Quicksort: Algoritmo di Ordinamento Ricorsivo

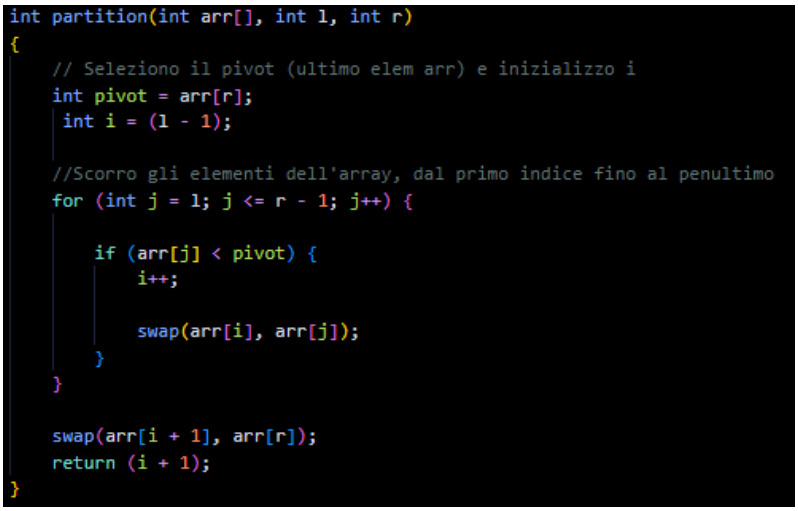
Created	@January 19, 2025 4:37 PM
Class	Prog2

1. Cos'è il Quicksort?

Il **Quicksort** è un algoritmo di ordinamento basato sulla strategia **divide et impera**, in cui un array viene diviso in sottoarray più piccoli, che vengono ordinati ricorsivamente. L'idea principale è che un array venga diviso in due parti, ognuna delle quali può essere ordinata separatamente. L'ordinamento finale avviene quando i sottoarray raggiungono una dimensione di 1 o 0 elementi, che per definizione sono già ordinati.

Principali caratteristiche:

- **Ricorsivo**: l'algoritmo richiama sé stesso su sottogruppi sempre più piccoli dell'array.
- **In-place**: l'ordinamento avviene senza bisogno di strutture dati ausiliarie, a meno che non venga implementata una versione modificata.
- **Scelta del pivot**: la qualità dell'algoritmo dipende dalla strategia di scelta del pivot.



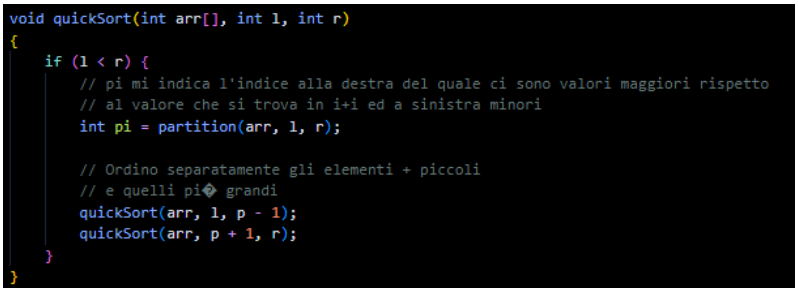
```c
int partition(int arr[], int l, int r)
{
    // Seleziono il pivot (ultimo elem arr) e inizializzo i
    int pivot = arr[r];
     int i = (l - 1);

    //Scorro gli elementi dell'array, dal primo indice fino al penultimo
    for (int j = l; j <= r - 1; j++) {

        if (arr[j] < pivot) {
            i++;

            swap(arr[i], arr[j]);
        }
    }

    swap(arr[i + 1], arr[r]);
    return (i + 1);
}
```



```c
void quickSort(int arr[], int l, int r)
{
    if (l < r) {
        // pi mi indica l'indice alla destra del quale ci sono valori maggiori rispetto
        // al valore che si trova in i+i ed a sinistra minori
        int pi = partition(arr, l, r);

        // Ordino separatamente gli elementi + piccoli
        // e quelli pi grandi
        quickSort(arr, l, p - 1);
        quickSort(arr, p + 1, r);
    }
}
```

---

2. Passaggi principali dell'algoritmo

2.1. Scegliere il pivot

Il **pivot** è un elemento scelto dall'array che serve come punto di riferimento per suddividere l'array. Nella tua implementazione, il pivot è sempre l'**ultimo elemento** del sottoarray. Esistono altre strategie di selezione, come la scelta del primo elemento, un pivot casuale, o il mediano tra il primo, centrale e ultimo elemento, ma l'ultimo elemento è un'opzione semplice.

2.2. Partizionamento

Il passaggio successivo è **partizionare** l'array. Questo significa separare gli elementi più piccoli o uguali al pivot a sinistra e quelli più grandi a destra. La fase di partizione è cruciale per garantire che l'algoritmo funzioni correttamente.

Come funziona il partizionamento?

- Utilizziamo due indici:

- `i` (inizialmente `low-1`): separa gli elementi che sono minori o uguali al pivot.
- `j` (inizialmente `low`): scorre attraverso l'array.

- Durante il ciclo, se l'elemento `arr[j]` è minore o uguale al pivot, incrementiamo `i` e scambiamo `arr[i]` con `arr[j]`.
- Alla fine del ciclo, il pivot viene scambiato con `arr[i+1]`, mettendolo nella sua posizione finale.

Partizione completata:

- Dopo il partizionamento, il pivot è nella sua posizione finale, e a sinistra troviamo tutti gli elementi minori o uguali a lui, mentre a destra troviamo gli elementi maggiori.

2.3. Ricorsione

Dopo il partizionamento, l'algoritmo applica **ricorsivamente** il Quicksort ai sottoarray:

- Sottoarray sinistro (elementi a sinistra del pivot).
- Sottoarray destro (elementi a destra del pivot).
  La ricorsione termina quando un sottoarray ha una lunghezza di 1 o 0 elementi (è già ordinato).

2.4. Stop della ricorsione

Quando l'array o il sottoarray contiene 1 o 0 elementi, la ricorsione si ferma, poiché questi sono già ordinati per definizione.

---

3. Partizionamento: Dettagli

La fase di **partizionamento** è il cuore del Quicksort. In un'implementazione con il pivot come ultimo elemento, si segue questo processo:

1. Inizializziamo due indici:
   - `i` viene inizializzato a `low - 1`.
   - `j` scorre da `low` a `high - 1`.

2. Per ogni elemento `arr[j]`, se `arr[j] <= pivot`, incrementiamo `i` e scambiamo `arr[i]` e `arr[j]`. Questo passo è essenziale per spostare gli elementi più piccoli del pivot a sinistra.
3. Alla fine del ciclo, il pivot (l'elemento in posizione `high`) viene scambiato con `arr[i+1]`, così che il pivot venga posizionato nella sua corretta posizione finale, con tutti gli elementi minori a sinistra e quelli maggiori a destra.

---

4. Complessità dell'Algoritmo

La complessità di **Quicksort** dipende dalla strategia di scelta del pivot e dalla configurazione iniziale dell'array.

4.1. Caso migliore:

- **Complessità**: O(log n)
- Questo accade quando il pivot divide l'array in due metà più o meno uguali ad ogni passo. In questo caso, la profondità della ricorsione è O(log n), e per ogni livello della ricorsione vengono effettuati O(n) confronti.

4.2. Caso peggiore:

- **Complessità**: O(n^2)
- Il caso peggiore si verifica quando il pivot è sempre l'elemento più grande o più piccolo dell'array, ad esempio se l'array è già ordinato o quasi ordinato. Questo porta a partizioni sbilanciate, con la ricorsione che si sviluppa su array di dimensioni quasi n, causando un numero quadratico di confronti.

4.3. Caso medio:

- **Complessità**: O(n log n)
- In media, scegliendo un pivot casuale o ben distribuito, la complessità è O(n log n). Questa è la situazione più comune, quando l'array viene diviso abbastanza equamente ad ogni passo.

---

5. Pro e Contro del pivot come ultimo elemento

Pro:

1. **Implementazione semplice**:
   - È una scelta facile e veloce da implementare.
   - Non richiede calcoli aggiuntivi o logiche complesse per determinare quale elemento scegliere come pivot.
2. **Efficienza su array casuali**:
   - Funziona bene in molti casi, specialmente con array casuali, poiché la probabilità di avere partizioni sbilanciate è bassa.
3. **In-place**:
   - L'algoritmo può essere implementato senza bisogno di array ausiliari, il che lo rende **efficientissimo in termini di memoria** (salvo l'uso della pila di ricorsione).

Contro:

1. **Casi pessimi su array ordinati o quasi ordinati**:
   - Se l'array è già ordinato o quasi ordinato, scegliere l'ultimo elemento come pivot porta a partizioni sbilanciate, con una complessità di O(n^2), che è molto più lenta.
2. **Dipendenza dalla disposizione iniziale**:
   - La qualità delle partizioni dipende fortemente dall'ordinamento dell'array. Se i dati sono già ordinati o parzialmente ordinati, l'algoritmo peggiora.
3. **Limitata flessibilità**:
   - Non si adatta dinamicamente alla distribuzione dei dati. Altre strategie di selezione del pivot, come il pivot casuale o il mediano di tre, sono più robuste.

Strategie alternative per migliorare la scelta del pivot:

1. **Pivot casuale**:
   - Si sceglie un elemento casuale come pivot. Questo riduce la probabilità di incontrare il caso peggiore, anche se non lo elimina del tutto.

- Complessità media: O(n log n).

2. **Mediana di tre (Median-of-Three)**:
   - Si prende il pivot come la mediana tra il primo, l'ultimo e l'elemento centrale dell'array.
   - Questo metodo tende a evitare pivot estremi, migliorando l'equilibrio delle partizioni nella maggior parte dei casi.
3. **Mediana reale (Median-of-Medians)**:
   - Si divide l'array in gruppi più piccoli (spesso di 5 elementi), si calcola la mediana di ogni gruppo, e si utilizza la mediana di queste mediane come pivot.
   - Questo approccio garantisce un pivot vicino alla mediana, ma è più costoso computazionalmente, quindi viene usato solo in applicazioni specifiche.
