2023-04-28

Upcoming Schedule

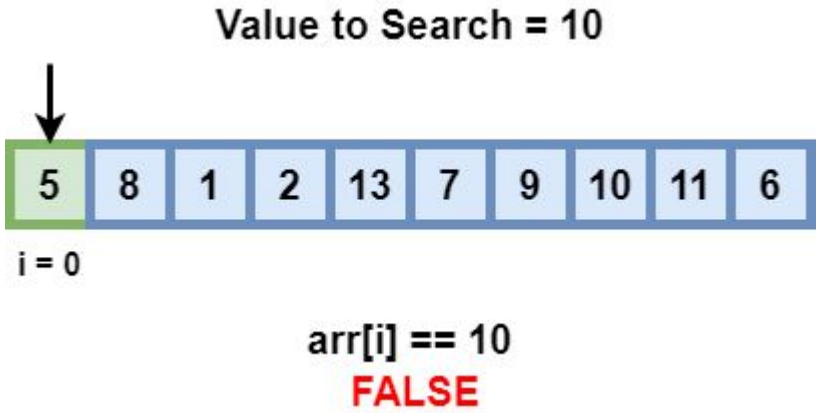
Monday	Wednesday	Friday
04/24/2023 (90) • Review: Unit 8 **• AP CS Question 4: 2D Array**	**04/26/2023 (90)** **• AP CS Multiple Choice Game**	**04/28/2023 (45)** • Review: Unit 7, Unit 10 • Algorithms: Iterative/recursive binary search, selection sort, insertion sort, merge sort
05/01/2023 **• FINAL**	**05/03/2023** **• AP EXAM**	

Searching and Sorting

(from Units 7 and 10)

Sequential Search (aka Linear Search)

```
for i ← 0 to length(array)-1
      if array[i] == targetValue
            return i
      end if
end for
return not found
```

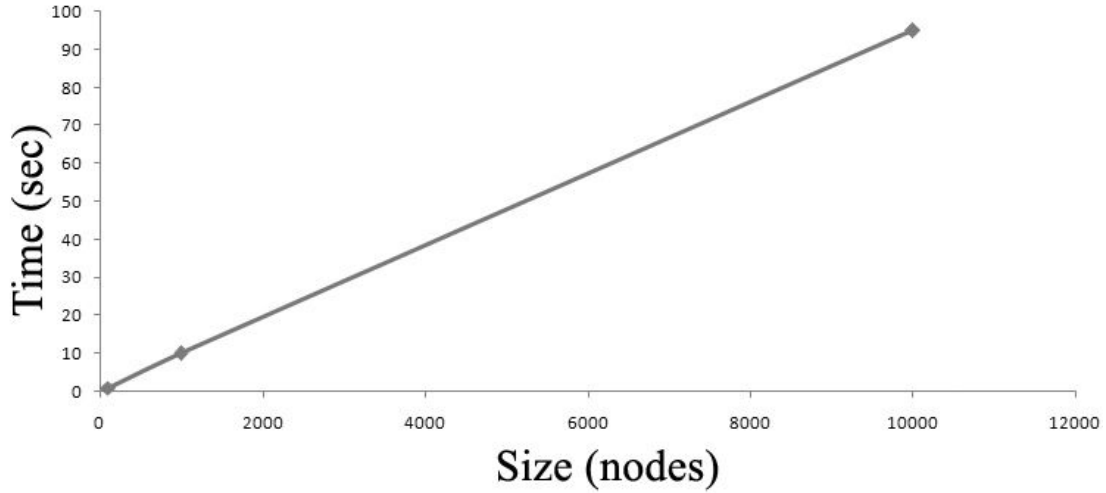


Pro: You can search for an item in any unsorted list and it is guaranteed to be found if it exists in the array

Con: If you're searching a long list, this can be a very slow approach

Sequential Search

If your list contains N elements, and the target element is at a random position in the list then on average it will take N/2 checks to find your element. When the time an algorithm takes to complete is directly related to the size of the array (N), we call this algorithm **linear** and the notation is O(N).

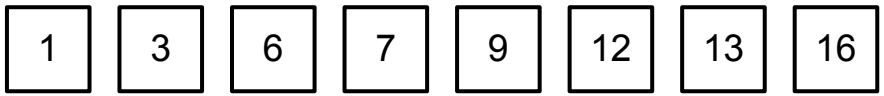


Binary Search: Divide and Conquer

Binary search is a much faster algorithm, but requires that the list be **sorted**.

We can use the fact that the list is sorted to reduce the problem to smaller problems.

An algorithm that breaks the problem into smaller subproblems is called a **divide-and-conquer algorithm**.



Loop Invariants

A **loop invariant** is a condition that must be true at the beginning and end of the body of a loop. (It might not be true while the loop body is doing its work, like swaps.)

```
// Precondition: values must be non-empty.
// Postcondition: The minimum value in "values" is returned.
public int minValue(int[] values) {
  int minResult = values[0];
  for (int i=1, n=values.length; i<n; i++) {
    // Loop invariant: minResult contains the minimum value in a[0]..a[i-1]
    if (values[i] < minResult) {
      minResult = values[i];
    }
  }
  return minResult;
}
```

Iterative Binary Search Algorithm

```
// Precondition: array must be in sorted order
low ← 0, high ← length(array)-1
while low <= high
        // Loop invariant: A[0...low-1] < targetValue < A[high+1...length(array)-1]
        middle ← (low + high) / 2
        if array[middle] == targetValue
                return middle
        else if array[middle] < targetValue
                low ← middle + 1
        else
                high ← middle - 1
        end if
end while
return not found
```

Recursive Binary Search Algorithm

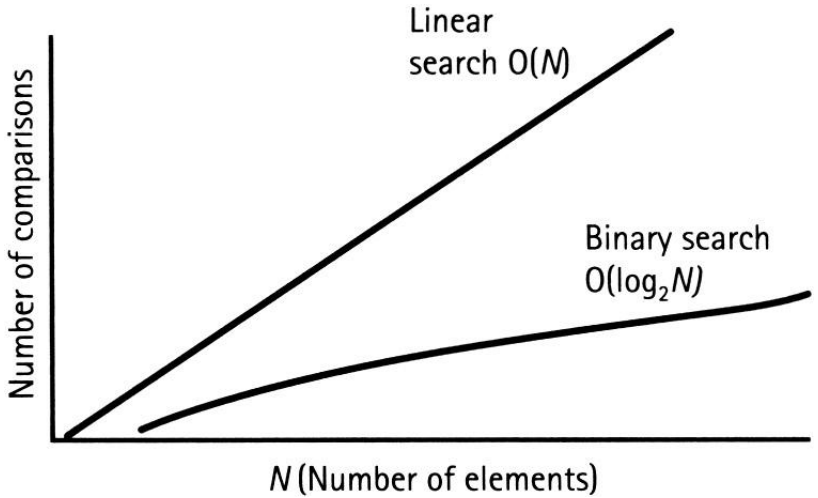
```
// Precondition: array must be in sorted order
def binarySearch(array):
        return binarySearchHelper(array, 0, length(array)-1)

def binarySearchHelper(array, low, high):
        if low > high, return not found
        middle ← (low + high) / 2
        if array[middle] == targetValue
                return middle
        else if array[middle] < targetValue
                return binarySearchHelper(array, middle + 1, high)
        else
                return binarySearchHelper(array, low, middle - 1)
        end if
end
```

Iterative or recursive binary search?

Iterative binary search is marginally faster and more memory-efficient, but the difference is ultimately insignificant.

Binary Search

If your list contains N elements, and the target element is at a random position in the list then on average it will take $\log_2 N$ checks to find your element. To give you a sense of this speedup, the $\log_2 1024 = 10$ (since $2^{10} = 1024$). **This means that binary search runs 10x faster on sorted lists of length ~1K.** The notation for this algorithm is O(log N)



Comparing Sequential Search vs Binary Search (These are "worst case" numbers.)

N	**Sequential Search Comparison Count**	**Binary Search Comparison Count**
2	2	2
4	4	3
8	8	4
16	16	5
100	100	7

Insertion Point

```
// Precondition: array must be in sorted order
low ← 0, high ← length(array)-1
while low <= high
        // Loop invariant:
        // A[0...low-1] < targetValue < A[high+1...length(array)-1]
        middle ← (low + high) / 2
        if array[middle] == targetValue
                return middle
        else if array[middle] < targetValue
                low ← middle + 1
        else
                high ← middle - 1
        end if
end while
return not found, insertion point is low
```

One thing we didn't cover the first time around on binary search is how it can be used to find the **insertion point**.

A loop invariant of binary search is that all elements at indexes `0...low-1` must **precede** the target value, that is

A[i] < targetValue for all i=0...low-1

If they didn't, we would still be looking for the target value in that range.

If we get to the return at the end, `low` represents is the **insertion point**... the place in the array where `targetValue` should be inserted to preserve the ordering.

Insertion Point

`mystery` is recursive binary search... but it returns `low` if the value is not found.

If `mystery` finds `num`, it returns its index. This index equals the count of elements that are less than `num`.
(Remember, no duplicates.)

If mystery doesn't find `num`, the loop invariant says everything to the left of `low` is less than `num`. So, `low` will still equal the count of elements less than `num`.

This question was really hard!
But it seemed to be something that you'd actually see on the AP exam.
It probably is from a real AP exam.

_____9) Consider the following instance variable and method.

```
private int[] arr;

/** Precondition: arr contains no duplicates
*          the elements in arr are in ascending order
* @param low is an int value 0 <= low <= arr.length
* @param high is an int value low - 1 <= high < arr.length
* @param num an int value
*/
public int mystery(int low, int high, int num) {
     int mid = (low + high) / 2;
     if (low > high) {
          return low;
     } else if(arr[mid] < num) {
          return mystery(mid + 1, high, num);
     } else if(arr[mid] > num) {
          return mystery(low, mid - 1, num);
     } else {
          return mid;
     }
}
```

What is returned by the call `mystery(0, arr.length - 1, num)` ?

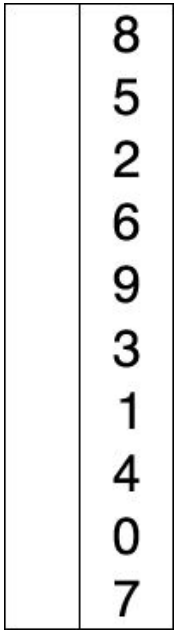
(A) The number of elements in `arr` that are less than `num`
(B) The number of elements in `arr` that are less than or equal to `num`
(C) The number of elements in `arr` that are equal to `num`
(D) The number of elements in `arr` that are greater than `num`
(E) The index of the middle element in `arr`

Sorting Algorithms we covered

Selection Sort	**Insertion Sort**	**Merge Sort**
• Slow, and consistently slow • Always performs $N^2/2$ comparisons and 0...N swaps • Best case, average case and worst case all the same	• Slow for large numbers of elements • One of fastest algorithms for small numbers of elements • Does well on partially sorted data. • **Best case:** Already sorted array. N comparisons, 0 swaps • **Average case:** About half the comparisons of Selection Sort • **Worst case:** Entirely reversed array. N^2-N comparisons and about same number of swaps	• Fast, efficient algorithm that scales up to large numbers of elements • Due to complexity, can be slower than Insertion Sort on very small numbers of elements • Best case, average case and worst case are O(N log N) comparisons • Can be combined with algorithms like Insertion Sort (used for small sublists)

Selection Sort Algorithm

```
for i ← 0 to length(array)-1
        // Loop invariant: The array to the left of i contains
        // the i smallest values in the array, in sorted order.
        jMin ← i
        for j ← i+1 to length(array)-1
                // Loop invariant: A[jMin] is smallest value in range A[i]...A[j-1]
                if A[j] < A[jMin] then jMin ← j
        end for
        if jMin != i then swap A[i], A[jMin]
end for
```



Let's run through it on the whiteboard...

Selection Sort – an intuitive but slow algorithm

$O(N^2)$ is also known as **quadratic time**. Many simple sort algorithms are quadratic time.

It can be OK for a small number of elements, but as N gets big, the algorithm's running time becomes very long.

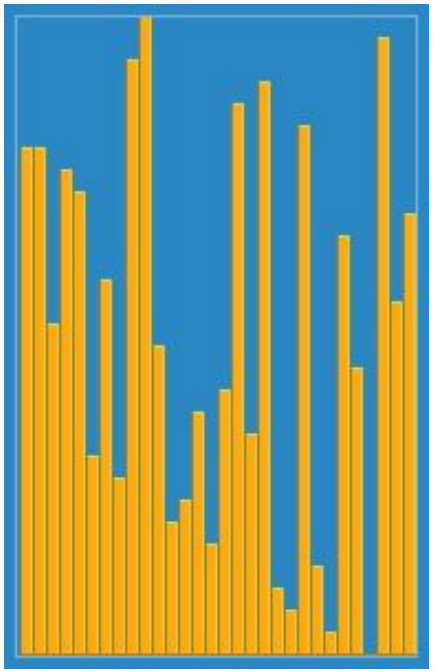
In Unit 10, we will learn a sorting algorithm, **merge sort**, that has much better running time: O(N log N). But it's also more complicated to code!

N	N^2
1	1
10	100
100	10,000
1000	1,000,000
10000	100,000,000
100000	10,000,000,000
1000000	10^{12}

Insertion Sort

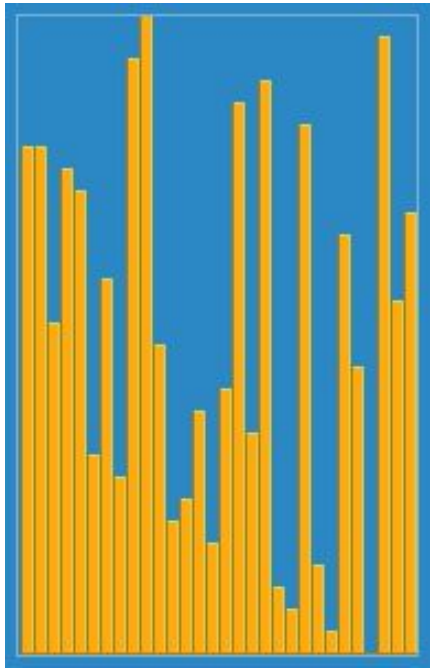
Insertion Sort is more complex than Selection Sort, but is much faster when the data is partially sorted.

- Insertion Sort is still a quadratic algorithm, that is, $O(N^2)$.
- Insertion Sort is faster in practice than other quadratic algorithms such as Selection Sort.
- Insertion Sort is actually one of the fastest known algorithms for sorting **very small** arrays. (Around <=10 items.)



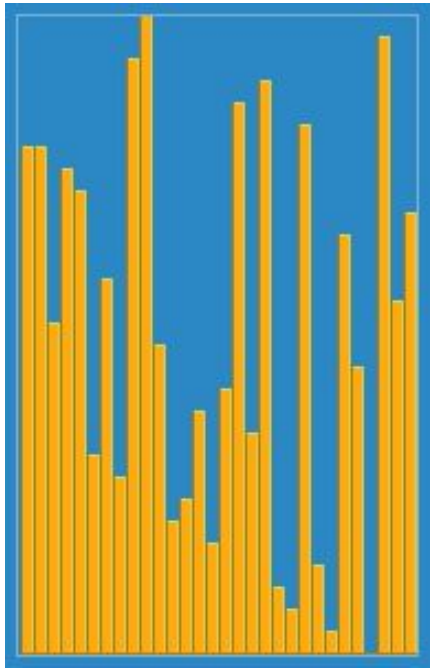
Insertion Sort Algorithm

- Insertion Sort has a loop invariant that for index i, **the entire sub-array to the left of i is in sorted order.**
  - This is slightly different from Selection Sort's loop invariant... how?
- Insertion Sort's outer loop starts with i = 1, that is, pointing to the **second** element in the array.
- Why? The sub-array to the left of i = 1, a[0..0], is in sorted order, because a one-element array is always in sorted order!



Insertion Sort Algorithm: Swapless Edition

```
// A bit more efficient than swapping. Less read/write operations.
// Note we start with second element i = 1
for i ← 1 to length(array)-1
        // Loop invariant: The array to the left of i is a sorted sub-array.

        // Save the value at A[i], as it may be overwritten
        x ← A[i]

        // j will represent the insertion point of the value.
        j ← i
        while j > 0 and A[j-1] > x
                // As we scan for insertion point, we move elements
                // to the right.
                A[j] ← A[j-1]
                j ← j - 1
        end while

        // Finally, write the element being inserted into its final spot.
        A[j] ← x
end for
```



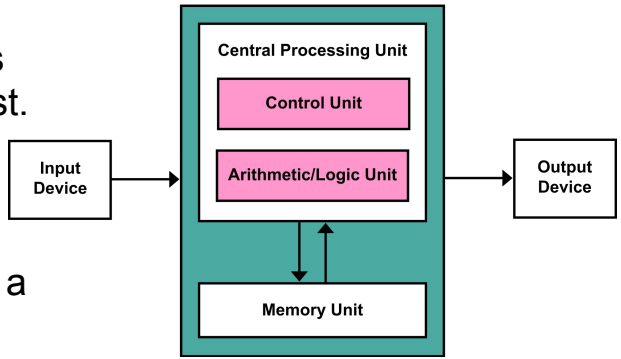
Merge Sort

Merge Sort is a **divide and conquer** sorting algorithm, much like Binary Search is a divide and conquer search algorithm.

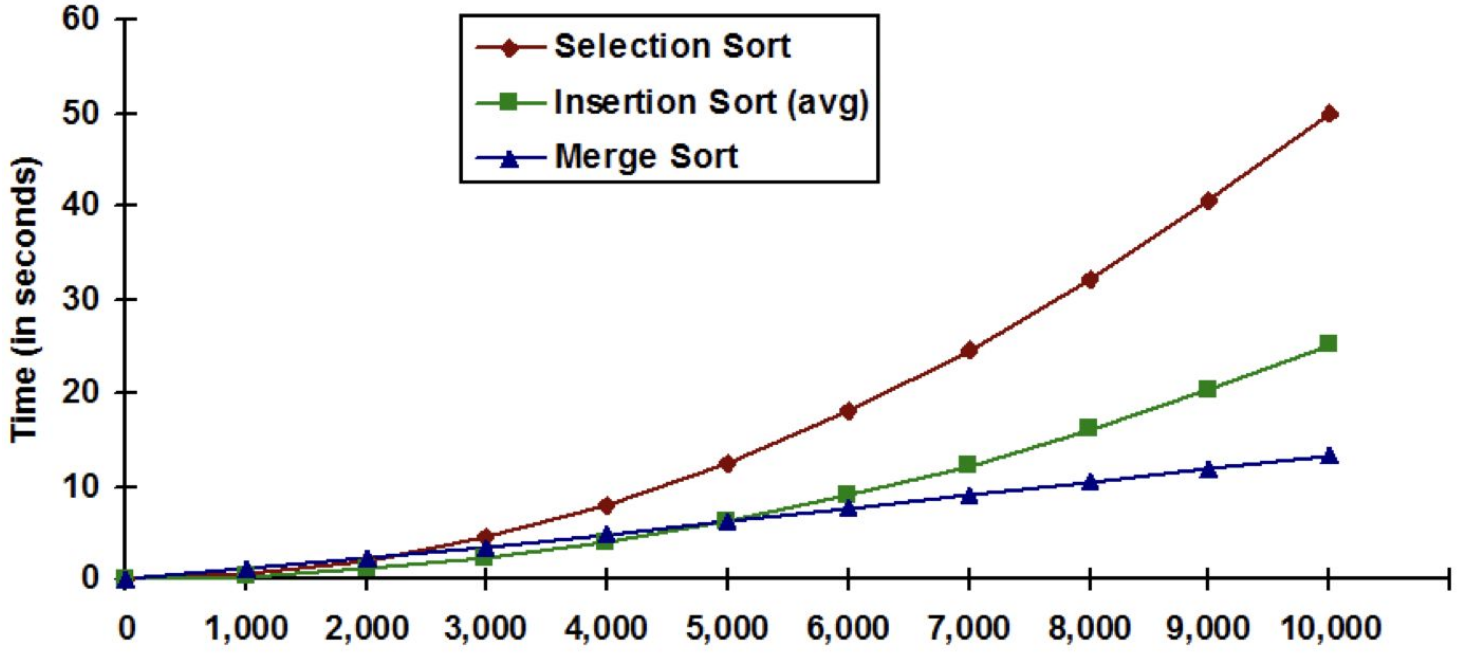
Merge Sort divides the input array in half, and recursively Merge Sorts the halves.

This algorithm was invented in 1945 (!) by John Von Neumann (1903-1957), a genius mathematician and early computer scientist.

Among other things, Von Neumann invented the Von Neumann Architecture, which is still the conceptual model for how a computer works.

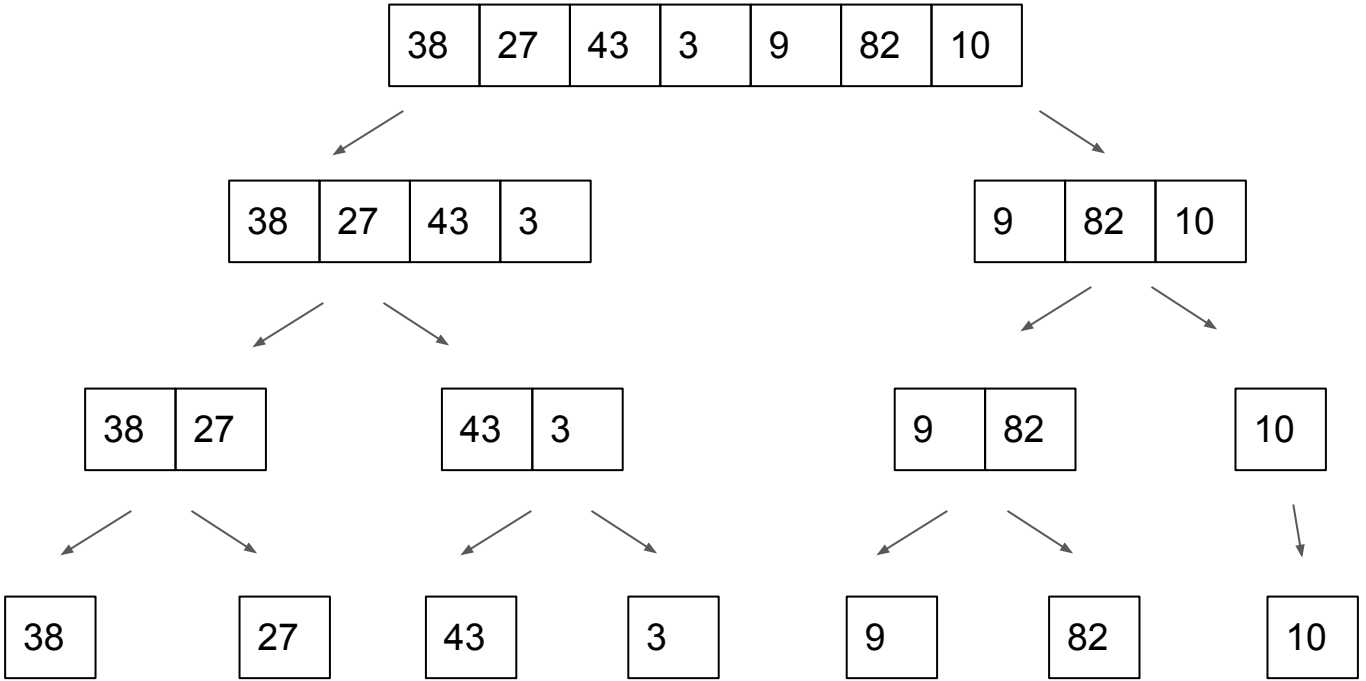


Sorting Algorithm Efficiency

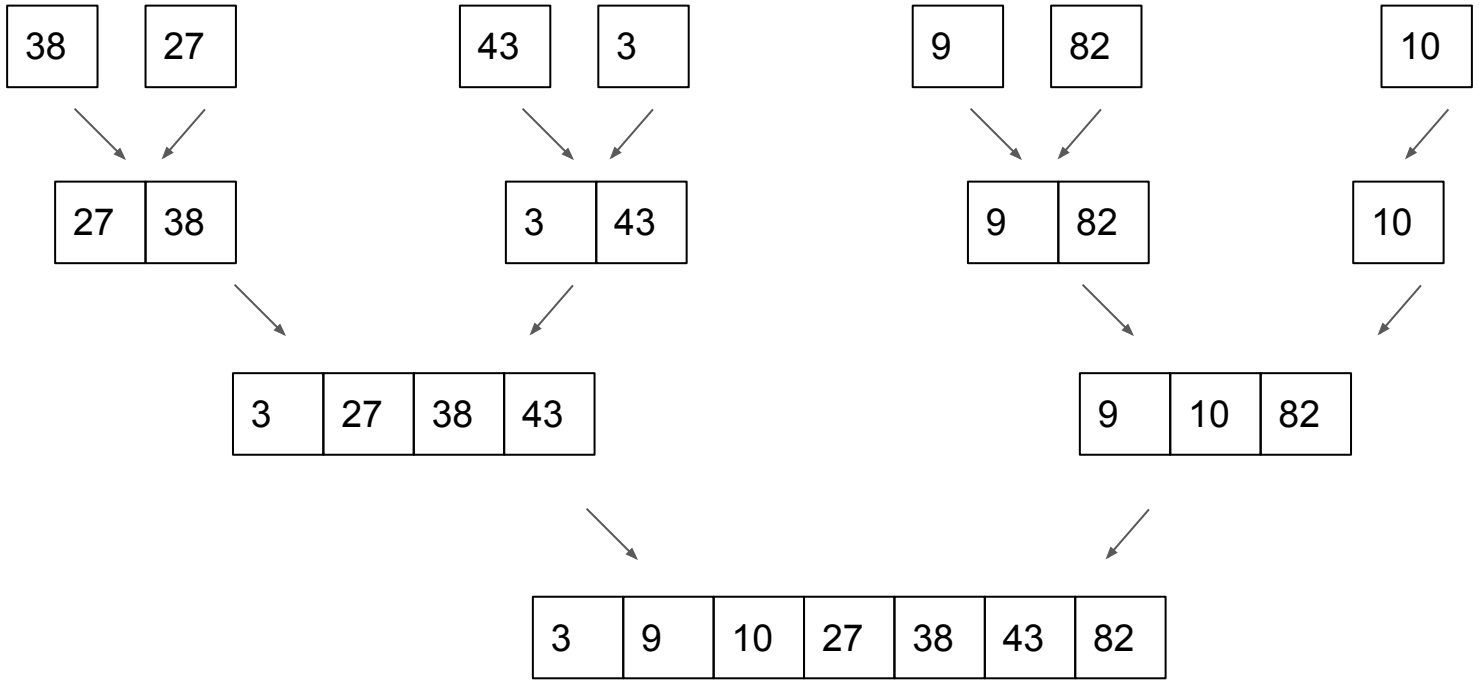


Merge sort worst case performance is O(n log n)

Merge Sort: Split the array into halves down to one element



Merge Sort: Sort the halves and merge them together



Merge Sort Pseudocode

1. If the list's size is 0 or 1, just return the original list (as it is sorted)
2. Split the list parameter int two lists of roughly equal size
3. Recursively Merge Sort both split lists, list 1 and list 2
4. Merge the two sorted lists and return the result

Merge Sort "Lists"

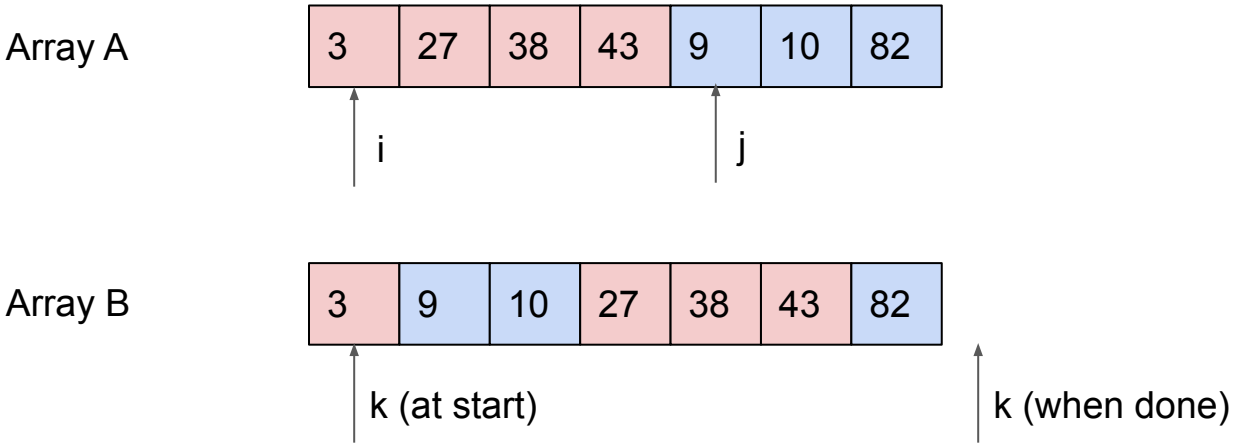
Conceptually, you can think of Merge Sort as splitting the input into a bunch of smaller sub-lists, and recursively merging all the sub-lists together.

In practice, we don't want to create tons of lists! We want to allocate as little memory as possible, and copy as little data as possible. So we want to work as much as possible in the array being sorted.

This means that a "sub-list" is really a "view" into the original array, tracked with two indices, begin and end.

Because of Merge Sort's merge operation, we do need some temporary storage, but that comes out to one additional temporary array that is the same size as the array being sorted.

Merge Sort: The Merge



i = current index in left half, j = current index in right half. k = index in output array.

While i <= ending index of left half and j <= ending index of right half:
  B[k++] = smaller of A[i] and A[j].
  Advance i if we used A[i], or advance j if we used A[j].

After, one of the halves may still have something left... output anything left over into B.

CSAwesome's Merge Sort (edited for brevity) [1/2]

```
public static void mergeSort(int[] elements) {
   int n = elements.length;
   int[] temp = new int[n];
   mergeSortHelper(elements, 0, n - 1, temp);
}

private static void mergeSortHelper(int[] elements,
                                    int from, int to, int[] temp) {
    if (from < to) {
       int middle = (from + to) / 2;
       mergeSortHelper(elements, from, middle, temp);
       mergeSortHelper(elements, middle + 1, to, temp);
       merge(elements, from, middle, to, temp);
    }
}
```

CSAwesome's Merge Sort implementation [2/2]

```
private static void merge(int[] elements, int from, int mid, int to, int[] temp) {
   int i = from, j = mid + 1, k = from;

   while (i <= mid && j <= to) {
      if (elements[i] < elements[j]) {
         temp[k++] = elements[i++];
      } else {
         temp[k++] = elements[j++];
      }
   }

   while (i <= mid) {
      temp[k++] = elements[i++];
   }

   while (j <= to) {
      temp[k++] = elements[j++];
   }

   for (k = from; k <= to; k++) {
      elements[k] = temp[k];
   }
}
```

Here, we're comparing the current elements from each half, and taking the smaller one. We only advance the pointer for the side we took from!

If we got here, one or both halves are done. Output anything left over from the left half.

Output anything left over from the right half. (Only one of these loops will run.)

Copy everything back from the temp array. (This can be avoided with some extra fanciness.)

Final slide before the Final and AP Exam

GOOD LUCK!