Lesson 3
Average Case Analysis

Wholeness of the Lesson

Average-case analysis of performance of an algorithm provides a measure of the typical running time of an algorithm. Although computation of average-case performance generally requires deeper mathematics than computation of worst-case performance, it often provides more useful information about the algorithm, especially when worst case analysis yields exaggerated estimates. Likewise, as discussed in SCI, more successful action results from a deeper dive into silence, into pure intelligence, just as, in archery, the arrow flies truer and hits its mark more consistently if it is pulled back farther on the bow.

Analysis of Simple Sorting Algorithms

BubbleSort, SelectionSort and InsertionSort are among the simplest sorting methods and admit straightforward analysis of running time. For each we will consider best case, worst case and average case running times.

Analysis of BubbleSort.

```
void sort(){
    int len = arr.length;
    for(int i = 0; i < len; ++i) {
      for(int j = 0; j < len−1; ++j) {
        if(arr[j] > arr[j+1]){
          swap(j,j+1);
        }
      }
    }
}
void swap(int i, int j){
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

A. *Running Time Analysis.*
   - Because there are two loops, nested, both depending on n it is $\Theta(n^2)$
   - Best case, average case, and worst case for this algorithm have the same asymptotic running times.

B. *Possible Improvements.*

- It is possible to implement BubbleSort slightly differently so that when the input becomes sorted after some runs of outer for loop, the algorithm will stop. (Exercise)
- At the end of iteration i, the values in `arr[n-i-1]` through `arr[n-1]` are in final sorted order. This observation can be used to shorten the inner loop. The result is to cut the running time in half (though it must still be $\Theta(n^2)$). (Exercise)

Analysis of SelectionSort.

```
void sort(){
    int len = arr.length;
    for(int i = 0; i < len; ++i) {
      int nextMinPos = minpos(i,len-1);
      swap(i,nextMinPos);
    }
}
void swap(int i, int j){
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

```
int minpos(int bottom, int top){
    int m = arr[bottom];
    int index = bottom;
    for(int i = bottom+1; i <= top; ++i) {
      if(arr[i]<m){
        m=arr[i];
        index=i;
      }
    }
    return index;
}
```

A. *Running Time Analysis.*
   - Because there are two loops, nested, both depending on n it is $\Theta(n^2)$
   - Best case, average case, and worst case for this algorithm have the same asymptotic running times.

Analysis of InsertionSort

```
void sort(){
    int len = arr.length;
    int temp = 0;
    int j = 0;
    for(int i = 1; i < len; ++i) {
      temp = arr[i];
      j=i;
      while(j>0 && temp < arr[j-1]){
        arr[j] = arr[j-1];
        j--;
      }
      arr[j]=temp;
    }
}
```

A. *Best-Case Analysis.* The best case for InsertionSort occurs when the input array is already sorted. In this case, the condition in the inner while loop always fails, so the code inside the loop never executes. The result is that execution time inside each outer loop is constant, and so running time is $O(n)$.

B. *Worst-Case Analysis.* Since there are two loops, nested, even in the worst case, the running time is only $\Theta(n^2)$. The worst case for InsertionSort occurs when the input array is reverse-sorted. In this case, in pass $\#i$ of the outer for loop the inner while loop must execute all its statements i times approximately, and so execution time is proportional to $1 + 2 + \ldots + n - 1 = \Theta(n^2)$. Therefore, worst-case running time is $\Theta(n^2)$.

C. *Average-Case Analysis.* It is reasonable to expect that typically, the inner while loop will not work as hard as it does in the worst-case. The result of average case analysis here actually applies to many simple sorting algorithms.

Inversion-Bound Sorting Algorithms

A. In an array `arr` of integers, an *inversion* is a pair $(\texttt{arr}[i], \texttt{arr}[j])$ for which $i < j$ and $\texttt{arr}[i] > \texttt{arr}[j]$.

Example. The array $\texttt{arr} = \{34, 8, 64, 51, 32, 21\}$ has nine inversions:

$$(34,8), (34, 32), (34, 21), (64, 51), (64, 32),$$
$$(64, 21), (51, 32), (51, 21), (32, 21).$$

B. **Theorem** (*Number of Inversions Theorem*). Assuming that input arrays contain no duplicates and values are randomly generated, the expected number of inversions in an array of size n is $\frac{n(n-1)}{4}$.

Proof. Given a list L of n distinct integers, consider L_r, obtained by listing L in reverse order. Suppose x, y are elements of L and $x < y$. These elements must occur in inverted order in either L or L_r (but not both). Therefore every one of the $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs of elements from L occurs as an inversion exactly once in L or L_r. Therefore, on average, one-half of these inversions occur in L itself.

C. **Corollary.** Suppose a sorting algorithm always performs at least as many comparisons as there are inversions on any input array `arr`. Then the average-case running time of this algorithm acting on arrays of distinct elements is $\Omega(n^2)$.

D. **Definition.** A sorting algorithm that always performs at least as many comparisons as there are inversions on any input array `arr` is called an *inversion-bound* algorithm.

Observations About Inversion-Bound Algorithms

We assume all elements of input arrays are distinct — average case analysis does not address the case in which the array has duplicates.

A. BubbleSort, SelectionSort, and InsertionSort can all be shown to be inversion-bound.

B. **Proof that InsertionSort Is Inversion-Bound.** Suppose `arr` is an input array, $i < j <$ `arr.length` and $x =$ `arr`$[i] >$ `arr`$[j] = y$. At some point the initial part of the array consisting of all values preceding y will be in sorted order, and y will need to be placed. y will still be to the right of x. So, in the loop that does the placing, y will have to be compared with x.

Optional: Proofs that BubbleSort and SelectionSort are Inversion-Bound

BubbleSort. Suppose `arr` is an input array, $i < j <$ `arr.length` and $x =$ `arr`$[i] >$ `arr`$[j] = y$. BubbleSort always compares adjacent elements. During BubbleSort, wherever x and y may be in the array, they will eventually become adjacent and then will be compared:

i All elements between the ith and the jth that are bigger than $x =$ `arr`$[i]$ are pushed to the right of $y =$ `arr`$[j]$ (result: no element between x and y is bigger than x)

ii x is pushed to the right of all elements between x and y, including y.

This shows that for every inversion in the input array, at least one (uniquely determined) comparison is performed in BubbleSort.

SelectionSort. Suppose `arr` is an input array, $i < j <$ `arr.length` and $x =$ `arr`$[i] >$ `arr`$[j] = y$. At some point during SelectionSort, all elements of the array less than y will have been moved to the front of the array, to the left of x. In the next pass — called the y *pass* — a crucial comparison (called the *crucial comparison with* x) will determine that x is not the min of the remaining elements. (This comparison may or may not be a comparison between x and y.) This gives us a mapping between inversions and unique comparisons: For any inversion (x, y) that occurs in the array, there is a unique crucial comparison of x that occurs in the y-pass.

Comparing Performance of Simple Sorting Algorithms

A. Demos give empirical data for comparison. (Demos)

B. *Swaps are expensive.* Notice swaps involve roughly seven primitive operations. This is more significant than copies and comparisons. BubbleSort performs (on average)
$\Theta(n^2)$ swaps whereas SelectionSort and InsertionSort perform only $O(n)$ swaps (a "swap" for InsertionSort begins when an element is placed in `temp` and ends when it is placed in its final position). This difference explains why BubbleSort is so much slower than the other two. (Empirical studies show BubbleSort is 5 times slower than InsertionSort and 40% slower than SelectionSort.)

A Refinement of InsertionSort: LibrarySort

Bender, Farach-Colton, Mosteiro observed that InsertionSort is slower than necessary for two reasons:

1. In the ith iteration, the search for where to place the next element among the first (already sorted) i elements is not optimized (*binary search* could be used instead)
2. When the correct place for the next element has been found, the effort to shift all larger elements to the right is slower than necessary (library analogy: leave gaps to make room for new additions)

They implement their ideas for optimizations in a paper that introduces *LibrarySort*. (Their paper is entitled *INSERTION SORT is* $O(n \log n)$.) Their algorithm achieves average case running time of $O(n \log n)$. Demo of LibrarySort

Main Point

A sorting algorithm is *inversion-bound* if it requires, on any given input array, at least as many comparisons as there are inversions in the array. Inversion-bound sorting algorithms always have an asymptotic running time that is $\Theta(n^2)$. Selection Sort, Insertion Sort and Bubble Sort are examples of inversion-bound sorting algorithms.

Maharishi explains in SCI that knowledge is different in different states of consciousness. When consciousness is bounded, it is simply not possible to see higher possibilities in life. When consciousness expands, more possibilities are seen; new directions can unfold; old problems can be solved in new ways.

Connecting the Parts of Knowledge To the Wholeness of Knowledge

Inversion-Bound Algorithms

1 Insertion Sort is an inversion-bound algorithm that sorts by examining each successive value x in the input list and searches the already sorted section of the array for the proper location for x.

2 Library Sort is also a sorting algorithm which, like Insertion Sort, proceeds by examining each successive vaule x in the input list and searches the already sorted section of the array for proper placement. However, in this algorithm, spaces are created in the already sorted section in each pass, and searching the already sorted section is done using Binary Search. The result of these refinements is that Library Sort exceeds the limitations of an inversion-bound sorting algorithm and has average case running time that is O($nlogn$).

3 *Transcendental Consciousness* is the field pure intelligence, the home of all knoweldge, that field "by which all else is known."

4 *Impulses within the Transcendental field.* Maharishi explains that knowledge has organizing power; pure knowledge has infinite organizing power.

5 *Wholeness moving within itself.* In Unity Consciousness, the field of unlimited boundlessness is appreciated in each boundary of existence as its true nature, no different from one's own Self.