MergeSort: Algoritmo di Ordinamento Ricorsivo

Created	@January 19, 2025 5:19 PM
Class	Prog2

1. Cos'è il MergeSort?

Il **MergeSort** è un algoritmo di ordinamento basato sulla strategia **divide et impera**. La sua idea principale è dividere ricorsivamente l'array in sottoarray sempre più piccoli fino a che ogni sottoarray non ha un singolo elemento, che per definizione è già ordinato. Successivamente, i sottoarray vengono **uniti (fusi)** in ordine crescente. Questo processo di fusione è ciò che dà all'algoritmo il suo nome, "MergeSort".

Principali caratteristiche:

- **Ricorsivo**: l'algoritmo richiama sé stesso su sottogruppi sempre più piccoli dell'array.
- **Non in-place**: rispetto a Quicksort, MergeSort necessita di spazio aggiuntivo per la fusione dei sottoarray.
- **Stabile**: mantiene l'ordine degli elementi uguali, il che significa che non cambia la posizione degli elementi equivalenti nell'array originale.

```cpp
void merge(int array[], int const left, int const mid, int const right)
{
    int const subArrayOne = mid - left + 1;
    int const subArrayTwo = right - mid;

    // Crea 2 array temporanei
    int *leftArray = new int[subArrayOne],
        *rightArray = new int[subArrayTwo];

    // Copia i dati sugli array temporanei
    for (int i = 0; i < subArrayOne; i++)
        leftArray[i] = array[left + i];
    for (int j = 0; j < subArrayTwo; j++)
        rightArray[j] = array[mid + 1 + j];

    int indexOfSubArrayOne = 0, // Indice iniziale del primo subarray
        indexOfSubArrayTwo = 0; // Indice iniziale del secondo subarray
    int indexOfMergedArray = left; // Indice iniziale dell'array merged

    // Esegue il merge degli array temporanei in array[left..right]
    while (indexOfSubArrayOne < subArrayOne  && indexOfSubArrayTwo < subArrayTwo)
    {

        if (leftArray[indexOfSubArrayOne] <= rightArray[indexOfSubArrayTwo])
       {
            array[indexOfMergedArray]  = leftArray[indexOfSubArrayOne];
            indexOfSubArrayOne++;
        }
        else
       {
            array[indexOfMergedArray]  = rightArray[indexOfSubArrayTwo];
            indexOfSubArrayTwo++;
        }


        indexOfMergedArray++;
    }
    // Copia gli elementi rimanenti di left sull'array merged se ne esistono
    while (indexOfSubArrayOne < subArrayOne)
    {
        array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
        indexOfSubArrayOne++;
        indexOfMergedArray++;
    }
    // Copia gli elementi rimanenti di right sull'array merged se ne esistono
    while (indexOfSubArrayTwo < subArrayTwo)
    {
        array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
        indexOfSubArrayTwo++;
        indexOfMergedArray++;
    }
    delete[] leftArray;
    delete[] rightArray;
}
```

```cpp
void mergeSort(int array[], int const begin, int const end)
{
    if (begin >= end)
        return; // Returns recursively

    int mid = begin + (end - begin) / 2; //elemento medio
    mergeSort(array, begin, mid); //primo sottoarray
    mergeSort(array, mid + 1, end); //secondo sottoarray
    merge(array, begin, mid, end);
}
```

2. Passaggi principali dell'algoritmo

2.1. Divisione dell'array

Il primo passo consiste nel **dividere** l'array in due sottoarray, ognuno dei quali contiene metà degli elementi dell'array originale. Questo passaggio viene effettuato ricorsivamente, quindi ciascun sottoarray viene ulteriormente suddiviso fino a che ciascun sottoarray ha al massimo un elemento.

- Se l'array ha più di un elemento, lo dividiamo a metà.
- Se l'array ha un solo elemento (o nessuno), è già ordinato, quindi non viene ulteriormente diviso.

2.2. Fusione (Merge)

Dopo che l'array è stato diviso ricorsivamente, ogni sottoarray deve essere **fuso** (merge) per ottenere l'array ordinato. La fusione è un processo in cui due sottoarray ordinati vengono combinati in un solo sottoarray ordinato.

Come funziona la fusione?

1. Si crea un array temporaneo per contenere il risultato della fusione.
2. Si confrontano gli elementi dei due sottoarray, mettendo l'elemento più piccolo nell'array temporaneo.
3. Quando uno dei sottoarray è esaurito, si copiano tutti gli elementi dell'altro sottoarray rimanente nell'array temporaneo.
4. Alla fine, l'array temporaneo viene copiato nell'array originale.

La fusione garantisce che gli array rimangano ordinati durante la ricorsione.

2.3. Ricorsione

L'algoritmo di **MergeSort** applica ricorsivamente la divisione e la fusione su sottoarray di dimensioni sempre più piccole fino ad arrivare a un array di dimensione 1 (già ordinato). Ogni volta che i sottoarray vengono fusi, si ottiene un array più grande, fino a che l'intero array è ordinato.

2.4. Stop della ricorsione

La ricorsione termina quando un sottoarray ha una lunghezza di 1 o 0 elementi, in quanto questi sono già ordinati per definizione.

---

3. Fusione: Dettagli

La fase di **fusione** è il cuore dell'algoritmo MergeSort. Durante questo passaggio, due sottoarray ordinati vengono combinati in uno ordinato.

Come avviene la fusione?

- Si usano tre indici:
  - `i` per il primo sottoarray,
  - `j` per il secondo sottoarray,
  - `k` per l'array temporaneo che contiene il risultato della fusione.

1. Confrontiamo gli elementi `arr1[i]` e `arr2[j]` dei due sottoarray.
2. Se `arr1[i]` è più piccolo, lo copiamo in `arr[k]`, e incrementiamo `i` e `k`.
3. Se `arr2[j]` è più piccolo, lo copiamo in `arr[k]`, e incrementiamo `j` e `k`.
4. Quando uno dei sottoarray è esaurito, copiamo tutti gli elementi rimanenti dell'altro sottoarray nel risultato.
5. Alla fine, l'array risultante è ordinato e viene copiato nell'array originale.

---

4. Complessità dell'Algoritmo

La complessità di **MergeSort** è abbastanza prevedibile rispetto a quella di altri algoritmi di ordinamento. La sua complessità dipende principalmente dalla divisione e fusione ricorsiva.

4.1. Caso migliore:

- **Complessità**: O(n log n)
- Il caso migliore si verifica quando l'array è già ordinato. Tuttavia, a causa della divisione e della fusione ricorsiva, la complessità rimane comunque **O(n log n)**. MergeSort ha sempre questa complessità, indipendentemente dalla disposizione iniziale dei dati.

4.2. Caso peggiore:

- **Complessità**: O(n log n)
- Il caso peggiore si verifica quando l'array è completamente disordinato, ma anche in questo caso MergeSort mantiene la complessità **O(n log n)**, poiché l'algoritmo divide sempre l'array a metà e quindi il numero di divisioni è logaritmico, mentre la fusione richiede un tempo lineare.

4.3. Caso medio:

- **Complessità**: O(n log n)
- La complessità nel caso medio è anch'essa **O(n log n)**. La fusione è sempre lineare, mentre la ricorsione divide l'array logaritmicamente.

4.4. Ricorsione (profondità):

- La **profondità della ricorsione** è O(log n), poiché ogni divisione dell'array dimezza il numero di elementi.
- La fusione richiede un tempo lineare, O(n), per ciascun livello della ricorsione, e poiché ci sono O(log n) livelli, la complessità totale è O(n log n).

---

5. Pro e Contro del MergeSort

Pro:

1. **Complessità garantita O(n log n)**:
    - A differenza di algoritmi come Quicksort, MergeSort ha una **complessità garantita di O(n log n)** in tutti i casi (migliore, peggiore e medio). Questo lo rende affidabile per l'ordinamento di grandi quantità di dati.
2. **Stabilità**:
    - MergeSort è un algoritmo stabile, il che significa che se ci sono elementi uguali, la loro posizione relativa nell'array rimarrà invariata.
3. **Adatto per l'ordinamento di grandi dataset**:
    - Grazie alla sua complessità costante e prevedibile, MergeSort è adatto per l'ordinamento di **grandi dataset** e in contesti in cui la stabilità

dell'ordinamento è importante.

4. **Buono per dati già ordinati**:
    - Anche se la complessità è sempre O(n log n), MergeSort ha un comportamento più consistente su dati già ordinati rispetto ad altri algoritmi come Quicksort.

Contro:

1. **Non in-place**:
    - A differenza di Quicksort, che è in-place, MergeSort **necessita di spazio aggiuntivo** per memorizzare i sottoarray temporanei durante la fusione, il che significa che utilizza una memoria addizionale di O(n).
2. **Overhead di memoria**:
    - Il bisogno di un array temporaneo per la fusione porta a un **overhead di memoria**. In situazioni di memoria limitata, questo può diventare un problema.
3. **Non adatto a sistemi con memoria limitata**:
    - Poiché l'algoritmo utilizza spazio extra per la fusione, non è particolarmente adatto a sistemi con risorse di memoria molto limitate.