Clase 8

Fecha	@May 7, 2025 3:04 PM
Descripcion	Ordenamiento y Busqueda

Ordenamiento de Burbuja (Bubble Sort)

Concepto Básico

El ordenamiento de burbuja es un algoritmo simple para ordenar elementos en un arreglo o lista. Funciona comparando pares de elementos adyacentes y **intercambiándolos** si están en el orden incorrecto.

Características Principales

- **Complejidad temporal**:
  - Peor caso: O(n^2)
  - Mejor caso: O(n) (cuando el arreglo ya está ordenado)
  - Caso promedio: O(n^2)
- **Estable**: Mantiene el orden relativo de elementos iguales
- **In-place**: Solo requiere una cantidad constante de espacio adicional

Implementación en C

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        // Últimos i elementos ya están en su lugar
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                // Intercambiar arr[j] y arr[j+1]
                int temp = arr[j];
```

```
            arr[j] = arr[j+1];
            arr[j+1] = temp;
        }
    }
  }
}
```

Optimizaciones

1. **Bandera para detectar ordenación temprana**:

```
void optimizedBubbleSort(int arr[], int n) {
    int swapped;
    for (int i = 0; i < n-1; i++) {
        swapped = 0;
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
                swapped = 1;
            }
        }
        // Si no hubo intercambios, el arreglo está ordenado
        if (swapped == 0) break;
    }
}
```

1. **Registro de última posición intercambiada**:

```
void improvedBubbleSort(int arr[], int n) {
    int newn;
    while (n > 1) {
        newn = 0;
        for (int i = 1; i < n; i++) {
            if (arr[i-1] > arr[i]) {
                int temp = arr[i-1];
                arr[i-1] = arr[i];
```

```
            arr[i] = temp;
            newn = i;
          }
        }
        n = newn;
    }
}
```

Ventajas y Desventajas

Ventajas:

- Simple de entender e implementar
- No requiere espacio adicional significativo
- Eficiente para conjuntos de datos pequeños o casi ordenados

Desventajas:

- Ineficiente para conjuntos de datos grandes
- Rendimiento pobre comparado con algoritmos más avanzados como QuickSort o MergeSort

Ejemplo de Aplicación

```
#include <stdio.h>

void bubbleSort(int arr[], int n);

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);

    printf("Arreglo original:\\n");
    for (int i=0; i < n; i++)
        printf("%d ", arr[i]);

    bubbleSort(arr, n);
```

```
    printf("\\nArreglo ordenado:\\n");
    for (int i=0; i < n; i++)
        printf("%d ", arr[i]);

    return 0;
}
```

Este algoritmo es útil principalmente con fines educativos o cuando se necesita una implementación rápida y sencilla para conjuntos de datos pequeños.

Ordenamiento por Selección (Selection Sort)

Concepto Fundamental

El ordenamiento por selección es un algoritmo de ordenamiento **in-place** que divide el arreglo en dos partes: una sublista ordenada y otra sublista desordenada. Funciona encontrando repetidamente el elemento mínimo (o máximo) de la parte desordenada y colocándolo al final de la parte ordenada.

Características Clave

- **Complejidad temporal**: O(n^2) en todos los casos (peor, mejor y promedio)
- **In-place**: Solo requiere O(1) espacio adicional
- **No estable**: Puede cambiar el orden relativo de elementos iguales
- **Pocos intercambios**: Realiza exactamente n-1 intercambios (ventaja cuando los intercambios son costosos)

Implementación Básica en C

```
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        // Encontrar el mínimo elemento en el subarreglo desordenado
        int min_idx = i;
        for (int j = i+1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
```

```
            }
        }

        // Intercambiar el mínimo encontrado con el primer elemento
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

Ventajas y Desventajas

Ventajas:

- Simple de implementar
- Rendimiento predecible (siempre $O(n^2)$)
- Eficiente en términos de número de intercambios (solo n-1 intercambios)

Desventajas:

- Ineficiente para listas grandes
- Peor rendimiento que otros algoritmos $O(n^2)$ como el insertion sort en la mayoría de casos
- No estable (no mantiene el orden original de elementos iguales)

Comparación con Otros Algoritmos

Característica	Selection Sort	Bubble Sort	Insertion Sort
Complejidad	$O(n^2)$	$O(n^2)$	$O(n^2)$
Intercambios	O(n)	$O(n^2)$	$O(n^2)$
Estabilidad	No	Sí	Sí
Rendimiento práctico	Pobre	Pobre	Mejor para pequeños/con casi ordenados

Ejemplo Completo en C

```
#include <stdio.h>

void selectionSort(int arr[], int n);
void printArray(int arr[], int size);

int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);

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

    selectionSort(arr, n);

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

    return 0;
}

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

Optimizaciones Posibles

1. **Ordenamiento simultáneo de mínimo y máximo**:

```
void dualSelectionSort(int arr[], int n) {
    for (int i = 0, j = n-1; i < j; i++, j--) {
        int min = i, max = i;

        for (int k = i; k <= j; k++) {
            if (arr[k] < arr[min]) min = k;
            if (arr[k] > arr[max]) max = k;
```

```
        }

        // Intercambiar mínimo
        int temp = arr[i];
        arr[i] = arr[min];
        arr[min] = temp;

        // Manejo especial si el máximo estaba en i
        if (max == i) max = min;

        // Intercambiar máximo
        temp = arr[j];
        arr[j] = arr[max];
        arr[max] = temp;
    }
}
```

1. **Versión recursiva**:

```
void selectionSortRecursive(int arr[], int n, int index) {
    if (index >= n-1) return;

    int min_idx = index;
    for (int i = index+1; i < n; i++) {
        if (arr[i] < arr[min_idx]) {
            min_idx = i;
        }
    }

    int temp = arr[index];
    arr[index] = arr[min_idx];
    arr[min_idx] = temp;

    selectionSortRecursive(arr, n, index+1);
}
```

Casos de Uso Recomendados

El selection sort es adecuado cuando:

- El espacio de memoria es limitado
- Los intercambios son operaciones costosas
- Se está trabajando con conjuntos de datos pequeños
- La simplicidad de implementación es más importante que la eficiencia