Schema Array

Array = Insieme di elementi

```
int intArray[];    //dichiarazione array
intArray = new int[20];  // allocazione memoria
```

Possiamo descrivere delle `classi` per definire contesti di invocazione. In questo caso, definiamo delle variabili.

```
public class Array{
        int dimensione;
        int elemento;
}
```

Esistono alcuni metodi per `popolare, attraversare e inserire` un item in un array.

```
public class Array {
    int dimensione;
    int[] elementi;

    // Costruttore per inizializzare la dimensione e l'array
    public Array(int dimensione) {
        this.dimensione = dimensione;
        this.elementi = new int[dimensione];
    }

    // Metodo per popolare l'array con valori iniziali
    public void popolaArray() {
```

```
        for (int i = 0; i < dimensione; i++) {

            elementi[i] = i * 2;
        }
    }

    // Metodo per attraversare e stampare gli elementi dell'array
    public void attraversaArray() {
        System.out.print("Elementi dell'array: ");
        for (int elemento=0; elemento < dimensione; elemento ++) {
            System.out.print(elemento + " ");
        }
        System.out.println();
    }

    // Metodo per inserire un nuovo elemento in un'posizione
specifica dell'array
    public void inserisciElemento(int posizione, int nuovoElemento)
{
        if (posizione >= 0 && posizione < dimensione) {
            elementi[posizione] = nuovoElemento;
        } else {
            System.out.println("Posizione non valida.");
        }
    }
```

Similmente, abbiamo metodi per `eliminare e svuotare` un array:

```
    // Metodo per eliminare un elemento dall'array
    public void eliminaElemento(int posizione) {
        if (posizione >= 0 && posizione < dimensione) {
```

```
            // Sposta gli elementi successivi all'elemento da
eliminare
            for (int i = posizione; i < dimensione - 1; i++) {
                elementi[i] = elementi[i + 1];
            }
            // Riduci la dimensione dell'array
            dimensione--;
        } else {
            System.out.println("Posizione non valida per
l'eliminazione.");
        }
    }

    // Metodo per svuotare completamente l'array
    public void svuotaArray() {
        dimensione = 0;
        elementi = new int[0];
    }
```

Possiamo inoltre descrivere:

- algoritmo di ordinamento per inserimento (**insertion sort**)

Pro

1. **Semplicità:** L'Insertion Sort è uno degli algoritmi di ordinamento più semplici da implementare e capire. La sua implementazione richiede solo poche linee di codice.
2. **Adatto per piccoli dataset:** Funziona bene per dataset di piccole dimensioni o già parzialmente ordinati. In queste situazioni, l'Insertion Sort può essere più efficiente rispetto ad altri algoritmi più complessi.

3. **Efficienza con dati quasi ordinati:** Se l'array è già quasi ordinato, l'Insertion Sort può essere molto efficiente e richiedere meno operazioni rispetto ad altri algoritmi.
4. **In-place e stabile:** L'Insertion Sort ordina l'array "in-place", ovvero senza richiedere memoria aggiuntiva. Inoltre, è un algoritmo stabile, il che significa che preserva l'ordine relativo degli elementi con chiave identica.

Contro

1. **Inefficienza su grandi dataset:** L'Insertion Sort diventa inefficiente su dataset di grandi dimensioni, poiché la sua complessità è di O(n^2), dove n è la dimensione dell'array.
2. **Prestazioni non ottimali:** Anche se è efficiente per dataset piccoli o parzialmente ordinati, Insertion Sort non è l'opzione più veloce per dataset casuali o grandi.
3. **Sensibile all'ordine iniziale:** Le prestazioni dell'Insertion Sort sono sensibili all'ordine iniziale degli elementi. Se gli elementi sono disposti in ordine inverso, richiederà il massimo numero di confronti e scambi.
4. **Non adatto per dati complessi:** Se si tratta di strutture dati complesse anziché dati primitivi, l'Insertion Sort potrebbe richiedere una complessa implementazione della comparazione e dello spostamento degli elementi.

```
public class InsertionSort {

    public static void insertionSort(int[] array) {
        int n = array.length;

        for (int i = 1; i < n; ++i) {
            int key = array[i];
            int j = i - 1;
```

```
            while (j >= 0 && array[j] > key) {
                array[j + 1] = array[j];
                j = j - 1;
            }
            array[j + 1] = key;
        }
    }

    public static void main(String[] args) {
        int[] array = {12, 11, 13, 5, 6};
        insertionSort(array);
        System.out.println("Array ordinato usando Insertion
Sort:");
        for (int num : array) {
            System.out.print(num + " ");
        }
    }
}
```

- algoritmo **bubble sort**

Pro

1. **Semplicità:** Come l'Insertion Sort, il Bubble Sort è molto semplice da implementare. Richiede poche linee di codice ed è facile da capire.
2. **In-place:** Bubble Sort ordina l'array "in-place", ovvero senza richiedere memoria aggiuntiva. Questo può essere vantaggioso in termini di spazio.
3. **Stabile:** L'algoritmo Bubble Sort è stabile, il che significa che preserva l'ordine relativo degli elementi con chiave identica.

Contro

1. **Inefficienza su grandi dataset:** Come l'Insertion Sort, il Bubble Sort diventa inefficiente su dataset di grandi dimensioni. La sua complessità è di O(n^2), dove n è la dimensione dell'array (perché fa due cicli)
2. **Prestazioni povere:** Anche su dataset di dimensioni moderate, il Bubble Sort può avere prestazioni inferiori rispetto ad altri algoritmi di ordinamento più efficienti.
3. **Sensibile all'ordine iniziale:** Le prestazioni del Bubble Sort sono sensibili all'ordine iniziale degli elementi. Se gli elementi sono disposti in ordine inverso, richiederà il massimo numero di confronti e scambi.
4. **Non adatto per dataset parzialmente ordinati:** Anche se Insertion Sort è efficiente per dataset parzialmente ordinati, il Bubble Sort non offre miglioramenti significativi in queste situazioni.
5. **Algoritmo inefficiente:** Il Bubble Sort può richiedere molte iterazioni attraverso l'array anche se l'array è già ordinato. Altri algoritmi più efficienti possono evitare questo tipo di lavoro superfluo

```
public class BubbleSort {

    public static void bubbleSort(int[] array) {
        int n = array.length;

        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                if (array[j] > array[j + 1]) {
                    int temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
            }
        }
```

```
    }

    public static void main(String[] args) {
        int[] array = {64, 34, 25, 12, 22, 11, 90};
        bubbleSort(array);
        System.out.println("Array ordinato usando Bubble Sort:");
        for (int num : array) {
            System.out.print(num + " ");
        }
    }
}
```

- algoritmo **selection sort**

Pro

1. **Semplicità:** L'implementazione del Selection Sort è abbastanza semplice, e il suo concetto di selezionare ripetutamente l'elemento più piccolo è facile da capire.
2. **In-place:** Selection Sort ordina l'array "in-place", ovvero senza richiedere memoria aggiuntiva. Questo può essere vantaggioso in termini di spazio.
3. **Minimo numero di scambi:** Selection Sort effettua un numero minimo di scambi, poiché seleziona direttamente la posizione appropriata per ogni elemento.

Contro

1. **Inefficienza su grandi dataset:** La complessità dell'algoritmo è di O(n^2), dove n è la dimensione dell'array. Pertanto, diventa inefficiente su dataset di grandi dimensioni (fa due cicli quindi quadratico)

2. **Sensibile all'ordine iniziale:** Le prestazioni di Selection Sort sono sensibili all'ordine iniziale degli elementi. Se gli elementi sono disposti in ordine inverso, richiederà il massimo numero di confronti e scambi.
3. **Non adatto per dataset parzialmente ordinati:** Anche se Selection Sort è efficiente nel ridurre il numero di scambi, non offre miglioramenti significativi per dataset già parzialmente ordinati.
4. **Instabile:** Selection Sort non è un algoritmo stabile. Potrebbe cambiare l'ordine relativo degli elementi con chiavi identiche.
5. **Algoritmo inefficiente:** Anche se è più efficiente di Bubble Sort, Selection Sort può richiedere un numero significativo di confronti su dataset grandi e casuali.

```
public class SelectionSort {

    public static void selectionSort(int[] array) {
        int n = array.length;

        for (int i = 0; i < n - 1; i++) {
            int minIndex = i;
            for (int j = i + 1; j < n; j++) {
                if (array[j] < array[minIndex]) {
                    minIndex = j;
                }
            }

            int temp = array[minIndex];
            array[minIndex] = array[i];
            array[i] = temp;
        }
    }

    public static void main(String[] args) {
```

```
        int[] array = {64, 25, 12, 22, 11};
        selectionSort(array);
        System.out.println("Array ordinato usando Selection
Sort:");
        for (int num : array) {
            System.out.print(num + " ");
        }
    }
}
```

- algoritmo **shell sort**

Pro

1. **Miglioramento rispetto a algoritmi quadratici:** Shell Sort migliora le prestazioni rispetto agli algoritmi di ordinamento quadratici (a due dimensioni) come Bubble Sort, Insertion Sort e Selection Sort, specialmente su dataset di medie dimensioni.
2. **In-place:** Shell Sort ordina l'array "in-place", ovvero senza richiedere memoria aggiuntiva. Questo può essere vantaggioso in termini di spazio.
3. **Adatto a dataset parzialmente ordinati:** Shell Sort può funzionare bene su dataset parzialmente ordinati, poiché riduce la distanza tra gli elementi da confrontare.
4. **Stabile:** A seconda dell'implementazione, Shell Sort può essere reso stabile.

Contro

1. **Complessità variabile:** La complessità temporale di Shell Sort dipende dalla sequenza dei gap utilizzati. Trovare la sequenza ideale può richiedere tempo.

2. **Complessità di analisi:** L'analisi matematica della complessità di Shell Sort è complessa a causa della variabilità dei gap.
3. **Non il più veloce:** Anche se migliora rispetto agli algoritmi quadratici, Shell Sort di solito è meno veloce rispetto a algoritmi più avanzati come Quicksort o Merge Sort.
4. **Sensibile alla scelta della sequenza di gap:** L'efficienza di Shell Sort è influenzata dalla scelta della sequenza di gap. Una scelta non ottimale può portare a prestazioni meno efficienti.

```
public class ShellSort {

    public static void shellSort(int[] array) {
        int n = array.length;

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

    public static void main(String[] args) {
        int[] array = {12, 34, 54, 2, 3};
        shellSort(array);
        System.out.println("Array ordinato usando Shell Sort:");
        for (int num : array) {
```

```
            System.out.print(num + " ");
        }
    }
}
```