Algoritmos de Ordenamiento

Elizabeth Huang C23913

***Resumen*—En este trabajo se hace un análisis de varios algoritmos de ordenamiento con el objetivo de contrastar las complejidades temporales teóricos con resultados prácticos. Para esto se realiza una serie de experimentos y se recolectan los tiempos de ejecución. El resultado fue que en los algoritmos con complejidad más alta se ven significativamente afectados por la cantidad de datos teórica y prácticamente. En conclusión, las complejidades teóricas se ven representadas en los resultados de los tiempos de ejecución obtenidos.**

Palabras clave—ordenamiento, selección, inserción, mezcla, rápido, montículos, residuos, tiempos de ejecución.

I. Introducción

Los algoritmos de ordenamiento se utilizan a gran escala en el ámbito de informática, dado a que logran mejoran la complejidad de un problema en muchos casos. Estos se implementan en algoritmos de búsqueda, bases de datos, métodos de divide y vencerás, estructuras de datos, entre otros. Sin embargo, la decisión de cual usar depende de la naturaleza del problema. Se tiene que considerar que tan grande son los datos que se ordenaran, cuanta memoria hay disponible y si la cantidad debe de crecer.

Por estas razones, en esta investigación se busca recolectar información sobre el desempeño real de los algoritmos a través de experimentos y contrastarlo a la complejidad temporal teórica. En otras palabras, se estará enfocando en los tiempos de ejecución y las complejidades. Para este experimento se utilizarán una mezcla de algoritmos de ordenación iterativos y recursivos: inserción, selección, mezcla, rápido, montículos, residuos.

A) Selección: En este algoritmo el arreglo se divide en dos partes, el ordenado y el no ordenado. Se itera a través del arreglo para encontrar el elemento mas pequeño y se mueve a la posición inicial del subarreglo actual. Después de cada iteración, el arreglo se vuelve más pequeño y solo se revisa el lado no ordenado. En el peor caso tiene complejidad de tiempo $O(n^2)$.

B) Inserción: El algoritmo itera a través de un arreglo desde 2 hasta n, el número máximo de datos en el arreglo. Lo que hace es agarrar el elemento siguiente como clave y lo posiciona en su lugar correspondiente. Para lograr eso, se compara con el elemento anterior y se verifica si es menor o mayor. Si es mayor, los elementos cambian de posición, hasta que se encuentre uno menor que el elemento clave. En el peor caso tiene complejidad de tiempo $O(n^2)$.

C) Mezcla: Es un algoritmo de dividir y vencerás. Lo que hace es dividir el arreglo en dos partes, ordenar cada mitad de manera recursiva y al final las vuelve a unir. En el peor caso tiene complejidad de tiempo $O(nlogn)$.

D) Rápido: También un algoritmo de dividir y vencerás. Se escoge un elemento en el arreglo como pivote y parte el arreglo entre dos, un lado con los elementos mayores al pivote y aquellos que son menores. Esto se hace de manera recursiva hasta que el arreglo inicial este ordenado. En el peor caso tiene complejidad de tiempo $O(nlogn)$.

E) Montículos: Se basa en la estructura de datos de montículos. Primero se construye un montículo máximo a base del arreglo inicial. Después extrae repetitivamente el elemento más grande del arreglo y después vuelve a construir el montículo, hasta que este ordenado. En el peor caso tiene complejidad de tiempo $O(nlogn)$.

F) Residuos: El algoritmo de ordenamiento de residuos no utiliza comparaciones, sino agrupa enteros por dígitos individuales. Comienza desde el digito más izquierdo hasta el más derecho, utilizando ordenación por cuentas para cada paso. En el peor caso tiene complejidad de tiempo $O(n+k)$. La k en este caso sería el elemento máximo del arreglo de entrada.

II. Metodología

Para lograr lo propuesto se crea un programa utilizando el lenguaje c++ donde se implementarán los algoritmos de ordenamiento en un archivo llamado Ordenador.h. Los métodos se encontrarán codificados en la clase ordenador en la sección publica, mientras que los auxiliares se ubicarán en el privado.

En este caso se utilizará un arreglo como estructura de datos. Se estará probando cada algoritmo con diferentes tamaños y se tomará el tiempo que dura en ejecutarse en milisegundos. Estos resultados se estarán comparando con la complejidad temporal teórica.

Más a fondo, en este experimento 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. Cada prueba se realiza midiendo el tiempo que dura cada algoritmo para ordenar el arreglo de diferentes tamaños en milisegundos. Para cada tamaño se ejecutara el algoritmo 5 veces, de esta manera se puede sacar el promedio que se utiliza para hacer un análisis más profundo.

Para crear estos arreglos se utiliza la función rand() definida en <stdlb.h> que permite generar números

los tiempos de ejecución en milisegundos, se utiliza una función de cronometro de la librería <chrono> antes y después de llamar al algoritmo de ordenamiento. Nos funciona chrono ya que guarda el tiempo que empieza y el tiempo que termina, después se restan para encontrar el tiempo de ejecución.

El experimento se realiza en una computadora con sistema operativo macOS que tiene una CPU 3,1 GHz 6-Core Intel Core i5, una GPU AMD Radeon Pro 5300 4 GB y RAM 8 GB 2667 MHz DDR4. Además, se utiliza la IDE Visual Studio Code para trabajar el código. Es importante considerar el hardware de donde se conduce esta investigación dado a que los resultados pueden variar si se realiza en otro dispositivo.

El código se muestra en los apéndices. Este código está basado en el seudocódigo del libro de Cormen y colaboradores [1].

Cuadro I: Tiempo de ejecución de los algoritmos

		Tiempo (ms) *Corrida*					
Algoritmo	Tam.	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	26.0
Residuos	50000	0	0	0	0	0	0.0
	100000	1	1	1	2	1	1.2
	150000	2	2	2	2	2	2.0
	200000	3	3	3	3	3	3.0

III. Resultados

Los tiempos de ejecución de las 5 corridas, por tamaño de arreglo, de los algoritmos se muestran en el cuadro I.

pseudoaleatorios entre 0 y RAND_MAX. Luego para capturar

A) Selección: Los tiempos que se obtuvieron durante las pruebas demuestran estabilidad ya que hay consistencia entre los tiempos de ejecución de las 5 corridas en cada tamaño. Sin embargo, con los tamaños más grandes se nota una desviación estándar mayor a las de los arreglos pequeños. Desde una desviación de menos de 20 hasta una mayor de 200. A la hora de ver los promedios de los tiempos de ejecución basados en los tamaños, la diferencia es exponencial. Se puede asumir que este algoritmo es más útil para manejar una cantidad de datos pequeño.

B) Inserción: Igual que al de selección, tiene tiempos parecidos entre las 5 corridas. Pero, la desviación estándar entre los tiempos ya sea en los arreglos pequeños o en los grandes, no sobrepasa a los 80. Aun así, los tiempos promedios tienen una diferencia significativa entre los tamaños.

C) Mezcla: Después de los primero dos algoritmos, la diferencia entre los tiempos es relevante. No solo es más rápido, también tiene una desviación pequeña entre las corridas.

D) Montículos: Aunque sea un poco más lento que el de mezcla, también es destacado por su velocidad al ordenar los arreglos.

E) Rápido: Como el nombre implica, es uno de los algoritmos de ordenamiento más rápidos. Con una desviación mínima y maneja bien los arreglos de gran escala.

F) Residuos: Aun sin el uso de comparación, este algoritmo realiza su función de la manera más rápida entre los algoritmos utilizados.

La comparación de los promedios de cada algoritmo de ordenamiento utilizado se encuentra en la figura 2. Comparando los promedios de cada algoritmo se puede decir que las diferencias son significativas ente los iterativos, los recursivos, los que se basan en dividir y vencer, los que no usan comparaciones, entre otros.

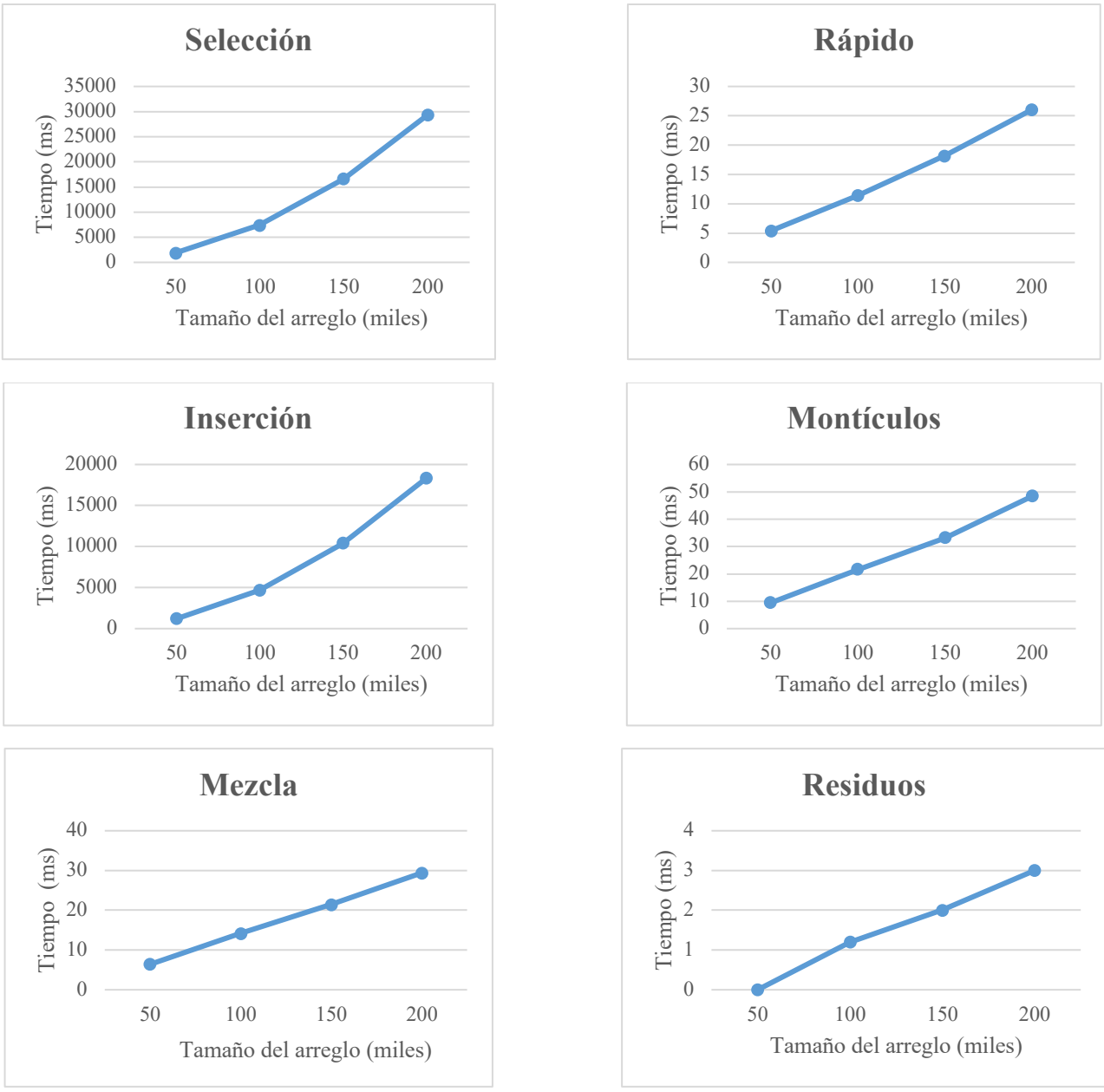

Figura 1 Tiempos promedio de ejecución de los algoritmos de ordenamiento por selección, inserción, mezcla, montículos, rápido y residuos.

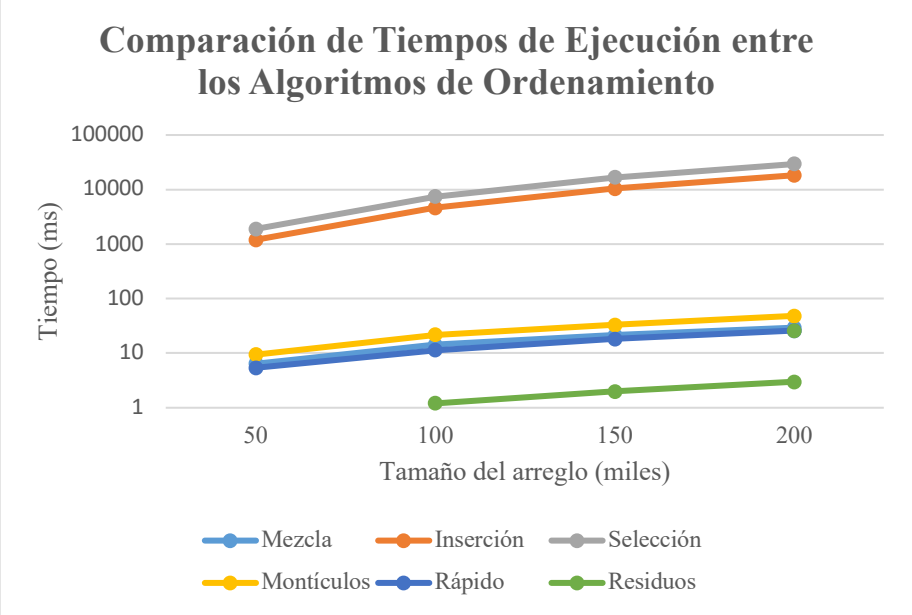

Figura 2 Gráfico comparativo de los tiempos promedio de ejecución de algoritmos de ordenamiento.

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 que tienen que recorrer el arreglo completo varias veces para poder ordenarlo. Esto se representa en sus complejidades cuadráticas $O(n^2)$.

Por el otro lado, mezcla, rápido y montículos son capaces de demostrar su eficiencia al tener tiempos de ejecución más bajos independiente del tamaño de los datos y con una complejidad logarítmica $O(nlogn)$. Estos algoritmos tienen una ventaja a los anteriores por su naturaleza de dividir el arreglo y utilizar la recursividad para completar su función.

El algoritmo residuos es el más eficiente entre los que se evaluaron, con los tiempos de ejecución más bajos y con una complejidad temporal de $O(nk)$. Esta depende del tamaño de los elementos, que en este caso se denomina k. Se debe considerar que este algoritmo no utiliza la comparación para cumplir con su función.

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.

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 y los resultados prácticos que lo reflejan.

Con esto se puede concluir que los resultados obtenidos de este experimento respaldan las complejidades temporales teóricas de cada uno de los algoritmos.

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.

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 6 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);
  }
}
```