

COMP250: Loop invariants

Jérôme Waldispühl

School of Computer Science

McGill University

Based on (CRLS, 2009) & slides from (Sora,2015)

Algorithm specification

- An algorithm is described by:
 - Input data
 - Output data
 - ***Preconditions***: specifies restrictions on input data
 - ***Postconditions***: specifies what is the result
- Example: Binary Search
 - Input data: `a:array of integer; x:integer;`
 - Output data: `index:integer;`
 - Precondition: `a` is sorted in ascending order
 - Postcondition: `index` of `x` if `x` is in `a`, and `-1` otherwise.

Correctness of an algorithm

An algorithm is correct if:

- for any correct input data:
 - it stops and
 - it produces correct output.
- Correct input data: satisfies precondition
- Correct output data: satisfies postcondition

Problem: Proving the correctness of an algorithm may be complicated when the latter is repetitive or contains loop instructions.

How to prove the correctness of an algorithm?

- Recursive algorithm \Rightarrow Induction proofs
- Iterative algorithm (For loops) \Rightarrow ???

Loop invariant

A **loop invariant** is a loop property that hold before and after each iteration of a loop.

Proof using loop invariants

We must show:

- 1. Initialization:** It is true prior to the first iteration of the loop.
- 2. Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.
- 3. Termination:** When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Analogy to induction proofs

Using loop invariants is like mathematical induction.

- You prove a **base case** and an **inductive step**.
- Showing that the invariant holds before the first iteration is like the base case.
- Showing that the invariant holds from iteration to iteration is like the inductive step.
- The **termination** part differs from classical mathematical induction. Here, we stop the “induction” when the loop terminates instead of using it infinitely.

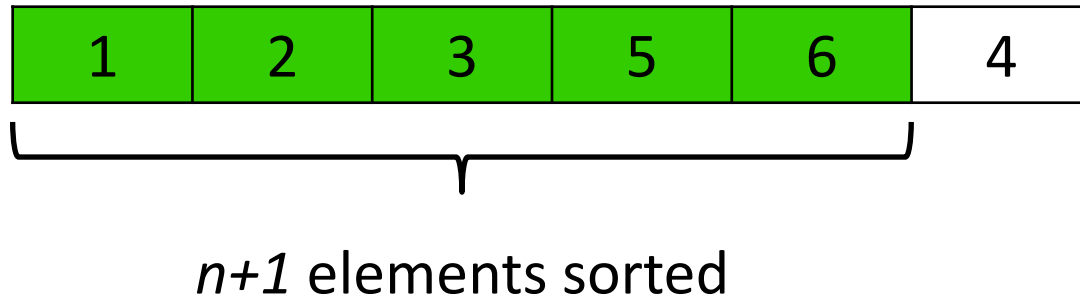
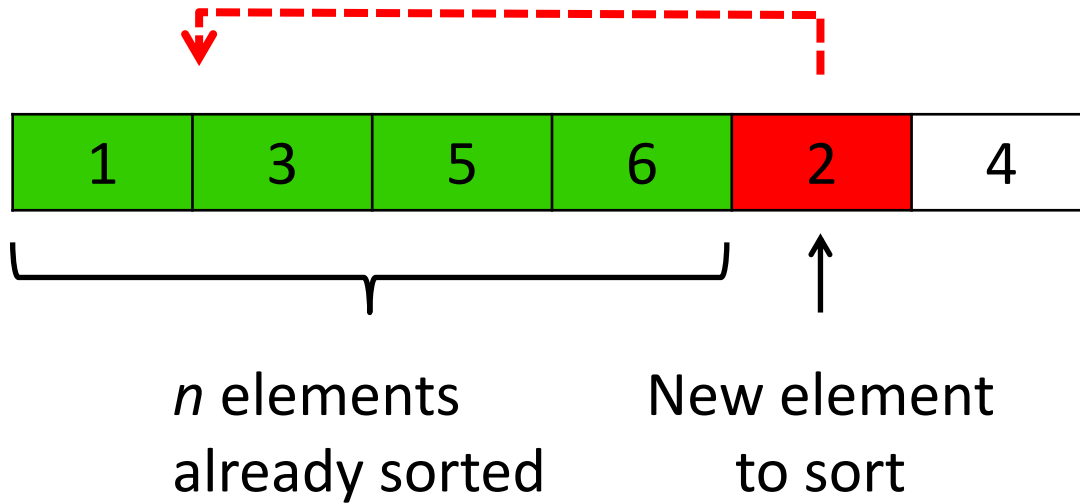
We can show the three parts in any order.

Insertion sort

```
for i ← 1 to length(A) - 1
  j ← i
  while j > 0 and A[j-1] > A[j]
    swap A[j] and A[j-1]
    j ← j - 1
  end while
end for
```

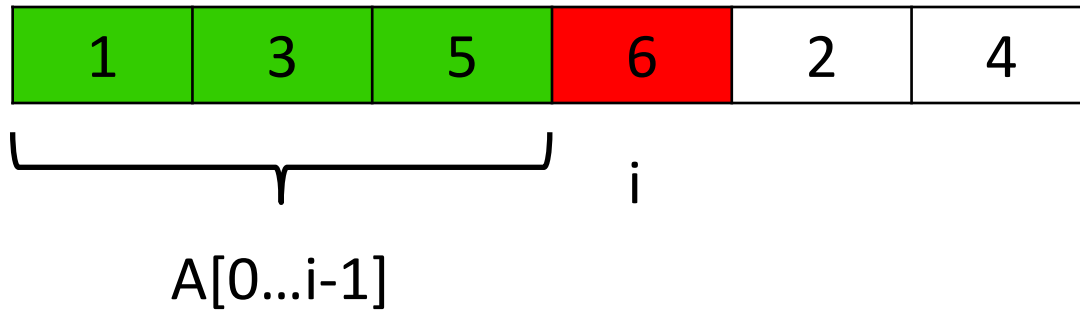
(Seen in Lecture 9)

Insertion sort



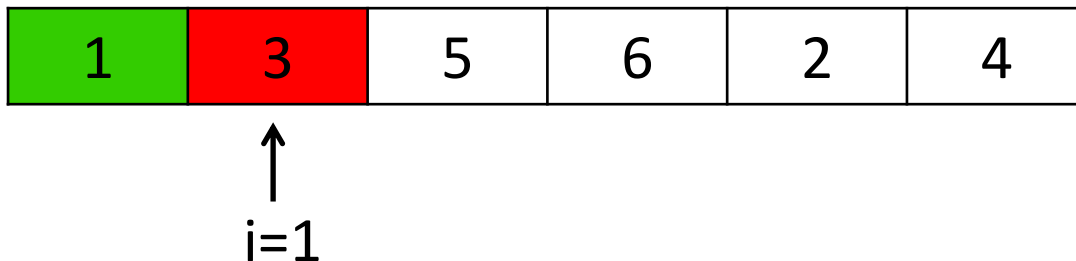
Loop invariant

The array $A[0\dots i-1]$ is fully sorted.

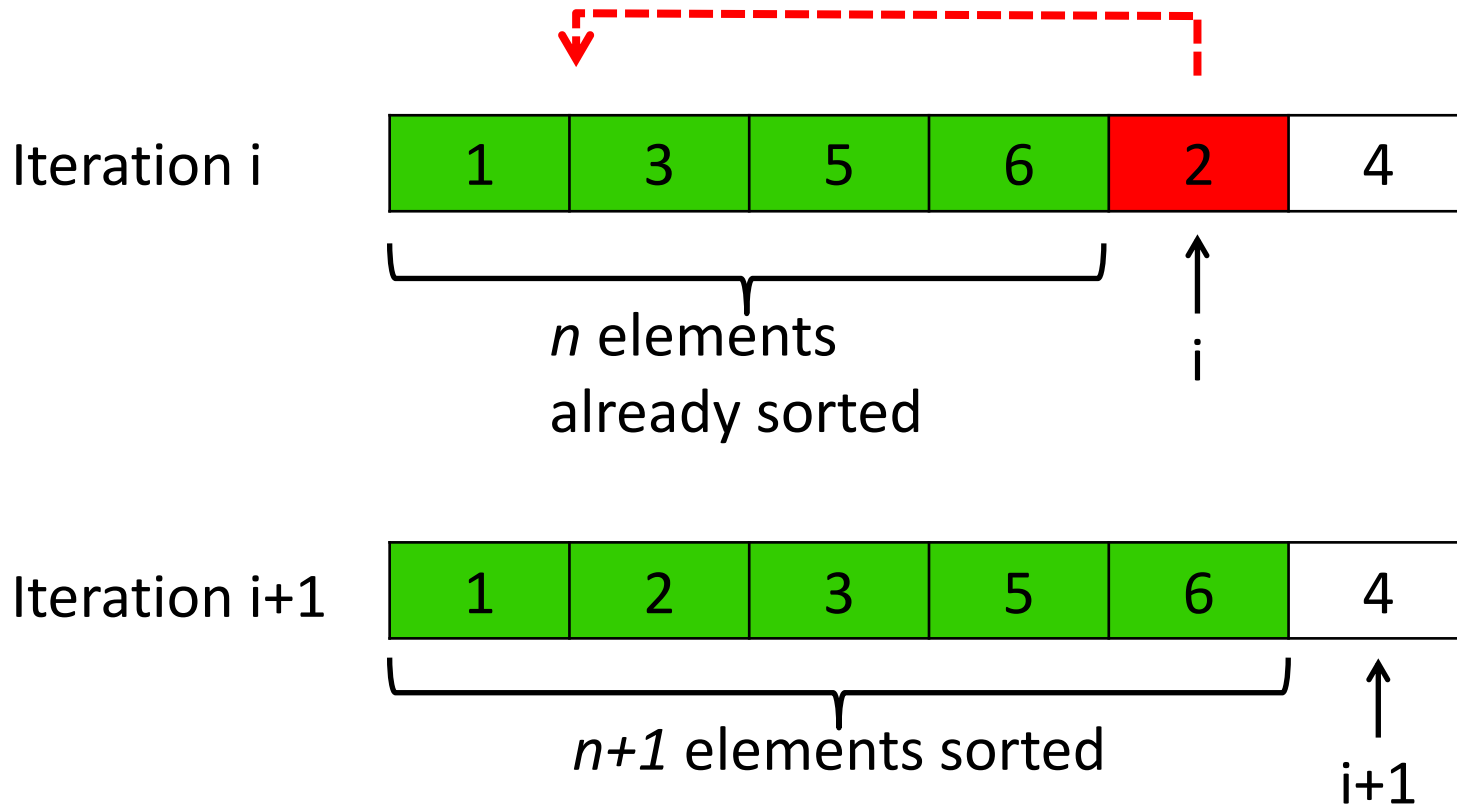


Initialization

Just before the first iteration ($i = 1$), the sub-array $A[0 \dots i-1]$ is the single element $A[0]$, which is the element originally in $A[0]$, and it is trivially sorted.



Maintenance



Note: To be precise, we would need to state and prove a loop invariant for the “inner” **while** loop.

Termination

The outer **for** loop ends when $i \geq \text{length}(A)$ and increment by 1 at each iteration starting from 1.

Therefore, $i = \text{length}(A)$.

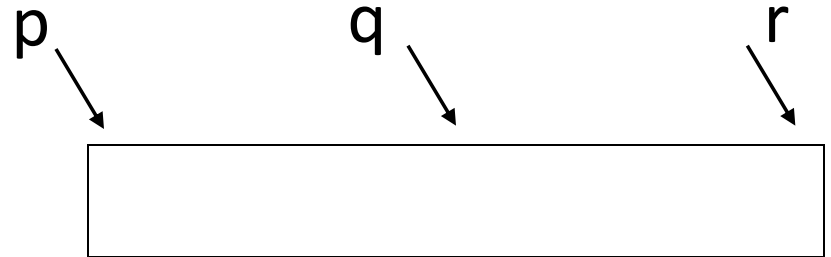
Plugging $\text{length}(A)$ in for $i-1$ in the loop invariant, the subarray $A[0 \dots \text{length}(A)-1]$ consists of the elements originally in $A[0 \dots \text{length}(A)-1]$ but in sorted order.

$A[0 \dots \text{length}(A)-1]$ contains $\text{length}(A)$ elements (*i.e. all initial elements!*) and no element is duplicated/deleted.

In other words, **the entire array is sorted.**

Merge Sort

```
MERGE-SORT (A, p, r)
  if p < r then
    q = (p+r) / 2
    MERGE-SORT (A, p, q)
    MERGE-SORT (A, q+1, r)
    MERGE (A, p, q, r)
```



Precondition:

Array A has at least 1 element between indexes p and r ($p \leq r$)

Postcondition:

The elements between indexes p and r are sorted

Merge Sort (Merge method)

- MERGE-SORT calls a function MERGE(A,p,q,r) to merge the sorted subarrays of A into a single sorted one
- The proof of MERGE can be done separately, using loop invariants

MERGE (A,p,q,r)

Precondition: A is an array and p, q, and r are indices into the array such that $p \leq q < r$. The subarrays A[p..q] and A[q+1..r] are sorted

Postcondition: The subarray A[p..r] is sorted

Procedure Merge

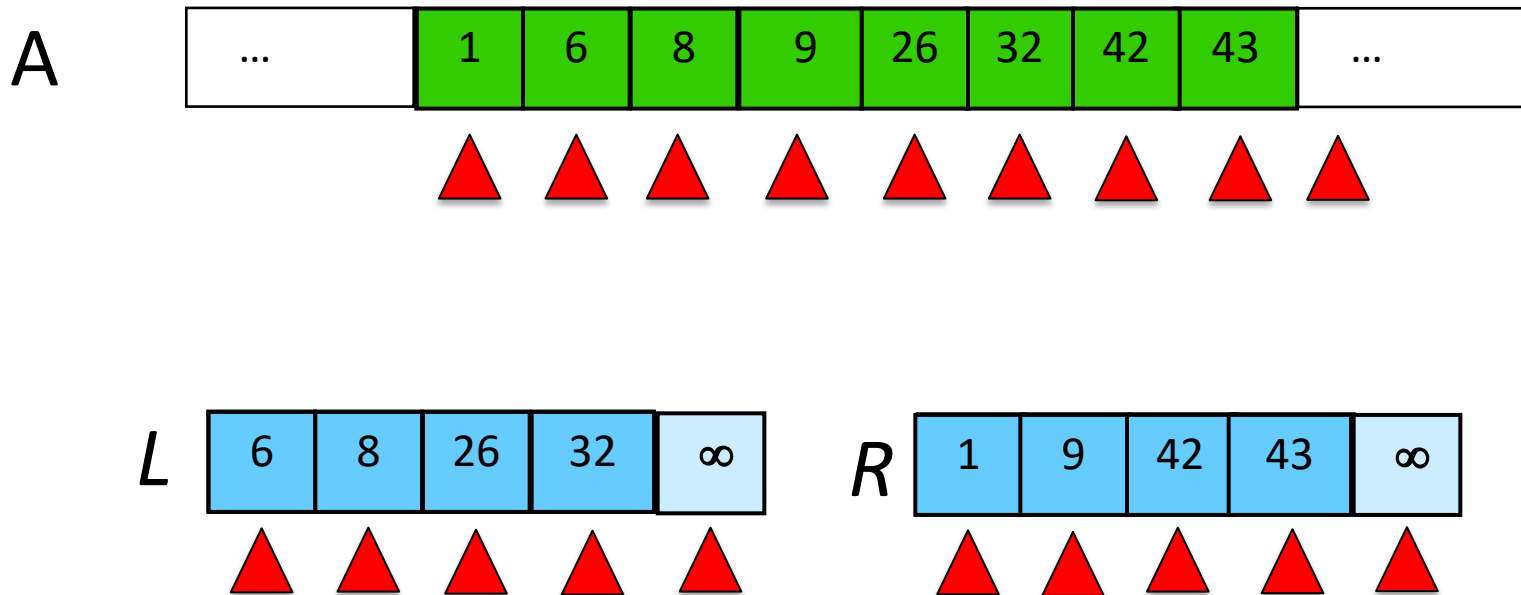
Merge(A, p, q, r)

```
1  $n_1 \leftarrow q - p + 1$ 
2  $n_2 \leftarrow r - q$ 
3   for  $i \leftarrow 1$  to  $n_1$ 
4     do  $L[i] \leftarrow A[p + i - 1]$ 
5   for  $j \leftarrow 1$  to  $n_2$ 
6     do  $R[j] \leftarrow A[q + j]$ 
7    $L[n_1 + 1] \leftarrow \infty$ 
8    $R[n_2 + 1] \leftarrow \infty$ 
9    $i \leftarrow 1$ 
10   $j \leftarrow 1$ 
11  for  $k \leftarrow p$  to  $r$ 
12    do if  $L[i] \leq R[j]$ 
13      then  $A[k] \leftarrow L[i]$ 
14             $i \leftarrow i + 1$ 
15      else  $A[k] \leftarrow R[j]$ 
16             $j \leftarrow j + 1$ 
```

Input: Array containing sorted subarrays $A[p..q]$ and $A[q+1..r]$.

Output: Merged sorted subarray in $A[p..r]$.

Merge/combine – Example



Idea: The lists *L* and *R* are **already sorted**.

Correctness proof for Merge

- **Loop Invariant:** The array $A[p,k]$ stored the $(k-p+1)$ smallest elements of L and R sorted in increasing order.
- **Initialization:** $k = p$
 - A contains a single element (which is trivially “sorted”)
 - $A[p]$ is the smallest element of L and R
- **Maintainance:**
 - Assume that Merge satisfy the loop invariant property until k .
 - $(k-p+1)$ smallest elements of L and R are already sorted in A.
 - Next value to be inserted is the smallest one remaining in L and R.
 - This value is large than those previously inserted in A.
 - Loop invariant property satisfied for $k+1$.
- **Termination Step:**
 - Merge terminates when $k=r$, thus when $r-p+1$ elements have been inserted in A \Rightarrow All elements are sorted.

Correctness proof for Merge Sort

- Recursive property: Elements in $A[p,r]$ are be sorted.
- **Base Case:** $n = 1$
 - A contains a single element (which is trivially “sorted”)
- **Inductive Hypothesis:**
 - Assume that MergeSort correctly sorts $n=1, 2, \dots, k$ elements
- **Inductive Step:**
 - Show that MergeSort correctly sorts $n = k + 1$ elements.
- **Termination Step:**
 - MergeSort terminate and all elements are sorted.

Note: There is no loop here, but we include this to complete the proof of Merge Sort.

Maintenance

- **Inductive Hypothesis:**
 - Assume MergeSort correctly sorts $n=1, \dots, k$ elements
- **Inductive Step:**
 - Show that MergeSort correctly sorts $n = k + 1$ elements.
- **Proof:**
 - First recursive call $n_1 = q - p + 1 = (k+1)/2 \leq k$
 - => subarray $A[p .. q]$ is sorted
 - Second recursive call $n_2 = r - q = (k+1)/2 \leq k$
 - => subarray $A[q+1 .. r]$ is sorted
 - A, p, q, r fulfill now the precondition of Merge
 - The post-condition of Merge guarantees that the array $A[p .. r]$ is sorted => post-condition of MergeSort satisfied.

Termination Step

We have to find a quantity that decreases with every recursive call: the length of the subarray of A to be sorted MergeSort.

At each recursive call of MergeSort, the length of the subarray is strictly decreasing.

When MergeSort is called on a array of size ≤ 1 (i.e. the base case), the algorithm terminates without making additional recursive calls.

Calling MergeSort(A,0,n) returns a fully sorted array.