Sorting

Sorting

- The process of putting elements in a collection in some kind of order

- E.g.:
  - numbers sorted numerically
  - words sorted alphabetically

- Can be ascending or descending order
  - We will work with **ascending order as the default**
  - ....the algorithm for descending would be very similar

- There are many, many, (way too many) sorting algorithms out there.
  - Utility varies depending on size of problem, problem context, etc.

- Typical operations required for sorting would be:
  - comparing
  - changing (or exchanging) and element's position
  - in some cases, copying parts of a list back and forth

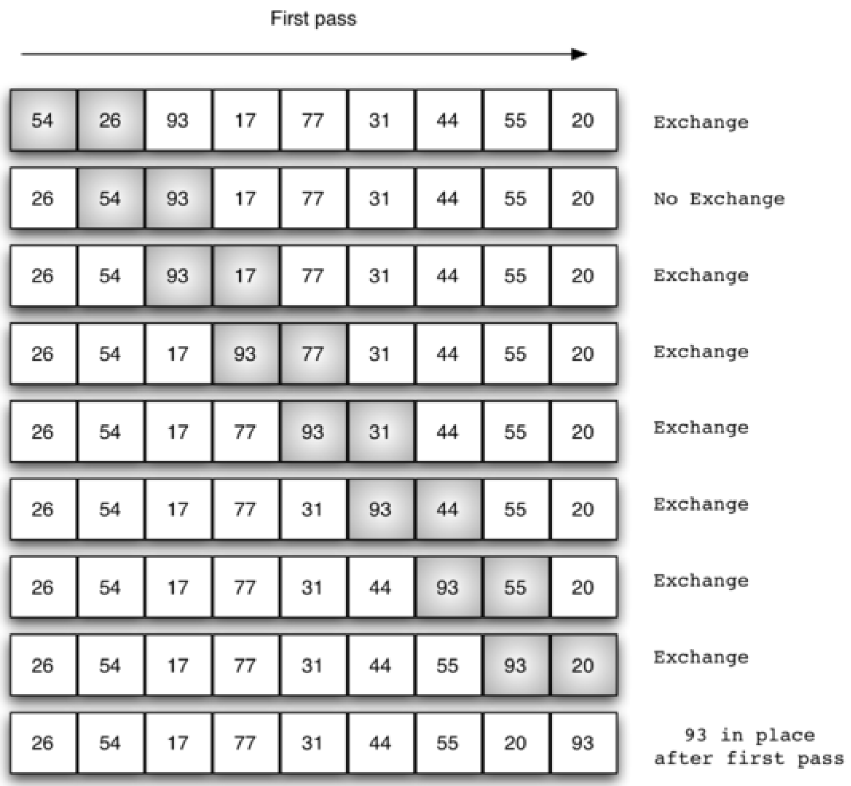
Bubble Sort

Bubble Sort

- We have done this earlier:
  - Multiple passes through a list
  - Compare adjacent items, exchange if out order.
  - Larger items “bubble” to the right (in case of ascending order)

Bubble Sort

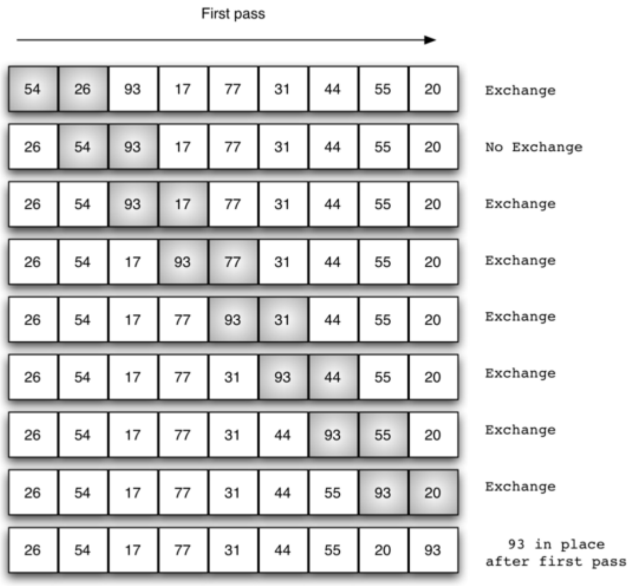
- We have done this earlier:
  - Multiple passes through a list
  - Compare adjacent items, exchange if out order.
  - Larger items “bubble” to the right (in case of ascending order)



From:
https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheBubbleSort.html

Bubble Sort

- We have done this earlier:
  - Multiple passes through a list
  - Compare adjacent items, exchange if out order.
  - Larger items "bubble" to the right (in case of ascending order)

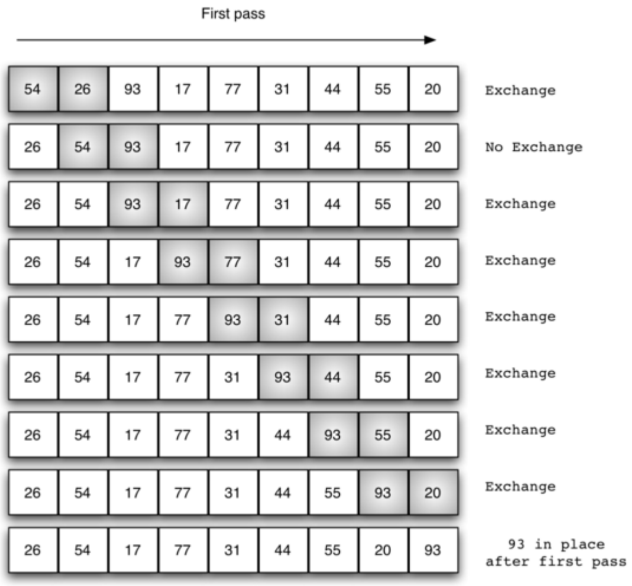



```python
#%% Bubble sort
def bubble_sort(A):
    n = len(A)
    for outer in range(n-1,0,-1):
        for i in range(outer):
            if (A[i] > A[i+1]):
                temp = A[i]
                A[i] = A[i+1]
                A[i+1] = temp
```

From:
https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheBubbleSort.html

Bubble Sort

- We have done this earlier:
  - Multiple passes through a list
  - Compare adjacent items, exchange if out order.
  - Larger items “bubble” to the right (in case of ascending order)



[Visualize it here]

```python
#%% Bubble sort
def bubble_sort(A):
    n = len(A)
    for outer in range(n-1,0,-1):
        for i in range(outer):
            if (A[i] > A[i+1]):
                temp = A[i]
                A[i] = A[i+1]
                A[i+1] = temp
```

This is **O(n^2)** complexity
(double nested loop, each in the order of n iterations)

From:
https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheBubbleSort.html

Insertion Sort

Insertion Sort

- Quite likely the go-to way most people would sort their playing cards in a game
- Create a **sorted sub-list**
  - start with one element in that sorted sub-list (single element list is always sorted!)
  - one element from the un-sorted portion of the list
  - **insert it at the correct position in the sorted sub-list**
  - keep doing this until you reach the last element

Get your deck of cards!

- Or, go to: https://deck.of.cards/
- Note:
  - You have to loop through the entire list, one iteration for each item of the list (leaving the first element, which is already sorted) → n operations
  - For each item of the list, you need to place it *at the correct place in the ordered sub-list*, which means *another loop to find its correct place in the sub-list.*
    - This can take up to n operations too in the worst case → ~n operations
  - This is a $O(n^2)$ operation too, like bubble sort

Insertion Sort - Psuedocode

```
INSERTION-SORT(A)
```

Insertion Sort - Psuedocode

```
INSERTION-SORT(A)
   #A[0] already sorted, start outer loop from 1
   for outer = 1 to len(A)-1:  #outerloop goes through all elements (other than first one)
       key = A[outer]  #this is the "key" element that will be moved to its correct position in the sorted sublist
                       #we move this into a temp variable. This creates a vacant slot at A[outer] which we
                       #can move to keep shifting elements right until we find the correct location for  "key"
       inner = outer   #the sorted sub-list ends at the outer index. we set inner index to that value and then traverse down to zero

       #inner loop will work back from the right-edge of the sorted sub-list, down to zero, shifting the inner index elements right
      #until a value smaller than key is encountered. At that point, the key will be inserted
      #recall that A[outer] = A[inner] has been stored as "key", so we can overwrite for our "shift right" operation
       while inner > 0 and A[inner-1] > key:
           shift A[inner-1] to A[inner]
           inner -= 1
       #now, inner points to where the key should go
       A[inner] = key
```

Insertion sort complexity

```
INSERTION-SORT(A)
    #A[0] already sorted, start outer loop from 1
    for outer = 1 to len(A)-1:  #outerloop goes through all elements (other than first one)
        key = A[outer]  #this is the element that will be moved to its correct position in the sorted sublist
        inner = outer-1  #the sorted sub-list ends at the outer index-1. we then traverse down to zero
        #inner loop will work back from the right-edge of the sorted sub-list, down to zero, shifting the inner index elements right
       #until a value smaller than key is encountered. At that point, the key will be inserted
        while inner > 0 and A[inner-1] > key:
            shift A[inner-1] to A[inner]
            inner -= 1
        #now, inner points to where the key should go
        A[inner] = key
```

$O(n)$

$O(1)$

$O(1)$

$\sim O(n^2)$ { goes upto n, so we simplify to O(n), which is then multiplied by parent loops complexity to give $O(n^2)$ }

$\sim O(n^2)$

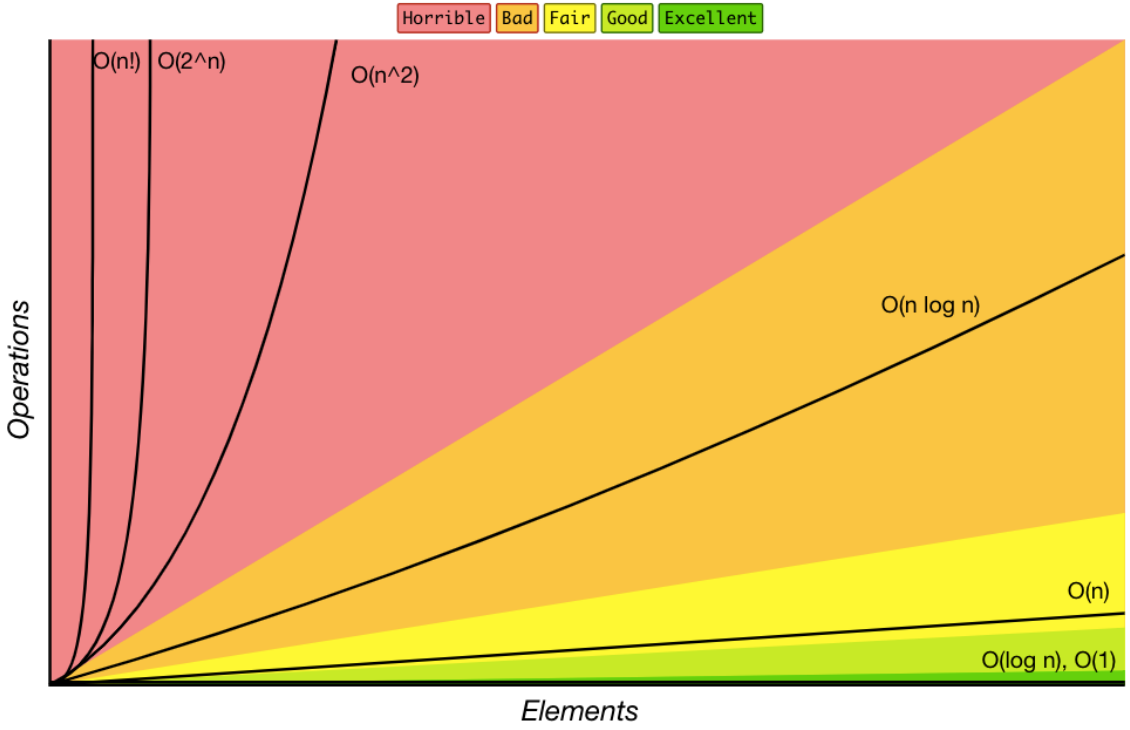
$\sim O(n^2)$

$O(n)$

$$T(n) = O(n) + O(1) + O(1) + O(n^2) + O(n^2) + O(n^2) + O(1)$$
$$= O(n^2)$$

Note

- Both Bubble Sort and Insertion sort are: $\mathbf{O(n^2)}$



Breaking the $O(n^2)$ barrier

DIVIDE AND CONQUER!

Merge Sort

MERGE-SORT ILLUSTRATION

[Deck of cards]

MERGE-SORT: ILLUSTRATION

https://visualgo.net/

MERGE-SORT

- Efficient divide-and-conquer sorting algorithm
- Intuitively it operates as follows
  - Divide the n-element array to be sorted into two subarrays of n/2 elements each
  - Conquer: Sort the two subarrays recursively using MERGE-SORT
  - Combine: Merge the two sorted subarrays to produce the sorted answer
- Key Operation: How to “merge” two sorted arrays
  - Scan L (left) and R (right)
  - At each iteration, copy the minimum to A
  - Advance the iteration on L (or R) only if the minimum is picked from L (or R)
- We will use this idea plus some some improvements to define algorithm MERGE

MERGE vs MERGE-SORT

- **MERGE is an "interesting" algorithm in its own right**
  - Given two sorted lists L and R, MERGE merges them into a single list such that the merge list is also sorted.

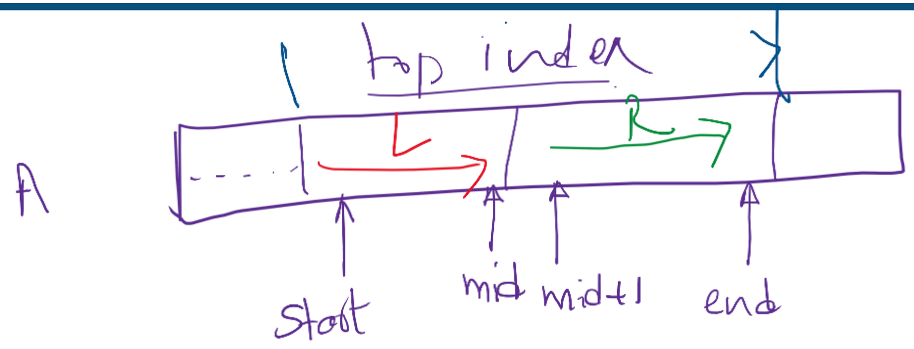
- **It is useful/easier to think about MERGE separately first**

- **Once we have defined the MERGE operation correctly, then we can just use it in a MERGE–SORT algorithm**
  - Breaking down an algorithm into smaller algorithms is a very important design pattern that you should be comfortable with. Most algorithms in real application are built in this hierarchical manner.

First, we MERGE

```
MERGE(A,start, mid, end)
```

First, we MERGE



- L : start → mid   R : mid+1 → end
- ti : start → end
  - if A[L] < A[R]
    A[ti] = A[L]
    L++
  else A[ti] = A[R]
    R++.

```
MERGE(A,start,mid,end)
   LEFT = A[start:mid]
   RIGHT = A[mid+1:end]
   ti = start
   li = 0
   ri = 0
   for ti from start to end
        if(LEFT[li] < RIGHT[ri])
             A[ti] = LEFT[li]
             li += 1
         else
             A[ti] = RIGHT[ri]
             ri += 1
```

Merge has a problem

- Image merging these to left and right sub-arrays

LEFT = [1,3,4], RIGHT = [6,8,9]

```
MERGE(A,start,mid,end)
  LEFT = A[start:mid]
  RIGHT = A[mid+1:end]
  ti = start
  li = 0
  ri = 0
  for ti from start to end
      if(LEFT[li] < RIGHT[ri])
           A[ti] = LEFT[li]
           li += 1
       else
           A[ti] = RIGHT[ri]
           ri += 1
```

Merge has a problem

- Image merging these to left and right sub-arrays

LEFT = [1,3,4], RIGHT = [6,8,9]

- You will consume all items from LEFT first; but how do you know you've reached the end of LEFT and stop looking there?
  - the "top index" (ti) is only keeping track of the *total* size of the output (merged) array
  - we could check left index and right index too separately against the size of left/right sub-lists
  - but another, clean and elegant way to do it is **ensure whichever of LEFT and RIGHT is completely consumed first, is never picked again**

```
MERGE(A,start,mid,end)
  LEFT = A[start:mid]
  RIGHT = A[mid+1:end]
  ti = start
  li = 0
  ri = 0
  for ti from start to end
      if(LEFT[li] < RIGHT[ri])
            A[ti] = LEFT[li]
            li += 1
       else
            A[ti] = RIGHT[ri]
            ri += 1
```

Merge has a problem

- Image merging these to left and right sub-arrays

LEFT = [1,3,4, ∞], RIGHT = [6,8,9, ∞]

- You will consume all items from LEFT first; but how do you know you've reached the end of LEFT and stop looking there?
  - the "top index" (ti) is only keeping track of the total size of the output (merged) array
  - we could check left index and right index too separately
  - but another, clean and elegant way to do it is **ensure whichever of LEFT and RIGHT is completely consumed first, is never picked again**
  - we put **INFINITY** at the end of both LEFT and RIGHT!
    - Since ∞ is never less than an actual number, once you get to the value ∞ either in LEFT or RIGHT, you will not pick them again

```
MERGE(A,start,mid,end)
  LEFT = A[start:mid]
  RIGHT = A[mid+1:end]
  ti = start
  li = 0
  ri = 0
  for ti from start to end
      if(LEFT[li] < RIGHT[ri])
           A[ti] = LEFT[li]
           li += 1
       else
           A[ti] = RIGHT[ri]
           ri += 1
```

The value INFINITY (∞) in Python?

- We can't actually have a ∞ as a value in a Python list.

- We use next best thing: the largest possible integer representable in Python

- You can get it like so: sys.maxsize

- These "maximal" values (∞ or its equivalent) when used in algorithms like we did, are called sentinels
  - which is a really cool name if you ask me

MERGE

(Required for MERGE-SORT)

Now adding sentinels

```
MERGE(A,start,mid,end)
  LEFT = A[start:mid]
  RIGHT = A[mid+1:end]

  LEFT.append(INF)
  RIGHT.append(INF)

  ti = start
  li = 0
  ri = 0

  for ti from start to end
      if(LEFT[li] < RIGHT[ri])
          A[ti] = LEFT[li]
          li += 1
       else
          A[ti] = RIGHT[ri]
          ri += 1
```

Sentinels: https://en.wikipedia.org/wiki/Sentinel_value

MERGE

(Required for MERGE-SORT)

- Input: Array **A** and three indexes **start**, **mid**, **end** for **A** such that **start ≤ mid < end**
  - Subarrays LEFT = A[start...mid] and RIGHT = A[mid+1..end] are assumed sorted
- Output: sorted subarray **A[start..end]**
- We make copies of the two subarrays as **LEFT** and **RIGHT**.
- They are "merged" in a sorted fashion back into **A**
- We use **sentinels** (∞) to avoid checking at every step if **LEFT** or **REFT** have been entirely scanned.

```
MERGE(A,start,mid,end)
   LEFT = A[start:mid]
   RIGHT = A[mid+1:end]
   LEFT.append(∞)
   RIGHT.append(∞)
   ti = start
   li = 0
   ri = 0

   for ti from start to end
         if(A[li] < A[ri])
                A[ti] = A[li]
                li += 1
          else
                A[ti] = A[ri]
                ri += 1
```


Sentinels: https://en.wikipedia.org/wiki/Sentinel_value

Example execution of MERGE(A,0,2,5)

	start		mid			end
A	1	3	4	2	5	6

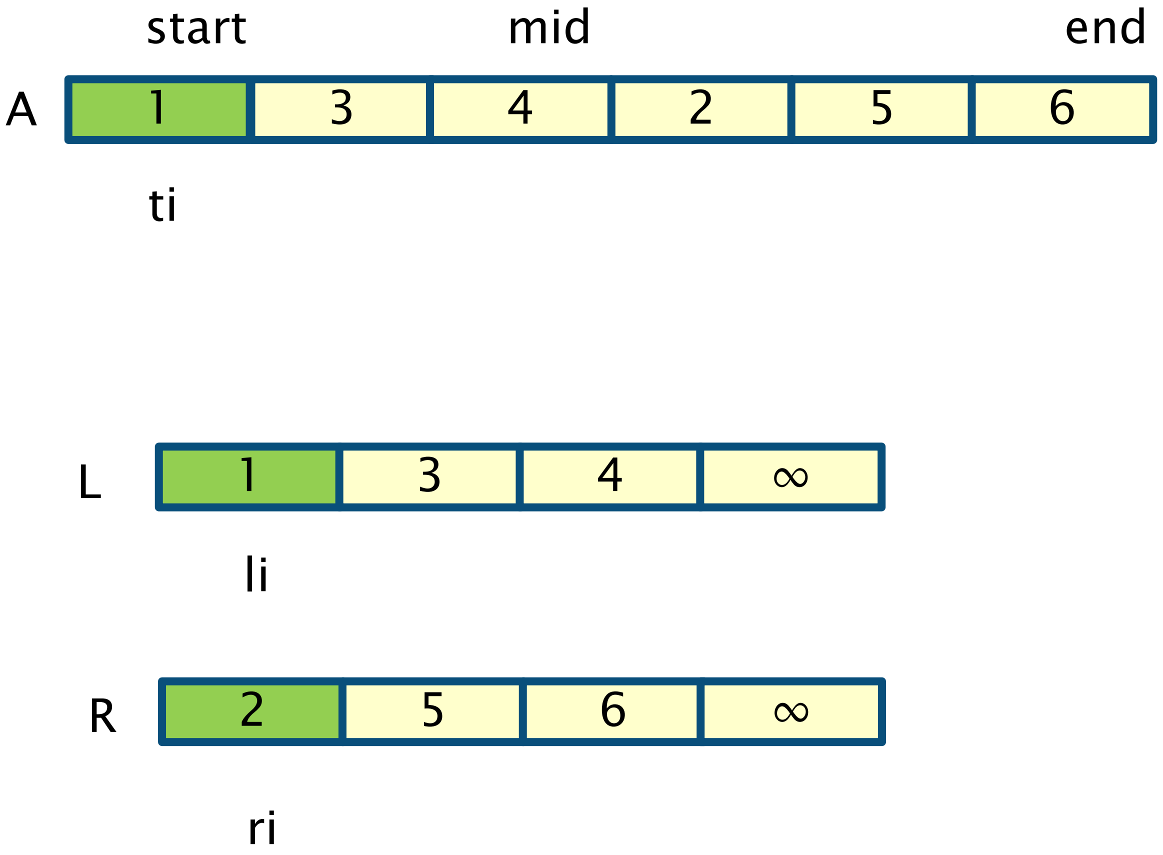
```
MERGE(A,start,mid,end)
  LEFT = A[start:mid]
  RIGHT = A[mid+1:end]

  LEFT.append(INF)
  RIGHT.append(INF)

  ti = start
  li = 0
  ri = 0

  for ti from start to end
      if(LEFT[li] < RIGHT[ri])
          A[ti] = LEFT[li]
          li += 1
      else
          A[ti] = RIGHT[ri]
          ri += 1
```

Example execution of MERGE(A,0,2,5)



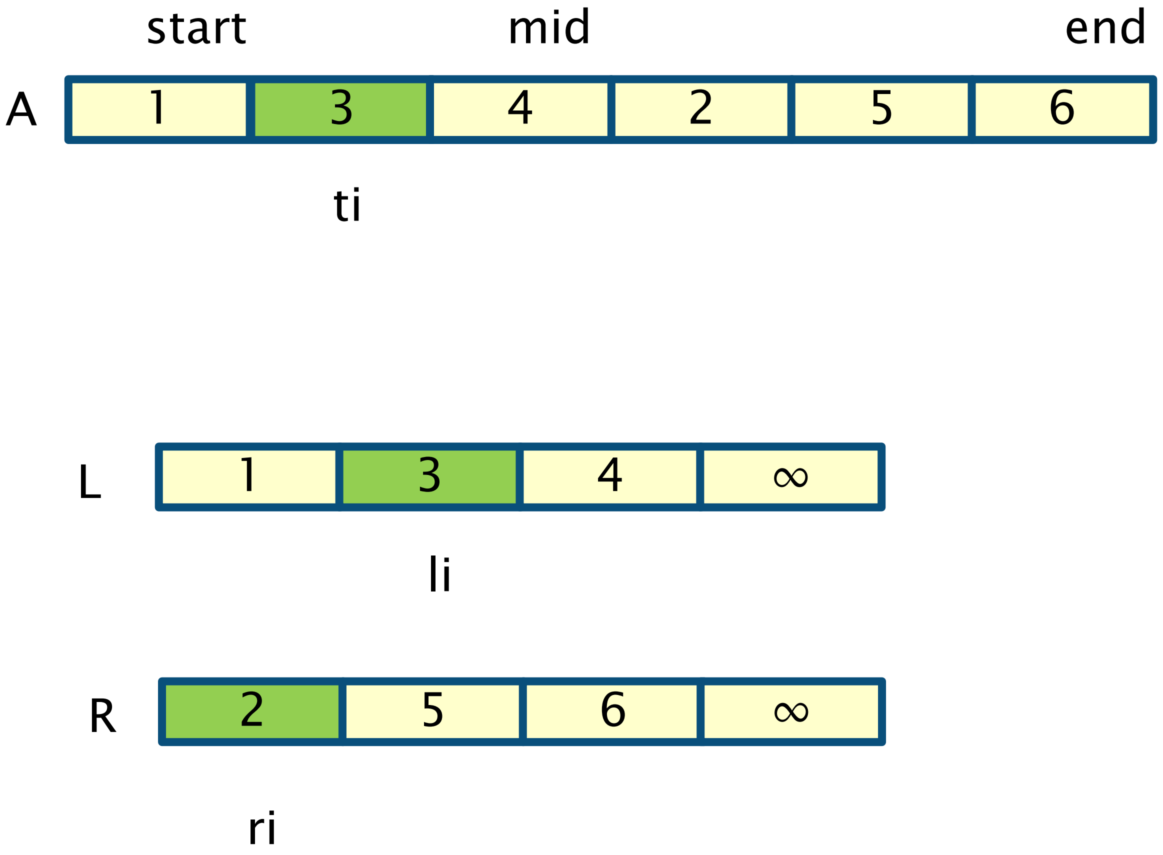

```
MERGE(A,start,mid,end)
   LEFT = A[start:mid]
   RIGHT = A[mid+1:end]

   LEFT.append(INF)
   RIGHT.append(INF)

   ti = start
   li = 0
   ri = 0

   for ti from start to end
       if(LEFT[li] < RIGHT[ri])
             A[ti] = LEFT[li]
             li += 1
        else
             A[ti] = RIGHT[ri]
             ri += 1
```

Example execution of MERGE(A,0,2,5)



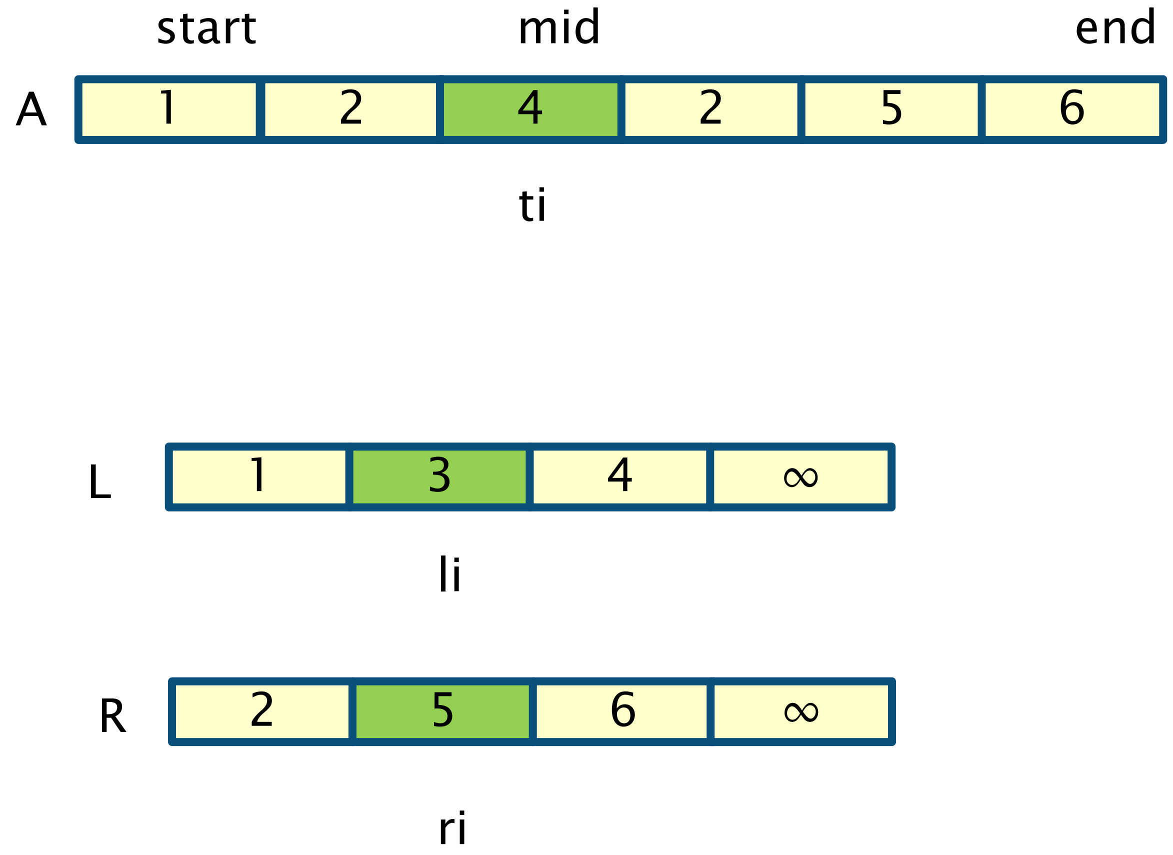

```
MERGE(A,start,mid,end)
   LEFT  = A[start:mid]
   RIGHT  = A[mid+1:end]

   LEFT.append(INF)
   RIGHT.append(INF)

   ti  = start
   li  = 0
   ri  = 0

   for ti from start to end
       if(LEFT[li] < RIGHT[ri])
              A[ti] = LEFT[li]
              li += 1
        else
              A[ti] = RIGHT[ri]
              ri += 1
```

Example execution of MERGE(A,0,2,5)



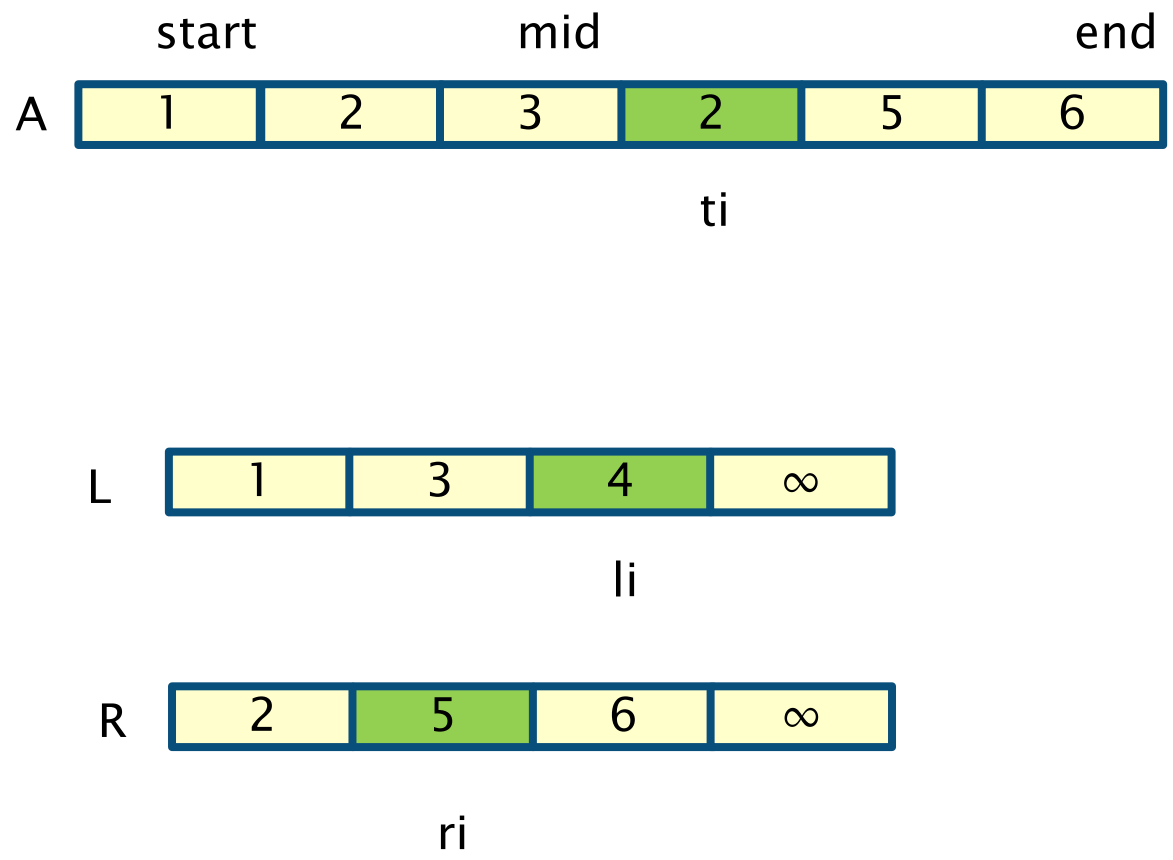

```
MERGE(A,start,mid,end)
  LEFT = A[start:mid]
  RIGHT = A[mid+1:end]

  LEFT.append(INF)
  RIGHT.append(INF)

  ti = start
  li = 0
  ri = 0

  for ti from start to end
      if(LEFT[li] < RIGHT[ri])
          A[ti] = LEFT[li]
          li += 1
       else
          A[ti] = RIGHT[ri]
          ri += 1
```

Example execution of MERGE(A,0,2,5)



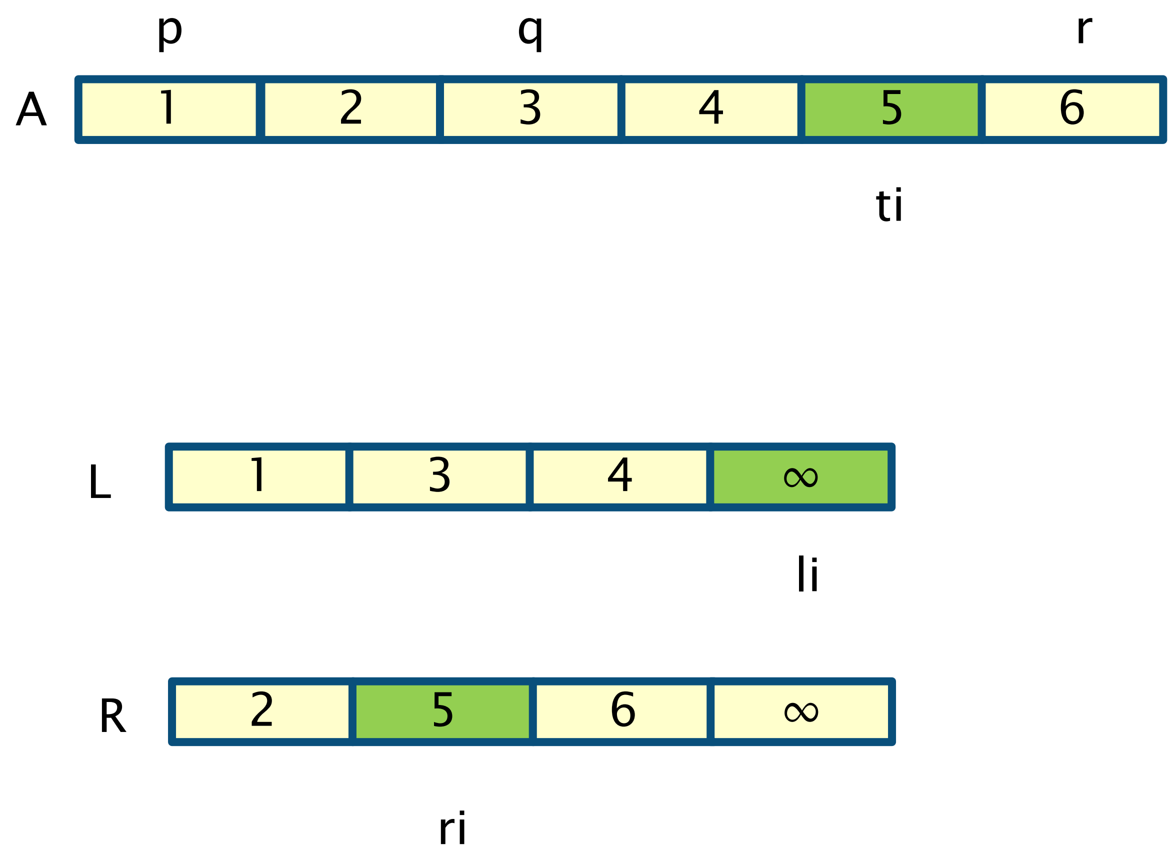

```
MERGE(A,start,mid,end)
  LEFT = A[start:mid]
  RIGHT = A[mid+1:end]

  LEFT.append(INF)
  RIGHT.append(INF)

  ti = start
  li = 0
  ri = 0

  for ti from start to end
      if(LEFT[li] < RIGHT[ri])
            A[ti] = LEFT[li]
            li += 1
       else
            A[ti] = RIGHT[ri]
            ri += 1
```

Example execution of MERGE(A,0,2,5)



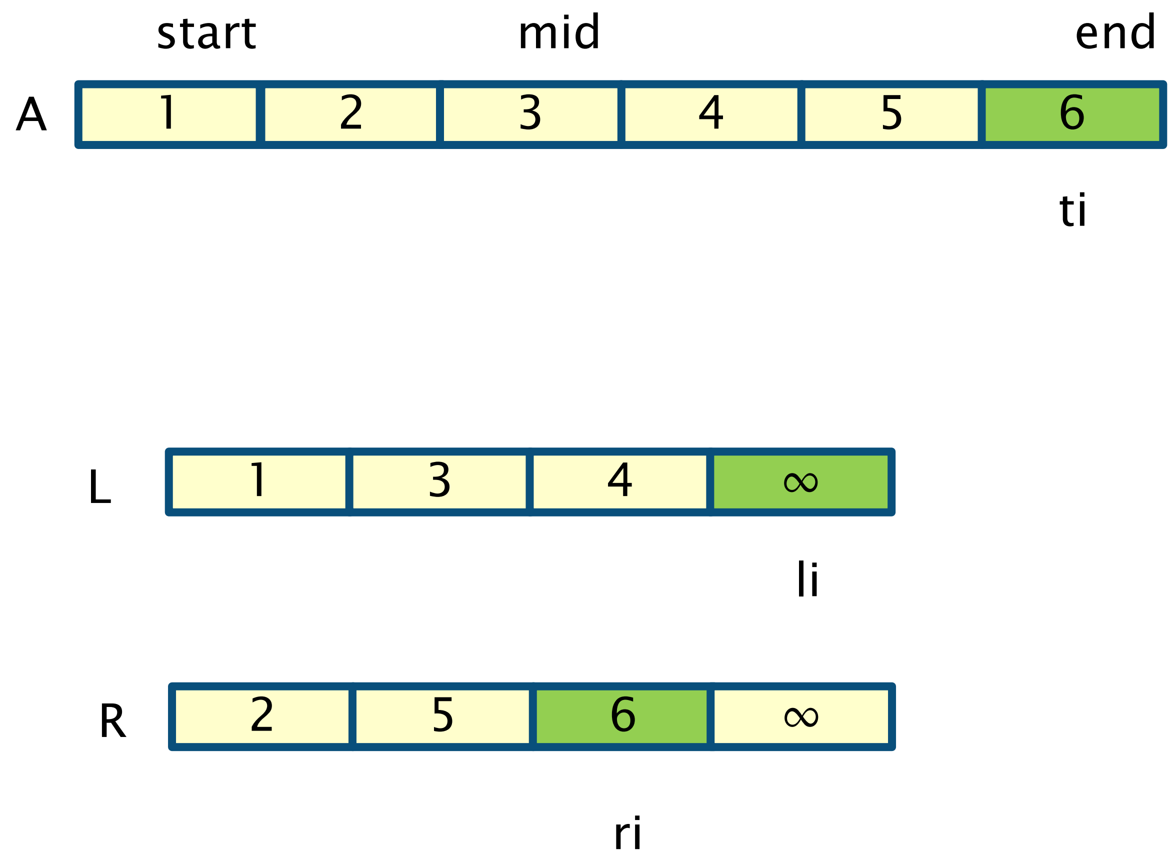

```
MERGE(A,start,mid,end)
  LEFT  = A[start:mid]
  RIGHT = A[mid+1:end]

  LEFT.append(INF)
  RIGHT.append(INF)

  ti = start
  li = 0
  ri = 0

  for ti from start to end
      if(LEFT[li] < RIGHT[ri])
           A[ti] = LEFT[li]
           li += 1
       else
           A[ti] = RIGHT[ri]
           ri += 1
```

Example execution of MERGE(A,0,2,5)



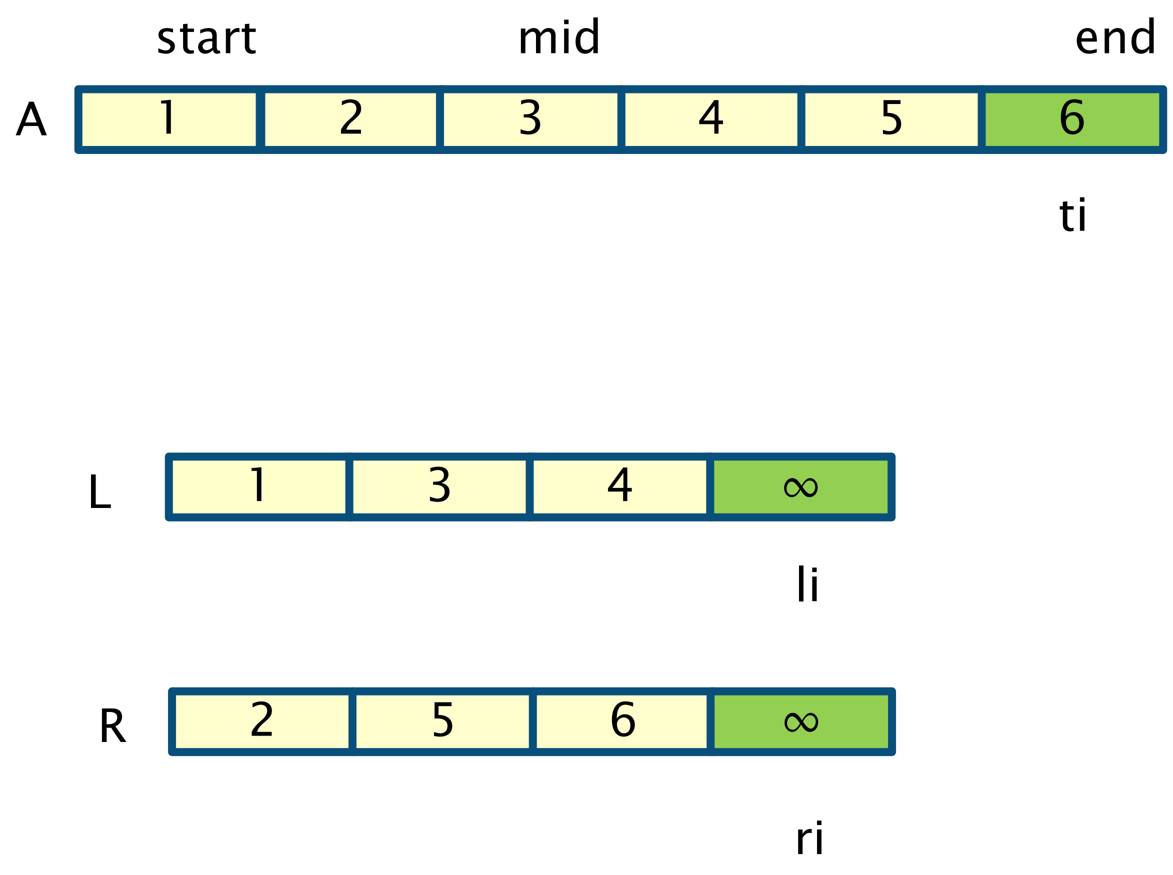

```
MERGE(A,start,mid,end)
  LEFT = A[start:mid]
  RIGHT = A[mid+1:end]

  LEFT.append(INF)
  RIGHT.append(INF)

  ti = start
  li = 0
  ri = 0

  for ti from start to end
      if(LEFT[li] < RIGHT[ri])
          A[ti] = LEFT[li]
          li += 1
       else
          A[ti] = RIGHT[ri]
          ri += 1
```

Example execution of MERGE(A,0,2,5)




```
MERGE(A,start,mid,end)
   LEFT = A[start:mid]
   RIGHT = A[mid+1:end]

   LEFT.append(INF)
   RIGHT.append(INF)

   ti = start
   li = 0
   ri = 0

   for ti from start to end
       if(LEFT[li] < RIGHT[ri])
           A[ti] = LEFT[li]
           li += 1
        else
           A[ti] = RIGHT[ri]
           ri += 1
```

Properties of MERGE

- Running time: O(n)
  - Initialisation of L and R is O(n)
  - For loop is executed n times and contains only constant operations
- "Stable" (If we encounter 2 or more keys with same value, their original order is maintained)
- O(n) working memory requirement
  - To store LEFT and RIGHT
- Keep in mind, this is *one* MERGE operation, not a complete MERGE-*SORT*

```
MERGE(A,start,mid,end)
  LEFT = A[start:mid]
  RIGHT = A[mid+1:end]

  LEFT.append(INF)
  RIGHT.append(INF)

  ti = start
  li = 0
  ri = 0

  for ti from start to end
      if(LEFT[li] < RIGHT[ri])
          A[ti] = LEFT[li]
          li += 1
       else
          A[ti] = RIGHT[ri]
          ri += 1
```

Properties of MERGE

- Running time: O(n)
  - Initialisation of L and R is O(n)
  - For loop is executed n times and contains only constant operations
- Stable (If we encounter 2 or more keys with same value, their original order is maintained)
- O(n) working memory requirement
  - To store L and R
- Keep in mind, this is *one* MERGE operation, not a complete MERGE-SORT

O(n) additional working memory

```
MERGE(A,start,mid,end)
  LEFT = A[start:mid]       O(n)
  RIGHT = A[mid+1:end]      O(n)

  LEFT.append(INF)
  RIGHT.append(INF)         O(1)

  ti = start
  li = 0                    O(1)
  ri = 0

  for ti from start to end
      if(LEFT[li] < RIGHT[ri])
          A[ti] = LEFT[li]
          li += 1           O(n)
      else
          A[ti] = RIGHT[ri]
          ri += 1
```

Then, we MERGE-SORT

MERGE-SORT

- Input: Array A and two indexes start, end for A such that start ≤ end
- Output: sorted array A[start..end]

```
MERGE-SORT(...)
```

- To sort an array A with n elements the initial call is MERGE-SORT(A,0,n-1)

MERGE-SORT

- Input: Array A and two indexes start, end for A such that start ≤ end
- Output: sorted array A[start..end]

```
MERGE-SORT(A,start,end)
  if start < end
    mid := (start+end)/2
    MERGE-SORT(A,start,mid)
    MERGE-SORT(A,mid+1,end)
    MERGE(A,start,mid,end)
```

- To sort an array A with n elements the initial call is MERGE-SORT(A,0,n-1)

Recursion tree

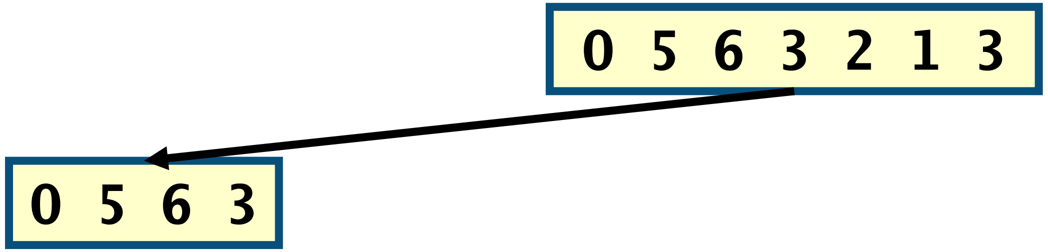
- MERGE-SORT(A,0,6) with A=[0,5,6,3,2,1,3]

0	5	6	3	2	1	3

```
MERGE-SORT(A,start,end)
  if start < end
    mid := (start+end)/2
    MERGE-SORT(A,start,mid)
    MERGE-SORT(A,mid+1,end)
    MERGE(A,start,mid,end)
```

Recursion tree

- MERGE-SORT(A,0,6) with A=[0,5,6,3,2,1,3]

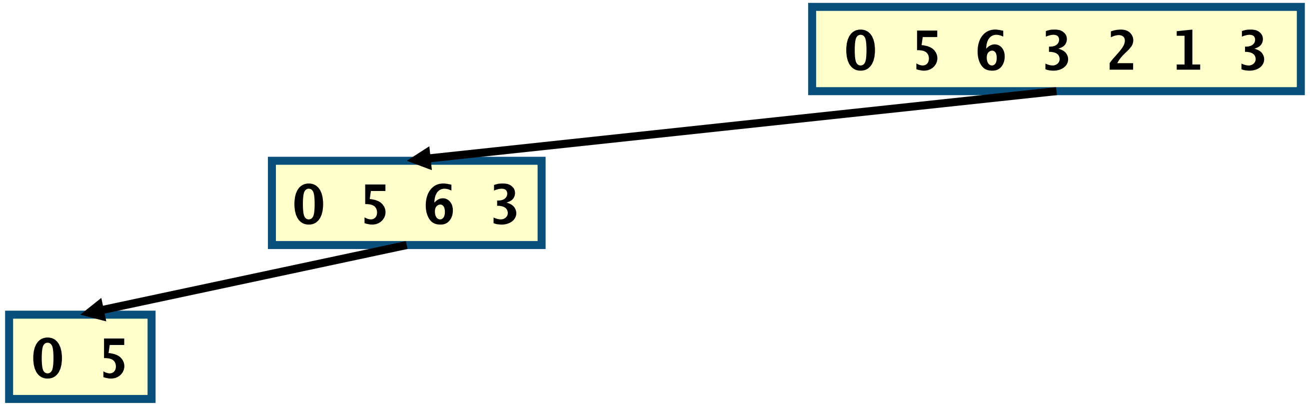


  - mid = start+end/2 = 0+6/2 = 3

```
MERGE-SORT(A,start,end)
  if start < end
    mid := (start+end)/2
    MERGE-SORT(A,start,mid)
    MERGE-SORT(A,mid+1,end)
    MERGE(A,start,mid,end)
```

Recursion tree

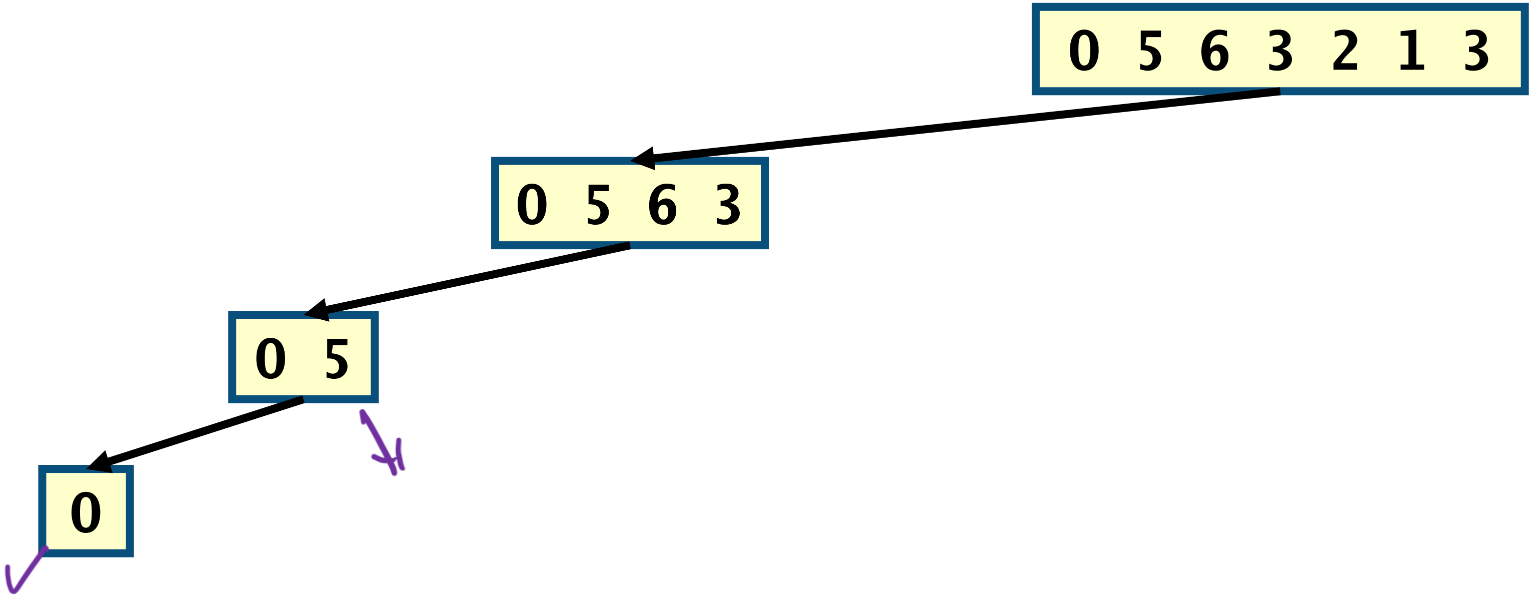
- MERGE-SORT(A,0,6) with A=[0,5,6,3,2,1,3]




```
MERGE-SORT(A,start,end)
  if start < end
    mid := (start+end)/2
    MERGE-SORT(A,start,mid)
    MERGE-SORT(A,mid+1,end)
    MERGE(A,start,mid,end)
```

Recursion tree

- MERGE-SORT(A,0,6) with A=[0,5,6,3,2,1,3]

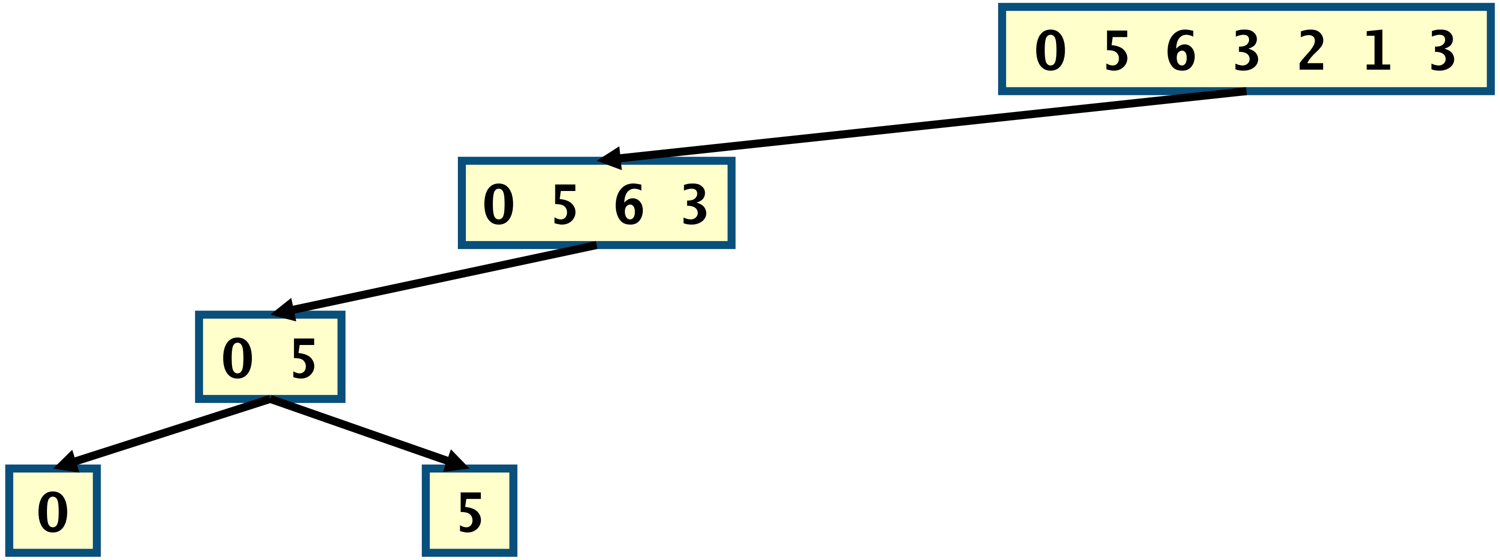


  – Recursion stopping condition
  – Now we execute the second recursive call

```
MERGE-SORT(A,start,end)
  if start < end
    mid := (start+end)/2
    MERGE-SORT(A,start,mid)
    MERGE-SORT(A,mid+1,end)
    MERGE(A,start,mid,end)
```

Recursion tree

- MERGE-SORT(A,0,6) with A=[0,5,6,3,2,1,3]

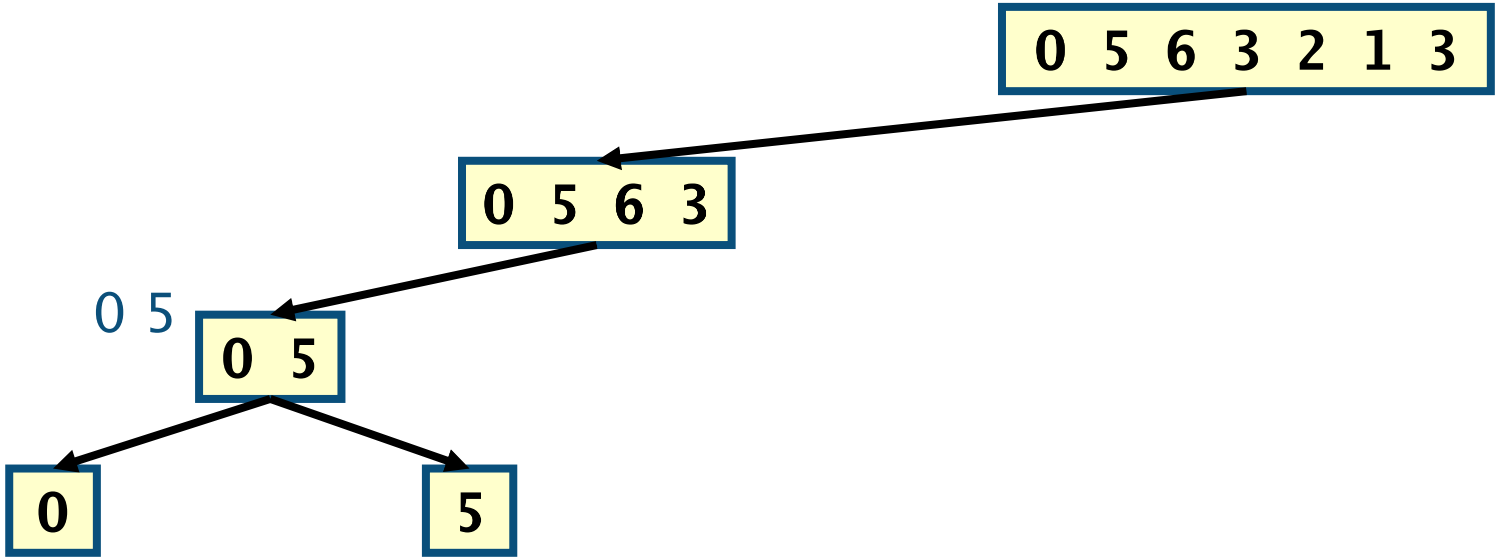


  – Now we perform the combine step by calling MERGE on the two subarrays

```
MERGE-SORT(A,start,end)
  if start < end
    mid := (start+end)/2
    MERGE-SORT(A,start,mid)
    MERGE-SORT(A,mid+1,end)
    MERGE(A,start,mid,end)
```

Recursion tree

- MERGE-SORT(A,0,6) with A=[0,5,6,3,2,1,3]

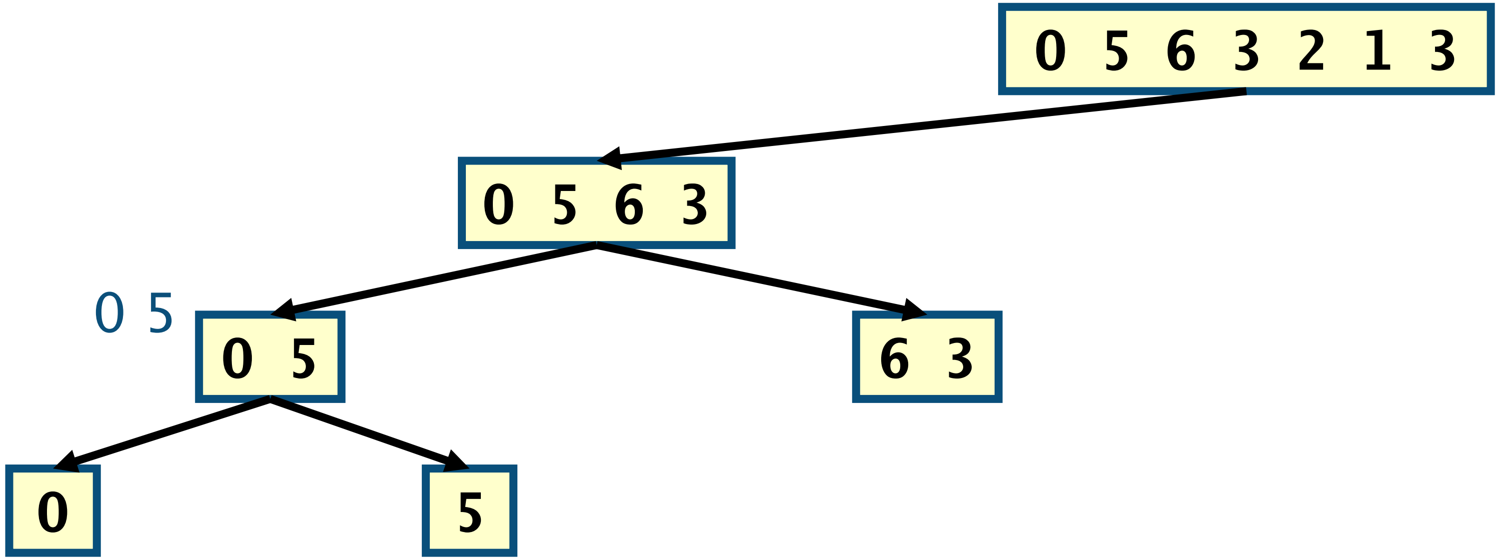


  - Now we perform the combine step by calling MERGE on the two subarrays

```
MERGE-SORT(A,start,end)
  if start < end
    mid := (start+end)/2
    MERGE-SORT(A,start,mid)
    MERGE-SORT(A,mid+1,end)
    MERGE(A,start,mid,end)
```

Recursion tree

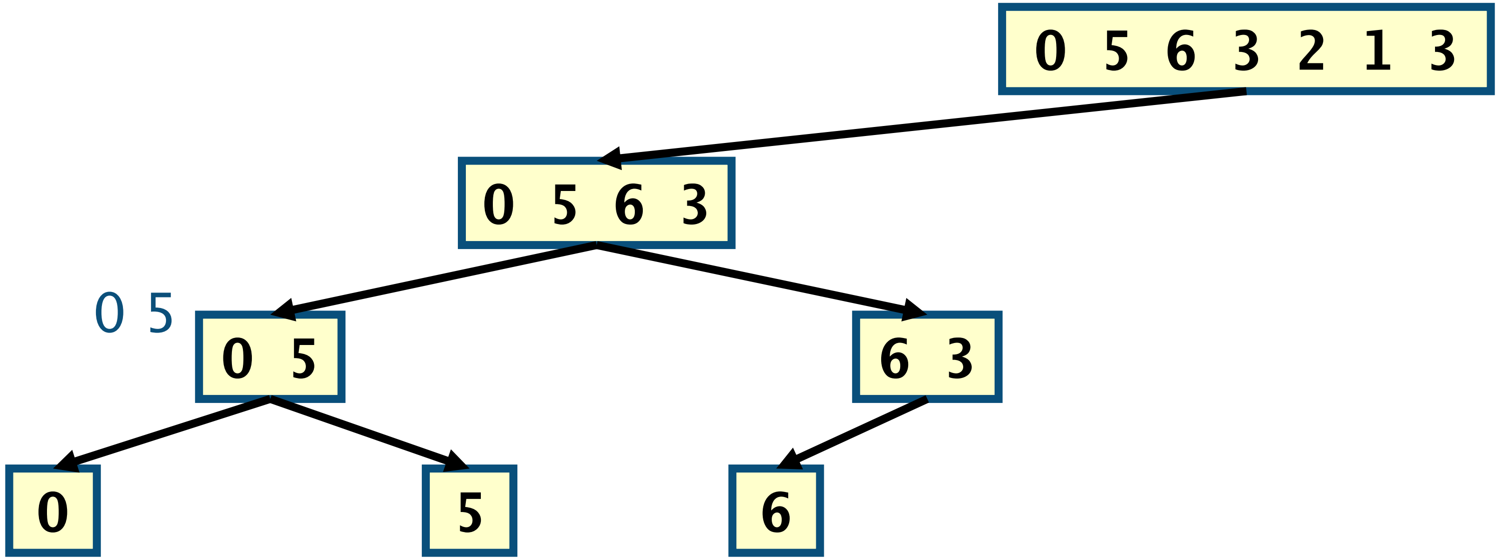
- MERGE-SORT(A,0,6) with A=[0,5,6,3,2,1,3]




```
MERGE-SORT(A,start,end)
  if start < end
    mid := (start+end)/2
    MERGE-SORT(A,start,mid)
    MERGE-SORT(A,mid+1,end)
    MERGE(A,start,mid,end)
```

Recursion tree

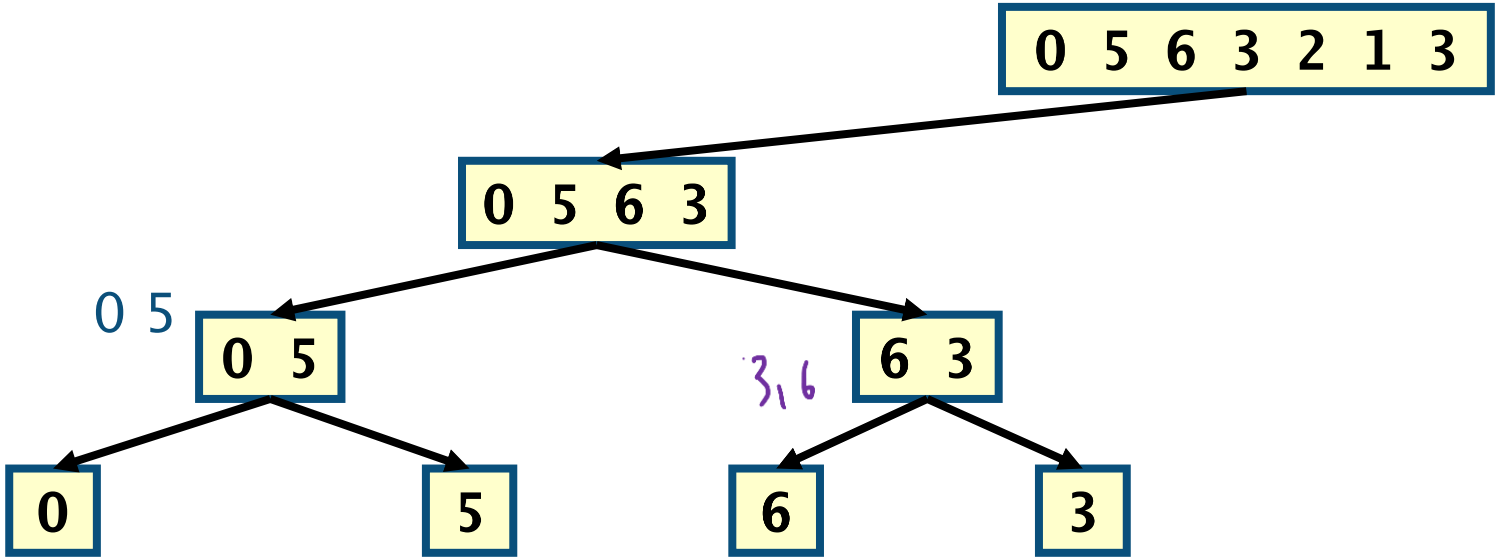
- MERGE-SORT(A,0,6) with A=[0,5,6,3,2,1,3]




```
MERGE-SORT(A,start,end)
  if start < end
    mid := (start+end)/2
    MERGE-SORT(A,start,mid)
    MERGE-SORT(A,mid+1,end)
    MERGE(A,start,mid,end)
```

Recursion tree

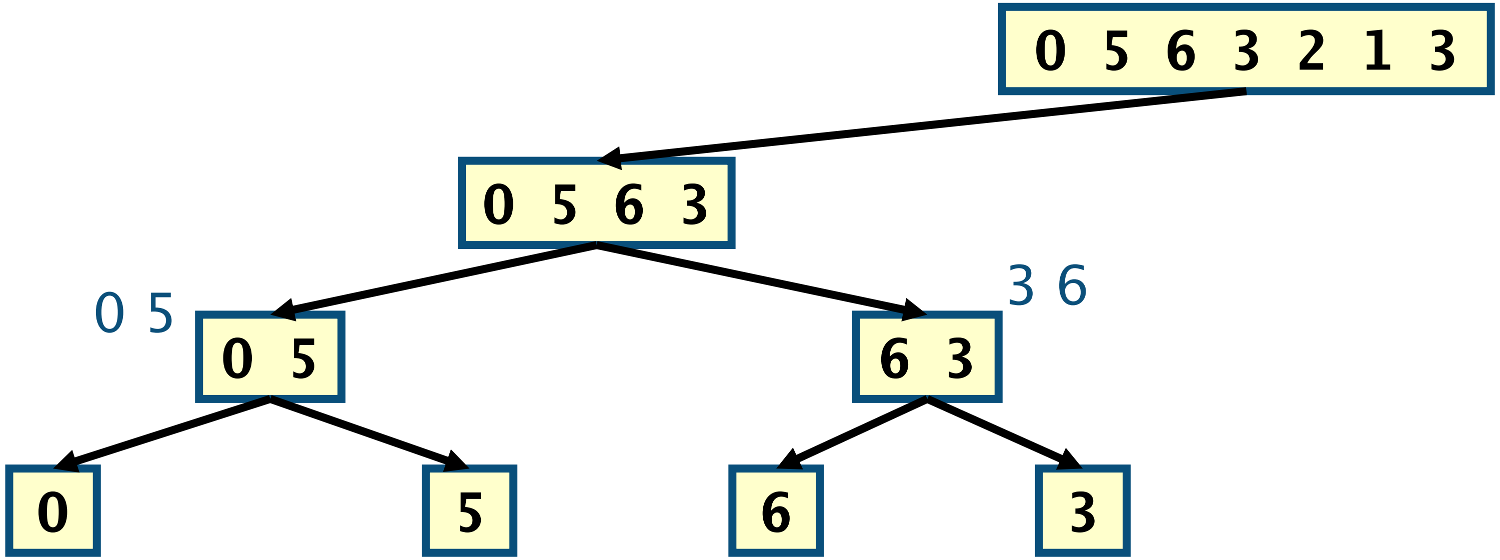
- MERGE-SORT(A,0,6) with A=[0,5,6,3,2,1,3]




```
MERGE-SORT(A,start,end)
  if start < end
    mid := (start+end)/2
    MERGE-SORT(A,start,mid)
    MERGE-SORT(A,mid+1,end)
    MERGE(A,start,mid,end)
```

Recursion tree

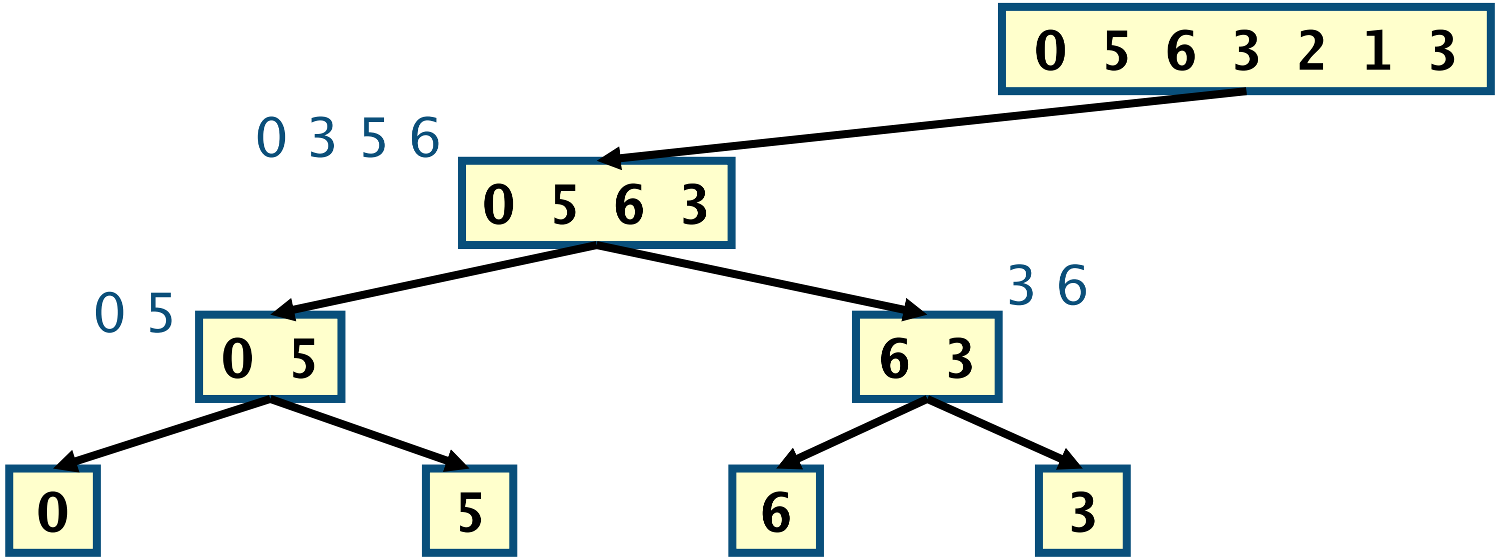
- MERGE-SORT(A,0,6) with A=[0,5,6,3,2,1,3]




```
MERGE-SORT(A,start,end)
  if start < end
    mid := (start+end)/2
    MERGE-SORT(A,start,mid)
    MERGE-SORT(A,mid+1,end)
    MERGE(A,start,mid,end)
```

Recursion tree

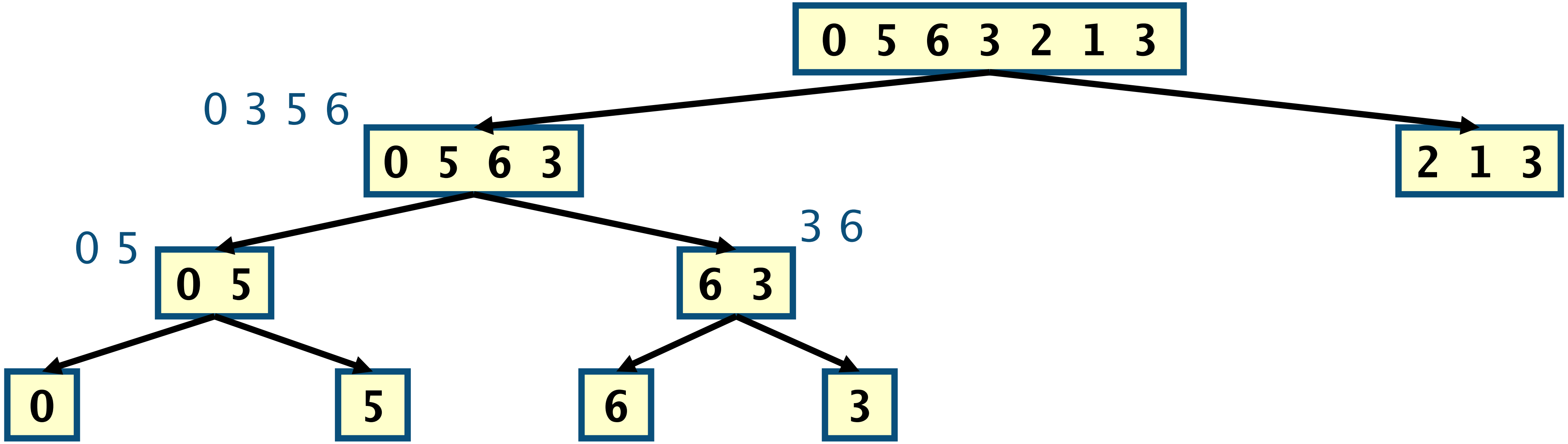
- MERGE-SORT(A,0,6) with A=[0,5,6,3,2,1,3]




```
MERGE-SORT(A,start,end)
  if start < end
    mid := (start+end)/2
    MERGE-SORT(A,start,mid)
    MERGE-SORT(A,mid+1,end)
    MERGE(A,start,mid,end)
```

Recursion tree

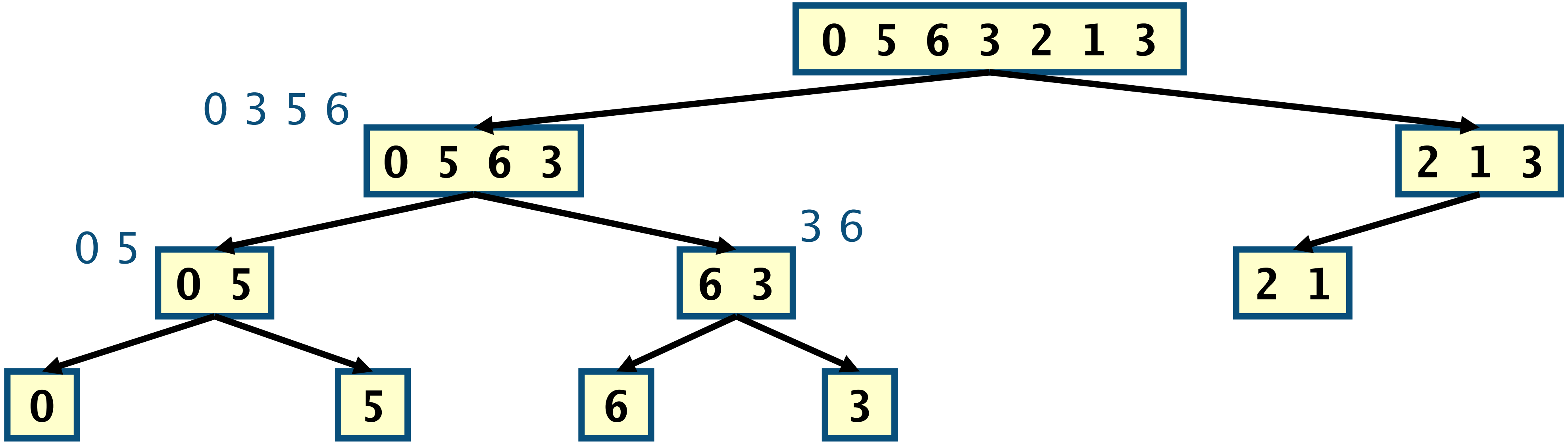
- MERGE-SORT(A,0,6) with A=[0,5,6,3,2,1,3]




```
MERGE-SORT(A,start,end)
  if start < end
    mid := (start+end)/2
    MERGE-SORT(A,start,mid)
    MERGE-SORT(A,mid+1,end)
    MERGE(A,start,mid,end)
```

Recursion tree

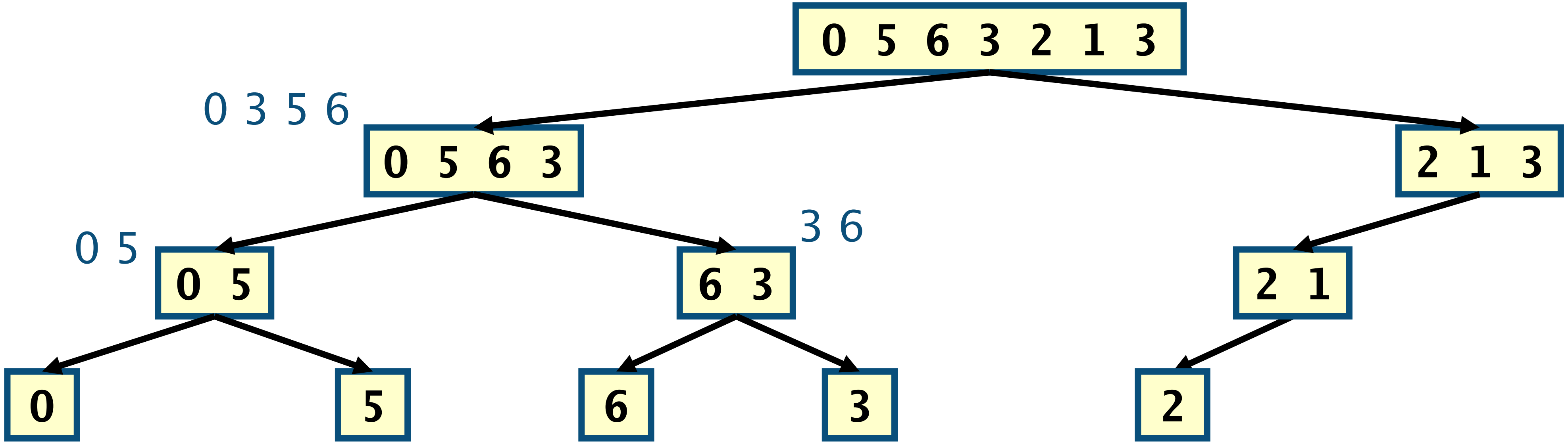
- MERGE-SORT(A,0,6) with A=[0,5,6,3,2,1,3]




```
MERGE-SORT(A,start,end)
  if start < end
    mid := (start+end)/2
    MERGE-SORT(A,start,mid)
    MERGE-SORT(A,mid+1,end)
    MERGE(A,start,mid,end)
```

Recursion tree

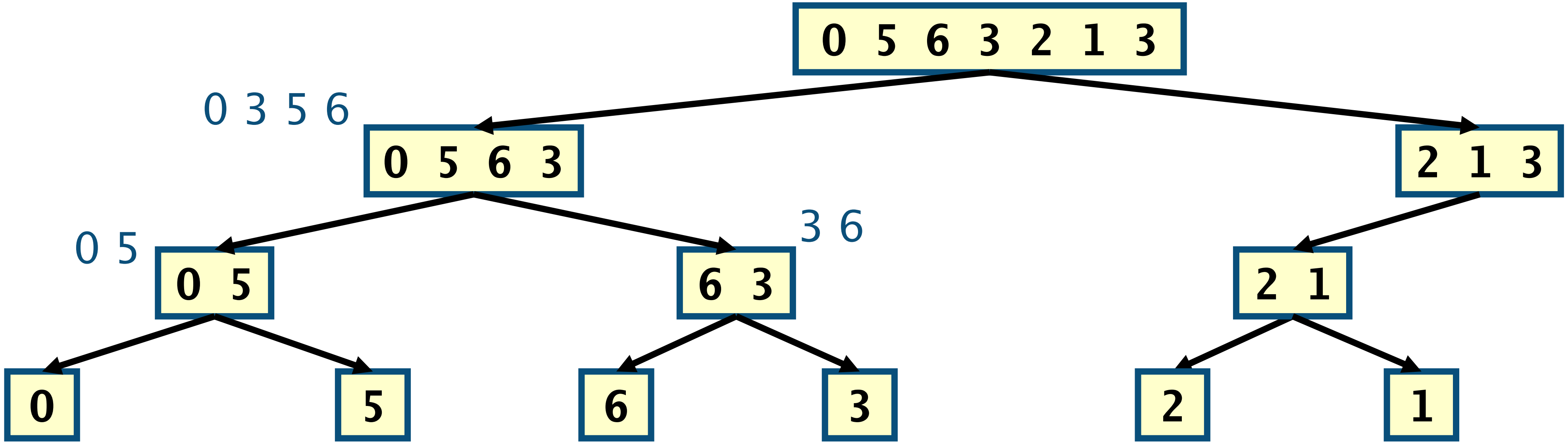
- MERGE-SORT(A,0,6) with A=[0,5,6,3,2,1,3]




```
MERGE-SORT(A,start,end)
  if start < end
    mid := (start+end)/2
    MERGE-SORT(A,start,mid)
    MERGE-SORT(A,mid+1,end)
    MERGE(A,start,mid,end)
```

Recursion tree

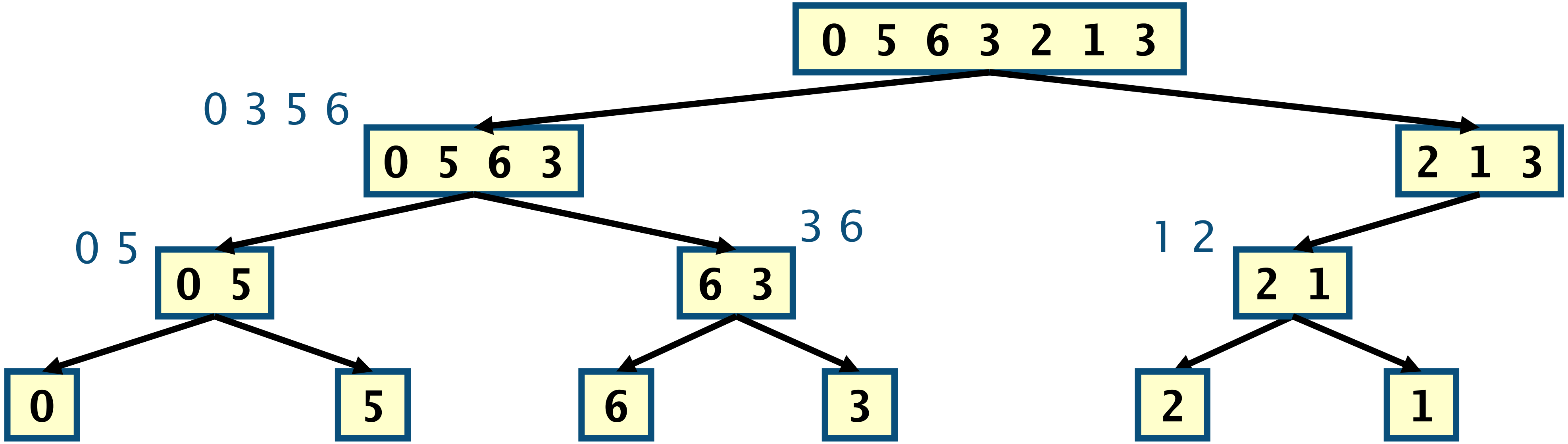
- MERGE-SORT(A,0,6) with A=[0,5,6,3,2,1,3]




```
MERGE-SORT(A,start,end)
  if start < end
    mid := (start+end)/2
    MERGE-SORT(A,start,mid)
    MERGE-SORT(A,mid+1,end)
    MERGE(A,start,mid,end)
```

Recursion tree

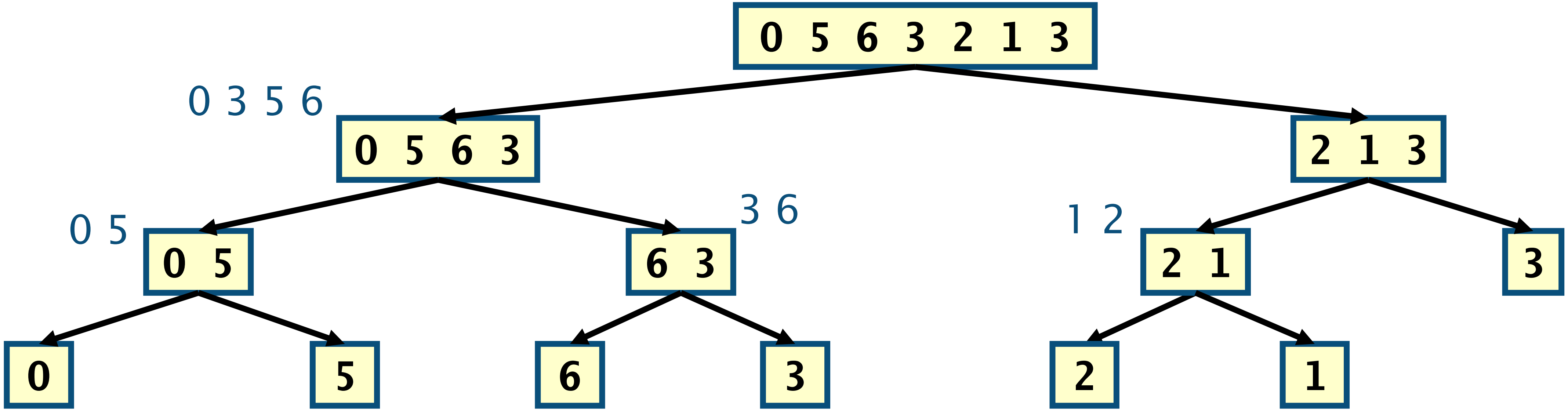
- MERGE-SORT(A,0,6) with A=[0,5,6,3,2,1,3]




```
MERGE-SORT(A,start,end)
  if start < end
    mid := (start+end)/2
    MERGE-SORT(A,start,mid)
    MERGE-SORT(A,mid+1,end)
    MERGE(A,start,mid,end)
```

Recursion tree

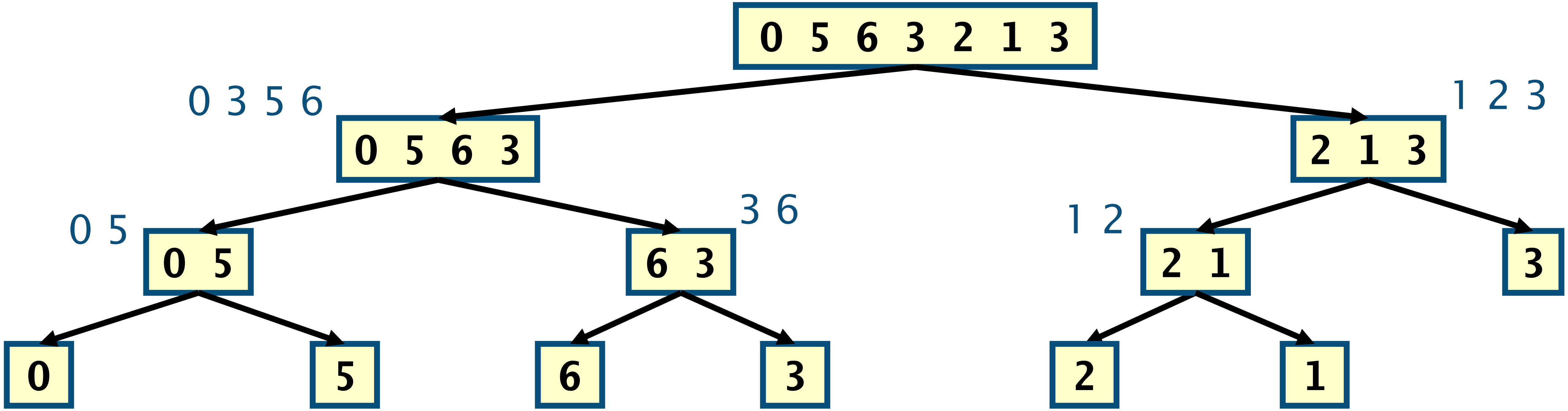
- MERGE-SORT(A,0,6) with A=[0,5,6,3,2,1,3]




```
MERGE-SORT(A,start,end)
  if start < end
    mid := (start+end)/2
    MERGE-SORT(A,start,mid)
    MERGE-SORT(A,mid+1,end)
    MERGE(A,start,mid,end)
```

Recursion tree

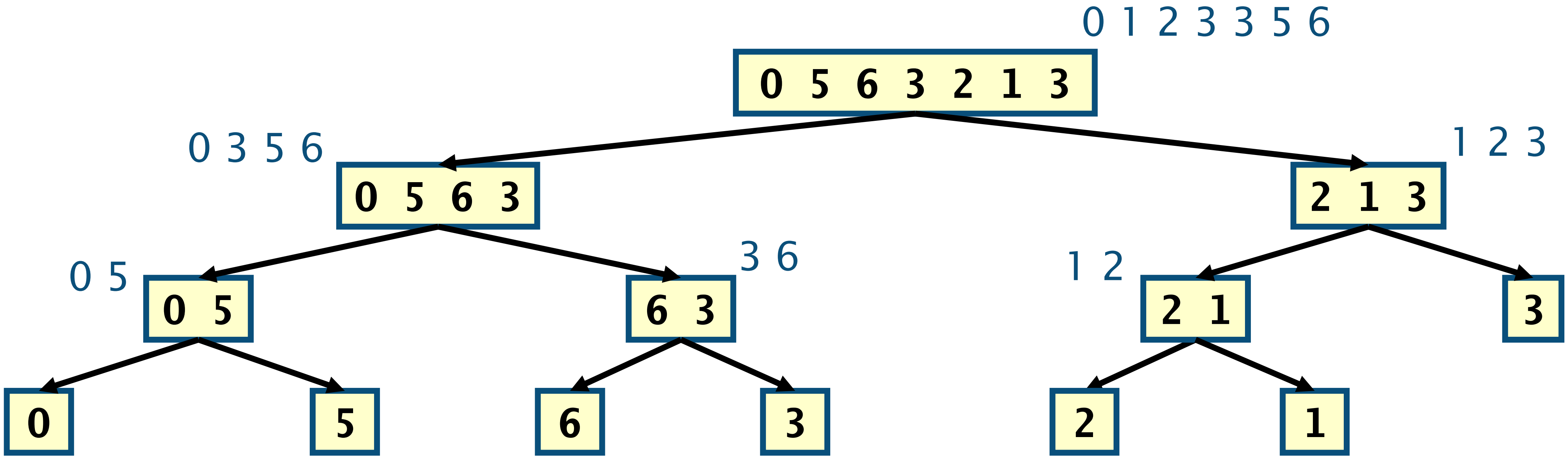
- MERGE-SORT(A,0,6) with A=[0,5,6,3,2,1,3]




```
MERGE-SORT(A,start,end)
  if start < end
    mid := (start+end)/2
    MERGE-SORT(A,start,mid)
    MERGE-SORT(A,mid+1,end)
    MERGE(A,start,mid,end)
```

Recursion tree for Merge Sort

- MERGE-SORT(A,0,6) with A=[0,5,6,3,2,1,3]



  – Termination

```
MERGE-SORT(A,start,end)
  if start < end
    mid := (start+end)/2
    MERGE-SORT(A,start,mid)
    MERGE-SORT(A,mid+1,end)
    MERGE(A,start,mid,end)
```

Properties of MERGE-SORT

- Stable as MERGE is stable*
- Not in-place as MERGE requires O(n) memory
- Running time is O(n log n) both in the best and worst cases
  - We will come to this again later…

```
MERGE-SORT(A,start,end)
  if start < end
    mid := (start+end)/2
    MERGE-SORT(A,start,mid)
    MERGE-SORT(A,mid+1,end)
    MERGE(A,start,mid,end)
```

*A sorting algorithm is said to be **stable** if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.

https://www.geeksforgeeks.org/stability-in-sorting-algorithms/

QUICK SORT

Quick sort Illustration

- Deck of cards

QUICKSORT

- Efficient divide-and-conquer sorting algorithm, like merge-sort

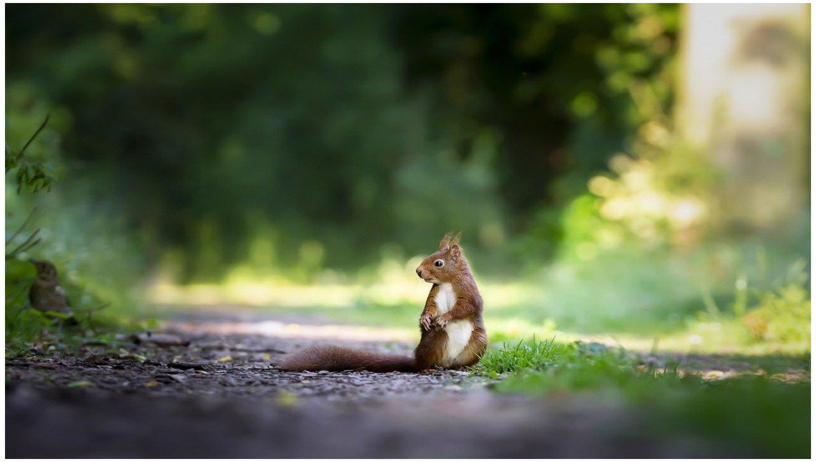
  - But, unlike merge-sort, is **in-place**, so does **not use additional** memory

  - and can often (though not always) be quicker in practice than merge sort

  - but the trade-off is: sometimes, the “divide” step does not divide in half, and in the worst case, can go back to $O(n^2)$ time complexity

QUICKSORT

- It operates as follows to sort a subarray A[start..end]
  - Divide: Pick an index pivot and partition the array in two subarrays A[start..pivot−1] and A[pivot+1..end] such that A[start..pivot−1] contains all the elements less than or equal to A[pivot], which is less than or equal to each element of A[pivot+1..end]
  - Conquer: Sort subarrays A[start..pivot−1] and A[pivot+1..end] recursively using QUICKSORT
  - Combine: no work is needed as the entire array is already sorted
- The key operation of the QUICKSORT algorithm is the partitioning of the input array in the Divide step



QUICKSORT COMPONENTS

- Quicksort (too) can be thought of as being built in top of a PARTITION operation
  - (recall the “MERGE_SORT” algorithm was build on top of the MERGE operation.
- We first define PARTITION (“interesting”)

- Then we define QUICKSORT (piece of cake)

Partition example

- Input array is A[0,7,5,2,4,4,1,3] start=0 and end=7
- Select pivot = end = 7
  - This is only one possible partitioning scheme: we will study other methods later on

lrbi start, **ti** end, pivot

start, ti							end, pivot
0	7	5	2	4	4	1	3

Pivot

- Pivot, A[pivot]=3 — 3
- Elements x ≤ 3 — x
- Elements x > 3 — x
- Unrestricted elements — x

Partition example

- Input array is A[0,7,5,2,4,4,1,3] start=0 and end=7
- Select pivot = end
  - This is only one possible partitioning scheme: we will study other methods later on

lrbi	start,ti	ti					end,pivot
	0	7	5	2	4	4	1	3

Lrbi = left-right boundary index
ti = top_index

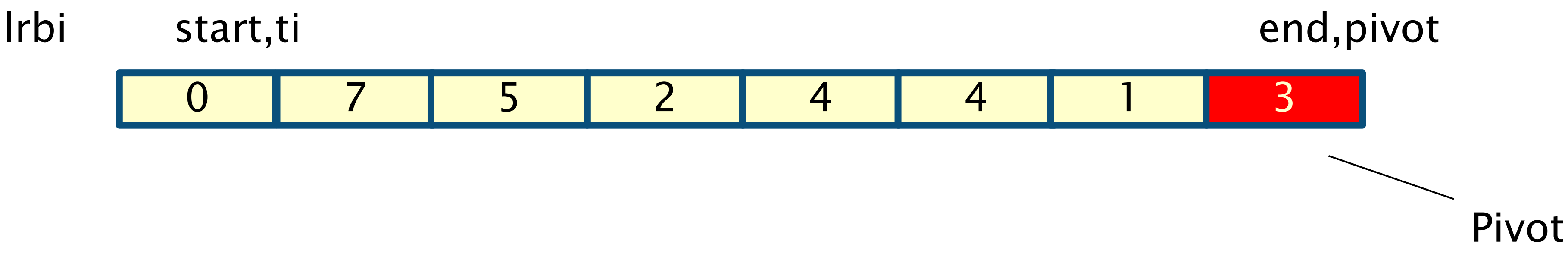
Pivot

- Lrbi points to the last element identified as $\leq x$
- Lrbi keeps track of the boundary between the LEFT ($x \leq 3$) and RIGHT ($x > 3$)
- ti is the counter keeping track of the iterations over the array from left to right
- Lrbi and ti together mark the swap operations

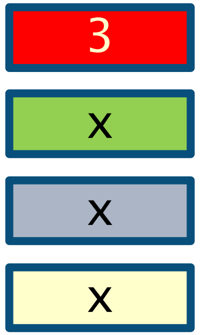
  - Pivot A[q]=3 — 3
  - Elements x ≤ 3 — x
  - Elements x > 3 — x
  - Unrestricted elements — x

Partition example

- Input array is A[0,7,5,2,4,4,1,3] start=0 and end=7
- Select pivot = end
  - This is only one possible partitioning scheme: we will study other methods later on



- Pivot A[pivot]=3
- Elements x ≤ 3
- Elements x > 3
- Unrestricted elements



Partition example

- Input array is A[0,7,5,2,4,4,1,3] start=0 and end=7
- Select pivot = end
  - This is only one possible partitioning scheme: we will study other methods later on

start,lrbi	ti						end,pivot
0	7	5	2	4	4	1	3

  - $0 \leq 3$, increase i, swap A[lrbi] with A[ti] and then increase ti (swap 0 with itself in this case)
  - This expands the green region (i.e. the region with values <= pivot)

Pivot A[pivot]=3	3
Elements $x \leq 3$	x
Elements $x > 3$	x
Unrestricted elements	x

Partition example

- Input array is A[0,7,5,2,4,4,1,3] start=0 and end=7
- Select pivot = end
  - This is only one possible partitioning scheme: we will study other methods later on

start,lrbi		ti					end,pivot
0	7	5	2	4	4	1	3

  - 7 > 3, increase ti
  - This expand the grey region (i.e. the region with values > pivot)

Pivot A[pivot]=3	3
Elements x ≤ 3	x
Elements x > 3	x
Unrestricted elements	x

Partition example

- Input array is A[0,7,5,2,4,4,1,3] start=0 and end=7
- Select pivot = end
  - This is only one possible partitioning scheme: we will study other methods later on

start,lrbi			ti				end,pivot
0	7	5	2	4	4	1	3

  - 5 > 3, increase ti
  - Expand grey region

Pivot A[pivot]=3	3
Elements $x \leq 3$	x
Elements $x > 3$	x
Unrestricted elements	x

Partition example

- Input array is A[0,7,5,2,4,4,1,3] start=0 and end=7
- Select pivot = end
  - This is only one possible partitioning scheme: we will study other methods later on

start	lrbi			ti			end,pivot
0	2	5	7	4	4	1	3

  - $2 \leq 3$, increase lrbi, swap A[lrbi] with A[ti] and then increase ti
  - Expand green region

Pivot A[pivot]=3	3
Elements $x \leq 3$	x
Elements $x > 3$	x
Unrestricted elements	x

Partition example

- Input array is A[0,7,5,2,4,4,1,3] start=0 and end=7
- Select pivot = end
  - This is only one possible partitioning scheme: we will study other methods later on

start	lrbi				ti		end,pivot
0	2	5	7	4	4	1	3

  - 4 > 3, increase ti
  - Expand grey region

Pivot A[pivot]=3 — 3

Elements $x \leq 3$ — x

Elements $x > 3$ — x

Unrestricted elements — x

Partition example

- Input array is A[0,7,5,2,4,4,1,3] start=0 and end=7
- Select pivot = end
  - This is only one possible partitioning scheme: we will study other methods later on

start	lrbi					ti	end,pivot
0	2	5	7	4	4	1	3

  - 4 > 3, increase ti
  - Expand grey region

Pivot A[pivot]=3	3
Elements $x \leq 3$	x
Elements $x > 3$	x
Unrestricted elements	x

Partition example

- Input array is A[0,7,5,2,4,4,1,3] start=0 and end=7
- Select pivot = end
  - This is only one possible partitioning scheme: we will study other methods later on

start		lrbi					end,pivot
0	2	1	7	4	4	5	3

  - 1 ≤ 3, increase lrbi, swap A[lrbi] with A[ti]
  - Expand green region

Pivot A[pivot]=3	3
Elements x ≤ 3	x
Elements x > 3	x
Unrestricted elements	x

Partition example

- Input array is A[0,7,5,2,4,4,1,3] start=0 and end=7
- Select pivot = end
  - This is only one possible partitioning scheme: we will study other methods later on

lrbi+1 now marks the correct position for the pivot

start		lrbi					end,pivot
0	2	1	7	4	4	5	3

  - 1 ≤ 3, increase lrbi, swap A[lrbi] with A[ti]
  - Expand green region

Pivot A[pivot]=3	3
Elements x ≤ 3	x
Elements x > 3	x
Unrestricted elements	x

Partition example

- Input array is A[0,7,5,2,4,4,1,3] start=0 and end=7
- Select pivot = end
  - This is only one possible partitioning scheme: we will study other methods later on

start		lrbi					end,pivot
0	2	1	3	4	4	5	7

  - No more unrestricted elements left
  - Swap A[lrbi+1] with A[end] to place the pivot in the middle
  - Termination

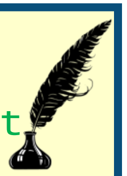
Pivot A[pivot]=3	3
Elements $x \leq 3$	x
Elements $x > 3$	x
Unrestricted elements	x

PARTITION – The Algorithm

- Input: Array **A** and two indexes **start**, **end** for **A** such that **start ≤ end**
  - No assumptions on the input
- Output: re-arranged A and index **pivot** such that
  - A[start..pivot-1] ≤ A[pivot] < A[pivot+1..end]

- **A** is rearranged in place
- Running time is **O(n)**

```
PARTITION(A,start,end)
   x := A[end]  #x=value at pivot
   lrbi := start - 1
   for ti = start to end - 1
      #if you find a value less then the pivot value
      #move it to the left of left-right boundary
      if A[ti] ≤ x
         lrbi := lrbi + 1
         SWAP(A[lrbi],A[ti])
   #after ti loop is done, lrbi marks the place
   #where pivot should end up
   SWAP(A[lrbi+1],A[end])
   return lrbi + 1
```



Now that we have defined PARTITION, we will use it to define the RECURSIVE algorithm QUICKSORT

QUICKSORT

- Input: Array A and two indexes start, end for A such that start ≤ end
- Output: sorted array A[start..end]

```
QUICKSORT(A,start,end)
  if start < end
    pivot := PARTITION(A,start,end)      → pivot now in place
    QUICKSORT(A,start,pivot-1)           → sort left of pivot
    QUICKSORT(A,pivot+1,end)             → sort right of pivot
```

- To sort an array A with n elements the initial call is QUICKSORT(A,0,n−1)
- After each partition, the first recursive call operates on the green region while the second call operates on the grey region of A

QUICKSORT recursion tree

- Try to derive the recursion tree of QUICKSORT(A,0,6) with A= [6,5,0,4,1,8,3]

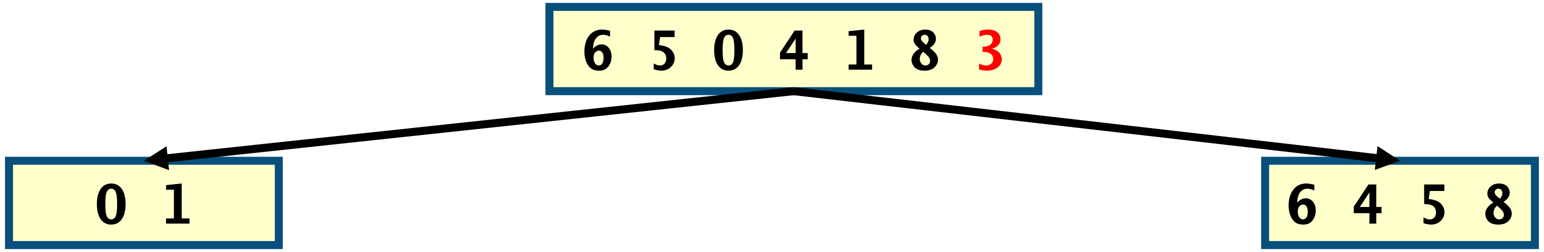
QUICKSORT recursion tree

- Try to derive the recursion tree of QUICKSORT(A,0,6) with A= [6,5,0,4,1,8,3]

6	5	0	4	1	8	3

QUICKSORT recursion tree

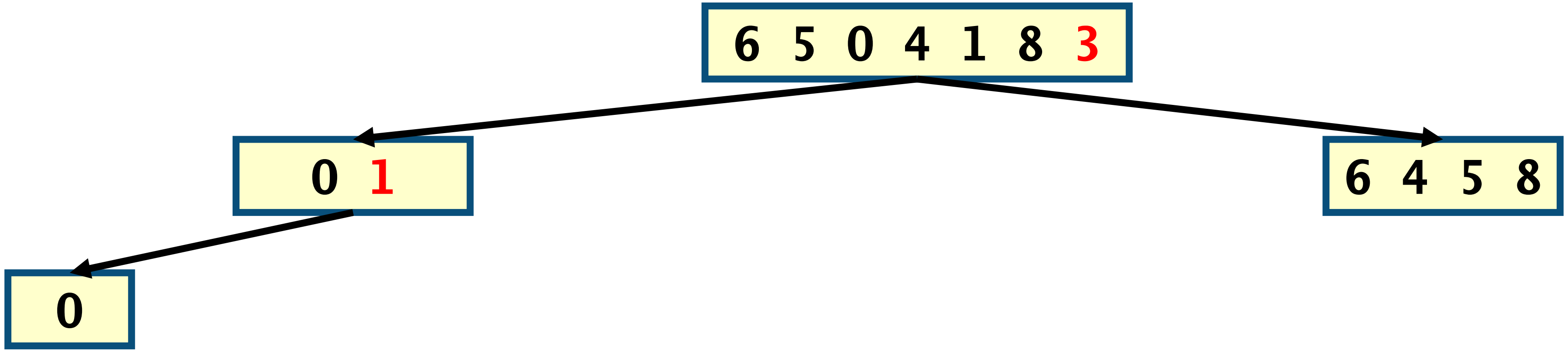
- Try to derive the recursion tree of QUICKSORT(A,0,6) with A= [6,5,0,4,1,8,3]



  – Partition of [6,5,0,4,1,8,3] with pivot [3] yelds [0,1] and [6,4,5,8]

QUICKSORT recursion tree

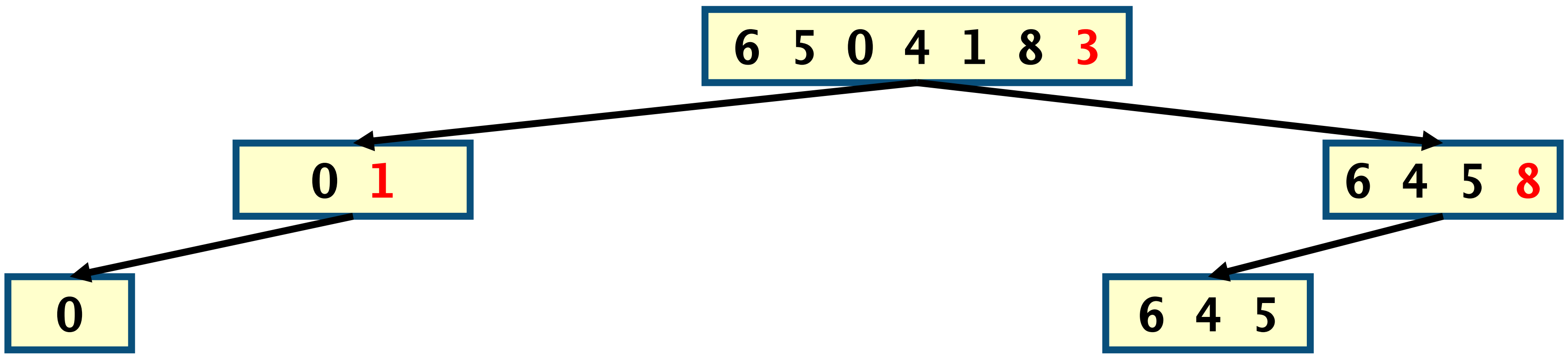
- Try to derive the recursion tree of QUICKSORT(A,0,6) with A= [6,5,0,4,1,8,3]



  – Partition of [0,1] with pivot [1] yelds [0] and []

QUICKSORT recursion tree

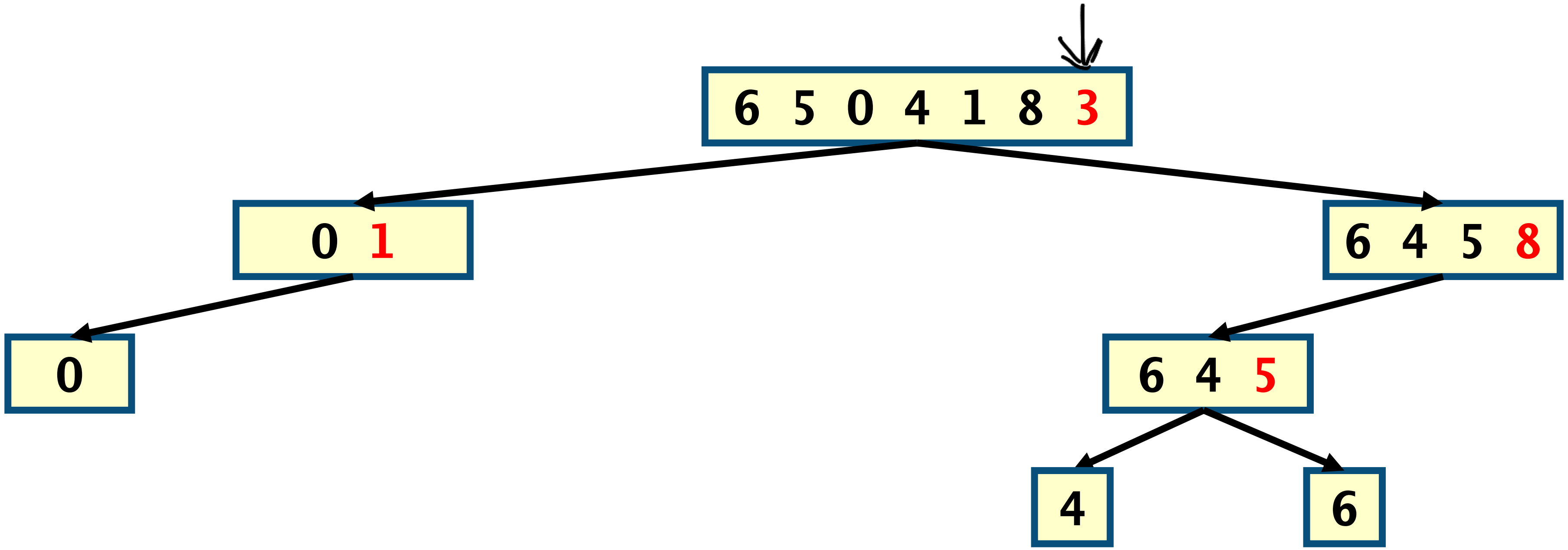
- Try to derive the recursion tree of QUICKSORT(A,0,6) with A= [6,5,0,4,1,8,3]



  – Partition of [6,4,5,8] with pivot [8] yelds [6,4,5] and []

QUICKSORT recursion tree

- Try to derive the recursion tree of QUICKSORT(A,0,6) with A= [6,5,0,4,1,8,3]



  - Partition of [6,4,5] with pivot [5] yelds [4] and [6]
  - Termination. A sorted **in place**: [0,1,3,4,5,6,8]

Some alternative partitioning schemes

- **Choice of pivot can play an important role**
- **Ideally, pivot should be the median value, so that partition operation cuts the array into exact halves**
  - On the flip side, if the pivot happens to be largest of smallest element, then one of the two recursive call does nothing, and the other one ends up with all the remaining elements, which means we don’t really benefit from the “divide and conquer” approach
  - BUT: finding the *exact* median is itself an expensive operation!
  - So, we use some light-weight mechanisms to chose better (though not necessarily optimal) pivot
  - E.g. *Choose the median of three (start,mid,end)*
  - Other options are there too, which work better in some situations over others
    - Choose the middle element
    - Choose the pivot randomly

Merge Sort and Quick Sort, Comparison

- MERGE-SORT and QUICKSORT are two efficient divide-and-conquer sorting algorithms

	MERGE-SORT	QUICKSORT
Best case running time	$O(n \log n)$	$O(n \log n)$
Average case running time	$O(n \log n)$	$O(n \log n)$
Worst case running time	$O(n \log n)$	$O(n^2)$
Space complexity	$O(n)$	$O(\log n)$
Stable	Yes	No