Esercitazione 5 - 26/10

Argomenti:

- funzioni ricorsive: caso base e passo ricorsivo
- tracciare le chiamate ricorsive e capire in che ordine avvengono
- variabili locali e globali, scope e shadowing
- ripasso dei cicli annidati

Indice degli esercizi

Esercizio	Argomento
Esercizio 1	Ricorsione: cosa stampa e in che ordine
Esercizio 2	Ricorsione: trovare l'errore
Esercizio 3	Fattoriale
Esercizio 4	Somma dei quadrati
Esercizio 5	Somma tra due numeri
Esercizio 6	Variabili locali e globali: cosa stampa
Esercizio 7	Variabili e visibilità: errori e correzioni
Esercizio 8	Cicli annidati: cosa stampa
Esercizio 9	Figure di asterischi e numeri
Esercizio 10	Tavola pitagorica
Esercizio 11	Coppie di dadi
Esercizio 12	Numeri primi

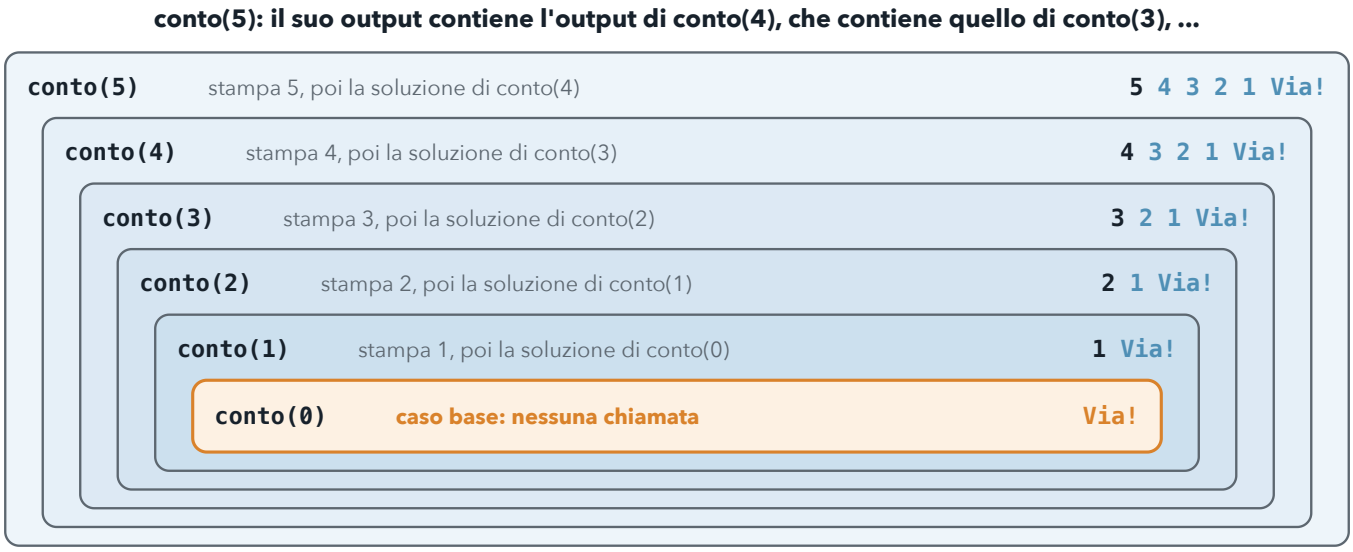
Recap 1 · Funzioni ricorsive

Una funzione è **ricorsiva** quando, nel suo corpo, **richiama se stessa**. Ogni chiamata lavora su un problema **più piccolo**, finché il problema diventa così semplice da avere una risposta immediata.

```
function conto(n) {
  if (n == 0) {                                  // caso base: ci si ferma
    console.log("Via!");
  } else {                                       // passo ricorsivo: la funzione richiama se
stessa
    console.log(n);
    conto(n - 1);                                // con un valore più piccolo
  }
}

conto(5);                                        // stampa: 5 4 3 2 1 Via!
```

Il problema che `conto(5)` risolve contiene una **copia più piccola di se stesso**: l'output `5 4 3 2 1 Via!` è `5` seguito dall'output di `conto(4)`, cioè `4 3 2 1 Via!`; questo a sua volta è `4` seguito dall'output di `conto(3)`, e così via fino a `conto(0)`, che stampa solo `Via!`. Come scatole una dentro l'altra:



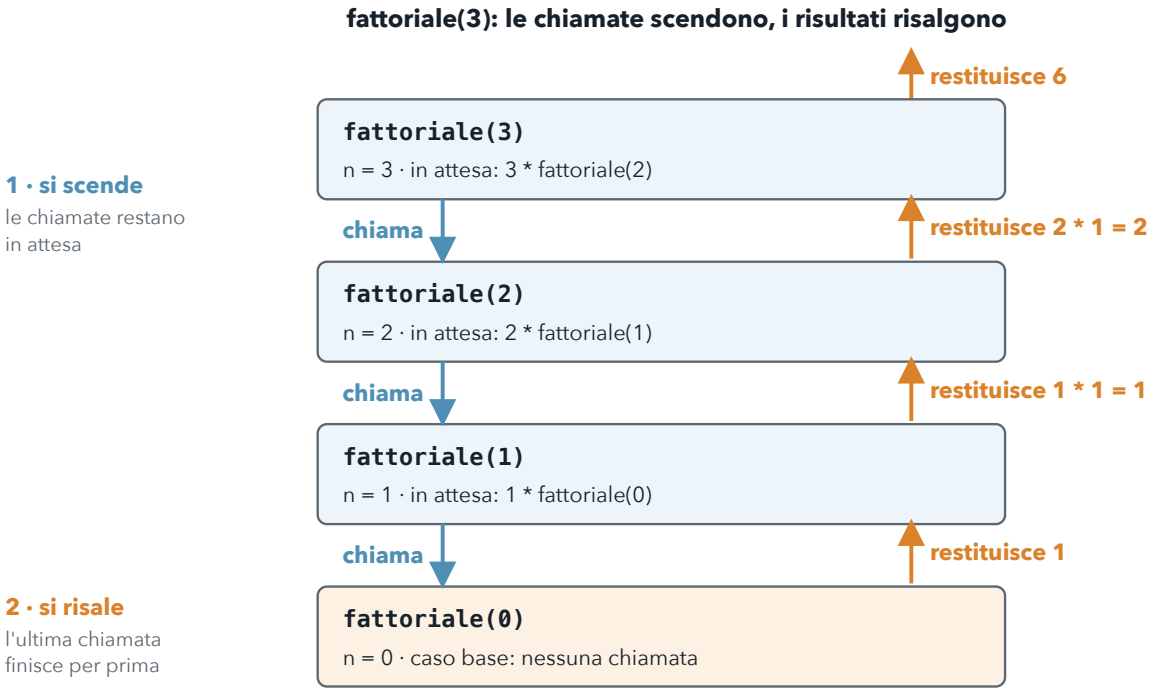
A destra, l'output di ogni chiamata: in **blu** la parte prodotta dalla chiamata più interna,
cioè la soluzione dello stesso problema con un valore più piccolo («4 3 2 1 Via!», poi «3 2 1 Via!», ... fino a «Via!»).

Una funzione ricorsiva ha sempre due parti:

- il **caso base**: il caso più semplice, in cui la risposta è immediata e **non** si richiama la funzione;
- il **passo ricorsivo**: la funzione si richiama su un valore che si **avvicina al caso base** (qui `n - 1`).

Ogni chiamata ha le **proprie variabili**: il `n` della chiamata `conto(5)` è diverso dal `n` della chiamata `conto(4)`. Le chiamate restano "in attesa" una dentro l'altra, e **ritornano in ordine inverso**: l'ultima chiamata (quella del caso base) finisce per prima.

Per `fattoriale(3)` (che vedremo nell'esercizio 3), ecco come le chiamate si aprono e si chiudono:



Per questo conta **dove** si scrive un'istruzione rispetto alla chiamata ricorsiva: quello che sta **prima** viene eseguito andando "in giù", quello che sta **dopo** viene eseguito al ritorno, "in su".

Attenzione: senza caso base, o con un passo che non si avvicina mai al caso base, le chiamate non finiscono più e JavaScript si ferma con l'errore `RangeError: Maximum call stack size exceeded`.

Esercizio 1

Per ogni programma, dite cosa stampa **senza eseguirlo**; poi eseguitelo per controllare.

A. Questi due programmi sono quasi uguali: cosa stampa ciascuno? **Cosa cambia nell'output**, e perché?

Programma 1

```
function conto(n) {
  if (n == 0) {
    console.log("Via!");
  } else {
    console.log(n);
    conto(n - 1);
  }
}
conto(3);
```

Programma 2

```
function conto(n) {
  if (n == 0) {
    console.log("Via!");
  } else {
    conto(n - 1);
    console.log(n);
  }
}
conto(3);
```

B.

```
function f(n) {
  if (n > 0) {
    console.log("entro", n);
    f(n - 1);
    console.log("esco", n);
  }
}
f(2);
```

C. Nel programma B, quante volte viene **chiamata** la funzione `f` (compresa la chiamata iniziale `f(2)`)? E se si chiama `f(5)`?

Esercizio 2

Ciascuna di queste funzioni ha un **problema**. Dite cosa succede quando le si esegue (stampa all'infinito? un errore? un risultato strano?), e correggetele.

A.

```
function conto(n) {
  console.log(n);
  conto(n - 1);
}
conto(3);
```

B.

```
function conto(n) {
  if (n == 0) {
    return;
  }
  console.log(n);
  conto(n);
}
conto(3);
```

C. Funziona con `conto(3)`. Cosa succede con `conto(-1)`? E con `conto(2.5)`?

```
function conto(n) {
  if (n == 0) {
    return;
  }
  console.log(n);
  conto(n - 1);
}
```

D. Cosa stampa?

```
function somma(n) {
  if (n == 0) {
    return 0;
  }
  n + somma(n - 1);
}
console.log(somma(3));
```

Esercizio 3

Il **fattoriale** di un numero naturale `n` è `n! = n * (n-1) * (n-2) * ... * 1`, con `0! = 1`. Si può scrivere come formula **ricorsiva**:

- `0! = 1` (caso base)
- `n! = n * (n-1)!` se `n > 0` (passo ricorsivo)

A. Calcolate a mano `3!` usando la formula ricorsiva: scrivete `3! = 3 * 2! = ...` fino al caso base, poi tornate indietro sostituendo i valori.

B. Scrivere la funzione `fattoriale(n)` che **restituisce** `n!` usando la ricorsione. Provarla con `0`, `1`, `5`, `10`.

C. Scrivere anche una versione **con un ciclo** `for`, `fattoriale_ciclo(n)`, e confrontare i risultati delle due versioni.

Recap 2 · Come si scrive una funzione ricorsiva

Per scrivere una funzione ricorsiva ci si pongono sempre le stesse domande:

1. **Caso base:** qual è il caso più semplice, in cui so già la risposta senza fare calcoli?
2. **Passo ricorsivo:** se conosco la risposta per un problema **più piccolo** (ad esempio `n - 1`), come la uso per ottenere la risposta per `n`?
3. **Fidarsi:** nel passo ricorsivo si **assume** che la chiamata ricorsiva funzioni già. Non serve seguire tutte le chiamate col pensiero, basta che caso base e passo siano giusti.

Un esempio, la somma dei numeri da 1 a `n`:

- caso base: `somma(0)` vale `0`;
- passo: `somma(n)` è `n` più la somma dei numeri da 1 a `n - 1`, cioè `n + somma(n - 1)`.

```
function somma(n) {
  if (n == 0) {
    return 0;
  }
  return n + somma(n - 1);
}
```

Spesso la formula del problema è **già** ricorsiva (fattoriale, somma, ...): in questi casi la funzione si scrive "copiando" la formula. Ricordate il `return` anche nel passo ricorsivo!

Esercizio 4

La somma dei **quadrati** dei primi `n` numeri naturali si può definire per ricorrenza:

- `Q(0) = 0`
- `Q(n) = Q(n−1) + n²` se `n > 0`

A. Scrivere la funzione ricorsiva `sommaQuadrati(n)` che **restituisce** `Q(n)`, cioè `1² + 2² + ... + n²` (non deve stampare). Ad esempio `sommaQuadrati(3)` restituisce `14`.

B. Nel programma principale, stampare `Q(3)`, `Q(10)` e `Q(100)`. Qual è il caso base? Cosa cambia, rispetto alla funzione `somma(n)` del Recap 2, nel passo ricorsivo?

Esercizio 5

A. Scrivere la funzione ricorsiva `sommaTra(a, b)` che restituisce la somma di tutti gli interi da `a` a `b` (estremi inclusi), e `0` se `a > b`. Ad esempio `sommaTra(3, 6)` restituisce `18` (`3 + 4 + 5 + 6`).

B. Qual è il caso base? Qui cosa **cambia** a ogni chiamata ricorsiva: `a`, `b` o entrambi?

C. Usando `sommaTra`, scrivere una riga che stampa la somma dei numeri da `1` a `100`, e una che stampa la somma dei numeri da `50` a `100`.

Recap 3 · Variabili locali e globali, scope e shadowing

Lo **scope** di una variabile è la parte di programma in cui la variabile **esiste** e si può usare.

- una variabile dichiarata **dentro una funzione** (e i suoi **parametri**) è **locale**: esiste solo durante la chiamata, e sparisce quando la funzione finisce. È per questo che ogni chiamata ricorsiva ha il **proprio** `n`;

- una variabile dichiarata **fuori** da ogni funzione è **globale**: si può leggere e modificare da qualunque punto (ad esempio un contatore dichiarato all'inizio del programma);
- con `let`, una variabile esiste solo nel **blocco** `{ ... }` in cui è dichiarata: anche la `i` di un `for` o una variabile dentro un `if`.

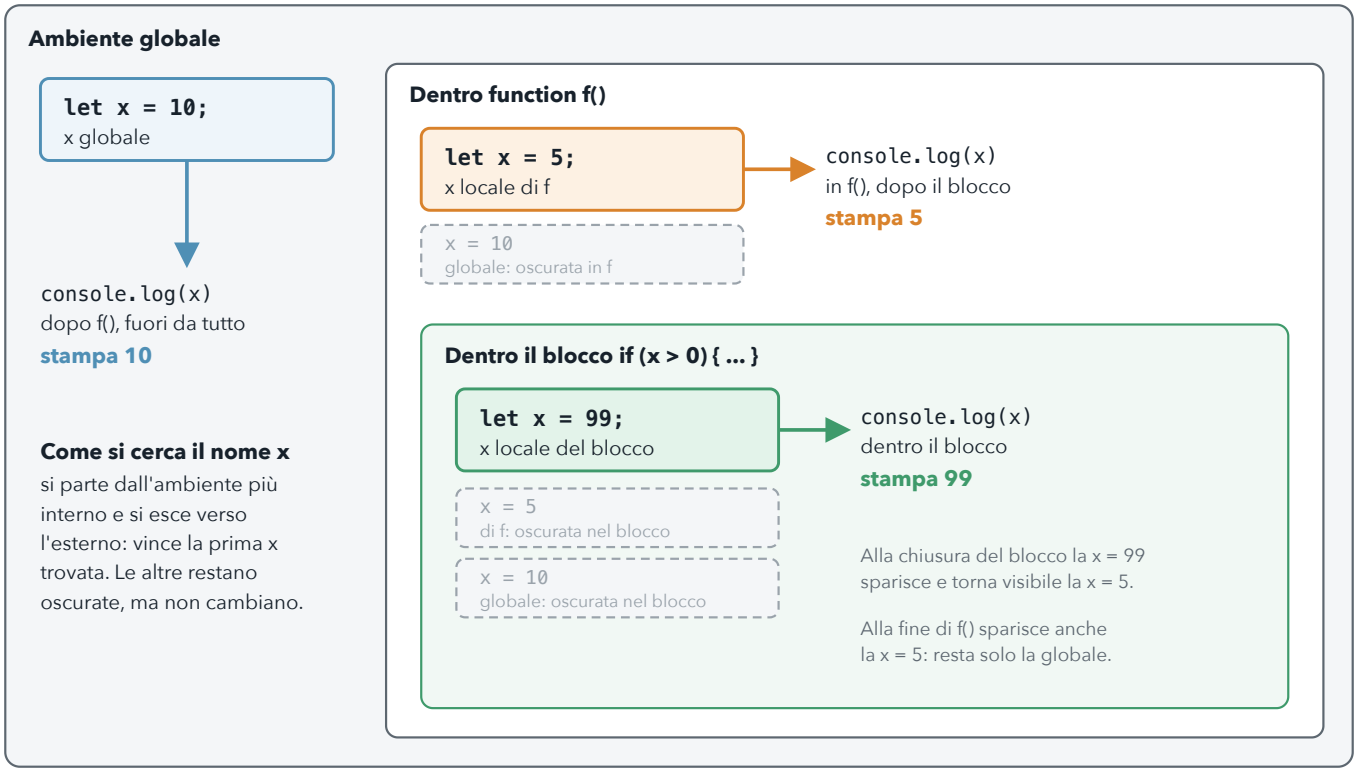
Se una variabile locale ha **lo stesso nome** di una variabile esterna, la locale **oscura** (in inglese *shadowing*) quella esterna: dentro la funzione il nome si riferisce alla locale, e la globale non è raggiungibile **e non viene toccata**.

```
let x = 10;                                  // globale

function f() {
  let x = 5;                                 // locale a f: oscura la x globale
  if (x > 0) {
    let x = 99;                              // locale al blocco: oscura la x di f
    console.log(x);                          // 99
  }
  console.log(x);                            // 5: il blocco è finito
}

f();
console.log(x);                              // 10: la x globale non è cambiata
```

Ogni livello (programma, funzione, blocco) ha le sue variabili. Per trovare il significato di `x`, JavaScript parte dal livello **più interno** e risale verso l'esterno, fermandosi alla **prima** `x` che trova.



Senza il `let` la situazione è diversa: l'assegnamento modifica la variabile **globale**.

```
function g() {
  x = x + 1;                                  // niente let: usa la x globale
}

g();
console.log(x);                               // 11
```

Lo stesso vale per i **parametri**: nella funzione `h(x)` il parametro `x` è una variabile locale che oscura la `x` globale, e cambiarlo non cambia il valore fuori.

```
function h(x) {
  x = x * 2;
  return x;
}

console.log(h(3), x);                         // 6 11: la x globale è ancora 11
```

Regole pratiche:

- si preferiscono le variabili **locali** e i **parametri**: la funzione si capisce da sola, senza guardare il resto del programma;
- evitate di usare per una variabile locale il nome di una globale: funziona, ma è facile confondersi;
- una variabile globale che una funzione modifica "di nascosto" (come `g`) rende il programma difficile da seguire: se serve un valore, meglio passarlo come parametro e restituire il risultato con `return`.

Esercizio 6

Per ogni programma, dite cosa stampa **senza eseguirlo**; se c'è un errore, dite quale e perché. Poi eseguitelo per controllare.

A.

```
let x = 1;
function f() {
  let x = 2;
  console.log(x);
}
f();
console.log(x);
```

B.

```
let x = 1;
function g() {
  x = x + 10;
}
g();
g();
console.log(x);
```

C.

```
function raddoppia(n) {
  n = n * 2;
  return n;
}
let n = 5;
let r = raddoppia(n);
console.log(n, r);
```

D.

```
let x = 10;
if (x > 5) {
  let x = 20;
  x = x + 1;
  console.log(x);
}
console.log(x);
```

E.

```
for (let i = 0; i < 3; i++) {
  console.log(i);
}
console.log(i);
```

Esercizio 7

A. Cosa stampa questo programma? Se c'è un problema, dite quale e correggetelo in modo da stampare il risultato.

```
function calcola() {
  let risultato = 5 * 2;
}
calcola();
console.log(risultato);
```

B. Cosa stampa ciascuno di questi due programmi? Se c'è un problema, dite quale e perché.

```
// programma 1
let y = 3;
function stampa() {
  console.log(y);
}
stampa();
```

```
// programma 2
function stampa() {
  console.log(y);
}
stampa();
let y = 3;
```

C. La funzione dovrebbe restituire quanti numeri ci sono da `n` fino a `1` (quindi `conta(3)` dovrebbe dare `3`). Cosa restituisce davvero? Se non è il valore atteso, dite perché e correggetela.

```
function conta(n) {
  let passi = 0;
  if (n > 0) {
    passi = passi + 1;
    conta(n - 1);
  }
  return passi;
}
console.log(conta(3));
```

D. Questa funzione modifica la variabile **globale** `totale`. Riscrivetela in modo che **non** tocchi variabili globali: riceva il totale come parametro e **restituisca** il nuovo totale.

```
let totale = 0;
function aggiungi(x) {
  totale = totale + x;
}
aggiungi(3);
aggiungi(4);
console.log(totale);          // 7
```

Recap 4 · Cicli annidati

Un ciclo può contenere un altro ciclo. Il ciclo **interno** viene eseguito **per intero** a ogni giro del ciclo **esterno**. Esempio: le prime cinque righe delle tabelline del 2, del 3 e del 4.

```
for (let i = 2; i <= 4; i++) {          // esterno: una tabellina per giro (2, 3, 4)
  let riga = "";                        // si azzera a ogni giro esterno
  for (let j = 1; j <= 5; j++) {        // interno: 5 giri, ricomincia ogni volta
    riga = riga + (i * j) + " ";
  }
  console.log(riga);                    // una riga per ogni tabellina
}
```

```
2 4 6 8 10
3 6 9 12 15
4 8 12 16 20
```

Il ciclo esterno fa 3 giri (`i` vale `2` , `3` , `4`); a ognuno, il ciclo interno fa **tutti** i suoi 5 giri (`j` da `1` a `5`) e costruisce una riga.

- il corpo del ciclo interno viene eseguito (giri esterni) × (giri interni) volte: qui `3 * 5 = 15` ;
- le variabili dei due cicli (`i` , `j`) devono avere **nomi diversi**;
- le variabili che servono "da capo" a ogni giro esterno (come `riga`) vanno **inizializzate dentro** il ciclo esterno, prima del ciclo interno;
- il numero di giri interni può dipendere dalla variabile del ciclo esterno (ad esempio in un triangolo di asterischi, dove la riga `i` ne ha `i` : il ciclo interno va da `1` a `i`);
- per **seguire** un programma a mano, si scrive una tabella con i valori di `i` e `j` a ogni giro.

Esercizio 8

Per ogni programma, dite cosa stampa **senza eseguirlo** (aiutatevi con una tabella dei valori di `i` e `j`); poi eseguitelo per controllare.

A.

```
for (let i = 1; i <= 3; i++) {
  for (let j = 1; j <= 2; j++) {
    console.log(i, j);
  }
}
```

B.

```
for (let i = 1; i <= 3; i++) {
  let riga = "";
  for (let j = 1; j <= 3; j++) {
    riga = riga + (i * j) + " ";
  }
  console.log(riga);
}
```

C. Quanto vale `conta` alla fine? E se i due cicli arrivassero a `100` invece che a `4`?

```
let conta = 0;
for (let i = 1; i <= 4; i++) {
  for (let j = 1; j <= i; j++) {
    conta++;
  }
}
console.log(conta);
```

D. Questo programma doveva stampare un triangolo di asterischi, ma non fa quello che ci si aspetta. Cosa stampa? Come si corregge?

```
let riga = "";
for (let i = 1; i <= 3; i++) {
  for (let j = 1; j <= i; j++) {
    riga = riga + "*";
  }
  console.log(riga);
}
```

Esercizio 9

Chiedere all'utente un intero `n >= 1` (facendo ripetere la domanda finché non è valido) e stampare le seguenti figure. Per `n = 4`:

A. Un **quadrato** di asterischi, `n` righe di `n` asterischi:

```
****
****
****
****
```

B. Un triangolo **rovesciato**, con `n` asterischi nella prima riga e uno nell'ultima:

```
****
***
**
*
```

C. Un triangolo di **numeri**: nella riga `i` ci sono i numeri da `1` a `i`, separati da uno spazio:

```
1
1 2
1 2 3
1 2 3 4
```

D. Un triangolo **allineato a destra**, con `n` righe: nella riga `i` ci sono `n - i` spazi seguiti da `i` asterischi:

```
   *
  **
 ***
****
```

Suggerimento: per ogni figura chiedetevi quanti giri fa il ciclo interno **in funzione di `i`**; poi, se serve, costruite la riga come stringa.

Esercizio 10

La **tavola pitagorica** mostra i prodotti `i * j` per `i` e `j` da `1` a `n`.

A. Chiedere `n` all'utente e stampare la tavola, una riga per ogni `i`, con i numeri allineati in colonne. Ad esempio per `n = 5`:

```
   1   2   3   4   5
   2   4   6   8  10
   3   6   9  12  15
   4   8  12  16  20
   5  10  15  20  25
```

Ogni numero occupa una colonna larga **4 caratteri**, allineato a destra.

B. Stampare solo la parte "sopra la diagonale" (`j >= i`), lasciando vuote le caselle sotto. Per `n = 5`:

```
  1   2   3   4   5
      4   6   8  10
          9  12  15
             16  20
                 25
```

C. Calcolare e stampare la **somma di tutti i numeri della tavola** per `n = 10`. C'è una formula per ottenerla?

Esercizio 11

Si lanciano due dadi a sei facce. Ogni risultato è una **coppia** `(a, b)` con `a` e `b` da `1` a `6`.

A. Stampare **tutte** le coppie la cui somma è `7`, e dire quante sono. Quante coppie in tutto si controllano?

B. Chiedere all'utente una somma `s` (facendo ripetere la domanda finché non è un intero tra `2` e `12`), e stampare quante coppie danno quella somma. Provate con `2`, `7` e `12`.

C. Stampare, per ogni somma da `2` a `12`, **quante** coppie la producono. Quale somma è la più frequente?

Esercizio 12

Un numero `n >= 2` è **primo** se non ha divisori tra `2` e `n - 1`.

A. Stampare tutti i numeri primi da `2` a `50` e quanti sono. Per ogni `n` il ciclo interno cerca un divisore, usando una **variabile flag** (come nell'esercitazione 3).

B. Il ciclo interno continua a girare anche dopo aver trovato un divisore, ma non serve più. Aggiungete un `break` per fermarlo, e verificate con un contatore globale che il numero di giri interni **diminuisce**. Di quanto?

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