Laboratorio Nro. 2: Notación O

Santiago Escobar Mejía
Universidad Eafit
Medellín, Colombia
sescobarm@eafit.edu.co

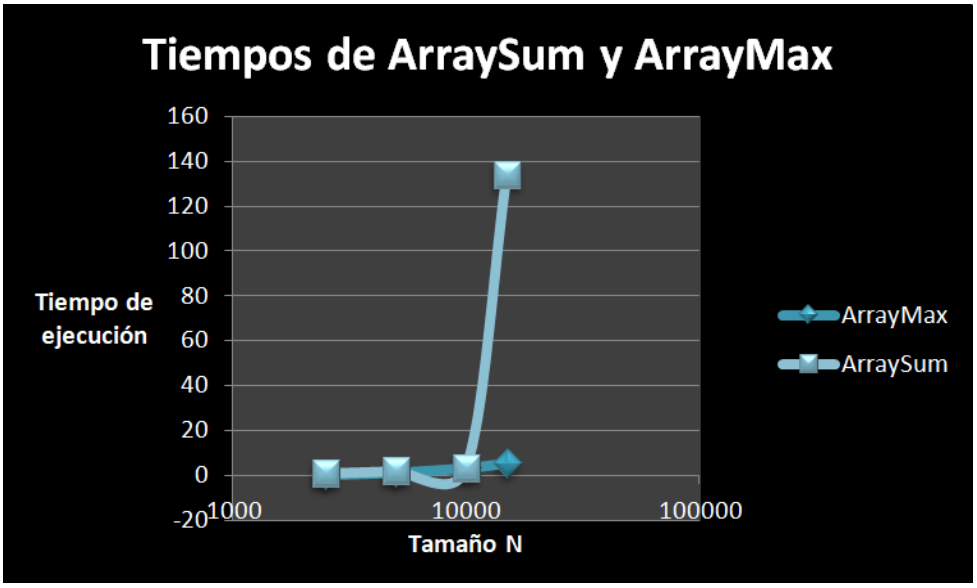
Sebastián Giraldo Gómez
Universidad Eafit
Medellín, Colombia
sgiraldog@eafit.edu.co

Luisa María Vásquez
Universidad Eafit
Medellín, Colombia
lmvasquez@eafit.edu.co

3) Simulacro de preguntas de sustentación de Proyectos

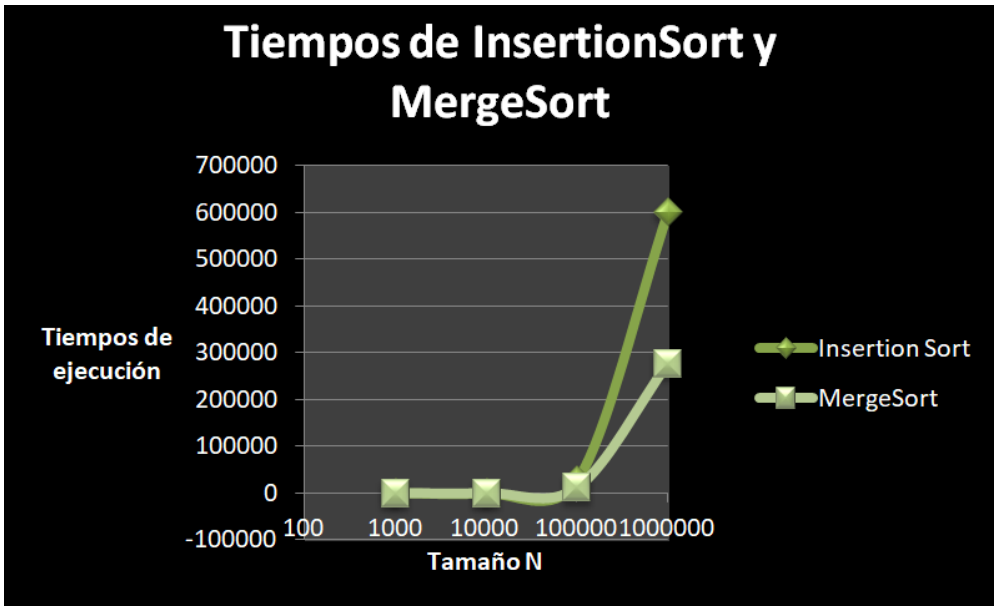
3.1 y 3.2)

N	ArrayMax	ArraySum
2500	0	1
5000	1	2
10000	3	3
15000	5	134



DOCENTE MAURICIO TORO BERMÚDEZ
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627
Correo: mtorobe@eafit.edu.co

N	Insertion Sort	MergeSort
1000	3	3
10000	340	140
100000	27492	12342
1000000	Más de 600000	277239



3.3 ¿Qué concluyen con respecto a los tiempos obtenidos en el laboratorio y los resultados teóricos obtenidos con la notación O?

Después de comparar los resultados obtenidos con las gráficas de complejidad de la notación O podemos concluir lo siguiente de cada uno de los métodos:

- MergeSort: La grafica de los tiempos de ejecución de este método se asemeja a la gráfica de O(n log n)
- InsertionSort: La grafica de los tiempos de ejecución de este método se asemeja a la de la complejidad exponencial, es decir a la gráfica de O(n^2).
- ArraySum: La grafica de los tiempos de ejecución de este método se asemeja a la de la complejidad lineal, es decir a la gráfica de O(n).
- ArrayMax: La grafica de los tiempos de ejecución de este método se asemeja a la de la complejidad exponencial, es decir a la gráfica de O(n^2).

Debido a esto, el método que tiene la complejidad más baja y que consume menos recursos es el ArraySum, con su complejidad lineal de O(n).

Además también podemos concluir que hay factores que afectan la relación de los datos obtenidos con los teóricos, como lo puede ser lentitud de la máquina en un lapso de tiempo, llamados a otros métodos como “Math.max ó Math.random”, entre otros.

3.4 Teniendo en cuenta lo anterior, ¿Qué sucede con *Insertion Sort* para valores grandes de N?

El algoritmo Insertion Sort se va volviendo más lento a medida que va recibiendo valores más grandes de N, debido a su complejidad O(n^2), es decir, tiene una complejidad exponencial, ya que cada que aumente un valor se tienen que hacer más y más ejecuciones, lo que ralentiza el tiempo de ejecución y deja de ser tan óptimo a la hora de implementarse en un manejo de grandes cantidades de datos.

Todo esto se puede vivenciar en los tiempos de ejecución tomados anteriormente, en los cuales se ve que a medida que crece demora más y más.

3.5 Teniendo en cuenta lo anterior, ¿Qué sucede con *ArraySum* para valores grandes de N? ¿Por qué los tiempos no crecen tan rápido como *Insertion Sort*?

Como es de esperarse a medida que aumenta la N se hace más lenta la ejecución de ArraySum, pero no se ve muy afectada con el aumento de N debido a su complejidad lineal en la que el tiempo crece proporcionalmente a la N.

En el método ArraySum los tiempos de ejecución no crecen tan rápido como en Insertion Sort debido a su diferencia de complejidad: ArraySum – O(n) e Insertion Sort – O(n^2), en base a lo anterior se puede ver que los tiempos de ejecución de Insertion Sort crecen exponencialmente por lo que son mucho más lentos a los de ArraySum.

3.6 Teniendo en cuenta lo anterior, ¿Qué tan eficiente es *Merge sort* con respecto a *Insertion sort* para arreglos grandes? ¿Qué tan eficiente es *Merge sort* con respecto a *Insertion sort* para arreglos pequeños?

Para arreglos grandes y pequeños Merge sort es mucho más eficiente que Insertion Sort debido a su diferencia de complejidad :

- Merge Sort: O(n logn)
- Insertion Sort: O(n^2)

Debido a esto el Merge Sort es eficiente en todos los casos, ya que se ejecuta menos veces y reduce drásticamente los tiempos de ejecución.

Por ejemplo:

- Con arreglos de tamaño 3 (n=3):
  - Merge Sort: 3 x log (3)= 1.4
  - Insertion Sort: 3^2= 9
- Con arreglos de tamaño 100 (n=100)
  - Merge Sort: 100 x log (100)= 200
  - Insertion Sort: 100^2= 10000

3.7 Expliquen con sus propias palabras cómo funciona el ejercicio *maxSpan y ¿por qué?*

MaxSpan:

```
public int maxSpan(int[] nums) {
  int span = 0;
  int tmp = 0;

  for (int i = 0; i < nums.length; i++) {
    for (int j = 0; j < nums.length; j++) {
      if (nums[i] == nums[j]) {
        tmp = j-i+1;
        span = Math.max(tmp,span);
      }
    }
  }
  return span;
}
```

Explicación: El método hace recorridos por intervalos (i , j) usando los ciclos anidados buscando cuándo i es igual a j, una vez hecho esto calcula la cantidad de elementos incluyendo entre i y j incluyéndolos, después de hacerlo evalúa si esa distancia es mayor a la encontrada anteriormente y así sucesivamente, al final retorna la distancia más grande entre dos elementos iguales incluyéndolos.

3.8 Complejidad numerales 2.1 y 2.2

Array 2:

CountEvens
Complejidad: O(n)

```
public int countEvens(int[] nums) {
  int cont=0;
  for(int i=0;i<nums.length;i++){    //C1*n
    if(nums[i]%2==0){
      cont++;
    }
  }
  return cont;
}
```

BigDiff
Complejidad: O(n)

```
public int bigDiff(int[] nums) {
  int min=9999;
  int max=0;
  for(int i=0;i<nums.length;i++){  //C1*n
    max=Math.max(max,nums[i]);
    min=Math.min(min,nums[i]);
  }
  return max-min;
}
```

Sum13
Complejidad: O(n)

```
public int sum13(int[] nums) {
  int sum=0;
  for(int i=0;i<nums.length;i++){  //C1*n
    if(nums[i]!=13) sum+= nums[i];
    else i++;
```

```
  }
  return sum;
}
```

Has22
Complejidad: O(n)

```
public boolean has22(int[] nums) {
  for(int i=0;i<nums.length-1;i++){  //C1*n
    if(nums[i]==2&&nums[i+1]==2){
      return true;
    }
  }
  return false;
}
```

FizzArray
Complejidad: O(n)

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

Array 3:

MaxSpan
Complejidad: O(n^2)

```
public int maxSpan(int[] nums) {
    if (nums.length > 0) {
        int maxSpan = 1;
        for (int i = 0; i < nums.length; i++)         //C1*n
            for (int j = nums.length - 1; j > i; j--)   //C2*n^2
                if (nums[j] == nums[i]) {
                    int count = (j - i) + 1;
                    if (count > maxSpan) maxSpan = count;
```

```
                    break;
                }
        return maxSpan;
    } else return 0;
}
```

Fix34
Complejidad: O(n^2)

```
public int[] fix34(int[] nums) {
  int pos4=0;
  for(int i=0;i<nums.length-1;i++){        //C1*n

    if(nums[i]==3){
      for(int j=pos4;j<nums.length;j++){   //C2*n^2
        if(nums[j]==4){
          pos4=j;
          nums[j]= nums[i+1];
          nums[i+1]=4;
          break;
        }
      }
    }
  }
  return nums;
}
```

CanBalance
Complejidad: O(n^2)

```
public boolean canBalance(int[] nums) {
    for (int i = 0; i < nums.length; i++) {
        int sumatoria = 0;
        for (int j = 0; j < i; j++) {         //C1*n
          sumatoria += nums[j];
        }
        for (int e = i; e < nums.length; e++){    //C2*n^2
          sumatoria -= nums[e];
        }
        if (sumatoria == 0) {
        return true;
```

```
        }
    }
    return false;
}
```

SquareUp
Complejidad: O(n^2)

```
public int[] squareUp(int n) {
   int a[]=new int[n*n];
   int pos=a.length-1-n;
   int cont=n;
  for(int i=a.length-1;i>=0;i-=n){      //C1*n
    int ii=i;
    for(int j=1;j<=cont;j++){            //C2*n^2
      a[ii]=j;
      ii--;
    }
    cont--;
  }
  return a;
}
```

CountClumps
Complejidad: O(n)

```
public int countClumps(int[] nums) {
  int rep=-1;
  int cont=0;
  for(int i=0; i<nums.length-1;i++){              //C1*n
    if(nums[i]==nums[i+1]&&nums[i]!=rep){
      rep=nums[i];
      cont++;
    }else if(nums[i]!=rep){
      rep=0;
    }
  }
  return cont;
}
```

DOCENTE MAURICIO TORO BERMÚDEZ
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627
Correo: mtorobe@eafit.edu.co

3.9 Expliquen con sus palabras las variables (qué es 'n', qué es 'm', etc.) del cálculo de complejidad del numeral anterior

En la notación O(n) , "n" se utiliza como una variable de referencia que indica la complejidad mediante el número y tipo de ejecuciones, un ejemplo de esto puede ser un algoritmo de que diga los números de n hasta 1, este algoritmo se ejecutaría n-1 veces.

Pero además de la n existen otros valores como:

a* T(n **op c**) + **C**

-a: Que indica el número de llamadas recursivas que se generan a partir de cada llamada recursiva.

-op c: Expresa cómo decrece el problema:
-Aritméticamente (restando una unidad) -1.
-Geométricamente (dividiendo) /2.

-**Constante:** Indica el coste de acciones realizadas por el método sin considerar la llamada recursiva.
Una vez obtenida está formula se utilizan teoremas para llegar a la notación O(n)

4) Simulacro de Parcial

1. C. O(n+m)
2. D. O(m x n)
3. B. O(ancho)
4. B. $O(n^3)$
5. D. $O(n^2)$

DOCENTE MAURICIO TORO BERMÚDEZ
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627
Correo: mtorobe@eafit.edu.co