Comparison-Based Array Sorting

Reading: CLRS Ch. 7, CLR Ch. 8

In these notes, we study some classical comparison-based array sorting algorithms, prove their correctness, and analyze their running times.

Terminology and Notation

- An array $A$ is a finite function from indices in the range $[1..\text{length}(A)]$ to elements of the array.

- The array segment notation $A[lo..hi]$ indicates the contiguous segment of $A$ between indices $lo$ and $hi$, inclusive. If $hi < lo$, then $A[lo..hi]$ is the empty segment. For definitions based on array segments, the array $A$ is understood to be the segment $A[1..\text{length}(A)]$.

- The elements of an array segment $A[lo..hi]$, written $\text{elts}(A[lo..hi])$, is defined as the bag $\{A[i] \mid lo \leq i \leq hi\}$.

- We assume that array elements are compared via a comparison predicate $\leq(v_1, v_2)$ that determines if $v_1 \leq v_2$.
  
  - An array segment $A[lo..hi]$ is sorted, written $\text{sorted}(A[lo..hi])$, if for all $lo \leq i < hi$, $\leq(A[i], A[i+1])$.
  
  - We typically imagine array elements as records with a distinguished key field and extra satellite data. In practice, may have pointers to components or entire record. The $\leq$ operator abstracts over these details. In examples, values are usually numbers or letters.
  
  - We can define all other comparison operators in terms of $\leq$. E.g.:
    
    ```
    \begin{align*}
    \text{eq}(a,b) &\quad \text{return } \leq(a,b) \text{ and } \leq(b,a) \\
    \text{lt}(a,b) &\quad \text{return } \leq(a,b) \text{ and } \not(\leq(b,a))
    \end{align*}
    ```
    
    $\text{gt}$ and $\geq$ can be defined similarly.
    
  - In some contexts, it’s helpful to assume that all elements are distinct.

- The array sorting algorithms we study will be expressed in CLR/CLRS pseudocode, not Haskell. Unlike Haskell, such pseudocode has variables whose values change over time. To distinguish the values of a variable at different points in time, we will often subscript variables with some measure of time $t$, as in $A_t$ or $i_t$. 

1
The Comparison-Based Array Sorting

Comparison-based Array Sorting Specification:

Let \( A \) be an array \( A[1..n] \) with comparison predicate \( \leq \). \texttt{sort}(A) is a comparison-based sorting algorithm if the following relationships hold between the initial array \( A_{init} \) before the call to \texttt{sort} and the final array \( A_{final} \) after \texttt{sort} returns:

\[
\begin{align*}
(\text{sort1}) & \quad \text{elts}(A_{final}) = \text{bag elts}(A_{init}) \\
(\text{sort2}) & \quad \text{sorted}(A_{final})
\end{align*}
\]

Notes:

- The running time of a comparison-based sorting algorithm is usually measured in terms of the number of \( \leq \) operations performed. In addition to counting number of comparisons, may also want to measure the number of assignments to array slots and temporary variables.

- Often want to model the temporary space required (space in addition to the given array). A sorting algorithm is in-place if the amount of temporary space is constant. If the space depends on the length \( n \) of the array, then it is not in-place. Non-tail-recursive sorting procedures require “stack” space, and so are not in-place.

- A sorting algorithm is stable if it preserves the relative positions of elements with equal keys.

- The following swap procedure is handy for many array sorting algorithms:

  ```
  swap(A, i, j)
  \text{temp} \leftarrow A[i]
  A[i] \leftarrow A[j]
  A[j] \leftarrow \text{temp}
  ```
Insertion Sort


Example:

```
1 2 3 4 5 6 7 8 9 10
```

Algorithm:

```
1 Insertion-Sort(A)
2 for i ← 2 to length[A]
3    do Insert(A, i)
4
5 Insert(A, h)
6 key ← A[h]
7 j ← h
8 ▷ Search for index j of insertion point.
9 while j > 1 and lt(key, A[j-1]) do
11    j ← j - 1
12 ▷ Insert key at insertion point.
13 A[j] ← key
```

Note: Both Insertion-Sort and Insert may also be expressed recursively.

Analysis:

<table>
<thead>
<tr>
<th>Case</th>
<th>Recurrence</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Best</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Loop Invariants (Review)

Proving the correctness of algorithms involving loops is usually accomplished via the method of loop invariants. The steps of this method are:

1. State one or more invariants that hold among the state variables of the loop.
2. Show that the invariants hold the first time the loop is entered.
3. Show that if the loop invariants are true at the beginning of an iteration, they are true at the beginning of the next iteration i.e., after the body of the loop has executed, regardless of the path taken through the body.
4. Show that the loop terminates. This is usually done by defining a non-negative integer metric function that characterizes the size of the problem at each iteration, and showing that the metric strictly decreases at every iteration.
5. Show that the terminating state satisfies the desired correctness properties. Note that the terminating state corresponds to the beginning of an iteration that is not executed.

Insert Specification

Let \( A_{\text{init}} \) denote the state of the array before a call to Insert, and \( A_{\text{final}} \) denote the state of the array when Insert returns.

(insert1)  \( A_{\text{final}}[\text{h+1}..\text{length}(A_{\text{final}})] = A_{\text{init}}[\text{h+1}..\text{length}(A_{\text{init}})] \)

(insert2)  \( \text{elts}(A_{\text{final}}[1..h]) =_{\text{bag}} \text{elts}(A_{\text{init}}[1..h]). \)

(insert3)  If \( \text{sorted}(A_{\text{init}}[1..h-1]) \) then \( \text{sorted}(A_{\text{final}}[1..h]). \)
Correctness of Insertion-Sort

We want to show that Insertion-Sort is a correct array sorting algorithm – i.e., that it satisfies (sort1) and (sort2).

Since Insertion-Sort is defined using a for loop, it is natural to prove correctness via the method of loop invariants.

State invariants In this case there are three invariants, each of which is indexed by the index variable \( i \) of the for loop, which ranges between 2 and \( \text{length}(A) \).

- (isortLI1) \( A_i[1..\text{length}(A_i)] = A_{\text{init}}[1..\text{length}(A_{\text{init}})] \)
- (isortLI2) \( \text{elts}(A_i[1..i-1]) = \text{bag els}(A_{\text{init}}[1..i-1]). \)
- (isortLI3) \( \text{sorted}(A_i[1..i-1]). \)

Show invariants hold on entry to loop On entry to loop, \( i = 2 \) and \( A_{\text{init}} = A_2 \).

- (isortLI1) \( A_2[2..\text{length}(A_2)] \) and \( A_{\text{init}}[2..\text{length}(A_{\text{init}})] \) are just different notations for the same array segment.
- (isortLI2) \( A_2[1..1] \) and \( A_{\text{init}}[1..1] \) are just different notations for the same array segment.
- (isortLI3) \( A[1..1] \) is trivially sorted.

Show each loop iteration preserves invariants Assume that the invariants hold at the beginning of iteration \( i \). We wish to show they also hold at the beginning of iteration \( i + 1 \) – i.e., after line 3 is executed.

- (isortLI1) \[
A_{i+1}[i+1..\text{length}(A_{i+1})] \\
= A_i[i+1..\text{length}(A_i)] \text{ by (insert1)} \\
= A_{\text{init}}[i+1..\text{length}(A_{\text{init}})] \text{ by (isortLI1) assumption}
\]

- (isortLI2) \[
\text{elts}(A_{i+1}[1..(i+1) - 1]) \\
= \text{bag elts}(A_{i+1}[1..i]) \text{ by arithmetic} \\
= \text{bag elts}(A_i[1..i]) \text{ by (insert2)} \\
= \text{bag bagIns}(A_i[i], A_i[1..i-1]) \text{ by defn. of elts} \\
= \text{bag bagIns}(A_{\text{init}}[i], A_{\text{init}}[1..i-1]) \text{ by (isortLI1) assumption} \\
= \text{bag elts}(A_{\text{init}}[1..i]) \text{ by (isortLI2) assumption} \\
= \text{bag elts}(A_{\text{init}}[1..i]) \text{ by defn. of elts}
\]

- (isortLI3) By assumption, \( A_i[1..i-1] \) is sorted. By (insert3), after line 3 executes, \( A_{i+1}[1..i] \) is sorted. Since \( A_{i+1}[1..i] = A_{i+1}[1..(i+1) - 1] \), (isortLI3) holds for iteration \( i + 1 \).

Show Termination The loop is a for loop over a bounded range, and each execution of line 3 terminates, the loop terminates. (We could be more formal and show this via the metric function that characterizes the size of the problem on the \( i \)th iteration as \( M(i) = \text{length}(A) - i \).

Show desired properties At the final loop test, \( i = \text{length}(i) + 1 \). (The index of a for loop is one beyond the limit when loop test fails.) (sort1) follows from (isortLI2) and (sort2) follows from (isortLI3). Note that (isortLI3) is only needed “internally”, in the proof of preservation of (isortLI2).
Correctness of Insert

Since Insert is defined using a while loop, it is natural to prove correctness via the method of loop invariants.

State invariants We assume that sorted($A_{init}[1..h-1]$). We also assume that the loop is indexed by an implicit variable $k$ that is initially 1. Here are the invariants we shall use:

- $insertLI1$ \( A_{k}[h+1..\text{length}(A_{k})] = A_{init}[h+1..\text{length}(A_{init})] \).
- $insertLI2$ \( \text{elts}(A_{k}[1..j_{k}-1]) \cup_{\text{bag}} \text{elts}(A_{k}[j_{k}+1..h]) \cup_{\text{bag}} \{\text{key}\} =_{\text{bag}} \text{elts}(A_{init}[1..h]) \).
- $insertLI3$ \( \text{sorted}(A_{k}[1..h]) \).
- $insertLI4$ \( (j_{k} = h) \text{ or } \text{key} < A_{k}[j_{k}+1]. \) (Alternatively, could use the stronger invariant: $\text{key}$ is $< \text{each element in } A_{k}[j_{k}+1..h].$)

Show invariants hold on entry to loop On entry to loop, $A_{1} = A_{init}$ and $j_{1} = h$. Show each invariant is true:

Show each loop iteration preserves invariants Assume that all six invariants hold at line 9. We shall write $j_{k}$ for the value of $j$ at the beginning of the $k$th iteration of the loop. Note that $j_{k+1} = j_{k} - 1$. Similarly, we write $A_{k}$ for the value of the array at the beginning of the $k$th iteration. Note that by line 10, $A_{k+1}$ differs from $A_{k}$ only at index $j_{k}$: \( A_{k+1}[j_{k}] = A_{k}[j_{k} - 1] \) and \( A_{k+1}[j_{k}+1..h] = A_{k}[j_{k}+1..h] \).
Prove Termination

Show desired properties
Selection Sort


*Example:*  

```
UNSORTED  
DSORTED  
DSORTED  
DSORTED  
DSORTED  
DSORTED  
DSORTED  
DSORTED  
```

*Algorithm:*  

```
Selection-Sort(A)
   for i ← 1 to length[A] - 1 do
      swap(A, i, Min-Index(A, i, length[A]))

Min-Index(A, lo, hi)
   ▷ Assume lo and hi are legal subscripts,
   ▷ and hi ≥ lo.
   min_index ← lo
   for i ← lo + 1 to hi do
      if lt(A[i], A[min_index]) then
         min_index ← i
   return min_index
```

*Correctness:*  

- Claim 1: $\text{Min-Index}(A, lo, hi)$ returns the index of (an occurrence of) the least element in $A[lo..hi]$. Can be proven by loop invariants (details omitted).
- Claim 2: $A_{final}$ is a sorted permutation of $A_{init}$ after $\text{Selection-Sort}(A_{init})$. Can be proven by loop invariants (details omitted).

*Analysis:*  

<table>
<thead>
<tr>
<th>Case</th>
<th>Recurrence</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Best</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Merge Sort

Idea: Recursively sort subarrays and then merge them into a single sorted array.

Example:

```
USORTED
USORTED
USORTED
USORTED
USORTED
NOSURTDE
NOSURTDE
NOSURTDE
U S O R T D E
U S O R T D E
U S O R T D E
UNORT
UNORT
SDUNORT
SDUNORT
```

Algorithm:

```
Merge-Sort(A)
MSort(A, 1, length[A])

MSort(A, lo, hi)
if lo < hi then
    mid ← (lo + hi) div 2
    MSort(A, lo, mid)
    MSort(A, mid + 1, hi)
    Merge(A, lo, mid, hi)
```

Correctness:

- Claim 1: If A[lo..mid] is sorted and A[mid+1..hi] is sorted, then A is sorted after `Merge(A, lo, hi)`. Can be proven via loop invariants (details omitted).

- Claim 2: When `MSort(A_{init}, lo, hi)` returns, A_{final}[lo..hi] is sorted. Can be proven by induction on (hi - lo) (details omitted).

Analysis:

<table>
<thead>
<tr>
<th>Case</th>
<th>Recurrence</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Best</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
QuickSort

Idea: Partition the elements about a “pivot” so that all those less than the pivot are moved to the left-hand side of the array, and all those greater than the pivot are moved to the right-hand side of the array. Recursively apply this approach to each partition.

Algorithm:

Quick-Sort(A)

    QSort(A, 1, length[A])

QSort(A, lo, hi)

    if lo < hi then
        p <- Partition(A, lo, hi)
        QSort(A, lo, p)
        QSort(A, p + 1, hi)

Partition(A, lo, hi)

    ▶ Permutes the elements of A and returns a p (lo ≤ p < hi)
    ▶ such that all elements in A[lo..p] are less than
    ▶ or equal to all elements in A[p+1..hi].

Correctness:

Assuming Partition works as advertised, use induction to prove the following claim: When QSort(A, lo, hi) returns, A[lo..hi] is sorted.

• Base Case:

• Inductive Case:

What would happen if Partition allowed one of the partitions to be empty?
Lomuto Partitioning

Idea: Scan from left to right, maintaining three areas: those less than or equal to the pivot; those greater than the pivot, and those as yet unprocessed.

Algorithm:

Lomuto-Partition(A, lo, hi)
  pivot ← A[hi]
  lastless ← lo - 1
  scan ← lo
  while scan ≤ hi do
    ▶ Loop invariants at this point:
    ▶ (1) A_{scan[lo..lastless\,scan]} contains all elements of A_{init[1..(scan - 1)]} that are ≤ pivot.
    ▶ (2) A_{scan[lastless\,scan+1..scan-1]} contains all elements of A_{init[1..(scan - 1)]} that are > pivot.
    ▶ (3) A_{scan[scan..hi]} = A_{init[scan..hi]}.
    if leq(A[scan], pivot) then
      swap(A, lastless + 1, scan)
      lastless ← lastless + 1
    scan ← scan + 1
    ▶ Guarantee that returned partitions are non-empty.
    ▶ If no elements are > pivot, make upper partition
    ▶ a singleton of pivot.
    if lastless = hi then
      return hi - 1
    else
      return lastless

Example:

```
  ALGORITHM
```

What is the worst-case running time of Lomuto-Partition?
Two-Finger Partitioning

*Idea:* Place two fingers at opposite ends of array, and move them until each comes to an element out of position. Swap these and continue.

*Algorithm:*

Two-Finger-Partition(A, lo, hi)

1. pivot ← A[lo]
2. left ← lo - 1
3. right ← hi + 1
4. while true do
5.     Let i be the index variable of this loop, starting at 1.
6.     Loop invariants at this point:
7.     (1) elts(A_i[lo..left_i]) ∪_bag elts(A_i[right_i..hi])
8.            =_bag elts(A_init[lo..left_i]) ∪_bag elts(A_init[right_i..hi]).
9.     (2) All elements of A_i[lo..left_i] are ≤ pivot.
10.    (3) All elements of A_i[right_i..hi] are ≥ pivot.
11.    (4) A_i[left_i+1..right_i-1] = A_init[left_i+1..right_i-1].
12.     repeat right ← right - 1
13.       until leq(A[right], pivot)
14.     repeat left ← left + 1
15.       until leq(pivot, A[left])
16.     if left < right then
17.       swap(A, left, right)
18.     else
19.       return right ▷ Non-local exit from loop

*Example:*

```
QUICKSORT
```
Analysis of Two-Finger Algorithm

Fleshing out the following details is left as an exercise for the reader.

- Show that loop invariant is true when the while loop is first entered.
- Show that each iteration of the loop preserves the loop invariant.
- Show that loop terminates.
- What are possible configurations of left and right at termination?
- Show that the terminating state verifies the desired property.
- Show the array is never accessed out-of-bounds.
- Show that partitions are always non-empty.
- Are partitions always non-empty if we change pivot to $A[hi]$?
- Show every element of $A[lo..right]$ is $\leq$ every element of $A[right+1..hi]$ at termination.
- What is the worst-case running time of Two-Finger-Partition?
Analysis of QuickSort

Worst-case partitioning: \( T(n) = \) 
Solution =

Best-case partitioning: \( T(n) = \) 
Solution =

If best- and worst-case alternated? \( T(n) = \) 
Solution =

If every split was 99:1? \( T(n) = \) 
Solution =

How would the following partitioning algorithms affect behavior? Running time?

\( \text{Median-Of-3-Partition}(A, \, \text{lo}, \, \text{hi}) \)
\( \quad \triangleright \text{Middle-Index}(A, \, \text{i}, \, \text{j}, \, \text{k}) \text{ returns index of middle elt of } A[i], \, A[j], \, A[k] \)
\( \quad \text{swap}(A, \, \text{lo}, \, \text{Middle-Index}(A, \, \text{lo}, \, \text{hi}, \, (\text{lo} + \text{hi}) \text{ div } 2)) \)
\( \quad \text{Two-Finger-Partition}(A, \, \text{lo}, \, \text{hi}) \)

\( \text{Randomized-Partition}(A, \, \text{lo}, \, \text{hi}) \)
\( \quad \triangleright \text{Random}(\text{lo}, \, \text{hi}) \text{ returns a random integer between lo and hi, inclusive} \)
\( \quad \text{swap}(A, \, \text{lo}, \, \text{Random}(\text{lo}, \, \text{hi})) \)
\( \quad \text{Two-Finger-Partition}(A, \, \text{lo}, \, \text{hi}) \)
Average-Case Analysis of Randomized QuickSort

Assume all elements distinct.

What is probability that lower partition has 1 element?

What is probability that lower partition has \( i \) elements (\( 2 \leq i \leq n - 1 \))?

\[ T(n) = \]

Use substitution method to show that \( T(n) \leq an \lg(n) + b \).
Use fact (CLR 8.4-5) that \( \sum_{k=1}^{n-1}(k \cdot \lg(k)) \leq (1/2)n^2 \lg(n) - (1/4)n^2 \).