Algoritmi di Ordinamento in C++

Gli algoritmi di ordinamento sono fondamentali in informatica. Servono per organizzare una serie di elementi in un ordine specifico, solitamente crescente o decrescente. Vediamo tre algoritmi di base:

1. Bubble Sort

Teoria

Il Bubble Sort confronta coppie adiacenti di elementi e li scambia se sono nell'ordine sbagliato. Questo processo viene ripetuto fino a quando non sono necessari ulteriori scambi.

- Complessità temporale: O(n^2)
- Complessità spaziale: O(1)
- Stabile: Sì

Implementazione

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++)
        for (int j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1])
                swap(arr[j], arr[j+1]);
}
```

2. Selection Sort

Teoria

Il Selection Sort divide l'input in una parte ordinata e una non ordinata. Ad ogni iterazione, trova l'elemento minimo nella parte non ordinata e lo sposta nella parte ordinata.

- Complessità temporale: O(n^2)
- Complessità spaziale: O(1)
- Stabile: No

Implementazione

```
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;
        swap(arr[min_idx], arr[i]);
    }
}
```

3. Insertion Sort

Teoria

L'Insertion Sort costruisce l'array finale un elemento alla volta. Prende un elemento dalla parte non ordinata e lo inserisce nella posizione corretta nella parte ordinata.

- Complessità temporale: O(n^2)
- Complessità spaziale: O(1)
- Stabile: Sì

Implementazione

```
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Confronto e Considerazioni

1. **Efficienza**: Tutti e tre gli algoritmi hanno una complessità temporale di O(n^2) nel caso peggiore, rendendoli inefficienti per grandi dataset. Tuttavia, l'Insertion Sort può essere efficiente per array quasi ordinati.
2. **Semplicità**: Questi algoritmi sono semplici da implementare e comprendere, rendendoli utili per scopi didattici.
3. **Stabilità**: Bubble Sort e Insertion Sort sono stabili, mentre Selection Sort non lo è. Un algoritmo stabile mantiene l'ordine relativo degli elementi con chiavi uguali.
4. **Uso pratico**: Per grandi dataset, algoritmi più avanzati come QuickSort, MergeSort o HeapSort sono preferibili.

Esempio di utilizzo

```
#include <iostream>
using namespace std;

// ... [inserire qui le funzioni di ordinamento] ...

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

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

    cout << "Array originale: ";
    printArray(arr, n);

    // Scegliere l'algoritmo di ordinamento
    bubbleSort(arr, n);
    // selectionSort(arr, n);
    // insertionSort(arr, n);

    cout << "Array ordinato: ";
    printArray(arr, n);

    return 0;
}
```

Questa mini dispensa fornisce una base solida per comprendere e implementare algoritmi di ordinamento di base in C++. Per applicazioni reali, considera l'uso di `std::sort()` della libreria standard C++, che implementa una versione ottimizzata dell'algoritmo IntroSort.