Análisis De Algoritmos De Ordenamiento: ShakerSort, QuickSort, ShellSort, StoogeSort

Nombres y Códigos:

Isaac David Canabal Martinez - T00068229

Antonio José Mendoza Simarra - T00069670

Jorge Andrés Herrera Monsalve - T00068111

Aarón Dali López Fortich - T00068394

Complejidad del método ShakerSort

```
void shakerSort(int arr[], int n) {
  // Inicializar los índices para el barrido bidireccional
  int left = 0, right = n - 1;
  // Bucle principal para el ordenamiento
  while (left < right) {
    // Barrido izquierdo para llevar el elemento más grande al final
    for (int i = left; i < right; i++) {
      if (arr[i] > arr[i + 1]) {
        swap(arr[i], arr[i + 1]);
      }
    }
    right--; // Reducir el límite derecho

    // Barrido derecho para llevar el elemento más pequeño al inicio
    for (int i = right; i > left; i--) {
      if (arr[i] < arr[i - 1]) {
        swap(arr[i], arr[i - 1]);
      }
    }
    left++; // Ampliar el límite izquierdo
  }
}
```

Peor de los casos

O(N^2)

Análisis de costos para peor caso

Instrucción	Costo
int left = 0, right = n − 1;	2
while (left < right)	n/2 +1
1° For interno	n/2 * 4n - 2
int i = left	1
i < right;	n - 1 + 1
i ++;	n-1
if (arr[i] < arr[i - 1])	n - 1
swap(arr[i], arr[i - 1]);	n-1
2° For interno	n/2 * 4n - 2
int i = right;	1
i > left;	n - 1 + 1
i--;	n - 1
arr[i] < arr[i - 1])	n-1
swap(arr[i], arr[i - 1])	n-1
right--;	n/2
left++;	n/2

Complejidad del método ShakerSort

```
void shakerSort(int arr[], int n) {
  // Inicializar los índices para el barrido bidireccional
  int left = 0, right = n - 1;
  // Bucle principal para el ordenamiento
  while (left < right) {
    // Barrido izquierdo para llevar el elemento más grande al final
    for (int i = left; i < right; i++) {
      if (arr[i] > arr[i + 1]) {
        swap(arr[i], arr[i + 1]);
      }
    }
    right--; // Reducir el límite derecho

    // Barrido derecho para llevar el elemento más pequeño al inicio
    for (int i = right; i > left; i--) {
      if (arr[i] < arr[i - 1]) {
        swap(arr[i], arr[i - 1]);
      }
    }
    left++; // Ampliar el límite izquierdo
  }
}
```

Mejor de los casos

O(n)

Análisis de costos para mejor caso

Instruccion	Costo
int left = 0, right = n - 1;	2
`while (left < right)`	n/2 +1
1° For interno	
int i = left	1
i < right;	n - 1 + 1
i ++;	n-1
if (arr[i] < arr[i - 1])	n - 1
swap(arr[i], arr[i - 1]);	
2° For interno	
int i = right;	1
i > left;	n - 1 + 1
i--;	n - 1
arr[i] < arr[i - 1])	n - 1
swap(arr[i], arr[i - 1])	
right--;	n/2
left++;	n/2

Complejidad del método QuickSort

```
int partition(int array[], int left, int right) {
    int pivot = array[left];
    int pIndex = left;

    for (int i = left; i < right; i++) {
        if (array[i] <= pivot) {
            std::swap(array[i], array[pIndex]);
            pIndex++;
        }
    }

    std::swap(array[pIndex], array[right]);
    return pIndex;
}
```

Análisis de costos para mejor caso

Instrucción	Costo
int pivot = array[left]	1
int pIndex = left	1
int i = left	1
i < right	$n - 1 + 1$
i++	$n - 1$
if (array[i] <= pivot)	$n - 1$
std::swap(array[pIndex], array[right]);	1
return pIndex;	1

6+3(n-1)
O(n)

Suma o ecuación final

Complejidad del método QuickSort

```
int partition(int array[], int left, int right) {
    int pivot = array[left];
    int pIndex = left;

    for (int i = left; i < right; i++) {
        if (array[i] <= pivot) {
            std::swap(array[i], array[pIndex]);
            pIndex++;
        }
    }

    std::swap(array[pIndex], array[right]);
    return pIndex;
}
```

Análisis de costos para peor caso

Instrucción	Costo
int pivot = array[left]	1
int pIndex = left int i = left	1 1
i < right	$n - 1 + 1$
i++	$n - 1$
if (array[i] <= pivot)	$n - 1$
std::swap(array[i], a[pIndex])	$n - 1$
pIndex++	$n - 1$
std::swap(array[pIndex], array[right])	1
return pIndex	1

6+5(n-1)
O(n)

Suma o ecuación final

Complejidad del método QuickSort

Universidad Tecnológica de Bolívar

```
void quicksort(int array[], int left, int right) {
  if (left >= right) {
    return;
  }

  int pivotIndex = partition(array, left, right);
  quicksort(array, left, pivotIndex - 1);
  quicksort(array, pivotIndex + 1, right);
}
```

La function partition por si sola tiene una complejidad de O(n), por lo que ajustandolo en la formula

2 * log(n/2)
2 * log(n)
n * log(n)

Análisis de costos para mejor caso

Instrucción	Costo
if (left>= right)	1
partition(array, left, right)	n

Cada llamada recursive ordena la mitad del arreglo (izquierda y derecha).

El arreglo se ordena dividiéndolo a la mitad en cada llamada, lo que resulta en una complejidad de log(n/2).

n * log(n)

Suma o ecuación final

Complejidad del método QuickSort

```
void quicksort(int array[], int left, int right) {
  if (left ≥ right) {
    return;
  }

  int pivotIndex = partition(array, left, right);
  quicksort(array, left, pivotIndex - 1);
  quicksort(array, pivotIndex + 1, right);
}
```

Resultando en la ecuacion final de: O(n) * O(n – 1)

Terminando en una complejidad de O(n^2)

Análisis de costos para peor caso

Instrucción	Costo
if (left>= right)	1
partition(array, left, right)	n

Cuando se elige el pivote como el primer elemento del arreglo (índice 0), se crean dos arreglos: uno vacío (izquierda) y otro con todos los elementos menos el pivote (derecha).

Este proceso se repite sucesivamente, creando sub-arreglos, lo que resulta en que este proceso se ejecute n - 1 veces.

O(n^2)

Suma o ecuación final

Complejidad del método ShellSort

```c
void shellSort(int arr[], int n) {
  int gap = n / 2;

  while (gap > 0) {
    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;
    }

    gap /= 2;
  }
}
```

Análisis de costos para peor caso

Instrucción	Costo
int gap = n / 2;	1
while (gap > 0)	n/2 + 1
for (int i = gap; i < n; i++)	n - gap
int temp = arr[i];	n - gap
int j;	n - gap
for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {	n^2 - n
arr[j] = arr[j - gap];	n^2 - n
arr[j] = temp;	n^2 -n
gap /= 2;	n/2

El algoritmo tiene en cuenta para las ejecuciones el valor del gap o brecha. Este gap puede tomarse de diferentes formas, pero el caso base es n/2.

COMPLEJIDAD ESPACIAL O(1)

Complejidad del método ShellSort

Universidad Tecnológica de Bolívar

```
void shellSort(int arr[], int n) {
  int gap = n / 2;

  while (gap > 0) {
    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;
    }

    gap /= 2;
  }
}
```

Análisis de costos para mejor caso

Instrucción	Costo
int gap = n / 2;	1
while (gap > 0)	log n
for (int i = gap; i < n; i++)	n
int temp = arr[i];	n
int j;	n
for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {	1
arr[j] = arr[j - gap];	0
arr[j] = temp;	0
gap /= 2;	log n

Complejidad del método StoogeSort

```cpp
#include <iostream>
#include <vector>

void stoogeSort(std::vector<int>& arr, int low, int high) {
    if (low >= high) {
        return;
    }

    if (arr[low] > arr[high]) {
        std::swap(arr[low], arr[high]);
    }

    if (high - low + 1 > 2) {
        int t = (high - low + 1) / 3;

        stoogeSort(arr, low, high - t);
        stoogeSort(arr, low + t, high);
        stoogeSort(arr, low, high - t);
    }
}

int main() {
    std::vector<int> arr = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};

    std::cout << "Array before StoogeSort: ";
    for (int num : arr) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    stoogeSort(arr, 0, arr.size() - 1);

    std::cout << "Array after StoogeSort: ";
    for (int num : arr) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Análisis de costos para mejor y peor caso

Instrucción	Costo
if(low ≥ high	O(1)
return	O(1)
if(arr[low]>arr[high])	O(1)
std::swap(arr[low], arr[high])	O(1)
if(high-low+1>2)	O(1)
int t = (high - low + 1) / 3	O(1)
stoogeSort(arr, low, high - t)	T(2n/3)
stoogeSort(arr, low + t, high)	T(2n/3)
stoogeSort(arr, low, high - t)	T(2n/3)

RELACIÓN DE RECURRENCIA

T(n) = 3T(2n/3) + O(1)

O(n^(log 3 / log 1.5))

O(n^2.7095 ...)

COMPLEJIDAD ESPACIAL O(n)