Algorithms and Data Structures 2

5 – MERGE-SORT

Dr Michele Sevegnani

School of Computing Science
University of Glasgow

michele.sevegnani@glasgow.ac.uk

Outline

- **MERGE**
  - Properties
- **MERGE-SORT**
  - Properties
  - Improvements

MERGE-SORT

- **Efficient divide-and-conquer sorting algorithm**
  - According to Knuth, it was invented in 1938 to merge two decks of punched cards in one pass
  - First known implementation due to von Neumann in 1945
- **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
- **The key operation of the MERGE-SORT algorithm is the merging of two sorted arrays in the Combine step**
  - Via the MERGE procedure

MERGE

- **Input: Array A and three indexes p, q, r for A such that $p \le q < r$**
  - Subarrays A[p..q] and A[q+1..r] are assumed sorted
- **Output: sorted subarray A[p..r]**

- **We only copy L and R. Sorted array is stored in A**
- **We use sentinels (∞) to avoid checking at every step if L or R have been entirely scanned**

```
MERGE(A,p,q,r)
  $n_1$ := q – p + 1
  $n_2$ := r – q
  copy A[p..q] to L[0..$n_1$]
  copy A[q+1..r] to R[0..$n_2$]
  L[$n_1$] := ∞
  R[$n_2$] := ∞
  i,j := 0
  for k = p to r
    if L[i] ≤ R[j]
      A[k] := L[i]
      i := i + 1
    else
      A[k] := R[j]
      j := j + 1
```

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

	p		q			r
A	1	3	4	2	5	6

```
MERGE(A,p,q,r)
  $n_1$ := q – p + 1
  $n_2$ := r – q
  copy A[p..q] to L[0..$n_1$]
  copy A[q+1..r] to R[0..$n_2$]
  L[$n_1$] := ∞
  R[$n_2$] := ∞
  i,j := 0
  for k = p to r
    if L[i] ≤ R[j]
      A[k] := L[i]
      i := i + 1
    else
      A[k] := R[j]
      j := j + 1
```

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

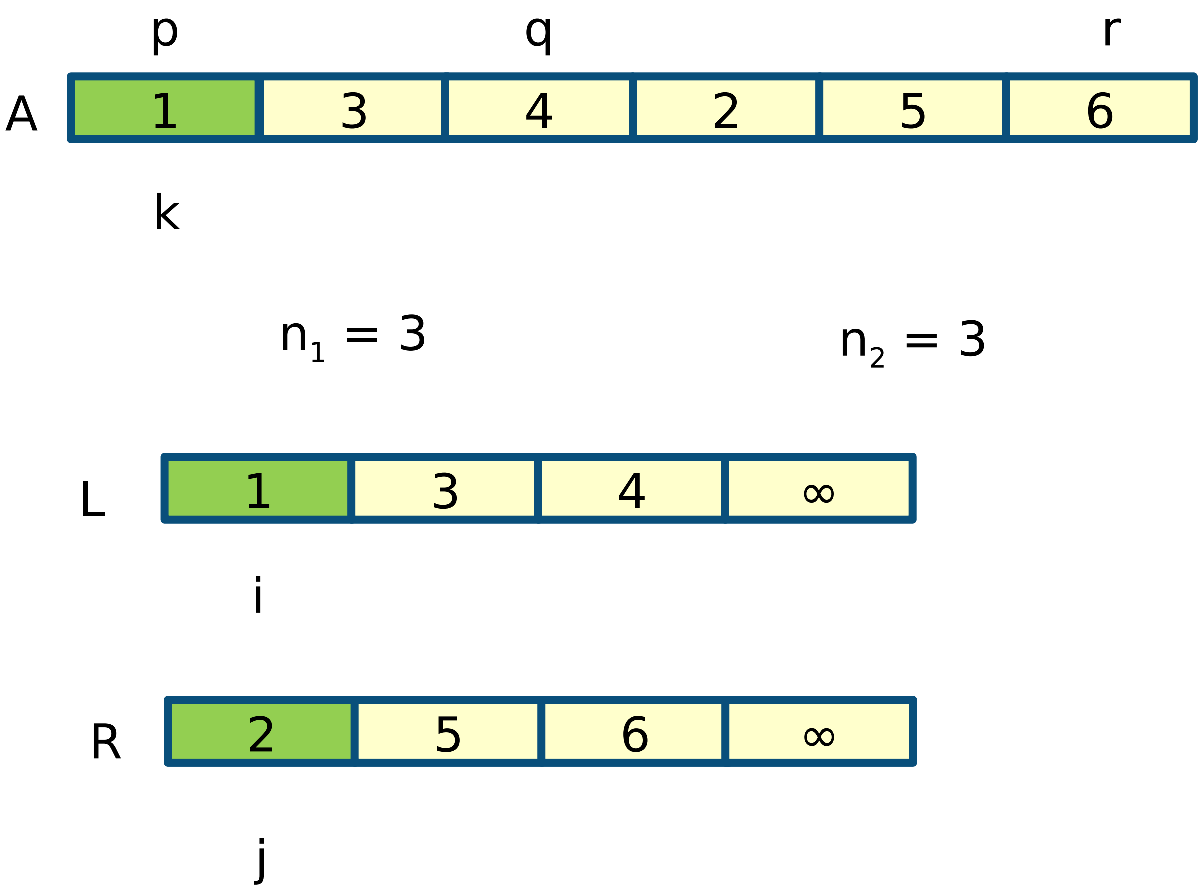
	p		q			r
A	1	3	4	2	5	6

$n_1 = 3$

$n_2 = 3$

```
MERGE(A,p,q,r)
  $n_1$ := q − p + 1
  $n_2$ := r − q
  copy A[p..q] to L[0..$n_1$]
  copy A[q+1..r] to R[0..$n_2$]
  L[$n_1$] := ∞
  R[$n_2$] := ∞
  i,j := 0
  for k = p to r
    if L[i] ≤ R[j]
      A[k] := L[i]
      i := i + 1
    else
      A[k] := R[j]
      j := j + 1
```

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



p q r

A | 1 | 3 | 4 | 2 | 5 | 6 |

k

$n_1 = 3$ $n_2 = 3$

L | 1 | 3 | 4 | ∞ |

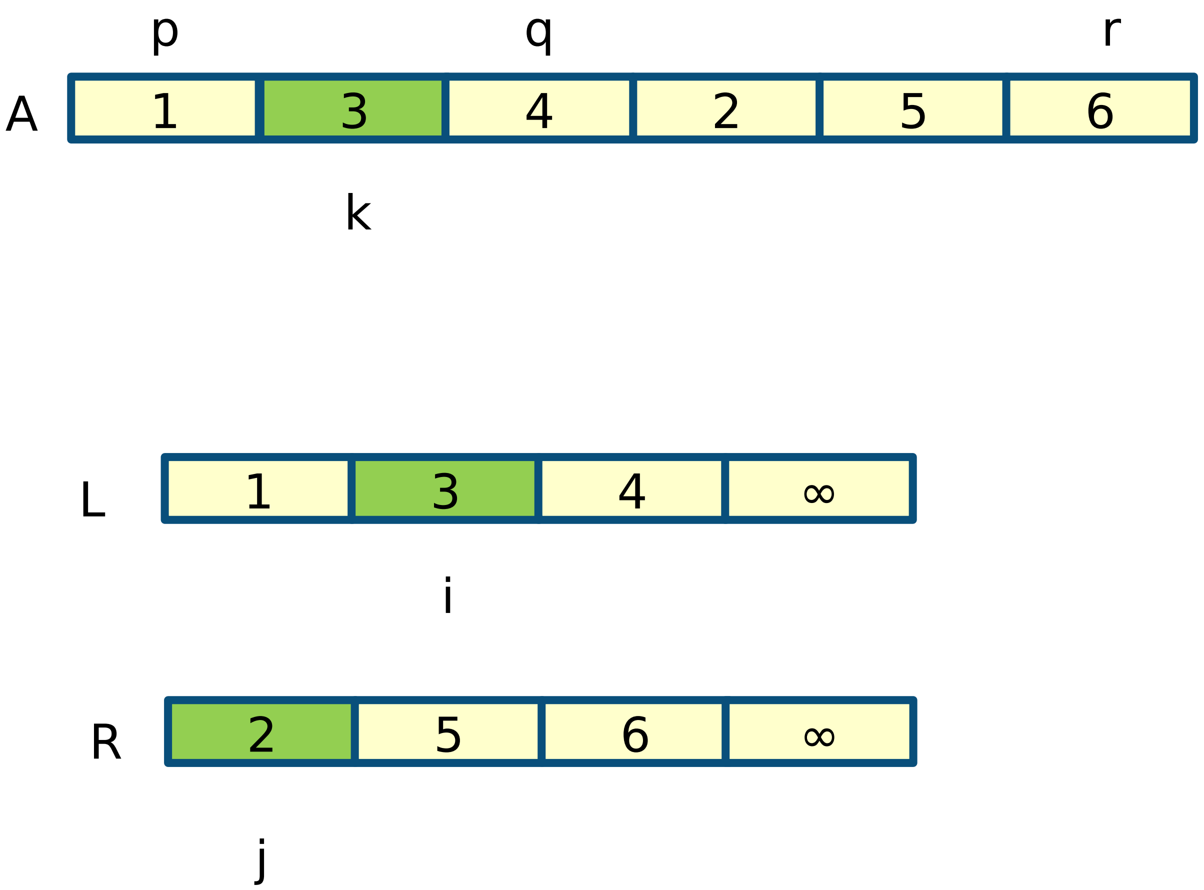
i

R | 2 | 5 | 6 | ∞ |

j

```
MERGE(A,p,q,r)
  $n_1$ := q − p + 1
  $n_2$ := r − q
  copy A[p..q] to L[0..$n_1$]
  copy A[q+1..r] to R[0..$n_2$]
  L[$n_1$] := ∞
  R[$n_2$] := ∞
  i,j := 0
  for k = p to r
    if L[i] ≤ R[j]
      A[k] := L[i]
      i := i + 1
    else
      A[k] := R[j]
      j := j + 1
```

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



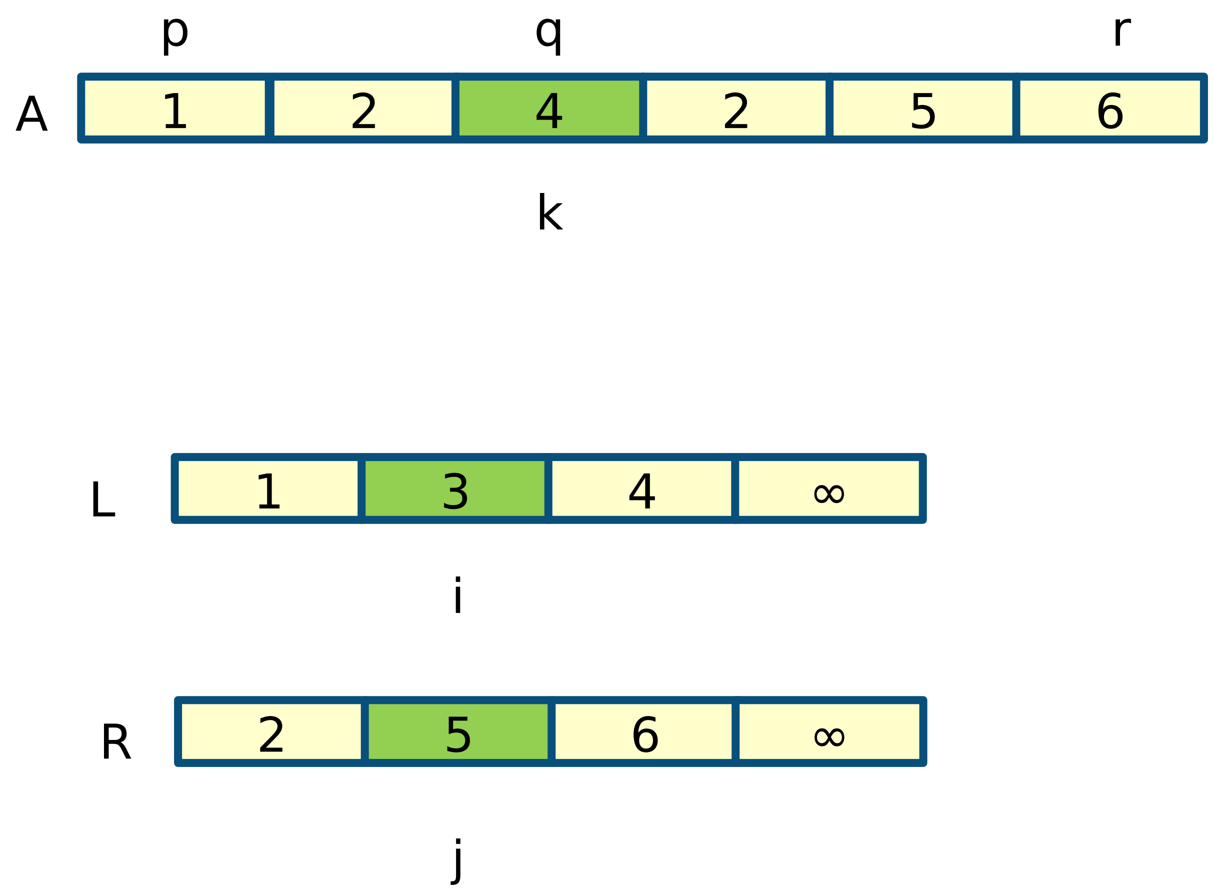
	p		q			r
A	1	3	4	2	5	6
		k				

L	1	3	4	∞
		i		

R	2	5	6	∞
	j			

```
MERGE(A,p,q,r)
  $n_1$ := q – p + 1
  $n_2$ := r – q
  copy A[p..q] to L[0..$n_1$]
  copy A[q+1..r] to R[0..$n_2$]
  L[$n_1$] := ∞
  R[$n_2$] := ∞
  i,j := 0
  for k = p to r
    if L[i] ≤ R[j]
      A[k] := L[i]
      i := i + 1
    else
      A[k] := R[j]
      j := j + 1
```

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



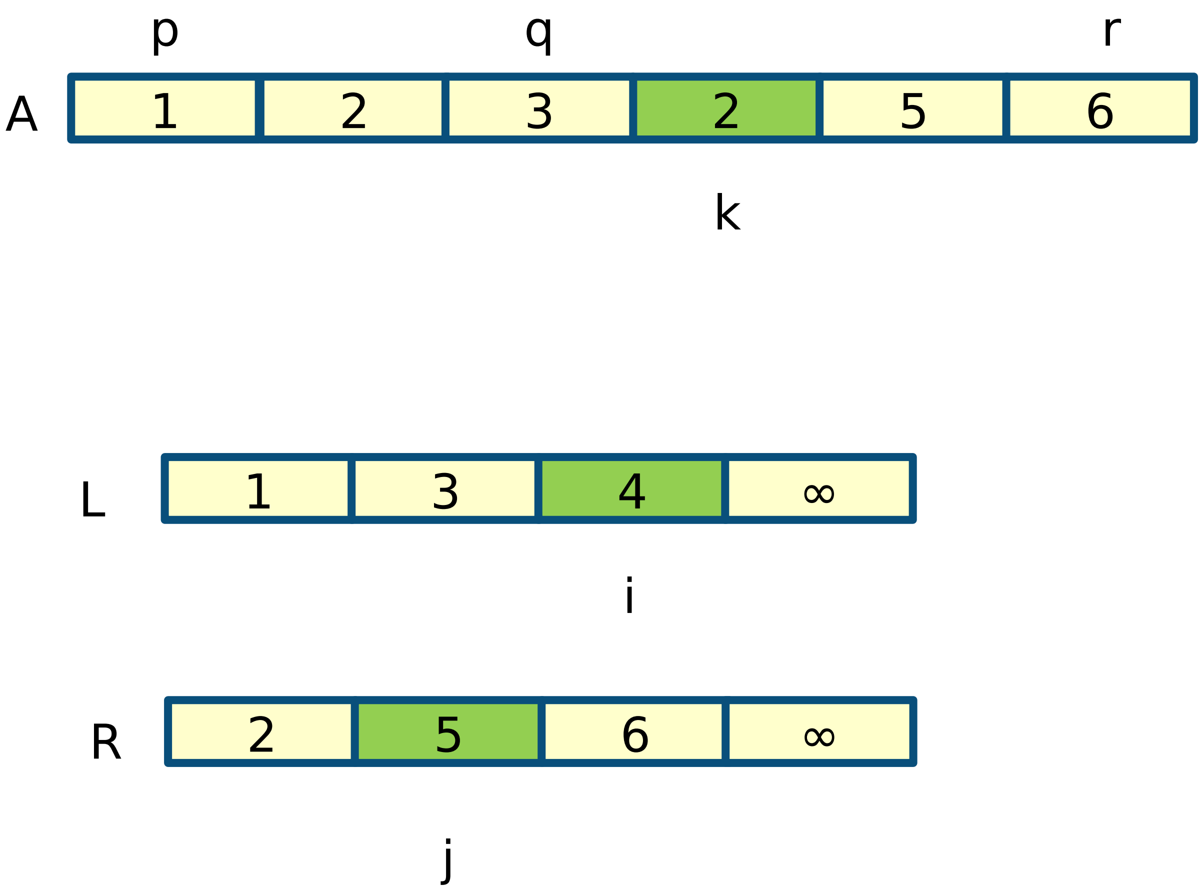
	p		q			r
A	1	2	4	2	5	6
			k			

L	1	3	4	∞
		i		

R	2	5	6	∞
		j		

```
MERGE(A,p,q,r)
  $n_1$ := q – p + 1
  $n_2$ := r – q
  copy A[p..q] to L[0..$n_1$]
  copy A[q+1..r] to R[0..$n_2$]
  L[$n_1$] := ∞
  R[$n_2$] := ∞
  i,j := 0
  for k = p to r
    if L[i] ≤ R[j]
      A[k] := L[i]
      i := i + 1
    else
      A[k] := R[j]
      j := j + 1
```

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



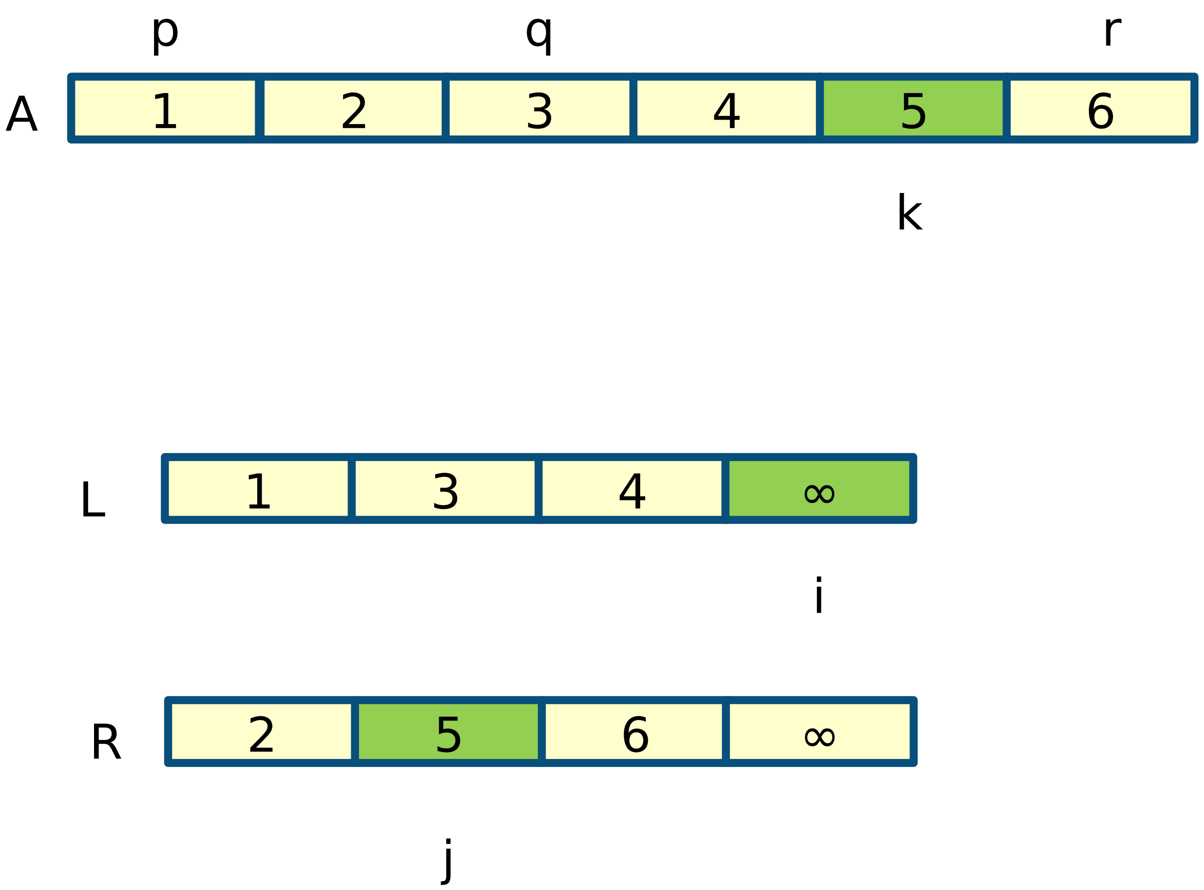
	p		q			r
A	1	2	3	2	5	6
				k		

L	1	3	4	∞
			i	

R	2	5	6	∞
		j		

```
MERGE(A,p,q,r)
  $n_1$ := q − p + 1
  $n_2$ := r − q
  copy A[p..q] to L[0..$n_1$]
  copy A[q+1..r] to R[0..$n_2$]
  L[$n_1$] := ∞
  R[$n_2$] := ∞
  i,j := 0
  for k = p to r
    if L[i] ≤ R[j]
      A[k] := L[i]
      i := i + 1
    else
      A[k] := R[j]
      j := j + 1
```

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



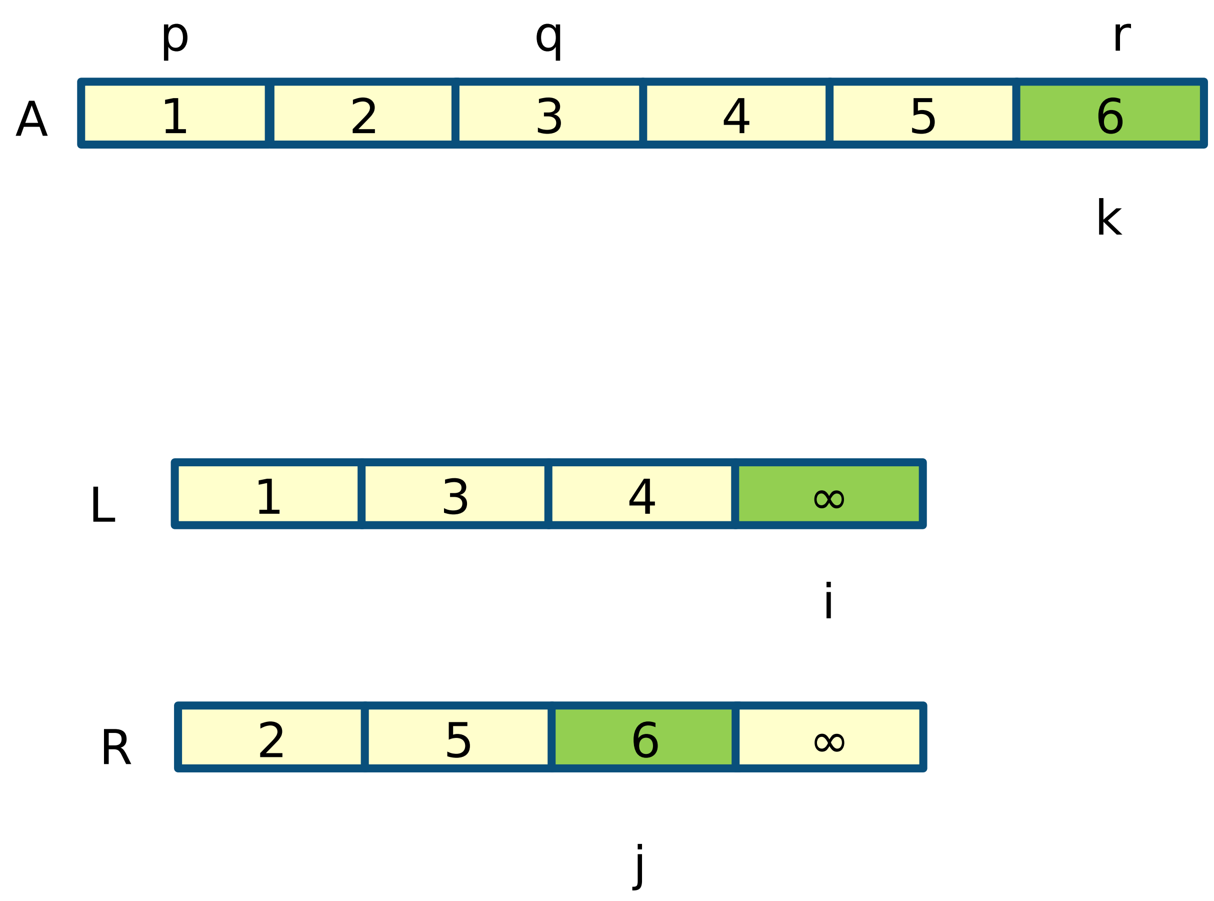
	p		q			r
A	1	2	3	4	5	6
					k	

L	1	3	4	∞
				i

R	2	5	6	∞
		j		

```
MERGE(A,p,q,r)
  $n_1$ := q – p + 1
  $n_2$ := r – q
  copy A[p..q] to L[0..$n_1$]
  copy A[q+1..r] to R[0..$n_2$]
  L[$n_1$] := ∞
  R[$n_2$] := ∞
  i,j := 0
  for k = p to r
    if L[i] ≤ R[j]
      A[k] := L[i]
      i := i + 1
    else
      A[k] := R[j]
      j := j + 1
```

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



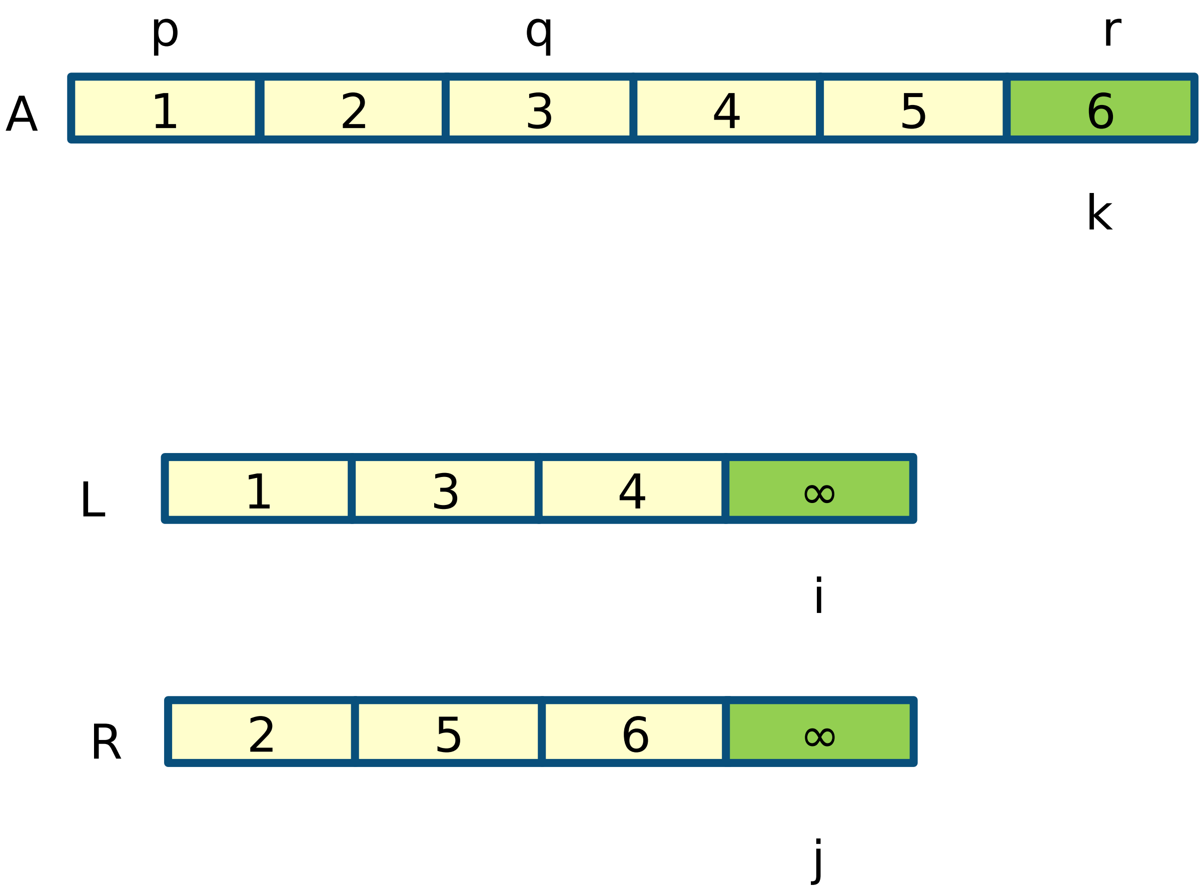
	p		q			r
A	1	2	3	4	5	6
						k

L	1	3	4	∞
				i

R	2	5	6	∞
			j	

```
MERGE(A,p,q,r)
  $n_1$ := q – p + 1
  $n_2$ := r – q
  copy A[p..q] to L[0..$n_1$]
  copy A[q+1..r] to R[0..$n_2$]
  L[$n_1$] := ∞
  R[$n_2$] := ∞
  i,j := 0
  for k = p to r
    if L[i] ≤ R[j]
      A[k] := L[i]
      i := i + 1
    else
      A[k] := R[j]
      j := j + 1
```

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



A					
1	2	3	4	5	6

L			
1	3	4	∞

R			
2	5	6	∞

```
MERGE(A,p,q,r)
  $n_1$ := q – p + 1
  $n_2$ := r – q
  copy A[p..q] to L[0..$n_1$]
  copy A[q+1..r] to R[0..$n_2$]
  L[$n_1$] := ∞
  R[$n_2$] := ∞
  i,j := 0
  for k = p to r
    if L[i] ≤ R[j]
      A[k] := L[i]
      i := i + 1
    else
      A[k] := R[j]
      j := j + 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**
  - It preserves the order of elements with the same sorting key
- **O(n) working memory requirement**
  - To store L and R
  - In-place version is possible, but stability is lost!

```
MERGE(A,p,q,r)
  $n_1$ := q – p + 1
  $n_2$ := r – q
  copy A[p..q] to L[0..$n_1$]
  copy A[q+1..r] to R[0..$n_2$]
  L[$n_1$] := ∞
  R[$n_2$] := ∞
  i,j := 0
  for k = p to r
    if L[i] ≤ R[j]
      A[k] := L[i]
      i := i + 1
    else
      A[k] := R[j]
      j := j + 1
```

MERGE-SORT

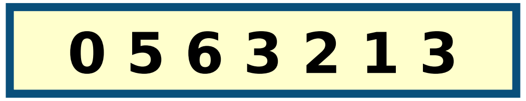
- **Input: Array A and two indexes p, r for A such that $p \le r$**
- **Output: sorted array A[p..r]**

```
MERGE-SORT(A,p,r)
  if p < r
    q := (p+r)/2
    MERGE-SORT(A,p,q)
    MERGE-SORT(A,q+1,r)
    MERGE(A,p,q,r)
```

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

Recursion tree

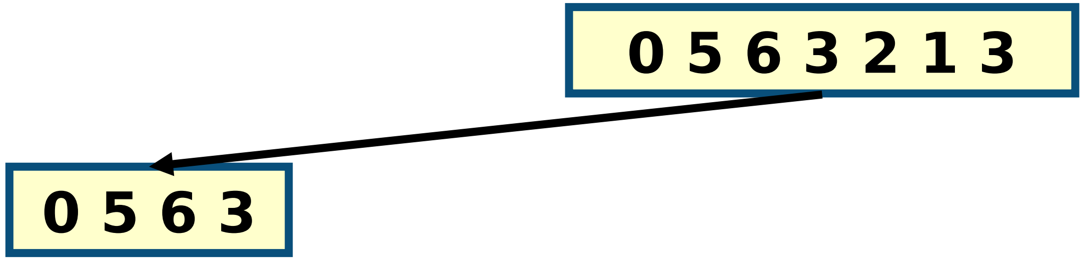
- **MERGE-SORT(A,0,6) with A=[0,5,6,3,2,1,3]**



0 5 6 3 2 1 3

Recursion tree

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



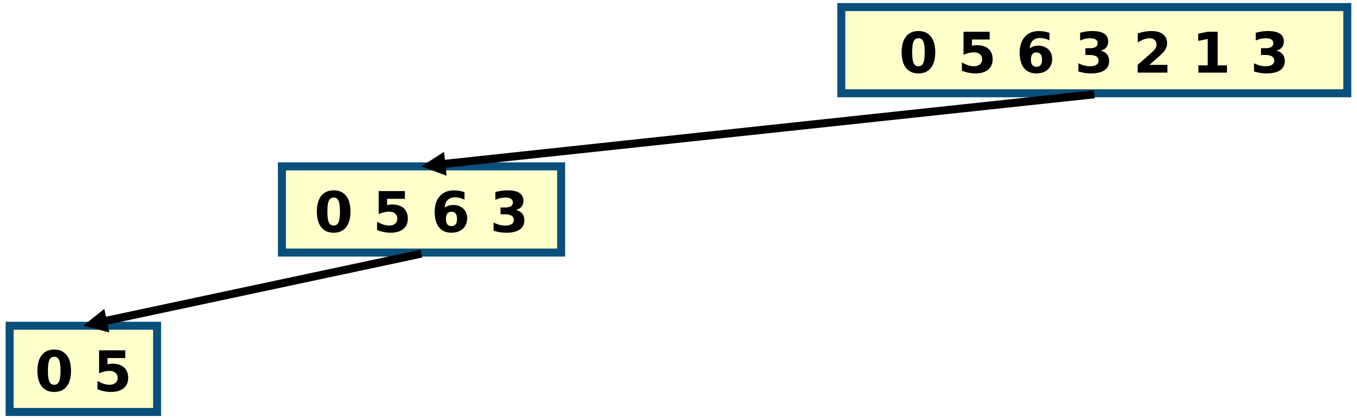
0 5 6 3 2 1 3

0 5 6 3

  - q = p+r/2 = 0+6/2 = 3

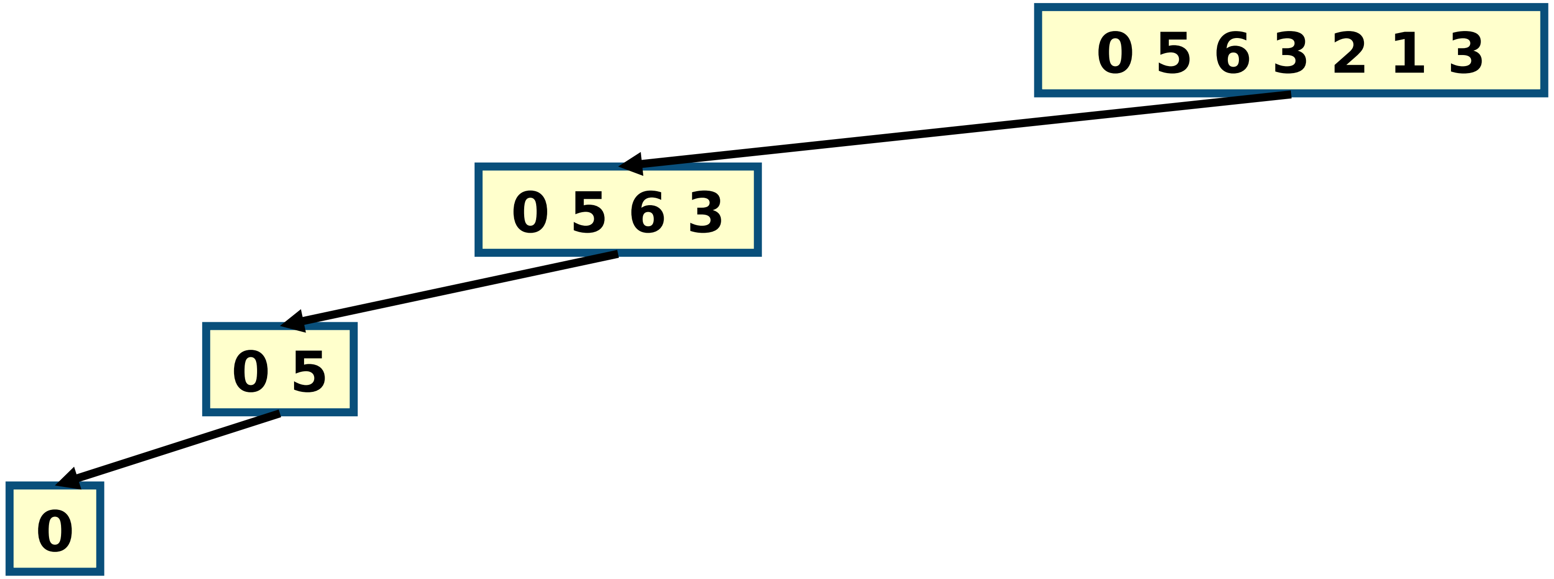
Recursion tree

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



Recursion tree

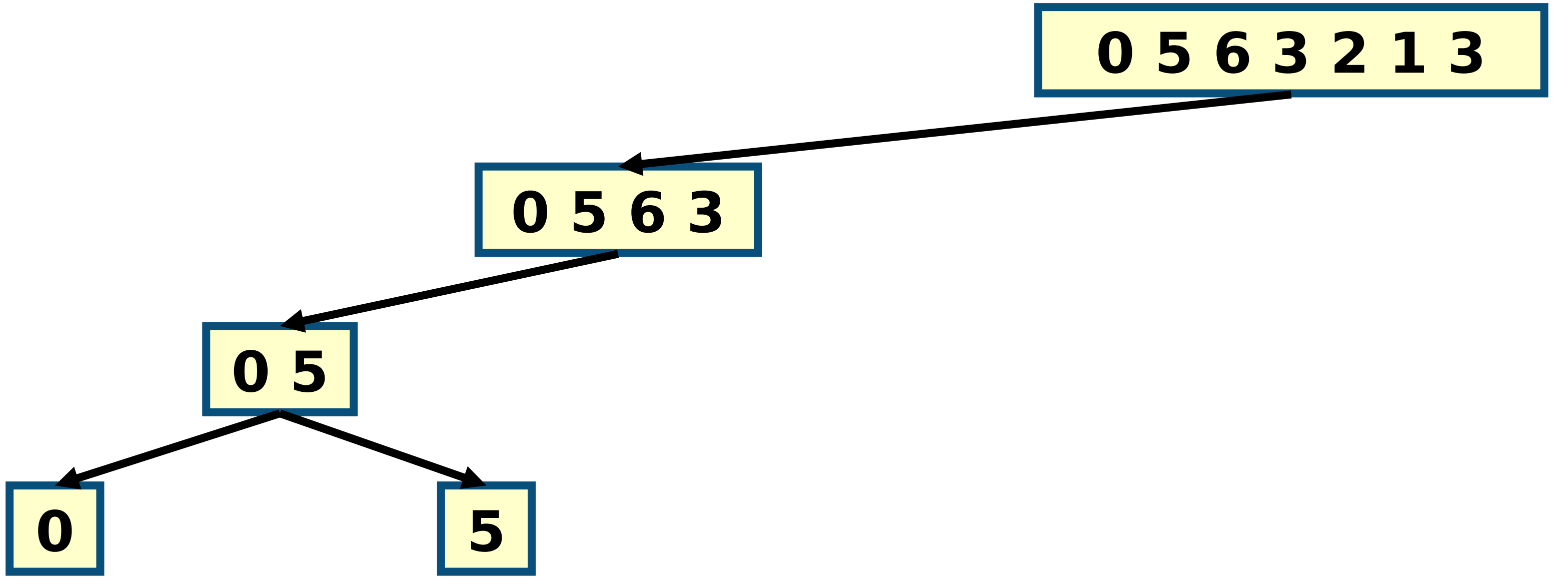
- **MERGE-SORT(A,0,6) with A=[0,5,6,3,2,1,3]**



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

Recursion tree

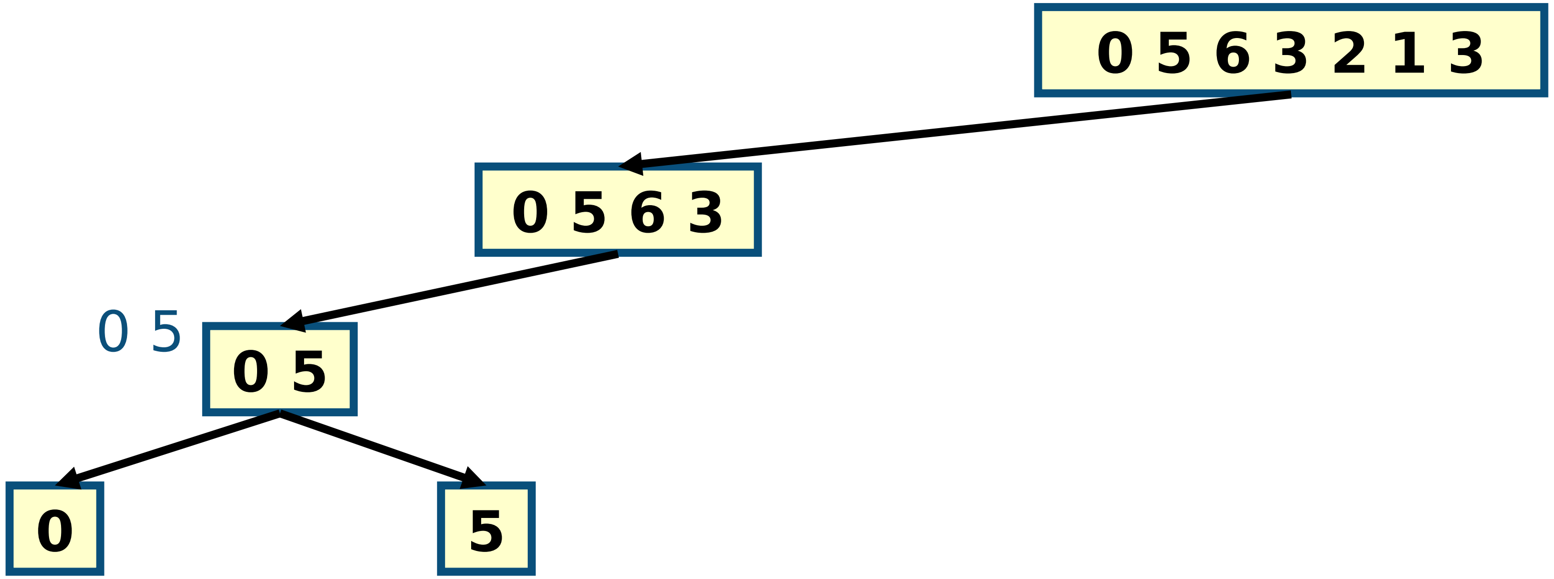
- **MERGE-SORT(A,0,6) with A=[0,5,6,3,2,1,3]**



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

Recursion tree

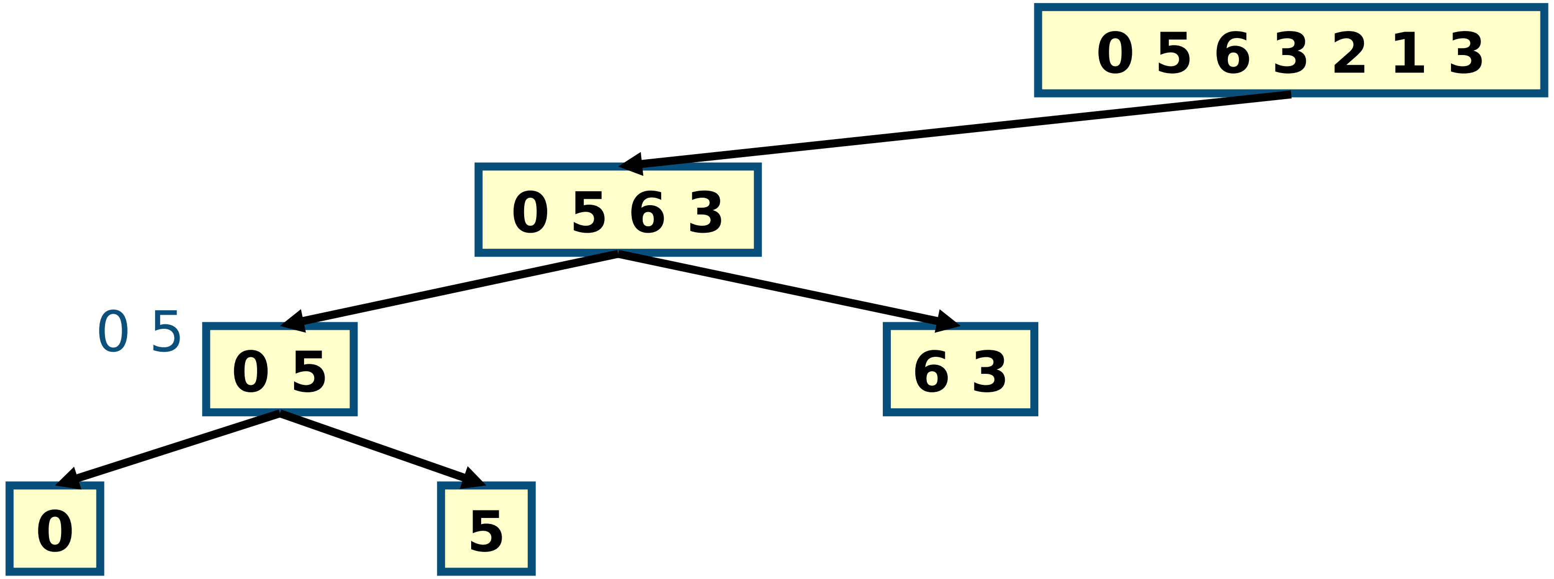
- MERGE-SORT(A,0,6) with A=[0,5,6,3,2,1,3]



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

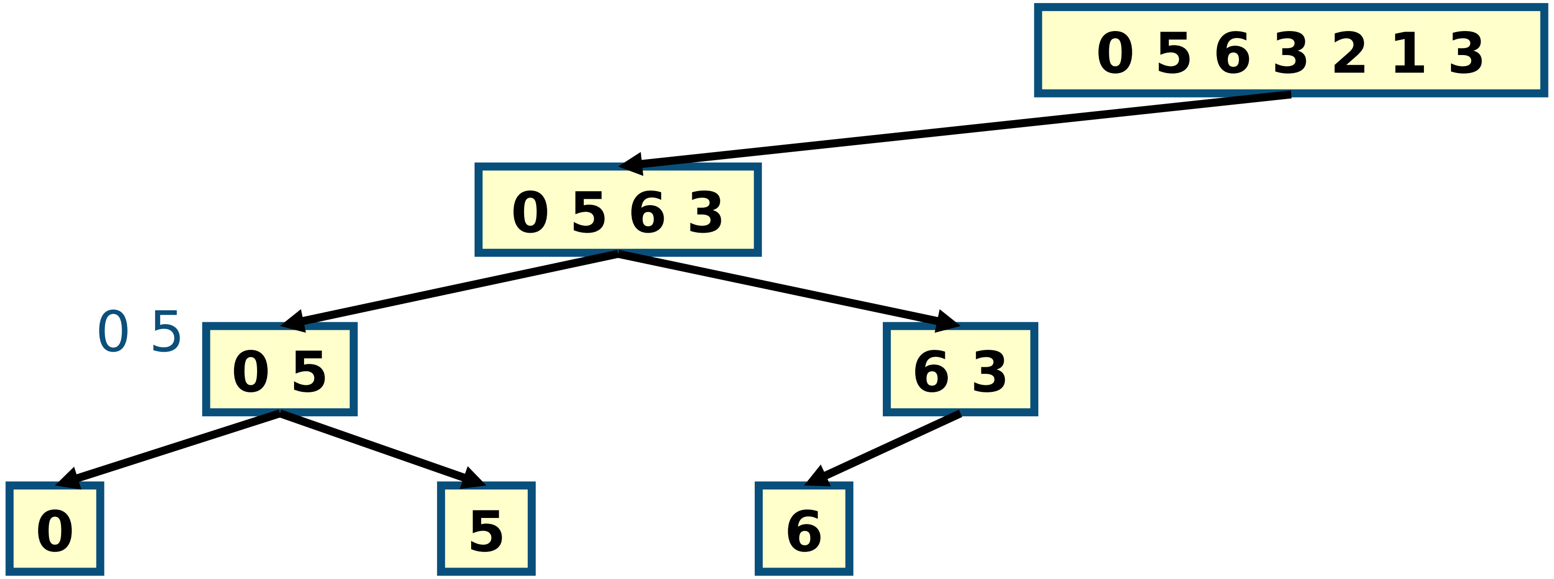
Recursion tree

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



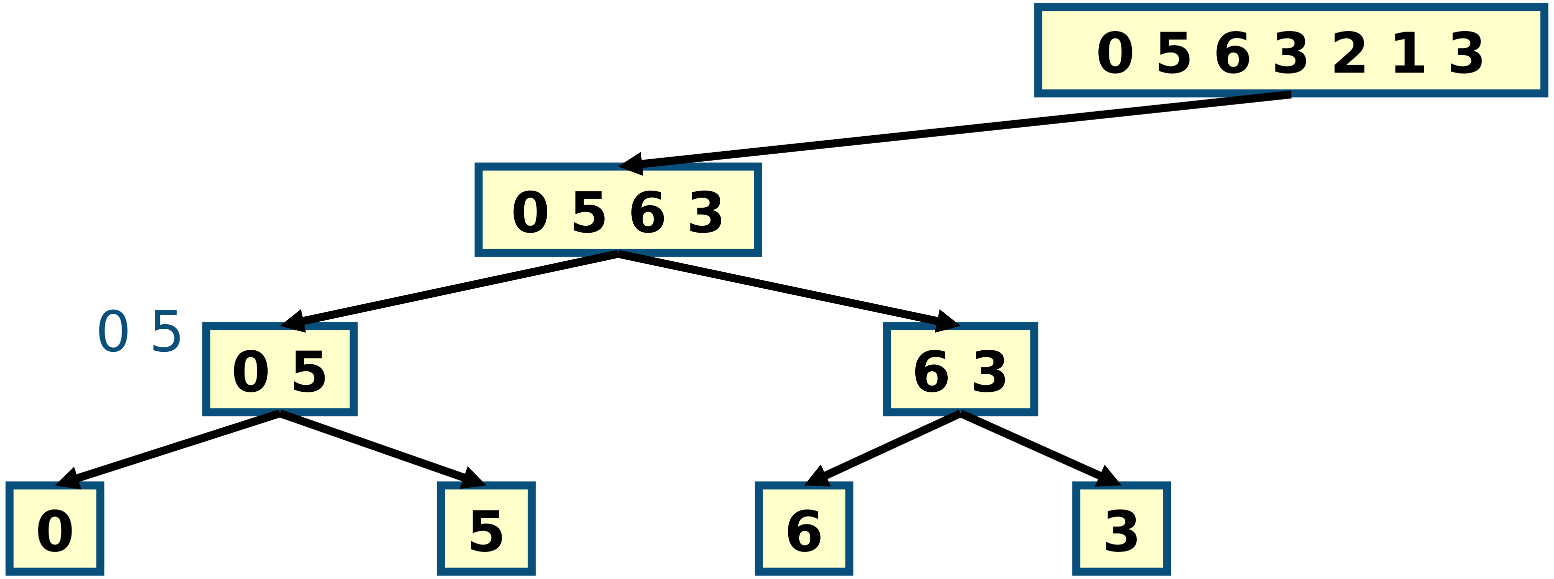
Recursion tree

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



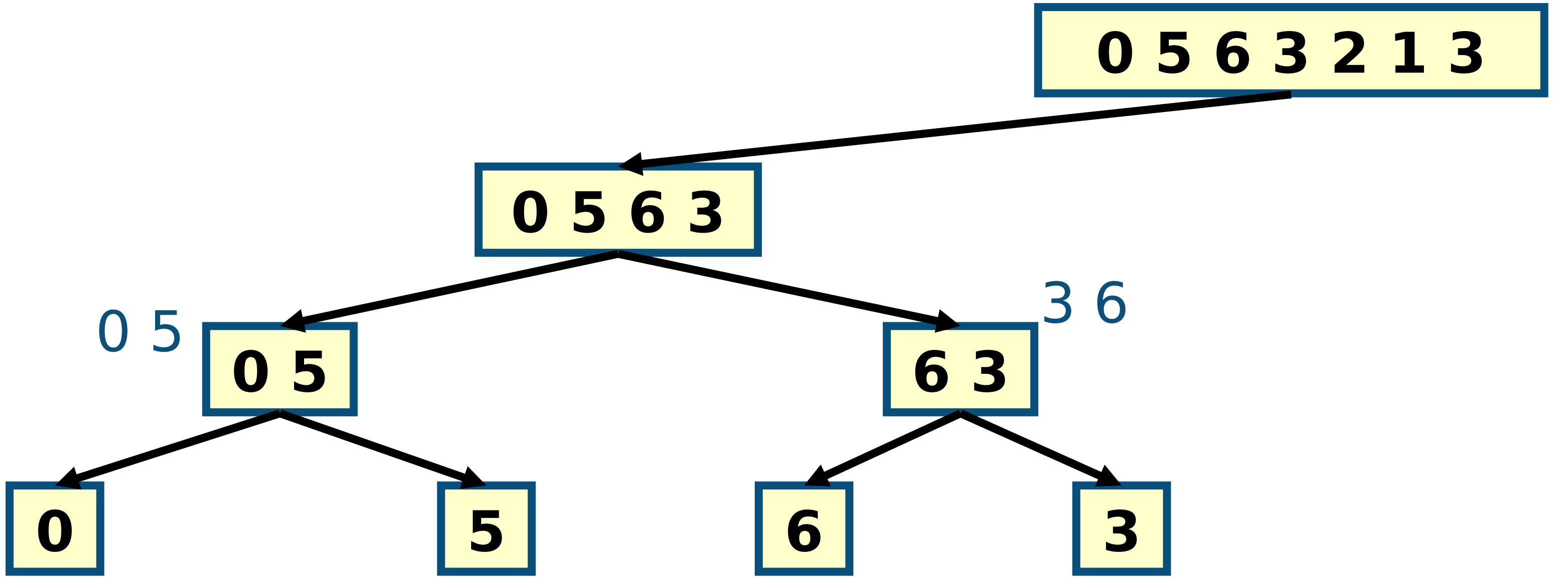
Recursion tree

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



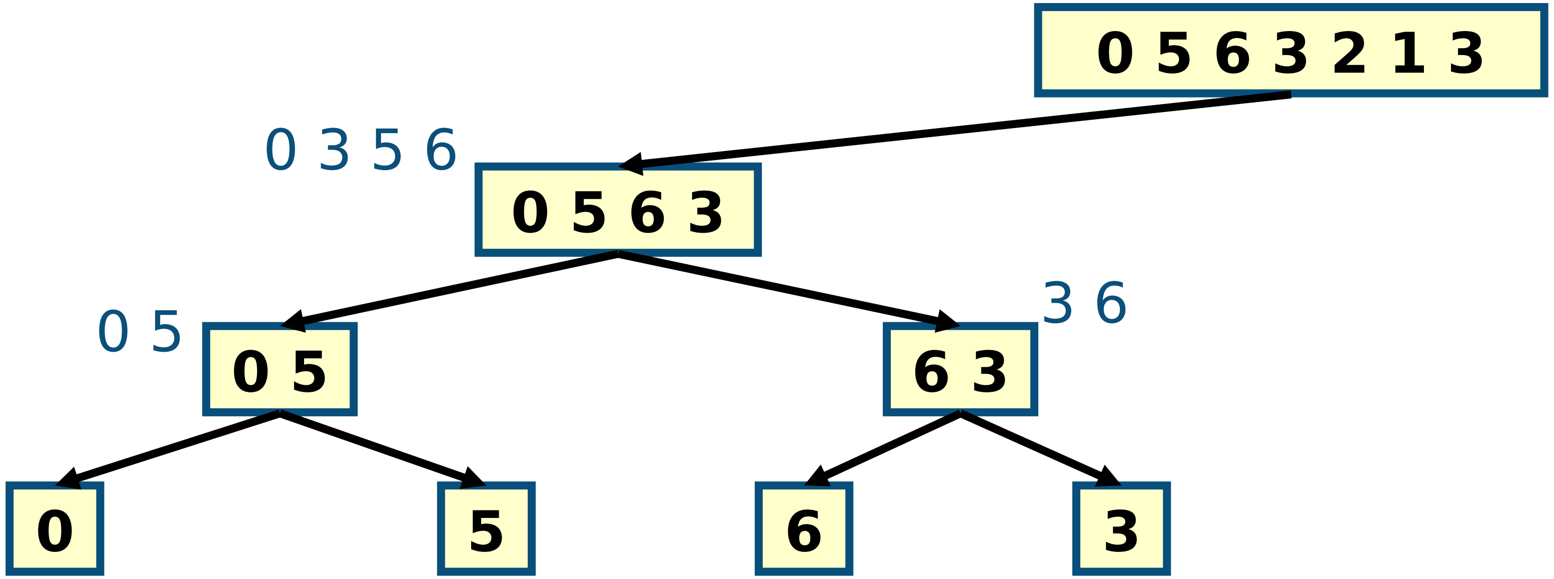
Recursion tree

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



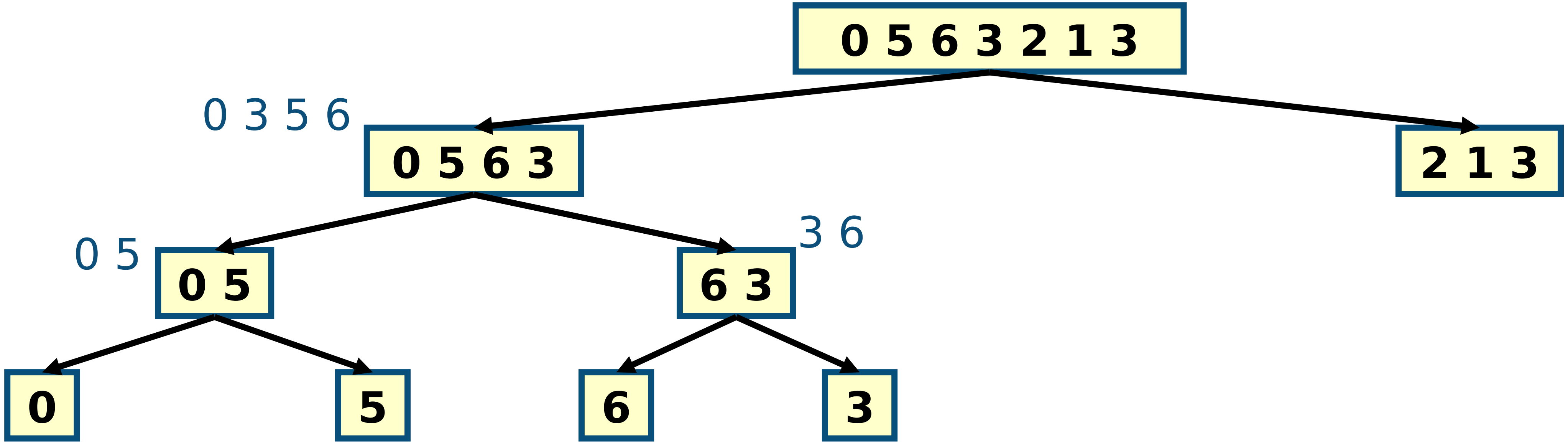
Recursion tree

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



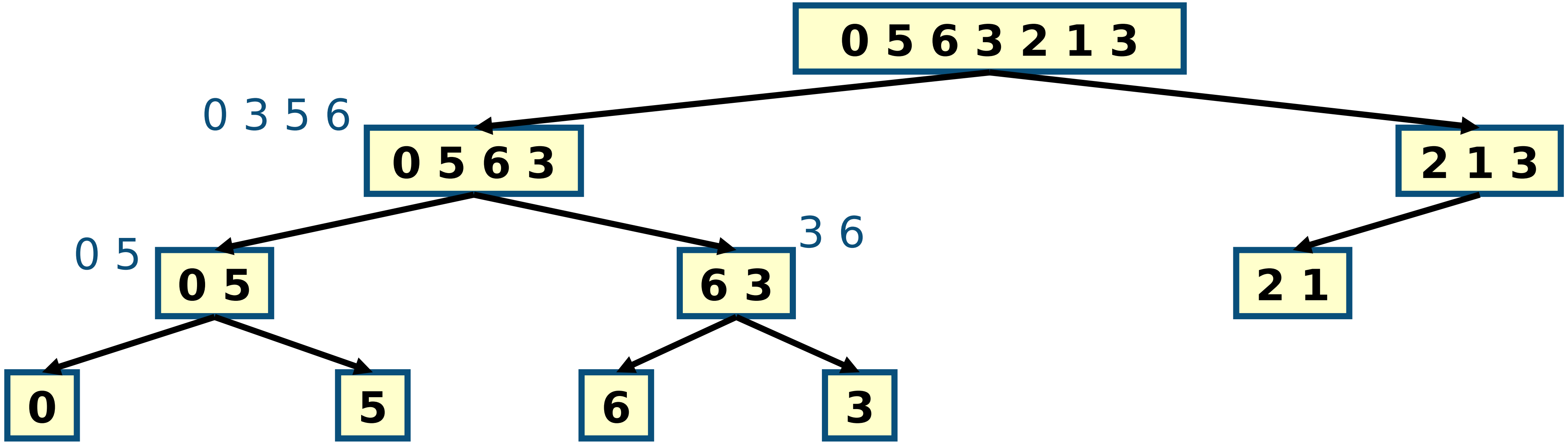
Recursion tree

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



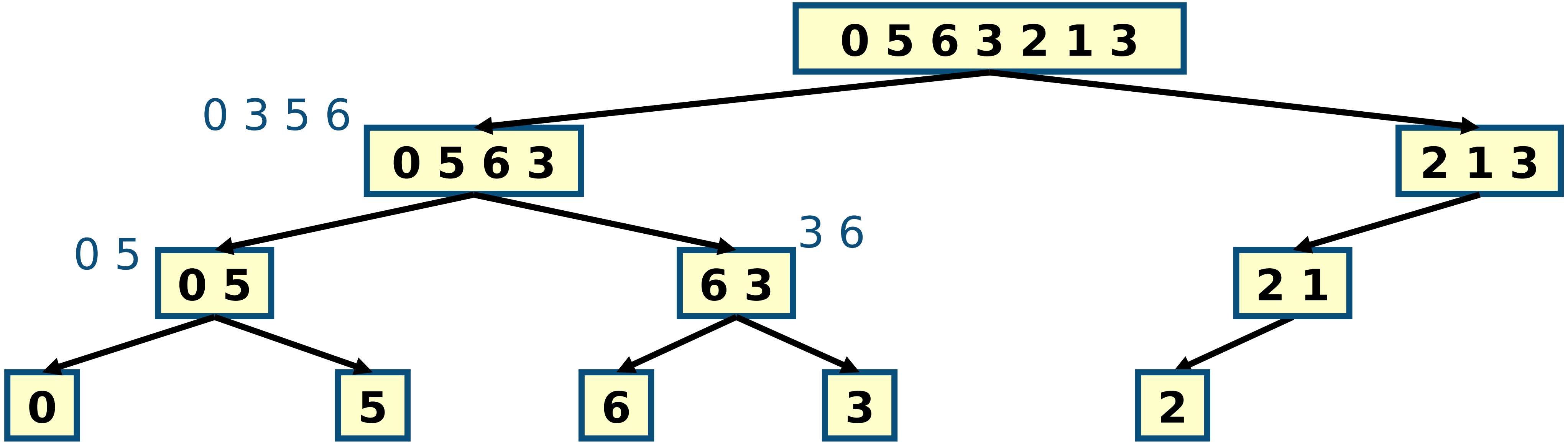
Recursion tree

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



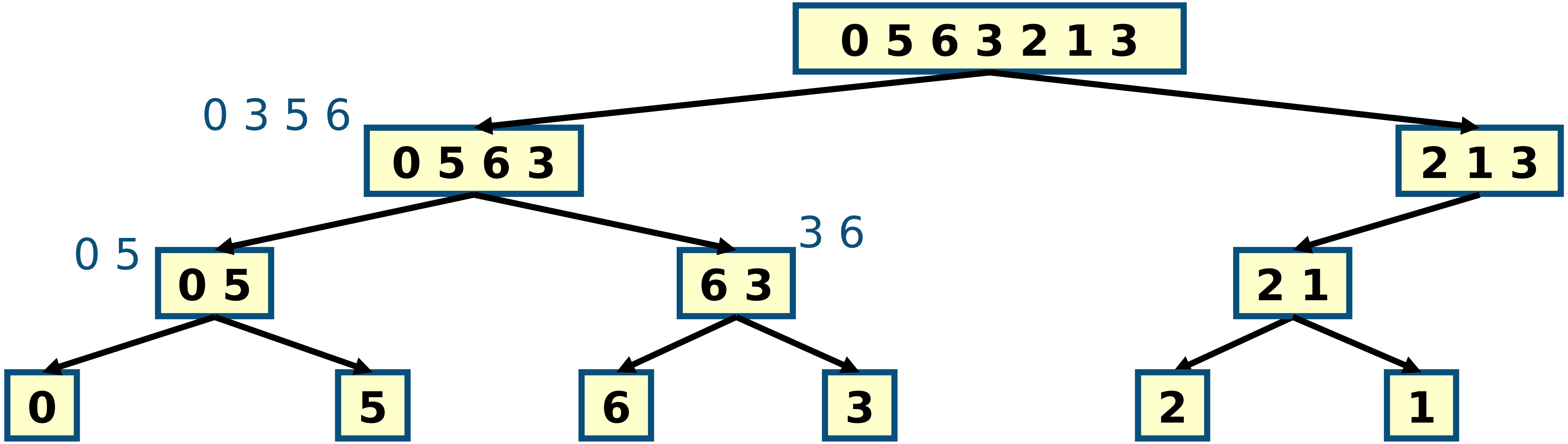
Recursion tree

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



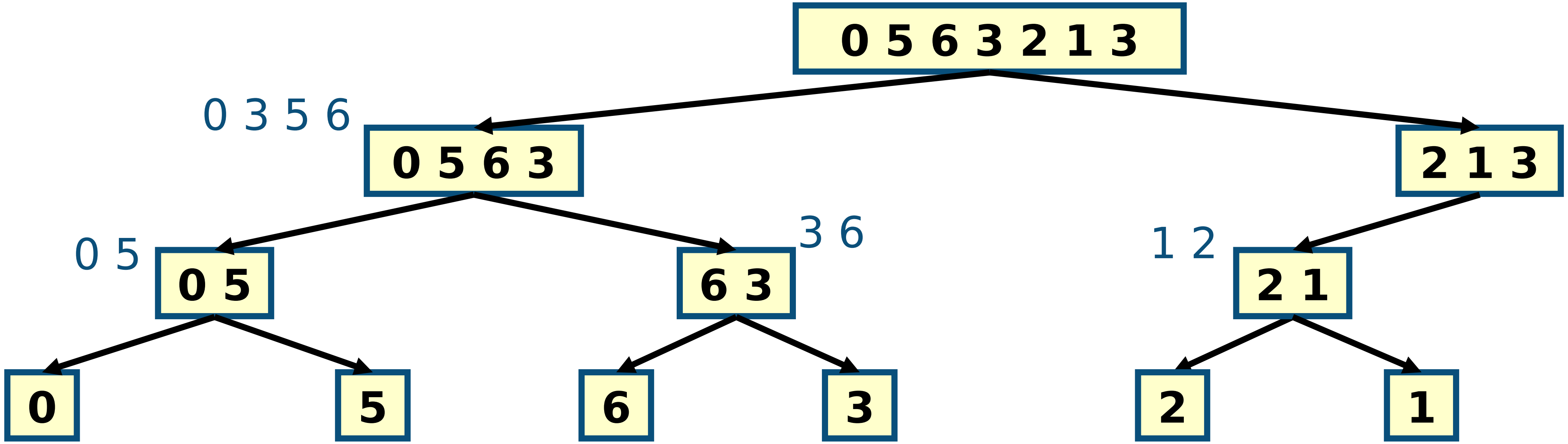
Recursion tree

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



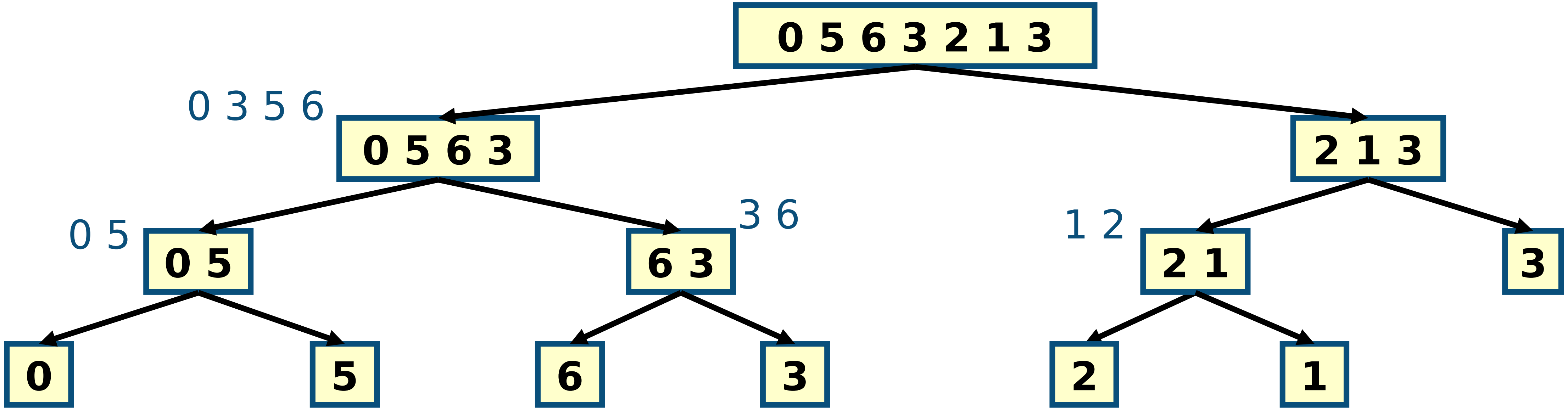
Recursion tree

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



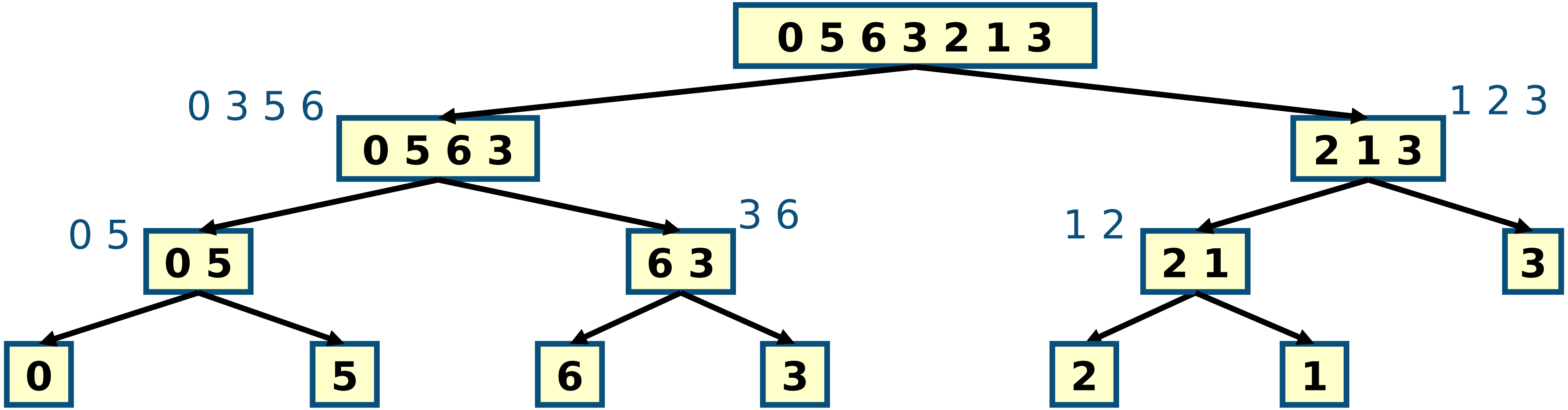
Recursion tree

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



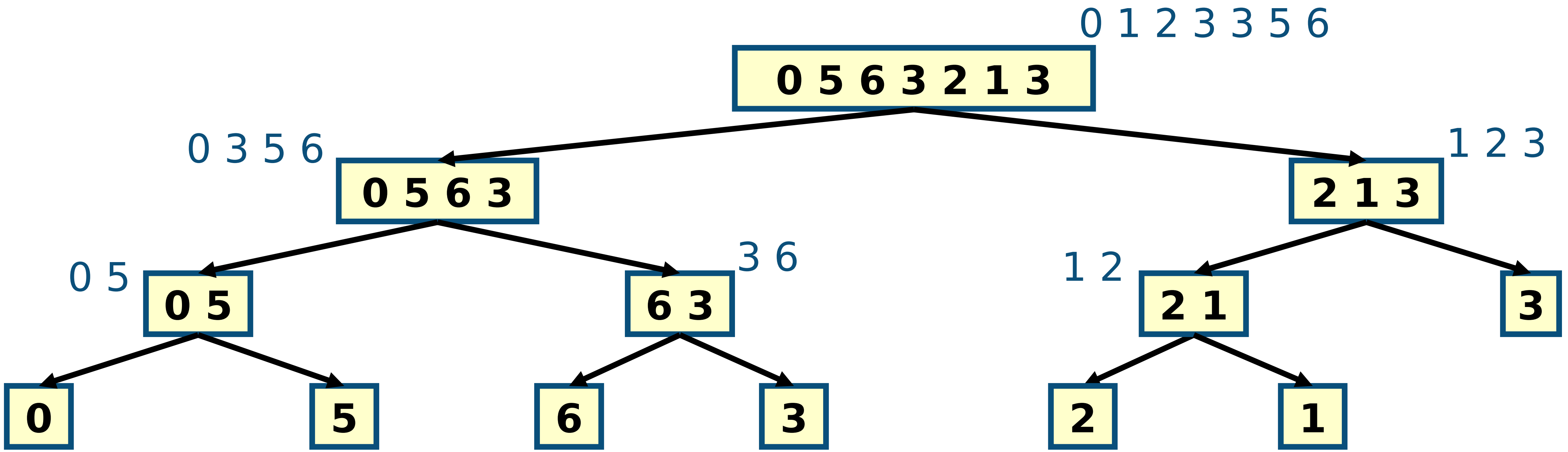
Recursion tree

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



Recursion tree

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



  - Termination

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 see how to compute that in the next lecture

Improvements on standard MERGE-SORT

- **In-place (through in-place MERGE)**
- **Bottom-up (iterative)**
  - Organise the merges so that all the merges of arrays of length 2 are done in one pass
  - Then do a second pass to merge those arrays in pairs, and so forth
  - Continue until we do a merge that encompasses the whole array
- **Use INSERTION-SORT on small instances**
  - Cut-off typically $5 \leq n \leq 20$

- **You will implement some of those in Lab 2**

Summary

- **MERGE**
  - Properties
- **MERGE-SORT**
  - Properties
  - Improvements