Esercitazione 8 - 16/11

Indice degli esercizi

Esercizio	Argomento
Esercizio 1	Inversione di un array
Esercizio 2	Ogni 2 è seguito da un 3
Esercizio 3	Riempire un array con numeri casuali
Esercizio 4	Riempire un array con numeri chiesti all'utente
Esercizio 5	Copiare un array: perché non basta `b = a`

Esercizio 1

Ripasso: Recap 1 · Gli array (indici, `length`, ciclo `for`) e Recap 2 · Array e funzioni (regola del nuovo array).

Dato un array `v`, scrivere un programma che costruisca un **nuovo array** `risultato` (senza modificare `v`) e lo stampi.

A. `risultato` contiene gli stessi elementi di `v` in ordine **inverso**. Ad esempio con `v = [1, 2, 3, 4]` si stampa `[4, 3, 2, 1]`.

B. In `risultato` gli elementi che si trovano in **posizione pari** (indici `0, 2, 4, ...`) sono presi in ordine inverso, mentre quelli in posizione dispari **restano dove sono**. Ad esempio:

- con `v = [1, 2, 3, 4, 5]` si stampa `[5, 2, 3, 4, 1]`
- con `v = [10, 11, 12, 13, 14, 15]` si stampa `[14, 11, 12, 13, 10, 15]`

Provate i programmi anche con un array di un solo elemento e con l'array vuoto.

▼ **Soluzione**

A lezione sono usciti tanti modi diversi di ragionare, tutti corretti: quale viene più naturale dipende da come si imposta il problema all'inizio. Qui li trovate nell'ordine in cui li abbiamo visti, con gli errori che abbiamo incontrato lungo la strada.

A · Invertire un array

Prima di tutto: come si crea `risultato`? Ci sono due modi.

```
let v = [1, 2, 3, 4];

let r1 = [];                                  // vuoto e lungo 0
let r2 = new Array(v.length);                 // vuoto ma già lungo 4

console.log(r1, r1.length);                   // [] 0
console.log(r2, r2.length);                   // [ <4 empty items> ] 4
```

Con `new Array(v.length)` le «celle» ci sono già e basta scriverci dentro; con `[]` le celle vanno create man mano. Sappiamo già che `risultato` avrà tanti elementi quanti `v`, quindi la seconda forma è la più naturale per questo esercizio.

A1 · Due indici che si muovono in direzioni opposte. Si scorre `v` **dalla fine** con `i` (da `v.length - 1` fino a `0`, con `i--`) e, intanto, si scrive in `risultato` **dall'inizio** con un secondo indice `j`:

```
let v = [1, 2, 3, 4];

let risultato = new Array(v.length);
let j = 0;                                    // dove scrivere in risultato
for (let i = v.length - 1; i >= 0; i--) {     // dove leggere in v
  risultato[j] = v[i];
  j++;
}

console.log(risultato);                       // [ 4, 3, 2, 1 ]
console.log(v);                               // [ 1, 2, 3, 4 ]: v non è cambiato
```

Si legge il `4` e lo si scrive in posizione `0`, poi si legge il `3` e lo si scrive in posizione `1`, e così via. `j` non è un array: è solo il numero della cella di `risultato` in cui scrivere. Per vedere passo passo come cambiano `i`, `j` e le celle, provate il codice su Python Tutor.

A2 · Un solo indice e una formula. In A1 le due variabili sono legate: ogni volta che `i` scende di uno, `j` sale di uno, e vale sempre `i + j == v.length - 1`. Allora una delle due si può ricavare dall'altra. Si scorre `risultato` dall'inizio e si legge da `v` in posizione `v.length - 1 - i`:

```
let v = [1, 2, 3, 4];

let risultato = new Array(v.length);
for (let i = 0; i < v.length; i++) {
  risultato[i] = v[v.length - 1 - i];
}

console.log(risultato);                          // [ 4, 3, 2, 1 ]
```

Con `i = 0` si legge `v[3]` (l'ultimo), con `i = 1` si legge `v[2]` (il penultimo), con `i = 3` si legge `v[0]`. Con una variabile in meno; A1 però si legge forse più facilmente.

A3 · Partire da `[]` e aggiungere in fondo. Se `risultato` nasce vuoto e lungo `0`, si può sfruttare una proprietà di JavaScript: **scrivendo** in una posizione che non esiste, l'array si allunga da solo. La prima posizione libera di un array è sempre `risultato.length`:

```
let v = [1, 2, 3, 4];

let risultato = [];
for (let i = v.length - 1; i >= 0; i--) {
  risultato[risultato.length] = v[i];            // scrive nella prima cella libera
}

console.log(risultato);                          // [ 4, 3, 2, 1 ]
```

Il ruolo di `j` lo fa `risultato.length`, che vale `0`, poi `1`, poi `2`... perché cresce di uno a ogni elemento aggiunto.

Due osservazioni:

- qui bisogna partire da `[]`: con `new Array(v.length)` la lunghezza sarebbe già `4`, la «prima cella libera» sarebbe la `4` e si otterrebbe `[ <4 empty items>, 4, 3, 2, 1 ]`;
- scrivere oltre la fine di un array è una comodità di **JavaScript**, non della programmazione in generale: altri linguaggi (per esempio Python) danno errore. Non è sbagliato usarla, ma conviene sapere che si sta facendo un'operazione un po' particolare. Creare l'array già della lunghezza giusta, come in A1 e A2, funziona invece in qualunque linguaggio.

B · Invertire solo le posizioni pari

Con `v = [7, -1, 33, 9, 0, 5]` le posizioni dispari (`1`, `3`, `5`) restano uguali, mentre i valori in posizione pari (`7`, `33`, `0`) vanno messi al contrario:

```
posizione:    0    1    2    3    4    5
v:          [ 7,  -1,  33,   9,   0,   5 ]
risultato:  [ 0,  -1,  33,   9,   7,   5 ]
```

La strategia: spezzare il problema. L'esercizio è fatto di due pezzi: **copiare** le posizioni dispari (facile) e **invertire** le posizioni pari (più difficile, ma è il punto A). Non serve scrivere un unico ciclo che fa tutto in un colpo: si risolve prima il pezzo facile, ci si assicura che funzioni, e poi ci si dedica all'altro.

B1 · Due cicli e due indici. È il ragionamento di A1 applicato ai due pezzi.

```
let v = [7, −1, 33, 9, 0, 5];

let risultato = new Array(v.length);

// pezzo facile: le posizioni dispari si copiano uguali
let j = 0;
for (let i = 0; i <= v.length − 1; i++) {
  if (i % 2 != 0) {
    risultato[j] = v[i];
  }
  j++;                                          // FUORI dall'if: j va di pari passo con i
}

// pezzo difficile: le posizioni pari al contrario
j = 0;                                          // si riparte dall'inizio di risultato
for (let i = v.length − 1; i >= 0; i−−) {       // v letto dalla fine, come in A1
  if (i % 2 == 0) {
    risultato[j] = v[i];
    j += 2;                                     // DENTRO l'if: solo dopo aver scritto
  }
}

console.log(risultato);                         // [ 0, −1, 33, 9, 7, 5 ]
```

Il punto delicato è **dove** si aggiorna `j`, ed è diverso nei due cicli:

- nel primo ciclo `j` deve avanzare **sempre**, insieme a `i`; in pratica vale sempre quanto `i`, quindi si potrebbe scrivere direttamente `risultato[i] = v[i]`;
- nel secondo ciclo `j` deve **aspettare**: parte da `0` e resta lì finché non si trova il primo elemento in posizione pari; dopo averlo scritto salta di **due**, alla posizione pari successiva di `risultato`.

Con questo ragionamento non bisogna chiedersi se la lunghezza di `v` è pari o dispari: leggendo `v` dalla fine, il primo elemento in posizione pari che si incontra è sempre l'ultimo, e finisce in posizione `0`.

B2 · Senza `j`: riusare la formula di A2. Si parte di nuovo dai due pezzi: un ciclo copia i dispari e un altro usa la formula di A2, ma solo sulle posizioni pari.

```
let risultato = new Array(v.length);
for (let i = 0; i < v.length; i++) {
  if (i % 2 != 0) {
    risultato[i] = v[i];
  }
}
for (let i = 0; i < v.length; i++) {
  if (i % 2 == 0) {
    risultato[i] = v[v.length - 1 - i];
  }
}
```

Rileggendo il codice ci si accorge che i due `for` sono **identici**: scorrono l'array nello stesso verso. Allora si possono fondere in uno solo, e i due `if` diventano un `if` / `else`, perché una posizione o è pari o è dispari:

```
let risultato = new Array(v.length);
for (let i = 0; i < v.length; i++) {
  if (i % 2 != 0) {
    risultato[i] = v[i];
  } else {
    risultato[i] = v[v.length - 1 - i];
  }
}
```

Anche lasciare due cicli andava benissimo: i programmi si scrivono a pezzi e si migliorano dopo. In B1 invece i due cicli non si potevano fondere, perché scorrono `v` in versi opposti.

Il problema della lunghezza pari. Con `v = [7, -1, 33, 9, 0]` questo codice funziona, ma con `v = [7, -1, 33, 9, 0, 5]` stampa `[ 5, -1, 9, 9, -1, 5 ]`: nelle posizioni pari sono finiti elementi che stavano in posizione **dispari**. Il motivo è che lo specchio parte da `v.length - 1`, che è una posizione pari solo se la lunghezza è dispari. Lo specchio va fatto **tra le sole posizioni pari**, quindi deve partire dall'**ultima posizione pari**:

`v.length`	**ultima posizione**	**ultima posizione pari**
dispari (es. `5`)	`4`, pari	`4`, cioè `length - 1`
pari (es. `6`)	`5`, dispari	`4`, cioè `length - 2`

La correzione «a mano» (che non basta). Scrivere `v[v.length - 2 - i]` sistema l'array lungo `6`, ma rompe quello lungo `5`: con `[7, -1, 33, 9, 0]` si ottiene `[ 9, -1, -1, 9, undefined ]`. Una soluzione che funziona solo perché l'input di prova è fatto in un certo modo non è una soluzione: vogliamo un codice che vada bene per **qualunque** array. Le due correzioni che seguono lo fanno.

B3 · Calcolare l'ultima posizione pari con un `if`. Si controlla prima se la lunghezza è pari o dispari, come nella tabella:

```
let v = [7, -1, 33, 9, 0, 5];

let ultimoPari;
if (v.length % 2 == 1) {
  ultimoPari = v.length - 1;
} else {
  ultimoPari = v.length - 2;
}

let risultato = new Array(v.length);
for (let i = 0; i < v.length; i++) {
  if (i % 2 != 0) {
    risultato[i] = v[i];                      // le posizioni dispari non si muovono
  } else {
    risultato[i] = v[ultimoPari - i];         // specchio tra le posizioni pari
  }
}

console.log(risultato);                       // [ 0, -1, 33, 9, 7, 5 ]
```

`ultimoPari` vale `4`: in posizione `0` va `v[4]`, in posizione `2` va `v[2]` (resta lì) e in posizione `4` va `v[0]`.

B4 · Un fattore calcolato con il modulo. Nella tabella i due casi differiscono solo per un `1` in più da togliere quando la lunghezza è pari, e quel numero si può **calcolare** invece di scegliere con un `if`: `1 - v.length % 2` vale `0` se la lunghezza è dispari e `1` se è pari.

```
let v = [7, -1, 33, 9, 0, 5];

let fattore = 1 - v.length % 2;               // 0 se length è dispari, 1 se è pari

let risultato = new Array(v.length);
for (let i = 0; i < v.length; i++) {
  if (i % 2 != 0) {
    risultato[i] = v[i];
  } else {
    risultato[i] = v[v.length - 1 - fattore - i];
  }
}

console.log(risultato);                       // [ 0, -1, 33, 9, 7, 5 ]
```

È la stessa soluzione di B3, perché `v.length - 1 - fattore` è proprio `ultimoPari`; cambia solo il modo di ottenerlo. B3 si legge più facilmente, B4 è più compatta: questo tipo di «raffinatezza» viene col tempo, quando il ragionamento di base è chiaro.

B5 · Ricondursi al punto A con un array di appoggio (non l'abbiamo vista a lezione). Si mettono da parte i soli elementi in posizione pari, in un array `pari`; a quel punto il problema è *invertire* `pari`, cioè il punto A, e la parità della lunghezza di `v` non conta più.

```
let v = [7, -1, 33, 9, 0, 5];

let pari = [];
for (let i = 0; i < v.length; i += 2) {
  pari[pari.length] = v[i];                    // [ 7, 33, 0 ]
}

let risultato = new Array(v.length);
for (let i = 0; i < v.length; i++) {
  if (i % 2 != 0) {
    risultato[i] = v[i];
  } else {
    risultato[i] = pari[pari.length - 1 - i / 2];   // formula di A2 su pari
  }
}

console.log(risultato);                        // [ 0, -1, 33, 9, 7, 5 ]
```

La posizione pari `i` di `v` corrisponde alla posizione `i / 2` di `pari` (`0 → 0`, `2 → 1`, `4 → 2`).

I casi limite. B1, B3, B4 e B5 danno sempre lo stesso risultato: con `[1, 2, 3, 4, 5]` si stampa `[ 5, 2, 3, 4, 1 ]`, con `[10, 11, 12, 13, 14, 15]` si stampa `[ 14, 11, 12, 13, 10, 15 ]`, con `[7]` si stampa `[ 7 ]` e con `[]` i cicli non partono e si stampa `[]`.

Errori che abbiamo incontrato

- **`lenght` invece di `length`.** L'editor non lo segnala, ma `v.lenght` non esiste e vale `undefined`, quindi il ciclo non fa quello che ci si aspetta. Stesso discorso se l'array si chiama `ris` in una riga e `risultato` in un'altra.
- **`j++` dentro l'`if` nel ciclo dei dispari.** `j` avanza solo quando si copia, quindi i valori finiscono attaccati all'inizio: `[ -1, 9, 5, <3 empty items> ]`.
- **`i < v.length - 1`.** L'ultimo elemento non viene mai letto. Con `[7, -1, 33, 9, 0]` non si nota (l'ultima posizione è pari), con `[7, -1, 33, 9, 0, 5]` il `5` non viene copiato. Va scritto `i <= v.length - 1` oppure `i < v.length`.
- **Non rimettere `j = 0`** prima del secondo ciclo di B1: `j` è rimasto al valore raggiunto alla fine del primo ciclo.
- **`j++` fuori dall'`if` nel ciclo dei pari.** Se l'ultima posizione è dispari, `j` si sposta senza aver scritto nulla e i valori pari finiscono nelle posizioni dispari: `[ <1 empty item>, 0, <1 empty item>, 33, <1 empty item>, 7 ]`.
- **Provare con un solo array.** Molti di questi errori si vedono solo con un array di lunghezza pari, o con un elemento in più: provate sempre più input diversi.

Esercizio 2

Ripasso: Recap 1 · Gli array; per la variabile flag, l'esercizio di ricerca dello zero dell'esercitazione 7.

Dato un array `v`, scrivere un programma che stampi `true` se **ogni** `2` presente nell'array è **immediatamente seguito** da un `3`, `false` altrimenti. Se non ci sono `2` la risposta è `true`; un `2` in ultima posizione non è seguito da niente, quindi dà `false`.

Provate il programma con questi array:

v	cosa stampa
`[1, 2, 3, 5, 2, 3]`	`true`
`[2, 4, 3]`	`false`
`[5, 2, 3, 2]`	`false`
`[1, 5, 7]`	`true`
`[2, 2, 3]`	`false`
`[]`	`true`

▼ **Soluzione**

È la solita **variabile flag**: si parte da `true` e si cambia idea solo se si trova un controesempio, cioè un `2` che **non** è seguito da un `3`. Per guardare l'elemento successivo si usa `v[i + 1]`: se `i` è l'ultima posizione, `v[i + 1]` non esiste e vale `undefined`, che è diverso da `3`, quindi il caso del `2` in fondo è già gestito.

```
let v = [1, 2, 3, 5, 2, 3];

let ok = true;
for (let i = 0; i < v.length; i++) {
  if (v[i] == 2 && v[i + 1] != 3) {            // un 2 non seguito da un 3
    ok = false;
  }
}

console.log(ok);                               // true
```

Per gli altri array della tabella si ottiene: `[2, 4, 3]` → `false`; `[5, 2, 3, 2]` → `false` (l'ultimo `2` non ha un successivo); `[1, 5, 7]` → `true` (nessun `2`); `[2, 2, 3]` → `false` (il primo `2` è seguito da un `2`); `[]` → `true`.

Due errori frequenti:

- mettere `ok = true` in un `else` dentro il ciclo: il flag verrebbe rimesso a `true` dagli elementi successivi, e conterebbe solo l'**ultimo** controllo;
- scrivere `v[i + 1] == 3` come condizione di errore, cioè invertire il test: il controesempio è un `2` seguito da qualcosa di **diverso** da `3`.

In alternativa si può restituire `false` appena si trova il controesempio e `true` **dopo** il ciclo.

Esercizio 3

Ripasso: Recap 2 · Array e funzioni; la funzione `contiene` è nell'Esercizio 11 dell'esercitazione 7.

Funzioni che **costruiscono** un array di numeri casuali rispettando una regola. Ricordate che un intero casuale tra `min` e `max` (estremi inclusi) è `Math.floor(Math.random() * (max - min + 1)) + min`.

A. Scrivere la funzione `casualiPari(n, min, max)` che restituisce un array di `n` numeri casuali **pari** compresi tra `min` e `max`.

B. Scrivere la funzione `casualiDistinti(n, min, max)` che restituisce un array di `n` numeri casuali compresi tra `min` e `max` **tutti diversi tra loro**. Suggerimento: estraete un numero con un `do ... while` finché non ne trovate uno che non è già nell'array.

Provatele, ad esempio, con `casualiPari(8, 1, 20)` e `casualiDistinti(5, 1, 10)`.

▼ **Soluzione**

A. Il modo più semplice per ottenere un pari è **estrarre finché non esce un pari**, con un `do ... while`:

```
function casualiPari(n, min, max) {
  let risultato = [];
  for (let i = 0; i < n; i++) {
    let x;
    do {
      x = Math.floor(Math.random() * (max - min + 1)) + min;
    } while (x % 2 != 0);                       // si ripete finché x è dispari
    risultato[i] = x;
  }
  return risultato;
}

console.log(casualiPari(8, 1, 20));             // ad esempio [ 12, 4, 18, 4, 2, 20, 10, 6 ]
```

Si può usare anche una formula che produce **direttamente** un pari (si estrae un numero casuale e lo si moltiplica per `2`), ma bisogna stare attenti agli estremi: con `min` e `max` dispari il primo e l'ultimo pari possibili sono diversi da `min` e `max`. Il `do ... while` evita questo problema. Attenzione però: se tra `min` e `max` **non c'è nessun pari** (ad esempio `min = max = 7`) il ciclo non finisce mai.

B. La regola «diverso da tutti gli altri» si controlla con la funzione `contiene` dell'esercitazione 7, applicata al risultato che si sta costruendo:

```
function contiene(v, x) {
  for (let i = 0; i < v.length; i++) {
    if (v[i] == x) {
      return true;
    }
  }
  return false;
}

function casualiDistinti(n, min, max) {
  let risultato = [];
  for (let i = 0; i < n; i++) {
    let x;
    do {
      x = Math.floor(Math.random() * (max - min + 1)) + min;
    } while (contiene(risultato, x));           // si ripete finché x c'è già
    risultato[risultato.length] = x;
  }
  return risultato;
}

console.log(casualiDistinti(5, 1, 10));         // ad esempio [ 7, 2, 9, 4, 10 ]
```

Qui si assegna a `risultato[risultato.length]` (e non a `risultato[i]`): è lo stesso, ma rende evidente che si aggiunge **in fondo**, e che `contiene` guarda solo gli elementi già inseriti.

Attenzione: se i valori possibili sono **meno** di `n` (ad esempio `casualiDistinti(5, 1, 3)`) non ci sono abbastanza numeri diversi e il programma non termina. Una funzione robusta controllerebbe prima che `max - min + 1 >= n`.

Esercizio 4

Ripasso: Recap 2 · Array e funzioni.

Funzioni che **costruiscono** un array con numeri **chiesti all'utente**. Si legge con `prompt` e si converte con `Number`, come negli esercizi precedenti.

A. Scrivere la funzione `leggiArray(n)` che chiede all'utente `n` numeri e restituisce l'array che li contiene, nell'ordine in cui sono stati inseriti.

B. Scrivere la funzione `leggiFinoAZero()` che chiede numeri **finché l'utente non inserisce `0`** e restituisce l'array dei numeri inseriti, **senza** lo zero finale. Non si sa in anticipo quanti saranno.

C. Scrivere la funzione `leggiPositivi(n)` che chiede all'utente `n` numeri **positivi**: se l'utente ne inserisce uno non positivo, il programma lo richiede finché non è valido.

Per ciascuna scrivete poi un breve programma che la usa e stampa l'array letto.

▼ **Soluzione**

A. Si sa quante volte ripetere: un ciclo `for` che a ogni giro aggiunge il numero letto:

```
function leggiArray(n) {
  let risultato = [];
  for (let i = 0; i < n; i++) {
    risultato[i] = Number(prompt("Numero " + (i + 1) + ": "));
  }
  return risultato;
}

let v = leggiArray(4);
console.log(v);                                  // ad esempio [ 5, -2, 9, 5 ]
```

B. Non si sa quante volte ripetere: serve un `while` con **sentinella** (lo zero), come nell'esercitazione 6. Si legge **prima** del ciclo e **di nuovo** alla fine di ogni giro:

```
function leggiFinoAZero() {
  let risultato = [];
  let x = Number(prompt("Numero (0 per finire): "));
  while (x != 0) {
    risultato[risultato.length] = x;            // lo zero finale non si aggiunge
    x = Number(prompt("Numero (0 per finire): "));
  }
  return risultato;
}

let w = leggiFinoAZero();
console.log(w);                                 // con 4, 7, 0 si stampa [ 4, 7 ]
```

Qui si assegna a `risultato[risultato.length]`, e **non** a `risultato[i]`: non c'è un contatore `i`, e si aggiunge sempre in fondo. Se l'utente scrive subito `0` si restituisce l'array vuoto `[]`.

C. È la validazione dell'input con `do ... while` (esercitazione 4), ripetuta per ognuno degli `n` numeri:

```
function leggiPositivi(n) {
  let risultato = [];
  for (let i = 0; i < n; i++) {
    let x;
    do {
      x = Number(prompt("Numero positivo " + (i + 1) + ": "));
    } while (x <= 0);                           // si richiede finché non è positivo
    risultato[i] = x;
  }
  return risultato;
}

let p = leggiPositivi(3);
console.log(p);
```

La variabile `x` va dichiarata **fuori** dal `do ... while` ma **dentro** il `for`: così il `while (x <= 0)` la vede, e a ogni giro del `for` se ne usa una nuova.

Esercizio 5

Ripasso: Recap 2 · Array e funzioni (la parte sul passaggio per riferimento).

Un array **non si copia** con l'assegnazione. Vediamo perché e come si fa.

A. Dite cosa stampa questo programma, **senza eseguirlo**. Poi spiegate con le vostre parole perché `a` è cambiato anche se abbiamo modificato solo `b`.

```
let a = [1, 2, 3];
let b = a;
b[0] = 99;
console.log(a);
console.log(b);
console.log(a == b);
```

B. Volevamo **conservare** `a` e lavorare su una copia. Scrivere la funzione `copia(v)` che restituisce un **nuovo array** con gli stessi elementi di `v`, e usarla per correggere il programma di A: dopo la modifica di `b`, `a` deve essere ancora `[1, 2, 3]`.

▼ **Soluzione**

A. Stampa `[ 99, 2, 3 ]`, poi `[ 99, 2, 3 ]`, poi `true`.

Una variabile che contiene un array **non contiene gli elementi**: contiene un **riferimento**, cioè l'«indirizzo» dell'array in memoria. Con `let b = a;` si copia solo il riferimento: `a` e `b` sono **due nomi per lo stesso array**. Modificare l'array tramite `b` lo cambia anche per `a`, perché c'è un solo array. Per questo `a == b` è `true`: sono lo stesso array.

È diverso dai numeri: con `let y = x;` si copia il **valore**, e cambiare `y` non tocca `x`.

B. Per avere una copia vera bisogna **creare un array nuovo** e riempirlo con gli elementi di `v`:

```
function copia(v) {
  let risultato = [];
  for (let i = 0; i < v.length; i++) {
    risultato[i] = v[i];
  }
  return risultato;
}

let a = [1, 2, 3];
let b = copia(a);                          // array nuovo, non un secondo nome per a
b[0] = 99;
console.log(a);                            // [ 1, 2, 3 ]
console.log(b);                            // [ 99, 2, 3 ]
console.log(a == b);                       // false: sono due array diversi
```

È anche il motivo per cui le nostre funzioni **costruiscono un array nuovo** invece di modificare quello ricevuto: un parametro array è un riferimento allo stesso array di chi chiama.

Ludovica Pannitto - Università di Pisa - CC BY-NC-SA 4.0