Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

Algoritmos y Estructuras de Datos

Tarea 3

Autor: Javier Gómez
Profesor: Benjamín Bustos
Auxiliares: Ignacio Valderrama
Manuel Olguín
Vanessa Peña Araya
Ayudantes: Fabián González
José Ignacio Moreno
Pablo Pizarro R.

Fecha de realización: 31 de octubre de 2016
Fecha de entrega: 1 de noviembre de 2016
Santiago, Chile

Índice de Contenidos

1. Introducción

A la hora de resolver problemas, buscamos utilizar programas que nos ayuden y nos hagan esta tarea más facil, pero estos programas alguien tiene que desarrollarlo.

Para un determinado problema, existen más de una manera de resolverlo. A cada una de estas formas la llamaremos "algoritmo". Un algoritmo diremos que es una serie de pasos determinados para resolver un problema. Para un computador, ejecutar estos distintos pasos puede tener diferente dificultad, por lo que comenzamos a hablar de "eficiencia del algoritmo". Un algoritmo eficiente intenta resolver el problema en la menor cantidad de pasos y operaciones posibles, para así dar la respuesta lo más rápido posible.

En este trabajo, analizaremos tres algoritmos distintos para resolver un mismo problema, su eficiencia y el tiempo que toman en realizar dicha tarea.

2. Diseño de la solución

Los algoritmos que utilizaremos en este trabajo serán los siguientes:

- Fuerza Bruta (mejorado)
- Dividir para reinar
- Algoritmo eficiente

La implementacion de estos algoritmos se realizó a partir de los casos de estudio vistos en clase (con una modificación ya que estos retornaban el valor de la suma y no los índices).

Se utilizará la herramienta *System.currentTimeMillis()* para medir el tiempo que toma cada algoritmo (y con esto, la eficiencia).

3. Implementación

A continuación, se analizará la implementación de cada uno de los algoritmos a estudiar.

3.1. Fuerza Bruta (mejorado)

Para el algoritmo de Fuerza Bruta (mejorado) se utilizó el siguiente método para un arreglo *"arry"* de largo n.

```
public static int[] FZBRUTA(int[] arry){
    int maxSum = 0;
    int d=0, u=0;
    int n = arry.length;
    for(int i=0; i<n; i++)
    {   int thisSum = 0;
        for (int j=i; j<=n-1; j++)
        {   thisSum += arry[j];
            if (thisSum > maxSum){
                maxSum = thisSum;
                d=i;
                u=j;}
            if (thisSum == maxSum){
                if(j-i < u-d) {
                    d=i;
```

```
                    u=j;}
            }
        }
    }
    int sec[] = new int[2];
    sec[0] = d;
    sec[1] = u;
    return sec;
}
System.out.println(d + "," + u);
//donde d es el indice inicial del sub arreglo y u el final
System.out.print(d+','+u);
```

En donde se utilizó el mismo código proporcionado en los casos de estudio, pero con la leve modificación de guardar los índices en los que se encuentra esta *subsecuencia*.

Analizando el código, se puede dar cuenta de que este algoritmo toma tiempo $O(n^2)$, ya que en cada posicion del arreglo de largo n, lo recorre una vez completamente, resultando $O(n^2)$.

3.2. Dividir para reinar

El algoritmo de dividir para reinar consiste en dividir el arreglo en subarreglos y así dividir los subarreglos en más subarreglos, para resolver el problema de manera recursiva, llegando a un caso base (que será cuando el subarreglo tenga largo 1).

Esto se implementó de la siguiente manera:

```
int L = sec.length;
        if(L==0)
            return new int[] {0,0,0};
        if(L==1)
            return new int[] {0,0,sec[0]};
        if(L==2)
            if(sec[0]>0 && sec[1]>0)
                return new int[] {0,1,sec[0]+sec[1]};
            else if(sec[0]>=sec[1])
                return new int[] {0,0,sec[0]};
            else
                return new int[] {1,1,sec[1]};

        int[] izq = Arrays.copyOfRange(sec, 0, L/2); //dividimos la secuencia
        int[] der = Arrays.copyOfRange(sec, L/2, L);
        int[] maxizq = submax(izq); //aplicamos la recursion para cada mitad
        int[] maxder = submax(der);
        int[] secmax = new int[] {0,0,-Integer.MAX_VALUE};
        int[] thissec = new int[] {maxizq[0],maxizq[0],0};

        for(int i = maxizq[0];i<maxder[1]+L/2+1;++i)
        {   thissec[1]=i;
            thissec[2]+=sec[i];
            if(thissec[2]>secmax[2] || (thissec[2]==secmax[2] &&
                thissec[1]-thissec[0]<secmax[1]-secmax[0]))
                secmax = new int[] {thissec[0],thissec[1],thissec[2]}; //actualizamos secmax

            if(thissec[2]<=0)
                thissec = new int[] {i+1,i,0}; //thissec la mandamos a 0
```

```
        }
        if(secmax[2]>maxizq[2] && secmax[2]> maxder[2])
            return secmax;

        else if(maxizq[2]>maxder[2])
            return maxizq;

        else if(maxder[2]>maxizq[2])
            return new int[] {maxder[0]+L/2,maxder[1]+L/2,maxder[2]};

        else if(maxder[2]==maxizq[2]) //debemos comparar el largo de las secuencias
        {   if(maxizq[1]-maxizq[0] > maxder[1]-maxder[0]) //secuencia derecha es mas corta
                return new int[] {maxder[0]+L/2,maxder[1]+L/2,maxder[2]};

            else //secuencia izq es mas corta
                return maxizq;
        }
        return maxizq; //esto arbitrario, no influye en la resolucion
```

El tiempo que toma este algoritmo es $T(n) = 2 * T(n/2) + O(n)$ y utilizando el teorema maestro en el caso de p=q, tenemos que el algoritmo toma tiempo $O(n \log(n))$

3.3. Algoritmo eficiente

La implementación se basó en el algoritmo que vimos en los casos de estudio en clases, pero se modificó para que devolviera los índices donde corresponde la subsecuencia máxima en lugar del valor de la suma máxima.

Para un arreglo *"arry"* de largo n, el método es el siguiente:

```
public static int[] ef(int[] arry){
    int maxSum = 0, thisSum = 0;
    int n = arry.length;
    int j, d=0, u=0, dg = 0, ug = 0;
    for( j=0; j<n; j++)
    {   u=j;
        thisSum += arry[j];
        if (thisSum > maxSum || thisSum==maxSum && u-d<ug-dg) {
            maxSum = thisSum;
            dg= d;
            ug= u;
        }
        else if (thisSum <= 0){
            thisSum = 0;
            d=j+1;
            u=j;}
    }
    int r[] = new int [2];
    r[0] = dg;
    r[1] = ug;
    return r;
}
//donde dg es el indice inicial del sub arreglo y ug el final
System.out.println(dg + "," + ug);
```

Como se puede dar cuenta al ver el código, este algoritmo toma tiempo $O(n)$, ya que sólo recorre el arreglo de largo n una vez.

4. Resultados y Conclusiones

Como se puede ver en la sección anterior, los algoritmos analizados toman distinto tiempo en realizar la misma tarea. Esto se debe a la forma en que cada uno piensa el problema, y con esto lo puede abordar de forma distinta, aprovechando la estructura del mismo para realizar el mínimo de operaciones necesarias.

En esta sección se analizará en profundidad el tiempo que toma cada algoritmo y lo se contrastará con los otros para poder ver la importancia de utilizar el algoritmo adecuado para cada problema.

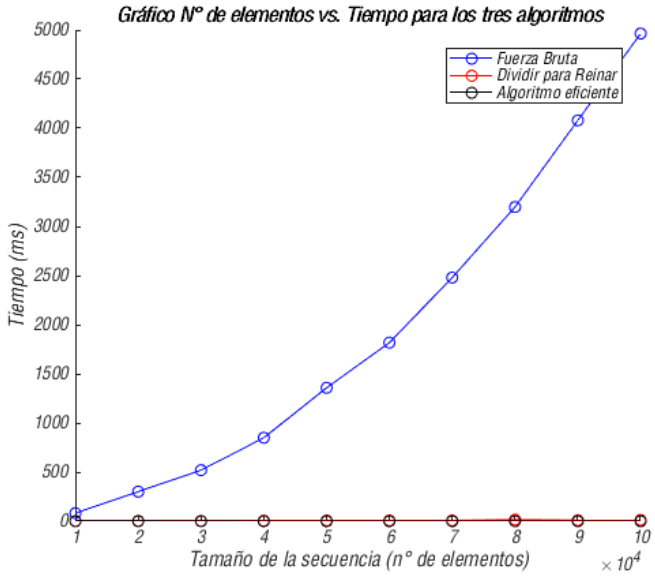
Para realizar la medición de la eficiencia de cada algoritmo, se creó un programa que calcula el tiempo (en ms) que toma cada uno en correr secuencias aleatorias de 10.000, 20.000, ... , 100.000 elementos. Este programa entrega los resultados en un archivo resultTime.txt ordenados (Ver Anexo 3).

Los resultados obtenidos fueron los siguientes:

Tamaño Secuencia	Fuerza Bruta	D-P-R	Alg. Eficiente
10000	82.2	3.0	0.3
20000	301.0	3.4	0.2
30000	519.7	3.6	0.2
40000	849.5	6.3	0.1
50000	1358.3	8.0	0.1
60000	1818.0	8.4	0.1
70000	2480.4	7.9	0.1
80000	3199.1	16.3	0.2
90000	4080.5	10.5	0.2
100000	4964.9	13.5	0.2

Tiempo medido en milisegundos (ms)

Como se puede ver, los distintos algoritmos generan diferencias de tiempo considerables (para las mismas secuencias). Aquí es donde se ve reflejada la importancia de la eficiencia de nuestro programa, ya que elegir correctamente el algoritmo a utilizar para un determinado problema puede significar varios segundos de diferencia, y en casos de procesar gran cantidad de información, pueden convertirse en minutos o incluso horas.



En este gráfico se puede observar las curvas (N° de elementos vs. Tiempo) de cada algoritmo. La curva de los algoritmos Dividir para reinar y Algoritmo eficiente se ven similares debido al escalado que se realizó para

poder observar bien la comparativa con el algoritmos Fuerza bruta. Para ver más detalladamente cada curva, ver Anexo 4.

En el gráfico queda claro que el algoritmo Fuerza bruta es mucho más lento que los otros dos, y que Dividir para reinar es más lento que el algoritmo eficiente (ver tabla en página 4). Esto obedece a los resultados teóricos vistos y obtenidos en clases.

Cabe destacar que en el algoritmo eficiente se produce una 'anomalía' ya que para las primeras secuencias (10.000 - 30.000 elementos) el tiempo que toma es considerablemente mayor que para las siguientes secuencias (Se produce una curva donde el tiempo decrece según crece la secuencia, para luego crecer junto con esta).

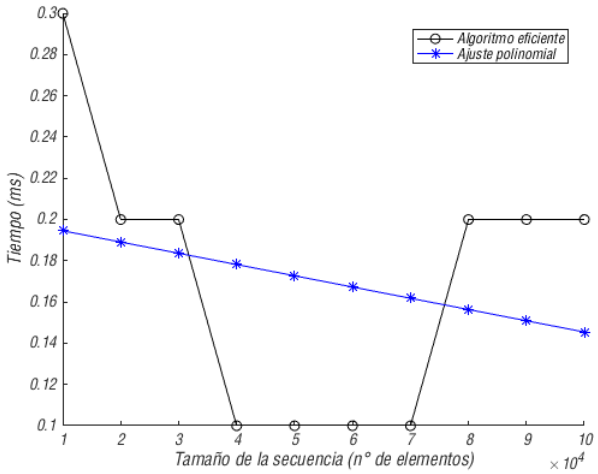


Gráfico N° de elementos vs. Tiempo para el algoritmo eficiente con ajuste polinomial

Esto se produce debido a la forma en que el procesador 'consulta' a la memoria de acceso aleatorio (RAM). Al iniciar un proceso toma más tiempo realizar las operaciones que cuando ya se ha iniciado el mismo (es decir, 'se demora más al principio').

Es necesario notar que, al ser secuencias generadas aleatoriamente, pueden aparecer secuencias 'más complejas' que otras en algunos algoritmos, donde esta tome más tiempo que una secuencia de mayor tamaño (Ver caso Algoritmo Dividir para reinar, la secuencia de 80.000 elementos toma más tiempo que la de 90.000. Ver anexo 4). Este algoritmo también cumple con los resultados esperados y vistos en clase.

En conclusión, los resultados obtenidos responden a los resultados esperados (análisis teórico en clases), lo que se puede comprobar viendo el ajuste polinomial en cada gráfico (Anexo 4). En el caso del algoritmo eficiente se produce una anomalía por lo explicado recientemente en esta sección.

5. Anexo 1: Código de las clases

5.1. Algoritmo Fuerza-Bruta (mejorado)

```
import java.util.Scanner;
public class fzabruta {
    public static int[] FZBRUTA(int[] arry){
        int maxSum = 0;
        int d=0, u=0;
        int n = arry.length;
        for(int i=0; i<n; i++)
        {
            int thisSum = 0;
            for (int j=i; j<=n-1; j++)
            {
                thisSum += arry[j];
                if (thisSum > maxSum){
                    maxSum = thisSum;
                    d=i;
                    u=j;}
                if (thisSum == maxSum){
                    if(j-i < u-d) {
                        d=i;
                        u=j;
                    }
                }
            }
        }
        int sec[] = new int[2];
        sec[0] = d;
        sec[1] = u;
        return sec;
    }

    public static void main(String [] args) {
        Scanner in = new Scanner(System.in);
        while(in.hasNextLine()) {
            String sec = in.nextLine();

            String[] secuencia = sec.split(" ");
            int n = secuencia.length;
            int arr[] = new int[n];
            for (int i=0; i<n; ++i){
                arr[i] = Integer.parseInt(secuencia[i]);
            }

            System.out.println(FZBRUTA(arr)[0] + "," + FZBRUTA(arr)[1]);

        }

    }
}
```

5.2. Algoritmo Dividir para reinar

```
import java.util.Arrays;
import java.util.Scanner;
public class dpr {

    public static int[] submax(int[] sec){
        int L = sec.length;
        if(L==0)
            return new int[] {0,0,0};
        if(L==1)
            return new int[] {0,0,sec[0]};
        if(L==2)
            if(sec[0]>0 && sec[1]>0)
                return new int[] {0,1,sec[0]+sec[1]};
            else if(sec[0]>=sec[1])
                return new int[] {0,0,sec[0]};
            else
                return new int[] {1,1,sec[1]};

        int[] izq = Arrays.copyOfRange(sec, 0, L/2); //dividimos la secuencia
        int[] der = Arrays.copyOfRange(sec, L/2, L);
        int[] maxizq = submax(izq); //aplicamos la recursion para cada mitad
        int[] maxder = submax(der);

        int[] secmax = new int[] {0,0,-Integer.MAX_VALUE};
        int[] thissec = new int[] {maxizq[0],maxizq[0],0};

        for(int i = maxizq[0];i<maxder[1]+L/2+1;++i)
        {   thissec[1]=i;
            thissec[2]+=sec[i];
            if(thissec[2]>secmax[2] || (thissec[2]==secmax[2] &&
                thissec[1]-thissec[0]<secmax[1]-secmax[0]))
                secmax = new int[] {thissec[0],thissec[1],thissec[2]}; //actualizamos secmax

            if(thissec[2]<=0)
                thissec = new int[] {i+1,i,0}; //thissec la mandamos a 0
        }

        if(secmax[2]>maxizq[2] && secmax[2]> maxder[2])
            return secmax;

        else if(maxizq[2]>maxder[2])
            return maxizq;

        else if(maxder[2]>maxizq[2])
            return new int[] {maxder[0]+L/2,maxder[1]+L/2,maxder[2]};

        else if(maxder[2]==maxizq[2]) //debemos comparar el largo de las secuencias
        {   if(maxizq[1]-maxizq[0] > maxder[1]-maxder[0]) //secuencia derecha es mas corta
                return new int[] {maxder[0]+L/2,maxder[1]+L/2,maxder[2]};

            else //secuencia izq es mas corta
                return maxizq;
        }
```

```
        return maxizq;
    }


    public static void main(String [] args) {
        Scanner in = new Scanner(System.in);
        while(in.hasNextLine()) {
            String sec = in.nextLine();

            String[] secuencia = sec.split(" ");
            int n = secuencia.length;
            int arr[] = new int[n];
            for (int i=0; i<n; ++i){
                arr[i] = Integer.parseInt(secuencia[i]);
            }



            int[] prueba;
            //prueba = maxsum(arr,0,n-1);
            prueba = submax(arr);


            System.out.println(prueba[0]+","+prueba[1]);


        }

    }
}
```

5.3. Algoritmo eficiente

```
import java.util.Scanner;
public class eficiente {
    public static int[] ef(int[] arry){
        int maxSum = 0, thisSum = 0;
        int n = arry.length;
        int j, d=0, u=0, dg = 0, ug = 0;
        for( j=0; j<n; j++)
        {
            u=j;
            thisSum += arry[j];
            if (thisSum > maxSum || thisSum==maxSum && u-d<ug-dg) {
                maxSum = thisSum;
                dg= d;
                ug= u;
            }

            else if (thisSum <= 0){
                thisSum = 0;
                d=j+1;
                u=j;
            }
```

```
        }
        int r[] = new int [2];
        r[0] = dg;
        r[1] = ug;
        return r;
    }

    public static void main(String [] args) {
        Scanner in = new Scanner(System.in);
        while(in.hasNextLine()) {
            String sec = in.nextLine();

            String[] secuencia = sec.split(" ");
            int n = secuencia.length;
            int arr[] = new int[n];
            for (int i=0; i<n; ++i){
                arr[i] = Integer.parseInt(secuencia[i]);
            }
            System.out.println(ef(arr)[0] + "," + ef(arr)[1]);
        }

    }
}
```

6. Anexo 2: Programa de medición de tiempo

```
import java.util.Arrays;
import java.util.Random;

public class tiempo {


    static int[] FZBRUTA(int[] arry){
        int maxSum = 0;
        int d=0, u=0;
        int n = arry.length;
        for(int i=0; i<n; i++)
        {
            int thisSum = 0;
            for (int j=i; j<=n-1; j++)
            {
                thisSum += arry[j];
                if (thisSum > maxSum){
                    maxSum = thisSum;
                    d=i;
                    u=j;}
                if (thisSum == maxSum){
                    if(j-i < u-d) {
                        d=i;
                        u=j;
                    }
                }
            }
        }
        int sec[] = new int[2];
        sec[0] = d;
        sec[1] = u;
        return sec;
    }

    static int[] submax(int[] sec){
        int L = sec.length;
        if(L==0)
            return new int[] {0,0,0};
        if(L==1)
            return new int[] {0,0,sec[0]};
        if(L==2)
            if(sec[0]>0 && sec[1]>0)
                return new int[] {0,1,sec[0]+sec[1]};
            else if(sec[0]>=sec[1])
                return new int[] {0,0,sec[0]};
            else
                return new int[] {1,1,sec[1]};

        int[] izq = Arrays.copyOfRange(sec, 0, L/2); //dividimos la secuencia
        int[] der = Arrays.copyOfRange(sec, L/2, L);
        int[] maxizq = submax(izq); //aplicamos la recursion para cada mitad
        int[] maxder = submax(der);
```

```
        int[] secmax = new int[] {0,0,-Integer.MAX_VALUE};
        int[] thissec = new int[] {maxizq[0],maxizq[0],0};

        for(int i = maxizq[0];i<maxder[1]+L/2+1;++i)
        {   thissec[1]=i;
            thissec[2]+=sec[i];
            if(thissec[2]>secmax[2] || (thissec[2]==secmax[2] &&
                thissec[1]-thissec[0]<secmax[1]-secmax[0]))
                secmax = new int[] {thissec[0],thissec[1],thissec[2]}; //actualizamos secmax

            if(thissec[2]<=0)
                thissec = new int[] {i+1,i,0}; //thissec la mandamos a 0
        }

        if(secmax[2]>maxizq[2] && secmax[2]> maxder[2])
            return secmax;

        else if(maxizq[2]>maxder[2])
            return maxizq;

        else if(maxder[2]>maxizq[2])
            return new int[] {maxder[0]+L/2,maxder[1]+L/2,maxder[2]};

        else if(maxder[2]==maxizq[2]) //debemos comparar el largo de las secuencias
        {   if(maxizq[1]-maxizq[0] > maxder[1]-maxder[0]) //secuencia derecha es mas corta
            return new int[] {maxder[0]+L/2,maxder[1]+L/2,maxder[2]};

        else //secuencia izq es mas corta
            return maxizq;
        }
        return maxizq;
    }

    static int[] ef(int[] arry){
        int maxSum = 0, thisSum = 0;
        int n = arry.length;
        int j, d=0, u=0, dg = 0, ug = 0;
        for( j=0; j<n; j++)
        {
            u=j;
            thisSum += arry[j];
            if (thisSum > maxSum || thisSum==maxSum && u-d<ug-dg) {
                maxSum = thisSum;
                dg= d;
                ug= u;
            }

            else if (thisSum <= 0){
                thisSum = 0;
                d=j+1;
                u=j;
            }
        }
        int r[] = new int [2];
        r[0] = dg;
        r[1] = ug;
```

```
        return r;
    }




     static double prom(long[] arr){
        double sum = 0;
        double media;
        for (int i = 0; i<10;++i){
            sum += arr[i];
        }
        media = sum/10;
        return media;
    }

    public static void main(String[] args) {
        //System.out.println("START");
        Random rand = new Random();
        double[] tiempos1 = new double[10];
        double[] tiempos2 = new double[10];
        double[] tiempos3 = new double[10];

        for (int i = 10000;i<=100000; i+=10000) {
            int[] sec1 = new int[i];
            int[] sec2 = new int[i];
            int[] sec3 = new int[i];
            int[] sec4 = new int[i];
            int[] sec5 = new int[i];
            int[] sec6 = new int[i];
            int[] sec7 = new int[i];
            int[] sec8 = new int[i];
            int[] sec9 = new int[i];
            int[] sec10 = new int[i];

            for (int j = 0; j < i; ++j) {
                sec1[j] = rand.nextInt(300) - 150;
                sec2[j] = rand.nextInt(300) - 150;
                sec3[j] = rand.nextInt(300) - 150;
                sec4[j] = rand.nextInt(300) - 150;
                sec5[j] = rand.nextInt(300) - 150;
                sec6[j] = rand.nextInt(300) - 150;
                sec7[j] = rand.nextInt(300) - 150;
                sec8[j] = rand.nextInt(300) - 150;
                sec9[j] = rand.nextInt(300) - 150;
                sec10[j] = rand.nextInt(300) - 150;
            }
            //tenemos 10 secuencias de i elementos aleatorios creadas


            long[] tiempofza = new long[10];

            long start = System.currentTimeMillis();
            FZBRUTA(sec1);
            long end = System.currentTimeMillis();
```

```
tiempofza[0]= end-start;

start = System.currentTimeMillis();
FZBRUTA(sec2);
end = System.currentTimeMillis();
tiempofza[1]= end-start;

start = System.currentTimeMillis();
FZBRUTA(sec3);
end = System.currentTimeMillis();
tiempofza[2]= end-start;

start = System.currentTimeMillis();
FZBRUTA(sec4);
end = System.currentTimeMillis();
tiempofza[3]= end-start;

start = System.currentTimeMillis();
FZBRUTA(sec5);
end = System.currentTimeMillis();
tiempofza[4]= end-start;

start = System.currentTimeMillis();
FZBRUTA(sec6);
end = System.currentTimeMillis();
tiempofza[5]= end-start;

start = System.currentTimeMillis();
FZBRUTA(sec7);
end = System.currentTimeMillis();
tiempofza[6]= end-start;

start = System.currentTimeMillis();
FZBRUTA(sec8);
end = System.currentTimeMillis();
tiempofza[7]= end-start;

start = System.currentTimeMillis();
FZBRUTA(sec9);
end = System.currentTimeMillis();
tiempofza[8]= end-start;

start = System.currentTimeMillis();
FZBRUTA(sec10);
end = System.currentTimeMillis();
tiempofza[9]= end-start;

tiempos1[(i/10000)-1] = prom(tiempofza);


long[] tiempodpr = new long[10];

start = System.currentTimeMillis();
submax(sec1);
end = System.currentTimeMillis();
tiempodpr[0]= end-start;
```

```
start = System.currentTimeMillis();
submax(sec2);
end = System.currentTimeMillis();
tiempodpr[1]= end-start;

start = System.currentTimeMillis();
submax(sec3);
end = System.currentTimeMillis();
tiempodpr[2]= end-start;

start = System.currentTimeMillis();
submax(sec4);
end = System.currentTimeMillis();
tiempodpr[3]= end-start;

start = System.currentTimeMillis();
submax(sec5);
end = System.currentTimeMillis();
tiempodpr[4]= end-start;

start = System.currentTimeMillis();
submax(sec6);
end = System.currentTimeMillis();
tiempodpr[5]= end-start;

start = System.currentTimeMillis();
submax(sec7);
end = System.currentTimeMillis();
tiempodpr[6]= end-start;

start = System.currentTimeMillis();
submax(sec8);
end = System.currentTimeMillis();
tiempodpr[7]= end-start;

start = System.currentTimeMillis();
submax(sec9);
end = System.currentTimeMillis();
tiempodpr[8]= end-start;

start = System.currentTimeMillis();
submax(sec10);
end = System.currentTimeMillis();
tiempodpr[9]= end-start;

tiempos2[(i/10000)-1] = prom(tiempodpr);


long[] tiempoef = new long[10];

start = System.currentTimeMillis();
ef(sec1);
end = System.currentTimeMillis();
tiempoef[0]= end-start;
```

```
        start = System.currentTimeMillis();
        ef(sec2);
        end = System.currentTimeMillis();
        tiempoef[1]= end-start;

        start = System.currentTimeMillis();
        ef(sec3);
        end = System.currentTimeMillis();
        tiempoef[2]= end-start;

        start = System.currentTimeMillis();
        ef(sec4);
        end = System.currentTimeMillis();
        tiempoef[3]= end-start;

        start = System.currentTimeMillis();
        ef(sec5);
        end = System.currentTimeMillis();
        tiempoef[4]= end-start;

        start = System.currentTimeMillis();
        ef(sec6);
        end = System.currentTimeMillis();
        tiempoef[5]= end-start;

        start = System.currentTimeMillis();
        ef(sec7);
        end = System.currentTimeMillis();
        tiempoef[6]= end-start;

        start = System.currentTimeMillis();
        ef(sec8);
        end = System.currentTimeMillis();
        tiempoef[7]= end-start;

        start = System.currentTimeMillis();
        ef(sec9);
        end = System.currentTimeMillis();
        tiempoef[8]= end-start;

        start = System.currentTimeMillis();
        ef(sec10);
        end = System.currentTimeMillis();
        tiempoef[9]= end-start;


        tiempos3[(i/10000)-1] = prom(tiempoef);

    }
    //ahora tenemos un vector con promedios de tiempo de 10 secuencias de distinto largo para
        cada uno de los algoritmos
    // (tres vectores en total)
    // tiempos1 = algoritmo1 [10000, 20000, 30000, ...]
    // tiempos2 = algoritmo1 [10000, 20000, 30000, ...]
    // tiempos3 = algoritmo1 [10000, 20000, 30000, ...]
```

```
        //debemos escribir el fichero
        for (int i = 10000; i<=100000; i+=10000) {
            System.out.println(i + " " + tiempos1[(i / 10000) - 1] + " " + tiempos2[(i / 10000) - 1]
                + " " + tiempos3[(i / 10000) - 1]);
        }

        //lo guardamos en un fichero .txt mediante 'java tiempo > resultTime.txt' ejecutandolo en
            simbolo de sistema
    }
}
```

7. Anexo 3: Tiempos medidos (analizados)

```
10000 82.2 3.0 0.3
20000 301.0 3.4 0.2
30000 519.7 3.6 0.2
40000 849.5 6.3 0.1
50000 1358.3 8.0 0.1
60000 1818.0 8.4 0.1
70000 2480.4 7.9 0.1
80000 3199.1 16.3 0.2
90000 4080.5 10.5 0.2
100000 4964.9 13.5 0.2
```

8. Anexo 4: Gráficos de cada algoritmo (detalle)

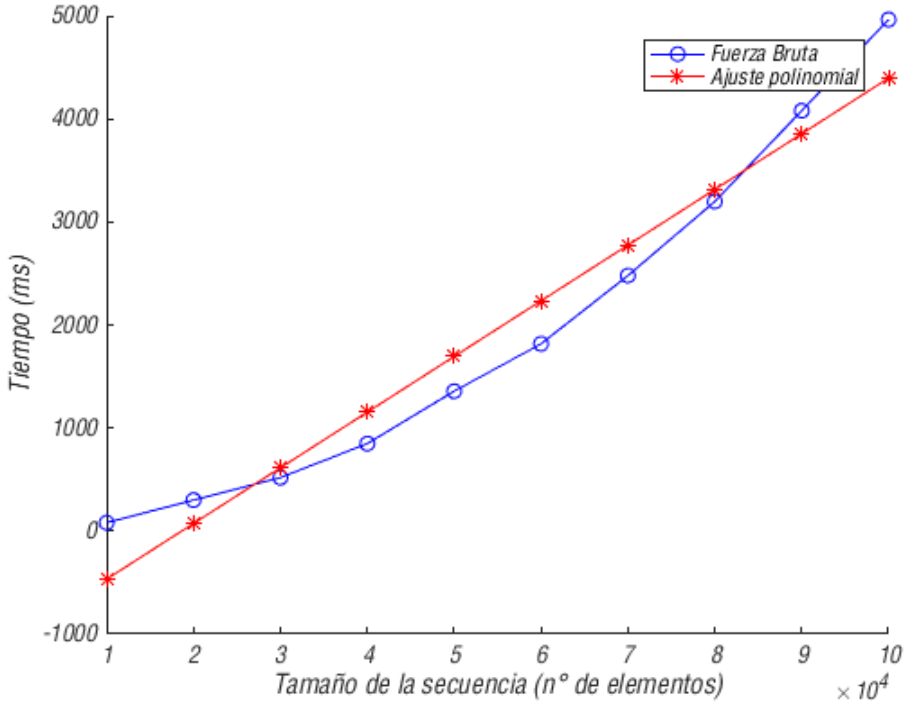


Gráfico N° de elementos vs Tiempo para algoritmo Fuerza bruta con ajuste polinomial

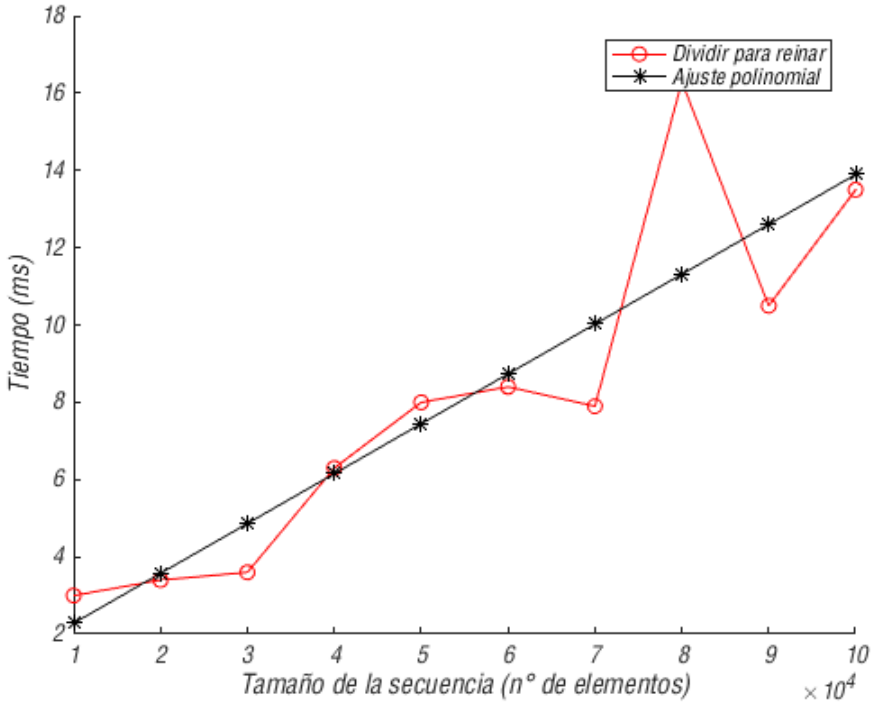


Gráfico N° de elementos vs Tiempo para algoritmo Dividir para reinar con ajuste polinomial

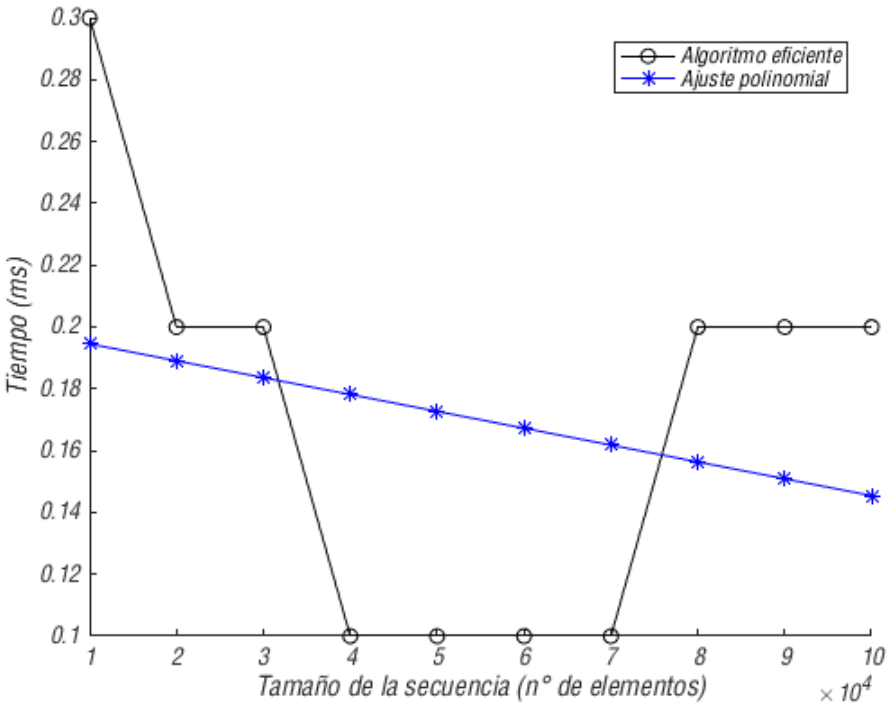


Gráfico N° de elementos vs Tiempo para algoritmo eficiente con ajuste polinomial

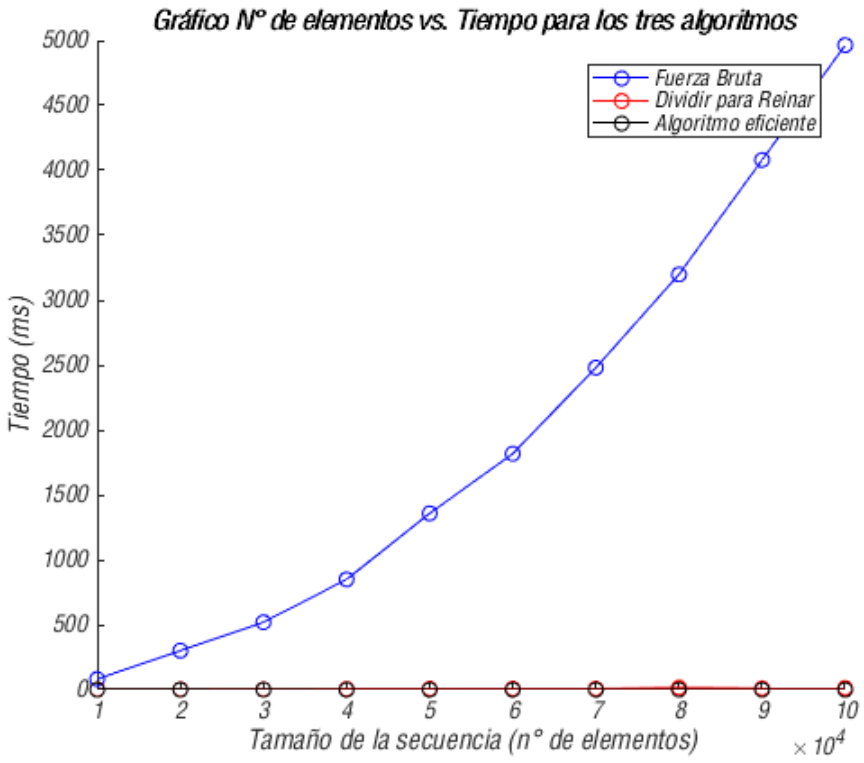


Gráfico N° de elementos vs Tiempo donde se muestran los tres algoritmos