More recursive algorithms

1

We have introduced the recursion paradigm for the binary tree data structure. Recursion is a general algorithm design approach, and here are additional examples.

Computing factorial

```
Input:
   n // factorial function
argument
Output:
   // n!
Factorial(n)
   if n == 1
      return 1
   endif
   return n*Factorial(n-1)
endMinimum
```

2

Factorial of n is the product of the first n numbers. Factorial of n can be computed recursively as factorial of n-1 times n.

The termination condition is if n==1 then return 1. (Note that the == sign is used to emphasize that we test for equality between the left and the right quantities, and we do not assign 1 to n.) This is a termination condition because when it is true there is no more recursive call made. The chain of recursive calls is interrupted.

The recursive call is the call to Factorial(n-1). The algorithm uses itself as a sub-algorithm.

Finding minimum in array

```
Input:
   A // array of integers
   n // number of elements in array (array size)
Output:
   Min // value of element with smallest value
Minimum(A, n) // non-recursive version
   Min = A[0] // initialize min. as first element
   for i = 1 to n-1 // look at remaining elements
      if A[i] < Min then
         Min = A[i]
      endif
   endfor
   return Min
endMinimum
```

```
Input:
   A // array of integers
   n // number of elements in array (array size)
   i0 // consider elements from i0 onwards
Output:
   Min // value of element with smallest value
MinR(A, n, i0) // recursive version
   if i0 == n-1 // last element
      return A[i0]
   endif
   tmp = MinR(A, n, i0+1) // min(A[i0+1], …, A[n-1])
   // min(A[i0], …, A[n-1]) is min(A[i0], tmp)
   if A[i0] < tmp
      return A[i0]
   else
      return tmp
endMinR
```

3

Left: conventional (non-recursive) minimum algorithm.
Right: recursive algorithm for finding minimum.

The recursive algorithm is based on the idea that the minimum in an array is the smaller of:

- The first element
- The minimum of remaining elements

MinR is first called as MinR(A, n, 0), since we want the minimum beginning from the first element, which has index 0.
Then the minimum of elements 2 to n-1 is found by calling MinR recursively with arguments (A, n, 1) and saved into variable tmp.
Finally, the overall minimum being returned is the smaller of the first element or tmp.

MinR(A, n, 2) makes another recursive call, MinR(A, n, 3) and so on.

When i0 becomes n-1, no more recursive call is made. “If i0 == n-1 then return A[i0]” is the termination condition.

Recursive sorting: *merge sort*

```
Input:
   A // array of integers
   l // the index of the first element of A to be considered
   r // the index of the last element of A to be considered
Output:
   B // array with elements of A in ascending order
MergeSortR(A, l, r) // initial call MergeSortR(A, 0, n-1)
   if l == r // termination condition (subarray with 1 el.)
      B[0] = A[l]
      return B
   endif
   m = (l+r)/2 // midpoint
   Bl = MergeSortR(A, l, m) // sort left subarray
   Br = MergeSortR(A, m+1, r) // sort right subarray
   B = MergeSorted(Bl, Br) // merge subarrays
   return B // return merged subarrays
endMergeSortR
```

4

Here is a recursive sorting algorithm. The idea is to sort the array by splitting the array in half, sorting each half independently, and then merging the two sorted halves in the overall sorted algorithm.

The recursion keeps splitting the array until the array has a single element. When l equals r, there is a single element. That is the termination condition. When an array has a single element, there is no sorting to be done. An array with a single element is sorted by default.

Once the left and right sub-arrays are sorted, the two sorted sub-arrays are merged with a sub-algorithm called MergeSorted. (Note the difference in name between MergeSortR and MergeSorted.) For now let's ignore the details of MergeSorted. In other words, let's just assume that we have that algorithm and it can merge two sorted arrays, and let's not worry about how MergeSorted actually works.

Merge sort trace

```
Input:
   A // array of integers
   l // the index of the first element of A to be considered
   r // the index of the last element of A to be considered
Output:
   B // array with elements of A in ascending order
MergeSortR(A, l, r) // initial call MergeSortR(A, 0, n-1)
   if l == r // termination condition (subarray with 1 el.)
      B[0] = A[l]
      return B
   endif
   m = (l+r)/2 // midpoint
   Bl = MergeSortR(A, l, m) // sort left subarray
   Br = MergeSortR(A, m+1, r) // sort right subarray
   B = MergeSorted(Bl, Br) // merge subarrays
   return B // return merged subarrays
endMergeSortR
```

A:

0	*1*	*2*	*3*	*4*	*5*
9	1	0	8	2	4

Call	Return
MergeSortR(A, 0, 5)	
MergeSortR(A, 0, 2)	
MergeSortR(A, 0, 1)	
MergeSortR(A, 0, 0)	{9}

5

Here is a trace of the algorithm on the given array A.

What makes the difference between the various recursive calls of MergeSortR is the pair of indices l and r, which delimit the sub-array on which we currently work on.

At first (0, 5) means we work on the entire array. We are not yet down to a single element array, so let's sort the first half (0, 2). Note that the result of the division (l+r)/2 is truncated.

To sort (0, 2) we have to first sort (0, 1).

To sort (0, 1) we have to first sort (0, 0).

Sorting (0, 0) is trivial, it's a single element array, it's already sorted, no more recursive calls (i.e. the termination condition is met).

Merge sort trace

```
Input:
   A // array of integers
   l // the index of the first element of A to be considered
   r // the index of the last element of A to be considered
Output:
   B // array with elements of A in ascending order
MergeSortR(A, l, r) // initial call MergeSortR(A, 0, n-1)
   if l == r // termination condition (subarray with 1 el.)
      B[0] = A[l]
      return B
   endif
   m = (l+r)/2 // midpoint
   Bl = MergeSortR(A, l, m) // sort left subarray
   Br = MergeSortR(A, m+1, r) // sort right subarray
   B = MergeSorted(Bl, Br) // merge subarrays
   return B // return merged subarrays
endMergeSortR
```

A:

0	*1*	*2*	*3*	*4*	*5*
9	1	0	8	2	4

Call	Return
MergeSortR(A, 0, 5)	
MergeSortR(A, 0, 2)	
MergeSortR(A, 0, 1)	
MergeSortR(A, 0, 0)	{9}
MergeSortR(A, 1, 1)	{1}

6

Now we have sorted the left half of (0, 1). Let's sort the right half (1, 1). That's also trivial.

Merge sort trace

```
Input:
   A // array of integers
   l // the index of the first element of A to be considered
   r // the index of the last element of A to be considered
Output:
   B // array with elements of A in ascending order
MergeSortR(A, l, r) // initial call MergeSortR(A, 0, n-1)
   if l == r // termination condition (subarray with 1 el.)
      B[0] = A[l]
      return B
   endif
   m = (l+r)/2 // midpoint
   Bl = MergeSortR(A, l, m) // sort left subarray
   Br = MergeSortR(A, m+1, r) // sort right subarray
   B = MergeSorted(Bl, Br) // merge subarrays
   return B // return merged subarrays
endMergeSortR
```

A:

0	*1*	*2*	*3*	*4*	*5*
9	1	0	8	2	4

Call	Return
MergeSortR(A, 0, 5)	
MergeSortR(A, 0, 2)	
MergeSortR(A, 0, 1)	{1, 9}
MergeSortR(A, 0, 0)	{9}
MergeSortR(A, 1, 1)	{1}

7

Now we are at the recursive call (0, 1) and we have Bl = {9} and Br = {1}. MergeSorted will create a B of {1, 9}.

Merge sort trace

```
Input:
   A // array of integers
   l // the index of the first element of A to be considered
   r // the index of the last element of A to be considered
Output:
   B // array with elements of A in ascending order
MergeSortR(A, l, r) // initial call MergeSortR(A, 0, n-1)
   if l == r // termination condition (subarray with 1 el.)
      B[0] = A[l]
      return B
   endif
   m = (l+r)/2 // midpoint
   Bl = MergeSortR(A, l, m) // sort left subarray
   Br = MergeSortR(A, m+1, r) // sort right subarray
   B = MergeSorted(Bl, Br) // merge subarrays
   return B // return merged subarrays
endMergeSortR
```

A:

0	*1*	*2*	*3*	*4*	*5*
9	1	0	8	2	4

Call	Return
MergeSortR(A, 0, 5)	
MergeSortR(A, 0, 2)	
MergeSortR(A, 0, 1)	{1, 9}
MergeSortR(A, 2, 2)	{0}

8

To sort (0, 2), we first sorted (0, 1), which is now done, we have it as {1, 9}. Next we need to sort the right half, (2, 2), which is straightforward since a single element.

Merge sort trace

```
Input:
   A // array of integers
   l // the index of the first element of A to be considered
   r // the index of the last element of A to be considered
Output:
   B // array with elements of A in ascending order
MergeSortR(A, l, r) // initial call MergeSortR(A, 0, n-1)
   if l == r // termination condition (subarray with 1 el.)
      B[0] = A[l]
      return B
   endif
   m = (l+r)/2 // midpoint
   Bl = MergeSortR(A, l, m) // sort left subarray
   Br = MergeSortR(A, m+1, r) // sort right subarray
   B = MergeSorted(Bl, Br) // merge subarrays
   return B // return merged subarrays
endMergeSortR
```

A:

0	*1*	*2*	*3*	*4*	*5*
9	1	0	8	2	4

Call	Return
MergeSortR(A, 0, 5)	
MergeSortR(A, 0, 2)	{0, 1, 9}
MergeSortR(A, 0, 1)	{1, 9}
MergeSortR(A, 2, 2)	{0}

9

Now we are ready to finalize the sorted sequence (0, 2), by merging the sorted (0, 1) and (2, 2), using MergeSorted, and we get {0, 1, 9}. Remember, for now, we do not worry about how MergeSorted works, it just does.

Merge sort trace

```
Input:
   A // array of integers
   l // the index of the first element of A to be considered
   r // the index of the last element of A to be considered
Output:
   B // array with elements of A in ascending order
MergeSortR(A, l, r) // initial call MergeSortR(A, 0, n-1)
   if l == r // termination condition (subarray with 1 el.)
      B[0] = A[l]
      return B
   endif
   m = (l+r)/2 // midpoint
   Bl = MergeSortR(A, l, m) // sort left subarray
   Br = MergeSortR(A, m+1, r) // sort right subarray
   B = MergeSorted(Bl, Br) // merge subarrays
   return B // return merged subarrays
endMergeSortR
```

A:

0	*1*	*2*	*3*	*4*	*5*
9	1	0	8	2	4

Call	Return
MergeSortR(A, 0, 5)	
MergeSortR(A, 0, 2)	{0, 1, 9}

10

We are done sorting the left half of (0, 5). Now we need to sort the right half of (0, 5), (3, 5), which is done similarly.

Merge sort trace

```
Input:
   A // array of integers
   l // the index of the first element of A to be considered
   r // the index of the last element of A to be considered
Output:
   B // array with elements of A in ascending order
MergeSortR(A, l, r) // initial call MergeSortR(A, 0, n-1)
   if l == r // termination condition (subarray with 1 el.)
      B[0] = A[l]
      return B
   endif
   m = (l+r)/2 // midpoint
   Bl = MergeSortR(A, l, m) // sort left subarray
   Br = MergeSortR(A, m+1, r) // sort right subarray
   B = MergeSorted(Bl, Br) // merge subarrays
   return B // return merged subarrays
endMergeSortR
```

A:

0	*1*	*2*	*3*	*4*	*5*
9	1	0	8	2	4

Call	Return
MergeSortR(A, 0, 5)	
MergeSortR(A, 0, 2)	{0, 1, 9}
MergeSortR(A, 3, 5)	

11

Merge sort trace

```
Input:
   A // array of integers
   l // the index of the first element of A to be considered
   r // the index of the last element of A to be considered
Output:
   B // array with elements of A in ascending order
MergeSortR(A, l, r) // initial call MergeSortR(A, 0, n-1)
   if l == r // termination condition (subarray with 1 el.)
      B[0] = A[l]
      return B
   endif
   m = (l+r)/2 // midpoint
   Bl = MergeSortR(A, l, m) // sort left subarray
   Br = MergeSortR(A, m+1, r) // sort right subarray
   B = MergeSorted(Bl, Br) // merge subarrays
   return B // return merged subarrays
endMergeSortR
```

A:

0	*1*	*2*	*3*	*4*	*5*
9	1	0	8	2	4

Call	Return
MergeSortR(A, 0, 5)	
MergeSortR(A, 0, 2)	{0, 1, 9}
MergeSortR(A, 3, 5)	
MergeSortR(A, 3, 4)	

12

Merge sort trace

```
Input:
   A // array of integers
   l // the index of the first element of A to be considered
   r // the index of the last element of A to be considered
Output:
   B // array with elements of A in ascending order
MergeSortR(A, l, r) // initial call MergeSortR(A, 0, n-1)
   if l == r // termination condition (subarray with 1 el.)
      B[0] = A[l]
      return B
   endif
   m = (l+r)/2 // midpoint
   Bl = MergeSortR(A, l, m) // sort left subarray
   Br = MergeSortR(A, m+1, r) // sort right subarray
   B = MergeSorted(Bl, Br) // merge subarrays
   return B // return merged subarrays
endMergeSortR
```

A:

0	*1*	*2*	*3*	*4*	*5*
9	1	0	8	2	4

Call	Return
MergeSortR(A, 0, 5)	
MergeSortR(A, 0, 2)	{0, 1, 9}
MergeSortR(A, 3, 5)	
MergeSortR(A, 3, 4)	
MergeSortR(A, 3, 3)	{8}

13

Merge sort trace

```
Input:
   A // array of integers
   l // the index of the first element of A to be considered
   r // the index of the last element of A to be considered
Output:
   B // array with elements of A in ascending order
MergeSortR(A, l, r) // initial call MergeSortR(A, 0, n-1)
   if l == r // termination condition (subarray with 1 el.)
      B[0] = A[l]
      return B
   endif
   m = (l+r)/2 // midpoint
   Bl = MergeSortR(A, l, m) // sort left subarray
   Br = MergeSortR(A, m+1, r) // sort right subarray
   B = MergeSorted(Bl, Br) // merge subarrays
   return B // return merged subarrays
endMergeSortR
```

A:

0	*1*	*2*	*3*	*4*	*5*
9	1	0	8	2	4

Call	Return
MergeSortR(A, 0, 5)	
MergeSortR(A, 0, 2)	{0, 1, 9}
MergeSortR(A, 3, 5)	
MergeSortR(A, 3, 4)	
MergeSortR(A, 3, 3)	{8}
MergeSortR(A, 4, 4)	{2}

Merge sort trace

```
Input:
   A // array of integers
   l // the index of the first element of A to be considered
   r // the index of the last element of A to be considered
Output:
   B // array with elements of A in ascending order
MergeSortR(A, l, r) // initial call MergeSortR(A, 0, n-1)
   if l == r // termination condition (subarray with 1 el.)
      B[0] = A[l]
      return B
   endif
   m = (l+r)/2 // midpoint
   Bl = MergeSortR(A, l, m) // sort left subarray
   Br = MergeSortR(A, m+1, r) // sort right subarray
   B = MergeSorted(Bl, Br) // merge subarrays
   return B // return merged subarrays
endMergeSortR
```

A:

0	*1*	*2*	*3*	*4*	*5*
9	1	0	8	2	4

Call	Return
MergeSortR(A, 0, 5)	
MergeSortR(A, 0, 2)	{0, 1, 9}
MergeSortR(A, 3, 5)	
MergeSortR(A, 3, 4)	{2, 8}
MergeSortR(A, 3, 3)	{8}
MergeSortR(A, 4, 4)	{2}

Merge sort trace

```
Input:
   A // array of integers
   l // the index of the first element of A to be considered
   r // the index of the last element of A to be considered
Output:
   B // array with elements of A in ascending order
MergeSortR(A, l, r) // initial call MergeSortR(A, 0, n-1)
   if l == r // termination condition (subarray with 1 el.)
      B[0] = A[l]
      return B
   endif
   m = (l+r)/2 // midpoint
   Bl = MergeSortR(A, l, m) // sort left subarray
   Br = MergeSortR(A, m+1, r) // sort right subarray
   B = MergeSorted(Bl, Br) // merge subarrays
   return B // return merged subarrays
endMergeSortR
```

A:

0	*1*	*2*	*3*	*4*	*5*
9	1	0	8	2	4

Call	Return
MergeSortR(A, 0, 5)	
MergeSortR(A, 0, 2)	{0, 1, 9}
MergeSortR(A, 3, 5)	
MergeSortR(A, 3, 4)	{2, 8}
MergeSortR(A, 5, 5)	{4}

Merge sort trace

```
Input:
   A // array of integers
   l // the index of the first element of A to be considered
   r // the index of the last element of A to be considered
Output:
   B // array with elements of A in ascending order
MergeSortR(A, l, r) // initial call MergeSortR(A, 0, n-1)
   if l == r // termination condition (subarray with 1 el.)
      B[0] = A[l]
      return B
   endif
   m = (l+r)/2 // midpoint
   Bl = MergeSortR(A, l, m) // sort left subarray
   Br = MergeSortR(A, m+1, r) // sort right subarray
   B = MergeSorted(Bl, Br) // merge subarrays
   return B // return merged subarrays
endMergeSortR
```

A:

0	*1*	*2*	*3*	*4*	*5*
9	1	0	8	2	4

Call	Return
MergeSortR(A, 0, 5)	
MergeSortR(A, 0, 2)	{0, 1, 9}
MergeSortR(A, 3, 5)	{2, 4, 8}
MergeSortR(A, 3, 4)	{2, 8}
MergeSortR(A, 5, 5)	{4}

17

Merge sort trace

```
Input:
   A // array of integers
   l // the index of the first element of A to be considered
   r // the index of the last element of A to be considered
Output:
   B // array with elements of A in ascending order
MergeSortR(A, l, r) // initial call MergeSortR(A, 0, n-1)
   if l == r // termination condition (subarray with 1 el.)
      B[0] = A[l]
      return B
   endif
   m = (l+r)/2 // midpoint
   Bl = MergeSortR(A, l, m) // sort left subarray
   Br = MergeSortR(A, m+1, r) // sort right subarray
   B = MergeSorted(Bl, Br) // merge subarrays
   return B // return merged subarrays
endMergeSortR
```

A:

0	*1*	*2*	*3*	*4*	*5*
9	1	0	8	2	4

Call	Return
MergeSortR(A, 0, 5)	{0, 1, 2, 4, 8, 9}
MergeSortR(A, 0, 2)	{0, 1, 9}
MergeSortR(A, 3, 5)	{2, 4, 8}

18

The final sorted array is obtained by merging the sorted halves (0, 2) and (3, 5).

Merge sort trace

```
Input:
   A // array of integers
   l // the index of the first element of A to be considered
   r // the index of the last element of A to be considered
Output:
   B // array with elements of A in ascending order
MergeSortR(A, l, r) // initial call MergeSortR(A, 0, n-1)
   if l == r // termination condition (subarray with 1 el.)
      B[0] = A[l]
      return B
   endif
   m = (l+r)/2 // midpoint
   Bl = MergeSortR(A, l, m) // sort left subarray
   Br = MergeSortR(A, m+1, r) // sort right subarray
   B = MergeSorted(Bl, Br) // merge subarrays
   return B // return merged subarrays
endMergeSortR
```

A:

0	*1*	*2*	*3*	*4*	*5*
9	1	0	8	2	4

Call	Return
MergeSortR(A, 0, 5)	{0, 1, 2, 4, 8, 9}

Recursive sorting: *merge sort*

```
Input:
   A // array of integers
   l // the index of the first element of A to be considered
   r // the index of the last element of A to be considered
Output:
   B // array with elements of A in ascending order
MergeSortR(A, l, r) // initial call MergeSortR(A, 0, n-1)
   if l == r // termination condition (subarray with 1 el.)
      B[0] = A[l]
      return B
   endif
   m = (l+r)/2 // midpoint
   Bl = MergeSortR(A, l, m) // sort left subarray
   Br = MergeSortR(A, m+1, r) // sort right subarray
   B = MergeSorted(Bl, Br) // merge subarrays
   return B // return merged subarrays
endMergeSortR
```

```
Input:
   A, n // sorted array of n integers
   B, m // sorted array of m integers
Output:
   C // sorted array with  elements of A and B
MergeSorted(A, n, B, m)
   i = 0; j = 0; k = 0
   while k < m+n
      if i < n and (j == m or A[i] < B[j])
         C[k] = A[i]; i = i+1
      else
         C[k] = B[j]; j = j+1
      endif
      k = k+1
   endwhile
endMergeSorted
```

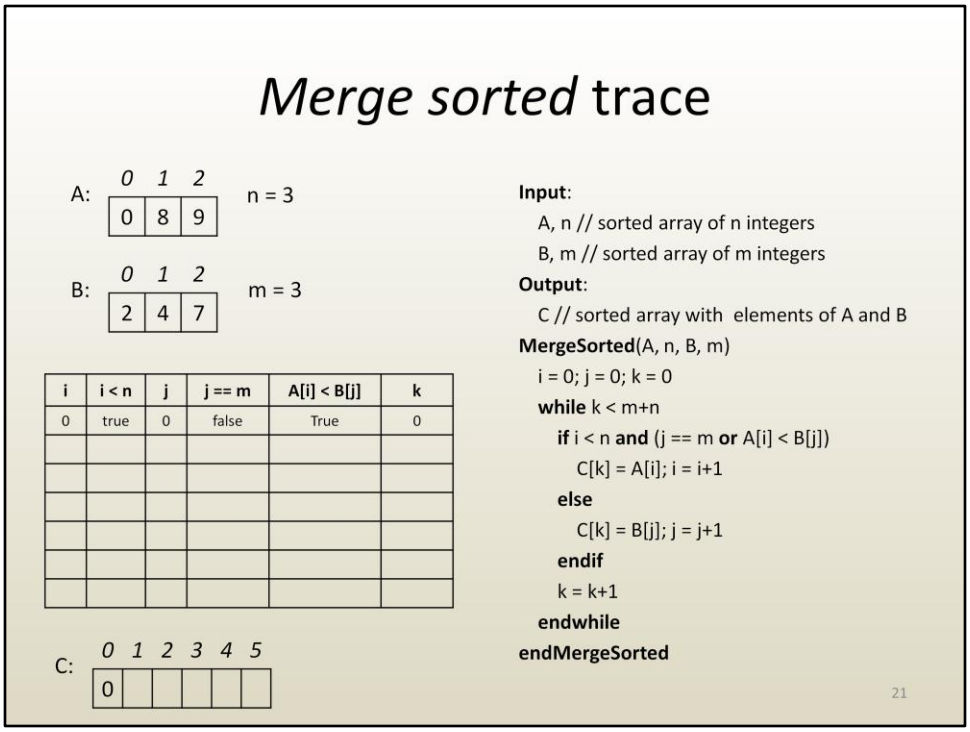
20

It is now time to worry about how MergeSorted works, see the pseudocode algorithm on the right.

MergeSorted works by traversing the two arrays A and B simultaneously. At each iteration of the while loop one element is selected from either A or B to be inserted into C. Indices i, j, and k keep track of the current element in A, B, and C, respectively.

If we have not yet reached the end of A, and we have reached the end of B or the current element in A is smaller than the current element in B, the current element from A is selected and copied into C. Otherwise, it is the current element of B that is copied into C. Since we want C to be sorted in ascending order, of course that we have to select the smaller of the two current elements. If one array has reached the end, the remaining array is copied.

The while loop condition is met when all elements have been copied into C, which implies a k of m+n.



Here is a trace of MergeSorted.

At the beginning we have not yet reached the end of A or of B. Since 0 is less than 2, 0 is copied. The index in A and C are advanced—not the index in B, since we have not yet copied element with index 0, i.e. 2.

Merge sorted trace

A:

0	*1*	*2*
0	8	9

n = 3

B:

0	*1*	*2*
2	4	7

m = 3

i	i < n	j	j == m	A[i] < B[j]	k
0	true	0	false	true	0
1	true	0	false	false	1

C:

0	*1*	*2*	*3*	*4*	*5*
0	2				

```
Input:
   A, n // sorted array of n integers
   B, m // sorted array of m integers
Output:
   C // sorted array with  elements of A and B
MergeSorted(A, n, B, m)
   i = 0; j = 0; k = 0
   while k < m+n
      if i < n and (j == m or A[i] < B[j])
         C[k] = A[i]; i = i+1
      else
         C[k] = B[j]; j = j+1
      endif
      k = k+1
   endwhile
endMergeSorted
```

22

Second iteration, between 8 and 2, it is 2 that gets copied into C.

Merge sorted trace

A:

0	*1*	*2*
0	8	9

n = 3

B:

0	*1*	*2*
2	4	7

m = 3

i	i < n	j	j == m	A[i] < B[j]	k
0	true	0	false	true	0
1	true	0	false	false	1
1	true	1	false	false	2

C:

0	*1*	*2*	*3*	*4*	*5*
0	2	4			

```
Input:
   A, n // sorted array of n integers
   B, m // sorted array of m integers
Output:
   C // sorted array with  elements of A and B
MergeSorted(A, n, B, m)
   i = 0; j = 0; k = 0
   while k < m+n
      if i < n and (j == m or A[i] < B[j])
         C[k] = A[i]; i = i+1
      else
         C[k] = B[j]; j = j+1
      endif
      k = k+1
   endwhile
endMergeSorted
```

23

Then 4.

Merge sorted trace

A:

0	*1*	*2*
0	8	9

n = 3

B:

0	*1*	*2*
2	4	7

m = 3

i	i < n	j	j == m	A[i] < B[j]	k
0	true	0	false	true	0
1	true	0	false	false	1
1	true	1	false	false	2
1	true	2	false	false	3

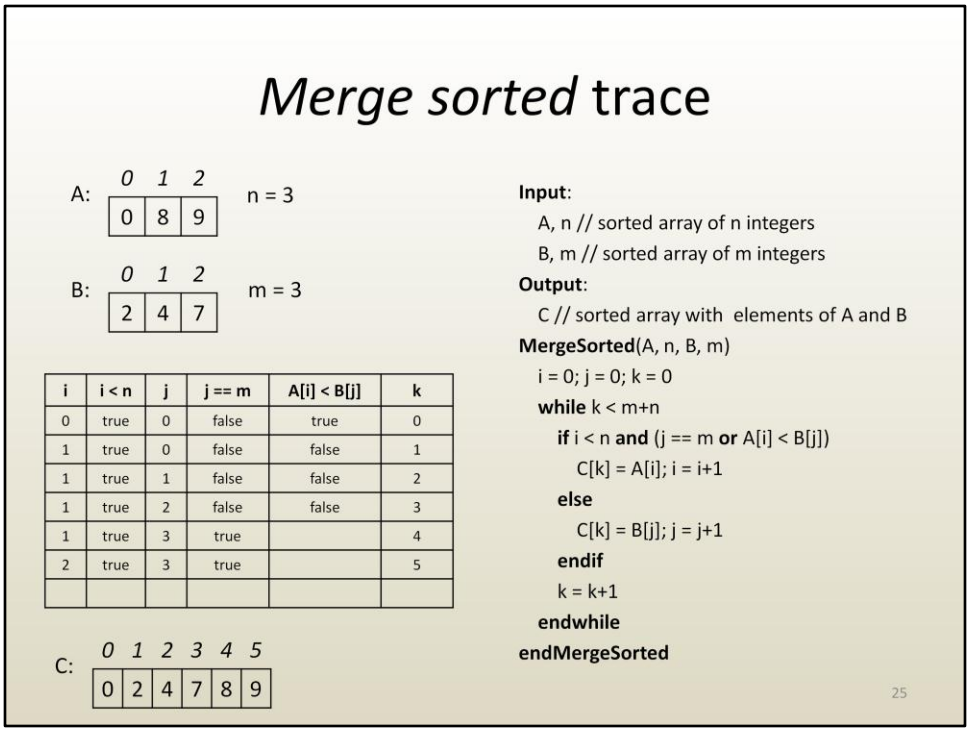
C:

0	*1*	*2*	*3*	*4*	*5*
0	2	4	7		

```
Input:
   A, n // sorted array of n integers
   B, m // sorted array of m integers
Output:
   C // sorted array with  elements of A and B
MergeSorted(A, n, B, m)
   i = 0; j = 0; k = 0
   while k < m+n
      if i < n and (j == m or A[i] < B[j])
         C[k] = A[i]; i = i+1
      else
         C[k] = B[j]; j = j+1
      endif
      k = k+1
   endwhile
endMergeSorted
```

24

And then 7. At this point we have reached the end of B.



i	i < n	j	j == m	A[i] < B[j]	k
0	true	0	false	true	0
1	true	0	false	false	1
1	true	1	false	false	2
1	true	2	false	false	3
1	true	3	true		4
2	true	3	true		5

From now on the condition i < n and (j == m or A[i] < B[j]) will always be true until i reaches n. This is because i is less than n and j is m. Since j is m, A[i] < B[j] isn't even evaluated anymore: true or whatever is true.

The remaining of the array A is copied into C.

Merge sorted trace

A:

0	*1*	*2*
0	8	9

n = 3

B:

0	*1*	*2*
2	4	7

m = 3

i	i < n	j	j == m	A[i] < B[j]	k
0	true	0	false	true	0
1	true	0	false	false	1
1	true	1	false	false	2
1	true	2	false	false	3
1	true	3	true		4
2	true	3	true		5
					6

C:

0	*1*	*2*	*3*	*4*	*5*
0	2	4	7	8	9

```
Input:
  A, n // sorted array of n integers
  B, m // sorted array of m integers
Output:
  C // sorted array with  elements of A and B
MergeSorted(A, n, B, m)
  i = 0; j = 0; k = 0
  while k < m+n
    if i < n and (j == m or A[i] < B[j])
      C[k] = A[i]; i = i+1
    else
      C[k] = B[j]; j = j+1
    endif
    k = k+1
  endwhile
endMergeSorted
```

26

When n+m elements are copied into C (i.e. k is m+n), we are done.

Merge sorted running time: n+m

- The while loop is executed n+m times
  - k starts at 0
  - k is incremented every time
- The while loop body takes constant time
  - 3 logical expressions + assignment + increment + increment

```
Input:
  A, n // sorted array of n integers
  B, m // sorted array of m integers
Output:
  C // sorted array with  elements of A and B
MergeSorted(A, n, B, m)
  i = 0; j = 0; k = 0
  while k < m+n
    if i < n and (j == m or A[i] < B[j])
      C[k] = A[i]; i = i+1
    else
      C[k] = B[j]; j = j+1
    endif
    k = k+1
  endwhile
endMergeSorted
```

27

The running time is m+n since k is advanced in every iteration (note that k = k+1 is not part of a conditional instruction, it is always executed). K starts at 0 and ends at n+m, thus the while loop is executed n+m times. The body of the while loop takes constant time, thus the overall running time of MergeSorted is n+m.

Merge sort: running time

```
Input:
   A // array of integers
   l // the index of the first element of A to be considered
   r // the index of the last element of A to be considered
Output:
   B // array with elements of A in ascending order
MergeSortR(A, l, r) // initial call MergeSortR(A, 0, n-1)
   if l == r // termination condition (subarray with 1 el.)
      B[0] = A[l]
      return B
   endif
   m = (l+r)/2 // midpoint
   Bl = MergeSortR(A, l, m) // sort left subarray
   Br = MergeSortR(A, m+1, r) // sort right subarray
   B = MergeSorted(Bl, Br) // merge subarrays
   return B // return merged subarrays
endMergeSortR
```

1 + 1 + … + 1	n "ones"
2 + 2 + … + 2	n/2 "twos"
4 + 4 + … + 4	n/4 "fours"
…	
n/2 + n/2	2 "n-over-two's"

$n\log_2 n$ (each row totals n, there are logn rows)

It has been shown that one cannot sort faster than nlogn.

28

To compute the running time of MergeSort we need to add up all the work done in MergeSorted.

MergeSorted is executed for the first time when we split the original array into single element sub-arrays.

MergeSorted is executed once for every single element. One execution of MergeSorted on a single element sub-array costs 1. There are n of them.

Then it is executed once for every sub-array of two elements. One execution of MergeSorted on a sub-array with two elements costs 2. There are n/2 of them.

And so on.

Finally MergeSorted is executed once on two sub-arrays of length n/2. One such execution costs n/2+n/2, and there is only one such execution.

There are $\log_2 n$ lines, each totals n, for a running time of $n\log_2 n$. This is the fastest running time possible for sorting. $n\log_2 n$ is a lot faster than quadratic, i.e. n^2.