Esercitazione 7 - 09/11

Argomenti:

- **array**: creare un array, indici, `length`
- scorrere un array con un ciclo `for` e `while`
- array e funzioni: array come parametro e come valore restituito
- il passaggio per riferimento: perché le nostre funzioni **restituiscono sempre un nuovo array** invece di modificare quello ricevuto

---

Indice degli esercizi

Esercizio	Argomento
Esercizio 1	Indici e `length`: cosa stampa
Esercizio 2	Scorrere un array: cosa stampa
Esercizio 3	Somme e conteggi con un ciclo `for`
Esercizio 4	Cercare lo zero: prima e ultima posizione
Esercizio 5	Array: trovare gli errori
Esercizio 6	Media e conteggio degli elementi
Esercizio 7	Posizione del massimo
Esercizio 8	Le posizioni di un elemento
Esercizio 9	Costruire un nuovo array: i quadrati
Esercizio 10	Filtrare un array
Esercizio 11	Togliere i duplicati
Esercizio 12	Due array: somma elemento per elemento
Esercizio 13	Due array: confronto delle somme
Esercizio 14	Unione di due insiemi

Strumento aggiuntivo · Python Tutor

Per **vedere passo passo** cosa fa un programma (il valore delle variabili a ogni giro di un ciclo) si può usare **Python Tutor**, che supporta anche JavaScript: <https://pythontutor.com/javascript.html>

Incolla il codice, premi **Visualize Execution** e avanza con **Next**. Prova con questo esempio:

```
let i = 1;
let somma = 0;
while (i <= 4) {
  somma += i;
  i++;
}
console.log(somma);                          // 10
```

Guarda come cambiano `i` e `somma` a ogni giro. È utile anche per controllare le tracce degli esercizi.

Recap 1 · Gli array

Finora ogni variabile conteneva **un solo valore** (uno scalare). Un **array** è una oggetto che contiene **più valori in fila**, come una scatola con tanti scomparti numerati.

```
let città = ["Roma", "Firenze", "Pisa", "Lucca"];
let voti = [28, 30, 25];
let pippo = [];                              // array vuoto: nessun elemento
```

Ogni elemento ha un **indice**, cioè la sua posizione. **Si parte da zero**:

indice	0	1	2	3
`città`	`"Roma"`	`"Firenze"`	`"Pisa"`	`"Lucca"`

```
console.log(città[0]);                       // Roma: il primo elemento
console.log(città[2]);                       // Pisa
città[1] = "Siena";                          // cambia il valore dell'elemento in
posizione 1
console.log(città);                          // [ 'Roma', 'Siena', 'Pisa', 'Lucca' ]
```

La **lunghezza** dell'array (il numero di elementi) si ottiene con `length`, **senza parentesi**:

```
console.log(città.length);                   // 4
console.log(città[città.length - 1]);        // Lucca: l'ultimo elemento
console.log(città[4]);                       // undefined: la posizione 4 non esiste
```

Con `length` elementi gli indici validi vanno da `0` a **`length - 1`**. Leggere fuori da questo intervallo **non dà errore**: dà `undefined`.

Assegnare a una posizione oltre la fine, invece, **allarga** l'array: le posizioni intermedie restano vuote e la lunghezza diventa (indice più grande usato) + 1.

```
let pippo = [];
pippo[2] = 5;                                   // oltre la fine: l'array si allarga
console.log(pippo.length);                      // 3: le posizioni 0 e 1 restano vuote
console.log(pippo[0]);                          // undefined: leggendole si ottiene undefined
console.log(pippo);                             // [ <2 empty items>, 5 ]
```

Le posizioni vuote sono dei «buchi»: non contengono il valore `undefined`, semplicemente **non esistono come elementi** (per questo `console.log` scrive `<2 empty items>`). Leggerle però dà `undefined`, come per qualsiasi posizione fuori dall'array.

Per **aggiungere un elemento in fondo** si assegna alla posizione `length`, che è la prima ancora libera:

```
let v = [4, 7];
v[v.length] = 9;                                // v.length vale 2: si assegna alla posizione
2, in fondo
console.log(v);                                 // [ 4, 7, 9 ]
```

Per **scorrere** un array si usa un ciclo `for` con l'indice come contatore:

```
let voti = [28, 30, 25];
for (let i = 0; i < voti.length; i++) {         // i vale 0, 1, 2: si ferma prima di length
  console.log("voto", i, "=", voti[i]);
}
```

Esercizio 1

Dato l'array:

```
let a = [10, 20, 30, 40, 50];
```

dite cosa stampa ciascuna istruzione, **senza eseguire**.

A. `console.log(a[1]);`

B. `console.log(a.length);`

C. `console.log(a[a.length - 1]);`

D. `console.log(a[a.length]);`

E. `console.log(a[1] + a[2]);`

F. `a[0] = a[4]; console.log(a);`

G. `a[7] = 99; console.log(a.length);`

▼ **Soluzione**

A. `20`: l'indice `1` è la **seconda** posizione, perché si parte da `0`.

B. `5`: gli elementi sono cinque.

C. `50`: l'ultimo elemento ha indice `length - 1`, cioè `4`.

D. `undefined`: l'indice `5` è già fuori dall'array (gli indici validi sono `0`, ..., `4`). JavaScript non dà errore, ma il valore è `undefined`.

E. `50`: `20 + 30`. Si sommano i **valori** degli elementi, non gli indici.

F. `[ 50, 20, 30, 40, 50 ]`: il primo elemento prende il valore dell'ultimo. Lo `=` è un **assegnamento**, non un confronto: viene scritta solo la posizione `0`, il resto dell'array non cambia (per confrontare servirebbe `==` o `===`).

G. `8`. Assegnare a una posizione oltre la fine **allarga** l'array: le posizioni `5` e `6` restano vuote e la lunghezza diventa `7 + 1 = 8`. Le posizioni `5` e `6` esistono, ma sono «vuote»: `console.log(a)` mostra `<2 empty items>` tra l'ultimo elemento e il `99`, e leggendole si ottiene `undefined`. In generale la lunghezza è sempre (indice più grande usato) + 1. Per aggiungere un elemento **in fondo** basta usare come indice proprio `length`: `a[a.length] = 99`.

Esercizio 2

Per ogni programma, dite cosa stampa **senza eseguirlo**. Per il programma B aiutatevi con una tabella (una riga per ogni valore di `i`).

A.

```
let v = [3, 8, 1, 6];
for (let i = 0; i < v.length; i++) {
  console.log(i, v[i]);
}
```

B.

```
let v = [2, 4, 6, 8];
for (let i = 1; i < v.length; i++) {
  v[i] = v[i] + v[i - 1];
}
console.log(v);
```

C.

```
let v = [5, 7, 9];
let s = 0;
for (let i = v.length - 1; i >= 0; i--) {
  s = s * 10 + v[i];
}
console.log(s);
```

▼ **Soluzione**

A. Stampa, una per riga, le coppie *indice, valore*: `0 3`, `1 8`, `2 1`, `3 6`.

B. Ogni elemento (dal secondo in poi) diventa la somma di se stesso e del **precedente già aggiornato**:

`i`	`v[i]` prima	`v[i-1]`	`v[i]` dopo	`v`
				`[2, 4, 6, 8]`
1	`4`	`2`	`6`	`[2, 6, 6, 8]`
2	`6`	`6`	`12`	`[2, 6, 12, 8]`
3	`8`	`12`	`20`	`[2, 6, 12, 20]`

Stampa `[ 2, 6, 12, 20 ]`. Ogni elemento è ora la somma di tutti quelli fino a lì (le **somme parziali**).

C. Il ciclo scorre l'array **al contrario** (`i` vale `2`, `1`, `0`): `s` diventa `0*10+9 = 9`, poi `9*10+7 = 97`, poi `97*10+5 = 975`. Stampa `975`.

Esercizio 3

Dato l'array:

```
let v = [4, -2, 7, 0, 9, -5];
```

scrivere un programma, con un ciclo `for`, che **stampi**:

A. la somma di **tutti** gli elementi di `v`.

B. la somma dei soli elementi **positivi**.

C. quanti elementi sono **pari** (ricordate che `0` è pari).

D. la somma degli elementi che si trovano in posizione **dispari** (indici `1`, `3`, `5`).

▼ **Soluzione**

A. Si usa un **accumulatore**: parte da `0` e a ogni giro si aggiunge l'elemento corrente.

```
let somma = 0;
for (let i = 0; i < v.length; i++) {
  somma = somma + v[i];
}
console.log(somma);                              // 13
```

Se l'array fosse vuoto il ciclo non partirebbe e si stamperebbe `0`, che è la somma giusta di nessun numero.

B. Come A, ma si aggiunge **solo se** l'elemento rispetta la condizione:

```
let somma = 0;
for (let i = 0; i < v.length; i++) {
  if (v[i] > 0) {
    somma = somma + v[i];
  }
}
console.log(somma);                              // 20: 4 + 7 + 9
```

C. Serve un **contatore** invece di un accumulatore: si aggiunge `1`, non il valore dell'elemento.

```
let conta = 0;
for (let i = 0; i < v.length; i++) {
  if (v[i] % 2 == 0) {
    conta++;
  }
}
console.log(conta);                              // 3: 4, -2 e 0
```

D. Qui conta la **posizione**, non il valore: basta far partire `i` da `1` e farlo avanzare di `2` alla volta.

```
let somma = 0;
for (let i = 1; i < v.length; i += 2) {         // i vale 1, 3, 5
  somma = somma + v[i];
}
console.log(somma);                             // -7: -2 + 0 + (-5)
```

Si poteva anche scorrere tutto l'array e sommare solo quando `i % 2 == 1`: stesso risultato, ma con più giri inutili.

Attenzione a cosa si controlla: la posizione o il valore? In D il controllo va fatto sulla **posizione** `i`, non sull'elemento `v[i]`. Con gli array bisogna leggere bene il testo: «gli elementi dispari» e «gli elementi in posizione dispari» sono due problemi diversi. Per i **valori** dispari il controllo sarebbe `v[i] % 2 != 0`, **non** `v[i] % 2 == 1`: in JavaScript il resto ha il segno del dividendo, e `-5 % 2` vale `-1`, non `1`. Con `== 1` il `-5` non verrebbe riconosciuto come dispari.

Esercizio 4

Cercare lo **zero**. Per ciascun punto provate il programma con `[5, 3, 0, 8, 0]`, con `[1, 2, 3]` e con `[0, 4]`.

A. Scrivere un programma che stampi la **posizione del primo zero** presente nell'array `v`, oppure `-1` se non ce ne sono. Con `[5, 3, 0, 8, 0]` deve stampare `2`.

B. Riscrivere il programma con un ciclo `while` che **si ferma appena trova lo zero** (senza continuare a scorrere l'array inutilmente).

C. Modificare il programma A in modo che stampi la posizione dell'**ultimo** zero. Con `[5, 3, 0, 8, 0]` deve stampare `4`.

▼ **Soluzione**

A. Si usa `pos = -1` per dire «non trovato finora». Lo si cambia solo alla **prima** volta che si incontra uno zero, cioè mentre `pos` vale ancora `-1`:

```
let v = [5, 3, 0, 8, 0];
let pos = -1;
for (let i = 0; i < v.length; i++) {
  if (v[i] == 0 && pos == -1) {
    pos = i;
  }
}
console.log(pos);                               // 2
```

Con `[1, 2, 3]` stampa `-1`; con `[0, 4]` stampa `0`. Il valore `-1` va bene proprio perché **non è un indice possibile**: se si usasse `0` come valore «non trovato» non si saprebbe distinguere tra «lo zero è in posizione 0» e «non c'è nessuno zero».

Un'altra scrittura usa il **ciclo `for` con `break`**: `break` fa **uscire dal ciclo** appena eseguito, quindi alla prima occorrenza non serve più il controllo `pos == -1`:

```
let v = [5, 3, 0, 8, 0];
let pos = -1;
for (let i = 0; i < v.length; i++) {
  if (v[i] == 0) {
    pos = i;
    break;                                    // si esce dal ciclo: gli zeri successivi
non interessano
  }
}
console.log(pos);                             // 2
```

Il ciclo è scritto per arrivare fino in fondo all'array, ma di fatto si ferma al primo zero trovato.

Un errore frequente è scrivere `console.log("non trovato")` **dentro** il ciclo, nel ramo `else`: verrebbe stampato a ogni elemento diverso da zero, e non una volta sola alla fine.

B. Il ciclo continua **finché non si è arrivati in fondo e l'elemento corrente non è zero**:

```
let v = [5, 3, 0, 8, 0];
let i = 0;
while (i < v.length && v[i] != 0) {
  i++;
}
if (i < v.length) {
  console.log(i);                             // 2
} else {
  console.log(-1);
}
```

Alla fine del ciclo ci sono due casi: o `i < v.length` (ci siamo fermati su uno zero, e `i` è la sua posizione), oppure `i == v.length` (siamo arrivati in fondo senza trovarlo). L'ordine delle due condizioni nel `while` conta: se si scrivesse `v[i] != 0 && i < v.length`, per `i == v.length` si leggerebbe `v[i]` fuori dall'array (che dà `undefined`, non errore, ma è comunque una lettura sbagliata).

C. Basta togliere il controllo `pos == -1`: così a **ogni** zero la posizione viene aggiornata, e alla fine resta quella dell'ultimo.

```
let v = [5, 3, 0, 8, 0];
let pos = -1;
for (let i = 0; i < v.length; i++) {
  if (v[i] == 0) {
    pos = i;
  }
}
console.log(pos);                                  // 4
```

È la stessa struttura di A: l'unica differenza, `&& pos == -1`, decide se si tiene il **primo** o l'**ultimo** zero.

Recap 2 · Array e funzioni

Un array può essere **parametro** di una funzione, e una funzione può **restituire** un array.

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

console.log(somma([1, 2, 3, 4]));                  // 10
console.log(somma([]));                            // 0: array vuoto, il ciclo non parte
```

C'è però una differenza importante rispetto ai numeri. Quando si passa un numero a una funzione, la funzione riceve una **copia**: cambiarla dentro non tocca l'originale. Quando si passa un array, la funzione riceve **lo stesso array** (non una copia): se lo modifica, la modifica si vede anche fuori.

```
function raddoppia(v) {
  for (let i = 0; i < v.length; i++) {
    v[i] = v[i] * 2;                               // modifica l'array originale!
  }
  return v;
}

let a = [1, 2, 3];
let b = raddoppia(a);
console.log(a);                                    // [ 2, 4, 6 ]: a è cambiato, anche se non
volevamo
```

Lo stesso succede con l'assegnazione `=`: **non copia l'array**, crea solo un secondo nome per lo stesso array.

```
let a = [1, 2, 3];
let b = a;                                  // b e a sono lo STESSO array
b[0] = 100;
console.log(a);                             // [ 100, 2, 3 ]
```

Per avere una **copia vera** si può usare `let b = [...a];` oppure `let b = a.slice();`.

La nostra regola: una funzione **non modifica mai** l'array che riceve. Se deve produrre un array, ne **costruisce uno nuovo** (partendo da `[]`) e lo **restituisce**. Così l'array di partenza resta com'era e non ci sono sorprese.

```
function raddoppia(v) {
  let risultato = [];                       // array nuovo
  for (let i = 0; i < v.length; i++) {
    risultato[i] = v[i] * 2;
  }
  return risultato;                         // restituito: v non è toccato
}

let a = [1, 2, 3];
console.log(raddoppia(a));                  // [ 2, 4, 6 ]
console.log(a);                             // [ 1, 2, 3 ]
```

Esercizio 5

Ciascun programma ha un **problema**. Dite cosa succede quando lo si esegue (errore? risultato strano?), e correggetelo.

A. Vogliamo stampare tutti gli elementi.

```
let v = [4, 5, 6];
for (let i = 0; i <= v.length; i++) {
  console.log(v[i]);
}
```

B. Vogliamo la somma degli elementi.

```
let v = [4, 5, 6];
let s = 0;
for (let i = 0; i < v.length(); i++) {
  s = s + v[i];
}
console.log(s);
```

C. Vogliamo l'ultimo elemento.

```
let v = [4, 5, 6];
console.log(v[v.length]);
```

D. Vogliamo sommare 1 a ogni elemento.

```
let v = [4, 5, 6];
for (let i = 0; i < v.length; i++) {
  v + 1;
}
console.log(v);
```

E. Vogliamo stampare la somma di tutti gli elementi di `v`.

```
function somma(v) {
  let s = 0;
  for (let i = 0; i < v.length; i++) {
    let s = s + v[i];
  }
  return s;
}
console.log(somma([4, 5, 6]));
```

▼ **Soluzione**

A. Stampa `4`, `5`, `6` e poi `undefined`: con `<=` l'ultima iterazione ha `i = 3`, che è fuori dall'array. È l'errore classico "di uno": la condizione corretta è `i < v.length`.

B. `length` **non è una funzione**: `v.length()` dà `TypeError: v.length is not a function`. Correzione: `v.length`, senza parentesi.

C. Stampa `undefined`: l'ultimo indice valido è `length - 1`. Correzione: `v[v.length - 1]`.

D. Stampa ancora `[ 4, 5, 6 ]`. L'istruzione `v + 1;` calcola qualcosa e **butta via** il risultato; per di più `v` è un array, non un numero. Correzione: `v[i] = v[i] + 1;` (oppure `v[i]++`).

E. Dà `ReferenceError: Cannot access 's' before initialization`: dentro il ciclo, `let s = ...` **dichiara una nuova variabile** `s` (che nasconde quella esterna) e per calcolare il suo valore usa se stessa, che ancora non esiste. Correzione: togliere `let` e scrivere `s = s + v[i];`. Se anche non desse errore, la `s` esterna resterebbe a `0`: è il tema dello shadowing visto nell'esercitazione 5.

Esercizio 6

A. Scrivere la funzione `media(v)` che, dato un array di numeri, **restituisce** la media dei suoi elementi. Provarla con `[10, 20, 30]`, `[7]`, `[-4, 4]`.

B. Scrivere la funzione `quantiMaggiori(v, x)` che restituisce **quanti** elementi di `v` sono maggiori di `x`. Ad esempio `quantiMaggiori([3, 9, 5, 12], 4)` restituisce `3`.

▼ **Soluzione**

A.

```
function media(v) {
  let somma = 0;
  for (let i = 0; i < v.length; i++) {
    somma = somma + v[i];                    // accumulatore
  }
  return somma / v.length;
}

console.log(media([10, 20, 30]));            // 20
console.log(media([7]));                     // 7
console.log(media([-4, 4]));                 // 0
```

B. Si usa un **contatore** invece di un accumulatore di somme:

```
function quantiMaggiori(v, x) {
  let conta = 0;
  for (let i = 0; i < v.length; i++) {
    if (v[i] > x) {
      conta++;
    }
  }
  return conta;
}

console.log(quantiMaggiori([3, 9, 5, 12], 4));  // 3
```

Esercizio 7

Scrivere la funzione `posizioneMassimo(v)` che restituisce **l'indice** (non il valore) del massimo. Con `[3, 9, 5, 12, 7]` deve restituire `3`.

▼ **Soluzione**

Si tiene traccia della **posizione** del massimo invece del suo valore:

```
function posizioneMassimo(v) {
  let pos = 0;                                  // il primo elemento è il massimo "per ora"
  for (let i = 1; i < v.length; i++) {          // si parte da 1: il primo è già stato
guardato
    if (v[i] > v[pos]) {
      pos = i;
    }
  }
  return pos;
}

console.log(posizioneMassimo([3, 9, 5, 12, 7])); // 3
console.log(posizioneMassimo([-3, -7]));         // 0
```

Si parte da `pos = 0`, cioè da un elemento **che c'è davvero**, e non da un valore fisso come `0`, che potrebbe non essere nell'array (con `[-3, -7]` il massimo non è `0`). Con un array vuoto `v[0]` è `undefined`: la funzione va chiamata solo su array con almeno un elemento.

Esercizio 8

Scrivere la funzione `posizioni(v, x)` che restituisce un **nuovo array** con **tutti gli indici** in cui compare `x`. Ad esempio `posizioni([0, 5, 0, 0, 7], 0)` restituisce `[0, 2, 3]`; se `x` non compare restituisce `[]`.

▼ **Soluzione**

Si parte da un array vuoto e, a ogni occorrenza di `x`, si aggiunge **in fondo** la posizione `i`:

```
function posizioni(v, x) {
  let risultato = [];                           // partiamo da un array vuoto
  for (let i = 0; i < v.length; i++) {
    if (v[i] == x) {
      risultato[risultato.length] = i;          // in fondo: l'INDICE, non il valore
    }
  }
  return risultato;
}

console.log(posizioni([0, 5, 0, 0, 7], 0));     // [ 0, 2, 3 ]
console.log(posizioni([1, 2, 3], 9));           // []
```

Qui **non** si può usare `risultato[i] = ...`: `risultato` è più corto di `v`, e usare `i` come indice lascerebbe dei buchi. Si assegna sempre alla posizione `risultato.length`, la prima libera.

Esercizio 9

Come sempre, la funzione **non modifica** l'array che riceve: costruisce e **restituisce un nuovo array**.

Scrivere la funzione `quadrati(v)` che restituisce un nuovo array in cui ogni elemento è il **quadrato** del corrispondente elemento di `v`.

▼ **Soluzione**

```
function quadrati(v) {
  let risultato = [];
  for (let i = 0; i < v.length; i++) {
    risultato[i] = v[i] * v[i];                // stessa posizione i: il nuovo array ha la
stessa lunghezza
  }
  return risultato;
}

let a = [1, 2, 3, -4];
console.log(quadrati(a));                      // [ 1, 4, 9, 16 ]
console.log(a);                                // [ 1, 2, 3, -4 ]: a non è cambiato
```

Lo schema è sempre lo stesso: **partire da** `[]`, scorrere `v`, riempire `risultato`, **restituire** il nuovo array. Qui ogni elemento di `v` ne produce uno in `risultato`, nella stessa posizione.

Esercizio 10

Anche qui le funzioni **restituiscono un nuovo array** e non toccano quello ricevuto.

Scrivere la funzione `rimuoviPiccoli(v, n)` che restituisce un nuovo array con gli elementi di `v` **maggiori di** `n`, cioè senza quelli minori o uguali a `n`. Ad esempio `rimuoviPiccoli([5, 1, 8, 3, 9, 2], 3)` restituisce `[5, 8, 9]`.

▼ **Soluzione**

Si copiano nel nuovo array solo gli elementi che **passano il test**. Il nuovo array può essere più corto di `v`: si aggiunge sempre **in fondo**, alla posizione `risultato.length`.

```
function rimuoviPiccoli(v, n) {
  let risultato = [];
  for (let i = 0; i < v.length; i++) {
    if (v[i] > n) {
      risultato[risultato.length] = v[i];
    }
  }
  return risultato;
}

console.log(rimuoviPiccoli([5, 1, 8, 3, 9, 2], 3)); // [ 5, 8, 9 ]
console.log(rimuoviPiccoli([1, 2], 3));             // []
```

Esercizio 11

Scrivere la funzione `senzaDuplicati(v)` che restituisce un nuovo array in cui ogni valore compare **una volta sola** (si tiene la prima occorrenza, nello stesso ordine). Ad esempio `senzaDuplicati([4, 7, 4, 9, 7, 4])` restituisce `[4, 7, 9]`. Suggerimento: prima di aggiungere un elemento al risultato, controllate se c'è già.

▼ **Soluzione**

Conviene scrivere prima una funzione `contiene(v, x)`, che dice se `x` compare in `v`, e usarla **sul risultato che si sta costruendo**:

```
function contiene(v, x) {
  for (let i = 0; i < v.length; i++) {
    if (v[i] == x) {
      return true;                              // trovato: inutile continuare
    }
  }
  return false;                                 // il ciclo è finito senza trovarlo
}

function senzaDuplicati(v) {
  let risultato = [];
  for (let i = 0; i < v.length; i++) {
    if (!contiene(risultato, v[i])) {           // non c'è ancora: lo aggiungiamo
      risultato[risultato.length] = v[i];
    }
  }
  return risultato;
}

console.log(senzaDuplicati([4, 7, 4, 9, 7, 4]));    // [ 4, 7, 9 ]
console.log(senzaDuplicati([]));                    // []
```

Il `return true` dentro il ciclo termina **subito** la funzione appena si trova `x`. Il `return false` va **dopo** il ciclo: si può dire "non c'è" solo dopo aver controllato **tutti** gli elementi. Se fosse dentro il ciclo (ad esempio in un `else`), la funzione risponderebbe `false` già al primo elemento diverso da `x`.

Esercizio 12

Funzioni che lavorano su **due array**. Come sempre, non modificano gli array ricevuti.

Scrivere la funzione `sommaElementi(a, b)` che, dati due array **della stessa lunghezza**, restituisce un nuovo array in cui l'elemento in posizione `i` è `a[i] + b[i]`. Ad esempio `sommaElementi([1, 2, 3], [10, 20, 30])` restituisce `[11, 22, 33]`.

▼ **Soluzione**

Un solo ciclo, con lo **stesso indice** su entrambi gli array:

```
function sommaElementi(a, b) {
  let risultato = [];
  for (let i = 0; i < a.length; i++) {
    risultato[i] = a[i] + b[i];
  }
  return risultato;
}

console.log(sommaElementi([1, 2, 3], [10, 20, 30]));   // [ 11, 22, 33 ]
```

Esercizio 13

Scrivere la funzione `stessaSomma(a, b)` che restituisce `true` se la somma degli elementi di `a` è uguale alla somma degli elementi di `b` (anche se gli array hanno lunghezze diverse), `false` altrimenti. Suggerimento: riusate la funzione `somma` del Recap 2.

▼ Soluzione

Si calcolano le due somme e le si confrontano:

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

function stessaSomma(a, b) {
  return somma(a) == somma(b);
}

console.log(stessaSomma([1, 5], [2, 2, 2]));          // true (6 e 6)
console.log(stessaSomma([1, 5], [2, 2]));             // false (6 e 4)
```

Esercizio 14

Scrivere la funzione `unisci(a, b)` che, dati due array `a` e `b` che rappresentano due insiemi, restituisce l'array che rappresenta l'insieme `a U b`. Ad esempio `unisci([1, 2], [1, 4, 5])` restituisce `[1, 2, 4, 5]`.

▼ Soluzione

Un insieme non ha duplicati: si parte da una **copia di `a`** e poi si aggiungono gli elementi di `b` che **non sono già in `a`**. Si riusa la funzione `contiene` dell'esercizio 11.

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

function unisci(a, b) {
  let risultato = [];
  for (let i = 0; i < a.length; i++) {
    risultato[i] = a[i];                        // copia di a
  }
  for (let i = 0; i < b.length; i++) {
    if (!contiene(a, b[i])) {                   // 1 è già in a: non si ripete
      risultato[risultato.length] = b[i];
    }
  }
  return risultato;
}

console.log(unisci([1, 2], [1, 4, 5]));               // [ 1, 2, 4, 5 ]
console.log(unisci([], [7]));                         // [ 7 ]
console.log(unisci([1, 2], [2, 1]));                  // [ 1, 2 ]
```

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