Data Structures and Algorithms

Khazhak Galstyan

Session 3:
Sorting Algorithms

What we will cover today

- Insertion Sort
- MergeSort
- Recursion Tree Method for MergeSort

Insertion Sort

The Sorting Problem

INPUT: a list of n elements

3	2	6	8	1	5	4	7

1	2	3	4	5	6	7	8

OUTPUT: a list with those n elements in sorted order!

Insertion Sort: The Algorithm

The Intuition: Maintain an iteratively growing sorted list. Put every next element in the sorted list in it’s “correct” position.

The Algorithm: Start from the beginning of the list, pick every next element and move it to the beginning of the list unless the element to its left is smaller than the one we were moving or we arrived to the leftmost position. Proceed to the next element.

Insertion Sort: The Algorithm

The Intuition: Maintain an iteratively growing sorted list. Put every next element in the sorted list in it’s “correct” position.

3	2	5	1	4

The Algorithm: Start from the beginning of the list, pick every next element and move it to the beginning of the list unless the element to its left is smaller than the one we were moving or we arrived to the leftmost position. Proceed to the next element.

Insertion Sort: The Algorithm

The Intuition: Maintain an iteratively growing sorted list. Put every next element in the sorted list in it's "correct" position.

3	2	5	1	4

At the first step our growing sorted list is just the first element of the array.

The Algorithm: Start from the beginning of the list, pick every next element and move it to the beginning of the list unless the element to its left is smaller than the one we were moving or we arrived to the leftmost position. Proceed to the next element.

Insertion Sort: The Algorithm

The Intuition: Maintain an iteratively growing sorted list. Put every next element in the sorted list in it's “correct” position.

3	2	5	1	4

We pick the next element (2).

The Algorithm: Start from the beginning of the list, pick every next element and move it to the beginning of the list unless the element to its left is smaller than the one we were moving or we arrived to the leftmost position. Proceed to the next element.

Insertion Sort: The Algorithm

The Intuition: Maintain an iteratively growing sorted list. Put every next element in the sorted list in it's "correct" position.

3	2	5	1	4

We pick the next element (2).
And "try" to move it to its left.

The Algorithm: Start from the beginning of the list, pick every next element and move it to the beginning of the list unless the element to its left is smaller than the one we were moving or we arrived to the leftmost position. Proceed to the next element.

Insertion Sort: The Algorithm

The Intuition: Maintain an iteratively growing sorted list. Put every next element in the sorted list in it's "correct" position.

2	3	5	1	4

We pick the next element (2).
And "try" to move it to its left.
As 2 < 3, we switch the places of the 2 and 3.

The Algorithm: Start from the beginning of the list, pick every next element and move it to the beginning of the list unless the element to its left is smaller than the one we were moving or we arrived to the leftmost position. Proceed to the next element.

Insertion Sort: The Algorithm

The Intuition: Maintain an iteratively growing sorted list. Put every next element in the sorted list in it’s “correct” position.

2	3	5	1	4

We arrived to the beginning of the list, so we stop moving.

The Algorithm: Start from the beginning of the list, pick every next element and move it to the beginning of the list unless the element to its left is smaller than the one we were moving or we arrived to the leftmost position. Proceed to the next element.

Insertion Sort: The Algorithm

The Intuition: Maintain an iteratively growing sorted list. Put every next element in the sorted list in it's "correct" position.

2	3	5	1	4

We arrived to the beginning of the list, so we stop moving.
Note that our growing list now has two elements.

The Algorithm: Start from the beginning of the list, pick every next element and move it to the beginning of the list unless the element to its left is smaller than the one we were moving or we arrived to the leftmost position. Proceed to the next element.

Insertion Sort: The Algorithm

The Intuition: Maintain an iteratively growing sorted list. Put every next element in the sorted list in it's “correct” position.

Then we again pick the next element (5)

The Algorithm: Start from the beginning of the list, pick every next element and move it to the beginning of the list unless the element to its left is smaller than the one we were moving or we arrived to the leftmost position. Proceed to the next element.

Insertion Sort: The Algorithm

The Intuition: Maintain an iteratively growing sorted list. Put every next element in the sorted list in it’s “correct” position.

2	3	5	1	4

Then we again pick the next element (5)
Try to move it to the left, but 3 < 5 so we stop right here.

The Algorithm: Start from the beginning of the list, pick every next element and move it to the beginning of the list unless the element to its left is smaller than the one we were moving or we arrived to the leftmost position. Proceed to the next element.

Insertion Sort: The Algorithm

The Intuition: Maintain an iteratively growing sorted list. Put every next element in the sorted list in it's “correct” position.

Then we apply the same algorithm to the rest of the list.

The Algorithm: Start from the beginning of the list, pick every next element and move it to the beginning of the list unless the element to its left is smaller than the one we were moving or we arrived to the leftmost position. Proceed to the next element.

Insertion Sort: The Algorithm

The Intuition: Maintain an iteratively growing sorted list. Put every next element in the sorted list in it's “correct” position.

Then we apply the same algorithm to the rest of the list.

The Algorithm: Start from the beginning of the list, pick every next element and move it to the beginning of the list unless the element to its left is smaller than the one we were moving or we arrived to the leftmost position. Proceed to the next element.

Insertion Sort: The Algorithm

The Intuition: Maintain an iteratively growing sorted list. Put every next element in the sorted list in it's "correct" position.

2	1	3	5	4

Then we apply the same algorithm to the rest of the list.

The Algorithm: Start from the beginning of the list, pick every next element and move it to the beginning of the list unless the element to its left is smaller than the one we were moving or we arrived to the leftmost position. Proceed to the next element.

Insertion Sort: The Algorithm

The Intuition: Maintain an iteratively growing sorted list. Put every next element in the sorted list in it's "correct" position.

1	2	3	5	4

Then we apply the same algorithm to the rest of the list.

The Algorithm: Start from the beginning of the list, pick every next element and move it to the beginning of the list unless the element to its left is smaller than the one we were moving or we arrived to the leftmost position. Proceed to the next element.

Insertion Sort: The Algorithm

The Intuition: Maintain an iteratively growing sorted list. Put every next element in the sorted list in it's "correct" position.

1	2	3	5	4

Then we apply the same algorithm to the rest of the list.

The Algorithm: Start from the beginning of the list, pick every next element and move it to the beginning of the list unless the element to its left is smaller than the one we were moving or we arrived to the leftmost position. Proceed to the next element.

Insertion Sort: The Algorithm

The Intuition: Maintain an iteratively growing sorted list. Put every next element in the sorted list in it’s “correct” position.

1	2	3	5	4

Then we apply the same algorithm to the rest of the list.

The Algorithm: Start from the beginning of the list, pick every next element and move it to the beginning of the list unless the element to its left is smaller than the one we were moving or we arrived to the leftmost position. Proceed to the next element.

Insertion Sort: The Algorithm

The Intuition: Maintain an iteratively growing sorted list. Put every next element in the sorted list in it’s “correct” position.

1	2	3	4	5

Then we apply the same algorithm to the rest of the list.

The Algorithm: Start from the beginning of the list, pick every next element and move it to the beginning of the list unless the element to its left is smaller than the one we were moving or we arrived to the leftmost position. Proceed to the next element.

Insertion Sort: The Algorithm

The Intuition: Maintain an iteratively growing sorted list. Put every next element in the sorted list in it's “correct” position.

1	2	3	4	5

Then we apply the same algorithm to the rest of the list.

The Algorithm: Start from the beginning of the list, pick every next element and move it to the beginning of the list unless the element to its left is smaller than the one we were moving or we arrived to the leftmost position. Proceed to the next element.

Insertion Sort: The Algorithm

The Intuition: Maintain an iteratively growing sorted list. Put every next element in the sorted list in it's "correct" position.

1	2	3	4	5

And that's it, we have sorted this particular array!

The Algorithm: Start from the beginning of the list, pick every next element and move it to the beginning of the list unless the element to its left is smaller than the one we were moving or we arrived to the leftmost position. Proceed to the next element.

Insertion Sort: The Proof

- We verified that Insertion Sort algorithm works on a particular input.

Insertion Sort: The Proof

- We verified that Insertion Sort algorithm works on a particular input.
- The next step is proving that it works for any input.

Insertion Sort: The Proof

- We verified that Insertion Sort algorithm works on a particular input.
- The next step is proving that it works for any input.
- Inductive proof time!

Insertion Sort: The Proof

The idea of the proof is fairly simple.

Insertion Sort: The Proof

The idea of the proof is fairly simple.

- We assume that first **k** elements are sorted

Insertion Sort: The Proof

The idea of the proof is fairly simple.

- We assume that first **k** elements are sorted
- Prove that the “moving to the beginning” part of the algorithm doesn’t break the order and we get first **k+1** elements sorted

Insertion Sort: The Proof

The idea of the proof is fairly simple.

- We assume that first **k** elements are sorted
- Prove that the “moving to the beginning” part of the algorithm doesn’t break the order and we get first **k+1** elements sorted
- Induction!

Insertion Sort: The Formal Proof

BASE CASE

After iteration 0 of the outer loop (i.e. start of algorithm), the list A[:1] is sorted (only 1 element). Thus, IH holds for $i = 0$.

INDUCTIVE HYPOTHESIS (IH)

After iteration i of the outer for-loop, A[:i+1] is sorted.

INDUCTIVE STEP

Let k be an integer, where $0 < k < n$. Assume that the IH holds for $i = k-1$, so A[:k] is sorted after the $(k-1)^{th}$ iteration. We want to show that the IH holds for $i = k$, i.e. that A[:k+1] is sorted after the k^{th} iteration.

Let j* be the largest position in {0, …, k-1} such that A[j*] < A[k]. Then, the effect of the inner while-loop is to turn:

[A[0], A[1], …, A[j*], …, A[k-1], **A[k]**]   into   [A[0], A[1], …, A[j*], **A[k]**, A[j*+1] …, A[k-1]]

We claim that the second list on the right is sorted. This is because A[k] > A[j*], and by the inductive hypothesis, we have A[j*] ≥ A[j] for all j ≤ j*, so A[k] is larger than everything positioned before it. Similarly, we also know that A[k] ≤ A[j*+1] ≤ A[j] for all j ≥ j*+1, so A[k] is also smaller than everything that comes after it. Thus, A[k] is in the right place, and all the other elements in A[:k+1] were already in the right place.

Thus, after the k^{th} iteration completes, A[:k+1] is sorted, and this establishes the IH for k.

CONCLUSION

By induction, we conclude that the IH holds for all $0 \leq i \leq n-1$. In particular, after the algorithm ends, A[:n] is sorted.

Insertion Sort: Pseudocode

```
InsertionSort(A):
  for i in range(len(A)):
    j = i - 1
    while j >= 0 and A[j+1] < A[j]:
      switch(A[j], A[j+1])
      j -= 1
  return A
```

Insertion Sort: Time Complexity

```
InsertionSort(A):
  for i in range(len(A)):
    j = i - 1
    while j >= 0 and A[j+1] < A[j]:
      switch(A[j], A[j+1])
      j -= 1
  return A
```

Insertion Sort: Time Complexity

```
InsertionSort(A):
  for i in range(len(A)):
    j = i - 1
    while j >= 0 and A[j+1] < A[j]:
      switch(A[j], A[j+1])
      j -= 1
  return A
```

n iterations

Insertion Sort: Time Complexity

n iterations

```
InsertionSort(A):
  for i in range(len(A)):
    j = i - 1
    while j >= 0 and A[j+1] < A[j]:
      switch(A[j], A[j+1])
      j -= 1
  return A
```

Insertion Sort: Time Complexity

```
InsertionSort(A):
  for i in range(len(A)):
    j = i - 1
    while j >= 0 and A[j+1] < A[j]:
      switch(A[j], A[j+1])
      j -= 1
  return A
```

n iterations

Less than **n** iterations

Insertion Sort: Time Complexity

```
InsertionSort(A):
  for i in range(len(A)):
    j = i - 1
    while j >= 0 and A[j+1] < A[j]:
      switch(A[j], A[j+1])
      j -= 1
  return A
```

n iterations

Less than **n** iterations

Insertion Sort: Time Complexity

```
InsertionSort(A):
  for i in range(len(A)):
    j = i - 1
    while j >= 0 and A[j+1] < A[j]:
      switch(A[j], A[j+1])
      j -= 1
  return A
```

n iterations

Less than **n** iterations

O(1) work

Insertion Sort: Time Complexity

```
InsertionSort(A):
  for i in range(len(A)):
    j = i - 1
    while j >= 0 and A[j+1] < A[j]:
      switch(A[j], A[j+1])
      j -= 1
  return A
```

n iterations

Less than **n** iterations

O(1) work

n x (O(1) + **n** x O(1))

Insertion Sort: Time Complexity

```
InsertionSort(A):
  for i in range(len(A)):
    j = i - 1
    while j >= 0 and A[j+1] < A[j]:
      switch(A[j], A[j+1])
      j -= 1
  return A
```

n iterations

Less than **n** iterations

O(1) work

$\mathbf{n} \times (O(1) + \mathbf{n} \times O(1)) = \mathbf{n} \times (O(\mathbf{n}))$

Insertion Sort: Time Complexity

```
InsertionSort(A):
    for i in range(len(A)):
        j = i - 1
        while j >= 0 and A[j+1] < A[j]:
            switch(A[j], A[j+1])
            j -= 1
    return A
```

n iterations

Less than **n** iterations

O(1) work

$\mathbf{n} \times (O(1) + \mathbf{n} \times O(1)) = \mathbf{n} \times (O(\mathbf{n})) = O(\mathbf{n}^2)$!

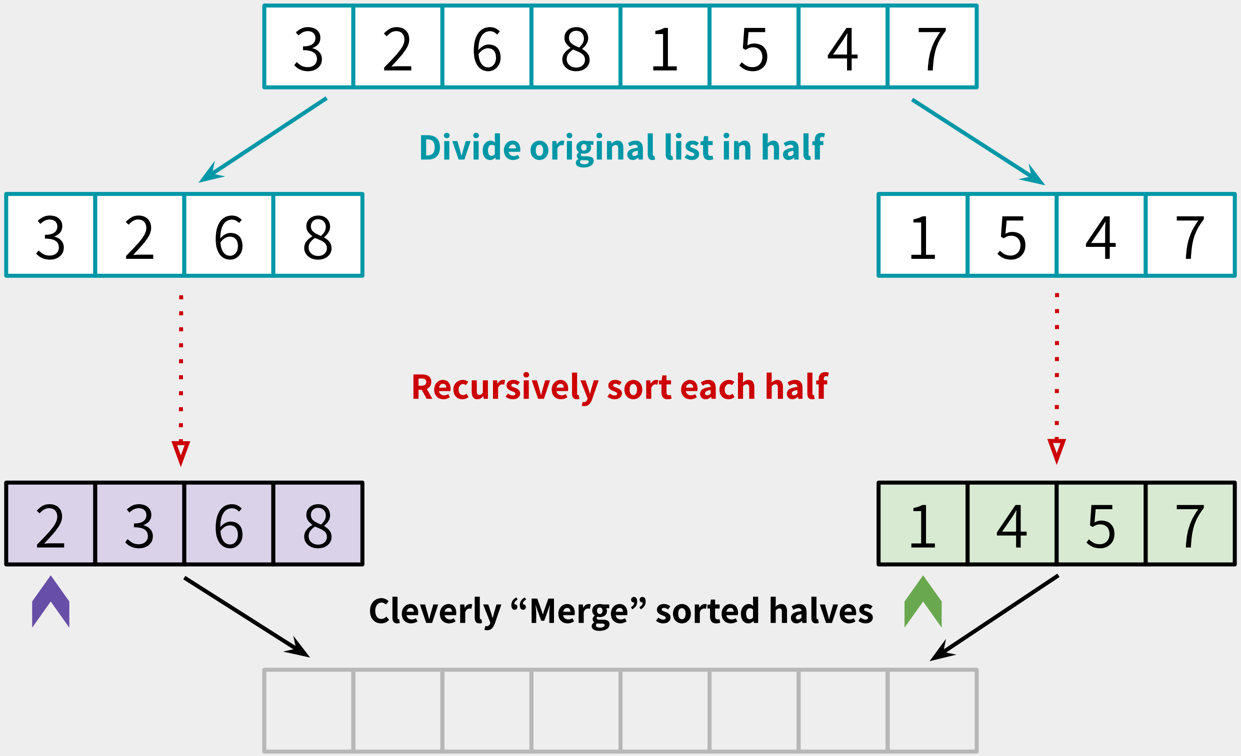
Insertion Sort: Time Complexity

- So Insertion Sort has time complexity of O($\mathbf{n^2}$).
- Can we do better?

MergeSort

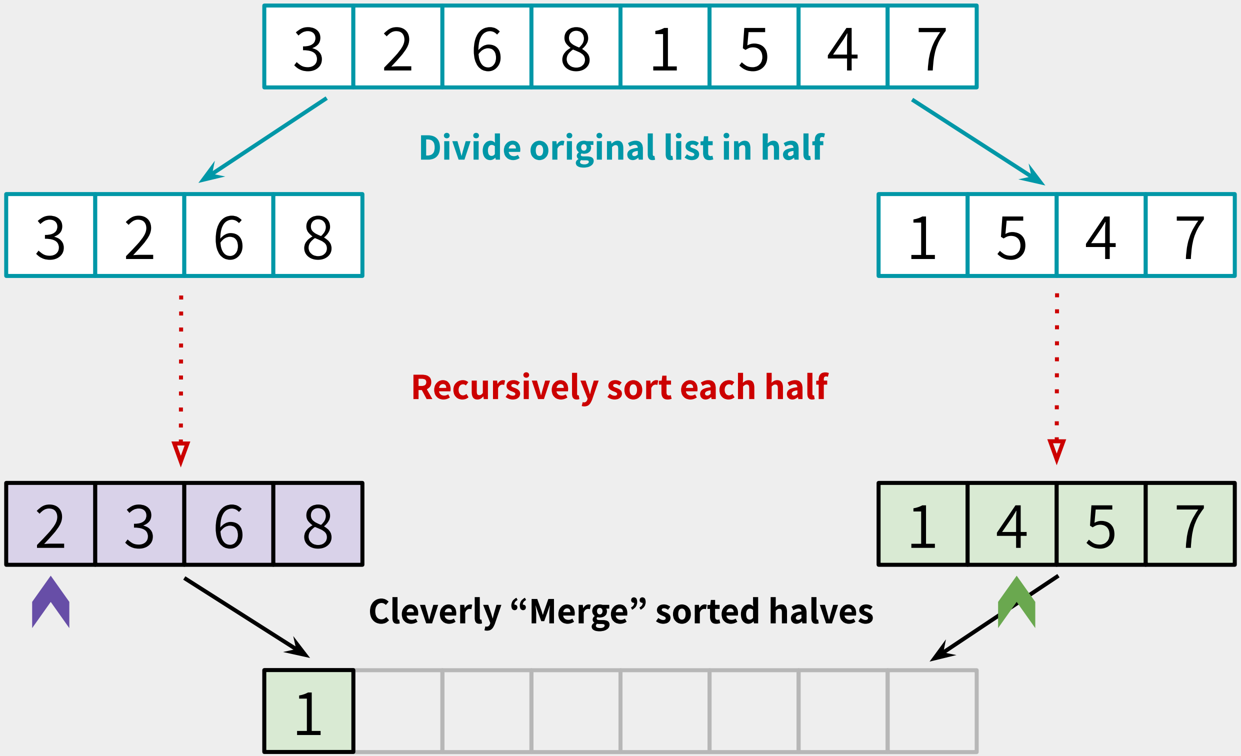
MergeSort

MergeSort



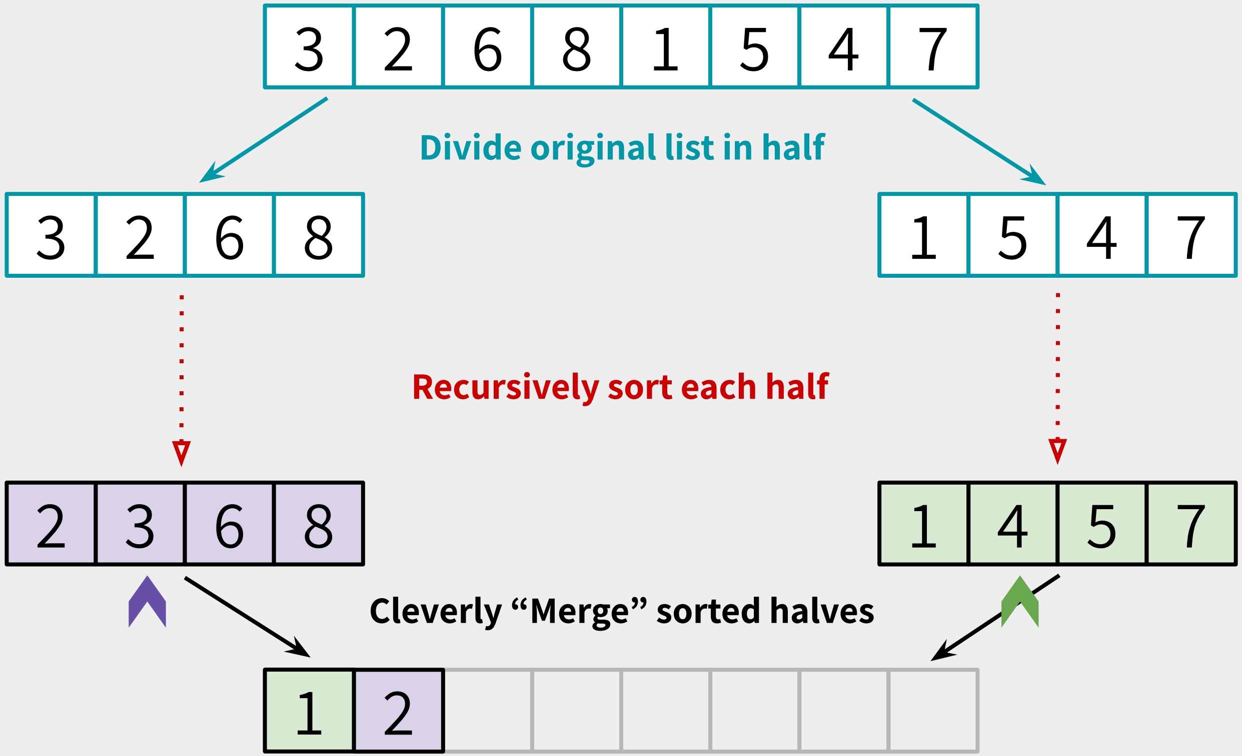
3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursively sort each half
2 3 6 8
1 4 5 7
Cleverly "Merge" sorted halves

MergeSort



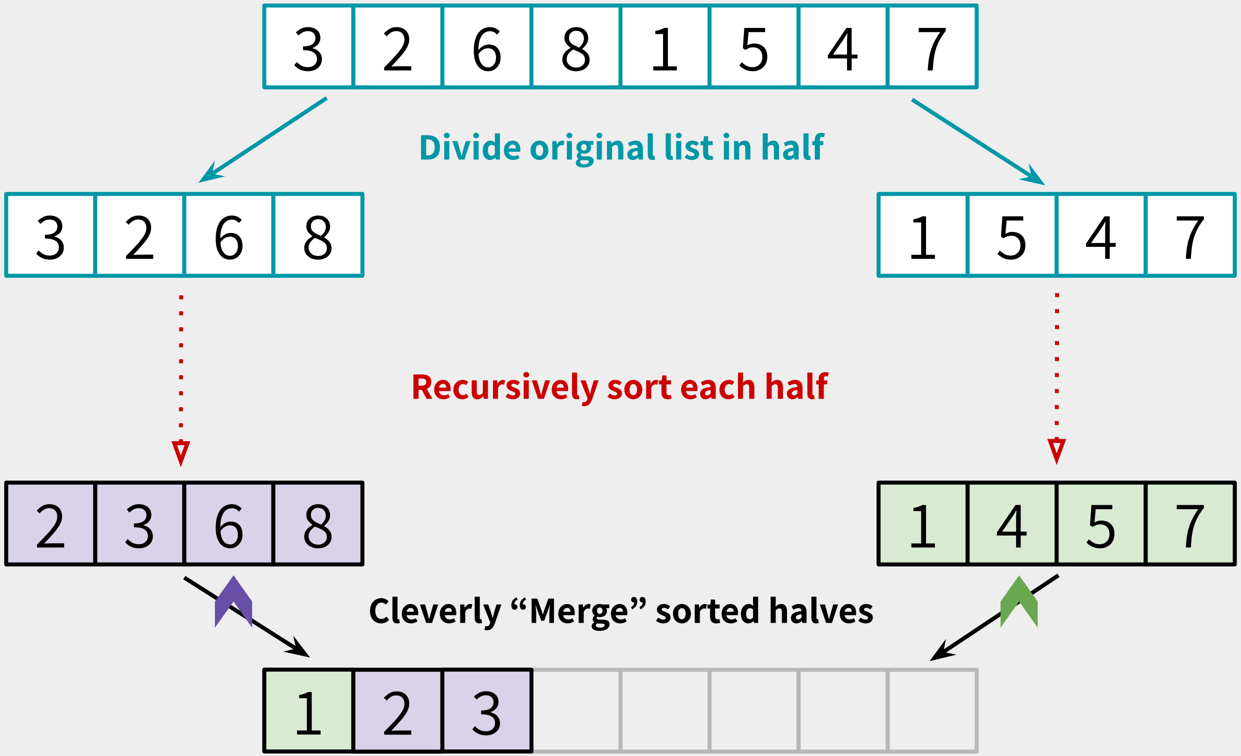
3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursively sort each half
2 3 6 8
1 4 5 7
Cleverly “Merge” sorted halves
1

MergeSort



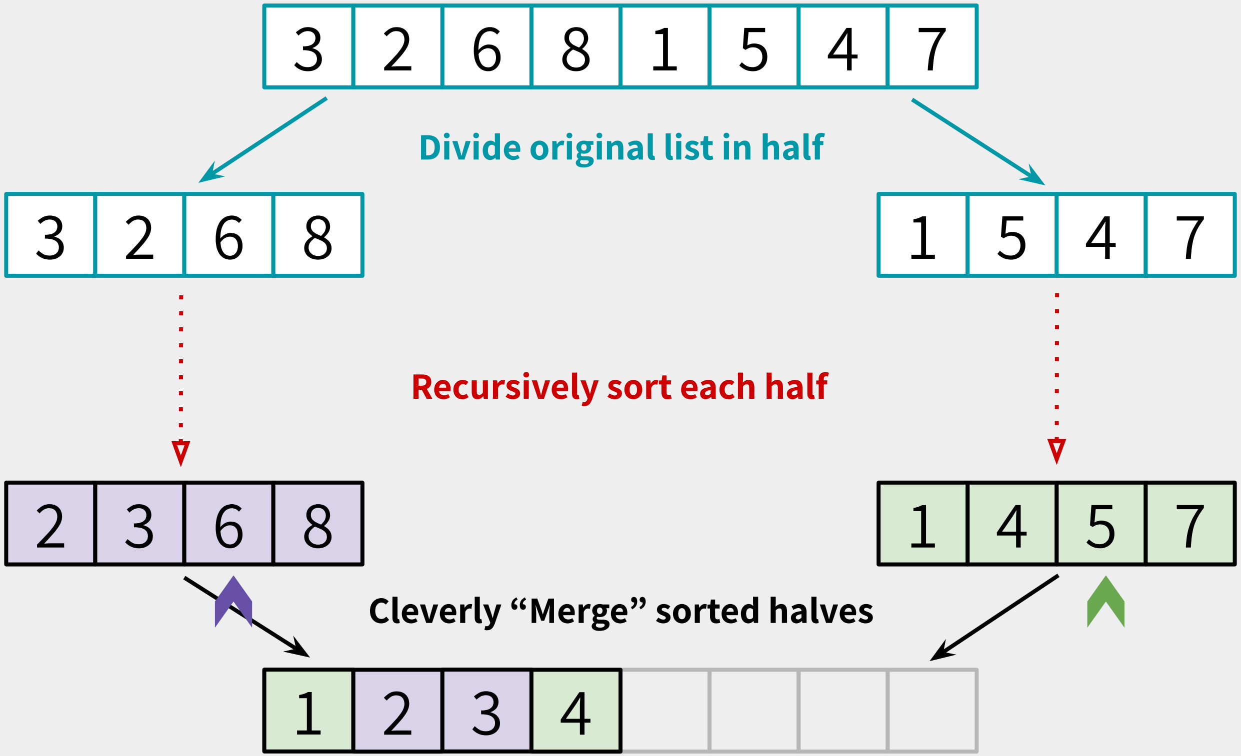
3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursively sort each half
2 3 6 8
1 4 5 7
Cleverly “Merge” sorted halves
1 2

MergeSort

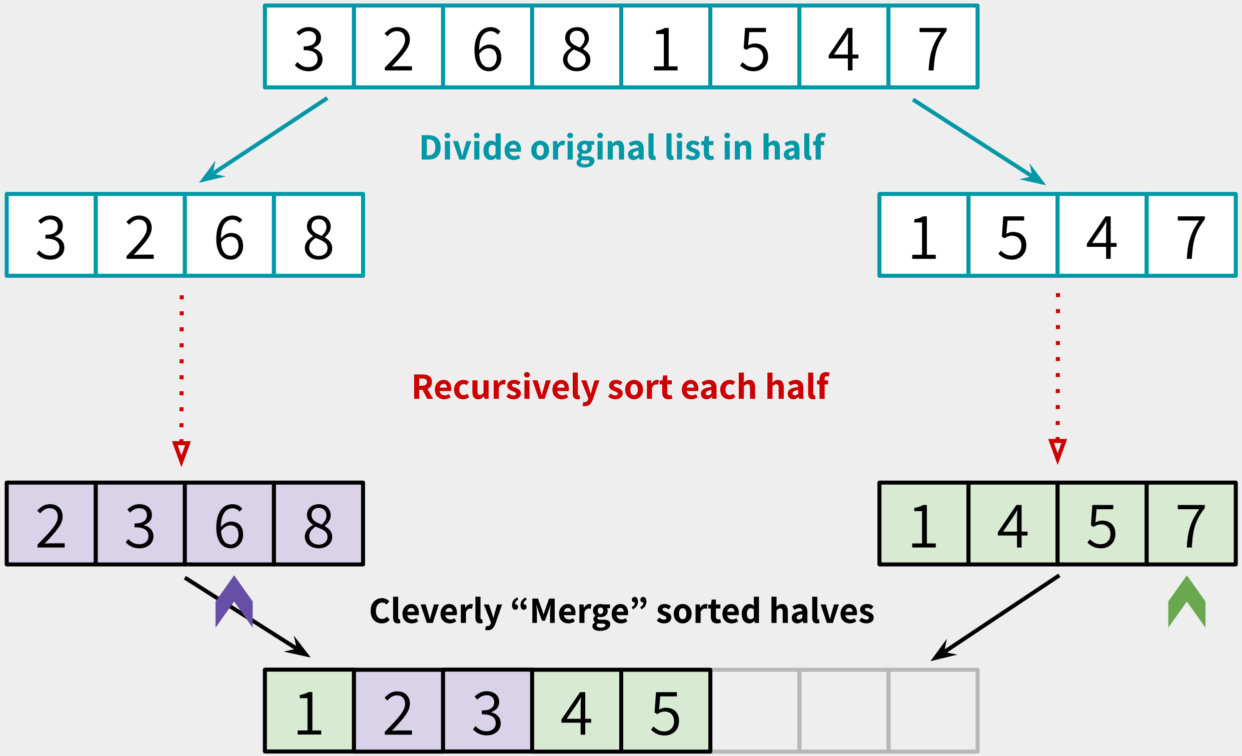


3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursively sort each half
2 3 6 8
1 4 5 7
Cleverly "Merge" sorted halves
1 2 3

MergeSort

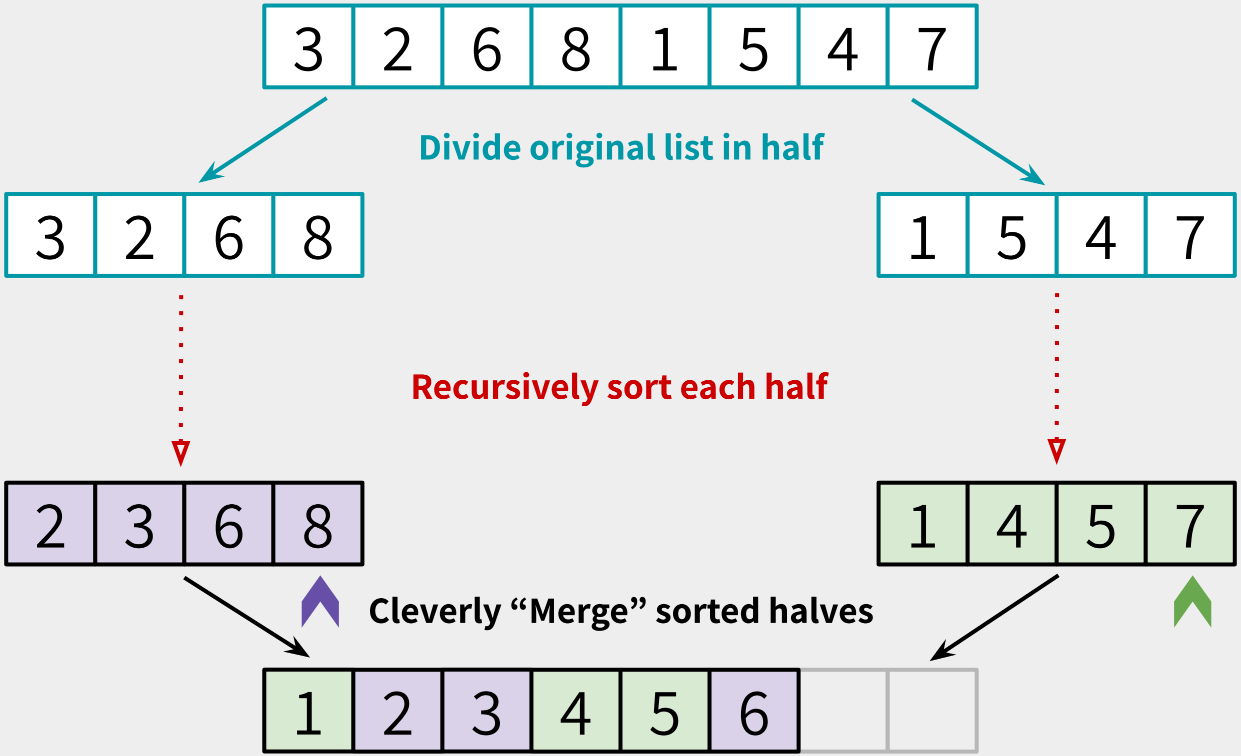

3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursively sort each half
2 3 6 8
1 4 5 7
Cleverly "Merge" sorted halves
1 2 3 4

MergeSort



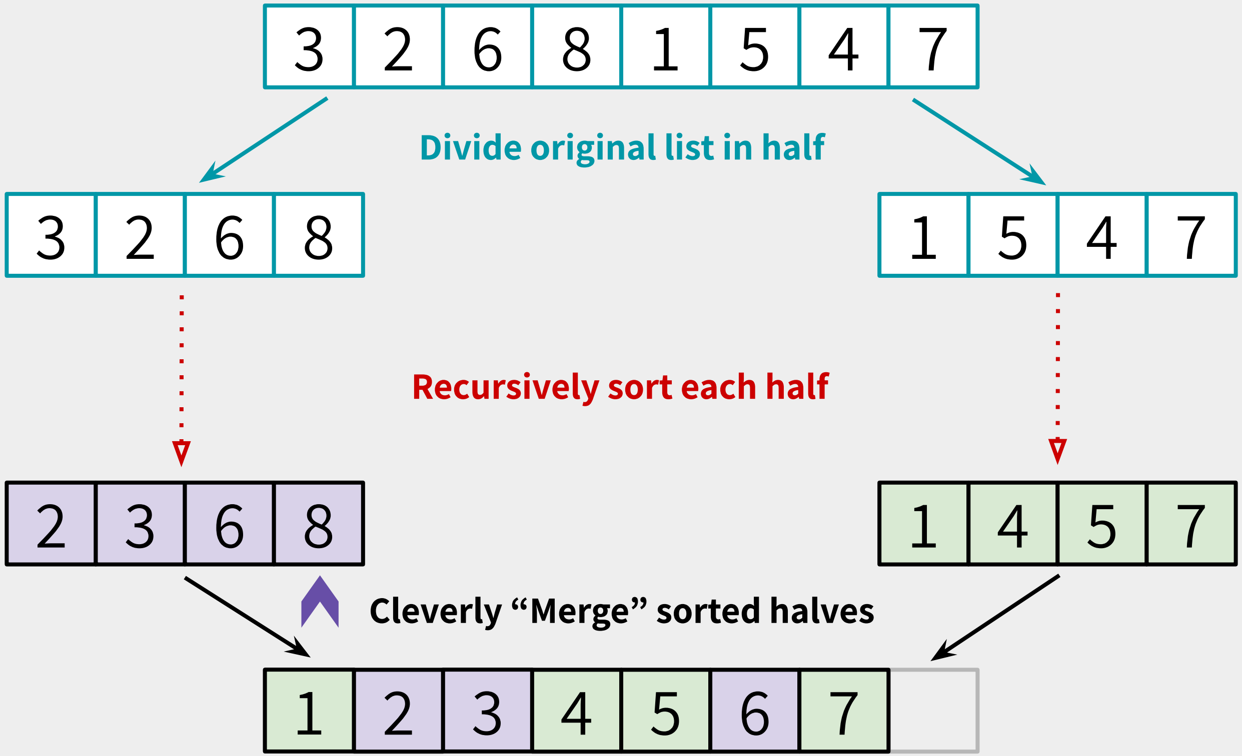
3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursively sort each half
2 3 6 8
1 4 5 7
Cleverly “Merge” sorted halves
1 2 3 4 5

MergeSort



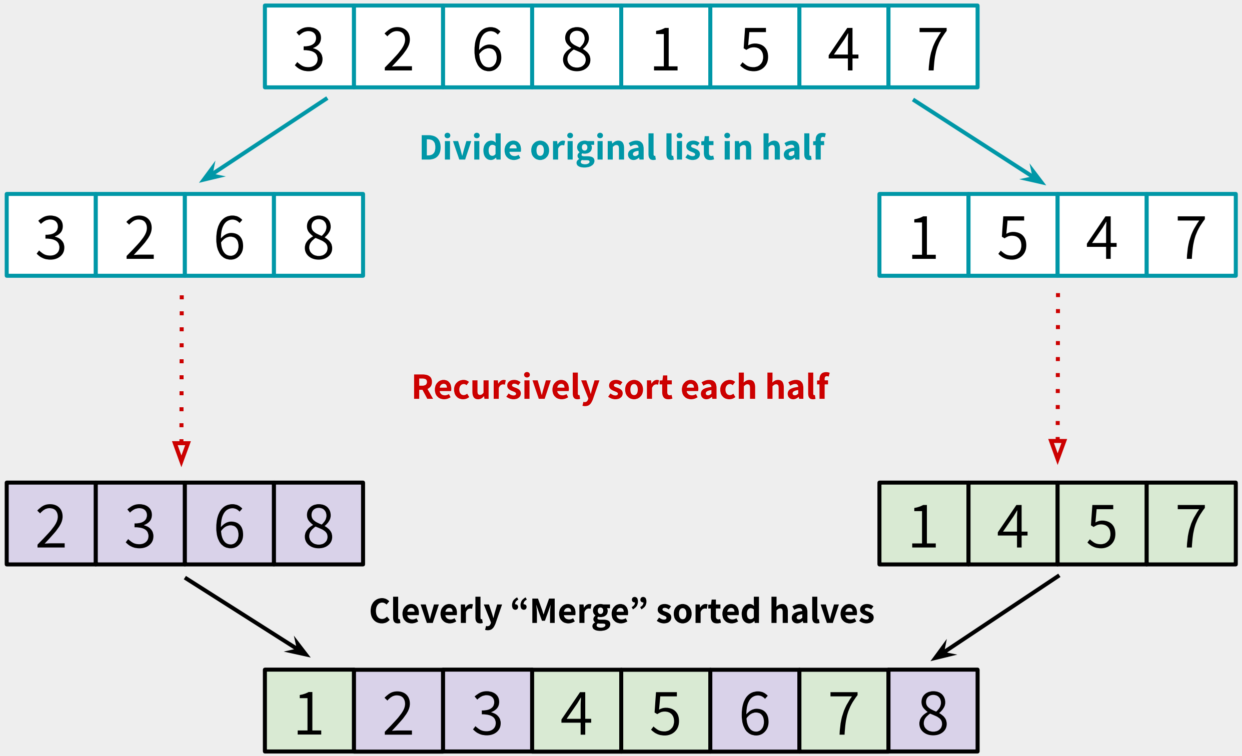
3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursively sort each half
2 3 6 8
1 4 5 7
Cleverly "Merge" sorted halves
1 2 3 4 5 6

MergeSort



3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursively sort each half
2 3 6 8
1 4 5 7
Cleverly “Merge” sorted halves
1 2 3 4 5 6 7

MergeSort

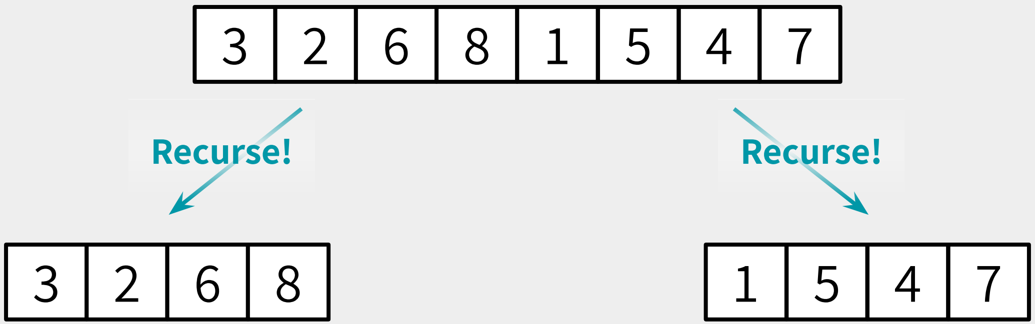


3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursively sort each half
2 3 6 8
1 4 5 7
Cleverly "Merge" sorted halves
1 2 3 4 5 6 7 8

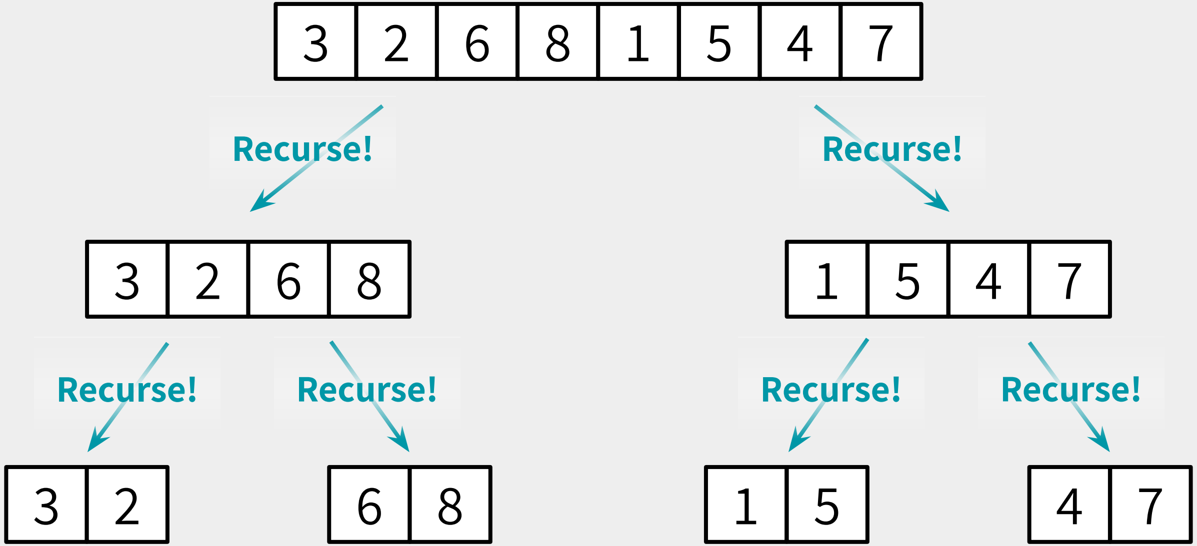
MergeSort: Recursion

3	2	6	8	1	5	4	7

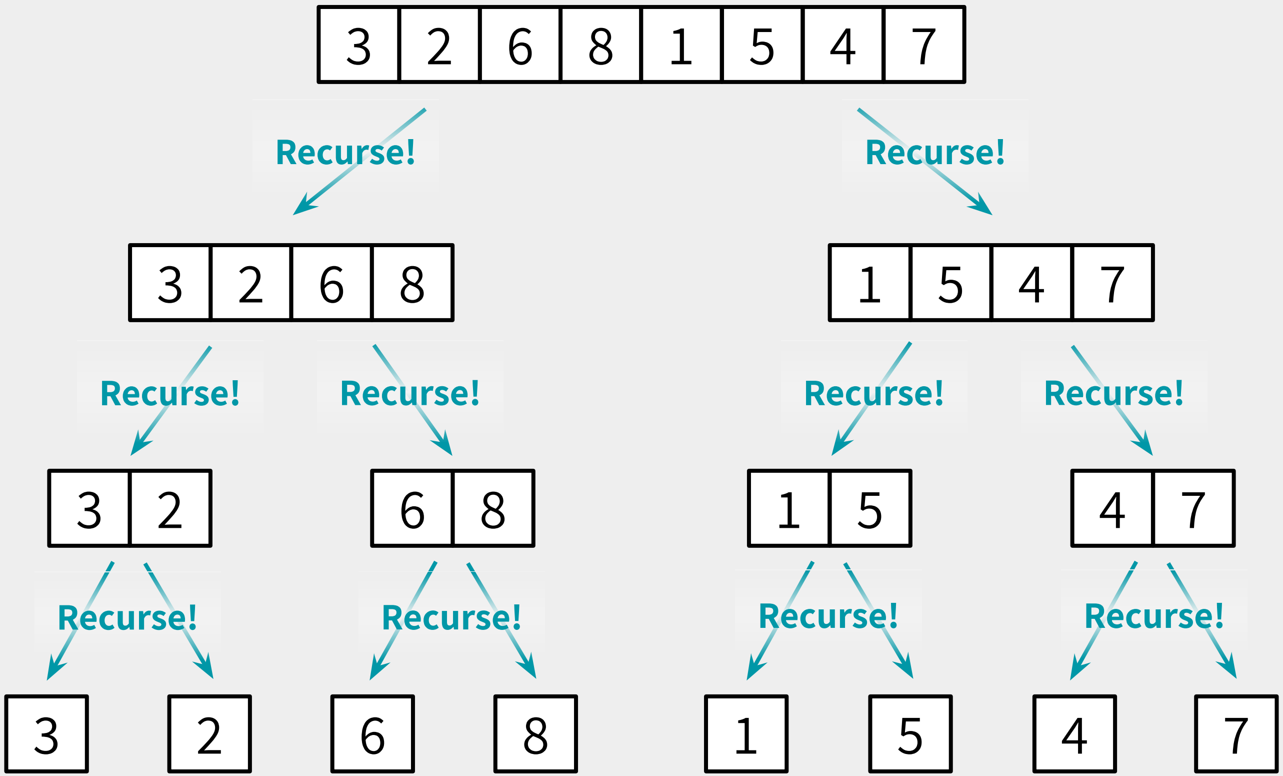
MergeSort: Recursion



MergeSort: Recursion



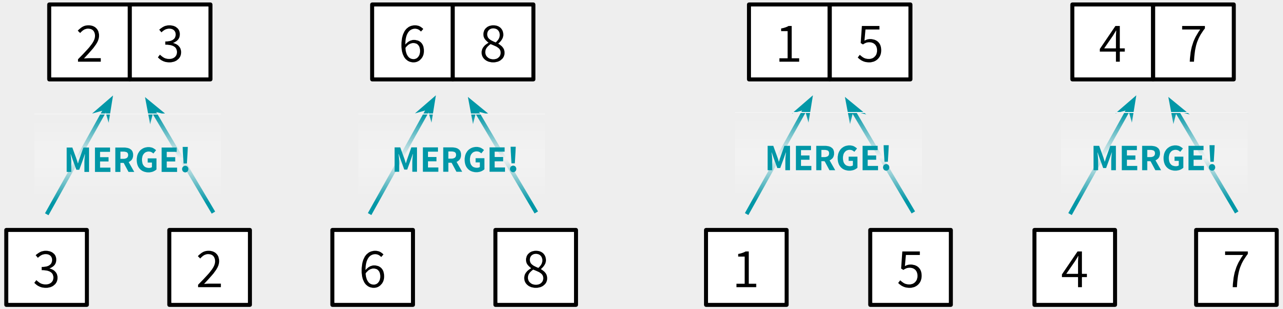
MergeSort: Recursion



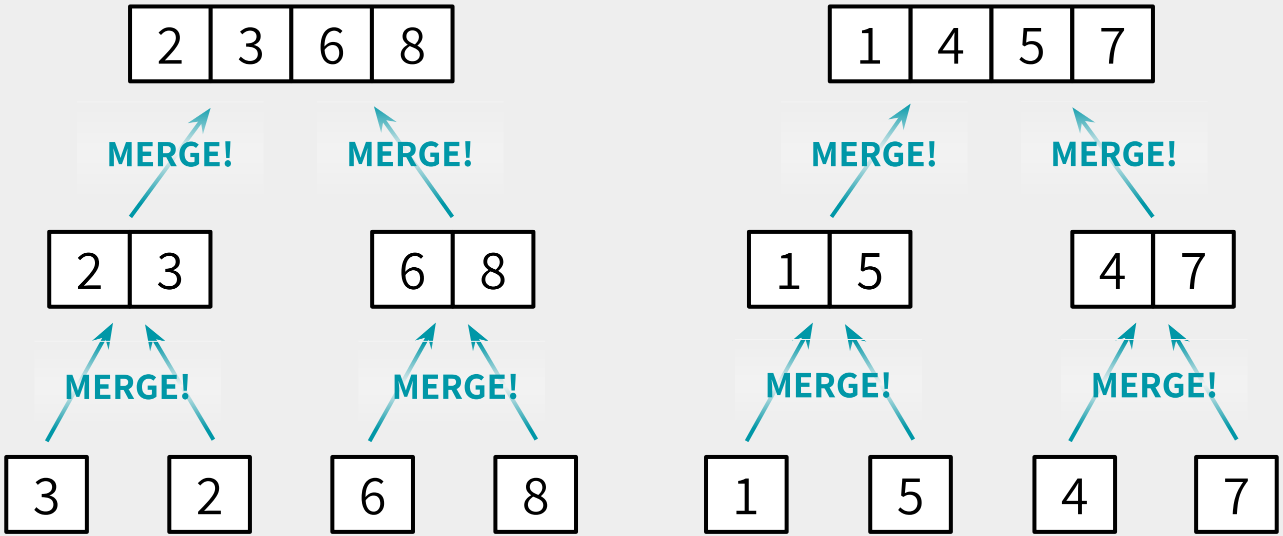
MergeSort: Recursion

3	2	6	8	1	5	4	7

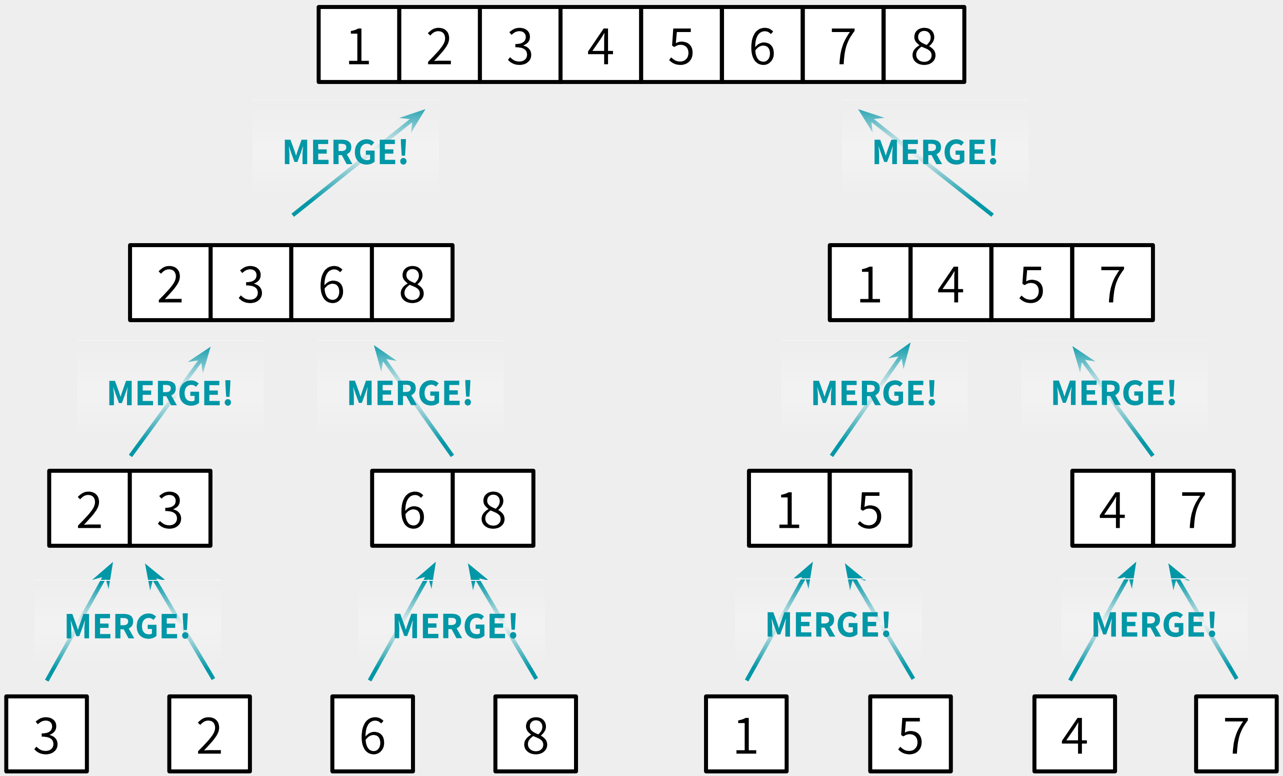
MergeSort: Recursion



MergeSort: Recursion



MergeSort: Recursion



MergeSort: Pseudocode

```
MergeSort(A):
  if len(A) <= 1:
    return A
  L = MergeSort(A[0:n/2])
  R = MergeSort(A[n/2:n])
  return Merge(L, R)
```

MergeSort: Pseudocode

```
MergeSort(A):
  if len(A) <= 1:
    return A
  L = MergeSort(A[0:n/2])
  R = MergeSort(A[n/2:n])
  return Merge(L, R)
```

```
Merge(L,R):
  result = length n array
  i = 0, j = 0
  for k in [0,...,n-1]:
    if L[i] < R[j]:
      result[k] = L[i]
      i += 1
    else:
      result[k] = R[j]
      j += 1
  return result
```

MergeSort: The Proof

- We verified that MergeSort algorithm works on a particular input.
- The next step is proving that it works for any input.
- Inductive proof time!

MergeSort: The Proof

The idea of the proof is again simple.

- We assume that the function works for inputs with less elements than **k**
- Prove that “merging” two sorted lists still results in a sorted list with **k+1** elements.
- Induction!

MergeSort: The Proof

INDUCTIVE HYPOTHESIS (IH)

In every recursive call on an array of length *at most* **i**, MERGESORT returns a sorted array.

BASE CASE

The IH holds for $i = 1$: A 1-element array is always sorted.

INDUCTIVE STEP *(strong/complete induction)*

Let k be an integer, where $1 < k \leq n$. Assume that the IH holds for $i < k$, so MERGESORT correctly returns a sorted array when called on arrays of length less than k. We want to show that the IH holds for $i = k$, i.e. that MERGESORT returns a sorted array when called on an array of length k.

[INSERT INDUCTION PROOF TO PROVE THE MERGE SUBROUTINE IS CORRECT WHEN GIVEN TWO SORTED ARRAYS]

Since the two "child" recursive calls are executed on arrays of length $k/2$ (which is strictly less than k), then our inductive hypothesis tells us that MERGESORT will correctly sort the left and right halves of our length-k array. Then, since the MERGE subroutine is correct when given two sorted arrays, we know that MERGESORT will ultimately return a fully sorted array of length k.

CONCLUSION

By induction, we conclude that the IH holds for all $1 \leq i \leq n$. In particular, it holds for $i = n$, so in the top recursive call, MERGESORT returns a sorted array.

MergeSort: Time Complexity (Merge)

```
Merge(L,R):
  result = length n array
  i = 0, j = 0
  for k in range(n-1):
    if L[i] < R[j]:
      result[k] = L[i]
      i += 1
    else:
      result[k] = R[j]
      j += 1
  return result
```

MergeSort: Time Complexity (Merge)

```
Merge(L,R):
  result = length n array
  i = 0, j = 0
  for k in range(n-1):
    if L[i] < R[j]:
      result[k] = L[i]
     i += 1
    else:
      result[k] = R[j]
      j += 1
   return result
```

MergeSort: Time Complexity (Merge)

At most **n** iterations

```
Merge(L,R):
  result = length n array
  i = 0, j = 0
  for k in range(n-1):
    if L[i] < R[j]:
      result[k] = L[i]
      i += 1
    else:
      result[k] = R[j]
      j += 1
  return result
```

MergeSort: Time Complexity (Merge)

At most **n** iterations

```
Merge(L,R):
  result = length n array
  i = 0, j = 0
  for k in range(n-1):
    if L[i] < R[j]:
      result[k] = L[i]
      i += 1
    else:
      result[k] = R[j]
      j += 1
  return result
```

MergeSort: Time Complexity (Merge)

At most **n** iterations

O(1) work

```
Merge(L,R):
  result = length n array
  i = 0, j = 0
  for k in range(n-1):
    if L[i] < R[j]:
      result[k] = L[i]
     i += 1
    else:
      result[k] = R[j]
      j += 1
  return result
```

MergeSort: Time Complexity (Merge)

At most **n** iterations

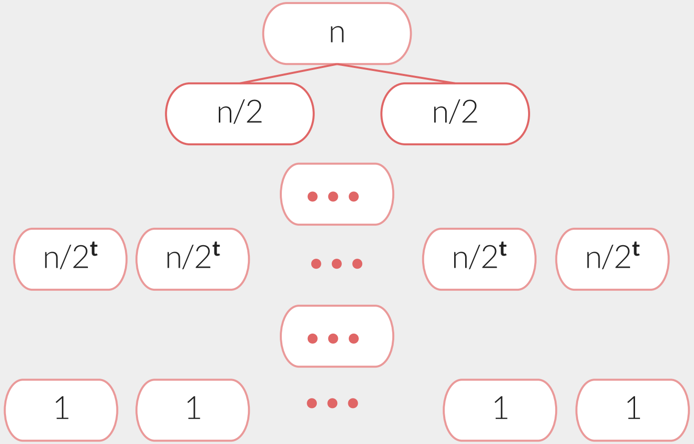
O(1) work

The complexity of `Merge` is O(**n**)!

```
Merge(L,R):
  result = length n array
  i = 0, j = 0
  for k in range(n-1):
    if L[i] < R[j]:
      result[k] = L[i]
     i += 1
    else:
      result[k] = R[j]
      j += 1
   return result
```

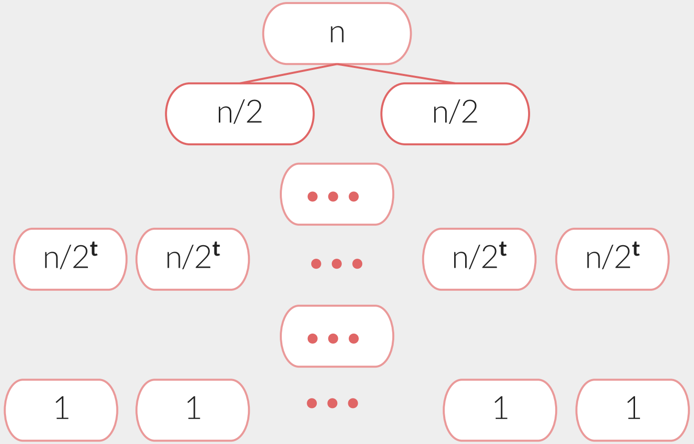
MergeSort Complexity Analysis: Recursion Tree Method

MergeSort: Recursion Tree



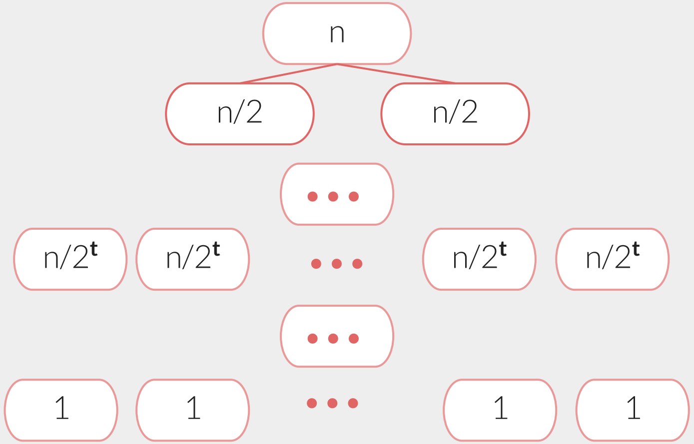
Level	# of Problems	Size of each Problem	Work done per Problem ≤	Total work on this level
0				
1				
...				
t				
...				
$\log_2 n$				

MergeSort: Recursion Tree



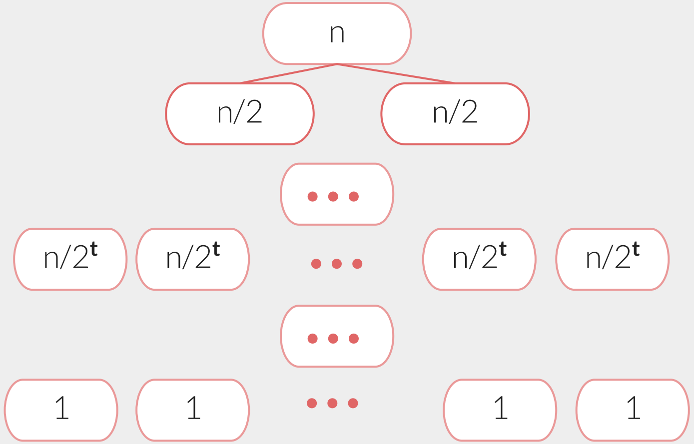
Level	# of Problems	Size of each Problem	Work done per Problem ≤	Total work on this level
0	1			
1				
…				
t				
…				
$\log_2 n$				

MergeSort: Recursion Tree



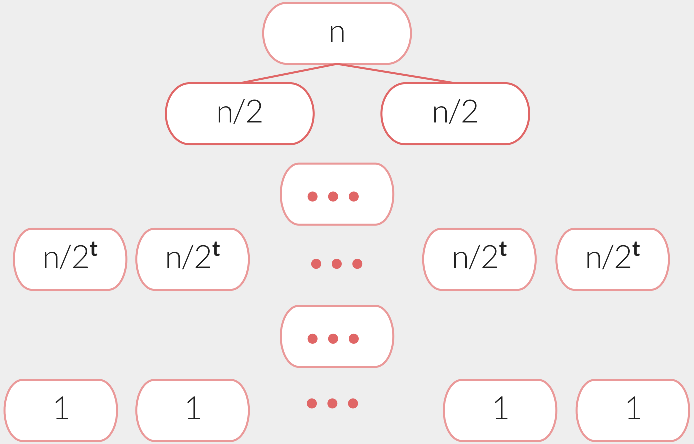
Level	# of Problems	Size of each Problem	Work done per Problem ≤	Total work on this level
0	1			
1	2^1			
		...		
t				
		...		
$\log_2 n$				

MergeSort: Recursion Tree



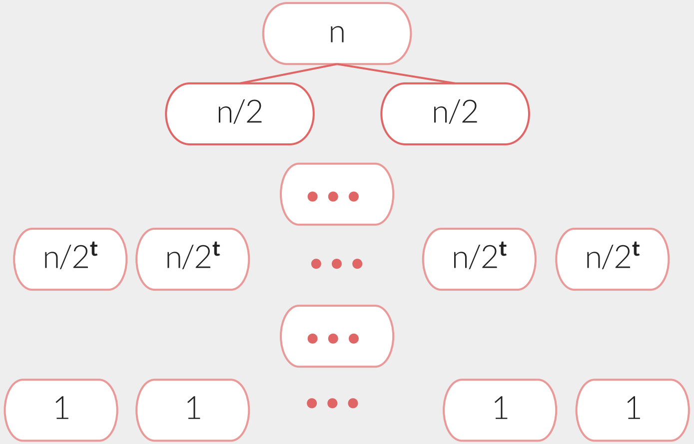
Level	# of Problems	Size of each Problem	Work done per Problem ≤	Total work on this level
0	1			
1	2^1			
		...		
t	2^t			
		...		
$\log_2 n$				

MergeSort: Recursion Tree



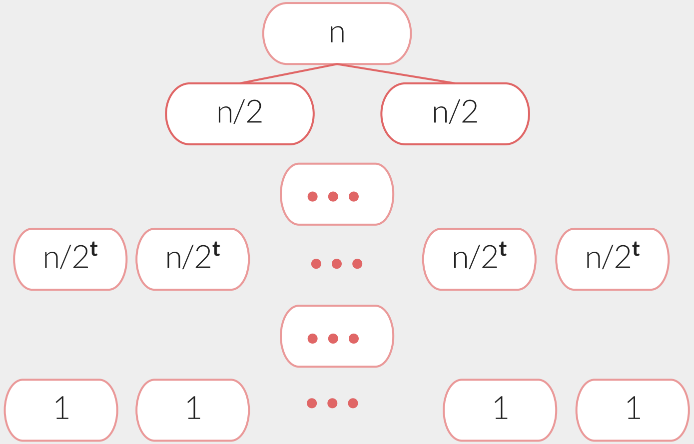
Level	# of Problems	Size of each Problem	Work done per Problem ≤	Total work on this level
0	1			
1	2^1			
		...		
t	2^t			
		...		
$\log_2 n$	$2^{\log2\ n} = n$			

MergeSort: Recursion Tree



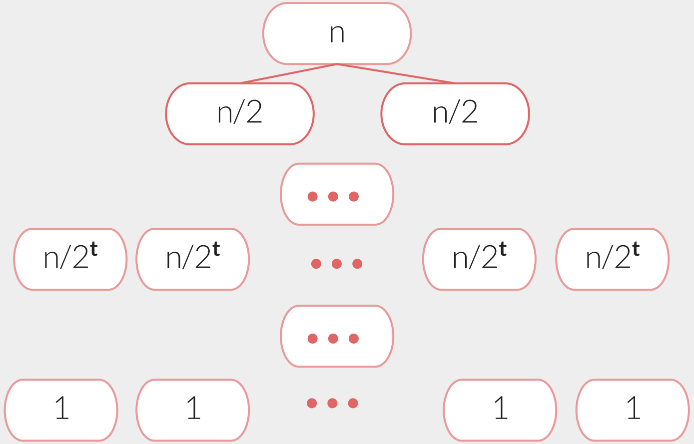
Level	# of Problems	Size of each Problem	Work done per Problem ≤	Total work on this level
0	1	n		
1	2^1			
		…		
t	2^t			
		…		
$\log_2 n$	$2^{\log 2\ n} = n$			

MergeSort: Recursion Tree



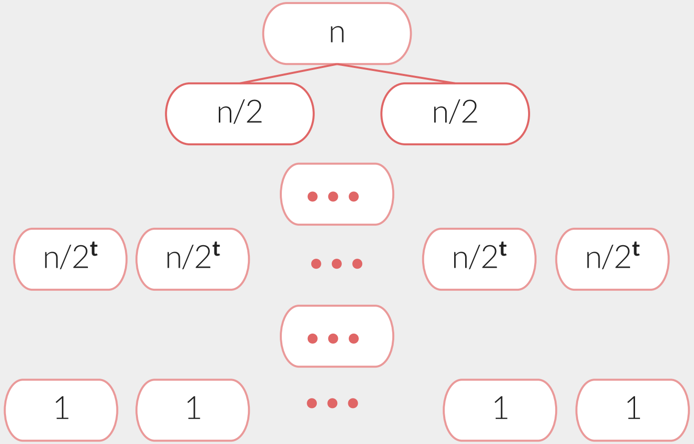
Level	# of Problems	Size of each Problem	Work done per Problem ≤	Total work on this level
0	1	n		
1	2^1	n/2		
		…		
t	2^t			
		…		
$\log_2 n$	$2^{\log2\ n} = n$			

MergeSort: Recursion Tree



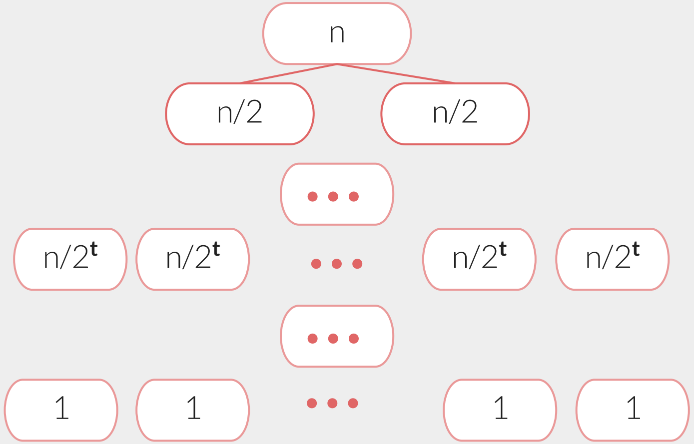
Level	# of Problems	Size of each Problem	Work done per Problem ≤	Total work on this level
0	1	n		
1	2^1	n/2		
…				
t	2^t	$n/2^t$		
…				
$\log_2 n$	$2^{\log 2\ n} = n$			

MergeSort: Recursion Tree



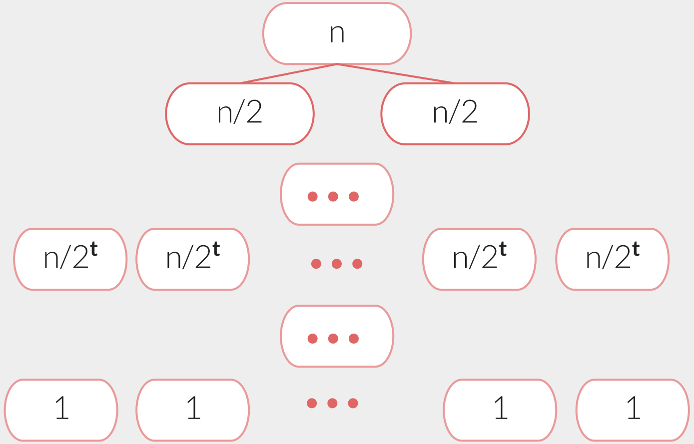
Level	# of Problems	Size of each Problem	Work done per Problem ≤	Total work on this level
0	1	n		
1	2^1	n/2		
		…		
t	2^t	$n/2^t$		
		…		
$\log_2 n$	$2^{\log 2\ n} = n$	1		

MergeSort: Recursion Tree



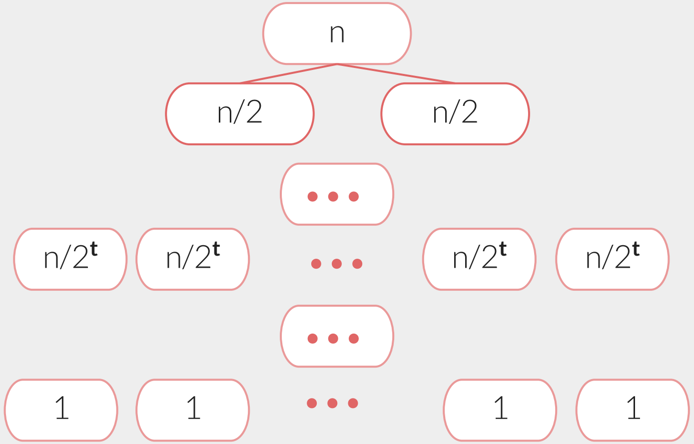
Level	# of Problems	Size of each Problem	Work done per Problem ≤	Total work on this level
0	1	n	$c \cdot n$	
1	2^1	n/2		
		…		
t	2^t	$n/2^t$		
		…		
$\log_2 n$	$2^{\log 2\ n} = n$	1		

MergeSort: Recursion Tree



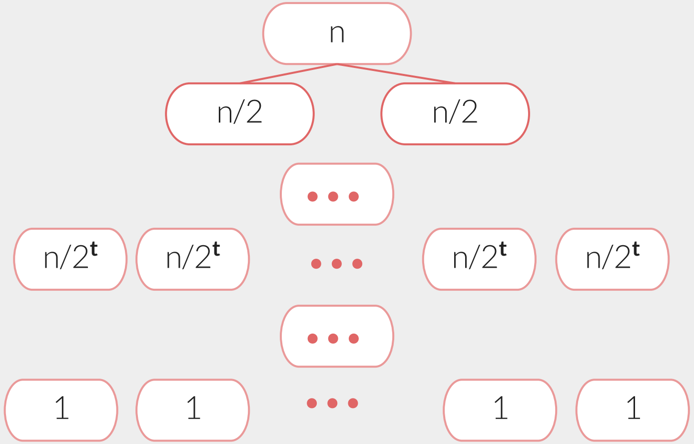
Level	# of Problems	Size of each Problem	Work done per Problem ≤	Total work on this level
0	1	n	$c \cdot n$	
1	2^1	n/2	$c \cdot (n/2)$	
...				
t	2^t	$n/2^t$		
...				
$\log_2 n$	$2^{\log 2\ n} = n$	1		

MergeSort: Recursion Tree



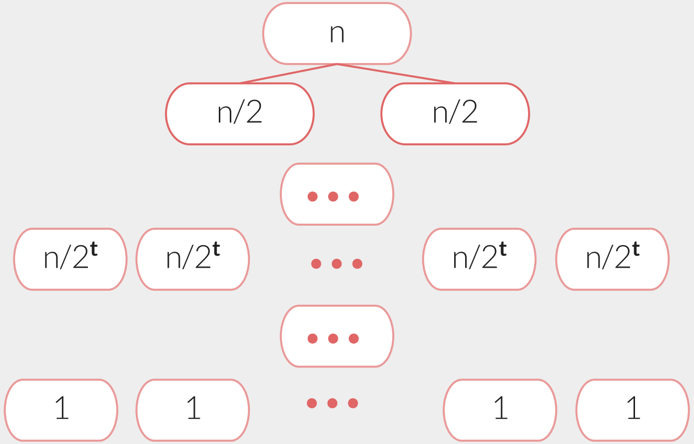
Level	# of Problems	Size of each Problem	Work done per Problem ≤	Total work on this level
0	1	n	$c \cdot n$	
1	2^1	n/2	$c \cdot (n/2)$	
		…		
t	2^t	$n/2^t$	$c \cdot (n/2^t)$	
		…		
$\log_2 n$	$2^{\log 2\ n} = n$	1		

MergeSort: Recursion Tree



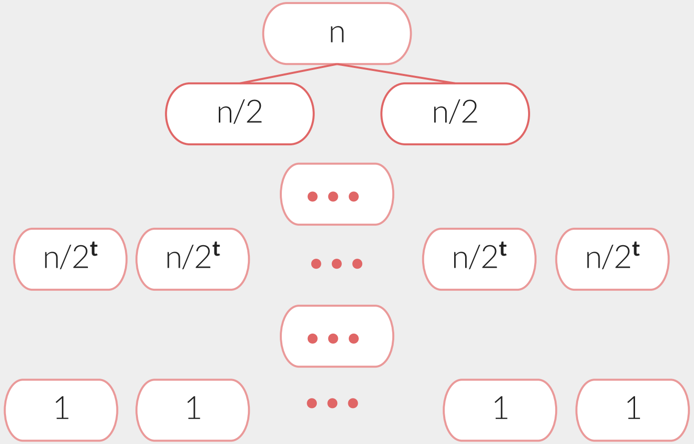
Level	# of Problems	Size of each Problem	Work done per Problem ≤	Total work on this level
0	1	n	$c \cdot n$	
1	2^1	n/2	$c \cdot (n/2)$	
…				
t	2^t	$n/2^t$	$c \cdot (n/2^t)$	
…				
$\log_2 n$	$2^{\log 2\ n} = n$	1	$c \cdot (1)$	

MergeSort: Recursion Tree



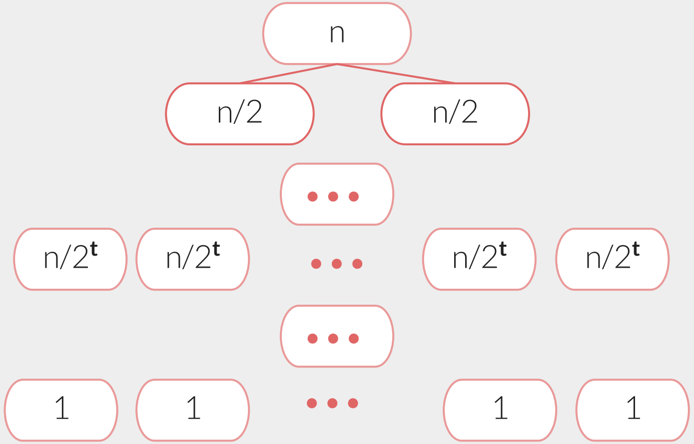
Level	# of Problems	Size of each Problem	Work done per Problem ≤	Total work on this level
0	1	n	$c \cdot n$	O(n)
1	2^1	n/2	$c \cdot (n/2)$	
		…		
t	2^t	$n/2^t$	$c \cdot (n/2^t)$	
		…		
$\log_2 n$	$2^{\log 2\ n} = n$	1	$c \cdot (1)$	

MergeSort: Recursion Tree



Level	# of Problems	Size of each Problem	Work done per Problem ≤	Total work on this level
0	1	n	$c \cdot n$	$O(n)$
1	2^1	$n/2$	$c \cdot (n/2)$	$2^1 \cdot c \cdot (n/2) = O(n)$
		...		
t	2^t	$n/2^t$	$c \cdot (n/2^t)$	
		...		
$\log_2 n$	$2^{\log 2\ n} = n$	1	$c \cdot (1)$	

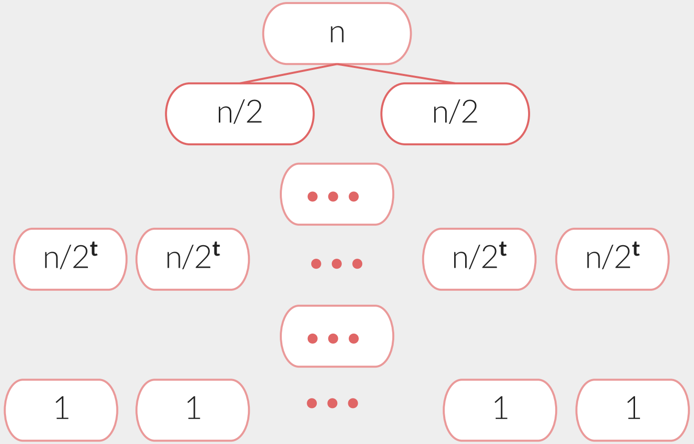
MergeSort: Recursion Tree



Level	# of Problems	Size of each Problem	Work done per Problem ≤	Total work on this level
0	1	n	$c \cdot n$	$O(n)$
1	2^1	$n/2$	$c \cdot (n/2)$	$2^1 \cdot c \cdot (n/2) = O(n)$
		…		
t	2^t	$n/2^t$	$c \cdot (n/2^t)$	$2^t \cdot c \cdot (n/2^t) = O(n)$
		…		
$\log_2 n$	$2^{\log 2\ n} = n$	1	$c \cdot (1)$	$n \cdot c \cdot (1) = O(n)$

MergeSort: Recursion Tree

We have ($\log_2 n + 1$) levels, each level has $O(n)$ work total $\Rightarrow$ **$O(n \log n)$** work overall!



Level	# of Problems	Size of each Problem	Work done per Problem ≤	Total work on this level
0	1	n	$c \cdot n$	$O(n)$
1	2^1	$n/2$	$c \cdot (n/2)$	$2^1 \cdot c \cdot (n/2) = O(n)$
		…		
t	2^t	$n/2^t$	$c \cdot (n/2^t)$	$2^t \cdot c \cdot (n/2^t) = O(n)$
		…		
$\log_2 n$	$2^{\log2\, n} = n$	1	$c \cdot (1)$	$n \cdot c \cdot (1) = O(n)$

Recap

- We discussed **Insertion Sort** and showed that it runs in **$O(n^2)$**.
- Introduced **MergeSort** algorithm.
- With **Recursion Tree Method** proved that **MergeSort** runs in **$O(nlogn)$** time.