Projekt je financirala Europska unija iz Europskog Socijalnog fonda

Algorithmic basics: Presentation and Comparison of the main Sorting Algorithms according to their Complexity

Valérie-Anne Nicolas

vnicolas@univ-brest.fr

Motivation: why study sorting algorithms?

Sorting problem: find a permutation of a linear collection of elements such that elements are in increasing (or decreasing) order

- *order relation* is needed (over elements or keys)
- *arrays* are the most appropriate underlying data structure (random access vs list sequential access)

- Data sorting is a *common feature* in software engineering (e.g. to ease processing, to display the evolution of a data over the years, the ranking of a sporting event...)
- To prepare data so that *more efficient algorithms* can be used (e.g. dichotomic search instead of sequential search)
- For *educational purposes*, to illustrate many algorithmic concepts: complexity, design paradigms (iterative, recursive, divide and conquer), data structures...

Sorting algorithms

- Internal sorting algorithms

  → When data to be sorted fits into main memory

- External sorting algorithms
  - When data to be sorted is stored on an external memory device (for example a disk), because it is too large to fit into main memory
  - The cost of data access (swap) is very disadvantageous

  → *Merge sort*

- Complexity: order of approximation with the big O notation (*worst*, best, average cases)
  - *Number of comparisons*
  - Number of swaps
  - Space complexity (« in place » or not...)

Overview of main sorting algorithms

	comparisons	swaps	space
➢ Internal sorting algorithms			
◆ Basic selection based algorithms			
• Bubble sort (and Shaker sort)	$O(n^2)$	$O(n^2)$	**O(1)**
• Selection sort	$O(n^2)$	**O(n)**	**O(1)**
◆ Basic insertion based algorithms			
• Sequential insertion sort	$O(n^2)$	$O(n^2)$	**O(1)**
• Binary insertion sort	**$O(n \log_2(n))$**	$O(n^2)$	**O(1)**
◆ Advanced selection based algorithms			
• Quick sort (or Hoare sort)	*$O(n^2)$*	*$O(n^2)$*	$O(\log_2(n))$
• Heap sort	**$O(n \log_2(n))$**	**$O(n \log_2(n))$**	**O(1)**
➢ External sorting algorithm: *Merge sort*	**$O(n \log_2(n))$**	**$O(n \log_2(n))$**	$O(n)$

Selection based sorting algorithms

Aim: to sort an array in increasing order.

Principle of selection based sorting algorithms:

- *select one element* (in general the smallest),
- put it in its final place,
- then sort out the rest of the elements.

Bubble sort

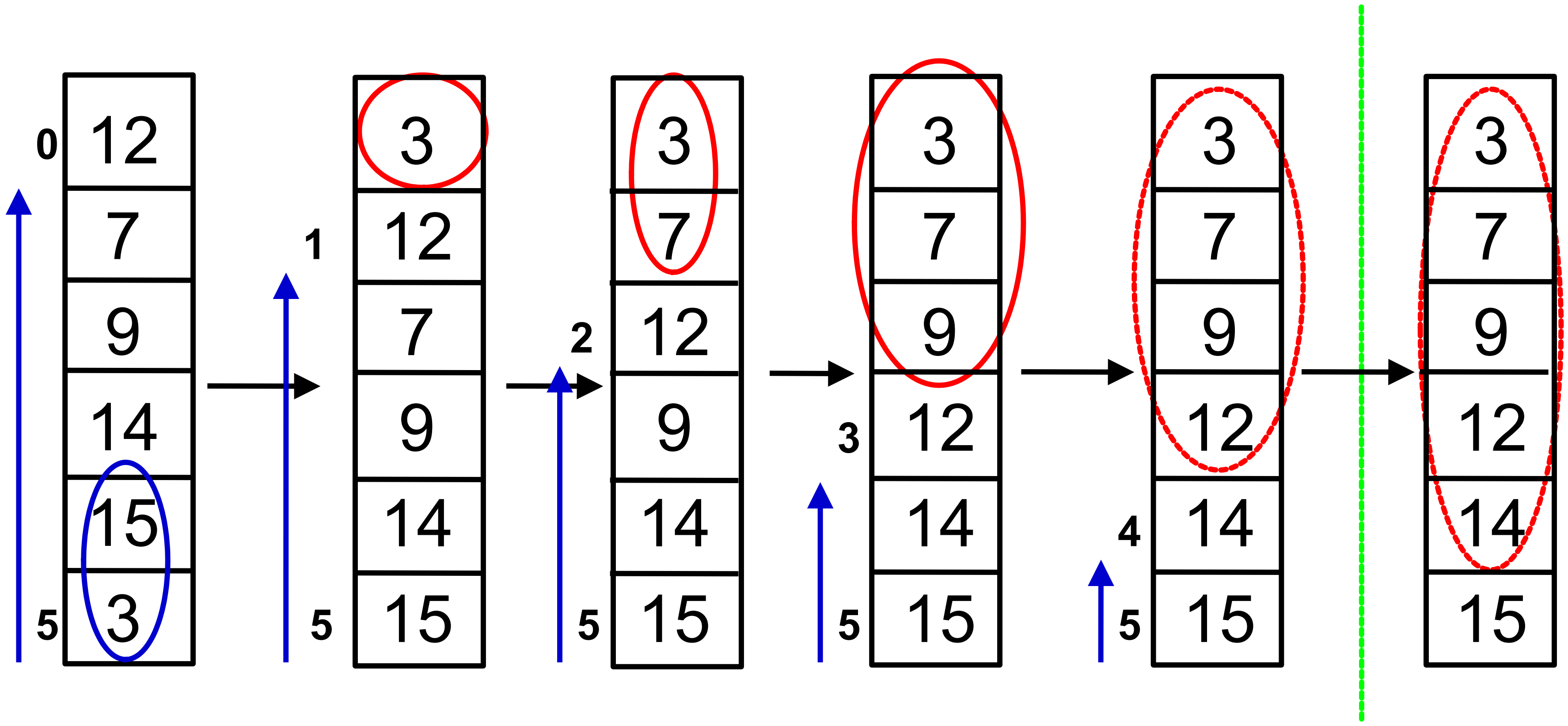
Principle: The smaller (lighter) elements are brought up to the top (beginning) of the array, like bubbles.

Method:

1. Scan the array starting from the end (the bottom),
2. exchange two successive elements each time they are not in the right order,
3. repeat steps 1 and 2 until all the elements are sorted.

Thus, at the end of the first run, the first (minimum) element of the array is in its place. At the end of the second run, the second (minimum) element of the array is in its place. And so on...

Bubble sort: illustration



→ Here, 5 steps are needed → But 4 steps when optimised

Iterative version of Bubble sort

```
void BubbleSort (t_elt T[], int n) {
   int i, j;                    // t_elt is the type of the elements in the array
   t_elt x;                     // n is the size of the array T
   for (i=0; i<n-1; i++)
     for (j=n-1; j> i; j--)
       if (T[j] < T[j-1])
           { x = T[j];
             T[j] = T[j-1];
             T[j-1] = x;   }
 }
```

Complexity: $O(n^2)$
comparisons and swaps

Recursive version of Bubble sort

```
void BubbleSortRec (t_elt T[], int l, int r) {
     int j;
     t_elt x;
     if (l < r)
        { for (j=r; j>l; j--)
              if (T[j] < T[j-1])
                       { x = T[j];
                         T[j] = T[j-1];
                         T[j-1] = x; }
           BubbleSortRec(T,l+1,r);
        }
  }
```

Initial call:
BubbleSortRec(T,0,n-1)

Optimised Bubble sort (Recursive version)

```
void BubbleSortRec2 (t_elt T[], int l, int r) {
    int j, b;          // boolean b
    t_elt x;
    if (l < r)
        {  b = 1;               // b = true
           for(j=r; j>l; j--)
              if (T[j] < T[j-1])
                     { x = T[j];
                       T[j] = T[j-1];
                       T[j-1] = x;
                       b = 0;  }     // b = false
           if (!b) BubbleSortRec2(T,l+1,r);
        }
}
```

→ Checks if the array is sorted before continuing

Initial call:
BubbleSortRec2(T,0,n-1)

Complexity: O(n^2) comparisons and swaps

Optimised Bubble sort (Iterative version)

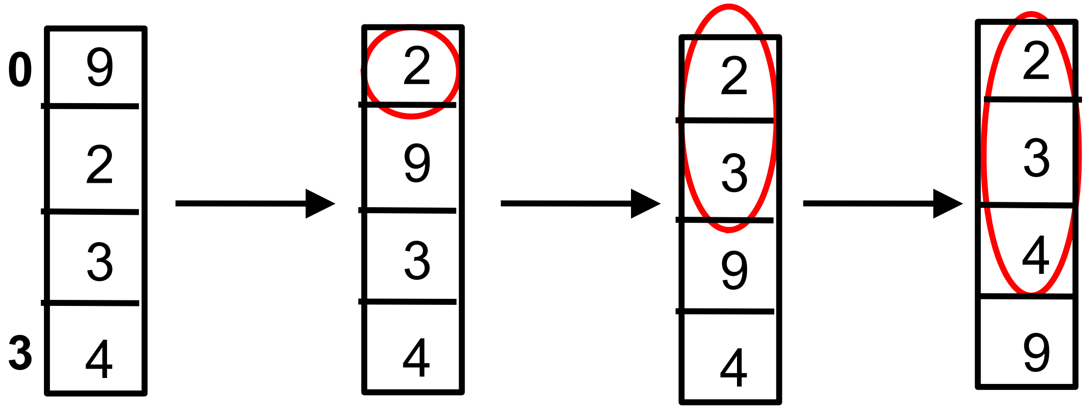
```
void BubbleSort2 (t_elt T[], int n) {
   int i=0, j, b=0;            // boolean b = false
   t_elt x;
   while ((i < n-1) && (b == 0)) {
        b = 1;                 // b = true
        for (j=n-1; j> i; j--)
            if (T[j] < T[j-1])
              { x = T[j];
                T[j] = T[j-1];
                T[j-1] = x;
                b = 0; }    // b = false
        i++;
    }
}
```

→ Checks if the array is sorted before continuing

Shaker sort

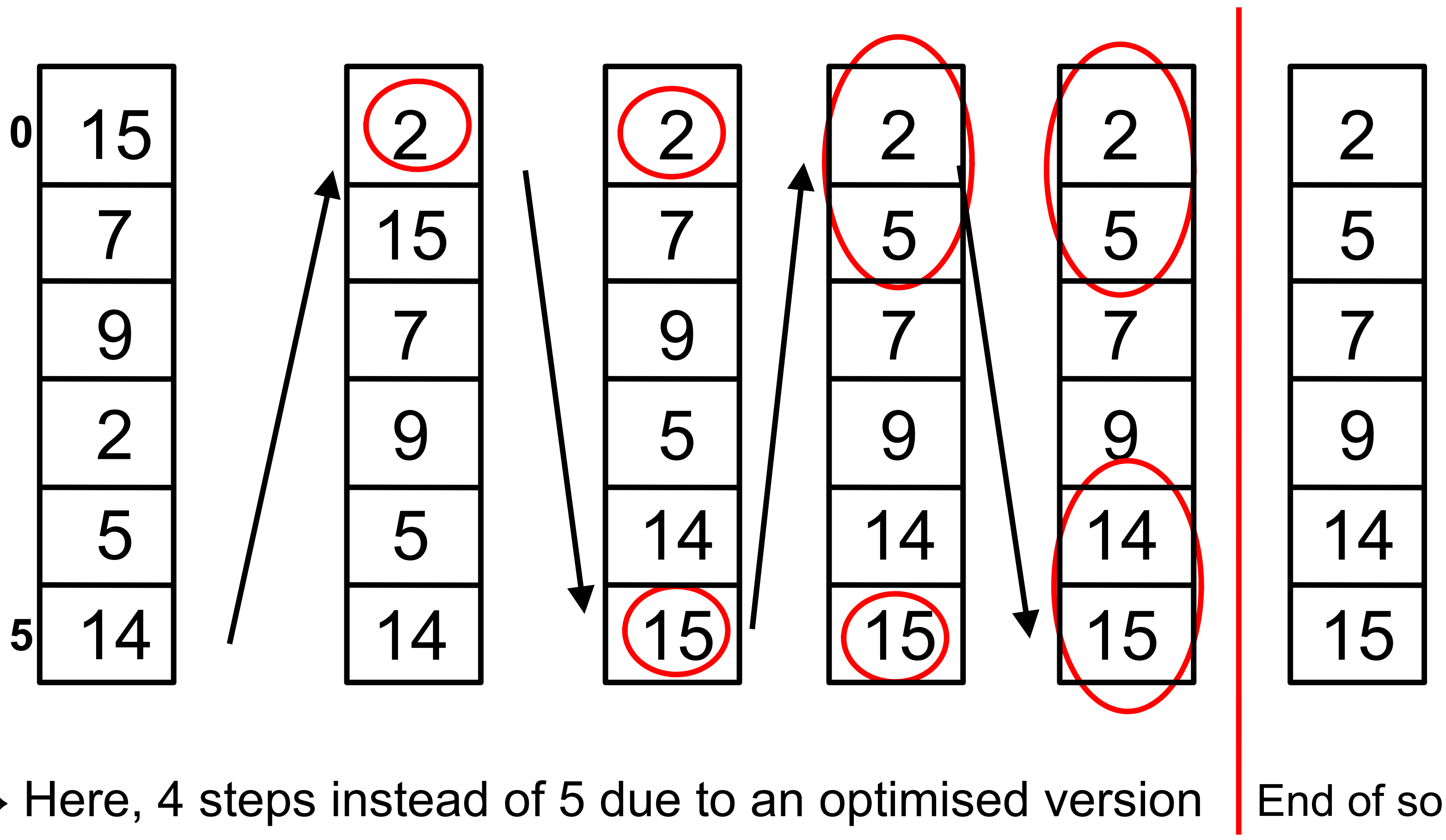
- Shaker sort is an alternative optimised version of Bubble sort, to optimise the case where a large value is at the beginning of the array.

Typical very expensive case by simple Bubble sort:



Principle of Shaker sort: The array is scanned *alternately* from bottom to top *and from top to bottom*.

Shaker sort: illustration



→ Here, 4 steps instead of 5 due to an optimised version

End of sorting

Iterative version of Shaker sort

```
void ShakerSort (t_elt T[], int n) {
  int i=0, j, b=0;              // boolean b = false
  t_elt x;
  while ((i < n-1) && (b == 0)) {
      b = 1;                    // b = true
      if ((i % 2) == 0)                          // bottom-up traversal
         { for (j=(n-1-(i/2)); j>(i/2); j--)
               if (T[j] < T[j-1])
                        { x = T[j]; T[j] = T[j-1]; T[j-1] = x; b = 0; } }
      else
         for (j=i/2+1; j<(n-1-i/2); j++)        // top-down traversal
              if (T[j] > T[j+1])
                       { x = T[j]; T[j] = T[j+1]; T[j+1] = x; b = 0; }
       i++;
   }
}
```

Complexity: O(n^2)
comparisons and swaps

Recursive version of Shaker sort

```
void ShakerSortRec (t_elt T[], int l, int r) {
   int j, b;              // boolean b
   t_elt x;
   if (l < r) {
        b = 1;                         // b = true
        if ((l % 2) == 0)
                { for (j=r; j>l; j--)
                        if (T[j] < T[j-1])    { x = T[j]; T[j] = T[j-1]; T[j-1] = x; b = 0; }
                  if (!b)  ShakerSortRec(T,l+1,r); }
        else
                { for (j=l; j<r; j++)
                        if (T[j+1] < T[j])    { x = T[j]; T[j] = T[j+1]; T[j+1] = x; b = 0; }
                  if (!b)  ShakerSortRec(T,l,r-1); }
   }
}
```

Initial call:
ShakerSortRec(T,0,n-1)

Selection sort

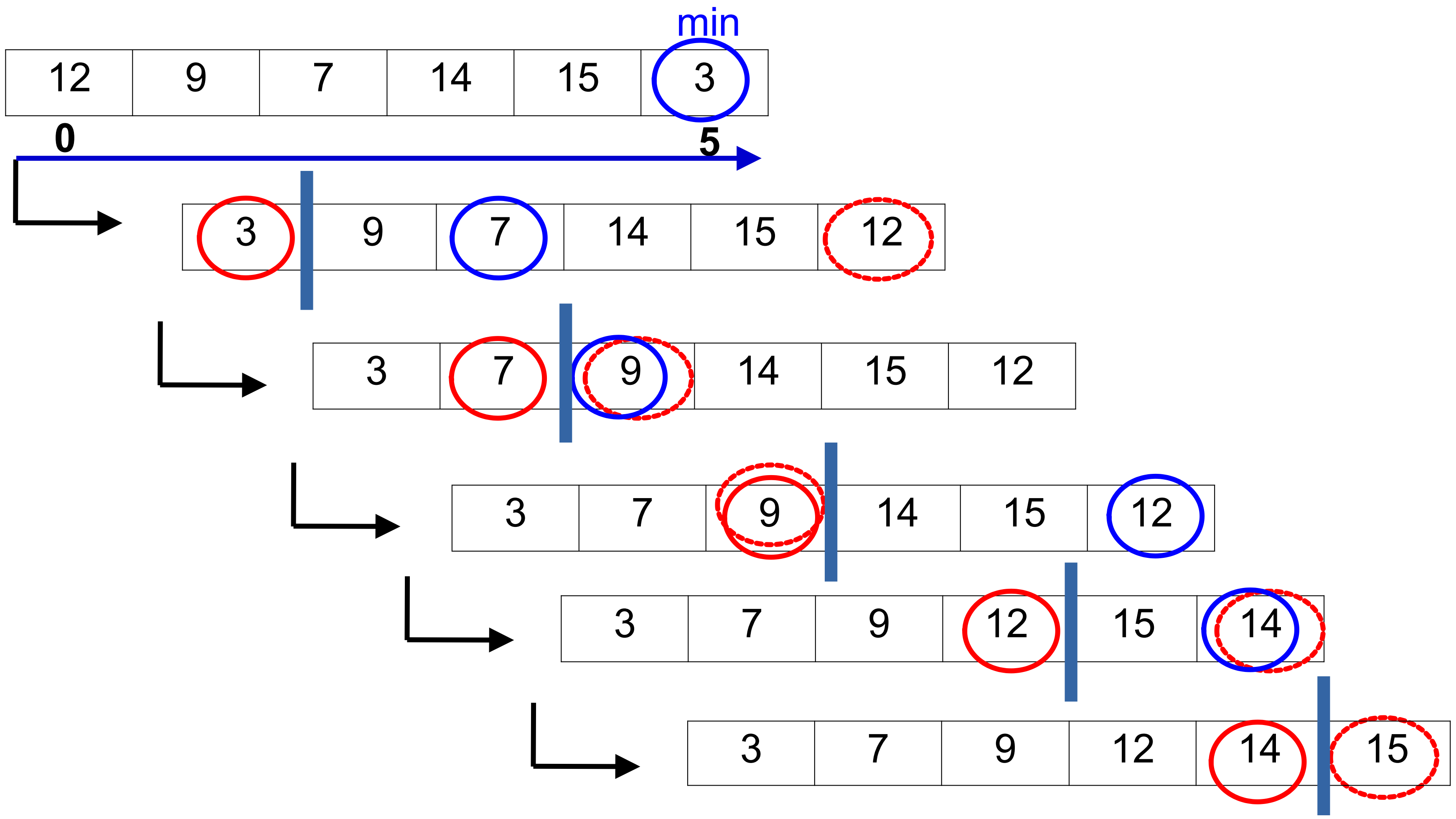
Method:

- Find the minimum element of the array and exchange it with the first element of the array.
- Next, find the minimum element of the rest of the array and exchange it with the second element.
- And so on...

➢ This method aims to reduce the number of swaps compared to Bubble sort.

Projekt je financirala Europska unija iz Europskog Socijalnog fonda

Selection sort: illustration



Iterative version of Selection sort

```
void SelectionSort (t_elt T[], int n) {
   int i, k, min;
   t_elt x;
   for(i=0; i<n-1; i++) {
       min = i;
       for(k=i+1; k<n; k++)
             if (T[k] < T[min])  min = k;
       x = T[min];
       T[min] = T[i];
       T[i] = x;
   }
}
```

Complexity:
O(n) swaps,
$O(n^2)$ comparisons

Projekt je financirala Europska unija iz Europskog Socijalnog fonda

Recursive version of Selection sort

```
void SelectionSortRec (t_elt T[], int l, int r) {
  int min, k;
  t_elt x;
  if (l < r)
    { min = l;
      for(k=l+1; k<=r; k++)
         if (T[k] < T[min])  min = k;
      x = T[min];
      T[min] = T[l];
      T[l] = x;
      SelectSortRec(T,l+1,r);
    }
}
```

Initial call:
SelectionSortRec(T,0,n-1)

Insertion based sorting algorithms

Principle of insertion based sorting algorithms:

- sort the beginning of the array,
- then insert the not yet sorted elements.

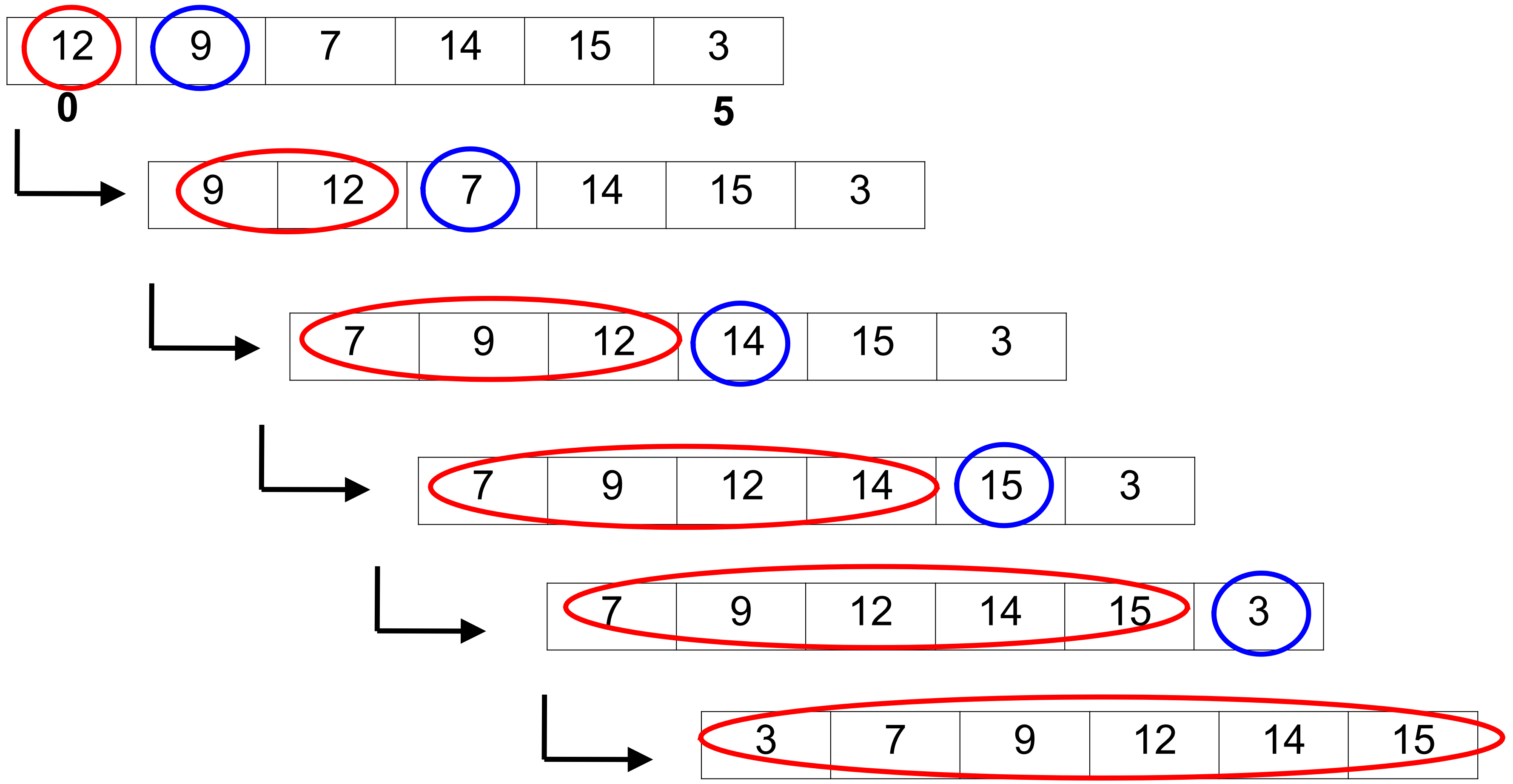
Thus, the first elements of the array are successively sorted.

➔ At step number i, the (i+1)th element is inserted in its place among the i previous elements which are already sorted between them, if necessary.

- ➢ This is the *card player method*.
- ➢ There are *several types of insertion sorts* depending on the method used to find the insertion position: *sequential* or *binary search*.

Projekt je financirala Europska unija iz Europskog Socijalnog fonda

Insertion sort: illustration



Iterative version of (Sequential) Insertion sort

```
void InsertionSort (t_elt T[], int n) {
  int i, k;
  t_elt x;
  for( i=1; i<n; i++)
     if (T[i-1] > T[i])
           { k = i-1;
             x = T[i];
             while ((k >= 0) && (T[k] > x))
                      { T[k+1] = T[k];
                        k = k-1; }
             T[k+1] = x;
           }
}
```

Complexity: $O(n^2)$
comparisons and swaps

Recursive version of (Sequential) Insertion sort

```
void InsertionSortRec (t_elt T[], int i) {
   int k;
   t_elt x;
   if (i > 0) { InsertionSortRec(T,i-1);
                    if (T[i-1] > T[i])
                         { k = i-1;
                           x = T[i];
                           while ((k >= 0) && (T[k] > x))
                                        { T[k+1] = T[k]; k = k-1; }
                           T[k+1] = x;
                         }
               }
}
```

Initial call: InsertionSortRec(T,n-1)

Iterative version of Binary Insertion sort

```
void BinaryInsertionSort (t_elt T[], int n)
{
   int i, j, k;
   t_elt x;
   for( i=1; i<n; i++)
      if (T[i-1] > T[i])
            { k = rank(T, 0, i-1, T[i]);
              x = T[i];
              for( j= i-1 ; j>=k ; j--)
                   { T[j+1] = T[j]; }
              T[k] = x;
            }
}
```

```
int rank (t_elt T[], int l, int r, t_elt x)
{
   int m;
   while ( l != r ) {
      m = (l+r)/2 ;
      if (x < T[m])   { r = m; }
      else { l = m+1; }
   }
   return l;
}
```

Complexity: O(n^2) swaps and O($n \log_2(n)$) comparisons

Recursive version of Binary Insertion sort

```
void BinaryInsertionSortRec (t_elt T[n], int i) {
   int j, k;
   t_elt x;
   if ( i>0)
    { BinaryInsertionSortRec(T,i-1);
      if (T[i-1] > T[i])
            { k = rankRec(T, 0, i-1, T[i]);
              x = T[i];
              for( j= i-1 ; j>=k ; j--)
                  { T[j+1] = T[j]; }
              T[k] = x;
            }
    }
}
```

```
int rankRec (t_elt T[], int l, int r, t_elt x)
{
   int m;
   if ( l == r ) { return l; }
   else { m = (l+r)/2 ;
            if (x < T[m])
               { return rankRec(T,l,m,x); }
            else
               { return rankRec(T,m+1,r,x); }
         }
}
```

Initial call:
BinaryInsertionSortRec(T,n-1)

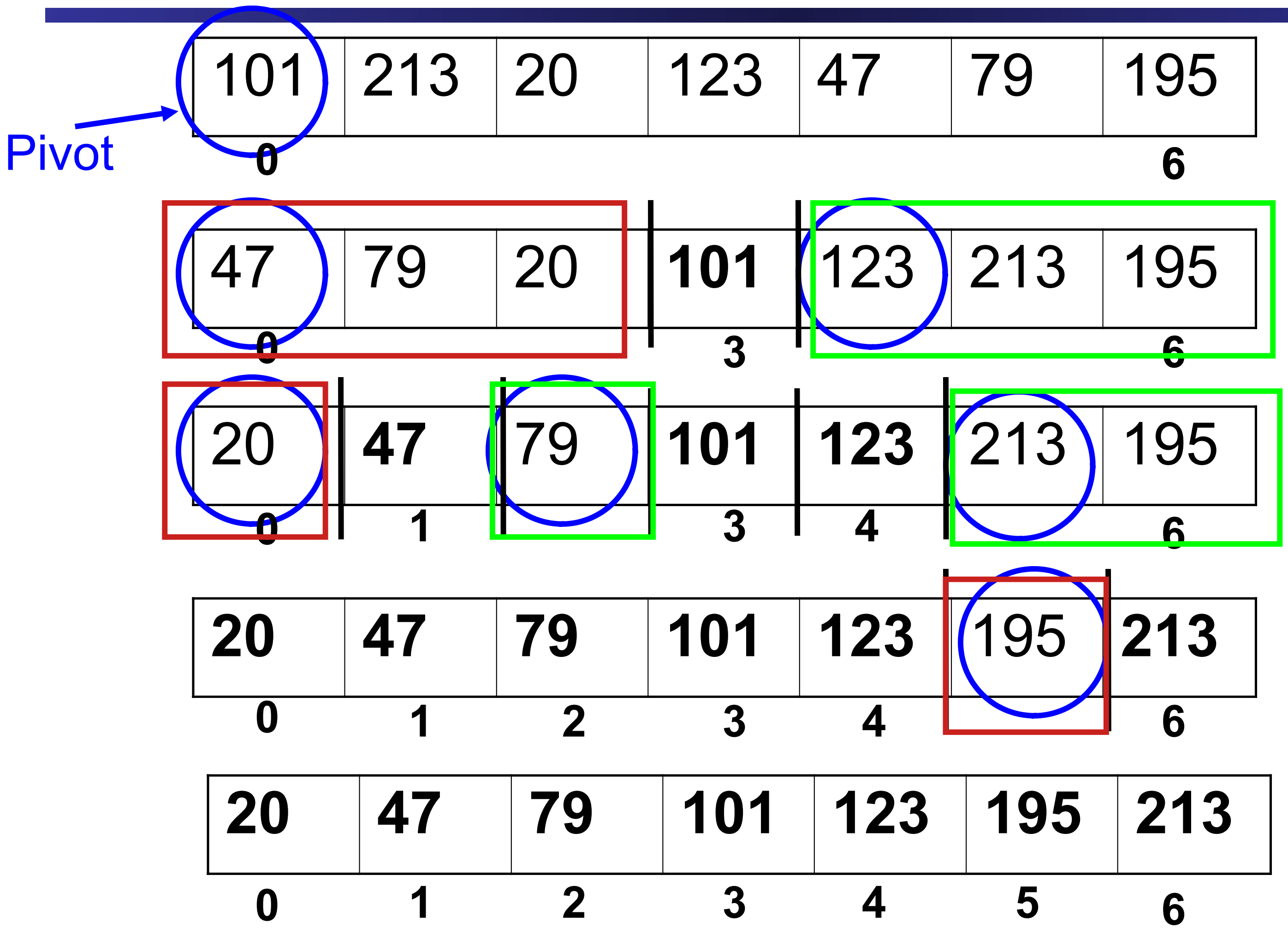
Quick sort

Quick sort is a *dichotomous recursive* algorithm and a *generalized selection based* sorting algorithm: an element is put in its definitive place, but which is not necessarily the first one.

Method:

1. Choose a pivot element,
2. divide the array to be sorted into two sub-arrays T1 and T2 such that the elements of T1 are less than or equal to the pivot, and the elements of T2 are greater than the pivot,
3. sort each of the sub-arrays T1 and T2 according to the same method (restart from step 1 until the size of the processed (sub-)array is equal to 1),
4. finally, replace the pivot between the two sorted sub-arrays.

Quick sort: illustration



Here, the pivot is chosen as the first element of the array.

Quick sort algorithm

```
void QuickSort (t_elt T[], int l, int r) {
  int k;
  if (l < r)
    { k = partition(T,l,r);      // Partition T in T1 and T2 :
      QuickSort(T,l,k-1);        //  - T1 : l → k, elts ≤ pivot,
      QuickSort(T,k+1,r);        //          and T1[k] = pivot
    }                            //  - T2 : k+1 → r, elts > pivot
}
```

Initial call: QuickSort(T,0,n-1)

Quick sort: function *partition*

```
int partition (t_elt T[], int l, int r) {
   int i = l+1, k = r;                          // T[l] is the pivot
   t_elt x;
   while (i <= k) {
        while (T[k]  > T[l])   k = k-1;
        while (i <= r && T[i] <= T[l])   i = i+1;
        if (i < k)  { x = T[i]; T[i] = T[k]; T[k] = x;
                      i = i+1;
                      k = k-1;}
   }
   x = T[l];
   T[l] = T[k];
   T[k] = x;
   return k;
}
```

Complexity: O(n)
comparisons and swaps

Complexity of the Quick sort algorithm

- **Complexity of QuickSort in number of recursive calls:**

  the calls to QuickSort represent a binary tree:

  – If the tree is balanced, the number of recursive calls is $O(\log_2(n))$

  – If the tree is totally unbalanced, the number of recursive calls is $O(n)$.

  ➔ Hence the importance of the choice of the pivot to try to obtain each time two sub-arrays of equivalent size. The choice of this pivot must however remain simple (random, median value...).

- **Overall complexity of QuickSort in number of comparisons and swaps:**

  ➔ $O(n \log_2(n))$ if the tree is balanced and $O(n^2)$ at worst.

Heap Sort

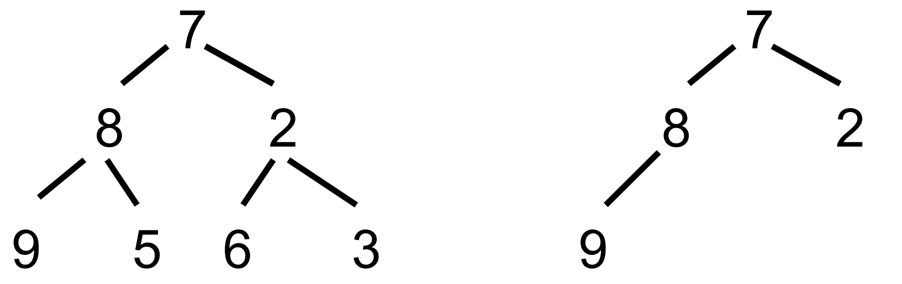
Method based on a specific data structure, called heap, where are kept the result of the comparisons made at each step.

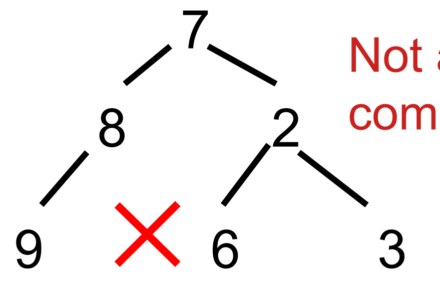
- **Definition:** heap

  A heap is a *partially ordered* (almost) complete binary tree.

- **Definition:** (almost) complete binary tree

  It is a binary tree with all levels complete, except possibly the last one, in which case the missing leaves are the rightmost ones.





Not almost complete

Properties of a Heap

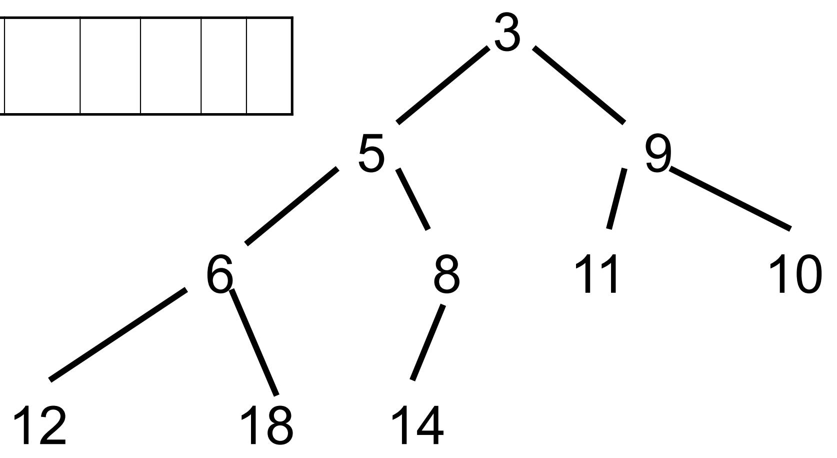
- **Property:** *heap partial order*

  In a heap, the value (key) of each node is less than or equal to the values of its children. The elements of the tree must therefore be part of a set with a total order.

  → The minimum element of the tree is always at its root.

3	5	9	6	8	11	10	12	18	14					

Example of a heap:



Typical representation of a Heap

3	5	9	6	8	11	10	12	18	14					

- **Definition:** heap

  A heap is a *partially ordered* (almost) complete binary tree. represented as an array T[0..max] such that:

  – The root element is in T[0]

  – T[(i-1)/2] is the parent node of T[i], ∀ i>0

  – T[2*i +1] and T[2*i +2] are the left and right children, if they exist, of T[i], ∀ i ≥ 0

  – If p is the size of the tree (≤ max+1) and p=2*(i+1), T[i] has a single (left) child T[p-1]

  – If i ≥ p/2 then T[i] is a leaf.

Projekt je financirala Europska unija iz Europskog Socijalnog fonda

Heap sort: principle

- Heap sort is done in two steps:
  1. Construction of a heap by successively adding to it all the elements of the array to be sorted
  2. Extraction of the minimum of the heap (then reorganisation of the heap) until it is empty

- Notes:
  - All is done in the array (no extra memory needed)
  - At the end of the algorithm, T is sorted in descending order ("backwards")

- Complexity:
  - $O(n \log_2(n))$ comparisons and swaps

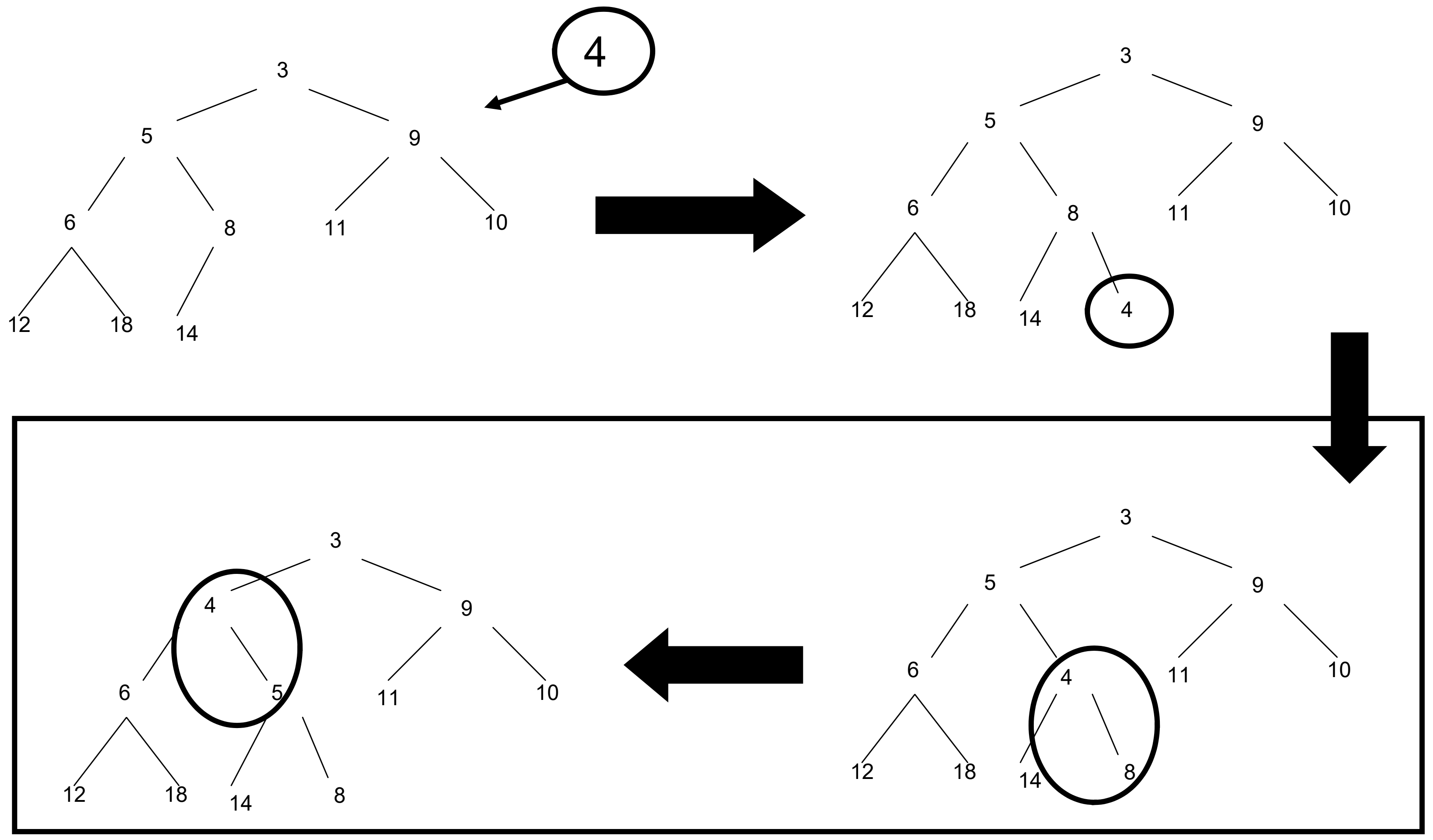
Operations on the Heap data structure: adding a new element to a heap

- **Method:**
  1. Add a leaf to the last level of the tree,
  2. put the new element in that leaf (attention, the tree is no longer necessarily partially ordered),
  3. *reorganisation: the new value is moved up the tree (by exchange with its parent) until it is in its final place.*

- **Complexity:**

  O($\log_2$(p)) if p is the size of the treee (heap).

Adding to a heap: illustration



Reorganisation after adding of an element to a heap

The new element is in T[p]

```
void BottomUpReordering (t_elt T[], int p) {
   int i = p;                   // p = size of the heap
   t_elt x;
   while ((i > 0) && (T[i] < T[(i-1)/2]))
      { x = T[i];
        T[i] = T[(i-1)/2];
        T[(i-1)/2] = x;
         i = (i-1)/2; }
}
```

Complexity: $O(\log_2(p))$ comparisons and swaps

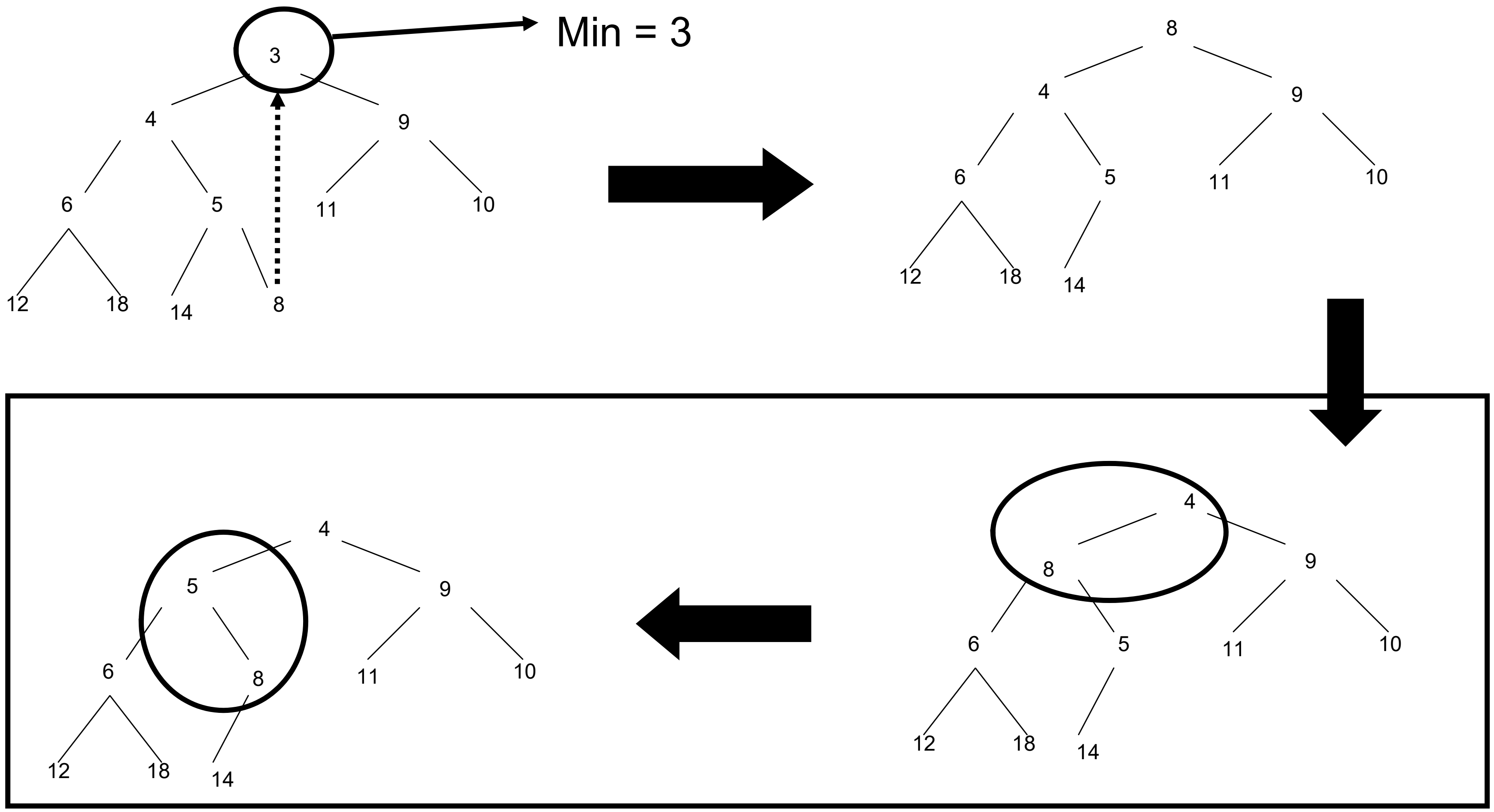
Operations on the Heap data structure: removing the minimum element of a heap (the root)

- **Method:**
  1. Extract the value of the root,
  2. replace the value of the root by the value of the last leaf (attention, the tree is no longer necessarily partially ordered),
  3. *reorganisation: this value is moved down the tree (by swapping with its smallest child) until it is in its final place.*

- **Complexity:**

  $O(\log_2(p))$ if p is the size of the heap.

Deletion from a heap: illustration



Reorganisation after deletion of the minimum element of a heap

```
void TopDownReordering (t_elt T[], int p) {
   int i = 0, c;              // p = size of the remaining heap
   t_elt x;
   while (i < p/2)
      {   if (p == (2*(i+1)) || (T[2*i + 1] < T[2*i +2]))
                c = 2*i + 1;
          else  c = 2*i +2;
          if (T[i] > T[c])
                { x = T[i];
                  T[i] = T[c];
                  T[c] = x;
                   i = c; }
          else  i = p;
      }
}
```

T[0] = element to be moved

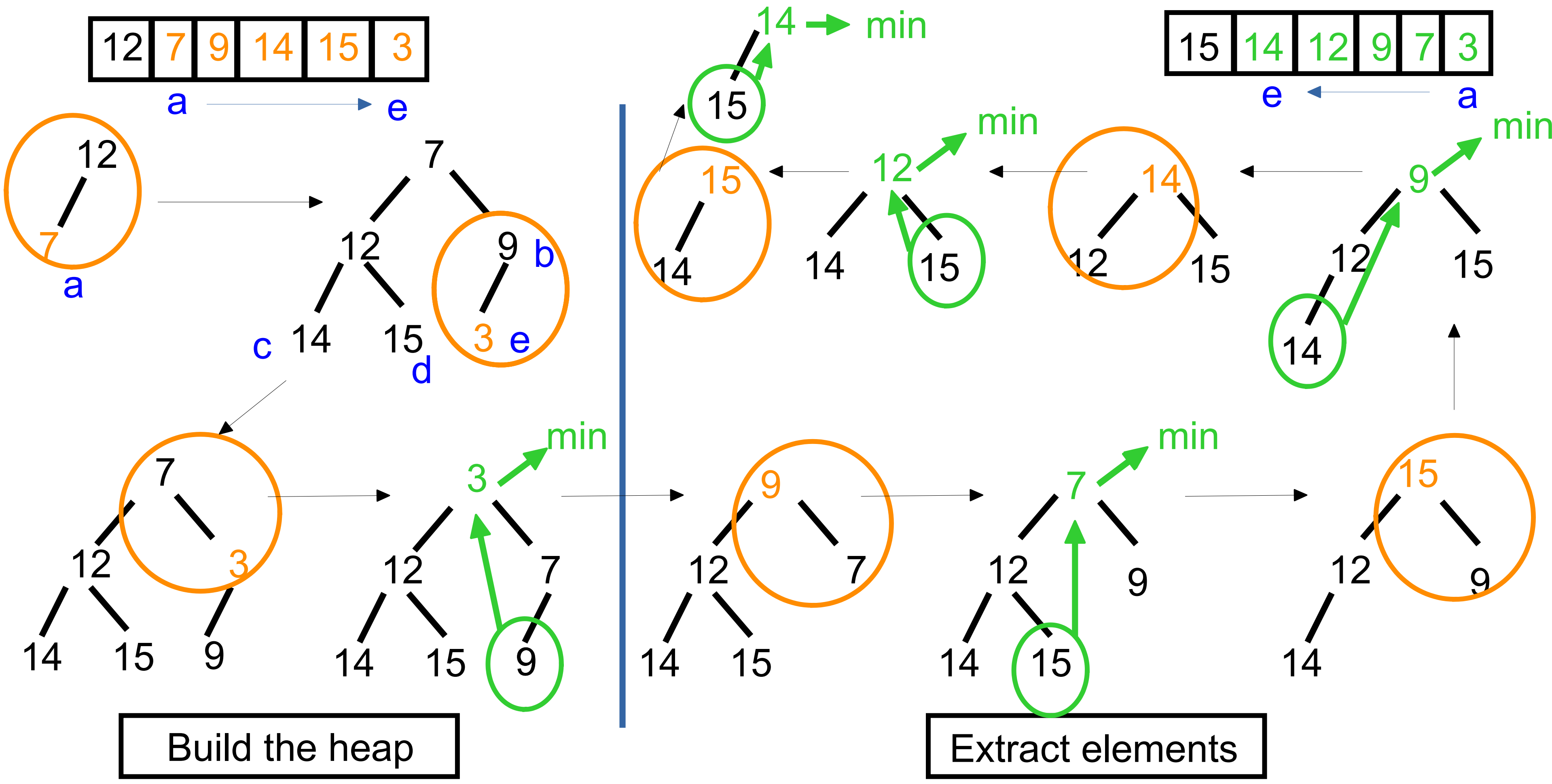
Complexity: $O(\log_2(p))$ comparisons and swaps

Heap sort algorithm

```
void HeapSort (t_elt T[], int n) {
   int p;
   t_elt min;
   for(p = 1; p < n; p++)
      BottomUpReordering(T,p);
   for(p = n-1; p > 0; p--)
     { min = T[0];
       T[0] = T[p];
       TopDownReordering(T,p);
       T[p] = min; }
}
```

Complexity: $O(n \log_2(n))$ comparisons and swaps

Heap sort: illustration



V.A. Nicolas

Projekt je financirala Europska unija iz Europskog Socijalnog fonda

Overview of main sorting algorithms

➢ Internal sorting algorithms	comparisons	swaps	space
◆ Basic selection based algorithms			
• Bubble sort (and Shaker sort)	$O(n^2)$	$O(n^2)$	**O(1)**
• Selection sort	$O(n^2)$	**O(n)**	**O(1)**
◆ Basic insertion based algorithms			
• Sequential insertion sort	$O(n^2)$	$O(n^2)$	**O(1)**
• Binary insertion sort	$\mathbf{O(n \log_2(n))}$	$O(n^2)$	**O(1)**
◆ Advanced selection based algorithms			
• Quick sort (or Hoare sort)	$O(n^2)$	$O(n^2)$	$O(\log_2(n))$
• Heap sort	$\mathbf{O(n \log_2(n))}$	$\mathbf{O(n \log_2(n))}$	**O(1)**
➢ External sorting algorithm: *Merge sort*	$\mathbf{O(n \log_2(n))}$	$\mathbf{O(n \log_2(n))}$	O(n)

Conclusion

- *Optimal complexity* in number of comparisons: *$O(n \log_2 n)$*
- *Stable algorithms* except Selection sort, Quick sort and Heap sort
- Small size arrays: basic algorithm → *Insertion sort* (also well suited when the array is already partially sorted)
- Bigger size arrays: advanced algorithm → *Heap sort, Quick sort* ; Quick sort is in general faster than Heap sort (average complexity is $O(n \log_2 n)$)
- Very large arrays: external algorithm → *Merge sort* (parallelizable…)

- Many other (specific) sorting algorithms. Here were presented comparison sorts, but there is also non comparison sorts (e.g. counting sorts...)

- **Laboratory session: Merge sort and experimental evaluation of sorting algorithm complexities**

Merge sort

Merge sort is an *external sorting* algorithm.

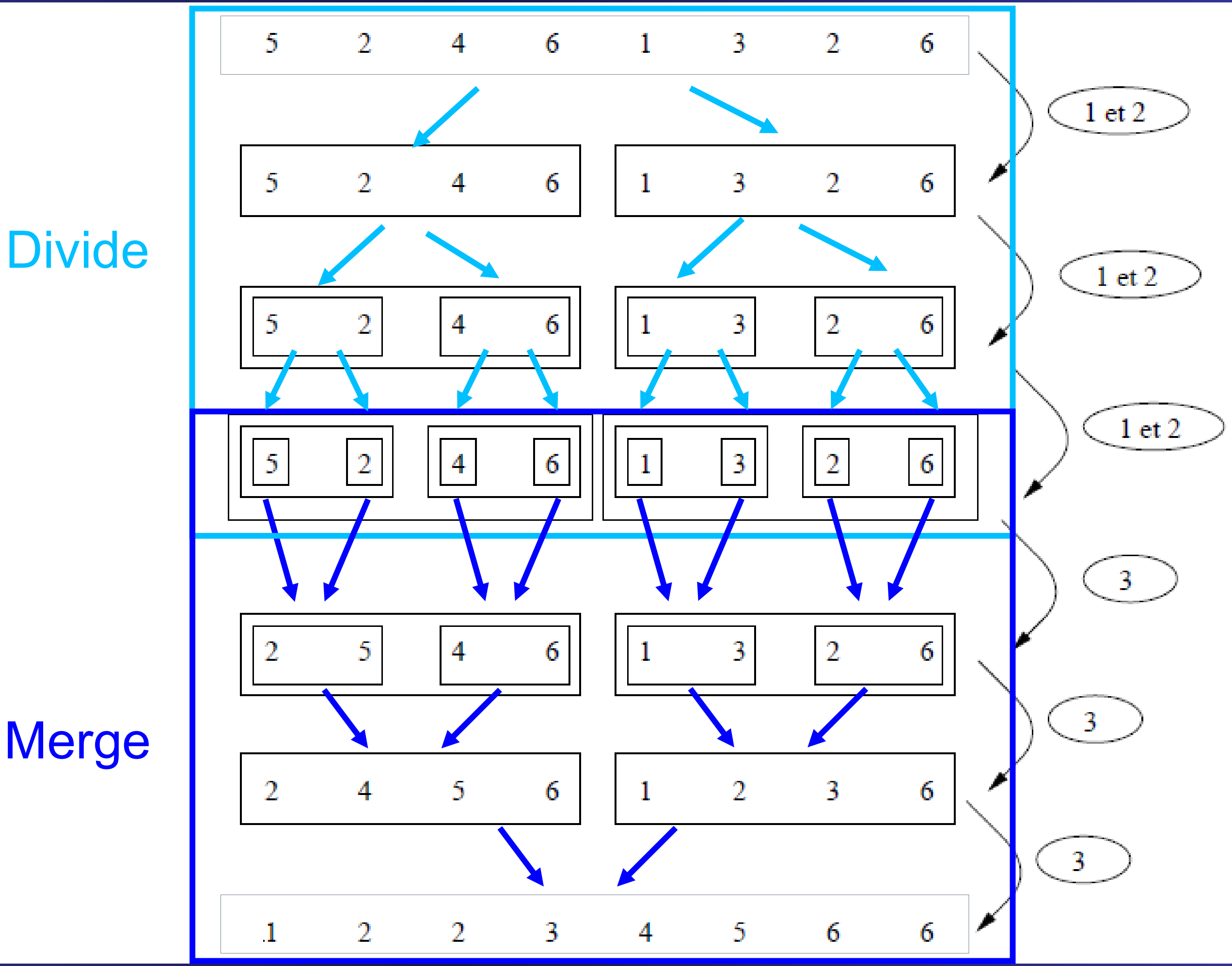
It is a *dichotomous* algorithm aimed at reducing the number of data moves in order to limit the memory loadings.

Method: To sort an array of size n:

1. Divide the array into two sub-arrays of rough size n/2 (first and second half),
2. sort the two sub-arrays independently (by recursive calls to Merge sort),
3. merge the two sorted sub-arrays.

Merge sort is not an « in place » algorithm: its space complexity is O(n), however comparisons and swaps are optimal in $O(n \log_2(n))$.

Merge sort: illustration



Université de Bretagne Occidentale
Projekt je financirala Europska unija iz Europskog Socijalnog fonda