Laboratorio 2 – Estructuras de Datos y Algoritmos 1

Isaías Labrador Sánchez
Universidad Eafit
Medellín, Colombia
ilabradors@eafit.edu.co

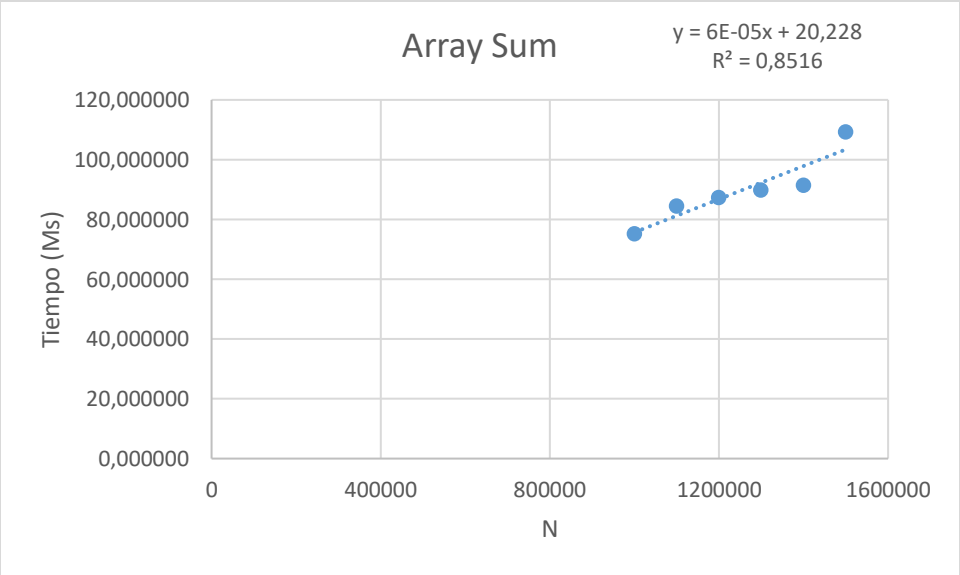
Nombre completo de integrante 2
Universidad Eafit
Medellín, Colombia
megarciaj@eafit.edu.co

3.1

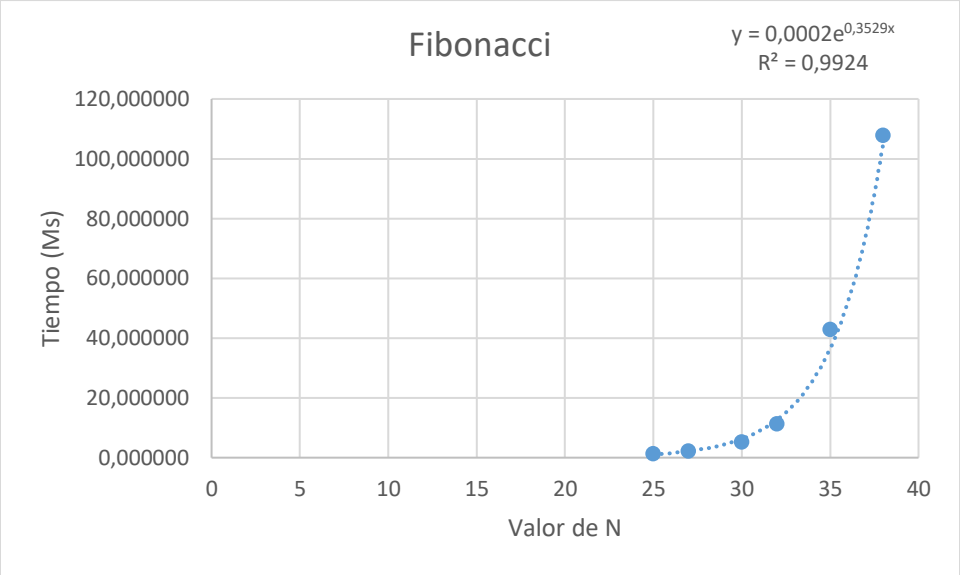
	N = 100.000	N = 1 000 000	N = 10 000 000	
ArraySum	6.666958	71.995513	432.551968	
ArrayMax	6.906146	43.669682	473.951259	
	N = 25	N = 27	N = 30	N = 32
Fibonacci	1.283923	2.056257	5.176186	11.160832

3.2

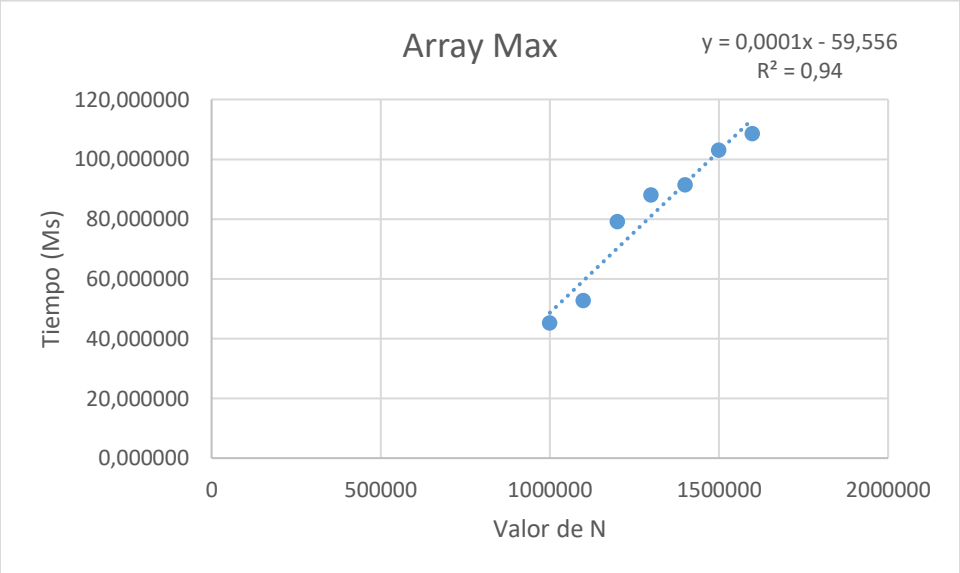
Array Sum - O(n):



Fibonacci – O(n^2):



Array Max – O(n):

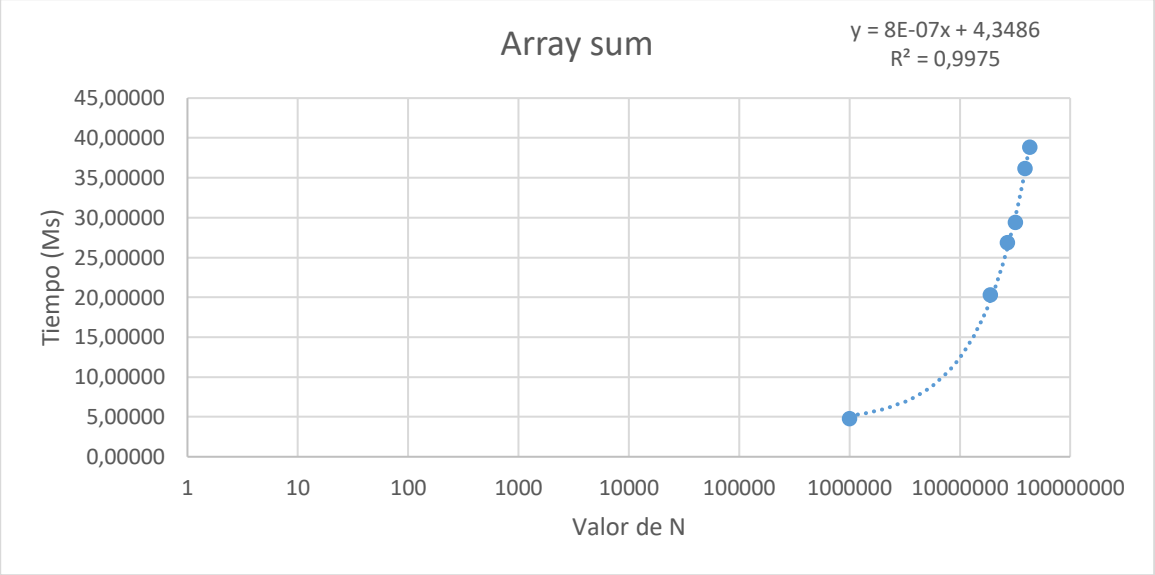


3.3 Las gráficas y tiempos tomados experimentalmente demuestran que la notación O define el comportamiento y complejidad de un algoritmo y así se puede predecir su funcionamiento y de igual manera comparar

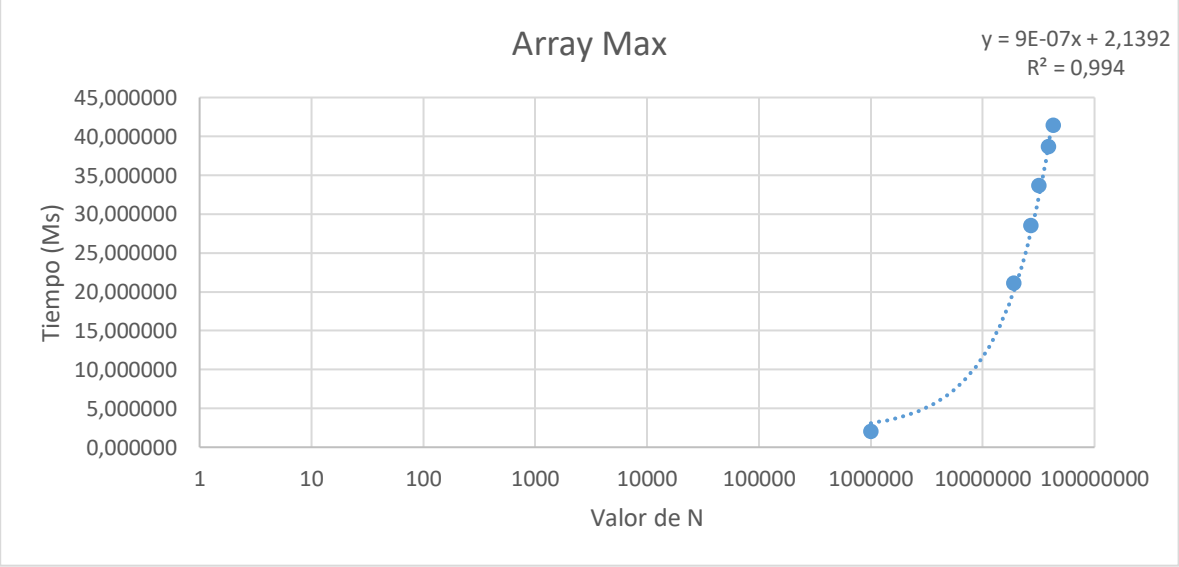
3.4

	N =100000	N = 1000000	N = 10000000	N = 100000000
Array Sum	2.03936	4.771810	13.350522	80.557037
Array maximun	2.372941	7.286632	13.883087	88.853151
Insertion sort	2106.598493	Más de 5 Min	Más de 5 Min	Más de 5 Min
Merge sort	18.587306	137.127708	1419.24398	Mas de 5 Min

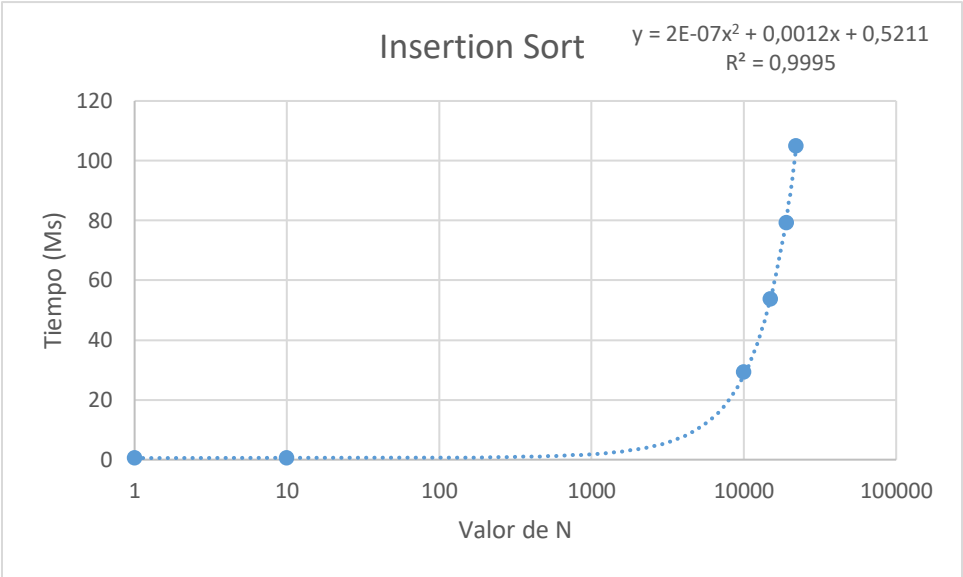
3.5 Array Sum: Lineal O(n)



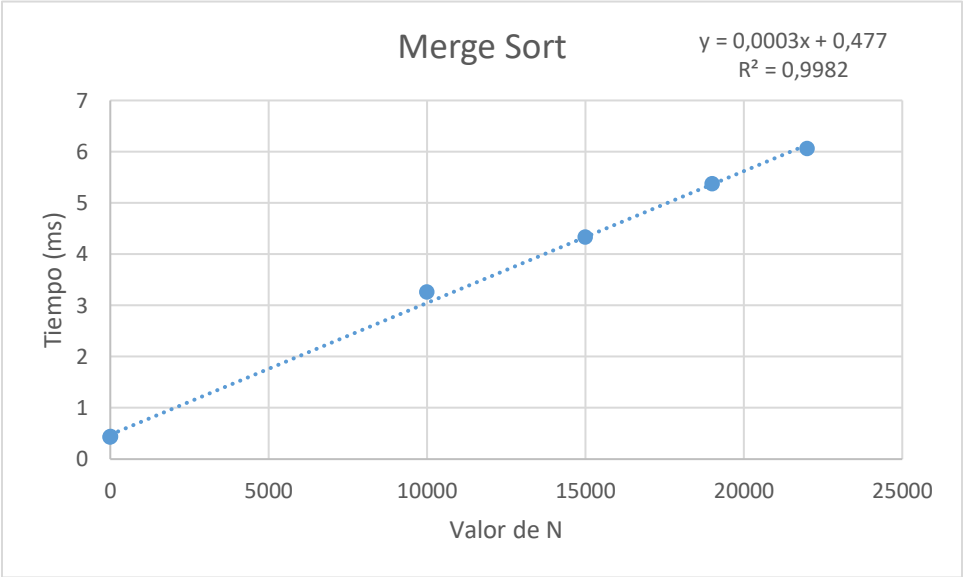
Array Max: O(n)



Insertion Sort: O(n^2)



Merge Sort: O(n)



3.6

En general siempre se cumple la regla; los tiempos obtenidos experimentalmente, ilustrados en una gráfica, siempre tienen una línea de tendencia definida a partir de la notación O del algoritmo que se haya probado.

Además, notamos como al transformar un algoritmo de recursivo a cíclico se reducen, tanto los tiempos de respuesta como la memoria utilizada y gastada en nuestro PC.

3.7

Insertion Sort se vuelve muy poco eficiente con arreglos de tamaño grande, por no decir que es posible no obtener una respuesta el mismo día que se hace la petición. Esto gracias a que su complejidad está dada por O(n^2), es decir es exponencial.

3.8

Obviamente para valores grandes de n se va a demorar más, pero gracias a que la complejidad de Array sum está dada por O(n), su ejecución por lo general va a ser rápida.

En comparación con Insertion Sort cuya complejidad está dada por O(n^2) para valores grandes de n, comparando ambos casos, Array sum seria mucho más eficiente gracias a que es lineal.

3.9

Gracias a las gráficas podemos asegurar y decir que Insertion sort es más eficiente, o igual de eficiente, que merge sort para arreglos pequeños. Pero hay una gran diferencia que yace en que merge sort actúa de manera más eficiente para arreglos grandes donde la N tiende a los millones puesto que la complejidad de merge sort es O(n) y la de Insertion Sort O(n^2)

3.10

```
public class Laboratory1
public int maxSpan(int[] nums) {
    if (nums.length > 0) {    // C1
        int maxSpan = 1;      // C2
        for (int i = 0; i < nums.length; i++) //C3*N
            for (int j = nums.length - 1; j > i; j--) //C4*(N^2)
                if (nums[j] == nums[i]) {             //C5*(N^2)
                    int count = (j - i) + 1;          //C6*(N^2)
                    if (count > maxSpan) maxSpan = count; // C7*(N^2)
                    break;         //C8*(N^2)
                }
        return maxSpan; //C9
    } else return 0;    //C10
}
```

*Solución del ejercicio MaxSpan Tomada de http://gregorulm.com/codingbat-java-array-3-part-i/

** Ulm, G (2013) CodingBat: Java. Array-3, Part I [Source Code]. http://gregorulm.com/codingbat-java-array-3-part-i/

MaxSpan: Considerar las apariciones más a la izquierda y más a la derecha de cada valor en el arreglo inicial. Devolver el número mayor de elementos dentro de los subconjuntos formados por las apariciones de los valores. Un arreglo de un solo elemento tiene un "span" de 1.

Ejemplos: {0, 1, 2, 3, 4, 4, 1} -> 6

{6} -> 1

{6, 2, 6, 4, 5, 2} -> 5

La solución trata de esto: si el número del arreglo que me pasen es mayor que 0 se considera que mínimo ya hay un elemento por lo que se declara span = 1. después de esto se entra a un ciclo que me recorre los elementos del arreglo parámetro, 1 por 1 dentro del cual hay otro ciclo en que se empieza desde el último elemento hacia atrás hasta llegar a la posición actual, 1 por 1, allí se pregunta el momento en que ambos números en los que se está "parado" actualmente son iguales, si son iguales se restan las posiciones de los números actuales y se les suma 1 para dar el tamaño del "span2" y se pasa a preguntar si el "span2" que se encontró es mayor a el "span" que se declaró inicialmente, si lo es, se iguala "span" a "span2" sino se sale del ciclo.

Al momento de terminar el ciclo se retorna el valor de span, el cual va a ser el mayor encontrado.

3.11 Complejidad O(n^2) – fix34 || * desde C6 hasta C15 se multiplican por n^2

```
public class Laboratory1
public int Fix34(int[] nums) {

  int []ans= new int[nums.length];  //C1
   int pos4=0;                      //C2
  for(int i=0; i<nums.length; i++){ //C3*n
    if(nums[i]==3){                 //C4
     ans[i]=nums[i];                //C5
     for(int j=pos4+1; j<nums.length; j++){ //C*N^2
       if(nums[j]==4){                      // C7
         pos4= j;                           //C8
         int temp= nums[i+1];               //C9
         ans[i+1]=nums[j];                  //C10
         ans[j]=temp;                       //C11
         break;                             //C12
       }
     }
     for(int k=i+1; k<nums.length; k++){    //C*N^2
       if(ans[k]>0){                        //C13
         i++;                               //C14
       }else break;                         //C15
     }

    }else if(ans[i]<=0){                    //C16
     ans[i]=nums[i];                        //C17
    }
  }
  return ans;                               //C18
}
```

Complejidad O(n^2) – countClumps || * desde C6 hasta C9 se multiplican por n^2

```
public class Laboratory1{

  public int countClumps(int[] nums) {
  int count=0;                              //C1
  int tam=0;                                //C2
  for(int i=0; i<nums.length-1; i++){       //C3*n
    if(nums[i]==nums[i+1]){                 //C4
      count++;                              //C5
     for(int j=i; j<nums.length; j++){      //C6*n^2
       if(nums[j]==nums[i]){                //C7
        tam++;                              //C8
       }else break;                         //C9
     }
    i+=tam-1;                               //C10
     tam=0;                                 //C11
    }
  }
  return count;                             //C12
}


}
```

Complejidad: O(n^2) – canBalance

```
public class Laboratory1{

  public boolean canBalance(int[] nums) {
  int izq=0;     //C1
  int der=0;     //C2
  for(int i=1; i<nums.length; i++){    //C3*n
    for(int j=0; j<i; j++){            //C4*n^2
      izq+=nums[j];                    //C5
    }
    for(int k=i; k<nums.length; k++){ //C6*(n^2)
      der+=nums[k];                    //C7
    }
    if(izq==der)return true;           //C8
    izq=0;                             //C9
    der=0;                             //C10
  }
  return false;                        //C11
}


}
```

*Desde C4 hasta C7 se multiplica por n^2

Complejidad: O(n^2) — linearIn

```java
public class Laboratory1{

    public boolean linearIn(int[] outer, int[] inner) {
     boolean found=false;                              //C1
     for(int i=0; i<inner.length; i++){                //C2*n
        for(int j=0; j<outer.length; j++){             //C3*n^2
            if(inner[i]==outer[j]){                    //C4
            found=true;                                //C5
            break;                                     //C6
            }
        }
        if(!found)return false;                        //C7
        found=false;                                   //C8
        }
    return true;                                       //C9
    }

}
```

*desde C3 hasta C6 se multiplican por N^2

Complejidad: O(n^2) – MaxSpan

```java
public class Laboratory1
public int maxSpan(int[] nums) {
    if (nums.length > 0) {     // C1
        int maxSpan = 1;       // C2
        for (int i = 0; i < nums.length; i++) //C3*N
            for (int j = nums.length - 1; j > i; j--) //C4*(N^2)
                if (nums[j] == nums[i]) {              //C5*(N^2)
                    int count = (j - i) + 1;           //C6*(N^2)
                    if (count > maxSpan) maxSpan = count; // C7*(N^2)
                    break;         //C8*(N^2)
                }
        return maxSpan; //C9
    } else return 0;     //C10
}
```

Complejidad: O(n) – matchUp

```java
public class Laboratory1{

    public int matchUp(int[] nums1, int[] nums2) {
      int count=0;                                                          // C1
      for(int i=0; i<nums1.length; i++){                                    // C2 * n
        if(nums1[i]-nums2[i]<=2 &&nums1[i]-nums2[i]>=-2&&nums1[i]-nums2[i]!=0)count++; // C3*n
       }
     return count;      //c4
    }

}
```

Complejidad: O(n) – fizzArray

```java
public class Laboratory1{

    public int[] fizzArray(int n) {
      int [] a= new int [n];     //C1
      for(int i=0; i<n; i++){    //C2*n
        a[i]=i;                  //C3*n
      }
     return a;                   //C4
}

}
```

Complejidad: O(n) -- evenOdd

```java
public class Laboratory1{

      public ArrayList evenOdd(int[] nums){

     ArrayList <Integer> ans = new ArrayList<>();     //C1
     ArrayList <Integer> evens = new ArrayList<>();    //C2
     ArrayList <Integer> odds = new ArrayList<>();     //C3
     for(int i=0; i<nums.length; i++){                //C4*n
      if(nums[i]%2==0){                               //C5*n
       evens.add(nums[i]);                            //C6*n
      }else odds.add(nums[i]);                        //C7*n
     }
     ans.addAll(evens);                               //C8
     ans.addAll(odds);                                //C9
     return ans;                                      //C10
    }
}
```

Complejidad: O(n) – only14

```java
public class Laboratory1{
      public boolean only14(int[] nums) {
     if(nums.length<1)return true;          //C1
      for(int i=0; i<nums.length; i++){     //C2*n
       if(nums[i]!=1 && nums[i]!=4)return false;  //C3*n
       }
     return true;                           //C4
    }
}
```

Complejidad: O(n) – sum13

```
public class Laboratory1{
    public int sum13(int[] nums) {
      int sum=0;                              //C1
        if(nums.length>0){                    //C2
         if(nums[0]!=13)sum+=nums[0];         //C3
            for(int i=1; i<nums.length; i++){   //C4*n
             if(nums[i]!=13 && nums[i-1]!=13){  //C5*n
             sum+=nums[i];                      //C5*n
            }
        }
     }
     return sum;                          //C6
    }
}
```

3.12

N y m representan el número de iteraciones que se debe de hacer una operación se representa para ciclos y llamados recursivo se utilizan para representar una constante que se diferencie de C que representa un tiempo. Es decir esa C que es un tiempo, al tener que realizar muchas veces una operación, se dice que ese tiempo se hace n, m, etc. Veces.

4.

1) a

2) b

3) length-1

4) x+1, a[i]