

Capítulo 1

Algoritmos de ordenamiento

1.1. Ordenamiento por comparación

1.1.1. Burbuja (Bubble Sort)

El algoritmo de ordenamiento por burbuja (Bubble Sort) es uno de los algoritmos de ordenamiento más simples. Consiste en recorrer repetidamente la lista a ordenar, comparando elementos adyacentes e intercambiándolos si están en el orden incorrecto. Este proceso se repite hasta que no se requieran más intercambios.

Análisis de Complejidad

- **Mejor caso:** $O(n)$ (cuando la lista ya está ordenada).
- **Peor caso:** $O(n^2)$ (cuando la lista está ordenada en orden inverso).
- **Caso promedio:** $O(n^2)$.

Pseudocódigo

```
BubbleSort(A)
  n ← longitud(A)
  repetir
    intercambiado ← falso
    para i ← 0 hasta n-2 hacer
      si A[i] > A[i+1] entonces
        intercambiar A[i] y A[i+1]
      intercambiado ← verdadero
```

```
        fin si
    fin para
    hasta que intercambiado = falso
fin
```

Implementación en C

```
1 #include <stdio.h>
2
3 void bubbleSort(int arr[], int n) {
4     int temp;
5     for (int i = 0; i < n-1; i++) {
6         int swapped = 0;
7         for (int j = 0; j < n-i-1; j++) {
8             if (arr[j] > arr[j+1]) {
9                 // Intercambiar
10                temp = arr[j];
11                arr[j] = arr[j+1];
12                arr[j+1] = temp;
13                swapped = 1;
14            }
15        }
16        if (!swapped) break;
17    }
18 }
19
20 void printArray(int arr[], int n) {
21     for (int i = 0; i < n; i++) {
22         printf("%d ", arr[i]);
23     }
24     printf("\n");
25 }
26
27 int main() {
28     int arr[] = {64, 34, 25, 12, 22, 11, 90};
29     int n = sizeof(arr)/sizeof(arr[0]);
30     bubbleSort(arr, n);
31     printf("Array ordenado: \n");
32     printArray(arr, n);
33     return 0;
34 }
```

Implementación en Python

```

1 def bubble_sort(arr):
2     n = len(arr)
3     for i in range(n-1):
4         swapped = False
5         for j in range(n-i-1):
6             if arr[j] > arr[j+1]:
7                 # Intercambiar
8                 arr[j], arr[j+1] = arr[j+1], arr[j]
9                 swapped = True
10        if not swapped:
11            break
12
13 # Ejemplo de uso
14 arr = [64, 34, 25, 12, 22, 11, 90]
15 bubble_sort(arr)
16 print("Array ordenado:", arr)

```

1.1.2. Inserción (Insertion Sort)

El algoritmo de ordenamiento por inserción (Insertion Sort) ordena una lista construyendo gradualmente una porción ordenada. En cada iteración, un elemento se extrae de la parte desordenada y se inserta en la posición correcta dentro de la parte ordenada.

Análisis de Complejidad

- **Mejor caso:** $O(n)$ (cuando la lista ya está ordenada).
- **Peor caso:** $O(n^2)$ (cuando la lista está ordenada en orden inverso).
- **Caso promedio:** $O(n^2)$.

Pseudocódigo

```

InsertionSort(A)
  n ← longitud(A)
  para i ← 1 hasta n-1 hacer
    clave ← A[i]
    j ← i - 1
    mientras j >= 0 y A[j] > clave hacer
      A[j+1] ← A[j]

```

```
        j ← j - 1
    fin mientras
    A[j+1] ← clave
fin para
fin
```

Implementación en C

```
1  #include <stdio.h>
2
3  void insertionSort(int arr[], int n) {
4      for (int i = 1; i < n; i++) {
5          int key = arr[i];
6          int j = i - 1;
7          // Mover elementos mayores que la clave hacia la
           derecha
8          while (j >= 0 && arr[j] > key) {
9              arr[j + 1] = arr[j];
10             j--;
11         }
12         arr[j + 1] = key;
13     }
14 }
15
16 void printArray(int arr[], int n) {
17     for (int i = 0; i < n; i++) {
18         printf("%d ", arr[i]);
19     }
20     printf("\n");
21 }
22
23 int main() {
24     int arr[] = {12, 11, 13, 5, 6};
25     int n = sizeof(arr)/sizeof(arr[0]);
26     insertionSort(arr, n);
27     printf("Array ordenado: \n");
28     printArray(arr, n);
29     return 0;
30 }
```

Implementación en Python

```

1 def insertion_sort(arr):
2     for i in range(1, len(arr)):
3         key = arr[i]
4         j = i - 1
5         # Mover elementos mayores que la clave hacia la
           derecha
6         while j >= 0 and arr[j] > key:
7             arr[j + 1] = arr[j]
8             j -= 1
9         arr[j + 1] = key
10
11 # Ejemplo de uso
12 arr = [12, 11, 13, 5, 6]
13 insertion_sort(arr)
14 print("Array ordenado:", arr)

```

1.1.3. Selección (Selection Sort)

El algoritmo de ordenamiento por selección (Selection Sort) divide la lista en dos partes: una parte ordenada y otra desordenada. En cada iteración, encuentra el elemento más pequeño (o más grande, según el orden deseado) de la parte desordenada y lo intercambia con el primer elemento de esta parte.

Análisis de Complejidad

- Mejor caso: $O(n^2)$.
- Peor caso: $O(n^2)$.
- Caso promedio: $O(n^2)$.

Pseudocódigo

```

SelectionSort(A)
  n ← longitud(A)
  para i ← 0 hasta n-2 hacer
    min_idx ← i
    para j ← i+1 hasta n-1 hacer
      si A[j] < A[min_idx] entonces
        min_idx ← j
    fin si
  fin para

```

```

    fin para
    intercambiar A[i] y A[min_idx]
  fin para
fin

```

Implementación en C

```

1  #include <stdio.h>
2
3  void selectionSort(int arr[], int n) {
4      for (int i = 0; i < n - 1; i++) {
5          int min_idx = i;
6          for (int j = i + 1; j < n; j++) {
7              if (arr[j] < arr[min_idx]) {
8                  min_idx = j;
9              }
10         }
11         // Intercambiar el minimo con el primer elemento
           desordenado
12         int temp = arr[min_idx];
13         arr[min_idx] = arr[i];
14         arr[i] = temp;
15     }
16 }
17
18 void printArray(int arr[], int n) {
19     for (int i = 0; i < n; i++) {
20         printf("%d ", arr[i]);
21     }
22     printf("\n");
23 }
24
25 int main() {
26     int arr[] = {64, 25, 12, 22, 11};
27     int n = sizeof(arr)/sizeof(arr[0]);
28     selectionSort(arr, n);
29     printf("Array ordenado: \n");
30     printArray(arr, n);
31     return 0;
32 }

```

Implementación en Python

```

1 def selection_sort(arr):
2     n = len(arr)
3     for i in range(n):
4         min_idx = i
5         for j in range(i + 1, n):
6             if arr[j] < arr[min_idx]:
7                 min_idx = j
8             # Intercambiar el minimo con el primer elemento
              desordenado
9         arr[i], arr[min_idx] = arr[min_idx], arr[i]
10
11 # Ejemplo de uso
12 arr = [64, 25, 12, 22, 11]
13 selection_sort(arr)
14 print("Array ordenado:", arr)

```

1.1.4. Shell Sort

El algoritmo Shell Sort es una mejora del algoritmo de inserción. Se basa en comparar y mover elementos que están separados por un cierto *gap* (o intervalo). A medida que el algoritmo avanza, el *gap* se reduce gradualmente hasta que se convierte en 1, momento en el cual el algoritmo actúa como un ordenamiento por inserción.

Análisis de Complejidad

- **Mejor caso:** $O(n \log n)$ (dependiendo de la secuencia de incrementos).
- **Peor caso:** $O(n^2)$ (con una mala elección del *gap*).
- **Caso promedio:** Depende de la secuencia de incrementos, pero generalmente mejor que $O(n^2)$.

Pseudocódigo

```

ShellSort(A)
  n ← longitud(A)
  gap ← n // 2
  mientras gap > 0 hacer
    para i ← gap hasta n-1 hacer
      temp ← A[i]
      j ← i

```

```
    mientras j >= gap y A[j-gap] > temp hacer
        A[j] ← A[j-gap]
        j ← j - gap
    fin mientras
    A[j] ← temp
fin para
gap ← gap // 2
fin mientras
fin
```

Implementación en C

```
1  #include <stdio.h>
2
3  void shellSort(int arr[], int n) {
4      for (int gap = n / 2; gap > 0; gap /= 2) {
5          for (int i = gap; i < n; i++) {
6              int temp = arr[i];
7              int j;
8              for (j = i; j >= gap && arr[j - gap] > temp;
9                  j -= gap) {
10                 arr[j] = arr[j - gap];
11             }
12             arr[j] = temp;
13         }
14     }
15
16     void printArray(int arr[], int n) {
17         for (int i = 0; i < n; i++) {
18             printf("%d ", arr[i]);
19         }
20         printf("\n");
21     }
22
23     int main() {
24         int arr[] = {12, 34, 54, 2, 3};
25         int n = sizeof(arr) / sizeof(arr[0]);
26         shellSort(arr, n);
27         printf("Array ordenado: \n");
28         printArray(arr, n);
29         return 0;
30     }
```


Implementación en Python

```

1 def shell_sort(arr):
2     n = len(arr)
3     gap = n // 2
4     while gap > 0:
5         for i in range(gap, n):
6             temp = arr[i]
7             j = i
8             while j >= gap and arr[j - gap] > temp:
9                 arr[j] = arr[j - gap]
10                j -= gap
11            arr[j] = temp
12        gap //= 2
13
14 # Ejemplo de uso
15 arr = [12, 34, 54, 2, 3]
16 shell_sort(arr)
17 print("Array ordenado:", arr)

```

1.1.5. Merge Sort

El algoritmo Merge Sort es un algoritmo de ordenamiento basado en el paradigma divide y vencerás. Divide repetidamente la lista en mitades más pequeñas hasta que cada sublista tiene un solo elemento y luego combina estas sublistas de forma ordenada para formar la lista completa.

Análisis de Complejidad

- Mejor caso: $O(n \log n)$.
- Peor caso: $O(n \log n)$.
- Caso promedio: $O(n \log n)$.

Pseudocódigo

```

MergeSort(A, inicio, fin)
    si inicio < fin entonces
        medio ← (inicio + fin) // 2
        MergeSort(A, inicio, medio)
        MergeSort(A, medio+1, fin)
        Mezclar(A, inicio, medio, fin)

```

```

    fin si

Mezclar(A, inicio, medio, fin)
    n1 ← medio - inicio + 1
    n2 ← fin - medio
    L ← subarray(A, inicio, medio)
    R ← subarray(A, medio+1, fin)
    i ← 0, j ← 0, k ← inicio
    mientras i < n1 y j < n2 hacer
        si L[i] <= R[j] entonces
            A[k] ← L[i]
            i ← i + 1
        si no
            A[k] ← R[j]
            j ← j + 1
        fin si
        k ← k + 1
    fin mientras
    copiar elementos restantes de L y R en A
fin

```

Implementación en C

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void merge(int arr[], int left, int mid, int right) {
5     int n1 = mid - left + 1;
6     int n2 = right - mid;
7
8     int L[n1], R[n2];
9     for (int i = 0; i < n1; i++)
10         L[i] = arr[left + i];
11     for (int j = 0; j < n2; j++)
12         R[j] = arr[mid + 1 + j];
13
14     int i = 0, j = 0, k = left;
15     while (i < n1 && j < n2) {
16         if (L[i] <= R[j]) {
17             arr[k] = L[i];
18             i++;
19         } else {

```

```
20         arr[k] = R[j];
21         j++;
22     }
23     k++;
24 }
25
26 while (i < n1) {
27     arr[k] = L[i];
28     i++;
29     k++;
30 }
31
32 while (j < n2) {
33     arr[k] = R[j];
34     j++;
35     k++;
36 }
37 }
38
39 void mergeSort(int arr[], int left, int right) {
40     if (left < right) {
41         int mid = left + (right - left) / 2;
42         mergeSort(arr, left, mid);
43         mergeSort(arr, mid + 1, right);
44         merge(arr, left, mid, right);
45     }
46 }
47
48 void printArray(int arr[], int size) {
49     for (int i = 0; i < size; i++)
50         printf("%d_", arr[i]);
51     printf("\n");
52 }
53
54 int main() {
55     int arr[] = {12, 11, 13, 5, 6, 7};
56     int arr_size = sizeof(arr) / sizeof(arr[0]);
57
58     printf("Array original: \n");
59     printArray(arr, arr_size);
60
61     mergeSort(arr, 0, arr_size - 1);
62
63     printf("Array ordenado: \n");
64     printArray(arr, arr_size);
```

```
65     return 0;  
66 }
```

Implementación en Python

```
1 def merge_sort(arr):  
2     if len(arr) > 1:  
3         mid = len(arr) // 2  
4         L = arr[:mid]  
5         R = arr[mid:]  
6  
7         merge_sort(L)  
8         merge_sort(R)  
9  
10        i = j = k = 0  
11  
12        while i < len(L) and j < len(R):  
13            if L[i] <= R[j]:  
14                arr[k] = L[i]  
15                i += 1  
16            else:  
17                arr[k] = R[j]  
18                j += 1  
19            k += 1  
20  
21        while i < len(L):  
22            arr[k] = L[i]  
23            i += 1  
24            k += 1  
25  
26        while j < len(R):  
27            arr[k] = R[j]  
28            j += 1  
29            k += 1  
30  
31 # Ejemplo de uso  
32 arr = [12, 11, 13, 5, 6, 7]  
33 print("Array original:", arr)  
34 merge_sort(arr)  
35 print("Array ordenado:", arr)
```

1.1.6. Quick Sort

Quick Sort es un algoritmo de ordenamiento basado en el paradigma divide y vencerás. Selecciona un elemento como *pivote* y particiona el array en dos subarrays: uno con elementos menores al pivote y otro con elementos mayores. Luego, aplica Quick Sort recursivamente a ambos subarrays.

Análisis de Complejidad

- **Mejor caso:** $O(n \log n)$ (cuando el pivote divide el array en partes iguales).
- **Peor caso:** $O(n^2)$ (cuando el pivote es el elemento más grande o más pequeño).
- **Caso promedio:** $O(n \log n)$.

Pseudocódigo

```
QuickSort(A, inicio, fin)
  si inicio < fin entonces
    pivote ← Particionar(A, inicio, fin)
    QuickSort(A, inicio, pivote-1)
    QuickSort(A, pivote+1, fin)
  fin si
```

```
Particionar(A, inicio, fin)
  pivote ← A[fin]
  i ← inicio - 1
  para j ← inicio hasta fin-1 hacer
    si A[j] ≤ pivote entonces
      i ← i + 1
      intercambiar A[i] y A[j]
  fin si
  fin para
  intercambiar A[i+1] y A[fin]
  retornar i+1
fin
```

Implementación en C

```
1 #include <stdio.h>
2
3 void swap(int* a, int* b) {
4     int temp = *a;
5     *a = *b;
6     *b = temp;
7 }
8
9 int partition(int arr[], int low, int high) {
10     int pivot = arr[high];
11     int i = low - 1;
12
13     for (int j = low; j < high; j++) {
14         if (arr[j] <= pivot) {
15             i++;
16             swap(&arr[i], &arr[j]);
17         }
18     }
19     swap(&arr[i + 1], &arr[high]);
20     return i + 1;
21 }
22
23 void quickSort(int arr[], int low, int high) {
24     if (low < high) {
25         int pi = partition(arr, low, high);
26         quickSort(arr, low, pi - 1);
27         quickSort(arr, pi + 1, high);
28     }
29 }
30
31 void printArray(int arr[], int size) {
32     for (int i = 0; i < size; i++)
33         printf("%d ", arr[i]);
34     printf("\n");
35 }
36
37 int main() {
38     int arr[] = {10, 7, 8, 9, 1, 5};
39     int n = sizeof(arr) / sizeof(arr[0]);
40     quickSort(arr, 0, n - 1);
41     printf("Array ordenado: \n");
42     printArray(arr, n);
43     return 0;
44 }
```

Implementación en Python

```
1 def partition(arr, low, high):
2     pivot = arr[high]
3     i = low - 1
4
5     for j in range(low, high):
6         if arr[j] <= pivot:
7             i += 1
8             arr[i], arr[j] = arr[j], arr[i]
9     arr[i + 1], arr[high] = arr[high], arr[i + 1]
10    return i + 1
11
12 def quick_sort(arr, low, high):
13     if low < high:
14         pi = partition(arr, low, high)
15         quick_sort(arr, low, pi - 1)
16         quick_sort(arr, pi + 1, high)
17
18 # Ejemplo de uso
19 arr = [10, 7, 8, 9, 1, 5]
20 quick_sort(arr, 0, len(arr) - 1)
21 print("Array ordenado:", arr)
```

1.1.7. Heap Sort

Heap Sort es un algoritmo de ordenamiento basado en una estructura de datos llamada *heap* binario. Utiliza un *max-heap* (para ordenar de manera ascendente) o un *min-heap* (para ordenar de manera descendente) para construir un árbol binario completo en el que cada nodo padre es mayor (o menor) que sus hijos. Una vez construido el *heap*, el elemento más grande (la raíz) se coloca en su posición correcta y se reorganiza el *heap* para los elementos restantes.

Análisis de Complejidad

- Mejor caso: $O(n \log n)$.
- Peor caso: $O(n \log n)$.
- Caso promedio: $O(n \log n)$.

Pseudocódigo

```
HeapSort(A)
    construirMaxHeap(A)
    para i ← longitud(A)-1 hasta 1 hacer
        intercambiar A[0] y A[i]
        maxHeapify(A, 0, i)
    fin para
fin

construirMaxHeap(A)
    para i ← longitud(A)//2-1 hasta 0 hacer
        maxHeapify(A, i, longitud(A))
    fin para
fin

maxHeapify(A, i, n)
    izquierda ← 2*i + 1
    derecha ← 2*i + 2
    mayor ← i
    si izquierda < n y A[izquierda] > A[mayor] entonces
        mayor ← izquierda
    fin si
    si derecha < n y A[derecha] > A[mayor] entonces
        mayor ← derecha
    fin si
    si mayor != i entonces
        intercambiar A[i] y A[mayor]
        maxHeapify(A, mayor, n)
    fin si
fin
```

Implementación en C

```
1 #include <stdio.h>
2
3 void swap(int* a, int* b) {
4     int temp = *a;
5     *a = *b;
6     *b = temp;
7 }
```



```
8
9 void heapify(int arr[], int n, int i) {
10     int largest = i;
11     int left = 2 * i + 1;
12     int right = 2 * i + 2;
13
14     if (left < n && arr[left] > arr[largest])
15         largest = left;
16
17     if (right < n && arr[right] > arr[largest])
18         largest = right;
19
20     if (largest != i) {
21         swap(&arr[i], &arr[largest]);
22         heapify(arr, n, largest);
23     }
24 }
25
26 void heapSort(int arr[], int n) {
27     for (int i = n / 2 - 1; i >= 0; i--)
28         heapify(arr, n, i);
29
30     for (int i = n - 1; i >= 0; i--) {
31         swap(&arr[0], &arr[i]);
32         heapify(arr, i, 0);
33     }
34 }
35
36 void printArray(int arr[], int n) {
37     for (int i = 0; i < n; i++)
38         printf("%d_", arr[i]);
39     printf("\n");
40 }
41
42 int main() {
43     int arr[] = {12, 11, 13, 5, 6, 7};
44     int n = sizeof(arr) / sizeof(arr[0]);
45
46     heapSort(arr, n);
47
48     printf("Array_ordenado:_\n");
49     printArray(arr, n);
50     return 0;
51 }
```

Implementación en Python

```
1 def heapify(arr, n, i):
2     largest = i
3     left = 2 * i + 1
4     right = 2 * i + 2
5
6     if left < n and arr[left] > arr[largest]:
7         largest = left
8
9     if right < n and arr[right] > arr[largest]:
10        largest = right
11
12    if largest != i:
13        arr[i], arr[largest] = arr[largest], arr[i]
14        heapify(arr, n, largest)
15
16 def heap_sort(arr):
17     n = len(arr)
18
19     for i in range(n // 2 - 1, -1, -1):
20         heapify(arr, n, i)
21
22     for i in range(n - 1, 0, -1):
23         arr[0], arr[i] = arr[i], arr[0]
24         heapify(arr, i, 0)
25
26 # Ejemplo de uso
27 arr = [12, 11, 13, 5, 6, 7]
28 heap_sort(arr)
29 print("Array ordenado:", arr)
```

1.1.8. TimSort

TimSort es un algoritmo híbrido que combina *Insertion Sort* y *Merge Sort*. Divide el array en pequeñas corridas (*runs*), las ordena con *Insertion Sort* y luego combina estas corridas usando *Merge Sort*. Está diseñado para ser eficiente en datos reales.

Análisis de Complejidad

- **Mejor caso:** $O(n)$ (cuando los datos ya están ordenados).
- **Peor caso:** $O(n \log n)$.

- **Caso promedio:** $O(n \log n)$.

Pseudocódigo

```
TimSort(A)
  MIN_RUN ← calcularMinRun(longitud(A))
  para cada subarray en A de tamaño MIN_RUN hacer
    usarInsertionSort(subarray)
  fin para
  tamaño ← MIN_RUN
  mientras tamaño < longitud(A) hacer
    para cada par de subarrays de tamaño "tamaño" en A hacer
      mezclar(subarray1, subarray2)
    fin para
    tamaño ← 2 * tamaño
  fin mientras
fin
```

Implementación en C

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define MIN_RUN 32
6
7 void insertionSort(int arr[], int left, int right) {
8     for (int i = left + 1; i <= right; i++) {
9         int temp = arr[i];
10        int j = i - 1;
11        while (j >= left && arr[j] > temp) {
12            arr[j + 1] = arr[j];
13            j--;
14        }
15        arr[j + 1] = temp;
16    }
17 }
18
19 void merge(int arr[], int left, int mid, int right) {
20     int len1 = mid - left + 1, len2 = right - mid;
21     int *leftArr = (int *)malloc(len1 * sizeof(int));
22     int *rightArr = (int *)malloc(len2 * sizeof(int));
23 }
```

```
24     for (int i = 0; i < len1; i++)
25         leftArr[i] = arr[left + i];
26     for (int i = 0; i < len2; i++)
27         rightArr[i] = arr[mid + 1 + i];
28
29     int i = 0, j = 0, k = left;
30     while (i < len1 && j < len2) {
31         if (leftArr[i] <= rightArr[j])
32             arr[k++] = leftArr[i++];
33         else
34             arr[k++] = rightArr[j++];
35     }
36
37     while (i < len1)
38         arr[k++] = leftArr[i++];
39     while (j < len2)
40         arr[k++] = rightArr[j++];
41
42     free(leftArr);
43     free(rightArr);
44 }
45
46 void timSort(int arr[], int n) {
47     for (int i = 0; i < n; i += MIN_RUN) {
48         int end = (i + MIN_RUN - 1 < n) ? i + MIN_RUN - 1
49             : n - 1;
50         insertionSort(arr, i, end);
51     }
52
53     for (int size = MIN_RUN; size < n; size = 2 * size) {
54         for (int left = 0; left < n; left += 2 * size) {
55             int mid = (left + size - 1 < n) ? left + size
56                 - 1 : n - 1;
57             int right = (left + 2 * size - 1 < n) ? left
58                 + 2 * size - 1 : n - 1;
59
60             if (mid < right)
61                 merge(arr, left, mid, right);
62         }
63     }
64 }
65
66 void printArray(int arr[], int n) {
67     for (int i = 0; i < n; i++)
68         printf("%d ", arr[i]);
```

```
66     printf("\n");
67 }
68
69 int main() {
70     int arr[] = {5, 21, 7, 23, 19, 4, 2, 8};
71     int n = sizeof(arr) / sizeof(arr[0]);
72
73     printf("Array original:\n");
74     printArray(arr, n);
75
76     timSort(arr, n);
77
78     printf("Array ordenado:\n");
79     printArray(arr, n);
80
81     return 0;
82 }
```

Implementación en Python

```
1 MIN_RUN = 32
2
3 def insertion_sort(arr, left, right):
4     for i in range(left + 1, right + 1):
5         key = arr[i]
6         j = i - 1
7         while j >= left and arr[j] > key:
8             arr[j + 1] = arr[j]
9             j -= 1
10        arr[j + 1] = key
11
12 def merge(arr, left, mid, right):
13     left_part = arr[left:mid + 1]
14     right_part = arr[mid + 1:right + 1]
15
16     i = j = 0
17     k = left
18
19     while i < len(left_part) and j < len(right_part):
20         if left_part[i] <= right_part[j]:
21             arr[k] = left_part[i]
22             i += 1
23         else:
24             arr[k] = right_part[j]
```

```

25         j += 1
26         k += 1
27
28     while i < len(left_part):
29         arr[k] = left_part[i]
30         i += 1
31         k += 1
32
33     while j < len(right_part):
34         arr[k] = right_part[j]
35         j += 1
36         k += 1
37
38 def tim_sort(arr):
39     n = len(arr)
40     for start in range(0, n, MIN_RUN):
41         end = min(start + MIN_RUN - 1, n - 1)
42         insertion_sort(arr, start, end)
43
44     size = MIN_RUN
45     while size < n:
46         for left in range(0, n, size * 2):
47             mid = min(n - 1, left + size - 1)
48             right = min((left + 2 * size - 1), n - 1)
49
50             if mid < right:
51                 merge(arr, left, mid, right)
52
53         size *= 2
54
55 # Ejemplo de uso
56 arr = [5, 21, 7, 23, 19, 4, 2, 8]
57 tim_sort(arr)
58 print("Array ordenado:", arr)

```

1.1.9. Tree Sort

Tree Sort es un algoritmo de ordenamiento basado en un árbol binario de búsqueda (BST). Consiste en insertar todos los elementos en un árbol binario y luego realizar un recorrido *in-order* del árbol para recuperar los elementos en orden ascendente.

Análisis de Complejidad

- **Mejor caso:** $O(n \log n)$ (cuando el árbol está equilibrado).
- **Peor caso:** $O(n^2)$ (cuando el árbol es completamente desbalanceado, como una lista enlazada).
- **Caso promedio:** $O(n \log n)$.

Pseudocódigo

```
TreeSort(A)
  Crear un arbol binario vacio
  para cada elemento en A hacer
    Insertar el elemento en el arbol
  fin para
  Realizar un recorrido in-order del arbol
  Guardar los elementos en A
fin
```

Implementación en C

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 // Definicion del nodo del arbol binario
5 struct Node {
6     int data;
7     struct Node* left;
8     struct Node* right;
9 };
10
11 // Crear un nuevo nodo
12 struct Node* newNode(int data) {
13     struct Node* node = (struct Node*)malloc(sizeof(
14         struct Node));
15     node->data = data;
16     node->left = node->right = NULL;
17     return node;
18 }
19 // Insertar un nodo en el arbol
20 struct Node* insert(struct Node* node, int data) {
21     if (node == NULL)
```

```
22         return newNode(data);
23     if (data < node->data)
24         node->left = insert(node->left, data);
25     else if (data > node->data)
26         node->right = insert(node->right, data);
27     return node;
28 }
29
30 // Recorrido in-order del arbol
31 void inorder(struct Node* root, int arr[], int* index) {
32     if (root != NULL) {
33         inorder(root->left, arr, index);
34         arr[*index++] = root->data;
35         inorder(root->right, arr, index);
36     }
37 }
38
39 // Tree Sort
40 void treeSort(int arr[], int n) {
41     struct Node* root = NULL;
42
43     // Insertar elementos en el arbol
44     for (int i = 0; i < n; i++) {
45         root = insert(root, arr[i]);
46     }
47
48     // Recorrido in-order para ordenar
49     int index = 0;
50     inorder(root, arr, &index);
51 }
52
53 // Imprimir un array
54 void printArray(int arr[], int size) {
55     for (int i = 0; i < size; i++) {
56         printf("%d ", arr[i]);
57     }
58     printf("\n");
59 }
60
61 int main() {
62     int arr[] = {5, 3, 7, 1, 9, 4, 6};
63     int n = sizeof(arr) / sizeof(arr[0]);
64
65     printf("Array original:\n");
66     printArray(arr, n);
```



```
67
68     treeSort(arr, n);
69
70     printf("Array ordenado:\n");
71     printArray(arr, n);
72
73     return 0;
74 }
```

Implementación en Python

```
1 class Node:
2     def __init__(self, key):
3         self.key = key
4         self.left = None
5         self.right = None
6
7 # Insertar un nodo en el arbol
8 def insert(root, key):
9     if root is None:
10         return Node(key)
11     if key < root.key:
12         root.left = insert(root.left, key)
13     elif key > root.key:
14         root.right = insert(root.right, key)
15     return root
16
17 # Recorrido in-order del arbol
18 def inorder(root, sorted_list):
19     if root is not None:
20         inorder(root.left, sorted_list)
21         sorted_list.append(root.key)
22         inorder(root.right, sorted_list)
23
24 # Tree Sort
25 def tree_sort(arr):
26     if not arr:
27         return []
28
29     root = None
30     for key in arr:
31         root = insert(root, key)
32
33     sorted_list = []
```

```

34     inorder(root, sorted_list)
35     return sorted_list
36
37 # Ejemplo de uso
38 arr = [5, 3, 7, 1, 9, 4, 6]
39 print("Array original:", arr)
40 sorted_arr = tree_sort(arr)
41 print("Array ordenado:", sorted_arr)

```

1.2. Ordenamiento no comparativo

1.2.1. Counting Sort

Counting Sort es un algoritmo de ordenamiento no comparativo que se basa en contar el número de ocurrencias de cada valor. Es adecuado para ordenar arrays de enteros en un rango definido. Los elementos se cuentan, y estas cuentas se utilizan para colocar los elementos en su posición correcta en el array ordenado.

Análisis de Complejidad

- Mejor caso: $O(n + k)$, donde k es el rango de valores.
- Peor caso: $O(n + k)$.
- Caso promedio: $O(n + k)$.

Pseudocódigo

```

CountingSort(A, rango)
    Crear un array de conteo C de tamaño rango y llenarlo con ceros
    Crear un array de salida B del mismo tamaño que A
    para cada elemento x en A hacer
        Incrementar C[x]
    fin para
    para i ← 1 hasta rango-1 hacer
        C[i] ← C[i] + C[i-1]
    fin para
    para cada elemento x en A, en orden inverso, hacer
        B[C[x]-1] ← x
        C[x] ← C[x] - 1

```

```
    fin para
    Copiar elementos de B a A
fin
```

Implementación en C

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 void countingSort(int arr[], int n, int range) {
6     int* count = (int*)calloc(range, sizeof(int));
7     int* output = (int*)malloc(n * sizeof(int));
8
9     // Contar las ocurrencias
10    for (int i = 0; i < n; i++) {
11        count[arr[i]]++;
12    }
13
14    // Acumular las posiciones
15    for (int i = 1; i < range; i++) {
16        count[i] += count[i - 1];
17    }
18
19    // Construir el array ordenado
20    for (int i = n - 1; i >= 0; i--) {
21        output[count[arr[i]] - 1] = arr[i];
22        count[arr[i]]--;
23    }
24
25    // Copiar el array ordenado al original
26    for (int i = 0; i < n; i++) {
27        arr[i] = output[i];
28    }
29
30    free(count);
31    free(output);
32 }
33
34 void printArray(int arr[], int n) {
35     for (int i = 0; i < n; i++) {
36         printf("%d ", arr[i]);
37     }
38     printf("\n");
39 }
```

```
39 }
40
41 int main() {
42     int arr[] = {4, 2, 2, 8, 3, 3, 1};
43     int n = sizeof(arr) / sizeof(arr[0]);
44     int range = 9; // Rango de los valores (0 a 8)
45
46     printf("Array original:\n");
47     printArray(arr, n);
48
49     countingSort(arr, n, range);
50
51     printf("Array ordenado:\n");
52     printArray(arr, n);
53
54     return 0;
55 }
```

Implementación en Python

```
1 def counting_sort(arr, max_val):
2     n = len(arr)
3     count = [0] * (max_val + 1)
4     output = [0] * n
5
6     # Contar las ocurrencias
7     for num in arr:
8         count[num] += 1
9
10    # Acumular las posiciones
11    for i in range(1, len(count)):
12        count[i] += count[i - 1]
13
14    # Construir el array ordenado
15    for num in reversed(arr):
16        output[count[num] - 1] = num
17        count[num] -= 1
18
19    return output
20
21 # Ejemplo de uso
22 arr = [4, 2, 2, 8, 3, 3, 1]
23 print("Array original:", arr)
24 sorted_arr = counting_sort(arr, max(arr))
```

```
25 | print("Array ordenado:", sorted_arr)
```

1.2.2. Radix Sort

Radix Sort es un algoritmo de ordenamiento no comparativo que ordena los números agrupándolos por dígitos significativos (de menor a mayor o viceversa). Generalmente, utiliza Counting Sort como subrutina para clasificar los dígitos en cada posición.

Análisis de Complejidad

- **Mejor caso:** $O(nk)$, donde n es el número de elementos y k es el número de dígitos.
- **Peor caso:** $O(nk)$.
- **Caso promedio:** $O(nk)$.

Pseudocódigo

```
RadixSort(A, d)
  para i ← 0 hasta d-1 hacer
    UsarCountingSort(A, dígito i)
  fin para
fin
```

Implementación en C

```
1 | #include <stdio.h>
2 | #include <stdlib.h>
3 |
4 | // Obtener el dígito mas significativo en la posicion exp
5 | int getMax(int arr[], int n) {
6 |     int max = arr[0];
7 |     for (int i = 1; i < n; i++) {
8 |         if (arr[i] > max) {
9 |             max = arr[i];
10 |        }
11 |    }
12 |    return max;
13 | }
14 |
```

```
15 // Counting Sort modificado para Radix Sort
16 void countingSort(int arr[], int n, int exp) {
17     int* output = (int*)malloc(n * sizeof(int));
18     int count[10] = {0};
19
20     // Contar ocurrencias de digitos
21     for (int i = 0; i < n; i++) {
22         count[(arr[i] / exp) % 10]++;
23     }
24
25     // Acumular posiciones
26     for (int i = 1; i < 10; i++) {
27         count[i] += count[i - 1];
28     }
29
30     // Construir el array ordenado
31     for (int i = n - 1; i >= 0; i--) {
32         output[count[(arr[i] / exp) % 10] - 1] = arr[i];
33         count[(arr[i] / exp) % 10]--;
34     }
35
36     // Copiar el array ordenado al original
37     for (int i = 0; i < n; i++) {
38         arr[i] = output[i];
39     }
40
41     free(output);
42 }
43
44 // Radix Sort
45 void radixSort(int arr[], int n) {
46     int max = getMax(arr, n);
47
48     // Ordenar para cada digito
49     for (int exp = 1; max / exp > 0; exp *= 10) {
50         countingSort(arr, n, exp);
51     }
52 }
53
54 // Imprimir un array
55 void printArray(int arr[], int n) {
56     for (int i = 0; i < n; i++) {
57         printf("%d ", arr[i]);
58     }
59     printf("\n");
```

```
60 }
61
62 int main() {
63     int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};
64     int n = sizeof(arr) / sizeof(arr[0]);
65
66     printf("Array original:\n");
67     printArray(arr, n);
68
69     radixSort(arr, n);
70
71     printf("Array ordenado:\n");
72     printArray(arr, n);
73
74     return 0;
75 }
```

Implementación en Python

```
1 def counting_sort(arr, exp):
2     n = len(arr)
3     output = [0] * n
4     count = [0] * 10
5
6     # Contar ocurrencias de dígitos
7     for num in arr:
8         index = (num // exp) % 10
9         count[index] += 1
10
11    # Acumular posiciones
12    for i in range(1, 10):
13        count[i] += count[i - 1]
14
15    # Construir el array ordenado
16    for num in reversed(arr):
17        index = (num // exp) % 10
18        output[count[index] - 1] = num
19        count[index] -= 1
20
21    return output
22
23 def radix_sort(arr):
24     if not arr:
25         return []
```

```

26
27     max_val = max(arr)
28     exp = 1
29     while max_val // exp > 0:
30         arr = counting_sort(arr, exp)
31         exp *= 10
32
33     return arr
34
35 # Ejemplo de uso
36 arr = [170, 45, 75, 90, 802, 24, 2, 66]
37 print("Array original:", arr)
38 sorted_arr = radix_sort(arr)
39 print("Array ordenado:", sorted_arr)

```

1.2.3. Bucket Sort

Bucket Sort es un algoritmo de ordenamiento no comparativo que distribuye los elementos en varias cubetas (*buckets*). Cada cubeta se ordena de manera individual, generalmente usando un algoritmo como *Insertion Sort*. Finalmente, las cubetas se concatenan para formar el array ordenado.

Análisis de Complejidad

- **Mejor caso:** $O(n + k)$, donde n es el número de elementos y k es el número de cubetas.
- **Peor caso:** $O(n^2)$ (cuando todos los elementos caen en una sola cubeta y deben ser ordenados).
- **Caso promedio:** $O(n + k)$.

Pseudocódigo

```

1 BucketSort(A, k)
2   Crear k cubetas vacias
3   para cada elemento x en A hacer
4       Calcular la cubeta correspondiente para x
5       Agregar x a la cubeta
6   fin para
7   para cada cubeta B hacer
8       Ordenar la cubeta B
9   fin para

```



```
10 Concatenar las cubetas en el array A
11 fin
```

Implementación en C

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 // Funcion para ordenar una cubeta usando Insertion Sort
5 void insertionSort(float arr[], int n) {
6     for (int i = 1; i < n; i++) {
7         float key = arr[i];
8         int j = i - 1;
9         while (j >= 0 && arr[j] > key) {
10             arr[j + 1] = arr[j];
11             j--;
12         }
13         arr[j + 1] = key;
14     }
15 }
16
17 // Bucket Sort
18 void bucketSort(float arr[], int n) {
19     // Crear cubetas
20     int numBuckets = 10;
21     float* buckets[numBuckets];
22     int bucketSizes[numBuckets];
23
24     for (int i = 0; i < numBuckets; i++) {
25         buckets[i] = (float*)malloc(n * sizeof(float));
26         bucketSizes[i] = 0;
27     }
28
29     // Distribuir elementos en cubetas
30     for (int i = 0; i < n; i++) {
31         int bucketIndex = (int)(arr[i] * numBuckets);
32         buckets[bucketIndex][bucketSizes[bucketIndex]++]
            = arr[i];
33     }
34
35     // Ordenar cubetas y concatenar resultados
36     int index = 0;
37     for (int i = 0; i < numBuckets; i++) {
38         insertionSort(buckets[i], bucketSizes[i]);
```

```
39         for (int j = 0; j < bucketSizes[i]; j++) {
40             arr[index++] = buckets[i][j];
41         }
42         free(buckets[i]);
43     }
44 }
45
46 // Imprimir un array
47 void printArray(float arr[], int n) {
48     for (int i = 0; i < n; i++) {
49         printf("%.2f", arr[i]);
50     }
51     printf("\n");
52 }
53
54 int main() {
55     float arr[] = {0.78, 0.17, 0.39, 0.26, 0.72, 0.94,
56                   0.21, 0.12, 0.23, 0.68};
57     int n = sizeof(arr) / sizeof(arr[0]);
58
59     printf("Array original:\n");
60     printArray(arr, n);
61
62     bucketSort(arr, n);
63
64     printf("Array ordenado:\n");
65     printArray(arr, n);
66
67     return 0;
68 }
```

Implementación en Python

```
1 def bucket_sort(arr):
2     n = len(arr)
3     if n <= 0:
4         return arr
5
6     # Crear cubetas vacías
7     num_buckets = 10
8     buckets = [[] for _ in range(num_buckets)]
9
10    # Distribuir elementos en cubetas
11    for num in arr:
```

```
12         index = int(num * num_buckets)
13         buckets[index].append(num)
14
15     # Ordenar cada cubeta y concatenar resultados
16     sorted_arr = []
17     for bucket in buckets:
18         sorted_arr.extend(sorted(bucket))
19
20     return sorted_arr
21
22 # Ejemplo de uso
23 arr = [0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12,
24        0.23, 0.68]
25 print("Array original:", arr)
26 sorted_arr = bucket_sort(arr)
27 print("Array ordenado:", sorted_arr)
```

1.3. Ordenamiento adaptativo

1.3.1. TimSort

1.4. Algoritmos notables

1.4.1. Pancake Sort

1.4.2. Gnome Sort

1.4.3. Cocktail Shaker Sort

1.4.4. Comb Sort

1.4.5. Bogo Sort

End Notes

Capítulo 2

Busqueda

End Notes

Capítulo 3

Estructuras de datos

End Notes