

GRADO EN INGENIERÍA INFORMÁTICA

CURSO 2023-2024



UNIVERSIDAD DE GRANADA

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
1. La mayoría absoluta	2
a. Diseñar un método básico (no Divide y Vencerás) que resuelva el problema. Estudiar su eficiencia teórica.	2
b. Estudiar como el problema puede ser abordado mediante la técnica Divide y Vencerás y realizar el diseño completo. Estudiar su eficiencia teórica.	2
c. Implementar los algoritmos básico y Divide y Vencerás. Resolver el problema del umbral de forma experimental.	3
2. Tuercas y tornillos.	7
a. Diseñar un método básico (no Divide y Vencerás) que resuelva el problema. Estudiar su eficiencia teórica.	7
b. Estudiar cómo el problema puede ser abordado mediante la técnica Divide y Vencerás y realizar el diseño completo. Estudiar su eficiencia teórica.	8
c. Implementar los algoritmos básico y Divide y Vencerás. Resolver el problema del umbral de forma experimental.	8
3. Producto de tres elementos	10
a. Diseñar un método básico (no Divide y Vencerás) que resuelva el problema. Estudiar su eficiencia teórica.	10
b. Estudiar cómo el problema puede ser abordado mediante la técnica Divide y Vencerás y realizar el diseño completo. Estudiar su eficiencia teórica.	11
c. Implementar los algoritmos básico y Divide y Vencerás. Resolver el problema del umbral de forma experimental.	13
4. Eliminar elementos repetidos	15
a. Diseñar un método básico (no Divide y Vencerás) que resuelva el problema. Estudiar su eficiencia teórica.	15
b. Estudiar cómo el problema puede ser abordado mediante la técnica Divide y Vencerás y realizar el diseño completo. Estudiar su eficiencia teórica.	16
c. Implementar los algoritmos básico y Divide y Vencerás. Resolver el problema del umbral de forma experimental.	17
5. Organización del calendario de un campeonato.	20
d. Diseñar un método básico (no Divide y Vencerás) que resuelva el problema. Estudiar su eficiencia teórica.	20
e. Estudiar como el problema puede ser abordado mediante la técnica Divide y Vencerás y realizar el diseño completo. Estudiar su eficiencia teórica.	21
f. Implementar los algoritmos básico y Divide y Vencerás. Resolver el problema del umbral de forma experimental.	22
6. Ejemplos de compilación y ejecución.	25

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: $\text{Card}\{i \mid v[i]=x\} > n/2$. 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:

- a. 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 incrementa 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 las 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 a recorrer todo el vector de tamaño de n por cada iteración.

- b. 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 $O(n \cdot \log(n))$ ya que contar votos es $O(n)$ y mayoría_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

c. 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:**

```
bool mayoria_absoluta_iter(int v[],int n,int k){
    int cont[k]={};
    int mas_votado=0;
    bool mayoria_absoluta=false;

    for(int i=0;i<k;i++){
        for(int j=0;j<n;j++){
            if(v[j]==i){
                cont[i]++;
            }
        }
        if(cont[i]>=cont[mas_votado] && i!=0){
            mas_votado=i;
        }
    }
    if(cont[mas_votado]>n/2){
        mayoria_absoluta=true;
    }
}
```

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

Votos VERSION ITERATIVA para 4 candidatos				
Tam. Caso	Tiempo (us)	K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)	
100000	807	0,0000000807	231,2731586	
200000	1550	0,00000003875	925,0926342	O(N ²)
300000	2508	0,000000027866666	2081,458427	
400000	3052	0,000000019075	3700,370537	
500000	4001	0,000000016004	5781,828964	
600000	4322	0,000000012005555	8325,833708	
700000	5297	0,000000010810204	11332,38477	
800000	5997	0,000000009370312	14801,48215	
900000	7117	0,000000008786419	18733,12584	
1000000	7905	0,000000007905	23127,31586	
	K promedio:	0,000000023127315		

- Divide y Vencerás:

```
int contarVotos(int v[], int candidato, int inicio, int fin) {
    int conteo = 0;
    for (int i = inicio; i <= fin; ++i) {
        if (v[i] == candidato) {
            conteo++;
        }
    }
    return conteo;
}

int mayoria_absoluta_dyv(int v[], int n, int inicio, int fin) {
    if (inicio == fin){
        return v[inicio];
    }

    // Dividir el rango en dos partes
    int mitad = (inicio + fin) / 2;

    int mayoriaIzquierda = mayoria_absoluta_dyv(v, n, inicio, mitad);
    int mayoriaDerecha = mayoria_absoluta_dyv(v, n, mitad + 1, fin);

    // Combinar las soluciones de ambas partes
    if (mayoriaIzquierda == mayoriaDerecha && mayoriaDerecha != -1) {
        return mayoriaDerecha;
    }

    else{
        if (mayoriaIzquierda != -1) {
            if ( contarVotos(v, mayoriaIzquierda, inicio, mitad)+contarVotos(v, mayoriaIzquierda, mitad + 1, fin)> (mitad +1- inicio)) {
                return mayoriaIzquierda;
            }
        }

        if (mayoriaDerecha != -1) {
            if ( contarVotos(v, mayoriaDerecha, inicio, mitad)+contarVotos(v, mayoriaDerecha, mitad + 1, fin)> (fin- mitad+1)) {
                return mayoriaDerecha;
            }
        }
    }

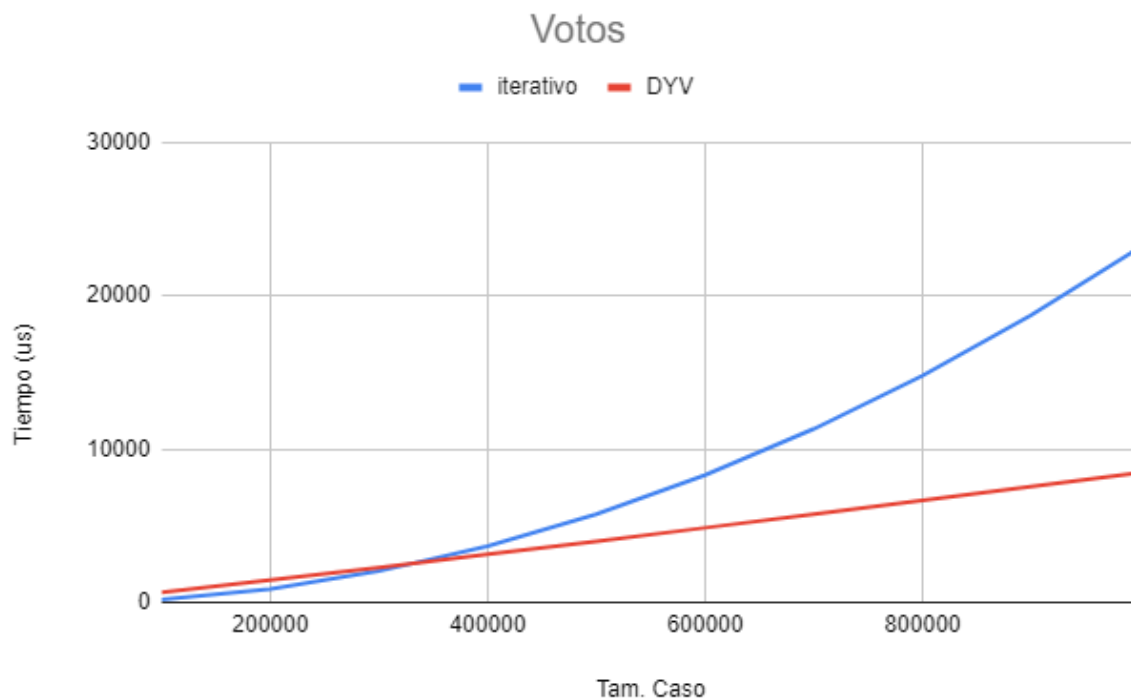
    return -1;
}

int encontrarMayoria(int votos[], int n) {
    return (mayoria_absoluta_dyv(votos, n, 0, n - 1) );
}
```

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

votos VERSION DIVIDE Y VENCERAS				
Tam. Caso	Tiempo (us)	$K = \text{Tiempo}/f(n)$	Tiempo teórico estimado= $K \cdot f(n)$	
100000	980	0,00196	706,9145684	
200000	1640	0,001546869195	1498,950132	
300000	2560	0,00155799606	2323,114085	$O(n \cdot \log(n))$
400000	2971	0,001325851564	3168,142257	
500000	3726	0,001307604707	4028,684921	
600000	4563	0,001316164924	4901,591156	
700000	5790	0,00141510519	5784,778941	
800000	5874	0,001243840093	6676,768496	
900000	6746	0,001258859636	7576,453392	
1000000	7236	0,001206	8482,97482	
K promedio:		0,001413829137		

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:

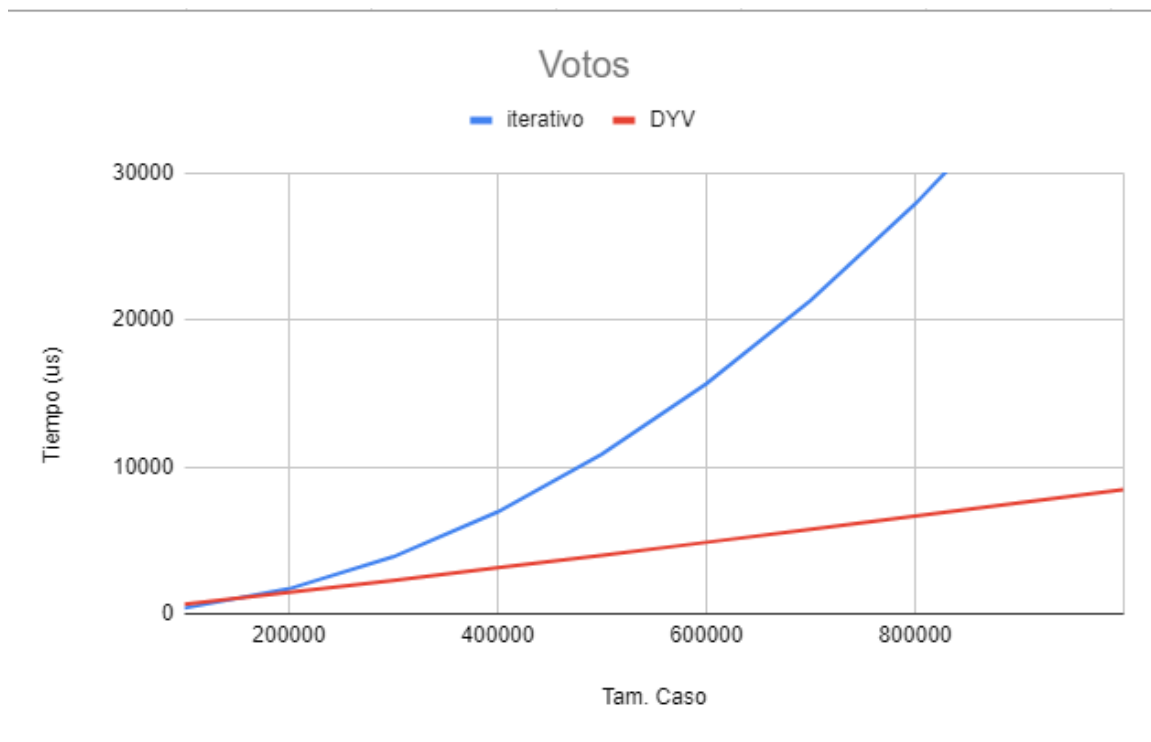


Se puede ver que a partir de $n=300000$ aproximadamente el algoritmo divide y vencerás es bastante mejor que el iterativo ya que está hecho para ser mejor en grandes casos. Además hay que tener en cuenta que si hubiera más candidatos afectaría en tiempos al iterativo pero no al divide y vencerás lo que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

Iterativo con 12 candidatos:

Tam. Caso	Tiempo (us)	K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)	
100000	1573	0,0000001573	435,285004	
200000	2932	0,0000000733	1741,140016	$O(N^2)$
300000	4420	0,0000000491111111	3917,565036	
400000	5861	0,00000003663125	6964,560064	
500000	7257	0,000000029028	10882,1251	
600000	8387	0,000000023297222	15670,26014	
700000	9847	0,000000020095918	21328,9652	
800000	10963	0,000000017129687	27858,24026	
900000	12567	0,000000015514814	35258,08533	
1000000	13877	0,000000013877	43528,5004	

(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 mínimo 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:

- a. 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 tornillostuerkas(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 recorreremos cada uno de ellos de la manera 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 sería $O(n^2)$ 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 $n \cdot n$ 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.

b. 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 y 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 $O(n^2)$ 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 esta eficiencia es algo no común, normalmente veremos una complejidad $O(n \log(n))$.

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

Una implementación para el método iterativo podría ser la siguiente:

```
void emparejarTornillosYTuercas(vector<int> &tornillos, vector<int> &tuercas, int n) {  
    for (int i = 0; i < n; i++) { //Recorrer vector tornillos  
        for (int j = 0; j < n; j++) { //recorrer vector tuercas  
            if (tornillos[i] == tuercas[j]) { //Si el componente del vector tornillos y del vector tuercas es igual  
                swap(tuercas[i], tuercas[j]); //intercambiar posiciones  
                break; // Cuando se ha encontrado la pareja que coincide se  
            }  
        }  
    }  
}
```

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:

Tam. Caso	Tiempo (us)	K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)	
500	451	0,001804	473,2664271	O(N ²)
1000	1502	0,001502	1893,065708	
1500	3114	0,001384	4259,397844	
2000	9499	0,00237475	7572,262833	
2500	16383	0,00262128	11831,66068	
3000	23931	0,002659	17037,59138	
3500	20967	0,001711591837	23190,05493	
4000	27750	0,001734375	30289,05133	
4500	31946	0,001577580247	38334,58059	
5000	39052	0,00156208	47326,64271	
	K promedio:	0,001893065708		

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

```
void matchPairs(int tornillos[], int tuercas[], int low, int high) {
    if (low < high) {
        // Seleccionamos un tornillo como pivote
        int pivot = tornillos[high];

        // Utilizamos el tornillo pivote para particionar las tuercas
        int index = partitionTuercas(tuercas, low, high, pivot);

        // Utilizamos la tuerca pivote para particionar los tornillos
        partition(tornillos, low, high, tuercas[index]);

        // Recursivamente emparejamos los tornillos y las tuercas en cada lado del pivote
        matchPairs(tornillos, tuercas, low, index - 1);
        matchPairs(tornillos, tuercas, index + 1, high);
    }
}
```

Se han obtenido los siguientes resultados: Siendo el tiempo teórico remarcado en azul estimado en $O(n^2)$ y en rojo $O(n\log(n))$.

Tam. Caso	Tiempo (us)	K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)		K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)
500	94	0,000376	39,57885393	O(N ²)	0,06965620207	135,958581
1000	195	0,000195	158,3154157		0,065	302,245436
1500	369	0,000164	356,2096854		0,07745369384	479,9795437
2000	502	0,0001255	633,2616629		0,07603687344	665,1475002
2500	621	0,00009936	989,4713483		0,07310311523	855,8432162
3000	761	0,00008455555556	1424,838741		0,07295306896	1050,94403
3500	4462	0,000364244898	1939,363843		0,3597157636	1249,708123
4000	1036	0,00006475	2533,046652		0,07190329995	1451,608257
4500	1128	0,0000557037037	3205,887168		0,06861540787	1656,250214
5000	1351	0,00005404	3957,885393		0,07304736175	1863,328003
	K promedio:	0,0001583154157			0,1007484787	



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 $1*2*3=6 \neq N$
4. Para $i=2$, $2*3*4=24 \neq N$
5. Para $i=3$, $3*4*5=60 \neq N$
6. Para $i=4$, $4*5*6=120 == N \rightarrow$ 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 $O(\sqrt[3]{n})$ 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.

- b. 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 $n = \log n$.

$$T(n) = c1 + \log(n)/\log(2)$$

$$T(n) \in O(\log(n))$$

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

Una implementación para el método iterativo sería la siguiente:

```
bool producto_tres_consecutivos(int N, int iteraciones) {
    int raiz_cubica = ceil(pow(N, 1.0/3.0));
    for (int i = 1; i <= raiz_cubica; ++i) {
        int producto = i * (i + 1) * (i + 2);
        if (producto == N) {
            return true;
        }
        if(producto >N){
            return false;
        }
    }
    return false;
}
```

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:

TIEMPOS PRODUCTO VERSION ITERATIVA				
Tam. Caso	Tiempo (us)	K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)	
500	51679	6511,145994	130171,3322	
1000	74200	7420	164005,6016	O(RAIZCUBICA(N))
1500	116135	10145,32673	187739,548	
2000	160935	12773,41941	206634,1097	
2500	207796	15310,54019	222589,847	
3000	302933	21004,20109	236537,0084	
3500	292848	19287,95782	249008,8005	
4000	343353	21629,88361	260342,6644	
4500	394203	23877,1463	270767,2824	
5000	445380	26045,9804	280445,6338	
K promedio:		16400,56016		

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:

```

bool es_producto_tres_consecutivos(int N, int a, int b, int c) {
    return a * b * c == N;
}

bool buscar_producto_tres_consecutivos(int N, int low, int high) {
    if (low > high) {
        return false;
    }

    int mid = low + (high - low) / 2;

    if (es_producto_tres_consecutivos(N, mid, mid + 1, mid + 2)) {
        return true;
    } else if (N < mid * (mid + 1) * (mid + 2)) {
        return buscar_producto_tres_consecutivos(N, low, mid - 1);
    } else {
        return buscar_producto_tres_consecutivos(N, mid + 1, high);
    }
}

```

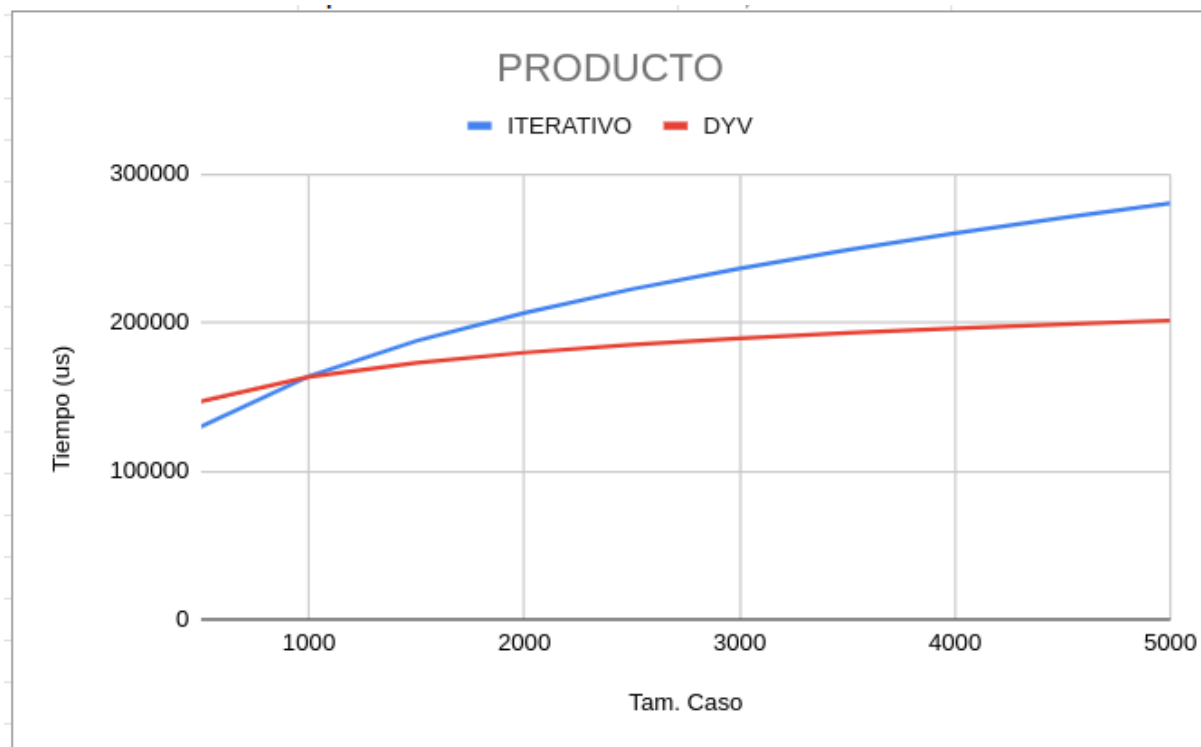
El caso base de este algoritmo viene determinado por ($low > high$) ya que si los índices por los que se está buscando son lo suficientemente pequeños que no se puedan dividir más o dichos índices se cruzan, no se ha encontrado solución.

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:

TIEMPOS PRODUCTO VERSION DIVIDE Y VENCERAS				
Tam. Caso	Tiempo (us)	K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)	
500	62784	7002,622192	147043,8845	
1000	117128	11753,01378	163444,4447	O(LOG N)
1500	172700	16368,5096	173038,1573	
2000	220203	20080,91663	179845,0048	
2500	254797	22572,95291	185124,8059	
3000	307247	26599,75201	189438,7175	
3500	359635	30547,07786	193086,0778	
4000	411911	34424,07035	196245,565	
4500	469664	38700,99299	199032,4302	
5000	528695	43026,31635	201525,366	
	K promedio:	25107,62247		

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 $O(\sqrt{n})$ 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 $O(n^2)$, 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.

b. 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 $O(n \cdot \log(n))$ 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.

- c. 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**

```
void quitar_duplicados ( vector<int>& v){
    for (int i=0; i < v.size(); i++){
        if(v[i]!=-1){
            for (int j=i+1; j < v.size(); j++){

                if (v[i] == v[j] ){
                    v[j]=-1;
                }
            }
        }
    }

    vector<int> aux;
    for(int i=0;i<v.size();i++){
        if(v[i]!=-1){
            aux.push_back(v[i]);
        }
    }
    v=aux;
}
```

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

Tam. Caso	Tiempo (us)	K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)	
1000	87	0,000087	21,11453549	O(n ²)
2000	163	0,00004075	84,45814198	
3000	175	0,00001944444444	190,0308194	
4000	321	0,0000200625	337,8325679	
5000	270	0,0000108	527,8633873	
6000	308	0,000008555555556	760,1232778	
7000	343	0,000007	1034,612239	
8000	402	0,00000628125	1351,330272	
9000	508	0,000006271604938	1710,277375	
10000	498	0,00000498	2111,453549	
K promedio:		0,00002111453549		

- Divide y vencerás

```
void quitar_duplicados_dyv(vector<int>& v) {
    int n = v.size();
    ordenar_merge(v, 0, n - 1);
    auto ini=v.begin();
    auto siguiente=ini+1;
    while(siguiente!=v.end()){
        if(*ini==*siguiente){
            siguiente=v.erase(siguiente);
        }else{
            ini=siguiente;
            ++siguiente;
        }
    }
}
```

```
void ordenar_merge(vector<int>& v, int inicio, int fin) {
    if (inicio < fin) {

        int mitad = inicio + (fin - inicio) / 2;

        ordenar_merge(v, inicio, mitad);
        ordenar_merge(v, mitad + 1, fin);

        combinar_subvectores(v, inicio, mitad, fin);
    }
}
```

```
void combinar_subvectores(vector<int>& v, int inicio, int mitad, int fin) {
    // Tamaño de los subvectores
    int n1 = mitad - inicio + 1;
    int n2 = fin - mitad;

    // Vectores temporales para almacenar los subvectores
    vector<int> der, izq;

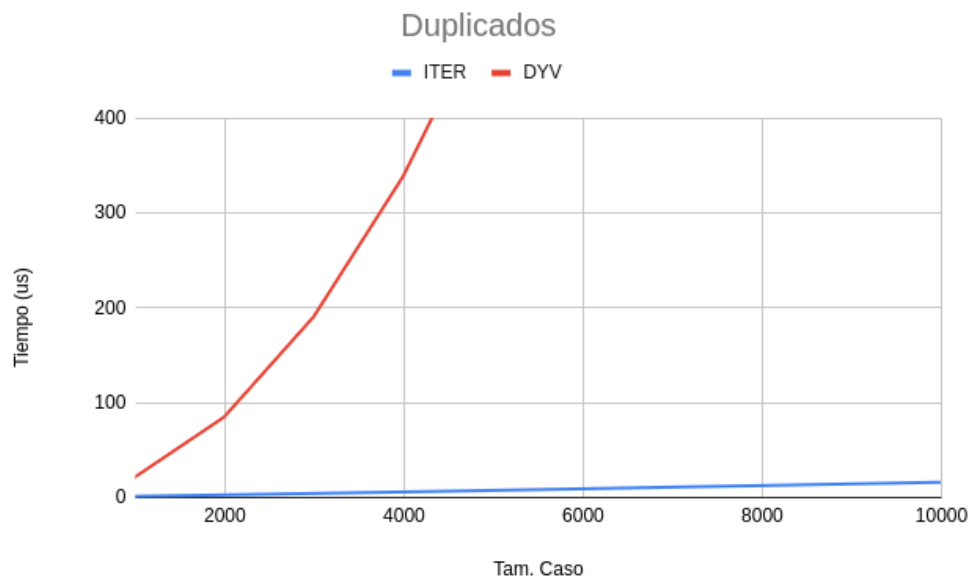
    for (int i = 0; i < n1; i++){
        izq.push_back( v[inicio + i]);
    }
    for (int j = 0; j < n2; j++){
        der.push_back( v[mitad + 1 + j]);
    }

    int i = 0, j = 0, k = inicio;
    while (i < n1 && j < n2) {
        if (izq[i] <= der[j]) {
            v[k] = izq[i];
            i++;
        } else {
            v[k] = der[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        v[k] = izq[i];
        i++;
        k++;
    }
    while (j < n2) {
        v[k] = der[j];
        j++;
        k++;
    }
}
```

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

Tam. Caso	Tiempo (us)	$K = \text{Tiempo}/f(n)$	Tiempo teórico estimado= $K*f(n)$	
1000	1327	0,001327	1,195533142	
2000	2415	0,00060375	2,630993841	
3000	3892	0,0004324444444	4,157013697	$O(n*\log(n))$
4000	5892	0,00036825	5,741842797	
5000	7266	0,00029064	7,37040205	
6000	9400	0,0002611111111	9,033810067	
7000	10161	0,0002073673469	10,72619833	
8000	11069	0,000172953125	12,44339582	
9000	13887	0,0001714444444	14,18228391	
10000	15015	0,00015015	15,94044189	
	K promedio:	0,0003985110472		

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.

d. 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 $O(n^2)$ 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 $n \cdot n$ iteraciones, y por tanto tendremos una **eficiencia teórica de $O(n^2)$** .

e. 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, dialInicio, 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, dialInicio, calendario)

organizaCampeonatoDyV(jugadores2, dialInicio + $n/2$, 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[dialInicio + i]

end

end

end

La eficiencia teórica de este algoritmo será **$O(n\log(n))$** 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 **$O(n\log n)$** . 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 $O(n^2)$, como es el caso del algoritmo iterativo visto en el apartado anterior.

f. 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:

```
struct Partido {
    int jugador1;
    int jugador2;
};

void organizaCampeonato(unsigned int n) {
    vector<vector<Partido>> calendario(n-1);

    unsigned int i, j;

    for (i = 0; i < n; ++i) {
        for (j = i + 1; j < n; ++j) {
            Partido partido{i, j};
            calendario[j- i-1].push_back(partido);
        }
    }
}
```

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:

Ejercicio 2.5 Campeonato				
Versión NO Divide y Vencerás (Iterativo)				
k	Tam. Caso	Tiempo(us)	$K = \text{Tiempo}/f(n)$	Tiempo teórico estimado= $K*f(n)$
1	2	8	4	8,319140625
2	4	4	1	16,63828125
3	8	10	1,25	33,2765625
4	16	26	1,625	66,553125
5	32	86	2,6875	133,10625
6	64	231	3,609375	266,2125
7	128	692	5,40625	532,425
8	256	1886	7,3671875	1064,85
9	512	2196	4,2890625	2129,7
10	1024	10610	10,36132813	4259,4
11	2048	36591	17,86669922	8518,8
12	4096	133669	32,6340332	17037,6
13	8192	510435	62,30895996	34075,2
14	16384	2729340	166,5856934	68150,4
15	32768	14102397	430,3710022	136300,8
16	65536	116480133	1777,345779	272601,6
		K promedio:	4,159570313	

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

```

struct Partido {
    int jugador1;
    int jugador2;
};

void organizaCampeonatoDyV(vector<int>& jugadores, int diaInicio,
vector<vector<Partido>>& calendario) {
    int n = jugadores.size();
    if (n == 1) {
        return;
    }

    vector<int> jugadores1(jugadores.begin(), jugadores.begin() + n/2);
    vector<int> jugadores2(jugadores.begin() + n/2, jugadores.end());

    organizaCampeonatoDyV(jugadores1, diaInicio, calendario);
    organizaCampeonatoDyV(jugadores2, diaInicio + n/2, calendario);

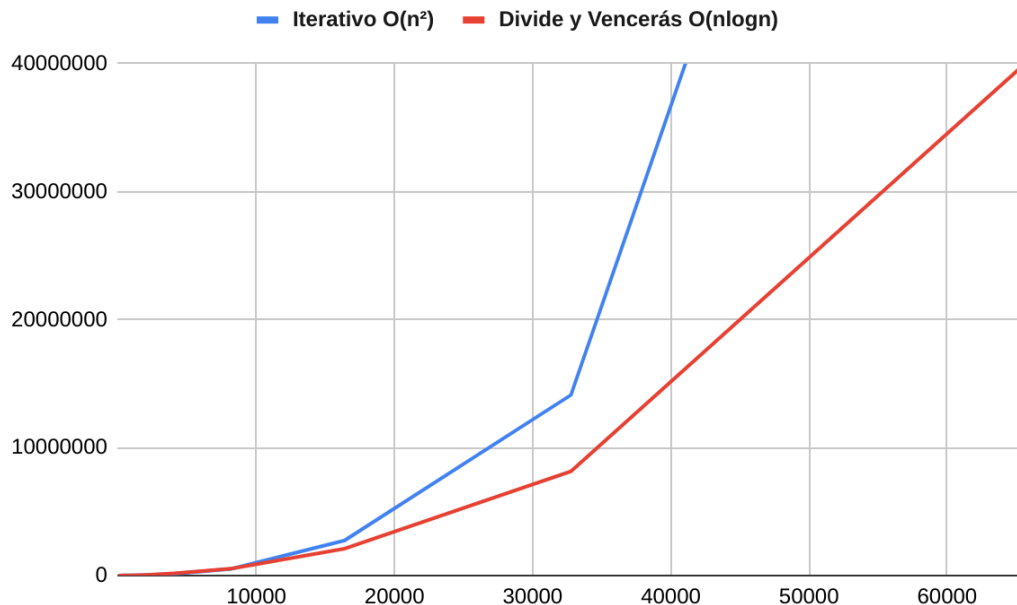
    for (int i = 0; i < n/2; ++i) {
        for (int j = 0; j < n/2; ++j) {
            Partido partido{jugadores1[j], jugadores2[(j+i) % (n/2)]};
            calendario[diaInicio + i].push_back(partido);
        }
    }
}

```


De la que hemos obtenido la siguiente tabla:

Ejercicio 2.5 Campeonato					
Versión Divide y Vencerás					
k	Tam. Caso	Tiempo(us)	$K = \text{Tiempo} / f(n)$	Tiempo teórico estimado= $K * f(n)$	
1	2	2	1	5,599609375	
2	4	3	0,75	11,19921875	
3	8	8	1	22,3984375	
4	16	12	0,75	44,796875	
5	32	34	1,0625	89,59375	
6	64	99	1,546875	179,1875	
7	128	293	2,2890625	358,375	
8	256	795	3,10546875	716,75	
9	512	2889	5,642578125	1433,5	
10	1024	11112	10,8515625	2867	
11	2048	38723	18,90771484	5734	
12	4096	160546	39,19580078	11468	
13	8192	529566	64,64428711	22936	
14	16384	2090422	127,5892334	45872	
15	32768	8144799	248,5595398	91744	
16	65536	39894008	608,7342529	183488	
K promedio:			2,799804688		

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
```

```
Error: El programa se debe ejecutar de la siguiente forma.
```

```
./ej1 NombreFicheroSalida semilla n°casosAProbar n°votantes1 n°votantes2 n°votantes3.... n°votantes n n°candidatos1 .... n°candidatos n
```

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:

```
miguel@miguel-X550JX:~/Escritorio/ALG24/P2$ ./tuercasytornillos_iter salidastuer  
cas1.txt 12345 20  
Tornillos antes de emparejar: 18 12 2 14 8 1 0 13 9 3 6 11 17 4 15 5 10 19 16 7  
Tuercas antes de emparejar: 1 2 13 11 19 0 9 14 16 6 12 15 3 8 18 7 5 4 10 17  
Emparejando tornillos y tuercas para tamaño de caso: 20  
Tiempo de ejec. (us): 5 para tamaño de caso 20  
Tornillos después de emparejar: 18 12 2 14 8 1 0 13 9 3 6 11 17 4 15 5 10 19 16  
7  
Tuercas después de emparejar: 18 12 2 14 8 1 0 13 9 3 6 11 17 4 15 5 10 19 16  
7  
-----
```

VERSIÓN DYV->

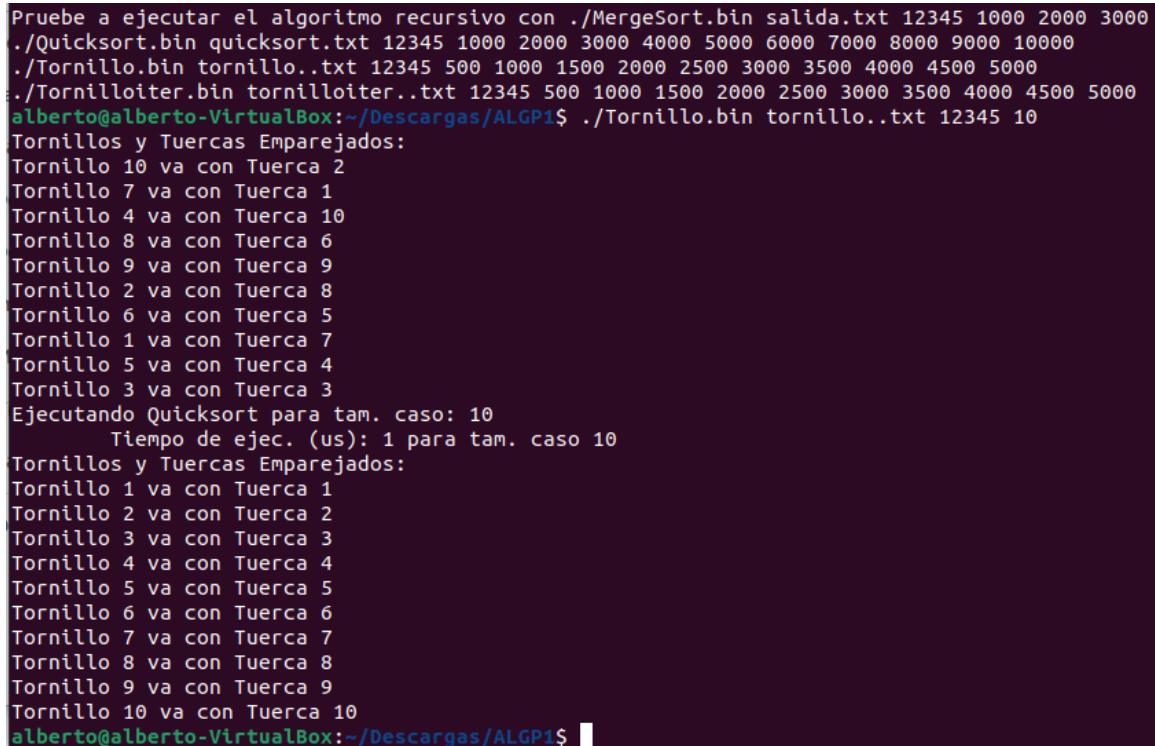
-compilación:

```
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:



```
Pruebe a ejecutar el algoritmo recursivo con ./MergeSort.bin salida.txt 12345 1000 2000 3000
./Quicksort.bin quicksort.txt 12345 1000 2000 3000 4000 5000 6000 7000 8000 9000 10000
./Tornillo.bin tornillo.txt 12345 500 1000 1500 2000 2500 3000 3500 4000 4500 5000
./Tornilloiter.bin tornilloiter.txt 12345 500 1000 1500 2000 2500 3000 3500 4000 4500 5000
alberto@alberto-VirtualBox:~/Descargas/ALGP1$ ./Tornillo.bin tornillo.txt 12345 10
Tornillos y Tuercas Emparejados:
Tornillo 10 va con Tuerca 2
Tornillo 7 va con Tuerca 1
Tornillo 4 va con Tuerca 10
Tornillo 8 va con Tuerca 6
Tornillo 9 va con Tuerca 9
Tornillo 2 va con Tuerca 8
Tornillo 6 va con Tuerca 5
Tornillo 1 va con Tuerca 7
Tornillo 5 va con Tuerca 4
Tornillo 3 va con Tuerca 3
Ejecutando Quicksort para tam. caso: 10
Tiempo de ejec. (us): 1 para tam. caso 10
Tornillos y Tuercas Emparejados:
Tornillo 1 va con Tuerca 1
Tornillo 2 va con Tuerca 2
Tornillo 3 va con Tuerca 3
Tornillo 4 va con Tuerca 4
Tornillo 5 va con Tuerca 5
Tornillo 6 va con Tuerca 6
Tornillo 7 va con Tuerca 7
Tornillo 8 va con Tuerca 8
Tornillo 9 va con Tuerca 9
Tornillo 10 va con Tuerca 10
alberto@alberto-VirtualBox:~/Descargas/ALGP1$
```

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:

```

miguel@miguel-X550JX:~/Escritorio/ALG24/P2/2.3$ g++ -std=c++11 producto_iter2.cpp -o productoiter2
miguel@miguel-X550JX:~/Escritorio/ALG24/P2/2.3$ ./productoiter2 salidasproductoiter2test.txt 100 200
6 verdadero
24 verdadero
60 verdadero
6 verdadero
24 verdadero
60 verdadero
120 verdadero
Proceso completado. Los resultados se han guardado en 'salidasproductoiter2test.txt'.

```

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:

```

miguel@miguel-X550JX:~/Escritorio/ALG24/P2/2.3$
g++ -std=c++11 productoDyV.cpp -o productodyv
miguel@miguel-X550JX:~/Escritorio/ALG24/P2/2.3$ ./productodyv salidasproductodyv
2test.txt 100 200
6 verdadero
24 verdadero
60 verdadero
6 verdadero
24 verdadero
60 verdadero
120 verdadero
Proceso completado. Los resultados se han guardado en 'salidasproductodyv2test.t
xt'.
miguel@miguel-X550JX:~/Escritorio/ALG24/P2/2.3$

```

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
```

```

tamaño: 1000
Ejecutando algoritmo para tam. caso: 1000
    Tiempo de ejec. (us): 64 para tam. caso 1000
tamaño: 4
Vector sin duplicados: [ 3 1 0 2 ]
tamaño: 2000
Ejecutando algoritmo para tam. caso: 2000
    Tiempo de ejec. (us): 146 para tam. caso 2000
tamaño: 4
Vector sin duplicados: [ 3 0 1 2 ]
tamaño: 3000
Ejecutando algoritmo para tam. caso: 3000
    Tiempo de ejec. (us): 206 para tam. caso 3000
tamaño: 4
Vector sin duplicados: [ 1 3 0 2 ]
tamaño: 4000
Ejecutando algoritmo para tam. caso: 4000
    Tiempo de ejec. (us): 228 para tam. caso 4000
tamaño: 4
Vector sin duplicados: [ 1 0 3 2 ]
tamaño: 5000
Ejecutando algoritmo para tam. caso: 5000
    Tiempo de ejec. (us): 299 para tam. caso 5000
tamaño: 4
Vector sin duplicados: [ 2 3 0 1 ]
tamaño: 6000
Ejecutando algoritmo para tam. caso: 6000
    Tiempo de ejec. (us): 282 para tam. caso 6000
tamaño: 4
Vector sin duplicados: [ 0 2 3 1 ]
tamaño: 7000
Ejecutando algoritmo para tam. caso: 7000
    Tiempo de ejec. (us): 343 para tam. caso 7000
tamaño: 4
Vector sin duplicados: [ 0 3 1 2 ]
tamaño: 8000
Ejecutando algoritmo para tam. caso: 8000
    Tiempo de ejec. (us): 374 para tam. caso 8000
tamaño: 4
Vector sin duplicados: [ 0 1 2 3 ]
tamaño: 9000
Ejecutando algoritmo para tam. caso: 9000
    Tiempo de ejec. (us): 423 para tam. caso 9000
tamaño: 4
Vector sin duplicados: [ 0 3 1 2 ]
tamaño: 10000
Ejecutando algoritmo para tam. caso: 10000
    Tiempo de ejec. (us): 488 para tam. caso 10000
tamaño: 4
Vector sin duplicados: [ 2 3 1 0 ]

```

```

tamaño: 1000
Ejecutando algoritmo para tam. caso: 1000
    Tiempo de ejec. (us): 87 para tam. caso 1000
tamaño: 4
Vector sin duplicados: [ 3 1 0 2 ]
tamaño: 2000
Ejecutando algoritmo para tam. caso: 2000
    Tiempo de ejec. (us): 163 para tam. caso 2000
tamaño: 4
Vector sin duplicados: [ 3 0 1 2 ]
tamaño: 3000
Ejecutando algoritmo para tam. caso: 3000
    Tiempo de ejec. (us): 175 para tam. caso 3000
tamaño: 4
Vector sin duplicados: [ 1 3 0 2 ]
tamaño: 4000
Ejecutando algoritmo para tam. caso: 4000
    Tiempo de ejec. (us): 321 para tam. caso 4000
tamaño: 4
Vector sin duplicados: [ 1 0 3 2 ]
tamaño: 5000
Ejecutando algoritmo para tam. caso: 5000
    Tiempo de ejec. (us): 270 para tam. caso 5000
tamaño: 4
Vector sin duplicados: [ 2 3 0 1 ]
tamaño: 6000
Ejecutando algoritmo para tam. caso: 6000
    Tiempo de ejec. (us): 308 para tam. caso 6000
tamaño: 4
Vector sin duplicados: [ 0 2 3 1 ]
tamaño: 7000
Ejecutando algoritmo para tam. caso: 7000
    Tiempo de ejec. (us): 343 para tam. caso 7000
tamaño: 4
Vector sin duplicados: [ 0 3 1 2 ]
tamaño: 8000
Ejecutando algoritmo para tam. caso: 8000
    Tiempo de ejec. (us): 402 para tam. caso 8000
tamaño: 4
Vector sin duplicados: [ 0 1 2 3 ]
tamaño: 9000
Ejecutando algoritmo para tam. caso: 9000
    Tiempo de ejec. (us): 508 para tam. caso 9000
tamaño: 4
Vector sin duplicados: [ 0 3 1 2 ]
tamaño: 10000
Ejecutando algoritmo para tam. caso: 10000
    Tiempo de ejec. (us): 498 para tam. caso 10000
tamaño: 4
Vector sin duplicados: [ 2 3 1 0 ]

```

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 tamano_k_1 tamano_k_2 ... tamano_k_n
```

-ejemplo de prueba de ejecución:

```
./campeonato_N_DyV prueba_N_DyV.txt 2
```

```

lg/Practica-2$ ./campeonato_N_DyV prueba_N_DyV.txt 2
Ejecutando 2.5 Campeonato N_DyV para tam. caso: 4
Dia 1:
Partido entre el jugador 1 y el jugador 2
Partido entre el jugador 2 y el jugador 3
Partido entre el jugador 3 y el jugador 4
Dia 2:
Partido entre el jugador 1 y el jugador 3
Partido entre el jugador 2 y el jugador 4

```

VERSIÓN DyV->

- compilación:

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

- ejecución:

./campeonato_DyV NombreFicheroSalida tamano_k_1 tamano_k_2 ... tamano_k_n

-ejemplo de prueba de ejecución:

./campeonato_DyV prueba_DyV.txt 2

```
LG/Practica-2$ ./campeonato_DyV prueba_DyV.txt 2
Ejecutando 2.5 Campeonato DyV para tam. caso: 4
    Tiempo de ejec. (us): 15 para tam. caso 4
```