Tarea de recuperación 2 – Investigación sobre algoritmos de ordenación

Alumno: Yamil Gazal Harika Vera – 5749243

ShellSort

Ordenamiento Shell es un algoritmo de ordenación altamente eficiente basado en la comparación. Se considera la generalización del algoritmo de ordenación por burbujas o un algoritmo de ordenación por inserción optimizado. Comienza ordenando elementos muy lejanos y reduce gradualmente la distancia basándose en una secuencia para ordenar todo el array.

La ordenación de shell divide la matriz en partes más pequeñas según el valor del intervalo y ejecuta la ordenación por inserción en esas partes.

Gradualmente, el valor del intervalo disminuye y el tamaño de las piezas divididas aumenta. Como las piezas se clasifican individualmente de antemano, fusionarlas requiere menos pasos que el proceso.

la secuencia que ShellSort utiliza es: N/2 , N/4 , ..., 1

```
#include <stdio.h>


// Ordena el arreglo utilizando el método de Shell Sort
Void shellSort(int array[], int n) {
  // Reorganizar elementos en intervalos de n/2, n/4, n/8, ...
  For (int interval = n / 2; interval > 0; interval /= 2) {
    For (int i = interval; i < n; i += 1) {
      Int temp = array[i];
      Int j;
      For (j = i; j >= interval && array[j – interval] > temp; j -= interval) {
        Array[j] = array[j – interval];
      }
      Array[j] = temp;
    }
  }
}
```

```
// Imprimir un arreglo
Void printArray(int array[], int size) {
  For (int i = 0; i < size; ++i) {
    Printf("%d  ", array[i]);
  }
  Printf("\n");
}

// Código principal
Int main() {
  Int data[] = {9, 8, 3, 7, 5, 6, 4, 1};
  Int size = sizeof(data) / sizeof(data[0]);
  shellSort(data, size);
  printf("Arreglo ordenado: \n");
  printArray(data, size);
}
```

El codigo de ShellSort implementado con memoria dinamica:

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

Void shellSort(int array[], int n) {
  For (int interval = n / 2; interval > 0; interval /= 2) {
    For (int i = interval; i < n; i += 1) {
      Int temp = array[i];
      Int j;
      For (j = i; j >= interval && array[j – interval] > temp; j -= interval) {
        Array[j] = array[j – interval];
      }
```

```
      Array[j] = temp;
    }
  }
}


Void printArray(int array[], int size) {
  For (int i = 0; i < size; ++i) {
    Printf("%d  ", array[i]);
  }
  Printf("\n");
}

Int main() {
  Int n;

  Printf("Ingrese el tamaño del arreglo: ");
  Scanf("%d", &n);

  Int *data = (int *)malloc(n * sizeof(int));
  If (data == NULL) {
    Printf("No se pudo asignar memoria.\n");
    Return 1;
  }
  Printf("Ingrese los elementos del arreglo:\n");
  For (int i = 0; i < n; i++) {
    Printf("Elemento %d: ", i + 1);
    Scanf("%d", &data[i]);
  }
```

```
  shellSort(data, n);


  Printf(“Arreglo ordenado:\n”);
  printArray(data, n);
  Free(data);
  Return 0;
}
```

Se usa malloc para asignar memoria para el arreglo data en tiempo de ejecución.

Big O en ShellSort

La complejidad temporal del Shellsort varía según la secuencia de gaps utilizada y puede oscilar entre O(N^1.5) y O(Nlog2N) en el peor de los casos, aunque para secuencias específicas puede acercarse a O(N log N) en casos promedio.

HeapSort

HeapSort es un algoritmo de ordenación eficiente que utiliza una estructura de datos llamada heap o montículo para ordenar una lista de elementos. El monticulo es una estructura de datos especial basada en un árbol binario.

La ordenación en pila funciona de forma muy similar a la ordenación por selección. Selecciona el elemento máximo del array usando max-heap y lo coloca en su posición al final del array. Hace uso de un procedimiento llamado heapify() para construir el monticulo.

Todos los nodos padres son más pequeños/más grandes que sus dos nodos hijos. Si son más pequeños, el montón se llama min-heap, y si son más grandes, entonces el montón se llama max-heap. Para un índice dado i, el padre está dado por (i-1)/2, el hijo izquierdo está dado por (2*i+1) y el hijo derecho está dado por (2*i+2).

```
#include <stdio.h>


// Función para intercambiar la posición de dos elementos
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}


void heapify(int arr[], int n, int i) {
    // Encontrar el mayor entre la raíz, el hijo izquierdo y el hijo derecho
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;


    if (left < n && arr[left] > arr[largest])
        largest = left;


    if (right < n && arr[right] > arr[largest])
```

```
        largest = right;

    // Intercambiar y continuar heapificando si la raíz no es la mayor
    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        heapify(arr, n, largest);
    }
}

// Función principal para realizar el ordenamiento por heapsort
void heapSort(int arr[], int n) {
    // Construir el max-heap
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Ordenar usando heap
    for (int i = n - 1; i >= 0; i--) {
        swap(&arr[0], &arr[i]);

        // Aplicar heapify a la raíz para mantener el max-heap
        heapify(arr, i, 0);
    }
}

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

```
// Código del driver
int main() {
    int arr[] = {1, 12, 9, 5, 6, 10};
    int n = sizeof(arr) / sizeof(arr[0]);

    heapSort(arr, n);

    printf("El arreglo ordenado es \n");
    printArray(arr, n);
    return 0;
}
```

El codigo de ShellSort implementado con memoria dinamica:

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

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

void heapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest])
        largest = left;
```

```
    if (right < n && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    for (int i = n - 1; i >= 0; i--) {
        swap(&arr[0], &arr[i]);

        heapify(arr, i, 0);
    }
}

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

int main() {
    int n;
    printf("Ingrese el número de elementos: ");
    scanf("%d", &n);
```

```
    int *arr = (int*)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Error de asignación de memoria\n");
        return 1;
    }

    printf("Ingrese los elementos del arreglo:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    heapSort(arr, n);

    printf("El arreglo ordenado es:\n");
    printArray(arr, n);
    free(arr);
    return 0;
}
```

malloc asigna un bloque de memoria de tamaño especificado y devuelve un puntero a la memoria asignada.

Big O en HeapSort

Para construir el Max-Heap se necesita recorrer todos los nodos no hoja, aplicando la función de heapify. Dado que el número de nodos no hoja es aproximadamente la mitad del total de nodos y cada llamada a heapify toma O(logn), la complejidad de esta fase es O(n)

Durante la fase de ordenación, se realizan n−1 intercambios y cada intercambio va seguido por una llamada a la función heapify, que toma O(log n) Por lo tanto, la complejidad de esta fase es O(n log n).

MergeSort

El MergeSort es uno de los algoritmos de ordenación más populares y eficientes. Se basa en el principio del algoritmo divide y vencerás. Funciona dividiendo el array en dos mitades repetidamente hasta que obtenemos el array dividido en elementos individuales. Un elemento individual es un array ordenado en sí mismo. El ordenamiento por mezcla combina repetidamente estas pequeñas matrices ordenadas para producir submatrices ordenadas más grandes hasta que obtenemos un array final ordenado.

El MergeSort comienza desde la parte superior con el array completo y procede hacia abajo a los elementos individuales con recursión.

```
#include <stdio.h>


// Fusionar dos subarreglos L y M en arr
void merge(int arr[], int p, int q, int r) {

  // Crear L ← A[p..q] y M ← A[q+1..r]
  int n1 = q - p + 1;
  int n2 = r - q;

  int L[n1], M[n2];

  for (int i = 0; i < n1; i++)
    L[i] = arr[p + i];
  for (int j = 0; j < n2; j++)
    M[j] = arr[q + 1 + j];

  // Mantener el índice actual de los subarreglos y el arreglo principal
  int i = 0, j = 0, k = p;

  // Hasta que lleguemos al final de L o M, escoger el mayor entre
  // los elementos L y M y colocarlos en la posición correcta en A[p..r]
  while (i < n1 && j < n2) {
```

```
    if (L[i] <= M[j]) {
      arr[k] = L[i];
      i++;
    } else {
      arr[k] = M[j];
      j++;
    }
    k++;
  }

  // Cuando se acaben los elementos en L o M,
  // tomar los elementos restantes y ponerlos en A[p..r]
  while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
  }

  while (j < n2) {
    arr[k] = M[j];
    j++;
    k++;
  }
}

// Dividir el arreglo en dos subarreglos, ordenarlos y fusionarlos
void mergeSort(int arr[], int l, int r) {
  if (l < r) {

    // m es el punto donde el arreglo se divide en dos subarreglos
```

```
    int m = l + (r - l) / 2;

    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);

    // Fusionar los subarreglos ordenados
    merge(arr, l, m, r);
  }
}

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

// Programa principal
int main() {
  int arr[] = {6, 5, 12, 10, 9, 1};
  int size = sizeof(arr) / sizeof(arr[0]);

  mergeSort(arr, 0, size - 1);

  printf("Arreglo ordenado: \n");
  printArray(arr, size);
}
```

El codigo de MergeSort implementado con memoria dinamica:

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

void merge(int arr[], int p, int q, int r) {
    int n1 = q - p + 1;
    int n2 = r - q;

    int *L = (int *)malloc(n1 * sizeof(int));
    int *M = (int *)malloc(n2 * sizeof(int));

    if (L == NULL || M == NULL) {
        fprintf(stderr, "Error de asignación de memoria\n");
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < n1; i++)
        L[i] = arr[p + i];
    for (int j = 0; j < n2; j++)
        M[j] = arr[q + 1 + j];
    int i = 0, j = 0, k = p;

    while (i < n1 && j < n2) {
        if (L[i] <= M[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = M[j];
```

```
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = M[j];
        j++;
        k++;
    }
    free(L);
    free(M);
}

void mergeSort(int arr[], int l, int r) {
    if (l < r) {

        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}
```

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

int main() {
    int n;
    printf("Ingrese el tamaño del arreglo: ");
    scanf("%d", &n);
    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL) {
        fprintf(stderr, "Error de asignación de memoria\n");
        return 1;
    }

    printf("Ingrese los elementos del arreglo:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    mergeSort(arr, 0, n - 1);

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

    free(arr);

    return 0;
```

}

En la función merge, se utilizan las funciones malloc para asignar memoria dinámica a los subarreglos L y M.

Big O en MergeSort

Merge Sort tiene una complejidad temporal de O(n log n) en todos los casos (mejor, promedio y peor caso).

El arreglo se divide recursivamente en dos mitades hasta que cada subarreglo tiene un solo elemento. Esta fase requiere log n divisiones, ya que cada división corta el problema a la mitad.

La fusión de los subarreglos requiere un tiempo proporcional al número de elementos en el arreglo. Dado que hay n elementos y cada paso de fusión se realiza en tiempo lineal, la fase de fusión tiene una complejidad de O(n).

https://www.delftstack.com/es/tutorial/algorithm/shell-sort/

https://www.guru99.com/es/shell-sort-algorithm.html

Codigo de ShellSort traducido de: https://www.programiz.com/dsa/shell-sort

https://www.programiz.com/dsa/shell-sort

https://en.wikipedia.org/wiki/Shellsort

Traducido de: https://www.sanfoundry.com/c-program-heap-sort-algorithm/#google_vignette

https://www.delftstack.com/es/tutorial/algorithm/heap-sort/

Codigo de HeapSort traducido de: https://www.programiz.com/dsa/heap-sort

Traducido de: https://brilliant.org/wiki/heap-sort/

https://www.delftstack.com/es/tutorial/algorithm/merge-sort/

Traducido de: https://www.geeksforgeeks.org/merge-sort/

Codigo de MergeSort traducido de: https://www.programiz.com/dsa/merge-sort