2° lezione algoritmi

Due Date	@October 4, 2024
Materia	Algoritmi e Laboratorio
Status	Done

Algoritmi Di Ordinamento e ricorsione

Algortmo di ordinamento	Caso Pessimo	Caso migliore	Adattività	Stabile	In loco
Selection Sort	O(n^2)	O(n^2)	NO	X	X
Insertion Sort	O(n^2)	O(n)	X	X	X
Bubble Sort	O(n^2)	O(n^2)	NO	X	X

Caso migliore, peggiore del Bubble sort

Se hanno è formato da due cicli, forme e scambio elementi a due a due finché non li ordina si ha anche una versione con la sentinella nella quale se non è stata trovata una coppia elementi discordi l'algoritmo viene interrotto.

Caso Peggiore, Caso Migliore Selection Sort

Non esiste un input con il quale questo algoritmo si comporta meglio in termini di complessità infatti, la complessità del caso migliore e il caso peggiore sono uguali. Infatti si dice che questo algoritmo è stabile.

Caso migliore peggiore Insertion Sort

questo algoritmo ha la caratteristica di essere adattivo, cioè la capacità di sfruttare alcune caratteristiche dell'istanza che viene fornita in input per poter migliorare il suo tempo di esecuzione, cioè cercare di fare meno passi possibili. Nel caso pessimo, il comportamento del Inserion Sort è simile al Selection Sort questo si verifica solo in una situazione cioè quando istanza è ordinata al contrario. In suo tempo di esecuzione dipende dalla distanza massima si alza l'elemento e la sua posizione finale dell'evento stesso. Pertanto, se la distanza è piccola(tra l'elemento dell' e la sua posizione finale), il tempo di esecuzione dell'algoritmo si abbassa. Nel caso migliore in assoluto di questo algoritmo, è quando tutti gli elementi sono ordinati.

Cosa significa che un algoritmo lavoro in loco?

Un algoritmo del lavoro in loco non usa memoria aggiuntiva cioè non creare array di supporto.

Cio significa che non esegue gli spostamenti nell' array stesso.

Cos'è la stabilità di un algoritmo?

Un **algoritmo** è considerato **stabile** quando, applicandolo su input simili, non presenta grandi variazioni nel **tempo di esecuzione**. In altre parole, un algoritmo stabile ha un comportamento prevedibile e costante in termini di efficienza, indipendentemente da piccole variazioni nell'input.

Tuttavia, nel contesto specifico degli **algoritmi di ordinamento**, la **stabilità** assume un significato diverso. Un algoritmo di ordinamento è detto **stabile** quando, nel caso in cui due elementi abbiano lo stesso valore, il loro **ordine relativo** rimane invariato rispetto all'input originale. In pratica, se due elementi sono uguali, un algoritmo stabile li manterrà nella stessa posizione relativa all'interno della lista ordinata. Ad esempio, supponendo di voler ordinare tutti gli studenti dell'Università per nome, sicuramente ci sarà un caso di omonimia. Ogni elemento, anche se è uguale, ma si porta dietro degli elementi satellite che rendono gli elementi diversi.

[(Mario, 101), (Luigi, 102), (Mario, 103), (Luigi, 104), (Anna, 105)]

[(Anna, 105), (Luigi, 102), (Luigi, 104), (Mario, 101), (Mario, 103)]

Se applichiamo un **algoritmo di ordinamento stabile** basato sul nome, l'ordine relativo degli studenti con lo stesso nome (ad esempio, i due "Mario" o i due "Luigi") rimarrà invariato rispetto all'input originale.

[(Anna, 105), (Luigi, 104), (Luigi, 102), (Mario, 103), (Mario, 101)]

Qui notiamo che l'ordine relativo degli studenti con lo stesso nome è stato modificato:

- "Luigi, 104" ora compare prima di "Luigi, 102".
- "Mario, 103" ora compare prima di "Mario, 101".

Se un algoritmo di ordinamento riesce a mantenere la stabilità, potrebbe essere implementato con una cosa con priorità. La chiave che verrà associata a questi elementi è una priorità riduce pertanto che elementi con la chiave più piccola hanno la priorità più alta. Ma a parità di proprietà venga prima l'elemento che è arrivato prima. Gli algoritmi che hanno la dareste di stabilità ottengono questo risultato senza saperlo è una caratteristica dell'algoritmo implicita.

13_A	3_A	10_A	3_B	10_B	13_B	10_C	10_D

Al 2° passaggio 10_D e 13_D disturbano l'ordinamento originale.

```
MAX (A,M)
    M=0
    For I<-1 to N-1 DO
        IF A[M]<=A[I]
            THEN M<-I
    RETURN M

SELECTION SORT (A,N)
FOR I<-N DOWN TO 1 DO
    M<-MAX(A,I)
    SCAMBIA(A,M,I-1)
```

Da ciò di cui sopra si deduce che questo algoritmo non è stabile.

Una delle soluzioni potrebbe essere quella di utilizzare un array di appoggio per effettuare gli scambi.

L'insercion sort è stabile?

Nel Inserction invece gli scambi avvengono in posizione vicine nel while devo inserire il ≤.

Il Bubble sort è stabile?

Nel Bubble Sort gli scambi avvengono in posizione vicine e nel for dobbiamo mettere ≤.

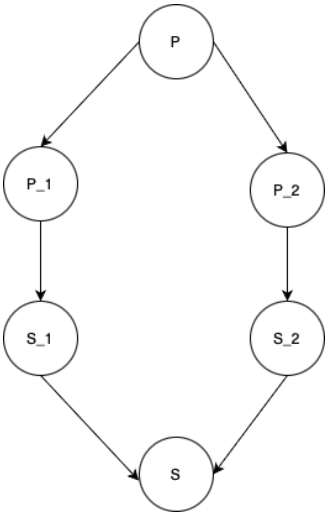
La ricorsione

Algortmo di ordinamento	Caso Pessimo	Caso migliore	Adattività	Stabile	In loco
Selection Sort	O(n^2)	O(n^2)	NO	X	X
Insertion Sort	O(n^2)	O(n)	X	X	X
Bubble Sort	O(n^2)	O(n^2)	NO	X	X
Quick Sort	O(n^2)	O(logn)	NO	NO	X
Mege Sort	O(nlogn)	O(nlogn)	NO	X	NO

Cos'è un algoritmo ricorsivo?

Un algoritmo ricorsivo si può applicare a diversi problemi che possono essere scomposti in dei sotto problemi che hanno la stessa natura. In due algoritmi che abbiamo citato prima dividono il problema in due sotto problemi.

In generale è un algoritmo ricorsivo, prende un problema P lo suddivide in due, so problemi P_1 e P_2 della stessa natura. Dopodiché si assume che la soluzione di questi due sotto problemi sia data S_1 S_2. Dopo questo passaggio, si deve trovare un modo per ricombinare le soluzioni di questi due sotto problemi per arrivare a quella del problema principale. Il vantaggio della ricorsione sta nel modo in cui si risolve il problema. Un algoritmo ricorsivo deve terminare ed ha anche un caso base quando il problema è così piccolo, che non ha bisogno di essere ulteriormente diviso.

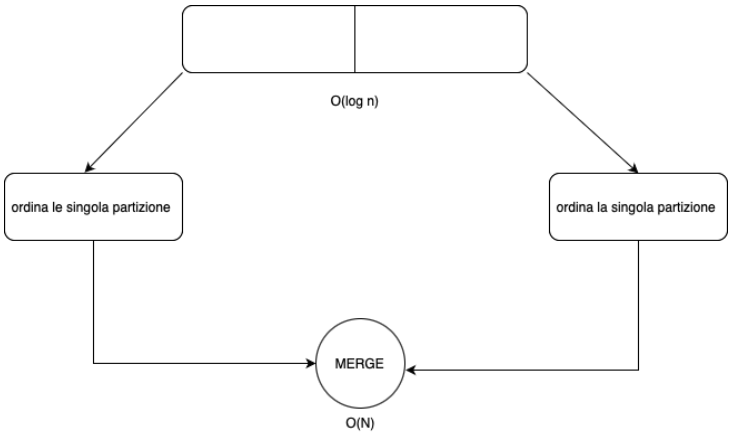


MergeSort

Questo algoritmo è stabile nei tempi di esecuzione come il Selection sort riesce ad avere un comportamento che sempre uguale in tutte le istanze che gli vengono passate. Da questo si deduce che non è un algoritmo adattivo. Si può dimostrare che la complessità di questa algoritmo non può essere migliorata da com'è. Il MergeSort è un algoritmo ottimo.

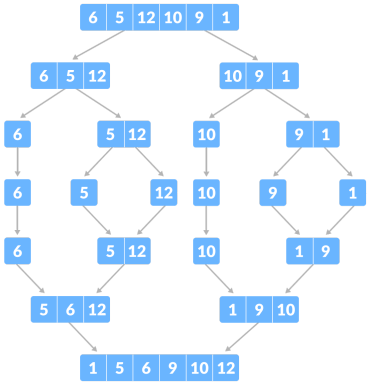
Come funziona il MergeSort?

Si va a dividere l' istanza in due parti e si vanno a ordinare queste due parti ricorsivamente.



1. D**ivide sempre le sequenze a metà impiegando un tempo O(log n)**
2. **Le unisce impiegando un tempo lineare**

```
MERGESORT(A,N)
    IF N <= 1 THEN RETURN
    M<-[N/2]
    MERGESORT(A,M)
    MERGESORT(A+M,N-M)
    MERGE(A,M,N)
```



QuckSort

l **Quicksort** è considerato uno degli algoritmi di ordinamento più efficienti grazie alla sua struttura basata sulla tecnica del *divide et impera*. L'algoritmo divide il vettore in tre partizioni fondamentali:

1. **Partizione centrale** contenente un solo elemento, chiamato *pivot*.
2. **Partizione sinistra** contenente tutti gli elementi minori del *pivot*.
3. **Partizione destra** contenente tutti gli elementi maggiori del *pivot*.

Dopo aver effettuato questa suddivisione, l'algoritmo viene applicato ricorsivamente sulle partizioni sinistra e destra fino a ordinare completamente il vettore.

Scelta del Pivot

La **scelta del pivot** è un aspetto cruciale per le prestazioni del Quicksort. Se il pivot è scelto in maniera inadeguata, può condurre al **caso peggiore**, dove la complessità dell'algoritmo diventa $O(N^2)$ tipicamente quando l'array è già ordinato o quasi ordinato. Per mitigare questo problema, esistono diverse strategie, tra cui quella della **"mediana di tre"**, dove il pivot viene scelto come la mediana tra il primo, l'ultimo e l'elemento centrale dell'array.

Complessità dell'algoritmo

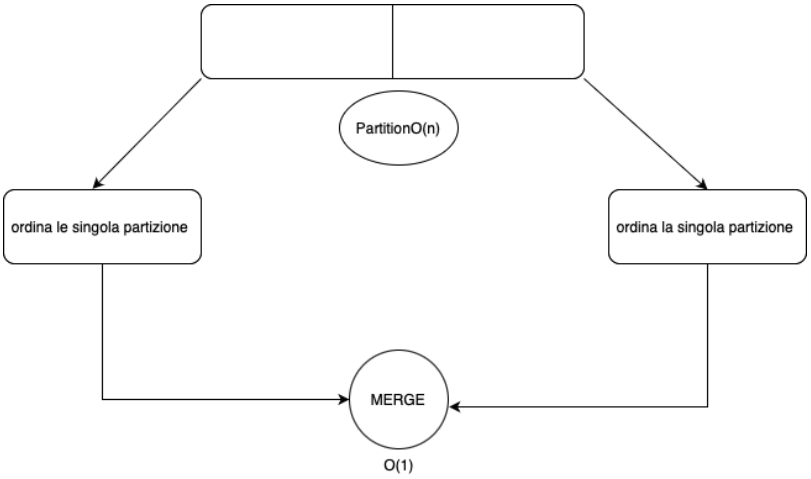
La complessità di Quicksort varia in base al bilanciamento delle partizioni:

- **Caso peggiore**: si verifica quando una delle due partizioni è molto sbilanciata, ovvero con 0 elementi in una partizione e n−1nell'altra. In questo scenario, la complessità è O(N^2) perché il partizionamento costa O(n) ad ogni livello, e il processo si ripete per nn livelli di ricorsione.

- **Caso migliore**: accade quando le partizioni sono bilanciate, ovvero entrambe le partizioni hanno dimensione approssimativa pari a n/2. In questo caso, la complessità è O(nlogn) dato che la profondità della ricorsione è lognlogn e ad ogni livello il costo per il partizionamento è O(n).
- **Caso medio**: anche in presenza di partizioni sproporzionate ma non estreme, Quicksort ha una complessità attesa di O(nlogn). Questo perché, anche con una certa sproporzione costante nella divisione, la profondità della ricorsione resta O(logn), e il costo di ogni livello rimane O(n).

Partitioning

La funzione di **partizionamento** è fondamentale per eseguire la divisione in sottoproblemi e restituire la posizione del pivot nel vettore, assicurando che gli elementi a sinistra del pivot siano minori e quelli a destra siano maggiori.




```
procedure QuickSort(array, left, right)
    if left < right then
        // Effettua la partizione e ottieni la posizione del pivot
        pivotIndex = Partition(array, left, right)

        // Richiama ricorsivamente il QuickSort sulla partizione sinistra
        QuickSort(array, left, pivotIndex - 1)

        // Richiama ricorsivamente il QuickSort sulla partizione destra
        QuickSort(array, pivotIndex + 1, right)
    end if
end procedure
```

Selection Sort (Ricorsivo)

```
SELECTION SORT(A,M)
    IF N<1 THEN RETURN
    M<-MAX(A,I)
    SCAMBIA(A,M,N-1)
    SELECTION SORT(A,N-1)
```

In questo caso, si ha un solo sotto problema.

Insertion Sort (Ricorsivo)

```
INSERTION SORT(A,N)
        FOR I<-1 TO N-1 DO
        INSERT(A,I)

INSERT(A,I)
    J<-I
    WHILE (J>0 AND A[J-1]>A[J]) DO
    SCAMBIA(A[J-1],J)
    J<-J-1
```