**Insertion Sort**

Table of Contents

[Space Complexity 5](#_Toc66035895)

[Proving Algorithm Correctness 6](#_Toc66035896)

The insertion sort algorithm has a time complexity of O(n2). Consider the following array.

5 2 4 6 1 3

The algorithm loops through the array and selects each index position as a key, and compares it to the values before it. This results in having a sorted array every time a new key is selected. The loop does not begin from the first index since the first index is already ‘sorted’. The first key is thus index 1. Inserting it into the correct position we have.

2 5 4 6 1 3

The second key is index 2. Thus,

2 4 5 6 1 3

Continuing in this pattern:

2 4 5 6 1 3

1 2 4 5 6 3

1 2 3 4 5 6

Although the insertion sort algorithm has a time complexity of O(n2), in practice it is faster than most sorting algorithms. This is due to the fact that it maintains a sorted part in the array. When we insert a value into some position, we know for certain that everything before it is already sorted, so we do not need to go over that part. This reduces the number of operations required.

Here is the code for the algorithm:

void insertionSort(int arr[], int n)  
{  
 for (int i=1; i<n; i++) //   
 {  
 int key = arr[i]; //   
 int j = i; //   
 while (arr[j-1] > key && j > 0) //   
 {  
 arr[j] = arr[j-1]; //   
 j--; //   
 }  
 arr[j] = key; //   
 }  
}

C++

The number of times each line of code runs is given as comments on the right.

The for loop has to run a total of (n-1) times and the check has to be done n times since, the extra check being the one that breaks the loop. The while loop however, performs the check depending on the value of j, and the number of times the code inside runs depends on the array, so the exact value cannot be predicted. The while loop thus performs its check an unknow number of times tj, for each run of the for loop, from 1 to (n-1), while the code inside runs (tj-1) times.

Thus, for the algorithm, the total time T(n) is

In the best-case scenario, the array is already sorted. The while loop is checked (n-1) times, and is immediately found to be false every time (so each break of the loop takes constant time). The code inside the while loop does not run. Thus,

So, we have a time complexity of O(n).

In the worst-case scenario, the array is sorted in reverse. From j=1 to j=n-1, the while loop must check its condition j times, and the code inside runs (j-1) times. Thus,

So, we have a time complexity of O(n2).

A question could be raised about the while loop in the insertion sort algorithm. Since it only exists to search for a value, binary search could be used, which would reduce the time complexity to O(log n) for that part instead of O(n). However, after finding the position the key needs to be inserted in, the array would need to be shifted, which would take linear time, so the overall effect on the algorithm is the same. On the other hand, if a linked list were used, insertion would take constant time, but binary search cannot be used on a linked list, since it is not an array. So, the search part would again take linear time here, which would again cause the overall algorithm to have the same time complexity.

## Space Complexity

Space complexity is a measure of the additional space needed other than the input data, to run the program.

Take two functions, one which, for some reason, creates a copy of the input array, and another which loops through the input array, swapping values with the help of an extra variable. The space needed by the first function grows linearly with the input size, so it has a space complexity of O(n). The second function however, will only need that one extra variable regardless of how large the input array is. It has a space complexity of O(1).

The insertion sort algorithm needs three new variables to run, the integers i, j and key. Regardless of the input size, it always needs just these three variables. So, although it has a time complexity of O(n2), it has a space complexity of O(1). There are other sorting algorithms, like merge sort, that have a better time complexity, but worse space complexity.

## Proving Algorithm Correctness

One of the methods used to prove that an algorithm is correct for any given input is called the loop invariant method. It essentially says that the algorithm must have a property, called the loop invariant property, that satisfies the conditions of initialization, maintenance and termination. This means:

* The property exists before the loop is initialized
* The property is true before every iteration, and after every iteration
* The property is true at the end of the loop

For insertion sort, this property is that it always has a sorted part of the array. During initialization, the value of index 0 is ‘sorted’. At the beginning and end of each loop, the sorted part is still there. At the end of the loop we have a completely sorted array.