UNIVERSITÀ DEGLI STUDI DI BARI ALDO MORO

Corso di Laurea in Informatica *(Track B)* - A.A. 2018/2019

Laboratorio di Informatica

Algoritmi Fondamentali

(Parte 2)

docente: Veronica Rossano

veronica.rossano@uniba.it

Slides ispirate ai contenuti proposti **dal dott.Pasquale Lops. Grazie.**

Algoritmi di Ordinamento

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

Ordinamento

- 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

Ordinamento

- 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

Ordinamento

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

Ordinamento

- **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

Ordinamento

- **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 ← 1
finchè i < n
   trova il minimo valore nella lista (i … n)
   scambia la posizione del valore minimo con lista(i)
   i ← i + 1
```

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

Ordinamento per Selezione - Algoritmo

Pseudocodice

Perché effettuiamo esattamente ***n* cicli?**

```
i ← 1
finchè i < n
   trova il minimo valore nella lista (i … n)
   scambia la posizione del valore minimo con lista(i)
   i ← i + 1
```

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

Ordinamento per Selezione - Algoritmo

Pseudocodice

Perché effettuiamo esattamente ***n* cicli?**

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

```
i ← 1
finchè i < n
   trova il minimo valore nella lista (i … n)
   scambia la posizione del valore minimo con lista(i)
   i ← i + 1
```

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

Ordinamento per Selezione - Programma C

```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









        }
}
```


Ordinamento per Selezione - Programma C

```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
    }
}
```

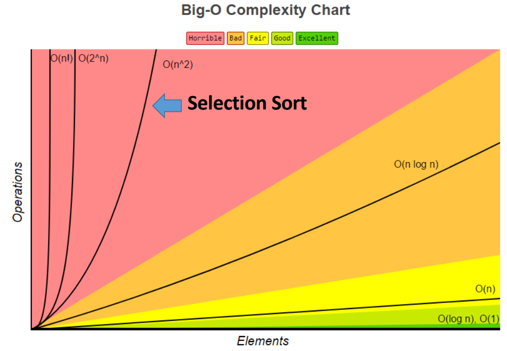

Ordinamento per Selezione - Complessità

- **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



Ordinamento per Selezione - Complessità

- **Confronti**
  - <u>**Sempre**</u> $(n-1) * n \rightarrow O(n^2)$
    - *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**
  - <u>**Sempre**</u> $(n-1) * n \rightarrow O(n^2)$
    - *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))

Ordinamento **a Bolle**

- 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*

Ordinamento a Bolle
Esempio

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.**

	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

Ordinamento a Bolle
Esempio

	II	II/1	II/2	II/3	III
array(1)	1	1	1	1	1
array(2)	4	4	4	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

Ordinamento a Bolle
Esempio

	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

Ordinamento a Bolle

- Quando **termina** l'algoritmo?

Ordinamento a Bolle

- Quando **termina** l'algoritmo?
  - Dipende dal numero di scambi che vengono effettuati.

Ordinamento a Bolle

- 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.**

Ordinamento a Bolle

- 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 ← 0 // passo di esecuzione
ordinato ← falso // indicatore di scambi
finchè p < n e non ordinato esegui  // esce dal ciclo dopo max n-1 passi
```

Ordinamento a Bolle - Algoritmo

```
p ← 0 // passo di esecuzione
ordinato ← falso // indicatore di scambi
finchè p < n e non ordinato esegui  // esce dal ciclo dopo max n-1 passi
  p ← p + 1
  ordinato ← vero
  i ← n      // memorizza l indice dell'elemento «in esame»
```


Ordinamento a Bolle - Algoritmo

```
p ← 0 // passo di esecuzione
ordinato ← falso // indicatore di scambi
finchè p < n e non ordinato esegui  // esce dal ciclo dopo max n-1 passi
  p ← p + 1
  ordinato ← vero
  i ← n      // memorizza l indice dell'elemento «in esame»
   finchè i > p esegui
       se lista(i) < lista (i - 1) allora
              scambia lista(i) con lista(i - 1)
              ordinato ← falso  // se effettuo uno scambio l'insieme
       i ← i - 1               // non è più ordinato
```


Ordinamento a Bolle - Algoritmo

Perché questa condizione?

```
p ← 0 // passo di esecuzione
ordinato ← falso // indicatore di scambi
finchè p < n e non ordinato esegui  // esce dal ciclo dopo max n-1 passi
  p ← p + 1
  ordinato ← vero
  i ← n      // memorizza l indice dell'elemento «in esame»
   finchè i > p esegui
       se lista(i) < lista (i - 1) allora
              scambia lista(i) con lista(i - 1)
              ordinato ← falso  // se effettuo uno scambio l'insieme
       i ← i - 1               // non è più ordinato
```


Ordinamento a Bolle - Algoritmo

Ad ogni passo noi abbiamo esattamente p elementi già ordinati (al primo passo abbiamo zero elementi già ordinati. *Dopo il primo passo avremo per certo l'elemento più piccolo in alto*).

Termina il ciclo quando la variabile *i* (che viene decrementata ad ogni passaggio) diventa uguale a ***p*, dunque quando abbiamo raggiunto la porzione di vettore che sappiamo essere ordinata.**

```
p ← 0 // passo di esecuzione
ordinato ← falso // indicatore di scambi
finchè p < n e non ordinato esegui  // esce dal ciclo do
  p ← p + 1
  ordinato ← vero
  i ← n      // memorizza l indice dell'elemento «in esame»
   finchè i > p esegui
       se lista(i) < lista (i - 1) allora
              scambia lista(i) con lista(i - 1)
              ordinato ← falso  // se effettuo uno scambio l'insieme
       i ← i - 1               // non è più ordinato
```

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--) {
                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 - Complessità

- **Caso migliore (lista già ordinata):**

Ordinamento a Bolle - Complessità

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

Ordinamento a Bolle - Complessità

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

Ordinamento a Bolle - Complessità

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

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)

Ordinamento per Inserzione

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

Ordinamento per Inserzione

- **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 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

Ordinamento per Inserzione

- 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**

Ordinamento per Inserzione

- 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 | →

| $\leq x$ | *x* | $> x$ | parte non ordinata | → x viene collocato nel punto corretto della parte ordinata

Ordinamento per Inserzione - Algoritmo

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


Ordinamento per Inserzione - Algoritmo

```
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
           fai scalare il precedente
```


Ordinamento per Inserzione - Algoritmo

```
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
           fai scalare il precedente
           se sei arrivato in prima posizione allora
                piazzalo; inserito ← vero
           fine_se
      altrimenti
           piazzalo; inserito ← vero
      fine_se
  fine_finchè
fine_perogni
```


Ordinamento per Inserzione – Implementazione C

```c
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
    j = i-1;               // memorizza l'indice dell'ultimo valore ordinato
    while (j>=0 && x[j]>app) {  // ciclo per individuare la posizione corretta
```

Ordinamento per Inserzione – Implementazione C

```
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
    j = 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;
}
```


Ordinamento per Inserzione – Implementazione C

```
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
    j = 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;
}
```


Ordinamento per Inserzione – Implementazione C

```
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
    j = 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;  // 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
```

Ordinamento per Inserzione - Complessità

- **Quanti passi?**

Ordinamento per Inserzione - Complessità

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

Ordinamento per Inserzione - Complessità

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

Ordinamento per Inserzione - Complessità

- **Quanti passi? Sempre *n* – 1**
  - Uno scambio per ogni confronto, salvo (eventualmente) l'ultimo
  - **Caso ottimo (lista già ordinata)**
    - $n - 1$ confronti, 0 scambi → 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 ($n \leq 25$)
  - 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
  - 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`

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)

Shell Sort

- **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)

Shell Sort

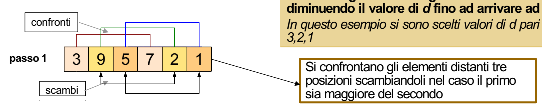
- **Algoritmo Evoluto**
- Deve il suo nome all'ideatore D.L. Shell

- 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!

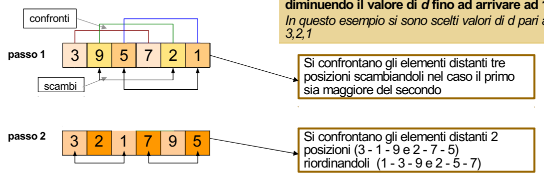
Shell Sort

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



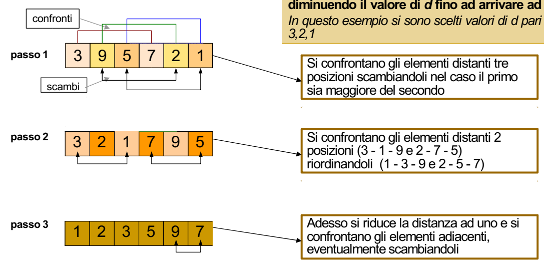
Shell Sort

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



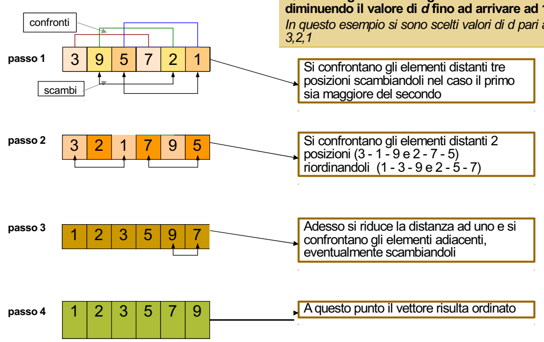
Shell Sort

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



Shell Sort

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



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.)

Shell Sort – Implementazione

```
void ShellSort(int* vett, int dim) {
int i,j,gap,k;
int x, 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++) {
                        x = vett[i];
                        for(j=i-gap;(x<vett[j]) && (j>=0); j=j-gap) {
                                vett[j+gap]=vett[j];
                                vett[j]=x;  // scambio elementi
                }}
}}
```


Shell Sort – Implementazione

Al primo ciclo.
gap = 9; i = 9;
X = vett[9]; J = i-gap = 0

Si confronta **x** con **vett[0]**, cioè **vett[9]** con **vett[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.

```
void ShellSort(int* vett, int dim) {
int i,j,gap,k;
int x, 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++) {
                        x = vett[i];
                        for(j=i-gap;(x<vett[j]) && (j>=0); j=j-gap) {
                                vett[j+gap]=vett[j];
                                vett[j]=x;  // scambio elementi
                }}
}}
```


Shell Sort – Implementazione

Al primo ciclo.
gap = 9; i = 9;
X = vett[9]; J = i-gap = 0

Si confronta **x** con **vett[0]**, cioè **vett[9]** con **vett[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.

```
void ShellSort(int* vett, int dim) {
int i,j,gap,k;
int x, 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++) {
                        x = vett[i];
                        for(j=i-gap;(x<vett[j]) && (j>=0); j=j-gap) {
                                vett[j+gap]=vett[j];
                                vett[j]=x;  // scambio elementi
                }}
}}
```

Serve ad effettuare più di uno scambio a ciclo, se necessario. **(Saltando gap posizioni per volta)**

Shell Sort - Complessità

- **Prestazioni «migliori» rispetto agli algoritmi «semplici»**
- **La complessità media** è O(n $\log^2$ 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 $\log^2$ 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**

Quick Sort

- **Algoritmo Ricorsivo**

Quick Sort

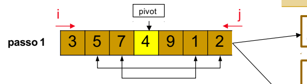
- **Algoritmo Ricorsivo**
  - **Algoritmo che richiama sé stesso**
  - Gli algoritmi ricorsivi sono più semplici ed eleganti, ma la loro esecuzione comporta, a causa della annidarsi della funzione che si richiama da se **un uso a volte esagerato dello stack**
- Complessità computazionale pari a `O(n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

Quick Sort

- **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.

Quick Sort

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

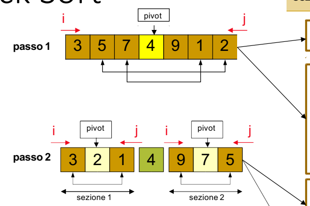


Si sceglie il quarto elemento (valore 4 come elemento pivot

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)

Quick Sort

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

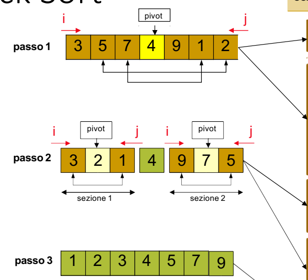
Si seleziona, partendo dall'estremità **sinistra**, il primo elemento maggiore o uguale del pivot e, partendo dall'estremità **destra**, il primo elemento minore o uguale del pivot. Dopodiché si scambiano e si continua fino alla scansione di tutti gli elementi (i<=J)

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

Quick Sort

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)

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

A questo punto il vettore risulta ordinato

Quick Sort - Considerazioni

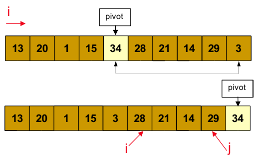
- **Il comportamento generale del QuickSort è influenzato dalla scelta dell'elemento pivot**
  - L'esempio precedente si riferisce al caso migliore che avviene quando la scelta dell'elemento pivot ricade sull'elemento mediano del vettore comportando una suddivisione del vettore in sezioni di pari dimensioni

Quick Sort - Considerazioni

- **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**

Quick Sort - Considerazioni

Se scegliamo come pivot l'elemento di posto centrale (indice m=(inf+sup)/2), **ovvero il 34, questo, casualmente, coincide con l'elemento maggiore del vettore.**



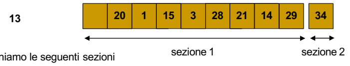
L'indice **i** viene incrementato **fino a quando non viene trovato un elemento più grande o uguale al pivot**. Nel nostro esempio l'indice **i** si arresta in corrispondenza del pivot.

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 continuamente incrementato oltre la dimensione del vettore.

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.



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

Quick Sort - Considerazioni

- **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

Quick Sort – Linguaggio C

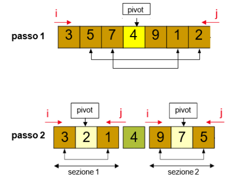
```
void quickSort(int v[], int l, int r){
        int i;
        if(r<=l) return;  // se i due estremi corrispondono, l'algoritmo è completo
        i=partition(v,l,r);
        quickSort(v,l,i-1); // richiama il quickSort ricorsivamente sulle due partizioni
        quickSort(v,i+1,r);
}
```


Quick Sort – Linguaggio C


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

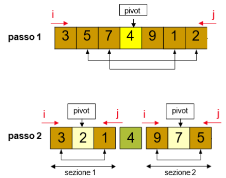

Al primo ciclo, le variabili `l` e `r` *(left/right)* sono i due estremi del vettore

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)*



Quick Sort – Linguaggio C

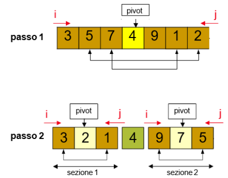
```
void quickSort(int v[], int l, int r){
        int i;
        if(r<=l) return;
        i=partition(v,l,r);
        quickSort(v,l,i-1);
        quickSort(v,i+1,r);
}
int partition(int v[], int l, int r) {
int x, i, j, temp, p = (l + r) / 2;
x = v[p]; i=l-1; j=r+1
while (i < j)    {
        while(v[--j]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

Quick Sort – Linguaggio C


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

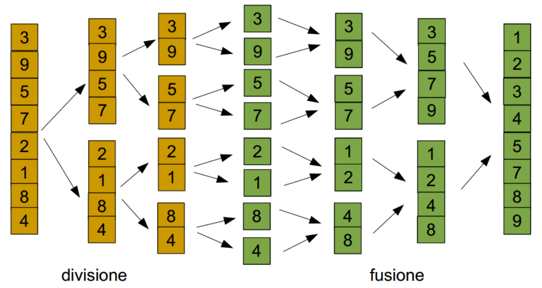
Merge Sort

- 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



Merge Sort – Linguaggio C (Parte 2)

```
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);
}
```


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];

                               for(k=l;k<=r;k++) {
                                       if(aux[j]<aux[i])
                                               a[k]=aux[j--];
                                       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.

Complessità Computazionale – Tabella Riepilogativa

Algoritmo	Caso Migliore	Caso Medio	Caso Peggiore
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Shell Sort	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$