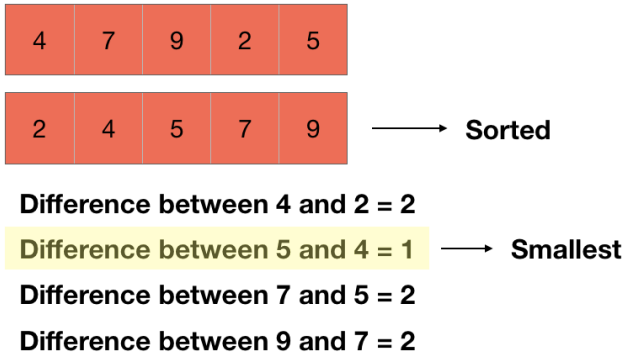
Sorting

So far, we have studied about algorithms and its analysis. Now let's start learning some real algorithms.

The first set of algorithms we are going to learn is sorting algorithms. You must have already guessed it from the name that these algorithms arrange data in order. So, why we chose to learn sorting algorithms as our first algorithms?

No doubt, as per functionality, sorting algorithms are important. For example, let it be features like sort by price, sort by relevance, etc on a shopping site or sorting data in school, office, bank, etc, sorting algorithms are used everywhere.

But this is not the only reason why we are learning these as our first algorithm. Sorting algorithms are building blocks of many other algorithms i.e., many algorithms emerge from sorting algorithms. Also, many algorithms work on sorted data, so we first need to sort the data, if it is not already sorted, to use such algorithms. In many cases, sorting is not even a necessary part of getting the result of a problem but if the data is sorted then it becomes much easier to find the solution. For example, take a problem of finding two numbers with the smallest difference from a set of numbers. The solution for the problem can be very easily found if we first sort the numbers and then just find the differences between the adjacent numbers and return the smallest number among these differences.



In this section, we are going to learn 4 sorting algorithms - Insertion sort (/course/algorithms-sort-it-out/), Bubble sort (/course/algorithms-bubble-sort/), Heapsort (/course/algorithms-heapsort/), and Counting sort (/course/algorithms-count-and-sort/). Quicksort and Mergesort are discussed in the further sections.

The first three algorithms (Insertion sort, Bubble sort, and Heapsort) are all comparison sort i.e., they compare numbers to arrange them in order. However, counting sort is a non-comparison sort algorithm i.e., it arranges numbers without comparing them. Interesting, isn't it? So, let's start learning all these algorithms.

Insertion Sort

Many times we develop an algorithm based on the way we accomplish a given task manually and Insertion sort is also that kind of algorithm. It is similar to sorting by hands. We pick up an element and then place it in the right position. Let's take an example of placing 5 bars in sorted order, with the largest at the bottom.

So, it is clear from the animation that we are taking an element and placing it where it should be like we do when we sort something with our hands. Picking up two element and placing them at each other's position is basically swapping two elements and we achieve this thing in programming using swapping.

Thus, we start with iterating over an array and compare the current element with the previous one and if the elements are not in order then we swap the elements. We repeat the comparison process until the previous element to the current element and the current element itself are in order. This will become clear from the illustration given below.

You can see from the picture given above that we stopped the swapping process once the previous adjacent element to the current element and the current element itself are in order.

Let's look at the illustration of insertion sort.

So, the steps of the insertion sort are :-

1. Start iterating over the array to be sorted.
2. Compare the current element in the iteration with the previous element adjacent to it. If they are in order, continue the iteration, otherwise, go to the next step.
3. Swap the two elements.
4. Repeat the second step.

Code for Insertion Sort

Now, we know the working of insertion sort and the logic behind it. So, let's see how to convert those logics into code.

The first thing we need to do is to start the iteration, so let's use a for loop for this i.e., `for i in 1 to A.length`.

Now, we have to compare the current element and the previous element and swap them if they are in the wrong order and this process needs to be repeated (as shown in the above picture). So, let's use another loop to iterate back in the array and do this. `while j>1 and a[j-1]>a[j]` - Here, 'j' is the current element and the statements of this loop will be executed if the elements are in the wrong order (`a[j-1]>a[j]`) and we will decrease the value of 'j' to make this loop iterate backward.

So, let's make a working code out of these.

```
INSERTION-SORT(A)
  for i in 1 to A.length
    j = i
    while j>1 and a[j-1]>a[j]
      swap(a[j-1],a[j])
      j=j-1
```

`INSERTION-SORT(A)` - 'INSERTION-SORT' is the name of the function and 'A' is the array passed to it.

`for i in 1 to A.length` - We are iterating over the array 'A'. Now, we have to compare the current element `a[j]` with the adjacent previous element `a[j-1]` and check whether they are in order or not. This is done by the condition `a[j-1] > a[j]` of the while loop. If they are not in order, then we are swapping them `swap(a[j-1],a[j])`, otherwise, the outer for loop will just continue the iteration.

</>
There can be different ways to implement these logics into code. So, you can come up with a different style of code to implement the logic of the insertion sort. And this is the entire point of algorithms, that one should know how to solve a problem and then coding up that solution is a different part.

C **Python** **Java**

```c
#include <stdio.h>

// function to swap values of two variables
void swap(int *a, int *b) {
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}

void insertion_sort(int a[], int size) {
  int i;
  for(i=0; i<size; i++) {
    int j = i;
    while((j>0) && (a[j-1]>a[j])) {
      swap(&a[j-1], &a[j]);
      j = j-1;
    }
  }
}

int main() {
  int a[] = {4, 8, 1, 3, 10, 9, 2, 11, 5, 6};
  insertion_sort(a, 10);

  //printing array
  int i;
  for(i=0; i<10; i++) {
    printf("%d ",a[i]);
  }
  printf("\n");
  return 0;
}
```

Analysis of Insertion Sort

At the first glance, we can see that there is one loop inside another and rest lines of the codes are going to take constant time, so intuitively we can tell that the algorithm is of $O(n^2)$ but let's analyze the algorithm properly and see if this intuition is correct or not.

The first statement is going to run n+1 times (n times for iterating every element of the array and 1 more time when the condition will be checked and failed). And the statement `j=i` is going to run n times, every time the outer loop is executed.

The number of times inner loop is going to run will depend upon the input because the condition `a[j-1]>a[j]` is going to vary from input to input. So, let's assume it is going to run t_i times for each value of i. For example, when i is 1, it will run t_1 times, when i is 2, it will run t_2 times and so on. Since the value of i is changing from 1 to n, we can write that the total number of times while loop is going to execute is $t_1 + t_2 + \ldots + t_n = \sum_{i=1}^{n} t_i$.

Now, the statements `swap(a[j-1],a[j])` and `j=j-1` are going to execute one time less than the number of times while loop is running (they will not run when the while loop will run and its condition will be failed). So, they are going to run $t_i - 1$ times for each value of i and thus a total of $\sum_{i=i}^{n}(t_i - 1)$ times.

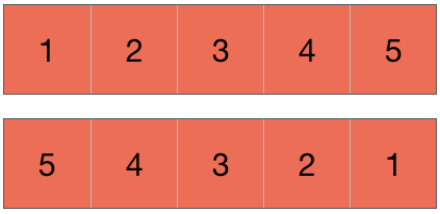
	Cost	Times
INSERTION-SORT(A)		
for i in 1 to A.length	c_1	n+1
j = i	c_2	n
while j>1 and a[j-1]>a[j]	c_3	$\sum_{i=1}^{n} t_i$
swap(a[j-1],a[j])	c_4	$\sum_{i=1}^{n}(t_i - 1)$
j=j-1	c_5	$\sum_{i=1}^{n}(t_i - 1)$

So, we can write the total running time

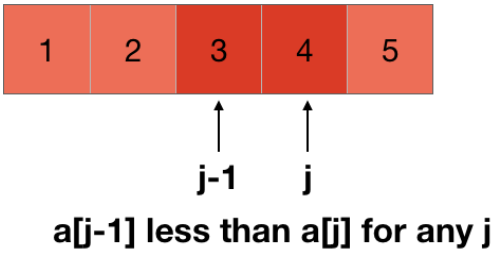
$T(n) = c_1(n+1) + c_2 n + c_3 \sum_{i=1}^{n} t_i + c_4 \sum_{i=1}^{n}(t_i - 1) + c_5 \sum_{i=1}^{n}(t_i - 1)$

Best Case Analysis of Insertion Sort

In the algorithm of insertion sort, we know that the for loop is going to iterate from 1 to n in any case and it is while loop which is going to execute differently for different inputs. For example, take these two cases.



Now, the best case would be when the while loop will run minimum possible times and it is possible when the condition `a[j-1]>a[j]` fails every time and this will happen in the case when the array is already sorted because in that case only every j-1th will be smaller than the jth element of the array.



In this case, the value of t_i is going to be 1 for any value of i because every time the statement of while loop will be executed single time and its condition will be failed. Thus, we can write $T(n)$ as:

$T(n) = c_1(n+1) + c_2n + c_3\sum_{i=1}^{n} 1 + c_4\sum_{i=1}^{n}(1-1) + c_5\sum_{i=1}^{n}(1-1)$

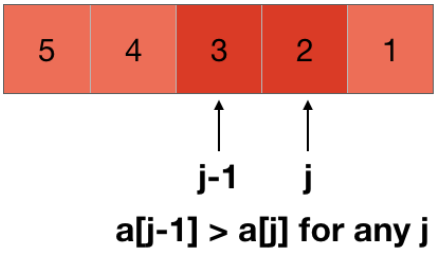
$= c_1(n+1) + c_2n + c_3\sum_{i=1}^{n} 1$

$= c_1(n+1) + c_2n + c_3n$

And the above expression is **linear**. So, the insertion sort has a linear best case running time i.e., $\Omega(n)$ and this is going to occur when the array is already sorted.

Worst Case Analysis of Insertion Sort

The maximum number of times the while loop will run when the condition of the while loop `a[j-1]>a[j]` will be satisfied every time. And this means that any j-1th element is greater than the jth element of the array which means that our array is sorted in reverse order.



In this case, the while loop will iterate from j=i to j=2 and thus the value of t_i will be i-1 (2 to i).

So, the total running time, $T(n) = c_1(n+1) + c_2n + c_3\sum_{i=1}^{n}(i-1) + c_4\sum_{i=1}^{n}(i-2) + c_5\sum_{i=1}^{n}(i-2)$

$= c_1(n+1) + c_2n + c_3\frac{n(n-1)}{2} + c_4\frac{(n-2)(n-1)}{2} + c_5\frac{(n-2)(n-1)}{2}$

The above equation is a quadratic one, so the insertion sort has a quadratic worst-case running time i.e., $O(n^2)$ when the array is sorted in the reverse order.

In this chapter, we dealt with the first sorting algorithm - Insertion sort and its analysis. So, let's move to the next chapter to learn the next sorting algorithm - Bubble Sort (/course/algorithms-bubble-sort/).

> Programming is the art of algorithm design and the craft of debugging errant code.
>
> - Ellen Ullman

PREV (/course/algorithms-masters-theorem/) (/course/algorithms-bubble-sort/) NEXT

Further Readings

- ➔ Sorting a list using selection sort in Python (/blog/article/sorting-a-list-using-selection-sort-in-python/)
- ➔ Sorting a list using insertion sort in Python (/blog/article/sorting-a-list-using-insertion-sort-in-python/)
- ➔ Sorting an array using selection sort in C (/blog/article/sorting-an-array-using-selection-sort-in-c/)
- ➔ Sorting an array using insertion sort in C (/blog/article/sorting-an-array-using-insertion-sort-in-c/)
- ➔ Sorting an array using insertion sort in Java (/blog/article/sorting-an-array-using-insertion-sort-in-java/)
- ➔ Sorting an array using selection sort in Java (/blog/article/sorting-an-array-using-selection-sort-in-java/)