More recursive algorithms

Computing factorial

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

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
```

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
```

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}

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}

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}

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}

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}

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}

Merge sort trace

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)	

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

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)	

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}

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}

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}

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
```

Merge sorted trace

A: n = 3

0	1	2
0	8	9

B: m = 3

0	1	2
2	4	7

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

C:

0	1	2	3	4	5
0					

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
```

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
```

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
```

Merge sorted trace

A: n = 3

0	1	2
0	8	9

B: m = 3

0	1	2
2	4	7

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

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

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

Merge sorted trace

A: n = 3

0	1	2
0	8	9

B: m = 3

0	1	2
2	4	7

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
```

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
```

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.