la mayoría de casos “n” es mucho mayor que “m”, podemos determinar aplicando la regla del máximo que esta función tiene un orden de eficiencia de **$O(n\log_2(n))$**

Aplicando la fórmula maestra:

$$T\,(n)\;=\;k\cdot T\,(n/b)\;+\;g\,(n)$$

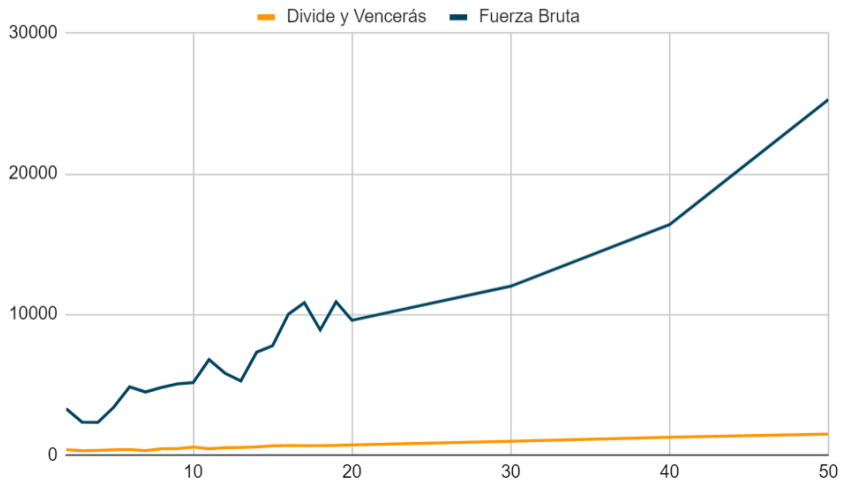
- k y b serían 2 porque cada vez que aplicamos la recursión obtenemos 2 subinstancias de aproximadamente n/2 componentes.
- Como k = 2 y b = 2^1:

$$T(n)\;=\;O(n\;*\;log(n))\textbf{.}$$

1.3. Resolución del problema del umbral.

Como podemos observar en la gráfica que se ha obtenido con los tiempos experimentales de ambos algoritmos, incluso partiendo desde el caso base con vectores de una componente, la versión Divide y Vencerás es siempre mejor que el algoritmo básico. Por tanto, no tendría sentido utilizar un método híbrido y es mejor optar en este caso por el método Divide y Vencerás. El número de candidatos en este caso es 5.

Tamaño de Caso (n)	Fuerza Bruta (ns)	Divide y Vencerás (ns)
1	3911	341
2	3297	380
3	2340	305
4	2322	328
5	3407	372
6	4845	390
7	4484	309
8	4807	443
9	5064	454
10	5150	562
11	6790	448
12	5824	519
13	5276	533
14	7319	577
15	7771	654
16	10030	672
17	10838	664
18	8904	666
19	10903	680
20	9588	717
30	12013	978
40	16404	1264
50	25295	1490



2 - Tuercas y tornillos

2.1. Diseño del método básico.

Para resolver este problema de una forma sencilla recorremos el vector de tornillos y para cada tornillo buscamos su tuerca correspondiente e intercambiamos las posiciones en el vector de tuercas.

```
procedimiento ordena(v1: vector de enteros, v2: vector de enteros)
    para i desde 0 hasta v1.size() - 1
        para j desde 0 hasta v2.size() - 1
            si v1[i] igual v2[j]
                intercambia v2[i] con v2[j]
            fin si
        fin para
    fin para
fin procedimiento
```

*Donde v1 es el vector de tornillos y v2 es el vector de tuercas

2.1.1. Estudio de la eficiencia teórica.

Este algoritmo cuenta con 2 bucles for anidados que recorren ambos vectores. Suponiendo que el tamaño de ambos vectores (siempre es el mismo ya que deben cuadrar los tornillos y las tuercas) es n, nos queda una eficiencia de **$O(n^2)$**.

2.2. Diseño del método Divide y Vencerás.

Para la resolución del problema usando el concepto de Divide y Vencerás hemos decidido usar quicksort para ordenar ambos vectores, editándolo un poco ya que no podemos comparar tornillos con tornillos ni tuercas con tuercas.

```
procedimiento quickSort(v1: vector de enteros, v2: vector de
enteros, bajo: entero, alto: entero)
    si bajo < alto
        declara pi: entero
        asigna pi a particion(v1, v2, bajo, alto, (alto + bajo) /
2)
        llama particion(v2, v1, bajo, alto, pi)
        llama quickSort(v1, v2, bajo, pi)
        llama quickSort(v1, v2, pi + 1, alto)
    fin si
fin procedimiento
```

```
procedimiento particion(v1: vector de enteros, v2: vector de
enteros, bajo: entero, alto: entero, posPivote: entero) retorna
entero
    declara pivote: entero
    declara i: entero
    declara posicion: entero

    asigna pivote a v2[posPivote]
    asigna i a (bajo - 1)

    para j desde bajo hasta alto - 1
        si v1[j] < pivote
            incrementa i en 1
            intercambia v1[i] con v1[j]
        fin si
    fin para

    para j desde bajo hasta alto - 1
        si pivote igual a v1[j]
            asigna posicion a j
        fin si
    fin para

    intercambia v1[i + 1] con v1[posicion]
    retorna (i + 1)
fin procedimiento
```

La solución consiste en elegir una tuerca como pivote para dejar los tornillos de menor tamaño a su izquierda y luego dejar el tornillo correspondiente a esa tuerca en la posición adecuada. Hacemos lo mismo con las tuercas pero eligiendo como pivote el tornillo con el mismo valor que la tuerca usada anteriormente. Repetimos este proceso llamando recursivamente con un tamaño menor.

2.2.1. Estudio de la eficiencia teórica.

Aplicando la fórmula:

$$T\,(n) \;=\; k \cdot T\,(n/b) \;+\; g\,(n)$$

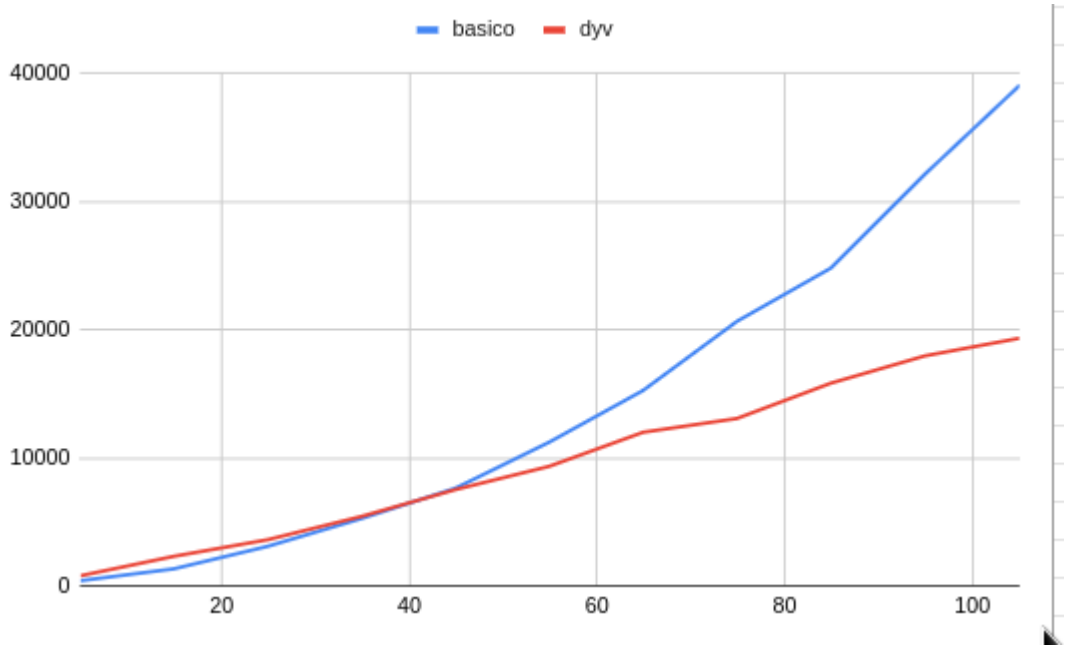
Teniendo en cuenta que tenemos dos subinstancias y que el tamaño de ellas es aproximadamente n/2 (teniendo en cuenta que el pivote sea un valor medio) y no nos cuesta nada descomponer en las 2 subinstancias.
Como k = 2, b = 2^1 entonces:

$$T(n) \;=\; O(n \;*\; log(n))\textbf{.}$$

2.3. Resolución del problema del umbral.

Para resolver el problema del umbral, comparamos los tiempos de ejecución del algoritmo «divide y vencerás» diseñado y el algoritmo básico y buscamos un cruce que nos permita determinar qué algoritmo debemos usar dependiendo del tamaño de caso que queramos ejecutar.

Tamaño de Caso (n)	Fuerza Bruta (ns)	Divide y Vencerás (ns)
5	336	716
15	1264	2241
25	3030	3541
35	5195	5353
45	7562	7449
55	11158	9272
65	15202	11924
75	20593	13005
85	24748	15760
95	32072	17892
105	39051	19336



Comparando las gráficas de tiempos de los dos métodos podemos observar que a partir de un tamaño de 45 aproximadamente el algoritmo de divide o vencerás es mejor que el básico

$$T(n) = \{ \quad n^2 \quad si\ n <= 45$$

$$n * log_2(n) \quad si\, n > 45$$

3 - Producto de tres elementos

3.1. Diseño del método básico.

Para llevar a cabo la resolución del problema usando un método básico diseñamos el siguiente pseudocódigo:

```
Introducimos un valor entero "v" por pantalla.

Para i ← 1 Hasta i*(i+1)*(i+2) < v Con Paso 1 Hacer
      prod = i * (i+1) * (i+2)
      Si v = prod Entonces
            Dejamos de iterar, hemos encontrado el elemento
      Si (v > prod) Entonces
            "v" no es producto de tres números consecutivos
Fin Para
```

3.1.1. Estudio de la eficiencia teórica.

Como vemos, el diseño del algoritmo básico no consta más que de un bucle for que itera mientras se cumpa la condición de que i*(i+1)*(i+2) sea menor o igual que v (aunque se trata de un bucle simple, evitamos iterar más veces de las necesarias).

Es por eso que el orden de eficiencia de este algoritmo es **O(∛n)** debido a la comprobación del bucle, puesto que dentro de sólo se están haciendo comprobaciones sencillas de orden de eficiencia 1.

3.2. Diseño del método Divide y Vencerás.

Para abordar el diseño del algoritmo de Divide y Vencerás, hemos optado por una versión que no utiliza recursividad para así evitar llamadas a la pila:

```
Función DyV(valor, Cini, Cfin):
    encontrado = -1
    fin = falso
    multi, mitad = entero largo

    mientras Cini <= Cfin y no fin:
        mitad = (Cini + Cfin) / 2
        multi = mitad * (mitad + 1) * (mitad + 2)
        fin = multi igual a valor

        si no fin entonces
            si multi > valor entonces
                Cfin = mitad - 1
            sino
                Cini = mitad + 1
```

```
        sino
            encontrado = mitad

    retornar encontrado
```

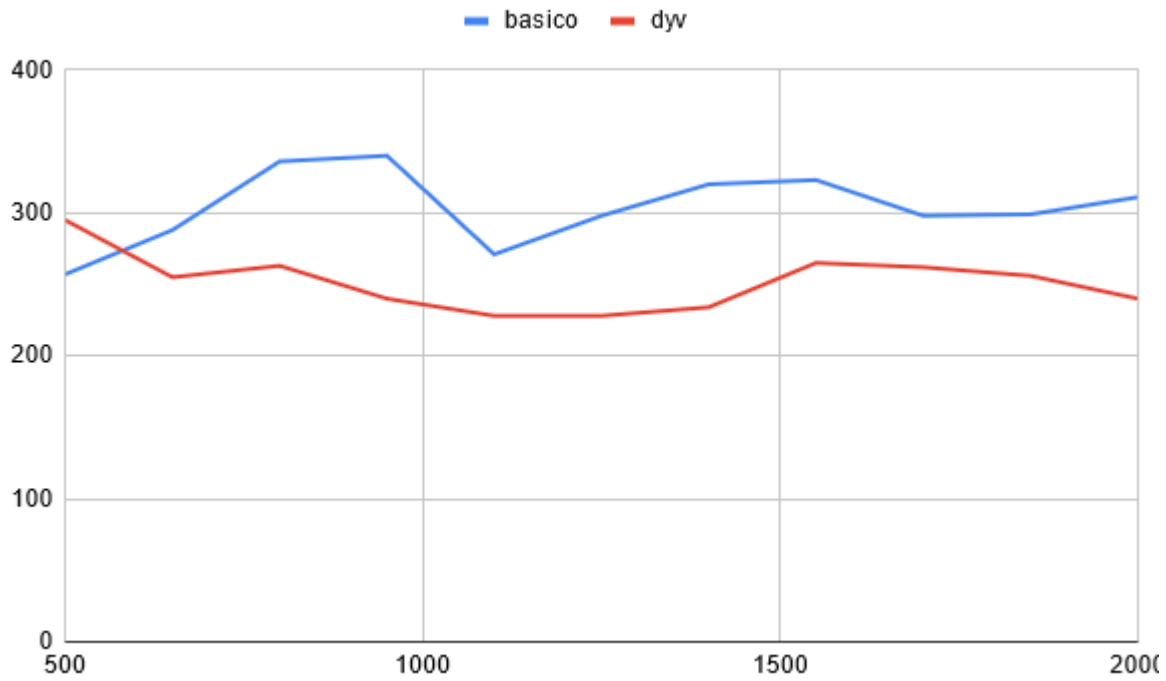
El funcionamiento del algoritmo consiste en acotar el número de búsquedas a la mitad del valor introducido, de modo que esa mitad se va haciendo cada vez más pequeña hasta encontrar el número introducido. Si el producto de los elementos correspondientes es mayor que el valor buscado, entonces volvemos a dividir el intervalo de búsqueda estableciendo los límites pertinentes en el caso del extremo final, hacemos lo mismo en caso de que sea menor ajustando el extremo inicial

3.2.1. Estudio de la eficiencia teórica.

Al no tratarse de una función recursiva, la eficiencia del algoritmo radica en cómo va reduciendo el rango de búsqueda a la mitad en cada iteración del bucle while. Es por eso que el orden de eficiencia es **$O(\log_2(n))$**

3.3. Resolución del problema del umbral.

Las implementaciones quedan descritas en los ficheros cpp entregados. Un ejemplo que ilustra el tiempo de ejecución de cada versión implementada es el siguiente, donde hemos cogido valores entre 500 y 2000 porque con un rango menor no se aprecian bien las diferencias:



Como vemos, para tamaños de caso que van a partir del 500, el algoritmo de Divide y Vencerás comienza a ser más eficiente.

4 - Eliminar elementos repetidos.

En este problema, se nos pide un algoritmo capaz de eliminar las posiciones repetidas de un vector. Para ello, vemos conveniente ordenarlo primero, de manera que las posiciones repetidas se encontrarán consecutivamente.

4.1. Diseño del método básico.

Para llevar a cabo la resolución del problema usando un método básico diseñamos el siguiente pseudocódigo:

```
Procedimiento BorraElementosRepetidos(V)
    Si tamaño de V es menor o igual a 1 entonces
        Retornar
    Fin Si

    it_1 = Iterador que apunta al principio de V
    it_2 = Iterador que apunta al elemento siguiente a it_1

    Mientras it_1 no sea el final de V hacer
        it_2 = it_1 + 1
        Mientras it_2 no sea el final de V hacer
            Si valor en it_1 es igual al valor en it_2 entonces
                it_2 = Borrar elemento apuntado por it_2 en V
            Sino
                it_2 = it_2 + 1
            Fin Si
        Fin Mientras
        it_1 = it_1 + 1
    Fin Mientras
Fin Procedimiento
```

4.1.1. Estudio de la eficiencia teórica.

Este algoritmo cuenta con dos bucles con tantas iteraciones como elementos cuenta el vector v. Como cada bucle tiene una eficiencia de **O(n)** y las demás operaciones son elementales con orden de eficiencia **O(1)**, concluimos que el orden de eficiencia de este algoritmo es **$O(n^2)$**.

4.2. Diseño del método Divide y Vencerás.

Para este diseño, se nos ocurrieron multitud de soluciones válidas que podían servir para realizar el algoritmo. Sin embargo, hemos optado por un método Divide y Vencerás que, además de eliminar los elementos repetidos del vector, ordena el vector al mismo tiempo.

Esta solución va dividiendo el vector en subvectores de mismo tamaño y, en cada subvector, se comprobará inicialmente el primer valor y se irá añadiendo en un vector auxiliar el menor de ellos, incrementando el contador (i o j) que contenga el elemento del subvector almacenado.

Contamos con dos funciones, una función recursiva Divide y Vencerás con dos instancias que se llaman con un tamaño de vector n/2 y una llamada a la función “joinVector”, que realiza las comparaciones entre elementos y los añade de forma ordenada sin incluir elementos repetidos.

```
Función DyV(V):
    tam_vector = Tamaño de V

    Si tam_vector es igual a 1 entonces:
        Devolver V

    Sino:
        mitad = tam_vector / 2
        vIzquierda = Subvector de V desde el principio hasta (final - mitad)
        vDerecha = Subvector de V desde (final - mitad) hasta el final

        vIzquierda = DyV(vIzquierda)
        vDerecha = DyV(vDerecha)

        V = joinVector(vIzquierda, vDerecha)

    Devolver V

Función joinVector(VIzquierda, VDerecha):
    i = 0
    j = 0
    fin = Falso
    VFinal = Nuevo vector

    Mientras no fin hacer:
        Si VIzquierda[i] < VDerecha[j] entonces:
            Agregar VIzquierda[i] a VFinal
            Si i es igual a (tamaño de VIzquierda - 1) entonces:
                fin = Verdadero
                Para cada k desde j hasta tamaño de VDerecha - 1 hacer:
                    Agregar VDerecha[k] a VFinal
            Sino:
                Incrementar i en 1
        Sino, si VIzquierda[i] > VDerecha[j] entonces:
            Agregar VDerecha[j] a VFinal
            Si j es igual a (tamaño de VDerecha - 1) entonces:
                fin = Verdadero
                Para cada k desde i hasta tamaño de VIzquierda - 1 hacer:
                    Agregar VIzquierda[k] a VFinal
            Sino:
                Incrementar j en 1
        Sino:
            Agregar VIzquierda[i] a VFinal
```

```
            Si i es igual a (tamaño de VIzquierda - 1) entonces:
                fin = Verdadero
                Para cada k desde j+1 hasta tamaño de VDerecha - 1 hacer:
                    Agregar VDerecha[k] a VFinal
            Sino, si j es igual a (tamaño de VDerecha - 1) entonces:
                fin = Verdadero
                Para cada k desde i+1 hasta tamaño de VIzquierda - 1 hacer:
                    Agregar VIzquierda[k] a VFinal
            Sino:
                Incrementar i en 1
                Incrementar j en 1

    Devolver VFinal
```

4.2.1. Estudio de la eficiencia teórica.

Para realizar el análisis, debemos tener en cuenta que vamos dividiendo el vector dependiendo de su tamaño n hasta que el tamaño del subvector sea 1.

Por lo tanto, nos encontramos un caso base en el que si n=1, se realiza una operación elemental con orden de eficiencia **O(1)**.

Sin embargo, el caso general consiste en tener un vector (subvector) con un tamaño mayor que uno, que se va dividiendo en **dos subvectores de igual tamaño n/2** y, finalmente, unir las soluciones sin elementos repetidos con un orden de eficiencia **O(n)**.

$$T(n) = \begin{cases} T(1) & si\ n = 1 \\ 2 \cdot T(n/2) + n & si\ n > 1 \end{cases}$$

Teniendo en cuenta esta ecuación de recurrencias, vamos a analizar el segundo caso. Para ello, será necesario realizar un cambio de variable, $n = 2^k$ y $k = log_2(n)$:

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

$$Raíces \ -> (x-2)(x-2) = 0$$

$$T(k) = c_1 2^k + c_2 k 2^k$$

Deshacemos el cambio de variable y llegamos a la conclusión.

$$T(n) = c_1 n + c_2 n * log_2(n)$$

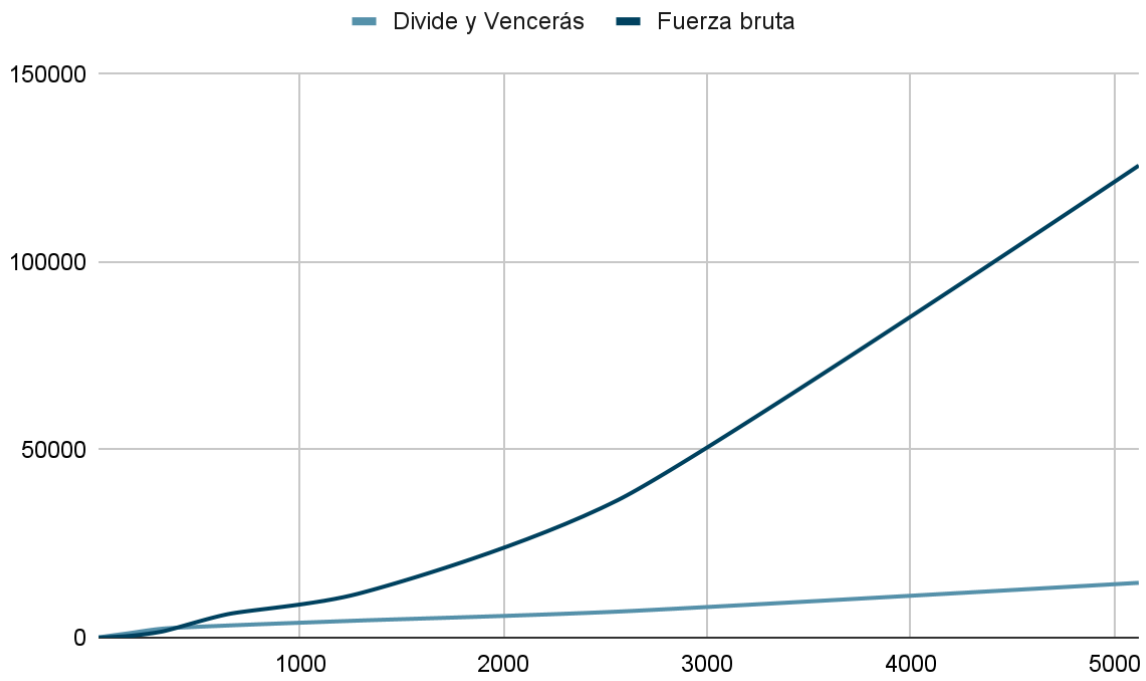
Es por ello que el orden de eficiencia del algoritmo es de:

$$O(n \times log_2(n))\textbf{.}$$

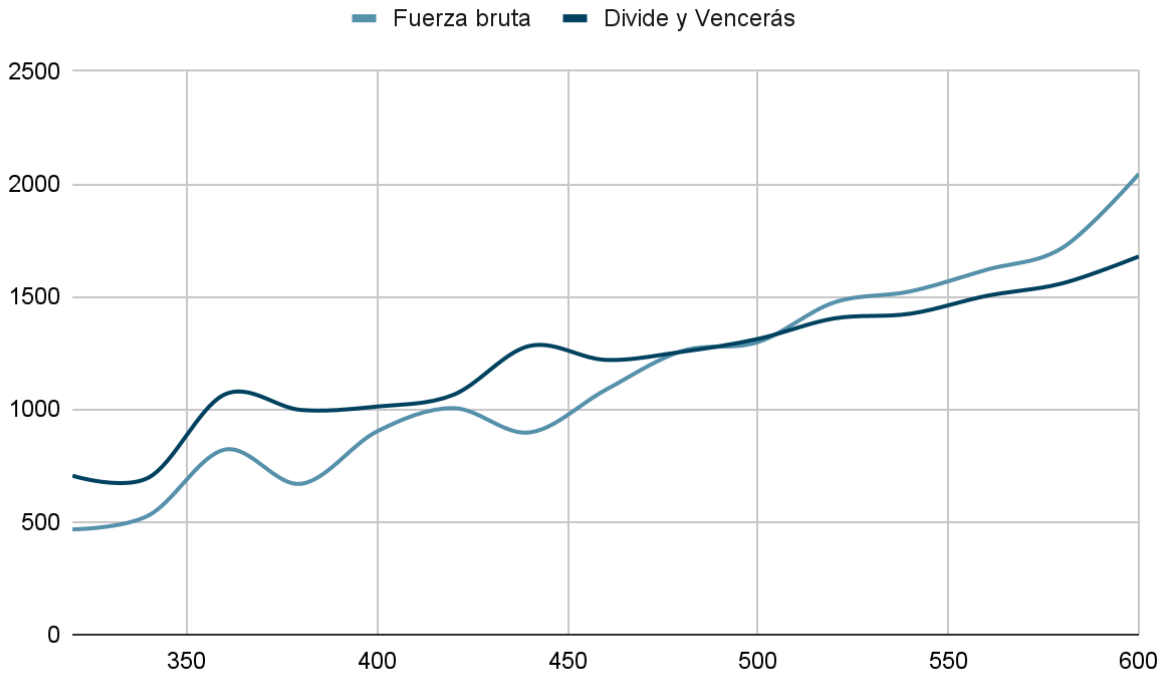
4.3 Resolución del problema del umbral.

Para resolver el problema del umbral, buscamos el valor en el que se “equilibran” las gráficas y encontramos un cruce que nos permita determinar qué algoritmo debemos usar dependiendo del tamaño de caso que queramos ejecutar.

Tamaño de Caso (n)	Fuerza Bruta (ms)	Divide y Vencerás (ms)
10	12	72
20	13	137
40	39	285
80	171	589
160	442	1168
320	1589	2363
640	6188	3205
1280	11610	4516
2560	36583	6917
5120	125686	14584



Para resolver el problema del umbral, hemos reducido el tamaño de caso y hemos obtenido una gráfica con los valores acotados, dando así un umbral más preciso. El resultado de éstas se realiza en un tiempo de ejecución menor que los vistos en la gráfica anterior, tanto un algoritmo como el otro. Esto se debe al valor de K promedio (0,005728941534).



Tras el análisis de ambas gráficas, concluimos que el umbral queda fijado en un tamaño de caso “n” de en torno a **500** (aproximado). Por lo tanto, para obtener un algoritmo eficiente en el que se combinen ambas soluciones:

$$T(n)\begin{cases} n^2 & si\ n \leq 500 \\ n \times log_2 n & si\ n > 500 \end{cases}$$

5 - Organización del calendario de un campeonato

5.1. Diseño del método básico.

Un método sencillo para resolver este problema sería tener una matriz booleana auxiliar la cual muestra para una fila y columna si esos dos participantes se han enfrentado ya, y un vector de booleanos para cada jornada que indica si el participante ya ha jugado esa jornada.

Para cada jornada si ese participante no ha jugado le buscamos un rival con el que no haya jugado ya y lo asignamos en la tabla del campeonato.

Hay una restricción de tamaño de caso , el cuál tiene que ser una potencia de dos.

```
/ Función para generar una tabla básica de enfrentamientos
Procedimiento basico(t: matriz de enteros)
     matriz enfrentados(t.size(), vector de booleanos(t.size(),
falso))

     Para i desde 0 hasta t.size() - 1 hacer
          Para j desde 0 hasta enfrentados.size() - 1 hacer
               Si i == j entonces
                    enfrentados[i][j] ← verdadero
               Fin Si
          Fin Para
     Fin Para


     Para jornada desde 1 hasta t.size() - 1 hacer
     vector enfrentados_jornadas(t.size(), falso)
     Para equipo desde 1 hasta t.size() hacer
          Para rival desde 1 hasta t.size() hacer
               Si no enfrentados[equipo-1][rival-1] entonces
                    Si no enfrentados_jornadas[equipo-1] y no
enfrentados_jornadas[rival-1] entonces
                    t[equipo-1][jornada-1] ← rival
                    t[rival-1][jornada-1] ← equipo
                    enfrentados_jornadas[equipo-1] ← verdadero
                    enfrentados_jornadas[rival-1] ← verdadero
                    enfrentados[equipo-1][rival-1] ← verdadero
                    enfrentados[rival-1][equipo-1] ← verdadero
                    Fin Si
               Fin Si
          Fin Para
     Fin Para
     Fin Para
Fin Procedimiento
```

5.1.1. Estudio de la eficiencia teórica.

Como podemos observar la función se compone de 3 bucles for que van desde 1 hasta n aproximadamente (siendo n el tamaño de la matriz) así que nos queda una eficiencia de **$O(n^3)$**

5.2. Diseño del método Divide y Vencerás.

Para crear el método Divide y Vencerás que mejore el orden de eficiencia del algoritmo anterior se ha decidido crear dos funciones:

- La primera es completarTabla, la cuál asigna valores a la matriz según los rangos dados y un valor inicial.
- La segunda es formarTabla , divide recursivamente un rango dado en dos mitades y llama a completarTabla para llenar la tabla con valores específicos según las reglas definidas.

```
Procedimiento completarTabla(t: matriz de enteros, eqInf: entero,
eqSup: entero, diaInf: entero, diaSup: entero, eqInicial: entero)
     Para j desde diaInf hasta diaSup hacer
          t[eqInf-1][j-1] ← eqInicial + j - diaInf
     Fin Para
     Para i desde eqInf + 1 hasta eqSup hacer
          t[i-1][diaInf-1] ← t[i - 1][diaSup]
          Para j desde diaInf + 1 hasta diaSup hacer
               t[i-1][j-1] ← t[i - 2][j - 2]
          Fin Para
     Fin Para
Fin Procedimiento


Procedimiento formarTabla(t: matriz de enteros, primero: entero,
ultimo: entero)
     entero medio
     Si ultimo - primero = 1 entonces
          t[primero-1][0] ← ultimo
          t[ultimo-1][0] ← primero
     Sino
     medio ← (primero + ultimo) / 2
     formarTabla(t, primero, medio)
     formarTabla(t, medio + 1, ultimo)subsolución con los de la
segunda
     completarTabla(t, primero, medio, medio - primero + 1, ultimo
- primero, medio + 1)
     completarTabla(t, medio + 1, ultimo, medio - primero + 1,
ultimo - primero, primero)
     Fin Si
Fin Procedimiento
```

5.2.1. Estudio de la eficiencia teórica.

Aplicando la fórmula:

$$T\,(n) \;=\; k \cdot T\,(n/b) \;+\; g\,(n)$$

Donde:

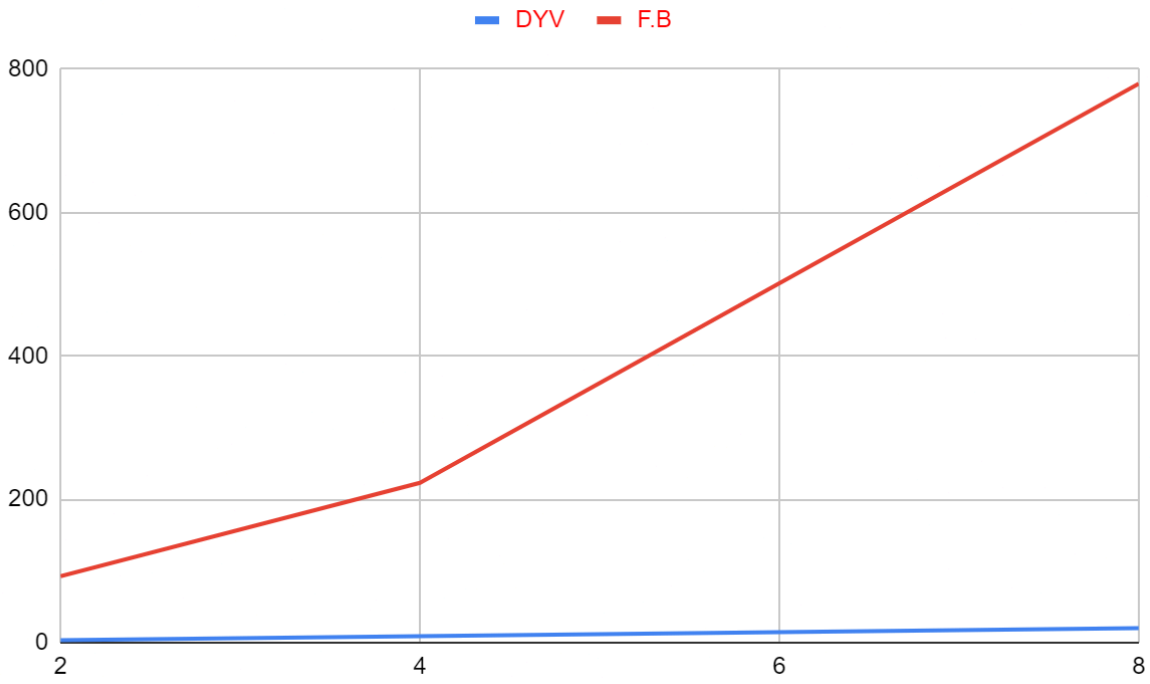
- (T(n)) es el tiempo requerido para resolver un problema de tamaño (n).
- (k) es el número de subproblemas generados en cada nivel de recursión el cuál es 2.
- (n/b) es el tamaño de cada subproblema , en este caso b = 2^1.
- (g(n)) representa el tiempo adicional necesario para combinar las soluciones de los subproblemas y realizar otras operaciones el cual es n

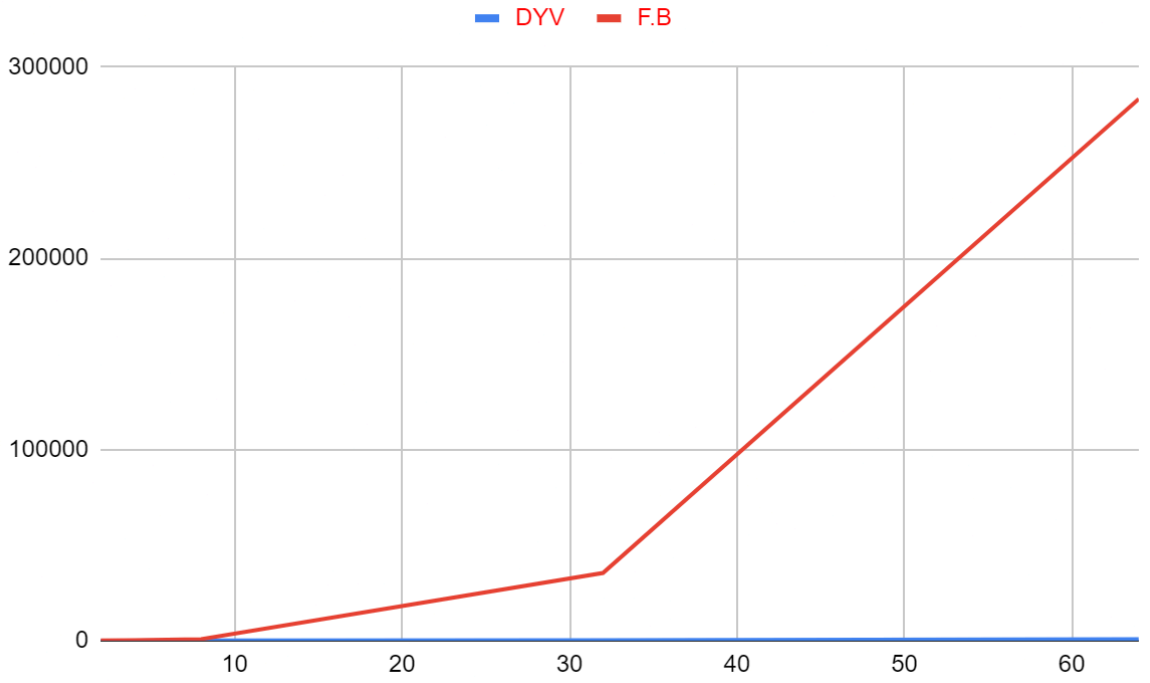
Por lo tanto, como tanto k como b son 2 :

$$.T\,(n) \;=\; O(n^2)$$

5.3 Resolución del problema del umbral.

Tamaño de Caso (n)	Fuerza Bruta (ns)	Divide y Vencerás (ns)
2	336	716
4	1264	2241
8	3030	3541
16	5195	5353
32	7562	7449
64	11158	9272
128	15202	11924
256	20593	13005
512	24748	15760





Como podemos observar en ambas gráficas, el método divide y vencerás es mejor en todos los casos.

6 - Conclusión

Como se ha podido observar a lo largo de la memoria, está claro que la resolución mediante la técnica Divide y Vencerás puede ayudarnos a resolver problemas que a priori presentan una gran complejidad mediante la división en instancias más simples.
Sin embargo, es necesario tener en cuenta que algunas veces, los algoritmos que resuelven estos mismos problemas mediante fuerza bruta pueden sernos más útiles en los casos en los que los volúmenes de datos sean muy pequeños o solo haya que realizar operaciones elementales, por eso la elección de uno u otro dependerá de nuestras necesidades en cuanto a eficiencia se refiere y sobre todo, tener presente que muchas veces la técnica Divide y Vencerás implica hacer uso de llamadas recursivas que involucran el uso de la pila, este factor también es relevante a la hora de hacer problemas en equipos cuya memoria no es muy elevada.