INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO

ANÁLISIS Y DISEÑO DE ALGORITMOS

PRÁCTICA No. 1

“Análisis experimental de la eficiencia de algoritmos de ordenamiento”

GRUPO: 3CM7

ALUMNO:

Acevedo Juárez Sebastián

PROFESORA:

Miriam Pescador Rojas

Introducción:

Durante el curso, hemos analizado el comportamiento de algunos algoritmos para entender su eficiencia de manera teórica. Cada algoritmo se desempeña diferente dependiendo del contexto en el que se encuentre, ya que no sólo se tiene que tomar en cuenta el tiempo de ejecución, sino también los elementos que involucran a este.

En este reporte, reflejaremos las implementaciones y resultados experimentales arrojados para algoritmos de búsqueda de valor máximo y de ordenamiento, tomando en cuenta el tiempo de ejecución y el número de elementos involucrados. De esta manera, nos es posible comparar su eficacia, y determinar qué tan convenientes pueden ser en determinadas situaciones.

Desarrollo:

Para los dos ejercicios los algoritmos fueron implementados en Python, haciendo uso de las siguientes bibliotecas:

- Random: Hacemos uso de esta cuando generamos el arreglo de N números.
- Timeit: Nos permite reportar el tiempo de ejecución de manera más precisa que si utilizamos la biblioteca time.
- Openpyxl: Los datos reflejados en las pruebas, son transferidos a una hoja de cálculo de Excel con el apoyo de esta biblioteca, donde son registrados.

Se realizó un archivo por cada ejercicio, teniendo funciones por cada algoritmo.

Algoritmos de búsqueda de valor máximo

Para esta primera prueba, utilizaremos dos códigos que tienen como fin encontrar el valor máximo dentro de un arreglo de N números. El primer algoritmo se trata de un algoritmo iterativo, mientras que el segundo es recursivo.

Pseudocódigo

Algorithm 1: Algoritmo iterativo para encontrar el valor máximo en una lista de enteros.

Input: L: lista con n numeros enteros
Output: max: el valor máximo de la lista

```
1  begin
2      int Maximo1(L)
3      {
4      max = L[0]; for i ← 1 to n do
5          if L[i] > max then
6              max = L[i];
7      return max;
8      }
```

Algorithm 2: Algoritmo recursivo para encontrar el valor máximo en una lista de enteros.

Input: $v[]$: lista con n numeros enteros
Output: el valor máximo de la lista

```
1  begin
2      int Maximo2(const int v[], n/2)
3      {
4      if n == 1 then
5          return n[0]
6      else
7          int izq = Maximo2(v, n/2);
8          int der = Maximo2(v + n/2, n − n/2);
9          return izq > der : der;
10     }
```

Implementación en Python:

```
import random, timeit
import openpyxl
#Abre el documento.
wb = openpyxl.load_workbook("pruebas.xlsx")
#Selecciona la hoja en la cuál escribir.
hoja = wb['Ejec1']

#Iterativo.
def Maximo1(L,n):
    max = L[0]
    for i in range(n):
        if L[i] > max:
            max = L[i]
    return max

#Recursivo.
def Maximo2(L, n):
    if n == 1:
        return L[0]
    mitad = n//2
    Lizq = L[0:mitad]
    Lder = L[mitad:]
    izq = Maximo2(Lizq,mitad)
    der = Maximo2(Lder,n-mitad)
    if izq > der: return izq
    else: return der

for i in range (2,52,2):
    #Obtiene el número de elementos en base a i.
    n = i*100
    print("n = ", n)
    #Genera arreglo(Numeros entre 0 y 100,000)
    L = [random.randint(0,100000) for j in range(n)]
    #Toma el tiempo del algoritmo iterativo.
    tiempo1 = round((timeit.timeit(lambda: Maximo1(L,n) ,
number=1)*1000000), 3)
    #Toma el tiempo del algoritmo recusrivo.
    tiempo2 = round ((timeit.timeit (lambda: Maximo2(L,n) ,
number=1)*1000000), 3)

    print("Tiempo iterativo =", tiempo1, " || Tiempo recursivo:
",tiempo2)
    print("=========================================================")

    #Coloca los resultados en la Hoja de Cálculo.
    hoja.cell(row=(2+i/2),column=2,value=tiempo1)
    hoja.cell(row=(2+i/2),column=3,value=tiempo2)

#Guarda el archivo.
wb.save("pruebas.xlsx")
```

Pantallas de prueba:

```
n =  200
Tiempo iterativo = 8.7  || Tiempo recursivo:  73.6
=====================================================
n =  400
Tiempo iterativo = 16.8  || Tiempo recursivo:  139.8
=====================================================
n =  600
Tiempo iterativo = 25.4  || Tiempo recursivo:  197.8
=====================================================
n =  800
Tiempo iterativo = 32.4  || Tiempo recursivo:  256.8
=====================================================
n =  1000
Tiempo iterativo = 43.5  || Tiempo recursivo:  324.4
=====================================================
n =  1200
Tiempo iterativo = 54.3  || Tiempo recursivo:  406.4
=====================================================
n =  1400
Tiempo iterativo = 63.5  || Tiempo recursivo:  464.5
=====================================================
n =  1600
Tiempo iterativo = 73.3  || Tiempo recursivo:  537.6
=====================================================
n =  1800
Tiempo iterativo = 80.5  || Tiempo recursivo:  596.1
=====================================================
n =  2000
Tiempo iterativo = 97.6  || Tiempo recursivo:  655.4
=====================================================
n =  2200
Tiempo iterativo = 105.9  || Tiempo recursivo:  723.2
=====================================================
n =  2400
Tiempo iterativo = 110.7  || Tiempo recursivo:  791.8
=====================================================
n =  2600
Tiempo iterativo = 124.2  || Tiempo recursivo:  864.4
=====================================================
n =  2800
Tiempo iterativo = 132.2  || Tiempo recursivo:  960.4
=====================================================
n =  3000
Tiempo iterativo = 139.7  || Tiempo recursivo:  1028.3
=====================================================
n =  3200
Tiempo iterativo = 147.0  || Tiempo recursivo:  1088.3
=====================================================
n =  3400
Tiempo iterativo = 155.4  || Tiempo recursivo:  1135.2
=====================================================
n =  3600
Tiempo iterativo = 164.4  || Tiempo recursivo:  1200.5
```

```
n =  3800
Tiempo iterativo = 173.4  || Tiempo recursivo:  1250.0
=====================================================
n =  4000
Tiempo iterativo = 183.4  || Tiempo recursivo:  1321.9
=====================================================
n =  4200
Tiempo iterativo = 192.2  || Tiempo recursivo:  1453.4
=====================================================
n =  4400
Tiempo iterativo = 206.0  || Tiempo recursivo:  1489.3
=====================================================
n =  4600
Tiempo iterativo = 207.4  || Tiempo recursivo:  1527.9
=====================================================
n =  4800
Tiempo iterativo = 225.6  || Tiempo recursivo:  1665.5
=====================================================
n =  5000
Tiempo iterativo = 232.4  || Tiempo recursivo:  1781.0
=====================================================
```

Resultados arrojados:

Análisis experimental para la búsqueda de valor máximo		
n	**Máximo Iterativo**	**Máximo Recursivo**
200	8.7	73.6
400	16.8	139.8
600	25.4	197.8
800	32.4	256.8
1000	43.5	324.4
1200	54.3	406.4
1400	63.5	464.5
1600	73.3	537.6
1800	80.5	596.1
2000	97.6	655.4
2200	105.9	723.2
2400	110.7	791.8
2600	124.2	864.4
2800	132.2	960.4
3000	139.7	1028.3
3200	147	1088.3
3400	155.4	1135.2
3600	164.4	1200.5
3800	173.4	1250
4000	183.4	1321.9
4200	192.2	1453.4
4400	206	1489.3
4600	207.4	1527.9
4800	225.6	1665.5
5000	232.4	1781

Tabla 1: Tiempos de ejecución (microsegundos) de 2 algoritmos para la búsqueda del valor máximo

Como es evidente en la tabla, el algoritmo Iterativo tuvo un mejor desempeño general, lo que nos permite decir que es el mejor método en esta prueba.

Algoritmos de ordenamiento

Para este ejercicio, se implementarán los algoritmos Insertion, Bubble, y Merge Sort. Estos tres algoritmos tienen como fin, ordenar una lista de N números. Para este reporte, se tomará N inicial como 1000, y aumentará mil hasta llegar a 10,000.

En este caso, la recursividad no es algo tan común, ya que es utilizada únicamente en el Merge Sort.

Pseudocódigo:

Algorithm 3: Insertion Sort Algorithm

```
Data: A : list of sortable items
1  begin
2      InsertionSort(A)
3      for i ← 2 to n do
4          j ← i − 1;
5          while j ≥ 1 and A[j] > A[j + 1] do
6              swap(A[j], A[j+1]);
7              j ← j − 1;
8      return A;
```

Algorithm 4: Bubble Sort Algorithm

```
Data: A : list of sortable items
1  begin
2      n ← length(A);
3      repeat
4          swapped ← false;
5          for i ← 1 to n do
6              if A[i − 1] > A[i] then
7                  swap(A[i-1], A[i]);
8                  swapped ← true;
9          n ← n − 1;
10     until not swapped;
11     return A;
```

Algorithm 5: Merge Sort Algorithm

```
Data: A : list of sortable items
1  begin
2      MergeSort(A,p,r)
3      if p < r then
4          q ← ⌊(p + r)⌋/2;
5          MergeSort(A,p,q);
6          MergeSort(A,q+1,r);
7          Merge(A, p, q, r)
8      return A;
```

Implementación en Python

```
1.  import random, timeit
2.
3.  import openpyxl
4.
5.  def insertionSort(A,n):
6.      for i in range(1,n):
7.          j = i-1
8.          while j >= 0 and A[j] > A[j+1]:
9.                  A[j + 1],A[j] = A[j],A[j+1]
10.                 j -= 1
11.     return A
12.
13.
14. def bubbleSort(A,n):
15.     while True:
16.         swap = False
17.         for i in range(1,n):
18.             if A[i-1] > A[i]:
19.                 A[i-1],A[i] = A[i], A[i-1]
20.                 swap = True
21.         n -= 1
22.         if swap == False: break
23.     return A
24.
25. def mergeSort(A):
26.     if len(A) == 1:
27.         return A
28.     izq = mergeSort(A[:len(A)//2])
29.     der = mergeSort(A[len(A)//2:])
30.     return merge(izq,der)
31.
32. def merge(izq, der):
33.     temp = []
34.     i = 0
35.     j = 0
36.     while i < len(izq) and j< len(der):
37.         if izq[i] < der[j]:
38.             temp.append(izq[i])
39.             i += 1
40.         else:
41.             temp.append(der[j])
42.             j += 1
43.     if i < len(izq):
44.         while i < len(izq):
45.             temp.append(izq[i])
46.             i += 1
47.     if j < len(der):
48.         while j < len(der):
49.             temp.append(der[j])
50.             j += 1
51.     return temp
52.
53.
```

```
#Funcion que toma un arreglo de números ordenados, y la convierte en un arreglo que satisface el peor caso para Merge Sort
def peorCaso(A):
    if len(A) <= 1:
        return A
    if len(A) == 2:
        return A[::-1]

    izq = []
    der = []

    for i in range(0,len(A),2):
        izq.append(A[i])

    for i in range(1,len(A),2):
        der.append(A[i])

    temp = peorCaso(izq) + peorCaso(der)
    return temp

#Abre el documento.
wb = openpyxl.load_workbook("pruebas.xlsx")
#Selecciona la hoja en la cuál escribir.
hoja = wb['Ejec2']

#Insertion Sort
print("==============================INSERTION SORT:==============================")
for i in range(1,11):
    A=[]
    n = i*1000
    print("\t\t\tPara n = ", n)

    #Lista A, de N numeros, ordenada. (Mejor caso para Bubble, Insertion y Merge)
    for j in range(1,n+1):A.append(j)
    #Lista A invertida (Peor caso para Insertion y Bubble)
    B= A[::-1]
    #Lista A desordenada (Caso promedio)
    C = random.sample(A,len(A))

    #Mejor Caso
    print("Ejecutando: Insertion Mejor Caso", end=" ")
    tiempo1IS = round((timeit.timeit(lambda: insertionSort(A,n) , number=1)*100000), 3)
    hoja.cell(row=(i+3),column=3,value=tiempo1IS)
    print("=> EJECUCIÓN FINALIZADA tiempo =", tiempo1IS)

    #Caso Promedio
    print("Ejecutando: Insertion Caso Promedio", end=" ")
            tiempo2IS = round((timeit.timeit(lambda: insertionSort(C,n) , number=1)*100000), 3)
            hoja.cell(row=(i+3),column=6,value=tiempo2IS)
            print("=> EJECUCIÓN FINALIZADA tiempo =", tiempo2IS)
```

```
            #Peor Caso
            print("Ejecutando: Insertion Peor Caso", end=" ")
            tiempo3IS = round((timeit.timeit(lambda: insertionSort(B,n) , number=1
    )*100000), 3)
            hoja.cell(row=(i+3),column=9,value=tiempo3IS)
            print("=> EJECUCIÓN FINALIZADA tiempo =", tiempo3IS)

        #Bubble Sort
        print("============================BUBBLE SORT:=========================
    =====")
        for i in range(1,11):
            A=[]
            n = i*1000
            print("\t\t\tPara n = ", n)

            #Lista A, de N numeros, ordenada. (Mejor caso para Bubble, Insertion y
     Merge)
            for j in range(1,n+1):A.append(j)

            #Lista A invertida (Peor caso para Insertion y Bubble)
            B= A[::-1]

            #Lista A desordenada (Caso promedio)
            C = random.sample(A,len(A))

            #Mejor Caso
            print(f"Ejecutando: Bubble Mejor Caso", end=" ")
            tiempo1BS = round((timeit.timeit(lambda: bubbleSort(A,n) , number=1)*1
    00000), 3)
            hoja.cell(row=(i+3),column=2,value=tiempo1BS)
            print("=> EJECUCIÓN FINALIZADA tiempo =", tiempo1BS)

            #Caso Promedio
            print("Ejecutando: Bubble Caso Promedio", end=" ")
            tiempo2BS = round((timeit.timeit(lambda: bubbleSort(C,n) , number=1)*1
    00000), 3)
            hoja.cell(row=(i+3),column=5,value=tiempo2BS)
            print("=> EJECUCIÓN FINALIZADA tiempo =", tiempo2BS)

            #Peor Caso
            print("Ejecutando: Bubble Peor Caso", end=" ")
            tiempo3BS = round((timeit.timeit(lambda: bubbleSort(B,n) , number=1)*1
    00000), 3)
            hoja.cell(row=(i+3),column=8,value=tiempo3BS)
            print("=> EJECUCIÓN FINALIZADA tiempo =", tiempo3BS)
```

```
142.        #Merge Sort
143.        print("============================MERGE SORT:=========================
    ====")
144.        for i in range(1,11):
145.            A=[]
146.            n = i*1000
147.            print("\t\t\tPara n = ", n)
148.
149.            #Lista A, de N numeros, ordenada. (Mejor caso para Bubble, Insertion y
     Merge)
150.            for j in range(1,n+1):A.append(j)
151.
152.            #Lista A desordenada (Caso promedio)
153.            C = random.sample(A,len(A))
154.            #Lista A desordenada de cierta manera en la función peorCaso(Peor Caso
     Merge)
155.            E= peorCaso(A)
156.
157.            #Mejor Caso
158.            print("Ejecutando: Merge Mejor Caso", end=" ")
159.            tiempo1MS = round((timeit.timeit(lambda: mergeSort(A) , number=1)*1000
    00), 3)
160.            hoja.cell(row=(i+3),column=4,value=tiempo1MS)
161.            print("=> EJECUCIÓN FINALIZADA tiempo =", tiempo1MS)
162.
163.            #Caso Promedio
164.            print("Ejecutando: Merge Caso Promedio", end=" ")
165.            tiempo2MS = round((timeit.timeit(lambda: mergeSort(C) , number=1)*1000
    00), 3)
166.            hoja.cell(row=(i+3),column=7,value=tiempo2MS)
167.            print("=> EJECUCIÓN FINALIZADA tiempo =", tiempo2MS)
168.
169.            #Peor Caso
170.            print("Ejecutando: Merge Peor Caso", end=" ")
171.            tiempo3MS = round((timeit.timeit(lambda: mergeSort(E) , number=1)*1000
    00), 3)
172.            hoja.cell(row=(i+3),column=10,value=tiempo3MS)
173.            print("=> EJECUCIÓN FINALIZADA tiempo =", tiempo3MS)
174.
175.        print("=====================================================")
176.
177.        wb.save("pruebas.xlsx")
```

Pantallas de prueba:

```
============================INSERTION SORT:=============================
                         Para n =  1000
Ejecutando: Insertion Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 9.41
Ejecutando: Insertion Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 4414.58
Ejecutando: Insertion Peor Caso => EJECUCIÓN FINALIZADA tiempo = 8663.92
                         Para n =  2000
Ejecutando: Insertion Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 20.57
Ejecutando: Insertion Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 17923.24
Ejecutando: Insertion Peor Caso => EJECUCIÓN FINALIZADA tiempo = 36485.39
                         Para n =  3000
Ejecutando: Insertion Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 29.31
Ejecutando: Insertion Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 42819.11
Ejecutando: Insertion Peor Caso => EJECUCIÓN FINALIZADA tiempo = 79943.89
                         Para n =  4000
Ejecutando: Insertion Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 39.27
Ejecutando: Insertion Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 77061.41
Ejecutando: Insertion Peor Caso => EJECUCIÓN FINALIZADA tiempo = 146083.19
                         Para n =  5000
Ejecutando: Insertion Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 51.85
Ejecutando: Insertion Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 112920.41
Ejecutando: Insertion Peor Caso => EJECUCIÓN FINALIZADA tiempo = 234229.52
                         Para n =  6000
Ejecutando: Insertion Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 61.32
Ejecutando: Insertion Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 167824.12
Ejecutando: Insertion Peor Caso => EJECUCIÓN FINALIZADA tiempo = 321717.78
                         Para n =  7000
Ejecutando: Insertion Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 69.68
Ejecutando: Insertion Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 223885.93
Ejecutando: Insertion Peor Caso => EJECUCIÓN FINALIZADA tiempo = 444818.18
                         Para n =  8000
Ejecutando: Insertion Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 87.92
Ejecutando: Insertion Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 291731.57
Ejecutando: Insertion Peor Caso => EJECUCIÓN FINALIZADA tiempo = 582252.6
                         Para n =  9000
Ejecutando: Insertion Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 92.73
Ejecutando: Insertion Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 370179.19
Ejecutando: Insertion Peor Caso => EJECUCIÓN FINALIZADA tiempo = 739282.36
                         Para n =  10000
Ejecutando: Insertion Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 101.31
Ejecutando: Insertion Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 453350.49
Ejecutando: Insertion Peor Caso => EJECUCIÓN FINALIZADA tiempo = 911480.04
```

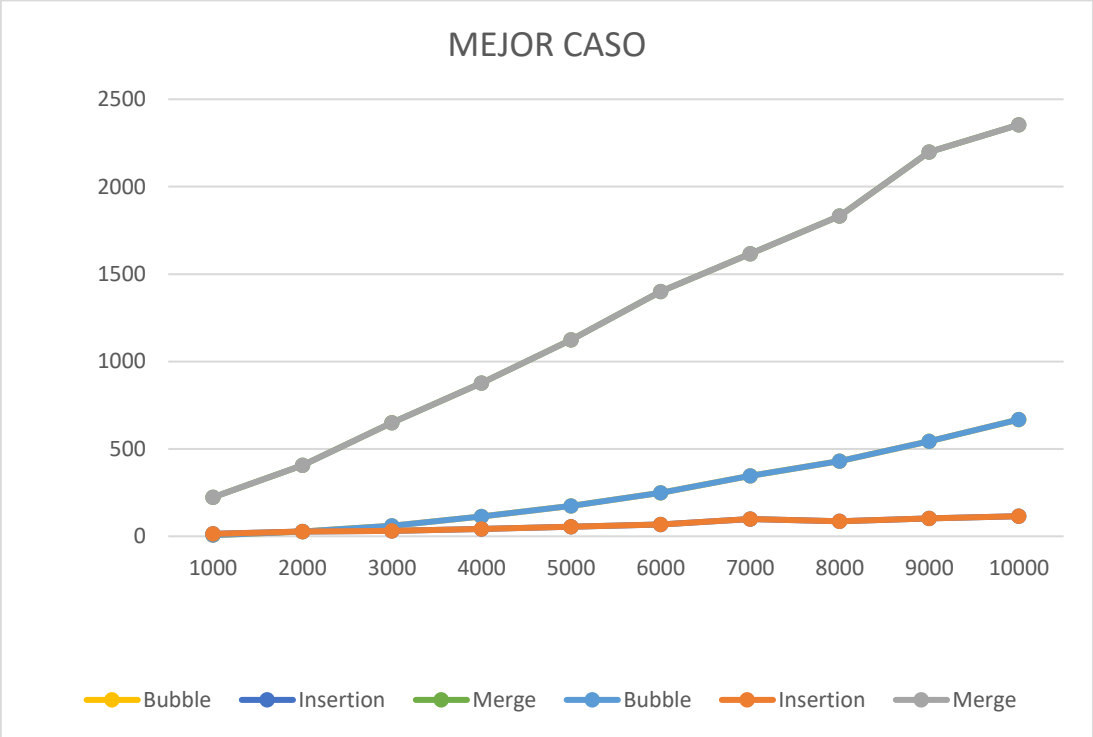
```
============================BUBBLE SORT:============================
                        Para n =  1000
Ejecutando: Bubble Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 7.29
Ejecutando: Bubble Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 5407.29
Ejecutando: Bubble Peor Caso => EJECUCIÓN FINALIZADA tiempo = 7194.71
                        Para n =  2000
Ejecutando: Bubble Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 13.09
Ejecutando: Bubble Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 22479.19
Ejecutando: Bubble Peor Caso => EJECUCIÓN FINALIZADA tiempo = 31190.16
                        Para n =  3000
Ejecutando: Bubble Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 21.42
Ejecutando: Bubble Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 52226.0
Ejecutando: Bubble Peor Caso => EJECUCIÓN FINALIZADA tiempo = 77547.0
                        Para n =  4000
Ejecutando: Bubble Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 26.43
Ejecutando: Bubble Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 95308.55
Ejecutando: Bubble Peor Caso => EJECUCIÓN FINALIZADA tiempo = 129144.24
                        Para n =  5000
Ejecutando: Bubble Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 33.11
Ejecutando: Bubble Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 146571.6
Ejecutando: Bubble Peor Caso => EJECUCIÓN FINALIZADA tiempo = 203983.34
                        Para n =  6000
Ejecutando: Bubble Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 39.85
Ejecutando: Bubble Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 215774.2
Ejecutando: Bubble Peor Caso => EJECUCIÓN FINALIZADA tiempo = 299445.59
                        Para n =  7000
Ejecutando: Bubble Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 48.57
Ejecutando: Bubble Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 288880.2
Ejecutando: Bubble Peor Caso => EJECUCIÓN FINALIZADA tiempo = 404723.63
                        Para n =  8000
Ejecutando: Bubble Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 53.79
Ejecutando: Bubble Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 383677.97
Ejecutando: Bubble Peor Caso => EJECUCIÓN FINALIZADA tiempo = 532177.41
                        Para n =  9000
Ejecutando: Bubble Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 60.56
Ejecutando: Bubble Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 486401.7
Ejecutando: Bubble Peor Caso => EJECUCIÓN FINALIZADA tiempo = 672168.51
                        Para n =  10000
Ejecutando: Bubble Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 67.58
Ejecutando: Bubble Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 597385.81
Ejecutando: Bubble Peor Caso => EJECUCIÓN FINALIZADA tiempo = 833330.45
                                MERGE SORT:
```

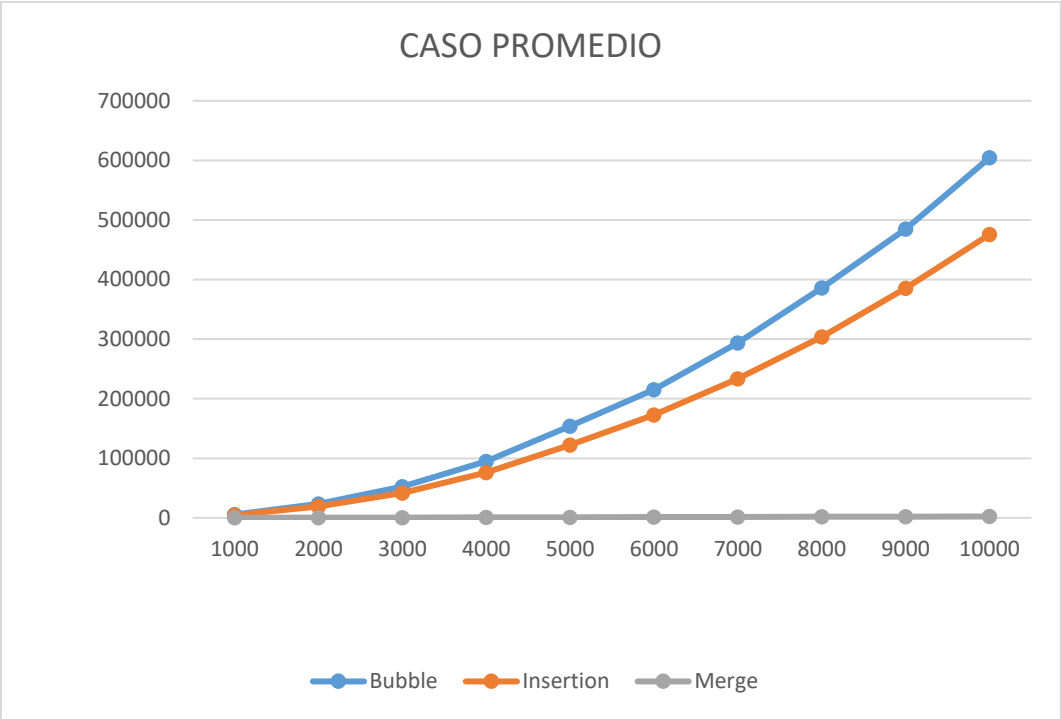
```
============================MERGE SORT:============================
                        Para n =  1000
Ejecutando: Merge Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 220.07
Ejecutando: Merge Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 206.98
Ejecutando: Merge Peor Caso => EJECUCIÓN FINALIZADA tiempo = 204.61
                        Para n =  2000
Ejecutando: Merge Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 391.58
Ejecutando: Merge Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 447.4
Ejecutando: Merge Peor Caso => EJECUCIÓN FINALIZADA tiempo = 446.12
                        Para n =  3000
Ejecutando: Merge Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 614.21
Ejecutando: Merge Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 743.03
Ejecutando: Merge Peor Caso => EJECUCIÓN FINALIZADA tiempo = 765.41
                        Para n =  4000
Ejecutando: Merge Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 818.37
Ejecutando: Merge Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 1015.9
Ejecutando: Merge Peor Caso => EJECUCIÓN FINALIZADA tiempo = 997.52
                        Para n =  5000
Ejecutando: Merge Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 1095.27
Ejecutando: Merge Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 1335.09
Ejecutando: Merge Peor Caso => EJECUCIÓN FINALIZADA tiempo = 1387.44
                        Para n =  6000
Ejecutando: Merge Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 1337.43
Ejecutando: Merge Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 1606.0
Ejecutando: Merge Peor Caso => EJECUCIÓN FINALIZADA tiempo = 1618.18
                        Para n =  7000
Ejecutando: Merge Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 1563.86
Ejecutando: Merge Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 1886.99
Ejecutando: Merge Peor Caso => EJECUCIÓN FINALIZADA tiempo = 1872.04
                        Para n =  8000
Ejecutando: Merge Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 1783.11
Ejecutando: Merge Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 2168.92
Ejecutando: Merge Peor Caso => EJECUCIÓN FINALIZADA tiempo = 2154.66
                        Para n =  9000
Ejecutando: Merge Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 2053.86
Ejecutando: Merge Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 2479.84
Ejecutando: Merge Peor Caso => EJECUCIÓN FINALIZADA tiempo = 2470.83
                        Para n =  10000
Ejecutando: Merge Mejor Caso => EJECUCIÓN FINALIZADA tiempo = 2260.08
Ejecutando: Merge Caso Promedio => EJECUCIÓN FINALIZADA tiempo = 2892.98
Ejecutando: Merge Peor Caso => EJECUCIÓN FINALIZADA tiempo = 2757.03
```

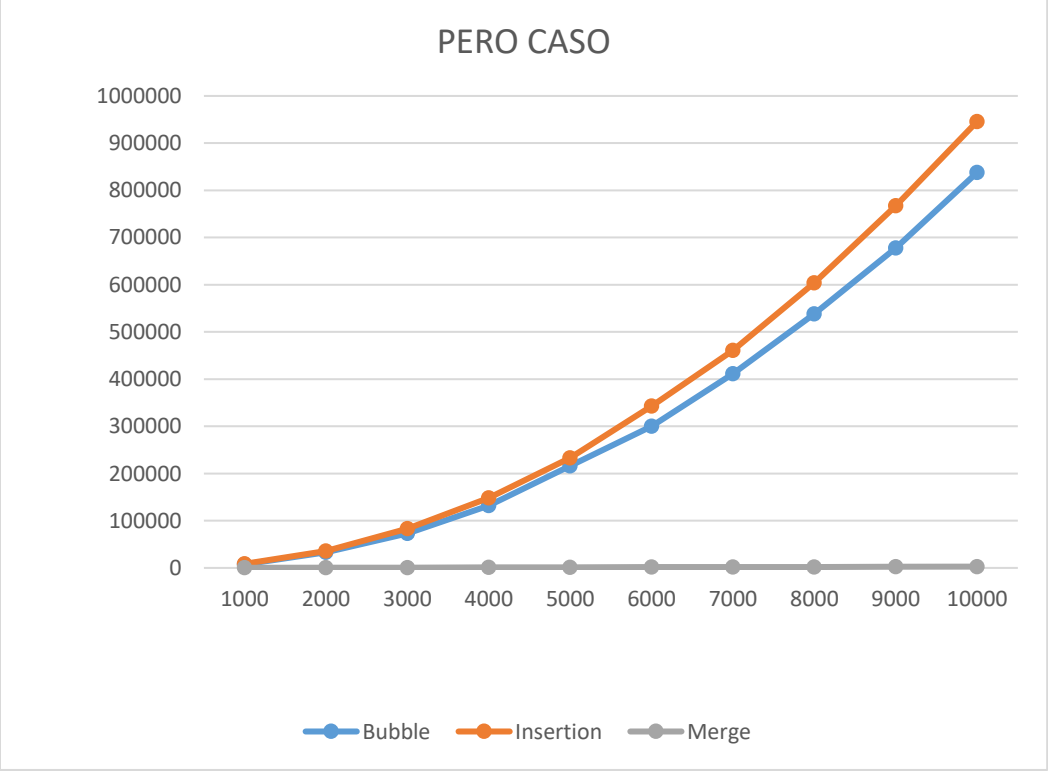
Resultados arrojados:

ANÁLISIS EXPERIMENTAL DE ALGORTIMOS DE ORDENAMIENTO									
	Mejor Caso			Caso Promedio			Peor Caso		
n	Bubble	Insertion	Merge	Bubble	Insertion	Merge	Bubble	Insertion	Merge
1000	6.59	9.78	207.69	5627.67	4697.71	212.85	7444.84	8721.36	264.28
2000	26.05	20.49	379.53	23678.64	19078.24	394.63	33016.98	36066.26	464.35
3000	59.75	31.56	597.01	52664.05	41752.14	648.26	73334.49	83174.71	728.58
4000	105.1	61.53	813.32	94973.7	76157.26	858.48	132101.03	148590.74	995.04
5000	166.64	61.53	1089.12	153924.21	122098.29	1117.39	216062.77	232974.13	1339.73
6000	244.38	63.35	1283.13	215124.19	172875.35	1276.41	300162.38	343211.9	1557.31
7000	331.24	76.85	1503.38	293827.59	233336.44	1574.98	411576.91	461126.78	1854.66
8000	429.84	83.62	1747.66	386045.08	303646.22	1783.93	538028.38	604155.37	2118.27
9000	543.25	95.97	2006.59	484831.4	385506.65	2067.62	678080.26	767313.59	2478.28
10000	678.49	103.91	2289.58	604398.84	475526.66	2325.32	837933.68	945943.31	2792.44

Tabla 1: Tiempos de ejecución (microsegundos) de 2 algoritmos para la búsqueda del valor máximo




CASO PROMEDIO
700000
600000
500000
400000
300000
200000
100000
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Bubble
Insertion
Merge


PERO CASO
1000000
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Bubble
Insertion
Merge

De manera teórica, revisamos la complejidad de los algoritmos, la cual se refleja en la siguiente tabla.

	Bubble Sort	Insertion Sort	Merge Sort
Mejor Caso	O(n)	O(n)	O(nLogn)
Caso Promedio	$O(n^2)$	$O(n^2)$	O(nLogn)
Peor Caso	$O(n^2)$	$O(n^2)$	O(nLogn)

Tomando en cuenta los resultados de la prueba, y graficándolos, nos podemos percatar que la representación gráfica coincide con su respectiva complejidad dentro de la tabla.

Conforme la cantidad de números dentro del arreglo, podemos observar cómo la complejidad tanto para Bubble e Insertion Sort, aumenta exponencialmente. Esto es más evidente en el peor caso. Precisamente en este último caso podemos observar la diferencia de comportamiento del método Merge sort con los demás. Conforme se le exige más al programa, las gráficas de Bubble e Insertion sort se despegan a tal punto de hacer ver a la gráfica del Merge sort como si fuese lineal.

Sin embargo, en los casos ideales, el comportamiento de los algoritmos iterativos fue mejor.

Conclusión:

Con esta práctica pudimos ver cómo se comportan diferentes algoritmos haciendo sus respectivas tareas y qué tan convenientes pueden ser.
En el caso del primer ejercicio, es evidente que el algoritmo iterativo tiene un mejor desempeño ya que requiere menos tiempo de ejecucu;on para encontrar el número mayor.

En el caso del segundo ejercicio, varía mucho dependiendo de la situación. Basándonos en las pruebas realizadas, en un caso ideal (lo cuál es bastante complicado que se encuentre en una implementación real) el mejor de todos es Insertion Sort, debido a que tuvo un menor tiempo de ejecución. En general, Bubble e Insertion Sort tienen un desempeñó muy similar. En los demás casos, Merge sort es el método a elegir. Como sabemos, Merge sort tiene una complejidad constante independientemente del caso. Esto lo hace muy conveniente cuando tratamos con grandes cantidades de datos, como lo fue en la pruebas.

Referencias:

Wikipedia contributors. (2021a, julio 29). *Insertion sort*. Wikipedia.

https://en.wikipedia.org/wiki/Insertion_sort#Best,_worst,_and_average_cases

Wikipedia contributors. (2021b, septiembre 3). *Merge sort*. Wikipedia.

https://en.wikipedia.org/wiki/Merge_sort

Wikipedia contributors. (2021c, septiembre 15). *Bubble sort*. Wikipedia.

https://en.wikipedia.org/wiki/Bubble_sort