David M. Keil, Framingham State University

CSCI 252 Computer Science I Using Java

2. Arrays and loop design

1. Defining and populating arrays
2. Array operations and boundary errors
3. Searching arrays
4. Nested loops and sorting algorithms

Inquiry

- Is there a way to work conveniently with multiple items of the same type in memory?
- How do we store a collection of *numbers*?
- A collection of *objects*?

Topic objective

Define and safely manipulate arrays, designing nested loops and applying search and sorting algorithms

1. Defining and populating arrays

- How can a program define a collection of integers that can be traversed with a loop?
- How is memory allocated for such data?
- How can a file be retrieved into memory?

Subtopic objective

2.1b Describe Java arrays**

Java strings are sequences

- *String* class declares objects that are sequences of *char* indexed by a *subscript*
- Methods of *String* class: *length(), substring(), charAt(), length(), equals(t), compareTo(t), startsWith(t), startsWith(t, i), endsWith(t), contains(c, s), indexOf(t), trim(), indexOf(c, i), replace(c1, c2)*
- Each *String* method operates on the sequence

Using arrays

- *Array:* An indexed sequence of storage locations for data items all of the same type
- Arrays are *compound* data items
- Elements are accessed by *subscript* (index), ranging from 0 to (# elements – 1):

```
score[0] = 75;
out.print(score[0]);
```

- Address of an element of an array is calculated using its offset from the first element

Declaring arrays

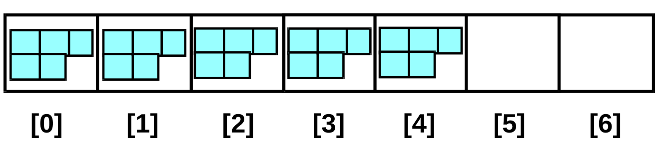
- Arrays are declared in Java using brackets, the *new* operator, and the number of elements:

```
int[] score = new int[7];
```

- Arrays are *objects* with the special constant public data member *length*:

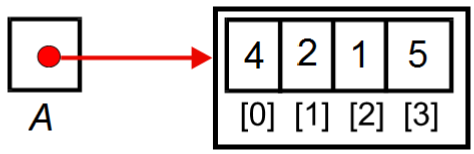
  `score.length` above is 7

- Array may have *occupancy* smaller than length



Arrays in memory

- Array delcaration creates a reference to an array
  `int A[4];`



- Length of this array is 4:
- Length of array (capacity) does *not* tell how many elements have *meaning*

Array initialization

```
int[] days_in_month =
  {
     31,28,31,30,31,30,
     31,31,30,31,30,31
  };
out.print("February has “ +
  days_in_month[1] + " days“);
```

Array of integers

- When array is initialized in this way, memory allocation by use of *new* is implicit and number of elements in array is number of initial values

Using an array

```
final int MAX=3;
int[] income = new int[MAX];
int sum=0;

income[0] = 258;
income[1] = 192;
income[2] = 467;
for (int i=0; i < MAX; ++i)
{
  out.print(income[i]);
  sum += income[i];
}
out.print("Total: “ + sum);
```


Reading a file into an array

```
final int MAX_SZ = 50;
String f_name = "Readfile.txt";
int[] A = new int[MAX_SZ];

System.out.println("Reading "+f_name);
FileReader reader=new FileReader(f_name);
Scanner fin = new Scanner(reader);

int i = 0;
while (fin.hasNextInt() && i < MAX_SZ)
  A[i++] = fin.nextInt(); // Read
fin.close();

for (int j=0; j < i; j++)
  System.out.print(A[j] + " "); //Display
```

Random-number generation

- *Uses:* simulation, testing
- An instance of *Random* is a stream, like a *Scanner*
- To assign random numbers in {1 .. 10} to an array:

```
Random r = new Random();
for (int i = 0; i < 10; i++)
  A[i] = r.nextInt(10)+1;
```


Coin flips

- Chance makes games more interesting
- *Example*, simulating roll of die:

```
Random gen = new Random();
boolean is_heads = (gen.nextInt(2) > 1)
```

- *nextInt(n)* returns a random integer in (0..*n*]
- *nextDouble(1)* returns a *double* in (0..1]
- *Note:* the *Random* class generates *pseudo-random* numbers, not literally random ones

Arrays of Booleans

- An array of truth values can contain yes/no information about elements of another sequence
- *Example:* to represent a *set* of numbers in the range of {1..8}, use an array of eight Booleans, each of which tells whether its subscript is in the set
- *Example:* The sequence (F, F, F, T, T, F, F, F) represents the set {4, 5}

Java array types

- Array type names are primitive-type or class names, followed by brackets:

```
int[] A;
String[] roster;
```

- To allocate memory for an array, *new* must normally be used:

```
int[] A = new int[10];
String[] roster =
  new String[100];
```

2. Array operations and boundary errors

- What are pitfalls with arrays?
- How are they overcome?

Subtopic objectives

2.2a Traverse an array**

2.2b Argue for the correctness of a loop design

2.2c Write a simulation using a random number generator†

2.2d Define a 2D array†

Array applications

Looping operations on arrays:

- *Search* or search and replace
- *Sort* in ascending order
- Calculate statistics such as *sum*, *average*, *maximum*, *minimum*, *median*, *variance*, *mode*

Longest ascending consecutive sequence (A)

$max \leftarrow 0$
for $i \leftarrow 1$ to $|A| - 1$
  $j \leftarrow i$
  while $A[j] \leq A[j+1] \wedge j < i$
    $j \leftarrow j + 1$
  if $j > max$
    $max \leftarrow j$
return max

Example: Calculating variance

```
int[] score = {90,75,84,94,89,97,81},
   i,j,total,avg,
   num_scores = sizeof score / sizeof(int);
// Find average:
for (i=0,total=0; i < num_scores; ++i)
  total += score[i];
avg = total / num_scores;
// Display variances from average:
out.printf("Average = %d\n",avg);
out.printf("Score    Variance\n");
for (j=0; j < num_scores; ++j)
  out.printf("%4d%12d\n",
  score[j],score[j] - avg);
```

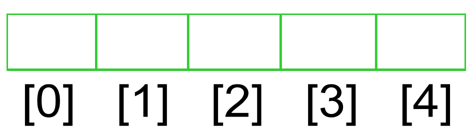
Output:

```
Average = 87
Score  Variance
  90         3
  75       -12
  84        -3
  94         7
  89         2
  97        10
  81        -6
```

Variance is the difference between one item's value and average value

Observing array boundaries

- **`int[] n = new int[5];`**
  allocates five *int*s



- Semantically valid subscripts here: 0 to 4
- A boundary violation throws an exception
- "**`n[5]`**" is valid syntax, but refers to memory that is outside the bounds of the declared array

Boundary errors

- It's recommended to write code that checks for valid subscripts, 0 .. ($|A|$ – 1)
- *Example:* for declaration
  `int[] A = new int[5];`
  accesses to `A[-1] or A[5]` are errors
- Both assignments are *bounds errors*; the Java run-time environment will throw exceptions
- Any access to an uninitialized array is an error: `int[]A; A[0] = 1; // error`

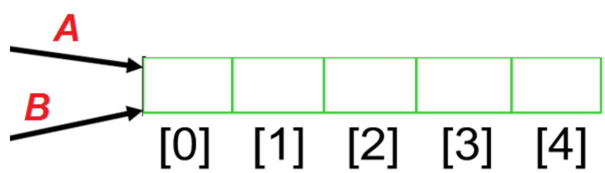
Arrays are objects

- An *array variable* is a reference to an *array object*
- Predefined methods for all arrays: *equals(), clone()*
- Constructor initializes all elements to zero or null value
- A method that has an array as a parameter may change values of elements

Assignment copies references

- Assigning an array reference as the value of an array object results in two references to the same object:

```
int[] A = new int[5];
int[] B = A;
```



Copying arrays correctly

- *Deep copying* makes a separate copy:

```
int[] B = A.clone();
int[] B = A.copyOf(A,n);
```

- `System.arraycopy(A,si,B,di,3)` copies three elements of *A*, starting at subscript *si*, to *B*, starting at *di*

Loops and associative operators

- Operations with associative operators may be evaluated for arrays using loops or induction (recursion)
- *Examples:* $+$, $\times$, $\wedge$, $\vee$, $\cap$, $\cup$
- Other operations on arrays may be computed using loops:
  - Inversion ($\neg$)
  - *Is-ascending*
  - *All-identical*
  - Count
  - Search
  - Search-replace

Loop invariant:

An assertion, about the state of an algorithmic process, that is true at the start of each iteration of a loop, and that helps to establish the validity of a postcondition

Rationale: If we can show that an assertion is true at the top of the loop and true throughout its execution, then we can show that the assertion is true after the loop terminates

Traversing an array

- A loop invariant helps see that the array-summing algorithm is correct:

  sum ← 0
  for i ← 1 to $|A|$

  $A[1] + A[2] + \ldots + A[i - 1]$

  LI: *sum* $= \sum_{k=1}^{i-1} A[k]$

  sum ← *sum* + $A[i]$

- $\sum_{k=1}^{i-1} A[k]$ holds at the start of this loop's body on *every* iteration, so it will also be valid after the loop terminates, when $i = |A|$.

Recurrences for arrays

- *Function definition for adding a sequence:*

  Sum(A) =

  $$\begin{cases} 0 & \text{if } |A| = 0 \\ A[1] & \text{if } |A| = 1 \\ A[1] + Sum(A[2 \,..\, |A|]) & \text{otherwise} \end{cases}$$

- *Algorithm suggested by this definition:*

  sum ← 0
  for i ← 1 to $|A|$
  sum ← *sum* + $A[i]$

- Similar recurrences exist for other functions

C-style strings are arrays of *char*

- The C string library and I/O functions treat the ASCII value 0 (‘\0’) as a sentinel that terminates a string

```
char name[6] = “Bill”;
```

B	i	l	l	'\0'	
[0]	[1]	[2]	[3]	[4]	[5]

- Assigning ‘\0’ to a character element of a string may shorten the string

```
name[3] = ’\0’;
out.printf(”%s”,name);
```


Enhanced *for* loop

- Traverses an array without a counter
- *Example:*

```
int[] A = {3, 8, 1, 5, 6}
int y = 0;
for (n: A)
   y += n;
```

- *n* takes the value of the next element of array *A* as the loop proceeds
- *Limitation:* Does not enable assigning values to array elements

Two-dimensional arrays

- Java can represent a *matrix* as an array of arrays; each element has two subscripts

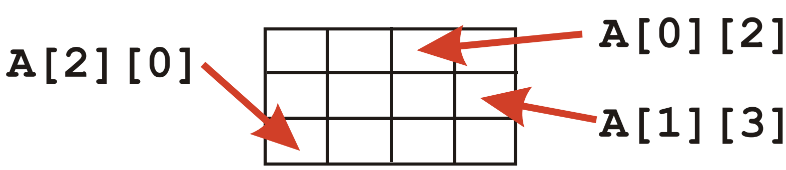
0	1	2	← Index, the day
3552.2	3560.0	3540.0	← High
3539.2	3544.8	3530.6	← Low
3550.5	3557.7	3530.6	← Close

```
double[][] daily_average =
  new double [CLOSE+1][MAX_DAYS];
```

Rows *Columns*

Rows and columns in 2D array

- First subscript represents *row*



- Second subscript represents *column*
- This is called *row-major ordering*

Initializing a 2D array

```
const int NUM_DAYS = 5;

public static void main()
{
  double price[CLOSE+1][NUM_DAYS] =
     {
          {76.2,81.3,78.5,79.2,80.7},
          {70.9,75.4,71.3,71.8,74.1},
          {74.0,73.6,78.2,76.6,79.5}
     };
…
```


Example: Tic-tac-toe board

```
char board[3][3] = {'-'};
board[0][0] = 'X';
board[1][1] = 'X';
board[2][2] = 'X';
for(int row = 0; row < 3; ++row)
{
  for(int col = 0; col < 3; ++col)
     out.print(board[row][col]);
}
```

```
X - -
- X -
- - X
```

Array traversal problems

Accept an array of integers as a parameter, and an integer x, and return

1. the array with *x* inserted at the beginning
2. index of the first occurrence of *x*; –1 if *x* not found
3. number of occurrences of *x*
4. *true* iff the sum of the first half of the array is greater than the sum of the second half.
5. original array, with an integer, *x*, inserted at the *beginning* of the array

Limits of testing

This program can hang. What is the bug and how many tests would be expected to find it?

```
void main()
{
    int x1 = in.nextInt(),
      x2 = in.nextInt();
    srand(&time);
    if (x1 == rand() && x2 == rand())
        while(1);
    out.println("x1 + x2 = " + (x1 + x2));
}
```

Standard random-number functions

Conditions for software correctness

- Program eventually halts under all conditions

- Thus, values that control termination of a loop must *converge*
- For *all* possible inputs or parameter values, a program or function must have a correct result
- By reasoning about the code, we can judge correctness and find bugs

Assertions and correctness

- A comment that is an assertion tells not what *occurs*, but something about *values* of variables and expressions
- Valid assertions can help us establish that our code does what we claim
- Chief tools: *preconditions*, *postconditions*, *loop invariants*

Pre- and post- conditions

- *Precondition:* An assertion about what the state of inputs is before an algorithm executes
- *Postcondition:* An assertion that is claimed to hold after execution if the precondition holds
- *Example:* Adding a series of numbers
  - Precondition: *total* is 0
  - Postcondition: *total* stores the sum of all input values

Invariants help verify correctness

```
sum ← 0
For i ← 1 to 5
    Input term_i
    sum ← sum + term_i
```

- Invariant: *sum* stores $\sum_{k=1}^{i} term_k$
- Postcondition of this pseudocode: *sum* stores $\sum_{k=1}^{5} term_k$

Loop-invariant example

```
void main()
{
   double input=0,largest = in.nextInt();
   out.println("Enter a number (< 0 to exit): ");
   // Precondition: <largest> stores input
   while (input >= 0)
   {
      // Loop invariant: <largest> stores
      // maximum input so far
      cout << "Enter a number (< 0 to exit): ");
      input = in.nextInt();
      if (input > largest)
           largest = input;
   };
   // Postcondition: 'largest' is the
   // maximum value entered
}
```


What are the invariants?

```
int input, total = 0;
input = in.nextInt();
while (input > 0)
{
  input = in.nextInt();
  total += input;
}
```

```
int i = 0, count = 0;
String s = in.nextLine();
while (i < s.length())
  if (s.charAt(i++) == ' ')
     count++;
```

Formal verification

- *Preconditions:* Comments that state assumptions of a method: e.g., a precondition of a method that computes *sqrt(x)* is $x \geq 0$
- *Postconditions:* Specification of code, or what the user can expect; e.g., a postcondition of *sqrt(x)* is that return value squared approximates x
- *Loop invariants:* Assertions that hold at the beginning of a loop body throughout execution of the loop

3. Searching arrays

How would we find

- the tallest person in this class?
- everyone whose SS#s start with ‘2’?
- the person closest to six feet tall?
- a phone number in a phone book?
- whether a deck of cards is complete, without duplicates?

Subtopic objectives

2.3a Explain a search algorithm

2.3b Search an array or merge arrays*†

Searching for the first ‘1’

```
final int ARR_SZ = 6;
int match = -1;
char A[ARR_SZ] =
{‘0’,’0’,’1’,’0’,’1’,’1’};
for (int i = 0; i < ARR_SZ; ++i)
  if(A{i] == ‘1’) {
    match = i;
    break;
  }
if match >= 0
  out.print( “Found at A[“ + match + “]”);
else out.print(“Not found”);
```

Output:

```
Found at A[2]
```

- How does time rise as *ARR_SZ* rises?

Search algorithms

- *Linear*
  - inspects each element of array
  - slow
  - works on any array
  - simple to code
- *Binary*
  - Similar to alphabetic-list lookup
  - fast
  - works only on ordered arrays
  - more complex to code

Linear-search (*A*, *key*)

i ← 1
while $i \leq$ size of A
    $i \leftarrow i + 1$
    if $A[\,i\,]$ matches *key*
        return *true*
return *false*

- Finds element with value *key* in array *A*
- Takes up to one pass through *A*

Verifying an array is sorted

- *Fact:* The best search algorithm requires array be in ascending order
- *Problem:* Determine whether the elements of array *A* are in ascending order

2	3	5	4	7	8	11	15

- How many steps for this array?
- How many for *any* 8-element array?
- How many for *any* *n*-element array?

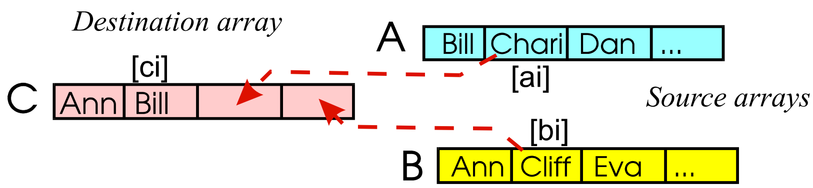
Inserting into a sorted array

```
int insert(float A[], double new_item)
// Preconditions:  A not full; A is ascending
// Postconditions:  <A> contains <new_item>,
//                 <A> is still ascending.
{
   int i = size;
   while (new_item < A[i-1] && i > 0)
   {
      A[i] = A[i-1];
      i = i - 1;
   }
   A[i] = new_item;
   return ++size;
}
```

*Move elements greater than **new_item** to the right*

*Drop **new_item** in place*

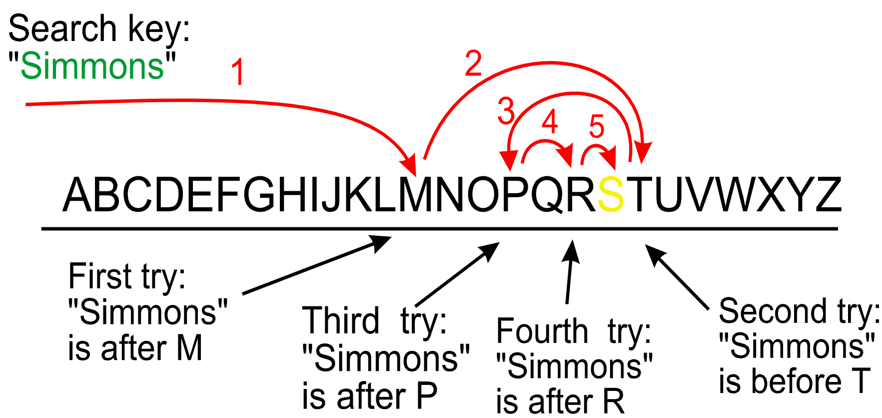
Merging two sorted arrays



Repeat until A and B are exhausted:
Append the lesser of $\{A_{ai}, B_{bi}\}$ to C, incrementing the indexes *ai, bi*, and *ci* as appropriate

- *C* should be as large as *A, B,* together

Binary phone-book search



- Each step eliminates half the unsearched data, cuts remaining work in half

Binary search algorithm

Compares middle array element to search key; repeats as necessary for left or right half

Binary-search(*A, key*)
first ← 1
last ← *Size*(*A*)
While *first* ≤ *last*
    middle ← $\lfloor$(*first* + *last*) ÷ 2$\rfloor$
    if *A*[*middle*] = *key*   return *true*
    otherwise
        if *A*[*middle*] > *key*   *last* ← *middle* − 1
        otherwise   *first* ← *middle* + 1
Return *false*

A driver to test a binary-search method

```
public static void main()
{
  int A[] = { 3,4,6,7,9,10,12,14,17,18,20};
  System.out.print("Enter search key: ");
  int key = in.nextInt();
  if (binary_srch(A,key))
    System.out.println("Found");
  else
    System.out.println("Not found");
}
```

Sample I/O:
`Enter search key: `<u>`2`</u>` Found`
`Enter search key: `<u>`8`</u>` Not found`

Code for iterative binary search

```
bool binary_search(int A[],int key,int size)
{
  int first = 0, last = size-1, middle;
  bool found = false;
  // Preconditions: <size> is size of A;A is sorted
  while (first <= last && ! found)
  {
    // Loop invariant:
    // key is in range A[first..last] or not in A
    middle = (first + last) / 2;
    if (A[middle] == key)     found = true;
    else
      if (key < A[middle])    last = middle - 1;
      else                    first = middle + 1;
  }
  return found;
  // Postcondition: <found> tells whether <key> in A
}
```

[*binsrch.cpp*]

Search algorithm running times

Array size	Linear search	Binary search
10	10	3
100	100	6
1000	1000	10
1M	1M	~20
1G	1G	~30

4. Nested loops and sorting algorithms

- Did you study *nested loops* in CS II?
- Have you sorted data in a text editor or spreadsheet?
- What is the postcondition of any sorting algorithm?
- What is true for every pair of consecutive elements $A[i]$, $A[i + 1]$, in a sorted array?

Subtopic objectives

2.4a Give the output of a nested loop**

2.4b Write a nested loop**

2.4c Explain a sorting algorithm

2.4d Sort an array*†

Problems that require nested loops

- Finding duplicates: Must compare each element with each other element
- Finding the pair of array elements that are closest to each other in value
- Finding *mode* (most frequent value): must create a new array that stores counts of each value; or must sort first
- Printing addition, multiplication tables

Finding duplicates

<u>Has-duplicates(*A*)</u>

```
y ← false
for i ← 1 to |A| – 1
      > Loop inv: y = true iff
      > A[1..|A|] has some duplicates
      for j ← 1 to |A| – 1
            if A[ i ] = A[ j ]
                  y ← true
return y
Post y = true iff A has some duplicates
```

Sorting

- *Sort:* an algorithm that arranges array elements in ascending order
- Sorting is a precondition for the binary search and the merge algorithm
- Simple sorts: bubble, selection, insertion
- *Postcondition for sort:* every element is at least as large as the one to its left

Bubble sort intuition

```
Repeat
  for each element
    if it is greater than its successor
      swap them
until none are found out of order
```

- The exit condition helps prove that bubble sort works
- The use of nested loops suggests it may take (n^2) comparisons to sort an n-element array

Bubble sort

> Precondition: *A* is an array
Repeat
  swapped ← *false*
  for *i* ← 1 to *Size*(*A*) – 1
    if *A*[*i*] > *A*[*i* +1]
      swap *A*[*i*] with *A*[*i* +1]
      swapped ← *true*
until *swapped* = *false*
> Postcondition: *A* is ascending

Bubble-sort(*A*)

//Precondition: *A* is an array
np ← 0 // number of passes
Repeat
  //Invariant: Rightmost (*np*) elements are sorted;
  //if *swapped* is false and *np* > 0, then *A* is sorted
  swapped ← *false*
  for *i* ← 1 to *size*(*A*) – 1
    //Invariant: $(\forall x \in A[0..i-1])\ A[i] \geq x$
    if *A*[*i*] > *A*[*i* + 1]
      swap(*A*[*i*], *A*[*i* + 1])
      swapped ← *true*
  np ← *np* + 1
until *swapped* is *false*
//Postcondition: *A* is sorted ascending

Swapping array elements

- Let method *swap(A, i, j)* return parameter array, *A*, modified so that *A*[*i*] and *A*[*j*] have exchanged positions
- ```
  int t = A[i];
  A[i] = A[j];
  A[j] = t;
  ```
  Why is the first line needed?

How selection sort works

Repeatedly

- Select the smallest value in the array
- Move it to the right end of a growing sub-sequence to the left of the array
  ```

Selection Sort(*A, start, size*)

If *size* > 1
  i ← *min-index*(A[*start..size*])
  Swap (*A*[*start*], A[*i*])
  Return *Selection-sort* (*A*, *start* + 1, *size*)
else return *A*

Min-index(*A*) =

$$\begin{cases} 1 & \text{if } |A| = 1 \\ \min(A[1], A[\textit{min-index}(A[2..\,|A|])]) & \text{otherwise} \end{cases}$$

Selection sort

```
void main()
{
  int score[] = {3,2,9,7,4,6,5,1,8,0};
  int n = sizeof(score)/sizeof(int);
  selection_sort(score,n);
  for (int i = 0; i < n; ++i)
     cout << score[i] << " ";
}
void selection_sort(int A[],int size)
{
  for (int i=0; i < size-1; ++i) {
     // Find min { A[i..size-1]}, swap w/ A[i]:
     int smallest = i; // Index of least value
     for (int j = i+1; j < size; ++j)
        if (A[j] < A[smallest])
           smallest = j;
     swap(A[i],A[smallest]);
  }
}
```

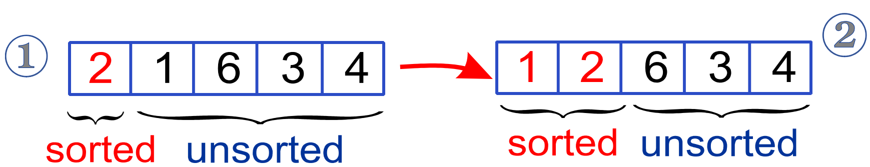
Steps:
21634
12634
12634
12364
12346
12346

How insertion sort works

- Repeatedly move the *first* element of the unsorted (right-hand) part of the array …
- into its sorted position in a growing subsequence at the left

Insertion-sort(*A*)

Precondition: *A* is an array
num_sorted ← 1
Repeat
    Invariant: *A*[1...*num-sorted*] is ascending
    i ← *num-sorted* + 1
    A ← *Array-insert* (*A*, *num_sorted*, *A*[*i*])
    num_sorted ← *num_sorted* + 1
until *num_sorted* = |*A*|
Postcondition: *A* is sorted ascending



What do these sorting algorithms have in common?

- *Hint:* What *loops* occur, and where, in these algorithms?
- *Terminology:*
  - a loop that iterates from the start to the end of an array is called a *traversal*
  - A loop inside a loop is *nested*

References

A. Downey. *Think Java*. [Available online.]

Cay Horstmann. *Big Java Early Objects*. Wiley, 2014.

D. Keil. Displaying an array. Classroom handout.

Reges and Stepp. *Building Java Programs*.