

# PRÁCTICA 2:

## ALGORITMOS DIVIDE Y VENCERÁS



### ***Integrantes del grupo:***

*Ana López Mohedano*  
*Adrián Anguita Muñoz*  
*Marina Jun Carranza Sánchez*  
*Pedro Antonio Mayorgas Parejo*  
*Rafael Jiménez Márquez*

# ÍNDICE

<b>0. Aclaraciones previas.....</b>	<b>3</b>
<b>1. La mayoría absoluta.....</b>	<b>4</b>
a) Algoritmo básico.....	4
Implementación.....	4
Eficiencia teórica.....	4
Eficiencia práctica.....	5
b) Algoritmo Divide y Vencerás.....	5
Implementación.....	5
Eficiencia teórica.....	7
c) Comparación gráfica y umbral.....	8
<b>2. Tuercas y tornillos.....</b>	<b>9</b>
a) Algoritmo básico.....	9
Implementación.....	9
Eficiencia práctica.....	9
b) Algoritmo Divide y Vencerás.....	10
Implementación.....	10
Eficiencia teórica.....	11
Eficiencia práctica.....	11
c) Comparación gráfica y umbral.....	12
<b>3. Producto de tres elementos.....</b>	<b>13</b>
a) Algoritmo básico.....	13
Implementación.....	13
Eficiencia teórica.....	13
Eficiencia práctica.....	13
b) Algoritmo Divide y Vencerás.....	14
Implementación.....	14
Eficiencia teórica.....	14
Eficiencia práctica.....	15
c) Comparación gráfica y umbral.....	15
<b>4. Eliminar elementos repetidos.....</b>	<b>16</b>
a) Algoritmo básico.....	16
Implementación.....	16
Eficiencia teórica.....	16
Eficiencia práctica.....	17
b) Algoritmo Divide y Vencerás.....	17
Implementación.....	17
Eficiencia teórica.....	18
Eficiencia práctica.....	18
c) Comparación gráfica y umbral.....	18
<b>5. Organización del calendario de un campeonato.....</b>	<b>19</b>
a) Algoritmo básico.....	19
Implementación.....	19

Eficiencia teórica.....	20
Eficiencia práctica.....	20
b) Algoritmo Divide y Vencerás.....	20
Implementación.....	20
Eficiencia teórica.....	22
Eficiencia práctica.....	22
c) Comparación gráfica y umbral.....	22

---

## 0. Aclaraciones previas

Para la resolución de problemas propuestos, se va a llevar a cabo el análisis de los siguientes algoritmos, cada cual se dividirá en tres apartados:

- El **algoritmo básico** o de fuerza bruta, del que se estudiará su diseño y/o pseudocódigo, su eficiencia teórica y la práctica (pruebas de ejecución con make).
- La **versión Divide y Vencerás** del algoritmo anterior, con el análisis de los mismos apartados para facilitar la comparación entre ellos.
- La **comparación gráfica y cálculo del umbral** experimental.

### Requisitos de un problema para aplicar Divide y Vencerás:

- El problema inicial debe poder ser divisible en subproblemas que tengan aproximadamente el mismo tamaño, independientes entre sí, que sean de la misma naturaleza del inicial y puedan resolverse por separado.
- Las soluciones de los subproblemas deben poder combinarse entre sí para poder dar lugar a la solución del problema inicial.
- Debe existir un caso base en el que el problema sea ya indivisible o, en su defecto, un algoritmo básico que pueda resolverlo

### Cálculo del umbral de forma experimental:

Para ello, se estudia la eficiencia práctica de los algoritmos y se muestran los tiempos de ejecución de la versión básica y divide y vencerás en una gráfica de líneas. El umbral es aquel valor a partir del cual el básico comienza a dominar asintóticamente al divide y vencerás (pierde eficiencia a largo plazo).

### Pautas para probar el código con makefile:

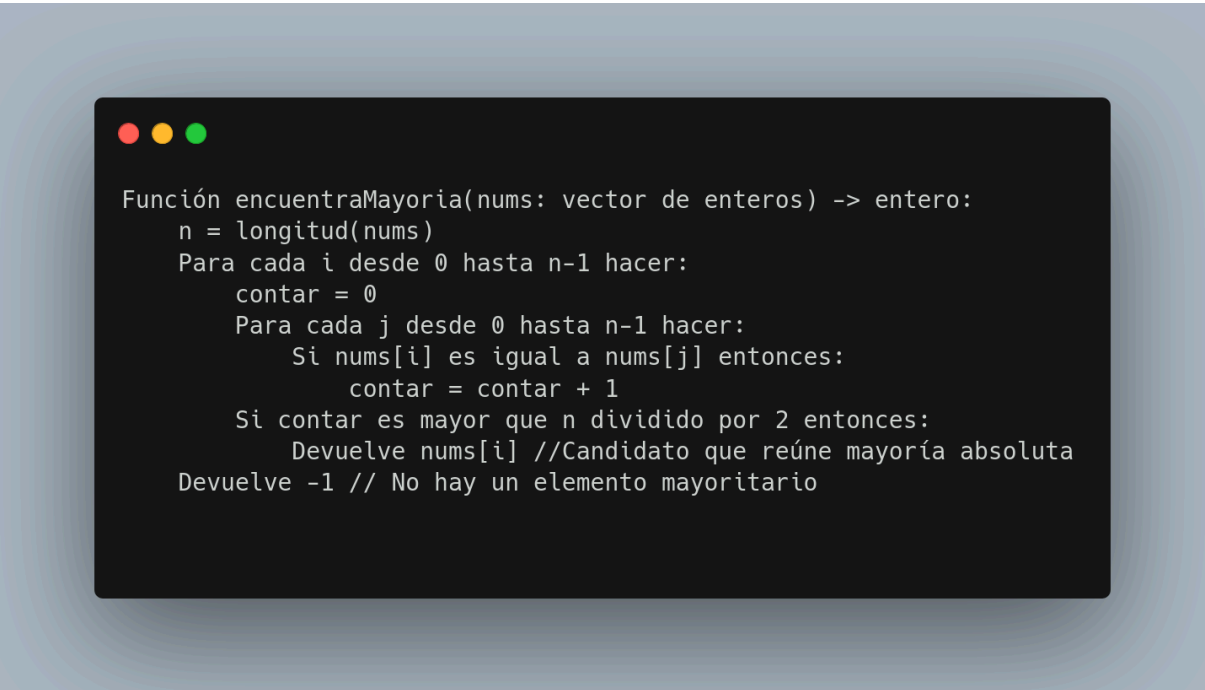
Se incluye una tarea “all” para compilar y probar todos los algoritmos, y también una orden de make de compilación y de ejecución para cada uno de ellos, por ejemplo: “make pX\_compile” y “make pX\_test”, siendo X el número del problema.

# 1. La mayoría absoluta

## a) Algoritmo básico

### Implementación

Para realizar el algoritmo básico hemos realizado el siguiente algoritmo en el que mediante una función a la que le pasamos un vector de enteros en el que cada posición contiene el voto a uno de los candidatos y comprobamos el número de veces que aparece un voto a un candidato. Si este aparece un número de veces mayor que el total de votos entre 2, habrá un candidato con mayoría absoluta. El pseudocódigo sería el siguiente:



```
Función encuentraMayoria(nums: vector de enteros) -> entero:
  n = longitud(nums)
  Para cada i desde 0 hasta n-1 hacer:
    contar = 0
    Para cada j desde 0 hasta n-1 hacer:
      Si nums[i] es igual a nums[j] entonces:
        contar = contar + 1
    Si contar es mayor que n dividido por 2 entonces:
      Devuelve nums[i] //Candidato que reúne mayoría absoluta
  Devuelve -1 // No hay un elemento mayoritario
```

### Eficiencia teórica

Analizando la eficiencia teórica de este algoritmo, podemos observar que el algoritmo tiene dos bucles anidados:

El bucle externo recorre el vector de votos desde  $n=0$  hasta  $n-1$ , con lo que tenemos una complejidad de  $O(n)$ , donde  $n$  es el tamaño del vector. El bucle interno también recorre el vector una vez por cada iteración del bucle externo, con una complejidad nuevamente de  $O(n)$  en cada iteración.

Dado que las inicializaciones, las comparaciones, el aumento del contador y el bloque if tienen una complejidad de  $O(1)$  y que ambos bucles están anidados, aplicando la regla del máximo en los bucles for obtenemos que la complejidad total del algoritmo es  $O(n^2)$ .

Podemos comprobar que la eficiencia resultante no es óptima y puede ser ineficiente para conjuntos de datos grandes, ya que el algoritmo es cuadrático. En términos prácticos,

igualmente este algoritmo podría ser inaceptablemente lento para conjuntos de datos grandes. Este problema es el que trataremos de solucionar con la aplicación de la técnica Divide y Vencerás.

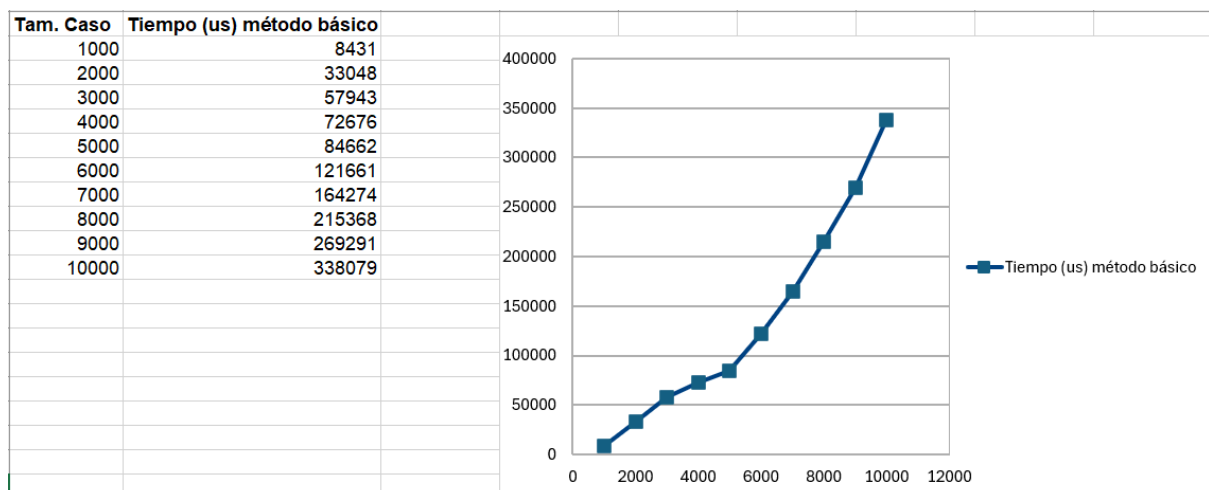
## Eficiencia práctica

Una vez implementado el algoritmo, veamos los resultados de la eficiencia práctica que obtenemos tras la ejecución del algoritmo con varios tamaños de  $n$ .

```

rafael@LAPTOP-AUQNSK3C:/mnt/c/Users/rafael/OneDrive/Escritorio/UGR/8º cuatrimestre/ALG/P2$ g++ -o mayoriaAbsoluta mayoriaAbsoluta.cpp
rafael@LAPTOP-AUQNSK3C:/mnt/c/Users/rafael/OneDrive/Escritorio/UGR/8º cuatrimestre/ALG/P2$ ./mayoriaAbsoluta 12345 1000 2000 3000 4000 5000 6000 7000 8000 9000 10000
n: 1000      Tiempo (us): 8431
n: 2000      Tiempo (us): 33048
n: 3000      Tiempo (us): 57943
n: 4000      Tiempo (us): 72676
n: 5000      Tiempo (us): 84662
n: 6000      Tiempo (us): 121661
n: 7000      Tiempo (us): 164274
n: 8000      Tiempo (us): 215368
n: 9000      Tiempo (us): 269291
n: 10000     Tiempo (us): 338079
  
```

Si representamos los datos obtenidos mediante un gráfico, obtenemos lo siguiente:



## b) Algoritmo Divide y Vencerás

### Implementación

Vamos a usar la estrategia de Divide y Vencerás implementando una función que recibe como parámetros un vector de enteros y dos índices: izquierda (abajo) y derecha (arriba) del rango en el que se busca el candidato con mayoría absoluta.

Si el rango es de un solo elemento, simplemente devuelve ese elemento, ya que es el único posible con mayoría absoluta (caso base). Si no, se calcula el punto medio del rango y divide el problema en dos subproblemas, llamando recursivamente a la función para los subrangos izquierdo y derecho. Si el elemento mayoritario en ambos subrangos es el mismo, ese elemento es el elemento mayoritario en todo el rango, y se devuelve. Si los elementos mayoritarios en los subrangos son diferentes, se cuentan las frecuencias de ambos elementos mayoritarios en todo el rango utilizando la función `cuentaFrecuencia` y se

compara la frecuencia de ambos elementos mayoritarios con la mitad del tamaño del rango. Si alguno de los elementos aparece más de la mitad del tamaño del rango, entonces ese elemento es el que tiene mayoría absoluta y se devuelve. En caso contrario, se devuelve -1.

El pseudocódigo del algoritmo es el siguiente:

```
Función encuentraMayoriaDyV(nums: vector de int, abajo: entero, arriba: entero) ->
int:
    Si abajo es igual a arriba entonces:
        Devuelve nums[abajo]

    medio = abajo + (arriba - abajo) / 2

    mayoriaIzquierda = encuentraMayoriaDyV(nums, abajo, medio)
    mayoriaDerecha = encuentraMayoriaDyV(nums, medio + 1, arriba)

    Si mayoriaIzquierda es igual a mayoriaDerecha entonces:
        Devuelve mayoriaIzquierda

    cuentaIzquierda = cuentaFrecuencia(nums, abajo, arriba, mayoriaIzquierda)
    cuentaDerecha = cuentaFrecuencia(nums, abajo, arriba, mayoriaDerecha)

    Si cuentaIzquierda es mayor que (arriba - abajo + 1) / 2 entonces:
        Devuelve mayoriaIzquierda
    Sino si cuentaDerecha es mayor que (arriba - abajo + 1) / 2 entonces:
        Devuelve mayoriaDerecha
    Sino:
        Devuelve -1
```

El pseudocódigo de la función que cuenta la frecuencia es el siguiente:

```
Función cuentaFrecuencia(nums: vector de enteros, abajo: entero, arriba: entero,
valor: entero) -> entero:
    contar = 0
    Para cada i desde abajo hasta arriba hacer:
        Si nums[i] es igual a valor entonces:
            contar = contar + 1
    Devuelve contar
```

## Eficiencia teórica

Vamos a analizar la eficiencia teórica del algoritmo, empezando por la función encuentraMayoríaDyV y también analizaremos la eficiencia de la función que cuenta la frecuencia.

El algoritmo realiza para cada llamada recursiva, una división del problema original en dos subproblemas de tamaño la mitad del tamaño original del vector. El algoritmo va realizando llamadas recursivas hasta que el rango del vector se reduce a un solo elemento. Por último, combinamos las soluciones para encontrar el elemento con mayoría en el rango original.

Dado que en cada nivel de recursión se reduce el tamaño del problema a la mitad, la complejidad del algoritmo puede expresarse mediante la recurrencia  $T(n) = 2T(n/2) + O(n)$ , donde  $n$  es el tamaño del vector de entrada, la parte  $2T(n/2)$  se refiere al tiempo necesario para resolver los dos subproblemas generados por la división del rango original en dos mitades. Como el tamaño de cada subproblema es la mitad del tamaño del problema original y la parte  $O(n)$  se refiere al tiempo necesario para contar las frecuencias de los elementos mayoritarios en los subrangos, lo cual se hace llamando a la función cuentaFrecuencia. Esta operación se realiza en tiempo lineal en función del tamaño del rango ( $O(n)$ ).

Usando la fórmula maestra  $T(n) = k \cdot T(n/b) + g(n)$ , siendo  $g(n)$  de la forma  $n^a$  tenemos que para nuestra ecuación de recurrencia:  $k = 2$ ;  $b = 2$ ;  $a = 1$ ;

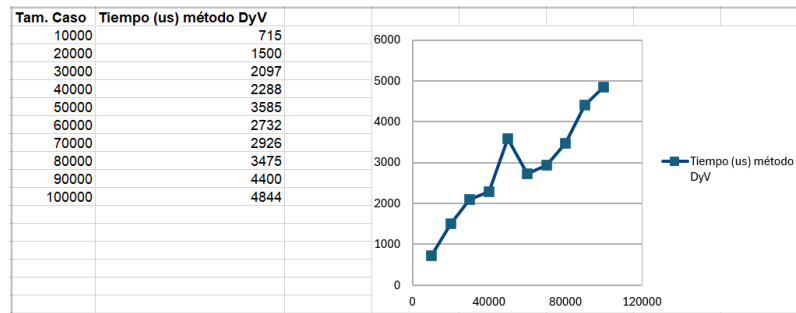
Aplicando las reglas de la fórmula maestra obtenemos que:  $2 = 2^1$ , es decir, que  $k = b^a$  y, por tanto, la complejidad del algoritmo tras aplicar esta regla es de  $O(n \log n)$ . Esto indica que el algoritmo es mucho más eficiente en comparación con la implementación básica anterior (bucle doble), especialmente para conjuntos de datos grandes, ya que su tiempo de ejecución crece de manera más lenta con el tamaño del vector de entrada.

## Eficiencia práctica

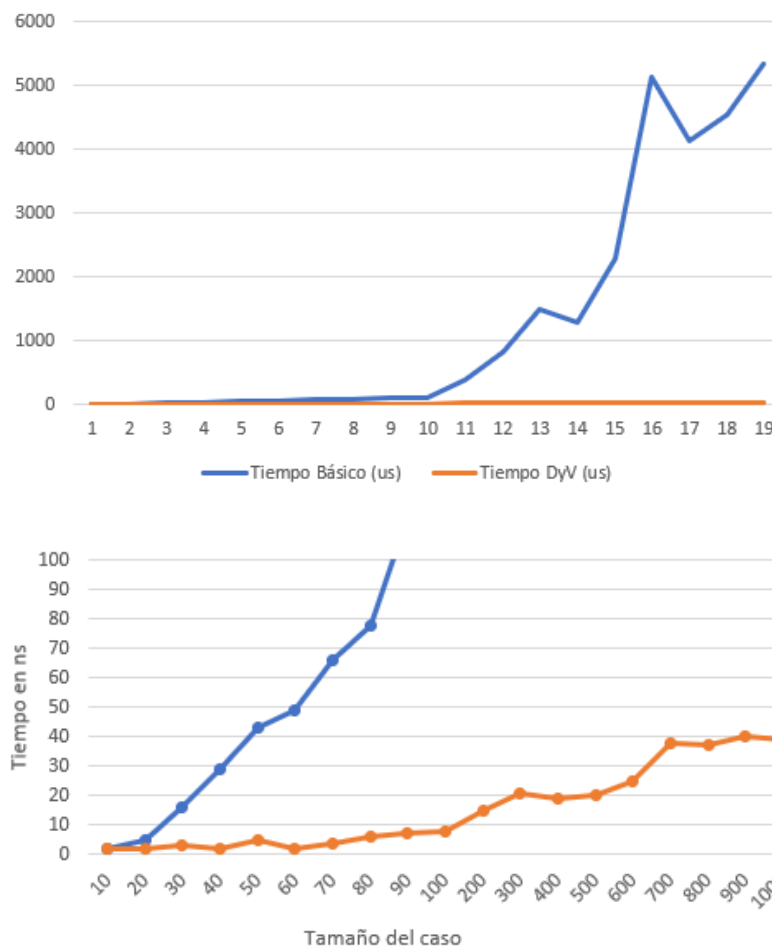
Una vez implementado el algoritmo, veamos los resultados obtenidos tras la ejecución del algoritmo para extraer la eficiencia práctica. Los resultados obtenidos son los siguientes:

```
rafajm02@LAPTOP-AUQNSK3C: /mnt/c/Users/rafaj/OneDrive/Escritorio/UGR/8º cuatrimestre/ALG/P2$ ./mayoriaAbsolutaDyV 12345 10000 20000 30000 40000 50000 60000 70000 80000 90000 100000
n: 10000      Tiempo (us): 715
n: 20000      Tiempo (us): 1500
n: 30000      Tiempo (us): 2097
n: 40000      Tiempo (us): 2288
n: 50000      Tiempo (us): 3585
n: 60000      Tiempo (us): 2732
n: 70000      Tiempo (us): 2926
n: 80000      Tiempo (us): 3475
n: 90000      Tiempo (us): 4400
n: 100000     Tiempo (us): 4844
```

Si representamos los datos obtenidos nuevamente mediante un gráfico, obtenemos lo siguiente:



### c) Comparación gráfica y umbral



Como podemos observar el umbral podría encontrarse en el intervalo **(2,3)** lo que implica que el algoritmo básico es bastante peor que el divide y vencerás, ya que incluso para tamaños de caso pequeños tienen un tiempo muy similar. Esto es debido a que la eficiencia del básico es  $O(n^2)$  por lo que el tiempo va aumentando muy rápidamente, (primera gráfica), por lo que el divide y vencerás ( $O(n \cdot \log_2(n))$ ) obtiene menor tiempo (es más eficiente que el básico). A su vez, también existen otros algoritmos, los cuales resuelven este problema y que mejoran al divide y vencerás, y tienen una eficiencia  $O(n)$ . Este algoritmo es el conocido como el algoritmo de votación de Boyer-Moore.



## 2. Tuercas y tornillos

### a) Algoritmo básico

#### Implementación

```

Procedimiento tornillosTuercasIterativo(cajonA, cajonB)
begin
  PARA cada elemento i en el rango [0, size)
    PARA cada elemento j en el rango [0, size)
      Si cajon[A] igual a cajonB[j]
        intercambia cajonB[j] con cajonB[i]
      Fin Si
    Fin PARA
  Fin PARA
end

```

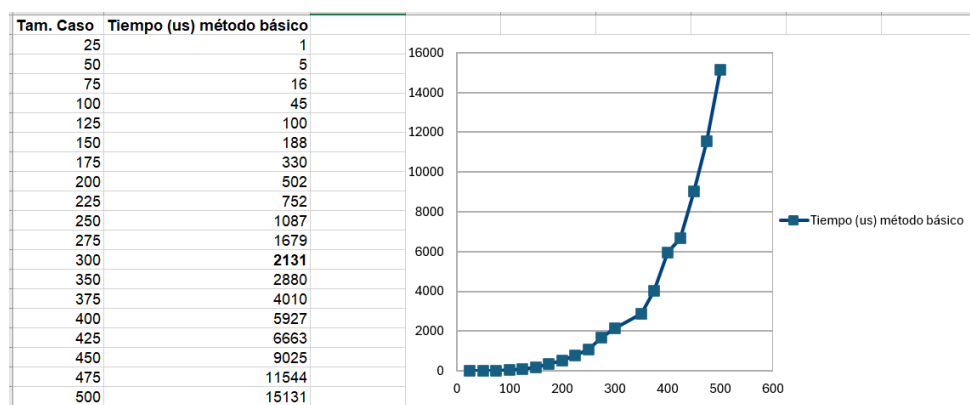
El algoritmo consiste en 2 bucles for el primer bucle for consiste en la selección de un tornillo/tuerca que esté en el conjunto del cajonA. Luego procede a buscar sus pares de tornillo/tuerca de manera iterativa en el conjunto del cajonB.

#### Eficiencia teórica

La eficiencia en el peor caso es  $O(n^2)$ , ya que tenemos dos bucles for anidados, cada uno recorriendo todo el tamaño del vector ( $n$ ), por lo tanto, cada uno es  $O(n)$  y al estar anidados tenemos una eficiencia total de  $O(n^2)$ .

#### Eficiencia práctica

Una vez implementado el algoritmo, veamos los resultados obtenidos tras la ejecución del algoritmo para extraer la eficiencia práctica. Los resultados obtenidos son los siguientes:



## b) Algoritmo Divide y Vencerás

### Implementación

Para diseñar una solución Divide y Vencerás, se propone utilizar un “doble quicksort” que ordene los tornillos usando una tuerca como pivote y tuercas, usando un tornillo pivote.

Esta función a partir de un vector de tuercas (o tornillos) cuyos índices extremos son [bajo, alto] y un pivote (de distinto tipo al del vector), coloca el pivote en la posición ordenada, de tal manera que todos los elementos a su izquierda son menores y a su derecha, mayores:

```
FUNCIÓN particionar(caja, bajo, alto, pivote)
  PARA cada elemento i en el rango [bajo, alto)
    SI caja[i] == pivote
      INTERCAMBIAR caja[i] con caja[alto]
  FIN PARA

  pivote = caja[alto]
  i = bajo

  PARA cada elemento j en el rango [bajo, alto)
    SI caja[j] < pivote
      INTERCAMBIAR caja[i] con caja[j]
      INCREMENTAR i
  FIN PARA

  INTERCAMBIAR caja[i] con caja[alto]
  RETORNAR i
FIN FUNCIÓN

Dado que se aplica el esquema Divide y Vencerás, se parte el problema en dos
subproblemas, que deberán resolverse por separado y luego combinar sus soluciones:

FUNCIÓN ordenarTuercasTornillos(tuercas, tornillos, bajo, alto)
  SI bajo < alto
    pivote = particionarTuercas(tuercas, bajo, alto, tornillos[alto])
    particionarTornillos(tornillos, bajo, alto, tuercas[pivote])
    ordenarTuercasTornillos(tuercas, tornillos, bajo, pivote - 1)
    ordenarTuercasTornillos(tuercas, tornillos, pivote + 1, alto)
  FIN FUNCIÓN

  RETORNAR i
FIN FUNCIÓN
```

## Eficiencia teórica

Las funciones de “particionar” tienen una complejidad temporal lineal ( $O(n)$ ), ya que en el peor caso recorre todos los elementos del vector (tornillos o tuercas), que es de tamaño “n”.

En la función de ordenarTuercasTornillos(), las dos llamadas a “particionar” son  $O(n)$  y las dos recursivas,  $T(n/2) + 1$ , porque se divide el problema en dos partes.

Se resuelve la recurrencia lineal no homogénea mediante un cambio de variable:  $n=2^m$   
 $T(2^m)=T(2^{m-1})+1 \rightarrow T_m - T_{m-1}=1 \rightarrow x-1=1$

Se analiza la parte homogénea:  $x-1=0 \rightarrow (x-1)$ . Y la no homogénea:  $1 = (b \cdot q(m))$  (grado d)  $\rightarrow b=1, q(m)=0, d=0 \rightarrow (x-b)^{d+1} \rightarrow (x-1)$

Queda el polinomio característico:  $P(x)=(x-1)(x-1)=(x-1)^2 \rightarrow$  cuya raíz es  $x=1$  (doble).

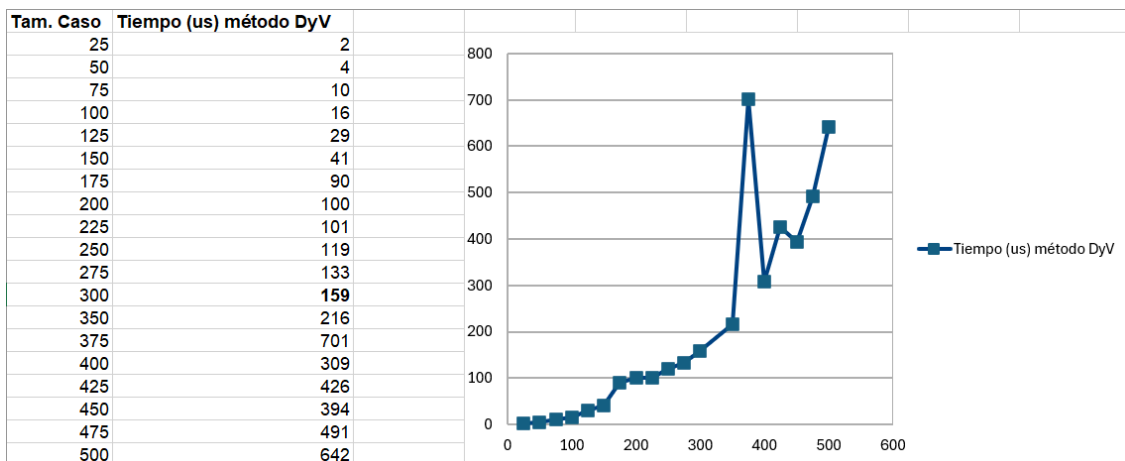
$T_m = c_1 \cdot 1^m + c_2 \cdot 1^m \cdot m \rightarrow$  Deshaciendo el cambio  $n=2^m$  queda:  $T_n = c_1 + c_2 \cdot \log_2(n)$

Por tanto, se puede concluir que el orden de eficiencia de las llamadas recursivas tienen  $O(\log_2(n))$ .

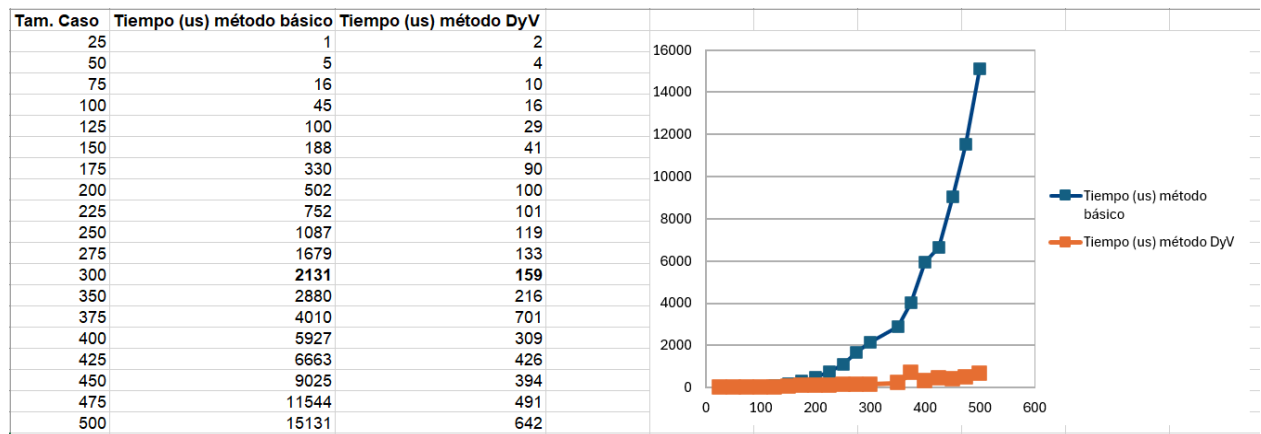
Entonces, la eficiencia del algoritmo se determina por el rendimiento del particionamiento, que es  $O(n)$ , multiplicado por el número de niveles de recursión, que es  $O(\log_2(n))$ . Por lo tanto, en el peor caso, la eficiencia del algoritmo es  **$O(n \cdot \log_2(n))$** .

## Eficiencia práctica

Una vez implementado el algoritmo DyV, veamos los resultados obtenidos tras la ejecución del algoritmo para extraer la eficiencia práctica. Los resultados obtenidos son los siguientes:



### c) Comparación gráfica y umbral



Podemos ver que para  $n=25$ , el método básico tiene un tiempo de respuesta mejor que el algoritmo Divide y vencerás pero para  $n=50$  esto cambia, resultando más eficiente el algoritmo DyV. Por tanto, situamos el umbral entre un  $n=25$  y  $n=50$ .

### 3. Producto de tres elementos

#### a) Algoritmo básico

##### Implementación

Se propone como diseño un algoritmo de búsqueda iterativo del tipo fuerza bruta.

Dado un número natural “n”, mientras que no se haya encontrado la solución (un valor “y” tal que  $n=y*(y+1)*(y+2)$ , si existe), se van recorriendo en orden los elementos en  $[1, n-1]$ :

- Si el elemento actual cumple la condición para ser solución, finaliza el bucle.
- Si no, pasa el siguiente elemento, y así sucesivamente hasta que encuentre la solución o haya realizado todas las comprobaciones hasta llegar a  $n-1$ .

##### Eficiencia teórica

Las operaciones de multiplicación y comprobación son sentencias simples ( $O(1)$ ), y el código del bucle se ejecutará en el peor de los casos  $n-1$  veces ( $O(n-1)$ ), que redondeando es lo mismo que decir  $O(n)$ .

Por tanto la eficiencia teórica será:  $\max\{O(1), O(n)\} = O(n)$  (lineal).

##### Eficiencia práctica

Tam. Caso (n)	Tiempo Básico (ns)
25	1800
50	1300
75	1700
100	1300
125	1300
150	1500
175	1300
200	1900
225	1700
250	1800
275	1700
300	1700
325	2000
375	2000
400	2300
425	2100
450	2700
475	3300
500	3500

## b) Algoritmo Divide y Vencerás

### Implementación

Teniendo en cuenta las tres condiciones mencionadas al principio del documento, se propone el siguiente diseño:

- División del problema: en cada iteración se divide en dos subproblemas de igual tamaño y se descarta uno de ellos (el que no tiene la solución).
- Al trabajar realmente con un único subproblema, no es necesario un método para combinar subsoluciones.
- El caso base se da cuando  $n=6$ , que tendría solución  $y=1$ , puesto que  $6=1*(1+1)*(1+2)=1*2*3$  y solo tendría que hacer una comprobación ( $O(1)$ ).

**NOTA:** En este caso, no ha sido necesario crear vectores ni una función que fusione las soluciones debido a la simplicidad del problema (la lista de posibles soluciones para un caso “n” son todos los números naturales menores a “n”).

Para diseñar una solución Divide y Vencerás para este problema se ha planteado el siguiente esquema, basado en una **búsqueda binaria de “y” en un arreglo [1, n-1]**:

- Se inicializan dos variables para el extremo izquierdo y el derecho (min, max) y una para el centro (mid), dividiendo el arreglo en dos mitades.
- Mientras que  $\text{min} \leq \text{max}$  (los extremos no se cruzan) y no se haya encontrado la solución, se calcula el centro (mid):
  - Si mid es la solución, finaliza el bucle y se ha resuelto el problema.
  - Si mid no es la solución, pueden darse dos casos:
    - El resultado de la multiplicación es mayor que n: el valor que buscamos se encuentra en el subarreglo de la izquierda, por lo que se descarta el derecho, actualizando el extremo  $\text{max}=\text{mid}-1$ .
    - El resultado de la multiplicación es menor que n: la solución está en el subarreglo de la derecha. Se descarta el izquierdo ( $\text{min}=\text{mid}+1$ ).

### Eficiencia teórica

Como en cada iteración se divide el problema en dos, por tanto quedaría:  $T(n)=T(n/2)+1$

Se resuelve la recurrencia lineal no homogénea mediante un cambio de variable:  $n=2^m$

$$T(2^m)=T(2^{m-1})+1 \rightarrow T_m - T_{m-1}=1 \rightarrow x-1=1$$

Se analiza la parte homogénea:  $x-1=0 \rightarrow (x-1)$

Y la no homogénea:  $1 = (b*q(m))$  (grado d)  $\rightarrow b=1, q(m)=0, d=0 \rightarrow (x-b)^{d+1} \rightarrow (x-1)$

Queda el polinomio característico:  $P(x)=(x-1)(x-1)=(x-1)^2 \rightarrow$  cuya raíz es  $x=1$  (doble).

$$T_m = c_1 * 1^m + c_2 * 1^m * m \rightarrow \text{Deshaciendo el cambio } n=2^m \text{ queda: } T_n = c_1 + c_2 * \log_2(n)$$

Por tanto, se puede concluir que el orden de eficiencia de la función será:

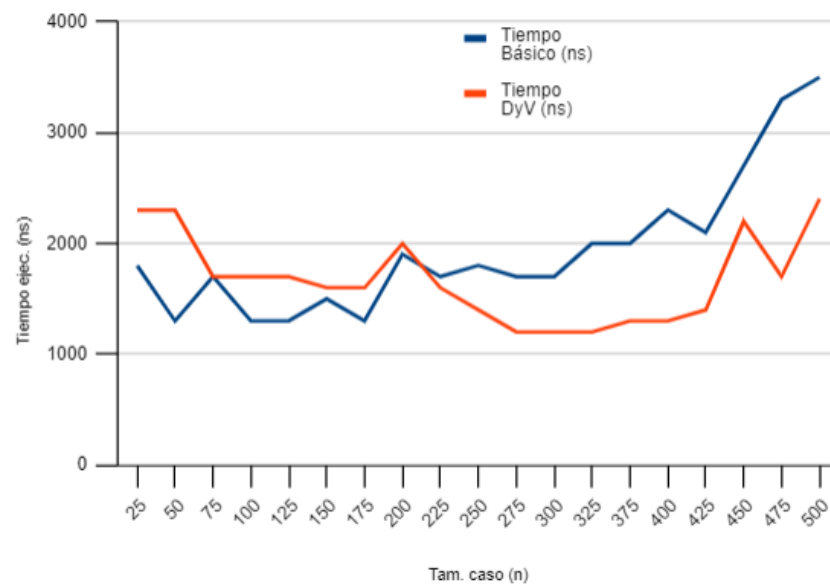
$$\max\{O(1), O(\log_2(n))\} = O(\log_2(n)).$$

## Eficiencia práctica

Tam. Caso (n)	Tiempo DyV (ns)
25	2300
50	2300
75	1700
100	1700
125	1700
150	1600
175	1600
200	2000
225	1600
250	1400
275	1200
300	1200
325	1200
375	1300
400	1300
425	1400
450	2200
475	1700
500	2400

### c) Comparación gráfica y umbral

Comparación algoritmo básico vs DyV + umbral



Como puede observarse, el umbral está contenido en el intervalo **(200,225)**, ya que a partir de este valor de “n”, suficientemente grande, el algoritmo Divide y Vencerás supera en términos de eficiencia al básico. Esto tiene sentido, puesto que la eficiencia de orden logarítmico es mejor que la lineal a medida que el tamaño de caso aumenta.

## 4. Eliminar elementos repetidos

### a) Algoritmo básico

#### Implementación

El pseudocódigo del algoritmo de fuerza bruta para eliminar duplicados de un vector de enteros podría ser de este tipo:

```
Función eliminarDuplicados(vec)
    unique = Vector de Enteros

    Si tamaño de vec > 0 Entonces
        Para cada elemento en vec
            encontrado = Falso
            Para cada elemento en unique y mientras encontrado sea Falso
                Si elemento actual de vec es igual a elemento actual de unique
                    encontrado = Verdadero
            Fin Si
        Fin Para

        Si encontrado es Falso Entonces
            Agregar elemento actual de vec a unique
        Fin Si
    Fin Para

    Devolver unique
Fin Función
```

#### Eficiencia teórica

Para analizar la eficiencia del algoritmo, vamos a desglosar su complejidad en términos del tamaño de entrada que sería  $n$ , que es la longitud del vector pasado como argumento.

El bucle externo recorre cada elemento del vector  $vec$ , lo que lleva  $O(n)$  operaciones. Dentro de este bucle, el bucle interno también realiza hasta  $O(n)$  operaciones, ya que recorre el vector  $unique$  en busca de duplicados. Esto da como resultado un tiempo total de ejecución de  $O(n^2)$ . El resto de operaciones son elementales y, por tanto,  $O(1)$ .

Como consecuencia, dado que hay un bucle anidado dentro de otro bucle, la eficiencia del algoritmo es  **$O(n^2)$** .



**Eficiencia práctica**

Tam. Caso (n)	Tiempo Básico (us)
1000	174
2000	723
3000	2437
4000	6559
5000	11206
6000	19517
7000	35372
8000	42178
9000	54001
10000	63824

**b) Algoritmo Divide y Vencerás****Implementación**

El algoritmo DyV, está basado en el MergeSort, donde ordenamos todos los elementos del vector para poder después eliminar los elementos repetidos del vector con una función a la que le pasemos como parámetro el vector previamente ordenado con MergeSort. El pseudocódigo de esta función es el siguiente:

```
Función uniqueElements(orderedVector)
    unique = Nuevo Vector de Enteros
    actual = -1

    Para cada elemento en orderedVector
        Si actual es diferente al elemento actual entonces
            actual = elemento actual
            Agregar actual a unique
        Fin Si
    Fin Para

    Devolver unique
Fin Función
```

## Eficiencia teórica

Para analizar la eficiencia del código proporcionado, primero analizaremos la eficiencia del algoritmo Merge Sort y luego la eficiencia de la función uniqueElements.

La eficiencia de Merge Sort ya sabemos que es de  $O(n \log n)$ .

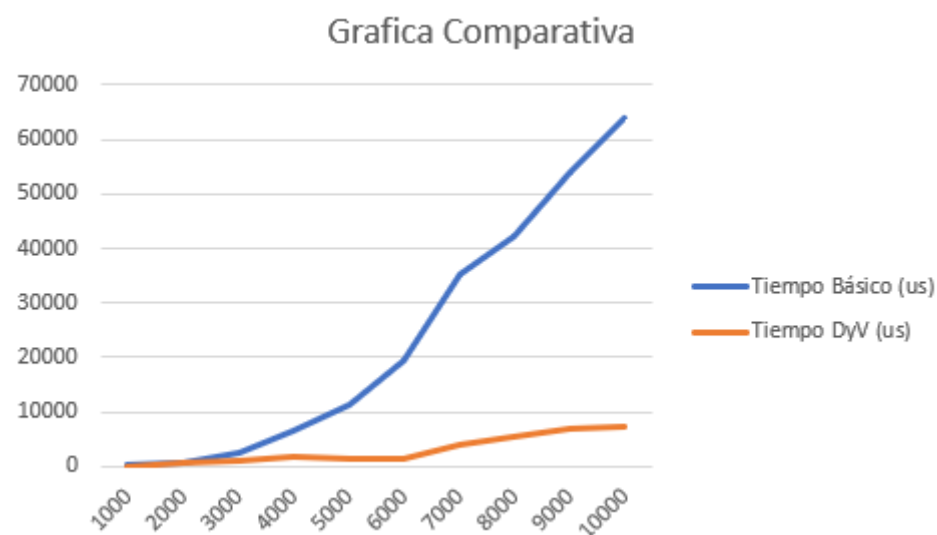
En la función uniqueElements tenemos un bucle que itera sobre cada elemento del vector una vez, lo que hace que la eficiencia sea  $O(n)$  operaciones en el peor de los casos.

Aplicando la regla del máximo tenemos que la eficiencia de este algoritmo es  **$O(n \log n)$** .

## Eficiencia práctica

Tam. Caso (n)	Tiempo DyV (us)
1000	108
2000	543
3000	1058
4000	1627
5000	1494
6000	1554
7000	3956
8000	5553
9000	6889
10000	7200

## c) Comparación gráfica y umbral



Como podemos observar, el umbral se encuentra entre los valores **(2000, 2500)**, con esto observamos que el algoritmo Divide y Vencerás empieza a ser mejor que el algoritmo básico a partir de esos valores ya que el algoritmo básico presenta una eficiencia de  $O(n^2)$  y el algoritmo Divide y Vencerás presenta una eficiencia  $O(n \log_2(n))$  por lo que vemos que el algoritmo Divide y Vencerás mejora el algoritmo básico.

## 5. Organización del calendario de un campeonato

### a) Algoritmo básico

#### Implementación

Para el diseño de este algoritmo, se ha aplicado la técnica de **"Round Robin Tournament"** (Torneo de todos vs todos), con las siguientes consideraciones:

- **Generación del calendario de enfrentamientos:** para cada día desde el primer día hasta el día  $n-1$ , se itera sobre la mitad de los equipos ( $n / 2$ ) para generar los enfrentamientos.
  - Se seleccionan dos equipos, uno del principio de la lista y otro del final, garantizando que no hayan jugado antes.
  - Si los equipos seleccionados ya han jugado antes, se aumenta el número del equipo1 y el equipo2 según sea necesario.
  - Se registra el enfrentamiento en la matriz del calendario.
  - Se añade el par de equipos al conjunto de "jugados" para evitar duplicados.
- **Rotación de equipos:** después de generar los enfrentamientos para un día, se realiza una rotación de los equipos. Esto garantiza que los equipos no se enfrenten siempre en el mismo orden en los días siguientes, agregando variedad al calendario.

#### Pseudocódigo:

```

FUNCIÓN crearCalendario(n):
  COMPRUEBA que n es par, si no lo es FINALIZA

  INICIALIZAR equipos (lista de tamaño n) con índices de 1 a n
  INICIALIZAR matriz calendario (dimensiones: nx(n-1)) a 0
  INICIALIZAR conjunto vacío de "jugados"

  PARA cada día desde 1 hasta n-1:
    PARA cada i desde 0 hasta n / 2 - 1:
      equipo1 = equipos[i]
      equipo2 = equipos[n - 1 - i]
      MIENTRAS (equipo1, equipo2) esté en "jugados":
        INCREMENTAR equipo1
        SI equipo1 > n / 2:
          equipo1 = 1
        INCREMENTAR equipo2
      FIN MIENTRAS

      calendario[equipo1 - 1][día - 1] = equipo2
      calendario[equipo2 - 1][día - 1] = equipo1

      AÑADIR (equipo1, equipo2) a "jugados"
    FIN PARA

    // ROTACIÓN
    temp = equipos[n - 1]
    PARA cada i desde n - 1 hasta 1:
      equipos[i] = equipos[i - 1]
    FIN PARA
    equipos[1] = temp
  FIN PARA

FIN FUNCIÓN

```

## Eficiencia teórica

Se realiza el análisis de las secciones del código de la función:

- La inicialización de una matriz de dimensiones  $n \times (n-1)$  a cero tiene una complejidad de  $O(n^2)$ , ya que depende del tamaño cuadrático de la matriz.
- Bucle for externo (días): itera  $n-1$  veces, aportando un  $O(n)$  al total.
- Bucle for interno (partidos): itera aproximadamente  $n/2$  veces en cada iteración del bucle externo, contribuyendo con  $O(n)$  al total.
- Búsqueda en conjunto "jugados": en el peor caso, la complejidad total de esta operación es  $O(n)$  para cada iteración del bucle externo.
- Rotación de equipos: esta operación se realiza una vez en cada iteración del bucle externo y tiene una complejidad de  $O(n)$  ya que implica el desplazamiento de los elementos del vector de equipos.

Dado que el bucle externo domina el tiempo de ejecución y las otras operaciones son proporcionales a  $n$ , la complejidad total de tiempo de la función `crearCalendario(n)` es  **$O(n^2)$** .

## Eficiencia práctica

Tam. Caso (n)	Tiempo Básico (us)
2	25
4	5
8	13
16	133
32	415
64	691
128	3911

## b) Algoritmo Divide y Vencerás

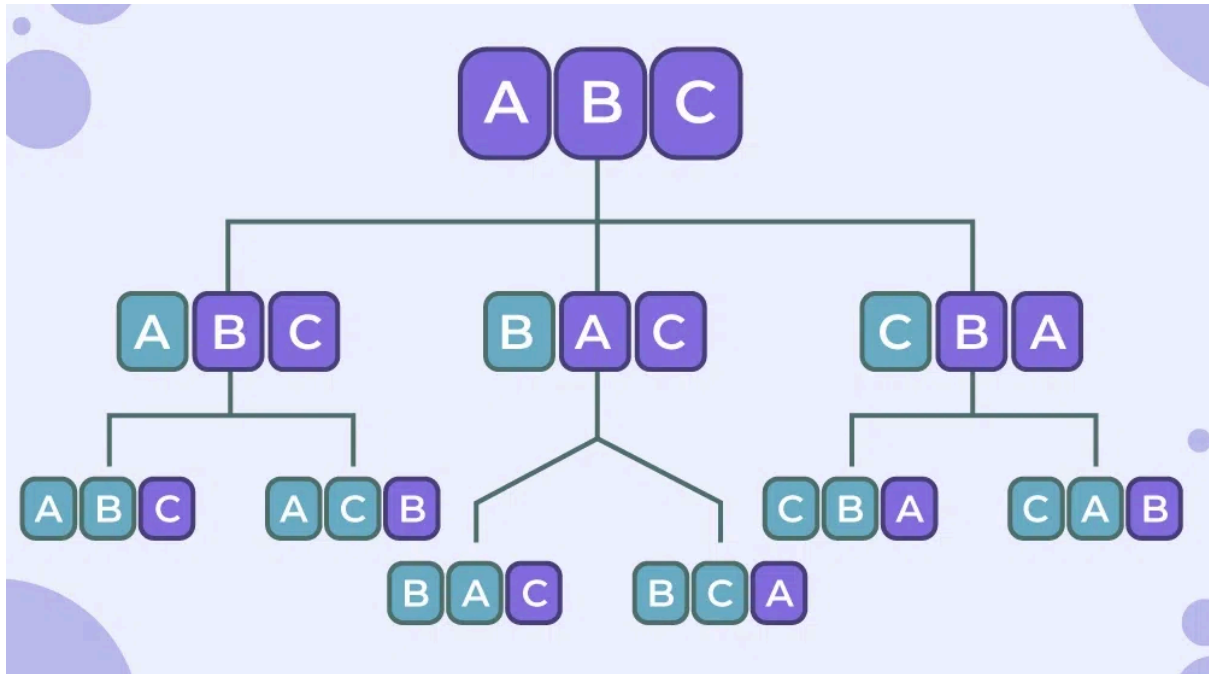
### Implementación

En la propia definición del problema, se puede entrever que es un problema de permutaciones, donde al tener un número par de oponentes, siempre tiene que permutarse entre ellos para realizar el enfrentamiento.

El algoritmo DyV consiste en generar todas las permutaciones de manera recursiva. Ya que la restricción del problema es que para cada uno de los oponentes  $N$  se tienen que generar  $N-1$  partidos excluyéndose a sí mismo.

El algoritmo tiene "la desventaja" de que tiene que generar casi todos los casos ya que no podemos determinar de manera eficiente las "repeticiones". Conocemos como repetición aquellos casos en los que AB supone lo mismo que BA.

En todo caso sí podemos limitarnos a coger la rama del equipo 1, en el que cuando se llega a un caso base, el primer elemento que se inserta es 1,2,3,4....N por lo que conociendo que la primera rama es 1 (o cualquier otro en realidad), podemos obtener los partidos del torneo para todas las permutaciones posibles con el 1 fijo.



En el ejemplo nos limitaremos a la rama del A, pero cualquier otra rama es igual de válida.

```

Procedimiento permutations(res, nums, l, h)
  // Caso base si no se tienen más permutaciones
  Si l es igual a h entonces
    // Siguierte inserción se permite solo si el primer elemento está en el mismo equipo
    // PRIMERO determinamos si tiene algún elemento en la estructura de datos de la permutación
    // siguiente a insertar
    Si tamaño(res) > 0
      Y res[tamaño(res)-1][0] es igual a nums[0]
      Y res[tamaño(res)-1][1] no es igual a nums[1] entonces
        res.agregar(nums)
    // Primera inserción de todas
    Sino Si tamaño(res) es igual a 0 entonces
      res.agregar(nums)
    Fin Si
    retornar
  Fin Si

  // Haciendo permutaciones desde l hasta h inicialmente
  Para i desde l hasta h hacer
    // Intercambiar permutaciones
    intercambiar(nums[l], nums[i])
    // Incrementar el siguiente número a intercambiar
    permutations(res, nums, l + 1, h)
  Fin Para
Fin Procedimiento
  
```

```

// Función para obtener las permutaciones
permute(nums)
begin
  // Declarar la variable resultado
  res
  x = tamaño(nums) - 1

  permutations(res, nums, 0, x)

  retornar res
end

```

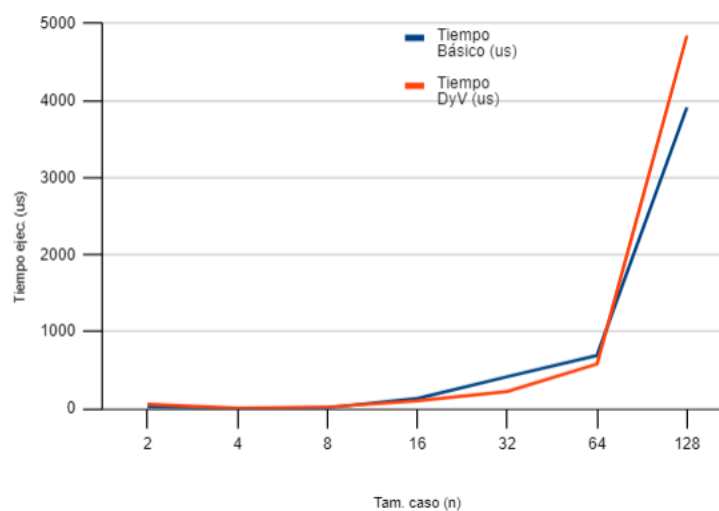
### Eficiencia teórica

Para cada equipo, tenemos que realizar una permutación en un orden del factorial:  $O(N*N!)$

### Eficiencia práctica

Tam. Caso (n)	Tiempo DyV (us)
2	58
4	7
8	23
16	103
32	222
64	581
128	4843

### c) Comparación gráfica y umbral



Como se puede observar en la gráfica de líneas, el umbral a partir del cual el algoritmo Divide y Vencerás presenta una mejor eficiencia práctica que el básico es  $n=64$ .