COMPARISON-BASED SORTING

Reading: CLR Chapters 7 & 8.

The Comparison-Based Sorting Problem

Given

- An array \texttt{A[1..n]} of values.
- A comparison predicate \texttt{less(v1, v2)} that determines if \( v_1 < v_2 \).

modify \texttt{A} so that \texttt{lesseq(A[i], A[i+1])} for \( 0 < i < n \). Measure the running time of a comparison-based sorting algorithm by the number of \texttt{less} operations performed.

Notes:

- The notation \texttt{A[i..j]} indicates the contiguous segment of \texttt{A} between indices \( i \) and \( j \), inclusive. If \( j < i \), then \texttt{A[i..j]} is the empty segment.

- Typically imagine array elements as records with a distinguished \texttt{key} field and extra \texttt{satellite data}. In practice, may have pointers to components or entire record. The \texttt{less} 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 \texttt{eq} and \texttt{lesseq} operators:

\[
\begin{align*}
\text{\texttt{eq(a,b)}} & \quad \text{return } \neg(\text{\texttt{less(a,b)}}) \text{ and } \neg(\text{\texttt{less(b,a)}}) \\
\text{\texttt{lesseq(a, b)}} & \quad \text{return } \text{\texttt{less(a,b)}} \text{ or } \text{\texttt{eq(a,b)}}
\end{align*}
\]

- In some contexts, helpful to assume that all elements are distinct.

- May also want to measure the number of assignments to arrays and temporary variables.

- May want to model the temporary space required (space in addition to the given array). A sorting algorithm is \texttt{in-place} if the amount of temporary space is constant.

- A sorting algorithm is \texttt{stable} if it preserves the relative positions of equal elements.

- Array-based sorting algorithms can be adapted to lists.

- The \texttt{swap} procedure is handy for many sorting algorithms:

\[
\begin{align*}
\text{\texttt{swap(A, i, j)}} & \quad \text{\texttt{temp } \leftarrow \texttt{A[i]}} \\
& \quad \text{\texttt{A[i] } \leftarrow \texttt{A[j]}} \\
& \quad \text{\texttt{A[j] } \leftarrow \texttt{temp}}
\end{align*}
\]
Insertion Sort


Invariant: After the ith step of the algorithm, $A[1..i]$ is sorted.

Example:

```
UNSORTED
UNSORTED
USORTED
USORTED
USORTED
USORTED
USORTED

D E N O R S T U
```

Algorithm:

```
Insertion-Sort(A)
    for i ← 2 to length[A]
        do Insert(A, i)

Insert(A, i)
    key ← A[i]
    j ← i
    {Search for index j of insertion point.}
    while j > 1 and less(key, A[j - 1])
        j ← j - 1
    {Insert key at insertion point.}
    A[j] ← key
```

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


Invariant: After step i, the elements A[1..i] are in their final sorted positions.

Example:

```
UNSORTED
DNSTEU
DSTNU
DNSTU
DNSTU
DNSTU
DNSTU
```

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 less(A[i], A[min_index])
            then min_index ← i
    return min_index
```

---------------------------------------------------------------
Bubble Sort

**Idea:** On the every step of the algorithm, scan $A$ from left to right and exchange adjacent elements that are out of order. Repeat until a scan finds no elements out of order.

**Invariant:** After step $i$, (at least) the elements $A[(n + 1 - i)..n]$ are in their final sorted positions.

**Example:**

```
UNSORTED
NSORTEDU
NORSEDTU
NOREDSTU
NOEDRSTU
NEDORSU
EDNORSTU
DENORSTU
```

**Algorithm:**

```
Bubble-Sort(A)
hi ← length[A]
changed ← false
repeat
    changed ← Bubble-Up(A, 1, hi)
    hi ← hi - 1
until not changed;

Bubble-Up(A, lo, hi)
changed ← false
for i ← lo to hi - 1 do
    if less(A[i + 1], A[i]) then
        swap(A, i, i+1);
        changed ← true
return changed
```
Merge Sort

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

Example:

```
UNSORTED
UNSORTED
UNSORTED
UNSORTED
UNSORTED
UNSORTED
UNSORTED
UNSORTED
UNSORTED

NOSUDERT
NOSUDERT
NOSUDERT
NOSUDERT
NOSUDERT
NOSUDERT
NOSUDERT
NOSUDERT
NOSUDERT

DENORSTU
DENORSTU
DENORSTU
DENORSTU
DENORSTU
DENORSTU
DENORSTU
DENORSTU
DENORSTU
```

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 temporary array B.}
  B ← newArray(n)
  left ← lo
  right ← mid + 1
  for i = 1 to n do
    if left ≤ mid and (right > hi or less(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
```
HeapSort

A heap is a mutable priority queue data structure supporting the following operations:

EmptyHeap()
   Return an empty heap.

BuildHeap(A)
   Construct and return a heap containing the n elements of array A in O(n) time.

Heap-Insert(H, key)
   Insert key into an n-element heap H in O(lg(n)) time.
   (Really want to insert value with key key, but this simplifies description of algorithm.)

Heap-Extract-Max(H)
   Remove and return largest-keyed value of n-element heap H in O(lg(n)) time.

Given the above operations, it's easy to construct a guaranteed O(n(lg(n))) sorting algorithm:

HeapSort(A)
   H ← BuildHeap(A)
   for i ← length[A] downto 1 do
      A[i] ← Heap-Extract-Max(H)

We will see below that the heap used by HeapSort can be stored within the argument array A, so that HeapSort can be an in-place sorting algorithm.

-----------------------------------------------------------------------------------------------
Heaps

The **binary address** of a node in a binary tree specifies the order in which it would be visited in a breadth first traversal.

Operations on binary addresses:

\[
\text{Left}(\text{address}) = 2 \times \text{address} \\
\text{Right}(\text{address}) = (2 \times \text{address}) + 1 \\
\text{Parent}(\text{address}) = \text{address} \div 2
\]

An n-element binary tree is **complete** if the set of binary addresses of its nodes = \{1, 2, ..., n\}

A **heap** is a complete binary tree satisfying the heap condition:

At every node in a heap, the node value is \(\geq\) all the values in its subtrees.

A heap of with \texttt{heap_size} elements can be represented as an array segment \texttt{A[1..heap_size]}
Insertion and Extraction

HeapInsert(A, key)
  heap_size[A] ← heap_size[A] + 1
  A[heap_size[A]] ← key
  Bubble-Up(A, heap_size[A])

Bubble-Up(A, address)
  while address > 1 and A[Parent(address)] < A[address] do
    swap(A, address, Parent(address)) {Can get by with fewer assignments; See CLR}
    address ← Parent(address)

Analysis:

Heap-Extract-Max(A)
if heap_size[A] < 1 then
  error "heap underflow"
  max ← A[1]
  heap_size[A] ← heap_size[A] - 1
  BubbleDown(A, 1)
return max

Bubble-Down(A, address) {Called Heapify in CLR}
if Left(address) <= heap_size[A]
  and less(A[address], A[Left(address)]) then
  largest ← Left(address)
else
  largest ← address
if Right(address) <= heap_size[A]
  and less(A[largest], A[Right(address)]) then
  largest ← Right(address)
if largest ≠ address then
  swap(A, address, largest)
  Bubble-Down(A, largest)

Analysis:
Build-Heap

Naive version of Build-Heap:

Build-Heap(A)
  for i ← 1 to length[A] do
    Heap-Insert(A, A[i]) (Uses array slots for heap storage!)

Analysis:

Clever version of Build-Heap:

Build-Heap(A)
  heap_size[A] ← length[A]
  for i ← (length[A] div 2) downto 1 do
    Bubble-Down(A, i)

Analysis:

  Note that never more than \( n/2^h \) nodes of height \( h \) in a tree with \( n \) elements.
QuickSort

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.}

Assuming Partition works as advertised, prove that Quick-Sort is correct by induction:

• Base Case:

• General Case:

What happens if one of the partitions is empty?
Two-Finger Partitioning

```
Two-Finger-Partition(A, lo, hi)
pivot ← A[lo]
left ← lo - 1
right ← hi + 1
while true do
  {Loop invariant at this point:
   (3) A[left+1..right-1] hasn't been processed yet.)
  repeat
    right ← right - 1
  until lesseq(A[right], pivot)
  repeat
    left ← left + 1
  until lesseq(pivot, A[left])
  if left < right then
    swap(A, left, right)
  else
    return right \{Non-local exit from loop\}
```
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 pivot = A[hi]?
- Show every element of A[lo..right] is ≤ every element of A[right+1..hi] at termination.
- What is the worst case running time of Two-Finger-Partition?

Lomuto Partitioning

Lomuto-Partition(A, lo, hi)

\[
\begin{align*}
\text{pivot} &\leftarrow A[hi] \\
\text{lastless} &\leftarrow lo - 1 \\
\text{scan} &\leftarrow lo \\
\text{while} \ scan \leq hi \text{ do} \\
&\{\text{Loop invariant at this point:}
\begin{align*}
(1) & \ A[lo..lastless] \text{ contains elements } \leq \text{ pivot.} \\
(2) & \ A[lastless+1..scan-1] \text{ contains elements } > \text{ pivot.} \\
(3) & \ A[scan..hi] \text{ hasn’t been processed yet.}
\end{align*}
\}
\text{if lesseq}(A[scan], \text{pivot}) \text{ then} \\
&\quad \text{swap}(A, \text{lastless + 1}, \text{scan}) \\
&\quad \text{lastless} \leftarrow \text{lastless + 1} \\
&\quad \text{scan} \leftarrow \text{scan + 1}
\text{(Guarantee that returned partitions are non-empty.}
\text{If no elements are } > \text{ pivot, make upper partition a singleton of pivot.)}
\text{if lastless = hi then} \\
&\quad \text{return hi - 1}
\text{else} \\
&\quad \text{return lastless}
\end{align*}
\]
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\( (A, l_o, h_i) \)
\( (\text{Middle-Index}(A, i, j, k) \text{ returns index of middle elt of } A[i], A[j], A[k]) \)
swap\( (A, l_o, \text{ Middle-Index}(A, l_o, h_i, (l_o + h_i) \text{ div } 2)) \)
Two-Finger-Partition\( (A, l_o, h_i) \)

Randomized-Partition\( (A, l_o, h_i) \)
\( (\text{Random}(l_o, h_i) \text{ returns a random integer between } l_o \text{ & } h_i, \text{ inclusive}) \)
swap\( (A, l_o, \text{ Random}(l_o, h_i)) \)
Two-Finger-Partition\( (A, l_o, h_i) \)
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 ≤ i ≤ n-1)?

\[ T(n) = \]

Use substitution method to show that \( T(n) \leq an(\lg(n)) + b. \)

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