Algorithms: Sorting Algorithms

Given a sequence of numbers, you have to arrange them in the ascending or descending order. This is called sorting. Sorting algorithms can be iterative or recursive.

Let’s learn about the different sorting algorithms and analyze their space and time complexity. First we are going to start with the iterative Insertion Sort.

**INSERTION SORT ALGORITHM AND ANALYSIS**

“Insertion sort”, the name suggests that there is an insertion happening. Understand it with an example where you want to arrange the playing cards which are placed on a table, face down.

1. Take the first card in your left hand.
2. Take the second card in your right hand, compare it with the left hand card

If smaller then keep this card in the most left side on the left hand  
 If larger then keep this card in the right most side on the left hand

1. Repeat step 2 over again.

At every new picking of the card, your left hand will always have sorted cards. Let us right the algorithm in the context of an array containing integers which we will sort using **Insertion Sort.**

Insertion\_Sort(A)

{

for( j = 2 to A.length)  
{  
 key = A[ j ];  
 //insert A[ j ] into sorted sequence A[1 … j-1]  
 i = j – 1;  
 while( i > 0 and A[ i ] > key)  
 A[i+1] = A[i];  
 i = i - 1;  
  
 A[i+1] = key;  
  
}

}

Let array = 9, 6, 5, 0, 8, 2, 7, 1, 3 (please follow the algorithm above to sort the list).

**Time complexity:**

**The worst case time complexity for insertion sort** can be calculated as below. For worst case to happen, the elements should be already in descending order in the array for which we have to do ascending order sorting using Insertion Sort.

|  |  |  |  |
| --- | --- | --- | --- |
| For j | Comparison | Movements | Total Operations |
| 2 | 1 | 1 | 2 |
| 3 | 2 | 2 | 4 |
| 4 | 3 | 3 | 6 |
| … | … | … | … |
| N | n-1 | n-1 | 2(n-1) |

Therefore, total operations done: 2 + 4 + 6 + … + 2(n-1)

Or: 2(2-1) + 2(3-1) + 2(4-1) + … + 2(n-1)

Or: 2(1) + 2(2) + 2(3) + … + 2(n-1)

Or: 2 (1 + 2 + 3 + … + n-1)

* 2(n(n-1))/2
* **O(n2)**

**The best case time complexity for Insertion Sort** can be calculated as below. For best case to happen, the array should already have the sorted elements inside it.

|  |  |  |  |
| --- | --- | --- | --- |
| For j | Comparison | Movements | Total Operations |
| 2 | 1 | 0 | 1 |
| 3 | 1 | 0 | 1 |
| 4 | 1 | 0 | 1 |
| … | … | … | … |
| N | 1 | 0 | N - 1 |

Total operations for best case time complexity: 1 + 1 + 1 + … + N-1

When the array is already sorted you will just compare the element in you right hand (card analogy) with the rightmost element in the left hand only once for each element in the array.

**Therefore, best case time complexity = Ω(n)**

**Space Complexity of Insertion Sort:**

Apart from the input array A[], I only need three variables ‘key’, ‘i’ and ‘j’. So, whatever is the size of the input, I only need 3 extra variables for Insertion sort to work.

Therefore, the space complexity for Insertion Sort is **O(1)** or constant space.

**Note: Whenever a sorting algorithm executes in constant space, we call such a sorting algorithm INPLACE algorithm**

In Insertion Sort, we saw that the time complexity depends on the number of comparisons involved and the number of movements involved. So how can we decrease this complexity? Let’s try some methods.

1. Using Binary Search instead of sequential search in the sorted list

|  |  |  |
| --- | --- | --- |
| Comparisons using Binary Search | Movements using Binary Search | Total complexity |
| O(logn) | O(n) (no reduction) | O(nx(n-1) elements)) = O(n2) |

Using binary search will not reduce the time complexity of the Insertion Sort algorithm.

1. Using Linked List for insertion

|  |  |  |
| --- | --- | --- |
| Comparisons using Linked List | Movements using Linked List | Total complexity |
| O(n) (no reduction) | O(1) (direct insertion) | O(nx(n-1 elements)) = O(n2) |

Using linked list will also not reduce the time complexity of the Insertion Sort Algorithm.

**Time and Space complexity for Insertion Sort**

|  |  |  |  |
| --- | --- | --- | --- |
| **Time Complexity** | **Best Case** | **Worst Case** | **Average Case** |
|  | **Ω(n)** | **O(n2)** | **Θ(n2)** |

|  |  |
| --- | --- |
| **Space Complexity** | **O(1)** |