

The background of the slide is a spiral-bound notebook with a light brown, textured cover and a silver metal spiral binding on the left side. The paper is a light beige color with faint horizontal lines.

Ragionare sulle astrazioni procedurali: una dimostrazione

Stile delle dimostrazioni di correttezza (induzione sulle chiamate di procedura)

Procedura P

```
// REQUIRES:  $pre_P$   
// EFFECTS:  $post_P$ 
```

```
{...;  
Q;  
...}
```

Procedura Q

```
// REQUIRES:  $pre_Q$   
// EFFECTS:  $post_Q$ 
```

1. **assumo** pre_P
2. ...
3. **dimostro** pre_Q
4. **assumo** $post_Q$
5. ...
6. **dimostro** $post_P$

Dimostrazione di correttezza 1

```
public static void sort (int[] a)
    // EFFECTS: riordina gli elementi di a in modo
    // crescente, se a=[3,1,6,1] a_post=[1,1,3,6]
    {if (a == null) return;
    quickSort(a, 0, a.length - 1);}
private static void quickSort (int[] a, int mi, int ma)
    // REQUIRES: a non è null, 0<=mi, ma<a.length
    // EFFECTS: riordina gli elementi tra a[mi] e
    // a[ma] in modo crescente
```

- ✓ chiamata di quicksort: a non è null, mi=0, ma=a.length-1
- ✓ $pre_{quicksort}$ verificata
- ✓ assumo $post_{quicksort}$: gli elementi di a tra mi=0 e ma=a.length-1 sono ordinati in modo crescente
- ✓ $post_{sort}$ verificata

Dimostrazione di correttezza 2

```
private static void quickSort (int[] a, int mi, int ma)
    // REQUIRES: a non è null, 0<=mi, ma<a.length
    // EFFECTS: riordina gli elementi tra a[mi] e
    // a[ma] in modo crescente
    {if (mi >= ma) return;
     int mid = partition(a, mi, ma);
     quickSort(a, mi, mid);
     quickSort(a, mid + 1, ma);}
```

```
private static int partition (int[] a, int mi, int ma)
    // REQUIRES: a non è null, 0<=mi<ma<a.length
    // EFFECTS: riordina gli elementi tra a[mi] e
    // a[ma] in due gruppi mi..ris e ris+1..ma, tali
    // che tutti gli elementi del secondo gruppo sono
    // >= di quelli del primo; ritorna ris
```

- ✓ *assumo* $pre_{quickSort}$: a non è null, $0 \leq mi$, $ma < a.length$
- ✓ se $(mi \geq ma)$ a deve essere vuoto ed è quindi ordinato
- ✓ *altrimenti vale* $mi < ma$
- ✓ chiamata di *partition*. $pre_{partition}$ verificata: a non è null, $0 \leq mi < ma < a.length$
- ✓ *assumo* $post_{partition}$: gli elementi tra $a[mi]$ e $a[mid]$ sono tutti minori di quelli tra $a[mid+1]$ e $a[ma]$
- ✓ $pre_{quickSort}$ verificata per la prima chiamata
- ✓ *assumo* $post_{quickSort}$ per la prima chiamata: gli elementi di a tra mi e mid sono ordinati in modo crescente
- ✓ $pre_{quickSort}$ verificata per la seconda chiamata
- ✓ *assumo* $post_{quickSort}$ per la seconda chiamata: gli elementi di a tra $mid+1$ e ma sono ordinati in modo crescente
- ✓ $post_{quickSort}$ verificata: gli elementi tra mi ed ma sono ordinati

Dimostrazione di correttezza 3

```
private static int partition (int[] a, int mi, int ma)
    // REQUIRES: a non è null, 0<=mi<ma<a.length
    // EFFECTS: riordina gli elementi tra a[mi] e
    // a[ma] in due gruppi mi..ris e ris+1..ma, tali
    // che tutti gli elementi del secondo gruppo sono
    // >= di quelli del primo; ritorna ris
    {int x = a[mi];
    while (true) {
        while (a[ma] > x) ma--;
        while (a[mi] < x) mi++;
        if (mi < ma) {
            int temp = a [mi]; a[mi] = a[ma]; a[ma] = temp;
            ma--; mi++;}
        else return ma; }
    }}
```

✓ più difficile: bisogna ragionare sull'algoritmo!