L09. ORDINAMENTI RICORSIVI

Esistono algoritmi di ordinamento, in forma ricorsiva, come il **Merge Sort** e **Quicksort**, che sono più efficienti rispetto ad altri visti in precedenza.
Abbiamo detto che il problema dell'ordinamento consiste nell'elencare gli elementi di un insieme secondo una sequenza stabilità da una relazione d'ordine. Questo problema si può presentare in forme differenti, dalle più semplici alle più complesse, ad esempio ci si può chiedere di ordinare una sequenza di numeri oppure di mettere un elenco di nomi in ordine alfabetico, quindi possiamo avere tipi di dati differenti o una forma più complessa potrebbe essere ordinare i record degli studenti secondo la data di nascita, questo problema un po' più complesso sia perché abbiamo a che fare con dati strutturati sia perché abbiamo una taglia maggiore dell'input. In quest'ultimo caso dobbiamo ordinare dei record secondo una chiave che può essere un singolo campo, ma anche come in altri casi la combinazione di diversi campi.
Ricordiamo che questi algoritmi di ordinamento hanno diverse proprietà, ovvero:

- **Stabile**, due elementi con la medesima chiave mantengono lo stesso ordine con cui si presentavano prima dell'ordinamento.
  Per tipi di dati semplici come gli interi o anche le stringhe non ci rendiamo conto se un algoritmo è stabile, quando invece abbiamo a che fare con dei record, un algoritmo stabile ci garantisce che dei record, che si presentavano ordinati prima dell'esecuzione dell'algoritmo, mantengono questo stesso ordinamento anche dopo l'esecuzione dell'algoritmo;
- **In loco**, in ogni istante al più è allocato un numero costante di variabili, oltre all'array da ordinare;
- **Adattivo**, il numero di operazioni effettuate dipende dall'input.
  Ad esempio, tra gli algoritmi come Insertion Sort e Bubble Sort possono essere implementati in modo adattivo;
- **Interno** (dati contenuti nella memoria RAM) vs **esterno** (dati contenuti su disco o nastro o file).

Tutti gli algoritmi visti, compresi questi nuovi, ordinano per confronti e non tutti gli algoritmi di ordinamento fanno questo, infatti esistono algoritmi che non ordinano per confronti.

Nome	Migliore	Medio	Peggiore	Memoria	Stabile	In Loco
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No	Sì
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Sì	Sì
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Sì	Sì
Quicksort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(n)$*	No	Sì
Merge Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	Sì	No

**spazio aggiuntivo dovuto alla gestione della ricorsione*

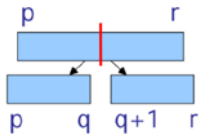
DIVIDE ET IMPERA:

Per quanto riguarda il Merge Sort ed il Quicksort, entrambi questi algoritmi utilizzano, per la risoluzione del problema dell'ordinamento, un approccio chiamato **Divide et impera**. Questo approccio può essere usato per risolvere vari problemi computazionali ed ha diverse caratteristiche:

- **Divide**: si procede alla suddivisione dei problemi in problemi di dimensione minore;
- **Impera**: i problemi vengono risolti in modo ricorsivo. Quando i sotto-problemi arrivano ad avere una dimensione sufficientemente piccola (si arriva al caso base), essi vengono risolti in modo immediato;
- **Combina**: si ricombina l'output, dopo la chiamata ricorsiva, ottenuto dalle precedenti chiamate ricorsive al fine di ottenere il risultato finale.

MERGE SORT:

È un algoritmo che utilizza il paradigma del **Divide et impera**, possiamo ottenere una versione stabile però richiede uno spazio ausiliario O(n), quindi non possiamo dire che ordina in loco, ma è molto efficiente per tutte le dimensioni dell'input, in genere viene fornito come algoritmo standard in molte librerie di alcuni linguaggi di programmazione, come ad esempio Java.
Per quanto riguarda la progettazione del Merge Sort, si divide a metà il vettore, quindi avremo un sotto-vettore sinistro e un sotto-vettore destro rispetto al centro:



- **Divide**:
  Supponiamo che abbiamo un vettore che vada dall'indice p all'indice r e che il centro si trova all'indice q. Quindi avrei un sotto-vettore sinistro che va da p a q, e non sotto-vettore destro che vada q+1 ad r.
- **Impera**:
  Identifichiamo il caso base che è quello in cui abbiamo un sotto-vettore di taglia unitaria oppure di taglia zero, quindi quando p=r oppure p>r, questo sotto-vettore si può considerare già ordinato. Nel caso in cui invece non abbiamo un vettore semplice di taglia unitarie dovremmo fare due chiamate ricorsive, quindi chiameremo ricorsivamente il merge sort sul sotto-vettore di sinistra e poi sul sotto-vettore di destra.
- **Combina**:
  Chiamiamo una procedura, detta **merge**, che fonde i due sotto-vettori ordinati in un unico vettore ordinato, sceglie ripetutamente il minimo dei due sotto-vettori e lo mette in una sequenza di output.

ANALISI FUNZIONE MERGE

- Dati di ingresso: Array a1 di n1 elementi, a2 di n2 elementi
  - Precondizione: $\forall\ 1<i<n1\ a1[i-1] <= a1[i]; \forall\ 1<j<n2\ a2[j-1] <= a2[j]$
- Dati di uscita: Array a di n1+n2 elementi
  - Postcondizione: $\forall\ el1 \in a1: el1 \in a$ AND $\forall\ el2 \in a2: el2 \in a$
    $\forall\ 1<i<n1+n2: a[i-1] <= a[i]$

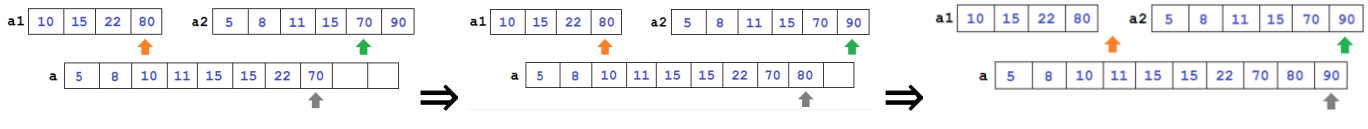
Dizionario dei dati

Identificatore	Tipo	Descrizione
a1,a2	array	array di interi in input
n1, n2	intero	# di elementi negli array a1, a2
a	array	array di interi in output
I, j	intero	usati per indicizzare i vettori

PROGETTAZIONE FUNZIONE MERGE

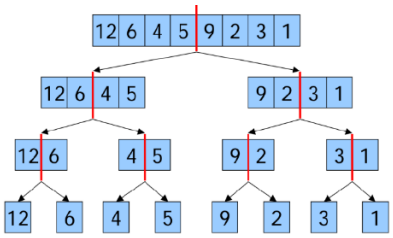
- Scorriamo i due vettori a1 e a2 utilizzando due indici i e j, rispettivamente;
- Confrontiamo a1[i] e a2[j] finché i<n1 e j<n2:
  - Se a1[i] <= a2[j]: Inseriamo a1[i] in a e incrementiamo i;
  - Altrimenti: inseriamo a2[j] in a e incrementiamo j;
- Riversiamo tutti gli elementi restanti in a1 o in a2 in a.

Esempio Merge:

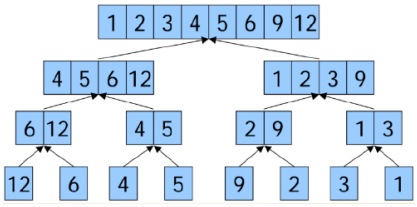


Esempio Merge Sort:

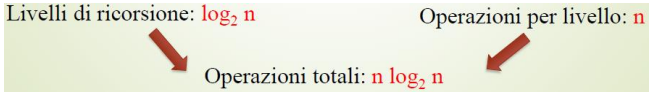
Divisione ricorsiva:



⇒ Combinazione:



Per quanto riguarda il costo computazionale:

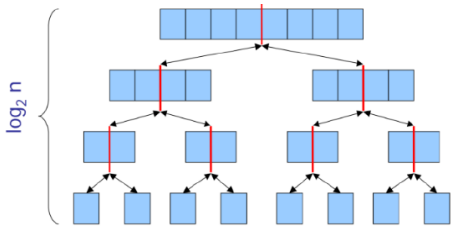


Equazione alle ricorrenze:

- $T(n) = 2T(n/2) + \Theta(n)$ per $n \geq 2$
- $T(1) = 1$

Soluzione:

- $T(n) = \Theta(n \log n)$



QUICKSORT:

Come nel caso del merge sort, è un algoritmo ricorsivo che fa uso del paradigma **Divide et impera**, a differenza del merge sort è in grado di ordinare un Array in loco, cioè non richiede una Array addizionale in cui riversare i valori durante l'esecuzione, però a differenza del merge sort non è stabile, quindi se prendiamo due elementi con la stessa chiave che prima dell'ordinamento si presentano in un certo ordine, non è detto che dopo l'esecuzione del quicksort questi si ripresentino nello stesso ordine.

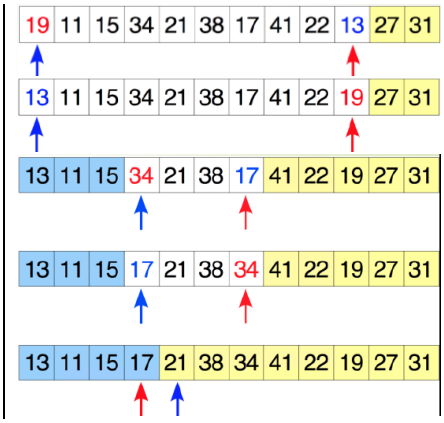
Il quicksort si basa su un algoritmo che viene chiamato **Partition**, questo algoritmo, dato un elemento **pivot**, mette tutti gli elementi minori uguali del pivot a sinistra e tutti gli elementi maggiori uguali del pivot sul lato destro dell'Array, restituirà la posizione in cui l'Array è stato suddiviso in questi numeri che sono minori e maggiori rispetto al pivot.

Supponiamo di aver scelto come pivot il valore 17, a questo punto, partiremo con due indici il blu sul lato sinistro dell'Array e il rosso sul lato destro.

Il puntatore rosso arretra finché non trovo un elemento che sia minore o uguale del pivot, mentre il puntatore blu avanza finché non trovo un elemento maggiore o uguale del pivot.

Il puntatore rosso scenderà di 3 posizioni fino ad arrivare a puntare al valore 13 e si fermerà poiché il 13 è minore o uguale del 17, il puntatore blu avanza e si ferma alla prima casella dell'array in quanto il contenuto è 19, che è maggiore uguale rispetto al pivot. A questo punto scambiamo i due elementi, cioè quello puntato dal puntatore blu e quello puntato dal puntatore rosso.

La procedura termina quando i due puntate si incrociano, infatti il puntatore posso scendere fino ad arrivare alla locazione contenente il 17, ma a questo punto i due puntatori si saranno incrociati e quindi termineremo la nostra procedura.



DESCRIZIONE PROCEDURA PARTITION:

Il primo passo è quello di scegliere il pivot, quindi mettiamo in x il valore del pivot che indicheremo con A[p], poi eseguiremo un ciclo infinito che terminerà quando i due puntato si incroceranno, quindi individueremo A[i] ed A[j] i due elementi che si trovano fuori posto, in particolare troveremo A[j] decrementando il puntatore j fino a trovare un elemento minore uguale del pivot x e incrementeremo i fino a trovare un elemento maggiore o uguale del pivot x. Se i due puntatori si sono incrociati restituiremo il valore di j che sarà esattamente il punto in cui l'Array risulta diviso da questa procedura, quindi si troverà il puntatore j sul valore dell'ultimo elemento del sotto-array di sinistra. Infine, come ultima istruzione del while scambieremo A[i] ed A[j].

Procedura Partition:

- Pivot x=A[p];
- While(1)
- Individua A[i] e A[j] elementi "fuori posto"
  - Decrementa jfino a trovare un elemento minore o uguale del pivot x
  - Incrementa ifino a trovare un elemento maggiore o uguale del pivot x
- If(i>=j) return j
- Scambia A[i] e A[j]

Vediamo quindi come utilizzare questa procedura Partition all'interno dell'algoritmo di ordinamento. Quicksort è un algoritmo ricorsivo che utilizza il paradigma **Divide et impera**, in particolare avremo a che fare con un array A i cui indici per comodità indicheremo da p ad r, A[p…r].

- **Divide**:
  Prevede di partizionare tale vettore in due sotto-vettori SX e DX rispetto ad un pivot x che si trova in una posizione q, questa posizione per come abbiamo progettato il nostro algoritmo Partition verrà restituito come valore di ritorno di tale procedura.
- **Impera**:
  Eseguiremo il quicksort prima sul sotto-vettore SX A[p…q] e poi chiameremo ricorsivamente quicksort su DX A[q+1…r]. La base della ricorsione avviene quando il vettore ha un unico elemento e quindi lo possiamo considerare un sotto-vettore ordinato.

Analizziamo il comportamento asintotico del quicksort, questo algoritmo ha un metodo di ordinamento basato su confronti, quindi quello che faremo è contare il numero di confronti che questo algoritmo effettua, inclusi i confronti che fa la Partition.

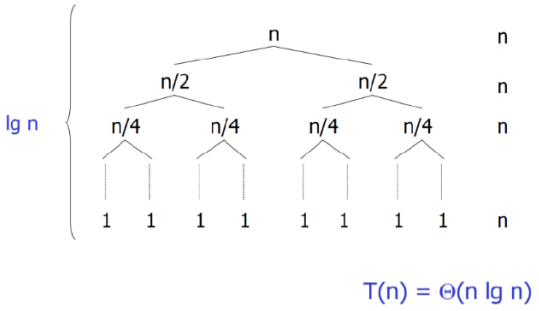
In realtà tutti i confronti vengono eseguiti nella procedura Partition poiché la procedura quicksort è molto semplice e contiene soltanto la chiamata da Partition e le due chiamate ricorsive. La procedura Partition fa avanzare quei due puntatori finché non si incrociano, ad ogni avanzamento o arretramento di uno dei due puntatori avremo che si eseguirà un confronto, quindi è corretto dire che se abbiamo un array di lunghezza m, eseguiremo m confronti.

L'array viene quindi partizionato, cioè diviso in due parti, e faremo delle chiamate ricorsive prima sulla parte sinistra e poi sulla parte destra, l'equazione di ricorrenza sarà:

$$T(n) = \begin{cases} 0 & \text{se } n = 1 \\ T(r) + T(n-r) + n & \text{se } n > 1 \end{cases}$$

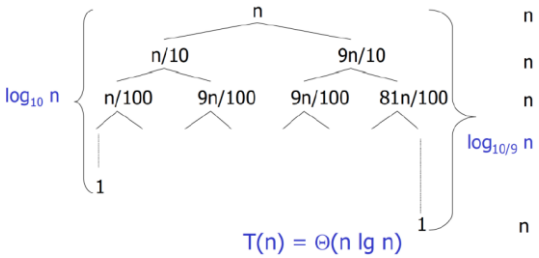
La forma di questa ricorsione non rientra nei casi che abbiamo studiato finora, a meno che non specializziamo i nostri casi (peggiore, migliore e medio). L'efficienza del Quicksort dipende da come va il bilanciamento delle partizioni, cioè come la Partition avrà diviso l'array, se è diviso in parti quasi uguali allora avremo un comportamento quasi ottimale, se invece l'algoritmo di Partition divide in modo molto sbilanciato gli Array avremmo che l'algoritmo avrà un tempo di esecuzione peggiore, in particolare il **caso peggiore** avviene quando la divisione della Partition ci restituisce un Array da un singolo elemento e l'altro da n-1 elementi, ma si può anche presentare quando l'array è già ordinato (sia decrescente che crescente).

Il **caso migliore** invece si verifica quando l'array viene diviso esattamente a metà, quindi i due array saranno di n mezzi elementi, possiamo vederlo con questo grafico:



È chiaro che il bilanciamento può essere più o meno buono a seconda di come viene scelto il pivot nella Partition.

Ed infine abbiamo un **caso medio**, possiamo vedere vari casi e capire se dobbiamo aspettarci più frequentemente un tempo quadratico oppure un tempo n log n. Supponiamo di essere sfortunati ed abbiamo che le due regioni sia una 9 volte più grande dell'altra:



Riassumendo, se abbiamo delle partizioni bilanciate, l'algoritmo ordina in tempo Θ(n log n), se invece siamo sfortunati e abbiamo delle partizioni sempre sbilanciate, avremo un costo di n^2. Tuttavia, considerando vari casi intermedi il tempo di esecuzione può essere sempre ricondotto ad un tempo asintotico Θ(n log n), quindi possiamo dire che nel caso medio, il quicksort ordina in tempo n log n.

RANDOM PIVOTING:

Abbiamo detto che la scelta del pivot determina il bilanciamento della partizione e che la scelta del primo elemento come pivot, nel caso in cui la sequenza di ingresso si presenti già ordinata, risulta in un partizionamento sfavorevole, quello che possiamo fare per evitare che delle sequenze particolari di ingresso ci creino questo problema di sbilanciamento è cambiare schema di scelta del pivot, non più il primo elemento ma una scelta casuale del pivot.

Infatti, il tempo di esecuzione del quicksort con **pivot casuale** è Θ(n log n) con una probabilità molto alta. Tuttavia, la scelta casuale ci costringe a chiamare una procedura di scelta di numeri pseudocasuali ad ogni esecuzione della Partition, questa cosa può diventare svantaggiosa dal momento che gli algoritmi di scelta numeri casuali hanno un loro costo, potrebbero rallentare l'esecuzione del quicksort, per questo abbiamo degli altri schemi di scelta del pivot che sono molto più rapidi, in particolare possono essere fatti in tempo costante.

Possiamo scegliere la strategia "**medio di 3**" che consiste nel considerare elementi che sono nella prima e ultima posizione dell'array e l'elemento che si trova in posizione mediana e scegliere di questi il valore mediano.

Supponiamo di avere un Array di interi, scegliamo come pivot il valore intermedio tra 39 che è il primo elemento, 31 è l'ultimo e 18 che è quello intermedio.

39	15	34	21	38	18	47	22	13	27	31

Scegliamo il valore di mezzo che in questo caso è il 31, che sarà il nostro pivot, è possibile scegliere questo valore mediano facendo due confronti quindi può essere fatta questa scelta in tempo costante.

MERGESORT

```
void merge(Item *a,int na, Item *b, int nb, Item *c){
   int i=0,j=0,k=0;
   Item v[na+nb];
   for(;i<na && j<nb;++k)
      if(cmpItem(a[i], b[j]) <= 0)
         v[k] = a[i++];
      else
         v[k]=b[j++];
   while(i<na)
      v[k++]=a[i++];
   while(j<nb)
      v[k++]=b[j++];
   for(k=0;k<na+nb;++k)
      c[k]=v[k];
}

void mergeSort(Item *a,int n){
   if(n>1){
      mergeSort(a,n/2);
      mergeSort(a+n/2,n-n/2);
      merge(a,n/2,a+n/2,n-n/2,a);
   }
}
```

QUICK SORT

```
void quickSort(Item*a,int n){      qSort(a,0,n-1);      }

void qSort(Item*a, int low, int high){
   if(high>low){
      int x = partition(a,low,high);
      qSort(a,low,x);
      qSort(a,x+1,high);
   }
}
int partition(Item*a, int low, int high){
   Item pivot = a[low];
   int i = low-1,j = high+1;
   while(1){
      do{
            j--;
      }while(cmpItem(a[j],pivot)>0);
      do{
            i++;
      }while(cmpItem(a[i],pivot)<0);
      if(i>=j) return j;
      swap(&a[i],&a[j]);
   }
}
```