ساختمان داده و الگوریتم ها

مبحث چهارم:
مرتب سازی درجی و ادغامی

سجاد شیرعلی شهرضا

پائیز 1402

شنبه، 15 مهر 1402

Based on Stanford CS161 Slides from Summer 2020 by Karey Shi

اطلاع رسانی

- بخش مرتبط کتاب برای این جلسه: 2.1، 2.3
- امتحانک اول
  - دوشنبه هفته آینده: 24 مهر 1402
  - در ساعت و محل کلاس
  - تا آخر مبحث تدریس شده در روز دوشنبه این هفته

مرتب سازی درجی

الگوریتم، اثبات درستی، تحلیل زمان اجرا

THE SORTING TASK

INPUT: a list of n elements (for today, we'll assume all elements are distinct)

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: PSEUDOCODE

Intuition: Maintain a growing sorted list. For each element, put it into its “correct” place in this growing list.

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

INSERTION SORT: EXAMPLE

Intuition: Maintain a growing sorted list. For each element, put it into its “correct” place in this growing list.

3	2	5	1	4

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

INSERTION SORT: EXAMPLE

Intuition: Maintain a growing sorted list. For each element, put it into its “correct” place in this growing list.

3	2	5	1	4

At the start, our growing sorted list only has one element (the first element):
3 is in its “correct” place within the growing list (shaded)

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

INSERTION SORT: EXAMPLE

Intuition: Maintain a growing sorted list. For each element, put it into its “correct” place in this growing list.

3	**2**	5	1	4

Now we look at A[1] = 2. We’ll move 2 into its “correct” place in the growing sorted list.

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

INSERTION SORT: EXAMPLE

Intuition: Maintain a growing sorted list. For each element, put it into its “correct” place in this growing list.

2	3	5	1	4

Now we look at A[1] = 2. We’ll move 2 into its “correct” place in the growing sorted list.
In other words, move 2 towards the start of the list until it hits something smaller (or if it can’t go any further).

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

INSERTION SORT: EXAMPLE

Intuition: Maintain a growing sorted list. For each element, put it into its “correct” place in this growing list.

2	3	**5**	1	4

Now we look at A[2] = 5. We’ll move 5 into its “correct” place in the growing sorted list.

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

INSERTION SORT: EXAMPLE

Intuition: Maintain a growing sorted list. For each element, put it into its “correct” place in this growing list.

2	3	**5**	1	4

Now we look at A[2] = 5. We’ll move 5 into its “correct” place in the growing sorted list.
It’s already where it should be in the growing sorted list, so we don’t need to move it anywhere. Moving on!

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

INSERTION SORT: EXAMPLE

Intuition: Maintain a growing sorted list. For each element, put it into its “correct” place in this growing list.

2	3	5	**1**	4

Now we look at A[3] = 1. We’ll move 1 into its “correct” place in the growing sorted list.

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

INSERTION SORT: EXAMPLE

Intuition: Maintain a growing sorted list. For each element, put it into its "correct" place in this growing list.

1	2	3	5	4

Now we look at A[3] = 1. We'll move 1 into its "correct" place in the growing sorted list.
We move it all the way to the front, since that's its "correct" position in this growing sorted list.

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

INSERTION SORT: EXAMPLE

Intuition: Maintain a growing sorted list. For each element, put it into its “correct” place in this growing list.

1	2	3	5	**4**

Finally, we look at A[4] = 4. We’ll move 4 into its “correct” place in the growing sorted list.

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

INSERTION SORT: EXAMPLE

Intuition: Maintain a growing sorted list. For each element, put it into its "correct" place in this growing list.

1	2	3	**4**	5

Finally, we look at A[4] = 4. We'll move 4 into its "correct" place in the growing sorted list.
It just needs to squeeze in right before 5.

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

INSERTION SORT: EXAMPLE

Intuition: Maintain a growing sorted list. For each element, put it into its "correct" place in this growing list.

1	2	3	4	5

And, that's it! We've finished performing Insertion Sort on this example array of five elements.

Now we ask… does it work?

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

سوال؟

اثبات درستی مرتب سازی درجی

آیا واقعا ورودی را مرتب میکند؟

INSERTION SORT: DOES IT WORK?

Since the algorithm isn't too complex, it might feel pretty obvious… but it won't be so obvious later, so let's take some time now to see how to prove the correctness of this algorithm rigorously..

3	2	5	1	4

We verified Insertion Sort worked for this particular input list.
However, we need to prove that the algorithm works for *all* possible input lists.

INSERTION SORT: DOES IT WORK?

Since the algorithm isn't too complex, it might feel pretty obvious… but it won't be so obvious later, so let's take some time now to see how to prove the correctness of this algorithm rigorously..

3	2	5	1	4

We verified Insertion Sort worked for this particular input list.
However, we need to prove that the algorithm works for *all* possible input lists.

HERE'S WHAT WE FOCUS ON:

Insertion Sort is an *iterative* algorithm - what does each iteration promise?

INSERTION SORT: DOES IT WORK?

HERE'S WHAT WE FOCUS ON:

Each iteration of the algorithm promises to add one more element to the sorted region.

*In other words: by the end of iteration i, we're guaranteed that the first **i+1** elements in the array are sorted.*

INSERTION SORT: DOES IT WORK?

HERE'S WHAT WE FOCUS ON:

Each iteration of the algorithm promises to add one more element to the sorted region.

In other words: by the end of iteration i, we're guaranteed that the first ***i+1*** *elements in the array are sorted.*

THIS IS A JOB FOR: ***PROOF BY INDUCTION!***

4 INGREDIENTS OF INDUCTION

INDUCTIVE HYPOTHESIS (IH)

This is a statement that's basically what you're trying to prove, except it's written in terms of some variable (e.g. **i**). We need to set up the inductive hypothesis clearly, and our goal in the next three steps is to prove that the IH holds for a whole *range* of values for **i**.

4 INGREDIENTS OF INDUCTION

INDUCTIVE HYPOTHESIS (IH)

This is a statement that's basically what you're trying to prove, except it's written in terms of some variable (e.g. **i**). We need to set up the inductive hypothesis clearly, and our goal in the next three steps is to prove that the IH holds for a whole *range* of values for **i**.

BASE CASE

First establish that the inductive hypothesis holds for some base case value(s) of **i**.

4 INGREDIENTS OF INDUCTION

INDUCTIVE HYPOTHESIS (IH)

This is a statement that's basically what you're trying to prove, except it's written in terms of some variable (e.g. **i**). We need to set up the inductive hypothesis clearly, and our goal in the next three steps is to prove that the IH holds for a whole *range* of values for **i**.

BASE CASE

First establish that the inductive hypothesis holds for some base case value(s) of **i**.

INDUCTIVE STEP *(weak induction version)*

Next, assume that the inductive hypothesis holds when **i** takes on some value **k**.
Now prove that the IH holds as well when **i** takes on the value **k+1**.

4 INGREDIENTS OF INDUCTION

INDUCTIVE HYPOTHESIS (IH)

This is a statement that's basically what you're trying to prove, except it's written in terms of some variable (e.g. **i**). We need to set up the inductive hypothesis clearly, and our goal in the next three steps is to prove that the IH holds for a whole *range* of values for **i**.

BASE CASE

First establish that the inductive hypothesis holds for some base case value(s) of **i**.

INDUCTIVE STEP *(weak induction version)*

Next, assume that the inductive hypothesis holds when **i** takes on some value **k**.
Now prove that the IH holds as well when **i** takes on the value **k+1**.

CONCLUSION

By induction, conclude that the IH holds across the range of **i** you're dealing with.

4 INGREDIENTS OF INDUCTION

INDUCTIVE HYPOTHESIS (IH)

This is a statement that's basically what you're trying to prove, except it's written in terms of some variable (e.g. **i**). We need to set up the inductive hypothesis clearly, and our goal in the next three steps is to prove that the IH holds for a whole *range* of values for **i**.

BASE CASE

First establish that the inductive hypothesis holds for some base case value(s) of **i**.

INDUCTIVE STEP *(strong/complete induction version)*

Next, assume that the IH holds when **i** takes on any value *between [base case value(s)] and some number* **k**. Now prove that the IH holds as well when **i** takes on the value **k+1**.

CONCLUSION

By induction, conclude that the IH holds across the range of **i** you're dealing with.

INSERTION SORT: INDUCTION PROOF

INDUCTIVE HYPOTHESIS (IH)

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

INSERTION SORT: INDUCTION PROOF

INDUCTIVE HYPOTHESIS (IH)

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

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.

INSERTION SORT: INDUCTION PROOF

INDUCTIVE HYPOTHESIS (IH)

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

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 STEP *(weak induction)*

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 $(k\text{-}1)^{th}$ 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.

INSERTION SORT: INDUCTION PROOF

INDUCTIVE HYPOTHESIS (IH)

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

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 STEP *(weak induction)*

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 ≤ i ≤ n-1. In particular, after the algorithm ends, A[:n] is sorted.

INSERTION SORT: INDUCTION PROOF

INDUCTIVE HYPOTHESIS (IH)

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

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 STEP *(weak induction)*

Let k be an in… …tion. We want to sho…

Let j* be th… …rn:

[A[0], A[1], …

We claim t… …s, we have A[j*] ≥ … A[k] ≤ A[j*+1] ≤ A[… …place, and all the …

TLDR, this inductive step is saying "if we assume the growing list on the left of A is properly sorted by iteration k-1, then when we're on iteration k, the algorithm correctly moves A[k] into the right place, and the growing list on the left of A is still going to be properly sorted."

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 ≤ i ≤ n-1. In particular, after the algorithm ends, A[:n] is sorted.

INSERTION SORT: DOES IT WORK?

We just used induction to prove that the Insertion Sort algorithm correctly produces a sorted array given *any input array of length n.*

(This is also what we mean by worst case analysis - even if a “bad guy” comes up with a worst-case input for our algorithm, we’ve proven that our algorithm will work).

سوال؟

زمان اجرای مرتب سازی درجی

چقدر سریع است؟

INSERTION SORT: IS IT FAST?

FROM LAST SESSION!

THE POINT OF ASYMPTOTIC NOTATION

suppress $\underbrace{\text{constant factors}}_{\textit{too system dependent}}$ and $\underbrace{\text{lower-order terms}}_{\textit{irrelevant for large inputs}}$

- **Some guiding principles:** we care about how the running time/number of operations *scales* with the size of the input (i.e. the runtime's *rate of growth*), and we want some measure of runtime that's independent of hardware, programming language, memory layout, etc.
    - We want to reason about high-level algorithmic approaches rather than lower-level details

INSERTION SORT: IS IT FAST?

Instead of counting every little operation, we can think about:

How many iterations take place

How much work happens within each iteration

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

INSERTION SORT: IS IT FAST?

Instead of counting every little operation, we can think about:
How many iterations take place
How much work happens within each iteration

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

At most n outer for-loop iterations

At most n inner while-loop iterations

INSERTION SORT: IS IT FAST?

Instead of counting every little operation, we can think about:

How many iterations take place

How much work happens within each iteration

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

At most n outer for-loop iterations

At most n inner while-loop iterations

We have n for-loop iterations. Each iteration does O(n) work.

(Each for-loop iteration performs an inner-while loop which iterates up to n times and does O(1) work in each iteration).

INSERTION SORT: IS IT FAST?

Instead of counting every little operation, we can think about:
How many iterations take place
How much work happens within each iteration

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

At most n outer for-loop iterations

At most n inner while-loop iterations

OVERALL RUNTIME OF INSERTION SORT: $O(n^2)$

INSERTION SORT: IS IT FAST?

Instead of counting every littl… …eration, we can think about:

How … place

How much … h iteration

```
Insertio…
    for …
        c…
        j…
        wh…                    …_value:
            A…
            j…
        A[j+1] = c…
```

At most n outer for-loop iterations

At most n inner while-loop iterations

THE QUESTION IS…

CAN WE DO BETTER?

OVERALL RUNTIME OF INSERTION SORT: $O(n^2)$

سوال؟

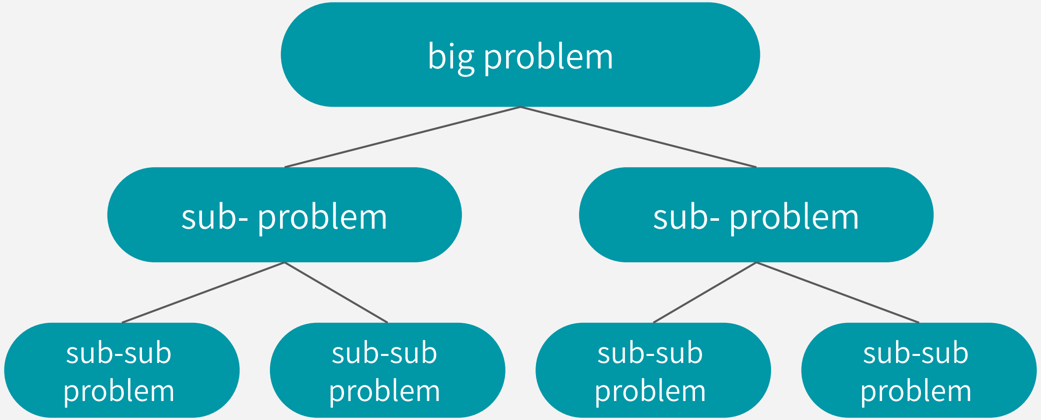
مرتب سازی ادغامی

الگوریتم، اثبات درستی، تحلیل زمان اجرا

MERGESORT

FROM PREVIOUS WEEKS!

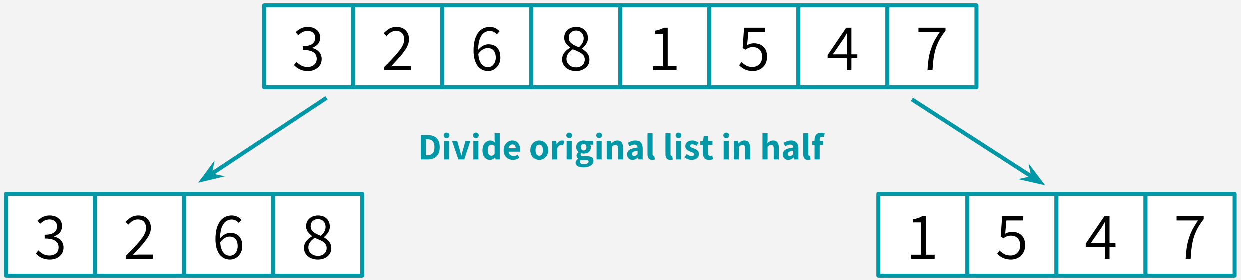
- **DIVIDE-AND-CONQUER: an algorithm design paradigm**
    1. break up a problem into smaller subproblems
    2. solve those subproblems *recursively*
    3. combine the results of those subproblems to get the overall answer



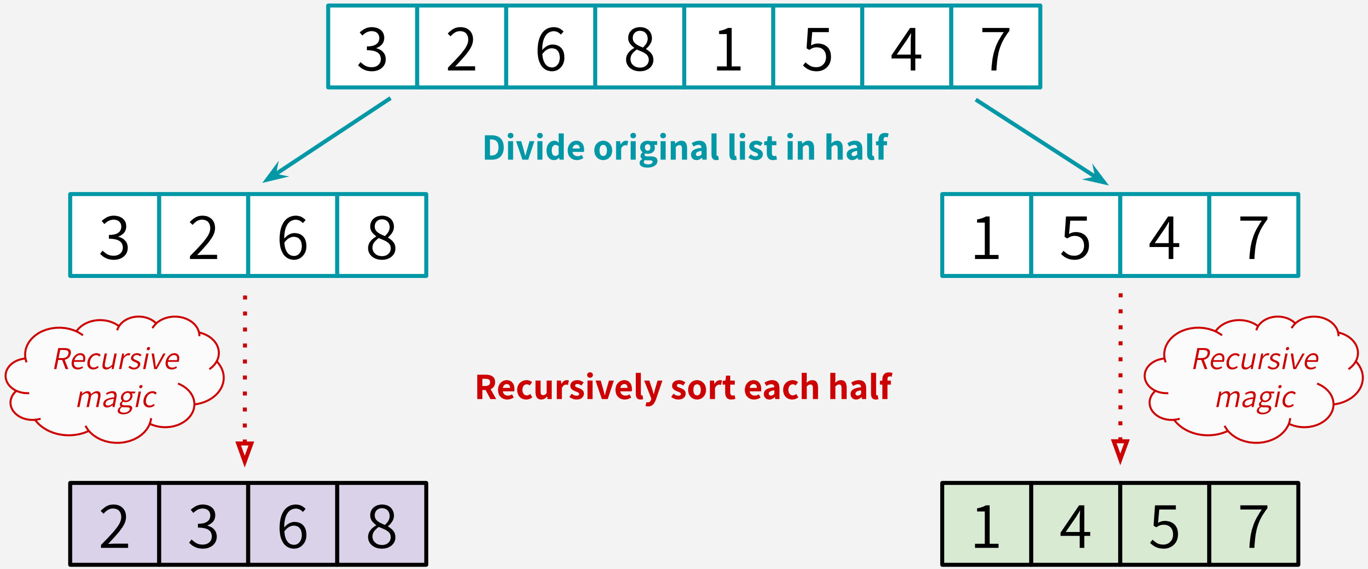
MERGESORT

3	2	6	8	1	5	4	7

MERGESORT

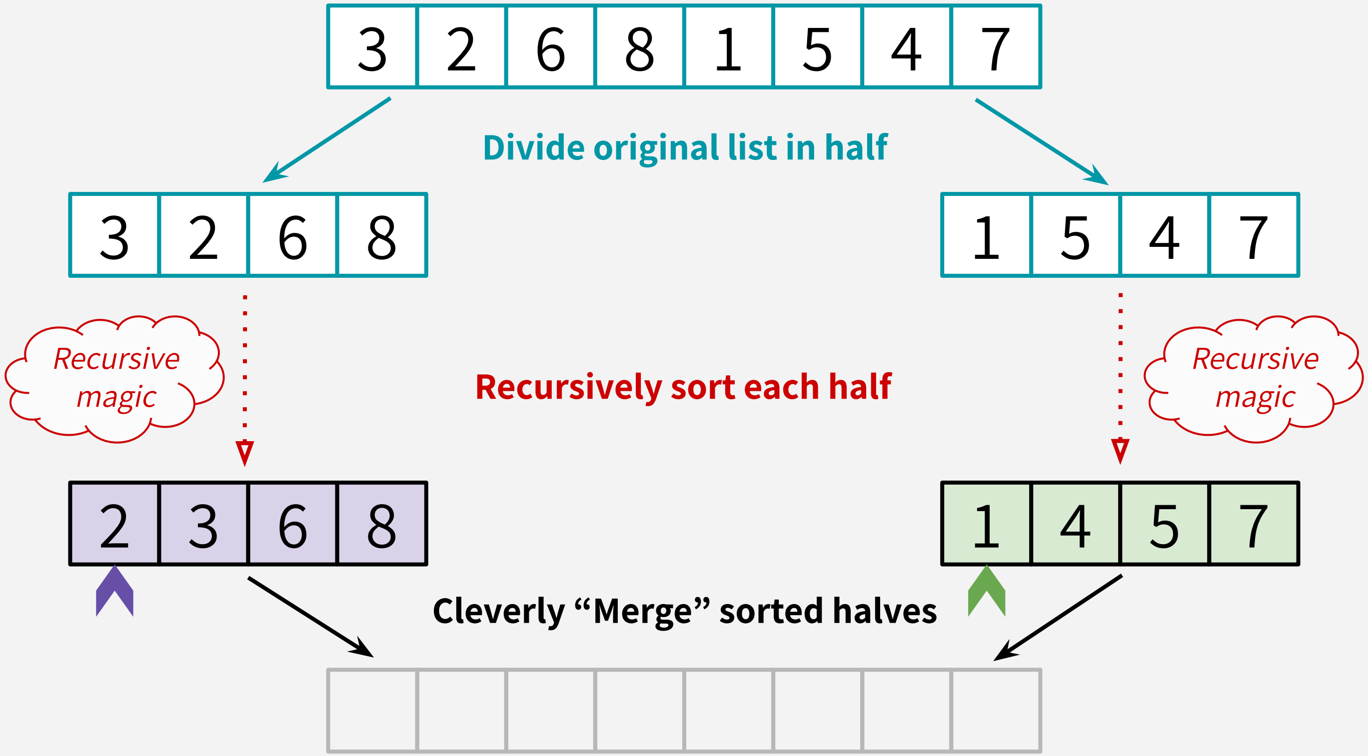


MERGESORT

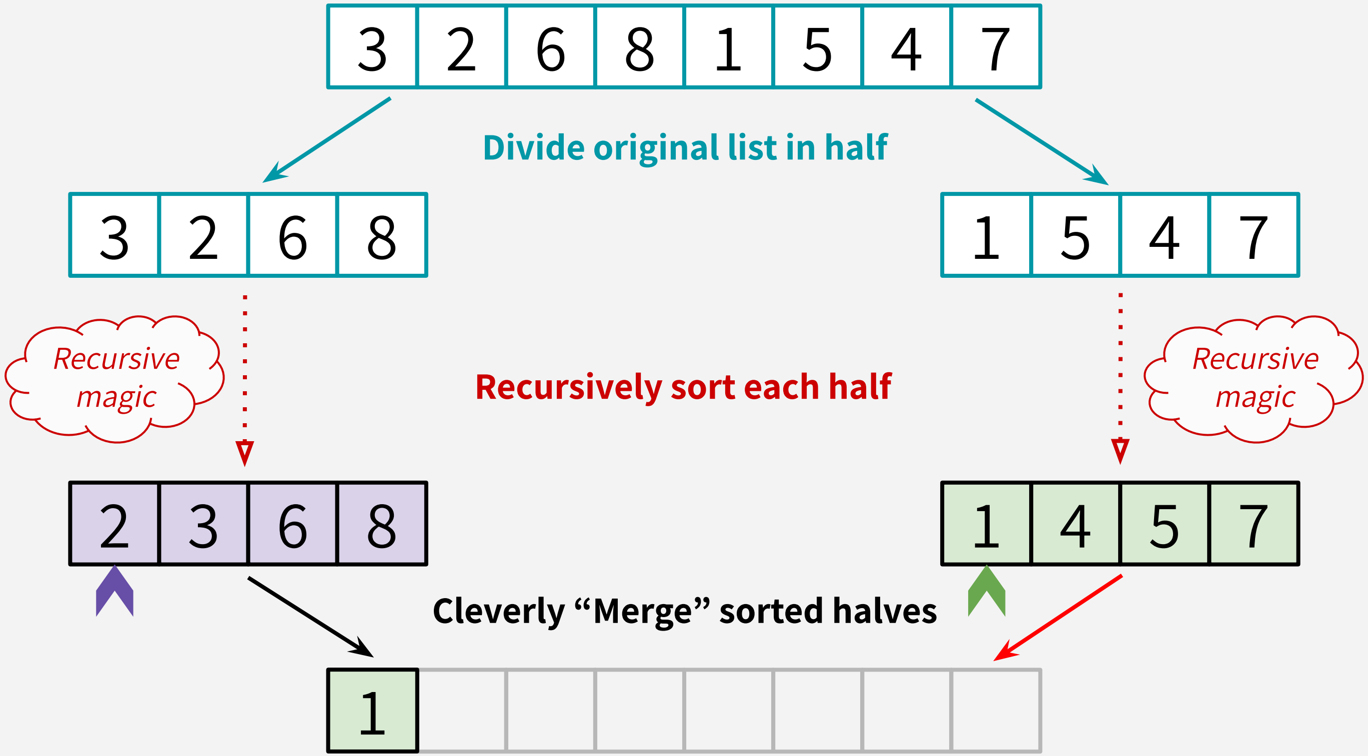


3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursive magic
Recursively sort each half
Recursive magic
2 3 6 8
1 4 5 7

MERGESORT

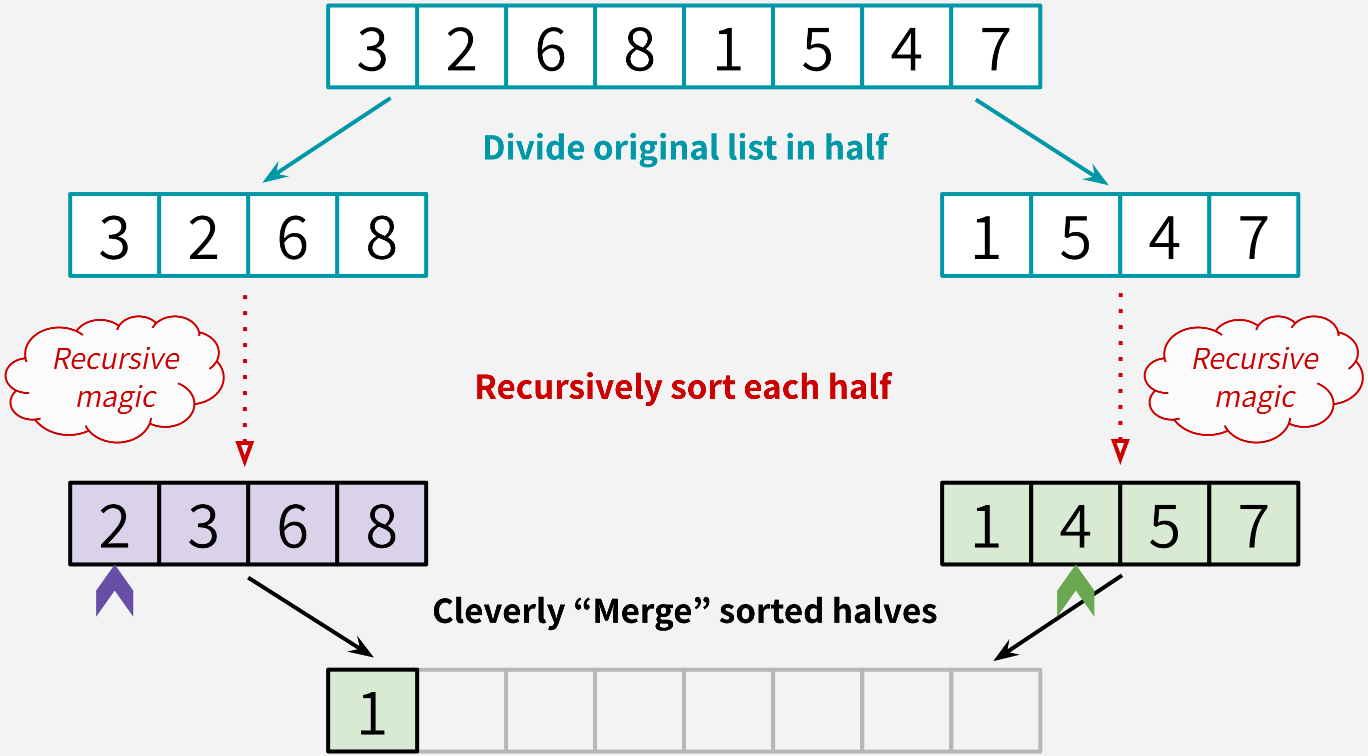

3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursive magic
Recursively sort each half
Recursive magic
2 3 6 8
1 4 5 7
Cleverly "Merge" sorted halves

MERGESORT



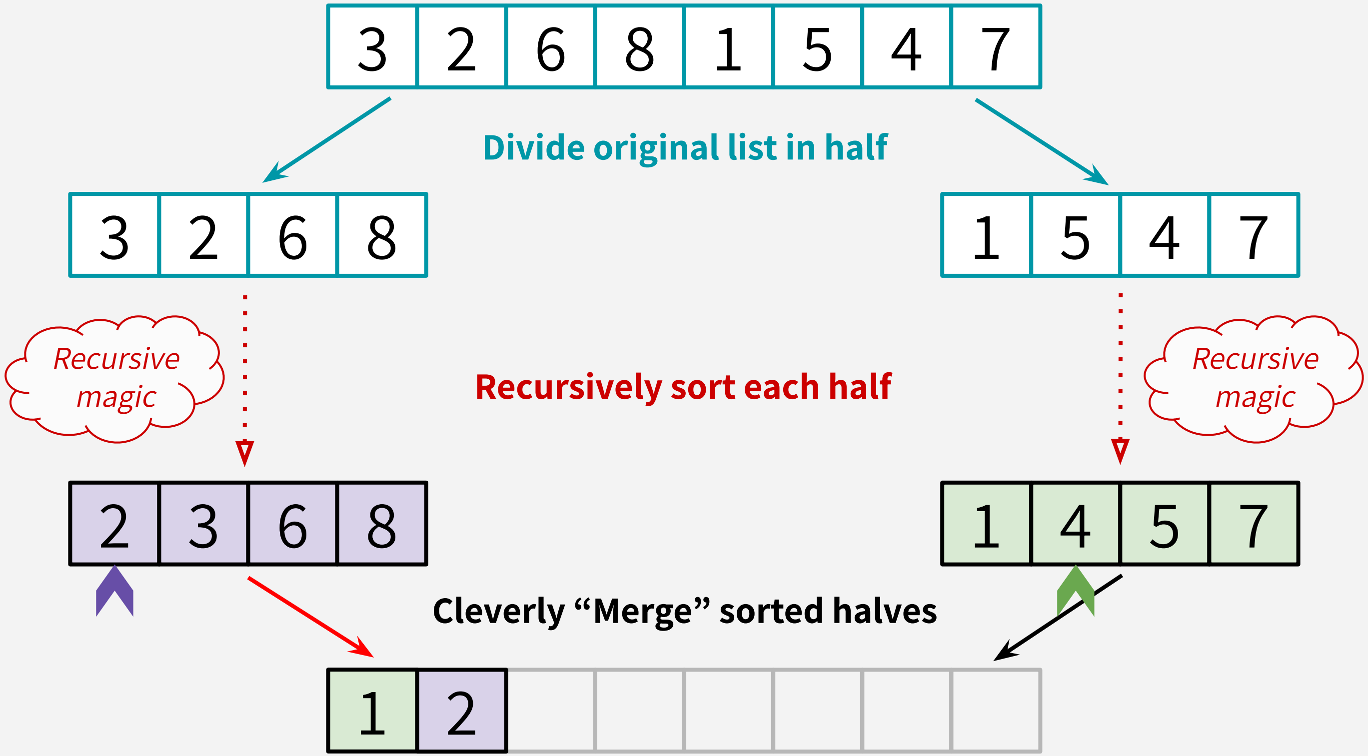
3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursive magic
Recursively sort each half
Recursive magic
2 3 6 8
1 4 5 7
Cleverly "Merge" sorted halves
1

MERGESORT



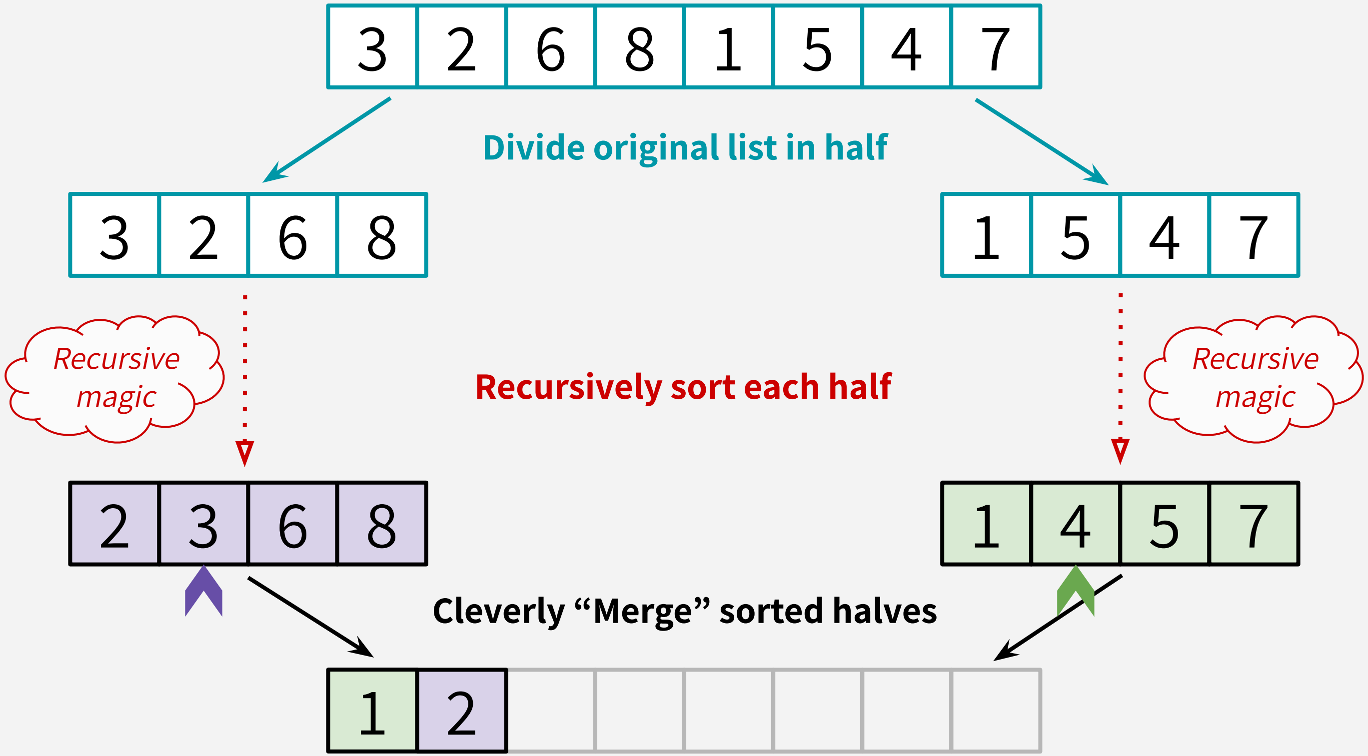
3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursive magic
Recursively sort each half
Recursive magic
2 3 6 8
1 4 5 7
Cleverly "Merge" sorted halves
1

MERGESORT



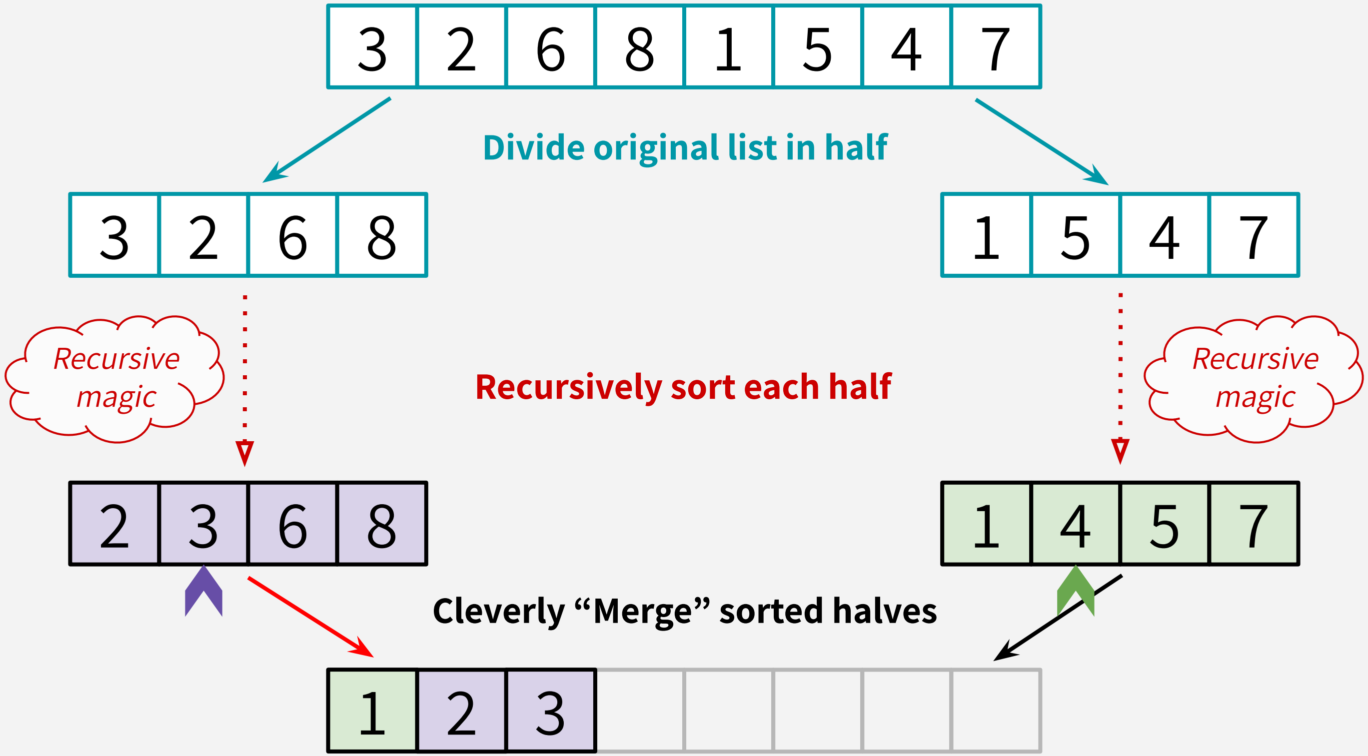
3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursive magic
Recursively sort each half
Recursive magic
2 3 6 8
1 4 5 7
Cleverly "Merge" sorted halves
1 2

MERGESORT



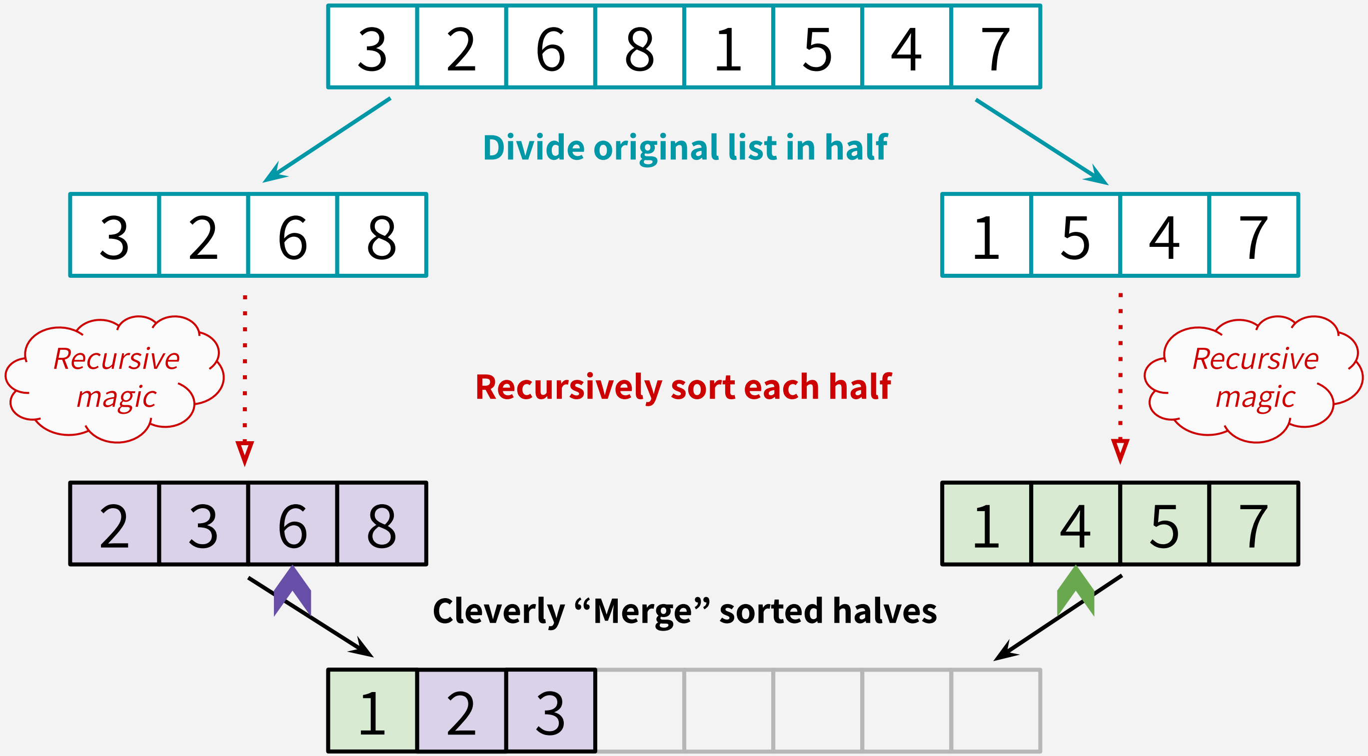
3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursive magic
Recursively sort each half
Recursive magic
2 3 6 8
1 4 5 7
Cleverly "Merge" sorted halves
1 2

MERGESORT



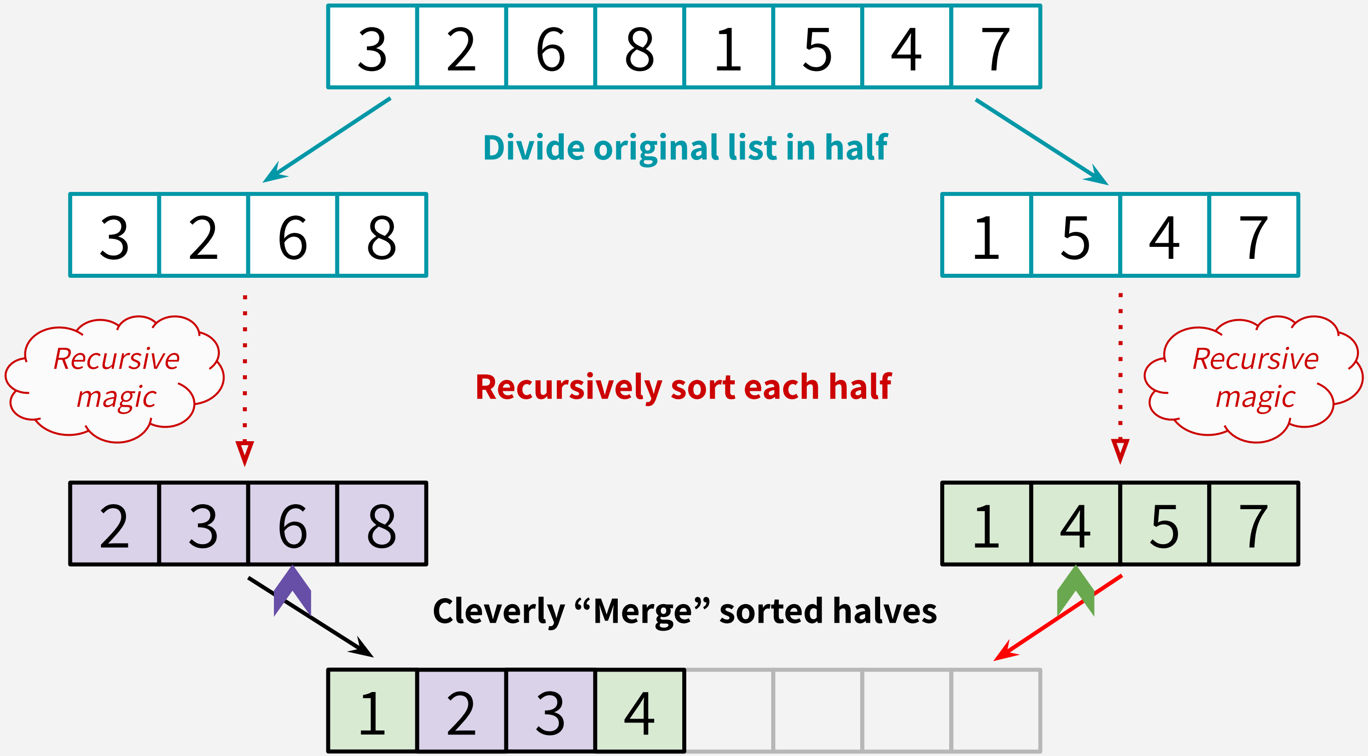
3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursive magic
Recursively sort each half
Recursive magic
2 3 6 8
1 4 5 7
Cleverly “Merge” sorted halves
1 2 3

MERGESORT



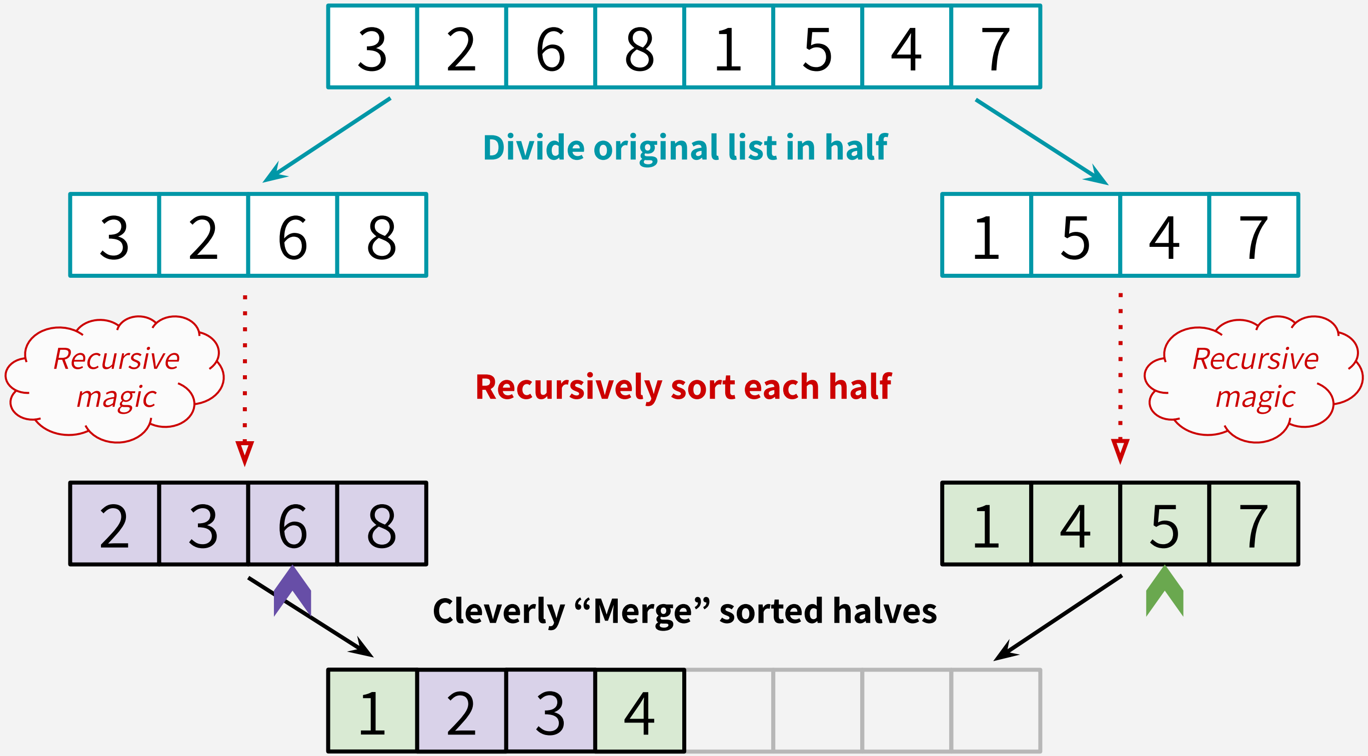
3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursive magic
Recursively sort each half
Recursive magic
2 3 6 8
1 4 5 7
Cleverly "Merge" sorted halves
1 2 3

MERGESORT



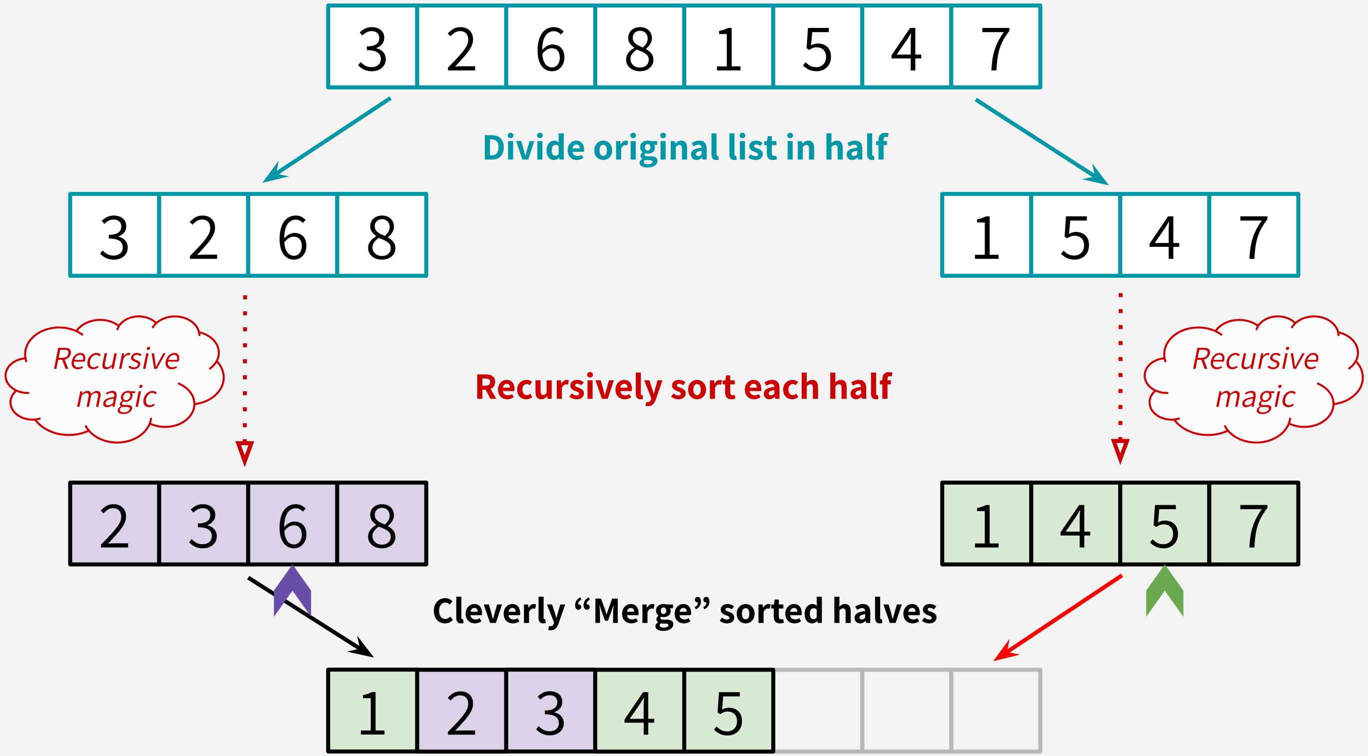
3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursive magic
Recursively sort each half
Recursive magic
2 3 6 8
1 4 5 7
Cleverly "Merge" sorted halves
1 2 3 4

MERGESORT



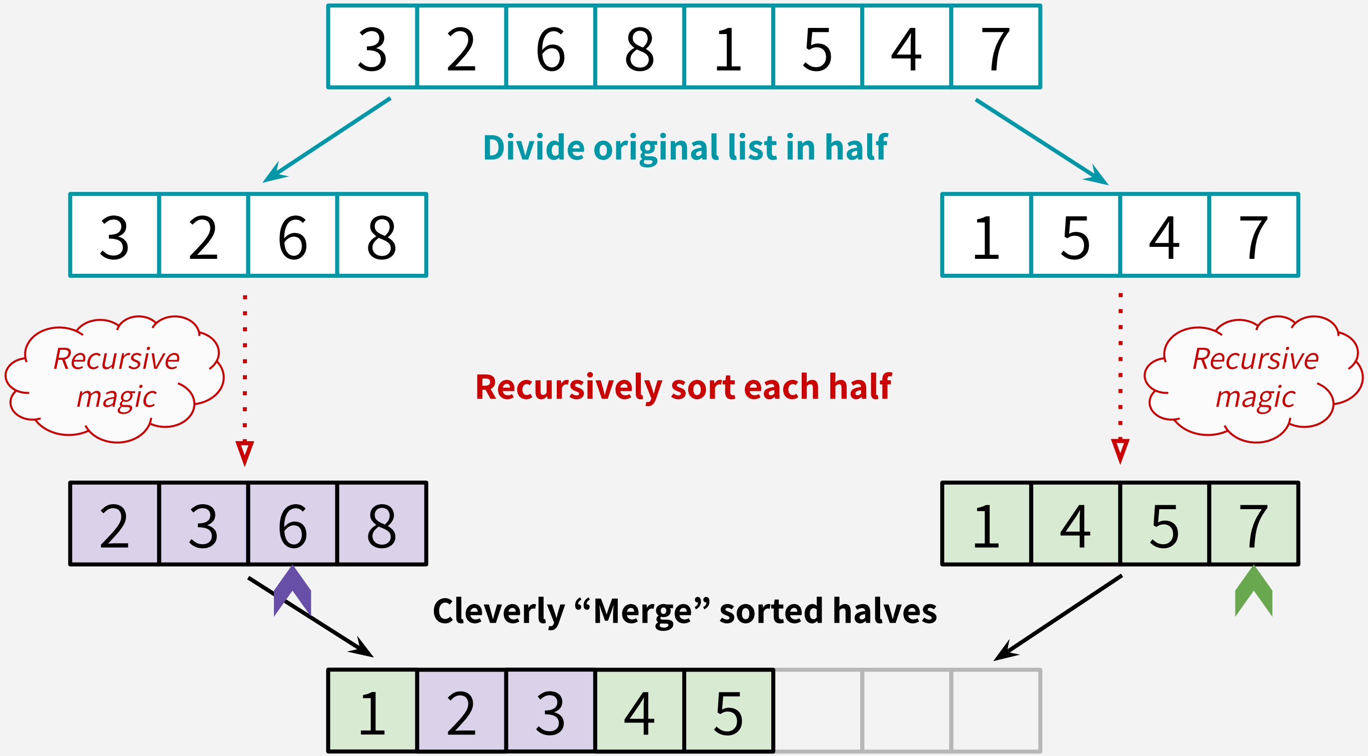
3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursive magic
Recursively sort each half
Recursive magic
2 3 6 8
1 4 5 7
Cleverly "Merge" sorted halves
1 2 3 4

MERGESORT



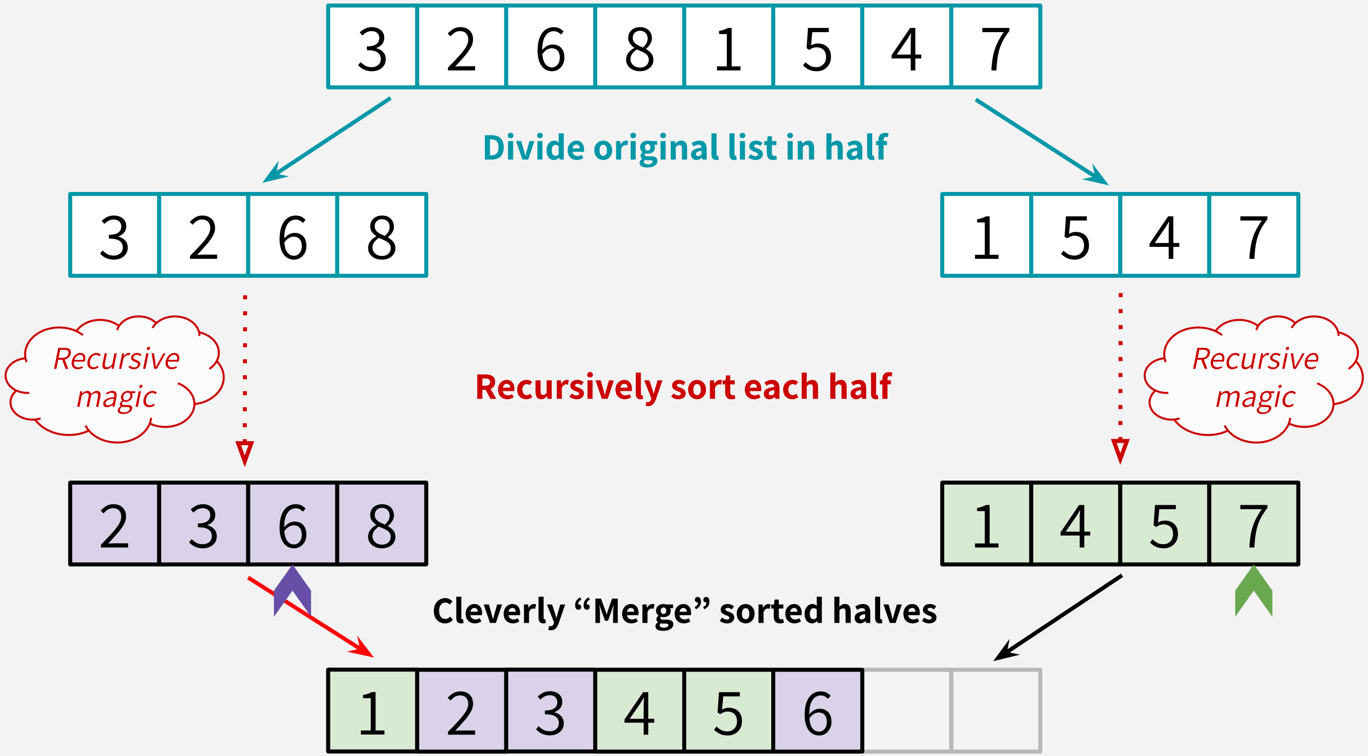
3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursive magic
Recursively sort each half
Recursive magic
2 3 6 8
1 4 5 7
Cleverly "Merge" sorted halves
1 2 3 4 5

MERGESORT

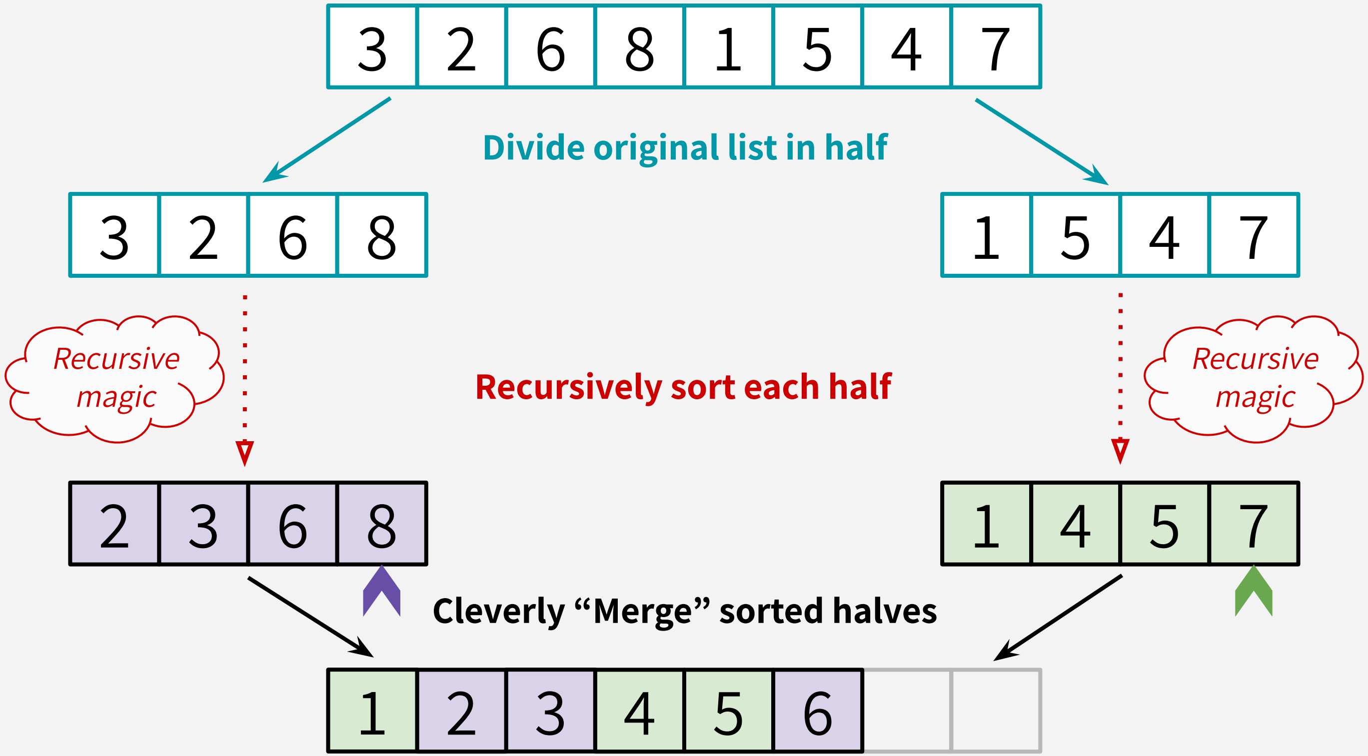


3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursive magic
Recursively sort each half
Recursive magic
2 3 6 8
1 4 5 7
Cleverly "Merge" sorted halves
1 2 3 4 5

MERGESORT

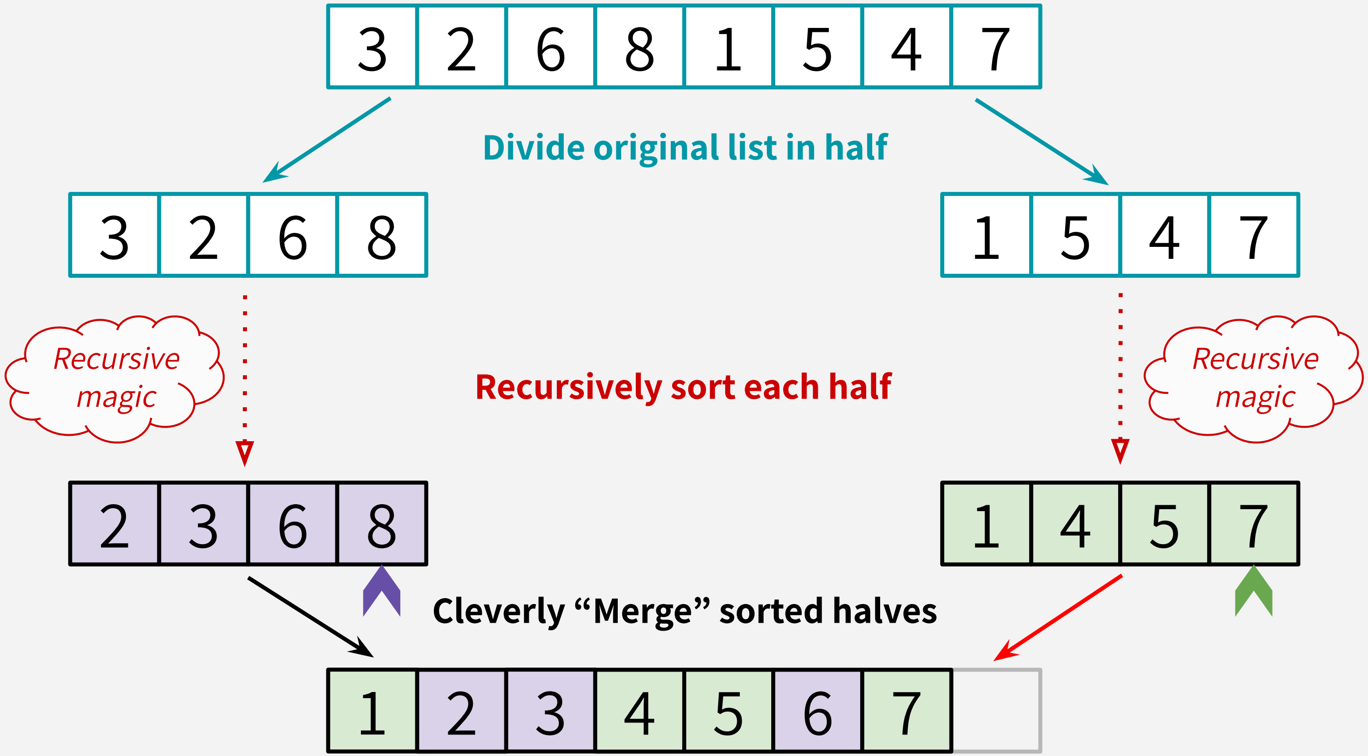

3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursive magic
Recursively sort each half
Recursive magic
2 3 6 8
1 4 5 7
Cleverly “Merge” sorted halves
1 2 3 4 5 6

MERGESORT



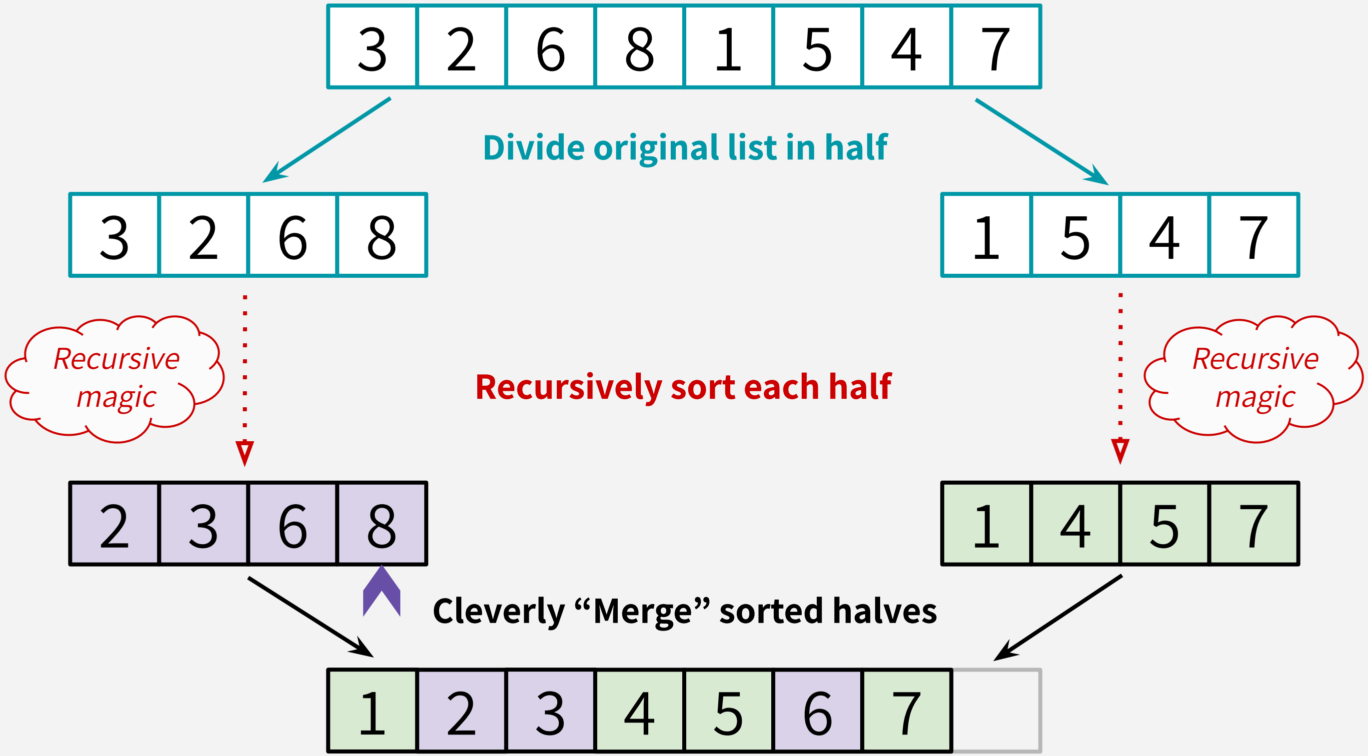
3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursive magic
Recursively sort each half
Recursive magic
2 3 6 8
1 4 5 7
Cleverly "Merge" sorted halves
1 2 3 4 5 6

MERGESORT



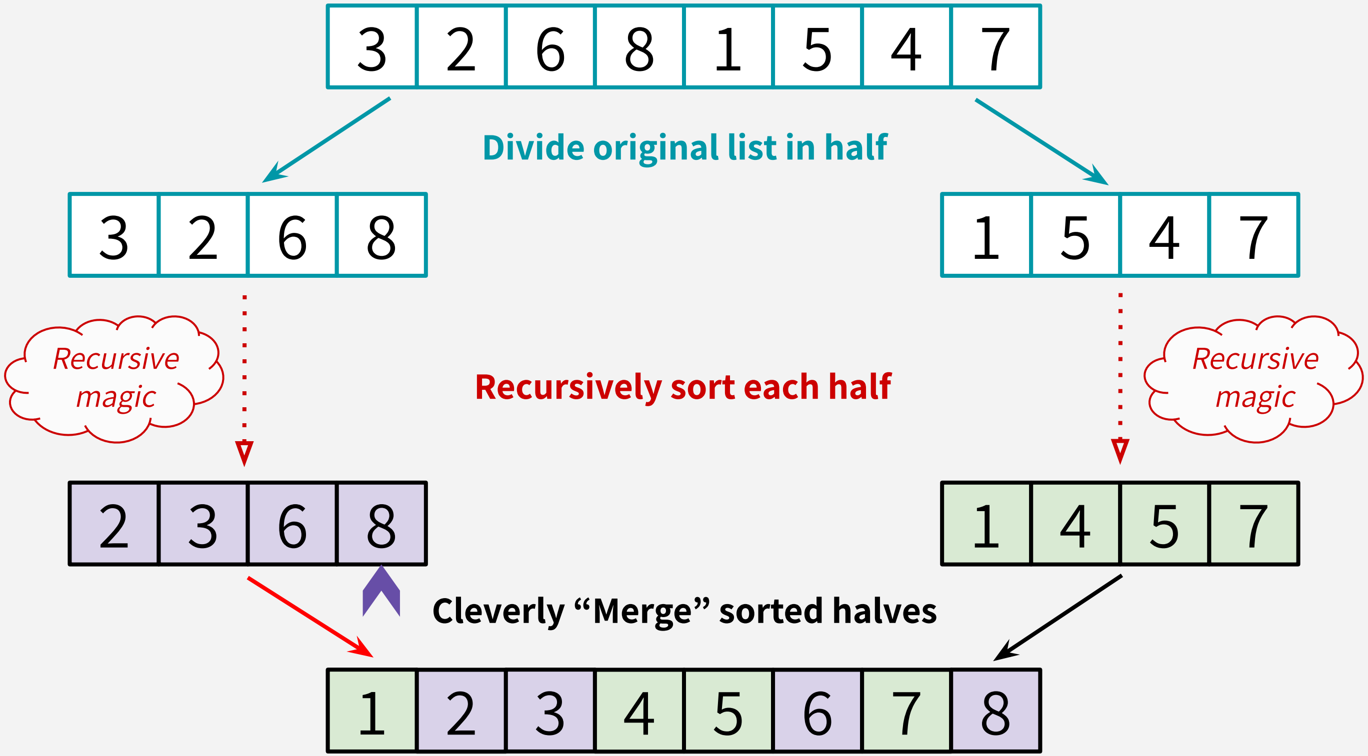
3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursive magic
Recursively sort each half
Recursive magic
2 3 6 8
1 4 5 7
Cleverly "Merge" sorted halves
1 2 3 4 5 6 7

MERGESORT

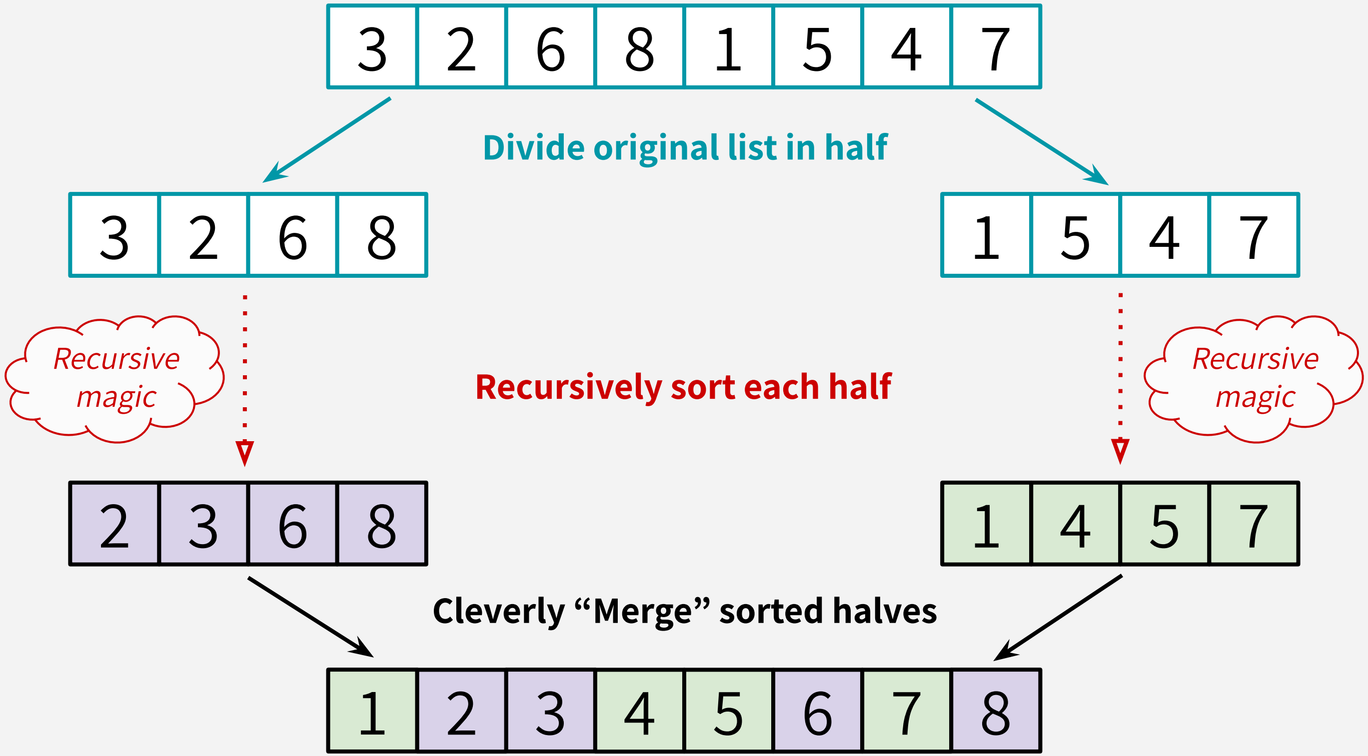


3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursive magic
Recursively sort each half
Recursive magic
2 3 6 8
1 4 5 7
Cleverly “Merge” sorted halves
1 2 3 4 5 6 7

MERGESORT


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

MERGESORT



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

MERGESORT: PSEUDOCODE

Intuition: Divide and Conquer. If you sort your left and right halves, it's easier to "Merge" them into a sorted list.

```
MERGESORT(A):
```

MERGESORT: PSEUDOCODE

Intuition: Divide and Conquer. If you sort your left and right halves, it's easier to "Merge" them into a sorted list.

```
MERGESORT(A):
    n = len(A)
    if n <= 1:
        return A
```

MERGESORT: PSEUDOCODE

Intuition: Divide and Conquer. If you sort your left and right halves, it's easier to "Merge" them into a sorted list.

```
MERGESORT(A):
    n = len(A)
    if n <= 1:
        return A
    L = MERGESORT(A[0:n/2])
```

MERGESORT: PSEUDOCODE

Intuition: Divide and Conquer. If you sort your left and right halves, it's easier to "Merge" them into a sorted list.

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

MERGESORT: PSEUDOCODE

Intuition: Divide and Conquer. If you sort your left and right halves, it's easier to "Merge" them into a sorted list.

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

For today, let's assume that n is a power of 2.

MERGESORT: PSEUDOCODE

Intuition: Divide and Conquer. If you sort your left and right halves, it's easier to "Merge" them into a sorted list.

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

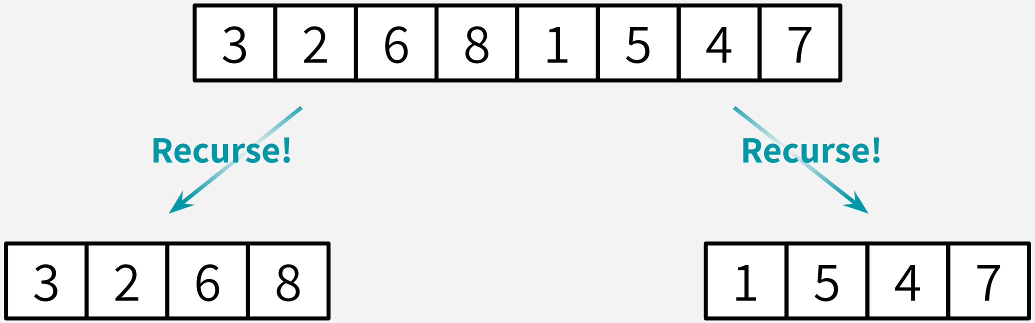
```
MERGE*(L,R):
    result = length n array
    i = 0, j = 0
    for k in [0,...,n-1]:
        if L[i] < R[j]:
            result[k] = L[i]
            i += 1
        else:
            result[k] = R[j]
            j += 1
    return result
```

* Not complete! Some corner cases are missing.

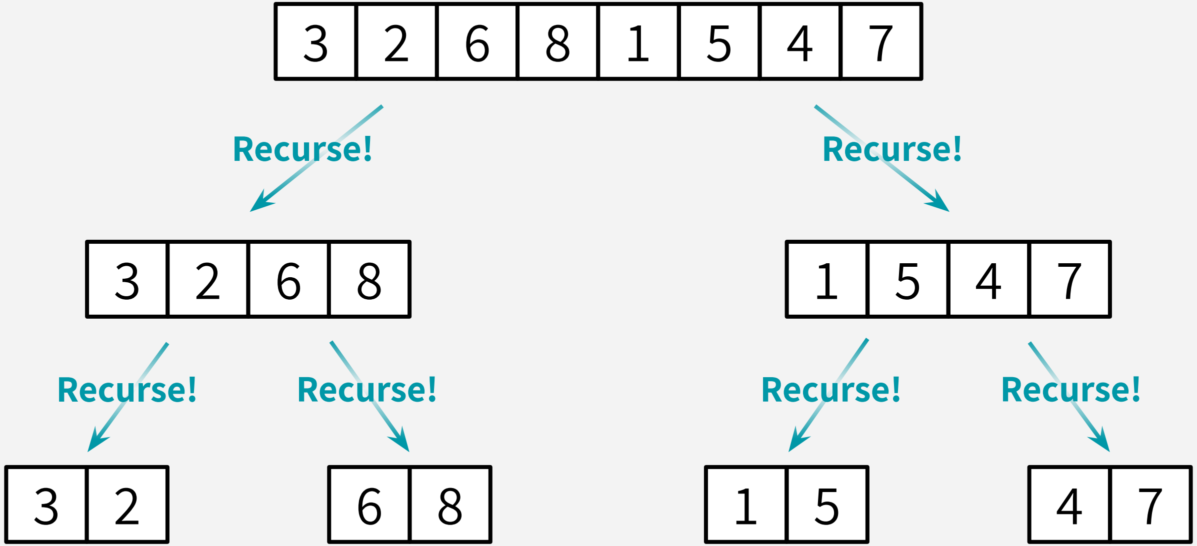
MERGESORT: RECURSIVE CALLS

3	2	6	8	1	5	4	7

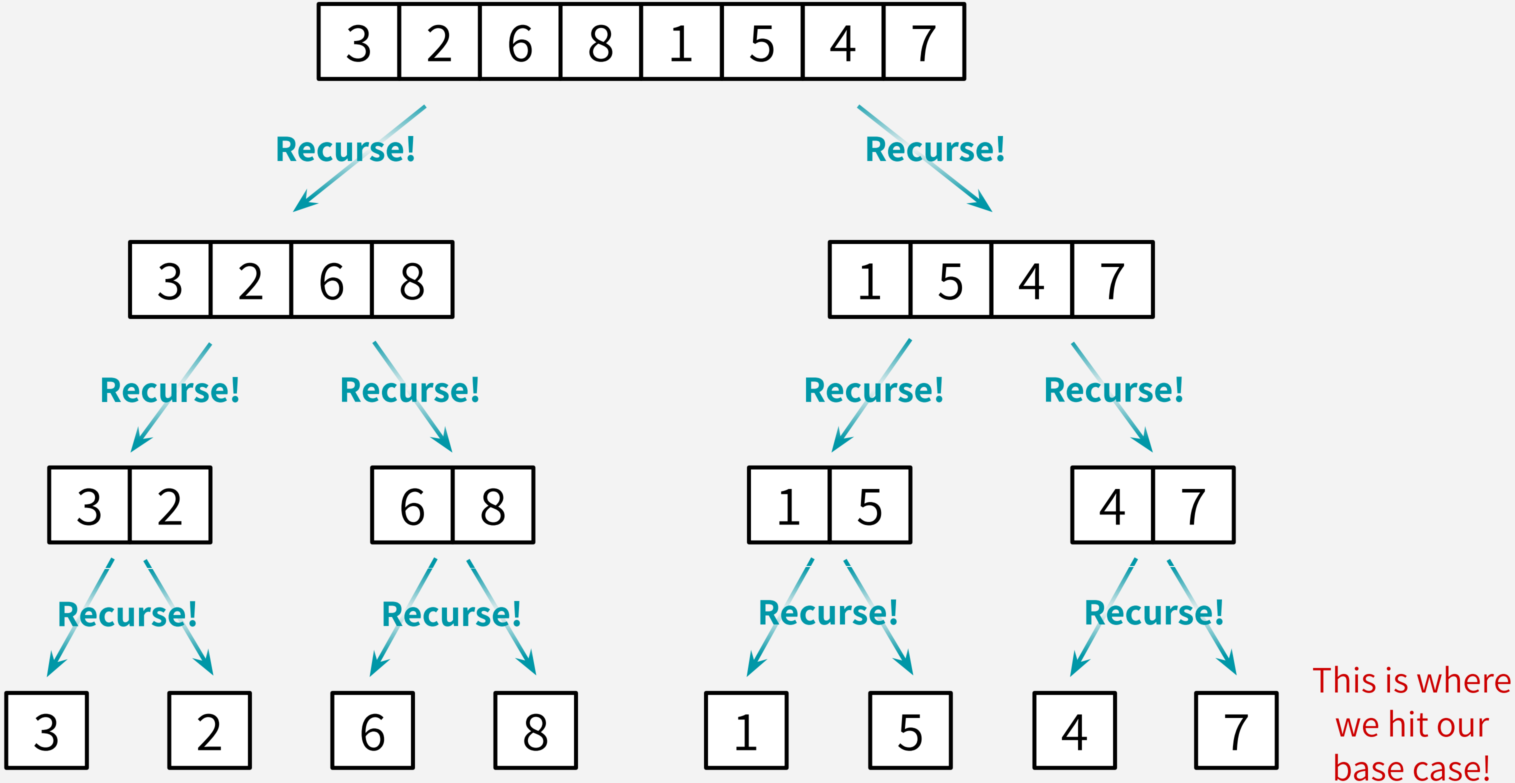
MERGESORT: RECURSIVE CALLS



MERGESORT: RECURSIVE CALLS



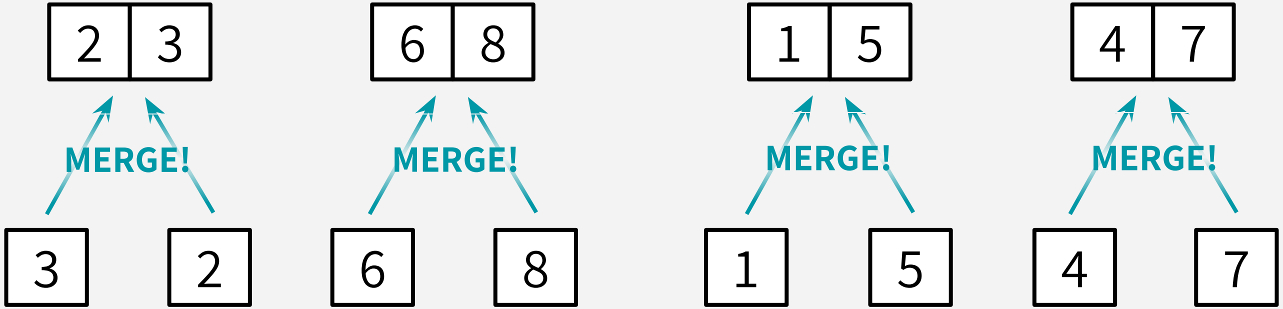
MERGESORT: RECURSIVE CALLS



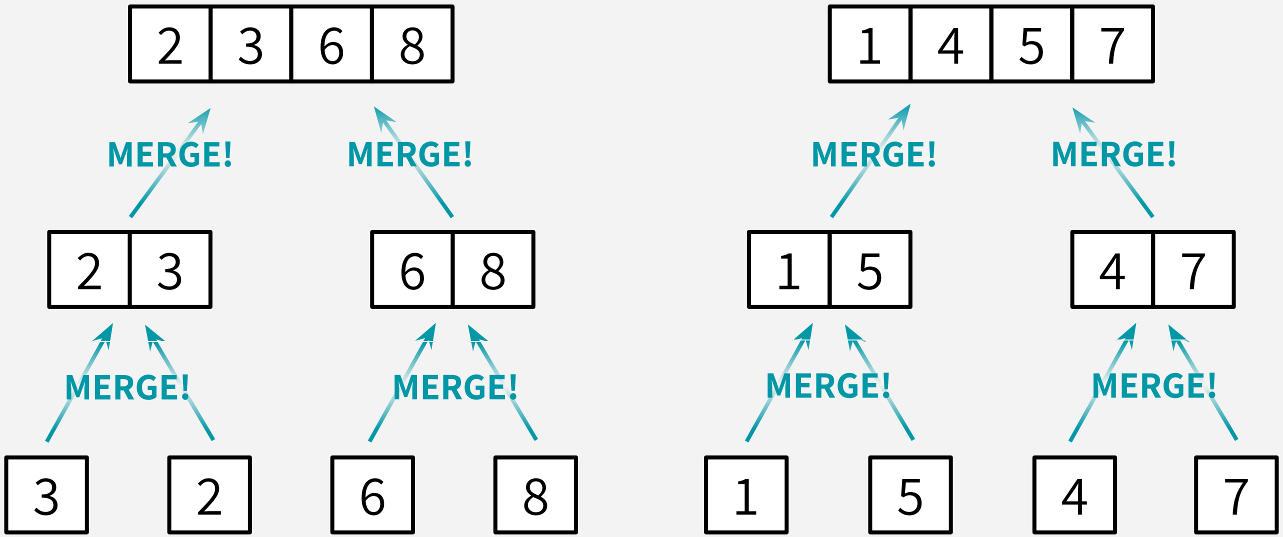
MERGESORT: MERGE STEPS

3 2 6 8 1 5 4 7

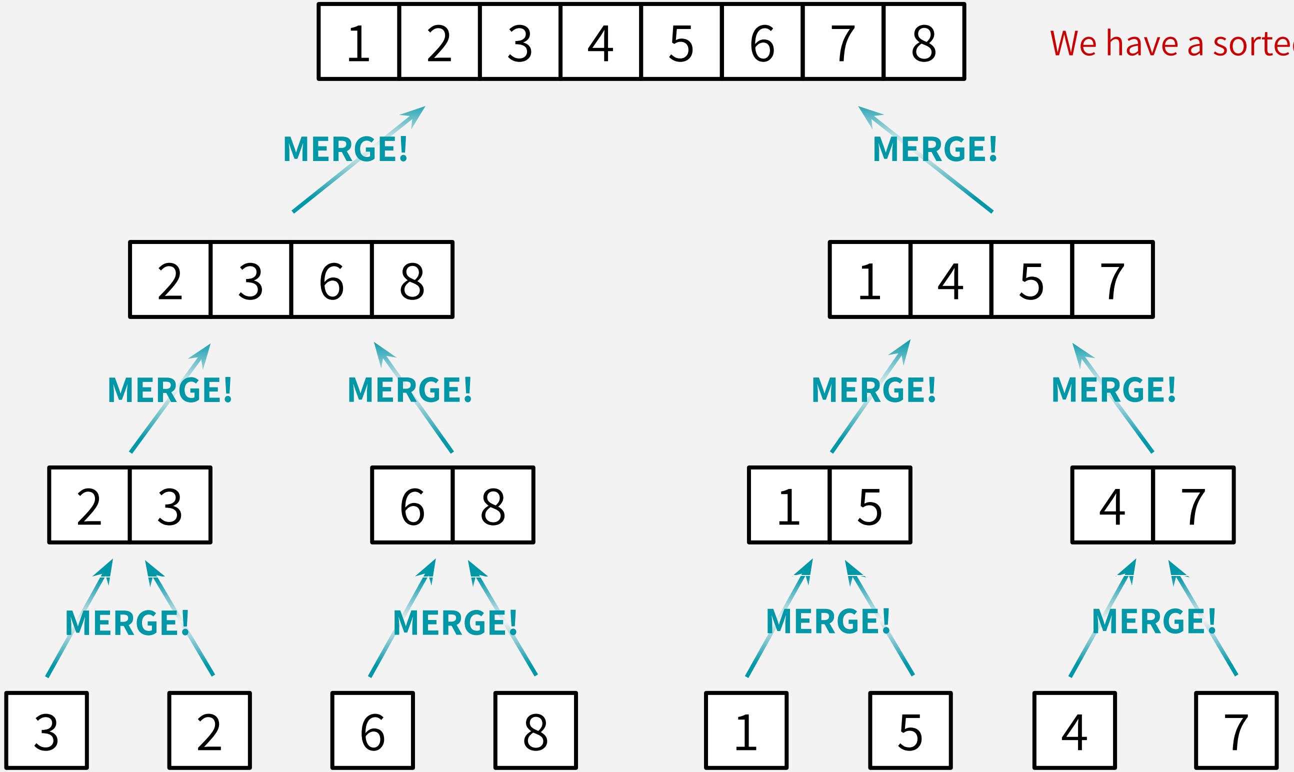
MERGESORT: MERGE STEPS



MERGESORT: MERGE STEPS



MERGESORT: MERGE STEPS



We have a sorted sequence!

سوال؟

اثبات درستی مرتب سازی ادغامی

آیا واقعا ورودی را مرتب میکند؟

MERGESORT: DOES IT WORK?

HERE'S WHAT WE FOCUS ON:

Whenever we make two "child" recursive calls, as long as those calls successfully sort our left and right halves, we'll safely merge them to create a fully sorted array.

In other words: as long as our recursive calls work on arrays of <u>smaller</u> lengths, then our algorithm will correctly return a sorted array.

MERGESORT: DOES IT WORK?

HERE'S WHAT WE FOCUS ON:

Whenever we make two "child" recursive calls, as long as those calls successfully sort our left and right halves, we'll safely merge them to create a fully sorted array.

In other words: as long as our recursive calls work on arrays of <u>smaller</u> lengths, then our algorithm will correctly return a sorted array.

THIS IS A JOB FOR: ***PROOF BY INDUCTION!***

(This time, we perform induction on the *length of input list*, rather than # of iterations)

MERGESORT: INDUCTION PROOF

INDUCTIVE HYPOTHESIS (IH)

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

MERGESORT: INDUCTION 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.

MERGESORT: INDUCTION 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.

Try out this inner proof on your own!

MERGESORT: INDUCTION 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]

Try out this inner proof on your own!

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.

PROVE CORRECTNESS w/ INDUCTION

ITERATIVE ALGORITHMS

RECURSIVE ALGORITHMS

PROVE CORRECTNESS w/ INDUCTION

ITERATIVE ALGORITHMS

1. **Inductive hypothesis**: some state/condition will always hold throughout your algorithm by any iteration **i**
2. **Base case**: show IH holds for iteration 0 (i.e. start of algorithm)
3. **Inductive step**: Assume IH holds for k ⇒ prove k+1
4. **Conclusion**: IH holds for i = # total iterations ⇒ yay!

RECURSIVE ALGORITHMS

PROVE CORRECTNESS w/ INDUCTION

ITERATIVE ALGORITHMS

1. **Inductive hypothesis**: some state/condition will always hold throughout your algorithm by any iteration **i**
2. **Base case**: show IH holds for iteration 0 (i.e. start of algorithm)
3. **Inductive step**: Assume IH holds for k ⇒ prove k+1
4. **Conclusion**: IH holds for i = # total iterations ⇒ yay!

RECURSIVE ALGORITHMS

1. **Inductive hypothesis**: your algorithm is correct for sizes *up to* **i**
2. **Base case**: IH holds for i < small constant
3. **Inductive step**:
    - assume IH holds for k ⇒ prove k+1, *OR*
    - assume IH holds for {1,2,...,k-1} ⇒ prove k.
4. **Conclusion**: IH holds for i = n ⇒ yay!

سوال؟

زمان اجرای مرتب سازی ادغامی

چقدر سریع است؟

MERGESORT: IS IT FAST?

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

CLAIM: MergeSort runs in time **O(n log n)**

AN ASIDE: $O(n \log n)$ vs. $O(n^2)$?

log(n) grows very slowly! (Much more slowly than n)

AN ASIDE: O(n log n) vs. O(n^2)?

log(n) grows very slowly! (Much more slowly than n)

ALL LOGARITHMS IN THIS COURSE ARE BASE 2 UNLESS EXPLICITLY MENTIONED

log(2) = 1

log(4) = 2

…

log(64) = 6

log (128) = 7

…

log(4096) = 12

…

log(**# particles in the universe**) < 280

AN ASIDE: O(n log n) vs. O(n^2)?

log(n) grows very slowly! (Much more slowly than n)

ALL LOGARITHMS IN THIS COURSE ARE BASE 2 UNLESS EXPLICITLY MENTIONED

log(2) = 1

log(4) = 2

…

log(64) = 6

log (128) = 7

…

log(4096) = 12

…

log(**# particles in the universe**) < 280

n log n grows much more slowly than n^2

Punchline: A running time of O(n log n) is a LOT better than O(n^2)

MERGESORT: O(n log n) PROOF

Instead of counting every little operation and tracing all recursive calls, we can think about:

THE RECURSION TREE!

(and we'll add up all the work done across levels to compute the Big-O runtime)

```
MERGESORT(A):
    n = len(A)
    if n <= 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: O(n log n) PROOF

Instead of counting every little operation and tracing all recursive calls, we can think about:

THE RECURSION TREE!

(and we'll add up all the work done across levels to compute the Big-O runtime)

```
MERGESORT(A):
    n = len(A)
    if n <= 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
```

n iterations, O(1) work per iteration

We can see that MERGE is **O(n)**

MERGESORT: O(n log n) PROOF

Instead of counting every little operation and tracing all recursive calls, we can think about:

THE RECURSION TREE!

(and we'll add up all the work done across levels to compute the Big-O runtime)

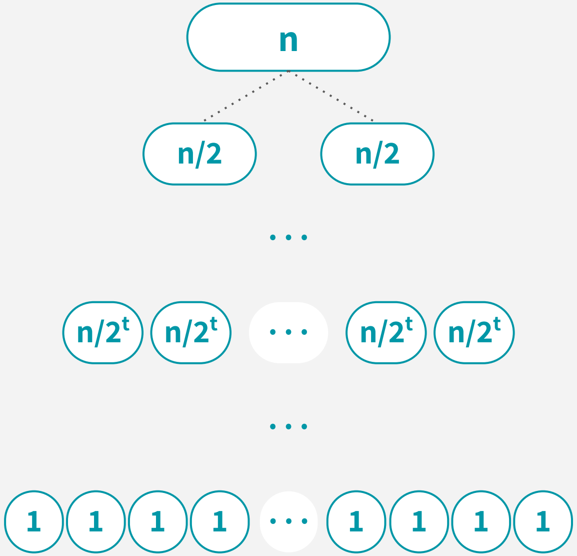
```
MERGESORT(A):
    n = len(A)
    if n <= 1:
        return A
    L = MERGESORT(A[0:n/2])
    R = MERGESORT(A[n/2:n])
    return MERGE(L,R)
```

```
MERGE(L,R):
    result = length n array
    ...
    return result
```

This means that within one recursive call that processes an array/subarray of length ***n***, the work done in that subproblem (creating subproblems & "merging" those results) is **O(n)**.

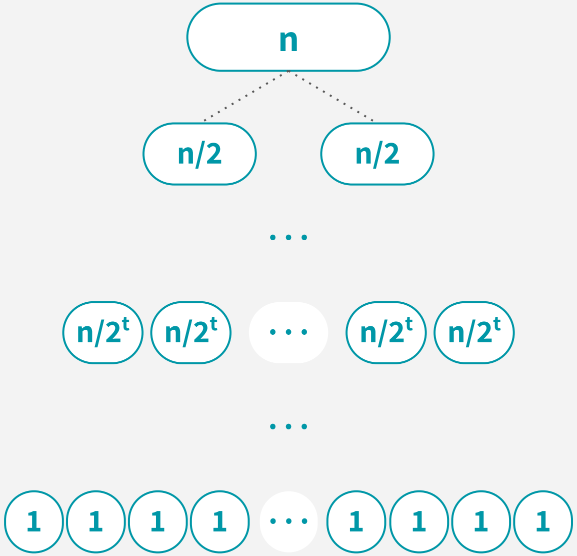
We can see that MERGE is **O(n)**

MERGESORT RECURSION TREE



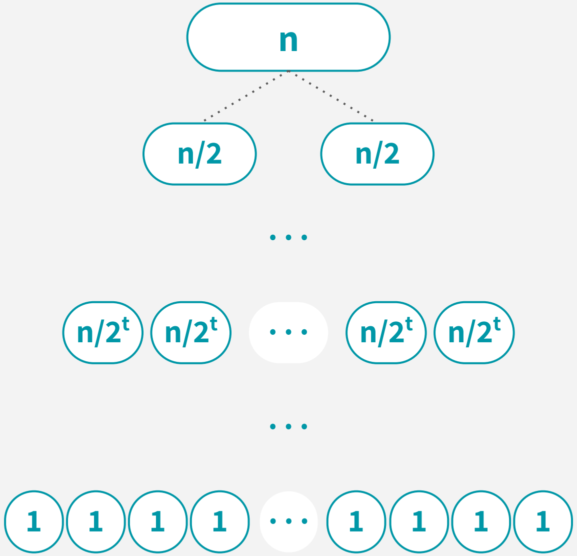
Level	Size of each Problem	# of Problems	Work done per Problem	Total work on this level
0				
1				
…				
t				
…				
?				

MERGESORT RECURSION TREE



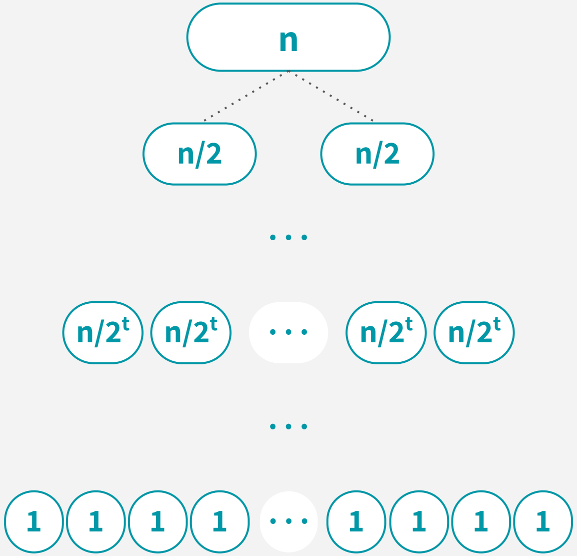
Level	Size of each Problem	# of Problems	Work done per Problem	Total work on this level
0				
1				
...				
t				
...				
$\log_2 n$				

MERGESORT RECURSION TREE



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

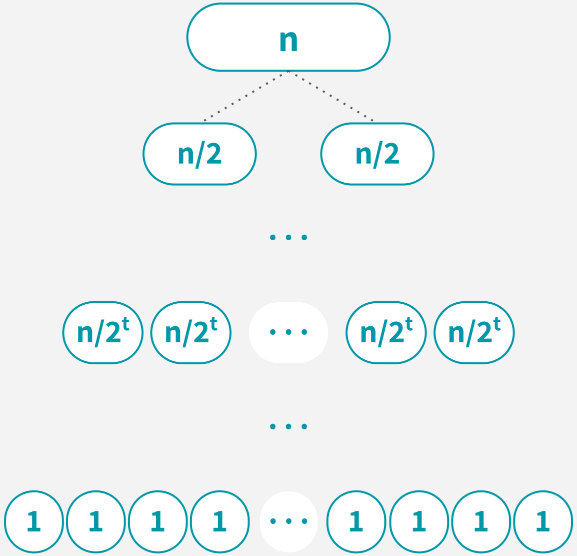
MERGESORT RECURSION TREE



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

MERGESORT RECURSION TREE

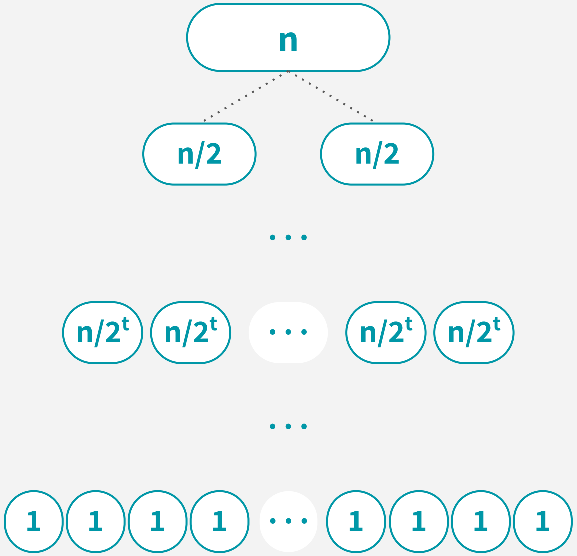
If a subproblem is of size **n**, then the work done in that subproblem is **O(n)**.
⇒ **Work ≤ c · n** (c is a constant)



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

MERGESORT RECURSION TREE

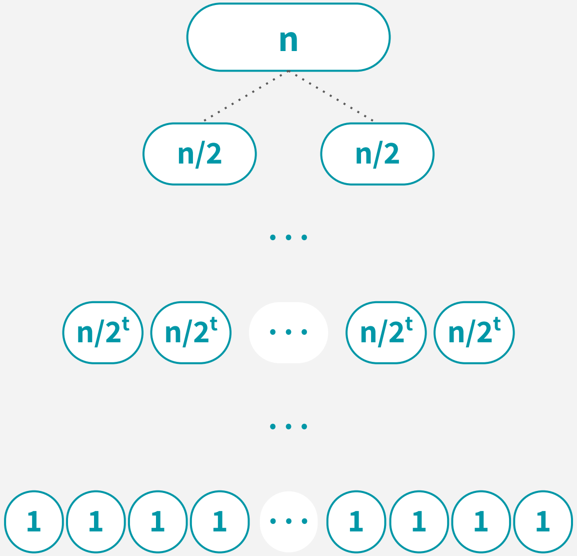
If a subproblem is of size **n**, then the work done in that subproblem is **O(n)**.
⇒ **Work ≤ c · n** (c is a constant)



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

MERGESORT RECURSION TREE

If a subproblem is of size **n**, then the work done in that subproblem is **O(n)**.
⇒ **Work ≤ c · n** (c is a constant)

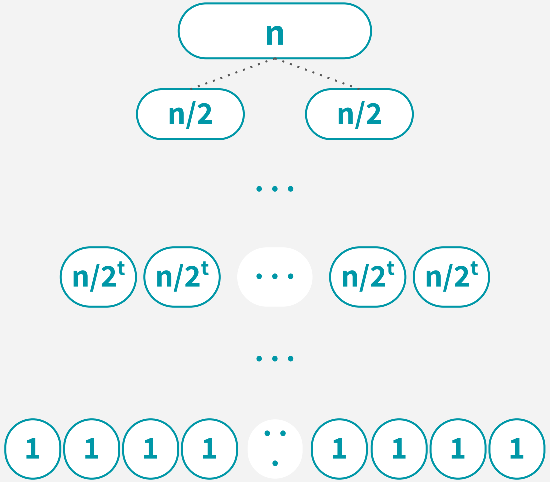


Level	Size of each Problem	# of Problems	Work done per Problem	Total work on this level
0	n	1	$c \cdot n$	**O(n)**
1	$n/2^1$	2^1	$c \cdot (n/2)$	$2^1 \cdot c \cdot (n/2) =$ **O(n)**
. . .				
t	$n/2^t$	2^t	$c \cdot (n/2^t)$	$2^t \cdot c \cdot (n/2^t) =$ **O(n)**
. . .				
$\log_2 n$	1	$2^{\log_2 n} =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

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

MERGESORT: O(n log n) RUNTIME

Using the "Recursion Tree Method" (i.e. drawing the tree & filling out the table),
we showed that the runtime of MergeSort is **O(n log n)**



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

سوال؟