Algoritmos de Ordenamiento

Ericka Araya-C20553

***Resumen*—En este trabajo, se evaluó el tiempo de duración en que se tardan los seis algoritmos de ordenamiento: por selección, por inserción, por mezcla, rápido, por montículos y por residuos mediante la ejecución de experimentos con arreglos de diferentes tamaños. Las gráficas resultantes mostraron las tendencias esperadas, reflejando las diferencias en la complejidad temporal teórica de los algoritmos. Estos hallazgos proporcionan información sobre cómo se comportan estos algoritmos en términos prácticos cuando se enfrentan a conjuntos de datos de diversos tamaños. Este análisis contribuye significativamente al campo de la ciencia de la computación al proporcionar orientación sobre la selección adecuada de algoritmos de ordenamiento en función de las necesidades específicas de las aplicaciones y las características de los conjuntos de datos.**

***Palabras clave*—algoritmo, ordenamiento, selección, inserción, mezcla, rápido, montículos, residuos, arreglo.**

I. Introducción

En este trabajo, se profundiza en el análisis de los experimentos llevados a cabo para observar los tiempos de ejecución de los algoritmos de ordenamiento por selección, por inserción, por mezcla, por montículos, rápido y por residuos. Cada uno de estos algoritmos posee características únicas en cuanto a su complejidad temporal y eficiencia en diferentes tipos de datos de entrada.

Dada una lista de números, el algoritmo de ordenamiento por selección recorre repetidamente la lista de elementos para encontrar el elemento mínimo y lo coloca al principio de la lista, intercambiándolo con el elemento en la posición actual. Este proceso se repite hasta que toda la lista esté ordenada. Por otro lado, el algoritmo de ordenamiento por inserción toma uno de los elementos de la lista y lo inserta en la posición correcta en la porción ya ordenada de la lista. Este proceso se repite hasta que todos los elementos están ordenados.

El algoritmo de ordenamiento por mezcla sigue el enfoque de "divide y conquista", dividiendo repetidamente la lista en mitades hasta que son lo suficientemente pequeñas para ordenarlas fácilmente, y luego fusionando las sublistas ordenadas.

El algoritmo de ordenamiento de montículos utiliza una estructura de datos de montículo para organizar los elementos de la lista. Los elementos se colocan en un montículo y luego se extraen uno por uno para obtener la lista ordenada.

El algoritmo de ordenamiento rápido elige un elemento pivote y divide la lista en dos sublistas alrededor del pivote. Luego, las sublistas se ordenan recursivamente. Por último, el algoritmo de ordenamiento de residuos es adecuado para clasificar números enteros dentro de un rango específico. Cuenta el número de ocurrencias de cada valor en la lista y luego reconstruye la lista ordenada.

En las pruebas se utilizaron arreglos de distintos tamaños (50.000, 100.000, 150.000, 200.000 elementos) y se registraron los tiempos de ejecución en cinco corridas para cada tamaño del arreglo. Esto con el fin de ver como estos algoritmos se comportan ante arreglos con números y tamaños diferentes.

El artículo se estructura en dos partes principales: la primera se centra en el análisis individual de cada algoritmo, evaluando la consistencia de los tiempos de ejecución y las variaciones en el promedio de duración para diferentes tamaños de arreglos. La segunda parte se encarga de comparar los tiempos promedio de los distintos algoritmos de ordenamiento, con el objetivo de comprender cómo varían los tiempos de ejecución en diferentes configuraciones de entrada. Más específicamente, esta comparación tiene como objetivo identificar fortalezas y debilidades de cada algoritmo en términos de eficiencia y escalabilidad.

Además, al comparar los algoritmos, también se pueden identificar casos de uso específicos para cada uno. Por ejemplo, un algoritmo puede destacarse en la clasificación de datos parcialmente ordenados, mientras que otro puede ser más adecuado para datos completamente desordenados. Esta comprensión más profunda de las fortalezas y limitaciones de cada algoritmo puede guiar la selección y el diseño de algoritmos en aplicaciones del mundo real, lo que puede tener un impacto significativo en la eficiencia operativa y el rendimiento del sistema.

II. Metodología

Para llevar a cabo la evaluación del rendimiento de los algoritmos de ordenamiento, se siguió un enfoque metodológico estructurado que implicaba varias etapas bien definidas.

1. Implementación de los algoritmos: Se implementaron los algoritmos de ordenamiento por selección, por inserción, por mezcla, por montículos, rápido y por residuos en un entorno de desarrollo en C++. Estos algoritmos se encapsularon en funciones miembro de una clase llamada Ordenador, con el objetivo de facilitar su reutilización y mantenimiento. Cada algoritmo se diseñó específicamente para ordenar arreglos de enteros en orden ascendente.

2. Implementación del código de los algoritmos: El código de los algoritmos de ordenamiento se basó en el pseudocódigo proporcionado en los libros "Data Structures and Algorithms" de Aho, Hopcroft y Ullman [1], así como en "Introduction to Algorithms" de Cormen, Leiserson, Rivest y Stein [2]. Se garantizó que la implementación de los algoritmos fuera fiel a los conceptos y técnicas descritas en estos libros.

3. Diseño del procedimiento experimental: Se diseñó un procedimiento experimental detallado para evaluar el tiempo de ejecución de los algoritmos de ordenamiento. Este procedimiento incluyó la selección de conjuntos de datos de diferentes tamaños (50.000, 100.000, 150.000 y 200.000 elementos) y la realización de múltiples ejecuciones de cada algoritmo sobre estos conjuntos de datos.

4. Registro y análisis de tiempos de ejecución: Se registraron los tiempos de cada ejecución de los algoritmos y se calculó el tiempo promedio de ejecución para cada tamaño de arreglo. Estos datos se organizaron en una tabla para su posterior análisis. Además, se utilizaron técnicas de visualización de datos para crear gráficos que representen los tiempos promedio, lo que permitió analizar con mayor claridad las diferencias entre los tiempos promedio y observar las tendencias generales de los algoritmos en función del tamaño del conjunto de datos.

5. Análisis de resultados: Se llevó a cabo un análisis detallado de los resultados obtenidos, centrándose en identificar patrones, tendencias y diferencias significativas en el tiempo de ejecución de los algoritmos. Se examinaron las curvas producidas por las gráficas para comprender cómo variaban los tiempos de ejecución en función del tamaño del conjunto de datos.

III. RESULTADOS

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

Cuadro I
TIEMPO DE EJECUCIÓN DE LOS ALGORITMOS.

		Tiempo (ms)					
		Corrida					
Algoritmo	Tam. (k)	1	2	3	4	5	Prom.
Selección	50000	2981	2979	2934	2928	2938	2952
	100000	11756	11922	11937	11879	11654	11829
	150000	26505	26166	26402	26416	26088	26315
	200000	46502	46485	46927	46473	46345	46546
Inserción	50000	1664	1613	1616	1624	1624	1628
	100000	6499	6453	6504	6479	6473	6481
	150000	14595	14613	14578	14560	14617	14592
	200000	25876	26771	25902	26122	26886	26311
Mezcla	50000	13	12	11	14	14	13
	100000	26	25	22	24	25	24
	150000	38	33	36	33	37	35
	200000	45	44	42	43	44	43
Montículos	50000	36	35	35	37	32	35
	100000	79	80	78	79	62	75
	150000	92	90	87	89	100	91
	200000	120	115	115	115	117	116
Rápido	50000	18	18	16	19	15	17
	100000	36	32	31	29	27	31
	150000	41	40	40	40	41	40
	200000	66	63	65	66	57	63
Residuos	50000	7	8	7	6	7	7
	100000	22	24	21	15	20	20
	150000	30	21	17	19	25	22
	200000	21	23	28	21	21	23

En el caso del algoritmo de ordenamiento por selección, se observa que los tiempos de ejecución varían considerablemente, entre las cinco corridas para cada tamaño de arreglo. Por ejemplo, para un tamaño de arreglo de 50,000 elementos, los tiempos oscilan entre 2,928 ms y 2,981 ms. Esta variabilidad se observa también en los otros tamaños de arreglos. Aunque los tiempos promedio tienden a ser consistentes entre las corridas para un mismo tamaño de arreglo, la variabilidad es notable.

Para el algoritmo de ordenamiento por inserción, también se observa variabilidad en los tiempos de ejecución entre las corridas, aunque en menor medida que en el algoritmo de ordenamiento de selección. Los tiempos oscilan en un rango más estrecho y los promedios son más consistentes entre las corridas para un mismo tamaño de arreglo.

Por otro lado, para el algoritmo de ordenamiento por mezcla, los tiempos de ejecución son más consistentes entre las corridas, con una variabilidad mínima. Los tiempos promedio son bastante estables para cada tamaño de arreglo.

Los tiempos de ejecución para el algoritmo de ordenamiento por montículos muestran una variabilidad moderada entre las diferentes corridas para cada tamaño de arreglo. Sin embargo, los rangos de tiempo de ejecución son relativamente estrechos, lo que indica una consistencia general en el rendimiento del algoritmo. Los tiempos promedio entre las corridas son bastante estables para cada tamaño de arreglo. A medida que aumenta el tamaño del arreglo, se observa un aumento gradual en el tiempo de ejecución.

El algoritmo de ordenamiento rápido exhibe una variabilidad similar a la de ordenamiento por montículos, con rangos de tiempo de ejecución moderadamente estrechos entre las corridas para cada tamaño de arreglo. Los tiempos promedio también son consistentes dentro de cada tamaño de arreglo. A medida que el tamaño del arreglo aumenta, se observa un incremento gradual en el tiempo de ejecución, lo cual es coherente con la complejidad esperada del algoritmo.

Por último, para el algoritmo de ordenamiento por residuos, se observa una variabilidad similar en los tiempos de ejecución entre las diferentes corridas para cada tamaño de arreglo. Los rangos de variación son moderados, y los tiempos promedio son consistentes entre las corridas para un mismo tamaño de arreglo. Al aumentar el tamaño del arreglo, se registra un incremento gradual en el tiempo de ejecución, en línea con las expectativas del algoritmo.

Los tiempos promedio se muestran gráficamente en las figuras 1, 2, 3, 4, 5, 6.

Al observar la gráfica que muestra los tiempos promedio de ejecución de varios algoritmos de ordenamiento en una escala logarítmica, se pueden extraer diversas conclusiones sobre la eficiencia relativa de estos algoritmos 7.

En primer lugar, se nota que los algoritmos de ordenamiento basados en comparaciones, como ordenamiento por selección, ordenamiento por inserción, ordenamiento por mezcla y ordenamiento rápido, exhiben un aumento exponencial en el tiempo de ejecución a medida que aumenta el tamaño del arreglo. Esta tendencia sugiere que, para conjuntos de datos más grandes, estos algoritmos pueden volverse más lentos.

Dentro de los algoritmos de ordenamiento basados en comparaciones, se pueden identificar diferencias en la eficiencia relativa. Por ejemplo, ordenamiento por inserción parece ser el menos eficiente, seguido de cerca por el ordenamiento por selección, los cuales tienen igual una complejidad de $O(n^2)$. En contraste, el ordenamiento rápido y el ordenamiento por mezcla, con una complejidad $O(n\log n)$, parecen ser los más eficientes, con tiempos de ejecución significativamente menores en todos los tamaños de arreglo.

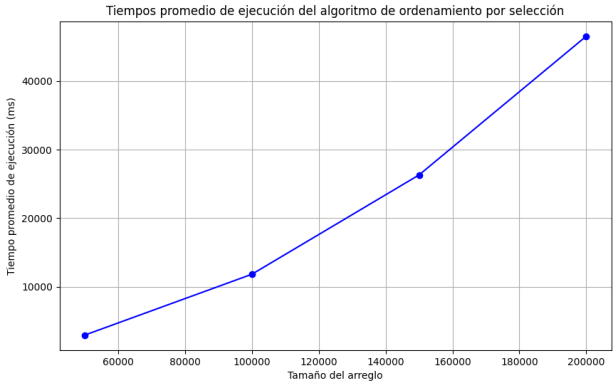


Figura 1. Tiempos promedio de ejecución del algoritmo de ordenamiento por selección.

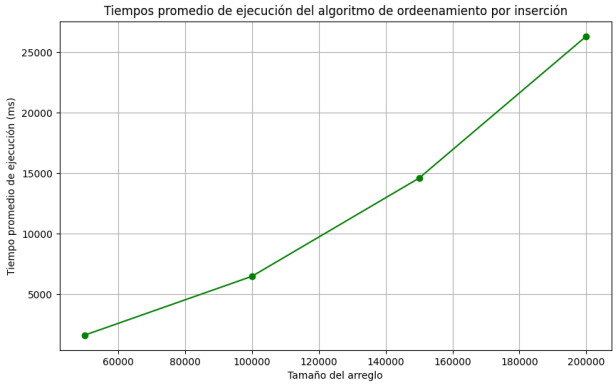


Figura 2. Tiempos promedio de ejecución del algoritmo de ordenamiento por inserción.

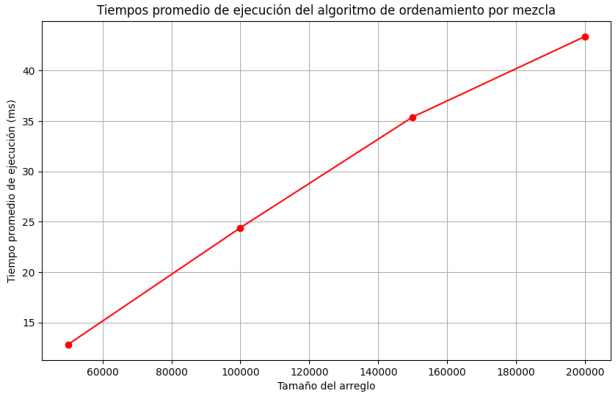


Figura 3. Tiempos promedio de ejecución del algoritmo de ordenamiento por mezcla.

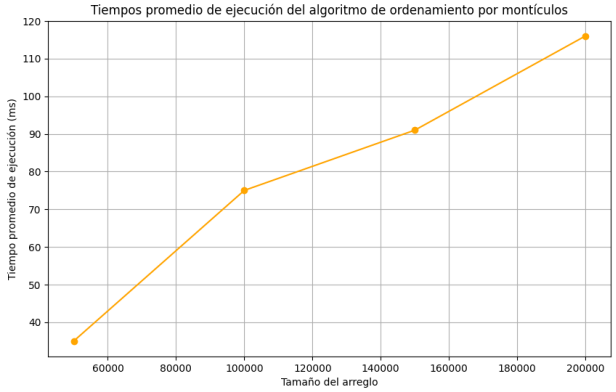


Figura 4. Tiempos promedio de ejecución del algoritmo de ordenamiento por montículos.

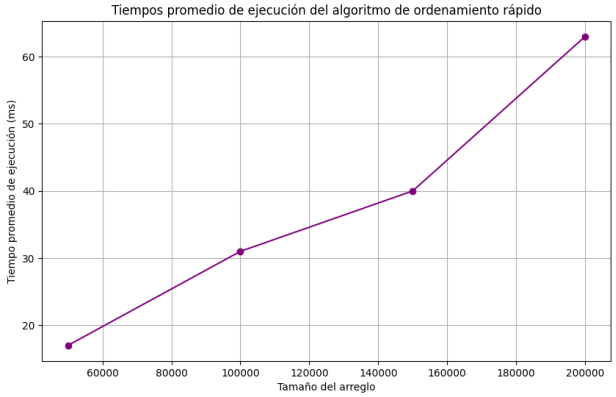


Figura 5. Tiempos promedio de ejecución del algoritmo de ordenamiento rápido.

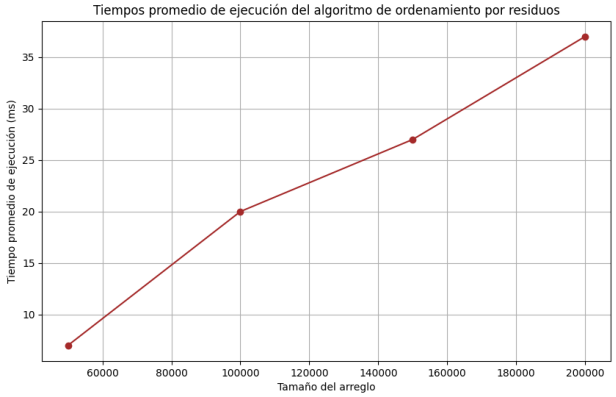


Figura 6. Tiempos promedio de ejecución del algoritmo de ordenamiento por residuos.

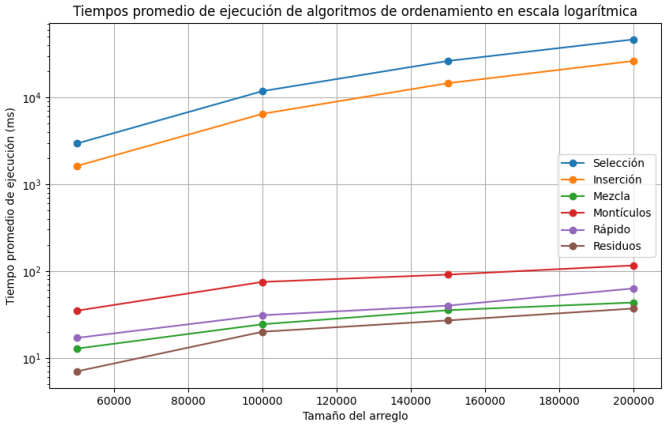


Figura 7. Gráfico comparativo de los tiempos promedio de ejecución de algoritmos de ordenamiento.

Por otro lado, los algoritmos de ordenamiento por montículos(con complejidad O(nlogn)) y por residuos(con complejidad O(nk)) muestran un comportamiento más estable en comparación con los que son basados en comparaciones, aunque todavía muestran un ligero aumento en el tiempo de ejecución con tamaños de arreglo más grandes. Para conjuntos de datos más grandes, estos algoritmos podrían ser preferibles en términos de eficiencia.

Sin embargo, la elección del algoritmo de ordenamiento más adecuada dependerá de varios factores, como el tamaño y las características específicas del conjunto de datos, así como las restricciones de rendimiento de la aplicación. Además del tiempo de ejecución, otros factores como la facilidad de implementación, la estabilidad del algoritmo y el espacio de memoria requerido también deben considerarse al seleccionar un algoritmo de ordenamiento para una tarea particular.

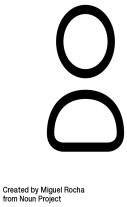
IV. Conclusiones

En conclusión, al examinar los resultados de las pruebas de los diferentes algoritmos de ordenamiento, se destaca la importancia de seleccionar cuidadosamente el algoritmo adecuado según las características del conjunto de datos y los requisitos de rendimiento. Los algoritmos de ordenamiento rápido y por mezcla sobresalen por su eficiencia y estabilidad en los tiempos de ejecución, lo que los convierte en opciones favorables para conjuntos de datos grandes. Por otro lado, aunque los algoritmos de ordenamiento por selección y de por inserción son simples de implementar, muestran una mayor variabilidad en sus tiempos de ejecución, lo que sugiere que pueden ser menos adecuados para aplicaciones sensibles al tiempo. Los algoritmos de ordenamiento por montículos y por residuos ofrecen una alternativa con tiempos de ejecución estables, aunque su eficiencia puede ser menor en comparación con los algoritmos de complejidad O(nlogn), a pesar de que el ordenamiento por montículos cuente con esta misma complejidad. En última instancia, la elección del algoritmo de ordenamiento debe considerar no solo la complejidad algorítmica, sino también la variabilidad en los tiempos de ejecución y las necesidades específicas del problema a resolver.

Referencias

[1] A. Alfred V, H. John E, and U. Jeffrey D, *Data Structures and Algorithms*, 1st ed. Micheal A, Harrison, 1983.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. The MIT Press, 2009.



Ericka Me considero alguien a quien le gusta la carrera que está estudiando, tengo otros intereses como el fútbol, hacer hiking, escuchar música. Me considero alguien feliz y con mucha perseverancia.

APÉNDICE A
CÓDIGO DE LOS ALGORITMOS

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

Algoritmo 1 Algoritmo de ordenamiento por selección que recorre repetidamente el arreglo para encontrar el elemento más pequeño y lo coloca en la posición correcta. A medida que avanza, va seleccionando el siguiente elemento más pequeño y lo intercambia con el elemento actual en la posición correcta:

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

Algoritmo 2 Algoritmo de ordenamiento por inserción que recorre el arreglo desde el segundo elemento hasta el final, insertando cada elemento en su posición correcta entre los elementos ya ordenados. Para hacer esto, compara cada elemento con los elementos previos hasta encontrar su posición correcta. :

```
void insercion(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 = j - 1;
    }
    A[j + 1] = key;
  }
}
```

Algoritmo 3 Algoritmo de ordenamiento por mezcla que utiliza la estrategia de dividir y conquistar. Divide el arreglo en dos mitades, ordena cada mitad de forma recursiva y luego combina las dos mitades ordenadas en un solo arreglo ordenado. Para combinar las mitades, compara los elementos uno por uno y los coloca en orden en un nuevo arreglo auxiliar:

```
void mergesort(int *A, int n){
    mergeSortHelper(A, 0, n - 1);
}
void mergeSortHelper(int *A,
    int left, int right){
  if (left < right) {
    int mid = left + (right - left) / 2;

    mergeSortHelper(A, left, mid);
    mergeSortHelper(A, mid + 1, right);

    merge(A, left, mid, right);
  }
}

void merge(int *A, int left,
    int mid, int right){
  int n1 = mid - left + 1;
  int n2 = right - mid;

  int L[n1], R[n2];

  for (int i = 0; i < n1; i++)
    L[i] = A[left + i];
  for (int j = 0; j < n2; j++)
    R[j] = A[mid + 1 + j];

  int i = 0;
  int j = 0;
  int k = left;
  while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
      A[k] = L[i];
      i++;
    } else {
      A[k] = R[j];
      j++;
    }
    k++;
  }

  while (i < n1) {
    A[k] = L[i];
    i++;
    k++;
  }

  while (j < n2) {
    A[k] = R[j];
    j++;
    k++;
  }
}
```

Algoritmo 4 Algoritmo de ordenamiento por montículos que utiliza una estructura de datos de montículo para organizar los elementos del arreglo. Se construye un montículo máximo a partir del arreglo y luego se extraen los elementos uno por uno para obtener el arreglo ordenado:

```
void heapsort(int *arr, int n){
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}
void heapify(int *arr, int n, int i){
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left<n && arr[left]>arr[largest])
        largest = left;
    if (right<n && arr[right]>arr[largest])
        largest = right;
    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}
```

Algoritmo 5 Algoritmo de ordenamiento rápido que elige un elemento pivote y divide el arreglo en dos subarreglos alrededor del pivote. Luego, los subarreglos se ordenan recursivamente:

```
void quicksort(int *arr, int n) {
    quicksortHelper(arr, 0, n - 1);
}
void quicksortHelper(int *arr,int low,int high){
    if (low < high) {
        int pi = partition(arr, low, high);
        quicksortHelper(arr, low, pi - 1);
        quicksortHelper(arr, pi + 1, high);
    }
}
int partition(int *arr,int low,int high){
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return (i + 1);
}
```

Algoritmo 6 Algoritmo de ordenamiento por residuos que es adecuado para clasificar números enteros dentro de un rango específico. Cuenta el número de ocurrencias de cada valor en el arreglo y luego reconstruye el arreglo ordenado:

```
void radixsort(int *A, int n) {
    int max_num = A[0];
    for (int i = 1; i < n; i++) {
        if (A[i] > max_num) {
            max_num = A[i];
        }
    }
   long long base = pow(2,log2(n));
    for (int i = 1; max_num / i > 0; i *= base){
        counting_sort(A, i, base, n);
    }
}
void counting_sort(int* A, int digit,
    int base, int n) {

    int* B = new int[n]();
    int* C = new int[base]();
    for (int i = 0; i < base; i++){
      C[i] = 0;
    }
    for (int i = 0; i < n; i++) {
        C[(A[i] / digit) % base]++;
    }
    for (int j = 1; j < base; j++) {
        C[j] += C[j - 1];
    }
    for (int i = n - 1; i >= 0; i--) {
        B[C[(A[i] / digit) % base] - 1] = A[i];
        C[(A[i] / digit) % base]--;
    }
    for (int i = 0; i < n; i++) {
        A[i] = B[i];
    }
    delete[] B;
    delete[] C;
}
```