DIVIDE Y VENCERÁS

8 DE MARZO DE 2018

GEMA RICO POZAS
UO238096

TRABAJO PEDIDO 1

Realizar un análisis de las complejidades y empírico de las 12 clases anteriores. Estudiando el código y ejecutándolo. Escribir el código para Sustracción4.java y Division4.java

Division1

Ésta implementa un método con esquema por División, cuyos parámetros son: a=1, b=3 y k=1.

Aplicando la relación O(n^K) si a<b^k → su complejidad temporal es O(n).

La complejidad de la memoria de la pila es O(N*log(n)) por lo que por mucho que n crezca nunca se desbordará.

```
public static boolean rec1 (int n){
        if (n<=0)
            cont++;
        else{
            for (int i=1;i<n;i++) cont++ ;  //O(n)
            rec1 (n/3);}  //llamada recursiva por division
        return true;  }
```

Carga de trabajo (n)	tiempo en micros
1	0,003
2	0,004
4	0,005
8	0,007
16	0,016
32	0,031
64	0,056
128	0,107
256	0,206
512	0,403
1024	0,827
2048	1,626

Division2

Ésta implementa un método con esquema por División, cuyos parámetros son: a=2, b=2 y k=1.

Aplicando la relación O(n^K*logn) si a=b^k → su complejidad temporal es O(nlogn).

La complejidad de la memoria de la pila es O(log(n)) por lo que por mucho que n crezca nunca se desbordará.

```
public static boolean rec2 (int n){
        if (n<=0)
            cont++;
        else{
            for (int i=1;i<n;i++) cont++ ;  //O(n)
            rec2 (n/2); //llamada recursiva por division
            rec2 (n/2)}; //llamada recursiva por division
        return true;  }
```

Carga de trabajo (n)	tiempo en micros
1	0,003
2	0,009
4	0,013
8	0,042
16	0,078
32	0,208
64	0,39
128	0,949
256	1,788
512	4,26
1024	8,117
2048	18,99

Division3

Ésta implementa un método con esquema por División, cuyos parámetros son: a=2, b=2 y k=0.

Aplicando la relación O(n^log b^a) si a>b^k → su complejidad temporal es O(n).

La complejidad de la memoria de la pila es O(log(n)) por lo que por mucho que n crezca nunca se desbordará.

```
public static boolean rec3 (int n){
        if (n<=0)
            cont++;
        else{
            cont++ ; // O(1)
            rec3 (n/2); //llamada recursiva division
            rec3 (n/2); //llamada recursiva division
        return true;  }
```

Carga de trabajo (n)	tiempo en micros
1	0,003
2	0,009
4	0,013
8	0,04
16	0,058
32	0,166
64	0,246
128	0,671
256	1,003
512	2,697
1024	4,028
2048	10,798

Division4

En esta clase creada se debe implementar un método recursivo por division con una complejidad O(n^2) y 4 subproblemas .

Sus parámetros son: a=4, b=4 y k=2.

Aplicando la relación O(n^k) si a<b^k → su complejidad temporal es O(n^2).

```
public static boolean rec4(int n) {
    long cont = 0;
    if (n <= 0)
      cont++;
    else {
      for (int i = 0; i < n; i++){{
            for (int j = 0; j < n; j++){
                cont++;}}
       rec4(n / 4);
      rec4(n / 4);
      rec4(n / 4);
      rec4(n / 4);}
      return true; }
```

Carga de trabajo (n)	tiempo en micros
1	0,015
2	0,016
4	0,047
8	0,079
16	0,281
32	1,469
64	4,563
128	15,8
256	66,183
512	234,693
1024	875,423
2048	3380,423

Sustracción 1

Ésta implementa un método con esquema por Sustracción, cuyos parámetros son: a=1, b=1 y k=0.

Aplicando la relación O(n^k+1) si a=1 → su complejidad temporal es O(n).

La complejidad de la memoria de la pila es O(n) por lo que la pila se desborda.

```
public static boolean rec1 (int n){
        if (n<=0)
                cont++;
        else{
                cont++ ; // O(1)
                rec1 (n-1); //llamada recursiva division
        return true;  }
```

Carga de trabajo (n)	tiempo en micros
1	0,015
2	0,016
4	0,047
8	0,015
16	0,016
32	0,031
64	0,078
128	0,141
256	0,298
512	0,714
1024	1,624
2048	3,813

Sustracción 2

Ésta implementa un método con esquema por Sustracción, cuyos parámetros son: a=1, b=1 y k=1.

Aplicando la relación O(n^k+1) si a=1 → su complejidad temporal es O(n^2).

La complejidad de la memoria de la pila es O(n) por lo que la pila se desborda.

```
public static boolean rec2 (int n){
        if (n<=0)
                cont++;
        else{
                for (int i=0;i<n;i++) cont++; // O(n)
                rec2 (n-1);
                for (int i=0;i<n;i++) cont++; // O(n)}
    return true;  }
```

Carga de trabajo (n)	tiempo en micros
1	0,003
2	0,015
4	0,016
8	0,047
16	0,172
32	0,609
64	2,594
128	10,128
256	38,838
512	150,94
1024	544,124
2048	2131,423

Sustracción 3

Ésta implementa un método con esquema por Sustracción, cuyos parámetros son: a=2, b=1 y k=0.

Aplicando la relación O(a^(n/b)) si a>1 → su complejidad temporal es O(2^n).

La complejidad de la memoria de la pila es O(n) pero no se desborda porque mucho antes el tiempo de ejecución se hace intratable.

```
public static boolean rec3 (int n){
        if (n<=0)
                cont++;
        else{
                cont++;  // O(1)
                rec3 (n-1);
                rec3 (n-1);}
return true; }
```

Carga de trabajo (n)	tiempo en micros
1	0,004
2	0,016
3	0,016
4	0,031
5	0,047
6	0,141
7	0,187
8	0,547
9	0,797
10	2,188
11	3,219
12	8,751
13	12,792

Sustracción 4

En esta clase creada implementa un método recursivo por sustracción con una complejidad O(3n/2).

Sus parámetros son: a=3, b=2 y k=0.

Aplicando la relación O(a^(n/b)) si a>1 → su complejidad temporal es O(3^n/2).

La complejidad de la memoria de la pila es O(n) pero no se desborda porque mucho antes el tiempo de ejecución se hace intratable.

```
public static boolean rec4 (int n){
        if (n<=0)
                cont++;
        else{
                cont++;
                rec4(n - 2);
                rec4(n - 2);
                rec4(n - 2);
    return true; }
```

Carga de trabajo (n)	tiempo en micros
1	0,006
2	0,01
3	0,016
4	0,016
5	0,031
6	0,016
7	0,156
8	0,156
9	0,266
10	0,289
11	1,36
12	1,38
13	2,5

Fibonacci 1

Ésta implementa un método iterativo con una complejidad O(n).

```
public static int fib1 (int n)
        {
                int n1= 0;
                int n2= 1;
                for (int i= 1; i <= n; i++)
                {
                        int s= n1+n2;
                        n1= n2;
                        n2= s;
                }
                return n1;}
```

Carga de trabajo (n)	tiempo en micros
10	0,00188
11	0,00297
12	0,00312
13	0,0036
14	0,00406
15	0,00438
16	0,00515
17	0,00328
18	0,00344
19	0,00375
20	0,00422
21	0,00453
22	0,00485
23	0,00547
24	0,00425

Fibonacci 2

Ésta implementa un método iterativo con una complejidad O(n) utilizando un vector.

```
public static int fib2 (int n, int[]v)
        {
                v[0] =0;
                v[1]=1;
                for (int i=2; i <= n; i++)
                        v[i]=v[i-1]+v[i-2];
                return v[n];
        }
```

Carga de trabajo (n)	tiempo en micros
10	0,0048
11	0,00666
12	0,00812
13	0,00951
14	0,00897
15	0,00971
16	0,01072
17	0,01154
18	0,01102
19	0,01148
20	0,01289
21	0,01378
22	0,01342
23	0,01446
24	0,01547

Fibonacci 3

Ésta implementa un método recursivo con una complejidad O(n).

```
public static int fib3 (int n)
        {
                return aux(0, 1, n);
        }

private static int aux (int n1, int n2, int n)
        {
                if (n < 1) return n1;
                return aux(n2, n1+n2, n-1);
        }
```

Carga de trabajo (n)	tiempo en micros
10	0,0078
11	0,0078
12	0,0078
13	0,0079
14	0,0093
15	0,0094
16	0,0094
17	0,0109
18	0,011
19	0,0125
20	0,014
21	0,0141
22	0,0156
23	0,0157
24	0,018

Fibonacci 4

Ésta implementa un método recursivo con una complejidad O(1.6^n).

```
public static int fib4 (int n)
        {
                if (n<=1)
                        return n;
                return fib4(n-1) + fib4(n-2);
        }
```

Carga de trabajo (n)	tiempo en micros
10	0,015
11	0,032
12	0,031
13	0,078
14	0,109
15	0,172
16	0,282
17	0,453
18	0,734
19	1,188
20	1,937
21	3,126
22	5,063
23	8,19
24	13,255

SumaVector 1

Ésta implementa un método iterativo con una complejidad O(n).

```
public static int suma1(int[]a)
        {
                int n= a.length;
                int s=0;
                for(int i=0;i<n;i++)
                        s=s+a[i];
                return s;
        } // fin de suma1
```

Carga de trabajo (n)	tiempo en micros
3	0,0016
6	0,0031
12	0,0031
24	0,0047
48	0,011
96	0,0234
192	0,0531
384	0,1001
769	0,1937
1536	0,3844

SumaVector 2

Ésta implementa un método recursivo con una complejidad O(n).

```
public static int suma2(int[]a){
                return recSust (0,a);
        } // fin de suma2
private static int recSust(int i, int[]a){
                if (i==a.length) return 0;
                else return a[i]+recSust(i+1,a);
        } // fin de recSust
```

Carga de trabajo (n)	tiempo en micros
3	0,009
6	0,016
12	0,018
24	0,041
48	0,075
96	0,148
192	0,397
384	0,868
769	2,023
1536	4,37

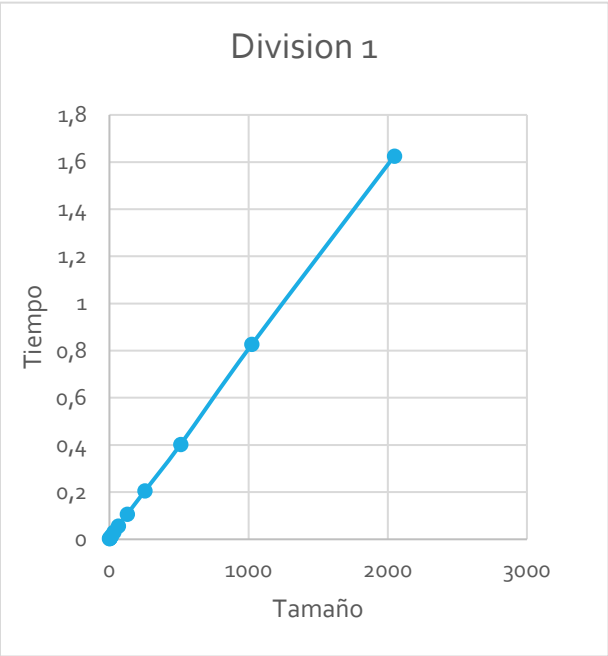
SumaVector 3

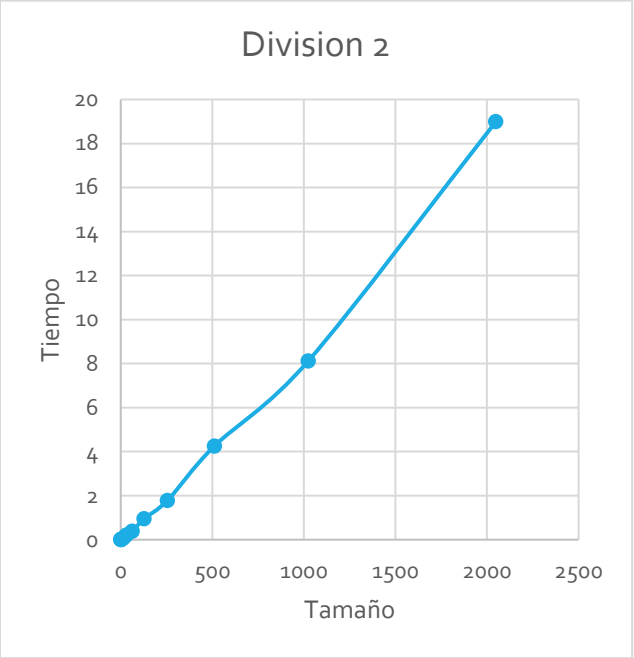
Ésta implementa un método recursivo con una complejidad O(n).

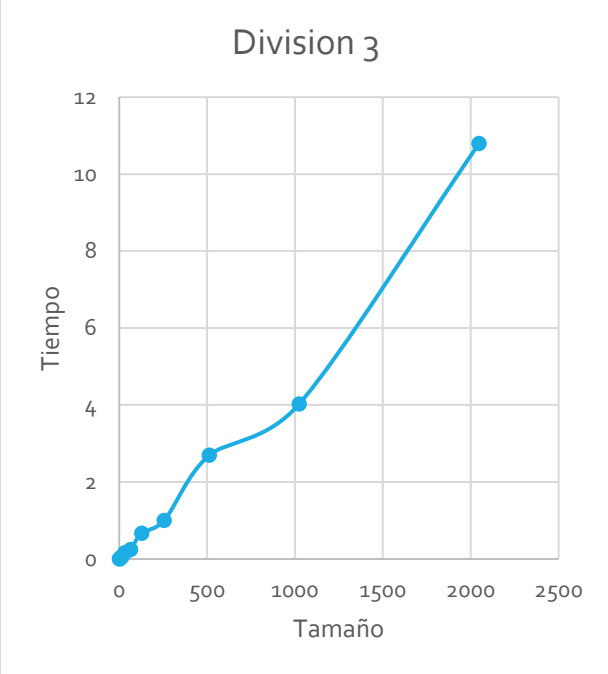
```
public static int suma3 (int[]a){
                return recDiv (0,a.length-1,a);
        } // fin de suma3Vector

private static int recDiv(int iz,int de,int[]a){
                if (iz==de)
                        return a[iz];
                else{
                        int m= (iz+de)/2;
                        return recDiv(iz,m,a)+ recDiv(m+1,de,a); }
        }
```

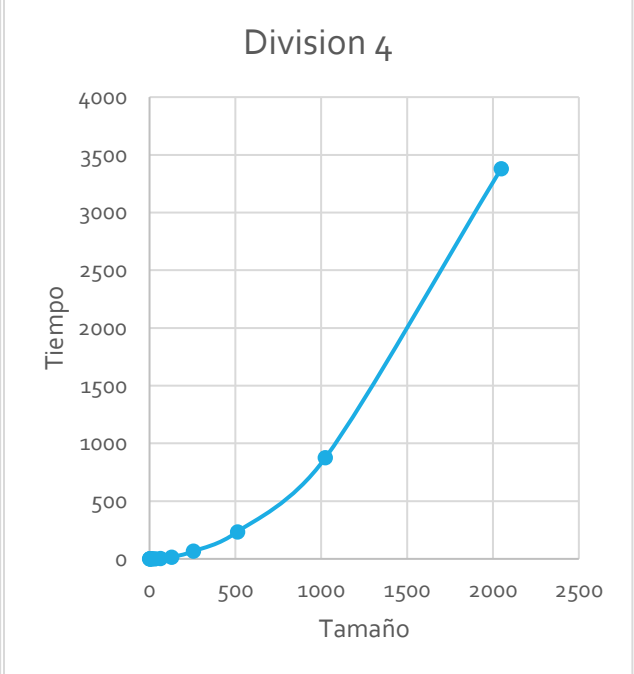
Carga de trabajo (n)	tiempo en micros
3	0,015
6	0,016
12	0,016
24	0,047
48	0,109
96	0,203
192	0,422
384	0,828
769	1,733
1536	3,219

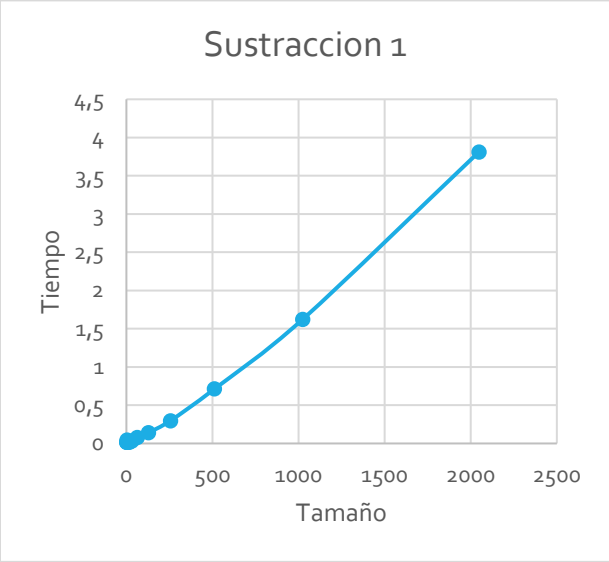
GRÁFICAS

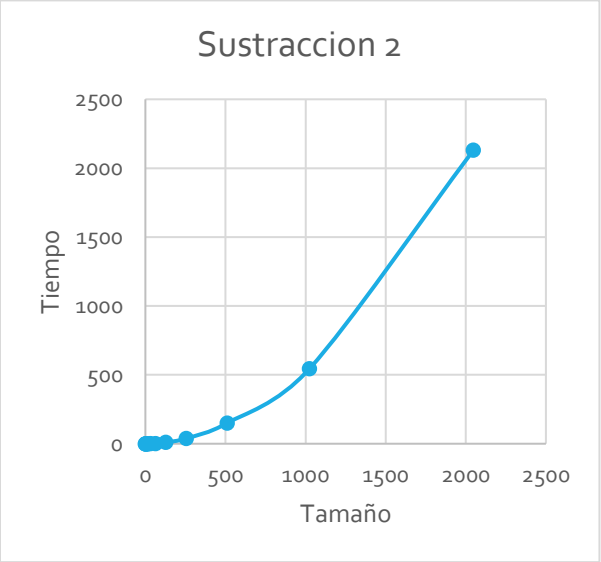

Division 1
1,8
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
Tiempo
0
1000
2000
3000
Tamaño

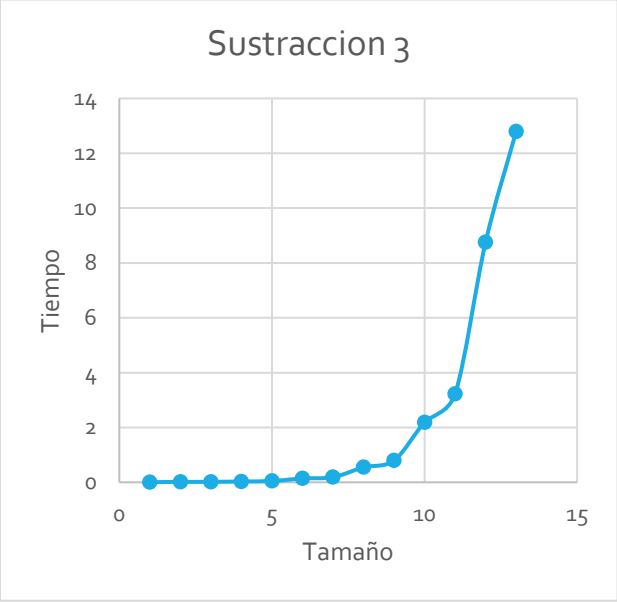

Division 2
20
18
16
14
12
10
8
6
4
2
0
Tiempo
0
500
1000
1500
2000
2500
Tamaño

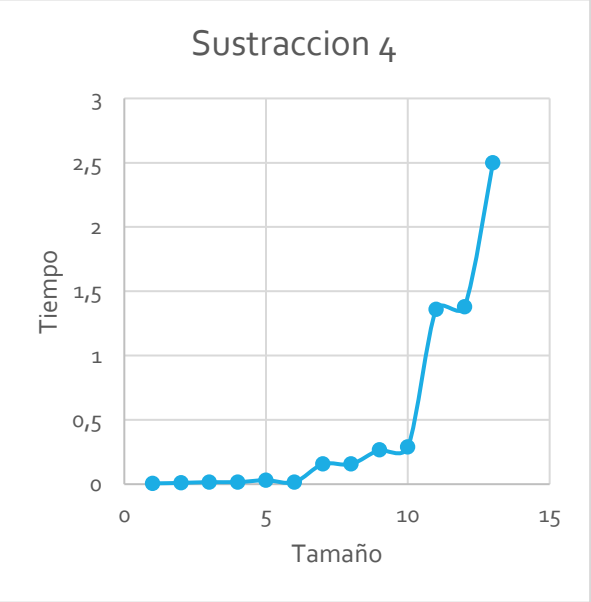

Division 3
12
10
8
6
4
2
0
Tiempo
0
500
1000
1500
2000
2500
Tamaño

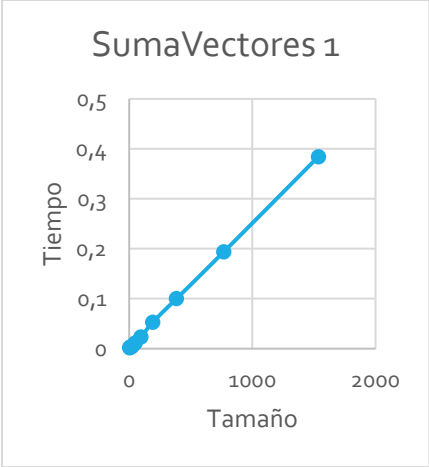

Division 4
4000
3500
3000
2500
2000
1500
1000
500
0
Tiempo
0
500
1000
1500
2000
2500
Tamaño

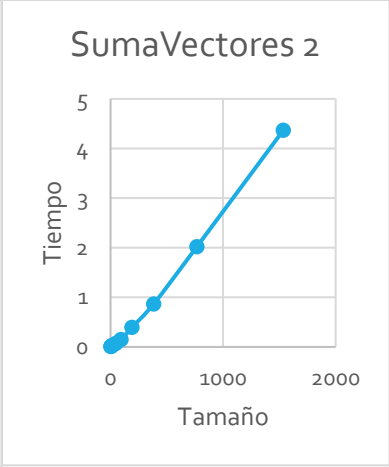

Sustraccion 1
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
Tiempo
0
500
1000
1500
2000
2500
Tamaño

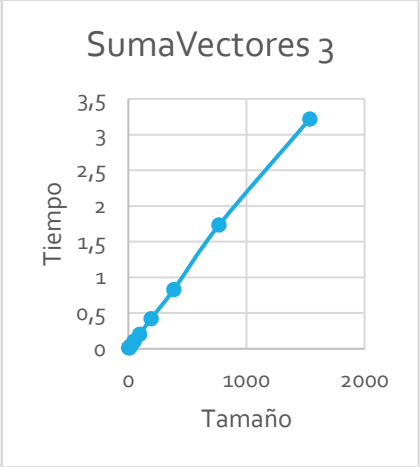

Sustraccion 2
2500
2000
1500
1000
500
0
Tiempo
0
500
1000
1500
2000
2500
Tamaño

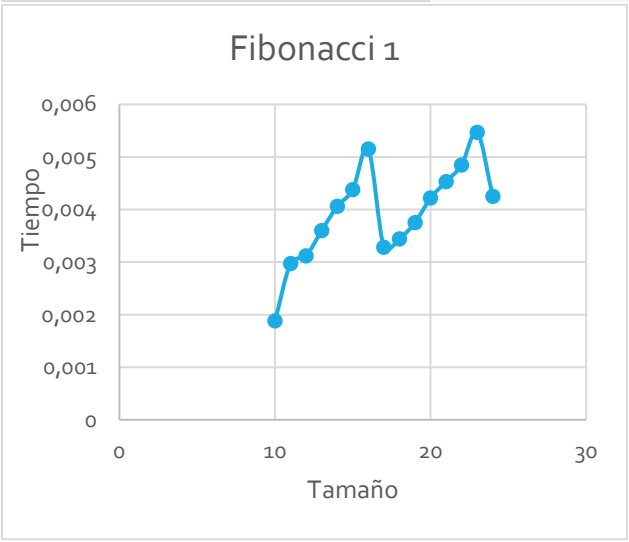

Sustraccion 3
14
12
10
8
6
4
2
0
Tiempo
0
5
10
15
Tamaño

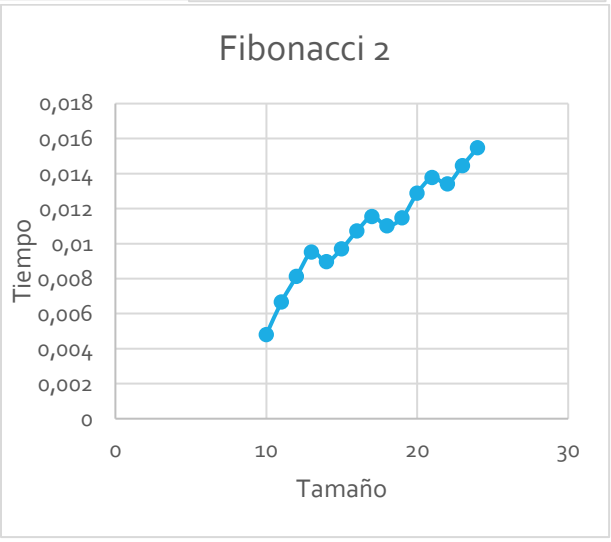

Sustraccion 4
3
2,5
2
1,5
1
0,5
0
Tiempo
0
5
10
15
Tamaño

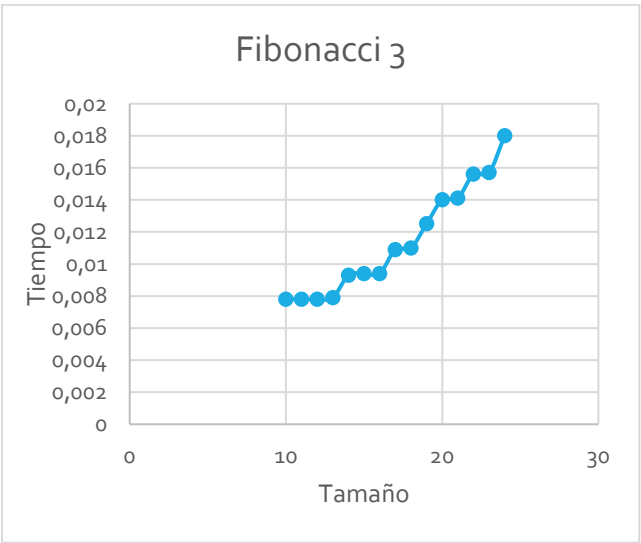

SumaVectores 1
Tiempo
0,5
0,4
0,3
0,2
0,1
0
0
1000
2000
Tamaño

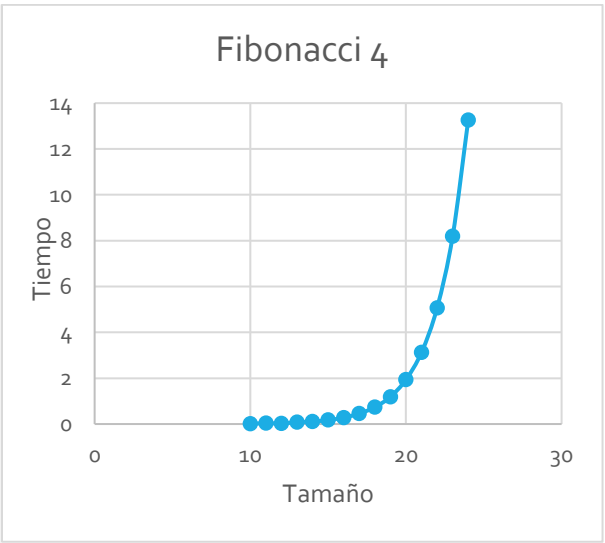

SumaVectores 2
Tiempo
5
4
3
2
1
0
0
1000
2000
Tamaño


SumaVectores 3
Tiempo
3,5
3
2,5
2
1,5
1
0,5
0
0
1000
2000
Tamaño


Fibonacci 1
Tiempo
0,006
0,005
0,004
0,003
0,002
0,001
0
0
10
20
30
Tamaño


Fibonacci 2
Tiempo
0,018
0,016
0,014
0,012
0,01
0,008
0,006
0,004
0,002
0
0
10
20
30
Tamaño


Fibonacci 3
Tiempo
0,02
0,018
0,016
0,014
0,012
0,01
0,008
0,006
0,004
0,002
0
0
10
20
30
Tamaño


Fibonacci 4
Tiempo
14
12
10
8
6
4
2
0
0
10
20
30
Tamaño

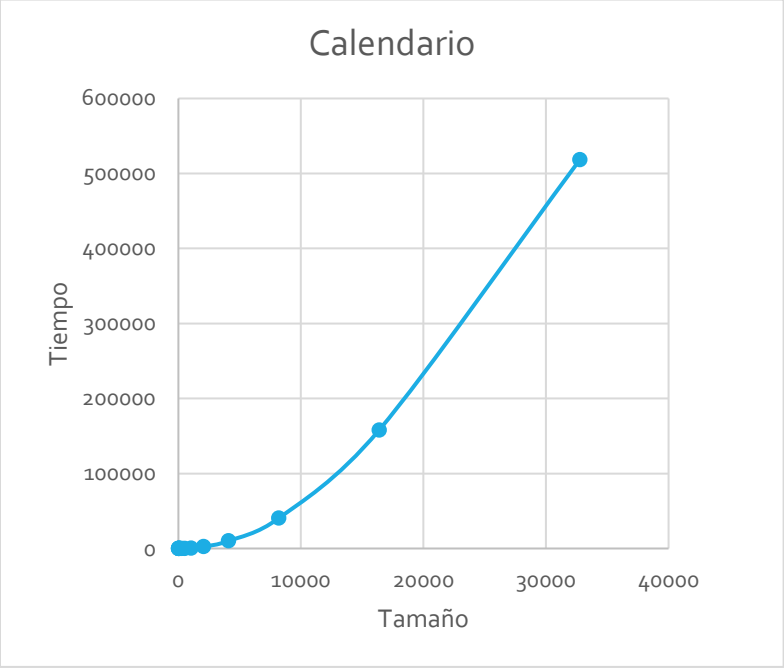
TRABAJO PEDIDO 2

En este apartado nos encomiendan la tarea de simular un algoritmo capaz de organizar un torneo de petanca con x jugadores potencia de 2.Se debe de realizar con la técnica divide y vencerás.
La idea es minimizar la duración total del campeonato, ya que así se ahorrarán muchos costes.
Otro dato que tenemos es que, según las normas del campeonato, todos los días cada participante podrá jugar únicamente contra otro de los participantes.

Dado que es un algoritmo previamente creado y comprobado, he aplicado con ayuda de los apuntes de teoría y de otras fuentes de información el agoritmo Torneo, que se encuentra en la clase Calendar. Todo ello se encuentra en el zip adjunto.

→¿Qué complejidad tiene el algoritmo creado?

La complejidad del algoritmo implementado es O(n^2) tal y como se puede apreciar en la gráfica y en la tabla de datos medidos.



Carga de trabajo (n)	tiempo en micros
2	0,003
4	0,023
8	0,053
16	0,173
32	0,597
64	2,173
128	8,471
256	35,962
512	144
1024	566
2048	2464
4096	10115
8192	40383
16384	157976
32768	518542