CSE 332

JULY 17TH – SORTING

ASSORTED MINUTIAE

- Exams almost finished

ASSORTED MINUTIAE

- **Exams almost finished**
  - Scores will be out tomorrow

ASSORTED MINUTIAE

- **Exams almost finished**
  - Scores will be out tomorrow
  - Exams back in class on Wednesday

ASSORTED MINUTIAE

- **Exams almost finished**
  - Scores will be out tomorrow
  - Exams back in class on Wednesday
- **P2 checkpoint is also on Wednesday**

ASSORTED MINUTIAE

- **Exams almost finished**
  - Scores will be out tomorrow
  - Exams back in class on Wednesday
- **P2 checkpoint is also on Wednesday**
  - Make sure you’ve completed at least the ckpt1 tests on gitlab

SORTING

- **Problem statement:**

SORTING

- **Problem statement:**
  - Given some collection of **comparable** data, arrange them into an organized order

SORTING

- **Problem statement:**
  - Given some collection of **comparable** data, arrange them into an organized order
  - Important to note that you may be able to “organize” the same data different ways

SORTING

- **Why sort at all?**

SORTING

- **Why sort at all?**
  - Data pre-processing

SORTING

- **Why sort at all?**
  - Data pre-processing
  - If we do the work now, future operations may be faster

SORTING

- **Why sort at all?**
  - Data pre-processing
  - If we do the work now, future operations may be faster
  - Unsorted v. Sorted Array, e.g.

SORTING

- **Why sort at all?**
  - Data pre-processing
  - If we do the work now, future operations may be faster
  - Unsorted v. Sorted Array, e.g.
- **Why not just maintain sortedness as we add?**

SORTING

- **Why sort at all?**
  - Data pre-processing
  - If we do the work now, future operations may be faster
  - Unsorted v. Sorted Array, e.g.
- **Why not just maintain sortedness as we add?**
  - Most times, if we can, we should

SORTING

- **Why sort at all?**
  - Data pre-processing
  - If we do the work now, future operations may be faster
  - Unsorted v. Sorted Array, e.g.
- **Why not just maintain sortedness as we add?**
  - Most times, if we can, we should
  - Why would we not be able to?

SORTING

- Maintaining Sortedness v. Sorting

SORTING

- **Maintaining Sortedness v. Sorting**
  - Why **don’t** we maintain sortedness?

SORTING

- **Maintaining Sortedness v. Sorting**
  - Why **don’t** we maintain sortedness?
    - Data comes in batches

SORTING

- **Maintaining Sortedness v. Sorting**
  - Why **don’t** we maintain sortedness?
    - Data comes in batches
    - Multiple “sorted” orders

SORTING

- **Maintaining Sortedness v. Sorting**
  - Why **don’t** we maintain sortedness?
    - Data comes in batches
    - Multiple “sorted” orders
    - Costly to maintain!

SORTING

- **Maintaining Sortedness v. Sorting**
  - Why **don’t** we maintain sortedness?
    - Data comes in batches
    - Multiple “sorted” orders
    - Costly to maintain!
- **We need to be sure that the effort is worth the work**

SORTING

- **Maintaining Sortedness v. Sorting**
  - Why **don’t** we maintain sortedness?
    - Data comes in batches
    - Multiple “sorted” orders
    - Costly to maintain!
- **We need to be sure that the effort is worth the work**
  - No free lunch!

SORTING

- **Maintaining Sortedness v. Sorting**
  - Why **don’t** we maintain sortedness?
    - Data comes in batches
    - Multiple “sorted” orders
    - Costly to maintain!
- **We need to be sure that the effort is worth the work**
  - No free lunch!
- **What does that even mean?**

BOGO SORT

- Consider the following sorting algorithm

BOGO SORT

- **Consider the following sorting algorithm**
  - Shuffle the list into a random order

BOGO SORT

- **Consider the following sorting algorithm**
  - Shuffle the list into a random order
  - Check if the list is sorted

BOGO SORT

- **Consider the following sorting algorithm**
  - Shuffle the list into a random order
  - Check if the list is sorted,
  - if so return the list

BOGO SORT

- **Consider the following sorting algorithm**
    - Shuffle the list into a random order
    - Check if the list is sorted,
    - if so return the list
    - if not, try again

BOGO SORT

- **Consider the following sorting algorithm**
  - Shuffle the list into a random order
  - Check if the list is sorted,
  - if so return the list
  - if not, try again

BOGO SORT

- **Consider the following sorting algorithm**
  - Shuffle the list into a random order
  - Check if the list is sorted,
  - if so return the list
  - if not, try again
- **What is the problem here?**

BOGO SORT

- **Consider the following sorting algorithm**
  - Shuffle the list into a random order
  - Check if the list is sorted,
  - if so return the list
  - if not, try again
- **What is the problem here?**
  - Runtime!

BOGO SORT

- **Consider the following sorting algorithm**
  - Shuffle the list into a random order
  - Check if the list is sorted,
  - if so return the list
  - if not, try again
- **What is the problem here?**
  - Runtime! Average **O(n!)!**

BOGO SORT

- **Consider the following sorting algorithm**
  - Shuffle the list into a random order
  - Check if the list is sorted,
  - if so return the list
  - if not, try again
- **What is the problem here?**
  - Runtime! Average **O(n!)!**
  - Why is this so bad?

BOGO SORT

- **Consider the following sorting algorithm**
  - Shuffle the list into a random order
  - Check if the list is sorted,
  - if so return the list
  - if not, try again
- **What is the problem here?**
  - Runtime! Average **O(n!)!**
  - Why is this so bad?
- **The computer isn't thinking, it's just guess-and-checking**

SORTING

- Guess-and-check

SORTING

- **Guess-and-check**
  - Not a bad strategy when nothing else is obvious

SORTING

- **Guess-and-check**
  - Not a bad strategy when nothing else is obvious
    - Breaking RSA

SORTING

- **Guess-and-check**
  - Not a bad strategy when nothing else is obvious
    - Breaking RSA
    - Greedy-first algorithms

SORTING

- **Guess-and-check**
  - Not a bad strategy when nothing else is obvious
    - Breaking RSA
    - Greedy-first algorithms
  - If you don’t have a lot of time, or if the payoff is big, or if the chance of success is high, then it might be a good strategy

SORTING

- **Guess-and-check**
  - Not a bad strategy when nothing else is obvious
    - Breaking RSA
    - Greedy-first algorithms
  - If you don’t have a lot of time, or if the payoff is big, or if the chance of success is high, then it might be a good strategy
  - Random/Approximized algs

SORTING

- Why not guess-and-check for sorting?

SORTING

- **Why not guess-and-check for sorting?**
  - Not taking advantage of the biggest constraint of the problem

SORTING

- **Why not guess-and-check for sorting?**
  - Not taking advantage of the biggest constraint of the problem
  - Items must be comparable!

SORTING

- **Why not guess-and-check for sorting?**
  - Not taking advantage of the biggest constraint of the problem
  - Items must be comparable!
  - You should be comparing things!
  - Looking at two items next to each other tells a lot about where they belong in the list, there’s no reason not to use this information.

SORTING

- Types of sorts

SORTING

- **Types of sorts**
    - Comparison sorts

SORTING

- **Types of sorts**
  - Comparison sorts
    - Bubble sort

SORTING

- **Types of sorts**
  - Comparison sorts
    - Bubble sort
    - Insertion sort

SORTING

- **Types of sorts**
  - Comparison sorts
    - Bubble sort
    - Insertion sort
    - Selection sort

SORTING

- **Types of sorts**
  - Comparison sorts
    - Bubble sort
    - Insertion sort
    - Selection sort
    - Heap sort

SORTING

- **Types of sorts**
  - Comparison sorts
    - Bubble sort
    - Insertion sort
    - Selection sort
    - Heap sort
  - “Other” sorts
    - Bucket sort – will talk about later

SORTING

- **Types of sorts**
  - Comparison sorts
    - Bubble sort
    - Insertion sort
    - Selection sort
    - Heap sort
  - “Other” sorts
    - Bucket sort – will talk about later
    - Bogo sort

DEFINITION: COMPARISON SORT

A computational problem with the following input and output

Input:

An array `A` of length *n* comparable elements

Output:

The same array `A`, containing the same elements where:

for any `i` and `j` where $0 \leq i < j < n$

then $A[i] \leq A[j]$

MORE REASONS TO SORT

General technique in computing:

Preprocess data to make subsequent operations faster

Example: Sort the data so that you can

- Find the `k`th largest in constant time for any `k`
- Perform binary search to find elements in logarithmic time

Whether the performance of the preprocessing matters depends on

- How often the data will change (and how much it will change)
- How much data there is

MORE DEFINITIONS

In-Place Sort:

A sorting algorithm is in-place if it requires only O(1) extra space to sort the array.

- Usually modifies input array
- Can be useful: lets us minimize memory

Stable Sort:

A sorting algorithm is stable if any equal items remain in the same relative order before and after the sort.

- Items that ’compare’ the same might not be exact duplicates
- Might want to sort on some, but not all attributes of an item
- Can be useful to sort on one attribute first, then another one

STABLE SORT EXAMPLE

Input:

```
[(8, "fox"), (9, "dog"), (4, "wolf"), (8, "cow")]
```

Compare function: compare pairs by number only

Output (stable sort):

```
[(4, "wolf"), (8, "fox"), (8, "cow"), (9, "dog")]
```

Output (unstable sort):

```
[(4, "wolf"), (8, "cow"), (8, "fox"), (9, "dog")]
```

SORTING: THE BIG PICTURE

Simple algorithms: $O(n^2)$

- Insertion sort
- Selection sort
- Shell sort
- . . .

Fancier algorithms: $O(n \log n)$

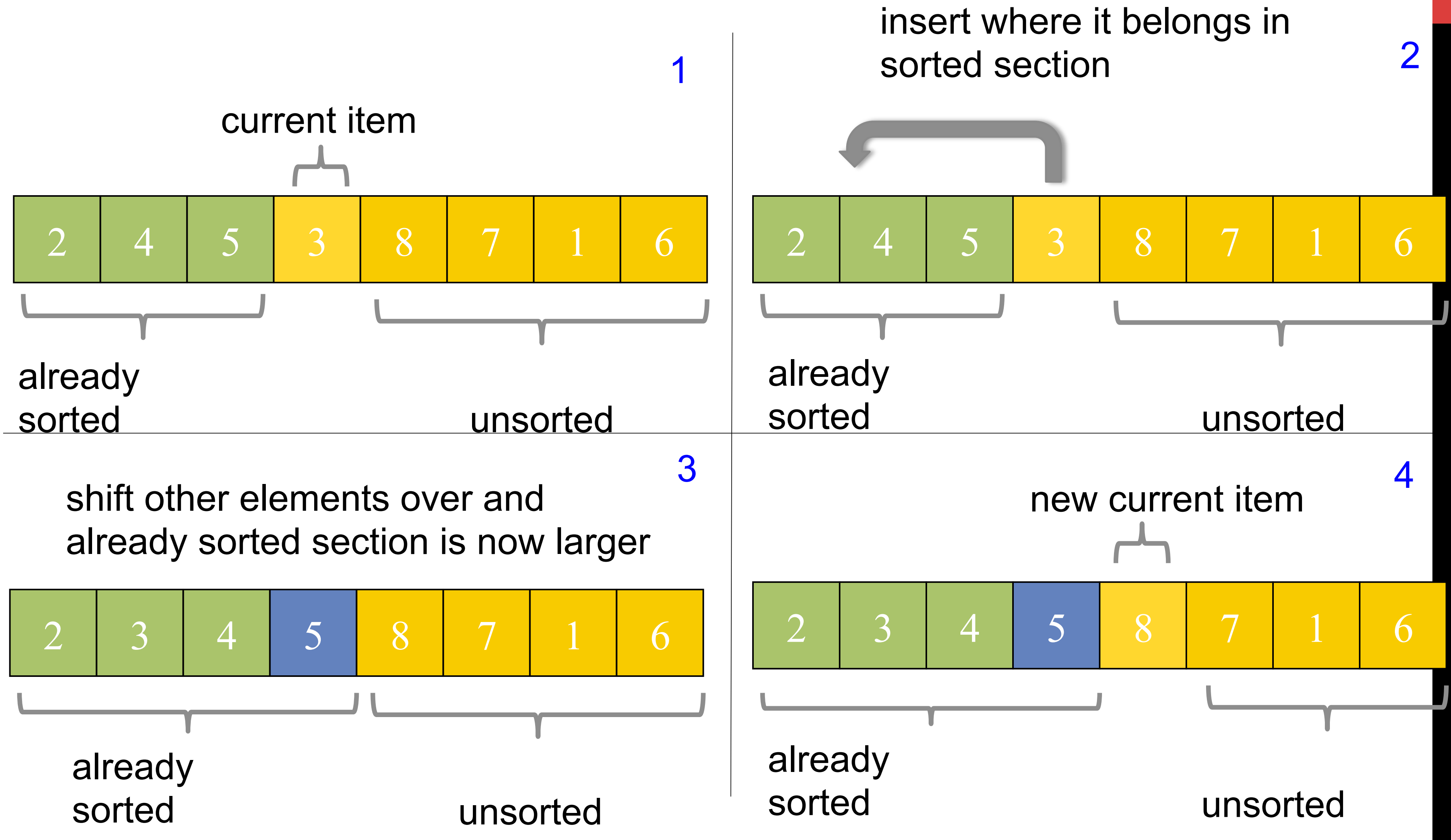
- Heap sort
- Merge sort
- Quick sort (avg)
- …

Comparison lower bound: $\Omega(n \log n)$

Specialized algorithms: $O(n)$

- Bucket sort
- Radix sort

INSERTION SORT



1

current item

2	4	5	3	8	7	1	6

already sorted

unsorted

insert where it belongs in sorted section

2

2	4	5	3	8	7	1	6

already sorted

unsorted

3

shift other elements over and already sorted section is now larger

2	3	4	5	8	7	1	6

already sorted

unsorted

4

new current item

2	3	4	5	8	7	1	6

already sorted

unsorted

INSERTION SORT

Idea: At step `k`, put the `k`th element in the correct position among the first `k` elements

```
for (int i = 0; i < n; i++) {
       // Find index to insert into
       int newIndex = findPlace(i);
       // Insert and shift nodes over
       shift(newIndex, i);
}
```

What can we say about the list at loop i? first `i` elements are sorted

(not necessarily lowest in the list)

Runtime?

INSERTION SORT

Idea: At step `k`, put the `k`th element in the correct position among the first `k` elements

```
for (int i = 0; i < n; i++) {
       // Find index to insert into
       int newIndex = findPlace(i);
       // Insert and shift nodes over
       shift(newIndex, i);
}
```

What can we say about the list at loop i? first `i` elements are sorted

(not necessarily lowest in the list)

Runtime? Best case: O(n), Worst case: $O(n^2)$ Why?

INSERTION SORT

Idea: At step `k`, put the `k`th element in the correct position among the first `k` elements

```
for (int i = 0; i < n; i++) {
       // Find index to insert into
       int newIndex = findPlace(i);
       // Insert and shift nodes over
       shift(newIndex, i);
}
```

What can we say about the list at loop i? first `i` elements are sorted

(not necessarily lowest in the list)

Runtime? Best case: $O(n)$, Worst case: $O(n^2)$ Why?

Stable? **In-place?**

INSERTION SORT

Idea: At step `k`, put the `k`th element in the correct position among the first `k` elements

```
for (int i = 0; i < n; i++) {
        // Find index to insert into
        int newIndex = findPlace(i);
        // Insert and shift nodes over
        shift(newIndex, i);
}
```

What can we say about the list at loop i? first `i` elements are sorted

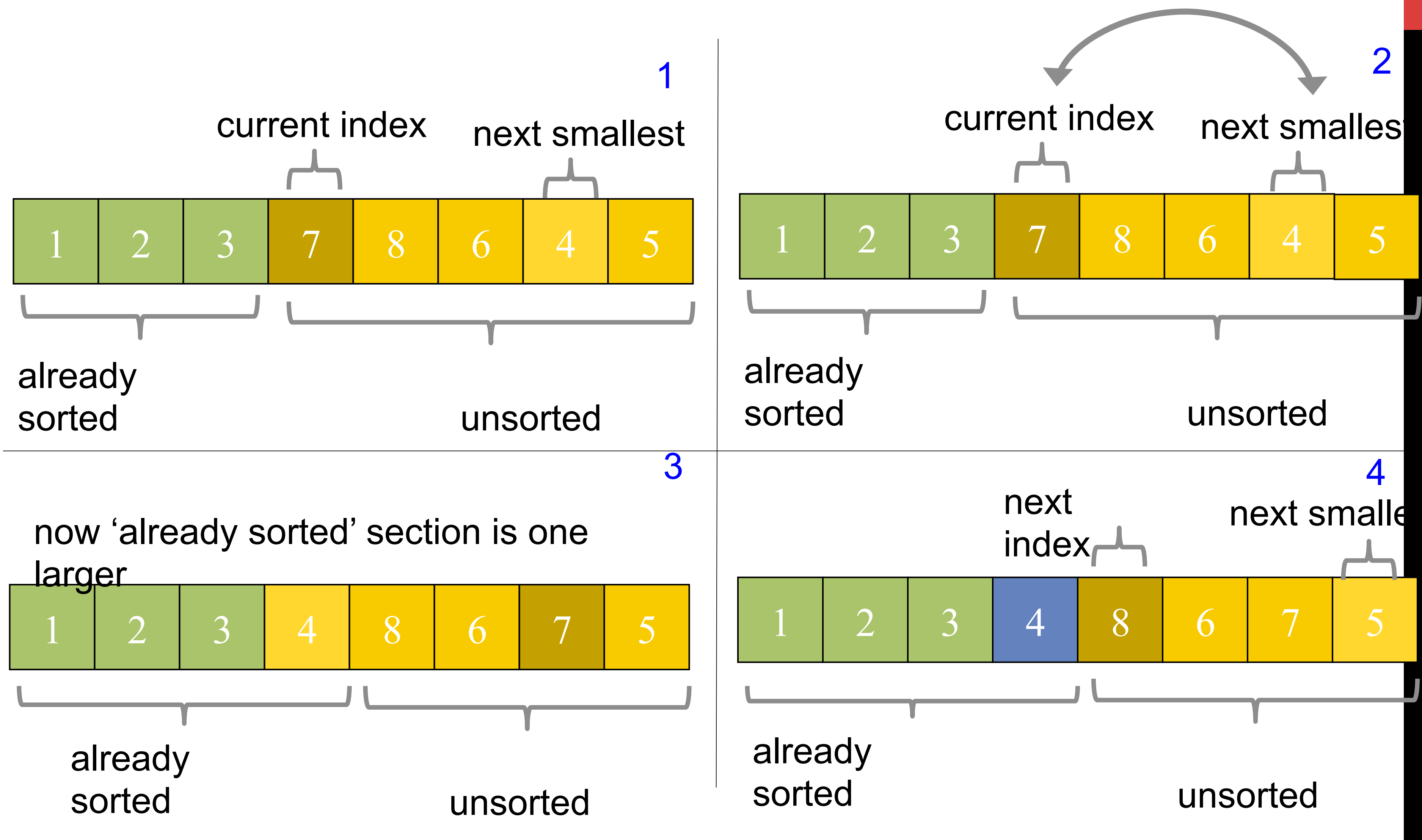
(not necessarily lowest in the list)

Runtime? Best case: O(n), Worst case: O(n^2) Why?

Stable? Usually **In-place?** Yes

SELECTION SORT

swap



1

current index

next smallest

1	2	3	7	8	6	4	5

already sorted

unsorted

2

current index

next smalles

1	2	3	7	8	6	4	5

already sorted

unsorted

3

now ‘already sorted’ section is one larger

1	2	3	4	8	6	7	5

already sorted

unsorted

4

next index

next smalle

1	2	3	4	8	6	7	5

already sorted

unsorted

SELECTION SORT

- Can be interrupted (don’t need to sort the whole array to get the first element)
- Doesn’t need to mutate the original array (if the array has some other sorted order)
- Stable sort

INSERTION SORT VS. SELECTION SORT

Have the same worst-case and average-case asymptotic complexity

- Insertion-sort has better best-case complexity; preferable when input is “mostly sorted”

Useful for small arrays or for mostly sorted input

SORTING

- Important definitions

SORTING

- **Important definitions**
  - In-place:

SORTING

- **Important definitions**
    - In-place: Requires only O(1) extra memory

SORTING

- **Important definitions**
  - In-place: Requires only O(1) extra memory
    - **usually means the array is mutated**

SORTING

- **Important definitions**
  - In-place: Requires only O(1) extra memory
    - **usually means the array is mutated**
  - Stable: For any two elements have the same comparative value, then after the sort, which ever came first will stay first

SORTING

- **Important definitions**
  - In-place: Requires only O(1) extra memory
    - **usually means the array is mutated**
  - Stable: For any two elements have the same comparative value, then after the sort, which ever came first will stay first
    - Sorting by first name and then last name will give you **last then first** with a stable sort.

SORTING

- **Important definitions**
  - In-place: Requires only O(1) extra memory
    - **usually means the array is mutated**
  - Stable: For any two elements have the same comparative value, then after the sort, which ever came first will stay first
    - Sorting by first name and then last name will give you **last then first** with a stable sort.
    - The most recent sort will always be the primary

SORTING

- **Important definitions**
  - Interruptable:

SORTING

- **Important definitions**
  - Interruptable: the algorithm can run only until the first *k* elements are in sorted order

SORTING

- **Important definitions**
  - Interruptable: the algorithm can run only until the first *k* elements are in sorted order
  - Comparison sort: utilizes comparisons between elements to produce the final sorted order.

SORTING

- **Important definitions**
  - Interruptable: the algorithm can run only until the first *k* elements are in sorted order
  - Comparison sort: utilizes comparisons between elements to produce the final sorted order.
    - Bogo sort is not a comparison sort

SORTING

- **Important definitions**
  - Interruptable: the algorithm can run only until the first *k* elements are in sorted order
  - Comparison sort: utilizes comparisons between elements to produce the final sorted order.
    - Bogo sort is not a comparison sort
    - Comparison sorts are $\Omega(n \log n)$, they cannot do better than this

SORTING

- **What other sorting techniques can we consider?**

SORTING

- **What other sorting techniques can we consider?**
  - We know O(n log n) is possible. How do we do it?

SORTING

- **What other sorting techniques can we consider?**
  - We know O(n log n) is possible. How do we do it?
  - Heap sort works on principles we already know.

SORTING

- **What other sorting techniques can we consider?**
  - We know O(n log n) is possible. How do we do it?
  - Heap sort works on principles we already know.
    - Building a heap from an array takes O(n) time

SORTING

- **What other sorting techniques can we consider?**
  - We know O(n log n) is possible. How do we do it?
  - Heap sort works on principles we already know.
    - Building a heap from an array takes O(n) time
    - Removing the smallest element from the array takes O(log n)

SORTING

- **What other sorting techniques can we consider?**
  - We know O(n log n) is possible. How do we do it?
  - Heap sort works on principles we already know.
    - Building a heap from an array takes O(n) time
    - Removing the smallest element from the array takes O(log n)
    - There are n elements.

SORTING

- **What other sorting techniques can we consider?**
  - We know O(n log n) is possible. How do we do it?
  - Heap sort works on principles we already know.
    - Building a heap from an array takes O(n) time
    - Removing the smallest element from the array takes O(log n)
    - There are n elements.
    - N + N*log N = O(N log N)

SORTING

- **What other sorting techniques can we consider?**
  - We know O(n log n) is possible. How do we do it?
  - Heap sort works on principles we already know.
    - Building a heap from an array takes O(n) time
    - Removing the smallest element from the array takes O(log n)
    - There are n elements.
    - N + N*log N = O(N log N)
    - Using Floyd’s method does not improve the asymptotic runtime for heap sort, but it is an improvement.

HEAP SORT

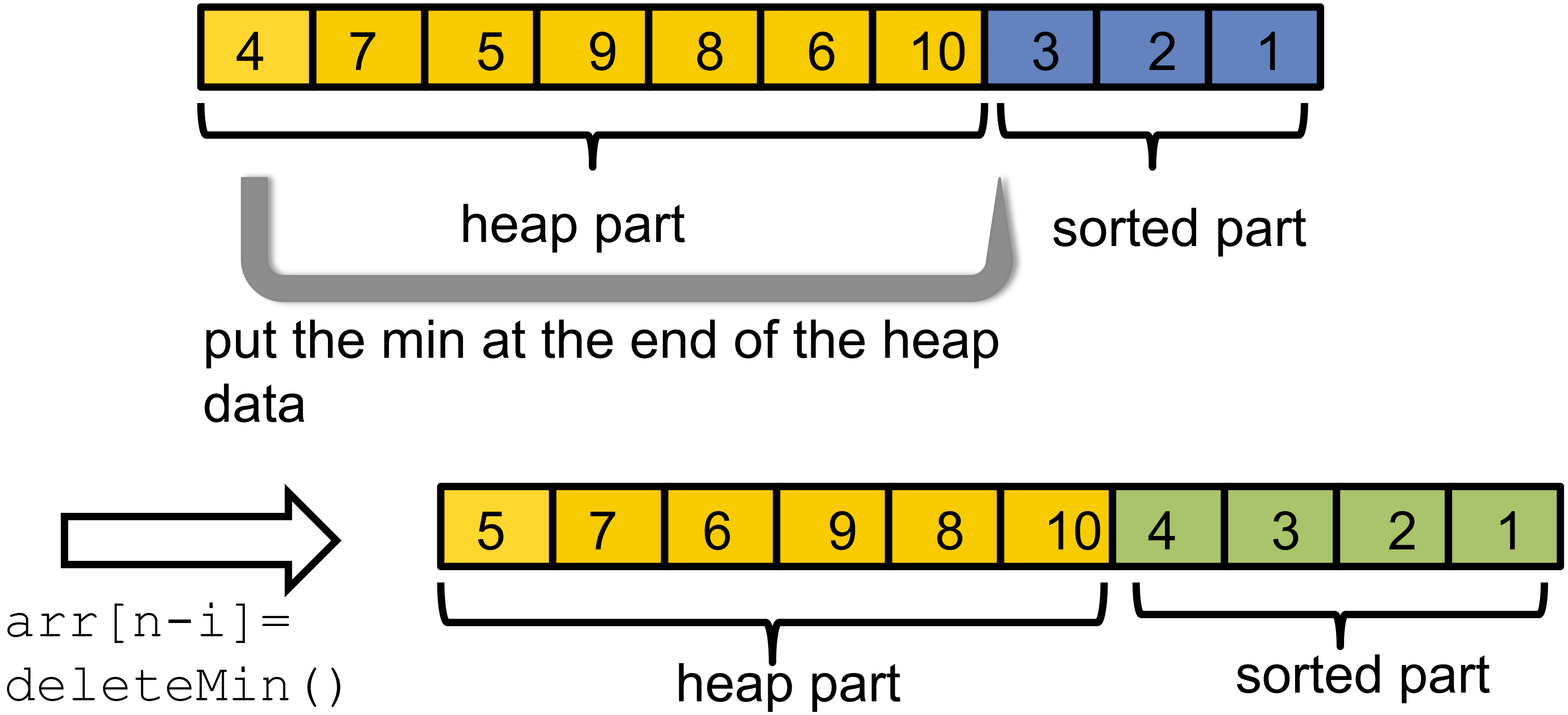
- How do we actually implement this sort?
- Can we do it in place?

HEAP SORT

- How do we actually implement this sort?
- Can we do it in place?

IN-PLACE HEAP SORT

- Treat the initial array as a heap (via **`buildHeap`**)
- When you delete the **`i`**$^{\text{th}}$ element, put it at **`arr[n-i]`**
    - That array location isn't needed for the heap anymore!



4	7	5	9	8	6	10	3	2	1

heap part

sorted part

put the min at the end of the heap data

`arr[n-i]= deleteMin()`

5	7	6	9	8	10	4	3	2	1

heap part

sorted part

HEAP SORT

- How do we actually implement this sort?
- Can we do it in place?
- Is this sort stable?

HEAP SORT

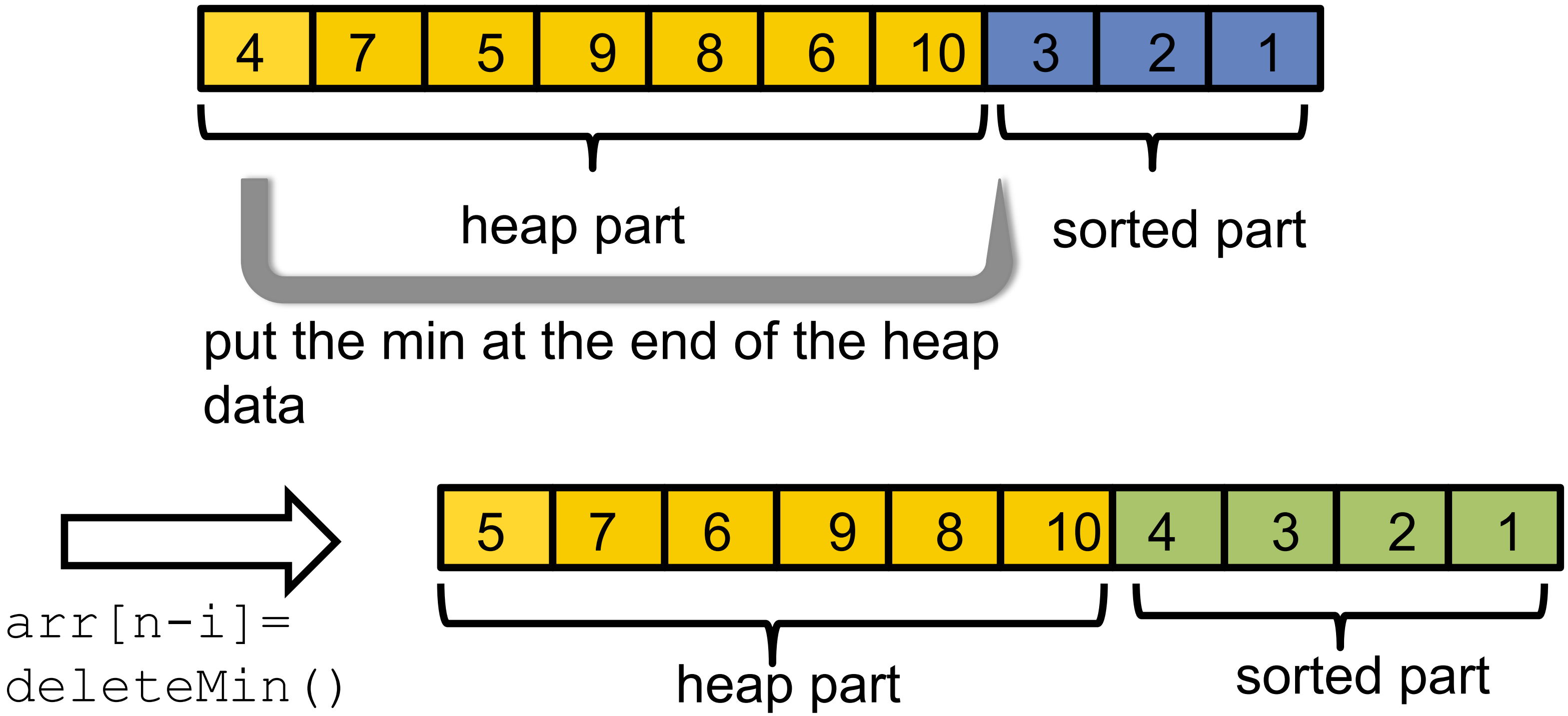
- **How do we actually implement this sort?**
- **Can we do it in place?**
- **Is this sort stable?**
  - No. Recall that heaps do not preserve FIFO property

HEAP SORT

- **How do we actually implement this sort?**
- **Can we do it in place?**
- **Is this sort stable?**
  - No. Recall that heaps do not preserve FIFO property
  - If it needed to be stable, we would have to modify the priority to indicate its place in the array, so that each element has a unique priority.

IN-PLACE HEAP SORT

What is undesirable about this method?



4	7	5	9	8	6	10	3	2	1

heap part

sorted part

put the min at the end of the heap data

`arr[n-i]=`
`deleteMin()`

5	7	6	9	8	10	4	3	2	1

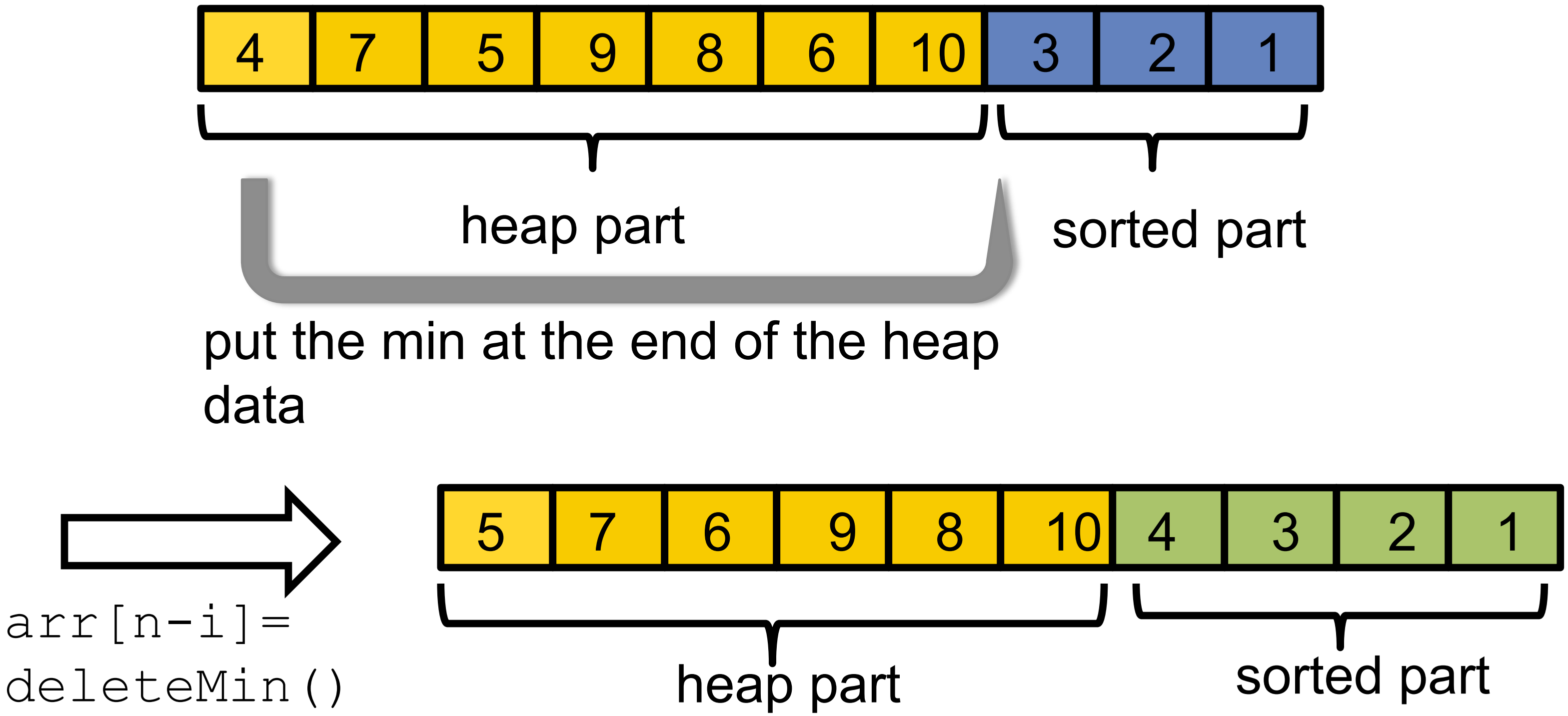
heap part

sorted part

IN-PLACE HEAP SORT

What is undesirable about this method?

You must reverse the array at the end.



4	7	5	9	8	6	10	3	2	1

put the min at the end of the heap data

`arr[n-i]=`
`deleteMin()`

5	7	6	9	8	10	4	3	2	1

HEAP SORT

- Can implement with a max-heap, then the sorted portion of the array fills in from the back and doesn’t need to be reversed at the end.

“AVL SORT”? “HASH SORT”?

AVL Tree: sure, we can also use an AVL tree to:

“AVL SORT”? “HASH SORT”?

AVL Tree: sure, we can also use an AVL tree to:

- **`insert`** each element: total time $O(n \texttt{ log } n)$
- Repeatedly **`deleteMin`**: total time $O(n \texttt{ log } n)$
    - Better: in-order traversal $O(n)$, but still $O(n \texttt{ log } n)$ overall
- But this cannot be done in-place and has worse constant factors than heap sort

"AVL SORT"? "HASH SORT"?

AVL Tree: sure, we can also use an AVL tree to:

- `insert` each element: total time *O*(*n* `log` *n*)
- Repeatedly `deleteMin`: total time *O*(*n* `log` *n*)
  - Better: in-order traversal *O*(*n*), but still *O*(*n* `log` *n*) overall
- But this cannot be done in-place and has worse constant factors than heap sort

Hash Structure: don't even think about trying to sort with a hash table!

“AVL SORT”? “HASH SORT”?

AVL Tree: sure, we can also use an AVL tree to:

- `insert` each element: total time *O*(*n* `log` *n*)
- Repeatedly `deleteMin`: total time *O*(*n* `log` *n*)
    - Better: in-order traversal *O*(*n*), but still *O*(*n* `log` *n*) overall
- But this cannot be done in-place and has worse constant factors than heap sort

Hash Structure: don’t even think about trying to sort with a hash table!

- Finding min item in a hashtable is *O*(n), so this would be a slower, more complicated selection sort

DIVIDE AND CONQUER

Divide-and-conquer is a useful technique for solving many kinds of problems (not just sorting). It consists of the following steps:

1. Divide your work up into smaller pieces (recursively)
2. Conquer the individual pieces (as base cases)
3. Combine the results together (recursively)

```
algorithm(input) {
    if (small enough) {
        CONQUER, solve, and return input
    } else {
        DIVIDE input into multiple pieces
        RECURSE on each piece
        COMBINE and return results
    }
}
```

DIVIDE-AND-CONQUER SORTING

Two great sorting methods are fundamentally divide-and-conquer

Mergesort:

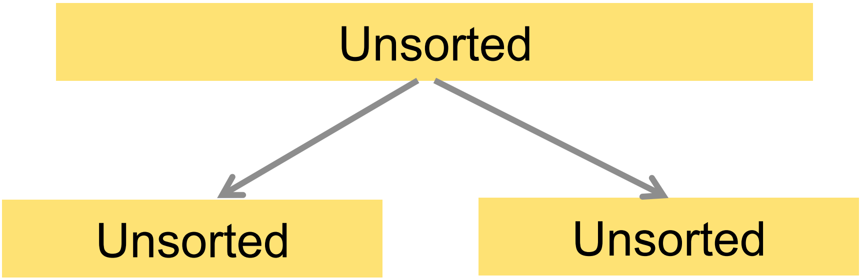
Sort the left half of the elements (recursively)
Sort the right half of the elements (recursively)
Merge the two sorted halves into a sorted whole

Quicksort:

Pick a “pivot” element
Divide elements into less-than pivot and greater-than pivot
Sort the two divisions (recursively on each)
Answer is: sorted-less-than....pivot....sorted-greater-than

MERGE SORT

Divide: Split array roughly into half

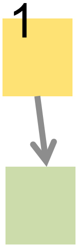


Unsorted

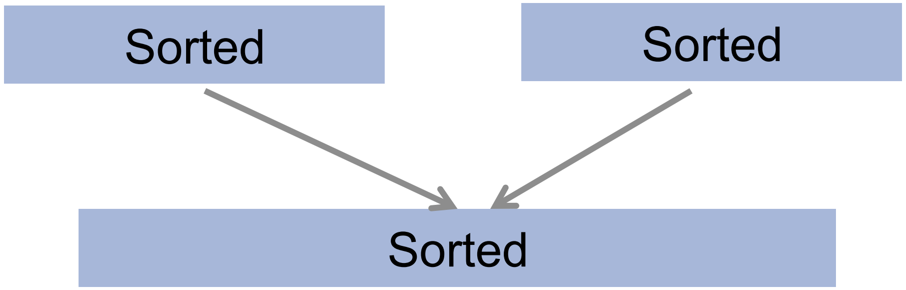
Unsorted

Unsorted

Conquer: Return array when length ≤ 1



Combine: Combine two sorted arrays using merge



Sorted

Sorted

Sorted

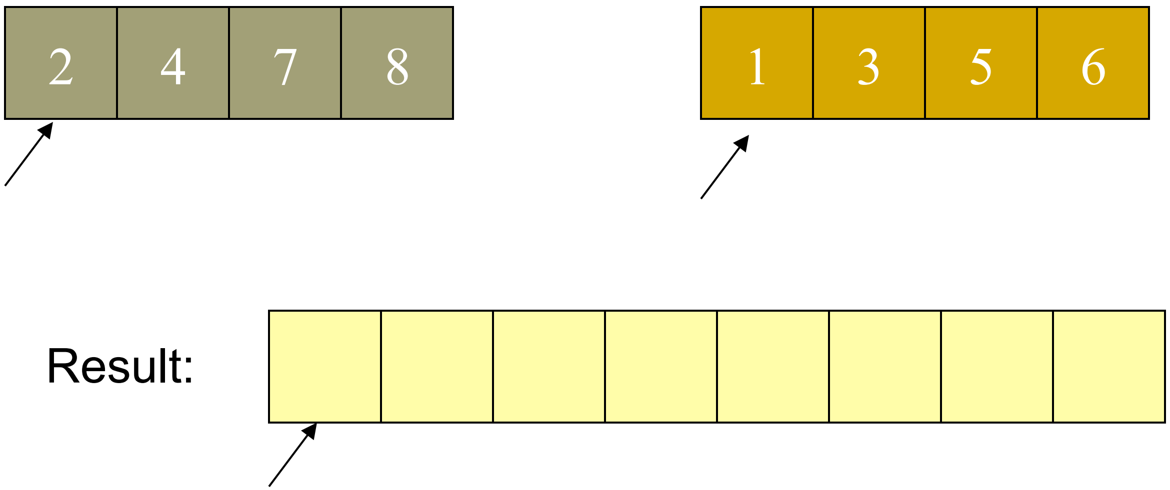
MERGE SORT: PSEUDOCODE

Core idea: split array in half, sort each half, merge back together. If the array has size 0 or 1, just return it unchanged

```
mergesort(input) {
    if (input.length < 2) {
        return input;
    } else {
        smallerHalf = sort(input[0, ..., mid]);
        largerHalf = sort(input[mid + 1, ...]);
        return merge(smallerHalf, largerHalf);
    }
}
```

MERGE EXAMPLE

Merge operation: Use 3 pointers and 1 more array



First half after sort:

2	4	7	8

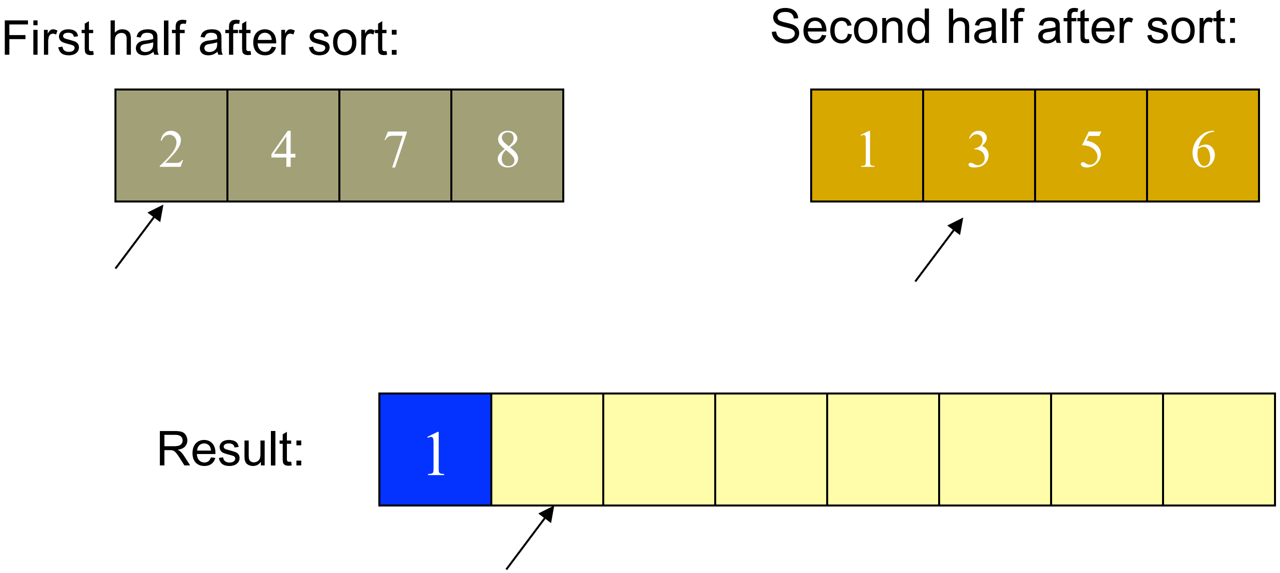
Second half after sort:

1	3	5	6

Result:

MERGE EXAMPLE

Merge operation: Use 3 pointers and 1 more array



First half after sort:

2	4	7	8

Second half after sort:

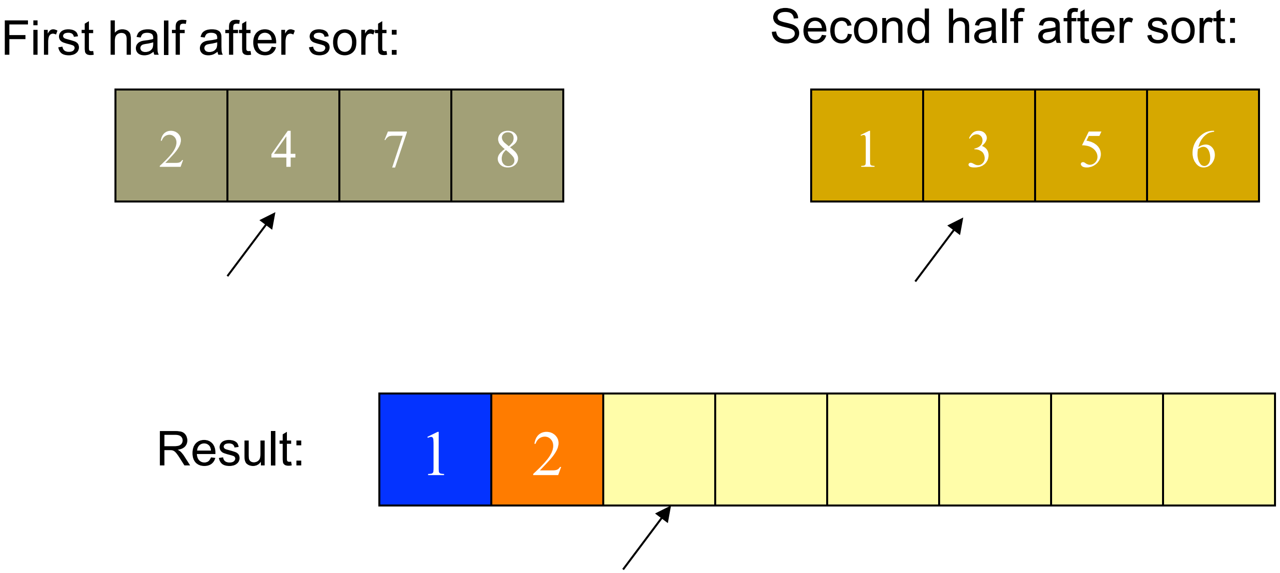
1	3	5	6

Result:

1							

MERGE EXAMPLE

Merge operation: Use 3 pointers and 1 more array



First half after sort:

2	4	7	8

Second half after sort:

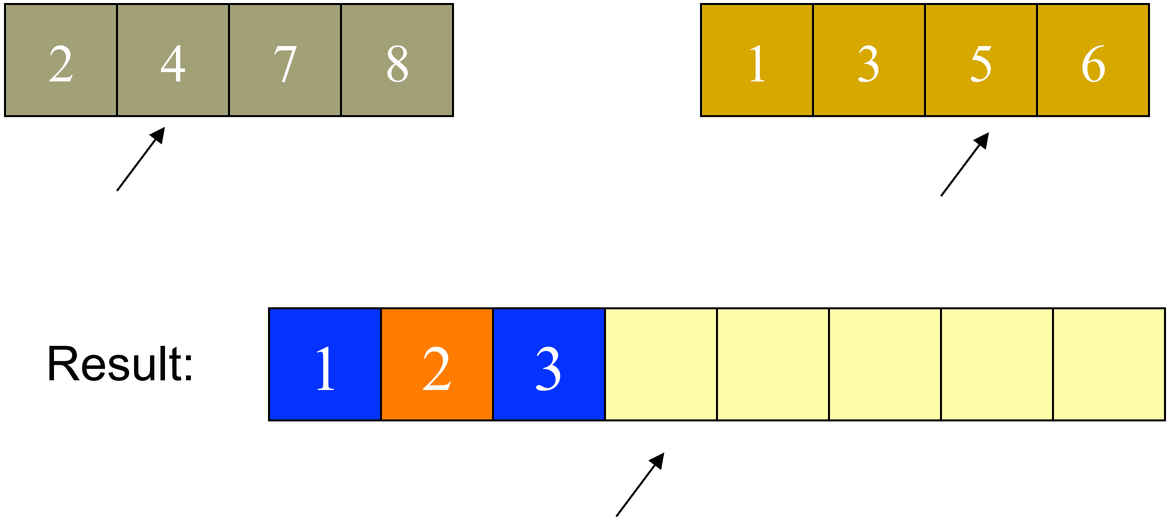
1	3	5	6

Result:

1	2						

MERGE EXAMPLE

Merge operation: Use 3 pointers and 1 more array



First half after sort:

2	4	7	8

Second half after sort:

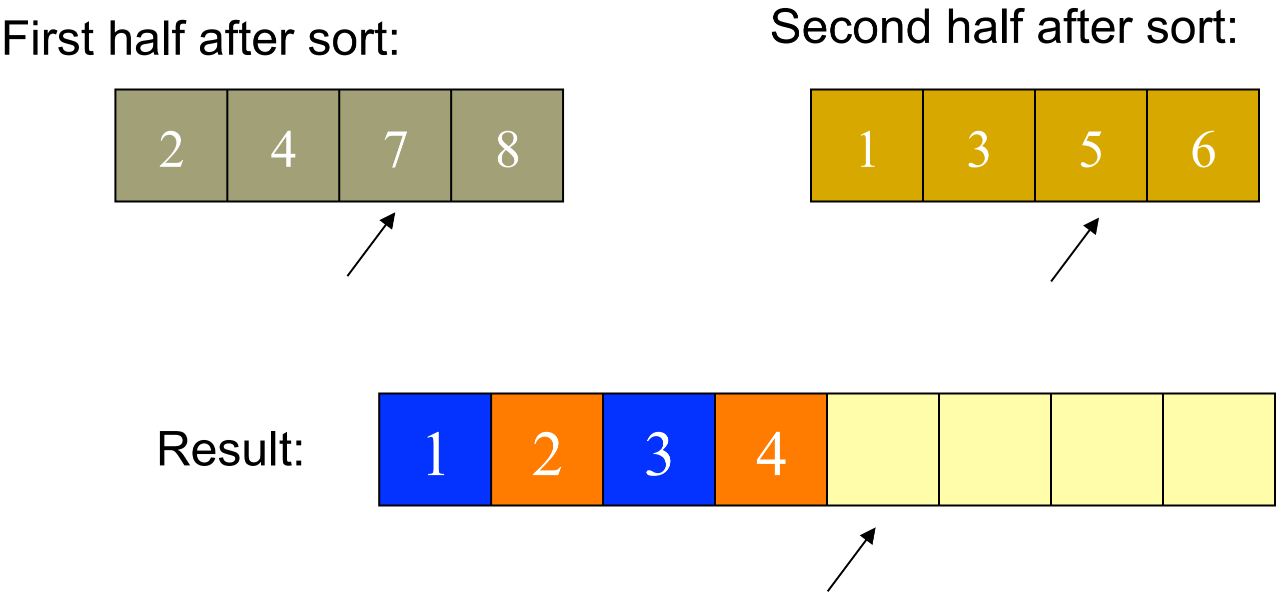
1	3	5	6

Result:

1	2	3					

MERGE EXAMPLE

Merge operation: Use 3 pointers and 1 more array



First half after sort:

2	4	7	8

Second half after sort:

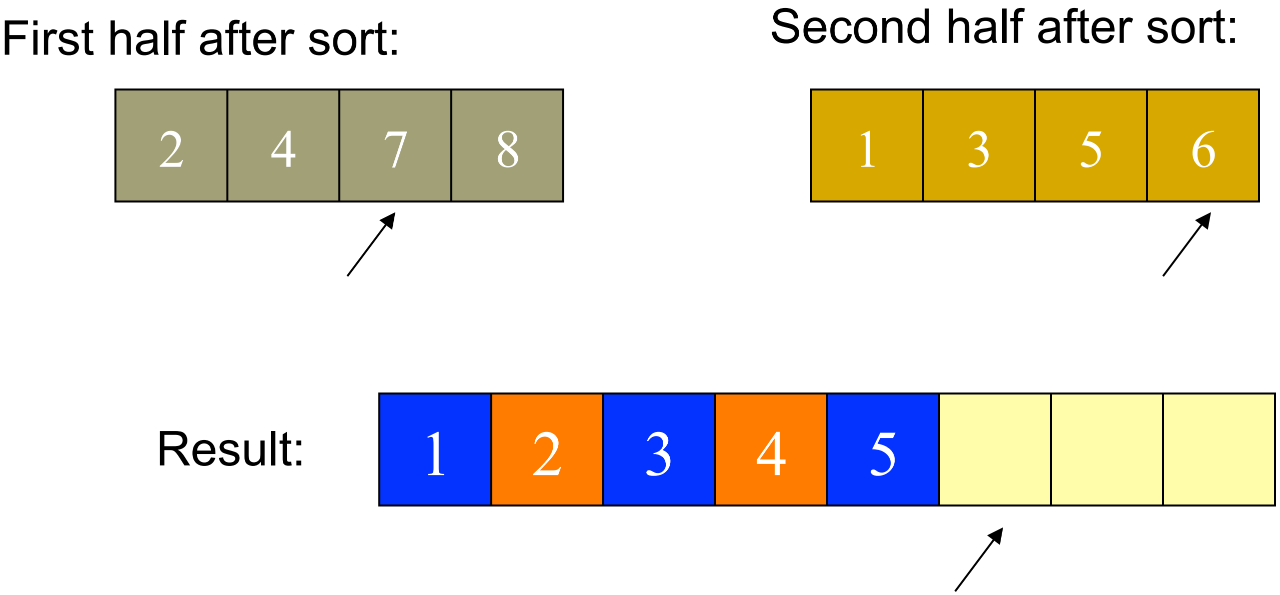
1	3	5	6

Result:

1	2	3	4				

MERGE EXAMPLE

Merge operation: Use 3 pointers and 1 more array



First half after sort:

2	4	7	8

Second half after sort:

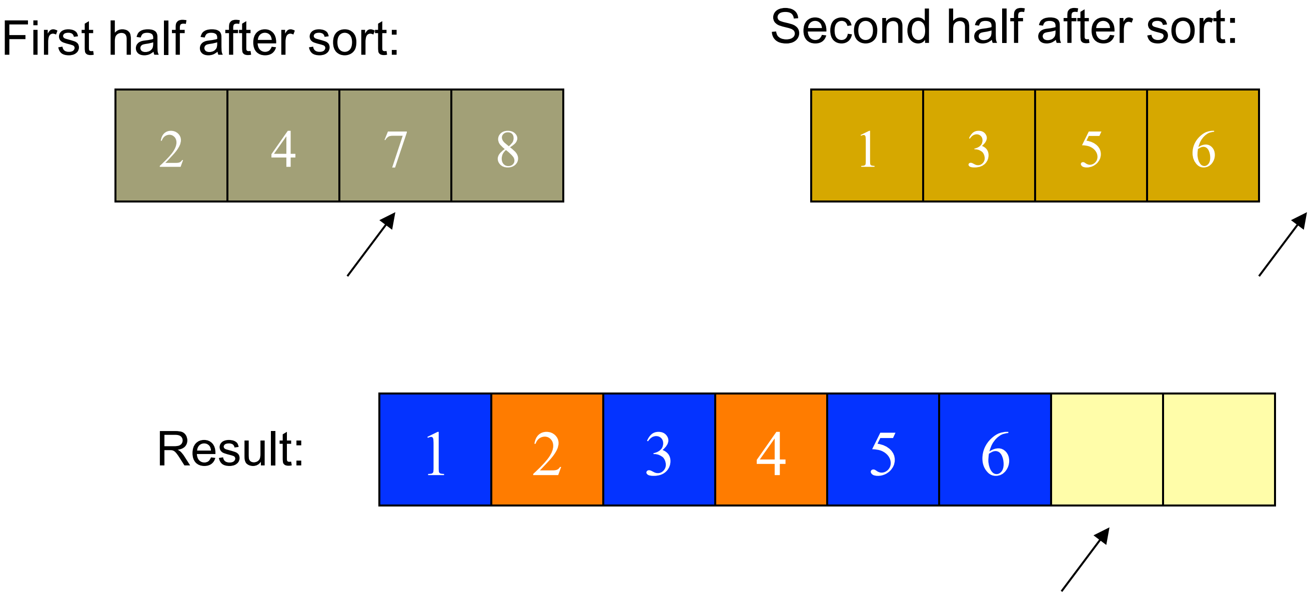
1	3	5	6

Result:

1	2	3	4	5			

MERGE EXAMPLE

Merge operation: Use 3 pointers and 1 more array



First half after sort:

2	4	7	8

Second half after sort:

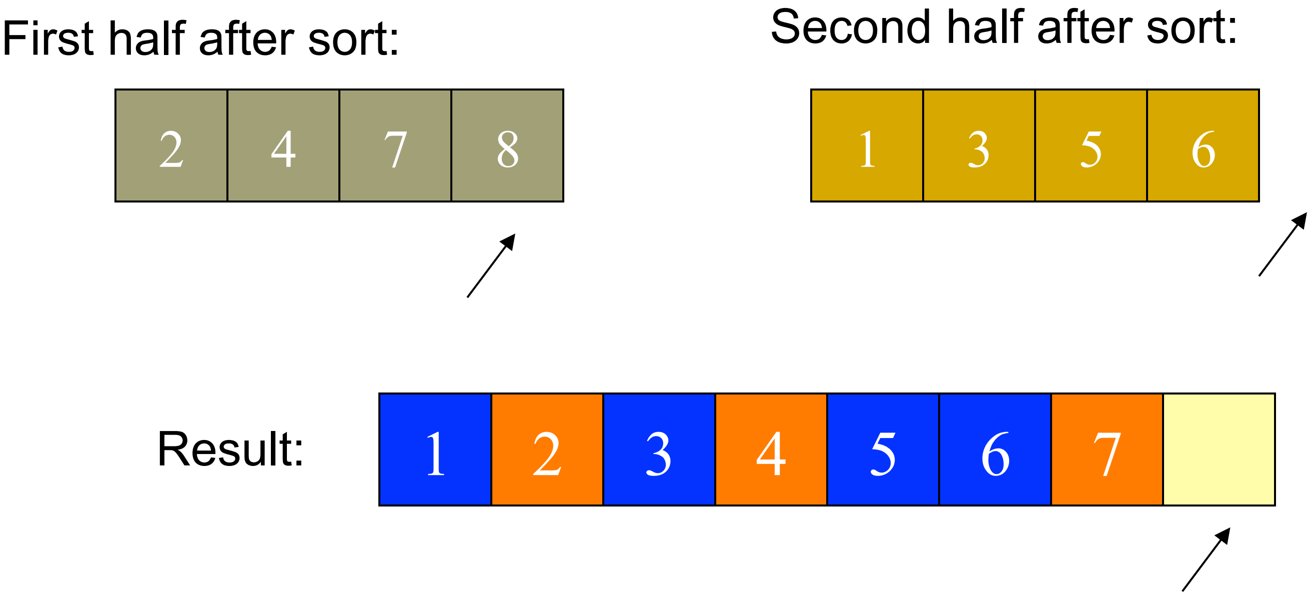
1	3	5	6

Result:

1	2	3	4	5	6		

MERGE EXAMPLE

Merge operation: Use 3 pointers and 1 more array



First half after sort:

2	4	7	8

Second half after sort:

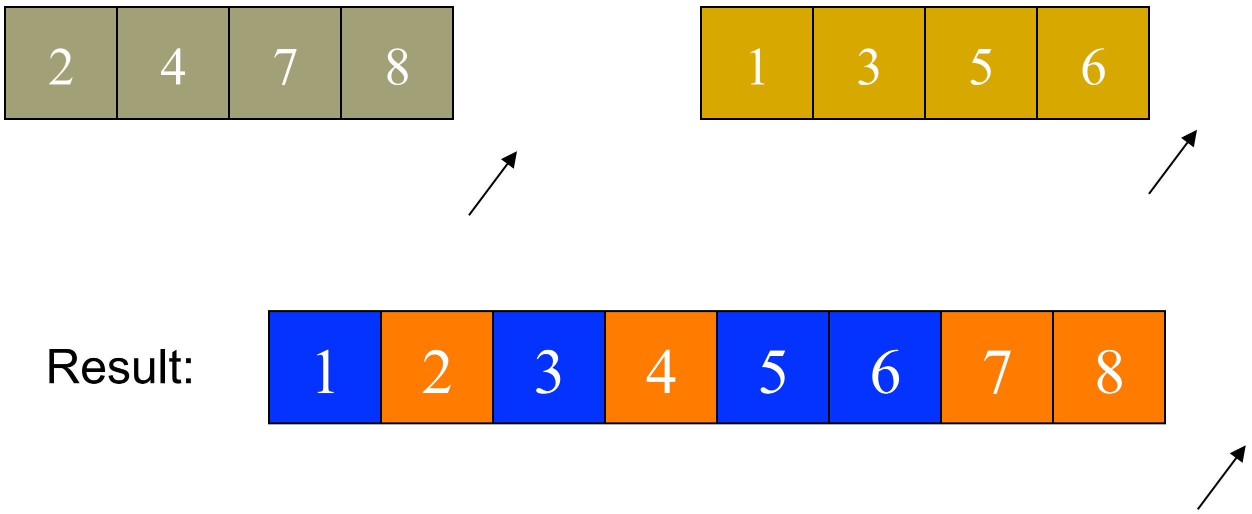
1	3	5	6

Result:

1	2	3	4	5	6	7	

MERGE EXAMPLE

Merge operation: Use 3 pointers and 1 more array



First half after sort:

2	4	7	8

Second half after sort:

1	3	5	6

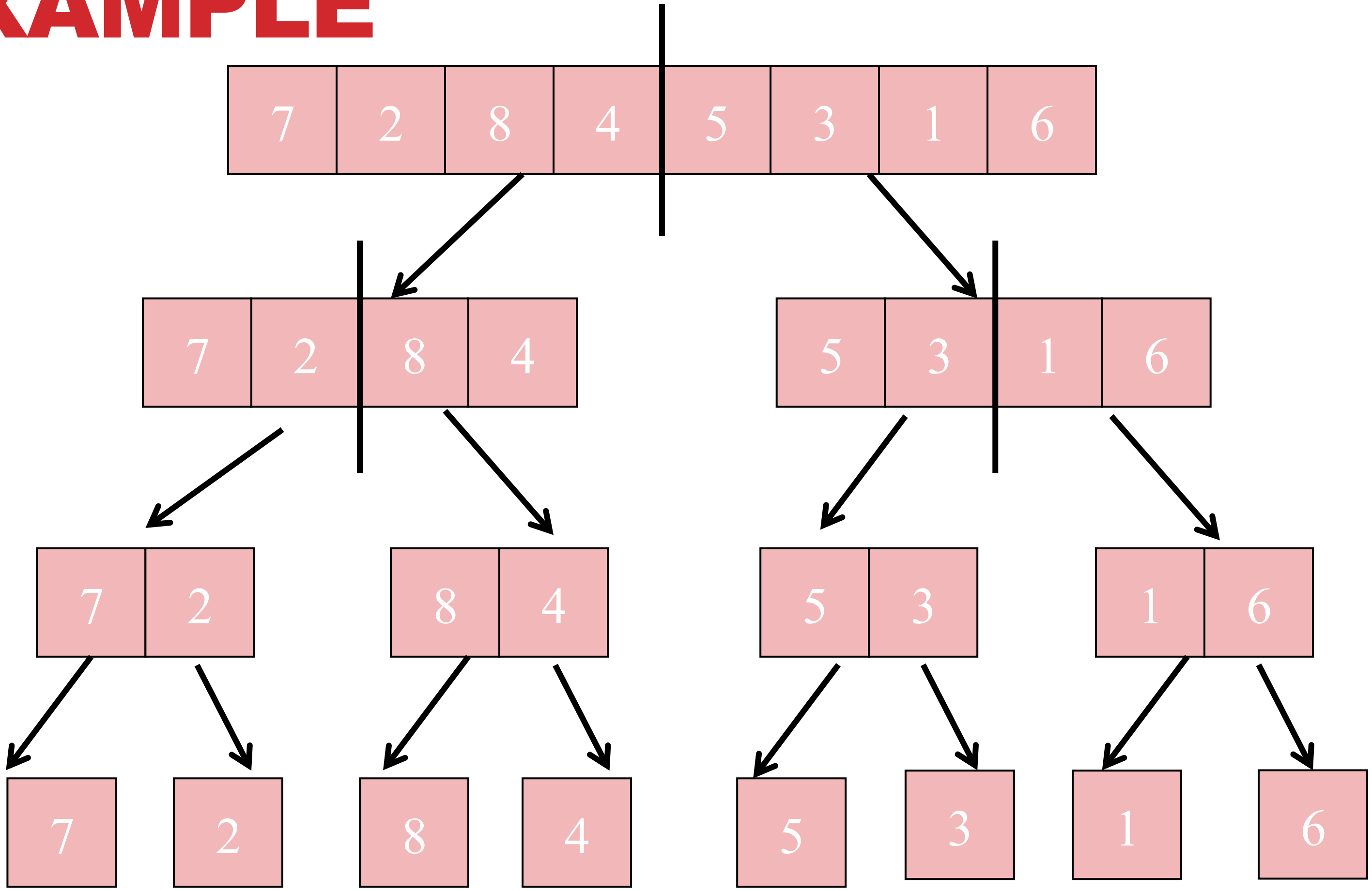
Result:

1	2	3	4	5	6	7	8

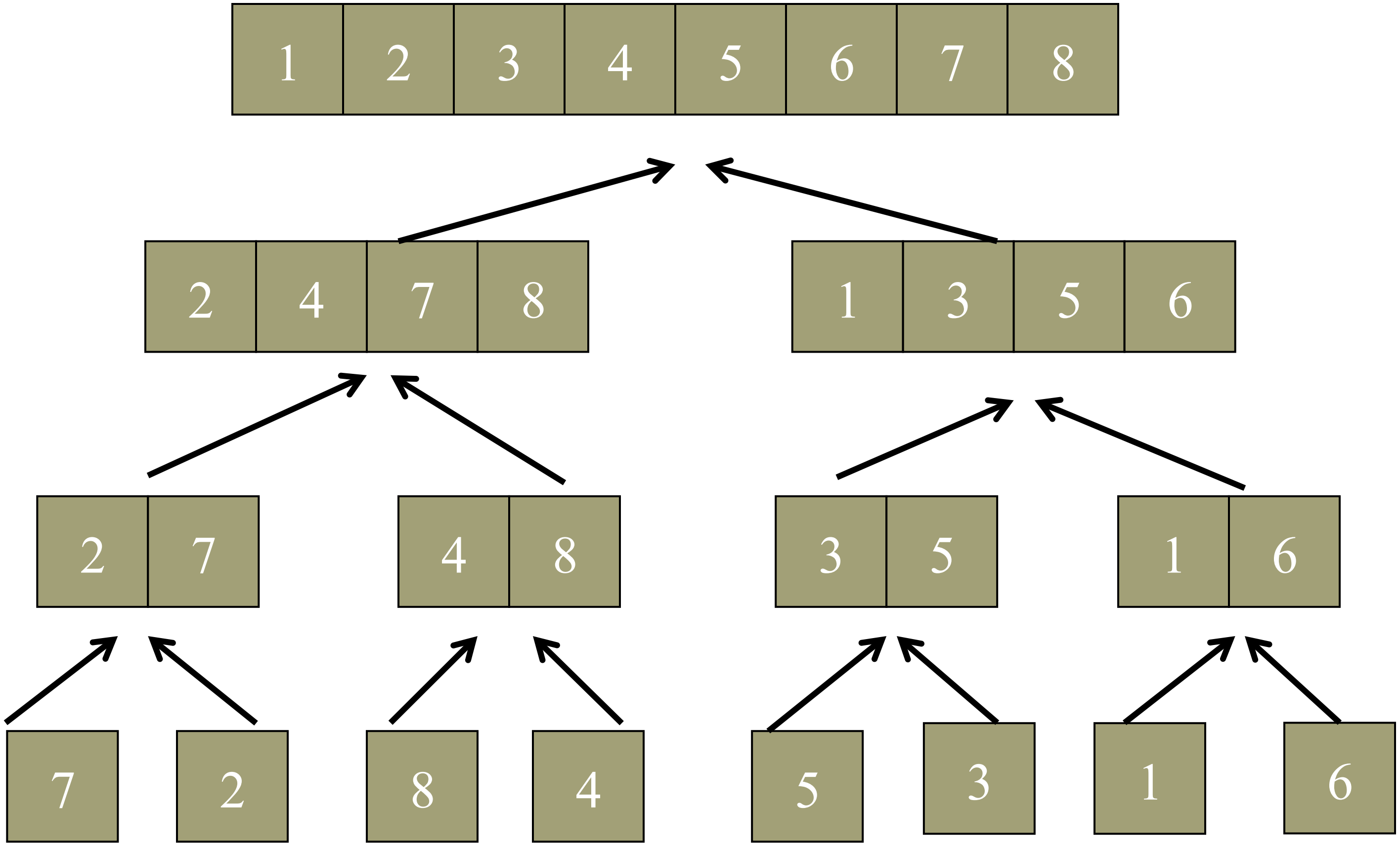
After Merge: copy result into original unsorted array.

Or alternate merging between two size n arrays.

MERGE SORT EXAMPLE



MERGE SORT EXAMPLE

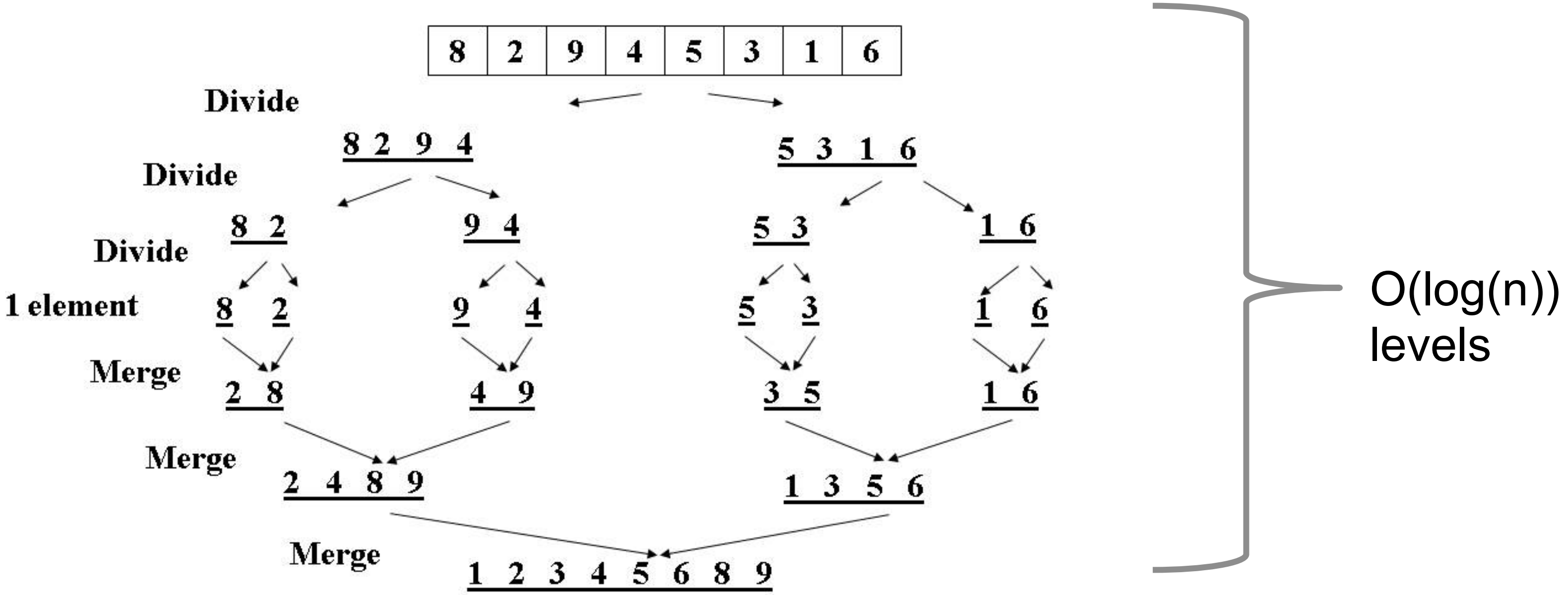


MERGE SORT ANALYSIS

Runtime:

- subdivide the array in half each time: O(log(n)) recursive calls
- merge is an O(n) traversal at each level

So, the best and worst case runtime is the same: O(n log(n))



MERGE SORT ANALYSIS

Stable?

Yes! If we implement the merge function correctly, merge sort will be stable.

In-place?

No. Unless you want to give yourself a headache. Merge must construct a new array to contain the output, so merge sort is not in-place.

We're constantly copying and creating new arrays at each level...

One Solution: (less of a headache than actually implementing in-place) create a single auxiliary array and swap between

it and the original on each level.