Capitolo 9

Funzioni

Introduzione

- Una funzione è una sequenza di istruzioni
  - Identificate (nel loro insieme) da un nome
- Ogni funzione è essenzialmente un piccolo programma
  - Con proprie dichiarazioni di variabili e istruzioni

Introduzione

- Vantaggi delle funzioni:
  - Un programma può essere diviso in moduli
    - Più facile da capire e da modificare
  - Codice comune a varie parti del programma non ha necessità di essere duplicato
  - Una funzione sviluppata per un programma può essere riutilizzata anche per un altro programma

Definizione e chiamata di funzione

- Prima di analizzare le regole formali per la definizione di una funzione, guardiamo 3 programmi semplici che definiscono delle funzioni

- Calcolo della media
- Conto alla rovescia
- Gioco di parole

Programma: calcolo della media

- La funzione `average` calcola la media di due `double`:

```
double average(double a, double b)
{
  return (a + b) / 2;
}
```

- La parola `double` all'inizio della definizione specifica il ***tipo del valore restituito da*** `average`.

- Gli identificatori `a` e `b` (i ***parametri*** della funzione) rappresentano i valori che verranno forniti in input

Programma: calcolo della media

- Ogni funzione ha una parte eseguibile, chiamata il ***corpo,*** che è racchiusa fra parentesi graffe

- Il corpo della funzione `average` è molto semplice: contiene solo l'istruzione `return`

- L'esecuzione dell'istruzione `return` restituisce il controllo al programma chiamante

- Il valore `(a + b) / 2` sarà il valore "restituito" dalla funzione

Programma: calcolo della media

- Una chiamata alla funzione è specificata scrivendo il nome della funzione e una lista di ***argomenti***:
  - `average(x, y)`
- Gli argomenti forniscono l'input alla funzione
  - La chiamata `average(x, y)` fa sì che i valori di `x` e `y` vengano ***copiati*** nei parametri definiti dalla funzione (le variabili `a` e `b`)
- Un argomento non deve necessariamente essere una variabile; una qualsiasi espressione va bene:
  `average(5.1, 8.9)`
  `average(x/2, y/3)` sono chiamate corrette

Programma: calcolo della media

- Useremo una chiamata alla funzione `average` nei punti in cui ci serve il valore medio calcolato dalla funzione
- Ecco un'istruzione che stampa la media di `x` e `y`:

```
printf("Average: %g\n", average(x, y));
```

Il valore di ritorno di `average` non viene salvato ma solo stampato

- Se abbiamo bisogno di memorizzare il valore dobbiamo assegnarlo ad una variabile:

```
avg = average(x, y);
```

Programma: calcolo della media

- Il programma `average.c` legge 3 numeri ed usa la funzione `average` per calcolare la media di ogni coppia di due numeri:

```
Enter three numbers: 3.5 9.6 10.2
Average of 3.5 and 9.6: 6.55
Average of 9.6 and 10.2: 9.9
Average of 3.5 and 10.2: 6.85
```

average.c

```
/* Computes pairwise averages of three numbers */

#include <stdio.h>

double average(double a, double b)
{
  return (a + b) / 2;
}

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}
```

Programma: conto alla rovescia

- Per specificare che una funzione non restituisce nessun valore si usa il tipo `void`:

```
void print_count(int n)
{
  printf("T minus %d and counting\n", n);
}
```

- Una chiamata a `print_count` deve essere scritta come istruzione a sé:

```
print_count(i);
```

- Il programma `countdown.c` chiama la funzione `print_count` 10 volte nel ciclo

countdown.c

```
/* Prints a countdown */

#include <stdio.h>

void print_count(int n)
{
  printf("T minus %d and counting\n", n);
}

int main(void)
{
  int i;

  for (i = 10; i > 0; --i)
    print_count(i);

  return 0;
}
```

Programma: gioco di parole (rivisitato)

- Quando una funzione non ha parametri, la parola `void` viene messa al posto della lista dei parametri:

```
void print_pun(void)
{
  printf("To C, or not to C: that is the question.\n");
}
```

- Per una chiamata ad una funzione senza parametri si usa il nome della funzione seguito da parentesi tonde senza niente dentro:

```
print_pun();
```

  Le parentesi *devono* esserci

- Il programma `pun2.c` usa la funzione `print_pun`

pun2.c

```
/* Prints a bad pun */

#include <stdio.h>

void print_pun(void)
{
  printf("To C, or not to C: that is the question.\n");
}

int main(void)
{
  print_pun();
  return 0;
}
```

Definizione di funzioni

- La forma generale di una ***definizione di funzione*** è:

```
return-type  function-name (parameters)
{
    declarations
    statements
}
```

Definizione di funzioni

- Il tipo della funzione è il tipo del valore restituito dalla funzione
- Nota:
  - Le funzioni **non** possono restituire array
  - Specificare che il tipo è `void` significa dire che la funzione non restituisce nessun valore
- Se la dichiarazione di tipo viene omessa, nel C89, la funzione ha tipo `int`
- Nel C99, non è ammesso omettere il tipo nella dichiarazione della funzione.

Definizione di funzioni

- Alcuni programmatori scrivono il tipo sulla linea precedente il nome della funzione:

```
double
average(double a, double b)
{
  return (a + b) / 2;
}
```

- È una questione di stile di scrittura
- Può essere utile quando il tipo è molto lungo, come ad esempio `unsigned long int`.

Definizione di funzioni

- Dopo il nome della funzione bisogna specificare, fra parentesi tonde, la lista dei parametri
- Ogni parametro è preceduto dalla dichiarazione del suo tipo
- Se ci sono più parametri, vengono separati da una virgola
- Se non ci sono parametri viene scritta la parola `void`

Definizione di funzioni

- Il corpo di una funzione può includere sia istruzioni che dichiarazioni di variabili.
- Una versione alternativa della funzione `average`:

```
double average(double a, double b)
{
  double sum;          /* declaration */

  sum = a + b;         /* statement */
  return sum / 2;      /* statement */
}
```

Definizione di funzioni

- Le variabili dichiarate in una funzione sono ***locali*** alla funzione
  - Non possono essere esaminate e/o modificate al di fuori della funzione; cioè non esistono al di fuori della funzione
- Nel C89 le dichiarazioni delle variabili locali devono essere messe all'inizio della funzione, prima del corpo

- Nel C99 questa restrizione è stata tolta. Possono apparire ovunque

Definizione di funzioni

- Il corpo di una funzione il cui tipo è `void` può essere vuoto:

```
void print_pun(void)
{
}
```

- Una tale situazione può avere senso durante lo sviluppo di un programma:
  - Abbiamo previsto la funzione ma non abbiamo ancora scritto il codice; possiamo comunque compilare il programma

Chiamate di funzioni

- Una chiamata di funzione si effettua specificando il nome della funzione seguito da una lista di argomenti fra parentesi tonde:

```
average(x, y)
print_count(i)
print_pun()
```

Chiamate di funzioni

- Una chiamata ad una funzione `void` deve necessariamente essere eseguita come istruzione singola:

```
print_count(i);
print_pun();
```

- Una chiamata ad una funzione non-`void` produce un valore che può essere usato in una qualsiasi espressione:

```
avg = average(x, y);
if (average(x, y) > 0)
  printf("Average is positive\n");
printf("The average is %g\n", average(x, y));
```

Chiamate di funzioni

- Il valore restituito da una funzione non-`void` può essere ignorato se non serve:

```
average(x, y);  /* discards return value */
```

Chiamate di funzioni

- Ad esempio, solitamente il valore restituito dalla funzione `printf`, che è il numero di caratteri stampati, viene ignorato

- Questa chiamata a `printf` restituisce il valore 9, `num_chars` varrà quindi 9:

```
num_chars = printf("Hi, Mom!\n");
```

- Solitamente useremo la `printf` ignorando il valore di ritorno:

```
printf("Hi, Mom!\n”)
  /* discards return value */
```

Programma: numeri primi

- Il programma `prime.c` controlla se un numero, fornito in input, è primo:

```
Enter a number: 34
Not prime
```

- Il programma usa una funzione chiamata `is_prime` che restituisce `true` se il parametro è primo e `false` se non lo è
- `is_prime` divide il parametro `n` per ogni numero intero fra 2 e la radice quadrata di `n`; se almeno una volta il resto è 0, `n` non è primo

prime.c

```
/* Tests whether a number is prime */

#include <stdbool.h>   /* C99 only */
#include <stdio.h>

bool is_prime(int n)
{
  int divisor;

  if (n <= 1)
    return false;
  for (divisor = 2; divisor * divisor <= n; divisor++)
    if (n % divisor == 0)
      return false;
  return true;
}
```

```
int main(void)
{
  int n;

  printf("Enter a number: ");
  scanf("%d", &n);
  if (is_prime(n))
    printf("Prime\n");
  else
    printf("Not prime\n");
  return 0;
}
```

Dichiarazione di funzioni

- Il C **non** richiede che la definizione di una funzione preceda il suo utilizzo

Dichiarazione di funzioni

```
#include <stdio.h>

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}

double average(double a, double b)
{
  return (a + b) / 2;
}
```

Problema …

Dichiarazione di funzioni

- Il C permette di ***dichiarare*** una funzione prima di definirla:
  - La definizione potrà essere messa dopo
- Una ***dichiarazione di funzione*** fornisce al compilare le informazioni di cui ha bisogno
- La forma generale di una dichiarazione è:

  return-type function-name (*parameters*) **;**
- Ovviamente la **dichiarazione** deve essere **consistente** con la successiva **definizione** della funzione
- Ecco il programma `average.c` con una dichiarazione della funzione `average`

Dichiarazione di funzioni

```
#include <stdio.h>

double average(double a, double b);   /* DECLARATION */

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}

double average(double a, double b)    /* DEFINITION */
{
  return (a + b) / 2;
}
```

Dichiarazione di funzioni

- Queste dichiarazioni di funzioni sono dette ***prototipi di funzione***

- Un prototipo di funzione non deve necessariamente specificare i nomi dei parametri, ma basta mettere i tipi:

```
double average(double, double);
```

- È prassi comune non omettere i nomi

Argomenti

- In C gli argomenti delle funzioni vengono ***passati per valore***: quando una funzione viene chiamata, ogni argomento viene valutato ed il valore viene assegnato al parametro corrispondente

- Poichè il parametro contiene una **copia** del valore, qualunque cambiamento fatto all'interno della funzione **non avrà effetto** sull'argomento passato alla funzione

Argomenti

- Il passaggio per valore degli argomenti ha vantaggi e svantaggi

- Poichè i parametri possono essere modificati senza che gli argomenti siano cambiati, possiamo usare i parametri come variabili locali

Argomenti

- Consideriamo la seguente funzione che calcola la potenza `n`-esima di un numero `x` :

```
int power(int x, int n)
{
  int i, result = 1;

  for (i = 1; i <= n; i++)
    result = result * x;

  return result;
}
```

Argomenti

- Poichè `n` è una *copia* dell'esponente originale (argomento della chiamata) possiamo modificarlo senza avere effetti altrove; pertanto non ci servirà la variabile `i`:

```
int power(int x, int n)
{
  int result = 1;

  while (n-- > 0)
    result = result * x;

  return result;
}
```

Argomenti

- Il passaggio per valore crea delle difficoltà per certi tipi di funzioni
- Supponiamo di aver bisogno di una funzione che decompone un `double` nella sua parte intera ed in quella frazionaria
- Poichè la funzione **non può restituire due valori,** potremmo voler passare alla funzione due variabili e farle modificare dalla funzione stessa.

```
void decompose(double x, long int_part,
               double frac_part)
{
  int_part = (long) x;
  frac_part = x - int_part;
}
```

Argomenti

- Una chiamata alla funzione del tipo:

```
decompose(3.14159, i, d);
```

non produrrà l'effetto desiderato: ad `i` e `d` non verrano assegnati i valori di `int_part` e `frac_part`.

- Vedremo come risolvere questo problema in seguito.

Array come argomenti

- Quando il parametro di una funzione è un array mono-dimensionale, la lunghezza dell'array non viene specificata:

```
int f(int a[])  /* no length specified */
{
  …
}
```

- Il C non fornisce un modo semplice per determinare la lunghezza di un array passato come argomento
- Se serve è necessario passarla come argomento a parte

Array come argomenti

- Esempio:

```
int sum_array(int a[], int n)
{
  int i, sum = 0;

  for (i = 0; i < n; i++)
    sum += a[i];

  return sum;
}
```

- Poichè la funzione `sum_array` deve sapere la lunghezza dell'array `a`, dobbiamo fornirla attraverso un altro parametro

Array come argomenti

- Il prototipo della funzione `sum_array` ha la forma seguente:

```
int sum_array(int a[], int n);
```

Array come argomenti

- Quando `sum_array` viene chiamata il primo argomento sarà il nome dell'array, mentre il secondo sarà la sua lunghezza:

```
#define LEN 100

int main(void)
{
  int b[LEN], total;
  …
  total = sum_array(b, LEN);
  …
}
```

- Si noti che nella chiamata **non** si mettono le parentesi quadre per passare l'array:

```
total = sum_array(b[], LEN);    /*** WRONG ***/
```

Array come argomenti

- Una funzione non ha modo di sapere se il parametro passato come lunghezza è corretto
- Possiamo sfruttare questo fatto per far credere alla funzione che l'array sia più piccolo della sua reale dimensione
- Supponiamo di aver memorizzato solo 50 numeri nell'array `b` anche se l'array ne può contenere 100.
- Allora possiamo sommare solo i primi 50 elementi chiamando la funzione in questo modo

```
total = sum_array(b, 50);
```

Array come argomenti

- Attenzione: non far credere ad una funzione che l'array passato sia più *grande* della sue reali dimensioni

```
total = sum_array(b, 150);   /*** DANGER
***/
```

In questo caso la funzione `sum_array` leggerà oltre la fine dell'array, causando un comportamento imprevedibile

Array come argomenti

- A differenza dei parametri che sono singole variabili, una funzione **può cambiare i valori degli elementi di un array**
  - Cambiamenti fatti nella funzione si riflettono nell'array originale
- Ecco una funzione che modifica l'array passato come argomento scrivendo 0 in ogni posizione:

```
void store_zeros(int a[], int n)
{
  int i;

  for (i = 0; i < n; i++)
    a[i] = 0;
}
```

Array come argomenti

- Ecco una chiamata a `store_zeros`:

```
store_zeros(b, 100);
```

- La capacità di modificare gli elementi di un array **sembra** contraddire il fatto che gli argomenti sono passati per valore

- Vedremo che non c'è contraddizione
  - Il passaggio degli argomenti avviene per valore

Array come argomenti

- Se un parametro è un array con più di una dimensione **le lunghezze delle altre dimensioni** vanno specificate:
- Ecco una versione di `sum_array` per una array bi-dimensionale. Dobbiamo specificare il numero di colonne:

```
#define LEN 10

int sum_two_dimensional_array(int a[][LEN], int n)
{
  int i, j, sum = 0;

  for (i = 0; i < n; i++)
    for (j = 0; j < LEN; j++)
      sum += a[i][j];

  return sum;
}
```

L'istruzione `return`

- Serve per specificare il valore di ritorno
- L'istruzione `return` ha la seguente forma

  `return` *expression* `;`

- L'espressione è spesso una costante o una variabile:

```
return 0;
return status;
```

- È possibile usare una qualsiasi espressione:

```
return n >= 0 ? n : 0;
```

L'istruzione `return`

- Se il tipo dell'espressione di un `return` non corrisponde al tipo della funzione, essa sarà implicitamente convertita al tipo della funzione

- Esempio: se la funzione restituisce un `int`, ma il `return` fornisce un `double`, il valore verrà convertito in `int`.

L'istruzione `return`

- `return` può apparire alla fine di una funzione `void`, anche se non serve:

```
void print_pun(void)
{
  printf("To C, or not to C: that is the question.\n");
  return;   /* OK, but not needed */
}
```

- Se una funzione non-`void` non esegue l'istruzione `return`, il comportamento del programma non è definito se viene chiamata la funzione

Terminazione di un programma

- Normalmente, il tipo della funzione `main` è `int`:

```
int main(void)
{
  …
}
```

- Molti programmi C (vecchi) omettono il tipo della funzione `main` poichè il suo valore di default è `int`:

```
main()
{
  …
}
```

Terminazione di un programma

- In C99 non è permesso omettere il tipo della funzione `main`, quindi è buona norma specificarlo
- Omettere invece la parola `void` al posto della lista dei parametri è permesso

- `int main(void)`

- `int main()`

Terminazione di un programma

- Il valore restituito da `main` è un codice (*status code*) che può essere controllato quando il programma termina
- È prassi comune far restituire dal `main` il valore 0 se il programma termina in modo normale
- Per indicare una terminazione anormale il `main` ritorna un valore diverso da 0
- È buona prassi usare sempre un valore di ritorno (*status code*) per il `main`

La funzione `exit`

- Eseguire un `return` dalla funzione `main` è uno dei modi per terminare un programma
- Un altro modo è quello fornito dalla funzione `exit`, che fa parte della libreria standard `<stdlib.h>`
- L'argomento passato alla funzione `exit` ha lo stesso significato del valore di ritorno del `main`: è il codice di terminazione (*status code*)
- Esempio:

```
exit(0);    /* normal termination */
```

La funzione `exit`

- Poiché 0 non è molto esplicativo, possiamo usare `EXIT_SUCCESS` (l'effetto è lo stesso)

```
exit(EXIT_SUCCESS);
```

- La costante `EXIT_FAILURE` indica una terminazione anormale

```
exit(EXIT_FAILURE);
```

- `EXIT_SUCCESS` e `EXIT_FAILURE` sono macro definite in `<stdlib.h>`.
- I valore `EXIT_SUCCESS` e `EXIT_FAILURE` sono dipendenti dall'implementazione. Tipicamente valgono, rispettivamente, 0 e 1

La funzione `exit`

- L'istruzione

  `return` *expression*`;`

  nel `main` è equivalente a

  `exit(`*expression*`);`
- La differenza fra `return` e `exit` è che `exit` causa la terminazione del programma **da qualunque funzione** la si chiami
- Il `return` invece fa terminare **solo la funzione** che lo chiama (quindi il programma nel caso venga chiamata dal `main`)

Ricorsione

- Una funzione è ***ricorsiva*** se chiama se stessa

- La seguente funzione calcola $n!$ ricorsivamente usando la formula $n! = n \times (n-1)!$

```
int fact(int n)
{
  if (n <= 1)
    return 1;
  else
    return n * fact(n - 1);
}
```

Ricorsione

- Per vedere come opera la ricorsione, vediamo cosa succede con la chiamata:

```
i = fact(3);
```

`fact(3)` vede che 3 non è <= 1, quindi chiama
  `fact(2)`, che vede che 2 non è <= 1, quindi chiama
    `fact(1)`, che vede che 1 è <= 1 quindi restituisce 1 a
  `fact(2)`, che restituisce 2 × 1 = 2 a
`fact(3)`, che restituisce 3 × 2 = 6

Ricorsione

- La seguente funzione ricorsiva calcola x^n, usando la formula $x^n = x \times x^{n-1}$.

```
int power(int x, int n)
{
  if (n == 0)
    return 1;
  else
    return x * power(x, n - 1);
}
```

Ricorsione

- Sia `fact` che `power` hanno come punto fermo il controllo di una **condizione di "terminazione"**

- Tutte le funzioni ricorsive hanno bisogno di questo tipo di controllo per prevenire una ricorsione infinita

L'algoritmo Quicksort

- La ricorsione è molto utile per implementare algoritmi **non banali** che richiedono la risoluzione dello stesso problema di partenza su istanze più piccole
- La tecnica di progettazione di algoritmi conosciuta come ***divide-et-impera,*** in cui un problema è diviso in sottoproblemi più piccoli che possono essere risolti dalla stesso algoritmo, genera algoritmi ricorsivi

L'algoritmo Quicksort

- Un esempio classico di algoritmo basato sul divide-et-impera è l'algoritmo ***Quicksort***
- Un array, indicizzato da 1 a *n*, deve essere ordinato

Algoritmo Quicksort

1. Per ordinare l'array, scegli un elemento *e* dell'array (il *pivot*), riorganizza l'array in modo tale che
   1. gli elementi con indice 1, …, $i - 1$ siano minori o uguali ad *e*
   2. la posizione *i* contenga *e*
   3. gli elementi con indice $i + 1$, …, *n* siano maggiori o uguali ad *e*.
2. Ordina gli elementi da 1 a $i - 1$ usando Quicksort
3. Ordina gli elementi da $i + 1$ a *n* usando Quicksort.

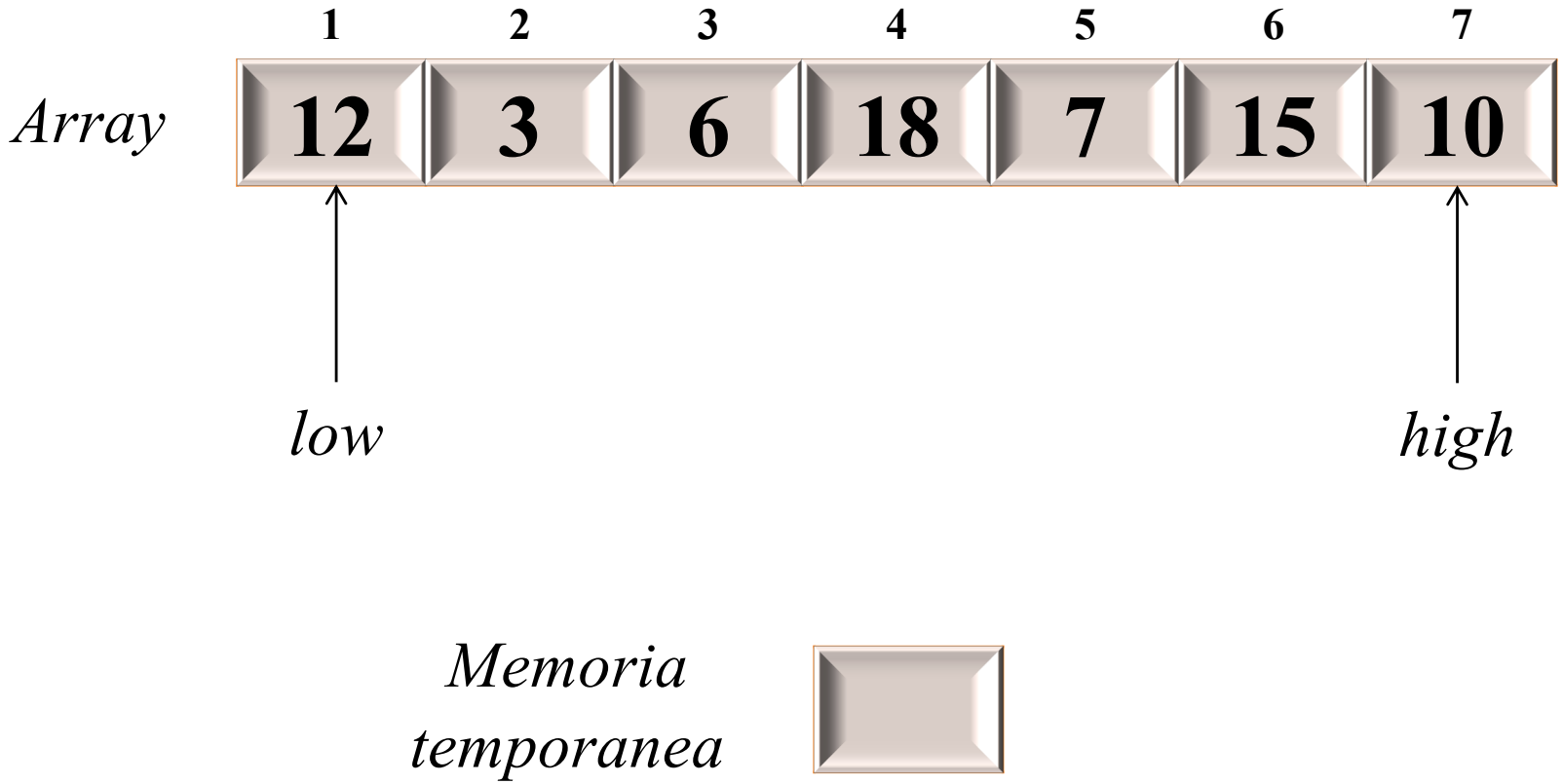
L'algoritmo Quicksort

- Il passo 1 (la "partizione") di Quicksort è fondamentale
- Ci sono vari modi per effettuarlo
- Utilizzeremo una tecnica facile da capire, anche se non particolarmente efficiente
- Usa due indici chiamati *low* e *high*, che tengono traccia delle posizioni all'interno dell'array

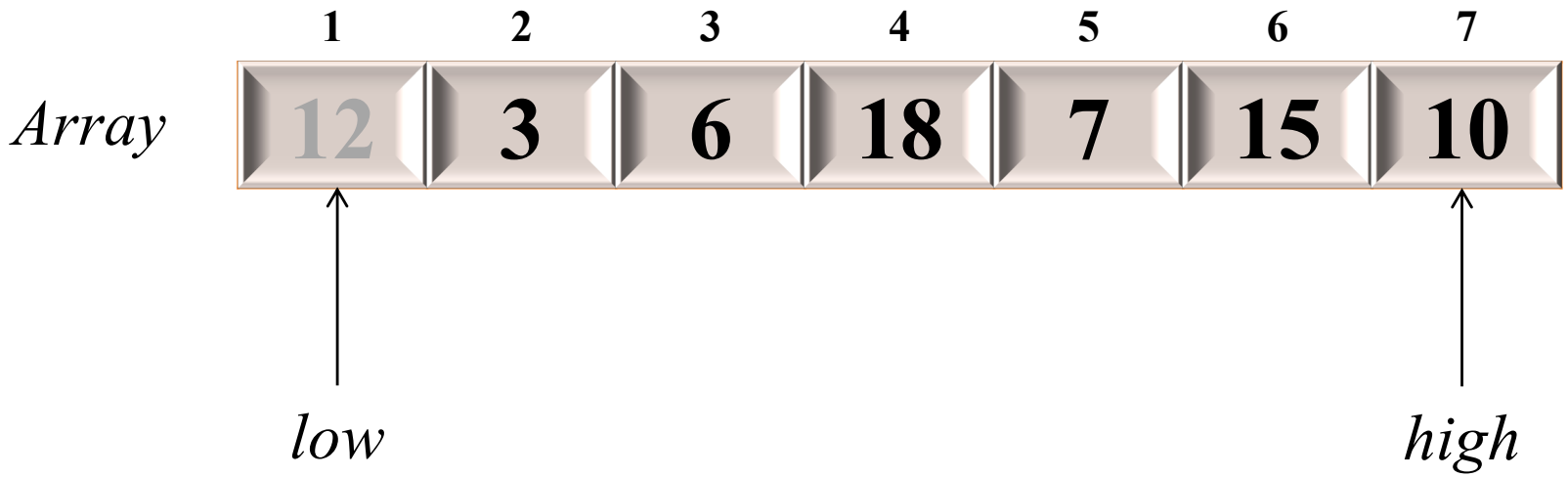
L'algoritmo Quicksort

- Inizialmente *low* indica il primo elemento e *high* l'ultimo
- Copiamo il primo elemento (il pivot) in una locazione temporanea esterna, lasciando un "buco" nell'array
- Quindi spostiamo l'indice *high* da destra a sinistra fino a trovare un elemento che è più piccolo del pivot
- Copiamo l'elemento trovato nel "buco" indicato da *low*. Ciò crea un "nuovo" buco, nella posizione indicata da *high*
- Spostiamo l'indice *low* da sinistra a destra fino a trovare un elemento che è maggiore del pivot
- Copiamo l'elemento trovato nel buco indicato da *high*
- Il processo si ripete fino a che *low* e *high* indicano entrambi la stessa posizione (buco)
- A questo punto copiamo il pivot nel buco indicato da *low* e *high*

L'algoritmo Quicksort: Esempio

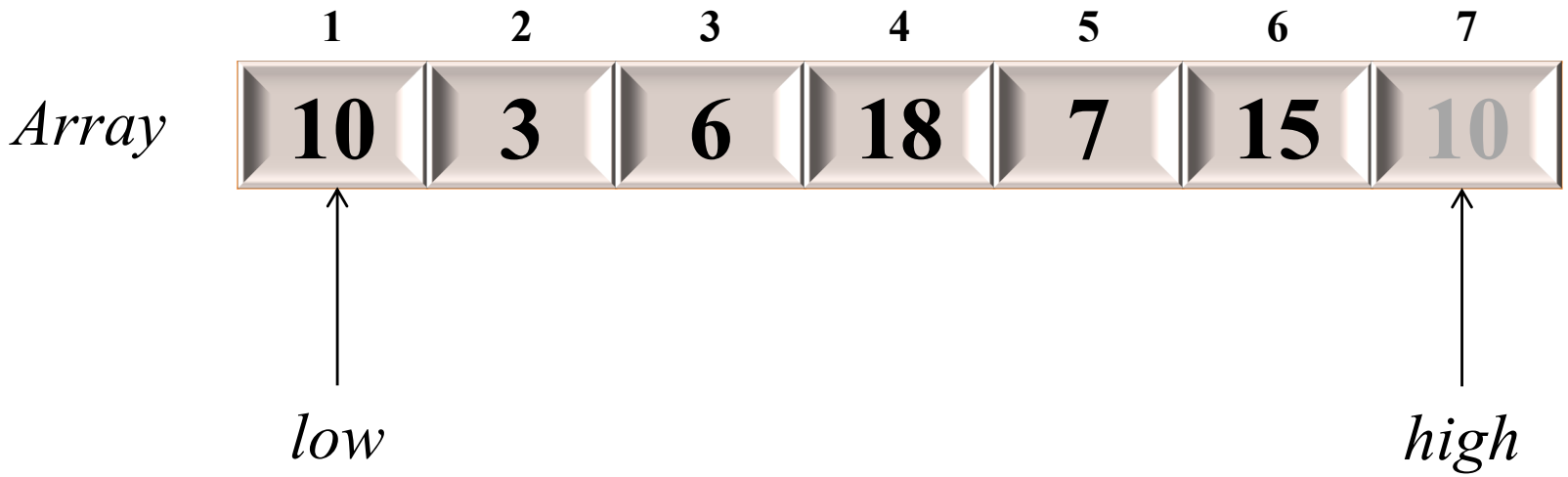


L'algoritmo Quicksort: Esempio



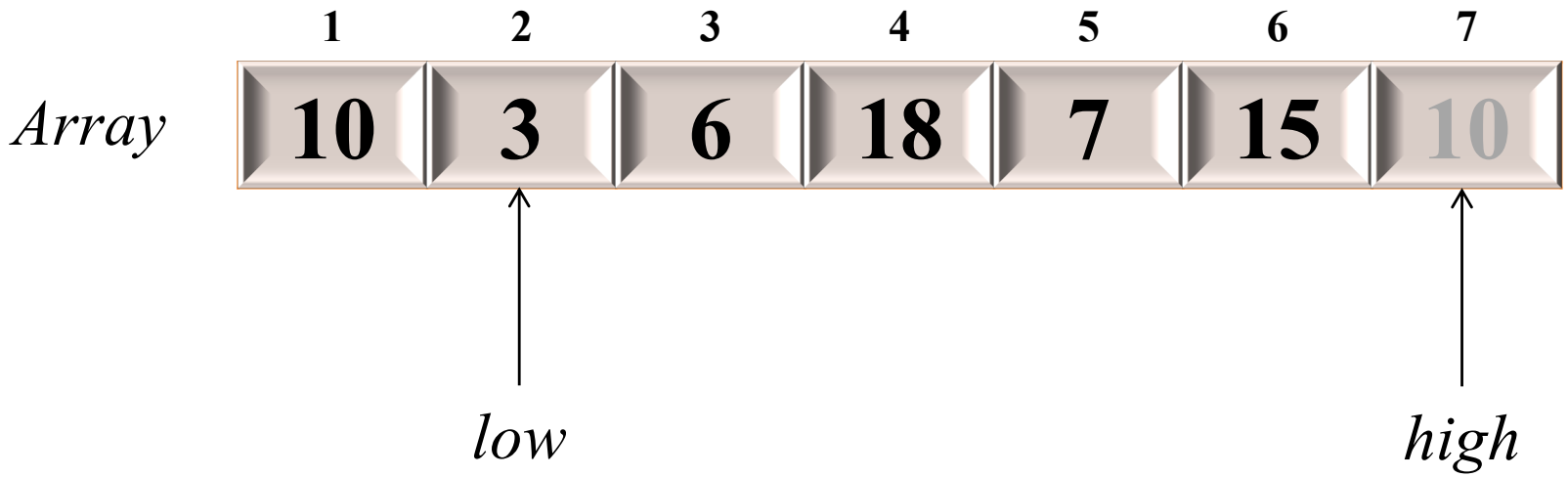
Memoria temporanea

L'algoritmo Quicksort: Esempio



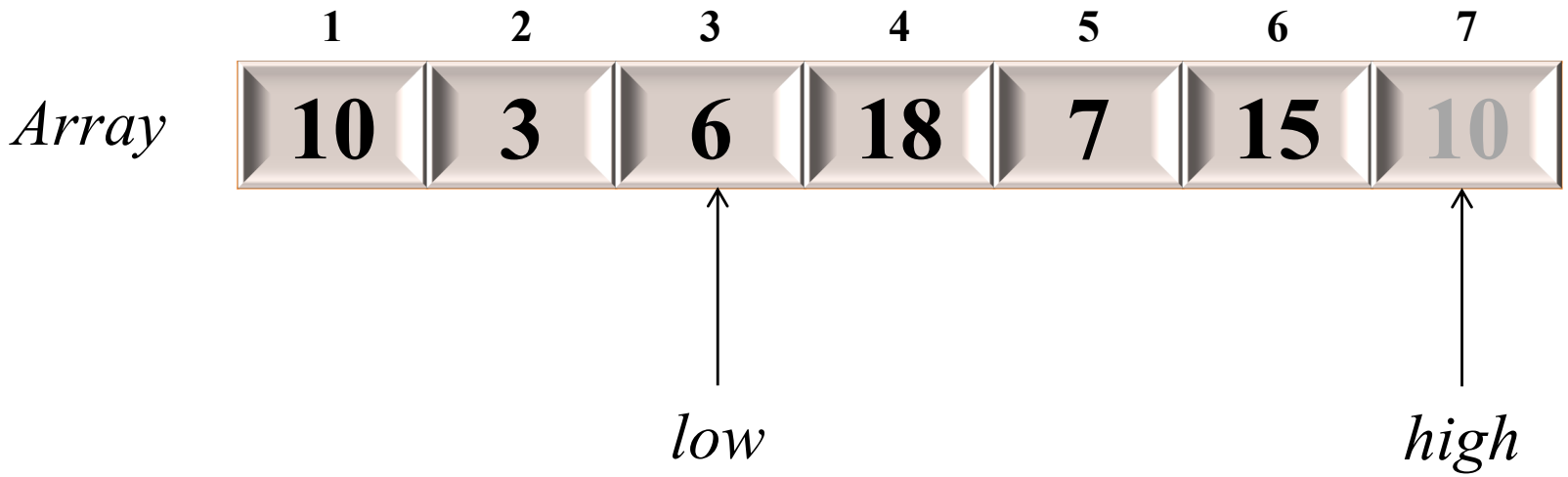
Memoria temporanea

L'algoritmo Quicksort: Esempio



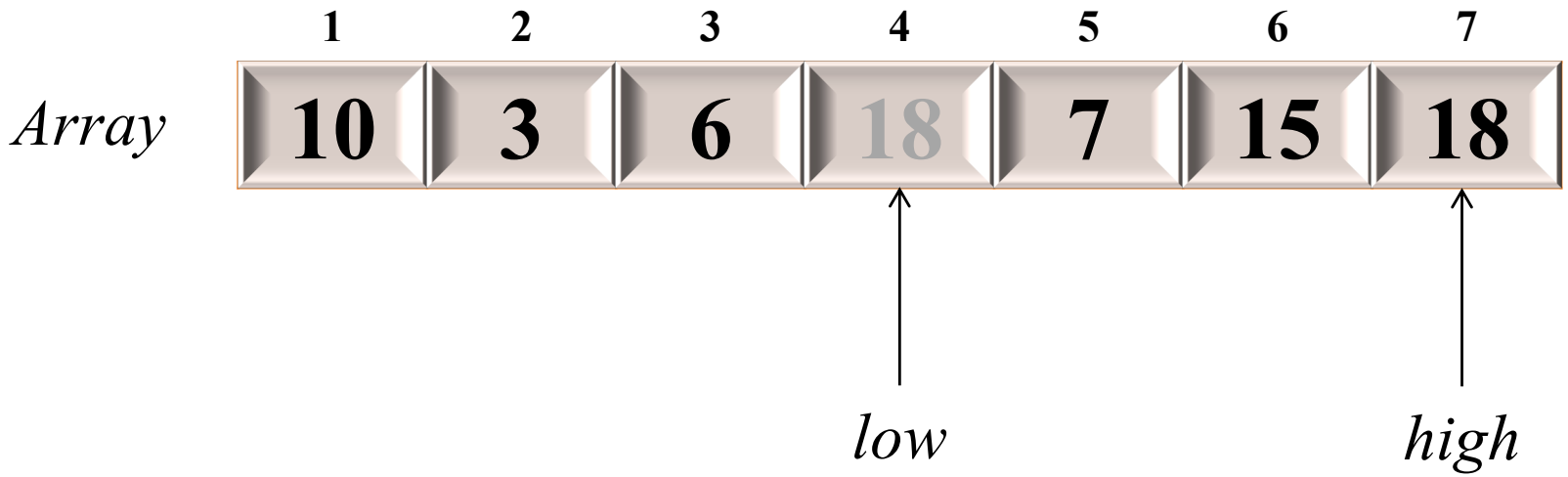
Memoria temporanea

L'algoritmo Quicksort: Esempio



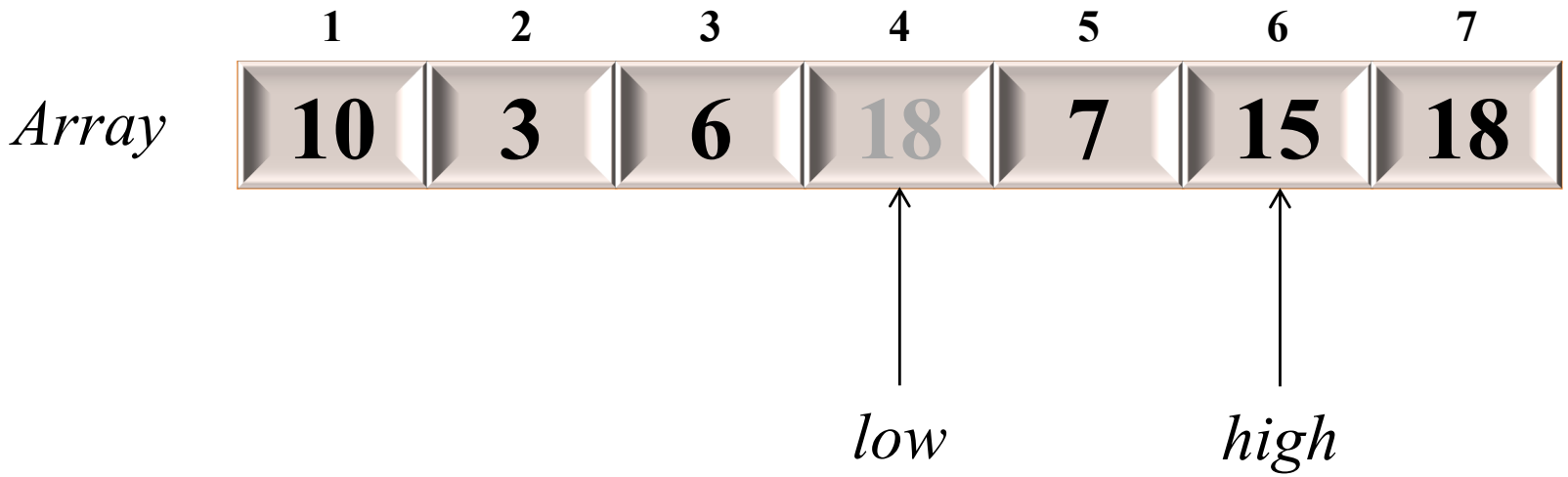
Memoria temporanea

L'algoritmo Quicksort: Esempio



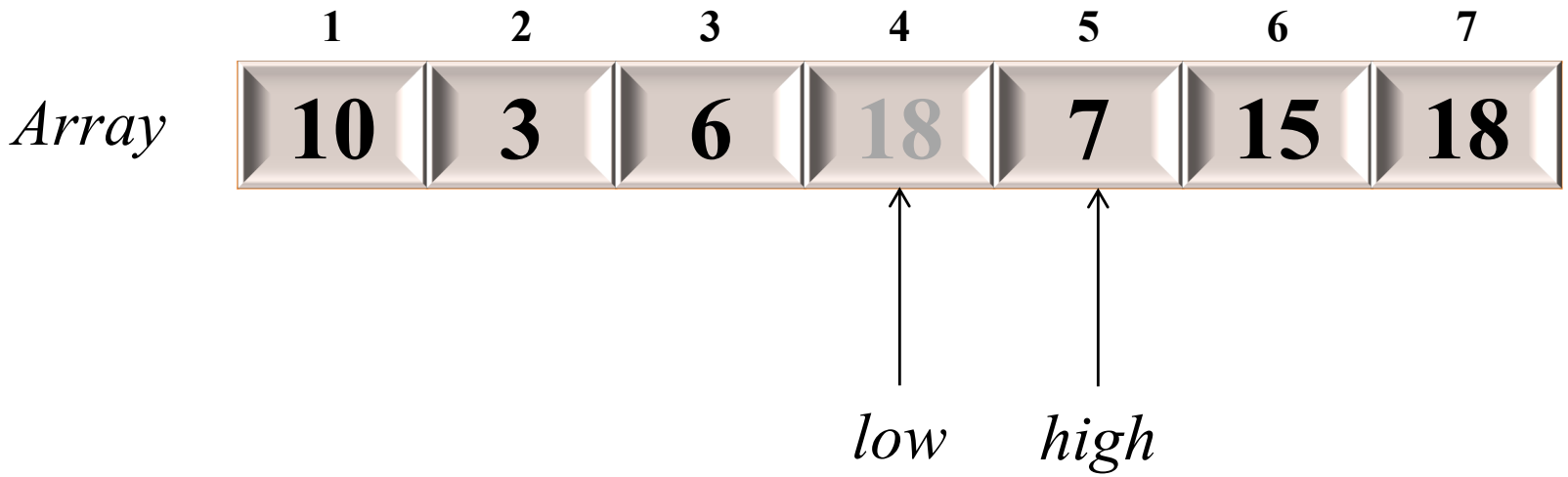
Memoria temporanea

L'algoritmo Quicksort: Esempio



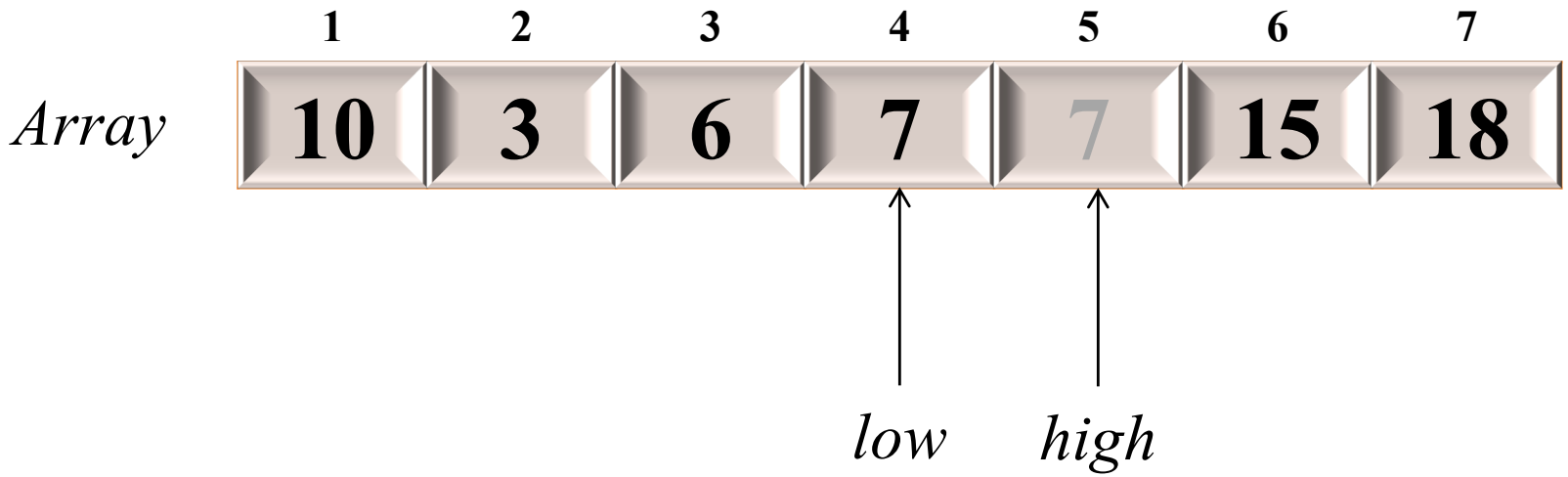
Memoria temporanea

L'algoritmo Quicksort: Esempio



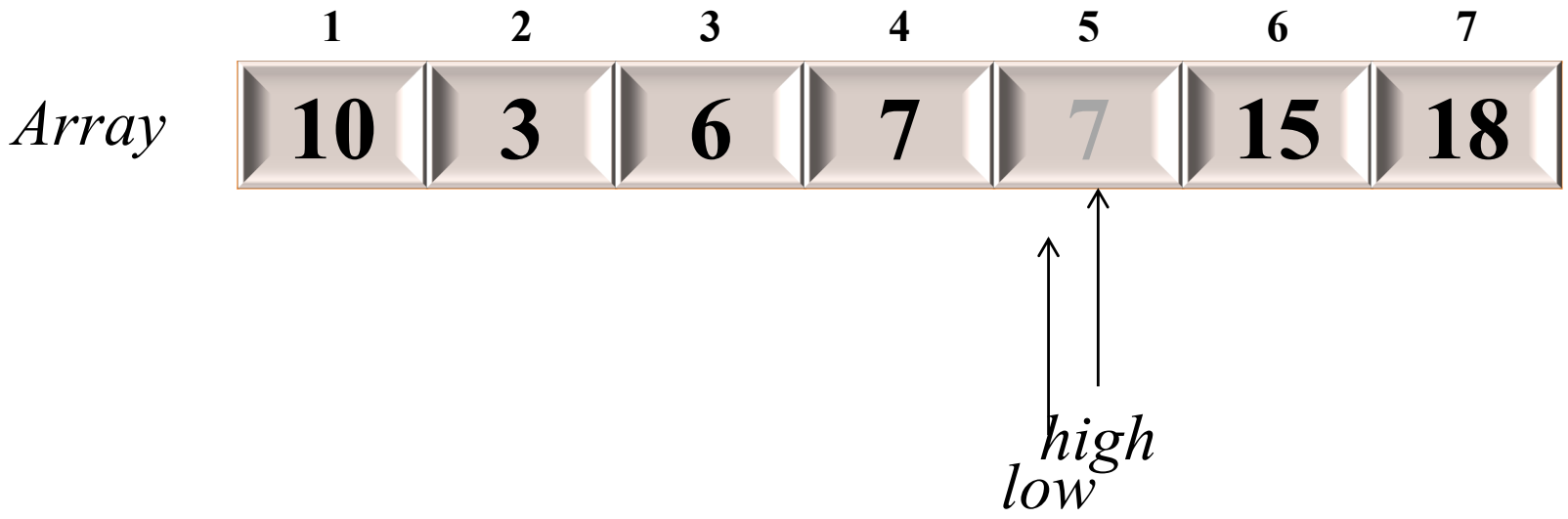
Memoria temporanea

L'algoritmo Quicksort: Esempio



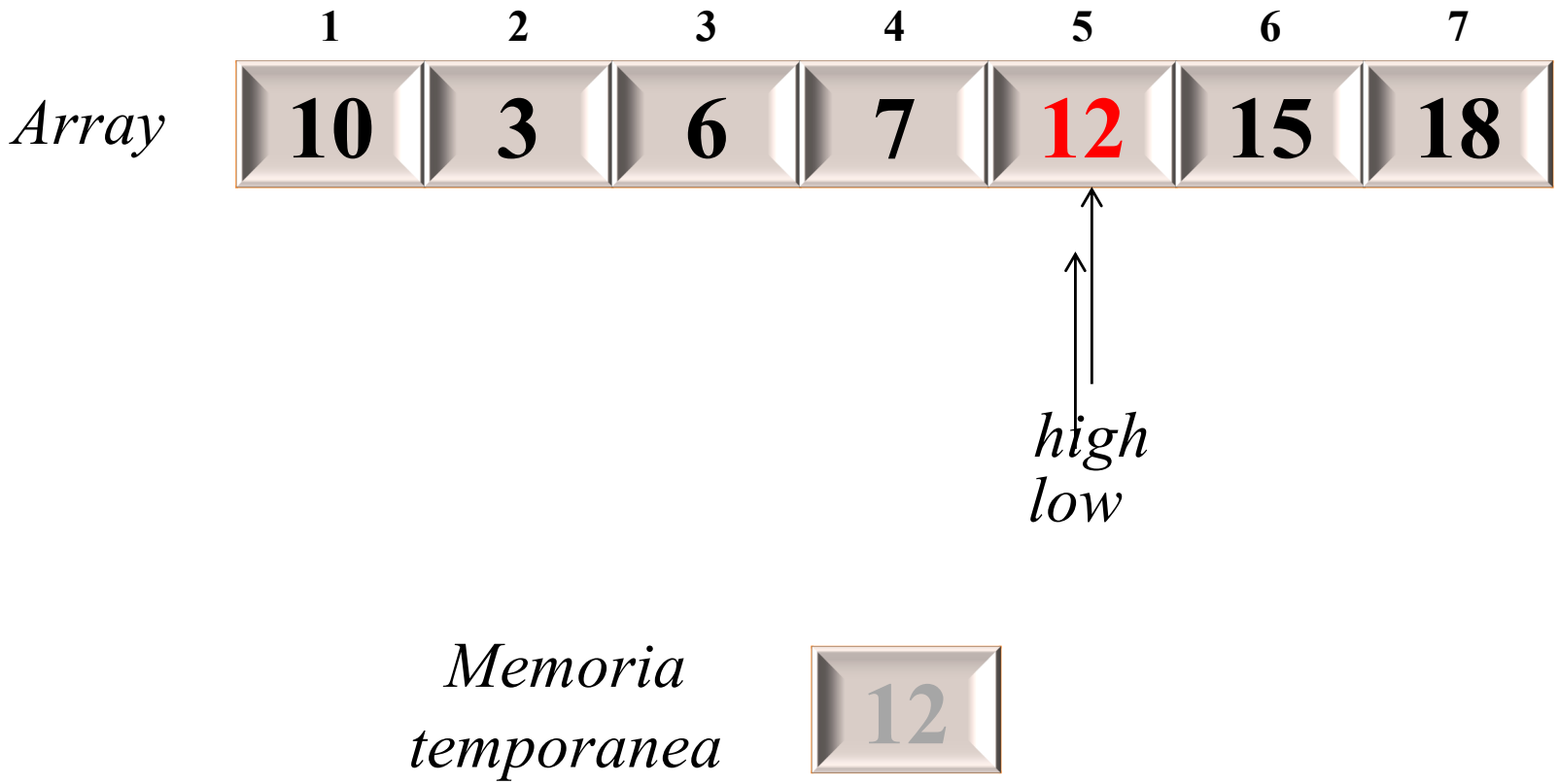
Memoria temporanea

L'algoritmo Quicksort: Esempio

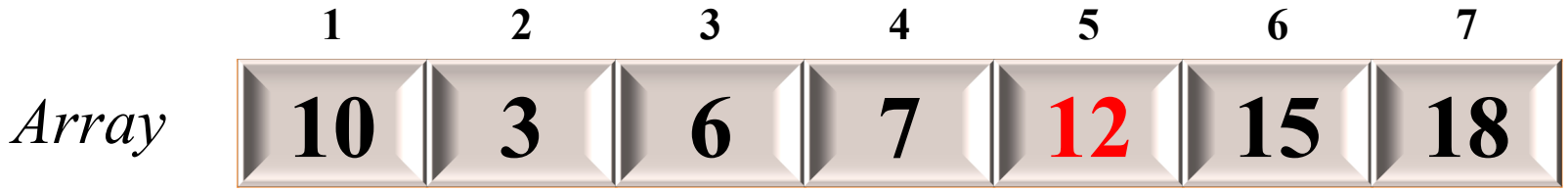


Memoria temporanea

L'algoritmo Quicksort: Esempio

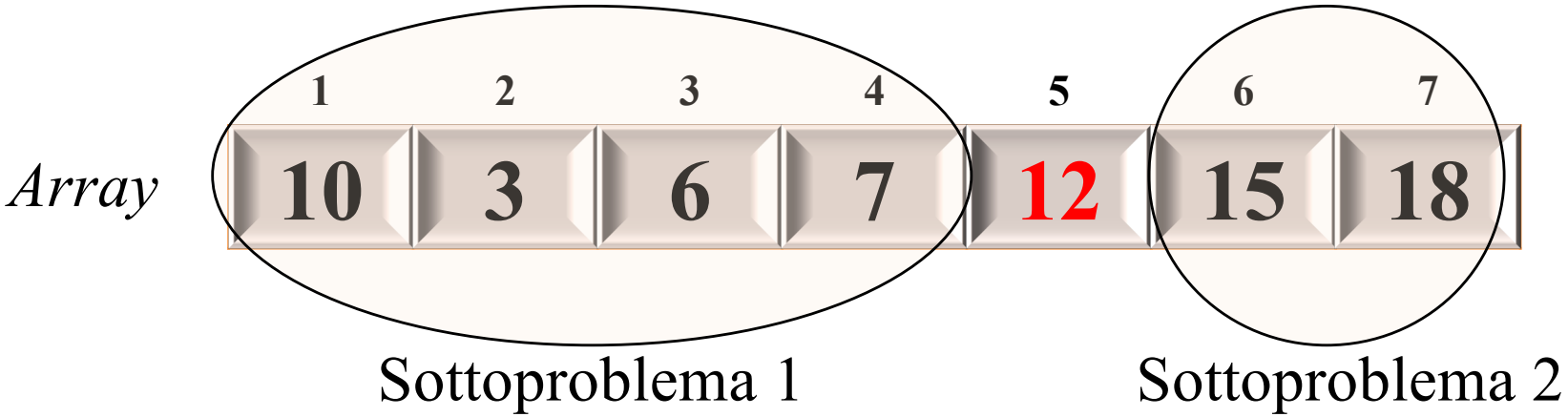


L'algoritmo Quicksort: Esempio



Tutti gli elementi alla sinistra del pivot (che è nella posizione 5) sono minori o uguali a 12, mentre tutti quelli alla destra sono maggiori o uguali a 12

L'algoritmo Quicksort: Esempio

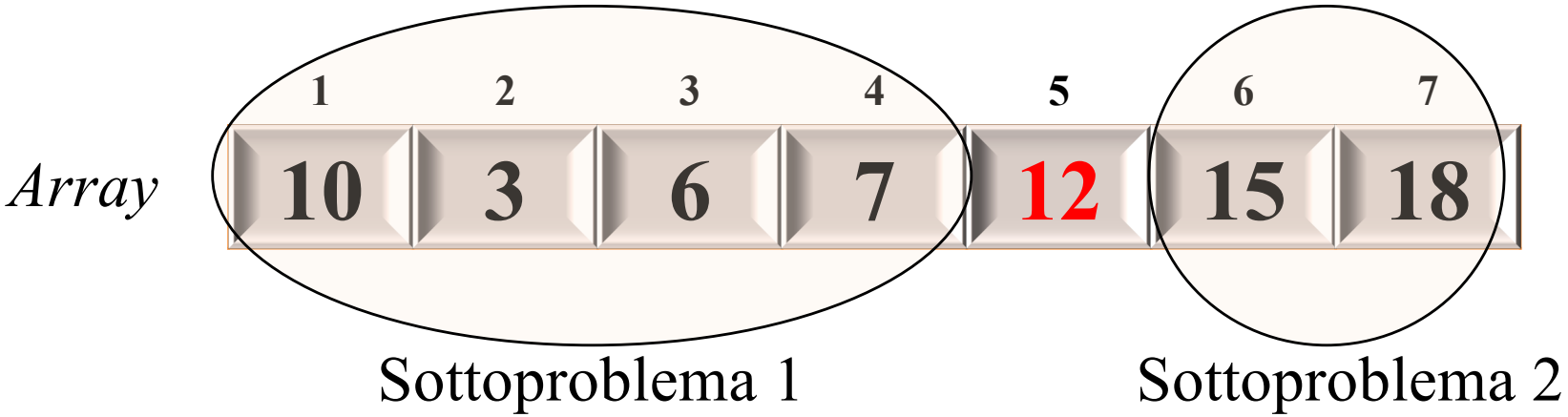


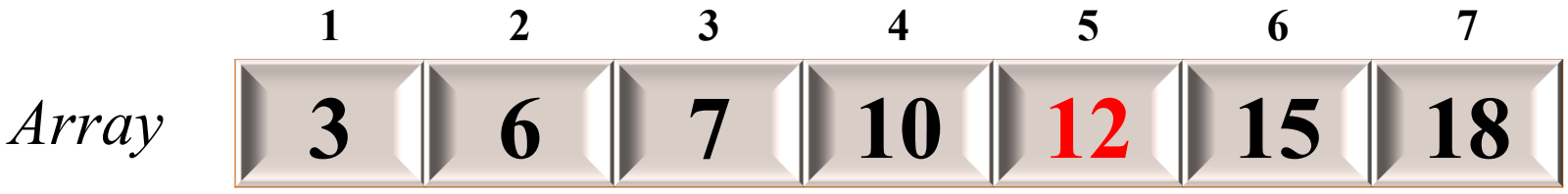
L'algoritmo Quicksort

- Ora che l'array è stato partizionato, possiamo usare Quicksort ricorsivamente
  - sui primi 4 elementi (10, 3, 6, e7)
  - e poi sugli ultimi 2 (15 e 18)

- La ricorsione termina quando si deve ordinare un array con un solo elemento: a quel punto non c'è niente da ordinare!

L'algoritmo Quicksort: Esempio





Programma: Quicksort

- Sviluppiamo una funzione ricorsiva `quicksort` che ordina un array di interi usando l'algoritmo Quicksort

- Il programma `qsort.c` legge 10 numeri memorizzandoli in un array e chiama la funzione `quicksort` per ordinarli ed infine stampa l'array ordinato:

```
Enter 10 numbers to be sorted: 9 16 47 82 4 66 12 3 25 51
In sorted order: 3 4 9 12 16 25 47 51 66 82
```

- Il codice per partizionare l'array è una funzione a parte che chiameremo `split`

qsort.c

```
/* Sorts an array of integers using Quicksort algorithm */

#include <stdio.h>

#define N 10

void quicksort(int a[], int low, int high);
int split(int a[], int low, int high);

int main(void)
{
  int a[N], i;

  printf("Enter %d numbers to be sorted: ", N);
  for (i = 0; i < N; i++)
    scanf("%d", &a[i]);

  quicksort(a, 0, N - 1);

  printf("In sorted order: ");
  for (i = 0; i < N; i++)
    printf("%d ", a[i]);
  printf("\n");

  return 0;
}
```

```
void quicksort(int a[], int low, int high)
{
  int middle;

  if (low >= high) return;
  middle = split(a, low, high);
  quicksort(a, low, middle - 1);
  quicksort(a, middle + 1, high);
}
```

```
int split(int a[], int low, int high)
{
  int part_element = a[low];

  for (;;) {
    while (low < high && part_element <= a[high])
      high--;
    if (low >= high) break;
    a[low++] = a[high];

    while (low < high && a[low] <= part_element)
      low++;
    if (low >= high) break;
    a[high--] = a[low];
  }

  a[high] = part_element;
  return high;
}
```

Programma: Quicksort

- Si può migliorare l'efficienza dell'implementazione
  - Migliorando il metodo di partizione
  - Usando un metodo diverso per ordinare array piccoli
  - Usando una versione non ricorsiva dell'algoritmo

… arrivederci alla prossima lezione