Lecture Summary

In this lecture, we begin our exploration of sorting. We learn what it means to sort, what are applications of sorting, what stability of sorting algorithms means, and we see our first three sorting algorithms: Bubble Sort, Selection Sort, and Heapsort.

In the next few lectures, we will learn how to sort items. In order to sort items, we need to be able to compare them, i.e., to determine whether some item $x$ is smaller, greater, or equal to some other item $y$. The goal of sorting is to put an array/list/vector of such items into non-decreasing or non-increasing order. Non-decreasing is basically the same as increasing, except that we allow multiple equal numbers. For example, $2 \ 3 \ 3 \ 5 \ 7 \ 9$ is an array of integers in non-decreasing order.

Formally, to say that an array or list is sorted means that $\forall i \in S : a[i+1] \geq a[i]$, where $S = \{0,1,2,...,\text{size} - 2\}$ is the set of all indices in the array. Furthermore, in order to claim that an algorithm sorts an array, not only does the final array need to be sorted, it also has to still contain the same elements as before. Otherwise, we could use the following silly algorithm to obtain a sorted array:

```java
for (int i = 0; i < n; i++)
a[i] = 0;
```

Clearly, this is not what we intended.

1 Applications of Sorting

Sorting is useful for a number of reasons, including the following:

- Understanding sorting algorithms helps to illustrate central algorithm design and analysis concepts which will be covered in more depth in an algorithms class.
- Humans usually prefer sorted data to read.
- A sorted array is much easier to search in, e.g., using Binary Search.
- Sorting makes it much easier to discover patterns or statistics of data items, such as the median or other moments.
- Sorting often helps in comparing lists (or sets), and performing operations like intersection of sets, finding out if two lists contain the same elements, finding if there are duplicates in a list, etc.

As an illustration of the last type of problem, we solved in class the following problem: given two lists of numbers, find the intersection, i.e., the numbers occurring in both lists. Half of the class was asked to solve this problem with unsorted lists, while the other half was using sorted lists. While the experiment did not confirm the algorithmic insight (apparently, the students who got unsorted lists were extremely motivated to be faster), we still talked about the algorithms.

If the lists are sorted, we can simply go through both lists in parallel. Whenever we are looking at the same number in both lists, we can add it to the intersection. When the element we’re looking at in list 1 is larger, then we increment our counter in list 2, and vice versa. The running time is $\Theta(n + m)$ when the lists have sizes $n$ and $m$. 

1
When the lists are unsorted, we can’t really do anything much better than going through one list, and for each element, look for it in the other list, which requires a linear search. Since we need to go through every element in the second list for every element in the first list, the runtime is $\Theta(nm)$.

Thus, sorting really speeds up the goal of computing the intersection. This is true for many other similar problems on one or two lists, where sorting improves the performance from $\Theta(nm)$ to $\Theta(n+m)$. These types of questions are also classic interview questions, since they test both knowledge of sorting, and problem solving skills.

We will learn in the next lectures that there are algorithms sorting an array of $n$ elements in $\Theta(n \log n)$ time. Thus, sorting both lists first is actually faster in our intersection example than trying to compute the intersection on unsorted lists. In general, when an interviewer tells you to solve a problem in time $O(n \log n)$, there is a good chance that his intended solution uses sorting of an array. (Of course, there are other problems that can be nicely solved in time $O(n \log n)$. But among the interview questions for students early in their career, sorting is very prominent among $O(n \log n)$ algorithms.)

## 2 Stability of sorting algorithm

We said above that two properties are absolutely essential for sorting algorithms: (1) The output must be sorted, and (2) It must still contain the same elements. A third property is not essential, but often useful. It is called stability: an algorithm is stable if every two elements that have the same value have the same relative order after sorting as before. As an example, consider the following input: 3(red) 5 1 3(blue): we have two distinct elements, both with the value 3. The following would be the stably sorted output: 1 3(red) 3(blue) 5. The following output is also sorted, but unstably: 1 3(blue) 3(red) 5.

Stability of an algorithm can be very useful when we have multiple sorting criteria, which happens routinely when implementing a Spreadsheet or Database system. For example, suppose that you want to sort all USC students by GPA; if two students have the same GPA, the two students should be ordered alphabetically.

One solution of this problem would be to define the relationship of $a$ and $b$ to be: $a < b$ if ($a.GPA < b.GPA$ || ($a.GPA == b.GPA$ && $a.name < b.name$)). This would work, but doing things this way requires us to define more and more sorting rules whenever we want another tie breaker.

An alternative, perhaps more convenient, solution would be to first sort by name, then sort by GPA. If we are using a stable sorting algorithm, the alphabetical order among students will still be in place, i.e., will be the tie breaker among students with the same GPA.

## 3 Bubble Sort

The first sorting algorithm we came up with in class was Bubble Sort. In Bubble Sort, we compare the first with the second entry and swap them if they are out of order, then the second with the third, swapping if necessary, etc. Once we reach the end of the array, we know that the largest element is there — it has “bubbled up” to the top. Now, we start again with the first element, and so on, terminating one spot earlier. This gets repeated at most $n-1$ times. The code is as follows:

```java
for (int i = 0; i < n - 1; i++) {
    for (int j = 0; j < n - i - 1; j++) {
        if (a[j] > a[j+1]) {
            a.swap(j, j+1);
        }
    }
}
```

This algorithm is stable, since it only swaps two items if the latter one is strictly greater, so equal-valued items will stay in their original order.
To formally prove correctness of an algorithm based on \texttt{for} loops, the typical technique is to use induction on the iterations. The induction hypothesis for this type of proof is also called the \textit{loop invariant}: it captures a property of the variables that is true after each iteration of the loop when one runs the algorithm. While we will not do a formal proof by induction here, it is worth thinking about the loop invariant, since it also sheds insight into how an algorithm works. Here, the loop invariant is that after $k$ iterations, the following are true:

- The last $k$ positions of the array contain the largest $k$ elements, and are sorted. (The first $n - k$ positions are not necessarily sorted yet.)
- The entire array still contains the same elements as at the beginning.

We would prove this claim by induction on $k$. It holds trivially for $k = 0$, and when we establish it for $k = n$, it implies that the entire array is sorted. A pictorial representation of the loop invariant of BubbleSort is given in Figure 1.

![Figure 1: A pictorial representation of the loop invariant of Bubble Sort.](image)

To analyze the running time of the algorithm, we start — as with any other loop-based algorithm we analyze — from the innermost loop. The inner \texttt{for} loop contains one statement, and it is always going to run in constant time regardless of size of input, giving us $\Theta(1)$ The \texttt{for} loop causes this statement to run $n - i$ times, so the inner loop has a total run time of $\Theta(n - i) \cdot \Theta(1) = \Theta(n - i)$.

Notice that $i$ in the inner \texttt{for} loop runtime is really dependent on which iteration the algorithm is currently on, and so is the time that the inner loop takes. For example, the first time that the loop is run, $i$ is 0, so the loop takes $n$ steps. The last time the loop is run, $i$ is $n-2$, meaning that the inner loop only runs over 1 or 2 steps. To analyze the running time of the entire algorithm, we need to take the total time for the outer loop. That time is obtained by summing the execution time of the inner loop over all iterations of the outer loop. Thus — as always — when analyzing a loop, we get a sum in our formula:

$$
\sum_{i=0}^{n-2} \Theta(n - i) = \Theta(\sum_{i=0}^{n-2} n - i) \overset{(*)}{=} \Theta(\sum_{i=2}^{n} i) = \Theta((\sum_{i=1}^{n} i) - 1) = \Theta(\sum_{i=1}^{n} i) = \Theta(n(n + 1)/2) = \Theta(n^2).
$$

Most of these steps are very simple manipulations, or applications of the arithmetic series (which you should have down pat, along with the geometric one). The one interesting step is the one labeled $(*)$: there, we used that the first sum sums up 2, 3, 4, 5, \ldots as $i$ ranges over its possible values, and instead summed them up in opposite order. Such tricks are often useful to simplify sums.
So we have seen that Bubble Sort takes time $\Theta(n^2)$, i.e., it is a quadratic algorithm. Notice that it also takes a quadratic number of swaps, not just comparisons. Sometimes, people consider comparisons a little cheaper and distinguish carefully between the two. In practice, Bubble Sort is actually quite bad, and it is the worst of the three quadratic algorithms we will see in this class. In fact, as you saw from the link that was posted on the course website, even President Obama knows not to use Bubble Sort for large arrays.

4 Selection Sort

The basic idea of Selection Sort is to find the smallest element, and bring it once and for all to its correct position (first place in the array), then continue with the second position and second smallest element, and continue this way until all elements are in their correct position.\(^1\) So we can write Selection Sort as follows:

```java
for (int i = 0; i < n - 1; i++) {
    int smallestNumIndex = i;
    for (j = i + 1; j < n; j++) {
        if (a[j] < a[smallestNumIndex]) { smallestNumIndex = j; }
    }
    a.swap(i, smallestNumIndex);
}
```

Selection Sort, too, is stable, as we always choose the earlier of equal elements to swap. This happens because we only update `smallestNumIndex` if we find a strictly smaller element.

Again, we would like to think about the loop invariant we would need to prove this algorithm correct. Here, it looks similar to Bubble Sort, only for the beginning of the array instead of the end. After any number $k$ of iterations, the following holds:

- The first $k$ positions of the array contain the smallest $k$ elements, and are sorted. (The last $n - k$ positions are not necessarily sorted yet.)
- The entire array still contains the same elements as at the beginning.

Again, we would use induction on $k$ to prove this claim. Pictorially, we can represent it as in Figure 2:

![Figure 2: A pictorial representation of the loop invariant of Selection Sort.](image)

\(^1\)The textbook describes this algorithm as finding the largest element, then putting it in the last position, and continuing this way. Obviously, it is pretty easy to trade off between those two.
In the analysis, the innermost part takes time $\Theta(1)$, and the inner loop therefore has running time $\Theta(n - i)$. So the total running time is $\sum_{i=0}^{n-2} \Theta(n - i) = \Theta(n^2)$, since we did exactly the same calculation for Bubble Sort. In practice, Selection Sort is a little better than Bubble Sort, since it only has $O(n)$ swaps (though still $\Theta(n^2)$ comparisons).

5 Heapsort

At a more abstract level, we can describe the Selection Sort algorithm as follows:

```java
for (int i = 0; i < n - 1; i++) {
    // find the smallest remaining element and put it in position i
}
```

The thing is that in the implementation given, finding the smallest remaining element takes us time $\Theta(n - i)$, because the remaining array is unsorted. However, we have already seen a data structure earlier that is designed specifically for the purpose of repeatedly finding the smallest remaining element, namely, a heap.

So an alternative version would be to first put all elements into a Min-Heap, repeatedly remove them and copy them over to their target position. If we don’t want to use extra space for a heap, we can actually reuse the same array. The easiest way to do this is to use a Max-Heap instead of a Min-Heap. Then, we repeatedly find the largest remaining element, and swap it into the last position, then decreasing the size of the array used for a heap by 1 (as we do whenever we delete the root element of the heap).

Of course, we first need to ensure that our array satisfies the heap property: we can do this by running `trickleUp` from each element (starting at the root), or `trickleDown` for each element (starting at the leaves).

To analyze the running time, we notice that we first have to put everything into a heap, or turn the array itself into one. Either way, we’re going to spend $\Theta(\log n)$ on each of $n$ elements, for a total of $\Theta(n \log n)$. Then, we run $n$ iterations, and in each iteration, we find and remove the maximum (or minimum) element, which takes $\Theta(\log n)$. So this also takes $\Theta(n \log n)$. The total running time of Heapsort is therefore $\Theta(n \log n)$, which is significantly faster than either Bubble Sort or Selection Sort.

Since it is really just a much faster implementation of Selection Sort, its loop invariant is the same.