**Estructura de Datos**

Tema n.° 7

Algoritmos de Ordenación Interna

**Índice**

**Pág.**

* + 1. Intercambio 3
    2. Burbuja 5
    3. Quicksort 8
    4. ShellSort 9
    5. Ordenación por Distribución 11
    6. Ordenación por Radix 17

Recursos complementarios 21

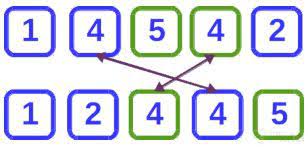
Referencias 22

Autoevaluación 23

## Intercambio

El algoritmo de intercambio (Exchange sort en inglés) funciona intercambiando elementos adyacentes en la lista si están en orden incorrecto (Cormen et al., 2022). Este proceso se repite hasta que no requiere realizar más intercambios dando a entender que la lista está ordenada

**Figura 1**



*funcionamiento del algoritmo de intercambio*

**codificación**

#include <iostream>

**void** **intercambio**(**int** arr[], **int** n) {

**for** (**int** i = **0**; i < n - **1**; i++) {

**for** (**int** j = **0**; j < n - i - **1**; j++) {

// Compara elementos adyacentes

**if** (arr[j] > arr[j + **1**]) {

// Intercambia si están en el orden incorrecto

std::swap(arr[j], arr[j + **1**]);

}

}

}

}

**int** **main**() {

**int** arreglo[] = {**64**, **34**, **25**, **12**, **22**, **11**, **90**};

**int** n = **sizeof**(arreglo) / **sizeof**(arreglo[**0**]);

std::cout << "Arreglo original: ";

**for** (**int** i = **0**; i < n; i++) {

std::cout << arreglo[i] << " ";

}

// Llamada a la función de intercambio

intercambio(arreglo, n);

std::cout << "**\n**Arreglo ordenado: ";

**for** (**int** i = **0**; i < n; i++) {

std::cout << arreglo[i] << " ";

}

**return** **0**;

}

**Ejecución**

**Texto

Descripción generada automáticamente**

**Análisis**

Función “***Intercambio”***

* La función **intercambio** toma un arreglo **arr** y su longitud **n** como parámetros.
* Utiliza dos bucles anidados para comparar y, si es necesario, intercambiar elementos adyacentes.
* El bucle externo **(for (int i = 0; i < n - 1; i++))** controla el número total de pasadas a través del arreglo.
* El bucle interno **(for (int j = 0; j < n - i - 1; j++))** compara y, si es necesario, intercambia elementos.
* La condición **if (arr[j] > arr[j + 1])** verifica si el elemento actual es mayor que el siguiente.
* Si la condición es verdadera**, std::swap(arr[j], arr[j + 1])** intercambia los elementos.

Función principal “**Main**”

* Se declara un arreglo de números desordenados llamado arreglo.
* La variable n se calcula como la longitud del arreglo.
* Se imprime el arreglo original utilizando un bucle for.
* Se llama a la función intercambio para ordenar el arreglo.
* Se imprime el arreglo ordenado utilizando otro bucle for.

**Conclusiones**

En resumen, el algoritmo de ordenamiento por intercambio, es simple pero menos eficiente en comparación con otros algoritmos más avanzados. Funciona intercambiando repetidamente elementos adyacentes hasta que el arreglo esté ordenado. Aunque es fácil de entender e implementar, su rendimiento no es óptimo, especialmente en conjuntos de datos grandes.

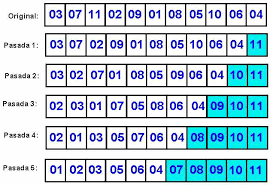
## Burbuja

### Definición

El algoritmo de ordenamiento Burbuja (bubble sort en inglés) funciona intercambiando repetidamente elementos adyacentes de la lista si están en el orden incorrecto. Es decir, si el elemento de la izquierda es mayor a la derecha, se intercambian. Este proceso se repite hasta que se completa el barrido si realizar ningún intercambio, dando a entender que la lista está ordenada (Cormen et al., 2022).

Específicamente, bubble sort parte del inicio de la lista y compara cada par de elementos adyacentes. Si el primer elemento es mayor que el segundo, los intercambia. Luego pasa al segundo y tercer elemento, intercambiándolos si están desordenados, y así sucesivamente. Cuando llega al final, vuelve a empezar desde el principio. Esto se repite hasta que en una iteración no se realiza ningún intercambio, indicando que la lista está ordenada (Cormen et al., 2022).

**Figura 2**



*Funcionamiento del algoritmo Burbuja de forma gráfica*

### Codificación (C++)

#include <iostream>

**void** **intercambiar**(**int**& a, **int**& b) {

**int** temp = a;

a = b;

b = temp;

}

**void** **burbuja**(**int** arr[], **int** n) {

**for** (**int** i = **0**; i < n - **1**; i++) {

**for** (**int** j = **0**; j < n - i - **1**; j++) {

**if** (arr[j] > arr[j + **1**]) {

intercambiar(arr[j], arr[j + **1**]);

}

}

}

}

**int** **main**() {

**int** arreglo[] = {**64**, **34**, **25**, **12**, **22**, **11**, **90**};

**int** n = **sizeof**(arreglo) / **sizeof**(arreglo[**0**]);

std::cout << "Arreglo original: ";

**for** (**int** i = **0**; i < n; i++) {

std::cout << arreglo[i] << " ";

}

// Llamada a la función de burbuja para ordenar el arreglo

burbuja(arreglo, n);

std::cout << "**\n**Arreglo ordenado: ";

**for** (**int** i = **0**; i < n; i++) {

std::cout << arreglo[i] << " ";

}

**return** **0**;

}

**Ejecución**

**Texto

Descripción generada automáticamente**

**Analisis**

**Función *“intercambiar”*:**

Esta función toma dos referencias a enteros (int& a y int& b) y realiza un intercambio de valores entre ellas utilizando una variable temporal. Es una función auxiliar utilizada para intercambiar elementos en el arreglo.

**Función “*burbuja”*:**

* La función burbuja toma un arreglo arr y su longitud n como parámetros.
* Utiliza dos bucles anidados para recorrer el arreglo y comparar elementos adyacentes.
* Si encuentra que un elemento es mayor que el siguiente, llama a la función intercambiar para intercambiar esos dos elementos.
* Este proceso se repite hasta que todo el arreglo esté ordenado. En cada iteración del bucle externo, el elemento más grande "burbujea" hasta su posición correcta al final del arreglo.

**Función *“main”*:**

* Se declara un arreglo de números desordenados llamado arreglo.
* Se imprime el arreglo original.
* Se llama a la función burbuja para ordenar el arreglo.
* Se imprime el arreglo ordenado.

**Conclusiones**

El método de burbuja es fácil de entender e implementar, su rendimiento no es ideal para conjuntos de datos grandes. Otros algoritmos de ordenamiento son generalmente preferidos en la práctica para mejorar la eficiencia.

## Quicksort

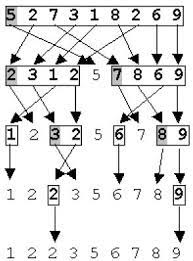
**Definición**

Quicksort es considerado uno de los algoritmos de ordenamiento más eficientes ampliamente utilizados. Pertenece a la categoría de algoritmos “divide y vencerás” (divide and conquer en inglés) (Cormen et al., 2022).

Funciona seleccionando un valor pivote en la lista y reordenando elementos para que aquellos menores al pivote queden a la izquierda, y los mayores a la derecha. Luego se aplica el algoritmo recursivamente a las sublistas izquierda y derechas obtenidas. El proceso sigue hasta que las sublistas tienen longitud 1 o 0, momento en el cual ya están ordenadas (Goodrich et al., 2022).

La selección apropiada del pivote es clave para la eficiencia del algoritmo. Existen distintas estrategias, como elegir el pivote al azar o utilizar la mediana de 3 elementos (el primero, central y último). El mejor caso ocurre cuando el pivote divide la lista en mitades iguales en cada paso (Goodrich et al., 2022).

**Figura 3**



*Funcionamiento del algoritmo Quicksort de forma gráfica*

**Codificación**

#include <iostream>

#include <algorithm>

**using** **namespace** std;

// Partición usando el esquema de partición de Lomuto

**int** **partition**(**int** a[], **int** start, **int** end)

{

// Elija el elemento más a la derecha como un pivote de la array

**int** pivot = a[end];

// los elementos menores que el pivote serán empujados a la izquierda de `pIndex`

// elementos más que el pivote serán empujados a la derecha de `pIndex`

// elementos iguales pueden ir en cualquier dirección

**int** pIndex = start;

// cada vez que encontramos un elemento menor o igual que el pivote, `pIndex`

// se incrementa, y ese elemento se colocaría antes del pivote.

**for** (**int** i = start; i < end; i++)

{

**if** (a[i] <= pivot)

{

swap(a[i], a[pIndex]);

pIndex++;

}

}

// intercambiar `pIndex` con pivote

swap (a[pIndex], a[end]);

// devuelve `pIndex` (índice del elemento pivote)

**return** pIndex;

}

// Rutina de clasificación rápida

**void** **quicksort**(**int** a[], **int** start, **int** end)

{

// condición base

**if** (start >= end) {

**return**;

}

// reorganizar los elementos a través del pivote

**int** pivot = partition(a, start, end);

// recurre en un subarray que contiene elementos que son menores que el pivote

quicksort(a, start, pivot - **1**);

// se repite en el subarray que contiene elementos que son más que el pivote

quicksort(a, pivot + **1**, end);

}

// Implementación en C++ del algoritmo Quicksort

**int** **main**()

{

**int** a[] = { **9**, -**3**, **5**, **2**, **6**, **8**, -**6**, **1**, **3** };

**int** n = **sizeof**(OPCION 1:/**sizeof**(a[**0**]);

quicksort(a, **0**, n - **1**);

// imprime la array ordenada

**for** (**int** i = **0**; i < n; i++) {

cout << a[i] << " ";

}

**return** **0**;

}

**Ejecución**

****

**Analisis**

**Función *“partition”*:**

* Esta función toma un arreglo a, y dos índices start y end como parámetros.
* Elige el último elemento (a[end]) como pivote.
* Itera sobre el arreglo y coloca los elementos menores o iguales al pivote a la izquierda del índice pIndex.
* Intercambia los elementos menores o iguales al pivote con a[pIndex].
* Finalmente, intercambia a[pIndex] con el pivote (a[end]) y devuelve pIndex.

**Función *“quicksort”*:**

* Esta función toma un arreglo a, y dos índices start y end como parámetros.
* Si start es mayor o igual a end, la función retorna, estableciendo así la condición base.
* Llama a la función partition para obtener el índice del pivote.
* Recursivamente aplica quicksort al subarreglo que contiene elementos menores que el pivote (start a pivot - 1).
* Recursivamente aplica quicksort al subarreglo que contiene elementos mayores que el pivote (pivot + 1 a end).

**Función *“main”:***

* Declara un arreglo de números desordenados a.
* Calcula el tamaño del arreglo n.
* Llama a la función quicksort para ordenar el arreglo.
* Imprime el arreglo ordenado.

**Conclusiones**

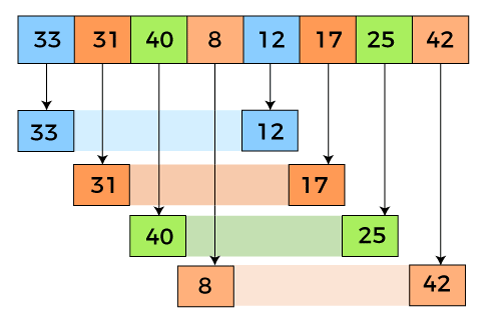
El algoritmo de ordenamiento Quicksort destaca por su eficiencia promedio, utilizando la estrategia de divide y conquista. La selección adecuada del pivote es crucial, y aunque puede tener un peor caso de O(n2), en la práctica suele tener un rendimiento excepcional con una complejidad de tiempo de O(n log n) en situaciones promedio. Quicksort es implementado in situ y es ampliamente utilizado en la práctica debido a su eficiencia y aplicaciones prácticas en bibliotecas y sistemas operativos.

## ShellSort

La ordenación Shell debe el nombre a su inventor, D. L. Shell. Se suele denominar también ordenación por inserción con incrementos decrecientes. Se considera que es una mejora del método de inserción directa. En el algoritmo de inserción, cada elemento se compara con los elementos contiguos de su izquierda, uno tras otro. Si el elemento a insertar es el más pequeño hay que realizar muchas comparaciones antes de colocarlo en su lugar definitivo.

El algoritmo de Shell modifica los saltos contiguos por saltos de mayor tamaño y con ello consigue que la ordenación sea más rápida. Generalmente, se toma como salto inicial n / 2 (siendo n el número de elementos), luego en cada iteración se reduce el salto a la mitad, hasta que el salto es de tamaño 1.

**Figura 4**



*Funcionamiento del algoritmo ShellSort de forma gráfica*

### Algoritmo:

Los pasos a seguir por el algoritmo para una lista de n elementos:

1. Se divide la lista original en n/2 grupos de dos, considerando un incremento o salto entre los elementos de n/2.

2. Se clasifica cada grupo por separado, comparando las parejas de elementos y si no están ordenados se intercambian.

3. Se divide ahora la lista en la mitad de grupos (n/4), con un salto entre los elementos también mitad (n/4), y nuevamente se clasifica cada grupo por separado.

4. Así sucesivamente, se sigue dividiendo la lista en la mitad de grupos que en el recorrido anterior con un salto decreciente en la mitad que el salto anterior, y luego clasificando cada grupo por separado.

5. El algoritmo termina cuando el tamaño del salto es 1.

Por lo tanto, los recorridos por la lista están condicionados por el bucle "for" con la condición de salto, donde se reduce a la mitad en cada iteración. Para dividir la lista en grupos y clasificar cada uno, se utiliza el siguiente código anidado: desde i hasta j con un paso de (salto + 1) hasta n hacer, donde j toma valores desde i hasta n con un paso de salto. Dentro de este bucle, se comparan pares de elementos con índices j y k, separados por una distancia de salto. Si el elemento en la posición j es menor o igual al elemento en la posición k, se intercambian; de lo contrario, no se realiza ningún intercambio. Este proceso continúa hasta recorrer toda la lista. Por ejemplo, cuando n = 8, el primer valor de salto es 4, y los índices i, j y k toman valores como i = 5, j = 1, k = 6. Los siguientes valores que toman son i = 6, j = 2, k = 7, y así sucesivamente hasta completar el recorrido de la lista.

### Codificación

Al codificar el algoritmo se considera que el rango de elementos es 0 ... n-1 y, por consiguiente, se ha de desplazar una posición a la izquierda las variables índice respecto a lo expuesto en el algoritmo.

#include <iostream>  
using **namespace** std;  
  
void **shellSort**(**int** arr[], **int** n) {

// Empieza con un gran intervalo y lo reduce gradualmente

**for** (**int** gap = n / **2**; gap > **0**; gap /= **2**) {  
 // Realiza el ordenamiento de inserción dentro del intervalo actual  
 **for** (**int** i = gap; i < n; i++) {  
 **int** temp = arr[i];  
 **int** j;  
   
 // Mueve los elementos anteriores al elemento seleccionado  
 // hasta que se encuentra la posición adecuada para insertar el elemento  
 **for** (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {  
 arr[j] = arr[j - gap];  
 }  
   
 // Inserta el elemento en su posición correcta  
 arr[j] = temp;  
 }  
 }  
}  
  
// Función para imprimir un arreglo  
void **printArray**(**int** arr[], **int** size) {  
 **for** (**int** i = **0**; i < size; i++) {  
 cout << arr[i] << " ";  
 }  
 cout << endl;  
}  
  
int **main**() {  
 **int** arr[] = {**64**, **25**, **12**, **22**, **11**};  
 **int** n = **sizeof**(arr) / **sizeof**(arr[**0**]);  
  
 cout << "Arreglo original: ";  
 printArray(arr, n);  
  
 shellSort(arr, n);  
  
 cout << "Arreglo ordenado: ";  
 printArray(arr, n);  
  
 **return** **0**;  
}

**Ejecución:**



**Análisis del algoritmo**

Este código implementa el algoritmo Shell Sort en C++. En el programa, la función `shellSort` realiza el ordenamiento utilizando un bucle externo que itera sobre brechas decrecientes y un bucle interno que implementa el ordenamiento por inserción en subarreglos. El algoritmo de Shell Sort mejora el rendimiento del ordenamiento por inserción al reducir el número de desplazamientos necesarios para ubicar elementos en sus posiciones correctas. El programa principal ilustra el uso de esta función en un pequeño arreglo de enteros, imprimiendo el arreglo original y el arreglo ordenado después de aplicar Shell Sort. Este algoritmo es una elección adecuada para ordenar conjuntos de datos de tamaño moderado y tiene un rendimiento aceptable, aunque puede no ser tan eficiente como algunos algoritmos más avanzados en ciertos contextos.

## Ordenación por distribución o Bucketsort

Los algoritmos de ordenamiento por distribución ordenan un arreglo tomando cada número e insertándolo en la posición que toma su valor, es decir, si se tiene un cinco se coloca en la posición cinco del arreglo.

Por lo que, no se pueden ordenar aquellos arreglos que tengan valores repetidos, a su vez el tamaño del arreglo será del número más grande que se encuentre en el mismo. En el caso donde se repitan los mismos valores, se incrementa la capacidad de la posición.

El ordenamiento BucketSort es un algoritmo de clasificación que divide los elementos de matriz no ordenados en varios grupos llamados "buckets". Luego, cada depósito se ordena utilizando cualquiera de los algoritmos de clasificación adecuados o aplicando recursivamente el mismo algoritmo de depósito. Finalmente, los "buckets" ordenados se combinan para formar una matriz ordenada final.

EL proceso de clasificación de "buckets" se entienden desde un enfoque de dispersión y recopilación, ya que los elementos en una primera instancia se dispersan en cubos y luego se clasifican los elementos en cada "bucket", para finalmente recompilarlos y ordenarlos.

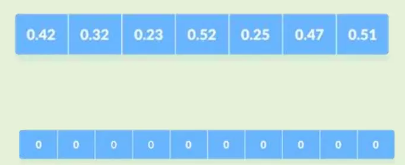
**Ejemplo:**

Si obtenemos la siguiente matriz con valores de entrada:

Primero creamos una matriz de un tamaño de diez casillas. Cada ranura de la matriz dada se usará como un "bucket" para almacenar elementos.

Ya creado nuestra matriz, se insertan los elementos en los respectivos cubos según el rango de cada casilla, que van de (n-1) a n.

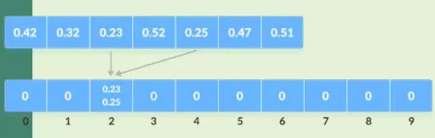
**Figura 4.**



*Creación de los Buckets*

Por ejemplo, si tomamos el elemento de la entrada 0.23, este valor se multiplicaría por el tamaño del arreglo, en nuestro caso 10, realizando el cálculo resulta un valor de 2.3, el cual se aproxima a 2, por lo que se insertará en la casilla 2. Caso contrario, si ingresamos números enteros, son divididos para el intervalo de nuestro tamaño, para obtener un valor mínimo.

**Figura 5.**



*Multiplicando valores*

Luego, se deben ordenar los elementos de cada “Bucket” utilizando cualquiera de Los algoritmos de clasificación estable. En nuestro ejemplo, queda:

**Figura 6.**



*Ordenando los elementos del Bucket*

Finalmente, se reúnen los elementos de cada “Bucket”, lo cual se realiza mediante la iteración a través del mismo “Bucket” e insertando un elemento por elemento en la matriz original, y siendo eliminado en la matriz generada.

**Figura 7.**



*Reunir los elementos del Bucket*

**Algoritmo :**

El algoritmo de Bucket Sort divide el conjunto de elementos en "cubos" o "buckets" según algún criterio, generalmente basado en el valor de los elementos. Luego, cada cubo se ordena individualmente, y finalmente, se concatenan para formar la secuencia ordenada completa. La clave de Bucket Sort es elegir un criterio de distribución que permita una distribución uniforme de los elementos entre los cubos. Este algoritmo puede ser eficaz cuando los elementos están distribuidos de manera relativamente uniforme y la cantidad de elementos en cada cubo es manejable. La eficiencia de Bucket Sort depende en gran medida de la elección del criterio de distribución y de cómo se implementa la ordenación dentro de cada cubo.

**Codificación:**

#include <iostream>  
#include <vector>  
#include <algorithm>  
  
using **namespace** std;  
  
void **bucketSort**(vector<**float**>& arr) {  
 **int** n = arr.size();  
  
 // Crear un vector de buckets  
 vector<vector<**float**>> buckets(n);  
  
 // Colocar los elementos en los buckets  
 **for** (**int** i = **0**; i < n; i++) {  
 **int** bucketIndex = n \* arr[i];  
 buckets[bucketIndex].push\_back(arr[i]);  
 }  
  
 // Ordenar individualmente cada bucket usando cualquier algoritmo de ordenación (aquí se utiliza sort de la STL)  
 **for** (**int** i = **0**; i < n; i++) {  
 sort(buckets[i].begin(), buckets[i].end());  
 }  
  
 // Concatenar los buckets ordenados en el arreglo final  
 **int** index = **0**;  
 **for** (**int** i = **0**; i < n; i++) {  
 **for** (**int** j = **0**; j < buckets[i].size(); j++) {  
 arr[index++] = buckets[i][j];  
 }  
 }  
}  
  
int **main**() {  
 vector<**float**> arr = {**0.42**, **0.32**, **0.33**, **0.52**, **0.37**, **0.47**, **0.51**};  
 **int** n = arr.size();  
  
 cout << "Arreglo original: ";  
 **for** (**int** i = **0**; i < n; i++) {  
 cout << arr[i] << " ";  
 }  
 cout << endl;  
  
 bucketSort(arr);  
  
 cout << "Arreglo ordenado: ";  
 **for** (**int** i = **0**; i < n; i++) {  
 cout << arr[i] << " ";  
 }  
 cout << endl;  
  
 **return** **0**;  
}

**Ejecución:**



**Análisis**

Este código implementa el algoritmo de ordenación Bucket Sort en C++ para ordenar un vector de números en punto flotante. Utiliza la idea de distribuir los elementos en buckets según sus valores normalizados, ordenando cada bucket individualmente y luego concatenando los buckets ordenados para obtener el resultado final. El código utiliza la STL de C++ para simplificar algunas operaciones, como la ordenación de los buckets. Es importante destacar que este ejemplo asume que los elementos del arreglo son números en el rango [0, 1) y utiliza el número de buckets igual al tamaño del arreglo. Para adaptarse a diferentes conjuntos de datos, es necesario ajustar el código según el rango y el número de buckets deseados.

### Pasos del Algoritmo:

* **Encontrar el valor máximo:** El algoritmo determina el valor máximo en el array para conocer el rango de valores posibles.
* **Inicialización de contenedores:** Se crea un array de contenedores, donde cada contenedor corresponde a un rango de valores posibles.
* **Distribución de elementos:** Cada elemento del array de entrada se coloca en el contenedor correspondiente según su valor.
* **Concatenación de contenedores:** Los contenedores se concatenan en orden para obtener el array ordenado.

# Ordenamiento Radix (RadixSort u ordenación por residuos)

Este método de ordenación es un caso particular del algoritmo de clasificación por urnas. La manera de ordenar, manualmente, un conjunto de fichas nos da una idea intuitiva de este método de ordenación: se forman montones de fichas, cada uno caracterizado por tener sus componentes un mismo dígito (letra, si es ordenación alfabéticOPCION 1: en la misma posición.

Inicialmente se forman los montones por las unidades (dígito de menor peso); estos montones se recogen y agrupan en orden ascendente, desde el montón del dígito 0 al montón del dígito 9. Entonces, las fichas están ordenadas respecto a las unidades, a continuación, se vuelve a distribuir las fichas en montones, según el dígito de las decenas.

El proceso de distribuir las fichas por montones y posterior acumulación en orden se repite tantas veces como número de dígitos tiene la ficha de mayor valor. Suponer que las fichas están identificadas por un campo entero de tres dígitos, los pasos del algoritmo RadixSort para los siguientes valores.

## Algoritmo de ordenamiento:

La idea clave de la ordenación RadixSort es clasificar por urnas tantas veces como máximo número de dígitos (o de letras) tengan los elementos de la lista. En cada paso se realiza una distribución en las urnas y una unión de éstas en orden ascendente, desde la urna 0 a la urna 9. Al igual que en el método de BinSort, las urnas se representan mediante un array de listas enlazadas. Se ha de disponer de tantas urnas como dígitos, 10, numeradas de 0 a 9. Si la clave respecto a la que se ordena es alfabética, habrá tantas urnas como letras distintas, desde la urna que representa a la letra a hasta la z. El algoritmo que se escribe, en primer lugar determina el número máximo de dígitos que puede tener una clave. Un bucle externo, de tantas iteraciones como el máximo de dígitos, realiza las acciones de distribuir por urnas los elementos y concatenar.

**Codificación**

#include <iostream>  
#include <vector>  
  
using **namespace** std;  
  
int **getDigit**(**int** num, **int** pos) {  
 **while** (pos > **0**) {  
 num /= **10**;  
 pos--;  
 }  
 **return** num % **10**;  
}  
  
int **getMax**(vector<**int**>& arr) {  
 **int** max = arr[**0**];  
 **for** (**int** i = **1**; i < arr.size(); i++) {  
 **if** (arr[i] > max) {  
 max = arr[i];  
 }  
 }  
 **return** max;  
}  
  
void **radixSort**(vector<**int**>& arr) {  
 **int** max = getMax(arr);  
  
 **for** (**int** pos = **1**; max / pos > **0**; pos \*= **10**) {  
 vector<**int**> output(arr.size());  
 vector<**int**> count(**10**, **0**);  
  
 **for** (**int** i = **0**; i < arr.size(); i++) {  
 count[getDigit(arr[i], pos)]++;  
 }  
  
 **for** (**int** i = **1**; i < **10**; i++) {  
 count[i] += count[i - **1**];  
 }  
  
 **for** (**int** i = arr.size() - **1**; i >= **0**; i--) {  
 output[count[getDigit(arr[i], pos)] - **1**] = arr[i];  
 count[getDigit(arr[i], pos)]--;  
 }  
  
 **for** (**int** i = **0**; i < arr.size(); i++) {  
 arr[i] = output[i];  
 }  
 }  
}  
  
int **main**() {  
 vector<**int**> arr = {**170**, **45**, **75**, **90**, **802**, **24**, **2**, **66**};  
 **int** n = arr.size();  
  
 cout << "Original Array: ";  
 **for** (**int** i = **0**; i < n; i++) {  
 cout << arr[i] << " ";  
 }  
 cout << endl;  
  
 radixSort(arr);  
  
 cout << "Sorted Array: ";  
 **for** (**int** i = **0**; i < n; i++) {  
 cout << arr[i] << " ";  
 }  
 cout << endl;  
  
 **return** **0**;  
}

# Ejecución:



# Recursos complementarios

* Libros

Introduction to Algorithms - 4th - Leiserson – Stein - Rivest – Cormen

Programación en C. Metodología, algoritmos y estructura de datos. Madrid, España: McGraw-Hill Interamericana

* Video sobre Ordenamiento Radix

<https://www.youtube.com/watch?v=W_euZjKoHkM>

* Video sobre Ordenamiento Binsort

<https://www.youtube.com/watch?v=bOY83qCkrkc>

* Video sobre Ordenamiento Burbuja

<https://www.youtube.com/watch?v=Iv9PfEXrbXE>

* Video sobre Ordenamiento Quicksort

<https://www.youtube.com/watch?v=PupaczLYxmA>

# Referencias

Joyanes Aguilar, L. & Zahonero Martínez, I. (2007). *Programación en C. Metodología, algoritmos y estructura de datos.* Madrid, España: McGraw-Hill Interamericana. <https://intprog.files.wordpress.com/2013/08/programacion-en-c-metodologia-algoritmos-y-estructura-de-datos-editorial-mcgraw-hill1.pdf>

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press.

<https://dl.ebooksworld.ir/books/Introduction.to.Algorithms.4th.Leiserson.Stein.Rivest.Cormen.MIT.Press.9780262046305.EBooksWorld.ir.pdf>

# Autoevaluación

**Pregunta 1:**

¿Cuál de los siguientes algoritmos de ordenamiento se caracteriza por comparar y, si es necesario, intercambiar elementos adyacentes en cada paso?

OPCION 1: Quicksort

OPCION 2: ShellSort

OPCION 3: Distribución

OPCION 4: Burbuja

**Pregunta 2:**

¿Cuál de los siguientes algoritmos de ordenamiento utiliza la técnica de dividir el arreglo en subconjuntos y aplicar recursivamente la misma estrategia a cada subconjunto?

OPCION 1: Distribución

OPCION 2: Radix

OPCION 3: Quicksort

OPCION 4: Burbuja

**Pregunta 3:**

¿Cuál de los siguientes algoritmos de ordenamiento es conocido por ser eficiente para ordenar grandes cantidades de datos, especialmente cuando los elementos tienen un gran número de dígitos?

OPCION 1: Distribución

OPCION 2: Quicksort

OPCION 3: ShellSort

OPCION 4: Radix

**Pregunta 4:**

¿Qué algoritmo de ordenamiento es conocido por tener un rendimiento eficiente en el mejor caso, pero sufre en el peor caso cuando el arreglo ya está parcialmente ordenado?

OPCION 1: Quicksort

OPCION 2: ShellSort

OPCION 3: Burbuja

OPCION 4: Radix

**Pregunta 5:**

En la inserción de elementos en el ordenamiento BucketSort, ¿cómo se define el rango de cada casilla?

OPCION 1: De (n-1) a n

OPCION 2: De (n-2) a 2n

OPCION 3: de n a n+1

**Pregunta 6:**

Si el dato de entrada es un valor decimal, ¿Como se calcula el rango para conocer la casilla donde se deberá insertar?

OPCION 1: Se multiplica por el tamaño del arreglo, y luego se aproxima al número más cercano.

OPCION 2: Se divide para el tamaño del arreglo y luego se aproxima al número más cercano.

OPCION 3: Únicamente se aproxima al número más cercano

**Pregunta 7:**

Si el dato de entrada es un valor entero, ¿Como se calcula el rango para conocer la casilla donde se deberá insertar?

OPCION 1: Se multiplica por el tamaño del arreglo, y luego se aproxima al número más cercano.

OPCION 2: Se divide para el tamaño del arreglo, y luego se aproxima al número más cercano.

OPCION 3: Únicamente se aproxima al número más cercano.

**Pregunta 8:**

¿Qué define la estabilidad en un algoritmo de ordenamiento?

OPCION 1: La eficiencia en el mejor caso.

OPCION 2: La capacidad de ordenar grandes conjuntos de datos.

OPCION 3: La capacidad de mantener el orden relativo de elementos con valores iguales.

OPCION 4: El uso de la estrategia de divide y conquista.

**Pregunta 9:**

¿Cuál es una característica clave del algoritmo de ordenamiento ShellSort?

OPCION 1: Utiliza la estrategia de divide y conquista.

OPCION 2: Selecciona un pivote y organiza los elementos alrededor de él.

OPCION 3:Utiliza distancias fijas para comparar y mover elementos.

OPCION 4: Organiza los elementos en subconjuntos mediante la técnica de distribución.

**Pregunta 10:**

En el algoritmo de Radix Sort, ¿en qué se basa la clasificación de los elementos?

OPCION 1: Comparación de elementos mediante distancias fijas.

OPCION 2: Divide y conquista.

OPCION 3: Dígitos individuales o grupos de dígitos.

OPCION 4: Distribución de elementos en subconjuntos.