Thinking algorithmically (also Class Quiz 1)

You are given an n-element sequence of S such that each element in S represents a different vote for the president, where each vote is given as an integer representing a particular candidate, yet the integers may be arbitrarily large (even if the number of candidates are not).

Design an O(n log n) time algorithm to see who wins the election S represents, assuming candidates with the most votes win.

Thinking algorithmically (also Class Quiz 2)

Thinking algorithmically (also Class Quiz 2)

Natsu wishes to find **Igneel**, the dragon who is like a father to him. But to find him he has to complete a certain number of tasks. Tasks are defined by a string of lower case English alphabets and he may be asked to complete the same task more than once. He has a list of N tasks, but he wants to complete the tasks which occur least frequently first. If two tasks have the same frequency the lexicographically smaller will be completed first.

Help him prepare such a list representing frequency of tasks and task name separated by a space.

MergeSort: The Algorithm

The first non-trivial algorithm!

John von Neumann (1945)

Recall key idea of Divide and Conquer

- Break problem into smaller sub-problems
- Solve the subproblems recursively
- Combine the solutions of the subproblems

Guiding principles for algorithm analysis

- Quest for running time bounds that hold for every input of a given size
- Rate of growth of the running time, as a function of the input size

Will analyse using recursion tree method

Problem	Sorting
Input	An array of n numbers, in arbitrary order
Output	An array of the same numbers, sorted from smallest to largest

Options

SelectionSort	scan to identify the minimum, then repeat
BubbleSort	identify adjacent pairs of elements that are out of order, and perform repeated swaps

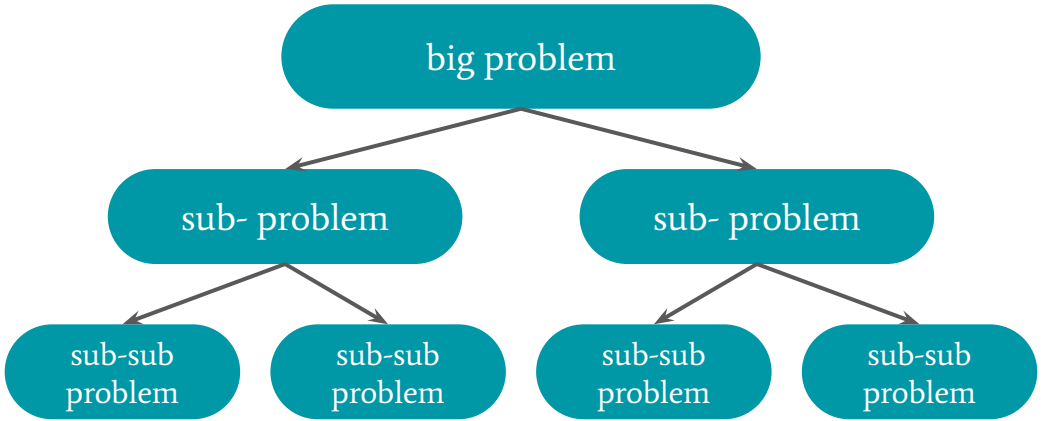
-- Quadratic running times! Can we do better?

Divide & Conquer

break up a problem into smaller subproblems

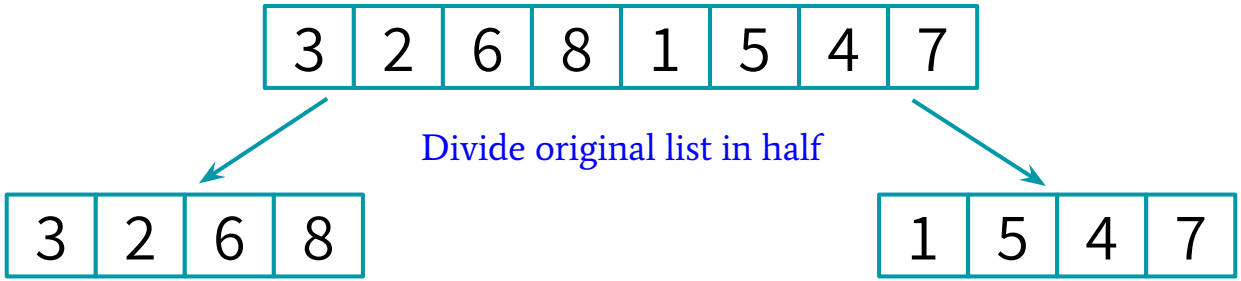
solve those subproblems *recursively*

combine the results of those subproblems to get the overall answer

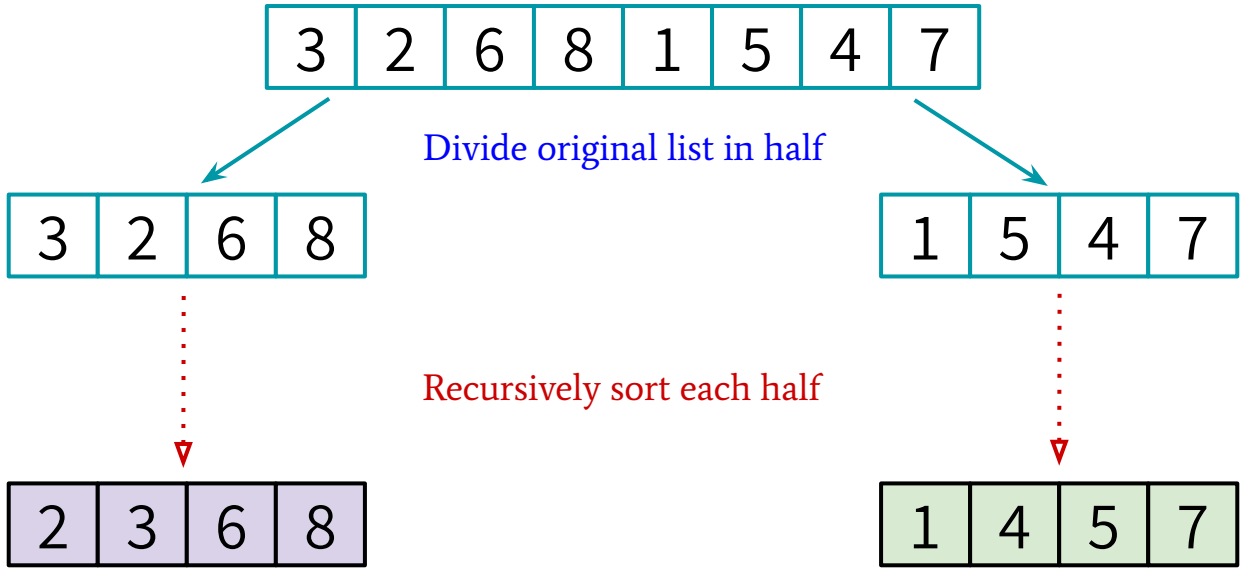


Merge Sort

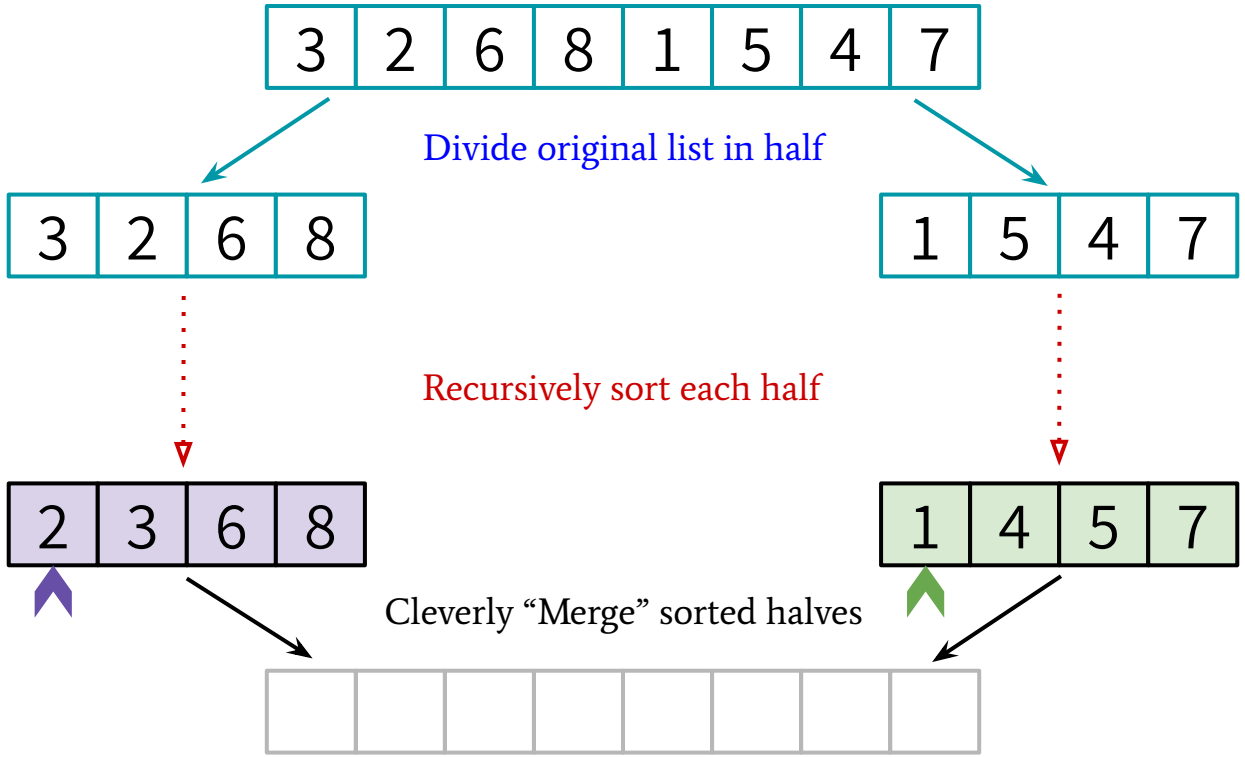
Merge Sort



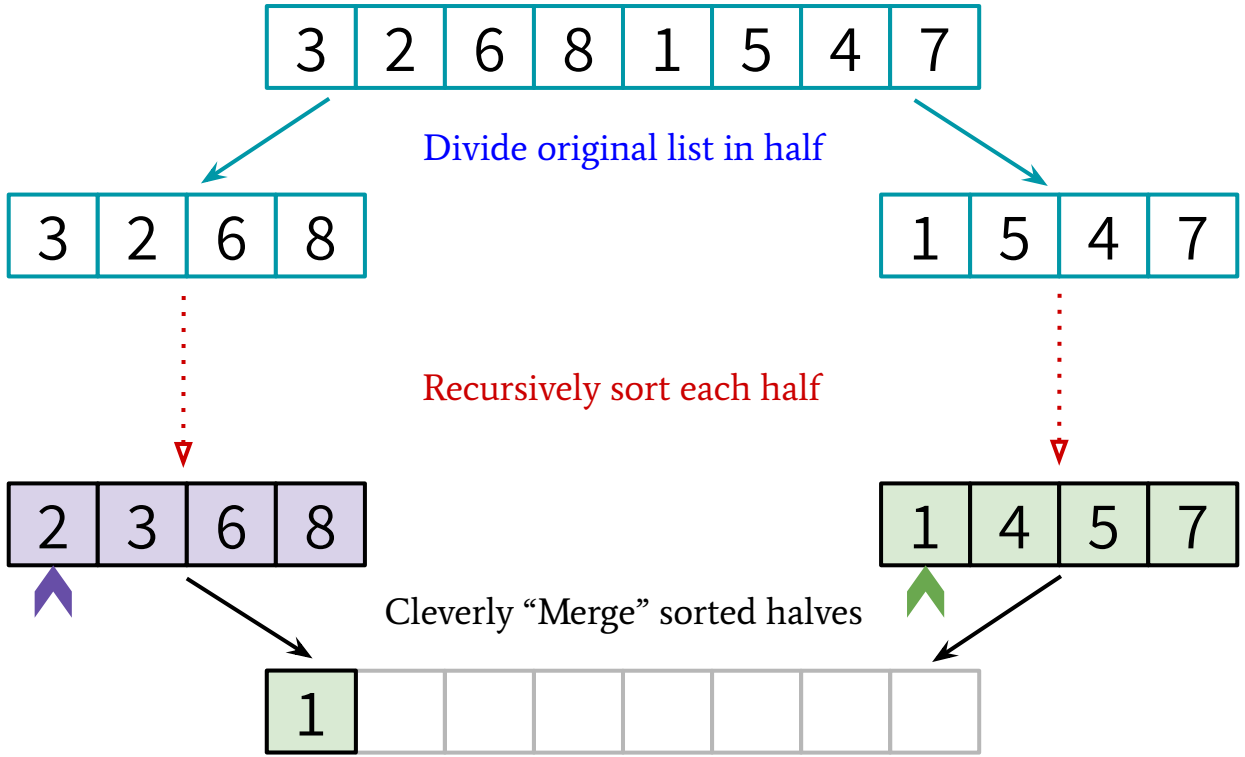
Merge Sort


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

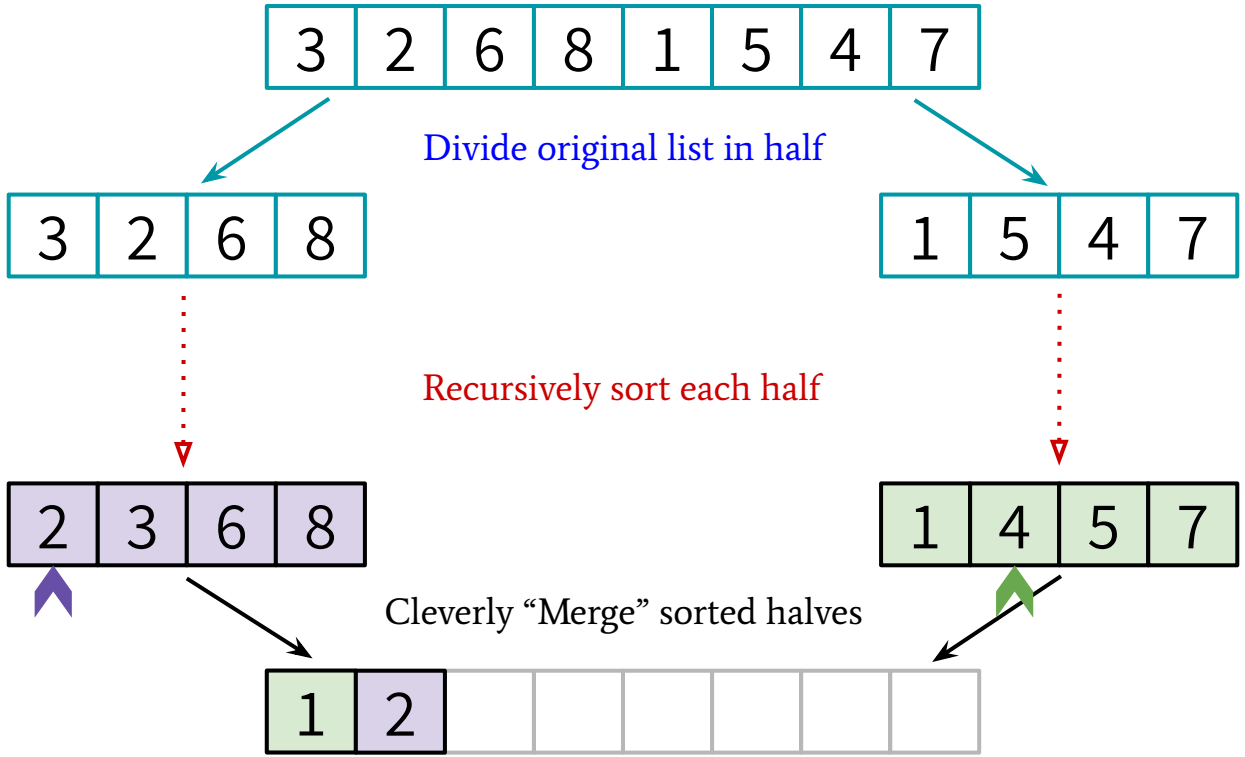
Merge Sort


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

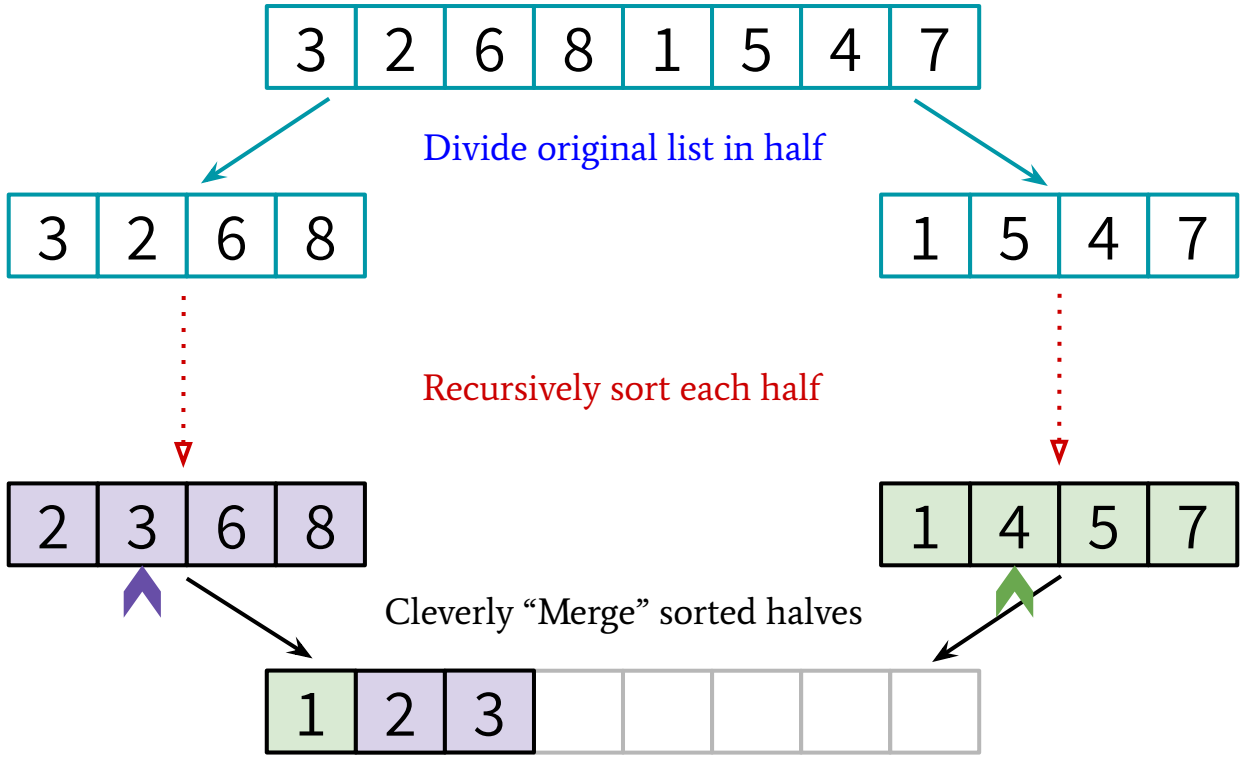
Merge Sort


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

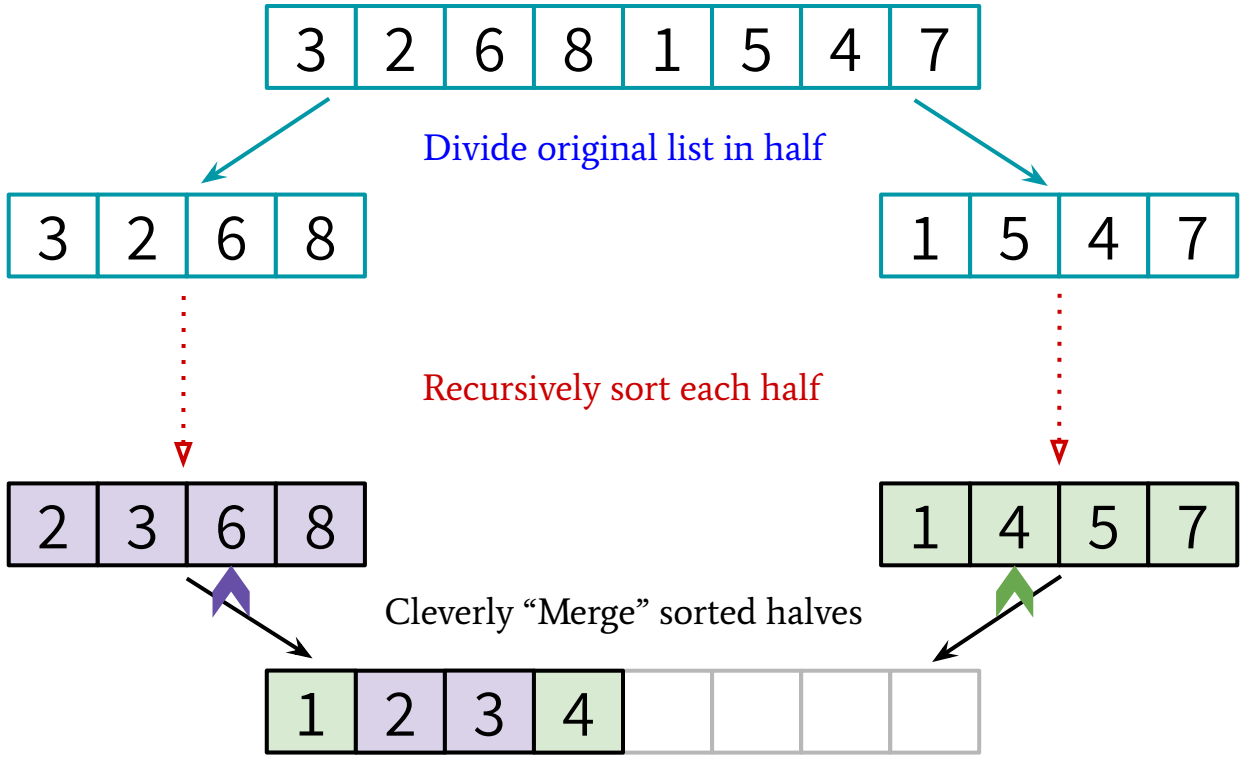
Merge Sort


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

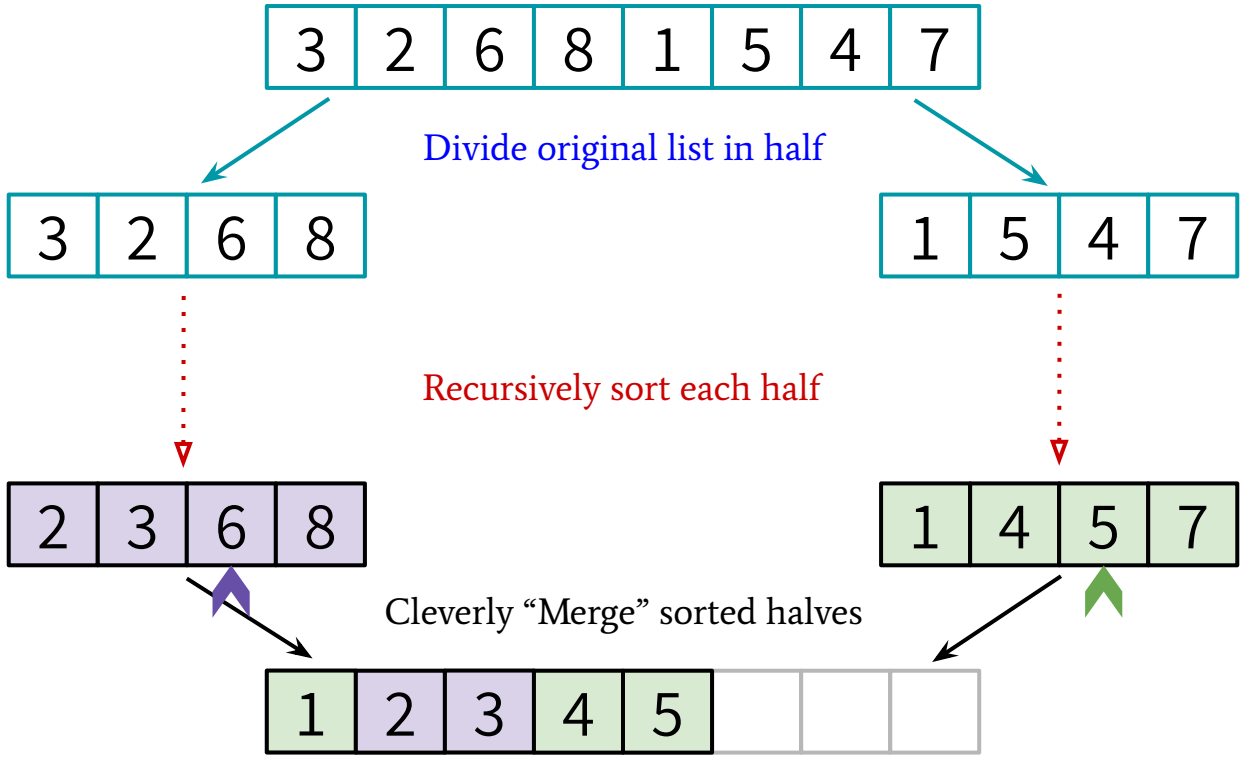
Merge Sort


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

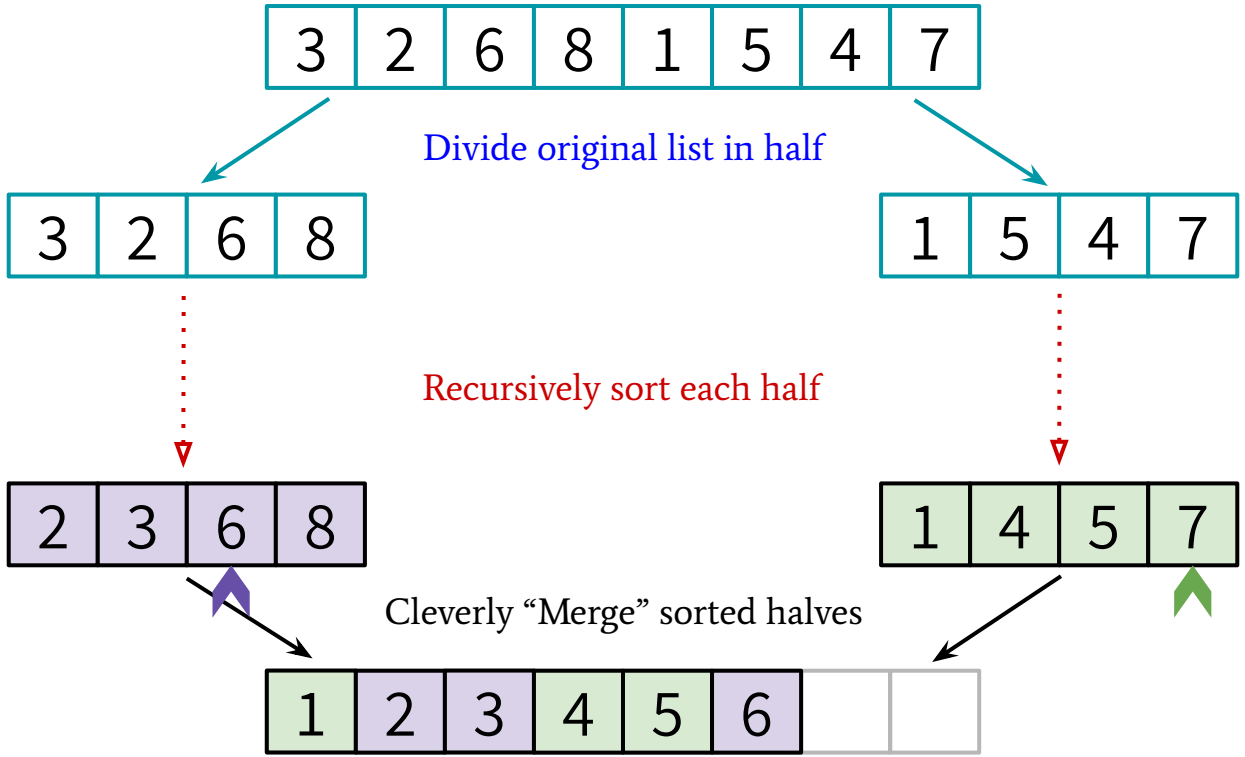
Merge Sort


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

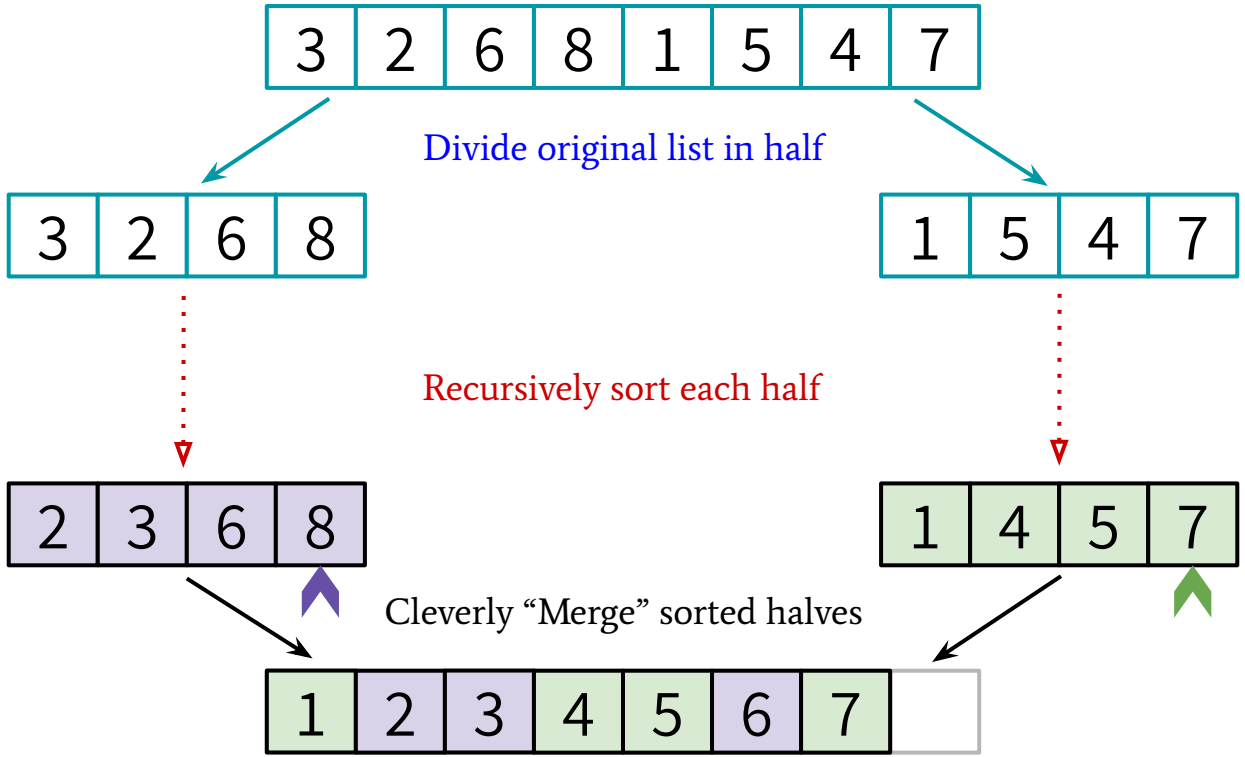
Merge Sort


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

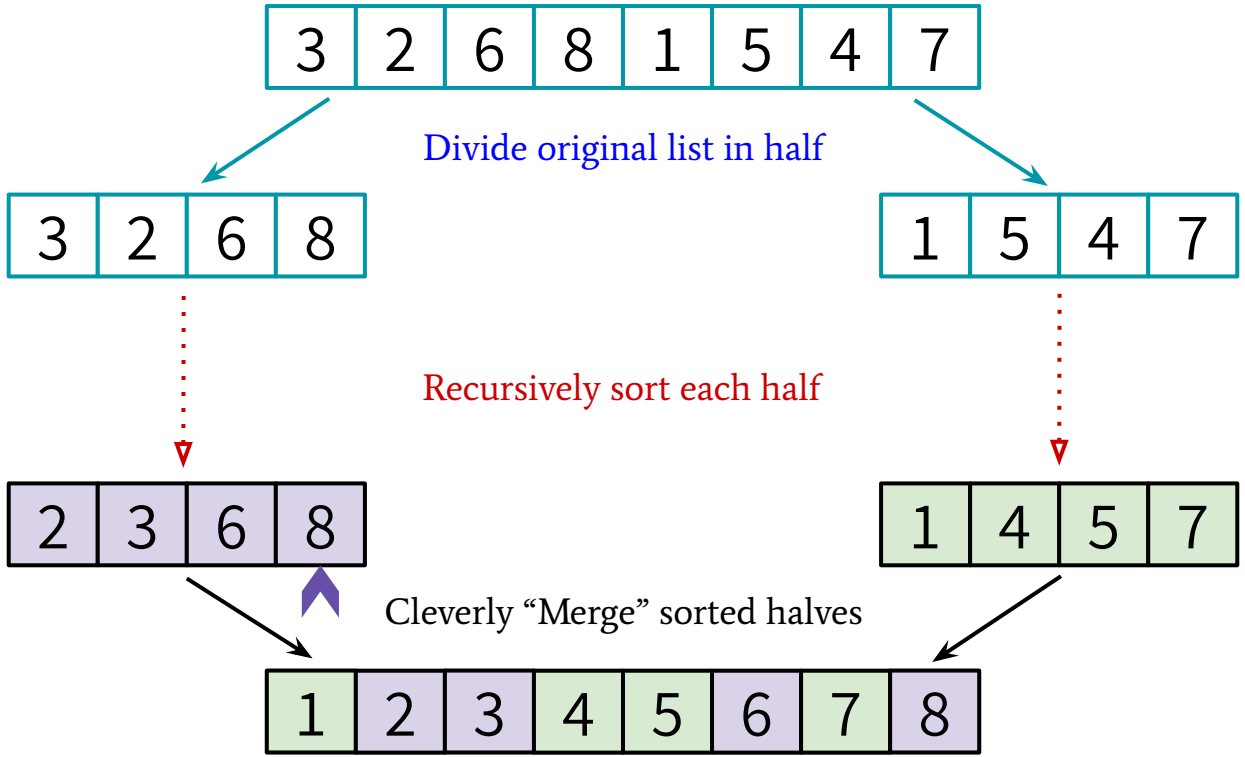
Merge Sort


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

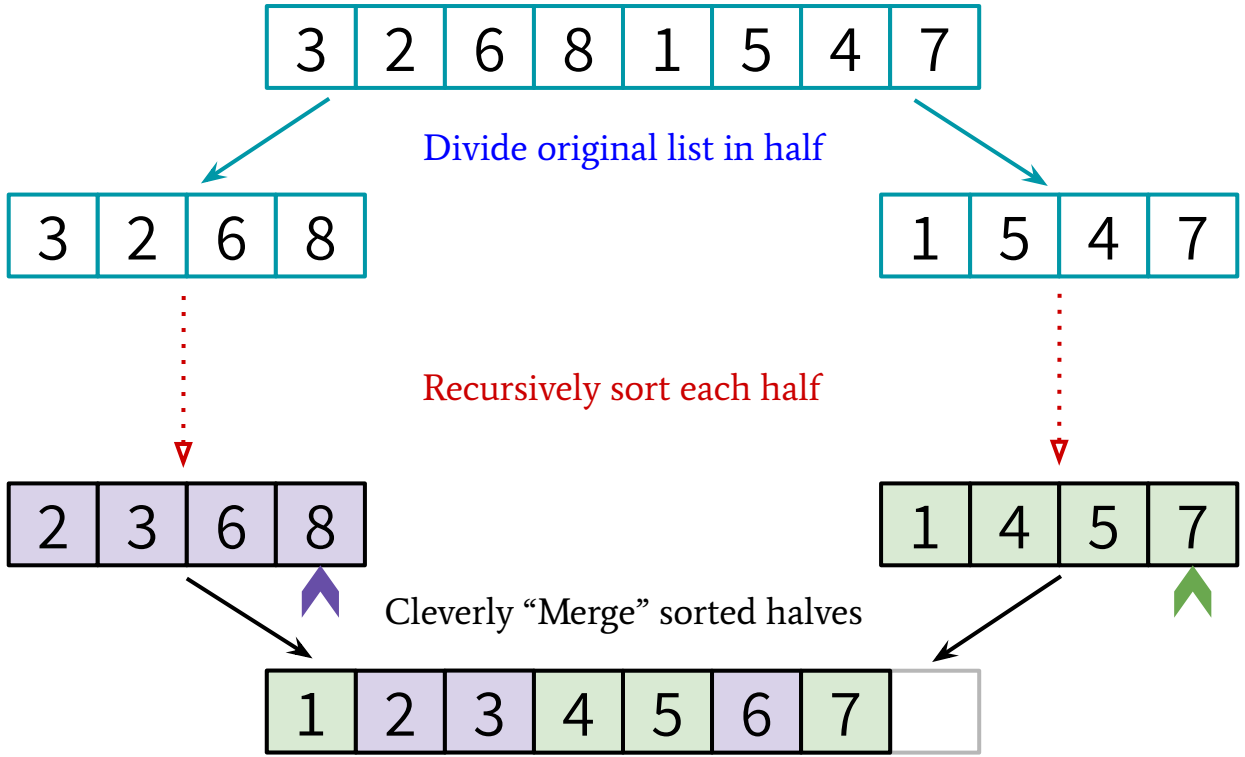
Merge Sort


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

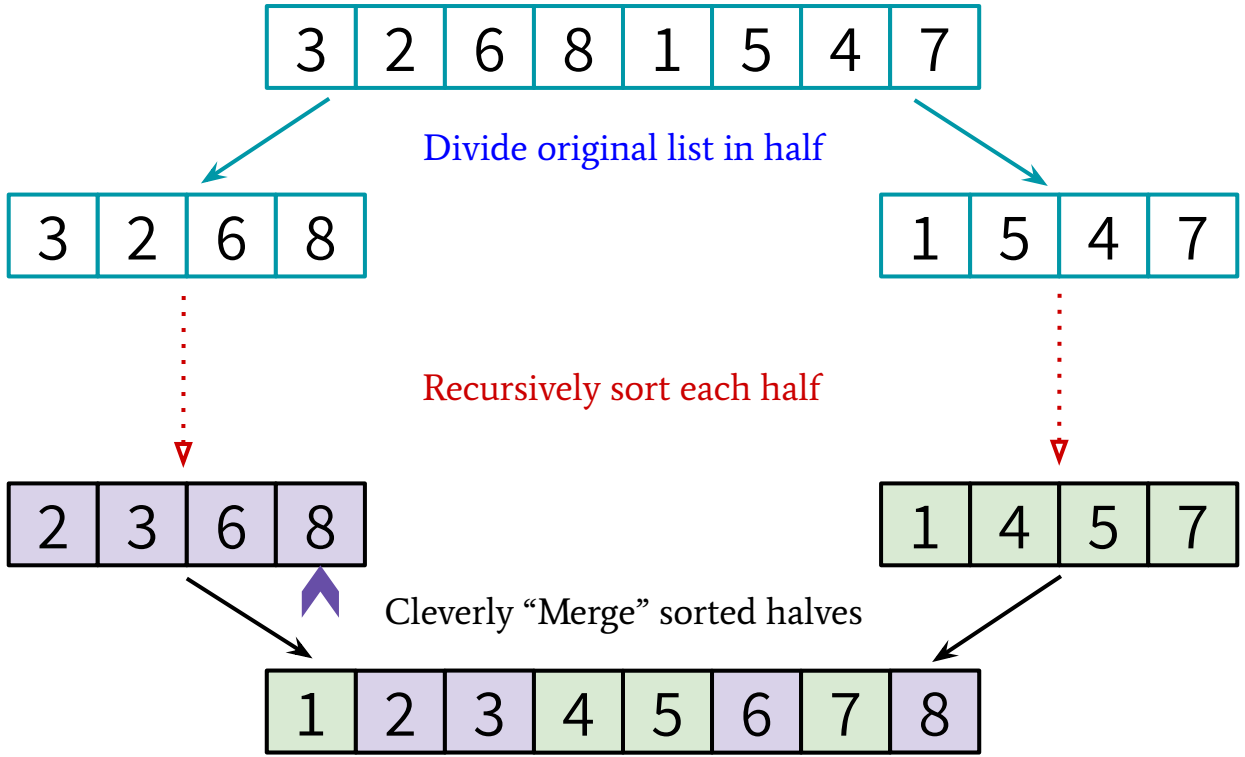
Merge Sort


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

Merge Sort


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

Merge Sort


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

Merge Sort: Pseudo Code

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

```
MERGESORT(A):
```

Merge Sort: Pseudo Code

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
```

Merge Sort: Pseudo Code

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])
```

Merge Sort: Pseudo Code

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])
```

Merge Sort: Pseudo Code

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)
```

Assume that n is a power of 2.

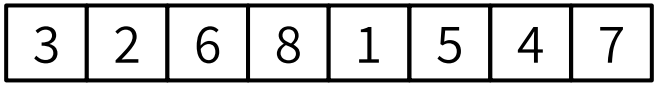
Merge Sort: Pseudo Code

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

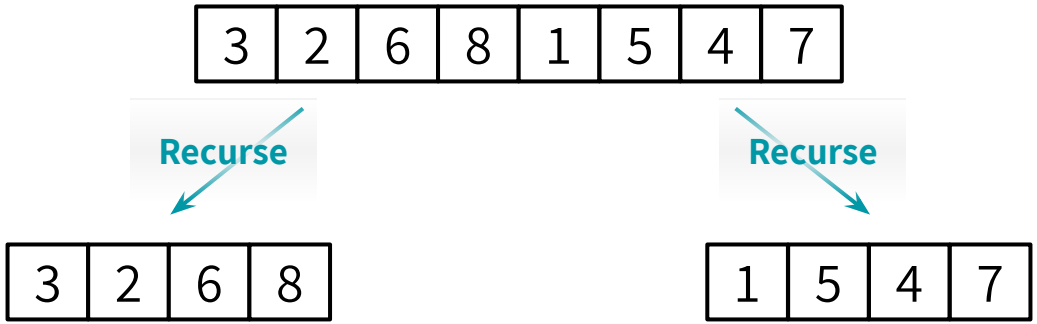
```
MERGESORT(A):
    n = len(A)
    if n <= 1:
        return A
    L = MERGESORT(A[0:n/2])
    R = MERGESORT(A[n/2:n])
    return MERGE(L,R)
```

```
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
```

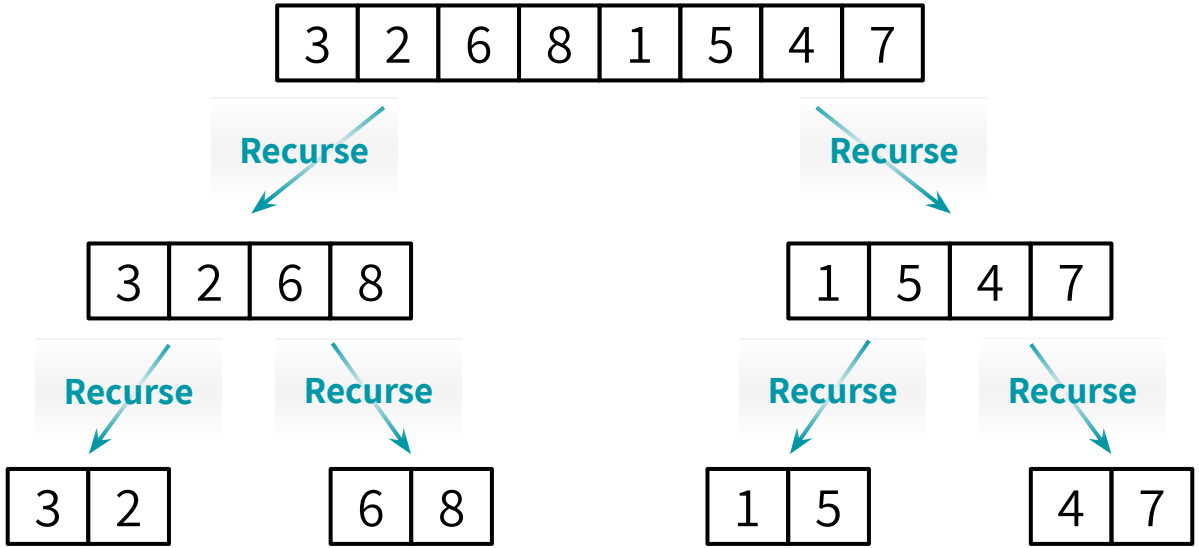
Recursive Calls



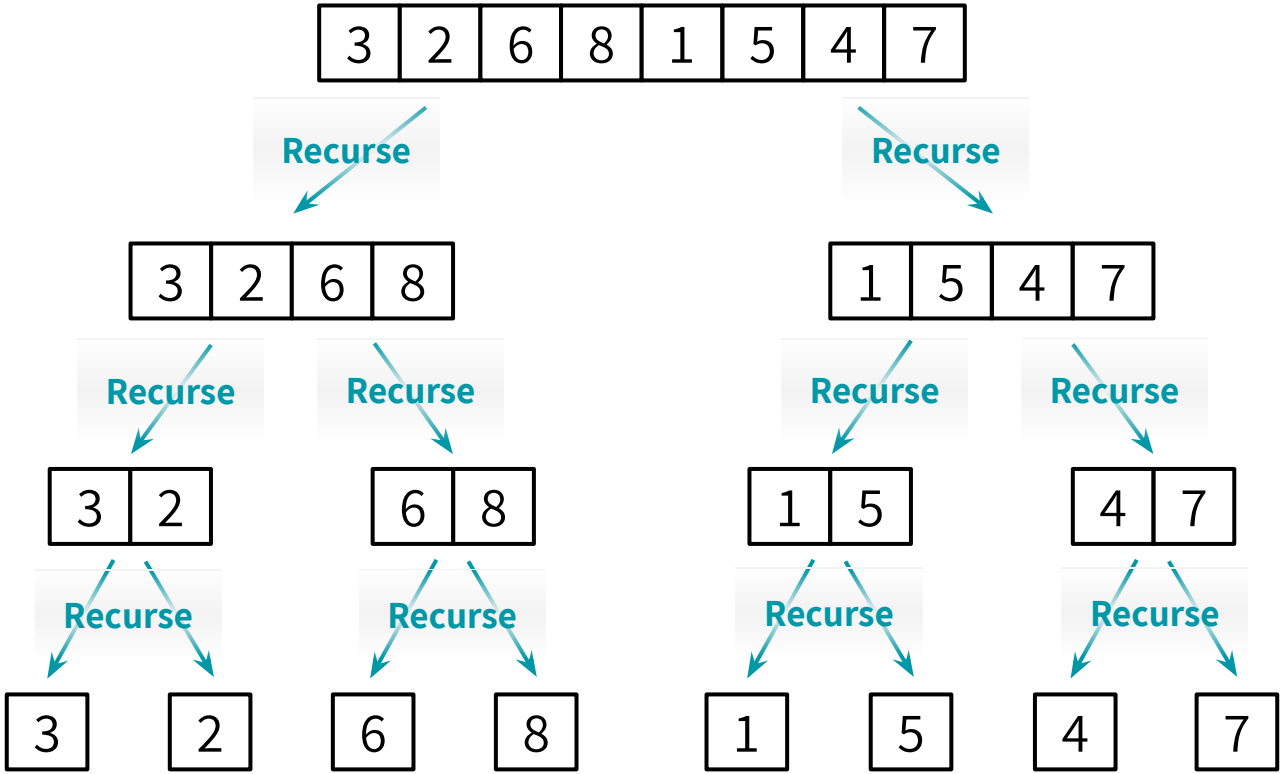
Recursive Calls



Recursive Calls

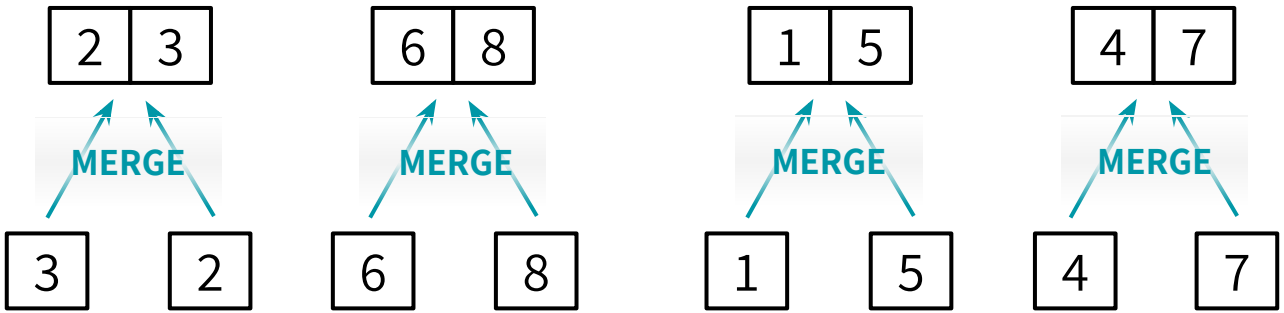


Recursive Calls

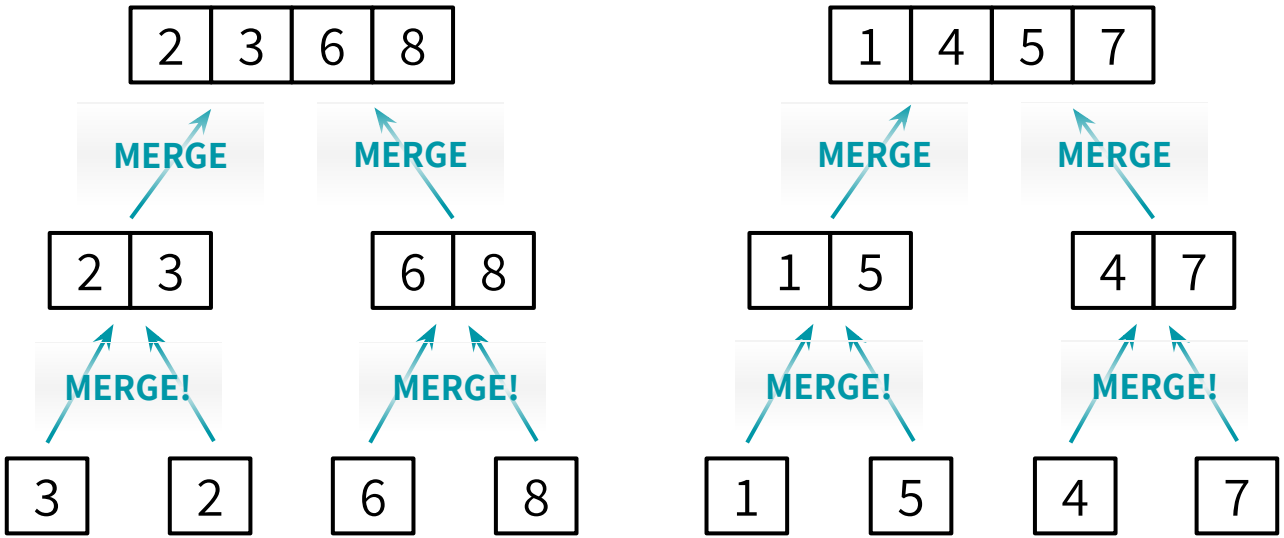


Merge Steps

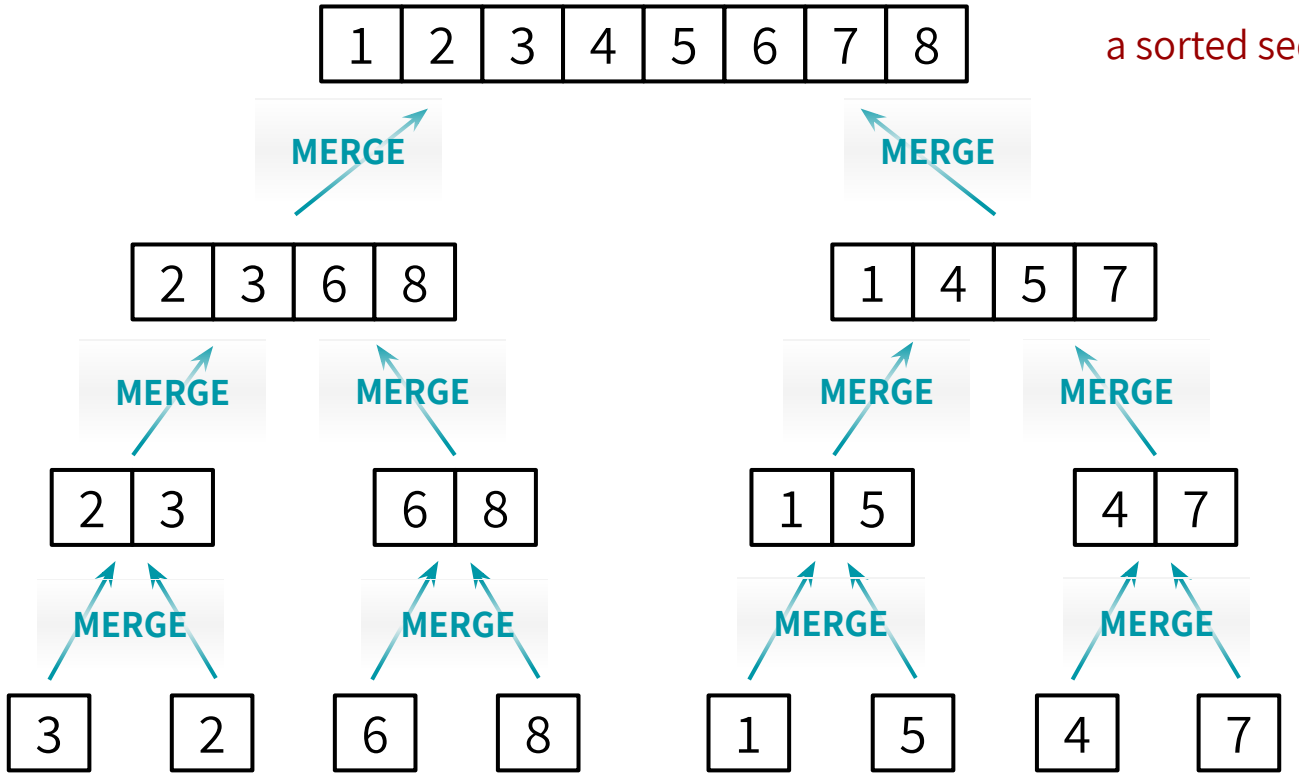
Merge Steps



Merge Steps


2 3 6 8
MERGE
MERGE
2 3
6 8
MERGE!
MERGE!
3
2
6
8
1 4 5 7
MERGE
MERGE
1 5
4 7
MERGE!
MERGE!
1
5
4
7

Merge Steps



a sorted sequence

Merge Sort: Does It Work?

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 the recursive calls work on arrays of smaller lengths, then the algorithm will correctly return a sorted array.

Merge Sort: Does It Work?

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 the recursive calls work on arrays of smaller lengths, then our algorithm will correctly return a sorted array.

Proof By Induction

Perform induction on the *length of input list*, rather than # of iterations

Merge Sort: 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 Inductive Hypothesis holds for i = 1: A 1-element array is always sorted.

Merge Sort: Induction Proof

Inductive Step

- 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.
- 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.

Since the two "child" recursive calls are executed on arrays of length k/2 (which is strictly less than k), 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.

Merge Sort: Induction Proof

Concluding step

By induction, we conclude that the Inductive Hypothesis (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.

Proving stuff

Iterative Algorithms

1. **Inductive hypothesis (IH)**: 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 $\Rightarrow$ prove k+1
4. **Conclusion**: IH holds for i = # total iterations

Recursive Algorithms

Proving stuff

Iterative Algorithms

1. **Inductive hypothesis (IH)**: 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

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

Merge Sort: 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: Merge Sort runs in time O(n log n)

Class Quiz: Prove the claim

Recall SelectionSort, BubbleSort had quadratic running times