Incremental Approach & Loop Invariant

Department of Computer Science, Tsinghua University

Pseudocode Conventions

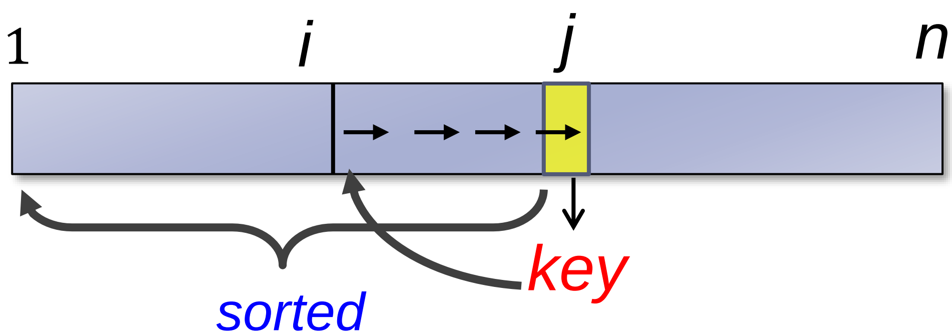
```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j]
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

- Organize compound data into objects, which are composed of attributes. E.g., $A.length$
- $A[i]$ indicates the ith element of the array A, $1 \leq i \leq A.length$
- $A[1..j]$ indicates the subarray of A consisting of the j elements $A[1], A[2], \cdots A[j]$.
- The loop counter (loop variable) retains its value after exiting the loop. Its value is the value that first exceeded the loop bound.

Incremental Approach

- Let's start with our first example *Insertion Sort*:
- Input:
  - A sequence of n numbers $< a_1, a_2, \cdots, a_n >$
- Output:
  - A permutation of the input sequence $< a'_1, a'_2, \cdots, a'_n >$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

$A[1..n]$



Design Points

- Incremental Approach
  - Main loop
  - Solving the problem by incrementally growing the solution
  - Pre-condition, Post-condition

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j]
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

Insertion Sort

INSERTION-SORT(A)

```
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

1.	5	2	4	6	1	3
2.	5	2	4	6	1	3
3.	2	5	4	6	1	3
4.	2	5	4	6	1	3
5.	2	4	5	6	1	3

Verification

- Mathematical Induction
  - First, we begin by establishing the result for an initial case.
  - Next, we prove the result (or a "statement") for a later case, say case n, by using the result for earlier cases.

- Algorithm verification
  - Use mathematical induction for verifying the correctness of a loop.
  - The "statement" established in this proof is called a *loop invariant*, a statement that is true at the beginning of every iteration of the loop.
  - Termination condition.

Loop Invariants

- Applying mathematical induction to iterative algorithms.
  - *Initialization:* verify the loop invariant is TRUE before the first iteration
  - *Maintenance:* the loop invariant remains TRUE for each iteration
  - *Termination:* the loop invariant leads to the correctness.

- *Note that:*
  - *A loop invariant:* contains the *loop variable* and other *useful variables.*

Insertion Sort

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j]
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

- *Loop Invariant: At the start of each iteration, $A[1..j-1]$ consists of the elements originally in $A[1..j-1]$, but in sorted order.*

Insertion Sort

- *Initialization*
  - $j = 2$, $A[1..1]$ *contains the same first element and is sorted by itself.*

INSERTION-SORT(A)

```
for j = 2 to A.length
    key = A[j]
    // Insert A[j]
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

Loop Invariant: At the start of each iteration, $A[1..j-1]$ *consists of the elements originally in* $A[1..j-1]$, *but in sorted order.*

Insertion Sort

- *Maintenance: At the start of each iteration*
  - $A[1..j-1]$ *consists of the elements originally in* $A[1..j-1]$, *but in sorted order. Why after the iteration,* $A[1..j]$ *is sorted?*

INSERTION-SORT(A)

```
for j = 2 to A.length
    key = A[j]
    // Insert A[j]
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

Loop Invariant: At the start of each iteration, $A[1..j-1]$ *consists of the elements originally in* $A[1..j-1]$, *but in sorted order.*

Insertion Sort

- *Termination: When $j = n + 1$*
  - *the loop invariant becomes $A[1..n]$ consists of the elements originally in $A[1..n]$, but in sorted order.*

INSERTION-SORT(A)

for $j = 2$ **to** $A.length$
    $key = A[j]$
    // Insert $A[j]$
    $i = j - 1$
    while $i > 0$ and $A[i] > key$
        $A[i + 1] = A[i]$
        $i = i - 1$
    $A[i + 1] = key$

Loop Invariant: At the start of each iteration, $A[1..j-1]$ consists of the elements originally in $A[1..j-1]$, but in sorted order.

Majority Element

```
candidate = NIL
count = 0
for i = 1 to n
        if count == 0
                candidate = A[i]
        if candidate == A[i]
                count = count + 1
        else count = count - 1
```

- Loop invariant:
  - Remove *count* number of *candidate* from $A[1..i-1]$, the remaining array does not contain any majority element.
- In-class exercise: different cases of maintenance.

Assertion

Invariant {I}

```
Precondition;
While (condition)
{
        loop body;
}
Postcondition;
```

```
Precondition => {I}
While (condition)
{       {I}
        loop body;
}
{I ∧ !condition} =>
Postcondition
```

```
assert(Precondition);
assert(I);
While (condition)
{       assert(I);
        loop body;
}
assert(I);
assert(Postcondition)
```

Group Exercise: *DNF Problem*

- 2-Color Dutch National Flag Problem: $A[1..n]$ contains ***red*** elements and ***blue*** elements; rearrange the array so that ***red*** elements are ahead of ***blue*** ones. Give a correct algorithm and its loop invariant.

- Postcondition:
  - $A[1..k]$ is **red** & $A[k+1..n]$ is **blue**

$A[1]$	…	$A[k]$	$A[k+1]$	…	$A[n]$

Group Exercise: DNF Problem

$A[1]$	…	$A[k]$	$A[k+1]$	…	$A[n]$

Summary

- Incremental Approach
  - Solving the problem by incrementally growing the solution.
- Correctness
  - Setup loop invariant (mathematical induction).
- Efficiency
  - Determined by loops, easy to be optimized by compilers.
- Design
  - Pre-condition, Post-condition
  - Loop invariant inspired by post-condition can help your algorithm design as well.