Reasoning with invariants

Jordi Cortadella
Jordi Petit (Python version)
Department of Computer Science

Invariants

- Invariants help to …
  - Define how variables must be initialized before a loop
  - Define the necessary condition to reach the post-condition
  - Define the body of the loop
  - Detect whether a loop terminates

- It is crucial, but not always easy, to choose a good invariant.

- Recommendation:
  - Use invariant-based reasoning for all loops
    (possibly in an informal way)
  - Use formal invariant-based reasoning for non-trivial loops

General reasoning for loops

Invariant: a proposition that holds
- at the beginning of the loop
- at the beginning of each iteration
- at the end of each iteration

```
Initialization
# Invariant
while condition:
    # Invariant ∧ condition
    Body of the loop
    # Invariant
Invariant ∧ ¬ condition
```

Strategy:
- Stop the loop
- Look at the end of the body
- Take a picture
- Describe what you see

Variables and properties about their contents

Example with invariants

- Given $n \geq 0$, calculate $n!$

- Definition of factorial:

$$n! = 1 * 2 * 3 * \ldots * (n-1) * n$$

(particular case: $0! = 1$)

- Let's pick an invariant:
  - At each iteration we will calculate $f = i!$
  - We also know that $i \leq n$ at all iterations

Computing n!

```
def factorial(n: int) -> int:
    """Returns n!. Pre: n ≥ 0"""
    i = 0
    f = 1
    # Invariant: f = i! and i ≤ n
    while  i != n :
        # f = i! and i < n
        i = i + 1
        f = f * i
        # f = i! and i ≤ n

    # f = i! and i ≤ n and i == n
    # f = n!
    return f
```

Reversing digits

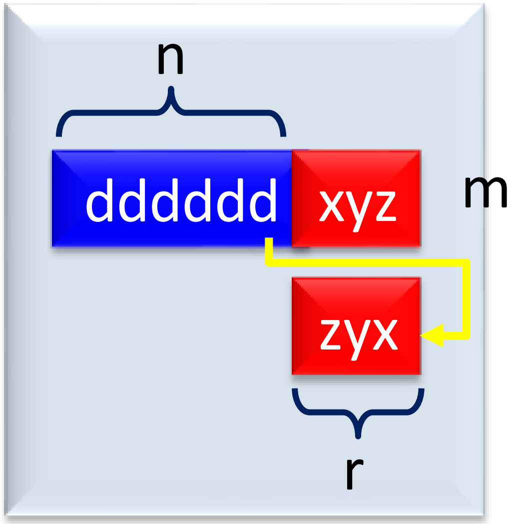
- Write a function that reverses the digits of a number (representation in base 10)

- Examples:

```
35276  →  67253
   19  →  91
    3  →  3
    0  →  0
```

Reversing digits

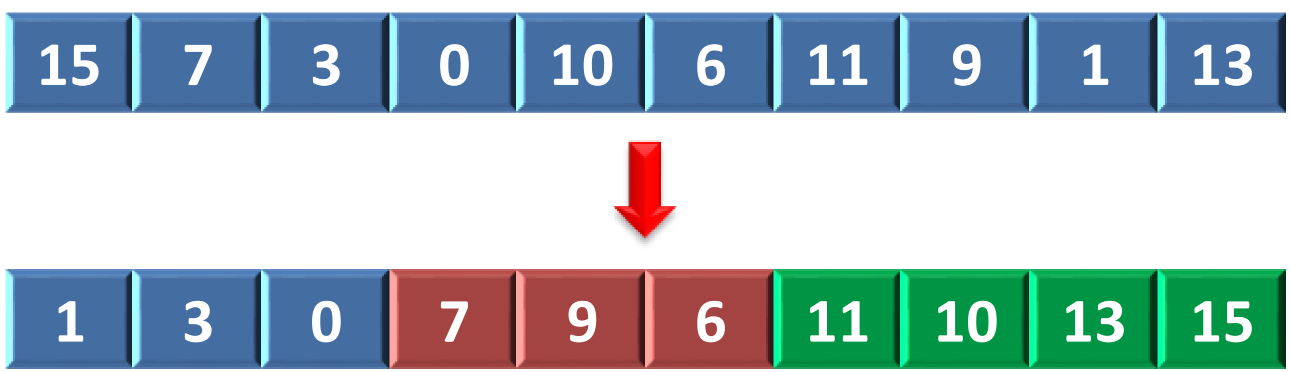
```
def reverse_digits(n: int) -> int:
    """Returns m with reversed digits (base 10)
       Pre: m ≥ 0"""

    n, r = m, 0
    # Invariant (graphical): →
    while n != 0  :
        r = 10 * r + n % 10
        n = n // 10

    return r
```



Classify elements

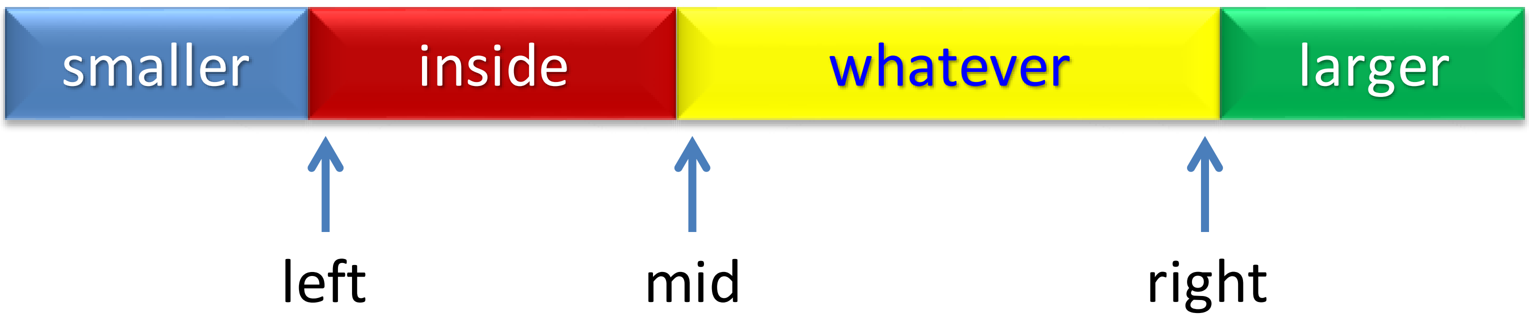
- We have a list of elements V and an interval [x,y] (x ≤ y). Classify the elements of the list by putting those smaller than x in the left part of the list, those larger than y in the right part and those inside the interval in the middle. The elements do not need to be ordered.

- Example: interval [6,9]



Classify elements

- Invariant:



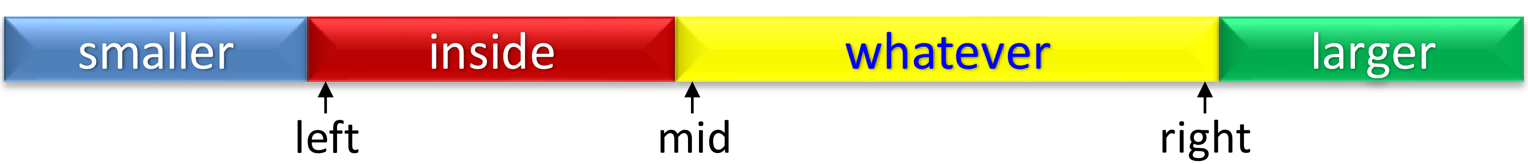
- At each iteration, we treat the element in the middle
  - If it is smaller, swap the elements in left and the middle (left→, mid→)
  - If larger, swap the elements in the middle and the right (←right)
  - If inside, do not move the element (mid→)

- End of classification: when mid > right.
  Termination is guaranteed since mid and right get closer at each iteration.

- Initially: left = mid = 0, right = len-1

Classify elements

```python
def classify(L: list[int], x: int, y: int) -> None:
    """Pre:  x <= y
       Post: the elements of V have been classified moving those
       smaller than x to the left, those larger than y to the
       right and the rest in the middle."""
    left, mid, right = 0, 0, len(L) - 1
    # Invariant: see the previous slide
    while mid <= right:
        if L[mid] < x:                      # Move to the left  part
            L[mid], L[left] = L[left], L[mid]
            left, mid = left + 1, mid + 1
        elif L[mid] > y:                    # Move to the right part
            L[mid], L[right] = L[right], L[mid]
            right = right - 1
        else:                               # Keep in the middle

            mid = mid + 1
```



List fusion

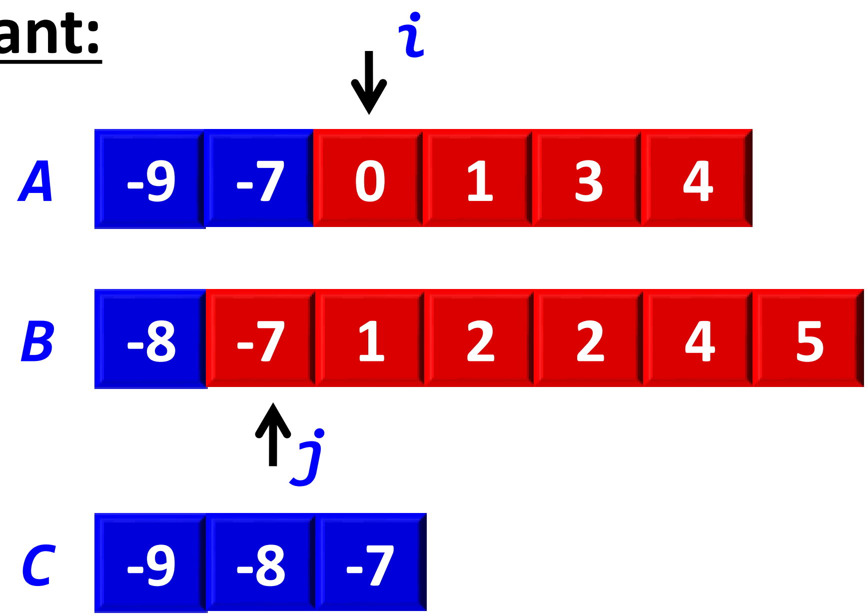
Design a function that returns the fusion of two ordered lists. The returned list must also be ordered. For example, C is the fusion of A and B:

A	-9	-7	0	1	3	4							
B	-8	-7	1	2	2	4	5						
C	-9	-8	-7	-7	0	1	1	2	2	3	4	4	5

Vector fusion

```python
def fusion(A: list[int], B: list[int]) -> list[int]:
    ''' Returns the sorted fusion of A and B.
        Pre: A and B are sorted in ascending order.'''
```

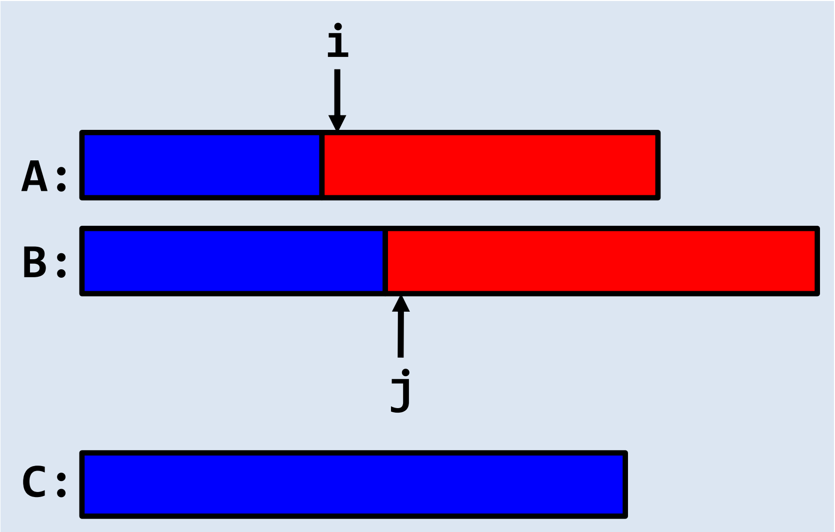
Invariant:



- C contains the fusion of A[0:i] and B[0:j]
- All the blue elements are smaller than or equal to the red ones.

Vector fusion

```
def fusion(A: list[int], B: list[int]) -> list[int]:
    """Returns the sorted fusion of A and B.
       Pre: A and B are sorted in ascending order."""

    C: list[int] = []
    i, j = 0, 0
    while i < len(A) and j < len(B):
        if A[i] <= B[j]:
            C.append(A[i])
            i = i + 1
        else:
            C.append(B[j])
            j = j + 1

    C.extend(A[i:])
    C.extend(B[j:])
    return C
```



Summary

- Using invariants is a powerful methodology to derive correct and efficient iterative algorithms.

- Recommendation to find a good invariant for a loop:
  - Consider the iterative progress of the algorithm.
  - Try to describe the state of the program at the beginning of an iteration (this is the invariant!).
  - Declare the variables required to describe the invariant.
  - Derive the condition, loop body and initialization of the variables of the loop (the order is not important)