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

```c
#include <stdio.h>

void bubbleSort(int arr[], int n) {
    int temp;
    for (int i = 0; i < n-1; i++) {
        int swapped = 0;
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                // Intercambiar
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
                swapped = 1;
            }
        }
        if (!swapped) break;
    }
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Array ordenado: \n");
    printArray(arr, n);
    return 0;
}
```

Implementación en Python

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n-1):
        swapped = False
        for j in range(n-i-1):
            if arr[j] > arr[j+1]:
                # Intercambiar
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        if not swapped:
            break

# Ejemplo de uso
arr = [64, 34, 25, 12, 22, 11, 90]
bubble_sort(arr)
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

```c
#include <stdio.h>

void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        // Mover elementos mayores que la clave hacia la
           derecha
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr)/sizeof(arr[0]);
    insertionSort(arr, n);
    printf("Array ordenado: \n");
    printArray(arr, n);
    return 0;
}
```

Implementación en Python

```python
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        # Mover elementos mayores que la clave hacia la derecha
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

# Ejemplo de uso
arr = [12, 11, 13, 5, 6]
insertion_sort(arr)
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
    intercambiar A[i] y A[min_idx]
  fin para
fin
```

Implementación en C

```c
#include <stdio.h>

void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int min_idx = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        // Intercambiar el minimo con el primer elemento
           desordenado
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
    printf("Array ordenado: \n");
    printArray(arr, n);
    return 0;
}
```

Implementación en Python

```python
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        # Intercambiar el minimo con el primer elemento
            desordenado
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

# Ejemplo de uso
arr = [64, 25, 12, 22, 11]
selection_sort(arr)
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

```c
#include <stdio.h>

void shellSort(int arr[], int n) {
    for (int gap = n / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i++) {
            int temp = arr[i];
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp;
                j -= gap) {
                arr[j] = arr[j - gap];
            }
            arr[j] = temp;
        }
    }
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d␣", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {12, 34, 54, 2, 3};
    int n = sizeof(arr) / sizeof(arr[0]);
    shellSort(arr, n);
    printf("Array␣ordenado:␣\n");
    printArray(arr, n);
    return 0;
}
```

Implementación en Python

```python
def shell_sort(arr):
    n = len(arr)
    gap = n // 2
    while gap > 0:
        for i in range(gap, n):
            temp = arr[i]
            j = i
            while j >= gap and arr[j - gap] > temp:
                arr[j] = arr[j - gap]
                j -= gap
            arr[j] = temp
        gap //= 2

# Ejemplo de uso
arr = [12, 34, 54, 2, 3]
shell_sort(arr)
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

```c
#include <stdio.h>
#include <stdlib.h>

void merge(int arr[], 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] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

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

```
    return 0;
}
```

Implementación en Python

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]

        merge_sort(L)
        merge_sort(R)

        i = j = k = 0

        while i < len(L) and j < len(R):
            if L[i] <= R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1

# Ejemplo de uso
arr = [12, 11, 13, 5, 6, 7]
print("Array original:", arr)
merge_sort(arr)
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

```c
#include <stdio.h>

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

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

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d␣", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, n - 1);
    printf("Array␣ordenado:␣\n");
    printArray(arr, n);
    return 0;
}
```

Implementación en Python

```python
def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1

    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

def quick_sort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)

# Ejemplo de uso
arr = [10, 7, 8, 9, 1, 5]
quick_sort(arr, 0, len(arr) - 1)
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

```c
#include <stdio.h>

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```
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

```python
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left

    if right < n and arr[right] > arr[largest]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heap_sort(arr):
    n = len(arr)

    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    for i in range(n - 1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]
        heapify(arr, i, 0)

# Ejemplo de uso
arr = [12, 11, 13, 5, 6, 7]
heap_sort(arr)
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 tamanio MIN_RUN hacer
    usarInsertionSort(subarray)
  fin para
  tamanio ← MIN_RUN
  mientras tamanio < longitud(A) hacer
    para cada par de subarrays de tamanio "tamanio" en A hacer
      mezclar(subarray1, subarray2)
    fin para
    tamanio ← 2 * tamanio
  fin mientras
fin
```

Implementación en C

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MIN_RUN 32

void insertionSort(int arr[], int left, int right) {
    for (int i = left + 1; i <= right; i++) {
        int temp = arr[i];
        int j = i - 1;
        while (j >= left && arr[j] > temp) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = temp;
    }
}

void merge(int arr[], int left, int mid, int right) {
    int len1 = mid - left + 1, len2 = right - mid;
    int *leftArr = (int *)malloc(len1 * sizeof(int));
    int *rightArr = (int *)malloc(len2 * sizeof(int));

```

```
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
                : n - 1;
49          insertionSort(arr, i, end);
50      }
51
52      for (int size = MIN_RUN; size < n; size = 2 * size) {
53          for (int left = 0; left < n; left += 2 * size) {
54              int mid = (left + size - 1 < n) ? left + size
                    - 1 : n - 1;
55              int right = (left + 2 * size - 1 < n) ? left
                   + 2 * size - 1 : n - 1;
56
57              if (mid < right)
58                  merge(arr, left, mid, right);
59          }
60      }
61  }
62
63  void printArray(int arr[], int n) {
64      for (int i = 0; i < n; i++)
65          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]
```

```
            j += 1
        k += 1

    while i < len(left_part):
        arr[k] = left_part[i]
        i += 1
        k += 1

    while j < len(right_part):
        arr[k] = right_part[j]
        j += 1
        k += 1

def tim_sort(arr):
    n = len(arr)
    for start in range(0, n, MIN_RUN):
        end = min(start + MIN_RUN - 1, n - 1)
        insertion_sort(arr, start, end)

    size = MIN_RUN
    while size < n:
        for left in range(0, n, size * 2):
            mid = min(n - 1, left + size - 1)
            right = min((left + 2 * size - 1), n - 1)

            if mid < right:
                merge(arr, left, mid, right)

        size *= 2

# Ejemplo de uso
arr = [5, 21, 7, 23, 19, 4, 2, 8]
tim_sort(arr)
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

```c
#include <stdio.h>
#include <stdlib.h>

// Definicion del nodo del arbol binario
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Crear un nuevo nodo
struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(
        struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// Insertar un nodo en el arbol
struct Node* insert(struct Node* node, int data) {
    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 = []
```

```
    inorder(root, sorted_list)
    return sorted_list

# Ejemplo de uso
arr = [5, 3, 7, 1, 9, 4, 6]
print("Array original:", arr)
sorted_arr = tree_sort(arr)
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

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void countingSort(int arr[], int n, int range) {
    int* count = (int*)calloc(range, sizeof(int));
    int* output = (int*)malloc(n * sizeof(int));

    // Contar las ocurrencias
    for (int i = 0; i < n; i++) {
        count[arr[i]]++;
    }

    // Acumular las posiciones
    for (int i = 1; i < range; i++) {
        count[i] += count[i - 1];
    }

    // Construir el array ordenado
    for (int i = n - 1; i >= 0; i--) {
        output[count[arr[i]] - 1] = arr[i];
        count[arr[i]]--;
    }

    // Copiar el array ordenado al original
    for (int i = 0; i < n; i++) {
        arr[i] = output[i];
    }

    free(count);
    free(output);
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d␣", arr[i]);
    }
    printf("\n");
```

```
}

int main() {
    int arr[] = {4, 2, 2, 8, 3, 3, 1};
    int n = sizeof(arr) / sizeof(arr[0]);
    int range = 9; // Rango de los valores (0 a 8)

    printf("Array original:\n");
    printArray(arr, n);

    countingSort(arr, n, range);

    printf("Array ordenado:\n");
    printArray(arr, n);

    return 0;
}
```

Implementación en Python

```
def counting_sort(arr, max_val):
    n = len(arr)
    count = [0] * (max_val + 1)
    output = [0] * n

    # Contar las ocurrencias
    for num in arr:
        count[num] += 1

    # Acumular las posiciones
    for i in range(1, len(count)):
        count[i] += count[i - 1]

    # Construir el array ordenado
    for num in reversed(arr):
        output[count[num] - 1] = num
        count[num] -= 1

    return output

# Ejemplo de uso
arr = [4, 2, 2, 8, 3, 3, 1]
print("Array original:", arr)
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

```c
#include <stdio.h>
#include <stdlib.h>

// Obtener el digito mas significativo en la posicion exp
int getMax(int arr[], int n) {
    int max = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}

```

```
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");
```

```
}

int main() {
    int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Array original:\n");
    printArray(arr, n);

    radixSort(arr, n);

    printf("Array ordenado:\n");
    printArray(arr, n);

    return 0;
}
```

Implementación en Python

```
def counting_sort(arr, exp):
    n = len(arr)
    output = [0] * n
    count = [0] * 10

    # Contar ocurrencias de digitos
    for num in arr:
        index = (num // exp) % 10
        count[index] += 1

    # Acumular posiciones
    for i in range(1, 10):
        count[i] += count[i - 1]

    # Construir el array ordenado
    for num in reversed(arr):
        index = (num // exp) % 10
        output[count[index] - 1] = num
        count[index] -= 1

    return output

def radix_sort(arr):
    if not arr:
        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
```

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

Implementación en C

```c
#include <stdio.h>
#include <stdlib.h>

// Funcion para ordenar una cubeta usando Insertion Sort
void insertionSort(float arr[], int n) {
    for (int i = 1; i < n; i++) {
        float key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

// Bucket Sort
void bucketSort(float arr[], int n) {
    // Crear cubetas
    int numBuckets = 10;
    float* buckets[numBuckets];
    int bucketSizes[numBuckets];

    for (int i = 0; i < numBuckets; i++) {
        buckets[i] = (float*)malloc(n * sizeof(float));
        bucketSizes[i] = 0;
    }

    // Distribuir elementos en cubetas
    for (int i = 0; i < n; i++) {
        int bucketIndex = (int)(arr[i] * numBuckets);
        buckets[bucketIndex][bucketSizes[bucketIndex]++]
           = arr[i];
    }

    // Ordenar cubetas y concatenar resultados
    int index = 0;
    for (int i = 0; i < numBuckets; i++) {
        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,
            0.21, 0.12, 0.23, 0.68};
56      int n = sizeof(arr) / sizeof(arr[0]);
57
58      printf("Array original:\n");
59      printArray(arr, n);
60
61      bucketSort(arr, n);
62
63      printf("Array ordenado:\n");
64      printArray(arr, n);
65
66      return 0;
67  }
```

Implementación en Python

```
1  def bucket_sort(arr):
2      n = len(arr)
3      if n <= 0:
4          return arr
5
6      # Crear cubetas vacias
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,
        0.23, 0.68]
24  print("Array original:", arr)
25  sorted_arr = bucket_sort(arr)
26  print("Array ordenado:", sorted_arr)
```

1.3. Ordenamiento adaptativo

1.3.1. TimSort

TimSort es un algoritmo híbrido que combina *Insertion Sort* y *Merge Sort*. Aprovecha segmentos ya ordenados (*runs*) dentro del array para reducir el trabajo adicional. Esto lo hace adaptativo, ya que su rendimiento mejora si los datos están parcialmente ordenados.

Características Adaptativas

- Identifica automáticamente los segmentos ya ordenados (*runs*).
- Ordena los *runs* pequeños usando *Insertion Sort*.
- Fusiona (*merge*) los *runs* ordenados eficientemente para formar el resultado final.

Análisis de Complejidad

- **Mejor caso:** $O(n)$ (cuando los datos están completamente ordenados o casi ordenados).
- **Peor caso:** $O(n \log n)$ (cuando no hay ordenación preexistente).
- **Caso promedio:** $O(n \log n)$.

1.3.2. 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

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MIN_RUN 32

// Insertion Sort para runs pequenios
void insertionSort(int arr[], int left, int right) {
    for (int i = left + 1; i <= right; i++) {
        int temp = arr[i];
        int j = i - 1;
        while (j >= left && arr[j] > temp) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = temp;
    }
}

// Mezclar dos subarrays
void merge(int arr[], int left, int mid, int right) {
    int len1 = mid - left + 1, len2 = right - mid;
    int *leftArr = (int *)malloc(len1 * sizeof(int));
    int *rightArr = (int *)malloc(len2 * sizeof(int));

```

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

```c
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {5, 21, 7, 23, 19, 4, 2, 8};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Array original:\n");
    printArray(arr, n);

    timSort(arr, n);

    printf("Array ordenado:\n");
    printArray(arr, n);

    return 0;
}
```

Implementación en Python

```python
MIN_RUN = 32

def insertion_sort(arr, left, right):
    for i in range(left + 1, right + 1):
        key = arr[i]
        j = i - 1
        while j >= left and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

def merge(arr, left, mid, right):
    left_part = arr[left:mid + 1]
    right_part = arr[mid + 1:right + 1]

    i = j = 0
    k = left

    while i < len(left_part) and j < len(right_part):
        if left_part[i] <= right_part[j]:
            arr[k] = left_part[i]
```

```
            i += 1
        else:
            arr[k] = right_part[j]
            j += 1
        k += 1

    while i < len(left_part):
        arr[k] = left_part[i]
        i += 1
        k += 1

    while j < len(right_part):
        arr[k] = right_part[j]
        j += 1
        k += 1

def tim_sort(arr):
    n = len(arr)
    for start in range(0, n, MIN_RUN):
        end = min(start + MIN_RUN - 1, n - 1)
        insertion_sort(arr, start, end)

    size = MIN_RUN
    while size < n:
        for left in range(0, n, size * 2):
            mid = min(n - 1, left + size - 1)
            right = min((left + 2 * size - 1), n - 1)

            if mid < right:
                merge(arr, left, mid, right)

        size *= 2

# Ejemplo de uso
arr = [5, 21, 7, 23, 19, 4, 2, 8]
tim_sort(arr)
print("Array ordenado:", arr)
```

Conclusión

TimSort es un algoritmo adaptativo eficiente diseñado para aprovechar la estructura preexistente en los datos. Es ampliamente utilizado en Python y Java debido a su rendimiento superior en datos reales.

1.4. Algoritmos notables

1.4.1. Pancake Sort

El algoritmo Pancake Sort simula la ordenación de una pila de panqueques usando una espátula. Permite "voltear"los elementos desde la parte superior hasta una posición deseada. El objetivo es colocar el elemento más grande en la posición final, reduciendo el rango de ordenación en cada iteración.

Análisis de Complejidad

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

Pseudocódigo

```
PancakeSort(A)
  para i ← longitud(A) hasta 2 hacer
    maxIndex ← encontrarÍndiceDelMáximo(A, 1, i)
    si maxIndex != i entonces
      voltear(A, maxIndex)
      voltear(A, i)
    fin si
  fin para
fin

voltear(A, k)
  invertir los primeros k elementos de A
fin
```

Implementación en C

```c
#include <stdio.h>

// Voltear los primeros k elementos del array
void flip(int arr[], int k) {
    int start = 0;
    while (start < k) {
        int temp = arr[start];
        arr[start] = arr[k];
```

```
9           arr[k] = temp;
10          start++;
11          k--;
12      }
13  }
14
15  // Encontrar el indice del elemento maximo en un rango
16  int findMaxIndex(int arr[], int n) {
17      int maxIndex = 0;
18      for (int i = 1; i < n; i++) {
19          if (arr[i] > arr[maxIndex]) {
20              maxIndex = i;
21          }
22      }
23      return maxIndex;
24  }
25
26  // Pancake Sort
27  void pancakeSort(int arr[], int n) {
28      for (int size = n; size > 1; size--) {
29          int maxIndex = findMaxIndex(arr, size);
30
31          if (maxIndex != size - 1) {
32              // Llevar el maximo a la parte superior
33              flip(arr, maxIndex);
34
35              // Llevar el maximo a su posicion final
36              flip(arr, size - 1);
37          }
38      }
39  }
40
41  // Imprimir un array
42  void printArray(int arr[], int n) {
43      for (int i = 0; i < n; i++) {
44          printf("%d␣", arr[i]);
45      }
46      printf("\n");
47  }
48
49  int main() {
50      int arr[] = {3, 6, 2, 7, 4, 5};
51      int n = sizeof(arr) / sizeof(arr[0]);
52
53      printf("Array␣original:\n");
```

```
54      printArray(arr, n);
55
56      pancakeSort(arr, n);
57
58      printf("Array␣ordenado:\n");
59      printArray(arr, n);
60
61      return 0;
62  }
```

Implementación en Python

```
1   def flip(arr, k):
2       start = 0
3       while start < k:
4           arr[start], arr[k] = arr[k], arr[start]
5           start += 1
6           k -= 1
7
8   def find_max_index(arr, n):
9       max_index = 0
10      for i in range(1, n):
11          if arr[i] > arr[max_index]:
12              max_index = i
13      return max_index
14
15  def pancake_sort(arr):
16      n = len(arr)
17      for size in range(n, 1, -1):
18          max_index = find_max_index(arr, size)
19
20          if max_index != size - 1:
21              # Llevar el maximo a la parte superior
22              flip(arr, max_index)
23
24              # Llevar el maximo a su posicion final
25              flip(arr, size - 1)
26
27  # Ejemplo de uso
28  arr = [3, 6, 2, 7, 4, 5]
29  print("Array␣original:", arr)
30  pancake_sort(arr)
31  print("Array␣ordenado:", arr)
```

Conclusión

Pancake Sort es un algoritmo interesante por su enfoque único basado en volteos. Aunque no es práctico para la mayoría de las aplicaciones debido a su complejidad $O(n^2)$, tiene un valor educativo y es útil en problemas específicos.

1.4.2. Gnome Sort

El algoritmo Gnome Sort es un algoritmo simple de ordenamiento basado en el principio de "volver atrásçuando se encuentra un elemento fuera de orden. Es similar a *Insertion Sort*, pero en lugar de insertar directamente, realiza intercambios repetidos hasta que el orden sea correcto.

Análisis de Complejidad

- **Mejor caso:** $O(n)$ (cuando los datos están casi ordenados).
- **Peor caso:** $O(n^2)$.
- **Caso promedio:** $O(n^2)$.

Pseudocódigo

```
GnomeSort(A)
  i ← 0
  mientras i < longitud(A) hacer
    si i = 0 o A[i-1] <= A[i] entonces
      i ← i + 1
    si no
      intercambiar A[i] y A[i-1]
      i ← i - 1
    fin si
  fin mientras
fin
```

Implementación en C

```c
#include <stdio.h>

// Gnome Sort
void gnomeSort(int arr[], int n) {
```

```
    int i = 0;
    while (i < n) {
        if (i == 0 || arr[i - 1] <= arr[i]) {
            i++;
        } else {
            int temp = arr[i];
            arr[i] = arr[i - 1];
            arr[i - 1] = temp;
            i--;
        }
    }
}

// Imprimir un array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {34, 2, 10, -9, 7};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Array original:\n");
    printArray(arr, n);

    gnomeSort(arr, n);

    printf("Array ordenado:\n");
    printArray(arr, n);

    return 0;
}
```

Implementación en Python

```
def gnome_sort(arr):
    i = 0
    while i < len(arr):
        if i == 0 or arr[i - 1] <= arr[i]:
            i += 1
        else:
```

```
            arr[i], arr[i - 1] = arr[i - 1], arr[i]
            i -= 1

# Ejemplo de uso
arr = [34, 2, 10, -9, 7]
print("Array original:", arr)
gnome_sort(arr)
print("Array ordenado:", arr)
```

Conclusión

Gnome Sort es un algoritmo simple pero ineficiente para la mayoría de los casos. Es más útil como ejemplo educativo para entender los principios básicos de ordenamiento.

1.4.3. Cocktail Shaker Sort

El algoritmo Cocktail Shaker Sort es una variación del Bubble Sort que ordena en ambas direcciones en cada pasada. Esto permite que los elementos más pequeños y más grandes se muevan simultáneamente hacia sus posiciones correctas, mejorando el rendimiento en comparación con Bubble Sort en algunos casos.

Análisis de Complejidad

- **Mejor caso:** $O(n)$ (cuando los datos ya están ordenados).
- **Peor caso:** $O(n^2)$.
- **Caso promedio:** $O(n^2)$.

Pseudocódigo

```
CocktailShakerSort(A)
  inicio ← 0
  fin ← longitud(A) - 1
  cambiado ← verdadero
  mientras cambiado = verdadero hacer
    cambiado ← falso
    para i ← inicio hasta fin-1 hacer
      si A[i] > A[i+1] entonces
        intercambiar A[i] y A[i+1]
```

```
        cambiado ← verdadero
      fin si
    fin para
    si cambiado = falso entonces
      romper
    fin si
    fin ← fin - 1
    para i ← fin-1 hasta inicio hacer
      si A[i] > A[i+1] entonces
        intercambiar A[i] y A[i+1]
        cambiado ← verdadero
      fin si
    fin para
    inicio ← inicio + 1
  fin mientras
fin
```

Implementación en C

```c
#include <stdio.h>

// Cocktail Shaker Sort
void cocktailShakerSort(int arr[], int n) {
    int start = 0, end = n - 1;
    int swapped = 1;

    while (swapped) {
        swapped = 0;

        // Pasada de izquierda a derecha
        for (int i = start; i < end; i++) {
            if (arr[i] > arr[i + 1]) {
                int temp = arr[i];
                arr[i] = arr[i + 1];
                arr[i + 1] = temp;
                swapped = 1;
            }
        }

        if (!swapped) break;

        swapped = 0;
```

```
        end--;

        // Pasada de derecha a izquierda
        for (int i = end - 1; i >= start; i--) {
            if (arr[i] > arr[i + 1]) {
                int temp = arr[i];
                arr[i] = arr[i + 1];
                arr[i + 1] = temp;
                swapped = 1;
            }
        }

        start++;
    }
}

// Imprimir un array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {5, 3, 8, 6, 2, 7, 4, 1};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Array original:\n");
    printArray(arr, n);

    cocktailShakerSort(arr, n);

    printf("Array ordenado:\n");
    printArray(arr, n);

    return 0;
}
```

Implementación en Python

```
def cocktail_shaker_sort(arr):
    start = 0
    end = len(arr) - 1
```

```
    swapped = True

    while swapped:
        swapped = False

        # Pasada de izquierda a derecha
        for i in range(start, end):
            if arr[i] > arr[i + 1]:
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                swapped = True

        if not swapped:
            break

        swapped = False
        end -= 1

        # Pasada de derecha a izquierda
        for i in range(end - 1, start - 1, -1):
            if arr[i] > arr[i + 1]:
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                swapped = True

        start += 1

# Ejemplo de uso
arr = [5, 3, 8, 6, 2, 7, 4, 1]
print("Array original:", arr)
cocktail_shaker_sort(arr)
print("Array ordenado:", arr)
```

Conclusión

Cocktail Shaker Sort mejora el rendimiento de Bubble Sort al ordenar en ambas direcciones. Sin embargo, sigue siendo ineficiente en grandes conjuntos de datos debido a su complejidad cuadrática.

1.4.4. Comb Sort

Comb Sort es una mejora del Bubble Sort que reduce significativamente los intercambios necesarios al comparar elementos distantes. Esto se logra utilizando un *gap* (brecha) inicial grande, que se reduce en cada iteración según un factor de reducción (*shrink factor*), generalmente 1.3.

Análisis de Complejidad

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

Pseudocódigo

```
CombSort(A)
  n ← longitud(A)
  gap ← n
  shrink ← 1.3
  cambiado ← verdadero
  mientras gap > 1 o cambiado = verdadero hacer
    gap ← máximo(1, entero(gap / shrink))
    cambiado ← falso
    para i ← 0 hasta n-gap-1 hacer
      si A[i] > A[i+gap] entonces
        intercambiar A[i] y A[i+gap]
        cambiado ← verdadero
      fin si
    fin para
  fin mientras
fin
```

Implementación en C

```c
#include <stdio.h>

// Comb Sort
void combSort(int arr[], int n) {
    int gap = n;
    const float shrink = 1.3;
    int swapped = 1;

    while (gap > 1 || swapped) {
        gap = (int)(gap / shrink);
        if (gap < 1) gap = 1;

        swapped = 0;

```

```
        for (int i = 0; i + gap < n; i++) {
            if (arr[i] > arr[i + gap]) {
                int temp = arr[i];
                arr[i] = arr[i + gap];
                arr[i + gap] = temp;
                swapped = 1;
            }
        }
    }
}

// Imprimir un array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {8, 4, 1, 56, 3, -44, 23, -6, 28, 0};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Array original:\n");
    printArray(arr, n);

    combSort(arr, n);

    printf("Array ordenado:\n");
    printArray(arr, n);

    return 0;
}
```

Implementación en Python

```
def comb_sort(arr):
    n = len(arr)
    gap = n
    shrink = 1.3
    swapped = True

    while gap > 1 or swapped:
        gap = int(gap / shrink)
```

```
        if gap < 1:
            gap = 1

        swapped = False

        for i in range(n - gap):
            if arr[i] > arr[i + gap]:
                arr[i], arr[i + gap] = arr[i + gap], arr[
                    i]
                swapped = True

# Ejemplo de uso
arr = [8, 4, 1, 56, 3, -44, 23, -6, 28, 0]
print("Array original:", arr)
comb_sort(arr)
print("Array ordenado:", arr)
```

Conclusión

Comb Sort es un algoritmo eficiente para datos pequeños a medianos y mejora significativamente el rendimiento en comparación con Bubble Sort. Sin embargo, sigue siendo menos eficiente que algoritmos avanzados como Quick Sort o Merge Sort.

1.4.5. Bogo Sort

Bogo Sort, también conocido como Stupid Sort, es un algoritmo de ordenamiento extremadamente ineficiente. Genera permutaciones aleatorias del array hasta que encuentra una que esté ordenada. Debido a su naturaleza aleatoria, no es práctico para ningún caso de uso real, pero tiene valor educativo y humorístico.

Análisis de Complejidad

- **Mejor caso:** $O(n)$ (si el array ya está ordenado).
- **Peor caso:** Infinito (si la generación aleatoria nunca produce un array ordenado).
- **Caso promedio:** $O((n!)n)$.

Pseudocódigo

```
BogoSort(A)
  mientras noEstáOrdenado(A) hacer
    permutarAleatoriamente(A)
  fin mientras
fin

noEstáOrdenado(A)
  para i ← 1 hasta longitud(A)-1 hacer
    si A[i-1] > A[i] entonces
      retornar verdadero
    fin si
  fin para
  retornar falso
fin
```

1.4.6. Implementaciones

Implementación en C

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>

// Funcion para verificar si el array esta ordenado
bool isSorted(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        if (arr[i - 1] > arr[i]) {
            return false;
        }
    }
    return true;
}

// Funcion para permutar aleatoriamente el array
void shuffle(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        int r = rand() % n;
        int temp = arr[i];
        arr[i] = arr[r];
        arr[r] = temp;
```

```
    }
}

// Bogo Sort
void bogoSort(int arr[], int n) {
    while (!isSorted(arr, n)) {
        shuffle(arr, n);
    }
}

// Imprimir un array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    srand(time(0));

    int arr[] = {3, 2, 5, 1, 4};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Array original:\n");
    printArray(arr, n);

    bogoSort(arr, n);

    printf("Array ordenado:\n");
    printArray(arr, n);

    return 0;
}
```

Implementación en Python

```
import random

# Funcion para verificar si el array esta ordenado
def is_sorted(arr):
    return all(arr[i] <= arr[i + 1] for i in range(len(
        arr) - 1))

```

```
# Bogo Sort
def bogo_sort(arr):
    while not is_sorted(arr):
        random.shuffle(arr)

# Ejemplo de uso
arr = [3, 2, 5, 1, 4]
print("Array original:", arr)
bogo_sort(arr)
print("Array ordenado:", arr)
```

Conclusión

Bogo Sort es un algoritmo extremadamente ineficiente, cuyo uso práctico está limitado a fines educativos y humorísticos. Debido a su complejidad promedio de $O((n!)n)$, no es adecuado para ningún propósito realista.

End Notes

Capítulo 2

Busqueda

End Notes

Capítulo 3

Estructuras de datos

End Notes