Algoritmi di ordinamento (*sorting algorithms*)

Liceo G.B. Brocchi
Classi seconde Scientifico – opzione scienze applicate
Bassano del Grappa, Novembre 2022

Motivazioni

- Il nostro obiettivo è **ordinare** una struttura dati sulla base del valore di una chiave e del significato dell'operatore «confronto»:
  - ad esempio, per gli interi sappiamo che è definito il concetto di ordine: 5 < 6, 8 < 9 etc…
  - altro esempio: per le stringhe di caratteri di un alfabeto è definito l'ordinamento *lessicografico* (l'ordine con cui le parole sono presenti nel dizionario):
    - *aceto* < *acqua* < *birra* < *gingerino* < *grappa* < *vino* < *vodka*
  - per ora proveremo ad ordinare soltanto array di interi

- Questo problema ha innumerevoli applicazioni in informatica:
  - ordinare per poi cercare una chiave (pensate alla ricerca binaria)
  - produrre output ordinati da input disordinati
  - fare statistiche su insiemi di dati (pensare ad esempio al concetto di *mediana*)

Una provocazione: l'algoritmo *bogosort*

- Ci sono diversi modi (che si tradurranno in *algoritmi*) per ordinare un array di interi
- Alcuni metodi sono *intuitivi ma poco efficienti,* altri sono un po' meno intuitivi ma molto più efficienti
- **Bogosort** è un algoritmo che mostra come sia effettivamente possibile ordinare un array in un modo molto intuitivo, ma spaventosamente lento. Lo pseudocodice di questo algoritmo è il seguente:

```
int v[10] = {0, 4, 7, 2, 8, 6, 2, 3, 10, 9};
bool sorted = false;
while (sorted == false){
        if (is_sorted(v)){
                sorted = true;
        }
        shuffe(v);
}
```

Significa: *mescola gli elementi di v a caso finché v non è ordinato.*
Se l'array è già ordinato vi va anche bene *(best-case)*, altrimenti può finire veramente male, anche con array non troppo grandi

Una provocazione: l'algoritmo *bogosort*

- Eseguire **Bogosort** su un array equivale a ordinare un mazzo di carte lanciandolo in aria diverse volte finché non cade ordinato

- Ordinereste un mazzo di 3 carte in questo modo? → *perché no?*

- Ordinereste un mazzo di 4 carte in questo modo? → *perché no?*
- Ordinereste un mazzo di 7 carte in questo modo? → beh…

- Ordinereste un mazzo di 9 carte in questo modo? → anche no

- Ordinereste un mazzo di 500 000 carte in questo modo? → *esistono altri algoritmi di ordinamento?*

Questo algoritmo esegue un numero di operazioni proporzionale al fattoriale della dimensione dell'array. Quando sentite parlare di fattoriali e esponenziali in informatica dovete spaventarvi…

Insertion sort

[5, 2, 4, 6, 1, 3]

Idea: cicliamo su ogni elemento, verificando se è posizionato correttamente rispetto agli elementi precedenti (vogliamo ordinare l'array in senso crescente).
Se non è posizionato correttamente, lo spostiamo all'indietro fino alla posizione corretta.

Insertion sort

[5, 2, 4, 6, 1, 3]

L'elemento *0-esimo* è posizionato correttamente rispetto agli elementi precedenti?

Per saperlo, dobbiamo confrontarlo con gli elementi precedenti fintantoché questi sono maggiori dell'elemento stesso (di 5 in questo caso):

1. 5 < ?: non c'è un elemento precedente, quindi, per quel che ne sappiamo finora, è già al posto giusto

Insertion sort

[5, 2, 4, 6, 1, 3]

L'elemento *1-esimo* è posizionato correttamente rispetto agli elementi precedenti?
Per saperlo, dobbiamo confrontarlo con gli elementi precedenti, fintantoché questi sono maggiori dell'elemento stesso (di 2 in questo caso):

- 2 < 5, l'elemento in posizione **0**
- STOP: non si può più andare indietro

Quindi 2 deve essere spostato in posizione **0**. Tutti gli elementi di cui è minore vanno shiftati a destra di una posizione.

Risultato dell'iterazione corrente: **[2, 5, 4, 6, 1, 3]**

Insertion sort

[2, 5, 4, 6, 1, 3]

L'elemento 2-esimo è posizionato correttamente rispetto agli elementi precedenti?
Per saperlo, dobbiamo confrontarlo con gli elementi precedenti, fintantoché questi sono maggiori dell'elemento stesso (di 4 in questo caso):

- 4 < 5, l'elemento in posizione **1**
- 4 > 2, l'elemento in posizione 0
- STOP
- Quindi 2 deve essere spostato in posizione **1**. Tutti gli elementi di cui è minore vanno *shiftati* a destra di una posizione.

Risultato dell'iterazione corrente: **[2, 4, 5, 6, 1, 3]**

Insertion sort

[2, 4, 5, 6, 1, 3]

L’elemento 3-esimo è posizionato correttamente rispetto agli elementi precedenti?
Per saperlo, dobbiamo confrontarlo con gli elementi precedenti, fintantoché questi sono maggiori dell’elemento stesso (di 4 in questo caso):

- 6 > 5
- STOP
- Quindi 6 non deve essere spostato, era già al posto giusto.

Insertion sort

[2, 4, 5, 6, 1, 3]

L'elemento 4-esimo è posizionato correttamente rispetto agli elementi precedenti?
Per saperlo, dobbiamo confrontarlo con gli elementi precedenti, fintantoché questi sono maggiori dell'elemento stesso (di 4 in questo caso):

- 1 < 6, l'elemento in posizione 3
- 1 < 5, l'elemento in posizione 2
- 1 < 4, l'elemento in posizione 1
- 1 < 2, l'elemento in posizione **0**
- STOP
- Quindi 1 va spostato in posizione **0**. Tutti gli elementi di cui è minore vanno shiftati a destra di una posizione

Risultato dell'iterazione corrente: **[1, 2, 4, 5, 6, 3]**

Insertion sort

[1, 2, 4, 5, 6, 3]

L'elemento 5-esimo è posizionato correttamente rispetto agli elementi precedenti?
Per saperlo, dobbiamo confrontarlo con gli elementi precedenti, fintantoché questi sono maggiori dell'elemento stesso (di 4 in questo caso):

- 3 < 6, l'elemento in posizione 4
- 3 < 5, l'elemento in posizione 3
- 3 < 4, l'elemento in posizione **2**
- 3 > 2
- STOP
- Quindi 3 va spostato in posizione **2**. Gli elementi di cui è minore vanno shiftati a destra di una posizione.

Insertion sort

[1, 2, 3, 4, 5, 6]

Ecco il risultato finale. Meglio *insertion sort* o *bogosort*?

È abbastanza intuitivo come algoritmo, ma se non esistessero algoritmi «intelligenti» come *insertion sort*, gran parte delle conquiste dell'informatica e della scienza in generale non sarebbero state possibili.

Vedere https://en.wikipedia.org/wiki/Insertion_sort per un'animazione fatta benissimo.
Vi consiglio di fare una donazione a Wikipedia quando sarete più grandi.

Insertion sort

```
*: l'elemento in esame
_n_: gli elementi con cui viene confrontato l'elemento in esame

input: [1, 2, 5, 7, 8, 9, 10, 12, 13]

it.1: [1*, 2, 5, 7, 8, 9, 10, 12, 13] -> nessuna iterazione interna
it.2: [_1_, 2*, 5, 7, 8, 9, 10, 12, 13] -> nessuna iterazione interna
it.3: [1, _2_, 5*, 7, 8, 9, 10, 12, 13] -> nessuna iterazione interna
it.4: [1, 2, _5_, 7*, 8, 9, 10, 12, 13] -> nessuna iterazione interna
it.5: [1, 2, 5, _7_, 8*, 9, 10, 12, 13] -> nessuna iterazione interna
it.6: [1, 2, 5, 7, _8_, 9*, 10, 12, 13] -> nessuna iterazione interna
it.7: [1, 2, 5, 7, 8, _9_, 10*, 12, 13] -> nessuna iterazione interna
it.8: [1, 2, 5, 7, 8, 9, _10_, 12*, 13] -> nessuna iterazione interna
it.9: [1, 2, 5, 7, 8, 9, 10, _12_, 13*] -> nessuna iterazione interna

output: [1, 2, 5, 7, 8, 9, 10, 12, 13]
```

L'array era già ordinato. Chiaramente l'algoritmo non ha gli occhi, per cui non poteva vederlo fin da subito.
Quante operazioni base (confronti) ha effettuato *insertion sort* per ordinare un array già ordinato?

Insertion sort

- In caso di array già ordinato, siamo in presenza del cosiddetto *best-case* dell'algoritmo.

- Per ordinare un array già ordinato, *insertion sort* impiega lo stesso tempo che serve per *scannerizzare* l'array elemento per elemento

- Vediamo cosa succede se l'array è ordinato al contrario, ossia in senso decrescente. Intuitivamente dovrebbe andarci molto peggio.

Insertion sort

```
*: l'elemento in esame
_n_: gli elementi con cui viene confrontato l'elemento in esame

input: [15, 14, 12, 10, 9, 7, 6, 5]

it.1: [15*, 14, 12, 10, 9, 7, 6, 5]
  ->nessuna iterazione interna

it.2: [_15_, 14*, 12, 10, 9, 7, 6, 5]
  ->1 iterazione interna che porta 14 in posizione 0 e shifta a destra il 15

it.3: [_14_, _15_, 12*, 10, 9, 7, 6, 5]
  ->2 iterazioni interne che portano 12 in posizione 0 e shiftano a destra 14 e 15

it.4: [_12_, _14_, _15_, 10*, 9, 7, 6, 5]
  ->3 iterazioni interne che portano 10 in posizione 0 e shiftano a destra 12, 14 e 15
```

Insertion sort

```
it.5: [_10_, _12_, _14_, _15_, 9*, 7, 6, 5]
 --> 4 iterazioni interne che portano 9 in posizione 0 e shiftano a destra 10,12,14,15


it.6: [_9_, _10_, _12_, _14_, _15_, 7*, 6, 5]
 ->5 iterazioni interne che portano 7 in posizione 0 e shiftano a destra 9,10,12,14,15


it.7: [_7_, _9_, _10_, _12_, _14_, _15_, 6*, 5]
 ->6 iterazioni interne che portano 6 in posizione 0 e shiftano a destra 7,9,10,12,14,15


it.8: [_6_, _7_, _9_, _10_, _12_, _14_, _15_, 5*]
 ->7 iterazioni interne che portano 5 in posizione 0 e shiftano
    a destra   6,7,9,10,12,14,15
```

Insertion sort

- Visto quanto lavoro deve fare insertion sort quando l'array è ordinato al contrario?
- Per ogni elemento, deve tornare indietro e posizionarlo all'inizio
- Costa molto di più che scannerizzare l'array elemento per elemento
- Senza essere troppo formali, questo è il *worst-case* e richiede un tempo di esecuzione *quadratico* sulla dimensione dell'array. Effettivamente se sommate tutte le iterazioni fatte su un array di dimensione n, risulta
  - *0 + 1 + 2 + 3 + … + n – 1, che è uguale a (n-1)*n / 2, che è a sua volta uguale a:*
    - *(n^2 - n) / 2*

Drills su insertion sort

- Mostrare i passaggi eseguiti da insertion sort sui seguenti array:
  - [1, 6, 7, 3, 2, -1]
  - [5, 7, 8, 9, 10, 11]
  - [60, 50, -20, -30]
  - [45, 56, 30, 1, 5, 6, 9, 51]

Drills su insertion sort

- Mostrare i passaggi eseguiti da insertion sort sui seguenti array:
  - [1, 6, 7, 3, 2, -1]
  - [5, 7, 8, 9, 10, 11]
  - [60, 50, -20, -30]
  - [45, 56, 30, 1, 5, 6, 9, 51]

Insertion sort

```
const int size = 10;
int v[size] = {20, 25, 5, 80, 2, 56, 1, 30, -7, -96};

for (int i = 1; i < size; i++){  //scan each item
      int scanned_item = v[i];  //save current item
      int j = i - 1;    //start to check from previous item
      while (j >= 0 && scanned_item < v[j]){
            v[j + 1] = v[j];  //shift each element to the right
            j--;
      }
      v[j + 1] = item;  //put the current item in the correct position
}
```

NB: se *item* (l'elemento scannerizzato all'iterazione i-esima) è minore dell'elemento *v[j]*, e *j>=0,* allora viene fatta un'altra iterazione, visto che la condizione di permanenza è vera. Se si tratta dell'ultima volta in cui questa condizione è vera, significa che si è trovata la posizione giusta per *item,* che è la *j* corrente
Bisogna però ricordare che al termine dell'iterazione in questione viene eseguita l'istruzione *j--*, per cui quando si esce dal ciclo, la posizione giusta viene indicizzata da *j + 1*.
Lo spazio per *item* c'è, visto che gli elementi maggiori di item sono stati *shiftati* a destra di una posizione.

Insertion sort - correctness

- Ovviamente dal ciclo interno si esce sicuramente, ma vediamo un po' meglio in quali casi possiamo trovarci una volta usciti:
  - *j* diventa -1 e *item < v[0]*, quindi diventa falsa la condizione *j >= 0*. Significa che l'elemento corrente (i-esimo, salvato in *item*), è minore dell'elemento in posizione 0.
    Quindi è corretto posizionare *item* all'indice *0: j + 1: -1 + 1*
  - *item >= v[j] e j>=0*, quindi è diventa falsa la condizione *item < v[j]*. Significa che l'elemento corrente (*item*) è maggiore o uguale all'elemento in posizione *j*. Questo significa che item va posizionato appena dopo l'elemento in posizione *j*, quindi in posizione *j + 1*
  - Può anche darsi che non venga eseguita alcuna iterazione interna: in questo caso *j* non viene decrementato e resta uguale a *i – 1*. Per cui si esegue l'istruzione inutile v[j + 1] = v[i] = item. È inutile perché a *v[i]* viene assegnato il suo stesso valore immutato

Insertion sort – srotolamento del ciclo esterno

*40	9	98	98	147	27	133	60	22	63
*9	*40	98	98	147	27	133	60	22	63
*9	*40	*98	98	147	27	133	60	22	63
*9	*40	*98	*98	147	27	133	60	22	63
*9	*40	*98	*98	*147	27	133	60	22	63
*9	*27	*40	*98	*98	*147	133	60	22	63
*9	*27	*40	*98	*98	*133	*147	60	22	63
*9	*27	*40	*60	*98	*98	*133	*147	22	63
*9	*22	*27	*40	*60	*98	*98	*133	*147	63
*9	*22	*27	*40	*60	*63	*98	*98	*133	*147

Gli elementi «asteriscati» costituiscono la parte ordinata all'iterazione i-esima

Bubble sort

- Considerazioni intuitive:
  - un array di interi è ordinato se ogni coppia di elementi dell'array è sistemata correttamente, ossia se:
    - `array[i] <= array[i + 1]  per i >= 0 e i <= array_size – 1`

- Basta che anche una sola coppia non sia «aggiustata correttamente» per dire che l'array NON è ordinato

- Procediamo con degli esempi

Bubble sort

Questo array non è ordinato perché ci sono 4 coppie di elementi *non aggiustate*, cerchiate in rosso

Aggiustiamo le 4 coppie non aggiustate

Abbiamo ordinato magicamente l'array con 4 passaggi?

Analizzando le singole coppie una sola volta è impossibile ordinare tutto l'array. Non possiamo avere una visione d'insieme analizzando l'array coppia per coppia una sola volta. Ad esempio, non possiamo posizionare il 13 prima del 14 in una sola «passata», perché sono elementi «distanti», non appartenenti ad una coppia di elementi consecutivi

Bubble sort

Gli elementi con l'asterisco costituiscono la coppia in esame all'iterazione corrente

la coppia (array[0], array[1]) è scombinata, va effettuato uno *swap*

[6*, 2*, 7, 3, 10, 9, 11, 13, 8]

array di input

la coppia (array[1], array[2]) è ok, nessuno *swap*

[2, 6*, 7*, 3, 10, 9, 11, 13, 8]

la coppia (array[2], array[3]) è scombinata, va effettuato uno *swap*

[2, 6, 7*, 3*, 10, 9, 11, 13, 8]

Bubble sort

Gli elementi con l'asterisco costituiscono la coppia in esame all'iterazione corrente

la coppia (array[3], array[4]) è ok, nessuno *swap*

[2, 6, 3, 7*, 10*, 9, 11, 13, 8]

la coppia (array[4], array[5]) è scombinata, va effettuato uno *swap*

[2, 6, 3, 7, 10*, 9*, 11, 13, 8]

la coppia (array[5], array[6]) è ok, nessuno *swap*

[2, 6, 3, 7, 9, 10*, 11*, 13, 8]

Bubble sort

Gli elementi con l'asterisco costituiscono la coppia in esame all'iterazione corrente

la coppia (array[6], array[7]) è ok, nessuno *swap*

[2, 6, 3, 7, 9, 10, 11*, 13*, 8]

la coppia (array[7], array[8]) è scombinata, va effettuato uno *swap*

[2, 6, 3, 7, 9, 10, 11, 13*, 8*]

[2, 6, 3, 7, 9, 10, 11, 8, 13]

Output della prima *passata*.
Vi sembra ordinato?

Bubble sort

- L'output della *passata* precedente è l'input di un'altra passata
- Per ora non poniamoci il problema di quante passate servono
- **NB**: *passata* è un termine informale che indica una «scannerizzazione» dell'array coppia per coppia, con eventuali scambi *(swap)*
- Fra un po' capiremo anche da dove salta fuori le *bubble* (bolle)

Bubble sort

Gli elementi con l'asterisco costituiscono la coppia in esame all'iterazione corrente

la coppia (array[0], array[1]) è ok, nessuno *swap*

[2*, 6*, 3, 7, 9, 10, 11, 8, 13]

la coppia (array[1], array[2]) è scombinata, va effettuato uno *swap*

[2, 6*, 3*, 7, 9, 10, 11, 13*, 8*]

la coppia (array[2], array[3]) è ok, nessuno *swap*

[2, 3, 6*, 7*, 9, 10, 11, 8, 13]

Bubble sort

Gli elementi con l'asterisco costituiscono la coppia in esame all'iterazione corrente

la coppia (array[3], array[4]) è ok, nessuno *swap*

[2, 3, 6, 7*, 9*, 10, 11, 8, 13]

la coppia (array[4], array[5]) è scombinata, nessuno *swap*

[2, 3, 6, 7, 9*, 10*, 11, 8, 13]

la coppia (array[5], array[6]) è ok, nessuno *swap*

[2, 3, 6, 7, 9, 10*, 11*, 8, 13]

Bubble sort

Gli elementi con l'asterisco costituiscono la coppia in esame all'iterazione corrente

la coppia (array[6], array[7]) è scombinata

[2, 3, 6, 7, 9, 10, 11*, 8*, 13]

la coppia (array[4], array[5]) è scombinata, nessuno *swap*

[2, 3, 6, 7, 9, 10, 8, 11, 13]

la coppia (array[7], array[8]) è ok, nessuno *swap*

[2, 3, 6, 7, 9, 10, 8, 11, 13]

Output della seconda *passata*.
Vi sembra ordinato?

Bubble sort

- Ecco tutte le passate stampate su stdout da un codice C++ che realizza l'algoritmo (*----* indica una coppia scombinata):

```
*6------*2      *7------*3      *10------*9     11      *13------*8
2       *6------*3      7       9       10      *11------*8     13
2       3       6       7       9       *10------*8     11      13
2       3       6       7       *9------*8      10      11      13
2       3       6       7       8       9       10      11      13
```

Bubble sort

- Ecco tutte le passate partendo da un array ordinato al contrario (spiegare perché alcuni elementi sono preceduti da due caratteri '*'):

```
*9------**8------**7------**6------**5------**4------**3------**2------**1------*0

*8------**7------**6------**5------**4------**3------**2------**1------*0       9

*7------**6------**5------**4------**3------**2------**1------*0        8       9

*6------**5------**4------**3------**2------**1------*0 7       8       9

*5------**4------**3------**2------**1------*0  6       7       8       9

*4------**3------**2------**1------*0   5       6       7       8       9

*3------**2------**1------*0    4       5       6       7       8       9

*2------**1------*0     3       4       5       6       7       8       9

*1------*0      2       3       4       5       6       7       8       9

0       1       2       3       4       5       6       7       8       9
```

Bubble sort

- Ecco tutte le passate partendo da un array ordinato al contrario (spiegare perché alcuni elementi sono preceduti da due caratteri '*'):

```
*9------**8------**7------**6------**5------**4------**3------**2------**1------*0

*8------**7------**6------**5------**4------**3------**2------**1------*0       9

*7------**6------**5------**4------**3------**2------**1------*0        8       9

*6------**5------**4------**3------**2------**1------*0 7       8       9

*5------**4------**3------**2------**1------*0  6       7       8       9

*4------**3------**2------**1------*0   5       6       7       8       9

*3------**2------**1------*0    4       5       6       7       8       9

*2------**1------*0     3       4       5       6       7       8       9

*1------*0      2       3       4       5       6       7       8       9

0       1       2       3       4       5       6       7       8       9
```

Bubble sort

- Ecco tutte le passate partendo da un array già ordinato:

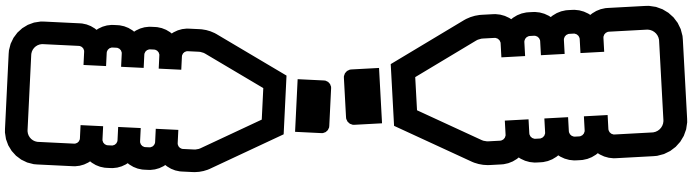
```
5       6       10      20      25      30      40      55      60      102
```

Bubble sort

- Ci serve un mattoncino fondamentale per realizzare l'algoritmo

- Come si scambia il contenuto di due variabili, ossia, come si effettuano i famosi *swap?*

- Avete mai scambiato il contenuto di due bottiglie così?



Bubble sort – versione lievemente *naive* (ingenua)

```
bool is_sorted = false;

while (!is_sorted){
        is_sorted = true;  //suppose there won't be any swap, i.e. has been sorted
        for (int i = 0; i <= size - 2; i++){
                if (v[i] > v[i + 1]){
                        int t = v[i];
                        v[i] = v[i + 1];
                        v[i + 1] = t;
                        is_sorted = false;
                }
        }
}
```

Bubble sort – srotolamento

```
22      *148------*60   *145------*97   *116------*3    38      *93------*47
22      60      *145------*97   *116------*3    38      *93------*47    148
22      60      97      *116------*3    38      *93------*47    145     148
22      60      *97------*3     38      *93------*47    116     145     148
22      *60------*3     38      *93------*47    97      116     145     148
*22------*3     38      *60------*47    93      97      116     145     148
3       22      38      47      60      93      97      116     145     148
```

Bubble sort – versione ottimizzata

- Premessa: per semplicità, numeriamo le iterazioni partendo da **0**:
  - quindi, chiamiamo la prima iterazione ***iterazione 0-esima***
- Sappiamo che *bubble sort* lavora per diverse «passate» sull'array
- Notiamo che alla passata 0, viene posizionato correttamente, cioè all'indice *size – 1*, lo 0-esimo elemento massimo dell'array (ossia il più grande). Nell'esempio l'elemento massimo è indicato con un asterisco:
  - `[5, 67*, 45, 34, 24]` → `[5, 45, 67, 34, 24]` → `[5, 45, 34, 67, 24]` →
    `[5, 45, 34, 24, 67*]`

  Se **m** è l'elemento massimo di un array, allora in ogni confronto (**m**, y), **m** andrà sempre a destra. Andando sempre più a destra si posiziona all'indice *size-1*, che è quello corretto per l'elemento massimo.

Bubble sort – correctness e versione ottimizzata

- Premessa: per semplicità, numeriamo le iterazioni partendo da **0**:
  - quindi, chiamiamo la seconda iterazione ***iterazione 1-esima***
- Notiamo che alla passata/iterazione 1-esima, viene posizionato correttamente, cioè all'indice *size – 2*, il 1-esimo elemento massimo dell'array (ossia il secondo più grande). Nell'esempio l'elemento massimo è indicato con un asterisco:
  - `[5, 45*, 34, 24, 67]` → `[5, 34, 45*, 24, 67]` → `[5, 34, 24, 45*, 67]`

  Se $\mathbf{m_1}$ è l'*1-esimo* elemento massimo di un array, allora in ogni confronto ($\mathbf{m_1}$, y), $\mathbf{m_1}$ andrà sempre a destra. Andando sempre più a destra si posiziona all'indice *size–2*, per il 1-esimo elemento massimo, visto che non può «sorpassare» lo 0-esimo elemento massimo.

Bubble sort – correctness e versione ottimizzata

- Il discorso precedente vale per tutti gli elementi

- **Iterazione 0: viene posizionato correttamente lo 0-esimo elemento più grande in posizione size – 1 - 0**
- **Iterazione 1: viene posizionato correttamente il 1-esimo elemento più grande in posizione size – 1 – 1**
- **Iterazione 2: viene posizionato correttamente il 2-esimo elemento più grande in posizione size – 1 – 2**
- **Iterazione 3: viene posizionato correttamente il 3-esimo elemento più grande in posizione size – 1 – 3**
- **……..**
- **Iterazione size – 1: viene posizionato correttamente il size-1 elemento più grande (cioè il più piccolo) in posizione size – 1 – (size – 1) = 0**

Bubble sort – correctness e versione ottimizzata

- Quindi, se all'iterazione i-esima si «sistema» l'elemento giusto in posizione size – 1 – i
- All'iterazione (i + 1)-esima ha senso controllare se gli elementi nelle posizioni successive sono ordinati? No, sono ordinati per la prova precedente

- In pratica, meno formalmente e senza utilizzare indici scomodi:
  - alla prima passata il ciclo interno deve controllare e eventualmente sistemare le coppie fino all'ultima coppia
  - alla seconda passata il ciclo interno deve controllare e eventualmente sistema le coppie fino alla penultima coppia
  - alla passata n-esima (dove n è la dimensione dell'array) il ciclo interno interno deve controllare e eventualmente sistemare solo la prima coppia

- A questo punto è più comodo indicizzare il ciclo esterno (la passata) partendo da *size-1*

Bubble sort – correctness e versione ottimizzata

```
/*
  [5, 8, 45, 35, 1]
  Result of first scan, inner loop until index 3:
    [5, 8, 35, 1, 45*]
  Result of second scan, inner loop until index 2:
    [5, 8, 1, 35*, 45*]
  Result of third scan, inner loop until index 1:
    [5, 1, 8*, 35*, 45*]
  Result of fourth scan, inner loop until index 0:
    [1, 5*, 8*, 35*, 45*]
*/
```

Bubble sort – correctness e versione ottimizzata

```
bool is_sorted = false;

for (int i = size - 1; i > 0 && !is_sorted; i--){
        is_sorted = true;
        for (int j = 0; j < i; j++){
                if (v[j] > v[j + 1]){
                        //swap
                        int t = v[j];
                        v[j] = v[j + 1];
                        v[j + 1] = t;
                        is_sorted = false;
                }
        }
}
```

se ad una certa iterazione non avvengono più swap non ha senso fare un'altra passata, per questo la condizione && swap_happened

Benchmarking

```
srand(time(NULL));
const int size = 10000;
int array[size];
int array_copy[size];

for (int i = 0; i < size; i++){
                array[i] = rand() % 150;
                array_copy[i] = array[i];
}

/*print_iterative(array, size);
bubble_sort(array, size);
print_iterative(array, size);*/

auto t0 = chrono::high_resolution_clock::now();
insertion_sort(array, size);
auto t1 = chrono::high_resolution_clock::now();
cout << "Insertion sort:  " << chrono::duration_cast<chrono::nanoseconds>(t1 - t0).count() << " nanoseconds passed\n";

auto t2 = chrono::high_resolution_clock::now();
bubble_sort(array_copy, size);
auto t3 = chrono::high_resolution_clock::now();
cout << "Bubble sort:    " << chrono::duration_cast<chrono::nanoseconds>(t3 - t2).count() << " nanoseconds passed\n";
```

```
Insertion sort:  50713300 nanoseconds passed
Bubble sort:    195993500 nanoseconds passed
```