Chapter 6: Sorting Algorithms

1. **Introduction**
2. **Selection Sort**
3. **Insertion sort**
4. **Bubble sort**
5. **Quick sort**
6. **Merge Sort**

Introduction

- The term "sort " means "distribute objects according to certain criteria ."
- In algorithms, the term “Sort" is often attached to the process of classifying a sequence of elements in a given order.

Example:

- Sort N integers in ascending order.
- Sort N students in descending order of their averages.
- Sort N names in ascending alphabetical order.
- Sort files by their sizes.

Introduction

- Generally, any set with a total order can provide a sequence of elements to be sorted.
- Two sorting categories:
  - **Internal sorts :** methods intended for limited masses of data, stored in a data structure located in central memory (Example: tables).
  - **External sorts :** methods intended for large masses of data, stored in data structures such as files.

Introduction

- Sorting is fundamental to many other problems, and after sorting, many problems become easy to solve. For example :
  - Searching for an element.
  - Uniqueness of elements: after sorting test adjacent elements.
  - Determine the smallest and largest element.
  - Determine the k^{th} largest element in $O(1)$.

Selection Sort

Principle:

Repeat:

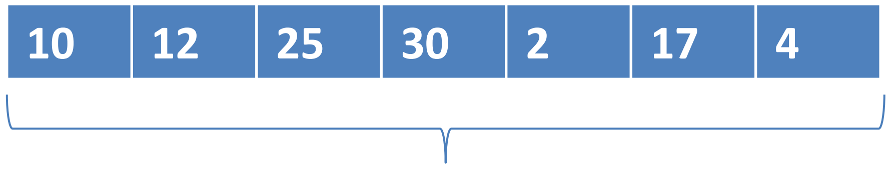
- Find the smallest (largest) element => **Selection** .
- Put the selected item at the Begin of the unsorted part.

<u>In another way</u>

- **Find the smallest element and put it at the Begin of the array**
- **Find the 2nd smallest element and put it in second position**
- **Find the 3rd smallest element and put it in 3rd place,**
- **. . .**

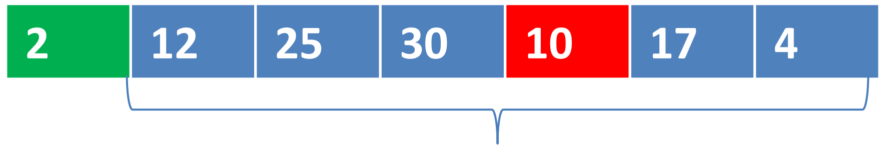
Sort by selection

Example: T



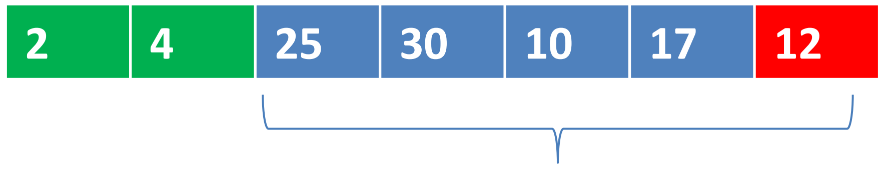
min = 2

Step 1 : Swap 2 (min) and 10



min = 4

Step 2 : Sawp 4 (min) and 12



min = 10

Sort by selection

<u>Step 3 </u>: Exchange between 10 (min) and 25

2	4	10	30	25	17	12

<u>Step 4 </u>: Exchange between 12 (min) and 30

2	4	10	12	25	17	30

<u>Step 5 </u>: Exchange between 17 (min) and 25

2	4	10	12	17	25	30

<u>Step 6 </u>: 25 (min) is in its place

2	4	10	12	17	25	30

Sort by selection (Algorithm)

```
Procedure SelectionSort (Var T : Array of integers , N : integer)
    i , j, indMin , Temp : integer ;
Begin
    For i ←1 to N-1 do
        indMin = i ;
        For j ranging from i+1 to N
            if (T[j] < T[ indMin ]) then
                indMin = j;
            End If
        End For

        Temp = t[i ];
        T[i ] = T[ indMin ];
        T[ indMin ] = Temp ;
    End for
END
```

/* search for the minimum index */

/* swap values between the current box and the minimum */

Sort by selection (Algorithm)

```
Procedure SelectionSort (Var T : Array of integers , N : integer)
    i , indMin : integer ;
Begin
    For i going from 1 to N-1 do
        indMin ← MinIndex (T,i,N)
        Swap (T, i, indMin );
    End for
END
```

- The function MinIndex returns the index of the minimum of the [i, N] part of the array
- The function Swap performs the permutation between the element i and the minimum.

Selection Sort (Complexity)

- The worst case and the average case are the same.
    - To find the smallest elements, (n-1) iterations are needed,
    - For the 2nd smallest element, (n-2) iterations are performed, .
    - To find the last smallest element, 0 iterations are performed.
- Selection Sort always performs **exactly N–1 swaps**:
- One swap per outer iteration

Complexity :

$(n - 1) + (n - 2) + \ldots + 2 + 1 = (n(n - 1)) / 2 \rightarrow O(N^2)$

Disadvantage :

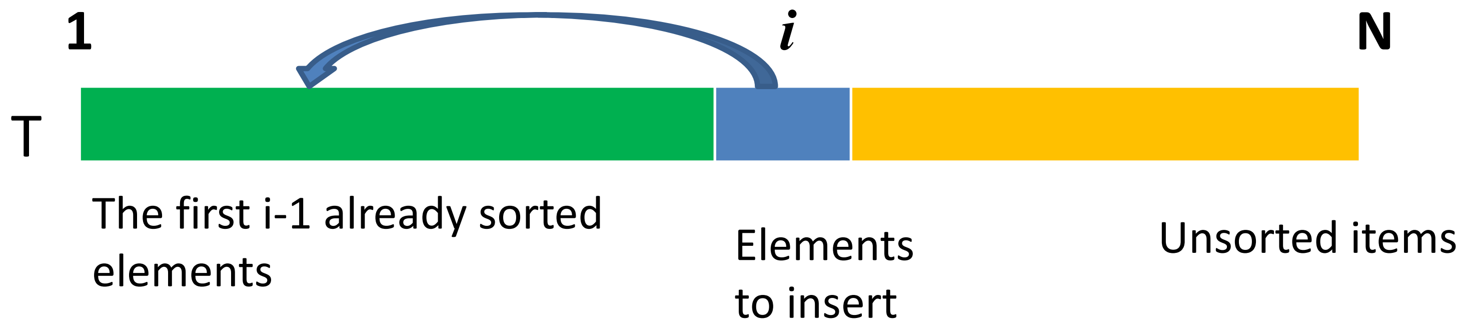
- even if the array is sorted, it still scans all remaining elements at every step.
- It never breaks early.

Insertion Sort

Principle:

Repeat:

- Insert the next element to sort into the part that is already sorted previously.
- The starting part considered as sorted is the first element.
- May have to move several elements for insertion .



Sort by Insertion

Principle: (the card player's sorting!)

- Order the first two elements
- Insert the 3rd element so that the first 3 elements are sorted
- Insert the 4th element in “its” place so that...
- . . .
- Insert the n^{th} element in its place .

At the end of the i^{th} iteration, the first i elements of T are sorted.

Sort by Insertion

Example:

T	10	12	5	30	2	17	4

Step 1 : Insert 12 in its place

10	12	5	30	2	17	4

Step 2 : Insert 5 in its place

5	10	12	30	2	17	4

Sort by Insertion

<u>Step 3 </u>: Insert 30 in its place

5	10	12	30	2	17	4

<u>Step 4 </u>: Insert 2 in its place

2	5	10	12	30	17	4

<u>Step 5 </u>: Insert 17 in its place

2	5	10	12	17	30	4

<u>Step 6 </u>: Insert 4 in its place

2	4	5	10	12	17	30

Insertion Sort (Algorithm)

```
Procedure InsertionSort (Var T : Array of integers , N : integer)
   i, k: natural ;
   temp : integer ;
Begin
   For i← 2 to N do
      temp ← t[i ]; k ← i ;
        While (k > 1 and t[k-1] > temp ) do
                  T [k ] ← T[k - 1 ];
                  k ← k - 1 ;
        End While
        T[k ] ← temp ;
   End for
END
```

Insertion Sort (Algorithm)

- At iteration i, the subarray T[1..i−1] is already sorted.
- We take the new value, Then we “push” bigger values to the right
- Insertion sort does this:
  - Take value temp
  - Move bigger elements one step to the right
  - Insert temp in the right spot
- It is a **shift**, not a swap.

- Insertion sort does this:
- Take value temp
- Move bigger elements one step to the right
- Insert temp in the right spot
- It is a **shift**, not a swap.

Insertion Sort (Complexity)

- Since we don't necessarily have to scan the entire already sorted part, the worst case, best case and average case may differ from each other .

- **Best case:** Each element is inserted at the end of the sorted part. In this case, we do not have to move any elements. Since we have to insert (n-1) elements, each generating only one comparison, the complexity is O(n).

- **Worst case:** Each element is inserted at the Begin of the sorted part. In this case, all elements in the sorted part must be moved at each iteration.

  The I^th iteration generates (i-1) comparisons and exchanges of values. $O\ (\ n^{\,2}\ )$

Bubble sort

Principle:

Repeat:

- Go through the array comparing successive elements two by two, swapping them if they are not in order.
- Repeat as many permutations are made.

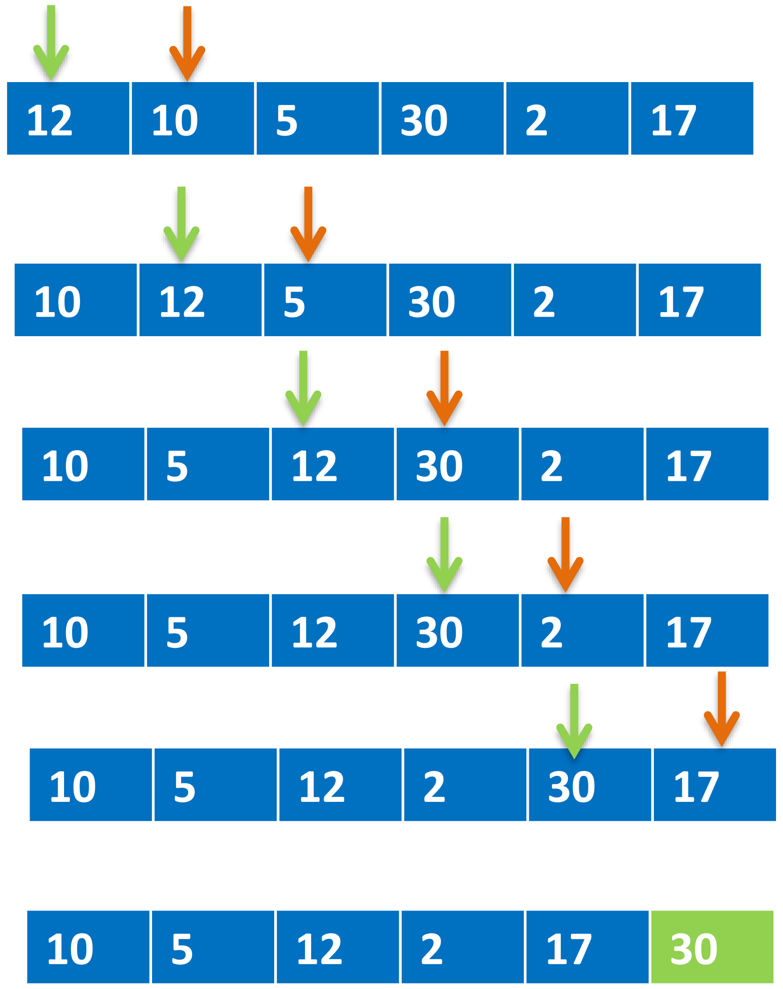
- After the first run, the largest element being in its final position, it no longer needs to be processed.
- The rest of the table, however, is still in disarray. We must therefore go through it again, stopping at the penultimate element.
- After this second travarsel, the two largest elements are in their final position.
- It is therefore necessary to repeat the traversal of the array, until the two smallest elements are placed in their final position .

Sorting in Bubble

Example: T

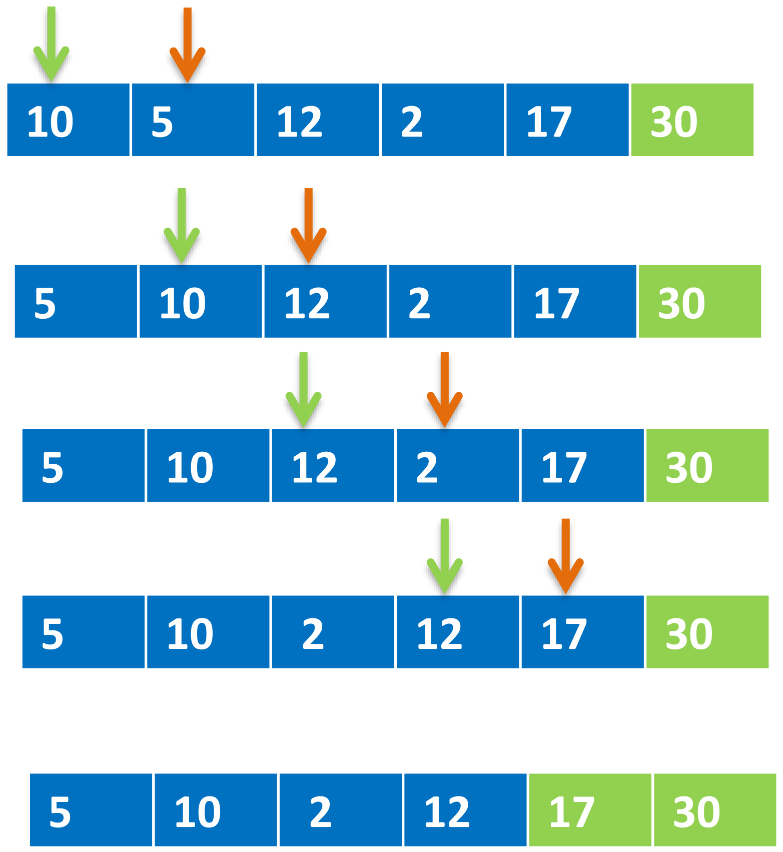
12	10	5	30	2	17

Iteration 1 :



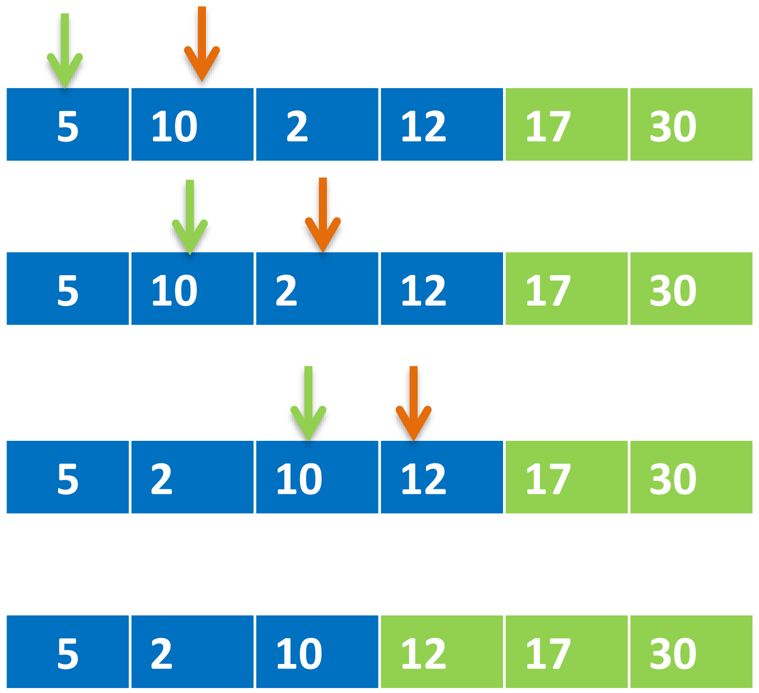
Sorting in Bubble

<u>Iteration 2</u> :



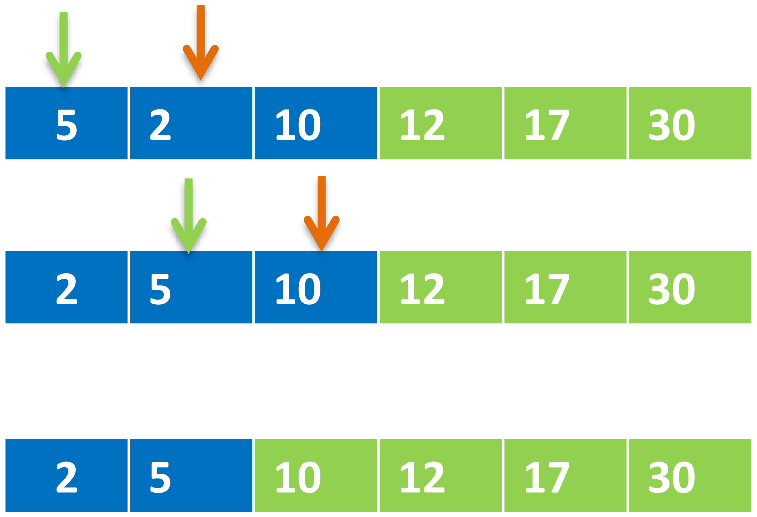
Sorting in Bubble

Iteration 3 :



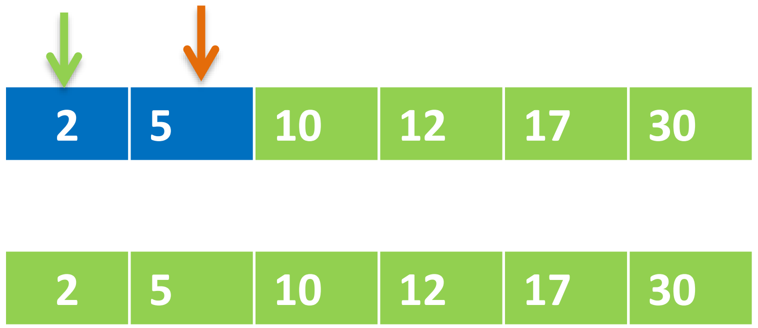
Sorting in Bubble

Iteration 4 :



Sorting in Bubble

Iteration 5 :



No change algorithm stop

Bubble Sort (Algorithm)

```
TriBulle Procedure (Var T : Array of integers , N : integer)
  i : integer;
 B: Boulean ;
Begin
     Repeat
              B ← False;
              For i ← 1 to N -1 do
                   If T[i] >T[i+1] then
                        Swap (T, i, I+1);
                        B ← True;
                   Endif
              End for
              N ←N-1;
     Until B=false
END
```

Bubble Sort (Complexity)

- Bubble sort is often teached as an algorithmic example, because its principle is simple. But it is the slowest of the commonly taught sorting algorithms, and is therefore rarely used in practice.

- **Best case:** the array is sorted. In this case, we don't have to move any elements. The complexity is O(n).

- **Worst case:** the array is sorted in reverse order. In this case , the complexity is O (n^2).

Quick sort

Principle:

Invented in 1960 by Sir Charles Antony Richard Hoare based on the ***divide and conquer paradigm*** consists of :

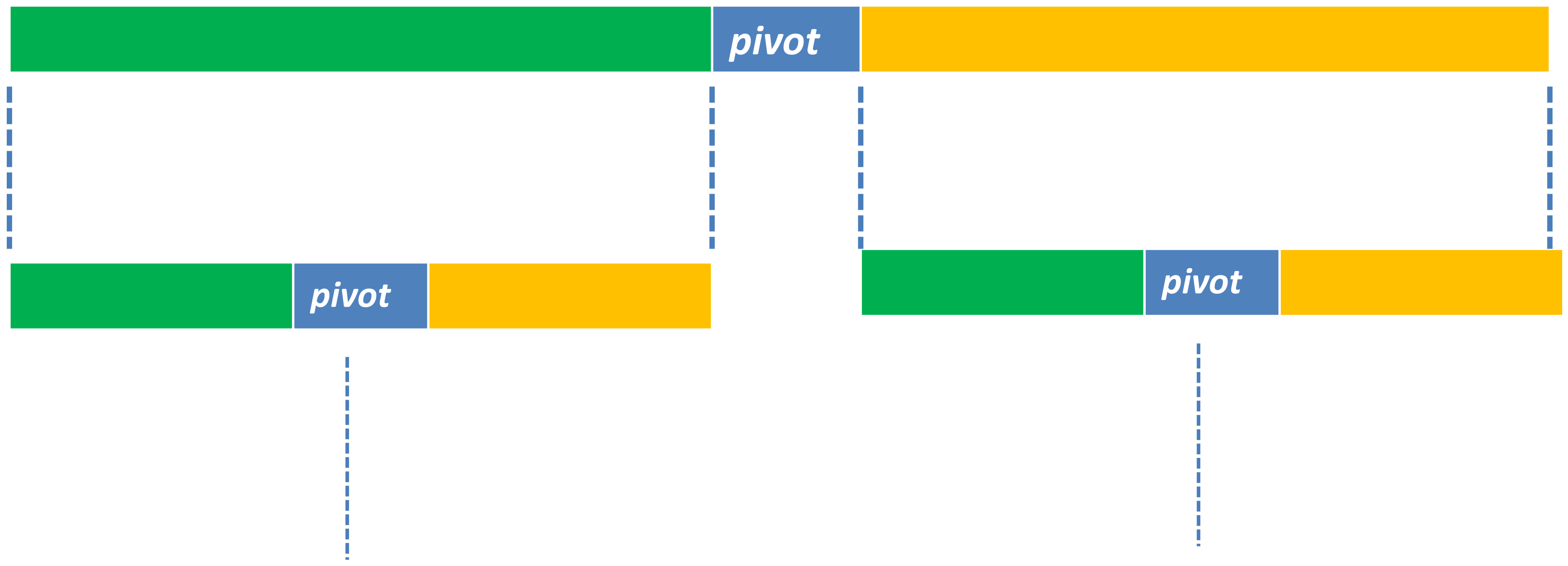
- Choose a “pivot ” element
- Divide the set or array into two subsets
- One subset contains the elements less than the pivot and the second contains the elements greater than the pivot. **(Partitioning)**
- Repeat the procedure recursively until each set contains only one element.

Quick sort

Principle:

Elements below the pivot

Elements above the pivot

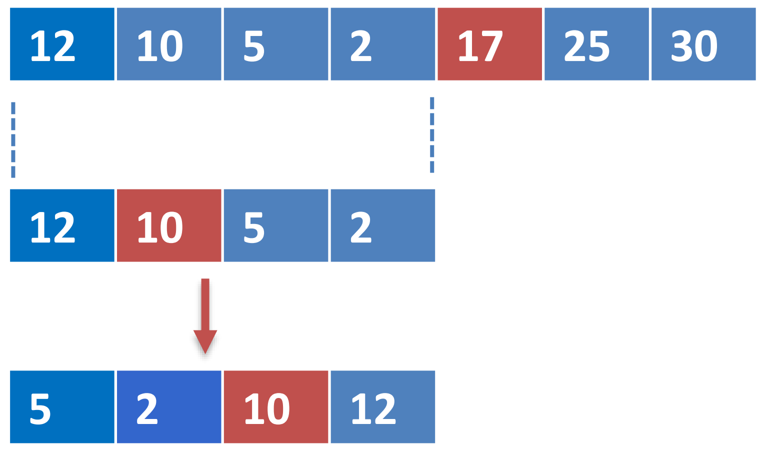


Quick Sort

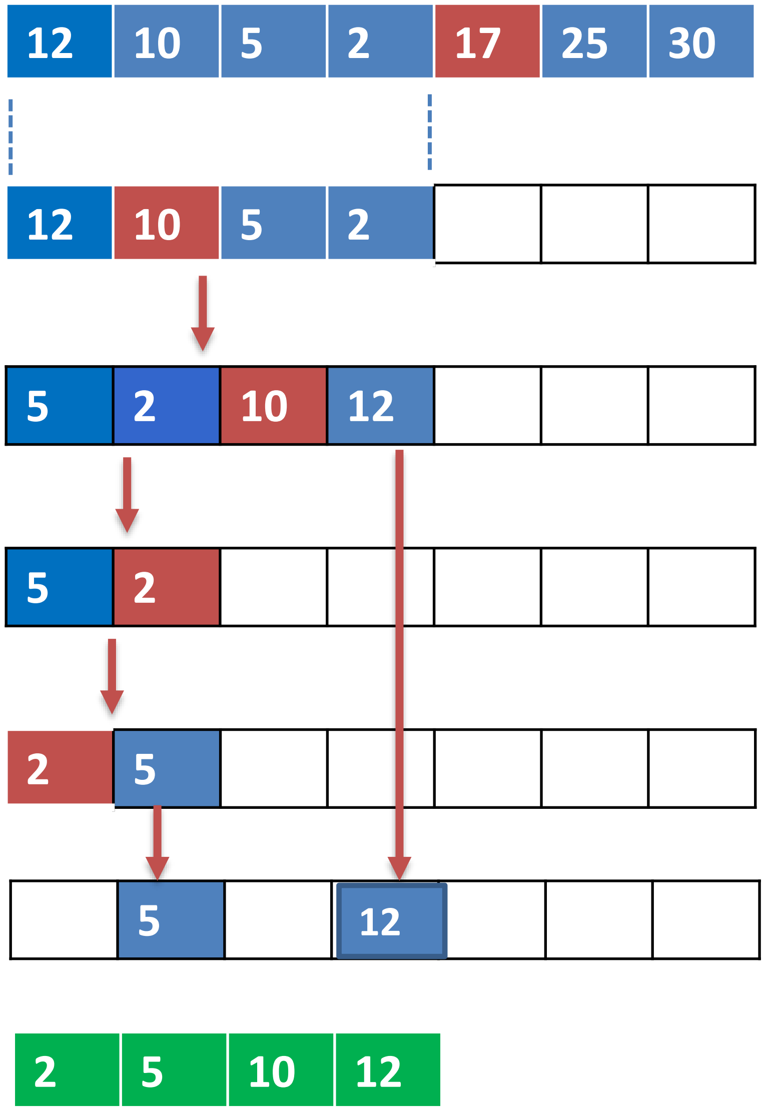
Example: T

12	10	5	30	2	25	17

- **Choose a pivot (17)**
- **Put elements <17 in the left part of the pivot and elements >=17 in the right part**



Quick Sort



Quick Sort (Algorithm)

```
QuickSort Procedure (Var T : Array of integers , start, end: integer)
    pivot: integer
Begin
    If start <end then
        pivot ← partition(T, start, end);
        QuickTri (T, start, pivot -1);
        QuickTri (T, pivot +1, end);
    Finished
END
```

Quick Sort (Algorithm)

```
Function partition(Var T: Array of integers , start, end:integer): integer
    pivot: integer
Begin
    pivot ← T[(start + end) /2];
     i←start ; j←end ;
    While i<j do
          While T[i] < pivot do
                    i ←i+1;
          End While
          While T[j] > pivot do
              j ←j-1 ;
          End While
          Swap (T, i, j);
    End While
          Returns i;
END
```

Quick Sort (Complexity)

- The most sensitive part of sorting is choosing the pivot. The choice can be disastrous: if the pivot is the smallest element in the Array for each choice, then the quick sort degenerates into a selection sort.

- The complexity of this sorting is :
  - Best case, $O(N \log_2 N)$;
  - On average, in $O(N \log_2 N)$;
  - In the worst case, in $O(N^2)$.

Merge Sort

Principle:

Invented in 1948 by Goldstine and Von Neumann in 1948.

- Applies the principle of " **divide and conquer**".
- Based on the idea that if there are two sequences of sorted elements, it is very easy to obtain a third sequence of sorted elements, by "interclassing" (or merging) the two previous sequences. The principle of this sorting consists of:
  - Divide the set or array into two subsets
  - Sort the two subsets.
  - Merge the two subsets

Merge Sort

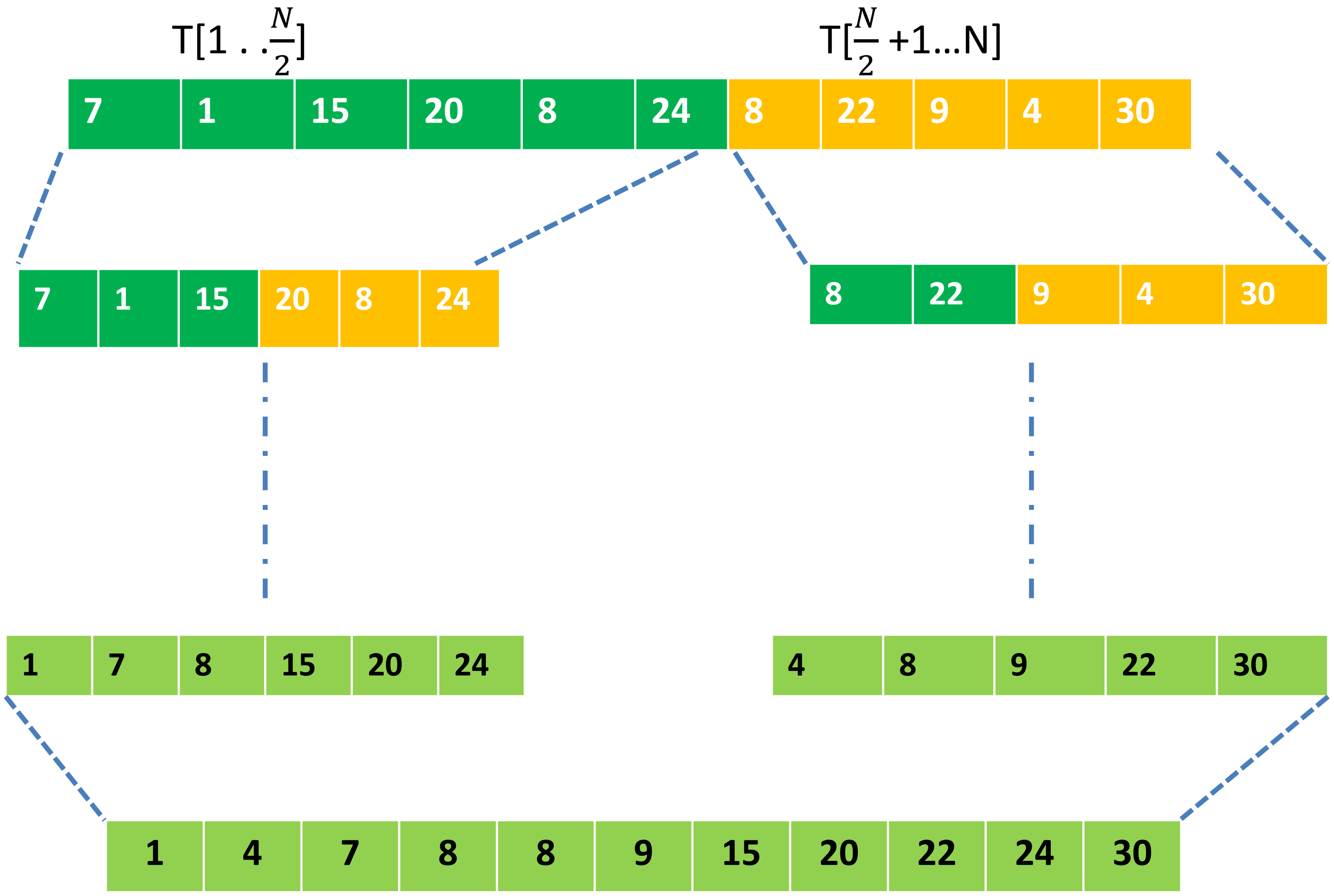
Principle:

Given an array (or list) T[1, ...,n]:

- If n = 1, return the table T !
- Else :
    - Sort the subarray T[1 . . .n/2]
    - Sort the subarray T[n/2 + 1 . . . n]
    - Merge these two sub-arrays.
- This is a divide-and-conquer algorithm.

Merge Sort

Principle:



Merge Sort (Algorithm)

```
Procedure MergeSort (Var T : Array of integers , start, end: integer)
    Mil: Integer
Begin
    If start < end then
        Mil ← (start + end)/ 2
         MergeSort(T,deb,mil);
        MergeSort(T, mil+1, end);
        Merge (T, Start, Mil, End);
     Endif
END
```

Merge Sort

```
Procedure Merge (var T: Array, deb, mil, end: integer)
        T1: Array
Begin
        i ← 1; j←Mil+1;
        For k ← Start to end do
                If (j > end) or (i ≤ mil and T[i] ≤ T[j] ) then
                        T1 [k] ← T[i];
                        i ←i+1
                Else
                        T1 [k] ← T[j];
                        j ←j+1
                End if
        End for
                Copy (T, T1 , K);
END
```

Merge Sort (Complexity)

- Merge sort is an optimal sort on lists (arrays), with complexity O(n log (n)). It involves breaking down a list into two sublists each half as small, sorting them separately, and then merging the results into a sorted list.
- The number of comparisons required is of the order O(n logn).
- The memory space required is O (n).

Properties of sorting algorithms

Stable sorting :

- A sort is said to be stable if it preserves the initial ordering of elements that the order considers equal.
- To define this notion, it is necessary that the collection to be sorted is ordered in a certain way (for example lists or array).

Sorting in place:

- A sorting is said to be *in place* if it only uses a very limited number of variables and directly modifies the structure it is sorting.
- This character can be very important if you don't have a lot of memory.