Laboratorio Nro. 2: Notación O grande

Alejandro Murillo González
Universidad Eafit
Medellín, Colombia
amurillog@eafit.edu.co

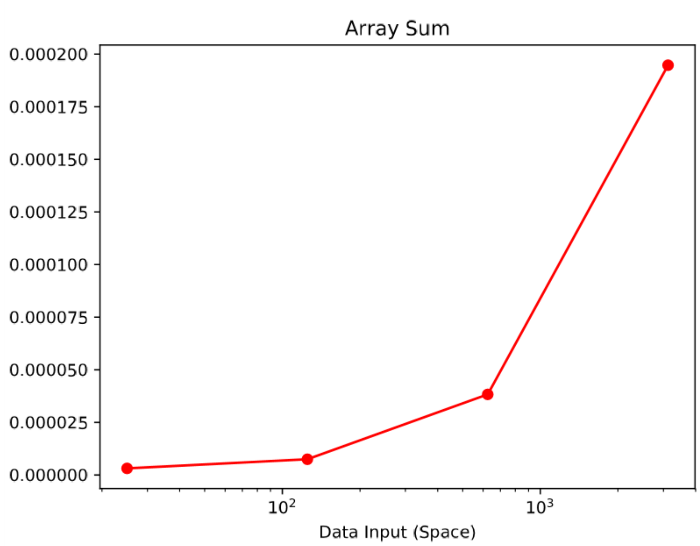
Juan Pablo Vidal Correa
Universidad Eafit
Medellín, Colombia
Jpvidalc@ eafit.edu.co

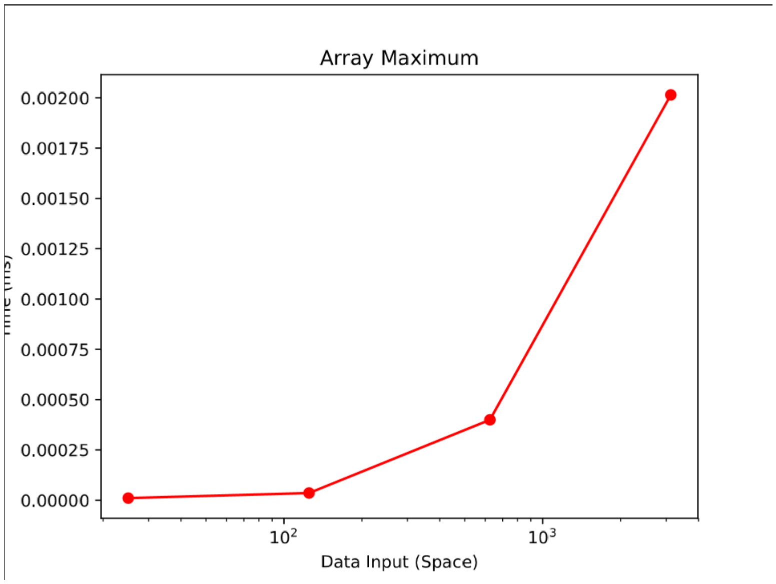
3) *Ejercicios en línea sin documentación HTML en GitHub:*

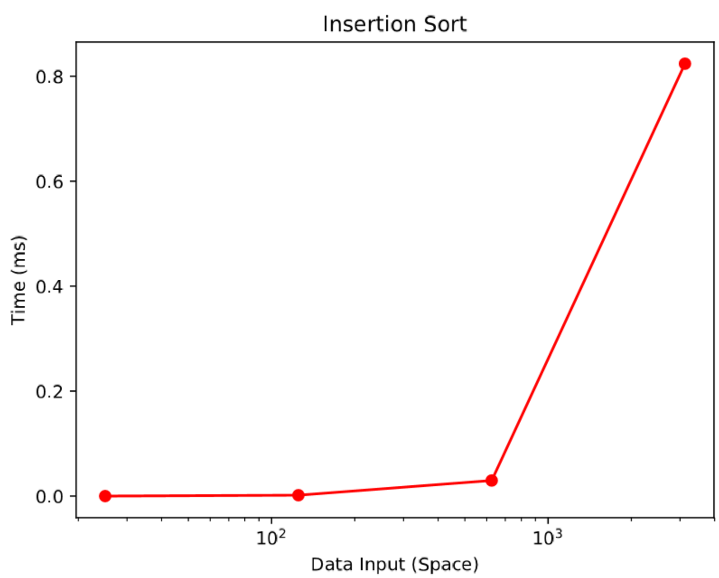
3.1 Completar la siguiente tabla con tiempos en milisegundos:

	N = 25	N = 125	N = 625	N = 3125
Array Sum	3.160492578570833e-06	7.5061698741057275e-06	3.832097251517135e-05	0.00019476535515442758
Array Max	1.0666662452676586e-05	3.5950603081243235e-05	0.00039980231118921046	0.002014418957266585
Insertion Sort	6.123454370980967e-05	0.0017303696867675311	0.02972206233202476	0.823923032524234
Merge Sort	7.822219131958796e-05	0.00040533317320168294	0.014741722571171345	0.04724620355705533

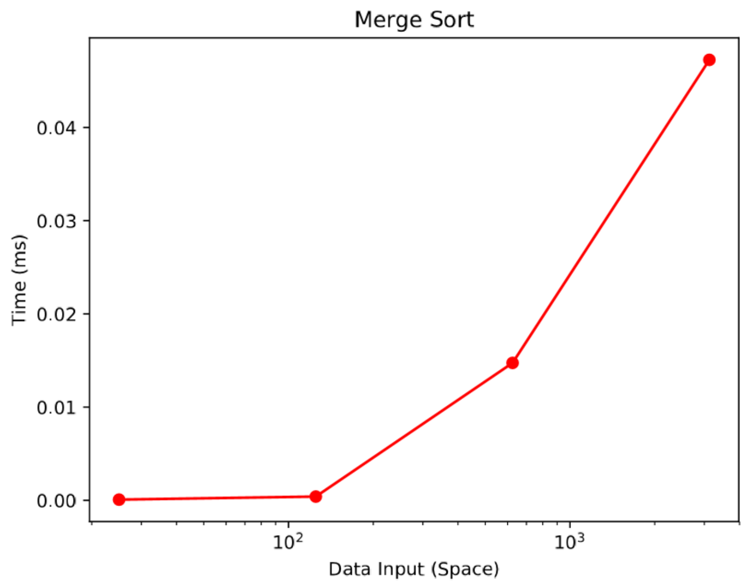
3.2 Grafiquen los tiempos que tomó en ejecutarse array sum, array maximum, insertion sort y merge sort, para entradas de diferentes tamaños:







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



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

En la práctica, las gráficas indican que arraySum y arrayMax son funciones lineales (no se evidencia muy bien en la gráfica, pero observando los valores y los intervalos se puede ver que es muy cercano a una gráfica lineal) , lo que es correcto porque su complejidad es O(n), ya que solo hace un llamado o solo utilizan un ciclo; por otro lado, en la gráfica de merge sort se puede identificar una función exponencial, lo que es válido ya que su complejidad es O(n^2), el caso anterior es muy similar a lo que ocurrió con insertion sort, ya que ambos tienen una complejidad parecida, esto se puede evidenciar al ver la similitud de las gráficas, pero en este último se puede destacar que la complejidad es ligeramente mayor a merge sort, ya que la gráfica y la tabla demuestran tiempos de ejecución ligeramente más largos, por lo que su complejidad es ligeramente mayor, pero se sigue ajustando a una gráfica exponencial.

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

Teniendo en cuenta que Insertion Sort es una función exponencial, y es creciente, como se muestra en la gráfica, cada vez que el número de N es mayor, a este algoritmo le tomara más tiempo ejecutarse.

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?

ArraySum es un algoritmo que solo emplea un ciclo para ejecutarse, esto implica que tenga una complejidad de O(n), lo que significa que puede identificarse como una función lineal, lo que permite que ha valores grandes necesite más tiempo de ejecución, pero siempre de una forma proporcional, mucho menor a una función exponencial, como es el caso de Insertion Sort, cuyo tiempo de ejecución evidencia una función exponencial que hace crecer los tiempo de ejecución, y esto debido a que se emplean 2 ciclos.

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?

La eficiencia en este tipo de algoritmos depende mucho del arreglo con el que se trabaje, esto se muestra en las gráficas cuando observamos que Insertion sort tiene unos tiempo de ejecución muy pequeños para una N pequeña, por lo que es muy eficiente para arreglos de menor tamaño, pero cuando se le pasan datos más grandes, el tiempo de ejecución se acrecienta de manera exponencial, por otro lado Merge sort, es lo contrario, cuando se le pasaban arreglos datos pequeños los tiempos de ejecución eran mayores a los de Insertion Sort, pero eran menores al momento de trabajar con una N de mayor tamaño, por lo que era más eficiente en este tipo arreglos grandes. Por lo tanto, si comparamos Insertion Sort con Merge sort, encontramos que el primero es más eficiente con datos de menor tamaño que el segundo, pero esto se invierte cuando se trabaja con arreglos de mayor tamaña.

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

El ejercicio maxspan funciona observando los elementos de un arreglo, para encontrar el “span” máximo, es por ello que se debe hacer un doble recorrido

(2 ciclos) para revisar los números dentro del arreglo, e ir observando cual es el mayor “span”, es por esto que dentro de n de los ciclos se encuentra un if, para ir comparando cada número con el siguiente.

3.8 Calculen la complejidad de los ejercicios en línea, numerales 2.1 y 2.2, y agréguenla al informe PDF

Array 2:

//sum13

```
    public int sum13(int[] nums){
    int result = 0;
    for (int i = 0; i < nums.length; i++){   // C x n
        if (nums[i] != 13)   //C x n
            result += nums[i];
        else if (i <= nums.length )   //C x n
        i++;
    }
    return result;
}
```

T(n) = O(2(C x n) + C x n) == T(n) es O(n)

//has22

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

T(n) = O(C x n + C x n) == T(n) es O(n)

//sum28

```
public boolean sum28(int[] nums){
  int result = 0;
  for (int i = 0; i < nums.length; i++){   // C x n
    if (nums[i] == 2){   //C x n
      result += 2;
    }
```

```
  }
  return result == 8;
}
```

T(n) = O(C x n + C x n) == T(n) es O(n)

//isEverywhere

```
public boolean isEverywhere(int[] nums, int val){
    for(int i = 0; i < nums.length-1; i++){  // C x n
        if(nums[i] != val && nums[i+1] != val){  //C x n
            return false;
        }
    }
    return true;
}
```

T(n) = O(C x n + C x n) == T(n) es O(n)

//matchUp

```
public int matchUp(int[] nums1, int[] nums2){
  int total = 0;
  for (int i = 0; i < nums1.length; i++){  // C + n
    if(nums1[i] - 2 == nums2[i] || nums1[i] - 1 == nums2[i]
    || nums1[i] + 1 == nums2[i] || nums1[i] + 2 == nums2[i]){  //C x n
      total++;
    }
  }
  return total;
}
}
```

T(n) = O(C x n + C x n) == T(n) es O(n)

Array 3:

//maxSpan

```
public int maxSpan(int[] nums) {
  if (nums.length > 0){  //C
    int maxSpan = 1;
    for (int i = 0; i < nums.length; i++){   // C x n
      for (int j = nums.length - 1; j > i; j--){  // (C x n) x n
        if (nums[j] == nums[i]) { // n x n
```

```
                    int total = (j - i) + 1;
                    if (total > maxSpan){
                        maxSpan = total;
                    }
                }
            }
        }
        return maxSpan;
    }
    else{
        return 0;
    }
}
```

T(n) = O(C x n + n x n + (C x n) x n) == T(n) es O(n^2)

//fix34

```
public int[] fix34(int[] nums){
    for (int i = 0; i < nums.length - 1 ; i++)   // C x n
        if (nums[i] == 3 && nums[i+1] != 4){  // n x n
        int count = nums[i + 1];
        nums[i + 1] = 4;
            for (int j = i + 2; j < nums.length; j++){   // (C x n) x n
              if (nums[j] == 4){   // n x n
             nums[j] = count;
         }
        }
    }
    return nums;
}
```

T(n) = O(C x n + n x n + (C x n) x n) == T(n) es O(n^2)

//fix45

```
public int[] fix45(int[] nums) {
    for (int i = 0; i < nums.length; i++){   // C x n
        if (nums[i] == 5 && i == 0 || nums[i] == 5 && nums[i - 1] != 4) { // C x n
            for (int j = 0; j < nums.length; j++){   // (C x n) x n
                if (nums[j] == 4 && nums[j + 1] != 5) {  // n x n
                    int count = nums[j + 1];
                    nums[j + 1] = 5;
                    nums[i] = count;
```

```
          break;
        }
      }
    }
  }
  return nums;
}
```

T(n) = O(C x n + n x n + (C x n) x n) == T(n) es O(n^2)

//canBalance

```
public boolean canBalance(int[] nums) {
  for (int i = 0; i < nums.length ; i++) {    // C x n
    int side = 0;
    for (int j = 0; j < i; j++){   // (C x n) x n
    side += nums[j];
    }
    for (int j = i; j < nums.length; j++){  // (C x n) x n
    side -= nums[j];
    }
    if (side == 0){   // n x n
    return true;
    }
  }
  return false;
}
```

T(n) = O(C x n + n x n + 2((C x n) x n)) == T(n) es O(n^2)

//linearIn

```
public boolean linearIn(int[] outer, int[] inner){
boolean notFound;
  for(int inI = 0; inI < inner.length; inI++){   // C x n
    notFound = true;
    for(int  outI = 0 ; outI < outer.length && notFound; outI++){  // (C x n) x n
      if(inner[inI] == outer[outI]){   // n x n
        notFound = false;
         }
    }
    if(notFound){
      return false;
```

```
        }
  }
  return true;
}
}
```

T(n) = O(C x n + n x n + (C x n) x n) == T(n) es O(n^2)

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

Las variables "n”, "m" y/o "y" son las variables de entradas de los ejercicios en línea de codingBat, esto con el objetivo de facilitar los cálculos de complejidad.

4) Simulacro de Parcial

1. C
2. B
3. B
4. B
5. D

5) Lectura recomendada (opcional)

a) Resumen:

Complejidad de los Algoritmos y Cotas Inferiores de los Problemas
Para medir que tan eficiente es un algoritmo, el tiempo es la medida principal; y el tamaño del problema (n) permite calcular el costo de ejecución.
Al analizar el algoritmo, se hace un análisis matemático de una operación fundamental para el algoritmo, para determinar así el número de operaciones necesarias para completar el algoritmo.
Un algoritmo es el óptimo si ya no es posible mejorar más ni la cota inferior (asumiendo que esta sea lo más alta posible) ni el algoritmo.
Se acota la complejidad así:

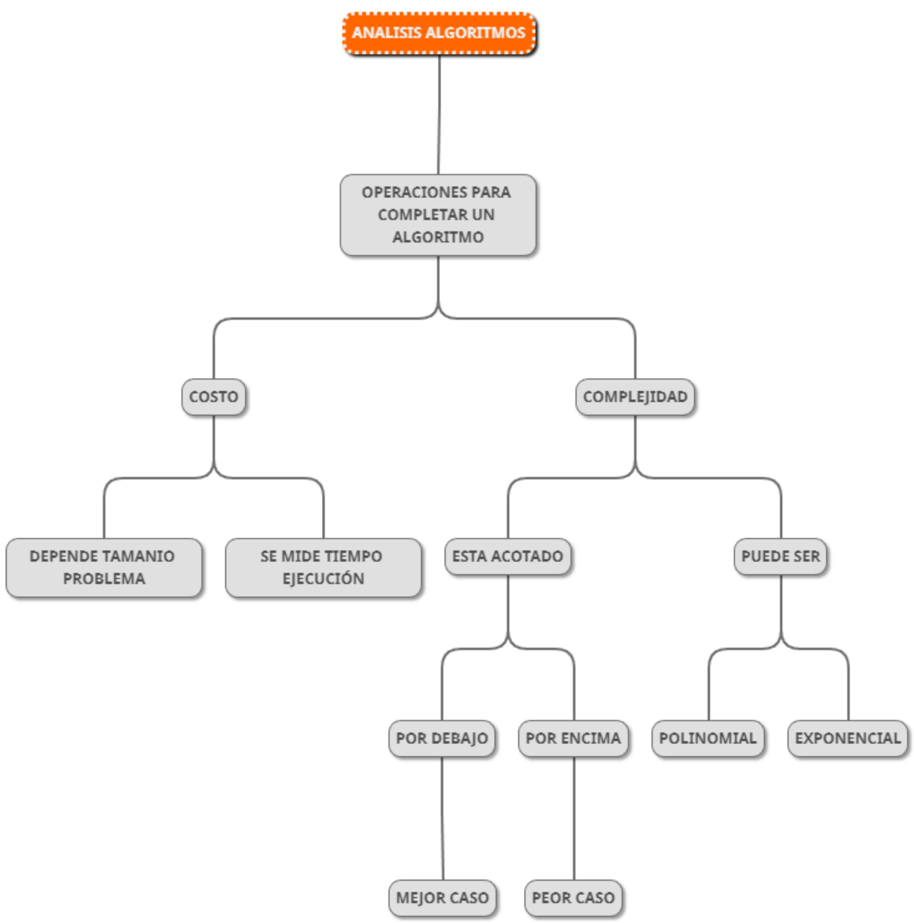
- Peor Caso:

  $f(n) = O(g(n))$ si y sólo si existen dos constantes positivas c y n_0 tales que $|f(n)| <= c|g(n)|$ para toda $>= n_0$.

- Mejor Caso:

$f(n) = (g(n))$ si y sólo si existen constantes positivas c y n_0 tales que para toda $n > n_0$, $|f(n)| >= c|g(n)|$.

b) Mapa Conceptual:



6) Trabajo en Equipo y Progreso Gradual (Opcional)

a) Actas de reunión:

Integrante	Fecha	Hecho	Haciendo	Por Hacer
Murillo	10/09/2017	Implemento codigo ArraySum y ArrayMax	Codigo Insertion Sort	Merge Sort
Vidal	12/09/2017	Implemento codigo en linea array2	Codigo en linea array3	Calcular complejidad codigos en linea
Murillo	15/09/2017	Implemento gráficas y tablas del codigo del punto 1		resumen y mapa conceptual opcional
Vidal	15/09/2017	Calculo complejidad	Contestando preguntas numeral 3	Simulacro del parcial