Classical $O(N^2)$ algorithms for sorting a list

One common problem that comes up in computer science is to sort a list of $N$ objects. Let’s say we want the elements to be in increasing order. Here we are assuming that objects are comparable, in the sense that for any two objects $A$ and $B$ that we are considering, either $A < B$ or $A > B$. It could also be that $A == B$, if there are multiple copies of the same element in the list. For example, the elements in the list might be numbers, or they might be strings which can be ordered alphabetically. Today we will look at a few simple algorithms. Later in the course, we’ll look at more complicated algorithms which are much faster when the list is large.

Bubble Sort

The first algorithm is the simplest to describe. You traverse through the list repeatedly, and whenever you find two neighboring elements that are out of order, you swap them. Elements gradually find their way to their correct position in the list, kind of like bubbles rising in a fluid.

```python
for k = 0 to N-1 // a counter, not an index
    for i = 0 to N-2-k
        if list[i] > list[i + 1]{
            swap( list[i], list[i + 1] )
        }
```

[CORRECTION: Oct 15] Note that after the $k=0$ pass, the largest element in the list will be at position $N-1$. Thus in the next pass when $k=1$, the inner loop only needs to go from $k=0$ to $N-2-1$. In general, after pass $k$, the last $k+1$ elements in the list are in their correct position so the inner loop only needs to go to $N-2-k$.

Swapping two variables can be done easily using a temporary variable. It should be something you learned in COMP 202 but I mention it here for completeness. You can’t implement swap with:

```python
swap(x,y){
    x = y
    y = x
}
```

because the $x = y$ operation wipes out the previous $x$ value. Instead, you use:

```python
swap(x,y){
    tmp = x
    x = y
    y = tmp
}
```

Anyhow, back to bubblesort. Think what happens if the smallest element of the list starts out at the end of the list, i.e. in position $N-1$. In the first pass through the inner loop, the element will be moved to position $N-2$. In the second pass, it will be moved to position $N-3$. etc. Thus, it will take $N-1$ passes for the smallest element to arrive at the beginning.

How long does the bubblesort algorithm take? There are two nested loops, so the operations in the inner loop are executed in time proportional to $N^2$. Specifically, there are $(N - 2) + (N - 1)$
3) \[1 + 2 + 1 = \frac{(N-2)(N-1)}{2}\] \[N-2\] \[N-1\] passes through the inner loop. (One can make the code a bit faster in practice by checking to see if there is a swap during the k-th pass of the outer loop, and then stopping if there is no swap. But for simplicity, I will leave out that code here.)

The algorithm is most naturally expressed and implemented using an array, but it could be implemented with a linked list too. Since one iterates through the list each time, one just needs access to the next and prev elements. I will omit the details here, since they are distracting. But if you have any doubt, then you should try and implement this algorithm using the DLinkedList class that I gave you in the linked list Exercises.

**Selection sort**

The second algorithm we examine is called *Selection Sort*. The algorithm repeatedly finds the smallest element and moves it to its correct position in the list. Specifically, for each \(i\), the algorithm finds the minimum element in positions \(i\) to \(N-1\). If this minimum element is at a position different from \(i\), then it swaps this minimum element with the element at position \(i\).

```java
for (i = 0; i < N-1; i++) { // check for yourself that i==N-1
    iMin = i // is unnecessary.
    valueMin = list[i]
    for ( k = i+1; k < N; k++) {
        if (list[k] < valueMin)
            iMin = k; // remember which one it was
            valueMin = list[k] // rather than swapping right away.
    }
    if ( iMin != i )
        swap(list[i], list[iMin]);
}
```

One advantage of this algorithm over bubblesort is that this algorithm does fewer swaps. Indeed, it does at most one swap in each pass in the outer loop. Another advantage is that each loop examines fewer elements than the previous one.

```java
for (i = 0; i < N-1; i++)
:
    for ( k = i+1; k < N; k++)
```

The first \(k\) iteration takes \(N-1\) steps (from 0 to N-2); the second takes \(N-2\) steps; the last iteration takes 1 step, for a total of

\[1 + 2 + 3 + \ldots + N - 1 = \frac{N(N-1)}{2}\]

This is less than half as much as \(N^2\), although it is still \(O(N^2)\).
Insertion Sort

The third algorithm is similar to the second, in that it maintains a list of sorted elements at the front of the list, and then increases the size of the sorted list by one each time. However, the mechanism is different. Rather than considering all the remaining unsorted elements and finding the smallest one, as in the previous algorithm, it takes the next element in the list and finds where it belongs relative to the ones that have already been sorted, and inserts this next element into the proper slot. For this reason, the algorithm is called "insertion sort".

It is most common to describe this algorithm using an array list. Let’s say we have \( N \) objects. The algorithm goes through a loop \( N \) times. In the \( k \)th pass through the loop, the algorithm inserts element that is at index \( k \) into its correct position with respect to the elements up to position \( k - 1 \), which are already in their correct order.

How does the algorithm put the element at index \( k \) into its correct position with respect to elements at indices 0 to \( k - 1 \)? The idea is to search backwards from index \( k \) until it finds the right place for this element. As it searches back, it shifts forward by 1 position any element that is bigger than element \( k \). Once an element is found that less than or equal to \( \text{list}[k] \), we know that \( \text{list}[k] \) must be inserted directly after that element. Here is the algorithm:

```plaintext
for k = 1 to N - 1 { // don’t need to consider k = 0
    elementK = list[k] // copy current element so it doesn’t get erased
    i = k
    while (i > 0) and (list[i - 1] > elementK ){
        list[i] = list[i - 1] // shift up larger elements
        i = i -1
    }
    list[i] = elementK // insert into correct position
} // (has no effect elementK was in correct position)
```

What is the time complexity of insertion sort? As in the first two algorithms, we have two nested for loops. The first goes from \( k = 1 \) to \( N-1 \), and the second goes from \( i = k \) down to 1 in the worst case. So the number of passes through the inner loop is:

\[
1 + 2 + \ldots + N - 1 = \frac{N(N-1)}{2}
\]

which again is \( O(N^2) \).

Note that the inner loop only continues as long as the while condition is met. If \( \text{elementK} \) is already greater than \( \text{list}[k-1] \) then the inner loop ends immediately because element \( k \) is already in the correct position.

What is the best and worst case scenario? In the best case, the array is already sorted from smallest to largest. Then the condition tested in the while loop will be false every time, and so the inner loop will take constant time. Since there are \( N \) passes through the for loop, the time taken is proportional to \( N \) in the best case.

The worst case is that the list is already sorted, but it is sorted in the wrong direction, namely from largest to smallest. In this case, the while loop is executed the maximum number of times, namely the expression above.
Summary

In trying to understand these algorithms, it is best to start at a high level description and work ones way eventually to the code. Don’t start with the code. It may also help to look at various website that give visualizations of different sorting algorithms. For example:

[https://www.toptal.com/developers/sorting-algorithms](https://www.toptal.com/developers/sorting-algorithms)