10671406

DATA ANALYSIS AND ALGORITHMS

ASSIGMENT

SUMMARY OF INSERTION SORT

There are many different ways to sort. As selection sort runs, the subarray at the beginning of the array is sorted, but the subarray at the end is not. Selection sort scans the unsorted subarray for the next element to include in the sorted subarray.

The the idea behind insertion sort is to loop over positions in the array, starting with index 1. Each new position is a new element to be added to the sorted subarray, and you need to insert it into the correct place in the sorted subarray to the left of that position.

The main step in insertion sort is making space in an array to put the current value, which is stored in the variable key. We go through the subarray to the left of key's initial position, right to left, sliding each element that is greater than key one position to the right. Once an elementless than, or equal to key one is found, it stops and copies key into the vacated position just to the right of this element. The position is not truly vacated, but its element was slid over to the right.

In insertion sort, every iteration of while loop reduces one inversion. Therefore total number of while loop iterations (For all values of i) is same as number of inversions. Therefore overall time complexity of the insertion sort is O(n + f(n)) where f(n) is inversion count. The worst case occurs when the array is sorted in reverse order. So the worst case time complexity of insertion sort is O(n2).

How about the best case? A call to insert, ( which insertion sort calls insert on the elements at indices 1, 2, 3, ,…,n−1) causes no elements to slide over if the key being inserted is greater than or equal to every element to its left. So, if every element is greater than or equal to every element to its left, the running time of insertion sort is Θ(n). This situation occurs if the array starts out already sorted, and so an already-sorted array is the best case for insertion sort.

Insertion sort is very similar to [selection sort](https://en.wikipedia.org/wiki/Selection_sort). As in selection sort, after *k* passes through the array, the first *k* elements are in sorted order. However, the fundamental difference between the two algorithms is that for selection sort these are the *k* smallest elements of the unsorted input, while in insertion sort they are simply the first *k* elements of the input.

The primary advantage of insertion sort over selection sort is that selection sort must always scan all remaining elements to find the absolute smallest element in the unsorted portion of the list, while insertion sort requires only a single comparison when the (*k* + 1)-st element is greater than the *k*-th element; when this is frequently true (such as if the input array is already sorted or partially sorted), insertion sort is distinctly more efficient compared to selection sort. On average (assuming the rank of the (*k* + 1)-st element rank is random), insertion sort will require comparing and shifting half of the previous *k* elements, meaning that insertion sort will perform about half as many comparisons as selection sort on average.

In the worst case for insertion sort (when the input array is reverse-sorted), insertion sort performs just as many comparisons as selection sort.

However, a disadvantage of insertion sort over selection sort is that it requires more writes due to the fact that, on each iteration, inserting the (*k* + 1)-st element into the sorted portion of the array requires many element swaps to shift all of the following elements, while only a single swap is required for each iteration of selection sort. In general, insertion sort will write to the array O(*n*2) times, whereas selection sort will write only O(*n*) times. For this reason selection sort may be preferable in cases where writing to memory is significantly more expensive than reading, such as with [EEPROM](https://en.wikipedia.org/wiki/EEPROM) or [flash memory](https://en.wikipedia.org/wiki/Flash_memory).