

Corso di Laurea in Informatica e Tecnologie per la Produzione del Software (Track B) - A.A. 2018/2019

Laboratorio di Informatica

Algoritmi Fondamentali

(Parte 2)

docente: Cataldo Musto

cataldo.musto@uniba.it

Algoritmi di Ordinamento

- Disporre gli elementi di un insieme secondo una prefissata relazione d'ordine
 - Dipendente dal tipo di informazione
 - Numerica
 - Ordinamento numerico
 - Alfanumerica
 - Ordinamento lessicografico
 - Criterio di ordinamento
 - Crescente
 - Decrescente

- Una delle attività di elaborazione più importanti
 - Stima: 30% del tempo di calcolo di un elaboratore
- Non esiste un algoritmo migliore in assoluto
 - La bontà dipende da fattori connessi ai dati su cui deve essere applicato
 - Dimensione dell'insieme di dati
 - Numerosità
 - Grado di pre-ordinamento dell'insieme di dati
 - Già ordinato, parzialmente, ordine opposto, casuale

- Una delle attività di elaborazione più importanti
 - Stima: 30% del tempo di calcolo di un elaboratore
- Non esiste un algoritmo migliore in assoluto
 - La bontà dipende da fattori connessi ai dati su cui deve essere applicato
 - Dimensione dell'insieme di dati
 - Numerosità = Alcuni algoritmi funzionano bene su insiemi piccoli, meno bene su insiemi grandi
 - Grado di pre-ordinamento dell'insieme di dati
 - Già ordinato, parzialmente, ordine opposto, casuale = Alcuni algoritmi funzionano meglio se l'insieme è già pre-ordinato

- Gran varietà di algoritmi
 - Basati su confronti e scambi fra gli elementi

- Gran varietà di algoritmi
 - Basati su confronti e scambi fra gli elementi
- Obiettivo: efficienza
 - Sfruttare "al meglio" i confronti ed i conseguenti spostamenti degli elementi
 - Piazzare gli elementi prima possibile più vicino alla loro posizione finale nella sequenza ordinata

Algoritmi esterni

- Usano un array di appoggio
 - Occupazione di memoria doppia
 - Necessità di copiare il risultato nell'array originale
 - Esempio: Algoritmo Enumerativo
 - Ciascun elemento confrontato con tutti gli altri per determinare il numero degli elementi dell'insieme che sono più piccoli in modo da stabilire la sua posizione finale

Algoritmi interni

- Eseguono l'ordinamento lavorando sullo stesso array da ordinare
 - Basati su due concetti: confronti tra valori e scambi di posizione degli elementi

Algoritmo di Ordinamento di Base

Per Selezione (Selection Sort)

 Elemento più piccolo localizzato e separato dagli altri, quindi selezione del successivo elemento più piccolo, e così via

A bolle (Bubble Sort)

 Coppie di elementi adiacenti fuori ordine scambiate, finché non è più necessario effettuare alcuno scambio

Per Inserzione (Insert Sort)

• Elementi considerati uno alla volta e inseriti al posto che gli compete all'interno degli altri già ordinati

Ordinamento per Selezione

- Basato sul concetto di «Minimi successivi»
 - Trovare il più piccolo elemento dell'insieme e porlo in prima posizione
 - Scambio con l'elemento in prima posizione
 - Trovare il più piccolo dei rimanenti (n 1) elementi e sistemarlo in seconda posizione
 - •
 - Finché si trovi e collochi il penultimo elemento
 - Ultimo sistemato automaticamente

Ordinamento per Selezione - Esempio

	Inizio	I	II	III	IV	V
array(1)	44	44	11	11	11	11
array(2)	33	33	33	22	22	22
array(3)	66	66	66	66	33	33
array(4)	11	11	44	44	44)	44
array(5)	55	55	55	55	55	55
array(6)	22	22	22	33	66	66

Ordinamento per Selezione - Algoritmo

Pseudocodice

```
i \leftarrow 1

finchè i < n

trova il minimo valore nella lista (i \dots n)

scambia la posizione del valore minimo con lista(i)
i \leftarrow i + 1
```

- Basato su algoritmi già noti
 - Ricerca del minimo
 - Scambio

Ordinamento per Selezione - Algoritmo

Perché abbiamo bisogno di n cicli?

Pseudocodice

```
i \leftarrow 1

finchè i n

trova il minimo valore nella lista (i \dots n)

scambia la posizione del valore minimo con lista(i)
i \leftarrow i + 1
```

- Basato su algoritmi già noti
 - Ricerca del minimo
 - Scambio

Ordinamento per Selezione - Algoritmo

Pseudocodice

 $i \leftarrow 1$ finchè $i \in n$ trova il mi

Perché abbiamo bisogno di *n* cicli?

Perché ad ogni passo «ordino» un elemento (il più piccolo) quindi mi servono **n cicli** in tutto

trova il minimo valore nella lista (i ... n) scambia la posizione del valore minimo con lista(i) $i \leftarrow i + 1$

- Basato su algoritmi già noti
 - Ricerca del minimo
 - Scambio

Ordinamento per Selezione - Programma C

```
void selectionSort(int a[], int n) {
  for (i=0; i < n-1; i++) {
     min = a[i]; p = i; // p = posizione del minimo, min = val. minimo</pre>
```

.

Ordinamento per Selezione - Programma C

```
void selectionSort(int a[], int n) {
 for (i=0; i < n-1; i++) {
              min = a[i]; p = i; // p = posizione del minimo, min = val. minimo
              for (j = i+1; j < n; j++) \{ // ricerca del minimo \}
```

Ordinamento per Selezione - Programma C

```
void selectionSort(int a[], int n) {
 for (i=0; i < n-1; i++) {
             min = a[i]; p = i; // p = posizione del minimo, min = val. minimo
             for (j = i+1; j < n; j++) \{ // ricerca del minimo \}
                     if (a[j] < min) {
                           min = a[j];
                           p = j;
              a[p] = a[i]; // una volta individuato il minimo, effettuo
              a[i] = min; // lo scambio tra i valori
```

Ordinamento per Selezione - Complessità

Confronti

• La complessità totale è data dalla complessità dei due cicli

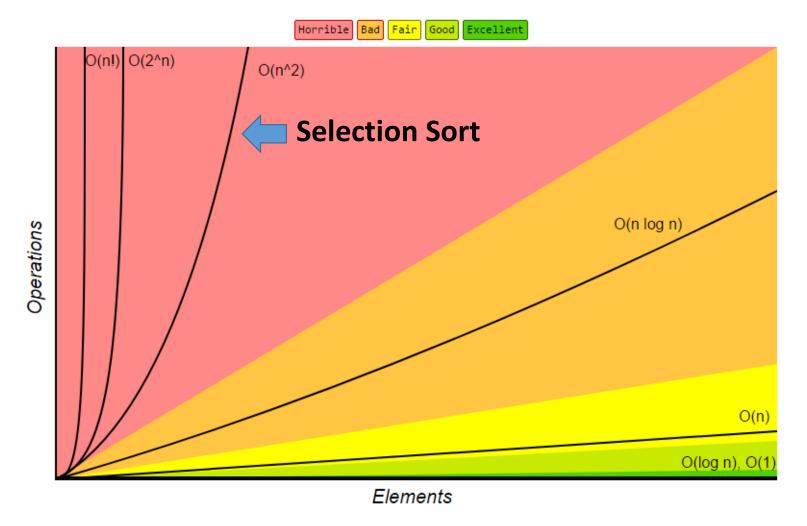
Ordinamento per Selezione - Complessità

Confronti

- Sempre $(n-1) * n \rightarrow O(n^2)$
 - La complessità totale è data dalla complessità dei due cicli
 - Il primo ciclo si ripete n-1 volte
 - Ciascun ciclo a sua volta si innesta in un ciclo che si ripete circa n volte
- Complessità quadratica -> possiamo trovare algoritmi più efficienti

(Recap) Complessità di un Algoritmo

Big-O Complexity Chart



Ordinamento per Selezione - Complessità

Confronti

- Sempre $(n-1) * n \rightarrow O(n^2)$
 - La complessità totale è data dalla complessità dei due cicli
 - Il primo ciclo si ripete n-1 volte
 - Ciascun ciclo a sua volta si innesta in un ciclo che si ripete circa n volte
- Complessità quadratica -> possiamo trovare algoritmi più efficienti

Scambi ?

Ordinamento per Selezione - Complessità

Confronti

- Sempre $(n-1) * n \rightarrow O(n^2)$
 - La complessità totale è data dalla complessità dei due cicli
 - Il primo ciclo si ripete n-1 volte
 - Ciascun ciclo a sua volta si innesta in un ciclo che si ripete circa n volte
- Complessità quadratica -> possiamo trovare algoritmi più efficienti

Scambi

- Al più (*n* − 1)
 - 1 per ogni passo

Ordinamento per Selezione - Considerazioni

- Ogni ciclo scorre tutta la parte non ordinata
- Limite importante!
 - Non trae vantaggio dall'eventuale preordinamento
 - Non c'è un caso migliore/peggiore.
 - Numero fisso di confronti
- Vantaggio: Pochi scambi
 - Ogni scambio richiede tre operazioni (ciascuna di complessità O(1))

- Far "affiorare" ad ogni passo l'elemento più piccolo fra quelli in esame
 - Confronto fra coppie di elementi adiacenti e
 - Se sono fuori ordine, **scambio**
 - Ripetendo il tutto fino ad ottenere la sequenza ordinata
 - Simile alle bolle di gas in un bicchiere
 - Ad ogni passo l'elemento più piccolo «sale» in prima posizione

Il passo di esecuzione I termina nel momento in cui l'elemento più piccolo viene posizionato nella prima posizione. Ad ogni passo si ordina un elemento (ma possiamo avere più scambi!).

	Inizio	I/1	I/2	I/3	I/4	II
array(1)	4	4	4	4	4	1
array(2)	3	3	3	3	1	4
array(3)	6	6	6	1	3	3
array(4)	1	1	1	6	6	6
array(5)	5	2	2	2	2	2
array(6)	2	5	5	5	5	5

Si parte dall'ultimo elemento e lo si scambia finchè non si trova un elemento più piccolo. Si ripete il processo finchè l'elemento più piccolo è portato in testa

	Inizio	I/1	I/2	I/3	I/4	II
array(1)	4	4	4	4	4	1
array(2)	3	3	3	3	1	4
array(3)	6	6	6	1	3	3
array(4)	1	1	1	6	6	6
array(5)	5	2	2	2	2	2
array(6)	2	5	5	5	5	5

Si parte dall'ultimo elemento e lo si scambia finchè non si trova un elemento più piccolo. Si ripete il processo finchè l'elemento più piccolo è portato in testa

	II	II/1	II/2	II/3	III
array(1)	1	1	1	1	1
array(2)	4	4	4	<i>→</i> 4	2
array(3)	3	3	3	2	4
array(4)	6	6	2	3	3
array(5)	2	2	6	6	6
array(6)	5	5	5	5	5

Quando uno scambio non viene effettuato non significa che L'elemento che stiamo analizzando sia il più grande Significa solo che l'ordinamento con il suo adiacente è corretto

	III	III/1	III/2	IV	IV/1	Fine
array(1)	1	1	1	1	1	1
array(2)	2	2	2	2	2	2
array(3)	4	4	4	3	3	3
array(4)	3	3	3	4	4	4
array(5)	6	5	5	5	5	5
array(6)	5	6	6	6	6	6

• Quando **termina** l'algoritmo?

- Quando termina l'algoritmo?
 - Dipende dal numero di scambi che vengono effettuati.
- Quando ci accorgiamo che l'algoritmo deve continuare?

- Quando termina l'algoritmo?
 - Dipende dal numero di scambi che vengono effettuati.
- Quando ci accorgiamo che l'algoritmo deve continuare?
 - Quando viene effettuato almeno uno scambio!

- Quando termina l'algoritmo?
- Se in una passata non viene effettuato nessuno scambio, l'insieme è già ordinato
 - L'algoritmo può già terminare anche in meno di n-1 passi
 - Vantaggio rispetto al Selection Sort, che richiede sempre n-1 passi.

- Quando termina l'algoritmo?
- Se in una passata non viene effettuato nessuno scambio, l'insieme è già ordinato
 - L'algoritmo può già terminare
 - Vantaggio rispetto al Selection Sort, che richiede sempre n-1 passi.
- Come implementare questo concetto?
 - Usare un indicatore di scambi effettuati (variabile booleana)
 - Impostato a vero all'inizio di ogni passata
 - Impostato a falso non appena si effettua uno scambio
 - Si termina se alla fine di un passo è rimasto inalterato

Ordinamento a Bolle - Algoritmo

```
p \leftarrow 0 // passo di esecuzione (ma indica anche la porzione del vettore ordinata) ordinato \leftarrow falso // indicatore di scambi finchè p < n e non ordinato esegui // esce dal ciclo dopo max n-1 passi
```

Ordinamento a Bolle - Algoritmo

```
p \leftarrow 0 // passo di esecuzione (ma indica anche la porzione del vettore ordinata) ordinato \leftarrow falso // indicatore di scambi finchè p < n e non ordinato esegui // esce dal ciclo dopo max n-1 passi p \leftarrow p + 1 ordinato \leftarrow vero i \leftarrow n // memorizza l indice dell'elemento «in esame»
```

Ordinamento a Bolle - Algoritmo

```
p \leftarrow 0 // passo di esecuzione (ma indica anche la porzione del vettore ordinata)
ordinato ← falso // indicatore di scambi
finchè p < n e non ordinato esegui // esce dal ciclo dopo max n-1 passi
 p \leftarrow p + 1
 ordinato ← vero
 i ← n // memorizza l indice dell'elemento «in esame» (parto dall'ultimo)
  finchè i > p esegui
       se lista(i) < lista (i - 1) allora</pre>
              scambia lista(i) con lista(i - 1)
              ordinato ← falso // se effettuo uno scambio l'insieme
       i \leftarrow i - 1
                                 // non è più ordinato
```

Ordinamento a Bolle - Algoritmo

Perché questa condizione?

```
p \leftarrow 0 // passo di esecuzione (ma indica anche la porzione d
ordinato ← falso // indicatore di scambi
finchè p < n e non ordinato esegui // esce dal ciclo dopo max n-1 passi
 p \leftarrow p + 1
 ordinato \leftarrow vero
               // memorizza l indice dell'elemento «in esame» (parto dall'ultimo)
 finch( i > p g/segui
       se lista(i) < lista (i - 1) allora</pre>
               scambia lista(i) con lista(i - 1)
               ordinato \leftarrow falso // se effettuo uno scambio l'insieme
       i \leftarrow i - 1
                                    // non è più ordinato
```

Ordinamento a Bolle - Algoritmo

```
primo passo avremo per certo l'elemento
p \leftarrow 0 // passo di esecuzione (ma indica anche la porzione
                                                                        più piccolo in alto).
ordinato ← falso // indicatore di scambi
                                                                        Termina il ciclo quando la variabile i (che
                                                                        viene decrementata ad ogni passaggio)
finchè p < n e non ordinato esegui // esce dal ciclo do
                                                                        diventa uguale a p, dunque quando
                                                                        abbiamo raggiunto la porzione di vettore
  p \leftarrow p + 1
                                                                        che sappiamo essere ordinata.
  ordinato \leftarrow vero
                  memorizza l indice dell'elemento «in esame» (parto dall'ultimo)
  finch( i > p esegui
        se lista(i) < lista (i - 1) allora</pre>
                 scambia lista(i) con lista(i - 1)
                 ordinato ← falso // se effettuo uno scambio l'insieme
        i \leftarrow i - 1
                                         // non è più ordinato
```

Ad ogni passo noi abbiamo esattamente p

abbiamo zero elementi già ordinati. Dopo il

elementi già ordinati (al primo passo

Ordinamento a Bolle - Programma C

```
void bubbleSort(int[] a, int n) {
 sorted = 0; p = 0;
 while (p < n) && (!sorted) {
       sorted = true; p = p + 1;
       for (i = n-1; i >= p; i--) { //scorro dal fondo fino all'ultimo elemento «ordinato»
               if (a[i] < a[i-1]) {
                       t = a[i]; // scambio
                       a[i] = a[i-1];
                       a[i-1] = t;
                       sorted = false // modifico la variabile booleana
```

Ordinamento a Bolle Recap (parziale)

	Inizio	I/1	I/2	I/3	I/4	II
array(1)	4	4	4	4	4	1
array(2)	3	3	3	3	1	4
array(3)	6	6	6	1	3	3
array(4)	1	1	1	6	6	6
array(5)	5	2	2	2	2	2
array(6)	2	5	5	5	5	5

• Caso migliore (lista già ordinata):

- Caso migliore (lista già ordinata): 1 passo
 - La variabile «sorted» resta settata a true

- Caso migliore (lista già ordinata): 1 passo
 - La variabile «sorted» resta settata a true
 - Il ciclo «esterno» viene eseguito solo una volta, il ciclo «interno» n volte

- Caso migliore (lista già ordinata): 1 passo
 - La variabile «sorted» resta settata a true
 - n-1 confronti, 0 scambi \rightarrow O(n)
- Caso peggiore (ordine opposto): n-1 passi

- Caso migliore (lista già ordinata): 1 passo
 - La variabile «sorted» resta settata a true
 - n-1 confronti, 0 scambi \rightarrow O(n)
- Caso peggiore (ordine opposto): n − 1 passi
 - All'i-mo passo
 - n-i confronti \rightarrow in tutto (n-1) * $n/2 \sim O(n^2)$
 - Come per Selezione
 - n-i scambi \rightarrow in tutto (n-1) * $n/2 \sim O(n^2)$
 - Molto maggiore della Selezione
- Caso medio
 - Scambi pari alla metà dei confronti $\rightarrow O(n^2)$

Ordinamento a Bolle - Considerazioni

- Ogni ciclo scorre tutta la parte non ordinata
- Prestazioni medie inferiori agli altri metodi
 - Nel caso peggiore, numero di confronti uguale all'ordinamento per selezione, ma numero di scambi molto maggiore
 - Molto veloce per insiemi con alto grado di preordinamento

Ordinamento a Bolle - Considerazioni

- Ogni ciclo scorre tutta la parte non ordinata
- Prestazioni medie inferiori agli altri metodi
 - Nel caso peggiore, numero di confronti uguale all'ordinamento per selezione, ma numero di scambi molto maggiore
 - Molto veloce per insiemi con alto grado di preordinamento

Bubble Sort > Selection Sort se l'insieme è pre-ordinato
Selection Sort > Bubble Sort, viceversa (perché Bubble Sort effettua
più scambi)

- Metodo intuitivo
 - Simile all'ordinamento eseguito sulle carte da gioco

- Metodo intuitivo
 - Simile all'ordinamento eseguito sulle carte da gioco
- Ricerca la giusta posizione d'ordine di ogni elemento rispetto alla parte già ordinata
 - Inizialmente già ordinato solo il primo elemento
 - Al passo «zero» avremo un elemento ordinato, al primo passo avremo due elementi ordinati, al secondo passo avremo tre elementi ordinati, dopo n-1 passi avremo tutti gli elementi ordinati
 - Elementi da ordinare considerati uno per volta
 - Si confronta l'elemento con tutti quelli della «parte ordinata» e lo si colloca nella giusta posizione
- Effettua n-1 passi

Ordinamento per Inserzione Esempio

	Inizio	I	II	III	IV	V
array(1)	40	30	30	10	10	10
array(2)	30	40	40	30	∂ 30	20
array(3)	60	60	60	40	40	30
array(4)	10	10	10	60	50	40
array(5)	50	50	50	50	60	50
array(6)	20	20	20	20	20	60

- Ad ogni passo dell'algoritmo abbiamo una parte del vettore ordinata e una parte non ordinata
 - Quanti elementi ordinati e quanti non ordinati al ciclo i?

- Ad ogni passo dell'algoritmo abbiamo una parte del vettore ordinata e una parte non ordinata
 - Al ciclo *i*, avremo *i+1* elementi ordinati e n-(i+1) non ordinati
 - Determinare la posizione in cui inserire l'elemento nella sequenza ordinata, facendo scalare le altre
 - Scansione sequenziale

- Ad ogni passo dell'algoritmo abbiamo una parte del vettore ordinata e una parte non ordinata
 - Al ciclo *i*, avremo *i+1* elementi ordinati e *n-(i+1)* non ordinati
 - Determinare la posizione in cui inserire l'elemento nella sequenza ordinata, facendo scalare le altre
 - Scansione sequenziale
- Soluzione completa costruita inserendo un elemento della parte non ordinata nella parte ordinata, estendendola di un elemento

```
| parte ordinata | x | parte non ordinata | \rightarrow
| \leq x | x | >x | parte non ordinata | \rightarrow x viene collocato nel punto corretto della parte ordinata
```

per ogni elemento dal secondo fino all'ultimo esegui
inserito ← falso // variabile booleana per «capire» se l'elemento è stato inserito
finchè non è stato inserito esegui

```
per ogni elemento dal secondo fino all'ultimo esegui
  inserito ← falso // variabile booleana per «capire» se l'elemento è stato inserito
  finchè non è stato inserito esegui
    se è minore del precedente allora
    inverti i due valori
```

```
per ogni elemento dal secondo fino all'ultimo esegui
  inserito ← falso // variabile booleana per «capire» se l'elemento è stato inserito
  finchè non è stato inserito esegui
    se è minore del precedente allora
    inverti i due valori
```

```
altrimenti
     piazzalo; inserito ← vero
fine_se
```

```
per ogni elemento dal secondo fino all'ultimo esegui
 inserito ← falso // variabile booleana per «capire» se l'elemento è stato inserito
 finchè non è stato inserito esegui
       se è minore del precedente allora
               inverti i due valori
               se sei arrivato in prima posizione allora
                       piazzalo; inserito ← vero
               fine se
       altrimenti
               piazzalo; inserito ← vero
       fine_se
 fine finchè
fine perogni
```

```
void insertion_sort(int x[], int n) {
 int i, j, app;
 for (i=1; i<n; i++) { // ciclo che scorre tutti gli elementi (parte dal secondo)
   app = x[i];
                          // salva il valore da posizionare
   i = i-1;
                         // memorizza l'indice dell'ultimo valore ordinato
   while (j>=0 && x[j]>app) { // ciclo per individuare la posizione corretta
     x[j+1] = x[j]; // quando esce dal ciclo?
     j--;
   x[j+1] = app;
  return;
```

```
void insertion_sort(int x[], int n) {
 int i, j, app;
 for (i=1; i<n; i++) { // ciclo che scorre tutti gli elementi (parte dal secondo)
   app = x[i];
                          // salva il valore da posizionare
   i = i-1;
                          // memorizza l'indice dell'ultimo valore ordinato
   while (j>=0 && x[j]>app) { // ciclo per individuare la posizione corretta
     x[j+1] = x[j]; // se si è arrivati al primo elemento (j==0) oppure
     j--; // se l'elemento corrente è più piccolo di quello da posizionare
   x[j+1] = app;
  return;
```

```
void insertion_sort(int x[], int n) {
 int i, j, app;
 for (i=1; i<n; i++) { // ciclo che scorre tutti gli elementi (parte dal secondo)</pre>
   app = x[i];
                          // salva il valore da posizionare
   i = i-1;
                           // memorizza l'indice dell'ultimo valore ordinato
   while (j>=0 && x[j]>app) { // ciclo per individuare la posizione corretta
                          // se si è arrivati al primo elemento (j==0) oppure
     x[i+1] = x[i];
     j--; // se l'elemento corrente è più piccolo di quello da posizionare
   x[j+1] = app; // esce dal ciclo quando ha trovato la posizione per l'elemento
                  // dunque inserisce il valore esattamente lì
 return;
```

Ordinamento per Inserzione - variante

- Variante più efficace, proposta da Dromey
 - Inserire subito il più piccolo in prima posizione
 - Evita di dover effettuare appositi controlli sull'indice per evitare che esca fuori dall'array

```
Cerca il minimo
Prima posizione ← minimo
mentre c'è una parte non ordinata
   Considera il primo elemento di tale parte
   Confrontalo a ritroso con i precedenti, facendoli via via scalare finché sono maggiori
```

Quanti passi?

- Quanti passi? Sempre n − 1
 - Uno scambio per ogni confronto, salvo (eventualmente) l'ultimo
 - Caso ottimo (lista già ordinata)

- Quanti passi? Sempre n − 1
 - Uno scambio per ogni confronto, salvo (eventualmente) l'ultimo
 - Caso ottimo (lista già ordinata)
 - Non entra mai nel ciclo «while», effettua solo il ciclo «for»

- Quanti passi? Sempre n − 1
 - Uno scambio per ogni confronto, salvo (eventualmente) l'ultimo
 - Caso ottimo (lista già ordinata)
 - n-1 confronti, 0 scambi \rightarrow O(n) confronti, O(1) scambi
 - Come il metodo a bolle
 - Caso pessimo (ordine opposto)

- Quanti passi? Sempre n − 1
 - Uno scambio per ogni confronto, salvo (eventualmente) l'ultimo
 - Caso ottimo (lista già ordinata)
 - n-1 confronti, 0 scambi \rightarrow O(n) confronti, O(1) scambi
 - Come il metodo a bolle
 - Caso pessimo (ordine opposto)
 - *i*-mo passo
 - i-1 confronti e scambi -> (n-1) * $n/2 \sim O(n^2)$
 - Caso medio: metà confronti e scambi, ma la complessità è sempre $O(n^2)$

Ordinamento per Inserzione - Considerazioni

- Per sequenze con distribuzione casuale abbiamo
 - Molti confronti
 - Molti scambi
 - Caso migliore come l'ordinamento a bolle

Valido per

- Piccole sequenze $(n \le 25)$
- Sequenze note a priori essere parzialmente ordinate
- Ogni ciclo scorre una porzione della parte ordinata

Ordinamento - Considerazioni

- Scambio più costoso del confronto
 - Confronto operazione base del processore (abbiamo un operatore!)
 - Scambio composto da tre assegnamenti
 - Un assegnamento richiede due accessi alla memoria
- Ad ogni passo La porzione ordinata cresce di una unità
 - La porzione disordinata decresce di una unità

Esercizio 15.1

- Realizzare una funzione C che implementi un algoritmo di ordinamento a scelta, tra Selection Sort, Bubble Sort e Insertion Sort
- Richiamare la funzione in un main(), contenente un vettore di valori e stampare la sequenza di valori non ordinata e la sequenza di valori ordinata
- (A casa) Testare il programma con una suite di test Cunit

- Prototipo (ideale): void sort(int array[], int n)
 - Opzionalmente si potrebbe aggiungere anche un parametro «criterio»

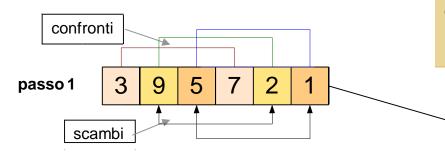
Algoritmi (Avanzati) di Ordinamento

Tecniche Avanzate di Ordinamento

- Gli algoritmi di ordinamento di base hanno una complessità troppo elevata (nel caso medio/pessimo) per risolvere in modo efficace problemi reali
 - Mediamente, O(n²) per tutti gli algoritmi
- E' necessario cercare algoritmi con una complessità lineare (o vicina a quella lineare)

- Algoritmo Evoluto
- Deve il suo nome all'ideatore D.L. Shell
- Metodo basato sul concetto di «riduzione degli incrementi»
 - Basato sul confronto/scambio, ma non tra elementi adiacenti
 - Si confrontano tutti gli elementi che si trovano ad una distanza d e si continua riducendo il valore di d fino ad arrivare agli elementi adiacenti (d=1)

- Algoritmo Evoluto
- Deve il suo nome all'ideatore D.L. Shell
- Metodo basato sul concetto di «riduzione degli incrementi»
 - Basato sul confronto/scambio, ma non tra elementi adiacenti
 - Si confrontano tutti gli elementi che si trovano ad una distanza d e si continua riducendo il valore di d fino ad arrivare agli elementi adiacenti (d=1)
- E' una «variante» del Bubble Sort
 - Nel selection sort si confrontano solo elementi adiacenti!

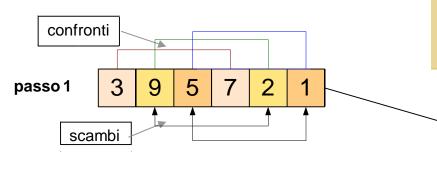


Si confrontano gli elementi distanti *d*, scambiandoli nel caso non siano ordinati.

Quindi vengono ordinati gli elementi diminuendo il valore di *d* fino ad arrivare ad 1.

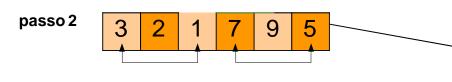
In questo esempio si sono scelti valori di d pari a: 3,2,1

Si confrontano gli elementi distanti tre posizioni scambiandoli nel caso il primo sia maggiore del secondo

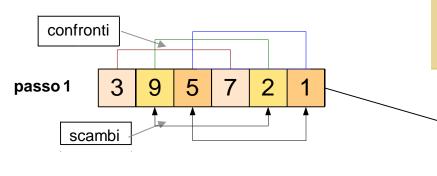


Si confrontano gli elementi distanti d, scambiandoli nel caso non siano ordinati.
Quindi vengono ordinati gli elementi diminuendo il valore di d fino ad arrivare ad 1. In questo esempio si sono scelti valori di d pari a: 3,2,1

Si confrontano gli elementi distanti tre posizioni scambiandoli nel caso il primo sia maggiore del secondo



Si confrontano gli elementi distanti 2 posizioni (3 - 1 - 9 e 2 - 7 - 5) riordinandoli (1 - 3 - 9 e 2 - 5 - 7)

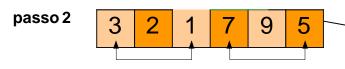


Si confrontano gli elementi distanti *d*, scambiandoli nel caso non siano ordinati.

Quindi vengono ordinati gli elementi diminuendo il valore di *d* fino ad arrivare ad 1.

In questo esempio si sono scelti valori di d pari a: 3,2,1

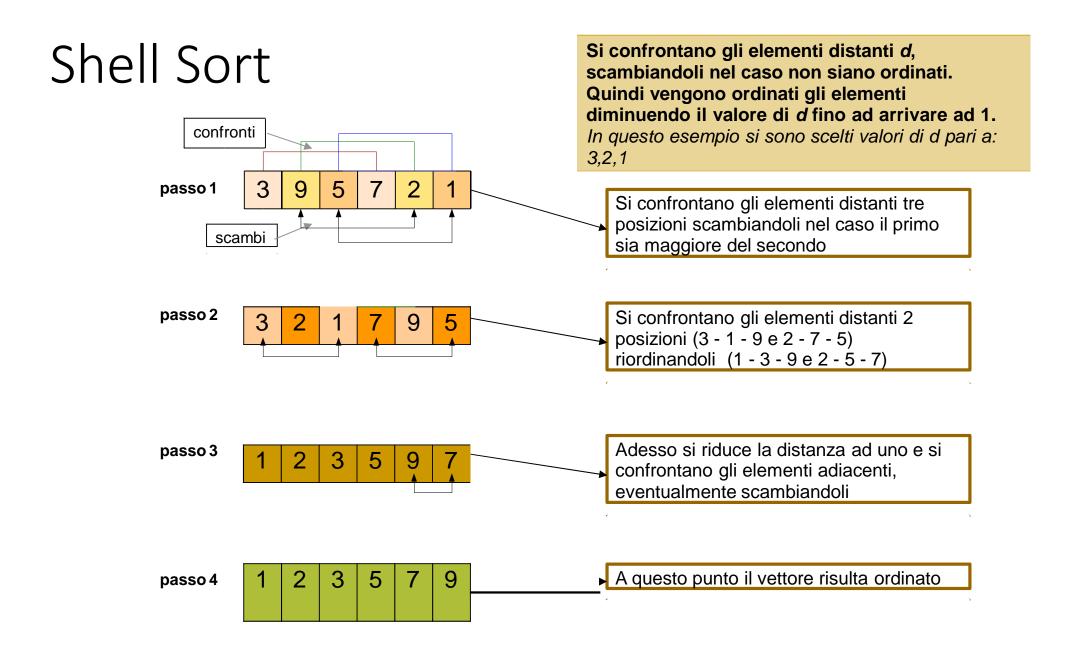
Si confrontano gli elementi distanti tre posizioni scambiandoli nel caso il primo sia maggiore del secondo



Si confrontano gli elementi distanti 2 posizioni (3 - 1 - 9 e 2 - 7 - 5) riordinandoli (1 - 3 - 9 e 2 - 5 - 7)



Adesso si riduce la distanza ad uno e si confrontano gli elementi adiacenti, eventualmente scambiandoli



Shell Sort - Considerazioni

- Problema
 - Come scegliere «d»?
 - Valutazione teorica molto complessa

Shell Sort - Considerazioni

- Problema
 - Come scegliere «d»?
 - Valutazione teorica molto complessa
- L'unico vincolo è rappresentato dall'ultimo passo, che deve avere distanza=1
 - Sequenza tipicamente utilizzata: 9, 5, 3, 2, 1
 - Importante: per avere problemi di efficienza evitare potenze di due (2, 4, 8, 16, etc.)

```
void ShellSort(int* vett, int dim) {
int i,j,gap,k;
int temp, a[5];
a[0]=9; a[1]=5; a[2]=3; a[3]=2; a[4]=1; // a = vettore dei gap
```

}}

}}

}}

}}

```
void ShellSort(int* vett, int dim) {
int i,j,gap,k;
int temp, a[5];
a[0]=9; a[1]=5; a[2]=3; a[3]=2; a[4]=1; // a = vettore dei gap
       for (k=0; k<5; k++){ // ciclo ripetuto per tutti i gap
              gap = a[k];
               for(i=gap; i<dim; i++) {</pre>
                      temp = vett[i];
                      for(j=i-gap;(temp < vett[j]) && (j>=0); j=j-gap) {
                              vett[j+gap] = vett[j];
                              vett[j] = temp; // scambio elementi
               }}
}}
```

```
void ShellSort(int* vett, int dim) {
int i,j,gap,k;
int temp, a[5];
a[0]=9; a[1]=5; a[2]=3; a[3]=2; a[4]=1; // a = vettore dei gap
       for (k=0; k<5; k++){ // ciclo ripetuto per tutti i gap
              gap = a[k];
               for(i=gap; i<dim; i++) {</pre>
                      temp = vett[i];
                      for(j=i-gap;(temp < vett[j]) && (j>=0); j=j-gap) {
                              vett[j+gap] = vett[j];
                              vett[j] = temp; // scambio elementi
               }}
}}
```

```
Al primo ciclo.
gap = 9; i = 9;
X = \text{vett}[9]; J = i-gap = 0
```

Si confronta x con vett[0], cioè vett[9] con vet[0], quindi il primo elemento con quello pari al gap.

Al ciclo successivo i valori si incrementano, quindi vett[1] con vett[10] (se esiste), e così via.

```
pari al gap.
void ShellSort(int* vett, int dim) {
                                                                 Al ciclo successivo i valori si incrementano, quindi
int i,j,gap,k;
                                                                 vett[1] con vett[10] (se esiste), e così via.
int temp, a[5];
a[0]=9; a[1]=5; a[2]=3; a[3]=2; a[4]=1; // a = vettore dei gap
        for (k=0; k<5; k++){ // ciclo ripetuto per tutti i gap
                gap = a[k];
                for(i=gap; i<dim; i++) {</pre>
                         temp = vett[i];
                         for(j=i-gap;(temp < vett[j]) && (j>=0); j=j-gap) {
                                 vett[j+gap] = vett[j];
                                                                                Se temp > vett[i]
                                 vett[j] = temp; // scambio elementi
                                                                                Non entra nel ciclo!
                 }}
                                                                                (Nessun scambio, in
}}
```

Al primo ciclo.

gap = 9; i = 9;

X = vett[9]; J = i-gap = 0

Si confronta x con vett[0], cioè vett[9] con vet[0], quindi il primo elemento con quello

tal caso!)

```
pari al gap.
void ShellSort(int* vett, int dim) {
int i,j,gap,k;
int temp, a[5];
a[0]=9; a[1]=5; a[2]=3; a[3]=2; a[4]=1; // a = vettore dei gap
       for (k=0; k<5; k++){ // ciclo ripetuto per tutti i gap
               gap = a[k];
               for(i=gap; i<dim; i++) {</pre>
                      temp = vett[i];
                      for(j=i-gap;(temp < vett[j]) && (j>=0); j=j-gap) {
                              vett[j+gap] = vett[j];
                              vett[j] = temp; // scambio elementi
               }}
}}
```

```
Al primo ciclo.
gap = 9; i = 9;
X = \text{vett}[9]; J = i-gap = 0
Si confronta x con vett[0], cioè vett[9] con
vet[0], quindi il primo elemento con quello
Al ciclo successivo i valori si incrementano, quindi
vett[1] con vett[10] (se esiste), e così via.
                   Serve ad effettuare più
                   di uno scambio a ciclo.
                   se necessario.
```

Shell Sort - Complessità

- Prestazioni «migliori» rispetto agli algoritmi «semplici»
- La complessità media è O(n log² n)
 - Dipende molto dalla distribuzione dei dati
 - La complessità è confermata anche nel caso peggiore, quindi tende ad avere prestazioni migliori.

Shell Sort - Complessità

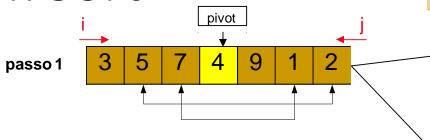
- Prestazioni «migliori» rispetto agli algoritmi «semplici»
- La complessità media è O(n log² n)
 - Dipende molto dalla distribuzione dei dati
 - La complessità è confermata anche nel caso peggiore, quindi tende ad avere prestazioni migliori.
- Intuitivamente: gli elementi vengono spostati più rapidamente utilizzando meno confronti quantomeno nella «zona» corretta

Algoritmo Ricorsivo

- Algoritmo Ricorsivo
 - Algoritmo che richiama sé stesso
 - Gli algoritmi ricorsivi sono più semplici ed eleganti, ma la loro esecuzione comporta, a causa della annidarsi della funzione che si richiama da se un uso a volte esagerato dello stack
- Complessità computazionale pari a O(n log n) nel caso ottimo e nel caso medio.

- · L'algoritmo è basato sul concetto di partizione
 - La procedura generale consiste nella selezione di un valore (detto **pivot**) e nella **suddivisione del vettore in due sezioni.**
 - Tutti gli elementi maggiori o uguali al valore del pivot andranno da una parte e tutti i valori minori dall'altra.
 - Questo processo viene ripetuto per ognuna delle sezioni rimanenti fino all'ordinamento dell'intero vettore.





Si seleziona l'elemento pivot e si spostano gli elementi più piccoli alla sua sinistra e i più grandi alla sua destra. Si suddivide, quindi, il vettore in due sezioni e si procede ricorsivamente

Si sceglie il quarto elemento (valore 4 come elemento pivot

Si seleziona, partendo dall'estremità sinistra, il primo elemento maggiore o uguale del pivot e, partendo dall'estremità destra, il primo elemento minore o uguale del pivot. Dopodiché si scambiano e si continua fino alla scansione di tutti gli elementi (i<=J)

passo 1

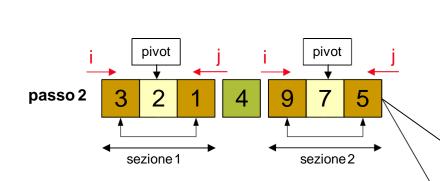
Si seleziona l'elemento pivot e si spostano gli elementi più piccoli alla sua sinistra e i più grandi alla sua destra. Si suddivide, quindi, il vettore in due sezioni e si procede ricorsivamente



Si seleziona, partendo dall'estremità sinistra, il primo elemento maggiore o uguale del pivot e, partendo dall'estremità destra, il primo elemento minore o uguale del pivot. Dopodiché si scambiano e si continua fino alla scansione di tutti gli elementi (i<=J)

Si divide il vettore in due sezioni e si procede ricorsivamente sulle due sezioni scegliendo gli elementi pivot

Si scorrono gli elementi delle due sezioni selezionando e scambiando gli elementi che risultano maggiori (da sinistra) e minori (da destra) dei rispettivi pivot



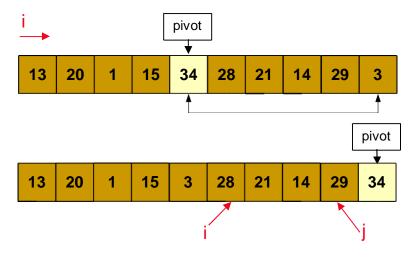
pivot

elementi più piccoli alla sua sinistra e i più grandi alla sua destra. Si suddivide, quindi, il vettore in due Quick Sort sezioni e si procede ricorsivamente pivot Si sceglie il quarto elemento (valore 4 come elemento pivot passo 1 Si seleziona, partendo dall'estremità sinistra, il primo elemento maggiore o uguale del pivot e, partendo dall'estremità destra, il primo elemento minore o uguale del pivot. Dopodiché si pivot scambiano e si continua fino alla scansione di tutti gli elementi (i<=J) passo 2 Si divide il vettore in due sezioni e si sezione 1 sezione 2 procede ricorsivamente sulle due sezioni scegliendo gli elementi pivot Si scorrono gli elementi delle due sezioni 3 5 4 passo 3 9 selezionando e scambiando gli elementi che risultano maggiori (da sinistra) e minori (da destra) dei rispettivi pivot A questo punto il vettore risulta ordinato

Si seleziona l'elemento pivot e si spostano gli

- Il comportamento generale del QuickSort è influenzato dalla scelta dell'elemento pivot
 - L'esempio precedente si riferisce al caso migliore che avviene quando la scelta dell'elemento pivot ricade sull'elemento mediano del vettore comportando una suddivisione del vettore in sezioni di pari dimensioni

- Il comportamento generale del QuickSort è influenzato dalla scelta dell'elemento pivot
 - L'esempio precedente si riferisce al caso migliore che avviene quando la scelta dell'elemento pivot ricade sull'elemento mediano del vettore comportando una suddivisione del vettore in sezioni di pari dimensioni
 - Il caso peggiore avviene quando il vettore viene decomposto in due sottovettori, di cui il primo ha dimensione uguale alla dimensione originaria meno 1, e l'altro ha una dimensione unitaria. Il pivot coincide con l'elemento massimo (o minimo) del vettore.
 - La scelta del pivot è determinante



L'esempio mette in evidenza il motivo di incrementare l'indice i fino a quando si trova un elemento più grande o uguale al pivot. Se non ci fosse la condizione uguale, nel nostro esempio l'indice i verrebbe

elemento più grande o uguale al pivot. Nel nostro esempio l'indice i si

centrale (indice m=(inf+sup)/2),

L'indice i viene incrementato fino a quando non viene trovato un

Se scegliamo come pivot l'elemento di posto

con l'elemento maggiore del vettore.

ovvero il 34, questo, casualmente, coincide

continuamente incrementato oltre la dimensione del vettore.

arresta in corrispondenza del pivot.

Per quanto riguarda l'indice **j** esso non viene spostato in quanto l'elemento **j**-esimo è inferiore al pivot.

Gli elementi di indice **i** e **j** vengono scambiati, e l'indice **i** viene incrementato, mentre **j** viene decrementato.

L'indice i viene quindi fatto incrementare fino a quando arriva all'elemento 34, che è pari al pivot. L'indice j non viene fatto decrementare perché si riferisce ad un elemento già inferiore al pivot.

20 1 15 3 28 21 14 29 34

sezioni sezione 1 sezione 2

Siccome i > j, la prima passata è finita: otteniamo le seguenti sezioni

13

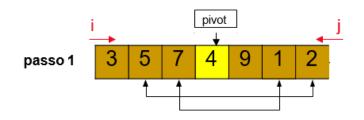
- Il comportamento generale del QuickSort è influenzato dalla scelta dell'elemento pivot
 - Se si fosse in grado di risalire all'elemento mediano la scelta ricadrebbe su di esso. Normalmente tale operazione comporta una conoscenza a priori sul vettore o un'analisi opportuna il cui tempo deve essere preso in considerazione
 - Necessario del tempo aggiuntivo per il calcolo della mediana → aumento della complessità
 - Il metodo più usato rimane quello della scelta casuale dell'elemento pivot, selezionando, ad esempio, l'elemento che occupa la posizione centrale

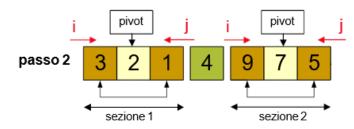
```
void quickSort(int vector[], int leftLimit, int rightLimit){
    int i;
    if(rightLimit <= leftLimit) return; // se i due estremi corrispondono, l'algoritmo è completo
    i = partition(vector,leftLimit,rightLimit);
    quickSort(vector,leftLimit,i-1); // richiama il quickSort ricorsivamente sulle due partizioni
    quickSort(vector,i+1,rightLimit);
}</pre>
```

```
void quickSort(int vector[], int leftLimit, int rightLimit){
    int i;
    if(rightLimit <= leftLimit) return; // se i due estremi corrispondono, l'algoritmo è completo
    i = partition(vector,leftLimit,rightLimit);
    quickSort(vector,leftLimit,i-1); // richiama il quickSort ricorsivamente sulle due partizioni
    quickSort(vector,i+1,rightLimit);
}</pre>
```

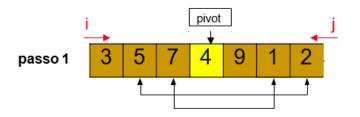
Al primo ciclo, le variabili leftLimit e rightLimit (left/right) sono i due estremi del vettore

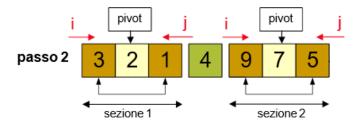
Quindi ricorsivamente l'algoritmo viene poi rilanciato sulle due sezioni, e così via finché le sezioni non avranno dimensione pari a 1 (uscita dalla funzione)





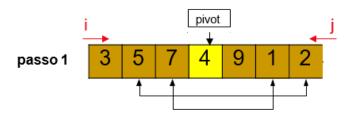
```
void quickSort(int vector[], int leftLimit, int rightLimit){
        int i;
        if(rightLimit <= leftLimit) return;</pre>
        i = partition(vector,leftLimit,rightLimit);
        quickSort(vector,leftLimit,i-1);
        quickSort(vector,i+1,rightLimit);
int partition(int v[], int l, int r) {
int x, i, j, temp, p = (1 + r) / 2;
x = v[p]; i=1-1; j=r+1
while (i < j)
        while(v[--i] > x);
        while(v[++i] < x);
        if (i < j) {
             temp = v[i];
             v[i] = v[i];
             v[j] = temp;
return j;
```

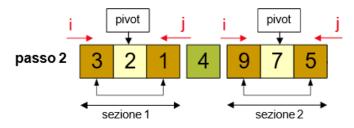




partition è invece la funzione che si occupa di individuare il pivot e di effettuare eventuali scambi per riordinare parzialmente le due partizioni.

```
void quickSort(int vector[], int leftLimit, int rightLimit){
        int i;
        if(rightLimit <= leftLimit) return;</pre>
        i = partition(vector,leftLimit,rightLimit);
        quickSort(vector,leftLimit,i-1);
        quickSort(vector,i+1,rightLimit);
int partition(int v[], int l, int r) {
int x, i, j, temp, p = (1 + r) / 2;
x = v[p]; i=1-1; j=r+1
while (i < j)
        while(v[--i] > x);
        while(v[++i] < x);
        if (i < j) {
             temp = v[i];
             v[i] = v[i];
             v[j] = temp;
return j;
```





I due cicli while()
cercano i due elementi
da scambiare. Una volta
individuati, si effettua
lo scambio

Quick Sort - Complessità

 Ad ogni passo il QuickSort confronta n elementi. La complessità è quindi determinata dalla «qualità» del partizionamento

Quick Sort - Complessità

- Ad ogni passo il QuickSort confronta n elementi. La complessità è quindi determinata dalla «qualità» del partizionamento
- **Nel caso migliore**, siccome l'array viene diviso in due ad ogni passo, e l'algoritmo deve comunque esaminare tutti gli n elementi, il tempo di esecuzione risulta **O(n log(n))**.

Quick Sort - Complessità

- Ad ogni passo il QuickSort confronta n elementi. La complessità è quindi determinata dalla «qualità» del partizionamento
- **Nel caso migliore**, siccome l'array viene diviso in due ad ogni passo, e l'algoritmo deve comunque esaminare tutti gli n elementi, il tempo di esecuzione risulta **O(n log(n))**.
- Nel caso peggiore ogni chiamata ricorsiva a Quicksort ridurrebbe solo di un'unità la dimensione dell'array da ordinare. Sarebbero quindi necessarie n chiamate ricorsive per effettuare l'ordinamento, portando a un tempo di esecuzione di $O(n^2)$.
 - Una soluzione a questo problema si può ottenere scegliendo a caso un elemento come pivot. Questo renderebbe estremamente improbabile il verificarsi del caso peggiore.

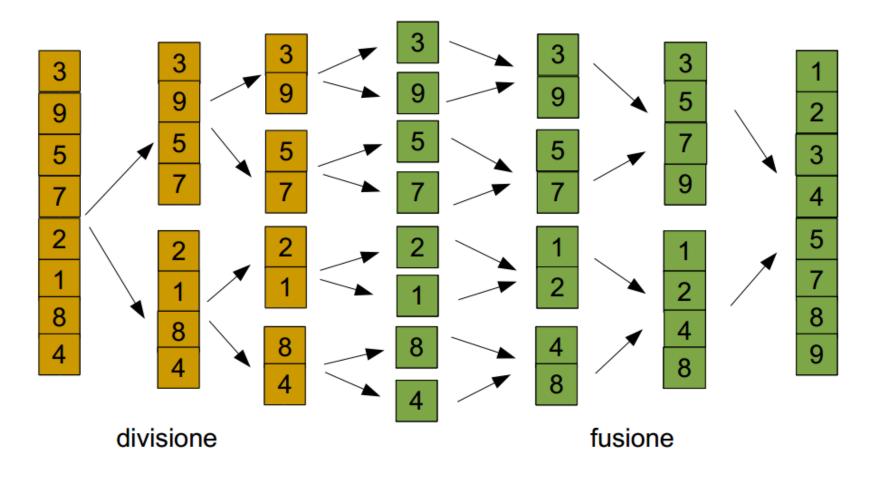
Merge Sort

- Si tratta di un algoritmo evoluto che ha complessità computazionale O(nlog(n)) anche nel caso peggiore
- È un algoritmo ricorsivo
 - Si basa sul principio "divide et impera" sfruttando il concetto di fusione (merging) di array ordinati

Merge Sort

- Si tratta di un algoritmo evoluto che ha complessità computazionale O(nlog(n)) anche nel caso peggiore
- È un algoritmo ricorsivo
 - Si basa sul principio "divide et impera" sfruttando il concetto di fusione (merging) di array ordinati
- Il Merge Sort utilizza uno spazio ausiliario proporzionale a N
- Le risorse di tempo e spazio impiegate dal Merge Sort non dipendono dall'ordinamento iniziale del file di input

Merge Sort - Idea



```
void mergeSort(int a[], int l, int r){
   if(r<=l)
      return;

int m=(r+l)/2;
   mergeSort(a,l,m);
   mergeSort(a,m+1,r);

merge(a,l,m,r);
}</pre>
```

```
void mergeSort(int a[], int 1, int r){
   if(r<=1)
      return; //ci fermiamo quando arriviamo a dimensione uno
   int m=(r+1)/2;
   mergeSort(a,1,m);
   mergeSort(a,m+1,r);

   merge(a,1,m,r);
}</pre>
```

```
void mergeSort(int a[], int l, int r){
   if(r<=l)
      return; //ci fermiamo quando arriviamo a dimensione uno

int m=(r+1)/2; // fisso l'elemento centrale
   mergeSort(a,l,m); // ripeto ricorsivamente sulle due partizioni
   mergeSort(a,m+1,r);

merge(a,l,m,r);
}</pre>
```

```
void mergeSort(int a[], int l, int r){
   if(r<=l)
      return; //ci fermiamo quando arriviamo a dimensione uno

int m=(r+l)/2; // fisso l'elemento centrale
   mergeSort(a,l,m); // ripeto ricorsivamente sulle due partizioni
   mergeSort(a,m+1,r);

merge(a,l,m,r);
}</pre>
```

Quando le chiamate ricorsive sono terminate, comincia la fase di merge

Merge Sort – Linguaggio C

```
void merge(int a[], int l, int m, int r){
    int i,j,k,*aux;
    aux=(int*)malloc((r-l+1)*sizeof(int));
     for(i=m+1;i>l;i--) {
        aux[i-1]=a[i-1];
                 for(j=m; j<r; j++) {
    aux[r+m-j]=a[j+1];</pre>
                                  for(k=l;k<=r;k++) {
    if(aux[j]<aux[i])
        a[k]=aux[j--];</pre>
                                            else
                                   a[k]=aux[i++];
        } }
```

Tale implementazione utilizza un array ausiliario di dimensione proporzionale all'output,

Per fare ciò il secondo array viene trascritto in maniera inversa alla fine del primo.

Merge Sort – Linguaggio C

```
void merge(int a[], int l, int m, int r){
    int i,j,k,*aux;
    aux=(int*)malloc((r-l+1)*sizeof(int));
     for(i=m+1;i>l;i--) {
        aux[i-1]=a[i-1];
                 for(j=m; j<r; j++) {
    aux[r+m-j]=a[j+1];</pre>
                                  for(k=l;k<=r;k++) {
    if(aux[j]<aux[i])
        a[k]=aux[j--];</pre>
                                            else
                                   a[k]=aux[i++];
        } }
```

r-l+1 = dimensione
del sottovettore

Tale implementazione utilizza un array ausiliario di dimensione proporzionale all'output,

Per fare ciò il secondo array viene trascritto in maniera inversa alla fine del primo.

Complessità Computazionale – Tabella Riepilogativa

Algoritmo	Caso Migliore	Caso Medio	Caso Peggiore
Selection Sort	O(n ²)	O(n²)	O(n ²)
Bubble Sort	O(n)	O(n²)	O(n ²)
Insertion Sort	O(n)	O(n²)	O(n ²)
Shell Sort	O(n log n)	O(n log² n)	O(n log ² n)
Quick Sort	O(n log n)	O(n log n)	O(n ²)
Merge Sort	O(n log n)	O(n log n)	O(n log n)

Esercizio 15.2 (a casa)

- Realizzare una funzione C che implementi un algoritmo di ordinamento avanzato a scelta, tra **Shell Sort, Quick Sort e Merge Sort**
- Richiamare la funzione in un main(), contenente un vettore di valori e stampare la sequenza di valori non ordinata e la sequenza di valori ordinata
- Testare il programma con una suite di test CUnit



