

Comparison-Based Sorting (Revised)

Reading: *CLR* §7,8,13

The Comparison-Based Sorting Problem

Given (1) an array $A[1..n]$ and (2) a comparison predicate $lt(v_1, v_2)$ that determines if $v_1 < v_2$, modify A so that $lt(A[i], A[i+1])$ for $0 < i < n$. The running time of a comparison-based sorting algorithm is measured in terms of the number of lt operations performed.

In these notes, we present classical comparison-based sorting algorithms, prove that they are correct (i.e., the algorithms terminate and leave the input array sorted), and analyze their running times.

Notes:

- The notation $A[i..j]$ indicates the contiguous segment of A between indices i and j , inclusive. If $j < i$, then $A[i..j]$ is the empty segment.
- 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 lt operator abstracts over these details. In examples, values are usually numbers or letters.
- In some contexts, care if $v_1 = v_2$. For these cases, can use eq and leq operators, defined in terms of lt :

```
eq(a,b)
  return not(lt(a,b)) and not(lt(b,a))
```

```
leq(a,b)
  return lt(a,b) or eq(a,b)
```

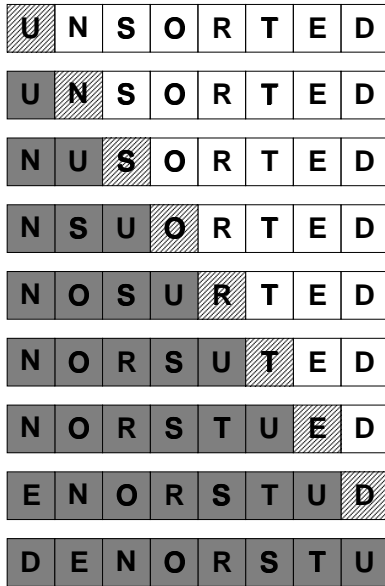
- In some contexts, it's helpful to assume that all elements are distinct.
- In addition to counting number of comparisons, may also want to measure the number of assignments to array slots and temporary variables.
- May also want to model the temporary space required (space in addition to the given 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.
- A sorting algorithm is **stable** if it preserves the relative positions of equal elements.
- Array-based sorting algorithms can be adapted to lists.
- The following **swap** procedure is handy for many sorting algorithms:

```
swap(A, i, j)
  temp ← A[i]
  A[i] ← A[j]
  A[j] ← temp
```

Insertion Sort

Idea: On the i th step of the algorithm, insert $A[i]$ into the sorted segment $A[1..i-1]$.

Example:



Algorithm:

```

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

Case	Recurrence	Solution
<i>Worst</i>		
<i>Best</i>		
<i>Average</i>		

Loop Invariants

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. Showing that if the loop invariants are true at the beginning of an iteration, they are true 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 property.

Correctness of Insert

Correctness Claim: If $A[1..i-1]$ is sorted when $\text{Insert}(A, i)$ is invoked, then A is sorted when the invocation returns.

Proof Correctness Claim: Since Insert is defined as a loop, it is natural to prove correctness via the method of loop invariants.

State invariants Here are the invariants we shall use:

1. $A[1..j-1]$ is sorted.
2. $A[j+1..i]$ is sorted.
3. $(j = i)$ or $(j = 1)$ or $\text{leq}(A[j-1], A[j+1])$. (Alternatively, could use the stronger invariant: each element in $A[1..j-1]$ is less than or equal to each element in $A[j+1..i]$.)
4. $(j = i)$ or $\text{lt}(\text{key}, A[j+1])$. (Alternatively, could use the stronger invariant: key is less than or equal to each element in $A[j+1..i]$.)

Show invariants hold on entry to loop On entry to loop, $j = i$. We show each invariant is true when j is substituted for i :

1. $A[1..i-1]$ is sorted by assumption.
2. $A[i+1..i]$ is empty, and therefore trivially sorted
3. $(i = i)$ is trivially true.
4. $(i = i)$ is trivially true.

Show each loop iteration preserves invariants Assume that all four 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 .

1. The segment $A_k[1..j_k-1]$ is assumed to be sorted. In the loop body, only slot j_k of A_k is changed, so $A_{k+1}[1..j_k-1]$ is still sorted, and so certainly is $A_{k+1}[1..j_k-2] = A_{k+1}[1..j_{k+1}-1]$.
2. By assumption, $A_k[j_k+1..i]$ is sorted. In the loop body, only slot j_k of A_k is changed, so $A_{k+1}[j_k+1..i]$ is still sorted. By line 10, $A_{k+1}[j_k] = v = A_k[j_k-1]$. By the third invariant, v is less than or equal to $A_k[j_k+1]$, so $A_{k+1}[j_k..i] = A_{k+1}[j_{k+1}+1..i]$ is sorted. In the boundary case where $j_k = i$, $A_{k+1}[j_{k+1}+1..i]$ contains a single element and so is trivially sorted. (Note that when $j_k = 1$, the loop body is not executed.)
3. By the definition of j_{k+1} , $A_{k+1}[j_{k+1}+1] = A_{k+1}[j_k]$, and by line 10, this is equal to $A_k[j_k-1]$. By invariant 1, is sorted, so $\text{leq}(A_k[j_k-2], A_k[j_k-1])$. Since line 10 only affects slot j_k of A , $A_k[j_k-2] = A_{k+1}[j_k-2] = A_{k+1}[j_{k+1}-1]$. Combining these facts gives our goal: $\text{leq}(A_{k+1}[j_{k+1}-1], A_{k+1}[j_{k+1}+1])$. Note that invariant 3 was not needed to prove itself.

4. The body on executes if $\text{lt}(\text{key}, A_k[j_k-1])$. By line 10, $A_{k+1}[j_k] = A_k[j_k-1]$, By line 11, $A_{k+1}[j_{k+1}+1] = A_{k+1}[j_k]$, Therefore, it follows that $\text{lt}(\text{key}, A_{k+1}[j_{k+1}+1])$.

Prove Termination Here a good metric function characterizing the size of the problem on the k th iteration is $m(k) = j_k$. At the entry to the loop, $m(k) = i$, and $m(k)$ strictly decreases by one on each iteration due to line 11. Because the loop only continues as long as $j > 1$, it will eventually stop when $m(k)$ decreases to 1. It may also stop before this point if $\text{lt}(\text{key}, A[j-1])$.

Show desired property The loop terminates under one of two conditions: (a) $j_k=1$ or (b) $\text{leq}(A[j_k]-1, \text{key})$. In case (a), invariant 4 shows that $\text{lt}(\text{key}, A[2])$ and invariant 2 shows that $A[2..i]$ is sorted, so $A[1..i]$ is sorted. In case (b), invariant 1 shows that $A[1..j_k-1]$, condition (b) itself shows $\text{leq}(A[j_k]-1, \text{key})$, invariant 4 shows $\text{lt}(\text{key}, A[j_k]+1)$, and invariant 2 shows that $A[j_k..i]$ is sorted. We conclude from these four facts that $A[1..i]$ is sorted. Note that invariant 3 is not used to show the desired property. It is only used “internally” to show that the other invariants hold.

Correctness of Insertion-Sort

Correctness Claim: A is sorted when the invocation `Insertion-Sort(A)` returns.

Proof Correctness Claim: Since `Insertion-Sort` is defined as a loop, it is natural to prove correctness via the method of loop invariants.

State invariants In this case there is a single invariant: each time the loop is entered, $A[1..i-1]$ is sorted.

Show invariants hold on entry to loop On entry to loop, $i = 2$, and $A[1..1]$ is trivially sorted.

Show each loop iteration preserves invariants Assume that the invariant holds a line 2. We wish to show it also holds after line 3 is executed. We shall write i_k for the value of i at the beginning of the k th iteration of the loop, where k starts at 1. Note that i_k is simply $k + 1$. By assumption, $A[1..i_k-1]$ is sorted. This satisfies the precondition for the correctness of `Insert`. By the correctness of `Insert`, after line 3 executes, $A[1..i_k]$ is sorted. Because $i_{k+1} = i_k + 1$, we know $A[1..i_{k+1}-1]$ is sorted.

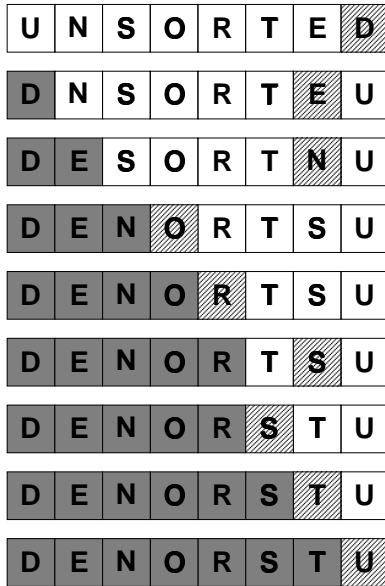
Prove Termination The loop is a `for` loop over a bounded range, so it clearly terminates. (We could be more formal and show this via the metric function that characterizes the size of the problem on the k th iteration as $m(k) = \text{length}[A] - i_k$.)

Show desired property At the final loop test, $i = \text{length}[A] + 1$. (The index of a `for` loop is one beyond the limit when loop test fails.) By the invariant, $A[1..\text{length}[A]]$ is sorted, which is what we wanted to prove.

Selection Sort

Idea: On the i th step, store into $A[i]$ the smallest element of $A[i..n]$.

Example:



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 the least element in $A[lo..hi]$. Can prove this via the following invariant, which holds at the beginning of the i th iteration of the **for** loop: $A[\text{min_index}]$ is a least element in $A[lo..i-1]$.
- Claim 2: A is sorted after $\text{Selection-Sort}(A)$. Can prove this via the following invariants: after step i , (1) The elements of $A[1..i]$ are less than or equal to all elements in $A[i+1..n]$ and (2) $A[1..i]$ is sorted.

Analysis:

Case	Recurrence	Solution
<i>Worst</i>		
<i>Best</i>		
<i>Average</i>		

Bubble Sort

Idea: On the i th step, scan $A[i..n]$ from right to left, exchanging adjacent elements that are out of order. Repeat until a scan finds no elements out of order.

Example:

U	N	S	O	R	T	E	D
D	U	N	S	O	R	T	E
D	E	U	N	S	O	R	T
D	E	N	U	O	S	R	T
D	E	N	O	U	R	S	T
D	E	N	O	R	U	S	T
D	E	N	O	R	S	U	T
D	E	N	O	R	S	T	U
D	E	N	O	R	S	T	U

Algorithm:

```

Bubble-Sort(A)
  lo ← 1
  changed ← false
  repeat
    changed ← Bubble-Down(A, lo, length[A])
    lo ← lo + 1
  until not changed

Bubble-Down(A, lo, hi)
  changed ← false
  for i ← hi downto lo+1 do
    if lt(A[i], A[i-1]) then
      swap(A, i, i-1)
      changed ← true
  return changed
  
```

Correctness:

- Claim 1: If `Bubble-Down(A, lo, hi)` returns false, $A[lo..hi]$ is sorted.
- Claim 1: If `Bubble-Down(A, lo, hi)` returns true, then after it returns, $A[lo]$ is a least element of $A[lo..hi]$. Can prove this via the following invariant, which holds at the beginning of each iteration of the `for` loop: $A[i]$ is a least element of $A[i..hi]$.
- Claim 3: A is sorted after `Bubble-Sort(A)`. Can prove this via the following invariant: after step i , either the array is completely sorted, or (1) The elements of $A[1..i]$ are less than or equal to all elements in $A[i+1..n]$ and (2) $A[1..i]$ is sorted.

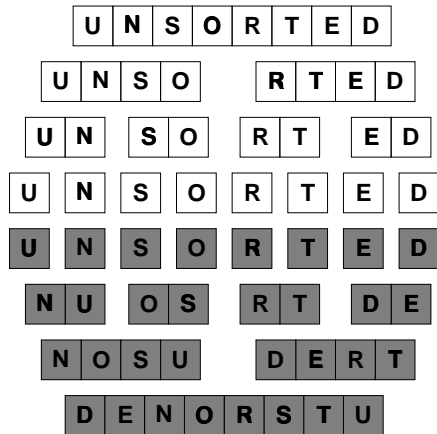
Analysis:

Case	Recurrence	Solution
<i>Worst</i>		
<i>Best</i>		
<i>Average</i>		

Merge Sort

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

Example:



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)
  
```

```

Merge(A, lo, mid, hi)
  n ← (hi - lo) + 1
  ▷ Merge elements into length-n
  ▷ temporary array B.
  B ← newArray(n)
  left ← lo
  right ← mid + 1
  for i = 1 to n do
    if left ≤ mid
      and (right > hi
         or lt(A[left], A[right]))
    then
      B[i] ← A[left]
      left ← left + 1
    else
      B[i] ← A[right]
      right ← right + 1
  ▷ Copy elements from B back to A
  left ← lo
  for i = 1 to n do
    A[left] ← B[i]
    left ← left + 1
  
```

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 by a loop invariant (details omitted).
- Claim 2: When $MSort(A, lo, hi)$ returns, $A[i..s]$ sorted. Can be proven by induction on $(hi - lo)$.

Analysis:

Case	Recurrence	Solution
<i>Worst</i>		
<i>Best</i>		
<i>Average</i>		

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)
  ▷ Rearrange A into non-empty segments A[lo..p] and A[p+1..hi]
  ▷ such that all elements in the left segment are less than
  ▷ all elements in the right one. Return partitioning index p.
```

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:

- General Case:

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

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)
  pivot ← A[lo]
  left ← lo - 1
  right ← hi + 1
  while true do
    ▷ Loop invariant at this point:
    ▷ (1)  $A[lo..left]$  contains elements  $\leq$  pivot.
    ▷ (2)  $A[right..hi]$  contains elements  $\geq$  pivot.
    ▷ (3)  $A[left+1..right-1]$  hasn't been processed yet.
    repeat right ← right - 1
      until leq(A[right], pivot)
    repeat left ← left + 1
      until leq(pivot, A[left])
    if left < right then
      swap(A, left, right)
    else
      return right ▷ Non-local exit from loop
```

Example:

Q	U	I	C	K	S	O	R	T
---	---	---	---	---	---	---	---	---

Analysis of Two-Finger Algorithm

- Show that loop invariant is correct.
- Show that loop terminates.
- What are possible configurations of left and right at termination?
- Show arrays never accessed out-of-bounds.
- Show that partitions are always non-empty.
- Are partitions always non-empty if we change pivot to $A[\text{hi}..?]$
- Show every element of $A[\text{lo}..\text{right}]$ is leq every element of $A[\text{right}+1..\text{hi}]$ at termination.
- What is the worst case running time of **Two-Finger-Partition**?

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 invariant at this point:
    ▷ (1)  $A[lo..lastless]$  contains elements  $\leq pivot$ .
    ▷ (2)  $A[lastless+1..scan-1]$  contains elements  $> pivot$ .
    ▷ (3)  $A[scan..hi]$  hasn't been processed yet.
    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 gt pivot, make upper partition
    ▷ a singleton of pivot.
  if lastless = hi then
    return hi - 1
  else
    return lastless
```

Example:

Q	U	I	C	K	S	O	R	T
---	---	---	---	---	---	---	---	---

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?

`Median-Of-3-Partition(A, lo, hi)`

▷ *Middle-Index(A, i, j, k)* returns index of middle elt of $A[i]$, $A[j]$, $A[k]$

`swap(A, lo, Middle-Index(A, lo, hi, (lo + hi) div 2))`

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

`Randomized-Partition(A, lo, hi)`

▷ *Random(lo, hi)* returns a random integer between lo and hi , inclusive

`swap(A, lo, Random(lo, hi))`

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

Analysis of QuickSort

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 $T(n) \leq \lg(n) + b$.
Use fact (CLR 8.4-5) that $\sum_{k=1}^{n-1} k \leq (1/2)n^2 \lg(n) - (1/4)n^2$.

Tree Sort

Idea: Insert elements of array from left to right into a binary search tree. Once the tree is built, perform an in-order traversal over the tree that puts the elements back into the array.

Example: Use tree sort to sort UNSORTED.

Analysis:

Case	Recurrence	Solution
<i>Worst</i>		
<i>Best</i>		
<i>Average</i>		