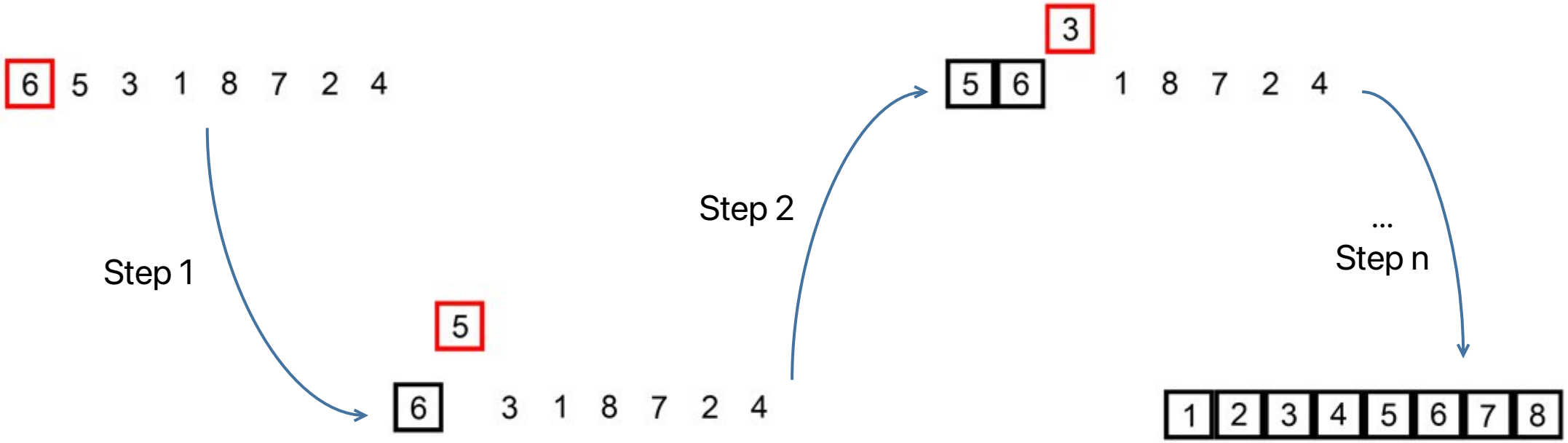
Insertion Sort

An overview, by Group 3

What is “Insertion Sort”?

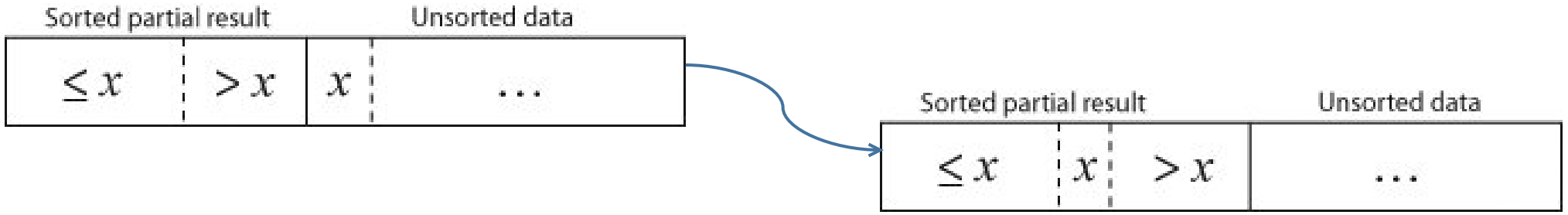
- Historically predates computing – humans instinctively use it to sort various items like playing cards.
- First formally defined by John Mauchly in 1946 during the Moore School Lectures.
- A sorting algorithm that works by iterating through the elements in the array and growing the final sorted version 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.

What is “Insertion Sort”?



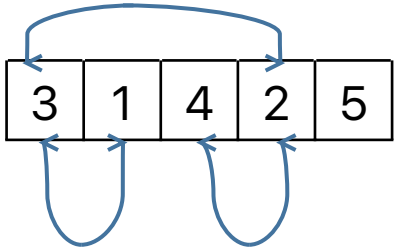
How does it work?

- We go from left to right, element by element, and then divide the array internally into a sorted section (kept to the left) and the remainder of the array.
- As we select an element, we compare it with the largest value in the sorted list (which would be its immediate predecessor) — if it's larger, then there's no inversion and we leave it as it is.
- If it's smaller (there's an inversion), we shift every element in the sorted list to the right until we find a suitable position for it.



Okay…what's an inversion?

- An inversion between two elements in an array occurs when they're not positioned in ascending order:

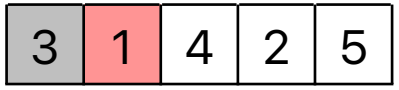


- This array has 3 inversions: (3, 1), (3, 2), and (4, 2).

Let's see an example?

- Let's have an array we'll call *A*:

- We make the sorted subsection include just the first element *A[0]*, and we attempt to place the second, *A[1]*:



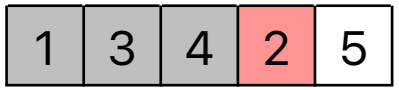
- There's an inversion, as 3 > 1, so we shift 3 to the right by swapping it with 1:

- We're at the left now, so we assume it's solved and go on to the next element *A[2]*:

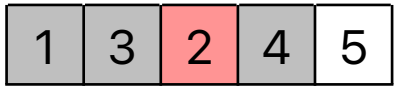
Let's see an example?

- There's no inversion, because 3 < 4, so we extend the sorted portion into *A[2]*:

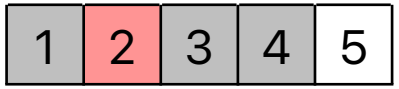
- Next up is *A[3]*:



- There's an inversion because 4 > 2, so let's shift 4 to the right:



- Now comparing with the largest in the sorted list to the left of 2, which is 3, we see that there's another inversion, and we shift 3 to the right:



Let's see an example?

- 2 > 3, so we move on to the next element, *A[4]*:

- There's no inversion, and we're at the end of the array, so we can conclude that it's sorted!

We want a formal definition, please.

- We can define this formally as:

```
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.

Some code...perchance?

- A C++ function that implements this algorithm could be written as:

```
// 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;
    }
}
```

- The report has more code to handle user input etc., it's omitted here for brevity.

Space complexity

- The algorithm uses no additional memory — it works with the already allocated space for the array, so the auxiliary space complexity is $\mathrm{O}(1)$.
- The total space complexity is $\mathrm{O}(\mathrm{n})$, as the space used increases linearly with the number of elements in the array.

Timeeeeeeeeeeee complexity

- To calculate the time complexity, we need a way to quantify the "work done" in the algorithm.
- The algorithm has two main loops:

```
procedure insertion_sort (array)
    ...
    while i < length(array)
        ...
        while j > 0 and array [j - 1] > array[j]
            ...
        end while
        i ⟵ i + 1
    end while
end procedure
```

runs n times
"let's check if this element n is correctly positioned"

runs a variable number of times depending on the value of element n
"do we need to keep shifting element n to the left?"

Time complexity

- The outer loop always runs n times, proportional to the number of elements in the array.
- We can prove that swapping two elements can only decrease the number of inversions in the array – therefore if an array has 0 inversions, we can consider it sorted.
- The “work done” by the algorithm can now be considered as:

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

Time complexity – Best case

- The best case is an array that's already sorted – there are 0 inversions.
- In this case, our equation becomes:

$$T(n) = n + 0 = n$$

- We can therefore conclude that we have a best running time of $\Omega(n)$ which is linear.

Time complexity – Worst case

- The worst case would be an array sorted in reverse order, with the number of inversions:
  - If there's 2 elements, there would be 1 inversion.
  - 3 elements, 3 inversion
  - 4 elements, 6 inversions
  - ...and so on
- We can deduce that for n elements, the maximum number of inversions is the n - 1th triangular number:

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

Time complexity – Worst case

- Now our $T(n)$ is:

$$T(n) = \left(n + \frac{n(n-1)}{2}\right)$$

- We can now infer our running time to be quadratic, i.e. $\mathrm{O}(\mathrm{n}^2)$.

Time complexity – Average case

- Out of $\frac{n(n-1)}{2}$ possible inversions, on average about half of them will be inverted (meaning they'll need to be swapped), and half will not.
- Our equation thus becomes:

$$T(n) = \left(n + \frac{n(n-1)}{4}\right)$$

- This still results in a quadratic running time - $\Theta(n^2)$.

Real life applications

- *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.
- *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 as is capable of sorting data while receiving it.
- *Sorting a small list of numbers and datasets:* Insertion sort is efficient for sorting a small list of numbers – most sorting implementations fall from quicksort to insertion sort once the number of elements dips below a certain threshold.

Image Attributions

- *https://www.pexels.com/photo/person-shuffling-playing-cards-9859348/*
- *https://en.wikipedia.org/wiki/File:Insertion-sort-example-300px.gif*
- *https://en.wikipedia.org/wiki/File:Insertionsort-before.png*
- *https://en.wikipedia.org/wiki/File:Insertionsort-after.png*