Universidad Nacional de San Agustín de Arequipa

Escuela Profesional de Ciencia de la Computación

Algoritmos y Estructuras de Datos

Laboratorio Nro. 01

Presentado por:
Fiorela Villarroel Ramos

Docente :
Rolando Jesus Cardenas Talavera

1. Competencia del Curso

Comprende la importancia e impacto de los algoritmos estudiados y las nuevas propuestas.

2. Competencia del Laboratorio

- Describir, implementar y analizar algoritmos de ordenamiento.
- Interpretar el costo computacional en algoritmos de estudio.

3. Equipos y Materiales

- Un computador.
- Lenguaje de Programación (c++, python, java, c)

4. Actividad

4.1. Ejercicio 1

Genere un archivo con números aleatorios (mayor a un millón), este representará el vector a ordenar en las pruebas de los algoritmos de ordenamiento, un número puede repetirse más de una vez

```cpp
#include <bits/stdc++.h>

using namespace std;

#define f(i, a, b) for (int i = a; i < b; i++)

int main()
{

  ofstream datos1("Entrada/datos1.txt");
  ofstream datos2("Entrada/datos2.txt");
  ofstream datos3("Entrada/datos3.txt");
  ofstream datos4("Entrada/datos4.txt");
  ofstream datos5("Entrada/datos5.txt");
```

```
15
16  int tams[11] = {100,
17                  200,
18                  500,
19                  800,
20                  1000,
21                  2000,
22                  5000,
23                  8000,
24                  10000,
25                  20000,
26                  50000};
27
28  int tams2[3] = {80000,
29                  100000,
30                  200000
31                  };
32  f(i, 0, 11)
33  {
34    datos1 << tams[i] << "\n";
35    f(j, 0, tams[i]) datos1 << rand() << " ";
36    datos1 << "\n";
37  }
38  datos1.close();
39
40  f(i, 0, 3)
41  {
42    datos2 << tams2[i] << "\n";
43    f(j, 0, tams2[i]) datos2 << rand() << " ";
44    datos2 << "\n";
45  }
46  datos2.close();
47
48  datos3 << 500000 << "\n";
49  f(j, 0, 500000) datos3 << rand() << " ";
50  datos3 << "\n";
51  datos3.close();
52
53  datos4 << 800000 << "\n";
54  f(j, 0,800000) datos4 << rand() << " ";
55  datos4 << "\n";
56  datos4.close();
57
58  datos5 << 1000000 << "\n";
59  f(j, 0,1000000) datos5 << rand() << " ";
60  datos5 << "\n";
61  datos5.close();
```

```
62   return 0;
63 }
```

```
100
1804289383 846930886 1681692777 1714636915 1957747793 424238335
. . . .
500
1036140795 463480570 2040651434 1975960378 317097467 1892066601
1376710097 927612902 1330573317 603570492 1687926652 660260756
. . . .
1000
1635550270 2069110699 712633417 864101839 1204275569 1190668363
1336092622 410228794 1026413173 773319847 1404196431 1968217462
. . . .
5000
. . . .
8000
. . . .
10000
. . . .
20000
. . . .
50000
. . . .
80000
. . . .
100000
. . . .
200000
. . . .
500000
. . . .
800000
. . . .
1000000
. . . .
```

4.2. Ejercicio 2

Implemente los siguientes algoritmos:

- Bubble sort
- Heap sort

- Insertion sort
- Selection sort
- Shell sort
- Merge sort
- Quick sort

4.2.1. Bubble Sort

Es el algoritmo de clasificación más simple que funciona intercambiando repetidamente los elementos adyacentes si estan en el orden incorrecto. Debido a su complejidad este algoritmo no es adecuado para grandes conjuntos de datos.

Comlejidad peor de los casos : $\mathcal{O}(n^2)$

Espacio Auxiliar: $\mathcal{O}(1)$

```cpp
#include <bits/stdc++.h>

using namespace std;

#define f(i, a, b) for (int i = a; i < b; i++)

void swap(int *a, int *b)
{
  int aux = *a;
  *a = *b;
  *b = aux;
}

void bubbleSort(int A[], int tam)
{
  f(i, 0, tam - 1)
      f(j, 0, tam - i - 1)
        if (A[j] > A[j + 1])
           swap(&A[j], &A[j + 1]);
}
```

4.2.2. Heap Sort

Este algoritmo hace uso de una estructura de datos llamada **heap** , para administrar la información y se representa como un **arreglo**.

Para el desarrollo de este algoritmo se usa el **max-heap** el cual tiene la propiedad de que el nodo padre es mayor que los nodos hijos , además las operaciones básicas de los **heaps** se ejecutan en un tiempo proporcional a la altura del árbol y toma un tiempo de *O(lg n)*.

Operaciones Básicas

- **Max-Heapify** Se ejecuta en un tiempo de *O(lg n)* el cual se encarga de mantener la propiedad del Max-Heap.
- **Buil-Max-Heap** Se ejecuta en un tiempo lineal.
- **HeapSort** Se ejecuta en un tiempo de *O(n lg n)*

```
1     #include <bits/stdc++.h>
2  #include <time.h>
3  #include <iostream>
4  #include <fstream>
5
6  using namespace std;
7
8  #define f(i, a, b) for (int i = a; i < b; i++)
9  #define INF std::numeric_limits<int>::max();
10
11 bool check(int A[], int tam)
12 {
13   f(i, 1, tam) if (A[i] < A[i - 1]) return 0;
14
15   return 1;
16 }
17
18 void swap(int *a, int *b)
19 {
20   int aux = *a;
21   *a = *b;
22   *b = aux;
23 }
24 int LEFT(int i)
25 {
26   return (i << 1);
27 }
```

```
28
29 int RIGHT(int i)
30 {
31   return (i << 1) + 1;
32 }
33
34 void MaxHeapify(int A[], int tam, int i)
35 {
36   int l = LEFT(i);
37   int r = RIGHT(i);
38
39   int largest;
40   if (l < tam && (A[l] > A[i]))
41     largest = l;
42   else
43     largest = i;
44
45   if (r < tam && (A[r] > A[largest]))
46     largest = r;
47
48   if (largest != i)
49   {
50     swap(&A[i], &A[largest]);
51     MaxHeapify(A, tam, largest);
52   }
53 }
54
55 void BuildMaxHeap(int A[], int tam)
56 {
57   for (int i = tam / 2 - 1; i >= 0; i--)
58   {
59     MaxHeapify(A, tam, i);
60   }
61 }
62
63 void heapSort(int A[], int tam)
64 {
65   BuildMaxHeap(A, tam);
66   for (int i = tam - 1; i >= 0; i--)
67   {
68     swap(&A[0], &A[i]);
69     MaxHeapify(A, i, 0);
70   }
71 }
```

4.2.3. Insertion sort

El arreglo se divide abstractamente en una parte ordenada y en otro no ordenada , los valores de la parte desordenada se seleccionan y se colocan en la parte ordenada.

Características:

- Simple implementación
- Eficientes para cantidad de datos pequeños
- De naturaleza adaptativa , es decir que es adecuada para valores de datos pequeños.

4.2.3.1. Ejemplo: Tomaremos el críterio ascendente.

$$arr[] = \mathbf{12}\ ,\ \mathbf{11}, 13, 5, 6$$

1. En el primer paso los dos primeros elementos de la matriz se comparan , podemos observar que 12 es mayor que 11 , por lo tanto como estan ubicados de manera equívoca , los intercambiamos , por lo que por ahora 11 se encuentra en un arreglo ordenado.

$$arr[] = 11, \mathbf{12}\ ,\ \mathbf{13}, 5, 6$$

2. Comparamos el 12 y 13 y observamos que estan en la posición correcto entonces el 12 permanece a un subconjunto ordenado junto con el elemento 11.

$$arr[] = 11, 12, \mathbf{13}\ ,\ \mathbf{5}, 6$$

3. Ahora comparamos el 13 y 5 como no esta ubicados correctamente , entonces los intercambiamos , luego el 5 con el 12 , como tienen el mismo criterio anteriro lo intercambiamos , después el 11 con el 5 , hasta que el 5 este en sus posición correcta.

$$arr[] = 5, 11, 12, \mathbf{13}\ ,\ \mathbf{6}$$

4. En el último paso hacemos lo mismo con el 6 hasta que se ubique en su posición correcta.

```cpp
#include <bits/stdc++.h>
#include <time.h>
#include <iostream>
#include <fstream>

using namespace std;

#define f(i, a, b) for (int i = a; i < b; i++)
```

```
#define INF std::numeric_limits<int>::max();

bool check(int A[], int tam)
{
  f(i, 1, tam) if (A[i] < A[i - 1]) return 0;

  return 1;
}

void swap(int *a, int *b)
{
  int aux = *a;
  *a = *b;
  *b = aux;
}

void insertionSort(int A[], int n)
{
  for (int i = 1; i < n; i++)
  {
    int key = A[i];
    int j = i - 1;
    while (j >= 0 && A[j] > key)
    {
      A[j + 1] = A[j];
      j--;
    }
    A[j + 1] = key;
  }
}

```

4.2.4. Selection sort

El algoritmo consiste encuentra el elemento mínimo de la parte no ordenada y colocarlo al principio , mateniendo asi dos subarreglos del arreglo original , la parte del arreglo ordenado y el subarreglo restante que no se clásifico, su complejidad de $\mathcal{O}(n^2)$ lo hace más ineficientes en listas grandes

```
#include <bits/stdc++.h>
#include <time.h>
#include <iostream>
#include <fstream>
```

```cpp

using namespace std;

#define f(i, a, b) for (int i = a; i < b; i++)
#define INF std::numeric_limits<int>::max();

bool check(int A[], int tam)
{
  f(i, 1, tam) if (A[i] < A[i - 1]) return 0;

  return 1;
}

void swap(int *a, int *b)
{
  int aux = *a;
  *a = *b;
  *b = aux;
}

void selectionSort(int A[], int tam)
{
  int ind;
  f(i, 0, tam - 1)
  {
    ind = i;
    f(j, i + 1, tam)
    {
      if (A[j] < A[ind])
      {
        ind = j;
      }
    }

    if (ind != i)
      swap(&A[i], &A[ind]);
  }
}

int main()
{
  ifstream datos("Entrada/datos3.txt");
  ofstream salida;

  salida.open("Salida/selectionSort.txt");
  int N;
  while (datos >> N)
```

```
52    {
53      int A[N];
54      f(i, 0, N) datos >> A[i];
55
56      salida << check(A, N) << " ";
57      cout << check(A, N) << " ";
58      salida << N << " ";
59      cout << N << " ";
60
61      auto start2 = std::chrono::steady_clock::now();
62      selectionSort(A, N);
63      auto end2 = std::chrono::steady_clock::now();
64      std::chrono::duration<double> elapsed_seconds2 = end2 - start2;
65      salida << elapsed_seconds2.count() * 1000 << ' ';
66      cout << elapsed_seconds2.count() * 1000 << " ";
67
68      salida
69          << check(A, N) << "\n";
70
71      cout
72          << check(A, N) << "\n";
73    }
74    datos.close();
75    salida.close();
76    return 0;
77  }
```

4.3. Shell sort

Es principalmete en una variación del insertion sort , la diferencia es que el shell sort toma rangos de comparación más grandes , este rango empieza hacer **h** y se va reduciendo hasta que llegue a ser 1.

4.3.1. Aplicaciones:

- Para llamar a la sobrecarga de la pila , usamos ordenación de shell.
- Cuando la recursión excede un límite particular.
- Para conjuntos de tamaño mediano a grande.

```
1 #include <bits/stdc++.h>
2 #include <time.h>
```

```cpp
#include <iostream>
#include <fstream>

using namespace std;

#define f(i, a, b) for (int i = a; i < b; i++)
#define INF std::numeric_limits<int>::max();

bool check(int A[], int tam)
{
  f(i, 1, tam) if (A[i] < A[i - 1]) return 0;

  return 1;
}

void swap(int *a, int *b)
{
  int aux = *a;
  *a = *b;
  *b = aux;
}

void shellSort(int A[], int tam)
{
  for (int gap = tam / 2; gap > 0; gap /= 2)
  {
    for (int i = 0, m; i < tam - gap; i++)
    {
      m = i;
      while (m >= 0 && A[m + gap] < A[m])
      {
        swap(&A[m], &A[m + gap]);
        m = m - gap;
      }
    }
  }
}
```

4.3.2. Merge sort

El algoritmo Merge Sort es un algoritmo de clasificación que se basa en el paradigma Divide and Conquer . En este algoritmo, el arreglo se divide inicialmente en dos mitades iguales y luego se combinan de manera ordenada.

Un algoritmo recursivo que divide continuamente el arreglo por la mitad hasta

que ya no se puede dividir más. Esto significa que si el arreglo esta vacío o solo le queda un elemento, la división se detendrá, es decir, es el caso base para detener la recursividad. Si el arreglo tiene varios elementos, se divide el arreglo en mitades y se invoca la recursión , para después hacer la ordenación por fusión en cada una de las mitades. Finalmente, cuando ambas mitades están ordenadas, se aplica la operación de fusión. La operación de fusión es el proceso de tomar dos arreglos ordenados más pequeños y combinarlos para eventualmente formar uno más grande.

```cpp
#include <bits/stdc++.h>
#include <time.h>
#include <iostream>
#include <fstream>

using namespace std;

#define f(i, a, b) for (int i = a; i < b; i++)
#define f_2(i, a, b) for (int i = a; i < b; i = i * 2)
#define MIN(a, b) ((a < b) ? a : b)
#define INF std::numeric_limits<int>::max();

bool check(int A[], int tam)
{
  f(i, 1, tam) if (A[i] < A[i - 1]) return 0;

  return 1;
}

void swap(int *a, int *b)
{
  int aux = *a;
  *a = *b;
  *b = aux;
}

void merge(int A[], int p, int q, int r)
{
  int n1 = q - p + 1;
  int n2 = r - q;

  int L[n1 + 1], R[n2 + 1];

  f(i, 0, n1) L[i] = A[p + i];
  f(i, 0, n2) R[i] = A[q + i + 1];

  L[n1] = INF;
  R[n2] = INF;
```

```
  int i = 0, j = 0;
  f(k, p, r + 1)
  {
    if (L[i] <= R[j])
    {
      A[k] = L[i];
      i++;
    }
    else
    {
      A[k] = R[j];
      j++;
    }
  }
}

void mergeSort(int A[], int p, int r)
{
  if (p < r)
  {
    int q = (p + r) / 2;
    mergeSort(A, p, q); // 0 2 ->
    mergeSort(A, q + 1, r);
    merge(A, p, q, r);
  }
}
```

4.3.3. Quick sort

Al igual que Merge Sort , QuickSort es un algoritmo Divide and Conquer . Selecciona un elemento como pivote y divide la matriz dada alrededor del pivote elegido. .

- Elegir siempre el primer elemento como pivote.
- Elegir siempre el último elemento como pivote en nuestro caso de implementacion.
- Elegir un elemento aleatorio como pivote.
- Elegir la mediana como pivote.

El proceso clave en QuickSort es una **partición()**. El objetivo de las particiones es, dado un arreglo y un elemento x de un arreglo como pivote, colocar x en su posición correcta en un arreglo ordenado y colocar todos los elementos más pequeños antes de x, y colocar todos los elementos mayores después de x. Todo esto debe hacerse en tiempo lineal.

```cpp
#include <bits/stdc++.h>
#include <time.h>
#include <iostream>
#include <fstream>

using namespace std;

#define f(i, a, b) for (int i = a; i < b; i++)
#define INF std::numeric_limits<int>::max();

bool check(int A[], int tam)
{
  f(i, 1, tam) if (A[i] < A[i - 1]) return 0;

  return 1;
}

void swap(int *a, int *b)
{
  int aux = *a;
  *a = *b;
  *b = aux;
}

int Partition(int A[], int p, int r)
{
  int x = A[r];
  int i = p - 1;
  for (int j = p; j < r; j++)
  {
    if (A[j] <= x)
    {
      i++;
      swap(&A[i], &A[j]);
    }
  }

  swap(&A[i + 1], &A[r]);
  return i + 1;
}

void quickSort(int A[], int p, int r)
{
  if (p < r)
  {
    int q = Partition(A, p, r);
```

```
47      quickSort(A, p, q - 1);
48      quickSort(A, q + 1, r);
49    }
50  }
51
```

5. Ejercicio 3

Analizar la complejidad computacional de cada uno.

5.1. Bubble Sort:

```
1  void swap(int *a, int *b)
2  {
3    int aux = *a; //1 c1
4    *a = *b; //1 c2
5    *b = aux; //1 c3
6  }
7
8  void bubbleSort(int A[], int tam)
9  {
10   f(i, 0, tam - 1)  //tam c4
11       f(j, 0, tam - i - 1)  // (tam-i)  c5
12         if (A[j] > A[j + 1]) // 1 c6
13             swap(&A[j], &A[j + 1]); //  1 c7
14 }
```

El algoritmo tiene que realizar siempre el mismo numero de comparaciones por los dos bucles:

$$T(n) = \frac{(n^2 - n)}{2}$$

Como tomaremos la notación asintótica del peor de los casos entonces tomamos el n más grande , puesto que los demás elementos seran insignificantes para el crecimiento cuadrático del algoritmo.

por lo tanto :

$$T(n) = n^2$$

Complejidad en el peor de los casos : $\mathcal{O}(n^2)$

Complejidad en el mejor de los casos : $\Omega(n^2)$

5.2. Selection sort

```
1  void swap(int *a, int *b)
2  {
3    int aux = *a;
4    *a = *b;
5    *b = aux;
6  }
7
8  void selectionSort(int A[], int tam)
9  {
10   int ind;
11   f(i, 0, tam - 1) // tam c2
12   {
13     ind = i; // 1 c1
14     f(j, i + 1, tam) //tam - i + 2 c3
15     {
16       if (A[j] < A[ind]) // 1 c4
17         ind = j;
18     }
19     if (ind != i) // 1 c6
20       swap(&A[i], &A[ind]);
21   }
22 }
```

Como se puede observar en el seguno bucle este depende de las iteraciones del anterior bucle , por lo tanto seleccionar el mínimo requiere recorrer todo el arreglo de elementos , como una aprte ya se ordena entonces las siguientes comparaciones serásn unas **n-2** y asy susesivamente ya que vamos trabajando con el arreglo no ordenado , por lo tanto :

$$T(n) = c_1(n-1) + c_2(n-3) + ... + 1 = \sum_{i=1}^{n-1}(i)$$

$$T(n) = \frac{1}{2}(n^2 - n)$$

Que da como resultado una complejidad de $O(n^2)$

5.3. Insertion sort

```
1  void swap(int *a, int *b)
2  {
3    int aux = *a;
4    *a = *b;
5    *b = aux;
6  }
7
8  void insertionSort(int A[], int n)
9  {
10   for (int i = 1; i < n; i++)
11   {
12     int key = A[i];
13     int j = i - 1;
14     while (j >= 0 && A[j] > key)
15     {
16       A[j + 1] = A[j];
17       j--;
18     }
19     A[j + 1] = key;
20   }
21 }
```

$$T(n) = c * 1 + c * 2 + c * 3 + c(n-1)$$
$$T(n) = c * (1 + 2 + 3 + +(n-1))$$

Esa suma es una serie aritmética, que va hasta n-1, usando la formula de sumatorias obtenemos que:

$$c * (n - 1 + 1)(\frac{(n-1)}{2}) = \frac{cn^2}{2} - \frac{cn}{2c}$$

Para usar la notación big-0 , solo nos importa tomar el n con mayor grado , puesto que los factores constantes , son insignificantes para el crecimiento de la función , por lo que en este caso obtenemos una complejidad en el peor de los casos de $\mathcal{O}(nlgn)$.

5.4. Shell sort

```
{
  int aux = *a;
  *a = *b;
  *b = aux;
}

void shellSort(int A[], int tam)
{
  for (int gap = tam / 2; gap > 0; gap /= 2)
  {
    for (int i = 0, m; i < tam - gap; i++)
    {
      m = i;
      while (m >= 0 && A[m + gap] < A[m])
      {
        swap(&A[m], &A[m + gap]);
        m = m - gap;
      }
    }
  }
}
```

La secuencia de espacios que fue originalmente sugerida por Donald Shell debía comenzar con N/2 y dividir por la mitad el número hasta alcanzar 1. Aunque esta secuencia proporciona mejoras de rendimiento significativas sobre los algoritmos cuadráticos como el ordenamiento por inserción, se puede cambiar ligeramente para disminuir más el tiempo necesario medio y el del peor caso. El libro de texto de Weiss demuestra que esta secuencia permite un ordenamiento $\mathcal{O}(n^2)$ del peor caso, si los datos están inicialmente en el vector es decir, la mitad alta de los números están situados, de forma ordenada, en las posiciones con índice par y la mitad baja de los números están situados de la misma manera en las posiciones con índice impar.

Quizás la propiedad más crucial del Shell sort es que los elementos permanecen k-ordenados incluso mientras el espacio disminuye.

Dependiendo de la elección de la secuencia de espacios, Shell sort tiene un tiempo de ejecución en el peor caso de $\mathcal{O}(n^2)$ (usando los incrementos de Shell que comienzan con n/2 y se dividen por 2 cada vez),

5.5. Merge Sort

5.5.1. T(n) = 2T(n/2) + n

$$a = 2 \qquad b = 2$$
$$log_b(a) \Rightarrow log_2(2) \Rightarrow 0$$

$f(n) = n^{log_b(a)}$ ***donde*** $T(n) = n^{log_b(a)} lgn$

$$n = n^{log_b(a)} \Longrightarrow 2^n = n^e$$

La complejidad de acuerdo al caso 2:

$$\boxed{T(n) = \theta(nlgn)}$$

Algoritmo	Peor caso	Caso promedio	Espacio
Bubble Sort	$\mathcal{O}(n^2)$	$\Theta(n^2)$	O(1)
Insertion Sort	$\mathcal{O}(n^2)$	$\Theta(n^2)$	O(1)
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	O(1)
Shell Sort	$\mathcal{O}(n^2)$	$\Theta(nlgn)$	$\Theta(1)$
Heap Sort	$\mathcal{O}(nlgn)$	$\Theta(nlgn)$	$\Theta(1)$
Merge Sort	$\Theta(nlgn)$	$\Theta(nlgn)$	$\Theta(n)$
Quick Sort	$\Theta(n^2)$	$\Theta(nlgn)$	$\Theta(n)$

Cuadro 1: Complejidad de los algoritmo de ordenamiento

6. Ejercicio 4

Evaluar y comparar sus algoritmos usando el archivo generado (variando el tama no del vector) y construir una(s) gráfica(s) comparativa(s).

En la máquina se corrierron las mismas pruebas en todos los algoritmos , el tamaño de los casos de prueba , refiriendonos al arreglo a ser ordenado ,fuerón de tamaños 100 , 200 , 5000 , 8000 , 10000 , 20000 , 50000, 80000 , 100000 , 200000 , 500000 , 800000 , 1000000 el valor de los elementos es mayor a 1 millon , por lo que los valores de los arreglos tienen menor posibilidad a repetirse , todos los arreglos estan completamente desordenados.

Maquina : 4 núcleos , 4GB ram.

Después de analizar los gráficos de las complejidades podemos notar que el algoritmo más rápido es el **quicksort** , seguido muy cerca de este el algoritmo **Merge Sort**

N	Bubble	Selection	Insertion	Shell	Heap	Merge	Quick
100	0.0269	0.0167	0.0075	0.0178	0.0249	0.0198	0.0134
200	0.0963	0.0572	0.0268	0.0419	0.0736	0.039	0.0273
500	0.5695	0.3502	0.1616	0.1252	0.1313	0.1127	0.0776
800	1.3462	0.8577	0.4288	0.238	0.2627	0.1789	0.1584
1000	2.1032	1.3542	0.6411	0.2358	0.3888	0.2271	0.1654
2000	8.7137	5.4095	2.4647	0.6263	0.6512	0.4071	0.3636
5000	63.2934	35.7762	16.9053	1.3742	1.5649	1.1253	1.0163
8000	175.141	79.64	46.2383	1.8383	2.4336	1.8585	1.6821
10000	279.589	120.431	62.7923	2.2465	2.6866	1.7176	1.8857
20000	1221.88	481.351	255.205	5.4457	5.9963	3.446	3.0657
50000	8750.97	3034.54	1608.41	15.2463	16.2769	9.6023	8.2557
80000	21739.8	7596.05	4107.08	26.6961	28.2127	15.6242	13.9178
100000	34383	11992.9	6347.62	32.8336	40.4101	19.5158	18.5544
200000	136670	50224.2	25427.4	74.9888	77.0498	41.2753	41.9629
500000	951700	315627	185185	220.802	206.24	110.921	98.7986
800000	2.44E+06	1.42E+06	458112	423.446	357.814	180.544	155.873
1000000	-	-	737923	537.51	465.037	219.119	197.174

Cuadro 2: Complejidad de los algoritmo de ordenamiento

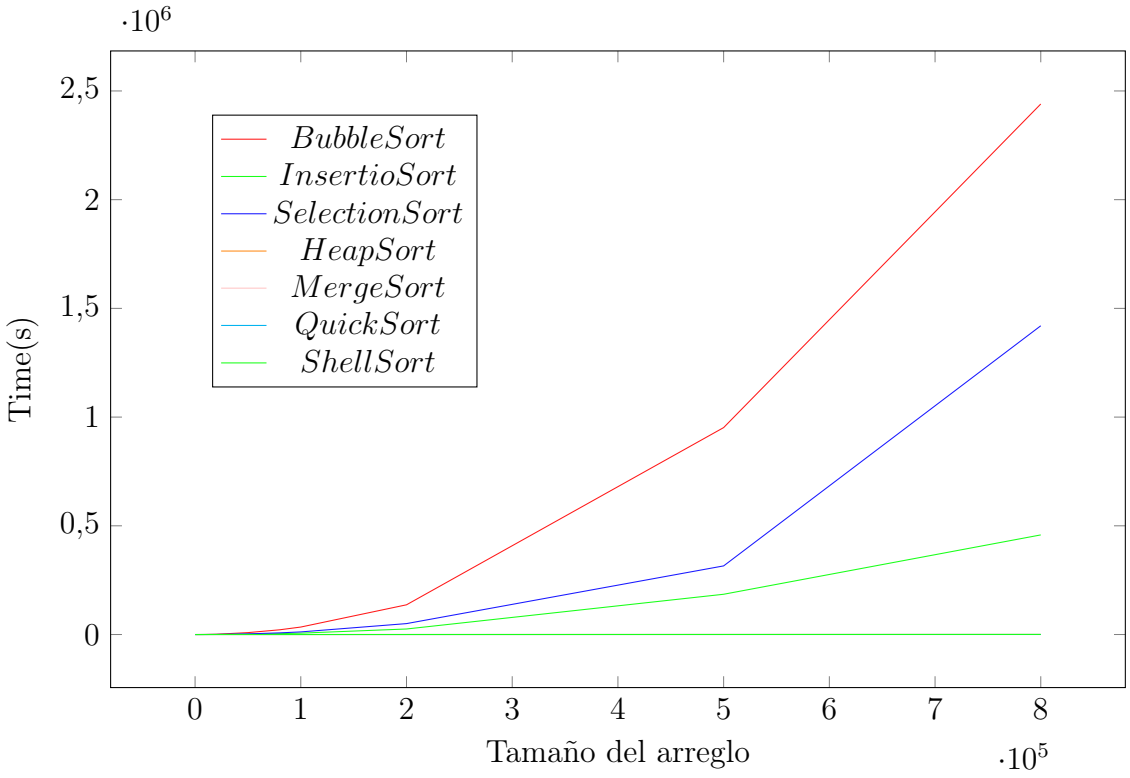

Figura 1: Comparación de complejidades

despues le sigue el heapSort y shellSort también podemos notar que el peor algoritmo , es el más lento es el algoritmo burbuja que sin duda tiene un crecimiento cuadrático , seguido del Insertion Sort y Selection Sort os cuales tienen igual una complejidad cua-

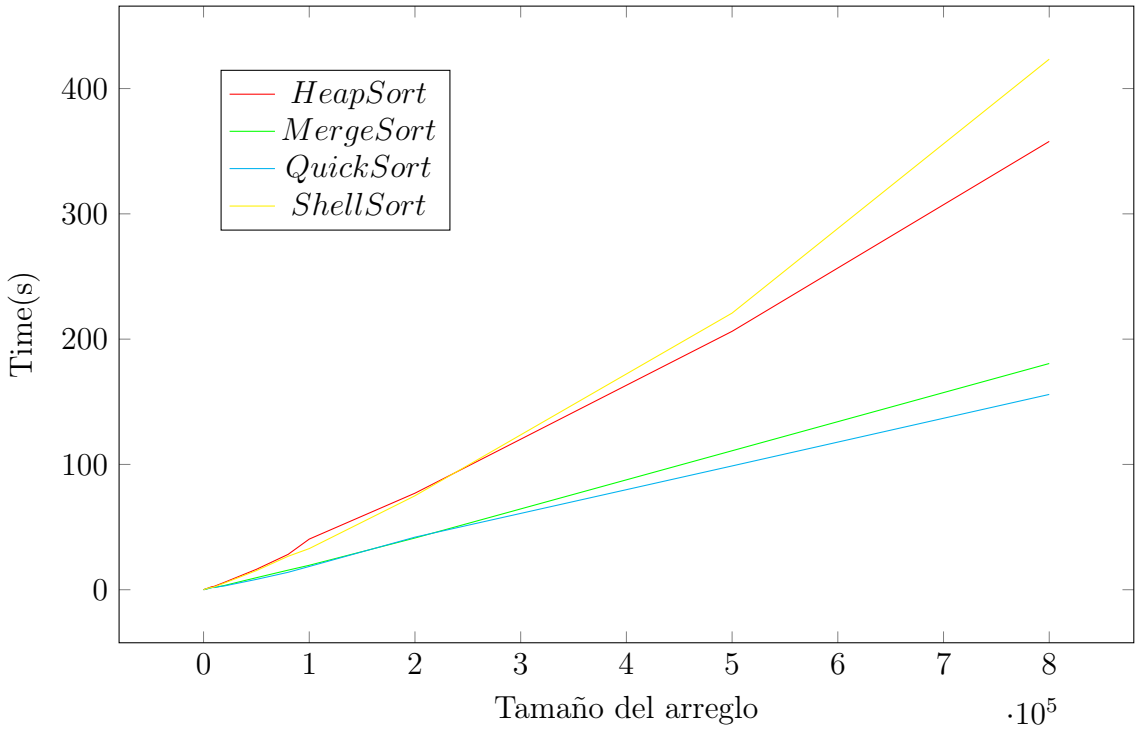

Figura 3: Comparación de complejidades

drática , pero sus implementaciones son un poco más eficientes dado que se hacen menos comparaciones, pero no dejan de ser algoritmos poco competentes.

7. Conclusiones

Podemos observar que existen una variedad de algoritmos , como herramientas y soluciones diversas a un mismo problema pero cada una con un campo de acción enfocadas a condiciones específicas , es nuestro objetivo encontrar la mejor solución que sea eficiente y rápida para poder solucionar un problema.