Algoritmos de Ordenamiento

Elizabeth Huang C23913

***Resumen*—En este trabajo se hace un análisis de varios algoritmos de ordenamiento con el objetivo de evaluar sus eficiencias, hacer comparaciones y comprender sus funciones. Esto se realiza a través de una comparación de tiempos de ejecución al ordenar un arreglo. El resultado de estas pruebas muestra de manera clara el rendimiento de cada uno de los algoritmos, lo cual puede ser útil al decidir cual aplicar en situaciones reales.**

Palabras clave—ordenamiento, selección, inserción, mezcla.

I. Introducción

Los algoritmos de ordenamiento son importantes en el ámbito de informática. La eficiencia de estos algoritmos puede garantizar un rendimiento ideal en los sistemas modernos. En este trabajo se implementa y se analiza el desempeño de los algoritmos de ordenamiento a través de un programa. Entre estos algoritmos se encuentra el de selección (selection sort), inserción (insertion sort), mezcla (merge sort), rápido (quick sort), montículos (heap sort) y residuos (radix sort).

II. Metodología

Para lograr lo propuesto se crea un programa utilizando el lenguaje c++ donde se implementan los algoritmos. En este caso se utilizará un arreglo como estructura de datos. Para poder determinar la eficiencia de cada método, se estará probando cada algoritmo con diferentes tamaños y se tomará el tiempo que dura en ejecutarse en milisegundos. Esto nos ayudara a la hora de decidir cual algoritmo es mejor adaptado a la situación.

Para obtener la comparación de eficiencia de los algoritmos, se generan arreglos de enteros aleatorios, con tamaños de 50,000, 100,000, 150,000 y 200,000 elementos. Estos representan diferentes escalas de datos y permiten una evaluación completa de cómo se comportan los algoritmos en diversas situaciones. Para crear estos arreglos y generar los rangos deseados, se utiliza la función rand() de c++ en la siguiente manera:

```
srand(time(NULL));
for (int i = 0; i < tam; i ++){
      A[i] = rand() % tam + 1;
}
```

Cuadro I: Tiempo de ejecución de los algoritmos

Algoritmo	Tam.	Tiempo (ms) *Corrida* 1	2	3	4	5	Prom.
Selección	50000	1920	1881	1881	1896	1889	1893,4
	100000	7415	7415	7400	7407	7400	7407,4
	150000	16467	16513	16956	16404	16716	16611,2
	200000	29504	29477	29276	29264	29228	29349,8
Inserción	50000	1226	1193	1190	1194	1190	1198,6
	100000	4690	4620	4658	4625	4690	4656,6
	150000	10346	10552	10345	10311	10348	10380,4
	200000	18358	18301	18218	18279	18298	18290,8
Mezcla	50000	7	7	6	6	6	6,4
	100000	15	14	14	14	14	14,2
	150000	22	21	21	21	22	21,4
	200000	30	29	30	29	29	29,4
Montículos	50000	9	10	9	10	9	9,4
	100000	22	21	22	21	22	21,6
	150000	33	33	34	33	33	33,2
	200000	46	54	48	47	47	48.4
Rápido	50000	5	6	5	5	6	5,4
	100000	11	11	11	12	12	11,4
	150000	18	18	18	18	19	18,2
	200000	26	25	26	27	26	26
Residuos	50000	0	0	0	0	0	0
	100000	1	1	1	2	1	1,2
	150000	2	2	2	2	2	2
	200000	3	3	3	3	3	3

Cada prueba se realiza midiendo el tiempo en milisegundos y los algoritmos ordenan los números de menor a mayor. Para obtener el tiempo en que se ejecuta el programa, se utiliza con crónometro de la librería chrono antes de llamar al algoritmo. Se ejecuta y luego para obtener el tiempo en milisegundos, se le resta al transcurrido. Para conocer el promedio de cada algoritmo según su tamaño, se ejecutan 5 veces con cada uno.

III. Resultados

Los tiempos de ejecución de las 5 corridas de los algoritmos se muestran en el cuadro I. Los tiempos que se obtuvieron durante las pruebas demuestran estabilidad ya que hay consistencia entre los tiempos de ejecución de las 5 corridas.

Comparando los promedios de cada algoritmo se puede decir que las diferencias son significativas. Con los algoritmos actuales, el Mezcla es más eficiente dado a que su tiempo de ejecución es el más bajo en todos los tamaños de entrada. Mientras que Selección y Inserción tienen tiempos de ejecución considerablemente más grandes, es decir, tienen un rendimiento inferior al de Mezcla.

Los tiempos promedio de los algoritmos de ordenamiento se muestran gráficamente en la figura 1. La forma de la curva en la gráfica creada es predecible dado a las diferencias en complejidades de los algoritmos utilizados. Inserción y Selección tienen como peor caso $O(n^2)$, por lo cual se puede anticipar un incremento significativo mientras mas grande sea n.

El algoritmo Mezcla tiene como complejidad temporal en el pero caso a $O(nlogn)$. En este algoritmo se destaca la rapidez al ordenar los datos sin importar el tamaño de la variable n.

Montículos (heapsort) tiene $O(nlogn)$ de peor caso utilizando el método dividir y conquistar. Con rendimiento similar al rápido (quicksort), montículos se destaca por que no depende del estado de los datos de entrada. El Quicksort igual divide y conquista, sin embargo, su peor caso puede llegar a ser $O(n^2)$.

Residuos es un algoritmo de ordenamiento que se basa en bits y tiene una complejidad de $O(nk)$. En este caso, k es la cantidad de dígitos/bits de los números en la entrada. Su eficiencia a la hora de ejecutarse depende de los datos de entrada.

La comparación de los promedios de cada algoritmo de ordenamiento utilizado se encuentra en la figura 2.

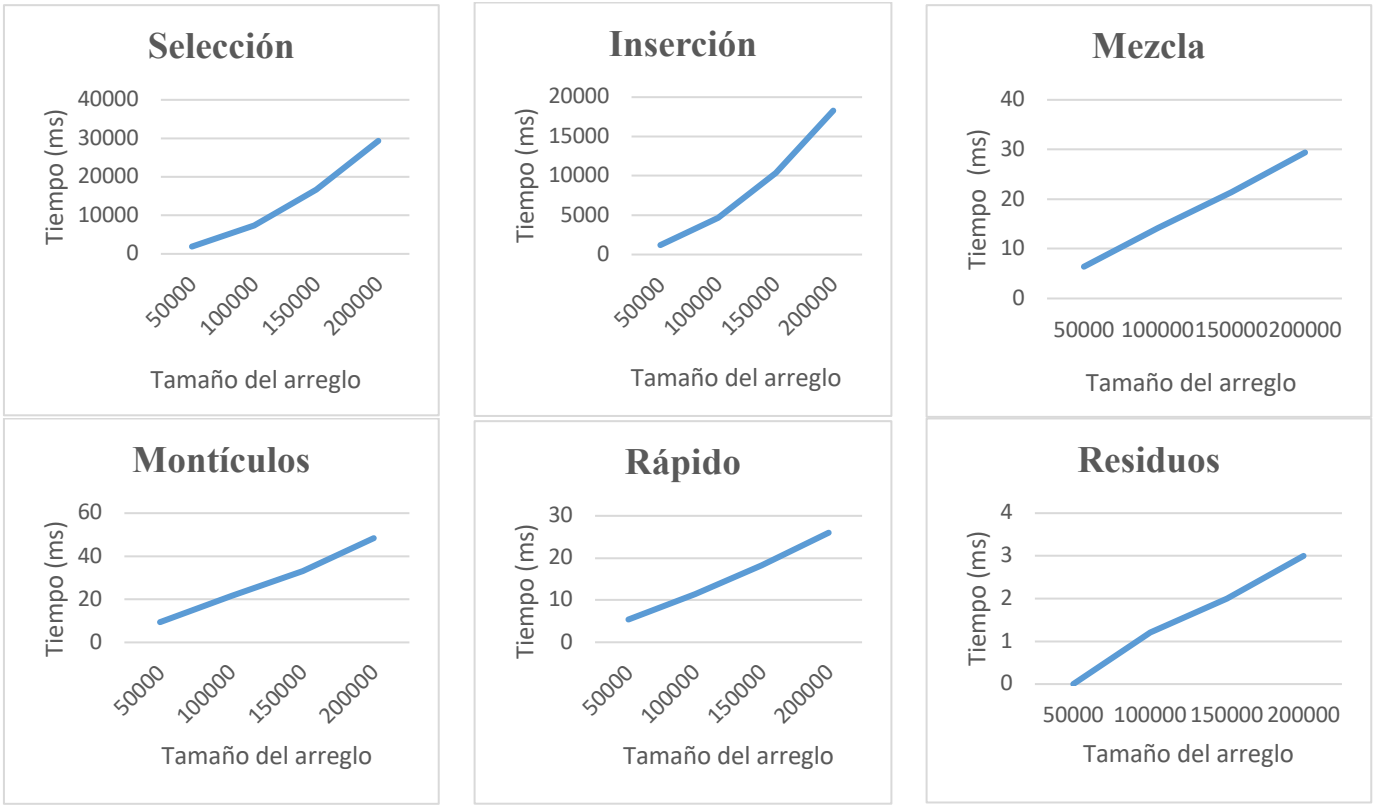

Figura 1 Tiempos promedio de ejecución de los algoritmos de ordenamiento por selección, inserción, mezcla, montículos, rápido y residuos.

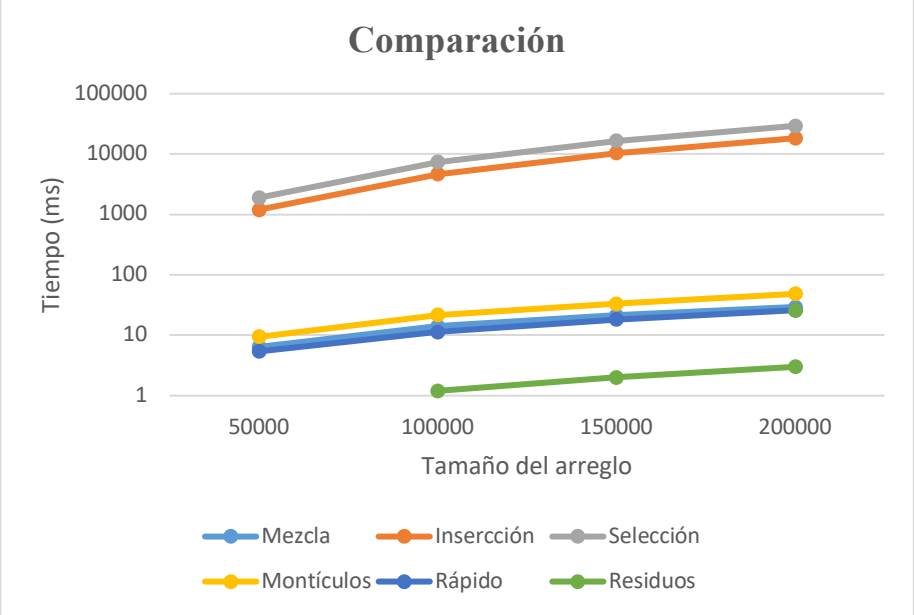

Figura 2 Gráfico comparative de los tiempos promedio de ejecución de algoritmos de ordenamiento.

IV. Conclusiones

A partir de los resultados obtenidos durante esta investigación, se puede concluir que los algoritmos de Inserción y Selección tienen limitaciones al manejar tamaños grandes de datos dado a sus complejidades cuadráticas. Por el otro lado, Mezcla es capaz de demostrar su eficiencia al tener tiempos de ejecución más bajos independiente del tamaño de los datos y con una complejidad logarítmica.

De esta investigación se puede concluir que los algoritmos de selección e inserción se ven limitados en el momento de manejar datos grandes dado a sus complejidades cuadráticas $O(n^2)$, lo cual resulta en tiempos de ejecución notablemente mas largos. Mientras que el algoritmo de mezcla, con una complejidad de $O(nlogn)$, es significativamente mas eficiente independiente del tamaño del arreglo.

Del otro lado, algoritmos rápido, montículos y residuos también tienen buena eficiencia general. El algoritmo de montículos tiene un rendimiento consistente y tiene una complejidad de $O(nlogn)$, esto lo hace útil para una gran variedad de situaciones. Rápido (Quicksort) tiene una complejidad promedia de $O(nlogn)$, sin embargo, hay que tener en mente que el peor caso puede llegar a ser una complejidad cuadrática $O(n^2)$. El algoritmo residuos es mas eficiente cuando hay una cantidad fija de dígitos o bits dado a su complejidad $O(nk)$.

El análisis que se ha llevado a cabo nos ayuda a decidir entre los algoritmos de ordenamiento, dependiendo del tamaño de los datos, cual es mejor para situaciones reales. Nos deja un conocimiento más profundo de las funciones de cada algoritmo y nos ayuda a reconocer el rendimiento y eficiencia entre ellos.

Referencias

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L. y Stein, C. Introduction to Algorithms, IV ed. MIT Press, 2022.

Elizabeth Huang
Estudiante de la Universidad de Costa Rica, interesada en aprender más sobre algoritmos de ordenamiento

APÉNDICE A

Código de los Algoritmos

El código se muestra en los algoritmos 1, 2, 3, 4, 5 y 6.

Algoritmo 1 Ordenamiento de un arreglo usando Selection-Sort.

```
void seleccion(int *A, int n){
  int menor;
  for (int i = 0; i < n-1; i++){
    menor = i;
    for (int j = i+1; j < n; j++){
      if (A[j] < A[menor]){
        menor = j;
      }
    }
    int aux = A[i];
    A[i] = A[menor];
    A[menor] = aux;
  }
}
```

Algoritmo 2 Ordenamiento de un arreglo usando Insertion-Sort.

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

Algoritmo 3 Ordenamiento de un arreglo usando Merge-Sort.

```
void mergesort(int *A, int n){
  int *tmp = new int[n];
  ord_intercalacion (A, tmp, 0, n - 1);
}

void ord_intercalacion(int * A, int * tmp, int izq, int der){
  if (izq < der){
    int centro = (izq + der) / 2;
    ord_intercalacion (A, tmp, izq, centro);
    ord_intercalacion (A, tmp, centro + 1, der);
    intercalar (A, tmp, izq, centro + 1, der);
  }
}

void intercalar (int * A, int * tmp, int izq, int centro, int der){
  int ap = izq, bp = centro, cp = izq;
  while ((ap < centro) && (bp <= der)){
    if (A[ap] <= A[bp]){
      tmp[cp] = A[ap];
      ap++;
    } else {
      tmp[cp] = A[bp];
      bp++;
    }
    cp++;
```

```
  }

  while (ap < centro){
    tmp[cp] = A[ap];
    cp++;
    ap++;
  }

  while (bp <= der){
    tmp[cp] = A[bp];
    cp++;
    bp++;
  }

  for (ap = izq; ap <= der; ap++){
    A[ap] = tmp[ap];
  }
}
```

Algoritmo 4 Ordenamiento de un arreglo usando Heap-Sort.

```
void max_heapify(int *A, int n, int i){
  int largest = i;
  int left = 2 * i + 1;
  int right = 2 * i + 2;
  if (left < n && A[left] > A[largest])
    largest = left;

  if (right < n && A[right] > A[largest])
    largest = right;

  if (largest != i) {
    std::swap(A[i], A[largest]);
    max_heapify(A, n, largest);
  }
}

void buildMaxHeap(int *A, int n) {
  for (int i = n / 2 - 1; i >= 0; i--) {
    max_heapify(A, n, i);
  }
}

void heapsort(int *A, int n){
  buildMaxHeap(A, n);
  for (int i = n - 1; i >= 0; i--) {
    std::swap(A[0], A[i]);
    max_heapify(A, i, 0);
  }
}
```

Algoritmo 5 Ordenamiento de un arreglo usando Quick-Sort.

```
void quicksort(int *A, int n){
  quicksortAux(A, 0, n - 1);
}

void quicksortAux(int *A, int p, int r) {
  if (p < r) {
    int q = partition(A, p, r);
    quicksortAux(A, p, q - 1);
    quicksortAux(A, q + 1, r);
  }
}

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++;
      std::swap(A[i], A[j]);
    }
  }
  std::swap(A[i + 1], A[r]);
  return i + 1;
}
```

Algoritmo 5 Ordenamiento de un arreglo usando Radix-Sort.

```
void radixsort(int *A, int n){
  int tempArray[n];
  int* temp = tempArray;

  const int numBytes = 4;

  for (int byteIndex = 0; byteIndex < numBytes; byteIndex++) {
    size_t byteCount[256] = {};
    size_t inicio = 0;
    if (byteIndex == 3) {
      inicio = 128;
    }

    for (int i = 0; i < n; i++) {
      int byteValue = (A[i] >> (8 * byteIndex)) & 0xFF;
      byteCount[byteValue]++;
    }

    for (int i = 1 + inicio; i < 256 + inicio; i++) {
      byteCount[i % 256] += byteCount[(i - 1) % 256];
    }

    for (int i = n - 1; i >= 0; i--) {
      int byteValue = (A[i] >> (8 * byteIndex)) & 0xFF;
      temp[--byteCount[byteValue]] = A[i];
    }

    std::swap(A, temp);
  }
}
```