Sorting Algorithms - Insertion Sort

CSC 413 - *Algorithms & Complexity Analysis*

David Peculiar Olorunsola - 19CD026583

Abiola Oluwafikunayomi - 20CD028223

Arokoyo Miracle Omonori - 20CD028233

Ebofin Iyioluwa Emmanuella - 20CD028243

Mojoyinola Omotola - 20CD028253

First Okonkwo - 20CD028263

Chukwuebuka Ubah - 20CD028273

Oluwatimilehin Adekunle - 20CG028033

Tochukwu Akpa - 20CG028043

Oluwafisayomi Babatunde - 20CG028053

John Daudu - 20CG028063

Oghenetega Eko-Brotobor - 20CG028073

Daniel Faji - 20CG028083

Daniel Joseph-Osawe - 20CG028093

Fumnanya Mowete - 20CG028103

Joyce Oludele - 20CG028123

Toluwani Osin Oluwapelumi - 20CG028133

Sophia Oghenekvwe Umukoro - 20CG028143

Description

The insertion sort algorithm was formally defined by John Mauchly in 1946, in the first published discussion of computer sorting - *The Theory and Techniques for Design of Electronic Digital Computers* (Donald Knuth, 1998).

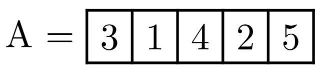
However, its use predates computing as it is a natural way for humans to sort items - when people manually sort cards in a hand, most use a method that is similar to insertion sort (Robert Sedgewick & Kevin Wayne, 2014).

Its efficient for tiny sets of data and is stable (it does not change the relative order of equal keys). In addition to these, it also sorts elements “in place” – without allocating extra memory.

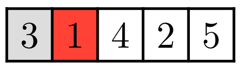
It works by iterating through the elements in the array and growing the final sorted version by comparing each element to the largest value in the sorted sub-list (which would be the previous element) and placing them in the correct order.

Example

Consider an example array:

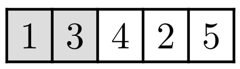


1. Initially, the first two elements - `A[0]` and `A[1]` are compared:

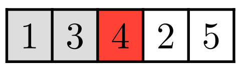


   Here, 3 is greater than 1, hence they are not in the ascending order and `A[1]` is not at its correct position.

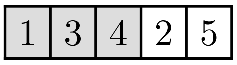
   Thus, swap 1 and 3 - making our sorted sub-list now comprise of the first two elements.



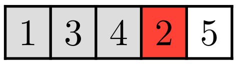
2. Now, we compare the next two elements - `A[1]` and `A[2]`:



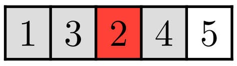
   4 is already greater than 3, so we continue on.



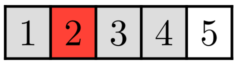
3. We compare the next two, `A[2]` and `A[3]`:



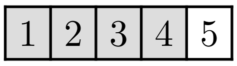
   We swap 4 and 2, as they're not in order.



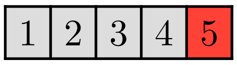
   After swapping, 3 and 2 are still not in order, so we swap them too.



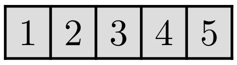
   2 is greater than 1, so we continue on.



4. We compare the final two elements, `A[3]` and `A[4]`:



   4 is greater than 5, so the list is sorted.



Formal definition

We formally define the algorithm as follows (for a zero-based indexing array):

```
procedure insertion_sort (array: list of sortable items)
    i ← 1

    while i < length(array)
        j ← i
        while j > 0 and array [j - 1] > array[j]
            swap array [j - 1] and array[j]
            j ← j - 1
        end while
        i ← i + 1
    end while
end procedure
```

Here, ← denotes assignment, e.g. `x ← y` means that the value of `x` changes to the value of `y`.

The outer loop runs over the elements starting from the second one - `array[1]`. The inner loop then moves element `array[i]` to its correct place so that after the loop, the first `i + 1` elements are sorted.

Implementation

An implementation that takes input from the user in C++:

```
#include <iostream>
#include <string>
#include <sstream>
#include <vector>

using namespace std;

// n is the length of arr[]
void insertionSort (int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int val = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > val) {
          arr [j + 1] = arr[j];
          j -= 1;
        }

        arr[j + 1] = val;
    }
}

// helper func. to print an array
void printArray (int arr[], int n) {
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }

    cout << endl;
}

// entry point
int main() {
    string input;
    cout << "Enter the array elements separated by spaces: ";

    // get an entire line of input,
    // not just up to the first space
    getline(cin, input);

    // create a dynamically-sized vector
    // for splitting the input into
    vector<int> arr_vec;
    arr_vec.reserve(input.length());

    // chunk the stream on every space occurrence
    stringstream ss(input);
    string chunk;

    while (getline(ss, chunk, ' ')) {
        try {
            // transform to an int and add to `arr_vec`
            arr_vec.push_back(stoi(chunk));
        } catch (invalid_argument e) {
            // if stoi() fails
            cout << "please enter numbers..." << endl;
            break;
        }
    }

    // vectors store data contiguously
    // so we use v.data() to get the address
    // use it to sort and print
    insertionSort(arr_vec.data(), arr_vec.size());
    printArray(arr_vec.data(), arr_vec.size());

    return 0;
}
```

Sample run:

```
$ g++ insertion.cpp -o ./insertion
$ ./insertion
Enter the array elements separated by spaces: 3 1 4 2 5
1 2 3 4 5
```

Time complexity

The runtime of the algorithm is closely related to the number of *inversions* in the array – where an inversion is a pair of elements that are out of order (templatetypedef, 2013).

While sorting, the algorithm shifts elements around to their correct position by swapping them if they form an inversion. Therefore, a sorted array would have 0 inversions - because every element is bigger than the one before and smaller than the one after.

From our algorithm, the time complexity $T(n)$ can be the addition of the number of elements in the array, n comparisons (the outer loop) and the number of swaps that occur in each of the inner loops.

Given that the number of swaps equals the number of inversions I, we can then define out time complexity $T(n)$ as,

$$T(n) = n \text{ comparisons} + I \text{ swaps}$$

Best case - $\Omega(n)$

The best case input is an array that is already sorted – which means it has no inversions. In this case the running time is linear – $\Omega(n+0) = \Omega(n)$.

In the main iteration every element is compared with the last element of the sorted sub-section, i.e. the element just before, and no swaps occur.

Worst case - $O(n^2)$

The simplest worst case input is an array that's in reverse order. This results in every iteration having to scan and shift the entire sorted sub-section in order to insert the elements.

Therefore, for every element in the array, the inner loop will run i times, where i is the index of the element. This leads to

$$I = 1 + 2 + 3 + ... + (n-1) = \frac{n(n-1)}{2}$$

$\therefore T(n) = \left(n + \frac{n(n-1)}{2}\right)$, which results in a quadratic running time – $O(n^2)$.

Average case - $\Theta(n^2)$

For each of the $\frac{n(n-1)}{2}$ possible inversions between distinctly-positioned pairs of elements, on average about half of them will be inversions, and half will not. So we can establish that the average number of inversions is $I = \frac{n(n-1)}{4}$.

$\therefore T(n) = \left(n + \frac{n(n-1)}{4}\right)$, which also results in a quadratic running time – $\Theta(n^2)$.

Space complexity

As the algorithm only moves elements around in the array and does not allocate any extra memory, the total space complexity in all cases is $O(n)$, where n is the number of elements in the array.

The auxilary space complexity is $O(1)$ – it has a constant space requirement regardless of the size of the input data.

Real life applications

1. *Sorting a small list of numbers and datasets:* Insertion sort is efficient for sorting a small list of numbers. Its advantage lies in its simplicity and effectiveness when dealing with a limited number of elements. For example, if you have a list of 10 or 20 numbers that need sorting, insertion sort can perform this task quickly without requiring much computational overhead.

2. *Online Algorithms:* Its ability to work efficiently with live data makes insertion sort suitable for online algorithms. For instance, when continuously receiving new data elements and needing to maintain a sorted list, insertion sort can be effective.

3. *Organizing cards in a card game:* In card games, particularly those that involve maintaining a player's hand in a sorted order (like a hand of playing cards), insertion sort can be useful. When a player receives a new card, insertion sort can quickly find the correct position for the new card within the already sorted hand by comparing it to the existing cards, maintaining the hand's order efficiently.

4. *Educational Purposes:* Insertion sort is often used as an introductory algorithm in computer science courses to teach the concept of sorting algorithms due to its simple logic and ease of understanding.

Bibliography

Donald Knuth. (1998). *The Art of Computer Programming: Vol. 3 - Sorting and Searching* (2nd ed., p. 82).

Robert Sedgewick, & Kevin Wayne. (2014). *Algorithms* (4th ed., p. 250).

templatetypedef. (2013). https://stackoverflow.com/a/17055342