Algorithms - Ch2 - Sorting

Goals:

1. Start using frameworks for describing and analyzing algorithms.
2. Examine two algorithms for sorting: insertion sort and merge sort.
3. See how to describe algorithms in pseudocode.
4. Begin using asymptotic notation to express running-time analysis.
5. Learn the technique of "divide and conquer" in the context of merge sort.

Sorting

Input: a sequence of numbers \( a_1, a_2, \ldots, a_n \).

Output: a permutation (reordering) of \( a'_1, a'_2, \ldots, a'_n \) such that

\[ a'_1 \leq a'_2 \leq \ldots \leq a'_n. \]

Algorithms - Ch2 - Sorting

1. Problem first described in a "human" language - usually in English here...
2. Problem must be restated so all ambiguity is removed... this may still be in English.
3. Problem must be restated so that it can be unambiguously translated into a computer language... this is usually done using some form of 'pseudo-code'.
4. Problem must be restated in 'code'... using an appropriate computer language.

Here is the algorithm for Insertion Sort - as in 91.102

\[
\text{INSERTION-SORT}(A) \\
\text{for } j \leftarrow 2 \text{ to } n \\
\quad \text{do } k \leftarrow A[j] \\
\quad \text{while } i > 0 \text{ and } A[i] > k \\
\quad \quad \text{do } A[i+1] \leftarrow A[i] \\
\quad \quad i \leftarrow i - 1 \\
\quad A[i+1] \leftarrow k \\
\]

\[
\begin{array}{|c|c|}
\hline
\text{cost} & \text{times} \\
\hline
0 & n \times(n-1) \\
1 & n \times(n-1) \\
1 & \frac{n(n-1)}{2} \\
1 & \frac{n(n-1)}{2} \\
0 & n \times(n-1) \\
1 & n \times(n-1) \\
1 & \frac{n(n-1)}{2} \\
0 & n \times(n-1) \\
\hline
\end{array}
\]
Correctness - how?

Loop Invariants: statements such that:
1. Initialization: The statement is true before the first execution of the body of the loop.
2. Maintenance: If the statement is true before any iteration of the loop, it remains true before the next iteration
3. Termination: At loop termination, the statement can be used to derive a property that helps to show the algorithm correct.

How do we relate this to the SPECIFIC problem we have?
That’s YOUR problem - all anyone can do is show enough examples so that you have (hopefully) enough templates to figure out your own.

Analysis...

1. Initialization: since $j = 2$, the supposedly sorted part of the array is $A[1..j-1] = A[1..1]$: any array of ONE element is sorted (obvious, right?).
2. Maintenance: at the beginning of each iteration we have a sorted array $A[1..j-1]$, and an element $A[j]$ that is to be inserted in the correct position, possibly moving some elements in $A[1..j-1]$. We have to convince ourselves that this DOES the job we want: $A[1..j]$ IS sorted before the next iteration. Does it? Keep challenging it until you are completely convinced it works.

Analysis...Add things up, to get $T(n)$, the total time.

- Look at the best case (array already sorted in the order you wanted it): the while loop test gets done only ONCE for each iteration of the for loop. Thus each iteration of the for loop takes a constant amount of time (add up all the constants with their repetition). Total time is a linear function of $n$.
- Look at the worst case (array sorted in inverse order): the while loop tests are carried out the maximum possible number of times, with its body executed just once less. Total time is a quadratic function of $n$. 
Algorithms - Ch2 - Sorting

Look at the average case - if you can figure out what it is... The average case is, probably, the most important metric - especially if you can prove that the probability your data will ever be presented to you in "worst case format" is small... After all, why should you care much if the worst case is only remotely likely, and the behavior of your algorithm on all "real" cases is very good??? Unfortunately, like all good questions, this is very hard to answer...

In this case, we can do it, and we conclude that the (now) expected time T(n) is still given by a quadratic function of n.

Orders of growth and their notation.

O, o, Ω, ω, and Θ.

Divide and Conquer

The previous algorithm uses a method that could be called "incremental" since it solves a problem "one step at a time": given a problem with an input of size n, we solve it for size 1, use that solution to provide a solutions for size 2, and so on...

A second method involves splitting the input of size n into two sets, solving the problem, independently, for the two sets, and then gluing the two solved problems together in such a way that we solve the original problem. This method is called "divide and conquer" - for obvious reasons...

Note: when is divide and conquer the same as the incremental method?
Algorithms - Ch2 - Sorting

Divide and Conquer: MergeSort

This is the "classic example" of a divide and conquer solution to the sorting problem. Start with an array A of size n - the formal parameters below expect actual values between 1 and n. The initial call will be MERGE-SORT(A, 1, n):

MERGE-SORT(A, p, r)
if p < r
then q ← \((p + r)/2\)
▷ Check for base case
MERGE-SORT(A, p, q)
▷ Divide
MERGE-SORT(A, q + 1, r)
▷ Conquer
MERGE(A, p, q, r)
▷ Combine

What's a loop invariant for this algorithm?
Here are the two parts of this algorithm:

1. The initial call to Merge-Sort: this involves a check for size and, if passed, recursive calls on the two halves, followed by a call to Merge.
2. The call to Merge, which involves a loop.

So the whole thing requires a bit more than just a "loop invariant".

- "Recursive" Induction - assuming the Merge is correct - will allow us to conclude the full Merge-Sort is correct
1. Loop invariant (essentially a "linear induction") to prove that Merge is correct.
Algorithms - Ch2 - Sorting

"Recursive" induction

1. Base Cases:
   1. The array is empty: clearly sorted and the body of Merge-Sort - after the test - is a NoOp.
   2. The array has one element: n=1, the array is clearly sorted and the body of Merge-Sort - after the test - is a NoOp.
   3. The array has two elements: n = 2, q = 1, and the body consists of recursive calls on subarrays of size 1, sorted by Case 2 above. We assume Merge correct, so the result is correct

2. Inductive Case:
   1. The array has \( n \geq 3 \) elements. The split results into two subarrays each of size \( < n \). The calls to Merge-Sort result in sorted subarrays by the induction hypothesis; the final result is correct by the correctness of Merge.

Loop Invariant for Merge

What must be true at the beginning of each iteration of the loop in Merge?

\[
\text{MERGE}(A, p, q, r)
\]

\[
n_1 \leftarrow q - p + 1
\]

\[
n_2 \leftarrow r - q
\]

create arrays \( L[1..n_1 + 1] \) and \( R[1..n_2 + 1] \)

for \( i \leftarrow 1 \) to \( n_1 \)
   do \( L[i] \leftarrow A[p + i - 1] \)

for \( j \leftarrow 1 \) to \( n_2 \)
   do \( R[j] \leftarrow A[q + j] \)

\( L[n_1 + 1] \leftarrow \infty \)

\( R[n_2 + 1] \leftarrow \infty \)

\( i \leftarrow 1 \)

\( j \leftarrow 1 \)

5. At the next pass of the loop (that increments \( i \) or \( j \)), \( A[p] \) will contain a smallest element of \( L \) and \( R \), and the new \( L[i] \) and \( L[j] \) are the smallest elements of the subarrays that have NOT yet been recopied into \( A \).

6. Repeat until the loop terminates...
Loop Invariant for Merge: Three Phases

1. **Initialization.** Before the first pass through the loop, we have $k = p$ -- the subarray $A[p..k-1]$ is empty. This empty subarray contains the smallest elements of $L$ and $R$, and, since $i = j = 1$, $L[i]$ and $R[j]$ are the smallest elements of their arrays not yet copied back into $A$.

2. **Maintenance.** Each iteration maintains the loop invariant.
   1. Suppose $L[i] \leq R[j]$. Then $L[i]$ is the smallest element not yet copied into $A$. $A[p..k-1]$ contains (by induction) the $k-p$ smallest elements, so the copying (then branch of the conditional) will ensure that $A[p..k]$ will contain the $k-p+1$ smallest elements. Incrementing $k$ and $i$ re-establishes the loop invariant for the next iteration.
   2. Suppose $L[i] > R[j]$. The else branch of the conditional will copy $R[j]$ and increment $k$ and $j$, maintaining the loop invariant.

3. **Termination.** $k = r+1$. By the loop invariant, the array $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n+1]$ and $R[1..n+1]$, in sorted order. The arrays $L$ and $R$ together contain $n_{1} + n_{2} + 2 = r - p + 3$ elements. All but the two largest have been copied back into $A$, and these two largest elements are sentinels, not in $A$ to begin with.

Analyzing MergeSort: Divide and Conquer

Recursive calls can often be analyzed via recurrence equations. Such equations describe the running time on a problem of size $n$ in terms of the running times on smaller problems.

- Assume that we start with a set of $n$ elements,
- that we break this set of $n$ elements into a subsets, each of $n/b$ items. In many cases $a = b$, but we don’t need it (and algorithms that manipulate pictures need to “bleed over” so that each subpicture has at least two extra rows and 2 extra columns),
- and that we keep breaking the subsets in the same ratios until we reach a size at which we solve the problem directly.

Analyzing MergeSort: Divide and Conquer

We start the analysis from the last bullet: if $T(n)$ denotes the time to run the algorithm on $n$ elements, we must have:

- $T(n) = \Theta(1)$ if $n \leq c$ (for some appropriate “small” size $c$).

Now we look at the larger sets (the two prior bullets):

- $T(n) = a \cdot T(n/b) + D(n) + C(n)$

Where $D(n)$ is the cost of carrying out the divide operation, and $C(n)$ is the cost of the Combine one. Hopefully those costs will be $\Theta(n)$ or less…
What is the cost of \textit{Divide}\? That depends:\n\begin{itemize}\n\item dividing an \textit{array} involves finding the midpoint which can be done in time independent of the size of the array; assuming the size of the array is known and is small enough to fit in the hardware. \(\Theta(1)\).\n\item or can be done in time \(\Theta(n)\), if we use \textit{linked lists}.
\end{itemize}

What is the cost of \textit{Combine}\?\n\begin{itemize}\n\item whether we use lists or arrays, we must compare the elements of the two subsets. At each comparison, we place one element. Total cost for \(n\) elements: \(\Theta(n)\).
\end{itemize}

The recursion relation becomes:\n\begin{equation}
T(n) = \Theta(1) = c \text{ for } n = 1;
\end{equation}
Dividing and Combining:\n\begin{equation}
T(n) = 2 T(n/2) + D(n) + C(n) = 2 T(n/2) + \Theta(n) \text{ if } n > 1.
\end{equation}

Solution method: Figure 2.5 – p. 38.