San Skulrattanakulchai
Gustavus Adolphus College Sep 9, 2014

Insertion Sort

CLRS: Ch 2.1–2.2

Insertion Sort

The array sorting problem is as follows:

Given n numbers $A[1..n]$, rearrange them so that $A[1] \le A[2] \le \cdots \le A[n-1] \le A[n]$.

Insertion sort is a simple algorithm, and works well for very short files.

Iterative Insertion Sort

High-level Algorithm

We repeatedly insert the "next unsorted number" $A[j]$ into its correct position in the "sorted subarray" $A[1..j-1]$.

for $j \leftarrow 2$ to n **do**
  insert $A[j]$ in its correct position in $A[1..j-1]$;

Example Given input array of numbers 12, 8, 1, 10, 4, 7, 9, 2, the array values at the beginning of each successive iteration are

12	**8**	1	10	4	7	9	2
8	12	**1**	10	4	7	9	2
1	8	12	**10**	4	7	9	2
...							
1	4	7	8	9	10	12	**2**
1	2	4	7	8	9	10	12

Low-level Algorithm

$A[0] \leftarrow -\infty$
for $j \leftarrow 2$ to n **do**
{
    $oldAj \leftarrow A[j]$
    $i \leftarrow j - 1$
    while $A[i] > oldAj$ **do**
    {
        $A[i+1] \leftarrow A[i]$
        $i \leftarrow i - 1$
    }
    $A[i+1] \leftarrow oldAj$
}

Note
This algorithm illustrates how a **sentinel** helps simplify the code.

Correctness
It suffices to show that

(i) When the **for** loop terminates, A contains all and only the original elements in sorted order.

(ii) the **for** loop does terminate.

To prove (i), we come up with this claim.
Loop Invariant: At the start of the iteration of the **for** loop, the subarray $A[0..j-1]$ consists of the elements that were originally in $A[0..j-1]$ but in sorted order.

We prove the loop invariant by induction.
Base Case. $A[0..1]$ is sorted before the for loop starts, i.e., the invariant holds when j is 2.
Induction Step. Assume the invariant holds at a particular value of j, it will continue to holds when j is incremented by 1.

Running Time Analysis

high-level algorithm

There are $O(n)$ iterations of the for loop.
Each iteration takes time $O(n)$ in any "reasonable" implementation since we insert $A[j]$ by comparing it to at most every other member.
So the total time is $O(n) \times O(n) = O(n^2)$.

low-level algorithm

We will show that the worst-case running time of insertion sort is $O(n^2)$. We do this by labeling each statement with the amount of time it takes to execute **o**nce. To do this, work outwards from the innermost blocks.

$O(1)$ $A[0] \leftarrow -\infty$
$O(n^2)$ **for** $j \leftarrow 2$ to n **do**
  $O(n)$ {
            $O(1)$ $oldAj \leftarrow A[j]$
            $O(1)$ $i \leftarrow j - 1$
            $O(n)$ **while** $A[i] > oldAj$ **do**
                $O(1)$ {
                    $O(1)$ $A[i+1] \leftarrow A[i]$
                    $O(1)$ $i \leftarrow i - 1$
                        }
            $O(1)$ $A[i+1] \leftarrow oldAj$
            }

Instead of labeling each statement by the amount of time it takes to execute once, we can also label it by the amount of time it takes to execute **throughout the algorithm**.

Recursive Insertion Sort

The algorithm consists of the procedure INSERTION-SORT that calls procedure INSERT. Assume A is a global variable.

```
// Precondition: Array A contains integers.
// Postcondition: Array A contains the original numbers in sorted order.
INSERTION-SORT(A) {
    n ← A.length
    if n > 1 then {
        INSERTION-SORT(A[1..n − 1])
        INSERT(n)
    }
}

/*
    Precondition: Subarray A[1..j − 1] is sorted.
    Postcondition: Subarray A[1..j] contains original elements in A[1..j] in sorted order.
*/
INSERT(j) {
    if j = 1 then return
    else if A[j − 1] < A[j] then return
    else {
        swap A[j − 1] & A[j]
        INSERT(j − 1)
    }
}
```

Correctness

First we show INSERT(j) works correctly, then show INSERTION-SORT(A) works correctly. For each task we show that *if the precondition holds before the procedure starts, then the postcondition holds when the procedure finishes.* For correctness of INSERT(j) we prove by induction on j. For correctness of INSERTION-SORT(A) we prove by induction on length of A.

Running Time Analysis

Let $I(j)$ be the worst-case running time of INSERT(j). The recurrence for $I(j)$ is

$$I(j) = \begin{cases} O(1) & \text{if } j = 1 \\ O(1) + I(j-1) & \text{if } j > 1. \end{cases}$$

Iterating the recurrence gives $I(j) = O(j)$.

Let $T(n)$ be the worst-case running time of INSERTION-SORT(n). The recurrence for $T(n)$ is

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ T(n-1) + I(n) & \text{if } n > 1. \end{cases}$$

Pluggging in $I(n) = O(n)$ and iterating the recurrence gives $T(n) = O(n^2)$.

Notes

1. The items to be sorted do not have to be numbers. They can be characters, strings, etc. In general, they can come from any totally-ordered universe.

2. We can show that the (worst-case) running time of insertion sort is $\Theta(n^2)$ by proving the $\Omega(\cdot)$ estimate separately from proving the $O(\cdot)$ estimate. (Use arrays that are already sorted in decreasing order.) When we can prove that the big-O estimate of an algorithm agrees with its big-Theta estimate, we sometimes say that we get a "tight estimate."

3. The exact time bound for insertion sort is $\Theta(n + I)$ where I is the number of "inversions" in the input. This explains why insertion sort is fast on files that are "almost-sorted."

Exercise

Explain why it's a bad idea to implement the "insert $A[j]$ in its correct position in $A[1..j-1]$" by scanning the subarray $A[1..j-1]$ from the left instead of from the right as we did in this handout.