CMSC 351: BubbleSort (Basic)

Justin Wyss-Gallifent

February 3, 2022

1 What it Does

Sorts a list of elements such as integers or real numbers.

2 How it Works

We pass through the list from left to right swapping elements which are out of order. If we pass through once, the final entry is in the right place because the largest element will have been swapped repeatedly until it is at the end. Thus the second pass through can stop before the final entry. If we pass through twice, the final two entries are in the right place and so the third pass through can stop before the second-to-last entry. We continue onwards. Thus we pass through $n-1$ times total since once the final $n-1$ entries are in order, all are.

3 Visualization

Consider the following list:

8	4	3	4	1

Here is the result of the first pass through. We indicate in red pairs the swaps.

8	4	3	4	1
4	8	3	4	1
4	3	8	4	1
4	3	4	8	1
4	3	4	1	8

Notice after the first pass through the one rightmost element is correct.
Second pass through:

4	3	4	1	8
3	4	4	1	8
3	4	1	4	8

Notice after the second pass through the two rightmost elements are correct.
Third pass through:

3	4	1	4	8
3	1	4	4	8

Notice after the third pass through the three rightmost elements are in order.
The fourth pass through:

3	1	4	4	8
1	3	4	4	8

Notice after the fourth pass through the four rightmost elements are in order but since there are only five elements, they are all in order.

4 Pseudocode and Time Complexity

Here is the pseudocode with time assignments:

```
PRE: A is a list of length n.
for i = 0 to n-2                    n times
    for j = 0 to n-i-2              n - i - 2 - 0 + 1 = n - i - 1 times
        if A[j] > A[j+1]            }
            swap A[j] and A[j+1]    } c
        end                         }
    end
end
POST: A is sorted.
```

This is a terrible implementation. The time required is always the same in the best-, worst-, and average-cases:

$$\begin{aligned}
T(n) &= \sum_{i=0}^{n-2} \sum_{j=0}^{n-i-2} c \\
&= \sum_{i=0}^{n-2} (n-i-2+1)c \\
&= \sum_{i=0}^{n-2} c(n-1) - c\sum_{i=0}^{n-2} i \\
&= (n-2+1)c(n-1) - c\left(\frac{(n-2)(n-1)}{2}\right) \\
&= c(n-1)^2 - \frac{1}{2}c(n-2)(n-1) \\
&= \frac{1}{2}cn^2 - \frac{1}{2}cn \\
&= \Theta(n^2)
\end{aligned}$$

A quick note here that if we were interested in every little detail of time then we would need to label separate time constants for both the conditional check and for the conditional body and then best-, worst-, and average-cases would have different total times depending upon the frequency with which the body of the conditional executed.
However in all cases the conditional is still taking constant time, even if those constant times are different, and hence the time complexity does not change.

5 Auxiliary Space

Our BubbleSort (Basic) pseudocode uses $\mathcal{O}(1)$ auxiliary space - two indices and possibly, depending on the implementation, a swap variable This is of course

pretty much ideal.

6 Stability

BubbleSort (Basic) is stable. This means that the order of identical entries is preserved. In the opening example if we think of the two 4s as different, perhaps the one on the left is red and one on the right is blue, then the red one is still on the left at the end. The reason for this is that equal elements are never swapped, meaning that the order of equal elements is always preserved.

7 In-Place

BubbleSort (Basic) is in-place. This means that the list is sorted by moving elements within the list, rather than creating a new list.

8 Notes

After k iterations the last k elements are correctly placed and sorted. If BubbleSort were running on a very long list and it was forced to stop early this means that some amount of the end of the list would be sorted but the beginning would not. This is the opposite of SelectionSort, as we'll see.

9 Bubble Sort (Variation)

We can improve the code somewhat because if any give pass through causes no swaps then the list is in order and we can exit. Here is the adjusted pseudocode with time assignments:

```
PRE: A is a list of length n.
stillgoing = True                          c_1
for i = 0 to n-1                           ??? times because of break
    didaswap = False                       c_3
    for j = 0 to n-i-2                     n - i - 1 times
        if A[j] > A[j+1]                   }
            swap A[i] and A[i+1]           } c_2
            didaswap = True                }
        end                                }
    end
    if didaswap == False                   }
        break                              } c_4
    end                                    }
end
POST: A is sorted.
```

In this case we have:

- Worst-Case: If the list is in reverse order then each `i` iteration results in $n-i-1$ comparisons and swaps. To see this observe that if we start with $[5,4,3,2,1]$ then the `i=0` pass has $n-i-1 = 5-0-1 = 4$ comparisons and swaps and results in $[4,3,2,1,5]$. The `i=1` pass has $n-i-1 = 5-1-1 = 3$ comparisons and swaps and results in $[3,2,1,4,5]$, and so on. We therefore use all of `i=0,...,n-1` iterations (no `break`) and the time is then:

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

- Best-Case: If the list is already sorted then we have one pass through for `i=0` with $n-i-1$ iterations and since no swaps occur we exit. The time is then:

$$T(n) = c_1 + c_3 + (n-i-1)c_2 + c_4 = ... = \Theta(n)$$

- Average-Case: We need to clarify what an "average" list looks like. It turns out that in an average list we end up swapping about half the time. (his takes some combinatorics) and the result is still $\Theta(n^2)$.

This variation also uses $\mathcal{O}(1)$ auxiliary space, is stable, and is in-place.

10 Closing Notes

While BubbleSort is fairly useless in practice since it is slow and inefficient it is nonetheless useful for basic algorithmic understanding and some of the ideas therein can be used in other algorithms.

11 Thoughts, Problems, Ideas

1. Fill in the calculation:

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-i-2} c_1 = ... = \Theta(n^2)$$

   At the end, provide a rigorous proof from the definition of Θ that the result is $\Theta(n^2)$.

2. Fill in the calculation:

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

   At the end, provide a rigorous proof from the definition of Θ that the result is $\Theta(n^2)$.

3. Suppose BubbleSort (Basic) is used to sort the list `[34,2,17,5,10]`. Show the list after each swap and count the number of swaps.

4. Modify the BubbleSort (Basic) pseudocode to produce an algorithm which moves all `0`s (if any) to the right and otherwise preserves order.

5. BubbleSort (Basic) is stable. Explain in your own words what that means. What single-character change in the pseudocode would make it not stable?

6. How many comparisons occur in BubbleSort (Basic)?

7. In BubbleSort (Basic) how many swaps occur in a worst-case scenario? How about a best-case scenario?

8. In BubbleSort (Basic) think about how many swaps might occur in an average-case scenario. What if the list was mostly sorted? This is a qualitative rather than a quantitative question.

9. Each integer greater than 2 has a unique prime factorization. For example $243936 = 2^5 \cdot 3^2 \cdot 7^1 \cdot 11^2$. Suppose you want to write an algorithm which sorts the powers using BubbleSort and modifies the number accordingly, so we'd have, for example:

$$243936 = 2^5 \cdot 3^2 \cdot 7^1 \cdot 11^2 \Longrightarrow 2^1 \cdot 3^2 \cdot 7^2 \cdot 11^5 = 142046982$$

   You wish to use BubbleSort on the exponents but you are only given the integer n and you cannot construct or use a mutable list. You do have access to the commands:

   - `ppow(n,p)` which gives the exponent corresponding to the power of `p` in `n`, so `ppow(243936,3)` returns `2`.

- `parr(n)` which returns the list of primes whcih occur in the prime factorization of `n`, so `parr(243936)` returns `[2,3,7,11]`. You can index it with `parr(243935)[0]` yielding `2`, for example.

Write the associated pseudocode.

10. If the algorithm in the previous problem is used to define a function `primesort(n)` (so for example `primesort(243936)=142046982`) explain why the output is never smaller than the input. For which `n` is it identical?

11. Modify BubbleSort so that after k iterations the first k elements are correct.

12. Modify BubbleSort so that after $2k$ iterations the first k and last k elements are correct. You can assume that the list has an even number of elements. Hint: Alternate!

13. Here is a re-labeling of BubbleSort (Basic) time requirements in which the time of the conditional and its body assignment have been separated:

```
PRE: A is a list of length n.
for i = 0 to n-1                    n times
    for j = 0 to n-i-2              n − i − 2 − 0 + 1 = n − i − 1 times
        if A[j] > A[j+1]            c1
            swap A[j] and A[j+1]    c2
        end
    end
end
POST: A is sorted.
```

    (a) If $c_1 = 3$ and $c_2 = 10$, both in seconds, how long will it take to sort the list `[4,5,1,2,7,7]`?

    (b) Suppose the inputs to this implementation are such that on average the inequality is true with probability $0 \leq p \leq 1$. Calculate the resulting average time requirements exactly. When you have your result suppose n were some fixed value. Explain how the time required changes as p does. Is it linear or something else?

14. Provide a rigorous mathematical proof that the first iteration through BubbleSort places the largest element in the final position.

15. Suppose you hate `break` statements. Rewrite the BubbleSort (Variation) using a `while` loop and no `break`.

12 Python Test (Basic Version) with Output

Code:

```
import random
A = []
for i in range(0,10):
    A.append(random.randint(0,100))
n = len(A)

print('Start: '+str(A))
for i in range(0,n):
    for j in range(0,n-i-1):
        if A[j] > A[j+1]:
            temp = A[j]
            A[j] = A[j+1]
            A[j+1] = temp
    print('Iterate: '+str(A))
print(A)
```

Output:

```
Start: [74, 85, 2, 13, 82, 36, 29, 66, 31, 12]
Iterate: [74, 2, 13, 82, 36, 29, 66, 31, 12, 85]
Iterate: [2, 13, 74, 36, 29, 66, 31, 12, 82, 85]
Iterate: [2, 13, 36, 29, 66, 31, 12, 74, 82, 85]
Iterate: [2, 13, 29, 36, 31, 12, 66, 74, 82, 85]
Iterate: [2, 13, 29, 31, 12, 36, 66, 74, 82, 85]
Iterate: [2, 13, 29, 12, 31, 36, 66, 74, 82, 85]
Iterate: [2, 13, 12, 29, 31, 36, 66, 74, 82, 85]
Iterate: [2, 12, 13, 29, 31, 36, 66, 74, 82, 85]
Iterate: [2, 12, 13, 29, 31, 36, 66, 74, 82, 85]
Iterate: [2, 12, 13, 29, 31, 36, 66, 74, 82, 85]
[2, 12, 13, 29, 31, 36, 66, 74, 82, 85]
```