**GRADO EN INGENIERÍA INFORMÁTICA**

**CURSO 2023-2024**

**ALGORÍTMICA**

**Práctica 2 - Algoritmos Divide y Vencerás**

**Miguel Martinez Azor**

**Ángel Rodríguez Faya**

**Alejandro Botaro Crespo**

**Alberto Parejo Bellido**

**Alejandro Ocaña Sánchez**

**02/04/2024**

# **ÍNDICE**

[**ÍNDICE 1**](#_hf5vqotbil0)

[**1. La mayoría absoluta 2**](#_l0htbnwle4tn)

[a. Diseñar un método básico (no Divide y Vencerás) que resuelva el problema. Estudiar su eficiencia teórica. 2](#_okvhj62nxjyf)

[b. Estudiar como el problema puede ser abordado mediante la técnica Divide y Vencerás y realizar el diseño completo. Estudiar su eficiencia teórica. 2](#_au3ri24jf5zc)

[c. Implementar los algoritmos básico y Divide y Vencerás. Resolver el problema del umbral de forma experimental. 3](#_l30kjksktqst)

[**2. Tuercas y tornillos. 7**](#_tuvcki47xahx)

[a. Diseñar un método básico (no Divide y Vencerás) que resuelva el problema. Estudiar su eficiencia teórica. 7](#_ttargqaqoixc)

[b. Estudiar cómo el problema puede ser abordado mediante la técnica Divide y Vencerás y realizar el diseño completo. Estudiar su eficiencia teórica. 8](#_7gqwo259r3oc)

[c. Implementar los algoritmos básico y Divide y Vencerás. Resolver el problema del umbral de forma experimental. 8](#_xeaov8d3ca5)

[**3. Producto de tres elementos 10**](#_uhkamy45ole)

[a. Diseñar un método básico (no Divide y Vencerás) que resuelva el problema. Estudiar su eficiencia teórica. 10](#_xykfwie9jejv)

[b. Estudiar cómo el problema puede ser abordado mediante la técnica Divide y Vencerás y realizar el diseño completo. Estudiar su eficiencia teórica. 11](#_cmz4t0gwpf5n)

[c. Implementar los algoritmos básico y Divide y Vencerás. Resolver el problema del umbral de forma experimental. 13](#_5d2raa21mje0)

[**4. Eliminar elementos repetidos 15**](#_yut7yc59q9xn)

[a. Diseñar un método básico (no Divide y Vencerás) que resuelva el problema. Estudiar su eficiencia teórica. 15](#_mnvtb9wsvj2n)

[b. Estudiar cómo el problema puede ser abordado mediante la técnica Divide y Vencerás y realizar el diseño completo. Estudiar su eficiencia teórica. 16](#_o3f44qwdpmdv)

[c. Implementar los algoritmos básico y Divide y Vencerás. Resolver el problema del umbral de forma experimental. 17](#_6ngbkr6jf04o)

[**5. Organización del calendario de un campeonato. 20**](#_hi4t6a404jdy)

[d. Diseñar un método básico (no Divide y Vencerás) que resuelva el problema. Estudiar su eficiencia teórica. 20](#_r875gnqkd67k)

[e. Estudiar como el problema puede ser abordado mediante la técnica Divide y Vencerás y realizar el diseño completo. Estudiar su eficiencia teórica. 21](#_atumy7tsxxup)

[f. Implementar los algoritmos básico y Divide y Vencerás. Resolver el problema del umbral de forma experimental. 22](#_8dh74nqjscvp)

[**6. Ejemplos de compilación y ejecución. 25**](#_c7c5t9vq7jts)

# 

# **1. La mayoría absoluta**

ENUNCIADO:

Dado un vector de enteros V de tamaño n que en cada posición contiene el código numérico del candidato votado por una persona (hay n votos), se desea conocer si hay algún candidato "x" que tenga mayoría absoluta (puede haber uno solo, o ninguno), es decir necesita más (estrictamente mayor) de n/2 votos. O lo que es lo mismo: Card{i | v[i]=x} > n/2. No se conoce a priori quienes son los candidatos. Por ejemplo si n = 10 tiene que tener 6 votos o más. Si n = 11, necesita también 6 votos o más. No se puede suponer que exista una relación de orden entre los elementos del vector.

SE PIDE:

#### Diseñar un método básico (no Divide y Vencerás) que resuelva el problema. Estudiar su eficiencia teórica.

Este algoritmo resuelve el problema con dos bucles for y 3 condicionales. Consiste en recorrer el vector de votos por cada candidato y cada vez que se encuentre su número asignado(que serían los votos a su favor) se incremente su contador en 1. Los números de votos que tiene de cada candidato se guarda en un vector de enteros y cada posición representa a un candidato. Después nos encontramos con un condicional que compara los votos del candidato al que se le han sumado los votos en esa iteración con el del más votado(que al principio es el primero v[0]), y si es mayor , pasa a ser el más votado y su índice se guarda en la variable más votado. Al salir de los bucles esta variable guardará el índice del candidato mas\_votado y con la última condición if se comprueba si esa cantidad de votos supera a la mitad votos totales más 1, se devolverá true, en cualquier otro caso se devolverá false.

for i=0 hasta k

for j=0 hasta n

if v[j]==i

contador[i]++;

if contador[i] >= contador[mas\_votado]

mas[votado]=i;

La eficiencia de este algoritmo es O(k\*n), ya que todas la sentencias dentro de las condicionales y las mismas condicionales son O(1). El primer bucle siempre va a hacer k iteraciones, por lo tanto es O(k) y el segundo bucle es O(n) ya que siempre va recorrer todo el vector de tamaño de n por cada iteración.

#### Estudiar como el problema puede ser abordado mediante la técnica Divide y Vencerás y realizar el diseño completo. Estudiar su eficiencia teórica.

Para abarcar este problema mediante la técnica divide y vencerás, hemos utilizado un algoritmo parecido a la búsqueda binaria pero centrado a buscar mayoría de un número en los subvectores de la siguiente manera:

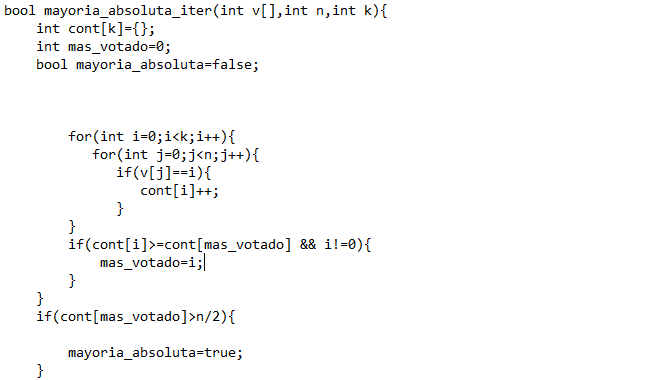
Este algoritmo divide el vector de los votos en sub vectores hasta que los vectores sean de 1 posición y empiezan a juntarse los subvectores y ver si el mismo número es mayoría absoluta en cada subvector(la parte derecha e izquierda), si es así se devuelve el número del candidato que ha salido mayoría absoluta , en caso contrario se mira si hay mayoría absoluta tanto en la parte izquierda como en la derecha se mira si uniendo las dos partes el elemento que era mayoría en una de las partes sumándole los votos en la otra, es mayoría en el vector unido, de lo contrario se devuelve -1. se cuentan votos con la función contar\_votos que recorre el área designada del vector y devuelve el número de veces que sale cierto candidato. Todo eso se mete dentro de la función encontrar mayoría para solo tener que pasar el vector y el tamaño del mismo y dentro se ponen el resto del parámetro(principio y final). Se parece a la búsqueda binario ya que se divide el vector con la variable mitad y se resuelve recursivamente cada parte.

La eficiencia teórica de este algoritmo es O(n\*log(n)) ya que contar votos es O(n) y mayoria\_absoluta\_dyv se divide recursivamente en log(n) niveles. Esta eficiencia para grandes casos es bastante bueno y eso se verá en las tablas y gráficas donde se mide el tiempo para tamaños de casod e más pequeños a mas grandes

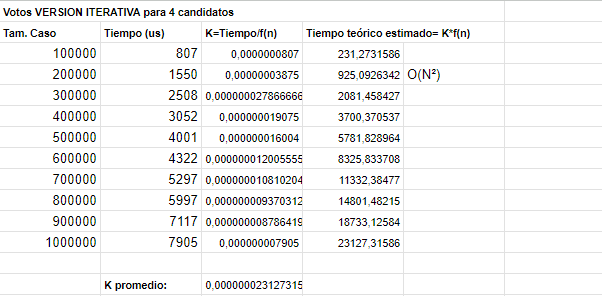
#### Implementar los algoritmos básico y Divide y Vencerás. Resolver el problema del umbral de forma experimental.

Al pasar a código los algoritmos nos quedamos con las siguientes implementaciones:

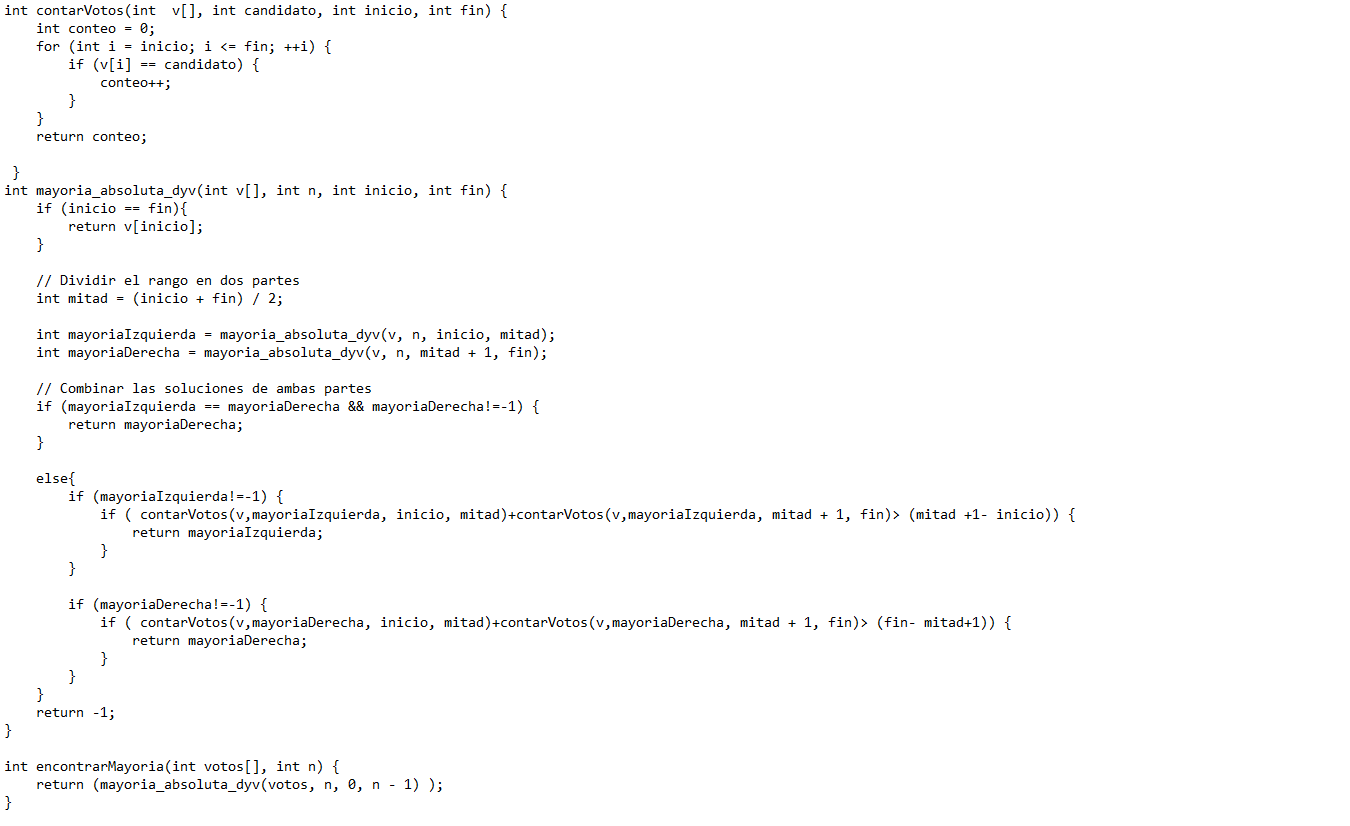
* **Iterativo:**



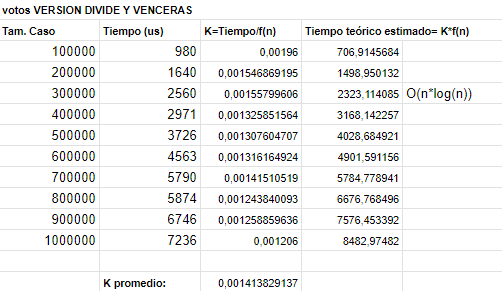
Que presenta los siguientes tiempos con valores de 100000 a 1000000:



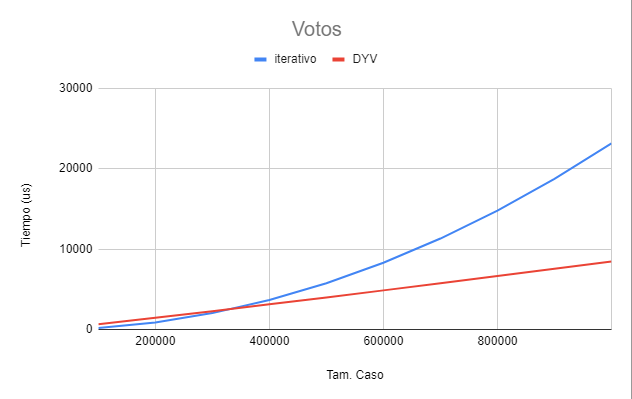
* **Divide y Vencerás:**



Que presentan los siguientes datos con los mismos valores que el iterativo:

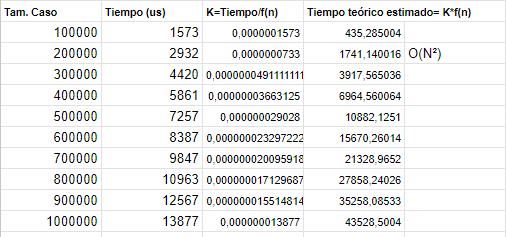


Tras implementar los algoritmos y haber recopilado los datos de tiempo de manera experimental de los dos algoritmos se puede notar en la siguiente gráfica la diferencia de tiempos:

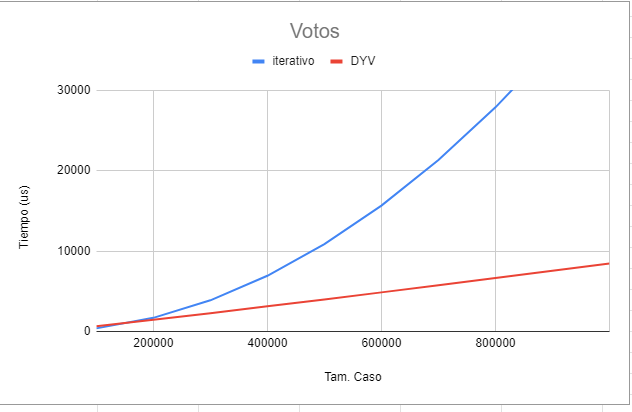


Se puede ver que a partir de n=300000 aproximadamente el algoritmo divide y vencerás es bastante mejor que el iterativo ya que está hecho para ser mejor en grandes casos. Además hay que tener en cuenta que si hubiera más candidatos afectaría en tiempos al iterativo pero no al divide y vencerás lo que lo hace más eficiente aún. como por ejemplo en esta otra gráfica con 12 candidatos:

Iterativo con 12 candidatos:



(El divide y vencerás tiene los mismos tiempos ya que no influye el número de candidatos)



Aquí se puede ver que el algoritmo divide y vencerás empieza a ser más eficiente en los 200000. por lo tanto cuantos más candidatos haya, más rentable será escoger divide y vencerás(aunque con 2 candidatos que es lo minimo también es más rentable en casos grandes).

# **2. Tuercas y tornillos.**

ENUNCIADO:

En una habitación oscura se tienen dos cajones, en uno de los cuales hay n tornillos de varios tamaños, todos distintos, y en el otro las correspondientes n tuercas. Es necesario emparejar cada tornillo con su tuerca correspondiente, pero debido a la oscuridad no se pueden comparar tornillos con tornillos ni tuercas con tuercas, y la única comparación posible es la de intentar enroscar una tuerca en un tornillo para comprobar si es demasiado grande, demasiado pequeña, o se ajusta perfectamente al tornillo.Para ello, Tenemos dos sectores a y b de tamaño n que contienen los mismos números enteros (que son distintos entre sí (no se repiten números) pero en diferente orden. Se pretende emparejarlos elementos de los dos vectores, de forma que el orden de los elementos en ambos vectores sea el mismo (no importa qué orden es, lo importante es que los elementos iguales en ambos vectores están en la misma posición, o sea que los dos vectores sean iguales). La restricción es que no es posible comparar entre sí los elementos de un mismo vector, solo se pueden realizar comparaciones (de mayor, menor o igual) entre elementos de distintos vectores (por ejemplo se puede preguntar si a[i] < b[j], pero no se puede preguntar si a[i] < a[j]).

SE PIDE:

#### Diseñar un método básico (no Divide y Vencerás) que resuelva el problema. Estudiar su eficiencia teórica.

Para resolver este problema utilizando un algoritmo iterativo, la opción más sencilla que se plantea sería ir comparando cada elemento del vector tornillos con un elemento del vector tuercas, es decir, esto sería lo mismo que coger un tornillo e ir probando en cada una de las tuercas a ver en cual encaja. Esta sería una posible solución y la que se ha llevado a cabo ya que no se pueden comparar elementos de un mismo vector.

Un posible pseudocódigo que ilustra el funcionamiento sería el siguiente:

function tornillostuercas(tornillos[], tuercas[]):

n = longitud(a)

para i desde 0 hasta n-1 hacer:

para j desde 0 hasta n-1 hacer:

si tornillos[i] == tuercas[j] :

swap(tuercas[i], tuercas[j])

break

end

Suponemos que ambos vectores tienen la misma longitud y a partir de ahí recorremos cada uno de ellos de l amanera que para cada componente del primer vector, exploramos el segundo vector hasta encontrar coincidencia, una vez se haya encontrado coincidencia se intercambia la posición del elemento del segundo vector por la posición del segundo vector pero con respecto al vector primero, en este caso tornillos.

En este algoritmo iterativo su eficiencia teórica seria O(n²) ya que la función al estar formada por dos bucles cuyos recorridos son los mismos en este caso tamaño n, en el peor de los casos se harán n\*n iteraciones, pero también hay que destacar que aunque es poco probable que esto ocurra ya que tendrían que estar los vectores ordenados de manera diferente y además que coincida que a la hora de hacer el cambio de componente, la componente afecta quede en una posición del vector la cual se necesite recorrer el segundo vector siempre entero.

#### Estudiar cómo el problema puede ser abordado mediante la técnica Divide y Vencerás y realizar el diseño completo. Estudiar su eficiencia teórica.

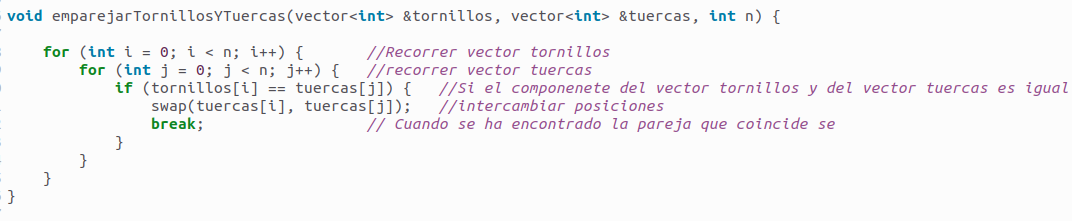
Para resolver el problema de las tuercas hemos estudiado los distintos algoritmos de la práctica anterior como por ejemplo Quicksort y ver cómo se adapta al problema.

Usando Quicksort como base podemos decir que un elemento del vector contrario al que queremos ordenar será nuestro pivote. Es decir, seleccionó un tornillo como pivote y utilizó ese tornillo para particionar las tuercas, al particionar las tuercas podremos sacar la pareja del tornillo he utilizarla como pivote de tornillos. De este modo podríamos llamar recursivamente a la función para así enlazar todas las parejas simplemente ordenando los vectores.

Su eficiencia teórica en el peor caso debe de ser semejante a la del algoritmo Quicksort, recordemos que era O(n2) y qué sucedía cuando el vector está ordenado ya que escogeremos como pivote el menor o mayor elemento, en este caso no es semejante, es la misma.No obstante no nos fijamos normalmente en el peor caso de quicksort, ya que obtener está eficiencia es algo no común, normalmente veremos una complejidad O(nlog(n)).

#### Implementar los algoritmos básico y Divide y Vencerás. Resolver el problema del umbral de forma experimental.

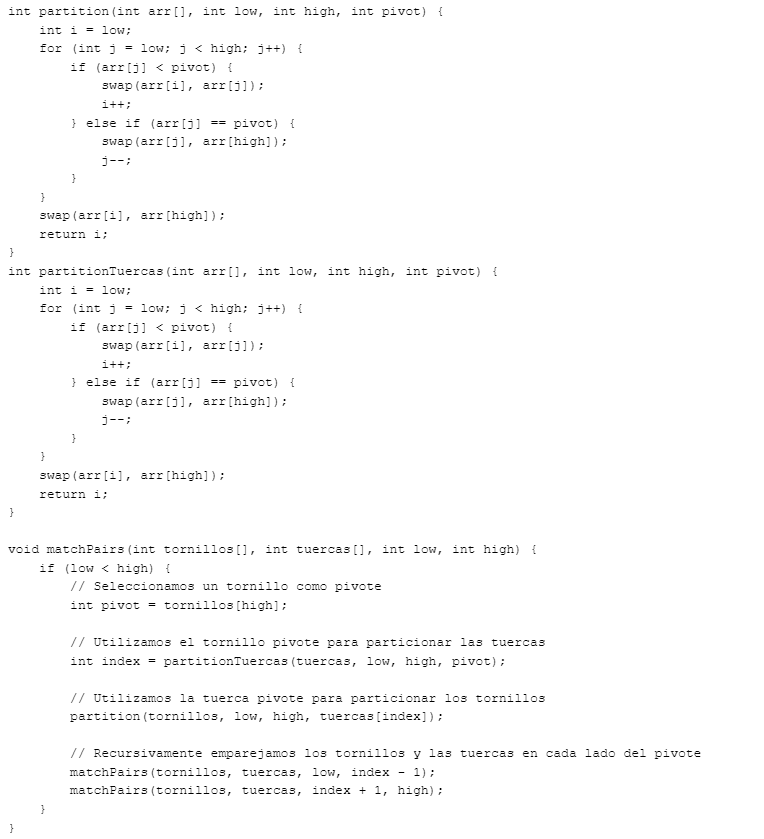
Una implementacion para el método iterativo podria ser la siguiente:

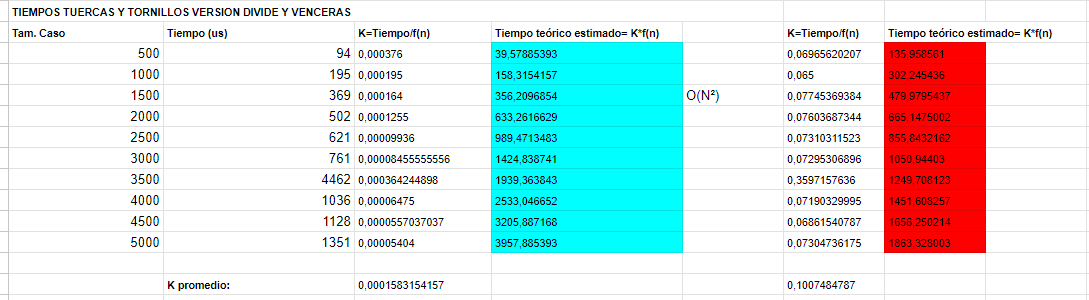


Esta implementación sigue la dinámica explicada en el punto a) de este mismo ejercicio.

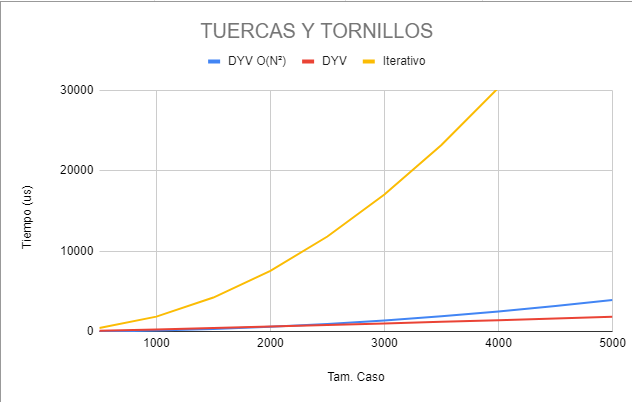
Se han obtenido los siguientes tiempos para tamaños desde 500 a 5000:

Desde el punto de vista de divide y vencerás se ha implementado el método match pairs:



Se han obtenido los siguientes resultados: Siendo el tiempo teórico remarcado en azul estimado en y en rojo .

Al unir los resultados de ambos caminos que resuelven el problema de las tuercas y los tornillos obtenemos la siguiente gráfica:

En la práctica se puede apreciar como divide y vencerás en este problema es mejor en todos los aspectos, grandes y pequeños tamaños de casos, esto lo podemos apreciar en la k, donde en el iterativo tenemos una k con dos ceros en divide y vencerás vemos una k con 4 ceros. Por lo que no hay umbral visible donde el algoritmo básico supere al divide y vencerás.

# **3. Producto de tres elementos**

ENUNCIADO:

Determinar si un cierto número natural N puede expresarse como producto de tres números naturales consecutivos.

SE PIDE:

#### Diseñar un método básico (no Divide y Vencerás) que resuelva el problema. Estudiar su eficiencia teórica.

Para resolver este problema usando una técnica iterativa se ha optado por la siguiente solución:

Function producto\_tres\_consecutivos(N)

raiz\_cubica = RedondeoalaAlta(RaízCúbica(N))

Para i desde 1 hasta raiz\_cubica:

producto = i \* (i + 1) \* (i + 2)

Si producto es igual a N:

Devolver true

Si producto es mayor a N:

Devolver true

end

Devolver false

end

El funcionamiento de este algoritmo es el siguiente:

Al programa principal se le pasa como argumento un número natural N, el cual se pasa como parámetro de una función bool que devuelve true o false dependiendo si dicho número es producto de número consecutivos o no. Realmente es como si fuera un vector de números naturales desde 1 hasta N, y para componente de ese “vector de naturales” se calcula si es producto o no.

Para calcular si es o no producto se ilustrara con el siguiente ejemplo:

Supongamos que queremos verificar si el número N=120 puede expresarse como el producto de tres números naturales consecutivos.

1. Comenzamos calculando la raíz cúbica de N que es 5.84 y redondeando a la alta nos quedamos con 6.
2. Iteramos desde 1 hasta 6.
3. Para i=1, calculamos el producto de i, i+1, i+2, que es 1\*2\*3=6 !=N
4. Para i=2, 2\*3\*4=24 !=N
5. Para i=3, 3\*4\*5=60 !=N
6. Para i=4, 4\*5\*6=120 ==N --> Devoldemos true

Para este caso se ha encontrado que para 120 es producto de números consecutivos por lo tanto devolverá true, si otro número no se encuentra el producto se devuelve false y pasa a la siguiente iteración.

La eficiencia teórica de este algoritmo es O() ya que en el peor de los casos como mucho para cada iteración va a hacer raíz Cubica de N iteraciones. Esto se hace para evitar que se recorra el vector entero cada vez que se quiera comprobar un número, ya que para números mayores que raíz cúbica de N, el producto de tres consecutivos nunca se cumplirá y por lo tanto se estarían haciendo iteraciones tontamente.

A este programa se le pasaría como argumento valores de N, entonces supongamos que la llamada a la ejecución sería ./producto\_iterativo 500, en este caso iría comprobando cada número desde 1 hasta 500 y sacaría por pantalla cual de esos 500 números cumplen que son productos de números consecutivos.

#### Estudiar cómo el problema puede ser abordado mediante la técnica Divide y Vencerás y realizar el diseño completo. Estudiar su eficiencia teórica.

Para resolver este problema utilizando una técnica de Divide y Vencerás podemos dividir el rango de números a probar en subintervalos más pequeños y luego realizar una búsqueda recursiva en estos subintervalos. La idea es reducir gradualmente el espacio de búsqueda hasta encontrar una solución o determinar que no existe ninguna.

Un pseudocódigo que ilustra esta solución sería el siguiente:

Función esProductoDeTresConsecutivos(N, a, b, c):

Si a \* b \* c es igual a N, Devuelve Verdadero

Sino, Devuelve Falso

end

Función buscarProductoTresConsecutivos(N, bajo, alto):

Si bajo es mayor que alto:

Devuelve Falso

Calcular medio como (bajo + alto) / 2

Si el producto de los tres números consecutivos en el medio es igual a N:

Devuelve Verdadero

Si N es menor que el producto de los tres números consecutivos en el medio:

Devuelve buscarProductoTresConsecutivos(N, bajo, medio - 1)

Sino:

Devuelve buscarProductoTresConsecutivos(N, medio + 1, alto)

end

* La función “esProductoDeTresConsecutivos” verifica si un número dado “N” es el producto de tres números consecutivos “a”, “b” y “c”.
* La función “buscarProductoTresConsecutivos” Divide el rango de búsqueda en dos mitades y luego realiza una búsqueda recursiva en la mitad que puede contener la solución. Si encuentra la solución, devuelve Verdadero. Si el número dado “N” es menor que el producto de los tres números consecutivos en el medio, la búsqueda se realiza en la mitad izquierda; de lo contrario, se realiza en la mitad derecha.

La eficiencia teórica de este algoritmo viene determinada por la siguiente recurrencia:

**T(n) = T(n/2) + O(1)**

donde,

**T(n)**-> número de operaciones de la función buscarProductotresconsecutivos para un tamaño n.

**T(n/2)**-> en cada llamada el tamaño de búsqueda se reduce a la mitad

**O(1)-**> operaciones y comparaciones restantes

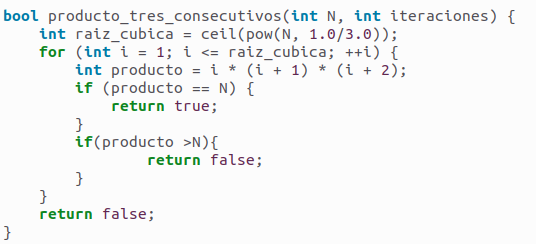
Para resolver esta recurrencia utilizamos un cambio de n=log n.

T(n) = c1 +

T(n) ∈O(log(n))

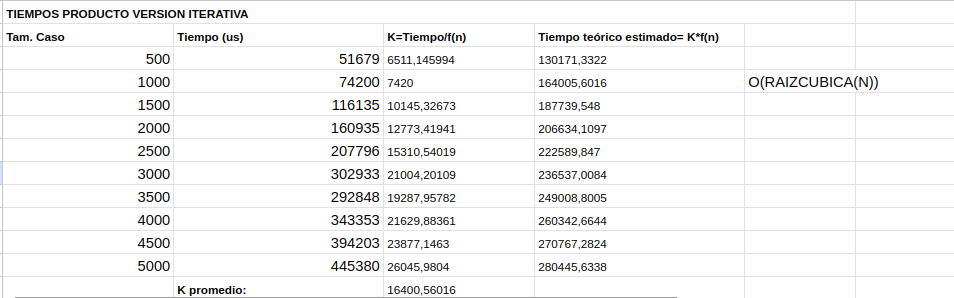
#### Implementar los algoritmos básico y Divide y Vencerás. Resolver el problema del umbral de forma experimental.

Una implementacion para el método iterativo seria la siguiente:

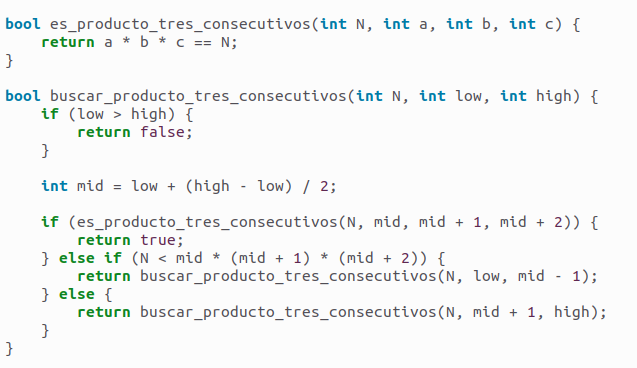


Se puede observar que como se ha explicado en el apartado a) de este mismo problema, la implementación se adapta al pseudocódigo proporcionado anteriormente, donde básicamente se puede ver que el bucle solo llegará hasta raíz cúbica de n en el peor de los casos ya que si por ejemplo N=120, La Raíz de 120 es 6 y 6 \* 7 \* 8 = 336 y se pasaría de N, por lo tanto esto quiere decir que ningún producto mayor que la raíz de N sería el N buscado.

Para una evaluación gráfica de este algoritmo los tiempos que se han obtenido con tiempos desde 500 a 5000 en intervalos de 500 en 500 son los siguientes:



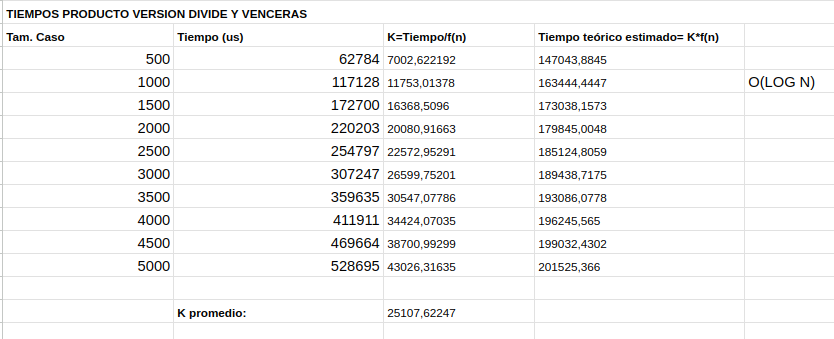
A continuación se presenta una implementación de este mismo algoritmo usando una técnica Divide y Vencerás, siguiendo los pasos descritos en el apartado b) de este mismo problema:



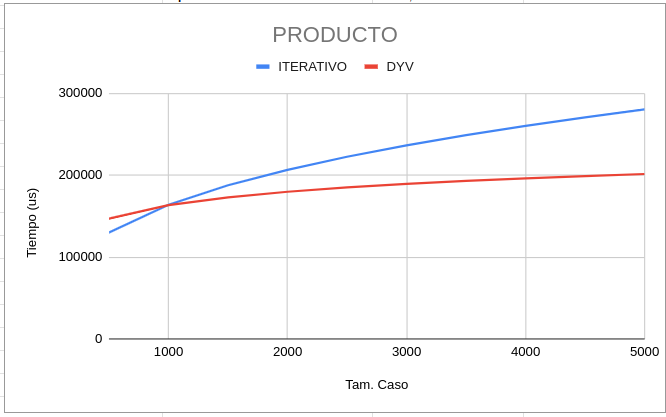
El caso base de este algoritmo viene determinado por (low > high) ya que si los índices por los que se está buscando son lo suficientemente pequeños que no se puedan dividir más o dichos índices se cruzan, no se ha encontrado solución.

Por otro lado como se ha explicado anteriormente la recursión va dividiendo el vector en tamaños aproximadamente de la mitad y va realizando una búsqueda hasta encontrar solución o llegar al caso base. Como se puede observar en el segundo condicional anidado, si el número es producto de tres números consecutivos, solución encontrada devolverá verdadero; si el número es menor que el producto de la mitad, buscamos en la mitad inferior y si no buscamos en la superior.

Para una evaluación gráfica de este algoritmo los tiempos que se han obtenido con tiempos desde 500 a 5000 en intervalos de 500 en 500 son los siguientes:



Finalmente como una comparación gráfica de ambos se ve de la siguiente forma:



Como se observa en el gráfico, para tamaños de caso menores de 1000 aproximadamente, la versión del algoritmo en su forma iterativa (fuerza bruta) es mejor que la versión Divide y Vencerás. A partir de este tamaño de caso, el algoritmo Divide y Vencerás reduce significativamente su tiempo debido a su crecimiento logarítmico. Un algoritmo con O() su crecimiento es muy parecido a un O(log(n)), pero conforme se va aumentando el tamaño un logarítmico tiende asintóticamente antes a hacerse lineal que una raíz.

Si miramos los tiempos vemos que en tamaño de caso 1000 el iterativo su tiempo teórico ha sido 164005.6016 y para el DyV 163444.4447 lo que indica que el umbral a partir del cual ambas grafican se cruzan y cambian su tendencia general es menor a 1000 .

# **4. Eliminar elementos repetidos**

ENUNCIADO:

Dado un vector de n elementos, se pide eliminar todos los elementos duplicados, es decir, que estén más de una vez en el vector.

SE PIDE:

#### Diseñar un método básico (no Divide y Vencerás) que resuelva el problema. Estudiar su eficiencia teórica.

El algoritmo resuelve el problema con dos bucles for anidados y cada uno con un condicional y otro bucle for con un condicional dentro. Este algoritmo recorre el vector pasando por todos los números que haya en el vector y luego con un if compara cada vez que el elemento del vector sea distinto de -1, si es así se mete en otro bucle for. Este recorre de nuevo el vector con una iteración más que el primer bucle, y tiene de nuevo un condicional que compara el elemento del primer for con el elemento del segundo for, es decir, va comparando cada uno de los elementos del vector con el resto de elementos del vector. Si son iguales modifica el elemento del segundo for sustituyéndolo por un -1. Una vez recorra el vector entero, creamos un vector auxiliar y nos metemos en un nuevo for que tiene dentro un condicional que compara los elementos del vector con -1, si son diferentes se almacenan en el vector auxiliar. Al final los números guardados en el vector auxiliar se pasan al vector de enteros, obteniendo así el vector sin los repetidos.

for i=0 hasta tamaño de v

if v[i] != -1

for j=i+1 hasta tamaño de v

if v[i] == v[j]

v[j] = -1

vector aux

for i hasta tamaño de v

if v[i] != -1

aux añade v[i]

La eficiencia de este algoritmo es O(n2), ya que tiene un bucle for anidado dentro de otro y dentro condicionales que son O(1). Luego el otro bucle no afecta porque no está anidado.

#### Estudiar cómo el problema puede ser abordado mediante la técnica Divide y Vencerás y realizar el diseño completo. Estudiar su eficiencia teórica.

Para abarcar el ejercicio mediante la técnica divide y vencerás, hemos utilizado un algoritmo al estilo merge sort:

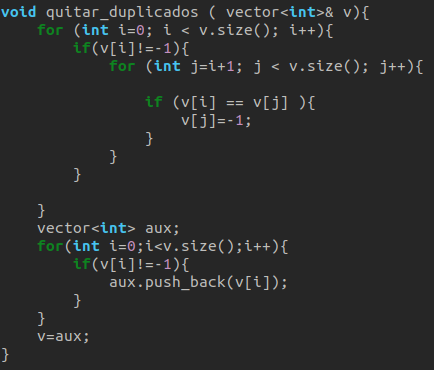
Este algoritmo funciona recursivamente desde una función principal que va ir dividiendo el vector de números repetidos en dos partes repetidamente hasta obtener subvectores de una posición. Luego combina los subvectores de una posición ordenándolos en el proceso con una función aparte. En esta función se combinan en orden los dos subvectores en otro vector que devolveremos al final del proceso. Y se repite esta secuencia una y otra vez hasta llegar al vector base ahora ordenado. Finalmente con el vector ordenado creamos un bucle donde usamos dos iteradores para comparar si hay números repetidos (si los hay estarán todos seguidos) por lo que se iría eliminando posiciones hasta encontrar un número diferente.

La eficiencia teórica de este algoritmo es O(n\*log(n)) ya que la función principal tendrá un while que es O(n) y luego la combinación de los subvectores también poseerá como mucho un for que sería O(n) también. Finalmente añadimos log(n) para contar el número de iteraciones que hará el algoritmo.

#### Implementar los algoritmos básico y Divide y Vencerás. Resolver el problema del umbral de forma experimental.

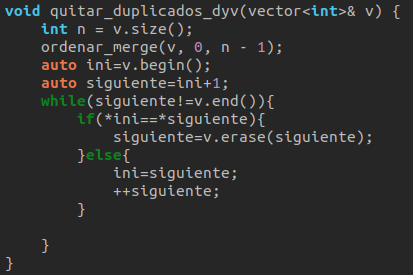
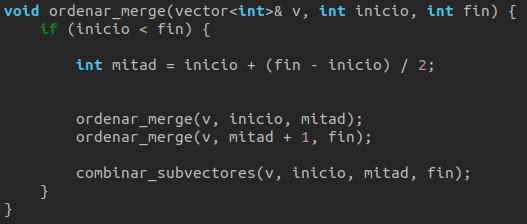
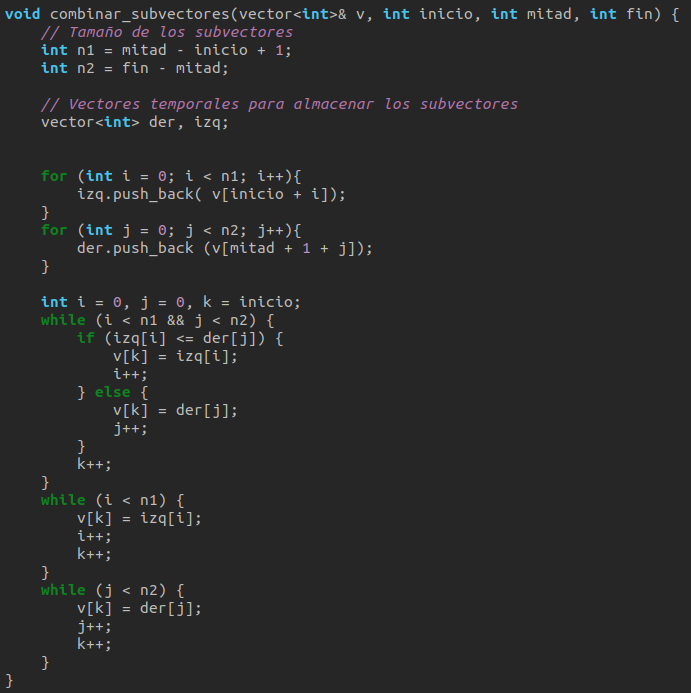
Una implementación para los algoritmos podría ser:

- **Método iterativo**



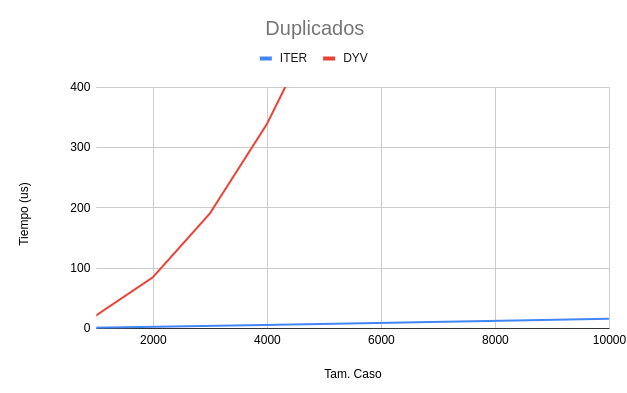
Que da los siguientes tiempos en un intervalo de 1000 a 10000:



- **Divide y vencerás** 

Que presentan los siguientes datos con los mismos valores que el iterativo:



Tras implementar los algoritmos y haber recopilado los datos de tiempo de manera experimental de los dos algoritmos se puede notar en la siguiente gráfica la diferencia de tiempos: 

En este problema se ve claramente como este algoritmo divide y vencerás es mucho peor en cualquier aspecto, esto lo podemos apreciar en la k, donde en el iterativo tenemos una k con cinco ceros y en divide y vencerás vemos una k con 3 ceros. Por lo que no hay umbral visible donde el algoritmo básico supere al divide y vencerás.

# **5. Organización del calendario de un campeonato.**

ENUNCIADO:

Se organiza un torneo con n participantes. Cada participante tiene que competir exactamente una vez con todos los posibles oponentes. Además, cada participante tiene que jugar exactamente un partido cada día. Por concreción, y sin pérdida de generalidad, puede suponerse que las competiciones se celebran en días sucesivos y que cada participante compite una vez por día. Podemos suponer que el número de participantes es potencia de dos, lo que nos simplificará el problema (no es necesario que haya jornadas de descanso). Por lo tanto n = 2^k participantes, con k entero positivo. Se pide construir un calendario que permita que el torneo concluya en n-1 días.

#### Diseñar un método básico (no Divide y Vencerás) que resuelva el problema. Estudiar su eficiencia teórica.

La solución propuesta para este ejercicio consiste en crear un vector de vectores (del tipo Partido, que es un struct que hemos creado), donde se almacenará para cada día de torneo, una lista de partidos que se disputarán entre jugadores.   
  
 El pseudocódigo de nuestra solución propuesta es el siguiente:

Estructura Partido

jugador1

jugador2

end

Función organizaCampeonato(n)

Inicializar calendario como un vector de vectores de tamaño n-1

Para cada i desde 0 hasta n:

Para cada j desde i+1 hasta n:

Crear un partido con jugador1 = i y jugador2 = j

Añadir el partido al calendario en el día[j - i - 1]

end

end

end  
  
 En el main se calculará la n, que recordamos que **n = 2^k,** y se la pasamos como argumento a nuestra función organizaCampeonato. Debemos de tener en cuenta que nosotros le pasamos **k** como parámetro al ejecutar el programa y no **n.**

La eficiencia teórica de este algoritmo será **O(n²)** donde **n** es el número de participantes en el campeonato**.** La razón de esto es que hay dos bucles “for" anidados los cuáles podemos ver como:

* El primer bucle (el externo) va desde i = 0 hasta n, recorriendo así todos los jugadores.
* El segundo bucle (el interno) va desde j = i + 1, que al principio j será 1, hasta n, generando un partido para cada uno de los jugadores restantes.

Podemos obtener como conclusión que tendremos que hacer n\*n iteraciones, y por tanto tendremos una **eficiencia teórica de O(n²).**

#### Estudiar como el problema puede ser abordado mediante la técnica Divide y Vencerás y realizar el diseño completo. Estudiar su eficiencia teórica.

La solución al problema mediante un algoritmo Divide y Vencerás puede ser la siguiente.

En primer lugar, se divide el conjunto de jugadores obteniendo dos subconjuntos de igual tamaño, jugadores1 y jugadores2.

Después se llama recursivamente a sí mismo para los dos subconjuntos de jugadores y para los días en la primera mitad y la segunda mitad del campeonato. Esta finalizará una vez que el tamaño del subconjunto es 1 (es decir, solo hay un jugador), no se necesita hacer nada más, por lo que la función retornará. Esto es condicional que aparece al principio de la función.

Por último encontramos los dos bucles “for” anidados, en los que tanto i como j irán desde 0 hasta n/2. Esta parte es la encargada de coger un jugador de la primera mitad y otro de la segunda, creando un partido entre ellos y añadiéndolo al calendario.  
  
 El pseudocódigo de este algoritmo sería el siguiente:  
  
Estructura Partido

jugador1

jugador2

end

Función organizaCampeonatoDyV(jugadores, diaInicio, calendario)

n = numero de jugadores

Si n == 1

Devolver

jugadores1 = primera mitad de n/2 de jugadores

jugadores2 = segunda mitad de n/2 de jugadores

organizaCampeonatoDyV(jugadores1, diaInicio, calendario)

organizaCampeonatoDyV(jugadores2, diaInicio + n/2, calendario)

Para cada i desde 0 hasta n/2:

Para cada j desde 0 hasta n/2:

Crear un partido con jugador1 = i y jugador2 = (j+i) % (n/2)

Añadir el partido al calendario en el día[diaInicio + i]

end

end

end

La eficiencia teórica de este algoritmo será **O(nlog(n))** donde **n** es el número de participantes en el campeonato**.** Esto es debido a que el algoritmo divide el problema en dos subproblemas de igual tamaño (la mitad del tamaño del problema original), los resuelve de manera recursiva y luego combina las soluciones en tiempo lineal.

La eficiencia del algoritmo se puede describir con la ecuación de recurrencia

**T(n)=2T(2n​)+O(n)**

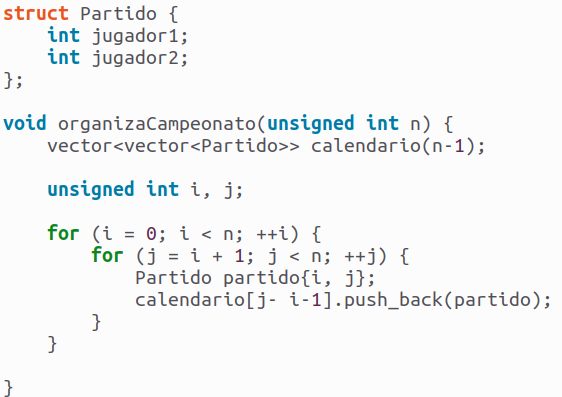
Donde, **T(n)** es el tiempo de ejecución del algoritmo,

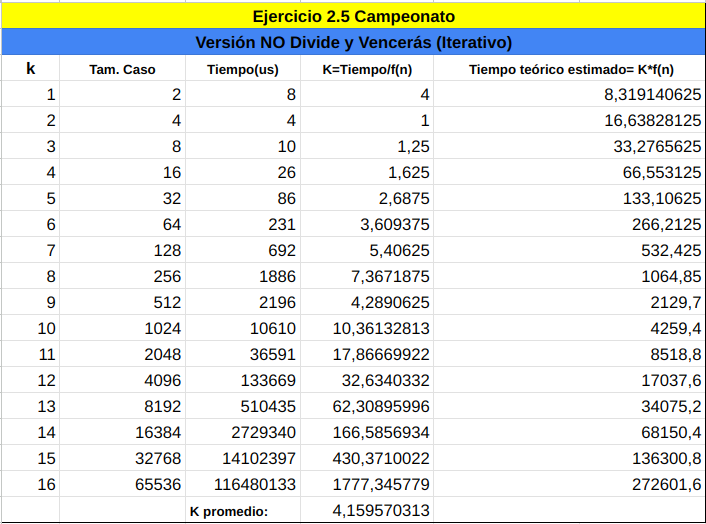
**2T(2n​)** es el tiempo para resolver los dos subproblemas

**O(n)** es el tiempo para combinar las soluciones de los subproblemas.

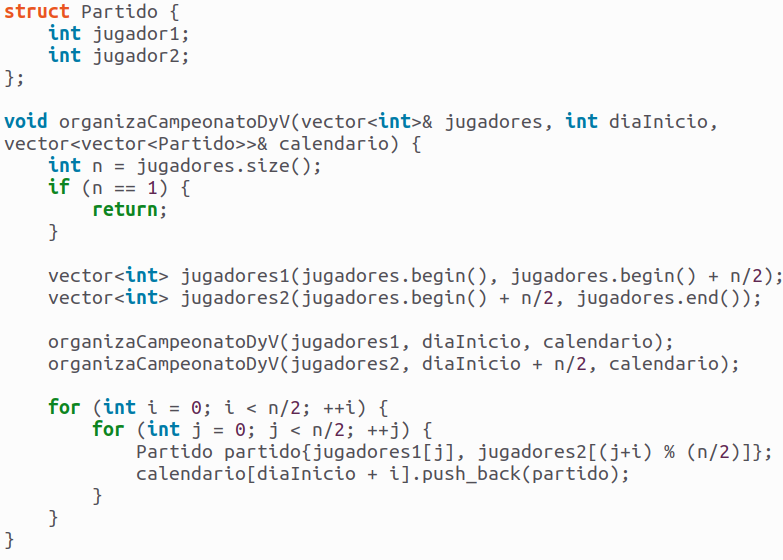
Según el teorema maestro para ecuaciones de recurrencia, esta ecuación de recurrencia tiene una solución de **O(nlogn)**. Esto nos deja que el tiempo de ejecución del algoritmo crece logarítmicamente con el tamaño del problema, lo que es significativamente más eficiente que un algoritmo de fuerza bruta que podría tener una eficiencia de O(n2), como es el caso del algoritmo iterativo visto en el apartado anterior.

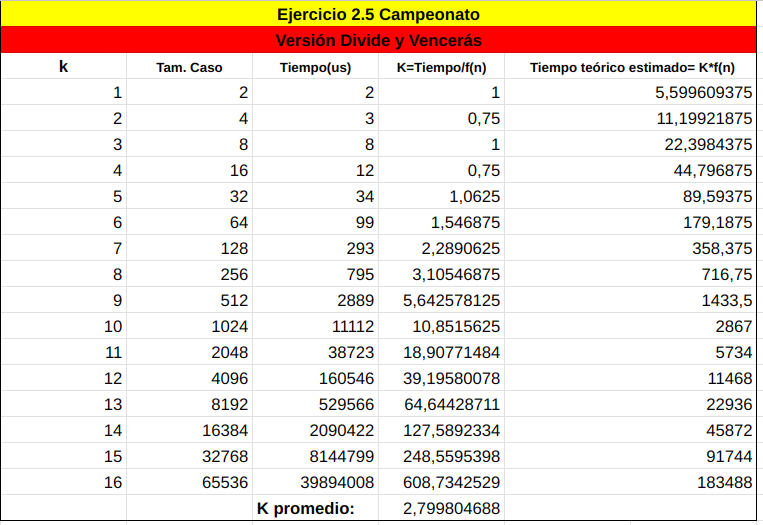
#### Implementar los algoritmos básico y Divide y Vencerás. Resolver el problema del umbral de forma experimental.

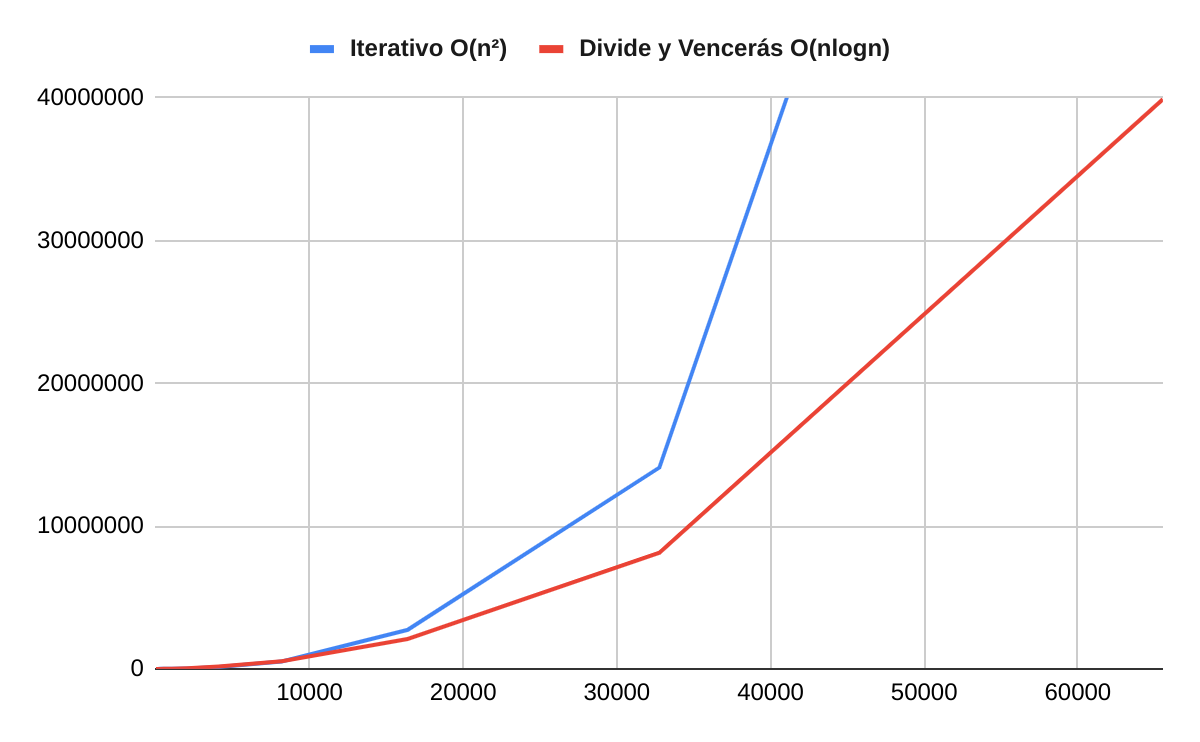
Una implementación para el algoritmo 1 (iterativo) podría ser la siguiente:  
 Se trata de la implementación del pseudocódigo del apartado a) del algoritmo iterativo.  
  
 Para realizar la comparación gráfica entre los dos algoritmos se han realizado unas mediciones experimentales que han dado los siguientes resultados en el caso del algoritmo iterativo:  




La implementación para el algoritmo 2 (divide y vencerás) es la siguiente:



De la que hemos obtenido la siguiente tabla:  
 

Comparando los Tiempos en microsegundos de cada uno de los algoritmos, obtendremos esta gráfica:  
 Como se puede observar, por debajo de n=10000 (participantes del campeonato), los algoritmos son idénticos. Pero es a partir de ahí cuando a medida que aumenta el tamaño del caso, el algoritmo Iterativo comienza a ser mucho más ineficiente que el algoritmo Divide y Vencerás, eso es debido a que crecen de forma cuadrática y logarítmica respectivamente. Esto nos muestra porque los algoritmos Divide y Vencerás suelen ser más eficientes para tamaños de casos relativamente grandes con respecto a otros como lo son los iterativos o fuerza bruta.

# **6. Ejemplos de compilación y ejecución.**

En este apartado explicamos cómo compilar y ejecutar cada .cpp de cada ejercicio para mostrar como lo hemos realizado nosotros, y obtener los tiempos de ejecución que aparecen en las tablas del apartado c).

**6.1. La mayoría absoluta.**

-compilación:

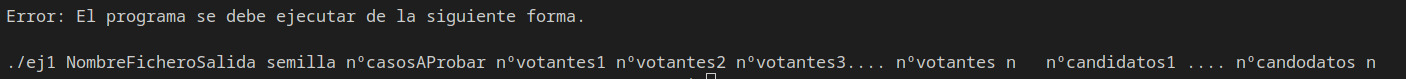
g++ ej1.cpp -o ej1

g++ ej1\_dyv.cpp -o ej1\_dyv

-ejemplos de prueba:

./ej1 tiempos\_ej1.txt 12345 10 100000 200000 300000 400000 500000 600000 700000 800000 900000 1000000 12 12 12 12 12 12 12 12 12 12

./ej1\_dyv tiempos\_ej1\_dyv.txt 12345 10 100000 200000 300000 400000 500000 600000 700000 800000 900000 1000000 12 12 12 12 12 12 12 12 12 12



**6.2. Tuercas y tornillos.**

VERSIÓN ITERATIVA->

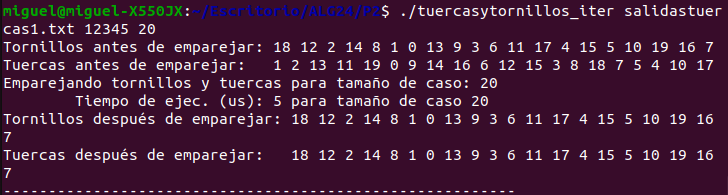
-compilación:

g++ -std=c++11 tuercasytornillos\_iter.cpp -o tuercasytornillos\_iter

-ejemplo de prueba utilizado en graficas:

./tuercasytornillos\_iter salidastuercas\_iter.txt 12345 500 1000 1500 2000 2500 3000 3500 4000 4500 5000

-ejemplo de prueba de funcionamiento correcto:



VERSIÓN DYV->

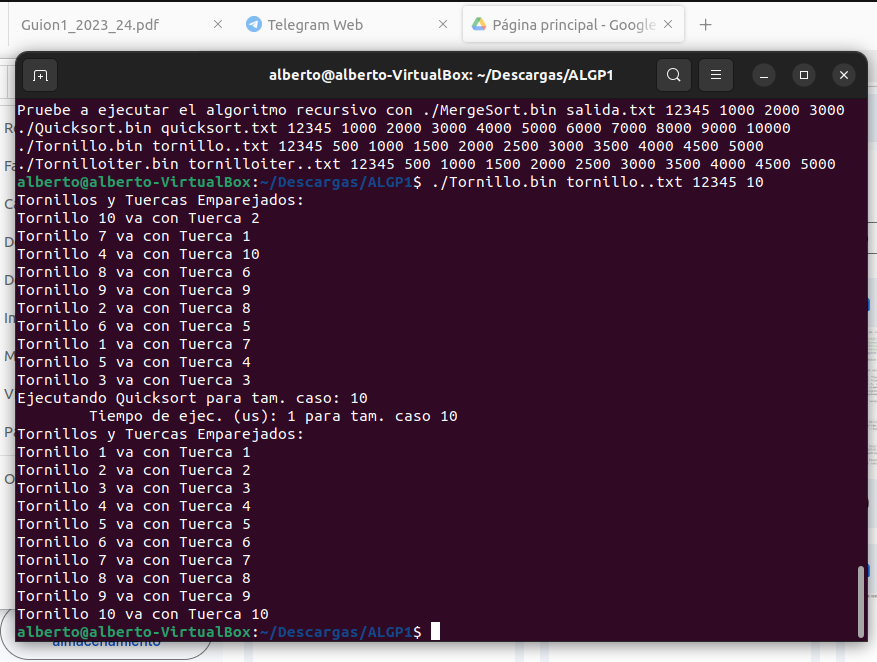
-compilación:

g++ -std=c++11 tornillos.cpp -o tornillos

-ejemplo de prueba utilizado en graficas:

./tornillos tornillosdyv.txt 12345 500 1000 1500 2000 2500 3000 3500 4000 4500 5000

-ejemplo de prueba de funcionamiento correcto:



**6.3. Producto de tres elementos.**

VERSIÓN ITERATIVA->

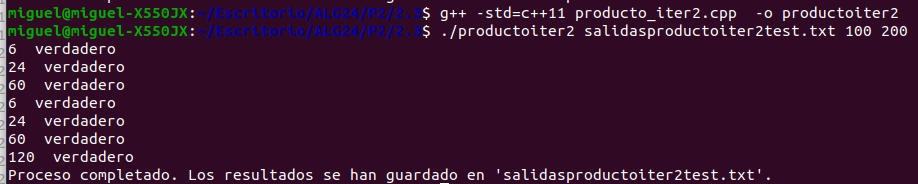
-compilación:

g++ -std=c++11 producto\_iter2.cpp -o productoiter2

-ejemplo de prueba utilizado en graficas:

./productoiter2 salidasproductoiter2.txt 500 1000 1500 2000 2500 3000 3500 4000 4500 5000

-ejemplo de prueba de funcionamiento correcto:



VERSIÓN DYV->

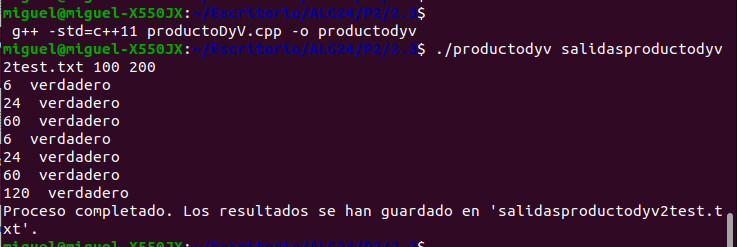
-compilación:

g++ -std=c++11 productoDyV.cpp -o productodyv

-ejemplo de prueba utilizado en graficas:

./productodyv salidasproductodyv2.txt 500 1000 1500 2000 2500 3000 3500 4000 4500 5000

-ejemplo de prueba de funcionamiento correcto:



**6.4. Eliminar elementos repetidos.**

-compilación:

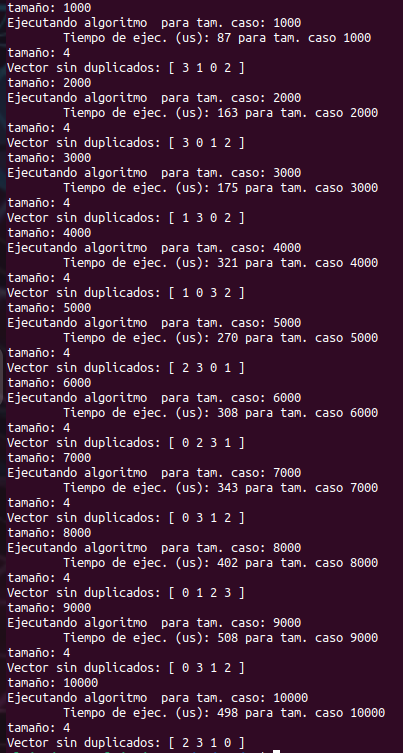
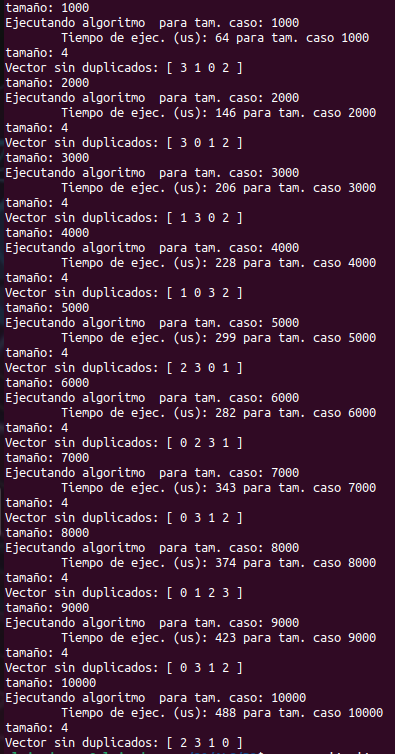
g++ -o ej4 ej4.cpp

g++ -o ej4\_dyv ej4\_dyv.cpp

-ejemplos de prueba:

./ej4 salida\_ej4.txt 12345 1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

./ej4\_dyv salida\_ej4\_dyv.txt 12345 1000 2000 3000 4000 5000 6000 7000 8000 9000 10000



**6.5. Organización del calendario de un campeonato**

Para este ejercicio, cómo ya se indicó en los apartados anteriores, le pasamos el parámetro **k,** mediante el cuál calcula **n** como **n = 2^k.** Esto es importante tenerlo en cuenta, y por eso a continuación mostraré los valores de k que tendremos que introducir para que se ejecute el algoritmo para el correspondiente tamaño de caso **n.**  
  
k = 1, n = 2 k = 2, n = 4 k = 3, n = 8 k = 4, n = 16

k = 5, n = 32 k = 6, n = 64 k = 7, n = 128 k = 8, n = 256

k = 9, n = 512 k = 10, n = 1024 k = 11, n = 2048 k = 12, n = 4096

k = 13, n = 8192 k = 14, n = 16384 k = 15, n = 32768 k = 16, n = 65536

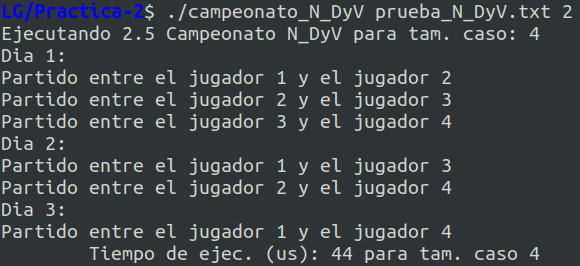
**VERSIÓN ITERATIVA->**

- compilación:

g++ -std=c++11 campeonato\_N\_DyV.cpp -o campeonato\_N\_DyV

- ejecución:  
./campeonato\_N\_DyV NombreFicheroSalida tamanio\_k\_1 tamanio\_k\_2 ... tamanio\_k\_n

-ejemplo de prueba de ejecución:

./campeonato\_N\_DyV prueba\_N\_DyV.txt 2

**VERSIÓN DyV->**

- compilación:

g++ -std=c++11 campeonato\_DyV.cpp -o campeonato\_DyV

- ejecución:  
./campeonato\_DyV NombreFicheroSalida tamanio\_k\_1 tamanio\_k\_2 ... tamanio\_k\_n

-ejemplo de prueba de ejecución:

./campeonato\_DyV prueba\_DyV.txt 2