Computer Science 331
Classical Sorting Algorithms

Mike Jacobson

Department of Computer Science
University of Calgary

Lecture #22-23

Outline

The "Sorting Problem"

Precondition:

A: Array of length n, for some integer $n \geq 1$, storing objects of some ordered type

Postcondition:

A: Elements have been permuted (reordered) but not replaced, in such a way that

$$A[i] \leq A[i+1] \quad \text{for } 0 \leq i < n-1$$

Two Classical Algorithms

Discussed today: two "classical" sorting algorithms

- Reasonably simple
- Work well on small arrays
- Each can be used to sort an array of size n using $\Theta(n^2)$ operations (comparisons and exchanges of elements) in the worst case
- None is a very good choice to sort large arrays: asymptotically faster algorithms exist!

A third (bubble sort) will be considered in the tutorials.

Selection Sort

Idea:

- Repeatedly find "i^{th}-smallest" element and exchange it with the element in location $A[i]$
- Result: After i^{th} exchange,

$$A[0], A[1], \ldots, A[i-1]$$

  are the i smallest elements in the entire array, in sorted order — and array elements have been reordered but are otherwise unchanged

Pseudocode

```
void Selection Sort(int[] A)
  for i from 0 to n − 2 do
    min = i
    for j from i + 1 to n − 1 do
      if A[j] < A[min] then
        min = j
      end if
    end for
    {Swap A[i] and A[min]}
    tmp = A[i]
    A[i] = A[min]
    A[min] = tmp
  end for
```


Example

A:

2	6	3	1	4

Idea: find smallest element in $A[i], \ldots, A[4]$ for each i from 0 to $n-1$

$\underline{i = 0}$

- set $min = 3$ ($A[3] = 1$ is minimum of $A[0], \ldots, A[4]$)
- swap $A[0]$ and $A[3]$ ($A[0]$ sorted)

A:

1	6	3	2	4

$\underline{i = 1}$

- set $min = 3$ ($A[3] = 2$ is minimum of $A[1], \ldots, A[4]$)
- swap $A[1]$ and $A[3]$ ($A[0], A[1]$ sorted)

A:

1	2	3	6	4

Example (cont.)

$\underline{i = 2}$

- set $min = 2$ ($A[2] = 3$ is minimum of $A[2], \ldots, A[4]$)
- swap $A[2]$ and $A[2]$ ($A[0], A[1], A[2]$ sorted)

A:

1	6	3	2	4

$\underline{i = 3}$

- set $min = 4$ ($A[4] = 4$ is minimum of $A[3], A[4]$)
- swap $A[3]$ and $A[4]$ ($A[0], A[1], A[2], A[3]$ sorted)

A:

1	2	3	4	6

Finished! $A[0], \ldots, A[4]$ sorted

Inner Loop: Semantics

The inner loop is a **for** loop, which does the same thing as the following code (which includes a **while** loop):

$j = i + 1$
while $j < n$ **do**
  if $(A[j] < A[min])$ **then**
    $min = j$
  end if
  $j = j + 1$
end while

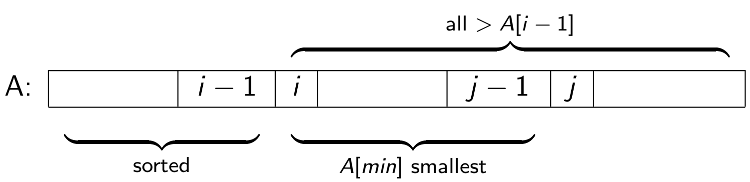
We will supply a "loop invariant" and "loop variant" for the above **while** loop in order to analyze the behaviour of the corresponding **for** loop

Inner Loop: Loop Invariant

Loop Invariant: At the beginning of each execution of the inner loop body

- $i, min \in \mathbb{N}$
- First subarray (with size i) is sorted with smallest elements:
  - $0 \leq i \leq n-2$
  - $A[h] \leq A[h+1]$ for $0 \leq h \leq i-2$
  - if $i > 0$ then $A[i-1] \leq A[h]$ for $i \leq h \leq n-1$
- Searching for the next-smallest element:
  - $i+1 \leq j < n$
  - $i \leq min < j$
  - $A[min] \leq A[h]$ for $i \leq h < j$
- Entries of A have been reordered; otherwise unchanged

Inner Loop: Interpretation of the Loop Invariant



Interpretation:

- $A[0] \leq A[1] \leq \cdots \leq A[i-1]$
- If $i > 0$ then $A[i-1] \leq A[\ell]$ for every integer ℓ such that $i \leq \ell \leq n$
- $i \leq min \leq j-1$ and $A[min] \leq A[h]$ for every integer h such that $i \leq h \leq j-1$
- entries of A have been reordered, otherwise unchanged

Application of the Loop Invariant

Loop invariant, final execution of the loop body, and *failure of the loop test* ensures that:

- $j = n$ immediately after the final execution of the inner loop body
- $i \leq min < n$ and $A[min] \leq A[\ell]$ **for all** ℓ such that $i \leq \ell < n$
- $A[min] \geq A[h]$ for all h such that $0 \leq h < i$

In other words, $A[min]$ is the value that should be moved into position $A[i]$

Inner Loop: Loop Variant and Application

Loop Variant: $f(n,i,j) = n - j$

- decreasing integer function
- when $f(n,i,j) = 0$ we have $j = n$ and the loop terminates

Application:

- initial value is $j = i + 1$
- worst-case number of iterations is $f(n,i,i+1) = n - (i+1) = n - 1 - i$

Outer Loop: Semantics

The outer loop is a **for** loop whose index variable i has values from 0 to $n-2$, inclusive

This does the same thing as a sequence of statements including

- an initialization statement, $i = 0$
- a **while** loop with test "$i \leq n-2$" whose body consists of the body of the **for** loop, together with a final statement $i = i + 1$

We will provide a loop invariant and a loop variant for this **while** loop in order to analyze the given **for** loop

Outer Loop: Loop Invariant and Loop Variant

Loop Invariant: At the beginning of each execution of the outer loop body

- i is an integer such that $0 \leq i < n-1$
- $A[h] \leq A[h+1]$ for $0 \leq h < i$
- if $i > 0$, $A[i-1] \leq A[\ell]$ for $i \leq \ell < n$
- Entries of A have been reordered; otherwise unchanged

Thus: $A[0], \ldots, A[i-1]$ are sorted and are the i smallest elements in A

Loop Variant: $f(n,i) = n - 1 - i$

- decreasing integer function
- when $f(n,i) = 0$ we have $i = n-1$ and the loop terminates
- worst-case number of iterations is $f(n,0) = n - 1$

Analysis of Selection Sort

Worst-case:

- inner loop iterates $n-1-i$ times (constant steps per iteration)
- outer loop iterates $n-1$ times
- total number of steps is at most

$$c_0 + \sum_{i=0}^{n-2} c_1(n-1-i) = c_0 + c_1\frac{n(n-1)}{2}$$

Conclusion: Worst-case running time is in $\Theta(n^2)$

Analysis of Selection Sort, Concluded

Best-Case:

- Both loops are **for** loops and a *positive* number of steps is used on each execution of the inner loop body
- Total number of steps is therefore *at least*

$$\widehat{c}_0 + \sum_{i=0}^{n-2} \widehat{c}_1 (n-1-i) \in \Omega(n^2)$$

Conclusion: *Every* application of this algorithm to sort an array of length n uses $\Theta(n^2)$ steps

Insertion Sort

Idea:

- Sort progressively larger subarrays
- $n-1$ stages, for $i = 1, 2, \ldots, n-1$
- At the end of the i^{th} stage
  - Entries originally in locations

$$A[0], A[1], \ldots, A[i]$$

    have been reordered and are now sorted
  - Entries in locations

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

    have not yet been examined or moved

Pseudocode

```
void Insertion Sort(int [] A)
  for i from 1 to n − 1 do
    j = i
    while ((j > 0) and (A[j] < A[j − 1])) do
      {Swap A[j − 1] and A[j]}
      tmp = A[j]
      A[j] = A[j − 1]
      A[j − 1] = tmp
      j = j − 1
    end while
  end for
```


Example

A:					
	2	6	3	1	4

Idea: insert $A[i]$ in the correct position in $A[0], \ldots, A[i-1]$

- initially, $i = 0$ and $A[0] = 2$ is sorted

$\underline{i = 1}$

- no swaps
- $A[0], A[1]$ sorted

A:					
	2	6	3	1	4

$\underline{i = 2}$

- swap $A[2]$ & $A[1]$
- $A[0], A[1], A[2]$ sorted

A:					
	2	3	6	1	4

Example (cont.)

$\underline{i = 3}$

- swap $A[3]$ & $A[2]$, swap $A[2]$ & $A[1]$, swap $A[1]$ & $A[0]$
- $A[0], A[1], A[2], A[3]$ sorted

A:

1	2	3	6	4

$\underline{i = 4}$

- swap $A[4]$ & $A[3]$
- $A[0], A[1], A[2], A[3], A[4]$ sorted

A:

1	2	3	4	6

Finished! $A[0], \ldots, A[4]$ sorted

Inner Loop: Loop Invariant

Loop Invariant: at the beginning of each execution of the inner loop body

- $i, j \in \mathbb{N}$
- $1 \leq i < n$ and $0 < j \leq i$
- $A[h] \leq A[h+1]$ for $0 \leq h < j-1$ and $j \leq h < i$
- if $j > 0$ and $j < i$ then $A[j-1] \leq A[j+1]$
- Entries of A have been reordered; otherwise unchanged

Inner Loop: Interpretation of Loop Invariant

A:

	$j-1$	j	$j+1$		i	

sorted sorted

Can be used to establish that the following holds at the *end* of each execution of the inner loop body:

- i and j are integers such that $0 \leq j \leq i-1 \leq n-2$
- $A[0], \ldots, A[j-1]$ are sorted
- $A[j], \ldots, A[i]$ are sorted (so that $A[0], \ldots, A[i]$ are sorted if $j = 0$)
- if $j > 0$ and $j < i$, then $A[j-1] \leq A[j+1]$, so that $A[0], \ldots, A[i]$ are sorted if $A[j-1] \leq A[j]$

It follows that $A[0], \ldots, A[i]$ are sorted when this loop terminates.

Inner Loop: Loop Variant and Application

Loop Variant: $f(n, i, j) = j$

- decreasing integer function
- when $f(n, i, j) = 0$ we have $j = 0$ and the loop terminates

Application:

- initial value is i
- worst-case number of iterations is i

Outer Loop: Semantics

Once again, the outer **for** loop can be rewritten as a **while** loop for analysis. Since the inner loop is already a **while** loop, the new outer **while** loop would be as follows.

$i = 1$
while $i \leq n - 1$ **do**
  $j = i$
  Inner loop of original program
  $i = i + 1$
end while

This program will be analyzed in order establish the correctness and efficiency of the original one.

Outer Loop

Loop Invariant: at the beginning of each execution of the outer loop body:

- $1 \leq i < n$
- $A[0], A[1], \ldots, A[i-1]$ are sorted
- Entries of A have been reordered; otherwise unchanged.

Thus, the loop invariant, final execution of the loop body, and failure of the loop test establish that

- $A[0], \ldots, A[i-1]$ are sorted,
- as $i = n$ when the loop terminates, A is sorted

Loop Variant: $f(n, i) = n - i$

- number of iterations is $f(n, 1) = n - 1$

Analysis of Insertion Sort

Worst-case:

- inner loop iterates i times (constant steps per iteration)
- outer loop iterates $n - 1$ times
- total number of steps is at most

$$c_0 + \sum_{i=1}^{n-1} c_1 i = c_0 + c_1 \frac{n(n-1)}{2}$$

Conclusion: Worst-case running time is in $O(n^2)$

Analysis of Insertion Sort, Concluded

Worst-Case, Continued: For every integer $n \geq 1$ consider the operation on this algorithm on an input array A such that

- the length of A is n
- the entries of A are *distinct*
- A is sorted in **decreasing** order, instead of increasing order

It is possible to show that the algorithm uses $\Omega(n^2)$ steps on this input array.

Conclusion: The worst-case running time is in $\Theta(n^2)$

Best-Case: $\Theta(n)$ steps are used in the best case

- Proof: *Exercise.* Consider an array whose entries are already sorted as part of this.

Bubble Sort

Idea:

- Similar, in some ways, to "Selection Sort"
- Repeatedly sweep from right to left over the unsorted (rightmost) portion of the array, keeping the smallest element found and moving it to the left
- Result: After the i^{th} stage,

$$A[0], A[1], \ldots, A[i-1]$$

are the i smallest elements in the entire array, in sorted order

Pseudocode

```
void Bubble Sort(int [] A)
  for i from 0 to n − 1 do
    for j from n − 2 down to i do
      if A[j] > A[j + 1] then
        {Swap A[j] and A[j + 1]}
        tmp = A[j]
        A[j] = A[j + 1]
        A[j + 1] = tmp
      end if
    end for
  end for
```


Comparisons

All three algorithms have worst-case complexity $\Theta(n^2)$

- Selection sort only swaps $O(n)$ elements, even in the worst case. This is an advantage when exchanges are more expensive than comparisons.
- On the other hand, Insertion sort has the best "best case" complexity. It also performs well if the input as already partly sorted.
- Bubble sort is generally not used in practice.

Note: Asymptotically faster algorithms exist and will be presented next. These "asymptotically faster" algorithms are better choices when the input size is large and worst-case performance is critical.

Reference

Further reading and Java code:

Data Structures & Algorithms in Java, Chapter 3

and,

Introduction to Algorithms, Chapter 2.1