CMSC 351: InsertSort

Justin Wyss-Gallifent

February 19, 2021

1 What it Does

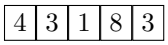
Sorts a list of elements on which there is a total order. Think of integers or real numbers.

2 How it Works:

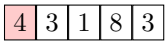
We pass through the list from left to right. If we encounter an entry which is smaller than some of its predecessors then we need to move it as far left as is appropriate. We shift as many elements to the right as needed to make room for it and then insert it (hence the name) in the proper position.

3 Visualization

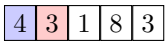
Consider the following list:



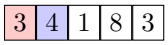
We first look at `A[0]==4`. There is nothing larger to the left of it, in fact there is nothing to the left of it at all:



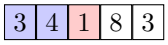
We do nothing. Officially the pseudocode will actually shift nothing and then assign `A[0]=4`, which is a bit wasteful but there we go.
We then look at `A[1]==3` and note that there is one larger element to the left:



We shift it to the right and insert the 3 where it belongs:

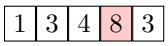


We then look at `A[2]==1` and note that there are two larger elements to the left:



We shift those to the right and insert the 1 where it belongs:

We next look at `A[3]==8` and note that there are no larger elements to the left:



We do nothing. Again officially the pseudocode will actually shift nothing and then assign `A[3]=8`, which is a bit wasteful but there we go.
We then look at `A[4]==3` and note that there are two larger elements to the left:

We shift those to the right and insert the 3 where it belongs:

Then we are done.

4 Outline of Pseudocode

To help see how the pseudocode work, here is the general outline. Pseudo-pseudocode, if you wish!

```
for i = 0 to n-1
    key = A[i], the value to potentially be moved left.
    j = i - 1, the first index to the left.
        shift A[j] to the right while key < A[j] and j >= 0
        j = j - 1
        this while loop is shifting things right
        to make room for key in the right place
    once this ends either key >= A[j] and should be left alone,
    meaning key should go into index j+1,
    or j = -1, meaning key should go into index 0 = j+1
    A[j+1] = key, so key is now in the right place
end
once this ends, everything is right
```

5 Pseudocode with Time Assignments

Here is the pseudocode with time assignments. Here I've gone old-school and tagged the `while` loop condition check. We'll see why.

```
\\PRE: A is a list of length n.
for i = 0 to n-1                          n iterations
    key = A[i]                            } c_1
    j = i-1                               }
    while j >= 0 and key < A[j]           ??? iterations at c_2 each
        A[j+1] = A[j]                     } c_3
        j = j - 1                         }
    end
    A[j+1] = key                          c_4
end
\\POST: A is sorted.
```

Here `key` holds the value at index `i` needs to be stored as we check to the left of it and potentially shift any larger left element to the right. Once we have moved those to the right we insert `key` into its appropriate position.
Notice that the `while` loop checks not just whether the element to the left is larger but also whether that searching process falls off the front of the list. It stops in either case, of course. In the case where it falls off the front of the list we have `j==-1` and we need to insert `key` at index `0==j+1`. In the case where it encounters an element at index `j` which is smaller or equal than `key` and which does not need to be moved, it must insert `key` to the right of it at index `j+1`. This is why the line after the `while` loop is `A[j+1] = key`.

The time complexity analysis for InsertSort is very different than BubbleSort and SelectionSort. This is due to the `while` loop which results in an unknown number of iterations. For a particular input of length n this loop could iterate never or perhaps a number of times linearly dependent on n. We thus have to separate cases.

1. **Best-Case:**

   Observe that the `while` loop executes while we are in the process of shifting larger elements right to make room for our smaller `key`. If the list is already sorted then the condition of the `while` loop fails (but still takes c_2 time for the conditional check) for `i=0,...,n-1`.

   The total time requirement is:

$$T(n) = n(c_1 + c_2 + c_4) = \Theta(n)$$

2. **Worst Case:**

   In a worst-case scenario:

   - For $i = 0$ the `while` statement checks+fails (because $j = -1$) and hence the time required is:

   $$c_1 + \underbrace{c_2}_{C+F} + c_4$$

   - For $i = 1$ we have to move `A[1]` all the way to the left, meaning the `while` loop checks+passes once and checks+fails once and hence the time required is:

   $$c_1 + \underbrace{1(c_2 + c_3)}_{C+P} + \underbrace{c_2}_{C+F} + c_4$$

   - For $i = 2$ we have to move `A[2]` all the way left, meaning the `while` loop checks+passes twice and checks+fails once and hence the time required is:

   $$c_1 + \underbrace{2(c_2 + c_3)}_{C+P} + \underbrace{c_2}_{C+F} + c_4$$

   - ...and so on until...
   - For $i = n-1$ we have to move `A[n-1]` all the way to the left, meaning the `while` loop checks+passes $n-1$ times and checks+fails once and hence the time required is:

   $$c_1 + \underbrace{(n-1)(c_2 + c_3)}_{C+P} + \underbrace{c_2}_{C+F} + c_4$$

   In general for $0 \leq i \leq n-1$ we require time:

   $$c_1 + \underbrace{i(c_2 + c_3)}_{C+P} + \underbrace{c_2}_{C+F} + c_4$$

   If follows that the total time requirement is:

$$T(n) = \sum_{i=0}^{n-1} [c_1 + i(c_2 + c_3) + c_2 + c_4] = ... = \Theta(n^2)$$

3. **Average Case:**

   First we'll observe that the running time can be written in terms of the number of *inversions* which exist in the original list. An *inversion* is a situation whereby there exist `i > j` with `A[i] < A[j]`. In such a situation when the `for` loop encounters `A[i]` it will require a number of shifts equal to the inversion count involving `A[i]`. This is because each shift corrects an inversion. Thus the total number of shifts will equal the total number of inversions.

   For example in our original list:

4	3	1	8	3

   In this list there are 5 inversions: $(4,3)$, $(4,1)$, $(4,3)$, $(3,1)$, $(8,3)$

   We also can look back and see that 5 shifts were required.

   Now then, let's consider the time requirements again:

   - For each of `i=0,...,n-1` we have $c_1 + c_4$ before and after the `while` loop.
   - For each inversion the `while` loop criteria will pass exactly once, requiring time $c_2 + c_3$.
   - For each of `i=0,...,n-1` the `while` loop criteria will fail exactly once, either after shifts have occurred or at the start if no shifts were necessary, requiring time c_2.

   Thus if there are I inversions then the total time requirement will be:

$$T(n, I) = \underbrace{n(c_1 + c_4)}_{\text{Pre+Post-While}} + \underbrace{I(c_2 + c_3)}_{\text{C+P}} + \underbrace{n(c_2)}_{\text{C+F}}$$

   For the sake of simplicity let's assume no duplicates in what follows.

   In an arbitrary list if we look at all possible $0 \leq i < j \leq n-1$, noting that there are $C(n,2)$ such pairs, half of the time we will have `A[i]<A[j]` and half the time we will have `A[j]<A[i]`. Thus on average we will have $I = I(n) = C(n,2)/2$ inversions and on average the time requirement will be:

$$T\left(n, I(n)\right) = T\left(n, \frac{C(n,2)}{2}\right) = ... = \Theta(n^2)$$

6 Auxiliary Space

The auxiliary space is $\mathcal{O}(1)$.

7 Stability

InsertSort is stable. This is because InsertSort shifts elements to the left while they are smaller than those elements. Consequently an element will not be shifted to the left of an equal element and so the order of equal elements will be preserved.

8 In-Place

InsertSort is in-place.

9 Notes

After k iterations the first k elements are sorted but not necessarily correctly placed. If InsertSort were running on a very long list and it was forced to stop early this means that some amount of the beginning of the list would be sorted but in the wrong position and the end would not be sorted at all. This is different from both BubbleSort and SeledtionSort.

Example 9.1. Consider the list:

```
[7,8,9,4,5,6,1,2,3]
```

The first six iterations operate on the elements 7, 8, 9, 4, 5, and 6. The first three don't move but they still count as iterations as far as the psedocode is concerned.
The result is then:

```
[4,5,6,7,8,9,1,2,3]
```

Notice that the first six elements are sorted relative to one another but they are certainly not in their correct overall locations. It will take three more iterations to move the `1,2,3` for this to be the case.

10 Thoughts, Problems, Ideas

1. Show the step-by-step operation of InsertSort on the list $\{5, 10, 3, 0, -6, 9\}$.

2. Assuming all constant times equal 1, fill in the details of the worst-case time calculation:

$$T(n) = \sum_{i=0}^{n-1} c_1 + i(c_2 + c_3) + c_2 + c_4 = ... = \Theta(n^2)$$

3. Assuming all constant times equal 1, fill in the details of the average-case time calculation:

$$T\left(n, I(n)\right) = T\left(n, \frac{C(n,2)}{2}\right) = ... = \Theta(n^2)$$

4. Modify InsertSort so the list is sorted in decreasing order.

5. The basic premise of InsertSort is that for each index `i` we effectively shift everything to the left of this index which smaller than the value at this index to the right by 1 and then insert (hence the name) the value in the empty left location. Suppose you had a function `rshift(A,a,b)` which shifted the subarray `A[a,...,b]` to the right by 1. Use this to develop a pseudocode version of InsertSort. For each `i` this version should scan left until it finds where to put `A[i]`, then shift everything necessary to the right, and then insert. If `rshift` operates with constant time complexity what is the worst-case $\mathcal{O}$ time complexity of your new pseudocode?

6. Another way to think of InsertSort is that for each index `i` we want to know where to the left to insert it. Note that for each `i` the sublist to the left is always sorted as the algorithm progresses. Call this target index `k`. The series of swaps effectively shifts `A[k,...,i-1]` to the right and inserts the original `A[i]` to index `k`. Suppose you had a function `locateindex(A,i)` which would find the correct index in `A[0,...,i-1]` to insert `A[i]`. Use this to develop a pseudocode version of InsertSort. You'll still need to do the swapping manually.

7. Suppose you had both `rshift` and `locateindex` from the previous two problems. Now write the new pseudocode. If the time complexity of `locateindex` is $L(n)$ and the time complexity of `rshift` is $R(n)$ what would need to be true of these in order for the resulting search to be $\mathcal{O}$-better than InsertSort?

8. Arguably the pseudocode is a bit confusing because the search index `j` is actually always one less than the target position. Tweak the pseudocode to change this.

9. We saw that in the average case the pseudocode has time requirement:

$$T(n, I) = n(c_1 + c_4) + I(c_2 + c_3) + n(c_2)$$

where I is the number of inversions in the input list. The reason that the average case turns out to be $\mathcal{O}(n^2)$ is that on average the number of inversions satisfies $I(n) = \mathcal{O}(n^2)$. We'll say a list of length n is *logarithmically sorted* (I made that term up) if the number of inversions is at most $\lceil (\lg n/n)C(n, 2) \rceil$. For example a logarithmically sorted list of length 10 will have at most $\lceil (\lg 10/10)C(10, 2) \rceil = 12$ inversions. Calculate the $\mathcal{O}$ average time complexity of the pseudocode if it only encounters such lists.

11 Python Test

Code:

```
import random
A = []
for i in range(0,10):
    A.append(random.randint(0,100))
n = len(A)
print(A)
for i in range(0,n):
    key = A[i]
    print('Checking ' + str(key) + ' at index ' + str(i))
    j = i-1
    howmanytoleft = 0
    while j>=0 and key<A[j]:
        howmanytoleft = howmanytoleft + 1
        A[j+1] = A[j]
        j = j - 1
    if howmanytoleft == 0:
        print('There are 0 larger elements to the left.
            Leave alone.')
    else:
        A[j+1] = key
        print('There are ' + str(howmanytoleft) + ' larger
            elements to the left.')
        print('Shift and insert: ' + str(A))
print(A)
```

Output:

```
[31, 24, 77, 79, 53, 20, 20, 75, 46, 0]
Checking 31 at index 0
There are 0 larger elements to the left. Leave alone.
Checking 24 at index 1
There are 1 larger elements to the left.
Shift and insert: [24, 31, 77, 79, 53, 20, 20, 75, 46, 0]
Checking 77 at index 2
There are 0 larger elements to the left. Leave alone.
Checking 79 at index 3
There are 0 larger elements to the left. Leave alone.
Checking 53 at index 4
There are 2 larger elements to the left.
Shift and insert: [24, 31, 53, 77, 79, 20, 20, 75, 46, 0]
Checking 20 at index 5
There are 5 larger elements to the left.
Shift and insert: [20, 24, 31, 53, 77, 79, 20, 75, 46, 0]
Checking 20 at index 6
There are 5 larger elements to the left.
Shift and insert: [20, 20, 24, 31, 53, 77, 79, 75, 46, 0]
Checking 75 at index 7
There are 2 larger elements to the left.
Shift and insert: [20, 20, 24, 31, 53, 75, 77, 79, 46, 0]
Checking 46 at index 8
There are 4 larger elements to the left.
Shift and insert: [20, 20, 24, 31, 46, 53, 75, 77, 79, 0]
Checking 0 at index 9
There are 9 larger elements to the left.
Shift and insert: [0, 20, 20, 24, 31, 46, 53, 75, 77, 79]
[0, 20, 20, 24, 31, 46, 53, 75, 77, 79]
```